FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Optimization Challenges of the Future Federated Internet

## Heuristic and Exact Approaches

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

**Doktor der technischen Wissenschaften**

by

**Dipl.-Ing. Johannes Inführ**
Registration Number 0625654

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: a.o.Univ.-Prof. Dipl.-Ing. Dr.techn. Günther R. Raidl

The dissertation has been reviewed by:

(a.o.Univ.-Prof. Dipl.-Ing. Dr.techn. Günther R. Raidl)

(Prof. Dr. Kurt Tutschku)

Wien, 10.10.2013

(Dipl.-Ing. Johannes Inführ)

# Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Johannes Inführ
Kaposigasse 60, 1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

# Acknowledgements

This work as it lies before you would not be what it is without the numerous and significant contributions by my co-workers and family. I want to express my gratitude and ensure them of my perpetual thankfulness. The order of contributors that follows is not based on importance or quality of contribution, but rather on a call to `std::shuffle`. I thank the following people:

My colleagues within the Algorithms and Data Structures Group, currently Günther Raidl, Doris Dicklberger, Andreas Müller, Christopher Bacher, Benjamin Biesinger, Frederico Dusberger, Igor Grujicic, Bin Hu, Christian Kloimüllner, Petrina Papazek, Mario Ruthmair, and Christian Schauer, for being great guys and gals all around, supportive, fun, and helpful in any way they can.

Markus Leitner, the expert on the theory of Integer Linear Programming, for guiding me through the morass that is Column Generation.

Günther Raidl, for being an excellent supervisor, offering me the opportunity for working on the OptFI project, and being a great beta-reader of this thesis, always offering valuable feed-back.

Bin Hu, for initial discussions about OptFI and for having the right proof at the right time.

Kurt Tutschku, for initiating the OptFI project, numerous discussions and guidance on a topic that was completely new to me, and valuable comments on this thesis.

David Stezenbach, for many discussions on the concrete problems to be tackled within the OptFI project and a different perspective.

My family, for offering the nurturing environment and save haven that made this work possible.

Doris Dicklberger, for helping with any- and everything and for being a great insulation against the bureaucracy of the university.

Andi Müller, for remaining calm and supportive, even if some unnamed third party *cough* floods our shared home directory with log-files, causes literally thousands of automated emails due to configuration errors, and always requests the most bleeding-edge software to be present on our cluster.

Susan, for being supportive, patient, and understanding, even when the weekends became work-days and for being a beta-reader of remarkable stamina and endurance, only rarely requiring some poking, prodding or alternatively chocolate bananas.

Christian Schauer, the expert on heuristics, for cheerfully picking up my slack with respect to my teaching obligations when the thesis deadline loomed and for being a valuable beta-reader.

Mario Ruthmair, the expert on practical programming issues especially with Integer Linear Programming, for offering the project culminating in the tool that made the evaluations in this work possible, for cementing my interest for doing my studies at the ADS and for being a very thorough beta-reader.

The Vienna Science and Technology Fund (WWTF) for financing the OptFI project and helpfully supplying one line to every work connected with it (like this thesis): “This work has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT10-027”.

# Abstract

The Internet has ossified. It has lost its capability to adapt as requirements change. A fitting practical example for ossification is the introduction of IPv6. It has been specified in 1998 to solve, among other things, the forseeable Internet address shortage. The addresses have begun to run out in 2011 and still IPv6 does not see any wide-spread usage; hacks like network address translation reduce the need to switch.

A promising approach for solving this problem is the introduction of network virtualization. Instead of directly using the single physical network, unchangeable to a large degree and working just well enough for a limited range of applications, multiple virtual networks are embedded on demand into the physical network, each of them perfectly adapted to a specific application class. Compute capabilities within the network are provided to the virtual networks, enabling them to offer their own customized topologies, routing, resource management and naming services.

There are still numerous unsolved problems regarding network virtualization, ranging from the implementation of virtualizable routers to economic aspects. In this thesis, we focus on the problem of resource allocation. All virtual networks, with all the resources they require (e.g., bandwidth), still need to fit into the available physical network. Our aim is not merely finding an arbitrary solution, we want to fit the virtual networks in a cost-optimal way. This is the core of the Virtual Network Mapping Problem (VNMP), an $\mathcal{NP}$-complete Combinatorial Optimization Problem.

We present several heuristic and exact approaches for solving the VNMP. As heuristic methods we investigate Construction Heuristics, Local Search, Variable Neighborhood Descent, Memetic Algorithms, Greedy Randomized Adaptive Search Procedures, and Variable Neighborhood Search. The exact approaches we develop are based on Constraint Programming and Integer Linear Programming. In addition to analyzing different solution methods and comparing their various strengths and weaknesses, we present a strong preprocessing method for VNMP instances. This preprocessing method can determine which parts of the physical network each virtual network can and cannot use. We show that preprocessing is essential for solving large VNMP instances with exact methods.

For finding a valid mapping of virtual networks into substrate networks, the preprocessing method is powerful enough to make Integer Linear Programming the solution method of choice. For low-cost solutions, the situation is more complex. Integer Linear Programming is the best method for small to medium instance sizes. If run-time is a concern, our Memetic Algorithm and Variable Neighborhood Search approaches can be used to great effect, achieving costs within 5% of the exact method. For large instances, we conclude that Variable Neighborhood Descent performs best.

# Kurzfassung

Das Internet wie wir es heute kennen hat seine Fähigkeit verloren, sich an ändernde Bedingungen anzupassen. Es gilt als “erstarrt”. Ein prägnantes Beispiel ist die Einführung von IPv6. Dieses Protokoll wurde schon 1998 spezifiziert, um unter anderem der bevorstehenden Internet-Adressknappheit entgegenzuwirken. Obwohl die Adressen seit 2011 zur Neige gehen, wird IPv6 noch immer nicht großflächig eingesetzt. Notlösungen wie Netzwerk-Adressübersetzung reduzieren die Notwendigkeit eines Wechsels.
Ein vielversprechender Ansatz um wieder Flexibilität in das Internet zu bringen ist Netzwerkvirtualisierung. Statt eines einzigen unflexiblen physischen Netzwerks, das eine Reihe von Anwendungen gerade noch ausreichend unterstützt, werden mehrere virtuelle Netzwerke, die voll und ganz auf verschiedene Anwendungsfälle ausgerichtet sind, in das physische Netz eingebettet.
Bevor Netzwerkvirtualisierung großflächig eingesetzt werden kann, gilt es noch eine Vielzahl von Problemen zu lösen, von der Implementierung von virtualisierbaren Routern bis hin zu wirtschaftlichen Aspekten. In dieser Dissertation konzentrieren wir uns auf Ressourcenverteilung und -belegung. Die verschiedenen virtuellen Netze, samt ihren benötigten Ressourcen (z.B. Bandbreite), müssen ein einem einzigen physischen Netz untergebracht werden. Unser Ziel ist jedoch nicht, eine beliebige Einbettung der virtuellen Netze in das physische Netz zu finden, sondern eine kosten-optimale. Das ist der Kern des Virtual Network Mapping Problems (VNMP), ein $\mathcal{NP}$-vollständiges kombinatorisches Optimierungsproblem.
In dieser Arbeit untersuchen wir heuristische und exakte Ansätze zur Lösung des VNMP. Die heuristischen Methoden sind Konstruktionsheuristiken, Lokale Suche, Variable Neighborhood Descent, Memetische Algorithmen, Greedy Randomized Adaptive Search Procedures und Variable Neighborhood Search. Als exakte Verfahren entwickeln wir Ansätze, die auf Constraint Programming und Integer Linear Programming basieren. Zusätzlich zur Analyse der vorgestellten Algorithmen und des Vergleichs ihrer Stärken und Schwächen präsentieren wir auch eine Vorverarbeitungsmethode für VNMP Instanzen. Wir zeigen, dass diese Vorverarbeitung ein essenzieller Schritt für die Anwendung von exakten Verfahren auf große VNMP Instanzen ist.
Nur durch die Vorverarbeitung ist es möglich, dass unser Integer Linear Programming Ansatz unabhängig von der Instanzgröße ein exzellentes Verfahren ist, wenn es um das Finden einer beliebigen Einbettung geht. Für die Suche einer kosten-optimalen Lösung ist die Wahl der besten Methode komplizierter. Integer Linear Programming liefert die besten Ergebnisse bis zu mittleren Instanzgrößen, jedoch nur unter hohem Zeitaufwand. Gilt es diesen zu minimieren, sind unsere Memetischen Algorithmen und Variable Neighborhood Search Ansätze vielversprechend. Die damit erreichten Kosten liegen nur 5% höher als die der exakten Methode. Für große Instanzen bietet Variable Neighborhood Descent die beste Lösungsqualität.

# Contents

CHAPTER 1

# Introduction

The Internet has ossified [72, 132]. This means it has lost its ability to react to changing requirements, its ability to innovate. It has fallen victim to its own success [170], and only just works [72].
Why this negativity? Without a doubt, the Internet works. Customers can expect ever increasing data transfer speeds. With higher speeds new and innovative services become possible, like video streaming. The video streaming service YouTube [181], founded in 2005, is now visited by more than a billion users and streams six billion hours of video in one single month. Other services like Twitch [99] or Ustream [173] are not satisfied with video on demand and offer true live video streams. A camera and Internet access is all that is required to instantly reach an audience of thousands. Since the advent of mobile phones with Internet connectivity, it is no longer a challenge to be online anytime, anywhere.
So, where are the big problems? If the Internet has ossified, how did it come to that? Where are the cracks that show that something is amiss? How can these problems be solved and what contribution is this thesis going to make? We will answer these questions in the following sections.

## 1.1 The History of a Changing Internet

The history of the Internet is one of explosive growth and change. Technologies and protocols have been discarded or modified as it became clear that they could not keep up. An excellent review can be found in [72], which forms the basis of this section.
The Internet had its beginnings as the ARPAnet, the first large-scale packet switched network. Its foundation was the Network Control Program (NCP) [35], responsible for addressing and data transport. As the ARPAnet grew, it became clear that NCP was not flexible enough. The task of addressing and data transport fell to the Internet Protocol (IP), but ensuring reliability, i.e., that sent data is actually received, was from now on achieved by the Transmission Control Protocol (TCP) on top of IP. An alternative that does not guarantee reliability, the User Datagram

Protocol (UDP) was also introduced. The switch from NCP to TCP/IP occurred on a single day in 1983, when the remaining ARPAnet nodes started using the new protocols. This procedure encompassed about 400 nodes and was probably the last time core functionality could be replaced by just moving every component to the new technology. From then on, every change has been deployed incrementally.

As the Internet grew, components failed to scale and were replaced. An early example for this is the "hosts.txt" file [111] used for name resolution in addressing. It rapidly grew infeasible to distribute a file with the names and addresses of all available servers in the Internet to each computer with access to the Internet. In 1982, this system was replaced by the Domain Name System (DNS) which in addition to solving the distribution problem also introduced namespaces which could be administrated in a decentralized fashion. Handley [72] notes that this system could have been developed years earlier but only as the scaling limit of the previous system was reached the need was pressing enough to deploy a replacement.

Other systems replacing their predecessors because they could not handle the increasing size of the Internet are link-state routing protocols [124] and the Exterior Gateway Protocol [151].

The strategy of fixing problems just in time worked rather well, until a series of congestion collapses occurred in the mid-1980s. The network was moving data at full capacity, but no useful work was done. The problem turned out to be TCP's retransmission policy, the network was flooded with data that was unnecessarily retransmitted. This problem occurred with TCP, but also UDP can cause congestion. Indeed, congestion is a consequence of trying to send more data than possible, independent of the employed protocols. Therefore, the correct solution might have been to add a layer to the protocol stack that handles congestion. This would have been a serious change to the core working principles of the Internet and in 1988 it was already too large to attempt it. Instead, a congestion control mechanism for TCP was introduced [96], not a solution for the general problem, but good enough. This change was backwards compatible, incrementally deployable and probably the first of what are called "architectural barnacles" by Peterson et al. [6], unsightly outcroppings that have affixed themselves to an unmoving architecture.

After the Border Gateway Protocol (BGP) [116] was introduced in the early 1990s to allow the commercialization of the Internet, the last major change to the core Internet was the introduction of Classless Inter-Domain Routing (CIDR) [182] in 1993, changing how addressing worked. It was basically lucky chance that made this switch possible. Firstly, it was backwards compatible and the end-hosts could use the previous system until an operating system upgrade fixed the issue eventually. Secondly, the routing hardware within the network was supplied by a single vendor, and the affected functionality was implemented in software and thus easily changed. Such a fix would be unthinkable today, as the core functionality of the Internet protocols is implemented in hardware for speed reasons.

At the time of writing, 1993 was two decades ago. What happened in the meantime? Except the explosive growth of the Internet, nothing. Numerous improvements to the Internet have been suggested, all bringing an immediate benefit. Examples include Explicit Congestion Notification [144], Integrated Services [18], Differentiated Services [26], and Mobile IP [137]. All of them failed. They might be available in small, isolated parts of the Internet [171], but never saw general adoption.

The history of the Internet shows that changes to the network only occur if there is an immediate monetary gain, or if the network is about to collapse. Achieving monetary gain by improving the core Internet is hard, because the Internet Service Providers (ISPs) need to agree on the changes. If all ISPs offer the improvement, then there is no benefit for any of them [6]. Improvements become impossible, the Internet ossifies.
In [132], the general diagnosis of ossification is further refined. First, there is intellectual ossification. Any new technology has to be compatible with the current technology from the outset. This stifles innovation. Secondly, there is infrastructure ossification. Suggested improvements are not deployed in the infrastructure, not even for testing purposes. Thirdly, there is system ossification, describing basically the same effect as the architectural barnacles [6]. Instead of fixing problems at their root, workarounds and fixes are employed to keep the system running while making it more fragile and susceptible to even more problems.

## 1.2 Current Problems

The Internet, the general purpose network flexible enough to evolve and meet new challenges head on, has been lost. Instead, we have the Internet, the global network working really well, as long as nothing too extreme, like Quality-of-Service (QoS) guaranty, is asked of it. How does this ossification show in practice, which problems do occur? Handley [72] offers a list of short, medium, and long term problems due to ossification.
Spam, security and Denial-of-Service (DoS) attacks are short term and immediate problems. One of the main reasons why these problems exist is that at its core, the Internet is a transport network. It efficiently transfers data from A to B. Whether B wants the data is irrelevant. As workarounds, firewalls and Network Address Translators (NATs) are used and consequently, the Internet loses transparency. Data is dropped or modified along the way for no apparent reason and the deployment of applications that need to transfer data is much more complicated than it needs to be. As a result, many applications dress up their data as HyperText Transfer Protocol (HTTP) traffic, since this is understood and accepted by most firewalls. Problems only arise if HTTP does not correspond to the communication requirements of the application.
A fitting example is the Voice-over-IP (VoIP) software Skype [162]. It uses UDP to transfer voice data since short delays are important and reliability is not an issue as long as enough packets arrive. The data has to be sent directly from one user to another for latency reasons, which is hard to do when NATs are involved. NATs change IP addresses and UDP ports and the precise mapping has to be determined by the Skype client. Techniques to do so involve contacting a remote server to determine the IP address and using heuristics to determine the port mapping. These techniques are complicated and error prone. It gets even worse if both Skype clients that want to communicate are behind NATs. Then a third party, not hidden by a NAT, has to be used as a relay station, which adds delay and reduces reliability, since the third party (another Skype client) can quit at any time. This approach fails completely if most clients are behind NATs, if the heuristics cannot determine the mappings, or if a firewall blocks UDP traffic. In those cases, Skype falls back to using TCP as a last resort. In the end, Skype works well enough to be successful, but consider the amount of engineering that was necessary. It should not have been required.

The medium term problems identified by Handley [72] are congestion control, inter-domain routing, mobility, multi-homing and architectural ossification. Congestion control is problematic, because with rising link speeds it takes longer and longer for the data transfer rate to converge to a suitable value. As for inter-domain routing, this is facilitated by BGP [116]. It is basically the glue that holds the Internet together and when it fails, connectivity is affected. In [122], the frequency of BGP misconfigurations is measured, and they have been found to be prevalent. For every misconfiguration, there is a 4% chance that it affects the connectivity, i.e., some parts of the Internet cannot be reached. Efforts to improve and secure BGP have failed. The mobility of users, or the possibility that they have multiple simultaneous connections to the Internet (multi-homing) remain open problems. An example of architectural ossification is that not even the extension paths already built into the protocols can be used. IP for instance was designed to be extensible by using IP options. However, packets without options can be handled in a hardware-accelerated manner within routers. With options, packets have to be processed in software. Using IP options would be equal to an DoS attack on the router, so those packets are highly likely to be dropped.
Address space depletion is a long term problem that will be very hard to deal with due to ossification. Handley [72] states that it was already clear in 1990 that the Internet addresses would run out. In the meantime, CIDR and NATs kept the network running. An alternative without the problem of address depletion, IPv6 [42], was specified in 1998. To this day, adoption is slow [12, 170], even though the organizations responsible for distributing the addresses have begun to run out of addresses in 2011 [84]. This is probably the best example for ossification in practice, but what are the reasons for this slow adoption? One component is, that NATs alleviate the address shortage, another, that IPv6 is complicated to implement. Maybe the problem is that IPv6 is just more of the same and does not enable the required fundamental change.
Ossification would not be a problem if the requirements were static. But they are not. According to [132], users do not only want more bandwidth. They want more reliability, predictability, manageability, configurability, security, and privacy. Improving those characteristics requires changes. The Internet was meant to be a general-purpose network, but now more than that is needed [72].

## 1.3 Network Virtualization: The Cure?

The Internet has stopped evolving, but how can we start the evolution up again? We have already seen that a new technology, that wants to have any hope of actually being deployed, has to have two properties. It needs to be backwards compatible and incrementally deployable. Network virtualization has been put forward as a suitable candidate [6, 12, 66, 168, 171, 172].
The basic idea of network virtualization is straight forward. Instead of using one physical network that can do everything well, use multiple virtual networks embedded in the physical network, each one specialized and perfectly adapted for a particular application. To allow for this adaptation, the nodes of the virtual networks receive compute capabilities within the routers of the physical network. Therefore, virtual networks can offer their own (and application specific) topology, routing, naming services, and resource management [171]. The alternative to adapted virtual networks, multiple physical networks, is clearly infeasible on a global scale.

That sounds promising, but is it realistic? At the very least, network virtualization requires support from the physical network by offering routers that can host virtual machines (the routers of the virtual networks). This technology is already available [32]. In addition, network virtualization is actively and successfully used in large scale scientific network testbeds such as GENI [62], PlanetLab [31] or G-Lab [156], not as an enhancement to be studied, but as a central enabling technology for carrying out experiments. Virtual networks are used to partition the network testbeds so that different research groups can perform their experiments without interference from each other. As Tutschku et al. [172] state, the "virtualization of telecommunication services or applications is no longer an academic concept".
Network virtualization is not the only proposed concept to break the ossification of the Internet. Alternatives include OpenFlow [123] and Software Defined Networks [112], which allow very flexible routing that can be administrated centrally.

### 1.3.1 Advantages and Applications

Allowing network virtualization in the Internet offers some advantages and applications, which are currently unthinkable, become possible. As a central advantage, Berl et al. [12] mention the flexibility of the system. Virtual networks can be dynamically reconfigured, new networks can be added and old ones can be removed, suspended or discarded. The state of the network can be frozen and reverted if the need should arise. Unused parts of the network can be shut down to conserve energy. In aggregated services, it is possible to fix or replace parts transparently [27].
Some interesting possibilities also emerge on the user side of things. Turner et al. [170] describe a virtual network offering a learning environment with high quality audio and video multicast mechanisms. Format translators are available at the virtual nodes to enhance compatibility. With virtual networks, it may be possible to switch Internet Service Providers (ISPs) on the spot, like it is possible for electricity or phone providers [171].

### 1.3.2 Challenges

Adoption is a central problem for every new technology. How will (or should) the adoption process of network virtualization work? Anderson et al. [6] envision the following adoption process. Virtual networking will start with a single daring Next Generation Service Provider (NGSP) offering virtualization services for its own network. Customers not directly connected to the network of the NGSP can connect via standard ways through the current Internet. If the NGSP is successful, it can expand its network to reach more customers. Local ISPs may be forced to offer the same services to stay competitive. For Turner et al. [170], adoption will resemble more the introduction of the Internet. First, virtual networks will be offered as an overlay in the existing Internet. Then, a government-supported experimental backbone infrastructure will be built, which natively supports virtualization. As the last step follows the commercial operation of virtual networks.
After adoption is achieved, the tasks of the current ISPs will be carried out by two different business entities [29], the infrastructure providers (InPs) and the virtual network providers (VNPs). InPs will manage the physical infrastructure necessary for hosting the virtual net-

works. VNPs create virtual networks from a federation of the resources offered by the different InPs [28, 50, 170].
This structure offers a rich environment for business opportunities [170]. InPs can compete by offering better services for VNPs using their networks, such as high quality resources, management tools, operation support, and fault tolerance. VNPs can distinguish themselves by offering shorter virtual network setup times, higher quality virtual networks, or guaranteed resources. In addition, VNPs do not own any physical resources, they do not need to deploy and maintain infrastructure. Thus, there is a low barrier to entry for VNPs. There are also opportunities for network equipment vendors, as there will be demand for high performance virtualizable routers.
Network virtualization is not ready to be deployed in the Internet. The authors of [29, 170] identify key research questions that still need to be answered, which we summarize below.

**Interfacing** How can InPs and VNPs communicate, for instance about available resources or requirements?

**Signaling and Bootstrapping** How can VNPs set up their virtual resources, if they have no communication capabilities besides the resources that they want to set up?

**Resource Allocation** How can a VNP best fit its virtual networks into the resources it has leased from the InPs?

**Resource Discovery** How can InPs keep track of the resources they offer and their connectivity, especially to other InPs.

**Admission Control** How can it be ensured that the capacities of an InP cannot be exceeded?

**Virtualization** How are the physical routers to be designed to allow low overhead virtualization?

**Resource Scheduling** How can an InP efficiently distribute its resources among the interested VNPs. How long should an InP guarantee the availability of resources?

**Naming and Addressing** How can the situation be handled that a single host may connect to multiple different virtual networks, each with different naming and addressing schemes?

**Dynamism and Mobility** How can the dynamic nature of virtual networks and the changing location of users be efficiently handled in terms of routing?

**Operation and Management** How can virtual networks be efficiently monitored and managed?

**Security and Privacy** How can it be prevented that a hostile virtual network breaks out of its virtualized environment and takes control of the physical infrastructure?

**Heterogeneity of Technology** How can the plurality of different virtualization technologies be handled efficiently?

**Economics** How should the economics of virtual networks work?

**Service Duplication** How can the overhead caused by multiple virtual networks offering the same basic services be avoided?

In this thesis, we cannot solve all those problems. In the following section, we will outline our aim.

## 1.4 Scope and Structure of this Thesis

There are a lot of unsolved problems surrounding network virtualization. In this thesis, we will focus exclusively on resource allocation. We will try to answer the question of how VNPs may best utilize the resources they have acquired from the InPs so that all virtual networks they want to create actually fit and do not incur excessive operational costs. Simply put, we will study algorithms for mapping multiple virtual networks into a physical (substrate) network in a reasonable way. We call this problem the Virtual Network Mapping Problem (VNMP). The remainder of this thesis is structured as follows:
In Chapter 2, we give an overview of the relevant theory and define our experimental methodology, followed by a rigorous definition of the VNMP and its variants in Chapter 3. Related work will be discussed in Chapter 4. Chapter 5 presents the methods used to create realistic benchmark instances for the VNMP. In the following chapters, we present different algorithms for solving the VNMP. Since the VNMP is $\mathcal{NP}$-complete and also hard to solve in practice, we consider (meta)heuristic approaches in addition to exact solution methods. First, we will focus on heuristic methods to generate good solutions in a reasonable amount of time, later parts will focus on exact approaches to find optimal solutions. Chapter 6 contains basic Construction Heuristics, Local Search, and Variable Neighborhood Descent approaches. A Memetic Algorithm is presented in Chapter 7 while Greedy Randomized Search Procedures and Variable Neighborhood Search algorithms may be found in Chapter 8. Preprocessing techniques for VNMP instances are discussed in Chapter 9. Then we leave the heuristic solution methods behind and concentrate on exact approaches for solving the VNMP. In Chapter 10, we apply Constraint Programming and in Chapter 11 Integer Linear Programming methods. In Chapter 12, we study how the developed algorithms might be used to support a VNP when deciding where to increase capacities. Chapter 13 contains an overall comparison of the main algorithms presented and final conclusions. In Appendix A, the detailed results of all algorithms may be found.
Parts of this thesis have been presented at the following conferences and published in the corresponding proceedings (all reviewed). The Construction Heuristics, Local Search and Variable Neighborhood Descent algorithms from Chapter 6 have been published in

> J. Inführ and G. R. Raidl. Solving the Virtual Network Mapping Problem with Construction Heuristics, Local Search, and Variable Neighborhood Descent. In M. Middendorf and C. Blum, editors, *Evolutionary Computation in Combinatorial Optimisation – 13th European Conference, EvoCOP 2013*, volume 7832 of *Lecture Notes in Computer Science*, pages 250–261. Springer, 2013.

The Memetic Algorithm we present in Chapter 7 has been published in

> J. Inführ and G. R. Raidl. A Memetic Algorithm for the Virtual Network Mapping Problem. In H. C. Lau, P. Van Hentenryck, and G. R. Raidl, editors, *Proceedings of the 10th Metaheuristics International Conference*, pages 28–1–28–10, Singapore, 2013.

The Greedy Randomized Search Procedures and Variable Neighborhood Search algorithms that can be found in Chapter 8 have been published in

> J. Inführ and G. R. Raidl. GRASP and Variable Neighborhood Search for the Virtual Network Mapping Problem. In M. J. Blesa et al., editors, *Hybrid Metaheuristics, 8th International Workshop (HM 2013)*, volume 7919 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2013.

A very early form of the Integer Linear Programming approach for solving the VNMP as presented in Chapter 11 has been published in

> J. Inführ and G. R. Raidl. Introducing the Virtual Network Mapping Problem with Delay, Routing and Location Constraints. In J. Pahl, T. Reiners, and S. Voß, editors, *Network Optimization: 5th International Conference (INOC 2011)*, volume 6701 of *Lecture Notes in Computer Science*, pages 105–117, Hamburg, Germany, 2011. Springer.

The application study we perform in Chapter 12 has been published in

> J. Inführ, D. Stezenbach, M. Hartmann, K. Tutschku, and G. R. Raidl. Using Optimized Virtual Network Embedding for Network Dimensioning. In *Proceedings of Networked Systems 2013*, pages 118–125, Stuttgart, Germany, 2013. IEEE.

CHAPTER 2

# Theory and Methodology

## 2.1 Introduction

In this chapter, we give an overview of the theoretical concepts we use in this work, as well as a description of the employed experimental methodology. In Section 2.2, we introduce the basics of combinatorial optimization, give an overview of the associated complexity theory, and present some of the major principles used to solve Combinatorial Optimization Problems (COPs). Section 2.3 covers the basic definitions and associated algorithms from graph theory. The experimental setup, describing for example the computational environment and the employed statistical tests, is outlined in 2.4.

We want to state clearly that it is not in the scope of this work to even give a proper overview of the concepts that we are going to introduce in the following. We will mainly concentrate on the parts relevant for this work and give references to fill the gaps.

## 2.2 Combinatorial Optimization Problems and Solution Methods

### 2.2.1 Combinatorial Optimization Problems

Before we can start outlining different heuristic and exact solution methods for solving instances of Combinatorial Optimization Problems, we first require a definition what problems, instances of problems, and solutions are.

A problem is an abstract description of what needs to be done, usually specified by defining what data is required as input and what is requested as output. The output has to satisfy some constraints. If we are dealing with an optimization problem, the output has an attached value that either has to be minimized or maximized. An instance of a problem is a concrete set of inputs, following the rules as defined by the problem. An instance can be defined more formally as follows [15, 134]:

**Definition 2.2.1** (Instance of a Problem)**.** *Given a finite tuple of variables $X = (x_1, \ldots, x_n)$, domains $D_1, \ldots, D_n$ for those variables (with $D = D_1 \times \ldots \times D_n$), constraints $C$ defined on a subset of $D$ and limiting the allowed combinations of values assigned to variables $X$, and an objective function $f : D \to \mathbb{R}$ that has to be minimized or maximized, a problem instance is defined as quadruple $(X, D, C, f)$.*

A problem can also be viewed as the set of all its instances. We will use the terms problem and instance interchangeably. Given a problem instance, the set of candidate solutions $S$ is the set of all possible assignments of values to variables according to their domains, but not necessarily satisfying the constraints. $S$ is also referred to as search space or solution space. Every $s \in S$ has an assigned objective value $f(s)$.
The set of feasible solutions $S_{\text{feas}}$ is a subset of $S$, containing all candidate solutions that fulfill the constraints. When solving a feasibility (decision) problem, we try to find any member of $S_{\text{feas}}$. When solving an optimization problem, we are searching for a special feasible solution; one that has the best possible objective value. In the following, we assume that smaller values are better, i.e., we focus on minimization problems. The definitions for maximization problems are analogous.

**Definition 2.2.2** (Optimal Solution)**.** *A solution $s_{opt} \in S_{feas}$ is said to be globally optimal, if $\forall s \in S_{feas} : f(s_{opt}) \leq f(s)$. Note that there may be multiple globally optimal solutions with the unique globally optimal objective value.*

If the domains of the variables of a problem (instance) are continuous, we are dealing with a Continuous Optimization Problem. The problems we are going to solve in this work have discrete variable domains and therefore belong to the class of Combinatorial Optimization Problems.

### 2.2.2 Complexity Theory

When we use an algorithm to solve a problem, we are usually interested in how the algorithm behaves in terms of run-time or memory requirements when the size of the problem (e.g., number of variables) increases. The tools available from the field of computational complexity theory [61, 67, 110, 134, 159, 160] can help us to find useful answers.
From this field, we get the following definitions:

**Definition 2.2.3** (Time Complexity)**.** *The time complexity function of an algorithm gives the largest amount of time needed to solve problem instances of a particular size (denoted by $n$), for all possible values of $n$.*

**Definition 2.2.4** (Big-Oh)**.** *A function $f(n)$ is in $\mathcal{O}(g(n))$, iff there exist constants $c > 0$, $n_0 > 0$ such that $\forall n > n_0 : |f(n)| \leq c \cdot |g(n)|$. Informally, this means that $f(n)$ grows asymptotically not faster than $g(n)$ for $n \to \infty$ when neglecting scaling by a constant.*

**Definition 2.2.5** (Big-Theta)**.** *A function $f(n)$ is in $\Theta(g(n))$, iff there exist constants $c_1 > 0$, $c_2 > 0$ and $n_0 > 0$, such that $\forall n > n_0 : c_1 \cdot |g(n)| \leq |f(x)| \leq c_2 \cdot |g(n)|$. Informally, this means that $f(n)$ grows asymptotically as much as $g(n)$ for $n \to \infty$ when neglecting scaling by a constant.*

**Definition 2.2.6** (Polynomial Time Algorithm)**.** *An algorithm runs in polynomial time, if its time complexity is in $\mathcal{O}(n^k)$, for a constant $k < \infty$.*

**Definition 2.2.7** (Exponential Time Algorithm)**.** *If the time complexity function of an algorithm cannot be bounded by a polynomial, it is called an exponential time algorithm.*

**Definition 2.2.8** (Well Solved Problem)**.** *A problem is considered well solved, if a polynomial time algorithm solving the problem is known. A well solved problem is also called efficiently solvable.*

In complexity theory, all efficiently solvable problems belong to the class $\mathcal{P}$ (for polynomial time). Unfortunately, for many of the important COPs the known algorithms require exponential time. Exponential time basically means that an exponential number of solutions have to be enumerated to find the optimal solution, at least in the worst case. The problems not belonging to $\mathcal{P}$ can be split into two groups, those for which it is easy (in $\mathcal{P}$) to check whether a potential solution is valid, and those for which even this check is not in $\mathcal{P}$. The problems for which it is easy to determine if a candidate solution satisfies all constraints form the class $\mathcal{NP}$ (for nondeterministic polynomial). It holds that $\mathcal{P} \subseteq \mathcal{NP}$, but whether $\mathcal{P} = \mathcal{NP}$ is still an open problem. It is conjectured that $\mathcal{P} \neq \mathcal{NP}$. The "hardest" problems within $\mathcal{NP}$ are $\mathcal{NP}$-complete:

**Definition 2.2.9** ($\mathcal{NP}$-complete)**.** *A problem is $\mathcal{NP}$-complete, if it is in $\mathcal{NP}$ and every problem in $\mathcal{NP}$ is reducible to it in polynomial time.*

Based on this definition, to prove that a problem A is $\mathcal{NP}$-complete, we first need to prove membership in $\mathcal{NP}$. If there is a polynomial time algorithm capable of checking whether a candidate solution for A is feasible, A belongs to $\mathcal{NP}$. The second condition for $\mathcal{NP}$-completeness is that all problems in $\mathcal{NP}$ can be reduced to it in polynomial time. That means, for every problem B in $\mathcal{NP}$, there has to be a polynomial time algorithm capable of transforming any input of B into an input of A and a solution of A to a solution of B. More informally, A can solve problem B, and therefore cannot be easier to solve than B. Following the definition of $\mathcal{NP}$-completeness, it is sufficient to show a polynomial time algorithm capable of reducing another $\mathcal{NP}$-complete problem to A. The first proof determining the $\mathcal{NP}$-completeness of a problem was achieved by Cook [33], which of course could not use this shortcut. In addition to $\mathcal{NP}$-completeness, there is also the notion of $\mathcal{NP}$-hardness. A problem is $\mathcal{NP}$-hard if any problem in $\mathcal{NP}$ can be reduced to it, but it does not necessarily lie in $\mathcal{NP}$ itself.
To sum it all up, COPs can be split up roughly into "easy" or tractable problems (those belonging to $\mathcal{P}$), and "hard" or intractable problems (those being $\mathcal{NP}$-complete). The "hopeless" problems not even belonging to $\mathcal{NP}$ are usually not considered. It is important to note that the (in)tractability of a problem is not in lockstep with its solvability in practice. It might be infeasible to solve a problem in $\mathcal{P}$ (possibly due to instance size), and a problem in $\mathcal{NP}$ might be easily solvable for the relevant instance sizes. In addition, there exist pseudo-polynomial algorithms for some $\mathcal{NP}$-complete problems that are very effective in practice. An algorithm is pseudo-polynomial if its run-time does not only depend polynomially on the input size, but also on the input numbers themselves. A pseudo-polynomial algorithm is allowed to run longer when

for instance a cost value contained in the problem instance is changed from 100 to 1000. $\mathcal{NP}$-complete problems, for which a pseudo-polynomial algorithms exist are called weakly $\mathcal{NP}$-complete. Otherwise, they are called strongly $\mathcal{NP}$-complete.

### 2.2.3 Multiobjective Problems and Pareto Optimality

In the previous sections, we have only considered the case of a single objective, which tells us how good a solution to a problem is. However, in reality there are frequently multiple objectives to be considered and to make matters worse, no solution might be best according to all objectives. Just think about the trade-off between quality and cost or execution time and solution quality.
The concept of Pareto optimality has been introduced to deal with these kinds of problems and is based on domination. A solution $s$ strictly dominates a solution $s'$, if $s$ is at least as good as $s'$ according to all objectives and better than $s$ according to some objectives. The Pareto-optimal solutions are those which are not strictly dominated by another solution. An interpretation of this is that the Pareto-optimal solutions can only be improved according to one objective if another objective is made worse. The set of Pareto-optimal solutions is called the Pareto-front and is the dividing line between the best feasible and infeasible solutions to a problem.
Now that we have covered the required theory, we can start discussing solution methods for COPs. Broadly speaking, they are divided into two categories. First we will present some heuristic methods. Heuristic methods try to find good solutions in a short amount of time, but cannot give any guarantees with respect to the solution quality. The other category are the exact methods. Exact methods can give quality guarantees and even find provably optimal solutions. Since they are applied to $\mathcal{NP}$-complete problems, their run-time is exponential in the worst case.

### 2.2.4 Construction Heuristics

The first, and very basic, heuristic method we cover are Construction Heuristics. A Construction Heuristic tries to build a solution by iteratively adding components to an initially empty solution until it is complete. The selection of the components usually follows a greedy rule. That means that the component that currently seems to be the best choice (e.g., increases profit the most) is chosen, without regard for possible future consequences of this choice. In addition, once a decision has been made and a component has been added to the solution, it cannot be removed again. These properties make Construction Heuristics very fast and easy to implement and analyze. As downside, the created solutions are frequently far away from the optimal solution [16]. Depending on the problem that has to be solved, the created solutions may not even be feasible. Nevertheless, the application of a construction heuristic is in practice often the first step when solving a problem, since more sophisticated improvement heuristics may build upon it.

### 2.2.5 Local Search

Local Search [1, 15, 134] is a basic improvement heuristic that starts from a starting solution $s \in S$ and tries to find a solution $s'$ in a neighborhood of $s$ that is better than $s$. If an improved

**Algorithm 2.1:** Local Search

**Input** : Solution $s$
**Output**: Possibly improved solution $s$

```
while stopping criteria not met do
    choose s' ∈ N(s);
    if f(s') ≤ f(s) then
        s = s';
    end
end
return s;
```

solution can be found, it replaces the starting solution and the search continues. A neighborhood structure is defined as follows.

**Definition 2.2.10** (Neighborhood Structure)**.** *A function $N : S \to 2^S$ is called a neighborhood structure. It assigns a set of neighbors $N(s) \subseteq S$, called neighborhood, to each solution $s \in S$.*

A neighborhood structure is usually not defined by explicit enumeration, but rather implicitly by giving a transformation rule how a solution is to be changed to create its set of neighbors, i.e., its neighborhood. These transformations are usually local changes, like swapping or replacing single components of the solution. The larger the neighborhood, i.e., the more solutions are reachable from an initial solution, the higher the probability of finding an improving solution in general. Of course, larger neighborhoods usually lead to higher run-time requirements.

**Definition 2.2.11** (Locally Optimal Solution)**.** *If $N(s)$ of solution $s$ does not contain solutions better than $s$, $s$ is locally optimal with respect to $N$. A locally optimal solution can be arbitrarily bad compared to the globally optimal solution.*

The main strategic choice for Local Search, besides the definition of the used neighborhood, is the method of selecting a neighbor. There are three commonly used selection strategies. First-improvement enumerates the solutions of $N(s)$ and stops when the first improving one has been found. With best-improvement, all solutions of $N(s)$ are enumerated and the best solution among them is chosen. Random-neighbor randomly generates neighbors from $N(s)$ and selects the first improving solution.
A natural stopping point of Local Search is when a local optimum has been reached. However, based on the size of the employed neighborhood, other limits such as the number of iterations, elapsed run-time, or no improvement in a certain number of iterations might be chosen. Algorithm 2.1 shows the outline of Local Search.
An important type of neighborhoods are ruin-and-recreate neighborhoods [154]. Instead of specifying transformation rules, they define how a solution is to be partially destroyed, i.e., how some components of the solution are to be removed. For recreating, any method for solving COPs can be applied, as long as it can handle partially fixed solutions. Depending on the chosen recreation method, these neighborhoods can be very powerful.

**Algorithm 2.2:** Variable Neighborhood Descent

**Input**: Initial solution $s$
**Output**: Possibly improved solution $s$

```
1  l = 1;
2  while l ≤ k do
3      select s' ∈ N_l(s) by first- or best-improvement;
4      if f(s') < f(s) then
5          s = s';
6          l = 1;
7      end
8      else
9          l = l + 1;
10     end
11 end
12 return s;
```

### 2.2.6 Variable Neighborhood Descent

An extension of Local Search is Variable Neighborhood Descent [74], which is shown in Algorithm 2.2. Instead of one neighborhood structure, a set $\{N_1, \ldots, N_k\}$ is utilized. An initial solution is improved by $N_1$ until no more improvements can be found, then $N_2$ is applied. If this neighborhood is not able to improve the solution, the next one is tried. If $N_k$ fails, VND terminates. If at any point an improvement is found, the algorithm goes back to $N_1$. At the end of VND, the solution is locally optimal for all considered neighborhoods. The neighborhoods are usually searched in first-improvement or best-improvement fashion.
This method is especially promising if there is a set of neighborhood structures which complement each other very well. For example, if a problem that has to be solved can be decomposed into a part that deals with route planning and in another part that deals with packing, it is natural to use one neighborhood structure that focuses on the routing and another one that focuses on the packing aspect. Usually, the neighborhood structures are ordered according to their size, with the smallest ones first.

### 2.2.7 Variable Neighborhood Search

The main drawback of VND is that it only focuses its search on the part of the search space around the initial solution. As such, it has a heavy focus on intensification, that means it only tries to improve the initial solution, but does not diversify the search by sampling solutions from other parts of the search space. The General Variable Neighborhood Search (VNS) [73–75] rectifies this problem by using the very scheme of VND around VND once again, with another set of larger neighborhood structures $\mathcal{N}_1, \ldots, \mathcal{N}_k$ that are only sampled by the random-neighbor strategy. These are the so-called shaking neighborhoods and not meant for improving the solution directly, but rather to move the search to another part of the search space and leave

**Algorithm 2.3:** General Variable Neighborhood Search

**Input**: Initial solution $s$
**Output**: Possibly improved solution $s$

```
while stopping criteria not met do
    l = 1;
    while l ≤ k' do
        randomly select s' ∈ N_l(s) // diversification;
        s' = VND(s') // intensification;
        if f(s') < f(s) then
            s = s';
            l = 1;
        end
        else
            l = l + 1;
        end
    end
end
return s;
```

the basin of attraction of the VND's neighborhoods. The basin of attraction of a VND solution $s$ is the set of all solutions $s'$ which are transformed into $s$ by VND. A move in $\mathcal{N}_1$ causes the least change to the current solution, while $\mathcal{N}_k$ perturbs the current solution a lot. VNS is a very successful metaheuristic for Combinatorial Optimization Problems, for more details and a survey of applications see [76]. The general outline is presented in Algorithm 2.3.

### 2.2.8 Greedy Randomized Adaptive Search Procedure

The Greedy Randomized Adaptive Search Procedure (GRASP) [52, 53] is an extension of Construction Heuristics in combination with Local Search. It works by continually repeating two steps. The first step is the randomized greedy construction of a solution to the problem to be solved. A second step is applying a local improvement technique to the constructed solution. These two steps are repeated until a termination criterion (like run-time or number of iterations) is reached. The best found solution is the final result of GRASP. How the randomized greedy solution construction works is a central aspect of GRASP. It iteratively builds a solution by adding components that seem good (but not necessarily the best) according to a greedy criterion. All possible components are collected in a candidate list (CL). A restricted candidate list (RCL) is derived from the CL, usually by selecting the best $k$ candidates, where $k$ is a specified parameter, or all parts whose greedy evaluation lies above a certain quality threshold. The actual component that is added to the solution is selected uniformly at random from this RCL. This procedure usually leads to promising and at the same time diversified solutions for local optimization. Comprehensive overviews of GRASP can be found in [54, 147]. For hybridization techniques see [55]. The outline of GRASP is shown in Algorithm 2.4.

**Algorithm 2.4:** Greedy Randomized Adaptive Search Procedure

**Input**: Instance of a problem as quadruple $(X, D, C, f)$
**Output**: Solution $s$

```
1  s = ∅;
2  while stopping criteria not met do
3      build CL from unassigned variables of X and their domains;
4      s' = ∅;
5      while s' is not a complete solution do
6          build RCL from CL;
7          randomly select an element s_i from RCL;
8          s' = s' ∪ {s_i};
9          CL = CL \ {s_i};
10     end
11     (locally) improve s';
12     if s == ∅ ∨ f(s') < f(s) then
13         s = s';
14     end
15 end
16 return s;
```

### 2.2.9 Genetic Algorithm

A Genetic Algorithm (GA) [80] is a nature-inspired population-based algorithm that can be used to solve Combinatorial Optimization Problems. An overview can be found in [161]. It mimics natural evolution as described by Darwin [40] and Mendel [126] by applying three different methods to a set of solutions, called population: selection, crossover, and mutation. The task of the selection procedure is to choose promising solutions from the population as a basis for creating new solutions. The crossover procedure combines two selected solutions (the parents) such that characteristics of both parents are inherited to the offspring. Mutation changes the offspring in a small way so that new solution properties may emerge. Depending on the particular GA variant, multiple offspring may be collected to form a new population (generational GA) or the offspring is immediately reinserted into the population (steady-state GA) where it replaces a solution.

Common methods for selecting solutions from the population are tournament selection and roulette-wheel selection. With tournament selection, $k$ solutions are randomly chosen from the population and the best one is the result of the selection procedure. The parameter $k$ controls the level intensification caused by the selection procedure, higher values of $k$ mean that only the very best solutions have a significant chance of being selected. For $k = 2$, this method is called binary tournament selection. In roulette-wheel selection, every solution in the population is assigned a selection probability proportional to its solution quality, the best solutions having the highest probability of being selected.

For a description of the common crossover operators, we assume that the solutions are repre-

**Algorithm 2.5:** Genetic Algorithm

**Input**: Instance of a COP
**Output**: Solution $s$

```
1  P ... initial population;
2  while stopping criteria not met do
3      O ... empty set of offspring;
4      while offspring O not sufficient do
5          if crossover condition satisfied then
6              select parent solutions P' from P;
7              select crossover parameter;
8              o = crossover(P');
9          end
10         if mutation condition satisfied then
11             select mutation parameters;
12             o = mutate(o);
13         end
14         evaluate fitness of offspring o;
15         O = O ∪ {o};
16     end
17     P = select(P, O);
18 end
19 return best solution s ∈ P;
```

sented by a vector of integers. To create an offspring with one-point crossover, a location within the vector of integers is selected randomly. All integers up to this location are copied from one parent, the remaining values are copied from the other parent. For two-point crossover, two locations where the source of the values changes are selected. An extreme form of this crossover type is the uniform crossover. For uniform crossover, there is a random decision at every location to determine whether the value will be copied from the first or from the second parent.

The applied mutation operator is usually point mutation. With point mutation, a single value in the vector of integers of a solution is changed to another allowed value. This change might be random, or biased by the previous value so that a result close to the previous value is more likely.

Algorithm 2.5 shows a general GA template [146], which is one of many possible ways of constructing a GA. Additional information about GAs can be found in the literature, e.g., [129, 146, 178].

An extension of GA is the Memetic Algorithm. The Memetic Algorithm (MA) is a combination of GA (or other population based optimization method) and a local improvement technique [130, 131, 142]. The main idea is to use the GA to find promising regions in the search space and then use the local improvement technique to find excellent solutions in those promising regions. There is a tradeoff between the time spent in the GA and the time spent executing the local improvement technique. Without enough time for the GA, it will fail to find promising

regions, without enough time for the local improvement method, the found solutions will not be excellent. Usually, the local improvement technique is only applied to a fraction of the generated solutions. Sometimes, only the very best solution in the population is improved to save on execution time. Another implementation issue surrounding Memetic Algorithms is the treatment of an improved solution. This solution could replace the solution in the population it was derived from. The other possibility is that the improved solution is discarded and the original solution is treated as if it had the solution quality of the improved solution. Both approaches have their drawbacks. If we replace solutions, the diversity in the population might suffer. In the worst case, every solution in the population is transformed into the same solution by the local improvement method. Discarding the improved solution is problematic, because we throw away a lot of work done by the local improvement method. The properties of a highly successful solution are not inserted into the population so that related, therefore it is not possible to create related, possibly even better, individuals.

### 2.2.10 Tree Search and Branch & Bound

We now leave the area of heuristics and focus on exact methods for solving COPs. As outlined previously, for $\mathcal{NP}$-complete problems we have to expect that we need to check an exponential number of solutions in the search space to find the best one. Enumerating all possible solutions only works for the very smallest instances, due to the effect of combinatorial explosion. For every variable added to a problem instance, we multiply the size of the search space by the size of the domain of the added variable. This becomes untenable very quickly, so some better approach is required.

A common concept for exact solution methods is Tree Search, named so because its execution builds a tree graph (see Definition 2.3.18). The root node represents the complete search space to be explored. Tree Search then recursively partitions the search space in mutually disjoint spaces by restricting variable domains or adding constraints. For instance, starting from the root node, we may create two children representing each one half of the search space by fixing a variable with domain $\{0, 1\}$ in one child node to 0 and in the other one to 1. These child nodes may be partitioned further by restricting other variables. In its simplest form, the search tree is explored in a depth-first fashion, which means we partition the search space until we either find a feasible solution, or can prove that given the constraints added during the partitioning, no feasible solution can exist (without having to enumerate all candidate solutions in the current sub-space). If we prove that no feasible solution exists, the current node has failed, and we need to back-track to continue the search, i.e., we need to find another unexplored part of the search space. This is done by going to the parent of the currently failed node and checking if it has children that have not been explored yet. If no more unexplored children exist, we repeat this procedure with the parent of the current node. When we reach the root node in this manner, and it too has no more unexplored children, Tree Search is finished, and in this case has proven that no feasible solution exists. It is easy to see that Tree Search is a complete search method, i.e., if there is a solution it will be found.

In this description, we have only focused on finding a feasible solution (i.e., solving a Constraint Satisfaction Problem), but could already see the strength of Tree Search: whole parts of the search space can be discarded, if we can prove that they do not contain a feasible solution.

The same applies when Tree Search is used to solve COPs, but now, once we have found a feasible solution, we can also discard parts of the search space for which we can prove that no solution better than the currently best found solution exists. This procedure is called Branch & Bound. Branching refers to the partitioning of the search space and bounding to the calculation of an upper bound for solution quality (note that this is a lower bound in case of minimization problems) for the partitions. If the upper bound on solution quality of a part of the search space not better than the currently best known solution, it can be discarded. How branching and bounding are implemented depends on the concrete method employing the Branch & Bound principle.

### 2.2.11 Constraint Programming

Constraint Programming (CP) [7, 119, 152] is an exact method following the Tree Search principle. It is mostly used to solve Constraint Satisfaction Problems but can also solve optimization problems. The branching works as described before, variables are assigned values (or their domains are reduced) to partition the search space. In CP, this is also called the labeling strategy. However, the main power of CP comes from propagation [9], also called filtering in the context of CP. Once a new node in the search tree is created, propagation is performed on the sub-problem it represents. That means that values, which cannot occur in a feasible solution, are removed from the domains of the variables. Assume variables $a$, $b$, and $c$ with domains $D_a = D_b = \{0, 1\}$ and $D_c = \{0, 1, 2, 3\}$ respectively and a constraint $a + b = c$. Propagation would remove 3 from $D_c$, since this value clearly cannot be produced by the sum of $a$ and $b$. Propagation cannot reduce the domains any further, a fixed point is reached and branching is required. Assume we branch by assigning $c$ every value from its domain, i.e., one branch where $c = 0$, one where $c = 1$, and one where $c = 2$. When we consider the first and the third branch, propagation alone is sufficient to find a complete assignment to $a$ and $b$. For the second branch, we end up with $D_a = D_b = \{0, 1\}$ and $D_c = \{1\}$. Here we can see that by propagation alone, we are not able to describe the set of feasible solutions. The domains suggest that the set of feasible solutions (in terms of assignments to $a$ and $b$) is $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$, when in reality it should be $\{(0, 1), (1, 0)\}$. Branching has to be used to find all feasible solutions. If propagation removes all values from the domain of a variable, then we know that the current partial assignment (variables that have been assigned a value due to branching or propagation) is inconsistent, i.e., it is not possible to assign values to the remaining variables such that all constraints are satisfied. In this case, CP backtracks and evaluates another partial assignment.
Propagation methods come in different strengths, i.e., how well they can reduce the domains of variables. This is called the consistency of a propagator. The most common consistency level is domain (or arc) consistency. Two variables are domain consistent if for every value allowed for the first variable there is a value for the second variable such that the constraint between the variables is satisfied. Other consistency levels are node consistency or path consistency.
CP is very flexible with respect to the employed variables and constraints, as long as suitable propagation and branching procedures can be defined. Regarding the types of variables, we have already seen that binary ($a$ and $b$) and integer variables ($c$) can be handled. Real-valued variables are also possible. Set variables are going to be relevant later on. The domain of set variables is the set of all possible sets of integers. Due to combinatorial explosion, it is not

possible to represent the domain of a set variable directly. Indeed, direct representation is already problematic for integer variables with large domains. Instead, the domain is approximated by two sets, the greatest lower bound (GLB) and the least upper bound (LUB). The domain of a set variable contains all sets that can be built by using all elements of the greatest lower bound and any selection of elements from the least upper bound. It is written as $[GLB\ ..\ LUB]$. Intuitively speaking, the value (which is a set) of the set variable has to contain all elements of GLB and may contain elements of the LUB. This representation cannot represent every set of sets. The best representation of the domain $\{\{1,2\},\{1,3\},\{2,3\}\}$ for example is $[\emptyset\ ..\ \{1,2,3\}]$, which also includes sets such as $\{1\}$ or $\{1,2,3\}$. To improve on this, the domain of a set variable usually also contains information on the cardinality of the set, i.e., lower and upper bounds on its size. The example domain can be represented exactly by specifying a lower and upper cardinality bound of 2.
For optimization problems, CP works the same way as for satisfaction problems, but every time a feasible solution is found, the remaining search space is restricted by adding the constraint that the solution has to be better than the current one. This is usually implemented by adding a variable $f$ to the model, which represents the value of the objective function. If propagation can remove all values from the domain of $f$ which are better than the currently best known solution at some node in the search tree, then this node can be pruned immediately.

### 2.2.12 Integer Linear Programming

Integer Linear Programming is a class of methods for solving COPs. In contrast to CP, it is far less flexible with respect to the types of COPs that can be solved. Only if the COP can be modeled as shown by Equations (2.1)–(2.3), i.e., as a system of linear inequalities with a linear objective function and using integer variables, Integer Linear Programming can be applied.

$$z = \min \quad \boldsymbol{c}'\boldsymbol{x} \tag{2.1}$$
$$\text{subject to} \quad \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \tag{2.2}$$
$$\boldsymbol{x} \in \mathbb{Z}^n_+. \tag{2.3}$$

Using additional real-valued variables (Mixed Integer Linear Programming) is also allowed, but it is not possible to model an $\mathcal{NP}$-complete COP exclusively with real-valued variables (unless the model is exponential in size). In the following, we will consider Integer Linear Programming, but everything is applicable to Mixed Integer Linear Programming as well. To solve Constraint Satisfaction Problems, set $\boldsymbol{c} = \boldsymbol{0}$.
Branch & Bound can be applied to solve Integer Linear Programs (ILPs). Some definitions are required before we can cover the details.

**Definition 2.2.12** (Set of Integer-Feasible Solutions)**.** *The set of integer-feasible solutions $X$ of ILP (2.1)–(2.3) is defined as $X = \{\boldsymbol{x} \in \mathbb{Z}^n_+ \mid \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b}\}$.*

Following this definition, an ILP can also be written as $z = \min\{\boldsymbol{c}'\boldsymbol{x} \mid \boldsymbol{x} \in X\}$.

**Definition 2.2.13** (LP Relaxation)**.** *Given an ILP (2.1)–(2.3), the LP relaxation is obtained by replacing the integrality constraints (2.3) by $\boldsymbol{x} \in \mathbb{R}^n_+$.*

**Algorithm 2.6:** LP-based Branch & Bound (for minimization problems)

**Input**: ILP $\min\{\boldsymbol{c}'\boldsymbol{x} : \boldsymbol{x} \in X\}$
**Output**: Optimal solution $\boldsymbol{x}^*$

problem list $L = \{X\}$; `// initialize problem list`
$\boldsymbol{x}^* = \mathbf{0}$; `// best known feasible solution`
$\overline{z} = \infty$; `// objective of best known feasible solution`
**while** $L \neq \emptyset$ **do**
    choose set $X_i$ and remove it from $L$;
    $(\boldsymbol{x}^i_{\mathrm{LP}}, \underline{z}^i) = LP(X_i)$; `// get optimal LP solution and objective`
    **if** $\mathrm{LP}(X_i) = \emptyset$ **then** prune $X_i$ by infeasibility;
    **else if** $\underline{z}^i \geq \overline{z}$ **then** prune $X_i$ by bound;
    **else if** $\boldsymbol{x}^i_{\mathrm{LP}} \in X$ **then** `// LP solution is integer`
        **if** $z^i \leq \overline{z}$ **then**
            $\boldsymbol{x}^* = \boldsymbol{x}^i_{\mathrm{LP}}$; `// update best known solution`
            $\overline{z} = z^i$; `// update best known objective`
        **end**
        prune $X_i$ by optimality;
    **end**
    **else** $L = L \cup \{X_{i,1}, X_{i,2}\}$; `// partition search-space` $X_i$
**end**
return $\boldsymbol{x}^*$;

The LP relaxation can be solved efficiently (in polynomial time) by ellipsoid [104] or interior point methods [103]. Much better in practice and more widely used is the simplex method [37], even though it can have an exponential run-time in the worst case [13]. Details on the simplex method can be found in [13]. By using this method, we can define the LP based Branch & Bound scheme (for minimization problems), as presented in Algorithm 2.6.

The problem list $L$ is the central data structure of this algorithm. It contains all partitions of the search-space that have not yet been considered. When this list is implemented as a stack, we get the standard depth-first search (as we have outlined for Branch & Bound). In practice, there are a lot of improvements possible by intelligently choosing which partition to consider next. In addition to $L$, we also need to store the best known feasible solution (which is the optimal solution at the end of this algorithm), and its objective. During the execution, this objective is an upper bound on the optimal value, hence the overline and initial value of $\infty$.

After the initialization, the algorithm starts in earnest by considering every search space partition (subproblem) in turn, until $L$ is empty. For the currently considered subproblem, we solve the LP relaxation and get the optimal solution of the LP relaxation $\boldsymbol{x}^i_{\mathrm{LP}}$ and its objective $\underline{z}^i$. Since we do not enforce variables $\boldsymbol{x}$ to be integer in the LP relaxation, the LP solution will be fractional in the general case. Its objective is a lower bound (hence underlined) on the objective of the optimal integer solution contained in $X_i$. This is the bounding step, and depending on the result, different actions are executed. If $X_i$ is not even feasible when considering its LP-relaxation

(i.e., there is not even a fractional assignment to all variables that satisfies all inequalities), then $X_i$ cannot contain any integer feasible solutions and can be discarded. If the best fractional solution contained in $X_i$ is not better than the globally best known integer feasible solution, $X_i$ can be discarded without further consideration, since no integer feasible solution that will lead to an improvement exists within it. This step has the potential of quickly recognizing large parts of the search space as uninteresting and therefore hugely increasing performance. If the LP solution happens to be integer, then we store it, if it is better than the best known solution. As a side-effect, this increases the probability of being able to prune by bound in subsequent iterations. Then we discard $X_i$, since we have already identified the optimal solution contained within. If none of those conditions apply, i.e., the LP solution is fractional and better than the currently best known integer solution, then $X_i$ might contain an improving integer solution and further analysis is required.

This leads us to the branching step, where we partition $X_i$ into two sets. This is done by selecting a variable $x$ of the LP solution that has a fractional value $v$, and adding for one partition the constraint $x \leq \lfloor v \rfloor$ and for the other partition $x \geq \lceil v \rceil$. Note that if $x$ is a binary variable (its domain is $\{0, 1\}$), it is now forced to be integer in both partitions. By partitioning $X_i$ in that way, $\boldsymbol{x}^i_{\text{LP}}$ is no longer a feasible solution for both partitions. The optimal LP solution of both partitions is likely worse than $\underline{z}^i$, so in the best case it may be possible to immediately prune both partitions of $X_i$, while we could not prune $X_i$ itself. Due to this branching, we can guarantee that the LP solution will be integer feasible at some point going down the search tree (or no LP solution exists and we need to track back), and the search tree depth is finite.

When modeling a COP with ILP, one aim is finding a strong model, that means that the optimal objective of its relaxation is close to the objective of the optimal integer solution. It is possible to add inequalities to a model to strengthen it, i.e., the inequalities are redundant and do not change the set of integer feasible solutions, but they remove feasible solutions of the relaxation. These inequalities might be added to the model at the beginning or added during search as necessary. An added inequality is also called cut, because it cuts away previously feasible fractional solutions. The problem of finding such a violated cut for a given LP solution is called the (cut) separation problem. It is theoretically possible to add cuts to the model until the solution of the LP relaxation yields the optimal integer solution by means of the cutting plane algorithm [38]. Usually, the cutting plane algorithm is computationally too expensive.

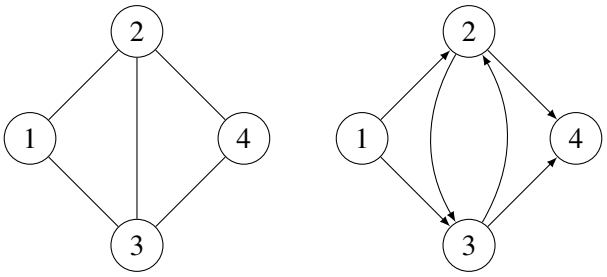
Far more promising is the combination of the idea of cut generation and Branch & Bound, yielding highly effective Branch & Cut algorithms. At each node in the Branch & Bound tree, cuts are separated to strengthen the LP relaxation of the current subproblem. Procedures to automatically derive certain types of generic cuts, such as Gomory cuts [68] or Lift-and-Project cuts [8], are included in the major ILP solvers. With Branch & Cut, it is also possible so solve models which are exponential in the number of required inequalities, for instance forbidding all cycles of a specific length when formulating a problem on a graph structure. The initial model does not contain these constraints. Only if a constraint is violated by a LP solution, the cut enforcing the constraint is added.

Instead of adding constraints, it is also possible to dynamically add variables to a model. For the LP relaxation, this is called column generation [64], as a new variable adds a column to the coefficient matrix $\boldsymbol{A}$. When combined with Branch & Bound for solving ILPs, it is called

Branch & Price. With this method, it is possible to use models with an exponential number of variables. Initially, an LP containing only a small fraction of the variables is solved. An important condition here is that this small fraction of variables has to permit a feasible solution. This is problematic when even finding a feasible solution is $\mathcal{NP}$-complete. After the LP is solved, a variable that is currently not included in the model and may improve the solution of the LP has to be identified. This is called the pricing problem. Only variables with negative reduced costs may improve the solution, see the simplex algorithm [37] for details. By means of Danzig-Wolfe decomposition [39], it is possible transform a compact model into a model with exponentially many variables. The transformed model usually has much tighter LP bounds when compared to the original compact model which reduces the number of Branch & Bound nodes that need to be considered.

As an example for a model with exponentially many variables, consider a variant of the Vehicle Routing Problem (VRP), where packets have to be delivered to customers by a fleet of vehicles. Each vehicle starts from the central depot, delivers packets to a set of customers and returns to the depot. The total length of the driven distance by all vehicles has to be minimized. All packets that need to be delivered by a vehicle need to fit into the vehicle. One decision variable could represent a specific route of a vehicle that is valid with respect to the capacity constraint of the vehicle. There are exponentially many such variables. In the model for this VRP, we select one route for each vehicle, such that every customer is visited exactly once, and the total cost of the routes is minimized.

One point of interest of the column generation approach (besides potentially better LP bounds) is that it is possible to hide arbitrary constraints within the pricing subproblem and the LP model deals only with variables that respect those constraints. However, there are also some downsides. First of all, column generation cannot be stopped during execution and still yield a valid LP bound. Only after we have proven that no more improving variables exist is the LP bound valid. Otherwise it might be too high (for a minimization problem), which means we might then prune subproblems during Branch & Bound that may still contain improving integer feasible solutions. The cutting plane method can be stopped and we get a valid (if not as strong as possible) lower bound. Proving that no further cuts exist is only necessary when an integer solution is found, otherwise the integer solution could be invalid for the complete (exponential) model. Branching sometimes is also problematic for Branch & Price. Assume we need to chose (i.e., set to 1) one of exponentially many variables. When we branch on one of those variables, we either fix it to 1 (and are finished unless there are also other variables), or we fix it to 0 and have basically no additional restriction since there are still exponentially many other variables that may need to be generated and branched upon. As a result, the search tree is very skewed. The third major hurdle of column generation are convergence issues. Without taking care, a lot of variables will be generated that improve the LP bound only a tiny bit which leads to long convergence times. Stabilization techniques [45] are required to combat this. Despite these challenges, Branch & Price has been very successfully applied to Cutting & Packing and Network Design problems. For an overview on column generation, see [44, 118].



(a) Simple graph (b) Directed graph

Figure 2.1: A simple graph and a directed graph.

## 2.3 Graph Theory

In this section, we review the basics of graph theory and related algorithms required in later chapters of this work. We follow the definitions from [176].

**Definition 2.3.1** (Graph)**.** *A graph $G = (V, E)$ with $n = |V|$ vertices an $m = |E|$ edges consists of a vertex set $V(G) = \{v_1, \ldots, v_n\}$ and edge set $E(G) = \{e_1, \ldots, e_m\}$, where each edge consists of two (possibly equal) vertices called its endpoints. If $e = \{u, v\} \in E(G)$, then $u$ and $v$ are adjacent. A loop is an edge whose endpoints are equal. Parallel edges are edges that have the same pair of endpoints. A simple graph is a graph having no loops or multiple edges.*

We will use $V$ and $E$ as shorthand for $V(G)$ and $E(G)$ when $G$ is clear from the context. We will use the word graph to denote simple graphs unless stated otherwise.

**Definition 2.3.2** (Directed Graph)**.** *A directed graph $G = (V, A)$ consists of a vertex set $V(G)$ and arc set $A(G)$, where each arc is an ordered pair of vertices. If $a = (u, v) \in A(G)$, $u$ is the head and $v$ the tail of the arc. The choice of head and tail gives an arc a direction, from head to tail. A simple directed graph is a directed graph in which each ordered pair of vertices occurs at most once as an arc.*

We will use $A$ as shorthand for $A(G)$ when $G$ is clear from the context.
Figure 2.1 shows an example for a simple graph and a directed graph. Both graphs have a vertex (node) set $V = \{1, 2, 3, 4\}$. The simple (undirected) graph 2.1a has edge set $E = \{\{1, 2\}, \{1, 3\}, \{2, 4\}, \{2, 3\}, \{3, 4\}\}$. The directed graph 2.1b has arc set $A = \{(1, 2), (1, 3), (2, 3), (3, 2), (2, 4), (3, 4)\}$.

**Definition 2.3.3** (Dense and Sparse Graphs)**.** *A graph $G$ is considered to be dense if $m \propto n^2$, it is sparse if $m \propto n$.*

**Definition 2.3.4** (Source and Target of an Arc)**.** *Given an arc $a = (u, v) \in A$, $s(a)$ denotes the source (head) of the arc, i.e., $s(a) = u$ while $t(a)$ denotes the target (tail) of the arc, i.e., $t(a) = v$.*

**Definition 2.3.5** (Subgraph)**.** *A subgraph of a graph G is a graph H such that* $V(H) \subseteq V(G)$ *and* $E(H) \subseteq E(G)$.

**Definition 2.3.6** (Degree)**.** *The degree of vertex* $v$ *of a simple graph G, written* $\delta(v)$*, is the number of edges containing* $v$.

**Definition 2.3.7** (Incident Edges)**.** *The incident edges of vertex* $v$ *of a simple graph G are the edges containing* $v$.

**Definition 2.3.8** (In-Degree)**.** *The in-degree of a vertex* $v$ *of a directed graph G, written* $\delta_v^-$*, is the number of arcs with* $v$ *as target.*

**Definition 2.3.9** (Out-Degree)**.** *The out-degree of a vertex* $v$ *of a directed graph G, written* $\delta_v^+$*, is the number of arcs with* $v$ *as source.*

**Definition 2.3.10** (Shadow)**.** *The shadow of a directed graph G is an undirected graph S with the same vertex set. The edge set of S is chosen such that adjacent vertices in G are adjacent in S and vice versa.*

**Definition 2.3.11** (Reversal Graph)**.** *The reversal graph* $G_R$ *of a directed graph G contains all vertices of G, and* $(u, v) \in A(G_R)$ *if and only if* $(v, u) \in A(G)$.

The degree of node 2 of the simple graph in Figure 2.1a is 3, i.e., $\delta(2) = 3$. The incident edges of node 1 of this graph are $\{1, 2\}$ and $\{1, 3\}$. The in-degree of node 3 of the directed graph in Figure 2.1b is 2, i.e., $\delta_3^- = 2$. The out-degree of node 4 is 0, i.e., $\delta_4^+ = 0$. Graph 2.1a is the shadow of graph 2.1b.

**Definition 2.3.12** (Path)**.** *A path p of length k is a sequence* $v_0, e_1, v_1, e_2, \ldots, e_k, v_k$ *of vertices and edges such that* $e_i = \{v_{i-1}, v_i\}$, $\forall i \in [1, k]$*. A path with no repeated vertices is called simple. The source or start of p,* $s(p)$*, is* $v_0$*, the target or end,* $t(p)$*, is* $v_k$*. The nodes of the path p are* $N(p) = \{v_0, \ldots, v_k\}$*, the edges of the path are* $E(p) = \{e_1, \ldots, e_k\}$.

**Definition 2.3.13** ((Simple) Cycle)**.** *A path p of length at least 2 is called a cycle, if* $s(p) = t(p)$*. If p is simple (with the exception of its source and target node), the cycle is called simple.*

**Definition 2.3.14** (Weighted Path)**.** *Given a path p in graph G and a function* $w : E(G) \to \mathbb{R}$*, the weight of p is* $\sum_{e \in E(p)} w(e)$*. If a weight function for edges is available, the length of a path refers to its weight instead of the number of its edges.*

**Definition 2.3.15** (Node and Edge Disjoint Path)**.** *Two paths* $p_1, p_2$ *are node disjoint if* $N(p_1) \cap N(p_2) = \emptyset$*. They are edge disjoint if* $E(p_1) \cap E(p_2) = \emptyset$.

The definitions of a path for directed graphs and arc disjointness are analogous. Let $p_1 = 1, (1, 2), 2, (2, 4), 4$ in graph 2.1b. Then the length of $p_1$ is 2, $s(p_1) = 1$, $t(p_1) = 4$, $N(p_1) = \{1, 2, 4\}$ and $A(p_1) = \{(1, 2), (2, 4)\}$. Let $p_2 = 2, (2, 3), 3$ and $p_3 = 3$. Then $p_1$ and $p_2$ are arc disjoint but not node disjoint, $p_1$ and $p_3$ are arc and node disjoint.

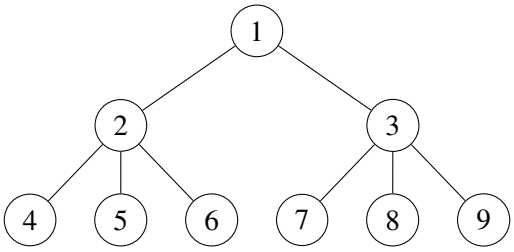


Figure 2.2: A tree.

**Definition 2.3.16** (Connected Graph)**.** *A graph $G$ is connected if there is a path between every pair of vertices from $V(G)$.*

**Definition 2.3.17** (Connected Components)**.** *The connected components of a graph $G$ are its maximal connected subgraphs. A connected graph has one connected component.*

**Definition 2.3.18** (Tree)**.** *A graph $G$ is a tree, if it is connected and the path between each pair of vertices is unique. For trees, $n = m + 1$. The root of a tree is a node of $V(G)$ and often used as starting point for algorithms on trees.*

Figure 2.2 shows an example tree. Node 1 is the root of the tree. Nodes 4–9 are the leafs. Nodes 2 and 3 are intermediate nodes and the children of 1. The parent of 4 is 2. Nodes 7–9 are siblings. Node 1 is ancestor of Node 7. The height (or depth) of the tree is 2 and is the length of the longest path from the root to one of the leafs.

**Definition 2.3.19** (Depth-First Search)**.** *Depth-first search is a traversal order of the nodes of the tree. Starting at the root, we select one of its children and then one of the children of the child and so on, until we have reached a leaf. Then we go back in direction of the root (back-tracking). We stop at the first node that still has unexplored children and continue with one of those. The search is finished if all children of the root have been explored.*

Depth-first search applied to the tree in Figure 2.2 could visit nodes in the following order (not showing nodes visited during back-tracking): $1, 3, 8, 9, 7, 2, 4, 6, 5$.

**Definition 2.3.20** (Biconnected Graph)**.** *A graph $G$ is biconnected, if any vertex of $V(G)$ or edge of $E(G)$ can be removed and $G$ remains connected.*

**Definition 2.3.21** (Articulation Point)**.** *Vertex $v$ of graph $G$ is an articulations point, if its removal increases the number of connected components.*

**Definition 2.3.22** (Bridge)**.** *Edge $e$ of graph $G$ is a bridge, if its removal increases the number of connected components.*

**Definition 2.3.23** (Block)**.** *A block of a graph $G$ is a maximal connected subgraph of $G$ that has no articulation points.*

If a block has more than two vertices, then it is biconnected. If it has two vertices, the edge connecting them has to be a bridge. Two blocks in the same graph share at most one vertex, hence the blocks of a graph partition its edge set, i.e., all edges belong to exactly one block. A shared vertex has to be an articulation point, every articulation point belongs to at least two blocks.

**Definition 2.3.24** (Block Tree)**.** *The block tree $B$ is built from a connected graph $G$, by adding all articulation points of $G$ to $B$, and one vertex for every block of $G$. Vertices $v_1, v_2$ of $B$ are connected, if $v_1$ represents an articulation point of $G$ which belongs to the block represented by $v_2$.*

**Definition 2.3.25** (Weakly Connected Graph)**.** *A directed graph is weakly connected, if its shadow is connected.*

**Definition 2.3.26** (Strongly Connected Graph)**.** *A directed graph is strongly connected, if there is a path in both directions between every pair of vertices.*

**Definition 2.3.27** (Strongly Connected Components)**.** *The strongly connected components of a directed graph are its maximal strongly connected subgraphs.*

The strongly connected components of a graph $G$ can be calculated in $\mathcal{O}(m + n)$ by using Tarjan's algorithm [166] based on depth-first search.

**Definition 2.3.28** (Strong Articulation Point)**.** *A vertex is a strong articulation point, if its removal increases the number of strongly connected components of a directed graph.*

**Definition 2.3.29** (Strong Bridge)**.** *An arc is a strong bridge, if its removal increases the number of strongly connected components of a directed graph.*

**Definition 2.3.30** (Flowgraph)**.** *A flowgraph $G(s) = (V, A, s)$ is a directed graph with a start vertex $s$ in $V$ such that every vertex in $V$ is reachable from $s$.*

**Definition 2.3.31** (Dominator)**.** *Given a flowgraph $G(s)$, vertex $u$ is a dominator of vertex $v$ if all paths from $s$ to $v$ include $u$. The trivial dominators of $u$ are $s$ and $u$. $D(s)$ is the set of non-trivial dominators in $G(s)$.*

**Definition 2.3.32** (Immediate Dominator)**.** *Given a flowgraph $G(s)$, vertex $u$ is an immediate dominator of $v$ if $u$ is a dominator of $v$ and every other non-trivial dominator of $v$ also dominates $u$. The immediate dominator is unique.*

**Definition 2.3.33** (Dominator Tree)**.** *The dominator tree $DT(s)$ of a flowgraph $G(s)$ contains all vertices of $G$. There is an arc from a vertex $u$ to a vertex $v$ in $DT(s)$ if $u$ is the immediate dominator of $v$. $DT(s)$ is a tree rooted at $s$, the dominators of a vertex in $G(s)$ are all its ancestors in $DT(s)$*

**Definition 2.3.34** (Planar Graph)**.** *A graph $G$ is called planar, if it can be drawn in the plane without edge crossings.*

**Definition 2.3.35** (Diameter of a Graph)**.** *The diameter of a graph is the length of the longest shortest path between any pair of vertices.*

**Definition 2.3.36** (Small World Graph)**.** *A graph is a small world graph, if the average degree of each node is small, but the graph also has a small diameter. The diameter grows proportionally to the logarithm of the number of vertices.*

### 2.3.1 Dominators

Efficiently calculating dominators and the dominator tree has been an open problem for a long time. Lengauer and Tarjan [115] presented an algorithm solving this problem in $\mathcal{O}(m\alpha(m,n))$, where $\alpha(m,n)$ is the extremely slow-growing functional inverse of the Ackermann function, in 1979. Truly linear-time algorithms have been proposed by Harel [77], Alstrup [4] and Buchsbaum [22]. These algorithms either turned out to be wrong or far too complicated for a practical implementation. Georgiadis et al. [63] were able to present an implementable algorithm for finding dominators in $\mathcal{O}(m+n)$ in 2004 and Buchsbaum et al. [23] were able to correct their algorithm in 2005. The best source for actually implementing a linear time dominator algorithm seems to be the work of Buchsbaum, Georgiadis and Tarjan et al. [21] from 2008. For an easily implementable algorithm for dominators in $\mathcal{O}(n^2)$ see the work of Cooper et al. [34].

### 2.3.2 Strong Articulation Points

The advances made with algorithms for finding the dominators in a flowgraph enabled Italiano et al. [94, 95] to formulate a $\mathcal{O}(m+n)$ algorithm for finding all strong articulation points in a directed strongly connected graph $G$. We will just present the main ideas here and refer to the referenced work for more details and proofs.
The first step of the algorithm is to determine for an arbitrary node $s$ if it is a strong articulation point. This is done by removing $s$ from $G$ and checking if the remainder is still strongly connected, which can be done in $\mathcal{O}(m+n)$. In the second step, we calculate the dominators in $G(s)$ and its reversal $G_R(s)$, which is also in $\mathcal{O}(m+n)$. These dominators (possibly together with $s$ depending on the outcome of the first step) give all strong articulation points of $G$.
To see why this is so, consider the following argument. It is clear that every dominator has to be an articulation point, since crossing a dominator is the only way to reach the node it dominates. Removing the dominator means that the dominated nodes cannot be reached any more, which increases the number of strongly connected components, the defining property of articulation points. Therefore, we only need to be certain that we do not miss any strong articulation points, i.e., every strong articulation point has to be a dominator in either $G(s)$ or $G_R(s)$. Assume there is a strong articulation point $a$ and a node $b$, node $b$ being in another strongly connected component than $s$ if $a$ were to be removed. In $G$, there have to be paths from $s$ to $b$ and from $b$ to $s$. In one direction, there is only allowed to be a single path, which has to cross $a$, otherwise this would violate the assumption that $a$ is a strong articulation point. If the path from $s$ to $b$ crosses $a$, then $a$ is a dominator in $G(s)$. If the path from $b$ to $s$ crosses $a$, then $a$ is a dominator in $G_R(s)$. Therefore, it is not possible to miss a strong articulation point by using the outlined algorithm. Basically the same method can be used to find all strong bridges.

### 2.3.3 All Pair Shortest Path

The All Pair Shortest Path Problem is defined as follows:

**Definition 2.3.37** (All Pair Shortest Path Problem)**.** *Given a graph $G$ and a function $w : E(G) \to \mathbb{R}$, determine for each pair of vertices of $G$ the shortest path.*

There are two well-known algorithms for solving this problem. Both allow negative edge weights, but no cycles of negative length. The first algorithm is Johnson's algorithm [97], solving the problem in $\mathcal{O}(n^2 \log(n) + nm)$ by essentially calculating for each vertex in the $G$ the shortest path to all other vertices. The alternative is the Floyd-Warshall algorithm [56] requiring a run-time of $\mathcal{O}(n^3)$. The modern implementation of this algorithm is essentially a series of $n - 1$ matrix multiplications [93].
Based on the run-time complexities, Johnson's algorithm is the fastest choice for sparse graphs, while the Floyd-Warshall algorithm has an advantage for dense graphs. In this work, we deal with very sparse graphs, so Johnson's algorithm is used to solve the All Pair Shortest Path Problem.

## 2.4 Experimental Setup

### 2.4.1 Computational Environment

All computational results presented in this thesis have been achieved on Intel Xeon E5540 multi-core systems with 2.53 GHz and 24 GB RAM, which corresponds to 3 GB RAM per core. The implemented algorithms only utilize one core. All reported run-times are CPU-times, as opposed to wall-clock time. A memory limit of 5 GB has been used, unless otherwise specified. If an algorithm exceeds the memory limit during execution, it is aborted.

### 2.4.2 Statistical Tests

In the course of this thesis, we will often compare the performance of two different algorithms solving the same problem, for instance with respect to the required run-time. A natural question of course is, which algorithm is faster? When algorithm A always requires a run-time in the order of seconds for a particular set of problem instances, and algorithm B always requires a run-time in the order of hours, then we may rightly conclude (assuming that the set of problem instances is representative of the problem) that algorithm A is faster than algorithm B. However, the situation is not so clear most of the time. If there is only a difference of some percent between the average run-times of the two algorithms, it could very well be that it is caused by random chance.
In such cases, statistical hypothesis testing becomes important to distinguish between random chance and a true difference. In statistical hypothesis testing, one first states a null and alternative hypotheses. For our example, the null hypothesis would be that algorithm A is not faster than algorithm B, while the alternative hypothesis is that A is indeed faster than B. Now we need to select an appropriate statistical test to determine if we can reject the null hypothesis in favor of the alternative given a certain error probability. In our case, we are dealing with paired data.

Table 2.1: Example table showing how the results of the statistical test are presented.

| Size | A | B |
|---|---|---|
| 20 | 5.1 $>$ | 1.0 $=$ |
| 30 | 11.0 $=$ | 10.9 $=$ |

For every problem instance, we know the run-time of algorithm A and algorithm B. A common assumption for hypothesis testing is that the data (run-time) follows a normal distribution. In the context of this work, this assumption will not hold, as we compare run-times (and other properties) across different classes of problem instances, some more readily solved than others. As a result, we get clusters of different run-times according to the hardness of the instances. Therefore, we need a statistical test that does not assume a normal distribution and can deal with paired data: the Wilcoxon Signed-Rank test [179]. Information on how this test works exactly is readily available [100], so we will not go into detail here. It is just important to note that the null hypothesis for the Wilcoxon Signed-Rank test is that the difference between the run-times of A and B is centered at a value $\geq 0$ and the alternative that it is centered at a value $< 0$. The main point here is that this is different from saying that the arithmetic mean (which we usually report) of the run-times of A is smaller than the mean of the run-times of B. In rare cases, this will become visible, as a mean run-time of one algorithm might be marked as the lowest run-time, even though other algorithms have lower reported mean run-times.

On the topic of presentation of the results of the statistical tests, Table 2.1 shows an example. The average run-time in seconds of algorithms A and B are compared for two different instance sizes. For size 20, we can see that B requires only 1 second, while A requires 5.1 seconds. A Wilcoxon Signed-Rank Test has shown that $A-B$ is $> 0$ (with a certain error probability), hence the "$>$"-sign next to the reported run-time of A. B is equal to itself (we always compare to the best algorithm) and therefore marked with an "="-sign. The situation is not as clear for size 30. Again, algorithm B is faster, but now the difference between A and B is not significantly greater than 0, therefore A is also marked with an "=" sign and shaded, but slightly less so than B.

In this thesis, we always perform the Wilcoxon Signed-Rank test with a level of significance of 5%, which is the probability of mistakenly rejecting the null hypothesis. We used the Wilcoxon test available in the statistics software R [86]. For an introduction to statistical hypothesis testing, see for example [24, 150, 169].

### 2.4.3 Used Software

This thesis would not have been possible without utilizing the work of others. In this section we want to acknowledge the authors of the libraries and software packages which we used. As programming environment, we used Eclipse Kepler [59, 81] with the C++ Development Tooling [138]. The employed compiler was gcc-4.7.1 [65]. Considerable support was provided by the various libraries of the boost project [17, 102], especially the graph library [158]. Other libraries and applications we used to handle graphs were Stanford GraphBase [107, 108], GT-ITM [184] and nem [120]. For statistical computations we used R [86].

As solver for (Integer) Linear Programs we used CPLEX 12.5 [85]. Unless otherwise specified, we used default settings, with the exception that we solved in single-threaded mode and that the time-limit was specified in CPU time and not wall-clock time. We used GECODE [155, 167] as Constraint Programming solver.

CHAPTER 3

# The Virtual Network Mapping Problem

## 3.1 Introduction

In Chapter 1, we have outlined the need for virtual networks within physical networks, but we did not go into detail on how this works exactly. The main aim is clear: we want to fit virtual networks into physical networks. Even this simple statement is ambiguous. It could mean that we want to find a part of the physical network that has the same structure as a virtual network, so the virtual network will fit there. Another possibility would be that we have to recreate the structure of a virtual network by using parts of the physical network. Additionally, resources should be involved in some way, since it is not possible to implement an unlimited amount of virtual networks within a physical network. This demonstrates that a rigorous definition of the problem we are going to solve in the following chapters, the Virtual Network Mapping Problem (VNMP), is required. In this chapter, we will present the definition of the VNMP. Chapter 4 will outline other definition possibilities that can be found in the literature.

## 3.2 Input of the VNMP

Two central components of the input of the VNMP are the physical network and the virtual networks. Another name for the physical network that we will use throughout this thesis is substrate network. The substrate network is modeled by a directed graph $G(V, A)$ with node set $V$ (the substrate nodes) and arc set $A$ (the substrate arcs). The virtual networks, also called slices, will be defined by a directed graph $G'(V', A')$ with node set $V'$ (the virtual nodes) and arc set $A'$ (the virtual arcs). Each separate connected component of $G'$ represents one virtual network. Based on these definitions, we will use the terms network and graph interchangeably. We chose directed graphs to model the physical and virtual networks because this allows us to distinguish between communication directions. As a result, we can model asymmetric con-

nections, i.e., the connections that have different properties in different directions, which often occurs in practice.

Virtual networks are meant to represent a specific application or application class. This implies that they have to connect the users of the application which are located at fixed points in the substrate network. As a result, we cannot place all the virtual nodes to which the users want to connect arbitrarily within the substrate network. They need to be close to the users. There are other reasons why the ability to limit the location of virtual nodes is important. One example would be that the operator of a virtual network for video streaming wants to ensure that each country in which he operates is covered by a virtual node the users can connect to. There might also be legal reasons why some locations are undesirable. Therefore, the third major input required to define an instance of the VNMP is a set $M \subseteq V' \times V$, which specifies allowed mappings. If and only if $(k, i) \in M$, virtual node $k$ is allowed to be located at substrate node $i$. By $M(k)$, we denote the set of substrate nodes where virtual node $k$ might be placed. Obviously, $M(k) \neq \emptyset,\ \forall k \in V'$ for a valid VNMP instance.

It is not physically possible to map an unlimited number of virtual networks into a substrate network, so we need to consider relevant resources which are used up by the virtual networks. First and foremost, virtual networks transfer data, so they require data transfer capacities, or bandwidth, from the substrate network. To be able to model this, we associate a bandwidth requirement $b_f \in \mathbb{N}^+$ with all $f \in A'$ and bandwidth capacities $b_e \in \mathbb{N}^+$ with all $e \in A$. With data transfer capacities, we are able to represent applications (i.e., virtual networks) that are just concerned with bulk data transfer. There are applications with this characteristic (e.g., Peer-to-Peer networks), but there are also many applications that require more than guaranteed transfer capacities. One might imagine a telephone conversation, where the time lag between the speaking of the caller and the listener being able to hear it is too high. Such situations are experienced as irritating by many people. Therefore, the time it takes to transfer data is a more important characteristic than the transfer capacity for Voice-over-IP applications. We take transmission delays into account by specifying an upper bound for delay $d_f \in \mathbb{N}^+$ for all $f \in A'$ and associating a transmission delay $d_e \in \mathbb{N}^+$ to all $e \in A$.

A virtual network is not simply a passive container for a specific application. It offers services to the application, be it in the form of customized routing protocols, naming services, or something else. To be able to do that, the virtual nodes require computing capacities that have to be available at the substrate node where they are hosted. So we associate CPU requirements $c_k \in \mathbb{N}^+$ to all $k \in V'$ and CPU capacities $c_i \in \mathbb{N}^+$ to all $i \in V$. The CPU capacities of the substrate nodes are not just used to host virtual nodes. In a physical network, the substrate nodes are typically represented by routers, and their main task is forwarding data. This data forwarding requires processing power, which is then unavailable for hosting virtual nodes (or vice versa). This interaction is approximately modeled by the VNMP by assuming that one unit of data (as a slight misuse of terminology, we will also call this one unit of bandwidth) from a virtual arc that crosses a substrate node requires one unit of processing capacity there. It is inconsequential whether this bandwidth is simply relayed by the substrate node or has originated from a virtual node mapped to the substrate node. Even if both, the sending and receiving virtual node are mapped to the same substrate node, CPU capacity is required to route data from one virtual node to the other.

These are all the inputs that are required for a very basic form of the VNMP, when we are only interested in finding valid solutions (the exact definition of a valid solution will be discussed later). For richer forms of the VNMP, we require additional inputs. We want to map virtual networks into a physical network in a cost-optimal way. Again, we have to refer to a later part of this chapter for the definition of the objective. For now it is sufficient to state that each substrate node $i$ has an associated usage cost $p_i^V \in \mathbb{N}^+$ and every substrate arc $e$ has a cost $p_e^A \in \mathbb{N}^+$. If the resources within the substrate network are not sufficient, it might be possible to buy additional resources. For this, we have a defined price of one unit of CPU capacity $p^{\text{CPU}} \in \mathbb{N}^+$ and one unit of bandwidth $p^{\text{BW}} \in \mathbb{N}^+$.
This concludes the discussion of the input necessary to define a VNMP instance. In the next section, we will define the output of the VNMP.

## 3.3 Output of the VNMP

The output of a VNMP consists of two parts. First, a mapping $m : V' \to V$ such that $(k, m(k)) \in M,\ \forall k \in V'$. That means every virtual node has to be assigned to one of its allowed substrate nodes. The second part is the implementation of the virtual arcs. For every virtual arc $f$, we need to find a simple path $p_f$ from $m(s(f))$ to $m(t(f))$ in the substrate. Such a solution is only valid, if the following resource constraints are fulfilled. First of all, there is the delay constraint. The sum of the delays of the substrate arcs that are used to implement a virtual arc $f$ may not exceed the maximum allowed delay $d_f$. Then there is the bandwidth constraint. The data transferred across a substrate arc $e$ by the implementations of virtual arcs has to be less than or equal to the bandwidth capacity $b_e$. As last resource constraint, we have the CPU capacity at the substrate nodes. The total CPU load caused by hosting virtual nodes and by routing data for virtual arcs is not allowed to be greater than the CPU capacity $c_i$. If a solution to the VNMP satisfies those constraints, it is valid.
One thing to note for this definition of output is that we require the implementing paths for virtual arcs to be simple, i.e., to not contain loops. This is not strictly necessary, as long as the substrate has enough resources and the path does not exceed the delay limit, implementing paths with loops would not be a problem. However, given a valid VNMP solution (except the simple path requirement), we can always derive a solution that only contains simple paths by removing all loops. The result will be a valid solution and require strictly less resources, so it will be better in this sense. The simple path requirement also adds optimization potential, as outlined in Chapter 9, where we will heavily depend on the simpleness of implementing paths.

## 3.4 Example VNMP Instance

Before we define the objective for the VNMP, we will show a small example to demonstrate the main components of the VNMP.
Figure 3.1 shows a simple VNMP instance. The virtual network $G'$ contains two virtual nodes, showing their CPU requirement. A virtual arc connects them and is labeled with its bandwidth requirement and allowed delay. The substrate network $G$ contains the physical network nodes

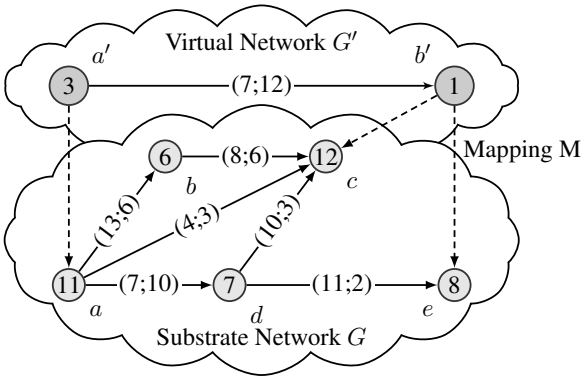



Figure 3.1: An illustrative VNMP instance.

showing their CPU resources and the available links (i.e., connections) between the nodes, labeled by their bandwidth capacity and the delay that is incurred when data is transmitted across them. The dashed lines show $M$, i.e., the allowed locations of the virtual nodes. This example instance has only one valid solution, as $b'$ cannot be mapped to $c$, even though $c$ has enough resources available. At least eight units of processing capacity are required at $c$, one for hosting $b'$ and seven for routing the data of the virtual arc to $b'$, so the 12 available units are sufficient. The problem is that the virtual connection cannot be implemented from $a$ to $c$. The implementing path cannot use $b$, because it does not have enough resources to route seven units of bandwidth. The direct connection from $a$ to $c$ lacks the required bandwidth capacity, and the path $(a, d, c)$ incurs too much delay. So the only valid solution is to map $a'$ to $a$, $b'$ to $e$ and use the path $(a, d, e)$ to implement the virtual arc between $a'$ and $b'$.

## 3.5 Objective

Until now, we have defined how a valid solution to the VNMP looks like and which constraints it has to fulfill. However, when considering practical applications it is clear that not all valid solutions are equally desirable. We have already touched on this previously during the discussion of the simple path requirement, where we noted that wasting resources is bad. What we now need is a measure of desirability of a valid solution and there are a lot of possibilities for defining it. We could say that it is desirable to spread the load on the substrate nodes so that the CPU requirement at each node is as far away from the available capacity as possible. Another possibility would be to minimize the length of the implementing paths (measured by the number of used arcs), because that reduces the total bandwidth load on the substrate and the probability that a defect of a substrate arc will break a virtual connection. Or we could minimize the length of the implementing paths (measured by the incurred delay), which would make a solution more resilient if the requirements of the virtual networks or the characteristics of the substrate change. We chose to define a model of the operational costs of a VNMP solution, which we try to minimize. In this model, the resources of the substrate network are available to a virtual network

provider. If it does not access these resources, they do not incur any costs. In case it utilizes the virtualization capabilities of a substrate node $i$, by hosting any amount of virtual nodes there, it has to pay the usage cost $p_i^V$ for this substrate node. For substrate arcs, the situation is the same. If any number of virtual arcs utilize a substrate arc $e$, usage cost $p_e^A$ has to be paid. The sum total of the incurred costs gives the substrate usage cost $C_u$, which has to be minimized. Therefore, the objective is to fit all virtual networks into a cheap part of the substrate network, subject to the resource and mapping constraints.

This choice of objective function has some implications. First and foremost, there is no immediate connection between a solution and its cost. By paying the substrate usage cost, we select a part of the substrate that we are allowed to use to implement virtual networks. How they are implemented within this selected part does not influence the costs. As a result, it can be expected that a VNMP instance will have a lot of different optimal solutions. The weak link between a solution and its cost may be problematic for heuristic methods. In Chapter 11 we will show problems this objective causes for Integer Linear Programming approaches. Another property of the objective function is that it provides a force away from feasibility. To create a very good solution, we need to aim for high utilization of the parts of the substrate we pay for. However, if the utilization is too high, we might fail to find a valid solution at all. To increase the probability of finding a valid solution, we need to spread the utilization of substrate nodes and arcs, which will result in a solution with very high $C_u$.

For practical purposes, just using this objective is insufficient. It might happen that the substrate network does not contain enough resources to implement all virtual networks. If we try to solve such a VNMP instance with heuristic methods, we will just fail to find a solution. If we apply exact methods, we might be able to prove that no solution exists. This is an important piece of information, but useless for a virtual network provider as it does not give any hints as how to solve this problem. There are two ways to deal with insufficient resources: reduce the amount of resources required or add more resources. To reduce the amount of required resources, there are multiple possible approaches. We could assign penalties to virtual networks that do not get the requested resources and try to minimize the penalties. Another possibility would be to assign profits to virtual networks and try to select a subset of virtual networks that still fits into the substrate network and maximizes the profits.

In this work, we choose the second way to deal with insufficient resources: we add the possibility of increasing the amount of available resources. This is more constructive than simply rejecting or not satisfying customers and also possible in practice. A virtual network provider might be able to rent additional resources for the controlled substrate network. As a first step, we need to identify where the resources are missing. In the general case, the location and amount of missing resources is ambiguous. It might be possible to make a VNMP instance solvable by adding CPU capacity at one node, or increasing the bandwidth somewhere else in the substrate, or a combination of both. In absence of any other constraints, we assume that we want to minimize the cost of renting additional resources, that one additional unit of resource has a fixed cost, and that the amount of resources that can be bought is unlimited. One unit of CPU resources costs $p^{\text{CPU}}$ and one unit of bandwidth $p^{\text{BW}}$. The total cost of additional resources is the additional resource cost $C_a$, which we want to minimize.

We have neglected to mention delay as resource that might be changed. This has three reasons, the first of which is the ease of change in practice. Changing the available CPU resources is simple, just add another server. Also increasing the available bandwidth is not problematic, just lease more bandwidth or activate additional fiber-optic connections if available. Changing the delay of a connection is far more complicated and usually means changing the employed mode of transmission, for instance from copper wire to fiber-optic cable. Also, the amount of CPU or bandwidth resources to be added can be controlled much better. For delay there basically is only the choice whether we want to have the delay of an electrical connection or of an optical one. The second reason why we neglect to discuss delay changes is that this is problematic to model with Integer Linear Programming approaches (see Chapter 11). For adding CPU and bandwidth, only a new variable per substrate node or arc is required to specify the amount of added resources. For delay changes, we would need to add new variables for every combination of virtual arc and substrate arc, which is a substantial amount of additional variables. The third reason for ignoring delay as a changeable resource is that changing only CPU or bandwidth capacities is sufficient to make a VNMP instance feasible, as long as there is a mapping for the source and target node of every virtual arc such that the delay bound is not exceeded. All these reasons notwithstanding, in Chapter 12 we will present an application where we also consider changes to delays.
A VNMP solution has two properties, its substrate usage cost $C_u$ and its additional resource cost $C_a$. This allows us to define the following:

**Definition 3.5.1** (Valid VNMP Solution)**.** *A VNMP solution is valid, if* $C_a = 0$.

**Definition 3.5.2** (The VNMP Satisfiability Problem VNMP-S)**.** *Given a VNMP instance, find a valid solution.*

**Definition 3.5.3** (The VNMP Optimization Problem VNMP-O)**.** *Given a VNMP instance, find a valid solution with minimal* $C_u$*. If no such solution exists, find a solution with minimal* $C_a$*.*

In this work, our main aim is solving the VNMP-O. In some cases we will also consider the VNMP-S. A slight variation of VNMP-S is looking for a solution with minimal $C_a$, which we will do for example in Chapter 11. For some applications, we do not allow the possibility of adding additional resources. It will be clear from the context when this is the case. For a compact mathematical definition of the VNMP, see Section 11.2.

## 3.6 Complexity

VNMP-S is $\mathcal{NP}$-complete. Quite obviously, it cannot be harder than $\mathcal{NP}$ since we can guess a solution and check if it satisfies all constraints in polynomial time. To show that it is $\mathcal{NP}$-hard, there are different possibilities for reducing other $\mathcal{NP}$-hard problems to VNMP-S. One way is shown in [5], but there are also simpler reductions, for which we will present the general ideas. Bin-packing can be reduced to VNMP-S by using virtual nodes with their CPU requirements as items to pack, substrate nodes as bins and allowing every mapping from virtual nodes to substrate nodes (i.e., every item can be packed in every bin).

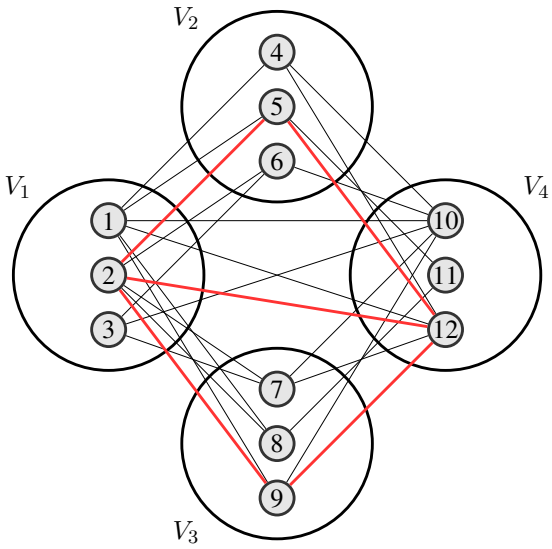



Figure 3.2: VNMP-S when just considering mapping and delay constraints.

A similar reduction can be performed by using the bandwidth requirements of virtual arcs as items to pack. Every item is a virtual network with two nodes and a virtual arc between those nodes. The bandwidth requirement of the virtual arc is the size of the item. The substrate contains one main source and one main target node. The source nodes of all virtual arcs are only allowed to be mapped to the main source node and the target nodes of the virtual arcs only to the main target node in the substrate. In addition, the substrate contains a node for every bin (the bin-nodes). Arcs from the main source node to all bin-nodes and from the bin-nodes to the main target node have the bandwidth capacity of the bin that is represented by the bin-node.
We have seen that by considering only CPU or bandwidth resources, $\mathcal{NP}$-hard problems can be reduced to VNMP-S. The same is possible for the delay constraint. Consider a relaxed version of the VNMP-S where only the delay and mapping restrictions are relevant. By calculating the shortest possible delay of paths between all pairs of substrate nodes, we know for each virtual arc $f$ which mapping combinations of $s(f)$ and $t(f)$ are allowed. Those are the combinations for which the delay between $m(s(f))$ and $m(t(f))$ is $\leq d_f$. Once we have identified the allowed combinations, we need not consider the delay constraint any further.
Figure 3.2 shows how the remaining problem looks like. In this example, we have to implement a virtual network containing four nodes ($V_1$–$V_4$). Each of those nodes can be mapped to three substrate nodes, for example $V_1$ can be mapped to substrate nodes 1–3. The edges present in the figure denote substrate connections that are feasible with respect to the delay constraint. For example, if $m(V_1) = 1$ and $m(V_2) = 4$, then there exists a delay feasible path for virtual arcs between $V_1$ and $V_2$. Note the use of the plural here as there might be a virtual arc from $V_1$ to $V_2$, from $V_2$ to $V_1$, or both. An edge is only present if a delay feasible path exists for all virtual arcs. What we need to do now is to choose a substrate node for each virtual node, such that edges for all virtual arcs can be chosen. Mapping $V_1$ to node 1 and $V_2$ to node 6 would not be valid for

example, since there is no edge between those nodes. We have chosen $m(V_1) = 2$, $m(V_2) = 5$, $m(V_3) = 9$, and $m(V_4) = 12$. The selected edges are colored and also show the structure of the virtual network (i.e., which virtual nodes have to be connected).

Given a clustered graph ($V_1$–$V_4$ are clusters) as shown in Figure 3.2, it is straight forward to construct a VNMP instance corresponding to it. The VNMP instance contains one virtual network, with one node for each cluster. If the graph contains at least one edge between clusters, then there is a corresponding arc between the virtual nodes. The orientation of the arc is arbitrary. The substrate network is given by the nodes within the clusters and the connections between them. The orientation of the substrate arcs has to be compatible to the corresponding virtual arc. For example, the VNMP instance derived from the graph shown in Figure 3.2 contains a virtual node for $V_1$ and a virtual node for $V_2$. Since there are edges between those clusters, there is also a virtual arc between the virtual nodes. We orient it arbitrarily from $V_1$ (more precisely the virtual node representing $V_1$) to $V_2$. In the substrate network we have nodes 1 and 4 (among others) and an arc between them. They represent a connection between $V_1$ and $V_2$. Since we chose an orientation from $V_1$ to $V_2$, also the substrate arc has to go from node 1 to node 4 in order to be compatible. The mapping constraint we set according to the clusters. All virtual arcs have a delay requirement of 1 and all substrate arcs have a delay of 1.

There is a transformation of the $\mathcal{NP}$-hard graph coloring problem [60] to a clustered graph as shown in Figure 3.2. This transformation was presented in [83, 114], we will just give the main idea here. For the graph coloring problem [60], we are given an undirected graph and a set of colors. To each node, we need to assign a color such that no pair of adjacent nodes is colored the same. For each node in the input graph, we create a cluster of nodes, one node representing a color the original node is allowed to have. For every arc in the input graph, we connect all nodes from the cluster of its source node to all compatible nodes of the cluster of its target node. Nodes are compatible if the colors they represent are different. This creates a structure as shown in Figure 3.2, which in turn can be transformed to a VNMP-S instance (only considering mapping and delay constraints). Thus VNMP-S is $\mathcal{NP}$-hard and the $\mathcal{NP}$-completeness of VNMP-O follows accordingly.

The $\mathcal{NP}$-completeness of VNMP-O is another reason why we defined additional resource costs $C_a$. We cannot expect that heuristic methods will always be able to find a valid solution to a VNMP instance if such a solution exists. Therefore, we need a way to guide the search process towards valid solutions, which is achieved by trying to minimize $C_a$. Another complication with respect to the delay constraints concerns the creation of a (not necessarily valid) solution to a VNMP instance. Previously, we have stated that just adding CPU and bandwidth resources is enough to ensure the existence of a feasible solution to a VNMP instance *if there is a mapping of the virtual nodes such that the delay constraint of all virtual arcs can be satisfied*. If we want to construct a feasible solution, we need to check if such a mapping exists and then use it to define the solution. Now we have shown that finding the mapping is $\mathcal{NP}$-complete, so we actually missed our goal that by allowing to buy additional resources, finding a feasible solution to an arbitrary VNMP instance will be easy (i.e., polynomially solvable). We circumvent this problem by using VNMP instances that allow delay feasible implementing paths for every mapping configuration (see Chapter 5). Chapter 12 contains techniques for dealing with delay changes, so that we do not need to restrict the structure of VNMP instances. Also note that using VNMP

instances that guarantee delay feasible paths for any mapping does not mean that the graph coloring aspect of VNMP can be ignored. It is very much relevant for VNMP-S and VNMP-O, since we want to find a solution with $C_a = 0$, which means that some delay feasible paths may not be allowed anymore due to bandwidth or CPU constraints.
There is one further problem caused by delay constraints, which occurs when trying to find implementing paths. When trying to find implementing paths for virtual arcs, it is easy to find a path that is valid with respect to the delay constraint by just using a shortest path calculation (with the delay used as distance). However, we are usually not interested in the shortest path, we want to find a good path that satisfies the delay constraint, for some definition of "good". If we want to find implementing paths such that the final VNMP solution is valid, one approach could be to use arcs where only few bandwidth resources have been used. Trying to find a path with the least amount of used resources on the arcs that satisfies the delay constraint is a Resource Constrained Shortest Path Problem, which is $\mathcal{NP}$-complete [125]. So even when we apply heuristics for finding solutions to a VNMP instance, we have to solve $\mathcal{NP}$-complete problems. Luckily, the Resource Constrained Shortest Path Problem can be solved in pseudo-polynomial time by Dynamic Programming approaches, for example the one presented in [69], if the arc costs are non-negative.

## 3.7 Ranking

For a solution to a VNMP instance we have defined two properties: its substrate usage cost $C_u$ and its additional resource cost $C_a$. When solving VNMP-S, there are no complications. Depending on the variant, we either set $C_a = 0$ and try to find a valid solution, or we try to minimize $C_a$. Given different solutions, we also need to be able to determine which is the best. Again, this is straight forward for VNMP-S as the solution with lowest $C_a$ is the best. In addition, the difference in $C_a$ also gives us a measure of how much a solution is better than another one.
For VNMP-O, the situation is more complicated, since we also care about $C_u$. In effect, we have a lexicographic objective measure. When comparing two solutions, the solution with lower $C_a$ is better. Only if those costs are equal, $C_u$ decides which solution is better. For one particular instance, this still allows to decide which solution (and therefore the algorithm that created it) is better. The problem arises when we want to analyze the average performance across multiple instances. In the case of VNMP-S, we can just compare average $C_a$ values. For VNMP-O, comparing average $C_u$ values and declaring the algorithm with the lowest value the best is not a valid approach, since it is very easy to reduce $C_u$ by increasing $C_a$. Therefore, we need a representative value that we can compare. An often used approach is to calculate this value by $KC_a + C_u$, with $K$ greater than the maximum usage cost. This would work and also give a measure of how much an algorithm is better than another one. The main downside with this approach is that it places a huge emphasis on valid solutions, more than we deem reasonable. Assume we have two algorithms for VNMP-O to compare and ten different instances. One algorithm ignores $C_u$ and creates ten valid solutions which utilize the complete substrate (and therefore incur the maximum $C_u$). The other algorithm finds very good (i.e., valid with low $C_u$) solutions for nine instances but fails to find a valid solution for one instance. The first

algorithm would seem to be better according to the representative value, even though there are good reasons to prefer the second one in practice.
Because of this, we chose to use another approach: ranking. Given different algorithms to compare and a set of VNMP instances, we can order the algorithms, for each instance separately, according to the solutions they achieve. Based on this order, we assign ranks. The algorithm with the best solution gets rank zero, the second best rank one and so on. If two algorithms create the same result, they share the same rank. Now every algorithm has a rank $R$ for each VNMP instance. Average ranks cannot be meaningfully compared because the total number of ranks is (potentially) different for every instance. Achieving rank one out of two ranks means that this algorithm achieved the worst result, but achieving rank one out of ten means that this algorithm achieved nearly the best result. So we need to normalize the rank by dividing it by the highest assigned rank. The created value is the relative rank $R_{\text{rel}}$. If $R_{\text{rel}} = 0$, the algorithm achieved the best result, $R_{\text{rel}} = 1$ means the worst result. If all algorithms created the same solution (which would cause a division by zero), we assign a relative rank of zero to all algorithms. Average relative ranks across multiple instances can be compared in a meaningful way. If an algorithm achieves an average relative rank of 0.2, that means its results are among the top 20% on average. The relative rank $R_{\text{rel}}$ will be the main metric for comparing results in this work.
The relative rank is not without its problems. The first problem is that we lose all information about how much better one solution is compared to another. Assume we have two algorithms (A and B) and two VNMP instances. Both algorithms create valid solutions for both instances. For the first instance, A's solution has half the substrate usage cost of B's. For the second instance, B's solution costs one monetary unit less than A's. Both algorithms have an average relative rank of 0.5, but clearly algorithm A should be preferred. The second problem is that the relative rank of an algorithm depends on the algorithms that are being compared. Therefore, Appendix A contains the full results of all algorithms that are going to be presented in the following chapters to allow comparison with future VNMP solution methods.

## 3.8 Extensions

The model of Virtual Network Mapping as presented in this chapter is greatly simplified. In this section, we are going to outline some possibilities for refining the model to better match it to the real world.
One extension could be to model virtualization overhead, i.e., depending on the number of virtual nodes mapped to a substrate node we have an additional demand for CPU resources caused by the overhead of managing the different virtual nodes. Also the delay behaviour can be improved. As a first step, instead of assuming a constant delay on an arc, we could use a normally distributed delay. Instead of a hard upper bound, virtual arcs could specify a delay limit and the maximum allowed probability of exceeding this delay. However, this is still a very limited model. The delay of network connection depends on its utilization, so a relation between used bandwidth and delay of a connection could be added. However, the relationship between delay and utilization is highly non-linear [135] and incurred delay is actually mostly a property of the routing nodes and not the links between them. Even the used hypervisor influences the delays [177]. Whether a more realistic delay model generates tangible benefits in practice remains

to be seen. There is research [188] that suggests that the higher statistical moments (skewness, kurtosis, ...) are also important to characterize the behaviour of a link with respect to delay.
For a valid VNMP solution, we require that we have exactly one implementing path for each virtual arc. It is possible to relax this constraint and allow an arbitrary number of paths. Assuming we already have fixed a particular mapping and relaxing the delay constraint, finding valid paths for the virtual arcs subject to the bandwidth and CPU constraints is a Multi-Commodity Flow Problem with fractional flows. This problem can be solved in polynomial time by Integer Programming or if fast solutions are required by approximation schemes [101]. Allowing multiple paths was also considered for example in [183]. For this work, we chose the restriction to a single path due to practical reasons. If we allow multiple paths, then the observed behaviour of the virtual connection will be much more erratic in terms of delay and probably harder to deal with by the employed protocols in the virtual network.
One additional enhancement to the model could be to allow redistribution of bandwidth between the directions of a connection between two substrate nodes. That would capture the flexibility inherent especially in fiber optic connections. Different wavelengths can be configured to be either used in sending or receiving direction. In this case it would also be possible to simplify the substrate network to an undirected graph where edges have a specific bandwidth capacity that can be used in arbitrary directions. As a downside, we would lose the ability to model cases when the distribution between sending and receiving capacity is fixed, for instance if the substrate resources are only rented.
We have now outlined some possibilities of modeling Virtual Network Mapping closer to the real world by including behaviours that can be observed in practice. However, we neglected an important point: Virtual Network Mapping is not a static process. For the VNMP we assume that we know all virtual networks and the available substrate network in advance. In practice, virtual network mapping will be very dynamic, new virtual networks will have to be added to the substrate network while other virtual networks are removed. Already present virtual networks might need to be reconfigured because their resource requirements have changed. Of course, also the substrate network is subject to modifications due to failures, maintenance, or because resources can be rented somewhere else much cheaper and we want to move the virtual networks there.
With all those sources of uncertainty, it might be useful to consider finding solutions that are robust when the environment changes [11, 14]. For instance, we could try to find a selection of resources in the substrate, which allows for a valid solution even if an arbitrary node or arc of the selection fails. Alternatively, we could try to find a selection that allows adding another virtual network (at least with high probability) without having to use additional parts of the substrate.

## 3.9 Summary

In this chapter, we have presented the definition of the Virtual Network Mapping Problem, VNMP. Virtual networks require CPU resources on the nodes, bandwidth resources on the arcs and have a maximum allowed data transfer delay. The physically available network (the substrate) offers CPU resources on the nodes, the arcs have data transfer capacities and delay is incurred when data is transferred across them. In addition, virtual nodes can only be hosted on a

Table 3.1: Input of the Virtual Network Mapping Problem.

| Input | | Description |
|---|---|---|
| $G(V, A)$ | | Substrate graph |
| $c_i \in \mathbb{N}^+$ | $\forall i \in V$ | Available CPU resources on substrate nodes |
| $b_e \in \mathbb{N}^+$ | $\forall e \in A$ | Available bandwidth on substrate arcs |
| $d_e \in \mathbb{N}^+$ | $\forall e \in A$ | Incurred delay on substrate arcs |
| $p_i^V \in \mathbb{N}^+$ | $\forall i \in V$ | Cost of using a substrate node |
| $p_e^A \in \mathbb{N}^+$ | $\forall e \in A$ | Cost of using a substrate arc |
| $p^{\text{CPU}} \in \mathbb{N}^+$ | | Cost of one additional unit of CPU capacity |
| $p^{\text{BW}} \in \mathbb{N}^+$ | | Cost of one additional unit of bandwidth capacity |
| $G'(V', A')$ | | Virtual network graph |
| $c_k \in \mathbb{N}^+$ | $\forall k \in V'$ | Required CPU resources of a virtual node |
| $b_f \in \mathbb{N}^+$ | $\forall f \in A'$ | Required bandwidth of a virtual arc |
| $d_f \in \mathbb{N}^+$ | $\forall f \in A'$ | Maximum allowed delay for the implementation of a virtual arc |
| $M$ | | Set of allowed mappings of virtual nodes to substrate nodes |

Table 3.2: Output of the Virtual Network Mapping Problem.

| Output | Description |
|---|---|
| $m : V' \to V$ | Mapping of each virtual node to a substrate node |
| $p_f, \forall f \in A'$ | Implementing path for each virtual arc |
| $C_u$ | Substrate usage cost |
| $C_a$ | Additional resource cost |

specific set of substrate nodes. The aim is to find a mapping of virtual nodes to substrate nodes and a simple path in the substrate for each virtual arc subject to the resource constraints such that the operational costs are minimized. Every substrate node that is used to host virtual nodes incurs costs, as does every substrate arc that is used to implement a virtual arc. The sum of those costs is the substrate usage cost $C_u$. Just finding a solution that satisfies all constraints is $\mathcal{NP}$-complete. Therefore, it is allowed to buy additional resources so that an arbitrary solution can be made to satisfy all resource constraints. The cost of the additional resources is the additional resource cost $C_a$. We call a VNMP solution valid if no additional resources are necessary ($C_a = 0$). By VNMP-S we denote the problem of finding a valid solution, VNMP-O is the problem of finding a valid solution with minimal operational costs. Table 3.1 gives a summary of the input for the VNMP, Table 3.2 summarizes the output of the VNMP.

CHAPTER 4

# Related Work

## 4.1 Introduction

In this chapter, we present an overview of the available literature on Virtual Network Mapping and relate it to our approach. Work on the VNMP can also be found under the names Virtual Network Assignment [187], Virtual Network Embedding [30], Virtual Network Resource Allocation [164], and Network Testbed Mapping [148]. The different names are a result of slightly different application scenarios. Network Testbed Mapping for instance deals with the problem of embedding virtual networks into a network testbed to share it among different research groups and their experiments. Even though the core problem is always the same, i.e., mapping virtual networks into a physical network, there is a huge diversity in the details.
In Section 4.2, we will cover the basic setup of the related work in terms of employed network model, size, and structure of virtual and substrate networks. The types of resources that have been used will be discussed in Section 4.3. In Section 4.4, we consider the different possibilities with respect to the objective. The test methodology will be the main focus of Section 4.5 and Section 4.6 presents the different solution methods employed for solving the VNMP. We conclude in Section 4.7.

## 4.2 Network Models

In this work, we use directed graphs to represent substrate and virtual networks. From the sampled related work, only one [164] employs directed graphs. The vast majority [30, 48, 71, 82, 139, 145, 148, 175, 180, 185–187] uses undirected graphs. In between lies the work of Lu and Turner [117], who consider undirected substrate networks but directed virtual networks. Unfortunately, there is never a reason stated as to why directed graphs were rejected in favor of undirected graphs (or vice versa). We chose directed graphs, since they allow for asymmetric resource requests, which appears to be very meaningful in practice. For instance in some video streaming applications a virtual server node $a$ may send a virtual node $b$ a lot of data, while

$b$ sends only small amounts to $a$. Also, substrate connections may have different properties depending on direction, which cannot be modeled by undirected graphs. The one advantage of undirected graphs is that they can capture the situation when the Virtual Network Operator owns the substrate resources and is able for instance to split the capacity of a network link arbitrarily between the two directions. The directed model we employ could handle this with additional constraints, but we do not explore this possibility further.
The employed substrate networks are mostly synthetic, only [117, 164, 185] use substrate networks based on real topologies. Considering the sizes of the employed networks, they can be split roughly into three size classes: 10–50 nodes, 100 nodes and more than 100 nodes. Into the smallest class fall [164] (15 and 27 nodes), [117, 185] (20 nodes), [180] (40 nodes), and [30] (50 nodes). Note that every work using realistic substrate networks falls into this class. The representatives of the medium size class with 100 nodes are [48, 82, 145, 175, 187]. For the largest size, we have the work presented in [186] with 150-250 nodes and [148] with 120 and 525 nodes. The synthetic substrates are random graphs, where the probability of an edge (or arc) between two vertices was chosen either to be 50%, or less commonly 40%. As a result, the substrates are rather dense. For our work, we use substrates based on real topologies covering sizes from 20 to 1000 nodes.
The used virtual networks are usually very small synthetic random graphs with 50% chance of an edge between two nodes. The exception are the virtual networks used in [164], which are complete graphs with 2-4 nodes. Most work [30, 48, 175, 180, 185–187] uses random sizes between 2 and 10 nodes. The largest virtual networks we encountered were 2 to 20 nodes in [145], 25 nodes in [82] and 10 to 100 nodes in [148]. The only work considering realistic virtual network loads is [148]. In this case, realistic means that the topologies have been requested within a network testbed. Our work will use virtual networks between 5 and 30 nodes. In addition, they are not random graphs but rather try to capture different types of applications and their requirements with respect to topology and resources. Details will be presented in Chapter 5.

## 4.3 Resources and Constraints

Virtual networks have an impact on the substrate network. They require resources and only a finite number of virtual networks can fit into a substrate network. This is (nearly) universally acknowledged among the sampled work. The main differences lie in the considered resources.
The work presented in [187] does not use resources at all, but rather the count of virtual nodes mapped to a substrate node (node stress) and the count of virtual arcs crossing a substrate arc (link stress). The node and link stresses are not allowed to exceed a certain value. The remaining work at least considers the available bandwidth on substrate arcs, [71, 117, 164] exclusively use bandwidth as resource.
The next logical resource to consider is the CPU capacity available at the substrate nodes, which is consumed by the virtual nodes. The work presented in [30, 139, 145, 148, 175, 180, 185] considers CPU and bandwidth resources.
In addition to resource restrictions, limiting the mapping possibilities of virtual nodes is popular. One method to implement this is to assign a location to substrate nodes and virtual nodes. Virtual nodes may only be mapped to substrate nodes not too far distant. This approach is em-

ployed in [30, 117, 186]. Another way of limiting the mapping possibilities is to forbid that a substrate node hosts multiple virtual nodes of the same virtual network [48, 180, 187]. An in-depth discussion about the possibilities of restricting virtual node placement can be found in [180]. The authors mention location preference constraints, which means that a virtual node may only be mapped to a specific set of substrate nodes, location exclusion, which means that a virtual node is not allowed to be mapped somewhere and location separation, which means that two virtual nodes may not be mapped on the same substrate node or otherwise close together (e.g., on servers in the same rack).

Some less usual resources and restrictions are also covered in the literature. The power state (similar to the notion of a used substrate node in this work) is considered in [186]. Some nodes cannot host any virtual nodes in [164]. In [185], virtual nodes may be split up and mapped to multiple substrate nodes. This parallelization causes an additional overhead. The authors of [48] consider the available memory at the substrate nodes as additional resource. As a further extension, they consider the possibility of "overbooking" bandwidth resources on substrate arcs, since virtual arcs will not require their specified bandwidth constantly. A limit is placed on the probability that the bandwidth capacity of a substrate connection is exceeded. In [71, 117], the virtual networks are not even fixed. Given are data sending and receiving rates at the virtual nodes and the topology of the virtual network has to be created such that every traffic pattern conforming to these rates is possible. The authors of [71] also consider the restriction that the virtual network has to be a tree.

We have already mentioned in the previous chapter that an argument for allowing multiple paths to implement a single virtual arc can be made. This approach is used in [30, 164, 180, 183]. On the topic of the implementing paths, some work (e.g., [117, 145]) implicitly focuses on shortest paths, instead of allowing arbitrary implementations. For instance, the algorithm presented in [145] will reject a virtual network, if one of its virtual arcs cannot be implemented by using one of the $k \leq 6$ shortest paths. Longer paths which may have enough free resources are not considered. In tests with physical networks, it has been concluded that in 25% of cases the shortest path (with respect to hops) is not the shortest path with respect to transmission delay [171], so focusing only on shortest paths is really a restriction that might hurt the final solution quality.

The VNMP version considered in this work utilizes CPU, bandwidth and delay as main resources. From the sampled work, only the authors of [82] and [10] considered delay as factor for virtual network mapping. However, in the model proposed in [82], virtual networks cannot specify delay constraints and furthermore it is assumed that resources are unlimited. In [10], only the mapping of virtual nodes is considered, together with communication delays for the users of the virtual nodes. No virtual connections have to be implemented.

As location constraints we allow only a specific set of substrate nodes as host for each virtual node. We do not forbid the mapping of two virtual nodes from the same virtual network to the same substrate node, because if it is the cheapest way to implement a virtual network, why should it be forbidden. If two virtual nodes of a network are not allowed to be hosted on the same substrate node, this can be realized by choosing two disjoint sets of substrate nodes as possible hosts. Only if the sets of possible nodes cannot be disjoint, our employed method for specifying location constraints fails to capture this restriction. Restrictions based on distance between virtual node and substrate node can be realized by our model. For paths implementing

virtual arcs, we consider all paths that are short enough (with respect to delay), not only the shortest ones. We only use one resource on nodes, the CPU capacity. We chose not to consider other resources on the nodes, since they are basically the same as the CPU capacity and do not require any special handling. In fact, an additional resource on the nodes would be easier to handle than the CPU capacity in our model, since routing of data also requires the CPU. This causes a much tighter coupling between node mapping and arc implementing than in other work.

## 4.4 Objectives

The main objectives used for Virtual Network Mapping are either maximizing revenue or minimizing cost. The work published in [139, 145, 175, 185] focuses on maximizing revenue. Virtual networks with high resource demands generate high revenue.
The cost of a virtual network implementation is usually the sum of resources used in the substrate, which is used (with some variations) in [30, 71, 82, 148, 164, 180, 187].
We have already mentioned the work of Fajjari et al. [48], where it is allowed to exceed the bandwidth capacity of a substrate arc with a certain probability. Two objectives are used, maximizing the acceptance rate (fitting as many virtual networks as possible to maximize revenue) and minimizing the probability of exceeding the available capacity.
In [186], another interesting objective is used: minimizing the electricity cost. This is similar to the usage cost we use in this work (not using a substrate component means it can be shut down, thus saving on electricity other operational costs), but exceeds our approach by also considering the time dependence of the cost of electricity.

## 4.5 Testing Methodology

Most algorithms presented for the VNMP are tested in an online setting. That means that a substrate network is given and random virtual networks arrive over time that have to be mapped. These virtual networks also have a limited lifetime, after which they are removed from the substrate. As a representative of this methodology, we present the test setup as used in [175]. A single substrate network is used. Virtual networks arrive in a Poisson process with an average rate of 5 requests per 100 time units. Each request has an exponentially distributed lifetime with a mean of 1000 time units. The simulation is run for 50000 time units. The main performance metric is the acceptance ratio, i.e., how many of the arriving virtual networks can be mapped into the physical network. Also the work presented in [30,48,139,145,164,180,185,186] follows this evaluation style. It is important to note that the mapping algorithms only see one virtual network request at a time and cannot plan ahead or change how already accepted virtual networks are mapped to the substrate network. However, some methods allow to store a request for some time to wait for resources to become available.
In [187], the alternative of changing already mapped virtual networks is considered. This work also contains a discussion of the associated costs when changing a virtual network implementation (migrating virtual machines,...). However, also in this case the presented algorithm only considers one virtual network at a time. It could be shown that reconfiguring already present virtual networks leads to better performance.

Unfortunately, when this testing setup is used, the total number of virtual networks present in the substrate is not reported. Based on the available resources in the substrate networks and required resources by virtual networks, we estimate the total number of virtual networks to lie between 10 and 50. Another, in our opinion interesting, parameter that is usually not reported is the time needed to map a virtual network. In [82], a required time of 1 second was mentioned.
The work presented in [117, 148] uses an offline approach, where the total load of virtual networks is known from the start. Only the algorithm presented in [148] makes use of this information, the algorithm from [117] maps the virtual networks iteratively. A completely different type of evaluation is performed in [82], where the proposed distributed algorithm is simulated by means of a multi-agent system.
In our evaluation, we will focus on the offline case, which means we know which virtual networks need to be implemented. Every heuristic and exact approach we will introduce in the following chapters will deal with all virtual networks at once. In some sense, these methods can be viewed as complementary to the methods presented in the literature. While they concentrate on immediately answering virtual network requests, and by necessity have to focus on keeping the implementations of the virtual networks spread across the substrate so that critical resources remain available for future networks, our algorithms deal with the global view and can optimize the virtual network implementations to reduce operating costs. Instead of a single substrate network, we use 210 networks spanning 7 size classes and up to 40 virtual networks to evaluate the performance of our algorithms.

## 4.6 Solution Methods

In the previous section, we have shown that the approaches from the literature are meant to be employed in an online setting, which means they have to be able to decide nearly immediately if a virtual network is accepted or not. As a consequence, most of the presented methods are Construction Heuristics [48, 139, 145, 175, 185–187], with quite involved strategies for determining which virtual node or arc should be implemented next. The algorithm presented in [139] for instance uses Bayesian network analysis to improve the node selection strategy. Another interesting observation is that some methods, even though they only have to implement a single (usually small) virtual network, apply decomposition techniques to the virtual network and then concentrate on implementing the decomposed parts [48, 187].
The first step away from Construction Heuristics is the algorithm presented in [48], which in addition to greedy rules also utilizes backtracking to find good solutions. In [148], simulated annealing is utilized, in [117] Local Search.
The methods published in [30, 164, 180] are based on solving multi-commodity flow problems, which is possible since implementing virtual arcs with multiple paths is allowed in their version of Virtual Network Mapping. The algorithm from [164] uses a heuristic approach, while [180] uses Integer Linear Programming. The method presented in [30] uses the LP relaxation of the Virtual Network Mapping Problem to derive the mapping via rounding procedures and solves the multi-commodity flow problem afterwards. A distributed algorithm is presented in [82].
In this work, we will cover exact and heuristic approaches to solve the VNMP. In contrast to the work in the literature, we will focus less on execution speed and more on solution quality.

## 4.7 Conclusion

The VNMP as solved in the literature could be summarized as follows: Given is an synthetic undirected substrate network with 50 nodes and a capacity of about 30 virtual networks. From a continuous stream of synthetic virtual network requests with a size of 2 to 10 nodes, select those that can be implemented and implement them. The VNMP as solved in this work may be summarized as follows: Given are hundreds of realistic directed substrate networks from 20 to 1000 nodes in size and a capacity of at least 40 virtual networks. For each of those networks, implement at most 40 realistic virtual networks with a size between 5 and 30 nodes.
This already shows some areas in which we go beyond what is usually considered in the literature. We deal with a sizable set of substrates derived from networks occurring in the real world. Also the virtual networks are not synthetic but based on the requirements of different applications. Instead of CPU and bandwidth capacities, which are independent of each other, we consider an interaction between them. CPU capacity has to be used at substrate nodes to transfer data. In addition, we allow virtual arcs to specify delay constraints for their implementing paths. In contrast to the majority of the works in literature, we focus on the offline VNMP. This allows us to consider more powerful solution approaches, which would require too much run-time in an online setting.
Some interesting approaches from the literature go beyond what we will do in this work, for instance [186] with its complex operational costs model or [71, 117], where the virtual network structure has do be derived based on a set of traffic demands. We also do not consider the possibility of splitting up the workload of a virtual node as done in [185].

CHAPTER 5

# Towards a Realistic VNMP Benchmark Set

## 5.1 Introduction

In this chapter, we describe how we created the VNMP instance set we use to evaluate different solution methods. Our design goals during the development of the instance set were the following:

- Realistic substrate networks with a structure comparable to networks as they occur in the real world, i.e., no random or structured (e.g., Hub-and-Spoke) graphs.
- Wide range of substrate network sizes so that we have instances which can be solved by exact methods, but also instances for which heuristic methods are required.
- Virtual networks that cover different applications or application classes with a diverse set of requirements with respect to resources but also with respect to structure.
- Mapping restrictions that make sense in terms of the substrate network, i.e., not just random locations, but locations that cover specific parts of the substrate.
- Guaranteed existence of a valid solution.
- A way other than substrate network size to control the "hardness" of an instance. Put differently, a way to go from an instance where finding a valid solution is easy and optimality is hard to an instance where finding a valid solution is already a challenge.
- Any solution satisfying the mapping constraints can be made to satisfy the resource constraints by buying more CPU and bandwidth resources.

In the following sections we describe how we built the substrate and virtual networks to conform to these goals.

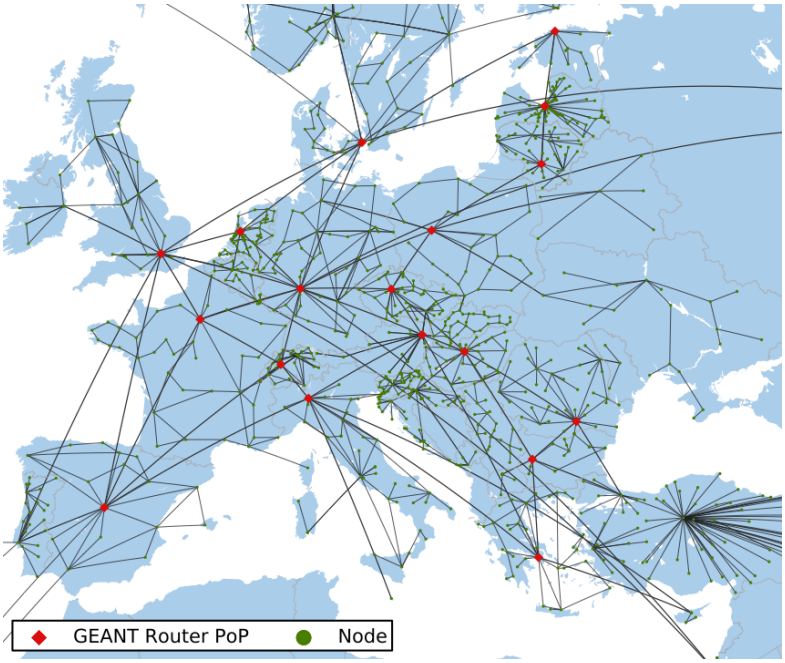



Figure 5.1: Topology map of national research networks and their interconnections by the GÉANT backbone network, as published in [106].

## 5.2 Substrate

Our main aim with respect to the substrate networks was that they have to have a realistic structure. Therefore, they need to be derived from real, measured topologies of the Internet, which are available in the literature. Candidates include the topology maps from the Rocketfuel project [163], the Scan-Lucient map (which contains the union of the topologies measured by the SCAN [70] and Lucent [25] Internet mapping projects) or the survivable network design library (SNDlib) [133]. In earlier work [88], we used the Rocketfuel and Scan-Lucient maps to derive substrate networks. The disadvantage with those topologies is that the Rocketfuel maps are rather small (about 100 nodes at most) and the Scan-Lucient map, while huge, is so sparse that most subgraphs are trees. The topologies of the SNDlib mostly do not exceed 50 nodes in size. Due to these reasons, we decided to look elsewhere for suitable substrate networks.

In the end, we decided to generate the substrate networks from a network of the Internet topology zoo [105], the European Research and Education Network GÉANT [106]. It contains national European research networks which are interconnected by the European Backbone Research Network. Figure 5.1 shows the structure of the complete network, the Points-of-Presence (PoPs) of the GÉANT network and their interconnects. For more details on how this topology was derived see [106].

This network proved to be very suitable for our needs. First of all, it contains 1157 nodes, which allows the creation of a wide range of substrate network sizes. Secondly, it associates with each node a geo-location (i.e., the county in which this node is located), which is very useful

for defining meaningful mapping constraints. The network also contains information about the capacity of different links. We did not utilize this information because it ranges across several orders of magnitude which is unsuitable for creating benchmark instances.
To derive substrate networks from the GÉANT map, we used the network manipulator nem-0.9.6 [120, 121]. Nem can extract sub-graphs of a given network that retain the main characteristics of the network, if the requested sub-graph size is small enough. If the target size exceeds 30% of the source network size, another approach is necessary. We selected a subset of the GÉANT map by choosing a random node from the map and then iteratively grew the selection by adding random nodes which are connected to the already selected part until the target size was reached. Since the GÉANT network is undirected, we added connections in both directions to the substrate graph. We created 30 substrates for each of the following sizes: 20, 30, 50, 100, 200, 500, and 1000 nodes. Up to (and including) size 200 nem was used. The costs $p_i^V$ and $p_e^A$ were assigned uniformly at random from $[1, 50]$ and the arc delays $d_e$ uniformly at random from $[5, 40]$. The other properties of the substrate (CPU and bandwidth capacity) are assigned based on the demand by the virtual networks that are added to the VNMP instance because we need to make sure that a valid solution exists. The detailed method will be discussed at a later point in this chapter.

## 5.3 Virtual Networks

Our main design goal for virtual networks was that they reflect different use-cases with a diverse set of resource demands. To fulfill this goal, we designed four different archetypes of virtual networks. After the creation of a substrate network as outlined in the previous section, we added ten instances of each of the four virtual network types to define a VNMP instance. The size of each virtual network was chosen uniformly at random from $[5, \min(30, 0.3 * |V|)]$. These bounds ensure that on one hand, the created virtual networks are not too small and on the other hand that they are significantly smaller than the substrate network into which they have to be mapped. Now follows a discussion of the four different virtual network archetypes.

### 5.3.1 Stream Network

This virtual network type models (video-)streaming services and has a random tree structure. The root node represents a streaming service which broadcasts its content consisting of multiple channels. Intermediate nodes in the distribution tree split this stream and forward only the channels that are actually watched by the customers connected to the leaf nodes of the tree. This is an example of an application that can directly benefit from custom routing protocols, as this splitting and forwarding of data is currently not possible within the network. A streaming service sends between 10 and 20 channels, each requiring between 2 and 10 units of bandwidth, which covers audio and video streaming. Every intermediate node forwards between 30% and 100% of received channels to each of its children, making sure each channel is forwarded at least once and each child receives at least one channel. The required bandwidth of each virtual arc is the product of transferred channels and the bandwidth requirement per channel. The required CPU power on each virtual node is three times the received bandwidth which models the required

processing power for unpacking and redistributing the stream. For the root node, the required CPU power is three times the sent bandwidth. Streaming applications are not delay constrained in our model, so we set the delay requirement of each virtual arc to the highest shortest path delay within the substrate network. As for the allowed mapping of the virtual nodes, we create groups of substrate nodes for the root and all leaf nodes. The node grouping procedure works as follows: Initially, substrate nodes are grouped according to their geo-location. If this creates more groups than required, some are randomly deleted so that the required number of groups is reached. Otherwise, we generate a new group by randomly moving half of the nodes of the largest group (rounding down in case of odd-sized groups) to this new group. New groups are generated until the required number of groups is reached. The location of the root node of the Stream virtual network is fixed to one of the nodes of its group. The leaf nodes can be mapped to any substrate node in their group. Intermediate nodes in the streaming tree can be mapped to any node in any group.

### 5.3.2 Web Network

This virtual network type models the typical web usage and has star structure. The center represents the web-server which serves the connected customers. Each customer receives between one and five units of bandwidth, the processing of which requires one unit of CPU power per received unit of bandwidth. The CPU requirement of the web-server is the sum of the transmitted bandwidth. As for the allowed mapping, the node grouping procedure is used to create mapping target candidates for each node of the virtual network. The virtual node of the web-server can only be mapped to one randomly selected substrate node of its group. All other nodes can be mapped to two randomly selected nodes of their group. As delay requirement for each arc (which connects the web-server node to one of the customer nodes) we set the smallest possible delay with which both possible locations of the customer node may be reached from the location of the web-server node. This models hard delay requirements, since users usually expect web-pages to load fast.

### 5.3.3 Peer-to-Peer Network

This virtual network type represents Peer-to-Peer (P2P) networks and is based on a directed small world graph [98, 149]. P2P applications do not have delay requirements (in our model), so the delays are set as they are for the Stream virtual networks. However, they have high bandwidth requirements; each arc requires between 10 and 70 units of bandwidth. The CPU requirements on each virtual node are twice the outgoing bandwidth which models compression or encryption services offered by this network type. As for the allowed mapping, the node grouping procedure is used to create groups for each of the nodes of the P2P network. Every virtual node is allowed to be mapped to any substrate node of its group.

### 5.3.4 Voice-over-IP Networks

This virtual network type models Voice-over-IP (VoIP) networks and is, like the P2P type, based on a directed small world graph. It requires 2 to 20 units of bandwidth for each virtual connection

and three times the outgoing bandwidth as CPU power on each virtual node, which models video compression and transcoding services. The virtual nodes can be mapped to any substrate node of their substrate node group created by the node grouping procedure. The delay requirement of a virtual connection $f$ for this virtual network type is set to the lowest possible value such that a delay constrained path between every location of $s(f)$ and $t(f)$ in the substrate exists.

To sum it all up, Stream networks have high bandwidth and CPU requirements but are barely delay constrained. Web networks on the other hand have very low bandwidth and CPU requirements but stringent delay constraints. P2P networks have high bandwidth and medium CPU requirements and lax delay constraints. VoIP networks have medium bandwidth and CPU requirements and are moderately delay constrained. P2P and VoIP networks also have a complex structure.

At this point, we have built a VNMP instance by creating a substrate network and assigning costs and delays to it. Then we added ten virtual networks for each of the four different network types together with their mapping constraints. To clarify, the 40 different networks are combined into one graph, the virtual network graph $G'$. The only missing component are the bandwidth and CPU capacities in the substrate network. We assign them by creating a random solution to the VNMP instance as it is now and use it as a guide to define the available resources.

As the first step for creating the guiding solution, we map each virtual node to one of its allowed locations in the substrate. Note that due to the chosen delay limits of the virtual networks, it is guaranteed that implementing paths satisfying the delay constraint exist for any mapping of the virtual nodes. To set the implementing paths of the solution, we could just use delay shortest paths. However, that would introduce too much structure into the solution, meaning that it could be found easily based on the final VNMP instance. Therefore, we construct implementing paths by solving a resource constrained shortest path problem. The delays within the substrate are used as resource, as lengths we set random values between one and ten for every substrate arc. The distance values are set differently for every path calculation to increase the randomness of the final solution. Based on the created solution, we assign CPU and bandwidth resources available in the substrate so that the solution is valid. Some nodes or arcs may not have been assigned resources. Substrate nodes get assigned the average amount of CPU resources (that have already been set) within their geo-location. If all nodes of a geo-location have not been assigned CPU resources, they are assigned the average amount of assigned CPU resources within the whole substrate network. The same process is applied to assign missing bandwidth resources. Arcs within a geo-location get assigned the average of the location, if no arc has been assigned resources the average bandwidth resources within the substrate are assigned. Arcs connecting different geo-locations get assigned the average bandwidth of arcs that have been assigned a bandwidth capacity and cross a geo-location. Note that it is not possible that no arc connecting different geo-locations has been assigned a bandwidth capacity based on the guiding random solution. Now all resources have been assigned. As a final step, we increase the CPU and bandwidth capacities at each node or arc separately by 20% to 50% so that there is more room for optimization.

One design goal is still unfulfilled: a way to control the "hardness" of a VNMP instance. We do this by modifying the number of virtual networks that have to be mapped, which we will denote by the load of an instance. The instances created by the outlined procedure have a load of 1, or

Table 5.1: Properties of the VNMP instances: average number of substrate nodes ($V$) and arcs ($A$), virtual nodes ($V'$) and arcs ($A'$), number of allowed mapping targets for each virtual node ($M_{V'}$) and the average total usage costs ($C_u$).

| **Size** | $\|\mathbf{V}\|$ | $\|\mathbf{A}\|$ | $\|\mathbf{V'}\|$ | $\|\mathbf{A'}\|$ | $\|\mathbf{M_{A'}}\|$ | $\mathbf{C_u}$ |
|---|---|---|---|---|---|---|
| 20 | 20 | 40.8 | 220.7 | 431.5 | 3.8 | 1536.0 |
| 30 | 30 | 65.8 | 276.9 | 629.0 | 4.9 | 2426.6 |
| 50 | 50 | 116.4 | 398.9 | 946.9 | 6.8 | 4298.1 |
| 100 | 100 | 233.4 | 704.6 | 1753.1 | 11.1 | 8539.1 |
| 200 | 200 | 490.2 | 691.5 | 1694.7 | 17.3 | 17584.2 |
| 500 | 500 | 1247.3 | 707.7 | 1732.5 | 30.2 | 44531.8 |
| 1000 | 1000 | 2528.6 | 700.2 | 1722.8 | 47.2 | 89958.4 |

full load. For lower loads, some virtual networks are removed. At load 0.9, one virtual network of each type has been removed. At load 0.1, the VNMP instance contains four virtual networks, one of each type. Based on the created 210 VNMP instances of full load (30 instances for seven different substrate network sizes), 1890 additional instances can be derived by reducing the virtual network load (0.1 to 0.9), yielding a total of 2100 VNMP instances. The created instances are available at [87].

## 5.4 Main VNMP Instance Properties

In this section, we show the main properties of the created VNMP (full load) instances, which are presented in Table 5.1. It can be seen that the created substrate graphs are very sparse. For instances of size 20 (which means that the substrate network contains 20 nodes), we observe an average of 40.8 substrate arcs. For a connected substrate, the lowest possible number of arcs is 38, which is twice the number of edges required to form a tree. This number is doubled since due to the construction of the substrate network, if two nodes are connected, they are connected in both directions. With rising instance size, the substrate networks also get marginally denser. The main point to note with respect to the virtual networks is that the number of virtual nodes to map and the number of virtual arcs to implement stays roughly constant from size 100 onward, because each virtual network has a size limit of 30 nodes. Even if the number of virtual nodes and arcs stays the same, with rising instance size we can expect that the required implementing paths for each virtual arc get longer, which complicates the problem of finding cheap and valid solutions. In addition, for larger instance sizes there are far more mapping possibilities for each virtual node. Table 5.1 also shows the complete substrate usage costs for reference, i.e., how much it would cost to use every part of the substrate network.

CHAPTER 6

# Construction Heuristics, Local Search, and Variable Neighborhood Descent

## 6.1 Introduction

In this chapter, we will introduce Construction Heuristics (Section 6.2), Local Search (Section 6.3), and Variable Neighborhood Descent algorithms (Section 6.4) for the VNMP. We present those algorithms together since they depend on each other and also share the property that they terminate by design, i.e., they are finished at some point and do not need to be aborted by a stopping criterion like the elapsed time. The heuristic algorithms we discuss in later chapters do not have this property. The deterministic termination also offers some great opportunities to analyze the tradeoff between required run-time and solution quality, which is more complicated if the run-time is an external parameter to the algorithm. The performance of the algorithms is compared in Section 6.5, Section 6.6 concludes and gives some directions for future work. The results presented in this chapter have been published in a more compact form in [91].

## 6.2 Construction Heuristics

A Construction Heuristic (CH) is used to create solutions to problems by following heuristic rules that guide the construction process towards feasible solutions of high quality. During each step, a partial solution is extended by the currently most promising component. For the VNMP, we can already see that these are conflicting objectives; guiding towards valid solutions means spreading resource usage across the whole substrate to decrease the probability of having to buy additional resources, which causes $C_u$ to be unnecessarily high. Trying to pack virtual networks densely to keep $C_u$ low will most likely lead to high $C_a$ as some substrate network components run out of resources, so some kind of balancing is required. To find the right balance, we first define a general outline of a CH for the VNMP. This outline defines sub-problems, which can

**Algorithm 6.1:** Outline of the Construction Heuristic for the VNMP

**Input** : A VNMP instance I
**Output**: A solution to I

```
1  Solution S(I);
2  if use node emphasis NE then
3      while not all virtual nodes mapped do
4          VirtualNode k=getVirtualNode(I,S); // virtual node selection SVN
5          SubstrateNode i=getMapTarget(I,S,k); // mapping target sel. IVN
6          S.setMapping(k,i);
7      end
8  end
9  while S incomplete do
10     while virtual arcs are implementable do
11         VirtualArc f=getVirtualArc(I,S); // virtual arc selection SVA
12         Path p=getImplementingPath(I,S,f); // path selection IVA
13         S.setPath(f,p);
14     end
15     if not all virtual nodes mapped then
16         VirtualNode k=getVirtualNode(I,S); // virtual node selection SVN
17         SubstrateNode i=getMapTarget(I,S,k); // mapping target sel. IVN
18         S.setMapping(k,i);
19     end
20 end
21 return S;
```

be solved by different heuristics. These heuristics can be tuned towards low $C_u$ or low $C_a$ and by selecting the right heuristics for the sub-problems, a CH for the VNMP can be derived that creates valid results with low $C_u$ with high probability.
Algorithm 6.1 shows the outline of the CH. It uses the solution to four sub-problems to build a VNMP solution. The four sub-problems are:

**SVN** Selecting a virtual node to map from the nodes that have not been mapped.

**IVN** Selecting an implementation of the virtual node, i.e., a substrate node to which the virtual node is mapped.

**SVA** Selecting a virtual arc $f$ to implement. This arc has to be implementable, which means that it is not implemented in the current solution and $s(f)$ and $t(f)$ have to be mapped. Otherwise we would not know, which substrate nodes the implementing path for $f$ has to connect.

**IVA** Selecting an implementation of the virtual arc, i.e., the path in the substrate network.

In addition to the heuristics used to solve those four sub-problems, there is another parameter which determines the behaviour of the CH. Basically, we can decide if we want to map all virtual nodes before we start to implement virtual arcs (node emphasis NE) or if we implement virtual arcs as soon as they are implementable (arc emphasis AE). We will call this the implementation emphasis. If NE is used, Algorithm 6.1 iteratively selects substrate nodes to implement (SVN) and a mapping target for them (IVN), until all virtual nodes are mapped. Only after that, the main loop of the CH is entered. This loop is executed until the solution is complete, that means every virtual node has been mapped and every virtual arc has an implementing path. In the main loop, we first implement all virtual arcs that are currently implementable by selecting such an arc (SVA) and finding a path for it (IVA). Once no more implementable arcs remain, an additional node has to be mapped to (potentially) make more arcs implementable. The structure of Algorithm 6.1 might seem a bit counterintuitive, because it contains the node mapping twice, which is not strictly necessary. We chose this structure because it can be easily adapted to work with a partial solution as input, which has to be completed. This functionality will be required for the Local Search and Variable Neighborhood Descent algorithms discussed in the following sections. The required modification to Algorithm 6.1 is removing its first line and adding Solution S, the solution to be completed, as second input.
We implemented four heuristics for each of the four different sub-problems to solve, which are described in the following. If the strategies define no specific order of nodes or arcs (or ties occur), it is arbitrary. Now follows the discussion of the SVN heuristics.

**NextVNode** Selects an unmapped virtual node from $V'$.

**CPUHeavy** Selects the virtual node with the highest sum of CPU requirement and connected bandwidth. The connected bandwidth is the sum of the bandwidth requirements of all virtual arcs that start, or end, at a virtual node. It is important to include this factor, since it is a (nearly) guaranteed CPU load caused by the virtual node. It slightly overestimates the CPU load because transferring data from one virtual node to another if both are hosted on the same substrate node requires CPU capacity only once but is counted twice according to this calculation model. We want to focus on virtual nodes that require a lot of resources because they are the most problematic to fit into the substrate network.

**CPUHeavyVN** Selects with CPUHeavy from the virtual network (VN) with the highest total CPU and bandwidth requirements that still has unmapped nodes left. Concentrating on one virtual network when selecting nodes supports AE, since virtual arcs become implementable much faster.

**DLHeavyVN** Selects with CPUHeavy from the VN with the lowest total sum of allowed delays that still has unmapped nodes left. This heuristic assumes that VNs that are highly delay constrained are the most critical to implement. Especially in concert with AE, this ensures that virtual arcs with stringent delay constraints are implemented first, when the substrate still has enough resources to allow a delay feasible path without any additional costs in terms of $C_a$.

Now follows the description of the node mapping heuristics. Note that for the node mapping strategies, only substrate nodes allowed by $M$ are regarded as candidates. If no substrate node would yield a valid solution, i.e., no allowed substrate node has enough CPU resources or connected bandwidth left, then the substrate node (allowed by $M$) with the most free resources is chosen. This is the substrate node with the least missing resources in case no substrate node has resources left.

**NextSNode** Maps a virtual node to the first substrate node with enough free CPU capacity.

**NextFree** Maps a virtual node to the first substrate node with enough free CPU capacity to support the CPU requirements and the total connected bandwidth of the virtual node (the total CPU load of a virtual node).

**MostFree** Maps to the substrate node with the most free CPU capacity. In case of ties, the substrate node with the most free connected bandwidth is chosen as map target.

**CheapHost** Maps to the substrate node with enough free resources (with respect to the total CPU load) and least increase of $C_u$, i.e., if possible to a substrate node that already hosts virtual nodes.

For the following description of virtual arc selection heuristics, keep in mind that the SVA strategies only consider implementable virtual arcs.

**NextVArc** Selects an arbitrary unimplemented virtual arc.

**BWHeavy** Selects the arc with the highest bandwidth requirement.

**DLHeavy** Selects the arc with the smallest delay.

**RelDLHeavy** Selects the arc with the smallest fraction of allowed delay to shortest possible delay between $m(s(f))$ and $m(t(f))$. This might be a more accurate measure of how delay constrained a connection actually is.

All four IVA strategies implement a virtual arc $f$ by finding a Delay Constrained Shortest Path in the substrate from $m(s(f))$ to $m(t(f))$ via the Dynamic Program from [69]. The only difference between the strategies is the calculation of the substrate arc costs, which define the length of a path.

**MinUse** If substrate arcs have already been used, they have a cost of 0. Otherwise, their usage cost $p_e^A$ is assigned. If arcs do not have enough free bandwidth, or their source and target node not enough CPU capacity to host $f$, a penalty cost of $10^6$ is applied.

**Spread-n** The cost of a substrate arc is the sum the fraction of the arc's remaining free bandwidth that would be used by the virtual arc and the fraction of free CPU capacity the virtual arc would use on the node the substrate arc connects to. This represents the relative resource usage the virtual arc would incur on a substrate arc. Low values mean that the virtual arc has a low impact on the available resources of a substrate arc (and the node

it connects to). The relative resource usage is then taken to the power of $n \in \{0.5, 1, 2\}$ so that it is possible to evaluate the influence of different biasing strategies. We will denote them by Spread-0.5, Spread-1 and Spread-2 respectively.

These methods result in a total of 512 different CH configurations, the results of their evaluation can be found in Section 6.5. The strategies were kept simple to keep running times short as the following heuristics build on the best CH variants.

## 6.3 Local Search

The basic idea of Local Search (LS) is that a found solution to a problem may be improved by iteratively making small changes. The solutions immediately reachable from a starting solution $S$ are defined by a neighborhood $N(S)$, which can be generated by the appropriate neighborhood structure. LS starts with a solution $S$ and replaces it with a better solution from $N(S)$ until no more improvements can be found. For selecting the neighbor, we use the two standard strategies first-improvement (select the first improving solution) and best-improvement (select the best solution from a neighborhood).
We implemented six different neighborhood structures for the VNMP. They are ruin-and-recreate [154] neighborhoods, which means (in the context of this work) that they remove a part of a complete solution, for instance the implementation of a virtual arc, and then rebuild the solution by applying a CH. Here we need the CH with the modified structure as discussed in the previous section. The discussion of the neighborhoods will skip this rebuilding step.

**RemapVarc ($\mathbf{N}_1$)** Removes the implementation of a virtual arc.

**RemapVnode ($\mathbf{N}_2$)** Removes a virtual node and the implementations of all adjacent virtual arcs.

**RemapSlice ($\mathbf{N}_3$)** Removes a virtual network from the solution by removing all virtual nodes and implementations of all virtual arcs of the virtual network.

**ClearSarc ($\mathbf{N}_4$)** Clears a substrate arc, which means it removes the implementation of all virtual arcs using this substrate arc.

**ClearSnode ($\mathbf{N}_5$)** Clears a substrate node, which means it removes the implementation of all virtual arcs that are crossing the substrate node and removes all virtual nodes that are mapped to the substrate node.

**RemapVnodeTAP ($\mathbf{N}_6$)** Works like RemapVnode, but instead of delegating the choice of substrate node for the removed virtual node to the CH, it explicitly tries to map the virtual node to all possible (TAP) substrate nodes.

Note that the description of the neighborhood structures only specifies how one neighboring solution is reached. How the neighborhood is used during Local Search is straight forward to derive from the description. For example, when performing Local Search with the RemapVarc

neighborhood and best-improvement, we start with an initial solution created by CH. Then we remove the implementation of a virtual arc and rebuild the solution with CH. If the created solution is better, we store it as the currently best one. This procedure is repeated for all remaining virtual arcs, always starting from the initial solution. After all virtual arcs have been tested, the best found solution replaces the initial solution and the process of finding an improving solution is repeated until no further improvements can be found.
For each neighborhood, there is a natural order in which to evaluate the neighbors, for instance the order in which virtual nodes are specified in the VNMP instance. This order is relevant when we use first-improvement. We might be able to speed up the search process if we try the most promising neighbors first. When the current solution for example is not valid, then the most promising neighbors are those which might reduce $C_a$. In case of RemapVnode, that means virtual nodes, which are mapped to substrate nodes that are overloaded (additional resources had to be bought there), should be tried first. For neighborhoods that focus on the substrate (ClearSarc, ClearSnode), overloaded substrate nodes or arcs should be cleared first. We will denote this neighbor ordering by OverloadingFirst. When solving VNMP-S instead of VNMP-O, it might make sense to only consider neighbors which might reduce $C_a$ instead of just prioritizing them. We will call this strategy OnlyOverloading. The choice of not focusing on any particular neighbors will be denoted by None. Note that even when solving VNMP-S, OnlyOverloading is not as strong as OverloadingFirst since changing parts of a solution that do not directly contribute to $C_a$ might make future improvements of $C_a$ possible.
For an evaluation of the different neighborhoods, see Section 6.5.

## 6.4 Variable Neighborhood Descent

The neighborhood structures discussed in the previous section can be applied in combination within a Variable Neighborhood Descent (VND) algorithm [74]. A VND utilizes a series of neighborhoods $N_1 \dots N_k$. An initial solution is improved by $N_1$ until no more improvements can be found, then $N_2$ is applied to the solution and so on. If $N_k$ fails, VND terminates. If an improved solution is found in some neighborhood, VND restarts with $N_1$. For more information, see Section 2.2.6.
We use the neighborhood structures defined in the previous section in two variants: as described without any neighbor prioritization and with OnlyOverloading. We will denote the second variant with a prime. For example $N_6'$ denotes RemapVnodeTAP in OnlyOverloading configuration. The following neighborhood orderings (VND configurations) were tested:

**All ($C_1$):** $N_1'$, $N_2'$, $N_3'$, $N_4'$, $N_5'$, $N_6'$, $N_1$, $N_2$, $N_3$, $N_4$, $N_5$, $N_6$
All neighborhoods, in order of their size.

**OnlyOverloading ($C_2$):** $N_1'$, $N_2'$, $N_3'$, $N_4'$, $N_5'$, $N_6'$
Only neighborhoods in OnlyOverloading configuration.

**Complete ($C_3$):** $N_1$, $N_2$, $N_3$, $N_4$, $N_5$, $N_6$
Only complete neighborhoods.

**RComplete ($\mathbf{C}_4$):** $N_6$, $N_5$, $N_4$, $N_3$, $N_2$, $N_1$
Like $C_3$, but in reverse order.

**ImprovCompA ($\mathbf{C}_5$):** $N_1$, $N_2$, $N_3$, $N_4$, $N_6$
An improvement to Complete based on preliminary results which showed that ClearSnode does not contribute in a significant way to VND.

**RImprovCompA ($\mathbf{C}_6$):** $N_6$, $N_4$, $N_3$, $N_2$, $N_1$
$C_5$ in reverse order.

**ImprovOverload ($\mathbf{C}_7$):** $N_1'$, $N_2'$, $N_3'$
The neighborhoods of OnlyOverloading which find improvements based on preliminary results.

**RImprovOverload ($\mathbf{C}_8$):** $N_3'$, $N_2'$, $N_1'$
$C_7$ in reverse order.

**ImprovCompB ($\mathbf{C}_9$):** $N_2$, $N_4$, $N_5$, $N_6$
Another selection of neighborhoods to improve Complete.

**ImprovCompC ($\mathbf{C}_{10}$):** $N_3$, $N_4$, $N_5$, $N_6$
$C_9$, but using RemapSlice instead of RemapVnode in the hope of speeding up the algorithm while achieving similar results.

**OnlyClear ($\mathbf{C}_{11}$):** $N_4$, $N_5$
Only the neighborhoods that try to clear parts of the substrate.

**ImprovCompD ($\mathbf{C}_{12}$):** $N_2$, $N_4$, $N_5$
A variant of $C_9$ which does not use RemapVnodeTAP to improve run-times.

## 6.5 Results

Each CH, LS and VND variant was tested on the full VNMP instance set as described in Chapter 5. In addition to load level 1, we also used 0.1, 0.5 and 0.8 to see how the different algorithms and neighborhoods react to changing levels of hardness. This gives a total of 840 instances. A time-limit of 1000 seconds was applied, which was only reached for some VND configurations for the largest instance sizes. The main objective of the evaluation was to find configurations which are suitable for solving VNMP-S or VNMP-O, but also to identify configurations which have a good tradeoff between performance and required run-time. We use the average additional resource cost $C_a$ to evaluate the performance of an algorithm with respect to VNMP-S and the relative rank $R_{\text{rel}}$ to determine the suitability for VNMP-O. The price of one unit of CPU resources $p^{\text{CPU}}$ was set to one, the price of one unit of bandwidth $p^{\text{BW}}$ to five.

### 6.5.1 Construction Heuristics

Before we can compare all presented heuristics, we needed to identify promising CH variants which can be used to generate the initial solution for LS and VND and perform the rebuilding step of the proposed ruin-and-recreate neighborhoods. To do this, we evaluated the 512 different CH configurations on all 840 VNMP instances. The following tables will show the influence of the different choices for the selection and implementation strategies.

Table 6.1 shows the average performance of the CH configurations depending on the chosen SVN strategy. Based on this data, using DLHeavyVN as the strategy for choosing a virtual node to be mapped is the obvious choice, as this strategy dominates with respect to $R_{\text{rel}}$ and $C_a$. Note that in some cases, it is significantly better to choose an arbitrary virtual node to implement (NextVNode) instead of one that requires a lot of resources (e.g., CPUHeavy) when solving VNMP-S. NextVNode also performs surprisingly well with respect to the $R_{\text{rel}}$.

The influence of the chosen IVN strategy is presented in Table 6.2. It is very clear that MostFree is the IVN strategy of choice, beating the other choices by a very large margin both with respect to $R_{\text{rel}}$ and to $C_a$. The additional resource cost is very close to zero for instances with the lowest load. That means nearly every CH configuration creates a valid VNMP solution for all VNMP instances of lowest load, no matter which configuration choices are made, as long as MostFree is used as IVN strategy (these are 128 different CH configurations in total).

Regarding the influence of the SVA strategy, Table 6.3 shows that in this case the best choice really depends on the instance size and load. For small instances, focusing on delay constrained virtual arcs works best. When the load is low, it is beneficial to focus on the virtual arcs that have a delay requirement close to the best achievable delay within the substrate. For higher loads, implementing virtual arcs with low allowed delay values is more important. Notice however, that the differences in $R_{\text{rel}}$ and $C_a$, while statistically significant, are very small. Beginning with instances of size 200, the critical virtual arcs are those with high bandwidth requirements.

Table 6.4 shows that the best choice of IVA strategy mostly depends on instance load. When trying to minimize $R_{\text{rel}}$, using MinUse for the lowest loads is advantageous, but in most cases Spread-1 performs the best. For low loads and large instances, squaring the relative resource usage of a virtual arc is advantageous. Using Spread-1 achieves the best results in nearly all cases when minimizing $C_a$ is the objective.

The last parameter of CH is the implementation emphasis and its influence is presented in Table 6.5. Using AE gives clearly better results, both in terms of $R_{\text{rel}}$ and $C_a$.

The presented results suggest the following CH configuration for the best performance: DLHeavyVN as SVN, MostFree as IVN, DLHeavy as SVA, Spread-1 as IVA and using AE. Using this would of course be wrong. We have only presented the average performance tendencies of the different strategic choices. It may very well be that a great configuration is drowned in a sea of average or bad configurations and as it turns out, the suggested configuration is neither the best with respect to $R_{\text{rel}}$, nor with respect to $C_a$. To find the truly best configurations, we order the compared configurations with respect to their average $R_{\text{rel}}$ (or $C_a$) over all VNMP instances and choose the top configuration. Note that we choose based on average performance over all instances. We have already shown that the strategy selection is very sensitive with respect to the instance size and also the load. We could have also defined the best configurations for particular size or load classes. However, this would make further discussion very hard to follow, especially

Table 6.1: Influence of the chosen SVN strategy on the performance of CH.

| | | $R_{rel}$ | | | | $C_a$ | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Size** | **Load** | **NextVNode** | **CPUHeavy** | **CPUHeavyVN** | **DLHeavyVN** | **NextVNode** | **CPUHeavy** | **CPUHeavyVN** | **DLHeavyVN** |
| 20 | 0.10 | 0.479 > | 0.537 > | 0.521 > | 0.454 = | 58.5 = | 61.2 > | 58.4 > | 44.8 = |
| | 0.50 | 0.423 > | 0.540 > | 0.501 > | 0.404 = | 1815.9 > | 2854.8 > | 2645.4 > | 1624.4 = |
| | 0.80 | 0.389 > | 0.512 > | 0.459 > | 0.356 = | 6302.6 > | 9385.4 > | 8586.0 > | 5572.5 = |
| | 1.00 | 0.425 > | 0.539 > | 0.487 > | 0.397 = | 10704.1 > | 14922.1 > | 13752.3 > | 9813.4 = |
| 30 | 0.10 | 0.535 > | 0.602 > | 0.596 > | 0.501 = | 42.4 > | 49.9 > | 45.8 > | 32.4 = |
| | 0.50 | 0.359 > | 0.518 > | 0.455 > | 0.321 = | 2880.0 > | 4934.5 > | 4265.0 > | 2429.7 = |
| | 0.80 | 0.352 > | 0.523 > | 0.461 > | 0.327 = | 7638.8 > | 13327.9 > | 11817.7 > | 6691.6 = |
| | 1.00 | 0.379 > | 0.540 > | 0.467 > | 0.362 = | 14479.2 > | 22861.1 > | 20134.8 > | 13488.4 = |
| 50 | 0.10 | 0.485 > | 0.516 > | 0.513 > | 0.434 = | 70.5 > | 81.1 > | 77.7 > | 67.3 = |
| | 0.50 | 0.342 > | 0.486 > | 0.429 > | 0.309 = | 3121.7 > | 5866.0 > | 4992.9 > | 2792.1 = |
| | 0.80 | 0.360 > | 0.504 > | 0.444 > | 0.329 = | 8829.4 > | 15524.2 > | 13555.8 > | 7999.0 = |
| | 1.00 | 0.387 > | 0.534 > | 0.465 > | 0.377 = | 17835.8 > | 27937.3 > | 24395.4 > | 17110.2 = |
| 100 | 0.10 | 0.530 > | 0.572 > | 0.557 > | 0.504 = | 181.6 = | 225.7 > | 195.5 > | 185.1 > |
| | 0.50 | 0.368 > | 0.507 > | 0.447 > | 0.336 = | 5813.2 > | 9900.6 > | 8450.1 > | 5108.6 = |
| | 0.80 | 0.372 > | 0.518 > | 0.453 > | 0.348 = | 17596.6 > | 28750.2 > | 25693.1 > | 15489.5 = |
| | 1.00 | 0.396 > | 0.533 > | 0.472 > | 0.372 = | 33843.5 > | 50388.3 > | 45585.4 > | 30608.5 = |
| 200 | 0.10 | 0.434 > | 0.471 > | 0.463 > | 0.424 = | 469.8 = | 629.3 > | 579.8 > | 480.1 = |
| | 0.50 | 0.394 > | 0.498 > | 0.447 > | 0.376 = | 6062.7 > | 10290.6 > | 8786.8 > | 5666.0 = |
| | 0.80 | 0.420 > | 0.526 > | 0.468 > | 0.396 = | 18516.0 > | 28589.2 > | 24682.5 > | 17042.0 = |
| | 1.00 | 0.442 > | 0.550 > | 0.492 > | 0.422 = | 38208.1 > | 53651.3 > | 47368.1 > | 35975.8 = |
| 500 | 0.10 | 0.424 > | 0.464 > | 0.456 > | 0.402 = | 2382.0 > | 2773.4 > | 2697.3 > | 2292.6 = |
| | 0.50 | 0.433 > | 0.518 > | 0.478 > | 0.395 = | 31111.8 > | 42214.1 > | 37565.6 > | 26359.6 = |
| | 0.80 | 0.449 > | 0.535 > | 0.493 > | 0.418 = | 67833.6 > | 89784.4 > | 79344.5 > | 58674.5 = |
| | 1.00 | 0.465 > | 0.557 > | 0.512 > | 0.428 = | 105206.9 > | 137353.0 > | 122249.9 > | 91504.3 = |
| 1000 | 0.10 | 0.432 > | 0.454 > | 0.443 > | 0.416 = | 4078.5 > | 4288.3 > | 4202.9 > | 3916.8 = |
| | 0.50 | 0.451 > | 0.522 > | 0.485 > | 0.425 = | 43036.6 > | 54687.8 > | 48947.4 > | 37740.8 = |
| | 0.80 | 0.470 > | 0.540 > | 0.504 > | 0.445 = | 88892.0 > | 113597.4 > | 101484.4 > | 78002.9 = |
| | 1.00 | 0.475 > | 0.555 > | 0.514 > | 0.448 = | 134388.2 > | 170552.6 > | 153323.9 > | 119068.0 = |

Table 6.2: Influence of the chosen IVN strategy on the performance of CH.

| | | $R_{rel}$ | | | | $C_a$ | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Size** | **Load** | **NextSNode** | **NextFree** | **MostFree** | **CheapHost** | **NextSNode** | **NextFree** | **MostFree** | **CheapHost** |
| 20 | 0.10 | 0.626 > | 0.612 > | 0.337 = | 0.416 > | 143.0 > | 76.4 > | 0.0 = | 3.3 > |
| | 0.50 | 0.716 > | 0.640 > | 0.151 = | 0.362 > | 3941.3 > | 3383.5 > | 21.4 = | 1594.5 > |
| | 0.80 | 0.653 > | 0.582 > | 0.118 = | 0.363 > | 12345.8 > | 11188.1 > | 408.1 = | 5904.5 > |
| | 1.00 | 0.674 > | 0.606 > | 0.168 = | 0.401 > | 19100.4 > | 17573.6 > | 2079.9 = | 10437.9 > |
| 30 | 0.10 | 0.672 > | 0.649 > | 0.420 = | 0.493 > | 85.2 > | 68.6 > | 0.0 = | 16.7 > |
| | 0.50 | 0.631 > | 0.559 > | 0.115 = | 0.348 > | 6313.9 > | 5403.6 > | 14.9 = | 2777.0 > |
| | 0.80 | 0.621 > | 0.554 > | 0.108 = | 0.380 > | 15577.8 > | 13888.6 > | 520.6 = | 9488.9 > |
| | 1.00 | 0.637 > | 0.577 > | 0.138 = | 0.396 > | 26807.1 > | 24639.0 > | 3264.1 = | 16253.3 > |
| 50 | 0.10 | 0.588 > | 0.525 > | 0.306 = | 0.529 > | 191.7 > | 85.9 > | 0.0 = | 19.0 > |
| | 0.50 | 0.582 > | 0.489 > | 0.119 = | 0.375 > | 7178.8 > | 5884.7 > | 45.8 = | 3663.5 > |
| | 0.80 | 0.599 > | 0.526 > | 0.098 = | 0.414 > | 17921.9 > | 15647.9 > | 646.9 = | 11691.6 > |
| | 1.00 | 0.624 > | 0.556 > | 0.139 = | 0.446 > | 31924.9 > | 28690.8 > | 4162.3 = | 22500.8 > |
| 100 | 0.10 | 0.687 > | 0.595 > | 0.344 = | 0.537 > | 508.6 > | 250.9 > | 0.2 = | 28.2 > |
| | 0.50 | 0.628 > | 0.519 > | 0.110 = | 0.401 > | 12717.7 > | 10092.6 > | 76.6 = | 6385.7 > |
| | 0.80 | 0.617 > | 0.532 > | 0.103 = | 0.439 > | 33583.7 > | 28642.0 > | 1313.5 = | 23990.2 > |
| | 1.00 | 0.633 > | 0.556 > | 0.132 = | 0.452 > | 58386.7 > | 51981.5 > | 6827.4 = | 43230.1 > |
| 200 | 0.10 | 0.598 > | 0.466 > | 0.209 = | 0.519 > | 1294.9 > | 676.9 > | 0.0 = | 187.1 > |
| | 0.50 | 0.603 > | 0.487 > | 0.095 = | 0.531 > | 10628.5 > | 7790.0 > | 150.3 = | 12237.3 > |
| | 0.80 | 0.631 > | 0.531 > | 0.101 = | 0.547 > | 30648.3 > | 25306.6 > | 1759.9 = | 31115.0 > |
| | 1.00 | 0.650 > | 0.570 > | 0.157 = | 0.529 > | 59623.6 > | 52057.9 > | 9869.4 = | 53652.3 > |
| 500 | 0.10 | 0.830 > | 0.546 > | 0.074 = | 0.297 > | 6385.5 > | 2971.1 > | 0.1 = | 788.6 > |
| | 0.50 | 0.776 > | 0.594 > | 0.089 = | 0.365 > | 74085.0 > | 45915.8 > | 152.4 = | 17097.8 > |
| | 0.80 | 0.773 > | 0.617 > | 0.102 = | 0.404 > | 146362.1 > | 101503.3 > | 4162.6 = | 43608.9 > |
| | 1.00 | 0.770 > | 0.628 > | 0.151 = | 0.413 > | 208378.7 > | 154279.4 > | 18410.3 = | 75245.7 > |
| 1000 | 0.10 | 0.844 > | 0.568 > | 0.069 = | 0.264 > | 11005.2 > | 4505.1 > | 1.5 = | 974.7 > |
| | 0.50 | 0.818 > | 0.604 > | 0.092 = | 0.368 > | 106735.2 > | 57087.9 > | 306.2 = | 20283.3 > |
| | 0.80 | 0.812 > | 0.625 > | 0.119 = | 0.402 > | 205072.2 > | 121937.6 > | 5953.1 = | 49013.9 > |
| | 1.00 | 0.808 > | 0.631 > | 0.157 = | 0.395 > | 287114.7 > | 183846.1 > | 23381.0 = | 82990.8 > |

Table 6.3: Influence of the chosen SVA strategy on the performance of CH.

| | | $R_{rel}$ | | | | $C_a$ | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Size** | **Load** | **NextVArc** | **BWHeavy** | **DLHeavy** | **RelDLHeavy** | **NextVArc** | **BWHeavy** | **DLHeavy** | **RelDLHeavy** |
| 20 | 0.10 | 0.499 > | 0.500 > | 0.496 > | 0.495 = | 55.7 = | 55.7 = | 55.7 = | 55.7 = |
| | 0.50 | 0.468 > | 0.469 > | 0.466 = | 0.466 > | 2235.0 > | 2242.3 > | 2229.9 = | 2233.4 > |
| | 0.80 | 0.430 > | 0.431 > | 0.428 = | 0.428 > | 7461.8 > | 7478.4 > | 7448.4 = | 7457.8 > |
| | 1.00 | 0.463 > | 0.464 > | 0.460 = | 0.462 > | 12302.3 > | 12323.7 > | 12277.9 = | 12288.0 = |
| 30 | 0.10 | 0.562 > | 0.563 > | 0.557 > | 0.552 = | 42.6 = | 42.7 = | 42.5 = | 42.7 = |
| | 0.50 | 0.415 > | 0.417 > | 0.411 > | 0.410 = | 3634.8 > | 3648.4 > | 3619.2 > | 3606.8 = |
| | 0.80 | 0.418 > | 0.417 > | 0.413 = | 0.415 > | 9882.4 > | 9911.4 > | 9818.2 = | 9863.9 > |
| | 1.00 | 0.438 > | 0.438 > | 0.436 = | 0.436 = | 17765.2 > | 17843.7 > | 17642.8 = | 17711.8 = |
| 50 | 0.10 | 0.491 > | 0.496 > | 0.482 > | 0.479 = | 74.1 = | 74.7 > | 73.9 = | 73.9 = |
| | 0.50 | 0.394 > | 0.394 > | 0.389 > | 0.388 = | 4206.6 > | 4209.1 > | 4178.9 = | 4178.1 = |
| | 0.80 | 0.413 > | 0.413 > | 0.406 > | 0.405 = | 11553.8 > | 11562.3 > | 11406.0 > | 11386.2 = |
| | 1.00 | 0.442 > | 0.444 > | 0.438 = | 0.439 = | 21887.7 > | 22054.7 > | 21685.6 = | 21650.7 = |
| 100 | 0.10 | 0.543 > | 0.551 > | 0.536 > | 0.534 = | 195.6 = | 196.2 = | 200.0 > | 196.1 > |
| | 0.50 | 0.417 > | 0.419 > | 0.410 = | 0.411 = | 7321.4 > | 7360.4 > | 7293.4 = | 7297.3 = |
| | 0.80 | 0.423 > | 0.426 > | 0.420 = | 0.422 > | 21872.9 > | 22097.8 > | 21724.9 = | 21833.7 > |
| | 1.00 | 0.442 > | 0.449 > | 0.439 = | 0.443 > | 40028.6 > | 40928.9 > | 39623.6 = | 39844.7 > |
| 200 | 0.10 | 0.450 > | 0.452 > | 0.445 > | 0.446 = | 538.5 > | 537.9 = | 541.1 > | 541.5 > |
| | 0.50 | 0.429 > | 0.432 > | 0.428 = | 0.427 = | 7694.3 = | 7726.6 = | 7693.9 = | 7691.4 = |
| | 0.80 | 0.452 = | 0.455 = | 0.450 > | 0.453 > | 22171.7 = | 22588.7 = | 21969.6 > | 22099.8 > |
| | 1.00 | 0.474 = | 0.482 > | 0.474 > | 0.477 > | 43488.1 = | 45005.0 > | 43247.7 > | 43462.5 > |
| 500 | 0.10 | 0.439 > | 0.434 = | 0.439 > | 0.435 = | 2542.9 > | 2517.5 = | 2546.8 > | 2538.1 > |
| | 0.50 | 0.457 > | 0.456 = | 0.455 > | 0.455 > | 34365.5 > | 34249.4 = | 34333.5 > | 34302.6 > |
| | 0.80 | 0.474 > | 0.473 = | 0.474 > | 0.475 > | 73999.8 > | 74337.1 = | 73647.7 > | 73652.4 > |
| | 1.00 | 0.490 > | 0.492 = | 0.489 > | 0.491 > | 114180.3 > | 115803.8 = | 113103.6 > | 113226.5 > |
| 1000 | 0.10 | 0.437 > | 0.430 = | 0.438 > | 0.439 > | 4122.2 > | 4080.5 = | 4135.5 > | 4148.3 > |
| | 0.50 | 0.471 > | 0.468 = | 0.471 > | 0.471 > | 46167.1 > | 45809.0 = | 46264.5 > | 46171.9 > |
| | 0.80 | 0.489 > | 0.487 = | 0.491 > | 0.491 > | 95570.4 > | 95466.0 = | 95531.1 > | 95409.3 > |
| | 1.00 | 0.497 > | 0.497 = | 0.498 > | 0.499 > | 144268.3 > | 145707.3 = | 143710.7 > | 143646.3 > |

Table 6.4: Influence of the chosen IVA strategy on the performance of CH.

| | | $R_{rel}$ | | | | $C_a$ | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Size** | **Load** | **MinUse** | **Spread-1** | **Spread-0.5** | **Spread-2** | **MinUse** | **Spread-1** | **Spread-0.5** | **Spread-2** |
| 20 | 0.10 | 0.477 = | 0.502 > | 0.520 > | 0.491 > | 55.9 > | 55.6 = | 55.7 = | 55.6 = |
| | 0.50 | 0.480 > | 0.461 = | 0.466 > | 0.462 > | 2347.5 > | 2195.4 = | 2200.6 > | 2197.1 > |
| | 0.80 | 0.450 > | 0.421 = | 0.422 > | 0.423 > | 7738.1 > | 7367.5 > | 7357.7 = | 7383.1 > |
| | 1.00 | 0.487 > | 0.452 = | 0.452 = | 0.458 > | 12876.3 > | 12083.8 = | 12080.3 = | 12151.4 > |
| 30 | 0.10 | 0.471 = | 0.589 > | 0.614 > | 0.561 > | 45.1 > | 41.8 = | 41.8 = | 41.8 = |
| | 0.50 | 0.441 > | 0.402 = | 0.408 > | 0.403 = | 3903.8 > | 3529.7 = | 3544.1 > | 3531.7 > |
| | 0.80 | 0.456 > | 0.399 = | 0.402 > | 0.404 > | 10838.0 > | 9520.4 = | 9534.3 > | 9583.2 > |
| | 1.00 | 0.491 > | 0.414 > | 0.414 = | 0.428 > | 19943.0 > | 16879.6 = | 16866.5 > | 17274.3 > |
| 50 | 0.10 | 0.397 = | 0.519 > | 0.540 > | 0.491 > | 81.5 > | 71.6 = | 71.6 = | 71.9 > |
| | 0.50 | 0.427 > | 0.376 = | 0.383 > | 0.379 > | 4557.0 > | 4052.4 = | 4058.9 > | 4104.4 > |
| | 0.80 | 0.466 > | 0.386 = | 0.390 > | 0.395 > | 12881.3 > | 10923.9 = | 10956.1 > | 11147.0 > |
| | 1.00 | 0.522 > | 0.407 = | 0.410 > | 0.425 > | 25347.6 > | 20428.6 = | 20487.0 > | 21015.5 > |
| 100 | 0.10 | 0.407 = | 0.579 > | 0.623 > | 0.554 > | 214.4 > | 190.9 = | 191.7 > | 191.0 > |
| | 0.50 | 0.454 > | 0.399 = | 0.406 > | 0.399 = | 8022.1 > | 7060.2 = | 7089.2 > | 7101.1 > |
| | 0.80 | 0.493 > | 0.395 = | 0.399 > | 0.403 > | 25293.0 > | 20642.6 = | 20713.2 > | 20880.6 > |
| | 1.00 | 0.537 > | 0.407 = | 0.408 > | 0.422 > | 48594.2 > | 36952.2 = | 36992.9 > | 37886.4 > |
| 200 | 0.10 | 0.374 = | 0.470 > | 0.506 > | 0.441 > | 602.7 > | 509.2 = | 526.9 > | 520.3 = |
| | 0.50 | 0.476 > | 0.408 = | 0.419 > | 0.413 > | 8717.5 > | 7304.9 = | 7396.7 > | 7387.0 > |
| | 0.80 | 0.555 > | 0.412 = | 0.419 > | 0.423 > | 27737.9 > | 20186.1 = | 20348.7 > | 20557.1 > |
| | 1.00 | 0.621 > | 0.420 = | 0.425 > | 0.441 > | 59471.9 > | 38104.1 = | 38343.9 > | 39283.4 > |
| 500 | 0.10 | 0.444 > | 0.429 > | 0.450 > | 0.424 = | 2721.5 > | 2455.5 > | 2518.7 > | 2449.5 = |
| | 0.50 | 0.486 > | 0.442 = | 0.452 > | 0.443 > | 37137.4 > | 33220.4 = | 33569.7 > | 33323.5 > |
| | 0.80 | 0.530 > | 0.451 = | 0.459 > | 0.457 > | 83113.8 > | 70507.0 = | 71029.3 > | 70986.8 > |
| | 1.00 | 0.577 > | 0.456 = | 0.463 > | 0.467 > | 137133.1 > | 105808.8 = | 106409.7 > | 106962.5 > |
| 1000 | 0.10 | 0.448 > | 0.427 > | 0.452 > | 0.418 = | 4438.4 > | 3971.8 > | 4108.7 > | 3967.6 = |
| | 0.50 | 0.504 > | 0.454 = | 0.466 > | 0.457 > | 50099.2 > | 44401.8 = | 45145.3 > | 44766.2 > |
| | 0.80 | 0.540 > | 0.466 = | 0.477 > | 0.475 > | 107190.5 > | 90793.7 = | 91997.8 > | 91994.8 > |
| | 1.00 | 0.580 > | 0.463 = | 0.473 > | 0.477 > | 172246.9 > | 133765.1 = | 135233.4 > | 136087.2 > |

Table 6.5: Influence of the choice of implementation emphasis on the performance of CH.

| | | $\mathbf{R_{rel}}$ | | $\mathbf{C_a}$ | |
|---|---|---|---|---|---|
| **Size** | **Load** | **NE** | **AE** | **NE** | **AE** |
| 20 | 0.10 | 0.494 = | 0.501 > | 58.6 > | 52.8 = |
| | 0.50 | 0.578 > | 0.356 = | 3747.1 > | 723.2 = |
| | 0.80 | 0.585 > | 0.273 = | 12778.5 > | 2144.7 = |
| | 1.00 | 0.646 > | 0.278 = | 20671.6 > | 3924.3 = |
| 30 | 0.10 | 0.566 > | 0.551 = | 56.7 > | 28.5 = |
| | 0.50 | 0.538 > | 0.289 = | 6169.9 > | 1084.7 = |
| | 0.80 | 0.574 > | 0.257 = | 16474.9 > | 3263.1 = |
| | 1.00 | 0.615 > | 0.259 = | 28356.6 > | 7125.1 = |
| 50 | 0.10 | 0.504 > | 0.469 = | 99.9 > | 48.4 = |
| | 0.50 | 0.520 > | 0.262 = | 7026.1 > | 1360.2 = |
| | 0.80 | 0.564 > | 0.254 = | 18924.2 > | 4029.9 = |
| | 1.00 | 0.618 > | 0.264 = | 34720.5 > | 8918.9 = |
| 100 | 0.10 | 0.563 > | 0.519 = | 265.0 > | 128.9 = |
| | 0.50 | 0.548 > | 0.281 = | 12218.7 > | 2417.6 = |
| | 0.80 | 0.589 > | 0.257 = | 36500.2 > | 7264.4 = |
| | 1.00 | 0.622 > | 0.265 = | 63745.9 > | 16467.0 = |
| 200 | 0.10 | 0.494 > | 0.402 = | 754.4 > | 325.1 = |
| | 0.50 | 0.566 > | 0.292 = | 12417.1 > | 2985.9 = |
| | 0.80 | 0.608 > | 0.296 = | 34072.4 > | 10342.5 = |
| | 1.00 | 0.642 > | 0.311 = | 64038.9 > | 23562.7 = |
| 500 | 0.10 | 0.485 > | 0.388 = | 2960.5 > | 2112.1 = |
| | 0.50 | 0.567 > | 0.345 = | 51195.5 > | 17430.0 = |
| | 0.80 | 0.611 > | 0.337 = | 111432.2 > | 36386.3 = |
| | 1.00 | 0.637 > | 0.344 = | 168010.0 > | 60147.1 = |
| 1000 | 0.10 | 0.484 > | 0.388 = | 4731.2 > | 3512.1 = |
| | 0.50 | 0.565 > | 0.376 = | 65001.9 > | 27204.4 = |
| | 0.80 | 0.607 > | 0.372 = | 136567.3 > | 54421.1 = |
| | 1.00 | 0.624 > | 0.371 = | 203415.5 > | 85250.8 = |

when different CH configurations are used as sub-solver within LS or VND. Because of that, we choose a selection based on the average performance over all instances.

Table 6.6 shows the ten best CH configurations with respect to achieved average $R_{\text{rel}}$. Due to space constraints we cannot show the full table containing all 512 configurations. The best configuration for solving VNMP-O is DLHeavyVN, MostFree, RelDLHeavy, MinUse, and AE. We will denote this configuration by CH-O. The presented table also shows the importance of the different strategic choices for the final outcome. Least important is the choice of SVA, as we can see groups of four configurations that only differ in this strategy. For the different choices of IVA, we can at least observe some differences in $R_{\text{rel}}$ based on the selected strategy. The 16th best configuration uses a different SVN strategy than CH-O (which is not visible in the table). The 53rd configuration is the first that does not use AE and the most important strategic choice for CH is the mapping strategy for virtual nodes. The 95 best configurations map virtual nodes to the substrate node with the most free resources. As a side note, the choice of AE explains

Table 6.6: Average $R_{\text{rel}}$ and $C_u$, number of valid solutions (# Valid) and average required CPU-time (t[s]) of the top 10 CH configurations according to $R_{\text{rel}}$. The implementation emphasis is denoted by Em.

| SVN | IVN | SVA | IVA | Em. | $\mathbf{R_{rel}}$ | $\mathbf{C_a}$ | # Valid | t[s] |
|---|---|---|---|---|---|---|---|---|
| DLHeavyVN | MostFree | RelDLHeavy | MinUse | AE | 0.091 = | 1950.8 > | 511 | 0.3 |
| DLHeavyVN | MostFree | DLHeavy | MinUse | AE | 0.092 > | 1958.8 > | 510 | 0.2 |
| DLHeavyVN | MostFree | BWHeavy | MinUse | AE | 0.092 > | 1964.1 > | 507 | 0.2 |
| DLHeavyVN | MostFree | NextVArc | MinUse | AE | 0.092 > | 1955.4 > | 508 | 0.2 |
| DLHeavyVN | MostFree | DLHeavy | Spread-2 | AE | 0.101 > | 702.7 > | 586 | 0.2 |
| DLHeavyVN | MostFree | BWHeavy | Spread-2 | AE | 0.101 > | 700.2 > | 588 | 0.2 |
| DLHeavyVN | MostFree | NextVArc | Spread-2 | AE | 0.101 > | 703.9 > | 586 | 0.2 |
| DLHeavyVN | MostFree | RelDLHeavy | Spread-2 | AE | 0.102 > | 705.8 > | 586 | 0.2 |
| DLHeavyVN | MostFree | BWHeavy | Spread-1 | AE | 0.108 > | 652.9 > | 590 | 0.2 |
| DLHeavyVN | MostFree | DLHeavy | Spread-1 | AE | 0.108 > | 659.5 > | 586 | 0.2 |

Table 6.7: Average $R_{\text{rel}}$ and $C_u$, number of valid solutions (# Valid) and average required CPU-time (t[s]) of the top 10 CH configurations according to $C_a$. The implementation emphasis is denoted by Em.

| SVN | IVN | SVA | IVA | Em. | $\mathbf{R_{rel}}$ | $\mathbf{C_a}$ | # Valid | t[s] |
|---|---|---|---|---|---|---|---|---|
| CPUHeavyVN | MostFree | NextVArc | Spread-1 | AE | 0.127 > | 263.4 = | 650 | 0.2 |
| CPUHeavyVN | MostFree | DLHeavy | Spread-1 | AE | 0.127 > | 264.0 = | 648 | 0.2 |
| CPUHeavyVN | MostFree | RelDLHeavy | Spread-1 | AE | 0.127 > | 264.7 = | 649 | 0.2 |
| CPUHeavyVN | MostFree | BWHeavy | Spread-1 | AE | 0.127 > | 265.1 = | 649 | 0.2 |
| CPUHeavyVN | MostFree | BWHeavy | Spread-2 | AE | 0.120 > | 267.9 = | 648 | 0.2 |
| CPUHeavyVN | MostFree | RelDLHeavy | Spread-2 | AE | 0.120 > | 269.1 = | 649 | 0.3 |
| CPUHeavyVN | MostFree | NextVArc | Spread-2 | AE | 0.120 > | 272.5 = | 647 | 0.2 |
| CPUHeavyVN | MostFree | DLHeavy | Spread-2 | AE | 0.120 > | 276.4 > | 647 | 0.2 |
| CPUHeavyVN | MostFree | BWHeavy | Spread-0.5 | AE | 0.137 > | 276.7 > | 638 | 0.2 |
| CPUHeavyVN | MostFree | RelDLHeavy | Spread-0.5 | AE | 0.137 > | 279.5 > | 636 | 0.3 |

why the choice of SVA strategy has nearly no influence on the outcome. When using AE, there are never a lot of implementable arcs to choose from, so the order among them is not critical. In addition, since we use a SVN strategy that focuses on virtual networks (only if all nodes from one virtual network are mapped, nodes from another one are chosen), the virtual arcs will have similar properties.

The ten best CH configurations for VNMP-S can be seen in Table 6.7. The best configuration uses CPUHeavyVN, MostFree, NextVArc, Spread-1, and AE. We will denote this configuration by CH-S. In contrast to minimizing $R_{\text{rel}}$, we can see that a lot of different configurations achieve the same level of performance. Also, the achieved average $C_a$ is far lower than those for the configurations minimizing $R_{\text{rel}}$. In contrast, the achieved $R_{\text{rel}}$ is just a bit higher. For far more

valid solutions, CH-S only has to sacrifice a bit of the cheapness of solutions found by CH-O. The importance of the different strategic choices is the same as before.
We have now defined two good CH configurations, CH-S for solving VNMP-S and CH-O for solving VNMP-O. Both use MostFree as IVN strategy, which focuses on finding valid solutions. In the context of LS and VND, we will use the CH configurations also to rebuild a small part of the solution. If we rebuild this part with a bias towards validity, we might hamper LS and VND during the search for minimal $C_u$ with neighborhood structures that remove the mapping of a node. Therefore, we define a third configuration, CH-R, which is used for cheaply rebuilding solutions. It is equivalent to CH-O, but uses CheapHost as IVN strategy. The full configuration of CH-R is DLHeavyVN, CheapHost, RelDLHeavy, MinUse, and AE.

### 6.5.2 Local Search

In this section, we will analyze the influence of the different configuration parameters of Local Search, similar to the analysis carried out in the previous section. We used CH-S and CH-O for creating initial solutions and in addition CH-R for rebuilding solutions. Together with the six presented neighborhoods, two step functions (first-improvement and best-improvement) and three neighbor priorities (None, OverloadingFirst, OnlyOverloading), this gives a total of 216 different LS configurations.
Table 6.8 shows the influence of the selected neighborhood on the performance of Local Search. For solving VNMP-O, the neighborhood structure of choice is clearly RemapVnodeTAP, the largest and most powerful neighborhood. Only for the largest instance sizes and loads, RemapSlice is better, since it can be searched much faster. For solving VNMP-S, RemapSlice is generally the best choice. For the lowest load (and all but the largest instance size), the choice of neighborhood structure does not matter, every configuration is able to find a valid solution. The worst neighborhood structure is RemapVarc, which is not surprising since it is the smallest neighborhood.
The influence of different neighbor priorities is presented in Table 6.9. The one thing that is clear from this data is that OnlyOverloading is a bad choice. It is not surprising that it does not perform well when solving VNMP-O, since LS stops once a valid solution has been found. One could have expected better performance with respect to $C_a$, but, as we have already pointed out, being able to change parts of the solution which are not directly causing overloaded substrate nodes or arcs is beneficial when trying to reduce $C_a$. The presented data shows this quite nicely. OnlyOverloading is not completely useless, we will show later that it has very attractive run-time characteristics. As for the best choice, OverloadingFirst is better than None for large instances and high loads, so focusing on overloaded parts of the substrate network first is beneficial.
In Table 6.10, we present the influence of the employed Construction Heuristics to create the initial solution and for rebuilding the solution. The employed configuration is given in pairs A-B, where A denotes the CH used to create the initial solution and B the CH for rebuilding the solution. O denotes CH-O, R denotes CH-R, and S denotes CH-S. It is clear that the idea of having a special CH configuration for the rebuilding task was successful. For solving VNMP-O and for solving VNMP-S, using CH-R achieves the best performance overall. Note how the choice of the initial solution influences the whole Local Search. When starting from a solution created by CH-O, we get the best solutions with respect to $R_{\text{rel}}$. By starting with a solution

Table 6.8: Influence of the selected neighborhood structure on the performance of LS.

| | | $\mathbf{R}_{\text{rel}}$ | | | | | | $\mathbf{C}_{\mathbf{a}}$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **Load** | $\mathbf{N}_1$ | $\mathbf{N}_2$ | $\mathbf{N}_3$ | $\mathbf{N}_4$ | $\mathbf{N}_5$ | $\mathbf{N}_6$ | $\mathbf{N}_1$ | $\mathbf{N}_2$ | $\mathbf{N}_3$ | $\mathbf{N}_4$ | $\mathbf{N}_5$ | $\mathbf{N}_6$ |
| 20 | 0.10 | 0.930 > | 0.873 > | 0.885 > | 0.898 > | 0.721 > | 0.697 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.946 > | 0.878 > | 0.886 > | 0.918 > | 0.738 > | 0.637 = | 0.0 = | 0.4 > | 0.0 = | 0.0 = | 0.4 > | 0.4 > |
| | 0.80 | 0.934 > | 0.889 > | 0.888 > | 0.901 > | 0.741 > | 0.643 = | 33.8 > | 18.9 > | 6.0 = | 33.8 > | 30.6 > | 14.9 = |
| | 1.00 | 0.938 > | 0.748 > | 0.722 > | 0.918 > | 0.756 > | 0.514 = | 230.5 > | 120.5 > | 46.5 = | 233.4 > | 191.8 > | 105.7 > |
| 30 | 0.10 | 0.889 > | 0.783 > | 0.806 > | 0.822 > | 0.679 > | 0.532 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.889 > | 0.800 > | 0.812 > | 0.825 > | 0.670 > | 0.560 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 0.888 > | 0.804 > | 0.813 > | 0.809 > | 0.668 > | 0.521 = | 0.0 = | 10.4 > | 0.0 = | 0.0 = | 10.4 > | 10.4 > |
| | 1.00 | 0.810 > | 0.691 > | 0.688 > | 0.758 > | 0.629 > | 0.496 = | 94.9 > | 100.7 > | 0.1 = | 101.3 > | 95.9 > | 100.5 > |
| 50 | 0.10 | 0.848 > | 0.748 > | 0.782 > | 0.748 > | 0.630 > | 0.502 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.909 > | 0.812 > | 0.820 > | 0.813 > | 0.674 > | 0.496 = | 0.0 = | 0.2 > | 0.0 = | 0.0 = | 0.2 > | 0.2 > |
| | 0.80 | 0.885 > | 0.796 > | 0.813 > | 0.802 > | 0.669 > | 0.490 = | 0.0 = | 4.6 > | 0.1 > | 0.0 = | 4.6 > | 4.6 > |
| | 1.00 | 0.814 > | 0.622 > | 0.606 > | 0.773 > | 0.556 > | 0.407 = | 529.4 > | 131.0 > | 10.3 = | 602.8 > | 88.9 > | 115.0 > |
| 100 | 0.10 | 0.876 > | 0.778 > | 0.804 > | 0.772 > | 0.652 > | 0.517 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.890 > | 0.790 > | 0.809 > | 0.789 > | 0.661 > | 0.481 = | 2.3 = | 7.2 > | 2.3 = | 2.3 = | 6.7 > | 7.2 > |
| | 0.80 | 0.884 > | 0.769 > | 0.784 > | 0.787 > | 0.665 > | 0.472 = | 40.2 > | 58.5 > | 2.4 = | 40.2 > | 75.3 > | 58.5 > |
| | 1.00 | 0.811 > | 0.593 > | 0.591 > | 0.760 > | 0.584 > | 0.376 = | 331.0 > | 243.5 > | 56.8 = | 341.2 > | 217.8 > | 215.7 > |
| 200 | 0.10 | 0.824 > | 0.731 > | 0.751 > | 0.663 > | 0.580 > | 0.524 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.884 > | 0.767 > | 0.794 > | 0.776 > | 0.654 > | 0.467 = | 66.5 > | 55.2 > | 45.7 > | 66.5 > | 55.0 > | 25.5 = |
| | 0.80 | 0.798 > | 0.640 > | 0.675 > | 0.717 > | 0.576 > | 0.385 = | 269.8 > | 238.1 > | 161.6 > | 267.2 > | 231.1 > | 139.0 = |
| | 1.00 | 0.803 > | 0.420 > | 0.414 > | 0.786 > | 0.490 > | 0.263 = | 1178.9 > | 710.5 > | 368.4 > | 1200.3 > | 579.9 > | 488.0 = |
| 500 | 0.10 | 0.789 > | 0.692 > | 0.736 > | 0.624 > | 0.553 > | 0.511 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.872 > | 0.742 > | 0.781 > | 0.770 > | 0.642 > | 0.450 = | 12.7 > | 9.0 > | 7.3 > | 12.5 > | 9.6 > | 5.5 = |
| | 0.80 | 0.740 > | 0.520 > | 0.541 > | 0.704 > | 0.536 > | 0.391 = | 931.7 > | 524.1 > | 366.0 = | 935.0 > | 705.9 > | 422.4 = |
| | 1.00 | 0.813 > | 0.319 = | 0.303 = | 0.803 > | 0.529 > | 0.418 > | 4949.5 > | 1927.2 = | 1296.0 = | 4891.4 > | 3086.1 > | 2586.3 > |
| 1000 | 0.10 | 0.817 > | 0.719 > | 0.776 > | 0.657 > | 0.586 > | 0.486 = | 1.1 > | 1.0 > | 0.8 > | 1.1 > | 1.0 > | 0.5 = |
| | 0.50 | 0.804 > | 0.640 > | 0.677 > | 0.724 > | 0.597 > | 0.526 = | 119.6 > | 100.8 > | 79.6 = | 121.4 > | 90.8 > | 94.3 = |
| | 0.80 | 0.760 > | 0.400 > | 0.345 = | 0.716 > | 0.465 > | 0.510 > | 1727.9 > | 852.3 > | 557.4 = | 1674.8 > | 911.7 > | 1165.2 > |
| | 1.00 | 0.809 > | 0.274 = | 0.252 = | 0.791 > | 0.462 > | 0.630 > | 7824.0 > | 2908.0 = | 2364.2 = | 7699.8 > | 4237.8 > | 7749.5 > |

Table 6.9: Influence of the selected neighbor priority on the performance of LS.

| | | $\mathbf{R_{rel}}$ | | | $\mathbf{C_a}$ | | |
|---|---|---|---|---|---|---|---|
| **Size** | **Load** | **None** | **OverloadingFirst** | **OnlyOverloading** | **None** | **OverloadingFirst** | **OnlyOverloading** |
| 20 | 0.10 | 0.779 = | 0.779 = | 0.944 > | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.773 = | 0.773 = | 0.956 > | 0.0 = | 0.0 = | 0.6 > |
| | 0.80 | 0.777 = | 0.777 = | 0.944 > | 17.2 = | 17.2 = | 34.6 > |
| | 1.00 | 0.694 = | 0.694 = | 0.910 > | 121.6 = | 121.6 = | 220.9 > |
| 30 | 0.10 | 0.675 = | 0.675 = | 0.905 > | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.685 = | 0.685 = | 0.909 > | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 0.666 > | 0.666 = | 0.918 > | 0.0 = | 0.0 = | 15.6 > |
| | 1.00 | 0.603 = | 0.603 = | 0.830 > | 37.8 = | 37.3 = | 171.6 > |
| 50 | 0.10 | 0.626 = | 0.626 = | 0.876 > | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.663 = | 0.663 = | 0.935 > | 0.0 = | 0.0 = | 0.4 > |
| | 0.80 | 0.652 = | 0.652 > | 0.925 > | 0.0 = | 0.0 = | 6.9 > |
| | 1.00 | 0.539 = | 0.539 > | 0.810 > | 190.1 = | 190.9 = | 357.7 > |
| 100 | 0.10 | 0.654 = | 0.654 = | 0.892 > | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.643 = | 0.644 = | 0.923 > | 2.2 = | 2.2 = | 9.6 > |
| | 0.80 | 0.630 > | 0.629 = | 0.922 > | 19.2 = | 19.1 = | 99.2 > |
| | 1.00 | 0.523 = | 0.522 = | 0.812 > | 148.3 = | 148.2 = | 406.6 > |
| 200 | 0.10 | 0.594 = | 0.594 = | 0.849 > | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.624 = | 0.624 = | 0.923 > | 44.6 = | 44.6 = | 68.0 > |
| | 0.80 | 0.522 = | 0.521 = | 0.852 > | 166.9 > | 166.5 = | 319.9 > |
| | 1.00 | 0.451 = | 0.450 = | 0.686 > | 573.6 > | 570.6 = | 1118.8 > |
| 500 | 0.10 | 0.571 = | 0.571 = | 0.810 > | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.609 = | 0.609 = | 0.910 > | 7.6 = | 7.5 = | 13.3 > |
| | 0.80 | 0.485 > | 0.479 = | 0.751 > | 554.3 > | 545.8 = | 842.5 > |
| | 1.00 | 0.483 > | 0.470 = | 0.639 > | 2864.5 > | 2731.5 = | 3772.3 > |
| 1000 | 0.10 | 0.589 = | 0.589 = | 0.843 > | 0.7 = | 0.7 = | 1.3 > |
| | 0.50 | 0.565 > | 0.562 = | 0.857 > | 88.4 > | 86.9 = | 128.0 > |
| | 0.80 | 0.470 > | 0.459 = | 0.669 > | 1026.8 > | 984.7 = | 1433.2 > |
| | 1.00 | 0.517 > | 0.487 = | 0.605 > | 5571.1 > | 4964.1 = | 5856.5 > |

Table 6.10: Influence of the selected CH methods for finding an initial solution and for rebuilding it on the performance of LS.

| | | $R_{rel}$ | | | | | | $C_a$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **Load** | **O-O** | **O-R** | **O-S** | **S-O** | **S-R** | **S-S** | **O-O** | **O-R** | **O-S** | **S-O** | **S-R** | **S-S** |
| 20 | 0.10 | 0.845 > | 0.712 = | 0.852 > | 0.897 > | 0.764 > | 0.935 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.866 > | 0.719 = | 0.876 > | 0.884 > | 0.739 > | 0.921 > | 0.4 > | 0.4 > | 0.4 > | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 0.837 > | 0.721 = | 0.845 > | 0.890 > | 0.767 > | 0.936 > | 25.9 > | 22.5 > | 25.9 > | 22.3 > | 18.9 = | 22.3 > |
| | 1.00 | 0.799 > | 0.691 = | 0.812 > | 0.793 > | 0.687 = | 0.815 > | 171.5 > | 162.4 > | 170.7 > | 144.1 > | 135.8 = | 143.8 > |
| 30 | 0.10 | 0.728 > | 0.581 = | 0.768 > | 0.822 > | 0.682 > | 0.930 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.735 > | 0.595 = | 0.762 > | 0.836 > | 0.702 > | 0.927 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 0.743 > | 0.613 = | 0.780 > | 0.799 > | 0.680 > | 0.886 > | 10.4 > | 10.4 > | 10.4 > | 0.0 = | 0.0 = | 0.0 = |
| | 1.00 | 0.683 > | 0.572 = | 0.724 > | 0.708 > | 0.607 > | 0.777 > | 121.5 > | 123.9 > | 110.0 > | 50.1 > | 44.8 = | 43.0 = |
| 50 | 0.10 | 0.633 > | 0.512 = | 0.694 > | 0.794 > | 0.681 > | 0.944 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.746 > | 0.601 = | 0.805 > | 0.806 > | 0.666 > | 0.900 > | 0.2 > | 0.2 > | 0.2 > | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 0.750 > | 0.607 = | 0.804 > | 0.774 > | 0.641 > | 0.881 > | 4.6 > | 4.6 > | 4.6 > | 0.0 = | 0.0 = | 0.0 = |
| | 1.00 | 0.652 > | 0.554 > | 0.717 > | 0.615 > | 0.523 = | 0.716 > | 289.1 > | 302.8 > | 270.6 > | 203.8 > | 213.0 > | 198.2 = |
| 100 | 0.10 | 0.673 > | 0.535 = | 0.731 > | 0.821 > | 0.696 > | 0.944 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.718 > | 0.563 = | 0.776 > | 0.803 > | 0.656 > | 0.905 > | 7.2 > | 6.9 > | 7.2 > | 2.3 > | 2.1 = | 2.3 > |
| | 0.80 | 0.713 > | 0.569 = | 0.784 > | 0.779 > | 0.632 > | 0.884 > | 69.0 > | 64.8 > | 69.0 > | 25.5 > | 21.2 = | 25.5 > |
| | 1.00 | 0.648 > | 0.531 > | 0.725 > | 0.610 > | 0.503 = | 0.699 > | 300.1 > | 290.5 > | 289.3 > | 179.4 > | 168.1 = | 178.8 > |
| 200 | 0.10 | 0.559 > | 0.446 = | 0.655 > | 0.772 > | 0.688 > | 0.954 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.693 > | 0.531 = | 0.774 > | 0.791 > | 0.639 > | 0.914 > | 62.4 > | 37.3 > | 62.4 > | 59.1 > | 34.1 = | 59.1 > |
| | 0.80 | 0.642 > | 0.508 = | 0.708 > | 0.652 > | 0.522 > | 0.761 > | 295.6 > | 204.9 > | 261.6 > | 208.4 > | 129.8 = | 206.5 > |
| | 1.00 | 0.605 > | 0.531 > | 0.583 > | 0.500 > | 0.441 = | 0.514 > | 994.3 > | 877.0 > | 842.4 > | 651.8 > | 541.4 = | 619.1 > |
| 500 | 0.10 | 0.495 > | 0.402 = | 0.586 > | 0.756 > | 0.700 > | 0.967 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.698 > | 0.530 = | 0.766 > | 0.764 > | 0.612 > | 0.887 > | 19.4 > | 12.6 > | 12.6 > | 4.1 > | 3.7 = | 4.1 > |
| | 0.80 | 0.627 > | 0.543 > | 0.671 > | 0.536 > | 0.446 = | 0.608 > | 750.9 > | 647.2 > | 732.4 > | 631.9 > | 494.9 = | 627.8 > |
| | 1.00 | 0.639 > | 0.582 > | 0.602 > | 0.469 > | 0.429 = | 0.464 > | 3820.8 > | 3500.8 > | 3544.5 > | 2735.3 > | 2456.7 = | 2678.6 > |
| 1000 | 0.10 | 0.547 > | 0.442 = | 0.648 > | 0.752 > | 0.691 > | 0.962 > | 0.9 > | 0.5 = | 0.9 > | 1.1 > | 0.7 > | 1.3 > |
| | 0.50 | 0.680 > | 0.517 = | 0.751 > | 0.686 > | 0.518 > | 0.815 > | 138.7 > | 111.5 > | 137.3 > | 81.9 > | 51.2 = | 86.1 > |
| | 0.80 | 0.643 > | 0.560 > | 0.587 > | 0.481 > | 0.430 = | 0.494 > | 1469.2 > | 1257.2 > | 1278.8 > | 1019.8 > | 875.7 = | 988.6 > |
| | 1.00 | 0.671 > | 0.599 > | 0.582 > | 0.480 > | 0.430 = | 0.455 > | 7143.5 > | 6556.7 > | 6323.1 > | 4463.0 > | 4044.9 = | 4252.1 > |

Table 6.11: Influence of the selected step function on the performance of LS.

| | | $\mathbf{R_{rel}}$ | | $\mathbf{C_a}$ | |
|---|---|---|---|---|---|
| **Size** | **Load** | **FirstImprove** | **BestImprove** | **FirstImprove** | **BestImprove** |
| 20 | 0.10 | 0.835 = | 0.834 = | 0.0 = | 0.0 = |
| | 0.50 | 0.835 > | 0.833 = | 0.2 = | 0.2 = |
| | 0.80 | 0.833 = | 0.832 = | 22.9 = | 23.1 > |
| | 1.00 | 0.767 = | 0.765 = | 154.6 = | 154.8 > |
| 30 | 0.10 | 0.754 > | 0.749 = | 0.0 = | 0.0 = |
| | 0.50 | 0.764 > | 0.755 = | 0.0 = | 0.0 = |
| | 0.80 | 0.753 > | 0.748 = | 5.2 = | 5.2 = |
| | 1.00 | 0.686 > | 0.671 = | 86.4 > | 78.1 = |
| 50 | 0.10 | 0.713 > | 0.706 = | 0.0 = | 0.0 = |
| | 0.50 | 0.757 > | 0.751 = | 0.1 = | 0.1 = |
| | 0.80 | 0.747 > | 0.738 = | 2.3 = | 2.3 = |
| | 1.00 | 0.635 > | 0.624 = | 253.4 > | 239.0 = |
| 100 | 0.10 | 0.736 > | 0.731 = | 0.0 = | 0.0 = |
| | 0.50 | 0.741 > | 0.732 = | 4.7 = | 4.7 = |
| | 0.80 | 0.732 > | 0.722 = | 45.8 = | 45.8 = |
| | 1.00 | 0.626 > | 0.612 = | 236.1 > | 232.6 = |
| 200 | 0.10 | 0.682 > | 0.676 = | 0.0 = | 0.0 = |
| | 0.50 | 0.727 > | 0.721 = | 52.4 = | 52.4 = |
| | 0.80 | 0.639 > | 0.625 = | 219.0 > | 216.6 = |
| | 1.00 | 0.541 > | 0.517 = | 788.1 > | 720.5 = |
| 500 | 0.10 | 0.654 > | 0.648 = | 0.0 = | 0.0 = |
| | 0.50 | 0.712 > | 0.707 = | 9.6 > | 9.3 = |
| | 0.80 | 0.590 > | 0.554 = | 676.0 > | 619.1 = |
| | 1.00 | 0.560 > | 0.501 = | 3366.8 > | 2878.7 = |
| 1000 | 0.10 | 0.677 > | 0.670 = | 0.9 = | 0.9 = |
| | 0.50 | 0.678 > | 0.645 = | 108.4 > | 93.8 = |
| | 0.80 | 0.563 > | 0.502 = | 1236.6 > | 1059.8 = |
| | 1.00 | 0.544 > | 0.529 = | 5506.2 > | 5421.6 = |

created by CH-S, we get the best solutions with respect to $C_a$. The exception to this rule are instances of high load. Building the initial solution with CH-S results almost universally in better $R_{\text{rel}}$, building it with CH-O is in one case significantly better than CH-S when trying to solve VNMP-S. It is surprising that CH-S is worse than CH-R as rebuilding heuristic when minimizing $C_a$, although the difference is small. The reason could be that, when used as rebuilding strategy, it repeats the same mistakes it made when creating the initial solution.

The influence of the step function on the performance of Local Search is shown in Table 6.11. For all but the smallest instances, best-improvement (labeled BestImprove) is significantly better than first-improvement (labeled FirstImprove) when trying to find valid solutions with low $C_u$, but the difference is quite small. First-improvement can keep up much longer when trying to minimize $C_a$, but is worse for high loads and large instances. Later we will show the difference in run-time between first-improvement and best-improvement.

Table 6.12: Average $R_{\text{rel}}$ and $C_u$, number of valid solutions (# Valid) and average required CPU-time (t[s]) of the top 10 LS Configurations according to $R_{\text{rel}}$.

| Neighborhood | Step-Function | Priority | Init. | Reb. | $\mathbf{R_{rel}}$ | $\mathbf{C_a}$ | # Valid | t[s] |
|---|---|---|---|---|---|---|---|---|
| ClearSnode | BestImprove | OverloadingFirst | CH-S | CH-R | 0.092 = | 202.6 > | 703 | 115.0 |
| ClearSnode | BestImprove | None | CH-S | CH-R | 0.092 = | 202.8 > | 703 | 114.8 |
| ClearSnode | BestImprove | OverloadingFirst | CH-O | CH-R | 0.093 = | 203.2 > | 695 | 121.6 |
| ClearSnode | BestImprove | None | CH-O | CH-R | 0.093 = | 203.2 > | 695 | 121.5 |
| ClearSnode | FirstImprove | OverloadingFirst | CH-O | CH-R | 0.125 > | 192.0 > | 696 | 59.0 |
| ClearSnode | FirstImprove | OverloadingFirst | CH-S | CH-R | 0.126 > | 193.7 > | 701 | 53.9 |
| ClearSnode | FirstImprove | None | CH-O | CH-R | 0.137 > | 255.2 > | 693 | 66.3 |
| ClearSnode | FirstImprove | None | CH-S | CH-R | 0.142 > | 252.4 > | 694 | 54.2 |
| RemapVnodeTAP | BestImprove | OverloadingFirst | CH-O | CH-O | 0.234 > | 540.8 > | 746 | 265.8 |
| RemapVnodeTAP | BestImprove | None | CH-O | CH-O | 0.234 > | 538.0 > | 746 | 266.4 |

Table 6.13: Average $R_{\text{rel}}$ and $C_u$, number of valid solutions (# Valid) and average required CPU-time (t[s]) of the top 10 LS Configurations according to $C_a$.

| Neighborhood | Step-Function | Priority | Init. | Reb. | $\mathbf{R_{rel}}$ | $\mathbf{C_a}$ | # Valid | t[s] |
|---|---|---|---|---|---|---|---|---|
| RemapVnode | BestImprove | None | CH-S | CH-R | 0.428 > | 64.5 = | 714 | 27.0 |
| RemapVnode | BestImprove | OverloadingFirst | CH-S | CH-R | 0.429 > | 64.8 = | 714 | 27.0 |
| RemapVnode | FirstImprove | OverloadingFirst | CH-S | CH-R | 0.442 > | 77.3 > | 708 | 8.9 |
| RemapVnode | FirstImprove | None | CH-S | CH-R | 0.442 > | 77.7 > | 709 | 9.8 |
| RemapVnode | BestImprove | OverloadingFirst | CH-O | CH-R | 0.343 > | 86.4 > | 706 | 35.4 |
| RemapVnode | BestImprove | None | CH-O | CH-R | 0.343 > | 86.4 > | 706 | 35.5 |
| RemapVnode | FirstImprove | None | CH-O | CH-R | 0.362 > | 100.7 > | 696 | 14.2 |
| RemapSlice | BestImprove | None | CH-S | CH-R | 0.497 > | 108.5 > | 703 | 4.6 |
| RemapSlice | BestImprove | OverloadingFirst | CH-S | CH-R | 0.497 > | 108.5 > | 703 | 4.5 |
| RemapVnode | FirstImprove | OverloadingFirst | CH-O | CH-R | 0.361 > | 109.0 > | 698 | 11.9 |

As in the previous section, we show the 10 best LS configurations with respect to the average $R_{\text{rel}}$ in Table 6.12. In comparison with the best CH configurations with respect to $C_a$, the best LS configurations according to the $R_{\text{rel}}$ beat them with respect to $C_a$ and also in terms of valid solutions. Of course, this has a substantial cost in terms of run-time. It can be observed that with all other settings being equal, there is no significant difference between OverloadingFirst and None. Also, the initialization CH is far less important for the best configurations than in the average case. The rebuilding CH however still has to be CH-R for the best performance, notice the large increase in relative rank for the configurations using CH-O. Observe how switching from BestImprove to FirstImprove increases the $R_{\text{rel}}$ by 30%, but halves the required run-time. Using RemapVnodeTAP instead of ClearSnode (which by the way was not indicated to be a promising neighborhood in the slightest) doubles the required run-time.
Table 6.13 shows the 10 best LS configurations with respect to the average $C_a$. Again we can observe that changing the neighbor priority has no significant effect on the final result (as long as we do not use OnlyOverloading). Using first-improvement reduces the required run-time to a

third for a slight increase in average $C_a$. The best configurations for solving VNMP-S are generally faster than those for VNMP-O. One important thing to note is that the best configuration with respect to average $C_a$ does not solve the most instances to validity. The best LS configuration is able to solve 764 out of the 840 instances. It uses RemapVnodeTAP, best-improvement, OverloadingFirst, CH-S for initialization, and CH-R for rebuilding.
The best CH configurations have shown that the mapping strategy for virtual nodes is essential. The same can be observed here. The best LS configuration with respect to $R_{\text{rel}}$ tries to clear substrate nodes (by removing the virtual nodes) to reduce $C_u$. To minimize $C_a$, the more conservative approach of removing a single virtual node achieves the best results.
We will denote the best LS configuration for solving VNMP-O (ClearSnode, best-improvement, OverloadingFirst, CH-S, and CH-R) by LS-O. The best LS configuration with respect to VNMP-S (RemapVnode, best-improvement, None, CH-S, and CH-R) will be called LS-S.

### 6.5.3 Variable Neighborhood Descent

For Variable Neighborhood Descent, we tested all combinations of the twelve neighborhood structure configurations, two step-functions, two initialization CHs and three rebuilding CHs, yielding in total 144 VND algorithms. The different step-functions were applied to all neighborhoods simultaneously, so either all neighborhoods were searched by first-improvement or all were searched by best-improvement.
Table 6.14 shows the influence of the different neighborhood configurations on the achieved average $R_{\text{rel}}$. For all instances up to and including size 200, $C_4$ (RComplete) is the best neighborhood structure configuration. Very similar performance levels are achieved by $C_1$ (All), $C_3$ (Complete), $C_9$ (ImprovCompB) and $C_{10}$ (ImprovCompC), which are all configurations that utilize the RemapVnodeTAP neighborhood structure. For the instances of the two largest size classes, reducing the number of employed neighborhood structures (ImprovCompB) is required to achieve the best levels of performance. A key factor is also that ImprovCompB uses RemapVnodeTAP as last neighborhood structure and not as first, as is done by RComplete, because RemapVnodeTAP requires a lot of time for large instances. This is because RemapVnodeTAP explicitly tries to map all virtual nodes to all possible mapping targets, and we know from Chapter 5 that for the largest instances we can expect 700 virtual nodes with an average of 50 allowed mapping targets. The configurations that use neighborhood structures in the Only-Overloading setting ($C_2$, $C_7$, and $C_8$) produce very bad results with respect to $R_{\text{rel}}$, as could be expected.
How different neighborhood configurations influence the achieved performance of VND with respect to average $C_a$ is shown in Table 6.15. It is striking that the choice of configuration basically has no influence on the performance up to instance size 100. In addition, every configuration choice is able to solve every instance of size 30, with the exception of $C_{11}$ (OnlyClear) and full load. It is interesting that instances of size 20 seem to be harder to solve than instances of size 30 or 50. For larger instances, configurations $C_1$, $C_3$, $C_5$, $C_9$, and $C_{10}$ achieve the best performance. Note that the configurations that reverse the order of neighborhood structures ($C_4$, $C_6$) perform far worse.
The influence of the CH choice on the performance of VND is presented in Table 6.16. The CH choice is encoded in the same way as for LS, i.e., S-R means that CH-S was used to create the

Table 6.14: Influence of the selected VND neighborhood structure configuration on the performance of VND with respect to average $R_{\text{rel}}$.

| | | $\mathbf{R}_{\text{rel}}$ | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **Load** | $\mathbf{C}_1$ | $\mathbf{C}_2$ | $\mathbf{C}_3$ | $\mathbf{C}_4$ | $\mathbf{C}_5$ | $\mathbf{C}_6$ | $\mathbf{C}_7$ | $\mathbf{C}_8$ | $\mathbf{C}_9$ | $\mathbf{C}_{10}$ | $\mathbf{C}_{11}$ | $\mathbf{C}_{12}$ |
| 20 | 0.10 | 0.454 = | 0.951 > | 0.454 = | 0.447 = | 0.634 > | 0.633 > | 0.951 > | 0.951 > | 0.454 = | 0.458 > | 0.640 > | 0.648 > |
| | 0.50 | 0.444 > | 0.959 > | 0.444 > | 0.443 = | 0.612 > | 0.608 > | 0.959 > | 0.960 > | 0.444 > | 0.440 = | 0.682 > | 0.682 > |
| | 0.80 | 0.473 > | 0.958 > | 0.470 > | 0.431 = | 0.613 > | 0.588 > | 0.958 > | 0.958 > | 0.469 > | 0.476 > | 0.707 > | 0.704 > |
| | 1.00 | 0.406 > | 0.889 > | 0.425 > | 0.381 = | 0.545 > | 0.502 > | 0.889 > | 0.891 > | 0.414 > | 0.426 > | 0.759 > | 0.654 > |
| 30 | 0.10 | 0.324 = | 0.927 > | 0.324 = | 0.322 = | 0.424 > | 0.413 > | 0.927 > | 0.927 > | 0.324 = | 0.333 > | 0.610 > | 0.611 > |
| | 0.50 | 0.408 = | 0.941 > | 0.408 = | 0.407 = | 0.534 > | 0.536 > | 0.941 > | 0.941 > | 0.409 = | 0.411 = | 0.651 > | 0.643 > |
| | 0.80 | 0.431 > | 0.961 > | 0.431 > | 0.421 = | 0.576 > | 0.567 > | 0.961 > | 0.961 > | 0.431 > | 0.440 > | 0.673 > | 0.665 > |
| | 1.00 | 0.394 > | 0.890 > | 0.391 > | 0.379 = | 0.509 > | 0.496 > | 0.890 > | 0.901 > | 0.387 = | 0.399 > | 0.667 > | 0.610 > |
| 50 | 0.10 | 0.343 > | 0.925 > | 0.343 > | 0.325 = | 0.427 > | 0.415 > | 0.925 > | 0.925 > | 0.343 > | 0.350 > | 0.607 > | 0.607 > |
| | 0.50 | 0.423 > | 0.962 > | 0.423 > | 0.407 = | 0.557 > | 0.536 > | 0.962 > | 0.960 > | 0.424 > | 0.432 > | 0.670 > | 0.671 > |
| | 0.80 | 0.460 > | 0.964 > | 0.460 > | 0.450 = | 0.603 > | 0.590 > | 0.964 > | 0.965 > | 0.463 > | 0.474 > | 0.705 > | 0.694 > |
| | 1.00 | 0.413 > | 0.905 > | 0.422 > | 0.364 = | 0.539 > | 0.489 > | 0.905 > | 0.901 > | 0.372 = | 0.381 > | 0.681 > | 0.621 > |
| 100 | 0.10 | 0.422 > | 0.956 > | 0.422 > | 0.400 = | 0.518 > | 0.498 > | 0.956 > | 0.956 > | 0.424 > | 0.426 > | 0.676 > | 0.684 > |
| | 0.50 | 0.485 > | 0.965 > | 0.485 > | 0.474 = | 0.637 > | 0.615 > | 0.965 > | 0.966 > | 0.488 > | 0.501 > | 0.697 > | 0.704 > |
| | 0.80 | 0.473 > | 0.952 > | 0.468 > | 0.457 = | 0.617 > | 0.595 > | 0.952 > | 0.956 > | 0.471 > | 0.487 > | 0.687 > | 0.681 > |
| | 1.00 | 0.408 > | 0.887 > | 0.382 > | 0.363 = | 0.500 > | 0.476 > | 0.887 > | 0.895 > | 0.397 > | 0.426 > | 0.685 > | 0.637 > |
| 200 | 0.10 | 0.417 > | 0.943 > | 0.417 > | 0.387 = | 0.505 > | 0.478 > | 0.943 > | 0.943 > | 0.418 > | 0.414 > | 0.660 > | 0.664 > |
| | 0.50 | 0.453 > | 0.972 > | 0.453 > | 0.439 = | 0.602 > | 0.580 > | 0.972 > | 0.974 > | 0.455 > | 0.475 > | 0.712 > | 0.704 > |
| | 0.80 | 0.401 > | 0.917 > | 0.391 > | 0.357 = | 0.531 > | 0.483 > | 0.917 > | 0.928 > | 0.391 > | 0.422 > | 0.689 > | 0.665 > |
| | 1.00 | 0.307 > | 0.810 > | 0.295 > | 0.282 = | 0.393 > | 0.372 > | 0.813 > | 0.838 > | 0.295 > | 0.361 > | 0.795 > | 0.593 > |
| 500 | 0.10 | 0.417 > | 0.938 > | 0.417 > | 0.402 = | 0.497 > | 0.478 > | 0.938 > | 0.938 > | 0.416 > | 0.421 > | 0.699 > | 0.718 > |
| | 0.50 | 0.428 = | 0.971 > | 0.430 = | 0.518 > | 0.548 > | 0.615 > | 0.971 > | 0.972 > | 0.431 = | 0.439 > | 0.732 > | 0.720 > |
| | 0.80 | 0.350 > | 0.828 > | 0.339 > | 0.508 > | 0.351 > | 0.510 > | 0.829 > | 0.841 > | 0.320 = | 0.373 > | 0.708 > | 0.617 > |
| | 1.00 | 0.310 > | 0.693 > | 0.298 > | 0.658 > | 0.302 > | 0.658 > | 0.704 > | 0.694 > | 0.286 = | 0.312 > | 0.848 > | 0.515 > |
| 1000 | 0.10 | 0.434 = | 0.958 > | 0.435 = | 0.458 = | 0.519 > | 0.547 > | 0.958 > | 0.958 > | 0.438 = | 0.434 = | 0.770 > | 0.781 > |
| | 0.50 | 0.400 > | 0.901 > | 0.392 > | 0.601 > | 0.425 > | 0.602 > | 0.901 > | 0.907 > | 0.375 = | 0.414 > | 0.669 > | 0.640 > |
| | 0.80 | 0.347 > | 0.670 > | 0.341 > | 0.701 > | 0.344 > | 0.703 > | 0.694 > | 0.641 > | 0.320 = | 0.386 > | 0.719 > | 0.495 > |
| | 1.00 | 0.264 = | 0.520 > | 0.343 > | 0.903 > | 0.331 > | 0.906 > | 0.607 > | 0.536 > | 0.305 > | 0.392 > | 0.870 > | 0.394 > |

Table 6.15: Influence of the selected VND neighborhood structure configuration on the performance of VND with respect to average $C_a$.

| | | $\mathbf{C_a}$ | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **Load** | $\mathbf{C}_1$ | $\mathbf{C}_2$ | $\mathbf{C}_3$ | $\mathbf{C}_4$ | $\mathbf{C}_5$ | $\mathbf{C}_6$ | $\mathbf{C}_7$ | $\mathbf{C}_8$ | $\mathbf{C}_9$ | $\mathbf{C}_{10}$ | $\mathbf{C}_{11}$ | $\mathbf{C}_{12}$ |
| 20 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 6.0 > | 6.0 > | 6.0 > | 0.0 = | 6.0 > | 0.0 = | 6.0 > | 6.0 > | 6.0 > | 6.0 > | 23.6 > | 6.0 > |
| | 1.00 | 20.7 = | 46.4 > | 29.5 > | 24.2 = | 29.5 > | 24.2 = | 46.4 > | 46.5 > | 29.5 > | 20.7 = | 151.4 > | 46.3 > |
| 30 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 1.00 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 49.1 > | 0.0 = |
| 50 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 1.00 | 1.8 = | 20.2 > | 2.9 > | 2.9 > | 2.9 > | 2.9 > | 20.7 > | 10.6 > | 2.9 > | 1.6 = | 76.3 > | 21.1 > |
| 100 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 1.6 = | 2.3 > | 1.6 = | 1.6 = | 2.3 > | 2.3 > | 2.3 > | 2.3 > | 1.6 = | 1.6 = | 1.6 = | 1.6 = |
| | 0.80 | 2.3 = | 2.3 = | 2.3 = | 2.3 = | 2.3 = | 2.3 = | 2.3 = | 2.3 = | 2.3 = | 2.3 = | 27.6 > | 2.3 = |
| | 1.00 | 9.1 = | 59.3 > | 9.1 = | 8.2 = | 9.1 = | 8.2 = | 59.3 > | 56.6 > | 12.9 > | 9.5 > | 111.0 > | 42.7 > |
| 200 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.0 = | 45.7 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 45.7 > | 45.7 > | 0.0 = | 0.0 = | 44.4 > | 44.4 > |
| | 0.80 | 0.9 = | 160.2 > | 0.0 = | 0.3 = | 0.3 = | 1.4 > | 160.4 > | 158.3 > | 0.0 = | 0.0 = | 153.6 > | 144.3 > |
| | 1.00 | 4.1 = | 348.1 > | 4.0 = | 10.2 > | 4.0 = | 11.3 > | 357.6 > | 334.9 > | 4.6 > | 5.6 > | 478.2 > | 279.2 > |
| 500 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.0 = | 6.9 > | 0.0 = | 0.9 = | 0.0 = | 1.6 = | 6.9 > | 7.2 > | 0.0 = | 0.0 = | 6.4 > | 5.5 > |
| | 0.80 | 97.0 > | 346.7 > | 82.6 > | 185.3 > | 47.6 = | 185.1 > | 347.2 > | 346.1 > | 71.1 = | 103.4 > | 596.7 > | 285.4 > |
| | 1.00 | 392.3 = | 1202.6 > | 396.6 = | 2393.8 > | 380.4 = | 2397.3 > | 1240.5 > | 1179.1 > | 388.9 = | 385.3 = | 2838.9 > | 847.2 > |
| 1000 | 0.10 | 0.0 = | 0.7 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.7 > | 0.7 > | 0.0 = | 0.0 = | 0.7 > | 0.7 > |
| | 0.50 | 24.3 > | 72.9 > | 24.1 > | 68.1 > | 18.8 = | 68.2 > | 72.9 > | 72.4 > | 23.8 = | 23.6 = | 70.5 > | 61.9 > |
| | 0.80 | 228.2 = | 524.1 > | 265.6 = | 1106.9 > | 264.9 = | 1111.2 > | 606.6 > | 478.9 > | 229.5 = | 227.4 = | 729.3 > | 416.6 > |
| | 1.00 | 999.7 = | 1794.6 > | 1288.1 > | 10411.5 > | 1232.8 > | 10437.2 > | 2176.9 > | 1913.7 > | 1190.2 > | 1447.0 > | 5547.8 > | 1480.4 > |

Table 6.16: Influence of the selected CH methods for finding an initial solution and for rebuilding it on the performance of VND.

| | | $R_{rel}$ | | | | | | $C_a$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **Load** | **O-O** | **O-R** | **O-S** | **S-O** | **S-R** | **S-S** | **O-O** | **O-R** | **O-S** | **S-O** | **S-R** | **S-S** |
| 20 | 0.10 | 0.736 > | 0.388 = | 0.739 > | 0.749 > | 0.395 = | 0.829 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.732 > | 0.417 = | 0.748 > | 0.739 > | 0.421 = | 0.781 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 0.724 > | 0.441 = | 0.754 > | 0.726 > | 0.447 = | 0.811 > | 7.3 = | 4.8 = | 7.3 = | 7.3 = | 4.8 = | 7.3 = |
| | 1.00 | 0.661 > | 0.440 > | 0.704 > | 0.657 > | 0.425 = | 0.704 > | 45.3 > | 38.5 = | 45.1 > | 45.0 > | 38.7 = | 45.0 > |
| 30 | 0.10 | 0.556 > | 0.328 = | 0.653 > | 0.583 > | 0.372 > | 0.742 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.642 > | 0.386 = | 0.710 > | 0.667 > | 0.416 > | 0.794 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 0.676 > | 0.411 = | 0.748 > | 0.691 > | 0.427 > | 0.806 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 1.00 | 0.615 > | 0.398 = | 0.693 > | 0.616 > | 0.395 = | 0.739 > | 5.0 = | 4.0 = | 2.5 = | 6.5 = | 3.8 = | 2.9 = |
| 50 | 0.10 | 0.527 > | 0.341 = | 0.658 > | 0.560 > | 0.367 > | 0.815 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.655 > | 0.421 = | 0.769 > | 0.648 > | 0.419 = | 0.802 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 0.708 > | 0.450 > | 0.798 > | 0.693 > | 0.434 = | 0.812 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 1.00 | 0.633 > | 0.412 = | 0.723 > | 0.613 > | 0.400 = | 0.714 > | 13.9 = | 17.9 > | 9.6 = | 11.2 > | 19.8 > | 11.1 > |
| 100 | 0.10 | 0.608 > | 0.405 = | 0.760 > | 0.624 > | 0.428 > | 0.845 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.708 > | 0.443 = | 0.817 > | 0.713 > | 0.452 > | 0.857 > | 2.3 > | 1.0 = | 2.3 > | 2.3 > | 1.0 = | 2.3 > |
| | 0.80 | 0.695 > | 0.439 = | 0.819 > | 0.681 > | 0.434 = | 0.832 > | 5.5 > | 2.3 = | 5.5 > | 5.5 > | 2.3 = | 5.5 > |
| | 1.00 | 0.630 > | 0.398 > | 0.736 > | 0.598 > | 0.387 = | 0.721 > | 37.0 > | 26.2 = | 37.2 > | 35.9 > | 25.3 = | 35.9 > |
| 200 | 0.10 | 0.548 > | 0.393 = | 0.760 > | 0.572 > | 0.415 > | 0.906 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.677 > | 0.420 = | 0.829 > | 0.682 > | 0.429 > | 0.858 > | 27.7 > | 1.1 = | 27.7 > | 27.7 > | 1.1 = | 27.7 > |
| | 0.80 | 0.631 > | 0.385 = | 0.772 > | 0.611 > | 0.382 = | 0.766 > | 104.6 > | 11.4 > | 89.9 > | 89.1 > | 5.7 = | 89.1 > |
| | 1.00 | 0.564 > | 0.381 > | 0.659 > | 0.508 > | 0.342 = | 0.622 > | 230.1 > | 91.8 > | 176.5 > | 186.6 > | 69.2 = | 166.6 > |
| 500 | 0.10 | 0.522 > | 0.397 = | 0.786 > | 0.561 > | 0.437 > | 0.938 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.678 > | 0.429 = | 0.843 > | 0.664 > | 0.426 = | 0.848 > | 8.6 > | 1.7 = | 3.8 > | 1.3 > | 0.9 = | 1.3 > |
| | 0.80 | 0.617 > | 0.408 > | 0.727 > | 0.545 > | 0.349 = | 0.643 > | 293.5 > | 135.6 > | 292.3 > | 274.8 > | 89.0 = | 261.8 > |
| | 1.00 | 0.626 > | 0.468 > | 0.624 > | 0.515 > | 0.363 = | 0.544 > | 1640.4 > | 1175.1 > | 1643.4 > | 1020.1 > | 553.1 = | 989.4 > |
| 1000 | 0.10 | 0.546 > | 0.432 = | 0.840 > | 0.594 > | 0.490 > | 0.943 > | 0.4 > | 0.0 = | 0.4 > | 0.4 > | 0.0 = | 0.4 > |
| | 0.50 | 0.635 > | 0.411 = | 0.763 > | 0.623 > | 0.400 > | 0.781 > | 70.4 > | 36.1 > | 84.3 > | 51.1 > | 9.3 = | 49.6 > |
| | 0.80 | 0.630 > | 0.428 > | 0.632 > | 0.521 > | 0.384 = | 0.586 > | 744.7 > | 436.9 > | 715.7 > | 467.0 > | 264.0 = | 466.3 > |
| | 1.00 | 0.696 > | 0.500 > | 0.588 > | 0.519 > | 0.402 = | 0.481 > | 4975.9 > | 4211.0 > | 4559.6 > | 2275.0 > | 1861.7 = | 2076.7 > |

Table 6.17: Influence of the selected step function on the performance of VND.

| | | $\mathbf{R}_{\text{rel}}$ | | $\mathbf{C}_{\mathbf{a}}$ | |
|---|---|---|---|---|---|
| **Size** | **Load** | **FirstImprove** | **BestImprove** | **FirstImprove** | **BestImprove** |
| 20 | 0.10 | 0.645 > | 0.634 = | 0.0 = | 0.0 = |
| | 0.50 | 0.646 > | 0.633 = | 0.0 = | 0.0 = |
| | 0.80 | 0.653 > | 0.648 = | 6.4 = | 6.5 = |
| | 1.00 | 0.602 > | 0.595 = | 44.3 = | 41.6 = |
| 30 | 0.10 | 0.546 > | 0.532 = | 0.0 = | 0.0 = |
| | 0.50 | 0.615 > | 0.590 = | 0.0 = | 0.0 = |
| | 0.80 | 0.634 > | 0.619 = | 0.0 = | 0.0 = |
| | 1.00 | 0.584 > | 0.568 = | 3.9 = | 4.3 = |
| 50 | 0.10 | 0.555 > | 0.535 = | 0.0 = | 0.0 = |
| | 0.50 | 0.629 > | 0.609 = | 0.0 = | 0.0 = |
| | 0.80 | 0.659 > | 0.640 = | 0.0 = | 0.0 = |
| | 1.00 | 0.578 = | 0.588 = | 12.0 = | 15.8 > |
| 100 | 0.10 | 0.618 > | 0.605 = | 0.0 = | 0.0 = |
| | 0.50 | 0.680 > | 0.650 = | 1.9 = | 1.9 = |
| | 0.80 | 0.663 > | 0.637 = | 4.4 = | 4.4 = |
| | 1.00 | 0.592 > | 0.565 = | 33.2 > | 32.6 = |
| 200 | 0.10 | 0.605 > | 0.593 = | 0.0 = | 0.0 = |
| | 0.50 | 0.660 > | 0.638 = | 18.8 = | 18.8 = |
| | 0.80 | 0.600 > | 0.582 = | 65.3 = | 64.6 = |
| | 1.00 | 0.524 > | 0.502 = | 159.7 > | 147.3 = |
| 500 | 0.10 | 0.619 > | 0.594 = | 0.0 = | 0.0 = |
| | 0.50 | 0.654 > | 0.642 = | 3.2 > | 2.7 = |
| | 0.80 | 0.590 > | 0.506 = | 284.1 > | 164.9 = |
| | 1.00 | 0.596 > | 0.450 = | 1610.2 > | 730.3 = |
| 1000 | 0.10 | 0.659 > | 0.623 = | 0.3 = | 0.3 = |
| | 0.50 | 0.641 > | 0.563 = | 70.8 > | 29.5 = |
| | 0.80 | 0.612 > | 0.449 = | 712.0 > | 319.5 = |
| | 1.00 | 0.562 > | 0.500 = | 3633.1 > | 3020.2 = |

initial solution and the solutions are rebuilt by CH-R. As with LS, we can see that using CH-R for rebuilding is essential for good performance. When solving VNMP-O, it can be observed that the best choice of CH configuration for initialization depends on the instance load. For low loads, initializing with CH-O is better, for higher loads an initial VNMP solution created by CH-S leads to the best results. When minimizing $C_a$, using CH-S for the initial solution is almost always the best choice.

Table 6.17 shows the influence of the selected step function on the performance of VND. The results are basically the same as for LS. For better ranks, using best-improvement is essential. When trying to minimize the additional resource cost, first-improvement can keep up until size 200.

One parameter that is always of interest for VND is the contribution of the different neighborhood structures to the final solution. In Table 6.18 we present the fraction of successful executions of a neighborhood structure, i.e., the number of times a neighborhood structure found an improvement divided by the number of its executions. Note that this data is based on all

Table 6.18: Fraction of improvements found per execution of the different neighborhood structures used for VND

| | | Fraction of Improvements [%] | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **Load** | $\mathbf{N'_1}$ | $\mathbf{N'_2}$ | $\mathbf{N'_3}$ | $\mathbf{N'_4}$ | $\mathbf{N'_5}$ | $\mathbf{N'_6}$ | $\mathbf{N_1}$ | $\mathbf{N_2}$ | $\mathbf{N_3}$ | $\mathbf{N_4}$ | $\mathbf{N_5}$ | $\mathbf{N_6}$ |
| 20 | 0.10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.9 | 21.6 | 3.1 | 9.7 | 39.3 | 33.8 |
| | 0.50 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 3.8 | 23.7 | 0.8 | 6.1 | 47.2 | 52.1 |
| | 0.80 | 0.0 | 0.0 | 1.6 | 0.0 | 0.0 | 0.0 | 5.6 | 16.6 | 3.0 | 11.2 | 49.8 | 46.1 |
| | 1.00 | 2.7 | 3.3 | 9.1 | 0.0 | 0.0 | 0.0 | 0.6 | 22.8 | 11.5 | 9.7 | 40.5 | 52.1 |
| 30 | 0.10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 7.5 | 30.9 | 0.9 | 18.8 | 42.6 | 48.3 |
| | 0.50 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 9.0 | 30.8 | 0.9 | 17.8 | 47.4 | 61.4 |
| | 0.80 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 7.2 | 24.4 | 3.6 | 21.4 | 49.3 | 63.7 |
| | 1.00 | 1.9 | 1.6 | 1.7 | 0.0 | 0.0 | 0.0 | 6.7 | 28.2 | 11.2 | 21.2 | 47.0 | 62.0 |
| 50 | 0.10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 8.2 | 34.3 | 3.0 | 27.5 | 41.7 | 59.3 |
| | 0.50 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 6.2 | 27.6 | 1.1 | 26.2 | 48.0 | 74.0 |
| | 0.80 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 9.6 | 32.2 | 1.4 | 21.5 | 46.5 | 78.5 |
| | 1.00 | 32.9 | 6.3 | 1.2 | 0.0 | 0.0 | 0.0 | 12.1 | 37.4 | 12.7 | 22.7 | 48.7 | 75.1 |
| 100 | 0.10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 9.0 | 35.0 | 2.0 | 35.4 | 45.4 | 75.8 |
| | 0.50 | 0.0 | 0.0 | 0.2 | 0.0 | 0.0 | 0.0 | 5.6 | 30.8 | 1.1 | 30.2 | 51.6 | 85.0 |
| | 0.80 | 0.0 | 0.0 | 0.4 | 0.0 | 0.0 | 0.0 | 5.5 | 32.4 | 2.0 | 25.3 | 52.8 | 86.0 |
| | 1.00 | 1.4 | 1.0 | 2.2 | 0.0 | 0.0 | 0.0 | 8.4 | 33.9 | 7.6 | 24.3 | 50.7 | 86.9 |
| 200 | 0.10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 13.2 | 33.7 | 1.4 | 53.1 | 33.6 | 76.6 |
| | 0.50 | 0.0 | 0.0 | 0.2 | 0.0 | 0.0 | 0.0 | 7.9 | 38.8 | 1.0 | 36.4 | 38.3 | 91.1 |
| | 0.80 | 0.7 | 0.3 | 0.8 | 0.0 | 0.0 | 0.1 | 8.5 | 38.3 | 4.0 | 27.8 | 39.4 | 93.9 |
| | 1.00 | 8.2 | 4.2 | 1.4 | 0.0 | 0.0 | 0.1 | 8.9 | 41.1 | 10.2 | 23.3 | 38.4 | 94.7 |
| 500 | 0.10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 15.5 | 35.3 | 0.6 | 59.6 | 22.9 | 81.5 |
| | 0.50 | 0.1 | 0.0 | 0.4 | 0.0 | 0.0 | 0.0 | 9.4 | 39.7 | 0.7 | 40.3 | 33.6 | 94.2 |
| | 0.80 | 1.5 | 1.2 | 1.6 | 0.0 | 0.0 | 0.0 | 10.8 | 43.3 | 6.8 | 37.4 | 45.7 | 98.0 |
| | 1.00 | 9.2 | 11.2 | 4.7 | 0.2 | 0.0 | 0.2 | 11.4 | 53.3 | 24.2 | 39.9 | 56.4 | 99.8 |
| 1000 | 0.10 | 0.1 | 0.0 | 0.2 | 0.0 | 0.0 | 0.0 | 17.6 | 34.5 | 0.2 | 60.7 | 16.7 | 85.9 |
| | 0.50 | 0.6 | 0.2 | 0.8 | 0.0 | 0.0 | 0.0 | 13.4 | 45.5 | 2.6 | 56.5 | 56.4 | 98.2 |
| | 0.80 | 6.6 | 3.8 | 2.6 | 0.0 | 0.0 | 0.0 | 17.0 | 57.2 | 17.7 | 62.2 | 85.6 | 100.0 |
| | 1.00 | 24.8 | 23.0 | 6.2 | 0.4 | 0.3 | 1.0 | 19.6 | 72.7 | 40.0 | 76.7 | 96.2 | 100.0 |

tested VND configurations. Not all neighborhood structures are present in all configurations and are also executed in different orders so what we show here is the average behaviour of a neighborhood structure.

The first thing noticeable from Table 6.18 is that the neighborhood structures used with Only-Overloading (marked with a prime) seldom contribute any improvements. Only for the highest loads and for the largest instance sizes do they improve solutions with a significant probability. The neighborhood structures that do not restrict themselves to parts of the substrate network which are overloaded have a far better success rate. However, there are still neighborhood structures that perform better than others. Just implementing virtual arcs in another way ($N_1$) does not find improvements very often. Only for instances of size 1000 is the success rate consistently above 10%. Reimplementing complete virtual networks ($N_3$) achieves this success rate only for instances of highest load. This is also the neighborhood structure that is most strongly influenced by the load of an instance. The best performance is consistently achieved by $N_6$ (RemapVnodeTAP). Note especially the 100% success rate for high load instances of size 1000.

Table 6.19: Top 10 VND Configurations according to $R_{\text{rel}}$.

| Conf. | Step-Function | Init. | Reb. | $\mathbf{R_{rel}}$ | $\mathbf{C_a}$ | # Valid | t[s] |
|---|---|---|---|---|---|---|---|
| $C_9$ | BestImprove | CH-S | CH-R | 0.095 = | 29.6 > | 772 | 221.4 |
| $C_1$ | BestImprove | CH-S | CH-R | 0.099 > | 31.9 > | 773 | 226.9 |
| $C_3$ | BestImprove | CH-S | CH-R | 0.100 > | 35.0 > | 773 | 225.3 |
| $C_9$ | BestImprove | CH-O | CH-R | 0.114 > | 36.9 > | 770 | 221.4 |
| $C_{10}$ | BestImprove | CH-S | CH-R | 0.117 > | 32.8 > | 776 | 232.7 |
| $C_3$ | BestImprove | CH-O | CH-R | 0.123 > | 39.3 > | 766 | 224.7 |
| $C_1$ | BestImprove | CH-O | CH-R | 0.124 > | 43.9 > | 769 | 226.0 |
| $C_9$ | FirstImprove | CH-S | CH-R | 0.133 > | 28.8 > | 769 | 205.0 |
| $C_{10}$ | BestImprove | CH-O | CH-R | 0.133 > | 50.8 > | 772 | 230.8 |
| $C_3$ | FirstImprove | CH-S | CH-R | 0.138 > | 34.0 > | 769 | 211.6 |

Table 6.20: Top 10 VND Configurations according to $C_a$.

| Conf. | Step-Function | Init. | Reb. | $\mathbf{R_{rel}}$ | $\mathbf{C_a}$ | # Valid | t[s] |
|---|---|---|---|---|---|---|---|
| $C_1$ | BestImprove | CH-S | CH-S | 0.620 > | 15.6 = | 773 | 271.9 |
| $C_5$ | BestImprove | CH-S | CH-S | 0.618 > | 22.2 > | 769 | 264.4 |
| $C_{10}$ | BestImprove | CH-S | CH-S | 0.628 > | 23.2 = | 774 | 269.5 |
| $C_9$ | BestImprove | CH-S | CH-S | 0.616 > | 23.4 > | 768 | 268.5 |
| $C_3$ | BestImprove | CH-S | CH-S | 0.620 > | 24.6 > | 768 | 271.8 |
| $C_1$ | BestImprove | CH-S | CH-O | 0.403 > | 28.3 > | 773 | 275.3 |
| $C_9$ | FirstImprove | CH-S | CH-R | 0.133 > | 28.8 > | 769 | 205.0 |
| $C_5$ | BestImprove | CH-S | CH-R | 0.408 > | 29.1 > | 772 | 216.2 |
| $C_9$ | BestImprove | CH-S | CH-R | 0.095 = | 29.6 > | 772 | 221.4 |
| $C_1$ | BestImprove | CH-S | CH-R | 0.099 > | 31.9 > | 773 | 226.9 |

First of all, this is based on very few executions compared to the other neighborhood structures (168 and 17, compared to 2000–5000). Secondly, that means that the VND was terminated due to run-time and not because it finished. When VND is finished, every neighborhood structure has failed to improve the solution at least once. We do not present an evaluation of the time required to search each neighborhood structure because the individual times were so small that no significant conclusion can be drawn from them.

Table 6.19 shows the 10 best VND configurations based on $R_{\text{rel}}$. We can see that in comparison to LS, the required run-time has doubled. There is a surprising diversity of employed neighborhood structure configurations, configurations $C_1$, $C_3$, $C_9$, and $C_{10}$ can be observed. Far more important seems to be the use of best-improvement and CH-R as rebuilding CH. The best VND configuration for solving VNMP-O uses $C_9$ with best improvement, creates its initial solution with CH-S and uses CH-R for rebuilding the solution. We will denote this configuration by VND-O.

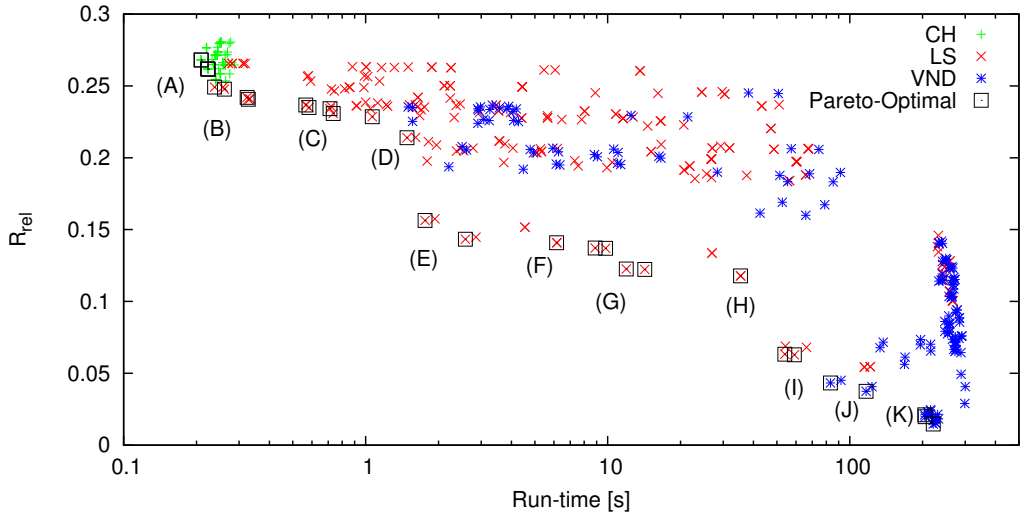



Figure 6.1: Pareto-front of the tested heuristics regarding average $R_{\text{rel}}$ and run-time over all instances.

The 10 best VND configurations with respect to $C_a$ are shown in Table 6.20. Even more neighborhood structure configurations are present, $C_5$ now also occurs. Rather surprising is that for the very best performance, CH-S has to be used for rebuilding instead of CH-R. There is even one VND configuration that uses CH-O for rebuilding before configurations using CH-R appear. It can be observed that for VND, concentrating on low average $C_a$ costs a lot of performance with respect to the relative rank. The best VND configuration achieves a relative rank of 0.62. For VND, the top configuration for low $R_{\text{rel}}$ (VND-O) also appears in the top 10 configurations for low $C_a$. This allows us to directly observe the cost (with respect to $R_{\text{rel}}$) of minimizing $C_a$. VND-O appears at the ninth position. Just by using CH-S for rebuilding instead of CH-R, the fourth position can be reached. This reduces the average $C_a$ from 29 to 23, but increases the $R_{\text{rel}}$ from 0.1 to 0.6.

The best configuration for solving VNMP-S uses $C_1$ with best-improvement and CH-S for both building the initial solution and rebuilding it. We will denote this configuration by VND-O. As with Local Search, the best configuration with respect to low $C_a$ is not the same as the best configuration with respect to the number of instances solved. The best configuration in this regard uses $C_{10}$ with best improvement, creates the initial solution with CH-O and rebuilds with CH-S. It is able to find a valid solution to 778 VNMP instances.

### 6.5.4 Comparing CH, LS and VND

Until now, we have considered the different CH, LS and VND algorithms separately. We have also mostly neglected a very important property of the presented algorithms: their required run-time. In this section, we will analyze the trade-off between the performance of the algorithms and the required run-time. We assume knowledge of Pareto-optimality (c.f. Section 2.2.3).

Figure 6.1 shows the trade-off between low $R_{\text{rel}}$ and low run-time for all tested heuristics over all instances. Note that most of the CH algorithms have been cut off, since they produce far worse results than the LS and VND configurations.

The best non-dominated construction heuristics are marked by label (A). Since we are trying to solve VNMP-O, we would expect that CH-O is present. However, this is not the case. The CH configurations at (A) are just related to CH-O in the sense that they too use DLHeavyVN as SVN strategy, MostFree as IVN strategy and emphasize the implementation of virtual arcs. The main difference to CH-O is that they use Spread-1 (faster but slightly worse) or Spread-2 as IVA strategy. CH-O itself does not occur in this graph (as Pareto-optimal CH configuration), it is dominated by the fastest LS configuration. This shows that the "best" configuration of an algorithm is not necessarily useful.

Two clusters of LS configurations are marked by (B). Since they perform on par with CHs in term of required run-time (and $R_{\text{rel}}$), it is not surprising that they use first-improvement and OnlyOverloading. They employ the RemapVnode neighborhood structure. The difference between the two clusters is the initialization CH, using CH-S is faster but CH-O gives better results. The three configurations at (C) use the ClearSarc, ClearSnode and RemapSlice neighborhood structures, still with first-improvement, OnlyOverloading and CH-O for initialization.

Marked with (D) is the first configuration using a complete neighborhood structure with OverloadingFirst. The employed neighborhood structure is still RemapSlice, which is not surprising because it has nearly the same size whether using OnlyOverloading or OverloadingFirst. Therefore, the increase in run-time is not large. At (E), a huge improvement in performance is evident. It is caused by switching from CH-O to CH-R as rebuilding CH. From here on out, CH-R is exclusively used as rebuilding strategy.

The configurations at (E) also mark the begin of a clearly visible pattern. (E) and (F) mark a group of three clusters, (G) and (H) mark a group of three clusters, and (I) and (J) (the dominated LS configurations) mark a group of three clusters. The three cluster groups represent different neighborhood structures, at (E) RemapSlice is used, at (G) RemapVnode and ClearSnode at (I). Analyzing one cluster group further reveals that the changes in performance levels are caused by going from CH-S to CH-O as initialization heuristic and then by using best-improvement instead of first-improvement. The differences at the lowest level are caused by using OverloadingFirst instead of None as neighborhood prioritization strategy. Putting this together, that means that for example at (G) and (H), the RemapVnode neighborhood structure is used. Starting from the fastest (and worst with respect to $R_{\text{rel}}$) configuration of LS, we know that it uses OverloadingFirst and CH-S for initialization. The next slower configuration switches to None. The following increase in performance is caused by using CH-O for initialization (and going back to OverloadingFirst). The last configuration at (G) again switches to None. (H) marks the same configuration, but now using best-improvement. Since this pattern repeats itself for three different neighborhood structures, we can state with some certainty the following: Using best-improvement instead of first-improvement has a significant performance penalty for a small gain in $R_{\text{rel}}$. Initial solutions created by CH-S let LS terminate earlier but with a slightly worse solution than when using CH-O. Using OverloadingFirst gives an edge with respect to required run-time while being $R_{\text{rel}}$ neutral, even when using first-improvement. Note that LS-O (like

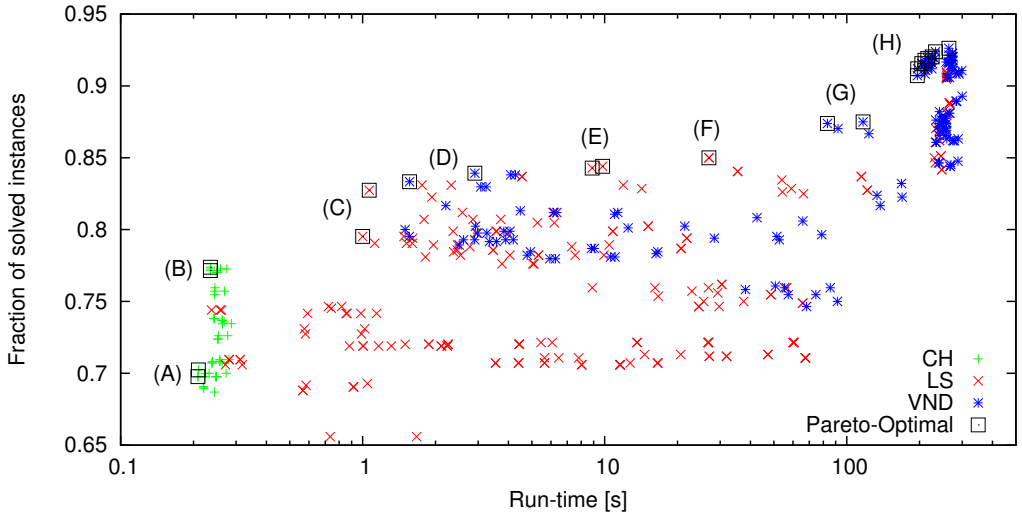



Figure 6.2: Pareto-front of the tested heuristics regarding the fraction of solved instances and run-time over all instances.

CH-O earlier) is dominated by VND configurations at (J), which achieve better results while requiring the same amount of time.

The VND configurations at (J) use $C_{12}$, CH-S for initialization and CH-R for rebuilding. The difference between the two Pareto-optimal configurations is once again the employed step function, with first-improvement being faster but producing slightly worse results. The cluster of VND configurations at (K) uses $C_9$. The best configuration is VND-O.

In Figure 6.2 we present the same trade-off analysis, but now with respect to the number of solved instances. Again, most of the tested CH configurations have been cut off to focus on the interesting configurations. At (A), we see the fastest non-dominated CH configurations, which are relatives of CH-O. The difference lies just in the selected SVA strategy. (B) marks (relatives of) CH-S.

At (C), we see the fastest LS configurations. As was the case for minimizing $R_{\text{rel}}$, they use RemapSlice, first-improvement, OnlyOverloading, and initialize with CH-S. The better configuration (in terms of solved instances) uses CH-R instead of CH-O for rebuilding. At (D), we can observe a phenomenon that did not occur previously. It marks VND configurations which are fast enough to beat LS configurations by a large margin and also achieve excellent results. Fast VND configurations could also be seen in the previous figure, but they were far worse than LS in terms of performance. The neighborhood structure configuration used at (D) is $C_8$. This is remarkable, since no analysis we have carried out previously indicated that $C_8$ is useful. Its main strength is speed, which is achieved by using three neighborhood structures in OnlyOverloading configurations. The difference in the two marked configurations at (D) is again the step-function.

After the incursion of VND, (E) and (F) mark the only complete (i.e., not using OnlyOverloading) neighborhood structures able to compete in terms of solved instances: RemapVnode. At (E) first-improvement is used, at (F) best-improvement.

At (G), VND is used in configuration $C_{12}$, which derives its advantage in execution speed from the fact that it does not utilize the RemapVnodeTAP neighborhood structure. (H) marks a mix of different configurations, all of which use RemapVnodeTAP, also containing VND-S. The slightly faster configurations use first-improvement, the slower ones best-improvement.

## 6.6 Conclusion & Future Work

In this chapter, we compared 512 CH, 216 LS, and 144 VND algorithms. We could show that for the VNMP, each algorithm class has its application area: CHs for finding solutions fast, VND for finding the best solutions and LS covering the range in-between, depending on the used neighborhood structures. The flexibility of LS was especially surprising, since it was able to outperform CH configurations on one end of the spectrum and VND configurations on the other. For CHs, the most important strategy is the target choice for virtual nodes, so this is a clear area of interest for future improvements. For LS we could see that best-improvement works slightly better than first-improvement, but at a significant run-time cost. Reducing the neighborhood size also reduced the performance, but brought the execution speed into CH territory. VND benefited from the reduced neighborhoods as well when searching for valid solutions. For LS, the initialization strategy has a pronounced influence on the result. The discussed VND variants produced the best results, but at a high run-time cost.
We could also show that for all of the compared algorithms, every parameter has an influence on the final outcome and in addition, synergy between settings was very important to produce the best results. Quite often it was the case that a combination of settings that produce the best results on average does not result in a configuration achieving the very best performance.
A major direction for promising future work is parameter tuning. For CH, we selected three out of 512 configurations and based all further results upon them. It might be the case that other configurations produce even better results. An indication of this is that the handcrafted CH-R works far better for rebuilding solutions than CH-O or CH-S. Searching the RemapVnodeTAP neighborhood structure more efficiently might also be a promising area for future research, leading to Very Large Neighborhood Search [2].

CHAPTER 7

# Memetic Algorithm

## 7.1 Introduction

This chapter presents a Memetic Algorithm (MA) for solving the VNMP. For the basics of Genetic Algorithms (GAs), and its extension the Memetic Algorithm, see Section 2.2.9. During the following discussions, our main focus will be to answer the following questions in the context of the VNMP:

- What are suitable solution representations and crossover operators, and what is their influence?
- Does the crossover operation have a beneficial influence on the final outcome, or is mutation as the only variation operator enough for good performance? In other words, does the application of a full Genetic Algorithm bring any benefit?
- Is the time for local improvement well spent, i.e., does it make sense to use a Memetic Algorithm instead of a Genetic Algorithm?

The main motivation for developing a Memetic Algorithm for solving the VNMP is that it is a population based algorithm. In the end, we get a collection of very good solutions instead of a single one. When we simply want to implement the cheapest solution, it does not matter that we have multiple high quality solutions. In practice however, there are often factors that require consideration, but are not included in the objective. By having a collection of good solutions available, we can select according to additional external criterions. By applying a local improvement technique, we hope to speed up the process of finding good solutions.
In Section 7.2, we will discuss the relevant background with respect to Genetic Algorithms with focus on different solution representations applicable to the VNMP. The proposed Memetic Algorithm is outlined in Section 7.3. For the evaluation of the algorithms see Section 7.4. We conclude in Section 7.5. The algorithm presented in this chapter has been published in [89].

**Algorithm 7.1:** Memetic Algorithm for the VNMP

**Input** : VNMP instance I
**Output**: Solution S for I

```
1  Population P;
2  InitializePopulation(P,I);
3  while !terminate() do
4  |   Solution p1=select(P);
5  |   Solution p2=select(P);
6  |   Solution offspring=crossover(p1,p2);
7  |   mutate(offspring);
8  |   copyArcs(offspring,p1,p2);
9  |   localImprovement(offspring);
10 |   insert(P,offspring);
11 end
12 return best(P);
```

## 7.2 Background and Related Work

The problem representation plays a crucial role for the performance of a Genetic Algorithm. As an example, its influence in the context of the Travelling Salesman Problem is discussed in [113]. In particular, there is a special representation designed for problems where entities have to be grouped together (grouping problems) [49], like in the case of the VNMP virtual nodes that are mapped to the same substrate node (see the following section for details). We will utilize this representation to define a GA. Successful applications of this representation include the access node location problem [3] and the Multiple Travelling Salesman Problem [47]. However, it is not clear that this representation is always advantageous when applied to grouping problems. For instance, [51] reports a successful application of a GA to the generalized assignment problem, without using this representation. Also, its robustness is questioned in [20]. Therefore, we set to goal of analyzing the performance implications of different representations for the VNMP.
For an overview on Memetic Algorithms, see Section 2.2.9.

## 7.3 A Memetic Algorithm for the VNMP

Algorithm 7.1 shows the Memetic Algorithm for the VNMP in pseudocode, which we will describe in this section. In Chapter 6 we have shown that the most important step while constructing a solution to the VNMP is the choice of the location of the virtual nodes in the substrate. Therefore, we designed the Genetic Algorithm to work primarily on finding good mapping targets for virtual nodes and use other algorithms to create a complete solution, i.e., to implement the virtual arcs. The main task of the GA is thus to assign virtual nodes to substrate nodes. But do we actually care where a single virtual node is mapped? It matters if it is the first node using a particular substrate node, because then the associated usage cost has to be paid. Otherwise,

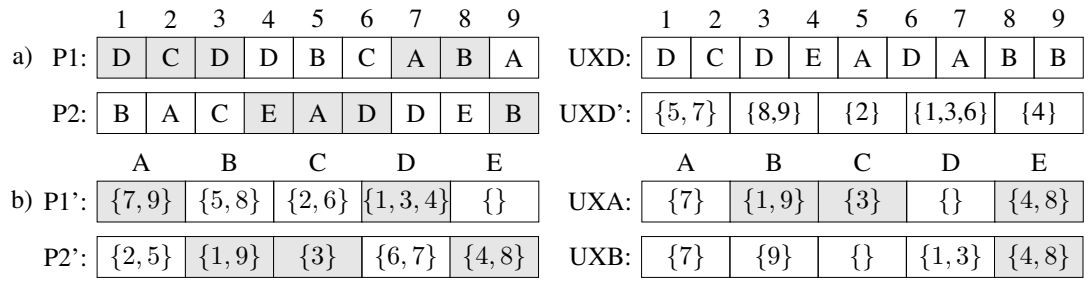



Figure 7.1: Comparison of different implementations of uniform crossover for the direct (a) and grouping (b) representations.

the concrete mapping decision does not directly influence the solution cost. The one thing that matters though is that there are enough free resources, and this is determined to a large extent by the other virtual nodes also mapped to the same substrate node. Therefore, it is important to find good groups of virtual nodes that can be mapped to the same substrate node, without requiring more than the available amount of resources. This gives raise to two different ways of solving the mapping problem: finding a substrate node for every virtual node or finding a good group of virtual nodes for every substrate node. We study the two corresponding solution representations.

The first representation is a simple vector that specifies the mapping target for each virtual node. We will call this representation the direct representation. The second representation focuses on the grouping aspect and represents a solution as a vector of sets, which specify the virtual nodes mapped to each substrate node. We will call this representation the grouping representation. Figure 7.1 shows examples of the direct and grouping representations of the same VNMP solution. P1 and P2 are solutions in direct representation, P1' and P2' are the grouping representations of the very same solutions. For instance, P1 shows that the virtual nodes 1, 3 and 4 are mapped to substrate node D. Correspondingly, P1' contains the set $\{1, 3, 4\}$ for substrate node D.

The employed representation influences how the crossover operator works. As its basic scheme, we chose uniform crossover. The simpler variants of one-point and two-point crossover were rejected based on preliminary results, which showed their inferiority.

For the direct representation, the uniform crossover works in a straight forward manner. For every virtual node the mapping target is randomly selected from one of the parents. Figure 7.1 shows a possible result of the uniform crossover of P1 and P2 in direct representation. We will call this crossover operator UXD. The marked components in the figure are the ones selected to be carried over to the offspring. UXD' shows the translation of UXD to the grouping representation for reference later on.

The uniform crossover for the grouping representation utilizes the same principle. For every substrate node, the virtual nodes mapped to it are chosen randomly from one of the parents. We will call the set of virtual nodes mapped to a substrate node a virtual node group from here on out. Due to the solution representation based on sets, two effects can occur that are not possible with the direct representation. In each solution, a virtual node is part of exactly one virtual node group. When none of those groups are selected to be present in the offspring, a virtual node

remains unmapped after the crossover operation. If both groups are selected, then the virtual node would be mapped twice, which is not allowed. The first problem can be remedied by just utilizing the mapping decision of one of the parents for all unmapped virtual nodes after the crossover procedure has finished. To solve the second problem, we override the old mapping with a newer mapping. This means that the sequence in which the groups are copied matters, since the later copy may disturb an earlier one. We will compare two different copying strategies: copying all groups of one parent, then all groups of the other (UXA), and copying the groups in order of the substrate node labels (UXB). Note that our decision to override the old mapping was arbitrary. Equally valid would be to keep the first mapping decision for a virtual node. In the end, this does not make a difference as all we will discuss in the following would still apply.

Figure 7.1 shows the result of applying these crossover operators using parents P1' and P2'. For UXA, we see for example that only node 7 is mapped to substrate node A, instead of 7 and 9, which are mapped for P1'. This is because after all groups for P1' have been copied (for substrate nodes A and D), those of P2' are transferred. In particular, the group for substrate node B, containing virtual nodes 1 and 9. Node 9 therefore cannot also be mapped to substrate node A and is removed from there; only virtual node 7 remains.

For UXB, the substrate node groups are copied according to the order of substrate nodes, first the node group from P1' for substrate node A, the node group of P2' for substrate node B, and so on. We can observe the same destruction of groups as for the UXA crossover. For example, after UXB has finished, no virtual node is mapped to substrate node C, even though the group containing virtual node 3 has been copied from P2'. This is because in the following step, the group for D was copied from P1', which also contains virtual node 3, so it is removed from C. Note that for both crossover operators, some virtual nodes remain unmapped.

The main idea of the crossover operator in general is to combine important solution properties from the parents to generate superior offspring, so we want as much information from both parents to be present in the offspring as possible. In our case, that means keeping the virtual node groups intact. The marked regions in the crossover results of UXD', UXA and UXB show the groups that have survived without node removal. We can see that for UXA three groups have survived, for UXB one group has survived, and no group survived UXD. The bad performance of UXD with respect to groupings was the reason why the grouping representation was introduced in the first place [49]. However, there is also a big difference between UXA and UXB. With UXA, at least all virtual node groups of the second parent, which are selected for crossover will survive (which are half of the groups in the expected case). With UXB, only the last group that is copied is guaranteed to survive. Therefore, we use UXA when comparing the different representation possibilities for the VNMP.

After the crossover operation we apply the mutation operator, which we will call ClearSnodes mutation, with a probability of $p_m$. The ClearSnodes mutation clears a fraction of substrate nodes by mapping virtual nodes to substrate nodes that are not selected to be cleared, if it is allowed by the mapping constraints. In Chapter 6, we introduced a neighborhood structure based on the same principle. It was in fact the inspiration for this mutation operator. This fraction of cleared nodes is chosen uniformly at random from $[0, r]$, but at least one node is cleared. In this work we used $p_m = 0.2$ and $r = 0.2$ based on preliminary results, which also showed that mutation is required for good performance.

In addition, we evaluated other, more standard approaches for mutation, like moving a virtual node to some other substrate node or swapping two virtual nodes (which are mapped to different substrate nodes). The ClearSnodes mutation was shown to be clearly superior.

Until now, we have neglected the implementation of virtual arcs. It is also a part of the solution representation, even though the crossover and mutation operators do not work on them directly. Since the arc implementation may represent a significant amount of work done by the local improvement, and the basic idea of crossover is to transfer as much information as possible from the parents to the offspring, we copy the arc implementation of the parents once the mapping for the virtual node is fixed. For every virtual arc $f$, we check the locations of $s(f)$ and $t(f)$ in the substrate graph for both parents and the offspring. If one parent utilizes the same mapping locations as the offspring, we copy its arc implementation. If both parents are compatible, the arc implementation is chosen randomly from one of the parents. If the mapping is different from both parents, the arc remains unimplemented. Unimplemented arcs will be assigned an implementation during the local improvement phase.

Since we want to check whether the time spent for local improvement actually improves the performance of the algorithm, we either use a Variable Neighborhood Descent [74] to perform local improvement, or we skip local improvement and apply a Construction Heuristic instead. The only reason for applying the Construction Heuristic is to implement all virtual arcs that have not been implemented yet to guarantee that after this step a complete solution has been generated. We selected the best Construction Heuristic for solving the VNMP-O presented in Chapter 6, CH-O, which means that virtual arc implementations are paths that cause the least increase in the substrate usage cost $C_u$ without increasing the additional resource cost $C_a$. In this chapter, we will call this method CH.

As for the Variable Neighborhood Descent, we chose $C_{12}$ as defined in Section 6.4. This configuration utilizes the neighborhood structures RemapVnode, ClearSarc, and ClearSnode (in this order). First-improvement is used to search the neighborhoods. CH-R is employed for solution reconstruction. This configuration was chosen based on the results presented in Section 6.5.4. It achieves the best possible results without using the RemapVnodeTAP neighborhood structure, which reduces the time requirements. We will denote this configuration by VND. We are also going to compare the Memetic Algorithm to VND-O, the best VND configuration for VNMP-O. The local improvement method is executed without time-limit.

The newly created and improved offspring is immediately inserted back into the population and replaces the worst solution present (steady-state GA), unless the offspring is already present in the population. Two individuals are considered to be equal, if they specify the same mapping. At this point, one GA iteration is complete, and the next one begins by utilizing a binary tournament to select the parents for the next crossover operation. Until now, we have neglected the problem of population initialization for reasons that will become obvious shortly. The main aim when initializing a population is the creation of a diverse set of good solutions. For the VNMP, there are different possibilities. One could simply randomly map virtual nodes to one of the allowed substrate nodes. Mapping virtual nodes in a way that tries to minimize the increase in $C_u$ is another. Preliminary results showed that these approaches, while creating a very diverse set of initial solutions, do not work well, because VND requires a lot of time to improve the offspring during the initial iterations. Therefore, we chose a different approach: we create one good

solution by using VND, and then apply the mutation operator with $r = 0.2$ to generate all other initial solutions. This has the additional benefit that the MA will have a good solution from the start. A population size of 10 was used.

## 7.4 Results

As in the previous chapter, we set $p^{\text{CPU}} = 1$ and $p^{\text{BW}} = 5$ to reflect the fact that it is easier to increase the CPU power of a router than to increase the bandwidth of a network connection. With these costs, even if we are unable to find a valid solution to a VNMP instance, we are able to derive a cost effective way to be able to host the current virtual network load.
To evaluate the performance of the proposed Memetic Algorithm, we used the VNMP instance set as presented in Chapter 5 in the same configuration as in Chapter 6 to allow meaningful comparisons. That means we tested all instances of the instance set with loads 0.1, 0.5, 0.8 and 1, which creates a total of 840 test instances. A run-time limit of 200 seconds was applied for instance sizes up to 100 nodes, 500 seconds were used for the larger instances. These run-time limits were chosen to reflect practical usage scenarios. Virtual networks are meant to be dynamic. Even if we assume that they only change on an hourly basis, spending 500 seconds to find a good solution means nearly 14% of the time the solution is going to be useful has already elapsed.
We compare four different MA configurations: direct representation with CH as local improvement (D-CH), with VND as local improvement (D-VND), grouping representation using UXA and CH as local improvement (G-CH) and with VND as local improvement (G-VND). To fully compare the influence of the employed crossover operator, we also test the grouping representation with UXB and CH as local improvement (G-CH-B) and VND as local improvement (G-VND-B). Furthermore, we investigate the G-VND variant with disabled crossover (G-VND-N), i.e., one individual is chosen from the population, mutated, improved and then reinserted. Preliminary experiments showed that the performance of G-VND-N cannot be improved by increasing (or decreasing) $r$. Finally, to be able to analyze the improvement caused by the GA around VND, we also present the results for VND alone and in addition compare to VND-O. In this chapter, our main focus will be solving VNMP-O, so we are mainly concerned with the average relative rank the different algorithms achieve. For reference, we also show the performance of the algorithms with respect to VNMP-S, both in terms of solved instances (i.e., the solution found to an instance has an additional resource cost of zero) and in terms of average $C_a$.
Table 7.1 shows the average performance of the tested algorithms for different instance sizes. It can be seen that D-VND and G-VND achieve the best results for all instances up to and including size 200. For sizes 500 and 1000 VND-O performs best. However, VND-O also takes more time (a maximum of 1000 seconds was allowed in Chapter 6) than the 500 seconds allowed for all GA variants for these sizes. The GA variants based on CH achieve the best results at sizes 100 and 200. With smaller instances, local improvement with VND is better than a higher number of iterations made possible by not spending time on local improvement. However, starting with size 100, performing more iterations gets more important and CH outperforms VND. Even though the configuration of VND was selected for low run-time requirements, the number of iterations for larger instances is very low. For the largest instances, the final result is basically the one

Table 7.1: Average relative rank $R_{\text{rel}}$ and its relation to the best result, average number of iterations (Its.) for GA based algorithms or average run-time for the other algorithms, fraction of solved instances (Solv.) in percent and average $C_a$ for all compared algorithms per instance size.

| | Size | D-CH | G-CH | D-VND | G-VND | G-CH-B | G-VND-B | G-VND-N | VND | VND-O |
|---|---|---|---|---|---|---|---|---|---|---|
| $R_{\text{rel}}$ | 20 | 0.352 > | 0.367 > | 0.206 = | 0.196 = | 0.366 > | 0.231 > | 0.205 = | 0.914 > | 0.761 > |
| | 30 | 0.442 > | 0.452 > | 0.232 = | 0.231 = | 0.445 > | 0.230 = | 0.247 = | 0.922 > | 0.727 > |
| | 50 | 0.475 > | 0.475 > | 0.249 = | 0.259 = | 0.471 > | 0.288 > | 0.378 > | 0.942 > | 0.746 > |
| | 100 | 0.409 = | 0.408 = | 0.388 = | 0.411 = | 0.419 = | 0.364 = | 0.546 > | 0.969 > | 0.614 > |
| | 200 | 0.393 = | 0.373 = | 0.425 = | 0.410 = | 0.389 = | 0.461 > | 0.633 > | 0.941 > | 0.379 = |
| | 500 | 0.438 > | 0.444 > | 0.609 > | 0.645 > | 0.402 > | 0.628 > | 0.757 > | 0.992 > | 0.143 = |
| | 1000 | 0.525 > | 0.547 > | 0.715 > | 0.729 > | 0.536 > | 0.715 > | 0.788 > | 0.664 > | 0.240 = |
| GA: Its. | 20 | 393268 | 357568 | 8185 | 8169 | 359589 | 8141 | 8265 | 0.2 | 0.4 |
| Other: t[s] | 30 | 259702 | 241245 | 3899 | 3854 | 238717 | 3869 | 3912 | 0.7 | 1.3 |
| | 50 | 163663 | 151068 | 1663 | 1671 | 151207 | 1657 | 1691 | 2.1 | 4.2 |
| | 100 | 63276 | 59591 | 314 | 325 | 59571 | 315 | 328 | 16.0 | 29.7 |
| | 200 | 109125 | 104063 | 333 | 352 | 103817 | 340 | 355 | 40.2 | 119.7 |
| | 500 | 43412 | 42076 | 94 | 95 | 42057 | 93 | 99 | 126.6 | 605.1 |
| | 1000 | 13631 | 13348 | 23 | 25 | 13407 | 24 | 27 | 397.1 | 828.1 |
| Solv. [%] | 20 | 97.5 | 97.5 | 100.0 | 100.0 | 97.5 | 100.0 | 100.0 | 96.7 | 97.5 |
| | 30 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 50 | 99.2 | 99.2 | 100.0 | 100.0 | 99.2 | 100.0 | 100.0 | 99.2 | 98.3 |
| | 100 | 95.0 | 95.0 | 100.0 | 100.0 | 95.0 | 99.2 | 99.2 | 95.0 | 97.5 |
| | 200 | 94.2 | 93.3 | 95.8 | 96.7 | 94.2 | 96.7 | 97.5 | 90.0 | 98.3 |
| | 500 | 77.5 | 78.3 | 76.7 | 79.2 | 78.3 | 77.5 | 76.7 | 73.3 | 90.8 |
| | 1000 | 60.0 | 59.2 | 58.3 | 57.5 | 59.2 | 61.7 | 57.5 | 57.5 | 61.7 |
| $C_a$ | 20 | 9.9 | 8.4 | 0.0 | 0.0 | 7.4 | 0.0 | 0.0 | 13.1 | 4.5 |
| | 30 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 50 | 4.9 | 4.9 | 0.0 | 0.0 | 4.9 | 0.0 | 0.0 | 4.9 | 2.1 |
| | 100 | 5.3 | 6.3 | 0.0 | 0.0 | 5.3 | 2.1 | 0.6 | 6.3 | 3.3 |
| | 200 | 5.5 | 4.1 | 3.0 | 3.4 | 5.6 | 7.6 | 4.4 | 19.0 | 1.0 |
| | 500 | 62.4 | 73.6 | 77.3 | 76.6 | 70.9 | 64.4 | 65.9 | 97.6 | 13.9 |
| | 1000 | 215.4 | 215.5 | 215.9 | 214.7 | 216.2 | 214.1 | 215.9 | 184.1 | 198.9 |

created during population initialization. Surprisingly, UXB achieves the same results as the algorithms using UXA. G-VND has a slight advantage compared to G-VND-B, but no clear pattern is visible. Disabling crossover (G-VND-N) however has a pronounced negative effect on the results for medium sized instances. Generally, no significant differences could be observed between direct and grouping representations. The influence of the type of local improvement is far more pronounced. The results for VND show that the combination with the GA has a significant positive effect on the achieved results.

Table 7.2 shows the average performance of the tested algorithms for different loads. For low loads, every tested GA achieves basically the same results, except G-VND-N, which performs far worse due to the disabled crossover operator. For medium load (0.5), a direct representation and CH as local improvement are essential. Interestingly, the grouping representation is only able to achieve the same level of performance by using UXB. It seems as if the additional disruption caused by the crossover operation is the key for good performance for this load case. Higher loads require a MA for the best performance. For load 0.8 the direct representation is

Table 7.2: Average relative rank $R_{\text{rel}}$ and its relation to the best result, average number of iterations (Its.) for GA based algorithms or average run-time for the other algorithms, fraction of solved instances (Solv.) in percent and average $C_a$ for all compared algorithms per load.

| | Load | D-CH | G-CH | D-VND | G-VND | G-CH-B | G-VND-B | G-VND-N | VND | VND-O |
|---|---|---|---|---|---|---|---|---|---|---|
| $R_{\text{rel}}$ | 0.10 | 0.302 = | 0.297 = | 0.317 = | 0.319 = | 0.289 = | 0.312 = | 0.411 > | 0.893 > | 0.527 > |
| | 0.50 | 0.355 = | 0.382 > | 0.449 > | 0.454 > | 0.358 = | 0.432 > | 0.561 > | 0.981 > | 0.479 > |
| | 0.80 | 0.492 > | 0.492 > | 0.425 = | 0.473 > | 0.500 > | 0.475 > | 0.559 > | 0.912 > | 0.495 > |
| | 1.00 | 0.584 > | 0.582 > | 0.423 = | 0.401 = | 0.582 > | 0.448 > | 0.499 > | 0.839 > | 0.562 > |
| GA: Its. | 0.10 | 430129 | 401525 | 6354 | 6321 | 401711 | 6323 | 6373 | 5.6 | 41.7 |
| Other: t[s] | 0.50 | 82299 | 75242 | 1016 | 1020 | 74971 | 1010 | 1027 | 50.2 | 218.3 |
| | 0.80 | 48183 | 43667 | 537 | 547 | 43522 | 533 | 566 | 111.2 | 316.2 |
| | 1.00 | 37147 | 33257 | 385 | 393 | 33147 | 385 | 420 | 166.0 | 331.6 |
| Solv. [%] | 0.10 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 0.50 | 99.0 | 98.6 | 97.1 | 97.6 | 98.6 | 99.0 | 97.1 | 95.7 | 99.0 |
| | 0.80 | 88.6 | 88.6 | 88.6 | 88.1 | 88.6 | 90.0 | 88.1 | 85.7 | 91.9 |
| | 1.00 | 68.6 | 68.6 | 74.8 | 76.2 | 69.0 | 73.8 | 75.2 | 68.1 | 77.1 |
| $C_a$ | 0.10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 0.50 | 0.1 | 0.1 | 0.6 | 0.4 | 0.1 | 0.2 | 0.5 | 0.7 | 0.1 |
| | 0.80 | 19.0 | 18.3 | 21.4 | 26.5 | 21.1 | 23.1 | 19.4 | 30.0 | 11.4 |
| | 1.00 | 154.3 | 160.3 | 147.3 | 141.6 | 156.1 | 141.5 | 144.0 | 155.0 | 116.3 |

significantly better, for load 1 the grouping representation has an advantage, but is not significantly better. Note that disabling the crossover operation (G-VND-N) results in bad solutions for every tested load case. Also, VND-O is outperformed by the MAs for every load case. Keep in mind that for the highest load, VND-O requires the same amount of time as all the tested GA variants, which have an average run-time of 328.5 seconds due to the set run-time limits. However, VND-O is able to solve more instances of the highest load than all other algorithms.

## 7.5 Conclusion and Future Work

In this chapter we have introduced a Memetic Algorithm for the VNMP. Based on the presented results, the MA configuration using the direct representation and VND as local improvement (D-VND) finds the best solutions. We will denote this configuration by MA-O. We could show that MA-O outperforms VND-O over a wide range of instance sizes and load cases. With reference to the main questions we set out to answer in this chapter, we could not observe any significant difference between the two tested representations if performance for specific instance sizes is relevant. For high loads, the grouping representation might offer an advantage. As for the difference between UXA and UXB, we have shown that UXB can cause performance degradations, but also seen one case where it is beneficial. We believe that analyzing the difference between those crossover variants warrants further research. Whether or not the time for local improvement is well spent depends on the instance size and load. For small sizes, local improvement increases performance, while for large instances executing more GA iterations is more important. For high loads, using local improvement is essential. We have shown that disabling crossover decreases performance in all cases. Chapter 13 offers a more in-depth comparison of MA-O with other algorithms.

We have already stated in Section 3.8 that looking into more dynamic variants of the VNMP (with changing configurations of virtual networks) is a promising area for future study. The MA presented in this chapter would be a prime candidate for solving the dynamic variant of VNMP, since it is population based. The hope here is that, when the configuration of virtual networks changes, some individuals within the population can be easily adapted to the new virtual network load. If this proves to be true, some more elaborate population management methods could be devised. For instance, multiple populations may be used: one population for finding good solutions for the current problem, another one for trying to find a solution that keeps some fraction of resources available at each substrate node or arc, and a third population that tries to minimize $C_u$, without regard to $C_a$ (within reason). The aim of the first population is clear: finding a good solution that is immediately useful. The second population's task is to find good solutions, which easily accommodate additional virtual networks in case some are added to the current instance. The third population tries to find solutions which are similarly flexible in the other direction. The solutions may not be valid at the moment, but once some virtual network is removed, they might lead to valid solutions of high quality. This is of course only a rough idea, and a lot of details such as possible interactions between populations need to be addressed. A multi-objective GA [41] might be used instead of separate populations. For an overview on the design of Genetic Algorithms for dynamic problems, see [19].

CHAPTER 8

# Greedy Randomized Adaptive Search Procedure and Variable Neighborhood Search

## 8.1 Introduction

In this chapter, we present a Greedy Randomized Adaptive Search Procedure (GRASP) and a Variable Neighborhood Search (VNS) algorithm for solving the VNMP. Instead of simple Local Search, both algorithms make use of a Variable Neighborhood Descent algorithm with ruin-and-recreate neighborhoods [154], which makes them belong to the class of Hybrid Metaheuristics [16, 143, 165]. We consider GRASP and VNS to be promising methods for solving the VNMP, since they extend the already developed algorithms from Chapter 6, which might lead to performance improvements. Indeed, we will show that the VNS approach significantly outperforms the algorithms presented in the previous chapters.
The rest of this chapter is structured as follows: Section 8.2 presents the GRASP algorithm, Section 8.3 the VNS approach. We refer to Sections 2.2.7 and 2.2.8 for background on VNS and GRASP respectively. The results of the experimental evaluation of the proposed algorithms and their comparison to other algorithms presented in this work can be found in Section 8.4. We conclude in Section 8.5. The algorithms and results presented in this chapter have been published in [90].

## 8.2 GRASP

The main idea of GRASP is to repeatedly construct solutions to a problem in a greedy, but randomized, fashion and then to improve it by a local improvement method. We refer to Section 2.2.8 for a general introduction to GRASP.

A key component for a well working GRASP approach is the randomized greedy construction heuristic. Since we will concentrate on solving VNMP-O in this chapter, we use CH-O (see Section 6.5.1) as basis for randomization. In a nutshell, CH-O works as follows: As long as virtual arcs are implementable (source and target node have been mapped), the virtual arc $f$ with the smallest fraction of $d_f$ to shortest possible delay between $m(s(f))$ to $m(t(f))$ is implemented by the path with the least increase in $C_u$ without increasing $C_a$. If no such virtual arc exists, the unmapped node with the highest total CPU requirement (CPU requirement of the virtual node and bandwidth of connected virtual arcs) is selected from the virtual network that has the smallest sum of total delay requirements. It is mapped to the substrate node with the highest amount of free CPU capacity.
We know from our experiments with Construction Heuristics (see Section 6.5.1), that the substrate node selection strategy is most influential for the overall performance of the construction heuristic. Therefore, we concentrate on randomizing this strategy and keep all other parts of the randomized construction heuristic deterministic. We introduce a parameter $\alpha \in [0, 1]$ that controls the level of randomization. When selecting a suitable substrate node for a virtual node, we collect a list of possible targets sorted by the available CPU and bandwidth, the candidate list. Let $f_{\text{Best}}^{\text{CPU}}$ denote the free CPU capacity and $f_{\text{Best}}^{\text{BW}}$ the free bandwidth capacity of the node that would have been selected by the deterministic strategy. We build the restricted candidate list by selecting all nodes $i$ with $f_i^{\text{CPU}} \geq \alpha f_{\text{Best}}^{\text{CPU}} \wedge f_i^{\text{BW}} \geq \alpha f_{\text{Best}}^{\text{BW}}$. If $f_{\text{Best}}^{\text{CPU}}$ or $f_{\text{Best}}^{\text{BW}}$ is negative (i.e., more resources are used than are actually available), $\alpha$ is replaced by $2 - \alpha$ in the relevant acceptance criterion. The mapping target is chosen uniformly at random from the restricted candidate list.
After a randomized greedy solution is generated, it is locally improved. For comparison purposes, we choose the same method that MA-O presented in the previous chapter uses. It is Variable Neighborhood Descent using configuration $C_{12}$ as defined in Section 6.4. This configuration utilizes the neighborhood structures RemapVnode, ClearSarc and ClearSnode (in this order). First-improvement is used to search the neighborhoods. CH-R is employed for solution reconstruction. This configuration was chosen because it achieves very good results for a rather small amount of required run-time. We will call this configuration simply VND and we use it without time-limit to improve solutions generated by the randomized construction heuristic up to local optimality. If the found solution after the improvement phase is better than the best solution found so far, we keep it. Then we repeat the randomized construction and improvement steps until the time-limit is reached. The best found solution is the result of GRASP.

## 8.3 VNS

VNS is an improvement over VND that focuses on diversification. Where deterministic VND would be finished, VNS adds a probabilistic shaking phase. That means that a set of neighborhood structures, which are different from the ones employed by the VND and typically larger, are used to change the currently best known solution to escape its basin of attraction. We refer to Section 2.2.7 for further details.
Our proposed VNS algorithm uses a single type of shaking neighborhood structure in multiple configurations. Let this neighborhood structure be called $N^s(v)$, with $v \in [0, 1]$ as parameter controlling the shaking vigor. $N^s$ is based on the idea of clearing substrate nodes. When $N^s(v)$

**Algorithm 8.1:** VNS for the VNMP

**Input** : VNMP instance I
**Output**: Solution S for I

```
1  Solution best=initialize(I);
2  n_ni=1;
3  while !terminate() do
4      Solution candidate=shake(N^s(v_b n_ni),best);
5      applyVND(candidate);
6      if candidate.value<best.value then // New best solution found
7          best=candidate;
8          n_ni=1;
9      end
10     else
11         ++n_ni;
12         if n_ni>n_max then n_ni=1
13     end
14 end
15 return best;
```

is applied to a VNMP solution, $N^s$ randomly selects $\lceil v \cdot |V| \rceil$ substrate nodes. All virtual arc implementations that traverse the selected nodes are removed from the solution. All virtual nodes mapped to the selected substrate nodes are mapped to a substrate node that is allowed by $M$ but not selected. If no such node exists, the mapping remains unchanged. The resulting solution is completed and improved by VND to create the final solution of one VNS iteration. During the execution of VNS we apply $N^s$ with different values for $v$. The used values are determined by two parameters, the base neighborhood size $v_\text{b}$ and the count of iterations that have not resulted in an improvement of the best found solution $n_\text{ni}$. At the beginning of a new iteration, $N^s(v_\text{b} n_\text{ni})$ is applied to the currently best found solution and the result is improved by VND. If the solution created in this manner is better than the currently best known solution, $n_\text{ni}$ is reset to one, otherwise $n_\text{ni}$ is increased by one. The upper limit for $n_\text{ni}$ is $n_\text{max}$. If this value is exceeded, $n_\text{ni}$ is reset to one. The largest shaking neighborhood searched during VNS is $N^s(v_\text{b} n_\text{max})$. Values for $v_\text{b}$ and $n_\text{max}$ have to be chosen such that $v_\text{b} n_\text{max} \leq 1$. The shaking and improvement steps are applied until the time-limit is reached. The initial solution for VNS is built by the same method as for GRASP, but without randomization, i.e., CH-O. Algorithm 8.1 shows the general outline of the proposed VNS.

## 8.4 Results

To test the proposed GRASP and VNS algorithms, we used the same selection of VNMP instances as in the previous two chapters, that means 210 full-load instances and 630 derived instances with loads 0.1, 0.5, and 0.8. In total, there are 120 instances for every size class and

Table 8.1: Average relative rank $R_{\text{rel}}$ and its relation to the best result, average number of iterations (Its.), fraction of solved instances (Solv.) in percent and average $C_a$ for different values of $\alpha$ per instance size.

| | Size | GR-0.00 | GR-0.10 | GR-0.20 | GR-0.30 | GR-0.40 | GR-0.50 | GR-0.60 | GR-0.70 | GR-0.80 | GR-0.90 | GR-0.99 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $R_{\text{rel}}$ | 20 | 0.278 = | 0.216 = | 0.219 = | 0.235 = | 0.215 = | 0.331 > | 0.384 > | 0.445 > | 0.516 > | 0.618 > | 0.842 > |
| | 30 | 0.431 > | 0.337 > | 0.318 = | 0.288 = | 0.266 = | 0.341 > | 0.337 > | 0.461 > | 0.551 > | 0.626 > | 0.826 > |
| | 50 | 0.549 > | 0.512 > | 0.463 > | 0.362 > | 0.311 = | 0.329 = | 0.402 > | 0.484 > | 0.536 > | 0.640 > | 0.868 > |
| | 100 | 0.870 > | 0.665 > | 0.468 > | 0.362 = | 0.319 = | 0.359 = | 0.410 > | 0.384 > | 0.460 > | 0.554 > | 0.692 > |
| | 200 | 0.889 > | 0.737 > | 0.488 > | 0.361 > | 0.301 = | 0.301 = | 0.313 = | 0.339 = | 0.436 > | 0.531 > | 0.740 > |
| | 500 | 0.856 > | 0.718 > | 0.511 > | 0.449 > | 0.381 > | 0.306 = | 0.325 = | 0.358 > | 0.390 > | 0.488 > | 0.617 > |
| | 1000 | 0.902 > | 0.665 > | 0.623 > | 0.529 > | 0.425 > | 0.354 = | 0.338 = | 0.341 = | 0.375 = | 0.426 > | 0.470 > |
| Its. | 20 | 1998 | 2323 | 2641 | 2916 | 3142 | 3291 | 3453 | 3677 | 3751 | 3836 | 3897 |
| | 30 | 780 | 896 | 1034 | 1153 | 1248 | 1399 | 1534 | 1588 | 1676 | 1667 | 1664 |
| | 50 | 282 | 329 | 390 | 442 | 481 | 497 | 531 | 553 | 565 | 567 | 566 |
| | 100 | 39 | 47 | 56 | 62 | 66 | 71 | 77 | 82 | 83 | 87 | 87 |
| | 200 | 45 | 54 | 66 | 78 | 90 | 98 | 103 | 112 | 116 | 127 | 129 |
| | 500 | 15 | 18 | 22 | 26 | 30 | 33 | 36 | 39 | 41 | 41 | 42 |
| | 1000 | 4 | 5 | 7 | 7 | 9 | 10 | 11 | 12 | 12 | 12 | 12 |
| Solv. | 20 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 99.2 |
| [%] | 30 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 50 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 100 | 100.0 | 100.0 | 100.0 | 99.2 | 98.3 | 97.5 | 97.5 | 99.2 | 99.2 | 97.5 | 95.8 |
| | 200 | 95.8 | 99.2 | 99.2 | 99.2 | 98.3 | 97.5 | 97.5 | 96.7 | 95.8 | 95.0 | 91.7 |
| | 500 | 71.7 | 80.8 | 81.7 | 81.7 | 76.7 | 80.8 | 78.3 | 75.8 | 77.5 | 75.0 | 70.0 |
| | 1000 | 34.2 | 58.3 | 57.5 | 55.8 | 60.0 | 55.0 | 54.2 | 55.8 | 55.0 | 55.0 | 55.8 |
| $C_a$ | 20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 8.6 |
| | 30 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 50 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 100 | 0.0 | 0.0 | 0.0 | 1.3 | 2.5 | 3.7 | 2.8 | 1.5 | 1.5 | 2.5 | 4.2 |
| | 200 | 0.4 | 0.0 | 0.0 | 0.1 | 0.4 | 1.6 | 0.9 | 1.8 | 2.0 | 4.9 | 20.4 |
| | 500 | 120.5 | 92.8 | 50.0 | 26.9 | 47.1 | 32.8 | 55.7 | 54.1 | 76.8 | 89.4 | 83.9 |
| | 1000 | 708.7 | 227.1 | 251.1 | 206.1 | 245.5 | 273.5 | 251.2 | 311.4 | 263.5 | 336.7 | 277.5 |

210 for every load. See Chapter 5 for more detail on those instances. A time-limit of 200 seconds was applied for sizes up to 100 nodes, 500 seconds for larger instances.

Our design goal for the proposed algorithms was solving VNMP-O. Therefore, the relative rank as defined in Section 3.7 will once again be our main performance metric. Section 8.4.1 compares the performance of the GRASP approach for different values of $\alpha$, Section 8.4.2 analyzes the performance of the VNS approach for different shaking neighborhood structure configurations and Section 8.4.3 shows a comparison of the best GRASP and VNS approaches, also considering previously presented algorithms.

### 8.4.1 GRASP

To evaluate the influence of $\alpha$ on the GRASP approach, we tested values for $\alpha$ from $0$ (completely random initial solution) to $0.9$ in $0.1$ increments and $0.99$ (very similar initial solutions). The average performance depending on the instance size can be seen in Table 8.1. Once again, we mark the relation of the results to the best observed result with $=$ if no significant difference

could be observed, or $>$ if the difference is indeed significant. See Section 2.4.2 for further details on the employed statistical tests.
Immediately visible in the presented data is the tendency of the best $\alpha$ value to rise with the instance size. For size 20, $\alpha \in [0, 0.4]$ yields the best results w.r.t. $R_{\text{rel}}$, while for size 1000 $\alpha \in [0.5, 0.8]$. The reason for this behaviour is that for small instances, the randomized construction heuristic does not have to make as many random choices as for the larger instance sizes. Therefore, to get the same search space coverage w.r.t. initial solutions, $\alpha$ has to be small for small instances. The results for the larger instances show that if $\alpha$ is too small, then the performance degrades, because the initial solution is far too random. Another contributing factor is that VND takes longer to optimize a very random initial solution, as can be seen by the iteration counts, which increase with rising values of $\alpha$. Therefore, fewer iterations can be performed in the same amount of time.
Note that for finding valid solutions, low $\alpha$ values seem to be beneficial, even for large instances. The GRASP approaches presented here are already an improvement when compared to MA-O as outlined in the previous chapter. GRASP with an $\alpha \in [0.1, 0.2]$ is nearly able to find a valid solution to every instance of size 200, just a single instance remains unsolved. The solution derived for the one remaining instance has so little additional resource cost $C_a$, that the reported average is zero. This is by no means certain, as the results for $\alpha = 0.3$ indicate. With this configuration, also one instance remains unsolved, but with a noticeable impact on the average $C_a$. Also for the two largest instance sizes, the average $C_a$ is lower than for the best MA configurations.
Table 8.2 shows the influence of $\alpha$ for different load cases. Again we can observe that higher values of $\alpha$ allow more iterations, but they do not lead to improved performance for high load. Instead, a value for $\alpha \in [0.4, 0.5]$ seems to be best suited when performance at a specific load level across different sizes is most important. Low $\alpha$ values are again beneficial for finding valid solutions and especially for the highest load, GRASP performs better than MA previously.
Based on these results, we select the GRASP approach with $\alpha = 0.4$ (GR-0.40) for further comparisons. We will denote this configuration by GRASP-O.

### 8.4.2 VNS

To analyze the influence of different shaking neighborhood configurations, we performed experiments with $n_{\text{max}} \in \{2, 5, 10\}$ and $v_{\text{b}} \in \{0.01, 0.05, 0.1\}$ to cover the range from very small changes with few shaking neighborhoods (i.e., few different configurations for $N^s$) to large changes with a lot of neighborhoods. Table 8.3 shows the performance of different neighborhood configurations based on instance size. The different configurations are labeled as "VNS-$n_{\text{max}}$.$v_{\text{b}}$", e.g., VNS-2.05 uses $n_{\text{max}} = 2$ and $v_{\text{b}} = 0.05$. We can see a similar behaviour to GRASP. For smaller sizes, large shaking neighborhoods are beneficial, while large instance sizes require small neighborhoods for the best levels of performance. Smaller shaking neighborhoods lead to an increased number of iterations in the same amount of time. Also note the similarity in number of iterations between VNS-5.05 and VNS-2.10, caused by the very similar maximum shaking neighborhood sizes. Indeed, between sizes 50 and 500, there is no significant difference between the two configurations. Larger shaking neighborhoods seem to increase the chance of finding valid solutions. Indeed, VNS-10.10 is able to solve all instances of size 200.

Table 8.2: Average relative rank $R_{\text{rel}}$ and its relation to the best result, average number of iterations (Its.), fraction of solved instances (Solv.) in percent and average $C_a$ for different values of $\alpha$ per load.

| | Load | GR-0.00 | GR-0.10 | GR-0.20 | GR-0.30 | GR-0.40 | GR-0.50 | GR-0.60 | GR-0.70 | GR-0.80 | GR-0.90 | GR-0.99 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $R_{\text{rel}}$ | 0.10 | 0.520 > | 0.413 > | 0.330 > | 0.288 = | 0.299 = | 0.335 = | 0.419 > | 0.495 > | 0.573 > | 0.683 > | 0.766 > |
| | 0.50 | 0.744 > | 0.586 > | 0.445 > | 0.384 > | 0.307 = | 0.313 = | 0.324 = | 0.354 = | 0.473 > | 0.574 > | 0.773 > |
| | 0.80 | 0.782 > | 0.634 > | 0.521 > | 0.430 > | 0.305 = | 0.318 = | 0.312 = | 0.350 > | 0.397 > | 0.482 > | 0.695 > |
| | 1.00 | 0.682 > | 0.568 > | 0.470 > | 0.377 = | 0.356 = | 0.360 = | 0.380 = | 0.407 > | 0.423 > | 0.482 > | 0.654 > |
| Its. | 0.10 | 1529 | 1737 | 1990 | 2193 | 2348 | 2490 | 2638 | 2781 | 2871 | 2917 | 2947 |
| | 0.50 | 108 | 142 | 173 | 208 | 239 | 267 | 298 | 322 | 330 | 333 | 336 |
| | 0.80 | 80 | 105 | 120 | 137 | 155 | 169 | 180 | 190 | 192 | 195 | 201 |
| | 1.00 | 90 | 114 | 126 | 139 | 151 | 160 | 167 | 173 | 175 | 176 | 173 |
| Solv. | 0.10 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| [%] | 0.50 | 90.0 | 97.1 | 97.6 | 97.1 | 97.1 | 96.7 | 96.7 | 96.7 | 96.2 | 95.7 | 95.7 |
| | 0.80 | 81.0 | 90.5 | 89.0 | 89.5 | 90.0 | 87.6 | 87.6 | 89.0 | 88.6 | 86.7 | 85.2 |
| | 1.00 | 72.9 | 77.1 | 78.1 | 76.7 | 74.8 | 76.2 | 74.3 | 72.9 | 73.8 | 73.3 | 69.0 |
| $C_a$ | 0.10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 0.50 | 46.5 | 7.7 | 0.4 | 0.5 | 0.5 | 0.5 | 0.5 | 0.5 | 0.6 | 0.7 | 7.1 |
| | 0.80 | 106.0 | 17.2 | 21.7 | 22.1 | 18.3 | 14.7 | 25.4 | 24.2 | 25.8 | 38.1 | 25.9 |
| | 1.00 | 321.5 | 158.0 | 149.9 | 111.3 | 150.0 | 162.8 | 151.6 | 185.9 | 170.0 | 209.0 | 192.5 |

The average $C_a$ however is higher for the largest instance sizes than for GRASP-O. The results also indicate that increasing the shaking neighborhood size in multiple small steps works better than few large steps. This can be seen with configurations that have the same maximum shaking neighborhood size. VNS-10.01 and VNS-2.05 show no significant difference in $R_{\text{rel}}$, except for sizes 200 and 1000 where using smaller steps is significantly better. The difference is more pronounced for VNS-10.05 and VNS-5.10. Until size 50 there is no difference in performance, for larger instances using smaller steps is significantly better.
The influence of the shaking neighborhood configuration across different load cases can be seen in Table 8.4. Small shaking neighborhoods lead to the best performance. Load $0.1$ is an exception, as larger shaking neighborhoods achieve the best results. As for the configurations with the same maximum shaking neighborhood size, smaller steps are a significant advantage for half of the load cases.
Based on these results, we chose VNS-10.01 for further comparison. We will denote this configuration by VNS-O.

### 8.4.3 Comparison

In this section, we compare our proposed algorithms GRASP-O (GR-0.4) and VNS-O (VNS-10.01) with previously presented algorithms. These are MA-O, the Memetic Algorithm for the VNMP introduced in Chapter 7, VND-O as introduced in Chapter 6, and the local improvement method VND on its own, to see improvement caused by GRASP and VNS when using it. Recall that the available run-time for MA-O was the same as for the GRASP and VNS algorithms and that MA-O also made use of VND for local improvement. The reported results of VND and VND-O are based on a time-limit of 1000 seconds. Note that we only show the average run-time for these two algorithms, since the others were run until the time-limit was reached.

Table 8.3: Average relative rank $R_{\text{rel}}$ and its relation to the best result, average number of iterations (Its.), fraction of solved instances (Solv.) in percent and average $C_a$ for different shaking neighborhood configurations per instance size.

| | Size | VNS-2.01 | VNS-5.01 | VNS-10.01 | VNS-2.05 | VNS-5.05 | VNS-10.05 | VNS-2.10 | VNS-5.10 | VNS-10.10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $R_{\text{rel}}$ | 20 | 0.389 > | 0.436 > | 0.396 > | 0.416 > | 0.240 > | 0.244 > | 0.340 > | 0.198 = | 0.163 = |
| | 30 | 0.402 > | 0.404 > | 0.394 > | 0.344 > | 0.305 > | 0.293 > | 0.229 = | 0.304 > | 0.283 = |
| | 50 | 0.457 > | 0.390 = | 0.356 = | 0.396 = | 0.333 = | 0.368 = | 0.338 = | 0.390 = | 0.472 > |
| | 100 | 0.432 > | 0.371 = | 0.372 = | 0.349 = | 0.486 > | 0.506 > | 0.425 > | 0.578 > | 0.630 > |
| | 200 | 0.460 > | 0.360 = | 0.316 = | 0.376 > | 0.490 > | 0.554 > | 0.479 > | 0.591 > | 0.685 > |
| | 500 | 0.467 > | 0.420 > | 0.344 = | 0.395 = | 0.500 > | 0.520 > | 0.547 > | 0.658 > | 0.624 > |
| | 1000 | 0.468 > | 0.339 = | 0.366 = | 0.462 > | 0.459 > | 0.518 > | 0.579 > | 0.596 > | 0.665 > |
| Its. | 20 | 7327 | 7327 | 7345 | 7325 | 6796 | 5812 | 6822 | 5701 | 4742 |
| | 30 | 3721 | 3699 | 3670 | 3590 | 3132 | 2577 | 3156 | 2511 | 2010 |
| | 50 | 1766 | 1758 | 1664 | 1583 | 1321 | 1042 | 1327 | 1001 | 786 |
| | 100 | 415 | 389 | 346 | 311 | 237 | 181 | 233 | 169 | 127 |
| | 200 | 504 | 450 | 399 | 349 | 270 | 208 | 260 | 192 | 147 |
| | 500 | 157 | 143 | 124 | 110 | 85 | 67 | 83 | 62 | 48 |
| | 1000 | 51 | 44 | 39 | 33 | 25 | 20 | 25 | 18 | 14 |
| Solv. | 20 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| [%] | 30 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 50 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 100 | 98.3 | 99.2 | 100.0 | 100.0 | 99.2 | 100.0 | 100.0 | 100.0 | 99.2 |
| | 200 | 93.3 | 95.8 | 97.5 | 95.8 | 95.0 | 99.2 | 99.2 | 98.3 | 100.0 |
| | 500 | 74.2 | 74.2 | 76.7 | 79.2 | 76.7 | 77.5 | 80.0 | 76.7 | 76.7 |
| | 1000 | 55.8 | 55.0 | 59.2 | 53.3 | 55.0 | 54.2 | 55.0 | 55.0 | 53.3 |
| $C_a$ | 20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 30 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 50 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 100 | 1.2 | 1.3 | 0.0 | 0.0 | 1.3 | 0.0 | 0.0 | 0.0 | 0.6 |
| | 200 | 13.6 | 2.5 | 6.1 | 3.9 | 4.1 | 1.4 | 0.5 | 0.4 | 0.0 |
| | 500 | 97.3 | 100.0 | 68.5 | 76.5 | 86.1 | 91.7 | 87.9 | 86.3 | 107.2 |
| | 1000 | 333.1 | 281.4 | 311.2 | 317.8 | 357.6 | 315.8 | 314.6 | 339.8 | 360.7 |

For the others, we show the number of performed iterations instead. One iteration is basically one execution of VND, which takes the majority of the required run-time, and some algorithm dependent actions. For GRASP, the execution of the randomized construction heuristic, for VNS the shaking and for MA the creation of a new individual. For reference, the average run-time of these three algorithms when considering different load cases is 328.5 seconds.

Table 8.5 shows the performance of the compared algorithms in relation to each other. It can be seen that the results achieved by GRASP-O are disappointing. It is significantly outperformed by the VNS and MA algorithms. However, using GRASP around VND is significantly better than using VND alone, except for size 1000, where both perform equally well. VND-O can only be beaten or matched by GRASP-O up to size 100, then VND-O achieves significantly better results. VNS-O works far better, achieving the best solutions for sizes 30 to 200. For size 20, MA-O works marginally better. Keep in mind however, that we selected a shaking configuration for the VNS that was significantly worse for the smallest instance sizes than the alternatives, so it should be possible to at least match the MA with a different configuration. For the two largest sizes, VNS-O is beaten by VND-O, partly because the VND-O had more run-time available (and

Table 8.4: Average relative rank $R_{\text{rel}}$ and its relation to the best result, average number of iterations (Its.), fraction of solved instances (Solv.) in percent and average $C_a$ for different shaking neighborhood configurations per load.

| | Load | VNS-2.01 | VNS-5.01 | VNS-10.01 | VNS-2.05 | VNS-5.05 | VNS-10.05 | VNS-2.10 | VNS-5.10 | VNS-10.10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $R_{\text{rel}}$ | 0.10 | 0.393 > | 0.289 = | 0.253 = | 0.242 = | 0.239 = | 0.250 = | 0.250 = | 0.304 > | 0.390 > |
| | 0.50 | 0.464 > | 0.408 = | 0.360 = | 0.407 > | 0.415 > | 0.458 > | 0.436 > | 0.486 > | 0.516 > |
| | 0.80 | 0.446 = | 0.451 = | 0.408 = | 0.467 > | 0.471 > | 0.479 > | 0.486 > | 0.559 > | 0.535 > |
| | 1.00 | 0.454 = | 0.407 = | 0.432 = | 0.449 = | 0.482 > | 0.528 > | 0.506 > | 0.546 > | 0.572 > |
| Its. | 0.10 | 6220 | 6176 | 6100 | 5992 | 5451 | 4640 | 5472 | 4542 | 3744 |
| | 0.50 | 924 | 905 | 877 | 853 | 712 | 550 | 708 | 526 | 410 |
| | 0.80 | 483 | 477 | 458 | 444 | 360 | 274 | 367 | 262 | 204 |
| | 1.00 | 339 | 335 | 329 | 312 | 257 | 196 | 256 | 186 | 141 |
| Solv. | 0.10 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| [%] | 0.50 | 95.2 | 96.7 | 98.1 | 96.7 | 97.1 | 97.1 | 97.6 | 97.1 | 96.2 |
| | 0.80 | 87.1 | 87.1 | 88.6 | 85.7 | 86.7 | 87.1 | 88.6 | 87.1 | 88.6 |
| | 1.00 | 72.9 | 72.9 | 75.2 | 76.7 | 73.8 | 76.2 | 76.2 | 75.7 | 74.8 |
| $C_a$ | 0.10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 0.50 | 7.2 | 7.0 | 6.9 | 0.6 | 6.9 | 0.4 | 0.4 | 7.5 | 1.3 |
| | 0.80 | 45.5 | 30.0 | 29.6 | 36.7 | 21.5 | 21.0 | 25.8 | 40.8 | 31.7 |
| | 1.00 | 201.6 | 183.2 | 183.9 | 190.2 | 228.2 | 212.3 | 204.1 | 195.4 | 234.7 |

also made use of it) as evidenced by the average run-times. Also, it is not a coincidence that there is no significant difference between the GRASP, VNS, MA, and VND approaches for size 1000. They all use VND as local improvement strategy, and as can be seen by the iteration count, not enough iterations could be performed to reap the benefits of the more involved heuristics within the available run-time. Based on the presented results, it seems that it is best to create one good solution and improve upon it (VNS-O), instead of creating a population of good solutions and profiting from their combination (MA-O) or trying to get lucky with randomly generated (and then improved) solutions (GRASP-O).

For solving instances at a specific load level, Table 8.6 shows that the VNS approach is the best choice across all load levels, achieving significantly better results than all of the other compared algorithms. There is no reason to use GRASP-O, it is matched or outmatched by VND-O within the same or lower run-time.

## 8.5 Conclusions

In this chapter, we have presented a GRASP and VNS algorithm for solving the Virtual Network Mapping Problem. We have shown that the VNS algorithm produces significantly better results than the MA and VND approaches previously introduced. Based on the presented results, we can conclude that the main idea of VNS (successively larger random moves away from local optima) works better than learning from a set of good solutions (MA) or improving good random solutions (GRASP) for the Virtual Network Mapping Problem, at least with a rather constraining run-time budget. The comparison is fair since the same local improvement strategy (VND) was used, the parameters of all algorithms have been optimized and the same time-limits were employed.

Table 8.5: Average relative rank $R_{\text{rel}}$ and its relation to the best result, average number of iterations (Its.) or run-time, fraction of solved instances (Solv.) in percent and average $C_a$ for different solution methods per instance size.

| | Size | GRASP-O | VNS-O | MA-O | VND | VND-O |
|---|---|---|---|---|---|---|
| $R_{\text{rel}}$ | 20 | 0.476 > | 0.222 = | 0.192 = | 0.912 > | 0.753 > |
| | 30 | 0.518 > | 0.222 = | 0.241 = | 0.920 > | 0.705 > |
| | 50 | 0.598 > | 0.214 = | 0.275 > | 0.930 > | 0.696 > |
| | 100 | 0.606 > | 0.187 = | 0.368 > | 0.916 > | 0.564 > |
| | 200 | 0.577 > | 0.197 = | 0.413 > | 0.859 > | 0.372 > |
| | 500 | 0.628 > | 0.489 > | 0.538 > | 0.846 > | 0.171 = |
| | 1000 | 0.623 > | 0.592 > | 0.589 > | 0.569 > | 0.228 = |
| Its. / t[s] | 20 | 3142 | 7345 | 8185 | 0.2 | 0.4 |
| | 30 | 1248 | 3670 | 3899 | 0.7 | 1.3 |
| | 50 | 481 | 1664 | 1663 | 2.1 | 4.2 |
| | 100 | 66 | 346 | 314 | 16.0 | 29.7 |
| | 200 | 90 | 399 | 333 | 40.2 | 119.7 |
| | 500 | 30 | 124 | 94 | 126.6 | 605.1 |
| | 1000 | 9 | 39 | 23 | 397.1 | 828.1 |
| Solv. [%] | 20 | 100.0 | 100.0 | 100.0 | 96.7 | 97.5 |
| | 30 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 50 | 100.0 | 100.0 | 100.0 | 99.2 | 98.3 |
| | 100 | 98.3 | 100.0 | 100.0 | 95.0 | 97.5 |
| | 200 | 98.3 | 97.5 | 95.8 | 90.0 | 98.3 |
| | 500 | 76.7 | 76.7 | 76.7 | 73.3 | 90.8 |
| | 1000 | 60.0 | 59.2 | 58.3 | 57.5 | 61.7 |
| $C_a$ | 20 | 0.0 | 0.0 | 0.0 | 13.1 | 4.5 |
| | 30 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 50 | 0.0 | 0.0 | 0.0 | 4.9 | 2.1 |
| | 100 | 2.5 | 0.0 | 0.0 | 6.3 | 3.3 |
| | 200 | 0.4 | 6.1 | 3.0 | 19.0 | 1.0 |
| | 500 | 47.1 | 68.5 | 77.3 | 97.6 | 13.9 |
| | 1000 | 245.5 | 311.2 | 215.9 | 184.1 | 198.9 |

A promising direction for future work could be to further look into the behaviour of the presented algorithms for the largest instances sizes, where their performance still leaves something to be desired. The main problem is that VND, while already selected for reduced run-time requirements, takes too much time. It might be promising to try for instance LS-O as local improvement strategy, or some other Local Search variants as presented in Section 6.5.4. Even setting a time-limit for VND might be sufficient to improve performance for the largest instance sizes.

Table 8.6: Average relative rank $R_{\text{rel}}$ and its relation to the best result, average number of iterations (Its.) or run-time, fraction of solved instances (Solv.) in percent and average $C_a$ for different solution methods load.

| | **Load** | **GRASP-O** | **VNS-O** | **MA-O** | **VND** | **VND-O** |
|---|---|---|---|---|---|---|
| $R_{\text{rel}}$ | 0.10 | 0.497 $>$ | 0.215 $=$ | 0.315 $>$ | 0.876 $>$ | 0.528 $>$ |
| | 0.50 | 0.616 $>$ | 0.294 $=$ | 0.384 $>$ | 0.913 $>$ | 0.450 $>$ |
| | 0.80 | 0.602 $>$ | 0.333 $=$ | 0.397 $>$ | 0.841 $>$ | 0.484 $>$ |
| | 1.00 | 0.586 $>$ | 0.371 $=$ | 0.400 $=$ | 0.771 $>$ | 0.532 $>$ |
| Its. / | 0.10 | 2348 | 6100 | 6354 | 5.6 | 41.7 |
| t[s] | 0.50 | 239 | 877 | 1016 | 50.2 | 218.3 |
| | 0.80 | 155 | 458 | 537 | 111.2 | 316.2 |
| | 1.00 | 151 | 329 | 385 | 166.0 | 331.6 |
| Solv. | 0.10 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| [%] | 0.50 | 97.1 | 98.1 | 97.1 | 95.7 | 99.0 |
| | 0.80 | 90.0 | 88.6 | 88.6 | 85.7 | 91.9 |
| | 1.00 | 74.8 | 75.2 | 74.8 | 68.1 | 77.1 |
| $C_a$ | 0.10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 0.50 | 0.5 | 6.9 | 0.6 | 0.7 | 0.1 |
| | 0.80 | 18.3 | 29.6 | 21.4 | 30.0 | 11.4 |
| | 1.00 | 150.0 | 183.9 | 147.3 | 155.0 | 116.3 |

CHAPTER 9

# Preprocessing of VNMP Instances

## 9.1 Introduction

In this chapter, we present preprocessing techniques to use on VNMP instances. The main aim is to extract as much information as possible from those instances and possibly reduce their size or complexity before we start solving them. As an example, one can determine if a virtual arc can never use a particular substrate arc. If we know this beforehand, we can reduce the model of the problem by removing the variable that would tell us if the virtual arc uses the substrate arc. In addition, it is also possible to remove some constraints. If we can detect that a virtual arc can never cross a particular substrate node, then we can omit the flow conservation constraint for this substrate node and virtual arc (for formulations based on network flows). The following chapters on exact approaches for solving the VNMP will make use of the preprocessing techniques discussed in this chapter.

There are also other preprocessing opportunities which we will not regard any further. One is checking the consistency of the mapping possibilities, i.e., for each allowed mapping of a virtual node, is it possible to find a valid implementing path for all virtual arcs going out of (going into) that virtual node for one of the allowed mapping targets of the target (source) of the virtual arc. Since the VNMP instances are generated in a way that ensures this property, we do not check it during preprocessing. Implementing this check would be straight forward. Another preprocessing possibility would be checking if the capacities of nodes or arcs are actually constraining (e.g., all virtual arcs that could traverse a substrate arc require more bandwidth than available) and only add a constraint if it is actually possible to violate it. These checks are performed within the exact solvers discussed in the next chapters, so we do not consider them any further. Note however that they benefit from the domain reductions that we present in this chapter.

Before we can introduce the preprocessing methods for the VNMP, we require the following definitions:

**Definition 9.1.1** (Set of Delay-Constrained Simple Paths)**.** *Given a directed graph $G(V, A)$ and delays $d_e$, $\forall e \in A$, $P^d_{s,t}$ denotes the set of all simple paths from $s \in V$ to $t \in V$ of length at most $d$.*

**Definition 9.1.2** (Possible Nodes of a Delay-Constrained Substrate Connection)**.** *The set of possible nodes of a substrate connection from $s \in V$ to $t \in V$ with delay limit $d$, $PN^d_{s,t}$, is defined as $PN^d_{s,t} = \{i \in V \mid \exists p^d_{s,t} \in P^d_{s,t} : i \in p^d_{s,t}\}$.*

**Definition 9.1.3** (Fixed Nodes of a Delay-Constrained Substrate Connection)**.** *The set of fixed nodes of a substrate connection from $s \in V$ to $t \in V$ with delay limit $d$, $FN^d_{s,t}$, is defined as $FN^d_{s,t} = \{i \in V \mid \forall p^d_{s,t} \in P^d_{s,t} : i \in p^d_{s,t}\}$.*

The definition of the set of possible arcs of a delay-constrained substrate connection $\mathrm{PA}^d_{s,t}$ and the set of fixed arcs $\mathrm{FA}^d_{s,t}$ is analogous.

**Definition 9.1.4** (Domain of a Delay-Constrained Substrate Connection)**.** *The domain of a substrate connection from $s \in V$ to $t \in V$ with delay limit $d$, $D^d_{s,t}$, is defined as the quadruple $(PN^d_{s,t}, PA^d_{s,t}, FN^d_{s,t}, FA^d_{s,t})$. We will refer to this also as substrate domain.*

**Definition 9.1.5** (Possible Nodes of a Virtual Arc)**.** *Given a VNMP instance, the set of possible nodes of a virtual arc $f \in A'$, $PN_f$, is defined as $PN_f = \bigcup_{s \in M(s(f)), t \in M(t(f))} PN^{d_f}_{s,t}$.*

**Definition 9.1.6** (Fixed Nodes of a Virtual Arc)**.** *Given a VNMP instance, the set of fixed nodes of a virtual arc $f \in A'$, $FN_f$, is defined as $FN_f = \bigcap_{s \in M(s(f)), t \in M(t(f))} FN^{d_f}_{s,t}$.*

The definition of the possible arcs $\mathrm{PA}_f$ and fixed arcs $\mathrm{FA}_f$ for a virtual arc $f \in A'$ is analogous.

**Definition 9.1.7** (Domain of a Virtual Arc)**.** *Given a VNMP instance, the domain of a virtual arc $f \in A'$, $D_f$, is defined as the quadruple $(PN_f, PA_f, FN_f, FA_f)$.*

These definitions allow us to state the VNMP preprocessing problem as follows:

**Definition 9.1.8** (The VNMP Preprocessing Problem)**.** *Given a VNMP instance, calculate the virtual arc domains $D_f, \forall f \in A'$.*

Given a solution to this problem, we can remove superfluous variables and constraints from the exact VNMP models. As for solving this problem, it is immediately apparent that the VNMP Preprocessing Problem can be decomposed into multiple instances of the following problem:

**Definition 9.1.9** (The Substrate Domain Problem (SDP))**.** *Given a source node $s$, a target node $t$ and a delay limit $d$, calculate $D^d_{s,t}$ for the substrate graph of a VNMP instance.*

The decomposition of the calculation of $D_f$ works as follows: Let $S = M(s(f))$ be the set of all allowed sources of $f$, $T = M(t(f))$ the set of all allowed targets and $d_f$ the allowed delay of $f$. Then $D_f$ is given as $(\bigcup_{s \in S, t \in T} \mathrm{PN}^{d_f}_{s,t}, \bigcap_{s \in S, t \in T} \mathrm{FN}^{d_f}_{s,t}, \bigcup_{s \in S, t \in T} \mathrm{PA}^{d_f}_{s,t}, \bigcap_{s \in S, t \in T} \mathrm{FA}^{d_f}_{s,t})$, which is the combination of all $D^{d_f}_{s \in S, t \in T}$.

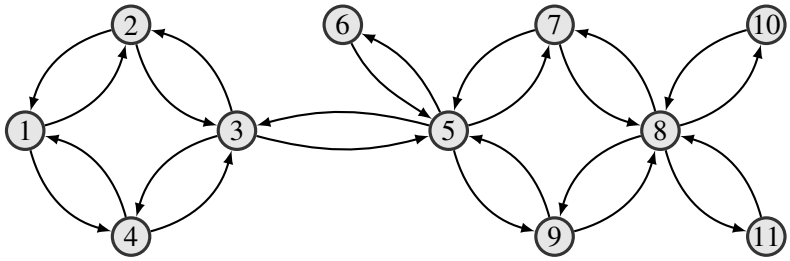



Figure 9.1: A small sample substrate network.

At first glance, solving the Preprocessing Problem in that way seems wasteful. For instance, we check if it is possible to use a substrate node for every mapping configuration, even if we have already found a mapping configuration for which it is possible. The same holds for the fixed parts. We check for every mapping configuration if we have to use a substrate arc, even if we have already found a mapping configuration where we do not have to use it. As it turns out, it is actually wasteful to skip evaluating nodes or arcs if we already know the result when solving the SDP. When nodes or arcs are skipped based on external assumptions, the solution of the SDP will be invalid for another set of assumptions. This slows down preprocessing considerably, because the solutions cannot be memoized. Section 9.6.3 shows an evaluation of this effect.

## 9.2 Solving the SDP

Central to the design of a solution method for the SDP is the question of how many SDP instances we need to solve. When considering the largest VNMP instances with 1000 substrate nodes, we know that each virtual node has about 50 different allowed locations in the substrate and that an instance contains about 1700 virtual arcs, which means that we can expect about 2500 SDP instances per virtual arc and 4.25 million instances in total. Section 9.6 will show that the real number is actually closer to 7.8 million. As there are only one million unique pairs of substrate nodes, we will need to solve the SDP for every pair multiple times with different delay values. We also know that the substrate graphs are sparse. Given these preconditions, we present the preprocessing procedure guided by an illustrative example.

Figure 9.1 shows a possible substrate network that will be the basis for solving SDPs. There are delays associated with the substrate, but they are inconsequential for now. Before we start to solve all SDPs, we perform a decomposition step that will allow us to derive partial SDP solutions to a whole range of SDP instances. The first step is calculating the biconnected components of the shadow of the substrate graph, as shown in Figure 9.2.

The red nodes are articulation points, all arcs labeled with the same number (and the nodes they connect) belong to the same biconnected component. As the second step, we build the block tree from this graph, by introducing a node for each biconnected component, which is then connected to all articulation points that connect the biconnected component to the rest of the network. Figure 9.3 shows the resulting block tree, the rectangular nodes represent biconnected

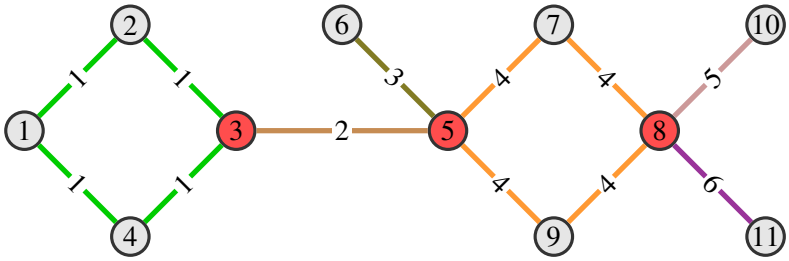



Figure 9.2: Biconnected components and articulation points of the shadow of the sample substrate network.

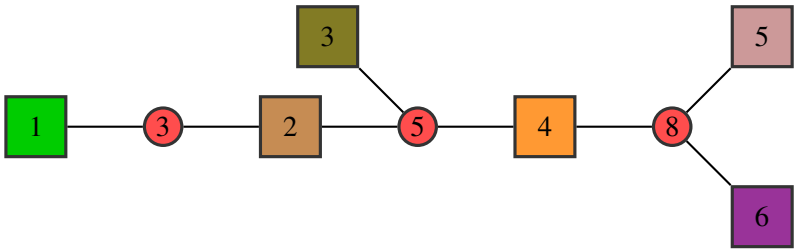



Figure 9.3: Block tree of the sample substrate network.


Figure 9.4: Simplified block tree.

components labeled by the component number, the circular ones are the articulation points which are labeled by their substrate node number.
This tree can be simplified further by removing all component nodes that are only connected to one articulation point (in this example nodes 1, 3, 5 and 6) and also all component nodes that represent bridges (i.e., only containing two articulation points and an edge between them), in this example component node 2. The final result can be seen in Figure 9.4.
During this decomposition, we keep track of the location of the original substrate nodes, so that we know for instance that substrate node 4 is represented by articulation point 3 in the simplified block tree, or that node 9 is represented by component node 4. We have now created a very compact representation of the original substrate graph that can be used to derive partial results to an SDP instance in the following way: Locate the representation of $s$ and $t$ in the simplified block tree and find a path between them. Since this path is unique, we immediately know that all traversed articulation points belong to $\mathrm{FN}^d_{s,t}$ (and $\mathrm{PN}^d_{s,t}$) and for every traversed bridge that the corresponding arc in the substrate belongs to $\mathrm{FA}^d_{s,t}$ (and $\mathrm{PA}^d_{s,t}$). We also know which components are traversed and in addition by which articulation point we have to enter a

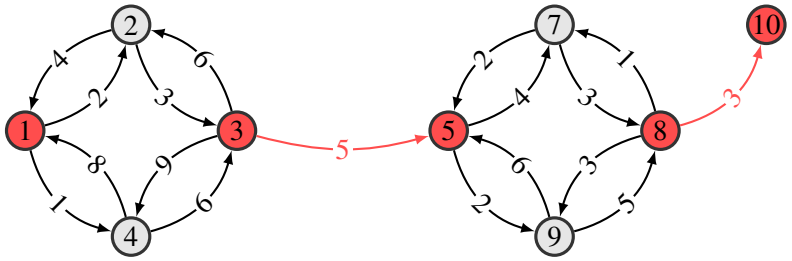



Figure 9.5: Partial domain for $D^{21}_{1,10}$.

component and by which we have to leave. If those articulation points are the same, we do not consider the component traversed. The only exception to this is when both $s$ and $t$ belong to the same component but are represented by an articulation point (i.e., the connected component node was removed during simplification). Then the representing articulation point is not in $\text{FN}^d_{s,t}$ (but possibly in $\text{PN}^d_{s,t}$). Due to the applied simplification, the found path in the block tree might still be incomplete. If $s$ is not an articulation point, but the start of the path is one, then we need to add the component that $s$ belongs to at the beginning of the path. As a refinement, if this component is a bridge, then we can immediately determine a fixed arc and do not need to consider the component. The same extension might be needed at the end of the path. The last piece of information that we can extract from the path in the simplified block tree is, that any node or arc that was not touched by the path (excluding all nodes and arcs contained in a traversed component) are definitely not in $\text{PN}^d_{s,t}$ or $\text{PA}^d_{s,t}$.

The $D^d_{s,t}$ we have calculated up to this point is a partial solution to the SDP, information concerning the traversed components is still missing. However, note that this $D^d_{s,t}$ is both independent of $d$ and valid for all SDP instances that start and end in the same component as the SDP that is currently being solved. As such, it is a prime candidate for memoization, as in storing partial domains given start and end nodes in the simplified block tree, so that they can be used while solving future SDP instances instead of calculating paths again. In some sense, this is the solution to the "easy" part of the SDP, and the "hard" part is determining the domains within the components.

Before we start outlining different methods for calculating the domains within the components, we will continue our example by considering the SDP with $s = 1$, $t = 10$, and $d = 21$ for our example substrate. Figure 9.5 shows the created partial domain $D^{21}_{1,10}$.

The process for deriving this is the following: Substrate node 1 is represented by articulation point 3 and substrate node 10 is represented by articulation point 8 in the simplified block tree. We search the corresponding path in the simplified block tree (which in this case is the complete tree). At the start of the path we need to extend with component node 1. At the end we can immediately fix the arc in the substrate from node 8 to 10, since the end component is a bridge. Also, we can add substrate nodes 1, 3, 5, 8, and 10 to $\text{FN}^{21}_{1,10}$ and the arc from 3 to 5 to $\text{FA}^{21}_{1,10}$. In Figure 9.5, $\text{FN}^{21}_{1,10}$ and $\text{FA}^{21}_{1,10}$ are marked in red, all nodes and arcs of the substrate that can not belong to $\text{PN}^{21}_{1,10}$ or $\text{PA}^{21}_{1,10}$ have been removed. Now we only need to solve the domain problem

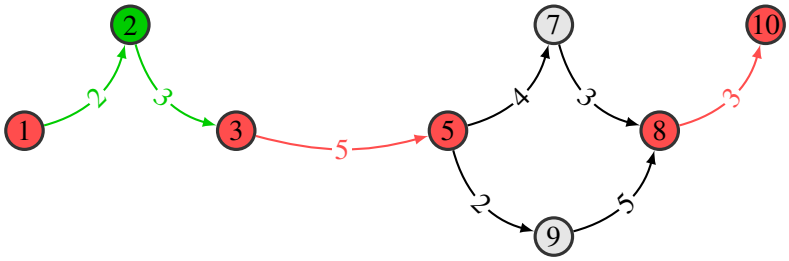



Figure 9.6: Complete domain for $D^{21}_{1,10}$.

within components 1 and 4. At this stage the specified delay limit becomes relevant. The figure contains the delay values for the arcs that might still be possible. For our path from node 1 to 10 we know the delay of all arcs currently in $\text{FA}^{21}_{1,10}$ and lower bounds for traversing components 1 and 4, determined by shortest path calculations. This allows us to derive upper bounds for the delay that we are allowed to spend while traversing the components. This bound for a particular component is calculated by taking the original delay bound $d$ and subtracting the delay of all fixed arcs and the lower delay bound of all other traversed components. In our example, we can not spend more delay than 6 in component 1 ($21 - 8 - 7$) and 8 in component 4 ($21 - 8 - 5$). Calculating $D^6_{1,3}$ and $D^8_{5,8}$ is the last step. A discussion of how to achieve this will follow, but for this example the solution can be derived easily. We can immediately discard all arcs that go into starting nodes within the component or leave the target node. Since we are only considering simple paths, these arcs cannot be used. For component 1, the path using node 4 requires a total delay of 7, which exceeds our delay bound, so this node and all connecting arcs cannot be used. Now only one path remains, therefore its nodes and arcs have to be fixed. A similar argument applies for component 4. The final result for $D^{21}_{1,10}$ is shown in Figure 9.6. The parts of the domain that were fixed by considering the component subproblem are marked in green, all arcs that cannot be used have been removed.

## 9.3 The SDP for One Component

In this section we will outline techniques for deriving domains within components. More formally, we cover methods for calculating $D^d_{s,t}$, with $s$ and $t$ belonging to the same component, and $d$ being the upper bound on the delay that we are allowed to spend within the component. To do this, we consider the extended component graph $G_c(V_c, A_c)$, which contains all nodes and arcs of the substrate that are contained within the same biconnected component $c$ in the shadow of the substrate. In addition, every substrate arc is represented by a node (called an arc node) and two arcs, one going from the source of the original substrate arc to the representing node and one going from the node to the original target of the substrate arc. The delay of the substrate arc is assigned to the arc going into the representing node, the outgoing arc has no delay. Distributing the delay any other way would not make sense, because we could for example get the situation that due to the delay bound the representing node is still allowed, but the following node is not. That would mean that using an arc is allowed, but the target of this arc is not. By

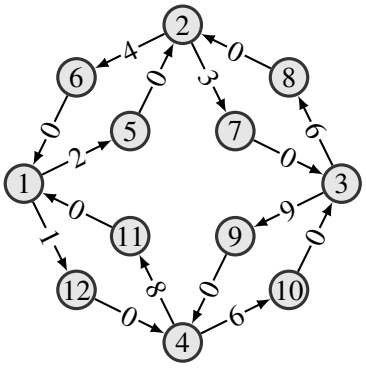



Figure 9.7: Extended component graph $G_1$.

using this extension, we only need to consider $\text{PN}^d_{s,t}$ and $\text{FN}^d_{s,t}$ for $G_c$. $\text{PA}^d_{s,t}$ and $\text{FA}^d_{s,t}$ can be reconstructed by checking which arc nodes are possible or fixed. Figure 9.7 shows $G_1$ of the example substrate. Without loss of generality, we assume that $s \neq t$ and $|V_c| \geq 6$, otherwise the problem is immediately solvable.

The following techniques are split into two categories, the first one are methods for determining $\text{PN}^d_{s,t}$. These methods will be called pruning methods, because starting from $V_c$ they remove nodes which cannot be part of $\text{PN}^d_{s,t}$. The second category are methods for calculating $\text{FN}^d_{s,t}$ from $\text{PN}^d_{s,t}$. We will call them fixing methods. The simplest pruning and fixing methods do just the bare minimum to create a valid domain for a substrate connection. All nodes of the component are added to $\text{PN}^d_{s,t}$ and only the nodes by which the component is entered and left are added to $\text{FN}^d_{s,t}$. Unsurprisingly, these methods are very fast and do not require additional memory, but of course also do not perform any additional pruning or fixing within the component. However, the final calculated domain for a substrate connection will still benefit from the information extracted from the simplified block tree and these simple pruning and fixing methods allow us to determine the additional benefit of using more elaborate methods within biconnected components.

### 9.3.1 Pruning by Simple Heuristics

Heuristic pruning was already applied during the calculation of $D^6_{1,3}$ for the example substrate. It works by removing all incoming arcs of the source node and all outgoing arcs of the target node in $G_c$. In addition, all nodes $n$ for which $\delta^-_n = 0$ or $\delta^+_n = 0$ can be removed, together with any incident arcs. This step is repeated until no further nodes can be removed. Figure 9.8 shows the result of applying heuristic pruning to $G_1$. Note that the path across substrate node $4$ cannot be excluded with this simple pruning strategy since it does not take delays into account. Therefore this pruning method is incomplete; it does not calculate $D^d_{s,t}$, but a superset of it. Its run-time is in $\mathcal{O}(n^2 + m)$, where $n = |N(G_c)|$ and $m = |A(G_c)|$.

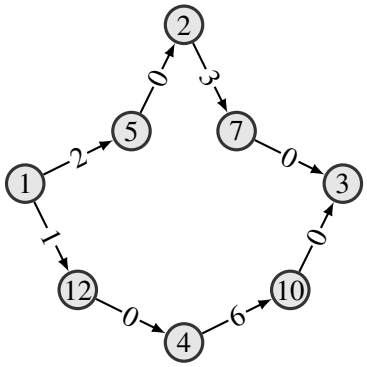



Figure 9.8: Extended component graph $G_1$ after heuristic pruning for $D^6_{1,3}$.

### 9.3.2 Pruning by All Pair Shortest Paths

In order to improve the pruning capabilities, we need to take delays into account. Therefore, we calculate the shortest possible delays between all pairs of nodes in $G_c$ with Johnson's Algorithm, see Section 2.3.3. A node $k \in V_c$ can only be in $\text{PN}^d_{s,t}$ if the shortest delay from $s$ to $k$ plus the shortest delay from $k$ to $t$ is $\leq d$. This pruning method is still incomplete, since we do not check if the two path segments $s \to k$ and $k \to t$ are disjoint, so again we get a superset of $D^d_{s,t}$. This pruning method is sufficient to generate the complete domain $D^{21}_{1,10}$ as shown in Figure 9.6. In general, we might miss pruning opportunities that are caught by the heuristic pruning, so we include the first step of heuristic pruning discussed in the previous section; the removal of incoming arcs of the source node and of outgoing arcs of the target node. Why we chose not to use the whole heuristic pruning method will become clear in Section 9.6.8. To keep this method fast, we do not repeat the delay checking after the heuristic pruning, even though it might be able to exclude further nodes. Doing so would require a costly recalculation or at least updating of the distance matrices (for instance with the algorithm presented in [43]) based on the pruned nodes. This pruning method runs in $\mathcal{O}(n)$, but requires an initial calculation of the shortest paths taking $\mathcal{O}(n^2 \log(n) + nm)$.

### 9.3.3 Pruning by Integer Linear Programming

Until now, we could only solve the problem of computing $\text{PN}^d_{s,t}$ approximately. This is not surprising, since for every node in $V_c$, we basically need to solve a 2-disjoint paths problem, which is $\mathcal{NP}$-complete [57].

**Definition 9.3.1** (The 2-Disjoint Paths Problem)**.** *Given a directed graph $G = (V, A)$ and two pairs $(s_1, t_1), (s_2, t_2)$ of pairwise different vertices of $G$. Find two vertex-disjoint directed paths $P_1$ and $P_2$ in $G$, where $P_1$ goes from $s_1$ to $t_1$ and $P_2$ from $s_2$ to $t_2$.*

If we want to test whether $k \in \text{PN}^d_{s,t}$, we need to split $k$ into two copies $k'$ and $k''$. All incoming arcs of $k$ go to $k'$, all outgoing arcs leave from $k''$. Then we set $G = G_c$, $s_1 = s$, $t_1 = k'$, $s_2 = k''$ and $t_2 = t$ to get a 2-disjoint paths problem. To reduce the 2-disjoint paths problem to

the problem of determining whether $k \in \text{PN}^d_{s,t}$, we just need to do this transformation in reverse. We include an additional node $k$ in $G$ and add an arc from $t_1$ to $k$ and from $k$ to $s_2$. By finding a simple path from $s_1$ to $t_2$ crossing $k$ within $G$, we solve the 2-disjoint paths problem. As a side-note, the special case of planar graphs is in $\mathcal{P}$ [153], but we do not restrict ourselves to planar extended component graphs. So if we want to calculate $\text{PN}^d_{s,t}$ exactly, we need to solve an $\mathcal{NP}$-complete problem for every node in $V_c$ (and we have not even considered the delay constraint yet). One possibility to do this is using Integer Linear Programming (ILP) to solve the following problem:

**Definition 9.3.2** (The Node Testing Problem)**.** *Given a directed graph $G(V, A)$ with arc delays $d_e$, $\forall e \in A$, three nodes $s$, $t$, and $k$ from $G$ and a delay limit $d$. Find a $p^d_{s,t} \in P^d_{s,t}$, such that $k \in p^d_{s,t}$.*

We now present two different ILP models based on network flows for solving the node testing problem.
The first model, denoted by (TWOFLOW), is based on the idea of finding a simple path from $s$ to $k$, and a simple path from $k$ to $t$, while forbidding that they share a node and enforcing that they do not exceed $d$. It utilizes decision variables $y^1_e \in \{0, 1\}$, $\forall e \in A$ to indicate if an arc is used for the first path (to $k$) and decision variables $y^2_e \in \{0, 1\}$, $\forall e \in A$ to indicate if an arc is used for the second path (to $t$).

$$\text{(TWOFLOW)} \quad \sum_{e \in A | s(e) = i} y^1_e - \sum_{e \in A | t(e) = i} y^1_e = \begin{cases} 1, & \text{if } i = s \\ -1, & \text{if } i = k \\ 0, & \text{otherwise} \end{cases} \qquad \forall i \in V \qquad (9.1)$$

$$\sum_{e \in A | s(e) = i} y^2_e - \sum_{e \in A | t(e) = i} y^2_e = \begin{cases} 1, & \text{if } i = k \\ -1, & \text{if } i = t \\ 0, & \text{otherwise} \end{cases} \qquad \forall i \in V \qquad (9.2)$$

$$\sum_{e \in A | s(e) = i} (y^1_e + y^2_e) \leq 1 \qquad \forall i \in V \qquad (9.3)$$

$$\sum_{e \in A | t(e) = i} (y^1_e + y^2_e) \leq 1 \qquad \forall i \in V \qquad (9.4)$$

$$\sum_{e \in A} (y^1_e + y^2_e) d_e \leq d \qquad (9.5)$$

$$y^1_e \in \{0, 1\} \qquad \forall e \in A \qquad (9.6)$$

$$y^2_e \in \{0, 1\} \qquad \forall e \in A \qquad (9.7)$$

Equalities (9.1) and (9.2) ensure flow conservation, so that the result is one connected path, inequalities (9.3) and (9.4) limit the incoming and outgoing flow at every node, so that the result is a simple path and that the two flows cannot share an arc. Inequality (9.5) is the delay constraint. The integrality constraints (9.6) and (9.7) ensure the integrality of the final solution. Note that this model allows disconnected flow circulations. As a result, for reconstructing the

solution to the model, we need to follow the flow values from $s$ over $k$ to $t$ to see which arcs (and nodes) are actually used for the path. After we have introduced the second model, we will discuss why we need to know the actual path and why we accept the overhead of having to reconstruct it instead of ensuring that no superfluous flow can be contained in the result. The presented model does not contain an objective function, since node testing is a satisfiability problem. (TWOFLOW) can be transformed to correspond to the definition of an ILP model as presented in Section 2.2.12 by adding the minimization of a constant as objective. This also holds for the following models for satisfiability problems.

The second model, denoted by (FLOWINFLOW) is based on the idea of finding a simple path from $s$ to $t$ which is limited by $d$ and contains a path from $s$ to $k$. It utilizes decision variables $y_e^1 \in \{0,1\},\ \forall e \in A$ to indicate if an arc is used for the path from $s$ to $t$ and decision variables $y_e^2 \in \{0,1\},\ \forall e \in A$ to indicate if an arc is used for the path from $s$ to $k$.

(FLOWINFLOW)

$$\sum_{e \in A | s(e)=i} y_e^1 - \sum_{e \in A | t(e)=i} y_e^1 = \begin{cases} 1, & \text{if } i = s \\ -1, & \text{if } i = t \\ 0, & \text{otherwise} \end{cases} \qquad \forall i \in V \tag{9.8}$$

$$\sum_{e \in A | s(e)=i} y_e^2 - \sum_{e \in A | t(e)=i} y_e^2 = \begin{cases} 1, & \text{if } i = s \\ -1, & \text{if } i = k \\ 0, & \text{otherwise} \end{cases} \qquad \forall i \in V \tag{9.9}$$

$$y_e^1 - y_e^2 \geq 0 \qquad \forall e \in A \tag{9.10}$$

$$\sum_{e \in A | s(e)=i} y_e^1 \leq 1 \qquad \forall i \in V \tag{9.11}$$

$$\sum_{e \in A | t(e)=i} y_e^1 \leq 1 \qquad \forall i \in V \tag{9.12}$$

$$\sum_{e \in A} y_e^1 d_e \leq d \tag{9.13}$$

$$y_e^1 \in \{0,1\} \qquad \forall e \in A \tag{9.14}$$

$$y_e^2 \in \{0,1\} \qquad \forall e \in A \tag{9.15}$$

Equalities (9.8) and (9.9) ensure flow conservation. The linking constraints (9.10) state that the flow from $s$ to $k$ has to be contained in the flow from $s$ to $t$. Inequalities (9.11) and (9.12) force the path from $s$ to $t$ to be simple, (9.13) realizes the delay constraint. The integrality of the solution is enforced with (9.14) and (9.15). As with the previous model, (FLOWINFLOW) allows disconnected flow circulations, so the actual path from $s$ to $t$ over $k$ has to be reconstructed after the model has been solved.

Why do we allow disconnected flow circulations? After all, it would be easy to add an objective to both models, stating for instance that the number of selected arcs should be minimized. However, keep in mind that we are not only interested in the solution of the model with one particular $k$, we need to check all $k \in V_c$, which would be very time-consuming if done in the naive way. Luckily, when we solve the model for one particular $k$, we can extract the path from $s$ to $t$

that has been found. This path proves not only that there exists a simple path from $s$ to $t$ using $k$ within the delay limit, it proves this for all nodes on the path, and those nodes do not have to be checked via the ILP models. So we actually want to get long paths as a result, because they reduce the need for further ILP solver invocations. By adding constraints to remove the disconnected flow circulations, we shorten the resulting path which causes more work later on. We could go towards the other extreme and add an objective to maximize the number of selected arcs and remove the circulations by using directed connection cuts, but then we get the overhead of actually having to find a longest path and proving its optimality, which again increases the run-time requirements.

This touches on another requirement for ILP models for solving the node testing problem: rapid reconfigurability. The structure of the ILP model (number of constraints, involved variables) is constant for one particular extended component graph. The only changing components are the coefficients on the right-hand side of the (in)equalities for different values of $s$, $t$, $k$ and $d$. So it is much more efficient to change the coefficients of the model to match new values of $s$, $t$, $k$, and $d$ than to rebuild the whole model. A consequence of this is that models that require cuts for correctness, like the variant previously mentioned that maximizes path length, are highly undesirable, because every cut that has to be separated represents work that has to be thrown away when the model is reconfigured. Section 10.5 will offer some results on this reconfiguration aspect.

As a result of these considerations, we rejected two other modeling possibilities that we know of for the node testing problem. One model is based on the idea of having two units of flow which leave $s$, at $k$ one unit is removed and at $t$ the second one. This model has the advantage of only needing one flow variable per arc. However, it can happen that we get solutions where one unit of flow goes from $s$ to $k$ and then loops back to $k$ again (so we have two units of flow entering $k$ and one leaving) and one unit of flow directly from $s$ to $t$. We would need cuts to ensure feasibility and as already explained, this is undesirable in our situation. Due to this reason, we also rejected another modeling idea: require a flow from $s$ to $t$ and use directed connection cuts to ensure that this flow crosses $k$.

### 9.3.4 Pruning by Path Enumeration

Until now, we have looked at the problem to calculate $\text{PN}^d_{s,t}$ from the point of view that we have to find paths that prove the membership of each node within this set. However, it also holds that $\text{PN}^d_{s,t} = \bigcup_{p^d_{s,t} \in P^d_{s,t}} N(p^d_{s,t})$. Therefore, if we know $P^d_{s,t}$, we can immediately calculate $\text{PN}^d_{s,t}$. The problem here is of course that the size of $P^d_{s,t}$ grows exponentially with the size of the input graph (in our case the extended component graph). There are three arguments that make the enumeration of all paths nevertheless a viable approach in this situation: The biconnected components are rather small (and adding one node for each arc makes paths only longer but does not increase the number of paths), they are still relatively sparse (even though they are denser than the parts of the network that could be represented by the simplified block tree) and we do not need to store all paths, we only need to keep the union of the nodes used by the paths. To put it more precisely, for each $s, t \in V_c$, determine each value of delay $d$ for which simple paths between $s$ and $t$ exist and store the union of the nodes used by those paths and the nodes used

by the next smaller possible value of $d$. The result is $\text{PN}^d_{s,t}$. Since it is not possible to enumerate only paths from $s$ to $t$, we enumerate all paths starting from $s$ to all $t' \in V_c$ within $d$ and calculate the corresponding $\text{PN}^d_{s,t'}$.

Algorithm 9.1 shows how the path enumeration procedure works. The most important data structure used by this algorithm is the `PathSearchState`. It contains information about how a particular node was reached, including: which node was reached, how much delay was incurred to reach this node and which nodes have been visited. One input of the algorithm is a collection of such states that have not been explored during the last invocation because they have exceeded the allowed delay value. These are called the continuation search states. Assume $\text{PN}^d_{s,t}$ has already been calculated. If we now have to calculate $\text{PN}^{d'}_{s,t'}$, with $d' > d$ and possibly $t \neq t'$, then we can simply continue the algorithm with the state it ended in when calculating until $d$, without having to repeat the computations. For the first invocation of the algorithm for a particular start node $s$, the continuation search state is initialized with one `PathSearchState`, the node set to $s$, delay to 0, and the path just containing $s$. The algorithm has two main stages. In the first stage all paths not exceeding $d$, starting from the continuation search states, are enumerated and stored for each target node in ascending order of delay. In the second phase, the store for $\text{PN}^d_{s,t}$ (in the algorithm referred to as `Unions`) is updated. So with one invocation of this algorithm to calculate $\text{PN}^d_{s,t}$, we also get $\text{PN}^{d'}_{s,t'}$ for all $t' \in V_c$ and $d' \leq d$. As an implementation side-note, in line 28 an empty set is returned for the first invocation of the algorithm.

Within the path enumeration algorithm, the union of (nodes of) paths is a central operation and therefore critical for the overall execution speed. In preliminary runs we have seen that using a bit-vector outperforms other alternatives, like sets or hash-sets, by a wide margin. In the bit-vector representation, a path is a vector of bits with the size of $|V_c|$. All nodes used by the path are set to 1, the others are set to 0. Note that in the case of the extended component graph, this representation of a path via its nodes is unique because $G_c$ contains nodes representing arcs in the original biconnected component. The union of paths in the bit-vector representation is a simple bitwise-or operation. Using this representation also has a very low overhead compared to the alternatives in terms of memory. However, even though we only store the path unions and not the paths themselves, the memory requirements are too high when applying this method to the largest VNMP instances. To combat this, there is a size limit for the biconnected component. If this size limit is exceeded, pruning by all pair shortest paths is performed instead of path enumeration. See Section 9.6.7 for an evaluation of this behavior.

This concludes the discussion of pruning techniques, we will now present fixing techniques to calculate $\text{FN}^d_{s,t}$ of $G_c$.

### 9.3.5 Fixing by Testing

To determine if a node $k \in V_c$ belongs to $\text{FN}^d_{s,t}$, we need to check if all paths from $s$ to $t$ not longer than $d$ have to use $k$. This is equivalent to checking if there are no paths not longer than $d$ that do not use $k$. The second statement can easily be tested by using a shortest path computation while forbidding the use of $k$. This is the basic idea of fixing by testing and immediately shows that determining $\text{FN}^d_{s,t}$ is in $\mathcal{P}$, in contrast to determining $\text{PN}^d_{s,t}$, which is in $\mathcal{NP}$. Of course, applying fixing by testing in the straight forward manner of forbidding every $k \in V_c$ in turn and

**Algorithm 9.1:** Pruning by path enumeration

**Input** : Extended Component Graph G, delay limit d, continuation search states CS
**Output**: For each node in G, complete set of nodes usable when reaching them within d

```
1  Queue<PathSearchState> states, Paths paths(num_vertices(G));
2  for PathSearchState continueState ∈ CS do
3      if continueState.delay≤d then
4          states.push(continueState);
5          CS.erase(continueState);
6      end
7  end
8  while !states.empty() do
9      PathSearchState s=states.pop_front();
10     if s.delay>d then  CS.push(s);
11     else // found a new way to reach a node
12         paths.at(s.node).add(s.delay,s.path);
13         forall the outgoing arcs oa from s.node do
14             n=target(oa);
15             nn=outgoingNeighbor(n); // n is an arcnode
16             if s.path.contains(nn) then continue; // would introduce a loop
17             PathSearchState newS=s; // store new state
18             newS.path.add(n,nn);
19             newS.node=nn;
20             newS.delay=s.delay+delay(oa);
21             states.push(newS);
22         end
23     end
24 end
25 forall the nodes n in G do // update path unions
26     npaths=paths.at(n); // new paths in ascending delay order
27     if npaths.empty() then continue;
28     union=Unions.at(n).latest(); // fetch union of previous call
29     delay=npaths.front().delay;
30     forall the paths p in npaths do
31         delayNow=p.delay;
32         if delayNow>delay then // previous delay-level complete
33             Unions.at(n).insert(delay,union);
34             delay=delayNow;
35         end
36         union.add(p);
37     end
38     Unions.at(n).insert(delay,union);
39 end
```

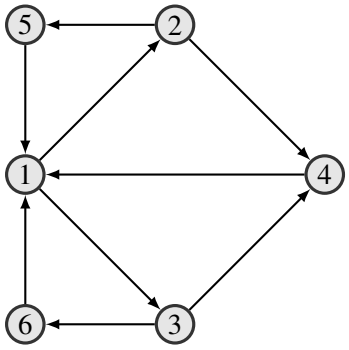



Figure 9.9: Nodes 2 and 3 are strong articulation points, but do not belong to $\text{FN}^d_{1,4}$.

checking if a path of length at most $d$ from $s$ to $t$ exists would be highly inefficient. Every node belonging to $\text{FN}^d_{s,t}$ has to be contained in every path not longer than $d$ from $s$ to $t$. So if we test a node $k$ and find an alternative path, we immediately know that only nodes within this path can belong to $\text{FN}^d_{s,t}$. If we test a node of this path, and again find an alternative path, we can iterate this argument. Only nodes belonging to the intersection of both paths can belong to $\text{FN}^d_{s,t}$. As a result, for every alternative path we find, we can exclude more than one node from $\text{FN}^d_{s,t}$ with high probability. This method runs in $\mathcal{O}(n^2 \log(n) + nm)$.

There are some possibilities to reduce the number of nodes that need to be tested even further. One of those is to calculate the strong articulation points (see Section 2.3.2) of $G_c$. Only strong articulation points can belong to $\text{FN}^d_{s,t}$, if delays are not considered. If a node different from $s$ and $t$ belongs to $\text{FN}^d_{s,t}$, then there is no path from $s$ to $t$ without this node. So removing this node would increase the number of strongly connected components, which means that this node is a strong articulation point. Note that the converse is not true, strong articulation points do not necessarily belong to $\text{FN}^d_{s,t}$, even if a path from $s$ to $t$ contains them.

An example of this can be seen in Figure 9.9. When trying to determine the fixed nodes for paths from node 1 to 4, it is necessary to use a strong articulation point (2 or 3), but they are not fixed. This example also shows that heuristics like "only cross a strong articulation point if it is really necessary, then it belongs to $\text{FN}^d_{s,t}$" are invalid. Restricting testing to strong articulation points can potentially speed up this fixing method. However, it is no longer complete since we do not consider delays. With delays, it might happen that a node is fixed, even though it is not a strong articulation point.

Another possibility for improving the performance of the fixing method is the use of dominators (see 2.3.1). When calculating $\text{FN}^d_{s,t}$, every dominator of $t$ (while using $s$ as a source node) has to belong to $\text{FN}^d_{s,t}$ and only the remaining nodes of $G_c$ have to be tested. In the results section we will analyze the fixing performance of the dominator technique on its own to be able to judge what additional solving performance it could add to the testing approach.

Note that both methods do not take the delay limit into account, so they can be strengthened by not using $G_c$ as a basis, but rather the graph that is derived from $G_c$ by deleting all nodes not in $\text{PN}^d_{s,t}$. Then, better pruning will improve the performance of the fixing by testing (but

not its (in)completeness). However, every determination of $\text{FN}^d_{s,t}$ depends on $\text{PN}^d_{s,t}$, which precludes the possibility of precomputation, so strong articulation points or dominators have to be calculated anew every time.

### 9.3.6 Fixing by Path Enumeration

Fixing by path enumeration is essentially the same as pruning. It is also basically free if the pruning is done by path enumeration. Algorithm 9.1 still applies, but in addition to the union we also keep track of the intersection of the paths to calculate $\text{FN}^d_{s,t}$. In the step corresponding to line 28, the intersection returned during the first execution of the algorithm is $V_c$.

### 9.3.7 Fixing by Integer Linear Programming

For completeness sake, it is also possible to test if a node has to be fixed by using an ILP formulation, which we will denote by (FIXFLOW). It is based on the idea of finding a simple path form $s$ to $t$ which is limited by $d$ and does not contain $k$. It utilizes decision variables $y_e \in \{0,1\},\ \forall e \in A$ to indicate if an arc is used for the path from $s$ to $t$.

$$\text{(FIXFLOW)} \qquad \sum_{e \in A | s(e) = i} y_e - \sum_{e \in A | t(e) = i} y_e = \begin{cases} 1, & \text{if } i = s \\ -1, & \text{if } i = t \\ 0, & \text{otherwise} \end{cases} \qquad \forall i \in V \tag{9.16}$$

$$\sum_{e \in A | s(e) = i} y_e \leq \begin{cases} 0, & \text{if } i = k \\ 1, & \text{otherwise} \end{cases} \qquad \forall i \in V \tag{9.17}$$

$$\sum_{e \in A} y_e d_e \leq d \tag{9.18}$$

$$y_e \in \{0,1\} \qquad \forall e \in A \tag{9.19}$$

Equalities (9.16) ensure flow conservation. Inequalities (9.17) forbid the use of $k$ for the path from $s$ to $t$ and force the path to be simple, (9.18) forces the path to respect the delay constraint. The integrality of the solution is enforced with (9.19). As with previous formulations, the actual path has to be reconstructed after the model has been solved.

## 9.4 Solving the SDP for One Component Efficiently

After outlining all of the pruning and fixing methods, one part of the algorithm for solving the SDP for one component is still missing: the procedure to execute these methods. As already discussed, we can expect to have to calculate $\text{PN}^d_{s,t}$ and $\text{FN}^d_{s,t}$ for extended component graphs with the same start and end nodes for different delay values. This means that is is possible (and effective) to use previous results as bounds for the current domain request. Algorithm 9.2 shows how this works in detail.

**Algorithm 9.2:** Calculating $\text{PN}^d_{s,t}$ and $\text{FN}^d_{s,t}$ for one component

**Input** : Extended component graph $G_c$, source node $s$, target node $t$, delay limit $d$ and domain store Domains
**Output**: $\text{PN}^d_{s,t}$ and $\text{FN}^d_{s,t}$ of $G$

```
1  PosNodes pn,FixNodes fn;
   // calculate pn (=PN^d_{s,t})
2  if Domains.hasPosNodes(s,t,d) then pn=Domains.getPosNodes(s,t,d) else
3      PosNodes pnUB=Domains.getPosUB(s,t,d);
4      PosNodes pnLB=Domains.getPosLB(s,t,d);
5      if pnUB==pnLB then pn=pnUB else
6          PosNodes pnToTest=pnUB∧(¬ pnLB); // bit-vector operations
7          prune(s,t,d,pnToTest); // call one of the pruning methods
8          pn=pnLB∨pnToTest;
9      end
10 end
   // calculate fn (=FN^d_{s,t})
11 if Domains.hasFixNodes(s,t,d) then fn=Domains.getFixNodes(s,t,d) else
12     FixNodes fnUB=Domains.getFixUB(s,t,d);
13     FixNodes fnLB=Domains.getFixLB(s,t,d);
14     if fnUB==fnLB then fn=fnUB else
15         RestrictGraph Gr=restrict(G_c,pn);
16         FixNodes fnToTest=pn∧fnLB∧(¬ fnUB);
17         fix(s,t,d,fnToTest,Gr); // call one of the fixing methods
18         fn=fnUB∨fnToTest;
19     end
20 end
   // store calculated domains, return
21 Domains.setPosDomain(s,t,d,pn);
22 Domains.setFixDomain(s,t,d,fn);
23 Domains.simplify(s,t,d);
24 return pn,fn;
```

The input of the preprocessing algorithm for a component $c$ is the extended component graph $G_c$, the data for the required connection (source and target nodes, maximum allowed delay) and the store for previously calculated $\text{PN}^d_{s,t}$ and $\text{FN}^d_{s,t}$ within this component.

The first step of the algorithm is calculating $\text{PN}^d_{s,t}$. If $\text{PN}^d_{s,t}$ is already present in the domain store, then there is nothing more to do. Otherwise, we use the results of previous calls as upper and lower bounds. The upper bound is a previously calculated $\text{PN}^{d'}_{s,t}$ with $d' > d$ but as small as possible. If no such domain has been calculated before, then it is $V_c$. Obviously, if there is no simple path using a node within $d'$, then there will be none within $d$, so $\text{PN}^d_{s,t}$ is a subset of $\text{PN}^{d'}_{s,t}$. Similarly, we try to find a lower bound $\text{PN}^{d''}_{s,t}$ with $d'' < d$ but as large as possible. If no such domain has been calculated, then it is the empty set. If there is a simple path using a node within $d''$, then this path is of course still valid with delay bound $d$, so all nodes in $\text{PN}^{d''}_{s,t}$ also belong to $\text{PN}^d_{s,t}$. If the upper and lower bounds are equal, then we already know $\text{PN}^d_{s,t}$ and nothing more needs to be done. Otherwise, the nodes that are included in $\text{PN}^{d'}_{s,t}$, but not in $\text{PN}^{d''}_{s,t}$, are the ones we still need to test whether they belong to $\text{PN}^d_{s,t}$, which is stated in line 6. Note that we assume a bit-vector representation of the domains, so we can apply bit-wise logical operators. After determining which nodes need to be tested, we apply one of the pruning methods discussed previously. The pruning method will remove all nodes from the nodes to test that do not belong to $\text{PN}^d_{s,t}$. It is free to ignore the input about nodes to test. The Path Enumeration method for instance cannot use it, the All Pair Shortest Path method on the other hand can and does use it. The final $\text{PN}^d_{s,t}$ is then assembled from the lower bound and the nodes that have passed the pruning.

The second step of the algorithm determines $\text{FN}^d_{s,t}$. It works basically the same way as the first step. One thing to note here is that the term upper bound still refers to a domain that was derived by allowing more delay and lower bound refers to a domain where less delay was allowed. The result is that for fixed nodes, the upper bound will contain fewer nodes than the lower bound. In case no suitable upper or lower bounds have been calculated, the upper bound will be the empty set and the lower bound $V_c$. For calculating the nodes for which we have to test whether they belong to $\text{FN}^d_{s,t}$ (line 16), that means we have to test all nodes that belong to $\text{PN}^d_{s,t}$ and are present in lower bound, but not in the upper bound. The restriction to $\text{PN}^d_{s,t}$ is especially useful at the beginning when no suitable lower bound is known. One of the discussed fixing methods is then called with the nodes to test and removes all nodes that do not belong to $\text{FN}^d_{s,t}$. As an additional input, the fixing method gets the extended component graph, restricted by the possible nodes. This for instance increases the efficiency of path calculations, because parts of the extended component graph are removed.

The last step of the algorithm is storing the calculated domains in the domain store, followed by a simplification step. The main idea here is that it is not necessary to store all calculated domains, because a lot of them will be the same. It is sufficient to store the same domain only twice; for the lowest and the highest delay for which this domain occurred. As a consequence, the memory requirements for storing domains can be reduced. On the other hand, the probability of finding exactly the requested domain already present in the domain store is decreased, but we will still get the same upper and lower bounds and save execution time that way.

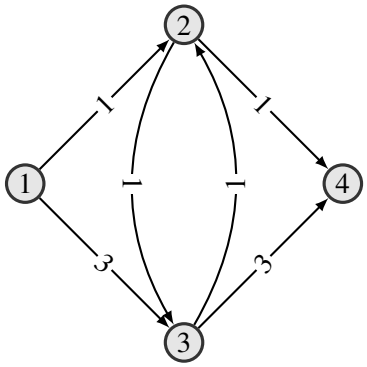



Figure 9.10: $\text{FN}^d_{s,t}$ can change without $\text{PN}^d_{s,t}$ changing.

As an implementation remark, note that $\text{PN}^d_{s,t}$ and $\text{FN}^d_{s,t}$ are stored completely independently from each other, because changing delays can add or remove nodes from one without affecting the other. It is easy to see that changing delay may change $\text{PN}^d_{s,t}$ without changing $\text{FN}^d_{s,t}$. Just think about a graph without fixed nodes where increasing the delay bound might add nodes to $\text{PN}^d_{s,t}$. The other direction is also possible, Figure 9.10 shows an example. For simplicity reasons we omitted the nodes for every arc. It holds that $\text{PN}^5_{1,4} = \text{PN}^6_{1,4} = \{1, 2, 3, 4\}$, but $\text{FN}^5_{1,4} = \{1, 2, 4\}$ and $\text{FN}^6_{1,4} = \{1, 4\}$. The main insight here is that even with perfect pruning, the resulting graph can contain delay-infeasible paths. This means that an incomplete fixing method that does not take delays into account applied to a pruned extended component graph derived by a complete pruning algorithm will still be incomplete. In this example, the fixing method cannot distinguish the situations for delay $5$ and $6$.

## 9.5 The Complete Preprocessing Algorithm

We have now finished the discussion of all components necessary for solving the VNMP preprocessing problem and are therefore able to show the algorithm for solving it. Algorithm 9.3 gives an high level overview of the procedure. It neglects the mentioned memoization steps for simplicity. In addition, there is a relevant implementation aspect to this algorithm. As presented, it solves the substrate domain problems one at a time for each virtual arc separately. We will refer to this as the default order. As a result, consecutive SDP instances will be very different in terms of starting node, end node, and delay constraint. It is possible to calculate the domains for the virtual arcs in another order. Instead of iterating through virtual arcs, we iterate through substrate nodes, and for each node solve the SDP for every virtual arc that may start at this node. We will refer to this order as substrate order. With this simple step, consecutive SDP instances will be more similar, which causes a performance increase up to 15%. See the results in Section 9.6.2 for more details.

**Algorithm 9.3:** Preprocessing for the VNMP

**Input** : A VNMP Instance
**Output**: Virtual arc domains $D_f, \forall f \in A'$

```
1  Calculate simplified block tree of substrate;
2  forall the virtual arcs f in VNMP instance do
3      VirtualArcDomain D;
4      forall the allowed locations ms of s(f) do
5          forall the allowed locations mt of t(f) do
               // solving the substrate domain problem
6              SubstrateDomain D';
7              Calculate path in block tree from ms to mt;
8              Add domain parts known from block tree path to D';
9              Calculate delay bounds for all crossed components;
10             forall the crossed components do
11                 Execute algorithm 9.2;
12                 Add result to D';
13             end
14             D.add(D');
15         end
16     end
17     Store D for f;
18 end
```

## 9.6 Results

In this section, we will analyze the different aspects of the preprocessing algorithm. Now follows a description of the default configuration of the preprocessing algorithm which is used unless specified otherwise. We also refer to following sections that offer further explanations and experimental evidence as to why a particular setting was chosen as default setting. The evaluation was done on instances with full load and a memory limit of 5 GB.
In the default setting, the preprocessing algorithm uses the block tree decomposition with all simplifications (see Section 9.6.1). The virtual arc domains are built in substrate order (see Section 9.6.2). Knowledge already gathered while calculating the domain of a virtual arc is not used when evaluating further mapping possibilities for the source and target node of the virtual arc (see Section 9.6.3). The following pruning methods are utilized:

**None** This method does not perform any pruning. All nodes and arcs within a component are possible.

**Heuristic** The heuristic pruning scheme described in Section 9.3.1.

**APSP** The pruning scheme based on shortest delays between nodes as described in 9.3.2.

**PathEnumeration** The path enumeration scheme presented in 9.3.4. A node limit of 40 nodes is employed, meaning that components containing more than 40 nodes will be solved by APSP. See Section 9.6.7 for details.

**TwoFlow** The ILP model TWOFLOW presented in 9.3.3 is used for pruning. If CPLEX has stored a solution from the last solving procedure, it is discarded. Section 9.6.6 contains details. The CPLEX solver object is not discarded when the model is modified (9.6.4).

**FlowInFlow** The ILP model FLOWINFLOW presented in 9.3.3 is used for pruning. Again, a possibly stored solution is discarded (see Section 9.6.6) and the solver object is kept (see Section 9.6.4).

Now follows a description of the employed fixing methods:

**None** Does not perform any fixing, only the nodes by which a component is entered and left are fixed.

**Dominators** Utilizes the idea of using dominators for fixing presented in 9.3.5.

**Testing** The fixing by testing approach introduced in 9.3.5.

**TestingSAP-0** A variant of testing that only tests strong articulation points. These strong articulation points are determined statically based on the complete extended component graph.

**TestingSAP-1** This method recomputes the strong articulation points depending on the nodes of the extended component graph that were determined to be possible ($\text{PN}^d_{s,t}$).

**PathEnumeration** The path enumeration approach presented in 9.3.4, extended to calculate fixed nodes as well. A node limit of 40 nodes is employed. If this limit is exceeded, Testing is used. See Section 9.6.7 for details.

**FixFlow** The ILP model FIXFLOW as presented in 9.3.7 is used for fixing. A possibly stored solution is discarded (see Section 9.6.6). The solver object is kept during modifications and inequality 9.17 is only applied for the testing node $k$. See Section 9.6.5 for details.

Unless stated otherwise, all combinations of pruning and fixing methods are evaluated. In the following sections, we will present the motivation for the choice of parameters and behaviors for the default configuration. Then we will analyze the characteristics of the preprocessing algorithm in more detail. The following definitions will apply during the discussion of the results. Symbol $t$ will be used to denote the run-time of the preprocessing algorithm in CPU-seconds.

**Definition 9.6.1** (Relative Node-Pruning Performance $\text{PN}_{\text{rel}}$)**.** *The relative node-pruning performance $PN_{rel}$ is defined as $PN_{rel} = \frac{\sum_{f \in A'} |PN_f|}{|A'||V|}$. Note that smaller values mean better performance because more nodes could be excluded from $PN_f$.*

Table 9.1: Influence of different block tree decomposition configurations on the average required CPU-time (t) in seconds and the pruning and fixing performance. TN is the average number of nodes of the tree decomposition.

| Configuration | t[s] | $PN_{rel}[\%]$ | $PA_{rel}[\%]$ | $FN_{rel}[\%]$ | $FA_{rel}[\%]$ | TN |
|---|---|---|---|---|---|---|
| No Decomposition | 204.2 | 67.9 | 62.6 | 4.0 | 0.9 | 1.0 |
| With Decomposition | 16.1 | 37.2 | 26.9 | 5.2 | 1.2 | 51.3 |
| With Bridge Opt. | 16.0 | 37.2 | 26.9 | 5.2 | 1.2 | 42.5 |
| With Node Opt. | 16.0 | 37.2 | 26.9 | 5.2 | 1.2 | 27.4 |
| With Both Opt. | 16.0 | 37.2 | 26.9 | 5.2 | 1.2 | 18.6 |

**Definition 9.6.2** (Relative Arc-Pruning Performance $PA_{rel}$)**.** *The relative arc-pruning performance $PA_{rel}$ is defined as $PA_{rel} = \frac{\sum_{f \in A'} |PA_f|}{|A'||A|}$.*

The definitions of the relative node-fixing performance $FN_{rel}$ and relative arc-fixing performance $FA_{rel}$ are analogous, but for fixing, larger values are better. As an example, $PN_{rel} = 0.2$ means that on average, every virtual arc can only use 20% of the substrate's nodes.

### 9.6.1 Influence of Block Tree Decomposition

Calculating the simplified block tree decomposition is the first step when executing the preprocessing algorithm. In this section, we test the influence this step has on the overall preprocessing algorithm, especially with respect to run-time, pruning and fixing performance. Additionally, we test the influence of the suggested simplifications. Those simplifications were the removal of component and articulation point nodes of degree one (henceforth called node optimization) and the removal of component nodes containing only two nodes (bridge optimization). We tested on instances of size 50. Testing with larger instances was not possible, because the required run-time exploded when omitting the block tree decomposition, especially for the ILP methods. Table 9.1 shows the behavior of the preprocessing algorithm depending on the chosen configuration. It can be seen that there is a huge difference between the performance with the decomposition or without it. By using the tree decomposition, the algorithm requires only 8% of the original run-time. Additionally, the pruning and fixing performance is better. For instance, without the decomposition, the preprocessing tells us that every virtual arc may use about 68% of the substrate nodes on average, with the decomposition this is reduced to 37%. The difference is even larger for the possible arcs. For fixing, the additional benefit of the decomposition is not as pronounced, which is due to the fact that not a lot of nodes or arcs can be fixed to begin with. This difference occurs because of the use of incomplete pruning or fixing methods. Keep in mind that for all configurations all combinations of pruning and fixing methods have been tested, for instance pruning with method None and fixing with method None. With those methods and without decomposition, nothing can be pruned or fixed. By performing the decomposition, we can exclude nodes and arcs that are not touched by the path within the block tree. For a method selection that uses complete methods, like PathEnumeration, the decomposition does not give

Table 9.2: Influence of the evaluation order of virtual arc domains on the average required run-time for preprocessing.

(a) without ILP methods

| Size | Evaluation Order | |
|---|---|---|
| | **Default** | **Substrate** |
| 20 | 0.01 = | 0.01 > |
| 50 | 0.12 = | 0.12 > |
| 200 | 2.99 > | 2.80 = |

(b) with ILP methods

| Size | Evaluation Order | |
|---|---|---|
| | **Default** | **Substrate** |
| 20 | 0.19 = | 0.18 = |
| 50 | 43.77 > | 37.38 = |
| 200 | 1261.19 > | 1253.36 = |

any additional advantages other than better run-time. The last column of Table 9.1 shows the average number of nodes of the block tree. Without decomposition, there is of course just one node, containing the complete substrate graph. More interesting is the situation with respect to the number of nodes with decomposition. Just using decomposition, the block tree contains 51 nodes on average, which is even more than the substrate size. By using bridge optimization, nine nodes can be removed, while node optimization can remove 24 nodes. By using both optimizations, the simplified block tree uses only 18 nodes. That means the preprocessing algorithm is able to find a structural representation of the input substrate graph that contains only 35% of the substrate's nodes. Unfortunately, this additional compactness does not lead to any performance gains.
Based on these results, we chose to use the block tree decomposition with both simplifications.

### 9.6.2 Influence of the Domain Evaluation Order

In Section 9.5, we have outlined two different ordering possibilities for calculating the domains of virtual arcs, the default order and the substrate order. To analyze the difference between these orderings, we have tested the preprocessing algorithm on instances of size 20, 50 and 200. The results are summarized in Table 9.2. Note that we have distinguished between preprocessing configurations that do not use ILP based pruning/fixing methods (9.2a) and configurations that use ILP based methods either for pruning, fixing, or both (9.2b), since the required run-times are very different.
We can see that for ILP based methods, using substrate order always requires less run-time than the default order and this difference is also statistically significant for all but the smallest instance size. The largest improvement is achieved for size 50, where using substrate order requires 15% less run-time. This advantage is reduced to 0.6% for size 200. Every reported value is the average of 540 runs.
As for the preprocessing configurations that do not use ILP methods, we can see that the overhead of having to determine for each substrate node the virtual arcs that may originate there is prohibitive for small instances. The default order performs better than the substrate order. Note that due to rounding the reported average run-times are the same, but still significantly different. For the largest instance size, using substrate order results in 6% lower run-times, so the overhead of using substrate order starts paying off. Every reported value is the average of 720 runs.

Table 9.3: Influence of utilizing partially known domains on the required preprocessing runtime.

| | | Fixing | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Pruning** | **Cfg.** | **None** | **Dominators** | **Testing** | **TestingSAP-0** | **TestingSAP-1** | **PathEnum.** | **FixFlow** |
| None | A | 2.2 = | 2.2 = | 2.5 = | 2.3 = | 4.1 = | 2.9 = | 31.2 = |
| | B | 7.6 > | 7.7 > | 28.2 > | 15.4 > | 160.7 > | 15.6 > | 2658.3 > |
| | C | 8.2 > | 8.3 > | 27.7 > | 16.0 > | 161.2 > | 15.9 > | 2417.4 > |
| Heuristic | A | 2.2 = | 2.2 = | 2.6 = | 2.5 = | 4.2 = | 3.2 = | 31.4 = |
| | B | 3.1 > | 3.1 > | 17.3 > | 10.3 > | 91.6 > | 10.6 > | 1493.3 > |
| | C | 8.4 > | 8.4 > | 28.5 > | 16.7 > | 166.4 > | 16.6 > | 2451.3 > |
| APSP | A | 2.3 = | 2.3 = | 2.4 = | 2.3 = | 3.6 = | 3.0 = | 31.0 = |
| | B | 7.7 > | 7.7 > | 18.2 > | 11.5 > | 93.3 > | 11.6 > | 1387.6 > |
| | C | 10.4 > | 10.5 > | 29.8 > | 18.0 > | 175.7 > | 17.2 > | 2496.9 > |
| PathEnum. | A | 2.9 = | 2.9 = | 3.0 = | 2.9 = | 4.1 = | 3.0 = | 32.5 = |
| | B | 8.3 > | 8.3 > | 17.1 > | 11.4 > | 80.7 > | 11.5 > | 1289.7 > |
| | C | 11.4 > | 11.4 > | 30.2 > | 18.8 > | 177.6 > | 17.3 > | 2447.9 > |
| TwoFlow | A | 1465.5 > | 1460.1 > | 1460.1 > | 1460.7 > | 1456.3 > | 1473.0 > | 1489.1 = |
| | B | 822.4 = | 824.0 = | 835.1 = | 825.8 = | 889.3 = | 830.4 = | 2041.1 > |
| | C | 1543.1 > | 1547.5 > | 1564.8 > | 1557.5 > | 1707.8 > | 1579.8 > | 3954.0 > |
| FlowInFlow | A | 1739.9 > | 1740.5 > | 1736.0 > | 1739.0 > | 1742.1 > | 1758.6 > | 1763.6 = |
| | B | 958.8 = | 960.7 = | 974.4 = | 963.4 = | 1029.0 = | 974.4 = | 2166.0 > |
| | C | 1775.7 > | 1775.6 > | 1797.5 > | 1787.3 > | 1942.5 > | 1831.2 > | 4200.6 > |

Based on these results, we decided to use substrate order as default setting for the preprocessing algorithm.

### 9.6.3 Influence of Partially Known Domains

When calculating the domain of a virtual arc, we combine the results of all relevant substrate domains. At the end of Section 9.1, we have argued that this might seem wasteful, because we test for instance whether a node belongs to $\mathrm{PN}^d_{s,t}$, even though we might already know from another substrate domain that it belongs to the domain of the virtual arc. In this section we show that the benefit of calculating valid substrate domains, which might be useful for the domains of other virtual arcs, outweighs the cost of possibly redundant testing in most cases.

We tested three different configurations. Configuration A is the standard configuration, which calculates every substrate domain, regardless of what is already known about the virtual arc domain. Configuration B skips the evaluation of a substrate domain (for one component), if it cannot give any new information. In algorithm 9.2, we calculate nodes that have to be tested for membership in $\mathrm{PN}^d_{s,t}$. If we already know for all nodes that they will be present in $\mathrm{PN}_f$, we skip the pruning step. The same holds for the fixing step: If we know for all nodes that would have to be tested, that they cannot belong to $\mathrm{FN}_f$, then we skip the fixing step. If pruning or fixing have been skipped, the calculated substrate domain is invalid and cannot be stored for later use.

Configuration C goes even further and just removes nodes for which it is already known if they either belong to $\mathrm{PN}_f$ or cannot belong to $\mathrm{FN}_f$ from the nodes that have to be tested during the pruning and fixing step respectively.

We have tested all three configurations on instances of size 20, 50, and 200. The results were consistent across all size classes, therefore Table 9.3 only shows the results for size 200.

Table 9.4: Influence of the method used for changing TwoFlow models on the required preprocessing time and the time required for changing the model (Setup Time).

| | t[s] | | Setup Time[s] | |
|---|---|---|---|---|
| **Size** | **Default** | **Tracking** | **Default** | **Tracking** |
| 20 | 1.1 > | 0.2 = | 0.8 | 0.0 |
| 50 | 79.0 > | 41.4 = | 35.0 | 0.4 |
| 200 | 2026.4 > | 1468.0 = | 518.9 | 5.6 |

For all pruning methods that are not ILP based, configuration A is the best choice by a very wide margin. For non-ILP-based fixing, using another configuration results in a 5 to 40 fold increase in run-time, when FixFlow is used we get an 70 fold increase. The situation is reversed for ILP based pruning methods. Here a 40% reduction in run-time can be achieved by using configuration B instead of A. Configuration C however still performs worst. Interestingly, the benefit of using configuration A for FixFlow outweighs the cost of using it for ILP based pruning. This demonstrates the tradeoff between possibly computing too much, but creating good domain bounds that can be utilized during future calculations (configuration A) and only calculating what is needed, but possibly repeatedly for different virtual arcs (configuration B). With good bounds, we can skip calling a pruning or fixing method altogether, which is beneficial if the bound creation did not take a lot of time. For slow pruning methods, the cost for creating good bounds is prohibitive, which is why configuration B performs better. The fraction of nodes that can be fixed is rather low (see results in Section 9.6.8), so good bounds are also essential when using FixFlow. Configuration C combines the worst of both worlds, the calculated domains cannot be reused and during domain calculation we still need to execute pruning and fixing methods (albeit with a reduced number of nodes).

### 9.6.4 Modification of TwoFlow

In this section, we will look at the performance differences caused by either destroying the solver object before the model is modified and then rebuilding it with the new model, or letting the CPLEX solver object track changes to the TwoFlow model. It is not clear beforehand which method is better, because both have overheads. For the first method, we have the cost of instantiating CPLEX solver objects, for the second we have the cost of the solver object having to change its internal model representation (e.g., the simplex tableau) to correspond to the modified model. As a side note, we do not offer an analysis of the alternative of destroying both solver and model and rebuilding from scratch, because we have seen in preliminary runs that preprocessing an instance with this method took more than two hours, while the other choices finished within seconds. Because the results for FlowInFlow are similar, we will only discuss TwoFlow.
We tested the preprocessing algorithm with TwoFlow as pruning method and None for fixing on instance sizes 20, 50, and 200. For TwoFlow, we applied the default changing method (change the model without an attached solver object) and the tracking method (change the model with an attached solver object). Table 9.4 shows the results.
It can be seen that the tracking method is vastly superior to the default changing method, which

Table 9.5: Influence of different configurations of FixFlow on the required preprocessing time.

| | Configuration | | | |
|---|---|---|---|---|
| **Size** | **A** | **B** | **C** | **D** |
| 20 | 0.3 > | 0.3 > | 0.0 = | 0.1 > |
| 50 | 6.1 > | 6.2 > | 2.6 > | 2.2 = |
| 200 | 76.1 > | 74.8 > | 40.3 > | 31.0 = |

requires 38% more time for preprocessing for the largest instance size. Table 9.4 also shows the total setup time required for the two configurations. The setup time is the time needed for changing the model from solving $\text{PN}^d_{s,t}$ to $\text{PN}^{d'}_{s',t'}$. There is a factor of 90 difference between the setup times for the default and tracking method, which means that it is 90 times faster to modify a model that is tracked by a CPLEX solver object than to repeatedly create new solver objects. Based on these results we chose the tracking method for TwoFlow and FlowInFlow.

### 9.6.5 Modification of FixFlow

In this section we test for FixFlow what we tested for TwoFlow in Section 9.6.4: The tradeoff between either tracking model changes with the CPLEX solver object or rebuilding the solver object once the model has been modified. However, for FixFlow we had a look at an additional possibility for modification. Inequalities 9.17 of FIXFLOW limit the flow through every node in the component and forbid flow through the node for which we want to determine whether it belongs to $\text{FN}^d_{s,t}$ (the testnode). Our design goal for the FIXFLOW model was that it can be reconfigured by just changing coefficients. Since every node may be the testnode, we need inequalities 9.17 for every node. As we need them for every node anyway, we can use them to limit the flow. There is, however, another possibility. We could only forbid flow through the testnode, which means we only have to add one inequality instead of $|V|$, which makes the model smaller. On the other hand, extracting the actual path from a solution to FIXFLOW gets more complicated, as the flow from $s$ to $t$ will no longer be a simple path. To extract a simple path, we need to perform a shortest path computation in the graph induced by the flow values. The second downside of only using one inequality is that removing constraints from a model is usually an expensive operation, which is why we look at this tradeoff in more detail in this section.

We tested four different FixFlow configurations. Configuration A uses the default changing setup (rebuilding the CPLEX solver object) and the original inequalities 9.17. In configuration B, only one inequality is used. Configuration C uses the tracking method (the CPLEX solver object gets notified about changes to the model) and the original inequalities. Configuration D uses the tracking method and only one inequality. These four configurations were used for preprocessing (with pruning method None) on instances of size 20, 50, and 200. Table 9.5 shows the results.

It can be seen that when the CPLEX solver object does not have to track modifications to the model, it does not matter whether coefficients are modified (A) or constraints are removed (B). The preprocessing will be slow either way. The situation becomes more interesting when we

use the tracking method, since we can see a significant difference here between coefficient modification (C) and constraint removal (D). For the smallest instance size, coefficient modification is faster, while the other instances constraint removal offers a distinct speed advantage. For instances of size 200, constraint removal is 25% faster. This means that it is more important to save on the number of constraints in the model for larger instances than it is to have simple model modification methods.
Based on these results, we used configuration D for FixFlow.

### 9.6.6 Removal of ILP solutions

During the discussion of ILP based pruning and fixing methods, we stated that rapid reconfigurability of an ILP formulation is very important since we need to solve very closely related problems in succession. By reconfigurability we mean the ability to solve a related problem by just changing some coefficients of the model. The alternative of course would be to rebuild the model from scratch. While conceptually simpler, this has a huge performance penalty. So, in CPLEX terms, we always use the same model and solver object, cf. Section 9.6.4. That means when we modify a model and try to solve it, CPLEX might have stored the solution to a previous variant of the model. Usually, this is done to speed up the re-solving process of the model. For instance when adding additional cuts, a previous solution might be modified to satisfy the new constraints so that a valid solution to the new model is available from the beginning. In our case however, even though the models are closely related, a solution to one model is not related to the solution of the modified model at all. Just think about changing the source node of a flow. The previous flow from another source (i.e., the previous solution) is not helpful for finding a new solution. Therefore, it might be the case that CPLEX spends too much time trying to adapt a solution instead of solving the new model. Removing a present solution before resolving might speed up the process. In this section, we will test this hypothesis.
We tested all three ILP based pruning/fixing methods without clearing and with clearing solutions. As the fixing method for the ILP based pruning methods (TwoFlow and FlowInFlow) we used None, which we also used as pruning method for FixFlow. From the instance set, we used the sizes 20, 50, and 200. Table 9.6 shows the result.
It can be seen that for all three methods, clearing the solution is significantly better than keeping it. The margin by which it is better however becomes smaller with larger instance size. This is not surprising, as the time spent on adapting a solution is small in relation to the time required for solving the model for larger instances. For the largest instances, clearing the solution results in a run-time reduction of 3% for TwoFlow, 1.5% for FlowInFlow, and 12% for FixFlow.
Based on these results, we chose to clear the solution for all three ILP based methods.

### 9.6.7 Cutoff Size for Path Enumeration

The time and memory requirements of the PathEnumeration method are exponential in the size of the component for which it is executed. Therefore, we only execute PathEnumeration if the number of nodes contained in a component is below a certain threshold. For larger components, we fall back to APSP if we perform pruning, or to Testing if we perform fixing. In this section, we analyze the influence of this threshold (referred to as node limit) on the performance of

Table 9.6: Influence of clearing a (possibly) present solution before resolving an ILP model on the required preprocessing run-time.

| | | Clear Solution | |
|---|---|---|---|
| **Method** | **Size** | **No** | **Yes** |
| TwoFlow | 20 | 0.2 > | 0.2 = |
| | 50 | 42.8 > | 41.2 = |
| | 200 | 1489.5 > | 1442.3 = |
| FlowInFlow | 20 | 0.3 > | 0.2 = |
| | 50 | 55.0 > | 53.1 = |
| | 200 | 1766.8 > | 1738.7 = |
| FixFlow | 20 | 0.1 > | 0.1 = |
| | 50 | 2.6 > | 2.1 = |
| | 200 | 35.1 > | 30.9 = |

PathEnumeration. As test set, we used all instances from the VNMP instance set and applied node limits from 10 to 100 nodes in increments of 10 nodes. We used PathEnumeration as pruning and fixing method. Table 9.7 shows the results.

It can be seen that for sizes up to 100, the run-time differences caused by different node limits are negligible and every configuration is able to preprocess all instances up to size 200. However, beginning with instance size 30, some calls to PathEnumeration have to fall back to APSP or Testing and with larger instance sizes ever larger node limits are required to avoid falling back. The tradeoff between node limit and fraction of fallbacks can be seen especially well for instance size 200 and node limits 50 and 60. We can see that for node limit 50, we fall back in 7% of all cases and require on average $3.4$ seconds of run-time. By increasing the node limit to 60, we do not fall back any longer, at the cost of an average run-time of $14.8$ seconds. That means we have more than a five fold increase of run-time when we try to solve 7% of calls to PathEnumeration exactly. This tradeoff gets even worse when we consider the increase of relative arc pruning performance $\text{PA}_{\text{rel}}$ we get by spending so much more time. By never falling back to APSP, we are able to exclude an additional $0.2\%$ of substrate arcs from $\text{PA}_f$ on average for every $f \in A'$.

By going beyond instance size 200, we can see the memory limit coming into effect as executions of the preprocessing algorithm start to fail. The algorithm can perform preprocessing for only six out of 30 instances when a node limit of 100 nodes is used, for instances of size 1000. Note that the reported run-times, fallbacks and arc pruning performances are averages of successful executions only, which means that the reported values cannot be directly compared if some instances could not be preprocessed. This is nicely demonstrated by the (apparent) increase in fallbacks when increasing the node limit from 90 to 100 for instance size 1000.

The choice of node limit generally has very little influence on $\text{PA}_{\text{rel}}$. The largest difference is caused by changing the node limit from 10 to 20 nodes, which results in about 1% more pruning for small to medium sized instances. Also note that we only show the results for $\text{PA}_{\text{rel}}$ because for $\text{PN}_{\text{rel}}$, $\text{FN}_{\text{rel}}$ and $\text{FA}_{\text{rel}}$, the differences are even smaller.

Based on these results, we chose to use a node limit of 40 nodes, since this is the largest value for the node limit so that all instances can be preprocessed. We could have reduced the node limit

Table 9.7: Influence of different node limits for PathEnumeration on the average required CPU-time (t) in seconds, the number of executions not terminated due to excessive memory requirements (# Successes), the fraction of APSP and Testing fallbacks and the relative arc pruning performance ($PA_{rel}$).

| | | Node Limit | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Size** | **10** | **20** | **30** | **40** | **50** | **60** | **70** | **80** | **90** | **100** |
| **t[s]** | 20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 30 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 50 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| | 100 | 0.7 | 0.7 | 0.9 | 0.9 | 0.9 | 1.2 | 1.2 | 1.2 | 1.2 | 1.2 |
| | 200 | 2.4 | 2.5 | 2.7 | 3.1 | 3.4 | 14.8 | 14.3 | 16.1 | 16.1 | 16.0 |
| | 500 | 13.9 | 14.1 | 16.1 | 26.4 | 58.3 | 117.3 | 133.5 | 150.0 | 162.0 | 216.6 |
| | 1000 | 68.9 | 69.8 | 73.4 | 75.6 | 151.6 | 324.1 | 329.9 | 326.2 | 318.7 | 330.4 |
| **# Successes** | 20 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| | 50 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| | 100 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| | 200 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 | 30 |
| | 500 | 30 | 30 | 30 | 30 | 29 | 23 | 19 | 17 | 15 | 11 |
| | 1000 | 30 | 30 | 30 | 30 | 29 | 18 | 18 | 18 | 17 | 6 |
| **Fallbacks[%]** | 20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 30 | 12.3 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 50 | 35.5 | 9.6 | 3.3 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 100 | 53.6 | 19.6 | 3.1 | 3.1 | 3.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 200 | 81.7 | 35.8 | 16.4 | 8.1 | 7.2 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 500 | 91.8 | 85.2 | 72.3 | 65.8 | 55.1 | 43.3 | 37.2 | 32.1 | 24.5 | 0.0 |
| | 1000 | 95.5 | 90.6 | 83.3 | 80.6 | 75.0 | 68.9 | 68.9 | 68.9 | 69.0 | 70.5 |
| **$PA_{rel}$[%]** | 20 | 26.9 | 26.9 | 26.9 | 26.9 | 26.9 | 26.9 | 26.9 | 26.9 | 26.9 | 26.9 |
| | 30 | 29.3 | 28.4 | 28.4 | 28.4 | 28.4 | 28.4 | 28.4 | 28.4 | 28.4 | 28.4 |
| | 50 | 25.2 | 24.6 | 24.3 | 23.9 | 23.9 | 23.9 | 23.9 | 23.9 | 23.9 | 23.9 |
| | 100 | 20.9 | 19.8 | 19.3 | 19.3 | 19.3 | 19.1 | 19.1 | 19.1 | 19.1 | 19.1 |
| | 200 | 19.4 | 18.2 | 17.8 | 17.5 | 17.5 | 17.3 | 17.3 | 17.3 | 17.3 | 17.3 |
| | 500 | 17.9 | 17.8 | 17.7 | 17.6 | 17.4 | 17.4 | 16.9 | 16.4 | 15.7 | 13.5 |
| | 1000 | 15.6 | 15.6 | 15.6 | 15.6 | 15.5 | 15.7 | 15.7 | 15.7 | 15.7 | 16.1 |

further, but then we would have moved even farther away from the ideal of a complete pruning (fixing) method. Note that this choice of node limit means that employing PathEnumeration is only complete for instances up to size 50. For sizes 100 and 200, it is close to complete, since less than 10% of PathEnumeration calls are solved by APSP or Testing. The pruning efficiency cost is at most $0.2\%$. For the largest instances, the behavior of PathEnumeration is actually closer to APSP or Testing than to its own. Another possibility would have been to set a node limit of 60 up to size 200 and use a limit of 40 for sizes 500 and 1000, which would have kept PathEnumeration exact up to size 200 while still being able to preprocess all instances. We decided against that to keep the configuration simple.

### 9.6.8 Comparison of Pruning and Fixing Methods

Now that we have determined the best configuration for the preprocessing algorithm and its pruning and fixing methods, we can evaluate the performance of preprocessing depending on the selected pruning and fixing methods, the general behavior of the algorithm and the usefulness of memoization.

We tested all combinations of pruning and fixing methods on the complete instance set with full load, with one exception. The ILP-based pruning methods (TwoFlow and FlowInFlow) were not tested on instance sizes 500 and 1000 due to their run-time requirements.

Table 9.8 shows the required run-time of preprocessing with different pruning and fixing methods. It can be seen that up to size 50, all configurations that do not use ILP based methods require basically the same run-time. The differences only start showing with larger instance sizes. For sizes 100 and 200, using TestingSAP-1 as fixing method requires consistently more run-time than the rest. For the larger instances, it is possible to see the influence of interactions between pruning and fixing methods on the required run-time. For instance, at size 500 and using Testing as fixing method, preprocessing using None as pruning is slower than preprocessing with APSP as pruning method. This is surprising, as None does nothing and should be faster. However, by not performing any pruning, Testing has to test more nodes, so in total it is more beneficial to use APSP as pruning method. For sizes 500 and 1000, there starts being a distinct run-time difference between PathEnumeration and the faster methods. Note that PathEnumeration is not exact any longer for these sizes, as discussed in Section 9.6.7. TestingSAP-1, however, is still slower. Also observe that TestingSAP-0 is faster than Testing, because it restricts testing only to the strong articulation points in the extended component graph. The required run-time when using None as pruning and fixing method gives the time necessary to calculate the virtual arc domains by just using the simplified block tree. Generally, the additional run-time cost of using pruning or fixing methods that actually do something is rather small.

Until now we have neglected to discuss the ILP based pruning and fixing methods. From Table 9.8, we see that they are not competitive in terms of required run-time, even though we spent some effort to implement them efficiently. FixFlow can be applied up to size 1000 (taking about a factor of 10 longer than other methods), but for TwoFlow and FlowInFlow, the run-time is too high for the largest sizes. Interestingly, size 200 is also the largest size for which PathEnumeration can be applied in an exact configuration. Note that we do not use such a configuration, but in Section 9.6.7 we have shown that it would require 16 seconds on average for instances of size 200, which compares quite favorably to the 1450 seconds TwoFlow requires. To calculate exact domains for larger instances, either more memory (for PathEnumeration) or more run-time (for TwoFlow) than we were able to supply has to be invested.

Of course, analyzing the required run-time alone is insufficient, since different methods for pruning and fixing have different pruning and fixing performances. Table 9.9 shows the relative node pruning performance $PN_{rel}$ for the presented pruning methods. As a general tendency, the larger the instances, the better the node pruning performance. For size 20, half of the substrate's nodes can be excluded from the domain of every virtual arc, which increases to nearly 85% for size 1000. Unsurprisingly, the exact methods achieve the best results and the difference to the other pruning methods is significant. However, by using exact methods we can only exclude about 2% of additional substrate nodes from $PN_f$, while the additional run-time cost, as shown

Table 9.8: Mean required run-time in seconds of preprocessing with different pruning and fixing methods.

| | | Fixing | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Size** | **Pruning** | **None** | **Dominators** | **Testing** | **TestingSAP-0** | **TestingSAP-1** | **PathEnum.** | **FixFlow** |
| 20 | None | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 |
| | Heuristic | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 |
| | APSP | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 |
| | PathEnum. | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 |
| | TwoFlow | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 |
| | FlowInFlow | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 | 0.2 | 0.3 |
| 30 | None | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.6 |
| | Heuristic | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.6 |
| | APSP | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.0 | 0.6 |
| | PathEnum. | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.6 |
| | TwoFlow | 5.3 | 5.3 | 5.3 | 5.3 | 5.3 | 5.2 | 5.8 |
| | FlowInFlow | 6.5 | 6.5 | 6.6 | 6.5 | 6.5 | 6.5 | 7.1 |
| 50 | None | 0.1 | 0.1 | 0.1 | 0.1 | 0.2 | 0.1 | 2.1 |
| | Heuristic | 0.1 | 0.1 | 0.1 | 0.1 | 0.2 | 0.1 | 2.2 |
| | APSP | 0.1 | 0.1 | 0.1 | 0.1 | 0.2 | 0.1 | 2.1 |
| | PathEnum. | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 2.0 |
| | TwoFlow | 41.8 | 42.2 | 42.2 | 41.5 | 42.5 | 41.5 | 43.8 |
| | FlowInFlow | 52.7 | 53.7 | 52.9 | 53.3 | 54.0 | 53.0 | 54.6 |
| 100 | None | 0.6 | 0.7 | 0.7 | 0.7 | 1.2 | 0.8 | 8.7 |
| | Heuristic | 0.7 | 0.7 | 0.8 | 0.7 | 1.2 | 0.9 | 8.4 |
| | APSP | 0.7 | 0.7 | 0.7 | 0.7 | 1.0 | 0.9 | 8.2 |
| | PathEnum. | 0.8 | 0.8 | 0.9 | 0.8 | 1.1 | 0.8 | 8.6 |
| | TwoFlow | 416.0 | 419.6 | 413.1 | 413.1 | 418.0 | 413.6 | 422.5 |
| | FlowInFlow | 490.2 | 491.5 | 496.3 | 493.4 | 491.3 | 495.5 | 503.0 |
| 200 | None | 2.1 | 2.2 | 2.4 | 2.3 | 4.1 | 2.9 | 30.8 |
| | Heuristic | 2.2 | 2.2 | 2.6 | 2.5 | 4.2 | 3.1 | 31.0 |
| | APSP | 2.2 | 2.2 | 2.4 | 2.3 | 3.6 | 3.0 | 30.5 |
| | PathEnum. | 2.8 | 2.9 | 3.0 | 2.9 | 4.0 | 2.9 | 31.3 |
| | TwoFlow | 1468.8 | 1472.8 | 1473.4 | 1467.3 | 1465.6 | 1479.7 | 1490.0 |
| | FlowInFlow | 1759.9 | 1763.9 | 1755.5 | 1747.6 | 1764.4 | 1777.4 | 1787.7 |
| 500 | None | 11.5 | 11.5 | 14.4 | 12.8 | 31.5 | 26.1 | 286.0 |
| | Heuristic | 12.0 | 11.9 | 15.8 | 14.1 | 32.6 | 27.1 | 287.1 |
| | APSP | 11.9 | 12.1 | 13.9 | 12.8 | 28.2 | 24.9 | 285.7 |
| | PathEnum. | 23.3 | 24.3 | 25.2 | 24.0 | 42.0 | 26.6 | 302.4 |
| 1000 | None | 59.3 | 59.6 | 70.3 | 64.7 | 128.5 | 76.1 | 988.5 |
| | Heuristic | 61.0 | 61.4 | 75.0 | 69.5 | 132.1 | 81.4 | 993.2 |
| | APSP | 61.7 | 61.6 | 67.8 | 64.3 | 118.1 | 74.3 | 983.9 |
| | PathEnum. | 67.8 | 68.0 | 74.3 | 70.9 | 125.0 | 74.1 | 988.0 |

Table 9.9: Relative node pruning performance in percent for the presented pruning methods.

| | Size | | | | | | |
|---|---|---|---|---|---|---|---|
| **Pruning** | **20** | **30** | **50** | **100** | **200** | **500** | **1000** |
| None | 50.20 > | 47.37 > | 38.94 > | 29.32 > | 24.12 > | 20.64 > | 17.65 > |
| Heuristic | 50.20 > | 47.37 > | 38.94 > | 29.32 > | 24.12 > | 20.64 > | 17.65 > |
| APSP | 49.10 > | 45.66 > | 36.76 > | 27.34 > | 22.36 > | 18.31 > | 15.63 > |
| PathEnum. | 48.90 = | 45.29 = | 36.10 = | 26.80 = | 21.93 = | 18.24 = | 15.62 = |
| TwoFlow | 48.90 = | 45.29 = | 36.10 = | 26.74 = | 21.88 = | - | - |
| FlowInFlow | 48.90 = | 45.29 = | 36.10 = | 26.74 = | 21.88 = | - | - |

Table 9.10: Relative arc pruning performance in percent for the presented pruning methods.

| | Size | | | | | | |
|---|---|---|---|---|---|---|---|
| **Pruning** | **20** | **30** | **50** | **100** | **200** | **500** | **1000** |
| None | 33.28 > | 37.39 > | 34.03 > | 27.50 > | 24.51 > | 22.29 > | 18.85 > |
| Heuristic | 29.73 > | 33.18 > | 29.57 > | 24.64 > | 22.32 > | 21.25 > | 18.17 > |
| APSP | 27.58 > | 30.22 > | 25.99 > | 21.32 > | 19.52 > | 17.90 > | 15.61 > |
| PathEnum. | 26.88 = | 28.42 = | 23.94 = | 19.28 = | 17.49 = | 17.57 = | 15.56 = |
| TwoFlow | 26.88 = | 28.42 = | 23.94 = | 19.15 = | 17.34 = | - | - |
| FlowInFlow | 26.88 = | 28.42 = | 23.94 = | 19.15 = | 17.34 = | - | - |

in Table 9.8, may lie between 30% (size 100) or 200% (size 500). Whether the benefit is worth the additional run-time will have to be seen. Note that for sizes 100 and 200, PathEnumeration does not produce the same results as TwoFlow or FlowInFlow (but is not significantly different). This is due to the cutoff rule discussed in detail in Section 9.6.7. For the two largest size classes, the performance of PathEnumeration is quite close to its fallback methods APSP and Testing. Generally, APSP performs very similar to PathEnumeration. The difference is basically the fraction of nodes for which a path that is not simple and which fulfills the delay constraints exists. As a further observation, the heuristic pruning is not able to prune any nodes (in addition to None), which means that there never is a situation where a node in the extended component graph has no incoming or outgoing arcs so that it can be removed. This is also the reason why APSP does not include this step.

Table 9.10 shows the relative arc pruning performance $\text{PA}_{\text{rel}}$ for the presented pruning methods. We can again observe that the pruning efficiency increases with the instance size. In contrast to the node pruning performance, now we can see the value of the heuristic pruning. The difference in pruning performance when compared to None is caused by removing the incoming arcs of $s$ and the outgoing arcs of $t$ while calculating $\text{PA}^d_{s,t}$ within an extended component. With this rule alone we can remove about 4% of arcs from $\text{PA}_f$, which is reduced to 0.5% for the largest instance sizes. Considering the delays of paths by using APSP allows us to remove another 3% of arcs. And finally, by calculating the exact domain, we can remove an additional 2%. As

Table 9.11: Relative node fixing performance in percent for the presented fixing methods.

| | Size | | | | | | |
|---|---|---|---|---|---|---|---|
| **Fixing** | **20** | **30** | **50** | **100** | **200** | **500** | **1000** |
| None | 7.74 < | 6.60 < | 5.13 < | 2.95 < | 1.53 < | 0.59 < | 0.30 < |
| Dominators | 7.74 < | 6.60 < | 5.13 < | 2.95 < | 1.53 < | 0.59 < | 0.30 < |
| Testing | 7.78 = | 6.69 = | 5.22 = | 3.03 = | 1.61 = | 0.62 = | 0.31 = |
| TestingSAP-0 | 7.74 < | 6.60 < | 5.13 < | 2.95 < | 1.53 < | 0.59 < | 0.30 < |
| TestingSAP-1 | 7.78 = | 6.69 = | 5.22 = | 3.03 = | 1.61 = | 0.62 = | 0.31 = |
| PathEnum. | 7.78 = | 6.69 = | 5.22 = | 3.03 = | 1.61 = | 0.62 = | 0.31 = |
| FixFlow | 7.78 = | 6.69 = | 5.22 = | 3.03 = | 1.61 = | 0.62 = | 0.31 = |

Table 9.12: Relative arc fixing performance in percent for the presented fixing methods.

| | Size | | | | | | |
|---|---|---|---|---|---|---|---|
| **Fixing** | **20** | **30** | **50** | **100** | **200** | **500** | **1000** |
| None | 1.86 < | 1.67 < | 1.15 < | 0.67 < | 0.26 < | 0.10 < | 0.05 < |
| Dominators | 1.86 < | 1.67 < | 1.15 < | 0.67 < | 0.26 < | 0.10 < | 0.05 < |
| Testing | 1.94 = | 1.77 = | 1.26 = | 0.73 = | 0.30 = | 0.11 = | 0.06 = |
| TestingSAP-0 | 1.86 < | 1.67 < | 1.15 < | 0.67 < | 0.26 < | 0.10 < | 0.05 < |
| TestingSAP-1 | 1.86 < | 1.67 < | 1.15 < | 0.67 < | 0.26 < | 0.10 < | 0.05 < |
| PathEnum. | 1.94 = | 1.77 = | 1.26 = | 0.73 = | 0.30 = | 0.11 = | 0.06 = |
| FixFlow | 1.94 = | 1.77 = | 1.26 = | 0.73 = | 0.30 = | 0.11 = | 0.06 = |

with the node pruning performance, there is nearly no benefit in using PathEnumeration for the largest instance size when comparing to APSP, as a lot of calls to PathEnumeration end up being answered by APSP.

As we are able to exclude about 75% of nodes from $\mathrm{PN}_f$ and 80% of arcs from $\mathrm{PA}_f$, one could hope that we are also able to fix a significant number of nodes and arcs. Unfortunately, this is not the case, as Table 9.11 shows for the node fixing performance and Table 9.12 for the arc fixing performance. The presented results are based on using None as pruning method. For node fixing, we can see that for small instance sizes, we can fix about 1.5 nodes for every virtual arc (7.7% of 20 nodes) which increases to three nodes for every virtual arc for instances of size 1000. The advantage of exact methods, while statistically significant, is negligible. It can be seen that Dominators has exactly the same performance as None, so the extended component graph does not contain dominators. For our instances this is not surprising since for every arc in the substrate there also exists the reverse arc. Therefore, the extended component graphs are strongly node biconnected and no dominators exist. This method is only interesting in situations when reverse arcs are missing. Also TestingSAP-0 fails to find any nodes to fix.

For the arc fixing performance, the situation is basically the same, but the performance is even lower. The astute reader will have noticed that we only used None as pruning method for com-

Table 9.13: Average relative node pruning (PN), node fixing (FN), arc pruning (AP) and arc fixing (AF) performance of PathEnumeration for an individual substrate domain evaluation in percent.

| Size | PN | FN | PA | FA |
|---|---|---|---|---|
| 20 | 23.15 | 19.34 | 10.06 | 6.32 |
| 30 | 24.16 | 14.12 | 12.36 | 3.91 |
| 50 | 20.96 | 9.27 | 12.30 | 2.23 |
| 100 | 16.04 | 5.37 | 11.81 | 1.31 |
| 200 | 14.27 | 2.72 | 12.40 | 0.52 |
| 500 | 12.91 | 1.10 | 14.70 | 0.20 |
| 1000 | 11.47 | 0.56 | 13.07 | 0.09 |

paring the different fixing methods, but the fixing methods depend on the employed pruning method. We have already shown the influence on the run-time. As for the relative node fixing performance, there is no detectable difference depending on the employed pruning. The only observable difference is the arc fixing performance of TestingSAP-1. By using APSP or PathEnumeration as pruning method, it reaches the fixing performance of Testing. However, it requires much more run-time, which makes TestingSAP-1 uninteresting as fixing method and is the reason why we do not show the more detailed data.

A related question regarding the pruning and fixing capabilities of the different methods is the performance cost of having to combine the different substrate domains. Until now, we have only looked at the final virtual arc domains. However, they are of course weaker (i.e., less restrictive) than the substrate domains they are built upon. Table 9.13 shows the average domain pruning and fixing performance of PathEnumeration (used both for pruning and fixing) for individual substrate domain calculations, before they are combined into a virtual arc domain. For instance, it shows that for size 20, a delay restricted simple path in the substrate can use about 23% of the available substrate nodes and has to use 19% of the available substrate nodes. That means we are only uncertain about 4% of substrate nodes. This uncertainty increases with instance size. When we compare these values to $PN_{rel}$ and $FN_{rel}$, we see that due to the combination of substrate domains, we loose roughly half of the pruning and fixing performance, i.e., $PN_{rel}$ is doubled and $FN_{rel}$ is halved. This, however, can also be interpreted in a positive way. Once we are able to fix the locations of the source and the target node of a virtual arc, we can expect to be able to remove half of the nodes that we considered usable and we can double the number of nodes that we know we have to use.

Until now we have concentrated on the final result of the preprocessing algorithm. We will now focus on its inner workings, for example how many domains have to be calculated and what the benefit of memoization is. The results presented are based on the preprocessing algorithm using PathEnumeration as pruning and fixing method. Table 9.14 shows the characteristics of the preprocessing algorithm 9.3. The average VNMP instance of size 1000 contains 1700 virtual arcs, which means that we have to calculate 1700 virtual arc domains (CDom). Because of the high average number of mapping locations for virtual nodes (see Chapter 5), this corresponds

Table 9.14: Properties of the preprocessing algorithm 9.3: Average number of virtual arcs $|A'|$, calculated virtual arc domains (CDom), substrate domain requests (Dom Req), domain requests that were not memoized (DR Miss) and domain requests within a component (DR Comp).

| Size | \|A'\| | CDom | Dom Req | DR Miss | DR Comp |
|---|---|---|---|---|---|
| 20 | 432 | 432 | 6291 | 803 | 358 |
| 30 | 629 | 629 | 16588 | 2062 | 1659 |
| 50 | 947 | 947 | 60044 | 5548 | 6417 |
| 100 | 1753 | 1753 | 420592 | 19280 | 26830 |
| 200 | 1695 | 1695 | 1086983 | 63321 | 121473 |
| 500 | 1732 | 1732 | 3197825 | 313422 | 661395 |
| 1000 | 1723 | 1723 | 7843384 | 1047682 | 2456685 |

Table 9.15: Properties of algorithm 9.2 for calculating $\text{PN}^d_{s,t}$ within a component: Average number of domain requests that are memoized (Pos Hit), that are not memoized (Pos Miss), that have equal upper and lower bounds (Pos EqB), that have to be calculated by executing the pruning method (Pruning), the stored $\text{PN}^d_{s,t}$ within the components (Sto. D.) and the domain simplification efficiency for possible nodes.

| Size | Pos Hit | Pos Miss | Pos EqB | Pruning | Sto. D. | SEff[%] |
|---|---|---|---|---|---|---|
| 20 | 46 | 312 | 206 | 106 | 59 | 33.2 |
| 30 | 166 | 1493 | 1017 | 476 | 306 | 45.9 |
| 50 | 563 | 5854 | 4411 | 1442 | 946 | 47.4 |
| 100 | 2170 | 24660 | 20265 | 4395 | 2783 | 47.3 |
| 200 | 9672 | 111801 | 96415 | 15386 | 9133 | 44.4 |
| 500 | 83308 | 578087 | 501803 | 76284 | 52939 | 32.8 |
| 1000 | 323052 | 2133633 | 1931410 | 202223 | 138851 | 31.5 |

to calculating 7.8 million substrate domains (Dom Req). However, only 1 million of those are unique (DR Miss). By storing the result of the substrate domain calculation, we can reduce the number of calculations to a seventh of what we would have to do without memoization. For calculating the unique substrate domains, we have to calculate 2.4 million substrate domains within the crossed components (DR Comp). It can be seen that the path of a substrate connection in the simplified block tree crosses about 2.4 components on average. For smaller instances, fewer components have to be crossed. This is the point where execution of algorithm 9.2 begins to calculate $\text{PN}^d_{s,t}$ and $\text{FN}^d_{s,t}$ within a component.

The properties of the $\text{PN}^d_{s,t}$ calculation are shown in Table 9.15. Out of the 2.4 million substrate domain requests within a component, only 300000 can be served directly from memory (PosHit). For the remaining 2.1 million requests (Pos Miss), we now calculate upper and lower bounds for $\text{PN}^d_{s,t}$. For 1.9 million requests, the upper and lower bounds are equal (Pos EqB), so we know $\text{PN}^d_{s,t}$ and do not need to call the pruning method. Note that the distribution between direct

Table 9.16: Properties of algorithm 9.2 for calculating $FN^d_{s,t}$ within a component: Average number of domain requests that are memoized (Fix Hit), that are not memoized (Fix Miss), that have equal upper and lower bounds (Fix EqB), that have to be calculated by executing the fixing method (Fixing), the stored $FN^d_{s,t}$ within the components (Sto. D.) and the domain simplification efficiency for fixed nodes.

| **Size** | **Fix Hit** | **Fix Miss** | **Fix EqB** | **Fixing** | **Sto. D.** | **SEff[%]** |
|---|---|---|---|---|---|---|
| 20 | 43 | 315 | 223 | 92 | 45 | 35.0 |
| 30 | 140 | 1519 | 1150 | 369 | 201 | 50.9 |
| 50 | 403 | 6014 | 5042 | 972 | 516 | 56.0 |
| 100 | 1136 | 25694 | 23252 | 2442 | 1142 | 58.5 |
| 200 | 2580 | 118893 | 111257 | 7636 | 3120 | 61.2 |
| 500 | 7361 | 654033 | 621246 | 32787 | 13097 | 60.9 |
| 1000 | 17699 | 2438987 | 2357299 | 81688 | 33148 | 59.5 |

domain hits and equal bounds is skewed because of the domain simplification that is performed at the end of algorithm 9.2. For the remaining 200000 requests, we have to actually calculate the domain by using the pruning method. Of those calculated domains, we have to store only 138000, which gives a simplification efficiency (SEff) of 30%, which means that we save 30% of memory.

The properties of the $FN^d_{s,t}$ calculation of algorithm 9.2 are shown in Table 9.16. For fixing nodes, even fewer requests can be served from memory (Fix Hit). However, the number of times we actually have to execute a fixing procedure is lower than for the pruning procedure, the rest of the domain requests can be answered due to equal bounds. In addition, the simplification efficiency is better than for the calculation of possible nodes. This is because generally, very few nodes can be fixed, so $FN^d_{s,t}$ stays the same for a wide range of delays, which makes simplification work very well.

To sum it all up, due to the design of the preprocessing algorithm, especially with respect to domain bounds and memoization techniques, the selected pruning method (PathEnumeration) is only called 200000 times and the fixing method 80000 times to calculate 7.8 million substrate domains. This is a reduction of the required computations of 97.4% for pruning and 99% for fixing. In addition, the substrate domain problems that have to be solved eventually are smaller than the original problem, since we only consider a component instead of the complete substrate graph, which also increases efficiency.

Based on the presented data regarding the node and arc pruning performances, one could ask if it is possible to prune the substrate graph of the input VNMP instance as a whole, i.e., remove nodes and arcs that do not occur in any $PN_f$ or $PA_f$. Related to this is which nodes or arcs have to be used, i.e., occur in any $FN_f$ or $FA_f$. To explore this, we executed the preprocessing algorithm using PathEnumeration as pruning and fixing method on the complete VNMP instance set with loads from 0.1 to 1. We tested with different loads because with fewer virtual networks the probability that some parts of the substrate cannot be used increases. The results are presented in Table 9.17. By $PN_S$ we denote the fraction of substrate nodes that occur in any $PN_f$, $PA_S$

Table 9.17: Influence of the VNMP instance load on the preprocessing algorithm's run-time ($t[s]$), the time required to calculate the simplified block tree ($t_b[s]$), the fraction of substrate nodes ($PN_S$) and arcs ($PA_S$) which are contained within at least one $PN_f$ or $PA_f$ and the fraction of substrate nodes ($FN_S$) and arcs ($FA_S$) which are contained within at least one $FN_f$ or $FA_f$.

| | | **Load** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Size** | **0.10** | **0.20** | **0.30** | **0.40** | **0.50** | **0.60** | **0.70** | **0.80** | **0.90** | **1.00** |
| **t[s]** | 20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 30 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 50 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| | 100 | 0.3 | 0.3 | 0.4 | 0.5 | 0.5 | 0.6 | 0.7 | 0.7 | 0.8 | 0.8 |
| | 200 | 1.0 | 1.3 | 1.5 | 1.7 | 1.9 | 2.1 | 2.2 | 2.5 | 2.7 | 2.9 |
| | 500 | 11.4 | 15.7 | 17.6 | 18.9 | 19.8 | 21.6 | 21.9 | 23.2 | 24.3 | 25.4 |
| | 1000 | 15.9 | 26.0 | 34.5 | 40.9 | 46.7 | 51.9 | 58.4 | 63.0 | 68.1 | 72.2 |
| **$t_b$[s]** | 20 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 30 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 50 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 100 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 200 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 500 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| | 1000 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 |
| **$PN_S$[%]** | 20 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 30 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 50 | 99.5 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 100 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 200 | 98.2 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 500 | 90.1 | 99.1 | 99.9 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 1000 | 86.9 | 97.0 | 99.4 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| **$PA_S$[%]** | 20 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 30 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 50 | 99.5 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 100 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 200 | 97.8 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 500 | 89.9 | 99.1 | 99.9 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| | 1000 | 87.1 | 96.7 | 99.3 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |
| **$FN_S$[%]** | 20 | 26.7 | 34.0 | 39.0 | 43.7 | 47.3 | 51.2 | 54.2 | 55.8 | 58.2 | 60.7 |
| | 30 | 28.9 | 34.6 | 38.4 | 43.4 | 46.4 | 48.1 | 50.4 | 52.2 | 54.7 | 56.6 |
| | 50 | 24.1 | 28.6 | 31.4 | 33.7 | 35.8 | 37.5 | 39.6 | 41.5 | 43.9 | 45.3 |
| | 100 | 21.7 | 25.9 | 27.5 | 29.9 | 31.7 | 33.3 | 34.6 | 35.7 | 37.1 | 38.2 |
| | 200 | 16.9 | 19.3 | 20.9 | 22.3 | 23.9 | 25.1 | 26.1 | 27.4 | 28.3 | 29.3 |
| | 500 | 9.0 | 11.0 | 12.1 | 13.1 | 13.9 | 14.9 | 15.6 | 16.4 | 17.1 | 17.8 |
| | 1000 | 5.4 | 6.7 | 7.5 | 8.4 | 9.0 | 9.6 | 10.1 | 10.5 | 11.0 | 11.5 |
| **$FA_S$[%]** | 20 | 15.4 | 19.7 | 22.8 | 25.7 | 28.0 | 30.5 | 32.3 | 33.6 | 35.0 | 36.4 |
| | 30 | 17.0 | 20.0 | 22.6 | 25.7 | 27.1 | 28.4 | 29.9 | 30.8 | 32.2 | 33.5 |
| | 50 | 14.8 | 17.7 | 19.6 | 21.2 | 22.5 | 23.5 | 24.8 | 25.9 | 27.2 | 28.1 |
| | 100 | 12.7 | 15.7 | 17.0 | 18.5 | 19.8 | 20.9 | 21.8 | 22.6 | 23.5 | 24.3 |
| | 200 | 8.2 | 10.2 | 11.7 | 12.8 | 13.8 | 14.6 | 15.3 | 16.1 | 16.8 | 17.4 |
| | 500 | 3.8 | 5.3 | 6.0 | 6.7 | 7.2 | 7.8 | 8.3 | 8.7 | 9.2 | 9.5 |
| | 1000 | 2.1 | 2.8 | 3.4 | 3.9 | 4.3 | 4.6 | 4.9 | 5.1 | 5.4 | 5.7 |

Table 9.18: Properties of the simplified block tree (average number of nodes N, number of component nodes C and articulation point nodes A), average number of fixed components (FC) and different ways of crossing (CR) them per virtual arc.

| Size | N | A | C | FC | CR |
|---|---|---|---|---|---|
| 20 | 6.8 | 6.1 | 0.7 | 0.5 | 2.5 |
| 30 | 11.2 | 9.7 | 1.5 | 0.9 | 4.3 |
| 50 | 18.6 | 15.8 | 2.8 | 1.5 | 4.2 |
| 100 | 35.0 | 30.8 | 4.2 | 1.7 | 3.5 |
| 200 | 71.8 | 62.6 | 9.2 | 2.3 | 4.6 |
| 500 | 154.9 | 134.9 | 20.0 | 2.4 | 7.9 |
| 1000 | 250.8 | 219.7 | 31.1 | 2.9 | 11.4 |

gives the same information about substrate arcs. The fraction of nodes that are fixed in any $\text{FN}_f$ is labeled by $\text{FN}_S$, for arcs by $\text{FA}_S$.

Before we have a look if pruning the substrate graph is feasible, we should analyze the required run-times. Until now we have only looked at instances with load 1, now we can see the run-time increase depending on the load. Significant run-times of the preprocessing algorithm only occur at size 200 and above. The interesting thing to note here is the sub-linear growth in required run-time. For size 1000, going from load 0.1 to 0.2 adds 10 seconds to the run-time, from 0.2 to 0.3 8.5 seconds, to 0.4 adds 6.4 seconds, and going from load 0.9 to 1 adds only 4.2 seconds. This is caused by the employed memoization and bounding techniques which get more efficient when more substrate domains have to be calculated. This is doubly true for PathEnumeration, since for evaluating the domain within a component from one source to one target, we have to enumerate the paths to all nodes within the component. This is inefficient if the domains to the other target nodes are never requested. So the 16 seconds required for load 0.1 is basically the warmup-time to calculate paths which get mostly reused when more virtual networks are added. Table 9.17 also shows that the time to build the simplified block graph (labeled $t_b$) is insignificant, even for the largest instances. Note that this time is independent of the load.

Going back to the substrate pruning idea, we see that this is not feasible. Only for the largest instance sizes and lowest loads some substrate nodes and arcs could be removed. Only for load 0.1 and sizes 500 and 1000 the savings are of a meaningful magnitude, around 10% for nodes and arcs. Far more interesting is the fraction of nodes $\text{FN}_S$ and and arcs $\text{FA}_S$ for which we know that we have to use them. In the extreme case of size 20 and full load, 60% of the substrate nodes and 36% of the arcs have to be used. For the largest instances, this is reduced to 11% for nodes and 6% for arcs. This information can be used for instance to derive lower bounds for the substrate usage cost.

To finalize the discussion of the preprocessing algorithm, we will have a look at miscellaneous properties that did not fit into the discussion before. They are presented in Table 9.18. Previously (in Table 9.1), we have only presented the average size of the simplified block tree over a limited range of instance sizes, Table 9.18 shows the size for all instance sizes (N). It can be seen that

the simplified block tree only requires 25% to 35% of the number of nodes in the substrate graph to describe the main structure of the substrate. The following two columns in the table show the distribution of nodes in the block tree. Most of the nodes represent articulation points (A), only 10% to 15% of the nodes represent components (C). Note that this number slightly underrepresents the number of components that are found while calculating the block tree, since components that are only connected via one articulation point to the rest of the graph get removed by the simplification procedure.
We have shown that only a few nodes can be fixed for virtual arcs. There is however a related property that we have not touched upon: fixed components. These are components that have to be touched by a virtual arc. This includes just using one node of the component and stands in contrast to the definition of crossed component, which excludes components of which only one node is used. This difference is essential when we take the resources within components into account. Even if a virtual arc only uses one node of a component, it consumes resources. This will become relevant in Chapter 10 for determining satisfiability of a VNMP instance. Table 9.18 shows that a virtual arc has to use between 0.5 and 2.9 components (column FC). The last column (CR) shows by how many ways these forced components can be traversed on average.

## 9.7 Conclusion

In this chapter, we have presented a preprocessing algorithm for VNMP instances. Its main aim is to calculate tight domains of the virtual arcs, i.e., which substrate nodes or arcs can be used by the implementation of a virtual arc and which substrate nodes or arcs have to be used. This information is especially useful for removing unnecessary variables and constraints in VNMP models.
The preprocessing algorithm works in two phases. The first phase is independent of the virtual network configuration of the instance. It calculates the simplified block tree of the substrate, which contains two types of nodes: Articulation points, which directly represent a node of the substrate graph, and component nodes, which represent a subgraph of the substrate whose shadow is biconnected. The results have shown that the simplified block tree is rather compact, requiring about a quarter of the substrate nodes to represent the global structure of the substrate.
The second phase of the preprocessing algorithm derives the domains for the virtual arcs by using the block tree as a guide. Just using the tree alone gives very good results, depending on the instance size 50% to 83% of substrate nodes and 67% to 81% of arcs can be excluded for each virtual arc as unusable. Note that these numbers are high, even though they are based on all mapping possibilities of the source and target node of the virtual arcs. We have demonstrated that when the locations are fixed, about half of the remaining nodes and arcs can be excluded. As for determining which nodes and arcs have to be fixed, only very few nodes and arcs can be identified per virtual arc.
To further improve this performance, we have introduced methods for determining possible nodes (the pruning methods) and fixed nodes (the fixing methods) within the graph represented by a component node of the block tree. We extend this graph by adding a node for every arc, which results in the extended component graph. This allows us to concentrate exclusively on

nodes. The pruning problem is $\mathcal{NP}$-complete, while fixing is in $\mathcal{P}$. For both problems we presented heuristic and exact solution methods and evaluated them. Based on the results, three interesting method selections for preprocessing can be identified: just using the simplified block tree (None as pruning and fixing method), which gives the best performance for invested time, using APSP as pruning and Testing as fixing method, which is the middle ground between run-time and pruning/fixing performance and using PathEnumeration for pruning and fixing for the best performance at the cost of doubling run-time (in relation to doing nothing) in the worst case. The ILP based pruning and fixing methods were not competitive, even though some effort was expended to increase their efficiency.

A very interesting point to note is that PathEnumeration beats the ILP methods to a large extent. This is surprising, since usually it is not a good idea to solve a problem by enumerating all solutions and one would expect that the more sophisticated methods perform better. There is a number of factors that make PathEnumeration perform so well. First of all, by enumerating simple paths originating at a specific node and limited by a certain delay, we do not only solve one pruning or fixing problem, we solve all pruning and fixing problems that use the originating node as source node and a delay not larger than the current bound. Also the ILP based approaches can solve more than one problem per execution (remember that a path proves for all contained nodes that they are possible in the case of pruning and that only nodes contained in a found path can be fixed), but not as many as PathEnumeration. For PathEnumeration, it is also possible to build upon previous executions when the delay bound is increased. This is not possible for the ILP based methods. The third factor is the sparseness of the extended component graphs which makes enumerating all paths feasible. We expect that the ILP based methods would not fail with denser graphs, but they will definitely require more memory and even more run-time. On the topic of memory, path enumeration starts to fail for larger instances due to its memory requirements, which is why we needed to add a fallback to simpler methods if the extended component graphs are too large. We performed some optimizations to reduce the memory requirements (like storing only unions and intersections of paths), but there is still significant room for improvement.

During the evaluation, we have shown some pruning and fixing methods that seem to be useless. We have included them because the pruning and fixing methods are applicable to more than just preprocessing. Due to external factors (for instance a partial solution for the VNMP instance), we might be able to deduce that a substrate node cannot be used any longer. Based on this new information, we can execute the pruning and fixing methods within the changed component and derive new information. In these cases, a fixing method like Dominators might be useful, even though it is not so during preprocessing, due to the structure of the VNMP instances.

We have shown that for good performance while executing the pruning and fixing methods it is essential that domain bounds are calculated and used, since a very large fraction of domain requests can be answered because the upper and lower bounds are equal. Even if they are not equal, they are useful to reduce the number of nodes for which we have to test whether they are possible or fixed. Memoization is a central component of fast preprocessing.

## 9.8 Future Work

In this section, we discuss possibilities for future research on the topic of preprocessing.

The preprocessing for VNMP instances produces a lot of useful information, especially when using Path Enumeration. It might be possible to develop specialized VNMP problem formulations that make use of this information, for instance a model that works on the simplified block tree with additional variables that select a specific path within crossed biconnected component.

One important caveat to the presented preprocessing method is that it works best with sparse graphs. A computational study is still necessary to determine the characteristics of this method for dense graphs, for applications outside of the telecommunication network domain. It is clear that the decomposition based on the block tree will not work any more, since we will most likely end up with a single biconnected component. Therefore well performing pruning and fixing algorithms will be key. We expect that methods that take the delay into account are going to perform far better in terms of pruning or fixing performance than the alternatives, but this also warrants further investigation.

We saw that the order in which the virtual arc domains are calculated can influence the execution speed. As a refinement, it could be tested if additional improvements can be achieved by looking at the target nodes of virtual arcs in addition of the source nodes. That means, when we know the virtual arcs that may start at a particular substrate node, we could group them according to their possible target nodes. For one particular pair of source and target nodes and the virtual arcs that may be implemented between them, it might be beneficial to impose an additional order based on the delay. Here different strategies (high delay first, low delay first, ...) could be tested. As an additional benefit, after we have solved the substrate domain problems for one particular source node in the substrate, it is possible to erase all memoized data associated with this node, since we will not need it again. This could make preprocessing applicable to even larger instances.

At the moment, the selection of pruning and fixing methods is static, with the exception of PathEnumeration, which falls back to other methods if the component graphs grow too large. It might be beneficial to add more fine-grained control here, and fall back in a staggered way, e.g., first try to apply PathEnumeration, if the problem is too complex fall back to Testing or APSP, if this still uses too much time, just use None. However, for the tested instance sizes, there was never a big run-time difference between applying Testing/APSP and None, so this might only be beneficial for even larger VNMP instances. Another possibility would be to apply PathEnumeration only for a fraction of calls to a pruning method to derive better bounds. It might also be possible to initialize the stored domains within a component with interesting values, for instance with the domains for all pairs of nodes while using the shortest possible delay as delay bound.

A refinement for the pruning methods could be using the All Pair Shortest Path pruning while keeping track of the shortest paths (instead of just their delays). If we accept a node, but the shortest paths from $s$ to this node and from this node to $t$ are not disjoint, then we use an exact method to determine if this node really belongs to $\mathrm{PN}^d_{s,t}$. Since the pruning performance of APSP is very close to the exact methods, it might be feasible to use an ILP based approach also for the largest instance sizes without too much of a performance hit.

We have discarded two ILP formulation ideas for the node testing problem because they did not fit our requirements. However, it might be interesting to see if they offer advantages for larger, denser graphs and in situations were reconfigurability is not a huge concern. For solving the FIXFLOW model, it was beneficial to only have one constraint which has to be removed instead of having $|V|$ constraints where coefficients have to be adapted. This modification might also be possible for the TWOFLOW and FLOWINFLOW models and speed up preprocessing.
A further idea to improve preprocessing efficiency while using ILP pruning or fixing would be collecting all required domains within components and then solving them in one go for the same $s$, $t$ and $k$ values, but with increasing $d$, because the possible nodes for low $d$ values are still possible for higher $d$ and do not need to be tested again. The same applies to fixed nodes, for increasing $d$, only nodes that have been fixed for lower $d$ need to be tested again. Depending on the number of different $d$ values that have to be tested, it might even be beneficial to solve another problem: what is the lowest $d$ such that a simple path from $s$ to $t$ over $k$ exists or what is the highest $d$ such that no simple path from $s$ to $t$ not containing $k$ exists respectively. With this, we would have to solve two optimization problems for each $k$, instead of $k$ satisfaction problems for each delay bound. This idea directly leads to a more space efficient method of storing domains for memoization. Currently, we store complete domains within components for one source-target node pair sorted by delay. Another possibility would be to store for each node (and source and target node) the minimum allowed delay required to make it possible and the maximum allowed delay to keep it fixed. Depending on the number of different values for the delay bound that occur this could lead to significant memory savings, but of course causes the domain extraction time to be linear in the number of nodes since we need to assemble the domain based on delay values.
We have shown that the combination of substrate domains to build the domains of virtual arcs doubles the possible nodes and arcs and halves the fixed ones. However, we usually combine a lot of different substrate domains (for size 1000 2500 for the expected case) to achieve this doubling. That means that each additional substrate domain only adds very little additional information. It might be possible to rank substrate domain calculation according to their potential of adding new information. For instance, if a substrate domain calculation is done for a connection that starts and ends at nodes for which no such calculation has been executed until now, then it has a high potential for adding new possible nodes/arcs and removing fixed nodes and arcs. It might be possible to calculate substrate domains exactly for the ones with the highest potential and then use the knowledge already gathered about the domain of the virtual arc to speed up the calculation of the complete domain. If we use the information about the already gathered domain only in this limited way, it might be possible to avoid the run-time penalty that we have observed.
A surprising number of substrate nodes or arcs can be fixed, i.e., they have to be used. This information might be useful for defining more intelligent neighborhoods, for instance the clearing neighborhood as discussed in Chapter 6 can skip evaluating nodes which have to be used anyway. Such nodes and arcs are also interesting in their own right. If they are removed from the substrate, the flexibility afforded by being able to chose mapping locations for virtual nodes is not enough to ensure a feasible solution to the VNMP instance, so they are very critical. Of course, when capacity constraints are considered also other nodes might be critical in this sense.

CHAPTER 10

# Constraint Programming

## 10.1 Introduction

In this chapter, we investigate Constraint Programming (CP) approaches for solving the VNMP. For a treatment of the basic working principles of CP, see Section 2.2.11. Section 10.2 introduces CP formulations for the VNMP. In Section 10.3, we apply the lessons learned while designing Construction Heuristics (Chapter 6) to devise heuristic branching rules that guide CP towards feasible solutions. Methods for strengthening propagation are discussed in Section 10.4. The analysis of the different improvements for the CP approach can be found in Section 10.5. Section 10.6 summarizes the results and Section 10.7 shows possible directions for future work.

## 10.2 Models

We will present two different CP models for solving the VNMP. Their main difference is the type of variables used for modeling. The presented model in Section 10.2.1 utilizes binary variables. For instance, that means for representing an implementing path for a virtual arc, we need variables for each substrate arc telling us whether this substrate arc is used to implement this particular virtual arc. Another possibility is using set variables as presented in Section 10.2.2. With set variables, we only need one variable to define the implementing path of a virtual arc. Usually, CP models based on set variables perform better, both in terms of run-time and memory requirements [46]. We will evaluate whether this is true for the VNMP models in Section 10.5.
The following CP models assume that preprocessing as defined in Chapter 9 has been executed. As a short summary, $\text{PN}_f \subseteq V$ is the set of substrate nodes that a virtual arc $f \in A'$ can use (the possible nodes). $\text{PA}_f \subseteq A$ is the set of substrate arcs that a virtual arc can use (the possible arcs). Correspondingly, $\text{FN}_f \subseteq V$ is the set of substrate nodes that a virtual arc has to use (the fixed nodes), and $\text{FA}_f \subseteq A$ is the set of substrate arcs that a virtual arc has to use (the fixed arcs). If no preprocessing has taken place, $\text{PN}_f = V$, $\forall f \in A'$, $\text{PA}_f = A$, $\forall f \in A'$ and $\text{FN}_f = \text{FA}_f = \emptyset$, $\forall f \in A'$.

Table 10.1: Variables of the CP model based on binary variables

| Variable | | Domain | Description |
|---|---|---|---|
| $x_i^k$ | $\forall (k,i) \in M$ | $[0,1]$ | Mapping for virtual nodes |
| $y_e^f$ | $\forall f \in A',\ \forall e \in \text{PA}_f$ | $[0,1]$ | Substrate arcs for virtual arcs |
| $z_i^f$ | $\forall f \in A',\ \forall i \in \text{PN}_f$ | $[0,1]$ | Substrate nodes for virtual arcs |
| $u_i^V$ | $\forall i \in V$ | $[0,1]$ | Used substrate nodes |
| $u_e^A$ | $\forall e \in A$ | $[0,1]$ | Used substrate arcs |
| $a_i^{\text{CPU}}$ | $\forall i \in V$ | $[0, \sum_{k \in V'} c_k]$ | Additional CPU resources |
| $a_e^{\text{BW}}$ | $\forall e \in A$ | $[0, \sum_{f \in A'} b_f]$ | Additional bandwidth |

### 10.2.1 Binary Model

To formulate a CP model for the VNMP based on binary variables, we need two sets of main variables. The first set of variables $x_i^k,\ \forall (k,i) \in M$, is used to specify the mapping of each virtual node. If $x_i^k$ is true, virtual node $k$ is mapped to substrate node $i$. We define $x_i^k = 0,\ \forall (k,i) \in (V' \times V) \backslash M$, to simplify the model definition. To define the paths used to implement the virtual arcs, we use variables $y_e^f,\ \forall f \in A',\ \forall e \in \text{PA}_f$. If $y_e^f$ is true, substrate arc $e$ is used to implement virtual arc $f$. These variables are sufficient to define a solution to the VNMP. However, to be able to express the constraints we need further auxiliary variables. First, there are variables $z_i^f,\ \forall f \in A',\ \forall i \in \text{PN}_f$. If $z_i^f$ is true, substrate node $i$ is crossed by the implementing path of virtual arc $f$. This information is required to define the CPU constraints. Variables $u_i^V,\ \forall i \in V$, are true for all substrate nodes that are used to host virtual nodes while variables $u_e^A,\ \forall e \in A$, are true for all substrate arcs which are used to implement virtual arcs. They are required to define the objective. To be able to buy additional CPU resources if necessary, we use integer variables $a_i^{\text{CPU}},\ \forall i \in V$. They specify for each substrate node how much additional CPU resources have been bought. Integer variables $a_e^{\text{BW}},\ \forall e \in A$, fulfill the same role for additional bandwidth for each substrate arc. A summary of the used variables and their domains is shown in Table 10.1.

The complete CP model for the VNMP based on binary variables (CPBIN) is defined by equations (10.1)–(10.11). We begin the discussion of the model with the objective function as shown in (10.1). The aim is to minimize the cost incurred by having to buy additional resources. If it is possible to realize the virtual network load with the available resources, we want to reduce the cost incurred by using the substrate network as much as possible. $K$ denotes a sufficiently large constant, for instance the sum of all $p_i^V$ and $p_e^A$.

$$\min \quad \sum_{i \in V} K p^{\text{CPU}} a_i^{\text{CPU}} + \sum_{e \in A} K p^{\text{BW}} a_e^{\text{BW}} + \sum_{i \in V} p_i^V u_i^V + \sum_{e \in A} p_e^A u_e^A \tag{10.1}$$

For a valid description of a VNMP solution, we must first enforce that every virtual node is mapped exactly to one substrate node.

$$\sum_{(k,i)\in M} x_i^k = 1 \qquad \forall k \in V' \tag{10.2}$$

The implementations of the virtual arcs have to be simple paths from the mapping location of the source of the virtual arc to the mapping location of the target of the virtual arc within the substrate. We use the idea of network flows to formulate this. Each virtual arc sends one unit of flow across the substrate network, which enters at the mapping location of the source of the virtual arc and leaves at the mapping location of the target. At the nodes we have flow conservation, which means that all incoming flow has to leave again. The flow defines the implementing path for each virtual arc. This is enforced by the following two constraints, which also link the flow to the $z_i^f$ variables. Remember that we have defined $x_i^k = 0$ for $(k,i) \notin M$.

$$\sum_{e\in \text{PA}_f | t(e)=i} y_e^f + x_i^{s(f)} = z_i^f \qquad \forall f \in A',\ \forall i \in \text{PN}_f \tag{10.3}$$

$$\sum_{e\in \text{PA}_f | s(e)=i} y_e^f + x_i^{t(f)} = z_i^f \qquad \forall f \in A',\ \forall i \in \text{PN}_f \tag{10.4}$$

The CPU load caused by hosting virtual nodes and routing data may not exceed the available resources. If necessary, more resources have to be bought. The amount of CPU resources that have to be bought for a particular substrate node are the resources that are missing after the mapped virtual nodes and all traversing virtual arcs have consumed the available resources, but at least zero. We have stated the CPU constraint in this way so that $a_i^{\text{CPU}}$ is always as small as possible. If we had chosen for example an inequality to formulate the CPU constraint, a complete assignment to the main variables ($x_i^k$ and $y_e^f$) would not cause $a_i^{\text{CPU}}$ to become assigned, it would just force a lower bound. Additional branching would be needed to find a valid assignment to $a_i^{\text{CPU}}$. Suffice it to say, this would be very inefficient, since we already know that the smallest possible value will be valid and also the best assignment according to the employed objective function.

$$\max(0, \sum_{(k,i)\in M} c_k x_i^k + \sum_{f\in A' | i\in \text{PN}_f} b_f z_i^f - c_i) = a_i^{\text{CPU}} \qquad \forall i \in V \tag{10.5}$$

In a similar fashion, the bandwidth required by virtual arcs crossing a substrate arc may not exceed the available resources. If necessary, more resources have to be bought.

$$\max(0, \sum_{f\in A' | e\in \text{PA}_f} b_f y_e^f - b_e) = a_e^{\text{BW}} \qquad \forall e \in A \tag{10.6}$$

The third resource we need to take care of is the delay. The implementing path for the virtual arcs may not exceed the respective allowed delays.

$$\sum_{e \in \text{PA}_f} d_e y_e^f \leq d_f \qquad \forall f \in A' \tag{10.7}$$

To make the objective function work as intended, we need to set $u_i^V$ if at least one virtual node is mapped to substrate node $i$.

$$\max(x_i^k \mid (k,i) \in M) = u_i^V \qquad \forall i \in V \tag{10.8}$$

In addition, we need to set $u_e^A$ if at least one virtual arc uses substrate arc $e$.

$$\max(y_e^f \mid e \in \text{PA}_f) = u_e^A \qquad \forall e \in A \tag{10.9}$$

As the last part of the model, we need to set variables which we know to be set based on the results of preprocessing.

$$z_i^f = 1 \qquad \forall f \in A',\ \forall i \in \text{FN}_f \tag{10.10}$$

$$y_e^f = 1 \qquad \forall f \in A',\ \forall e \in \text{FA}_f \tag{10.11}$$

### 10.2.2 Set Model

As with the discussion of CPBIN, before we can state the constraints we need to define the used variables. To define a VNMP solution, we again need two sets of main variables. The set variables $P_f,\ \forall f \in A'$, define for each virtual arc the set of substrate arcs that are used to implement the virtual arc. The set variables $X_i,\ \forall i \in V$, define for each substrate node which virtual nodes it hosts. Note that this is an indirect way of specifying the locations of the virtual nodes. Defining it this way will become useful later when stating the CPU constraint for substrate nodes.
In addition to those main variables, we need auxiliary variables that allow us to formulate the resource constraints. To define the set of virtual arcs that utilize a substrate arc, we use variables $Y_e,\ \forall e \in A$. These variables are in some sense the duals of $P_f$, so they cannot express any additional information. However, we need both variable types to specify (and implement) constraints, $P_f$ to define the delay constraints and $Y_e$ for the bandwidth constraints. Variables $Z_i,\ \forall i \in V$, contain for each substrate node the virtual arcs that are crossing it. The remaining variables are the same as for CPBIN. Binary variables $u_i^V,\ \forall i \in V$, to specify which substrate nodes are used and $u_e^A,\ \forall e \in A$, for the used substrate arcs. Integer variables $a_i^{\text{CPU}},\ \forall i \in V$, denote bought CPU resources for each substrate node, variables $a_e^{\text{BW}},\ \forall e \in A$, bought bandwidth capacities for each substrate arc. To simplify stating the model we define the following four (constant) sets: $\delta_k^+$ and $\delta_k^-,\ \forall k \in V'$, the sets of virtual arcs leaving and entering a virtual

Table 10.2: Variables and auxiliary constants of the CP model based on set variables

| Variable | | Domain | Description |
|---|---|---|---|
| $P_f$ | $\forall f \in A'$ | $[\emptyset \,..\, \text{PA}_f]$ | Substrate arcs for virtual arcs |
| $X_i$ | $\forall i \in V$ | $[\emptyset \,..\, \{k \mid (k,i) \in M\}]$ | Virtual nodes mapped to a substrate node |
| $Y_e$ | $\forall e \in A$ | $[\{f \in A' \mid e \in \text{FA}_f\} \,..\, \{f \in A' \mid e \in \text{PA}_f\}]$ | Virtual arcs using a substrate arc |
| $Z_i$ | $\forall i \in V$ | $[\{f \in A' \mid i \in \text{FN}_f\} \,..\, \{f \in A' \mid i \in \text{PN}_f\}]$ | Virtual arcs using a substrate node |
| $u_i^V$ | $\forall i \in V$ | $[0,1]$ | Used substrate nodes |
| $u_e^A$ | $\forall e \in A$ | $[0,1]$ | Used substrate arcs |
| $a_i^{\text{CPU}}$ | $\forall i \in V$ | $[0, \sum_{k \in V'} c_k]$ | Additional CPU resources |
| $a_e^{\text{BW}}$ | $\forall e \in A$ | $[0, \sum_{f \in A'} b_f]$ | Additional bandwidth |
| $\delta_k^+$ | $\forall k \in V'$ | $\{f \in A' \mid s(f) = k\}$ | Outgoing arcs of a virtual node |
| $\delta_k^-$ | $\forall k \in V'$ | $\{f \in A' \mid t(f) = k\}$ | Incoming arcs of a virtual node |
| $\delta_i^+$ | $\forall i \in V$ | $\{e \in A \mid s(e) = i\}$ | Outgoing arcs of a substrate node |
| $\delta_i^-$ | $\forall i \in V$ | $\{e \in A \mid t(e) = i\}$ | Incoming arcs of a substrate node |

node and $\delta_i^+$ and $\delta_i^-$, $\forall i \in V$, the sets of substrate arcs leaving and entering a substrate node. A summary of the used variables, constants and their domains is shown in Table 10.2.
The complete CP model for the VNMP based on set variables (CPSET) is defined by equations (10.12)–(10.20). We begin the discussion of CPSET with the objective function as shown in (10.12). As before, the aim is to minimize the cost incurred by having to buy additional resources. If it is possible to realize the virtual network load with the available resources, we want to reduce the cost incurred by using the substrate network as much as possible.

$$\min \quad \sum_{i \in V} Kp^{\text{CPU}} a_i^{\text{CPU}} + \sum_{e \in A} Kp^{\text{BW}} a_e^{\text{BW}} + \sum_{i \in V} p_i^V u_i^V + \sum_{e \in A} p_e^A u_e^A \tag{10.12}$$

One type of constraint that we will make heavy use of during the definition of the constraints for CPSET is the disjoint union, which we denote by the symbol $\sqcup$. Its semantics are the same as the regular union, with the additional constraint that the sets being combined are disjoint. As an example, the constraint $A \sqcup B = C$ states that the set $C$ contains all elements from sets $A$ and $B$ and each element within $C$ has to have a unique source, i.e., either $A$ or $B$ but not both. This allows us to state the mapping constraint as shown in equation (10.13) in an elegant way. All virtual nodes have to be hosted on exactly one substrate node.

$$\bigsqcup_{i \in V} X_i = V' \tag{10.13}$$

The CPSET model is based on the idea of network flows. The flow conservation constraints are shown in equations (10.14) and (10.15). They state that all virtual arcs that use one of the incoming arcs of a substrate node together with all virtual arcs whose source is mapped to the substrate node give the virtual arcs that are traversing the substrate node. All traversing virtual arcs need to leave either by using an outgoing arc or the mapping of the target of the virtual arc. By using the disjoint union we enforce that the implementing path for every virtual arc is simple.

$$\bigsqcup_{e \in \delta_i^-} Y_e \sqcup \bigsqcup_{k \in X_i} \delta_k^+ = Z_i \qquad \forall i \in V \tag{10.14}$$

$$\bigsqcup_{e \in \delta_i^+} Y_e \sqcup \bigsqcup_{k \in X_i} \delta_k^- = Z_i \qquad \forall i \in V \tag{10.15}$$

The following equalities take care of the resource constraints. The amount of CPU resources that have to be bought for a particular substrate node are the resources that are missing after the mapped virtual nodes and all traversing virtual arcs have consumed the available resources, but at least zero.

$$\max(0, \sum_{k \in X_i} c_k + \sum_{f \in Z_i} b_f - c_i) = a_i^{\text{CPU}} \qquad \forall i \in V \tag{10.16}$$

In a similar fashion, the bandwidth constraints state that the amount of extra bandwidth that has to be bought for a substrate arc is the bandwidth that is missing after the virtual arcs using that arc have used up the available resources, but at least zero.

$$\max(0, \sum_{f \in Y_e} b_f - b_e) = a_e^{\text{BW}} \qquad \forall e \in A \tag{10.17}$$

The delay constraints can be stated in a straight forward manner, the delay of all substrate arcs used to implement a virtual arc may not exceed the maximum allowed delay for the virtual arc.

$$\sum_{e \in P_f} d_e \leq d_f \qquad \forall f \in A' \tag{10.18}$$

To make the objective function work as intended, we need to connect the mapping decisions with the variable that is used to track whether a substrate node is used to host a virtual node. If the set of hosted virtual nodes is not empty, then the substrate node is used.

$$\min(1, |X_i|) = u_i^V \qquad \forall i \in V \tag{10.19}$$

In the same way, we connect the substrate arc usage to the virtual arcs using the substrate arc. If the set of virtual arcs using a substrate arc is not empty, then the substrate arc is used.

$$\min(1, |Y_e|) = u_e^A \qquad \forall e \in A \tag{10.20}$$

As a last step, we need to add the channeling between the substrate arcs used to implement the virtual arcs and the virtual arcs crossing substrate arcs.

$$e \in P_f \Leftrightarrow f \in Y_e \qquad \forall e \in A,\ \forall f \in A' \tag{10.21}$$

This completes the CPSET model. Note that in comparison to CPBIN, we do not need to add additional constraints to include information about fixed nodes or arcs. The domains calculated for the virtual arcs during preprocessing can be used directly to define the domains of the CPSET variables (see Table 10.2); no further constraints or considerations are necessary.
An additional characteristic of CPSET when compared to CPBIN is its greater compactness. For instance, CPBIN requires one constraint per virtual node to ensure valid mappings, for CPSET a single constraint is sufficient. Flow conservation only requires two constraints for every substrate node, while CPBIN needs $2|A'|$. On the other hand, we need an additional type of variables ($P_f$) to formulate a valid model of the VNMP. In Section 10.5 we will analyze the practical difference between CPBIN and CPSET.

## 10.3 Heuristic Branching

When trying to find solutions to the VNMP by using the CPBIN or CPSET formulations, we need to define a branching strategy. After propagation has been performed and the domains of the variables cannot be reduced any longer (and not all variables have been assigned a value), branching needs to be performed to basically try out some assignments to a variable and see if this leads to a valid solution.
As default strategy for CPBIN and CPSET we use the following. We first try to assign the mapping variables ($x_i^k$ and $X_i$). During branching, we select the unassigned mapping variable with the highest degree (i.e., occurs in the most constraints), and assign it the value one (for CPBIN) or one virtual node from its domain (for CPSET). In effect, with this branching decision we fix the mapping of one virtual node.
Once the mapping has been fixed, we branch on the variables concerned with the implementing path of a virtual arc ($y_e^f$ and $P_f$). We select an arbitrary variable and set its value to zero (CPBIN), or remove a substrate arc (CPSET). In effect, we forbid the usage of a substrate arc for a virtual arc. This might seem counter-intuitive, as we would like to directly fix an implementing path instead of removing substrate arcs until only one path remains. However, it is not possible to specify a branching strategy in GECODE that builds implementing paths in a logical way (e.g., for a virtual arc select the substrate arc going away from the mapping location of its source node, then the following arc and so on). Instead, arcs are basically randomly selected across the substrate network and only very late in the branching it is detected that the currently selected set of substrate arcs cannot be used to form a simple implementing path. Preliminary runs showed a very bad performance when using this kind of strategy.
However, also the strategy of forbidding the use of a single substrate arc for a virtual arc has its weakness. First of all, there are a lot of decisions required to reach a complete assignment for all variables, and secondly, the implementing paths are still not built in a coherent fashion. To alleviate those problems, we implemented a custom branching strategy, making use of the

results presented in Chapter 6. A branching decision is either the mapping of a virtual node or a complete assignment of an implementing path for a virtual arc. We used CH-O to calculate the branching decision. In the context of CP that means the following. As long as there is a virtual arc which is implementable (source and target nodes have been mapped and does not have a (complete) implementing path), implement a virtual arc, otherwise map a virtual node (arc emphasis). If we implement a virtual arc, we select the most delay constrained virtual arc from the virtual network that in total has the most stringent delay constraints. For this virtual arc, we need to find an implementing path. However, just generating one path as possible implementation would mean that the search is not complete (i.e., not all solutions can be found). Therefore, we create multiple paths by following all outgoing arcs from the mapping location of the source of the virtual arc (denoted as continuation arcs) and then finding a path that causes the least increase to the substrate usage cost. During branching, these alternatives are tried in order of the increase in substrate usage cost they cause. Still, this is not complete as we do not try all possible paths, so in case we cannot find a solution by using one of the calculated paths, we just assign one of the continuation arcs and complete the implementation of the virtual arc with a path in a subsequent branching decision.

When we have to map a virtual node, we select the node with the DLHeavyVN strategy, i.e., from the virtual network that is most delay constrained we select the node that has the highest CPU requirements (see Section 6.2). CH-O uses MostFree to determine the mapping location, i.e., the virtual node is mapped to the substrate node which still has the most resources left. In context of CP that means that we first try to map the virtual node to the substrate node with the most free resources. If we fail to find a valid solution using this decision, we try the substrate node with the second most free resources and so on.

By using this branching strategy, the performance of the CP models is at least as good as CH-O. We expect it to be better since we can make use of the results of propagation and for instance exclude mapping possibilities that would have been chosen by CH-O.

## 10.4 Strengthening Propagation

The task of propagation is to reduce the domains of variables as much as possible, based on the imposed constraints. If the domain of a variable becomes empty, we know that the current assignment of values to variables is inconsistent and it is not possible to find a solution that satisfies all constraints with it. If the current assignment is inconsistent, we want to detect that as early as possible. A way to facilitate this would be using the DomReachability propagator [140, 141], which is used to find constrained paths in a network. However, this propagator has a high memory overhead. We would need to store three times a graph of the size of the substrate network for each of the virtual arcs. In addition, it is based on dominator trees, which cannot be used when we want to consider the delay constraints. These problems were actually one of the main reasons for developing the preprocessing methods as presented in Chapter 9, as we needed a method which is less sensitive to a high number of virtual arcs and can also consider the delay constraints. So as a first step to strengthen propagation, we remove useless values from the domains of the variables by using the presented preprocessing techniques.

This is something we can do once at the beginning when we start searching for a solution, but does not help during the search. Based on mapping decisions or chosen implementing paths, some nodes or arcs might become forced for virtual arcs or might not be possible any longer. This is detected only in the simplest cases. To improve on this situation, it would of course be possible to re-apply the preprocessing methods on the residual substrate network (i.e., the substrate network with the capacities that are still available) and the virtual arcs that still need to be implemented. This would mean either a very high overhead or a complex implementation making use of upper and lower domain bounds for virtual arcs not only according to the allowed delay, but also the available parts of the substrate network. Therefore, we chose a simpler approach. As outlined in Chapter 9, the preprocessing does not directly calculate the domains for the virtual arcs. Instead, it calculates domains for delay constrained paths in the substrate network and combines the relevant domains (depending on the possible mapping locations of the source and target node of a virtual arc) to derive the final domain of the virtual arc. Every time the mapping targets change, there is a possibility that the domain of the virtual arc changes. We chose to update the domain only when a mapping has been fixed, because then we have the highest probability of actually reducing the domain for the virtual arc.
In total, we update the domain of a virtual arc $f$ three times during the search for a solution. The first time at the beginning, based on the results of the preprocessing procedure. Then, when either $s(f)$ or $t(f)$ has been mapped and the last time when both virtual nodes have been mapped. These domain updates just reuse information already generated during preprocessing, so they are very fast and do not require any additional memory (aside from keeping the preprocessing information during search). As a downside, these domains are based on the initial resources available in the substrate network and not on the current resource levels.

## 10.5 Results

In this section, we evaluate the performance of the two proposed formulations for the VNMP and the various improvements described in the previous section. We will mainly focus on solving the VNMP-S. We do that by removing the objective we presented for the CPBIN and CPSET models and instead add a constraint forcing $C_a$ to be zero. Another approach would have been to minimize $C_a$. Preliminary runs have shown that this reduces the number of VNMP instances for which a valid solution can be found since the search gets stuck at finding an improvement to a bad solution (in terms of $C_a$) instead of finding a valid one.
We test the following configurations:

**CPB-B** The CPBIN model with standard branching and no improvements.

**CPS-B** The CPSET model with standard branching and no improvements.

**CPS-PE** The CPSET model using PathEnumeration as pruning and fixing strategy for preprocessing.

**CPS-BR** The CPS-PE configuration, but in addition using the heuristic branching introduced in Section 10.3.

Table 10.3: Performance of different CP configurations depending on instance size.

| | Size | CPB-B | CPS-B | CPS-PE | CPS-BR | CPS-VA | CPS-NE | CPS-OP |
|---|---|---|---|---|---|---|---|---|
| **# Valid** | 20 | 41 | 38 | 80 | 110 | 111 | 108 | 112 |
| | 30 | 15 | 12 | 48 | 104 | 105 | 99 | 105 |
| | 50 | 0 | 0 | 23 | 98 | 102 | 100 | 102 |
| | 100 | 0 | 0 | 6 | 77 | 81 | 78 | 81 |
| **Mem.** | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 30 | 43 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 50 | 69 | 57 | 2 | 0 | 0 | 0 | 0 |
| | 100 | 112 | 60 | 38 | 23 | 21 | 10 | 21 |
| $\mathbf{C_u}$ | 20 | 1182.5 | 981.1 | 1011.2 | 1062.8 | 1067.5 | 1051.5 | 914.6 |
| | 30 | 1813.1 | 1238.2 | 1399.7 | 1543.4 | 1550.5 | 1504.7 | 1336.1 |
| | 50 | - | - | 2004.2 | 2332.3 | 2368.9 | 2324.4 | 2163.4 |
| | 100 | - | - | 3487.7 | 4203.7 | 4310.1 | 4222.6 | 4097.9 |
| **t[s]** | 20 | 6299.9 | 6810.1 | 3339.5 | 884.1 | 828.4 | 1066.0 | 6996.9 |
| | 30 | 8523.2 | 8890.8 | 5954.2 | 1332.0 | 1248.5 | 1751.5 | 8579.6 |
| | 50 | 9801.8 | 6300.8 | 7756.6 | 1831.3 | 1497.2 | 1664.8 | 9830.0 |
| | 100 | 9984.1 | 5659.1 | 6856.9 | 1822.2 | 1998.0 | 2668.1 | 8724.1 |
| $\mathbf{t_n}$**[ms]** | 20 | 1.1 | 1.1 | 0.9 | 0.6 | 0.6 | 0.7 | 0.8 |
| | 30 | 3.8 | 1.9 | 1.4 | 1.5 | 1.4 | 1.4 | 1.8 |
| | 50 | 13.1 | 4.2 | 2.7 | 2.8 | 2.5 | 2.5 | 3.3 |
| | 100 | 78.0 | 24.7 | 9.1 | 6.4 | 5.7 | 6.9 | 7.3 |
| **M. Peak [MB]** | 20 | 570 | 105 | 23 | 38 | 42 | 28 | 95 |
| | 30 | 1127 | 653 | 120 | 89 | 96 | 70 | 216 |
| | 50 | 1587 | 210 | 446 | 231 | 249 | 183 | 414 |
| | 100 | 3124 | 462 | 440 | 769 | 835 | 654 | 1271 |
| **# Opt** | 20 | 0 | 0 | 0 | 0 | 0 | 0 | 37 |
| | 30 | 0 | 0 | 0 | 0 | 0 | 0 | 18 |
| | 50 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| | 100 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**CPS-VA** The CPS-BR configuration enhanced by using the propagation strengthening technique for virtual arcs outlined in Section 10.4.

**CPS-NE** The same configuration as CPS-VA, but the heuristic branching uses node emphasis (i.e., maps all virtual nodes first), instead of arc emphasis.

**CPS-OP** The CPS-VA configuration used for solving VNMP-O instead of VNMP-S.

The motivation behind this choice of configurations will become clear when the computational results are being discussed. We use VNMP instances of size 20, 30, 50, and 100, at loads 0.1, 0.5, 0.8, and 1 as test instances. A time-limit of 10000 seconds and a memory limit of 5 GB was employed.

Table 10.3 shows the performance of the different CP configurations depending on instance size. First, we will have a look at the basic models and their characteristics, i.e., configuration CPB-B using the CPBIN model and CPS-B using CPSET. When considering the number of valid solutions (out of 120 instances) found by the CPBIN and CPSET models (labeled # Valid), it can be seen that those numbers are very low, and that CPBIN has a slight advantage. Not a single valid solution can be found for instances of size 50 and 100.

In terms of the number of times the solution procedure had to be aborted due to the memory limit (labeled Mem.), we can see that CPSET is far better than CPBIN, for instances of size 100 nearly every execution of CPBIN has to be aborted, while CPSET succeeds at least for half of the instances. With respect to the average substrate usage cost $C_u$, we can see that the solutions found by CPSET are cheaper. However, these numbers are only based on valid solutions and as CPBIN produces more of those we cannot conclude that solutions by CPSET are cheaper.

The average required run-time is labeled by t[s] in the table. If the execution had to be aborted, we assume a run-time of 10000 seconds. We can observe that CPSET is faster than CPBIN. This statement is also supported by the average time required to perform propagation before another branching decision is necessary (labeled as node-time $t_n$). Note that the reported values of $t_n$ are only based on instances where the solution process was not aborted due to the memory limit.

With respect to the peak memory consumption (labeled M. Peak), we can again observe an advantage of CPSET. These values are based on instances where CP did not fail due to memory reasons. The last shown property, the number of instances that could be solved to optimality (labeled Opt.), is of course zero for CPB-B and CPS-B, as they solve VNMP-S and not VNMP-O.

To sum it all up, while CPBIN is able to solve a bit more instances, CPSET requires far less memory and also has faster propagation. Therefore, we considered CPSET to be more promising and continued with this model as basis for further improvements.

When we activate preprocessing (CPS-PE), we can observe a huge improvement in terms of performance. Now, CP is also able to find valid solutions for instances of size 50 and 100, and fails due to memory reasons only for a third of instances of size 100. Also the propagation time is decreased, since preprocessing can remove a lot of superfluous variables.

With heuristic branching (CPS-BR), we get another boost in performance. CP fails due to memory reasons only for the instances of size 100. It is interesting to see that in some cases the peak memory requirements have increased. This is due to two effects. First of all, these numbers are based on more, and more challenging, instances (since fewer executions fail due to the memory limit). Secondly, due to the employed heuristic branching scheme, we get far deeper in the search tree than with the standard branching, which gets stuck early on due to undetected inconsistencies. However, most of the performance increase can be explained by the capability of CH-O for finding valid solutions. CH-O on its own finds 99 valid solutions for size 20, 96 for size 30, 91 for size 50 and 77 for size 100. Only the remaining instances that could be solved benefit from CP, for size 100, there is no benefit at all.

The performance of CP can be increased a little bit by using improved propagation (CPS-VA). Two instances do not fail any more due to memory limits and four more valid solutions can be found for the largest instance size. It is remarkable that the average propagation time does not increase, even though we perform more propagation. The peak memory requirements on the

Table 10.4: Performance of different CP configurations depending on instance load

| | Size | CPB-B | CPS-B | CPS-PE | CPS-BR | CPS-VA | CPS-NE | CPS-OP |
|---|---|---|---|---|---|---|---|---|
| **# Valid** | 0.10 | 19 | 19 | 66 | 120 | 120 | 120 | 120 |
| | 0.50 | 16 | 15 | 46 | 119 | 119 | 118 | 119 |
| | 0.80 | 13 | 11 | 31 | 101 | 106 | 99 | 106 |
| | 1.00 | 8 | 5 | 14 | 49 | 54 | 48 | 55 |
| **Mem.** | 0.10 | 30 | 24 | 0 | 0 | 0 | 0 | 0 |
| | 0.50 | 69 | 58 | 22 | 0 | 0 | 0 | 0 |
| | 0.80 | 72 | 29 | 16 | 3 | 2 | 1 | 2 |
| | 1.00 | 53 | 6 | 2 | 20 | 19 | 9 | 19 |
| $\mathbf{C_u}$ | 0.10 | 922.6 | 705.0 | 1166.3 | 1381.9 | 1382.0 | 1376.7 | 1236.7 |
| | 0.50 | 1471.4 | 1142.0 | 1489.4 | 2441.9 | 2440.8 | 2409.5 | 2217.2 |
| | 0.80 | 1607.7 | 1358.5 | 1579.3 | 2667.5 | 2740.4 | 2670.6 | 2516.3 |
| | 1.00 | 1713.4 | 1334.0 | 1475.5 | 2119.4 | 2319.8 | 2299.9 | 2115.4 |
| **t[s]** | 0.10 | 8230.2 | 7128.2 | 4455.1 | 0.1 | 0.1 | 0.0 | 6087.5 |
| | 0.50 | 8501.4 | 4707.4 | 4572.4 | 84.6 | 84.7 | 168.0 | 9407.4 |
| | 0.80 | 8753.7 | 6652.7 | 6287.2 | 1333.7 | 1000.6 | 1675.4 | 9733.1 |
| | 1.00 | 9123.6 | 9172.5 | 8592.5 | 4451.3 | 4486.7 | 5307.1 | 8902.6 |
| $\mathbf{t_n}$**[ms]** | 0.10 | 0.7 | 0.4 | 0.4 | 0.5 | 0.6 | 0.6 | 0.9 |
| | 0.50 | 7.0 | 1.1 | 1.2 | 1.8 | 1.9 | 1.9 | 2.9 |
| | 0.80 | 8.7 | 7.9 | 3.4 | 3.6 | 3.5 | 3.4 | 4.5 |
| | 1.00 | 13.1 | 11.2 | 7.0 | 5.1 | 3.9 | 5.4 | 4.4 |
| **M. Peak [MB]** | 0.10 | 616 | 252 | 89 | 12 | 13 | 10 | 30 |
| | 0.50 | 1461 | 434 | 289 | 199 | 209 | 156 | 513 |
| | 0.80 | 1273 | 540 | 325 | 450 | 490 | 354 | 778 |
| | 1.00 | 1046 | 278 | 281 | 395 | 443 | 395 | 552 |
| **# Opt** | 0.10 | 0 | 0 | 0 | 0 | 0 | 0 | 49 |
| | 0.50 | 0 | 0 | 0 | 0 | 0 | 0 | 7 |
| | 0.80 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| | 1.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

other hand increase due to the space required for preprocessing data. Now, CP is also able to solve more instances than CH-O for size 100.

The next tested configuration (CPS-NE) uses node emphasis instead of arc emphasis for the heuristic brancher. We tested this configuration because we suspected that there could be a possible benefit in combination with the better propagation. We propagate only if a virtual node has been mapped. If we map the virtual nodes early, then we get better domains for the virtual arcs faster, which might help with the search for valid solutions. However, the results show that this was not the case, fewer valid solutions could be found.

As the last experiment, we tried to tackle VNMP-O with the CPS-OP configuration. We can see that some instances can be solved to optimality, but these are not a significant fraction of the total VNMP instances. Based on the reported $C_u$ values, we can see an improvement of about 10% when compared to CPS-VA, at the expense of a six fold increase in required run-time.

Table 10.4 shows the performance of the different CP configurations depending on the load of the VNMP instances. Again we can observe the advantage of CPSET compared to CPBIN with respect to the number of memory aborts, especially for instances of high load. CPS-PE and CPS-BR give huge benefits across the board. It is interesting to see that CPS-BR fails more often for the instances of highest load than CPS-PE. Because of the heuristic branching, the search gets far deeper into the search tree until inconsistencies are encountered. That also means that the memory requirements are higher, which leads to more failed instances. With CPS-BR, CP is able to solve all instances of the lowest load. Improved propagation (CPS-VA) is only useful for instances of the highest load and using node emphasis causes performance regressions for those instances. CPS-OP can basically only prove optimality for instances of the lowest load levels.

## 10.6 Conclusion

In this chapter, we have introduced two Constraint Programming models for the VNMP, one based on binary variables and one based on sets. In addition, we presented methods for improving the performance of those approaches, such as utilizing preprocessing to reduce the number of variables, heuristic branching or additional propagation during search.

Our computational results showed that the formulation based on set variables is far better in terms of required memory than the formulation using binary variables, even though it requires more variable types. By using preprocessing, the memory requirements can be reduced further. The main component necessary for finding valid VNMP solutions turned out to be the use of the heuristic brancher. This brancher however was the sole determining factor if a valid solution could be found. Especially for instances of the largest size a solution was either found by following the first suggestion of the brancher or not at all. This situation could be slightly improved by using the advanced propagation. Only the smallest instances of lowest load could be solved to optimality.

Based on these results, the main problem of the CP approaches is weak propagation, i.e., inconsistent assignments are detected too late. This was also confirmed by inspection of the search tree. We saw cases where the branching decision that caused the partial assignment to be inconsistent was 50 levels above the location in the search tree where inconsistencies were detected. Even if the tree was binary, the search could never (in a practical amount of time) back-track to the wrong decision and revert it. In the following section, we will give some ideas on how to improve that situation.

The question that remains now is of course whether the CP approach is promising, since we have not presented another exact method for solving the VNMP. However, anticipating the results we will show in the following chapter, which is concerned with Integer Linear Programming approaches, the performance of CP is very bad. We were not even able to solve VNMP instances larger than 100 nodes. Nevertheless, the CP approach is interesting, since it does not require the constraints to be linear, which might make it the only promising exact solution method for extensions to the VNMP.

## 10.7 Future Work

We could show that the main problem of the CP method is the weak propagation. One possibility to alleviate this problem is to reapply the preprocessing methods during search, but as we have already pointed out, such an approach would have either a high overhead or be very intricate. Even if such an approach was used, it would still be focused only on one virtual arc. It is not possible to detect for instance that a group of nodes or arcs has not enough resources left to implement some virtual arcs. Just imagine three virtual arcs, each requiring one unit of bandwidth. The sources of the virtual arcs are mapped to the same substrate node, and all targets are mapped to another node. If there are two paths between those two substrate nodes, with a capacity of one unit of bandwidth each, it is immediately apparent that no solution can exist. The CP approach, even with the preprocessing techniques, is not able to detect this situation, so some techniques with a more global view are needed.
One possible approach would be to consider the subgraphs of the substrate network corresponding to one component of the block tree calculated during preprocessing. For each of the subgraphs, we keep track of the virtual arcs that need to cross it. Note that even in case the source and target node of a virtual arc have not been fixed, its implementation may be forced to cross a particular subgraph.
Based on this, we can define the following problem. We are given a subgraph of the substrate (with the currently remaining resources) and a set of virtual arcs that have to cross the subgraph. For each virtual arc, we have a set of configurations of how it may cross the subgraph, which are ultimately depending on the different mapping configurations for its source and target node. These configurations are pairs of nodes by which the virtual arc enters the subgraph and by which it leaves. A solution to this problem is a selection of one configuration for each virtual arc and an implementing path through the subgraph, such that the resource constraints are satisfied. In addition, the selection of configurations needs to be consistent. All virtual arcs with the same source node have to enter from the same substrate node and all virtual arcs with the same target node have to leave by the same node.
This problem can be solved at different levels of relaxation. We might employ single commodity flow algorithms or formulate this problem as an integer linear program. This approach might be strengthened further by also incorporating the mapping decisions, that means, when a virtual arc enters a subgraph due to mapping, then we have at the entering node not only the CPU cost caused by the transferred data, but also the cost for hosting the virtual node. We are convinced that this method can lead to further improvements, but in first prototype implementations inconsistencies were still not detected early enough.
As a further technique to improve performance, we could activate the advanced propagation techniques outlined in this section only when needed (i.e., when we start to find inconsistencies). The best thing to to in this case would be to track back in the search tree, and check every partial assignment (once again, but now with better propagation) for consistency, until we find a consistent assignment (or at least cannot prove inconsistency) and then continue with the search without the advanced propagation until it is needed again. However, this behaviour is not natively supported by GECODE and would require a custom search engine.

CHAPTER 11

# Mixed Integer Linear Programming

## 11.1 Introduction

In this chapter, we will introduce (Mixed) Integer Linear Programming (ILP) formulations for the Virtual Network Mapping Problem. Section 11.2 presents a multi-commodity flow based ILP formulation for the VNMP, while a Column Generation approach based on paths is shown in Section 11.3. Computational results can be found in Section 11.4. We conclude in Section 11.5 and give promising directions for future work in Section 11.6. A precursor to the work presented in this chapter was published in [88].

## 11.2 Multi-Commodity Flow Model

This section presents a multi-commodity flow ILP formulation for the VNMP. Its objective is to find the cheapest possible implementation of all virtual networks within the substrate (solving VNMP-O). Later on, we will present variants of this formulation with the aim of finding a feasible solution, and if this is not possible, the cheapest possible way of adding resources to the substrate so that all virtual networks fit into the substrate (VNMP-S).

For the following ILP model, we assume that preprocessing as defined in Chapter 9 has taken place. As a short summary, $\text{PN}_f \subseteq V$ is the set of substrate nodes that a virtual arc $f \in A'$ can use (the possible nodes). $\text{PA}_f \subseteq A$ is the set of substrate arcs that a virtual arc can use (the possible arcs). Correspondingly, $\text{FN}_f \subseteq V$ is the set of substrate nodes that a virtual arc has to use (the fixed nodes), and $\text{FA}_f \subseteq A$ is the set of substrate arcs that a virtual arc has to use (the fixed arcs). If no preprocessing has taken place, $\text{PN}_f = V,\ \forall f \in A'$, $\text{PA}_f = A,\ \forall f \in A'$ and $\text{FN}_f = \text{FA}_f = \emptyset,\ \forall f \in A'$.

The model utilizes decision variables $x_i^k \in \{0,1\},\ \forall (k,i) \in M$, to indicate where the virtual nodes are located in the substrate graph and $y_e^f \in \{0,1\},\ \forall f \in A',\ \forall e \in \text{PA}_f$, to indicate if a virtual connection is implemented by using a substrate connection. To simplify the model, we define $x_i^k = 0,\ \forall (k,i) \in (V' \times V) \backslash M$. The decision variable $z_i^f \in \{0,1\},\ \forall f \in A',\ \forall i \in \text{PN}_f$,

indicates that a substrate node is used to route a virtual connection. Further auxiliary decision variables are $u_i^V \in \{0,1\},\ \forall i \in V$, to indicate that a substrate node hosts at least one virtual node and $u_e^A \in \{0,1\},\ \forall e \in A$, to indicate that a substrate arc is used to implement at least one virtual connection.
The complete model is defined by inequalities (11.1)–(11.16).

$$\text{(FLOW)} \quad \min \quad \sum_{i \in V} p_i^V u_i^V + \sum_{e \in A} p_e^A u_e^A \tag{11.1}$$

$$\sum_{(k,i) \in M} x_i^k = 1 \qquad \forall k \in V' \tag{11.2}$$

$$\sum_{e \in \text{PA}_f | t(e)=i} y_e^f + x_i^{s(f)} - \sum_{e \in \text{PA}_f | s(e)=i} y_e^f - x_i^{t(f)} = 0 \qquad \forall f \in A',\ \forall i \in \text{PN}_f \tag{11.3}$$

$$\sum_{e \in \text{PA}_f | t(e)=i} y_e^f + x_i^{s(f)} \leq z_i^f \qquad \forall f \in A',\ \forall i \in \text{PN}_f \tag{11.4}$$

$$\sum_{(k,i) \in M} c_k x_i^k + \sum_{f \in A' | i \in \text{PN}_f} b_f z_i^f \leq c_i \qquad \forall i \in V \tag{11.5}$$

$$\sum_{f \in A' | e \in \text{PA}_f} b_f y_e^f \leq b_e \qquad \forall e \in A \tag{11.6}$$

$$\sum_{e \in \text{PA}_f} d_e y_e^f \leq d_f \qquad \forall f \in A' \tag{11.7}$$

$$y_e^f = 1 \qquad \forall f \in A',\ \forall e \in \text{FA}_f \tag{11.8}$$

$$z_i^f = 1 \qquad \forall f \in A',\ \forall i \in \text{FN}_f \tag{11.9}$$

$$x_i^k \leq u_i^V \qquad \forall (k,i) \in M \tag{11.10}$$

$$y_e^f \leq u_e^A \qquad \forall f \in A',\ \forall e \in \text{PA}_f \tag{11.11}$$

$$u_i^V \geq 0 \qquad \forall i \in V \tag{11.12}$$

$$u_e^A \geq 0 \qquad \forall e \in A \tag{11.13}$$

$$x_i^k \in \{0,1\} \qquad \forall (k,i) \in M \tag{11.14}$$

$$y_e^f \in \{0,1\} \qquad \forall e \in A,\ \forall f \in A' \tag{11.15}$$

$$z_i^f \in \{0,1\} \qquad \forall i \in V,\ \forall f \in A' \tag{11.16}$$

Table 11.1 gives a short summary of the used variables, constants, and functions.
The objective of FLOW (11.1) is to minimize the total cost incurred due to hosting virtual nodes on substrate nodes and using substrate arcs to implement virtual arcs. Equalities (11.2) ensure that each virtual node is mapped to exactly one substrate node, subject to the mapping constraints. The flow conservation constraints (11.3) make sure that for each virtual connection there is a connected path in the substrate network going from the location of the source of the virtual connection to the location of the target of the virtual connection. Linking constraints (11.4) make certain that variables $z_i^f$ are equal to one when the corresponding node is used to

Table 11.1: Summary of the used variables, constants and functions to define FLOW ($i \in V$, $e \in A$, $k \in V'$, $f \in A'$, $l \in A \cup A'$)

| Symbol | Meaning | Symbol | Meaning | Symbol | Meaning |
|---|---|---|---|---|---|
| $G(V, A)$ | Substrate graph | $G'(V', A')$ | Virtual graph | $x_i^k$ | Map node $k$ to $i$ |
| $c_i$ | Available CPU | $c_k$ | Required CPU | $y_e^f$ | Use arc $e$ for $f$ |
| $d_e$ | Delay | $d_f$ | Max. allowed delay | $z_i^f$ | Use node $i$ for $f$ |
| $b_e$ | Available bandwidth | $b_f$ | Required bandwidth | $u_i^V$ | Use node $i$ |
| $M$ | Set of allowed mappings | $s(l)$ | Source node of arc $l$ | $u_e^A$ | Use arc $e$ |
| $p_i^V$ | Node price | $t(l)$ | Target node of arc $l$ | | |
| $p_e^A$ | Arc price | | | | |

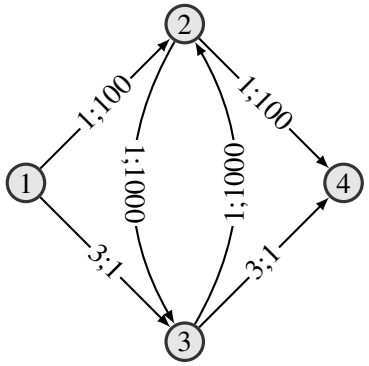



Figure 11.1: Substrate configuration showing that fixing nodes can strengthen FLOW. The arcs are labeled by their delay and costs.

route the traffic of a particular virtual connection. Since $z_i^f$ is binary, this also ensures that the implementing path of the virtual connection is simple. Inequalities (11.5)–(11.7) ensure that the solutions are valid with regard to CPU, bandwidth, and delay constraints. The constraints (11.8) and (11.9) incorporate the knowledge about fixed nodes and arcs into the model. To keep the exposition easy to follow, we chose to include superfluous variables that are explicitly set in the model instead of defining for instance $z_i^f$ only for arcs in $\text{PN}_f$ but not in $\text{FN}_f$. In addition, we know that only few nodes and arcs can be set, which limits the overhead caused by additional constraints. A discussion on the usefulness of those two constraint classes fill follow shortly. Linking constraints (11.10) and (11.11) force variables $u_i^V$ and $u_e^A$ to be (at least) one when the corresponding substrate node or arc is used by any virtual node or arc. Inequalities (11.12) and (11.13) ensure that the model is bounded even if some substrate nodes or arcs are not used by any virtual connection. The results presented in Chapter 9 show that this can occur for large instances from the VNMP instance set with low loads. Note that while the model only includes integrality constraints for $x_i^k$, $y_e^f$ and $z_i^f$ (11.14)–(11.16), constraints (11.10) and (11.11) together with the objective function (11.1) and the non-negativity constraints (11.12) and (11.13) also cause variables $u_i^V$ and $u_e^A$ to be integral (and binary).

The presented model makes good use of the information provided by preprocessing regarding the substrate nodes and arcs that might be used, by reducing the number of variables (see $y_e^f$) and constraints (see (11.3) and (11.4)) required to formulate the VNMP. By adding equations (11.8) and (11.9), we incorporate information about nodes and arcs that have to be used. Now the question is whether adding those constraints is actually beneficial. The first instinctive reaction would say that yes, adding these is beneficial because we constrain more and should be able to improve the formulation, which means a better (i.e., higher) lower bound on the solution cost derived via calculating the LP relaxation of FLOW. On second thought, however, we recall what it means for a node to be fixed: The implementation of the virtual arc has to use this node, i.e., there is no possible way of not using this node. Therefore, also the LP relaxation will have to use this node, even without adding an explicit constraint, so adding it does not improve the model. This argumentation is incorrect, because it neglects the interaction of the delay bound with the LP relaxation. Due to the nature of the LP relaxation, we can get multiple fractional flows as implementation of a virtual arc, instead of one unit flow. Referring to inequalities (11.7), we can see that crossing a substrate arc only fractionally also incurs the delay only fractionally. As an example, a flow of $0.5$ may cross an arc with a delay of 20 even if the delay bound is 10 (neglecting for now the other half of the flow which may incur additional delays). Therefore, we need to identify cases where a fractional flow can go around a fixed node or arc, because in these cases using (11.8) and (11.9) will be useful. One possible way of achieving this would be to use None as pruning and Testing as fixing method (as defined in Chapter 9), so arcs that are not actually possible remain within $\text{PN}_f$ which may then carry flow avoiding fixed nodes or arcs. But even with complete pruning, it is possible to avoid fixed nodes. We present an example for this in the following.

Figure 11.1 shows a substrate graph labeled with delay values and costs for using the arcs. We disregard bandwidths, CPU resources and node usage costs because they are not relevant for this example. Assume that a virtual arc needs to be implemented from node 1 to node 4 with a delay limit of 5. As a result, we know that node 2 is fixed. Note that this graph is perfectly pruned. It is easy to see that the optimal result of FLOW will be implementing the virtual arc via the path 1-2-4 for a total solution cost of 200. Now lets consider the LP relaxation of FLOW in this scenario, if we do not fix node 2. It will try to put as much flow as possible across the path 1-3-4, because this is the cheapest possibility. In this case, $\frac{3}{4}$ units of flow, i.e., $y^f_{(1,3)} = y^f_{(3,4)} = \frac{3}{4}$. The rest of the flow uses the path 1-2-4, i.e., $y^f_{(1,2)} = y^f_{(2,4)} = \frac{1}{4}$. In total, this gives a delay of 5 for the implementation of the virtual arc and a substrate usage cost of $51.5$, which is very far away from the integer optimal solution cost of 200. If we fix node 2, we require one unit of flow crossing node 2. With this additional constraint, the optimal solution of the LP relaxation will also be 200. This shows that equations (11.9) strengthen FLOW. A similar example can be constructed for equations (11.8). Note that for node fixing to work, inequalities (11.4) have to be equalities. In the presented form, we would only limit the incoming flow to one, but not force it to be one. We will show in Section 11.4 whether fixing nodes and arcs has any effect if inequalities (11.4) are used in their presented form.

As with previous network-flow based ILP formulations, we have the problem of disconnected flow circulations, i.e., there is the main flow from the appropriate source to the appropriate target node, but there may be also flow circulations disconnected from this main flow. Two

factors help reduce the occurrences of these circulations for FLOW. First of all, we have an objective function that tries to minimize substrate usage costs. Therefore, circulations can only occur within the part of the substrate network that is used by some part of the calculated VNMP solution. Note that if we had an objective that directly depends on the utilization of substrate nodes and arcs, circulations would not be possible for an optimal solution. The second factor is the delay constraint. Since any disconnected flow circulation also counts towards the delay limit, it puts a boundary on the length of the circulation. However, even though we have these mitigating factors, we still can have disconnected circulations. As a consequence, this requires a bit more thought when extracting a solution from the solved FLOW model, but this is not problematic. A bigger problem is that the circulations can circumvent the strengthening by fixed nodes or arcs. If a node or arc is fixed, it could simply cause a flow circulation, without having an influence on the main flow. The substrate shown in Figure 11.1 is too small to show this effect. Subtour elimination constraints or directed connection cuts could be used to solve this problem. Note that we do not need to remove all circulations, only those that involve fixed nodes. We will leave the inclusion of cuts as shown in (11.17) to FLOW as future work and concentrate on determining if fixing nodes and arcs has a measurable effect on the performance of FLOW. We expect that circulations involving fixed nodes only become a problem for the largest instances.

$$\sum_{e \in \{(i,j) \in \text{PA}_f | i,j \in S\}} y_e^f \leq |S| - 1 \qquad \forall f \in A',\ \forall S \subset \text{PN}_f : |S \cap \text{FN}_f| > 0 \tag{11.17}$$

The presented formulation of the VNMP has one weakness: it fails if a VNMP instance is unsolvable. By failing we mean it detects unsolvability, but does not give any more information, such as where additional resources might be bought so that all virtual networks can be implemented. We already used this aproach for the heuristic methods presented in Chapters 6, 7, and 8. We will now present a modification of FLOW that gives the cheapest possible way to make a VNMP instance feasible if it is not feasible and otherwise minimizes the substrate usage costs. We will call the adapted model FLOW-A.

FLOW-A requires additional information in the form of the cost of one unit of additional bandwidth $p^{\text{BW}}$ and the cost of one unit of additional CPU resources $p^{\text{CPU}}$. We also need two additional types of variables. Variables $a_i^{\text{CPU}},\ \forall i \in V$, to denote the amount of added CPU resources and $a_e^{\text{BW}},\ \forall e \in A$, to denote the amount of added bandwidth resources. The objective of FLOW-A is given in (11.18). $K$ denotes a sufficiently large constant, for instance the sum of all $p_i^V$ and $p_e^A$.

$$\min \quad \sum_{i \in V} K p^{\text{CPU}} a_i^{\text{CPU}} + \sum_{e \in A} K p^{\text{BW}} a_e^{\text{BW}} + \sum_{i \in V} p_i^V u_i^V + \sum_{e \in A} p_e^A u_e^A \tag{11.18}$$

To allow for added CPU capacity, inequalities (11.5) are replaced by inequalities (11.19).

$$\sum_{(k,i) \in M} c_k x_i^k + \sum_{f \in A' | i \in \text{PN}_f} b_f z_i^f \leq c_i + a_i^{\text{CPU}} \qquad \forall i \in V \tag{11.19}$$

For added bandwidth capacity, inequalities (11.6) are replaced by inequalities (11.20).

$$\sum_{f \in A' | e \in \text{PA}_f} b_f y_e^f \leq b_e + a_e^{\text{BW}} \qquad \forall e \in A \tag{11.20}$$

To forbid selling of resources, inequalities (11.21) and (11.22) are added to FLOW-A.

$$a_i^{\text{CPU}} \geq 0 \qquad \forall i \in V \tag{11.21}$$

$$a_e^{\text{BW}} \geq 0 \qquad \forall e \in A \tag{11.22}$$

In Section 11.4, we will show how the additional variables influence the behaviour of FLOW-A compared to FLOW.

## 11.3 Path-based Model

In this section, we present an ILP model for the VNMP based on paths. The main difference between this model and the one presented in Section 11.2 is that we now use decision variables to decide which path in the substrate implements a virtual connection, instead of deciding which substrate arc is part of the implementing arc. Since the set of paths may be exponential in size, so is this model (in principle). To get around this problem, we use delayed column generation (see Section 2.2.12).

Before we can show the complete model, some definitions are required. First, we need the extended graph $G_P(V_P, A_P)$ based on the substrate graph $G(V, A)$: $V_P = V' \cup V \cup V''$, where $V''$ contains a copy $\overline{k}$ of each node $k \in V'$, and $A_P = A \cup \{(k, i), (i, \overline{k}) \mid (k, i) \in M\}$. The delays of all additional arcs are zero, the available bandwidth larger than $\sum_{f \in A'} b_f$. By using this construction, the implementation of a virtual arc $f$ always has to connect the same nodes ($s(f)$ and $\overline{t(f)}$ in $V_P$), regardless of the mapping decision, which simplifies the model definition. Furthermore, since the virtual nodes contained in $G_P$ only have outgoing edges and the copies of the virtual nodes only have incoming edges, every path implementing a virtual arc is valid. Without the copies of the virtual nodes (i.e., incoming and outgoing arcs for virtual nodes) we could have paths that cross other virtual nodes before reaching the target node. On the topic of paths, for all virtual arcs $f \in A'$ let $P_f$ be the set of all simple paths from $s(f)$ to $\overline{t(f)}$ in $G_P$ not exceeding the delay bound $d_f$.

The model utilizes decision variables $w_{p_f} \in \{0, 1\}$, $\forall p_f \in P_f$ to indicate which path is chosen to implement virtual arc $f$. As for FLOW, we have variables $x_{ki}$ specifying if virtual node $k$ is mapped to substrate node $i$. The presented model is capable of adding resources if required (akin to FLOW-A), so we need variables $u_i^V$, $\forall i \in V$, to indicate if a substrate node is used to host virtual nodes, $u_e^A$, $\forall e \in A$, to indicate if a substrate arc is used to implement virtual arcs, $a_i^{\text{CPU}}$, $\forall i \in V$, to indicate the amount of added CPU resources and $a_e^{\text{BW}}$, $\forall e \in A$, to indicate the amount of added bandwidth resources. Like FLOW-A, $p^{\text{CPU}}$ denotes the cost of one additional unit of CPU resources, $p^{\text{BW}}$ denotes the cost of one additional unit of bandwidth and $K$ a sufficiently large constant.

The complete model is defined by inequalities (11.23)–(11.34). The variables in parenthesis denote the dual variables associated with the constraint.

$$\text{(PATH)} \quad \min \quad \sum_{i \in V} Kp^{\text{CPU}} a_i^{\text{CPU}} + \sum_{e \in A} Kp^{\text{BW}} a_e^{\text{BW}} + \sum_{i \in V} p_i^V u_i^V + \sum_{e \in A} p_e^A u_e^A \tag{11.23}$$

$$(\mu_f) \qquad \sum_{p_f \in P_f} w_{p_f} = 1 \qquad \forall f \in A' \tag{11.24}$$

$$(\pi_{fki}) \qquad \sum_{p_f \in P_f : (k,i) \in p_f \vee (i,\overline{k}) \in p_f} -w_{p_f} + x_{ki} \geq 0 \qquad \forall (k,i) \in M,\ \forall f \in A' \tag{11.25}$$

$$\sum_{(k,i) \in M} x_i^k = 1 \qquad \forall k \in V' \tag{11.26}$$

$$(\lambda_i) \quad -\sum_{f \in A'} \sum_{p_f \in P_f : i \in p_f} b_f w_{p_f} - \sum_{(k,i) \in M} c_k x_i^k \geq -c_i - a_i^{\text{CPU}} \qquad \forall i \in V \tag{11.27}$$

$$(\epsilon_e) \qquad \sum_{f \in A'} \sum_{p_f \in P_f : e \in p_f} -b_f w_{p_f} \geq -b_e - a_e^{\text{BW}} \qquad \forall e \in A \tag{11.28}$$

$$(\gamma_{ef}) \qquad \sum_{p_f \in P_f : e \in p_f} -w_{p_f} + u_e^A \geq 0 \qquad \forall e \in A,\ \forall f \in A' \tag{11.29}$$

$$-x_i^k + u_i^V \geq 0 \qquad \forall (k,i) \in M \tag{11.30}$$

$$a_i^{\text{CPU}} \geq 0 \qquad \forall i \in V \tag{11.31}$$

$$a_e^{\text{BW}} \geq 0 \qquad \forall e \in A \tag{11.32}$$

$$w_{p_f} \in \{0,1\} \qquad \forall f \in A',\ \forall p_f \in P_f \tag{11.33}$$

$$x_{ki} \in \{0,1\} \qquad \forall (k,i) \in M \tag{11.34}$$

The objective function (11.23) states that first the cost of adding additional resources has to be minimized. If those costs cannot be reduced any more, then the cost of using substrate components has to be reduced as much as possible. Equalities (11.24) ensure that for every virtual arc exactly one implementing path is chosen. The linking constraints (11.25) state that if a specific path is used, the corresponding mapping variables have to be set too. Equalities (11.26) state that exactly one mapping target for every virtual node has to be chosen. Inequalities (11.27) implement the CPU constraint, the bandwidth constraint is implemented by inequalities (11.28). The linking constraints (11.29) ensure that variables $u_e^A$ are set if paths are selected that use those arcs. Constraints (11.30) fulfill the same role for variables $u_i^V$, which are set if a virtual node is mapped to them. Inequalities (11.31) forbid selling CPU resources, inequalities (11.32) forbid it for bandwidth capacities. Note that while the model only includes constraints to restrict $w_{p_f}$ and $x_{ki}$ to binary values, due to the employed constraints and the objective function, $u_i^V$

and $u_e^A$ are binary as well. Since the CPU and bandwidth capacities are integral, $a_i^{\text{CPU}}$ and $a_e^{\text{BW}}$ are too. We do not need to add delay constraints since they are covered by the definition of $P_f$. The restricted master problem is derived from PATH by using only a small, nonempty subset $P_f'$ of $P_f$ to construct the model. For the pricing subproblem, we need to find a path $p_f$ with negative reduced costs. The reduced costs $\overline{c_{w_{p_f}}}$ of a path $p_f$ are calculated as follows:

$$\begin{aligned}\overline{c_{w_{p_f}}} &= 0 - (\mu_f - \sum_{(k,i)\in M|(k,i)\in p_f \vee (i,\overline{k})\in p_f} \pi_{fki} - \sum_{e\in p_f\cap A} b_f\epsilon_e - \sum_{e\in p_f\cap A} \gamma_{ef} - \sum_{i\in p_f\cap V} b_f\lambda_i) = \\ &= -\mu_f + \sum_{(k,i)\in M|(k,i)\in p_f \vee (i,\overline{k})\in p_f} \pi_{fki} + \sum_{e\in p_f\cap A} (b_f\epsilon_e + \gamma_{ef}) + \sum_{i\in p_f\cap V} b_f\lambda_i \qquad (11.35)\end{aligned}$$

Note that $\mu_f$ is unrestricted, while variables $\pi_{fki}$, $\epsilon_e$, $\gamma_{ef}$ and $\lambda_i$ have to be greater or equal to zero. Looking at the structure of equation (11.35), we can see that the costs are determined by two parts: one part ($\mu_f$) is constant for the virtual arc for which we are trying to find an improving path. The second part actually depends on the nodes and arcs used by the path. Since we add the values of the dual variables to the cost and the dual variables are always positive due to the way the corresponding constraints are stated, we need to minimize those costs. In effect, we are searching for a cheapest path within $G_P$ that satisfies the delay restrictions. The costs of the arcs $c$ are set as follows, the costs of nodes are added to all incoming arcs:

$$c_e^f = b_f\epsilon_e + \gamma_{ef} + b_f\lambda_{t(e)} \qquad \forall e \in A \qquad (11.36)$$

$$c_{ki}^f = \pi_{fki} + b_f\lambda_i \qquad \forall (k,i) \in M \qquad (11.37)$$

$$c_{i\overline{k}}^f = \pi_{fki} \qquad \forall (k,i) \in M \qquad (11.38)$$

Costs $c_e^f$ apply to arcs within $A_P$ that come from the substrate graph, costs $c_{ki}^f$ apply to arcs which connect a virtual node to the substrate and costs $c_{i\overline{k}}^f$ apply to arcs which connect the substrate to the copy of a virtual node. It is important to note that the costs can never be negative. Summing up, to create a path variable for a virtual arc $f$ that can potentially reduce the cost of the LP relaxation of PATH, we need to find a resource constrained shortest path in $G_P$ from $s(f)$ to $\overline{t(f)}$, with arc costs defined by equations (11.36)–(11.38). Note that in this case, the delay bound is the constrained resource. Since we are dealing with non-negative arc costs, we can use the Dynamic Program for the Resource Constrained Shortest Path Problem from [69] to identify such a path. If the sum of $-\mu_f$ and the cost of the path is negative, we have identified a path that we have to add to the model. If no such path exists for all $f \in A'$, then we have successfully solved the LP relaxation of PATH.

A central problem when applying delayed column generation is determining the initial set of variables, which has to be sufficient to allow a solution to the problem. For the VNMP, this condition is problematic, since just finding a valid solution (which could be used to initialize $P_f'$) is $\mathcal{NP}$-complete. One possible way around this problem would be the introduction of additional arcs between all pairs of substrate nodes with infinite bandwidth, zero delay, and very high costs. Paths using these arcs could be used to initialize $P_f'$.

By using the extension that we are allowed to purchase additional resources as needed it is possible to avoid that problem. A simple method for constructing initial paths could be to just randomly map virtual nodes to arbitrary substrate nodes (allowed by the mapping constraints) and implement every virtual arc with its delay shortest path. This path, together with the mapping decision, defines a path $p_f$ in $G_P$, which is the sole member of $P'_f$. This procedure assumes that for every virtual arc $f$ and all possible mappings of $s(f)$ and $t(f)$ a path within the delay bound $d_f$ exists. This condition holds for the VNMP instance set, otherwise selecting the mapping targets requires more thought or penalties for exceeding the delay. For a better selection of initial paths, the heuristics presented in the previous chapters can be used. These of course can also be used to speed up solving FLOW by supplying good upper bounds on the optimal solution. In Section 11.4.3 we will analyze the influence of an initial solution on the performance of FLOW, in Section 11.4.4 we will utilize CH-O (see Section 6.5.1) to derive an initial set of paths for PATH.

## 11.4 Results

In this section, we present the evaluation of the different ILP models for the VNMP. We executed the different models with a time-limit of 10000 seconds and a memory limit of 5 GB. The reported times include also the time necessary for preprocessing. We use the complete VNMP instance set with loads of 0.1, 0.5, 0.8 and 1. As constants we use $p^{\text{CPU}} = 1$, $p^{\text{BW}} = 5$, and $K = 1000000$. The motivation for the difference in cost between additional CPU and bandwidth resources is that it is easier to add another server than to add more bandwidth, which might mean constructing a new physical connection if other options like renting additional bandwidth are not available.
As for the different models, we will now compare the following FLOW configurations. A discussion of the influence of an initial solution on FLOW and the performance of PATH follows later in this section.

**FLOW-B** The basic FLOW model, without using any preprocessing.

**FLOW-P1** The FLOW model with activated preprocessing, but using None as pruning method. Preprocessing is executed with the same settings as outlined in Chapter 9. Fixing of nodes or arcs is not performed within FLOW. Nevertheless, the preprocessing algorithm requires a method specification for the fixing method. We use None, unless otherwise specified.

**FLOW-P2** FLOW-P1 with better preprocessing. APSP is used as pruning method.

**FLOW-P3** FLOW-P1 with the best preprocessing. PathEnumeration is used as pruning method.

**FLOW-F** FLOW-P3, but in addition, nodes and arcs are being fixed. PathEnumeration is used as fixing method.

**FLOW-FE** The FLOW-F configuration, but with equalities instead of the inequalities (11.4).

**FLOW-A** The FLOW-A model. Otherwise, the FLOW-FE configuration is used, i.e., preprocessing with PathEnumeration, nodes and arcs are fixed and node usage is linked to incoming flow by using equalities.

**FLOW-MA** A variant of FLOW-A which only minimizes the cost of additional resources to buy, solving VNMP-S. After this variant is solved, we either get a valid solution to the original VNMP instance (but not an optimal one) or the cheapest possible solution that requires additional resources if the instance is infeasible.

**FLOW-S** Another configuration that focuses on satisfiability instead of optimality. It is a variant of the FLOW-FE configuration, but removes the objective function. As with FLOW-MA, we get a valid solution to the original VNMP instance if possible. If the instance is infeasible, we get no hints about recourse actions to make it feasible.

### 11.4.1 Solving Characteristics of FLOW Configurations

In this section, we will compare the main solving characteristics of the different FLOW configurations. By solving characteristics we mean how far the different configurations get when trying to solve VNMP instances, for instance, do they find optimal solutions or do they run out of time before they are able to solve the LP relaxation in the root node. We also consider the standard characteristics like the required run-time or the gap between the lower bound and the best found integer feasible solution after all of the available run-time has been used. We will now discuss the different characteristics that we are going to report on in more detail.
When trying to solve a VNMP instance with one of the configurations of FLOW, we can either succeed (i.e., find an optimal solution) or we can fail to varying degrees. The most serious failure is exceeding the memory limit. Due to technical limitations, we do not get any information about the solving process in this case, even though useful results might have been created before the memory limit was hit. We will denote this type of failure by Mem. The next type of failure is failing to solve the LP relaxation of the root node in the Branch-and-Bound tree, which means we do not get any lower bounds. We will label this type of failure LP. If we succeed in solving the root node and start branching, we can fail to find an integer feasible solution to the VNMP instance that we are trying to solve (denoted by NS). Depending on the configuration of FLOW, it might be possible to buy additional resources. Even if we found an integer feasible solution, it might not be valid for the original VNMP instance because additional resources need to be bought. This type of failure we will denote with AR. If none of these failure reasons apply, then we succeeded in finding an integer feasible solution to the VNMP instance that we are trying to solve. However, we may fail to prove the optimality of this solution (if it is the optimal solution). We will call this condition Feas. If none of the previous failures apply, then we have found an optimal solution to the VNMP instance we are trying to solve, denoted by Opt. Since FLOW-S does not have an objective to optimize, it counts as Opt if a feasible solution has been found.
In addition to the state that the solving process ended in, we will also report on other characteristics, the first of which is the achieved gap between the final lower bound (lb) on the solution cost and cost of the best found integer feasible solution (ub). The gap (in %) is calculated as $100\frac{\text{ub}-\text{lb}}{\text{ub}}$. A gap of 1% means that the best found solution can be improved by at most 1%. If

we do not have access to lower or upper bounds (i.e., we failed before Feas), we assume a gap of 100%. We also report on the achieved objective values (Obj). However, for the objective we cannot define a useful default value if we fail before Feas, so the reported objective values are only based on instances where at least a feasible solution not requiring additional resources was found. The required run-time t in seconds is not as problematic, if we fail due to Mem we assume a run-time of 10000 seconds. For gap, objective, and run-time, we present average values. Last but not least, we are going to report on the number of variables (Vars) used to model the VNMP instances. This allows us to judge the effectiveness of preprocessing in terms of model size. Note that we report the size of the model after CPLEX has performed its own reductions. That has two consequences. First of all, this allows us to really see the benefit of preprocessing. If we would report on the size of the model before the reductions, any reduction in the number of necessary variables caused by preprocessing could in principle be matched by the reductions performed by CPLEX. Preprocessing only adds real value if it reduces the number of variables in addition to the reductions of CPLEX. The second consequence and downside of using the number of variables after the reductions is that the results are not necessarily consistent. We have seen instances where the final number of variables is smaller if weaker preprocessing is used. Of course, the number of variables in a model is only an indicator for the "hardness" of a model but by no means the sole determining factor. We could observe instances where a smaller model for the same instance took far longer to solve to optimality. The reported numbers of variables are based on instances that did not fail due to Mem.

Based on these definitions, we can present the main characteristics of different configurations of FLOW depending on the instance size in Table 11.2.

For the smallest instance sizes, we can see that every configuration is able to solve all of the 120 instances (30 instances with four load levels each). The average time required to to so however is very different. Unsurprisingly, FLOW-B requires the most run-time with more than 100 seconds on average. By adding preprocessing without any pruning and fixing, the run-time requirements can be reduced to one third of the original run-time.

Interestingly, adding additional pruning capabilities increases the required run-time in some cases. The reason is not the run-time required by the enhanced preprocessing methods, as we know from Chapter 9 that they are negligible compared to the total run-time. The reason is the performance variability of ILP [36, 109]. Simple changes to a model which should be performance neutral (or improve performance) like changing the order of constraints may sometimes cause unexpected degradations. At least the number of variables present in the ILP model is reduced by using enhanced preprocessing methods. Starting to fix nodes gives another boost to the preprocessing performance. FLOW-F only requires less than 17% of the run-time of FLOW-B, but linking flow with node usage by equality does not give an additional advantage. By adding the possibility of buying additional resources, we slow the solving process down significantly. It also causes a noticeable increase in variables necessary for modeling the VNMP instances. The two configurations that focus on finding a valid solution, FLOW-MA and FLOW-S, unsurprisingly are much faster than the optimizing configurations. The solutions that are found by them are about 50% more expensive than the optimal solutions with respect to substrate usage costs.

Starting with instance size 30, the configurations begin to fail. Most noticeable of them is the FLOW-B configuration, which is not able to find an integer feasible solution for one instance.

Table 11.2: Main characteristics of different configurations of FLOW depending on the instance size.

| Size | Method | Mem | LP | NS | AR | Feas | Opt | Gap[%] | Obj | t[s] | Vars |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | FLOW-B | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 803.2 | 102.2 | 6269.7 |
| | FLOW-P1 | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 803.2 | 28.6 | 2437.8 |
| | FLOW-P2 | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 803.2 | 34.6 | 2039.0 |
| | FLOW-P3 | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 803.2 | 29.6 | 1962.2 |
| | FLOW-F | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 803.2 | 16.9 | 1871.2 |
| | FLOW-FE | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 803.2 | 17.1 | 1884.4 |
| | FLOW-A | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 803.2 | 44.5 | 2070.4 |
| | FLOW-MA | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 1237.5 | 0.3 | 2077.4 |
| | FLOW-S | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 1271.0 | 0.2 | 2091.3 |
| 30 | FLOW-B | 0 | 0 | 1 | 0 | 9 | 110 | 0.1 | 1139.9 | 1155.9 | 18271.4 |
| | FLOW-P1 | 0 | 0 | 0 | 0 | 8 | 112 | 0.2 | 1141.3 | 905.8 | 8661.9 |
| | FLOW-P2 | 0 | 0 | 0 | 0 | 6 | 114 | 0.1 | 1141.3 | 690.6 | 7116.6 |
| | FLOW-P3 | 0 | 0 | 0 | 0 | 5 | 115 | 0.1 | 1140.9 | 690.9 | 6408.7 |
| | FLOW-F | 0 | 0 | 0 | 0 | 5 | 115 | 0.1 | 1140.8 | 653.6 | 6258.7 |
| | FLOW-FE | 0 | 0 | 0 | 0 | 4 | 116 | 0.1 | 1140.6 | 638.8 | 6301.2 |
| | FLOW-A | 0 | 0 | 0 | 0 | 6 | 114 | 0.1 | 1140.9 | 635.2 | 6473.0 |
| | FLOW-MA | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 1901.8 | 1.0 | 6670.3 |
| | FLOW-S | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 1934.1 | 0.9 | 6625.9 |
| 50 | FLOW-B | 1 | 1 | 4 | 0 | 23 | 91 | 0.6 | 1728.1 | 2895.2 | 51474.8 |
| | FLOW-P1 | 0 | 0 | 3 | 0 | 16 | 101 | 0.4 | 1742.4 | 2006.5 | 21382.1 |
| | FLOW-P2 | 0 | 0 | 1 | 0 | 14 | 105 | 0.6 | 1753.8 | 1718.0 | 17825.8 |
| | FLOW-P3 | 0 | 0 | 0 | 0 | 16 | 104 | 0.6 | 1760.0 | 1676.9 | 15193.8 |
| | FLOW-F | 0 | 0 | 1 | 0 | 15 | 104 | 0.5 | 1752.8 | 1639.7 | 15033.7 |
| | FLOW-FE | 0 | 0 | 2 | 0 | 12 | 106 | 0.3 | 1746.5 | 1601.0 | 15050.5 |
| | FLOW-A | 0 | 0 | 0 | 1 | 17 | 102 | 0.5 | 1751.9 | 1879.9 | 15971.0 |
| | FLOW-MA | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 3131.5 | 2.9 | 16106.5 |
| | FLOW-S | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 3179.0 | 2.7 | 15288.9 |
| 100 | FLOW-B | 1 | 45 | 6 | 0 | 27 | 41 | 1.4 | 2939.1 | 6780.5 | 212914.1 |
| | FLOW-P1 | 0 | 8 | 14 | 0 | 48 | 50 | 1.7 | 3291.9 | 6054.2 | 71727.4 |
| | FLOW-P2 | 0 | 3 | 6 | 0 | 53 | 58 | 1.6 | 3385.3 | 5534.4 | 57764.0 |
| | FLOW-P3 | 1 | 3 | 2 | 0 | 56 | 58 | 1.6 | 3415.5 | 5484.2 | 49632.6 |
| | FLOW-F | 0 | 3 | 2 | 0 | 59 | 56 | 1.5 | 3420.7 | 5648.9 | 49346.6 |
| | FLOW-FE | 0 | 3 | 3 | 0 | 55 | 59 | 1.4 | 3410.1 | 5432.7 | 49079.0 |
| | FLOW-A | 2 | 3 | 3 | 0 | 56 | 56 | 1.3 | 3403.4 | 5601.8 | 51844.1 |
| | FLOW-MA | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 6207.9 | 10.9 | 50784.4 |
| | FLOW-S | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 6189.7 | 10.6 | 48418.3 |
| 200 | FLOW-B | 0 | 69 | 4 | 0 | 13 | 34 | 0.4 | 3325.1 | 7410.6 | 457143.5 |
| | FLOW-P1 | 0 | 39 | 9 | 0 | 31 | 41 | 1.6 | 3973.1 | 6856.6 | 133156.1 |
| | FLOW-P2 | 0 | 24 | 13 | 0 | 37 | 46 | 1.7 | 4218.3 | 6514.9 | 109582.3 |
| | FLOW-P3 | 0 | 18 | 17 | 0 | 41 | 44 | 1.7 | 4253.3 | 6455.4 | 93957.8 |
| | FLOW-F | 0 | 17 | 18 | 0 | 38 | 47 | 1.5 | 4264.8 | 6343.9 | 92999.5 |
| | FLOW-FE | 0 | 15 | 15 | 0 | 41 | 49 | 1.5 | 4342.4 | 6275.2 | 92228.6 |
| | FLOW-A | 0 | 20 | 16 | 1 | 36 | 47 | 1.4 | 4191.6 | 6372.2 | 102392.0 |
| | FLOW-MA | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 8849.6 | 28.5 | 102075.4 |
| | FLOW-S | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 8872.3 | 27.1 | 91624.3 |
| 500 | FLOW-B | 73 | 6 | 10 | 0 | 5 | 26 | 0.2 | 3171.8 | 8131.2 | 433322.5 |
| | FLOW-P1 | 0 | 84 | 2 | 0 | 7 | 27 | 0.2 | 3448.6 | 7879.0 | 316870.9 |
| | FLOW-P2 | 0 | 68 | 3 | 0 | 15 | 34 | 0.7 | 4423.6 | 7384.5 | 250480.3 |
| | FLOW-P3 | 0 | 67 | 2 | 0 | 18 | 33 | 0.7 | 4503.6 | 7440.8 | 243814.4 |
| | FLOW-F | 0 | 64 | 6 | 0 | 17 | 33 | 0.6 | 4488.7 | 7385.6 | 241183.3 |
| | FLOW-FE | 0 | 63 | 2 | 0 | 22 | 33 | 0.7 | 4706.8 | 7334.0 | 240455.4 |
| | FLOW-A | 0 | 67 | 6 | 0 | 14 | 33 | 0.4 | 4293.6 | 7452.2 | 279763.4 |
| | FLOW-MA | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 13381.6 | 146.0 | 278027.4 |
| | FLOW-S | 0 | 0 | 0 | 0 | 0 | 120 | 0.0 | 13774.0 | 174.6 | 241346.1 |
| 1000 | FLOW-B | 90 | 3 | 4 | 0 | 15 | 8 | 1.5 | 3677.5 | 9707.1 | 416302.4 |
| | FLOW-P1 | 19 | 61 | 13 | 0 | 11 | 16 | 0.7 | 3721.4 | 9104.5 | 468323.0 |
| | FLOW-P2 | 3 | 66 | 14 | 0 | 13 | 24 | 0.6 | 4657.2 | 8209.1 | 420043.5 |
| | FLOW-P3 | 3 | 65 | 15 | 0 | 12 | 25 | 0.6 | 4617.8 | 8146.5 | 418783.2 |
| | FLOW-F | 2 | 64 | 14 | 0 | 13 | 27 | 0.6 | 4834.0 | 8058.6 | 420225.3 |
| | FLOW-FE | 2 | 59 | 14 | 0 | 18 | 27 | 0.7 | 5277.2 | 7978.8 | 417890.5 |
| | FLOW-A | 3 | 68 | 8 | 0 | 15 | 26 | 0.8 | 4948.1 | 8062.7 | 509628.8 |
| | FLOW-MA | 15 | 0 | 0 | 0 | 0 | 105 | 0.0 | 16698.6 | 1630.9 | 463200.4 |
| | FLOW-S | 1 | 0 | 0 | 0 | 0 | 119 | 0.0 | 21265.4 | 673.8 | 422138.7 |

Other than that, the solving performance of all configurations is similar, between 110 and 115 instances can be solved to optimality, for the rest we get feasible solutions. The average gap is also very small. Notice how FLOW-B seems to produce better results based on the reported objective value than the other configurations. This is because the result for the instance that failed with NS is missing from the average objective value. Again, we can see significant run-time improvements by activating preprocessing and this time using improved pruning (FLOW-P2) gives an additional advantage. Notice how preprocessing is able to reduce the number of required variables to a third of the original value. FLOW-FE shows a distinct advantage when compared to FLOW-F in terms of required run-time and is even able to prove optimality for one more instance. For size 30, FLOW-A is the fastest solution method, the ability to add additional resources no longer seems to be a disadvantage. The last two configurations are still very fast and able to find valid solutions to all instances. Note however that the gap to the optimal solution values increases, the found solutions now being 60% more expensive than the optimal solutions.

By increasing the substrate size to 50 nodes, we can observe the first failures of FLOW-B due to the memory limit. None of the other methods have problematic memory consumption, but they start failing due to NS. Also, the number of instances where only a valid solution could be found increases. Still, all methods employing preprocessing can solve more than 100 instances to optimality. FLOW-A is again slower than the other methods and also keeps being slower for larger instance sizes. Also notice how the run-time advantage of methods using preprocessing starts to shrink, but the difference in variables is still huge.

At size 100 we have reached the point where only half of the instances can be solved to optimality by the optimization configurations using preprocessing (FLOW-P1 – FLOW-A). For the remaining instances we mostly find feasible solutions, but for some we fail earlier at LP or NS. The most common result for FLOW-B is to fail at LP, which means gaps and objective values cannot be compared meaningfully with the other algorithms.

By doubling the instance size again to 200, we get a significant number of failures for all optimization configurations, but still a surprising number of instances can be solved to optimality by the configurations using preprocessing. This is the last size for which we can compare the number of variables in a meaningful way, as FLOW-B starts failing at Mem for the larger sizes. FLOW-B almost requires five times more variables than FLOW-FE. For this size class we also have the second and last occurrence of a solution found by FLOW-A that requires additional resources.

For size 500, FLOW-B requires too much memory in most cases. All other optimization configurations predominately fail at LP. Observe that if a valid solution can be found, in most cases also its optimality can be proven or the remaining gap is very small.

Applying the different FLOW configurations to the last instance size, we can see that also FLOW-P1 starts to fail a significant number of times due to Mem. The configurations using more preprocessing fail predominantly at LP. Again, if we find a valid solution, we either achieve low gaps or can prove optimality, but this happens less often than for the previous size class. Also the configurations concentrating on just finding valid solutions start to fail, especially FLOW-MA, which needs too much memory. FLOW-S just fails once due to Mem and manages to find valid solutions for all other instances. Here we can clearly see the run-time and memory cost of not only finding valid solutions, but also being able to derive cheap recourse actions if the VNMP

Table 11.3: Main characteristics of different configurations of FLOW depending on the instance load.

| Load | Method | Mem | LP | NS | AR | Feas | Opt | Gap[%] | Obj | t[s] | Vars |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.10 | FLOW-B | 0 | 3 | 4 | 0 | 23 | 180 | 0.2 | 1866.3 | 1812.7 | 97877.5 |
| | FLOW-P1 | 0 | 0 | 3 | 0 | 16 | 191 | 0.1 | 1906.7 | 1237.4 | 24635.7 |
| | FLOW-P2 | 0 | 0 | 0 | 0 | 7 | 203 | 0.0 | 1937.8 | 428.8 | 16877.0 |
| | FLOW-P3 | 0 | 0 | 0 | 0 | 7 | 203 | 0.0 | 1938.0 | 440.0 | 16362.7 |
| | FLOW-F | 0 | 0 | 0 | 0 | 5 | 205 | 0.0 | 1937.8 | 378.5 | 16209.0 |
| | FLOW-FE | 0 | 0 | 0 | 0 | 5 | 205 | 0.0 | 1937.8 | 330.3 | 15320.2 |
| | FLOW-A | 0 | 0 | 0 | 0 | 6 | 204 | 0.0 | 1938.3 | 395.0 | 20833.0 |
| | FLOW-MA | 0 | 0 | 0 | 0 | 0 | 210 | 0.0 | 3504.0 | 11.2 | 23754.1 |
| | FLOW-S | 0 | 0 | 0 | 0 | 0 | 210 | 0.0 | 3508.9 | 10.3 | 19896.2 |
| 0.50 | FLOW-B | 44 | 25 | 13 | 0 | 21 | 107 | 0.2 | 1985.6 | 5213.3 | 201956.9 |
| | FLOW-P1 | 0 | 58 | 3 | 0 | 32 | 117 | 0.5 | 2348.0 | 4614.2 | 130523.2 |
| | FLOW-P2 | 0 | 31 | 9 | 0 | 39 | 131 | 0.5 | 2891.2 | 4135.5 | 104773.1 |
| | FLOW-P3 | 0 | 27 | 11 | 0 | 42 | 130 | 0.5 | 2924.8 | 4061.7 | 100501.8 |
| | FLOW-F | 0 | 26 | 11 | 0 | 43 | 130 | 0.4 | 2959.5 | 4065.7 | 99785.8 |
| | FLOW-FE | 0 | 22 | 6 | 0 | 49 | 133 | 0.5 | 3201.5 | 3955.1 | 99474.9 |
| | FLOW-A | 0 | 27 | 10 | 0 | 42 | 131 | 0.4 | 2979.1 | 4084.8 | 117538.6 |
| | FLOW-MA | 0 | 0 | 0 | 0 | 0 | 210 | 0.0 | 6697.3 | 45.6 | 115672.5 |
| | FLOW-S | 0 | 0 | 0 | 0 | 0 | 210 | 0.0 | 6888.1 | 44.8 | 98781.1 |
| 0.80 | FLOW-B | 60 | 45 | 3 | 0 | 23 | 79 | 0.7 | 1766.8 | 6442.6 | 201773.8 |
| | FLOW-P1 | 0 | 75 | 11 | 0 | 38 | 86 | 1.0 | 2306.7 | 6128.9 | 209788.7 |
| | FLOW-P2 | 0 | 66 | 7 | 0 | 44 | 93 | 1.1 | 2586.2 | 5973.0 | 168153.4 |
| | FLOW-P3 | 1 | 64 | 6 | 0 | 49 | 90 | 1.1 | 2623.4 | 5923.0 | 161912.1 |
| | FLOW-F | 0 | 61 | 8 | 0 | 49 | 92 | 1.1 | 2686.2 | 5927.3 | 160086.1 |
| | FLOW-FE | 0 | 61 | 6 | 0 | 48 | 95 | 0.9 | 2734.8 | 5831.7 | 159641.8 |
| | FLOW-A | 2 | 63 | 8 | 0 | 48 | 89 | 1.0 | 2603.3 | 6004.9 | 188727.4 |
| | FLOW-MA | 0 | 0 | 0 | 0 | 0 | 210 | 0.0 | 8959.1 | 111.9 | 185858.2 |
| | FLOW-S | 0 | 0 | 0 | 0 | 0 | 210 | 0.0 | 9879.9 | 113.1 | 158224.1 |
| 1.00 | FLOW-B | 61 | 51 | 9 | 0 | 25 | 64 | 1.0 | 1739.9 | 7207.3 | 253519.9 |
| | FLOW-P1 | 19 | 59 | 24 | 0 | 35 | 73 | 1.4 | 2157.8 | 6782.6 | 194607.1 |
| | FLOW-P2 | 3 | 64 | 21 | 0 | 48 | 74 | 1.8 | 2493.8 | 6654.7 | 201272.1 |
| | FLOW-P3 | 3 | 62 | 19 | 0 | 50 | 76 | 1.8 | 2549.5 | 6674.9 | 192672.3 |
| | FLOW-F | 2 | 61 | 22 | 0 | 50 | 75 | 1.5 | 2542.8 | 6626.9 | 194292.1 |
| | FLOW-FE | 2 | 57 | 24 | 0 | 50 | 77 | 1.5 | 2634.3 | 6612.9 | 193650.0 |
| | FLOW-A | 3 | 68 | 15 | 2 | 48 | 74 | 1.3 | 2430.0 | 6686.0 | 223339.1 |
| | FLOW-MA | 15 | 0 | 0 | 0 | 0 | 195 | 0.0 | 9717.4 | 871.7 | 179565.3 |
| | FLOW-S | 1 | 0 | 0 | 0 | 0 | 209 | 0.0 | 11956.2 | 340.3 | 194892.7 |

instance does not have a valid solution. For the smaller sizes, there was no clear difference between FLOW-MA and FLOW-S.

We have discussed the characteristics of different FLOW configurations based on the instance size. Table 11.3 shows the same characteristics, but now based on the instance load. For the lowest load of 0.1, finding at least feasible solutions is not a problem, just FLOW-B fails at LP or NS for some instances. All other configurations manage to find optimal solutions for most instances. Note that for low loads the run-time requirements are very different for the compared

configurations, higher loads will cause the run-time requirements to be more similar. The same holds true for the number of variables. Also observe that the solutions found by FLOW-F and FLOW-FE are the same, but FLOW-FE is faster. Both configurations are better than FLOW-P3. By increasing the load to 0.5, we see that FLOW-B already starts to fail due to Mem, all other configurations start failing due to LP. Most instances can still be solved to optimality. Going to load 0.8 increases the number of failures due to LP further, we also start seeing failures due to Mem for configurations other than FLOW-B. The average gap reaches 1%. The main difference when using VNMP instances with full load is that for some instances the different configurations fail to find valid solutions more often. It can also be seen that the two instances for which FLOW-A produces a solution which requires additional resources are those with highest load. There is not a lot of observable difference between the characteristics of the configurations FLOW-P2 to FLOW-A, only FLOW-P1 and FLOW-B are worse.
Based on these results, one thing is clear: if we want to tackle instances with more than 50 substrate nodes and with more than 50% load with FLOW, then preprocessing in some form is essential. For the largest instance sizes and highest loads, using more advanced preprocessing techniques becomes important. Valid solutions can be found to nearly all instances in surprisingly short time. Especially for large instances with high loads, solving the LP relaxation in the root node is very time consuming. The differences between the configurations from FLOW-P1 to FLOW-A are not very visible and require further analysis. Preprocessing has a very pronounced effect on the number of variables, which also translates to shorter run-times, albeit to a lesser extent. This is not surprising since in some sense preprocessing just removes unnecessary ballast but does not make the core problem easier to solve.

### 11.4.2 Comparison of FLOW Configurations

In the previous section, we have analyzed the properties of different configurations of FLOW. It could be seen that adding preprocessing improves the performance, but a more detailed analysis (for instance if fixing nodes reduces run-times in a statistically significant way or if going from FLOW-P2 to FLOW-P3 is beneficial) could not be presented. This is the main goal for this section. We will present a comparison of the configurations FLOW-P1 to FLOW-A and determine which one actually performs best. We excluded FLOW-B, because the results presented in the previous section showed it to be very clearly worse than the alternatives. FLOW-MA and FLOW-S are not considered since they do not perform optimization. To be able to perform a meaningful comparison, we restrict ourselves to instances for which all compared configurations at least managed to find a valid solution that does not require additional resources.
Table 11.4 shows the comparison of the different FLOW configurations based on instance size. The values labeled # Inst. give the count of instances used as basis for the other presented properties. For instance the data for size 20 is based on all 120 instances, but for size 1000 the different configurations produced comparable results only for 27 instances. The other properties have the same meaning as before. The presented values are geometric means, except for the gap values, which are arithmetic means (since values of zero can occur).
Based on the presented results, we can see that FLOW-P1 is outclassed for all but the smallest instance sizes in terms of the achieved solution and always significantly slower than the best configurations. For the largest instance size, FLOW-P1 is nearly ten times slower than the best

Table 11.4: Comparison of FLOW configurations utilizing preprocessing, depending on the instance size.

| | Size | FLOW-P1 | FLOW-P2 | FLOW-P3 | FLOW-F | FLOW-FE | FLOW-A |
|---|---|---|---|---|---|---|---|
| **# Inst.** | 20 | 120 | 120 | 120 | 120 | 120 | 120 |
| | 30 | 120 | 120 | 120 | 120 | 120 | 120 |
| | 50 | 117 | 117 | 117 | 117 | 117 | 117 |
| | 100 | 97 | 97 | 97 | 97 | 97 | 97 |
| | 200 | 71 | 71 | 71 | 71 | 71 | 71 |
| | 500 | 34 | 34 | 34 | 34 | 34 | 34 |
| | 1000 | 27 | 27 | 27 | 27 | 27 | 27 |
| **Gap[%]** | 20 | 0.00 = | 0.00 = | 0.00 = | 0.00 = | 0.00 = | 0.00 = |
| | 30 | 0.16 > | 0.11 = | 0.10 = | 0.08 = | 0.08 = | 0.10 = |
| | 50 | 0.41 > | 0.35 > | 0.32 = | 0.26 = | 0.26 = | 0.31 = |
| | 100 | 1.58 > | 1.10 > | 0.97 > | 0.96 = | 0.84 = | 0.94 > |
| | 200 | 1.51 > | 0.93 > | 0.80 > | 0.73 = | 0.63 = | 0.72 = |
| | 500 | 0.23 > | 0.06 = | 0.05 = | 0.04 = | 0.06 = | 0.06 = |
| | 1000 | 0.73 > | 0.10 = | 0.11 = | 0.08 = | 0.07 = | 0.12 = |
| **Obj** | 20 | 758.9 = | 758.9 = | 758.9 = | 758.9 = | 758.9 = | 758.9 = |
| | 30 | 1086.2 > | 1086.2 = | 1085.9 = | 1085.8 = | 1085.7 = | 1085.9 = |
| | 50 | 1659.2 > | 1658.6 > | 1658.2 = | 1657.7 = | 1657.6 = | 1657.8 = |
| | 100 | 3104.7 > | 3095.2 > | 3092.9 = | 3093.5 = | 3091.4 = | 3092.9 = |
| | 200 | 3673.6 > | 3660.6 > | 3656.2 > | 3655.7 = | 3653.8 = | 3654.4 = |
| | 500 | 3326.3 = | 3325.4 = | 3325.4 = | 3325.1 = | 3325.5 = | 3325.1 = |
| | 1000 | 3677.2 > | 3665.4 = | 3665.7 = | 3665.8 = | 3665.5 = | 3666.2 = |
| **t[s]** | 20 | 1.0 > | 0.8 > | 0.8 = | 0.8 = | 0.7 = | 0.8 > |
| | 30 | 7.8 > | 5.6 > | 5.2 > | 4.9 = | 4.8 = | 4.7 = |
| | 50 | 41.1 > | 27.2 > | 24.3 = | 22.0 = | 22.0 = | 25.0 = |
| | 100 | 437.5 > | 283.4 > | 275.2 > | 284.4 > | 250.7 > | 237.3 = |
| | 200 | 511.2 > | 270.4 > | 247.3 > | 225.5 = | 215.9 = | 225.3 = |
| | 500 | 417.0 > | 91.8 > | 105.9 > | 90.3 > | 81.9 = | 84.2 = |
| | 1000 | 3373.9 > | 507.1 > | 469.5 > | 404.9 = | 352.9 = | 399.8 > |
| **Vars** | 20 | 1350.5 > | 1240.7 > | 1213.0 > | 1152.6 = | 1160.2 = | 1230.4 > |
| | 30 | 4478.5 > | 3669.4 > | 3455.1 > | 3464.5 > | 3366.5 = | 3538.5 > |
| | 50 | 11442.5 > | 9838.5 > | 8480.6 > | 8275.8 > | 8432.2 = | 9446.0 > |
| | 100 | 32832.4 > | 27831.2 > | 24132.6 > | 23648.2 > | 23223.7 = | 25757.1 > |
| | 200 | 46666.0 > | 36512.6 > | 30493.7 > | 30487.8 > | 28645.3 = | 36273.1 > |
| | 500 | 44079.5 > | 28478.0 > | 27718.1 > | 22596.9 = | 25789.7 = | 32490.7 > |
| | 1000 | 87962.1 > | 49807.1 > | 50308.2 > | 50537.9 > | 44325.1 = | 69993.1 > |

configuration. Adding additional preprocessing capabilities by using pruning by APSP improves the situation a bit. For instance sizes 20, 30, 500 and 1000 FLOW-P2 achieves solutions not significantly different from the best, both in terms of gap and objective. For the instances of medium size, PathEnumeration manages better pruning and FLOW-P2 cannot keep up. For the largest instances, the distance between APSP and PathEnumeration shrinks, due to the fallback behaviour of PathEnumeration, and FLOW-P2 manages to create comparable results. It also speeds up the solving process itself, giving a factor of 4 to 6 improvement over FLOW-P1. By using PathEnumeration as pruning method (FLOW-P3), it is possible to achieve the best known objectives for all but one size class and the best gaps except for medium sized instances. FLOW-P3 also starts getting competitive in terms of run-time, but is still more than 30% slower than the best for the largest instances.

By switching on fixing (FLOW-F), we are now able to achieve the best possible results in terms of gap and objective function. The difference to FLOW-P3 shows that utilizing the information about nodes that have to be used results in a meaningful improvement and is not only a theoretical advantage. It is moderate in terms of gap and objective value, but very noticeable in terms of required run-time. To achieve the very best performance, we need to link the use of substrate nodes to the incoming flow via equalities (FLOW-FE). This configuration achieves the best values for gap, objective, required run-time and variables in nearly all cases.

The case of FLOW-A is interesting to analyze. On one hand, the model is larger in principle, because we have to add variables that allow buying additional resources. These additional variables even increase the number of variables in the simplified model in a disproportionate way, i.e., the difference in variables between FLOW-FE and FLOW-A is larger than the number of added variables. Therefore, we could expect worse performance. On the other hand, allowing to buy additional resources makes finding an integer feasible solution easier, which potentially speeds up the solving process. Based on the results shown in Table 11.4, it seems that these influences balance each other. FLOW-A requires far more variables, but is still competitive with respect to solving time, especially for instances of medium size.

Table 11.5 shows the same data, but this time depending on instance load. Regarding the quality of the solutions, we get basically the same results as previously. One difference to note is that FLOW-P1 is not even competitive for the lowest loads, both in terms of average gap and objective. By increasing the level of preprocessing and utilizing fixing information, we get the best solutions. With respect to the time required to achieve a solution, we can see that for loads 0.1 and 0.5, FLOW-A is the fastest configuration. For higher loads, FLOW-FE is fastest.

### 11.4.3 Starting with a Valid Solution

The aim of this section is to analyze the influence of using an initial solution created by one of the heuristics presented in the previous chapters on FLOW. As base configurations we use FLOW-A and FLOW-MA, to solve VNMP-O and VNMP-S respectively. Note that we chose FLOW-A instead of FLOW-FE and FLOW-MA instead of FLOW-S because they can use an initial solution which requires additional resources. Such solutions are likely to be found by heuristics, especially for the larger instance sizes and could not be used otherwise (e.g., when using FLOW-FE), so all the effort expended on creating them would have been wasted.

Table 11.5: Comparison of FLOW configurations utilizing preprocessing, depending on the instance load.

| | Size | FLOW-P1 | FLOW-P2 | FLOW-P3 | FLOW-F | FLOW-FE | FLOW-A |
|---|---|---|---|---|---|---|---|
| **# Inst.** | 0.10 | 207 | 207 | 207 | 207 | 207 | 207 |
| | 0.50 | 149 | 149 | 149 | 149 | 149 | 149 |
| | 0.80 | 123 | 123 | 123 | 123 | 123 | 123 |
| | 1.00 | 107 | 107 | 107 | 107 | 107 | 107 |
| **Gap[%]** | 0.10 | 0.12 > | 0.01 = | 0.02 = | 0.01 = | 0.01 = | 0.02 > |
| | 0.50 | 0.50 > | 0.28 > | 0.26 > | 0.23 = | 0.22 = | 0.21 = |
| | 0.80 | 0.98 > | 0.73 > | 0.58 > | 0.54 = | 0.47 = | 0.55 = |
| | 1.00 | 1.26 > | 0.91 > | 0.86 = | 0.79 = | 0.73 = | 0.86 > |
| **Obj** | 0.10 | 1476.4 > | 1475.7 = | 1475.7 = | 1475.7 = | 1475.7 = | 1475.7 = |
| | 0.50 | 1868.9 > | 1866.6 > | 1866.5 > | 1866.1 = | 1866.0 = | 1866.0 = |
| | 0.80 | 1863.5 > | 1860.7 > | 1858.3 = | 1858.2 = | 1857.5 = | 1858.0 = |
| | 1.00 | 1818.2 > | 1814.3 > | 1813.5 = | 1813.6 = | 1812.4 = | 1813.6 > |
| **t[s]** | 0.10 | 6.6 > | 2.8 > | 2.7 > | 2.5 > | 2.5 > | 2.2 = |
| | 0.50 | 43.3 > | 26.6 > | 24.4 > | 23.6 > | 21.8 > | 21.5 = |
| | 0.80 | 127.4 > | 103.1 > | 94.4 > | 91.2 > | 84.1 = | 96.3 = |
| | 1.00 | 244.5 > | 189.6 > | 186.6 = | 174.9 = | 168.4 = | 228.6 > |
| **Vars** | 0.10 | 6039.1 > | 4596.8 > | 4218.4 > | 4044.8 > | 3965.8 = | 4891.2 > |
| | 0.50 | 12578.3 > | 10699.3 > | 9623.5 > | 9380.9 > | 9410.1 = | 10205.0 > |
| | 0.80 | 14696.6 > | 12567.9 > | 11381.9 > | 11101.7 > | 10958.3 = | 11741.8 > |
| | 1.00 | 13974.3 > | 12081.4 > | 11126.3 > | 10930.4 > | 10881.9 = | 11872.2 > |

Table 11.6: Main solving characteristics of FLOW-A (FA) and FLOW-MA (FMA) when an initial solution is created by CH-O (CH), LS-O (LS), or VNS-O (VNS).

| Method | Mem | LP | NS | AR | Feas | Opt |
|---|---|---|---|---|---|---|
| FA | 3 | 165 | 36 | 3 | 138 | 495 |
| FA-CH | 29 | 0 | 0 | 116 | 198 | 497 |
| FA-LS | 28 | 0 | 0 | 74 | 239 | 499 |
| FA-VNS | 28 | 0 | 0 | 58 | 255 | 499 |
| FMA | 15 | 0 | 0 | 0 | 0 | 825 |
| FMA-CH | 28 | 0 | 0 | 0 | 0 | 812 |
| FMA-LS | 27 | 0 | 0 | 0 | 0 | 813 |
| FMA-VNS | 27 | 0 | 0 | 0 | 0 | 813 |

As heuristics to create initial solutions we use CH-O, LS-O, and VNS-O with a time-limit of 500 seconds. The time required to create the initial solution of course counts towards the total time-limit of 10000 seconds.

The solving characteristics of the different configurations of FLOW-A and FLOW-MA are shown in Table 11.6. We use the same definitions as in Section 11.4.1. When solving VNMP-O with FLOW-A, using a heuristic to create an initial solution has a very pronounced influence on the solving characteristics. While the basic FLOW-A configuration (FA) fails to solve the LP relaxation in the root node 165 times or fails to find an integer feasible solution 36 times (out of 840 instances), it never fails due to those reasons when an initial solution is supplied. Interestingly, an initial solution increases the number of instances that fail due to Mem. This is most likely caused by FLOW-A being able to solve the LP relaxation when an initial solution is supplied, but later running out of memory during Branch & Bound. For most instances that failed due to LP, a valid solution, or at least a solution requiring additional resources can be found. Supplying an initial solution has barely any influence on the number of instances that can be solved to optimality. Based on the number of instances for which a valid solution could be found, VNS-O is the best heuristic to use to create the initial solution. When solving VNMP-S with FLOW-MA (FMA), the only difference caused by supplying an initial solution is that more instances fail due to Mem.

Table 11.7 shows a comparison of the different FLOW-A configurations. As in the previous section, we restrict the comparison to those instances for which all configurations where able to find at least a valid solution. The number of instances used for comparison is labeled by # Inst. It can be seen that beginning with size 200 a significant fraction of instances is rejected. The influence of an initial solution on the achieved gap and objective values is negligible. Only with respect to the required run-time is it advantageous to supply an initial solution, which has to be created by CH-O. Using CH-O has the most benefit for the largest instance sizes. The run-time overhead of LS-O is too high to make it worthwhile. Note that we use VNS-O exactly in the configuration as introduced in Chapter 8. That means that the only termination criterion is the elapsed time and as a consequence FA-VNS requires at least 500 seconds to solve an instance. However, even if we subtract 500 seconds from the reported run-times, it still is far from competitive. Using VNS-O does not show any advantage with respect to gap or objective value, so an evaluation of more practical termination criteria does not seem warranted. Also with respect to the required Branch & Bound nodes (# Nodes) VNS-O is only as good as the other heuristics are.

The same data, but now according to instance load is presented in Table 11.8. Again we can observe that the presence of an initial solution only has a significant influence on the required run-time and CH-O seems to be the right choice. A solution created by LS-O is only beneficial for instances of the highest load.

The performance of the different FLOW-MA configurations when solving VNMP-S is shown in Table 11.9. Note that we do not present the average objective and gap values in this case, because we have already shown that the different configurations of FLOW-MA either find a valid solution or fail due to the memory limit. For solving VNMP-S it is very clear that using an initial solution created by CH is an advantage in terms of required run-time. In contrast to VNMP-O, better initial solutions consistently lead to fewer required nodes in the Branch & Bound tree, with the

Table 11.7: Comparison of FLOW-A (FA) configurations depending on the instance size.

| | Size | FA | FA-CH | FA-LS | FA-VNS |
|---|---|---|---|---|---|
| **# Inst.** | 20 | 120 | 120 | 120 | 120 |
| | 30 | 120 | 120 | 120 | 120 |
| | 50 | 120 | 120 | 120 | 120 |
| | 100 | 113 | 113 | 113 | 113 |
| | 200 | 79 | 79 | 79 | 79 |
| | 500 | 46 | 46 | 46 | 46 |
| | 1000 | 35 | 35 | 35 | 35 |
| **Gap[%]** | 20 | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 30 | 0.1 = | 0.1 = | 0.1 = | 0.1 = |
| | 50 | 0.5 = | 0.5 = | 0.5 = | 0.5 = |
| | 100 | 1.4 = | 1.3 = | 1.4 = | 1.4 = |
| | 200 | 1.4 > | 1.2 = | 1.2 = | 1.2 = |
| | 500 | 0.4 = | 0.4 = | 0.4 = | 0.3 = |
| | 1000 | 0.6 > | 0.4 = | 0.4 = | 0.4 = |
| **Obj** | 20 | 803.2 = | 803.2 = | 803.2 = | 803.2 = |
| | 30 | 1140.8 = | 1141.3 = | 1140.7 = | 1140.9 = |
| | 50 | 1758.0 = | 1758.1 = | 1757.9 = | 1758.2 = |
| | 100 | 3398.2 = | 3397.1 = | 3398.2 = | 3395.6 = |
| | 200 | 4140.7 > | 4134.4 = | 4134.6 = | 4134.6 = |
| | 500 | 4248.8 = | 4249.6 = | 4250.2 = | 4248.5 = |
| | 1000 | 4353.7 = | 4346.1 = | 4347.4 = | 4346.4 = |
| **t[s]** | 20 | 0.9 = | 0.8 = | 1.0 > | 516.5 > |
| | 30 | 4.9 = | 4.9 = | 5.7 > | 650.2 > |
| | 50 | 30.3 > | 28.7 = | 32.4 > | 963.9 > |
| | 100 | 421.7 > | 410.5 = | 488.3 > | 2742.7 > |
| | 200 | 341.6 > | 341.3 = | 400.8 > | 2336.1 > |
| | 500 | 306.6 > | 277.2 = | 330.4 > | 1499.3 > |
| | 1000 | 730.1 > | 645.5 = | 674.4 = | 1545.7 > |
| **# Nodes** | 20 | 921.7 = | 583.2 = | 500.4 = | 698.5 = |
| | 30 | 1452.4 = | 1532.6 = | 1993.7 = | 2047.2 = |
| | 50 | 2756.6 > | 2873.8 > | 2219.0 = | 2245.4 = |
| | 100 | 5657.1 = | 4409.3 = | 4840.8 = | 5728.0 = |
| | 200 | 1528.4 = | 1513.4 = | 1599.3 = | 1574.5 = |
| | 500 | 558.0 = | 503.4 = | 539.8 = | 625.7 = |
| | 1000 | 715.1 = | 656.3 = | 732.7 = | 636.7 = |

Table 11.8: Comparison of FLOW-A (FA) configurations depending on the instance load.

| | Load | FA | FA-CH | FA-LS | FA-VNS |
|---|---|---|---|---|---|
| **# Inst.** | 0.10 | 210 | 210 | 210 | 210 |
| | 0.50 | 165 | 165 | 165 | 165 |
| | 0.80 | 135 | 135 | 135 | 135 |
| | 1.00 | 123 | 123 | 123 | 123 |
| **Gap[%]** | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.4 = | 0.3 = | 0.3 > | 0.3 = |
| | 0.80 | 1.0 > | 0.9 = | 0.8 = | 0.9 = |
| | 1.00 | 1.4 = | 1.4 = | 1.4 = | 1.4 = |
| **Obj** | 0.10 | 1938.3 = | 1937.9 = | 1937.9 = | 1937.9 = |
| | 0.50 | 2749.6 > | 2748.4 = | 2749.4 = | 2748.2 = |
| | 0.80 | 2528.9 > | 2525.5 = | 2524.9 = | 2525.0 = |
| | 1.00 | 2447.6 = | 2447.2 = | 2447.6 = | 2446.3 = |
| **t[s]** | 0.10 | 2.5 > | 2.3 = | 3.1 > | 597.0 > |
| | 0.50 | 43.5 = | 43.2 = | 50.3 > | 1206.5 > |
| | 0.80 | 155.6 = | 147.9 = | 154.2 = | 1669.0 > |
| | 1.00 | 357.0 > | 360.1 = | 352.2 = | 2041.1 > |
| **# Nodes** | 0.10 | 170.9 > | 158.4 = | 177.4 = | 162.9 = |
| | 0.50 | 1280.1 = | 937.4 = | 1365.4 = | 1632.8 = |
| | 0.80 | 4501.3 = | 3847.3 = | 3597.9 = | 3937.8 = |
| | 1.00 | 4647.0 > | 4515.3 = | 4399.6 = | 4767.5 = |

fewest nodes required by FMA-VNS. This method might even be competitive with respect to run-time with a better termination criterion. The data presented in Table 11.10 shows the same tendencies. When considering run-time, FMA-CH performs best. The higher the instance load, the more important good initial solutions are, and again FMA-VNS performs best. It is notable that solving VNMP-S requires barely any branching.

In conclusion, we have shown that using an initial solution (preferably one created by CH-O) can reduce the time required to solve an instance. However, it does not increase the number of instances that can be solved to optimality or decrease the achieved gap. The influence of the employed heuristic is much more pronounced when solving VNMP-S. We will denote FA-CH by ILP-O and FLOW-S by ILP-S.

### 11.4.4 Feasibility of PATH

In this section, we are mainly concerned with PATH and whether this formulation is substantially better than FLOW. Using PATH, we only implemented a Column Generation approach, instead of full Branch & Price. Therefore, we will compare lower bounds on the objective instead of found feasible solutions. PATH should be able to achieve significantly better (i.e., higher) lower bounds than FLOW. For FLOW, we only showed the influence of utilizing knowledge about

Table 11.9: Comparison of FLOW-MA (FMA) configurations depending on the instance size.

| | Size | FMA | FMA-CH | FMA-LS | FMA-VNS |
|---|---|---|---|---|---|
| **# Inst.** | 20 | 120 | 120 | 120 | 120 |
| | 30 | 120 | 120 | 120 | 120 |
| | 50 | 120 | 120 | 120 | 120 |
| | 100 | 120 | 120 | 120 | 120 |
| | 200 | 120 | 120 | 120 | 120 |
| | 500 | 120 | 120 | 120 | 120 |
| | 1000 | 92 | 92 | 92 | 92 |
| **t[s]** | 20 | 0.2 > | 0.2 = | 0.3 > | 500.2 > |
| | 30 | 0.8 > | 0.6 = | 0.9 > | 500.4 > |
| | 50 | 2.5 > | 1.8 = | 2.4 > | 501.1 > |
| | 100 | 9.1 > | 7.3 = | 15.1 > | 504.6 > |
| | 200 | 23.9 > | 19.5 = | 52.1 > | 510.3 > |
| | 500 | 119.0 > | 93.9 = | 310.8 > | 572.4 > |
| | 1000 | 244.0 > | 189.4 = | 502.5 > | 652.9 > |
| **# Nodes** | 20 | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 30 | 0.1 = | 0.0 = | 0.0 = | 0.0 = |
| | 50 | 0.2 = | 0.0 = | 0.0 = | 0.0 = |
| | 100 | 0.1 = | 0.0 = | 0.0 = | 0.0 = |
| | 200 | 4.2 > | 0.6 = | 0.0 = | 0.0 = |
| | 500 | 46.4 > | 1.7 = | 2.4 = | 0.2 = |
| | 1000 | 98.5 > | 14.4 > | 12.8 = | 3.7 = |

Table 11.10: Comparison of FLOW-MA (FMA) configurations depending on the instance load.

| | Load | FMA | FMA-CH | FMA-LS | FMA-VNS |
|---|---|---|---|---|---|
| **# Inst.** | 0.10 | 210 | 210 | 210 | 210 |
| | 0.50 | 210 | 210 | 210 | 210 |
| | 0.80 | 209 | 209 | 209 | 209 |
| | 1.00 | 183 | 183 | 183 | 183 |
| **t[s]** | 0.10 | 9.7 > | 8.9 = | 16.2 > | 508.5 > |
| | 0.50 | 37.8 > | 32.3 = | 116.4 > | 528.9 > |
| | 0.80 | 86.1 > | 67.9 = | 177.7 > | 549.1 > |
| | 1.00 | 71.8 > | 51.3 = | 147.7 > | 536.3 > |
| **# Nodes** | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 1.6 > | 0.2 = | 0.0 = | 0.0 = |
| | 0.80 | 34.3 > | 5.7 > | 0.0 = | 0.7 = |
| | 1.00 | 41.9 > | 2.1 = | 8.0 > | 1.3 = |

Table 11.11: Number of instances for which all configurations created a result (# Inst.), average lower bounds (LB.) and average required run-time (t[s]) for FLOW and PATH configurations.

| | Size | PATH | F-P3N | F-FEN | F-P3 | F-FE |
|---|---|---|---|---|---|---|
| **# Inst.** | 20 | 120 | 120 | 120 | 120 | 120 |
| | 30 | 120 | 120 | 120 | 120 | 120 |
| | 50 | 116 | 116 | 116 | 116 | 116 |
| | 100 | 45 | 45 | 45 | 45 | 45 |
| | 200 | 31 | 31 | 31 | 31 | 31 |
| | 500 | 28 | 28 | 28 | 28 | 28 |
| | 1000 | 13 | 13 | 13 | 13 | 13 |
| **LB.** | 20 | 737.6 < | 737.0 < | 737.1 < | 785.5 = | 785.0 = |
| | 30 | 1073.8 < | 1072.4 < | 1072.4 < | 1117.8 = | 1118.5 = |
| | 50 | 1653.1 < | 1650.8 < | 1650.8 < | 1711.2 = | 1712.3 = |
| | 100 | 2391.8 < | 2386.3 < | 2386.4 < | 2440.4 = | 2440.2 = |
| | 200 | 2534.3 < | 2523.4 < | 2523.5 < | 2566.0 = | 2566.2 = |
| | 500 | 3082.9 < | 3071.4 < | 3072.2 < | 3110.6 = | 3111.6 = |
| | 1000 | 3996.8 < | 3990.8 < | 3992.8 < | 4052.8 = | 4053.2 = |
| **t[s]** | 20 | 22.7 > | 1.6 > | 0.9 = | 1.3 > | 1.2 > |
| | 30 | 243.5 > | 12.3 > | 7.5 = | 15.8 > | 12.9 > |
| | 50 | 1476.0 > | 34.5 > | 20.8 = | 47.0 > | 45.2 > |
| | 100 | 1648.2 > | 24.2 > | 6.6 = | 15.6 > | 13.3 > |
| | 200 | 884.3 > | 24.8 > | 5.4 = | 8.6 > | 6.6 > |
| | 500 | 1557.5 > | 79.5 > | 31.7 = | 46.1 > | 38.7 > |
| | 1000 | 4321.2 > | 324.2 > | 62.9 = | 96.6 > | 64.5 = |

fixed nodes on the final result and not on the achieved lower bound in the root node. We will also consider this aspect in this section.
The following PATH and FLOW configurations are going to be compared:

**PATH** The PATH model. The initial set of paths is derived from the solution found by CH-O.

**F-P3** The FLOW-P3 configuration, all cuts CPLEX uses in its default setting are used.

**F-P3N** The same as F-P3, but all automatic cuts of CPLEX are deactivated.

**F-FE** The FLOW-FE configuration making use of all cuts. This configuration utilizes information about fixed nodes or arcs, which F-P3 does not.

**F-FEN** The same as F-FE, but all cuts are deactivated.

Note that in addition to the influence of fixing nodes and arcs, we also compare the performance of the FLOW models with and without automatic cuts. For a fair comparison of model strength, PATH has to be compared to FLOW without cuts, and is expected to achieve better bounds. For practical purposes, it should also create better bounds than FLOW with cuts to be considered useful.

Table 11.11 shows a comparison of the different configurations. We only consider instances for which all methods were able to derive a valid lower bound. It can be seen that beginning with size 100, only for a small fraction of instances a bound could be obtained. This low number is due to PATH, the FLOW based configurations were far more successful. For size 1000, PATH fails mostly because of its memory requirements, for smaller sizes too many pricing iterations are required.
When comparing the achieved lower bounds, it can be observed that PATH indeed creates better bounds than both FLOW configurations without automatic cuts, and the difference is statistically significant (not shown in the table). However, the difference is small, especially compared to the bound increase achieved by using the automatic cuts. As for the difference in bounds when fixing nodes and arcs, we see that fixing results in slightly higher bounds, but the difference is not significant. This is surprising since we have already shown in Section 11.4.2 that using information about fixed nodes or arcs has a significant positive influence on the final result.
When considering the required run-time, we see that PATH is not competitive at all. This alone however should not be discouraging, since it is well known [174] that Column Generation requires careful tuning to achieve lower run-times, which we did not perform in this case. Automatic cuts and fixing nodes have an interesting influence on the required run-time. Without the overhead of the automatic cuts, fixing nodes allows solving the LP much faster than without, especially for larger instances. With the automatic cuts, fixing is still faster, but only barely. In addition, activating the cuts decreases the required run-time when not fixing nodes, but increases it when fixing. We do not show the results depending on instance load since they show essentially the same.

## 11.5 Conclusion

In this chapter, we could show that the preprocessing techniques as presented in Chapter 9 are essential for solving the VNMP with FLOW. The largest improvement is achieved by activating preprocessing. Using more powerful preprocessing techniques gives another boost to performance. The achieved objective values are usually very similar, the influence of preprocessing mostly concerns the required run-time. In total, FLOW-FE performed best. This configuration uses FLOW with the highest levels of preprocessing, utilizing also information about fixed nodes and links incoming flow and node usage with equality. FLOW-A, which allows buying additional resources in the substrate, could have been better than FLOW-FE since integer feasible solutions can be found more easily, but this advantage did not materialize. Nevertheless, FLOW-A has a performance very similar to FLOW-FE, and for instances of medium size or low loads offers a small advantage with respect to run-time. The run-time can be further improved by starting from a solution created by CH-O.
The difference between methods for solving VNMP-S and VNMP-O is huge, both in number of solved instances and required run-time. FLOW-S fails only to produce a valid solution for a single instance of the largest size due to memory reasons, which also exceeds the capabilities of the best heuristics presented in the previous chapters. The FLOW variants solving VNMP-O fail earlier and more often. Additionally, there is a run-time difference of a factor between 10 and 500.

Concerning PATH, we have shown that it is not a promising approach for solving the VNMP. While it is able to derive better bounds than FLOW without automatic cuts used by CPLEX, it does not even come close to the bounds achieved by using automatic cuts. In addition, implementing a full Branch & Price scheme based on PATH would require significant effort, since this is not supported by CPLEX and additional work would be required to make it competitive with respect to run-time. The performance of PATH depends critically on the delays, since this constraint is hidden within the pricing problem. In preliminary work with another instance set, where most virtual arcs just used a large value for the allowed delay if it was not meant to be constraining, PATH achieved even worse results. It stands to reason that for instances where delays are more critical, PATH might offer a useful advantage over FLOW.

## 11.6 Future Work

The incorporation of knowledge about fixed nodes or arcs could be refined. We have shown that the information about fixed nodes can strengthen the model, because it helps forbidding fractional flows that would exceed the delay limit if they were integer. However, this is only true if the fixed node is not an articulation point in the shadow of the substrate. If it is an articulation point, then there is by definition no other way than to use this node, regardless of delay limits. The same holds true for fixed arcs, if they are bridges in the shadow of the substrate. Therefore, the corresponding constraints could be omitted in those cases.
During the definition of FLOW-A, we have forbidden the possibility of selling resources that we do not use. However, there are some applications where this would be an interesting possibility. For instance, if we own contended resources that we do not need to use ourselves, it might be profitable to rent them out. This of course raises the question of how to determine which resources to keep so that future virtual networks can still be accommodated.
If a VNMP instance has no valid solution, we only considered changes to the substrate network. It might beneficial to study the possibility of changing the virtual network requests. For instance, we could allocate less bandwidth to a virtual arc than it requires, which would incur a penalty. Depending on the costs for adding additional resources, this approach might make more sense. For full flexibility, one could consider both options at the same time. Some care would be required to keep the model linear.
The presented ILP models (and the VNMP in general) use a very simple delay model. One possible enhancement would be to assume the delay on a substrate connection to be normally distributed instead of constant. In addition to the maximum allowed delay, a virtual arc also specifies the probability of exceeding this delay. Since the means and variances of normal distributions are additive (assuming independent distributions), it is possible to modify FLOW by adding another variable type that tracks the sum of the variances for each virtual arc. The only challenge is that to check whether we exceed the delay bounds, we need the standard deviation, which is the square root of the variance. To calculate this, linear approximations of the square root function have to be employed, as presented for instance in [136].
Based on the presented results of FLOW-S and previous heuristics, it might make sense to combine both, since FLOW-S is rather fast when trying to find valid solutions for an VNMP instance. These valid solutions could be further optimized by one of the heuristics.

PATH does not work for the VNMP, but that does not mean that the Column Generation idea itself is without merit for the VNMP. It may be possible to devise other formulations, for instance focusing on finding good groups of virtual nodes for a substrate node instead of finding good paths for virtual arcs.

CHAPTER 12

# Application Study

## 12.1 Introduction

In the previous chapters, we have presented different methods for solving the VNMP. In this chapter, we will give an example of how these methods can be used to solve problems occurring in practice. In particular, we use one of the presented methods to evaluate at which locations in the substrate we have to invest (i.e., increase the amount of available resources) to be able to host more virtual networks.

Note the difference between this scenario and what we did previously for solving the VNMP where we allow to buy additional resources. In the VNMP scenario, we know which virtual networks we want to embed into the substrate and are able to determine, if the substrate does not have enough resources, the cheapest way of buying additional resources to fit all virtual networks into the substrate. Now we try to determine beforehand where we need to buy more resources, so that future virtual networks can be successfully embedded and do not need to be rejected. Of course, we do not know how future virtual networks will look like, so our objective is to increase the probability that a new virtual network can be embedded into the substrate, the embedding probability $p_{\mathrm{em}}$.

The remainder of this chapter is structured as follows. We give an overview on related work in Section 12.2. In Section 12.3, we present the Virtual Network Mapping model used in this chapter. It is slightly different from the VNMP, since the work presented here actually partly preceded the results from previous chapters. This section also outlines the differences in the used VNMP instances, as precursors of the VNMP instances as presented in Chapter 5 have been used. The method of determining the embedding probabilities is outlined in Section 12.4. The results of the evaluation can be found in Section 12.5. We conclude in Section 12.6. The work presented in this chapter has been published in [92].

## 12.2 Related Work

A physical network has to be able to transfer the occurring traffic and avoid packet loss and delay. The amount of data to be transferred during normal operation is not known beforehand and has to be estimated. There are different mechanisms available to ensure adequate quality of service by avoiding or minimizing link overload in the network.
To a certain extent, Traffic Engineering (TE) can be used to optimize the resource-usage of flows in the network [58]. By using TE, the physical network can carry more traffic, but the optimization is difficult and requires a careful selection of objective functions [78]. In addition to TE, the mechanism of Admission Control (AC) can be used. With AC, the load in the network is analyzed before new traffic is allowed to enter the network to avoid overload situations. This mechanism was proposed for the Internet in [157].
Both mechanisms are rather complex and require constant monitoring of the network. A much simpler approach is to use Capacity Overprovisioning (CO). With CO, the network is overdimensioned, which makes overload in unexpected scenarios very unlikely. It is often stated that CO has high initial costs in comparison to other methods, but as the network usually must also provide backup capacity for unexpected failure scenarios, the bandwidth requirements for CO are similar to those of AC [128]. To determine the best capacities for links and routers is not easy, even when accurate information about traffic patterns in a network are provided. A method to find network elements which might have insufficient capacity is implemented in the Resilyzer framework [79, 127]. It can analyze all network failure scenarios with given probability and determine for example the link overload probability, which can be used for CO.
In this chapter we present a new method to determine such bottlenecks that can be alleviated by CO, which is not based on failure probabilities but on the probability that a new virtual network can be embedded. To the best of our knowledge, no other previous work examined the use of a virtual network embedding algorithm for finding bottlenecks in the substrate.

## 12.3 Network Traffic Model

The network traffic model employed in this chapter is the same model as for the VNMP, with one exception. For the VNMP, we have only one type of resource available on the substrate nodes, the CPU capacity. It is used for hosting virtual nodes and to transfer data across the node. Here, the CPU capacity does not have this dual role and is only required for hosting virtual networks. For routing, there is a second type of resource at the substrate nodes, the routing capacity. For one unit of transferred data, one unit of routing capacity is used at the substrate node. Within this chapter, we will denote this problem by VNMP.
The second major difference in the employed methodology in contrast to the previous chapters concerns the used instance set [87]. Here we use substrate networks derived from the rocketfuel [163], scan [70] and lucient [25] projects. The instances were created similarly to the method presented in Chapter 5. They still contain virtual networks designed to represent Stream, Web, P2P and VoIP applications, albeit not the same number of each of those networks. Further differences will be mentioned as relevant, for much more detail on the design of the instances and this variant of the VNMP, see [88].

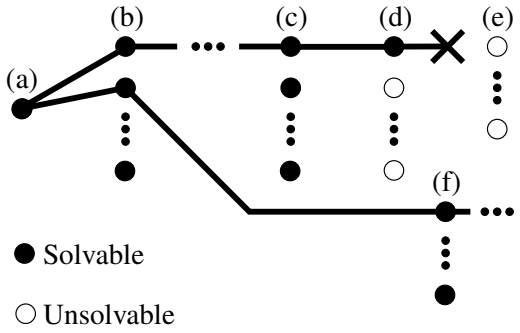



Figure 12.1: Example of an extension procedure execution.

## 12.4 Methodology

The basic idea of the conducted experiments is to show that by solving VNMP instances that do not have a valid solution we can extract valuable information in order to improve the substrate network. By improvement we mean that the probability that an additional virtual network (VN) can be embedded into the substrate (the embedding probability $p_{\mathrm{em}}$) is increased, and that more virtual networks can be embedded into the substrate before $p_{\mathrm{em}}$ reaches zero.

To show this, we perform two experiments. We start with the VNMP instances available at [87]. For those instances, we determine $p_{\mathrm{em}}$ separately for each VN type because we want to show the influence of different VN types on $p_{\mathrm{em}}$. The embedding probability of an instance is determined by adding a VN of a specific type and checking whether this new problem is solvable or not. This is repeated $n_{\mathrm{S}}$ times and $p_{\mathrm{em}}$ is calculated as the fraction of solvable cases in relation to the number of tested ones. In this work, we use $n_{\mathrm{S}} = 50$ as tradeoff between required run-time and confidence in the calculated $p_{\mathrm{em}}$. We continue this process with the first found solvable problem (i.e., the added VN could be successfully embedded in the substrate) in a depth-first fashion until $p_{\mathrm{em}}$ reaches zero or we have added ten additional VNs. Then the procedure tracks back to an instance with the least amount of added VNs that has not had its $p_{\mathrm{em}}$ determined. The motivation for this traversal order is that we want to see how $p_{\mathrm{em}}$ develops while adding additional VNs but also cover a reasonable fraction of the search space of instances reachable from the initial problem. We chose the upper limit of ten additional VNs because as more VNs are added, the resource bottlenecks in the substrate depend more and more on the exact configuration of the added VNs and not on the initial situation in the substrate and therefore become less relevant when we want to determine where we want to add additional resources to the substrate. Additionally, we used a limit of 101 $p_{\mathrm{em}}$ evaluations, which allows the cutoff point of ten added VNs to be reached ten times. For the whole procedure (which we will call extension procedure from now on) we set a time limit of one day and we executed it for each of the four VN types used for the VNMP.

Figure 12.1 shows an example execution of the extension procedure. It starts with problem instance $(a)$, an instance from the instance set. It then determines $p_{\mathrm{em}}$ for this instance by repeatedly extending $(a)$ with one random VN (generated as in [88]) and checking whether the

new instances $(b)$ are solvable or not. In the illustrated case, every newly generated instance was solvable. We continue with the first found instance and repeat this process until we reach $c$. For $c$, we determine a $p_{\mathrm{em}}$ of 2%, because only one of 50 instances is solvable. We continue with the only found solvable instance $(d)$. Node $(d)$ has a $p_{\mathrm{em}}$ of 0%, so there are no instances with which to continue; we track back to an instance with the least amount of added VNs that has not had its $p_{\mathrm{em}}$ determined, in this case the second solvable instance created from $(a)$. The extension procedure then continues with $(f)$ and so on until one of the mentioned limits is reached or no more instances remain.

During the extension procedure, a lot of unsolvable instances (i.e., VN configurations that cannot be embedded into the substrate) are created. For the second experiment, we add resources to the substrates of the starting instances based on the reasons why those instances from the first run of the extension procedure were unsolvable and then execute the extension procedure again.

The following sections describe in detail how we determined the solvability or unsolvability of a problem instance during the calculation of $p_{\mathrm{em}}$, how the reasons for unsolvability were extracted and how they were translated into substrate changes.

### 12.4.1 Proving Unsolvability and Extracting Reasons

During the execution of the extension procedure, we want to collect reasons (i.e., location and amount of missing resources) why some VN configurations could not be mapped into the substrate. From these reasons, we want to derive changes to the substrate network. To get the unsolvability reasons, we solve a modified version of the Integer Linear Program presented in [88]. It calculates the cheapest changes to the substrate (i.e., added resources), so that a specific VN configuration can be embedded. We have looked at this problem also in previous chapters. Here we consider an extended version, because we do not only allow changing bandwidth and CPU capacities, but also changing the delay on substrate arcs and the routing capacity on substrate nodes. We will now present the used ILP in detail.

The directed graph $G = (V, A)$ with node set $V$ and arc set $A$ represents the substrate network. The available CPU power of a substrate node is denoted with $c_i \in \mathbb{N}^+,\ \forall i \in V$, its routing capacity with $r_i \in \mathbb{N}^+,\ \forall i \in V$. The delay of a substrate arc is denoted by $d_e \in \mathbb{N}^+,\ \forall e \in A$, its available bandwidth with $b_e \in \mathbb{N}^+,\ \forall e \in A$. The components of the VN graph $G' = (V', A')$ are the VNs that have to be embedded into the substrate. The constant $c_k \in \mathbb{N}^+,\ \forall k \in V'$ determines the required CPU power of a virtual node. The required bandwidth by a virtual connection is defined by $b_f \in \mathbb{N}^+,\ \forall f \in A'$ and the maximum allowed delay by $d_f \in \mathbb{N}^+, \forall f \in A'$. The set $M \subseteq V' \times V$ defines the allowed mappings between virtual and substrate nodes. The functions $s : A \cup A' \to V \cup V'$ and $t : A \cup A' \to V \cup V'$ associate each arc of $G$ and $G'$ with their source and target nodes, respectively. The coefficients $p^{\mathrm{CPU}}$, $p^{\mathrm{RC}}$, $p^{\mathrm{BW}}$ and $p^{\mathrm{DL}}$ define the cost of adding one unit of CPU or routing capacity to a substrate node, a unit of bandwidth to a substrate arc or removing a unit of delay from a substrate arc. We use $p^{\mathrm{CPU}} = p^{\mathrm{RC}} = 1$, $p^{\mathrm{BW}} = 5$ and $p^{\mathrm{DL}} = 20$ to reflect the fact that it is easy to increase the resources at the substrate nodes, but adding bandwidth at substrate arcs may be difficult and expensive because new cables may need to be run. Most expensive is a change to delays, which basically means a switch of data transmission technology.

The ILP utilizes the decision variables $x_i^k \in \{0,1\},\ \forall k \in V',\ \forall i \in V$ to indicate where the virtual nodes are located in the substrate graph and $y_e^f \in \{0,1\},\ \forall f \in A',\ \forall e \in A$ to indicate if a virtual connection is implemented by using a substrate connection. The decision variable $z_i^f \in \{0,1\},\ \forall f \in A',\ \forall i \in V$ indicates that a substrate node is used to route a virtual connection. The variables $a_i^{\text{CPU}},\ \forall i \in V$ represent the added CPU capacity to a substrate node, $a_i^{\text{RC}},\ \forall i \in V$ the added routing capacity. Variables $a_e^{\text{BW}},\ \forall e \in A$ represent the added bandwidth to a substrate arc and $a_e^{\text{DL}},\ \forall e \in A$ the removed delay. Note that $a_e^{\text{DL}}$ is always non negative. The auxiliary variable $d_e^f,\ \forall e \in A,\ \forall f \in A'$ represents the delay a substrate arc has when used to implement a virtual arc and is required due to technical reasons pertaining to the implementation of delay constraints when delay changes on substrate arcs are possible.
The complete ILP is defined by inequalities (12.1)–(12.18).

$$\min \quad \sum_{i \in V} (p^{\text{CPU}} a_i^{\text{CPU}} + p^{\text{RC}} a_i^{\text{RC}}) + \sum_{e \in A} (p^{\text{BW}} a_e^{\text{BW}} + p^{\text{DL}} a_e^{\text{DL}}) \tag{12.1}$$

$$\sum_{(k,i) \in M} x_i^k = 1 \qquad \forall k \in V' \tag{12.2}$$

$$\sum_{e \in A | t(e) = i} y_e^f + x_i^{s(f)} - \sum_{e \in A | s(e) = i} y_e^f - x_i^{t(f)} = 0 \qquad \forall i \in V,\ \forall f \in A' \tag{12.3}$$

$$\sum_{e \in A | t(e) = i} y_e^f + x_i^{s(f)} \leq z_i^f \qquad \forall i \in V,\ \forall f \in A' \tag{12.4}$$

$$\sum_{(k,i) \in M} c_k x_i^k \leq c_i + a_i^{\text{CPU}} \qquad \forall i \in V \tag{12.5}$$

$$\sum_{f \in A'} b_f z_i^f \leq r_i + a_i^{\text{RC}} \qquad \forall i \in V \tag{12.6}$$

$$\sum_{f \in A'} b_f y_e^f \leq b_e + a_e^{\text{BW}} \qquad \forall e \in A \tag{12.7}$$

$$d_e y_e^f - a_e^{\text{DL}} \leq d_e^f \qquad \forall e \in A,\ \forall f \in A' \tag{12.8}$$

$$a_e^{\text{DL}} \leq d_e - 1 \qquad \forall e \in A,\ \forall f \in A' \tag{12.9}$$

$$\sum_{e \in A} d_e^f \leq d_f \qquad \forall f \in A' \tag{12.10}$$

$$x_i^k \in \{0,1\} \qquad \forall (k,i) \in M \tag{12.11}$$

$$y_e^f \in \{0,1\} \qquad \forall e \in A,\ \forall f \in A' \tag{12.12}$$

$$z_i^f \in \{0,1\} \qquad \forall i \in V,\ \forall f \in A' \tag{12.13}$$

$$a_i^{\text{CPU}} \in \mathbb{R}_0^+ \qquad \forall i \in V \tag{12.14}$$

$$a_i^{\text{RC}} \in \mathbb{R}_0^+ \qquad \forall i \in V \tag{12.15}$$

$$a_e^{\text{BW}} \in \mathbb{R}_0^+ \qquad \forall e \in A \tag{12.16}$$

$$a_e^{\text{DL}} \in \mathbb{R}_0^+ \qquad \forall e \in A \tag{12.17}$$

$$d_e^f \in \mathbb{R}_0^+ \qquad \forall e \in A,\ \forall f \in A' \tag{12.18}$$

The objective is given by (12.1), the total cost of added resources is to be minimized. Equalities (12.2) ensure that each virtual node is mapped to exactly one substrate node, subject to the mapping constraints. The flow conservation constraints (12.3) make sure that for each virtual connection there is a connected path in the substrate. Linking constraints (12.4) ensure that variables $z_i^f$ are equal to one when the corresponding node is used to route the traffic of a particular virtual connection. Inequalities (12.5)–(12.7) ensure that the solution is valid with regard to CPU, routing capacity and bandwidth constraints while also considering added resources. Incorporating changes to the substrate delay is not as straight forward. Inequalities (12.8) together with the domain of $d_e^f$ ensure that $d_e^f$ is either zero if substrate arc $e$ is not used by virtual arc $f$ (i.e., $y_e^f$ is zero) or has the correct delay value (i.e., defined delay minus delay reduction) if the substrate arc is used. Inequalities (12.9) make certain that even after reduction, the delay of a substrate connection is at least 1. Inequalities (12.10) ensure that the solution is valid with regards to the delay constraints. Equations (12.11)–(12.18) define the domains of the used variables. Note that the model only includes integrality constraints for $x_i^k$, $y_e^f$ and $z_i^f$ (12.11)–(12.13), the already covered constraints together with the objective function cause $a_i^{\text{CPU}}$, $a_i^{\text{RC}}$, $a_e^{\text{BW}}$ and $a_e^{\text{DL}}$ to be integral as well. Every non-zero $a$ variable in an optimal solution counts as one failure reason because a resource had to be added in order to embed all VNs.

### 12.4.2 Reacting to Failure Reasons

Executing the extension procedure for one instance typically creates $\approx 3000$ unsolvable VN configurations. With the help of the ILP formulation from the previous section, $\approx 8000$ reasons for unsolvability can be extracted. These have to be condensed into concrete changes for the substrate. The following lists the four parameters we used for calculating the changes to the substrate:

$f$ Function that calculates a descriptive value from a set of missing resources, e.g., mean or max

$s$ Scaling factor for the result of $f$ to calculate the added amount of resources

$r$ How often resources (of one type) have to be missing at a specific location in the substrate in relation to the maximum number of reported missing resources in order for this location to receive additional resources

$n$ Maximum number of resource changes in the substrate (per resource type)

Our aim is to add more resources to the most critical parts of the substrate network. Critical parts are those that are frequently reported as having too few resources of a specific type available. Let this report count be called $m$, and the highest count $m_{\max}$. Note that the amount of missing resources that is reported is not yet relevant. So, for routing capacity, bandwidth, CPU capacity and delay separately, we regard substrate nodes (in case of routing and CPU capacity) and arcs (for bandwidth and delay) in descending order of $m$. All locations with $m \geq r \cdot m_{\max}$, but at most $n$, will receive additional resources. These cutoff rules ensure that we do not add resources to too many locations in the substrate (which would not be economical) and also not to locations

which only rarely miss resources. The amount of additional resources is determined by applying $f$ to the reported amounts of missing resources at the selected locations and multiplying the result by $s$. In this work, we used $f = mean$, $s = 3$, $r = 0.3$, $n = 5$. That means that for each resource type separately, at most five locations receive additional resources. If, for example, bandwidth was reported to be missing on a substrate arc ten times, and bandwidth was not required on another arc more often, then every substrate arc that required more bandwidth at least three times gets additional bandwidth. If these are more than five arcs, the five arcs that reported missing bandwidth most often are selected to receive additional resources. For one particular substrate arc selected to receive additional bandwidth, we calculate the mean reported missing bandwidth and add three times as much. For consistency reasons, we apply $s$ also to delay changes (ensuring that the resulting delay is still at least one), even though this is not strictly necessary, since delay is not consumed in the same way as for instance bandwidth is by additional virtual networks.

## 12.5 Results

From the instance set, we chose 80 instances with a load of 0.8. Due to the computational demand of the extension procedure, we could not use the complete instance set. We set a time-limit of 300 seconds to solve the presented ILP model, for substrates of size 100 (the largest size for this instance set) a time-limit of 500 seconds was employed. If the optimal solution to the ILP was not found within the time-limit, we used the best found feasible solution if the gap to the optimal solution was smaller than 95%. Preliminary runs showed that for larger substrates the feasible solution reported by CPLEX sometimes was the result of heuristics CPLEX runs before it actually starts to solve the ILP. For this problem, the solutions generated in this way add a lot of resources to almost all nodes and arcs in the substrate and are therefore not helpful for finding the real bottlenecks in the substrate. The gap limit ensures that we do not use those solutions.

### 12.5.1 VNMP Instance Properties

In this section we show the properties of the 80 VNMP instances used as base for the performed experiments. Table 12.1 shows the number of nodes and arcs and the number of virtual networks of each type contained in the instances. The first thing to note is that we only tested 10 instances of size 100, instead of the 14 for all other instance sizes. The reason for this is that for size 100, the instance set does not contain as many instances as for the other sizes since some of the rocketfuel networks used as basis for the substrate networks are smaller than 100 nodes. The development of the number of substrate arcs, virtual nodes and virtual arcs is similar to the instances presented in Chapter 5. The remaining part of the table shows the average number of virtual networks of each type contained in the instances. Previously, this was constant at 10 virtual networks for each type. These instances however have been created by completely specifying a substrate network and then adding virtual networks of a random type and checking if a valid solution exists until no more virtual networks could be added. As a result, the instances contain more Stream and Web slices, since they require fewer connections and also not as many

Table 12.1: Number of used VNMP instances per size (#) and the average number of substrate arcs ($A$), virtual nodes ($V'$), virtual arcs ($A'$) and contained Stream, Web, P2P, and VoIP virtual networks for those instances.

| Size | # | A | V' | A' | Stream | Web | P2P | VoIP |
|---|---|---|---|---|---|---|---|---|
| 20 | 14 | 53.7 | 118.2 | 146.0 | 6.6 | 8.5 | 4.4 | 4.1 |
| 30 | 14 | 95.4 | 215.0 | 264.0 | 13.1 | 14.6 | 7.6 | 7.8 |
| 40 | 14 | 126.1 | 275.4 | 382.0 | 16.4 | 18.5 | 6.9 | 7.1 |
| 50 | 14 | 172.1 | 308.0 | 523.9 | 14.4 | 16.5 | 7.5 | 6.4 |
| 70 | 14 | 259.4 | 478.7 | 835.6 | 18.1 | 17.9 | 6.1 | 7.6 |
| 100 | 10 | 399.0 | 648.0 | 1276.9 | 14.0 | 15.8 | 6.8 | 9.1 |

Table 12.2: Fraction of failure reasons of a specific type (missing CPU- (C) or routing- (R) capacity, missing bandwidth (B) or too much delay (D)) during the first run of the extension procedure for each VN type in relation to the total number of found failure reasons in percent. (May not add up to exactly 100% due to rounding.)

| | Stream | | | | Web | | | | P2P | | | | VoIP | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Size | C | R | B | D | C | R | B | D | C | R | B | D | C | R | B | D |
| 20 | 39.9 | 56.0 | 4.1 | 0.0 | 12.5 | 66.2 | 0.0 | 21.3 | 11.9 | 80.3 | 7.8 | 0.0 | 11.5 | 88.0 | 0.2 | 0.2 |
| 30 | 35.5 | 43.7 | 20.7 | 0.0 | 10.6 | 40.8 | 12.3 | 36.3 | 5.1 | 53.3 | 41.6 | 0.0 | 4.3 | 48.1 | 40.3 | 7.3 |
| 40 | 42.9 | 46.1 | 11.0 | 0.0 | 8.9 | 37.5 | 9.1 | 44.4 | 13.3 | 58.8 | 27.9 | 0.0 | 16.2 | 42.2 | 12.9 | 28.7 |
| 50 | 39.5 | 38.6 | 21.9 | 0.0 | 6.2 | 38.0 | 12.7 | 43.1 | 1.8 | 78.7 | 19.0 | 0.5 | 4.6 | 45.5 | 31.6 | 18.4 |
| 70 | 53.7 | 33.4 | 12.8 | 0.0 | 9.2 | 28.0 | 13.3 | 49.4 | 6.0 | 61.7 | 31.7 | 0.6 | 5.6 | 58.7 | 34.0 | 1.7 |
| 100 | 48.9 | 36.5 | 14.6 | 0.0 | 4.0 | 42.6 | 3.3 | 50.1 | 5.2 | 88.8 | 6.1 | 0.0 | 19.1 | 74.1 | 5.1 | 1.6 |

resources as the P2P and VoIP networks and therefore have a higher probability to fit into the substrate network. The total number of virtual networks contained in the instances does not rise because their size grows with the substrate network size.

### 12.5.2 Extension Procedure

After we executed the extension procedure for all 80 instances and for each of the four virtual network (VN) types, the extracted failure reasons were distributed as shown in Table 12.2.
It can be seen that in general missing routing capacities are prevalent, no matter which type the additionally added VNs are. For Stream VNs, the fraction of missing routing capacities is reduced while CPU capacities are missing more often with increasing substrate size. Normally, one would expect the development to be the other way around, because routing capacities are needed at multiple nodes in the substrate network to implement a virtual connection, while CPU capacities are only needed at the start and the end of a virtual connection. When the substrate grows, so do the average lengths of the implementations of virtual connections and the total required routing capacity increases while the CPU capacity stays the same. This would cause the probability of missing routing capacity to increase. The reason why missing CPU resources become more prominent is that the nodes of stream VNs require a lot of CPU capacity (because they have to split and distribute video streams). The VN sizes grow proportional to the substrate sizes, therefore the Stream VNs for larger substrates can contain more nodes which perform

video stream splitting which in turn requires more CPU resources. In relation to the missing CPU and routing capacities, the substrate link bandwidths only play a subsidiary role and no link was found to have too much delay, which is not surprising since the virtual connections in Stream VNs are not delay constrained.

This is not true for Web VNs which are heavily delay constrained. For substrates of size 100, more than half of all found failure reasons are too high delay on some substrate arc. This fraction rises with increasing substrate size because the average length of virtual arc implementations and therefore the total accumulated delay grows.

When embedding additional P2P VNs into the substrate, missing bandwidths on substrate links become a significant issue for the first time, even though missing routing capacities still dominate. Again, this is because the (already high) bandwidth requirements of P2P VNs require a lot of routing capacity in the substrate nodes and depending on the particular substrate configuration one resource is more limiting than the other. Also note that a bias against bandwidth changes exists, since adding one unit of bandwidth costs five times more than a unit of routing capacity. That means that if an additional VN can be added either by adding four units of routing capacity or adding one unit of bandwidth, four units of routing capacity are reported missing because this is the cheaper solution. However, this effect can also be witnessed the other way around with this VN type. Even though P2P VNs are not delay constrained, for sizes 50 and 70 delays were reported missing. This can happen because the initial instances contain all four VN types, i.e., also some with are delay constrained. By reducing the delay somewhere, the virtual connections with stringent delay requirements can be implemented by using a previously impossible substrate path which frees up resources for new VNs where they are needed. This can be the cheaper solution, even though decreasing link delay by one unit costs twenty times more than adding a unit of routing capacity for instance.

For VoIP VNs, missing bandwidths are again prominent (after missing routing capacities). Also link delays play some role, but since VoIP VNs are not as delay constrained as Web VNs it is not surprising to see that delays are less often a failure reason when compared to Web VNs. Unfortunately we can offer no conclusive reason as to why the fraction of delay reasons fluctuates so much for VoIP VNs. Larger instances reduce the probability that the presented ILP can be solved to optimality, so one possible explanation could be that feasible solutions which only change other resources are predominantly found. We will not go into detail regarding the performance of the ILP, but in a nutshell, it works well up to substrate sizes of 50, i.e., more than 99% of executions yield useful results (a valid solution within the gap limit). For sizes 70 and 100, there is a split between Stream and Web VNs on the one hand and P2P and VoIP VNs on the other. For Stream and Web VNs, we still get useful results more than 90% of the time. For P2P and VoIP VNs, this is reduced to 85% for instances of size 70 and even 45% for size 100. Note that at the time these experiments were performed, the preprocessing techniques outlined in Chapter 9 had not been developed.

Based on the reported missing resources for each instance, we added resources to each VNMP instance as outlined in Section 12.4.2. Table 12.3 shows the change to the available resources in the substrate because of the added resources.

It can be seen that only a very small amount of resources was added to the substrate, as was our goal. Generally, the total available amount of resources was increased by less than one percent.

Table 12.3: Relative change of the available resources in the substrate due to the found failure reasons in percent.

| | Stream | | | | Web | | | | P2P | | | | VoIP | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **C** | **R** | **B** | **D** | **C** | **R** | **B** | **D** | **C** | **R** | **B** | **D** | **C** | **R** | **B** | **D** |
| 20 | 1.4 | 0.7 | 0.2 | 0.0 | 0.1 | 0.3 | 0.0 | -8.4 | 0.1 | 2.1 | 0.0 | 0.0 | 0.2 | 2.2 | 0.0 | -1.2 |
| 30 | 0.6 | 0.4 | 0.2 | 0.0 | 0.0 | 0.1 | 0.0 | -5.9 | 0.1 | 1.0 | 0.1 | 0.0 | 0.1 | 0.9 | 0.1 | -3.8 |
| 40 | 0.4 | 0.3 | 0.1 | 0.0 | 0.0 | 0.1 | 0.0 | -7.5 | 0.1 | 1.2 | 0.1 | 0.0 | 0.2 | 1.2 | 0.1 | -3.5 |
| 50 | 0.2 | 0.1 | 0.2 | -0.2 | 0.0 | 0.1 | 0.1 | -6.5 | 0.0 | 0.8 | 0.3 | -0.9 | 0.1 | 0.9 | 0.3 | -5.7 |
| 70 | 0.2 | 0.1 | 0.0 | -0.2 | 0.0 | 0.0 | 0.0 | -2.3 | 0.0 | 0.5 | 0.1 | -0.3 | 0.1 | 0.5 | 0.1 | -0.4 |
| 100 | 0.1 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -1.0 | 0.0 | 0.3 | 0.0 | 0.0 | 0.0 | 0.3 | 0.0 | -0.1 |

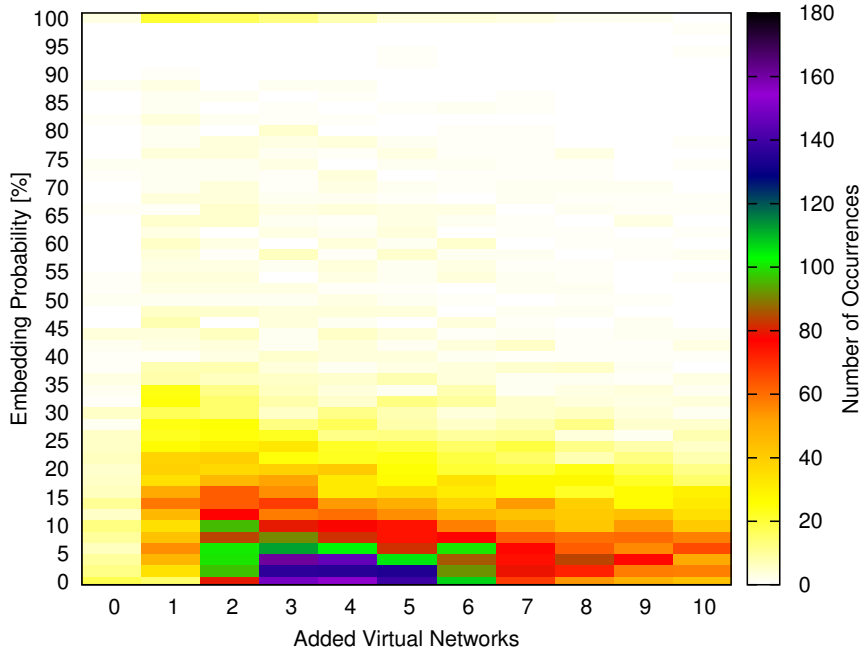
Table 12.4: Average $p_{\mathrm{em}}$ in percent for the first run of the extension procedure and for the second run with additional substrate resources.

| | Stream | | Web | | P2P | | VoIP | |
|---|---|---|---|---|---|---|---|---|
| **Size** | **1st** | **2nd** | **1st** | **2nd** | **1st** | **2nd** | **1st** | **2nd** |
| 20 | 13.6 | 38.8 | 18.4 | 59.7 | 7.4 | 20.2 | 7.9 | 20.4 |
| 30 | 13.1 | 39.6 | 14.7 | 60.8 | 3.3 | 18.2 | 3.7 | 15.5 |
| 40 | 13.5 | 42.7 | 11.7 | 60.6 | 4.3 | 15.4 | 3.1 | 14.3 |
| 50 | 15.7 | 40.7 | 9.0 | 51.7 | 11.6 | 13.6 | 7.2 | 14.7 |
| 70 | 12.7 | 33.5 | 11.3 | 31.5 | 2.6 | 11.7 | 2.9 | 15.4 |
| 100 | 8.0 | 31.7 | 12.2 | 30.9 | 0.0 | 0.9 | 0.0 | 0.6 |

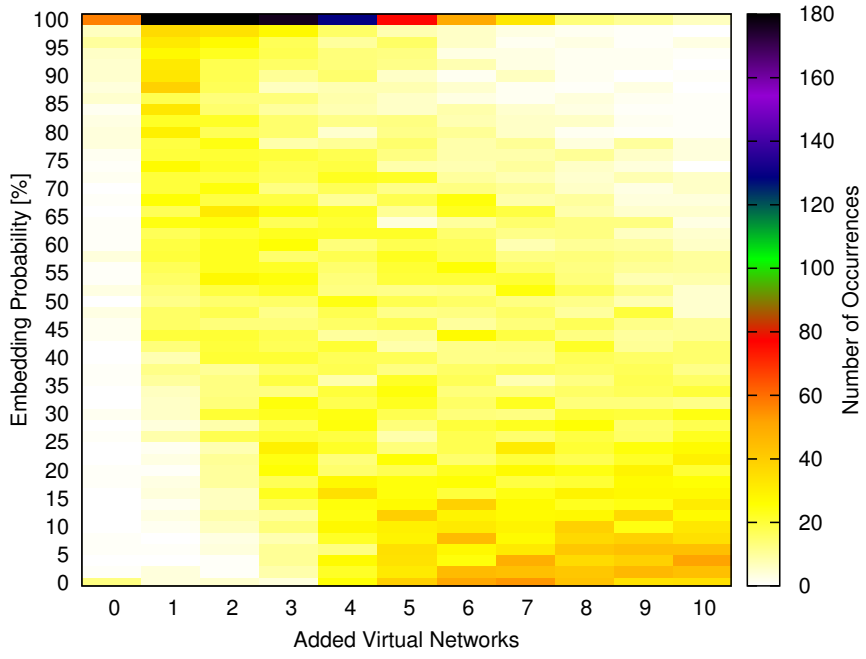
The most notable exception to this are the delays when adding Web or VoIP VNs. There are three factors that work together to cause this. First of all, Web and VoIP VNs are delay constrained, so even though delays are not very often reported to be too high, the reported magnitude *is* high. Secondly, as we have already pointed out, $s$ is also applied when calculating delay changes for uniformity reasons, even though it is not strictly necessary. If the delay of a substrate connection is reduced all future VNs will benefit, at least in our simplified model where substrate link delays are independent of link load. The same is not true for the other resource types, since they are used up by additional VNs, which is the reason why we add more resources than are reported missing. The third contributing factor to the large delay changes is the fact that the sum of all delays in the substrate is less than the sum of the other resource types, so changing the delay by some fixed amount will be a larger relative change than for the other resources.
The following section will answer the question whether the resources added to the substrate could influence $p_{\mathrm{em}}$ in a meaningful way.

### 12.5.3 Change to the Embedding Probability

The complete development of $p_{\mathrm{em}}$ during both runs of the extension procedure can be seen in Figures 12.2 and 12.3. They show how often a specific $p_{\mathrm{em}}$ occurred depending on the number of added VNs. Stream and Web have been combined into one graph, as have P2P and VoIP, because they show very similar behaviour. Clearly visible is the shift of the clusters to the right, i.e., more VNs can be added before the substrate is full. Figure 12.2b shows that for

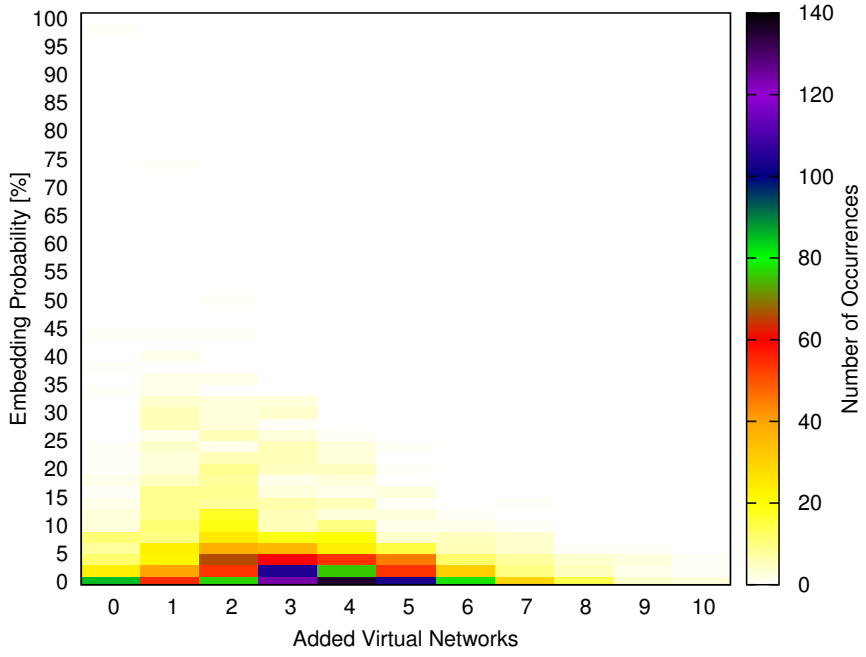



(a) Stream+Web

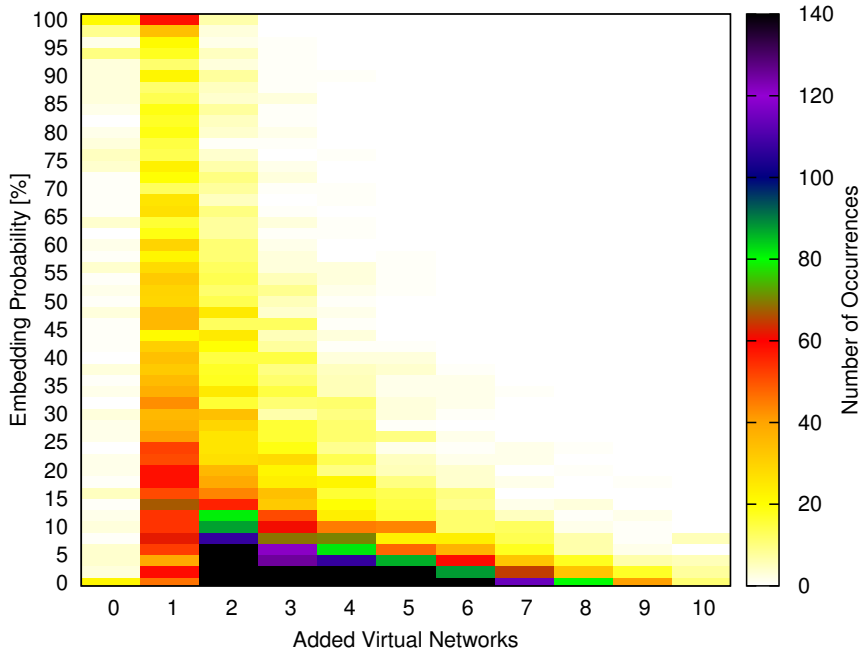



(b) Stream+Web-R

Figure 12.2: $p_{\text{em}}$ development for both runs of the extension procedure for Stream and Web VNs. Suffix R denotes the second run with additional resources for the substrates.




(a) P2P+VoIP




(b) P2P+VoIP-R

Figure 12.3: $p_{\text{em}}$ development for both runs of the extension procedure for P2P and VoIP VNs. Suffix R denotes the second run with additional resources for the substrates.

Stream and Web VNs, the added resources cause a substantial fraction of instances to stay at $p_{\mathrm{em}} = 1$ during the first five added VNs. This is in contrast to the situation before additional resources were added, as seen in Figure 12.2a. After five added VNs, most instances show a $p_{\mathrm{em}}$ smaller than $0.2$. A similar, although not as pronounced, development can be seen for P2P and VoIP VNs in Figure 12.3. It can be seen that during the first run of the extension procedure (Figure 12.3a) it happens very often that the initial VNMP instance cannot be extended with an additional VN. Nearly every initial instance has $p_{\mathrm{em}} \leq 0.12$. After adding additional resources, $p_{\mathrm{em}}$ of the initial instance covers the whole range of possibilities. Adding further VNs reduces $p_{\mathrm{em}}$ far more rapidly than for Stream and Web VNs. On the whole the added resources are more effective for influencing $p_{\mathrm{em}}$ for Stream and Web VNs than they are for P2P and VoIP VNs. There are two possible explanations for this behaviour. First of all, since the first run of the extension procedure for P2P and VoIP VNs often could not extend the initial instance, the collected failure reasons only contain reasons why adding one VN might fail. Therefore, after adding resources, only adding the first VN works well and $p_{\mathrm{em}}$ rapidly approaches zero afterwards. Another possible explanation is that the used settings for determining where and how much resources are added might not cause enough resources to be added so that multiple VNs can be embedded in the substrate. The chosen settings work well for Stream and Web VNs, but might be too conservative for P2P and VoIP VNs, which require more resources per VN.
In Table 12.4, we show the average embedding probability for all instances generated by both runs of the extension procedure. It can be seen that the greatest gains have been achieved for Web VNs (which also had the largest resource change). A close second are Stream VNs, where less than $1\%$ change in resources increased $p_{\mathrm{em}}$ by $20\%$ and more on average. The improvements for P2P and VoIP VNs were not as great, especially for instances of size 100 where the ILP fails to extract useful failure reasons due to time and memory constraints.

## 12.6 Conclusion

In this chapter, we presented a method how an ILP as presented in Chapter 11 can be adapted to successfully identify bottlenecks in a substrate network. Our results show that less than one percent of additional resources in the substrate network can increase the probability that additional VNs can be embedded by $20\%$ or more. We also showed that different use cases (i.e., different VNs) lead to different resources being added to the substrate. Therefore, Virtual Network Operators (VNOs) are able to optimize their networks to cater best to the VN types they encounter the most, or are most profitable.
With the presented method, a VNO could monitor its current network situation and decide whether leasing additional resources at a specific location is necessary to provide a certain service availability. Knowing these bottlenecks in advance offers the VNO the possibility to add additional resources before it is too late.
We have shown in this chapter the flexibility of the VNMP solution approaches. They can be easily adapted to solve related practical problems occurring when operating VNs.

CHAPTER 13

# Comparison and Conclusions

## 13.1 Introduction

In the final chapter of this work, we present a comparison of the main algorithms introduced previously, both for solving VNMP-S and VNMP-O. The algorithms chosen for comparison are CH-S and CH-O as defined in Section 6.5.1, LS-S and LS-O as defined in Section 6.5.2, VND-S and VND-O as defined in Section 6.5.3, MA-O as defined in Section 7.5, GRASP-O as defined in Section 8.4.1, VNS-O as defined in Section 8.4.2, and ILP-S and ILP-O as defined in Section 11.4.3. For an even more extensive presentation of the results achieved by those algorithms, we refer to Appendix A.

The presented data is based on the full VNMP instance set as defined in Chapter 5 at loads 0.1, 0.5, 0.8 and 1, giving a total of 840 VNMP instances. Comparing the exact algorithms (ILP-S and ILP-O) to the others is problematic, since they may fail to produce any solution at all, for instance when they are aborted due to the memory limit. To offer a meaningful comparison, we chose the following approach: For every VNMP instance, we keep track of the highest additional resource cost $C_a$ and usage cost $C_u$ produced by any compared algorithm. If one of the exact methods failed to generate a result, we assume it generated a result with the worst $C_u$ increased by one, the worst $C_a$ increased by one, and required 10000 seconds (the time-limit used for the exact methods).

In Section 13.2, we analyze the number of valid solutions found by each algorithm. Section 13.3 is concerned with the $C_a$ of the found solutions, while the achieved relative ranks $R_{\text{rel}}$ are covered in Section 13.4. The gap of the substrate usage cost (and the definition of this metric) is discussed in Section 13.5. Required run-times are presented in Section 13.6. We conclude in Section 13.7 and outline future work in Section 13.8.

Table 13.1: Number of valid solutions found by the main algorithms presented in this work.

| Size | Load | CH-O | LS-O | VND-O | MA-O | GRASP-O | VNS-O | ILP-O | CH-S | LS-S | VND-S | ILP-S |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 0.10 | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = |
| | 0.50 | 29 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = |
| | 0.80 | 27 = | 29 = | 29 = | 30 = | 30 = | 30 = | 30 = | 29 = | 29 = | 29 = | 30 = |
| | 1.00 | 13 < | 24 < | 27 = | 30 = | 30 = | 30 = | 30 = | 23 < | 27 = | 28 = | 30 = |
| 30 | 0.10 | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = |
| | 0.50 | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = |
| | 0.80 | 23 < | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = |
| | 1.00 | 13 < | 28 = | 30 = | 30 = | 30 = | 30 = | 30 = | 28 = | 30 = | 30 = | 30 = |
| 50 | 0.10 | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = |
| | 0.50 | 29 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = |
| | 0.80 | 25 < | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = |
| | 1.00 | 7 < | 27 = | 28 = | 30 = | 30 = | 30 = | 30 = | 27 = | 28 = | 29 = | 30 = |
| 100 | 0.10 | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = |
| | 0.50 | 26 < | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 29 = | 29 = | 30 = |
| | 0.80 | 18 < | 29 = | 29 = | 30 = | 30 = | 30 = | 30 = | 28 = | 29 = | 29 = | 30 = |
| | 1.00 | 3 < | 26 < | 27 = | 30 = | 28 = | 30 = | 28 = | 27 = | 24 < | 28 = | 30 = |
| 200 | 0.10 | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = |
| | 0.50 | 22 < | 30 = | 30 = | 30 = | 30 = | 30 = | 29 = | 26 < | 29 = | 30 = | 30 = |
| | 0.80 | 6 < | 28 = | 30 = | 28 = | 29 = | 30 = | 25 < | 21 < | 27 = | 30 = | 30 = |
| | 1.00 | 0 < | 12 < | 27 = | 27 = | 29 = | 27 = | 21 < | 7 < | 19 < | 28 = | 30 = |
| 500 | 0.10 | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = |
| | 0.50 | 21 < | 27 = | 30 = | 28 = | 28 = | 29 = | 27 = | 24 < | 26 < | 30 = | 30 = |
| | 0.80 | 3 < | 20 < | 30 = | 26 < | 24 < | 24 < | 7 < | 9 < | 18 < | 30 = | 30 = |
| | 1.00 | 0 < | 0 < | 18 < | 8 < | 10 < | 9 < | 0 < | 0 < | 4 < | 16 < | 30 = |
| 1000 | 0.10 | 27 = | 30 = | 30 = | 30 = | 30 = | 30 = | 30 = | 26 < | 30 = | 30 = | 30 = |
| | 0.50 | 9 < | 25 < | 28 = | 26 < | 26 < | 27 = | 20 < | 14 < | 24 < | 28 = | 30 = |
| | 0.80 | 0 < | 8 < | 17 < | 12 < | 16 < | 12 < | 0 < | 1 < | 10 < | 17 < | 30 = |
| | 1.00 | 0 < | 0 < | 2 < | 2 < | 0 < | 2 < | 0 < | 0 < | 1 < | 2 < | 29 = |

## 13.2 Number of Valid VNMP Solutions

In this section, we focus on the capabilities of the main algorithms with respect to VNMP-S and present the number of valid solutions found in Table 13.1. The very best performance is achieved by ILP-S, which only fails to find a valid solution for one single instance. However, most of the other compared algorithms achieve nearly the same performance for instances up to size 100. VND-S is competitive up to size 200, but then fails often for instances with load 0.8 and 1. The same is true for MA-O, GRASP-O, and VNS-O. The hardest VNMP instances with respect to VNMP-S are those of size 500 and 1000 with full load. For those, ILP-S is able to find valid solutions for nearly all instances. The next best algorithm (VND-O) finds 18 valid solutions for size 500 and 2 for size 1000. Therefore, there is still room for improvement for heuristics when solving VNMP-S for the largest instances.

## 13.3 Additional Resource Cost

Table 13.2 presents the average achieved $C_a$ of the compared algorithms. This allows us to judge how far the found solutions are away from validity ($C_a = 0$). The most interesting comparison can be performed for instances of size 1000 and load 1. We know from the previous section that all algorithms (with the exception of ILP-S) fail to find valid solutions for basically all instances. Now we can see, that while those algorithms are the same with respect to the number of solved instances, there are huge differences as to their distance to optimality. The best algorithm is VND-S, with an average $C_a$ of 303. The construction heuristic geared towards solving VNMP-S is more than ten times worse, with a cost of 3200. When we use CH-O, the cost is 23300, nearly a 77-fold increase compared to the best value. ILP-O achieves basically the same cost. This is the result of how we set the costs if no solution is found. Note how the average cost of 1352 for ILP-S is marked as the best value. From the previous section, we know that this average is the result of 29 solved instances (i.e., 0 for $C_u$) and one instance for which no result was produced, which is punished by using the $C_a$ value from the worst algorithm (CH-O). This is better than failing for nearly every instance with small values of $C_a$.

## 13.4 Relative Rank

We now start considering the performance of the different algorithms with respect to VNMP-O. In Table 13.3, we show their average relative ranks. Unsurprisingly, ILP-O dominates according to this metric, only starting with size 100 other methods achieve better results. This is mostly due to ILP-O failing because of excessive memory requirements. For larger instances, ILP-O cannot be beaten for a load of 0.1. When we consider the performance of the heuristic methods, we see that MA-O and VNS-O are close to ILP-O up to size 100. For larger sizes VND-O is best and also able to outperform ILP-O in some instances. The algorithms focusing on VNMP-S generally have the worst ranks. Considering that CH-O often fails to find valid solutions, it is surprising that ILP-S is even worse. For size 500 and load 0.1, this method consistently finds the most expensive solutions. However, this is reversed for the highest loads and largest instance sizes, where ILP-S performs best since all other methods struggle to find valid solutions.

Table 13.2: Average $C_a$ achieved by the main algorithms presented in this work.

| Size | Load | CH-O | LS-O | VND-O | MA-O | GRASP-O | VNS-O | ILP-O | CH-S | LS-S | VND-S | ILP-S |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 2.3 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 55.3 = | 6.0 = | 6.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 1.7 = | 6.0 = | 6.0 = | 0.0 = |
| | 1.00 | 386.0 > | 101.3 > | 29.5 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 36.0 > | 46.3 = | 11.8 = | 0.0 = |
| 30 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 62.5 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 1.00 | 786.9 > | 50.7 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 2.8 = | 0.0 = | 0.0 = | 0.0 = |
| 50 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 1.5 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.80 | 27.5 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 1.00 | 1905.5 > | 124.8 = | 8.4 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 21.6 = | 22.7 = | 5.3 = | 0.0 = |
| 100 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 31.5 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 2.3 = | 2.3 = | 0.0 = |
| | 0.80 | 301.3 > | 2.3 = | 2.3 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 12.5 = | 2.3 = | 2.3 = | 0.0 = |
| | 1.00 | 1470.6 > | 30.7 = | 14.7 = | 0.0 = | 10.0 = | 0.0 = | 119.5 = | 144.0 = | 53.7 > | 5.5 = | 0.0 = |
| 200 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 86.2 > | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 8.3 = | 69.1 = | 0.8 = | 0.0 = | 0.0 = |
| | 0.80 | 648.9 > | 2.2 = | 0.0 = | 1.2 = | 1.3 = | 0.0 = | 74.0 > | 132.6 > | 3.0 = | 0.0 = | 0.0 = |
| | 1.00 | 5480.0 > | 229.8 > | 4.8 = | 11.0 = | 0.2 = | 24.3 = | 1742.9 > | 585.0 > | 99.4 > | 4.0 = | 0.0 = |
| 500 | 0.10 | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 25.4 > | 1.9 = | 0.0 = | 1.8 = | 1.3 = | 1.1 = | 20.0 = | 16.7 > | 2.3 = | 0.0 = | 0.0 = |
| | 0.80 | 2075.8 > | 339.3 > | 0.0 = | 35.8 = | 29.1 > | 34.1 > | 1902.3 > | 381.6 > | 106.2 > | 0.0 = | 0.0 = |
| | 1.00 | 13765.3 > | 1963.2 > | 57.9 > | 271.7 > | 158.1 > | 238.8 > | 13765.3 > | 2214.2 > | 496.1 > | 47.6 > | 0.0 = |
| 1000 | 0.10 | 1.1 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 0.0 = | 1.9 = | 0.0 = | 0.0 = | 0.0 = |
| | 0.50 | 237.9 > | 2.5 > | 0.4 = | 2.1 = | 2.1 = | 47.3 = | 158.4 > | 34.8 > | 3.1 > | 0.6 = | 0.0 = |
| | 0.80 | 3993.5 > | 290.0 > | 45.5 > | 112.9 > | 98.0 > | 173.4 > | 3993.5 > | 510.1 > | 151.0 > | 49.3 > | 0.0 = |
| | 1.00 | 23278.3 > | 2527.5 > | 659.1 > | 748.5 > | 881.8 > | 1024.0 > | 23279.2 > | 3211.0 > | 811.4 > | 303.2 > | 1352.6 = |

Table 13.3: Average $R_{\text{rel}}$ achieved by the main algorithms presented in this work.

| Size | Load | CH-O | LS-O | VND-O | MA-O | GRASP-O | VNS-O | ILP-O | CH-S | LS-S | VND-S | ILP-S |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 0.10 | 0.619 > | 0.209 > | 0.186 > | 0.013 = | 0.086 > | 0.031 > | 0.000 = | 0.749 > | 0.506 > | 0.657 > | 0.971 > |
| | 0.50 | 0.698 > | 0.382 > | 0.321 > | 0.096 > | 0.220 > | 0.103 > | 0.000 = | 0.844 > | 0.571 > | 0.703 > | 0.969 > |
| | 0.80 | 0.726 > | 0.460 > | 0.411 > | 0.124 > | 0.266 > | 0.130 > | 0.000 = | 0.863 > | 0.648 > | 0.764 > | 0.910 > |
| | 1.00 | 0.883 > | 0.523 > | 0.421 > | 0.132 > | 0.278 > | 0.150 > | 0.000 = | 0.837 > | 0.613 > | 0.688 > | 0.765 > |
| 30 | 0.10 | 0.560 > | 0.199 > | 0.167 > | 0.014 = | 0.099 > | 0.028 > | 0.000 = | 0.771 > | 0.490 > | 0.715 > | 0.990 > |
| | 0.50 | 0.626 > | 0.384 > | 0.304 > | 0.092 > | 0.231 > | 0.107 > | 0.000 = | 0.852 > | 0.587 > | 0.760 > | 0.993 > |
| | 0.80 | 0.745 > | 0.449 > | 0.359 > | 0.150 > | 0.277 > | 0.123 > | 0.000 = | 0.854 > | 0.604 > | 0.758 > | 0.960 > |
| | 1.00 | 0.859 > | 0.482 > | 0.403 > | 0.183 > | 0.286 > | 0.136 > | 0.000 = | 0.846 > | 0.621 > | 0.715 > | 0.889 > |
| 50 | 0.10 | 0.541 > | 0.276 > | 0.165 > | 0.019 = | 0.179 > | 0.048 > | 0.000 = | 0.831 > | 0.551 > | 0.775 > | 1.000 > |
| | 0.50 | 0.680 > | 0.454 > | 0.366 > | 0.155 > | 0.277 > | 0.111 > | 0.000 = | 0.884 > | 0.618 > | 0.785 > | 0.988 > |
| | 0.80 | 0.729 > | 0.484 > | 0.371 > | 0.183 > | 0.319 > | 0.129 > | 0.000 = | 0.868 > | 0.630 > | 0.779 > | 0.983 > |
| | 1.00 | 0.937 > | 0.514 > | 0.421 > | 0.188 > | 0.331 > | 0.119 > | 0.000 = | 0.794 > | 0.636 > | 0.706 > | 0.874 > |
| 100 | 0.10 | 0.630 > | 0.411 > | 0.319 > | 0.151 > | 0.270 > | 0.092 > | 0.000 = | 0.876 > | 0.644 > | 0.818 > | 1.000 > |
| | 0.50 | 0.697 > | 0.448 > | 0.309 > | 0.270 > | 0.325 > | 0.142 > | 0.000 = | 0.881 > | 0.659 > | 0.784 > | 0.975 > |
| | 0.80 | 0.797 > | 0.450 > | 0.323 > | 0.231 > | 0.362 > | 0.140 > | 0.019 = | 0.853 > | 0.644 > | 0.749 > | 0.940 > |
| | 1.00 | 0.967 > | 0.465 > | 0.339 > | 0.240 > | 0.393 > | 0.137 > | 0.067 = | 0.762 > | 0.647 > | 0.667 > | 0.838 > |
| 200 | 0.10 | 0.596 > | 0.464 > | 0.324 > | 0.219 > | 0.254 > | 0.121 > | 0.000 = | 0.892 > | 0.694 > | 0.809 > | 0.978 > |
| | 0.50 | 0.710 > | 0.461 > | 0.240 > | 0.272 > | 0.364 > | 0.125 > | 0.070 = | 0.892 > | 0.673 > | 0.757 > | 0.956 > |
| | 0.80 | 0.915 > | 0.450 > | 0.174 = | 0.289 > | 0.397 > | 0.155 = | 0.185 = | 0.859 > | 0.647 > | 0.669 > | 0.850 > |
| | 1.00 | 0.997 > | 0.680 > | 0.190 = | 0.265 > | 0.324 > | 0.201 = | 0.300 = | 0.808 > | 0.647 > | 0.562 > | 0.677 > |
| 500 | 0.10 | 0.598 > | 0.445 > | 0.254 > | 0.341 > | 0.282 > | 0.169 > | 0.000 = | 0.878 > | 0.706 > | 0.828 > | 1.000 > |
| | 0.50 | 0.694 > | 0.399 > | 0.077 = | 0.301 > | 0.404 > | 0.230 > | 0.244 = | 0.909 > | 0.691 > | 0.721 > | 0.891 > |
| | 0.80 | 0.964 > | 0.474 > | 0.017 = | 0.261 > | 0.361 > | 0.320 > | 0.804 > | 0.817 > | 0.642 > | 0.528 > | 0.671 > |
| | 1.00 | 0.996 > | 0.794 > | 0.096 = | 0.366 > | 0.342 > | 0.407 > | 0.993 > | 0.830 > | 0.584 > | 0.332 > | 0.239 > |
| 1000 | 0.10 | 0.630 > | 0.423 > | 0.128 > | 0.378 > | 0.373 > | 0.213 > | 0.000 = | 0.889 > | 0.696 > | 0.813 > | 0.975 > |
| | 0.50 | 0.868 > | 0.348 > | 0.070 = | 0.316 > | 0.365 > | 0.331 > | 0.411 > | 0.917 > | 0.683 > | 0.641 > | 0.716 > |
| | 0.80 | 0.997 > | 0.555 > | 0.111 = | 0.388 > | 0.377 > | 0.432 > | 1.000 > | 0.764 > | 0.598 > | 0.420 > | 0.342 > |
| | 1.00 | 0.913 > | 0.702 > | 0.269 > | 0.346 > | 0.425 > | 0.478 > | 1.000 > | 0.769 > | 0.446 > | 0.179 > | 0.057 = |

## 13.5 Substrate Usage Cost Gap

The previous section allowed a small insight into the relative performance of the compared algorithms when solving VNMP-O. However, as we have outlined when defining the ranking procedure, by ranking we lose the information about the distance between solutions (in terms of cost). In this section, we will focus on the difference of the created solutions with respect to $C_u$. As a measure, we chose the substrate usage cost gap $C_u$-gap. For an algorithm A, it is calculated by dividing the difference between the $C_u$ of the result created by A and the best $C_u$ by the $C_u$ of A. If A's result is not valid, we assume a $C_u$-gap of 100%. A $C_u$-gap of 1% means that the best result is 1% cheaper than the result of the algorithm in question. Table 13.4 presents the average $C_u$-gap values achieved by the algorithms compared in this chapter.
It can be seen, that for instances below size 200, VNS-O is very close to the results achieved by ILP-O. For load 0.1, the gap is about 0.5%, for load 0.5 it is 2%, for 0.8 4%, and 7% for load 1. Also MA-O is close to the performance of ILP-O. It can be observed that the results of ILP-S are very often the most expensive. They are only exceeded for some instances by CH-O due to the 100% gap assumed when the final result is not valid. Even the results of CH-S, LS-S, and VND-S are in most cases not as expensive. Note that the presented gaps are in relation to the best found solution, which is usually created by ILP-O. For a significant fraction of instances, ILP-O cannot prove the optimality of the found solution and a gap of about 1% remains, as shown in Chapter 11. Therefore, to get an approximation of the gap between the heuristic solutions and the best solution value that might be obtainable, 1% can be added to the reported $C_u$-gaps.

## 13.6 Required Run-time

The last property of the presented algorithms we compare is their required run-time, which is presented in Table 13.5. For MA-O, GRASP-O, and VNS-O, the required run-time is constant since the algorithms were executed until the time-limit was reached. The applied time-limit of 1000 seconds for VND-based methods can be observed for the largest instances. Also note how the run-time, especially for full load and a size of 1000, exceeds the time-limit by a large margin. This is due to the fact that the time-limit is checked after a neighborhood has been searched. For the very largest instances, even LS-O had to be aborted due to the time-limit. That may be an indication that the employed neighborhoods grow too large when the instance size rises and some alternatives need to be developed. It is remarkable how close the run-time of ILP-S is to the heuristic alternatives.

## 13.7 Conclusion

The Internet has ossified. It cannot react to changing demands placed upon it by different applications. A way to get around this problem is to use virtual networks, which can be adapted in terms of structure, employed protocols, and available resources to perfectly fit to specific applications or application classes. These virtual networks need to be hosted in the available physical network. In this work, we have considered an abstraction of this problem, the Virtual Network Mapping Problem (VNMP), which is $\mathcal{NP}$-hard. More precisely, we considered two

Table 13.4: Average $C_u$-gap in percent achieved by the main algorithms presented in this work.

| Size | Load | CH-O | LS-O | VND-O | MA-O | GRASP-O | VNS-O | ILP-O | CH-S | LS-S | VND-S | ILP-S |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 0.10 | 19.4 > | 5.7 > | 5.0 > | 0.0 = | 2.0 > | 0.5 > | 0.0 = | 23.0 > | 17.9 > | 20.3 > | 48.2 > |
| | 0.50 | 29.5 > | 8.7 > | 7.7 > | 1.7 > | 4.8 > | 2.1 > | 0.0 = | 31.3 > | 24.2 > | 28.0 > | 38.2 > |
| | 0.80 | 32.5 > | 15.2 > | 14.1 > | 2.8 > | 6.3 > | 3.3 > | 0.0 = | 32.8 > | 27.5 > | 29.4 > | 34.4 > |
| | 1.00 | 68.3 > | 29.9 > | 20.9 > | 4.0 > | 8.1 > | 4.4 > | 0.0 = | 44.9 > | 30.8 > | 30.3 > | 29.3 > |
| 30 | 0.10 | 22.6 > | 3.8 > | 2.7 > | 0.2 = | 1.7 > | 0.4 > | 0.0 = | 26.9 > | 20.5 > | 25.5 > | 47.0 > |
| | 0.50 | 25.5 > | 7.4 > | 6.1 > | 1.6 > | 4.3 > | 1.9 > | 0.0 = | 31.9 > | 25.1 > | 28.5 > | 40.6 > |
| | 0.80 | 44.5 > | 11.9 > | 10.1 > | 4.0 > | 7.2 > | 3.5 > | 0.0 = | 34.5 > | 27.8 > | 30.8 > | 40.8 > |
| | 1.00 | 67.9 > | 19.6 > | 13.5 > | 6.7 > | 9.1 > | 6.0 > | 0.0 = | 38.3 > | 28.8 > | 30.6 > | 36.5 > |
| 50 | 0.10 | 19.8 > | 4.7 > | 2.7 > | 0.2 = | 2.4 > | 0.4 > | 0.0 = | 25.7 > | 20.3 > | 24.6 > | 45.2 > |
| | 0.50 | 30.3 > | 8.4 > | 6.3 > | 2.5 > | 4.7 > | 1.8 > | 0.0 = | 32.4 > | 27.2 > | 29.6 > | 45.4 > |
| | 0.80 | 42.0 > | 12.2 > | 9.4 > | 5.0 > | 7.6 > | 4.3 > | 0.0 = | 34.3 > | 29.7 > | 32.0 > | 44.4 > |
| | 1.00 | 83.8 > | 24.7 > | 20.3 > | 9.0 > | 12.7 > | 7.2 > | 0.0 = | 41.9 > | 36.0 > | 35.0 > | 42.1 > |
| 100 | 0.10 | 21.7 > | 6.0 > | 4.3 > | 1.2 > | 2.7 > | 0.8 > | 0.0 = | 28.9 > | 24.1 > | 26.7 > | 40.0 > |
| | 0.50 | 38.0 > | 8.4 > | 6.6 > | 5.9 > | 6.7 > | 4.5 > | 0.0 = | 34.5 > | 32.1 > | 33.4 > | 46.3 > |
| | 0.80 | 57.6 > | 14.1 > | 12.7 > | 8.6 > | 10.0 > | 7.4 > | 0.9 = | 40.3 > | 32.6 > | 33.8 > | 44.6 > |
| | 1.00 | 93.1 > | 25.5 > | 21.0 > | 11.9 > | 19.1 > | 10.6 > | 6.7 = | 42.3 > | 44.3 > | 36.1 > | 42.1 > |
| 200 | 0.10 | 19.9 > | 6.5 > | 3.7 > | 2.2 > | 2.9 > | 1.1 > | 0.0 = | 30.0 > | 26.0 > | 28.4 > | 41.1 > |
| | 0.50 | 49.1 > | 8.8 > | 5.6 > | 5.9 > | 7.2 > | 4.2 > | 5.0 = | 44.4 > | 34.1 > | 33.0 > | 45.8 > |
| | 0.80 | 86.7 > | 17.2 > | 7.8 = | 14.8 > | 13.3 > | 7.5 = | 17.7 = | 57.1 > | 39.8 > | 34.3 > | 45.2 > |
| | 1.00 | 100.0 > | 65.1 > | 19.0 = | 19.2 > | 15.5 > | 18.8 = | 30.0 = | 86.0 > | 57.3 > | 37.5 > | 43.6 > |
| 500 | 0.10 | 16.8 > | 5.8 > | 2.9 > | 4.4 > | 3.7 > | 2.5 > | 0.0 = | 29.5 > | 26.0 > | 28.2 > | 44.5 > |
| | 0.50 | 51.0 > | 16.4 > | 3.3 = | 12.3 > | 13.2 > | 8.8 > | 17.9 = | 50.9 > | 41.9 > | 34.2 > | 43.6 > |
| | 0.80 | 93.3 > | 38.0 > | 1.6 = | 18.4 > | 25.8 > | 25.1 > | 78.7 > | 81.6 > | 58.8 > | 33.3 > | 42.7 > |
| | 1.00 | 100.0 > | 100.0 > | 40.1 = | 74.4 > | 68.3 > | 71.4 > | 100.0 > | 100.0 > | 89.8 > | 61.0 > | 27.2 = |
| 1000 | 0.10 | 28.2 > | 7.7 > | 3.3 > | 7.1 > | 7.1 > | 4.7 > | 0.0 = | 41.9 > | 28.7 > | 30.7 > | 46.9 > |
| | 0.50 | 80.3 > | 23.2 > | 9.7 = | 20.1 > | 20.6 > | 17.5 > | 37.8 = | 73.3 > | 49.3 > | 40.9 > | 44.8 > |
| | 0.80 | 100.0 > | 73.9 > | 43.3 = | 61.5 > | 49.9 > | 61.8 > | 100.0 > | 98.1 > | 76.9 > | 59.7 = | 36.7 = |
| | 1.00 | 100.0 > | 100.0 > | 93.3 > | 93.3 > | 100.0 > | 93.6 > | 100.0 > | 100.0 > | 97.4 > | 94.7 > | 9.1 = |

Table 13.5: Average run-time in seconds required by the main algorithms presented in this work.

| Size | Load | CH-O | LS-O | VND-O | MA-O | GRASP-O | VNS-O | ILP-O | CH-S | LS-S | VND-S | ILP-S |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 0.10 | 0.0 = | 0.0 > | 0.0 > | 200.0 > | 200.0 > | 200.0 > | 0.1 > | 0.0 = | 0.0 > | 0.0 = | 0.0 > |
| | 0.50 | 0.0 = | 0.1 > | 0.3 > | 200.0 > | 200.0 > | 200.0 > | 0.8 > | 0.0 = | 0.0 > | 0.0 = | 0.1 > |
| | 0.80 | 0.0 = | 0.2 > | 0.5 > | 200.0 > | 200.0 > | 200.0 > | 10.7 > | 0.0 = | 0.0 > | 0.0 = | 0.3 > |
| | 1.00 | 0.0 = | 0.3 > | 0.6 > | 200.0 > | 200.0 > | 200.0 > | 85.8 > | 0.0 = | 0.0 > | 0.0 > | 0.4 > |
| 30 | 0.10 | 0.0 = | 0.0 > | 0.1 > | 200.0 > | 200.0 > | 200.0 > | 0.2 > | 0.0 > | 0.0 > | 0.0 = | 0.1 > |
| | 0.50 | 0.0 = | 0.3 > | 0.8 > | 200.0 > | 200.0 > | 200.0 > | 96.6 > | 0.0 = | 0.0 > | 0.0 = | 0.5 > |
| | 0.80 | 0.0 = | 0.6 > | 1.6 > | 200.0 > | 200.0 > | 200.0 > | 466.1 > | 0.0 = | 0.0 > | 0.0 = | 1.0 > |
| | 1.00 | 0.0 = | 0.9 > | 1.9 > | 200.0 > | 200.0 > | 200.0 > | 1899.2 > | 0.0 = | 0.1 > | 0.0 > | 1.8 > |
| 50 | 0.10 | 0.0 = | 0.1 > | 0.3 > | 200.0 > | 200.0 > | 200.0 > | 0.4 > | 0.0 = | 0.0 > | 0.0 = | 0.2 > |
| | 0.50 | 0.0 = | 1.1 > | 2.7 > | 200.0 > | 200.0 > | 200.0 > | 372.9 > | 0.0 = | 0.1 > | 0.0 = | 1.8 > |
| | 0.80 | 0.0 = | 2.2 > | 5.3 > | 200.0 > | 200.0 > | 200.0 > | 2576.0 > | 0.0 = | 0.1 > | 0.0 = | 3.7 > |
| | 1.00 | 0.0 = | 3.3 > | 6.7 > | 200.1 > | 200.0 > | 200.1 > | 4565.6 > | 0.0 = | 0.3 > | 0.6 > | 5.3 > |
| 100 | 0.10 | 0.0 > | 1.0 > | 2.5 > | 200.0 > | 200.0 > | 200.0 > | 4.1 > | 0.0 > | 0.0 > | 0.0 = | 1.3 > |
| | 0.50 | 0.1 = | 9.1 > | 19.3 > | 200.1 > | 200.1 > | 200.1 > | 3108.9 > | 0.1 = | 0.2 > | 0.9 = | 7.1 > |
| | 0.80 | 0.1 = | 18.2 > | 37.0 > | 200.2 > | 200.2 > | 200.2 > | 8933.8 > | 0.1 > | 0.3 > | 1.9 = | 13.3 > |
| | 1.00 | 0.1 = | 24.3 > | 47.9 > | 200.3 > | 200.2 > | 200.3 > | 9912.9 > | 0.1 > | 1.0 > | 4.3 > | 20.8 > |
| 200 | 0.10 | 0.0 > | 2.3 > | 7.7 > | 500.1 > | 500.0 > | 500.1 > | 48.5 > | 0.0 = | 0.1 > | 0.0 = | 3.4 > |
| | 0.50 | 0.1 = | 25.0 > | 63.6 > | 500.2 > | 500.2 > | 500.3 > | 6470.8 > | 0.1 = | 0.5 > | 1.8 = | 16.8 > |
| | 0.80 | 0.2 = | 55.0 > | 153.1 > | 500.4 > | 500.5 > | 500.3 > | 9768.4 > | 0.2 > | 1.6 > | 6.6 > | 32.8 > |
| | 1.00 | 0.2 = | 77.6 > | 209.5 > | 500.5 > | 500.6 > | 500.7 > | 10006.3 > | 0.3 > | 6.2 > | 51.5 > | 55.5 > |
| 500 | 0.10 | 0.1 > | 7.3 > | 41.7 > | 500.2 > | 500.2 > | 500.2 > | 536.9 > | 0.0 = | 0.2 > | 0.0 = | 24.3 > |
| | 0.50 | 0.3 > | 96.1 > | 366.0 > | 500.7 > | 500.5 > | 500.7 > | 8988.5 > | 0.3 = | 2.3 > | 5.7 = | 79.3 > |
| | 0.80 | 0.6 > | 245.4 > | 909.8 > | 501.2 > | 501.0 > | 501.4 > | 10030.9 > | 0.5 = | 19.3 > | 80.0 > | 161.9 > |
| | 1.00 | 0.8 > | 427.3 > | 1013.2 > | 501.5 > | 501.3 > | 502.1 > | 10038.8 > | 0.8 = | 80.8 > | 579.8 > | 432.9 > |
| 1000 | 0.10 | 0.3 > | 25.1 > | 216.6 > | 500.5 > | 500.3 > | 500.4 > | 2264.7 > | 0.1 = | 0.5 > | 1.8 > | 42.4 > |
| | 0.50 | 0.8 > | 328.6 > | 1011.3 > | 502.3 > | 501.6 > | 501.8 > | 10075.2 > | 0.7 = | 12.2 > | 129.5 > | 208.2 > |
| | 0.80 | 1.5 > | 860.3 > | 1064.9 > | 502.7 > | 502.8 > | 503.8 > | 10124.3 > | 1.3 = | 109.3 > | 717.5 > | 579.0 > |
| | 1.00 | 2.1 > | 1009.4 > | 1013.9 > | 503.9 > | 504.8 > | 506.3 > | 10019.6 > | 1.9 = | 356.4 > | 1076.5 > | 1865.5 > |

variants, VNMP-S, where just a valid solution (i.e., an embedding of the virtual networks into the physical network) needs to be found, and VNMP-O, where we want to find a valid and cheap embedding.

As a first approach for solving the VNMP, we presented Construction Heuristics. Then we defined neighborhood structures to improve solutions found by the Construction Heuristics to create Local Search algorithms. We studied how the different neighborhood structures can be used to form Variable Neighborhood Descent algorithms. Based on the experience of solving the VNMP with those methods, we developed Greedy Randomized Adaptive Search Procedures, Memetic Algorithms and Variable Neighborhood Search approaches to further increase performance. Having covered some of the most important meta-heuristics for combinatorial optimization problems, we focused on exact methods for solving the VNMP. As preparation for that, we developed a preprocessing algorithm which makes use of the properties of telecommunication networks to extract useful knowledge from a VNMP instance, such as which nodes have to be used in any case. The preprocessing methods proved to be essential when we applied Constraint Programming and Integer Linear Programming to the VNMP.

As for the performance of the different developed algorithms, some of course proved to be better suited than others. The Constraint Programming and Column Generation approaches we had to reject out of hand. Constraint Programming showed poor performance when trying to find valid VNMP solutions and very poor performance when solving VNMP-O. The main problems where high memory consumption and insufficient detection of inconsistent partial assignments. The Column Generation approach was dismissed, since it offered no advantages over a compact ILP formulation for the VNMP while substantially increasing the required run-time.

The other algorithms were more successful. Due to the performance of ILP-S, VNMP-S can be considered solved, at least for the instance types and sizes used in this work. ILP-S was also surprising because of its extremely low run-time requirements for an exact method. Just for one single instance from the VNMP instance set ILP-S failed to find a valid solution due to memory requirements. In fact, without the memory limit, ILP-S is able to solve this instance within the time-limit. Among the tested heuristic methods, VND-S showed the best performance for solving VNMP-S. This algorithm is able to solve all but the instances from the largest two size classes. In addition, the results it creates are very close to validity.

When considering the VNMP-O, the overall best performing algorithm in terms of solution quality is ILP-O. We could demonstrate the importance of our developed preprocessing algorithm, which reduces the required run-time and memory for solving VNMP instances significantly. However, the dominance of ILP-O is not as complete as ILP-S's on VNMP-S. For the largest instances, VND-O produces the best results. For the most challenging instances, ILP-S is the best algorithms since it is the only one capable of finding valid solutions. Even though ILP-O produces the best results, the heuristic methods, especially MA-O and VNS-O are very close. On average, the solutions found by ILP-O cost about 5% less than those of the best heuristic methods.

## 13.8 Future Work

As the previous chapters already contained remarks on promising research avenues for each of the investigated subtopics, we will focus here on those ideas which are supported by data from this chapter. At the end, we present directions for future work on Virtual Network Mapping.
One major area for future improvement is the performance of the heuristic approaches for the larger instance sizes and especially in cases of high load. We have seen that the load of an instance has a far more pronounced influence on the results achieved by a heuristic than the instance size. The first step for doing this is to incorporate the information derived by preprocessing, similar to what we did for the exact methods. Especially the defined neighborhood structures can be refined. For instance, it does not make sense to remove the implementation of a virtual arc from a substrate node if we know that the virtual arc has to use that substrate node. We have also seen an indication that the neighborhoods become too large when the instance size increases. As the number of virtual arcs and nodes stays roughly the same for instances larger than 100 nodes, the problem lies in the much increased mapping possibilities. Methods that do not slow to a crawl when confronted with a high number of mapping possibilities need to be developed.
We have shown that ILP-S is excellent for solving the VNMP-S, but the created solutions leave something to be desired in terms of $C_u$. Combined with one of the presented heuristic methods, we could create a method that can basically guarantee that it will find a valid solution to a VNMP instance (if it exists) and that the solution will be close to optimal.
If the utilization of an ILP approach is not desirable, another approach to increase the capability of the heuristics to find valid solutions could be to solve VNMP instances incrementally. That means, we first solve the instance with load 0.1 and once a valid solution has been found, we add further virtual networks until the target load is reached.
In this work, we have considered the offline variant of the VNMP and built a solid foundation for future work on this topic. The following lists what we believe are core research directions for the VNMP:

**Online VNMP:** As we have outlined in Chapter 4, most work on the VNMP considers the online variant of VNMP in the sense that virtual networks arrive and depart. The performance of the algorithms presented in this work needs to be evaluated in such a setting. The required run-time will be of prime importance as it determines how fast virtual networks can be accepted. However, arriving and departing virtual networks are only a part of the possible dynamic behaviours of the VNMP. Virtual networks may change their requirements, for instance if more people connect to them. At the very least, bandwidth requirements will increase and the behaviour of the presented algorithms needs to be studied when such small scale changes occur. The substrate network is also dynamic and can change in structure, for example if leasing resources somewhere else in the physical network gets cheaper, or if hardware fails. One interesting question would be how long it takes to modify an existing solution, such that it does not use a particular substrate node or arc any longer. Models for the cost of changing a virtual network configuration need to be studied, together with their implications on the performance and complexity of algorithms solving the VNMP.

**Distributed VNMP:** Current models of the VNMP assume global knowledge of the substrate network. However, this is not possible for networks on the scale of planets, so algorithms that work in a distributed fashion need to be studied further. The methods presented in this work could be used as efficient solvers for parts of the network where global knowledge is feasible.

**Stochastic VNMP:** We have already stated that the whole system of network virtualization is very dynamic. This can be handled in two ways, we could solve the problem again when something changes, or we could add the possibility of change to the model of the VNMP. The first approach is the online VNMP. Here, we deal with the second one. As an example, instead of assuming that a virtual network has a constant requirement for bandwidth, we could model it using a suitable statistical distribution depending on the application. Instead of a solution that does not exceed the available resource capacities, we would search for a solution with a probability of capacity violations below a certain threshold.

**Nonlinear VNMP:** Models of the VNMP usually do not consider non-linear behaviour in the network. As an example, the observed delay of a link in the physical network heavily depends on the amount of data that is transferred across this link. This dependency is highly non-linear, especially when a link's capacity limit is approached. Furthermore, virtual nodes are not free of overhead and mapping multiple virtual nodes on a substrate node should at least cause some additional CPU load. Algorithms able to cope with non-linear behaviour, or which are at least able to approximate it, need to be studied.

**Split-Flow VNMP:** A major assumption of the definition of VNMP in this work is, that a virtual connection can only be implemented by using a single path in the substrate network. Doing otherwise would cause erratic behaviour with respect to transmission delays on the virtual link. However, a lot of applications are not delay sensitive or can deal with high delay variances. In addition, allowing multiple paths simplifies the VNMP. There are some interesting possibilities that arise when multiple paths are allowed. For instance, virtual links can be implemented that have more capacity than any single connection in the substrate network. Depending on the application, it might be possible to split up the workload of a single virtual node and spread it across multiple substrate nodes.

**Physical VNMP:** At the moment, the behaviour of algorithms for solving the VNMP is only evaluated theoretically. There are no experiments in this work where the developed algorithms are used to perform Virtual Network Mapping in a real network, or at least a simulated network. Such experiments would be highly informative, as they can give hints which aspects of the VNMP model should be refined to get it close enough to real world behaviour. For instance, it could turn out that the behaviour of a physical network with non-linearities is not so far removed from the assumed linear behaviour as to make the investigation of algorithms capable of dealing with non-linearities absolutely essential.

# Bibliography


[1] E. H. L. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization*. Princeton University Press, 2003.

[2] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen. A Survey of Very Large-Scale Neighborhood Search Techniques. *Discrete Applied Mathematics*, 123(1):75–102, 2002.

[3] O. Alonso-Garrido, S. Salcedo-Sanz, L. E. Agustín-Blas, E. G. Ortiz-García, A. M. Pérez-Bellido, and J. A. Portilla-Figueras. A Hybrid Grouping Genetic Algorithm for the Multiple-Type Access Node Location Problem. In E. Corchado and H. Yin, editors, *Intelligent Data Engineering and Automated Learning - IDEAL 2009*, volume 5788 of *Lecture Notes in Computer Science*, pages 376–383. Springer Berlin Heidelberg, 2009.

[4] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in Linear Time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.

[5] D. Andersen. Theoretical Approaches To Node Assignment. `http://www.cs.cmu.edu/~dga/papers/andersen-assign.ps`, December 2002. Unpublished Manuscript.

[6] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse Through Virtualization. *Computer*, 38(4):34–41, 2005.

[7] K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[8] E. Balas, S. Ceria, and G. Cornuéjols. A Lift-and-Project Cutting Plane Algorithm for Mixed 0 – 1 Programs. *Mathematical Programming*, 58(1-3):295–324, 1993.

[9] R. Barták. Theory and Practice of Constraint Propagation. In *Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control*, volume 50, 2001.

[10] I. L. Bedhiaf, R. B. Ali, and O. Cherkaoui. On the Problem of Mapping Virtual Machines to Physical Machines for Delay Sensitive Services. In *IEEE Global Communications Conference (GLOBECOM 2012)*, pages 2628–2633. IEEE, 2012.

[11] A. Ben-Tal and A. Nemirovski. Robust Optimization – Methodology and Applications. *Mathematical Programming*, 92(3):453–480, 2002.

[12] A. Berl, A. Fischer, and H. de Meer. Virtualisierung im Future Internet. *Informatik-Spektrum*, 33:186–194, 2010.

[13] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

[14] H. Beyer and B. Sendhoff. Robust Optimization – A Comprehensive Survey. *Computer Methods in Applied Mechanics and Engineering*, 196(33–34):3190–3218, 2007.

[15] C. Blum and A. Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

[16] C. Blum and A. Roli. Hybrid Metaheuristics: An Introduction. In C. Blum, M. Aguilera, A. Roli, and M. Sampels, editors, *Hybrid Metaheuristics*, volume 114 of *Studies in Computational Intelligence*, pages 1–30. Springer Berlin Heidelberg, 2008.

[17] Boost.org. boost 1.54. `http://www.boost.org/`.

[18] Clark D. Braden, R. and S. Shenker. Integrated Services in the Internet Architecture: An Overview. *IETF, RFC 1633*, 1994.

[19] J. Branke and H. Schmeck. Designing Evolutionary Algorithms for Dynamic Optimization Problems. In *Advances in Evolutionary Computing*, pages 239–262. Springer, 2003.

[20] E. C. Brown and R. T. Sumichrast. Impact of the Replacement Heuristic in a Grouping Genetic Algorithm. *Computers & Operations Research*, 30(11):1575–1593, 2003.

[21] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. Tarjan, and J. Westbrook. Linear-Time Algorithms for Dominators and Other Path-Evaluation Problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008.

[22] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time Pointer-Machine Algorithms for Least Common Ancestors, MST Verification, and Dominators. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 279–288. ACM, 1998.

[23] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Corrigendum: A New, Simpler Linear-Time Dominators Algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):383–387, 2005.

[24] M. G. Bulmer. *Principles of Statistics*. Dover Publications, 1979.

[25] H. Burch and B. Cheswick. Mapping the Internet. *Computer*, 32(4):97–98, 102, 1999.

[26] M. Carlson, W. Weiss, S. Blake, Z. Wang, D. Black, and E. Davies. An Architecture for Differentiated Services. *IETF, RFC 2475*, 1998.

[27] P. Cholda, A. Mykkeltveit, B. E. Helvik, O. J. Wittner, and A. Jajszczyk. A Survey of Resilience Differentiation Frameworks in Communication Networks. *Communications Surveys & Tutorials, IEEE*, 9(4):32–55, 2007.

[28] N. M. M. K. Chowdhury and R. Boutaba. Network Virtualization: State of the Art and Research Challenges. *Communications Magazine, IEEE*, 47(7):20–26, 2009.

[29] N. M. M. K. Chowdhury and R. Boutaba. A Survey of Network Virtualization. *Computer Networks*, 54(5):862–876, 2010.

[30] N. M. M. K. Chowdhury, M. R. Rahman, and R. Boutaba. Virtual Network Embedding with Coordinated Node and Link Mapping. In *28th IEEE International Conference on Computer Communications (INFOCOM 2009)*, pages 783–791. IEEE, 2009.

[31] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33:3–12, 2003.

[32] Cisco Systems, Inc. Logical Routers Commands on Cisco IOS XR Software, 2013. `www.cisco.com/en/US/docs/ios_xr_sw/iosxr_r3.2/interfaces/command/reference/hr32lr.html`.

[33] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[34] K. D. Cooper, T. J. Harvey, and K. Kennedy. A Simple, Fast Dominance Algorithm. *Software Practice & Experience*, 4:1–10, 2001.

[35] S. Crocker. Protocol Notes. *Network Working Group, RFC 36 (updated by RFC 44, RFC 39)*, 1970.

[36] E. Danna. Performance Variability in Mixed Integer Programming. In *Workshop on Mixed Integer Programming*, Columbia University, New York, 2008. `http://coral.ie.lehigh.edu/mip-2008/abstracts.html#Danna`.

[37] G. B. Dantzig. Maximization of a Linear Function of Variables Subject to Linear Inequalities. In T. C. Koppmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. Wiley, 1951.

[38] G. B. Dantzig, R. Fulkerson, and S. Johnson. Solution of a Large-Scale Traveling-Salesman Problem. *Journal of the Operations Research Society of America*, pages 393–410, 1954.

[39] G. B. Dantzig and P. Wolfe. Decomposition Principle for Linear Programs. *Operations Research*, 8(1):101–111, 1960.

[40] C. Darwin. *On the Origin of Species by Means of Natural Selection of the Preservation of Favored Races in the Struggle for Life*. Murray, 1859.

[41] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[42] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification, RFC 2460, 1998. `http://tools.ietf.org/html/rfc2460`.

[43] C. Demetrescu and G. F. Italiano. A New Approach to Dynamic All Pairs Shortest Paths. *Journal of the ACM*, 51(6):968–992, 2004.

[44] G. Desaulniers, J. Desrosiers, and M. M. Solomon. *Column Generation*, volume 5. Springer, 2005.

[45] O. Du Merle, D. Villeneuve, J. Desrosiers, and P. Hansen. Stabilized Column Generation. *Discrete Mathematics*, 194(1):229–237, 1999.

[46] Ronald R. Escobar, G. Constraints on Set Variables for Constraint-based Local Search. Master's thesis, Uppsala University, Department of Information Technology, 2011.

[47] C. B. Evelyn, T. R. Cliff, and E. C. Arthur. A Grouping Genetic Algorithm for the Multiple Traveling Salesperson Problem. *International Journal of Information Technology & Decision Making*, 6(02):333–347, 2007.

[48] I. Fajjari, N. Aitsaadi, G. Pujolle, and H. Zimmermann. Adaptive-VNE: A Flexible Resource Allocation for Virtual Network Embedding Algorithm. In *IEEE Global Communications Conference (GLOBECOM 2012)*, pages 2640–2646. IEEE, 2012.

[49] E. Falkenauer and A. Delchambre. A Genetic Algorithm for Bin Packing and Line Balancing. In *IEEE International Conference on Robotics and Automation*, pages 1186–1192. IEEE, 1992.

[50] N. Feamster, L. Gao, and J. Rexford. How to Lease the Internet in Your Spare Time. *ACM SIGCOMM Computer Communication Review*, 37(1):61–64, 2007.

[51] H. Feltl and G. R. Raidl. An Improved Hybrid Genetic Algorithm for the Generalized Assignment Problem. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 990–995. ACM, 2004.

[52] T. A. Feo and M. G. C. Resende. A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem. *Operations Research Letters*, 8(2):67–71, 1989.

[53] T. A. Feo and M. G. C. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[54] P. Festa and M. G. C. Resende. An Annotated Bibliography of GRASP – Part I: Algorithms. *International Transactions in Operational Research*, 16(1):1–24, 2009.

[55] P. Festa and M. G. C. Resende. Hybrid GRASP Heuristics. *Foundations of Computational Intelligence*, 3:75–100, 2009.

[56] R. W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345–345, 1962.

[57] S. Fortune, Hopcroft J., and J. Wyllie. The Directed Subgraph Homeomorphism Problem. *Theoretical Computer Science*, 10(2):111–121, 1980.

[58] B. Fortz and M. Thorup. Internet Traffic Engineering by Optimizing OSPF Weights. In *19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*, pages 519–528, Tel-Aviv, Israel, 2000. IEEE.

[59] Eclipse Foundation. Eclipse 4.3 "Kepler". `http://www.eclipse.org/`.

[60] M. R. Garey and D. S. Johnson. *Computers and Intractability*, volume 174. Freeman New York, 1979.

[61] M. R. Gary and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness, 1979.

[62] GENI.net. Global Environment for Network Innovations. `http://www.geni.net`, 2012.

[63] L. Georgiadis and R. E. Tarjan. Finding Dominators Revisited. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 869–878. Society for Industrial and Applied Mathematics, 2004.

[64] P. C. Gilmore and R. E. Gomory. A Linear Programming Approach to the Cutting-Stock Problem. *Operations Research*, 9(6):849–859, 1961.

[65] GNU Compiler Collection. The GNU Compiler Collection, gcc 4.7.3. `http://gcc.gnu.org/`.

[66] R. Gold, P. Gunningberg, and C. Tschudin. A Virtualized Link Layer with Support for Indirection. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, pages 28–34. ACM, 2004.

[67] O. Goldreich. *P, NP, and NP-Completeness: The Basics of Computational Complexity*. Cambridge University Press, Cambridge, 2010.

[68] R. E. Gomory. Outline of an Algorithm for Integer Solutions to Linear Programs. *Bulletin of the American Mathematical Society*, 64(5):275–278, 1958.

[69] L. Gouveia, A. Paias, and D. Sharma. Modeling and Solving the Rooted Distance-Constrained Minimum Spanning Tree Problem. *Computers & Operations Research*, 35(2):600–613, 2008.

[70] R. Govindan and H. Tangmunarunkit. Heuristics for Internet Map Discovery. In *Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*, volume 3, pages 1371–1380. IEEE Computer Society Press, 2000.

[71] A. Gupta, J. Kleinberg, A. Kumar, R. Rastogi, and B. Yener. Provisioning a Virtual Private Network: A Network Design Problem for Multi-Commodity Flow. In *STOC '01*, pages 389–398, 2001.

[72] M. Handley. Why the Internet Only Just Works. *BT Technology Journal*, 24:119–129, 2006.

[73] P. Hansen and N. Mladenović. An Introduction to Variable Neighborhood Search. In S. Voß, S. Martello, I. H. Osman, and C. Roucairol, editors, *MIC-97: Meta-Heuristics International Conference*, pages 433–458. Kluwer Academic Publishers, 1999.

[74] P. Hansen and N. Mladenović. Variable Neighborhood Search: Principles and Applications. *European Journal of Operational Research*, 130(3):449–467, 2001.

[75] P. Hansen, N. Mladenović, J. Brimberg, and J. A. Moreno Pérez. Variable Neighborhood Search. In M. Gendreau and J. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 61–86. Springer, 2010.

[76] P. Hansen, N. Mladenović, and J. A. Moreno Pérez. Variable Neighbourhood Search: Methods and Applications. *4OR*, 6:319–360, 2008.

[77] D. Harel. A Linear Algorithm for Finding Dominators in Flow Graphs and Related Problems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 185–194. ACM, 1985.

[78] M. Hartmann, D. Hock, M. Menth, and C. Schwartz. Objective Functions for Optimization of Resilient and Non-Resilient IP Routing. In *7th International Workshop on Design of Reliable Communication Networks (DRCN 2009)*, pages 289–296. IEEE, 2009.

[79] D. Hock, M. Menth, M. Hartmann, C. Schwartz, and D. Stezenbach. ResiLyzer: A Tool for Resilience Analysis in Packet-Switched Communication Networks. *Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance*, pages 302–306, 2010.

[80] J. H. Holland. Genetic Algorithms. *Scientific American*, 267(1):66–72, 1992.

[81] Steve Holzner. *Eclipse*. O'Reilly, 2004.

[82] I. Houidi, W. Louati, and D. Zeghlache. A Distributed Virtual Network Mapping Algorithm. In *IEEE International Conference on Communications (ICC '08)*, pages 5634–5640. IEEE, 2008.

[83] B. Hu, M. Leitner, and G. R. Raidl. The Generalized Minimum Edge Biconnected Network Problem: Efficient Neighborhood Structures for Variable Neighborhood Search. *Networks*, 55(3):257–275, 2010.

[84] G. Huston. IPv4 Address Report, daily generated, 2013. http://www.potaroo.net/tools/ipv4/index.html.

[85] IBM ILOG. CPLEX 12.5. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.

[86] R. Ihaka and R. Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996. http://www.R-project.org/.

[87] J. Inführ and G. R. Raidl. The Virtual Network Mapping Problem Benchmark Set and Achieved Solutions by Heuristic and Exact Methods. https://www.ads.tuwien.ac.at/projects/optFI/.

[88] J. Inführ and G. R. Raidl. Introducing the Virtual Network Mapping Problem with Delay, Routing and Location Constraints. In J. Pahl, T. Reiners, and S. Voß, editors, *Network Optimization: 5th International Conference (INOC 2011)*, volume 6701 of *Lecture Notes in Computer Science*, pages 105–117, Hamburg, Germany, 2011. Springer.

[89] J. Inführ and G. R. Raidl. A Memetic Algorithm for the Virtual Network Mapping Problem. In H. C. Lau, P. Van Hentenryck, and G. R. Raidl, editors, *Proceedings of the 10th Metaheuristics International Conference*, pages 28–1–28–10, Singapore, 2013.

[90] J. Inführ and G. R. Raidl. GRASP and Variable Neighborhood Search for the Virtual Network Mapping Problem. In M. J. Blesa et al., editors, *Hybrid Metaheuristics, 8th International Workshop (HM 2013)*, volume 7919 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2013.

[91] J. Inführ and G. R. Raidl. Solving the Virtual Network Mapping Problem with Construction Heuristics, Local Search, and Variable Neighborhood Descent. In M. Middendorf and C. Blum, editors, *Evolutionary Computation in Combinatorial Optimisation – 13th European Conference, EvoCOP 2013*, volume 7832 of *Lecture Notes in Computer Science*, pages 250–261. Springer, 2013.

[92] J. Inführ, D. Stezenbach, M. Hartmann, K. Tutschku, and G. R. Raidl. Using Optimized Virtual Network Embedding for Network Dimensioning. In *Proceedings of Networked Systems 2013*, pages 118–125, Stuttgart, Germany, 2013. IEEE.

[93] P. Z. Ingerman. Algorithm 141: Path Matrix. *Communications of the ACM*, 5(11):556–556, 1962.

[94] G. F. Italiano, L. Laura, and F. Santaroni. Finding Strong Bridges and Strong Articulation Points in Linear Time. In W. Wu and O. Daescu, editors, *Combinatorial Optimization and Applications*, volume 6508 of *Lecture Notes in Computer Science*, pages 157–169. Springer Berlin Heidelberg, 2010.

[95] G. F. Italiano, L. Laura, and F. Santaroni. Finding Strong Bridges and Strong Articulation Points in Linear Time. *Theoretical Computer Science*, 447:74–84, 2012.

[96] V. Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988.

[97] D. B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

[98] M. Jovanović, F. Annexstein, and K. Berman. Modeling Peer-to-Peer Network Topologies Through Small-World Models and Power Laws. In *IX Telecommunications Forum, TELFOR*, pages 1–4, 2001.

[99] Justin.tv, Inc. All about Twitch, 2013. `http://de.twitch.tv/p/about`.

[100] G. K. Kanji. *100 Statistical Tests*. Sage, 2006.

[101] G. Karakostas. Faster Approximation Schemes for Fractional Multicommodity Flow Problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 166–173, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[102] B. Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost*. Pearson Education, 2005.

[103] N. Karmakar. A New Polynomial-Time Algorithm for Linear Programming. *Combinatorica*, 4:373–395, 1984.

[104] L. Khachiyan. A Polynomial Algorithm in Linear Programming (english translation). *Soviet Mathematics Doklady*, 20:191–194, 1979.

[105] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[106] S. Knight, H. X. Nguyen, N. Falkner, and M. Roughan. Realistic Network Topology Construction and Emulation from Multiple Data Sources. Technical report, The University of Adelaide, 2012. `http://www.topology-zoo.org/publications/eu_nren_tech/eu_nren_tech.html`.

[107] D. E. Knuth. The Stanford GraphBase: A Platform for Combinatorial Algorithms. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 41–43, Philadelphia, PA, USA, 1993. Society for Industrial and Applied Mathematics.

[108] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison-Wesley Professional, 1st edition, 2009.

[109] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, et al. MIPLIB 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[110] B. Korte and J. Vygen. *Combinatorial Optimization*, volume 21 of *Algorithms and Combinatorics*. Springer, 5th edition, 2012.

[111] M. Kudlick. Host Names On-Line. *IETF, RFC 608*, 1974.

[112] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[113] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, and S. Dizdarevic. Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artificial Intelligence Review*, 13:129–170, 1999.

[114] M. Leitner. Solving Two Generalized Network Design Problems with Exact and Heuristic Methods. Master's thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, May 2006.

[115] T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.

[116] K. Lougheed and Y. Rekhter. Border Gateway Protocol BGP. *RFC 1105*, 1989.

[117] J. Lu and J. Turner. Efficient Mapping of Virtual Networks Onto a Shared Substrate. Technical report, Washington University in St. Louis, 2006.

[118] M. E. Lübbecke and J. Desrosiers. Selected Topics in Column Generation. *Operations Research*, 53(6):1007–1023, 2005.

[119] I. J. Lustig and J. Puget. Program Does Not Equal Program: Constraint Programming and its Relationship to Mathematical Programming. *Interfaces*, 31(6):29–53, 2001.

[120] D. Magoni. Nem: A Software for Network Topology Analysis and Modeling. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '02, pages 364–371, Washington, DC, USA, 2002. IEEE Computer Society.

[121] D. Magoni. Network Topology Analysis and Internet Modelling with Nem. *International Journal of Computers and Applications*, 27(4):252–259, 2005.

[122] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfiguration. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 3–16. ACM, 2002.

[123] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[124] J. McQuillan, I. Richer, and E. Rosen. The New Routing Algorithm for the ARPANET. *IEEE Transactions on Communications*, 28(5):711–719, 1980.

[125] K. Mehlhorn and M. Ziegelmann. Resource Constrained Shortest Paths. In Mike S. Paterson, editor, *Algorithms - ESA 2000*, volume 1879 of *Lecture Notes in Computer Science*, pages 326–337. Springer Berlin Heidelberg, 2000.

[126] G. Mendel. Versuche über Pflanzen-Hybriden (Experiments on Plant Hybridization). In *Verhandlungen des naturforschenden Vereins Brünn (Proceedings of the Natural History Society of Brünn)*, volume 4, pages 3–47, 1866.

[127] M. Menth, M. Duelli, R. Martin, and J. Milbrandt. Resilience Analysis of Packet-Switched Communication Networks. *IEEE ACM Transactions on Networking*, 17(6):1950–1963, 2009.

[128] M. Menth, R. Martin, and J. Charzinski. Capacity Overprovisioning for Networks with Resilience Requirements. *ACM SIGCOMM Computer Communication Review*, 36(4):87–98, 2006.

[129] Z. Michalewicz. *Genetic Algoritms + Data Structures = Evolution Programs*. Springer, Berlin, 1994.

[130] P. Moscato and C. Cotta. A Modern Introduction to Memetic Algorithms. In M. Gendreau and J. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 141–183. Springer, 2010.

[131] P. Moscato and M. G. Norman. A Memetic Approach for the Traveling Salesman Problem Implementation of a Computational Ecology for Combinatorial Optimization on Message-Passing Systems. In *Proceedings of International Conference on Parallel Computing and Transputer Applications*, volume 1, pages 177–186. IOS Press, 1992.

[132] National Research Council. *Looking Over the Fence at Networks*. National Academy Press, Washington D.C., 2001.

[133] S. Orlowski, R. Wessäly, M. Pióro, and A. Tomaszewski. SNDlib 1.0 – Survivable Network Design Library. *Networks*, 55(3):276–286, 2010.

[134] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998.

[135] K. Papagiannaki, S. Moon, C. Fraleigh, P. Thiran, F. Tobagi, and C. Diot. Analysis of Measured Single-Hop Delay from an Operational Backbone Network. In *Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, volume 2, pages 535–544, 2002.

[136] I. Park and T. Kim. Multiplier-Less and Table-Less Linear Approximation for Square and Square-Root. In *Proceedings of the 2009 IEEE International Conference on Computer Design*, pages 378–383, Piscataway, NJ, USA, 2009. IEEE Press.

[137] Charles E Perkins. Mobile IP. *IEEE Communications Magazine*, 35(5):84–99, 1997.

[138] CDT Project. C++ Development Tooling - CDT 8.2, 2013. `http://eclipse.org/cdt/`.

[139] S. Qing, Q. Qi, J. Wang, T. Xu, and J. Liao. Topology-Aware Virtual Network Embedding through Bayesian Network Analysis. In *IEEE Global Communications Conference (GLOBECOM 2012)*, pages 2621–2627. IEEE, 2012.

[140] L. Quesada. *Solving Constrained Graph Problems Using Reachability Constraints Based on Transitive Closure and Dominators*. PhD thesis, Université Catholique de Louvain, 2006.

[141] L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using Dominators for Solving Constrained Path Problems. In *Practical Aspects of Declarative Languages*, pages 73–87. Springer, 2006.

[142] N. Radcliffe and P. Surry. Formal Memetic Algorithms. *Evolutionary Computing*, pages 1–16, 1994.

[143] G. R. Raidl. A Unified View on Hybrid Metaheuristics. In F. Almeida, M. J. Blesa Aguilera, C. Blum, J. M. Moreno Vega, M. Pérez Pérez, A. Roli, and M. Sampels, editors, *Hybrid Metaheuristics*, volume 4030 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2006.

[144] K. K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. *IETF, RFC 3168*, 2001.

[145] A. Razzaq and M. S. Rathore. An Approach Towards Resource Efficient Virtual Network Embedding. In *Proceedings of the 2010 2nd International Conference on Evolving Internet*, INTERNET '10, pages 68–73. IEEE Computer Society, 2010.

[146] C. R. Reeves. Genetic Algorithms. In M. Gendreau and J. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 109–139. Springer, 2010.

[147] M. G. C. Resende and C. Ribeiro. Greedy Randomized Adaptive Search Procedures. *Handbook of Metaheuristics*, pages 219–249, 2003.

[148] R. Ricci, C. Alfeld, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. *Special Interest Group on Data Communication Computer Communication Review*, 33(2):65–81, 2003.

[149] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. *arXiv preprint cs/0209028*, 2002.

[150] J. P. Romano. *Testing Statistical Hypotheses*. Springer, 2005.

[151] E. Rosen. Exterior Gateway Protocol (EGP). *RFC 827*, 1982.

[152] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[153] A. Schrijver. Finding $k$ Disjoint Paths in a Directed Planar Graph. *SIAM Journal on Computing*, 23(4):780–788, 1994.

[154] G. Schrimpf, J. Schneider, H. Stamm-Wilbrandt, and G. Dueck. Record Breaking Optimization Results using the Ruin and Recreate Principle. *Journal of Computational Physics*, 159(2):139–171, 2000.

[155] C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling and Programming with Gecode, 2013. `http://www.gecode.org/`.

[156] D. Schwerdel, D. Günther, R. Henjes, B. Reuther, and P. Müller. German-Lab Experimental Facility. In A. Berre, A. Gómez-Pérez, K. Tutschku, and D. Fensel, editors, *Future Internet - FIS 2010*, volume 6369 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2010.

[157] S. Shenker. Fundamental Design Issues for the Future Internet. *IEEE Journal on Selected Areas in Communications*, 13(7):1176–1188, 1995.

[158] J. G. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Pearson Education, 2001.

[159] M. Sipser. The History and Status of the P versus NP Question. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, pages 603–618. ACM, 1992.

[160] M. Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[161] S. N. Sivanandam and S. N. Deepa. *Introduction to Genetic Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[162] Skype. `http://www.skype.com`.

[163] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, pages 133–145, New York, NY, USA, 2002. ACM.

[164] W. Szeto, Y. Iraqi, and R. Boutaba. A Multi-Commodity Flow based Approach to Virtual Network Resource Allocation. In *Global Telecommunications Conference (GLOBECOM 2003)*, volume 6, pages 3004–3008. IEEE, 2003.

[165] E. G. Talbi. A Taxonomy of Hybrid Metaheuristics. *Journal of Heuristics*, 8(5):541–564, 2002.

[166] R. Tarjan. Depth-first Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[167] Gecode Team. Gecode: Generic Constraint Development Environment, Version 4.1.0, 2013. `http://www.gecode.org/`.

[168] J. Touch, Y. Wang, L. Eggert, and G. Finn. A Virtual Internet Architecture. *ISI Technical Report ISI-TR-2003-570*, 2003.

[169] M. F. Triola, W. M. Goodman, G. LaBute, R. Law, and L. MacKay. *Elementary Statistics*. Pearson/Addison-Wesley, 2006.

[170] J. S. Turner and D. E. Taylor. Diversifying the Internet. In *IEEE Global Telecommunications Conference (GLOBECOM 2005)*, volume 2, pages 755–760. IEEE, 2005.

[171] K. Tutschku. Towards the Future Internet: Virtual Networks for Convergent Services. *e & i Elektrotechnik und Informationstechnik*, 126(7-8):250–259, 2009.

[172] K. Tutschku, P. Tran-Gia, and F. Andersen. Trends in Network and Service Operation for the Emerging Future Internet. *AEU-International Journal of Electronics and Communications*, 62(9):705–714, 2008.

[173] Ustream, Inc. Company Info, 2013. `http://www.ustream.tv/our-company`.

[174] F. Vanderbeck. Implementing Mixed Integer Column Generation. In G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors, *Column Generation*, pages 331–358. Springer US, 2005.

[175] Z. Wang, Y. Han, T. Lin, H. Tang, and S. Ci. Virtual Network Embedding by Exploiting Topological Information. In *IEEE Global Communications Conference (GLOBECOM 2012)*, pages 2603–2608. IEEE, 2012.

[176] D. B. West. *Introduction to Graph Theory*, volume 2. Prentice Hall, 2001.

[177] J. Whiteaker, F. Schneider, and R. Teixeira. Explaining Packet Delays Under Virtualization. *ACM SIGCOMM Computer Communication Review*, 41(1):38–44, 2011.

[178] D. Whitley. A Genetic Algorithm Tutorial. *Statistics and Computing*, 4(2):65–85, 1994.

[179] F. Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[180] W. Yeow, C. Westphal, and U. Kozat. Designing and Embedding Reliable Virtual Infrastructures. In *Proceedings of the Second ACM Special Interest Group on Data Communication Workshop on Virtualized Infrastructure Systems and Architectures*, VISA '10, pages 33–40, New York, NY, USA, 2010. ACM.

[181] YouTube, Inc. Statistics, 2013. `http://www.youtube.com/yt/press/statistics.html`.

[182] J. Yu, K. Varadhan, T. Li, and V. Fuller. Classless Inter-Domain Routing (CIDR): An Address Assignment and Aggregation Strategy. *IETF, RFC 1519*, 1993.

[183] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration. *ACM SIGCOMM Computer Communication Review*, 38(2):17–29, 2008.

[184] E. W. Zegura. GT-ITM: Georgia Tech Internetwork Topology Models (Software). *Georgia Tech*, 1996. `http://www.cc.gatech.edu/fac/Ellen.Zegura/gt-itm/gt-itm/tar.gz`.

[185] S. Zhang, J. Wu, and S. Lu. Virtual Network Embedding with Substrate Support for Parallelization. In *IEEE Global Communications Conference (GLOBECOM 2012)*, pages 2615–2620. IEEE, 2012.

[186] Z. Zhang, S. Su, X. Niu, J. Ma, X. Cheng, and K. Shuang. Minimizing Electricity Cost in Geographical Virtual Network Embedding. In *IEEE Global Communications Conference (GLOBECOM 2012)*, pages 2609–2614. IEEE, 2012.

[187] Y. Zhu and M. Ammar. Algorithms for Assigning Substrate Network Resources to Virtual Network Components. In *25th IEEE International Conference on Computer Communications (INFOCOM 2006)*, pages 1–12. IEEE, 2006.

[188] T. Zinner, P. Tran-Gia, K. Tutschku, and A. Nakao. Performance Evaluation of Packet Reordering on Concurrent Multipath Transmissions for Transport Virtualisation. *International Journal of Communication Networks and Distributed Systems*, 6(3):322–340, 2011.

APPENDIX A

# Solutions in Detail

Here, we present the detailed results achieved by CH-O, CH-R, CH-S, LS-O, LS-S, VND-O, VND-S, GRASP-O, MA-O, VNS-O, CP-O, CP-S, ILP-O, and ILP-S for each individual instance of the VNMP instance set, which is the basis for the analysis carried out in the previous chapters. This data allows for comparisons with future algorithms solving the VNMP and is also available online [87]. We report the additional resource cost $C_a$ and substrate usage cost $C_u$ of the final VNMP solution. The required run-time is labeled by t[s]. For the ILP methods, we also report the final gaps in percent. The gap is 100% if the LP relaxation of the root node could not be solved. For the ILP and CP methods, column Opt shows if the optimality of the solution could be proven. Note that for ILP-S, any valid solution is optimal. Values may be missing due to early termination because of the applied memory limit. Table A.1 gives an overview where the results can be found.

Table A.1: Tables showing the achieved results for the VNMP instance set, depending on employed methods, the instance size, and the instance number (Nr.)

| Methods | Size | Nr. 0–14 | Nr. 15–29 | Size | Nr. 0–14 | Nr. 15–29 |
|---|---|---|---|---|---|---|
| CH-O, CH-R, CH-S, LS-O, LS-S | 20 | A.2 | A.3 | 30 | A.4 | A.5 |
| | 50 | A.6 | A.7 | 100 | A.8 | A.9 |
| | 200 | A.10 | A.11 | 500 | A.12 | A.13 |
| | 1000 | A.14 | A.15 | | | |
| VND-O, VND-S, GRASP-O, MA-O, VNS-O | 20 | A.16 | A.17 | 30 | A.18 | A.19 |
| | 50 | A.20 | A.21 | 100 | A.22 | A.23 |
| | 200 | A.24 | A.25 | 500 | A.26 | A.27 |
| | 1000 | A.28 | A.29 | | | |
| CP-O, CP-S | 20 | A.30 | A.31 | 30 | A.32 | A.33 |
| | 50 | A.34 | A.35 | 100 | A.36 | A.37 |
| ILP-O, ILP-S | 20 | A.38 | A.39 | 30 | A.40 | A.41 |
| | 50 | A.42 | A.43 | 100 | A.44 | A.45 |
| | 200 | A.46 | A.47 | 500 | A.48 | A.49 |
| | 1000 | A.50 | A.51 | | | |

Table A.2: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 20, instance numbers (Nr.) 0–14.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 20 | 0 | 0.10 | 0 | 572 | 0.0 | 0 | 607 | 0.0 | 0 | 595 | 0.0 | 0 | 513 | 0.0 | 0 | 560 | 0.0 |
| | | 0.50 | 0 | 1063 | 0.0 | 0 | 921 | 0.0 | 0 | 1048 | 0.0 | 0 | 829 | 0.2 | 0 | 931 | 0.0 |
| | | 0.80 | 0 | 1223 | 0.0 | 0 | 1161 | 0.0 | 0 | 1284 | 0.0 | 0 | 1107 | 0.2 | 0 | 1223 | 0.0 |
| | | 1.00 | 625 | 1382 | 0.0 | 520 | 1224 | 0.0 | 0 | 1382 | 0.0 | 0 | 1184 | 0.3 | 0 | 1336 | 0.0 |
| | 1 | 0.10 | 0 | 377 | 0.0 | 0 | 474 | 0.0 | 0 | 377 | 0.0 | 0 | 361 | 0.0 | 0 | 377 | 0.0 |
| | | 0.50 | 0 | 791 | 0.0 | 545 | 1017 | 0.0 | 0 | 910 | 0.0 | 0 | 737 | 0.1 | 0 | 791 | 0.0 |
| | | 0.80 | 0 | 1040 | 0.0 | 695 | 980 | 0.0 | 0 | 1030 | 0.0 | 0 | 804 | 0.2 | 0 | 958 | 0.0 |
| | | 1.00 | 0 | 1168 | 0.0 | 1626 | 1003 | 0.0 | 0 | 1153 | 0.0 | 113 | 994 | 0.3 | 0 | 1139 | 0.0 |
| | 2 | 0.10 | 0 | 511 | 0.0 | 0 | 604 | 0.0 | 0 | 511 | 0.0 | 0 | 484 | 0.0 | 0 | 511 | 0.0 |
| | | 0.50 | 0 | 960 | 0.0 | 0 | 1017 | 0.0 | 0 | 1032 | 0.0 | 0 | 791 | 0.1 | 0 | 960 | 0.0 |
| | | 0.80 | 0 | 1297 | 0.0 | 0 | 1106 | 0.0 | 0 | 1388 | 0.0 | 0 | 972 | 0.2 | 0 | 1249 | 0.0 |
| | | 1.00 | 0 | 1470 | 0.0 | 430 | 1431 | 0.0 | 0 | 1623 | 0.0 | 0 | 1201 | 0.3 | 0 | 1360 | 0.0 |
| | 3 | 0.10 | 0 | 739 | 0.0 | 0 | 553 | 0.0 | 0 | 798 | 0.0 | 0 | 535 | 0.0 | 0 | 698 | 0.0 |
| | | 0.50 | 0 | 1147 | 0.0 | 0 | 1018 | 0.0 | 0 | 1147 | 0.0 | 0 | 960 | 0.1 | 0 | 1147 | 0.0 |
| | | 0.80 | 0 | 1309 | 0.0 | 535 | 1235 | 0.0 | 0 | 1397 | 0.0 | 0 | 1107 | 0.1 | 0 | 1271 | 0.0 |
| | | 1.00 | 0 | 1582 | 0.0 | 947 | 1437 | 0.0 | 0 | 1550 | 0.0 | 0 | 1380 | 0.2 | 0 | 1474 | 0.0 |
| | 4 | 0.10 | 0 | 543 | 0.0 | 0 | 589 | 0.0 | 0 | 543 | 0.0 | 0 | 506 | 0.0 | 0 | 511 | 0.0 |
| | | 0.50 | 0 | 1048 | 0.0 | 0 | 914 | 0.0 | 0 | 1048 | 0.0 | 0 | 818 | 0.1 | 0 | 1011 | 0.0 |
| | | 0.80 | 0 | 1248 | 0.0 | 0 | 1090 | 0.0 | 0 | 1350 | 0.0 | 0 | 989 | 0.2 | 0 | 1248 | 0.0 |
| | | 1.00 | 658 | 1399 | 0.0 | 385 | 1236 | 0.0 | 146 | 1488 | 0.0 | 0 | 1124 | 0.4 | 0 | 1399 | 0.1 |
| | 5 | 0.10 | 0 | 464 | 0.0 | 0 | 486 | 0.0 | 0 | 530 | 0.0 | 0 | 401 | 0.0 | 0 | 464 | 0.0 |
| | | 0.50 | 0 | 978 | 0.0 | 105 | 1020 | 0.0 | 0 | 1116 | 0.0 | 0 | 832 | 0.1 | 0 | 978 | 0.0 |
| | | 0.80 | 0 | 1166 | 0.0 | 0 | 1018 | 0.0 | 0 | 1245 | 0.0 | 0 | 888 | 0.2 | 0 | 1131 | 0.0 |
| | | 1.00 | 0 | 1177 | 0.0 | 0 | 1134 | 0.0 | 0 | 1311 | 0.0 | 0 | 1057 | 0.1 | 0 | 1142 | 0.0 |
| | 6 | 0.10 | 0 | 493 | 0.0 | 0 | 389 | 0.0 | 0 | 493 | 0.0 | 0 | 389 | 0.0 | 0 | 479 | 0.0 |
| | | 0.50 | 0 | 1143 | 0.0 | 0 | 963 | 0.0 | 0 | 1280 | 0.0 | 0 | 860 | 0.1 | 0 | 1071 | 0.0 |
| | | 0.80 | 0 | 1406 | 0.0 | 0 | 1267 | 0.0 | 0 | 1422 | 0.0 | 0 | 1229 | 0.1 | 0 | 1406 | 0.0 |
| | | 1.00 | 425 | 1422 | 0.0 | 0 | 1422 | 0.0 | 0 | 1446 | 0.0 | 0 | 1422 | 0.1 | 0 | 1422 | 0.0 |
| | 7 | 0.10 | 0 | 831 | 0.0 | 0 | 573 | 0.0 | 0 | 819 | 0.0 | 0 | 551 | 0.0 | 0 | 725 | 0.0 |
| | | 0.50 | 0 | 1289 | 0.0 | 0 | 1069 | 0.0 | 0 | 1334 | 0.0 | 0 | 982 | 0.1 | 0 | 1207 | 0.0 |
| | | 0.80 | 0 | 1418 | 0.0 | 225 | 1285 | 0.0 | 0 | 1506 | 0.0 | 0 | 1237 | 0.2 | 0 | 1418 | 0.0 |
| | | 1.00 | 0 | 1590 | 0.0 | 965 | 1503 | 0.0 | 155 | 1630 | 0.0 | 0 | 1461 | 0.3 | 0 | 1506 | 0.0 |
| | 8 | 0.10 | 0 | 656 | 0.0 | 0 | 418 | 0.0 | 0 | 716 | 0.0 | 0 | 351 | 0.0 | 0 | 559 | 0.0 |
| | | 0.50 | 0 | 1092 | 0.0 | 0 | 884 | 0.0 | 0 | 1219 | 0.0 | 0 | 841 | 0.1 | 0 | 1049 | 0.0 |
| | | 0.80 | 0 | 1171 | 0.0 | 0 | 1162 | 0.0 | 0 | 1378 | 0.0 | 0 | 1070 | 0.2 | 0 | 1214 | 0.0 |
| | | 1.00 | 0 | 1442 | 0.0 | 0 | 1326 | 0.0 | 55 | 1442 | 0.0 | 0 | 1273 | 0.1 | 0 | 1353 | 0.0 |
| | 9 | 0.10 | 0 | 782 | 0.0 | 0 | 672 | 0.0 | 0 | 754 | 0.0 | 0 | 643 | 0.0 | 0 | 753 | 0.0 |
| | | 0.50 | 0 | 1094 | 0.0 | 0 | 1129 | 0.0 | 0 | 1121 | 0.0 | 0 | 1011 | 0.1 | 0 | 1059 | 0.0 |
| | | 0.80 | 0 | 1163 | 0.0 | 0 | 1392 | 0.0 | 0 | 1254 | 0.0 | 0 | 1106 | 0.1 | 0 | 1163 | 0.0 |
| | | 1.00 | 305 | 1455 | 0.0 | 572 | 1445 | 0.0 | 0 | 1560 | 0.0 | 0 | 1287 | 0.3 | 0 | 1355 | 0.0 |
| | 10 | 0.10 | 0 | 660 | 0.0 | 0 | 722 | 0.0 | 0 | 717 | 0.0 | 0 | 598 | 0.0 | 0 | 660 | 0.0 |
| | | 0.50 | 0 | 1093 | 0.0 | 0 | 1124 | 0.0 | 0 | 1236 | 0.0 | 0 | 923 | 0.1 | 0 | 1027 | 0.0 |
| | | 0.80 | 0 | 1374 | 0.0 | 0 | 1459 | 0.0 | 50 | 1454 | 0.0 | 0 | 1252 | 0.1 | 0 | 1302 | 0.0 |
| | | 1.00 | 1035 | 1538 | 0.0 | 825 | 1502 | 0.0 | 65 | 1538 | 0.0 | 1035 | 1502 | 0.1 | 1035 | 1538 | 0.0 |
| | 11 | 0.10 | 0 | 308 | 0.0 | 0 | 272 | 0.0 | 0 | 362 | 0.0 | 0 | 217 | 0.0 | 0 | 308 | 0.0 |
| | | 0.50 | 0 | 933 | 0.0 | 0 | 793 | 0.0 | 0 | 997 | 0.0 | 0 | 638 | 0.1 | 0 | 829 | 0.0 |
| | | 0.80 | 0 | 1077 | 0.0 | 0 | 839 | 0.0 | 0 | 1077 | 0.0 | 0 | 977 | 0.1 | 0 | 1077 | 0.0 |
| | | 1.00 | 0 | 1077 | 0.0 | 0 | 1023 | 0.0 | 145 | 1224 | 0.0 | 0 | 942 | 0.1 | 0 | 997 | 0.0 |
| | 12 | 0.10 | 0 | 573 | 0.0 | 0 | 650 | 0.0 | 0 | 573 | 0.0 | 0 | 557 | 0.0 | 0 | 573 | 0.0 |
| | | 0.50 | 70 | 1153 | 0.0 | 235 | 1004 | 0.0 | 0 | 1236 | 0.0 | 0 | 1022 | 0.1 | 0 | 1121 | 0.0 |
| | | 0.80 | 625 | 1380 | 0.0 | 700 | 1173 | 0.0 | 0 | 1465 | 0.0 | 0 | 1137 | 0.4 | 0 | 1407 | 0.0 |
| | | 1.00 | 790 | 1607 | 0.0 | 3302 | 1343 | 0.0 | 0 | 1613 | 0.0 | 0 | 1238 | 0.5 | 0 | 1476 | 0.0 |
| | 13 | 0.10 | 0 | 638 | 0.0 | 0 | 802 | 0.0 | 0 | 644 | 0.0 | 0 | 480 | 0.0 | 0 | 621 | 0.0 |
| | | 0.50 | 0 | 900 | 0.0 | 0 | 1004 | 0.0 | 0 | 972 | 0.0 | 0 | 733 | 0.1 | 0 | 888 | 0.0 |
| | | 0.80 | 0 | 1267 | 0.0 | 530 | 1279 | 0.0 | 0 | 1294 | 0.0 | 0 | 1109 | 0.2 | 0 | 1277 | 0.0 |
| | | 1.00 | 50 | 1293 | 0.0 | 1320 | 1338 | 0.0 | 0 | 1338 | 0.0 | 0 | 1224 | 0.2 | 0 | 1330 | 0.0 |
| | 14 | 0.10 | 0 | 636 | 0.0 | 0 | 681 | 0.0 | 0 | 636 | 0.0 | 0 | 535 | 0.0 | 0 | 576 | 0.0 |
| | | 0.50 | 0 | 1195 | 0.0 | 0 | 977 | 0.0 | 0 | 1203 | 0.0 | 0 | 828 | 0.1 | 0 | 1195 | 0.0 |
| | | 0.80 | 0 | 1310 | 0.0 | 0 | 1231 | 0.0 | 0 | 1310 | 0.0 | 0 | 1214 | 0.2 | 0 | 1310 | 0.0 |
| | | 1.00 | 250 | 1344 | 0.0 | 2405 | 1301 | 0.0 | 0 | 1456 | 0.0 | 250 | 1301 | 0.1 | 0 | 1344 | 0.0 |

Table A.3: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 20, instance numbers (Nr.) 15–29.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 20 | 15 | 0.10 | 0 | 430 | 0.0 | 0 | 539 | 0.0 | 0 | 524 | 0.0 | 0 | 424 | 0.0 | 0 | 524 | 0.0 |
| | | 0.50 | 0 | 905 | 0.0 | 0 | 1057 | 0.0 | 0 | 980 | 0.0 | 0 | 722 | 0.1 | 0 | 863 | 0.0 |
| | | 0.80 | 0 | 1094 | 0.0 | 0 | 964 | 0.0 | 0 | 1140 | 0.0 | 0 | 964 | 0.2 | 0 | 1140 | 0.0 |
| | | 1.00 | 0 | 1299 | 0.0 | 354 | 1169 | 0.0 | 0 | 1474 | 0.0 | 0 | 1029 | 0.3 | 0 | 1224 | 0.0 |
| | 16 | 0.10 | 0 | 931 | 0.0 | 0 | 837 | 0.0 | 0 | 1092 | 0.0 | 0 | 736 | 0.0 | 0 | 961 | 0.0 |
| | | 0.50 | 0 | 1373 | 0.0 | 0 | 1204 | 0.0 | 0 | 1398 | 0.0 | 0 | 895 | 0.3 | 0 | 1273 | 0.0 |
| | | 0.80 | 0 | 1542 | 0.0 | 0 | 1415 | 0.0 | 0 | 1564 | 0.0 | 0 | 1227 | 0.4 | 0 | 1462 | 0.0 |
| | | 1.00 | 2575 | 1564 | 0.0 | 2363 | 1543 | 0.0 | 0 | 1584 | 0.0 | 0 | 1285 | 0.4 | 0 | 1564 | 0.0 |
| | 17 | 0.10 | 0 | 563 | 0.0 | 0 | 583 | 0.0 | 0 | 563 | 0.0 | 0 | 453 | 0.0 | 0 | 549 | 0.0 |
| | | 0.50 | 0 | 1061 | 0.0 | 0 | 1132 | 0.0 | 0 | 1136 | 0.0 | 0 | 982 | 0.1 | 0 | 1061 | 0.0 |
| | | 0.80 | 0 | 1356 | 0.0 | 0 | 1311 | 0.0 | 0 | 1489 | 0.0 | 0 | 1169 | 0.2 | 0 | 1299 | 0.0 |
| | | 1.00 | 758 | 1561 | 0.0 | 1048 | 1560 | 0.0 | 129 | 1623 | 0.0 | 0 | 1297 | 0.5 | 0 | 1561 | 0.1 |
| | 18 | 0.10 | 0 | 528 | 0.0 | 0 | 507 | 0.0 | 0 | 528 | 0.0 | 0 | 439 | 0.0 | 0 | 517 | 0.0 |
| | | 0.50 | 0 | 1112 | 0.0 | 0 | 920 | 0.0 | 0 | 1179 | 0.0 | 0 | 891 | 0.1 | 0 | 1074 | 0.0 |
| | | 0.80 | 1015 | 1271 | 0.0 | 0 | 1197 | 0.0 | 0 | 1271 | 0.0 | 180 | 1093 | 0.2 | 180 | 1271 | 0.1 |
| | | 1.00 | 1050 | 1328 | 0.0 | 1734 | 1262 | 0.0 | 0 | 1374 | 0.0 | 940 | 1157 | 0.3 | 325 | 1328 | 0.1 |
| | 19 | 0.10 | 0 | 643 | 0.0 | 0 | 627 | 0.0 | 0 | 662 | 0.0 | 0 | 584 | 0.0 | 0 | 643 | 0.0 |
| | | 0.50 | 0 | 1401 | 0.0 | 0 | 1030 | 0.0 | 0 | 1493 | 0.0 | 0 | 1008 | 0.1 | 0 | 1309 | 0.0 |
| | | 0.80 | 0 | 1500 | 0.0 | 845 | 1405 | 0.0 | 0 | 1535 | 0.0 | 0 | 1226 | 0.3 | 0 | 1491 | 0.0 |
| | | 1.00 | 905 | 1535 | 0.0 | 1118 | 1473 | 0.0 | 0 | 1573 | 0.0 | 0 | 1318 | 0.3 | 0 | 1535 | 0.1 |
| | 20 | 0.10 | 0 | 396 | 0.0 | 0 | 546 | 0.0 | 0 | 434 | 0.0 | 0 | 396 | 0.0 | 0 | 403 | 0.0 |
| | | 0.50 | 0 | 1182 | 0.0 | 0 | 739 | 0.0 | 0 | 1182 | 0.0 | 0 | 640 | 0.1 | 0 | 1083 | 0.0 |
| | | 0.80 | 20 | 1322 | 0.0 | 280 | 892 | 0.0 | 0 | 1322 | 0.0 | 0 | 835 | 0.2 | 0 | 1307 | 0.0 |
| | | 1.00 | 0 | 1366 | 0.0 | 2568 | 1102 | 0.0 | 0 | 1391 | 0.0 | 0 | 1008 | 0.3 | 0 | 1366 | 0.0 |
| | 21 | 0.10 | 0 | 484 | 0.0 | 0 | 507 | 0.0 | 0 | 535 | 0.0 | 0 | 475 | 0.0 | 0 | 484 | 0.0 |
| | | 0.50 | 0 | 898 | 0.0 | 0 | 893 | 0.0 | 0 | 902 | 0.0 | 0 | 758 | 0.1 | 0 | 898 | 0.0 |
| | | 0.80 | 0 | 1052 | 0.0 | 0 | 1050 | 0.0 | 0 | 1221 | 0.0 | 0 | 952 | 0.1 | 0 | 1052 | 0.0 |
| | | 1.00 | 421 | 1300 | 0.0 | 646 | 1158 | 0.0 | 0 | 1360 | 0.0 | 0 | 1097 | 0.2 | 0 | 1268 | 0.1 |
| | 22 | 0.10 | 0 | 647 | 0.0 | 0 | 662 | 0.0 | 0 | 647 | 0.0 | 0 | 646 | 0.0 | 0 | 647 | 0.0 |
| | | 0.50 | 0 | 748 | 0.0 | 0 | 801 | 0.0 | 0 | 794 | 0.0 | 0 | 747 | 0.1 | 0 | 748 | 0.0 |
| | | 0.80 | 0 | 1120 | 0.0 | 0 | 898 | 0.0 | 0 | 1110 | 0.0 | 0 | 898 | 0.4 | 0 | 1120 | 0.0 |
| | | 1.00 | 0 | 1127 | 0.0 | 0 | 905 | 0.0 | 0 | 1171 | 0.0 | 0 | 905 | 0.5 | 0 | 1127 | 0.0 |
| | 23 | 0.10 | 0 | 612 | 0.0 | 0 | 593 | 0.0 | 0 | 754 | 0.0 | 0 | 535 | 0.0 | 0 | 591 | 0.0 |
| | | 0.50 | 0 | 1180 | 0.0 | 0 | 919 | 0.0 | 0 | 1256 | 0.0 | 0 | 779 | 0.1 | 0 | 1039 | 0.0 |
| | | 0.80 | 0 | 1490 | 0.0 | 0 | 1066 | 0.0 | 0 | 1500 | 0.0 | 0 | 990 | 0.2 | 0 | 1390 | 0.0 |
| | | 1.00 | 0 | 1515 | 0.0 | 0 | 1338 | 0.0 | 0 | 1564 | 0.0 | 0 | 1131 | 0.2 | 0 | 1415 | 0.0 |
| | 24 | 0.10 | 0 | 716 | 0.0 | 0 | 767 | 0.0 | 0 | 716 | 0.0 | 0 | 683 | 0.0 | 0 | 716 | 0.0 |
| | | 0.50 | 0 | 1044 | 0.0 | 0 | 910 | 0.0 | 0 | 1128 | 0.0 | 0 | 910 | 0.1 | 0 | 930 | 0.0 |
| | | 0.80 | 0 | 1184 | 0.0 | 0 | 1153 | 0.0 | 0 | 1247 | 0.0 | 0 | 1153 | 0.1 | 0 | 1184 | 0.0 |
| | | 1.00 | 0 | 1258 | 0.0 | 283 | 1226 | 0.0 | 0 | 1395 | 0.0 | 0 | 1257 | 0.2 | 0 | 1257 | 0.0 |
| | 25 | 0.10 | 0 | 525 | 0.0 | 0 | 533 | 0.0 | 0 | 568 | 0.0 | 0 | 432 | 0.0 | 0 | 525 | 0.0 |
| | | 0.50 | 0 | 955 | 0.0 | 0 | 1058 | 0.0 | 0 | 990 | 0.0 | 0 | 827 | 0.1 | 0 | 902 | 0.0 |
| | | 0.80 | 0 | 1195 | 0.0 | 345 | 1164 | 0.0 | 0 | 1394 | 0.0 | 0 | 1043 | 0.1 | 0 | 1131 | 0.0 |
| | | 1.00 | 0 | 1481 | 0.0 | 30 | 1214 | 0.0 | 0 | 1522 | 0.0 | 0 | 1043 | 0.3 | 0 | 1357 | 0.0 |
| | 26 | 0.10 | 0 | 559 | 0.0 | 0 | 508 | 0.0 | 0 | 559 | 0.0 | 0 | 420 | 0.0 | 0 | 527 | 0.0 |
| | | 0.50 | 0 | 1024 | 0.0 | 0 | 946 | 0.0 | 0 | 1058 | 0.0 | 0 | 937 | 0.1 | 0 | 1022 | 0.0 |
| | | 0.80 | 0 | 1047 | 0.0 | 0 | 1069 | 0.0 | 0 | 1101 | 0.0 | 0 | 1024 | 0.1 | 0 | 1047 | 0.0 |
| | | 1.00 | 730 | 1240 | 0.0 | 0 | 1112 | 0.0 | 0 | 1213 | 0.0 | 0 | 1112 | 0.2 | 0 | 1240 | 0.1 |
| | 27 | 0.10 | 0 | 687 | 0.0 | 0 | 699 | 0.0 | 0 | 687 | 0.0 | 0 | 607 | 0.0 | 0 | 654 | 0.0 |
| | | 0.50 | 0 | 1086 | 0.0 | 0 | 1271 | 0.0 | 0 | 1347 | 0.0 | 0 | 901 | 0.1 | 0 | 1086 | 0.0 |
| | | 0.80 | 0 | 1284 | 0.0 | 0 | 1461 | 0.0 | 0 | 1538 | 0.0 | 0 | 1068 | 0.3 | 0 | 1286 | 0.0 |
| | | 1.00 | 590 | 1446 | 0.0 | 385 | 1511 | 0.0 | 0 | 1538 | 0.0 | 385 | 1233 | 0.4 | 0 | 1538 | 0.0 |
| | 28 | 0.10 | 0 | 568 | 0.0 | 0 | 531 | 0.0 | 0 | 630 | 0.0 | 0 | 552 | 0.0 | 0 | 568 | 0.0 |
| | | 0.50 | 0 | 1249 | 0.0 | 0 | 1225 | 0.0 | 0 | 1247 | 0.0 | 0 | 1034 | 0.1 | 0 | 1187 | 0.0 |
| | | 0.80 | 0 | 1367 | 0.0 | 0 | 1347 | 0.0 | 0 | 1576 | 0.0 | 0 | 1243 | 0.2 | 0 | 1368 | 0.0 |
| | | 1.00 | 315 | 1581 | 0.0 | 0 | 1409 | 0.0 | 385 | 1581 | 0.0 | 315 | 1390 | 0.2 | 30 | 1436 | 0.1 |
| | 29 | 0.10 | 0 | 460 | 0.0 | 0 | 431 | 0.0 | 0 | 460 | 0.0 | 0 | 387 | 0.0 | 0 | 460 | 0.0 |
| | | 0.50 | 0 | 1029 | 0.0 | 175 | 866 | 0.0 | 0 | 1050 | 0.0 | 0 | 593 | 0.1 | 0 | 1040 | 0.0 |
| | | 0.80 | 0 | 1249 | 0.0 | 40 | 1079 | 0.0 | 0 | 1285 | 0.0 | 0 | 886 | 0.2 | 0 | 1192 | 0.0 |
| | | 1.00 | 97 | 1377 | 0.0 | 333 | 1326 | 0.0 | 0 | 1433 | 0.0 | 0 | 1334 | 0.1 | 0 | 1377 | 0.0 |

Table A.4: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 30, instance numbers (Nr.) 0–14.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 30 | 0 | 0.10 | 0 | 633 | 0.0 | 0 | 641 | 0.0 | 0 | 633 | 0.0 | 0 | 611 | 0.0 | 0 | 619 | 0.0 |
| | | 0.50 | 0 | 1332 | 0.0 | 0 | 1289 | 0.0 | 0 | 1327 | 0.0 | 0 | 1036 | 0.2 | 0 | 1287 | 0.0 |
| | | 0.80 | 0 | 1509 | 0.0 | 0 | 1560 | 0.0 | 0 | 1714 | 0.0 | 0 | 1454 | 0.3 | 0 | 1553 | 0.0 |
| | | 1.00 | 0 | 1711 | 0.0 | 905 | 1882 | 0.0 | 0 | 2091 | 0.0 | 0 | 1586 | 0.3 | 0 | 1716 | 0.0 |
| | 1 | 0.10 | 0 | 793 | 0.0 | 0 | 887 | 0.0 | 0 | 932 | 0.0 | 0 | 713 | 0.1 | 0 | 876 | 0.0 |
| | | 0.50 | 0 | 1470 | 0.0 | 0 | 1539 | 0.0 | 0 | 1789 | 0.0 | 0 | 1149 | 0.5 | 0 | 1649 | 0.0 |
| | | 0.80 | 0 | 1951 | 0.0 | 2808 | 1927 | 0.0 | 0 | 2382 | 0.0 | 0 | 1510 | 0.7 | 0 | 2077 | 0.1 |
| | | 1.00 | 4041 | 2330 | 0.0 | 4782 | 2067 | 0.0 | 0 | 2474 | 0.0 | 0 | 1689 | 1.3 | 0 | 2457 | 0.1 |
| | 2 | 0.10 | 0 | 1288 | 0.0 | 0 | 1028 | 0.0 | 0 | 1467 | 0.0 | 0 | 860 | 0.1 | 0 | 1250 | 0.0 |
| | | 0.50 | 0 | 1818 | 0.0 | 0 | 1634 | 0.0 | 0 | 2135 | 0.0 | 0 | 1628 | 0.3 | 0 | 1925 | 0.0 |
| | | 0.80 | 0 | 2176 | 0.0 | 0 | 1866 | 0.0 | 0 | 2384 | 0.0 | 0 | 1810 | 0.6 | 0 | 2210 | 0.1 |
| | | 1.00 | 233 | 2288 | 0.0 | 3102 | 2084 | 0.0 | 0 | 2546 | 0.0 | 0 | 2216 | 0.4 | 0 | 2432 | 0.1 |
| | 3 | 0.10 | 0 | 1000 | 0.0 | 0 | 701 | 0.0 | 0 | 1000 | 0.0 | 0 | 679 | 0.0 | 0 | 844 | 0.0 |
| | | 0.50 | 0 | 1347 | 0.0 | 0 | 1177 | 0.0 | 0 | 1558 | 0.0 | 0 | 1095 | 0.2 | 0 | 1365 | 0.0 |
| | | 0.80 | 565 | 1879 | 0.0 | 0 | 1234 | 0.0 | 0 | 1871 | 0.0 | 0 | 1203 | 0.7 | 0 | 1815 | 0.0 |
| | | 1.00 | 1636 | 2096 | 0.0 | 1467 | 1869 | 0.0 | 0 | 2162 | 0.0 | 0 | 1745 | 1.3 | 0 | 2110 | 0.1 |
| | 4 | 0.10 | 0 | 1036 | 0.0 | 0 | 979 | 0.0 | 0 | 1024 | 0.0 | 0 | 701 | 0.1 | 0 | 994 | 0.0 |
| | | 0.50 | 0 | 1610 | 0.0 | 0 | 1563 | 0.0 | 0 | 1733 | 0.0 | 0 | 1376 | 0.4 | 0 | 1602 | 0.0 |
| | | 0.80 | 0 | 1696 | 0.0 | 80 | 1595 | 0.0 | 0 | 1838 | 0.0 | 0 | 1436 | 0.6 | 0 | 1688 | 0.0 |
| | | 1.00 | 0 | 1788 | 0.0 | 385 | 1938 | 0.0 | 0 | 2219 | 0.0 | 0 | 1647 | 0.5 | 0 | 1807 | 0.1 |
| | 5 | 0.10 | 0 | 737 | 0.0 | 0 | 641 | 0.0 | 0 | 801 | 0.0 | 0 | 612 | 0.0 | 0 | 712 | 0.0 |
| | | 0.50 | 0 | 1429 | 0.0 | 0 | 1163 | 0.0 | 0 | 1480 | 0.0 | 0 | 1105 | 0.4 | 0 | 1426 | 0.0 |
| | | 0.80 | 0 | 1649 | 0.0 | 0 | 1588 | 0.0 | 0 | 1969 | 0.0 | 0 | 1402 | 0.6 | 0 | 1614 | 0.0 |
| | | 1.00 | 0 | 1906 | 0.0 | 657 | 1618 | 0.0 | 0 | 2002 | 0.0 | 0 | 1466 | 0.9 | 0 | 1774 | 0.0 |
| | 6 | 0.10 | 0 | 853 | 0.0 | 0 | 786 | 0.0 | 0 | 971 | 0.0 | 0 | 624 | 0.1 | 0 | 899 | 0.0 |
| | | 0.50 | 0 | 1467 | 0.0 | 0 | 1414 | 0.0 | 0 | 1637 | 0.0 | 0 | 1206 | 0.4 | 0 | 1515 | 0.0 |
| | | 0.80 | 330 | 1796 | 0.0 | 145 | 1642 | 0.0 | 0 | 1824 | 0.0 | 0 | 1333 | 0.7 | 0 | 1755 | 0.0 |
| | | 1.00 | 5 | 1779 | 0.0 | 5376 | 1769 | 0.0 | 0 | 2051 | 0.0 | 0 | 1544 | 0.6 | 0 | 1864 | 0.1 |
| | 7 | 0.10 | 0 | 897 | 0.0 | 0 | 769 | 0.0 | 0 | 1002 | 0.0 | 0 | 684 | 0.0 | 0 | 768 | 0.0 |
| | | 0.50 | 0 | 1589 | 0.0 | 0 | 1355 | 0.0 | 0 | 1661 | 0.0 | 0 | 1226 | 0.5 | 0 | 1596 | 0.0 |
| | | 0.80 | 0 | 1834 | 0.0 | 0 | 1686 | 0.0 | 0 | 2031 | 0.0 | 0 | 1510 | 0.8 | 0 | 1965 | 0.1 |
| | | 1.00 | 415 | 2296 | 0.0 | 3097 | 1948 | 0.0 | 0 | 2362 | 0.0 | 0 | 1752 | 1.3 | 0 | 2285 | 0.1 |
| | 8 | 0.10 | 0 | 1489 | 0.0 | 0 | 1275 | 0.0 | 0 | 1561 | 0.0 | 0 | 1192 | 0.1 | 0 | 1375 | 0.0 |
| | | 0.50 | 0 | 1659 | 0.0 | 0 | 1457 | 0.0 | 0 | 1746 | 0.0 | 0 | 1269 | 0.6 | 0 | 1541 | 0.0 |
| | | 0.80 | 0 | 1758 | 0.0 | 0 | 1761 | 0.0 | 0 | 1773 | 0.0 | 0 | 1629 | 0.6 | 0 | 1746 | 0.1 |
| | | 1.00 | 0 | 1758 | 0.0 | 200 | 1915 | 0.0 | 0 | 1838 | 0.0 | 0 | 1625 | 1.0 | 0 | 1773 | 0.1 |
| | 9 | 0.10 | 0 | 743 | 0.0 | 0 | 654 | 0.0 | 0 | 822 | 0.0 | 0 | 571 | 0.0 | 0 | 762 | 0.0 |
| | | 0.50 | 0 | 1638 | 0.0 | 0 | 1423 | 0.0 | 0 | 1924 | 0.0 | 0 | 1164 | 0.3 | 0 | 1784 | 0.0 |
| | | 0.80 | 0 | 2065 | 0.0 | 0 | 1939 | 0.0 | 0 | 2234 | 0.0 | 0 | 1486 | 0.8 | 0 | 2145 | 0.0 |
| | | 1.00 | 0 | 2242 | 0.0 | 678 | 2179 | 0.0 | 0 | 2382 | 0.0 | 0 | 1788 | 0.9 | 0 | 2280 | 0.1 |
| | 10 | 0.10 | 0 | 610 | 0.0 | 0 | 811 | 0.0 | 0 | 610 | 0.0 | 0 | 593 | 0.0 | 0 | 610 | 0.0 |
| | | 0.50 | 0 | 1375 | 0.0 | 0 | 1292 | 0.0 | 0 | 1375 | 0.0 | 0 | 1122 | 0.3 | 0 | 1289 | 0.0 |
| | | 0.80 | 0 | 1662 | 0.0 | 0 | 1436 | 0.0 | 0 | 1717 | 0.0 | 0 | 1259 | 0.8 | 0 | 1538 | 0.0 |
| | | 1.00 | 0 | 1930 | 0.0 | 494 | 1539 | 0.0 | 0 | 1977 | 0.0 | 0 | 1349 | 1.3 | 0 | 1817 | 0.1 |
| | 11 | 0.10 | 0 | 908 | 0.0 | 0 | 625 | 0.0 | 0 | 1064 | 0.0 | 0 | 597 | 0.1 | 0 | 914 | 0.0 |
| | | 0.50 | 0 | 1430 | 0.0 | 405 | 1360 | 0.0 | 0 | 1627 | 0.0 | 0 | 1188 | 0.2 | 0 | 1459 | 0.0 |
| | | 0.80 | 140 | 2088 | 0.0 | 3507 | 2094 | 0.0 | 0 | 2155 | 0.0 | 0 | 1663 | 0.4 | 0 | 1894 | 0.0 |
| | | 1.00 | 3089 | 2184 | 0.0 | 5516 | 2278 | 0.0 | 10 | 2327 | 0.0 | 1481 | 2043 | 0.3 | 0 | 2184 | 0.2 |
| | 12 | 0.10 | 0 | 754 | 0.0 | 0 | 867 | 0.0 | 0 | 871 | 0.0 | 0 | 754 | 0.0 | 0 | 827 | 0.0 |
| | | 0.50 | 0 | 1439 | 0.0 | 0 | 1369 | 0.0 | 0 | 1539 | 0.0 | 0 | 1102 | 0.2 | 0 | 1392 | 0.0 |
| | | 0.80 | 0 | 1923 | 0.0 | 125 | 1713 | 0.0 | 0 | 2081 | 0.0 | 0 | 1434 | 0.6 | 0 | 1821 | 0.0 |
| | | 1.00 | 0 | 2160 | 0.0 | 262 | 1972 | 0.0 | 0 | 2216 | 0.0 | 0 | 1742 | 0.7 | 0 | 2150 | 0.0 |
| | 13 | 0.10 | 0 | 674 | 0.0 | 0 | 655 | 0.0 | 0 | 698 | 0.0 | 0 | 542 | 0.0 | 0 | 617 | 0.0 |
| | | 0.50 | 0 | 1424 | 0.0 | 0 | 1476 | 0.0 | 0 | 1523 | 0.0 | 0 | 1102 | 0.3 | 0 | 1386 | 0.0 |
| | | 0.80 | 0 | 1677 | 0.0 | 422 | 1910 | 0.0 | 0 | 1921 | 0.0 | 0 | 1517 | 0.3 | 0 | 1703 | 0.0 |
| | | 1.00 | 0 | 1929 | 0.0 | 436 | 1791 | 0.0 | 0 | 2132 | 0.0 | 0 | 1555 | 0.6 | 0 | 1926 | 0.1 |
| | 14 | 0.10 | 0 | 902 | 0.0 | 0 | 940 | 0.0 | 0 | 935 | 0.0 | 0 | 809 | 0.1 | 0 | 884 | 0.0 |
| | | 0.50 | 0 | 1771 | 0.0 | 0 | 1593 | 0.0 | 0 | 1909 | 0.0 | 0 | 1413 | 0.5 | 0 | 1802 | 0.0 |
| | | 0.80 | 0 | 1825 | 0.0 | 0 | 1631 | 0.0 | 0 | 2065 | 0.0 | 0 | 1490 | 1.1 | 0 | 1923 | 0.1 |
| | | 1.00 | 1875 | 2019 | 0.0 | 3089 | 1799 | 0.0 | 0 | 2347 | 0.0 | 0 | 1812 | 1.2 | 0 | 2109 | 0.1 |

Table A.5: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 30, instance numbers (Nr.) 15–29.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 30 | 15 | 0.10 | 0 | 1245 | 0.0 | 0 | 1160 | 0.0 | 0 | 1226 | 0.0 | 0 | 1019 | 0.1 | 0 | 1163 | 0.0 |
| | | 0.50 | 0 | 1633 | 0.0 | 0 | 1582 | 0.0 | 0 | 1633 | 0.0 | 0 | 1298 | 0.4 | 0 | 1623 | 0.0 |
| | | 0.80 | 0 | 1867 | 0.0 | 0 | 1650 | 0.0 | 0 | 2084 | 0.0 | 0 | 1531 | 1.1 | 0 | 1779 | 0.1 |
| | | 1.00 | 0 | 2304 | 0.0 | 623 | 1901 | 0.0 | 0 | 2304 | 0.0 | 0 | 1773 | 2.7 | 0 | 2206 | 0.1 |
| | 16 | 0.10 | 0 | 633 | 0.0 | 0 | 534 | 0.0 | 0 | 649 | 0.0 | 0 | 461 | 0.0 | 0 | 589 | 0.0 |
| | | 0.50 | 0 | 1382 | 0.0 | 0 | 1196 | 0.0 | 0 | 1556 | 0.0 | 0 | 997 | 0.2 | 0 | 1279 | 0.0 |
| | | 0.80 | 0 | 1777 | 0.0 | 1070 | 1505 | 0.0 | 0 | 1797 | 0.0 | 0 | 1229 | 0.5 | 0 | 1704 | 0.0 |
| | | 1.00 | 289 | 1924 | 0.0 | 1814 | 1796 | 0.0 | 0 | 1931 | 0.0 | 0 | 1521 | 1.2 | 0 | 1900 | 0.1 |
| | 17 | 0.10 | 0 | 976 | 0.0 | 0 | 999 | 0.0 | 0 | 1065 | 0.0 | 0 | 904 | 0.1 | 0 | 992 | 0.0 |
| | | 0.50 | 0 | 1392 | 0.0 | 0 | 1412 | 0.0 | 0 | 1427 | 0.0 | 0 | 1132 | 0.5 | 0 | 1370 | 0.0 |
| | | 0.80 | 305 | 1779 | 0.0 | 1725 | 1654 | 0.0 | 0 | 1972 | 0.0 | 0 | 1463 | 1.0 | 0 | 1772 | 0.1 |
| | | 1.00 | 3630 | 1901 | 0.0 | 933 | 1711 | 0.0 | 0 | 2053 | 0.0 | 0 | 1604 | 1.2 | 0 | 1922 | 0.1 |
| | 18 | 0.10 | 0 | 579 | 0.0 | 0 | 658 | 0.0 | 0 | 629 | 0.0 | 0 | 561 | 0.0 | 0 | 561 | 0.0 |
| | | 0.50 | 0 | 1458 | 0.0 | 25 | 1530 | 0.0 | 0 | 1612 | 0.0 | 0 | 1200 | 0.3 | 0 | 1381 | 0.0 |
| | | 0.80 | 0 | 1831 | 0.0 | 727 | 2061 | 0.0 | 0 | 1928 | 0.0 | 0 | 1534 | 0.4 | 0 | 1724 | 0.0 |
| | | 1.00 | 0 | 2181 | 0.0 | 597 | 2292 | 0.0 | 0 | 2304 | 0.0 | 0 | 1750 | 0.5 | 0 | 2066 | 0.1 |
| | 19 | 0.10 | 0 | 955 | 0.0 | 0 | 810 | 0.0 | 0 | 1005 | 0.0 | 0 | 548 | 0.0 | 0 | 984 | 0.0 |
| | | 0.50 | 0 | 1643 | 0.0 | 0 | 1614 | 0.0 | 0 | 1833 | 0.0 | 0 | 1432 | 0.4 | 0 | 1706 | 0.0 |
| | | 0.80 | 0 | 1928 | 0.0 | 270 | 1820 | 0.0 | 0 | 2136 | 0.0 | 0 | 1570 | 0.9 | 0 | 1932 | 0.0 |
| | | 1.00 | 315 | 2248 | 0.0 | 3234 | 2168 | 0.0 | 0 | 2261 | 0.0 | 0 | 1818 | 1.2 | 0 | 2210 | 0.1 |
| | 20 | 0.10 | 0 | 1195 | 0.0 | 0 | 1165 | 0.0 | 0 | 1222 | 0.0 | 0 | 941 | 0.1 | 0 | 1050 | 0.0 |
| | | 0.50 | 0 | 1771 | 0.0 | 0 | 1513 | 0.0 | 0 | 1807 | 0.0 | 0 | 1490 | 0.4 | 0 | 1761 | 0.0 |
| | | 0.80 | 0 | 2011 | 0.0 | 0 | 1710 | 0.0 | 0 | 1985 | 0.0 | 0 | 1686 | 1.0 | 0 | 1973 | 0.0 |
| | | 1.00 | 0 | 2129 | 0.0 | 38 | 1885 | 0.0 | 0 | 2200 | 0.0 | 0 | 1842 | 1.4 | 0 | 2129 | 0.1 |
| | 21 | 0.10 | 0 | 897 | 0.0 | 0 | 684 | 0.0 | 0 | 878 | 0.0 | 0 | 640 | 0.0 | 0 | 812 | 0.0 |
| | | 0.50 | 0 | 1283 | 0.0 | 0 | 1243 | 0.0 | 0 | 1332 | 0.0 | 0 | 1010 | 0.1 | 0 | 1283 | 0.0 |
| | | 0.80 | 0 | 1636 | 0.0 | 0 | 1629 | 0.0 | 0 | 1769 | 0.0 | 0 | 1460 | 0.3 | 0 | 1565 | 0.0 |
| | | 1.00 | 46 | 1875 | 0.0 | 1314 | 1887 | 0.0 | 75 | 2012 | 0.0 | 41 | 1743 | 0.3 | 0 | 1875 | 0.1 |
| | 22 | 0.10 | 0 | 968 | 0.0 | 0 | 1013 | 0.0 | 0 | 1003 | 0.0 | 0 | 929 | 0.0 | 0 | 968 | 0.0 |
| | | 0.50 | 0 | 1554 | 0.0 | 0 | 1442 | 0.0 | 0 | 1706 | 0.0 | 0 | 1400 | 0.3 | 0 | 1479 | 0.0 |
| | | 0.80 | 75 | 1813 | 0.0 | 0 | 1693 | 0.0 | 0 | 2048 | 0.0 | 0 | 1491 | 0.6 | 0 | 1727 | 0.1 |
| | | 1.00 | 529 | 2249 | 0.0 | 100 | 1757 | 0.0 | 0 | 2292 | 0.0 | 0 | 1631 | 1.1 | 0 | 2110 | 0.1 |
| | 23 | 0.10 | 0 | 766 | 0.0 | 0 | 1213 | 0.0 | 0 | 876 | 0.0 | 0 | 738 | 0.0 | 0 | 841 | 0.0 |
| | | 0.50 | 0 | 1589 | 0.0 | 0 | 1582 | 0.0 | 0 | 1949 | 0.0 | 0 | 1214 | 0.5 | 0 | 1643 | 0.0 |
| | | 0.80 | 115 | 2240 | 0.0 | 40 | 1974 | 0.0 | 0 | 2300 | 0.0 | 0 | 1492 | 1.0 | 0 | 2135 | 0.1 |
| | | 1.00 | 270 | 2410 | 0.0 | 785 | 2360 | 0.0 | 0 | 2380 | 0.0 | 0 | 1745 | 1.0 | 0 | 2184 | 0.1 |
| | 24 | 0.10 | 0 | 626 | 0.0 | 0 | 571 | 0.0 | 0 | 630 | 0.0 | 0 | 564 | 0.0 | 0 | 626 | 0.0 |
| | | 0.50 | 0 | 1398 | 0.0 | 10 | 1498 | 0.0 | 0 | 1799 | 0.0 | 0 | 1278 | 0.3 | 0 | 1479 | 0.0 |
| | | 0.80 | 0 | 1936 | 0.0 | 0 | 1862 | 0.0 | 0 | 2187 | 0.0 | 0 | 1700 | 0.6 | 0 | 2155 | 0.0 |
| | | 1.00 | 309 | 2234 | 0.0 | 5062 | 2445 | 0.0 | 0 | 2514 | 0.0 | 0 | 2139 | 0.6 | 0 | 2349 | 0.1 |
| | 25 | 0.10 | 0 | 1123 | 0.0 | 0 | 1381 | 0.0 | 0 | 1222 | 0.0 | 0 | 972 | 0.1 | 0 | 1187 | 0.0 |
| | | 0.50 | 0 | 1645 | 0.0 | 0 | 1709 | 0.0 | 0 | 1886 | 0.0 | 0 | 1533 | 0.5 | 0 | 1694 | 0.0 |
| | | 0.80 | 0 | 2029 | 0.0 | 615 | 1872 | 0.0 | 0 | 2147 | 0.0 | 0 | 1587 | 0.9 | 0 | 1925 | 0.1 |
| | | 1.00 | 1370 | 2095 | 0.0 | 6659 | 2057 | 0.0 | 0 | 2310 | 0.0 | 0 | 1722 | 1.5 | 0 | 2125 | 0.1 |
| | 26 | 0.10 | 0 | 750 | 0.0 | 0 | 528 | 0.0 | 0 | 763 | 0.0 | 0 | 518 | 0.0 | 0 | 763 | 0.0 |
| | | 0.50 | 0 | 1479 | 0.0 | 0 | 1068 | 0.0 | 0 | 1528 | 0.0 | 0 | 1009 | 0.2 | 0 | 1410 | 0.0 |
| | | 0.80 | 345 | 1941 | 0.0 | 590 | 1736 | 0.0 | 0 | 1938 | 0.0 | 0 | 1364 | 0.5 | 0 | 1825 | 0.0 |
| | | 1.00 | 4841 | 2047 | 0.0 | 7421 | 2034 | 0.0 | 0 | 2143 | 0.0 | 0 | 1554 | 0.5 | 0 | 2024 | 0.1 |
| | 27 | 0.10 | 0 | 773 | 0.0 | 0 | 651 | 0.0 | 0 | 751 | 0.0 | 0 | 561 | 0.0 | 0 | 638 | 0.0 |
| | | 0.50 | 0 | 1396 | 0.0 | 0 | 1174 | 0.0 | 0 | 1504 | 0.0 | 0 | 1106 | 0.2 | 0 | 1267 | 0.0 |
| | | 0.80 | 0 | 1699 | 0.0 | 0 | 1665 | 0.0 | 0 | 1958 | 0.0 | 0 | 1407 | 0.6 | 0 | 1659 | 0.0 |
| | | 1.00 | 0 | 1968 | 0.0 | 686 | 1910 | 0.0 | 0 | 2057 | 0.0 | 0 | 1517 | 0.9 | 0 | 1919 | 0.1 |
| | 28 | 0.10 | 0 | 1031 | 0.0 | 0 | 1131 | 0.0 | 0 | 1073 | 0.0 | 0 | 962 | 0.0 | 0 | 1006 | 0.0 |
| | | 0.50 | 0 | 1422 | 0.0 | 0 | 1457 | 0.0 | 0 | 1619 | 0.0 | 0 | 1238 | 0.1 | 0 | 1422 | 0.0 |
| | | 0.80 | 0 | 1769 | 0.0 | 0 | 1555 | 0.0 | 0 | 2033 | 0.0 | 0 | 1553 | 0.3 | 0 | 1671 | 0.1 |
| | | 1.00 | 0 | 2017 | 0.0 | 20 | 1800 | 0.0 | 0 | 2229 | 0.0 | 0 | 1608 | 0.7 | 0 | 1914 | 0.1 |
| | 29 | 0.10 | 0 | 872 | 0.0 | 0 | 495 | 0.0 | 0 | 935 | 0.0 | 0 | 415 | 0.1 | 0 | 798 | 0.0 |
| | | 0.50 | 0 | 1524 | 0.0 | 0 | 1508 | 0.0 | 0 | 1668 | 0.0 | 0 | 1090 | 0.2 | 0 | 1445 | 0.0 |
| | | 0.80 | 0 | 1858 | 0.0 | 0 | 1771 | 0.0 | 0 | 1944 | 0.0 | 0 | 1437 | 0.4 | 0 | 1761 | 0.0 |
| | | 1.00 | 715 | 1920 | 0.0 | 0 | 2067 | 0.0 | 0 | 1944 | 0.0 | 0 | 1743 | 0.3 | 0 | 1920 | 0.1 |

Table A.6: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 50, instance numbers (Nr.) 0–14.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** |
| 50 | 0 | 0.10 | 0 | 1313 | 0.0 | 0 | 1291 | 0.0 | 0 | 1311 | 0.0 | 0 | 1020 | 0.1 | 0 | 1289 | 0.0 |
| | | 0.50 | 0 | 2410 | 0.0 | 0 | 2522 | 0.0 | 0 | 2571 | 0.0 | 0 | 1872 | 1.5 | 0 | 2411 | 0.1 |
| | | 0.80 | 0 | 2967 | 0.0 | 4241 | 2977 | 0.0 | 0 | 3275 | 0.0 | 0 | 2445 | 2.8 | 0 | 2991 | 0.1 |
| | | 1.00 | 7008 | 3614 | 0.1 | 18804 | 3663 | 0.0 | 0 | 3859 | 0.0 | 2640 | 3240 | 4.0 | 0 | 3731 | 1.3 |
| | 1 | 0.10 | 0 | 1472 | 0.0 | 0 | 1449 | 0.0 | 0 | 1638 | 0.0 | 0 | 1340 | 0.1 | 0 | 1470 | 0.0 |
| | | 0.50 | 0 | 2262 | 0.0 | 0 | 2246 | 0.0 | 0 | 2395 | 0.0 | 0 | 1826 | 0.7 | 0 | 2208 | 0.0 |
| | | 0.80 | 0 | 2720 | 0.0 | 13 | 2643 | 0.0 | 0 | 3180 | 0.0 | 0 | 2088 | 2.2 | 0 | 2937 | 0.1 |
| | | 1.00 | 757 | 3669 | 0.0 | 5574 | 3489 | 0.0 | 0 | 3756 | 0.0 | 0 | 2634 | 2.8 | 0 | 3678 | 0.1 |
| | 2 | 0.10 | 0 | 1126 | 0.0 | 0 | 1071 | 0.0 | 0 | 1175 | 0.0 | 0 | 749 | 0.1 | 0 | 1069 | 0.0 |
| | | 0.50 | 0 | 2453 | 0.0 | 0 | 2474 | 0.0 | 0 | 2610 | 0.0 | 0 | 2043 | 0.5 | 0 | 2372 | 0.1 |
| | | 0.80 | 0 | 3002 | 0.0 | 0 | 2862 | 0.0 | 0 | 3116 | 0.0 | 0 | 2373 | 1.1 | 0 | 2915 | 0.1 |
| | | 1.00 | 1148 | 3097 | 0.0 | 1287 | 3273 | 0.0 | 0 | 3386 | 0.0 | 520 | 2668 | 1.6 | 95 | 3049 | 0.3 |
| | 3 | 0.10 | 0 | 1410 | 0.0 | 0 | 1444 | 0.0 | 0 | 1588 | 0.0 | 0 | 1211 | 0.1 | 0 | 1512 | 0.0 |
| | | 0.50 | 0 | 2602 | 0.0 | 0 | 2315 | 0.0 | 0 | 2957 | 0.0 | 0 | 1905 | 1.4 | 0 | 2575 | 0.1 |
| | | 0.80 | 0 | 3282 | 0.0 | 0 | 2536 | 0.0 | 0 | 3416 | 0.0 | 0 | 2370 | 2.4 | 0 | 3223 | 0.1 |
| | | 1.00 | 958 | 3633 | 0.0 | 5673 | 3013 | 0.0 | 0 | 3776 | 0.0 | 0 | 2502 | 4.0 | 0 | 3573 | 0.1 |
| | 4 | 0.10 | 0 | 1313 | 0.0 | 0 | 1300 | 0.0 | 0 | 1360 | 0.0 | 0 | 1194 | 0.1 | 0 | 1317 | 0.0 |
| | | 0.50 | 0 | 2232 | 0.0 | 0 | 2275 | 0.0 | 0 | 2303 | 0.0 | 0 | 1904 | 0.7 | 0 | 2175 | 0.1 |
| | | 0.80 | 0 | 2604 | 0.0 | 0 | 2746 | 0.0 | 0 | 2750 | 0.0 | 0 | 2188 | 1.8 | 0 | 2560 | 0.1 |
| | | 1.00 | 218 | 3106 | 0.0 | 2794 | 3124 | 0.0 | 0 | 3219 | 0.0 | 0 | 2723 | 2.2 | 0 | 3124 | 0.1 |
| | 5 | 0.10 | 0 | 1397 | 0.0 | 0 | 1376 | 0.0 | 0 | 1470 | 0.0 | 0 | 1280 | 0.1 | 0 | 1431 | 0.0 |
| | | 0.50 | 0 | 2143 | 0.0 | 0 | 1888 | 0.0 | 0 | 2290 | 0.0 | 0 | 1664 | 1.2 | 0 | 2111 | 0.0 |
| | | 0.80 | 0 | 2676 | 0.0 | 0 | 1989 | 0.0 | 0 | 2837 | 0.0 | 0 | 2031 | 2.7 | 0 | 2720 | 0.1 |
| | | 1.00 | 0 | 2907 | 0.0 | 616 | 2406 | 0.0 | 0 | 3172 | 0.0 | 0 | 2221 | 4.0 | 0 | 2831 | 0.1 |
| | 6 | 0.10 | 0 | 900 | 0.0 | 145 | 1205 | 0.0 | 0 | 947 | 0.0 | 0 | 825 | 0.1 | 0 | 911 | 0.0 |
| | | 0.50 | 0 | 2096 | 0.0 | 30 | 2500 | 0.0 | 0 | 2431 | 0.0 | 0 | 1732 | 0.8 | 0 | 2064 | 0.0 |
| | | 0.80 | 100 | 2727 | 0.0 | 385 | 2655 | 0.0 | 0 | 2988 | 0.0 | 0 | 2085 | 2.3 | 0 | 2640 | 0.1 |
| | | 1.00 | 370 | 3012 | 0.0 | 4080 | 3132 | 0.0 | 0 | 3264 | 0.0 | 0 | 2460 | 3.0 | 0 | 2954 | 0.1 |
| | 7 | 0.10 | 0 | 1152 | 0.0 | 0 | 1617 | 0.0 | 0 | 1227 | 0.0 | 0 | 1129 | 0.1 | 0 | 1198 | 0.0 |
| | | 0.50 | 0 | 2351 | 0.0 | 1387 | 2558 | 0.0 | 0 | 2513 | 0.0 | 0 | 2010 | 1.2 | 0 | 2268 | 0.1 |
| | | 0.80 | 40 | 2746 | 0.0 | 905 | 2823 | 0.0 | 0 | 2731 | 0.0 | 0 | 2464 | 1.4 | 0 | 2653 | 0.1 |
| | | 1.00 | 360 | 3060 | 0.0 | 1674 | 3192 | 0.1 | 0 | 3412 | 0.0 | 0 | 2643 | 3.3 | 0 | 3086 | 0.1 |
| | 8 | 0.10 | 0 | 1053 | 0.0 | 130 | 1224 | 0.0 | 0 | 1194 | 0.0 | 0 | 946 | 0.1 | 0 | 1097 | 0.0 |
| | | 0.50 | 0 | 2202 | 0.0 | 190 | 2219 | 0.0 | 0 | 2337 | 0.0 | 0 | 1619 | 0.8 | 0 | 2159 | 0.1 |
| | | 0.80 | 0 | 2732 | 0.0 | 200 | 2609 | 0.0 | 0 | 3180 | 0.0 | 0 | 2127 | 2.2 | 0 | 2824 | 0.1 |
| | | 1.00 | 7306 | 3537 | 0.0 | 6770 | 3608 | 0.0 | 0 | 3873 | 0.0 | 0 | 2956 | 3.2 | 0 | 3648 | 0.8 |
| | 9 | 0.10 | 0 | 1623 | 0.0 | 0 | 1566 | 0.0 | 0 | 1736 | 0.0 | 0 | 1392 | 0.2 | 0 | 1623 | 0.0 |
| | | 0.50 | 0 | 2369 | 0.0 | 0 | 2263 | 0.0 | 0 | 2370 | 0.0 | 0 | 1818 | 1.1 | 0 | 2229 | 0.1 |
| | | 0.80 | 0 | 2683 | 0.0 | 0 | 2542 | 0.0 | 0 | 2803 | 0.0 | 0 | 2076 | 2.1 | 0 | 2596 | 0.1 |
| | | 1.00 | 0 | 3144 | 0.0 | 1251 | 3038 | 0.0 | 0 | 3438 | 0.0 | 0 | 2404 | 4.7 | 0 | 3122 | 0.1 |
| | 10 | 0.10 | 0 | 1079 | 0.0 | 0 | 1256 | 0.0 | 0 | 1094 | 0.0 | 0 | 988 | 0.1 | 0 | 1046 | 0.0 |
| | | 0.50 | 0 | 2677 | 0.0 | 0 | 2346 | 0.0 | 0 | 2894 | 0.0 | 0 | 1862 | 1.4 | 0 | 2637 | 0.1 |
| | | 0.80 | 0 | 3001 | 0.0 | 20 | 2780 | 0.0 | 0 | 3284 | 0.0 | 0 | 2165 | 2.1 | 0 | 2908 | 0.1 |
| | | 1.00 | 1657 | 3380 | 0.0 | 3819 | 3073 | 0.0 | 0 | 3840 | 0.0 | 0 | 2726 | 3.6 | 0 | 3330 | 0.1 |
| | 11 | 0.10 | 0 | 1380 | 0.0 | 0 | 1249 | 0.0 | 0 | 1402 | 0.0 | 0 | 978 | 0.2 | 0 | 1336 | 0.0 |
| | | 0.50 | 0 | 2498 | 0.0 | 45 | 2110 | 0.0 | 0 | 2507 | 0.0 | 0 | 1703 | 0.9 | 0 | 2375 | 0.0 |
| | | 0.80 | 400 | 3011 | 0.0 | 17 | 2928 | 0.0 | 0 | 3261 | 0.0 | 0 | 2353 | 1.6 | 0 | 2941 | 0.1 |
| | | 1.00 | 3373 | 3428 | 0.0 | 4288 | 3412 | 0.0 | 0 | 3671 | 0.0 | 585 | 3129 | 2.2 | 585 | 3204 | 0.8 |
| | 12 | 0.10 | 0 | 1042 | 0.0 | 0 | 981 | 0.0 | 0 | 1098 | 0.0 | 0 | 889 | 0.1 | 0 | 1026 | 0.0 |
| | | 0.50 | 0 | 2326 | 0.0 | 0 | 2049 | 0.0 | 0 | 2493 | 0.0 | 0 | 1837 | 1.1 | 0 | 2343 | 0.1 |
| | | 0.80 | 0 | 2600 | 0.0 | 0 | 2647 | 0.0 | 0 | 2827 | 0.0 | 0 | 2364 | 1.1 | 0 | 2598 | 0.1 |
| | | 1.00 | 2609 | 2810 | 0.0 | 4501 | 2865 | 0.0 | 25 | 3174 | 0.0 | 0 | 2682 | 1.4 | 0 | 2810 | 0.5 |
| | 13 | 0.10 | 0 | 1375 | 0.0 | 0 | 1312 | 0.0 | 0 | 1438 | 0.0 | 0 | 1086 | 0.1 | 0 | 1358 | 0.0 |
| | | 0.50 | 0 | 2247 | 0.0 | 0 | 2132 | 0.0 | 0 | 2303 | 0.0 | 0 | 1897 | 0.9 | 0 | 2222 | 0.0 |
| | | 0.80 | 0 | 3116 | 0.0 | 105 | 2728 | 0.0 | 0 | 3175 | 0.0 | 0 | 2389 | 2.1 | 0 | 2995 | 0.1 |
| | | 1.00 | 5020 | 3454 | 0.0 | 4965 | 3490 | 0.0 | 80 | 3495 | 0.0 | 0 | 3190 | 3.1 | 0 | 3428 | 0.8 |
| | 14 | 0.10 | 0 | 1278 | 0.0 | 0 | 1360 | 0.0 | 0 | 1286 | 0.0 | 0 | 1096 | 0.1 | 0 | 1250 | 0.0 |
| | | 0.50 | 0 | 2144 | 0.0 | 0 | 2098 | 0.0 | 0 | 2202 | 0.0 | 0 | 1859 | 0.8 | 0 | 2149 | 0.0 |
| | | 0.80 | 0 | 2606 | 0.0 | 38 | 2458 | 0.0 | 0 | 2673 | 0.0 | 0 | 2096 | 2.7 | 0 | 2562 | 0.1 |
| | | 1.00 | 5 | 2724 | 0.0 | 544 | 2575 | 0.0 | 0 | 2918 | 0.0 | 0 | 2283 | 3.0 | 0 | 2665 | 0.1 |

Table A.7: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 50, instance numbers (Nr.) 15–29.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 50 | 15 | 0.10 | 0 | 1396 | 0.0 | 0 | 1297 | 0.0 | 0 | 1449 | 0.0 | 0 | 1161 | 0.1 | 0 | 1398 | 0.0 |
| | | 0.50 | 0 | 2618 | 0.0 | 55 | 2688 | 0.0 | 0 | 2775 | 0.0 | 0 | 2091 | 1.2 | 0 | 2577 | 0.1 |
| | | 0.80 | 275 | 3074 | 0.1 | 480 | 2997 | 0.0 | 0 | 3137 | 0.0 | 0 | 2278 | 3.1 | 0 | 3029 | 0.1 |
| | | 1.00 | 4494 | 3581 | 0.0 | 2336 | 3397 | 0.1 | 0 | 3770 | 0.0 | 0 | 3227 | 3.6 | 0 | 3689 | 0.2 |
| | 16 | 0.10 | 0 | 1305 | 0.0 | 0 | 1549 | 0.0 | 0 | 1369 | 0.0 | 0 | 1180 | 0.1 | 0 | 1295 | 0.0 |
| | | 0.50 | 0 | 2234 | 0.0 | 0 | 2237 | 0.0 | 0 | 2326 | 0.0 | 0 | 1509 | 1.9 | 0 | 2151 | 0.1 |
| | | 0.80 | 0 | 2682 | 0.0 | 1529 | 2754 | 0.0 | 0 | 2711 | 0.0 | 0 | 2142 | 3.1 | 0 | 2627 | 0.1 |
| | | 1.00 | 25 | 3083 | 0.0 | 3606 | 3047 | 0.0 | 0 | 3409 | 0.0 | 0 | 2390 | 4.5 | 0 | 3062 | 0.1 |
| | 17 | 0.10 | 0 | 1276 | 0.0 | 0 | 1475 | 0.0 | 0 | 1422 | 0.0 | 0 | 1103 | 0.1 | 0 | 1314 | 0.0 |
| | | 0.50 | 0 | 2502 | 0.0 | 0 | 2445 | 0.0 | 0 | 2576 | 0.0 | 0 | 2191 | 0.9 | 0 | 2555 | 0.0 |
| | | 0.80 | 0 | 2945 | 0.0 | 0 | 2670 | 0.0 | 0 | 3249 | 0.0 | 0 | 2388 | 2.1 | 0 | 2991 | 0.1 |
| | | 1.00 | 430 | 3665 | 0.0 | 1667 | 3124 | 0.0 | 0 | 4059 | 0.0 | 0 | 2755 | 4.1 | 0 | 3763 | 0.1 |
| | 18 | 0.10 | 0 | 1034 | 0.0 | 0 | 995 | 0.0 | 0 | 1093 | 0.0 | 0 | 863 | 0.1 | 0 | 1025 | 0.0 |
| | | 0.50 | 45 | 2144 | 0.0 | 0 | 2250 | 0.0 | 0 | 2367 | 0.0 | 0 | 1945 | 0.7 | 0 | 2148 | 0.1 |
| | | 0.80 | 0 | 2937 | 0.0 | 0 | 2545 | 0.0 | 0 | 3089 | 0.0 | 0 | 2433 | 2.3 | 0 | 2994 | 0.1 |
| | | 1.00 | 2401 | 3438 | 0.0 | 7945 | 3049 | 0.0 | 0 | 3680 | 0.0 | 0 | 2894 | 3.9 | 0 | 3542 | 0.4 |
| | 19 | 0.10 | 0 | 1187 | 0.0 | 0 | 1090 | 0.0 | 0 | 1433 | 0.0 | 0 | 1065 | 0.1 | 0 | 1311 | 0.0 |
| | | 0.50 | 0 | 2133 | 0.0 | 0 | 1753 | 0.0 | 0 | 2331 | 0.0 | 0 | 1628 | 1.3 | 0 | 2234 | 0.1 |
| | | 0.80 | 0 | 2769 | 0.0 | 0 | 2298 | 0.0 | 0 | 2737 | 0.0 | 0 | 2042 | 1.9 | 0 | 2605 | 0.1 |
| | | 1.00 | 5028 | 3012 | 0.0 | 6512 | 2932 | 0.0 | 0 | 3128 | 0.0 | 0 | 2585 | 2.6 | 0 | 3012 | 0.4 |
| | 20 | 0.10 | 0 | 1837 | 0.0 | 0 | 1831 | 0.0 | 0 | 2150 | 0.0 | 0 | 1659 | 0.2 | 0 | 2027 | 0.0 |
| | | 0.50 | 0 | 2353 | 0.0 | 295 | 2504 | 0.0 | 0 | 2590 | 0.0 | 0 | 2042 | 0.9 | 0 | 2462 | 0.1 |
| | | 0.80 | 0 | 2838 | 0.0 | 1210 | 3096 | 0.0 | 0 | 2946 | 0.0 | 0 | 2513 | 1.4 | 0 | 2870 | 0.1 |
| | | 1.00 | 2774 | 3256 | 0.1 | 5274 | 3320 | 0.1 | 0 | 3234 | 0.0 | 0 | 2841 | 4.3 | 0 | 3306 | 0.1 |
| | 21 | 0.10 | 0 | 1482 | 0.0 | 0 | 1523 | 0.0 | 0 | 1489 | 0.0 | 0 | 1210 | 0.1 | 0 | 1322 | 0.0 |
| | | 0.50 | 0 | 2187 | 0.0 | 0 | 2416 | 0.0 | 0 | 2410 | 0.0 | 0 | 1814 | 1.0 | 0 | 2144 | 0.1 |
| | | 0.80 | 10 | 2817 | 0.0 | 230 | 2906 | 0.0 | 0 | 3225 | 0.0 | 0 | 2431 | 3.0 | 0 | 2949 | 0.1 |
| | | 1.00 | 590 | 3403 | 0.0 | 10 | 3341 | 0.0 | 0 | 3553 | 0.0 | 0 | 2801 | 5.1 | 0 | 3352 | 0.1 |
| | 22 | 0.10 | 0 | 1532 | 0.0 | 0 | 1602 | 0.0 | 0 | 1853 | 0.0 | 0 | 1377 | 0.1 | 0 | 1684 | 0.0 |
| | | 0.50 | 0 | 2661 | 0.0 | 0 | 2707 | 0.0 | 0 | 3092 | 0.0 | 0 | 2093 | 1.5 | 0 | 2778 | 0.1 |
| | | 0.80 | 0 | 3072 | 0.0 | 0 | 3120 | 0.0 | 0 | 3264 | 0.0 | 0 | 2425 | 2.6 | 0 | 3075 | 0.1 |
| | | 1.00 | 79 | 3334 | 0.0 | 3984 | 3673 | 0.0 | 0 | 3766 | 0.0 | 0 | 2841 | 3.3 | 0 | 3385 | 0.1 |
| | 23 | 0.10 | 0 | 1209 | 0.0 | 0 | 1151 | 0.0 | 0 | 1318 | 0.0 | 0 | 1035 | 0.1 | 0 | 1165 | 0.0 |
| | | 0.50 | 0 | 2700 | 0.0 | 0 | 2718 | 0.0 | 0 | 2861 | 0.0 | 0 | 2184 | 1.2 | 0 | 2655 | 0.1 |
| | | 0.80 | 0 | 3100 | 0.0 | 450 | 3071 | 0.0 | 0 | 3272 | 0.0 | 0 | 2484 | 2.6 | 0 | 3060 | 0.1 |
| | | 1.00 | 0 | 3148 | 0.0 | 557 | 3159 | 0.0 | 0 | 3378 | 0.0 | 0 | 2680 | 2.5 | 0 | 3185 | 0.1 |
| | 24 | 0.10 | 0 | 1047 | 0.0 | 0 | 1013 | 0.0 | 0 | 1137 | 0.0 | 0 | 803 | 0.1 | 0 | 1057 | 0.0 |
| | | 0.50 | 0 | 2259 | 0.0 | 80 | 2092 | 0.0 | 0 | 2883 | 0.0 | 0 | 1645 | 1.9 | 0 | 2461 | 0.1 |
| | | 0.80 | 0 | 3193 | 0.0 | 449 | 2715 | 0.0 | 0 | 3510 | 0.0 | 0 | 2276 | 2.8 | 0 | 3189 | 0.1 |
| | | 1.00 | 0 | 3415 | 0.0 | 3401 | 3503 | 0.0 | 0 | 3732 | 0.0 | 0 | 2925 | 3.7 | 0 | 3462 | 0.1 |
| | 25 | 0.10 | 0 | 1243 | 0.0 | 0 | 1139 | 0.0 | 0 | 1343 | 0.0 | 0 | 962 | 0.1 | 0 | 1214 | 0.0 |
| | | 0.50 | 0 | 2420 | 0.0 | 0 | 1856 | 0.0 | 0 | 2438 | 0.0 | 0 | 1737 | 1.4 | 0 | 2347 | 0.0 |
| | | 0.80 | 0 | 2637 | 0.0 | 0 | 2201 | 0.0 | 0 | 2716 | 0.0 | 0 | 2115 | 1.6 | 0 | 2532 | 0.1 |
| | | 1.00 | 0 | 2844 | 0.0 | 2202 | 2473 | 0.0 | 0 | 3084 | 0.0 | 0 | 2280 | 3.0 | 0 | 2820 | 0.1 |
| | 26 | 0.10 | 0 | 1071 | 0.0 | 0 | 1222 | 0.0 | 0 | 1193 | 0.0 | 0 | 929 | 0.1 | 0 | 1085 | 0.0 |
| | | 0.50 | 0 | 2357 | 0.0 | 0 | 2070 | 0.0 | 0 | 2507 | 0.0 | 0 | 1709 | 1.3 | 0 | 2354 | 0.0 |
| | | 0.80 | 0 | 2824 | 0.0 | 115 | 2656 | 0.0 | 0 | 3048 | 0.0 | 0 | 2308 | 2.1 | 0 | 2840 | 0.1 |
| | | 1.00 | 4 | 2980 | 0.0 | 2238 | 2957 | 0.0 | 0 | 3266 | 0.0 | 0 | 2579 | 2.5 | 0 | 3055 | 0.1 |
| | 27 | 0.10 | 0 | 1497 | 0.0 | 0 | 1631 | 0.0 | 0 | 1609 | 0.0 | 0 | 1397 | 0.0 | 0 | 1502 | 0.0 |
| | | 0.50 | 0 | 2469 | 0.0 | 0 | 2376 | 0.0 | 0 | 2563 | 0.0 | 0 | 2019 | 1.1 | 0 | 2398 | 0.1 |
| | | 0.80 | 0 | 2842 | 0.0 | 0 | 2553 | 0.0 | 0 | 2877 | 0.0 | 0 | 2179 | 1.9 | 0 | 2733 | 0.1 |
| | | 1.00 | 0 | 3099 | 0.0 | 1411 | 2884 | 0.0 | 0 | 3066 | 0.0 | 0 | 2370 | 3.4 | 0 | 3012 | 0.1 |
| | 28 | 0.10 | 0 | 1787 | 0.0 | 0 | 1857 | 0.0 | 0 | 2044 | 0.0 | 0 | 1154 | 0.3 | 0 | 1807 | 0.0 |
| | | 0.50 | 0 | 2941 | 0.0 | 0 | 2875 | 0.0 | 0 | 3192 | 0.0 | 0 | 2279 | 1.1 | 0 | 2990 | 0.1 |
| | | 0.80 | 0 | 3703 | 0.0 | 1977 | 3599 | 0.0 | 0 | 4059 | 0.0 | 0 | 2926 | 2.8 | 0 | 3832 | 0.1 |
| | | 1.00 | 10551 | 4483 | 0.1 | 17697 | 4325 | 0.0 | 543 | 4298 | 0.1 | 0 | 3643 | 2.8 | 0 | 4232 | 0.5 |
| | 29 | 0.10 | 0 | 953 | 0.0 | 0 | 844 | 0.0 | 0 | 1207 | 0.0 | 0 | 620 | 0.1 | 0 | 974 | 0.0 |
| | | 0.50 | 0 | 2583 | 0.0 | 0 | 2321 | 0.0 | 0 | 2705 | 0.0 | 0 | 1977 | 0.9 | 0 | 2660 | 0.1 |
| | | 0.80 | 0 | 3125 | 0.0 | 0 | 2880 | 0.0 | 0 | 3332 | 0.0 | 0 | 2463 | 1.9 | 0 | 3082 | 0.1 |
| | | 1.00 | 0 | 3652 | 0.0 | 25 | 3275 | 0.0 | 0 | 3507 | 0.0 | 0 | 2745 | 2.4 | 0 | 3368 | 0.1 |

Table A.8: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 100, instance numbers (Nr.) 0–14.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 100 | 0 | 0.10 | 0 | 2527 | 0.0 | 0 | 2959 | 0.0 | 0 | 2943 | 0.0 | 0 | 2286 | 0.8 | 0 | 2854 | 0.0 |
| | | 0.50 | 0 | 4209 | 0.1 | 0 | 4028 | 0.1 | 0 | 4557 | 0.1 | 0 | 3235 | 9.0 | 0 | 4145 | 0.1 |
| | | 0.80 | 580 | 5556 | 0.1 | 1355 | 5123 | 0.1 | 0 | 6047 | 0.1 | 0 | 4014 | 26.9 | 0 | 5627 | 0.3 |
| | | 1.00 | 4264 | 5991 | 0.1 | 7294 | 5728 | 0.1 | 0 | 6831 | 0.2 | 0 | 5042 | 26.9 | 0 | 6128 | 0.4 |
| | 1 | 0.10 | 0 | 3310 | 0.0 | 0 | 3714 | 0.0 | 0 | 3533 | 0.0 | 0 | 2788 | 1.5 | 0 | 3264 | 0.0 |
| | | 0.50 | 0 | 4966 | 0.1 | 1915 | 5018 | 0.1 | 0 | 5855 | 0.1 | 0 | 4279 | 7.6 | 0 | 5300 | 0.2 |
| | | 0.80 | 1305 | 5744 | 0.1 | 4514 | 5758 | 0.1 | 0 | 6503 | 0.1 | 0 | 4867 | 11.2 | 0 | 5911 | 0.2 |
| | | 1.00 | 7855 | 6617 | 0.1 | 7721 | 6583 | 0.1 | 0 | 7135 | 0.1 | 0 | 5664 | 17.3 | 0 | 6742 | 0.3 |
| | 2 | 0.10 | 0 | 1522 | 0.0 | 0 | 1687 | 0.0 | 0 | 1531 | 0.0 | 0 | 1401 | 0.3 | 0 | 1504 | 0.0 |
| | | 0.50 | 0 | 3898 | 0.1 | 0 | 3984 | 0.1 | 0 | 4366 | 0.0 | 0 | 2965 | 9.1 | 0 | 3945 | 0.2 |
| | | 0.80 | 0 | 4832 | 0.1 | 1019 | 4445 | 0.1 | 0 | 5550 | 0.1 | 0 | 3769 | 18.6 | 0 | 4851 | 0.3 |
| | | 1.00 | 416 | 5649 | 0.1 | 3743 | 5494 | 0.1 | 0 | 6227 | 0.1 | 0 | 4668 | 21.6 | 0 | 5772 | 0.6 |
| | 3 | 0.10 | 0 | 2913 | 0.0 | 0 | 3436 | 0.0 | 0 | 3497 | 0.0 | 0 | 2553 | 2.1 | 0 | 3169 | 0.0 |
| | | 0.50 | 0 | 5305 | 0.1 | 0 | 5126 | 0.1 | 0 | 6172 | 0.1 | 0 | 4096 | 12.8 | 0 | 5849 | 0.2 |
| | | 0.80 | 0 | 6301 | 0.1 | 60 | 5767 | 0.1 | 0 | 7232 | 0.1 | 0 | 4747 | 25.5 | 0 | 6590 | 0.3 |
| | | 1.00 | 1626 | 7346 | 0.1 | 5479 | 6792 | 0.1 | 0 | 7713 | 0.1 | 0 | 5666 | 31.0 | 0 | 7346 | 0.4 |
| | 4 | 0.10 | 0 | 2414 | 0.0 | 0 | 2356 | 0.0 | 0 | 2699 | 0.0 | 0 | 1980 | 1.0 | 0 | 2628 | 0.0 |
| | | 0.50 | 0 | 4938 | 0.1 | 905 | 4998 | 0.1 | 0 | 5361 | 0.1 | 0 | 4099 | 9.3 | 0 | 5150 | 0.2 |
| | | 0.80 | 355 | 5885 | 0.1 | 2385 | 5591 | 0.1 | 0 | 6278 | 0.1 | 0 | 4651 | 18.1 | 0 | 5789 | 0.3 |
| | | 1.00 | 539 | 6312 | 0.1 | 2175 | 6035 | 0.1 | 0 | 6733 | 0.1 | 410 | 5141 | 25.1 | 0 | 6354 | 0.9 |
| | 5 | 0.10 | 0 | 2139 | 0.0 | 0 | 2231 | 0.0 | 0 | 2557 | 0.0 | 0 | 1927 | 0.8 | 0 | 2475 | 0.0 |
| | | 0.50 | 0 | 4643 | 0.1 | 0 | 5026 | 0.1 | 0 | 5040 | 0.1 | 0 | 3692 | 11.6 | 0 | 4733 | 0.2 |
| | | 0.80 | 0 | 5307 | 0.1 | 735 | 5714 | 0.1 | 0 | 6049 | 0.1 | 0 | 4375 | 18.3 | 0 | 5362 | 0.3 |
| | | 1.00 | 0 | 6272 | 0.1 | 8113 | 6426 | 0.2 | 0 | 6701 | 0.2 | 0 | 4863 | 24.8 | 0 | 5950 | 0.5 |
| | 6 | 0.10 | 0 | 2577 | 0.0 | 0 | 2648 | 0.0 | 0 | 3249 | 0.0 | 0 | 2297 | 1.3 | 0 | 3088 | 0.0 |
| | | 0.50 | 0 | 4578 | 0.1 | 30 | 4213 | 0.1 | 0 | 5276 | 0.1 | 0 | 3750 | 10.8 | 0 | 5029 | 0.2 |
| | | 0.80 | 250 | 5590 | 0.1 | 312 | 5056 | 0.1 | 0 | 6365 | 0.1 | 0 | 4527 | 21.0 | 0 | 5943 | 0.3 |
| | | 1.00 | 1705 | 6983 | 0.1 | 4566 | 6170 | 0.1 | 0 | 7251 | 0.1 | 0 | 5502 | 28.7 | 0 | 6992 | 0.4 |
| | 7 | 0.10 | 0 | 3981 | 0.0 | 0 | 3216 | 0.0 | 0 | 4044 | 0.0 | 0 | 2761 | 1.8 | 0 | 3856 | 0.0 |
| | | 0.50 | 65 | 5187 | 0.1 | 65 | 4727 | 0.1 | 0 | 5304 | 0.0 | 0 | 3930 | 8.8 | 0 | 5244 | 0.2 |
| | | 0.80 | 75 | 6144 | 0.1 | 305 | 5835 | 0.1 | 6 | 6591 | 0.1 | 0 | 4791 | 19.0 | 0 | 6171 | 0.3 |
| | | 1.00 | 310 | 6624 | 0.1 | 3041 | 6385 | 0.1 | 6 | 7189 | 0.1 | 0 | 5381 | 27.2 | 0 | 6819 | 0.4 |
| | 8 | 0.10 | 0 | 2380 | 0.0 | 0 | 2359 | 0.0 | 0 | 2785 | 0.0 | 0 | 1831 | 0.7 | 0 | 2378 | 0.0 |
| | | 0.50 | 0 | 4620 | 0.0 | 0 | 4514 | 0.1 | 0 | 5077 | 0.0 | 0 | 3661 | 5.7 | 0 | 4684 | 0.1 |
| | | 0.80 | 675 | 5400 | 0.1 | 68 | 5822 | 0.1 | 0 | 5985 | 0.1 | 0 | 4236 | 14.4 | 0 | 5461 | 0.2 |
| | | 1.00 | 1557 | 6541 | 0.1 | 6343 | 6973 | 0.1 | 0 | 6916 | 0.1 | 160 | 5202 | 22.6 | 160 | 6177 | 3.7 |
| | 9 | 0.10 | 0 | 2531 | 0.0 | 0 | 2755 | 0.0 | 0 | 2796 | 0.0 | 0 | 2153 | 1.1 | 0 | 2530 | 0.0 |
| | | 0.50 | 70 | 4521 | 0.1 | 186 | 4276 | 0.1 | 0 | 4860 | 0.1 | 0 | 3365 | 14.3 | 70 | 4156 | 1.3 |
| | | 0.80 | 70 | 5275 | 0.1 | 354 | 4982 | 0.1 | 0 | 5701 | 0.1 | 70 | 4055 | 25.0 | 70 | 4946 | 2.5 |
| | | 1.00 | 70 | 5984 | 0.1 | 4406 | 5788 | 0.1 | 0 | 6638 | 0.1 | 70 | 4959 | 25.5 | 70 | 5728 | 2.7 |
| | 10 | 0.10 | 0 | 2937 | 0.0 | 0 | 2998 | 0.0 | 0 | 3291 | 0.0 | 0 | 2179 | 1.1 | 0 | 2844 | 0.0 |
| | | 0.50 | 0 | 4004 | 0.0 | 0 | 3624 | 0.1 | 0 | 4449 | 0.0 | 0 | 2910 | 6.0 | 0 | 3979 | 0.1 |
| | | 0.80 | 0 | 4969 | 0.1 | 661 | 4914 | 0.1 | 0 | 5438 | 0.1 | 0 | 4141 | 12.4 | 0 | 4898 | 0.2 |
| | | 1.00 | 448 | 5824 | 0.1 | 4094 | 5500 | 0.1 | 0 | 6114 | 0.1 | 0 | 4865 | 21.7 | 0 | 5791 | 0.3 |
| | 11 | 0.10 | 0 | 3882 | 0.0 | 0 | 3646 | 0.0 | 0 | 4325 | 0.0 | 0 | 2778 | 1.9 | 0 | 3875 | 0.0 |
| | | 0.50 | 0 | 5648 | 0.1 | 70 | 4845 | 0.1 | 0 | 5704 | 0.1 | 0 | 4362 | 8.2 | 0 | 5579 | 0.2 |
| | | 0.80 | 0 | 6424 | 0.1 | 1764 | 5955 | 0.1 | 0 | 7006 | 0.1 | 0 | 5164 | 16.8 | 0 | 6538 | 0.3 |
| | | 1.00 | 1960 | 7043 | 0.1 | 18262 | 7313 | 0.2 | 1840 | 7701 | 0.2 | 0 | 5803 | 26.2 | 925 | 6370 | 4.9 |
| | 12 | 0.10 | 0 | 2896 | 0.0 | 0 | 2570 | 0.0 | 0 | 3265 | 0.0 | 0 | 2273 | 1.1 | 0 | 2850 | 0.0 |
| | | 0.50 | 0 | 5184 | 0.1 | 0 | 4324 | 0.1 | 0 | 5444 | 0.1 | 0 | 3441 | 9.3 | 0 | 5178 | 0.2 |
| | | 0.80 | 205 | 6193 | 0.1 | 85 | 5355 | 0.1 | 0 | 6736 | 0.1 | 0 | 4553 | 16.2 | 0 | 6299 | 0.3 |
| | | 1.00 | 550 | 6967 | 0.1 | 2149 | 6430 | 0.1 | 0 | 7395 | 0.1 | 0 | 5590 | 20.5 | 85 | 6114 | 3.0 |
| | 13 | 0.10 | 0 | 2705 | 0.0 | 0 | 2774 | 0.0 | 0 | 2820 | 0.0 | 0 | 2192 | 0.8 | 0 | 2710 | 0.0 |
| | | 0.50 | 0 | 4967 | 0.1 | 185 | 5199 | 0.1 | 0 | 5252 | 0.1 | 0 | 3966 | 11.3 | 0 | 5108 | 0.2 |
| | | 0.80 | 0 | 5739 | 0.1 | 90 | 5845 | 0.1 | 0 | 6332 | 0.1 | 0 | 4898 | 20.0 | 0 | 5977 | 0.3 |
| | | 1.00 | 635 | 6670 | 0.1 | 6815 | 6628 | 0.1 | 0 | 7339 | 0.1 | 0 | 5792 | 22.3 | 0 | 6608 | 0.4 |
| | 14 | 0.10 | 0 | 3196 | 0.0 | 0 | 3339 | 0.0 | 0 | 3409 | 0.0 | 0 | 2783 | 1.8 | 0 | 3273 | 0.0 |
| | | 0.50 | 0 | 5138 | 0.1 | 0 | 5175 | 0.1 | 0 | 6012 | 0.1 | 0 | 4181 | 10.1 | 0 | 5216 | 0.2 |
| | | 0.80 | 0 | 6414 | 0.1 | 0 | 6161 | 0.1 | 0 | 6810 | 0.1 | 0 | 4986 | 22.1 | 0 | 6264 | 0.3 |
| | | 1.00 | 1811 | 7048 | 0.1 | 2304 | 6773 | 0.1 | 0 | 7323 | 0.1 | 0 | 5591 | 31.8 | 0 | 6816 | 0.3 |

Table A.9: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 100, instance numbers (Nr.) 15–29.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 100 | 15 | 0.10 | 0 | 2541 | 0.0 | 0 | 2304 | 0.0 | 0 | 2591 | 0.0 | 0 | 1940 | 0.9 | 0 | 2495 | 0.0 |
| | | 0.50 | 0 | 4275 | 0.0 | 0 | 3881 | 0.0 | 0 | 4338 | 0.0 | 0 | 3359 | 5.1 | 0 | 4191 | 0.1 |
| | | 0.80 | 0 | 5350 | 0.1 | 405 | 4894 | 0.1 | 0 | 5798 | 0.1 | 0 | 4445 | 12.2 | 0 | 5439 | 0.2 |
| | | 1.00 | 550 | 6054 | 0.1 | 1760 | 6019 | 0.1 | 0 | 6912 | 0.1 | 0 | 5148 | 15.1 | 0 | 6108 | 0.3 |
| | 16 | 0.10 | 0 | 3185 | 0.0 | 0 | 3039 | 0.0 | 0 | 3553 | 0.0 | 0 | 2547 | 2.3 | 0 | 3299 | 0.0 |
| | | 0.50 | 0 | 4801 | 0.0 | 301 | 4722 | 0.1 | 0 | 5151 | 0.0 | 0 | 3934 | 10.3 | 0 | 4822 | 0.2 |
| | | 0.80 | 0 | 5656 | 0.1 | 1746 | 5895 | 0.1 | 0 | 5930 | 0.1 | 0 | 4610 | 21.1 | 0 | 5695 | 0.3 |
| | | 1.00 | 909 | 6199 | 0.1 | 3343 | 6242 | 0.1 | 0 | 6667 | 0.1 | 0 | 5474 | 33.6 | 0 | 6382 | 0.7 |
| | 17 | 0.10 | 0 | 2529 | 0.0 | 0 | 2583 | 0.0 | 0 | 2793 | 0.0 | 0 | 2157 | 0.6 | 0 | 2595 | 0.0 |
| | | 0.50 | 0 | 5283 | 0.0 | 90 | 5111 | 0.0 | 0 | 5884 | 0.0 | 0 | 4300 | 9.7 | 0 | 5393 | 0.1 |
| | | 0.80 | 0 | 5953 | 0.1 | 1258 | 6088 | 0.1 | 0 | 6681 | 0.1 | 0 | 5143 | 14.2 | 0 | 6005 | 0.2 |
| | | 1.00 | 0 | 6842 | 0.1 | 8986 | 7118 | 0.1 | 0 | 7128 | 0.1 | 0 | 5794 | 19.8 | 0 | 6776 | 0.3 |
| | 18 | 0.10 | 0 | 2399 | 0.0 | 0 | 2585 | 0.0 | 0 | 2419 | 0.0 | 0 | 2048 | 0.7 | 0 | 2304 | 0.0 |
| | | 0.50 | 0 | 4357 | 0.1 | 0 | 3984 | 0.1 | 0 | 4669 | 0.1 | 0 | 3284 | 7.5 | 0 | 4256 | 0.1 |
| | | 0.80 | 0 | 5185 | 0.1 | 1543 | 5233 | 0.1 | 370 | 5719 | 0.1 | 0 | 3966 | 14.1 | 0 | 5037 | 0.2 |
| | | 1.00 | 3130 | 6252 | 0.1 | 2798 | 6450 | 0.1 | 2475 | 6579 | 0.1 | 0 | 4881 | 21.9 | 115 | 5526 | 3.6 |
| | 19 | 0.10 | 0 | 2624 | 0.0 | 0 | 2703 | 0.0 | 0 | 2996 | 0.0 | 0 | 2305 | 1.1 | 0 | 2884 | 0.0 |
| | | 0.50 | 0 | 4377 | 0.1 | 0 | 4511 | 0.1 | 0 | 4561 | 0.1 | 0 | 3547 | 9.5 | 0 | 4265 | 0.2 |
| | | 0.80 | 0 | 5009 | 0.1 | 390 | 5315 | 0.1 | 0 | 5608 | 0.1 | 0 | 4201 | 15.2 | 0 | 4953 | 0.3 |
| | | 1.00 | 3461 | 5766 | 0.1 | 5447 | 5605 | 0.1 | 0 | 6120 | 0.1 | 0 | 4626 | 22.8 | 0 | 5579 | 0.4 |
| | 20 | 0.10 | 0 | 1946 | 0.0 | 0 | 2009 | 0.0 | 0 | 2091 | 0.0 | 0 | 1637 | 0.5 | 0 | 2025 | 0.0 |
| | | 0.50 | 0 | 4777 | 0.0 | 0 | 4573 | 0.0 | 0 | 5357 | 0.0 | 0 | 3733 | 6.5 | 0 | 4983 | 0.1 |
| | | 0.80 | 0 | 5752 | 0.1 | 890 | 4950 | 0.1 | 0 | 6236 | 0.1 | 0 | 4322 | 17.6 | 0 | 5939 | 0.2 |
| | | 1.00 | 509 | 6347 | 0.1 | 1460 | 5811 | 0.1 | 0 | 6614 | 0.1 | 0 | 5136 | 24.6 | 0 | 6610 | 0.3 |
| | 21 | 0.10 | 0 | 2596 | 0.0 | 0 | 2660 | 0.0 | 0 | 3029 | 0.0 | 0 | 2051 | 1.4 | 0 | 2777 | 0.0 |
| | | 0.50 | 470 | 4481 | 0.1 | 0 | 4618 | 0.1 | 0 | 4937 | 0.1 | 0 | 3492 | 8.2 | 0 | 4709 | 0.2 |
| | | 0.80 | 3895 | 5448 | 0.1 | 4520 | 5802 | 0.1 | 0 | 6070 | 0.1 | 0 | 4415 | 18.5 | 0 | 5694 | 0.3 |
| | | 1.00 | 7014 | 5840 | 0.1 | 9414 | 6263 | 0.1 | 0 | 6942 | 0.1 | 0 | 4765 | 21.8 | 0 | 6027 | 0.4 |
| | 22 | 0.10 | 0 | 1209 | 0.0 | 0 | 1413 | 0.0 | 0 | 1238 | 0.0 | 0 | 1097 | 0.2 | 0 | 1218 | 0.0 |
| | | 0.50 | 0 | 4152 | 0.0 | 0 | 3809 | 0.1 | 0 | 4379 | 0.0 | 0 | 3013 | 7.1 | 0 | 4139 | 0.1 |
| | | 0.80 | 0 | 5441 | 0.1 | 152 | 4887 | 0.1 | 0 | 6014 | 0.1 | 0 | 3957 | 17.0 | 0 | 5316 | 0.3 |
| | | 1.00 | 0 | 5848 | 0.1 | 790 | 5522 | 0.1 | 0 | 6649 | 0.1 | 0 | 4450 | 24.0 | 0 | 5947 | 0.3 |
| | 23 | 0.10 | 0 | 2007 | 0.0 | 0 | 2916 | 0.0 | 0 | 2254 | 0.0 | 0 | 1681 | 0.3 | 0 | 2108 | 0.0 |
| | | 0.50 | 340 | 4380 | 0.1 | 520 | 4408 | 0.1 | 0 | 5235 | 0.1 | 0 | 3182 | 9.6 | 0 | 4525 | 0.1 |
| | | 0.80 | 790 | 5419 | 0.1 | 1045 | 5413 | 0.1 | 0 | 6075 | 0.1 | 0 | 4153 | 19.4 | 0 | 5612 | 0.3 |
| | | 1.00 | 1640 | 6321 | 0.1 | 14984 | 6145 | 0.1 | 0 | 7157 | 0.1 | 0 | 5311 | 19.6 | 0 | 6316 | 0.4 |
| | 24 | 0.10 | 0 | 3153 | 0.0 | 0 | 3063 | 0.0 | 0 | 3233 | 0.0 | 0 | 2678 | 0.7 | 0 | 3150 | 0.0 |
| | | 0.50 | 0 | 4961 | 0.0 | 0 | 4568 | 0.1 | 0 | 5563 | 0.1 | 0 | 4043 | 6.9 | 0 | 5154 | 0.1 |
| | | 0.80 | 0 | 6337 | 0.1 | 1023 | 5696 | 0.1 | 0 | 6753 | 0.1 | 0 | 4801 | 16.2 | 0 | 6337 | 0.3 |
| | | 1.00 | 273 | 6595 | 0.1 | 2731 | 6275 | 0.1 | 0 | 7196 | 0.1 | 0 | 5162 | 21.6 | 0 | 6648 | 0.3 |
| | 25 | 0.10 | 0 | 2520 | 0.0 | 65 | 2736 | 0.0 | 0 | 2747 | 0.0 | 0 | 2088 | 1.4 | 0 | 2467 | 0.0 |
| | | 0.50 | 0 | 4727 | 0.0 | 0 | 4290 | 0.1 | 0 | 4831 | 0.1 | 0 | 3662 | 9.8 | 0 | 4433 | 0.2 |
| | | 0.80 | 0 | 5600 | 0.1 | 2986 | 5690 | 0.1 | 0 | 6464 | 0.1 | 0 | 4651 | 18.8 | 0 | 5461 | 0.3 |
| | | 1.00 | 181 | 6649 | 0.1 | 5600 | 6907 | 0.1 | 0 | 7156 | 0.1 | 0 | 5271 | 33.5 | 0 | 6653 | 0.3 |
| | 26 | 0.10 | 0 | 2868 | 0.0 | 0 | 3052 | 0.0 | 0 | 3335 | 0.0 | 0 | 2450 | 0.6 | 0 | 2924 | 0.0 |
| | | 0.50 | 0 | 4437 | 0.0 | 60 | 4484 | 0.0 | 0 | 4707 | 0.0 | 0 | 3749 | 3.7 | 0 | 4513 | 0.1 |
| | | 0.80 | 55 | 5751 | 0.1 | 325 | 5625 | 0.1 | 0 | 6244 | 0.1 | 0 | 4585 | 17.9 | 0 | 5989 | 0.2 |
| | | 1.00 | 615 | 6491 | 0.1 | 2856 | 6141 | 0.1 | 0 | 6909 | 0.1 | 0 | 5214 | 21.8 | 0 | 6468 | 0.3 |
| | 27 | 0.10 | 0 | 2113 | 0.0 | 0 | 2321 | 0.0 | 0 | 2153 | 0.0 | 0 | 1745 | 0.3 | 0 | 2073 | 0.0 |
| | | 0.50 | 0 | 4514 | 0.0 | 0 | 4595 | 0.0 | 0 | 4961 | 0.0 | 0 | 3318 | 6.3 | 0 | 4433 | 0.1 |
| | | 0.80 | 0 | 5577 | 0.1 | 510 | 5624 | 0.1 | 0 | 6194 | 0.1 | 0 | 4417 | 14.1 | 0 | 5551 | 0.3 |
| | | 1.00 | 385 | 5960 | 0.1 | 952 | 5930 | 0.1 | 0 | 6580 | 0.1 | 280 | 4950 | 13.3 | 255 | 5641 | 2.1 |
| | 28 | 0.10 | 0 | 2790 | 0.0 | 0 | 2675 | 0.0 | 0 | 2976 | 0.0 | 0 | 2297 | 1.1 | 0 | 2902 | 0.0 |
| | | 0.50 | 0 | 5380 | 0.1 | 703 | 5108 | 0.1 | 0 | 5829 | 0.1 | 0 | 4358 | 15.7 | 0 | 5321 | 0.2 |
| | | 0.80 | 785 | 6060 | 0.1 | 902 | 5812 | 0.1 | 0 | 6634 | 0.1 | 0 | 4793 | 23.5 | 0 | 6058 | 0.3 |
| | | 1.00 | 1465 | 6874 | 0.1 | 3094 | 6528 | 0.1 | 0 | 7206 | 0.1 | 0 | 5312 | 38.4 | 0 | 6869 | 0.4 |
| | 29 | 0.10 | 0 | 2807 | 0.0 | 0 | 3021 | 0.0 | 0 | 3363 | 0.0 | 0 | 2526 | 1.1 | 0 | 3014 | 0.0 |
| | | 0.50 | 0 | 5076 | 0.1 | 0 | 4545 | 0.1 | 0 | 5566 | 0.1 | 0 | 3661 | 14.3 | 0 | 5198 | 0.2 |
| | | 0.80 | 0 | 5955 | 0.1 | 1923 | 5699 | 0.1 | 0 | 6959 | 0.1 | 0 | 4680 | 22.1 | 0 | 6167 | 0.3 |
| | | 1.00 | 240 | 6266 | 0.1 | 3814 | 6090 | 0.1 | 0 | 7390 | 0.1 | 0 | 5100 | 24.4 | 0 | 6537 | 0.4 |

Table A.10: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 200, instance numbers (Nr.) 0–14.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 200 | 0 | 0.10 | 0 | 2781 | 0.0 | 0 | 3139 | 0.0 | 0 | 3071 | 0.0 | 0 | 2415 | 1.3 | 0 | 2948 | 0.0 |
| | | 0.50 | 15 | 6499 | 0.1 | 738 | 7015 | 0.1 | 0 | 7063 | 0.1 | 0 | 4804 | 16.6 | 0 | 6683 | 0.2 |
| | | 0.80 | 70 | 7771 | 0.1 | 4537 | 8931 | 0.2 | 0 | 8984 | 0.1 | 0 | 6100 | 31.5 | 0 | 8033 | 0.4 |
| | | 1.00 | 1911 | 9075 | 0.2 | 9313 | 10248 | 0.2 | 0 | 10329 | 0.2 | 16 | 7008 | 46.3 | 0 | 9453 | 0.5 |
| | 1 | 0.10 | 0 | 2229 | 0.0 | 0 | 2804 | 0.0 | 0 | 2809 | 0.0 | 0 | 1915 | 1.2 | 0 | 2428 | 0.1 |
| | | 0.50 | 0 | 5705 | 0.1 | 1558 | 6056 | 0.1 | 152 | 6682 | 0.1 | 0 | 4276 | 25.7 | 0 | 5847 | 0.4 |
| | | 0.80 | 1782 | 7610 | 0.2 | 5283 | 7825 | 0.2 | 214 | 8383 | 0.2 | 0 | 6010 | 45.7 | 0 | 7443 | 0.9 |
| | | 1.00 | 6584 | 9578 | 0.2 | 11628 | 9646 | 0.3 | 2009 | 10351 | 0.2 | 291 | 7798 | 62.8 | 0 | 9357 | 2.8 |
| | 2 | 0.10 | 0 | 3032 | 0.0 | 0 | 3293 | 0.0 | 0 | 3685 | 0.0 | 0 | 2690 | 1.9 | 0 | 3351 | 0.1 |
| | | 0.50 | 0 | 5777 | 0.1 | 385 | 5646 | 0.1 | 0 | 6131 | 0.1 | 0 | 4326 | 16.4 | 0 | 5720 | 0.3 |
| | | 0.80 | 0 | 7261 | 0.2 | 1901 | 7329 | 0.2 | 0 | 8438 | 0.2 | 0 | 5715 | 32.3 | 0 | 7375 | 0.5 |
| | | 1.00 | 2647 | 9158 | 0.2 | 8931 | 8881 | 0.2 | 0 | 10935 | 0.2 | 0 | 7109 | 75.6 | 0 | 9390 | 2.7 |
| | 3 | 0.10 | 0 | 4387 | 0.0 | 527 | 4505 | 0.0 | 0 | 5014 | 0.0 | 0 | 3779 | 4.6 | 0 | 4788 | 0.1 |
| | | 0.50 | 100 | 7619 | 0.1 | 2832 | 7640 | 0.1 | 0 | 8340 | 0.1 | 0 | 5942 | 26.8 | 0 | 7922 | 0.7 |
| | | 0.80 | 270 | 9055 | 0.1 | 2138 | 8633 | 0.2 | 0 | 9769 | 0.2 | 0 | 7173 | 48.6 | 0 | 9354 | 1.0 |
| | | 1.00 | 393 | 10138 | 0.2 | 3829 | 9908 | 0.2 | 86 | 10612 | 0.2 | 140 | 8207 | 63.2 | 100 | 9155 | 10.9 |
| | 4 | 0.10 | 0 | 3356 | 0.0 | 0 | 3973 | 0.1 | 0 | 3930 | 0.0 | 0 | 3097 | 2.8 | 0 | 3740 | 0.1 |
| | | 0.50 | 0 | 6441 | 0.1 | 1110 | 7178 | 0.1 | 0 | 7071 | 0.1 | 0 | 5140 | 18.3 | 0 | 6675 | 0.2 |
| | | 0.80 | 0 | 8874 | 0.2 | 1618 | 8692 | 0.2 | 0 | 9774 | 0.2 | 0 | 6926 | 51.8 | 0 | 9445 | 0.5 |
| | | 1.00 | 2099 | 10999 | 0.2 | 4810 | 9913 | 0.2 | 0 | 11794 | 0.2 | 899 | 8622 | 78.5 | 0 | 11302 | 1.7 |
| | 5 | 0.10 | 0 | 3740 | 0.0 | 0 | 3779 | 0.0 | 0 | 4182 | 0.0 | 0 | 3230 | 2.7 | 0 | 4004 | 0.1 |
| | | 0.50 | 120 | 6923 | 0.1 | 1 | 7222 | 0.2 | 120 | 7687 | 0.1 | 0 | 5402 | 37.0 | 0 | 7163 | 0.3 |
| | | 0.80 | 1240 | 8773 | 0.2 | 3573 | 8914 | 0.3 | 140 | 9388 | 0.2 | 0 | 6881 | 76.2 | 40 | 7853 | 11.0 |
| | | 1.00 | 3029 | 10007 | 0.3 | 4080 | 9909 | 0.3 | 120 | 10895 | 0.3 | 382 | 8747 | 92.7 | 205 | 9135 | 15.9 |
| | 6 | 0.10 | 0 | 4577 | 0.0 | 90 | 4822 | 0.0 | 0 | 5297 | 0.0 | 0 | 3517 | 5.6 | 0 | 4919 | 0.1 |
| | | 0.50 | 0 | 7598 | 0.1 | 3860 | 7780 | 0.1 | 0 | 8231 | 0.1 | 0 | 5613 | 31.5 | 0 | 7765 | 0.4 |
| | | 0.80 | 739 | 9499 | 0.2 | 17089 | 10385 | 0.2 | 0 | 10569 | 0.2 | 0 | 7191 | 56.5 | 0 | 9553 | 0.6 |
| | | 1.00 | 4553 | 11680 | 0.2 | 32203 | 10890 | 0.3 | 169 | 11936 | 0.3 | 0 | 9246 | 59.7 | 0 | 10880 | 1.9 |
| | 7 | 0.10 | 0 | 3558 | 0.0 | 0 | 3927 | 0.0 | 0 | 4547 | 0.0 | 0 | 3108 | 3.0 | 0 | 4346 | 0.1 |
| | | 0.50 | 0 | 7712 | 0.1 | 1259 | 8374 | 0.1 | 0 | 8742 | 0.1 | 0 | 5698 | 27.7 | 0 | 8236 | 0.3 |
| | | 0.80 | 100 | 10352 | 0.2 | 3076 | 10421 | 0.2 | 0 | 11280 | 0.2 | 0 | 7586 | 63.8 | 0 | 10421 | 0.9 |
| | | 1.00 | 4007 | 11758 | 0.2 | 7036 | 11238 | 0.3 | 2278 | 12069 | 0.2 | 541 | 9019 | 92.9 | 1035 | 10310 | 15.2 |
| | 8 | 0.10 | 0 | 1873 | 0.0 | 0 | 2176 | 0.0 | 0 | 2046 | 0.0 | 0 | 1546 | 0.5 | 0 | 1971 | 0.0 |
| | | 0.50 | 0 | 6417 | 0.1 | 760 | 7049 | 0.1 | 0 | 7658 | 0.1 | 0 | 4899 | 24.1 | 0 | 6760 | 0.3 |
| | | 0.80 | 430 | 8930 | 0.2 | 2615 | 9901 | 0.2 | 261 | 10236 | 0.2 | 0 | 6726 | 49.2 | 0 | 9014 | 0.8 |
| | | 1.00 | 4614 | 11950 | 0.2 | 19800 | 11899 | 0.3 | 509 | 12330 | 0.3 | 0 | 8402 | 69.7 | 0 | 10764 | 1.1 |
| | 9 | 0.10 | 0 | 2428 | 0.0 | 0 | 2824 | 0.0 | 0 | 2765 | 0.0 | 0 | 2265 | 1.3 | 0 | 2661 | 0.1 |
| | | 0.50 | 25 | 6693 | 0.1 | 296 | 6868 | 0.1 | 25 | 7444 | 0.1 | 0 | 5216 | 33.3 | 25 | 6044 | 5.4 |
| | | 0.80 | 25 | 8198 | 0.2 | 3110 | 9068 | 0.3 | 25 | 8792 | 0.2 | 0 | 6332 | 69.7 | 25 | 7293 | 10.5 |
| | | 1.00 | 8678 | 9961 | 0.3 | 12752 | 9616 | 0.3 | 25 | 11106 | 0.3 | 135 | 8434 | 103.5 | 25 | 9157 | 15.0 |
| | 10 | 0.10 | 0 | 2140 | 0.0 | 0 | 2247 | 0.0 | 0 | 2388 | 0.0 | 0 | 1840 | 0.4 | 0 | 2324 | 0.0 |
| | | 0.50 | 0 | 6611 | 0.1 | 295 | 6635 | 0.1 | 0 | 7034 | 0.1 | 0 | 5045 | 17.2 | 0 | 6704 | 0.2 |
| | | 0.80 | 130 | 9180 | 0.1 | 1333 | 8967 | 0.2 | 0 | 9593 | 0.2 | 0 | 6911 | 47.8 | 0 | 8978 | 0.4 |
| | | 1.00 | 9394 | 10797 | 0.2 | 16382 | 10805 | 0.2 | 0 | 11413 | 0.2 | 0 | 8375 | 76.2 | 0 | 10563 | 1.9 |
| | 11 | 0.10 | 0 | 2449 | 0.0 | 0 | 2475 | 0.0 | 0 | 2893 | 0.0 | 0 | 2150 | 1.4 | 0 | 2653 | 0.1 |
| | | 0.50 | 250 | 6800 | 0.1 | 380 | 6825 | 0.1 | 0 | 7755 | 0.1 | 0 | 5101 | 35.9 | 0 | 7130 | 0.3 |
| | | 0.80 | 350 | 10157 | 0.2 | 11667 | 10463 | 0.3 | 0 | 11318 | 0.2 | 0 | 7943 | 79.5 | 0 | 10591 | 0.6 |
| | | 1.00 | 17462 | 11850 | 0.3 | 33827 | 11930 | 0.3 | 260 | 12920 | 0.3 | 140 | 8926 | 96.5 | 0 | 12127 | 0.8 |
| | 12 | 0.10 | 0 | 3200 | 0.0 | 45 | 3811 | 0.0 | 0 | 3806 | 0.0 | 0 | 2757 | 2.3 | 0 | 3631 | 0.1 |
| | | 0.50 | 0 | 6058 | 0.1 | 206 | 5988 | 0.1 | 0 | 6587 | 0.1 | 0 | 4839 | 19.8 | 0 | 6264 | 0.3 |
| | | 0.80 | 520 | 7670 | 0.2 | 4650 | 7444 | 0.2 | 0 | 8298 | 0.2 | 0 | 5813 | 45.2 | 0 | 7778 | 0.5 |
| | | 1.00 | 1374 | 9706 | 0.2 | 27009 | 9256 | 0.3 | 4768 | 10229 | 0.3 | 0 | 7488 | 71.4 | 0 | 9518 | 0.7 |
| | 13 | 0.10 | 0 | 2419 | 0.0 | 0 | 3275 | 0.0 | 0 | 2634 | 0.0 | 0 | 2231 | 0.9 | 0 | 2634 | 0.1 |
| | | 0.50 | 0 | 4928 | 0.1 | 615 | 5719 | 0.1 | 0 | 5392 | 0.1 | 0 | 4006 | 10.1 | 0 | 5128 | 0.2 |
| | | 0.80 | 0 | 7293 | 0.2 | 6790 | 8075 | 0.2 | 0 | 8728 | 0.2 | 0 | 5430 | 45.2 | 0 | 7677 | 0.5 |
| | | 1.00 | 11084 | 9999 | 0.2 | 18983 | 9255 | 0.3 | 0 | 10634 | 0.2 | 0 | 6829 | 73.2 | 0 | 9795 | 1.7 |
| | 14 | 0.10 | 0 | 3791 | 0.0 | 0 | 5439 | 0.1 | 0 | 4452 | 0.0 | 0 | 3487 | 4.5 | 0 | 4222 | 0.1 |
| | | 0.50 | 0 | 6231 | 0.1 | 40 | 7724 | 0.1 | 0 | 6908 | 0.1 | 0 | 4994 | 26.0 | 0 | 6363 | 0.3 |
| | | 0.80 | 600 | 8728 | 0.2 | 3419 | 10365 | 0.3 | 0 | 9801 | 0.2 | 0 | 7279 | 48.6 | 0 | 8650 | 0.6 |
| | | 1.00 | 3409 | 10386 | 0.3 | 5986 | 10436 | 0.3 | 0 | 10929 | 0.3 | 0 | 8046 | 65.2 | 0 | 9506 | 1.3 |

Table A.11: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 200, instance numbers (Nr.) 15–29.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 200 | 15 | 0.10 | 0 | 3101 | 0.0 | 0 | 3784 | 0.0 | 0 | 3096 | 0.0 | 0 | 2645 | 1.5 | 0 | 2966 | 0.1 |
| | | 0.50 | 0 | 5915 | 0.1 | 210 | 6156 | 0.1 | 0 | 6495 | 0.1 | 0 | 4545 | 22.7 | 0 | 6023 | 0.3 |
| | | 0.80 | 0 | 7723 | 0.2 | 1723 | 8040 | 0.2 | 0 | 8426 | 0.2 | 0 | 5770 | 46.7 | 0 | 7766 | 0.5 |
| | | 1.00 | 2295 | 9120 | 0.2 | 5349 | 9458 | 0.2 | 10 | 9858 | 0.2 | 45 | 7388 | 72.2 | 10 | 8008 | 13.0 |
| | 16 | 0.10 | 0 | 3021 | 0.0 | 0 | 3560 | 0.0 | 0 | 3647 | 0.0 | 0 | 2731 | 1.9 | 0 | 3477 | 0.1 |
| | | 0.50 | 0 | 5508 | 0.1 | 225 | 5593 | 0.1 | 0 | 6363 | 0.1 | 0 | 4313 | 16.3 | 0 | 5721 | 0.3 |
| | | 0.80 | 0 | 8416 | 0.2 | 3502 | 7661 | 0.2 | 0 | 9510 | 0.2 | 0 | 5656 | 67.9 | 0 | 8702 | 0.5 |
| | | 1.00 | 3588 | 10403 | 0.3 | 9093 | 9531 | 0.3 | 583 | 11989 | 0.3 | 395 | 7710 | 89.8 | 535 | 9241 | 16.0 |
| | 17 | 0.10 | 0 | 2381 | 0.0 | 0 | 2682 | 0.0 | 0 | 2962 | 0.0 | 0 | 1917 | 2.3 | 0 | 2684 | 0.1 |
| | | 0.50 | 0 | 7493 | 0.1 | 603 | 7155 | 0.2 | 0 | 8358 | 0.1 | 0 | 5491 | 39.8 | 0 | 7566 | 0.3 |
| | | 0.80 | 297 | 9681 | 0.2 | 3361 | 10053 | 0.3 | 0 | 11151 | 0.2 | 0 | 7390 | 68.0 | 0 | 9415 | 0.6 |
| | | 1.00 | 3826 | 11663 | 0.3 | 22415 | 11530 | 0.3 | 85 | 12826 | 0.3 | 0 | 8733 | 110.0 | 0 | 11426 | 3.4 |
| | 18 | 0.10 | 0 | 2763 | 0.0 | 0 | 2992 | 0.0 | 0 | 3070 | 0.0 | 0 | 2254 | 2.9 | 0 | 2865 | 0.1 |
| | | 0.50 | 0 | 6426 | 0.1 | 640 | 6827 | 0.1 | 0 | 6747 | 0.1 | 0 | 5074 | 25.1 | 0 | 6532 | 0.3 |
| | | 0.80 | 110 | 8632 | 0.2 | 4640 | 9351 | 0.3 | 0 | 9354 | 0.2 | 0 | 6773 | 65.0 | 0 | 8799 | 0.6 |
| | | 1.00 | 5073 | 9603 | 0.3 | 20267 | 10228 | 0.3 | 100 | 10157 | 0.3 | 0 | 7720 | 101.8 | 0 | 10024 | 2.4 |
| | 19 | 0.10 | 0 | 2649 | 0.0 | 0 | 3726 | 0.0 | 0 | 2926 | 0.0 | 0 | 2581 | 1.0 | 0 | 2824 | 0.1 |
| | | 0.50 | 0 | 5477 | 0.1 | 772 | 6578 | 0.1 | 0 | 5970 | 0.1 | 0 | 4464 | 11.7 | 0 | 5358 | 0.2 |
| | | 0.80 | 1245 | 8678 | 0.1 | 6243 | 8702 | 0.2 | 110 | 9785 | 0.2 | 0 | 6585 | 46.5 | 0 | 8825 | 0.8 |
| | | 1.00 | 2143 | 9615 | 0.2 | 8616 | 9980 | 0.2 | 148 | 10907 | 0.2 | 12 | 8061 | 59.4 | 0 | 9825 | 1.3 |
| | 20 | 0.10 | 0 | 3617 | 0.0 | 0 | 4096 | 0.1 | 0 | 4567 | 0.0 | 0 | 2716 | 3.7 | 0 | 3917 | 0.1 |
| | | 0.50 | 0 | 7468 | 0.1 | 1852 | 6701 | 0.1 | 0 | 7807 | 0.1 | 0 | 5109 | 39.0 | 0 | 7431 | 0.3 |
| | | 0.80 | 3975 | 9478 | 0.2 | 7573 | 8477 | 0.2 | 45 | 10479 | 0.2 | 25 | 6711 | 68.3 | 25 | 8575 | 8.4 |
| | | 1.00 | 11918 | 11079 | 0.3 | 17564 | 9841 | 0.3 | 470 | 11374 | 0.2 | 613 | 8528 | 78.3 | 623 | 9875 | 12.8 |
| | 21 | 0.10 | 0 | 3217 | 0.0 | 0 | 3500 | 0.0 | 0 | 3454 | 0.0 | 0 | 2617 | 2.8 | 0 | 3406 | 0.1 |
| | | 0.50 | 0 | 6330 | 0.1 | 1941 | 6682 | 0.1 | 0 | 6794 | 0.1 | 0 | 4919 | 29.2 | 0 | 6468 | 0.3 |
| | | 0.80 | 470 | 8652 | 0.2 | 975 | 8355 | 0.2 | 0 | 9463 | 0.2 | 0 | 6666 | 66.2 | 0 | 8760 | 0.6 |
| | | 1.00 | 6928 | 10102 | 0.2 | 11431 | 9306 | 0.3 | 315 | 10952 | 0.3 | 0 | 8050 | 78.9 | 0 | 9869 | 2.0 |
| | 22 | 0.10 | 0 | 3572 | 0.0 | 170 | 3636 | 0.0 | 0 | 4263 | 0.0 | 0 | 3144 | 2.5 | 0 | 4108 | 0.1 |
| | | 0.50 | 50 | 6925 | 0.1 | 1896 | 7194 | 0.2 | 0 | 7340 | 0.1 | 0 | 5413 | 32.1 | 0 | 7092 | 0.4 |
| | | 0.80 | 355 | 8576 | 0.2 | 2435 | 8970 | 0.2 | 0 | 9241 | 0.2 | 0 | 6783 | 58.9 | 0 | 8501 | 0.6 |
| | | 1.00 | 2747 | 10086 | 0.3 | 11568 | 9796 | 0.3 | 579 | 11023 | 0.3 | 0 | 7872 | 104.7 | 165 | 9116 | 15.8 |
| | 23 | 0.10 | 0 | 3933 | 0.0 | 0 | 4262 | 0.0 | 0 | 4242 | 0.0 | 0 | 3127 | 2.5 | 0 | 3964 | 0.1 |
| | | 0.50 | 0 | 7767 | 0.1 | 0 | 8269 | 0.1 | 0 | 8534 | 0.1 | 0 | 5625 | 28.6 | 0 | 8145 | 0.3 |
| | | 0.80 | 155 | 9409 | 0.2 | 3118 | 10009 | 0.2 | 0 | 10373 | 0.2 | 0 | 7196 | 63.0 | 0 | 9617 | 0.5 |
| | | 1.00 | 2579 | 11244 | 0.2 | 5574 | 11922 | 0.2 | 457 | 12530 | 0.2 | 175 | 9771 | 74.5 | 175 | 10804 | 10.1 |
| | 24 | 0.10 | 0 | 2643 | 0.0 | 0 | 2680 | 0.0 | 0 | 3063 | 0.0 | 0 | 2231 | 1.7 | 0 | 2875 | 0.1 |
| | | 0.50 | 0 | 6723 | 0.1 | 1323 | 6526 | 0.1 | 0 | 7494 | 0.1 | 0 | 5252 | 23.3 | 0 | 6912 | 0.3 |
| | | 0.80 | 130 | 8396 | 0.2 | 6897 | 8262 | 0.2 | 0 | 9253 | 0.2 | 0 | 6101 | 56.7 | 0 | 8395 | 0.5 |
| | | 1.00 | 7567 | 9878 | 0.2 | 15560 | 10260 | 0.3 | 10 | 11128 | 0.3 | 85 | 8506 | 68.7 | 0 | 10469 | 2.5 |
| | 25 | 0.10 | 0 | 3635 | 0.0 | 0 | 3623 | 0.0 | 0 | 4103 | 0.0 | 0 | 2976 | 3.2 | 0 | 3857 | 0.1 |
| | | 0.50 | 1751 | 7172 | 0.1 | 1958 | 7917 | 0.1 | 1775 | 8002 | 0.1 | 0 | 5192 | 34.0 | 0 | 7196 | 0.3 |
| | | 0.80 | 2721 | 9022 | 0.2 | 15645 | 9305 | 0.2 | 1781 | 9882 | 0.2 | 40 | 6483 | 56.6 | 0 | 8591 | 1.6 |
| | | 1.00 | 16072 | 9962 | 0.2 | 41028 | 10234 | 0.2 | 2372 | 10797 | 0.2 | 130 | 7333 | 65.7 | 0 | 9379 | 1.9 |
| | 26 | 0.10 | 0 | 3283 | 0.0 | 0 | 3916 | 0.0 | 0 | 3637 | 0.0 | 0 | 3027 | 2.2 | 0 | 3547 | 0.1 |
| | | 0.50 | 275 | 5500 | 0.1 | 550 | 5864 | 0.1 | 0 | 6108 | 0.1 | 0 | 4742 | 17.1 | 0 | 5853 | 0.3 |
| | | 0.80 | 2102 | 8641 | 0.2 | 5852 | 8351 | 0.2 | 1134 | 8842 | 0.2 | 0 | 6516 | 48.8 | 0 | 8607 | 0.6 |
| | | 1.00 | 5909 | 9427 | 0.2 | 19286 | 9415 | 0.3 | 1594 | 10481 | 0.3 | 1611 | 7839 | 60.8 | 0 | 9382 | 3.0 |
| | 27 | 0.10 | 0 | 2898 | 0.0 | 0 | 2986 | 0.0 | 0 | 3166 | 0.0 | 0 | 2561 | 1.1 | 0 | 3080 | 0.0 |
| | | 0.50 | 0 | 6054 | 0.1 | 80 | 6208 | 0.1 | 0 | 6162 | 0.1 | 0 | 4740 | 22.9 | 0 | 6129 | 0.3 |
| | | 0.80 | 0 | 7820 | 0.2 | 926 | 7950 | 0.2 | 0 | 8564 | 0.2 | 0 | 6095 | 51.4 | 0 | 7973 | 0.5 |
| | | 1.00 | 2153 | 9057 | 0.2 | 9719 | 8701 | 0.3 | 108 | 9768 | 0.2 | 0 | 7005 | 85.1 | 0 | 9116 | 2.0 |
| | 28 | 0.10 | 0 | 3278 | 0.0 | 0 | 3501 | 0.0 | 0 | 3676 | 0.0 | 0 | 2678 | 1.8 | 0 | 3454 | 0.1 |
| | | 0.50 | 0 | 5730 | 0.1 | 62 | 5972 | 0.1 | 0 | 6506 | 0.1 | 0 | 4476 | 11.9 | 0 | 5819 | 0.2 |
| | | 0.80 | 1456 | 8479 | 0.1 | 3647 | 8300 | 0.2 | 269 | 9061 | 0.2 | 0 | 5884 | 38.0 | 0 | 8382 | 0.4 |
| | | 1.00 | 9279 | 10623 | 0.2 | 21759 | 10169 | 0.2 | 496 | 11073 | 0.2 | 1255 | 8245 | 65.6 | 80 | 9000 | 13.8 |
| | 29 | 0.10 | 0 | 3421 | 0.0 | 0 | 3647 | 0.1 | 0 | 3865 | 0.0 | 0 | 2452 | 4.1 | 0 | 3571 | 0.1 |
| | | 0.50 | 0 | 7206 | 0.1 | 688 | 7998 | 0.2 | 0 | 7437 | 0.1 | 0 | 4874 | 30.4 | 0 | 7091 | 0.3 |
| | | 0.80 | 194 | 9262 | 0.2 | 2715 | 9810 | 0.3 | 0 | 10150 | 0.2 | 0 | 6823 | 56.2 | 0 | 9158 | 0.6 |
| | | 1.00 | 1084 | 10443 | 0.3 | 9158 | 10802 | 0.3 | 0 | 10918 | 0.3 | 30 | 8133 | 83.5 | 30 | 9182 | 13.4 |

Table A.12: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 500, instance numbers (Nr.) 0–14.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 500 | 0 | 0.10 | 0 | 4639 | 0.1 | 897 | 6702 | 0.2 | 0 | 5318 | 0.1 | 0 | 3991 | 14.4 | 0 | 4869 | 0.3 |
| | | 0.50 | 0 | 9322 | 0.3 | 2696 | 11370 | 0.4 | 0 | 10397 | 0.3 | 0 | 7148 | 106.0 | 0 | 9593 | 0.7 |
| | | 0.80 | 3395 | 12835 | 0.5 | 7780 | 15754 | 0.7 | 0 | 14895 | 0.5 | 0 | 9912 | 221.6 | 0 | 13114 | 1.5 |
| | | 1.00 | 16483 | 16274 | 0.8 | 34288 | 19718 | 1.0 | 3538 | 18925 | 0.8 | 1767 | 13745 | 385.7 | 26 | 14452 | 101.8 |
| | 1 | 0.10 | 0 | 4382 | 0.1 | 2235 | 5523 | 0.1 | 0 | 5021 | 0.0 | 0 | 4060 | 4.1 | 0 | 4799 | 0.2 |
| | | 0.50 | 145 | 9123 | 0.2 | 2974 | 10614 | 0.4 | 0 | 10409 | 0.2 | 0 | 7231 | 63.8 | 0 | 9645 | 0.6 |
| | | 0.80 | 1264 | 13307 | 0.4 | 7211 | 13727 | 0.5 | 270 | 13953 | 0.4 | 0 | 9135 | 198.9 | 0 | 10688 | 31.0 |
| | | 1.00 | 9320 | 15518 | 0.6 | 15351 | 16821 | 0.7 | 1139 | 17511 | 0.6 | 1797 | 12164 | 312.4 | 37 | 13361 | 75.7 |
| | 2 | 0.10 | 0 | 4213 | 0.1 | 2295 | 5164 | 0.1 | 0 | 4771 | 0.1 | 0 | 3563 | 9.0 | 0 | 4644 | 0.2 |
| | | 0.50 | 0 | 8279 | 0.2 | 1694 | 11026 | 0.4 | 0 | 9983 | 0.2 | 0 | 6749 | 63.7 | 0 | 9007 | 0.7 |
| | | 0.80 | 2459 | 13825 | 0.5 | 6911 | 16454 | 0.7 | 0 | 15069 | 0.5 | 0 | 9808 | 230.3 | 0 | 14273 | 1.4 |
| | | 1.00 | 16322 | 17531 | 0.8 | 26042 | 19283 | 1.0 | 1987 | 19586 | 0.7 | 718 | 13580 | 418.7 | 0 | 16982 | 29.8 |
| | 3 | 0.10 | 0 | 3569 | 0.1 | 170 | 3703 | 0.1 | 0 | 4012 | 0.0 | 0 | 3153 | 5.7 | 0 | 3822 | 0.1 |
| | | 0.50 | 0 | 7746 | 0.2 | 994 | 9867 | 0.3 | 0 | 8426 | 0.2 | 0 | 6142 | 52.4 | 0 | 7861 | 0.5 |
| | | 0.80 | 0 | 13022 | 0.5 | 7621 | 14865 | 0.6 | 0 | 13986 | 0.5 | 0 | 9798 | 193.1 | 0 | 12658 | 1.3 |
| | | 1.00 | 4795 | 16388 | 0.7 | 25696 | 17495 | 0.9 | 255 | 18825 | 0.7 | 390 | 12850 | 386.0 | 0 | 16322 | 10.9 |
| | 4 | 0.10 | 0 | 2728 | 0.1 | 0 | 4108 | 0.1 | 0 | 3405 | 0.0 | 0 | 2729 | 2.5 | 0 | 3262 | 0.1 |
| | | 0.50 | 0 | 8967 | 0.3 | 5901 | 10319 | 0.5 | 0 | 9980 | 0.3 | 0 | 6848 | 87.9 | 0 | 9231 | 0.8 |
| | | 0.80 | 45 | 12775 | 0.6 | 3284 | 14316 | 0.6 | 0 | 13575 | 0.5 | 0 | 9475 | 193.7 | 0 | 12261 | 1.4 |
| | | 1.00 | 5868 | 16026 | 0.8 | 10654 | 16537 | 0.9 | 5 | 16855 | 0.7 | 92 | 12075 | 348.9 | 5 | 12859 | 61.0 |
| | 5 | 0.10 | 0 | 3420 | 0.1 | 160 | 4547 | 0.1 | 0 | 3982 | 0.1 | 0 | 2971 | 5.6 | 0 | 3753 | 0.2 |
| | | 0.50 | 0 | 10418 | 0.4 | 3766 | 12711 | 0.5 | 0 | 11563 | 0.3 | 0 | 7368 | 168.4 | 0 | 11015 | 0.9 |
| | | 0.80 | 4851 | 14490 | 0.6 | 21752 | 15667 | 0.8 | 20 | 16297 | 0.6 | 2530 | 10558 | 335.8 | 0 | 14213 | 8.8 |
| | | 1.00 | 16673 | 17427 | 0.8 | 28631 | 17898 | 0.9 | 1079 | 18720 | 0.7 | 3472 | 13089 | 537.1 | 715 | 14721 | 89.5 |
| | 6 | 0.10 | 0 | 3757 | 0.1 | 541 | 4389 | 0.1 | 0 | 4367 | 0.0 | 0 | 3412 | 6.1 | 0 | 4201 | 0.2 |
| | | 0.50 | 5 | 9320 | 0.3 | 1458 | 11341 | 0.4 | 5 | 10598 | 0.2 | 0 | 7308 | 94.4 | 0 | 10059 | 0.7 |
| | | 0.80 | 886 | 12920 | 0.5 | 11505 | 14845 | 0.8 | 5 | 14779 | 0.5 | 0 | 9742 | 233.8 | 40 | 11393 | 40.8 |
| | | 1.00 | 8250 | 16495 | 0.8 | 28714 | 18642 | 1.0 | 544 | 18350 | 0.8 | 1006 | 13171 | 395.9 | 220 | 14095 | 97.6 |
| | 7 | 0.10 | 0 | 3763 | 0.1 | 520 | 4604 | 0.1 | 0 | 4290 | 0.0 | 0 | 3414 | 4.7 | 0 | 4197 | 0.1 |
| | | 0.50 | 12 | 9619 | 0.3 | 8211 | 12100 | 0.4 | 419 | 10680 | 0.3 | 0 | 8008 | 80.7 | 12 | 8978 | 11.4 |
| | | 0.80 | 5255 | 15456 | 0.7 | 14600 | 17041 | 0.8 | 3561 | 17611 | 0.6 | 0 | 11983 | 338.6 | 12 | 13023 | 56.8 |
| | | 1.00 | 12441 | 18699 | 0.9 | 19709 | 19006 | 1.0 | 3909 | 20477 | 0.8 | 321 | 15659 | 511.3 | 262 | 16255 | 103.4 |
| | 8 | 0.10 | 0 | 3607 | 0.1 | 0 | 5400 | 0.1 | 0 | 4893 | 0.1 | 0 | 3219 | 10.3 | 0 | 4505 | 0.2 |
| | | 0.50 | 0 | 9553 | 0.4 | 192 | 12537 | 0.6 | 0 | 10879 | 0.4 | 0 | 7036 | 122.8 | 0 | 9856 | 0.9 |
| | | 0.80 | 2349 | 15746 | 0.8 | 19266 | 18420 | 1.1 | 0 | 17910 | 0.7 | 0 | 11398 | 363.0 | 0 | 15925 | 1.9 |
| | | 1.00 | 19908 | 19533 | 1.0 | 30225 | 20327 | 1.2 | 906 | 22797 | 1.0 | 456 | 15974 | 559.5 | 20 | 16041 | 109.8 |
| | 9 | 0.10 | 0 | 2923 | 0.1 | 1147 | 4028 | 0.1 | 0 | 3622 | 0.0 | 0 | 2541 | 4.5 | 0 | 3279 | 0.1 |
| | | 0.50 | 20 | 8164 | 0.3 | 3747 | 9209 | 0.3 | 20 | 9104 | 0.2 | 0 | 5836 | 88.4 | 0 | 8489 | 0.6 |
| | | 0.80 | 3052 | 12907 | 0.5 | 15420 | 14233 | 0.6 | 739 | 13599 | 0.4 | 0 | 8655 | 209.6 | 0 | 12698 | 2.0 |
| | | 1.00 | 9392 | 15606 | 0.6 | 21374 | 16239 | 0.8 | 4967 | 17645 | 0.6 | 1215 | 12149 | 323.3 | 2220 | 13142 | 65.3 |
| | 10 | 0.10 | 0 | 3623 | 0.1 | 150 | 5686 | 0.1 | 0 | 4438 | 0.0 | 0 | 3192 | 6.7 | 0 | 4007 | 0.2 |
| | | 0.50 | 0 | 8586 | 0.3 | 2533 | 11985 | 0.5 | 0 | 10208 | 0.3 | 0 | 6117 | 99.1 | 0 | 9055 | 0.8 |
| | | 0.80 | 344 | 13430 | 0.6 | 9302 | 16689 | 0.7 | 214 | 15785 | 0.5 | 0 | 10272 | 239.0 | 0 | 13936 | 1.5 |
| | | 1.00 | 20951 | 17789 | 0.8 | 25223 | 18480 | 0.9 | 1982 | 20284 | 0.8 | 825 | 13128 | 463.8 | 385 | 14340 | 107.0 |
| | 11 | 0.10 | 0 | 4104 | 0.1 | 140 | 4510 | 0.1 | 0 | 4616 | 0.0 | 0 | 3567 | 9.8 | 0 | 4482 | 0.2 |
| | | 0.50 | 0 | 10219 | 0.3 | 1206 | 12049 | 0.4 | 0 | 10925 | 0.3 | 0 | 8037 | 90.3 | 0 | 10256 | 0.8 |
| | | 0.80 | 778 | 12513 | 0.5 | 3404 | 13959 | 0.6 | 75 | 14083 | 0.4 | 0 | 9849 | 174.5 | 0 | 12549 | 1.3 |
| | | 1.00 | 5612 | 15057 | 0.6 | 16226 | 16281 | 0.7 | 7489 | 17062 | 0.6 | 364 | 12588 | 249.7 | 3090 | 13441 | 53.6 |
| | 12 | 0.10 | 0 | 3760 | 0.1 | 95 | 4123 | 0.1 | 0 | 3948 | 0.0 | 0 | 3279 | 3.2 | 0 | 3876 | 0.1 |
| | | 0.50 | 435 | 10761 | 0.4 | 3352 | 12110 | 0.5 | 0 | 12073 | 0.3 | 0 | 7699 | 118.4 | 0 | 10947 | 0.9 |
| | | 0.80 | 7555 | 14944 | 0.6 | 16467 | 15788 | 0.7 | 3001 | 16505 | 0.5 | 2360 | 10995 | 284.8 | 30 | 13543 | 43.0 |
| | | 1.00 | 19224 | 17711 | 0.8 | 45729 | 17485 | 0.8 | 7480 | 19377 | 0.7 | 5911 | 14612 | 389.0 | 170 | 15902 | 90.9 |
| | 13 | 0.10 | 0 | 3197 | 0.1 | 3435 | 4570 | 0.1 | 0 | 3681 | 0.0 | 0 | 2901 | 2.4 | 0 | 3387 | 0.1 |
| | | 0.50 | 0 | 8910 | 0.3 | 3301 | 9558 | 0.4 | 0 | 10247 | 0.3 | 0 | 6097 | 105.2 | 0 | 9202 | 0.8 |
| | | 0.80 | 491 | 12995 | 0.5 | 11789 | 12357 | 0.6 | 170 | 13764 | 0.5 | 0 | 9074 | 210.5 | 0 | 13040 | 1.3 |
| | | 1.00 | 6984 | 17009 | 0.7 | 27600 | 16633 | 0.9 | 501 | 18842 | 0.7 | 1500 | 12886 | 389.8 | 224 | 14647 | 80.6 |
| | 14 | 0.10 | 0 | 4163 | 0.1 | 539 | 5325 | 0.1 | 0 | 4545 | 0.0 | 0 | 3661 | 6.9 | 0 | 4378 | 0.1 |
| | | 0.50 | 0 | 10190 | 0.3 | 5114 | 12951 | 0.5 | 0 | 11211 | 0.3 | 0 | 7528 | 115.8 | 0 | 10400 | 0.7 |
| | | 0.80 | 3412 | 15201 | 0.6 | 17179 | 17470 | 0.8 | 17 | 17351 | 0.6 | 770 | 11480 | 255.7 | 620 | 12310 | 50.1 |
| | | 1.00 | 21324 | 19392 | 0.9 | 39018 | 20434 | 1.1 | 1611 | 22932 | 0.9 | 1820 | 16084 | 561.5 | 645 | 16859 | 116.1 |

Table A.13: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 500, instance numbers (Nr.) 15–29.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 500 | 15 | 0.10 | 0 | 4337 | 0.1 | 0 | 5984 | 0.1 | 0 | 5471 | 0.1 | 0 | 3833 | 11.3 | 0 | 5204 | 0.2 |
| | | 0.50 | 0 | 8564 | 0.3 | 1701 | 10815 | 0.5 | 0 | 10023 | 0.3 | 0 | 6338 | 98.9 | 0 | 8986 | 0.7 |
| | | 0.80 | 822 | 13489 | 0.5 | 9295 | 15193 | 0.8 | 308 | 15349 | 0.5 | 415 | 10391 | 207.0 | 100 | 12308 | 33.9 |
| | | 1.00 | 18419 | 18031 | 0.8 | 34280 | 18153 | 1.0 | 3129 | 19613 | 0.8 | 2238 | 14331 | 419.2 | 403 | 15751 | 85.5 |
| | 16 | 0.10 | 0 | 3922 | 0.1 | 190 | 4793 | 0.1 | 0 | 4532 | 0.0 | 0 | 3078 | 5.9 | 0 | 4402 | 0.2 |
| | | 0.50 | 0 | 9070 | 0.3 | 1328 | 11054 | 0.4 | 0 | 10079 | 0.2 | 0 | 6445 | 77.0 | 0 | 9295 | 0.7 |
| | | 0.80 | 0 | 12612 | 0.5 | 4375 | 15005 | 0.6 | 0 | 14224 | 0.5 | 0 | 9235 | 214.8 | 0 | 12815 | 1.4 |
| | | 1.00 | 6602 | 16230 | 0.7 | 13534 | 18495 | 0.9 | 1345 | 17828 | 0.7 | 2520 | 13264 | 389.1 | 10 | 14113 | 91.6 |
| | 17 | 0.10 | 0 | 2567 | 0.1 | 0 | 3382 | 0.1 | 0 | 2955 | 0.0 | 0 | 2269 | 3.2 | 0 | 2890 | 0.1 |
| | | 0.50 | 17 | 7829 | 0.2 | 5112 | 9467 | 0.3 | 5 | 8516 | 0.2 | 5 | 5683 | 54.6 | 5 | 6540 | 12.1 |
| | | 0.80 | 17 | 11091 | 0.4 | 12934 | 12166 | 0.5 | 5 | 12464 | 0.4 | 5 | 7910 | 128.8 | 5 | 9627 | 28.2 |
| | | 1.00 | 3856 | 13976 | 0.5 | 33832 | 14764 | 0.7 | 938 | 16571 | 0.5 | 148 | 11585 | 203.7 | 143 | 12254 | 48.5 |
| | 18 | 0.10 | 0 | 3925 | 0.1 | 0 | 5084 | 0.1 | 0 | 4581 | 0.1 | 0 | 3588 | 6.6 | 0 | 4357 | 0.2 |
| | | 0.50 | 0 | 9346 | 0.3 | 424 | 11412 | 0.5 | 0 | 10631 | 0.3 | 0 | 6776 | 113.6 | 0 | 9683 | 0.9 |
| | | 0.80 | 783 | 14895 | 0.6 | 3583 | 16682 | 0.8 | 98 | 17355 | 0.7 | 0 | 10124 | 340.5 | 0 | 15150 | 1.8 |
| | | 1.00 | 8189 | 19675 | 1.0 | 25385 | 19737 | 1.2 | 1342 | 21090 | 1.0 | 989 | 14257 | 704.4 | 650 | 16383 | 119.4 |
| | 19 | 0.10 | 0 | 3746 | 0.1 | 445 | 4754 | 0.1 | 0 | 4561 | 0.0 | 0 | 3249 | 8.5 | 0 | 4401 | 0.2 |
| | | 0.50 | 0 | 9705 | 0.4 | 1283 | 11328 | 0.5 | 0 | 11099 | 0.3 | 0 | 7442 | 141.8 | 0 | 9819 | 1.0 |
| | | 0.80 | 2328 | 15313 | 0.8 | 13788 | 15842 | 0.9 | 0 | 17020 | 0.7 | 0 | 10744 | 383.9 | 0 | 15392 | 4.5 |
| | | 1.00 | 22383 | 19932 | 1.1 | 24383 | 20815 | 1.2 | 1454 | 21649 | 1.0 | 406 | 16481 | 659.6 | 10 | 17407 | 128.3 |
| | 20 | 0.10 | 0 | 3720 | 0.1 | 872 | 4680 | 0.1 | 0 | 4587 | 0.0 | 0 | 3304 | 9.7 | 0 | 4393 | 0.2 |
| | | 0.50 | 0 | 9524 | 0.3 | 2766 | 11041 | 0.5 | 0 | 10252 | 0.3 | 0 | 7394 | 100.6 | 0 | 9819 | 0.8 |
| | | 0.80 | 140 | 13534 | 0.6 | 7417 | 15145 | 0.8 | 5 | 15010 | 0.5 | 0 | 9603 | 252.7 | 0 | 13754 | 1.4 |
| | | 1.00 | 10625 | 17590 | 0.9 | 29717 | 18913 | 1.0 | 4471 | 19531 | 0.8 | 1188 | 13498 | 492.1 | 140 | 14724 | 99.8 |
| | 21 | 0.10 | 0 | 4110 | 0.1 | 0 | 4754 | 0.1 | 0 | 4601 | 0.0 | 0 | 3245 | 6.6 | 0 | 4170 | 0.2 |
| | | 0.50 | 20 | 8759 | 0.3 | 9236 | 11794 | 0.4 | 20 | 9953 | 0.2 | 20 | 6527 | 65.8 | 20 | 8207 | 10.6 |
| | | 0.80 | 1719 | 13366 | 0.4 | 16118 | 14810 | 0.6 | 558 | 16133 | 0.5 | 1055 | 10042 | 162.5 | 981 | 11649 | 35.7 |
| | | 1.00 | 10421 | 17057 | 0.6 | 38611 | 18385 | 0.8 | 2376 | 19933 | 0.7 | 3188 | 13273 | 345.3 | 2450 | 14595 | 67.4 |
| | 22 | 0.10 | 0 | 2848 | 0.1 | 105 | 4504 | 0.1 | 0 | 3519 | 0.0 | 0 | 2547 | 4.9 | 0 | 3409 | 0.1 |
| | | 0.50 | 33 | 8694 | 0.3 | 3434 | 10729 | 0.4 | 33 | 10256 | 0.3 | 33 | 6045 | 78.1 | 33 | 7219 | 14.0 |
| | | 0.80 | 2914 | 12548 | 0.5 | 6969 | 13449 | 0.6 | 1007 | 13975 | 0.5 | 53 | 9059 | 171.6 | 193 | 10618 | 34.7 |
| | | 1.00 | 9597 | 14935 | 0.7 | 23708 | 16508 | 0.8 | 1795 | 17428 | 0.6 | 993 | 12239 | 275.4 | 33 | 13119 | 66.4 |
| | 23 | 0.10 | 0 | 4083 | 0.1 | 0 | 4985 | 0.1 | 0 | 4694 | 0.1 | 0 | 3483 | 15.0 | 0 | 4561 | 0.2 |
| | | 0.50 | 0 | 9658 | 0.4 | 1269 | 10183 | 0.5 | 0 | 10429 | 0.3 | 0 | 6930 | 110.2 | 0 | 9562 | 0.9 |
| | | 0.80 | 444 | 13460 | 0.6 | 10164 | 13810 | 0.8 | 290 | 14859 | 0.6 | 194 | 10391 | 243.8 | 0 | 13262 | 7.4 |
| | | 1.00 | 29457 | 17271 | 0.9 | 39710 | 17403 | 1.1 | 3871 | 20132 | 0.9 | 14050 | 14235 | 539.6 | 315 | 15032 | 122.1 |
| | 24 | 0.10 | 0 | 3263 | 0.1 | 250 | 4395 | 0.1 | 0 | 4001 | 0.1 | 0 | 3044 | 5.4 | 0 | 3890 | 0.2 |
| | | 0.50 | 0 | 8513 | 0.4 | 1363 | 9432 | 0.5 | 0 | 9538 | 0.3 | 0 | 6397 | 109.4 | 0 | 9006 | 0.9 |
| | | 0.80 | 75 | 12952 | 0.6 | 5554 | 12786 | 0.7 | 15 | 14740 | 0.6 | 0 | 8971 | 291.1 | 15 | 10842 | 45.8 |
| | | 1.00 | 25367 | 17170 | 0.9 | 36228 | 16999 | 1.0 | 267 | 19446 | 0.8 | 104 | 13225 | 500.7 | 40 | 15004 | 86.8 |
| | 25 | 0.10 | 0 | 4055 | 0.1 | 859 | 5614 | 0.2 | 0 | 5008 | 0.1 | 0 | 3630 | 12.4 | 0 | 4822 | 0.2 |
| | | 0.50 | 0 | 10040 | 0.4 | 2762 | 12273 | 0.5 | 0 | 11636 | 0.3 | 0 | 7668 | 101.0 | 0 | 10443 | 0.9 |
| | | 0.80 | 2290 | 15194 | 0.6 | 10050 | 17291 | 0.8 | 390 | 18089 | 0.6 | 5 | 11519 | 277.9 | 485 | 13489 | 47.3 |
| | | 1.00 | 8822 | 18437 | 0.8 | 18950 | 20014 | 1.0 | 1477 | 21844 | 0.9 | 135 | 14532 | 508.8 | 520 | 16206 | 98.0 |
| | 26 | 0.10 | 0 | 3366 | 0.1 | 2326 | 5271 | 0.1 | 0 | 4184 | 0.0 | 0 | 2964 | 6.6 | 0 | 4081 | 0.2 |
| | | 0.50 | 75 | 10152 | 0.4 | 3026 | 10883 | 0.5 | 0 | 11443 | 0.3 | 0 | 7263 | 125.7 | 0 | 10593 | 1.0 |
| | | 0.80 | 5587 | 14590 | 0.6 | 18790 | 15660 | 0.8 | 230 | 16986 | 0.6 | 0 | 10760 | 297.0 | 0 | 14108 | 5.0 |
| | | 1.00 | 23692 | 17156 | 0.8 | 39414 | 15962 | 0.9 | 1069 | 19577 | 0.8 | 1612 | 13051 | 445.0 | 500 | 14460 | 79.7 |
| | 27 | 0.10 | 0 | 3876 | 0.1 | 1549 | 6053 | 0.2 | 0 | 4903 | 0.0 | 0 | 3619 | 10.1 | 0 | 4554 | 0.2 |
| | | 0.50 | 0 | 8821 | 0.2 | 4299 | 11196 | 0.4 | 0 | 10770 | 0.2 | 0 | 6712 | 71.5 | 0 | 9535 | 0.6 |
| | | 0.80 | 4348 | 14665 | 0.5 | 16860 | 15164 | 0.6 | 0 | 16322 | 0.5 | 0 | 10745 | 201.3 | 540 | 11893 | 40.7 |
| | | 1.00 | 18469 | 16927 | 0.7 | 29088 | 18139 | 0.8 | 1209 | 19218 | 0.7 | 1297 | 14397 | 293.8 | 0 | 17175 | 18.2 |
| | 28 | 0.10 | 0 | 3731 | 0.1 | 75 | 4690 | 0.1 | 0 | 4331 | 0.0 | 0 | 3099 | 7.2 | 0 | 4132 | 0.1 |
| | | 0.50 | 0 | 9044 | 0.3 | 7452 | 11418 | 0.5 | 0 | 11149 | 0.3 | 0 | 6582 | 103.9 | 0 | 9516 | 0.8 |
| | | 0.80 | 4672 | 14631 | 0.6 | 14427 | 15100 | 0.7 | 470 | 16131 | 0.5 | 2791 | 10204 | 282.7 | 165 | 12498 | 44.4 |
| | | 1.00 | 14827 | 17823 | 0.9 | 31728 | 17750 | 1.0 | 2965 | 18709 | 0.8 | 5457 | 13473 | 469.4 | 1650 | 15089 | 80.5 |
| | 29 | 0.10 | 0 | 4423 | 0.1 | 0 | 5809 | 0.2 | 0 | 5261 | 0.1 | 0 | 4019 | 10.5 | 0 | 5032 | 0.2 |
| | | 0.50 | 0 | 8437 | 0.3 | 990 | 10250 | 0.5 | 0 | 9194 | 0.3 | 0 | 6672 | 73.8 | 0 | 8731 | 0.7 |
| | | 0.80 | 0 | 12656 | 0.6 | 8556 | 14821 | 0.8 | 0 | 13500 | 0.6 | 0 | 9740 | 222.2 | 0 | 13124 | 1.5 |
| | | 1.00 | 8686 | 16546 | 0.9 | 24144 | 18253 | 1.1 | 1325 | 18572 | 0.9 | 2916 | 13776 | 340.1 | 0 | 16853 | 39.8 |

Table A.14: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 1000, instance numbers (Nr.) 0–14.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 1000 | 0 | 0.10 | 0 | 5196 | 0.3 | 0 | 6776 | 0.4 | 0 | 6322 | 0.1 | 0 | 4110 | 40.7 | 0 | 5788 | 0.5 |
| | | 0.50 | 41 | 15071 | 0.8 | 7216 | 18404 | 1.2 | 0 | 17968 | 0.7 | 0 | 10740 | 402.4 | 0 | 15908 | 2.1 |
| | | 0.80 | 3663 | 22502 | 1.5 | 17413 | 24043 | 2.0 | 251 | 25885 | 1.3 | 200 | 17226 | 1003.8 | 165 | 18760 | 183.1 |
| | | 1.00 | 20216 | 29398 | 2.1 | 29304 | 29570 | 2.5 | 2177 | 33235 | 2.0 | 1651 | 26417 | 1016.0 | 727 | 26087 | 473.6 |
| | 1 | 0.10 | 0 | 4094 | 0.2 | 466 | 6048 | 0.4 | 0 | 5319 | 0.1 | 0 | 3547 | 13.1 | 0 | 4717 | 0.4 |
| | | 0.50 | 15 | 12987 | 0.7 | 2033 | 16765 | 1.1 | 130 | 14191 | 0.5 | 0 | 9121 | 237.5 | 0 | 12703 | 5.0 |
| | | 0.80 | 2433 | 21052 | 1.3 | 8171 | 23812 | 1.7 | 493 | 22946 | 1.1 | 240 | 15189 | 821.6 | 180 | 16934 | 140.1 |
| | | 1.00 | 16846 | 26477 | 1.9 | 51031 | 29050 | 2.5 | 2159 | 30554 | 1.6 | 1637 | 23646 | 1007.4 | 888 | 23011 | 300.8 |
| | 2 | 0.10 | 0 | 5091 | 0.2 | 0 | 6681 | 0.3 | 0 | 6139 | 0.1 | 0 | 4284 | 27.9 | 0 | 5782 | 0.5 |
| | | 0.50 | 216 | 12957 | 0.8 | 3055 | 13865 | 1.0 | 0 | 14232 | 0.6 | 0 | 9831 | 320.1 | 0 | 13830 | 1.7 |
| | | 0.80 | 7705 | 20612 | 1.5 | 20259 | 20468 | 1.7 | 1114 | 23987 | 1.2 | 0 | 15050 | 849.1 | 105 | 16449 | 156.6 |
| | | 1.00 | 16323 | 25039 | 1.9 | 36308 | 25547 | 2.1 | 4966 | 28409 | 1.7 | 1661 | 21440 | 1015.6 | 500 | 21211 | 301.2 |
| | 3 | 0.10 | 0 | 3813 | 0.2 | 770 | 5177 | 0.2 | 0 | 4170 | 0.0 | 0 | 3234 | 6.8 | 0 | 4049 | 0.3 |
| | | 0.50 | 78 | 12973 | 0.7 | 5971 | 17540 | 1.1 | 15 | 14622 | 0.5 | 0 | 9527 | 283.6 | 0 | 13367 | 2.5 |
| | | 0.80 | 620 | 20989 | 1.5 | 28508 | 25024 | 2.0 | 126 | 23903 | 1.2 | 0 | 14518 | 867.3 | 0 | 20749 | 15.6 |
| | | 1.00 | 20795 | 25665 | 2.0 | 42092 | 27458 | 2.4 | 2106 | 29794 | 1.7 | 804 | 22216 | 1011.0 | 702 | 20716 | 320.7 |
| | 4 | 0.10 | 10 | 4879 | 0.2 | 2102 | 7188 | 0.4 | 10 | 5804 | 0.1 | 0 | 4497 | 26.1 | 0 | 5616 | 0.5 |
| | | 0.50 | 411 | 13991 | 1.0 | 8431 | 19291 | 1.5 | 50 | 15668 | 0.7 | 50 | 9719 | 313.1 | 50 | 10835 | 57.9 |
| | | 0.80 | 5337 | 18736 | 1.4 | 15900 | 23388 | 2.0 | 245 | 21161 | 1.2 | 301 | 13637 | 942.3 | 50 | 15212 | 165.4 |
| | | 1.00 | 21328 | 25619 | 2.1 | 36127 | 29322 | 2.6 | 3267 | 30343 | 1.8 | 5032 | 21518 | 1001.7 | 687 | 23039 | 426.8 |
| | 5 | 0.10 | 17 | 5767 | 0.3 | 255 | 8344 | 0.5 | 17 | 6978 | 0.1 | 0 | 5123 | 33.3 | 0 | 6841 | 0.6 |
| | | 0.50 | 66 | 15091 | 0.9 | 8281 | 18482 | 1.2 | 17 | 16693 | 0.7 | 0 | 10688 | 336.5 | 0 | 15907 | 3.1 |
| | | 0.80 | 6039 | 21893 | 1.4 | 14785 | 23403 | 1.8 | 417 | 24847 | 1.3 | 547 | 16205 | 984.3 | 0 | 22382 | 29.9 |
| | | 1.00 | 12657 | 26901 | 1.8 | 29003 | 27717 | 2.2 | 1451 | 29828 | 1.8 | 1172 | 23564 | 1005.2 | 716 | 22591 | 306.6 |
| | 6 | 0.10 | 0 | 5927 | 0.3 | 170 | 7551 | 0.4 | 0 | 6664 | 0.1 | 0 | 4994 | 36.2 | 0 | 6661 | 0.5 |
| | | 0.50 | 1247 | 12758 | 0.8 | 3821 | 17146 | 1.2 | 0 | 14312 | 0.6 | 0 | 9971 | 308.8 | 0 | 13228 | 1.9 |
| | | 0.80 | 4198 | 19513 | 1.6 | 26573 | 24380 | 2.2 | 20 | 23080 | 1.4 | 409 | 15486 | 750.3 | 209 | 16778 | 145.3 |
| | | 1.00 | 28518 | 25317 | 2.2 | 52866 | 29555 | 2.8 | 3164 | 29314 | 1.9 | 7573 | 22447 | 1000.5 | 965 | 22248 | 407.7 |
| | 7 | 0.10 | 0 | 3686 | 0.2 | 0 | 4993 | 0.3 | 0 | 4066 | 0.1 | 0 | 2963 | 12.5 | 0 | 3823 | 0.4 |
| | | 0.50 | 0 | 10625 | 0.7 | 8620 | 15125 | 1.2 | 0 | 12967 | 0.6 | 0 | 7477 | 247.4 | 0 | 11059 | 1.7 |
| | | 0.80 | 4890 | 19680 | 1.4 | 32328 | 23706 | 2.1 | 136 | 22255 | 1.2 | 951 | 15856 | 796.1 | 28 | 16496 | 154.5 |
| | | 1.00 | 25492 | 24462 | 2.0 | 40601 | 27330 | 2.5 | 2977 | 29990 | 1.8 | 5400 | 22799 | 1011.2 | 670 | 22894 | 428.2 |
| | 8 | 0.10 | 0 | 4595 | 0.3 | 860 | 7078 | 0.4 | 0 | 5596 | 0.1 | 0 | 3941 | 35.9 | 0 | 5039 | 0.5 |
| | | 0.50 | 0 | 11685 | 0.7 | 3519 | 15387 | 1.1 | 0 | 12875 | 0.6 | 0 | 7422 | 282.6 | 0 | 12322 | 1.7 |
| | | 0.80 | 8167 | 20085 | 1.7 | 25673 | 24750 | 2.3 | 363 | 22444 | 1.4 | 467 | 14977 | 1005.3 | 241 | 17441 | 189.5 |
| | | 1.00 | 20002 | 24877 | 2.2 | 31579 | 28807 | 2.6 | 2515 | 27968 | 2.0 | 2912 | 22309 | 1019.1 | 789 | 22448 | 393.6 |
| | 9 | 0.10 | 0 | 4041 | 0.3 | 95 | 6490 | 0.4 | 0 | 5029 | 0.1 | 0 | 3653 | 16.0 | 0 | 4539 | 0.5 |
| | | 0.50 | 0 | 12248 | 0.9 | 4475 | 16862 | 1.3 | 0 | 14389 | 0.7 | 0 | 8974 | 336.4 | 0 | 12258 | 1.9 |
| | | 0.80 | 5838 | 19365 | 1.4 | 14139 | 23388 | 1.9 | 145 | 21958 | 1.2 | 73 | 14346 | 913.4 | 140 | 16037 | 154.6 |
| | | 1.00 | 23001 | 25099 | 2.0 | 28037 | 26838 | 2.6 | 1439 | 29700 | 1.9 | 2231 | 21493 | 1000.7 | 450 | 21273 | 380.1 |
| | 10 | 0.10 | 0 | 3646 | 0.2 | 180 | 5128 | 0.3 | 0 | 4237 | 0.1 | 0 | 3328 | 13.0 | 0 | 4176 | 0.4 |
| | | 0.50 | 5 | 12952 | 0.8 | 6521 | 14687 | 1.1 | 5 | 14187 | 0.6 | 0 | 9219 | 255.5 | 0 | 12910 | 1.8 |
| | | 0.80 | 6926 | 21602 | 1.4 | 20906 | 21365 | 1.7 | 331 | 24471 | 1.3 | 993 | 15839 | 908.3 | 544 | 17259 | 192.1 |
| | | 1.00 | 13299 | 25286 | 1.8 | 30759 | 24897 | 2.2 | 3384 | 30429 | 1.8 | 4352 | 22692 | 1013.0 | 3806 | 21980 | 378.2 |
| | 11 | 0.10 | 0 | 4686 | 0.3 | 3574 | 6800 | 0.4 | 0 | 5715 | 0.1 | 0 | 4176 | 22.5 | 0 | 5394 | 0.5 |
| | | 0.50 | 542 | 12145 | 0.7 | 11967 | 16322 | 1.1 | 0 | 14044 | 0.6 | 0 | 8509 | 262.0 | 0 | 12242 | 2.9 |
| | | 0.80 | 2403 | 19234 | 1.3 | 13307 | 22450 | 1.8 | 169 | 22770 | 1.2 | 181 | 14173 | 707.2 | 181 | 16983 | 139.3 |
| | | 1.00 | 17012 | 25742 | 1.9 | 33457 | 28041 | 2.3 | 1993 | 29217 | 1.8 | 1754 | 21984 | 1000.3 | 1085 | 23480 | 282.1 |
| | 12 | 0.10 | 0 | 4532 | 0.2 | 432 | 6597 | 0.4 | 0 | 5091 | 0.1 | 0 | 3829 | 22.3 | 0 | 4970 | 0.5 |
| | | 0.50 | 1735 | 14863 | 1.0 | 11285 | 19083 | 1.8 | 190 | 18036 | 0.9 | 0 | 10308 | 409.1 | 0 | 14729 | 2.3 |
| | | 0.80 | 5453 | 21292 | 1.6 | 15778 | 25065 | 2.3 | 2103 | 25600 | 1.5 | 329 | 15930 | 804.4 | 8 | 17364 | 168.3 |
| | | 1.00 | 27342 | 27460 | 2.1 | 30033 | 28396 | 2.7 | 5144 | 31464 | 2.0 | 1377 | 22319 | 1017.0 | 756 | 22552 | 305.0 |
| | 13 | 0.10 | 0 | 5221 | 0.2 | 296 | 6592 | 0.4 | 0 | 5841 | 0.1 | 0 | 4616 | 26.8 | 0 | 5805 | 0.4 |
| | | 0.50 | 0 | 13085 | 0.9 | 7155 | 19246 | 1.4 | 0 | 14707 | 0.7 | 0 | 8252 | 375.0 | 0 | 12694 | 1.9 |
| | | 0.80 | 1964 | 17975 | 1.4 | 9505 | 23430 | 2.1 | 0 | 20666 | 1.3 | 0 | 12354 | 660.6 | 0 | 17397 | 8.0 |
| | | 1.00 | 41279 | 25331 | 2.1 | 38544 | 27805 | 2.6 | 2743 | 28391 | 2.0 | 3643 | 22878 | 1006.4 | 1155 | 23725 | 423.2 |
| | 14 | 0.10 | 0 | 4056 | 0.3 | 479 | 7611 | 0.5 | 0 | 4972 | 0.1 | 0 | 3734 | 23.0 | 0 | 4721 | 0.5 |
| | | 0.50 | 84 | 11605 | 0.8 | 5315 | 15639 | 1.2 | 10 | 13875 | 0.7 | 0 | 7936 | 267.2 | 0 | 11951 | 2.0 |
| | | 0.80 | 2927 | 17229 | 1.5 | 10246 | 21418 | 1.9 | 75 | 19858 | 1.2 | 0 | 11664 | 751.6 | 5 | 13994 | 113.7 |
| | | 1.00 | 33164 | 25902 | 2.3 | 40044 | 28222 | 2.7 | 4513 | 30282 | 2.0 | 1217 | 22897 | 1020.9 | 886 | 22179 | 333.2 |

Table A.15: Performance of CH-O, CH-R, CH-S, LS-O, and LS-S for instance size (Sz.) 1000, instance numbers (Nr.) 15–29.

| | | | CH-O | | | CH-R | | | CH-S | | | LS-O | | | LS-S | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 1000 | 15 | 0.10 | 0 | 5059 | 0.3 | 1675 | 6601 | 0.4 | 0 | 5843 | 0.1 | 0 | 4567 | 26.0 | 0 | 5690 | 0.5 |
| | | 0.50 | 0 | 13171 | 0.8 | 2381 | 16368 | 1.1 | 0 | 15489 | 0.6 | 0 | 10002 | 290.7 | 0 | 13783 | 1.8 |
| | | 0.80 | 755 | 20466 | 1.3 | 10115 | 22057 | 1.6 | 35 | 22961 | 1.2 | 415 | 15160 | 911.2 | 440 | 16561 | 157.8 |
| | | 1.00 | 11479 | 25693 | 2.1 | 28888 | 26727 | 2.4 | 3188 | 30088 | 1.9 | 2758 | 22403 | 1007.4 | 281 | 22455 | 353.8 |
| | 16 | 0.10 | 0 | 4785 | 0.3 | 0 | 5519 | 0.3 | 0 | 5813 | 0.1 | 0 | 3772 | 19.4 | 0 | 5400 | 0.4 |
| | | 0.50 | 0 | 12126 | 0.6 | 5998 | 14833 | 1.0 | 0 | 13759 | 0.5 | 0 | 9026 | 290.3 | 0 | 13007 | 1.5 |
| | | 0.80 | 1402 | 17668 | 1.2 | 15555 | 21353 | 1.6 | 155 | 21499 | 1.1 | 0 | 13390 | 618.1 | 0 | 18085 | 5.1 |
| | | 1.00 | 23767 | 25688 | 1.8 | 28961 | 26463 | 2.2 | 1577 | 29450 | 1.8 | 903 | 20220 | 1013.3 | 15 | 21646 | 314.4 |
| | 17 | 0.10 | 0 | 4004 | 0.3 | 4300 | 5514 | 0.3 | 0 | 4984 | 0.1 | 0 | 3389 | 22.4 | 0 | 4661 | 0.5 |
| | | 0.50 | 31 | 13453 | 0.8 | 4769 | 16688 | 1.1 | 1 | 16223 | 0.7 | 1 | 9936 | 405.5 | 1 | 11590 | 59.7 |
| | | 0.80 | 9575 | 22070 | 1.5 | 17569 | 24406 | 2.0 | 1889 | 26220 | 1.4 | 226 | 16870 | 850.7 | 156 | 17902 | 159.3 |
| | | 1.00 | 18155 | 24743 | 1.8 | 40331 | 27112 | 2.1 | 3879 | 30238 | 1.6 | 1012 | 21500 | 1014.9 | 420 | 22546 | 275.1 |
| | 18 | 0.10 | 0 | 4181 | 0.2 | 0 | 5835 | 0.3 | 0 | 5054 | 0.1 | 0 | 3779 | 23.8 | 0 | 4801 | 0.4 |
| | | 0.50 | 470 | 12661 | 0.9 | 13274 | 18247 | 1.3 | 35 | 15322 | 0.7 | 0 | 8971 | 378.0 | 0 | 13450 | 2.0 |
| | | 0.80 | 3164 | 20325 | 1.6 | 28276 | 24765 | 2.1 | 483 | 22891 | 1.4 | 0 | 13489 | 928.1 | 0 | 20619 | 6.0 |
| | | 1.00 | 40578 | 26496 | 2.3 | 71771 | 29686 | 3.0 | 2094 | 30215 | 2.2 | 389 | 21162 | 1001.7 | 490 | 21208 | 310.0 |
| | 19 | 0.10 | 0 | 4393 | 0.2 | 959 | 6433 | 0.4 | 0 | 5149 | 0.1 | 0 | 3739 | 26.8 | 0 | 4837 | 0.4 |
| | | 0.50 | 21 | 13148 | 0.9 | 4783 | 19214 | 1.4 | 21 | 14948 | 0.7 | 11 | 9297 | 404.5 | 21 | 10868 | 50.2 |
| | | 0.80 | 981 | 19866 | 1.5 | 15869 | 25079 | 2.1 | 21 | 22115 | 1.2 | 11 | 14222 | 883.4 | 36 | 15861 | 132.4 |
| | | 1.00 | 26342 | 27032 | 2.1 | 54438 | 29111 | 2.8 | 1846 | 29894 | 1.9 | 644 | 22904 | 1016.7 | 506 | 21920 | 317.2 |
| | 20 | 0.10 | 0 | 6019 | 0.3 | 782 | 8487 | 0.5 | 0 | 6842 | 0.2 | 0 | 5320 | 42.6 | 0 | 6492 | 0.7 |
| | | 0.50 | 201 | 12484 | 0.8 | 1994 | 14251 | 1.2 | 0 | 14119 | 0.6 | 0 | 9306 | 311.6 | 0 | 12735 | 1.8 |
| | | 0.80 | 809 | 20242 | 1.4 | 9509 | 22372 | 1.9 | 635 | 22541 | 1.2 | 625 | 14714 | 884.2 | 0 | 20371 | 10.5 |
| | | 1.00 | 16688 | 26072 | 2.0 | 38667 | 27244 | 2.4 | 5921 | 28323 | 1.7 | 4495 | 22454 | 1019.1 | 370 | 22364 | 373.7 |
| | 21 | 0.10 | 0 | 4786 | 0.3 | 373 | 7937 | 0.5 | 0 | 5932 | 0.2 | 0 | 4247 | 39.5 | 0 | 5313 | 0.7 |
| | | 0.50 | 0 | 12449 | 1.1 | 6377 | 16635 | 1.5 | 0 | 14753 | 0.9 | 0 | 8916 | 497.6 | 0 | 12943 | 2.5 |
| | | 0.80 | 1007 | 17922 | 1.5 | 17534 | 22065 | 1.9 | 138 | 20535 | 1.3 | 273 | 13164 | 1007.3 | 89 | 14567 | 166.2 |
| | | 1.00 | 21296 | 23839 | 2.0 | 43268 | 26489 | 2.7 | 2013 | 29574 | 1.9 | 1951 | 23347 | 1011.5 | 797 | 21333 | 452.4 |
| | 22 | 0.10 | 0 | 4254 | 0.3 | 1187 | 6232 | 0.4 | 0 | 5252 | 0.1 | 0 | 3867 | 26.2 | 0 | 4828 | 0.6 |
| | | 0.50 | 1661 | 13169 | 1.2 | 10071 | 18168 | 1.7 | 477 | 15233 | 0.9 | 0 | 8246 | 458.7 | 0 | 12930 | 2.4 |
| | | 0.80 | 3284 | 20370 | 1.8 | 29163 | 23258 | 2.6 | 494 | 22936 | 1.6 | 9 | 13864 | 1007.6 | 0 | 19849 | 20.1 |
| | | 1.00 | 20535 | 25283 | 2.5 | 68871 | 27012 | 3.1 | 3914 | 29400 | 2.3 | 4303 | 22148 | 1012.7 | 2298 | 21630 | 462.5 |
| | 23 | 0.10 | 5 | 4924 | 0.2 | 1250 | 6196 | 0.3 | 5 | 5429 | 0.1 | 0 | 4277 | 24.3 | 0 | 5379 | 0.5 |
| | | 0.50 | 36 | 13865 | 0.8 | 6538 | 16844 | 1.2 | 20 | 15186 | 0.6 | 0 | 9885 | 387.1 | 10 | 10944 | 56.8 |
| | | 0.80 | 3892 | 19010 | 1.4 | 17020 | 23117 | 1.8 | 50 | 22561 | 1.2 | 169 | 13448 | 779.1 | 30 | 15052 | 143.7 |
| | | 1.00 | 41686 | 27139 | 2.1 | 61185 | 29942 | 2.5 | 4382 | 31180 | 1.9 | 1978 | 22948 | 1005.2 | 784 | 22996 | 389.8 |
| | 24 | 0.10 | 0 | 3960 | 0.2 | 0 | 4600 | 0.3 | 0 | 4724 | 0.1 | 0 | 3505 | 13.4 | 0 | 4522 | 0.4 |
| | | 0.50 | 27 | 13225 | 0.7 | 2777 | 14779 | 1.1 | 2 | 15007 | 0.6 | 2 | 10199 | 341.4 | 2 | 11877 | 43.5 |
| | | 0.80 | 6348 | 21149 | 1.6 | 15229 | 22423 | 1.9 | 1239 | 23469 | 1.4 | 873 | 16355 | 990.4 | 1128 | 17690 | 197.4 |
| | | 1.00 | 14117 | 24242 | 2.0 | 26530 | 26305 | 2.5 | 4185 | 28953 | 2.0 | 2682 | 21888 | 1004.8 | 976 | 21728 | 358.4 |
| | 25 | 0.10 | 0 | 4824 | 0.3 | 793 | 7973 | 0.5 | 0 | 6523 | 0.1 | 0 | 4317 | 36.3 | 0 | 6159 | 0.6 |
| | | 0.50 | 0 | 12830 | 1.1 | 7232 | 19970 | 1.7 | 0 | 14713 | 0.8 | 0 | 8892 | 384.4 | 0 | 13129 | 2.2 |
| | | 0.80 | 6483 | 21993 | 1.8 | 20834 | 26704 | 2.4 | 599 | 24887 | 1.7 | 0 | 15233 | 1007.1 | 0 | 21138 | 20.7 |
| | | 1.00 | 26302 | 27529 | 2.5 | 55995 | 30587 | 3.1 | 2180 | 31566 | 2.4 | 1761 | 22097 | 1006.7 | 5 | 21479 | 417.0 |
| | 26 | 0.10 | 0 | 5847 | 0.3 | 356 | 7035 | 0.4 | 0 | 6429 | 0.1 | 0 | 4823 | 26.9 | 0 | 6229 | 0.5 |
| | | 0.50 | 20 | 13267 | 0.8 | 4944 | 15823 | 1.1 | 30 | 14646 | 0.6 | 0 | 9548 | 312.5 | 0 | 13624 | 1.7 |
| | | 0.80 | 2238 | 20437 | 1.4 | 12939 | 22721 | 1.8 | 524 | 23118 | 1.2 | 11 | 15514 | 950.1 | 0 | 21218 | 19.1 |
| | | 1.00 | 20349 | 25838 | 2.0 | 35679 | 26706 | 2.2 | 2452 | 28831 | 1.7 | 1534 | 21892 | 1007.1 | 313 | 22521 | 361.5 |
| | 27 | 0.10 | 0 | 3375 | 0.2 | 0 | 5355 | 0.4 | 25 | 4175 | 0.1 | 0 | 3144 | 13.9 | 0 | 4125 | 0.4 |
| | | 0.50 | 0 | 9773 | 0.6 | 4461 | 13440 | 1.1 | 25 | 11563 | 0.4 | 0 | 7396 | 190.8 | 0 | 10657 | 1.2 |
| | | 0.80 | 907 | 17975 | 1.6 | 10136 | 21828 | 2.3 | 25 | 19861 | 1.2 | 0 | 12740 | 758.8 | 0 | 17960 | 5.6 |
| | | 1.00 | 12904 | 25403 | 2.3 | 35436 | 28267 | 3.2 | 1510 | 28785 | 2.1 | 315 | 22238 | 1003.0 | 0 | 24187 | 97.5 |
| | 28 | 0.10 | 0 | 5476 | 0.3 | 1696 | 8154 | 0.5 | 0 | 6467 | 0.1 | 0 | 4821 | 25.9 | 0 | 6203 | 0.5 |
| | | 0.50 | 15 | 13716 | 0.7 | 3905 | 18345 | 1.3 | 15 | 14871 | 0.6 | 10 | 10416 | 262.9 | 10 | 11352 | 43.3 |
| | | 0.80 | 1613 | 20802 | 1.5 | 11516 | 26213 | 2.1 | 335 | 23182 | 1.4 | 10 | 15178 | 832.4 | 220 | 16584 | 140.0 |
| | | 1.00 | 23215 | 27435 | 2.3 | 49253 | 32750 | 3.0 | 2796 | 31563 | 2.1 | 2137 | 23282 | 1007.2 | 1569 | 24318 | 389.7 |
| | 29 | 0.10 | 0 | 5329 | 0.2 | 89 | 7895 | 0.4 | 0 | 5978 | 0.1 | 0 | 4442 | 30.0 | 0 | 5801 | 0.5 |
| | | 0.50 | 215 | 12357 | 0.9 | 6702 | 16262 | 1.4 | 0 | 13893 | 0.7 | 0 | 8549 | 305.7 | 0 | 13195 | 4.5 |
| | | 0.80 | 8784 | 19037 | 1.5 | 28753 | 23123 | 2.1 | 2694 | 22283 | 1.2 | 1388 | 14329 | 634.6 | 575 | 15769 | 138.0 |
| | | 1.00 | 43662 | 24147 | 2.0 | 51276 | 26103 | 2.5 | 10396 | 28020 | 1.7 | 6548 | 21979 | 1005.4 | 735 | 21436 | 356.8 |

Table A.16: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 20, instance numbers (Nr.) 0–14.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 20 | 0 | 0.10 | 0 | 469 | 0.1 | 0 | 595 | 0.0 | 0 | 469 | 200.0 | 0 | 469 | 200.0 | 0 | 469 | 200.0 |
| | | 0.50 | 0 | 829 | 0.3 | 0 | 1023 | 0.0 | 0 | 813 | 200.0 | 0 | 778 | 200.0 | 0 | 778 | 200.0 |
| | | 0.80 | 0 | 1102 | 0.4 | 0 | 1269 | 0.0 | 0 | 1073 | 200.0 | 0 | 1001 | 200.0 | 0 | 977 | 200.0 |
| | | 1.00 | 0 | 1164 | 0.9 | 0 | 1382 | 0.0 | 0 | 1166 | 200.0 | 0 | 1101 | 200.0 | 0 | 1084 | 200.0 |
| | 1 | 0.10 | 0 | 361 | 0.0 | 0 | 377 | 0.0 | 0 | 351 | 200.0 | 0 | 351 | 200.0 | 0 | 351 | 200.0 |
| | | 0.50 | 0 | 719 | 0.3 | 0 | 910 | 0.0 | 0 | 702 | 200.0 | 0 | 686 | 200.0 | 0 | 668 | 200.0 |
| | | 0.80 | 0 | 820 | 0.2 | 0 | 989 | 0.0 | 0 | 737 | 200.0 | 0 | 730 | 200.0 | 0 | 703 | 200.0 |
| | | 1.00 | 0 | 976 | 0.6 | 0 | 1139 | 0.0 | 0 | 871 | 200.0 | 0 | 813 | 200.0 | 0 | 871 | 200.0 |
| | 2 | 0.10 | 0 | 484 | 0.0 | 0 | 511 | 0.0 | 0 | 484 | 200.0 | 0 | 484 | 200.0 | 0 | 484 | 200.0 |
| | | 0.50 | 0 | 753 | 0.3 | 0 | 1026 | 0.0 | 0 | 753 | 200.0 | 0 | 722 | 200.0 | 0 | 717 | 200.0 |
| | | 0.80 | 0 | 954 | 0.6 | 0 | 1340 | 0.0 | 0 | 913 | 200.0 | 0 | 893 | 200.0 | 0 | 893 | 200.0 |
| | | 1.00 | 0 | 1139 | 1.1 | 0 | 1451 | 0.0 | 0 | 1086 | 200.0 | 0 | 1051 | 200.0 | 0 | 1061 | 200.0 |
| | 3 | 0.10 | 0 | 535 | 0.0 | 0 | 739 | 0.0 | 0 | 535 | 200.0 | 0 | 467 | 200.0 | 0 | 467 | 200.0 |
| | | 0.50 | 0 | 954 | 0.2 | 0 | 1147 | 0.0 | 0 | 953 | 200.0 | 0 | 869 | 200.0 | 0 | 869 | 200.0 |
| | | 0.80 | 0 | 1067 | 0.4 | 0 | 1309 | 0.0 | 0 | 1107 | 200.0 | 0 | 1060 | 200.0 | 0 | 1060 | 200.0 |
| | | 1.00 | 0 | 1318 | 0.6 | 0 | 1582 | 0.0 | 0 | 1291 | 200.0 | 0 | 1239 | 200.0 | 0 | 1239 | 200.0 |
| | 4 | 0.10 | 0 | 497 | 0.0 | 0 | 543 | 0.0 | 0 | 506 | 200.0 | 0 | 497 | 200.0 | 0 | 497 | 200.0 |
| | | 0.50 | 0 | 818 | 0.2 | 0 | 1048 | 0.0 | 0 | 840 | 200.0 | 0 | 772 | 200.0 | 0 | 772 | 200.0 |
| | | 0.80 | 0 | 989 | 0.4 | 0 | 1248 | 0.0 | 0 | 963 | 200.0 | 0 | 918 | 200.0 | 0 | 964 | 200.0 |
| | | 1.00 | 0 | 1217 | 0.5 | 0 | 1399 | 0.0 | 0 | 1090 | 200.0 | 0 | 1075 | 200.0 | 0 | 1075 | 200.0 |
| | 5 | 0.10 | 0 | 401 | 0.0 | 0 | 464 | 0.0 | 0 | 401 | 200.0 | 0 | 401 | 200.0 | 0 | 401 | 200.0 |
| | | 0.50 | 0 | 832 | 0.2 | 0 | 978 | 0.0 | 0 | 764 | 200.0 | 0 | 733 | 200.0 | 0 | 733 | 200.0 |
| | | 0.80 | 0 | 888 | 0.4 | 0 | 1166 | 0.0 | 0 | 888 | 200.0 | 0 | 854 | 200.0 | 0 | 854 | 200.0 |
| | | 1.00 | 0 | 1046 | 0.5 | 0 | 1177 | 0.0 | 0 | 1046 | 200.0 | 0 | 935 | 200.0 | 0 | 966 | 200.0 |
| | 6 | 0.10 | 0 | 382 | 0.0 | 0 | 493 | 0.0 | 0 | 374 | 200.0 | 0 | 374 | 200.0 | 0 | 374 | 200.0 |
| | | 0.50 | 0 | 832 | 0.2 | 0 | 1143 | 0.0 | 0 | 840 | 200.0 | 0 | 791 | 200.0 | 0 | 785 | 200.0 |
| | | 0.80 | 0 | 1154 | 0.5 | 0 | 1406 | 0.0 | 0 | 1075 | 200.0 | 0 | 1075 | 200.0 | 0 | 1041 | 200.0 |
| | | 1.00 | 0 | 1422 | 0.2 | 0 | 1446 | 0.0 | 0 | 1287 | 200.0 | 0 | 1231 | 200.0 | 0 | 1256 | 200.0 |
| | 7 | 0.10 | 0 | 543 | 0.1 | 0 | 819 | 0.0 | 0 | 521 | 200.0 | 0 | 521 | 200.0 | 0 | 521 | 200.0 |
| | | 0.50 | 0 | 982 | 0.3 | 0 | 1289 | 0.0 | 0 | 966 | 200.0 | 0 | 931 | 200.0 | 0 | 979 | 200.0 |
| | | 0.80 | 0 | 1237 | 0.4 | 0 | 1418 | 0.0 | 0 | 1079 | 200.0 | 0 | 1116 | 200.0 | 0 | 1076 | 200.0 |
| | | 1.00 | 0 | 1461 | 0.5 | 0 | 1593 | 0.0 | 0 | 1322 | 200.0 | 0 | 1263 | 200.0 | 0 | 1263 | 200.0 |
| | 8 | 0.10 | 0 | 351 | 0.0 | 0 | 656 | 0.0 | 0 | 347 | 200.0 | 0 | 347 | 200.0 | 0 | 347 | 200.0 |
| | | 0.50 | 0 | 826 | 0.3 | 0 | 1135 | 0.0 | 0 | 767 | 200.0 | 0 | 752 | 200.0 | 0 | 752 | 200.0 |
| | | 0.80 | 0 | 1070 | 0.4 | 0 | 1246 | 0.0 | 0 | 1039 | 200.0 | 0 | 1033 | 200.0 | 0 | 1022 | 200.0 |
| | | 1.00 | 0 | 1246 | 0.4 | 0 | 1442 | 0.0 | 0 | 1163 | 200.0 | 0 | 1109 | 200.0 | 0 | 1119 | 200.0 |
| | 9 | 0.10 | 0 | 649 | 0.1 | 0 | 754 | 0.0 | 0 | 643 | 200.0 | 0 | 643 | 200.0 | 0 | 643 | 200.0 |
| | | 0.50 | 0 | 984 | 0.4 | 0 | 1094 | 0.0 | 0 | 984 | 200.0 | 0 | 984 | 200.0 | 0 | 984 | 200.0 |
| | | 0.80 | 0 | 1071 | 0.6 | 0 | 1163 | 0.0 | 0 | 1071 | 200.0 | 0 | 1042 | 200.0 | 0 | 1042 | 200.0 |
| | | 1.00 | 0 | 1197 | 1.3 | 0 | 1466 | 0.0 | 0 | 1210 | 200.0 | 0 | 1109 | 200.0 | 0 | 1106 | 200.0 |
| | 10 | 0.10 | 0 | 598 | 0.0 | 0 | 660 | 0.0 | 0 | 598 | 200.0 | 0 | 559 | 200.0 | 0 | 559 | 200.0 |
| | | 0.50 | 0 | 944 | 0.2 | 0 | 1093 | 0.0 | 0 | 905 | 200.0 | 0 | 905 | 200.0 | 0 | 923 | 200.0 |
| | | 0.80 | 0 | 1252 | 0.2 | 0 | 1374 | 0.0 | 0 | 1143 | 200.0 | 0 | 1098 | 200.0 | 0 | 1098 | 200.0 |
| | | 1.00 | 530 | 1465 | 0.5 | 0 | 1538 | 0.3 | 0 | 1297 | 200.0 | 0 | 1220 | 200.0 | 0 | 1209 | 200.0 |
| | 11 | 0.10 | 0 | 217 | 0.0 | 0 | 308 | 0.0 | 0 | 217 | 200.0 | 0 | 217 | 200.0 | 0 | 217 | 200.0 |
| | | 0.50 | 0 | 703 | 0.1 | 0 | 933 | 0.0 | 0 | 579 | 200.0 | 0 | 579 | 200.0 | 0 | 579 | 200.0 |
| | | 0.80 | 0 | 873 | 0.5 | 0 | 1077 | 0.0 | 0 | 713 | 200.0 | 0 | 633 | 200.0 | 0 | 633 | 200.0 |
| | | 1.00 | 0 | 942 | 0.3 | 0 | 1077 | 0.0 | 0 | 919 | 200.0 | 0 | 919 | 200.0 | 0 | 854 | 200.0 |
| | 12 | 0.10 | 0 | 557 | 0.0 | 0 | 573 | 0.0 | 0 | 557 | 200.0 | 0 | 499 | 200.0 | 0 | 499 | 200.0 |
| | | 0.50 | 0 | 981 | 0.3 | 0 | 1153 | 0.0 | 0 | 887 | 200.0 | 0 | 848 | 200.0 | 0 | 801 | 200.0 |
| | | 0.80 | 0 | 1137 | 0.8 | 0 | 1488 | 0.0 | 0 | 1097 | 200.0 | 0 | 999 | 200.0 | 0 | 999 | 200.0 |
| | | 1.00 | 0 | 1227 | 1.1 | 0 | 1611 | 0.0 | 0 | 1184 | 200.0 | 0 | 1091 | 200.0 | 0 | 1136 | 200.0 |
| | 13 | 0.10 | 0 | 480 | 0.1 | 0 | 644 | 0.0 | 0 | 480 | 200.0 | 0 | 480 | 200.0 | 0 | 476 | 200.0 |
| | | 0.50 | 0 | 708 | 0.4 | 0 | 940 | 0.0 | 0 | 705 | 200.0 | 0 | 696 | 200.0 | 0 | 696 | 200.0 |
| | | 0.80 | 0 | 1109 | 0.5 | 0 | 1330 | 0.0 | 0 | 987 | 200.0 | 0 | 938 | 200.0 | 0 | 950 | 200.0 |
| | | 1.00 | 0 | 1224 | 0.5 | 0 | 1338 | 0.0 | 0 | 1122 | 200.0 | 0 | 1046 | 200.0 | 0 | 1020 | 200.0 |
| | 14 | 0.10 | 0 | 539 | 0.0 | 0 | 636 | 0.0 | 0 | 539 | 200.0 | 0 | 535 | 200.0 | 0 | 535 | 200.0 |
| | | 0.50 | 0 | 818 | 0.4 | 0 | 1195 | 0.0 | 0 | 828 | 200.0 | 0 | 818 | 200.0 | 0 | 818 | 200.0 |
| | | 0.80 | 0 | 1115 | 0.9 | 0 | 1310 | 0.0 | 0 | 1026 | 200.0 | 0 | 950 | 200.0 | 0 | 967 | 200.0 |
| | | 1.00 | 0 | 1267 | 0.6 | 0 | 1344 | 0.0 | 0 | 1220 | 200.0 | 0 | 1164 | 200.0 | 0 | 1164 | 200.0 |

Table A.17: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 20, instance numbers (Nr.) 15–29.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 20 | 15 | 0.10 | 0 | 424 | 0.0 | 0 | 524 | 0.0 | 0 | 318 | 200.0 | 0 | 318 | 200.0 | 0 | 318 | 200.0 |
| | | 0.50 | 0 | 678 | 0.2 | 0 | 951 | 0.0 | 0 | 678 | 200.0 | 0 | 570 | 200.0 | 0 | 597 | 200.0 |
| | | 0.80 | 0 | 930 | 0.5 | 0 | 1140 | 0.0 | 0 | 865 | 200.0 | 0 | 837 | 200.0 | 0 | 822 | 200.0 |
| | | 1.00 | 0 | 1077 | 0.5 | 0 | 1299 | 0.0 | 0 | 1012 | 200.0 | 0 | 967 | 200.0 | 0 | 967 | 200.0 |
| | 16 | 0.10 | 0 | 734 | 0.1 | 0 | 987 | 0.0 | 0 | 611 | 200.0 | 0 | 611 | 200.0 | 0 | 649 | 200.0 |
| | | 0.50 | 0 | 895 | 0.5 | 0 | 1373 | 0.0 | 0 | 895 | 200.0 | 0 | 895 | 200.0 | 0 | 893 | 200.0 |
| | | 0.80 | 0 | 1227 | 0.8 | 0 | 1564 | 0.0 | 0 | 1186 | 200.0 | 0 | 1137 | 200.0 | 0 | 1137 | 200.0 |
| | | 1.00 | 0 | 1285 | 0.8 | 0 | 1564 | 0.0 | 0 | 1263 | 200.0 | 0 | 1186 | 200.0 | 0 | 1263 | 200.0 |
| | 17 | 0.10 | 0 | 453 | 0.0 | 0 | 563 | 0.0 | 0 | 453 | 200.0 | 0 | 453 | 200.0 | 0 | 453 | 200.0 |
| | | 0.50 | 0 | 982 | 0.2 | 0 | 1061 | 0.0 | 0 | 935 | 200.0 | 0 | 935 | 200.0 | 0 | 950 | 200.0 |
| | | 0.80 | 0 | 1169 | 0.3 | 0 | 1356 | 0.0 | 0 | 1131 | 200.0 | 0 | 1093 | 200.0 | 0 | 1093 | 200.0 |
| | | 1.00 | 0 | 1382 | 0.5 | 0 | 1561 | 0.0 | 0 | 1280 | 200.0 | 0 | 1181 | 200.0 | 0 | 1215 | 200.0 |
| | 18 | 0.10 | 0 | 467 | 0.0 | 0 | 528 | 0.0 | 0 | 439 | 200.0 | 0 | 439 | 200.0 | 0 | 439 | 200.0 |
| | | 0.50 | 0 | 891 | 0.3 | 0 | 1112 | 0.0 | 0 | 858 | 200.0 | 0 | 786 | 200.0 | 0 | 783 | 200.0 |
| | | 0.80 | 180 | 1093 | 0.5 | 180 | 1271 | 0.2 | 0 | 964 | 200.0 | 0 | 906 | 200.0 | 0 | 956 | 200.0 |
| | | 1.00 | 325 | 1216 | 0.5 | 325 | 1328 | 0.2 | 0 | 1080 | 200.0 | 0 | 1030 | 200.0 | 0 | 1080 | 200.0 |
| | 19 | 0.10 | 0 | 603 | 0.0 | 0 | 662 | 0.0 | 0 | 579 | 200.0 | 0 | 579 | 200.0 | 0 | 584 | 200.0 |
| | | 0.50 | 0 | 981 | 0.4 | 0 | 1449 | 0.0 | 0 | 957 | 200.0 | 0 | 929 | 200.0 | 0 | 929 | 200.0 |
| | | 0.80 | 0 | 1226 | 0.7 | 0 | 1535 | 0.0 | 0 | 1225 | 200.0 | 0 | 1199 | 200.0 | 0 | 1199 | 200.0 |
| | | 1.00 | 0 | 1318 | 0.6 | 0 | 1535 | 0.0 | 0 | 1318 | 200.0 | 0 | 1318 | 200.0 | 0 | 1318 | 200.0 |
| | 20 | 0.10 | 0 | 371 | 0.0 | 0 | 434 | 0.0 | 0 | 371 | 200.0 | 0 | 371 | 200.0 | 0 | 381 | 200.0 |
| | | 0.50 | 0 | 640 | 0.3 | 0 | 1182 | 0.0 | 0 | 640 | 200.0 | 0 | 626 | 200.0 | 0 | 625 | 200.0 |
| | | 0.80 | 0 | 835 | 0.5 | 0 | 1322 | 0.0 | 0 | 773 | 200.0 | 0 | 742 | 200.0 | 0 | 773 | 200.0 |
| | | 1.00 | 0 | 1008 | 0.6 | 0 | 1366 | 0.0 | 0 | 1008 | 200.0 | 0 | 897 | 200.0 | 0 | 913 | 200.0 |
| | 21 | 0.10 | 0 | 475 | 0.0 | 0 | 484 | 0.0 | 0 | 475 | 200.0 | 0 | 421 | 200.0 | 0 | 421 | 200.0 |
| | | 0.50 | 0 | 758 | 0.1 | 0 | 898 | 0.0 | 0 | 711 | 200.0 | 0 | 675 | 200.0 | 0 | 675 | 200.0 |
| | | 0.80 | 0 | 952 | 0.3 | 0 | 1052 | 0.0 | 0 | 905 | 200.0 | 0 | 882 | 200.0 | 0 | 882 | 200.0 |
| | | 1.00 | 0 | 1148 | 0.4 | 0 | 1268 | 0.0 | 0 | 1024 | 200.0 | 0 | 1024 | 200.0 | 0 | 1024 | 200.0 |
| | 22 | 0.10 | 0 | 646 | 0.0 | 0 | 647 | 0.0 | 0 | 524 | 200.0 | 0 | 524 | 200.0 | 0 | 524 | 200.0 |
| | | 0.50 | 0 | 747 | 0.2 | 0 | 748 | 0.0 | 0 | 676 | 200.0 | 0 | 670 | 200.0 | 0 | 709 | 200.0 |
| | | 0.80 | 0 | 898 | 0.8 | 0 | 1120 | 0.0 | 0 | 849 | 200.0 | 0 | 802 | 200.0 | 0 | 841 | 200.0 |
| | | 1.00 | 0 | 905 | 1.1 | 0 | 1127 | 0.0 | 0 | 905 | 200.0 | 0 | 887 | 200.0 | 0 | 894 | 200.0 |
| | 23 | 0.10 | 0 | 535 | 0.0 | 0 | 591 | 0.0 | 0 | 535 | 200.0 | 0 | 499 | 200.0 | 0 | 535 | 200.0 |
| | | 0.50 | 0 | 779 | 0.3 | 0 | 1180 | 0.0 | 0 | 779 | 200.0 | 0 | 727 | 200.0 | 0 | 766 | 200.0 |
| | | 0.80 | 0 | 972 | 0.4 | 0 | 1490 | 0.0 | 0 | 923 | 200.0 | 0 | 908 | 200.0 | 0 | 908 | 200.0 |
| | | 1.00 | 0 | 1131 | 0.5 | 0 | 1515 | 0.0 | 0 | 1037 | 200.0 | 0 | 1062 | 200.0 | 0 | 1039 | 200.0 |
| | 24 | 0.10 | 0 | 683 | 0.1 | 0 | 716 | 0.0 | 0 | 643 | 200.0 | 0 | 620 | 200.0 | 0 | 620 | 200.0 |
| | | 0.50 | 0 | 910 | 0.2 | 0 | 1044 | 0.0 | 0 | 910 | 200.0 | 0 | 910 | 200.0 | 0 | 910 | 200.0 |
| | | 0.80 | 0 | 1153 | 0.3 | 0 | 1184 | 0.0 | 0 | 1044 | 200.0 | 0 | 965 | 200.0 | 0 | 1018 | 200.0 |
| | | 1.00 | 0 | 1247 | 0.6 | 0 | 1258 | 0.0 | 0 | 1153 | 200.0 | 0 | 1131 | 200.0 | 0 | 1131 | 200.0 |
| | 25 | 0.10 | 0 | 432 | 0.0 | 0 | 525 | 0.0 | 0 | 432 | 200.0 | 0 | 432 | 200.0 | 0 | 432 | 200.0 |
| | | 0.50 | 0 | 827 | 0.1 | 0 | 955 | 0.0 | 0 | 827 | 200.0 | 0 | 809 | 200.0 | 0 | 809 | 200.0 |
| | | 0.80 | 0 | 1043 | 0.2 | 0 | 1195 | 0.0 | 0 | 993 | 200.0 | 0 | 993 | 200.0 | 0 | 993 | 200.0 |
| | | 1.00 | 0 | 1043 | 0.5 | 0 | 1481 | 0.0 | 0 | 1043 | 200.0 | 0 | 1043 | 200.0 | 0 | 1043 | 200.0 |
| | 26 | 0.10 | 0 | 369 | 0.0 | 0 | 559 | 0.0 | 0 | 369 | 200.0 | 0 | 369 | 200.0 | 0 | 369 | 200.0 |
| | | 0.50 | 0 | 926 | 0.3 | 0 | 1024 | 0.0 | 0 | 819 | 200.0 | 0 | 866 | 200.0 | 0 | 866 | 200.0 |
| | | 0.80 | 0 | 1024 | 0.2 | 0 | 1047 | 0.0 | 0 | 939 | 200.0 | 0 | 939 | 200.0 | 0 | 939 | 200.0 |
| | | 1.00 | 0 | 1112 | 0.5 | 0 | 1240 | 0.0 | 0 | 1076 | 200.0 | 0 | 1065 | 200.0 | 0 | 1034 | 200.0 |
| | 27 | 0.10 | 0 | 607 | 0.0 | 0 | 687 | 0.0 | 0 | 607 | 200.0 | 0 | 607 | 200.0 | 0 | 607 | 200.0 |
| | | 0.50 | 0 | 872 | 0.4 | 0 | 1086 | 0.0 | 0 | 872 | 200.0 | 0 | 807 | 200.0 | 0 | 807 | 200.0 |
| | | 0.80 | 0 | 1068 | 0.8 | 0 | 1376 | 0.0 | 0 | 1068 | 200.0 | 0 | 1068 | 200.0 | 0 | 1068 | 200.0 |
| | | 1.00 | 0 | 1202 | 0.9 | 0 | 1538 | 0.0 | 0 | 1147 | 200.0 | 0 | 1147 | 200.0 | 0 | 1109 | 200.0 |
| | 28 | 0.10 | 0 | 552 | 0.0 | 0 | 584 | 0.0 | 0 | 531 | 200.0 | 0 | 531 | 200.0 | 0 | 531 | 200.0 |
| | | 0.50 | 0 | 953 | 0.5 | 0 | 1247 | 0.0 | 0 | 946 | 200.0 | 0 | 946 | 200.0 | 0 | 946 | 200.0 |
| | | 0.80 | 0 | 1201 | 0.6 | 0 | 1370 | 0.0 | 0 | 1179 | 200.0 | 0 | 1076 | 200.0 | 0 | 1141 | 200.0 |
| | | 1.00 | 30 | 1341 | 0.7 | 30 | 1436 | 0.6 | 0 | 1270 | 200.0 | 0 | 1196 | 200.0 | 0 | 1152 | 200.0 |
| | 29 | 0.10 | 0 | 387 | 0.0 | 0 | 460 | 0.0 | 0 | 383 | 200.0 | 0 | 369 | 200.0 | 0 | 369 | 200.0 |
| | | 0.50 | 0 | 593 | 0.2 | 0 | 1040 | 0.0 | 0 | 574 | 200.0 | 0 | 574 | 200.0 | 0 | 574 | 200.0 |
| | | 0.80 | 0 | 878 | 0.5 | 0 | 1285 | 0.0 | 0 | 876 | 200.0 | 0 | 876 | 200.0 | 0 | 855 | 200.0 |
| | | 1.00 | 0 | 1162 | 0.6 | 0 | 1377 | 0.0 | 0 | 1121 | 200.0 | 0 | 1064 | 200.0 | 0 | 1085 | 200.0 |

Table A.18: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 30, instance numbers (Nr.) 0–14.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 30 | 0 | 0.10 | 0 | 611 | 0.0 | 0 | 633 | 0.0 | 0 | 606 | 200.0 | 0 | 606 | 200.0 | 0 | 606 | 200.0 |
| | | 0.50 | 0 | 1020 | 0.5 | 0 | 1371 | 0.0 | 0 | 1036 | 200.0 | 0 | 1016 | 200.0 | 0 | 1016 | 200.0 |
| | | 0.80 | 0 | 1425 | 0.9 | 0 | 1598 | 0.0 | 0 | 1270 | 200.0 | 0 | 1254 | 200.0 | 0 | 1270 | 200.0 |
| | | 1.00 | 0 | 1545 | 1.2 | 0 | 1750 | 0.0 | 0 | 1482 | 200.0 | 0 | 1425 | 200.0 | 0 | 1389 | 200.0 |
| | 1 | 0.10 | 0 | 711 | 0.1 | 0 | 932 | 0.0 | 0 | 694 | 200.0 | 0 | 677 | 200.0 | 0 | 682 | 200.0 |
| | | 0.50 | 0 | 1211 | 0.9 | 0 | 1759 | 0.0 | 0 | 1126 | 200.0 | 0 | 1079 | 200.0 | 0 | 1082 | 200.0 |
| | | 0.80 | 0 | 1510 | 1.8 | 0 | 2183 | 0.0 | 0 | 1392 | 200.0 | 0 | 1359 | 200.1 | 0 | 1351 | 200.0 |
| | | 1.00 | 0 | 1820 | 2.6 | 0 | 2457 | 0.1 | 0 | 1641 | 200.0 | 0 | 1547 | 200.0 | 0 | 1547 | 200.0 |
| | 2 | 0.10 | 0 | 898 | 0.1 | 0 | 1366 | 0.0 | 0 | 826 | 200.0 | 0 | 826 | 200.0 | 0 | 826 | 200.0 |
| | | 0.50 | 0 | 1527 | 1.3 | 0 | 2027 | 0.0 | 0 | 1508 | 200.0 | 0 | 1378 | 200.0 | 0 | 1409 | 200.0 |
| | | 0.80 | 0 | 1734 | 1.3 | 0 | 2353 | 0.0 | 0 | 1629 | 200.0 | 0 | 1610 | 200.0 | 0 | 1539 | 200.0 |
| | | 1.00 | 0 | 2151 | 1.1 | 0 | 2432 | 0.0 | 0 | 1879 | 200.0 | 0 | 1835 | 200.1 | 0 | 1822 | 200.1 |
| | 3 | 0.10 | 0 | 679 | 0.1 | 0 | 1000 | 0.0 | 0 | 679 | 200.0 | 0 | 679 | 200.0 | 0 | 679 | 200.0 |
| | | 0.50 | 0 | 1083 | 0.7 | 0 | 1399 | 0.0 | 0 | 1095 | 200.0 | 0 | 1083 | 200.0 | 0 | 1083 | 200.0 |
| | | 0.80 | 0 | 1258 | 2.2 | 0 | 1879 | 0.0 | 0 | 1166 | 200.0 | 0 | 1118 | 200.0 | 0 | 1118 | 200.0 |
| | | 1.00 | 0 | 1700 | 2.0 | 0 | 2146 | 0.0 | 0 | 1641 | 200.0 | 0 | 1612 | 200.0 | 0 | 1593 | 200.1 |
| | 4 | 0.10 | 0 | 701 | 0.1 | 0 | 994 | 0.0 | 0 | 712 | 200.0 | 0 | 701 | 200.0 | 0 | 701 | 200.0 |
| | | 0.50 | 0 | 1265 | 0.9 | 0 | 1629 | 0.0 | 0 | 1251 | 200.0 | 0 | 1217 | 200.0 | 0 | 1251 | 200.0 |
| | | 0.80 | 0 | 1436 | 1.0 | 0 | 1715 | 0.0 | 0 | 1384 | 200.0 | 0 | 1365 | 200.0 | 0 | 1290 | 200.0 |
| | | 1.00 | 0 | 1628 | 1.2 | 0 | 1807 | 0.0 | 0 | 1628 | 200.0 | 0 | 1577 | 200.0 | 0 | 1539 | 200.0 |
| | 5 | 0.10 | 0 | 606 | 0.1 | 0 | 737 | 0.0 | 0 | 612 | 200.0 | 0 | 606 | 200.0 | 0 | 606 | 200.0 |
| | | 0.50 | 0 | 1164 | 0.8 | 0 | 1480 | 0.0 | 0 | 1076 | 200.0 | 0 | 1043 | 200.0 | 0 | 1043 | 200.0 |
| | | 0.80 | 0 | 1384 | 0.9 | 0 | 1700 | 0.0 | 0 | 1246 | 200.0 | 0 | 1263 | 200.0 | 0 | 1243 | 200.1 |
| | | 1.00 | 0 | 1459 | 1.4 | 0 | 1899 | 0.0 | 0 | 1387 | 200.0 | 0 | 1405 | 200.0 | 0 | 1405 | 200.0 |
| | 6 | 0.10 | 0 | 624 | 0.1 | 0 | 971 | 0.0 | 0 | 624 | 200.0 | 0 | 623 | 200.0 | 0 | 623 | 200.0 |
| | | 0.50 | 0 | 1199 | 0.6 | 0 | 1587 | 0.0 | 0 | 1127 | 200.0 | 0 | 1131 | 200.0 | 0 | 1136 | 200.0 |
| | | 0.80 | 0 | 1344 | 1.3 | 0 | 1786 | 0.0 | 0 | 1323 | 200.0 | 0 | 1326 | 200.0 | 0 | 1299 | 200.0 |
| | | 1.00 | 0 | 1456 | 1.9 | 0 | 1880 | 0.0 | 0 | 1473 | 200.0 | 0 | 1388 | 200.0 | 0 | 1384 | 200.0 |
| | 7 | 0.10 | 0 | 671 | 0.1 | 0 | 910 | 0.0 | 0 | 671 | 200.0 | 0 | 671 | 200.0 | 0 | 671 | 200.0 |
| | | 0.50 | 0 | 1226 | 1.2 | 0 | 1633 | 0.0 | 0 | 1222 | 200.0 | 0 | 1175 | 200.0 | 0 | 1175 | 200.0 |
| | | 0.80 | 0 | 1469 | 2.8 | 0 | 2000 | 0.0 | 0 | 1485 | 200.0 | 0 | 1408 | 200.1 | 0 | 1409 | 200.0 |
| | | 1.00 | 0 | 1774 | 3.3 | 0 | 2355 | 0.0 | 0 | 1543 | 200.0 | 0 | 1567 | 200.0 | 0 | 1529 | 200.0 |
| | 8 | 0.10 | 0 | 1192 | 0.2 | 0 | 1541 | 0.0 | 0 | 1111 | 200.0 | 0 | 1111 | 200.0 | 0 | 1111 | 200.0 |
| | | 0.50 | 0 | 1316 | 1.0 | 0 | 1707 | 0.0 | 0 | 1269 | 200.0 | 0 | 1269 | 200.0 | 0 | 1269 | 200.0 |
| | | 0.80 | 0 | 1569 | 1.6 | 0 | 1773 | 0.0 | 0 | 1521 | 200.0 | 0 | 1490 | 200.0 | 0 | 1504 | 200.0 |
| | | 1.00 | 0 | 1606 | 2.7 | 0 | 1773 | 0.0 | 0 | 1578 | 200.0 | 0 | 1578 | 200.0 | 0 | 1578 | 200.0 |
| | 9 | 0.10 | 0 | 571 | 0.1 | 0 | 822 | 0.0 | 0 | 571 | 200.0 | 0 | 571 | 200.0 | 0 | 571 | 200.0 |
| | | 0.50 | 0 | 1157 | 0.8 | 0 | 1924 | 0.0 | 0 | 1129 | 200.0 | 0 | 1124 | 200.0 | 0 | 1129 | 200.0 |
| | | 0.80 | 0 | 1468 | 1.7 | 0 | 2205 | 0.0 | 0 | 1441 | 200.0 | 0 | 1401 | 200.0 | 0 | 1401 | 200.0 |
| | | 1.00 | 0 | 1788 | 1.7 | 0 | 2336 | 0.0 | 0 | 1690 | 200.0 | 0 | 1626 | 200.0 | 0 | 1626 | 200.0 |
| | 10 | 0.10 | 0 | 593 | 0.0 | 0 | 610 | 0.0 | 0 | 593 | 200.0 | 0 | 593 | 200.0 | 0 | 593 | 200.0 |
| | | 0.50 | 0 | 1112 | 0.7 | 0 | 1375 | 0.0 | 0 | 1082 | 200.0 | 0 | 997 | 200.0 | 0 | 1039 | 200.0 |
| | | 0.80 | 0 | 1228 | 1.7 | 0 | 1662 | 0.0 | 0 | 1228 | 200.0 | 0 | 1153 | 200.0 | 0 | 1200 | 200.0 |
| | | 1.00 | 0 | 1349 | 2.5 | 0 | 1930 | 0.0 | 0 | 1345 | 200.0 | 0 | 1335 | 200.0 | 0 | 1307 | 200.0 |
| | 11 | 0.10 | 0 | 608 | 0.1 | 0 | 985 | 0.0 | 0 | 597 | 200.0 | 0 | 597 | 200.0 | 0 | 597 | 200.0 |
| | | 0.50 | 0 | 1155 | 0.7 | 0 | 1507 | 0.0 | 0 | 1144 | 200.0 | 0 | 1122 | 200.0 | 0 | 1109 | 200.0 |
| | | 0.80 | 0 | 1634 | 1.3 | 0 | 1990 | 0.0 | 0 | 1590 | 200.0 | 0 | 1447 | 200.0 | 0 | 1438 | 200.0 |
| | | 1.00 | 0 | 2071 | 1.5 | 0 | 2184 | 0.1 | 0 | 1868 | 200.0 | 0 | 1811 | 200.0 | 0 | 1722 | 200.0 |
| | 12 | 0.10 | 0 | 759 | 0.0 | 0 | 871 | 0.0 | 0 | 754 | 200.0 | 0 | 754 | 200.0 | 0 | 754 | 200.0 |
| | | 0.50 | 0 | 1071 | 0.5 | 0 | 1467 | 0.0 | 0 | 1071 | 200.0 | 0 | 1071 | 200.0 | 0 | 1071 | 200.0 |
| | | 0.80 | 0 | 1311 | 2.1 | 0 | 1923 | 0.0 | 0 | 1311 | 200.0 | 0 | 1311 | 200.1 | 0 | 1311 | 200.0 |
| | | 1.00 | 0 | 1719 | 1.8 | 0 | 2160 | 0.0 | 0 | 1694 | 200.0 | 0 | 1650 | 200.1 | 0 | 1634 | 200.0 |
| | 13 | 0.10 | 0 | 542 | 0.0 | 0 | 698 | 0.0 | 0 | 542 | 200.0 | 0 | 542 | 200.0 | 0 | 542 | 200.0 |
| | | 0.50 | 0 | 1081 | 0.7 | 0 | 1462 | 0.0 | 0 | 1081 | 200.0 | 0 | 1061 | 200.0 | 0 | 1061 | 200.0 |
| | | 0.80 | 0 | 1448 | 1.3 | 0 | 1717 | 0.0 | 0 | 1498 | 200.0 | 0 | 1407 | 200.0 | 0 | 1386 | 200.0 |
| | | 1.00 | 0 | 1505 | 1.4 | 0 | 2014 | 0.0 | 0 | 1505 | 200.0 | 0 | 1455 | 200.0 | 0 | 1482 | 200.0 |
| | 14 | 0.10 | 0 | 777 | 0.1 | 0 | 935 | 0.0 | 0 | 759 | 200.0 | 0 | 749 | 200.0 | 0 | 749 | 200.0 |
| | | 0.50 | 0 | 1346 | 0.9 | 0 | 1911 | 0.0 | 0 | 1318 | 200.0 | 0 | 1285 | 200.0 | 0 | 1285 | 200.0 |
| | | 0.80 | 0 | 1451 | 1.9 | 0 | 1974 | 0.0 | 0 | 1419 | 200.0 | 0 | 1376 | 200.0 | 0 | 1403 | 200.0 |
| | | 1.00 | 0 | 1652 | 2.0 | 0 | 2109 | 0.0 | 0 | 1595 | 200.0 | 0 | 1583 | 200.1 | 0 | 1583 | 200.1 |

Table A.19: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 30, instance numbers (Nr.) 15–29.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 30 | 15 | 0.10 | 0 | 993 | 0.1 | 0 | 1245 | 0.0 | 0 | 977 | 200.0 | 0 | 977 | 200.0 | 0 | 993 | 200.0 |
| | | 0.50 | 0 | 1298 | 0.9 | 0 | 1633 | 0.0 | 0 | 1298 | 200.0 | 0 | 1270 | 200.0 | 0 | 1270 | 200.0 |
| | | 0.80 | 0 | 1520 | 2.7 | 0 | 1911 | 0.0 | 0 | 1510 | 200.0 | 0 | 1512 | 200.1 | 0 | 1499 | 200.0 |
| | | 1.00 | 0 | 1726 | 3.0 | 0 | 2304 | 0.0 | 0 | 1729 | 200.0 | 0 | 1693 | 200.1 | 0 | 1689 | 200.0 |
| | 16 | 0.10 | 0 | 461 | 0.1 | 0 | 633 | 0.0 | 0 | 461 | 200.0 | 0 | 458 | 200.0 | 0 | 458 | 200.0 |
| | | 0.50 | 0 | 957 | 0.6 | 0 | 1382 | 0.0 | 0 | 969 | 200.0 | 0 | 916 | 200.0 | 0 | 916 | 200.0 |
| | | 0.80 | 0 | 1159 | 1.4 | 0 | 1778 | 0.0 | 0 | 1045 | 200.0 | 0 | 971 | 200.0 | 0 | 996 | 200.0 |
| | | 1.00 | 0 | 1547 | 1.0 | 0 | 1900 | 0.0 | 0 | 1371 | 200.0 | 0 | 1272 | 200.0 | 0 | 1249 | 200.0 |
| | 17 | 0.10 | 0 | 780 | 0.3 | 0 | 1027 | 0.0 | 0 | 795 | 200.0 | 0 | 776 | 200.0 | 0 | 776 | 200.0 |
| | | 0.50 | 0 | 1132 | 1.2 | 0 | 1427 | 0.0 | 0 | 1096 | 200.0 | 0 | 1066 | 200.0 | 0 | 1043 | 200.0 |
| | | 0.80 | 0 | 1463 | 1.5 | 0 | 1849 | 0.0 | 0 | 1388 | 200.0 | 0 | 1319 | 200.0 | 0 | 1313 | 200.1 |
| | | 1.00 | 0 | 1648 | 1.8 | 0 | 1999 | 0.0 | 0 | 1532 | 200.0 | 0 | 1542 | 200.0 | 0 | 1519 | 200.0 |
| | 18 | 0.10 | 0 | 561 | 0.0 | 0 | 579 | 0.0 | 0 | 561 | 200.0 | 0 | 561 | 200.0 | 0 | 561 | 200.0 |
| | | 0.50 | 0 | 1235 | 0.7 | 0 | 1458 | 0.0 | 0 | 1200 | 200.0 | 0 | 1192 | 200.0 | 0 | 1192 | 200.0 |
| | | 0.80 | 0 | 1534 | 0.9 | 0 | 1831 | 0.0 | 0 | 1532 | 200.0 | 0 | 1532 | 200.0 | 0 | 1532 | 200.0 |
| | | 1.00 | 0 | 1746 | 1.2 | 0 | 2181 | 0.0 | 0 | 1746 | 200.0 | 0 | 1714 | 200.0 | 0 | 1714 | 200.0 |
| | 19 | 0.10 | 0 | 516 | 0.1 | 0 | 1005 | 0.0 | 0 | 502 | 200.0 | 0 | 502 | 200.0 | 0 | 502 | 200.0 |
| | | 0.50 | 0 | 1270 | 1.3 | 0 | 1774 | 0.0 | 0 | 1263 | 200.0 | 0 | 1251 | 200.0 | 0 | 1205 | 200.0 |
| | | 0.80 | 0 | 1518 | 2.3 | 0 | 2065 | 0.0 | 0 | 1532 | 200.0 | 0 | 1420 | 200.0 | 0 | 1420 | 200.0 |
| | | 1.00 | 0 | 1760 | 2.5 | 0 | 2294 | 0.0 | 0 | 1695 | 200.0 | 0 | 1593 | 200.0 | 0 | 1624 | 200.0 |
| | 20 | 0.10 | 0 | 817 | 0.2 | 0 | 1222 | 0.0 | 0 | 896 | 200.0 | 0 | 803 | 200.0 | 0 | 803 | 200.0 |
| | | 0.50 | 0 | 1490 | 0.7 | 0 | 1824 | 0.0 | 0 | 1465 | 200.0 | 0 | 1450 | 200.0 | 0 | 1423 | 200.0 |
| | | 0.80 | 0 | 1686 | 2.3 | 0 | 2011 | 0.0 | 0 | 1686 | 200.1 | 0 | 1672 | 200.0 | 0 | 1646 | 200.0 |
| | | 1.00 | 0 | 1842 | 3.2 | 0 | 2170 | 0.0 | 0 | 1794 | 200.0 | 0 | 1792 | 200.1 | 0 | 1792 | 200.1 |
| | 21 | 0.10 | 0 | 640 | 0.1 | 0 | 897 | 0.0 | 0 | 640 | 200.0 | 0 | 581 | 200.0 | 0 | 581 | 200.0 |
| | | 0.50 | 0 | 993 | 0.4 | 0 | 1283 | 0.0 | 0 | 1003 | 200.0 | 0 | 928 | 200.0 | 0 | 928 | 200.0 |
| | | 0.80 | 0 | 1420 | 0.8 | 0 | 1636 | 0.0 | 0 | 1242 | 200.0 | 0 | 1239 | 200.0 | 0 | 1239 | 200.0 |
| | | 1.00 | 0 | 1647 | 1.1 | 0 | 1875 | 0.0 | 0 | 1422 | 200.0 | 0 | 1430 | 200.0 | 0 | 1399 | 200.0 |
| | 22 | 0.10 | 0 | 905 | 0.1 | 0 | 1003 | 0.0 | 0 | 929 | 200.0 | 0 | 881 | 200.0 | 0 | 881 | 200.0 |
| | | 0.50 | 0 | 1380 | 0.6 | 0 | 1554 | 0.0 | 0 | 1333 | 200.0 | 0 | 1312 | 200.0 | 0 | 1338 | 200.0 |
| | | 0.80 | 0 | 1418 | 1.6 | 0 | 1813 | 0.0 | 0 | 1430 | 200.0 | 0 | 1395 | 200.0 | 0 | 1395 | 200.0 |
| | | 1.00 | 0 | 1631 | 1.4 | 0 | 2249 | 0.0 | 0 | 1565 | 200.0 | 0 | 1531 | 200.0 | 0 | 1490 | 200.0 |
| | 23 | 0.10 | 0 | 738 | 0.1 | 0 | 876 | 0.0 | 0 | 738 | 200.0 | 0 | 738 | 200.0 | 0 | 738 | 200.0 |
| | | 0.50 | 0 | 1199 | 0.8 | 0 | 1724 | 0.0 | 0 | 1201 | 200.0 | 0 | 1187 | 200.0 | 0 | 1187 | 200.0 |
| | | 0.80 | 0 | 1413 | 2.2 | 0 | 2263 | 0.0 | 0 | 1440 | 200.0 | 0 | 1376 | 200.0 | 0 | 1346 | 200.0 |
| | | 1.00 | 0 | 1667 | 3.3 | 0 | 2282 | 0.0 | 0 | 1721 | 200.0 | 0 | 1598 | 200.0 | 0 | 1531 | 200.0 |
| | 24 | 0.10 | 0 | 547 | 0.1 | 0 | 630 | 0.0 | 0 | 525 | 200.0 | 0 | 525 | 200.0 | 0 | 525 | 200.0 |
| | | 0.50 | 0 | 1278 | 0.5 | 0 | 1547 | 0.0 | 0 | 1278 | 200.0 | 0 | 1205 | 200.0 | 0 | 1205 | 200.0 |
| | | 0.80 | 0 | 1627 | 1.9 | 0 | 2203 | 0.0 | 0 | 1575 | 200.0 | 0 | 1427 | 200.0 | 0 | 1352 | 200.0 |
| | | 1.00 | 0 | 2046 | 2.1 | 0 | 2375 | 0.0 | 0 | 1753 | 200.0 | 0 | 1701 | 200.0 | 0 | 1670 | 200.0 |
| | 25 | 0.10 | 0 | 972 | 0.3 | 0 | 1222 | 0.0 | 0 | 952 | 200.0 | 0 | 849 | 200.0 | 0 | 869 | 200.0 |
| | | 0.50 | 0 | 1514 | 1.5 | 0 | 1772 | 0.0 | 0 | 1389 | 200.0 | 0 | 1286 | 200.0 | 0 | 1355 | 200.0 |
| | | 0.80 | 0 | 1560 | 2.2 | 0 | 1970 | 0.0 | 0 | 1550 | 200.0 | 0 | 1532 | 200.1 | 0 | 1520 | 200.0 |
| | | 1.00 | 0 | 1765 | 2.2 | 0 | 2190 | 0.0 | 0 | 1679 | 200.0 | 0 | 1630 | 200.0 | 0 | 1652 | 200.1 |
| | 26 | 0.10 | 0 | 518 | 0.1 | 0 | 763 | 0.0 | 0 | 518 | 200.0 | 0 | 518 | 200.0 | 0 | 518 | 200.0 |
| | | 0.50 | 0 | 1009 | 0.6 | 0 | 1479 | 0.0 | 0 | 1009 | 200.0 | 0 | 1009 | 200.0 | 0 | 1009 | 200.0 |
| | | 0.80 | 0 | 1364 | 0.9 | 0 | 1941 | 0.0 | 0 | 1270 | 200.0 | 0 | 1258 | 200.0 | 0 | 1258 | 200.0 |
| | | 1.00 | 0 | 1517 | 1.6 | 0 | 2047 | 0.0 | 0 | 1517 | 200.0 | 0 | 1477 | 200.0 | 0 | 1505 | 200.0 |
| | 27 | 0.10 | 0 | 583 | 0.1 | 0 | 751 | 0.0 | 0 | 549 | 200.0 | 0 | 549 | 200.0 | 0 | 549 | 200.0 |
| | | 0.50 | 0 | 1044 | 0.3 | 0 | 1390 | 0.0 | 0 | 1029 | 200.0 | 0 | 1017 | 200.0 | 0 | 1013 | 200.0 |
| | | 0.80 | 0 | 1407 | 1.0 | 0 | 1740 | 0.0 | 0 | 1378 | 200.0 | 0 | 1279 | 200.0 | 0 | 1284 | 200.0 |
| | | 1.00 | 0 | 1540 | 1.6 | 0 | 2006 | 0.0 | 0 | 1434 | 200.0 | 0 | 1434 | 200.0 | 0 | 1434 | 200.0 |
| | 28 | 0.10 | 0 | 1002 | 0.1 | 0 | 1031 | 0.0 | 0 | 939 | 200.0 | 0 | 939 | 200.0 | 0 | 939 | 200.0 |
| | | 0.50 | 0 | 1232 | 0.5 | 0 | 1422 | 0.0 | 0 | 1232 | 200.0 | 0 | 1232 | 200.0 | 0 | 1232 | 200.0 |
| | | 0.80 | 0 | 1547 | 1.0 | 0 | 1769 | 0.0 | 0 | 1501 | 200.0 | 0 | 1413 | 200.0 | 0 | 1413 | 200.0 |
| | | 1.00 | 0 | 1608 | 1.5 | 0 | 2013 | 0.0 | 0 | 1568 | 200.0 | 0 | 1473 | 200.0 | 0 | 1473 | 200.0 |
| | 29 | 0.10 | 0 | 415 | 0.1 | 0 | 872 | 0.0 | 0 | 411 | 200.0 | 0 | 411 | 200.0 | 0 | 411 | 200.0 |
| | | 0.50 | 0 | 1065 | 0.6 | 0 | 1524 | 0.0 | 0 | 1024 | 200.0 | 0 | 963 | 200.0 | 0 | 963 | 200.0 |
| | | 0.80 | 0 | 1430 | 1.0 | 0 | 1858 | 0.0 | 0 | 1397 | 200.0 | 0 | 1328 | 200.0 | 0 | 1363 | 200.0 |
| | | 1.00 | 0 | 1686 | 0.8 | 0 | 1944 | 0.1 | 0 | 1494 | 200.0 | 0 | 1458 | 200.0 | 0 | 1503 | 200.0 |

Table A.20: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 50, instance numbers (Nr.) 0–14.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 50 | 0 | 0.10 | 0 | 1022 | 0.3 | 0 | 1311 | 0.0 | 0 | 1022 | 200.0 | 0 | 974 | 200.0 | 0 | 988 | 200.0 |
| | | 0.50 | 0 | 1840 | 3.7 | 0 | 2499 | 0.0 | 0 | 1832 | 200.0 | 0 | 1796 | 200.1 | 0 | 1763 | 200.0 |
| | | 0.80 | 0 | 2281 | 8.8 | 0 | 3030 | 0.0 | 0 | 2224 | 200.0 | 0 | 2242 | 200.0 | 0 | 2173 | 200.0 |
| | | 1.00 | 0 | 3116 | 6.4 | 0 | 3793 | 1.9 | 0 | 2828 | 200.0 | 0 | 2682 | 200.1 | 0 | 2501 | 200.1 |
| | 1 | 0.10 | 0 | 1340 | 0.2 | 0 | 1494 | 0.0 | 0 | 1340 | 200.0 | 0 | 1322 | 200.0 | 0 | 1322 | 200.0 |
| | | 0.50 | 0 | 1764 | 1.9 | 0 | 2302 | 0.0 | 0 | 1703 | 200.0 | 0 | 1673 | 200.0 | 0 | 1668 | 200.0 |
| | | 0.80 | 0 | 1956 | 5.1 | 0 | 3096 | 0.0 | 0 | 1960 | 200.0 | 0 | 1906 | 200.1 | 0 | 1937 | 200.0 |
| | | 1.00 | 0 | 2659 | 8.2 | 0 | 3678 | 0.0 | 0 | 2611 | 200.0 | 0 | 2454 | 200.1 | 0 | 2365 | 200.0 |
| | 2 | 0.10 | 0 | 749 | 0.1 | 0 | 1175 | 0.0 | 0 | 747 | 200.0 | 0 | 719 | 200.0 | 0 | 719 | 200.0 |
| | | 0.50 | 0 | 1968 | 2.5 | 0 | 2496 | 0.0 | 0 | 1949 | 200.0 | 0 | 1782 | 200.0 | 0 | 1844 | 200.0 |
| | | 0.80 | 0 | 2326 | 3.1 | 0 | 3045 | 0.0 | 0 | 2322 | 200.0 | 0 | 2222 | 200.1 | 0 | 2223 | 200.0 |
| | | 1.00 | 95 | 2636 | 5.6 | 0 | 3335 | 0.1 | 0 | 2597 | 200.0 | 0 | 2468 | 200.1 | 0 | 2534 | 200.1 |
| | 3 | 0.10 | 0 | 1118 | 0.4 | 0 | 1588 | 0.0 | 0 | 1140 | 200.0 | 0 | 1118 | 200.0 | 0 | 1131 | 200.0 |
| | | 0.50 | 0 | 1927 | 2.1 | 0 | 2740 | 0.0 | 0 | 1882 | 200.0 | 0 | 1841 | 200.0 | 0 | 1803 | 200.0 |
| | | 0.80 | 0 | 2355 | 5.3 | 0 | 3323 | 0.0 | 0 | 2245 | 200.0 | 0 | 2108 | 200.0 | 0 | 2082 | 200.1 |
| | | 1.00 | 0 | 2390 | 8.6 | 0 | 3573 | 0.1 | 0 | 2418 | 200.0 | 0 | 2268 | 200.1 | 0 | 2260 | 200.1 |
| | 4 | 0.10 | 0 | 1182 | 0.3 | 0 | 1360 | 0.0 | 0 | 1193 | 200.0 | 0 | 1182 | 200.0 | 0 | 1182 | 200.0 |
| | | 0.50 | 0 | 1881 | 1.5 | 0 | 2261 | 0.0 | 0 | 1878 | 200.0 | 0 | 1870 | 200.1 | 0 | 1870 | 200.0 |
| | | 0.80 | 0 | 2092 | 5.9 | 0 | 2646 | 0.0 | 0 | 2084 | 200.1 | 0 | 2018 | 200.0 | 0 | 2009 | 200.0 |
| | | 1.00 | 0 | 2598 | 9.0 | 0 | 3160 | 0.0 | 0 | 2494 | 200.1 | 0 | 2387 | 200.2 | 0 | 2357 | 200.1 |
| | 5 | 0.10 | 0 | 1295 | 0.3 | 0 | 1470 | 0.0 | 0 | 1240 | 200.0 | 0 | 1238 | 200.0 | 0 | 1238 | 200.0 |
| | | 0.50 | 0 | 1664 | 2.0 | 0 | 2272 | 0.0 | 0 | 1644 | 200.0 | 0 | 1631 | 200.0 | 0 | 1592 | 200.0 |
| | | 0.80 | 0 | 1920 | 7.1 | 0 | 2824 | 0.0 | 0 | 1924 | 200.1 | 0 | 1905 | 200.0 | 0 | 1905 | 200.1 |
| | | 1.00 | 0 | 2245 | 7.1 | 0 | 2935 | 0.0 | 0 | 2243 | 200.1 | 0 | 2145 | 200.1 | 0 | 2096 | 200.1 |
| | 6 | 0.10 | 0 | 825 | 0.2 | 0 | 947 | 0.0 | 0 | 801 | 200.0 | 0 | 787 | 200.0 | 0 | 787 | 200.0 |
| | | 0.50 | 0 | 1749 | 1.8 | 0 | 2096 | 0.0 | 0 | 1698 | 200.1 | 0 | 1698 | 200.0 | 0 | 1698 | 200.0 |
| | | 0.80 | 0 | 2052 | 4.5 | 0 | 2727 | 0.0 | 0 | 2023 | 200.0 | 0 | 1968 | 200.0 | 0 | 1968 | 200.1 |
| | | 1.00 | 0 | 2450 | 4.2 | 0 | 3087 | 0.0 | 0 | 2389 | 200.1 | 0 | 2320 | 200.1 | 0 | 2303 | 200.0 |
| | 7 | 0.10 | 0 | 1116 | 0.1 | 0 | 1227 | 0.0 | 0 | 1116 | 200.0 | 0 | 1116 | 200.0 | 0 | 1116 | 200.0 |
| | | 0.50 | 0 | 1994 | 2.1 | 0 | 2324 | 0.0 | 0 | 1980 | 200.0 | 0 | 1964 | 200.0 | 0 | 1964 | 200.0 |
| | | 0.80 | 0 | 2300 | 3.0 | 0 | 2703 | 0.0 | 0 | 2243 | 200.2 | 0 | 2207 | 200.1 | 0 | 2191 | 200.0 |
| | | 1.00 | 0 | 2524 | 7.0 | 0 | 3120 | 0.0 | 0 | 2481 | 200.0 | 0 | 2370 | 200.0 | 0 | 2414 | 200.1 |
| | 8 | 0.10 | 0 | 875 | 0.3 | 0 | 1119 | 0.0 | 0 | 841 | 200.0 | 0 | 798 | 200.0 | 0 | 807 | 200.0 |
| | | 0.50 | 0 | 1535 | 3.1 | 0 | 2214 | 0.0 | 0 | 1507 | 200.0 | 0 | 1432 | 200.1 | 0 | 1432 | 200.0 |
| | | 0.80 | 0 | 2002 | 6.0 | 0 | 2899 | 0.0 | 0 | 1993 | 200.0 | 0 | 1920 | 200.0 | 0 | 1865 | 200.0 |
| | | 1.00 | 0 | 2712 | 6.6 | 0 | 3786 | 2.8 | 0 | 2647 | 200.1 | 0 | 2553 | 200.0 | 0 | 2493 | 200.0 |
| | 9 | 0.10 | 0 | 1392 | 0.4 | 0 | 1683 | 0.0 | 0 | 1392 | 200.0 | 0 | 1371 | 200.0 | 0 | 1392 | 200.0 |
| | | 0.50 | 0 | 1793 | 2.5 | 0 | 2268 | 0.0 | 0 | 1815 | 200.0 | 0 | 1733 | 200.0 | 0 | 1733 | 200.0 |
| | | 0.80 | 0 | 2044 | 4.5 | 0 | 2684 | 0.0 | 0 | 2002 | 200.1 | 0 | 1959 | 200.1 | 0 | 1933 | 200.1 |
| | | 1.00 | 0 | 2437 | 4.9 | 0 | 3260 | 0.0 | 0 | 2422 | 200.0 | 0 | 2289 | 200.1 | 0 | 2293 | 200.0 |
| | 10 | 0.10 | 0 | 973 | 0.2 | 0 | 1094 | 0.0 | 0 | 973 | 200.0 | 0 | 973 | 200.0 | 0 | 973 | 200.0 |
| | | 0.50 | 0 | 1791 | 2.4 | 0 | 2798 | 0.0 | 0 | 1875 | 200.0 | 0 | 1780 | 200.1 | 0 | 1780 | 200.1 |
| | | 0.80 | 0 | 2132 | 3.8 | 0 | 3026 | 0.0 | 0 | 2145 | 200.0 | 0 | 2103 | 200.1 | 0 | 2040 | 200.0 |
| | | 1.00 | 0 | 2635 | 7.5 | 0 | 3419 | 0.0 | 0 | 2501 | 200.0 | 0 | 2433 | 200.0 | 0 | 2450 | 200.1 |
| | 11 | 0.10 | 0 | 916 | 0.4 | 0 | 1402 | 0.0 | 0 | 974 | 200.0 | 0 | 886 | 200.0 | 0 | 886 | 200.0 |
| | | 0.50 | 0 | 1701 | 2.0 | 0 | 2507 | 0.0 | 0 | 1583 | 200.0 | 0 | 1534 | 200.0 | 0 | 1575 | 200.0 |
| | | 0.80 | 0 | 2380 | 2.8 | 0 | 3020 | 0.0 | 0 | 2133 | 200.0 | 0 | 2097 | 200.1 | 0 | 2089 | 200.1 |
| | | 1.00 | 158 | 2992 | 5.4 | 158 | 3366 | 6.7 | 0 | 2888 | 200.1 | 0 | 2688 | 200.0 | 0 | 2555 | 200.1 |
| | 12 | 0.10 | 0 | 880 | 0.4 | 0 | 1098 | 0.0 | 0 | 881 | 200.0 | 0 | 872 | 200.0 | 0 | 872 | 200.0 |
| | | 0.50 | 0 | 1770 | 2.9 | 0 | 2396 | 0.0 | 0 | 1751 | 200.0 | 0 | 1768 | 200.0 | 0 | 1729 | 200.0 |
| | | 0.80 | 0 | 2305 | 2.9 | 0 | 2670 | 0.0 | 0 | 2183 | 200.0 | 0 | 2099 | 200.1 | 0 | 2130 | 200.1 |
| | | 1.00 | 0 | 2630 | 2.3 | 0 | 2988 | 0.5 | 0 | 2381 | 200.1 | 0 | 2364 | 200.0 | 0 | 2347 | 200.0 |
| | 13 | 0.10 | 0 | 1086 | 0.3 | 0 | 1438 | 0.0 | 0 | 1086 | 200.0 | 0 | 1081 | 200.0 | 0 | 1081 | 200.0 |
| | | 0.50 | 0 | 1833 | 2.6 | 0 | 2286 | 0.0 | 0 | 1791 | 200.0 | 0 | 1748 | 200.1 | 0 | 1706 | 200.1 |
| | | 0.80 | 0 | 2329 | 4.5 | 0 | 3094 | 0.0 | 0 | 2217 | 200.1 | 0 | 2108 | 200.0 | 0 | 2092 | 200.0 |
| | | 1.00 | 0 | 2897 | 7.2 | 0 | 3432 | 0.5 | 0 | 2794 | 200.0 | 0 | 2518 | 200.1 | 0 | 2514 | 200.0 |
| | 14 | 0.10 | 0 | 1061 | 0.3 | 0 | 1286 | 0.0 | 0 | 1074 | 200.0 | 0 | 1057 | 200.0 | 0 | 1057 | 200.0 |
| | | 0.50 | 0 | 1760 | 2.3 | 0 | 2202 | 0.0 | 0 | 1829 | 200.0 | 0 | 1736 | 200.0 | 0 | 1736 | 200.0 |
| | | 0.80 | 0 | 2037 | 5.5 | 0 | 2670 | 0.0 | 0 | 2096 | 200.0 | 0 | 1948 | 200.1 | 0 | 1988 | 200.1 |
| | | 1.00 | 0 | 2179 | 5.8 | 0 | 2773 | 0.0 | 0 | 2207 | 200.0 | 0 | 2157 | 200.0 | 0 | 2015 | 200.1 |

Table A.21: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 50, instance numbers (Nr.) 15–29.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | t[s] | $C_a$ | $C_u$ | t[s] | $C_a$ | $C_u$ | t[s] | $C_a$ | $C_u$ | t[s] | $C_a$ | $C_u$ | t[s] |
| 50 | 15 | 0.10 | 0 | 1132 | 0.3 | 0 | 1446 | 0.0 | 0 | 1132 | 200.0 | 0 | 1132 | 200.0 | 0 | 1132 | 200.0 |
| | | 0.50 | 0 | 2025 | 2.3 | 0 | 2648 | 0.0 | 0 | 1949 | 200.0 | 0 | 1880 | 200.1 | 0 | 1907 | 200.0 |
| | | 0.80 | 0 | 2205 | 9.8 | 0 | 3079 | 0.0 | 0 | 2191 | 200.1 | 0 | 2157 | 200.0 | 0 | 2128 | 200.0 |
| | | 1.00 | 0 | 2898 | 5.7 | 0 | 3713 | 0.6 | 0 | 2883 | 200.0 | 0 | 2672 | 200.1 | 0 | 2582 | 200.1 |
| | 16 | 0.10 | 0 | 1162 | 0.2 | 0 | 1315 | 0.0 | 0 | 1113 | 200.0 | 0 | 953 | 200.0 | 0 | 953 | 200.0 |
| | | 0.50 | 0 | 1635 | 4.1 | 0 | 2250 | 0.0 | 0 | 1472 | 200.0 | 0 | 1493 | 200.0 | 0 | 1456 | 200.0 |
| | | 0.80 | 0 | 2072 | 5.5 | 0 | 2719 | 0.0 | 0 | 1996 | 200.1 | 0 | 1955 | 200.0 | 0 | 1911 | 200.0 |
| | | 1.00 | 0 | 2319 | 8.7 | 0 | 3149 | 0.1 | 0 | 2310 | 200.2 | 0 | 2251 | 200.1 | 0 | 2244 | 200.1 |
| | 17 | 0.10 | 0 | 1103 | 0.2 | 0 | 1376 | 0.0 | 0 | 1110 | 200.0 | 0 | 1103 | 200.0 | 0 | 1110 | 200.0 |
| | | 0.50 | 0 | 2071 | 3.0 | 0 | 2603 | 0.0 | 0 | 2064 | 200.0 | 0 | 2039 | 200.0 | 0 | 2008 | 200.0 |
| | | 0.80 | 0 | 2290 | 4.9 | 0 | 3164 | 0.0 | 0 | 2294 | 200.1 | 0 | 2250 | 200.0 | 0 | 2216 | 200.0 |
| | | 1.00 | 0 | 2732 | 6.8 | 0 | 3886 | 0.0 | 0 | 2697 | 200.0 | 0 | 2726 | 200.0 | 0 | 2553 | 200.0 |
| | 18 | 0.10 | 0 | 839 | 0.1 | 0 | 1093 | 0.0 | 0 | 864 | 200.0 | 0 | 839 | 200.0 | 0 | 839 | 200.0 |
| | | 0.50 | 0 | 1945 | 1.4 | 0 | 2195 | 0.0 | 0 | 1796 | 200.0 | 0 | 1785 | 200.0 | 0 | 1740 | 200.1 |
| | | 0.80 | 0 | 2395 | 5.9 | 0 | 3104 | 0.0 | 0 | 2347 | 200.1 | 0 | 2281 | 200.0 | 0 | 2248 | 200.1 |
| | | 1.00 | 0 | 2959 | 7.7 | 0 | 3542 | 1.8 | 0 | 2789 | 200.0 | 0 | 2608 | 200.0 | 0 | 2547 | 200.1 |
| | 19 | 0.10 | 0 | 961 | 0.4 | 0 | 1361 | 0.0 | 0 | 973 | 200.0 | 0 | 949 | 200.0 | 0 | 949 | 200.0 |
| | | 0.50 | 0 | 1499 | 2.8 | 0 | 2322 | 0.0 | 0 | 1498 | 200.0 | 0 | 1494 | 200.0 | 0 | 1481 | 200.0 |
| | | 0.80 | 0 | 1963 | 5.2 | 0 | 2743 | 0.0 | 0 | 1865 | 200.0 | 0 | 1777 | 200.0 | 0 | 1736 | 200.1 |
| | | 1.00 | 0 | 2613 | 4.7 | 0 | 3012 | 1.9 | 0 | 2392 | 200.1 | 0 | 2351 | 200.0 | 0 | 2243 | 200.1 |
| | 20 | 0.10 | 0 | 1626 | 0.6 | 0 | 2150 | 0.0 | 0 | 1637 | 200.0 | 0 | 1626 | 200.0 | 0 | 1626 | 200.0 |
| | | 0.50 | 0 | 1998 | 2.6 | 0 | 2566 | 0.0 | 0 | 1968 | 200.0 | 0 | 1966 | 200.0 | 0 | 1966 | 200.1 |
| | | 0.80 | 0 | 2473 | 4.1 | 0 | 2901 | 0.0 | 0 | 2420 | 200.1 | 0 | 2360 | 200.1 | 0 | 2368 | 200.0 |
| | | 1.00 | 0 | 2891 | 7.2 | 0 | 3306 | 0.3 | 0 | 2807 | 200.1 | 0 | 2791 | 200.0 | 0 | 2718 | 200.1 |
| | 21 | 0.10 | 0 | 1210 | 0.2 | 0 | 1482 | 0.0 | 0 | 1210 | 200.0 | 0 | 1213 | 200.0 | 0 | 1213 | 200.0 |
| | | 0.50 | 0 | 1763 | 1.9 | 0 | 2237 | 0.0 | 0 | 1781 | 200.0 | 0 | 1735 | 200.0 | 0 | 1727 | 200.0 |
| | | 0.80 | 0 | 2352 | 6.2 | 0 | 3016 | 0.0 | 0 | 2257 | 200.0 | 0 | 2194 | 200.0 | 0 | 2228 | 200.0 |
| | | 1.00 | 0 | 2681 | 9.4 | 0 | 3399 | 0.0 | 0 | 2687 | 200.0 | 0 | 2599 | 200.1 | 0 | 2617 | 200.1 |
| | 22 | 0.10 | 0 | 1355 | 0.4 | 0 | 1808 | 0.0 | 0 | 1305 | 200.0 | 0 | 1305 | 200.0 | 0 | 1338 | 200.0 |
| | | 0.50 | 0 | 2113 | 2.9 | 0 | 2869 | 0.0 | 0 | 2047 | 200.0 | 0 | 2072 | 200.0 | 0 | 2073 | 200.0 |
| | | 0.80 | 0 | 2322 | 7.7 | 0 | 3219 | 0.0 | 0 | 2362 | 200.1 | 0 | 2275 | 200.1 | 0 | 2234 | 200.0 |
| | | 1.00 | 0 | 2833 | 7.2 | 0 | 3488 | 0.0 | 0 | 2618 | 200.0 | 0 | 2593 | 200.1 | 0 | 2583 | 200.0 |
| | 23 | 0.10 | 0 | 1035 | 0.1 | 0 | 1279 | 0.0 | 0 | 997 | 200.0 | 0 | 986 | 200.0 | 0 | 986 | 200.0 |
| | | 0.50 | 0 | 2117 | 3.6 | 0 | 2719 | 0.0 | 0 | 2144 | 200.0 | 0 | 2068 | 200.0 | 0 | 2065 | 200.1 |
| | | 0.80 | 0 | 2521 | 5.1 | 0 | 3142 | 0.0 | 0 | 2364 | 200.1 | 0 | 2347 | 200.1 | 0 | 2308 | 200.0 |
| | | 1.00 | 0 | 2616 | 4.4 | 0 | 3267 | 0.0 | 0 | 2551 | 200.0 | 0 | 2601 | 200.0 | 0 | 2516 | 200.0 |
| | 24 | 0.10 | 0 | 803 | 0.3 | 0 | 1137 | 0.0 | 0 | 784 | 200.0 | 0 | 784 | 200.0 | 0 | 784 | 200.0 |
| | | 0.50 | 0 | 1570 | 3.5 | 0 | 2570 | 0.0 | 0 | 1553 | 200.0 | 0 | 1444 | 200.0 | 0 | 1444 | 200.0 |
| | | 0.80 | 0 | 2159 | 6.2 | 0 | 3314 | 0.0 | 0 | 2181 | 200.0 | 0 | 2181 | 200.1 | 0 | 2144 | 200.1 |
| | | 1.00 | 0 | 2917 | 7.3 | 0 | 3533 | 0.0 | 0 | 2830 | 200.0 | 0 | 2699 | 200.0 | 0 | 2646 | 200.1 |
| | 25 | 0.10 | 0 | 856 | 0.6 | 0 | 1343 | 0.0 | 0 | 856 | 200.0 | 0 | 823 | 200.0 | 0 | 823 | 200.0 |
| | | 0.50 | 0 | 1720 | 3.6 | 0 | 2413 | 0.0 | 0 | 1556 | 200.0 | 0 | 1537 | 200.1 | 0 | 1537 | 200.1 |
| | | 0.80 | 0 | 2126 | 3.0 | 0 | 2620 | 0.0 | 0 | 2014 | 200.1 | 0 | 1931 | 200.0 | 0 | 1929 | 200.1 |
| | | 1.00 | 0 | 2217 | 3.7 | 0 | 2908 | 0.0 | 0 | 2206 | 200.1 | 0 | 2087 | 200.1 | 0 | 2073 | 200.1 |
| | 26 | 0.10 | 0 | 929 | 0.1 | 0 | 1193 | 0.0 | 0 | 937 | 200.0 | 0 | 920 | 200.0 | 0 | 920 | 200.0 |
| | | 0.50 | 0 | 1680 | 2.7 | 0 | 2406 | 0.0 | 0 | 1709 | 200.0 | 0 | 1646 | 200.0 | 0 | 1625 | 200.0 |
| | | 0.80 | 0 | 2216 | 6.2 | 0 | 2926 | 0.0 | 0 | 2189 | 200.0 | 0 | 2186 | 200.0 | 0 | 2142 | 200.1 |
| | | 1.00 | 0 | 2493 | 7.5 | 0 | 3055 | 0.0 | 0 | 2569 | 200.0 | 0 | 2376 | 200.0 | 0 | 2311 | 200.0 |
| | 27 | 0.10 | 0 | 1340 | 0.3 | 0 | 1545 | 0.0 | 0 | 1397 | 200.0 | 0 | 1340 | 200.0 | 0 | 1340 | 200.0 |
| | | 0.50 | 0 | 2008 | 2.5 | 0 | 2469 | 0.0 | 0 | 1999 | 200.0 | 0 | 1998 | 200.1 | 0 | 1998 | 200.1 |
| | | 0.80 | 0 | 2123 | 5.6 | 0 | 2842 | 0.0 | 0 | 2145 | 200.0 | 0 | 2140 | 200.0 | 0 | 2130 | 200.0 |
| | | 1.00 | 0 | 2357 | 7.0 | 0 | 3079 | 0.0 | 0 | 2376 | 200.1 | 0 | 2330 | 200.0 | 0 | 2328 | 200.0 |
| | 28 | 0.10 | 0 | 1154 | 0.6 | 0 | 2066 | 0.0 | 0 | 1099 | 200.0 | 0 | 1082 | 200.0 | 0 | 1082 | 200.0 |
| | | 0.50 | 0 | 2198 | 4.5 | 0 | 3064 | 0.0 | 0 | 2170 | 200.0 | 0 | 2142 | 200.0 | 0 | 2028 | 200.0 |
| | | 0.80 | 0 | 2865 | 6.1 | 0 | 3978 | 0.0 | 0 | 2787 | 200.1 | 0 | 2659 | 200.1 | 0 | 2648 | 200.0 |
| | | 1.00 | 0 | 3271 | 8.2 | 0 | 4279 | 0.5 | 0 | 3311 | 200.0 | 0 | 3065 | 200.2 | 0 | 2913 | 200.0 |
| | 29 | 0.10 | 0 | 620 | 0.2 | 0 | 1139 | 0.0 | 0 | 620 | 200.0 | 0 | 620 | 200.0 | 0 | 620 | 200.0 |
| | | 0.50 | 0 | 1844 | 3.1 | 0 | 2721 | 0.0 | 0 | 1808 | 200.0 | 0 | 1754 | 200.0 | 0 | 1685 | 200.0 |
| | | 0.80 | 0 | 2365 | 3.8 | 0 | 3171 | 0.0 | 0 | 2339 | 200.0 | 0 | 2295 | 200.0 | 0 | 2307 | 200.1 |
| | | 1.00 | 0 | 2630 | 8.6 | 0 | 3419 | 0.0 | 0 | 2635 | 200.1 | 0 | 2483 | 200.0 | 0 | 2515 | 200.1 |

Table A.22: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 100, instance numbers (Nr.) 0–14.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 100 | 0 | 0.10 | 0 | 2247 | 2.3 | 0 | 2943 | 0.0 | 0 | 2295 | 200.0 | 0 | 2272 | 200.0 | 0 | 2239 | 200.0 |
| | | 0.50 | 0 | 3207 | 14.8 | 0 | 4267 | 0.1 | 0 | 3186 | 200.0 | 0 | 3149 | 200.1 | 0 | 3129 | 200.0 |
| | | 0.80 | 0 | 4019 | 49.6 | 0 | 5723 | 0.1 | 0 | 3970 | 200.5 | 0 | 4115 | 200.2 | 0 | 4084 | 200.6 |
| | | 1.00 | 0 | 5066 | 40.7 | 0 | 6178 | 0.1 | 0 | 4918 | 200.2 | 0 | 4939 | 200.4 | 0 | 4945 | 200.2 |
| | 1 | 0.10 | 0 | 2788 | 2.3 | 0 | 3398 | 0.0 | 0 | 2765 | 200.0 | 0 | 2757 | 200.0 | 0 | 2757 | 200.0 |
| | | 0.50 | 0 | 4218 | 17.4 | 0 | 5398 | 0.1 | 0 | 4264 | 200.0 | 0 | 4259 | 200.1 | 0 | 4189 | 200.1 |
| | | 0.80 | 0 | 4778 | 28.0 | 0 | 6009 | 0.1 | 0 | 4979 | 200.1 | 0 | 4769 | 200.2 | 0 | 4739 | 200.2 |
| | | 1.00 | 0 | 5435 | 38.0 | 0 | 6742 | 0.6 | 0 | 5435 | 200.3 | 0 | 5417 | 200.4 | 0 | 5215 | 200.3 |
| | 2 | 0.10 | 0 | 1301 | 1.8 | 0 | 1531 | 0.0 | 0 | 1232 | 200.0 | 0 | 1219 | 200.0 | 0 | 1219 | 200.0 |
| | | 0.50 | 0 | 2975 | 18.3 | 0 | 4032 | 0.1 | 0 | 2940 | 200.1 | 0 | 2986 | 200.0 | 0 | 2939 | 200.1 |
| | | 0.80 | 0 | 3656 | 38.5 | 0 | 4956 | 0.1 | 0 | 3658 | 200.0 | 0 | 3748 | 200.1 | 0 | 3572 | 200.3 |
| | | 1.00 | 0 | 4405 | 42.0 | 0 | 5804 | 0.2 | 0 | 4562 | 200.4 | 0 | 4539 | 200.1 | 0 | 4492 | 200.1 |
| | 3 | 0.10 | 0 | 2504 | 3.9 | 0 | 3298 | 0.0 | 0 | 2364 | 200.0 | 0 | 2217 | 200.0 | 0 | 2197 | 200.0 |
| | | 0.50 | 0 | 3909 | 32.6 | 0 | 5929 | 0.1 | 0 | 3999 | 200.1 | 0 | 3936 | 200.3 | 0 | 3850 | 200.3 |
| | | 0.80 | 0 | 4655 | 49.3 | 0 | 6680 | 0.1 | 0 | 4692 | 200.3 | 0 | 4590 | 200.3 | 0 | 4573 | 200.1 |
| | | 1.00 | 0 | 5521 | 79.1 | 0 | 7346 | 0.2 | 0 | 5666 | 200.0 | 0 | 5544 | 200.1 | 0 | 5448 | 200.9 |
| | 4 | 0.10 | 0 | 1949 | 3.6 | 0 | 2699 | 0.0 | 0 | 1804 | 200.0 | 0 | 1785 | 200.0 | 0 | 1785 | 200.0 |
| | | 0.50 | 0 | 4006 | 24.2 | 0 | 5223 | 0.1 | 0 | 3954 | 200.3 | 0 | 3977 | 200.1 | 0 | 3811 | 200.3 |
| | | 0.80 | 0 | 4554 | 45.0 | 0 | 6061 | 0.2 | 0 | 4637 | 200.1 | 0 | 4543 | 200.1 | 0 | 4436 | 200.2 |
| | | 1.00 | 0 | 4961 | 74.1 | 0 | 6554 | 0.3 | 0 | 5288 | 200.1 | 0 | 5097 | 200.2 | 0 | 5173 | 200.0 |
| | 5 | 0.10 | 0 | 1928 | 1.4 | 0 | 2557 | 0.0 | 0 | 1917 | 200.0 | 0 | 1910 | 200.0 | 0 | 1910 | 200.0 |
| | | 0.50 | 0 | 3590 | 26.7 | 0 | 4799 | 0.1 | 0 | 3615 | 200.0 | 0 | 3628 | 200.2 | 0 | 3534 | 200.0 |
| | | 0.80 | 0 | 4311 | 35.5 | 0 | 5479 | 0.1 | 0 | 4371 | 200.2 | 0 | 4194 | 200.1 | 0 | 4046 | 200.0 |
| | | 1.00 | 0 | 4726 | 66.5 | 0 | 6065 | 0.2 | 140 | 4928 | 200.5 | 0 | 4828 | 200.6 | 0 | 4827 | 200.0 |
| | 6 | 0.10 | 0 | 2280 | 2.4 | 0 | 3207 | 0.0 | 0 | 2284 | 200.0 | 0 | 2296 | 200.1 | 0 | 2257 | 200.0 |
| | | 0.50 | 0 | 3723 | 21.1 | 0 | 5118 | 0.1 | 0 | 3725 | 200.2 | 0 | 3686 | 200.1 | 0 | 3618 | 200.0 |
| | | 0.80 | 0 | 4593 | 40.6 | 0 | 6052 | 0.1 | 0 | 4522 | 200.0 | 0 | 4539 | 200.0 | 0 | 4429 | 200.0 |
| | | 1.00 | 0 | 5492 | 53.4 | 0 | 7097 | 0.1 | 0 | 5561 | 200.0 | 0 | 5466 | 200.1 | 0 | 5456 | 200.0 |
| | 7 | 0.10 | 0 | 2755 | 2.9 | 0 | 4054 | 0.0 | 0 | 2596 | 200.1 | 0 | 2569 | 200.0 | 0 | 2556 | 200.1 |
| | | 0.50 | 0 | 3883 | 20.1 | 0 | 5304 | 0.1 | 0 | 3830 | 200.1 | 0 | 3901 | 200.3 | 0 | 3883 | 200.3 |
| | | 0.80 | 0 | 4635 | 46.7 | 0 | 6285 | 0.1 | 0 | 4732 | 200.2 | 0 | 4574 | 200.1 | 0 | 4607 | 200.1 |
| | | 1.00 | 0 | 5136 | 71.4 | 0 | 6874 | 0.2 | 0 | 5254 | 200.4 | 0 | 5188 | 200.2 | 0 | 5071 | 200.1 |
| | 8 | 0.10 | 0 | 1854 | 1.7 | 0 | 2444 | 0.0 | 0 | 1887 | 200.0 | 0 | 1765 | 200.0 | 0 | 1765 | 200.0 |
| | | 0.50 | 0 | 3362 | 23.2 | 0 | 4774 | 0.0 | 0 | 3488 | 200.1 | 0 | 3367 | 200.0 | 0 | 3292 | 200.1 |
| | | 0.80 | 0 | 4181 | 24.9 | 0 | 5539 | 0.1 | 0 | 4299 | 200.2 | 0 | 4160 | 200.0 | 0 | 4134 | 200.2 |
| | | 1.00 | 0 | 5158 | 41.0 | 0 | 6849 | 0.3 | 160 | 5347 | 200.2 | 0 | 5018 | 200.4 | 0 | 5082 | 200.0 |
| | 9 | 0.10 | 0 | 2085 | 3.7 | 0 | 2635 | 0.0 | 0 | 2029 | 200.0 | 0 | 2012 | 200.0 | 0 | 2024 | 200.0 |
| | | 0.50 | 0 | 3308 | 24.8 | 70 | 4095 | 25.5 | 0 | 3316 | 200.2 | 0 | 3310 | 200.1 | 0 | 3318 | 200.3 |
| | | 0.80 | 70 | 4081 | 39.6 | 70 | 4806 | 53.7 | 0 | 4229 | 200.0 | 0 | 4153 | 200.3 | 0 | 4178 | 200.0 |
| | | 1.00 | 70 | 4914 | 38.8 | 70 | 5650 | 48.7 | 0 | 4814 | 200.7 | 0 | 4824 | 200.1 | 0 | 5000 | 200.4 |
| | 10 | 0.10 | 0 | 2194 | 2.4 | 0 | 2947 | 0.0 | 0 | 2143 | 200.0 | 0 | 2143 | 200.0 | 0 | 2143 | 200.0 |
| | | 0.50 | 0 | 2945 | 14.0 | 0 | 4073 | 0.0 | 0 | 2998 | 200.0 | 0 | 2913 | 200.1 | 0 | 2835 | 200.0 |
| | | 0.80 | 0 | 4074 | 27.1 | 0 | 4961 | 0.1 | 0 | 4151 | 200.0 | 0 | 4128 | 200.1 | 0 | 3931 | 200.1 |
| | | 1.00 | 0 | 4771 | 40.3 | 0 | 5791 | 0.1 | 0 | 4706 | 200.0 | 0 | 4629 | 200.1 | 0 | 4706 | 200.1 |
| | 11 | 0.10 | 0 | 2786 | 4.6 | 0 | 4079 | 0.0 | 0 | 2784 | 200.0 | 0 | 2744 | 200.0 | 0 | 2744 | 200.0 |
| | | 0.50 | 0 | 4199 | 27.7 | 0 | 5654 | 0.1 | 0 | 4297 | 200.0 | 0 | 4332 | 200.0 | 0 | 4251 | 200.2 |
| | | 0.80 | 0 | 4987 | 52.6 | 0 | 6634 | 0.1 | 0 | 5138 | 200.4 | 0 | 5098 | 200.5 | 0 | 5056 | 200.2 |
| | | 1.00 | 0 | 5594 | 56.6 | 0 | 7202 | 12.6 | 0 | 5802 | 200.5 | 0 | 5549 | 200.6 | 0 | 5323 | 200.7 |
| | 12 | 0.10 | 0 | 2282 | 2.0 | 0 | 2896 | 0.0 | 0 | 2228 | 200.0 | 0 | 2206 | 200.0 | 0 | 2192 | 200.0 |
| | | 0.50 | 0 | 3420 | 14.3 | 0 | 5261 | 0.1 | 0 | 3444 | 200.1 | 0 | 3440 | 200.2 | 0 | 3393 | 200.1 |
| | | 0.80 | 0 | 4519 | 24.6 | 0 | 6353 | 0.1 | 0 | 4528 | 200.1 | 0 | 4411 | 200.1 | 0 | 4313 | 200.1 |
| | | 1.00 | 0 | 5680 | 26.8 | 0 | 7018 | 4.0 | 0 | 5615 | 200.2 | 0 | 5612 | 200.1 | 0 | 5405 | 200.1 |
| | 13 | 0.10 | 0 | 2192 | 1.3 | 0 | 2820 | 0.0 | 0 | 2212 | 200.0 | 0 | 2241 | 200.0 | 0 | 2158 | 200.0 |
| | | 0.50 | 0 | 4003 | 18.8 | 0 | 5228 | 0.1 | 0 | 3950 | 200.3 | 0 | 4028 | 200.1 | 0 | 3993 | 200.2 |
| | | 0.80 | 0 | 4892 | 40.9 | 0 | 6078 | 0.1 | 0 | 4912 | 200.2 | 0 | 4829 | 200.1 | 0 | 4759 | 200.1 |
| | | 1.00 | 0 | 5596 | 50.7 | 0 | 6755 | 0.1 | 0 | 5407 | 200.1 | 0 | 5494 | 200.5 | 0 | 5380 | 200.3 |
| | 14 | 0.10 | 0 | 2776 | 2.1 | 0 | 3330 | 0.0 | 0 | 2799 | 200.1 | 0 | 2733 | 200.0 | 0 | 2733 | 200.0 |
| | | 0.50 | 0 | 4090 | 20.0 | 0 | 5501 | 0.1 | 0 | 4136 | 200.2 | 0 | 4066 | 200.0 | 0 | 4068 | 200.3 |
| | | 0.80 | 0 | 4889 | 46.9 | 0 | 6377 | 0.1 | 0 | 4909 | 200.4 | 0 | 4822 | 200.1 | 0 | 4807 | 200.3 |
| | | 1.00 | 0 | 5379 | 61.5 | 0 | 6930 | 0.1 | 0 | 5492 | 200.2 | 0 | 5356 | 200.6 | 0 | 5316 | 200.6 |

Table A.23: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 100, instance numbers (Nr.) 15–29.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 100 | 15 | 0.10 | 0 | 1861 | 3.2 | 0 | 2591 | 0.0 | 0 | 1887 | 200.0 | 0 | 1850 | 200.0 | 0 | 1822 | 200.0 |
| | | 0.50 | 0 | 3293 | 11.6 | 0 | 4283 | 0.0 | 0 | 3183 | 200.0 | 0 | 3103 | 200.1 | 0 | 3064 | 200.1 |
| | | 0.80 | 0 | 4413 | 25.9 | 0 | 5504 | 0.1 | 0 | 4323 | 200.0 | 0 | 4317 | 200.1 | 0 | 4243 | 200.1 |
| | | 1.00 | 0 | 5251 | 25.5 | 0 | 6118 | 0.1 | 0 | 5347 | 200.4 | 0 | 5254 | 200.1 | 0 | 5221 | 200.0 |
| | 16 | 0.10 | 0 | 2531 | 4.7 | 0 | 3454 | 0.0 | 0 | 2530 | 200.1 | 0 | 2519 | 200.1 | 0 | 2499 | 200.1 |
| | | 0.50 | 0 | 3894 | 18.8 | 0 | 4885 | 0.1 | 0 | 3976 | 200.0 | 0 | 3957 | 200.2 | 0 | 3850 | 200.4 |
| | | 0.80 | 0 | 4493 | 47.9 | 0 | 5796 | 0.1 | 0 | 4466 | 200.0 | 0 | 4567 | 200.4 | 0 | 4527 | 200.1 |
| | | 1.00 | 0 | 5217 | 61.2 | 0 | 6609 | 0.3 | 0 | 5167 | 200.2 | 0 | 5198 | 200.3 | 0 | 5157 | 200.7 |
| | 17 | 0.10 | 0 | 2003 | 2.2 | 0 | 2744 | 0.0 | 0 | 2045 | 200.0 | 0 | 2003 | 200.0 | 0 | 2003 | 200.0 |
| | | 0.50 | 0 | 4339 | 16.7 | 0 | 5453 | 0.0 | 0 | 4226 | 200.1 | 0 | 4217 | 200.2 | 0 | 4151 | 200.0 |
| | | 0.80 | 0 | 4986 | 27.4 | 0 | 6123 | 0.1 | 0 | 4872 | 200.1 | 0 | 4693 | 200.4 | 0 | 4859 | 200.2 |
| | | 1.00 | 0 | 5791 | 31.0 | 0 | 6888 | 0.1 | 0 | 5905 | 200.1 | 0 | 5717 | 200.2 | 0 | 5702 | 200.2 |
| | 18 | 0.10 | 0 | 2002 | 1.5 | 0 | 2419 | 0.0 | 0 | 2013 | 200.0 | 0 | 1956 | 200.0 | 0 | 1945 | 200.0 |
| | | 0.50 | 0 | 3143 | 17.1 | 0 | 4376 | 0.1 | 0 | 3214 | 200.0 | 0 | 3158 | 200.0 | 0 | 3023 | 200.1 |
| | | 0.80 | 0 | 3914 | 28.1 | 0 | 5150 | 0.1 | 0 | 3863 | 200.1 | 0 | 3706 | 200.0 | 0 | 3641 | 200.2 |
| | | 1.00 | 115 | 4763 | 44.8 | 0 | 6356 | 0.7 | 0 | 4778 | 200.0 | 0 | 4699 | 200.2 | 0 | 4527 | 200.0 |
| | 19 | 0.10 | 0 | 2210 | 3.8 | 0 | 2947 | 0.0 | 0 | 2220 | 200.0 | 0 | 2219 | 200.0 | 0 | 2204 | 200.1 |
| | | 0.50 | 0 | 3514 | 21.8 | 0 | 4375 | 0.1 | 0 | 3451 | 200.0 | 0 | 3352 | 200.4 | 0 | 3346 | 200.2 |
| | | 0.80 | 0 | 4048 | 27.2 | 0 | 5060 | 0.1 | 0 | 4001 | 200.2 | 0 | 3977 | 200.4 | 0 | 3990 | 200.0 |
| | | 1.00 | 0 | 4543 | 43.5 | 0 | 5686 | 0.1 | 0 | 4374 | 200.4 | 0 | 4530 | 200.2 | 0 | 4359 | 200.0 |
| | 20 | 0.10 | 0 | 1630 | 0.8 | 0 | 2091 | 0.0 | 0 | 1609 | 200.0 | 0 | 1548 | 200.0 | 0 | 1548 | 200.0 |
| | | 0.50 | 0 | 3648 | 15.5 | 0 | 5106 | 0.0 | 0 | 3607 | 200.0 | 0 | 3600 | 200.0 | 0 | 3544 | 200.2 |
| | | 0.80 | 0 | 4338 | 37.8 | 0 | 6065 | 0.1 | 0 | 4366 | 200.0 | 0 | 4262 | 200.0 | 0 | 4170 | 200.3 |
| | | 1.00 | 0 | 4963 | 52.1 | 0 | 6644 | 0.2 | 0 | 5001 | 200.0 | 0 | 4853 | 200.4 | 0 | 4711 | 200.0 |
| | 21 | 0.10 | 0 | 1973 | 4.3 | 0 | 2866 | 0.0 | 0 | 2015 | 200.0 | 0 | 1950 | 200.0 | 0 | 1923 | 200.0 |
| | | 0.50 | 0 | 3414 | 19.7 | 0 | 4798 | 0.1 | 0 | 3440 | 200.1 | 0 | 3434 | 200.0 | 0 | 3434 | 200.2 |
| | | 0.80 | 0 | 4249 | 41.9 | 0 | 5769 | 0.1 | 0 | 4158 | 200.2 | 0 | 4154 | 200.5 | 0 | 4107 | 200.1 |
| | | 1.00 | 0 | 4708 | 46.6 | 0 | 6111 | 0.1 | 0 | 4815 | 200.0 | 0 | 4570 | 200.2 | 0 | 4570 | 200.1 |
| | 22 | 0.10 | 0 | 1097 | 0.6 | 0 | 1238 | 0.0 | 0 | 1065 | 200.0 | 0 | 1065 | 200.0 | 0 | 1065 | 200.0 |
| | | 0.50 | 0 | 2882 | 17.4 | 0 | 4200 | 0.0 | 0 | 2961 | 200.1 | 0 | 2855 | 200.0 | 0 | 2863 | 200.0 |
| | | 0.80 | 0 | 3881 | 35.2 | 0 | 5445 | 0.1 | 0 | 3958 | 200.3 | 0 | 3869 | 200.3 | 0 | 3901 | 200.1 |
| | | 1.00 | 0 | 4395 | 38.6 | 0 | 6033 | 0.1 | 0 | 4510 | 200.2 | 0 | 4365 | 200.2 | 0 | 4282 | 200.3 |
| | 23 | 0.10 | 0 | 1572 | 1.6 | 0 | 2254 | 0.0 | 0 | 1603 | 200.0 | 0 | 1518 | 200.0 | 0 | 1519 | 200.0 |
| | | 0.50 | 0 | 3084 | 17.9 | 0 | 4672 | 0.1 | 0 | 3023 | 200.0 | 0 | 2959 | 200.2 | 0 | 3018 | 200.0 |
| | | 0.80 | 0 | 4064 | 40.3 | 0 | 5685 | 0.1 | 0 | 4282 | 200.2 | 0 | 4160 | 200.3 | 0 | 4109 | 200.0 |
| | | 1.00 | 0 | 5315 | 39.7 | 0 | 6413 | 0.1 | 0 | 5420 | 200.4 | 0 | 5312 | 200.2 | 0 | 5134 | 200.1 |
| | 24 | 0.10 | 0 | 2584 | 2.1 | 0 | 3233 | 0.0 | 0 | 2503 | 200.0 | 0 | 2490 | 200.0 | 0 | 2482 | 200.1 |
| | | 0.50 | 0 | 3991 | 18.3 | 0 | 5237 | 0.0 | 0 | 3992 | 200.2 | 0 | 4020 | 200.2 | 0 | 3886 | 200.0 |
| | | 0.80 | 0 | 4810 | 31.2 | 0 | 6442 | 0.1 | 0 | 4707 | 200.1 | 0 | 4606 | 200.3 | 0 | 4554 | 200.2 |
| | | 1.00 | 0 | 4948 | 62.0 | 0 | 6741 | 0.1 | 0 | 5252 | 200.2 | 0 | 4988 | 200.1 | 0 | 4853 | 200.0 |
| | 25 | 0.10 | 0 | 2088 | 2.2 | 0 | 2535 | 0.0 | 0 | 2088 | 200.0 | 0 | 1940 | 200.0 | 0 | 1976 | 200.1 |
| | | 0.50 | 0 | 3587 | 23.4 | 0 | 4569 | 0.1 | 0 | 3539 | 200.0 | 0 | 3440 | 200.1 | 0 | 3518 | 200.3 |
| | | 0.80 | 0 | 4648 | 34.0 | 0 | 5558 | 0.1 | 0 | 4479 | 200.2 | 0 | 4555 | 200.2 | 0 | 4453 | 200.4 |
| | | 1.00 | 0 | 5150 | 51.6 | 0 | 6758 | 0.1 | 0 | 5145 | 200.4 | 0 | 5020 | 200.0 | 0 | 4860 | 200.4 |
| | 26 | 0.10 | 0 | 2384 | 1.6 | 0 | 3194 | 0.0 | 0 | 2357 | 200.0 | 0 | 2353 | 200.0 | 0 | 2360 | 200.0 |
| | | 0.50 | 0 | 3753 | 9.1 | 0 | 4552 | 0.0 | 0 | 3735 | 200.1 | 0 | 3751 | 200.1 | 0 | 3590 | 200.0 |
| | | 0.80 | 0 | 4535 | 33.5 | 0 | 6142 | 0.1 | 0 | 4652 | 200.2 | 0 | 4458 | 200.0 | 0 | 4355 | 200.3 |
| | | 1.00 | 0 | 5091 | 36.2 | 0 | 6596 | 0.1 | 0 | 5103 | 200.1 | 0 | 4954 | 200.2 | 0 | 5081 | 200.5 |
| | 27 | 0.10 | 0 | 1690 | 1.1 | 0 | 2153 | 0.0 | 0 | 1465 | 200.0 | 0 | 1480 | 200.0 | 0 | 1445 | 200.0 |
| | | 0.50 | 0 | 3175 | 13.2 | 0 | 4531 | 0.0 | 0 | 3221 | 200.1 | 0 | 3122 | 200.1 | 0 | 3013 | 200.0 |
| | | 0.80 | 0 | 4371 | 21.6 | 0 | 5636 | 0.1 | 0 | 4378 | 200.2 | 0 | 4197 | 200.2 | 0 | 4137 | 200.3 |
| | | 1.00 | 255 | 5031 | 27.2 | 95 | 5567 | 59.4 | 0 | 4777 | 200.2 | 0 | 4789 | 200.1 | 0 | 4688 | 200.4 |
| | 28 | 0.10 | 0 | 2265 | 2.8 | 0 | 2976 | 0.0 | 0 | 2178 | 200.0 | 0 | 2219 | 200.0 | 0 | 2219 | 200.0 |
| | | 0.50 | 0 | 4364 | 17.6 | 0 | 5456 | 0.1 | 0 | 4292 | 200.0 | 0 | 4277 | 200.2 | 0 | 4202 | 200.0 |
| | | 0.80 | 0 | 4693 | 38.7 | 0 | 6154 | 0.1 | 0 | 4675 | 200.2 | 0 | 4658 | 200.4 | 0 | 4629 | 200.4 |
| | | 1.00 | 0 | 5401 | 47.2 | 0 | 7019 | 0.1 | 0 | 5306 | 200.1 | 0 | 5519 | 200.1 | 0 | 5115 | 200.7 |
| | 29 | 0.10 | 0 | 2523 | 3.0 | 0 | 3087 | 0.0 | 0 | 2403 | 200.0 | 0 | 2421 | 200.0 | 0 | 2416 | 200.0 |
| | | 0.50 | 0 | 3622 | 24.4 | 0 | 5318 | 0.1 | 0 | 3616 | 200.2 | 0 | 3580 | 200.0 | 0 | 3501 | 200.0 |
| | | 0.80 | 0 | 4549 | 48.3 | 0 | 6258 | 0.1 | 0 | 4663 | 200.1 | 0 | 4577 | 200.1 | 0 | 4454 | 200.3 |
| | | 1.00 | 0 | 5045 | 49.6 | 0 | 6537 | 0.3 | 0 | 5156 | 200.4 | 0 | 4987 | 200.4 | 0 | 4948 | 200.1 |

Table A.24: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 200, instance numbers (Nr.) 0–14.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 200 | 0 | 0.10 | 0 | 2277 | 6.7 | 0 | 3035 | 0.0 | 0 | 2247 | 500.0 | 0 | 2265 | 500.0 | 0 | 2222 | 500.1 |
| | | 0.50 | 0 | 4616 | 65.1 | 0 | 6799 | 0.1 | 0 | 4760 | 500.2 | 0 | 4631 | 500.5 | 0 | 4612 | 500.4 |
| | | 0.80 | 0 | 6050 | 112.2 | 0 | 8179 | 0.1 | 0 | 6173 | 500.0 | 0 | 6047 | 500.1 | 0 | 5811 | 500.2 |
| | | 1.00 | 0 | 7003 | 121.9 | 0 | 9502 | 0.2 | 0 | 6885 | 500.1 | 0 | 7104 | 500.1 | 0 | 6857 | 500.3 |
| | 1 | 0.10 | 0 | 1915 | 2.7 | 0 | 2535 | 0.0 | 0 | 1879 | 500.0 | 0 | 1891 | 500.1 | 0 | 1858 | 500.1 |
| | | 0.50 | 0 | 4132 | 70.4 | 0 | 5974 | 0.1 | 0 | 4334 | 500.1 | 0 | 4302 | 500.3 | 0 | 4212 | 500.6 |
| | | 0.80 | 0 | 5706 | 159.1 | 0 | 7391 | 11.1 | 0 | 5932 | 500.8 | 0 | 5834 | 500.6 | 0 | 5904 | 500.1 |
| | | 1.00 | 0 | 7601 | 137.1 | 0 | 9234 | 20.0 | 0 | 7613 | 500.0 | 0 | 7394 | 500.6 | 0 | 7189 | 500.9 |
| | 2 | 0.10 | 0 | 2669 | 6.4 | 0 | 3478 | 0.0 | 0 | 2673 | 500.1 | 0 | 2650 | 500.1 | 0 | 2677 | 500.0 |
| | | 0.50 | 0 | 4261 | 31.6 | 0 | 5910 | 0.1 | 0 | 4389 | 500.2 | 0 | 4203 | 500.1 | 0 | 4135 | 500.3 |
| | | 0.80 | 0 | 5398 | 153.0 | 0 | 7485 | 0.2 | 0 | 5649 | 500.6 | 0 | 5516 | 500.0 | 0 | 5278 | 500.4 |
| | | 1.00 | 0 | 6676 | 178.2 | 0 | 9725 | 0.7 | 0 | 7036 | 500.2 | 0 | 7051 | 500.9 | 0 | 6722 | 500.5 |
| | 3 | 0.10 | 0 | 3699 | 13.6 | 0 | 4918 | 0.0 | 0 | 3612 | 500.1 | 0 | 3650 | 500.2 | 0 | 3579 | 500.0 |
| | | 0.50 | 0 | 5850 | 85.5 | 0 | 7894 | 30.9 | 0 | 5968 | 500.4 | 0 | 6144 | 500.2 | 0 | 5815 | 500.6 |
| | | 0.80 | 0 | 6965 | 139.6 | 0 | 9359 | 72.9 | 0 | 7104 | 500.2 | 0 | 7017 | 500.1 | 0 | 6921 | 500.0 |
| | | 1.00 | 100 | 8081 | 159.8 | 100 | 8951 | 363.9 | 0 | 8190 | 500.3 | 0 | 7952 | 500.3 | 100 | 7946 | 501.0 |
| | 4 | 0.10 | 0 | 2921 | 10.0 | 0 | 3805 | 0.0 | 0 | 2896 | 500.0 | 0 | 2882 | 500.0 | 0 | 2854 | 500.1 |
| | | 0.50 | 0 | 4870 | 41.4 | 0 | 6757 | 0.1 | 0 | 5080 | 500.4 | 0 | 4905 | 500.4 | 0 | 4819 | 500.3 |
| | | 0.80 | 0 | 6398 | 174.8 | 0 | 9562 | 0.1 | 0 | 6845 | 500.2 | 0 | 6524 | 500.1 | 0 | 6320 | 500.2 |
| | | 1.00 | 0 | 8240 | 218.1 | 0 | 11288 | 0.7 | 0 | 8569 | 500.1 | 0 | 8111 | 500.9 | 0 | 7991 | 501.3 |
| | 5 | 0.10 | 0 | 3012 | 12.4 | 0 | 4086 | 0.0 | 0 | 3089 | 500.1 | 0 | 2994 | 500.0 | 0 | 2994 | 500.0 |
| | | 0.50 | 0 | 5150 | 100.4 | 0 | 7151 | 6.0 | 0 | 5217 | 500.0 | 0 | 5298 | 500.4 | 0 | 5182 | 500.8 |
| | | 0.80 | 0 | 6435 | 224.4 | 0 | 8654 | 22.6 | 40 | 6840 | 500.1 | 10 | 6681 | 500.4 | 0 | 6516 | 500.1 |
| | | 1.00 | 20 | 7388 | 353.2 | 20 | 8429 | 523.5 | 0 | 7601 | 500.4 | 130 | 7742 | 500.0 | 40 | 7289 | 500.6 |
| | 6 | 0.10 | 0 | 3334 | 20.6 | 0 | 5047 | 0.0 | 0 | 3411 | 500.2 | 0 | 3262 | 500.2 | 0 | 3310 | 500.0 |
| | | 0.50 | 0 | 5307 | 92.3 | 0 | 7907 | 0.1 | 0 | 5546 | 500.5 | 0 | 5280 | 500.8 | 0 | 5272 | 500.0 |
| | | 0.80 | 0 | 6829 | 147.3 | 0 | 9469 | 2.7 | 0 | 7064 | 501.0 | 0 | 6929 | 501.0 | 0 | 6818 | 500.6 |
| | | 1.00 | 0 | 8076 | 191.4 | 0 | 10897 | 6.3 | 0 | 8231 | 501.4 | 0 | 7962 | 500.5 | 0 | 8030 | 500.3 |
| | 7 | 0.10 | 0 | 2883 | 10.0 | 0 | 4504 | 0.0 | 0 | 3025 | 500.0 | 0 | 2932 | 500.0 | 0 | 2802 | 500.1 |
| | | 0.50 | 0 | 5657 | 72.4 | 0 | 8369 | 0.1 | 0 | 5808 | 500.0 | 0 | 5640 | 500.2 | 0 | 5562 | 500.5 |
| | | 0.80 | 0 | 7221 | 172.1 | 0 | 10621 | 0.2 | 0 | 7499 | 500.3 | 0 | 7371 | 500.6 | 0 | 7131 | 500.0 |
| | | 1.00 | 0 | 8518 | 188.2 | 0 | 11595 | 15.8 | 0 | 8917 | 501.2 | 0 | 8705 | 500.2 | 0 | 8524 | 500.9 |
| | 8 | 0.10 | 0 | 1530 | 1.9 | 0 | 2046 | 0.0 | 0 | 1511 | 500.0 | 0 | 1520 | 500.0 | 0 | 1511 | 500.0 |
| | | 0.50 | 0 | 4746 | 56.4 | 0 | 7117 | 0.1 | 0 | 4851 | 500.1 | 0 | 4818 | 500.2 | 0 | 4779 | 500.1 |
| | | 0.80 | 0 | 6262 | 144.8 | 0 | 9732 | 13.2 | 0 | 6468 | 501.0 | 0 | 6449 | 500.3 | 0 | 6305 | 500.1 |
| | | 1.00 | 0 | 8083 | 159.1 | 0 | 10777 | 23.3 | 0 | 8337 | 500.2 | 0 | 8067 | 500.3 | 0 | 8133 | 500.0 |
| | 9 | 0.10 | 0 | 2264 | 3.5 | 0 | 2765 | 0.0 | 0 | 2241 | 500.0 | 0 | 2264 | 500.0 | 0 | 2262 | 500.0 |
| | | 0.50 | 0 | 5070 | 83.8 | 0 | 7299 | 7.3 | 0 | 5249 | 500.2 | 0 | 5184 | 500.4 | 0 | 5108 | 500.1 |
| | | 0.80 | 0 | 5890 | 254.8 | 0 | 8466 | 13.7 | 0 | 6078 | 500.4 | 0 | 6141 | 500.1 | 0 | 5924 | 500.2 |
| | | 1.00 | 0 | 7854 | 312.9 | 0 | 10055 | 20.0 | 0 | 8003 | 500.5 | 0 | 7813 | 500.4 | 0 | 7952 | 500.7 |
| | 10 | 0.10 | 0 | 1792 | 1.6 | 0 | 2375 | 0.0 | 0 | 1810 | 500.0 | 0 | 1734 | 500.0 | 0 | 1734 | 500.0 |
| | | 0.50 | 0 | 5040 | 34.6 | 0 | 6830 | 0.1 | 0 | 4944 | 500.2 | 0 | 4919 | 500.2 | 0 | 4812 | 500.0 |
| | | 0.80 | 0 | 6718 | 149.6 | 0 | 9066 | 0.1 | 0 | 6947 | 500.2 | 0 | 6722 | 500.3 | 0 | 6720 | 500.5 |
| | | 1.00 | 0 | 7897 | 154.0 | 0 | 10630 | 1.4 | 0 | 8236 | 501.1 | 0 | 7828 | 500.1 | 0 | 7894 | 500.8 |
| | 11 | 0.10 | 0 | 2070 | 2.6 | 0 | 2747 | 0.0 | 0 | 1999 | 500.0 | 0 | 1889 | 500.0 | 0 | 1869 | 500.0 |
| | | 0.50 | 0 | 4961 | 73.8 | 0 | 7254 | 0.1 | 0 | 5067 | 500.4 | 0 | 5032 | 500.8 | 0 | 5005 | 500.2 |
| | | 0.80 | 0 | 7695 | 184.8 | 0 | 10777 | 0.2 | 0 | 7474 | 500.3 | 0 | 7669 | 500.0 | 0 | 7343 | 500.4 |
| | | 1.00 | 0 | 8669 | 304.8 | 0 | 12127 | 0.3 | 0 | 9074 | 501.4 | 0 | 8704 | 501.2 | 0 | 8696 | 500.3 |
| | 12 | 0.10 | 0 | 2733 | 7.6 | 0 | 3761 | 0.0 | 0 | 2719 | 500.0 | 0 | 2718 | 500.1 | 0 | 2704 | 500.0 |
| | | 0.50 | 0 | 4635 | 57.0 | 0 | 6317 | 0.1 | 0 | 4781 | 500.1 | 0 | 4749 | 500.2 | 0 | 4630 | 500.0 |
| | | 0.80 | 0 | 5735 | 92.8 | 0 | 7884 | 0.2 | 0 | 5758 | 500.2 | 0 | 5745 | 500.2 | 0 | 5593 | 500.3 |
| | | 1.00 | 0 | 6990 | 225.8 | 0 | 9638 | 0.2 | 0 | 7257 | 500.8 | 0 | 7278 | 500.7 | 0 | 7070 | 500.0 |
| | 13 | 0.10 | 0 | 2209 | 2.9 | 0 | 2706 | 0.0 | 0 | 2201 | 500.0 | 0 | 2206 | 500.0 | 0 | 2201 | 500.0 |
| | | 0.50 | 0 | 3819 | 63.7 | 0 | 5247 | 0.1 | 0 | 3953 | 500.1 | 0 | 3873 | 500.4 | 0 | 3853 | 500.0 |
| | | 0.80 | 0 | 5031 | 149.3 | 0 | 7815 | 0.2 | 0 | 5204 | 500.2 | 0 | 5173 | 500.1 | 0 | 5055 | 500.3 |
| | | 1.00 | 0 | 6416 | 207.2 | 0 | 9871 | 0.5 | 0 | 6744 | 500.4 | 0 | 6442 | 500.8 | 0 | 6119 | 500.4 |
| | 14 | 0.10 | 0 | 3292 | 18.4 | 0 | 4318 | 0.0 | 0 | 3339 | 500.1 | 0 | 3412 | 500.3 | 0 | 3279 | 500.1 |
| | | 0.50 | 0 | 4803 | 63.0 | 0 | 6509 | 0.1 | 0 | 5023 | 500.0 | 0 | 4818 | 500.0 | 0 | 4932 | 500.4 |
| | | 0.80 | 0 | 6857 | 160.9 | 0 | 8785 | 0.2 | 0 | 7093 | 501.5 | 0 | 7032 | 500.9 | 0 | 6918 | 500.2 |
| | | 1.00 | 0 | 7482 | 186.6 | 0 | 9518 | 0.4 | 0 | 7678 | 501.7 | 0 | 7696 | 500.3 | 0 | 7639 | 501.6 |

Table A.25: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 200, instance numbers (Nr.) 15–29.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 200 | 15 | 0.10 | 0 | 2631 | 5.3 | 0 | 3045 | 0.0 | 0 | 2562 | 500.1 | 0 | 2546 | 500.0 | 0 | 2554 | 500.1 |
| | | 0.50 | 0 | 4419 | 58.1 | 0 | 6125 | 0.1 | 0 | 4485 | 500.2 | 0 | 4496 | 500.1 | 0 | 4355 | 500.2 |
| | | 0.80 | 0 | 5506 | 164.4 | 0 | 7859 | 0.1 | 0 | 5832 | 500.4 | 0 | 5706 | 500.9 | 0 | 5676 | 500.0 |
| | | 1.00 | 0 | 6968 | 134.0 | 0 | 9258 | 15.9 | 5 | 7264 | 500.9 | 0 | 6944 | 500.7 | 0 | 6745 | 500.0 |
| | 16 | 0.10 | 0 | 2700 | 6.3 | 0 | 3571 | 0.0 | 0 | 2694 | 500.0 | 0 | 2689 | 500.1 | 0 | 2652 | 500.0 |
| | | 0.50 | 0 | 4103 | 46.9 | 0 | 5862 | 0.1 | 0 | 4171 | 500.5 | 0 | 4124 | 500.1 | 0 | 4009 | 500.4 |
| | | 0.80 | 0 | 5450 | 158.5 | 0 | 8900 | 0.2 | 0 | 5735 | 500.1 | 0 | 5682 | 500.7 | 0 | 5660 | 500.2 |
| | | 1.00 | 0 | 7128 | 214.3 | 0 | 10720 | 22.1 | 0 | 7504 | 501.0 | 0 | 7558 | 500.1 | 0 | 7261 | 501.6 |
| | 17 | 0.10 | 0 | 1943 | 4.4 | 0 | 2837 | 0.0 | 0 | 1851 | 500.0 | 0 | 1851 | 500.1 | 0 | 1851 | 500.1 |
| | | 0.50 | 0 | 5287 | 82.7 | 0 | 7698 | 0.1 | 0 | 5286 | 500.5 | 0 | 5300 | 500.2 | 0 | 5094 | 500.5 |
| | | 0.80 | 0 | 7095 | 141.2 | 0 | 9643 | 0.2 | 0 | 7310 | 500.5 | 0 | 7183 | 500.3 | 0 | 7132 | 500.6 |
| | | 1.00 | 0 | 8576 | 251.3 | 0 | 11573 | 1.1 | 0 | 8770 | 500.2 | 0 | 8654 | 500.8 | 0 | 8365 | 500.9 |
| | 18 | 0.10 | 0 | 2159 | 12.4 | 0 | 2980 | 0.0 | 0 | 2065 | 500.1 | 0 | 2060 | 500.0 | 0 | 2042 | 500.1 |
| | | 0.50 | 0 | 4635 | 80.4 | 0 | 6669 | 0.1 | 0 | 4624 | 500.8 | 0 | 4568 | 500.1 | 0 | 4521 | 500.8 |
| | | 0.80 | 0 | 6442 | 172.7 | 0 | 8854 | 0.3 | 0 | 6364 | 500.4 | 0 | 6427 | 500.4 | 0 | 6265 | 500.8 |
| | | 1.00 | 0 | 7457 | 249.5 | 0 | 9800 | 28.6 | 0 | 7434 | 500.5 | 0 | 7272 | 501.4 | 0 | 7615 | 501.7 |
| | 19 | 0.10 | 0 | 2414 | 4.1 | 0 | 2889 | 0.0 | 0 | 2394 | 500.0 | 0 | 2415 | 500.1 | 0 | 2402 | 500.1 |
| | | 0.50 | 0 | 4347 | 35.7 | 0 | 5502 | 0.1 | 0 | 4297 | 500.1 | 0 | 4301 | 500.2 | 0 | 4182 | 500.1 |
| | | 0.80 | 0 | 6480 | 116.2 | 0 | 9378 | 0.3 | 0 | 6653 | 500.1 | 0 | 6710 | 500.9 | 0 | 6438 | 500.7 |
| | | 1.00 | 0 | 7581 | 119.4 | 0 | 10351 | 14.3 | 0 | 7742 | 500.1 | 0 | 7662 | 500.3 | 0 | 7374 | 500.3 |
| | 20 | 0.10 | 0 | 2627 | 8.2 | 0 | 4118 | 0.0 | 0 | 2619 | 500.1 | 0 | 2640 | 500.1 | 0 | 2604 | 500.1 |
| | | 0.50 | 0 | 5076 | 62.1 | 0 | 7554 | 0.1 | 0 | 5013 | 500.1 | 0 | 4971 | 500.2 | 0 | 4895 | 500.6 |
| | | 0.80 | 0 | 6384 | 168.6 | 0 | 9645 | 14.3 | 0 | 6743 | 500.7 | 25 | 6682 | 500.8 | 0 | 6561 | 500.0 |
| | | 1.00 | 25 | 8128 | 198.2 | 0 | 10652 | 69.4 | 0 | 8345 | 500.8 | 25 | 8162 | 501.3 | 588 | 8009 | 500.4 |
| | 21 | 0.10 | 0 | 2512 | 7.9 | 0 | 3518 | 0.0 | 0 | 2516 | 500.1 | 0 | 2545 | 500.1 | 0 | 2481 | 500.0 |
| | | 0.50 | 0 | 4811 | 52.4 | 0 | 6612 | 0.1 | 0 | 4878 | 500.2 | 0 | 4812 | 500.1 | 0 | 4758 | 500.4 |
| | | 0.80 | 0 | 6539 | 151.4 | 0 | 8874 | 0.2 | 0 | 6880 | 501.2 | 0 | 6510 | 500.1 | 0 | 6788 | 500.3 |
| | | 1.00 | 0 | 7445 | 199.1 | 0 | 9870 | 15.0 | 0 | 8055 | 501.1 | 0 | 7453 | 500.5 | 0 | 7908 | 501.0 |
| | 22 | 0.10 | 0 | 3046 | 8.5 | 0 | 4213 | 0.0 | 0 | 3084 | 500.0 | 0 | 3031 | 500.0 | 0 | 2997 | 500.0 |
| | | 0.50 | 0 | 5436 | 66.5 | 0 | 7159 | 0.1 | 0 | 5342 | 500.7 | 0 | 5350 | 500.1 | 0 | 5321 | 500.1 |
| | | 0.80 | 0 | 6731 | 140.1 | 0 | 8708 | 0.2 | 0 | 6809 | 501.0 | 0 | 6732 | 500.0 | 0 | 6598 | 500.2 |
| | | 1.00 | 0 | 7742 | 222.7 | 0 | 10890 | 6.0 | 0 | 7871 | 500.1 | 0 | 7853 | 500.5 | 0 | 7851 | 501.2 |
| | 23 | 0.10 | 0 | 2571 | 11.7 | 0 | 4196 | 0.0 | 0 | 2699 | 500.0 | 0 | 2797 | 500.0 | 0 | 2571 | 500.1 |
| | | 0.50 | 0 | 5191 | 58.2 | 0 | 8257 | 0.1 | 0 | 5352 | 500.5 | 0 | 5318 | 500.0 | 0 | 5025 | 500.0 |
| | | 0.80 | 0 | 6835 | 120.3 | 0 | 9738 | 0.2 | 0 | 6793 | 500.4 | 0 | 6532 | 500.2 | 0 | 6343 | 500.5 |
| | | 1.00 | 0 | 9031 | 256.4 | 0 | 11393 | 16.0 | 0 | 9102 | 500.0 | 175 | 8972 | 500.0 | 0 | 9171 | 501.1 |
| | 24 | 0.10 | 0 | 2278 | 2.7 | 0 | 2970 | 0.0 | 0 | 2209 | 500.0 | 0 | 2208 | 500.0 | 0 | 2207 | 500.0 |
| | | 0.50 | 0 | 5032 | 63.5 | 0 | 7111 | 0.1 | 0 | 5175 | 500.0 | 0 | 5009 | 500.2 | 0 | 4771 | 500.1 |
| | | 0.80 | 0 | 5899 | 134.6 | 0 | 8662 | 0.2 | 0 | 6176 | 500.3 | 0 | 5994 | 500.0 | 0 | 6009 | 500.1 |
| | | 1.00 | 0 | 8010 | 216.9 | 0 | 10681 | 3.5 | 0 | 7771 | 500.2 | 0 | 7780 | 500.0 | 0 | 7778 | 500.4 |
| | 25 | 0.10 | 0 | 2919 | 8.8 | 0 | 3957 | 0.0 | 0 | 2866 | 500.0 | 0 | 2829 | 500.0 | 0 | 2805 | 500.1 |
| | | 0.50 | 0 | 5100 | 69.8 | 0 | 7152 | 6.5 | 0 | 4992 | 500.1 | 0 | 5001 | 500.2 | 0 | 4913 | 500.4 |
| | | 0.80 | 0 | 6154 | 150.8 | 0 | 8577 | 20.3 | 0 | 6203 | 500.0 | 0 | 6108 | 500.2 | 0 | 6136 | 500.1 |
| | | 1.00 | 0 | 7074 | 218.7 | 0 | 9729 | 23.3 | 0 | 7364 | 500.6 | 0 | 7330 | 500.0 | 0 | 7286 | 500.3 |
| | 26 | 0.10 | 0 | 2992 | 12.6 | 0 | 3590 | 0.0 | 0 | 2997 | 500.0 | 0 | 2943 | 500.1 | 0 | 2920 | 500.2 |
| | | 0.50 | 0 | 4498 | 87.6 | 0 | 5922 | 0.1 | 0 | 4598 | 500.4 | 0 | 4608 | 500.0 | 0 | 4469 | 500.7 |
| | | 0.80 | 0 | 6348 | 152.8 | 0 | 8519 | 12.4 | 0 | 6408 | 501.0 | 0 | 6430 | 500.3 | 0 | 6247 | 500.1 |
| | | 1.00 | 0 | 7398 | 219.2 | 0 | 9393 | 16.1 | 0 | 7635 | 501.3 | 0 | 7476 | 501.5 | 0 | 7371 | 501.3 |
| | 27 | 0.10 | 0 | 2555 | 1.9 | 0 | 3166 | 0.0 | 0 | 2437 | 500.0 | 0 | 2478 | 500.1 | 0 | 2478 | 500.0 |
| | | 0.50 | 0 | 4632 | 48.3 | 0 | 6317 | 0.1 | 0 | 4643 | 500.1 | 0 | 4611 | 500.2 | 0 | 4620 | 500.2 |
| | | 0.80 | 0 | 5823 | 149.2 | 0 | 8099 | 0.2 | 0 | 6041 | 500.3 | 0 | 6013 | 500.0 | 0 | 5923 | 500.3 |
| | | 1.00 | 0 | 6655 | 202.7 | 0 | 9045 | 14.2 | 0 | 6889 | 500.3 | 0 | 6753 | 501.1 | 0 | 6815 | 500.2 |
| | 28 | 0.10 | 0 | 2630 | 7.1 | 0 | 3615 | 0.0 | 0 | 2545 | 500.0 | 0 | 2389 | 500.0 | 0 | 2397 | 500.0 |
| | | 0.50 | 0 | 4272 | 56.3 | 0 | 5952 | 0.1 | 0 | 4189 | 500.2 | 0 | 4151 | 500.1 | 0 | 4052 | 500.2 |
| | | 0.80 | 0 | 5713 | 114.1 | 0 | 8237 | 10.8 | 0 | 6128 | 500.2 | 0 | 5796 | 500.8 | 0 | 5845 | 500.7 |
| | | 1.00 | 0 | 7744 | 194.4 | 0 | 9275 | 187.7 | 0 | 8211 | 500.7 | 0 | 7808 | 500.6 | 0 | 7655 | 500.0 |
| | 29 | 0.10 | 0 | 2531 | 7.0 | 0 | 3760 | 0.0 | 0 | 2455 | 500.0 | 0 | 2391 | 500.2 | 0 | 2368 | 500.2 |
| | | 0.50 | 0 | 4826 | 47.9 | 0 | 7198 | 0.1 | 0 | 4894 | 500.4 | 0 | 4634 | 500.0 | 0 | 4676 | 500.2 |
| | | 0.80 | 0 | 7064 | 138.6 | 0 | 9158 | 0.3 | 0 | 6956 | 500.6 | 0 | 6770 | 500.8 | 0 | 6697 | 501.0 |
| | | 1.00 | 0 | 7805 | 288.2 | 0 | 9998 | 135.4 | 0 | 7977 | 500.5 | 0 | 7833 | 500.3 | 0 | 7595 | 500.4 |

Table A.26: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 500, instance numbers (Nr.) 0–14.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 500 | 0 | 0.10 | 0 | 3876 | 42.0 | 0 | 4988 | 0.1 | 0 | 3838 | 500.2 | 0 | 3846 | 500.0 | 0 | 3793 | 500.0 |
| | | 0.50 | 0 | 6952 | 229.7 | 0 | 9727 | 0.2 | 0 | 7167 | 500.1 | 0 | 7172 | 500.6 | 0 | 7118 | 500.6 |
| | | 0.80 | 0 | 9165 | 887.1 | 0 | 13114 | 0.5 | 0 | 9530 | 501.8 | 0 | 9787 | 500.3 | 0 | 9582 | 502.6 |
| | | 1.00 | 26 | 12890 | 1009.9 | 26 | 15806 | 1001.1 | 194 | 13827 | 500.2 | 231 | 13924 | 501.4 | 231 | 13062 | 501.4 |
| | 1 | 0.10 | 0 | 3749 | 51.1 | 0 | 4910 | 0.0 | 0 | 3828 | 500.1 | 0 | 3934 | 500.3 | 0 | 3817 | 500.3 |
| | | 0.50 | 0 | 6510 | 309.9 | 0 | 9735 | 0.2 | 0 | 6953 | 500.0 | 0 | 6813 | 500.3 | 0 | 6818 | 501.4 |
| | | 0.80 | 0 | 8452 | 643.1 | 0 | 13259 | 36.6 | 0 | 9321 | 500.4 | 0 | 9019 | 500.4 | 0 | 8982 | 502.8 |
| | | 1.00 | 35 | 11716 | 896.5 | 35 | 14438 | 1001.4 | 55 | 12580 | 500.4 | 13 | 13087 | 502.3 | 37 | 11838 | 503.5 |
| | 2 | 0.10 | 0 | 3475 | 53.3 | 0 | 4820 | 0.1 | 0 | 3520 | 500.1 | 0 | 3588 | 500.4 | 0 | 3384 | 500.1 |
| | | 0.50 | 0 | 6501 | 215.9 | 0 | 9112 | 0.2 | 0 | 6829 | 500.4 | 0 | 6722 | 500.3 | 0 | 6762 | 500.1 |
| | | 0.80 | 0 | 9319 | 994.7 | 0 | 14273 | 0.6 | 0 | 9768 | 500.7 | 0 | 9643 | 500.4 | 0 | 9743 | 501.8 |
| | | 1.00 | 0 | 13096 | 1031.7 | 0 | 18406 | 80.8 | 240 | 14091 | 500.6 | 0 | 13636 | 500.5 | 0 | 13921 | 500.8 |
| | 3 | 0.10 | 0 | 3082 | 30.8 | 0 | 3907 | 0.0 | 0 | 3108 | 500.0 | 0 | 3124 | 500.0 | 0 | 3046 | 500.1 |
| | | 0.50 | 0 | 6098 | 187.2 | 0 | 7963 | 0.2 | 0 | 6202 | 500.2 | 0 | 6121 | 500.6 | 0 | 6106 | 500.0 |
| | | 0.80 | 0 | 8799 | 1028.9 | 0 | 12861 | 0.4 | 0 | 9538 | 500.2 | 0 | 9651 | 502.0 | 0 | 9413 | 500.3 |
| | | 1.00 | 0 | 11894 | 1037.1 | 0 | 16369 | 4.4 | 0 | 12682 | 501.0 | 0 | 12177 | 500.1 | 0 | 12201 | 500.9 |
| | 4 | 0.10 | 0 | 2633 | 9.3 | 0 | 3323 | 0.0 | 0 | 2631 | 500.0 | 0 | 2630 | 500.0 | 0 | 2625 | 500.1 |
| | | 0.50 | 0 | 6540 | 333.1 | 0 | 9406 | 0.3 | 0 | 6843 | 501.2 | 0 | 6733 | 500.9 | 0 | 6783 | 500.0 |
| | | 0.80 | 0 | 9127 | 698.1 | 0 | 12390 | 0.5 | 0 | 9643 | 502.0 | 0 | 9300 | 500.9 | 0 | 9508 | 503.2 |
| | | 1.00 | 0 | 11498 | 1042.1 | 0 | 15213 | 96.6 | 0 | 12659 | 503.0 | 5 | 11987 | 500.7 | 5 | 12208 | 501.1 |
| | 5 | 0.10 | 0 | 2998 | 25.5 | 0 | 3862 | 0.0 | 0 | 2951 | 500.1 | 0 | 2944 | 500.0 | 0 | 2896 | 500.1 |
| | | 0.50 | 0 | 6967 | 610.1 | 0 | 11169 | 0.3 | 0 | 7409 | 501.4 | 0 | 7326 | 501.0 | 0 | 7399 | 500.2 |
| | | 0.80 | 0 | 9970 | 1000.6 | 0 | 14375 | 4.0 | 0 | 10697 | 501.3 | 0 | 10421 | 500.6 | 0 | 10295 | 503.2 |
| | | 1.00 | 308 | 12753 | 1001.8 | 93 | 15957 | 1037.3 | 468 | 14832 | 502.3 | 530 | 13490 | 503.9 | 615 | 13444 | 500.0 |
| | 6 | 0.10 | 0 | 3270 | 46.5 | 0 | 4386 | 0.0 | 0 | 3305 | 500.3 | 0 | 3223 | 500.1 | 0 | 3217 | 500.0 |
| | | 0.50 | 0 | 6890 | 405.4 | 0 | 10049 | 31.4 | 0 | 7350 | 500.1 | 0 | 7227 | 500.2 | 0 | 6984 | 501.4 |
| | | 0.80 | 0 | 8984 | 1030.9 | 0 | 13276 | 60.5 | 0 | 9592 | 500.3 | 0 | 9927 | 501.6 | 0 | 9851 | 500.7 |
| | | 1.00 | 185 | 12434 | 1039.0 | 120 | 15583 | 1002.3 | 120 | 13315 | 502.5 | 120 | 12932 | 500.5 | 120 | 12996 | 504.4 |
| | 7 | 0.10 | 0 | 3256 | 40.6 | 0 | 4290 | 0.0 | 0 | 3381 | 500.1 | 0 | 3272 | 500.1 | 0 | 3247 | 500.0 |
| | | 0.50 | 0 | 7664 | 513.1 | 0 | 10053 | 25.1 | 0 | 7783 | 501.2 | 0 | 7964 | 501.1 | 0 | 7725 | 501.3 |
| | | 0.80 | 0 | 11302 | 1000.1 | 0 | 15800 | 275.8 | 0 | 12010 | 501.5 | 0 | 11733 | 503.2 | 0 | 11877 | 500.3 |
| | | 1.00 | 115 | 14322 | 1001.6 | 115 | 17843 | 1012.9 | 12 | 16823 | 501.5 | 250 | 14718 | 501.4 | 250 | 15023 | 501.3 |
| | 8 | 0.10 | 0 | 3059 | 73.4 | 0 | 4606 | 0.1 | 0 | 3165 | 500.1 | 0 | 3207 | 500.3 | 0 | 3015 | 500.1 |
| | | 0.50 | 0 | 6557 | 657.0 | 0 | 9999 | 0.3 | 0 | 7000 | 500.1 | 0 | 6861 | 500.3 | 0 | 7104 | 500.0 |
| | | 0.80 | 0 | 10742 | 1045.4 | 0 | 16243 | 0.6 | 0 | 11073 | 500.5 | 0 | 10918 | 501.6 | 0 | 11105 | 500.9 |
| | | 1.00 | 0 | 13979 | 1010.7 | 0 | 19822 | 189.5 | 0 | 15110 | 500.9 | 0 | 14482 | 500.6 | 0 | 15607 | 505.8 |
| | 9 | 0.10 | 0 | 2480 | 30.9 | 0 | 3470 | 0.0 | 0 | 2490 | 500.4 | 0 | 2623 | 500.1 | 0 | 2434 | 500.2 |
| | | 0.50 | 0 | 5588 | 333.5 | 0 | 8472 | 26.0 | 0 | 5904 | 500.4 | 0 | 5991 | 500.1 | 0 | 5881 | 500.1 |
| | | 0.80 | 0 | 8510 | 784.1 | 0 | 12676 | 43.7 | 0 | 8724 | 500.3 | 0 | 8626 | 500.2 | 0 | 9067 | 500.6 |
| | | 1.00 | 0 | 11692 | 1029.5 | 0 | 16230 | 306.9 | 0 | 12509 | 501.1 | 0 | 12556 | 500.6 | 0 | 12235 | 500.7 |
| | 10 | 0.10 | 0 | 3145 | 41.8 | 0 | 4117 | 0.1 | 0 | 3062 | 500.0 | 0 | 3246 | 500.1 | 0 | 3128 | 500.2 |
| | | 0.50 | 0 | 6003 | 375.3 | 0 | 9254 | 0.3 | 0 | 6160 | 500.4 | 0 | 6073 | 500.9 | 0 | 6023 | 500.1 |
| | | 0.80 | 0 | 9655 | 676.5 | 0 | 14007 | 0.6 | 0 | 9659 | 501.0 | 0 | 10003 | 501.4 | 0 | 9840 | 500.7 |
| | | 1.00 | 0 | 13097 | 1037.9 | 46 | 16492 | 1007.1 | 0 | 13281 | 502.0 | 385 | 13164 | 500.1 | 0 | 13839 | 500.9 |
| | 11 | 0.10 | 0 | 3513 | 19.1 | 0 | 4610 | 0.0 | 0 | 3471 | 500.4 | 0 | 3513 | 500.5 | 0 | 3513 | 500.5 |
| | | 0.50 | 0 | 7540 | 310.8 | 0 | 10411 | 0.3 | 0 | 8085 | 500.5 | 0 | 7891 | 501.5 | 0 | 7953 | 500.4 |
| | | 0.80 | 0 | 9417 | 567.8 | 0 | 12731 | 0.4 | 10 | 9744 | 500.0 | 0 | 9692 | 500.2 | 0 | 9881 | 501.0 |
| | | 1.00 | 0 | 11712 | 1003.2 | 0 | 15413 | 244.1 | 0 | 12423 | 500.3 | 0 | 12330 | 501.8 | 280 | 12371 | 501.1 |
| | 12 | 0.10 | 0 | 3098 | 9.8 | 0 | 4014 | 0.0 | 0 | 3098 | 500.0 | 0 | 3096 | 500.1 | 0 | 3096 | 500.2 |
| | | 0.50 | 0 | 7209 | 453.8 | 0 | 11058 | 0.3 | 0 | 7716 | 501.8 | 0 | 7485 | 500.6 | 0 | 7357 | 500.8 |
| | | 0.80 | 0 | 10124 | 993.7 | 0 | 16037 | 125.9 | 0 | 11356 | 500.3 | 0 | 10830 | 500.3 | 5 | 11000 | 503.3 |
| | | 1.00 | 0 | 12797 | 1037.7 | 0 | 18317 | 440.7 | 440 | 14389 | 500.0 | 25 | 14220 | 501.2 | 5 | 13617 | 501.2 |
| | 13 | 0.10 | 0 | 2899 | 16.1 | 0 | 3434 | 0.0 | 0 | 2895 | 500.0 | 0 | 2858 | 500.0 | 0 | 2860 | 500.1 |
| | | 0.50 | 0 | 5966 | 203.4 | 0 | 9346 | 0.2 | 0 | 6087 | 500.1 | 0 | 5934 | 500.0 | 0 | 5894 | 500.4 |
| | | 0.80 | 0 | 8972 | 529.9 | 0 | 13215 | 0.4 | 0 | 9283 | 501.0 | 0 | 9095 | 502.4 | 0 | 9119 | 501.6 |
| | | 1.00 | 49 | 12696 | 1021.8 | 49 | 15625 | 1037.6 | 0 | 13146 | 500.7 | 49 | 12963 | 501.0 | 145 | 13196 | 501.1 |
| | 14 | 0.10 | 0 | 3549 | 44.7 | 0 | 4501 | 0.0 | 0 | 3535 | 500.1 | 0 | 3647 | 500.1 | 0 | 3451 | 500.1 |
| | | 0.50 | 0 | 7067 | 461.0 | 0 | 10534 | 0.2 | 0 | 7245 | 501.8 | 0 | 7186 | 500.7 | 0 | 7197 | 501.1 |
| | | 0.80 | 0 | 10278 | 1042.0 | 0 | 15870 | 71.4 | 750 | 10618 | 502.6 | 0 | 10630 | 500.2 | 750 | 10781 | 500.9 |
| | | 1.00 | 275 | 15289 | 1053.1 | 15 | 20098 | 1015.5 | 1248 | 17547 | 501.7 | 645 | 15412 | 502.9 | 800 | 15954 | 505.5 |

Table A.27: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 500, instance numbers (Nr.) 15–29.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 500 | 15 | 0.10 | 0 | 3641 | 73.2 | 0 | 5289 | 0.1 | 0 | 3673 | 500.3 | 0 | 3671 | 500.2 | 0 | 3647 | 500.0 |
| | | 0.50 | 0 | 6151 | 361.6 | 0 | 9280 | 0.2 | 0 | 6408 | 501.5 | 0 | 6357 | 501.4 | 0 | 6183 | 501.4 |
| | | 0.80 | 0 | 9656 | 1029.1 | 0 | 14306 | 115.3 | 70 | 10602 | 500.9 | 35 | 10365 | 500.8 | 100 | 10537 | 500.5 |
| | | 1.00 | 10 | 13722 | 1000.1 | 75 | 16961 | 1000.7 | 0 | 14463 | 501.4 | 35 | 14140 | 501.2 | 260 | 14365 | 504.8 |
| | 16 | 0.10 | 0 | 2960 | 38.8 | 0 | 4581 | 0.0 | 0 | 3030 | 500.3 | 0 | 3098 | 500.1 | 0 | 3007 | 500.0 |
| | | 0.50 | 0 | 6258 | 313.0 | 0 | 9436 | 0.2 | 0 | 6346 | 500.2 | 0 | 6377 | 500.3 | 0 | 6281 | 501.0 |
| | | 0.80 | 0 | 8758 | 1022.2 | 0 | 12919 | 0.4 | 0 | 9102 | 500.4 | 0 | 9180 | 501.9 | 0 | 9087 | 501.0 |
| | | 1.00 | 5 | 12286 | 1046.9 | 30 | 15699 | 1014.1 | 10 | 13036 | 500.2 | 10 | 12989 | 500.1 | 10 | 13006 | 500.2 |
| | 17 | 0.10 | 0 | 2215 | 11.7 | 0 | 2955 | 0.0 | 0 | 2183 | 500.0 | 0 | 2188 | 500.2 | 0 | 2160 | 500.0 |
| | | 0.50 | 0 | 5638 | 182.0 | 0 | 8191 | 20.5 | 5 | 5655 | 500.2 | 0 | 5713 | 500.0 | 0 | 5797 | 500.1 |
| | | 0.80 | 0 | 7552 | 687.8 | 0 | 11509 | 36.9 | 5 | 7914 | 500.7 | 0 | 7889 | 500.1 | 5 | 7706 | 500.1 |
| | | 1.00 | 0 | 10869 | 884.7 | 0 | 14079 | 109.4 | 23 | 11465 | 501.5 | 30 | 11242 | 500.2 | 5 | 10975 | 501.2 |
| | 18 | 0.10 | 0 | 3512 | 35.5 | 0 | 4432 | 0.0 | 0 | 3595 | 500.6 | 0 | 3594 | 500.3 | 0 | 3556 | 500.2 |
| | | 0.50 | 0 | 6626 | 301.7 | 0 | 9815 | 0.3 | 0 | 6877 | 500.5 | 0 | 6806 | 501.4 | 0 | 6728 | 501.7 |
| | | 0.80 | 0 | 9344 | 1018.7 | 0 | 15415 | 0.5 | 0 | 9999 | 501.0 | 0 | 9999 | 503.2 | 3 | 10055 | 501.1 |
| | | 1.00 | 0 | 13930 | 1002.3 | 0 | 19481 | 341.1 | 236 | 15025 | 500.1 | 514 | 14178 | 500.2 | 1045 | 14379 | 506.0 |
| | 19 | 0.10 | 0 | 3123 | 62.3 | 0 | 4511 | 0.1 | 0 | 3234 | 500.4 | 0 | 3193 | 500.3 | 0 | 3211 | 500.5 |
| | | 0.50 | 0 | 7380 | 404.2 | 0 | 9965 | 0.3 | 0 | 7612 | 500.4 | 0 | 7535 | 501.6 | 0 | 7488 | 501.6 |
| | | 0.80 | 0 | 10284 | 1002.8 | 0 | 15392 | 1.2 | 0 | 11730 | 502.4 | 0 | 10543 | 505.2 | 0 | 10611 | 503.4 |
| | | 1.00 | 0 | 15385 | 1042.9 | 0 | 20147 | 40.2 | 95 | 16302 | 506.6 | 10 | 15447 | 500.2 | 0 | 15708 | 502.5 |
| | 20 | 0.10 | 0 | 3336 | 21.9 | 0 | 4522 | 0.0 | 0 | 3284 | 500.1 | 0 | 3313 | 500.1 | 0 | 3275 | 500.0 |
| | | 0.50 | 0 | 6942 | 479.0 | 0 | 9923 | 0.3 | 0 | 7265 | 500.2 | 0 | 7205 | 500.1 | 0 | 7104 | 501.2 |
| | | 0.80 | 0 | 9157 | 832.0 | 0 | 13713 | 53.3 | 0 | 9771 | 501.4 | 0 | 9568 | 500.0 | 0 | 9538 | 502.4 |
| | | 1.00 | 140 | 13358 | 1003.5 | 140 | 18061 | 1011.9 | 95 | 14276 | 501.3 | 140 | 13641 | 503.2 | 210 | 14028 | 501.6 |
| | 21 | 0.10 | 0 | 3101 | 40.9 | 0 | 4578 | 0.0 | 0 | 3195 | 500.0 | 0 | 3110 | 500.3 | 0 | 3067 | 500.0 |
| | | 0.50 | 0 | 5902 | 406.9 | 0 | 9383 | 29.8 | 0 | 6404 | 500.2 | 20 | 6389 | 501.3 | 0 | 6633 | 500.1 |
| | | 0.80 | 0 | 8931 | 991.5 | 0 | 14010 | 136.4 | 5 | 10101 | 502.0 | 981 | 10173 | 501.6 | 0 | 9439 | 500.8 |
| | | 1.00 | 0 | 12602 | 1001.8 | 0 | 16905 | 462.5 | 852 | 13017 | 501.2 | 2450 | 13483 | 500.9 | 1394 | 13814 | 501.2 |
| | 22 | 0.10 | 0 | 2527 | 72.6 | 0 | 3519 | 0.0 | 0 | 2497 | 500.0 | 0 | 2497 | 500.0 | 0 | 2477 | 500.1 |
| | | 0.50 | 0 | 5853 | 423.4 | 0 | 9410 | 32.9 | 33 | 6291 | 500.5 | 33 | 6118 | 501.4 | 33 | 5943 | 500.1 |
| | | 0.80 | 0 | 8389 | 1012.2 | 0 | 13008 | 152.5 | 33 | 9284 | 501.2 | 53 | 9105 | 502.2 | 160 | 9408 | 500.2 |
| | | 1.00 | 0 | 11750 | 1017.4 | 0 | 16226 | 324.5 | 33 | 12819 | 500.9 | 53 | 12357 | 504.0 | 213 | 12213 | 500.4 |
| | 23 | 0.10 | 0 | 3336 | 63.6 | 0 | 4694 | 0.1 | 0 | 3481 | 500.2 | 0 | 3528 | 500.6 | 0 | 3500 | 500.6 |
| | | 0.50 | 0 | 6540 | 428.1 | 0 | 9842 | 0.3 | 0 | 6725 | 500.3 | 0 | 6731 | 500.4 | 0 | 6803 | 500.5 |
| | | 0.80 | 0 | 9106 | 1008.9 | 0 | 13632 | 1.8 | 0 | 9505 | 502.8 | 0 | 9553 | 500.6 | 0 | 9710 | 502.0 |
| | | 1.00 | 0 | 12634 | 1033.1 | 0 | 18518 | 237.2 | 1 | 14416 | 501.4 | 480 | 13006 | 505.2 | 315 | 14041 | 503.1 |
| | 24 | 0.10 | 0 | 2747 | 89.0 | 0 | 3948 | 0.0 | 0 | 2899 | 500.3 | 0 | 2922 | 500.0 | 0 | 2908 | 500.1 |
| | | 0.50 | 0 | 6230 | 410.6 | 0 | 9173 | 0.3 | 0 | 6399 | 500.2 | 0 | 6392 | 500.7 | 0 | 6356 | 500.2 |
| | | 0.80 | 0 | 8669 | 1001.9 | 0 | 12918 | 66.0 | 0 | 9133 | 500.0 | 0 | 8989 | 502.2 | 0 | 8996 | 500.3 |
| | | 1.00 | 0 | 12018 | 1006.0 | 0 | 16876 | 238.3 | 0 | 13248 | 500.1 | 0 | 12448 | 501.7 | 0 | 13204 | 503.6 |
| | 25 | 0.10 | 0 | 3631 | 35.4 | 0 | 4956 | 0.1 | 0 | 3507 | 500.3 | 0 | 3606 | 500.1 | 0 | 3596 | 500.2 |
| | | 0.50 | 0 | 7411 | 428.2 | 0 | 10623 | 0.3 | 0 | 7771 | 500.2 | 0 | 7674 | 500.8 | 0 | 7519 | 502.0 |
| | | 0.80 | 0 | 11048 | 1000.1 | 0 | 14185 | 999.4 | 0 | 11903 | 501.2 | 5 | 11496 | 500.9 | 0 | 11712 | 500.8 |
| | | 1.00 | 305 | 13828 | 1042.4 | 315 | 17860 | 1028.2 | 390 | 14827 | 500.1 | 480 | 14547 | 501.1 | 480 | 14849 | 501.1 |
| | 26 | 0.10 | 0 | 3033 | 42.0 | 0 | 4184 | 0.0 | 0 | 2972 | 500.0 | 0 | 3113 | 500.2 | 0 | 2928 | 500.2 |
| | | 0.50 | 0 | 6924 | 342.3 | 0 | 10625 | 0.5 | 0 | 7105 | 500.4 | 0 | 7184 | 500.9 | 0 | 6903 | 501.3 |
| | | 0.80 | 0 | 10142 | 1002.5 | 0 | 14119 | 103.8 | 0 | 10897 | 500.1 | 0 | 10931 | 501.0 | 0 | 11189 | 500.6 |
| | | 1.00 | 0 | 12362 | 1047.6 | 20 | 15650 | 1040.4 | 50 | 13326 | 500.3 | 405 | 13284 | 501.3 | 405 | 12926 | 504.8 |
| | 27 | 0.10 | 0 | 3503 | 31.7 | 0 | 4626 | 0.1 | 0 | 3576 | 500.0 | 0 | 3601 | 500.3 | 0 | 3531 | 500.0 |
| | | 0.50 | 0 | 6415 | 292.1 | 0 | 9717 | 0.2 | 0 | 6533 | 500.2 | 0 | 6740 | 500.9 | 0 | 6554 | 501.0 |
| | | 0.80 | 0 | 9944 | 927.5 | 0 | 14386 | 43.1 | 0 | 10540 | 501.0 | 0 | 10514 | 500.3 | 0 | 10513 | 500.9 |
| | | 1.00 | 0 | 13425 | 1003.6 | 0 | 17330 | 13.1 | 167 | 14332 | 500.3 | 0 | 13721 | 504.7 | 0 | 13177 | 502.9 |
| | 28 | 0.10 | 0 | 3032 | 35.7 | 0 | 4465 | 0.0 | 0 | 3035 | 500.1 | 0 | 3154 | 500.1 | 0 | 3050 | 500.3 |
| | | 0.50 | 0 | 6513 | 274.0 | 0 | 9716 | 0.3 | 0 | 6723 | 500.6 | 0 | 6562 | 500.0 | 0 | 6563 | 500.0 |
| | | 0.80 | 0 | 9860 | 800.9 | 0 | 14309 | 65.6 | 0 | 10478 | 500.8 | 0 | 9889 | 500.6 | 0 | 9862 | 501.0 |
| | | 1.00 | 285 | 12589 | 1001.2 | 350 | 15724 | 1034.8 | 15 | 14008 | 502.9 | 1290 | 13091 | 501.3 | 340 | 13464 | 500.1 |
| | 29 | 0.10 | 0 | 3814 | 62.8 | 0 | 5124 | 0.1 | 0 | 3997 | 500.1 | 0 | 3789 | 500.1 | 0 | 3815 | 500.5 |
| | | 0.50 | 0 | 6518 | 334.3 | 0 | 8895 | 0.2 | 0 | 6613 | 500.3 | 0 | 6560 | 501.1 | 0 | 6412 | 500.2 |
| | | 0.80 | 0 | 9009 | 1032.5 | 0 | 13180 | 0.7 | 0 | 10127 | 500.1 | 0 | 9738 | 500.6 | 0 | 9510 | 502.5 |
| | | 1.00 | 0 | 13274 | 1009.6 | 0 | 17571 | 18.8 | 0 | 13867 | 503.5 | 0 | 13503 | 501.0 | 0 | 13633 | 500.1 |

Table A.28: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 1000, instance numbers (Nr.) 0–14.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 1000 | 0 | 0.10 | 0 | 4002 | 169.5 | 0 | 6023 | 0.1 | 0 | 4066 | 500.1 | 0 | 4161 | 500.3 | 0 | 4105 | 501.3 |
| | | 0.50 | 0 | 10173 | 1001.8 | 0 | 15908 | 0.7 | 0 | 10479 | 500.9 | 0 | 11017 | 502.8 | 0 | 10435 | 504.7 |
| | | 0.80 | 165 | 16396 | 1004.9 | 0 | 23962 | 723.0 | 1120 | 17394 | 500.9 | 165 | 16757 | 501.3 | 15 | 17332 | 509.0 |
| | | 1.00 | 617 | 25140 | 1004.6 | 191 | 31778 | 1062.6 | 716 | 26359 | 503.7 | 523 | 25113 | 511.5 | 1291 | 25589 | 502.0 |
| | 1 | 0.10 | 0 | 3352 | 160.5 | 0 | 4841 | 0.1 | 0 | 3500 | 500.2 | 0 | 3564 | 500.0 | 0 | 3386 | 500.1 |
| | | 0.50 | 0 | 8718 | 804.7 | 15 | 11474 | 1042.5 | 0 | 9230 | 500.9 | 0 | 9417 | 500.2 | 0 | 9261 | 501.0 |
| | | 0.80 | 93 | 14461 | 1041.5 | 0 | 20847 | 380.6 | 0 | 14984 | 504.1 | 110 | 15082 | 503.3 | 110 | 15513 | 500.9 |
| | | 1.00 | 848 | 21414 | 1004.3 | 645 | 27135 | 1124.7 | 441 | 23130 | 502.1 | 779 | 22406 | 503.9 | 1059 | 22265 | 508.4 |
| | 2 | 0.10 | 0 | 4024 | 303.7 | 0 | 5945 | 0.1 | 0 | 4224 | 500.4 | 0 | 4266 | 501.1 | 0 | 4148 | 500.1 |
| | | 0.50 | 0 | 9526 | 1034.5 | 0 | 13551 | 115.2 | 0 | 9929 | 500.1 | 0 | 9907 | 500.7 | 0 | 9846 | 500.3 |
| | | 0.80 | 0 | 14721 | 1172.8 | 0 | 21191 | 1001.6 | 60 | 14826 | 500.0 | 105 | 14821 | 500.7 | 105 | 14778 | 502.4 |
| | | 1.00 | 240 | 19995 | 1003.3 | 0 | 25236 | 978.1 | 3476 | 21794 | 501.8 | 85 | 20292 | 502.7 | 0 | 21828 | 511.4 |
| | 3 | 0.10 | 0 | 3183 | 81.7 | 0 | 4225 | 0.0 | 0 | 3240 | 500.1 | 0 | 3296 | 500.1 | 0 | 3135 | 500.1 |
| | | 0.50 | 0 | 9127 | 1058.5 | 0 | 13362 | 83.2 | 0 | 9682 | 500.1 | 0 | 9472 | 500.6 | 0 | 9586 | 501.7 |
| | | 0.80 | 0 | 13998 | 1046.4 | 0 | 21102 | 349.0 | 0 | 14715 | 500.7 | 0 | 14091 | 500.1 | 0 | 14211 | 500.6 |
| | | 1.00 | 2 | 18835 | 1004.3 | 242 | 26054 | 1089.6 | 245 | 22159 | 505.3 | 0 | 19758 | 505.8 | 700 | 20647 | 505.3 |
| | 4 | 0.10 | 0 | 4415 | 216.7 | 0 | 5630 | 17.9 | 0 | 4616 | 500.2 | 0 | 4486 | 501.3 | 0 | 4511 | 500.2 |
| | | 0.50 | 0 | 9361 | 1044.0 | 0 | 14012 | 139.5 | 50 | 9970 | 502.4 | 50 | 9676 | 505.3 | 50 | 9913 | 502.1 |
| | | 0.80 | 0 | 13536 | 1006.5 | 0 | 19171 | 934.8 | 50 | 14336 | 501.6 | 50 | 13828 | 501.4 | 50 | 14242 | 508.7 |
| | | 1.00 | 583 | 21503 | 1016.7 | 289 | 29230 | 1195.9 | 1358 | 23893 | 511.8 | 607 | 22448 | 500.4 | 824 | 23125 | 510.0 |
| | 5 | 0.10 | 0 | 4875 | 198.9 | 0 | 6841 | 19.4 | 0 | 5189 | 500.2 | 0 | 5065 | 500.1 | 0 | 4952 | 500.2 |
| | | 0.50 | 0 | 9889 | 1000.8 | 0 | 15660 | 256.0 | 0 | 10776 | 503.5 | 0 | 10623 | 500.1 | 0 | 10714 | 500.2 |
| | | 0.80 | 0 | 15818 | 1039.9 | 0 | 22306 | 405.0 | 370 | 17524 | 501.8 | 0 | 16406 | 501.8 | 0 | 16274 | 505.3 |
| | | 1.00 | 586 | 21117 | 1010.3 | 70 | 28299 | 1001.5 | 1067 | 21961 | 502.9 | 361 | 21982 | 502.7 | 470 | 22525 | 505.6 |
| | 6 | 0.10 | 0 | 4957 | 223.4 | 0 | 6789 | 0.1 | 0 | 4900 | 500.7 | 0 | 5032 | 500.9 | 0 | 4947 | 500.4 |
| | | 0.50 | 0 | 9438 | 1075.6 | 0 | 13399 | 0.7 | 0 | 10120 | 500.3 | 0 | 9860 | 500.0 | 0 | 9940 | 502.6 |
| | | 0.80 | 209 | 15055 | 1071.8 | 209 | 19397 | 1135.0 | 0 | 15613 | 505.1 | 209 | 15360 | 502.6 | 0 | 15664 | 508.6 |
| | | 1.00 | 840 | 21214 | 1012.6 | 312 | 27707 | 1160.3 | 117 | 22356 | 501.1 | 870 | 21125 | 502.5 | 1589 | 23091 | 501.9 |
| | 7 | 0.10 | 0 | 2747 | 251.9 | 0 | 4115 | 0.1 | 0 | 2837 | 500.3 | 0 | 3086 | 500.0 | 0 | 2825 | 500.2 |
| | | 0.50 | 0 | 7407 | 726.3 | 0 | 11289 | 0.5 | 0 | 7640 | 501.4 | 0 | 7644 | 501.5 | 0 | 7782 | 501.7 |
| | | 0.80 | 0 | 14343 | 1055.3 | 15 | 19408 | 1083.0 | 0 | 15271 | 505.8 | 17 | 15605 | 501.2 | 17 | 14608 | 501.3 |
| | | 1.00 | 606 | 21510 | 1001.2 | 300 | 27212 | 1068.0 | 1250 | 22715 | 502.7 | 1152 | 22695 | 503.0 | 1576 | 22714 | 501.5 |
| | 8 | 0.10 | 0 | 3685 | 155.6 | 0 | 5300 | 0.1 | 0 | 4043 | 500.4 | 0 | 3760 | 500.2 | 0 | 3676 | 500.8 |
| | | 0.50 | 0 | 7209 | 1023.4 | 0 | 12741 | 0.5 | 0 | 7482 | 501.1 | 0 | 7924 | 501.0 | 0 | 7939 | 501.4 |
| | | 0.80 | 241 | 15233 | 1001.5 | 427 | 20567 | 1003.9 | 95 | 17283 | 507.3 | 168 | 15584 | 503.8 | 635 | 16356 | 506.6 |
| | | 1.00 | 664 | 21941 | 1010.3 | 293 | 25790 | 1152.5 | 1639 | 22590 | 511.2 | 787 | 21958 | 500.2 | 872 | 21856 | 510.2 |
| | 9 | 0.10 | 0 | 3490 | 202.5 | 0 | 4687 | 0.1 | 0 | 3566 | 500.3 | 0 | 3604 | 500.1 | 0 | 3537 | 500.2 |
| | | 0.50 | 0 | 8437 | 1003.9 | 0 | 12546 | 0.6 | 0 | 8920 | 500.3 | 0 | 8748 | 500.6 | 0 | 9253 | 500.1 |
| | | 0.80 | 0 | 14113 | 1046.4 | 54 | 19688 | 1103.9 | 0 | 14505 | 502.6 | 140 | 14227 | 502.2 | 140 | 14295 | 500.1 |
| | | 1.00 | 445 | 19812 | 1004.5 | 576 | 26125 | 1084.9 | 775 | 20566 | 503.5 | 635 | 19365 | 500.1 | 372 | 20820 | 508.8 |
| | 10 | 0.10 | 0 | 3114 | 566.1 | 0 | 4256 | 0.1 | 0 | 3383 | 500.0 | 0 | 3272 | 500.0 | 0 | 3200 | 500.4 |
| | | 0.50 | 0 | 8881 | 1104.5 | 0 | 12850 | 122.2 | 0 | 9197 | 501.0 | 0 | 9021 | 500.8 | 0 | 9135 | 500.9 |
| | | 0.80 | 195 | 14986 | 1077.6 | 180 | 21157 | 1160.4 | 240 | 16103 | 500.6 | 205 | 15345 | 509.4 | 485 | 15387 | 500.9 |
| | | 1.00 | 3571 | 20931 | 1012.3 | 964 | 29324 | 1180.4 | 270 | 21356 | 500.1 | 3494 | 21739 | 502.0 | 4061 | 21781 | 511.9 |
| | 11 | 0.10 | 0 | 3910 | 226.5 | 0 | 5517 | 0.1 | 0 | 4026 | 500.3 | 0 | 4152 | 500.8 | 0 | 4099 | 500.5 |
| | | 0.50 | 0 | 8113 | 1051.6 | 0 | 12241 | 83.1 | 0 | 9026 | 501.6 | 0 | 8501 | 501.7 | 0 | 8443 | 500.0 |
| | | 0.80 | 181 | 13879 | 1025.7 | 181 | 19710 | 1051.7 | 0 | 14538 | 501.4 | 181 | 14248 | 500.1 | 102 | 13947 | 503.9 |
| | | 1.00 | 895 | 21569 | 1004.4 | 444 | 28019 | 1010.0 | 237 | 21877 | 500.4 | 769 | 22065 | 501.4 | 854 | 22061 | 500.8 |
| | 12 | 0.10 | 0 | 3634 | 189.6 | 0 | 5372 | 0.1 | 0 | 3706 | 500.9 | 0 | 3863 | 500.4 | 0 | 3660 | 500.8 |
| | | 0.50 | 0 | 9669 | 1075.4 | 0 | 14584 | 154.0 | 0 | 10548 | 506.8 | 0 | 10792 | 507.1 | 0 | 10960 | 504.7 |
| | | 0.80 | 3 | 15518 | 1073.6 | 13 | 20770 | 1172.7 | 68 | 16812 | 500.7 | 78 | 15829 | 508.9 | 78 | 15979 | 507.0 |
| | | 1.00 | 756 | 21013 | 1010.0 | 420 | 26258 | 1171.0 | 1534 | 23337 | 506.8 | 796 | 22100 | 502.3 | 2650 | 23860 | 506.0 |
| | 13 | 0.10 | 0 | 4415 | 217.9 | 0 | 5895 | 0.1 | 0 | 4488 | 500.0 | 0 | 4453 | 500.0 | 0 | 4346 | 500.6 |
| | | 0.50 | 0 | 7777 | 1012.5 | 0 | 12976 | 0.6 | 0 | 8543 | 503.2 | 0 | 8002 | 504.5 | 0 | 8241 | 502.6 |
| | | 0.80 | 0 | 11924 | 1074.7 | 0 | 17614 | 217.1 | 0 | 14021 | 500.7 | 0 | 12399 | 501.3 | 0 | 12553 | 508.2 |
| | | 1.00 | 492 | 21687 | 1002.6 | 234 | 30050 | 1222.0 | 667 | 21642 | 500.6 | 927 | 21426 | 503.9 | 1349 | 23440 | 501.5 |
| | 14 | 0.10 | 0 | 3675 | 105.1 | 0 | 4816 | 0.1 | 0 | 3842 | 500.0 | 0 | 3765 | 500.3 | 0 | 3729 | 500.0 |
| | | 0.50 | 0 | 7609 | 990.7 | 0 | 12272 | 0.6 | 0 | 7609 | 503.1 | 0 | 7644 | 501.3 | 0 | 8179 | 504.0 |
| | | 0.80 | 0 | 11535 | 1030.9 | 0 | 17609 | 188.4 | 0 | 12810 | 500.7 | 0 | 11962 | 500.1 | 0 | 12439 | 507.5 |
| | | 1.00 | 300 | 20934 | 1008.4 | 171 | 26818 | 1048.9 | 866 | 23107 | 513.8 | 462 | 20678 | 501.3 | 601 | 22796 | 503.5 |

Table A.29: Performance of VND-O, VND-S, GRASP-O, MA-O, and VNS-O for instance size (Sz.) 1000, instance numbers (Nr.) 15–29.

| | | | VND-O | | | VND-S | | | GRASP-O | | | MA-O | | | VNS-O | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** | $C_a$ | $C_u$ | **t[s]** |
| 1000 | 15 | 0.10 | 0 | 4379 | 208.0 | 0 | 5923 | 0.1 | 0 | 4732 | 500.1 | 0 | 4523 | 501.1 | 0 | 4445 | 500.8 |
| | | 0.50 | 0 | 9442 | 1024.9 | 0 | 13953 | 0.6 | 0 | 9975 | 501.8 | 0 | 10023 | 500.0 | 0 | 10017 | 500.5 |
| | | 0.80 | 25 | 15244 | 1057.0 | 25 | 19378 | 1075.9 | 202 | 18378 | 500.8 | 415 | 15068 | 500.4 | 407 | 15623 | 505.5 |
| | | 1.00 | 271 | 21226 | 1019.0 | 129 | 26157 | 1123.0 | 75 | 22238 | 509.9 | 356 | 21376 | 506.0 | 341 | 22488 | 502.2 |
| | 16 | 0.10 | 0 | 3407 | 346.0 | 0 | 5535 | 0.1 | 0 | 3722 | 500.1 | 0 | 3722 | 500.2 | 0 | 3445 | 500.3 |
| | | 0.50 | 0 | 8562 | 1039.1 | 0 | 13259 | 0.5 | 0 | 9132 | 501.5 | 0 | 8977 | 500.2 | 0 | 8794 | 501.7 |
| | | 0.80 | 0 | 12980 | 1077.0 | 0 | 17842 | 186.6 | 0 | 13968 | 503.2 | 0 | 13296 | 501.0 | 0 | 13738 | 500.6 |
| | | 1.00 | 15 | 19855 | 1004.9 | 92 | 25731 | 1229.2 | 89 | 21632 | 500.4 | 0 | 19838 | 505.7 | 0 | 21329 | 505.6 |
| | 17 | 0.10 | 0 | 3277 | 140.2 | 0 | 4789 | 0.1 | 0 | 3335 | 501.0 | 0 | 3360 | 500.1 | 0 | 3318 | 500.3 |
| | | 0.50 | 0 | 9638 | 1000.8 | 0 | 14267 | 95.8 | 1 | 10148 | 503.0 | 1 | 9848 | 500.1 | 0 | 10037 | 500.1 |
| | | 0.80 | 131 | 16398 | 1040.0 | 130 | 20956 | 1139.3 | 202 | 17271 | 504.7 | 247 | 16226 | 504.6 | 202 | 16822 | 504.9 |
| | | 1.00 | 420 | 20852 | 1011.4 | 173 | 26541 | 1003.7 | 2079 | 22834 | 509.7 | 323 | 20633 | 503.2 | 513 | 21909 | 502.2 |
| | 18 | 0.10 | 0 | 3609 | 163.3 | 0 | 4996 | 0.1 | 0 | 3797 | 500.1 | 0 | 3808 | 501.1 | 0 | 3694 | 500.0 |
| | | 0.50 | 0 | 8722 | 967.1 | 0 | 14021 | 0.7 | 0 | 9226 | 501.4 | 0 | 8974 | 505.8 | 0 | 9003 | 502.0 |
| | | 0.80 | 0 | 13556 | 1107.8 | 0 | 20666 | 1.9 | 0 | 14504 | 503.2 | 0 | 14620 | 501.1 | 0 | 14519 | 502.4 |
| | | 1.00 | 0 | 19996 | 1008.2 | 0 | 27385 | 833.5 | 330 | 22020 | 503.6 | 145 | 20219 | 505.1 | 1092 | 22447 | 509.3 |
| | 19 | 0.10 | 0 | 3441 | 209.8 | 0 | 5071 | 0.1 | 0 | 3746 | 500.1 | 0 | 3548 | 500.2 | 0 | 3397 | 500.8 |
| | | 0.50 | 0 | 8623 | 1006.1 | 0 | 13490 | 196.9 | 0 | 9312 | 500.3 | 11 | 9274 | 500.2 | 0 | 9393 | 504.9 |
| | | 0.80 | 0 | 14182 | 1042.0 | 0 | 19998 | 396.4 | 0 | 14865 | 502.6 | 11 | 14463 | 502.1 | 471 | 15046 | 500.9 |
| | | 1.00 | 496 | 20822 | 1013.5 | 10 | 26340 | 1022.3 | 669 | 22114 | 500.4 | 506 | 20657 | 509.8 | 657 | 22706 | 508.2 |
| | 20 | 0.10 | 0 | 5129 | 234.3 | 0 | 6581 | 0.2 | 0 | 5239 | 500.1 | 0 | 5344 | 500.1 | 0 | 5278 | 500.1 |
| | | 0.50 | 0 | 8886 | 1041.4 | 0 | 13071 | 0.6 | 0 | 9170 | 500.9 | 0 | 9224 | 503.1 | 0 | 9297 | 500.6 |
| | | 0.80 | 0 | 14090 | 1020.5 | 0 | 20926 | 22.8 | 0 | 15476 | 505.4 | 0 | 15009 | 503.1 | 10 | 15026 | 502.9 |
| | | 1.00 | 350 | 20310 | 1013.2 | 70 | 27316 | 1104.1 | 113 | 22324 | 503.4 | 1306 | 21099 | 501.8 | 640 | 21640 | 512.0 |
| | 21 | 0.10 | 0 | 4076 | 286.3 | 0 | 5436 | 0.2 | 0 | 4215 | 501.5 | 0 | 4230 | 500.3 | 0 | 4140 | 501.4 |
| | | 0.50 | 0 | 8891 | 1002.1 | 0 | 13172 | 0.8 | 0 | 8998 | 500.8 | 0 | 9178 | 503.4 | 0 | 9299 | 500.8 |
| | | 0.80 | 89 | 13080 | 1009.4 | 89 | 17547 | 1001.2 | 0 | 14081 | 503.9 | 89 | 13284 | 500.4 | 89 | 13539 | 501.6 |
| | | 1.00 | 490 | 20092 | 1014.8 | 550 | 25878 | 1016.7 | 499 | 21015 | 505.4 | 642 | 20671 | 501.3 | 1159 | 20412 | 503.2 |
| | 22 | 0.10 | 0 | 3846 | 106.9 | 0 | 5040 | 0.1 | 0 | 3896 | 500.1 | 0 | 3830 | 501.0 | 0 | 3811 | 500.4 |
| | | 0.50 | 0 | 7637 | 1002.8 | 0 | 12930 | 115.0 | 0 | 8498 | 501.2 | 0 | 7736 | 505.2 | 1366 | 8050 | 503.7 |
| | | 0.80 | 0 | 13609 | 1167.7 | 0 | 19922 | 428.6 | 0 | 14097 | 503.2 | 0 | 13833 | 504.9 | 1366 | 14049 | 502.4 |
| | | 1.00 | 2293 | 20607 | 1005.1 | 758 | 28152 | 1012.5 | 860 | 20302 | 510.0 | 2268 | 20076 | 501.0 | 2138 | 21291 | 507.7 |
| | 23 | 0.10 | 0 | 4208 | 106.8 | 0 | 5379 | 13.9 | 0 | 4293 | 500.5 | 0 | 4224 | 500.0 | 0 | 4251 | 500.1 |
| | | 0.50 | 0 | 9397 | 1046.3 | 0 | 14073 | 94.4 | 0 | 9806 | 501.6 | 0 | 9689 | 501.8 | 0 | 9633 | 500.9 |
| | | 0.80 | 20 | 12808 | 1119.4 | 20 | 17995 | 1128.8 | 185 | 13746 | 502.2 | 20 | 13188 | 501.2 | 0 | 13434 | 504.3 |
| | | 1.00 | 774 | 21556 | 1001.2 | 540 | 27914 | 1091.1 | 1795 | 23157 | 506.0 | 950 | 21029 | 510.1 | 1418 | 22339 | 512.2 |
| | 24 | 0.10 | 0 | 3384 | 68.5 | 0 | 4662 | 0.1 | 0 | 3517 | 500.1 | 0 | 3459 | 500.8 | 0 | 3422 | 500.5 |
| | | 0.50 | 2 | 9596 | 1075.2 | 2 | 12818 | 1081.3 | 2 | 10189 | 500.8 | 2 | 10178 | 504.7 | 2 | 9859 | 503.7 |
| | | 0.80 | 2 | 16223 | 1107.3 | 2 | 20791 | 1045.3 | 102 | 16224 | 502.8 | 1028 | 16382 | 506.1 | 911 | 16547 | 508.1 |
| | | 1.00 | 971 | 20759 | 1013.9 | 528 | 24957 | 1068.8 | 551 | 22007 | 503.5 | 981 | 21229 | 503.8 | 1034 | 21980 | 512.6 |
| | 25 | 0.10 | 0 | 4114 | 379.7 | 0 | 6284 | 0.2 | 0 | 4250 | 500.1 | 0 | 4285 | 501.9 | 0 | 4209 | 500.8 |
| | | 0.50 | 0 | 8360 | 1112.3 | 0 | 13407 | 0.7 | 0 | 8705 | 503.9 | 0 | 8889 | 506.9 | 0 | 8783 | 501.5 |
| | | 0.80 | 0 | 14777 | 1106.2 | 0 | 21769 | 840.6 | 0 | 15925 | 505.6 | 0 | 15506 | 506.2 | 0 | 16295 | 501.2 |
| | | 1.00 | 5 | 20428 | 1020.6 | 338 | 29771 | 1081.9 | 1147 | 23330 | 500.6 | 270 | 20241 | 504.1 | 691 | 22011 | 501.6 |
| | 26 | 0.10 | 0 | 4570 | 166.6 | 0 | 6407 | 0.1 | 0 | 4766 | 500.0 | 0 | 4699 | 500.9 | 0 | 4807 | 500.1 |
| | | 0.50 | 0 | 9124 | 1029.4 | 0 | 13540 | 88.5 | 0 | 9672 | 502.0 | 0 | 9702 | 501.0 | 0 | 9394 | 505.1 |
| | | 0.80 | 0 | 15371 | 1137.7 | 0 | 20996 | 369.8 | 30 | 15910 | 502.6 | 0 | 15967 | 507.6 | 0 | 15970 | 502.2 |
| | | 1.00 | 308 | 21298 | 1000.9 | 194 | 26020 | 1033.0 | 1552 | 22214 | 509.1 | 308 | 21712 | 514.3 | 242 | 22463 | 505.1 |
| | 27 | 0.10 | 0 | 2977 | 231.3 | 0 | 4175 | 0.1 | 0 | 3216 | 500.4 | 0 | 3141 | 500.2 | 0 | 3023 | 500.0 |
| | | 0.50 | 0 | 7174 | 870.6 | 0 | 10707 | 0.4 | 0 | 7425 | 502.5 | 0 | 7298 | 500.6 | 0 | 7257 | 500.1 |
| | | 0.80 | 0 | 12510 | 1112.6 | 0 | 18046 | 1.2 | 205 | 14122 | 501.6 | 0 | 13030 | 502.6 | 0 | 13011 | 501.3 |
| | | 1.00 | 0 | 19228 | 1013.2 | 28 | 25302 | 1016.2 | 370 | 21183 | 502.4 | 240 | 19644 | 501.9 | 30 | 21289 | 507.5 |
| | 28 | 0.10 | 0 | 4536 | 260.1 | 0 | 6309 | 0.1 | 0 | 4679 | 500.1 | 0 | 4846 | 500.1 | 0 | 4666 | 500.9 |
| | | 0.50 | 10 | 9881 | 1070.3 | 0 | 13777 | 204.2 | 10 | 10552 | 500.2 | 0 | 10323 | 504.7 | 0 | 10287 | 500.1 |
| | | 0.80 | 10 | 14949 | 1023.9 | 135 | 19721 | 1079.9 | 10 | 15670 | 506.9 | 150 | 15129 | 500.6 | 10 | 15469 | 501.6 |
| | | 1.00 | 1569 | 23433 | 1013.9 | 451 | 29311 | 1006.6 | 125 | 25106 | 505.3 | 1753 | 23220 | 502.9 | 1882 | 25455 | 504.3 |
| | 29 | 0.10 | 0 | 4111 | 322.0 | 0 | 5978 | 0.1 | 0 | 4303 | 500.2 | 0 | 4417 | 500.4 | 0 | 4257 | 500.0 |
| | | 0.50 | 0 | 8080 | 1043.7 | 0 | 13549 | 4.5 | 0 | 8428 | 500.2 | 0 | 8648 | 502.3 | 0 | 8640 | 500.1 |
| | | 0.80 | 0 | 13828 | 1047.6 | 0 | 19509 | 896.3 | 0 | 17023 | 501.4 | 0 | 14389 | 500.0 | 0 | 14809 | 503.0 |
| | | 1.00 | 367 | 19753 | 1153.6 | 85 | 26128 | 1103.3 | 1542 | 21627 | 506.3 | 160 | 20018 | 502.5 | 615 | 21064 | 505.1 |

Table A.30: Performance of CP-O and CP-S for instance size (Sz.) 20, instance numbers (Nr.) 0–14.

| | | | CP-O | | | | CP-S | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** |
| 20 | 0 | 0.10 | 0 | 469 | 0.2 | 1 | 0 | 595 | 0.0 | 0 |
| | | 0.50 | 0 | 778 | 6.1 | 1 | 0 | 1042 | 0.2 | 0 |
| | | 0.80 | 0 | 1223 | 9991.3 | 0 | 0 | 1238 | 0.5 | 0 |
| | | 1.00 | 0 | 1288 | 9995.6 | 0 | 0 | 1382 | 0.8 | 0 |
| | 1 | 0.10 | 0 | 351 | 0.0 | 1 | 0 | 377 | 0.0 | 0 |
| | | 0.50 | 0 | 668 | 0.5 | 1 | 0 | 791 | 0.1 | 0 |
| | | 0.80 | 0 | 870 | 10000.3 | 0 | 0 | 1007 | 0.2 | 0 |
| | | 1.00 | 0 | 921 | 9995.6 | 0 | 0 | 1095 | 0.4 | 0 |
| | 2 | 0.10 | 0 | 484 | 2146.7 | 1 | 0 | 511 | 0.0 | 0 |
| | | 0.50 | 0 | 900 | 10000.5 | 0 | 0 | 990 | 0.1 | 0 |
| | | 0.80 | 0 | 983 | 9995.5 | 0 | 0 | 1298 | 0.4 | 0 |
| | | 1.00 | 0 | 1538 | 10000.4 | 0 | 0 | 1540 | 0.6 | 0 |
| | 3 | 0.10 | 0 | 467 | 31.6 | 1 | 0 | 739 | 0.0 | 0 |
| | | 0.50 | 0 | 1006 | 9994.5 | 0 | 0 | 1147 | 0.1 | 0 |
| | | 0.80 | 0 | 1175 | 9998.3 | 0 | 0 | 1305 | 0.2 | 0 |
| | | 1.00 | 0 | 1438 | 10000.5 | 0 | 0 | 1578 | 0.4 | 0 |
| | 4 | 0.10 | 0 | 497 | 3.7 | 1 | 0 | 543 | 0.0 | 0 |
| | | 0.50 | 0 | 981 | 10000.5 | 0 | 0 | 1048 | 0.1 | 0 |
| | | 0.80 | 0 | 1037 | 10000.5 | 0 | 0 | 1352 | 0.2 | 0 |
| | | 1.00 | 0 | 1295 | 10000.2 | 0 | 0 | 1488 | 0.4 | 0 |
| | 5 | 0.10 | 0 | 401 | 0.7 | 1 | 0 | 464 | 0.0 | 0 |
| | | 0.50 | 0 | 879 | 9970.9 | 0 | 0 | 978 | 0.1 | 0 |
| | | 0.80 | 0 | 990 | 10000.3 | 0 | 0 | 1166 | 0.3 | 0 |
| | | 1.00 | 0 | 1081 | 9997.3 | 0 | 0 | 1222 | 0.4 | 0 |
| | 6 | 0.10 | 0 | 374 | 1.7 | 1 | 0 | 493 | 0.0 | 0 |
| | | 0.50 | 0 | 841 | 9983.3 | 0 | 0 | 1143 | 0.1 | 0 |
| | | 0.80 | 0 | 1215 | 9979.4 | 0 | 0 | 1406 | 0.2 | 0 |
| | | 1.00 | 0 | 1422 | 10000.3 | 0 | 0 | 1446 | 0.4 | 0 |
| | 7 | 0.10 | 0 | 521 | 9.1 | 1 | 0 | 819 | 0.0 | 0 |
| | | 0.50 | 0 | 1034 | 9997.0 | 0 | 0 | 1289 | 0.2 | 0 |
| | | 0.80 | 0 | 1216 | 9994.3 | 0 | 0 | 1460 | 0.4 | 0 |
| | | 1.00 | 0 | 1503 | 9995.5 | 0 | 0 | 1590 | 0.6 | 0 |
| | 8 | 0.10 | 0 | 347 | 309.9 | 1 | 0 | 662 | 0.0 | 0 |
| | | 0.50 | 0 | 849 | 9995.5 | 0 | 0 | 1092 | 0.2 | 0 |
| | | 0.80 | 0 | 1122 | 9978.6 | 0 | 0 | 1171 | 0.5 | 0 |
| | | 1.00 | 0 | 1273 | 9999.2 | 0 | 0 | 1442 | 0.6 | 0 |
| | 9 | 0.10 | 0 | 643 | 37.2 | 1 | 0 | 754 | 0.0 | 0 |
| | | 0.50 | 0 | 1089 | 9995.6 | 0 | 0 | 1121 | 0.2 | 0 |
| | | 0.80 | 0 | 1142 | 10000.4 | 0 | 0 | 1230 | 0.4 | 0 |
| | | 1.00 | 0 | 1271 | 9995.5 | 0 | 0 | 1549 | 0.5 | 0 |
| | 10 | 0.10 | 0 | 559 | 44.3 | 1 | 0 | 660 | 0.0 | 0 |
| | | 0.50 | 0 | 979 | 10000.4 | 0 | 0 | 1093 | 0.1 | 0 |
| | | 0.80 | 0 | 1268 | 10000.5 | 0 | 0 | 1374 | 0.3 | 0 |
| | | 1.00 | 0 | 0 | 9994.5 | 0 | 0 | 0 | 9961.6 | 0 |
| | 11 | 0.10 | 0 | 217 | 0.5 | 1 | 0 | 308 | 0.0 | 0 |
| | | 0.50 | 0 | 711 | 9981.7 | 0 | 0 | 933 | 0.1 | 0 |
| | | 0.80 | 0 | 890 | 9993.7 | 0 | 0 | 1077 | 0.3 | 0 |
| | | 1.00 | 0 | 973 | 9995.5 | 0 | 0 | 1077 | 0.4 | 0 |
| | 12 | 0.10 | 0 | 499 | 4.0 | 1 | 0 | 573 | 0.0 | 0 |
| | | 0.50 | 0 | 882 | 10000.5 | 0 | 0 | 1153 | 0.2 | 0 |
| | | 0.80 | 0 | 1257 | 9995.5 | 0 | 0 | 1440 | 0.4 | 0 |
| | | 1.00 | 0 | 1382 | 10000.5 | 0 | 0 | 1586 | 0.6 | 0 |
| | 13 | 0.10 | 0 | 476 | 2733.9 | 1 | 0 | 689 | 0.0 | 0 |
| | | 0.50 | 0 | 800 | 9995.5 | 0 | 0 | 880 | 0.4 | 0 |
| | | 0.80 | 0 | 992 | 10000.5 | 0 | 0 | 1310 | 0.8 | 0 |
| | | 1.00 | 0 | 1315 | 10000.5 | 0 | 0 | 1340 | 1.2 | 0 |
| | 14 | 0.10 | 0 | 535 | 2.5 | 1 | 0 | 636 | 0.0 | 0 |
| | | 0.50 | 0 | 947 | 9997.0 | 0 | 0 | 1195 | 0.1 | 0 |
| | | 0.80 | 0 | 1219 | 10000.4 | 0 | 0 | 1310 | 0.4 | 0 |
| | | 1.00 | 0 | 0 | 9989.3 | 0 | 0 | 0 | 10000.4 | 0 |

Table A.31: Performance of CP-O and CP-S for instance size (Sz.) 20, instance numbers (Nr.) 15–29.

| | | | CP-O | | | | CP-S | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** |
| 20 | 15 | 0.10 | 0 | 318 | 3.0 | 1 | 0 | 474 | 0.0 | 0 |
| | | 0.50 | 0 | 525 | 2.8 | 1 | 0 | 949 | 0.2 | 0 |
| | | 0.80 | 0 | 880 | 10000.5 | 0 | 0 | 1138 | 0.5 | 0 |
| | | 1.00 | 0 | 0 | 10000.5 | 0 | 0 | 0 | 9997.9 | 0 |
| | 16 | 0.10 | 0 | 611 | 9.7 | 1 | 0 | 931 | 0.0 | 0 |
| | | 0.50 | 0 | 1047 | 9982.7 | 0 | 0 | 1373 | 0.2 | 0 |
| | | 0.80 | 0 | 1328 | 9995.6 | 0 | 0 | 1584 | 0.4 | 0 |
| | | 1.00 | 0 | 1482 | 9995.6 | 0 | 0 | 1584 | 0.5 | 0 |
| | 17 | 0.10 | 0 | 453 | 0.1 | 1 | 0 | 563 | 0.0 | 0 |
| | | 0.50 | 0 | 923 | 2.7 | 1 | 0 | 1061 | 0.1 | 0 |
| | | 0.80 | 0 | 1323 | 10000.4 | 0 | 0 | 1447 | 0.3 | 0 |
| | | 1.00 | 0 | 1489 | 10000.4 | 0 | 0 | 1589 | 0.5 | 0 |
| | 18 | 0.10 | 0 | 439 | 1.3 | 1 | 0 | 528 | 0.0 | 0 |
| | | 0.50 | 0 | 985 | 9994.7 | 0 | 0 | 1112 | 0.1 | 0 |
| | | 0.80 | 0 | 1245 | 10000.5 | 0 | 0 | 1271 | 95.1 | 0 |
| | | 1.00 | 0 | 0 | 9995.5 | 0 | 0 | 0 | 9893.5 | 0 |
| | 19 | 0.10 | 0 | 579 | 2575.0 | 1 | 0 | 662 | 0.0 | 0 |
| | | 0.50 | 0 | 987 | 10000.5 | 0 | 0 | 1416 | 0.2 | 0 |
| | | 0.80 | 0 | 1353 | 9993.8 | 0 | 0 | 1500 | 0.5 | 0 |
| | | 1.00 | 0 | 0 | 9995.5 | 0 | 0 | 0 | 9981.1 | 0 |
| | 20 | 0.10 | 0 | 371 | 0.2 | 1 | 0 | 434 | 0.0 | 0 |
| | | 0.50 | 0 | 958 | 9995.6 | 0 | 0 | 1172 | 0.1 | 0 |
| | | 0.80 | 0 | 1153 | 9949.1 | 0 | 0 | 1322 | 0.3 | 0 |
| | | 1.00 | 0 | 1366 | 9995.6 | 0 | 0 | 1391 | 0.4 | 0 |
| | 21 | 0.10 | 0 | 421 | 1.9 | 1 | 0 | 484 | 0.0 | 0 |
| | | 0.50 | 0 | 697 | 10000.4 | 0 | 0 | 898 | 0.2 | 0 |
| | | 0.80 | 0 | 909 | 9979.4 | 0 | 0 | 1103 | 0.5 | 0 |
| | | 1.00 | 0 | 1211 | 9996.3 | 0 | 0 | 0 | 9989.3 | 0 |
| | 22 | 0.10 | 0 | 524 | 0.1 | 1 | 0 | 647 | 0.0 | 0 |
| | | 0.50 | 0 | 669 | 0.3 | 1 | 0 | 748 | 0.1 | 0 |
| | | 0.80 | 0 | 802 | 2.0 | 1 | 0 | 1120 | 0.3 | 0 |
| | | 1.00 | 0 | 887 | 10000.4 | 0 | 0 | 1127 | 0.4 | 0 |
| | 23 | 0.10 | 0 | 499 | 2102.5 | 1 | 0 | 591 | 0.0 | 0 |
| | | 0.50 | 0 | 936 | 9995.5 | 0 | 0 | 1180 | 0.2 | 0 |
| | | 0.80 | 0 | 1066 | 9995.1 | 0 | 0 | 1461 | 0.4 | 0 |
| | | 1.00 | 0 | 1361 | 10000.5 | 0 | 0 | 1559 | 0.7 | 0 |
| | 24 | 0.10 | 0 | 620 | 14.3 | 1 | 0 | 716 | 0.0 | 0 |
| | | 0.50 | 0 | 930 | 9997.9 | 0 | 0 | 1075 | 0.2 | 0 |
| | | 0.80 | 0 | 1162 | 9995.0 | 0 | 0 | 1184 | 0.5 | 0 |
| | | 1.00 | 0 | 1247 | 10000.0 | 0 | 0 | 1307 | 0.7 | 0 |
| | 25 | 0.10 | 0 | 432 | 1.0 | 1 | 0 | 525 | 0.0 | 0 |
| | | 0.50 | 0 | 894 | 9995.5 | 0 | 0 | 955 | 0.1 | 0 |
| | | 0.80 | 0 | 1148 | 9999.0 | 0 | 0 | 1249 | 0.4 | 0 |
| | | 1.00 | 0 | 1357 | 9994.5 | 0 | 0 | 1481 | 9525.2 | 0 |
| | 26 | 0.10 | 0 | 369 | 9.1 | 1 | 0 | 559 | 0.0 | 0 |
| | | 0.50 | 0 | 1022 | 9995.5 | 0 | 0 | 1022 | 0.1 | 0 |
| | | 0.80 | 0 | 962 | 9992.6 | 0 | 0 | 1054 | 0.3 | 0 |
| | | 1.00 | 0 | 0 | 9995.4 | 0 | 0 | 0 | 9995.5 | 0 |
| | 27 | 0.10 | 0 | 607 | 1.9 | 1 | 0 | 687 | 0.0 | 0 |
| | | 0.50 | 0 | 807 | 0.8 | 1 | 0 | 1086 | 0.2 | 0 |
| | | 0.80 | 0 | 1115 | 9994.3 | 0 | 0 | 1446 | 0.4 | 0 |
| | | 1.00 | 0 | 0 | 9995.2 | 0 | 0 | 0 | 9975.3 | 0 |
| | 28 | 0.10 | 0 | 531 | 12.0 | 1 | 0 | 617 | 0.0 | 0 |
| | | 0.50 | 0 | 1128 | 9995.4 | 0 | 0 | 1247 | 0.1 | 0 |
| | | 0.80 | 0 | 1286 | 9995.4 | 0 | 0 | 1502 | 0.4 | 0 |
| | | 1.00 | 0 | 0 | 9999.8 | 0 | 0 | 0 | 10000.4 | 0 |
| | 29 | 0.10 | 0 | 368 | 10.3 | 1 | 0 | 460 | 0.0 | 0 |
| | | 0.50 | 0 | 681 | 9995.3 | 0 | 0 | 1029 | 0.1 | 0 |
| | | 0.80 | 0 | 1135 | 9995.5 | 0 | 0 | 1258 | 0.4 | 0 |
| | | 1.00 | 0 | 1354 | 9995.5 | 0 | 0 | 1418 | 0.6 | 0 |

Table A.32: Performance of CP-O and CP-S for instance size (Sz.) 30, instance numbers (Nr.) 0–14.

| | | | CP-O | | | | CP-S | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** |
| 30 | 0 | 0.10 | 0 | 606 | 368.4 | 1 | 0 | 633 | 0.0 | 0 |
| | | 0.50 | 0 | 1139 | 9994.0 | 0 | 0 | 1327 | 0.3 | 0 |
| | | 0.80 | 0 | 1430 | 9995.5 | 0 | 0 | 1509 | 0.7 | 0 |
| | | 1.00 | 0 | 1561 | 10000.3 | 0 | 0 | 1808 | 0.9 | 0 |
| | 1 | 0.10 | 0 | 836 | 9999.9 | 0 | 0 | 866 | 0.1 | 0 |
| | | 0.50 | 0 | 1383 | 10000.2 | 0 | 0 | 1579 | 0.8 | 0 |
| | | 0.80 | 0 | 0 | 9999.6 | 0 | 0 | 0 | 9989.1 | 0 |
| | | 1.00 | 0 | 0 | 9991.5 | 0 | 0 | 0 | 10000.2 | 0 |
| | 2 | 0.10 | 0 | 959 | 9998.8 | 0 | 0 | 1237 | 0.1 | 0 |
| | | 0.50 | 0 | 1700 | 9995.3 | 0 | 0 | 1861 | 0.6 | 0 |
| | | 0.80 | 0 | 0 | 9995.4 | 0 | 0 | 0 | 9971.4 | 0 |
| | | 1.00 | 0 | 0 | 10000.4 | 0 | 0 | 0 | 9981.4 | 0 |
| | 3 | 0.10 | 0 | 789 | 9995.4 | 0 | 0 | 1031 | 0.0 | 0 |
| | | 0.50 | 0 | 1348 | 10000.4 | 0 | 0 | 1380 | 0.3 | 0 |
| | | 0.80 | 0 | 1796 | 9995.4 | 0 | 0 | 1939 | 0.8 | 0 |
| | | 1.00 | 0 | 0 | 9999.7 | 0 | 0 | 0 | 10000.4 | 0 |
| | 4 | 0.10 | 0 | 701 | 16.4 | 1 | 0 | 994 | 0.0 | 0 |
| | | 0.50 | 0 | 1578 | 10000.3 | 0 | 0 | 1629 | 0.4 | 0 |
| | | 0.80 | 0 | 1710 | 9993.9 | 0 | 0 | 1745 | 0.8 | 0 |
| | | 1.00 | 0 | 1767 | 10000.2 | 0 | 0 | 2114 | 1.2 | 0 |
| | 5 | 0.10 | 0 | 606 | 283.0 | 1 | 0 | 737 | 0.0 | 0 |
| | | 0.50 | 0 | 1287 | 10000.4 | 0 | 0 | 1422 | 0.3 | 0 |
| | | 0.80 | 0 | 1533 | 9998.8 | 0 | 0 | 1736 | 0.7 | 0 |
| | | 1.00 | 0 | 0 | 9993.9 | 0 | 0 | 0 | 9999.5 | 0 |
| | 6 | 0.10 | 0 | 823 | 9995.6 | 0 | 0 | 875 | 0.1 | 0 |
| | | 0.50 | 0 | 1359 | 9999.5 | 0 | 0 | 1479 | 0.6 | 0 |
| | | 0.80 | 0 | 1689 | 10000.4 | 0 | 0 | 1731 | 1.6 | 0 |
| | | 1.00 | 0 | 0 | 9994.8 | 0 | 0 | 0 | 9940.7 | 0 |
| | 7 | 0.10 | 0 | 759 | 10000.0 | 0 | 0 | 910 | 0.0 | 0 |
| | | 0.50 | 0 | 1390 | 10000.4 | 0 | 0 | 1627 | 0.5 | 0 |
| | | 0.80 | 0 | 1780 | 10000.1 | 0 | 0 | 1939 | 1.4 | 0 |
| | | 1.00 | 0 | 2048 | 9995.4 | 0 | 0 | 2408 | 2.1 | 0 |
| | 8 | 0.10 | 0 | 1208 | 10000.5 | 0 | 0 | 1491 | 0.0 | 0 |
| | | 0.50 | 0 | 1395 | 10000.2 | 0 | 0 | 1688 | 0.5 | 0 |
| | | 0.80 | 0 | 1534 | 9998.2 | 0 | 0 | 1739 | 1.0 | 0 |
| | | 1.00 | 0 | 0 | 9994.3 | 0 | 0 | 0 | 10000.4 | 0 |
| | 9 | 0.10 | 0 | 571 | 1232.1 | 1 | 0 | 834 | 0.0 | 0 |
| | | 0.50 | 0 | 1409 | 9995.2 | 0 | 0 | 1813 | 0.5 | 0 |
| | | 0.80 | 0 | 1966 | 9999.1 | 0 | 0 | 2170 | 1.4 | 0 |
| | | 1.00 | 0 | 0 | 9995.5 | 0 | 0 | 0 | 10000.4 | 0 |
| | 10 | 0.10 | 0 | 593 | 3.9 | 1 | 0 | 610 | 0.0 | 0 |
| | | 0.50 | 0 | 1054 | 10000.5 | 0 | 0 | 1375 | 0.2 | 0 |
| | | 0.80 | 0 | 1393 | 10000.4 | 0 | 0 | 1643 | 0.5 | 0 |
| | | 1.00 | 0 | 1560 | 9997.7 | 0 | 0 | 1930 | 0.9 | 0 |
| | 11 | 0.10 | 0 | 646 | 9995.5 | 0 | 0 | 937 | 0.0 | 0 |
| | | 0.50 | 0 | 1297 | 9991.3 | 0 | 0 | 1459 | 0.4 | 0 |
| | | 0.80 | 0 | 1682 | 9994.6 | 0 | 0 | 2088 | 0.7 | 0 |
| | | 1.00 | 0 | 2151 | 9993.7 | 0 | 0 | 2327 | 1.7 | 0 |
| | 12 | 0.10 | 0 | 754 | 0.1 | 1 | 0 | 869 | 0.0 | 0 |
| | | 0.50 | 0 | 1300 | 9995.4 | 0 | 0 | 1439 | 0.3 | 0 |
| | | 0.80 | 0 | 1703 | 10000.5 | 0 | 0 | 2015 | 0.7 | 0 |
| | | 1.00 | 0 | 0 | 9998.1 | 0 | 0 | 0 | 9998.9 | 0 |
| | 13 | 0.10 | 0 | 542 | 3.7 | 1 | 0 | 686 | 0.0 | 0 |
| | | 0.50 | 0 | 1184 | 10000.3 | 0 | 0 | 1464 | 0.3 | 0 |
| | | 0.80 | 0 | 1535 | 10000.3 | 0 | 0 | 1787 | 0.8 | 0 |
| | | 1.00 | 0 | 1719 | 9996.9 | 0 | 0 | 1967 | 1.1 | 0 |
| | 14 | 0.10 | 0 | 749 | 2316.7 | 1 | 0 | 891 | 0.0 | 0 |
| | | 0.50 | 0 | 1391 | 10000.4 | 0 | 0 | 1813 | 0.4 | 0 |
| | | 0.80 | 0 | 1695 | 10000.4 | 0 | 0 | 1865 | 1.0 | 0 |
| | | 1.00 | 0 | 2162 | 10000.2 | 0 | 0 | 2162 | 1.7 | 0 |

Table A.33: Performance of CP-O and CP-S for instance size (Sz.) 30, instance numbers (Nr.) 15–29.

| | | | CP-O | | | | CP-S | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** |
| 30 | 15 | 0.10 | 0 | 977 | 70.0 | 1 | 0 | 1209 | 0.0 | 0 |
| | | 0.50 | 0 | 1270 | 143.2 | 1 | 0 | 1633 | 0.3 | 0 |
| | | 0.80 | 0 | 1691 | 9995.1 | 0 | 0 | 1903 | 0.9 | 0 |
| | | 1.00 | 0 | 1853 | 9993.5 | 0 | 0 | 2304 | 1.4 | 0 |
| | 16 | 0.10 | 0 | 458 | 288.9 | 1 | 0 | 633 | 0.0 | 0 |
| | | 0.50 | 0 | 1186 | 9995.5 | 0 | 0 | 1556 | 0.3 | 0 |
| | | 0.80 | 0 | 1700 | 9995.6 | 0 | 0 | 1778 | 0.6 | 0 |
| | | 1.00 | 0 | 1810 | 9999.8 | 0 | 0 | 1931 | 0.9 | 0 |
| | 17 | 0.10 | 0 | 784 | 9994.2 | 0 | 0 | 992 | 0.1 | 0 |
| | | 0.50 | 0 | 1392 | 10000.4 | 0 | 0 | 1392 | 0.5 | 0 |
| | | 0.80 | 0 | 1588 | 9896.3 | 0 | 0 | 1934 | 1.2 | 0 |
| | | 1.00 | 0 | 0 | 9995.5 | 0 | 0 | 0 | 10000.1 | 0 |
| | 18 | 0.10 | 0 | 561 | 0.1 | 1 | 0 | 646 | 0.0 | 0 |
| | | 0.50 | 0 | 1271 | 10000.4 | 0 | 0 | 1481 | 0.3 | 0 |
| | | 0.80 | 0 | 1596 | 9994.5 | 0 | 0 | 1831 | 0.6 | 0 |
| | | 1.00 | 0 | 1879 | 9995.6 | 0 | 0 | 2292 | 1.0 | 0 |
| | 19 | 0.10 | 0 | 498 | 4880.8 | 1 | 0 | 975 | 0.0 | 0 |
| | | 0.50 | 0 | 1394 | 10000.3 | 0 | 0 | 1640 | 0.7 | 0 |
| | | 0.80 | 0 | 1669 | 9999.0 | 0 | 0 | 1962 | 1.8 | 0 |
| | | 1.00 | 0 | 0 | 9995.2 | 0 | 0 | 0 | 9995.5 | 0 |
| | 20 | 0.10 | 0 | 910 | 9994.9 | 0 | 0 | 1171 | 0.0 | 0 |
| | | 0.50 | 0 | 1525 | 9995.2 | 0 | 0 | 1803 | 0.5 | 0 |
| | | 0.80 | 0 | 1791 | 9995.3 | 0 | 0 | 2023 | 1.2 | 0 |
| | | 1.00 | 0 | 1999 | 10000.1 | 0 | 0 | 2157 | 2.0 | 0 |
| | 21 | 0.10 | 0 | 640 | 10000.2 | 0 | 0 | 897 | 0.0 | 0 |
| | | 0.50 | 0 | 994 | 10000.4 | 0 | 0 | 1283 | 0.2 | 0 |
| | | 0.80 | 0 | 1613 | 9995.4 | 0 | 0 | 1636 | 0.6 | 0 |
| | | 1.00 | 0 | 1844 | 9995.5 | 0 | 0 | 1968 | 1.9 | 0 |
| | 22 | 0.10 | 0 | 881 | 116.0 | 1 | 0 | 1003 | 0.0 | 0 |
| | | 0.50 | 0 | 1437 | 10000.4 | 0 | 0 | 1554 | 0.3 | 0 |
| | | 0.80 | 0 | 1481 | 10000.2 | 0 | 0 | 1958 | 0.7 | 0 |
| | | 1.00 | 0 | 1881 | 9919.3 | 0 | 0 | 2268 | 1.1 | 0 |
| | 23 | 0.10 | 0 | 738 | 3.0 | 1 | 0 | 755 | 0.0 | 0 |
| | | 0.50 | 0 | 1753 | 9994.7 | 0 | 0 | 1764 | 0.5 | 0 |
| | | 0.80 | 0 | 2003 | 9995.6 | 0 | 0 | 2254 | 1.0 | 0 |
| | | 1.00 | 0 | 0 | 9962.3 | 0 | 0 | 0 | 9946.7 | 0 |
| | 24 | 0.10 | 0 | 525 | 52.9 | 1 | 0 | 630 | 0.1 | 0 |
| | | 0.50 | 0 | 1459 | 10000.3 | 0 | 0 | 1543 | 0.6 | 0 |
| | | 0.80 | 0 | 1707 | 9995.1 | 0 | 0 | 2123 | 1.8 | 0 |
| | | 1.00 | 0 | 0 | 10000.1 | 0 | 0 | 0 | 9995.5 | 0 |
| | 25 | 0.10 | 0 | 930 | 9995.5 | 0 | 0 | 1198 | 0.0 | 0 |
| | | 0.50 | 0 | 1592 | 9963.5 | 0 | 0 | 1756 | 0.5 | 0 |
| | | 0.80 | 0 | 1728 | 9877.1 | 0 | 0 | 2019 | 1.2 | 0 |
| | | 1.00 | 0 | 2154 | 9999.9 | 0 | 0 | 2247 | 1.9 | 0 |
| | 26 | 0.10 | 0 | 653 | 9999.6 | 0 | 0 | 763 | 0.0 | 0 |
| | | 0.50 | 0 | 1304 | 10000.3 | 0 | 0 | 1509 | 0.4 | 0 |
| | | 0.80 | 0 | 1598 | 9995.1 | 0 | 0 | 1977 | 1.0 | 0 |
| | | 1.00 | 0 | 0 | 9999.4 | 0 | 0 | 0 | 9979.9 | 0 |
| | 27 | 0.10 | 0 | 549 | 1.2 | 1 | 0 | 751 | 0.0 | 0 |
| | | 0.50 | 0 | 1198 | 10000.3 | 0 | 0 | 1432 | 0.3 | 0 |
| | | 0.80 | 0 | 1574 | 9995.5 | 0 | 0 | 1693 | 0.7 | 0 |
| | | 1.00 | 0 | 1843 | 10000.0 | 0 | 0 | 2124 | 1.3 | 0 |
| | 28 | 0.10 | 0 | 962 | 10000.4 | 0 | 0 | 1031 | 0.0 | 0 |
| | | 0.50 | 0 | 1317 | 9995.2 | 0 | 0 | 1422 | 0.3 | 0 |
| | | 0.80 | 0 | 1455 | 10000.4 | 0 | 0 | 2013 | 0.8 | 0 |
| | | 1.00 | 0 | 2013 | 9995.0 | 0 | 0 | 2191 | 1.3 | 0 |
| | 29 | 0.10 | 0 | 411 | 521.5 | 1 | 0 | 915 | 0.0 | 0 |
| | | 0.50 | 0 | 1094 | 10000.3 | 0 | 0 | 1524 | 0.3 | 0 |
| | | 0.80 | 0 | 1639 | 9999.6 | 0 | 0 | 1858 | 0.7 | 0 |
| | | 1.00 | 0 | 1751 | 9903.7 | 0 | 0 | 1920 | 0.9 | 0 |

Table A.34: Performance of CP-O and CP-S for instance size (Sz.) 50, instance numbers (Nr.) 0–14.

| | | | CP-O | | | | CP-S | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** |
| 50 | 0 | 0.10 | 0 | 1085 | 9908.4 | 0 | 0 | 1271 | 0.1 | 0 |
| | | 0.50 | 0 | 2186 | 9992.4 | 0 | 0 | 2410 | 2.0 | 0 |
| | | 0.80 | 0 | 2686 | 9988.8 | 0 | 0 | 3047 | 4.8 | 0 |
| | | 1.00 | 0 | 0 | 10000.3 | 0 | 0 | 0 | 9932.4 | 0 |
| | 1 | 0.10 | 0 | 1425 | 10000.4 | 0 | 0 | 1492 | 0.2 | 0 |
| | | 0.50 | 0 | 1955 | 9997.5 | 0 | 0 | 2277 | 1.0 | 0 |
| | | 0.80 | 0 | 2376 | 10000.1 | 0 | 0 | 2744 | 2.8 | 0 |
| | | 1.00 | 0 | 0 | 9999.9 | 0 | 0 | 0 | 9974.1 | 0 |
| | 2 | 0.10 | 0 | 830 | 9960.1 | 0 | 0 | 1175 | 0.0 | 0 |
| | | 0.50 | 0 | 2058 | 9995.4 | 0 | 0 | 2453 | 1.0 | 0 |
| | | 0.80 | 0 | 2794 | 10000.2 | 0 | 0 | 2988 | 2.3 | 0 |
| | | 1.00 | 0 | 0 | 9995.2 | 0 | 0 | 0 | 9995.4 | 0 |
| | 3 | 0.10 | 0 | 1171 | 9995.3 | 0 | 0 | 1425 | 0.1 | 0 |
| | | 0.50 | 0 | 2208 | 9995.4 | 0 | 0 | 2645 | 1.1 | 0 |
| | | 0.80 | 0 | 2864 | 10000.0 | 0 | 0 | 3430 | 2.7 | 0 |
| | | 1.00 | 0 | 0 | 10000.1 | 0 | 0 | 0 | 10000.1 | 0 |
| | 4 | 0.10 | 0 | 1213 | 9995.3 | 0 | 0 | 1317 | 0.1 | 0 |
| | | 0.50 | 0 | 2182 | 10000.1 | 0 | 0 | 2237 | 1.0 | 0 |
| | | 0.80 | 0 | 2443 | 9989.1 | 0 | 0 | 2625 | 2.2 | 0 |
| | | 1.00 | 0 | 3191 | 10000.3 | 0 | 0 | 3255 | 3.4 | 0 |
| | 5 | 0.10 | 0 | 1344 | 9901.0 | 0 | 0 | 1429 | 0.1 | 0 |
| | | 0.50 | 0 | 2000 | 9995.5 | 0 | 0 | 2065 | 0.8 | 0 |
| | | 0.80 | 0 | 2429 | 10000.2 | 0 | 0 | 2742 | 2.6 | 0 |
| | | 1.00 | 0 | 2555 | 9923.1 | 0 | 0 | 3188 | 58.9 | 0 |
| | 6 | 0.10 | 0 | 852 | 9921.5 | 0 | 0 | 861 | 0.1 | 0 |
| | | 0.50 | 0 | 2062 | 10000.4 | 0 | 0 | 2342 | 1.2 | 0 |
| | | 0.80 | 0 | 2519 | 9980.4 | 0 | 0 | 2708 | 2.8 | 0 |
| | | 1.00 | 0 | 2990 | 10000.5 | 0 | 0 | 3088 | 4.1 | 0 |
| | 7 | 0.10 | 0 | 1116 | 10000.3 | 0 | 0 | 1169 | 0.0 | 0 |
| | | 0.50 | 0 | 2449 | 9995.6 | 0 | 0 | 2480 | 0.9 | 0 |
| | | 0.80 | 0 | 2702 | 9998.0 | 0 | 0 | 2795 | 2.2 | 0 |
| | | 1.00 | 0 | 0 | 10000.5 | 0 | 0 | 0 | 9995.5 | 0 |
| | 8 | 0.10 | 0 | 1009 | 9995.6 | 0 | 0 | 1084 | 0.1 | 0 |
| | | 0.50 | 0 | 1960 | 9995.4 | 0 | 0 | 2181 | 2.2 | 0 |
| | | 0.80 | 0 | 2333 | 9968.5 | 0 | 0 | 2843 | 5.5 | 0 |
| | | 1.00 | 0 | 0 | 10000.2 | 0 | 0 | 0 | 10000.2 | 0 |
| | 9 | 0.10 | 0 | 1483 | 9999.4 | 0 | 0 | 1617 | 0.1 | 0 |
| | | 0.50 | 0 | 1992 | 9945.4 | 0 | 0 | 2289 | 0.6 | 0 |
| | | 0.80 | 0 | 2258 | 10000.2 | 0 | 0 | 2587 | 1.7 | 0 |
| | | 1.00 | 0 | 0 | 9987.6 | 0 | 0 | 0 | 9993.3 | 0 |
| | 10 | 0.10 | 0 | 1006 | 9990.2 | 0 | 0 | 1046 | 0.1 | 0 |
| | | 0.50 | 0 | 2307 | 9996.0 | 0 | 0 | 2674 | 1.3 | 0 |
| | | 0.80 | 0 | 2839 | 9933.2 | 0 | 0 | 2942 | 2.7 | 0 |
| | | 1.00 | 0 | 3473 | 9995.4 | 0 | 0 | 3782 | 4.7 | 0 |
| | 11 | 0.10 | 0 | 1201 | 10000.4 | 0 | 0 | 1402 | 0.0 | 0 |
| | | 0.50 | 0 | 2183 | 9995.6 | 0 | 0 | 2507 | 0.7 | 0 |
| | | 0.80 | 0 | 2591 | 9995.4 | 0 | 0 | 2893 | 2.0 | 0 |
| | | 1.00 | 0 | 0 | 9945.9 | 0 | 0 | 0 | 9997.3 | 0 |
| | 12 | 0.10 | 0 | 956 | 9999.6 | 0 | 0 | 1098 | 0.1 | 0 |
| | | 0.50 | 0 | 2202 | 10000.3 | 0 | 0 | 2326 | 1.1 | 0 |
| | | 0.80 | 0 | 2418 | 9995.5 | 0 | 0 | 2600 | 2.7 | 0 |
| | | 1.00 | 0 | 0 | 9999.4 | 0 | 0 | 0 | 10000.4 | 0 |
| | 13 | 0.10 | 0 | 1136 | 9876.1 | 0 | 0 | 1375 | 0.1 | 0 |
| | | 0.50 | 0 | 2058 | 9995.6 | 0 | 0 | 2291 | 1.0 | 0 |
| | | 0.80 | 0 | 2710 | 9995.5 | 0 | 0 | 3143 | 2.1 | 0 |
| | | 1.00 | 0 | 0 | 9998.7 | 0 | 0 | 0 | 10000.4 | 0 |
| | 14 | 0.10 | 0 | 1105 | 10000.5 | 0 | 0 | 1314 | 0.1 | 0 |
| | | 0.50 | 0 | 2038 | 9995.6 | 0 | 0 | 2144 | 0.7 | 0 |
| | | 0.80 | 0 | 2593 | 9995.5 | 0 | 0 | 2756 | 2.3 | 0 |
| | | 1.00 | 0 | 2564 | 10000.5 | 0 | 0 | 2937 | 3.2 | 0 |

Table A.35: Performance of CP-O and CP-S for instance size (Sz.) 50, instance numbers (Nr.) 15–29.

| | | | CP-O | | | | CP-S | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** |
| 50 | 15 | 0.10 | 0 | 1314 | 10000.4 | 0 | 0 | 1350 | 0.1 | 0 |
| | | 0.50 | 0 | 2353 | 9967.5 | 0 | 0 | 2582 | 1.4 | 0 |
| | | 0.80 | 0 | 2965 | 10000.5 | 0 | 0 | 3149 | 3.7 | 0 |
| | | 1.00 | 0 | 3793 | 9911.8 | 0 | 0 | 3793 | 5.5 | 0 |
| | 16 | 0.10 | 0 | 1232 | 9999.2 | 0 | 0 | 1306 | 0.1 | 0 |
| | | 0.50 | 0 | 1894 | 9995.5 | 0 | 0 | 2243 | 1.4 | 0 |
| | | 0.80 | 0 | 2443 | 10000.4 | 0 | 0 | 2739 | 3.1 | 0 |
| | | 1.00 | 0 | 2829 | 9937.9 | 0 | 0 | 3041 | 5.2 | 0 |
| | 17 | 0.10 | 0 | 1321 | 9999.6 | 0 | 0 | 1400 | 0.1 | 0 |
| | | 0.50 | 0 | 2518 | 9995.5 | 0 | 0 | 2633 | 1.0 | 0 |
| | | 0.80 | 0 | 3089 | 9926.8 | 0 | 0 | 3278 | 2.5 | 0 |
| | | 1.00 | 0 | 0 | 9745.9 | 0 | 0 | 0 | 9691.6 | 0 |
| | 18 | 0.10 | 0 | 839 | 408.5 | 1 | 0 | 1093 | 0.1 | 0 |
| | | 0.50 | 0 | 2134 | 9999.0 | 0 | 0 | 2239 | 1.1 | 0 |
| | | 0.80 | 0 | 2812 | 10000.3 | 0 | 0 | 2972 | 2.7 | 0 |
| | | 1.00 | 0 | 3142 | 10000.3 | 0 | 0 | 3494 | 4.6 | 0 |
| | 19 | 0.10 | 0 | 949 | 10000.4 | 0 | 0 | 1458 | 0.1 | 0 |
| | | 0.50 | 0 | 1955 | 10000.4 | 0 | 0 | 2236 | 1.2 | 0 |
| | | 0.80 | 0 | 2601 | 10000.3 | 0 | 0 | 2788 | 2.8 | 0 |
| | | 1.00 | 0 | 0 | 9995.5 | 0 | 0 | 0 | 9976.4 | 0 |
| | 20 | 0.10 | 0 | 1819 | 9908.4 | 0 | 0 | 1939 | 0.1 | 0 |
| | | 0.50 | 0 | 2402 | 9982.7 | 0 | 0 | 2485 | 0.9 | 0 |
| | | 0.80 | 0 | 2821 | 9995.5 | 0 | 0 | 2844 | 2.4 | 0 |
| | | 1.00 | 0 | 3282 | 9989.9 | 0 | 0 | 3290 | 3.2 | 0 |
| | 21 | 0.10 | 0 | 1254 | 10000.4 | 0 | 0 | 1482 | 0.1 | 0 |
| | | 0.50 | 0 | 1944 | 9980.3 | 0 | 0 | 2275 | 1.0 | 0 |
| | | 0.80 | 0 | 3023 | 9992.6 | 0 | 0 | 3065 | 2.7 | 0 |
| | | 1.00 | 0 | 3426 | 9915.0 | 0 | 0 | 3484 | 3.9 | 0 |
| | 22 | 0.10 | 0 | 1495 | 9991.1 | 0 | 0 | 1607 | 0.1 | 0 |
| | | 0.50 | 0 | 2666 | 10000.5 | 0 | 0 | 2778 | 1.3 | 0 |
| | | 0.80 | 0 | 2919 | 9991.9 | 0 | 0 | 3176 | 2.5 | 0 |
| | | 1.00 | 0 | 0 | 9994.6 | 0 | 0 | 0 | 9981.2 | 0 |
| | 23 | 0.10 | 0 | 1060 | 10000.5 | 0 | 0 | 1228 | 0.1 | 0 |
| | | 0.50 | 0 | 2444 | 9948.0 | 0 | 0 | 2700 | 0.9 | 0 |
| | | 0.80 | 0 | 2976 | 10000.5 | 0 | 0 | 3151 | 1.9 | 0 |
| | | 1.00 | 0 | 3061 | 10000.3 | 0 | 0 | 3201 | 3.0 | 0 |
| | 24 | 0.10 | 0 | 984 | 9994.1 | 0 | 0 | 1138 | 0.1 | 0 |
| | | 0.50 | 0 | 2203 | 10000.5 | 0 | 0 | 2480 | 1.3 | 0 |
| | | 0.80 | 0 | 3281 | 9995.4 | 0 | 0 | 3350 | 3.5 | 0 |
| | | 1.00 | 0 | 3572 | 9995.4 | 0 | 0 | 3705 | 5.3 | 0 |
| | 25 | 0.10 | 0 | 982 | 9867.5 | 0 | 0 | 1293 | 0.1 | 0 |
| | | 0.50 | 0 | 2223 | 9996.9 | 0 | 0 | 2420 | 1.1 | 0 |
| | | 0.80 | 0 | 0 | 9921.9 | 0 | 0 | 0 | 9995.5 | 0 |
| | | 1.00 | 0 | 0 | 9885.6 | 0 | 0 | 0 | 10000.4 | 0 |
| | 26 | 0.10 | 0 | 941 | 9995.5 | 0 | 0 | 1146 | 0.0 | 0 |
| | | 0.50 | 0 | 2345 | 10000.3 | 0 | 0 | 2405 | 1.0 | 0 |
| | | 0.80 | 0 | 2737 | 10000.4 | 0 | 0 | 2933 | 2.3 | 0 |
| | | 1.00 | 0 | 2867 | 9961.0 | 0 | 0 | 3106 | 3.7 | 0 |
| | 27 | 0.10 | 0 | 1397 | 9997.6 | 0 | 0 | 1497 | 0.1 | 0 |
| | | 0.50 | 0 | 2356 | 9995.6 | 0 | 0 | 2507 | 0.9 | 0 |
| | | 0.80 | 0 | 2537 | 9931.7 | 0 | 0 | 2856 | 2.0 | 0 |
| | | 1.00 | 0 | 2778 | 9995.2 | 0 | 0 | 3160 | 3.1 | 0 |
| | 28 | 0.10 | 0 | 1505 | 10000.3 | 0 | 0 | 1772 | 0.3 | 0 |
| | | 0.50 | 0 | 2672 | 9995.4 | 0 | 0 | 2978 | 3.5 | 0 |
| | | 0.80 | 0 | 0 | 9995.5 | 0 | 0 | 0 | 9998.1 | 0 |
| | | 1.00 | 0 | 0 | 10000.4 | 0 | 0 | 0 | 9995.4 | 0 |
| | 29 | 0.10 | 0 | 620 | 1433.3 | 1 | 0 | 865 | 0.1 | 0 |
| | | 0.50 | 0 | 2459 | 9995.5 | 0 | 0 | 2650 | 1.1 | 0 |
| | | 0.80 | 0 | 3332 | 9995.1 | 0 | 0 | 3383 | 4.0 | 0 |
| | | 1.00 | 0 | 0 | 10000.4 | 0 | 0 | 0 | 9950.3 | 0 |

Table A.36: Performance of CP-O and CP-S for instance size (Sz.) 100, instance numbers (Nr.) 0–14.

| | | | CP-O | | | | CP-S | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** |
| 100 | 0 | 0.10 | 0 | 2554 | 9995.8 | 0 | 0 | 2637 | 0.3 | 0 |
| | | 0.50 | 0 | 3969 | 9985.8 | 0 | 0 | 4261 | 6.1 | 0 |
| | | 0.80 | 0 | 0 | 9960.6 | 0 | 0 | 0 | 9995.5 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 1 | 0.10 | 0 | 3372 | 9929.4 | 0 | 0 | 3372 | 0.6 | 0 |
| | | 0.50 | 0 | 5134 | 10000.4 | 0 | 0 | 5345 | 6.0 | 0 |
| | | 0.80 | 0 | 0 | 9768.7 | 0 | 0 | 0 | 9760.6 | 0 |
| | | 1.00 | 0 | 0 | 9692.8 | 0 | 0 | 0 | 9685.4 | 0 |
| | 2 | 0.10 | 0 | 1360 | 9970.4 | 0 | 0 | 1522 | 0.3 | 0 |
| | | 0.50 | 0 | 3562 | 10000.1 | 0 | 0 | 3992 | 4.1 | 0 |
| | | 0.80 | 0 | 4797 | 9421.4 | 0 | 0 | 4797 | 11.0 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 3 | 0.10 | 0 | 2891 | 9924.9 | 0 | 0 | 2961 | 1.0 | 0 |
| | | 0.50 | 0 | 5407 | 10000.5 | 0 | 0 | 5541 | 8.2 | 0 |
| | | 0.80 | 0 | 0 | 9997.4 | 0 | 0 | 0 | 9998.0 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 4 | 0.10 | 0 | 2519 | 9985.2 | 0 | 0 | 2572 | 0.8 | 0 |
| | | 0.50 | 0 | 4973 | 9995.0 | 0 | 0 | 5055 | 9.9 | 0 |
| | | 0.80 | 0 | 0 | 9999.4 | 0 | 0 | 0 | 9999.6 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 5 | 0.10 | 0 | 2060 | 10000.5 | 0 | 0 | 2381 | 0.5 | 0 |
| | | 0.50 | 0 | 4506 | 10000.2 | 0 | 0 | 4618 | 6.6 | 0 |
| | | 0.80 | 0 | 5223 | 9995.1 | 0 | 0 | 5500 | 18.1 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 6 | 0.10 | 0 | 2706 | 9981.8 | 0 | 0 | 2759 | 1.9 | 0 |
| | | 0.50 | 0 | 4679 | 9729.9 | 0 | 0 | 4679 | 7.9 | 0 |
| | | 0.80 | 0 | 5946 | 9998.9 | 0 | 0 | 6019 | 15.8 | 0 |
| | | 1.00 | 0 | 0 | 9997.4 | 0 | 0 | 0 | 9988.3 | 0 |
| | 7 | 0.10 | 0 | 3959 | 9995.6 | 0 | 0 | 3981 | 0.5 | 0 |
| | | 0.50 | 0 | 5139 | 9779.4 | 0 | 0 | 5213 | 5.6 | 0 |
| | | 0.80 | 0 | 5960 | 9939.0 | 0 | 0 | 6165 | 14.8 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 8 | 0.10 | 0 | 2139 | 9989.1 | 0 | 0 | 2362 | 0.4 | 0 |
| | | 0.50 | 0 | 4236 | 10000.4 | 0 | 0 | 4602 | 3.6 | 0 |
| | | 0.80 | 0 | 0 | 9961.5 | 0 | 0 | 0 | 9995.4 | 0 |
| | | 1.00 | 0 | 0 | 9997.7 | 0 | 0 | 0 | 10000.3 | 0 |
| | 9 | 0.10 | 0 | 2465 | 9934.6 | 0 | 0 | 2622 | 0.6 | 0 |
| | | 0.50 | 0 | 0 | 10000.1 | 0 | 0 | 0 | 10000.1 | 0 |
| | | 0.80 | 0 | 0 | 10000.3 | 0 | 0 | 0 | 10000.3 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 10 | 0.10 | 0 | 2809 | 9995.4 | 0 | 0 | 2947 | 0.6 | 0 |
| | | 0.50 | 0 | 3331 | 9997.6 | 0 | 0 | 4073 | 3.2 | 0 |
| | | 0.80 | 0 | 4761 | 9994.5 | 0 | 0 | 5114 | 8.3 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 11 | 0.10 | 0 | 3759 | 9907.6 | 0 | 0 | 3965 | 4.9 | 0 |
| | | 0.50 | 0 | 5520 | 10000.2 | 0 | 0 | 5603 | 9.4 | 0 |
| | | 0.80 | 0 | 6380 | 9993.9 | 0 | 0 | 6535 | 21.5 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 12 | 0.10 | 0 | 2713 | 9999.0 | 0 | 0 | 2865 | 0.4 | 0 |
| | | 0.50 | 0 | 4408 | 9995.1 | 0 | 0 | 5229 | 5.3 | 0 |
| | | 0.80 | - | - | - | - | - | - | - | - |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 13 | 0.10 | 0 | 2761 | 9995.4 | 0 | 0 | 2777 | 0.3 | 0 |
| | | 0.50 | 0 | 4672 | 10000.0 | 0 | 0 | 5001 | 7.7 | 0 |
| | | 0.80 | 0 | 5621 | 9998.7 | 0 | 0 | 5996 | 16.2 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 14 | 0.10 | 0 | 3103 | 9992.2 | 0 | 0 | 3263 | 0.4 | 0 |
| | | 0.50 | 0 | 4858 | 9932.5 | 0 | 0 | 5117 | 6.0 | 0 |
| | | 0.80 | 0 | 6114 | 9999.1 | 0 | 0 | 6355 | 14.8 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |

Table A.37: Performance of CP-O and CP-S for instance size (Sz.) 100, instance numbers (Nr.) 15–29.

| | | | CP-O | | | | CP-S | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Opt** |
| 100 | 15 | 0.10 | 0 | 2421 | 9995.0 | 0 | 0 | 2548 | 0.4 | 0 |
| | | 0.50 | 0 | 4060 | 9964.9 | 0 | 0 | 4256 | 3.9 | 0 |
| | | 0.80 | 0 | 5273 | 9992.0 | 0 | 0 | 5415 | 8.4 | 0 |
| | | 1.00 | 0 | 0 | 9994.9 | 0 | 0 | 0 | 9994.9 | 0 |
| | 16 | 0.10 | 0 | 3285 | 9886.6 | 0 | 0 | 3320 | 0.3 | 0 |
| | | 0.50 | 0 | 4641 | 9995.3 | 0 | 0 | 4772 | 3.5 | 0 |
| | | 0.80 | 0 | 5309 | 9995.1 | 0 | 0 | 5814 | 10.2 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 17 | 0.10 | 0 | 2492 | 9992.1 | 0 | 0 | 2548 | 0.4 | 0 |
| | | 0.50 | 0 | 5346 | 10000.5 | 0 | 0 | 5412 | 3.9 | 0 |
| | | 0.80 | 0 | 6218 | 9993.8 | 0 | 0 | 6342 | 8.3 | 0 |
| | | 1.00 | 0 | 0 | 9949.1 | 0 | 0 | 0 | 9995.4 | 0 |
| | 18 | 0.10 | 0 | 2326 | 9966.6 | 0 | 0 | 2414 | 0.3 | 0 |
| | | 0.50 | 0 | 4012 | 9970.1 | 0 | 0 | 4356 | 4.4 | 0 |
| | | 0.80 | 0 | 4865 | 9992.9 | 0 | 0 | 5144 | 11.8 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 19 | 0.10 | 0 | 2762 | 9927.5 | 0 | 0 | 2762 | 0.4 | 0 |
| | | 0.50 | 0 | 3701 | 9994.9 | 0 | 0 | 4485 | 8.5 | 0 |
| | | 0.80 | - | - | - | - | - | - | - | - |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 20 | 0.10 | 0 | 1923 | 9999.7 | 0 | 0 | 1964 | 0.3 | 0 |
| | | 0.50 | 0 | 4539 | 9995.4 | 0 | 0 | 4730 | 5.0 | 0 |
| | | 0.80 | 0 | 5503 | 9974.4 | 0 | 0 | 5925 | 10.9 | 0 |
| | | 1.00 | 0 | 6091 | 9999.6 | 0 | 0 | 6401 | 15.7 | 0 |
| | 21 | 0.10 | 0 | 2722 | 9992.4 | 0 | 0 | 2740 | 0.7 | 0 |
| | | 0.50 | 0 | 3871 | 9995.4 | 0 | 0 | 4494 | 7.7 | 0 |
| | | 0.80 | 0 | 0 | 9995.2 | 0 | 0 | 0 | 9963.1 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 22 | 0.10 | 0 | 1172 | 10000.5 | 0 | 0 | 1246 | 0.2 | 0 |
| | | 0.50 | 0 | 3893 | 9997.2 | 0 | 0 | 4201 | 3.5 | 0 |
| | | 0.80 | 0 | 5498 | 9998.0 | 0 | 0 | 5715 | 10.1 | 0 |
| | | 1.00 | 0 | 0 | 9953.2 | 0 | 0 | 0 | 9995.0 | 0 |
| | 23 | 0.10 | 0 | 2056 | 10000.3 | 0 | 0 | 2066 | 0.4 | 0 |
| | | 0.50 | 0 | 4217 | 10000.3 | 0 | 0 | 4468 | 5.0 | 0 |
| | | 0.80 | 0 | 5638 | 9999.5 | 0 | 0 | 5799 | 13.9 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 24 | 0.10 | 0 | 2953 | 9992.8 | 0 | 0 | 3214 | 0.3 | 0 |
| | | 0.50 | 0 | 5056 | 10000.2 | 0 | 0 | 5091 | 4.8 | 0 |
| | | 0.80 | 0 | 6131 | 10000.1 | 0 | 0 | 6309 | 12.2 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 25 | 0.10 | 0 | 2473 | 9904.0 | 0 | 0 | 2549 | 0.3 | 0 |
| | | 0.50 | 0 | 4048 | 9997.9 | 0 | 0 | 4582 | 4.6 | 0 |
| | | 0.80 | 0 | 4960 | 9999.6 | 0 | 0 | 5609 | 13.2 | 0 |
| | | 1.00 | - | - | - | - | - | - | - | - |
| | 26 | 0.10 | 0 | 2833 | 9982.1 | 0 | 0 | 3021 | 0.2 | 0 |
| | | 0.50 | 0 | 4380 | 10000.1 | 0 | 0 | 4437 | 3.4 | 0 |
| | | 0.80 | 0 | 5641 | 9998.6 | 0 | 0 | 5831 | 12.8 | 0 |
| | | 1.00 | 0 | 6283 | 9994.0 | 0 | 0 | 6437 | 17.8 | 0 |
| | 27 | 0.10 | 0 | 2128 | 9979.0 | 0 | 0 | 2142 | 0.4 | 0 |
| | | 0.50 | 0 | 4448 | 9975.2 | 0 | 0 | 4599 | 3.5 | 0 |
| | | 0.80 | 0 | 5608 | 9988.1 | 0 | 0 | 5924 | 9.8 | 0 |
| | | 1.00 | 0 | 0 | 9999.4 | 0 | 0 | 0 | 9976.3 | 0 |
| | 28 | 0.10 | 0 | 2603 | 10000.4 | 0 | 0 | 2908 | 0.4 | 0 |
| | | 0.50 | 0 | 4883 | 9977.1 | 0 | 0 | 5320 | 5.9 | 0 |
| | | 0.80 | 0 | 5445 | 9954.1 | 0 | 0 | 5993 | 14.4 | 0 |
| | | 1.00 | 0 | 0 | 9980.7 | 0 | 0 | 0 | 9987.2 | 0 |
| | 29 | 0.10 | 0 | 2842 | 10000.0 | 0 | 0 | 2891 | 0.7 | 0 |
| | | 0.50 | 0 | 5015 | 9965.9 | 0 | 0 | 5130 | 7.1 | 0 |
| | | 0.80 | 0 | 0 | 9999.6 | 0 | 0 | 0 | 10000.0 | 0 |
| | | 1.00 | 0 | 0 | 9999.7 | 0 | 0 | 0 | 10000.0 | 0 |

Table A.38: Performance of ILP-O and ILP-S for instance size (Sz.) 20, instance numbers (Nr.) 0–14.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 20 | 0 | 0.10 | 0 | 469 | 0.0 | 0.0 | 1 | 0 | 835 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 778 | 1.7 | 0.0 | 1 | 0 | 1255 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 977 | 0.9 | 0.0 | 1 | 0 | 1284 | 0.4 | 0.0 | 1 |
| | | 1.00 | 0 | 1057 | 2.7 | 0.0 | 1 | 0 | 1475 | 0.6 | 0.0 | 1 |
| | 1 | 0.10 | 0 | 351 | 0.0 | 0.0 | 1 | 0 | 736 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 668 | 0.2 | 0.0 | 1 | 0 | 1048 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 668 | 0.4 | 0.0 | 1 | 0 | 1096 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 786 | 3.2 | 0.0 | 1 | 0 | 1269 | 0.2 | 0.0 | 1 |
| | 2 | 0.10 | 0 | 484 | 0.4 | 0.0 | 1 | 0 | 1010 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 706 | 0.5 | 0.0 | 1 | 0 | 1353 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 862 | 3.6 | 0.0 | 1 | 0 | 1606 | 0.5 | 0.0 | 1 |
| | | 1.00 | 0 | 1028 | 646.5 | 0.0 | 1 | 0 | 1586 | 0.7 | 0.0 | 1 |
| | 3 | 0.10 | 0 | 467 | 0.0 | 0.0 | 1 | 0 | 1122 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 826 | 0.5 | 0.0 | 1 | 0 | 1445 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1055 | 18.1 | 0.0 | 1 | 0 | 1493 | 0.1 | 0.0 | 1 |
| | | 1.00 | 0 | 1185 | 5.3 | 0.0 | 1 | 0 | 1493 | 0.2 | 0.0 | 1 |
| | 4 | 0.10 | 0 | 497 | 0.0 | 0.0 | 1 | 0 | 675 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 772 | 0.1 | 0.0 | 1 | 0 | 1043 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 915 | 0.4 | 0.0 | 1 | 0 | 1304 | 0.1 | 0.0 | 1 |
| | | 1.00 | 0 | 1043 | 0.7 | 0.0 | 1 | 0 | 1488 | 0.2 | 0.0 | 1 |
| | 5 | 0.10 | 0 | 401 | 0.0 | 0.0 | 1 | 0 | 771 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 733 | 0.3 | 0.0 | 1 | 0 | 1160 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 816 | 0.9 | 0.0 | 1 | 0 | 1310 | 0.1 | 0.0 | 1 |
| | | 1.00 | 0 | 935 | 2.8 | 0.0 | 1 | 0 | 1310 | 0.2 | 0.0 | 1 |
| | 6 | 0.10 | 0 | 374 | 0.0 | 0.0 | 1 | 0 | 757 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 768 | 1.4 | 0.0 | 1 | 0 | 1326 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 941 | 2.4 | 0.0 | 1 | 0 | 1369 | 0.1 | 0.0 | 1 |
| | | 1.00 | 0 | 1159 | 2.7 | 0.0 | 1 | 0 | 1446 | 0.2 | 0.0 | 1 |
| | 7 | 0.10 | 0 | 521 | 0.0 | 0.0 | 1 | 0 | 802 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 931 | 0.8 | 0.0 | 1 | 0 | 1508 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1076 | 2.0 | 0.0 | 1 | 0 | 1494 | 0.3 | 0.0 | 1 |
| | | 1.00 | 0 | 1158 | 56.4 | 0.0 | 1 | 0 | 1596 | 0.4 | 0.0 | 1 |
| | 8 | 0.10 | 0 | 347 | 0.1 | 0.0 | 1 | 0 | 618 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 752 | 3.7 | 0.0 | 1 | 0 | 1305 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1004 | 28.6 | 0.0 | 1 | 0 | 1405 | 0.4 | 0.0 | 1 |
| | | 1.00 | 0 | 1066 | 32.7 | 0.0 | 1 | 0 | 1442 | 0.5 | 0.0 | 1 |
| | 9 | 0.10 | 0 | 643 | 0.1 | 0.0 | 1 | 0 | 1056 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 984 | 0.5 | 0.0 | 1 | 0 | 1317 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1042 | 0.8 | 0.0 | 1 | 0 | 1591 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 1106 | 1.7 | 0.0 | 1 | 0 | 1585 | 0.3 | 0.0 | 1 |
| | 10 | 0.10 | 0 | 559 | 0.0 | 0.0 | 1 | 0 | 998 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 905 | 0.2 | 0.0 | 1 | 0 | 1306 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1095 | 0.4 | 0.0 | 1 | 0 | 1502 | 0.1 | 0.0 | 1 |
| | | 1.00 | 0 | 1172 | 5.5 | 0.0 | 1 | 0 | 1502 | 0.2 | 0.0 | 1 |
| | 11 | 0.10 | 0 | 217 | 0.0 | 0.0 | 1 | 0 | 639 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 576 | 0.3 | 0.0 | 1 | 0 | 1036 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 633 | 0.5 | 0.0 | 1 | 0 | 1217 | 0.1 | 0.0 | 1 |
| | | 1.00 | 0 | 854 | 3.9 | 0.0 | 1 | 0 | 1217 | 0.2 | 0.0 | 1 |
| | 12 | 0.10 | 0 | 499 | 0.0 | 0.0 | 1 | 0 | 939 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 801 | 0.4 | 0.0 | 1 | 0 | 1246 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 944 | 2.9 | 0.0 | 1 | 0 | 1558 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 1050 | 8.9 | 0.0 | 1 | 0 | 1611 | 0.5 | 0.0 | 1 |
| | 13 | 0.10 | 0 | 476 | 0.2 | 0.0 | 1 | 0 | 1329 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 671 | 2.5 | 0.0 | 1 | 0 | 1653 | 0.7 | 0.0 | 1 |
| | | 0.80 | 0 | 878 | 53.0 | 0.0 | 1 | 0 | 1739 | 1.3 | 0.0 | 1 |
| | | 1.00 | 0 | 954 | 918.8 | 0.0 | 1 | 0 | 1739 | 1.8 | 0.0 | 1 |
| | 14 | 0.10 | 0 | 535 | 0.0 | 0.0 | 1 | 0 | 776 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 818 | 0.3 | 0.0 | 1 | 0 | 1288 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 899 | 1.7 | 0.0 | 1 | 0 | 1456 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 1107 | 22.9 | 0.0 | 1 | 0 | 1409 | 0.3 | 0.0 | 1 |

Table A.39: Performance of ILP-O and ILP-S for instance size (Sz.) 20, instance numbers (Nr.) 15–29.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | **Gap** | **Opt** | $C_a$ | $C_u$ | **t[s]** | **Gap** | **Opt** |
| 20 | 15 | 0.10 | 0 | 318 | 0.1 | 0.0 | 1 | 0 | 847 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 525 | 0.3 | 0.0 | 1 | 0 | 1212 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 800 | 0.8 | 0.0 | 1 | 0 | 1521 | 0.4 | 0.0 | 1 |
| | | 1.00 | 0 | 919 | 1.0 | 0.0 | 1 | 0 | 1551 | 0.6 | 0.0 | 1 |
| | 16 | 0.10 | 0 | 611 | 0.0 | 0.0 | 1 | 0 | 1062 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 866 | 0.5 | 0.0 | 1 | 0 | 1499 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1094 | 1.2 | 0.0 | 1 | 0 | 1499 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 1186 | 2.4 | 0.0 | 1 | 0 | 1559 | 0.3 | 0.0 | 1 |
| | 17 | 0.10 | 0 | 453 | 0.0 | 0.0 | 1 | 0 | 1113 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 923 | 0.2 | 0.0 | 1 | 0 | 1235 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1076 | 0.7 | 0.0 | 1 | 0 | 1546 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 1164 | 1.9 | 0.0 | 1 | 0 | 1656 | 0.2 | 0.0 | 1 |
| | 18 | 0.10 | 0 | 439 | 0.0 | 0.0 | 1 | 0 | 820 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 783 | 0.6 | 0.0 | 1 | 0 | 934 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 906 | 3.7 | 0.0 | 1 | 0 | 1125 | 0.1 | 0.0 | 1 |
| | | 1.00 | 0 | 1030 | 5.9 | 0.0 | 1 | 0 | 1339 | 0.2 | 0.0 | 1 |
| | 19 | 0.10 | 0 | 579 | 0.1 | 0.0 | 1 | 0 | 1030 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 915 | 3.9 | 0.0 | 1 | 0 | 1520 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1174 | 103.8 | 0.0 | 1 | 0 | 1595 | 0.8 | 0.0 | 1 |
| | | 1.00 | 0 | 1256 | 644.4 | 0.0 | 1 | 0 | 1557 | 1.1 | 0.0 | 1 |
| | 20 | 0.10 | 0 | 371 | 0.0 | 0.0 | 1 | 0 | 565 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 598 | 0.3 | 0.0 | 1 | 0 | 1233 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 715 | 1.1 | 0.0 | 1 | 0 | 1293 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 860 | 26.4 | 0.0 | 1 | 0 | 1481 | 0.2 | 0.0 | 1 |
| | 21 | 0.10 | 0 | 421 | 0.0 | 0.0 | 1 | 0 | 813 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 675 | 1.8 | 0.0 | 1 | 0 | 1257 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 882 | 14.8 | 0.0 | 1 | 0 | 1420 | 0.4 | 0.0 | 1 |
| | | 1.00 | 0 | 950 | 94.9 | 0.0 | 1 | 0 | 1420 | 0.5 | 0.0 | 1 |
| | 22 | 0.10 | 0 | 524 | 0.0 | 0.0 | 1 | 0 | 995 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 669 | 0.2 | 0.0 | 1 | 0 | 912 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 802 | 0.4 | 0.0 | 1 | 0 | 1333 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 887 | 1.0 | 0.0 | 1 | 0 | 1168 | 0.2 | 0.0 | 1 |
| | 23 | 0.10 | 0 | 499 | 0.1 | 0.0 | 1 | 0 | 1078 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 727 | 0.8 | 0.0 | 1 | 0 | 1530 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 892 | 39.4 | 0.0 | 1 | 0 | 1477 | 0.5 | 0.0 | 1 |
| | | 1.00 | 0 | 937 | 8.7 | 0.0 | 1 | 0 | 1549 | 0.5 | 0.0 | 1 |
| | 24 | 0.10 | 0 | 620 | 0.1 | 0.0 | 1 | 0 | 1214 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 910 | 0.3 | 0.0 | 1 | 0 | 1497 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 965 | 0.6 | 0.0 | 1 | 0 | 1505 | 0.3 | 0.0 | 1 |
| | | 1.00 | 0 | 1131 | 2.2 | 0.0 | 1 | 0 | 1505 | 0.3 | 0.0 | 1 |
| | 25 | 0.10 | 0 | 432 | 0.0 | 0.0 | 1 | 0 | 1101 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 809 | 0.2 | 0.0 | 1 | 0 | 1076 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 993 | 1.3 | 0.0 | 1 | 0 | 1371 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 1043 | 1.7 | 0.0 | 1 | 0 | 1452 | 0.3 | 0.0 | 1 |
| | 26 | 0.10 | 0 | 369 | 0.0 | 0.0 | 1 | 0 | 813 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 811 | 0.3 | 0.0 | 1 | 0 | 1083 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 939 | 0.6 | 0.0 | 1 | 0 | 1129 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 1000 | 0.7 | 0.0 | 1 | 0 | 1389 | 0.2 | 0.0 | 1 |
| | 27 | 0.10 | 0 | 607 | 0.1 | 0.0 | 1 | 0 | 1018 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 807 | 0.3 | 0.0 | 1 | 0 | 1279 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1028 | 1.0 | 0.0 | 1 | 0 | 1608 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 1109 | 1.2 | 0.0 | 1 | 0 | 1497 | 0.5 | 0.0 | 1 |
| | 28 | 0.10 | 0 | 531 | 0.1 | 0.0 | 1 | 0 | 941 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 928 | 1.4 | 0.0 | 1 | 0 | 1308 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1042 | 2.0 | 0.0 | 1 | 0 | 1524 | 0.2 | 0.0 | 1 |
| | | 1.00 | 0 | 1084 | 22.4 | 0.0 | 1 | 0 | 1524 | 0.3 | 0.0 | 1 |
| | 29 | 0.10 | 0 | 368 | 0.0 | 0.0 | 1 | 0 | 908 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 574 | 0.4 | 0.0 | 1 | 0 | 1210 | 0.1 | 0.0 | 1 |
| | | 0.80 | 0 | 836 | 32.2 | 0.0 | 1 | 0 | 1395 | 0.3 | 0.0 | 1 |
| | | 1.00 | 0 | 1028 | 44.1 | 0.0 | 1 | 0 | 1486 | 0.6 | 0.0 | 1 |

Table A.40: Performance of ILP-O and ILP-S for instance size (Sz.) 30, instance numbers (Nr.) 0–14.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 30 | 0 | 0.10 | 0 | 606 | 0.1 | 0.0 | 1 | 0 | 1127 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1016 | 1.0 | 0.0 | 1 | 0 | 1497 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1208 | 5.0 | 0.0 | 1 | 0 | 2022 | 0.4 | 0.0 | 1 |
| | | 1.00 | 0 | 1300 | 97.4 | 0.0 | 1 | 0 | 2059 | 0.7 | 0.0 | 1 |
| | 1 | 0.10 | 0 | 649 | 0.5 | 0.0 | 1 | 0 | 1684 | 0.2 | 0.0 | 1 |
| | | 0.50 | 0 | 1047 | 2714.2 | 0.0 | 1 | 0 | 2330 | 1.4 | 0.0 | 1 |
| | | 0.80 | 0 | 1298 | 2165.2 | 0.0 | 1 | 0 | 2416 | 3.5 | 0.0 | 1 |
| | | 1.00 | 0 | 1401 | 9998.5 | 2.5 | 0 | 0 | 2570 | 5.4 | 0.0 | 1 |
| | 2 | 0.10 | 0 | 826 | 0.2 | 0.0 | 1 | 0 | 1752 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1365 | 18.7 | 0.0 | 1 | 0 | 2459 | 1.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1464 | 11.7 | 0.0 | 1 | 0 | 2706 | 2.0 | 0.0 | 1 |
| | | 1.00 | 0 | 1635 | 4737.1 | 0.0 | 1 | 0 | 2730 | 3.5 | 0.0 | 1 |
| | 3 | 0.10 | 0 | 679 | 0.1 | 0.0 | 1 | 0 | 1165 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 1083 | 2.9 | 0.0 | 1 | 0 | 1756 | 0.3 | 0.0 | 1 |
| | | 0.80 | 0 | 1118 | 4.8 | 0.0 | 1 | 0 | 2278 | 0.8 | 0.0 | 1 |
| | | 1.00 | 0 | 1526 | 1119.9 | 0.0 | 1 | 0 | 2448 | 0.9 | 0.0 | 1 |
| | 4 | 0.10 | 0 | 701 | 0.1 | 0.0 | 1 | 0 | 1516 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1213 | 4.7 | 0.0 | 1 | 0 | 1770 | 0.3 | 0.0 | 1 |
| | | 0.80 | 0 | 1267 | 6.5 | 0.0 | 1 | 0 | 2185 | 0.5 | 0.0 | 1 |
| | | 1.00 | 0 | 1486 | 18.1 | 0.0 | 1 | 0 | 2359 | 0.8 | 0.0 | 1 |
| | 5 | 0.10 | 0 | 606 | 0.1 | 0.0 | 1 | 0 | 1176 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 1043 | 4.1 | 0.0 | 1 | 0 | 1812 | 0.3 | 0.0 | 1 |
| | | 0.80 | 0 | 1209 | 510.5 | 0.0 | 1 | 0 | 1908 | 0.4 | 0.0 | 1 |
| | | 1.00 | 0 | 1309 | 50.2 | 0.0 | 1 | 0 | 2001 | 0.8 | 0.0 | 1 |
| | 6 | 0.10 | 0 | 617 | 1.3 | 0.0 | 1 | 0 | 1306 | 0.2 | 0.0 | 1 |
| | | 0.50 | 0 | 1079 | 5.1 | 0.0 | 1 | 0 | 2138 | 0.9 | 0.0 | 1 |
| | | 0.80 | 0 | 1207 | 9998.3 | 0.8 | 0 | 0 | 2296 | 2.5 | 0.0 | 1 |
| | | 1.00 | 0 | 1339 | 9998.4 | 7.0 | 0 | 0 | 2296 | 4.8 | 0.0 | 1 |
| | 7 | 0.10 | 0 | 671 | 0.1 | 0.0 | 1 | 0 | 1138 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1175 | 1.1 | 0.0 | 1 | 0 | 1814 | 0.7 | 0.0 | 1 |
| | | 0.80 | 0 | 1339 | 9.6 | 0.0 | 1 | 0 | 2015 | 0.9 | 0.0 | 1 |
| | | 1.00 | 0 | 1395 | 36.2 | 0.0 | 1 | 0 | 2013 | 1.7 | 0.0 | 1 |
| | 8 | 0.10 | 0 | 1111 | 0.1 | 0.0 | 1 | 0 | 1620 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1267 | 4.6 | 0.0 | 1 | 0 | 1978 | 0.5 | 0.0 | 1 |
| | | 0.80 | 0 | 1475 | 2.9 | 0.0 | 1 | 0 | 2214 | 0.6 | 0.0 | 1 |
| | | 1.00 | 0 | 1549 | 5.3 | 0.0 | 1 | 0 | 2257 | 0.9 | 0.0 | 1 |
| | 9 | 0.10 | 0 | 571 | 0.1 | 0.0 | 1 | 0 | 1503 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1094 | 4.9 | 0.0 | 1 | 0 | 2308 | 0.8 | 0.0 | 1 |
| | | 0.80 | 0 | 1374 | 243.8 | 0.0 | 1 | 0 | 2554 | 1.6 | 0.0 | 1 |
| | | 1.00 | 0 | 1530 | 9998.3 | 1.4 | 0 | 0 | 2474 | 2.6 | 0.0 | 1 |
| | 10 | 0.10 | 0 | 593 | 0.1 | 0.0 | 1 | 0 | 1068 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 997 | 0.3 | 0.0 | 1 | 0 | 1663 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1139 | 1.3 | 0.0 | 1 | 0 | 1987 | 0.4 | 0.0 | 1 |
| | | 1.00 | 0 | 1307 | 10.5 | 0.0 | 1 | 0 | 2168 | 0.4 | 0.0 | 1 |
| | 11 | 0.10 | 0 | 597 | 0.1 | 0.0 | 1 | 0 | 1606 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1109 | 0.6 | 0.0 | 1 | 0 | 2043 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1396 | 48.8 | 0.0 | 1 | 0 | 2200 | 0.5 | 0.0 | 1 |
| | | 1.00 | 0 | 1626 | 836.2 | 0.0 | 1 | 0 | 2278 | 0.9 | 0.0 | 1 |
| | 12 | 0.10 | 0 | 754 | 0.1 | 0.0 | 1 | 0 | 1147 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 1049 | 0.3 | 0.0 | 1 | 0 | 1551 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1311 | 1.0 | 0.0 | 1 | 0 | 2216 | 0.3 | 0.0 | 1 |
| | | 1.00 | 0 | 1546 | 1.3 | 0.0 | 1 | 0 | 2243 | 0.7 | 0.0 | 1 |
| | 13 | 0.10 | 0 | 542 | 0.1 | 0.0 | 1 | 0 | 1464 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1061 | 1.1 | 0.0 | 1 | 0 | 1972 | 0.5 | 0.0 | 1 |
| | | 0.80 | 0 | 1380 | 5.7 | 0.0 | 1 | 0 | 2209 | 0.5 | 0.0 | 1 |
| | | 1.00 | 0 | 1405 | 9.7 | 0.0 | 1 | 0 | 2205 | 0.9 | 0.0 | 1 |
| | 14 | 0.10 | 0 | 749 | 0.1 | 0.0 | 1 | 0 | 1660 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1282 | 3.4 | 0.0 | 1 | 0 | 1958 | 0.5 | 0.0 | 1 |
| | | 0.80 | 0 | 1323 | 1.6 | 0.0 | 1 | 0 | 2356 | 1.1 | 0.0 | 1 |
| | | 1.00 | 0 | 1488 | 85.6 | 0.0 | 1 | 0 | 2527 | 1.5 | 0.0 | 1 |

Table A.41: Performance of ILP-O and ILP-S for instance size (Sz.) 30, instance numbers (Nr.) 15–29.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $C_a$ | $C_u$ | **t[s]** | **Gap** | **Opt** | $C_a$ | $C_u$ | **t[s]** | **Gap** | **Opt** |
| 30 | 15 | 0.10 | 0 | 977 | 0.1 | 0.0 | 1 | 0 | 1581 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1270 | 0.6 | 0.0 | 1 | 0 | 2033 | 0.3 | 0.0 | 1 |
| | | 0.80 | 0 | 1406 | 23.3 | 0.0 | 1 | 0 | 1971 | 0.8 | 0.0 | 1 |
| | | 1.00 | 0 | 1590 | 51.7 | 0.0 | 1 | 0 | 2148 | 1.3 | 0.0 | 1 |
| | 16 | 0.10 | 0 | 458 | 0.1 | 0.0 | 1 | 0 | 1208 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 896 | 1.8 | 0.0 | 1 | 0 | 1842 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 971 | 3.2 | 0.0 | 1 | 0 | 1894 | 0.4 | 0.0 | 1 |
| | | 1.00 | 0 | 1202 | 36.2 | 0.0 | 1 | 0 | 1959 | 0.8 | 0.0 | 1 |
| | 17 | 0.10 | 0 | 776 | 0.2 | 0.0 | 1 | 0 | 1326 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1043 | 2.9 | 0.0 | 1 | 0 | 1797 | 0.6 | 0.0 | 1 |
| | | 0.80 | 0 | 1313 | 11.7 | 0.0 | 1 | 0 | 2036 | 0.7 | 0.0 | 1 |
| | | 1.00 | 0 | 1366 | 127.2 | 0.0 | 1 | 0 | 2195 | 1.4 | 0.0 | 1 |
| | 18 | 0.10 | 0 | 561 | 0.1 | 0.0 | 1 | 0 | 1035 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1192 | 0.5 | 0.0 | 1 | 0 | 1969 | 0.3 | 0.0 | 1 |
| | | 0.80 | 0 | 1473 | 1.2 | 0.0 | 1 | 0 | 2308 | 0.4 | 0.0 | 1 |
| | | 1.00 | 0 | 1669 | 13.5 | 0.0 | 1 | 0 | 2498 | 0.8 | 0.0 | 1 |
| | 19 | 0.10 | 0 | 498 | 2.4 | 0.0 | 1 | 0 | 1284 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1118 | 111.5 | 0.0 | 1 | 0 | 2514 | 1.5 | 0.0 | 1 |
| | | 0.80 | 0 | 1287 | 586.1 | 0.0 | 1 | 0 | 2443 | 3.7 | 0.0 | 1 |
| | | 1.00 | 0 | 1465 | 9998.5 | 4.8 | 0 | 0 | 2483 | 8.6 | 0.0 | 1 |
| | 20 | 0.10 | 0 | 803 | 0.1 | 0.0 | 1 | 0 | 1184 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1317 | 2.9 | 0.0 | 1 | 0 | 2015 | 0.8 | 0.0 | 1 |
| | | 0.80 | 0 | 1539 | 182.7 | 0.0 | 1 | 0 | 2275 | 2.2 | 0.0 | 1 |
| | | 1.00 | 0 | 1658 | 8166.4 | 0.0 | 1 | 0 | 2301 | 2.7 | 0.0 | 1 |
| | 21 | 0.10 | 0 | 581 | 0.1 | 0.0 | 1 | 0 | 1246 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 916 | 1.5 | 0.0 | 1 | 0 | 1616 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1169 | 2.9 | 0.0 | 1 | 0 | 1926 | 0.3 | 0.0 | 1 |
| | | 1.00 | 0 | 1273 | 30.7 | 0.0 | 1 | 0 | 1926 | 0.4 | 0.0 | 1 |
| | 22 | 0.10 | 0 | 881 | 0.1 | 0.0 | 1 | 0 | 1152 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 1312 | 0.4 | 0.0 | 1 | 0 | 1753 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1395 | 0.7 | 0.0 | 1 | 0 | 2428 | 0.3 | 0.0 | 1 |
| | | 1.00 | 0 | 1417 | 1.1 | 0.0 | 1 | 0 | 2453 | 0.4 | 0.0 | 1 |
| | 23 | 0.10 | 0 | 738 | 0.1 | 0.0 | 1 | 0 | 940 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1178 | 0.7 | 0.0 | 1 | 0 | 1731 | 0.5 | 0.0 | 1 |
| | | 0.80 | 0 | 1332 | 23.0 | 0.0 | 1 | 0 | 2303 | 0.8 | 0.0 | 1 |
| | | 1.00 | 0 | 1497 | 135.7 | 0.0 | 1 | 0 | 2182 | 1.0 | 0.0 | 1 |
| | 24 | 0.10 | 0 | 525 | 0.1 | 0.0 | 1 | 0 | 991 | 0.2 | 0.0 | 1 |
| | | 0.50 | 0 | 1202 | 3.1 | 0.0 | 1 | 0 | 2294 | 1.7 | 0.0 | 1 |
| | | 0.80 | 0 | 1339 | 32.9 | 0.0 | 1 | 0 | 2709 | 2.3 | 0.0 | 1 |
| | | 1.00 | 0 | 1371 | 866.2 | 0.0 | 1 | 0 | 2810 | 4.3 | 0.0 | 1 |
| | 25 | 0.10 | 0 | 849 | 0.1 | 0.0 | 1 | 0 | 1714 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1283 | 2.6 | 0.0 | 1 | 0 | 1850 | 0.6 | 0.0 | 1 |
| | | 0.80 | 0 | 1462 | 8.0 | 0.0 | 1 | 0 | 2132 | 0.8 | 0.0 | 1 |
| | | 1.00 | 0 | 1620 | 116.7 | 0.0 | 1 | 0 | 2328 | 2.7 | 0.0 | 1 |
| | 26 | 0.10 | 0 | 518 | 0.1 | 0.0 | 1 | 0 | 835 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 981 | 1.3 | 0.0 | 1 | 0 | 2028 | 0.3 | 0.0 | 1 |
| | | 0.80 | 0 | 1165 | 6.6 | 0.0 | 1 | 0 | 2288 | 0.5 | 0.0 | 1 |
| | | 1.00 | 0 | 1365 | 147.0 | 0.0 | 1 | 0 | 2293 | 0.8 | 0.0 | 1 |
| | 27 | 0.10 | 0 | 549 | 0.1 | 0.0 | 1 | 0 | 980 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 1013 | 0.5 | 0.0 | 1 | 0 | 1642 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1262 | 35.4 | 0.0 | 1 | 0 | 2137 | 0.7 | 0.0 | 1 |
| | | 1.00 | 0 | 1422 | 187.4 | 0.0 | 1 | 0 | 2251 | 0.9 | 0.0 | 1 |
| | 28 | 0.10 | 0 | 939 | 0.1 | 0.0 | 1 | 0 | 1505 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1232 | 0.6 | 0.0 | 1 | 0 | 1736 | 0.3 | 0.0 | 1 |
| | | 0.80 | 0 | 1385 | 4.5 | 0.0 | 1 | 0 | 2308 | 0.4 | 0.0 | 1 |
| | | 1.00 | 0 | 1448 | 8.2 | 0.0 | 1 | 0 | 2338 | 0.9 | 0.0 | 1 |
| | 29 | 0.10 | 0 | 411 | 0.1 | 0.0 | 1 | 0 | 1069 | 0.0 | 0.0 | 1 |
| | | 0.50 | 0 | 940 | 0.9 | 0.0 | 1 | 0 | 1718 | 0.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1278 | 45.2 | 0.0 | 1 | 0 | 1947 | 0.4 | 0.0 | 1 |
| | | 1.00 | 0 | 1354 | 86.3 | 0.0 | 1 | 0 | 2067 | 0.5 | 0.0 | 1 |

Table A.42: Performance of ILP-O and ILP-S for instance size (Sz.) 50, instance numbers (Nr.) 0–14.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 50 | 0 | 0.10 | 0 | 974 | 0.6 | 0.0 | 1 | 0 | 1731 | 0.4 | 0.0 | 1 |
| | | 0.50 | 0 | 1732 | 97.7 | 0.0 | 1 | 0 | 3859 | 2.5 | 0.0 | 1 |
| | | 0.80 | 0 | 1991 | 5787.4 | 0.0 | 1 | 0 | 4019 | 6.3 | 0.0 | 1 |
| | | 1.00 | 0 | 2131 | 9999.0 | 3.2 | 0 | 0 | 4378 | 7.0 | 0.0 | 1 |
| | 1 | 0.10 | 0 | 1322 | 0.5 | 0.0 | 1 | 0 | 2103 | 0.5 | 0.0 | 1 |
| | | 0.50 | 0 | 1612 | 25.2 | 0.0 | 1 | 0 | 3087 | 2.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1837 | 9998.8 | 0.7 | 0 | 0 | 3754 | 4.1 | 0.0 | 1 |
| | | 1.00 | 0 | 2162 | 9999.0 | 4.8 | 0 | 0 | 3868 | 11.0 | 0.0 | 1 |
| | 2 | 0.10 | 0 | 719 | 0.2 | 0.0 | 1 | 0 | 1537 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1776 | 2.5 | 0.0 | 1 | 0 | 3032 | 1.2 | 0.0 | 1 |
| | | 0.80 | 0 | 2124 | 49.4 | 0.0 | 1 | 0 | 3686 | 1.1 | 0.0 | 1 |
| | | 1.00 | 0 | 2370 | 398.5 | 0.0 | 1 | 0 | 3873 | 2.2 | 0.0 | 1 |
| | 3 | 0.10 | 0 | 1118 | 0.2 | 0.0 | 1 | 0 | 2028 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1779 | 3.0 | 0.0 | 1 | 0 | 3193 | 1.0 | 0.0 | 1 |
| | | 0.80 | 0 | 1972 | 11.4 | 0.0 | 1 | 0 | 3666 | 1.7 | 0.0 | 1 |
| | | 1.00 | 0 | 2022 | 16.4 | 0.0 | 1 | 0 | 4193 | 2.8 | 0.0 | 1 |
| | 4 | 0.10 | 0 | 1182 | 0.3 | 0.0 | 1 | 0 | 1763 | 0.2 | 0.0 | 1 |
| | | 0.50 | 0 | 1870 | 7.9 | 0.0 | 1 | 0 | 3033 | 1.4 | 0.0 | 1 |
| | | 0.80 | 0 | 1969 | 66.4 | 0.0 | 1 | 0 | 3606 | 2.6 | 0.0 | 1 |
| | | 1.00 | 0 | 2203 | 290.0 | 0.0 | 1 | 0 | 3678 | 3.2 | 0.0 | 1 |
| | 5 | 0.10 | 0 | 1238 | 0.3 | 0.0 | 1 | 0 | 2190 | 0.2 | 0.0 | 1 |
| | | 0.50 | 0 | 1571 | 11.7 | 0.0 | 1 | 0 | 2892 | 1.4 | 0.0 | 1 |
| | | 0.80 | 0 | 1866 | 20.5 | 0.0 | 1 | 0 | 3632 | 2.6 | 0.0 | 1 |
| | | 1.00 | 0 | 1905 | 931.3 | 0.0 | 1 | 0 | 3602 | 3.9 | 0.0 | 1 |
| | 6 | 0.10 | 0 | 787 | 0.2 | 0.0 | 1 | 0 | 1213 | 0.2 | 0.0 | 1 |
| | | 0.50 | 0 | 1657 | 2.1 | 0.0 | 1 | 0 | 2726 | 1.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1884 | 6.9 | 0.0 | 1 | 0 | 3425 | 2.1 | 0.0 | 1 |
| | | 1.00 | 0 | 2214 | 258.2 | 0.0 | 1 | 0 | 3463 | 2.6 | 0.0 | 1 |
| | 7 | 0.10 | 0 | 1116 | 0.2 | 0.0 | 1 | 0 | 1939 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1948 | 2.1 | 0.0 | 1 | 0 | 3577 | 1.2 | 0.0 | 1 |
| | | 0.80 | 0 | 2161 | 12.2 | 0.0 | 1 | 0 | 3485 | 2.2 | 0.0 | 1 |
| | | 1.00 | 0 | 2370 | 185.7 | 0.0 | 1 | 0 | 3987 | 3.2 | 0.0 | 1 |
| | 8 | 0.10 | 0 | 780 | 2.5 | 0.0 | 1 | 0 | 2015 | 0.5 | 0.0 | 1 |
| | | 0.50 | 0 | 1420 | 506.3 | 0.0 | 1 | 0 | 4054 | 4.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1720 | 9998.9 | 1.6 | 0 | 0 | 4377 | 10.9 | 0.0 | 1 |
| | | 1.00 | 0 | 2201 | 9999.1 | 8.1 | 0 | 0 | 4605 | 20.5 | 0.0 | 1 |
| | 9 | 0.10 | 0 | 1371 | 0.2 | 0.0 | 1 | 0 | 2305 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1709 | 1.3 | 0.0 | 1 | 0 | 2268 | 0.8 | 0.0 | 1 |
| | | 0.80 | 0 | 1873 | 116.3 | 0.0 | 1 | 0 | 2866 | 2.6 | 0.0 | 1 |
| | | 1.00 | 0 | 2117 | 10.2 | 0.0 | 1 | 0 | 3380 | 2.3 | 0.0 | 1 |
| | 10 | 0.10 | 0 | 973 | 0.2 | 0.0 | 1 | 0 | 1277 | 0.3 | 0.0 | 1 |
| | | 0.50 | 0 | 1765 | 129.0 | 0.0 | 1 | 0 | 3811 | 2.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1981 | 2813.1 | 0.0 | 1 | 0 | 4165 | 3.6 | 0.0 | 1 |
| | | 1.00 | 0 | 2141 | 9998.9 | 1.7 | 0 | 0 | 4325 | 4.5 | 0.0 | 1 |
| | 11 | 0.10 | 0 | 886 | 0.1 | 0.0 | 1 | 0 | 2184 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1534 | 2.0 | 0.0 | 1 | 0 | 3384 | 1.0 | 0.0 | 1 |
| | | 0.80 | 0 | 1935 | 9998.6 | 0.3 | 0 | 0 | 3685 | 1.9 | 0.0 | 1 |
| | | 1.00 | 0 | 2388 | 9998.7 | 2.0 | 0 | 0 | 3809 | 4.7 | 0.0 | 1 |
| | 12 | 0.10 | 0 | 872 | 0.3 | 0.0 | 1 | 0 | 1560 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1706 | 70.0 | 0.0 | 1 | 0 | 3246 | 1.5 | 0.0 | 1 |
| | | 0.80 | 0 | 1954 | 9998.6 | 0.4 | 0 | 0 | 3491 | 1.7 | 0.0 | 1 |
| | | 1.00 | 0 | 2089 | 9998.8 | 1.2 | 0 | 0 | 3640 | 3.5 | 0.0 | 1 |
| | 13 | 0.10 | 0 | 1081 | 0.2 | 0.0 | 1 | 0 | 1987 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1657 | 1.5 | 0.0 | 1 | 0 | 2962 | 1.1 | 0.0 | 1 |
| | | 0.80 | 0 | 1982 | 51.8 | 0.0 | 1 | 0 | 3303 | 1.8 | 0.0 | 1 |
| | | 1.00 | 0 | 2304 | 217.6 | 0.0 | 1 | 0 | 3778 | 2.8 | 0.0 | 1 |
| | 14 | 0.10 | 0 | 1057 | 0.2 | 0.0 | 1 | 0 | 1461 | 0.2 | 0.0 | 1 |
| | | 0.50 | 0 | 1736 | 1.8 | 0.0 | 1 | 0 | 2402 | 0.9 | 0.0 | 1 |
| | | 0.80 | 0 | 1925 | 37.4 | 0.0 | 1 | 0 | 3201 | 1.7 | 0.0 | 1 |
| | | 1.00 | 0 | 1962 | 8.2 | 0.0 | 1 | 0 | 3313 | 2.4 | 0.0 | 1 |

Table A.43: Performance of ILP-O and ILP-S for instance size (Sz.) 50, instance numbers (Nr.) 15–29.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 50 | 15 | 0.10 | 0 | 1132 | 0.3 | 0.0 | 1 | 0 | 1767 | 0.2 | 0.0 | 1 |
| | | 0.50 | 0 | 1865 | 56.2 | 0.0 | 1 | 0 | 3401 | 1.6 | 0.0 | 1 |
| | | 0.80 | 0 | 2010 | 129.1 | 0.0 | 1 | 0 | 3627 | 2.9 | 0.0 | 1 |
| | | 1.00 | 0 | 2358 | 6428.8 | 0.0 | 1 | 0 | 3987 | 3.3 | 0.0 | 1 |
| | 16 | 0.10 | 0 | 953 | 0.7 | 0.0 | 1 | 0 | 1899 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1456 | 2.2 | 0.0 | 1 | 0 | 3596 | 1.8 | 0.0 | 1 |
| | | 0.80 | 0 | 1860 | 27.7 | 0.0 | 1 | 0 | 3795 | 3.2 | 0.0 | 1 |
| | | 1.00 | 0 | 2038 | 2273.3 | 0.0 | 1 | 0 | 3939 | 3.5 | 0.0 | 1 |
| | 17 | 0.10 | 0 | 1103 | 0.2 | 0.0 | 1 | 0 | 1546 | 0.2 | 0.0 | 1 |
| | | 0.50 | 0 | 1963 | 2.6 | 0.0 | 1 | 0 | 3164 | 1.3 | 0.0 | 1 |
| | | 0.80 | 0 | 2185 | 120.4 | 0.0 | 1 | 0 | 3541 | 2.2 | 0.0 | 1 |
| | | 1.00 | 0 | 2436 | 3794.3 | 0.0 | 1 | 0 | 3828 | 3.8 | 0.0 | 1 |
| | 18 | 0.10 | 0 | 839 | 0.2 | 0.0 | 1 | 0 | 1773 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1725 | 2.6 | 0.0 | 1 | 0 | 2818 | 1.2 | 0.0 | 1 |
| | | 0.80 | 0 | 2222 | 4.3 | 0.0 | 1 | 0 | 3736 | 2.1 | 0.0 | 1 |
| | | 1.00 | 0 | 2418 | 818.4 | 0.0 | 1 | 0 | 4052 | 3.2 | 0.0 | 1 |
| | 19 | 0.10 | 0 | 949 | 0.2 | 0.0 | 1 | 0 | 1720 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1462 | 46.1 | 0.0 | 1 | 0 | 3128 | 1.3 | 0.0 | 1 |
| | | 0.80 | 0 | 1653 | 95.5 | 0.0 | 1 | 0 | 3376 | 2.6 | 0.0 | 1 |
| | | 1.00 | 0 | 1995 | 9998.8 | 0.6 | 0 | 0 | 3563 | 2.7 | 0.0 | 1 |
| | 20 | 0.10 | 0 | 1626 | 0.3 | 0.0 | 1 | 0 | 2865 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1965 | 3.1 | 0.0 | 1 | 0 | 3258 | 0.6 | 0.0 | 1 |
| | | 0.80 | 0 | 2249 | 53.5 | 0.0 | 1 | 0 | 3567 | 2.4 | 0.0 | 1 |
| | | 1.00 | 0 | 2647 | 6524.8 | 0.0 | 1 | 0 | 3878 | 3.1 | 0.0 | 1 |
| | 21 | 0.10 | 0 | 1208 | 0.7 | 0.0 | 1 | 0 | 2479 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1703 | 144.0 | 0.0 | 1 | 0 | 3157 | 1.4 | 0.0 | 1 |
| | | 0.80 | 0 | 2137 | 7210.6 | 0.0 | 1 | 0 | 4026 | 2.6 | 0.0 | 1 |
| | | 1.00 | 0 | 2467 | 9998.7 | 2.0 | 0 | 0 | 4115 | 4.0 | 0.0 | 1 |
| | 22 | 0.10 | 0 | 1305 | 0.9 | 0.0 | 1 | 0 | 2192 | 0.4 | 0.0 | 1 |
| | | 0.50 | 0 | 2020 | 11.5 | 0.0 | 1 | 0 | 3755 | 1.5 | 0.0 | 1 |
| | | 0.80 | 0 | 2207 | 116.5 | 0.0 | 1 | 0 | 3949 | 3.8 | 0.0 | 1 |
| | | 1.00 | 0 | 2445 | 9998.9 | 0.7 | 0 | 0 | 4300 | 6.2 | 0.0 | 1 |
| | 23 | 0.10 | 0 | 986 | 0.1 | 0.0 | 1 | 0 | 1960 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1985 | 11.5 | 0.0 | 1 | 0 | 2993 | 1.0 | 0.0 | 1 |
| | | 0.80 | 0 | 2239 | 162.5 | 0.0 | 1 | 0 | 3623 | 1.6 | 0.0 | 1 |
| | | 1.00 | 0 | 2361 | 591.2 | 0.0 | 1 | 0 | 3849 | 1.8 | 0.0 | 1 |
| | 24 | 0.10 | 0 | 784 | 0.3 | 0.0 | 1 | 0 | 2076 | 0.3 | 0.0 | 1 |
| | | 0.50 | 0 | 1412 | 31.8 | 0.0 | 1 | 0 | 3394 | 2.5 | 0.0 | 1 |
| | | 0.80 | 0 | 2072 | 9998.7 | 3.8 | 0 | 0 | 4222 | 6.6 | 0.0 | 1 |
| | | 1.00 | 0 | 2430 | 9998.8 | 5.5 | 0 | 0 | 4319 | 6.7 | 0.0 | 1 |
| | 25 | 0.10 | 0 | 823 | 0.2 | 0.0 | 1 | 0 | 1652 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1463 | 2.4 | 0.0 | 1 | 0 | 2674 | 1.2 | 0.0 | 1 |
| | | 0.80 | 0 | 1859 | 6.9 | 0.0 | 1 | 0 | 3197 | 1.9 | 0.0 | 1 |
| | | 1.00 | 0 | 2011 | 114.2 | 0.0 | 1 | 0 | 3336 | 2.2 | 0.0 | 1 |
| | 26 | 0.10 | 0 | 920 | 0.2 | 0.0 | 1 | 0 | 1553 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1575 | 2.5 | 0.0 | 1 | 0 | 2857 | 0.7 | 0.0 | 1 |
| | | 0.80 | 0 | 1994 | 314.1 | 0.0 | 1 | 0 | 3641 | 1.9 | 0.0 | 1 |
| | | 1.00 | 0 | 2147 | 998.5 | 0.0 | 1 | 0 | 3778 | 3.0 | 0.0 | 1 |
| | 27 | 0.10 | 0 | 1309 | 0.1 | 0.0 | 1 | 0 | 2185 | 0.1 | 0.0 | 1 |
| | | 0.50 | 0 | 1984 | 1.7 | 0.0 | 1 | 0 | 3296 | 0.6 | 0.0 | 1 |
| | | 0.80 | 0 | 2094 | 3.1 | 0.0 | 1 | 0 | 3020 | 1.3 | 0.0 | 1 |
| | | 1.00 | 0 | 2282 | 8.8 | 0.0 | 1 | 0 | 3464 | 2.0 | 0.0 | 1 |
| | 28 | 0.10 | 0 | 1082 | 1.7 | 0.0 | 1 | 0 | 3313 | 0.8 | 0.0 | 1 |
| | | 0.50 | 0 | 1976 | 9998.7 | 3.9 | 0 | 0 | 4406 | 10.9 | 0.0 | 1 |
| | | 0.80 | 0 | 2449 | 9999.1 | 8.1 | 0 | 0 | 4736 | 22.7 | 0.0 | 1 |
| | | 1.00 | 0 | 2676 | 9999.3 | 11.6 | 0 | 0 | 4716 | 28.0 | 0.0 | 1 |
| | 29 | 0.10 | 0 | 620 | 0.2 | 0.0 | 1 | 0 | 1741 | 0.2 | 0.0 | 1 |
| | | 0.50 | 0 | 1658 | 6.5 | 0.0 | 1 | 0 | 3000 | 1.9 | 0.0 | 1 |
| | | 0.80 | 0 | 2168 | 71.3 | 0.0 | 1 | 0 | 3441 | 5.0 | 0.0 | 1 |
| | | 1.00 | 0 | 2306 | 3110.4 | 0.0 | 1 | 0 | 4170 | 8.2 | 0.0 | 1 |

Table A.44: Performance of ILP-O and ILP-S for instance size (Sz.) 100, instance numbers (Nr.) 0–14.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 100 | 0 | 0.10 | 0 | 2226 | 0.8 | 0.0 | 1 | 0 | 3211 | 0.6 | 0.0 | 1 |
| | | 0.50 | 0 | 3015 | 10.4 | 0.0 | 1 | 0 | 5501 | 6.2 | 0.0 | 1 |
| | | 0.80 | 0 | 3691 | 10001.4 | 1.5 | 0 | 0 | 6764 | 13.5 | 0.0 | 1 |
| | | 1.00 | 0 | 4444 | 10001.7 | 5.2 | 0 | 0 | 7466 | 20.7 | 0.0 | 1 |
| | 1 | 0.10 | 0 | 2757 | 1.2 | 0.0 | 1 | 0 | 4302 | 1.1 | 0.0 | 1 |
| | | 0.50 | 0 | 3910 | 9999.8 | 0.2 | 0 | 0 | 6785 | 6.9 | 0.0 | 1 |
| | | 0.80 | 0 | 4361 | 10000.9 | 1.8 | 0 | 0 | 7133 | 10.7 | 0.0 | 1 |
| | | 1.00 | 0 | 4770 | 10001.0 | 3.5 | 0 | 0 | 7764 | 16.0 | 0.0 | 1 |
| | 2 | 0.10 | 0 | 1219 | 0.7 | 0.0 | 1 | 0 | 2049 | 0.9 | 0.0 | 1 |
| | | 0.50 | 0 | 2723 | 378.9 | 0.0 | 1 | 0 | 5891 | 4.9 | 0.0 | 1 |
| | | 0.80 | 0 | 3281 | 10000.7 | 2.0 | 0 | 0 | 6833 | 9.0 | 0.0 | 1 |
| | | 1.00 | 0 | 3884 | 10001.4 | 6.6 | 0 | 0 | 7351 | 15.1 | 0.0 | 1 |
| | 3 | 0.10 | 0 | 2185 | 71.3 | 0.0 | 1 | 0 | 4786 | 3.3 | 0.0 | 1 |
| | | 0.50 | 0 | 3595 | 10000.8 | 1.7 | 0 | 0 | 7480 | 29.3 | 0.0 | 1 |
| | | 0.80 | 0 | 6301 | 10002.2 | 39.4 | 0 | 0 | 7918 | 44.9 | 0.0 | 1 |
| | | 1.00 | 1626 | 7346 | 10003.5 | 100.0 | 0 | 0 | 8279 | 113.7 | 0.0 | 1 |
| | 4 | 0.10 | 0 | 1776 | 7.2 | 0.0 | 1 | 0 | 3257 | 1.9 | 0.0 | 1 |
| | | 0.50 | 0 | 3663 | 10000.5 | 2.4 | 0 | 0 | 6941 | 13.1 | 0.0 | 1 |
| | | 0.80 | 0 | 4119 | 10001.0 | 4.9 | 0 | 0 | 7706 | 20.9 | 0.0 | 1 |
| | | 1.00 | 0 | 4398 | 10001.5 | 6.2 | 0 | 0 | 7839 | 29.4 | 0.0 | 1 |
| | 5 | 0.10 | 0 | 1904 | 1.2 | 0.0 | 1 | 0 | 3067 | 1.4 | 0.0 | 1 |
| | | 0.50 | 0 | 3432 | 31.0 | 0.0 | 1 | 0 | 6101 | 6.3 | 0.0 | 1 |
| | | 0.80 | 0 | 3838 | 2059.4 | 0.0 | 1 | 0 | 6736 | 13.8 | 0.0 | 1 |
| | | 1.00 | 0 | 4199 | 10001.8 | 2.6 | 0 | 0 | 7150 | 18.1 | 0.0 | 1 |
| | 6 | 0.10 | 0 | 2250 | 5.5 | 0.0 | 1 | 0 | 3883 | 3.8 | 0.0 | 1 |
| | | 0.50 | 0 | 3497 | 944.7 | 0.0 | 1 | 0 | 5462 | 8.6 | 0.0 | 1 |
| | | 0.80 | 0 | 4133 | 10001.2 | 0.9 | 0 | 0 | 6727 | 21.9 | 0.0 | 1 |
| | | 1.00 | 0 | 4637 | 10001.9 | 3.4 | 0 | 0 | 7980 | 21.7 | 0.0 | 1 |
| | 7 | 0.10 | 0 | 2552 | 1.2 | 0.0 | 1 | 0 | 4132 | 1.1 | 0.0 | 1 |
| | | 0.50 | 0 | 3614 | 30.5 | 0.0 | 1 | 0 | 6073 | 6.0 | 0.0 | 1 |
| | | 0.80 | 0 | 4455 | 10001.3 | 2.6 | 0 | 0 | 7091 | 13.7 | 0.0 | 1 |
| | | 1.00 | 0 | 4782 | 10001.8 | 4.4 | 0 | 0 | 8264 | 17.2 | 0.0 | 1 |
| | 8 | 0.10 | 0 | 1765 | 1.0 | 0.0 | 1 | 0 | 3603 | 0.9 | 0.0 | 1 |
| | | 0.50 | 0 | 3167 | 257.5 | 0.0 | 1 | 0 | 6530 | 4.0 | 0.0 | 1 |
| | | 0.80 | 0 | 3772 | 10000.7 | 1.8 | 0 | 0 | 7266 | 7.2 | 0.0 | 1 |
| | | 1.00 | 0 | 4508 | 10001.6 | 5.0 | 0 | 0 | 7582 | 9.6 | 0.0 | 1 |
| | 9 | 0.10 | 0 | 1990 | 1.3 | 0.0 | 1 | 0 | 2937 | 1.3 | 0.0 | 1 |
| | | 0.50 | 0 | 3278 | 1616.1 | 0.0 | 1 | 0 | 6032 | 4.9 | 0.0 | 1 |
| | | 0.80 | 0 | 3829 | 3107.8 | 0.0 | 1 | 0 | 6647 | 8.6 | 0.0 | 1 |
| | | 1.00 | 0 | 4267 | 10001.5 | 0.8 | 0 | 0 | 7162 | 11.4 | 0.0 | 1 |
| | 10 | 0.10 | 0 | 2143 | 1.2 | 0.0 | 1 | 0 | 3741 | 1.1 | 0.0 | 1 |
| | | 0.50 | 0 | 2726 | 30.7 | 0.0 | 1 | 0 | 5431 | 4.5 | 0.0 | 1 |
| | | 0.80 | 0 | 3751 | 10001.0 | 3.1 | 0 | 0 | 6381 | 6.5 | 0.0 | 1 |
| | | 1.00 | 0 | 4018 | 10001.8 | 3.6 | 0 | 0 | 7016 | 10.9 | 0.0 | 1 |
| | 11 | 0.10 | 0 | 2724 | 5.4 | 0.0 | 1 | 0 | 4904 | 6.1 | 0.0 | 1 |
| | | 0.50 | 0 | 3984 | 6036.7 | 0.0 | 1 | 0 | 7065 | 15.5 | 0.0 | 1 |
| | | 0.80 | 0 | 4675 | 10001.7 | 2.4 | 0 | 0 | 8027 | 30.6 | 0.0 | 1 |
| | | 1.00 | 1960 | 7043 | 10003.0 | 100.0 | 0 | 0 | 8443 | 44.6 | 0.0 | 1 |
| | 12 | 0.10 | 0 | 2192 | 0.6 | 0.0 | 1 | 0 | 4178 | 0.4 | 0.0 | 1 |
| | | 0.50 | 0 | 3276 | 580.7 | 0.0 | 1 | 0 | 6776 | 3.4 | 0.0 | 1 |
| | | 0.80 | 0 | 3841 | 10001.1 | 0.9 | 0 | 0 | 7554 | 7.3 | 0.0 | 1 |
| | | 1.00 | 0 | 4563 | 10002.0 | 3.9 | 0 | 0 | 7780 | 10.0 | 0.0 | 1 |
| | 13 | 0.10 | 0 | 2145 | 1.3 | 0.0 | 1 | 0 | 3207 | 0.6 | 0.0 | 1 |
| | | 0.50 | 0 | 3732 | 10000.0 | 1.3 | 0 | 0 | 6353 | 7.6 | 0.0 | 1 |
| | | 0.80 | 0 | 4440 | 10001.1 | 4.1 | 0 | 0 | 7432 | 13.1 | 0.0 | 1 |
| | | 1.00 | 0 | 4802 | 10001.8 | 4.0 | 0 | 0 | 7880 | 21.9 | 0.0 | 1 |
| | 14 | 0.10 | 0 | 2693 | 3.2 | 0.0 | 1 | 0 | 4245 | 0.6 | 0.0 | 1 |
| | | 0.50 | 0 | 3802 | 668.9 | 0.0 | 1 | 0 | 6572 | 6.1 | 0.0 | 1 |
| | | 0.80 | 0 | 4357 | 10000.9 | 2.6 | 0 | 0 | 7184 | 13.0 | 0.0 | 1 |
| | | 1.00 | 0 | 4639 | 10001.9 | 4.4 | 0 | 0 | 7999 | 26.4 | 0.0 | 1 |

Table A.45: Performance of ILP-O and ILP-S for instance size (Sz.) 100, instance numbers (Nr.) 15–29.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 100 | 15 | 0.10 | 0 | 1822 | 0.7 | 0.0 | 1 | 0 | 3592 | 1.1 | 0.0 | 1 |
| | | 0.50 | 0 | 2955 | 191.0 | 0.0 | 1 | 0 | 5939 | 5.1 | 0.0 | 1 |
| | | 0.80 | 0 | 3830 | 10000.4 | 0.3 | 0 | 0 | 6803 | 7.7 | 0.0 | 1 |
| | | 1.00 | 0 | 4503 | 10001.2 | 1.9 | 0 | 0 | 7505 | 11.4 | 0.0 | 1 |
| | 16 | 0.10 | 0 | 2496 | 0.9 | 0.0 | 1 | 0 | 3639 | 0.6 | 0.0 | 1 |
| | | 0.50 | 0 | 3702 | 9.8 | 0.0 | 1 | 0 | 4941 | 3.2 | 0.0 | 1 |
| | | 0.80 | 0 | 4113 | 5490.9 | 0.0 | 1 | 0 | 6278 | 6.9 | 0.0 | 1 |
| | | 1.00 | 0 | 4622 | 10000.9 | 0.7 | 0 | 0 | 6902 | 10.2 | 0.0 | 1 |
| | 17 | 0.10 | 0 | 1973 | 2.4 | 0.0 | 1 | 0 | 3401 | 1.2 | 0.0 | 1 |
| | | 0.50 | 0 | 3861 | 2176.9 | 0.0 | 1 | 0 | 7456 | 6.5 | 0.0 | 1 |
| | | 0.80 | 0 | 4308 | 10000.5 | 2.6 | 0 | 0 | 8000 | 13.3 | 0.0 | 1 |
| | | 1.00 | 0 | 4877 | 10000.9 | 5.0 | 0 | 0 | 7986 | 16.6 | 0.0 | 1 |
| | 18 | 0.10 | 0 | 1889 | 0.7 | 0.0 | 1 | 0 | 2603 | 0.8 | 0.0 | 1 |
| | | 0.50 | 0 | 2980 | 9999.9 | 1.3 | 0 | 0 | 6363 | 6.3 | 0.0 | 1 |
| | | 0.80 | 0 | 3383 | 10001.2 | 2.7 | 0 | 0 | 7573 | 11.0 | 0.0 | 1 |
| | | 1.00 | 0 | 4004 | 10002.3 | 7.3 | 0 | 0 | 7890 | 16.5 | 0.0 | 1 |
| | 19 | 0.10 | 0 | 2148 | 0.8 | 0.0 | 1 | 0 | 3214 | 0.7 | 0.0 | 1 |
| | | 0.50 | 0 | 3182 | 1058.8 | 0.0 | 1 | 0 | 5594 | 7.8 | 0.0 | 1 |
| | | 0.80 | 0 | 3680 | 10001.8 | 3.7 | 0 | 0 | 6359 | 11.2 | 0.0 | 1 |
| | | 1.00 | 0 | 4039 | 10002.3 | 2.7 | 0 | 0 | 6769 | 15.4 | 0.0 | 1 |
| | 20 | 0.10 | 0 | 1548 | 0.9 | 0.0 | 1 | 0 | 2586 | 0.7 | 0.0 | 1 |
| | | 0.50 | 0 | 3468 | 9999.9 | 1.1 | 0 | 0 | 6784 | 7.4 | 0.0 | 1 |
| | | 0.80 | 0 | 3924 | 10001.3 | 1.4 | 0 | 0 | 7405 | 12.9 | 0.0 | 1 |
| | | 1.00 | 0 | 4336 | 10001.8 | 3.0 | 0 | 0 | 8055 | 19.4 | 0.0 | 1 |
| | 21 | 0.10 | 0 | 1923 | 2.3 | 0.0 | 1 | 0 | 3799 | 1.6 | 0.0 | 1 |
| | | 0.50 | 0 | 3283 | 4776.0 | 0.0 | 1 | 0 | 6795 | 7.7 | 0.0 | 1 |
| | | 0.80 | 0 | 3893 | 10002.0 | 4.3 | 0 | 0 | 7724 | 16.0 | 0.0 | 1 |
| | | 1.00 | 0 | 4168 | 10002.5 | 5.3 | 0 | 0 | 7971 | 19.8 | 0.0 | 1 |
| | 22 | 0.10 | 0 | 1065 | 0.5 | 0.0 | 1 | 0 | 1454 | 0.5 | 0.0 | 1 |
| | | 0.50 | 0 | 2776 | 12.1 | 0.0 | 1 | 0 | 5583 | 3.9 | 0.0 | 1 |
| | | 0.80 | 0 | 3686 | 9470.6 | 2.2 | 0 | 0 | 6795 | 8.3 | 0.0 | 1 |
| | | 1.00 | 0 | 3979 | 10001.4 | 1.7 | 0 | 0 | 7620 | 13.8 | 0.0 | 1 |
| | 23 | 0.10 | 0 | 1518 | 1.1 | 0.0 | 1 | 0 | 2845 | 1.2 | 0.0 | 1 |
| | | 0.50 | 0 | 2922 | 3318.1 | 0.0 | 1 | 0 | 5319 | 4.7 | 0.0 | 1 |
| | | 0.80 | 0 | 3857 | 10001.4 | 1.3 | 0 | 0 | 6785 | 10.2 | 0.0 | 1 |
| | | 1.00 | 0 | 4506 | 10002.1 | 2.2 | 0 | 0 | 7491 | 18.1 | 0.0 | 1 |
| | 24 | 0.10 | 0 | 2458 | 1.2 | 0.0 | 1 | 0 | 3910 | 1.0 | 0.0 | 1 |
| | | 0.50 | 0 | 3762 | 19.0 | 0.0 | 1 | 0 | 6516 | 6.1 | 0.0 | 1 |
| | | 0.80 | 0 | 4311 | 10000.9 | 1.3 | 0 | 0 | 7833 | 10.9 | 0.0 | 1 |
| | | 1.00 | 0 | 4505 | 10002.2 | 2.5 | 0 | 0 | 7773 | 14.5 | 0.0 | 1 |
| | 25 | 0.10 | 0 | 1940 | 2.0 | 0.0 | 1 | 0 | 4142 | 0.9 | 0.0 | 1 |
| | | 0.50 | 0 | 3291 | 1048.4 | 0.0 | 1 | 0 | 6984 | 5.2 | 0.0 | 1 |
| | | 0.80 | 0 | 3925 | 10001.3 | 2.4 | 0 | 0 | 7912 | 12.9 | 0.0 | 1 |
| | | 1.00 | 0 | 4505 | 10001.7 | 6.2 | 0 | 0 | 7932 | 19.3 | 0.0 | 1 |
| | 26 | 0.10 | 0 | 2340 | 1.7 | 0.0 | 1 | 0 | 3340 | 0.5 | 0.0 | 1 |
| | | 0.50 | 0 | 3534 | 39.2 | 0.0 | 1 | 0 | 6147 | 4.9 | 0.0 | 1 |
| | | 0.80 | 0 | 3997 | 10001.3 | 1.3 | 0 | 0 | 7801 | 12.7 | 0.0 | 1 |
| | | 1.00 | 0 | 4351 | 10001.8 | 3.4 | 0 | 0 | 8173 | 20.0 | 0.0 | 1 |
| | 27 | 0.10 | 0 | 1445 | 1.0 | 0.0 | 1 | 0 | 2193 | 0.9 | 0.0 | 1 |
| | | 0.50 | 0 | 2868 | 5.5 | 0.0 | 1 | 0 | 6244 | 3.6 | 0.0 | 1 |
| | | 0.80 | 0 | 3712 | 5991.7 | 0.0 | 1 | 0 | 7219 | 6.7 | 0.0 | 1 |
| | | 1.00 | 0 | 3985 | 7333.2 | 1.5 | 0 | 0 | 7465 | 9.2 | 0.0 | 1 |
| | 28 | 0.10 | 0 | 2137 | 0.8 | 0.0 | 1 | 0 | 3765 | 1.3 | 0.0 | 1 |
| | | 0.50 | 0 | 3872 | 24.8 | 0.0 | 1 | 0 | 7234 | 6.4 | 0.0 | 1 |
| | | 0.80 | 0 | 4183 | 1864.8 | 0.0 | 1 | 0 | 7817 | 12.0 | 0.0 | 1 |
| | | 1.00 | 0 | 4589 | 10001.5 | 3.1 | 0 | 0 | 8307 | 16.2 | 0.0 | 1 |
| | 29 | 0.10 | 0 | 2377 | 4.1 | 0.0 | 1 | 0 | 4260 | 1.5 | 0.0 | 1 |
| | | 0.50 | 0 | 3387 | 9999.8 | 1.9 | 0 | 0 | 6603 | 7.2 | 0.0 | 1 |
| | | 0.80 | 0 | 4213 | 10000.9 | 2.8 | 0 | 0 | 7541 | 11.5 | 0.0 | 1 |
| | | 1.00 | 0 | 4405 | 10001.3 | 2.4 | 0 | 0 | 7990 | 16.6 | 0.0 | 1 |

Table A.46: Performance of ILP-O and ILP-S for instance size (Sz.) 200, instance numbers (Nr.) 0–14.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 200 | 0 | 0.10 | 0 | 2215 | 6.3 | 0.0 | 1 | 0 | 3643 | 1.3 | 0.0 | 1 |
| | | 0.50 | 0 | 4452 | 10001.7 | 0.6 | 0 | 0 | 8102 | 14.0 | 0.0 | 1 |
| | | 0.80 | 0 | 5370 | 10004.0 | 2.3 | 0 | 0 | 10515 | 30.3 | 0.0 | 1 |
| | | 1.00 | 0 | 6080 | 10004.9 | 4.9 | 0 | 0 | 11662 | 41.3 | 0.0 | 1 |
| | 1 | 0.10 | 0 | 1841 | 5.4 | 0.0 | 1 | 0 | 2688 | 3.4 | 0.0 | 1 |
| | | 0.50 | 0 | 3922 | 10003.5 | 0.6 | 0 | 0 | 7578 | 17.6 | 0.0 | 1 |
| | | 0.80 | 0 | 5201 | 10005.2 | 3.8 | 0 | 0 | 9675 | 26.2 | 0.0 | 1 |
| | | 1.00 | 0 | 6407 | 10005.2 | 6.0 | 0 | 0 | 11754 | 32.2 | 0.0 | 1 |
| | 2 | 0.10 | 0 | 2650 | 5.6 | 0.0 | 1 | 0 | 4520 | 4.5 | 0.0 | 1 |
| | | 0.50 | 0 | 3998 | 126.6 | 0.0 | 1 | 0 | 7427 | 13.8 | 0.0 | 1 |
| | | 0.80 | 0 | 4828 | 10004.1 | 3.1 | 0 | 0 | 9940 | 22.7 | 0.0 | 1 |
| | | 1.00 | 0 | 5877 | 10006.5 | 7.5 | 0 | 0 | 11183 | 41.8 | 0.0 | 1 |
| | 3 | 0.10 | 0 | 3513 | 2.1 | 0.0 | 1 | 0 | 5700 | 1.8 | 0.0 | 1 |
| | | 0.50 | 0 | 5553 | 6799.3 | 0.0 | 1 | 0 | 8456 | 12.6 | 0.0 | 1 |
| | | 0.80 | 0 | 6422 | 10003.7 | 0.2 | 0 | 0 | 10759 | 21.2 | 0.0 | 1 |
| | | 1.00 | 0 | 7172 | 10005.6 | 1.7 | 0 | 0 | 11644 | 29.8 | 0.0 | 1 |
| | 4 | 0.10 | 0 | 2807 | 4.5 | 0.0 | 1 | 0 | 5494 | 2.7 | 0.0 | 1 |
| | | 0.50 | 0 | 4674 | 83.8 | 0.0 | 1 | 0 | 9154 | 11.9 | 0.0 | 1 |
| | | 0.80 | 0 | 5783 | 10003.9 | 1.4 | 0 | 0 | 11533 | 22.6 | 0.0 | 1 |
| | | 1.00 | 0 | 6715 | 10004.2 | 5.9 | 0 | 0 | 12783 | 38.8 | 0.0 | 1 |
| | 5 | 0.10 | 0 | 2980 | 1.4 | 0.0 | 1 | 0 | 3968 | 0.8 | 0.0 | 1 |
| | | 0.50 | 0 | 4816 | 110.4 | 0.0 | 1 | 0 | 8105 | 9.2 | 0.0 | 1 |
| | | 0.80 | 0 | 5797 | 10003.1 | 1.4 | 0 | 0 | 9541 | 17.0 | 0.0 | 1 |
| | | 1.00 | 0 | 6432 | 10005.6 | 6.6 | 0 | 0 | 11174 | 28.6 | 0.0 | 1 |
| | 6 | 0.10 | 0 | 3202 | 86.4 | 0.0 | 1 | 0 | 5947 | 4.1 | 0.0 | 1 |
| | | 0.50 | 0 | 4826 | 10002.0 | 2.2 | 0 | 0 | 8797 | 24.8 | 0.0 | 1 |
| | | 0.80 | 739 | 9499 | 10004.8 | 100.0 | 0 | 0 | 11354 | 48.0 | 0.0 | 1 |
| | | 1.00 | 4553 | 11680 | 10005.2 | 100.0 | 0 | 0 | 12118 | 56.5 | 0.0 | 1 |
| | 7 | 0.10 | 0 | 2776 | 3.4 | 0.0 | 1 | 0 | 5156 | 4.0 | 0.0 | 1 |
| | | 0.50 | 0 | 5243 | 10002.1 | 1.9 | 0 | 0 | 9020 | 16.3 | 0.0 | 1 |
| | | 0.80 | 100 | 10352 | 10004.1 | 100.0 | 0 | 0 | 10904 | 24.3 | 0.0 | 1 |
| | | 1.00 | 0 | 7157 | 10005.8 | 7.7 | 0 | 0 | 12432 | 41.4 | 0.0 | 1 |
| | 8 | 0.10 | 0 | 1511 | 4.7 | 0.0 | 1 | 0 | 3097 | 4.2 | 0.0 | 1 |
| | | 0.50 | 0 | 6417 | 10002.9 | 33.8 | 0 | 0 | 9390 | 24.8 | 0.0 | 1 |
| | | 0.80 | 430 | 8930 | 10006.1 | 100.0 | 0 | 0 | 11296 | 52.8 | 0.0 | 1 |
| | | 1.00 | 4614 | 11950 | 10008.0 | 100.0 | 0 | 0 | 12491 | 116.7 | 0.0 | 1 |
| | 9 | 0.10 | 0 | 2238 | 1.8 | 0.0 | 1 | 0 | 3222 | 1.1 | 0.0 | 1 |
| | | 0.50 | 0 | 4888 | 10002.6 | 0.5 | 0 | 0 | 9978 | 14.7 | 0.0 | 1 |
| | | 0.80 | 0 | 5541 | 10006.3 | 1.8 | 0 | 0 | 11119 | 27.8 | 0.0 | 1 |
| | | 1.00 | 8678 | 9961 | 10008.0 | 100.0 | 0 | 0 | 13220 | 47.2 | 0.0 | 1 |
| | 10 | 0.10 | 0 | 1734 | 1.2 | 0.0 | 1 | 0 | 2487 | 1.3 | 0.0 | 1 |
| | | 0.50 | 0 | 4733 | 10002.5 | 1.4 | 0 | 0 | 8743 | 13.2 | 0.0 | 1 |
| | | 0.80 | 0 | 6237 | 10004.5 | 1.6 | 0 | 0 | 11916 | 24.7 | 0.0 | 1 |
| | | 1.00 | 0 | 6978 | 10006.1 | 3.6 | 0 | 0 | 12960 | 38.4 | 0.0 | 1 |
| | 11 | 0.10 | 0 | 1849 | 17.3 | 0.0 | 1 | 0 | 3583 | 16.0 | 0.0 | 1 |
| | | 0.50 | 250 | 6800 | 10005.8 | 100.0 | 0 | 0 | 10490 | 49.9 | 0.0 | 1 |
| | | 0.80 | 350 | 10157 | 10008.9 | 100.0 | 0 | 0 | 13103 | 123.1 | 0.0 | 1 |
| | | 1.00 | 17462 | 11850 | 10012.1 | 100.0 | 0 | 0 | 14207 | 355.1 | 0.0 | 1 |
| | 12 | 0.10 | 0 | 2696 | 5.8 | 0.0 | 1 | 0 | 4656 | 6.1 | 0.0 | 1 |
| | | 0.50 | 0 | 4467 | 10002.4 | 0.1 | 0 | 0 | 7421 | 15.6 | 0.0 | 1 |
| | | 0.80 | 0 | 5195 | 9094.6 | 0.0 | 1 | 0 | 9823 | 25.4 | 0.0 | 1 |
| | | 1.00 | 0 | 6237 | 10006.3 | 5.4 | 0 | 0 | 12088 | 40.3 | 0.0 | 1 |
| | 13 | 0.10 | 0 | 2169 | 5.9 | 0.0 | 1 | 0 | 3486 | 3.0 | 0.0 | 1 |
| | | 0.50 | 0 | 3792 | 334.0 | 0.0 | 1 | 0 | 6706 | 17.8 | 0.0 | 1 |
| | | 0.80 | 0 | 7293 | 10006.8 | 37.6 | 0 | 0 | 9397 | 29.8 | 0.0 | 1 |
| | | 1.00 | 0 | 5577 | 10007.0 | 6.6 | 0 | 0 | 11255 | 44.0 | 0.0 | 1 |
| | 14 | 0.10 | 0 | 3194 | 1172.5 | 0.0 | 1 | 0 | 6366 | 7.5 | 0.0 | 1 |
| | | 0.50 | 0 | 6231 | 10002.5 | 31.7 | 0 | 0 | 8345 | 20.2 | 0.0 | 1 |
| | | 0.80 | 600 | 8728 | 10005.9 | 100.0 | 0 | 0 | 11720 | 60.7 | 0.0 | 1 |
| | | 1.00 | 3409 | 10386 | 10007.5 | 100.0 | 0 | 0 | 12888 | 94.8 | 0.0 | 1 |

Table A.47: Performance of ILP-O and ILP-S for instance size (Sz.) 200, instance numbers (Nr.) 15–29.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 200 | 15 | 0.10 | 0 | 2489 | 5.8 | 0.0 | 1 | 0 | 4206 | 1.8 | 0.0 | 1 |
| | | 0.50 | 0 | 4193 | 71.4 | 0.0 | 1 | 0 | 8536 | 11.2 | 0.0 | 1 |
| | | 0.80 | 0 | 5067 | 10003.7 | 1.1 | 0 | 0 | 9553 | 22.7 | 0.0 | 1 |
| | | 1.00 | 0 | 5756 | 10004.2 | 3.8 | 0 | 0 | 11220 | 35.2 | 0.0 | 1 |
| | 16 | 0.10 | 0 | 2649 | 9.0 | 0.0 | 1 | 0 | 4321 | 3.2 | 0.0 | 1 |
| | | 0.50 | 0 | 3941 | 7659.6 | 0.0 | 1 | 0 | 7721 | 17.3 | 0.0 | 1 |
| | | 0.80 | 0 | 5201 | 10004.4 | 2.4 | 0 | 0 | 10027 | 27.6 | 0.0 | 1 |
| | | 1.00 | 3588 | 10403 | 10005.4 | 100.0 | 0 | 0 | 12455 | 47.6 | 0.0 | 1 |
| | 17 | 0.10 | 0 | 1838 | 3.5 | 0.0 | 1 | 0 | 3533 | 3.4 | 0.0 | 1 |
| | | 0.50 | 0 | 4627 | 10002.0 | 1.6 | 0 | 0 | 8508 | 16.6 | 0.0 | 1 |
| | | 0.80 | 0 | 6196 | 10003.9 | 8.4 | 0 | 0 | 11135 | 28.8 | 0.0 | 1 |
| | | 1.00 | 3826 | 11663 | 10006.6 | 100.0 | 0 | 0 | 12952 | 49.9 | 0.0 | 1 |
| | 18 | 0.10 | 0 | 2037 | 4.2 | 0.0 | 1 | 0 | 3959 | 3.2 | 0.0 | 1 |
| | | 0.50 | 0 | 4313 | 10003.6 | 0.2 | 0 | 0 | 8541 | 19.6 | 0.0 | 1 |
| | | 0.80 | 0 | 5763 | 10005.7 | 4.9 | 0 | 0 | 10523 | 33.8 | 0.0 | 1 |
| | | 1.00 | 5073 | 9603 | 10009.1 | 100.0 | 0 | 0 | 11741 | 53.0 | 0.0 | 1 |
| | 19 | 0.10 | 0 | 2351 | 2.2 | 0.0 | 1 | 0 | 3668 | 1.9 | 0.0 | 1 |
| | | 0.50 | 0 | 3989 | 6174.1 | 0.0 | 1 | 0 | 7039 | 11.1 | 0.0 | 1 |
| | | 0.80 | 0 | 5797 | 10003.7 | 6.5 | 0 | 0 | 10318 | 23.0 | 0.0 | 1 |
| | | 1.00 | 0 | 6712 | 10004.5 | 9.5 | 0 | 0 | 11954 | 31.2 | 0.0 | 1 |
| | 20 | 0.10 | 0 | 2598 | 23.1 | 0.0 | 1 | 0 | 5224 | 3.9 | 0.0 | 1 |
| | | 0.50 | 0 | 4652 | 10001.7 | 0.4 | 0 | 0 | 8840 | 13.6 | 0.0 | 1 |
| | | 0.80 | 0 | 5735 | 10003.7 | 5.4 | 0 | 0 | 10642 | 24.6 | 0.0 | 1 |
| | | 1.00 | 0 | 6756 | 10004.8 | 9.0 | 0 | 0 | 11786 | 30.3 | 0.0 | 1 |
| | 21 | 0.10 | 0 | 2481 | 1.2 | 0.0 | 1 | 0 | 3710 | 1.1 | 0.0 | 1 |
| | | 0.50 | 0 | 4566 | 25.7 | 0.0 | 1 | 0 | 7960 | 8.4 | 0.0 | 1 |
| | | 0.80 | 0 | 5921 | 3776.0 | 0.0 | 1 | 0 | 10463 | 15.8 | 0.0 | 1 |
| | | 1.00 | 0 | 6341 | 10003.6 | 1.8 | 0 | 0 | 12437 | 19.3 | 0.0 | 1 |
| | 22 | 0.10 | 0 | 2997 | 3.9 | 0.0 | 1 | 0 | 5071 | 2.0 | 0.0 | 1 |
| | | 0.50 | 0 | 4950 | 169.8 | 0.0 | 1 | 0 | 9109 | 18.7 | 0.0 | 1 |
| | | 0.80 | 0 | 5970 | 10004.4 | 0.4 | 0 | 0 | 10918 | 26.9 | 0.0 | 1 |
| | | 1.00 | 0 | 6855 | 10006.6 | 4.5 | 0 | 0 | 12466 | 53.0 | 0.0 | 1 |
| | 23 | 0.10 | 0 | 2537 | 3.8 | 0.0 | 1 | 0 | 3988 | 2.9 | 0.0 | 1 |
| | | 0.50 | 0 | 4893 | 10003.1 | 1.5 | 0 | 0 | 8621 | 19.8 | 0.0 | 1 |
| | | 0.80 | 0 | 5711 | 10004.5 | 3.7 | 0 | 0 | 9939 | 32.2 | 0.0 | 1 |
| | | 1.00 | 0 | 7260 | 10006.1 | 9.2 | 0 | 0 | 12632 | 40.8 | 0.0 | 1 |
| | 24 | 0.10 | 0 | 2193 | 3.2 | 0.0 | 1 | 0 | 3424 | 1.8 | 0.0 | 1 |
| | | 0.50 | 0 | 4617 | 65.5 | 0.0 | 1 | 0 | 8088 | 9.5 | 0.0 | 1 |
| | | 0.80 | 0 | 5517 | 10005.6 | 2.9 | 0 | 0 | 9712 | 24.0 | 0.0 | 1 |
| | | 1.00 | 0 | 6342 | 10006.0 | 3.4 | 0 | 0 | 11662 | 31.8 | 0.0 | 1 |
| | 25 | 0.10 | 0 | 2790 | 21.4 | 0.0 | 1 | 0 | 5007 | 3.1 | 0.0 | 1 |
| | | 0.50 | 0 | 4645 | 10003.1 | 5.4 | 0 | 0 | 8918 | 21.2 | 0.0 | 1 |
| | | 0.80 | 0 | 5549 | 10006.1 | 8.3 | 0 | 0 | 11289 | 39.4 | 0.0 | 1 |
| | | 1.00 | 0 | 6347 | 10006.0 | 8.1 | 0 | 0 | 12373 | 36.9 | 0.0 | 1 |
| | 26 | 0.10 | 0 | 2895 | 17.9 | 0.0 | 1 | 0 | 4772 | 6.7 | 0.0 | 1 |
| | | 0.50 | 0 | 4330 | 2205.0 | 0.0 | 1 | 0 | 7295 | 16.8 | 0.0 | 1 |
| | | 0.80 | 0 | 5744 | 10005.9 | 2.0 | 0 | 0 | 10030 | 28.0 | 0.0 | 1 |
| | | 1.00 | 0 | 6257 | 10007.3 | 4.4 | 0 | 0 | 11055 | 45.8 | 0.0 | 1 |
| | 27 | 0.10 | 0 | 2416 | 1.1 | 0.0 | 1 | 0 | 3547 | 1.2 | 0.0 | 1 |
| | | 0.50 | 0 | 4503 | 253.7 | 0.0 | 1 | 0 | 7710 | 9.0 | 0.0 | 1 |
| | | 0.80 | 0 | 5544 | 10003.3 | 0.7 | 0 | 0 | 9842 | 17.5 | 0.0 | 1 |
| | | 1.00 | 0 | 5947 | 10005.2 | 3.1 | 0 | 0 | 10985 | 25.2 | 0.0 | 1 |
| | 28 | 0.10 | 0 | 2331 | 2.1 | 0.0 | 1 | 0 | 4481 | 1.4 | 0.0 | 1 |
| | | 0.50 | 0 | 3864 | 10001.4 | 2.3 | 0 | 0 | 8410 | 14.6 | 0.0 | 1 |
| | | 0.80 | 0 | 5286 | 10004.7 | 6.5 | 0 | 0 | 10270 | 27.4 | 0.0 | 1 |
| | | 1.00 | 0 | 6481 | 10007.1 | 8.3 | 0 | 0 | 12288 | 44.9 | 0.0 | 1 |
| | 29 | 0.10 | 0 | 2318 | 28.0 | 0.0 | 1 | 0 | 4985 | 2.6 | 0.0 | 1 |
| | | 0.50 | 0 | 4320 | 10002.2 | 3.4 | 0 | 0 | 9784 | 19.7 | 0.0 | 1 |
| | | 0.80 | 0 | 6018 | 10048.8 | 7.6 | 0 | 0 | 11948 | 55.3 | 0.0 | 1 |
| | | 1.00 | 1084 | 10443 | 10009.3 | 100.0 | 0 | 0 | 12903 | 74.3 | 0.0 | 1 |

Table A.48: Performance of ILP-O and ILP-S for instance size (Sz.) 500, instance numbers (Nr.) 0–14.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 500 | 0 | 0.10 | 0 | 3722 | 250.6 | 0.0 | 1 | 0 | 6596 | 13.5 | 0.0 | 1 |
| | | 0.50 | 0 | 6726 | 736.2 | 0.0 | 1 | 0 | 12177 | 46.1 | 0.0 | 1 |
| | | 0.80 | 0 | 8286 | 10027.2 | 2.9 | 0 | 0 | 15490 | 96.7 | 0.0 | 1 |
| | | 1.00 | 16483 | 16274 | 10025.7 | 100.0 | 0 | 0 | 20935 | 193.9 | 0.0 | 1 |
| | 1 | 0.10 | 0 | 3633 | 1985.5 | 0.0 | 1 | 0 | 6325 | 7.6 | 0.0 | 1 |
| | | 0.50 | 145 | 9123 | 10015.4 | 100.0 | 0 | 0 | 11265 | 60.7 | 0.0 | 1 |
| | | 0.80 | 1264 | 13307 | 10025.8 | 100.0 | 0 | 0 | 14666 | 136.4 | 0.0 | 1 |
| | | 1.00 | 9320 | 15518 | 10029.5 | 100.0 | 0 | 0 | 21198 | 1064.4 | 0.0 | 1 |
| | 2 | 0.10 | 0 | 3329 | 36.4 | 0.0 | 1 | 0 | 6433 | 7.2 | 0.0 | 1 |
| | | 0.50 | 0 | 6243 | 10010.4 | 1.2 | 0 | 0 | 10252 | 37.8 | 0.0 | 1 |
| | | 0.80 | 2459 | 13800 | 10021.6 | 100.0 | 0 | 0 | 15297 | 88.4 | 0.0 | 1 |
| | | 1.00 | 16322 | 17450 | 10024.7 | 100.0 | 0 | 0 | 20207 | 160.2 | 0.0 | 1 |
| | 3 | 0.10 | 0 | 2987 | 14.3 | 0.0 | 1 | 0 | 5033 | 11.4 | 0.0 | 1 |
| | | 0.50 | 0 | 5782 | 10015.3 | 1.1 | 0 | 0 | 10002 | 56.0 | 0.0 | 1 |
| | | 0.80 | 0 | 13022 | 10043.3 | 76.8 | 0 | 0 | 15190 | 187.8 | 0.0 | 1 |
| | | 1.00 | 4795 | 16388 | 10047.5 | 100.0 | 0 | 0 | 25129 | 408.4 | 0.0 | 1 |
| | 4 | 0.10 | 0 | 2555 | 31.7 | 0.0 | 1 | 0 | 5284 | 11.7 | 0.0 | 1 |
| | | 0.50 | 0 | 8967 | 10019.6 | 32.8 | 0 | 0 | 11402 | 57.5 | 0.0 | 1 |
| | | 0.80 | 45 | 12775 | 10025.6 | 100.0 | 0 | 0 | 15281 | 157.1 | 0.0 | 1 |
| | | 1.00 | 5868 | 16026 | 10042.8 | 100.0 | 0 | 0 | 22320 | 505.1 | 0.0 | 1 |
| | 5 | 0.10 | 0 | 2858 | 38.5 | 0.0 | 1 | 0 | 5262 | 7.1 | 0.0 | 1 |
| | | 0.50 | 0 | 6648 | 10014.6 | 3.0 | 0 | 0 | 12007 | 56.7 | 0.0 | 1 |
| | | 0.80 | 4851 | 14490 | 10023.0 | 100.0 | 0 | 0 | 16711 | 151.2 | 0.0 | 1 |
| | | 1.00 | 16673 | 17427 | 10033.8 | 100.0 | 0 | 0 | 19699 | 204.0 | 0.0 | 1 |
| | 6 | 0.10 | 0 | 3167 | 237.8 | 0.0 | 1 | 0 | 5495 | 327.7 | 0.0 | 1 |
| | | 0.50 | 0 | 6556 | 10039.3 | 0.1 | 0 | 0 | 12300 | 332.3 | 0.0 | 1 |
| | | 0.80 | 886 | 12920 | 10053.6 | 100.0 | 0 | 0 | 15354 | 368.3 | 0.0 | 1 |
| | | 1.00 | 8250 | 16495 | 10049.3 | 100.0 | 0 | 0 | 20361 | 412.6 | 0.0 | 1 |
| | 7 | 0.10 | 0 | 3220 | 15.3 | 0.0 | 1 | 0 | 4535 | 19.1 | 0.0 | 1 |
| | | 0.50 | 0 | 7170 | 10017.0 | 0.3 | 0 | 0 | 12733 | 101.7 | 0.0 | 1 |
| | | 0.80 | 5255 | 15456 | 10031.1 | 100.0 | 0 | 0 | 17862 | 165.2 | 0.0 | 1 |
| | | 1.00 | 12441 | 18699 | 10038.4 | 100.0 | 0 | 0 | 21870 | 210.3 | 0.0 | 1 |
| | 8 | 0.10 | 0 | 2973 | 2610.9 | 0.0 | 1 | 0 | 6539 | 17.9 | 0.0 | 1 |
| | | 0.50 | 0 | 9553 | 10027.4 | 76.5 | 0 | 0 | 12954 | 90.3 | 0.0 | 1 |
| | | 0.80 | 2349 | 15746 | 10032.4 | 100.0 | 0 | 0 | 22026 | 297.6 | 0.0 | 1 |
| | | 1.00 | 19908 | 19533 | 10035.8 | 100.0 | 0 | 0 | 25113 | 649.3 | 0.0 | 1 |
| | 9 | 0.10 | 0 | 2420 | 35.8 | 0.0 | 1 | 0 | 4565 | 7.5 | 0.0 | 1 |
| | | 0.50 | 20 | 8164 | 10021.5 | 100.0 | 0 | 0 | 10223 | 76.5 | 0.0 | 1 |
| | | 0.80 | 3052 | 12907 | 10029.1 | 100.0 | 0 | 0 | 14751 | 163.4 | 0.0 | 1 |
| | | 1.00 | 9392 | 15606 | 10035.2 | 100.0 | 0 | 0 | 23490 | 587.3 | 0.0 | 1 |
| | 10 | 0.10 | 0 | 2962 | 29.2 | 0.0 | 1 | 0 | 5691 | 13.0 | 0.0 | 1 |
| | | 0.50 | 0 | 8586 | 10017.5 | 36.4 | 0 | 0 | 11010 | 65.3 | 0.0 | 1 |
| | | 0.80 | 344 | 13336 | 10024.4 | 100.0 | 0 | 0 | 17645 | 136.4 | 0.0 | 1 |
| | | 1.00 | 20951 | 17789 | 10035.6 | 100.0 | 0 | 0 | 22149 | 207.8 | 0.0 | 1 |
| | 11 | 0.10 | 0 | 3446 | 11.1 | 0.0 | 1 | 0 | 5288 | 5.9 | 0.0 | 1 |
| | | 0.50 | 0 | 7218 | 10019.1 | 0.9 | 0 | 0 | 11641 | 67.4 | 0.0 | 1 |
| | | 0.80 | 778 | 12513 | 10028.2 | 100.0 | 0 | 0 | 13469 | 91.0 | 0.0 | 1 |
| | | 1.00 | 5612 | 15057 | 10036.1 | 100.0 | 0 | 0 | 17559 | 171.7 | 0.0 | 1 |
| | 12 | 0.10 | 0 | 3061 | 9.1 | 0.0 | 1 | 0 | 5015 | 6.7 | 0.0 | 1 |
| | | 0.50 | 435 | 10761 | 10020.8 | 100.0 | 0 | 0 | 10685 | 73.4 | 0.0 | 1 |
| | | 0.80 | 7555 | 14944 | 10025.3 | 100.0 | 0 | 0 | 15064 | 133.3 | 0.0 | 1 |
| | | 1.00 | 19224 | 17711 | 10030.1 | 100.0 | 0 | 0 | 18521 | 245.7 | 0.0 | 1 |
| | 13 | 0.10 | 0 | 2780 | 20.6 | 0.0 | 1 | 0 | 4777 | 9.3 | 0.0 | 1 |
| | | 0.50 | 0 | 8910 | 10019.0 | 36.4 | 0 | 0 | 12142 | 73.9 | 0.0 | 1 |
| | | 0.80 | 491 | 12995 | 10027.8 | 100.0 | 0 | 0 | 14288 | 115.6 | 0.0 | 1 |
| | | 1.00 | 6984 | 17009 | 10034.8 | 100.0 | 0 | 0 | 23228 | 352.7 | 0.0 | 1 |
| | 14 | 0.10 | 0 | 3451 | 12.7 | 0.0 | 1 | 0 | 5398 | 7.0 | 0.0 | 1 |
| | | 0.50 | 0 | 10190 | 10019.7 | 37.7 | 0 | 0 | 11941 | 65.2 | 0.0 | 1 |
| | | 0.80 | 3412 | 15201 | 10031.7 | 100.0 | 0 | 0 | 17504 | 120.1 | 0.0 | 1 |
| | | 1.00 | 21324 | 19392 | 10044.4 | 100.0 | 0 | 0 | 24708 | 689.4 | 0.0 | 1 |

Table A.49: Performance of ILP-O and ILP-S for instance size (Sz.) 500, instance numbers (Nr.) 15–29.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 500 | 15 | 0.10 | 0 | 3565 | 29.1 | 0.0 | 1 | 0 | 7629 | 12.0 | 0.0 | 1 |
| | | 0.50 | 0 | 8564 | 10018.3 | 35.0 | 0 | 0 | 11050 | 63.7 | 0.0 | 1 |
| | | 0.80 | 822 | 13489 | 10029.4 | 100.0 | 0 | 0 | 15173 | 141.8 | 0.0 | 1 |
| | | 1.00 | 18419 | 18031 | 10044.8 | 100.0 | 0 | 0 | 25124 | 686.2 | 0.0 | 1 |
| | 16 | 0.10 | 0 | 2891 | 104.6 | 0.0 | 1 | 0 | 6172 | 10.5 | 0.0 | 1 |
| | | 0.50 | 0 | 6033 | 7791.4 | 0.0 | 1 | 0 | 11562 | 69.8 | 0.0 | 1 |
| | | 0.80 | 0 | 7942 | 10031.5 | 2.5 | 0 | 0 | 14536 | 130.8 | 0.0 | 1 |
| | | 1.00 | 6602 | 16230 | 10035.2 | 100.0 | 0 | 0 | 23400 | 339.8 | 0.0 | 1 |
| | 17 | 0.10 | 0 | 2108 | 29.5 | 0.0 | 1 | 0 | 3660 | 5.4 | 0.0 | 1 |
| | | 0.50 | 0 | 5440 | 189.4 | 0.0 | 1 | 0 | 9449 | 43.8 | 0.0 | 1 |
| | | 0.80 | 0 | 7056 | 10027.8 | 1.4 | 0 | 0 | 14341 | 86.7 | 0.0 | 1 |
| | | 1.00 | 3856 | 13976 | 10037.1 | 100.0 | 0 | 0 | 18638 | 175.8 | 0.0 | 1 |
| | 18 | 0.10 | 0 | 3445 | 22.9 | 0.0 | 1 | 0 | 5716 | 12.9 | 0.0 | 1 |
| | | 0.50 | 0 | 9346 | 10022.0 | 34.1 | 0 | 0 | 11116 | 64.7 | 0.0 | 1 |
| | | 0.80 | 783 | 14895 | 10036.8 | 100.0 | 0 | 0 | 15484 | 129.7 | 0.0 | 1 |
| | | 1.00 | 8189 | 19675 | 10043.9 | 100.0 | 0 | 0 | 22375 | 311.2 | 0.0 | 1 |
| | 19 | 0.10 | 0 | 3022 | 56.7 | 0.0 | 1 | 0 | 5564 | 18.3 | 0.0 | 1 |
| | | 0.50 | 0 | 6978 | 10021.0 | 3.3 | 0 | 0 | 12364 | 84.1 | 0.0 | 1 |
| | | 0.80 | 2328 | 15313 | 10032.8 | 100.0 | 0 | 0 | 20738 | 340.4 | 0.0 | 1 |
| | | 1.00 | 22383 | 19932 | 10034.3 | 100.0 | 0 | 0 | 25188 | 1035.2 | 0.0 | 1 |
| | 20 | 0.10 | 0 | 3267 | 6.3 | 0.0 | 1 | 0 | 5563 | 6.7 | 0.0 | 1 |
| | | 0.50 | 0 | 6600 | 10014.9 | 2.4 | 0 | 0 | 11601 | 52.8 | 0.0 | 1 |
| | | 0.80 | 140 | 13534 | 10027.9 | 100.0 | 0 | 0 | 14839 | 86.8 | 0.0 | 1 |
| | | 1.00 | 10625 | 17590 | 10034.8 | 100.0 | 0 | 0 | 19820 | 147.0 | 0.0 | 1 |
| | 21 | 0.10 | 0 | 3029 | 52.6 | 0.0 | 1 | 0 | 5963 | 16.3 | 0.0 | 1 |
| | | 0.50 | 0 | 5619 | 10019.7 | 1.2 | 0 | 0 | 9973 | 62.4 | 0.0 | 1 |
| | | 0.80 | 0 | 7904 | 10026.6 | 4.8 | 0 | 0 | 15474 | 116.6 | 0.0 | 1 |
| | | 1.00 | 10421 | 17057 | 10038.8 | 100.0 | 0 | 0 | 24528 | 330.2 | 0.0 | 1 |
| | 22 | 0.10 | 0 | 2446 | 141.2 | 0.0 | 1 | 0 | 4599 | 14.2 | 0.0 | 1 |
| | | 0.50 | 0 | 5553 | 10022.9 | 2.9 | 0 | 0 | 9396 | 66.7 | 0.0 | 1 |
| | | 0.80 | 2914 | 12548 | 10021.1 | 100.0 | 0 | 0 | 15124 | 128.3 | 0.0 | 1 |
| | | 1.00 | 9597 | 14935 | 10035.2 | 100.0 | 0 | 0 | 17656 | 182.4 | 0.0 | 1 |
| | 23 | 0.10 | 0 | 3294 | 15.7 | 0.0 | 1 | 0 | 5565 | 5.8 | 0.0 | 1 |
| | | 0.50 | 0 | 6158 | 10011.8 | 1.8 | 0 | 0 | 10977 | 39.1 | 0.0 | 1 |
| | | 0.80 | 444 | 13460 | 10026.7 | 100.0 | 0 | 0 | 16007 | 99.1 | 0.0 | 1 |
| | | 1.00 | 29457 | 17271 | 10032.9 | 100.0 | 0 | 0 | 20383 | 194.2 | 0.0 | 1 |
| | 24 | 0.10 | 0 | 2738 | 73.9 | 0.0 | 1 | 0 | 5573 | 73.6 | 0.0 | 1 |
| | | 0.50 | 0 | 5766 | 10034.4 | 0.4 | 0 | 0 | 11998 | 176.1 | 0.0 | 1 |
| | | 0.80 | 0 | 7636 | 10044.0 | 3.6 | 0 | 0 | 14699 | 250.5 | 0.0 | 1 |
| | | 1.00 | 25367 | 17170 | 10076.2 | 100.0 | 0 | 0 | 23296 | 450.8 | 0.0 | 1 |
| | 25 | 0.10 | 0 | 3390 | 10003.1 | 0.7 | 0 | 0 | 6703 | 20.8 | 0.0 | 1 |
| | | 0.50 | 0 | 7041 | 10019.0 | 2.5 | 0 | 0 | 12706 | 90.2 | 0.0 | 1 |
| | | 0.80 | 2290 | 15194 | 10029.5 | 100.0 | 0 | 0 | 22136 | 335.3 | 0.0 | 1 |
| | | 1.00 | 8822 | 18437 | 10035.0 | 100.0 | 0 | 0 | 25256 | 2181.3 | 0.0 | 1 |
| | 26 | 0.10 | 0 | 2888 | 34.1 | 0.0 | 1 | 0 | 5585 | 9.7 | 0.0 | 1 |
| | | 0.50 | 0 | 6340 | 10014.9 | 2.9 | 0 | 0 | 11408 | 58.9 | 0.0 | 1 |
| | | 0.80 | 5587 | 14590 | 10035.1 | 100.0 | 0 | 0 | 18068 | 138.2 | 0.0 | 1 |
| | | 1.00 | 23692 | 17156 | 10036.3 | 100.0 | 0 | 0 | 19893 | 199.5 | 0.0 | 1 |
| | 27 | 0.10 | 0 | 3425 | 74.9 | 0.0 | 1 | 0 | 6049 | 13.0 | 0.0 | 1 |
| | | 0.50 | 0 | 6173 | 10022.9 | 1.0 | 0 | 0 | 11214 | 87.4 | 0.0 | 1 |
| | | 0.80 | 4348 | 14665 | 10032.7 | 100.0 | 0 | 0 | 16081 | 106.7 | 0.0 | 1 |
| | | 1.00 | 18469 | 16927 | 10038.9 | 100.0 | 0 | 0 | 20624 | 227.8 | 0.0 | 1 |
| | 28 | 0.10 | 0 | 3004 | 67.7 | 0.0 | 1 | 0 | 5919 | 14.4 | 0.0 | 1 |
| | | 0.50 | 0 | 9044 | 10026.3 | 73.9 | 0 | 0 | 12079 | 101.6 | 0.0 | 1 |
| | | 0.80 | 4672 | 14631 | 10036.5 | 100.0 | 0 | 0 | 20703 | 261.2 | 0.0 | 1 |
| | | 1.00 | 14827 | 17823 | 10042.2 | 100.0 | 0 | 0 | 23266 | 314.9 | 0.0 | 1 |
| | 29 | 0.10 | 0 | 3777 | 55.9 | 0.0 | 1 | 0 | 5939 | 23.4 | 0.0 | 1 |
| | | 0.50 | 0 | 6028 | 412.1 | 0.0 | 1 | 0 | 9413 | 55.9 | 0.0 | 1 |
| | | 0.80 | 0 | 12656 | 10038.0 | 40.6 | 0 | 0 | 15136 | 96.1 | 0.0 | 1 |
| | | 1.00 | 8686 | 16546 | 10054.0 | 100.0 | 0 | 0 | 20333 | 146.9 | 0.0 | 1 |

Table A.50: Performance of ILP-O and ILP-S for instance size (Sz.) 1000, instance numbers (Nr.) 0–14.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 1000 | 0 | 0.10 | 0 | 3912 | 705.0 | 0.0 | 1 | 0 | 7977 | 62.4 | 0.0 | 1 |
| | | 0.50 | 41 | 15071 | 10069.3 | 100.0 | 0 | 0 | 16389 | 194.1 | 0.0 | 1 |
| | | 0.80 | 3663 | 22502 | 10211.6 | 100.0 | 0 | 0 | 30527 | 721.7 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 37444 | 1640.8 | 0.0 | 1 |
| | 1 | 0.10 | 0 | 3262 | 10003.7 | 0.3 | 0 | 0 | 5873 | 32.2 | 0.0 | 1 |
| | | 0.50 | 0 | 8308 | 10041.0 | 3.0 | 0 | 0 | 14212 | 161.8 | 0.0 | 1 |
| | | 0.80 | 2433 | 21052 | 10087.1 | 100.0 | 0 | 0 | 27025 | 481.2 | 0.0 | 1 |
| | | 1.00 | 16846 | 26477 | 10147.8 | 100.0 | 0 | 0 | 35046 | 1097.6 | 0.0 | 1 |
| | 2 | 0.10 | 0 | 3966 | 83.2 | 0.0 | 1 | 0 | 7125 | 41.0 | 0.0 | 1 |
| | | 0.50 | 0 | 8850 | 10085.0 | 2.9 | 0 | 0 | 14804 | 246.5 | 0.0 | 1 |
| | | 0.80 | 7705 | 20612 | 10160.9 | 100.0 | 0 | 0 | 28623 | 586.4 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35462 | 1546.1 | 0.0 | 1 |
| | 3 | 0.10 | 0 | 3056 | 65.7 | 0.0 | 1 | 0 | 5323 | 46.3 | 0.0 | 1 |
| | | 0.50 | 78 | 12973 | 10074.6 | 100.0 | 0 | 0 | 13739 | 194.4 | 0.0 | 1 |
| | | 0.80 | 620 | 20989 | 10120.9 | 100.0 | 0 | 0 | 26914 | 692.8 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35649 | 1954.9 | 0.0 | 1 |
| | 4 | 0.10 | 0 | 4378 | 80.0 | 0.0 | 1 | 0 | 8169 | 37.8 | 0.0 | 1 |
| | | 0.50 | 411 | 13991 | 10102.2 | 100.0 | 0 | 0 | 21674 | 258.2 | 0.0 | 1 |
| | | 0.80 | 5337 | 18736 | 10149.8 | 100.0 | 0 | 0 | 27998 | 794.1 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35095 | 1845.7 | 0.0 | 1 |
| | 5 | 0.10 | 0 | 4801 | 298.4 | 0.0 | 1 | 0 | 8539 | 34.8 | 0.0 | 1 |
| | | 0.50 | 0 | 9183 | 10104.3 | 1.6 | 0 | 0 | 21531 | 266.0 | 0.0 | 1 |
| | | 0.80 | 6039 | 21893 | 10102.0 | 100.0 | 0 | 0 | 29258 | 532.4 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35228 | 1694.8 | 0.0 | 1 |
| | 6 | 0.10 | 0 | 4770 | 143.2 | 0.0 | 1 | 0 | 8831 | 61.9 | 0.0 | 1 |
| | | 0.50 | 0 | 8751 | 10102.6 | 2.3 | 0 | 0 | 14137 | 234.4 | 0.0 | 1 |
| | | 0.80 | 4198 | 19513 | 10140.7 | 100.0 | 0 | 0 | 28161 | 563.6 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35958 | 1269.1 | 0.0 | 1 |
| | 7 | 0.10 | 0 | 2642 | 174.9 | 0.0 | 1 | 0 | 5834 | 49.8 | 0.0 | 1 |
| | | 0.50 | 0 | 6971 | 10075.1 | 1.5 | 0 | 0 | 12615 | 168.6 | 0.0 | 1 |
| | | 0.80 | 4890 | 19680 | 10146.1 | 100.0 | 0 | 0 | 28173 | 516.8 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 34359 | 2069.2 | 0.0 | 1 |
| | 8 | 0.10 | 0 | 3484 | 131.3 | 0.0 | 1 | 0 | 7233 | 21.1 | 0.0 | 1 |
| | | 0.50 | 0 | 6724 | 10055.6 | 0.5 | 0 | 0 | 10636 | 173.9 | 0.0 | 1 |
| | | 0.80 | 8167 | 20085 | 10129.6 | 100.0 | 0 | 0 | 27346 | 514.0 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 33449 | 2172.6 | 0.0 | 1 |
| | 9 | 0.10 | 0 | 3376 | 10009.6 | 0.1 | 0 | 0 | 6783 | 46.7 | 0.0 | 1 |
| | | 0.50 | 0 | 12248 | 10075.6 | 75.3 | 0 | 0 | 18876 | 222.8 | 0.0 | 1 |
| | | 0.80 | 5838 | 19365 | 10090.7 | 100.0 | 0 | 0 | 27061 | 569.4 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 33438 | 1344.9 | 0.0 | 1 |
| | 10 | 0.10 | 0 | 3032 | 540.5 | 0.0 | 1 | 0 | 6460 | 87.6 | 0.0 | 1 |
| | | 0.50 | 0 | 8311 | 10099.9 | 2.7 | 0 | 0 | 14394 | 253.1 | 0.0 | 1 |
| | | 0.80 | 6926 | 21602 | 10155.7 | 100.0 | 0 | 0 | 28290 | 523.6 | 0.0 | 1 |
| | | 1.00 | 13299 | 25286 | 10153.0 | 100.0 | 0 | 0 | 35415 | 1169.9 | 0.0 | 1 |
| | 11 | 0.10 | 0 | 3754 | 2621.8 | 0.0 | 1 | 0 | 7068 | 28.9 | 0.0 | 1 |
| | | 0.50 | 0 | 7722 | 10039.5 | 2.1 | 0 | 0 | 12766 | 155.6 | 0.0 | 1 |
| | | 0.80 | 2403 | 19234 | 10084.5 | 100.0 | 0 | 0 | 28316 | 396.4 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 34903 | 1328.4 | 0.0 | 1 |
| | 12 | 0.10 | 0 | 3454 | 299.3 | 0.0 | 1 | 0 | 7075 | 24.3 | 0.0 | 1 |
| | | 0.50 | 1735 | 14863 | 10089.3 | 100.0 | 0 | 0 | 17087 | 227.3 | 0.0 | 1 |
| | | 0.80 | 5453 | 21292 | 10126.5 | 100.0 | 0 | 0 | 27696 | 420.5 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35966 | 1552.8 | 0.0 | 1 |
| | 13 | 0.10 | 0 | 4204 | 90.9 | 0.0 | 1 | 0 | 7622 | 54.2 | 0.0 | 1 |
| | | 0.50 | 0 | 13085 | 10090.5 | 80.8 | 0 | 0 | 19931 | 239.4 | 0.0 | 1 |
| | | 0.80 | 1964 | 17975 | 10137.9 | 100.0 | 0 | 0 | 26705 | 577.3 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35871 | 5516.8 | 0.0 | 1 |
| | 14 | 0.10 | 0 | 3457 | 129.8 | 0.0 | 1 | 0 | 6745 | 33.0 | 0.0 | 1 |
| | | 0.50 | 84 | 11605 | 10056.4 | 100.0 | 0 | 0 | 11500 | 175.6 | 0.0 | 1 |
| | | 0.80 | 2927 | 17229 | 10090.8 | 100.0 | 0 | 0 | 25899 | 602.7 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35056 | 3160.0 | 0.0 | 1 |

Table A.51: Performance of ILP-O and ILP-S for instance size (Sz.) 1000, instance numbers (Nr.) 15–29.

| | | | ILP-O | | | | | ILP-S | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Sz.** | **Nr.** | **Load** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** | $\mathbf{C_a}$ | $\mathbf{C_u}$ | **t[s]** | **Gap** | **Opt** |
| 1000 | 15 | 0.10 | 0 | 4310 | 155.7 | 0.0 | 1 | 0 | 8216 | 40.3 | 0.0 | 1 |
| | | 0.50 | 0 | 13171 | 10055.9 | 32.9 | 0 | 0 | 15171 | 160.3 | 0.0 | 1 |
| | | 0.80 | 755 | 20466 | 10088.0 | 100.0 | 0 | 0 | 26156 | 378.3 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 33574 | 769.7 | 0.0 | 1 |
| | 16 | 0.10 | 0 | 3379 | 56.9 | 0.0 | 1 | 0 | 6218 | 38.2 | 0.0 | 1 |
| | | 0.50 | 0 | 8248 | 10106.8 | 0.8 | 0 | 0 | 14510 | 250.1 | 0.0 | 1 |
| | | 0.80 | 1402 | 17668 | 10167.9 | 100.0 | 0 | 0 | 27824 | 604.6 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35507 | 1360.2 | 0.0 | 1 |
| | 17 | 0.10 | 0 | 3147 | 10008.0 | 0.5 | 0 | 0 | 6173 | 47.1 | 0.0 | 1 |
| | | 0.50 | 0 | 9120 | 10081.2 | 1.5 | 0 | 0 | 15934 | 243.1 | 0.0 | 1 |
| | | 0.80 | 9575 | 22070 | 10115.8 | 100.0 | 0 | 0 | 30242 | 600.5 | 0.0 | 1 |
| | | 1.00 | 18155 | 24743 | 10156.9 | 100.0 | 0 | 0 | 34411 | 889.2 | 0.0 | 1 |
| | 18 | 0.10 | 0 | 3561 | 221.5 | 0.0 | 1 | 0 | 6544 | 21.1 | 0.0 | 1 |
| | | 0.50 | 470 | 12661 | 10069.1 | 100.0 | 0 | 0 | 18935 | 177.4 | 0.0 | 1 |
| | | 0.80 | 3164 | 20325 | 10113.0 | 100.0 | 0 | 0 | 27722 | 643.3 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | - | - | - | - | - |
| | 19 | 0.10 | 0 | 3228 | 10003.9 | 2.1 | 0 | 0 | 6740 | 25.6 | 0.0 | 1 |
| | | 0.50 | 21 | 13148 | 10060.5 | 100.0 | 0 | 0 | 18448 | 177.5 | 0.0 | 1 |
| | | 0.80 | 981 | 19866 | 10103.3 | 100.0 | 0 | 0 | 27358 | 377.6 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 34735 | 1285.8 | 0.0 | 1 |
| | 20 | 0.10 | 0 | 4874 | 216.4 | 0.0 | 1 | 0 | 8248 | 31.1 | 0.0 | 1 |
| | | 0.50 | 0 | 8140 | 10064.8 | 1.8 | 0 | 0 | 13761 | 178.1 | 0.0 | 1 |
| | | 0.80 | 809 | 20242 | 10117.8 | 100.0 | 0 | 0 | 28112 | 388.9 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 33578 | 727.3 | 0.0 | 1 |
| | 21 | 0.10 | 0 | 3826 | 713.0 | 0.0 | 1 | 0 | 7456 | 68.9 | 0.0 | 1 |
| | | 0.50 | 0 | 8017 | 10122.7 | 2.0 | 0 | 0 | 20455 | 289.2 | 0.0 | 1 |
| | | 0.80 | 1007 | 17922 | 10163.7 | 100.0 | 0 | 0 | 28622 | 602.9 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35560 | 1055.6 | 0.0 | 1 |
| | 22 | 0.10 | 0 | 3666 | 10006.4 | 0.5 | 0 | 0 | 6880 | 35.7 | 0.0 | 1 |
| | | 0.50 | 1661 | 13169 | 10059.0 | 100.0 | 0 | 0 | 17223 | 193.9 | 0.0 | 1 |
| | | 0.80 | 3284 | 20370 | 10145.5 | 100.0 | 0 | 0 | 26152 | 1260.9 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35924 | 1528.4 | 0.0 | 1 |
| | 23 | 0.10 | 0 | 4162 | 109.6 | 0.0 | 1 | 0 | 6915 | 21.3 | 0.0 | 1 |
| | | 0.50 | 36 | 13865 | 10053.6 | 100.0 | 0 | 0 | 14934 | 206.0 | 0.0 | 1 |
| | | 0.80 | 3892 | 19010 | 10097.3 | 100.0 | 0 | 0 | 26239 | 386.9 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 36037 | 1443.3 | 0.0 | 1 |
| | 24 | 0.10 | 0 | 3361 | 37.2 | 0.0 | 1 | 0 | 6189 | 10.2 | 0.0 | 1 |
| | | 0.50 | 0 | 8857 | 10068.5 | 2.5 | 0 | 0 | 14901 | 215.8 | 0.0 | 1 |
| | | 0.80 | 6348 | 21149 | 10162.6 | 100.0 | 0 | 0 | 27857 | 591.4 | 0.0 | 1 |
| | | 1.00 | 14117 | 24242 | 10130.1 | 100.0 | 0 | 0 | 34531 | 915.9 | 0.0 | 1 |
| | 25 | 0.10 | 0 | 3887 | 10015.3 | 0.3 | 0 | 0 | 7466 | 60.7 | 0.0 | 1 |
| | | 0.50 | 0 | 12830 | 10083.4 | 76.8 | 0 | 0 | 19888 | 230.0 | 0.0 | 1 |
| | | 0.80 | - | - | - | - | - | 0 | 29531 | 587.1 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35631 | 1872.4 | 0.0 | 1 |
| | 26 | 0.10 | 0 | 4450 | 44.3 | 0.0 | 1 | 0 | 7523 | 21.8 | 0.0 | 1 |
| | | 0.50 | 0 | 8277 | 10050.9 | 2.9 | 0 | 0 | 13145 | 155.1 | 0.0 | 1 |
| | | 0.80 | 2238 | 20437 | 10113.4 | 100.0 | 0 | 0 | 27649 | 1115.4 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 32847 | 1291.7 | 0.0 | 1 |
| | 27 | 0.10 | 0 | 2881 | 228.2 | 0.0 | 1 | 0 | 5581 | 69.0 | 0.0 | 1 |
| | | 0.50 | 0 | 6943 | 10052.2 | 0.9 | 0 | 0 | 11883 | 180.1 | 0.0 | 1 |
| | | 0.80 | 907 | 17975 | 10135.7 | 100.0 | 0 | 0 | 24847 | 406.0 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 35397 | 1219.2 | 0.0 | 1 |
| | 28 | 0.10 | 0 | 4438 | 535.1 | 0.0 | 1 | 0 | 8098 | 54.6 | 0.0 | 1 |
| | | 0.50 | 0 | 9527 | 10087.5 | 2.5 | 0 | 0 | 16811 | 196.1 | 0.0 | 1 |
| | | 0.80 | 1613 | 20802 | 10158.6 | 100.0 | 0 | 0 | 27594 | 401.0 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 36884 | 1349.2 | 0.0 | 1 |
| | 29 | 0.10 | 0 | 4053 | 212.3 | 0.0 | 1 | 0 | 6991 | 63.7 | 0.0 | 1 |
| | | 0.50 | 215 | 12357 | 10078.9 | 100.0 | 0 | 0 | 12804 | 222.4 | 0.0 | 1 |
| | | 0.80 | 8784 | 19037 | 10112.0 | 100.0 | 0 | 0 | 26014 | 533.3 | 0.0 | 1 |
| | | 1.00 | - | - | - | - | - | 0 | 32722 | 895.0 | 0.0 | 1 |

# Curriculum Vitae

## Personal Details

| | |
|---|---|
| Name | Johannes Inführ |
| Date of Birth | October $29^{th}$, 1986 |
| Address | Kaposigasse 60, 1220 Vienna, Austria |
| E-Mail | infuehr@ads.tuwien.ac.at |

## Education

since 2010 **PhD studies in Computer Science**, *Vienna University of Technology*.
Dissertation on the topic of "Optimization Challenges of the Future Federated Internet"

2008–2010 **Master studies in *Computational Intelligence***, *Vienna University of Technology*.
Master thesis on the topic of "Automatic Generation of 2-AntWars Players with Genetic Programming
Master examination passed with distinction

2006–2008 **Bachelor studies in *Software & Information Engineering***, *Vienna University of Technology*.
Bachelor thesis on the topic of "Scatter Search"
Bachelor examination passed with distinction

2000–2005 **Upper Secondary School**, *School for higher technical education Donaustadt*, Vienna.
Department of computer engineering
Diploma thesis on the topic of 2-D motion capture
General qualification for university entrance passed with distinction

1996–2000 **Lower Secondary School**, *Polgarstraße*, Vienna.

1992–1996 **Elementary School**, *Essling*, Vienna.

## Work Experience

since 2011 **University Assistant**, *Algorithm and Data Structures Group, Institute of Computer Graphics and Algorithms, Vienna University of Technology*.

2010–2011 **Research Assistant**, *Algorithm and Data Structures Group, Institute of Computer Graphics and Algorithms, Vienna University of Technology*.

2005–2006 **Military Duty**, *Department of Performance Medicine, Van-Swieten-Kaserne*.

2001–2003 **Internship**, *Department of Information-management, Generali Holding Vienna AG (one month each year)*.

## Languages

German **native tongue**

English **fluent**

## Research Interests

My current research interests are telecommunication problems, with the focus on virtual network mapping. I studied various heuristic and exact methods for solving the offline variant of this problem and plan to extend those methods to solve online and dynamic variants, which are closer to the real world. Generally, I am interested in any method for automatic problem solving, like meta-heuristics or algorithms from the field of artificial intelligence. I find the emergent self-organization and evolutionary dynamics that can be observed in genetic programming fascinating.

## Publications


[Inf10] Johannes Inführ. Automatic Generation of 2-AntWars Players with Genetic Programming. Master's thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, July 2010. Supervised by G. R. Raidl.

[IR11] Johannes Inführ and Günther R. Raidl. Introducing the Virtual Network Mapping Problem with Delay, Routing and Location Constraints. In J. Pahl, T. Reiners, and S. Voß, editors, *Network Optimization: 5th International Conference, INOC 2011*, volume 6701 of *LNCS*, pages 105–117, Hamburg, Germany, June 2011. Springer.

[IR12] Johannes Inführ and Günther R. Raidl. Automatic Generation of 2-AntWars Players with Genetic Programming. In R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, editors, *Computer Aided Systems Theory – EUROCAST 2011*, volume 6927 of *Lecture Notes in Computer Science*, pages 248–255. Springer Berlin / Heidelberg, 2012.

[IR13a] Johannes Inführ and Günther R. Raidl. A Memetic Algorithm for the Virtual Network Mapping Problem. In H. C. Lau, P. Van Hentenryck, and G. R. Raidl, editors, *Proceedings of the 10th Metaheuristics International Conference*, pages 28/1–28/10, Singapore, 2013. Nominated for best paper award.

[IR13b] Johannes Inführ and Günther R. Raidl. GRASP and Variable Neighborhood Search for the Virtual Network Mapping Problem. In M. J. Blesa et al., editors, *Hybrid Metaheuristics, 8th Int. Workshop, HM 2013*, volume 7919 of *LNCS*, pages 159–173. Springer, 2013.

[IR13c] Johannes Inführ and Günther R. Raidl. Solving the Virtual Network Mapping Problem with Construction Heuristics, Local Search and Variable Neighborhood Descent. In M. Middendorf and C. Blum, editors, *Evolutionary Computation in Combinatorial Optimisation – 13th European Conference, EvoCOP 2013*, volume 7832 of *LNCS*, pages 250–261. Springer, 2013.

[ISH+13] Johannes Inführ, David Stezenbach, Matthias Hartmann, Kurt Tutschku, and Günther R. Raidl. Using Optimized Virtual Network Embedding for Network Dimensioning. In *Proceedings of Networked Systems 2013*, pages 118–125, Stuttgart, Germany, 2013. IEEE.