TU WIEN Informatics

Heuristic Approaches for Solving the Interdependent Lock Scheduling Problem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Peter Schmidt, BSc
Matrikelnummer 00526202

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Mag. Dipl.-Ing. Dr.techn. Matthias Prandtstetter
Dipl.-Ing. Ulrike Ritzinger, BSc, PhD
Dipl.-Ing. Dr.techn. Mario Ruthmair

Wien, 20. Dezember 2022

Peter Schmidt	Günther Raidl

TU WIEN Informatics

Heuristic Approaches for Solving the Interdependent Lock Scheduling Problem

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Peter Schmidt, BSc
Registration Number 00526202

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Mag. Dipl.-Ing. Dr.techn. Matthias Prandtstetter
Dipl.-Ing. Ulrike Ritzinger, BSc, PhD
Dipl.-Ing. Dr.techn. Mario Ruthmair

Vienna, 20th December, 2022

Peter Schmidt | Günther Raidl

Erklärung zur Verfassung der Arbeit

Peter Schmidt, BSc

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.

Wien, 20. Dezember 2022

Peter Schmidt

Danksagung

..., aber lass dir nicht zu lange Zeit. Schreib das gleich, solange du noch dabei bist und alles frisch ist.

Über die Worte meiner Kollegin musste ich damals noch schmunzeln. Inzwischen habe ich eine Familie gegründet. Zwei Kinder, ein Haus, einen Job. Und meine Diplomarbeit liegt Jahre später endlich vor mir. Das wäre definitiv auch einfacher und schneller gegangen, wenn nicht ständig irgendetwas "wichtiger" gewesen wäre. *Ach, das hat ja noch Zeit. Jetzt muss ich erst mal ...*

Mein Dank an dieser Stelle gebührt daher meinem Betreuer Professor Günther Raidl, der in den vielen Jahren trotz Unterbrechungen unentwegt ohne ein böses Wort für mich da war mit Unterstützung, Korrekturen und Ratschlägen. Ohne ihn gäbe es diese Arbeit nicht und das nicht nur aus organisatorischer Sicht.

Durch meine Fokussierung auf Algorithmen und Optimierungsverfahren im Zuge meiner Ausbildung an der TU Wien durfte ich neben Günther Raidl auch Matthias Prandtstetter, Mario Ruthmair und Ulrike Ritzinger kennen lernen. Sie haben als Vortragende großen Einfluss auf mich ausgeübt und die Richtung für meine weitere Laufbahn aufgezeigt. Als sich mir dann die Gelegenheit bot mit ihnen gemeinsam an einem praxisorientierten Optimierungsproblem zu arbeiten, konnte ich nicht nein sagen. Ich möchte mich an dieser Stelle daher ganz besonders bei ihnen und dem Austrian Institute of Technology bedanken: für die gute professionelle Zusammenarbeit, die Unterstützung, den angeregten Austausch, die Diskussionen und den Einblick in den praktischen Forschungsbetrieb. Besonders Ulrike Ritzinger sei hier nochmals hervorgehoben, die mich nach vielen Jahren Funkstille noch mit wertvollen Daten und Informationen versorgen und mir wichtiges Feedback zu meiner Arbeit geben konnte.

Ich möchte mich auch bei meinem Kollegen Sebastian Knopp für seine konstruktiven Anregungen und das Aufzeigen neuer Möglichkeiten und Wege bedanken. Er hat für die Evaluierung meiner Experimente wichtige Tipps für mich bereit gehabt und so maßgeblich zur Qualität dieser Diplomarbeit beigetragen.

Der Zufall spielt nicht nur bei randomisierten Algorithmen eine essentielle Rolle, sondern auch im Leben. Hätte ich an meinem ersten Tag an der Universität nicht zufällig diese drei Kollegen getroffen, wäre ich heute wahrscheinlich nicht da, wo ich bin. Lukas Crepaz, Alexander Eibner und Raimund Machacek waren nicht nur Studienkollegen, sondern

über viele Jahre tägliche Wegbegleiter und gehören heute mit ihren Familien zu meinen wertvollsten Freunden. Ich danke ihnen für die schöne Zeit, die wir hatten, die vielen Stunden, die wir gemeinsam gelernt, programmiert und gearbeitet haben.

Meine Familie hat mir im Leben immer starken Rückhalt und Sicherheit, aber vor allem auch die Freiheit gegeben, ohne die ein neugieriger junger Geist sich nicht in diesem Maße hätte entfalten können. Danke an meine Eltern, dass sie meinen Werdegang immer beobachtet, aber nie Fortschritte eingefordert und mich immer bedingungslos unterstützt haben. Ich hoffe, meinen Kindern einmal auch so viel Vertrauen und Liebe geben zu können und ihnen ein sicherer Hafen für ihre Expeditionen ins Leben zu sein. Ich danke auch meiner Großmutter, die mir mit ihrer fast schon kindlichen Offenherzigkeit und Neugierde bis ins hohe Alter immer imponiert hat (*Peter, wie ist das mit diesem Internet?*).

Quidquid agis, prudenter agas et respice finem.

Der größte Dank gebührt aber meiner Frau. Kerstin hat mich (nicht nur im Zuge dieser Arbeit) durch alle Höhen und Tiefen begleitet. Sie war und ist mein Antrieb, meine Motivation. Sie hat es geschafft, dass ich mich wieder und wieder hinsetzte, hat sich wochenlang um alles gekümmert um mir die Zeit zu geben diese Arbeit zu schreiben. Vielen Dank für deine Geduld und Unterstützung.

Acknowledgements

..., but don't loose too much time. Write that now, as long as you are on it and everything is still fresh.

I had to smile benignly about my colleague's words back then. In the meantime I have started a family. Two kids, a house, a job. And years later my diploma thesis now finally lies in front of me. This definitely could have been easier and faster, if not every now and then something "more important" had come along. *Yeah, there is still time. First I have to ...*

Therefore, my gratitude is owed to my advisor Professor Günther Raidl, who in all those years despite the discontinuity was always there for me, without a bad word, but with support, corrections and advice. Without him, this thesis would not exist, not only from the organisational aspect.

Due to my focus on algorithms and optimization techniques in the course of my education at the TU Vienna, aside from Günther Raidl I was able to meet Matthias Prandtstetter, Mario Ruthmair and Ulrike Ritzinger. As lecturers they had a strong influence on me and showed me the way for my career. When I got the opportunity to work with them together on a practical application of an optimization problem, I could not say no. I would like to thank them and the Austrian Institute of Technology for the good and professional cooperation, the support, the inspiring exchange, the discussions and the insight in practical research operations. Especially I want to thank Ulrike Ritzinger, who, after many years of radio silence, still provided me with valuable data and information and gave me important feedback to my work.

I want to thank my colleague Sebastian Knopp for his constructive suggestions and pointing me to new ways and possibilities. He had important advice for evaluating my experiments and significantly contributed to the quality of this thesis.

Coincidence does not only play an essential role in randomized algorithms, but also in life. If I had not met those three fellows on my first day at the university, I would likely not be where I am now. Lukas Crepaz, Alexander Eibner and Raimund Machacek were not only university colleagues, but over several years daily companions and today together with their families they belong to my closest friends. I am thankful for the good time we had, many hours of learning, programming and working together.

In my life my family always gave me the backing and safety, but most of all freedom, without which a curious young mind would not have been able to unfold. I want to thank my parents, that they have always watched my career but never demanded any achievement and instead supported me unconditionally. I hope one day I have the same trust in my children and that I will be able to give them so much love that I can be a safe harbor for their journey and expeditions in life. Further, I'd like to thank my grandmother, who always impressed me with her almost childlike frankness and curiosity up to her old age (*Peter, what about this internet?*).

Quidquid agis, prudenter agas et respice finem.

Above all, my biggest thanks go to my wife. Kerstin has always (not only throughout the work on this thesis) accompanied me through all ups and downs. She was and is my drive and motivation. She managed to get me working on this thesis over and over again, she took care of everything around to give me the time I needed to write this thesis. Thank you so much for your patience and support.

Kurzfassung

In der vorliegenden Diplomarbeit wird das verflochtene Schleusenplanungsproblem (Interdependent Lock Scheduling Problem) präsentiert. Es ist ein komplexes Optimierungsproblem, bei dem Schiffe zu Schleusungen an aufeinanderfolgenden Staudämmen entlang ihrer Route zugewiesen werden. Innerhalb einer Schleusung muss darüber hinaus die Reihenfolge der Schiffe spezifiziert werden. Die Komplexität des Problems entsteht dabei durch die Abhängigkeit zwischen aufeinanderfolgenden Schleusungen am selben Staudamm und die Verflechtung zwischen Schleusungen an benachbarten Dämmen, welche dieselben Schiffe befördern. Nach einer detaillierten Beschreibung des Problems im Allgemeinen und der spezifischen Situation an der Donau in Österreich werden existierende Lösungsansätze für ähnliche oder verwandte Probleme analysiert. Mehrere deterministische und randomisierte Konstruktionsheuristiken, sowie verschiedene Nachbarschaftsstrukturen werden vorgestellt, die letztendlich in unterschiedlichen Metaheuristiken zur Anwendung kommen. Als Datenstruktur für den Schleusenplan dient dabei ein Abhängigkeitsgraph, der die Abhängigkeiten zwischen den einzelnen Schleusungen verwaltet und es somit erlaubt bei Änderungen am Plan Neuberechnungen auf ein notwendiges Minimum zu reduzieren. Um anpassbare VNS Konfigurationen jenseits von einfachen Variable Neighborhood Descent oder Token-Ring Neighborhood Search zu erstellen wird ein neues Framework namens Neighbourhood Search with Configurable Neighbourhood Iteration eingeführt. Konstruktionsheuristiken und unterschiedliche VNS Konfigurationen werden anhand von zwei vielfältigen Sets von Probleminstanzen getestet: kleine synthetische Testinstanzen und große Instanzen mit Echtdaten, die auf Basis von Verkehrsdaten ganzer Tage erstellt wurden. Experimente zeigen, das das Interdependent Lock Scheduling Problem mit heuristischen Verfahren effektiv und mit annehmbaren Ressourcenverbrauch gelöst werden kann.

Abstract

This thesis introduces the Interdependent Lock Scheduling Problem. It is a complex optimization problem where ships need to be assigned to a lockage operation at consecutive water gates along their journey. Within a lockage operation, the order of ships has to be specified. The complexity of the problem arises from the interdependency between subsequent lockage operations at a single water gate and dependent operations at neighboring gates that serve the same ships. After a detailed description of the problem in general and of the specific situation at the Austrian Danube, an analysis of existing approaches to similar or related problems is conducted. Several deterministic and randomized construction heuristics are proposed as well as various neighborhoods structures that are eventually used in different metaheuristics. A dependency graph serves as backing data structure of the schedule, keeps track of the interdependence between lockage operations and allows to reduce the necessary recalculations when altering the schedule to the necessary minimum. To create highly customizable VNS configurations beyond simple Variable Neighborhood Descent or Token-Ring Neighborhood Search, a framework called Neighbourhood Search with Configurable Neighbourhood Iteration is introduced. Construction heuristics and different configurations of VNS are tested on two diverse sets of instances: small synthetic test instances and large real life instances that are build on a full days traffic information. Experiments show that the Interdependent Lock Scheduling Problem can be solved effectively with heuristic methods using a reasonable amount of computing resources.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Next to trains and trucks, inland navigation has a big share in overland transportation. Its importance is increasing as it is seen, for many applications, as the most efficient mean of transport with respect to ecological objectives [DHH+13]. The European Commission therefore shows efforts to shift transport volumes to trains and inland navigation [Eur11][Eur20].

From the increasing importance of transportation and its high ecological and economical impact arises the necessity of optimizing its processes. While container terminals, i.e., the end points of inland navigation, have been extensively researched [SV08], during the past years, optimizing processes at single locks have gained focus [VB09],[CS11]. But a major aspect of inland navigation has still been neglected in scientific research: Optimizing operations at several locks along a water way in combination. Inland navigation – in contrast to other means of overland transportation – uses a mainly natural network of routes (i.e., rivers), that cannot easily be adapted or expanded – unlike rails or roads. Therefore, from the economic point of view, it is not only important to reduce transportation costs, but also to use existing capacities as efficient as possible.

From European rivers, Danube and Rhine are most relevant, as they form long routes through several countries of the European Economic Area. The Austrian part of the Danube is especially challenging to transport planning, as vessels traveling along the 350 km have to pass nine water gates due to the intensive use of water power in Austria.

These water gates are bottlenecks for inland navigation. While the river as such allows an increase in transport volume (cargo volumes on the Danube are only 10 % – 20 % of those carried on the Rhine [Eur10]), the water gates are likely to cause congestion (with 27 % – 48 % lock chamber utilization [via20]), which need to be avoided by efficient lock management and scheduling.

The project "*imFluss*", partially funded by the Bundesministerium für Verkehr, Innovation und Technologie (Ministry for Transport, Innovation and Technology) (BMVIT) within the strategic programme I2VSplus under grant 835771, tried to find alternatives to the currently applied first-come-first-served policy at each individual water gate. It aimed to optimize overall scheduling s.t. the overall travel time of all ships is reduced. This requires a schedule of all ships and water gates at the river at a larger scope, to allow coordination of lockage operations at the water gates. A huge beneficial side effect of this is that by scheduling vessels along their total route lockage times at water gates are known in advance. By further dissolving the dependency of service times on arrival times (as given by a first-come-first-serve policy), vessel speeds and thus arrival times can be adapted in order to reduce waiting time, congestion, emissions and costs.

imFluss was conducted by the Austrian Institute of Technology (AIT) in cooperation with via donau – Österreichische Wasserstraßen-Gesellschaft mbH. The aim of the project was to analyze the scheduling problem and finding formal definitions for it, as well as developing exact and heuristic solution methods that can be used to simulate and optimize lockage schedules in a daily scope. Parts of this thesis have previously been published in [PRSR15].

1.2 Problem Statement

The Interdependent Lock Scheduling Problem (ILSP) is introduced and a formal definition is presented. The problem description is based on the situation on the Austrian part of the Danube and reflects local conditions and circumstances. To give an initial insight on the problem of scheduling vessels across several locks, we first present an overview on the simpler Lock Scheduling Problem.

1.2.1 The Lock Scheduling Problem

The optimization problem dealing with the management of a single water gate or lock is called the Lock Scheduling Problem (LSP) [VB09] or Lock Masters Problem [CS11]. Locks are installed in embankment dams that are used to assure tidal independence of a seaport, to overcome height differences along a waterway, or for generating power from water. They consist of one or multiple parallel lock chambers of possibly different dimensions. Depending on the water gate, parallel chambers can be operated independently or only sequentially. Some locks further allow splitting or joining of lock chambers. During a single lockage operation several vessels can be transferred together, depending on their dimensions and possibly legal constraints.

The LSP is closely related to batch scheduling problems with setup times and job compatibilities [CS11]. The machine (i.e., the lock) can handle several compatible jobs (i.e., ships traveling in the same direction) simultaneously, as long as its capacity (i.e., the lock chamber dimensions) suffices. The job release dates correlate to the ships arriving times, the setup time to the time needed by the ships to enter the chamber, and the jobs

processing times to the lock chambers time needed to change its level. Further there is a post-processing time, i.e., the time required by the ships to leave the chamber.

In the simple case, where only ships in one direction need to be served, the time needed to bring the lock chamber back to its initial level can be seen as additional post processing or setup time. But usually locks serve ships traveling in both directions, so in the next batch the batch machine can either handle jobs of the same direction again, which causes the mentioned additional post processing time, or handle jobs of the inverse direction, avoiding that penalty.

1.2.2 The Interdependent Lock Scheduling Problem

The LSP and its solution techniques are sufficient to cope with scheduling problems on single water gates, like in tide independent ports. When scheduling vessels along a route that passes several water gates, the problem becomes more complex due to the interdependence of lockage operations on neighboring water gates.

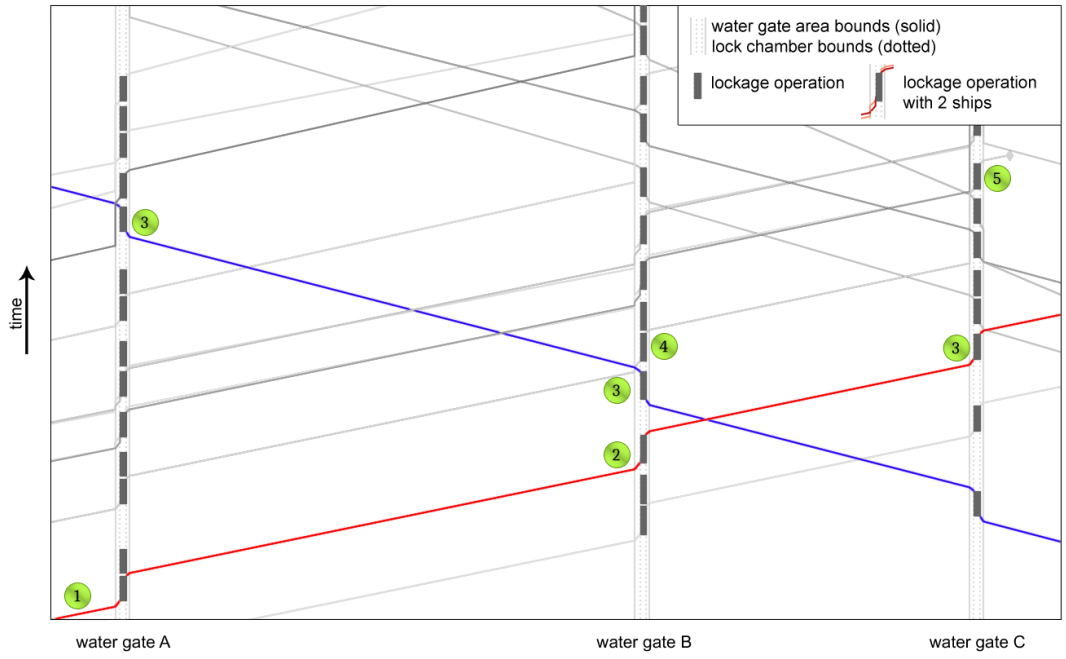


Figure 1.1: Detail of a schedule visualization showing interdependency.

We assume a combined schedule S of three neighboring water gates A, B, and C, as shown in Figure 1.1. Whenever the schedule of A changes s.t. any ship s (red) traveling from A to C arrives at B at a later time (1), the schedule of water gate B changes as well, if the introduced additional delay of s is bigger than its waiting time at B (2). Such an update to the schedule of B could either mean that the lockage operation at B serving s needs to be delayed, which possibly further delays subsequent lockage operations at

B (and consequently at C and A) (3), or that the ship s needs to be served by the lock B in a later lockage operation (4). In the former case, by delaying subsequent lockage operations, another ship s' (blue) traveling from B to A might get delayed, which might change the schedule of A again (3), such that the schedules of A and C need to be updated. The latter case will introduce a possibly huge delay for the ship s, which is usually not preferable and would still change the schedule of the third water gate C (5). Due to this characteristic of mutual influence and dependency between operations at water gates, we call the problem of scheduling vessels along a route with several water gates the Interdependent Lock Scheduling Problem (ILSP) [PRSR15].

It is important to point out that a change in the arrival time of a ship that triggers the above explained cascade of changes can have its source not only from direct, explicitly applied scheduling decisions concerning that ship. Obviously any change to a lockage's operation time affects all assigned ships and therefore their arrival times at the next gates, but also simply adding another ship to an existing operation affects the other ships departure times, because of constraints defined in Subsection 1.2.3.

The ILSP is proven to be at least NP-complete by Passchyn in [Pas16]. It is strongly related to Job-shop scheduling problems with batch processing, job compatibilities, release times, non-uniform setup times, transportation delays. It can approximately be written in α, β, γ notation as: $Jm|batch, comp, r_j, t_{jk}|\sum C_j$. As many Job-shop scheduling problems are proven to be NP-hard (see [BSW07] for an overview), the ILSP is likely at least NP-hard as well.

The ILSP at the Austrian Danube

In the special case of the Austrian Danube the problem gets even more complex, as it is in fact a multi objective optimization problem. The embankment dams (and thus the water gates) are built for energy production and whenever the locks are operated, retained water and thus energy gets lost. Another issue arises during dry seasons when the water level in the river is low. A decrease in water level reduces the fairway depth and width and can finally require the vessels to reduce their capacity utilization. To maintain or increase fairway depth and navigability of the river sections, water needs to be saved. The water gate operators are therefore highly interested in reducing the number of lockage operations, but still need to serve passing vessels. While the LSP and ILSP usually try to minimize the total waiting time of ships to optimize travel flow, it is in case of the Austrian Danube also important, to keep the number of lockage operations as small as possible.

Another property of the Austrian Danube's water gates is that all water gates are identical in terms of capacity and operation mode. They consist of two parallel lock chambers of equal size that can only be operated sequentially (i.e., no two lock chambers can alter their level at the same time). The only exception to the rule is the water gate Freudenau in Vienna, whose chambers are slightly longer and which further allows splitting of one of its lock chambers, but these features are neglected for the scope of this thesis.

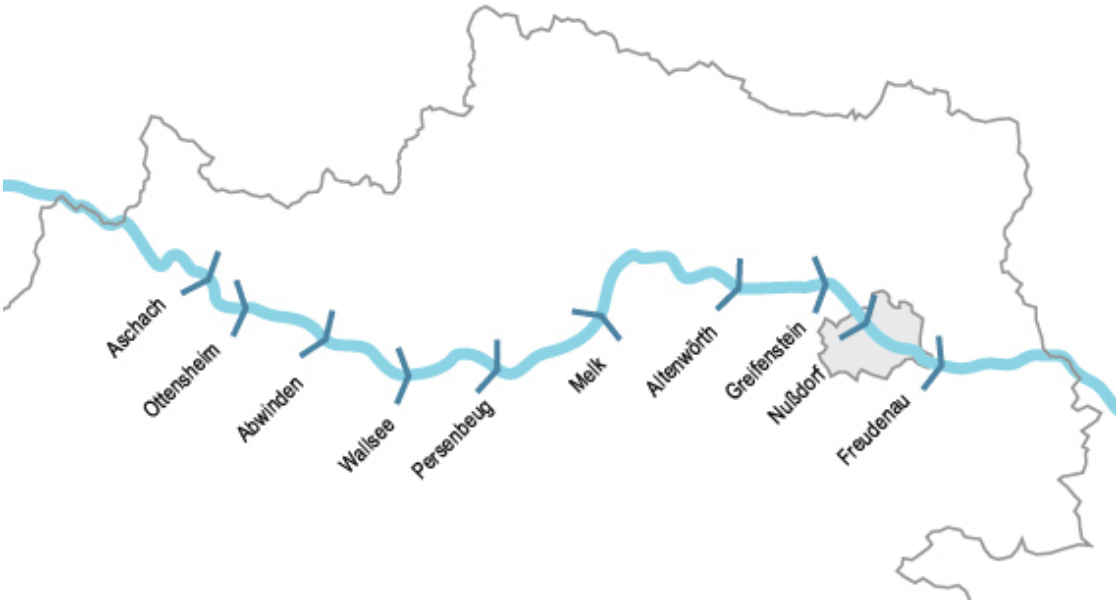


Figure 1.2: Austrian part of the Danube with sketched water gates. Parts of the river between two water gates (or the state border) are referenced as sections. ©viadonau and DoRIS (`http://www.doris.bmvit.at`)

1.2.3 Formal Definition

The ILSP was first defined in [PRSR15] with the according Mixed Integer Programming (MIP) model presented at [RRP15]. For this thesis the problem is slightly adapted and further constrained as additional and more detailed instance data became available.

In the ILSP we are given a set of m water gates $\mathcal{G} = \{1, \ldots, m\}$, successively arranged along the river from east (downstream) to west (upstream). The water gates and national borders divide the river into $m + 1$ sections $\mathcal{R} = \{1, \ldots, m + 1\}$, see Figure 1.2. Let $\mathcal{S}$ be the set of ships, partitioned into ships $\mathcal{S}^+$ going upstream and ships $\mathcal{S}^-$ going downstream, i.e., $\mathcal{S} = \mathcal{S}^+ \cup \mathcal{S}^-$, $\mathcal{S}^+ \cap \mathcal{S}^- = \emptyset$. Along its trip each ship $s \in \mathcal{S}$ passes one or more successive water gates $\mathcal{G}_s \subseteq \mathcal{G}$, either up- or downstream, and traverses a number of successive river sections $\mathcal{R}_s \subseteq \mathcal{R}$.

Each gate $g \in \mathcal{G}$ consists of two identical, not concurrently operable lock chambers. The length of a lock chamber is denoted by κ_g and for simplicity denotes the only capacity constraint for lockage operations, i.e., ships are arranged consecutively along their direct axis inside the lock chambers. The ships' positions in a lock chamber define their entering and exiting sequence. Overtaking inside the chamber or the vicinity around the damn (depending on the water gate between 220 and 360 meters) is forbidden. The chambers eastern and western location along the river and eastern and western area bounds are denoted as c_g^e, c_g^w, a_g^e, and a_g^w respectively (see Figure 1.3). From the eastern and western area bounds and national borders, the length of the river sections r_i can be derived. A

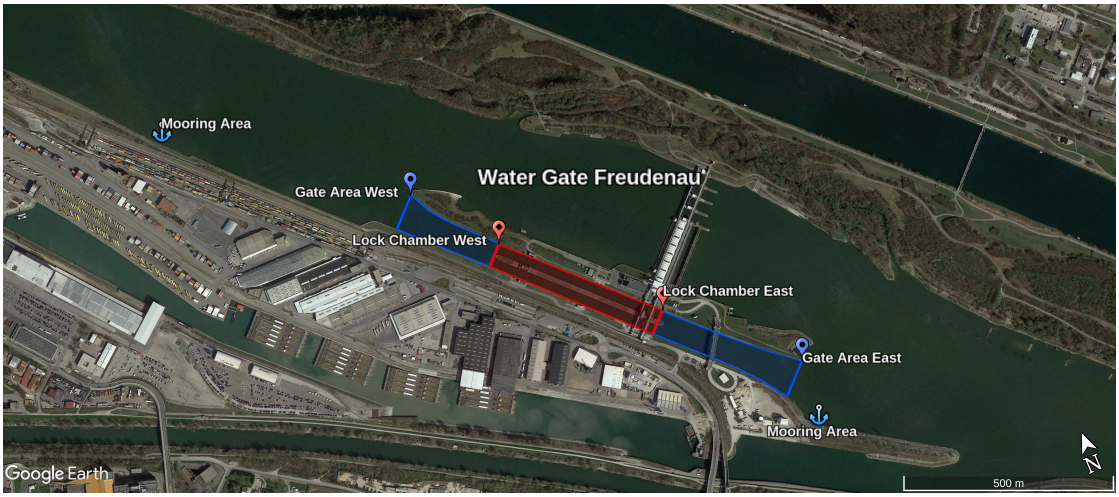


Figure 1.3: Satellite view of the water gate Freudenau with overlays showing the lock chamber area (red), water gate area (blue), and mooring places.
Created with Google Earth, (`https://earth.google.com`)

lockage operation $l_{i_g} \in \mathcal{L}_g$ at water gate g can either go upward or downward, depending on the water gate's current status. The time required to fill or empty the chamber is given by τ_g. For the scope of this paper and the proposed solution to the ILSP, it is irrelevant which individual lockchamber has which position (filled or empty). The only relevant information is the *water gate's status*, which can be *FULL* (both chambers are up, the water gate cannot serve another upward lockage, the next operation needs to go down), *EMPTY* (the inverse situation, the next operation needs to go up), or *NEUTRAL* (one chamber is up, the other one is down, the next operation can serve any direction).

The trip of ship $s \in \mathcal{S}$ has a start time (or release time) r_s and location λ_s and a required target position δ_s. There are no scheduled stops or breaks. Information about the ship length k_s, the ship type t_s, and the travel speed v_{s_i} in river section i is available. According to v_{s_i} and the distances between water gates, the constant in-motion travel time mt_{s_i} for each ship on each river section i can be determined. The time required by the ships to enter and exit each water gate g is denoted as $e^n_{s_g}$ and $e^x_{s_g}$ respectively. It includes mooring inside the chamber and for simplicity is independent of the ships position outside the area or inside the chamber.

During a vessels trip along the river, several events occur around water gates. We define the following times related to these events.

- The *arrival time* (denoted $at^a_{s_g}$) at a water gate defines the time that the ship arrives at the surrounding area of the gate, i.e., the ship arrives at a^e_g or a^w_g respectively.
- The *entering time* (denoted ent_{s_g}) at a water gate g defines the time that the ship s starts entering the surrounding area and the lock chamber.

- The *lockage time* (denoted z_{i_g}) defines the time at which the lockage operation i at water gate g begins.
- The *exiting time* (denoted ex_{s_g}) defines the time at which the ship s starts leaving the lock chamber at water gate g.
- The *departure time* (denoted dt_{s_g}) defines the time at which the ship s has left the surrounding area at water gate g.

A solution to the ILSP is represented by a set of lockage operations $\mathcal{L}_g$ for each water gate $g \in \mathcal{G}$. Each operation $l_{i_g} \in \mathcal{L}_g$ has a starting time z_{i_g} and a set of ships S_{i_g} assigned to it. The ship assignment also defines the ships position $p_{i_{g_s}}$ inside the lock chamber.

A feasible solution to the ILSP needs to adhere to the following constraints:

- Each ship $s \in \mathcal{S}$ must be assigned to exactly one lockage operation at each gate $g \in \mathcal{G}_s$.
- All ships $s \in \mathcal{S}$ have to arrive at their target position δ_s before planning horizon H.
- Only upstream going ships $s \in \mathcal{S}^+$ can be assigned to an upward lockage operation, whereas only downstream going ships $s' \in \mathcal{S}^-$ can be assigned to a downward lockage operation.
- The sum of lengths of all ships s assigned to the same lockage operation l_{i_g} must not exceed the length of the lock chamber κ_g.

For each lockage operation a starting time z_{g_i} has to be set, based on the following limitations:

- A simultaneous operation of both lock chambers at a water gate is not possible, so $z_{g_i} + \tau_g \leq z_{g_{i+1}}$.
- For any two successive operations at water gate g, the ships of the latter operation cannot enter the area $a_g^e - a_g^w$ until all exiting ships of the former operation have left the area. (The actual used lockchamber is irrelevant for this constraint as the area around the water gate is used by ships entering or leaving both lock chambers.) For simplicity, all ships wait at the same position and we assume the mooring areas have an infinite capacity.
- A lockage operation l at g can start earliest when the preceding lockage operation at g was either empty and has finished, or all ships have left the chamber. As the model and instance data only hold data about leaving the whole water gate area, and not just the chamber (i.e., the ship s' exiting duration $e_{s_g}^x$), the lockage operation can start earliest after the last ship's departure time of the preceding lockage operation. Further, if it is not an empty operation, all ships that are transported by this operation must have finished entering.

- Ships are only allowed to enter (or exit) the lock chamber one after the other according to their assigned position in the lock chamber for safety reasons and space limitations at the gates.

These constraints simplify the problem, as in a real world application, entering, leaving or operating the chambers could be allowed earlier than defined here, especially when two successive operations have the same direction. As those two operations obviously need to use different chambers, entering the second chamber would already be possible as soon as entering the first chamber is done. Also operating the second chamber would be possible immediately after the first chambers operation is finished, even though the ships have not yet left the first chamber. For simplicity and to keep the heuristic comparable with the mathematical model by Prandtstetter et al., we adhere to the reduced constraints mentioned above.

Adhering the constraints can lead to waiting times for ships at the water gates. Assuming the ships travel at their highest possible speed per river section v_{i_s}, the in-motion time is constant for each ship s. The total travel time tt_s of a ship s is the sum of constant in-motion time mt_s, constant time ct_s spent entering, waiting for operation of, and exiting the chambers, variable waiting time $wt^c_{s_g}$ inside the chamber (waiting for other ships to finish their entering/exiting process), and variable waiting time $wt^a_{s_g}$ outside the water gate area (waiting for permission to start entering the lockchamber).

$$tt_s = mt_s + ct_s + \sum_{g \in \mathcal{G}_s} (wt^a_{s_g} + wt^c_{s_g}), \qquad \forall s \in \mathcal{S} \tag{1.1}$$

The optimization goal of the ILSP is to find a feasible solution such that the sum of total travel times of all ships is minimized, implicitly minimizing the sum of variable waiting times over all ships. The second weighted term in the objective function is the total number of lockage operations, which needs to be minimized since each lockage operation reduces the amount of water available for electricity production.

$$min \quad c_1 \cdot \sum_{s \in \mathcal{S}} \sum_{g \in \mathcal{G}_s} (wt^a_{s_g} + wt^c_{s_g}) + c_2 \cdot \sum_{g \in \mathcal{G}} (|\mathcal{L}_g|) \tag{1.2}$$

The factor c_1 weights the waiting time and c_2 the number of lockage operations. We define the vector of cost factors $C(c_1, c_2)$.

1.3 Structure of the Work

The remainder of the thesis is structured as follows:

In Chapter 2, an overview of scientific research on inland navigation is given. It focuses not only on solution techniques to related scheduling problems, but also on interdependence and other aspects of the topic.

The scientific methods are described in Chapter 3. This chapter shows how the literature research was conducted, data model and algorithms were designed and how experimental results are compared and interpreted.

Chapter 4 provides detailed insight in different algorithms and data structures that where developed to solve the ILSP.

Experimental results are shown and discussed in Chapter 5.

Finally, Chapter 6, gives an overview, final thoughts on the proposed algorithms and some clues for future work in the field of lock scheduling.

CHAPTER 2

State of the Art

Gathering background information about the problem and possibly finding solutions to similar problems in literature helps not only to understand the problem as such but can also inspire new solution techniques.

2.1 Literature Studies on Inland Navigation

The initial research on related work showed that literature on the ILSP is limited. Scientific work on inland navigation is done under different view points and thus different problems are being identified and addressed.

The inland waterway network of the USA has mainly been analyzed under the perspective of congestion prevention and avoidance. On the Upper Mississippi River (UMR), barges are usually joined together into huge tows that are too large to be transferred by locks in a single operation. They have to be split up into smaller groups, transferred sequentially and need to be rejoined again at water gates along their voyage. Early work on the American inland navigation was done by Joseph S. de Salvo [SL68] in 1968. He analyzed the types of delay that towboats suffer in the Lower Mississippi, Upper Mississippi, Ohio and Illinois rivers, identifying lockage operations (including waiting time) as the main cause of delay taking 7.29 % of active time. De Salvo gives measurements to create a statistical model of delay times at locks, based on probability distributions of arrival times of ships at locks. From this statistical point of view, he assumes that operations at any two locks are independent, as long as congestion at one lock does not require a previous lock to stop serving.

Martinelli and Schonfeld in [MS95] introduce the interdependence coefficient of a series of locks, which defines the ratio of the total systems delay to the sum of the delay of all locks acting in isolation, to help in planning of investments to infrastructure improvements. In [TS98], Ting and Schonfeld propose a heuristic decision algorithm for tow dispatching at

a lock, considering estimated waiting times of the subject tow at the subsequent lock. Their results show that considering adjacent locks reduces the delay, but the relative benefit decreases as the distance between locks increases. In [TS99] they further show, how reducing speed to avoid waiting times affects the total travel costs of tows and provide methods to calculate the optimal speed. In their works, the authors show that the first come first serve (FCFS) policy, which is widely used in practice, yields a higher system wide delay, while other policies like shortest processing time first (SPF) may introduce a level of unfairness. In [TS01], Ting and Schonfeld add a fairness constraint to their SPF policy by limiting the number of tows that can overtake a waiting tow. The modified policy retains most benefits of the original policy, without sacrificing fairness, depending on the specified threshold.

The basic LSP is well described and analyzed by Coene et al. [CS11], who propose a polynomial time dynamic programming (DP) algorithm. In the basic case, where a single lock with a single lock chamber serves the ships coming from two directions, the instance can be divided into two sub problems, whenever there is enough time between ship arrivals, so the chamber can reach any position before and after the split. Without any further constraints, the LSP is closely related to batch scheduling problems with two job families/compatibilities (corresponding to the directions of the ships) which have to be served in alternation (with possibly empty batches). Coene et al. further show, that their algorithm can be extended to deal with non-uniform lockage times, capacity constraints (for uniform ships) and a constant number of parallel chambers with identical lockage times. If the number of chambers is part of the input, the problem becomes NP-hard.

An extension of the LSP is presented by Verstichel and Vanden Berghe in [VB09], where a single lock consists of multiple parallel chambers of different sizes. In addition to the basic scheduling problem, the authors further solve the problem of placing the ships inside the chamber and differ between two priority classes of ships. Beside minimizing the waiting times for the two classes of ships, the third objective is to minimize the total number of lockages. Therefore they use a construction heuristic that assigns the ships to chambers of appropriate size (resulting in a chronologically ordered list of ships per chamber type and direction), place the ships into the chambers according to a filling heuristic, and finally generate lockage operation events. The initial solution is then improved by a metaheuristic combining several local neighborhoods: merge/split subsequent lockages at the same chamber, change chamber for a single ship (changing the ships relative position in the list by a maximum range r), shifting a single ship in the same chambers list (constrained by the a maximum range r'), and swapping ships (from any chamber, within a maximum range r''). The compared metaheuristics are Variable Neighborhood Search (VNS), Multiple Neighbourhood Search (MNS) (i.e., searching all neighborhoods separately, using the best of all solutions for the next iteration), and the Composite Neighbourhood Search (CNS) (i.e., exploring the neighborhood containing all candidate solutions generated by different neighborhoods). The metaheuristics were further compared under two different acceptance criteria: best improvement (BI) (i.e., always take the best candidate solution), and late acceptance (LA) (i.e., keep a list L of

the most recent solutions; accept a new solution as long as it is better than the oldest solution in L). The LA list allows the heuristic to temporarily worsen solution results, enabling it to overcome local optima. Experiments on several instances with 100 or 1000 ships and an upstream fraction of 50 % and 30 % show, that LA always yields better results than BI (which is in fact equal to LA with $|L| = 1$), but the optimal length of L after which no significant improvement in solution quality can be found highly depends on the test instance and metaheuristic used. In general the authors conclude, that even an LA list of size two has a positive effect and computational costs are linear in the size of L. While no significant differences in solution quality between the compared metaheuristics can be attested, VNS seems to slightly outperform the others.

Recent work of Verstichel focuses on the (generalized) LSP in more detail. In his PhD thesis [Ver13] he defines the LSP as consisting of three sub problems: chamber assignment (assignment problem, constrained by ship dimensions and draught), lockage scheduling (parallel machine scheduling with sequence dependent setup times, release dates, time windows and different process speeds) and ship placement (2D bin packing problem with additional mooring constraints). Additionally he constrains the whole problem by a FCFS policy. He proposes a mixed integer linear programming model to the LSP and in experiments shows the impact of different lock chamber configurations on the objective value, depending on inter arrival times. Focusing on the ship placement problem, he further proposes a multi-order best-fit construction heuristic for the ship placement problem in a single lockage. Live tests showed, that the optimal solutions sometimes put ships in strange, uncommon positions in the lock chamber, possibly allowing additional ships to be transferred, even if no further ships were scheduled for the lockage. These plans would not have been accepted either by the lock master or the captains, not only because of the uncommon positioning, but also because feasibility and quality were hard to attest by humans. Further, the computation time was unpredictable and thus not favorable in a time constrained environment. The heuristic algorithm on the other hand performed well with an average optimality gap of 3.24 % in a short and stable processing time (below 6 ms). Its decisions for placing ships where much more natural and easier to understand, enabling easy attestation of solution quality and feasibility by the lock master.

Another interesting project under research is the scheduling of vessels at the Three Gorges Project and Gezhouba Dam (TGP-GD). This system of locks is more complex because of their different operation styles. While the Gezhouba Dam (GD) is a rather usual dam with three bi-directional lockages, the lockage system of the Three Gorges Project (TGP) consists of two parallel five-step locks with a fixed direction (in normal operation mode). Because of the two locks and their close location to each other (they are 38 km apart, whereas the next dam is about 700 km upstream), the problem at the TGP-GD is usually treated as a co-scheduling problem.

In [ZFY08], Zhang et al. propose a co-evolutionary algorithm, designing a queuing network consisting of several stations (the dams), servers (the lock chambers) and buffers (the mooring places for each direction at each dam). The algorithm is based on a discrete

time model and each lock chamber makes independent decisions for the next action at some time t_b. Therefore the costs for possible actions at any time $t \in [t_b, t_b + T_p]$ are calculated, where T_p is a predefined prediction period, which is set to either the transfer time through three of five chambers of the TGP lock system or the service time plus the rollback time of the GD lock chambers. The decision strategies for each lock are parametrized by a set of real numbers. Their values are optimized by a genetic algorithm that is used to co-evolve the decisions of the locks together.

At the TGP-GD, vessels have to announce their arrival at any dam 24 hours in advance and thus give their estimated time of arrival (ETA) at each of the two locks. The algorithm uses these ETAs to schedule the ships. This results in an estimated waiting time that is used as a factor for the cost function (beside the ships size and its priority). When looking at the used formulas in detail, one can see that in fact the two dams schedules are evaluated independently concerning the waiting time, as its calculation is based on the ETA for each lock which is known in advance. So a decision at the first lock does not change the ETA at the second dam and therefore does not necessarily change the waiting time there. This strongly reduces the level of interdependence between the two locks.

In [ZYY08], Zhang et al. analyse the situation at the TGP-GD given the problems similarity to the Flexible Manufacturing System, which is NP-complete. They present an exact model and propose a heuristic algorithm based on simulated annealing with local search. The solution space is divided in local domains, defined by lock chamber operational status and service direction (upstream, downstream, none) for a given operation. These values are constant for the domain and simulated annealing is used to find local optimal values for the service times. The problem is separated in a lock-operation scheduling problem (U; which lock operates when and in which direction) and a ship-dispatch scheduling problem (K; which ship takes part in the locking operation and the ships placement). After creation of an initial solution for K, the ship dispatching for the lockages is simulated and a solution for U is obtained. Then a neighbour solution K^c of K is searched, U^c is obtained accordingly and (U, K) and (U^c, K^c) are used for the simulated annealing. The neighborhoods for heuristic adjustments exclusively focus either on lockage operation activity and direction or on the service time. The used neighborhoods are “remove lock service” (removal is done randomly, weighted by the utilization costs), “add lock service” (the intervals between two consecutive lockage operations are calculated and one interval is selected randomly, weighted by length) and “convert lock service” (select one lockage randomly and change its direction). Similar biased random operations allow decreasing and increasing the service time for a lockage. The algorithm has been used in practice in the scheduling system that was put in operation in January 2006.

The model does not explicitly contain empty lockages, but requires two consecutive operations with equal directions to allow a lock conversion between them. Thus, removing or adding lock services via heuristic adjustments requires service times to be adjusted accordingly. This allows a different approach on the types of neighborhoods as e.g.

Verstichels model. The neighborhoods only affect the lockage times or directions (sub problem U), but do not directly affect the assignment of ships to the lockages (sub problem K). The authors do not define, how the different neighborhoods are searched.

The co-scheduling problem at the TGP-GD has also been studied by Wang and Ruan in [WR09]. Their approach differs slightly from the other ones, as they treat each lock-chamber individually by direction, resulting in twelve different service stations representing the five lock chambers and the planned ship-lift (in case of alternating operation mode of the five-step lock chambers of the TGP, each of them can offer up- and downstream operations as well). They use a hybrid genetic algorithm and tabu search combination to solve the scheduling problem. The chromosomes are composed of twelve lists of operation times of the corresponding service stations. For a given sequence, the ships are dispatched deterministically from a priority queue, based on their type and ETA at the lock. The fitness of the populations individuals is then calculated as the sum of several factors, including the normalized weight of the ships in the TGP locks, the normalized mean lock area utilization, the normalized sum of waiting times and additional penalty terms for constraint violations. Selection of individuals happens proportionally on the squared fitness. Crossover and Mutation are not explained in detail but refer to an unspecified tabu list and tabu search procedure, whose output is used as the result of mutation. Further, the mathematical model seems to be incomplete: For alternately operating lock chambers, it requires operations in the same direction to begin at least $2 \cdot E_i$ after the previous one (where E_i denotes the round trip time of the given lock chamber; see constraint 7). It further requires the j^{th} upstream operation to begin at least E_i seconds after the j^{th} downstream operation (constraint 8), but does not force the $(j+1)^{th}$ downstream operation to happen after the j^{th} upstream operation.

More recently, the co-scheduling problem at TGP-GD was studied by Ji et al. in [JYY19]. They separate the problem in three sub problems to first give the directions and timetables for lockage operations and then assigning ships to them. The Outer Layer Sub problem deals with deciding the number of lockage operations by heuristically increasing it in case of congestion, and decreasing it in case of redundancy for each chamber individually. The Inner Layer Sub problem deals with dispatching the waiting ships and putting them into the chambers following either a FCFS or a maximum area utilization rule. The Inter Layer Sub problem deals with coordinating the other two sub problems. It uses a quantum inspired binary gravitational search algorithm (QBGSA) optimizing the (binary represented) direction of lockages and a modified moth-flame optimization (MMFO) algorithm to optimize the (continuous) lockage times independently of each other. A comparison with optimal results is done by Ji et al. in [JZYZ21] which shows, that the algorithm has an optimality gap of 15 to 40 % on small instances of 10 to 20 ships. As with the two afore mentioned solutions, this one also specially applies to the co-scheduling problem at TGP-GD and can hardly be applied to an instance of the ILSP with more than two water gates.

The forth lock system that has been researched under the aspects of lock scheduling is the Kiel Canal or “Nord-Ostsee-Kanal” (NOK) in Germany. The NOK consists of two

locks that separate the about 100 km long canal from the sea. The canal is an alternating chain of wide "siding" sections and narrow "transit" sections, so congestion can occur not only at the locks, but also within the canal before entering a transit section. To allow efficient scheduling of ships, an algorithm has been developed, that separates the scheduling problem into three parts: the scheduling at each of the two locks and the scheduling inside the canal.

In [Luy11], Luy describes the lock scheduling problem and his solution to the problem at the two locks. His algorithm is used by a superior "canal-program", which is not explained in detail. Basically, each lock is scheduled independently. Interdependence is avoided by using overlapping time horizons, defined by the canal-program. Operations at the locks and related decisions from a previous interval are fixed for the subsequent interval. This eases the process drastically as it does not only remove the problems of interdependence, but divides the problem instance into smaller pieces that can be solved more efficiently. Luy mainly bases his methods on [VB09], especially the local search. He further proposes post-optimization steps that clean the solution from double empty lockages and trailing empty lockages. The third step considers unequal work load at parallel chambers. When - at some point - one chamber encounters a high traffic load, while another chamber is idle, the corresponding lists of assigned ships are crossed. This problem could also be countered by several applications of moves in a neighborhood (move ship to another chamber).

The aspect of scheduling ships inside the NOK is studied by Lübbecke in [Lü15] and Lübbecke et al. in [LLM19]. They define the Ship Traffic Control Problem (STCP) and provide a mathematical model and a heuristic solution algorithm, combining concepts from collision-free routing algorithms and machine scheduling. In the STCP ships have to pass several transit sections, that can (in general) only be used by ships traveling in the same direction simultaneously. Opposing ships have to wait in the broader siding sections (except for small vessels that can pass each other). While the STCP does not have all constraints of the ILSP, there are some similarities (see Section 2.2, Comparison and Summary of Existing Approaches). The proposed heuristic uses a labeling construction algorithm and local search with a simple neighborhood that is constructed by switching a ships precedence with any opposing ship it is waiting for. As the total impact of such a small local change is hard to calculate, the algorithm uses the difference in total waiting times only of the locally affected changes as an estimate. A subset of neighboring solutions with best local estimates is then evaluated (and possibly repaired) and the best of these valid candidate solutions is finally used as incumbent solution. The algorithm uses a tabu list and a limited number of worsening steps to escape cycles and local optima. It is further adopted to cope with a rolling horizon where the instance is iteratively processed in half overlapping time horizons with a length of two hours. As with the rolling horizons of the lockage operations at the NOK described by Luy in [Luy11], this does not only reduce the complexity as only a few decisions need to be made that are fixed for later iterations, but also reflects the practical application of such an algorithm as ships arrive at the canal continuously with only a short disclosure date. Computational

results of the heuristic show that the rolling horizon heuristic reduces the average waiting time by 25 % on average compared to (realized) real world instance data.

The model for the STCP is further refined by Meisel and Fagerholt in [MF19], where they add variable ship speeds (instead of default speeds), a maximum waiting time constraint in total (three or two hours, depending on the ship type) and per siding (90 or 60 minutes), and capacity constraints for the sidings. They present a matheuristic that solves a relaxed version (especially without binary decision variables) with a MIP solver (using an intermediate, sufficiently good solution after a fixed runtime) and then iteratively re-adds (in batches of size $\Lambda = 20$) only those variables and constraints that actually caused a conflict. Their computational experiments show that they can solve (generated) instances of relevant sizes within a few seconds that where not solvable exactly or even feasible within an hour, with low gaps of $\delta \leq 1.0\%$ (compared to the best known feasible exact solutions). They further experiment with different maximum waiting time constraints, showing that the canal could be passed with decreased guaranteed total travel times which could increase the canals attractiveness.

One of the scarce publications on the ILSP, especially concerning its complexity, is the thesis of Passchyn [Pas16]. He proves strong NP-completeness on two relaxed versions of the problem (identical ships, identical locks with unbounded capacity, and two identical locks, each ship has to pass both locks, all ships head for the same direction). The problem description assumes a single chamber per lock, all ships have to pass all given locks in either up- or downstream direction. The vessels travel at variable speeds within given upper and lower bounds, but speed along a section is always constant. There exists a deadline for each ship, before which it must have left the last lock. The considered objective functions are minimizing the total flow time or minimizing emissions. Passchyn provides two MIP formulations (time-indexed and lockage-based) and also a single-lock based heuristic, named RISL (repeated iterations of single lock scheduling). This algorithm schedules the ships directly approaching each single lock optimally for each lock. Then, arrival times at neighboring locks are updated accordingly and the whole process is repeated and solutions are updated again until they either converge or a maximum number of iterations is reached (to break cyclic updates). Comparing the time-indexed MIP with the RISL heuristic, Passchyn shows that on ten instances with 15 ships each, the heuristic finds the optimal solution in 40 % and an average optimality gap of 2.11 % (8.34 % maximum). The computational time is significantly reduced from 30.59 to 0.40 seconds.

Another MIP model for the ILSP (or Generalized Serial-Lock Scheduling Problem) based on a multi-commodity network (MCN) is proposed by Ji et al. [JZYZ21]. They allow a series of consecutive locks to have one or more parallel independently operating chambers of different types. Ships travel in both directions, passing one or multiple locks. They also consider the two dimensional ship placement problem with mooring constraints inside the chambers. The objective function aims on reducing the ships total waiting time (including a fairness constraint) and the total water consumption by minimizing the number of lockage operations. This seems to be the most elegant model defined so

far in the literature. The proposed general approach can also be used to solve (single) LSP instances and is shown to be on a par with Verstichels algorithms, depending on the setting. By relaxing the constraints to meet the features of Passchyn's model, they show that the time-indexed model of Passchyn, while being very restricted, performs significantly better, especially when the chambers capacity (in numbers of uniform units) increases. As stated above, the MCN model is also compared to the hybrid co-scheduling heuristic of Ji et al. for the TGP-GD setting. Instances of up to 16 ships can be solved to optimality within less than two hours.

2.2 Comparison and Summary of Existing Approaches

Various solution methods to the ILSP and related combinatorial optimization problems have been studied, as well as other publications covering related topics and aspects. This sections summarizes the research results and focuses on findings and insights valuable for our work.

2.2.1 Early Works on Interdependence

While research done on the UMR clearly addresses the problem of interdependence between locks, the main aim of Schonfeld et al. [MS95],[TS98],[TS99],[TS01] was to find a way to plan investments in infrastructure efficiently while relying on probabilistic data, and not actually to solve the scheduling problem. Still, these early works show interesting aspects like the definition of and measurements for interdependence, fairness of different processing policies, and especially the benefits of speed reduction to reduce travel costs and emissions of vessels.

2.2.2 (Single) Lock Scheduling Problem

The DP approach of Coene et al. [CS11] for the LSP cannot be used or extended to solve the ILSP as she relies on time windows where no ships arrive at a water gate and the water gate can reach any state, s.t. the problem instance P can be split into two sub problems P_1 and P_2. An optimal solution to P is the combination of optimal solutions of P_1 and P_2. Due to the interdependence of lockage operations in the ILSP and the resulting volatility of ship arrival times, such a time window cannot be found in instances of reasonable size and traffic density.

The publications of Verstichel et al. [VB09],[Ver13] are particularly interesting when looking at the ship placement sub problem. The proposed models, constraints and algorithms therefore might be a good starting point for any extension or adaption of the sub problem. Further, his evaluation and comparison results in a real life environment are valuable and show the challenges of exact methods (unpredictable runtime) and mathematically optimal solutions (comprehensibility) in such a scenario. Concerning the (main) scheduling problem, even though he only addresses a single water gate, the used neighborhoods and the late acceptance criterion seem to be promising techniques

even for the ILSP, especially because the available natural actions to change a river wide lockage schedule are all bound to decisions made at single water gates.

2.2.3 Ship Traffic Control Problem

While the locking situation at the NOK and its solutions do not provide more insight to the ILSP than the other LSP solutions, the proposed solutions to the STCP are very interesting. Comparing the STCP to the ILSP, sidings relate to open river sections where ships can pass each other in any direction, and transit segments relate to locks where (in general) ships can only travel in the same direction simultaneously while opposing ships have to wait. You could define transit segments as locks with a processing time of zero (which also allows instant empty lockage operations to reset the lock chambers to any state without costs). The surrounding areas, which cannot be entered by opposing ships would then resemble the transit segments.

Both problems share the concept of interdependence between scheduling decisions. Local changes at one point can (and in fact need to) propagate across the whole instance. When only considering the total travel or waiting time in both problems, they are both subject to the same challenge, that simply reducing waiting time at one point during the trip does not affect the total objective value at all. Instead the reduction of waiting time needs to be propagated to the end of the trip to finally improve the solution. Otherwise it is just another schedule with the same total waiting time distributed differently across the hindrances.

In the STCP the scheduling decisions are much simpler. Focusing on a single transit segment, the ships are processed like a continuous stream that can carry an unlimited number of (sequential) ships heading for the same direction. As can be seen by the neighborhood definition for the local search used by Lübbecke et al. [Lü15][LLM19], the only type of move is shifting precedence of ships, which correlates in a change of stream direction (inducing a tear down time to wait until the transit segment is free). On the other side, the ILSP works more like a buffered batch processing where a limited number of ships is processed at once. The ILSP has therefore much more scheduling decisions to make, not only which ships are combined to a batch and processed together, but also their relative order inside the batch. Most of all, in the ILSP the batch sizes are constrained and operation directions need to alternate, so a continuous stream of vessels in one direction needs to be interrupted anyway, no matter if there are opposing ships or not.

Another phenomenon that can be seen in both problems is, that sequential routing (i.e., iteratively routing ships, one after the other), which is used as a construction heuristic by Lübbecke et al., is unable to construct an optimal solution when this optimal schedule requires cyclic waiting. This is proven by Lübbecke et al. in [LLM19] by (in short) showing that in sequential routing, the first ship inserted into the schedule never waits for another ship. But if an optimal solution requires every ship to wait for at least one other ship, the constructed solution cannot be optimal, regardless of the ordering of ships.

2.2.4 Hybrid Approaches

Hybrid approaches that combine exact mathematical models with approximating heuristics are becoming increasingly popular when (possibly NP-) hard problems need to be solved in a feasible amount of time. This is especially relevant when these problems need to be solved in a restricted amount of time in real life environments. Examples are periodical scheduling tasks like weekly nursing plans, daily parcel delivery routes or online vehicle routing problems.

One way of hybridizing is a matheuristic as proposed for the STCP by Meisel and Fagerholt [MF19]. Their computational experiments show that such a combination of relaxed mathematical models that are heuristically tightened to get a feasible solution is not only very fast (as a heuristic), but also induces only a small optimality gap. The ILSP could be an interesting application of a matheuristic in the future.

Hybridization can also be achieved by using fast heuristic methods for cut generation as shown by Verstichel [Ver13] in his proposal for the ship placement sub problem, which can be seen as an extension to the basic problem. Like a matheuristic this Combinatorial Benders' decomposition highly improves the runtime of the algorithm without sacrificing the solution quality.

Another hybrid concept is the usage of exact solution techniques on small sub problems as part of a heuristic algorithm. Passchyn uses this combination as an integral component and not as an extension of his solution to the basic problem in [Pas16] by repeatedly finding optimal solutions for short term and local scheduling decisions. Especially the latter approach shows in computational experiments, that even small instances of 15 ships cannot always be solved to optimality by only finding locally optimal decisions at single water gates. This very well shows the reason why we strongly emphasize the term *interdependence* in this thesis. Still, Passchyn's approach could be used as a starting point for further improvements through metaheuristics or as an initial solution for further exact approaches.

2.2.5 Mathematical Models

Recent publications of Passchyn [Pas16] and Ji et al. [JZYZ21] propose exact models for the ILSP. Especially the latter one seems to be rather feature complete for many real life instances of the problem at hand, including two dimensional bin packing and mooring constraints inside lock chambers. Their inability to solve instances with 20 ships traveling along six locks optimally within two hours shows the necessity to develop heuristic solution techniques for the ILSP for real life applications.

CHAPTER 3

Methodology

This chapter describes the scientific aspects of how the project *imFluss* was conducted, especially the part relevant for this thesis.

3.1 Research on the Topic

The project *imFluss* was launched in 2014, which is when the initial research started. Searching relevant online catalogs of scientific publishers and other online databases of scientific articles for publications about scheduling of ships on inland waterways across multiple locks showed, that at that time the ILSP was not described in the literature. At most the ILSP was noted as possible future research in publications dealing with the (single) LSP (e.g. in [Ver13]). Some publications could be found, concerning the co-scheduling at the TGP-GD. While they are technically addressing the problem of scheduling ships passing multiple (i.e., two) locks, the specialized problem description and solution techniques can hardly be applied to the generalized problem.

As the initial research showed many solutions to the single LSP, but did not show any valuable results concerning the aspect of "interdependence" between multiple locks, research focused on that term in particular. Some interesting publications on congestion analysis and simulation could be found, showing that the problem of efficient scheduling along a series of locks is already known since the 1960's [SL68]. Interestingly no scheduling algorithms or solution methods to the ILSP have been provided until 2014.

Due to the long time between the initial research and the publication of this thesis, a second research was conducted in late 2020, to also incorporate the results of recent publications. It can be seen, that since 2014 a handful of relevant papers have been published, showing the increased interest in the field of optimizing transportation along inland waterways (especially [Pas16] and [JZYZ21]).

3.2 Multi-Criteria Optimization

The solution quality in the context of the ILSP depends on two criteria: the total waiting time of the vessels and the number of lockage operations performed. These criteria can be in conflict with each other to some extent. When improving a solution to the ILSP at some point one criterion cannot be improved further without worsening the other one. Whenever two ships are served by the same lockage operation together, both ships need to spend additional time waiting for the other ship (one during entering, the other during leaving the chamber), because no two ships are allowed to enter or leave the chamber simultaneously. From this, it follows that when reducing the number of lockage operations by serving multiple vessels at once, the waiting time increases.

A solution where no criterion can be improved without deteriorating another one is called a Pareto Optimum. One of the difficulties in multi criteria optimization is, to decide on which pareto optimal solution to choose. By multiplying each criterion's objective with a non-negative value, called the weight or cost factor, the project *imFluss* uses *linear scalarization* to combine the two criteria to a single scalar value, that can be minimized as a single objective.

This method is especially applicable, as the two criteria can be evaluated quantitatively and set in relation to each other. The cost factors could also be part of the input to represent instance specific circumstances. For example, during dry seasons when the water level is low, lockage operations could be more expensive yielding to increased waiting time in order to save water and thus increase navigability of upstream river sections.

3.3 Data Modeling

The project *imFluss* aimed on providing a definition, model and solution methods of the ILSP under the given circumstances at the Austrian part of the Danube (described in Subsection 1.2.3). While one part of the team focused on the mathematical model and exact solution methods, the other part which is covered by this thesis focused on heuristic approximations. A lot of resources like instance data, program subroutines or data structures for instance data processing, solution representation and output generation where shared between the two divisions to maintain a level of compatibility, interoperability and finally comparability.

The processes of lockage operations at water gates and their interdependency at the same and neighboring water gates was analyzed. From this, a formal problem description was defined that specifies constraints to define a feasible lockage schedule.

Given this model, an efficient solution representation was defined and implemented to support not only several heuristic methods, but also any exact approaches. By using the same data structures by different strategies, comparability and also interoperability could be achieved more easily. The generated solutions contain data about the arrival,

lockage and departure times of each ship at each passed water gate, or – from the water gates point of view – which ships are to be served when.

The performance of the heuristic algorithms highly depends on the performance of the backing data structures. A physical model for solution representation was developed, that allows efficient operations for evaluating and applying changes to the current incumbent solution and increase the efficiency of heuristic optimization techniques.

3.4 Development of Algorithms

The ILSP is proven to be at least NP-complete by Passchyn in [Pas16]. Exact solution methods are therefore likely to require an unsustainable amount of time to calculate solutions in a real world application. Heuristic optimization algorithms do not necessarily find optimal solutions, but can find sufficiently good solutions in an acceptable amount of time. While the scope of the project *imFluss* includes the development of exact mathematical models of the problem, the part that is covered by this thesis aims on developing heuristic methods.

Basic construction heuristics were developed to test the model and data structures. They were improved and finally create initial solutions for the developed metaheuristics. We developed several construction heuristics to assess the impact of the following characteristics.

- mutable and immutable construction heuristics (see Subsection 4.1.3)
- sequential routing of whole trips
- chronological scheduling according to ship arrival events
- deterministic and random heuristics

Good initial solutions can speed up later optimization steps, but might also narrow the search space and push later improvements towards local, non-global optima. Therefore several construction heuristics were used in combination with the metaheuristics to analyze the possible impact of the construction heuristics solution quality to the final result.

Secondly, several metaheuristic solution approaches were developed, adjusted and finally compared to each other. Various metaheuristics (e.g., GRASP, Variable Neighborhood Search, Variable Neighborhood Descent) are commonly used in the field of transport optimization. We examined some of them and finally created a versatile framework to create the Neighbourhood Search with Configurable Neighbourhood Iteration (NSCNI) (see Subsection 4.4.4).

Finally the developed heuristic strategies are compared to historical data and (if existing) optimal values achieved by exact approaches.

The development of any algorithm basically happens in a loop of implementation, evaluation and analysis. If applicable, algorithms are designed to use parameters that influence the result and need to be tuned. To find values for these parameters that are likely to yield good solutions, several settings are – possibly iteratively – evaluated.

3.5 Experimental Evaluation

The algorithms performance and solution quality is analyzed using quantitative methods. Two sets of benchmark instances are defined and solved by different algorithms or parameter values. These combinations are called *settings*.

All experiments use the same cost factors applied to the objective function as in [PRSR15] to maintain comparability not only with each other, but also to previous publications. Finding reasonable values for the cost factors would require research on the economical aspects of the topic, that are out of scope of this thesis, and would need to respect many aspects to the problem as described in Section 1.2.2.

The benchmark instances are created from historical data, that contain relevant data about ships, their gate passage times and positioning data during the course of several days. The instances can contain multiple trips per ship. While in real life, these trips would depend on each other and a later trip cannot start before all earlier trips of the same ship are completed, we handle them as independent trips in our instance data as if they were conducted by individual ships.

Comparing the settings result against the original historical data, using the actual travel times as a baseline, is not an option, as these do not contain information about the performed lockage operations. Even for solely comparing the travel time they are not well suited as the data is partially incomplete or inconsistent. They further don't contain data about the possible maximum speed of vessels, which makes it impossible to find improvements compared to the historical travel time.

As optimal solutions (calculated by a MIP solver using the mathematical model) cannot be found for all instances with reasonable effort, they are also not a good choice for a baseline by calculating the heuristics optimality gap. Therefore we need to use other techniques to evaluate the quality and performance of our proposed algorithms.

First, different settings are compared against each other using statistical hypothesis testing. While this does not quantify the absolute quality of a setting, it allows to at least find the best or most promising settings among the proposed candidates in relation to each other. The hypothesis testing is well suited for comparing two individual settings with each other, but does not directly allow a comparison of multiple settings. To create a ranking of n settings, we compare each setting with every other setting on the same set of instances, resulting in $n \cdot (n-1)/2$ pairwise comparisons. Whenever a setting performs significantly better than another one, it scores one *win* point. Whenever a setting performs not significantly different than another one, it scores one *tie* point. We rank the settings first by *wins* and then by *ties*. The setting with the highest total *wins*

score outperformed the most other settings and can be considered the best setting. If two or more settings have the same *wins* score, the setting with the most *ties* is itself outperformed by less remaining settings and can be considered better than the other settings with same *wins* but less *ties*. The resulting table of scores gives a good overview and can be used to rank the compared settings.

A second way of comparing optimization algorithm metrics are *performance profiles* as defined by Dolan and Moré in [DM02]. The performance profile for a solver is the (cumulative) distribution function for a performance metric. It shows the probability (i.e., the percentage of instances) for a solver or algorithm to find a solution with a relative threshold τ of the baseline. They used it to compare different solvers with each other by their computational time on a defined set of benchmark problems. But the proposed technique can be applied on any metric and can therefore also be used to assess the solution quality of heuristic optimization algorithms. Dolan and Moré show that performance profiles eliminate many problems of other comparison techniques like the influence of a small number of problem instances on the whole process, information loss by statistical aggregation or the need to choose arbitrary parameters or threshold values.

Performance profiles directly show the likeliness of an algorithm being able to solve an arbitrary instance within a relative range of the best known algorithm. This includes the probability of providing the best value of the metric (i.e., the optimal solution for objective values or the fastest result for computational time), as well as worse case behaviour. They further allow to show multiple profiles in a single graph to analyse multiple algorithms together. Usually, performance profiles compare against each other, using the total minimum value of a metric as the baseline. To let all performance profiles compare against the same baseline, we add a "best known" performance profile to all individual comparisons that holds the best known objective values per instance ever calculated by any setting, including proven optimal solution from the MIP model.

The following example shows, what information can be drawn from a performance profile plot. Figure 3.1 shows the comparison between two settings A (orange) and B (green) against the baseline *best known* (blue). From this visualization, we can read some information, that are hard to summarize in a textual or tabular form.

- Looking at the bottom left, we can see, that setting B is able to provide the best known result on 10 % of all instances, whereas setting A's best results are 20 % above the best known (best case performance).
- Looking at the top right, we see the settings worse case performance. Setting A is able to solve all instances with at most 130 % above the best known value, whereas setting B can yield results up to three times the best known value.
- The turning point is at around 50 % of all instances or at $\tau = 1.7$.
- When picking a random instance, there is a 50 % chance that setting A clearly outperformes setting B.

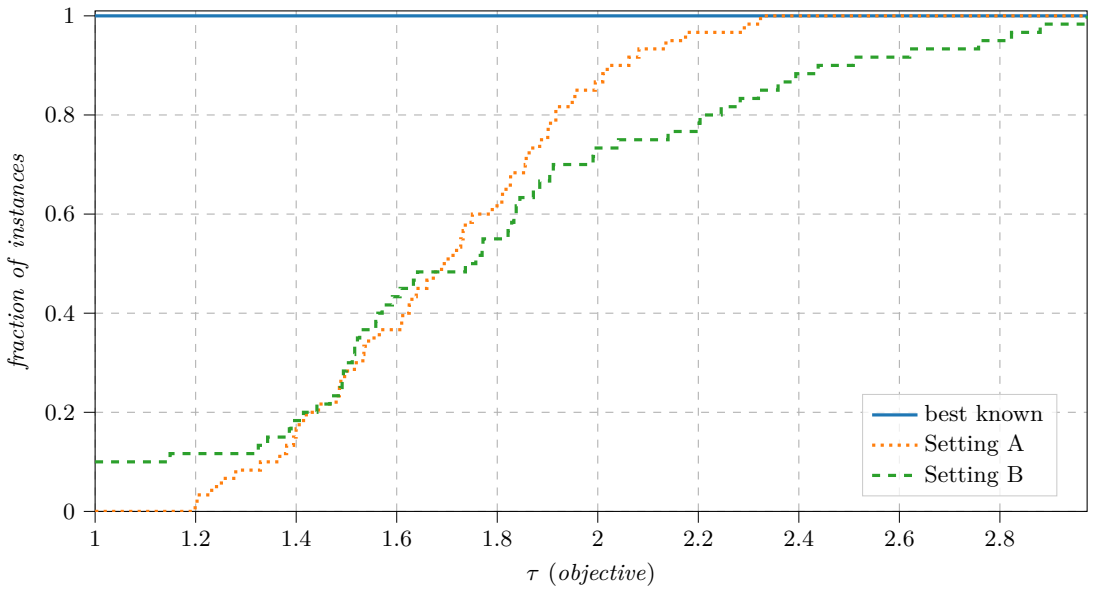


Figure 3.1: Example performance profiles of two hypothetical algorithms.

- When requiring a value of at most 200 % of the best known value (a gap of 100 %), setting A can satisfy this in approximately 85 % of all instances.

This thesis uses a Python tool to create performance profiles from collected data. The tool is based on the *perfprof.py* project by Daniel Steck. The original code is available under MIT license at github.com[1].

We further want to mention another visual online tool to create performance profiles by Van Cauwelaert, Lombardi and Schaus. The tool can be found online[2] and was presented and discussed in [VCLS17]. The source code is available via bitbucket.org[3].

[1] `https://github.com/dmsteck/perfprof.py`
[2] Performance Profile web tool: `https://performance-profile.info.ucl.ac.be/`
[3] `https://bitbucket.org/svancauwelaert/performance-profile-web-tool/`

CHAPTER 4

Algorithms

This chapter describes the various algorithms and the necessary backing data structures that were developed to solve the ILSP. This includes construction heuristics, local search neighborhoods and metaheuristics.

4.1 Preliminaries and Definitions

This section defines some basic terms and concepts, that will be used in this chapter. While some of them are commonly applied in the field of heuristic optimization, scheduling, transport and logistics, others are specific to the ILSP. For more detailed information on local search, see Michiels et al. [MAK07].

4.1.1 Move

In general all algorithms discussed in this thesis manipulate their underlying data structures by applying a *move* to the schedule. It defines a set of actions (the move instructions) needed to accomplish a task and results in a change of the current solution and (likely) of its objective value when it is applied to a schedule. This impact on the objective is called the *delta* or the *cost* of the move.

In the ILSP, moves can add or remove lockages in a water gate's schedule, add or remove ships from or rearrange ships within a lockage operation, or be a composition of the afore mentioned. Examples of moves are:

- *Add a ship to an existing lockage*,
- *Swap two ships positions within the same lockage*, or even
- *Completely rebuild a single water gates schedule.*

4.1.2 Neighborhood structure

A neighborhood structure (or neighborhood relation) is a function $\mathcal{N} : S \to 2^S$ that assigns each solution $x \in S$ a set of neighboring solutions $\mathcal{N}(x) \subseteq S$. A solution x' is considered a neighboring solution of x, if it can be created from x by applying a move (see above) on x.

For example, the above mentioned type of move that swaps two ships within the same lockage would create a neighborhood of solutions $\mathcal{N}_{swapInLockage}(x)$ that consists of all possible solutions that can be created by swapping any two ships that are scheduled for the same lockage operation in the given solution x.

Local search techniques use neighborhood structures to find optimal solutions. A *local optimum* or local optimal solution is a solution x with $f(x) \leq f(x')\forall x' \in \mathcal{N}(x)$, i.e., it is not possible to further improve the given solution by applying (single) moves of the given neighborhood. A *global optimum* or global optimal solution is the best solution that can be found within any possible neighborhood.

The challenge of local search techniques is twofold. We need to to define neighborhood structures that cover as much as needed of the solution space, to find global optima. On the other hand we want small neighborhoods that can be efficiently searched to reduce computational time.

Searching the neighborhood and selecting the next candidate solution is done by neighborhood explorers. These algorithms try to reduce the search space of a neighborhood structure by exploiting mathematical properties (e.g., symmetries) and eventually choose the next solution by adhering a step function. Typical step functions are

- best improvement (BI): Completely search through $\mathcal{N}(x)$ and choose the best neighbor.
- next improvement (NI): Systematically (deterministically) search through $\mathcal{N}(x)$ and choose the first neighbor, that is better than the current solution.
- random improvement (RI): Randomly search through $\mathcal{N}(x)$ and choose the first neighbor, that is better than the current solution.
- random neighbor (RN): Pick a random neighbor from $\mathcal{N}(x)$, no matter if its an improvement or not.

4.1.3 Mutable and Immutable construction heuristics

Construction heuristics start with an empty solution and iteratively extend it until a complete solution is found. In our scheduling problem this means that lockage operations and ships are added to an initially empty schedule one after the other, until all ships have been added to a lockage at each water gate they have to pass. In general, a construction

heuristic does not necessarily build a feasible or valid solution. The only requirement is, that the solution is complete.

When adding new elements to the (partial) solution, we differentiate algorithms by their capabilities:

- *Immutable construction heuristics* add the new element to the schedule without altering previous assignments and derived timing information of the existing partial solution. They only take into account local implications of their decisions and neglect implied additional costs of dependent operations.

  As stated in Subsection 1.2.2, by adding a ship to an existing lockage operation, the scheduled times concerning this operation and of any dependent operation need to be adapted. Immutable heuristics do not propagate those changes to dependent lockage operations, so they only allow an approximation of the objective value during decision making and base their future decisions on incorrect timing information.

  This inaccuracy is condoned as it makes the algorithm much simpler and reduces the computational time, as it only searches for gaps in the schedule that suffice for the new element to be inserted or added.

- A *mutable construction heuristic* can apply changes to the existing partial solution. As their immutable counterparts, they search for gaps or possibilities to add the new element to the existing schedule. But instead of neglecting implied global updates to the existing assignments, mutable heuristics propagate changes and allow a precise recalculation of the whole schedule. This increases the computational complexity of the algorithm but allows exact calculation of costs for the current operation and future decisions to be based on precise information.

4.2 Construction Heuristics

To get a basic understanding of the problem and to find initial solutions for local search based metaheuristics, several construction heuristics where developed. As stated in Section 3.4, Development of Algorithms, they consider different characteristics or features to analyse their impact on the ILSP.

4.2.1 Immutable Sequential Deterministic Construction (ISD)

A basic generic construction heuristic has been developed, which follows a simple process. It performs sequential scheduling, so one ship after the other is scheduled at the water gates it has to pass along its trip. The algorithm works on an immutable solution representation, so as described in Subsection 4.1.3, it can only estimate the impact of adding a ship to an existing lockage operations. Its behaviour is deterministic, depending on the ordering of the input list of trips. From these characteristics we call this algorithm the Immutable Sequential Deterministic Construction (ISD).

Algorithm 4.1: Immutable Sequential Deterministic Construction

Input: A (generically, arbitrarily) ordered list of trips $\mathcal{S}$
Data: p_s current position of ship s,
t current time
Output: A three dimensional schedule X

```
1  begin
2      X ← {}
3      foreach s ∈ S do
4          p_s ← λ_s
5          t ← r_s
6          foreach g ∈ G_s do
7              at^a_{s_g} ← calculateShipArrivalTime(g, s, p_s, t)
8              l^x ← earliestExistingLockageSupportingShip(g, s, at^a_{s_g})
9              m^x ← moveAddingShipToLockage(s, l^x)
10             c^x ← costApplyingMove(X, m^x)
11             l^n ← earliestNewLockage(g, s, at^a_{s_g})
12             m^n ← moveAddingNewLockageWithShip(s, l^n)
13             c^n ← costApplyingMove(X, m^n)
14             if c^n ≤ c^x then
15                 X ← apply(X, m^n)
16                 l ← l^n
17             else
18                 X ← apply(X, m^x)
19                 l ← l^x
20             end
21             p_s ← exitPosition(g)
22             t ← exitTime(s, l)
23         end
24     end
25     recalculateTimes(X)
26 end
```

When looking for existing supporting lockages at a water gate, all lockages after the ships arrival are investigated. A lockage supports a ship if its direction matches the ships direction and its remaining capacity suffices. As a general rule, this algorithm never delays existing lockages during insertion of new lockages or when adding a ship to a lockage operation. The only exception to this rule is an empty lockage operation that is scheduled right before the ships arrival at the gate. This lockage operation can be delayed without affecting the other ships, as long as the order of the lockage operations stays unchanged. Such an empty lockage operation is then also taken into account, when looking for a supporting lockage. If it supports the new ship and its delayed lockage time does not interfere with existing lockage operations, it is eventually delayed and the ship is added to it.

When adding the new ship to an existing lockage operation, the new ship tries to enter the chamber as early as possible. If entering the chamber could be finished before the lockage operation starts, the ship is assigned to the last position of the supporting lockage operation. Any possible collisions with already assigned ships during entering the chamber as well as a possible delay of the lockage operation itself are ignored. The algorithm stops searching for further possibilities as soon as the first (earliest) supporting lockage operation was found.

Beside looking for existing supporting operations, all gaps between operations after the ships arrival are examined if an additional lockage operations can be added to transfer the ship. When adding new lockages, the chain of operations needs to stay feasible, i.e., the water gate must not enter an invalid state, e.g., performing a downward operation when all chambers are already empty. Therefore, if additional empty lockages are necessary to keep the chain of lockages feasible, they are inserted as early as possible in the chain of existing lockages. If two empty lockages in inverse directions are detected, and the chain of lockages between them would be feasible without them, they are both removed. All these necessary actions (adding or removing additional empty lockages) to keep the schedule feasible are combined in the single $moveAddingNewLockageWithShip$. The algorithm stops searching for further possibilities as soon as the first (earliest) possibility for adding a new lockage operation was found.

For each ship at each gate, the algorithm creates at most one move that adds the ship to an existing lockage and exactly one move to add it to a new lockage. The cheapest move is applied greedily to the schedule to add the ships passing of the current water gate.

Cost evaluation is done incrementally and is approximated by only considering

- the change in the total number of lockages,
- the waiting time of the added trip before the lockage operation (before and after entering the chamber), and
- the total travel time of the added trip (i.e., the time required to pass the preceding river segment).

The waiting time when leaving the lockage is neglected, as well as an increase in waiting time of other ships due to a delayed operation of the lockage. Further, the delays of departing ships are not propagated to successive water gates. The ships' arrival times there remain unchanged.

As stated above, the heuristic does not check on collisions during entering a chamber and also does not update lockage operation times, so the generated solution is likely infeasible. To get a feasible solution, only the relative scheduling information (i.e., for each water gate which ship is served by which lockage operation and what is the ship's relative position inside the chamber) is kept and all timing information is cleared and recalculated from scratch.

The ISD is highly dependent on the ordering of the ships, because ships that are scheduled early in this process are less likely to encounter waiting times. Ships that are added to the schedule late in the process need to fit into the possibly densely populated schedule and are therefore more likely to encounter long waiting times. This is symptomatic for simple heuristics that perform sequential scheduling.

While any arbitrary ordering can be applied to the ISD, we implemented three different base orderings, each can be applied in ascending and descending order:

- by trip starting time
- by ship length
- by trip length (distance traveled)

The impact of the different orderings is shown in Subsection 5.3.1.

Given two characteristics of a problem instance, the number of ships $s = |\mathcal{S}|$ and the number of water gates $g = |\mathcal{G}|$, we calculate the asymptotic runtime as follows. For each gate passage of every ship ($O(s \cdot g)$) the current lockages at the given gate ($O(s)$) are iterated twice to find suitable moves, so $ISD(s, g) = (s \cdot g) \cdot s = O(s^2 \cdot g)$.

4.2.2 Immutable Sequential Randomized Construction (ISR)

Based on the deterministic construction heuristic, a randomized approach was developed. It utilizes the above described dependency of the algorithm on the initial ordering of the trip list and adds a parametrized degree of randomness to it. The result is a randomized construction heuristic that is based on the idea of semi-greedy heuristics as proposed by Hart and Shogan in [HS87].

The Immutable Sequential Randomized Construction (ISR) wraps around the ISD and provides the initial trip list to it. Therefore, the (deterministically) sorted list of ships $\mathcal{S}$ is used to build a restricted candidate list (RCL). The ships s in $\mathcal{S}$ are sorted by some considered property value $p(s), p : \mathcal{S} \to \mathbb{R}$. One trip is chosen randomly from the RCL and added to the new list of trips $\mathcal{S}'$, that is then used by the ISD construction heuristic

described above. This randomization does not make the heuristic less greedy in decision making, so it is itself not a semi-greedy heuristic as defined by Hart and Shogan.

Algorithm 4.2: Immutable Sequential Randomized Construction

Input: A list of ships $\mathcal{S}^I$, sorted by a property p of ships $p : \mathcal{S} \to \mathbb{R}$,
α or l defining the *RCL*
Data: restricted candidate list *RCL*,
$\mathcal{S}'$ random generated list of ships,
f number of iterations without improvement,
X current best solution
Output: A three dimensional schedule X

```
1  begin
2      f ← 0; X ← {}
3      while f ≤ |S^I| do
4          S ← S^I
5          while S ≠ {} do
6              if l > 0 then
7                  RCL ← head(S, l)
8              else
9                  RCL ← {s ∈ S : p(s) ∈ [p(s_1), p(s_1) + (p(s_|S|) − p(s_1)) · α]}
10             x ← random(|RCL|)
11             S' ← S' ∪ s_x
12             S ← S \ s_x
13         end
14         X' ← ISD(S')
15         if objective(X') < objective(X) then
16             X ← X'
17             f ← 0
18         else
19             f ← f + 1
20     end
21 end
```

The RCL is constructed in one of two ways:

1. *Cardinality-based RCL with constant size l*: The RCL is built on the first l ships of $\mathcal{S}$ (denoted by the function $head(\mathcal{S}, l)$).

2. *Percentage-based RCL with relative threshold α*: The first l' ships of $\mathcal{S}$ are taken, such that the considered property value $p(s)$ of each ship s falls in the interval of $[p(s_1), p(s_1) + (p(s_m) - p(s_1)) \cdot \alpha]$ (where s_1 is the first ship, and s_m is the last ship in $\mathcal{S}$, sorted by $p(s)$).

In either way, one ship s is chosen randomly from the RCL. s is removed from $\mathcal{S}$ and added to $\mathcal{S}'$. Once $\mathcal{S}'$ contains all ships, it is used as an input to the ISD construction heuristic.

This process is repeated, as long as a better solution can be found within n iterations, where n is the number of ships in the instance. Thus bigger instances get a higher chance of finding a better solution by simply rearranging the initial list of ships. As Figure 5.2 shows, this repeated construction requires a multiple of computing time, but results in significantly better solutions.

Obviously, with an RCL constructed on $l = 1$ the algorithm acts totally deterministic like the underlying ISD. If the RCL is constructed with $\alpha = 1.0$, the list of ships is constructed totally randomly, no matter by which property $p(s)$ the basic list $\mathcal{S}$ was sorted beforehand. (The same is true for small instances, if the size l of the RCL is constant and greater or equal to the number of ships.)

Again we calculate the asymptotic runtime from two characteristics of a problem instance, the number of ships $s = |\mathcal{S}|$ and the number of water gates $g = |\mathcal{G}|$. As the ISR calls a nested ISD multiple times, its runtime is $n \cdot O(ISD(s, g))$ where n is the number of iterations. In our implementation $n > s$, but is otherwise hard to estimate or restrict a priori. Our experiments finally showed, that n has an upper bound of approximately $n < 5 \cdot s$, so we define the asymptotic runtime of $ISR(s, g) = n \cdot O(ISD(s, g)) = O(s^3 \cdot g)$.

4.2.3 Efficient Iteration of Dependent Lockage Operations

First experiments with immutable construction heuristics showed their lack of precision due to the fact that they do not propagate changes to dependent lockage operations during construction. This inaccuracy leads to worse solution quality, as decisions are based on wrong data. Instead of expensively recalculating the whole schedule for any evaluated decision, we countered this issue by developing a solution representation that allows efficient propagation of updates and therefore precise calculation of costs when adding new ships to the partial solution.

This data model uses a directed acyclic dependency graph of nodes, which represent the lockage operations. Dependencies are stored as directed adjacency lists per node. Implicit dependencies are not stored and need to be checked by traversal. Dependencies are stored in both direction, i.e., each node knows and discerns its parent nodes and its child nodes.

A node (the child) is dependent on another node (the parent) if it is either the subsequent lockage operation at the same water gate (i.e., l_{i_g} is a parent of l_{i+1_g}) or a lockage operation at a neighboring water gate that shares at least one ship with the parent operation (i.e., $S_{i_g} \cap S_{j_{g+/-1}} \neq \{\}$). See Figure 4.1 for an example.

Dependencies can be marked as resolved or unresolved. Each node keeps a counter of unresolved parent dependencies to determine whether it is not dependent on unresolved parent nodes, i.e., whether it is *resolvable* itself.

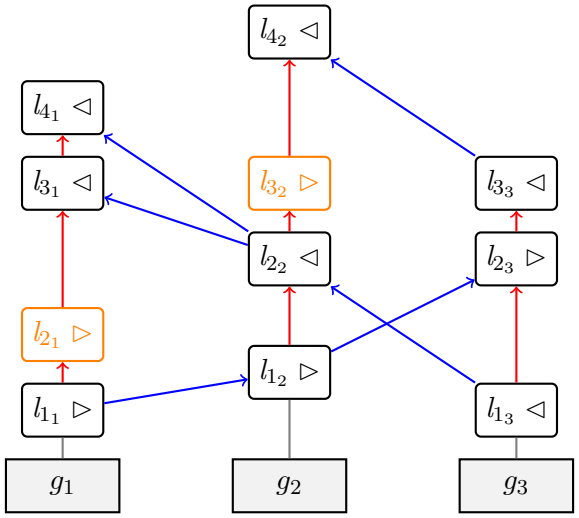


Figure 4.1: Dependency graph with three water gates (g_1, g_2, g_3). Rounded rectangles represent lockage operations (with direction indicators $\triangleleft, \triangleright$). Orange lockage operations (l_{2_1}, l_{3_2}) are empty. Red arrows indicate dependencies between subsequent operations at the same gate. Blue dependencies show dependencies between water gates because of travelling ships.

The data model further keeps a queue of resolvable items. Polling from that queue allows to traverse the graph node by node, s.t. only resolvable items are retrieved. When a node is resolved, the dependencies to its children can be resolved in two ways:

- *Change propagation*: If a change occurred at the parent node, all children need to react to that change and need to be resolved as well. Therefore the children are marked as *dirty* to indicate that they need an update. Further, if any child node becomes resolvable, it is added to the queue of resolvable items.
- *Stability propagation*: If no change occurred at the parent node, it is not necessary to update the children (from that parent's point of view). The children's update indicating dirty flag is not changed. If a child node becomes resolvable, it is added to the queue of resolvable items, if it requires an update. If it does not require an update, the child will not be added to the queue and will not be resolved later. Instead it propagates its stability to its successors. The queue of resolvable items does therefore not hold all resolvable items, but only those, that also need an update, i.e., are dirty.

As it is possible to have multiple directed paths between two arbitrary nodes, we keep a memory of already visited nodes during traversal to prevent revisiting them. As stated above, the graph is acyclic. This is not only necessary for easy traversal, but also for the problem to be feasible. A cycle in the dependency graph would resemble a deadlock in the

schedule where several ships would each wait for the other to complete their respective lockage operations. We don't use cycle detection in the graph but rather avoid adding cycles when building or altering the schedule (which is anyway necessary to keep the solutions feasible).

Whenever a change of timing information occurs at a lockage operation, the lockage operation time and ship departures are recalculated and all outgoing arcs are cascadingly marked as unresolved (i.e., the whole sub graph rooted at the initial node is unresolved). All direct child nodes are marked as dirty as the change is propagated (see *change propagation* above). The originating node's outgoing dependencies are marked as resolved which adds its direct child nodes to the queue of resolvable nodes. The first node is taken from the queue and resolution continues.

If during resolution at any successive node the change induced by the parent nodes can be compensated without requiring alteration of the corresponding lockage time and ship departures, change propagation stops for this node and instead stability is propagated to all direct children. This is the case for example if due to an update a ship arrives earlier at the gate, but the lockage operation cannot be advanced due to other circumstances, so the lockage operation cannot start earlier and therefore the ships departure does not change either. Stability would also be propagated if a ships arrival that already has some waiting time at a water gate is delayed, which reduces its waiting time without removing it totally. See Figure 4.2 for an illustration.

When a move to manipulate the schedule is evaluated, the move's costs are stored. If the move is later applied to the schedule, the incumbent solution's objective value is not recalculated from scratch but updated incrementally. This further improves the performance of the algorithms.

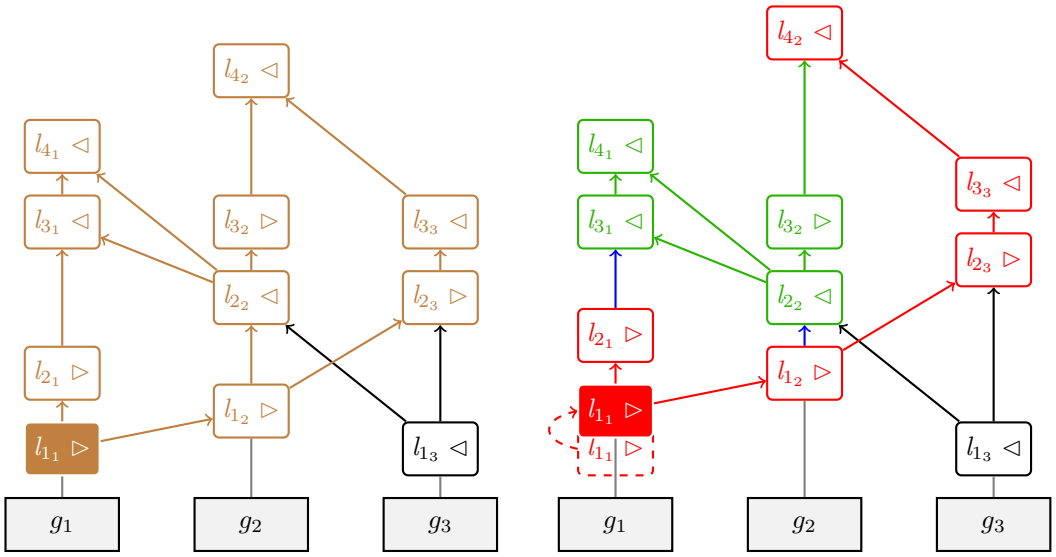
The proposed data structure allows efficient change propagation in the solution representation. This feature is necessary for algorithms to efficiently calculate exact costs of applied changes to a schedule. Those precise calculations are needed to enable the algorithms to base their decisions on correct information.

4.2.4 Mutable Sequential Deterministic Construction (MSD)

The Mutable Sequential Deterministic Construction (MSD) is based on the ISD heuristic, but uses the above mentioned graph based solution representation. It is a mutable construction heuristic.

Like the ISD it performs sequential scheduling. Instead of evaluating at most two possible moves it evaluates several move candidates. Those candidates are created by adding the new ship to all supporting lockage operations after the ships arrival. As the ISD, this heuristic also does not delay existing non-empty lockages until the new ship has arrived. This is a simple method to prevent cyclic dependencies in the schedule.

Further, between any two lockage operations after the ship's arrival, move candidates for inserting a new lockage serving the ship are created. As adding an additional lockage



A: Initial state; brown dependencies and operations are children of l_{1_1}. Applying a change to it marks them as unresolved.

B: State after resolving changes to l_{1_1} showing propagation of change (red) or stability (green).

Figure 4.2: An example dependency graph showing resolution of dependencies when applying a change. In this example, l_{1_1} is delayed which introduces a delay at dependent (red) nodes. After lockages l_{2_1} and l_{1_2} the delay can be compensated (blue), s.t. no further delay is applied to subsequent operations (green).

might break the feasibility of the chain of lockage operations (e.g., scheduling a downward operation when all chambers are already empty), all subsequent operations are tested for such an infeasible state after the insertion of the new lockage. If such a state is found, it is repaired by adding an additional empty lockage just before the violation. This is not necessarily the best position for the repairing empty lockage.

The algorithm chooses the cheapest move of all possible ones for each water gate (greedy best fit). This is not necessarily the best insertion for the trip as a whole.

The MSD has by design two major advantages over the ISD. It evaluates more candidates when inserting a new ship into the schedule and it operates on exact timing information, allowing precise decision making. The latter also makes the final recalculation of times obsolete.

We calculate the asymptotic runtime complexity similar to the ISD. For each gate passage of every ship ($O(s \cdot g)$) the current lockages at the given gate ($O(s)$) are iterated twice to find suitable moves. Additionally for every found move, the MSD recalculates the schedule by traversing dependent lockages ($O(s \cdot g)$), so $MSD(s, g) = (s \cdot g) \cdot s \cdot (s \cdot g) = O(s^3 \cdot g^2)$.

4.2.5 Mutable Sequential Randomized Construction (MSR)

The Mutable Sequential Randomized Construction (MSR) applies the randomization techniques of the ISR heuristic to the MSD heuristic explained above.

This leads to further improvement in solution quality at cost of a much higher computational time.

As the MSR calls a nested MSD multiple times, its runtime is $n \cdot O(MSD(s, g))$ where n is the number of iterations. For the MSR our experiments finally showed, that n has an upper bound of approximately $n < 6 \cdot s$, so we define the asymptotic runtime of $MSR(s, g) = n \cdot O(MSD(s, g)) = O(s^4 \cdot g^2)$.

4.2.6 Mutable Chronological Deterministic Construction (MCD)

The Mutable Chronological Deterministic Construction (MCD), in contrast to the previous heuristics, does not use sequential scheduling. Instead it keeps a sorted list E of ship arrival events at water gates. For every ship that has not yet been fully scheduled, E contains the next arrival event. Events are taken from the top of the sorted list and the ship is assigned to a lockage at the corresponding water gate. The algorithm greedily chooses the best candidate for adding the ship to the schedule (i.e., method $chooseMove(M)$ in algorithm 4.3, line 21 simply chooses the best solution). As this is a mutable construction heuristic that uses the above mentioned dependency graph, any effects on other already scheduled operations are taken into account and the schedule is updated. After the event has been processed, all ships that where affected by the alteration of the schedule need to update their arrival events in E. Then the current ships next arrival event is calculated and added to E. The algorithm is done, when the list of events is empty, which means that all ships have reached their target position.

In its basic principle, the algorithm resembles a naive FCFS policy. By specifying the lookahead time d^+ and d^*, it allows foresighted scheduling in cases where it would be beneficial to await another ship before processing the lockage operation. Without the lookahead time, adding a ship to an existing lockage would only be possible in case of a congestion (i.e., the ship arrives at the water gate while at least one operation has not yet begun).

The MCD can be parametrized by

- disallowing delaying of existing lockage operations (called *simple* setting, $d^+ = d^* = 0$),
- allowing delaying of existing lockages when adding a ship (called *delayAdding* setting, $d^+ = d$ and $d^* = 0$), or
- allowing delaying of existing lockages also when inserting new lockage operations (called *delayAll* setting, $d^+ = d^* = d$).

Algorithm 4.3: Mutable Chronological Deterministic Construction

Input: A list of trips $\mathcal{S}$,
maximum delay of existing lockage when adding to existing lockage d^+,
maximum delay of existing lockage when inserting a new lockage d^*
Data: A sorted list of arrival events E
Output: A three dimensional schedule X

begin
  $X \leftarrow \{\}$
  foreach $s \in \mathcal{S}$ **do**
    $E \leftarrow E \cup \mathit{arrivalAtFirstWaterGate}(s, \mathcal{G}_s)$
  end
  while $E \neq \{\}$ **do**
    $e \leftarrow \mathit{first}(E); E \leftarrow E \setminus e$
    $s \leftarrow \mathit{ship}(e)$
    $g \leftarrow \mathit{waterGate}(e)$
    $at^a_{s,g} \leftarrow \mathit{arrivalTime}(e)$
    $M \leftarrow \{\}$
    foreach $l_i \in \mathcal{L}_g$ **do**
      if $\mathit{shipDependsOnLockage}(s, l_i)$ **then**
        $M \leftarrow \{\}$
        next l_i
      if $z_{i,g} \geq (at^a_{s,g} - d^+)$ *AND* $\mathit{lockageSupportingShip}(l_i, s)$ **then**
        $M \leftarrow M \cup \mathit{movesAddingShipToExistingLockage}(s, l_i)$
      if $z_{i,g} \geq (at^a_{s,g} - d^*)$ **then**
        $M \leftarrow M \cup \mathit{moveAddingShipToNewLockage}(s, l_i)$
    end
    $m \leftarrow \mathit{chooseMove}(M)$
    $X \leftarrow \mathit{apply}(X, m)$
    $\mathcal{S}^* \leftarrow \mathit{shipsDependingOnLockage}(\mathit{lockage}(m))$
    foreach $s^* \in \mathcal{S}^*$ **do**
      $\mathit{updateArrivalEventOfShip}(E, s^*)$
    end
    $E \leftarrow E \cup \mathit{arrivalAtNextWaterGate}(s, \mathcal{G}_s)$
  end
end

The settings *delayAdding* and *delayAll* further require a parameter for the maximum lookahead time d.

Introducing the lookahead time adds the risk of creating circular dependencies between the lockages, as demonstrated in Figure 4.3. We assume a ship s passing water gates g and g'. The passage of g has already been scheduled at lockage operation l_2 (orange). We further analyse two possible scenarios.

In scenario A, there is a ship s' passing water gates g' and g in the inverse direction. It passes g at lockage l_1 right before l_2 (see picture subfigure A2). When s arrives at g', we assume both supporting lockage operations l'_1 and l'_3 are within the lookahead time. As subfigure A2 illustrates, s cannot be assigned to l'_1 because that would introduce a cyclic dependency: l_2 depends on l_1 (subsequent on the same gate), l_1 depends on l'_2 (ship s'), l'_2 depends on l'_1 (subsequent on the same gate), and finally l'_1 would depend on l_2 (ship s).

In scenario B, there is a ship s' passing only water gate g' in the inverse direction and reaching its destination $\delta_{s'}$ between the gates (see picture subfigure B1). Again, when s arrives at g', we assume both supporting lockage operations l'_1 and l'_3 are within the lookahead time. As subfigure B1 illustrates, s can be assigned to either lockage operation as no cyclic dependency is created. Subfigure B2 shows the consequences of assigning s to l'_1, which delays all lockage operations at g' and any dependent events (i.e., the arrival of s' at $\delta_{s'}$).

Generally, if a ship s has already been scheduled for some lockage operation l_x at g, scheduling it for an existing lockage l'_y or for a new lockage right before l'_y at g' is only possible if l_x does not depend on l'_y. To ensure this, while looking for candidates of l'_y all lockages (within the lookahead time) are iterated chronologically from earliest to latest. For each lockage l'_y at g', the algorithm fetches any direct dependent lockage l_z at g. If l_z is scheduled before l_x (i.e., $z < x$), l_x depends on l'_y. If such a dependency is found, all candidates for scheduling s at g found so far are removed, as l_x indirectly depends on all of them (implicitly as we are iterating the lockages at g' chronologically). This case can be seen in Figure 4.3, A2: l'_1 is added as a supporting lockage in first place, but when iterating over l'_2, the dependent l_1 is found, which is a dependency for l_2 and therefore all candidates found so far (i.e., l'_1) are removed.

We calculate the asymptotic runtime complexity as follows: For each gate passage of every ship ($O(s \cdot g)$) the current lockages at the given gate that start after the ship's arrival time (respecting the possible lookahead by d^+ or d^*) are iterated ($O(s)$) to find suitable moves. For every found move, the MCD recalculates the schedule by traversing dependent lockages ($O(s \cdot g)$). Finally, when the move is applied, all ships that are already (partially) scheduled and that depend on the newly added or altered lockage need to be identified ($O(s \cdot g)$) and their queued next arrival event needs to be updated ($O(s)$). From this we conclude $MCD(s, g) = (s \cdot g) \cdot (s \cdot (s \cdot g) + s \cdot g + s) = O(s^3 \cdot g^2 + s^2 \cdot g^2 + s^2 \cdot g) = O(s^3 \cdot g^2)$.

Our experiments finally show, that this estimation is very pessimistic. The MCD performs much better on larger instances than the MSD (see Figure 5.7). The reason for this is,

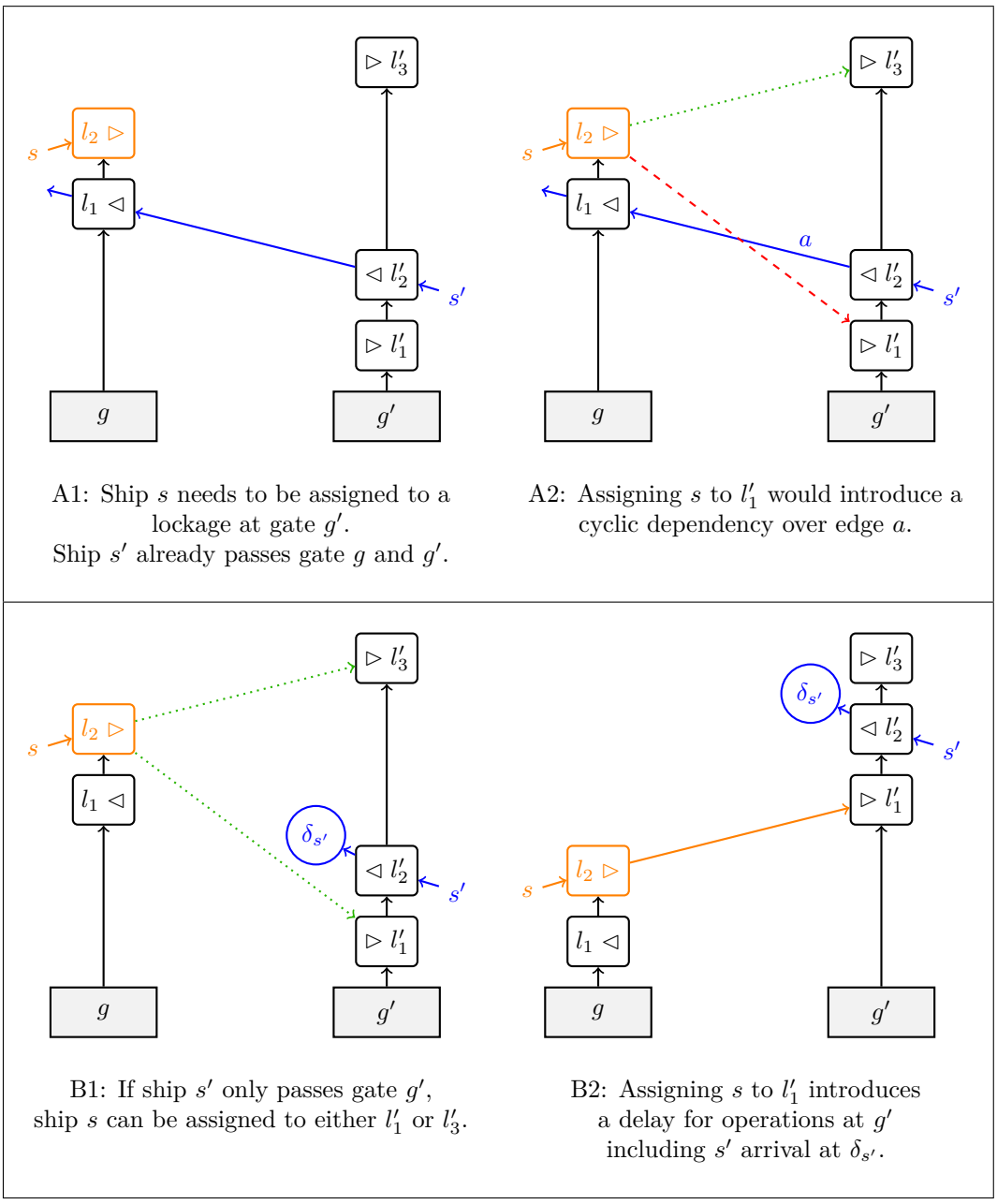


Figure 4.3: Example situation when a ship s needs to be assigned to an existing lockage at a gate g'. The graphs show time (i.e., lockage operations at a water gate) vertically, and space (i.e., water gates along the river) horizontally.

that due to the heuristics nature of chronological scheduling, the number of lockages that start at a gate after the lookahead is relatively small and rather constant with $O(1)$ than linear dependent on the number of ships, except for rare cases with highly congested water gates where many ships accumulate or very long lookahead times. Secondly, the number of dependent lockages is also relatively small and likely also has an expected runtime complexity of $O(1)$. When taking those two considerations into account, the average runtime complexity drops to $O(s^2 \cdot g)$.

4.2.7 Mutable Chronological Randomized Construction (MCR)

The Mutable Chronological Randomized Construction (MCR) is a randomized version of the MCD explained above. While previous randomized heuristics achieved randomization by perturbation of input data, the MCR does so by randomly choosing a possible candidate solution during construction, by changing the move selection method *chooseMove* in algorithm 4.3, line 21. As previous randomized construction heuristics, the MCR iteratively repeats the (adapted) MCD, as long as a better solution can be found within n iterations, where n is the number of ships in the instance.

We implemented two types of random move selection: biased and RCL.

- *Biased* move selection takes all possible candidate moves and chooses one of them randomly weighted by the reciprocal of their costs. Moves with high costs have a lower probability of being chosen.
- *RCL*-based move selection takes an additional parameter l that defines the length of the RCL. The algorithm takes the list of candidate moves sorted by their costs and chooses one of the first l moves with uniform probability. This results in a semi-greedy heuristics as proposed by Hart and Shogan in [HS87].

As the MCR calls a nested MSD multiple times, its runtime is $n \cdot O(MCD(s, g))$ where n is the number of iterations. For the MCR our experiments finally showed, that n has an upper bound of approximately $n < 7 \cdot s$, so we define the asymptotic runtime of $MCR(s, g) = n \cdot O(MCD(s, g)) = O(s^4 \cdot g^2)$ or $O(s^3 \cdot g)$ for the average case.

4.3 Neighborhood Structures

We developed several neighborhood structures for our local search based improvement heuristics. All neighborhood relations operate on solution representations based on the dependency graph introduced in Subsection 4.2.3. Initial solutions from construction heuristics that used another representation are migrated in an initial step.

In general, local search heuristics could allow neighborhood structures to contain moves that result in infeasible solutions. These infeasible solutions would need to be repaired later either by the neighborhood structure itself or by specialized subroutines, s.t. eventually

a feasible solution is returned. We decided to not allow our neighborhoods to contain infeasible solutions, i.e., solutions that contain cyclic dependencies between lockage operations. Still, when explaining individual neighborhood structures, we also state under which circumstances an infeasible solution could be created or repaired by the given neighborhood structure. If not noted otherwise, all neighborhood explorers support best improvement (BI), next improvement (NI) and random improvement (RI) step functions.

4.3.1 Reducing Total Waiting Time

In the ILSP we try to minimize the total waiting time of all scheduled ships. The ship's total travel time defined in Equation 1.1 can also be defined as $tt_s = at_s - r_s$ (i.e., the difference between start and arrival time). Assuming constant travel and service times, this implies that the challenge of the ILSP obviously is to minimize the ships arrival time. It further implies that the distribution of waiting time among the water gate passages of a ship does not affect the total travel or waiting time. When trying to improve a given schedule, it is necessary to shift waiting times to the end of the ship's trip to finally get the possibility to reduce it by improving the last lockage operation of a ship. To achieve this, we introduced an additional fractional value $\epsilon < 1$ that is added to the cost function and indicates the distribution of waiting time along the ships' trips. We define

$$\epsilon = \frac{\sum_{s \in \mathcal{S}} \sum_{g \in \mathcal{G}_s} \frac{wt^a_{s_g} + wt^c_{s_g}}{idx(g, \mathcal{G}_s)+2}}{\sum_{s \in \mathcal{S}} \sum_{g \in \mathcal{G}_s} wt^a_{s_g} + wt^c_{s_g}}, \tag{4.1}$$

where $idx(g, \mathcal{G}_s)$ is a function returning the 0-based index of g in $\mathcal{G}_s$.

For every gate passage of a ship, we weight the ship's waiting time by dividing it by the gate passage's number along the ship's trip (increased by two, to ensure that the weighted waiting time is always less than the unweighted waiting time). So waiting time at later gate passages is divided by a higher number and therefore accounts less to the weighted waiting time of the ship. The total sum over all ships is finally divided by the unweighted total waiting time, which is always strictly greater than the weighted waiting time, which enforces $\epsilon < 1$.

When applying a change to the schedule that does not change the (integral) total waiting time (i.e., the costs of the move are 0), but affects the waiting time distribution, we can compare ϵ before and ϵ' after the move to identify how the distribution changed. If $\epsilon > \epsilon'$, the move shifted waiting time closer to the end of the trips.

By integrating ϵ into the cost function when evaluating moves, we can use common local search techniques to improve the schedule, even when intermediate steps don't affect the ships' arrival times.

4.3.2 Generic Framework

To ease the development process, a framework was used that allows a generic definition of *move instructions*, *moves* and *neighborhood explorers* and *neighborhood structures*.

Most neighborhood structures operate on lockage operations or the ship assignments within the operations. We call them *Per-Lockage-Neighborhood Structures*. For those structures we implemented a generic neighborhood explorer that iterates over all water gates (outer loop) and their scheduled lockage operations (inner loop). If the lockage operation or the ships assigned to it can be used to generate neighboring solutions, all possible move instructions based on the given lockage are created. Generation of move instructions is very efficient in term of computational time as it only requires iterating possible combinations, depending on the neighborhood structure.

Depending on the chosen step function, move instructions are iterated and applied to the current solution. This results in individual moves that are evaluated and their results (i.e., the neighboring solutions) are compared to the incumbent solution. For RI or NI step functions, evaluation of move instructions stops as soon as an improvement is found and inner and outer loop are stopped preemptively. For BI, the best move found so far in this neighborhood structure is returned and used as a baseline for the next lockage operation (inner loop) or water gate (outer loop).

If RI is used, randomness is achieved by shuffling the order in which iterative loops are performed. First, the list of water gates is shuffled, but within a single water gate, all lockage operations are processed, before the next water gate is explored. Second, the list of lockage operations is shuffled, but again within a single lockage operation, all possible moves are evaluated, before the next lockage operation is explored. Finally, the list of move instructions within a lockage operation is shuffled. While this does not allow total randomness while exploring the neighborhoods search space, it is very efficient in terms of memory usage and computational time as it does not require a memory of already processed candidates or recalculation of random numbers if a candidate has already been explored.

A special case of Per-Lockage-Neighborhood Structures are *Ranged Neighborhood Structures*. These structures take lockage operations (or ships assigned to them) as the *source* element of a move and another lockage operation (or ships assigned to it) at the same gate as *target*. Source l_{i_g} and target l_{j_g} have a relative distance r_{ij} to each other, determined by their absolute index difference, i.e., $r_{ij} = |i - j|$. When calculating the possible move instructions for a certain source lockage operation, the explorer iterates over possible ranges in a third loop. Possible ranges can be constrained by defining a minimum and maximum range for the neighborhood structure. This allows to partition the neighborhood into multiple sub neighborhoods of the same type.

4.3.3 Swap Ships in Lockage (SwSiL)

The neighborhood structure Swap Ships in Lockage (SwSiL) creates neighboring solutions by swapping two ships that are assigned to the same lockage operation. It is a Per-Lockage-Neighborhood Structure that can handle any lockage operation with more than one ship assigned to it. Move instructions are created by iterating over all pairs of ships

assigned to the lockage operation, respecting the swap operations symmetry to avoid duplication with equivalent moves.

This neighborhood structure does not need to check for capacity or other constraints of ship assignments to lockage operations, as it only affects the order of ships within the operation, not their basic assignment to the operation as such. Also, it does not alter the dependency graph, so it neither creates nor repairs infeasible solutions.

4.3.4 Shift Ship in Lockage (ShSiL)

The neighborhood structure Shift Ship in Lockage (ShSiL) creates neighboring solutions by shifting a ship's position inside a lockage operation. It is a Per-Lockage-Neighborhood Structure that can handle any lockage operation with more than one ship assigned to it. Move instructions are created by iterating over all pairs of ships assigned to the lockage operation. As with the swap operation mentioned above, duplication with equivalent moves is avoided.

Like the SwSiL neighborhood above, the ShSiL does not affect capacity constraints or the dependency graph.

In general, any swap can be expressed by two shifts and any shift can be expressed by a series of swaps. Swaps with neighboring elements are even equal to shifts to neighboring positions.

The problem with basic local search (where only improvement steps are allowed) is, that any chain of multiple moves must consist of single steps that each provide an improvement for the whole chain to be accepted. As Table 4.1 shows, even for three elements swapping or shifting cannot reach all combinations with a single action. For four (and more) elements, more than half the possible permutations need multiple moves. For those reasonable sized element lists, shifting is more efficient in terms of maximum number of moves, than swapping. But the search space (i.e., the number of neighbors per move) of the ShSiL is nearly twice the size of the search space of SwSiL, which compensates the reduced number of moves.

4.3.5 Rearrange Ships in Lockage (ReSiL)

Driven by the consideration about necessary move chains for the SwSiL and ShSiL neighborhood structures, we developed another one that aimed on compensating the issue of requiring multiple improvement moves to reach any permutation of ship arrangements inside lockage operations. The Rearrange Ships in Lockage (ReSiL) neighborhood structure creates neighboring solutions by creating all permutations of ships that are assigned to the same lockage operation. It is a Per-Lockage-Neighborhood Structure that can handle any lockage operation with more than one ship assigned to it. Move instructions are created by iterating over all possible permutations of ships assigned to the lockage operation.

Like the SwSiL and ShSiL neighborhoods above, the ReSiL does not affect capacity constraints or the dependency graph.

	required moves		
ABC	swap	shift	rearrange
ACB	1	1	1
BAC	1	1	1
BCA	2	1	1
CAB	2	1	1
CBA	1	2	1
ABCD			
ABDC	1	1	1
ACBD	1	1	1
ACDB	2	1	1
ADBC	2	1	1
ADCB	1	2	1
BACD	1	1	1
BADC	2	2	1
BCAD	2	1	1
BCDA	3	1	1
BDAC	3	2	1
BDCA	2	2	1
CABD	2	1	1
CADB	3	2	1
CBAD	1	2	1
CBDA	2	2	1
CDAB	2	2	1
CDBA	3	2	1
DABC	3	1	1
DACB	2	2	1
DBAC	2	2	1
DBCA	1	2	1
DCAB	3	2	1
DCBA	2	3	1

Table 4.1: Possible combinations of three and four elements and how they can be reached.

As Table 4.1 shows, the number of required moves to reach any permutation of ships is always one, because a single move in this neighborhood applies the whole reordering in one step. But this drastically increases the search space per lockage operation, that now grows to $n! - 1$ (as opposed to $n \cdot (n-2) + 1$ for ShSiL and $\frac{n \cdot (n-1)}{2}$ for SwSiL), where n is the number of ships in a lockage.

4.3.6 Remove Empty Lockages (RemEmpty)

This neighborhood structure creates neighbors by removing one or two opposing empty lockage operation at the same water gate. It is a Per-Lockage-Neighborhood Structure that can handle only empty lockages. Move instructions are created by iterating over subsequent lockage operations at the same water gate.

When removing a lockage operation, whether it is empty or not, the subsequent lockage operations change their initial and target states. Removing an upstream lockage operation lowers all subsequent operations' states by one (i.e., *FULL* becomes *NEUTRAL*, *NEUTRAL* becomes *EMPTY*, *EMPTY* becomes *invalid*), while removing a downstream operation raises them. This can result in an infeasible chain of lockage operations when a subsequent downstream operation's new initial state gets *EMPTY* or an upstream operation's initial state gets *FULL*.

For each empty lockage operation, the neighborhood explorer determines an illegal state, that must not be reached by subsequent operations, for the given operation to be removable. If it tries to remove an upstream operation, subsequent operations must not have the *EMPTY* target state, as when the operation was removed, the target state would get invalid. If it tries to remove a downstream operation, subsequent operations must not have the *FULL* target state for the same reason.

The neighborhood explorer iterates over all subsequent operations as long as the invalid state is not found. If during iteration it finds another empty lockage with inverse direction, both lockages could be removed without breaking the lockage operations states between them. Subsequent operations would remain unchanged as the removal of the inverse operation negates the state change. In this case a possible move was found, consisting of the removal of two inverse empty lockages, and iteration of lockages continues. If the algorithm reaches the end of the list of lockage operations at the water gate without finding an illegal state, the empty lockage in question can also be removed solely without an inverse second operation.

Empty operations don't have dependencies to lockage operations at other water gates. They can be part of a cyclic dependency, but not the origin, so they can be removed or added without creating or repairing infeasible solutions.

4.3.7 Shift Ship (ShS)

The Shift Ship (ShS) neighborhood structure creates neighbors by shifting a single ship from one lockage operation to another at the same water gate. It is a Ranged Neighborhood Structure, so it is constrained by a minimum and maximum range r^{min} and r^{max}, that can handle only non-empty lockages.

For the current iterated range r at lockage operation l_{i_g} at water gate g it inspects the lockages l_{i-r_g} and l_{i+r_g}. Each target operation needs to exist and have the same direction as the source operation or be skipped otherwise. For each ship $s \in S_{i_g}$ assigned to the current lockage operation the capacity constraints at the target operation are checked.

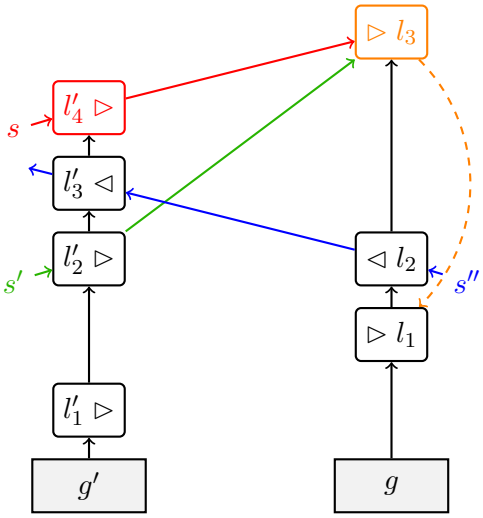


Figure 4.4: Dependency graph during a ShS evaluation. At water gate g, lockage l_{3_g} (orange), $r = 2$, there are two ships to move forward to l_{1_g}.
Shifting s (red) would result in a cyclic dependency because of s'' (blue).
Shifting s' (green) would be possible.

If the target operation has insufficient capacity or the move would introduce a cyclic dependency, the current ship s is skipped. For the remaining ships, move instructions for each possible target position in the target operation are created.

The lockage operations order at the water gate is not changed, so the feasibility of the current solution concerning the lockage chain at the water gate is unaffected. We distinguish between a forward shift to target operation l_{i-r_g} and a backward shift to l_{i+r_g}.

In case of a forward shift, the parent operation $l_{k_{g'}}$ on a neighboring water gate that l_{i_g} depends on because of ship s is queried from the graph (i.e., $s \in S_{k_{g'}}$). Iterating from l_{i-r_g} to l_{i_g}, we check if an inverse operation has a child operation $l_{k'_{g'}}$. If $k' < k$, then shifting s from l_{i_g} to l_{i-r_g} would introduce a cyclic dependency (see Figure 4.4).

In case of a backward shift, the inverse applies. $l_{k_{g'}}$ is the child operation that depends on l_{i_g} because of ship s. Iterating from l_{i+r_g} to l_{i_g}, we check if an inverse operation has a parent operation $l_{k'_{g'}}$. If $k' > k$, then shifting s from l_{i_g} to l_{i+r_g} would introduce a cyclic dependency.

4.3.8 Swap Ships (SwS)

The Swap Ships (SwS) neighborhood structure creates neighbors by swapping two ships from different lockage operations at the same water gate. It is a Ranged Neighborhood Structure, so it is constrained by a minimum and maximum range r^{min} and r^{max}, that can handle only non-empty lockages.

For the current iterated range r at lockage operation l_{i_g} at water gate g it inspects the lockage l_{i+r_g}. Due to the symmetry of swapping, there is no need to explore l_{i-r_g}. The target operation needs to exist and have the same direction as the source operation. Each ship $s \in S_{i_g}$ and $s' \in S_{i+r_g}$ is checked for cyclic dependencies if it was moved to the other lockage operation. This is similar to the check when shifting a single ship (see Subsection 4.3.7). If the ship would introduce a cyclic dependency, it is skipped. For each pair of remaining ships (s, s') the capacity constraints at both operations are checked. If any operation has insufficient capacity, the current pair of ships (s, s') is skipped. For the remaining pairs of ships, move instructions are created.

Like the ShS neighborhood above, the SwS does not affect the lockage operation's order at the water gate, so the feasibility of the current solution concerning the lockage chain at the water gate is unaffected.

Usually in combinatorics, a swap can be replaced by two shift operations. This is not true in this case, because of the capacity constraints. Consider two ships with lengths $k_s > \kappa_g/2$, so only one of them would fit in a lock chamber. They could never be swapped by applying two shifts (because of the restriction that moves must always result in feasible solutions).

4.3.9 Shift Lockage (ShL)

The Shift Lockage (ShL) neighborhood structure creates neighbors by shifting a single lockage operation to a new position at the same water gate. It is a Ranged Neighborhood Structure, so it can be constrained by a minimum and maximum range r^{min} and r^{max}. It can handle any lockage (empty and non-empty). For every lockage operation l_{i_g} at water gate g, it looks for potential new positions within the defined range $r \in [r^{min}, r^{max}$.

For the current iterated range r at lockage operation l_{i_g} at water gate g it inspects the lockages l_{i-r_g} and l_{i+r_g}. As shifting by one is a symmetric operation, the inspection of l_{i-r_g} is omitted in case $r = 1$. First, the shift is tested for cyclic dependencies. This is similar as described in Subsection 4.3.7, but tests all ships that are assigned to the lockage instead of just one. If a cyclic dependency would be introduced by the shift, it is skipped.

Shifting lockage operations does not only change the dependency graph but also the lockage chain at the water gate, which can become infeasible by the move. Therefore the ShL explorer can be configured to apply automatic repairing if the lockage chain would reach an infeasible state. Infeasible moves are repaired by removing or adding empty lockages in the lockage chain. Removal is always preferred over addition.

A shift divides the chain of lockages into three parts: the heading (indices 0 to $i - r - 1$) that is not directly affected by the shift, the middle (indices $i - r$ to i) that contains the lockage operations that are skipped, and the tailing (indices $i + 1$ and above) that is again not directly affected.

When moving a lockage operation l_{i_g} forward (to a lower index), it is first tested for feasibility on its new position. If l_{i_g} breaks the chain at its new position, the explorer checks the heading for an empty lockage with the same direction as l_{i_g} that can be removed without breaking the headings feasibility. If such an operation exists, it can be removed, which leaves the middle part unchanged. Otherwise a new empty operation is added right before the infeasible state at index $i - r$. If l_{i_g} can be inserted at the new position, the middle part is tested for infeasible states. Again, if such a state is found, the algorithm tries to fix it by removing a preceding empty lockage or adding an empty lockage right before the violation. If after checking the middle part no repairing was necessary, the tailing remains unchanged and does not need to be tested. If any empty lockages were added or removed, the same checks and repairs are applied to the tailing.

When moving a lockage operation l_{i_g} backward (to a higher index), the middle part is checked for infeasible states and (as in case of a forward shift) repaired by removing a preceding empty lockage or adding a new one. If the middle part stays feasible under the new circumstances, the tailing is unchanged and the check is done. Otherwise l_{i_g} is tested at its new position $i + r$. If it is infeasible, it is repaired and the tailing remains unchanged and can be skipped. If not, the remaining tailing needs to be checked and possibly repaired.

4.3.10 Swap Lockages (SwL)

The Swap Lockages (SwL) neighborhood structure creates neighbors by swapping two lockage operations at the same water gate. It is a Ranged Neighborhood Structure, so it can be constrained by a minimum and maximum range r^{min} and r^{max}. It can handle any lockage (empty and non-empty). For every lockage operation l_{i_g} at water gate g, it looks for potential swap partners within the defined range $[l_{i+r_g^{min}}, l_{i+r_g^{max}}]$. As swapping is a symmetric operation, only swaps with subsequent lockage operations are considered. Also swapping of empty lockage with equal directions is omitted.

First, the swap is tested for cyclic dependencies as described in Subsection 4.3.7. Like the ShL it tests all ships that are assigned to the lockages instead of just one. Cycle detection splits the swap operation in two shift operations, each of which is tested separately for the existence of cycles in the dependency graph. If a cyclic dependency would be introduced by the swap, it is skipped.

When swapping lockages of equal direction, the water gates lockage chain cannot become infeasible (or repaired if it was infeasible). The states of the lockage operations are not altered. When swapping inverse lockages, the lockage chain could become infeasible. Therefore this neighborhood structure allows automatic repairing of moves as described in Subsection 4.3.9. The lockage chain is copied and the swap is applied to the copy. Starting from the new l_{i_g} backward, the chain is tested for infeasible states. If one is found, it is repaired by removal or addition of empty lockages as described above. The iteration stops once it passes index $i + r$ and the current lockages state is unchanged or the end of the chain is reached. After index $i + r$ no further changes to the chain

have been applied, so whenever one lockage's state is unchanged, the remaining chain is unchanged as well.

4.3.11 Improve Trip (IT)

The Improve Trip (IT) neighborhood structure follows a different approach than the structures defined so far. While they operate on a single water gate's lockages either by shifting or swapping, IT operates on trips as a whole across all passed water gates. For each trip, the explorer creates all possible assignments to existing or new lockage operations using sequential scheduling and a depth first search (DFS). To narrow the search space and increase performance, the DFS stops if the resulting schedule would not be an improvement compared to the incumbent solution or the current best solution found so far during the DFS.

The ships are sorted by their total waiting time when using the NI step function or randomly when using RI. When using BI, the ordering is irrelevant as all trips are evaluated anyway. For each ship s, the algorithm starts with an initial schedule, based on the incumbent solution but with the ship removed from all lockage operations. As this might result in additional empty lockage operations that are no longer needed, unnecessary empty lockages are removed from the schedule by using the Remove Empty Lockages (RemEmpty) neighborhood structure with BI step function. This cleaning of the incumbent solution is deterministic, so it can be reapplied when finally a move candidate is chosen and applied.

The algorithm starts the DFS from the cleaned schedule. The incumbent solutions objective value serves as initial value for the threshold m. During the search, this threshold must not be exceeded, otherwise the resulting schedule would no longer be an improvement. For the first water gate, that the ship needs to pass, all possible schedules that result in objective values within m are created. We reuse functionality of the construction heuristic MCD to find all possible scheduling decisions for s at the current water gate. The IT can therefore be parametrized by the same settings (*simple*, *delayAdding*, *delayAll*) and lookahead time d. This allows for delaying existing lockage operations during addition and insertion as described in Subsection 4.2.6. For each possible decision, if the resulting schedules objective is below m, the DFS steps to the next water gate. This recursion stops if either the whole trip has been scheduled, which results in an improved schedule including s, or the threshold m cannot be undercut anymore, which means the current DFS branch cannot result in an improved schedule. Whenever an improvement is found, m is updated to the current best solutions objective value, so future evaluations are even more constrained to further narrow the search space.

The neighborhood has an exponential size of $O(m^3)$, as each ship is assigned to every lockage (and every possible new lockage) at every water gate. Such neighborhoods are called *very large-scale neighborhoods* (see Ahuja et al. [Rav02] for a thorough overview) and require a more elaborate way of finding and selecting promising neighbors than simple iteration. For that reason, we introduced the explained threshold m to only investigate

improving solutions. To further narrow the neighborhood, we only try to reschedule ships that at some water gate have a waiting time above a certain threshold t, which can be set as a parameter to the neighborhood explorer. That way the heuristic does not spend too much time looking for improvements of trips that are already relatively well scheduled and concentrates on trips that possibly bring bigger improvements. For the same reason, using the NI step function ships are evaluated sorted by their total waiting time.

By setting $t < 0$, all ships will be rescheduled, even those that do not encounter any waiting. This is the only neighborhood that is capable of joining or splitting lockages across multiple water gates simultaneously. All other neighborhoods operate on single water gates and therefore would need several steps to accomplish the same task, requiring them to allow intermediate worsening moves.

4.3.12 Rebuild Water Gate (RebWG)

The Rebuild Water Gate (RebWG) neighborhood structure operates on whole water gates, instead of individual lockages. It is the only neighborhood structure with an explorer that allows worsening moves. These are used to apply shakings in metaheuristics to escape local optima. This implies that the meaning of the step functions changes for the RebWG explorer. BI evaluates the rebuild of all water gates and returns the one with the best result, which is not necessarily an improvement. RI rebuilds a randomly chosen water gate and NI rebuilds the water gates sorted by the passing ships summed waiting time at the water gate. The basic idea of RebWG is to sort all ship arrivals at a water gate and reschedule them using FCFS with lookahead.

The RebWG can repair circular dependencies that include lockages at the rebuilt water gate, but requires the other water gates to not have any circular dependencies.

Rebuilding a water gate g is a greedy heuristic that breaks up and rebuilds a part of the schedule. This process is based on the Ruin and Recreate Principle defined by Schrimpf et al. in [SSSWD00]. Initially, all lockages at the water gate are removed. The algorithm keeps two lists for upstream and downstream going ships and their arrival time at g. A ship is only added to its according list if its arrival time is independent of any gate passage at g that has not yet been rebuilt. Iteratively ships are taken from those two lists and assigned to lockage operations which allows further dependencies at neighboring water gates to be resolved and new ships can be added to the lists of arriving ships at g. The RebWG defines a threshold t that defines the maximum in chamber waiting time for a ship and is used as the above mentioned lookahead time window when deciding to wait for another ship traveling in the same direction.

Depending on the earliest arriving ship and the current state of the water gate, possibly multiple ships are scheduled together in a new lockage operation. If the gate can handle the earliest arriving ship s^+, as many ships that arrive within t after $at^a_{s^+{}_g}$ as possible are assigned to a single new lockage operation. If the gate cannot handle the earliest arriving ship s^-, but the next supported ship s^+ arrives within the chamber service time τ_g after the next (unsupported) ship's arrival $at^a_{s^-{}_g}$, wait for s^+ and schedule as many

supported ships as possible that arrive within t after $at^a_{s^+{}_g}$. In any other case (i.e., the next supported ship's arrival time is more than τ_g after the next ship's arrival), an empty lockage operation is inserted. This makes the next earliest arriving ship a supported one and the decision making process can be repeated.

When new ships are scheduled at the water gate, dependencies to neighboring water gates are updated accordingly and resolved. This allows calculation of other ships' arrival times at the gate, so the lists of ship arrivals can be updated and queried. This loop continues until all ships have been rescheduled.

In general, Ruin and Recreate heuristics are considered very powerful for complex optimization problems, but their performance highly depends on the used recreation strategy (see Schrimpf et al. [SSSWD00]). The RebWG uses a very naive recreation method that does not primarily aim at optimizing an incumbent solution but rather on shaking a current local optimum (concerning other neighborhood structures). By changing a big part of the schedule at once, future improvement steps will hopefully not lead into the same local optimum again.

4.3.13 Asymptotic Runtime Complexity

We consider the same characteristics of the input data for our runtime complexity of the presented neighborhoods as for the construction heuristics, the number of ships $s = |\mathcal{S}|$ and the number of water gates $g = |\mathcal{G}|$.

The rather similar neighborhoods SwSiL, ShSiL and ReSiL follow the same basic algorithm. For every lockage operation ($O(s \cdot g)$), assuming n ships per lockage operation, all swap ($\frac{n \cdot (n-1)}{2}$), shift ($n \cdot (n-2)+1$) or rearrange ($n! - 1$) combinations are evaluated. For meaningful real life instances n typically has an upper bound of 3 or 4, depending on the dimensions of the lock chambers and ships. In our experiments with more than 300,000 calculated solutions, we encountered only five solutions that had a single lockage operation with 5 ships assigned to them. We therefore treat them as independent on the size of the input, i.e., $O(1)$. Finally each created move needs to be evaluated ($O(s \cdot g)$), so we conclude a runtime complexity of $\mathit{ShSiL}(s,g) = \mathit{SwSiL}(s,g) = \mathit{ReSiL}(s,g) = O(s^2 \cdot g^2)$

The RemEmpty neighborhood iterates over all water gates ($O(g)$) and for each lockage ($O(s)$) needs to test if it can be safely removed ($O(s)$). Again, the created moves need to be evaluated ($O(s \cdot g)$), so we conclude a runtime complexity of $\mathit{RemEmpty}(s,g) = O(g \cdot s^2 \cdot g \cdot s) = O(s^3 \cdot g^2))$

The ranged neighborhoods ShS, SwS, ShL and SwL create at most 2 moves per range. We define $r^\delta = r^{max} - r^{min}$. Again, this is not part of the input size and can therefore be ignored for the runtime complexity. When shifting or swapping whole lockage operations or single ships, circular dependencies can be introduced into the schedule. Those neighborhoods need to check if a move is feasible by checking for such potential cycles in the dependency graph. This requires iterating all lockages and ships that are

indirectly affected by the move, which are at most $n \cdot r^{\delta}$ ships. This feasibility check is therefore not depending on the input size and has $O(1)$.

For the ship moving neighborhoods, the number n of ships per lockage is also not dependent on the input size as stated above. For each lockage operation ($O(s \cdot g)$ they create at most $n \cdot r^{\delta} \cdot 2$ ($O(1)$) moves. As each move needs to be evaluated ($O(s \cdot g)$), we conclude $ShS(s, g) = SwS(s, g) = O(s^2 \cdot g^2)$.

The lockage moving neighborhoods need to additionally check for lockage chain feasibility and empty lockage addition or removal, which requires iterating through the whole lockage chain once ($O(s)$). This increases the runtime complexity to $ShL(s, g) = SwL(s, g) = O(s^2 \cdot g^2 + s^2 \cdot g)$.

The most complex neighborhood is IT, which uses a depth first search to test all possibilities of reinserting a ship into the schedule. Even though we use a couple of methods to decrease its runtime, its complexity can be summarized as $IT(s, g) = O(s^g)$

The RebWG neighborhood iterates over all water gates ($O(g)$) and removes all lockages ($O(s)$) and tries to reschedule them according to the above described strategy. Rescheduling the ships requires updating the schedule which has $O(s \cdot g)$, so we conclude $RebWG(s, g) = O((s^2 \cdot g^2)$.

4.4 Metaheuristics

In the above section we defined a set of neighborhood structures. Most of them only manipulate the schedule at some level, i.e., they exclusively change either lockage operations or the ship positioning inside the lock chamber. In a simple local search with only one single neighborhood structure, only a small fraction of the total solution space (i.e., the set of feasible solutions) could be explored. Those structures need to be used in conjunction with each other to unfold their full potential. This conjunction is provided by metaheuristics that operate on multiple neighborhoods. This basic principle is called Variable Neighborhood Search (VNS).

In this section we will explain the basic idea and elements of VNS, and show some extensions and derivatives of it and how we implemented them. A detailed discussion based on experimental results is provided in Chapter 5.

4.4.1 Variable Neighborhood Search (VNS)

The Basic VNS was first described by Mladenovic and Hansen in [MH97]. By systematically changing the used neighborhood for a local search, they provided a simple and effective metaheuristic for combinatorial optimization problems. The basic procedure is described in algorithm 4.4.

The VNS relies upon the following observations (see [HMMP10] for a detailed discussion):

Algorithm 4.4: Basic Variable Neighborhood Search

Input: An initial solution x,
a set of neighborhood structures $N_k, k = 1, ..., k_{max}$

```
1  begin
2      k ← 1
3      while k <= k_max do
4          x' ← applyRandomMove(x, N_k)
5          x'' ← findLocalOptimum(x', N_k)
6          if f(x'') < f(x) then
7              x ← x''
8              k ← 1
9          else
10             k ← k + 1
11         end
12     end
13 end
```

- A local minimum with respect to one neighbourhood structure is not necessarily a local minimum for another neighbourhood structure.
- A global minimum is a local minimum with respect to all possible neighbourhood structures.
- For many problems local minima with respect to one or several neighbourhoods are relatively close to each other.

When modelling a VNS or any derivative or extension of it, a couple of questions need to be answered:

- What neighborhood structures shall be used?
- How are they used to improve the current solution? (E.g., use local search with BI to find a local optimum.)
- What strategy is used to change the neighborhoods? (E.g., basic VNS uses N_{k+1} if no improvement was found or N_1 otherwise.)
- How do we escape local optima? (E.g., basic VNS applies random moves in N_k before looking for the local optimum in N_k.)

4.4.2 Variable Neighborhood Descent (VND)

The Variable Neighborhood Descent (VND) metaheuristic uses a predefined, deterministically ordered set of neighborhood structures. It does not apply random moves or any

kind of shaking to escape local optima. Instead, it tries to strictly improve the incumbent solution (i.e., descend "down" without stepping back "up") by switching to the next neighborhood structure, until no improvement can be found. In contrast to the basic VNS, VND does not explore the current neighborhood to its local optimum, but restarts from the first neighborhood structure if any improvement (usually either next or best) is found.

The local search usually (but not necessarily) uses a BI step function. If the local search used is deterministic (i.e., it uses BI or NI step function), the whole metaheuristic is deterministic. The neighborhood structures are often ordered by size or complexity, so smaller or more efficient neighborhood structures are explored more often and large structures are evaluated late in the process. Sometimes the neighborhoods are nested, i.e., $N_k \subset N_{k+1} \forall k < k_{max}$.

4.4.3 Token-Ring Neighborhood Search (TRNS)

The Token-Ring Neighborhood Search (TRNS) metaheuristic as defined by DiGaspero and Schaerf in [DGS03] follows a similar approach as the VND with a minor but significant difference. Instead of restarting the iteration of neighborhood structures upon the first improvement, the TRNS explores the current neighborhood to its local optimum and then always proceeds to the next neighborhood structure. If only two neighborhood structures are used in this way, this is also called a *tandem search*.

The TRNS was intended to be used to alternate between intensification (i.e., advancing towards an improved solution) and diversification (i.e., escaping from local optimum), but it can be used in general if multiple neighborhoods are coequal and the hierarchical ordering of a VND is not applicable.

In general, the TRNS stops after a predefined number of rounds without improvement. If no diversification neighborhood structures are used (i.e., no worsening moves are allowed), a single round without improvement suffices.

4.4.4 Neighbourhood Search with Configurable Neighbourhood Iteration (NSCNI)

Investigating the differences between VND and TRNS, we felt the need for a metaheuristic that gives the developer full control over the change of neighborhoods and offers more flexibility than the static default patterns of VND or TRNS. We developed a simple metaheuristic framework called Neighbourhood Search with Configurable Neighbourhood Iteration (NSCNI) that follows the same basic process, but adds a *neighborhood selector* that decides on which neighborhood structure is explored next if an improvement was found in the current neighborhood. Its decisions are based on simple predefined patterns called the *success mode* that can be set for each neighborhood structure individually:

- *FIRST*: Go to the first neighborhood. This corresponds to VND.

- *SAME*: Stay at the same neighborhood. This will result in a local optimum w.r.t the current neighborhood. This corresponds to TRNS.
- *NEXT*: Proceed to the next neighborhood. This resembles TRNS without local optimum.
- *SAMEFIRST*: Stay at the same neighborhood until a local optimum is found. If the current neighborhood yielded an improvement, go to the first neighborhood. This resembles VND with local optimum.

In case no improvement is found, the algorithm always proceeds to the next neighborhood structure.

By assigning the same success mode to all neighborhood structures, the NSCNI can behave identical as VND (if the mode is *NEXT*) or TRNS (if the mode is *SAME*). But by combining the individual setting of success mode and step function per neighborhood structure, the NSCNI offers more versatility. That way large or less efficient neighborhoods (like IT defined above) can be explored less often (e.g., without local optimum and with random or next improvement), while smaller neighborhoods can be explored more often and thoroughly (e.g., best improvement with local optimum).

Examples for NSCNI configurations and the respective results are given in the next chapter.

4.4.5 General Variable Neighborhood Search (GVNS)

An extension to the basic VNS algorithm that is considered powerful (see [HMMP10]) is called General Variable Neighborhood Search (GVNS). It uses two sets of neighborhood structures, where one is used for diversification moves (or shakings) and the other used for an embedded VND that replaces the nested local search of the VNS.

Our implementation of the GVNS metaheuristic uses the above defined NSCNI instead of a VND. Further, the algorithm terminates if after a shaking has been applied and the nested NSCNI could not find a better solution w.r.t. the solution before the shaking was applied.

Algorithm 4.5: General Variable Neighborhood Search with nested NSCNI

Input: An initial solution x,
a set of shaking neighborhood structures $N_k, k = 1, ..., k_{max}$,
a set of NSCNI neighborhood structures $N'_k, k' = 1, ..., k'_{max}$

```
1  begin
2      k ← 1
3      while k <= k_max do
4          x' ← shake(x, N_k)
5          x'' ← NSCNI(x', N')
6          if f(x'') < f(x) then
7              x ← x''
8              k ← 1
9          else
10             k ← k + 1
11         end
12     end
13 end
```

CHAPTER 5

Experimental Results

All experiments were conducted on a 64-bit Linux PC with an AMD Ryzen 5 2600 CPU (3.40 GHz) and 32 GB RAM. The cost factors applied to the objective function are $C(1, 1000)$, so a single lockage operation is as costly as 1000 seconds waiting time.

For our heuristic solution methods, we did not apply any runtime limitations, but let them try to improve their solution until no further improvement could be found or an iteration based stopping criterion defined for the respective heuristic was met. For our construction heuristics, this is defined in Subsection 4.2.2, Immutable Sequential Randomized Construction (ISR). We decided to do so because early experiments showed, that larger instances, especially those using real life data of a full day, require much more computational time than smaller synthetic instances. While the latter could be solved relatively well even to optimality, larger instances take much more computational time. The ILSP is a rather unknown computational problem with no established benchmarks or published problem instances, so we wanted to evaluate our algorithms capabilities in solving the problem instances in first place and also find the best possible solutions to our specific problem instances.

The basic procedure how the results of our experiments are compared and evaluated is described in Section 3.5, Experimental Evaluation. This chapter gives a more detailed insight on the instance data and discusses the results of the different algorithms.

5.1 Instance Data

Historic instance data was provided by viadonau and consisted of basic ship data (name, type, dimension), trip data (direction, start and destination time and location) and timestamps showing the date of passage at each passed water gate (time entering and leaving the surrounding area and the water gate itself). From this information - assuming completeness and correctness - a vessels voyage along the river can be reconstructed.

But this data lacks information about how fast a ship could have traveled along a river section and therefore does not allow assumptions about possible improvements to its total traveling time. Further, the historic data is partially incomplete (missing timestamps about water gate passages during a trip) or inconsistent (e.g., leaving time is before entering time).

The historic instance data originally contains more trips than stated here. This is because we used only trip information that starts and ends on the same day, passes at least one water gate and holds a simple feasibility check: We only take into account ships that - based on their travel speed per river section, entering durations and exiting durations, could hypothetically without any further waiting times arrive at their destination before their historic arrival time. All other trips are ignored as they obviously contain inconsistent data.

Some trips also started already inside of gate areas or even inside lock chambers. As the model (and the heuristics) require the ships to enter the area before entering the chamber, those trips where altered to start right at the entrance to the gates area. Even though this change of the original instance data is small and neglectable from a practical point of view, it makes our results incomparable with earlier publications by Prandtstetter et al. ([PRSR15] and [RRP15]).

As stated in Section 3.5, Experimental Evaluation, the instances can contain multiple trips per ship. Even though they would depend on each other in a real world application, we handle them as independent trips in our instance data as if conducted by individual ships.

5.1.1 Instance Sets

The first set of benchmark instances consists of 180 instances that are rather small with two to 25 ships and five to 153 gate passages. We call this set "*small instances*" therefore. It has previously been used by Ruthmair and Ritzinger in [RRP15].

Instances of this set are categorized by difficulty and distribution. We define three difficulty levels, depending on the number of gate passages per ship. *Easy* instances have on average at most three passages per ship, *medium* instances have at most five and *hard* instances have more than five. The distribution of an instance is determined by the average start time of the trips. It can be either *close* with on average two or more ships starting per hour, or *distributed* otherwise.

The second set of instances consists of 60 instances representing real traffic data of whole days (with the above mentioned restrictions). We call it "*one day instances*" therefore. Instance sizes range from 31 to 126 ships with 60 to 342 gate passages. Average gate passages per ship range from 1.54 to 2.95 with an average of 2.3. The average start time distribution ranges from 1.9 ships per hour to 5.9 with an average of 3.6. Compared with the classified small instances above, the one day instances would be *easy* and *close* (with one exception, that would be *distributed*).

5.2 Comparison and Evaluation Techniques

As described in Section 3.5, Experimental Evaluation, we use two methods to compare and evaluate our heuristic solution methods, especially because there are not enough optimal known solutions or tight bounds.

5.2.1 Optimal Solutions

As mentioned above, there are no published benchmark instances to the ILSP and therefore no optimal or best known solutions that can be used as a baseline to compare and evaluate the heuristics. We used the best heuristic solutions per instance we could find as an initial solution for the MIP model of Ruthmair et al. and passed them to a MIP solver to either get an optimal solution of the instance or at least a lower bound to compare with. We used ILog CPlex solver version 12.5 configured for single threaded execution, a time limit of 8 hours and a memory limit of 10 GB.

When comparing heuristic results to solutions from a MIP solver, we are mainly interested in two metrics:

- The relative gap defines the difference between the upper and lower bound of a solution obtained from a MIP solver. It is measured in percent of the difference between the bounds divided by the upper bound.
  A proven optimal solution has a relative gap of 0 %.
  A feasible solution with bounds $[90, 100]$ has a relative gap of 10 %.

- The optimality gap defines the difference between a solution obtained from a heuristic and the proven optimal solution. It is measured in percent of the optimal solution.
  If a heuristic is able to find the optimal solution, it has an optimality gap of 0 %.
  If a heuristic's solution is 1.2 times the proven optimal solution, it has an optimality gap of 20 %.

Small instances

- For 90 of 180 (50 %), a proven optimal solution could be found by at least one heuristic algorithm.

- For 3 of 180 (1.67 %), a proven optimal solution was only found by the MIP solver. The best heuristics have an optimality gap of 0.21 % to 2.75 % on those instances.

- For 6 of 180 (3.33 %), the MIP solver found an improved solution by 0.03 % to 1.92 %, that is not proven to be optimal (with relative gap from 18,12 % to 47,47 %).

- For 81 of 180 (45 %), the best known solution was found by at least one heuristic algorithm (with relative gap from 3.63 % to 63,14 %).

One day instances

- For no instance a proven optimal solution could be found by either the heuristics or the MIP solver.
- For 1 of 60 (1.67 %), the MIP solver found an improved solution by 0.15 %, that is not proven to be optimal (with a relative gap of 62.30 %).
- For 59 of 60 (98.33 %), the best known solution was found by at least one heuristic algorithm (with relative gap from 39.30 % to 81.40 %).

From these results we conclude, that the MIP model is insufficient for solving real life instances. The high relative gap on one day instances and even on larger small instances render the lower bound of the MIP solutions useless as a baseline for comparison. Still, the high number of optimal solutions (90 of 93) that where found by heuristics and the low number of improvements of the MIP solver compared to the heuristics (10 of 240) show the high solution quality of the heuristic solution methods.

5.2.2 Hypothesis Testing

We use statistical hypothesis testing to pairwise compare different settings and create a score board to rank the performance of multiple different settings on a set of instances. We use a (paired) Wilcoxon signed-rank test for these tests. Therefore two algorithms (or settings of an algorithm) are compared against each other, sorting all results by instance and objective value. Each tuple (pair of results) for the test now represents the n^{th} best solution of both compared algorithms for the same instance. For deterministic algorithms, each instance under test is run once, resulting in one pair of data for each instance. For non-deterministic algorithms, each instance is solved 5 times, where each run uses a different initial random seed ($i \cdot 1,000$ for the i^{th} run).

To compare these runs of different algorithms with each other, an unpaired Wilcoxon rank-sum test (or Mann-Whitney-U-Test) would usually be used. But as these runs are done on several instances, the Wilcoxon rank-sum test would be run on each instance separately. The results would then only show, which algorithm performs better on a specific instance. The overall performance comparison would require further statistical analysis of the results per instance. Alternatively, the paired test could be run on the average (mean or median) results of the 5 non-deterministic runs. In any case, statistical analysis would be less accurate.

Therefore the decision was made, to run a paired two-sided test on sorted results per instance, resulting in a single *p-value* showing if the two algorithms perform equally or not on all tested instances. If the p-value is below a threshold of 0.05 (meaning that the algorithms perform significantly unequal) a one-sided test is run to test which one performs better. (The threshold for the one-sided test is halved to 0.025.) The one-sided test now shows, if one algorithm performs significantly better than the other.

While this is not the most exact way to compare two algorithms, it allows fast evaluation of multiple runs on multiple instances with different algorithms.

When more than two algorithms (or settings) are compared with each other, each pair of algorithms is compared. As a result of this pairwise comparison, one algorithm is either considered better than the other (at a confidence interval of $p < 0.025$) or both are considered equal, i.e., not significantly different. As a fast evaluation method, all algorithms compete against each other and are rated by the number of algorithms they outperformed. Each algorithm scores a *win* point for each direct competition it won. The algorithms with the most points can be considered significantly better than the others, allowing a more detailed comparison of the few best performing algorithms. In case two algorithms won the same number of comparisons, we break the tie by considering the number of competitions that resulted in equal.

5.2.3 Performance Profiles

The second technique used are performance profiles as described by Dolan and Moré in [DM02]. The performance profile for an algorithm or solver is the (cumulative) distribution function for a performance metric.

It shows the probability (i.e., the percentage of instances) for an algorithm to find a solution with a relative threshold τ of the baseline. Usually, performance profiles are used to compare different solvers with each other by their computational time or resource consumption on a defined set of benchmark problems. We used the technique to assess the solution quality and runtime of heuristic optimization algorithms. We therefore show a combined figure of four graphs:

- The upper row shows the results on the small instance set, the bottom row on one day instances.
- The left graphs show the *normalized* performance profile on the objective value. Unlike standard performance profiles, the baseline here is not the best value of the compared settings shown, but the overall best known solution (including solutions from the MIP model) for each instance, so these graphs are even comparable with each other, just showing a different subset of competing settings.
- The right graph shows the (unmodified) performance profiles of the compared settings runtimes. We use a logarithmic scaling for the runtime, as some settings can divert from each others by high factors which would make the graphs unreadable in some cases.

5.3 Construction Heuristics

We first evaluate our construction heuristics and try to point out their pros and cons.

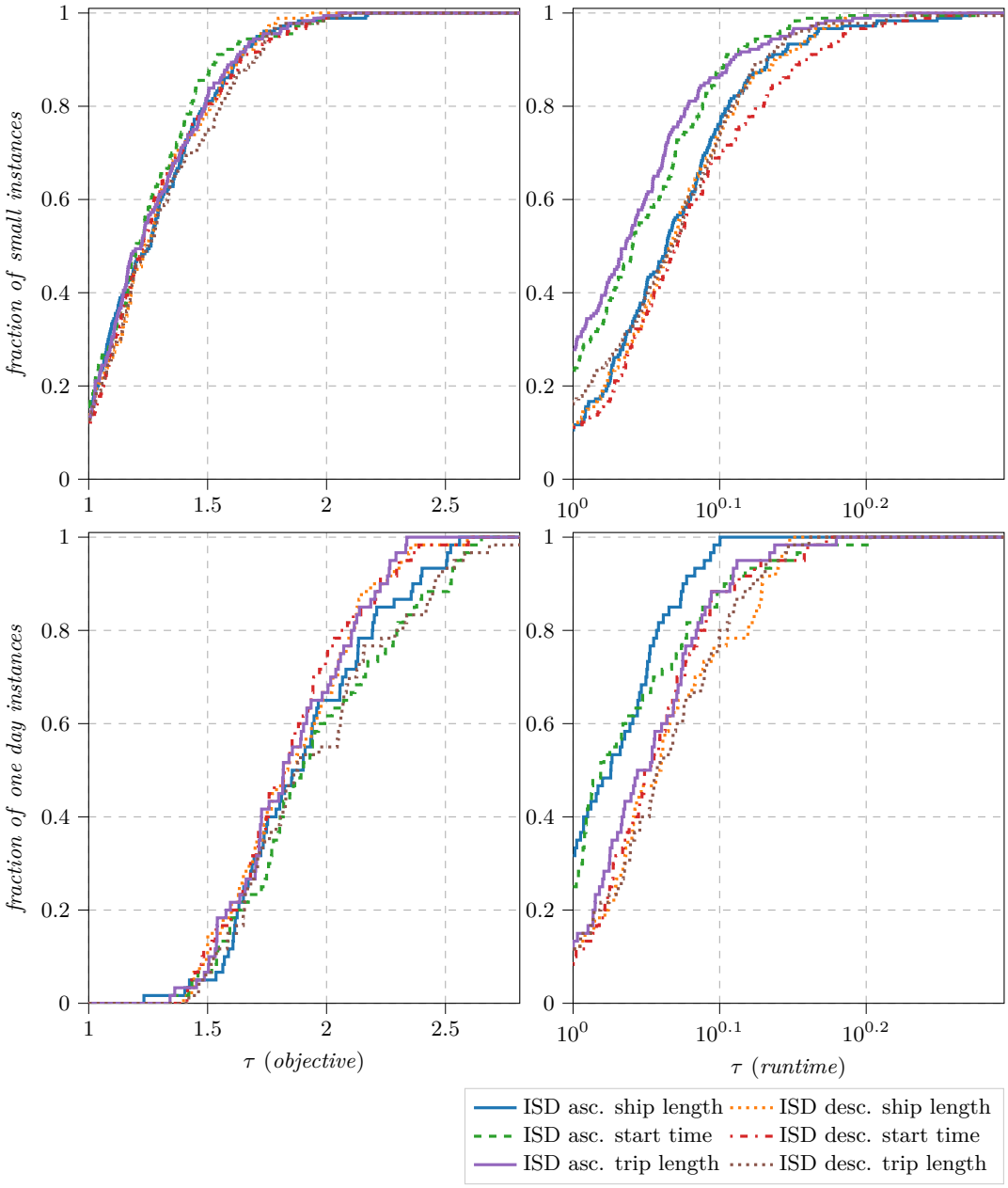


Figure 5.1: Performance profiles of ISD settings.

5.3.1 Immutable Sequential Deterministic Construction (ISD)

As stated in the description of the algorithm, this simple construction heuristic is highly affected by the ordering of the ships while they are added to the schedule. To show the impact of the initial ordering, six different orderings where analyzed:

- by trip starting time (ascending/descending)
- by ship length (ascending/descending)
- by trip length (distance traveled, ascending/descending)

Compared with each other the results shown in Table 5.1 where found. Figure 5.1 shows the performance profiles of the algorithms. In general, all settings perform comparably equal. Each of them can solve about 12 to 16 % of small instances even to optimality, the largest one with 13 ships, with a worst case gap of 89 to 118 %. Larger instances are at best solved within 32 to 42 % of the best known values, and within 133 to 175 % in the worst case.

It can be seen that *asc. start time* is in general significantly better on small instances, but decreases in performance on one day instances, compared to the other settings. Sorting by start time resembles a FCFS policy, based on release times (i.e., the departure time of the ship's trip). Our results indicate, that a simple, naive implementation of that policy might work well on small instances of the ILSP, but is not suitable for large instances.

Another interesting observation is, that on one day instances, the three better settings are significantly better than their reverse orderings. That correlation cannot be observed on small instances.

5.3.2 Immutable Sequential Randomized Construction (ISR)

For the randomized version the deterministic orderings from the ISD are adopted and different RCL parameters are evaluated. Cardinality-based RCL are constructed with $l \in 3, 5, 10, 20$ and percentage-based with $\alpha \in 0.10, 0.20, 0.30$. For direct comparison, we add the best ISD settings of each instance set to the ranking in Table 5.2.

As stated above, small instances have at most 25 ships, so any setting with an RCL of $l = 20$ behaves almost totally random. For big instances, the true random setting is significantly better than all other settings. Ranks 2 to 12 on one day instances are all but one taken by settings with the largest RCL parameters (either $l = 20$ or $\alpha = 0.3$). From this observation, we conclude, that the ISR benefits rather from randomization than from a reasonable ordering of the ships.

	small instances			one day instances		
rank	setting	wins	ties	setting	wins	ties
1	ISD asc. start time	4	1	ISD desc. start time	3	2
2	ISD asc. trip length	1	4	ISD desc. ship length	3	2
3	ISD desc. start time	1	3	ISD asc. trip length	3	2
4	ISD desc. ship length	1	3	ISD asc. ship length	2	0
5	ISD asc. ship length	1	3	ISD desc. trip length	0	1
6	ISD desc. trip length	0	0	ISD asc. start time	0	1

Table 5.1: Ranking of ISD heuristics.

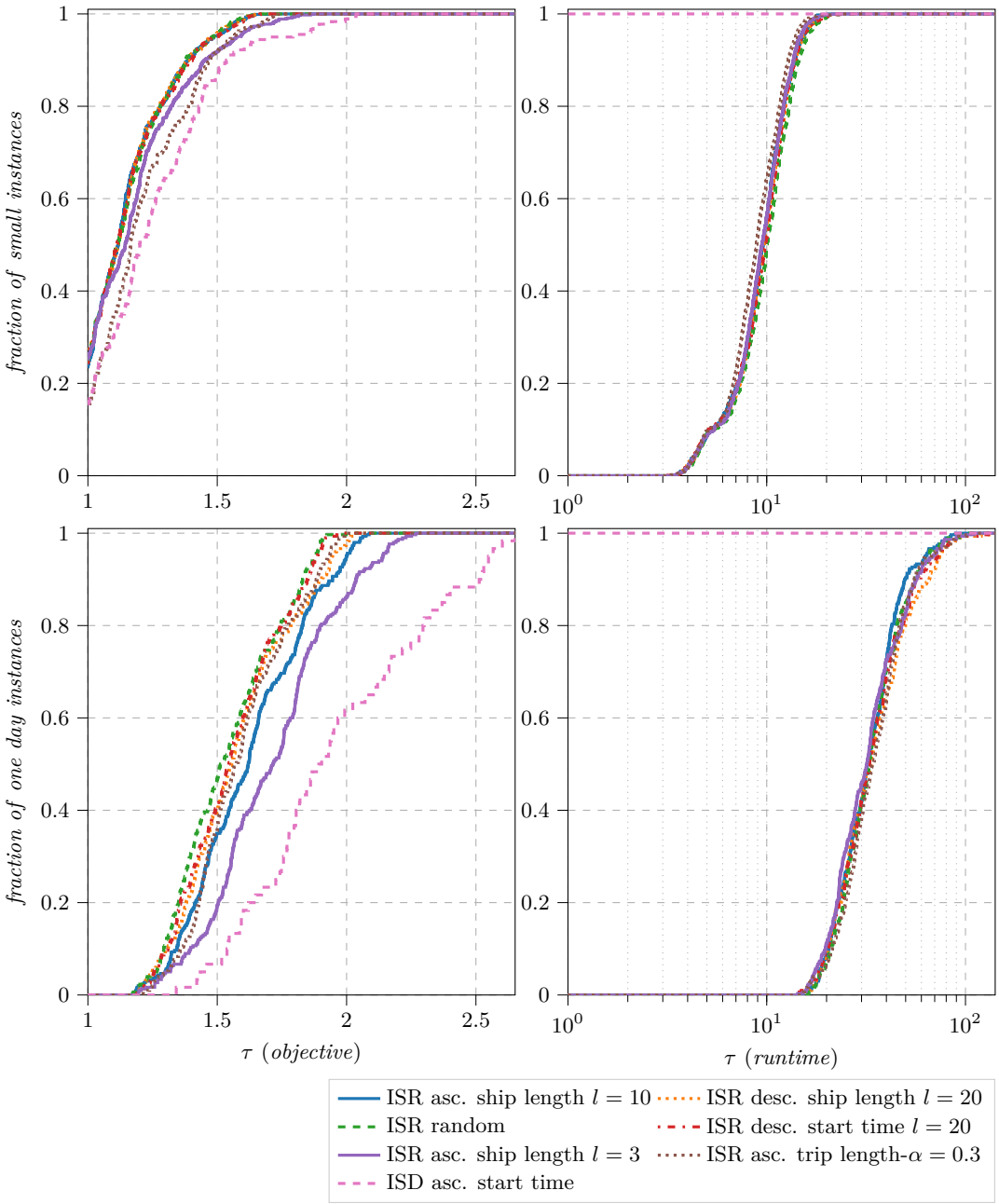


Figure 5.2: Performance profiles of selected ISR and ISD settings.

	small instances			one day instances		
rank	setting	wins	ties	setting	wins	ties
1	ISR asc. ship length $l = 10$*	42	2	ISR random*	44	0
2	ISR desc. ship length $l = 20$*	39	5	ISR desc. start time $l = 20$*	43	0
3	ISR asc. start time $l = 20$	36	7	ISR asc. trip length $l = 20$	41	1
4	ISR desc. ship length $l = 10$	35	9	ISR asc. ship length $l = 20$	38	3
5	ISR random*	35	8	ISR desc. ship length $l = 20$*	37	5
6	ISR desc. trip length $l = 20$	34	9	ISR asc. start time $l = 20$	37	4
7	ISR desc. start time $l = 20$*	34	8	ISR desc. start time $l = 10$	35	6
8	ISR asc. start time $l = 5$	34	8	ISR asc. trip length $\alpha = 0.3$*	34	6
9	ISR asc. start time $l = 10$	34	8	ISR asc. ship length $\alpha = 0.3$	32	6
10	ISR asc. trip length $l = 20$	34	6	ISR desc. ship length $\alpha = 0.3$	32	4
11	ISR asc. ship length $l = 20$	34	6	ISR desc. trip length $l = 20$	31	7
12	ISR asc. trip length $l = 10$	31	2	ISR asc. start time $\alpha = 0.3$	31	6
13	ISR desc. trip length $l = 10$	30	3	ISR asc. trip length $l = 10$	29	7
14	ISR desc. start time $l = 10$	30	3	ISR asc. trip length $\alpha = 0.2$	24	9
15	ISR asc. start time $l = 3$	29	3	ISR desc. start time $l = 5$	23	9
16	ISR asc. start time $\alpha = 0.3$	28	1	ISR desc. start time $\alpha = 0.3$	23	9
17	ISR asc. ship length $l = 5$	25	5	ISR desc. ship length $\alpha = 0.2$	23	9
18	ISR asc. start time $\alpha = 0.2$	24	4	ISR desc. ship length $l = 10$	23	8
19	ISR desc. ship length $l = 5$	23	4	ISR asc. start time $l = 10$	21	10
20	ISR asc. trip length $l = 5$	23	4	ISR asc. ship length $l = 10$*	21	10
21	ISR asc. start time $\alpha = 0.1$	23	4	ISR asc. ship length $\alpha = 0.2$	21	10
22	ISR asc. ship length $\alpha = 0.3$	22	6	ISR asc. start time $\alpha = 0.2$	20	10
23	ISR desc. start time $l = 5$	21	2	ISR desc. start time $\alpha = 0.2$	20	6
24	ISR desc. trip length $l = 5$	20	2	ISR asc. trip length $l = 5$	16	8
25	ISR asc. ship length $l = 3$*	18	3	ISR desc. ship length $\alpha = 0.1$	15	8
26	ISR desc. ship length $l = 3$	17	3	ISR asc. start time $\alpha = 0.1$	14	7
27	ISR asc. trip length $l = 3$	17	3	ISR asc. trip length $\alpha = 0.1$	13	10
28	ISR asc. ship length $\alpha = 0.2$	16	3	ISR desc. trip length $l = 10$	13	8
29	ISR desc. start time $l = 3$	15	2	ISR desc. start time $l = 3$	13	8
30	ISR desc. ship length $\alpha = 0.3$	14	2	ISR asc. ship length $\alpha = 0.1$	13	7
31	ISR desc. trip length $l = 3$	14	1	ISR desc. start time $\alpha = 0.1$	12	7
32	ISR asc. trip length $\alpha = 0.3$*	12	1	ISR desc. ship length $l = 5$	10	7
33	ISR asc. ship length $\alpha = 0.1$	11	2	ISR asc. start time $l = 5$	9	6
34	ISR desc. ship length $\alpha = 0.2$	8	4	ISR asc. trip length $l = 3$	8	5
35	ISR desc. start time $\alpha = 0.3$	8	3	ISR asc. ship length $l = 5$	8	5
36	ISR asc. trip length $\alpha = 0.2$	7	4	ISR desc. trip length $\alpha = 0.3$	8	4
37	ISR desc. trip length $\alpha = 0.3$	6	4	ISR desc. ship length $l = 3$	7	3
38	ISR desc. start time $\alpha = 0.2$	6	4	ISR desc. trip length $l = 5$	7	2
39	ISR desc. ship length $\alpha = 0.1$	6	2	ISR desc. trip length $\alpha = 0.2$	6	0
40	ISR asc. trip length $\alpha = 0.1$	4	1	ISR desc. trip length $l = 3$	3	2
41	ISR desc. start time $\alpha = 0.1$	2	3	ISR asc. start time $l = 3$	3	2
42	ISR desc. trip length $\alpha = 0.2$	2	2	ISR asc. ship length $l = 3$*	3	2
43	*ISD asc. start time**	2	2	ISR desc. trip length $\alpha = 0.1$	2	0
44	ISR desc. trip length $\alpha = 0.1$	1	0	*ISD desc. start time*	1	0
45	*ISD desc. start time*	0	0	*ISD asc. start time**	0	0

Table 5.2: Ranking of ISR and selected ISD settings. Settings marked with * have plotted performance profiles in Figure 5.2

	small instances			one day instances		
rank	setting	wins	ties	setting	wins	ties
1	MSD desc. ship length	4	2	MSD asc. start time	6	0
2	MSD desc. trip length	2	4	MSD desc. ship length	3	2
3	MSD asc. start time	2	4	MSD asc. trip length	3	2
4	MSD desc. start time	0	5	MSD desc. trip length	2	3
5	MSD asc. trip length	0	5	MSD asc. ship length	2	1
6	MSD asc. ship length	0	3	MSD desc. start time	1	0
7	ISR random	0	3	ISR random	0	0

Table 5.3: Ranking of MSD and a selected ISR setting.

The ISR settings all (but one) perform significantly better than the ISD and find approximately 25 % of small instances optimal values at an improved worst case gap of 65 to 88 % on small instances and 96 to 124 % on large instances. This improvement comes at a high cost, which is to be expected with a multi start random heuristic. Compared to the deterministic version, the ISR takes 3 to 20 times the runtime on small instances for 21.89 iterations on average. For one day instances the runtime factor increases to 14 to 140 times for 156.07 iterations on average.

5.3.3 Mutable Sequential Deterministic Construction (MSD)

As described in Subsection 4.2.4, the MSD construction heuristic is based on the ISD and uses the same orderings of trips for insertion. But it uses a sophisticated data structure to efficiently update the schedule after each insertion, which allows exact decision making which again results in much better final results. These advantages come at a price of increased runtime, which is rather comparable to the ISR than to the ISD.

For small instances, all MSD settings perform comparable equal with the *ISR random* setting, but we can already see that *asc. start time* has the best runtime, while the reverse setting *desc. start time* has the worst runtime. We conclude that this comes from the fact, that inserting ships ascending by starting time results in less lookups and updates to the schedule because the trips that are added late in the process, when the schedule is already large, appear late in the planning horizon and thus need fewer lookups and updates as their insertion affects fewer lockage operations. The opposite effect can be observed when adding ships in reverse order. Trips that are added late in the process appear early in the planning horizon and need to analyse many previously created lockages after their arrival date, and to update many lockage operations after insertion. This effect is even stronger on large instances, where *asc. start time* can not only calculate the best expected results, but is also the fastest MSD setting for all one day instances. The reverse ordering needs 42 to 200 % more computational time to finish.

The MSD settings have a worst case gap of 50 to 75 % on small instances and of 48 to 82 % on one day instances. Each of them finds about 17 to 20 % of the best known solutions to small instances. The best results for one day instances improve to 10 to 19 % off the best known solutions.

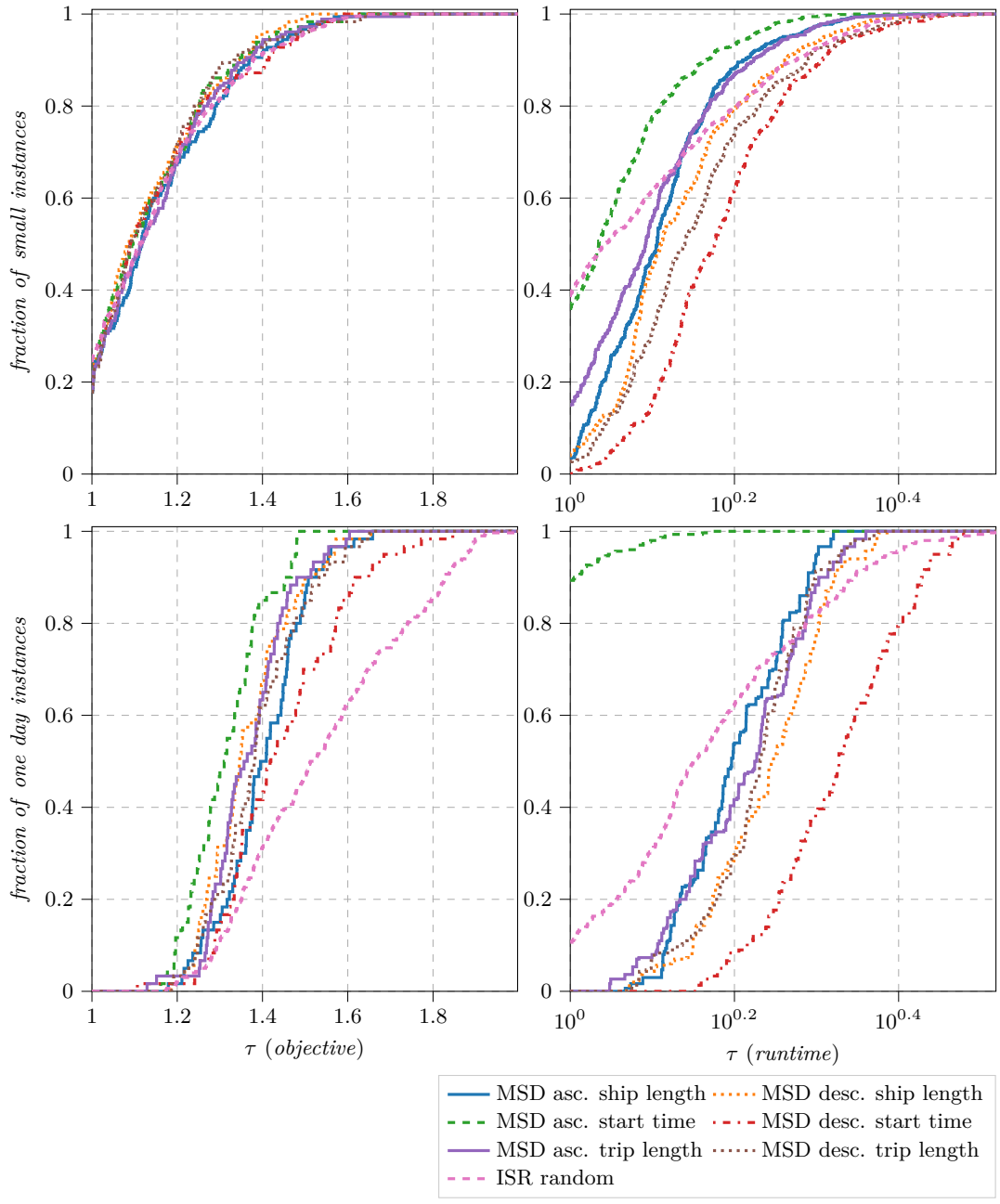


Figure 5.3: Performance profiles of MSD and a selected ISR setting.

5.3.4 Mutable Sequential Randomized Construction (MSR)

Just as the ISR is built from the ISD, the MSR is built as a randomized multi start implementation of the MSD. We used the same RCL parameters as for the ISR, i.e., cardinality-based RCL are constructed with $l \in 3, 5, 10, 20$ and percentage-based with $\alpha \in 0.10, 0.20, 0.30$. For direct comparison, we add the MSD setting *asc. start time* to the ranking in Table 5.4.

For small instances, we see similar effects like for the ISR settings. The best performing settings are the ones with the largest RCL settings and the worst performing ones have a shorter RCL. We can therefore conclude, that the MSR highly benefits from randomization on small problem instances.

Looking on the worst settings of an ordering, we can see, that *asc. start time* settings have a much better worst case performance per ordering (*MSR asc. start time* $\alpha = 0.1$ on ranks 24 and 9), than any other setting. We further see for large instances, that all but one *asc. start time* settings outperform all other settings and orderings significantly. We therefore conclude, that this ordering has a beneficial impact on the solution quality as well as a sufficient high randomization in the RCL.

Concerning the runtime performance, the *MSR asc. start time* settings benefit from the affects described above for the deterministic MSD. Even though the average number of iterations is slightly increased (155.566 compared to 154.02 over all MSR settings), the runtime is significantly shorter than of the other selected settings shown in Figure 5.4.

The MSR settings all (but one) perform significantly better than the MSD and find approximately 22 % of small instances optimal values at an improved worst case gap of 34 to 42 % on small instances and 31 to 49 % on large instances (for selected settings). Again, this improvement comes at a high cost for the multi start random heuristic. Compared to the deterministic version, the MSR takes 1.3 to 77 times the runtime on small instances for 20.19 iterations on average. For one day instances the runtime factor increases to 19 to 778 times for 154.02 iterations on average. Interestingly the average number of iterations of the MSR is almost equal to the average number of iterations of the ISR.

5.3.5 Mutable Chronological Deterministic Construction (MCD)

The MCD follows a different approach called *chronological* scheduling, where trips are not inserted into the schedule as a whole, but each ships arrival at a water gate is processed individually. These arrival events are processed chronologically, which basically resembles a FCFS policy on a per ship and water gate basis. The MCD can be parametrized by adding a lookahead time when a ship can be added to an existing lockage operation (*delayAdding*) or also when inserting a new lockage operation (*delayAll*). In both cases, the length of the lookahead (in minutes) is appended at the end of the setting name. The *simple* setting, does not use any lookahead and therefore represents a simple FCFS policy. The MCD further uses the backing data structure described in Subsection 4.2.3.

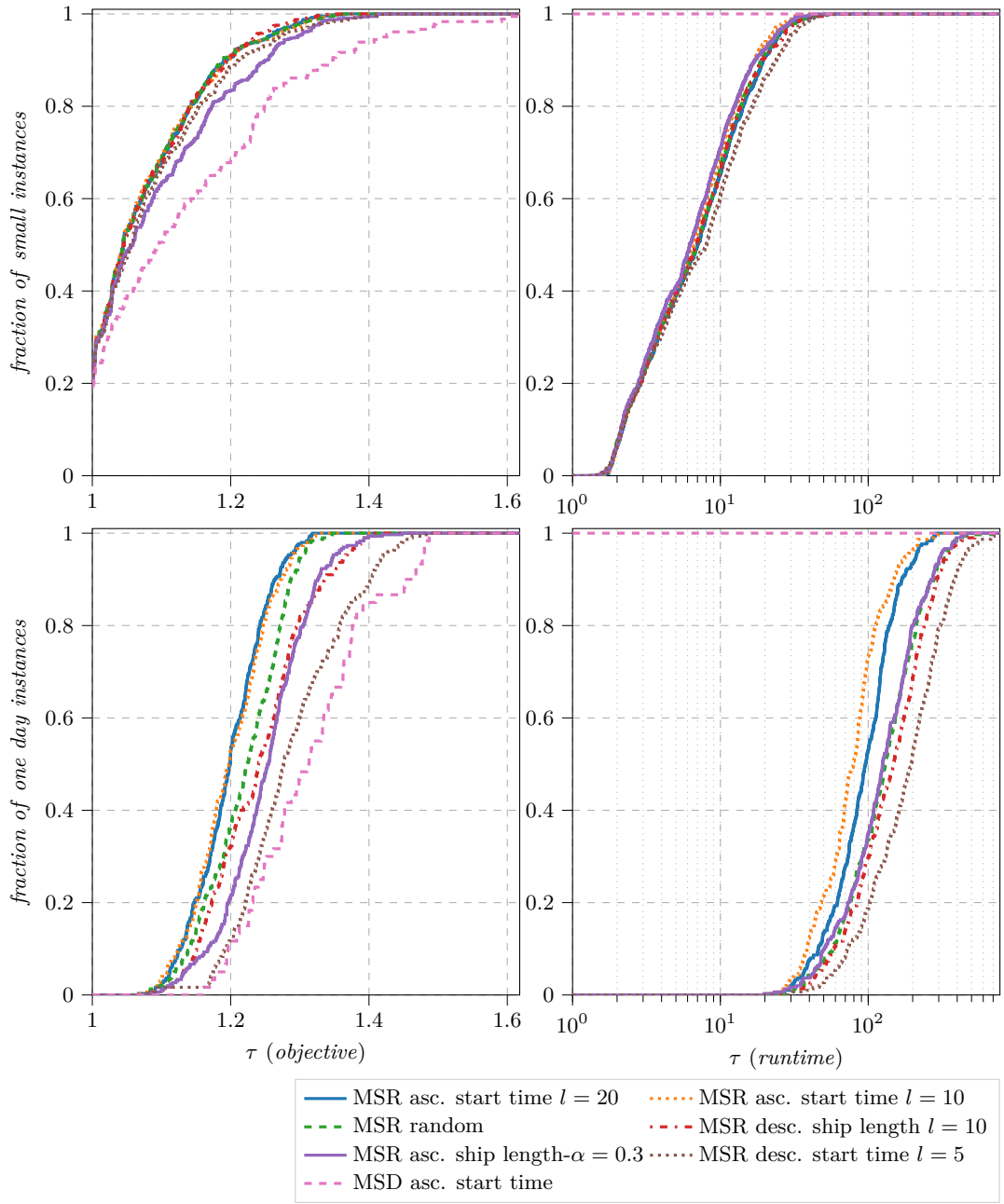


Figure 5.4: Performance profiles of selected MSR and MSD settings.

	small instances			one day instances		
rank	setting	wins	ties	setting	wins	ties
1	MSR desc. ship length $l = 10$*	37	6	MSR asc. start time $l = 20$*	41	2
2	MSR asc. start time $l = 10$*	36	7	MSR asc. start time $l = 10$*	41	2
3	MSR asc. ship length $l = 20$	35	8	MSR asc. start time $\alpha = 0.3$	41	2
4	MSR random*	34	9	MSR asc. start time $\alpha = 0.2$	39	1
5	MSR asc. start time $l = 20$*	34	9	MSR asc. start time $\alpha = 0.1$	39	1
6	MSR asc. trip length $l = 20$	33	10	MSR asc. start time $l = 5$	38	0
7	MSR desc. start time $l = 20$	32	11	MSR random*	35	2
8	MSR desc. start time $l = 10$	32	10	MSR desc. ship length $l = 20$	34	3
9	MSR desc. trip length $l = 20$	32	9	MSR asc. start time $l = 3$	34	3
10	MSR desc. trip length $l = 10$	32	8	MSR asc. trip length $l = 20$	33	3
11	MSR desc. ship length $l = 20$	32	6	MSR desc. trip length $l = 20$	32	2
12	MSR asc. ship length $l = 10$	32	5	MSR desc. ship length $\alpha = 0.3$	28	4
13	MSR desc. ship length $l = 5$	30	1	MSR asc. trip length $l = 10$	28	4
14	MSR asc. trip length $l = 10$	29	2	MSR asc. ship length $l = 20$	27	6
15	MSR asc. start time $l = 5$	29	1	MSR desc. ship length $l = 10$*	26	6
16	MSR desc. trip length $l = 5$	25	3	MSR desc. ship length $\alpha = 0.2$	25	6
17	MSR asc. trip length $l = 5$	25	3	MSR asc. trip length $\alpha = 0.3$	25	6
18	MSR asc. ship length $l = 5$	22	6	MSR desc. trip length $l = 10$	22	7
19	MSR desc. start time $l = 5$*	22	5	MSR desc. start time $l = 20$	21	5
20	MSR desc. ship length $l = 3$	21	6	MSR asc. ship length $\alpha = 0.3$*	20	7
21	MSR asc. start time $\alpha = 0.3$	21	5	MSR asc. trip length $\alpha = 0.2$	19	7
22	MSR asc. start time $\alpha = 0.2$	21	5	MSR desc. ship length $\alpha = 0.1$	18	9
23	MSR asc. start time $l = 3$	17	6	MSR desc. trip length $\alpha = 0.3$	18	7
24	MSR asc. start time $\alpha = 0.1$	16	5	MSR asc. trip length $l = 5$	16	7
25	MSR asc. ship length $\alpha = 0.3$*	15	7	MSR asc. ship length $l = 10$	15	7
26	MSR desc. trip length $l = 3$	15	6	MSR asc. ship length $\alpha = 0.2$	14	8
27	MSR asc. trip length $l = 3$	15	5	MSR desc. trip length $l = 5$	14	7
28	MSR desc. start time $l = 3$	15	4	MSR desc. ship length $l = 5$	14	7
29	MSR desc. ship length $\alpha = 0.3$	14	7	MSR asc. trip length $\alpha = 0.1$	11	8
30	MSR asc. ship length $l = 3$	13	2	MSR desc. trip length $\alpha = 0.2$	10	7
31	MSR desc. ship length $\alpha = 0.2$	13	1	MSR desc. start time $l = 10$	10	5
32	MSR desc. trip length $\alpha = 0.3$	11	1	MSR asc. ship length $\alpha = 0.1$	8	6
33	MSR desc. ship length $\alpha = 0.1$	10	2	MSR asc. trip length $l = 3$	7	8
34	MSR asc. ship length $\alpha = 0.2$	8	3	MSR desc. ship length $l = 3$	7	7
35	MSR desc. start time $\alpha = 0.3$	7	3	MSR desc. trip length $l = 3$	7	6
36	MSR desc. trip length $\alpha = 0.2$	6	3	MSR desc. trip length $\alpha = 0.1$	7	5
37	MSR desc. start time $\alpha = 0.2$	6	3	MSR asc. ship length $l = 5$	7	5
38	MSR asc. trip length $\alpha = 0.3$	5	4	MSR desc. start time $l = 5$*	5	1
39	MSR asc. ship length $\alpha = 0.1$	4	2	MSR desc. start time $\alpha = 0.3$	4	2
40	MSR desc. trip length $\alpha = 0.1$	2	3	MSR asc. ship length $l = 3$	3	2
41	MSR desc. start time $\alpha = 0.1$	2	2	MSR desc. start time $\alpha = 0.2$	2	2
42	MSR asc. trip length $\alpha = 0.2$	2	2	MSR desc. start time $l = 3$	1	1
43	MSR asc. trip length $\alpha = 0.1$	0	1	MSD asc. start time*	0	3
44	MSD asc. start time*	0	1	MSR desc. start time $\alpha = 0.1$	0	1

Table 5.4: Ranking of MSR and a selected MSD setting. Settings marked with * have plotted performance profiles in Figure 5.4.

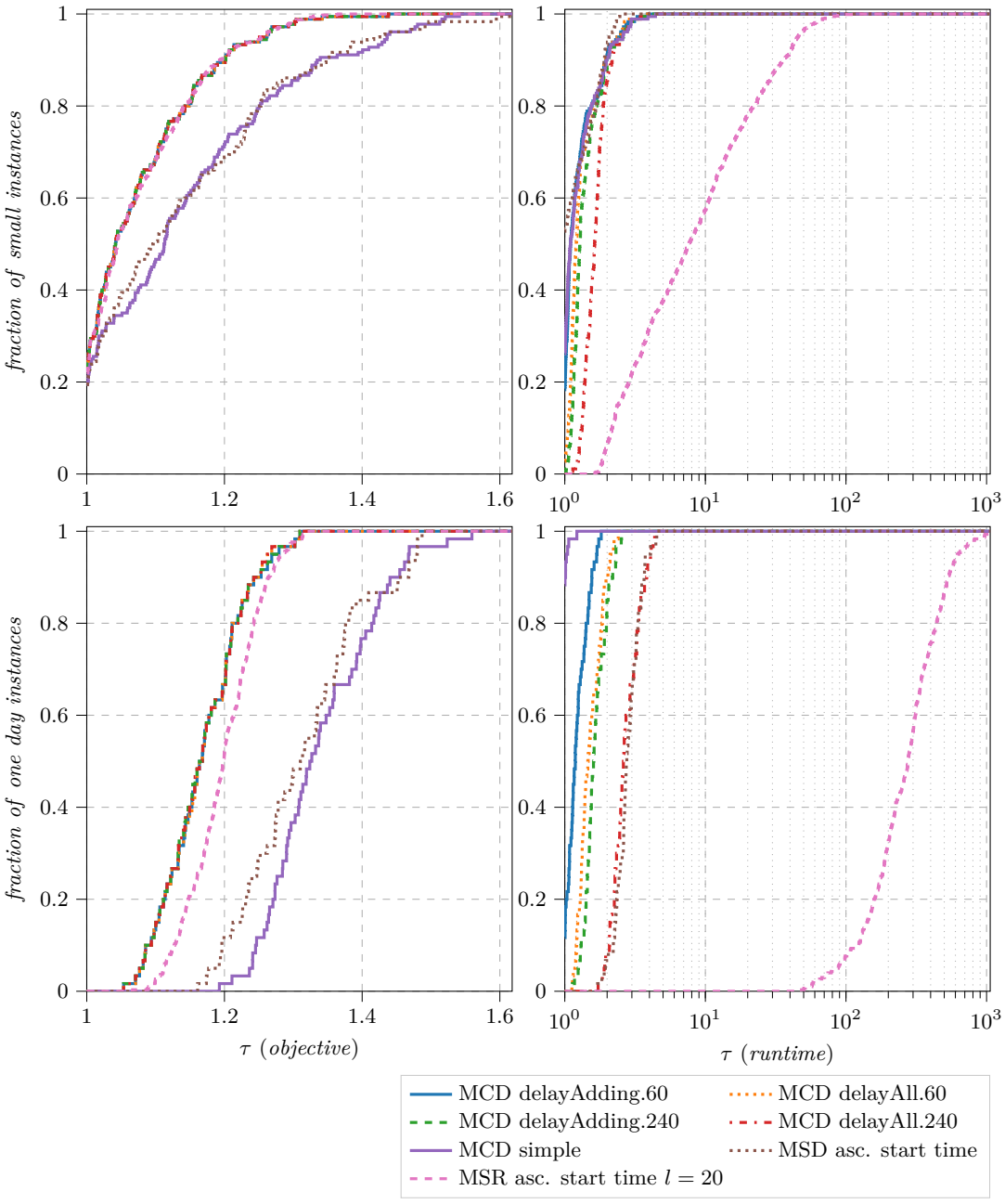


Figure 5.5: Performance profiles of selected MCD, MSD and MSR settings.

For comparability we added one MSD and MSR setting to the ranking and performance profiles.

From the ranking table Table 5.5 and the performance profiles Figure 5.5 it can be seen, that all settings with lookahead yield almost equal results on small and large instances. On small instances they are further almost equal to the *MSR asc. start time* $l = 20$ setting but are up to 112 times faster (12.81 times on average; compared to *delayAdding.60*). On large instances, the MCD settings with lookahead perform significantly better by 2.6 % on average than the MSR with a runtime improvement of up to 729 times (245.65 times on average).

Without any lookahead (the *simple* setting), the MCD performs comparable to the *MSD asc. start time* setting, both in terms of solution quality and runtime. The *MCD simple* is about 2.8 times faster on large instances.

From the runtime analysis in the performances profiles, one can clearly see that increasing the lookahead has a direct impact on the runtime as more possibilities to insert the ships passage into the schedule are evaluated. Comparing the fastest setting (*delayAll.60*) to the slowest (*delayAll.240*) shows the latter being 1.64 to 2.91 times slower. From the objective values we see that an increased lookahead has almost no impact and most instances (56 of 60) result in the exact same solutions.

5.3.6 Mutable Chronological Randomized Construction (MCR)

As described in Subsection 4.2.7, this semi-greedy construction heuristic achieves randomization by randomly picking a candidate to insert the new gate passage from the list of candidates. The candidate selection is either a *biased* decision weighted by the reciprocal of the candidate costs or based on a RCL of the best l candidates. As our other randomized construction heuristics, the MCR is a multi start heuristic that repeats until no improvement can be found for n iterations, where $n = |\mathcal{S}|$ is the number of ships in the problem instance. For direct comparison, we add the best MSD and MSR settings to the ranking in Table 5.6.

	small instances			one day instances		
rank	setting	wins	ties	setting	wins	ties
1	MSR asc. start time $l = 20$*	2	6	MCD delayAll.60*	3	5
2	MCD delayAll.60*	2	6	MCD delayAll.240*	3	5
3	MCD delayAll.240*	2	6	MCD delayAll.120	3	5
4	MCD delayAll.120	2	6	MCD delayAdding.60*	3	5
5	MCD delayAdding.60*	2	6	MCD delayAdding.240*	3	5
6	MCD delayAdding.240*	2	6	MCD delayAdding.120	3	5
7	MCD delayAdding.120	2	6	MSR asc. start time $l = 20$*	2	0
8	MSD asc. start time*	0	1	MSD asc. start time*	0	1
9	MCD simple*	0	1	MCD simple*	0	1

Table 5.5: Ranking of MCD and a selected MSD and MSR setting. Settings marked with * have plotted performance profiles in Figure 5.5

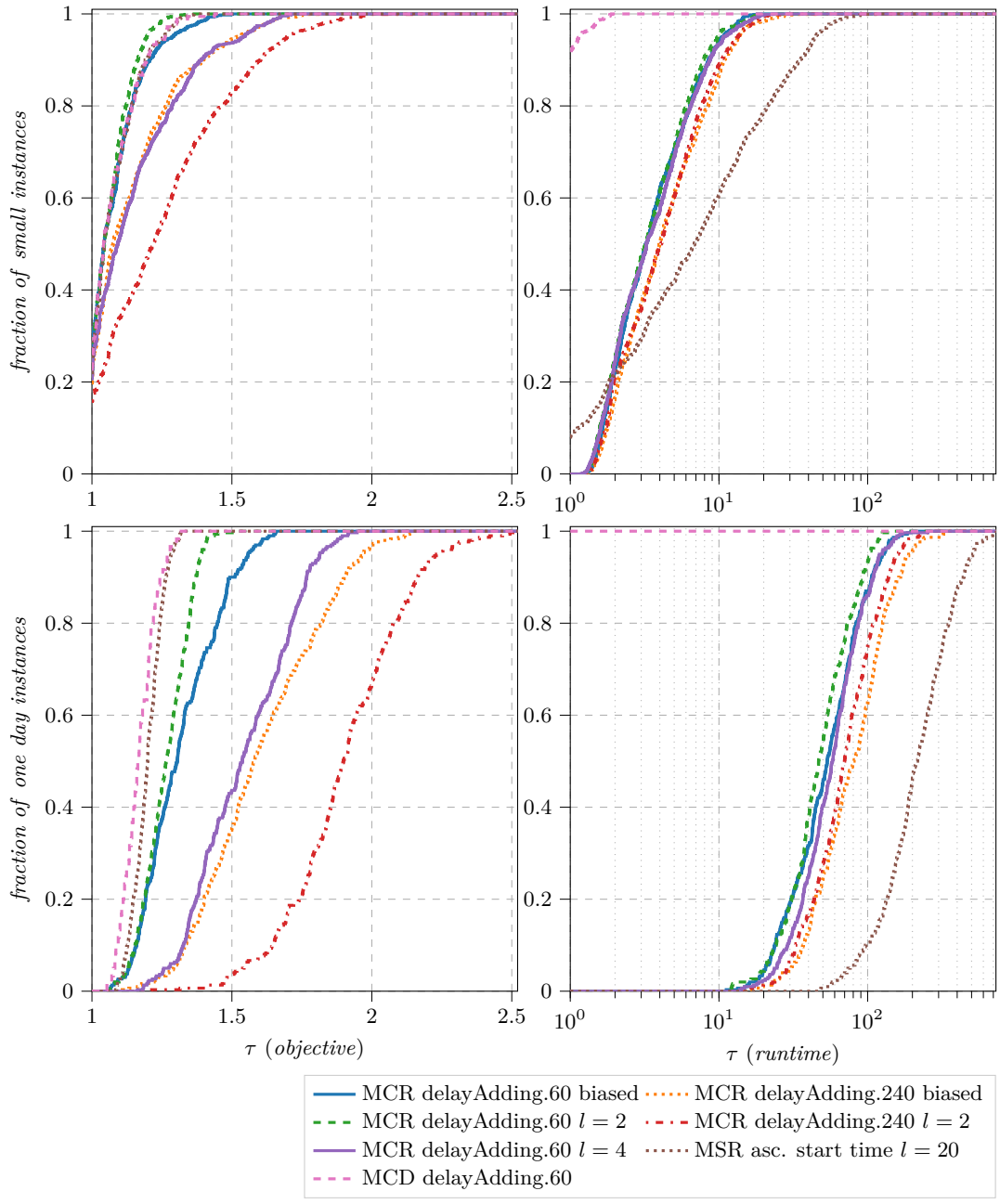


Figure 5.6: Performance profiles of selected MCR, MSD and MSR settings.

	small instances			one day instances		
rank	setting	wins	ties	setting	wins	ties
1	MCR delayAdding.60 $l = 2$*	22	0	MCD delayAdding.60*	22	0
2	MSR asc. start time $l = 20$*	20	1	MSR asc. start time $l = 20$*	21	0
3	MCD delayAdding.60*	19	2	MCR delayAdding.60 $l = 2$*	20	0
4	MCR delayAdding.60 biased*	19	1	MCR delayAdding.60 biased*	19	0
5	MCR delayAdding.120 biased	18	0	MCR simple $l = 2$	18	0
6	MCR delayAdding.120 $l = 2$	17	0	MCR simple biased	17	0
7	MCR simple $l = 2$	16	0	MCR delayAdding.120 $l = 2$	15	1
8	MCR simple biased	15	0	MCR delayAdding.120 biased	15	1
9	MCR delayAdding.240 biased*	14	0	MCR simple $l = 4$	13	1
10	MCR delayAdding.60 $l = 4$*	13	0	MCR delayAdding.60 $l = 4$*	13	1
11	MCR simple $l = 4$	11	1	MCR delayAll.60 $l = 2$	12	0
12	MCR delayAll.60 biased	11	1	MCR delayAdding.240 biased*	11	0
13	MCR delayAll.60 $l = 2$	10	0	MCR delayAll.60 biased	10	0
14	MCR delayAdding.240 $l = 2$*	9	0	MCR delayAll.120 $l = 2$	9	0
15	MCR delayAdding.120 $l = 4$	8	0	MCR delayAdding.240 $l = 2$*	8	0
16	MCR delayAll.120 biased	7	0	MCR delayAdding.120 $l = 4$	7	0
17	MCR delayAll.60 $l = 4$	6	0	MCR delayAll.240 $l = 2$	5	1
18	MCR delayAll.120 $l = 2$	5	0	MCR delayAll.120 biased	5	1
19	MCR delayAll.240 biased	4	0	MCR delayAll.60 $l = 4$	4	0
20	MCR delayAdding.240 $l = 4$	3	0	MCR delayAll.240 biased	3	0
21	MCR delayAll.240 $l = 2$	2	0	MCR delayAll.120 $l = 4$	2	0
22	MCR delayAll.120 $l = 4$	1	0	MCR delayAdding.240 $l = 4$	1	0
23	MCR delayAll.240 $l = 4$	0	0	MCR delayAll.240 $l = 4$	0	0

Table 5.6: Ranking of MCR and selected MSR and MCD settings. Settings marked with * have plotted performance profiles in Figure 5.6

Unlike the previous randomized heuristics, chronological scheduling does not benefit from randomization. For large instances, all analysed settings perform worse than any MCD setting with lookahead and also worse than the best MSR setting. Further it can be seen, that with increasing RCL length, results get worse. The worst setting *MCR delayAll.240* $l = 4$ has a worst case relative objective of about 523 % of the best known solution, which makes it the worst of all construction heuristics. All MCR settings ranked place 17 and below for any instance set in Table 5.6 are worse than any other construction heuristic of any type (see Table 5.7).

From this we conclude, that the deterministic lowest cost insertion of the MCD heuristic is much more reliable than any semi-greedy approach. In some rare cases, not choosing the lowest cost insertion can be beneficial, as can be seen for small instances. On the long run, especially for larger instances, taking the lowest costs is the better choice. As instances size and lookahead time increase, the number of possible candidates for insertion increases as well, so the setting *delayAll.240* has more candidates to insert while *delayAdding.60* has the least. With only a few candidates, a setting is likely to not having enough candidates to fill the RCL to its maximum length, so the RCL is in fact likely even shorter than stated in the settings parameters. This can be seen when looking at the ranking of settings with the same RCL length. The solution quality worsens as

the lookahead and thus the number of candidates increases.

From the runtime profiles in Figure 5.6 we further can see, that the RCL has no significant influence on the runtime. Only the lookahead parameter determines the number of candidates to be evaluated which influences the runtime. Interestingly, for this being a multi start randomized heuristic, the numbers of iterations are almost constant across all settings, despite the difference in solution quality with a mean of 154.91 average iterations (min 150.28, max 160.37 average iterations per setting) on large instances.

5.3.7 Comparison of Construction Heuristics

Looking at the overall ranking at Table 5.7 and Figure 5.7, we can clearly see the best performing algorithms and settings, especially when further taking their runtime into account. From the deterministic settings, the MCD algorithm with lookahead is by far the best choice with reliable results on small and large instances.

When randomization is needed, e.g., for creating a population of diverse initial solutions for population based or multi start metaheuristics, the answer is not so clear. MSR and MCR are highly competitive in solution quality on smaller instances, but for larger instances, MSR performs significantly better than the MCR. Depending on the instance size, *MCR delayAdding.60* $l = 2$ is about 2 to 5 times faster than *MSR asc. start time* $l = 20$ (up to 13.74 times). The high runtime of the randomized construction heuristics could be countered by decreasing the stopping criterion for the multi start heuristics, possibly down to only a single iteration. This would weaken the solution quality but reduce the runtime to approximately the deterministic versions of the algorithms.

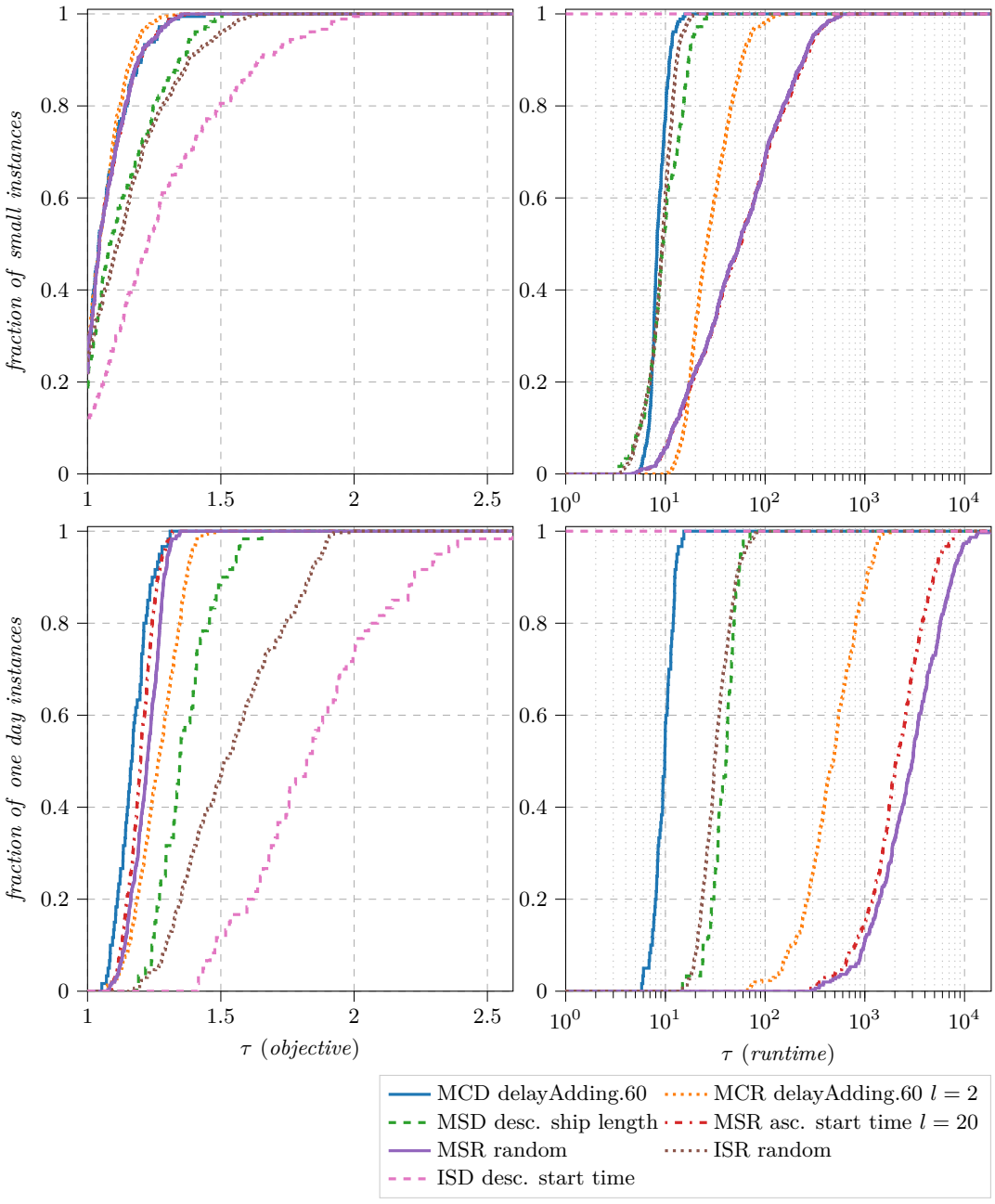


Figure 5.7: Performance profiles of selected construction heuristic settings.

	small instances			one day instances		
rank	setting	wins	ties	setting	wins	ties
1	MCR delayAdding.60 $l = 2$*	126	0	MCD delayAdding.60*	121	5
2	MSR desc. ship length $l = 10$	113	12	MCD delayAdding.240	121	5
3	MSR asc. start time $l = 10$	112	13	MCD delayAdding.120	121	5
4	MSR asc. ship length $l = 20$	111	14	MCD delayAll.60	121	5
5	MSR random*	110	15	MCD delayAll.240	121	5
6	MSR asc. start time $l = 20$*	110	15	MCD delayAll.120	121	5
7	MSR asc. trip length $l = 20$	109	16	MSR asc. start time $l = 20$*	118	2
8	MSR desc. start time $l = 20$	108	17	MSR asc. start time $l = 10$	118	2
9	MSR desc. start time $l = 10$	108	16	MSR asc. start time $\alpha = 0.3$	118	2
10	MSR desc. trip length $l = 20$	108	15	MSR asc. start time $\alpha = 0.2$	116	1
11	MSR desc. trip length $l = 10$	108	14	MSR asc. start time $\alpha = 0.1$	116	1
12	MSR desc. ship length $l = 20$	108	12	MSR asc. start time $l = 5$	115	0
13	MSR asc. ship length $l = 10$	108	11	MSR random*	112	2
14	MSR desc. ship length $l = 5$	106	1	MSR desc. ship length $l = 20$	111	3
15	MCD delayAdding.60*	105	20	MSR asc. start time $l = 3$	111	3
...						
36	MSR desc. start time $l = 3$	90	4	MCR delayAdding.60 $l = 2$*	87	7
...						
55	MSR asc. trip length $\alpha = 0.1$	68	5	MSD desc. ship length*	70	3
56	MSD desc. ship length*	67	6	MSD asc. trip length	70	3
...						
66	ISR random*	52	12	ISR random*	61	2
...						
113	ISR desc. trip length $\alpha = 0.1$	12	2	ISD desc. start time*	11	4
114	MCR delayAdding.120 $l = 4$	11	3	ISD desc. ship length	11	4
115	ISD asc. trip length	9	6	ISD asc. trip length	11	4
116	ISD asc. ship length	9	4	MCR delayAdding.240 $l = 2$	10	4
117	ISD desc. start time*	8	4	ISD asc. ship length	9	2
118	ISD desc. ship length	8	4	MCR delayAdding.120 $l = 4$	8	2
119	MCR delayAll.120 biased	8	2	ISD desc. trip length	7	2
120	ISD desc. trip length	7	0	ISD asc. start time	7	1
121	MCR delayAll.60 $l = 4$	6	0	MCR delayAll.240 $l = 2$	5	1
122	MCR delayAll.120 $l = 2$	5	0	MCR delayAll.120 biased	5	1
123	MCR delayAll.240 biased	4	0	MCR delayAll.60 $l = 4$	4	0
124	MCR delayAdding.240 $l = 4$	3	0	MCR delayAll.240 biased	3	0
125	MCR delayAll.240 $l = 2$	2	0	MCR delayAll.120 $l = 4$	2	0
126	MCR delayAll.120 $l = 4$	1	0	MCR delayAdding.240 $l = 4$	1	0
127	MCR delayAll.240 $l = 4$	0	0	MCR delayAll.240 $l = 4$	0	0

Table 5.7: Ranking of all construction settings. Settings marked with * have plotted performance profiles in Figure 5.7

5.4 Metaheuristics

Local search based metaheuristics require an initial solution to start from. From the various construction heuristics we evaluated in the section above, we chose the following settings to provide initial solutions for our improvement heuristics:

- deterministic construction heuristics:
  - *MCD delayAdding.60*
  - *MSD desc. ship length* (omitted for randomized metaheuristics)
- randomized construction heuristics:
  - *MCR delayAdding.60* $l = 2$
  - *MSR asc. start time* $l = 20$
  - *MSR random*

The randomized construction heuristics provided five initial solutions with different random seeds, so each metaheuristic using those initial solutions was again run five times.

We used our defined neighborhood structures from Section 4.3 or combinations thereof in different metaheuristics and evaluated them.

5.4.1 Local Search with Shift Ship (ShS)

We first analysed a simple neighborhood, Shift Ship (ShS) using hill climbing with BI step function. The ShS is a ranged neighborhood and in this experiment we were mainly interested on the impact of different ranges on the solution quality and runtime.

As can be seen in Table 5.8, the local search with ShS is not able to significantly improve the overall result. The different settings are mostly ranked according to their initial solution quality. Looking at the average improvements in Table 5.9, we still see that weak solutions benefit more from the ShS than good ones.

Increasing the ranges always improves the average relative improvements found by the local search, but especially for small instances and settings that have already relatively good initial solutions, the improvement becomes insignificant. Further analysing the individual settings' solutions shows, that hill climbing with the ShS neighborhood suffers from premature local optima. Increasing the range and thus the search space sometimes lead to early gains that yield worse local optima that are not escaped. Even though these early local optima exist for larger ranges, the average relative improvement never worsens.

As Figure 5.8 shows, the initial solution quality and the increase in range affects the runtime. Starting from better initial solutions yields shorter runtimes as the algorithm needs less iterations to find a (local) optimum. On the other hand, increasing the search space by increasing the range naturally also increases the runtime.

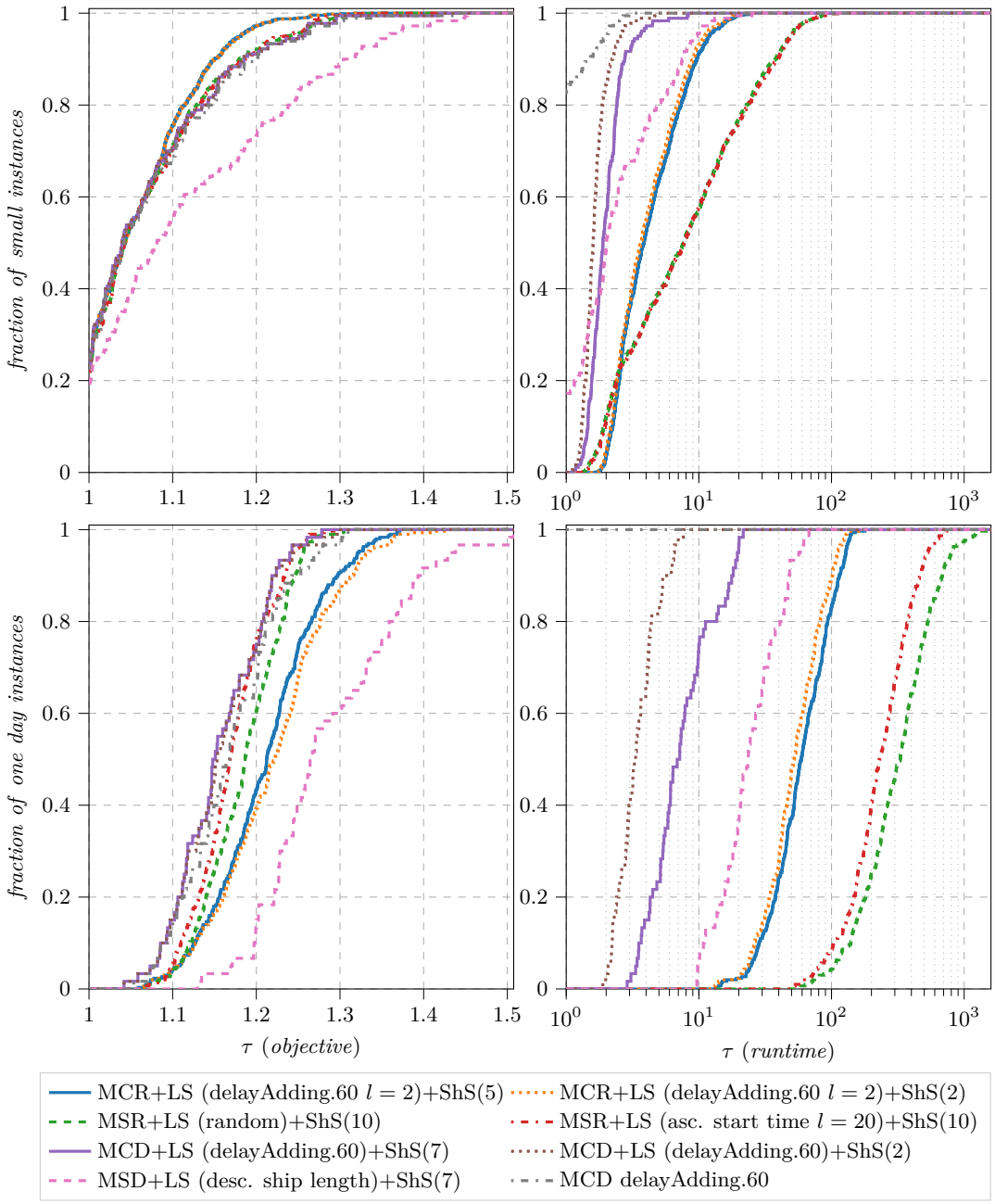


Figure 5.8: Performance profiles of selected LS:ShS settings and *MCD delayAdding.60* as a baseline.

	small instances		
rank	setting	wins	ties
1	MCR+LS (delayAdding.60 $l = 2$)+ShS(7)	17	2
2	MCR+LS (delayAdding.60 $l = 2$)+ShS(5)	17	2
3	MCR+LS (delayAdding.60 $l = 2$)+ShS(10)	17	2
4	MCR+LS (delayAdding.60 $l = 2$)+ShS(2)	16	0
5	MSR+LS (random)+ShS(10)	9	6
6	MSR+LS (random)+ShS(7)	7	7
7	MSR+LS (asc. start time $l = 20$)+ShS(10)	6	9
8	MSR+LS (random)+ShS(5)	6	7
9	MSR+LS (asc. start time $l = 20$)+ShS(7)	5	10
10	MSR+LS (asc. start time $l = 20$)+ShS(5)	5	8
11	MSR+LS (random)+ShS(2)	5	7
12	MCD+LS (delayAdding.60)+ShS(7)	4	11
13	MCD+LS (delayAdding.60)+ShS(5)	4	11
14	MCD+LS (delayAdding.60)+ShS(2)	4	11
15	MCD+LS (delayAdding.60)+ShS(10)	4	11
16	MSR+LS (asc. start time $l = 20$)+ShS(2)	4	4
17	MSD+LS (desc. ship length)+ShS(7)	2	1
18	MSD+LS (desc. ship length)+ShS(10)	2	1
19	MSD+LS (desc. ship length)+ShS(5)	1	0
20	MSD+LS (desc. ship length)+ShS(2)	0	0
	one day instances		
rank	setting	wins	ties
1	MCD+LS (delayAdding.60)+ShS(7)	17	2
2	MCD+LS (delayAdding.60)+ShS(5)	17	2
3	MCD+LS (delayAdding.60)+ShS(10)	17	2
4	MCD+LS (delayAdding.60)+ShS(2)	16	0
5	MSR+LS (asc. start time $l = 20$)+ShS(10)	15	0
6	MSR+LS (asc. start time $l = 20$)+ShS(7)	14	0
7	MSR+LS (asc. start time $l = 20$)+ShS(5)	13	0
8	MSR+LS (asc. start time $l = 20$)+ShS(2)	12	0
9	MSR+LS (random)+ShS(10)	11	0
10	MSR+LS (random)+ShS(7)	10	0
11	MSR+LS (random)+ShS(5)	9	0
12	MSR+LS (random)+ShS(2)	8	0
13	MCR+LS (delayAdding.60 $l = 2$)+ShS(10)	7	0
14	MCR+LS (delayAdding.60 $l = 2$)+ShS(7)	6	0
15	MCR+LS (delayAdding.60 $l = 2$)+ShS(5)	5	0
16	MCR+LS (delayAdding.60 $l = 2$)+ShS(2)	4	0
17	MSD+LS (desc. ship length)+ShS(10)	2	1
18	MSD+LS (desc. ship length)+ShS(7)	1	2
19	MSD+LS (desc. ship length)+ShS(5)	1	1
20	MSD+LS (desc. ship length)+ShS(2)	0	0

Table 5.8: Ranking of LS:ShS settings.

Construction Heuristic	Range 2	Range 5	Range 7	Range 10
	small instances			
MCR delayAdding.60 $l = 2$	0.3251 %	0.3372 %	0.3372 %	0.3372 %
MSR random	0.3846 %	0.4187 %	0.4220 %	0.4259 %
MSR asc. start time $l = 20$	0.3600 %	0.3949 %	0.3968 %	0.3997 %
MCD delayAdding.60	0.3511 %	0.3517 %	0.3517 %	0.3517 %
MSD desc. ship length	0.7301 %	0.8288 %	0.8999 %	0.8999 %
	one day instances			
MCD delayAdding.60	0.9040 %	1.0287 %	1.0469 %	1.0628 %
MSR asc. start time $l = 20$	1.8659 %	2.1913 %	2.2389 %	2.2616 %
MSR random	2.2754 %	2.6162 %	2.7137 %	2.7577 %
MCR delayAdding.60 $l = 2$	3.4637 %	4.1012 %	4.1537 %	4.2026 %
MSD desc. ship length	4.6134 %	5.4673 %	5.5677 %	5.7347 %

Table 5.9: Relative improvement found by LS:ShS settings by range and construction heuristic. Per set, results are sorted by the construction heuristics rank.

5.4.2 Local Search with Improve Trip (IT)

We used the Improve Trip (IT) neighborhood to build a basic large neighborhood search. As described in Subsection 4.3.11, it is a basic ruin and recreate algorithm that reuses the MCD construction heuristic to repair the incumbent solution after removal of a ship. It therefore has the same parameters as the MCD. In our experiments, we saw that the basic insertion type *simple*, *delayAdding*, *delayAll* did not affect the solution quality that much, so we conducted our experiments with the *delayAll.60* setting. For our evaluations we were mainly interested on the remaining parameters: the waiting threshold t and the step function.

The IT neighborhood explorer iterates over the list of ships and removes each ship from the schedule and tries to re-add it as if the MCD construction heuristic was used. The used step function controls this process as it affects the order in which the ships are iterated.

- Next improvement (NI) sorts the ships by their total waiting time, so ships with longer waiting time are processed first. If re-adding a ship improves the objective, the move is applied and the resulting schedule is used as the new incumbent solution.
- Random improvement (RI) shuffles the list of ships randomly and also takes the first found improved schedule for the next iteration.
- Best improvement (BI) iterates over all ships and finally applies the move with the best objective. The order of the ships does not affect the result in this case, except for the rare case when two moves result in the exact same objective value. This can be neglected, because we use the weighted waiting time as defined in Subsection 4.3.1 to compare the moves improvements.

As can be seen in tables 5.10 and 5.11, BI always performs better than the other step functions, but not always significantly. The drawback of BI is its high runtime, which for

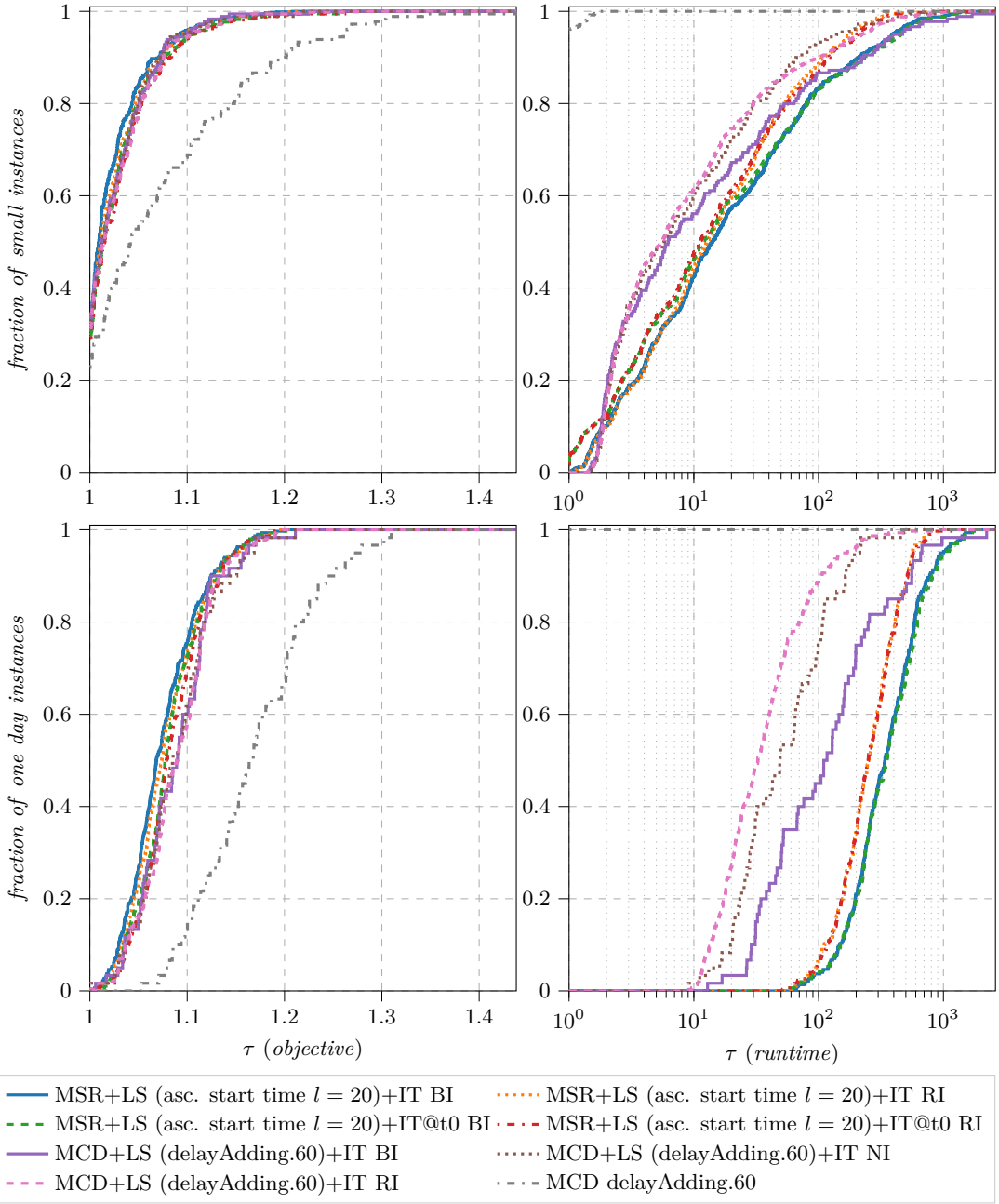


Figure 5.9: Performance profiles of selected LS:IT settings and *MCD delayAdding.60* as a baseline.

	small instances		
rank	setting	wins	ties
1	MSR+LS (random)+IT BI	23	0
2	MSR+LS (asc. start time $l = 20$)+IT BI	22	0
3	MSR+LS (asc. start time $l = 20$)+IT RI	17	4
4	MSR+LS (random)+IT RI	15	6
5	MSR+LS (random)+IT@t0 BI	15	6
6	MCR+LS (delayAdding.60 $l = 2$)+IT BI	14	7
7	MCD+LS (delayAdding.60)+IT NI	14	7
8	MSR+LS (asc. start time $l = 20$)+IT@t0 BI	14	5
9	MCD+LS (delayAdding.60)+IT BI	12	8
10	MCR+LS (delayAdding.60 $l = 2$)+IT RI	11	6
11	MCD+LS (delayAdding.60)+IT RI	9	6
12	MCR+LS (delayAdding.60 $l = 2$)+IT@t0 BI	9	4
13	MSR+LS (asc. start time $l = 20$)+IT@t0 RI	8	7
14	MSR+LS (random)+IT@t0 RI	7	7
15	MCR+LS (delayAdding.60 $l = 2$)+IT@t0 RI	6	5
16	MCD+LS (delayAdding.60)+IT@t0 BI	4	7
17	MCD+LS (delayAdding.60)+IT@t0 NI	4	5
18	MSD+LS (desc. ship length)+IT BI	3	11
19	MSD+LS (desc. ship length)+IT NI	2	6
20	MCD+LS (delayAdding.60)+IT@t0 RI	2	4
21	MSD+LS (desc. ship length)+IT@t0 BI	1	6
22	MCD+LS (delayAdding.60)+IT@t600 BI	1	5
23	MSD+LS (desc. ship length)+IT@t0 NI	0	3
24	MSD+LS (desc. ship length)+IT@t600 BI	0	1
	one day instances		
rank	setting	wins	ties
1	MSR+LS (asc. start time $l = 20$)+IT BI	22	1
2	MSR+LS (random)+IT BI	21	2
3	MSR+LS (asc. start time $l = 20$)+IT RI	21	1
4	MSR+LS (random)+IT RI	18	2
5	MSR+LS (asc. start time $l = 20$)+IT@t0 BI	18	2
6	MSR+LS (random)+IT@t0 BI	17	3
7	MSR+LS (asc. start time $l = 20$)+IT@t0 RI	15	3
8	MSR+LS (random)+IT@t0 RI	12	5
9	MCR+LS (delayAdding.60 $l = 2$)+IT BI	12	4
10	MCD+LS (delayAdding.60)+IT BI	11	6
11	MCD+LS (delayAdding.60)+IT RI	11	5
12	MCD+LS (delayAdding.60)+IT NI	10	6
13	MCR+LS (delayAdding.60 $l = 2$)+IT RI	9	5
14	MCD+LS (delayAdding.60)+IT@t0 BI	8	4
15	MCD+LS (delayAdding.60)+IT@t0 RI	6	3
16	MCD+LS (delayAdding.60)+IT@t0 NI	5	6
17	MCR+LS (delayAdding.60 $l = 2$)+IT@t0 BI	5	5
18	MCR+LS (delayAdding.60 $l = 2$)+IT@t0 RI	5	4
19	MCD+LS (delayAdding.60)+IT@t600 BI	5	3
20	MSD+LS (desc. ship length)+IT BI	3	1
21	MSD+LS (desc. ship length)+IT@t0 BI	1	2
22	MSD+LS (desc. ship length)+IT NI	0	4
23	MSD+LS (desc. ship length)+IT@t0 NI	0	3
24	MSD+LS (desc. ship length)+IT@t600 BI	0	2

Table 5.10: Ranking of LS:IT settings.

Construction Heuristic	Step BI	Step NI	Step RI
	small instances		
MCR delayAdding.60 $l = 2$	3.3534 %	N/A	3.2583 %
MSR random	4.4585 %	N/A	4.1932 %
MSR asc. start time $l = 20$	4.4269 %	N/A	4.2260 %
MCD delayAdding.60	4.0509 %	4.0226 %	3.8773 %
MSD desc. ship length	8.2169 %	7.9862 %	N/A
	one day instances		
MCD delayAdding.60	6.7522 %	6.6813 %	6.6552 %
MSR asc. start time $l = 20$	10.1948 %	N/A	9.8927 %
MSR random	11.4775 %	N/A	11.1558 %
MCR delayAdding.60 $l = 2$	13.7329 %	N/A	13.4880 %
MSD desc. ship length	17.7965 %	17.5620 %	N/A

Table 5.11: Relative improvement found by LS:IT settings without threshold by step function (BI, NI, RI) and construction heuristic. Per set, results are sorted by the construction heuristics rank.

larger instances is on average about 3.5 to 5.5 times the runtime of the RI step function (1.5 to 1.7 for small instances).

The threshold t is used to narrow the search space. A ship is only considered for removal and re-adding if it incurs a waiting time above t at any gate. A negative threshold means all ships are evaluated. Values above 0 are mainly meaningful to speed up the local search in a multi neighborhood metaheuristic, where the IT with $t > 0$ is used to improve trips that are hard to tackle by other means. As can be seen in tables 5.10 and 5.11, dropping the threshold always improves the objective significantly. We see the cause for this improvement in the fact that even ships without any waiting still give room for improving the schedule if adding waiting time for that ship can for example reduce the number of lockages. By setting a threshold of $t = 0$, such ships are not selected for removal and re-addition. Interestingly, removing the threshold does hardly affect the runtime, especially for larger instances. For small instances, it increases the runtime by approximately 25 %, while for larger instances it decreases by about 12 %. We conclude this is caused by the fact that large instances hardly contain ships that can be scheduled without any waiting time, so the larger and more complex the instance is, the less ships are excluded from re-addition by the threshold $t = 0$.

Table 5.11 shows, that, compared to the local search with ShS, the IT neighborhood allows far more improvement especially for randomized and weaker initial solutions. We assume the initial solutions by the MCD construction heuristic do not benefit that much from this local search for several reasons: They are already constructed using the same paradigm, as the IT neighborhood is just a repeatedly re-addition using the same algorithm. Further, the deterministic initial solutions lack the diversity that allows randomized construction heuristics to overcome local optima. Third and last, especially for large instances, the initial solutions are already the best ones, so there is less room for improvement. Still, it is notable that with this local search, settings with MSR as the construction heuristic perform significantly better than other settings.

As can be seen in Figure 5.9 the local search with the IT neighborhood eventually produces very good results even for large instances with a worst case gap of about 20 % to the best known solutions for all instances. The objective values of different settings are very close to each other, so if runtime is a crucial factor, faster settings can be chosen. For example the setting *MCD+LS (delayAdding.60)+IT RI* compared to *MSR+LS (asc. start time $l = 20$)+IT BI* is on average 3.8 times faster with only 1.3 % worse objective values. The runtime could be even improved further using a deterministic approach (dropping the overhead for multi start) using *MCD+LS (delayAdding.60)+IT NI*, which performs even slightly better than the RI approach but is only 2.5 times faster than *MSR+LS (asc. start time $l = 20$)+IT BI* on a single run.

5.4.3 Simple Token-Ring Neighborhood Search (TRNS)

We combined several simple neighborhoods to create a Token-Ring Neighborhood Search (TRNS), our first VNS:

- Rearrange Ships in Lockage (ReSiL)
- Swap Lockages (SwL)
- Shift Lockage (ShL)
- Swap Ships (SwS)
- Shift Ship (ShS)

All these neighborhoods only manipulate the schedule per water gate – in contrast to the above evaluated IT neighborhood. When shifting or swapping whole lockage operations, new empty lockages might be added to maintain feasibility of the schedule. Further, shifting single ships to other lockages might leave the original lockage operation empty. For those reasons, we also added the Remove Empty Lockages (RemEmpty) neighborhood to the VNS to clean the schedule from unnecessary empty lockages. We implemented the TRNS (with local optimum, denoted TR*) by using our previously introduced NSCNI framework, using *SAME* as the *success mode* of the neighborhood selector. (See Subsection 4.4.4 for details.)

All VNS settings start with RemEmpty and ReSiL neighborhoods. Then the ranged neighborhood structures to swap or shift lockage operations or single ships are combined, depending on the type. We evaluated three ways to combine them to see how the design of the neighborhood structures and their interaction affects the overall solution quality and computational time:

- Neighborhood loop with iterative range increment (*LIRI*): All ranges from 1 to 10 are iterated. For each range i, the four neighborhoods are visited in a cycle with a range of $r^{min} = r^{max} = i$.

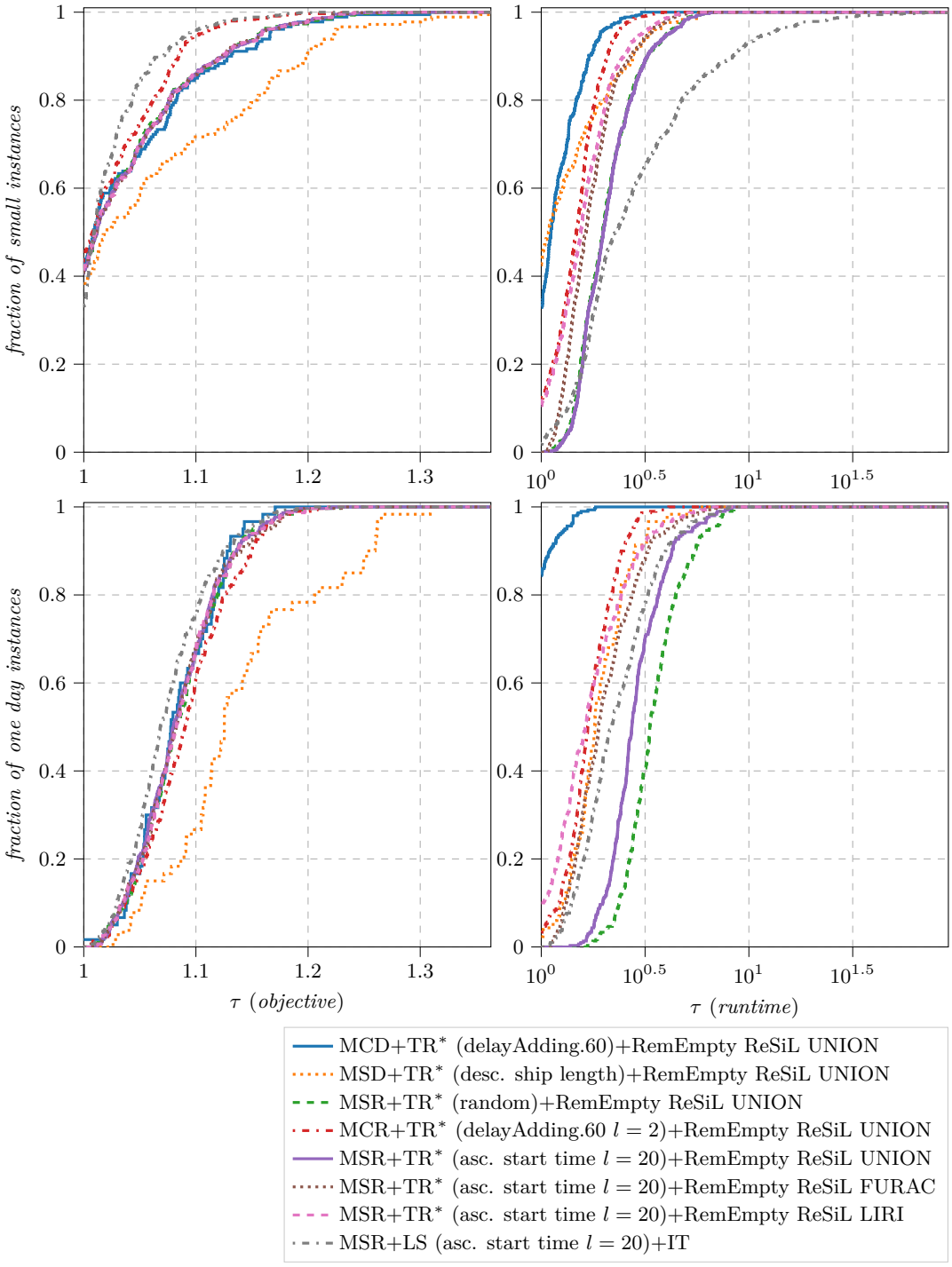


Figure 5.10: Performance profiles of combinations of SwS, ShS, SwL and ShL neighborhoods in a TRNS, compared with *MSR+LS (asc. start time* $l = 20$*)+IT* as a baseline. All settings use BI step function.

	small instances		
rank	setting	wins	ties
1	MCR+TR* (delayAdding.60 $l = 2$)+RemEmpty ReSiL UNION	13	1
2	MCR+TR* (delayAdding.60 $l = 2$)+RemEmpty ReSiL FURAC	12	2
3	MCR+TR* (delayAdding.60 $l = 2$)+LIRI	12	1
4	MSR+TR* (random)+LIRI	5	6
5	MSR+TR* (asc. start time $l = 20$)+LIRI	3	8
6	MSR+TR* (random)+RemEmpty ReSiL UNION	3	8
7	MSR+TR* (asc. start time $l = 20$)+RemEmpty ReSiL UNION	3	8
8	MSR+TR* (asc. start time $l = 20$)+RemEmpty ReSiL FURAC	3	8
9	MCD+TR* (delayAdding.60)+LIRI	3	8
10	MCD+TR* (delayAdding.60)+RemEmpty ReSiL UNION	3	8
11	MSR+TR* (random)+RemEmpty ReSiL FURAC	3	7
12	MCD+TR* (delayAdding.60)+RemEmpty ReSiL FURAC	3	7
13	MSD+TR* (desc. ship length)+LIRI	0	2
14	MSD+TR* (desc. ship length)+RemEmpty ReSiL UNION	0	2
15	MSD+TR* (desc. ship length)+RemEmpty ReSiL FURAC	0	2
	one day instances		
rank	setting	wins	ties
1	MCD+TR* (delayAdding.60)+RemEmpty ReSiL UNION	9	5
2	MSR+TR* (random)+RemEmpty ReSiL UNION	8	6
3	MSR+TR* (asc. start time $l = 20$)+RemEmpty ReSiL UNION	8	6
4	MSR+TR* (asc. start time $l = 20$)+LIRI	6	8
5	MSR+TR* (asc. start time $l = 20$)+RemEmpty ReSiL FURAC	6	8
6	MCD+TR* (delayAdding.60)+RemEmpty ReSiL FURAC	6	8
7	MCD+TR* (delayAdding.60)+LIRI	6	7
8	MSR+TR* (random)+LIRI	6	5
9	MSR+TR* (random)+RemEmpty ReSiL FURAC	6	5
10	MCR+TR* (delayAdding.60 $l = 2$)+RemEmpty ReSiL UNION	5	0
11	MCR+TR* (delayAdding.60 $l = 2$)+LIRI	3	1
12	MCR+TR* (delayAdding.60 $l = 2$)+RemEmpty ReSiL FURAC	3	1
13	MSD+TR* (desc. ship length)+LIRI	0	2
14	MSD+TR* (desc. ship length)+RemEmpty ReSiL UNION	0	2
15	MSD+TR* (desc. ship length)+RemEmpty ReSiL FURAC	0	2

Table 5.12: Ranking of TRNS settings with combinations of SwS, ShS, SwL and ShL neighborhoods. All settings use BI step function.

Construction Heuristic	LIRI	FURAC	UNION
	small instances		
MCR delayAdding.60 $l = 2$	3.1895 %	3.1972 %	3.2042 %
MSR random	3.1276 %	3.1109 %	3.1244 %
MSR asc. start time $l = 20$	3.1195 %	3.1022 %	3.1148 %
MCD delayAdding.60	2.8925 %	2.8717 %	2.8931 %
MSD desc. ship length	5.3855 %	5.4543 %	5.3503 %
	one day instances		
MCD delayAdding.60	6.9906 %	7.1314 %	7.2068 %
MSR asc. start time $l = 20$	9.3429 %	9.3456 %	9.4335 %
MSR random	10.7459 %	10.7365 %	10.8677 %
MCR delayAdding.60 $l = 2$	13.2029 %	13.2604 %	13.4597 %
MSD desc. ship length	16.0130 %	15.8835 %	16.3210 %

Table 5.13: Relative improvement found by TRNS settings by combination type and construction heuristic. Per set, results are sorted by the construction heuristics rank.

- Full range neighborhood chain (*FURAC*): Visit each neighborhood, each with a range of $r^{min} = 1$ and $r^{max} = 10$.
- Neighborhood union (*UNION*): This creates a single large neighborhood of all four neighborhoods with a range of $r^{min} = 1$ and $r^{max} = 10$.

As Table 5.13 shows, the methods perform quite equal, but *UNION* in general creates slightly better solutions than the others. In table 5.12 we see that *UNION* only once performs significantly better than both competitors (for *MCR delayAdding.60* $l = 2$ on large instances). Runtime analysis show that on average, *LIRI* performs fastest with *FURAC* taking 1.26 times longer on small instances and 2.19 times longer on large instances. *UNION* takes the longest time with an average of 2.05 (small instances) or 5.47 (large instances) the time of *LIRI*.

5.4.4 Enhanced Variable Neighborhood Descent (VND)

Our second VNS implementation was a VND. It was based on the TRNS algorithm with local optimum described in Subsection 5.4.3, but used *FIRST* as *success mode* of the neighborhood selector.

By using our NSCNI framework, we were able to optimize the VND slightly (and therefore call it an *enhanced* VND). Whenever a VND changes the neighborhood structure from N_k to the next one N_{k+1}, the current incumbent solution is locally optimal w.r.t. N_k. For the neighborhood Remove Empty Lockages (RemEmpty) (which is the first neighborhood in our VND), a locally optimal solution is any solution that does not contain an empty lockage operation that can be removed from the schedule without breaking the schedules feasibility. The second neighborhood following is Rearrange Ships in Lockage (ReSiL), which only changes the order of ships inside the lockage operations. It does therefore not change the number of assigned ships and does further not alter the lockage chain at a water gate. A standard VND implementation would return to the first neighborhood

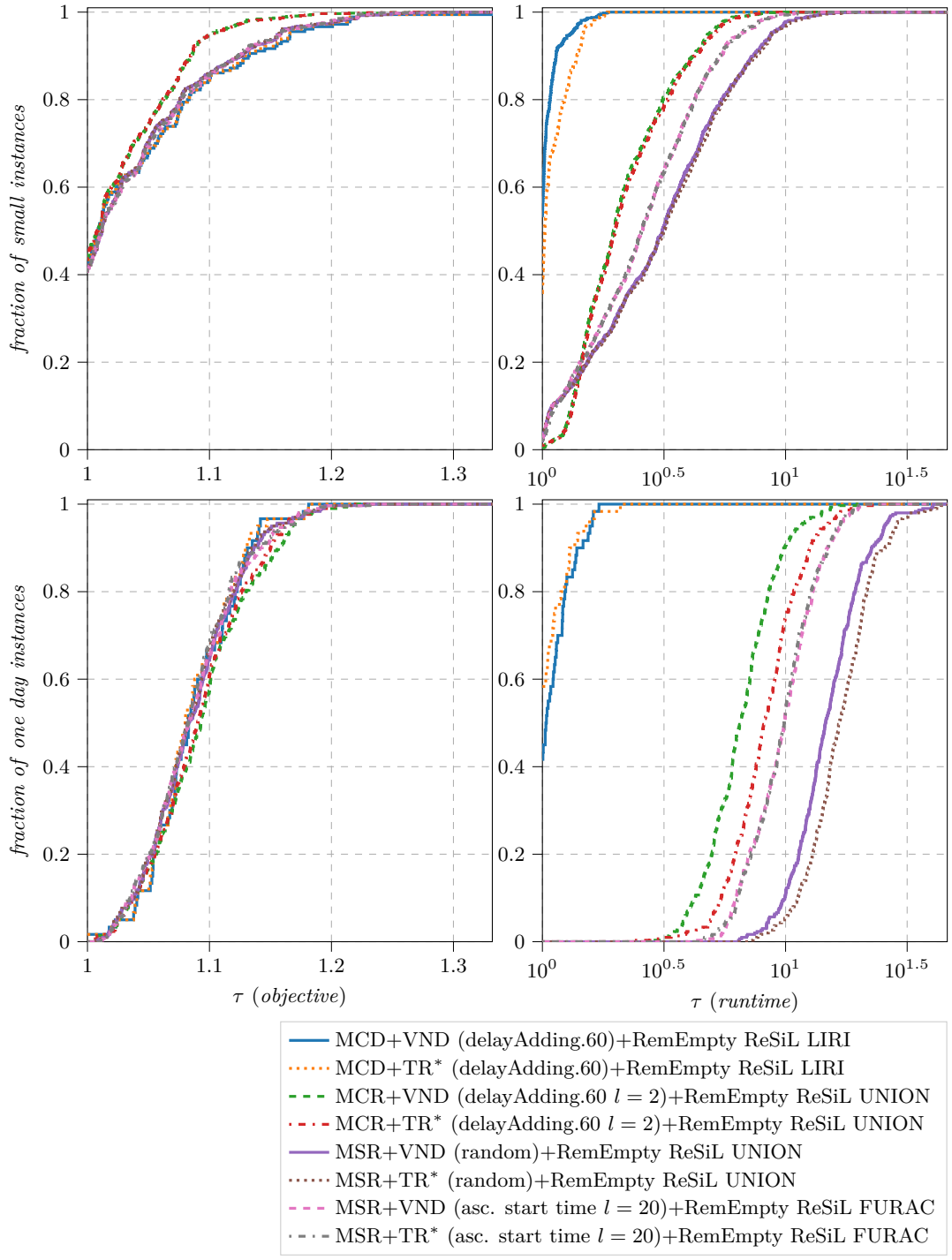


Figure 5.11: Performance profiles of combinations of SwS, ShS, SwL and ShL neighborhoods in a VND, compared with their TR^* counterpart. All settings use BI step function.

	small instances		
rank	setting	wins	ties
1	MCR+VND (delayAdding.60 $l = 2$)+RemEmpty ReSiL UNION	12	2
2	MCR+VND (delayAdding.60 $l = 2$)+RemEmpty ReSiL FURAC	12	2
3	MCR+VND (delayAdding.60 $l = 2$)+LIRI	12	2
4	MSR+VND (random)+RemEmpty ReSiL UNION	7	4
5	MSR+VND (random)+RemEmpty ReSiL FURAC	6	5
6	MSR+VND (random)+LIRI	6	5
7	MSR+VND (asc. start time $l = 20$)+RemEmpty ReSiL UNION	6	5
8	MSR+VND (asc. start time $l = 20$)+LIRI	6	5
9	MSR+VND (asc. start time $l = 20$)+RemEmpty ReSiL FURAC	6	4
10	MCD+VND (delayAdding.60)+RemEmpty ReSiL UNION	3	2
11	MCD+VND (delayAdding.60)+RemEmpty ReSiL FURAC	3	2
12	MCD+VND (delayAdding.60)+LIRI	3	2
13	MSD+VND (desc. ship length)+RemEmpty ReSiL UNION	0	2
14	MSD+VND (desc. ship length)+RemEmpty ReSiL FURAC	0	2
15	MSD+VND (desc. ship length)+LIRI	0	2
	one day instances		
rank	setting	wins	ties
1	MSR+VND (asc. start time $l = 20$)+RemEmpty ReSiL UNION	10	4
2	MCD+VND (delayAdding.60)+RemEmpty ReSiL UNION	10	4
3	MSR+VND (random)+RemEmpty ReSiL UNION	8	5
4	MCD+VND (delayAdding.60)+RemEmpty ReSiL FURAC	6	8
5	MSR+VND (asc. start time $l = 20$)+RemEmpty ReSiL FURAC	6	7
6	MSR+VND (asc. start time $l = 20$)+LIRI	6	7
7	MCD+VND (delayAdding.60)+LIRI	6	7
8	MSR+VND (random)+RemEmpty ReSiL FURAC	6	5
9	MSR+VND (random)+LIRI	6	5
10	MCR+VND (delayAdding.60 $l = 2$)+RemEmpty ReSiL UNION	5	0
11	MCR+VND (delayAdding.60 $l = 2$)+RemEmpty ReSiL FURAC	4	0
12	MCR+VND (delayAdding.60 $l = 2$)+LIRI	3	0
13	MSD+VND (desc. ship length)+RemEmpty ReSiL UNION	2	0
14	MSD+VND (desc. ship length)+RemEmpty ReSiL FURAC	0	1
15	MSD+VND (desc. ship length)+LIRI	0	1

Table 5.14: Ranking of VND settings with combinations of SwS, ShS, SwL and ShL neighborhoods. All settings use BI step function.

(i.e., RemEmpty) after an improvement was found in any neighborhood (e.g., ReSiL). So any improvement in ReSiL would result in a search through RemEmpty, that could never find an improvement because the lockage chain was not changed. To avoid this unnecessary step, we used our NSCNI framework to use the *SAME success mode* for the ReSiL neighborhood, which results in the same locally optimal incumbent solution w.r.t. the first two neighborhoods when proceeding to the third.

We used the same neighborhoods and combinations thereof for our VND as for the afore mentioned TRNS, i.e., *LIRI*, *FURAC* and *UNION*. The results in Table 5.14 and 5.15 mainly allow the same conclusions as for the TRNS in the previous section.

We focus our further analysis on the comparison of those two metaheuristics. From 15 settings per instance set, the VND could never perform significantly better than the

Construction Heuristic	LIRI	FURAC	UNION
	small instances		
MCR delayAdding.60 $l = 2$	3.1864 %	3.1925 %	3.2052 %
MSR random	3.0928 %	3.0741 %	3.1089 %
MSR asc. start time $l = 20$	3.0776 %	3.0684 %	3.1003 %
MCD delayAdding.60	2.8430 %	2.7922 %	2.8234 %
MSD desc. ship length	5.4447 %	5.3923 %	5.3278 %
	one day instances		
MCD delayAdding.60	6.8650 %	7.0071 %	7.1818 %
MSR asc. start time $l = 20$	9.2415 %	9.2533 %	9.3720 %
MSR random	10.6227 %	10.6296 %	10.7808 %
MCR delayAdding.60 $l = 2$	13.0648 %	13.1917 %	13.2898 %
MSD desc. ship length	15.5414 %	15.6605 %	16.1905 %

Table 5.15: Relative improvement found by VND settings by combination type and construction heuristic. Per set, results are sorted by the construction heuristics rank.

TRNS. The TRNS on the other hand was significantly better than the VND using two settings for small and nine settings for large instances, as indicated in Table 5.16. More interesting than the very close average objective results are the average runtimes. While for small instances, runtime differences between the two metaheuristics are within 6 % of each other, for large instances they highly differ by neighborhood combination type.

For *UNION*, the VND performs much faster than the TRNS. The *UNION* combination type results basically in three neighborhoods: RemEmpty, ReSiL and a combined very large neighborhood (VLN). Because of the initially explained optimization applied to the VND, the only difference in the two metaheuristics is the fact that the VND returns (or proceeds) to the first neighborhood after a single improvement in the VLN is found and the TRNS stays in that neighborhood until a local optimum is found. We conclude, that applying fast and easy improvements from the smaller neighborhoods after an improvement in the VLN is found is much faster with only a small regression in objective value than repeatedly inspecting the VLN until a local optimum is found. On the other hand, even if no improvements are found in the smaller neighborhoods, the time spend there is so much less than the time needed to search for a local optimum in the VLN, that in the end the VND (without local optima) is on average approximately 26 % faster than the TRNS (with local optima).

For the other types *FURAC* and *LIRI*, we see that with increasing number of neighborhoods, restarting often from the first neighborhood takes much longer (up to 23 %). A detailed analysis of a single problem instance is presented in the following subsection.

We finally conclude that - for the evaluated neighborhoods and combination types - VND (without local optima) is not better if not even worse than a TRNS with local optima.

5.4.5 Exemplary Comparison of TRNS and VND

We analysed a single problem instance's results solved with two different metaheuristics, TRNS and VND. Both settings use the same construction heuristic (MSR asc. start

	Construction Heuristic	small instances		one day instances	
		objective	runtime	objective	runtime
LIRI	MCD delayAdding.60	+0.057 %	-4.270 %	+0.140 %	+6.050 %
	MCR delayAdding.60 $l = 2$	+0.004 %	-3.440 %	*+0.167 %	+9.450 %
	MSD desc. ship length	-0.068 %	-2.980 %	+0.600 %	+23.500 %
	MSR random	+0.039 %	-2.480 %	*+0.146 %	+12.060 %
	MSR asc. start time $l = 20$	*+0.047 %	-3.040 %	*+0.114 %	+8.350 %
FURAC	MCD delayAdding.60	+0.090 %	+3.280 %	+0.145 %	-9.270 %
	MCR delayAdding.60 $l = 2$	+0.005 %	-2.790 %	+0.089 %	+4.240 %
	MSD desc. ship length	+0.074 %	+5.850 %	*+0.292 %	+9.810 %
	MSR random	+0.042 %	-0.610 %	*+0.125 %	+6.050 %
	MSR asc. start time $l = 20$	+0.039 %	-0.140 %	*+0.103 %	+3.630 %
UNION	MCD delayAdding.60	+0.078 %	+0.020 %	+0.030 %	-23.940 %
	MCR delayAdding.60 $l = 2$	-0.001 %	-1.840 %	*+0.207 %	-24.570 %
	MSD desc. ship length	+0.025 %	-3.880 %	+0.159 %	-28.920 %
	MSR random	*+0.017 %	-1.930 %	*+0.101 %	-26.700 %
	MSR asc. start time $l = 20$	+0.016 %	-2.360 %	*+0.069 %	-26.650 %

Table 5.16: Performance comparison between VND and TRNS. This table shows the average VND results relative to those of TRNS. Entries marked with * show significant differences in objective values.

time $l = 20$), neighborhoods (*LIRI*, $r^{max} = 10$), step function (BI) and random seed (1000). They only differ in their neighborhood selectors success mode, which is *SAME* for the TRNS (with local optimum) and *FIRST* for the VND (without local optimum). Both settings yielded the same objective value. The resulting schedules only diverge in a single empty lockages position, but are otherwise identical. Beside this neglectable difference the VND took 95.4 % longer than the TRNS. We want to emphasize, that this instance was chosen as an extreme example to illustrate the differences between the two metaheuristics. The numbers and relations are not the general case, as the average runtime of VND is only 8.35 % higher as can be seen in Table 5.16.

Looking at the traces of applied moves, it can be seen that both metaheuristics apply the same moves. The only exceptions are two ReSiL moves in the VND that eventually cancel each other out and one ShL move by the VND that results in the differing empty lockage compared to two SwL moves of the TRNS. The remaining 37 moves are identical, just applied in a slightly different order. The VNDs double in runtime does therefore not result from inefficient move selection, but from the number of neighborhood explorations, as can be seen in Table 5.17.

The *LIRI* neighborhood combination is not well suited for VND for several reasons. It results in a large number of small similar neighborhoods, while VND is typically used on only a few neighborhoods (see [HMMP10]). The VND works best, if an improvement in a later neighborhood moves the incumbent solution to a location, where many improvements in the earlier neighborhoods can be found, before descending into later neighborhoods becomes necessary again to escape a local optimum. In the given neighborhood combination, this is not given as can be seen from the large number of

Neighbourhood	VND					TRNS				
	Expl	Σ ms	#I	Σ I	Σ I/#I	Expl	Σ ms	#I	Σ I	Σ I/#I
RemEmpty	35+	106	2	4000	2000	5	91	2	4000	2000
ReSiL*	39	1702	6	3170	528	7	586	4	2532	633
SwL1*	33	5728	13	16158	1243	18	3279	15	19212	1281
ShL1	20	3305	0	0	0	3	469	0	0	0
SwS1	20	606	4	465	116	7	194	4	465	116
ShS1	16	650	2	157	79	4	173	2	157	79
SwL2	14	2059	0	0	0	2	293	0	0	0
ShL2*	14	4408	3	3445	1148	4	1274	2	1029	515
SwS2	11	997	2	304	152	4	360	2	304	152
ShS2	9	1867	3	8514	2838	5	999	3	8514	2838
SwL3	6	941	0	0	0	2	301	0	0	0
ShL3	6	1991	2	1706	853	4	1208	2	1706	853
SwS3	4	239	0	0	0	2	112	0	0	0
ShS3	4	477	0	0	0	2	216	0	0	0
SwL4	4	637	0	0	0	2	258	0	0	0
ShL4	4	1295	1	1626	1626	3	820	1	1626	1626
SwS4	3	211	0	0	0	2	111	0	0	0
ShS4	3	510	0	0	0	2	257	0	0	0
SwL5	3	467	0	0	0	2	274	0	0	0
ShL5	3	895	0	0	0	2	515	0	0	0
SwS5	3	182	0	0	0	2	107	0	0	0
ShS5	3	460	1	11362	11362	3	355	1	11362	11362
SwL6	2	218	0	0	0	2	208	0	0	0
ShL6	2	505	0	0	0	2	485	0	0	0
SwS6	2	88	0	0	0	2	86	0	0	0
ShS6	2	212	0	0	0	2	206	0	0	0
SwL7	2	210	0	0	0	2	186	0	0	0
ShL7	2	479	1	0	0	3	648	1	0	0
SwS7	1	39	0	0	0	2	82	0	0	0
ShS7	1	93	0	0	0	2	184	0	0	0
SwL8	1	82	0	0	0	2	156	0	0	0
ShL8	1	196	0	0	0	2	363	0	0	0
SwS8	1	37	0	0	0	2	58	0	0	0
ShS8	1	90	0	0	0	2	163	0	0	0
SwL9	1	65	0	0	0	2	127	0	0	0
ShL9	1	178	0	0	0	2	328	0	0	0
SwS9	1	19	0	0	0	2	44	0	0	0
ShS9	1	86	0	0	0	2	185	0	0	0
SwL10	1	54	0	0	0	2	107	0	0	0
ShL10	1	156	0	0	0	2	305	0	0	0
SwS10	1	15	0	0	0	2	34	0	0	0
ShS10	1	68	0	0	0	2	128	0	0	0
total	283	32623	40	50907	1273	128	16335	39	50907	1305

Table 5.17: Neighborhood exploration (Expl) and improvement (I) statistics of VND and TRNS on a single large instance. Neighborhoods marked with * have differing number of improvements found.
+: The number of explorations for the RemEmpty neighborhood in the VND is reduced by 6, because of the extension mentioned in Subsection 5.4.4.

unsuccessful neighborhood iterations in Table 5.17. For example neighborhood *ShL1* is visited 20 times without success by the VND, while the TRNS only needs less than three rounds though all neighborhoods without success, before finishing.

The VNDs nature of always iterating the neighborhoods from the beginning has on the other hand one major advantage while designing the neighborhood combination. For the given instance, we can clearly see that the neighborhoods of ranges above 7 never yield an improvement and can (for the given instance) be omitted. If such a threshold can be found for a significant number of instances, the VND can be optimized by removing those neighborhoods from the *LIRI*, as they never or hardly ever help to escape local optima. This process of finding the right parameters for a metaheuristic is often called parameter tuning. The impact of reducing the maximum range for *LIRI* can also be seen in Table 5.19. While we can be sure that the VND requires the latest neighborhood with an improvement (e.g., *ShL7*) to escape a local optimum, the same assumption is not necessarily true for the TRNS. There it could be possible, that the local optimum after the previous move (in *ShS5*) could be overcome by moves in earlier neighborhoods. So the necessity of the neighborhood *ShL7* is not granted for the TRNS. This characteristic makes parameter tuning for a TRNS much harder than for a VND.

5.4.6 Extended Token-Ring Neighborhood Search (TRNS)

Based on previous experiments, we combined the TRNS with *LIRI* (see Subsection 5.4.3) and the local search with IT (see Subsection 5.4.2) into a single VNS. We evaluated eight different types, based on the combination of three parameters:

- *LIRI* range (5 vs. 10)
- IT with or without local optimum
- IT with or without upstream neighborhoods RemEmpty, ReSiL

All settings use BI step function and *SAME* success mode, except for the IT without local optimum, where *FIRST* is used for that specific neighborhood (denoted by trailing *-F* in the settings name).

As table Table 5.18 shows, settings constructed with *MSR* clearly dominate the ranking. We draw similar conclusions as for the pure IT local search in Subsection 5.4.2: The randomized construction heuristics give a higher diversity that allow to overcome local optima and the initial solutions are worse than those of *MCD*, which might get trapped in a local optimum too early.

The Extended TRNS achieves by far the best results so far. As the performance profiles in Figure 5.12 show, it is able to find about 60 % of the best known results for small instances and can solve 90 % of all small instances within 5 % of the best known value (9 % for large instances). Worst case performance (across all instances) is 20 % above the best known objective.

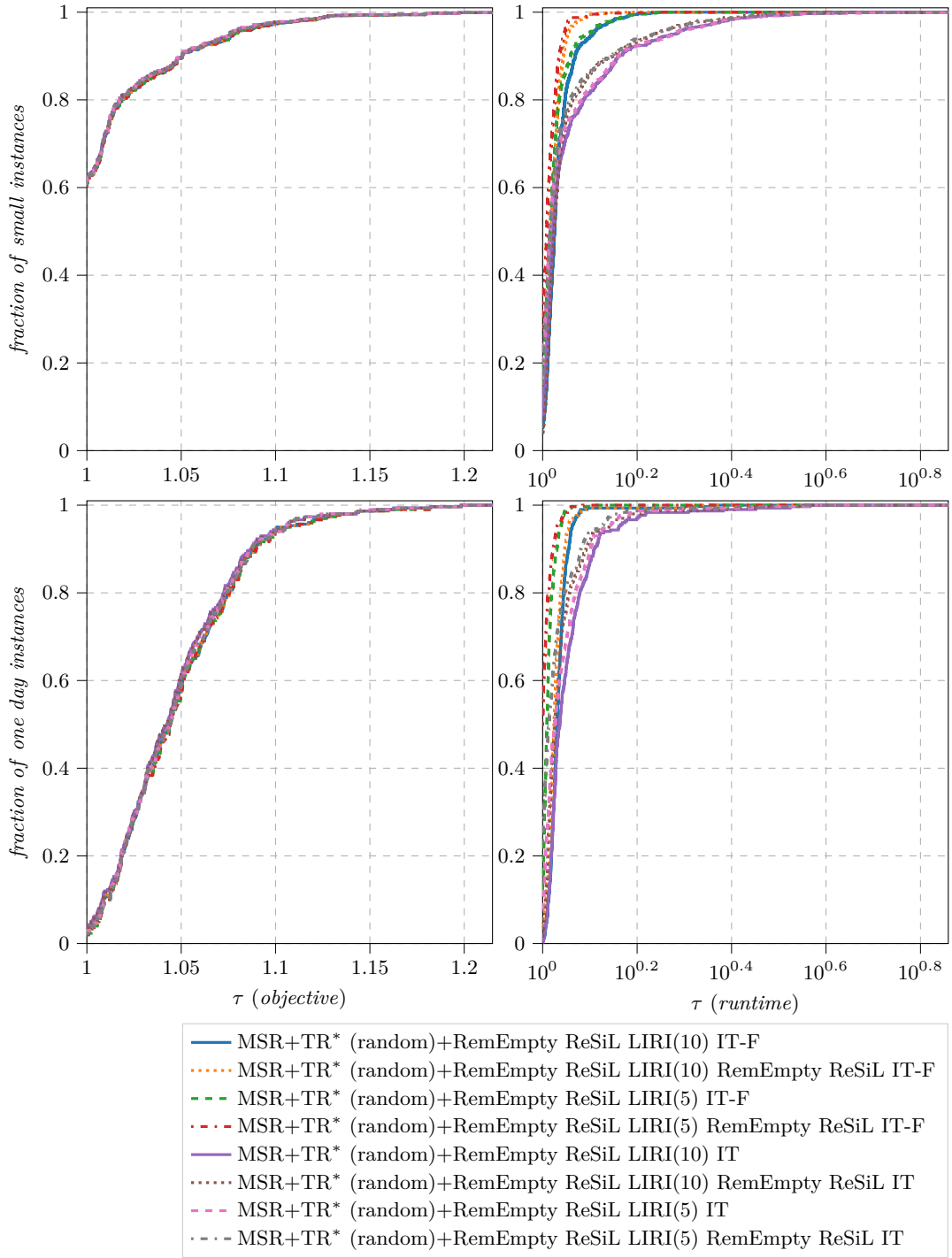


Figure 5.12: Performance profiles of TRNS with *LIRI*, combined with IT. All settings use BI step function and *MSR random* as the construction heuristic.

	small instances		
rank	setting	wins	ties
1	MSR+TR* (random)+LIRI(5) IT	31	8
2	MSR+TR* (random)+LIRI(10) RemEmpty ReSiL IT	30	9
3	MSR+TR* (random)+LIRI(10) IT	27	12
4	MSR+TR* (asc. start time $l = 20$)+LIRI(5) RemEmpty ReSiL IT	27	12
5	MSR+TR* (asc. start time $l = 20$)+LIRI(5) IT	27	12
6	MSR+TR* (asc. start time $l = 20$)+LIRI(10) RemEmpty ReSiL IT	27	12
7	MSR+TR* (asc. start time $l = 20$)+LIRI(10) IT	27	12
8	MSR+TR* (random)+LIRI(5) RemEmpty ReSiL IT	22	17
9	MSR+TR* (asc. start time $l = 20$)+LIRI(5) RemEmpty ReSiL IT-F	18	15
10	MSR+TR* (asc. start time $l = 20$)+LIRI(5) IT-F	18	15
11	MSR+TR* (asc. start time $l = 20$)+LIRI(10) RemEmpty ReSiL IT-F	18	14
12	MSR+TR* (asc. start time $l = 20$)+LIRI(10) IT-F	18	14
13	MCR+TR* (delayAdding.60 $l = 2$)+LIRI(10) IT	16	17
14	MCR+TR* (delayAdding.60 $l = 2$)+LIRI(5) IT	15	17
15	MCR+TR* (delayAdding.60 $l = 2$)+LIRI(10) RemEmpty ReSiL IT	14	17
16	MCR+TR* (delayAdding.60 $l = 2$)+LIRI(5) RemEmpty ReSiL IT	13	17
17	MSR+TR* (random)+LIRI(5) RemEmpty ReSiL IT-F	12	23
18	MSR+TR* (random)+LIRI(5) IT-F	12	23
19	MSR+TR* (random)+LIRI(10) RemEmpty ReSiL IT-F	12	23
20	MSR+TR* (random)+LIRI(10) IT-F	12	23
...			
	one day instances		
rank	setting	wins	ties
1	MSR+TR* (random)+LIRI(10) RemEmpty ReSiL IT	37	2
2	MSR+TR* (random)+LIRI(10) IT	37	2
3	MSR+TR* (random)+LIRI(5) IT	32	5
4	MSR+TR* (asc. start time $l = 20$)+LIRI(10) IT	31	8
5	MSR+TR* (random)+LIRI(5) RemEmpty ReSiL IT	30	7
6	MSR+TR* (random)+LIRI(10) RemEmpty ReSiL IT-F	29	8
7	MSR+TR* (random)+LIRI(10) IT-F	29	8
8	MSR+TR* (asc. start time $l = 20$)+LIRI(10) RemEmpty ReSiL IT	28	8
9	MSR+TR* (random)+LIRI(5) RemEmpty ReSiL IT-F	25	8
10	MSR+TR* (random)+LIRI(5) IT-F	25	8
11	MSR+TR* (asc. start time $l = 20$)+LIRI(10) IT-F	23	11
12	MSR+TR* (asc. start time $l = 20$)+LIRI(5) IT	23	10
13	MCR+TR* (delayAdding.60 $l = 2$)+LIRI(10) IT	23	9
14	MSR+TR* (asc. start time $l = 20$)+LIRI(10) RemEmpty ReSiL IT-F	22	11
15	MSR+TR* (asc. start time $l = 20$)+LIRI(5) RemEmpty ReSiL IT	22	8
16	MCR+TR* (delayAdding.60 $l = 2$)+LIRI(5) IT	21	8
17	MCR+TR* (delayAdding.60 $l = 2$)+LIRI(10) RemEmpty ReSiL IT	21	7
18	MCR+TR* (delayAdding.60 $l = 2$)+LIRI(5) RemEmpty ReSiL IT	20	6
19	MSR+TR* (asc. start time $l = 20$)+LIRI(5) IT-F	19	6
20	MSR+TR* (asc. start time $l = 20$)+LIRI(5) RemEmpty ReSiL IT-F	17	7
...			

Table 5.18: Ranking of extended TRNS settings (showing Top 20 of 40 settings). Initial RemEmpty and ReSiL neighborhoods are omitted due to lack of space.

Table Table 5.20 shows that the different parameter configurations lead to minimal changes in the objective values, that finally lead to very close results that differ only marginally. Again, the runtime analysis is more interesting. The runtime performance profile in Figure 5.12 shows that each evaluated parameter has an impact on the runtime. The same scheme can be observed with any evaluated construction heuristic.

- Reducing the range of *LIRI* has a moderate impact on the runtime, but does not improve worst case performance.
- Adding upstream neighborhoods before search IT has less impact on best or worst case performance, but improves the mid ranges.
- Restarting the VNS after each improvement in IT (i.e., not searching until a local optimum is found) mostly impacts worst case runtime.

Table Table 5.19 gives an overview on the average impact of the parameter values. For each parameter of our Extended TRNS it shows the average difference between all algorithm settings using parameter value *A* and all settings using parameter value *B*. For each of the four comparisons, the settings that only differ in the tested parameter are compared and average differences in objective and runtime are calculated. The table finally shows the average over those four value pairs. It can be seen, that no parameter value drastically changes the objective value, but small effects can accumulate to finally result in even significant differences. The biggest effect on the runtime (and the objective value) has the search for a local optimum at every visit of the IT neighborhood, which - as stated before - can be very large. This is also caused by the step function as the used BI method requires the inspection of the whole neighborhood at every step.

5.4.7 General Variable Neighborhood Search (GVNS)

Our final VNS implementation was a General Variable Neighborhood Search (GVNS) that used the Extended TRNS (see Subsection 5.4.6) as the nested local search. For shaking the incumbent solution before the TRNS iterations we used the Rebuild Water Gate (RebWG) neighborhood described in 4.3.12. We evaluated different step functions (BI and RI) that were applied to the *LIRI* neighborhood series and IT neighborhood and also different success modes for the IT neighborhood (*SAME* and *FIRST*), all denoted by the appended suffixes in the settings description.

As with the Extended TRNS above, Table 5.21 clearly shows the dominance of *MSR random* based settings. The GVNS with Extended TRNS achieves even slightly better results than the Extended TRNS. As the performance profiles in Figure 5.13 show, it is able to find about 60 % of the best known results for small instances and can solve 90 % of all small instances within 5 % of the best known value (8 % for large instances). Worst case performance (across all instances) is 20 % above the best known objective.

As table Table 5.23 shows, the single shaking used for our implementation is able to overcome a local optimum of the nested TRNS in about 32 % for large instances. The table

	Construction Heuristic	small instances		one day instances	
		objective	runtime	objective	runtime
LIRI range 10 vs. 5	MCD delayAdding.60	-0.037 %	+4.285 %	+0.017 %	+14.513 %
	MCR delayAdding.60 $l = 2$	-0.003 %	+4.048 %	-0.067 %	+12.820 %
	MSD desc. ship length	-0.022 %	+3.575 %	-0.154 %	+11.203 %
	MSR random	+0.002 %	+3.910 %	-0.083 %	+15.893 %
	MSR asc. start time $l = 20$	-0.002 %	+4.053 %	-0.171 %	+14.938 %
IT loc. opt. with vs. w/o	MCD delayAdding.60	-0.113 %	+13.355 %	-0.051 %	+25.873 %
	MCR delayAdding.60 $l = 2$	-0.027 %	+11.390 %	-0.257 %	+34.165 %
	MSD desc. ship length	+0.007 %	+15.728 %	-0.509 %	+37.045 %
	MSR random	-0.054 %	+12.750 %	-0.125 %	+20.838 %
	MSR asc. start time $l = 20$	-0.050 %	+12.063 %	-0.149 %	+18.510 %
IT up. NHs w/o vs. with	MCD delayAdding.60	-0.016 %	+3.293 %	-0.020 %	+11.518 %
	MCR delayAdding.60 $l = 2$	-0.003 %	+5.515 %	-0.037 %	+10.743 %
	MSD desc. ship length	-0.005 %	+3.092 %	+0.083 %	+8.110 %
	MSR random	-0.003 %	+3.720 %	-0.014 %	+8.188 %
	MSR asc. start time $l = 20$	-0.002 %	+3.815 %	-0.053 %	+10.125 %

Table 5.19: Performance comparison between different extended TRNS settings. This table shows the average results aggregated over 4 settings with parameter value A relative to those of parameter value B (stated as *A vs. B*).

Construction Heuristic	LIRI range 5	LIRI range 10	IT local optimum with	IT local optimum without	IT upstream NHs without	IT upstream NHs with
	small instances					
MCR delayAdding.60 $l = 2$	4.41 %	* 4.41 %	* 4.42 %	4.39 %	* 4.41 %	4.41 %
MSR random	* 5.29 %	5.29 %	* 5.31 %	5.26 %	* 5.29 %	5.29 %
MSR asc. start time $l = 20$	5.29 %	* 5.29 %	* 5.31 %	5.27 %	* 5.29 %	5.29 %
MCD delayAdding.60	5.04 %	* 5.08 %	* 5.11 %	5.01 %	* 5.07 %	5.05 %
MSD desc. ship length	8.88 %	* 8.90 %	* 8.89 %	8.90 %	* 8.89 %	8.89 %
	one day instances					
MCD delayAdding.60	* 9.47 %	9.46 %	* 9.49 %	9.44 %	* 9.48 %	9.45 %
MSR asc. start time $l = 20$	12.22 %	*12.37 %	*12.36 %	12.23 %	*12.32 %	12.27 %
MSR random	13.91 %	*13.98 %	*14.00 %	13.89 %	*13.95 %	13.94 %
MCR delayAdding.60 $l = 2$	16.46 %	*16.52 %	*16.60 %	16.39 %	*16.51 %	16.48 %
MSD desc. ship length	20.51 %	*20.63 %	*20.77 %	20.37 %	20.54 %	*20.60 %

Table 5.20: Relative improvement found by extended TRNS with different parameter values. Entries marked with * have higher improvements in objective values.

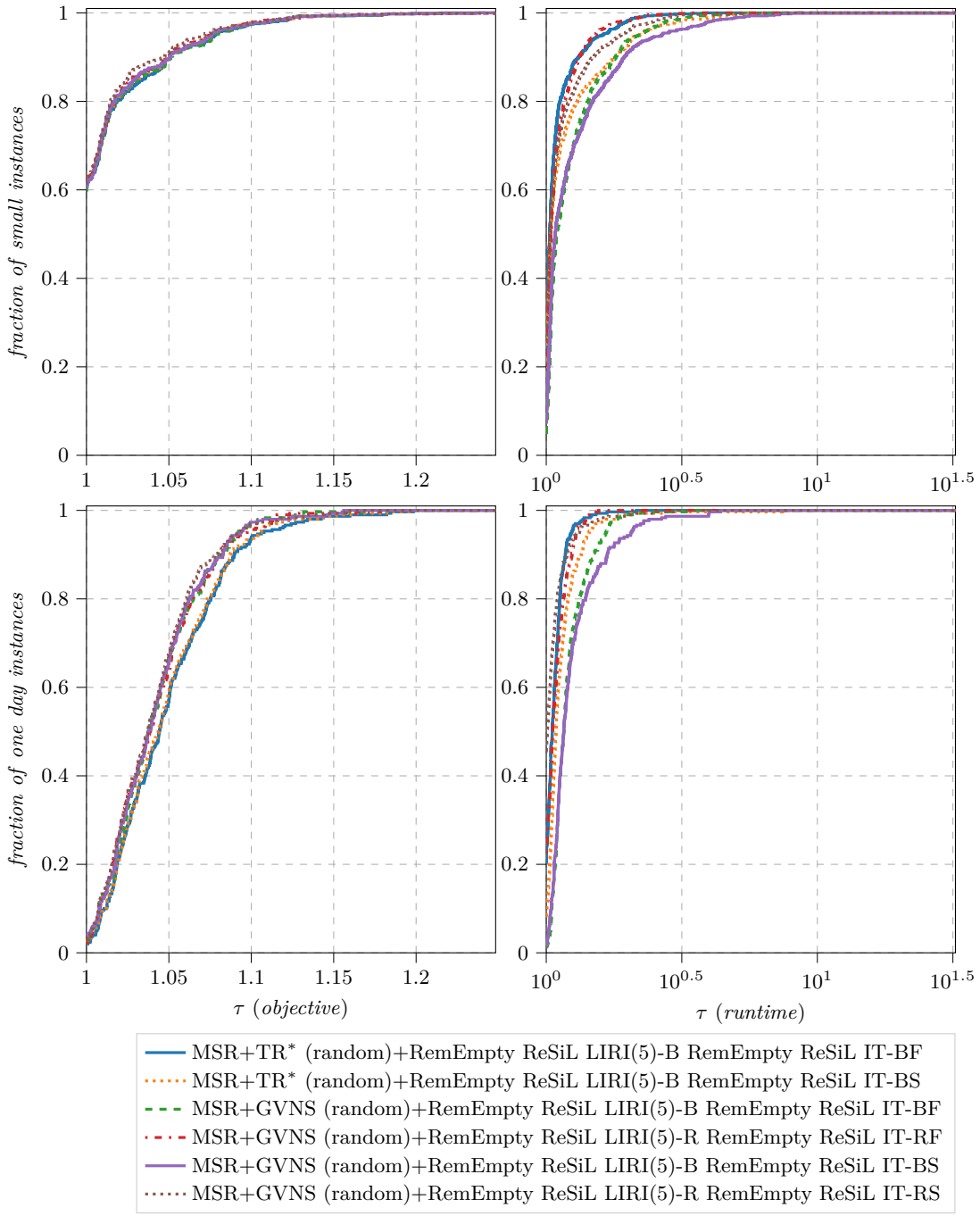


Figure 5.13: Performance profiles of GVNS with *Extended TRNS*. All settings use *MSR random* as the construction heuristic.

	small instances		
rank	setting	wins	ties
1	MSR+GVNS (random)+LIRI(5)-R RemEmpty ReSiL IT-RS	15	0
2	MSR+GVNS (random)+LIRI(5)-R RemEmpty ReSiL IT-RF	10	4
3	MSR+GVNS (random)+LIRI(5)-B RemEmpty ReSiL IT-BS	10	4
4	MSR+GVNS (asc. start time $l = 20$)+LIRI(5)-R RemEmpty ReSiL IT-RS	10	4
5	MSR+GVNS (asc. start time $l = 20$)+LIRI(5)-R RemEmpty ReSiL IT-RF	10	4
6	MSR+GVNS (asc. start time $l = 20$)+LIRI(5)-B RemEmpty ReSiL IT-BS	10	4
7	MSR+GVNS (random)+LIRI(5)-B RemEmpty ReSiL IT-BF	7	2
8	MSR+GVNS (asc. start time $l = 20$)+LIRI(5)-B RemEmpty ReSiL IT-BF	7	2
9	MCR+GVNS (delayAdding.60 $l = 2$)+LIRI(5)-R RemEmpty ReSiL IT-RS	3	6
10	MCR+GVNS (delayAdding.60 $l = 2$)+LIRI(5)-B RemEmpty ReSiL IT-BS	3	4
11	MCR+GVNS (delayAdding.60 $l = 2$)+LIRI(5)-R RemEmpty ReSiL IT-RF	2	5
12	MCR+GVNS (delayAdding.60 $l = 2$)+LIRI(5)-B RemEmpty ReSiL IT-BF	2	4
13	MCD+GVNS (delayAdding.60)+LIRI(5)-R RemEmpty ReSiL IT-RS	1	6
14	MCD+GVNS (delayAdding.60)+LIRI(5)-B RemEmpty ReSiL IT-BS	0	6
15	MCD+GVNS (delayAdding.60)+LIRI(5)-B RemEmpty ReSiL IT-BF	0	3
16	MCD+GVNS (delayAdding.60)+LIRI(5)-R RemEmpty ReSiL IT-RF	0	2
	one day instances		
rank	setting	wins	ties
1	MSR+GVNS (random)+LIRI(5)-R RemEmpty ReSiL IT-RS	13	2
2	MSR+GVNS (random)+LIRI(5)-B RemEmpty ReSiL IT-BS	11	4
3	MSR+GVNS (asc. start time $l = 20$)+LIRI(5)-B RemEmpty ReSiL IT-BS	11	4
4	MSR+GVNS (random)+LIRI(5)-B RemEmpty ReSiL IT-BF	11	3
5	MSR+GVNS (random)+LIRI(5)-R RemEmpty ReSiL IT-RF	10	4
6	MSR+GVNS (asc. start time $l = 20$)+LIRI(5)-R RemEmpty ReSiL IT-RF	5	6
7	MCR+GVNS (delayAdding.60 $l = 2$)+LIRI(5)-R RemEmpty ReSiL IT-RS	5	5
8	MCR+GVNS (delayAdding.60 $l = 2$)+LIRI(5)-B RemEmpty ReSiL IT-BS	5	5
9	MSR+GVNS (asc. start time $l = 20$)+LIRI(5)-R RemEmpty ReSiL IT-RS	4	6
10	MSR+GVNS (asc. start time $l = 20$)+LIRI(5)-B RemEmpty ReSiL IT-BF	4	6
11	MCR+GVNS (delayAdding.60 $l = 2$)+LIRI(5)-R RemEmpty ReSiL IT-RF	4	6
12	MCD+GVNS (delayAdding.60)+LIRI(5)-R RemEmpty ReSiL IT-RS	2	5
13	MCR+GVNS (delayAdding.60 $l = 2$)+LIRI(5)-B RemEmpty ReSiL IT-BF	0	4
14	MCD+GVNS (delayAdding.60)+LIRI(5)-B RemEmpty ReSiL IT-BS	0	4
15	MCD+GVNS (delayAdding.60)+LIRI(5)-R RemEmpty ReSiL IT-RF	0	3
16	MCD+GVNS (delayAdding.60)+LIRI(5)-B RemEmpty ReSiL IT-BF	0	3

Table 5.21: Ranking of GVNS with *Extended TRNS* settings. Initial RemEmpty and ReSiL neighborhoods are omitted due to lack of space.

Construction Heuristic	IT with local optimum BI	IT with local optimum RI	IT without local optimum BI	IT without local optimum RI
	small instances			
MCR delayAdding.60 $l = 2$	4.4261 %	* 4.4139 %	4.4071 %	4.3896 %
MSR random	5.3154 %	* 5.4178 %	5.2672 %	5.3396 %
MSR asc. start time $l = 20$	* 5.3330 %	* 5.3711 %	5.2881 %	* 5.3502 %
MCD delayAdding.60	5.0345 %	* 5.0537 %	4.9923 %	4.9880 %
	one day instances			
MCD delayAdding.60	10.0369 %	*10.1492 %	9.9659 %	9.9349 %
MSR asc. start time $l = 20$	*12.8637 %	12.6362 %	12.5964 %	12.6412 %
MSR random	14.4241 %	*14.5148 %	14.3936 %	14.3923 %
MCR delayAdding.60 $l = 2$	*17.0446 %	*17.1002 %	16.8479 %	16.9760 %

Table 5.22: Relative improvement found by GVNS settings by TRNS parameters. Per set, results are sorted by the construction heuristics rank. Settings marked with * are the best per construction heuristic.

Construction Heuristic	IT with local optimum BI	IT with local optimum RI	IT without local optimum BI	IT without local optimum RI
	small instances			
MCR delayAdding.60 $l = 2$	2.00 %	* 4.00 %	1.78 %	4.11 %
MSR random	2.56 %	* 4.22 %	2.11 %	2.89 %
MSR asc. start time $l = 20$	* 2.56 %	* 3.67 %	2.44 %	* 2.56 %
MCD delayAdding.60	2.00 %	* 4.33 %	2.67 %	3.78 %
	one day instances			
MCD delayAdding.60	29.33 %	*26.33 %	31.33 %	34.00 %
MSR asc. start time $l = 20$	*30.67 %	32.33 %	33.67 %	33.00 %
MSR random	34.00 %	*29.67 %	29.33 %	36.33 %
MCR delayAdding.60 $l = 2$	*32.00 %	*35.67 %	33.67 %	30.33 %

Table 5.23: Relative amount of instance runs with at least one successful shaking to escape a local optimum of the nested TRNS. Per set, results are sorted by the construction heuristics rank. Settings marked with * are the best per construction heuristic.

also shows, that the solution quality does not correlate with the amount of successfully applied shakings. A nested VNS can find good solutions even without applying random moves to step out of a local optimum. On the other hand, this also indicates that there is room for more elaborate shaking strategies than the one applied.

For small instances, the fraction of successful shakings drops to about 3 %. We see several reasons for this:

- About 22 % of all small instances are already solved to the best known value by the initial construction heuristic. The GVNS applies the first shaking initially after the construction, before running the nested VNS. So for 22 % of all instances the best known solution is destroyed right away. If the nested VNS is unable to find the initial solution, the GVNS stops and the shaking is considered unsuccessful. In 23.8 % of small instance runs, the nested TRNS was not able to find a better solution than the initial construction heuristic after the shaking was applied. This

never happens for large instances, so the initial solution after construction is always improved.

- About 60 % of all small instances can be solved to the best known value with the extended TRNS. As for the initial solutions, the same holds for the first loop of the nested VNS. In about 38 % of all small instance runs, the extended TRNS is able to improve the initial solution to the best known solution, so any shaking applied destroys a very good (possibly globally optimal) solution, which cannot be further improved by a subsequent VNS iteration. In 73.2 % of small instance runs (68 % for large instances), the nested TRNS was not able to find a better solution after the second shaking was applied.
- The applied shaking strategy is likely inappropriate, especially for small instances. Naively rescheduling a single water gate might produce too few diversification to escape a local optimum. With only a few ships scheduled, it might even happen, that no change to the schedule is applied at all. For large instances, the shaking might be too extreme, introducing too much diversification in a single step.

Looking at the runtime profiles in Figure 5.13 we clearly see, that the speed up from using RI as a step function is able to overcome the runtime costs of multiple VNS iterations using GVNS. For example the best performing GVNS *MSR+GVNS (random)+LIRI(5)-R RemEmpty ReSiL IT-RS* is on average 11 % faster (median 25 %) than its Extended TRNS counterpart using BI step function with an expected improvement of 0.7 % on the objective value.

5.4.8 Comparison of Heuristics

In this section, we evaluated several improvement heuristics based on different methods like Local Search, Token-Ring Neighborhood Search (TRNS), Variable Neighborhood Descent (VND) and General Variable Neighborhood Search (GVNS). The ranking in Table 5.24 and performance profiles in Figure 5.14 clearly show the dominance of TRNS based metaheuristics, especially the GVNS with RI step function. While we focus only on a single construction heuristic (*MSR random*) for the final ranking to keep results comparable, we still want to note that *MCD+GVNS (delayAdding.60)+LIRI(5)-R RemEmpty ReSiL IT-RS* is much faster than 97 % of all *MSR random* based heuristics for large instances. This solely comes from the construction heuristic, as the time spent in the improvement heuristic is approximately equal. The solution quality of this best *MCD* based setting is on average 0.8 % worse than the best *MSR random* based GVNS while only taking 25.9 % runtime on average for large instances.

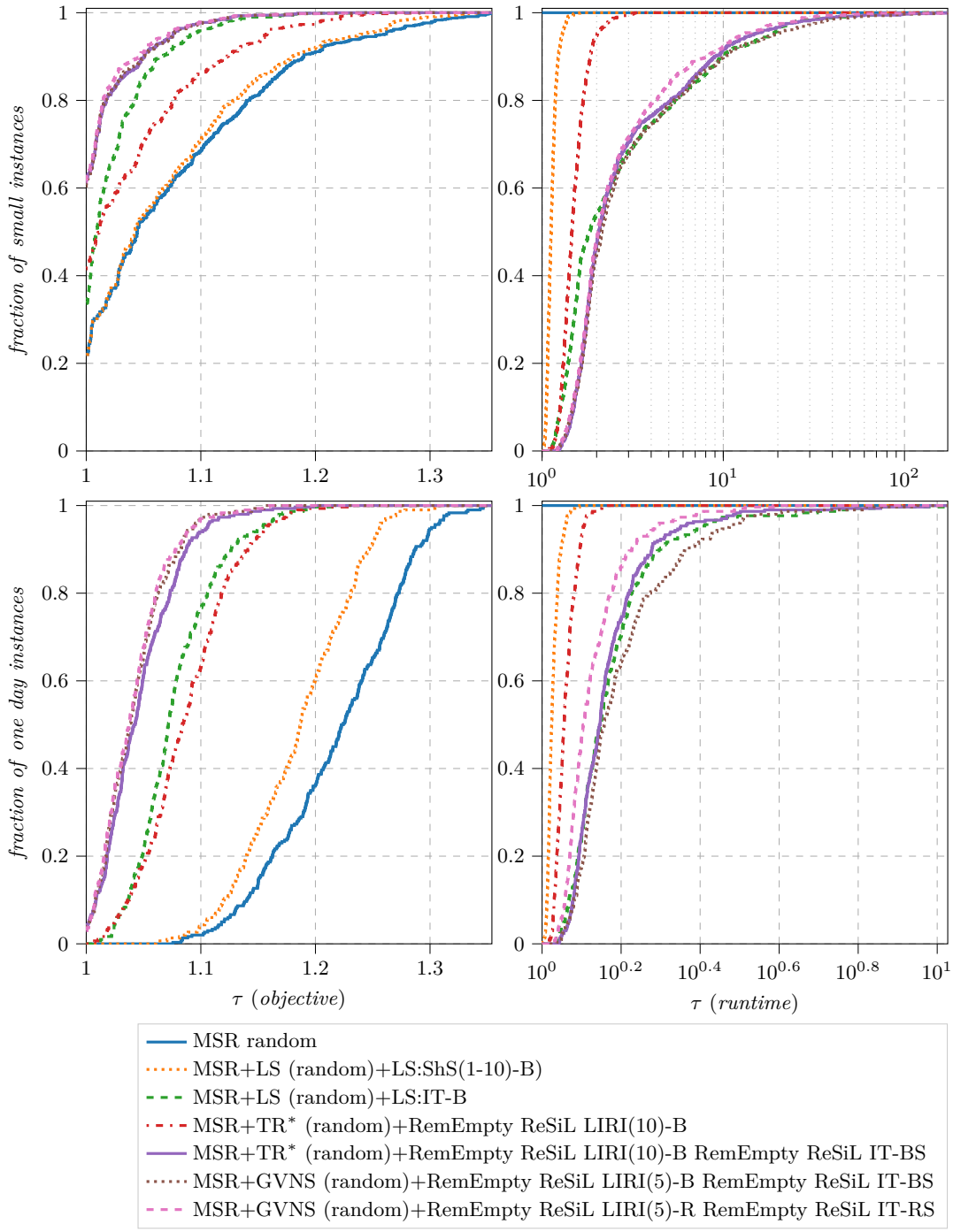


Figure 5.14: Performance profiles of selected heuristics. All settings use *MSR random* as the construction heuristic.

	small instances		
rank	setting	wins	ties
1	MSR+GVNS (random)+RR LIRI(5)-RS RR IT-RS	22	0
2	MSR+TR* (random)+RR LIRI(10)-B RR IT-BS	16	5
3	MSR+GVNS (random)+RR LIRI(5)-RS RR IT-RF	16	5
4	MSR+GVNS (random)+RR LIRI(5)-BS RR IT-BS	16	5
5	MSR+TR* (random)+RR LIRI(5)-B RR IT-BS	15	6
6	MSR+TR* (random)+RR LIRI(5)-B IT-BS	15	6
7	MSR+TR* (random)+RR LIRI(10)-B IT-BS	15	6
8	MSR+GVNS (random)+RR LIRI(5)-BS RR IT-BF	11	7
9	MSR+TR* (random)+RR LIRI(5)-B RR IT-BF	11	4
10	MSR+TR* (random)+RR LIRI(5)-B IT-BF	11	4
11	MSR+TR* (random)+RR LIRI(10)-B RR IT-BF	11	4
12	MSR+TR* (random)+RR LIRI(10)-B IT-BF	11	4
13	MSR+LS (random)+LS:IT-B	10	0
14	MSR+LS (random)+LS:IT-R	8	1
15	MSR+LS (random)+LS:IT@t0-B	8	1
16	MSR+LS (random)+LS:IT@t0-R	7	0
17	MSR+TR* (random)+RR LIRI(10)-B	4	16
18	MSR+TR* (random)+RR UNION(10)-B	3	3
19	MSR+TR* (random)+RR FURAC(10)-B	2	3
20	MSR+VND (random)+RR UNION(10)-B	1	4
21	MSR+VND (random)+RR LIRI(10)-B	1	4
22	MSR+VND (random)+RR FURAC(10)-B	1	2
23	MSR+LS (random)+LS:ShS(1-10)-B)	0	0
	one day instances		
rank	setting	wins	ties
1	MSR+GVNS (random)+RR LIRI(5)-RS RR IT-RS	21	1
2	MSR+GVNS (random)+RR LIRI(5)-BS RR IT-BS	19	3
3	MSR+GVNS (random)+RR LIRI(5)-RS RR IT-RF	19	2
4	MSR+GVNS (random)+RR LIRI(5)-BS RR IT-BF	19	2
5	MSR+TR* (random)+RR LIRI(10)-B RR IT-BS	17	1
6	MSR+TR* (random)+RR LIRI(10)-B IT-BS	17	1
7	MSR+TR* (random)+RR LIRI(5)-B RR IT-BS	13	3
8	MSR+TR* (random)+RR LIRI(5)-B IT-BS	13	3
9	MSR+TR* (random)+RR LIRI(10)-B RR IT-BF	13	3
10	MSR+TR* (random)+RR LIRI(10)-B IT-BF	13	3
11	MSR+TR* (random)+RR LIRI(5)-B RR IT-BF	11	1
12	MSR+TR* (random)+RR LIRI(5)-B IT-BF	11	1
13	MSR+LS (random)+LS:IT-B	10	0
14	MSR+LS (random)+LS:IT-R	8	1
15	MSR+LS (random)+LS:IT@t0-B	8	1
16	MSR+TR* (random)+RR UNION(10)-B	6	1
17	MSR+TR* (random)+RR LIRI(10)-B	3	8
18	MSR+LS (random)+LS:IT@t0-R	3	4
19	MSR+VND (random)+RR UNION(10)-B	3	3
20	MSR+TR* (random)+RR FURAC(10)-B	3	3
21	MSR+VND (random)+RR FURAC(10)-B	1	1
22	MSR+VND (random)+RR LIRI(10)-B	1	1
23	MSR+LS (random)+LS:ShS(1-10)-B	0	0

Table 5.24: Ranking of heuristics. All settings are based on *MSR random* construction heuristic. *RR* abbreviates *RemEmpty ReSiL*.

CHAPTER 6

Discussion and Conclusions

The Interdependent Lock Scheduling Problem is a complex combinatorial optimization problem. Ships need to be assigned to lockage operations at water gates along their journey. Within a lockage operation, the order of ships has to be specified. What makes these two simple decision a complex problem is the interdependency between subsequent lockage operations at a single gate and at dependent operations at neighboring gates that serve the same ships. On inland waterways that are intensively used for energy production with hydro power and on created canals with several altitudes, embankment damns are a major hindrance for traffic flow. By optimizing the ships lockages, operators can reduce waiting time, total travel time and therefore resource consumption. Secondly, water usage at the locks can be reduced by optimizing the chambers utilization and avoiding empty lockage operations. This leads to an improved efficiency and attractiveness of inland navigation.

In this thesis we introduced and studied the Interdependent Lock Scheduling Problem. The main goal was to provide sustainably applicable solution techniques, which we achieved by combining different neighborhoods into a metaheuristic. This chapter summarizes the main contributions, discusses known shortcomings and gives an outlook on possible future research topics.

6.1 Contributions

The complex interdependent optimization problem of scheduling ships along inland waterways can hardly be tackled by exact methods. Therefore we developed and evaluated several construction heuristics based on different paradigms. These were eventually able to solve real life problem instances in a matter of seconds and provided solutions where in most cases the ships experienced less waiting times than in the historical data. The different heuristics use different concepts like sequential and chronological scheduling. For each construction heuristic we proposed deterministic and randomized implementations.

We further developed multiple neighborhood structures that were used to create metaheuristics to further improve the constructed solutions. To keep our metaheuristics flexible and extendable, we developed a framework that allows each neighborhood in a VNS to have individual settings concerning its step function (next, random or best improvement) and its successive neighborhood when an improvement is found. We focused not only on the objective values but also tried to keep runtime short to propose techniques to solve the ILSP in a sustainable and applicable way. A tool implemented to keep runtime low was a dependency graph that keeps track of direct dependencies between lockage operations. These dependencies allow to propagate updates to the schedule only to affected elements until either the leaves of the graph are reached or the changes disperse. This avoids recalculation of the whole schedule and saves CPU time.

6.2 Discussion of Open Issues and Future Work

All our algorithms follow a simple hill climbing approach, which is prone to getting caught in local optima, except the GVNS which uses shaking to escape them. Alternative strategies that are used for similar scheduling problems would be interesting to evaluate, e.g. simulated annealing, tabu search or back tracking. Concerning the mentioned shaking in GVNS, we used a special neighborhood structure that destroys and naively recreates a single water gates schedule. More elaborate shakings could likely improve the GVNS performance in both solution quality and runtime. Possible alternatives are applying multiple shift/swap moves or reinsertion of one or multiple ships' trips at random.

We experimented with different neighborhood structures, some were created by combining multiple smaller neighborhoods into one. This resulted in very large neighborhoods (VLNs) that we simply explored using standard step functions (mainly best and random improvement), which (especially for BI) takes a lot of CPU time. For those VLNs more sophisticated methods could be used like Adaptive Large Neighborhood Search.

We also experimented with a big number of small neighborhoods. For those, the ordering is a crucial factor to the overall runtime. Self-Adaptive Variable Neighborhood Descend is a strategy that dynamically sorts the neighborhoods, depending on the current solution. Such a dynamic sorting could be a promising extension to our NSCNI framework.

Concerning VLN, for our Improve Trip (IT) neighborhood we already narrowed the search space by using the current best known solutions objective value as a threshold when reinserting a trip. This threshold is narrowed while different possibilities for inserting the same ship are evaluated. When using the BI step function, we could pass the narrowed threshold to the next ships evaluation instead of letting each ship start with the current incumbent solution as a starting point. Quick experiments show a potential reduction of CPU time spent exploring the IT neighborhood of about 30 to 60 %. This improvement was found late in the process and is therefore not part of the analysis in Chapter 5.

Another known weakness of our implementation is the insertion of empty lockage operations to keep the schedule feasible. They are usually added right before a feasibility

violation, e.g., if a lockage should go downstream while all chambers are already empty, an empty upstream operation is inserted right before the violation to be able to add the new downstream operation. This situation could also be repaired by adding the additional fixing operation any time before the critical state, as long as the lockage chain in between stays feasible. If there are multiple candidates for repairing the schedule, this could either result in multiple possible moves to accomplish the initial request (i.e., placing a lockage operation in an infeasible position) which increased the size of the search space, or a sub-heuristic chooses a candidate which keeps the search space size unchanged but increases the runtime of creating the move instruction by the sub-heuristics processing time.

The last issue known to us is with randomization. When exploring the neighborhoods, we improved the runtime by using a priori randomization. In most cases, moves pick a single lockage operation or ship in an operation and modify its schedule. When exploring the neighborhood deterministically, this is done in a two or three-staged manner by iterating over all water gates, lockage operations thereof and possibly ships assigned to them. With a priori randomization, we permute the order of water gates, lockage operations and ships at every stage, when creating the possible move instructions. Finally, we shuffle the list of created move instructions when evaluating them. This results in the issue, that if, e.g., a water gate has few improving moves, they have a higher probability per move to be chosen than moves of another water gate where many improving moves are situated, because all moves of a randomly chosen water gate are evaluated before those of other water gates. To get randomization with equal probabilities per move (a posteriori) all move instructions need to be created and the total pool of moves (i.e., the whole search space) needs to be shuffled before the moves are evaluated. This requires all moves instructions to be at least created (not evaluated) and stored which requires computational resources in terms of time and memory.

6.2.1 Towards a Practical Application

The whole project *imFluss* was driven by practical needs. The ILSP as a scheduling problem at the Danube is part of the daily business of lock masters, ship captains and stake holders along the river and its infrastructure. That is one reason for the need of sustainable, applicable algorithms to solve the ILSP.

But to finally get a usable software tool to actually manage lockage operations, more than a heuristic algorithm is needed. In daily business, the river flow is not that straight as written in the problem specifications. Changing water levels and weather conditions influence the velocity of ships along river sections. Throughout the seasons and during the day, different altitudes of the sun can hinder navigation and mooring. All this leads to variable speed limits and thus travel times for ships along the river sections. These factors were also analysed in the project *imFluss*, but were not part of this thesis.

Beside these factors that try to make a realistic model to allow accurate planing of inland navigation, another crucial aspect of an applicable planning tool is the ability to react

to change. In our algorithm, we assume that all travelling ships are known in advance. This is hardly feasible in a real life environment. Other scheduling software tools usually tackle this by introducing rolling horizons, i.e., the heuristic schedules only the yet known vessels for the next couple of hours. The plan is periodically extended by treating former decisions as fixed and tries to insert newly released vessels into the partially fixed schedule. For the ILSP this approach seems hardly realizable in a strict manner, as this would make it impossible to add new vessels to existing lockages. Whenever a ship is added to an existing lockage that already serves other vessels, some additional waiting time for the other vessels is introduced as they have to either wait for the new ship upon entering or leaving the chamber (given the constraint that ships cannot overtake during entering or leaving). Not being able to add newly released vessels to existing lockage operations would obviously be extremely inefficient. Analysing this aspect of practical application would be interesting for future studies.

List of Figures

List of Tables

List of Algorithms

Glossary

arrival time The arrival time at a water gate defines the time that the ship arrives at the surrounding area of the gate, i.e., the ship arrives at a_g^e or a_g^w respectively. 6

departure time The departure time defines the time at which the ship s has left the surrounding area at water gate g. 7

entering time The entering time at a water gate defines the time that the ship starts entering the chamber. 6

exiting time The exiting time defines the time at which the ship starts leaving the lock chamber at water gate. 7

immutable construction heuristic An immutable construction heuristic never reverts or changes a previous decision including derived values when adding new elements to the partial solution. 23, 29

inland navigation Transport with ships via inland waterways (canals, rivers, lakes etc.) between inland ports or quays 1, 9, 11, 107

lockage time The lockage time at a water gate defines the moment at which the lockage operation begins. At this time all assigned vessels must have finished entering. 7

metaheuristic A general framework to build a heuristic for combinatorial optimization problems 54

move A set of actions used to manipulate a schedule to accomplish a given task. A move results in a change of total costs of the schedule, called the *delta*. 27, 28, 43

move instructions The blueprint of a move. Applying move instructions to a given solution results in a move. 27, 43

mutable construction heuristic A mutable construction heuristic allows changes to previous decision and alter the solution representation including derived values while adding new elements to the partial solution. 23, 29, 36

neighborhood explorer An algorithm creating a single move within its neighborhood based on the current solution 28, 43

neighborhood structure A relation that assigns each solution a set of neighboring solutions by applying moves of a certain kind. 28, 42, 43, 54

optimality gap The optimality gap defines the difference between a solution obtained from a heuristic and the proven optimal solution. It is measured in percent of the optimal solution. 13, 15, 17, 20, 24, 61

parameter tuning The process of finding the most suitable values for a parametrized algorithm. 96

relative gap The relative gap defines the difference between the upper and lower bound of a solution obtained from a MIP solver. It is measured in percent of the difference between the bounds divided by the upper bound. 61, 62

setting A combination of an algorithm together with its relevant parameter assignment. Different parameter configurations for the same algorithm build different settings. 24

shaking A (possibly compound) non-improving move that is used to alter a local optimal solution, s.t. further improving steps can find another local optimal solution. 52, 56, 57

step function A defined rule for selecting the next candidate solution from a neighborhood. 28, 44, 108

Acronyms

AIT Austrian Institute of Technology 2

BI best improvement 12, 13, 28, 43, 44, 51, 52, 55, 56, 80, 83, 86, 88, 89, 91, 92, 94, 96, 97, 99, 104, 108

BMVIT Bundesministerium für Verkehr, Innovation und Technologie (Ministry for Transport, Innovation and Technology) 2

CNS Composite Neighbourhood Search 12

DFS depth first search 51

DP dynamic programming 12, 18

ETA estimated time of arrival 14, 15

FCFS first come first serve 12, 13, 15, 38, 52, 65, 70

GD Gezhouba Dam 13, 14

GVNS General Variable Neighborhood Search 57, 99, 101–104, 108

ILSP Interdependent Lock Scheduling Problem 2, 4–9, 11, 15–23, 27, 29, 43, 59, 61, 65, 108–110

ISD Immutable Sequential Deterministic Construction 29, 32, 34, 36, 37, 64–68, 70

ISR Immutable Sequential Randomized Construction 32, 34, 38, 65–70

IT Improve Trip 51, 54, 57, 83–87, 96, 97, 99, 108

LA late acceptance 12, 13

LSP Lock Scheduling Problem 2–4, 12, 13, 18, 19, 21

MCD Mutable Chronological Deterministic Construction 38, 40, 42, 51, 70, 73, 74, 76, 77, 83, 86

MCN multi-commodity network 17, 18

MCR Mutable Chronological Randomized Construction 42, 74–77

MIP Mixed Integer Programming 5, 17, 24, 25, 61–63

MMFO modified moth-flame optimization 15

MNS Multiple Neighbourhood Search 12

MSD Mutable Sequential Deterministic Construction 36–38, 40, 42, 68–75

MSR Mutable Sequential Randomized Construction 38, 70–77, 86

NI next improvement 28, 43, 44, 51, 52, 56, 83, 86

NOK "Nord-Ostsee-Kanal" 15, 16, 19

NSCNI Neighbourhood Search with Configurable Neighbourhood Iteration 23, 56–58, 87, 90, 92, 108

QBGSA quantum inspired binary gravitational search algorithm 15

RCL restricted candidate list 32–34, 42, 65, 70, 74, 76, 77

RebWG Rebuild Water Gate 52–54, 99

RemEmpty Remove Empty Lockages 51, 53, 87, 90, 92, 93, 95, 96

ReSiL Rearrange Ships in Lockage 45, 46, 53, 87, 90, 92–94, 96

RI random improvement 28, 43, 44, 51, 52, 83, 86, 87, 99, 104

RN random neighbor 28

ShL Shift Lockage 49, 50, 53, 87–89, 91, 92, 94

ShS Shift Ship 47–49, 53, 80–83, 86–89, 91, 92

ShSiL Shift Ship in Lockage 45, 46, 53

SPF shortest processing time first 12

STCP Ship Traffic Control Problem 16, 17, 19, 20

SwL Swap Lockages 50, 53, 87–89, 91, 92, 94

SwS Swap Ships 48, 49, 53, 87–89, 91, 92

SwSiL Swap Ships in Lockage 44–46, 53

TGP Three Gorges Project 13–15

TGP-GD Three Gorges Project and Gezhouba Dam 13–15, 18, 21

TRNS Token-Ring Neighborhood Search 56, 57, 87–90, 92–100, 103, 104

UMR Upper Mississippi River 11, 18

VLN very large neighborhood 93, 108

VND Variable Neighborhood Descent 55–57, 90–96, 104

VNS Variable Neighborhood Search 12, 13, 54–57, 87, 90, 96, 99, 103, 104, 108

Symbols

Water Gate Parameters

$\mathcal{G}$	Set of all water gates
$\mathcal{G}_s$	Set of all water gates passed by ship s
κ_g	Capacity of lock chamber at water gate g
τ_g	Constant time for lockage at water gate g
c_g^e	Location of eastern border or the lock chambers at water gate g
c_g^w	Location of western border or the lock chambers at water gate g
a_g^e	Location of eastern border or the surrounding area of water gate g
a_g^w	Location of western border or the surrounding area of water gate g

River Sections Parameters

$\mathcal{R}$	Set of all river sections
$\mathcal{R}_s$	Set of all river sections passed by ship s
r_i	Length of the river section i

Ship Parameters

$\mathcal{S}$	Set of all ships
k_s	Length of ship s
t_s	Type of ship s
v_{s_i}	Travelspeed of a ship s at section i
$e_{s_g}^n$	Time required by ship s to enter water gate g
$e_{s_g}^x$	Time required by ship s to exit water gate g
r_s	Start time of ship s
λ_s	Start postition of ship s
δ_s	Target postition of ship s
at_s	Arrival time of ship s at target postition

Lockage Variables

$\mathcal{L}_g$	Set of all lockages at water gate g
l_{i_g}	Lockage number i at water gate g
z_{i_g}	Timestamp of lockage operation number i at water gate g
S_{i_g}	Set of ships assigned to lockage operation number i at water gate g
$p_{i_{g_s}}$	Position of ship s in lockage operation number i at water gate g

Travel Time Variables and Derived Values

tt_s	Total travel time of ship s
mt_s	Total (constant) in-motion time of ship s
ct_s	Total (constant) time of ship s entering, exiting and waiting for lockage operation to complete
$wt^c_{s_g}$	Time of ship s waiting inside the lockchamber at water gate g
$wt^a_{s_g}$	Time of ship s waiting before entering the area of water gate g
$at^a_{s_g}$	Arrival time of ship s at area of water gate g
ent_{s_g}	Point in time, that ship s starts entering area and chamber of water gate g
ex_{s_g}	Point in time, that ship s starts exiting the chamber of water gate g
dt_{s_g}	Departure time of ship s from area of water gate g

Bibliography

[BSW07] Peter Brucker, Yuri Sotskov, and Frank Werner. Complexity of shop-scheduling problems with fixed number of jobs: A survey. *Mathematical Methods of Operational Research*, 65:461–481, 05 2007.

[CS11] Sofie Coene and Frits C. R. Spieksma. The Lockmaster's problem. In Alberto Caprara and Spyros Kontogiannis, editors, *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 27–37, Dagstuhl, Germany, 2011. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[DGS03] Luca Di Gaspero and Andrea Schaerf. Easylocal++: an object-oriented framework for the flexible design of local-search algorithms. *Software: Practice and Experience*, 33(8):733–765, 2003.

[DHH+13] M. Dolinsek, S. Hartl, T. Hartl, B. Hintergräber, V. Hofbauer, M. Hrusovsky, G. Maierbrugger, B. Matzner, L.-M. Putz, M. Sattler, J. Schweighofer, L. Seemann, M. Simoner, and D. Slavicek. *Handbuch der Donauschifffahrt*. bmvit, 2013.

[DM02] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, Jan 2002.

[Eur10] European Commission. Action Plan accompanying the European Union Strategy for the Danube Region. `http://www.danube-region.eu/component/edocman/action-plan-eusdr-pdf`, 2010.

[Eur11] European Commission. Roadmap to a Single European Traffic Transport Area – Towards a competitive and resource efficient transport system. `http://ec.europa.eu/transport/strategies/doc/2011_white_paper/white_paper_com(2011)_144_en.pdf`, 2011.

[Eur20] European Commission. Action Plan accompanying the European Union Strategy for the Danube Region. `https://ec.europa.eu/regional_policy/sources/cooperate/danube/eusdr_actionplan_swd202059_en.pdf`, 2020.

[HMMP10] Pierre Hansen, Nenad Mladenovic, and José Moreno-Pérez. Variable neighbourhood search: Methods and applications. *4OR*, 175:367–407, 02 2010.

[HS87] J.Pirie Hart and Andrew W. Shogan. Semi-greedy heuristics: An empirical study. *Operations Research Letters*, 6(3):107–114, 1987.

[JYY19] B. Ji, X. Yuan, and Y. Yuan. A hybrid intelligent approach for co-scheduling of cascaded locks with multiple chambers. *IEEE Transactions on Cybernetics*, 49(4):1236–1248, April 2019.

[JZYZ21] Bin Ji, Dezhi Zhang, Samson S. Yu, and Binqiao Zhang. Optimally solving the generalized serial-lock scheduling problem from a graph-theory-based multi-commodity network perspective. *European Journal of Operational Research*, 288(1):47 – 62, 2021.

[LLM19] Elisabeth Lübbecke, Marco E. Lübbecke, and Rolf H. Möhring. Ship traffic optimization for the kiel canal. *Operations Research*, 67(3):791–812, 2019.

[Luy11] Martin Luy. *Algorithmen zum Scheduling von Schleusungsvorgängen am Beispiel des Nord-Ostsee-Kanals*. Diplomica Verlag, Hamburg, 2011.

[Lü15] Elisabeth Lübbecke. *On- and Offline Scheduling of Bidirectional Traffic*. PhD thesis, TU Berlin, 2015.

[MAK07] Wil Michiels, Emile Aarts, and Jan Korst. *Theoretical Aspects of Local Search*. 01 2007.

[MF19] Frank Meisel and Kjetil Fagerholt. Scheduling two-way ship traffic for the kiel canal: Model, extensions and a matheuristic. *Computers & Operations Research*, 106:119 – 132, 2019.

[MH97] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[MS95] D. Martinelli and P. Schonfeld. Approximating delays at interdependent locks. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 121(6):300–307, 1995.

[Pas16] Ward Passchyn. *Scheduling locks on inland waterways.* PhD thesis, KU Leuven, 2016.

[PRSR15] Matthias Prandtstetter, Ulrike Ritzinger, Peter Schmidt, and Mario Ruthmair. A variable neighborhood search approach for the interdependent lock scheduling problem. In Gabriela Ochoa and Francisco Chicano, editors, *Evolutionary Computation in Combinatorial Optimization*, pages 36–47, Cham, 2015. Springer International Publishing.

[Rav02] Ravindra K. Ahuja and Özlem Ergun and James B. Orlin and Abraham P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1):75–102, 2002.

[RRP15] Mario Ruthmair, Ulrike Ritzinger, and Matthias Prandtstetter. Mixed integer linear programming models for the interdependent lock scheduling problem, 2015. *ISMP 2015 International Symposium on Mathematical Programming*, Pittsburgh, 12^{th}-17^{th} July 2015.

[SL68] Joseph S. De Salvo and Lester B. Lave. An analysis of towboat delays. *Journal of Transport Economics and Policy*, 2(2):pp. 232–241, 1968.

[SSSWD00] Gerhard Schrimpf, Johannes Schneider, Hermann Stamm-Wilbrandt, and Gunter Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159:139–171, 04 2000.

[SV08] Robert Stahlbock and Stefan Voß. Operations research at container terminals: a literature update. *OR Spectrum*, 30(1):1–52, 2008.

[TS98] C. Ting and P. Schonfeld. Integrated control for series of waterway locks. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 124(4):199–206, 1998.

[TS99] C. Ting and P. Schonfeld. Effects of speed control on tow travel costs. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 125(4):203–206, 1999.

[TS01] C. Ting and P. Schonfeld. Efficiency versus fairness in priority control: Waterway lock case. *Journal of Waterway, Port, Coastal, and Ocean Engineering*, 127(2):82–88, 2001.

[VB09] Jannes Verstichel and GreetVanden Berghe. A late acceptance algorithm for the lock scheduling problem. In Stefan Voß, Julia Pahl, and Silvia Schwarze, editors, *Logistik Management*, pages 457–478. Physica-Verlag HD, 2009.

[VCLS17] Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. A visual web tool to perform what-if analysis of optimization approaches. *arXiv preprint arXiv:1703.06042*, 2017.

[Ver13] Jannes Verstichel. *The lock scheduling problem*. PhD thesis, KU Leuven – Faculty of Engineering Science, nov 2013.

[via20] via donau – Österreichische Wasserstraßen-Gesellschaft mbH. Jahresbericht Donauschifffahrt in Österreich 2019. `https://www.viadonau.org/fileadmin/user_upload/Jahresbericht_Donauschifffahrt_2019.pdf`, 2020.

[WR09] Xiaoping Wang and Qian Ruan. Genetic algorithm and tabu search hybrid algorithm to co-scheduling model of three gorges-gezhou dam. In Wen Yu, Haibo He, and Nian Zhang, editors, *Advances in Neural Networks – ISNN 2009*, volume 5552 of *Lecture Notes in Computer Science*, pages 581–590. Springer Berlin Heidelberg, 2009.

[ZFY08] Xiaopan Zhang, Xide Fu, and Xiaohui Yuan. Co-evolutionary strategy algorithm to the lockage scheduling of the three gorges project. In *Computational Intelligence and Industrial Application, 2008. PACIIA '08. Pacific-Asia Workshop on*, volume 2, pages 583–587, 2008.

[ZYY08] Xiaopan Zhang, Xiaohui Yuan, and Yanbin Yuan. Improved hybrid simulated annealing algorithm for navigation scheduling for the two dams of the three gorges project. *Computers & Mathematics with Applications*, 56(1):151 – 159, 2008.