FAKULTÄT FÜR !NFORMATIK

Compressing Fingerprint Templates by Solving the k-Node Minimum Label Spanning Arborescence Problem by Branch-and-Price

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering / Internet Computing (066/937)

eingereicht von

Corinna Thöni, BSc.
Matrikelnummer 9826437

an der
Fakultät für Informatik der Technischen Universität Wien
Institut für Computergrafik und Algorithmen

Betreuung:
Univ.-Prof. Dipl.-Ing. Dr. techn. Günther Raidl
Univ.Ass. Mag. Dipl.-Ing. Andreas Chwatal

Wien, 05.02.2010

(Unterschrift Verfasserin) (Unterschrift Betreuer)

FAKULTÄT FÜR !NFORMATIK

Compressing Fingerprint Templates by Solving the k-Node Minimum Label Spanning Arborescence Problem by Branch-and-Price

MASTER'S THESIS

written at the
Institute for Computer Graphics and Algorithms
Vienna University of Technology

under supervision of
Univ.-Prof. Dipl.-Ing. Dr. techn. Günther Raidl
Univ.Ass. Mag. Dipl.-Ing. Andreas Chwatal

by

Corinna Thöni, BSc.
Software Engineering / Internet Computing (066/937), 9826437
Vienna, 05.02.2010

Abstract

This thesis deals with the development of exact algorithms based on mathematical programming techniques for the solution of a combinatorial optimization problem which emerged in the context of a new compression model recently developed at the *Algorithms and Data Structures Group* of the *Institute of Computer Graphics and Algorithms* at the Vienna University of Technology. This compression model is particularly suited for small sets of unordered and multidimensional data having its application background in the field of biometrics. More precisely, fingerprint data given as a set of characteristic points and associated properties should be compressed in order to be used as an additional security feature for e.g. passports by embedding the data into passport images by watermarking techniques.

The considered model is based on the well known *Minimum Label Spanning Tree Problem (MLST)* with the objective to find a small set of *labels*, associated to the edges of the graph, inducing a valid spanning tree. Based on this solution an encoding of the considered points being more compact than their trivial representation can be derived. So far, heuristic and exact algorithms have been developed, all of them requiring a time-consuming preprocessing step. The goal of this work is to develop an improved exact algorithm carrying out the tasks of the preprocessing algorithm during the execution of the exact methods in an profitable way.

Within this thesis the problem is solved by mixed integer programming techniques. For this purpose a new *flow formulation* is presented which can be directly solved by linear programming based branch-and-bound, or alternatively by *branch-and-price*, being the main approach of this thesis. Branch-and-price works by starting with a *restricted model*, and then iteratively adding new variables potentially improving the objective function value. These variables are determined within the *pricing problem* which has to be solved frequently during the overall solution process. For this purpose specialized data structures and algorithms, based on a k-d tree are used, with the intention to exploit possible structures of the input data. Within this tree, improving variables can be found by various traversing and bounding techniques. These algorithms can also be adapted in order to work as an alternative to the existing preprocessing.

All algorithms have been implemented and comprehensively tested. For the existing approaches, the new methods reduced the preprocessing run times with a factor of 50 and use now 2% of the original run times. With the presented branch-and-price approach it is possible to solve a greater amount of test instances to optimality than previously. Generally, for most model parameters the branch-and-price approach outperforms the previous exact method.

Zusammenfassung

Diese Diplomarbeit beschäftigt sich mit der Entwicklung von exakten Algorithmen, welche auf mathematischer Programmierung basieren. Ziel ist die Lösung eines kombinatorischen Optimierungsproblems, welches im Kontext eines neuen Komprimierungsverfahren auftritt, das kürzlich vom *Arbeitsbereich für Algorithmen und Datenstrukturen* des *Institut für Computergraphik und Algorithmen* an der Technischen Universität Wien entwickelt wurde. Dieses Komprimierungsverfahren ist insbesondere geeignet für kleine Mengen von ungeordneten und multidimensionalen Daten, wobei biometrische Applikationen den Anwendungshintergrund bilden. Insbesondere sollen Fingerabdruckdaten, sogenannte Minutien, gegeben als Menge charakteristischer Punkte mit zugeordneten Eigenschaften, komprimiert werden, um als zusätzliches Sicherheitsmerkmal für z.B. Reisepässe verwendet werden zu können, indem die Daten als Wasserzeichen in das Passbild eingebettet werden.

Das betrachtete Modell basiert auf dem bekannten *Minimum Label Spanning Tree Problem (MLST)*. Das Ziel des Problems ist es, eine kleine Menge an *Labels* zu finden, welche den Kanten des Graphen zugeordnet sind und so einen gültigen Spannbaum induzieren. Basierend auf dieser Lösung kann eine Kodierung der betrachteten Punkte abgeleitet werden, welche kompakter als ihre triviale Darstellung ist. Bislang wurden Heuristiken und ein exaktes Verfahren entwickelt, welche alle einen laufzeitintensiven Preprocessing-Schritt voraussetzen. Das Ziel dieser Arbeit ist es, einen verbesserten exakten Algorithmus zu entwickeln, welcher die Aufgaben des Preprocessing-Schritts in einer vorteilhaften Weise während der Ausführung der exakten Methode erledigt.

In dieser Arbeit wurde das Problem mit Mixed Integer Programmierungstechniken gelöst. Zu diesem Zweck wird eine neue *Flußnetzwerk Formulierung* vorgestellt, welche direkt mittels Branch-and-Bound, basierend auf Linearer Programmierung, gelöst werden kann. Alternativ dazu gelingt die Lösung auch mittels *Branch-and-Price*, welches den Hauptansatz darstellt. Branch-and-Price beginnt mit einem *reduzierten Problem* und fügt dann schrittweise neue Variablen, welche den Zielfunktionswert verbessern können, diesem reduzierten Problem hinzu. Diese Variablen werden mittels des *Pricing Problems* bestimmt, welches oftmals während des Lösungsvorgangs berechnet werden muss. Zu diesem Zweck wurden spezialisierte, auf einem k-d Tree basierende Datenstrukturen und Algorithmen entwickelt, welche mögliche Strukturen der Eingabedaten in geschickter Weise ausnützen. Mittels verschiedener Traversierungs- und Boundingtechniken können aus dieser Baumdatenstruktur verbessernde Variablen effizient extrahiert werden. Weiters können die entwickelten Algorithmen zu einer Alternative für den Preprocessing-Schritt adaptiert werden.

Alle Algorithmen wurden implementiert und umfangreich getestet. Für die bestehenden Ansätze wurde die Zeit, die für den ursprünglichen Vorberechnungsschritt gebraucht wurde, um Faktor 50 reduziert, die Laufzeiten sinken auf 2% der ursprünglichen Zeit. Mit dem vorgestellten Branch-and-Price Verfahren ist es möglich eine größere Anzahl an Testinstanzen optimal zu lösen als bisher. Für die meisten Modellparameter übertrifft der Branch-and-Price Ansatz die bisherige exakte Methode.

Acknowledgements

Ich möchte allen Personen danken, die mir den Abschluß dieser Diplomarbeit ermöglicht haben. Ich danke meinen Eltern Robert und Johanna sowie meinem Bruder Rudolf, die viel Geduld und Verständnis für mich aufgebracht und mich immer wieder zum Weitermachen motiviert haben. Ich danke Dieter, der mich in Allem unterstützt und bekräftigt hat.
Danke an Andreas Chwatal, dessen exzellente fachliche Kompetenz und unendliche Geduld diese Arbeit möglich gemacht haben. Danke an Günther Raidl für die großartigen Ideen und die viele Unterstützung. Danke an den gesamten Fachbereich für Algorithmen und Datenstrukturen für die gute technische Infrastruktur.

Erklärung zur Verfassung

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, 05.02.2010

Contents

Chapter 1

Introduction

The main goal of this diploma thesis is the compression of fingerprint data in such a way that additional encoded information can be stored in images by watermarking techniques. Application background is the use of such watermarks on identification cards and passports, to enhance the security of the stored images (for example a fingerprint image) by embedding additional compressed information. In the case of identification cards and passports, only a very limited amount of storage is at disposal for this task, which implies the need of a compact representation and a good compression mechanism, that is capable of compressing as much data as much as possible in a reasonable amount of time, without loosing too much precision.

Various methods for compressing fingerprint data were analysed in the course of a research project at the *Institute of Computer Graphics and Algorithms (Algorithms and Data Structures Group)* at the Vienna University of Technology. The performance and compression ratios of the approaches have been evaluated. While investigating common standard compression techniques the view fell upon dictionary based techniques. Thus, a special tree based compression model has been developed in order to extract a dictionary from the fingerprint data and represent the features of a fingerprint through this dictionary. To name a few details, the extraction of the dictionary revealed itself to be accomplished through various approaches, which again are classified into two main categories: algorithms that find approximate solutions for the dictionary and algorithms providing optimal dictionaries. Normally the former ones perform very fast but the quality of the result might not be sufficiently high, whereas the latter supply optimal results but require significant computational effort. Amongst approximating algorithms heuristics and memetic algorithms have been evaluated. While developing an exact algorithm (e.g. branch-and-cut) this class of problem solving approaches was found to have great potential. This thesis pursues the ideas and modelling approaches developed in [ChwRai09, Dietzel08]. In particular elaborate combinatorial optimization techniques are investigated and applied to the given problem. The main contribution is the development of a branch-and-price approach, and algorithms for solving the corresponding pricing problem.

The following sections give an overview of the application background where the problem emerges, which approaches were already examined and how the task is approached.

The document is segmented into two main parts: The first part includes a short introduction to the topic (chapter 1) and covers the underlying theory. Chapter 2 is concerned with graph theory and geometric algorithms, whereas chapter 3 covers linear and integer optimization and introduces the central topics column generation and branch-and-price. The following chapter 4 contains a detailed formal problem definition. The second part is concerned with specific algorithms and their evaluation: It is structured into chapter 5, which explains the branch-and-price approach including the formulation as a flow network, and chapter 6, which describes developed concepts and algorithms for solving the arising pricing problem. Chapter 7 introduces an alternative for the

intermediate step, called preprocessing, needed in all previously developed solution approaches. It is followed by chapter 8, where the implementation is outlined as well as architectural details, used programming language and the framework integrated for the solution of our problem. The chapter 9 lists test methods as well as experimental results, and is followed by the chapter 10, where final conclusions are drawn. The appendix summarizes some often used terms and naming conventions and includes additional tables for experimental results.

1.1 Biometric Background

The central problem of this thesis comes from the field of biometric data processing. The term *biometrics* (Greek: *Bio* life, *Metron* measure) collects many procedures of measuring and analyzing humans based on one or more physical and behavioral traits. Of the many fields of which biometrics is comprised in this diploma thesis the procedures of biometric recognition processing are of interest. The origins of biometrics are located in forensic sciences, dactyloscopy and cryptography. In the past decades, with the progression of information technology, the interest and distribution of biometric applications ever increased, since cataloguing and analyzing huge amounts of data became only possible with recent technology.

Biometric recognition bases upon several traits and characteristics of the human body and behaviour, named *biometric identifiers*. According to [MalJai05] these identifiers are categorized into physiological (fingerprints, iris, retina, vene structure, DNA, teeth alignment, hand, palm and face geometry) and behavioral (written signature, gait, voice, typing rhythm). In [Jain08], the author describes several properties that such a trait must suffice, in order to be used as a biometric identifier:

- *Universality*: The characteristic is found on every individual to be verified.
- *Uniqueness*: The characteristic is sufficiently different across the individuals.
- *Permanence*: The characteristic remains constant over time, and depends not on age or measure datum.
- *Measurability*: The characteristic is acquirable with a suited device, without incommoding the providing person. Digitalizing and further processing must be possible.
- *Performance*: Accuracy and resource consumption for acquiring and processing should have acceptable time, computation and hardware effort.
- *Acceptability*: Persons should be willing to present their biometric trait to the system.
- *Circumvention*: The characteristic should not be easily imitated, reproduced or acquired.

One main application for biometric recognition is the authentication of a person, for which following scenarios exist (from [Jain08]):

- *Verification*: A person claims an identity through a user name, key or PIN and provides the appropriate biometric trait for the system. The system validates the person by matching the captured biometric data with the samples stored in its database in a one-to-one comparison.
- *Identification*: A person provides only the biometric trait(s) for the system, for which then the system searches in the database of templates in order to find a possible match. The comparison is one-to-many and has the advantage that the person does not have to claim an identity.

Verification is accompanied by ID-cards, tokens, keys, passwords and user names, whereas identification is only established with biometrics. It lies at hand that biometric applications prefer stable, unique, physiological characteristics for their recognition process.

When realizing such a biometric recognition system, several components are necessary: First a *sensor* produces a digital sample image for further processing. The sensor depends on the selected characteristic and ranges from a simple video camera or recorder to a highly sophisticated eye scanner. Specialized algorithms extract the appropriate traits and construct an internal model.

Then a comparison algorithm matches the extracted traits with templates from a database and returns a score for the match of a template and the sample.

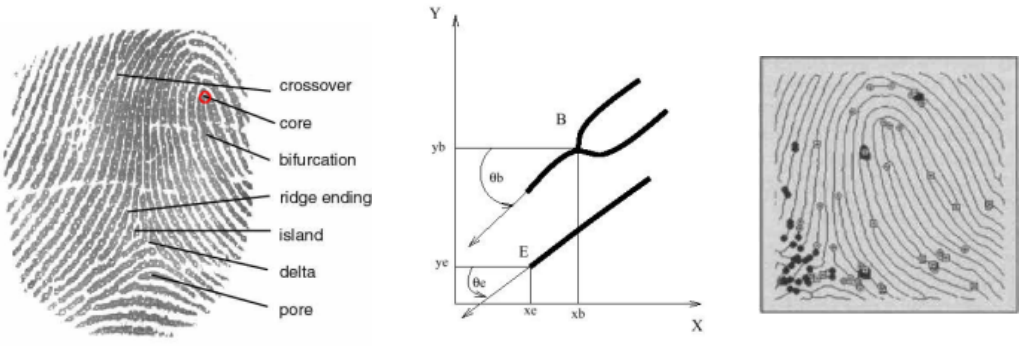
Biometric traits are used ever increasingly in governmental, forensic and commercial applications. In [Jain08], the authors give an overview of some of the uses:

- *Government*: National ID card, drivers licence, voter registration, welfare disbursement, border crossing, asylum seekers, and more.
- *Forensics*: corpse identification, criminal investigation, parenthood determination, missing children, and other applications from this field.
- *Commercial*: ATM, access control of all kinds (security doors, season tickets, casino visitors), computer login, mobile phone, e-commerce, internet banking, smart cards, and many more.

The basic subject of this thesis are fingerprints, which will be regarded more thoroughly in the next section. Fingerprints suffice all properties listed above [Jain08]. They are efficiently collected with affordable hardware requirements, easily digitalised and processable and do not require huge storage. Fingerprints are easily "at hand", unique, and in the use do not incommode users.

1.1.1 Dactyloscopy

Dactyloscopy, the science of analysing fingerprints, has enormous popularity in forensic applications. With the advent of computerization automated fingerprint cataloguing, identification and processing became well-investigated and many industrial applications, such as security systems, started to use and research the uniqueness, portability, simplicity of use of biometric data.



(a) Minutiae types [BasSchu05]. (b) Minutiae schema [webWong]. (c) Extracted minutiae [Jain08].

Figure 1.1: Fingerprint minutiae in dactyloscopy.

Fingerprints are catalogued with various methods. Of these, applications broadly use the *minutiae representation*, which is a schematic description of special features in a fingerprint like bifurcations, whorls, crossovers, and many others, as listed in figure 1.1a. Such minutiae are extracted out of a scanned fingerprint image by specialized algorithms. A minutiae data set consists of usually 15-80 minutiae per fingerprint. A formal representation of one minutia consists of [BasSchu05]:

- *Type* of the minutia: crossover, whorl, bifurcation, ridge ending, and others (see figure 1.1a).
- *Position* (x, y) coordinates based on an imaginary zero point (see figure 1.1b).
- *Local orientation* θ of the minutia.
- *Reliability* r_θ of orientation data[1].

[1]This datum is only used in representations extracting ridge orientation information. The value is the higher if the orientation is similar to the orientation of marks in the vicinity.

The processing of most biometric applications is based upon this minutiae representation. The minutiae data itself is extracted and compared efficiently and has many uses. Of these the most common are database searches for matching fingerprints, or in general person identification, such as in crime reconnaissance, person tracking, or access limitation with security door scanners. Further applications can be found in cryptographic context, where fingerprints are used as additional identification feature on important documents of various sorts, to secure and guarantee provenance of the documents themselves and to impede theft, falsification and prevent abuse. Since nowadays many documents provide facilities for digital storage, like microchips, magnet strips and so on, there are many concerns to enhance the security of the stored data. For this purpose, besides encryption watermarking is a commonly used technique. Since the considered devices have strong storage limitations, the interest for compression comes now into play.

Furthermore it has to be mentioned, that scanning and decomposing fingerprints is not the task of the fingerprint minutiae compression project and can be performed by highly specialized algorithms. Starting point for the work are data sets of already extracted minutiae which are then further processed for the intended use as a watermark. Digital watermarks and compression basics are presented in the following section.

1.1.2 Digital Watermarking, Steganography and Compression

Watermarking is a technique old as humankind. With the advent of information technology the concept of embedding information about provenience and ownership in digital media was adapted. A *digital watermark* is additional information interwoven in digital media files (audio, video, images, text, database records, geodata) in such a way that media and watermark information cannot be separated again. According to [Fraun08] digital watermarks are independent from data format and remain intact after modification of the medium, for example format conversion, compression, scaling and clipping, since the watermark is not embedded into some data format dependent meta header but into the data itself. Also the technique discerns from copy protection mechanisms and encryption methods since copying and displaying of content is not impeded. For that reason watermarks are called *non restrictive*.

Besides *visible* watermarks, which mostly consist of company logos and author names, there also exist *invisible* watermarks, whereby the main aim is to maintain data integrity. After signing, the original content appears not to be altered and enables the following security objectives [Fraun08]:

- Copyright protection for creators, authors and ownership holders. Unambiguous identification of media through an ID, for example the ISBN embedded as watermark in an e-book.
- Integrity protection against manipulation and alteration of content. Tracking of data flow through transaction numbers and purchaser ID, for example color laser printouts.
- Authenticity protection for definite identification of the origin (cryptographic applications).
- Other objectives are: marketing and advertising, additional services as lyrics in audio files, partial encryption for preventing previews.

Watermarks can be engineered in such a way that any alteration of the signed content destroys or damages the signature and the manipulation can be reconstructed. These techniques relate to steganography, the science of concealing information in content. Key features of digital watermarks are perceptibility, transparency, robustness against alterations, capacity (the amount of information to be embedded), security (manipulation of the key and watermark renders useless the original data), performance of embedding and retrieving (reading/playback in real time).
Steganographic embedding of information is technically realized by replacing bits or greater sections of the carrier medium, mostly irrelevant or redundant, with some value calculated out of the information part. Methods altering single bits at appropriate locations are *substitution methods* whereas greater portion altering methods perform *domain transformation*, with the result that the embedded parts are more robust against manipulation and alteration, but require more space.

In the course of the research project on fingerprint minutiae compression a very sophisticated mechanism in the fashion of domain transformation was developed, which also suffices the storage requirements of small devices. Since our storage capabilities are limited, *data compression* is applied. Data compression is the process of replacing original data with a representation of the data using fewer storage consumption on a digital medium. Compression mechanisms are divided into *lossless* and *lossy*, whereby the former ones make it possible to reconstruct the original data from the compressed data, whereas the latter do not permit this and information is lost after the compression process. In the present case a lossless compression is main target.

The following section summarizes the precedent work for minutiae compression. This section gives an overview about the compression model as well as problem solution strategies which have been already investigated in the course of this project.

1.2 Previous Work

In the course of the research project [ChwRai09] and [Dietzel08] analyzed quite a number of conventional standard compression algorithms for their performance and efficiency when set at work to compress minutiae data. A comprehensive introduction to data compression can be found in [Sayood06].

Amongst the examined algorithms is *arithmetic coding* and *Huffman coding*. Further on dictionary based algorithms originating from image, audio and video compression were tested. These algorithms identify redundant structures and condense them into a dictionary, in order to express information to be encoded through this dictionary. Mostly these standard algorithms perform very badly and were found not to be suited for the task. Investigation of the steganographic background and compression technologies in relation to fingerprint minutiae compression are described in [ChwRai09, Dietzel08]. Here their results are not presented further, and can be looked up in the named documents.

1.2.1 The Compression Model

The dictionary based coder approach named above is then pursued in the following way: Having in mind the construction of a minimal dictionary or *codebook* for a compact representation of minutiae information, the naturally contained structures and redundancies in minutiae data led to the development of a *graph based encoding model* as a basis for the compression mechanism. Out of this graph model the compression algorithm extracts the dictionary in form of so called *template vectors* and afterward expresses the minutiae from the input data by a reference to the appropriate codebook entry. The codebook or dictionary is schematized in figure 1.2. The authors of [ChwRai09] emphasize that "our approach follows this general idea of using a dictionary or codebook but its determination as well as its usage is very different to the existing methods."

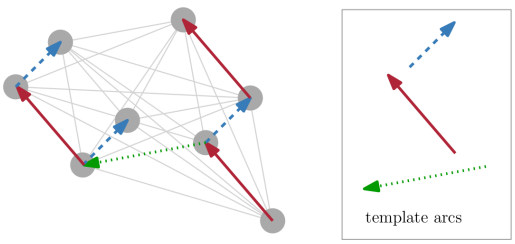


Figure 1.2: Encoding of points via a directed spanning tree using a codebook of template arcs, correction vectors are neglected. Image credits to [ChwRai09].

Presumed we have extracted fingerprint minutiae from a fingertip scan, we take now a closer look at these minutiae. As described in section 1.1.1 a minutia can be seen as a four dimensional vector consisting of position as x, y coordinates, type t and orientation θ. Let us denote such

a minutiae vector. Based on the set of all of their vector differences a minimal set of all possible dictionary entries can be derived in a preprocessing step. Compression is then achieved by determining a minimum cardinality subset that is sufficient for encoding k points. For matching purposes it is often sufficient to consider only 12–20 minutiae [SalAdh01], which has also been confirmed in [Dietzel08]. Based on the dictionary entries, which we call *template arcs*, the points can be encoded as a directed spanning tree (arborescence).

This arborescence is modeled in the fashion of a *k-Node Minimum Label Spanning Arborescence (k-MLSA)*, a variant of the well known *Minimum Label Spanning Tree (MLST) Problem* introduced by [ChLe97]. Both concepts will be referred to in chapter 2, section 2.1.1.
With the aid of this dictionary the minutiae data points are encoded in order to express a compact representation of the input: One arc id from the dictionary per point is used to express the approximate spatial location of the point to encode, each template arc accompanied by a correction vector that codes the deviation of the actual encoded point to the template arc endpoint. Figure 1.3 shows this concept.

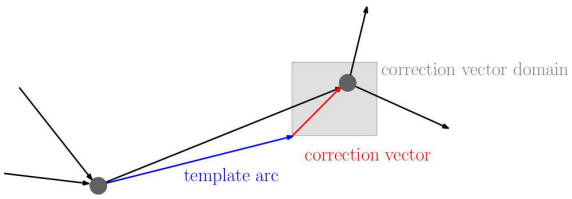


Figure 1.3: Outline of correction vector encoding. Image credits to [ChwRai09].

The complete formal description of the graph based compression model was already developed in [ChwMIC07, RaiChw07, ChwRai09] and will be presented in the problem definition chapter 4.

1.2.2 Determining the Codebook

Various algorithmic strategies, both exact and heuristic, have been developed so far. One exact approach is the *branch-and-cut* algorithm, a technique coming from the field of integer optimization, that is able to identify the optimal solution for the dictionary. The heuristic approaches include a *memetic algorithm* and a *greedy randomized adaptive search procedure (GRASP)*. The algorithms are summarized from [ChwRai09, Dietzel08]:

- **Heuristic Approaches:**
  - **MVCA Based Construction Heuristic**. In [ChLe97], the authors present a heuristic for solving minimum label spanning tree problems, called *Maximum Vertex Covering Algorithm (MVCA)*. Since our underlying problem is a variant of the MLST, the development of such a MVCA based greedy construction heuristic was at hand. According to [ChwRai09, Dietzel08], the adapted heuristic performs the following steps: The codebook is build from scratch, by starting with an empty tree and subsequently adding new template arcs from a candidate template arcs set in a greedy manner. The algorithm proceeds until a feasible k-node arborescence is found. This condition has to be verified often and is done by classical *Depth First Search (DFS)*. The presented heuristic performs fast, but the results are moderate.
  - **Greedy Randomized Adaptive Search Procedure (GRASP)**. This approach extends the heuristic presented above with a better template arc choosing mechanism. By starting again with an empty tree, the template arcs are now added based on a restricted candidate list (RCL). If at some point the solution is valid, local search improves it, if possible. The documents [ChwRai09, Dietzel08] describe approach, performance and test results in detail, especially the crucial task of computing a meaningful RCL.

- **Memetic Algorithm (MA)**. These algorithms come from the field of genetic algorithms and are a combination of evolutionary, population based algorithms and local improvement strategies like individual learning. Like heuristics, this type of algorithms also yield approximate solutions, but tend not to get stuck in local optima. Following the concepts presented by [XiGolWas05] a memetic algorithm for the k-MLSA was developed and presented in [ChwRai09]. According to the evolutionary background the algorithm produces an initial population of feasible solutions, which is then modified by selection, recombination and mutation - imitating biological evolution. An iteration produces an offspring solution from recombined parent solutions, and performs tournament selection and local improvement steps. Redundant arcs in the arborescence must be handled. After a huge, predefined number of iterations or a fixed number of steps in which no improvement is achieved, the best solution is chosen from the population. The document [ChwRai09] shows detailed results, i.e. good solutions are achieved very quickly for the most of the test cases.

- **Branch-and-Cut Approach**. Following a very different approach from the field of combinatorial optimization, an optimal solution for the k-MLSA was also attempted. The authors of [ChwRai09] modelled and formulated the problem as an *integer linear program (ILP)* based on cycle elimination and connectivity inequalities. Shortly outlined, since the theoretical background will be presented later in chapter 3, the branch-and-cut method begins with an incomplete model and derives the solution by iteratively adding further constraints on demand.

Main drawback of these methods is the need for an intermediate step. All solution procedures for the k-MLSA problem employ a *preprocessing step*, a precalculation of a candidate template arcs set, out of which then the actual minimal template codebook is determined by the actual algorithm.

Having now summarized all previous work, it is now time to come to the actual main subject of this diploma thesis. The task is now to follow on the exact problem solution of the k-MLSA with further techniques coming from the field of integer programming.

1.3 Contributions of this Thesis

This section gives an overview of the attempted task. Some terms and concepts are used straightforwardly and will be explained more in detail in the theory chapters 2 and 3.

As named several times, fingerprint minutiae compression is now attempted by integer linear programming. The main disadvantage of the presented approaches was already marked, it lies in the *preprocessing step*. This step precalculates the set of candidate template arcs, called T^c. Out of this set, in all previously presented strategies, the solution to the k-MLSA, the actual minimal template codebook is determined. This preprocessing step is imminent for all presented solution strategies. At the present time this step is realized by restricted enumeration. As the name implies, enumeration examines the whole multitude of possible candidate template arcs set permutations. Although the search was restricted somewhat to promising areas, the number of potential sets to examine is still huge. In practice, the preprocessing step consumes a relatively big amount of processing time, particularly for large input parameters[2] $\tilde{\delta}$.

In the following this weak point shall be removed: Consequently our goal is to skip this preprocessing step and incorporate it into the linear program part. To do so, the focus shifts from the tree to the candidate template arcs themselves as central point. Their number in practice is again very huge.

[2]This parameter will be addressed in the problem definition chapter 4.

The idea is now not to generate all possible candidate template arcs in advance, and then solve the resulting (large) integer linear program, but to start with a very small set of candidate template arcs, and then create new template arcs on demand during the solution of the ILP. Such approaches are called column generation, as new variables (and therefore columns in the coefficient matrix of the integer linear program) are generated continuously. Combined with branch-and-bound we obtain an approach called *branch-and-price*.

To realize this, the original formulation from the branch-and-cut approach must be altered and adapted accordingly. Since the original formulation has an exponential number of involved restrictions, we need a formulation consisting of a moderate amount of restrictions, but having instead a huge amount of variables. This task is realized by formulating the integer program in terms of a *single* and a *multi commodity flow network*. The resulting mixed integer program is then solved by branch-and-price. In other words the creation of candidate template arcs is incorporated into the integer program itself by creating new template arcs on demand.

So, the first step is to reformulate the original integer linear program so that it can be solved by our chosen approach. We have to set up the mixed integer program and solve it by branch-and-price. For the solution of huge sized integer programs there exist a lot of commercial and non-commercial frameworks, one of which will be selected for this task. The chosen framework will be described in the implementation chapter 8.

Branch-and-price starts with a small set of template arcs and then iteratively adds new template arcs potentially improving the objective function value. These variables are determined within a special step, the *pricing problem*. The solution of the pricing problem depends on intermediate solution values of our newly formulated integer program. Also some special structures of the input data are exploited.

In the course of the problem analysis a very promising approach to solve the pricing problem was developed, and realized with the aid of a very common and efficient geometric algorithm, a *k-d tree*. The development of efficient data structures and corresponding algorithms based on such a k-d tree is the second important contribution of this thesis. Here, efficiency is very crucial, as the pricing problem needs to be solved numerous times.

To solve the overall problem, the pricing problem solver is finally integrated in the branch-and-price framework, which then is tested thoroughly for its overall performance, speed and correctness.

Outlook

The following two chapters introduce theory prerequisites for the understanding of the problem definition and solution approach. When we look at the tasks, we identify two main theory subjects: Since the underlying compression model is a graph and the MLST solution bases on graph theoretical ideas, an overview of these graph theoretical basics as well as flow networks must be accounted (section 2.1). Also the solution of the pricing problem requires methods from its subarea of algorithmic geometry, the most important being the well known k-d tree, addressed in section 2.3. Furthermore, to complete the branch-and-price part, a deeper understanding of optimization theory is needed. Since this area is very vast, the chapter 3 concerning theory from this field focuses on linear and integer programming basics, but emphasizes the subjects column generation and branch-and-price, which are regarded more thoroughly in section 3.4.

Chapter 2

Graph Theory and Algorithmic Geometry Essentials

This chapter is dedicated to shortly outline and recapitulate the most important graph theory details and geometric algorithms, which are needed for the problem definition and solution, subject in chapters 4 until 7. Here the basics utilized for the construction of the minimum label spanning tree based compression model are summarized. Network flow problems are basis for the formulation of the mixed integer program in order to be solved by branch-and-price. Moreover the background of the solution strategy used in the arising pricing problem is regarded: The used k-d tree has its roots in binary search trees, which have a widespread application spectrum and come from the field of algorithmic geometry. How all these dissimilar topics are finally tied together will become clear in chapter 5.

Most graph theory concepts are summarized from the reference books [Sedgew01, CormLei07], which are recommended for further inquiries. Reference for multi-dimensional binary search trees (commonly known as k-d trees) is [Bentley75]. Sources for minimum label spanning trees are [ChLe97], [XiGolWas05] and [Kru98].

2.1 Graph Theory Basics

A *graph* is a tuple $G = (V, E)$ and consists of a set of *vertices* or nodes $V = \{v_1, v_2, \ldots, v_n\}$, $n \in \mathbb{N}$ and a set of *edges* $E = \{e_1, e_2, \ldots, e_m\}$, $m \in \mathbb{N}$, which are elements from $V \times V$. An edge e_{ij} connects two nodes v_i and v_j. An edge is *directed* if a direction is endowed, the set of edges then is called $A = \{a_{ij} \mid a_{ij} = (v_i, v_j),\ v_i, v_j \in V\}$. Here $a_{ij} \neq a_{ji}$, with $a_{ij} = (v_i, v_j)$ and $a_{ji} = (v_j, v_i)$. The set of edges in an *undirected* graph is $E = \{e_{ij} \mid e_{ij} = \{v_i, v_j\},\ v_i, v_j \in V \wedge v_i \neq v_j\}$. Here follows $e_{ij} = e_{ji}$. *Undirected graphs* contain no directed edges. *Directed graphs (digraphs)* contain only directed edges. Mixed graphs contain both. Edges e_{ij} in undirected graphs are called *incident* with the nodes v_i and v_j, the nodes v_i and v_j themselves are called *adjacent*. In directed graphs adjacency of v_j to v_i is only implied when an edge a_{ij} exists. A *loop* is an edge e_{ii} that connects the same vertex at its endpoints. *Multi-edged graphs* contain multiple edges connecting the same endpoints, a *simple graph* contains no multi-edges. *Complete* undirected graphs contain $\binom{|V|}{2}$ edges and all nodes $v_i \neq v_j$ with $v_i, v_j \in V$ are adjacent. In undirected graphs for each node v the *degree* $\delta(v)$ is defined as the number of adjacent edges. In directed graphs the *in-degree* $\delta^-(v)$ is the number of incoming edges and the *out-degree* $\delta^+(v)$ the number of outgoing edges. Nodes or edges can be endowed (depending on the modeled problem) with *weights* and thus form a *weighted graph*, with "labels" or "colours", forming a *labeled* or *coloured graph*.

A *path* denotes a p-sized sequence of vertices, such that from each vertex exists an edge to the next vertex in the sequence. Graphs are *connected*, if every point is reachable through a *path*,

else *unconnected*. A *cycle* exists, if the starting point can be reached through a path. A graph containing no cycles is called *acyclic*. A *tree* is a connected, undirected graph containing no cycles, where the removal of an edge renders it unconnected. A tree contains n vertices and $n-1$ edges. Vertices in a tree with $\delta(v) = 1$ are called *leafs*, nodes with greater degree are *intermediate nodes*.

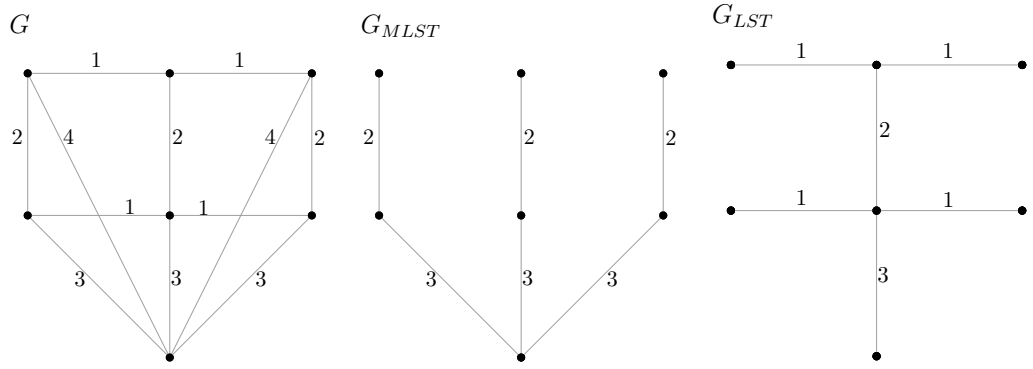
A *subgraph* G' of G has $V' \subseteq V$ and $E' \subseteq E$, the edges in E' connecting only vertices in V'. A *spanning tree* is a subgraph of some graph containing all vertices, but being a tree. An *arborescence* is a spanning tree on a directed graph, where exist (directed) paths from the root node to every other node. A *minimum spanning tree (MST)* is a spanning tree with minimal weight edges. To solve the MST problem Kruskal and Prim developed their eponymous algorithms which nowadays every student learns in her first algorithmic lessons.

2.1.1 Minimum Label Spanning Tree (MLST)

The MLST problem was first introduced by [ChLe97], where the authors also showed it to be NP-hard. Following definition was found in literature:

> **Definition 1** (Minimum Label Spanning Tree Problem). *"Let $G = (V, E)$ be a connected undirected graph and $c : E \longrightarrow \mathbb{N}$ be an edge labeling/coloring function. A K-colored spanning tree (V, T) is a spanning tree of G such that the number of used colors $\mid \{c(e) \mid e \in T\} \mid$ does not exceed K. A minimum label spanning tree is a K-colored spanning tree with minimum K." [Kru98].*

Figure 2.1 depicts examples for (minimum) label spanning trees. Solution approaches for the MLST problem (MVCA heuristic, genetic algorithms, exact algorithms) are described in [ChLe97, Kru98, XiGolWas05, Cons06, XiGolWas06].



(a) Original graph (b) MLST (c) LST

Figure 2.1: Figure 2.1b shows an optimal MLST, determined from the labeled graph in figure 2.1a. Figure 2.1c is a possible *Label Spanning Tree (LST)*. Images adopted from [XiGolWas05].

2.1.2 k-Cardinality Tree (k-CT)

In [ChiKa08], the authors developed a solution strategy for the following NP-hard problem:

> **Definition 2** (k-**Cardinality Tree Problem**). *"Given an undirected graph $G = (V, E)$ with edge weights and a positive integer number k, the k-Cardinality Tree problem consists of finding a subtree T of G with exactly k edges and the minimum possible weight." [ChiKa08].*

The problem was solved by an exact algorithm: After transforming the k-CT problem into a k-cardinality arborescence problem, the formulation as an integer linear program using directed cuts was implemented in a branch-and-cut framework.

2.1.3 k-node Minimum Label Spanning Arborescence (k-MLSA)

By combining the MLST problem and the k-cardinality tree problem, the k-node minimum label spanning arborescence problem was introduced by [RaiChw07]. The k-MLSA is a subset $V' \subseteq V$, which consists of a predefined number $k = |V'|$ of nodes (and therefore $k-1$ edges), which form a spanning tree and has a minimal label set.

2.1.4 Flow networks

A *network* N is a directed graph without multi edges $N = (V, E, s, t, c)$. It has two special nodes, the *source* s and the *sink* or *target* t, $s, t \in V$. Further a *capacity function* c defines for each edge $(u, v) \in E$ a capacity $c_{uv} \geq 0, c \in \mathbb{R}$. The graph is connected, so for every vertex $v \in V$ exists a path $s \rightsquigarrow v \rightsquigarrow t$. An *s-t-flow* is a function f, that defines for every edge in the network a non negative real flow value $f(u, v)$. The flow in a network is constrained as follows [CormLei07]:

- *Capacity Constraint*: The flow over an edge is at most the capacity of the edge:
$$\forall u, v \in V : \ f(u, v) \leq c(u, v).$$
- *Flow Conservation*: Except source and sink, the incoming flow at each node must be equal to the outgoing flow. For a node u, u^+ is the set of nodes connected to u by an outgoing edge, u^- is the set of nodes connected to u by an incoming edge:
$$\forall u \in V - \{s, t\} : \ \sum_{v \in u^+} f(u, v) = \sum_{v \in u^-} f(v, u).$$
- *Skew Symmetry*: $\forall u, v \in V$: $f(u, v) = -f(v, u)$.

Single Commodity Flow Problem (SCF)

This flow network has a *single commodity* flowing through it. A source s and a target t are defined. The commodity $f(u, v)$ flows along edge (u, v) and has a constraining capacity c_i. The flow is conserved by $\sum_{v \in V} f(u, v) = 0$.

Multi Commodity Flow Problem (MCF)

This flow network has *multiple commodities* flowing through it. These κ commodities $k_1, k_2, \ldots, k_\kappa$ can have varying sources and targets and are defined as: $k_i = (s_i, t_i, d_i)$, d_i being some demand. $f_i(u, v)$ is a flow of commodities along edge (u, v). The flow in the network is constrained as follows [CormLei07]:

- *Capacity Constraint*: $\sum_{i=1}^{k} f_i(u, v) \leq c(u, v)$.
- *Flow Conservation*: $\sum_{w \in V} f_i(u, w) = 0$ when $u \neq s_i, t_i$. Follows $\forall v, u : f_i(u, v) = -f_i(v, u)$.
- *Demand Satisfaction*: $\sum_{w \in V} f_i(s_i, w) = \sum_{w \in V} f_i(w, t_i) = d_i$.

If capacities or demands are not restricted, both networks become *uncapacitated commodity flow networks*. For our purposes no capacity is needed. According to [Evans78] under certain conditions multi commodity network flow problems can be transformed into equivalent uncapacitated single commodity flow problems. In 4 we reformulate an integer linear program in terms of an uncapacitated SCF problem as well as an uncapacitated MCF problem.

2.2 Algorithmic Graph Theory Essentials

A tree is a common data structure that organizes data in a set of linked nodes. Searching and inserting in such data structures can be done very efficiently. If the tree is balanced both operations perform with logarithmic complexity. The hierarchical tree structure is expressed by *predecessor node* (parent, supernode) and *successor node* (children, subnode). The number of successors and predecessors can be arbitrary. A tree has one *root node* that has no predecessors. A *leaf* is a node that has no successors. *Intermediate nodes* link the root and leafs together by having successors and a predecessor. A *subtree* is a subset of the tree. The edges are directed from predecessor to successor. The *tree height* denotes the length of the path from root node to the furthest leaf. The *depth* of node n is the length of the path from the root node to node n. All nodes at this depth are denoted with a *level*, the root node having level 0. A tree is *complete* if all levels, including the leaf level are fully occupied with data. A tree is *balanced* if each subtree of a node has an equal or almost equally big number of nodes. Multiple balancing schemes exist (AVL-Tree, B-Tree, Red-Black Tree, B*-Tree and more), that differ in their definition of "equal", number of nodes, effort for re-balancing and construction, storage consumption.

2.3 Algorithmic Geometry

Algorithmic geometry is a subarea of computational geometry. Computational geometry concerns all fields, where algorithmic problems are stated in geometrical terms. Algorithmic geometry is the subarea concerning geometric problems modeled through discrete entities.

2.3.1 Binary Search Tree

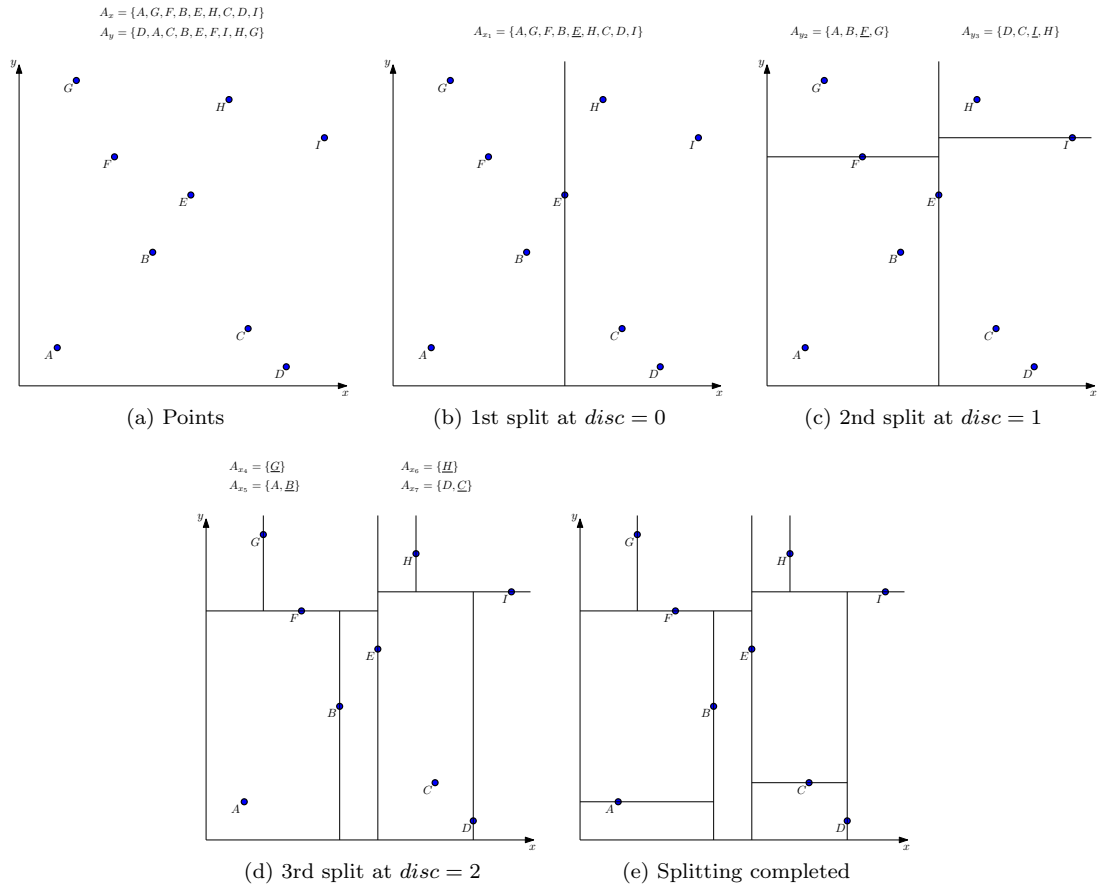
In a *binary search tree* each node has at most two successors *left* and *right*. It is commonly used to structure one-dimensional data for efficient search operations, based on a *key*. All data with a key less than the actual node key is located in the left subtree, the other data in the right subtree. Searching in a binary tree with n nodes has complexity $\mathcal{O}(\log n)$. A complete binary tree has $2^{level} - 1$ nodes.

2.3.2 2-d Tree and k-d Tree

A 2-*d tree* is a binary tree, that operates on two-dimensional data points subdividing the plane into subplanes with alternating x and y coordinates, which act also as keys. The subdivision is done by means of the coordinates of the points. The alternating dimension, upon which the splitting is based, is called *discriminator* (*disc*) and often implemented as an incrementing number. The actual dimension where the split occurs is extracted by a modulo operation.

A k-*d tree* expands the 2-d tree concept to k-dimensional space $\mathbb{R}^k$. The space is then subdivided sequentially for every dimension $d = 1, 2, \ldots, k$. The run time complexity of building such a tree is limited to $\mathcal{O}(n \log n)$, the complexity for a search is $\mathcal{O}(n^{1-\frac{1}{k}} + R)$. The variable R indicates the number of points in range when searching [Bentley75]. When regarding the dimensionality k, building has complexity $\mathcal{O}(k \cdot n \log n)$ and searching $\mathcal{O}(k \cdot n^{1-\frac{1}{k}} + R)$.

Figures 2.2 and 2.3 illustrate 2-d trees as well as k-d trees. As the insertion of further dimensions greater than 2 is simple, the pseudo code for inserting into a k-d tree (algorithm 2.1) and searching it (algorithm 2.2) are listed in generalized form. These algorithms were originally developed by [Bentley75], but were made recursive as we will later use these versions of the algorithms. Searching for a key works very similar to insertion, as can be seen in both listings.



(a) Points (b) 1st split at $disc = 0$ (c) 2nd split at $disc = 1$

(d) 3rd split at $disc = 2$ (e) Splitting completed

Figure 2.2: Construction of a balanced 2-d tree. The rounding median depends on the implementation. Here the median is round-up.

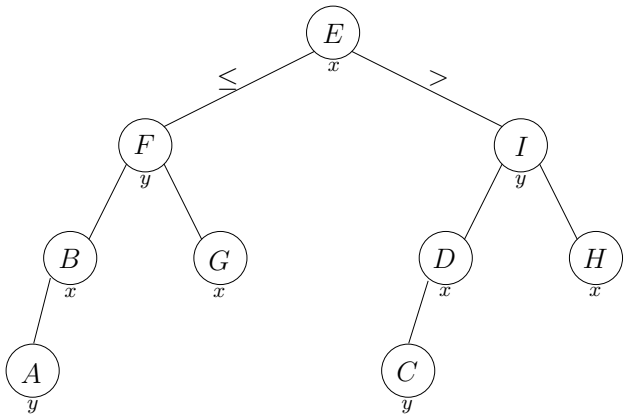


Figure 2.3: Balanced 2-d tree resulting from the construction steps in figure 2.2.

Balancing k-d Trees

We consider again the 2-dimensional case for simplification. The concept is easily extended for k dimensions. The input data set is sorted once by means of the x-coordinates as A_x and once by

Algorithm 2.1: Recursive-Insert-k-d-Tree(P, Q) [Bentley75]

Data: A node P not in the tree. Node Q stores the actual position.

```
1  if (LEFT(Q) = Λ) ∧ (RIGHT(Q) = Λ) then
2     if Q = ROOT then
3        ROOT ← P; LEFT(P) ← Λ; RIGHT(P) ← Λ; DISC(P) ← 0;
4     else
5        /* Append P at the appropriate left or right son of Q.  */
6        SON(Q) ← P; LEFT(P) ← Λ; RIGHT(P) ← Λ;
7        DISC(P) ← NEXTDISC(DISC(P));
8  else
9     /* Search insert position in the left or right subtree recursively.  */
10    if K_DISC(P) ≤ K_DISC(Q) then
11       if LEFT(Q) ≠ Λ then  Recursive-Insert-k-d-Tree (P, LEFT(Q));
12    else
13       if RIGHT(Q) ≠ Λ then  Recursive-Insert-k-d-Tree (P, RIGHT(Q));
```

Algorithm 2.2: Recursive-Search-k-d-Tree(P, Q) [Bentley75]

Data: A searched node P. Node Q stores the actual position.

```
1  if (LEFT(Q) = Λ) ∧ (RIGHT(Q) = Λ) then
2     if K_i(P) = K_i(Q) for 0 ≤ i ≤ k − 1 then
3        return Q;
4     else
5        return Λ;
6  else
7     /* If nodes are equal return Q, else search subtrees recursively.  */
8     if K_i(P) = K_i(Q) for 0 ≤ i ≤ k − 1 then
9        return Q;
10    else
11       if K_DISC(P) ≤ K_DISC(Q) then
12          if (LEFT(Q) ≠ Λ) then  Recursive-Search-k-d-Tree (P, LEFT(Q));
13       else
14          if (RIGHT(Q) ≠ Λ) then  Recursive-Search-k-d-Tree (P, RIGHT(Q));
```

means of the y-coordinates as A_y. At each split the median of the set corresponding to the actual discriminator x or y is inserted into the tree. In this manner a balanced tree is achieved. The concept is illustrated in figure 2.2. The first subdivision is undertaken at the median of A_x, the point E, with the discriminator $disc = 0$, which is x. All points with an x-coordinate $x \leq x_E$ lie in the left subtree of the 2-d tree and respectively the $x > x_E$ in the right subtree. The following subdivision is then done at the y-coordinate of point p_F in the "left" subarea and p_I in the "right" subarea. The left subplane is divided into $y \leq y_F$ (left subtree) and $y > y_F$ (right subtree), and so on. The algorithm pseudo code for building a balanced k-d tree based onto a median can be looked up in [Bentley75].

The following chapter is dedicated to introduce linear and integer optimization theory, with an emphasis on the methods column generation and branch-and-price.

Chapter 3

Optimization Theory Essentials

In this chapter a brief introduction to the topic of linear and integer optimization is given. The theoretical knowledge is needed later in chapter 4 and the following practice chapters. All topics are explained as short as possible, since the area of optimization theory is very huge. This chapter is divided into two main parts, i.e. *linear* and *integer linear optimization*. The first part concerning linear optimization will serve as an introduction to optimization theory in general and present the concepts: history and trivia, modeling linear programs and their standard form, solvability, duality as well as some general solution methods. The passages concerning integer programming explain what distinguishes integer from linear programming, give a classification of integer programs and summarize some of the known solution strategies. The part concerning advanced topics covers more in detail topics immanent for column generation. It encompasses the detailed solution process as well as arising topics like the pricing problem and the tailing off effect.
Mainly, the more general introductory parts linear and integer optimization adhere to [Wol98] and [Van98], which are the main reference books on this subject. Trivia are from [webDesSoum98]. [BoydVan04] is a source for nonlinear optimization. Sources for the second part concerning column generation were mostly [DesLu05], with [Luebbe01] being the related dissertation of one of the authors of the former work. Additional literature in this part is [VanWol96, Vanbk94, NemWol88, GuzRal07, webTrick97, Schrij99].

3.1 History and Trivia

The first concepts of *linear optimization*, also called *linear programming*, date back to the year 1939. The roots lie in the linear inequalities theory. First mathematical models were presented by Leonid W. Kantorovitch. The first formalizations were given by George Dantzig in the 1940ies. At the time of the Second World War the issue was at first held secret due to the advantage in planning war resource allocation. Later industry and production planning became aware of the potential of these methods and since then the field of linear optimization was ever researched. A mighty instrument to solve *linear programs* proved to be the *Simplex algorithm* created by George B. Dantzig in 1947, and with it many before calculation intensive problems suddenly were computable in significantly shorter time and were proved to produce optimal results.
Many real world applications seemed to have some additional properties. Very common in real world applications were problems of integer nature, e.g. they needed integer or binary values in their results like the following, very frequent problems:

- Scheduling of train, aeroplane, bus timetables.
- Personnel, task and machine scheduling.
- Resource planning, raw material needs and production planning.
- Route planning, networking.

In such problems, entities like personnel and resources cannot be separated or fractionated to rational or real values but have to be assigned entirely, e.g. one or more machines or workers to

a specified task regardless of some "optimal" value of 0.73 or $\frac{7}{8}$. Some well known problem formulations for the applications listed above are Scheduling Problem, Resource Allocation Problem, Cutting Stock and Bin Packing.
Problems of these nature are subject of *integer optimization*, which evolved on the basis of linear optimization. Also these problems showed not to be as easy to solve like standard linear problems, because of a "combinatorial explosion" [Wol98], page 8, based on the optimal result to be a large set of feasible solutions. As example the author names the Traveling Salesman Problem (TSP). Looking at an entity with 1000 cities, the solution space has $9.33 \cdot 10^{157}$ feasible tours.
So a multitude of techniques began to evolve around the fields of combinatorial optimization and integer programming which will be explored further in the following.

3.2 Introduction to Linear Optimization

In order to solve a problem with linear optimization, it has first to be analysed and formulated in a mathematical way. The according formulation is called *linear program (LP)* and represents a model of a real world problem. When we analyze such problems many different goals arise:

- Minimize manufacturing costs for prescribed outputs.
- Minimize overall production cost and/or time of products depending on varying resources.
- Maximize profit for varying products, producible with a limited stock of raw materials.
- Maximize output of products, which can be assembled on production lines with predefined time windows and resources.

These goals are limited by factors like raw materials, resources, production cost, processing time or market limits, which also have to be embedded into the model. The formulation in the most cases is the most intensive task in the entire solution process.

Linear programming is a subarea of convex optimization and basis for numerous solution techniques in nonlinear as well as integer optimization. Linear programs are usually interpreted as *general polyhedra*, and many ideas, concepts and proofs are based onto polyhedral theory.

3.2.1 Linear Programs

In order to formalize the concept of a linear program (LP), we introduce some terms and notions. First we introduce the variables x_i, $i = 1, 2, \ldots, n$, which will hold the values that are yet to be decided in an optimal way. These variables are called *decision variables*. For linear optimization, their values lie in $\mathbb{R}$. In the majority of cases they are endowed with some cost variables, which we denote with c_i, $i = 1, 2, \ldots, n$, $c_i \in \mathbb{R}$.

The *linear objective function* is composed of these variables and has the form

$$\max \quad z^* = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n, \tag{3.1}$$

where z^* is the *objective value*. The objective function also encodes the goal striven for: The objective value can either be *maximized* (problems concerning maximal profit or workload, and suchlike) or *minimized* (problems concerning minimal resources or personnel, or similar).

In addition to the objective function the requirements for a problem have to be formulated. This is done through equality and inequality constraints. These *constraints*, also called *restrictions*, express limitations onto a problem (e.g. disposition hours of machines, production plan change delay, sales limits, and so on), expressed in variables a_{ij}. Explicit inequalities whose goal is to prohibit negative numbers are called *non negativity constraints*.

All constraints form a set of restrictions:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m \\ x_1, x_2, \ldots, x_n &\geq 0 \end{aligned} \tag{3.2}$$

By summarizing we obtain a linear program:

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad i = 1, 2, \ldots, m \\ & x_j \geq 0 \quad j = 1, 2, \ldots, n. \end{aligned} \tag{3.3}$$

The above linear program (3.3) models a problem already in *standard form*. In this form the linear objective function is to be maximized and the problem constraints are in terms of $\leq$. Every alternative linear program formulation (minimization problems, constraints with $\geq$ or $=$ operators, negative variables constraints) can be converted into an equivalent formulation in standard form. In few words, a linear program is concerned with optimizing an objective function subject to linear (equality and inequality) constraints.

To render the notation more intuitive it became common to use the canonical *matrix form*. The matrix A denotes the $n \cdot m$ matrix of constraint parameters a_{ij}, $c^\top$ the n-dimensional (row) vector, b the m-dimensional (column) vector, x the n-dimensional column vector of variables[1]:

$$\begin{aligned} \max \quad & c^\top x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{3.4}$$

A shortened expression is common in literature, used by [Wol98], page 3: $\max\{cx : Ax \leq b, x \geq 0\}$.

3.2.2 Duality

Each linear program in standard form, called *primal problem*, can be converted into a *dual problem*. The according dual problem for the linear program in (3.4) is:

$$\begin{aligned} \min \quad & b^\top y \\ \text{s.t.} \quad & A^\top y \geq c \\ & y \geq 0 \end{aligned} \tag{3.5}$$

Here, y denotes the *dual variables*. The author of [Van98] formulates, that *"taking the dual of the dual returns us to the primal"* problem. Another fundamental idea of duality theory is that every feasible solution of an LP embodies also a bound on the optimal value of the objective function of the dual. The most essential theorems of duality theory are the *weak duality theorem* and the *strong duality theorem* [Van98], presented in the following.

[1] Common notation: $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$

Definition 3 (**Weak Duality Theorem**). *"If* $(x_1, x_2, \ldots, x_n)$ *is feasible for the primal and* $(y_1, y_2, \ldots, y_m)$ *is feasible for the dual, then:*

$$\sum_j c_j x_j \leq \sum_i b_i y_i, \qquad \textit{or in condensed form:} \qquad c^\top x^* \leq b^\top y^*."[Van98].$$

Definition 4 (**Strong Duality Theorem**). *"If the primal problem has an optimal solution* $x^* = (x_1^*, x_2^*, \ldots, x_n^*)$ *then the dual also has an optimal solution* $y^* = (y_1^*, y_2^*, \ldots, y_m^*)$ *such that:*

$$\sum_j c_j x_j^* = \sum_i b_i y_i^*, \qquad \textit{or in condensed form:} \qquad c^\top x^* = b^\top y^*."[Van98].$$

The essence of those two theorems, is depicted in figure 3.1. If there is no *duality gap* between the dual and primal objective values, the objective value is *optimal*. These observations are very useful, since they provide a method to easily get upper and lower bounds as well as to verify optimality. On page 66, [Van98] calls it a *"certificate of optimality"*.

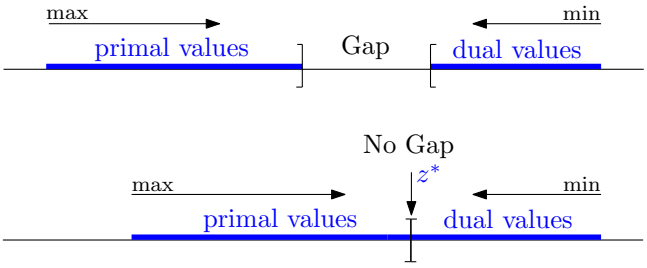


Figure 3.1: Duality Gap between largest primal and smallest dual value. Image credits to [Van98].

3.2.3 Polyhedral Theory, Solvability and Degeneration

The *solution* $x_1, x_2, \ldots, x_n$ to a linear program in standard form is a specific value for each of the decision variables. In the context of the objective function they form the objective value z^*. A solution is *feasible* if it satisfies all restrictions, and *optimal* if in addition the objective function value is maximal.
A geometric interpretation of an LP is that its equations form a *convex polyhedron*[2]. The polyhedron $P = \{x \mid Ax \geq b\}$ defines the *feasible region* where the solution lies within. The linearity of the objective function implies that the optimal solution can only be located on the boundaries of this feasible region, and there it is located on a vertex or facet of the polyhedron, since the solution is not stringently unique. The objective function hyperplane touches the polyhedron at the point where the optimum lies, the orientation depending on a minimization or maximization problem. An example of such a polyhedron is showed in figure 3.2.
Polyhedral theory makes clearer and more intuitive the nature of the solution space and solution process for linear programs. A formulation provides the *convex hull* of the formulated problem. Good formulations have tighter convex hulls. When solving a primal LP, the according polyhedron $P = \{x \mid Ax \geq b\}$ is interpreted the following [Van98]:

- If $P \neq \emptyset$ and a minimum $\min\{c^\top x \mid x \in P\}$ exists, the linear program is *solvable* and has the finite solution x^* with $c^\top x^* = \min\{c^\top x \mid x \in P\}$.
- If $P \neq \emptyset$, but infimum $\inf\{c^\top x \mid x \in P\}$ does not exist, the linear program is solvable, but no optimal solution exists. The polyhedron is *unbounded* in the direction of the objective function, the values growing to infinity. Example: $\max\{x \mid x \geq 0\}$.
- If $P = \emptyset$, the restrictions contradict each other and there is no solution, since the feasible region is empty. The linear program is called *infeasible*. Example: $\max\{x \mid x \geq 2;\ x \leq 1\}$.

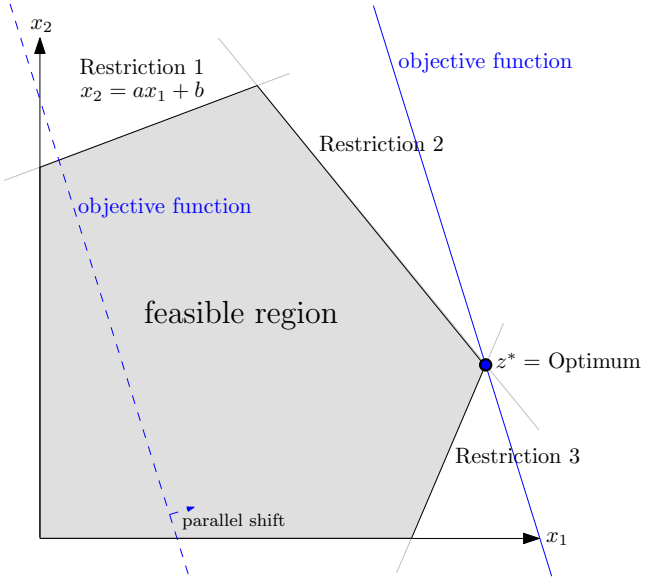


Figure 3.2: Example of a geometric interpretation.

In the latter two cases, the LP is called *degenerated*. Additional relations are found between primal and dual problems [Van98]:

- If the primal problem has an optimal solution, the dual has one also. There is no duality gap (strong duality theorem).
- If the primal problem is unbounded, the dual problem is infeasible (weak duality theorem).
- If the primal problem is infeasible, the dual problem is unbounded (weak duality theorem).
- The case exists that both, primal and dual problem are both infeasible. The duality gap extends from $-\infty$ to $+\infty$.

3.2.3.1 Farkas' Lemma

This lemma was set up by Julius Farkas in the 1900 and obtained importance for linear optimization, since also by using this lemma the strong duality theorem could be proved. Additionally, it may be used for giving a certificate or proof of *infeasibility*. Some variants exist in literature.

> **Lemma 1 (Farkas' Lemma).** *"Either there exists x in $\mathbb{R}^n$ with $x \geq 0$ such that $Ax \leq b$ or there exists y in $\mathbb{R}^m$ with $y \geq 0$ such that $y^\top A \geq 0$ and $y^\top b < 0$." [AvKa04], page 156.*

3.2.4 Solution Methods for Linear Programs

When solving linear problems, a multitude of solution methods can be applied. The approaches outlined in the following are not exclusively used for solving linear programs, but often are incorporated into the integer problem solution process as well.

[2] A bounded polyhedron with finite diameter is called *polytope*. Literature refers to a convex and bounded polytope also with the term polyhedron.

3.2.4.1 Geometric Method

The geometric interpretation of an LP was already presented in section 3.2.3 and constitutes a quick and easy way of getting solutions for small linear programs. But only small problems with few variables are solved by such graphical methods. With increasing dimensions or when having a somewhat greater quantity of restrictions and variables they become very unhandy and the precision in the most of cases is very inaccurate.

3.2.4.2 Simplex Algorithm

Inspired by this geometric interpretation G. Dantzig created the *Simplex algorithm*. Fundamentally, this algorithm searches the vertices of the polytope described by the problem constraints, starting by a feasible solution. The algorithm proceeds along the edges of the polytope having greater reduced costs, thus improving the objective function, and advances until the optimum is found or unboundedness is asserted. The Simplex algorithm is called an *edge following* method and constitutes of the following steps:

1. Check for infeasibility.
2. Convert LP into *augmented form*: Introduce non-negative *slack variables* in order to replace inequality with equality constraints. In this manner a block matrix form is achieved which constitutes the *Simplex tableau*.
3. Determine a feasible starting solution. The slack variables become basic, the main variables become non-basic[3].
4. While basis solution is not optimal do:
   (a) Determine *entering basic variable*: Select a nonbasic variable, having maximal reduced cost. This variable is called *pivot element*.
   (b) Determine *leaving basic variable*: Select the basic variable which will be dropped in order to obtain an improving adjacent edge. This is performed via the *ratio test*.
   (c) Transform the Simplex tableau into canonical form by pivoting the entering variable. The leaving variable becomes nonbasic and the entering variable becomes basic. This operation is called *pivot*.
   (d) Terminate if unboundedness or cycling (a previous state is revisited) is detected.

Since the Simplex algorithm considers only adjacent edges, in each iteration only one substitution of basic for a non-basic variables is performed. Dantzig designated the entering variable as the one having *maximal reduced cost*, but over time alternative strategies were developed.
Polyheder theory shows us that if an optimal solution exists at the edge of the polytope the Simplex algorithm always brings forth this optimal solution, or the problem is infeasible or unbounded. In special cases a degeneration in the Simplex tableau happens, in particular the occurrence of cycles. This can be avoided through *Bland's rule*, which performs the selection of pivot elements in a special way. The worst case complexity is exponential in problem size, but almost never 2^n steps are required. In practice the Simplex algorithm turned out to be very applicable and efficient. At the present time commercial and free software is able to solve systems with some millions of variables and restrictions.

Variants and improvements The selection of the pivot element influences highly the number of iterations as well as the numerical stability especially when having degenerated tableau's. Thus a good pivot element selection strategy is required. There exist a lot of methods for selecting pivot elements, or pivot columns and rows: Dantzigs approach (maximal reduced cost) resulted to be relatively calculation intensive. One prevalent selection strategy is *steepest edge pricing*, which selects column and row that together yield greatest overall improvement for the objective. It is computation intensive, but the number of iterations is significantly decreased. *Devex pricing* is an approximation of steepest edge pricing in combination with a normalization of the values

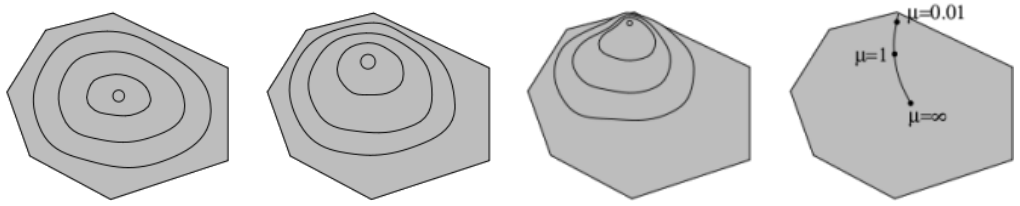
[3]Besides this approach, other more sophisticated methods exist

for greater expressiveness. All strategies are standard in most modern solver software. Also very common are the *Harris quotient* and the already named *Bland's rule*.

The classical Simplex algorithm proceeds until any further addition diminishes the objective value again and all slack variables are removed from the basis. Simplex steps can be performed for the primal and the dual problem as well, this leads to variants employing both primal and/or dual Simplex (*primal-dual-Simplex*). Other variants are two-phase-Simplex, M-method.

3.2.4.3 Nonlinear Optimization Techniques

Very fast algorithms for solving linear problems come from the area of nonlinear optimization. In contrast to the method presented above, these methods are *path following*. Foundation for all nonlinear techniques is the *ellipsoid method*, which "encircles" the problem solution by ellipsoids of decreasing sizes. Its importance lies in the evidence for *polynomial time solvability* of LP's, but it showed to be to slow to be of practical interest. The approach inspired many new ideas which disembogued in the development of *interior point methods*, which nowadays provide the basis for most nonlinear approach implementations.
As depicted in figure 3.3, the optimum is found by moving iteratively along a path through the interior of the polytope. Roughly outlined the problem is redesigned in terms of logarithmic barriers[4], which depend on a carefully selected value for pace μ, a step direction and a decreasing step width, calculated in each iteration. The central path converges to the optimal point as $t \to \infty$. Again, primal-dual symmetry is exploited as optimality criterion. This class of LP solving methods are characterized by polynomial complexity and fast convergence towards optimality. In practice interior point methods afford $\mathcal{O}(\log n)$, and are competitive to the Simplex algorithm. Modern solver software employ such methods for quick calculation of bounds and preliminary solutions.



(a) $\mu = \infty$ (b) $\mu = 1$ (c) $\mu = 0.01$ (d) central path

Figure 3.3: Figures 3.3a - 3.3c show contour lines of a logarithmic barrier function for three values for the pace parameter μ. The maximum value lies inside the innermost level set. Figure 3.3d depicts the *central path*. Image credits to [Van98].

3.3 Integer Optimization

At the present time Simplex-based and interior point methods are considered to be of similar efficiency and performance for routine applications in industry. But also there exists a huge amount of problems of vast complexity or size for which a careful analyzing, planning and use of tricks is necessary for solving them in an acceptable amount of time.

3.3.1 Introduction

As anticipated in section 3.1, a multitude of real world application are (in contrast to the LP solutions that lie in $\mathbb{R}$) of integer nature, e.g. their solution consists of integer or binary values or are of mixed nature. Integer problems are classified by [Wol98] as follows:

[4]Hence the synonym *barrier methods*.

Integer Program (IP) or **Integer Linear Program (ILP)**

$$\begin{aligned} \max \quad & c^\top x \\ & Ax \leq b \\ & x \in \mathbb{Z}_+^0 \quad (x \geq 0 \text{ and integer}) \end{aligned}$$

Binary Integer Program (BIP)

$$\begin{aligned} \max \quad & c^\top x \\ & Ax \leq b \\ & x \in \{0,1\}^n \end{aligned}$$

Mixed Integer Program (MIP)

$$\begin{aligned} \max \quad & c^\top x + h^\top y \\ & Ax + Gy \leq b \\ & x \geq 0,\ y \in \mathbb{Z}_+^0 \end{aligned}$$

In this context we name the *combinatorial optimization problem (COP)*, a problem that often can be formulated as an integer or binary integer program. Again the polyhedron formed by the problem constraints defines the feasible region for the solution. But in contrast to problems in $\mathbb{R}$ the optimal solution for IP's are not necessary located on the border, but can lie at a discrete point *within* the bounds of the feasible region. As illustrated in figure 3.4, one can be tempted to *round* a linear programming solution, but in the most of cases this procedure would be inadequate, since rounded values may lie far away from the optimal values.

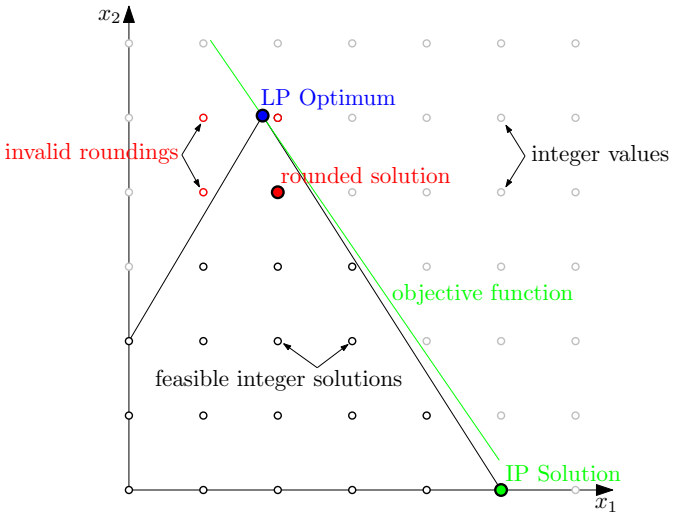


Figure 3.4: Rounding is not always a good idea in IP's and BIP's: The IP solution is far away from the rounded values. Image credits to [Wol98], page 4.

3.3.2 Relaxation

To solve IP's it sounds suggesting to resort to linear programming theory. Imagine an IP:

$$\begin{aligned} z^* = \max \ & c^\top x \\ & Ax \leq b \\ & Dx \leq d \\ & x \in \mathbb{Z}_+^n \end{aligned} \tag{3.6}$$

where the IP with one part of the constraints $Ax \leq b$ is "easy" to solve, but including $Dx \leq d$ extremely difficult. The idea is now to drop the "complicating constraints" ([Wol98], page 167)

$Dx \leq d$ and solve the IP without them. The resulting solution may be weak and non integer. This procedure is called *relaxation*. A relaxation of program P is commonly denoted as P'. Their respective optimal solutions are labelled z_P^* and $z_{P'}^*$.
A relaxation would also be the omission of the integer and/or binary requirements and solve the problem in $\mathbb{R}$, or the substitution of the complicating constraints with easier ones. Then, based on the interim solution from the relaxed program P', the optimal solution of the IP is searched. As previously stated, rounding is insufficient. The following sections cover alternative strategies. Another use for primal and dual relaxations is the calculation of upper and lower bounds. In this context we can state:

> **Proposition 1.** *"(i) If a relaxation P' is infeasible, the original problem IP is infeasible.*
> *(ii) Let $z_{P'}^*$ be an optimal solution of P'. If $z_{P'}^* \in X \subseteq \mathbb{Z}^n$ and $f(z_{P'}^*) = c(z_{P'}^*)$ then $z_{P'}^*$ is an optimal solution of IP." Proof see [Wol98], page 26*[5].

3.3.3 Exact Solution Methods for Integer Programs

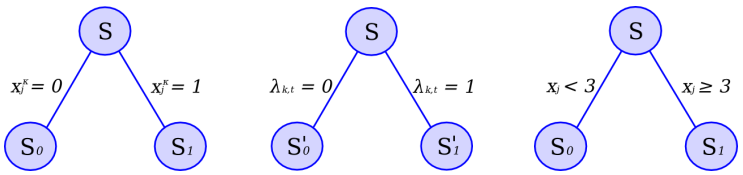
In the following sections various solution strategies for integer programs are presented.

3.3.3.1 Branch-and-Bound

Branch-and-bound (BB) constitutes a popular meta technique for solving problems of all possible kinds and is no exclusive concept of integer programming. The idea comes from the old concept of *divide & conquer*. First applications to linear optimization were made by A.H. Land and A.G. Doig in the 1960. Shortly outlined, BB enumerates systematically all candidate solutions. This is done by dividing a problem into subproblems (*branch*) and associating possibly good upper and lower limits (*bounds*) for the solution space in order to narrow the search regions. In this manner, an *enumeration tree* is constructed. Subtrees exceeding the bounds are *pruned*, since they never would produce an optimum. In [Wol98], page 94, the author presents the following pruning strategies, S_t denotes subtrees:

- Pruning by optimality: $z^t = \{\max c^\top x : x \in S_t\}$ has been solved.
- Pruning by bound: $\overline{z}^t \leq \underline{z}$, with $\overline{z}^t$ the upper and $\underline{z}$ the lower bound.
- Pruning by infeasibility: $S_t = \emptyset$.

The goal is to cut away most unnecessary iterations in the enumeration tree for performance gain and thus avoiding the worst case of searching the complete enumeration tree, which can be huge. Upper bounds are derived by relaxation, lower bounds through heuristics, trivial problem instances, interim solutions, and suchlike. Performance depends on the used branching scheme which should be carefully selected. Some branching strategies are depicted in figure 3.5. Suggesting branching strategies for integer programming are: If S is the set of all feasible solutions, it can be split into $S_0 = S \cap \{x : x_j^\kappa = 0\}$ and $S_1 = S \cap \{x : x_j^\kappa = 1\}$.



(a) 0-1 branching (b) column variables branching (c) range division branching

Figure 3.5: Branching Strategies for IP's. Image credits to [Wol98], page 193.

Modern IP solver software makes excessive use of BB, and the strategies branch-and-cut, branch-and-price are based on this method.

[5] the terms were adjusted to the used notation: RP to P', x^* to $z_{P'}^*$

3.3.3.2 Cutting Planes

These technique dates back to work from D.R. Fulkerson, G. Dantzig, R. Gomory in the 1950 and until now it is subject to actual research work. *Cutting plane methods* exploit the fact that frequently not all restrictions are needed to solve an integer linear program. Often a small number of constraints suffice to find the solution, and it is tried to observe only the problem constraints that configure the relevant facets of the polytope, by dropping irrelevant ones. Also there exist many problems comprised of an very huge number of constraints for which it is hard to even write down the complete ILP itself[6].
A cutting plane algorithm hence starts with a small amount of restrictions, the *restricted problem (RP)*, which was derived by relaxation. The algorithm computes a solution z' for the RP and determines if it is already optimal or not. If z' is not optimal the algorithm successively adds additional restrictions from the original problem. In order to select "good" restrictions for the next iteration and favor promising ones a *Separation Problem (SEP)* is constructed. Its solution is a valid inequality constraint that violates the optimality criterion in the previous step, which is called *cut*. The cut is added to our restricted problem. With the additional restriction the program produces a new solution that hopefully lies nearer to the optimal solution z^*. If this new solution is again not optimal the process begins anew until no restrictions can be added anymore. By adding cuts, we cut off the non integer solution space in our polytope, and thus narrowing it. A pseudo code outline for the cutting plane algorithm is presented in listing 3.1.

Algorithm 3.1: Cutting Plane Algorithm

Data: An integer linear program P
Result: Optimal solution z^*

```
1 P' ← relaxation(P);
2 z*_P' ← solve(P');
3 while (z*_P') is not optimal do
4     solve-separation-problem SEP;
5     /* Add solution constraint from SEP to the restricted problem.  */
6     P'' ← P' ∪ (πx ≤ π0)_SEP;
7     z*_P'' ← solve(P'');
8 return z*;
```

When considering the combinatorial optimization problem (COP) $max\{cx : x \in X \subseteq R^n\}$, the separation problem embodies the difficulty of the task and is defined as follows:

> **Definition 5** (**Separation Problem**). *"The* Separation Problem *associated with COP is the problem: Given $x^* \in R^n$, is $x^* \in conv(X)$? If not, find an inequality $\pi x \leq \pi_0$ satisfied by all points in X, but violated by the point x^*." [Wol98], page 37.*

In this context, the author names the polynomial equivalence of optimization and separation problem: A linear objective function over a polytope can be solved in polynomial time, if the corresponding separation problem is polynomially solvable [Wol98], page 88 ff.

3.3.3.3 Branch-and-Cut

Branch-and-cut is a hybridization of cutting plane methods and the branch-and-bound meta-algorithm. The integer program version of the algorithm starts as the cutting plane algorithm: after relaxing the master problem to a restricted master problem, the restricted problem is solved. If the solution is not optimal, e.g. there exist non-integer variables supposed to be integer, the algorithms searches again for an addable cut by solving the separation problem. At this point

[6] For example the the Traveling Salesman Problem can be constructed as an Acyclic Subgraph Problem having exponential many restrictions.

the problem is split into two subproblems: The standard approach effects branching based onto variable values. For example if an integer constrained variable x_i was found to be fractional, the two new subproblems divide the solution space into disjoint parts by setting in the first subproblem $x_i' \leq \lfloor value \rfloor$ and in the second subproblem $x_i'' \geq \lceil value \rceil$. Each feasible solution must suffice one of the subproblem constraints. In this manner we can systematically search the solution space by building a decision tree.
Branch-and-cut usually heavily depends on good relaxations and fast separation problem solving, but has the advantage of exerting small decision trees. Separating and optimality checking heavily influence execution time. Variants are known which limit run time to a predefined value or until the solution has reached a sufficiently small deviation from the optimum (duality gap). Branch-and-cut is a generic approach that is able to solve a wide variety of optimization problems.

3.3.3.4 Column Generation

Inspired by the dual nature of optimization theory the "dual" to cutting plane methods found also an application. G. Dantzig and P. Wolfe were the first to explore the special structures of problems with many variables and their ideas were continued by P.C. Gilmore and R.E. Gomory, who developed the first column generation algorithm. Column generation and branch-and-price are a promising research area, where many insights can be expected in the future. In the following column generation is presented straightforward, advanced topics are summarized in section 3.4.

Column generation in practice is applied for solving huge integer and linear problems, which consist of a very large amount of variables embodied in a less critical set of "linking" restrictions. In the most of cases a big part of variables in the optimal solution will assume zero values, so it is tried only to consider the promising ones. These variables or columns[7] in the restrictions depict *"incidence[8] vectors of certain subsets of a set, that is tours, client subsets and so on."* [Wol98], page 186. Also a typical sparseness in the constraint matrices of huge integer linear problems attracts attention.

The basic idea of the algorithm itself is similar to the cutting plane method: The column generation algorithm starts with a narrow set of columns in the restrictions and tries to advance towards the optimal solution by subsequently adding new columns to the current problem.

Problems with a large amount of variables sometimes arise naturally, but more often they arise through *decomposition* or *reformulation*. The purpose of the latter practices is to extend a *Compact Formulation (CF)* to an *Extensive Formulation (EF)* which has some positive properties in the subsequent solution process. The extended problem formulation represents[9] the *master problem (MP)* and provides the starting point for the algorithm, λ denotes the variables:

$$\begin{aligned} z^* = \min \quad & \sum_{j \in J} c_j \lambda_j \\ s.t. & \sum_{j \in J} a_j \lambda_j \geq b \qquad (MP) \\ & \lambda_j \geq 0 \;\; j \in J \end{aligned} \tag{3.7}$$

Basically the column generation algorithm is a Simplex algorithm that in each iteration identifies a non basic variable to enter the basis. In this step, we search for the variable having maximal reduced costs [DesLu05] $\arg\min\{\bar{c}_j = c_j - u^\top a_j \mid j \in J\}$, with $u \geq 0$ being the dual variables.

[7] Plenty of literature refers to "columns" and "variables" as the same in the field of column generation. A column is formed by one variable vector of the same index.

[8] Remark: Maybe the notion characteristic vector describes the intended meaning more precisely

[9] [Wol98] names the integer programming original and master problem integer program (IP), as well as integer programming master (IPM). The linear programming original, master and restricted problem are called linear program (LP), linear programming master (LPM) and restricted linear programming master (RLPM).

Since $|J|$ can be extremely large, an explicit search may not be possible in all cases. To circumvent this, out of the master problem an appropriate small set of non-negative columns $J' \subseteq J$ is chosen, which form the *restricted master problem (RMP)*. Thus we get a smaller, computable linear program. Its dual equivalent is the *dual restricted master problem (DRMP)*.

The RMP is solved by a Simplex algorithm and returns a feasible primal solution $\overline{\lambda}$ as well as a feasible dual solution $\overline{u}$. An optimality check is performed by comparing primal and dual solution. If there is a duality gap, the present solution is not optimal, and the actual RMP has to be extended, since not all vital variables are yet part of the actual solution.

In the following step, the algorithm decides which variable to add next. In order to enhance the current solution a column with associated greatest *negative reduced costs* (in the case of the minimization problem) is in search [DesLu05]:

$$\bar{c}^* = \min\{c(a) - u^\top a \mid a \in \mathcal{A}\}, \tag{3.8}$$

with $a_j, j \in J$ being the columns implicitly given as elements of set $\mathcal{A} \neq \emptyset$. The problem of finding such columns is commonly called *pricing problem (PP)* or *column generator*. If $\bar{c}^* < 0$ we extend the RMP with the column deduced from our pricing problem result and start again the RMP solution process. Otherwise, if no column is found, or when $\overline{c}^* \geq 0$, there exists no negative reduced cost coefficient $\bar{c}_j$ and the primal solution $\overline{\lambda}$ is an optimal solution for the master problem. Column generation is outlined in figure 3.6. The pseudo code outline is listed in algorithm 3.2.

Algorithm 3.2: Column Generation (LP version)

Data: Master problem MP
Result: Optimal solution z^*

```
1 /* Determine a feasible starting solution */
2 RMP ← Select-starting-subset-of-columns_{j∈J}(MP);
3 z*_RMP ← solve(RMP);
4 while ∃ a variable with reduced cost c̄_j < 0 do
5    Determine variable j with c̄_j < 0;
6    RMP' ← RMP ∪ j;
7    z*_RMP' ← solve(RMP');
8 return z*;
```

If the MP and thus the RMP are integer programs, the respective relaxation[10] is solved by the Simplex algorithm instead. Further precautions have to be taken to ensure the integrality of the final solution, see the following sections 3.3.3.5 and 3.4.

The pricing problem can be established in various ways: It may be possible to build a subproblem which maximizes or minimizes the reduced costs depending on the optimization sense. Approaches using enumeration, dynamic programming[11] or branch-and-bound were implemented, as well as heuristics and approximation algorithms. Methods and pricing schemes are listed in section 3.4.6.

The formulation of such pricing subproblems is a complex task and requires careful consideration. To find an appropriate method we need to keep in mind the structure of the columns and the *"interpretation of cost are naturally defined on these structures"* [DesLu05]. This means that the columns express structural properties for the modelled objects (sets, permutations, paths) they encode and by using this knowledge problem specific pricing problem solutions are possible.

[10] The integrality condition is dropped.
[11] As for example in the Cutting Stock Problem solved by Column Generation.

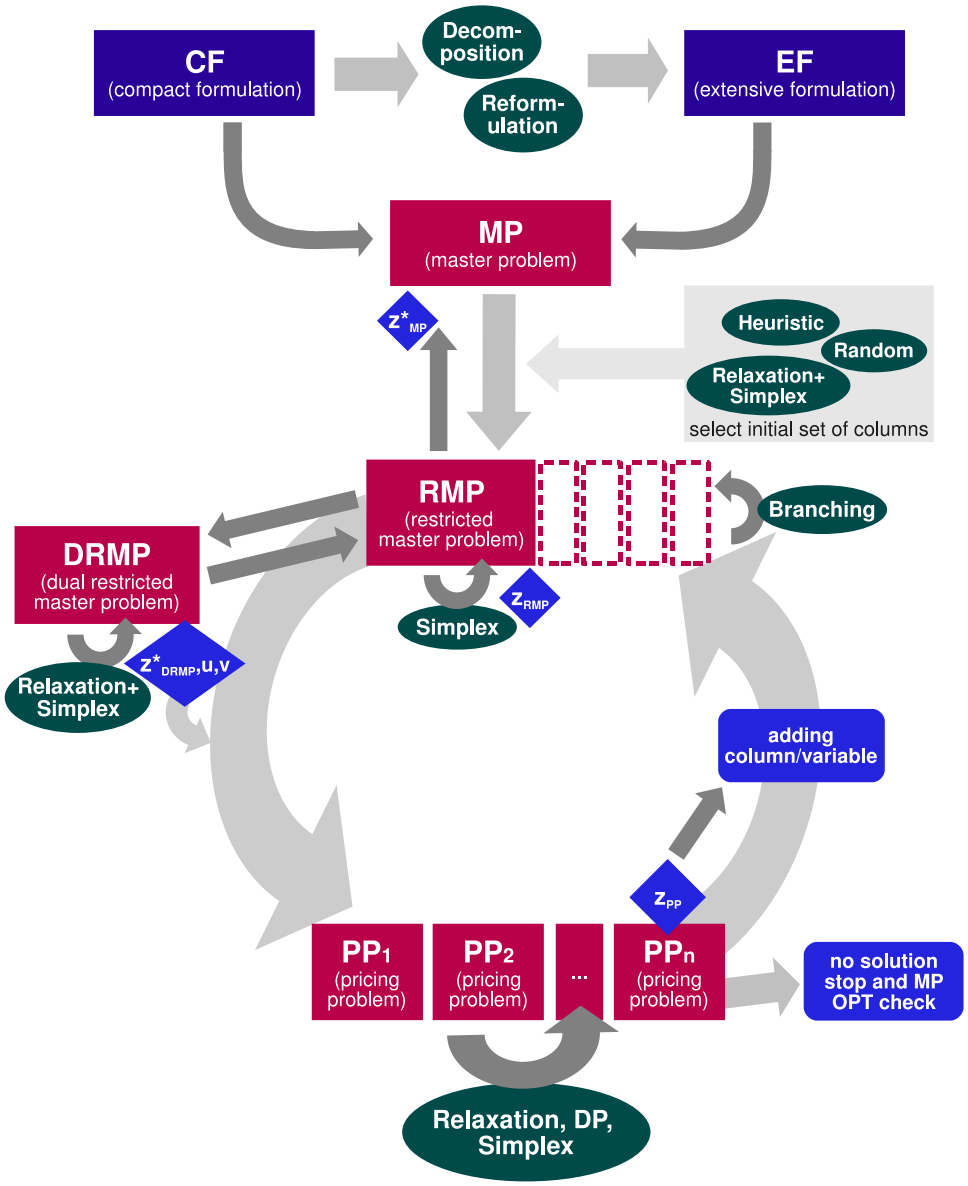


Figure 3.6: Column generation flowchart.

3.3.3.5 Branch-and-Price

Branch-and-price works similar to branch-and-cut. Accordingly, branch-and-price is a combination of column generation and branch-and-bound and mainly used for solving huge integer programs.

The algorithm derives again a restricted master problem, and generates new columns, if the interim solution is fractional, or until an optimal integer solution is achieved. While having fractional intermediate solutions, these values are used as bounds and the actual problem is split into two subproblems with some branching schema dividing the solution space into two complementary subspaces. Various branching schemes exist, for example rounding of some fractional variable up in one subproblem and down in the second one, or setting binary variables to 0, respective 1 in the corresponding subproblems. Thus for all variables found in the meantime, a decision tree is

built systematically, indicating if some variable is part of the actual subproblem or not. If subtrees are found to have upper bounds lower than the actual lower bound, they will be pruned in BB fashion, since for sure no optimal solution is found there. The optimal solution in the end is found at some leaf. The topics are presented more in detail in section 3.4.8.1.

3.3.3.6 Branch-and-Cut-and-Price

Finally, a fusion of all mentioned methods is named: *Branch-and-cut-and-price (BCP)* combines cutting plane separation, column generation and branch-and-bound techniques to a powerful tool. Starting from a very restricted master problem, new cuts and columns are dynamically created continuously during the branching and pricing process.

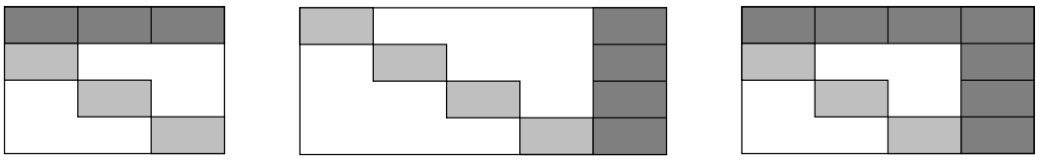
Target of this method is the solution of large scale problems, and the entire procedure has to be carefully planned and prepared. In [Luebbe01], the author annotates, that literature covers mostly "well behaving" examples and that the *"process is not very well understood, but nonetheless successful applications of this elaborate technique are reported."*

3.4 Advanced Topics in Column Generation

The principles presented in section 3.3.3.4 are now described more in detail. The distinct steps of the algorithm are presented separately.

3.4.1 Problem Formulation and Block Diagonal Structure

Problems with many variables have a special structure that can be taken advantage of. It originates from the underlying real world problems logical and hierarchical structure. It has been observed that such large problem matrices contain a relatively big amount of zero variables and a small amount of nonzero variables, and are therefore called *sparse*.



(a) linking constraints (b) linking variables (c) linking constraints and variables

Figure 3.7: Example schemes of block diagonal matrices, shaded areas indicate nonzero elements [Luebbe01].

Also most variables occur only in subsets of restrictions, plus all different variables together appear, if ever, only in very few restrictions. All of them are linked together by the constraints, and/or by being in the same column. The non zero variables form a sort of angular *block diagonal structure*, depicted in figure 3.7. This block diagonal structure forms the basis to the decomposition approach, superscripts indicate the column affiliation:

$$D = \begin{pmatrix} D^1 & & & \\ & D^2 & & \\ & & \ddots & \\ & & & D^\kappa \end{pmatrix} \qquad d = \begin{pmatrix} d^1 \\ d^2 \\ \vdots \\ d^\kappa \end{pmatrix} \tag{3.9}$$

> "The general idea behind the decomposition paradigm is to treat the linking structure at a superior, *coordinating*, level and to independently address the subsystem(s) at an subordinated level, exploiting their special structure algorithmically" [DesLu05].

If a compact problem formulation does non exhibit such a structure, it is transformed in a way to apply structure that can be weighed easily through some formula or quality calculation process. The derived extensive formulation and the original compact formulation correlate in the following way, a thesis addressed by Adler and Ülkücü (1973), Nazareth (1987), here cited from [DesLu05]:

> "Although the compact and the extensive formulation are equivalent in that they give the same optimal objective function value z^*, the respective polyhedra are not combinatorially equivalent".

Extensive formulations (EF) arise fairly often naturally or from the decomposition process and possess advantages over compact formulations (CF). Occasionally EF have increased memory or storage demands, whereas the CF tend to be quite compact. Motivations for extended formulations include [Luebbe01]:

- Decomposition may provide a very natural interpretation in terms of the original compact formulation, and allow the incorporation of complicated constraints.
- It is often noticed that extensive formulations are usually *stronger* than compact formulations and its LP relaxation approximates more tightly the convex hull of the problem.
- Compact formulations may have a symmetric structure that affects the branch-and-bound performance. Decomposition reduces or eliminates these difficulties, but influences the LP relaxation and solution process.

3.4.2 Decomposition, Reformulation, Convexification, Discretization

Various approaches exist for reformulating a compact formulation (CF) as an extensive formulation (EF). Further methods for strengthening the solution space exist, namely convexification and discretization. Since this area is wide and the topics are not immediately needed, only a short introduction is given.

Dantzig Wolfe Decomposition, first introduced by G. Dantzig and P. Wolfe in 1960, is a standard method for decomposing compact formulations into extended formulations having a large number of variables but possibly a lot fewer rows than the CF. The principle can be applied to linear and integer programs as well. Dantzig Wolfe Decomposition of an LP is referred to in [Luebbe01, DesLu05], the reformulation of an IP is presented in [Wol98], page 187 ff.

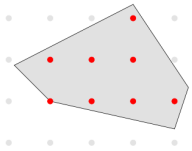
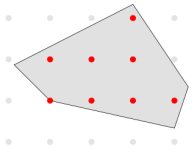
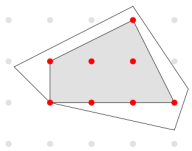
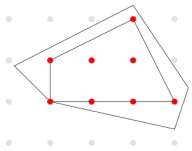
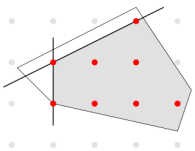
Lagrangian Relaxation is an extension to relaxation. It is a popular approach for dealing with weak bounds obtained by classical relaxations. The main concept is the penalization of the m complicating constraints $Dx \leq d$ in the objective function. Again, this principle can be applied to LP's and IP's and is described in [Wol98], page 28.

Convexification bases on the strong relation of Dantzig Wolfe decomposition and Lagrangian relaxation and is an approach for integer programming decomposition presented by [Vanbk94]. Goal is to restrict a formulation to its convex hull $conv(S)$. The occurrence of fractional multipliers led Vanderbeck to the development of *Discretization*, a true integer analogue to the decomposition principle. Basis principles and differences of discretization an convexification are depicted in table 3.1.

3.4.3 Set Covering, Set Partitioning and Set Packing Principles

Three kinds of structure are often found in integer optimization. Constraints often depict such structures, but there exist entire problem formulations as well. Concepts and descriptions are

Table 3.1: Survey to discretization an convexification. Image credits to [Luebbe01]

	Example of a feasible region S of a compact formulation: $S = \{x \in \mathbb{Z} \mid Dx \leq d, x \geq 0\}$
	Effect of reformulation of an LP. Dantzig Wolfe decomposition is applied to the CF's relaxation and merely the polyhedral representation of the linear relaxation of S is altered. The shape does *not* change and the quality of the lower bound provided by the LP relaxation does *not* improve.
	*Convexification* of S: "If $conv(X)$ is not already a integral polyhedron, the bound provided by the linear programming relaxation of the reformulation is stronger than the one obtained from the original formulation" [Luebbe01]. We convex combine the extreme points of $conv(S)$ in order to obtain interior integral points.
	Convexification may lead to fractional multipliers and extra arrangements to enforce integrality of the resulting integer program. This leads to direct reformulation of the integral points, i.e. S itself, called *Discretization*. Both convexification and discretization give the same bound obtained from the respective linear relaxation.
	In contrast: Effects of the addition of cutting planes to S.

from [Vanbk94], [webTrick97] and [Luebbe01]. The first author presents a general set partitioning problem. Basis is a ground set R of m elements $1, ..., m$:

- *Set Partitioning* can be described as finding a set S being the minimum cost selection out of the power set $\mathcal{R}$, but in a way that each element of the ground set R appears in *exactly* one subset S. The problem itself is formulated with *equality constraints* having a right hand side of 1. For the underlying application, not every set of $\mathcal{R}$ will be feasible. Problems with this kind of structure are found in flight scheduling, voting district allocation, crew scheduling, and is best characterized by: Every customer is served by exactly one server.
- *Set Covering* is an assignment (at minimum cost) of each element from the ground set R to *at least one* of the subsets S, i.e. the elements from the ground set must be covered at minimum once. Constraints of this type consist of sums of binary variables, a "greater than, or equal" relation and the right hand side 1. Problems of this type are Routing, Location allocation, Scheduling. One can say, each customer is served by some location/person/device/vehicle.
- *Set Packing* is given when every set element has to appear in *at most* one subset. In other words, it is tried to satisfy as much demand as possible avoiding conflicts. Constraints are of the form "less than, or equal", and are characteristic to certain scheduling problems.

3.4.4 The Restricted Master Problem (RMP)

As already stated, the *Restricted Master Problem (RMP)* consists of a manageable subset of columns selected from the master problem, where the size of the latter would exceed either time or memory at disposal when explicitly searching through the huge number of candidates. The RMP serves as starting point for the subsequent pricing problem iteration process, where RMP itself is consecutively expanded by adding the improving columns from the pricing subproblem outcomes. Any master program narrowed by omitting columns is called restricted. In the corresponding dual therefore the according rows are omitted.
The actual purpose of the RMP is to make available the dual variables needed in the pricing process and for the algorithm termination control, since the stopping criterion would be equaling dual and primal objective values. The additional task of the RMP is to combine the subproblems solutions in order to get primal feasible solutions. At all times, in the RMP is gathered all relevant information from already solved subproblems, which consists in a column subset, that was found to improve the objective value. Figure 3.8 outlines the information flow between MP and RMP(s).

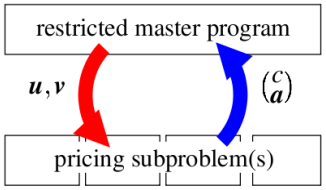


Figure 3.8: *"Information flow between master program and subproblem(s)."* Image credits to [Luebbe01].

In [DesLu05], the authors describe column generation as a method that works toward dual feasibility by maintaining primal feasibility. It is important to keep in mind that dual values influence the selection of new variables and therefore must be monitored and analyzed carefully. However, since an initial set of columns for a RMP does not come up by itself, it has to be deduced somehow.

3.4.4.1 Deducing an Initial Basis for the RMP

Independently from the used solution method, the RMP has to be initialized. The most common methods for solving the RMP, the Simplex algorithm, requires a good initial basis, since it is of crucial importance for the whole procedure. The author of [Luebbe01] names following approaches for deriving a starting basis:

- The *traditional approach* is to employ preprocessed values from a two-phase-Simplex.
- When performing an *artificial start*, a set of artificial variables is introduced, as many as constraints in the RMP. They are combined with some large penalty constant, in order to be eliminated in the latter solution process. The artificial columns form a feasible basis matrix which variables one by one exit the basis until all artificial variables are swapped out and a feasible solution is yielded.
- Often *primal heuristics* are applied, but in most cases they are problem specific.
- Sometimes, a *warm start* from primal solutions obtained in earlier, similar runs, is performed.
- A *trivial initialization* with a set of 0, the *unit basis* is often encountered.

In the most of the cases software solvers utilize general methods, since they will not know beforehand or even detect the particular problem structure. Good starting sets reduce the *heading in effect*, meaning that they produce little irrelevant columns for the latter solution process, as for example the artificial start. Bad ones may *"lead the algorithm astray"* [DesLu05]. The author concludes that best results are obtained with estimates of primal and dual solutions.

3.4.5 Solution Methods for the RMP

Although the Simplex algorithm is the dominant solution method for column generation problems, other approaches can be employed as well. Alongside primal-, dual-, and primal-dual-Simplex also specifically tailored algorithms for particular characteristics of the underlying linear program come to use. Nonlinear approaches like the barrier/interior point method (see section 3.2.4.3) as well as approximating approaches are very popular. Some of them are shortly outlined in the following [Luebbe01]: The *sprint method* divides the linear program into working sets, which are solved independently from each other. By combining the best results for each working set a solution to the main problem is approximated. The related *sifting method* combines interior point methods with a Simplex algorithm, the latter making a great progress for column selection in the beginning, whereas the interior point method then calculates or approximates the few remaining columns. When no exact solution is needed, the *volume algorithm* deduces quickly primal and dual solution approximates. Another approximating algorithm is *dual coordinate search*, which is especially designed for solving binary integer programs. It tries solving a RMP without relaxation and incorporates cost function information in combination with Lagrangian methods. The *analytic center cutting plane method* uses "central prices" and is based onto logarithmic barriers.

Generally the methods may perform very differently in varying problem types and the best practice has to be carefully selected, since there exists no formula that indicates the quality of a solution method. Additionally all kinds of heuristics for improving dual variables can be employed throughout the entire process.
Traditionally Simplex, sprint/sifting and interior point method perform very well in the most cases. If the aim is maximal efficiency the adaption of specialized methods should be considered. Main decision parameters for method choice are: efficiency, performance and speed of the algorithm, needed accuracy of the solution, problem peculiarities, available analytic and implementation skills, available industrial solver software.

3.4.6 The Pricing Problem - Pricing Strategies

The purpose of the pricing step is, as presented in 3.3.3.4, either to determine a column to be added to the restricted master problem, i.e. to select a new variable to enter the basis or to bring the evidence that no such one exists. The author of [Luebbe01] distinguishes between *pricing scheme* and *pricing rules*, the former representing the "(sub)set of non-basic variables to consider", and the latter being a "criterion according to which a column is selected from the chosen (sub)set". The most known such scheme-rule pair is the *Dantzig approach*: Choose among all the columns the one with the *most negative reduced cost coefficient* $\bar{c}^* = \min\{c_q^k - u^\top a_q^k - v_k \mid k \in K, q \in Q_k\}$, $(u, v)^\top$ being dual solutions for the RMP [Luebbe01], u being the dual variables for the so called linking or joint constraints.

Depending, if all variables, a subset of variables, or multiple considering of variables occurs, the according pricing schemes are called *full*, *partial* and *multiple pricing*.

For the pricing of linear programs following schemes exist: For (small) problems with some special structural properties, it is possible to determine the pricing variables by *enumeration* or *dynamic programming*[12]. In the majority of cases such methods are problem specific and poorly generalizable. An option would be to encode the pricing problem into an optimization problem as above. The formulation of a *single pricing subproblem* is

$$\bar{c}^* = \min\{\left(c^\top - u^\top A\right) x \mid Dx \leq d, x \geq 0\} \text{ [Luebbe01].} \tag{3.10}$$

When selecting columns there has to be taken into account that some columns could be redundant or dominated by other ones: The concepts *dominance* and *redundance* state a condition (in this case of the dual constraints) called *strength*.

[12] As in the parade example cutting stock problem.

Definition 6 (**Dominance**). *"A column k of model [M] is **dominated** if there exists another column $l \in Q$, such that the reduced cost of $k, \bar{c_k}$ is greater or equal to that of l, $\bar{c_l}$, for all dual values $(\pi, \mu, \nu) \in \mathbb{R}^n_+ \times \mathbb{R}^1_- \times \mathbb{R}^1_+$. On the other hand a column k is **undominated** if for all columns $q \in Q$, there exists a set of dual values $(\pi, \mu, \nu) \in \mathbb{R}^n_+ \times \mathbb{R}^1_- \times \mathbb{R}^1_+$ for which $\bar{c_k} < \bar{c_q}$." [Vanbk94], page 80.*

Definition 7 (**Redundance**). *"A column a_s is called redundant if the corresponding constraint is redundant for the dual problem. That is: $a_s = \sum_{r \subset s} a_r \lambda_r$ and $c_s \geq \sum_{r \subset r} c_r \lambda_r$. A column is* strictly *redundant if (above) holds with strict inequality." [Vanbk94]*

A strategy to handle dominated columns is: If a dominated column is discovered after the pricing step, replace this column by the dominant one in a post-processing step. Remove redundant constraints in the dual problem.

On page 56 in [Luebbe01] are summarized a variety of *alternative pricing rules* for LP's:

- *Steepest-edge pricing* eliminates the greedy character of the Dantzig pricing (choose a direction of steepest gradient) by taking into account in the pricing process all the edges and their directions.
- *Deepest-cut pricing* tries to cut away as much as possible from the dual solution space.
- *Lambda pricing* selects columns by normalizing the reduced cost.
- *Lagrangian pricing* exploits the original formulation by taking into account variable interactions, then falls back to the local reduced cost criterion. The method is mostly not applicable in complex programs but may perform well in problems with certain structures, for example vehicle routing.

Some additional pricing problem considerations are annotated: Whenever pricing is utilized, it has to be considered that every column with negative reduced costs improves the objective function, and therefore not all pricing problem solutions are finally required in the optimal problem solution. Also a pricing problem can be approached by approximation, since except for the final value no exact values are needed in the RMP. According to a many reports in literature (see [Vanbk94] and [Luebbe01] for references) the column generation algorithm may spend more than 90% of CPU time in pricing problems. Thus, this bottleneck has to be taken into account with careful planning since a slow pricing problem increases the execution time significantly.

3.4.6.1 Pricing Integer Programs

In integer problems, pricing is a little bit trickier. In fact, if the original compact formulation is an IP, the resulting pricing problem is one too, which has to be taken into account. Possible arrangements would be:

- Get lower and upper bounds through approximations and relaxations with a small duality gap between them. Use branch-and-price and divide and conquer solution space.
- Penalize the violation of integrality constraints, i.e. try to construct columns with integral coefficients, for example the *penalty function method*.
- Divide the problem into smaller subproblems, that are solved integrally and independent from each other, and adjoin the partial solutions together to form an ultimate integral solution. This strategy may involve structural characteristics of the problem itself.

3.4.7 The Tailing Off Effect

The *convergence* in IP column generation was also observed. Despite the algorithm is considered to be finite[13], sometimes it exhibits a very poor convergence behaviour, depicted in figure 3.9.

[13] Presupposed it does not cycle.

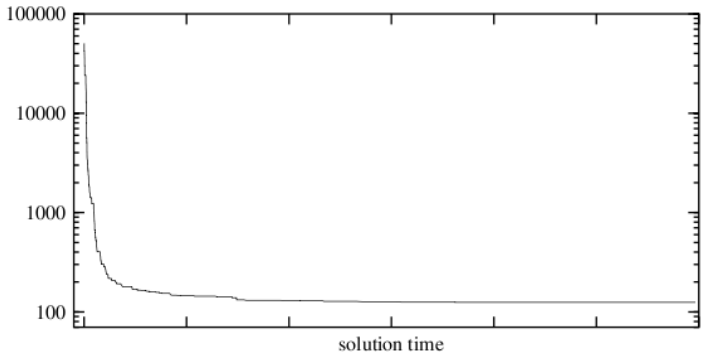


Figure 3.9: "The depicted development of the objective function value is typical for the tailing off effect." Image credits to [Luebbe01].

The algorithm comes close to a *near optimal solution* relatively fast, but thereafter only little advance is made and the progression in the distinct iterations is only very small. The dual variables have been observed to oscillate irregularly, rather than smoothly converge to an optimum. Also a time consuming proof of optimality (of a degenerate optimal solution) could occur. This behaviour in literature is often described as "exhibiting a long tail" and hence *tailing off effect*. According to [Luebbe01] at this point in time the entire process is only intuitively assessed and theoretically understood only in part and the absence of this oscillation as "(possibly *the*) desirable property" [DesLu05]. Main question is why an algorithm derived from the effective Simplex method, which only in exceptional situations produces long tails, can get so slow? Numerical instability and inaccuracy are discussed as causes as well as the big number of near optimal solutions in huge integer programs.

Generally outlined, remedies should aim for an *early termination*, e.g. the termination before the long tail occurs with a solution of assured quality. Also Devex pricing, introduced in 3.2.4.2, was observed to outperform the classical Dantzig pricing scheme. Another strategy would be to reformulate the problem and thus narrowing the set of considered columns. A simple idea is to apply quickly derived upper and lower bounds as well as valid inequalities and observe attentively the duality gap.

Following this last strategy, a *stabilized column generation* was developed, called *Boxstep algorithm*. It follows the bounding strategy and bounds the dual variables with an upper and a lower bound, that form a sort of box around the variables itself. The optimal solution is obtained in the interior of this boundary. Else, if an intermediate solution lies at the box boundary, the box is shifted towards its position while re-optimizing. Extensions to the Boxstep algorithm like the *Trust Region Method* generalize the box idea, parameters and use additional box constraints or penalize "bad" columns, as the *Weighted Dantzig Wolfe Decomposition*. The author of [Vanbk94] is confident that stabilized column generation produces large performance gains.

3.4.8 Integer Solutions

The explicit way to yield integer solution remains to be discussed, since until now we described only methods for maintaining the integer nature in the solution processes. Naturally, we do not want to depend on luck for integer value solutions for a RMP and avoid unsystematic rounding of some fractional values. The only approach for a systematical search in the solution space is *branch-and-price*, essentially presented in section 3.3.3.5 as well as *branch-and-cut-and-price*, section 3.3.3.6.

3.4.8.1 Branch-and-Price and Branching Schemes

The method was introduced by [NemWol88]. In [VanWol96], the author refers to it as *integer programming column generation*. Further literature is [Wol98] and [Vanbk94].

As already mentioned in subsection 3.3.3.5, branch-and-price builds a BB decision tree, where at each node the relaxation of the examined problem is solved. At each node the algorithm subdivides the solution space in a treelike fashion and excludes from search the subtrees where no optimal solution will be obtained for certain. This is done by means of bounds: If subtree S_1 has an upper bound inferior to a lower bound of another subtree S_2, it will for certain not yield a solution better than S_2 and can be pruned.
One crucial point in context of IP is how the problem is divided into subproblems, e.g. the used *branching scheme*. When having fractional intermediate solutions the subproblems can be built as follows:

- A standard branching schema is to split the solution space into two complementary subspaces, one having $x_i' \leq \lfloor value \rfloor$ and the other $x_i'' \geq \lceil value \rceil$.
- A very common branching scheme in binary integer programs (0-1 column generation) is: When having a fractional binary variable, the subproblems are constructed by dividing the solution space into one containing the respective variable $x_i' \geq 1$ and one not containing it $x_i' \leq 0$. Here it has to be ensured, that variables already set to 0 are not branched on again in the subsequent process. Unfortunately such a strategy branches very asymmetrically. Moreover, since constraints have to be added to the original problem, the original problem structure could be altered or become unsolvable. In [Vanbk94] this scheme is presented as *variable fixing branching scheme*.
- Both, [VanWol96], page 157 and [DesLu05] present the branching scheme developed by Ryan and Foster in 1981. If λ is a fractional solution, there exists a pair of rows $r, s \in \{1, \ldots, m\}$ such that $0 < \sum_{j \in J'} a_{rj} a_{sj} \lambda_j < 1$. We branch by setting once $\sum_{j \in J'} a_{rj} a_{sj} \lambda_j = 0$ and remove all columns with $a_r = a_s = 1$ by adding a subproblem constraint $x_r + x_s \leq 1$. The other branch is formed $\sum_{j \in J'} a_{rj} a_{sj} \lambda_j = 1$ and remove all columns with $a_r + a_s = 1$ by adding a subproblem constraint $x_r = x_s$.

Good and valid branching schemes partition the solution space in a way so that the current fractional solution is excluded. It leaves integer solutions intact and ensures the finiteness of the algorithm [DesLu05]. A general advice is to branch on *meaningful* sets of variables and make important decisions in the beginning of the branching process. Also a somewhat balanced tree should result from branching process, which is achieved by subdividing into branches of possibly similar sizes. The algorithm should be prevented to branch on variables already branched upon before. Note that bad branching and poor planning can lead to highly unbalanced trees and a multiplication of the tailing off effect.

We shift again to branch-and-price in general: Into its procedures, a rich reservoir of fine tuning can still be embedded:

> "All strategies from standard branch-and-bound apply, including depth first search for early integer solutions, heuristic fathoming of nodes, rounding and (temporary) fixing of variables, pre- and post processing, and many more." [DesLu05].

One big difference between IP and LP is that LP can be solved mostly in polynomial time (in $\mathcal{P}$) whereas IP are mostly $\mathcal{NP}$-hard. Since fast solution methods exist to deal even with comparatively vast problems, column generation for LP is deployed chiefly in especially huge problems, where solvability by use of the Simplex algorithm is not possible anymore. Column generation has the advantage of being especially designed for and applicable to big problems.

Chapter 4

Formal Problem Definition

Having summarized all theoretical prerequisites in the previous chapters, we proceed with the actual problem. Starting from the fingerprint minutiae data points, the process of encoding this data into a tree-like structure is described more in detail now. The goal is to compress this set. The result must be calculated efficiently in a reasonable amount of time. Further data processing of the compressed data should be possible.
First the tree based compression model is formally presented, including the central concept of the codebook and a description of the solution. It follows the formulation as a k-node minimum label spanning arborescence (k-MLSA), out of which the actual codebook is determined. Since we discuss optimal solution strategies, we present an integer linear program for k-MLSA, that [ChwRai09] solves by branch-and-cut. This ILP is the basis for the flow network formulation that will be solved by branch-and-price. All definitions adhere to [ChwRai09], who established the model solved by branch-and-cut.

4.1 Compression Model

In [ChwRai09], the authors define the n raw minutiae as d-dimensional points, forming a set $V = \{v_1, \ldots, v_n\}$. These points lie in a discrete domain $\mathbb{D} = \{0, \ldots, \tilde{v}^1 - 1\} \times \cdots \times \{0, \ldots, \tilde{v}^d - 1\}, \mathbb{D} \subseteq \mathbb{N}^d$, whose limits, denoted by $\tilde{v}^1, \ldots, \tilde{v}^d \in \mathbb{N}$, encode the individual sizes of each of the d dimensions[1].

As further input, a small parameter $\tilde{\delta} \in \mathbb{D}'$ is specified, with $\mathbb{D}' = \{0, \ldots, \tilde{\delta}^1 - 1\} \times \cdots \times \{0, \ldots, \tilde{\delta}^d - 1\}$, and $\mathbb{D}' \subseteq \mathbb{D}$. This is the domain of the correction vector, defined later. In our scenario, we constrain $\tilde{\delta}$ to $\tilde{\delta} < \frac{\tilde{v}}{2}$.

Onto the points V we define a complete directed graph $G = (V, A)$, V being the node set and A being the arc set $A = \{(i, j) \mid i, j \in V, i \neq j\}$. Each arc in this graph represents the relative geometric positions of the n points, meaning that each arcs end point encodes the relative position to the start point. As described in section 1.2.1 and in [ChwRai09], we extract out of this graph k of the n points, and $k-1$ specially selected arcs, which form an outgoing arborescence. Again, each arc in the arborescence encodes the relative positions of the points. The compression correlates with k in the following way: If $k = n = |V|$, a lossless compression is achieved. For any lower value of k a lossy compression is obtained.

Here a small set of specially selected *template arcs* comes into play. In order to achieve compression, we express each arc from the arborescence with an appropriate template arc ID, instead of storing each tree arc vector in full length for all d dimensions. We select for each tree arc one template arc, which resembles the tree arc the most and replace in the encoding the tree arc with a reference to the template arc. Since a small deviation from the tree arcs end point can occur we

[1] In our case the data is 4-dimensional, and hence $d = 4$.

further need a *correction vector* from the small domain $\mathbb{D}'$, to encode this deviation. Correction vectors were already introduced in section 1.2.1, figure 1.3.

In the end, all k nodes from the arborescence are connected to the root node by $k-1$ arcs, where each of these arcs is represented by its most similar template arc and an additional correction vector. The set of involved template arcs is called *codebook* or *dictionary*, depicted in figure 1.2. Compression is achieved either by minimizing the size of the codebook of template arcs, or by minimizing the correction vector domain, or by a combination of both alternatives. In this thesis, we concentrate on the first strategy, and enable compression by a small codebook, which size we consequently are going to minimize. An alternative would be to fix the size of the codebook to some prespecified value and minimize the domain of the correction vector.

In [ChwRai09], the solution to our problem is defined more formally. It consists of:

- A codebook of template arcs $T = (t_1, \ldots, t_m) \in \mathbb{D}^m$ of arbitrary, but minimal size m.
- A rooted, outgoing tree $G_T = (V_T, A_T)$ with $V_T \subseteq V$ and $A_T \subseteq A$, connecting $|V_T| = k$ nodes, in which each tree arc $(i,j) \in A_T$ has an associated template arc index $\kappa_{i,j} \in \{1, \ldots, m\}$ and a correction vector $\delta_{i,j} \in \mathbb{D}'$.

For two points v_i and v_j, connected by a tree arc $(i,j) \in A_T$, the following condition must hold:

$$v_j = \left(v_i + t_{\kappa_{i,j}} + \delta_{i,j}\right) \mod \tilde{v} \;\; \forall\, (i,j) \in A_T. \text{ [ChwRai09]} \tag{4.1}$$

The points to be encoded are transformed into the finite domain by a modulo operation in order to eliminate negative values and gain a finite ring, where domain borders must not be explicitly considered. The point v_j is calculated by adding to v_i the corresponding template arc and correction vector.

Having finally a codebook and a k-node tree, we store this tree and the template arc set as compressed information. We traverse the tree by depth first search and save our path as a bit sequence. Each time a new arc is traversed to reach an unvisited node we add 1 to our bit sequence. When following such a new arc, we save a reference to its representing template arc and associate a correction vector. In the case we have to backtrack along one arc, we write 0. Thus, our encoding finally contains the number of nodes k, the size of the codebook m, the domain limits $\tilde{v}$, the position of the root node where the path begins, a bit sequence that encodes the tree structure, the codebook of template arcs, and finally the remaining tree information. This tree information is a list of arcs, encoded by an index representing a template arc, the respective correction vector and the values of dimension values that were not considered for compression. Thus, we enable a good compression of the input data. Our goal is to calculate a minimal codebook, containing m template arcs, which is able to feasibly encode the input data set w.r.t. the correction vector domain. In the following we describe the steps for obtaining this smallest possible codebook. These steps will be improved in the further process, since branch-and-price needs not to precalculate the set of specially selected template arcs anymore.

4.2 Previous Approach: Selection of Candidate Template Arcs and Solving the k-MLSA Problem

In order to find a minimal codebook, the branch-and-cut approach requires an intermediate step. We mentioned the need for a set of specially chosen template arcs. This set is the set of candidate template arcs T^c, actually determined by *preprocessing*. Most part of this section was adopted from [ChwRai09], who formally introduced all terms and definitions.

4.2.1 Standard Template Arcs and Dominance

Basis for the preprocessing is the set of difference vectors of the input data points, calculated by:

$$B = \{v_{ij} = (v_j - v_i) \mod \tilde{v} \mid (i,j) \in A\} = \{b_1, \ldots, b_{|B|}\}$$

When considering the restricted domain $\mathbb{D}'$ for correction vectors, the template arcs $t \in \mathbb{D}$ define a subspace, where the vectors lie, that t represents: $D(t) = \{t^1, \ldots, (t^1 + \tilde{\delta}^1 - 1) \bmod \tilde{v}^1\} \times \cdots \times \{d^d, \ldots, (t^d + \tilde{\delta}^d - 1) \bmod \tilde{v}^d\}, D(t) \subseteq \mathbb{D}$. Thus for each template arc $t \in \mathbb{D}$ a subset $B(t)$ of vectors from B, $B(t) \subseteq B$, of represented vectors is established: $B(t) = \{b \in B \mid b \in D(t)\}$.

So, with respect to the $\tilde{\delta}$ constrained domain $\mathbb{D}'$, for each template arc a set of represented vectors $B' \subseteq B$, $B' \neq \emptyset$ can be derived. The vectors in B' are labeled $b'^l_1 \leq b'^l_2 \leq \cdots \leq b'^l_{|B'|}$ for dimension $l = 1, \ldots, d$. Based on this set B', the *standard template arc* τ is defined as:

> **Definition 8.** *"The* standard template arc *for B' is $\tau(B') = \left(\tau^1(B'), \ldots, \tau^d(B')\right)$ where $\tau^l(B') = b^l_{i^*_l}$ with $i^*_l = \arg\max_{i=1,\ldots,|B'|} b^l_i - b^l_{i-1} \;\; \forall\, l = 1, \ldots, d.$" [ChwRai09]*

B' spans a *smallest bounding box* $BB(B')$. It is a subspace of B' and includes all vectors from B', respecting the ring structure: $BB(B') = \{b^1_{i^*_1}, \ldots, b^1_{i^*_1 - 1} \mod \tilde{v}^1\} \times \cdots \times \{b^d_{i^*_d}, \ldots, b^d_{i^*_d - 1} \mod \tilde{v}^d\}$. This expression is shortened by: $\hat{\tau}(B') = (b^1_{i^*_1 - 1}, \ldots, b^d_{i^*_d - 1})$.

The standard template arc $\tau(B')$, illustrated in figure 4.1, is the corner point with the biggest coordinates of the smallest bounding box of all vectors in B'. $\hat{\tau}(B')$ is the corner point of the bounding box opposite to $\tau(B')$. In other words, the points from B' define a small region where all possible template arcs that express the same points B' lie. The standard template arc $\tau(B')$ lies at the "upper- and rightmost" corner, whereby $\hat{\tau}(B')$ is positioned at the opposite "lower- and leftmost" position of this bounding box.

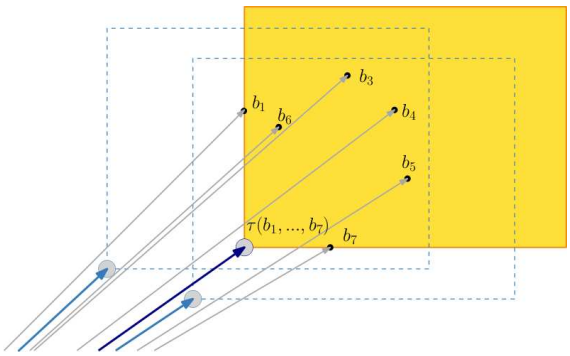


Figure 4.1: "The big gray dots are three of the possible representants for the tree arcs $b_1, \ldots, b_7$, but the standard template arc τ is the lower left point of the shaded rectangle. The rectangles depict the $\tilde{\delta}$ domain." Image credits to [ChwRai09].

In [ChwRai09], the authors establish the following lemmas including the proof:

> **Lemma 2.** *"If a subset $B' \subseteq B$ of vectors can be represented by a single template arc, then the standard template arc $\tau(B')$ is always such a template arc."*

> **Lemma 3.** *"A set $B' \subseteq B$ can be represented by a single template arc, particularly by $\tau(B')$, if: $\tilde{v}^l - (b^l_{i^*_l} - b^l_{i^*_l - 1}) < \tilde{\delta}^l, \quad \forall l = 1, \ldots, d.$"*

Based onto these lemmas, the set of standard template arcs T is restricted to the set induced by all nonempty subsets of vectors that can be represented by a single template arc: $T = \{\tau(B') \mid B' \subseteq B, B' \neq \emptyset \wedge B' \subseteq D(\tau(B'))\}$.

Definition 9 (**Domination of Template Arcs**). *"Let $t' = \tau(B')$ and $t'' = \tau(B'')$, $B' \subseteq B$, $B'' \subseteq B$. The standard template arc t' dominates t'' if and only if $B'' \subset B'$." [ChwRai09].*

By introducing this concept of domination, the set T is restricted further by discarding all dominated vectors. The resulting set of all *non-dominated template arcs* is the set of candidate template arcs T^c. It is determined in the next step, the preprocessing.

4.2.2 Preprocessing

So, the aim of the preprocessing is to provide this set of candidate template arcs T^c. This set T^c is derived out of the set of difference vectors B, from which all possible subsets B' are derived. A candidate template arc depends on the correction vector domain $\tilde{\delta}$, which induces the set $BB(B')$ for each tested vector. So, the check $b \in BB(B')$, with $b \in B$, is performed. The standard template arc $\tau(B')$ is derived from the bounding box $BB(B')$.

The algorithm partitions B into three disjoint index sets. The first set contains difference vectors b which are part of the currently considered bounding box $BB(B')$. The second holds all vectors that until now were actively excluded from being part of $BB(B')$. The third consists of all vectors to be yet considered. When regarding a current vector and no further addable represented vectors can be found, it is added to the solution set T^c. If there exist still representable vectors, branching occurs and by recursive calls these further vectors are identified. For each vector it has to be determined if it is dominated by another vector, or if in turn dominates others, by constantly updating the set of currently found T^c. In this manner a restricted enumeration is performed. This algorithm has the complexity $\mathcal{O}(d \cdot |B|^{3d})$ [ChwRai09] and in practice turns out to be the bottleneck of the codebook determination.

The set T^c is smaller than the set of all possible template arcs and generated to contain the vectors of the optimal solution. A lower bound for the size of T^c is 1, which means, that only one single template arc suffices to represent all vectors $b \in B$. As upper bound [ChwRai09] give $\mathcal{O}(|B|^d)$, and construct a worst case example consisting of a regular grid of non-dominated template arcs, which in practice should never occur. The authors expect that $|T^c| \ll \Theta(|B|^d)$.

4.2.3 Solving the k-Node Minimum Label Arborescence Problem

Having now T^c available, we proceed by determining the actual minimal codebook subset. For this, the graph G is extended by assigning to each arc $(i, j) \in A$ all template arcs $T^c(a_{i,j}) \subseteq T^c$ that are able to represent it w.r.t. equation (4.1). The next step is to extract a minimal subset $T \subseteq T^c$, where T must enable a feasible minimal tree encoding.

This problem of finding the minimal codebook is modeled as a variant of the Minimum Label Spanning Tree Problem, presented in section 2.1.1. With the use of a MLST we search for a minimum set of labels describing a spanning tree out of an undirected graph. Consequently, we define the arcs from T^c as labels. In contrast to the MLST problem we have to consider a complete directed graph, and allow multiple labels for the arcs. Further we only need a subset of k nodes to form our spanning tree. For matching purposes, a k of 12–20 minutiae is considered as sufficient by [SalAdh01]. For modeling k, additionally we use a k-cardinality tree (see section 2.1.2), whereby we need only $k - 1$ edges. The resulting problem is the k-node Minimum Label Spanning Arborescence Problem (k-MLSA).

Existing algorithms to the k-MLSA problem include heuristics, a memetic algorithm and an exact branch-and-cut approach. Focusing on the latter method, a few details from the entire process are summarized straightforwardly. First a directed cut formulation is established. This

model correlates to the k-cardinality tree problem formulation, solved by [ChiKa08] with branch-and-cut. The resulting directed cut formulation will be presented in the subsequent section 4.2.4. The ILP has an exponential number of inequalities and may not be directly solvable for big instances. Thus, branch-and-cut is applied, by starting with a restricted set of inequalities and subsequently adding newly separated inequalities in order solve it optimally. The cutting plane separations are organized in a branch-and-bound tree. The directed cut ILP is the basis for our branch-and-price approach.

4.2.4 Directed Cut Formulation for Branch-and-Cut

An integer linear programming formulation to the k-MLSA, based on directed connectivity cuts is presented in [ChwRai09]. In order to establish the model, the set of nodes V is altered to V^+, which contains an artificial root node with index 0. This is done in order to determine a root for the spanning tree, since we are interested in an outgoing arborescence. Additional outgoing arcs from this artificial root $(0, i), \forall i \in V$ are added to the set of edges A, which becomes A^+. Each template arc $t \in T^c$ represents a set of tree arcs, which we call $A(t) \subset A$. Further, $T(a) = \{t \in T^c \mid a \in A(t)\}$ is the set of template arcs that may represent a tree arc $a \in A$.
Following variables have been defined:

- $y_t \in \{0, 1\}$ encodes the candidate template arcs $t \in T^c$, the boolean value indicating t to be part of the codebook T or not.
- $z_i \in \{0, 1\}, \forall i \in V$ encode the covered tree nodes. The boolean value indicates which nodes belong to the tree.
- $x_{ij} \in \{0, 1\}, \forall (i, j) \in A^+$ encode the covered tree arcs. The boolean value indicates which arcs belong to the tree.

The complete ILP formulation for the k-MLSA is [ChwRai09]:

$$\min \quad \sum_{t \in T^c} y_t \tag{4.2}$$

$$\text{s.t.} \quad \sum_{t \in T(a)} y_t \geq x_a \qquad \forall a \in A \tag{4.3}$$

$$\sum_{i \in V} z_i = k \tag{4.4}$$

$$\sum_{a \in A} x_a = k - 1 \tag{4.5}$$

$$\sum_{i \in V} x_{(0,i)} = 1 \tag{4.6}$$

$$\sum_{(j,i) \in A^+} x_{ji} = z_i \qquad \forall i \in V \tag{4.7}$$

$$x_{ij} \leq z_i \qquad \forall (i, j) \in A \tag{4.8}$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall (i, j) \in A \tag{4.9}$$

$$\sum_{a \in C} x_a \leq |C| - 1 \qquad \forall \text{ cycles } C \text{ in } G, |C| > 2 \tag{4.10}$$

$$\sum_{a \in \delta^-(S)} x_a \geq z_i \qquad \forall i \in V, \forall S \subseteq V, i \in S, 0 \notin S \tag{4.11}$$

The objective function is given by equation (4.2) and states the number of template arcs, selected out of T^c to be *minimized*. Constraints (4.3) - (4.9) model the basis of the ILP: Inequalities (4.3) associate tree arcs with template arcs and enforce that for each selected tree arc x_a at minimum one template arc t is selected. With constraint (4.4) the number of required tree nodes is set to exactly k. Likewise, (4.5) constrains the number of required arcs to $k - 1$. We need these constraints (4.4) and (4.5) to make the selected edges and nodes form a minimum

spanning tree[2]. Next, we integrate the artificial root into the spanning tree, but again setting by equation (4.6) the number of outgoing arcs from this root node to some arbitrary tree node to one. Equation (4.7) defines for each node $v \in V$, that selected nodes can only have indegree 1, or else 0, if they are not selected. Further we must associate selected arcs and involved nodes: By constraints (4.8) we select only arcs, when their source node is selected.

Inequalities (4.9) are part of the *cycle elimination constraints*: These inequalities forbid any cycle of length 2. In order to also forbid cycles of greater length ($|C| > 2$), the inequalities (4.10) are added. The number of these cycle elimination inequalities is exponential, and therefore requires to be separated by cutting planes.

The directed *connectivity constraints* in (4.11) were introduced for strengthening reasons and are not immediately necessary: The constraints model a path from the artificial root node to any of the selected nodes in V. The node set S has the ingoing cut $\delta^-(S)$. Their number is also exponential.

The branch-and-cut algorithm starts with restrictions (4.2)–(4.9) and then successively separates and adds cycle elimination constraints as well as connectivity constraints.

Outlook

Having now defined all terms and prerequisites, the task is now to attempt the optimal problem solution with the aid of column generation techniques. Starting from this directed cut formulation, we proceed by formulating this ILP in such a way that it can be solved by branch-and-price. The following chapter 5 describes how this formulation is done.

[2]A spanning tree has n nodes and $n-1$ edges, in our case $k = n$.

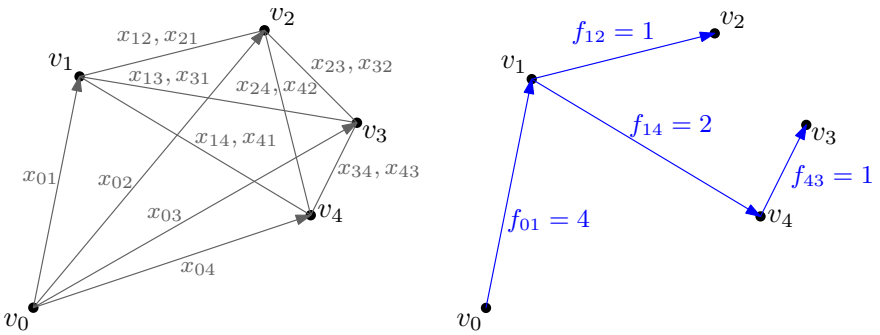
Chapter 5

Branch-and-Price

Based on the directed cut formulation from [ChwRai09] in section 4.2.4, we restate this model in terms of a flow network, so that it can be solved by branch-and-price. We describe more closely the branching and pricing process as well as the arising pricing problem. Section 5.5 outlines, how the solution to this pricing problem is defined. The actual solution approach to the pricing problem employing the k-d tree like structure is described in the following chapter 6.

5.1 Formulating the k-MLSA for Branch-and-Price

The branch-and-cut solution approach, presented in section 4.2.3, has one weak point that in the following shall be removed. The preprocessing step in practice consumes a relatively big amount of processing time, particularly for large input parameters $\tilde{\delta}$, which is due to the fact that the template arcs are able to express a huge number of tree arcs. Consequently our goal is to skip this preprocessing step and incorporate it into the linear programming part, which needs to be altered for this purpose. We model the k-MLSA by means of a flow network. Spanning tree formulations an corresponding polyhedra have been studied in detail in [MagWol94]. Also, in [MacPlLi03], flow formulations are discussed. Flow formulations usually have an polynomial number of flow variables and are therefore considered to be opportune. Figure 5.1 introduces the concept. It shows a flow network and a spanning tree induced by a single commodity flow network.



(a) Underlying arborescence (b) Single commodity flow spanning tree

Figure 5.1: Single commodity flow network in an arborescence.

In contrast to the MST problem no formulations corresponding to the convex hull of the incidence vectors of the solution are currently known for our problem. Although formulations based on directed connectivity cuts are known to provide tighter polyhedrons, we utilize a flow formulation to avoid the more complex handling of the exponential number of constraints of the former one.

5.2 Single Commodity Flow Formulation (SCF) for k-MLSA

The k-MLSA problem is now formulated in terms of a single commodity flow network. The number of flow variables as of constraints in this formulation is polynomial. Again we need an artificial root node to model the source of our flow network, since we do not know in advance which ones of the nodes will be selected to be part of the solution. We name this artificial root node r. The set of nodes V^+ again is V with an additional artificial root node r. The root node implies additional edges $(r,i), \forall i \in V$ to be added to the set of edges A, which becomes A^+. The variables x_{ij}, z_i remain as in the directed cut model:

- The boolean variables $x_{ij} \in \{0,1\}, \forall (i,j) \in A^+$ encode the covered tree arcs, its value indicating which arcs belong to the tree.
- The covered tree nodes are $z_i \in \{0,1\}, \forall i \in V$, respective.
- Further, since the set T^c is not any longer computed in a preprocessing step, we substitute it with T, the set of all possible template arcs. The variables $y_t \in \{0,1\}, t \in T$ encode in binary format the template arcs, that were selected for being part of the codebook.
- Additional flow variables are introduced. A variable f_{ij} indicates a real valued commodity flowing from node i to node j. Thus, $f_{ij} \in \mathbb{R}^0_+, \forall (i,j) \in A^+$.

$$\min \quad \sum_{t \in T(a)} y_t \tag{5.1}$$

$$\text{s.t.} \quad \sum_{t \in T(a)} y_t \geq x_a \qquad \forall a \in A \tag{5.2}$$

$$\sum_{i \in V} z_i = k \tag{5.3}$$

$$\sum_{a \in A} x_a = k - 1 \tag{5.4}$$

$$\sum_{i \in V} x_{ri} = 1 \tag{5.5}$$

$$\sum_{(j,i) \in A^+} x_{ji} = z_i \qquad \forall i \in V \tag{5.6}$$

$$x_{ij} \leq z_i \qquad \forall (i,j) \in A \tag{5.7}$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall (i,j) \in A \tag{5.8}$$

$$\sum_{(r,j) \in \Gamma^+(r)} f_{rj} - \sum_{(i,r) \in \Gamma^-(r)} f_{ir} = k \tag{5.9}$$

$$\sum_{(i,l) \in \Gamma^-(l)} f_{il} - \sum_{(l,j) \in \Gamma^+(l)} f_{lj} = z_l \qquad \forall l \in V^+ \setminus \{r\} \tag{5.10}$$

$$f_{ij} \leq (k-1) \cdot x_{ij} \qquad \forall (i,j) \in A \tag{5.11}$$

$$f_{rj} \leq k \cdot x_{rj} \qquad \forall (r,j) \in A^+ \tag{5.12}$$

$$f_{ij} \geq 0 \qquad \forall (i,j) \in A^+ \tag{5.13}$$

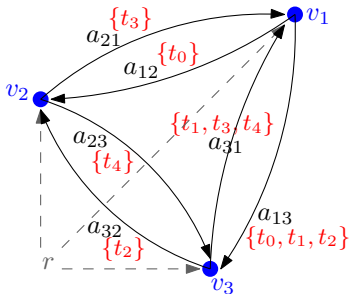
$$\sum_{t \in T(\Gamma^-(v_i))} y_t \geq z_i - x_{ri} \qquad \forall i \in V \tag{5.14}$$

The constraints (5.1) - (5.8) remain as in the directed cut formulation from subsection 4.2.4, but we replace T^c with the set T. The new constraints (5.9) - (5.13) substitute the cycle elimination and connectivity constraints and model a connected spanning tree by means of a single commodity flow. Inequalities (5.2) encode the tree arcs and the template arcs which can represent them. We denominate these constraints (5.2) as *arc-label constraints* and these are the constraints, that we are going to expand with variables by pricing.
The constraint (5.9) regulates the outflow from the artificial root node: Since there exist V outgoing

arcs from the root, but we defined only one of these arcs to be selected, a flow of exactly k must go out from this source. The following constraints (5.10) model "flow conservation" for each other node. The inflow must equal the outflow, but taking into account that at each node the flow of value 1 leaves the network.
Inequalities (5.11) link flow variables with tree arcs (linking constraints). The flow variables hold at maximum the value $k-1$. The variables x_{ij} for the solution are selected according to the flow variables. The only exception is the flow from the artificial root node. Going out from the root, we need a flow of exactly k, since we want to reach k nodes in our flow network, but defined only one arc $x_{rj} \in A^+$ to be selected. The situation is modeled by inequality (5.12). By additionally using inequalities (5.13) we constrain a flow to be $0 \leq f_{ij} \leq k$. Constraints (5.14), described subsequently, tighten our formulation in the primal.
In [CaClPa09] the authors show, that $x \in \mathbb{R}$ is sufficient to obtain a valid integer solution for the labels. Although likely, it is not clear if this property does also hold for the k-MLSA. We therefore restrict the x variables to a boolean domain. Computational tests support our assumption of real edge-variables already yielding valid label solutions also for the k-MLSA, however without a definite gain according to obtained run times.

5.2.1 Node-Label Constraints

Not all template arc are equally "good". Each template arc can represent a set of tree arcs $t_i = \{a_j | a_j \in A(t_i)\}$ which may be of the same size and seem equally suited. When we look at the template arcs and how they contribute to form a spanning tree, we see that not all template arcs come into question. So we embed information about the spanning tree. Figure 5.2 illustrates the concept. As we can see, some template arcs may certainly not contribute to form a spanning tree, e.g. by forming cycles, as t_3 in the figure.



$t_0 = \{a_{12}, a_{13}\}$
$t_1 = \{a_{13}, a_{31}\}$
$t_2 = \{a_{13}, a_{32}\}$
$t_3 = \{a_{21}, a_{31}\}$
$t_4 = \{a_{23}, a_{31}\}$

Figure 5.2: Correlation of tree arcs a and template arcs t (labels). When taking one of the template arcs t_0, t_2 or t_4, we get a spanning tree, whereby t_1 forms a cycle and hence no spanning tree. Also t_3 forms no spanning tree.

So, when selecting new template arc variables, we try to favor template arcs that enable spanning trees and penalize impossible template arcs. We introduce the constraints (5.14) for the template arcs variable y_t and denominate them as *node-label constraints*. With these constraints, the LP relaxation of our model is tighter. These constraints encode that the number of all template arcs that are incoming at node v_i must be greater equal one if the actual node z_i is part of the solution. Furthermore, since our k-MLSA is rooted, we have to take into account that one arc from the artificial root is always selected and therefore incoming at some node. At this node we have to subtract that incoming arc.
With these node-label constraints we provide additional dual variables, and use them in addition to the dual variables for arc-label inequalities (5.2), in order to favor good template arcs that contribute to form spanning trees and enhance thus the pricing strategy. In the branch-and-price process these constraints will be also expanded with new variables. This will be subject in section 5.5.

5.3 Multi Commodity Flow Formulation (MCF) for k-MLSA

The k-MLSA can also be formulated in terms of a *multi commodity flow network*. Hereby we have $|V|$ commodities $0 \leq f_{ij}^c \leq 1$ flowing through the network. The advantage of an MCF formulation is a better LP relaxation because of a tighter coupling of f_{ij}^c and x_{ij} when using multiple commodities. The disadvantage is the greater amount of variables and constraints.

$$\min \quad \sum_{t \in T(a)} y_t \tag{5.15}$$

$$\text{s.t.} \quad \sum_{t \in T(a)} y_t \geq x_a \qquad \forall a \in A \tag{5.16}$$

$$\sum_{i \in V} z_i = k \tag{5.17}$$

$$\sum_{a \in A} x_a = k - 1 \tag{5.18}$$

$$\sum_{i \in V} x_{ri} = 1 \tag{5.19}$$

$$\sum_{(j,i) \in A^+} x_{ji} = z_i \qquad \forall i \in V \tag{5.20}$$

$$x_{ij} \leq z_i \qquad \forall (i,j) \in A \tag{5.21}$$

$$x_{ij} + x_{ji} \leq 1 \qquad \forall (i,j) \in A \tag{5.22}$$

$$\sum_{j \in \Gamma^+(r)} f_{rj}^c = z_c \qquad \forall c \in V \tag{5.23}$$

$$\sum_{j \in \Gamma^+(i)} f_{ij}^c - \sum_{j \in \Gamma^-(i)} f_{ji}^c = 0 \qquad \forall i \in V \setminus \{c\}, c \in V \tag{5.24}$$

$$\sum_{j \in \Gamma^-(c)} f_{jc}^c - \sum_{j \in \Gamma^+(c)} f_{cj}^c = z_c \qquad \forall c \in V \setminus \{r\} \tag{5.25}$$

$$f_{ij}^c \leq x_{ij} \qquad \forall (i,j) \in A^+, c \in V \tag{5.26}$$

$$f_{ij}^c \geq 0 \qquad \forall (i,j) \in A^+, c \in V \tag{5.27}$$

$$\sum_{t \in T(\Gamma^-(v_i))} y_t \geq z_i - x_{ri} \qquad \forall i \in V \tag{5.28}$$

Again we have to restrict the flow from the artificial root node, but unlike the SCF formulation this has to be modelled for each commodity. We defined, that only one arc from the artificial root node r to some arbitrary node z_i can be selected. All c commodities must flow over this selected arc, by having the value one. We model this with the constraints (5.23). The following conditions (5.24) model flow conservation, but this time for each particular commodity. They state, that at each node, the inflow must be equal to the outflow for each commodity. Constraints (5.25) have been introduced to model the individual targets for each flow: A commodity c for flow f_{jc}^c has as target the node z_c, where again the commodity of value one leaves the network. Inequalities (5.26) associate a flow f_{ij}^c with an arc x_{ij} by determining it to be either selected when having a value of one, or deselected when having a value of zero. Finally, each of the c commodities is restricted by inequalities (5.27) to be ≥ 0. Again we can tighten the multi commodity flow formulation with node-label constraints (5.28).

5.4 Branch-and-Price

We formulated the problem as a single commodity flow network (section 5.2), and a multi commodity network (section 5.3). Both models have a polynomial number of flow variables. These

formulations are our master problem.

Both presented formulations can be solved entirely, without employing column generation. For this we substitute T with T^c, the set of non-dominated template arcs. The arc-label constraints (and node-label constraints) are build in advance by inserting *all* $y_t, t \in T^c$. This procedure is useful for determining if the models are correct. Usually, pricing is done when having an exponential number of variables, we nonetheless speed up the solution procedure of such a complete model by enabling pricing.

As already stated, the creation of candidate template arcs shall be interwoven into the integer program by only creating new template arcs on demand. As we perform column generation we start with a restricted master problem, which consists of a very small subset of variables or columns. Two alternatives for the construction of a RMP exist, which will be described subsequently.
The first option is to start with the MIP containing only a *feasible* small set of template arc variables, the starting set. This RMP forms a valid starting basis for the branching and pricing process. The feasibility is determined by solving the RMP and its relaxation. The feasibility condition is ensured by the starting template arcs which should enable to cover a sufficient number of nodes. The determination of this starting set will be described in section 5.6. The second option is to start with the MIP containing *no* template arc variables at all. The resulting MIP is *infeasible*.

One of the purposes of the RMP is to make available the dual problem variables. The solution of the LP relaxation of the RMP provides actual values for these dual variables. These values are then used to determine a column, which when added to the RMP, can improve its objective value. This is done via the pricing problem, which selects a variable that has minimal reduced cost, calculated out of the dual values. After having determined such a variable we add it to our RMP. Then, in the next iteration, the solution process is started anew by relaxing the new RMP, which in turn gives new values to the dual variables. Based on these new dual values we price again a new column and the process continues as long as new variables having negative reduced costs are found. In the case of an infeasible RMP the pricing algorithm must select columns in a fashion that the RMP becomes feasible.
The branch-and-price algorithm builds a decision tree and prices new columns at its nodes. Each tree node consists of a "configuration" of variable values. At each node the algorithm additionally determines upper and lower bounds for the number of selected template arcs and evaluates the gap between primal and dual solution values. With the help of these bounds the algorithm prunes subtrees in the decision tree, which exceed these bounds. In this manner the algorithm searches the tree efficiently for the optimal solution. If the gap between primal and dual values is zero, an optimal solution was found.
Implementation details will be presented in section 8. Before coming to this, we have to solve the pricing problem efficiently. The template arcs in k-MLSA have special geometric properties that will be exploited for this purpose.

5.5 Pricing Problem

As branch-and-price starts with the RMP, we have dual variables u_{ij} available. These variables are extracted based on the constraints (5.2),(5.16) which hold x_{ij} from the dual solution of the RMP in each branch-and-price iteration. The set $A(t)$ is the set of arcs (i, j) that are covered by a template arc t. If the according model employs node-label constraints (5.14) and (5.28), introduced in the previous subsections, we get additional dual variables μ_j. For each t the reduced cost is defined as:

$$\bar{c}_t = 1 - \left(\sum_{(i,j)\in A(t)} u_{ij} + \sum_{j\in\{v|(u,v)\in A(t)\}} \mu_j \right). \tag{5.29}$$

The solution to the pricing problem is the template arc t^* having the maximum sum of dual values. We add to the RMP the column that has maximal negative reduced cost.

$$t^* = \arg\min_{t \in T} \left\{ 1 - \left(\sum_{(i,j) \in A(t)} u_{ij} + \sum_{j \in \{v | (u,v) \in A(t)\}} \mu_j \right) \right\}. \tag{5.30}$$

If node-label constraints are not used, the corresponding variables μ_j are zero. Only variables with negative reduced costs can improve the objective function, as discussed in sections 3.3.3.4 and 3.4.6.
Now we need to determine values for each template arc variable, by regarding the values of the dual variables corresponding to a template arc. The idea is to build a *search tree*, successively named *segmentation tree*, wherein the template arcs and the points they represent, are encoded w.r.t. their geometrical information. This is done by means of a k-d tree, where in advance or dynamically we construct all branches or at least the branches needed for solving the pricing problem. By traversing the tree efficiently by appropriate bounding techniques we finally identify the solution to the pricing problem. All details for this step are described in chapter 6. As the pricing problem will be solved very frequently, the procedure must be very efficient. The dual variables are not needed for the construction of the segmentation tree, but are used only in the pricing process, where we determine the maximal negative reduced cost.

5.6 Determining a Starting Solution

There exist two alternatives for starting the branch-and-price process:

5.6.1 Trivial Starting Solution

If we want our initial RMP to be feasible, we must determine a valid starting solution first. Such a starting solution consists of a small amount of columns, that are a *spanning tree* or a *path*. Such a starting solution must contain k nodes in order to be feasible. It may be quickly derived, since it is only a helping starting solution for the actual minimization process. We use the star shaped spanning tree resulting when enabling the arcs $x_{0j}, j = 1, \ldots, k$ and x_{r0} and the nodes $v_0, \ldots, v_k$. We assign some big values to the dual variables $u_{0j}, j = 1, \ldots, k$ and determine the needed template arcs with the pricing algorithm. Thus we get a starting solution consisting of non-dominated template arcs.

5.6.2 Farkas Pricing

If we start with an infeasible RMP, without any template arc variables, we first have to make it feasible in order to begin the pricing process. With Farkas' lemma (subsection 3.2.3.1) we have a proof of infeasibility and we can extract the Farkas coefficients. These Farkas coefficients are then used in the same manner as the dual variables by performing *Farkas pricing*. With the pricing algorithm, a generic procedure that uses either dual or Farkas values depending on which pricing problem (standard reduced cost pricing or Farkas pricing) we want to solve, we determine the new template arc to be added to the RMP as the one having the maximal positive sum of Farkas values. As an RMP relaxation is determined to be infeasible, the algorithm, before starting the branch-and-price process, calls the Farkas pricing method. This method determines new columns and adds them to the actual RMP, until the RMP becomes feasible. Then, when finally having a feasible LP relaxation, the branch-and-price algorithm provides the dual variables for the current subproblem, the algorithm proceeds with the reduced cost pricing.

5.7 Pricing Methods

Once the data structures and algorithms that solve the pricing problem are implemented, the same procedure can be used for both the single and the multi commodity flow model to determine the

variable that is added to the RMP. The same algorithm enables Farkas pricing and determines columns that make some previously infeasible LP feasible. Subsequently we summarize some topics, which adhere to the branch-and-price process.

5.7.1 Standard and Multiple Pricing

We have two alternatives when adding columns. When performing *standard pricing* we determine in each pricing iteration the column having maximal reduced cost and add it to our RMP. After resolving the RMP relaxation we proceed in the same manner in the next pricing iteration until no further variable having negative reduced cost can be found. At the next branch-and-bound node, the algorithm proceeds in the same manner.

When doing *multiple pricing*, we determine *all* columns with negative reduced cost in one pricing iteration, and add them all to our RMP in one step. Only after all these variables were added, the RMP is relaxed and solved anew, before proceeding with the next pricing iteration. This is done in each branch-and-bound node, until no variables having reduced costs are found anymore.

5.7.2 Allowing Variable Duplicates

In the course of the branching and pricing process it may happen that some variables are found twice, or even more times in subsequent pricing iterations. When enabling such *variable duplicates* the algorithm may spend a lot of time with finding the same variable again and again. Such situations may occur due to the fact that the solution of the dual problem is not unique.

Adding duplicates each time they are found, makes the number of variables and thus the MIP very huge. The algorithm may spend much time with the pricing of plenty of duplicates. Although the problem is still solvable prevented this behaviour. Accordingly, each time we search for a new variable, we check if we already have found this variable and add another variable, if existing, with inferior reduced cost instead. This topic is also covered in chapter 9.

5.7.3 Algorithms

The overall branch-and-price algorithm is described by listing 5.1. This algorithm only outlines the procedure, which we will perform with the aid of a branch-and-price framework. We embed the parts for loading the model and implement the routines relevant to the pricing problem. Further we influence the solution process by selecting traversing strategy and according branching rules.

At line 1 the algorithm initializes the branch-and-bound tree and codebook. The variable z^* will save the best solution. At line 2, the algorithm loads the desired formulation, either the SCF or MCF model, determines a starting solution and adds it to the RMP. Naturally, when employing a Farkas pricer, the step for determining a starting solution is omitted. Line 3 generates new subproblems for the branch-and-bound process, and adds them to the branch-and-bound tree as nodes. Then, after having selected the next node and thus the subproblem to process (line 5), the pricer adds new variables t_{new} to the RMP as long as it finds some and relaxes/solves anew the RMP. While processing the nodes the algorithm descends into the tree by a traversing strategy (DFS or BFS) and selects the node to be processed next with the aid of a branching rule. While doing so, the algorithm determines the optimal solution by evaluating at each subproblem node the objective value, upper and lower bounds and dual gap. The solution strategy of the pricer is presented in the next chapter 6. In section 6.6, the instances for the abstract pricer algorithms are described.

Algorithm 5.1: Branch-and-Price(f, $\tilde{\delta}$, d, k)

Data: Input data file f, bounding box $\tilde{\delta}$, dimensionality d, nodes to connect k.
Result: Returns the optimal codebook for the input parameters.

```
1  bbtree ← ∅; codebook ← ∅; z* ← k;
2  RMP ← SCFModel and startingsolution; /* or MCFModel */
3  bbtree ← generate subproblem nodes n;
4  while bbtree has unprocessed nodes n do
5     Select next n ∈ bbtree with traversing strategy and branch rule;
6     /* Solve LP relaxation of RMP_n and determine dual values. */
7     u_ij, [μ_j] ← solve (RMP_n);
8     /* Based on u_ij, [μ_j] determine one or multiple new column(s) t_new. */
9     /* The algorithms for PricerAlgorithm are presented in section 6.6. */
10    for t_new ← PricerAlgorithm (u_ij, [μ_j]), where t_new ≠ ∅ do
11       if t_new has reduced cost c̄(t_new) < 0 then RMP_n ← RMP_n ∪ t_new;
12    z*_n ← solve (RMP_n);
13    if z*_n < z* AND z*_n valid for RMP_n then z* ← z*_n;
14    Prune nodes having z_nLB > z*;
15    bbtree ← generate new subproblem nodes n;
16 codebook ← extract template vectors for z*;
17 return codebook;
```

Chapter 6

Solving the Pricing Problem

This chapter describes the solution approach for the pricing problem in detail. First, we present the overall principle and proceed with defining a region abstraction, upon which the k-d-tree like structure is based on. In the following, we denominate this tree as *segmentation tree*. This chapter describes construction and management of this data structure and how we extract data from it. By traversing the segmentation tree systematically, we search for a node corresponding to the solution to the pricing problem.

6.1 Basic Ideas

The role of the pricing problem can be illustrated as follows. Imagine a rectangle of size $\tilde{\delta}$ defined by each point $b \in B$ in a finite 2-dimensional domain. This bounding box, with the point b at its upper, rightmost corner, defines an area in which a template arc, able to express this point, can lie. So by having such a rectangle for each b_i, these rectangles overlap. This results in areas where multiple rectangles overlap each other. These overlapping areas define template arcs, where more than one element b can be expressed and therefore corresponds to a template arc, which is able to express all points b_i associated with each one of the rectangles out of which the area was formed. In the two-dimensional case, the pricing problem can be illustrated as follows. If the bounding boxes are considered as transparent grey-shaded rectangles with a grey-tone corresponding to the value of the associated dual variable, overlapping regions imply darker areas. Our goal is to find the darkest areas.

6.1.1 Segmenting the Area by Means of a k-d Tree

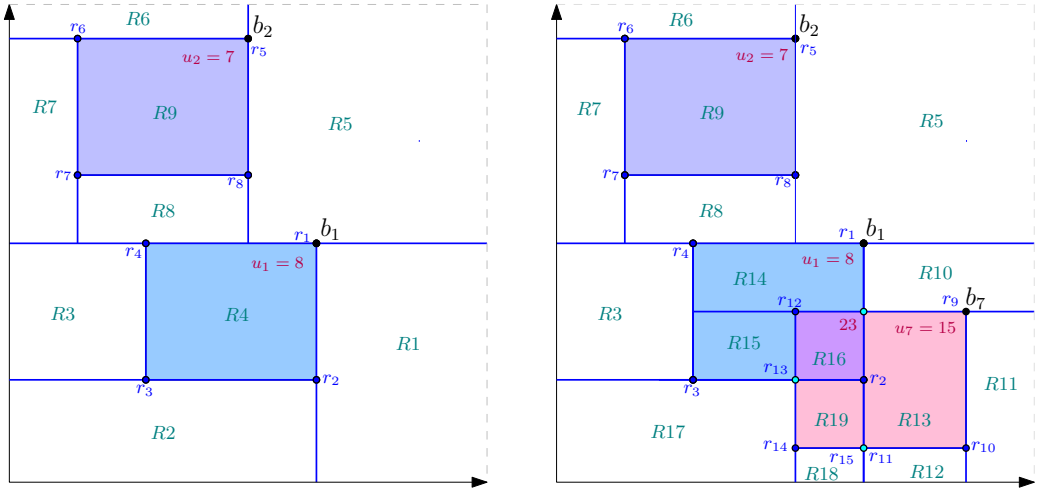
Input to the k-d tree are the difference vectors $b_i \in B$. In order to retain the meaning of k in the (usually so called) k-d tree naming, it must be annotated that we denote the dimensionality by d. As we defined k being the size of the subset of nodes from V forming the k-MLSA we must pay attention to not confuse both terms. Further input is the correction vector domain $(\tilde{\delta}^1, \ldots, \tilde{\delta}^d)$.

The basic idea for the solution of the pricing problem is to segment the finite domain $\mathbb{D}$ into subspaces in the fashion of a k-d tree. The division into subspaces is done by means of the *points* b_i and the *corner points of the* $\tilde{\delta}$ *sized bounding box* defined by each b_i:

$$(b_i^1, b_i^2, \ldots, b_i^d), (b_i^1 - \tilde{\delta}^1, b_i^2, \ldots, b_i^d), (b_i^1, b_i^2 - \tilde{\delta}^2, \ldots, b_i^d), \ldots, (b_i^1 - \tilde{\delta}^1, b_i^2 - \tilde{\delta}^2, \ldots, b_i^d - \tilde{\delta}^d).$$

The segmentation is done along the $(d-1)$-hyperplanes of the bounding boxes. Each point b_i implicitly defines 2^d corner points that represent the limits of the bounding box for each b_i. All relevant coordinate information is contained in the two points $(b_i^1, b_i^2, \ldots, b_i^d)$ and $(b_i^1 - \tilde{\delta}^1, b_i^2 - \tilde{\delta}^2, \ldots, b_i^d - \tilde{\delta}^d)$. Having such b_i, where each one defines a bounding box, we want to identify the areas where more bounding boxes overlap. We want to divide the finite domain $\mathbb{D}$ in a fashion,

so that its segments represent these areas. This we do by creating a splitting hyperplane successively for each relevant facet of each bounding box defined by a b_i. The required coordinates can be extracted from the two corner point vectors which uniquely define the bounding box. While dividing a segment into subsegments by inserting such a splitting hyperplane, we build the tree that encodes the structure. Having at some node a segment and a splitting hyperplane we get two subspaces, one that lies left or below ($<$) the actual splitting hyperplane, and one that lies right or above ($>$) the splitting hyperplane. We encode this into the tree by associating the left subspace with the left successor and the right subspace with the right successor. When performing the segmentation in such a way and process each b_i in random order, then the entire d-dimensional domain containing the points from B is structured into a tree. The general principle is depicted in figures 6.1, 6.2 and 6.3, where the nodes in the segmentation tree are labeled by r_i to emphasize the point where segmenting occurs. The images anticipate the bounding concept presented subsequently in section 6.1.2. At each node the first set is the upper bound set UB, the second set the lower bound set LB. The values u_i indicate some sample dual values for b_i and the bounding box defined by it. For overlapping areas we have a sum of dual values, derived from the according LB. The dual values may differ in each pricing iteration.



(a) Segmented area after insertion of b_1, b_2.

(b) Segmented area after insertion of point b_7.

Figure 6.1: Segmentation of a sequence of points b_i. Figures 6.2 and 6.3 show the corresponding trees.

Let now n denote the nodes of the segmentation tree. Further we denote the subspace corresponding to node n with $R(n)$. The subspace $R(n)$ is the area enclosed by the splitting hyperplane, which we denote by (c, dim). We can distinguish between two types of nodes:

- The *intermediate nodes* contain a splitting hyperplane and divide the subspace defined by $R(n)$ into two subspaces, positioned respectively in the left subtree n_{left} and right subtree n_{right} depending on their position to be left or right of the actual (c, dim).
- The *leaf nodes* hold and express *atomic* subspaces, which are not further divided.

These atomic subspaces are what we are actually interested in. When we segment the domain with the segmentation tree, then the leafs in the tree encode atomic subspaces where all template arcs contained in this area represent exactly the same points. That means that all template arcs that can express the same points correspond to vectors within this subspace. Every template arc within this subspace is equally suited and we only consider the corresponding *standard template arc* τ.

The solution to our pricing problem is the atomic subspace having the biggest sum of values of corresponding dual variables. Once such a subspace was found, we derive the appropriate standard

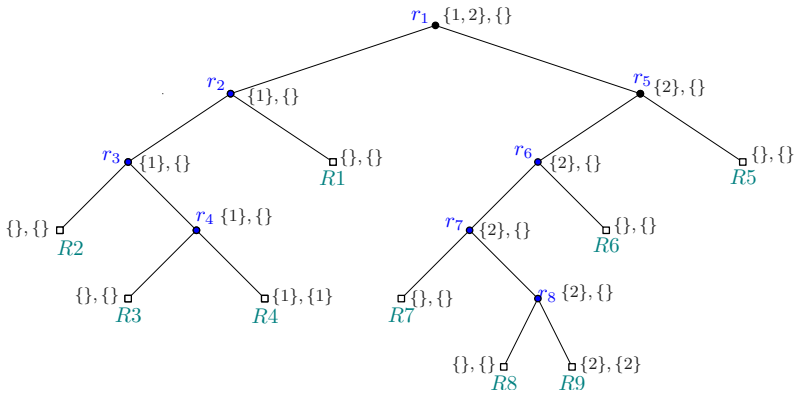


Figure 6.2: Segmentation tree after inserting b_1, b_2. A left son ≅ left, below; right son ≅ right, above.

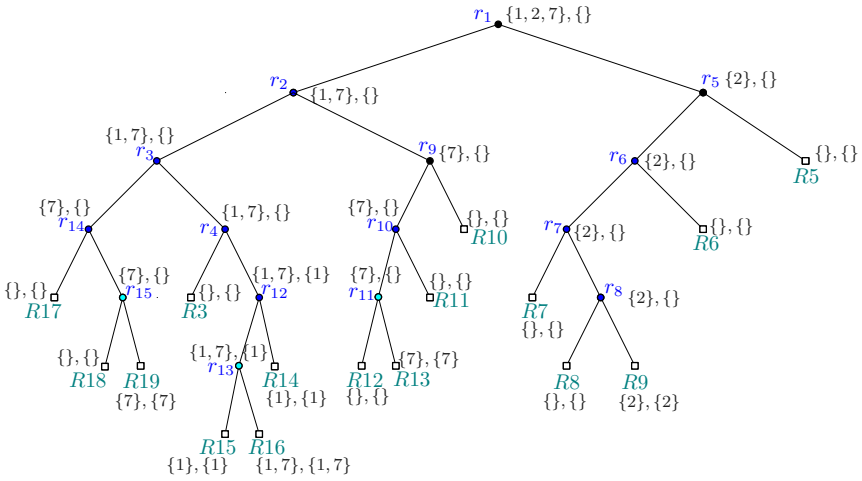


Figure 6.3: Segmentation tree after insertion of b_7. A left son ≅ left, below; right son ≅ right, above.

template arc τ from it. As we do not want to process all nodes in the tree in order to find the optimal solution, the search is restricted with bounding techniques, in a way that only interesting subtrees are traversed and the irrelevant ones are omitted.

6.1.2 Extracting Template Vectors by Bounding

In section 5.5 we defined the variable, that will be added to our actual RMP, as the one having the maximal sum of dual values, and thus maximal negative reduced cost $\bar{c}_t$. The dual values have now to be integrated into our segmentation tree. This is achieved by defining two sets for each node in the tree, each node representing a subspace $R(n)$ in which some points lie. Each set encodes the expressed points for the respective segment in the following manner:

Definition 10 (Upper Bound Set $UB(n)$)**.** *The set of points that could maximally be represented by the template arcs positioned in the respective subspace defined by the actual node:*

$$UB(n) = \{b \in B \mid \exists\, t \in R(n) \wedge b \in B(t)\}.$$

Definition 11 (Lower Bound Set $LB(n)$)**.** *The minimal set of points that can be represented by all template arcs positioned in the respective subspace defined by the actual node:*

$$LB(n) = \{b \in B \mid \forall\, t \in R(n) \wedge b \in B(t)\}.$$

As each point b_i has an associated dual variable u_i in the actual RMP, and there exist subspaces that can express more than one point[1], a numerical value for each segment can be calculated by computing $ub(n) = \sum_{i \in UB(n)} u_i$ and $lb(n) = \sum_{i \in LB(n)} u_i$. We use these bounds to restrict the search in our tree by bounding techniques, since we do not want to traverse the entire tree to find the variable that is the solution to the pricing problem. Let lb^* denote the global lower bound, i.e. the maximum $\bar{c}_t$ found so far. If $ub(n) < lb^*$ for some node n we do not have to follow this branch. The lower bounds $lb(n)$ are used to drive the search towards promising sections of the search tree. Also subtrees with an $UB \subset UB^*$ can be pruned.

Figures 6.2 and 6.3 contain already for each node n the upper bound set UB and lower bound set LB associated with the subspace $R(n)$. At each node UB is the first set, LB the second one. The optimum lb^* in 6.3 is the leaf $R16$ holding $UB = LB = \{1, 7\}$ with the dual value $ub = lb = 23$.

With the outlined procedure, the k-d tree like structure holds a segmentation for the finite domain $\mathbb{D}$ and with it the relations of the dual variables. Our segmentation tree encodes the relations of the node differences b_i in an upper and lower bound set at each node. For each node difference we can derive a dual value based on the current LP solution in each branch-and-price iteration. Special interest lies in the atomic subspaces at the leafs of the tree, where the upper and the lower bound are equal. The standard template arc τ at such a leaf is implicitly defined and calculated on the basis based on the set of node differences, as described in section 4.2.1 and depicted in figure 6.4. In the end, by traversing the tree in search for the leaf having the maximum maximal sum of corresponding dual values in its lower bound set, we identify the solution to the pricing problem, which is the standard template arc τ^*, that has maximal negative reduced cost. Summarizing, we can extract from the segmentation tree two types of results:

- The *solution to the pricing problem*: The standard template arc τ^* having maximal negative reduced costs. We obtain this result by searching in the segmentation tree for the leaf encoding an atomic subspace, having the maximal sum of dual values calculated out of its lower bound set.
- The *solution to the preprocessing*: The upper and lower bound sets may be used for a second purpose as well. The approach is an alternative to the existing preprocessing, and extracts the set of non-dominated template arcs T^c.

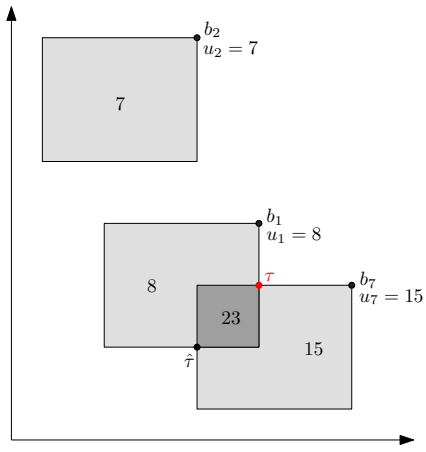


Figure 6.4: A standard template arc τ for the segmentation in figure 6.1b.

6.1.3 Using the Segmentation as an Alternative to Preprocessing

So, besides the solution to the pricing problem τ^* we can extract from this segmentation tree the entire set of non-dominated template arcs T^c. The idea is to build a segmentation tree in advance

[1] Except when having very small $\tilde{\delta}$.

or dynamically and to traverse it accordingly in order to identify *all non-dominated template arcs*, which form T^c. In this case, the traversal is not guided by the dual variables, since they represent an explicit state of the branch-and-price process. The focus shifts to the upper bound set UB and lower bound set LB. Both sets are now employed to identify dominated subtrees in order to prune unappealing subtrees. The procedures and algorithms will be explained in chapter 7.

6.2 Segmentation Trees

The segmentation tree can be constructed in two different ways:

- *Static tree*: Here we build the entire segmentation tree in advance. After the construction was concluded, we search for the solution to the pricing problem. The two operations *build* and *search* are executed sequentially. As all node differences b_i are already included in this segmentation tree, no further expansion needs to be performed in the search phase. This tree is presented in section 6.4.
- *Dynamic tree*: The segmentation tree is built while searching for solutions to the pricing problem. This method only expands branches of the search tree which might be needed due to the values of the dual variables. The resulting tree is incomplete and the number of nodes depends on the node differences b_i that are part of the solution to the actual pricing problem. This tree is described in section 6.5.

We first present the static approach, and describe the build and search procedures. Thereafter we present the dynamic approach which is basically an intertwined application of these two procedures. The algorithms and concepts are described in 2- and 3-dimensional space for better understanding, but work for all dimensions $d \geq 2$.

6.3 Region Abstraction

In the following we treat the discrete domain $\mathbb{D}$ as a continuous one as the algorithms have been designed and implemented for the general continuous case. In this manner, the algorithms may be used also for other applications, both continuous and discrete, and not only for this particular problem. Thus we have to operate with an enlarged domain ranging from 0 to $\tilde{v}$. This is necessary as $\tilde{v} = 0 \mod \tilde{v}$ and therefore regions crossing the domain border would otherwise not be connected within the continuous domain. This is, however, a property of this particular implementation and is no requirement for the segmentation tree to work correctly.

In order to make the construction of the segmentation tree recursive, flexible and easy to understand, we introduce a region abstraction. We defined the *bounding box* for b_i as the 2^d corner points $p^{(i)}$, d being the dimensionality:

$$(b_i^1, b_i^2, \ldots, b_i^d), (b_i^1 - \tilde{\delta}^1, b_i^2, \ldots, b_i^d), (b_i^1, b_i^2 - \tilde{\delta}^2, \ldots, b_i^d), \ldots, (b_i^1 - \tilde{\delta}^1, b_i^2 - \tilde{\delta}^2, \ldots, b_i^d - \tilde{\delta}^d).$$

A *region* R_i, defined by a b_i, is a tuple of points $\langle p_0^{(i)}, p_{2^d-1}^{(i)} \rangle$ that encodes two corner points from the bounding box. The points are $p_0^{(i)} = (b_i^1, b_i^2, \ldots, b_i^d) = b_i$ and $p_{2^d-1}^{(i)} = (b_i^1 - \tilde{\delta}^1, b_i^2 - \tilde{\delta}^2, \ldots, b_i^d - \tilde{\delta}^d)$. For the implementation only these two points are important, since all relevant data can be derived from them. Based on this definition, regions may not be part of our finite domain $\mathbb{D}$ since they are not yet transformed into the domain by a modulo transformation. The modulo transformation will be merged into the tree construction process.

If for a region R_i the condition $(p_0^{(i)})^d \geq (p_{2^d-1}^{(i)})^d$ holds for all dimensions d, the region is *regular*. If subtracting $\tilde{\delta}$ from a b_i leads to negative values, e.g. the resulting corner point $p_{2^d-1}^{(i)}$ is negative at one or more dimensions, we have a *(partially) negative region*, which lies outside domain $\mathbb{D}$. Such regions must be transformed into $\mathbb{D}$ and the respective negative corner point $p_{2^d-1}^{(i)}$

specially handled.
In the following we use the term $\langle p_0^{(i)}, p_{2^d-1}^{(i)} \rangle$ for regions R_i derived from a node difference b_i and the parameter $\tilde{\delta}$. The term $\langle p_0, p_{2^d-1} \rangle$ is used for regions in the general case. If $\langle p_0, p_{2^d-1} \rangle$ is regular and lies in the finite domain $\mathbb{D}$, we use the term *subspace*. The term was already introduced to denominate a segment encoded by a node n in the segmentation tree. Tree segments lie always in domain $\mathbb{D}$.

In the following we define abstract operations on general regions, some of them illustrated in figure 6.5. These operations have been consequently implemented as class operators, so that the code is not obfuscated by lots of flags and loops for iterating through dimensions.

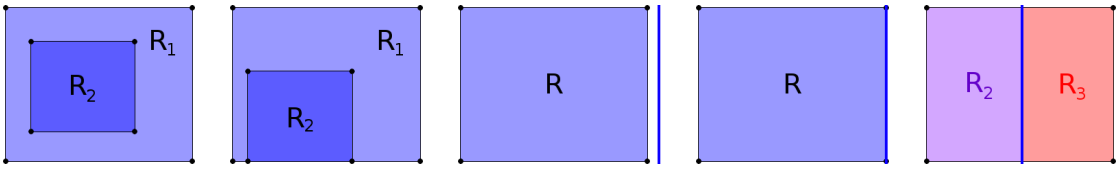
Equality: Two regions are completely equal $R_1 = R_2$, if the respective corner points $\langle p_0, p_{2^d-1} \rangle$ for both R_1 and R_2 are equal. The encoded subspace have the same size.

Inside Test: If $R_2 \subset R_1$, region R_2 is located *completely inside* R_1, and there is no overlap at the edges. We check if $p_0^{(R_2)} < p_0^{(R_1)}$ and $p_{2^d-1}^{(R_2)} < p_{2^d-1}^{(R_1)}$. If $R_2 \subseteq R_1$, then R_2 lies *inside* R_1, some edges may overlap. Here, $p_0^{(R_2)} \leq p_0^{(R_1)}$ and $p_{2^d-1}^{(R_2)} \leq p_{2^d-1}^{(R_1)}$. If region R_2 only partially overlaps R_1, then $R_2 \not\subset R_1$.

Greater, Less: Comparison of a region against a coordinate and dimension pair (c, dim), called splitting hyperplane: The check $R > (c, dim)$, $R \geq (c, dim)$ determines if an entire region, e.g. its two corner points are $>$, $\geq$ than the splitting hyperplane (c, dim) for the respective dimension d, or $<$, $\leq$ than (c, dim). In case of $\leq, \geq$, the points may lie upon the splitting plane, in the other case not.

Split: Divide a region R at a specified coordinate and dimension (c, dim). The result are two regions R_1 and R_2. The region $R_1 = \langle p_0', p_{2^d-1} \rangle$ has a new corner point p_0' with the value c at dim, and $R_2 = \langle p_0, p_{2^d-1}' \rangle$, where p_{2^d-1}' has c at dim.

Size: Size of a region, product of the facet length in each dimension.



(a) $R_2 \subset R_1$ (b) $R_2 \subseteq R_1$ (c) $R < (c, d^1)$ (d) $R \leq (c, d^1)$ (e) $split(R, (c, d^1))$

Figure 6.5: The operations $\subset$, $\subseteq$, $<$, $\leq$ and split.

6.4 The Static Segmentation Tree

We proceed with building the segmentation tree. As we are now considering the static variant, the entire tree is constructed at once by inserting all b_i in a random order. In the end, the tree contains a segmentation of *all* node differences b_i. When the construction is complete, in the following step we traverse it in search of our desired result.

Each node n in our segmentation tree contains the following information:

- The splitting coordinate n_c and the splitting discriminator n_{disc}. The discriminator is incremented in each tree level. The actual dimension value is extracted by $dim = disc \mod d$. Together c and dim form the splitting hyperplane (c, dim) of the actual node.
- The subspace n_R enclosing the splitting hyperplane, with $R = \langle p_0, p_{2^d-1} \rangle$.
- The upper bound set n_{UB} and the lower bound set n_{LB}, as introduced in section 6.1.2.
- Each node n has the successors n_{left}, n_{right}, as well as the predecessor n_{parent}. The terms n_{left_R}, $n_{left_{UB}}$, $n_{left_{LB}}$ and so on, denominate successor data.

6.4.1 Construction Main Algorithm

The main construction algorithm is outlined in pseudo code 6.1. Input to the algorithm are the set of fingerprint minutiae data points $v_i \in V$ and the parameter $\tilde{\delta}$. For the input parameter $\tilde{\delta} \leq \frac{\tilde{v}}{2}$ must hold, checked at line 1. At line 2, the algorithm calculates out of the input data points the set of node differences b_i by $b_i = (v_j - v_i) \mod \tilde{v} \mid v_i, v_j \in V,\ i \neq j$.

The node differences b_i define our working domain $\mathbb{D} = \{0, \ldots, \tilde{v}^1 - 1\} \times \cdots \times \{0, \ldots, \tilde{v}^d - 1\}$ with the resolution $\tilde{v}^1, \ldots, \tilde{v}^d \in \mathbb{N}$. We encode this domain into the tree root and generate a root node n_{root}, containing the regular subspace R_{root} defined by $p_0^{(root)} = (0, \ldots, 0)$ and $p_{2^d-1}^{(root)} = (\tilde{v}^1, \ldots, \tilde{v}^d)$. Further, for this root node, we set *disc* to the value for the first dimension to be segmented d_{start}, which normally is d^1. Yet, at this point we neither set the actual splitting coordinate, left and right successor nor the bound sets UB and LB. The root node has no parent. These initialization steps are encoded in listing 6.1 at lines 2-6. The dimension where we start segmenting, assigned in line 6, may be customized by adding an additional input parameter.

The following steps are performed for each $b_i \in B$: We create out of each b_i a region R_i with size $\tilde{\delta}$ (line 9) which may be negative, since it is not yet modulo calculated. We insert them into the tree in random order by calling algorithm `StaticInsert`(R_i, n_{root}) at line 10. The parameter n is a pointer to the node, where we insert R_i, in this case the root node. `StaticInsert` segments the domain sequentially for each b_i in a recursive way and is described in the following subsection 6.4.2. When the segmentation of the finite domain was finished, the algorithm searches for τ^* with algorithm `StaticTreeSearch-`τ^* (line 11). This algorithm is subject of section 6.4.3.

Algorithm 6.1: StaticSegmentation(V, $\tilde{\delta}$).

Data: A set of normalized data points V, delta domain $\tilde{\delta}$.

```
1  if  δ̃ ≤ ṽ/2  then
2      B = {(v_j − v_i) mod ṽ | v_i, v_j ∈ V, i ≠ j};
3      /* Initialize static segmentation tree t.  */
4      R_root ← ⟨(0, ..., 0), (ṽ^1, ..., ṽ^d)⟩;
5      n_root ← R_root ;
6      d_start ← d^1;
7      /* Build the static segmentation tree.  */
8      forall b_i ∈ B do
9          R_i ← (b_i, δ̃);
10         StaticInsert (R_i, n_root);
11 Extract τ* with the algorithm StaticTreeSearch-τ* in section 6.4.3;
```

6.4.2 Static Insert Algorithm

Input to the algorithm `StaticInsert` is a region R_i derived from a b_i and the pointer to the actual node position n. In the first iteration call this node position pointer is the tree root n_{root}. At this first call, the tree contains only the root node, which encodes the region encompassed at this point, namely the entire domain defined by the input data points.

First, we compare the new region R_i to be inserted to this root node subspace R_{root}. As we merge the modulo transformation into the tree construction process, we designed a region R_i to be constructed from a node difference b_i by subtracting $\tilde{\delta}$. Thus, it may happen that the corresponding $\tilde{\delta}$-sized bounding box is negative and reaches outside R_{root} and thus lies outside our domain. When this happens $b_i^d - \tilde{\delta}^d < 0$ for one or more dimensions. For identifying such cases we perform an *inside/outside test* and thus determine if the new region R_i lies inside or (partially)

outside our root node subspace R_{root}.
When inserting such regions into the segmentation tree, parts lying outside R_{root} have to be identified and modulo transformed into our domain, so that in the and a correct segmentation of the finite domain with ring structure is achieved. A region R_i lies outside the domain, if $R_i \subset R_{root}$ is false or $R_{root} \cap R_i = \emptyset$. If R_i bases on a node difference b_i having one or more coordinates equal zero, the respective bounding box lies outside R_{root} with exception of the $(d-1)$-hyperplane lying at 0. This hyperplane is irrelevant for our purposes. Having such a region, we transform this region entirely into R_{root} by a modulo operation of its defining corner points. Thus all coordinates equal to 0 become $\tilde{v}^d$. We re-insert the transformed region recursively into our tree.
If the region is only *partially negative*, as example R_1 in figure 6.6, and has one or more coordinates < 0 and $R_{root} \cap R_i \neq \emptyset$. Our region is partially located inside and outside our root node subspace R_{root}. We split R_i it into two subregions with the splitting hyperplane $(0, dim_{neg})$ where dim_{neg} is the first negative dimension found. So we get two regions, one lying inside our domain, and one lying outside. The region R_{right} is regular and we insert it normally by calling recursively `StaticInsert`(R_{right}, n_{root}). Region R_{left} has to be transformed into our domain R_{root} by a modulo transformation w.r.t. dim_{neg} and then recursively re-inserted. Further negative coordinates (as would be the case for R_2 in figure 6.6) are handled in the same manner after the re-insertion in the next iteration, as long there exist negative dimension coordinates. Parts representing irrelevant subspaces are omitted. By subsequently splitting away all negative parts of regions and transforming them into the domain at one dimension coordinate by one, we make them all regular. The example region R_2 in figure 6.6 would be first split at coordinate 0, dimension 0 and we get the regions $R_{2A} \cup R_{2D}$ and $R_{2B} \cup R_{2C}$, which in the next recursion are split to R_{2A} and R_{2D}, R_{2B} and R_{2C} respectively.

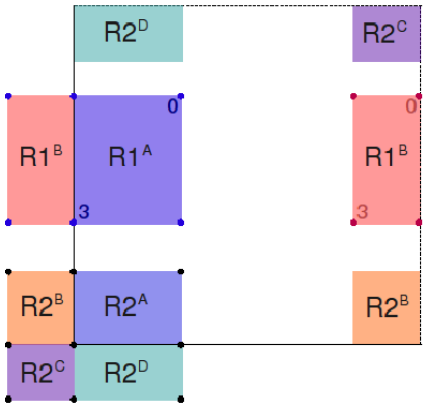


Figure 6.6: Partially negative regions and their transformation to the working domain. Regions R_{1A} and R_{2A} are regular, the other are modulo transformed into domain $\mathbb{D}$.

We proceed with the static insertion process and proceed with the case of having $R_i \subset R_{root}$. If R_i lies *inside* the domain borders and is regular, we determine: If the actual node position n is *no leaf* and has successors, we recursively descend into the tree by determining the appropriate successor. If our region to be inserted is $R_i \leq (c, dim)$, i.e. less[2] than the splitting hyperplane (c, dim) of the actual node position n, the new region lies for certain left of (c, dim) and thus in the left subtree. We save the actual index i from R_i into the actual node upper bound set UB, since the area encompassed at this node contains R_i for sure[3]. The set UB encodes all node difference ids, that lie in the subspace n_R defined by the actual node. Then we make a recursive call to `StaticInsert`(R_i, n_{left}), where n_{left} is the left son of the actual node position n. In this manner we descend into the left subtree and update the bound sets at the tree nodes on the way down. If $R_i \geq (c, dim)$, we save again the index i at the actual node and descend into the right subtree with a recursive call to `StaticInsert`(R_i, n_{right}).

[2] "left" or "below" in 2-dimensional case.
[3] The inside test before was positive.

If the actual node pointer n is a *leaf* and has no successors, we proceed with segmenting the subspace, defined by node n, by means of the corner points of the new region to be inserted.

Simple Segmentation

As we intend to segment the actual subspace along the facets of R_i, we perform a series of *splits* ς with alternating dimensions. The actual node n encodes the discriminator for the split, and we get the splitting dimension by $dim = n_{disc} \mod d$. We begin with point $p_0^{(i)}$ defined by the new region R_i and make d splits, one for each dimension, followed by d splits for point $p_{2^d-1}^{(i)}$. In the special case, when having overlapping points for a region with the actual subspace defined by a node, we can leave out the splits at this point.
By continuing the example, we split the actual domain at $c = (p_0^{(i)})^{dim}$ and save at node n the resulting splitting hyperplane $(c, dim) = ((p_0^{(i)})^{dim}, dim)$. Two successors n_{left} and n_{right} are generated and appended to the tree at the actual node n. As at the actual node encodes the subspace $R = \langle p_0, p_{2^d-1}\rangle$, the left son now defines the subspace $R_{left} = \langle p'_0, p_{2^d-1}\rangle$, where point p'_0 has the new value c at dim. The right son has $R_{right} = \langle p_0, p'_{2^d-1}\rangle$, where p'_{2^d-1} has c at dim. Figure 6.7 illustrates the working principle of a simple segmentation, e.g. a simple split sequence. The resulting segmentation tree is depicted in figure 6.8.

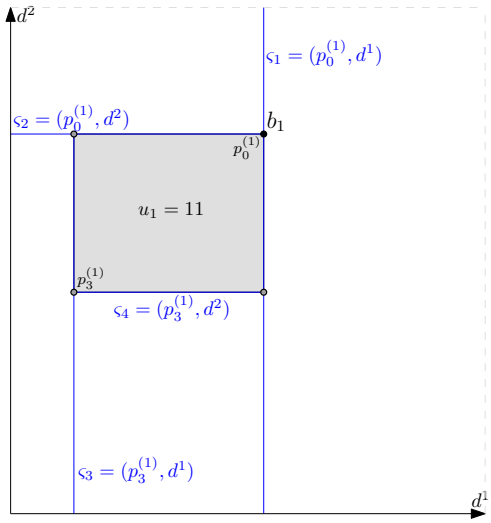


Figure 6.7: Simple segmentation of node difference b_1. The figure shows four splits ς, two at $p_0^{(1)}$ and two at $p_{2^d-1}^{(1)}$. The resulting segmentation tree is depicted in figure 6.8.

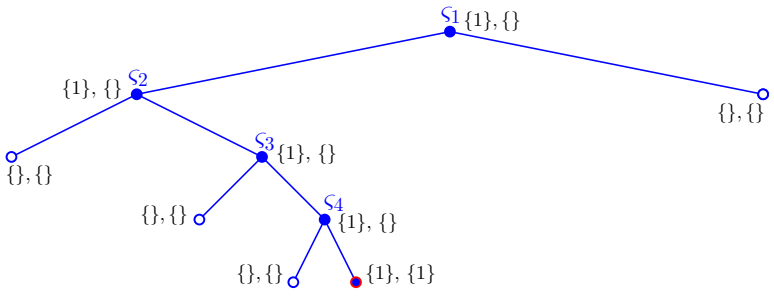


Figure 6.8: Segmentation tree after insertion of b_1.

The actual node upper bound set is again updated by saving index i, thus denoting that the subspace encompassed at the actual node includes R_i. Each time we append new nodes we also have to carefully inherit upper and lower bound sets, in order not to loose some previously inserted R_i. As the elements for LB are for sure in this subtree, they are simply propagated. Only the elements in UB must be checked against being part of the new subnodes.

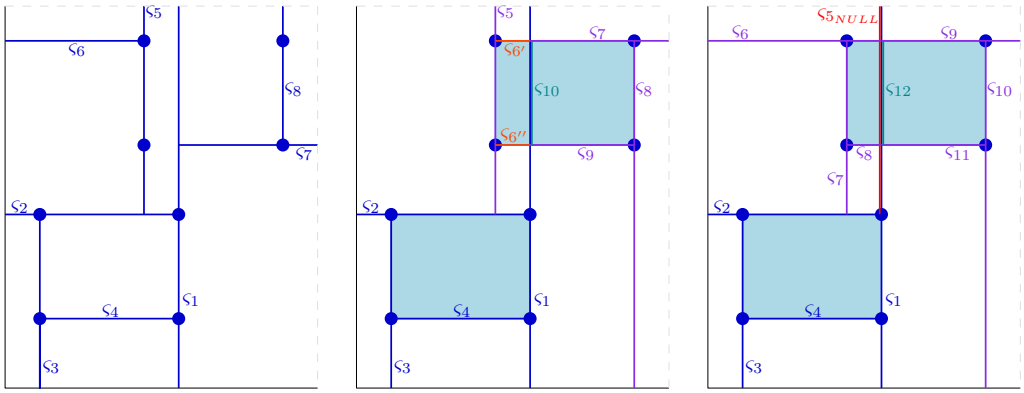
By performing such a split, the subspace encoded by node n is split into two subspaces, one containing our region to be inserted R_i and one not containing it. This is easily checked by $R_i \leq (c, dim)$ and $R_i \geq (c, dim)$. Having completed the split, the `StaticInsert` algorithm descends into the tree by determining the successor containing our R_i by deciding whether $R_i \leq (c, dim)$ or $R_i \geq (c, dim)$.
The splits for the remaining dimensions are performed in the same manner. By executing the last split, either R_{left} or R_{right} are equal to the actual region R_i. At this point, we save the index i additionally to lower bound set. Furthermore, we carefully propagate the sets UB and LB from the predecessor node to the successor nodes. The lower bound set LB then holds *all* node difference indices that are expressed by the subspace encompassed at the actual point.

The algorithm performs $2 \cdot d$ splits, d splits at corner point $p_0^{(i)}$ and d splits at $p_{2^d-1}^{(i)}$. This procedure can be tuned by leaving out overlapping or redundant corner points, but we must pay attention to not destroy the dimension sequence in the tree levels. We reduced these $2 \cdot d$ splits by leaving out equals corner points:

- If a new region R_i equals the subspace at the actual node n_R, we perform no splits at all, since both corner points are equal for R_i and n_R. We simply add the index i of b_i to UB and LB at n. We omit all $2 \cdot d$ splits.
- If one of the corner points for R_i is coincident to the respective corner point of n_R, omit these point, thus leaving out d splits.

We can safely omit these splits without disturbing the dimension sequence in the tree levels. Further optimizations require a more sophisticated handling.

On the other side we have to absolutely maintain the dimension sequence in the tree levels. The difference between a conventional k-d-tree and the segmentation tree is depicted in figure 6.9.



(a) Conventional k-d-tree (b) How segment $\varsigma_6', \varsigma_6''$? (c) Correct segmentation.

Figure 6.9: The reason, why null subspaces have to be inserted. How, in figure 6.9b, should $\varsigma_{6'}, \varsigma_{6''}$ be segmented and the dimension sequence retained? Figure 6.9c shows the correct segmentation by inserting a null subspace $\varsigma_{5_{NULL}}$.

Maintaining the dimension sequence means, that the sequence $d^0, d^1, \ldots, d^d$ always has to remain intact when descending into the tree levels. Since the segmentation tree is not a conventional k-d tree, situations arise, where a special split has to be performed, that divides n_R into a subregion containing n_R itself and an "empty" subspace R_{NULL} with size 0. We call such empty subspaces *null subspace*. Such a null subspace lies exactly onto the actual splitting hyperplane and is trivially performed by the split operation automatically. Such a split is depicted in figure 6.10b.

Normal Segmentation

Until now we do not take into account overlapping regions, which is the trickier part. When inserting more and more regions, the chance increases that somewhere an overlap of regions takes place. In such a case (and there are probably many such cases) at some node n none of the two conditions $R_i \leq (c, dim)$ and $R_i \geq (c, dim)$ is true, since the new region has parts on both sides of the splitting hyperplane. Having such a case, we split up R_i at the actual splitting hyperplane into $R_{i_{left}}$ and $R_{i_{right}}$ and insert the "left" (if $R_i \leq (c, d)$) part into the left subtree by invoking recursively `StaticInsert(`$R_{i_{left}}$`,` n_{left}`)` . The "right" part is inserted by `StaticInsert(`$R_{i_{right}}, n_{right}$`)` into the right subtree. At the actual node we save again the index of b_i to UB.

In each subtree, $R_{i_{left}}$ and $R_{i_{right}}$ are handled recursively in the same manner. Thus, an accurate segmentation of the domain is constructed. The figures 6.10a and 6.10b illustrate these steps, by continuing the example in figure 6.7. The resulting segmentation tree is depicted in figure 6.11. Both figures show also the handling of a null split $\varsigma_{7_{NULL}}$. The corresponding segmentation tree has a null subspace at the right successor of the node corresponding to split $\varsigma_{7_{NULL}}$.

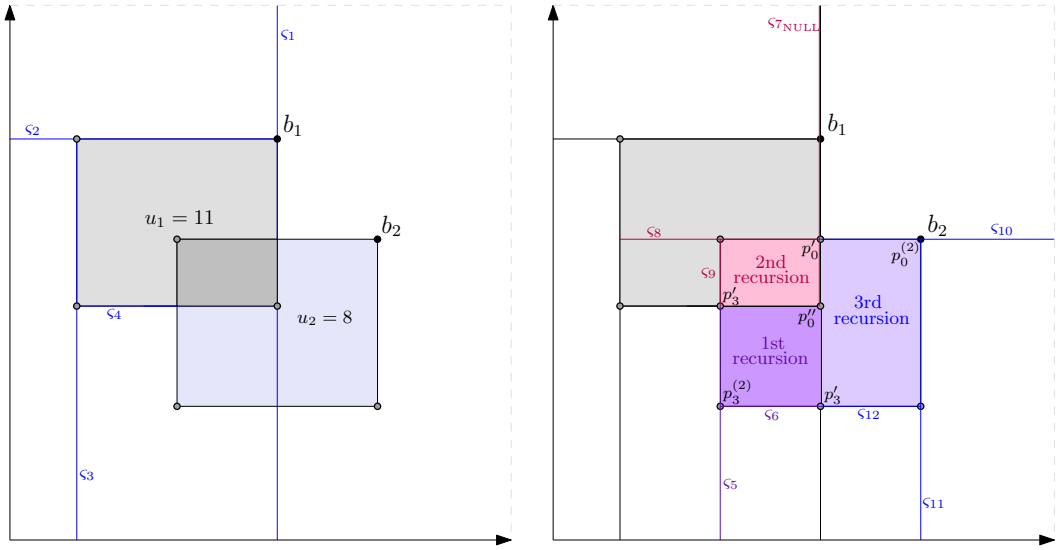


Figure 6.10: Figure 6.10a) shows the insertion of b_2, overlaps at splits ς_1 and ς_4 take place. Figure 6.10b) shows the segmentation of b_2: Since R_2 is located on both sides of ς_1, the algorithm splits R_2 into two subregions $R_2' = \langle p_0', p_3^{(2)} \rangle$ and $R_2'' = \langle p_0^{(2)}, p_3' \rangle$, and inserts them recursively into the left and right subtree respectively. First, the algorithm descends left with R_2' and encounters an overlap again at ς_4. Again R_2' is split up into $\langle p_0'', p_3^{(2)} \rangle$ and $\langle p_0', p_3' \rangle$, which are re-inserted recursively. These two subregions as well as R_2'' are further simply segmented. Figure 6.11) shows the resulting segmentation tree with bounds after insertion of b_2.

6.4.2.1 Algorithms

We describe now in detail the algorithm outlined in the preceding paragraphs. For simplicity reasons, the described method was split up into two procedures. Pseudo code 6.2 lists the algorithm `StaticInsert(`R_i, n`)`. This algorithm basically manages the tree descending process and the segmentation at the region corner points.

Input parameters for `StaticInsert` are the region R_i, generated from a b_i, and the actual node position n. In the first iteration $n = n_{root}$, and the algorithm decides first of all, if R_i lies inside or (partially) outside the subspace defined by n_{root}, e.g. our working domain. This is done

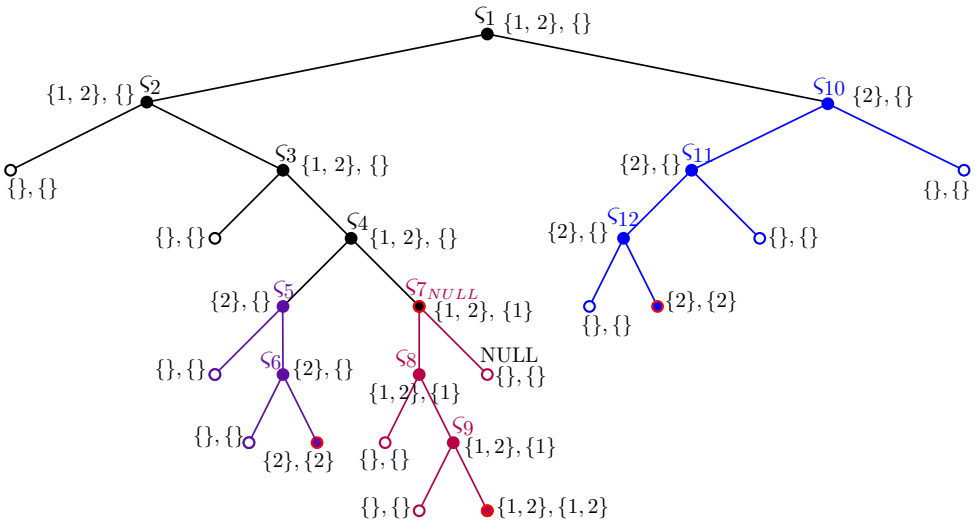


Figure 6.11: Segmentation tree with bounds after insertion of b_2.

at line 2 by performing the appropriate *inside* check. If the region to be inserted is partially negative ($p^{(i)}_{2^d-1}$ is negative for some dimensions), this region is split up the first negative dimension found (lines 40-43) and coordinate 0 and R_{left} transformed into our domain (lines 48, 49, 50). Then, R_{left} and R_{right} are re-inserted recursively (lines 46, 51). Parts representing null subspaces (determined at line 45) are omitted.

If the region to be inserted is regular and inside R_{root}, the algorithm proceeds normally. In the next step (line 4), the algorithm determines, it the actual node n is a leaf or an intermediate node. When n is a leaf the algorithm proceeds with segmenting the subspace encompassed by n_R (line block 6-24). If n is no leaf, the algorithm expands the UB set at n (line 25), and decides if one or both successors contain R_i. If R_i lies left of the actual splitting hyperplane at n, the algorithm descends recursively into the left subtree (lines 27-28), else if R_i lies right of the splitting hyperplane it descends into the right subtree (lines 29-30). If R_i lies on both sides of the splitting hyperplane, the algorithm performs a split and divides R_i at the splitting hyperplane (line 33). After this, it inserts the resulting subregions recursively into their respective subtree at the actual node position n (line 34 and 35). Next, if n is a leaf, the algorithm determines, if segmentation occurs. If both corner points for R_i and n_R are equal (line 6), the algorithm updates only the bound sets at n. It simply adds the index i of b_i to the actual nodes upper and lower bound set (line 7) and performs no segmentation. In the other case the algorithm determines the splits that must be made by constructing the set of split coordinates P: If the corner points p_0 are equal for n_R and R_i, only the splits at coordinates $P = (p^1_{2^d-1}, p^2_{2^d-1}, \ldots, p^d_{2^d-1})$ of R_i are performed (lines 13-14). If p_{2^d-1} are equal for n_R and R_i, the splits are performed at $P = (p^1_0, p^2_0, \ldots, p^d_0)$ for R_i (lines 15-16). If no corner points are equal, all $2 \cdot d$ splits are performed and set P holds all coordinates (line 18). With the loop at line 19, all splits are made sequentially by calling the algorithm `StaticAppendNodes` at line 20, which handles all the segmentation steps and adds each time it is called two new successors encoding the actual split. Since the segmentation is performed based on a R_i, which divides n_R into one subspace containing R_i and into one not containing it, the appropriate successor for descending into the tree for performing the next split is selected by identifying this successor (lines 22-23). Thus, the algorithm determines the correct successor by checking if $R_i \leq (n_c, n_d)$ (line 22), or $R_i \geq (n_c, n_d)$ (line 23) respectively and descends into the appropriate subtree by setting the node pointer n to the newly determined successor.

Algorithm 6.2: StaticInsert(R_i, n)

Data: The region to be inserted R_i, a pointer n to the actual tree node position.

```
1  /* Check if the region R_i lies inside the actual subspace n_R of node n.  */
2  if R_i ⊆ n_R then
3      /* If n is a leaf, segment and append new nodes.  */
4      if n_left = ∅ AND n_right = ∅ then
5          /* If subspace n_R equals our inserted region R_i, only add bounds.  */
6          if R_i = n_R then
7              n_UB ← n_UB ∪ i; n_LB ← n_LB ∪ i;
8          else
9              /* If R_i ≠ n_R, segment.  Before descending, add i to n_UB.  */
10             n_UB ← n_UB ∪ i;
11             Let P = (p_0^1, p_0^2, ..., p_0^d, p_{2^d-1}^1, p_{2^d-1}^2, ..., p_{2^d-1}^d) denote the set of coordinates
               belonging to the points ⟨p_0^(i), p_{2^d-1}^(i)⟩ of region R_i;
12             /* If for R_i and n_R a p_0 or p_{2^d-1} overlaps, leave it out.  */
13             if p_0^(n_R) = p_0^(R_i) then
14                 P ← (p_{2^d-1}^1, p_{2^d-1}^2, ..., p_{2^d-1}^d);
15             else if p_{2^d-1}^(n_R) = p_{2^d-1}^(R_i) then
16                 P ← (p_0^1, p_0^2, ..., p_0^d);
17             else
18                 P ← (p_0^1, p_0^2, ..., p_0^d, p_{2^d-1}^1, p_{2^d-1}^2, ..., p_{2^d-1}^d);
19             for p ∈ P do
20                 StaticAppendNodes (R_i, n, p); /* Algorithm 6.3 */
21                 /* After appending, descend for next append.  */
22                 if R_i ≤ (n_c, n_d) then n ← n_left;
23                 else if R_i ≥ (n_c, n_d) then n ← n_right;
24     else
25         n_UB ← n_UB ∪ i; /* If n no leaf, save i to n_UB */
26         /* If R_i lies left or right of (c, dim), descend recursively.  */
27         if R_i ≤ (n_c, n_d) AND (n_left ≠ ∅) then
28             StaticInsert (R_i, n_left);
29         else if R_i ≥ (n_c, n_d) AND (n_right ≠ ∅) then
30             StaticInsert (R_i, n_right);
31         else
32             /* If R_i on both sides of (c, dim), split, insert parts into respective subtrees */
33             R_{i_left}, R_{i_right} ← split(R_i, n_c, n_d);
34             StaticInsert (R_{i_left}, n);
35             StaticInsert (R_{i_right}, n);
36 else
37     /* This occurs if R_i ⊈ R_root:  R_i lies partially outside R_root */
38     for all dimensions dim do
39         /* Test if (p_{2^d-1}^(R_i))^dim from R_i reaches outside domain.  */
40         c ← (p_{2^d-1}^(R_i))^dim;
41         if c ≤ 0 then
42             /* If this point is negative, split the region at coordinate 0.  */
43             R_{i_left}, R_{i_right} ← split(R_i, 0, dim);
44             for r ∈ {R_{i_left}, R_{i_right}} do
45                 if size(r) > 0 then
46                     if r ≥ (0, dim) then StaticInsert (r, n);
47                     else
48                         /* Transform into 𝔻, then insert.  (p_0^(r))^d < 0 does not occur.  */
49                         if (p_0^(r))^dim = 0 then (p_0^(r))^dim ← ṽ^dim;
50                         if (p_{2^d-1}^(r))^dim ≤ 0 then (p_{2^d-1}^(r))^dim ← ṽ^dim + (p_{2^d-1}^(r))^dim;
51                         StaticInsert (r, n);
```

Listing 6.3 outlines the algorithm `StaticAppendNodes(`R_i`, `n`, `p`)`. This algorithm manages division of a node subspace at a splitting hyperplane, appends new nodes and organizes the bound sets in each node. This procedure is called at most $2 \cdot d$ times in `StaticInsert`, after the algorithm has determined the region corner point coordinates p where to perform segmentation. The algorithm `StaticAppendNodes` receives the actual p as parameter. Having determined coordinate and dimension for the splitting hyperplane at lines 2-3, the split is performed by dividing n_R at (c, dim) (line 4). The algorithm updates the splitting hyperplane data at node n (line 6), and generates both new successor nodes n_{left} (line 7-8) and n_{right} (line 9-10). The upper and lower bound sets from n are inherited accordingly to n_{left} (line 8) and n_{right} (line 10). By performing the last split from P for R_i, the index i for the actual b_i is additionally saved to the respective successors LB set, which contains finally R_i or some final part of R_i (line 11-15).

Algorithm 6.3: StaticAppendNodes(R_i, n, p)

Data: Region R_i, actual node position n, coordinate for next splitting hyperplane p.

```
1  /* The new split coordinate c lies at coordinate p and dimension dim, which
      is derived from the actual n.  The coordinate p was derived either from
      point p_0 or p_{2^d-1} of R_i.  */
2  dim ← n_disc mod d;
3  c ← (p, dim);
4  R_left, R_right ← split(n_R, c, dim);
5  /* Actualize the split at the actual node position n.  */
6  n_c ← c;  n_d ← dim;
7  n_{left_R} ← R_left;  /* Generate the new left node and inherit bounds.  */
8  n_{left_UB} ← n_UB;  n_{left_LB} ← n_LB;
9  n_{right_R} ← R_right;  /* Generate the new right node and inherit bounds.  */
10 n_{right_UB} ← n_UB;  n_{right_LB} ← n_LB;
11 if last split for R_i then
12     if R_i ≤ (n_c, n_d) then
13         n_{left_LB} ← n_{left_LB} ∪ i;
14     else if R_i ≥ (n_c, n_d) then
15         n_{right_LB} ← n_{right_LB} ∪ i;
```

6.4.3 Traversing the Static Segmentation Tree (Searching τ^*)

Having now built the static segmentation tree, we traverse it in search for τ^*, the solution to our pricing problem. The tree already encodes the node differences b_i as upper and lower bounds UB and LB. For each lower bound set a standard template arc can be derived. The solution to the pricing problem is τ^*, the standard template arc having maximal negative reduced cost. The reduced cost is negative, since we calculate $1 - \sum_{(i,j)\in A(t)} u_{ij}$. So, we have to find in our segmentation tree the leaf with the bound set having the maximal dual value sum and a negative reduced cost, derive the standard template arc τ^* from it, integrate this τ^* as a new variable into our actual LP and continue the branch-and-price process. This traversing process is executed multiple times, while pricing in each branch-and-bound node as long as there exist $\bar{c}_t$ having reduced cost less 0. Actually we have to search only branches that have sums of dual values greater than 1, since all other branches will produce a reduced cost greater equal 0.
So, at the beginning of each search, we determine the actual dual value u_i for each b_i from the current branch-and-price iteration, since these values vary from iteration to iteration. Now at each node we can compute a numerical value, upon which we base the search process. As traversing strategy we use standard *best first search*, in order to quickly find a good result in combination with bounding techniques for omitting irrelevant subtrees.

The algorithm StaticTreeSearch-τ^* proceeds in the following way: We begin at the root node and determine $lb(n) = \sum_{i \in LB(n)} u_i$ for both successor nodes. Then we descend first recursively into the branch having the greater $lb(n)$ and leaving the remaining branch to later recursions. At this new node, we determine again $lb(n)$ for each successor and proceed likewise. If we encounter equal $lb(n)$ for both successors, it is a matter of implementation, which node we visit first. So, at some point we reach a leaf and save lb^* as actual best found bound, if its $lb(n)$ is greater than 1. Now, all the remaining recursions, that were stalled until now, are resolved, but by taking into account our actual solution. At the parent of the node containing lb^* the remaining successor is visited, but only if it has an upper bound value $ub(n) \geq lb^*$ and $ub(n) > 1$. If such a branch is found, it is again visited in the same best first search manner. If $ub(n) < lb^*$ and $ub(n) \geq 1$, the subtree is pruned. By unwinding all recursions, we again ascend in the tree, but by visiting only branches that may improve our actual result. If an improvement for lb^* is found in the procedure, we save this new solution as lb'^* and continue the BFS process based on this new value. When we want to find only non-dominated bound sets when having equal sums of dual values, we have to insert a check for domination, if the actual LB at some leaf dominates LB^*, and in this case replace LB^*.

The algorithm StaticTreeSearch-τ^* is outlined in listing 6.4. The algorithm decides, if the actual node position n is a leaf or an intermediate node (line 3). If n is a leaf, the algorithm determines whether to save the actual n_{LB} (lines 6-11) to the global variable for the best found bound LB^* and lb^*, or not to save it. If n is no leaf, the algorithm determines which subtree to visit first at line 14. Each subtree is checked if its upper bounds sum of dual values is greater equal our actual lb^* and greater than 1. If so, the subtree is visited (lines 16-17 and 19-20), else it is omitted.

Algorithm 6.4: StaticTreeSearch-$\tau^*(n)$

Data: A node position n.
Result: Sets LB^* and lb^* to the highest value found during the traversing process.

```
1  if n ≠ ∅ then
2     /* If there are currently no children, n is a leaf.  */
3     if n_left = ∅ AND n_right = ∅ then
4        /* The actual node is a leaf node.  */
5        if n_UB = n_LB then
6           if lb(n_LB) > lb* then
7              /* Save the best found bounds and its random value sum.  */
8              lb* ← lb(n_LB); LB* ← n_LB;
9           else if lb(n_LB) = lb* then
10             /* Check if the new bound set dominates the best bound set.  */
11             if LB* ⊂ n_LB then LB* ← n_LB;
12    else
13       /* If n is no leaf, first follow the branch with higher bound.  */
14       if ub(n_leftUB) ≥ ub(n_rightUB) then
15          /* If upper bound of the left subtree < the actual best lower
               bound, do not follow this branch, else follow it.  */
16          if ub(n_leftUB) ≥ lb* AND ub(n_leftUB) > 1 then StaticTreeSearch-τ* (n_left);
17          if ub(n_rightUB) ≥ lb* AND ub(n_rightUB) > 1 then StaticTreeSearch-τ* (n_right);
18       else
19          if ub(n_rightUB) ≥ lb* AND ub(n_rightUB) > 1 then StaticTreeSearch-τ* (n_right);
20          if ub(n_leftUB) ≥ lb* AND ub(n_leftUB) > 1 then StaticTreeSearch-τ* (n_left);
```

6.4.3.1 Determining the Standard Template Arc

Once we have determined LB^*, we have to compute τ^*. The standard template arc τ is determined out of the bound set LB^* as described in section 4.2.1. For bound sets with size 1, the b_i is the standard template arc. For $|LB^*| > 1$, we save the first element from this set as reference and determine separately for each dimension dim, if the remaining b_i have an according dimension coordinate less than the first element. Here we have to take into account the ring structure. This means that if we have a bound set that encodes b_i crossing the domain border, which reach again into the domain $\mathbb{D}$ at the opposite side of the domain border, we save the greater coordinate. From these domain crossing coordinates, in turn we want the smallest value. The correct procedure for this is important. The segmentation tree does not encode information about template arcs encoding areas crossing domain borders. Although a τ may have two "parts" in the tree by crossing the domain border, both having the same bound set LB. The tree segments these two parts in two separate subtrees, one positioned somewhere wide left in the tree, the other wide right. It is irrelevant, which one of these two parts is finally found by `StaticTreeSearch-`τ^*, since the correct standard template arc will be derived out of it.

6.4.4 Upper Bounds for the Static Segmentation Tree

Since the segmentation tree bases on a k-d-tree, building has complexity $\mathcal{O}(k \cdot n \log n)$. Searching one element has complexity $\mathcal{O}(k \cdot n^{1-\frac{1}{k}} + R)$. The worst case for the number of nodes in the segmentation tree would be a segmentation of $\mathbb{D}$ into $\tilde{v}$ subspaces of size 1. Such a case would require $\mathcal{O}\left(\tilde{v}^2\right)$ nodes and thus at most a storage requirement quadratic in $\tilde{v}$. In our scenario such a case should never occur and thus the effective number of nodes is significantly lower.

6.4.5 Simulating `StaticTreeSearch-`τ^*

We verify the algorithm `StaticTreeSearch-`τ^* by simulating 1000 test iterations, based on random values. For this test we additionally need the set of non-dominated template arcs T^c. First we construct the entire static tree. Then, we start the simulation iterations, and in each iteration we assign new random values in $[1, 100]$ to each b_i. We determine from T^c the template arc $\tau^*_{T^c}$ having maximal sum of random values. Then we traverse the static tree and get the template arc τ^*_{seg}. Then we determine, if $\tau^*_{T^c} = \tau^*_{seg}$ and also their random value sums are equal. By an extensive series of computation tests we showed the method to work correctly.

6.5 The Dynamic Segmentation Tree

Basically, the dynamic segmentation tree works similar to the static version, but in the dynamic version we build the segmentation tree gradually, and enlarge it only when coming to a leaf that is "incomplete" and has to be expanded further. Thus we combine the two algorithms `StaticInsert` and `StaticTreeSearch-`τ^* into one procedure.

We begin again by creating a root node, but this time we initialize UB with all node differences b_i. The next step is to start the recursive search process. Since there are no successor nodes for the root node, the algorithm proceeds with the first segmentation. For this we have to select an element from UB, that has not yet been segmented. So we regard only elements from $UB \setminus LB$, which are the elements not segmented until yet. In the first insert iteration, this non symmetric difference set is naturally UB, since LB is empty. As soon we we segment some b_i, the sets LB are filled gradually. Since the sole elements b_i in this set allow no selection based on a numerical value, at this point we already need the current dual values u_{ij} or $u_{ij} + \mu_j$ respectively, which were previously derived from the current subproblem. Then we select the node difference b_i having the highest dual value, since this is the node difference that most likely leads fast to the greatest sum of dual values for the lower bound set lb^*. We continue by segmenting the respective R_i for this b_i.

As our actual node in the dynamic segmentation tree encodes a subspace n_R in our discrete domain $\mathbb{D}$ (with exception the root node, which holds the entire domain), the generated R_i may lie partially outside n_R or even R_{root}. In the first case, we have to crop the region to the subsegment which is contained in our node subspace by performing $R_i' = n_R \cap R_i$. In the second case, we have to pay attention to regions reaching outside our domain borders. For this we already transform a negative region into the domain R_{root} and then crop it to the subspace n_R. By doing so, we do not need lines 37–51 from algorithm `StaticInsert`, listing 6.2 anymore. This proceeding performs fast for the static segmentation and is only executed at the beginning of some insertion by respecting n_{root}, whereas in the dynamic version of the tree, this would lead to additional overhead. Thus, we incorporate the handling for negative regions directly into the tree by performing such a domain respecting crop operation $n_R \cap R_i$ for negative regions. By doing so, the overhead of ever beginning an insertion at n_{root} again is circumvented. Figure 6.12 shows this concept.

So, in the simple case, we segment R_i entirely in the first insert iteration at the root node and in situations, were the R_i lies completely inside the subspace defined by the actual tree node n. In the other case we only segment R_i', a small part of R_i contained in n_R. This cropped R_i' has adjusted corner points and lies always in R_{root}.

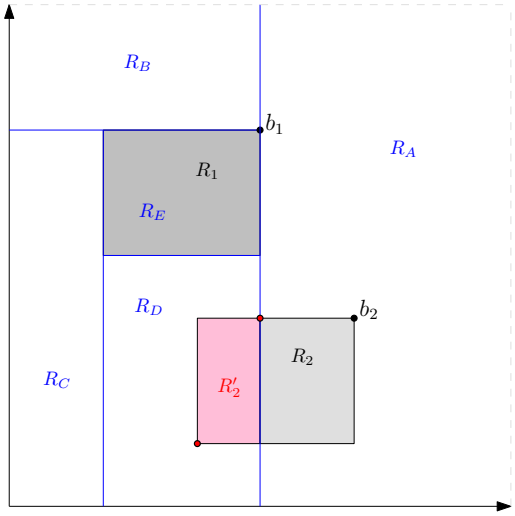


Figure 6.12: Rectangles show the regions that the node differences b_1 and b_2 define. If we arrive at node encoding subspace R_D and determine a part of R_2 to be inserted here, we crop the bounding box to the subspace by $R_2' = R_D \cap R_2$. The resulting R_2' is marked by two red dots. At node containing R_D, we insert only this part R_2'.

The segmentation works similar to the static counter part. For some part R_i' or an entire R_i, we first split at all d facets at point $p_0^{(i)}$, followed by d splits for point $p_{2^d-1}^{(i)}$. Again we leave out the d splits, if the two corner points overlap. The only difference is that at each split we must now regard the entire bounds set UB and LB at once for each successor node instead of simply adding the index of the b_i we currently insert. In each splitting iteration, where a node subspace n_R is divided into two subspaces R_{left} and R_{right}, we process the actual upper bound set n_{UB} and determine, if its elements lie again in the left subspace n_{left_R} and/or the right subspace n_{right_R}. For this, we create the regions $R_j, j \in UB$, already transform them into $\mathbb{D}$, and check if an overlap with the new successors tree node subspaces n_{left_R} and n_{right_R} occurs. If $n_{left_R} \cap R_j \neq \emptyset$, we save the index j to the respective left subnodes upper bound set, else not. The same we do with the right successor upper bound set. The lower bound set is simply propagated/inherited, since a subspace expressing at least LB is always split into two subspaces expressing again at least LB. Thus `StaticAppendNodes` is altered to `DynamicAppendNodes`, outlined in listing 6.7. The overlap check and crop operation are somewhat tricky because of the finite ring structure of the domain.

Having executed such a split sequence[4], the algorithm continues at the node, where we began

[4]We perform a full split sequence, so that complicate checks and additional data about already segmented corner

the splitting. Here new successors have taken their place, including a fully processed upper and lower bound set. So, we can descend into the tree, but in direction of the greatest sum of dual values, which must not necessarily be the subtree where b_i was just segmented. As example: n is a leaf and has $UB = \{b_1, b_2, b_3\}, LB = \emptyset$. Here, b_1, b_2, b_3 have to be segmented. Dual values are $u_1 = 9, u_2 = 6, u_3 = 5$. The algorithm segments b_1, since u_1 is greatest and produces for n the successors $n_{left_{UB}} = \{b_1\}$ and $n_{right_{UB}} = \{b_2, b_3\}$. In the left subtree at some point there is a $LB = \{b_1\}$. The right subtree is yet empty. As the dual value sum for the left subtree is less than in the right subtree, the algorithm chooses n_{right} for descending and finds a leaf, which is consequently segmented again.

As we employ the best first search strategy, at some point, we reach a leaf where $UB = LB$. Here no more b_i have to be segmented, since $UB \cap LB = \emptyset$. We save the actual LB and lb^*, and the algorithms proceeds with unwrapping the recursions. Hopefully, a lot of subtrees have now a sum of upper bound dual values less lb^* and less 1 and are omitted. If UB has a sum of dual values greater equal lb^*, this subtree must be searched. Maybe, there will be found some better lb^*, which we save and take as new global bound. If non-dominated bound sets are preferred, we add again a domination check. By proceeding thus, the procedure omits irrelevant subtrees by leaving them unconstructed. It expands new nodes only at the points determined to be relevant for the pricing problem. In the end, LB^* contains the optimum bound set out of which the standard template arc τ^* is determined. Its dual sum is lb^*. This is the solution to our pricing problem.

6.5.0.1 Algorithms

The pseudo code of `DynamicInsertAndSearch-`τ^* is outlined in listing 6.5. Algorithm 6.6 shows `DynamicInsertNode`. The altered `DynamicAppendNodes` is outlined in listing 6.7.

The algorithm `DynamicInsertAndSearch-`τ^*, listing 6.5, is divided again into two main blocks, one concerning the case for the actual node position n being a leaf (lines 2-28), the other performing the recursive BFS search (line 29-40). If n is a leaf and has empty UB and LB, this leaf is irrelevant (lines 4-5). If $UB > 0$ and LB empty, the algorithm determines at line 8 the element that must be segmented next, generates the according R_i (line 9), transforms and crops R_i to R_i' (line 11) and inserts it (line 12). The search is continued by a recursive call to `DynamicInsertAndSearch-`τ^* at line 13. If at n, the UB and LB are equal and not empty, the actual n_{lb} is evaluated against lb^* and 1 and saved, if greater (line 15-18). If upper and lower bound set are not equal the algorithm calculates the symmetric difference set Φ (line 23), and determines from this set Φ the region R_i to be segmented next (lines 24-26). Follow the calls to the algorithms for segmenting (line 27) and descending further (line 28).

The algorithm `DynamicInsertNode`, listing 6.6 takes as parameters the region R_i to be inserted and an actual node position n. First of all, the algorithm performs at line 1 the check if R_i lies in our actual node subspace n_R. Again, the algorithm differentiates between n being a leaf node and and n being an intermediate node. In case n is no leaf, the algorithm recursively descends into the tree and determines the appropriate position for segmenting R_i with lines 19-25. Having found this position, the upper bound set is expanded by i and the segmenting process is performed as in `StaticInsert` with lines 3-17. The algorithm `DynamicAppendNodes` at line 15 appends to n the new nodes n_{left} and n_{right} containing the newly segmented subspaces for R_i.

The algorithm `DynamicAppendNodes`, listing 6.7, is basically similar to the static version. First, the algorithm determines the next splitting hyperplane (lines 1-2), splits up n_R at this splitting hyperplane (line 3), and generates the new nodes n_{left} and n_{right} (lines 6-7). In the dynamic version, the actual bound set n_{UB} has to be specially handled: With lines 9-14, the algorithm

points/dimensions can be omitted. We exclude from segmentation only overlapping corner points and equal regions.

Algorithm 6.5: DynamicInsertAndSearch-$\tau^*(n)$

Data: A node position n. LB^*, lb^* are global and initialized with 0.
Result: Sets LB^* and lb^* to the highest value found during the traversing process.

```
1  if n ≠ ∅ then
2     if (n_left = ∅) AND (n_right = ∅) then
3        /* If n is a leaf, determine whether to save b_i or expand the tree. */
4        if (|n_UB| = 0) AND (|n_LB| = 0) then
5           /* Don't save this bound, since it is empty. */
6        else if (|n_UB| > 0) AND (|n_LB| = 0) then
7           /* At n, the LB is empty, expand this branch with max(UB). */
8           b_{i_max} ← max(UB); /* max(u_ij) or max(u_ij + μ_j) */
9           R_{i_max} ← (b_{i_max}, δ̃);
10          /* n_R includes R_{i_max} for sure. Crop/transform negative R_i into 𝔻. */
11          R'_{i_max} ← R_{i_max} ∩ R_n;
12          DynamicInsertNode (R'_{i_max}, n);
13          DynamicInsertAndSearch-τ* (n);
14       else if (|n_UB| > 0) AND (|n_LB| > 0) then
15          if n_UB = n_LB then
16             /* Do not expand n anymore. Save lb*, if > old value. */
17             if n_lb > lb* then
18                lb* ← n_lb; LB* ← n_LB;
19             else if lb(n_LB) = lb* then
20                if LB* ⊂ n_LB then LB* ← n_LB;
21          else
22             /* If UB ≠ LB determine the b_i to insert. */
23             Φ = UB \ LB;
24             b_{i_max} = max(Φ); /* max(u_ij) or max(u_ij + μ_j) */
25             R_{i_max} ← (b_{i_max}, δ̃);
26             R'_{i_max} ← R_{i_max} ∩ R_n;
27             DynamicInsertNode (R'_{i_max}, n);
28             DynamicInsertAndSearch-τ* (n);
29    else
30       /* If n no leaf, descend with BFS. */
31       if ub(n_{left_UB}) ≥ ub(n_{right_UB}) then
32          if ub(n_{left_UB}) ≥ lb* AND ub(n_{left_UB}) > 1 then
33             DynamicInsertAndSearch-τ* (n_left);
34          if ub(n_{right_UB}) ≥ lb* AND ub(n_{right_UB}) > 1 then
35             DynamicInsertAndSearch-τ* (n_right);
36       else
37          if ub(n_{right_UB}) ≥ lb* AND ub(n_{right_UB}) > 1 then
38             DynamicInsertAndSearch-τ* (n_right);
39          if ub(n_{left_UB}) ≥ lb* AND ub(n_{left_UB}) > 1 then
40             DynamicInsertAndSearch-τ* (n_left);
41 /* After finishing, calculate τ* for LB*. */
```

iterates through the elements of n_{UB} and determines for each element if it is part of $n_{left_{UB}}$ and $n_{right_{UB}}$, by determining if an overlap of R_j with n_{left_R} and n_{right_R} respectively occurs. Thus it builds the definite upper bound sets for both successor nodes. When performing the last split, the algorithm adds i of R_i to the appropriate left or right successors lower bound set (lines 15-17).

Algorithm 6.6: DynamicInsertNode(R_i, n)

Data: The region to be inserted R_i, a pointer n to the actual tree node position.

```
1  if R_i ⊆ n_R then
2      if n_left = ∅ AND n_right = ∅ then
3          if R_i = n_R then n_UB ← n_UB ∪ i; n_LB ← n_LB ∪ i;
4          else
5              n_UB ← n_UB ∪ i;
6              Let P = (p_0^1, p_0^2, ..., p_0^d, p_{2^d-1}^1, p_{2^d-1}^2, ..., p_{2^d-1}^d) denote the set of coordinates
               belonging to the points ⟨p_0^(i), p_{2^d-1}^(i)⟩ of region R_i;
7              /* If for R_i and n_R a p_0 or p_{2^d-1} overlaps, leave it out. */
8              if p_0^(n_R) = p_0^(R_i) then
9                  P ← (p_{2^d-1}^1, p_{2^d-1}^2, ..., p_{2^d-1}^d);
10             else if p_{2^d-1}^(n_R) = p_{2^d-1}^(R_i) then
11                 P ← (p_0^1, p_0^2, ..., p_0^d);
12             else
13                 P ← (p_0^1, p_0^2, ..., p_0^d, p_{2^d-1}^1, p_{2^d-1}^2, ..., p_{2^d-1}^d);
14             for p ∈ P in all dimensions do
15                 DynamicAppendNodes (R_i, n, p); /* Algorithm 6.7 */
16                 if R_i ≤ (n_c, n_d) then n ← n_left;
17                 else if R_i ≥ (n_c, n_d) then n ← n_right;
18     else
19         n_UB ← n_UB ∪ i;
20         if R_i ≤ (n_c, n_d) AND (n_left ≠ ∅) then DynamicInsertNode (R_i, n_left);
21         else if R_i ≥ (n_c, n_d) AND (n_right ≠ ∅) then DynamicInsertNode (R_i, n_right);
22         else
23             R_{i_left}, R_{i_right} ← split(R_i, n_c, n_d);
24             DynamicInsertNode (R_{i_left}, n);
25             DynamicInsertNode (R_{i_right}, n);
```

Algorithm 6.7: DynamicAppendNodes(Region R_i, Node n, coordinate p)

Data: Region R_i, actual node position n, coordinate for next splitting hyperplane p.

```
1  dim ← n_disc mod d;
2  c ← (p, dim);
3  R_left, R_right ← split(n_R, c, dim);
4  n_c ← c; n_d ← dim;
5  /* Generate the new left and right nodes. */
6  n_{left_R} ← R_left;
7  n_{right_R} ← R_right;
8  /* Generate entire bound set for both nodes left and right. */
9  forall j ∈ UB do
10     if R_j ∩ n_{left_R} ≠ ∅ then n_{left_UB} ← n_{left_UB} ∪ j;
11     if R_j ∩ n_{right_R} ≠ ∅ then n_{right_UB} ← n_{right_UB} ∪ j;
12 /* Propagate actual known lower bounds to the successors. */
13 n_{left_LB} ← n_{left_LB} ∪ n_LB;
14 n_{right_LB} ← n_{right_LB} ∪ n_LB;
15 if last split for R_i then
16     if R_i ≤ (n_c, n_d) then n_{left_LB} ← n_{left_LB} ∪ i;
17     else if R_i ≥ (n_c, n_d) then n_{right_LB} ← n_{right_LB} ∪ i;
```

6.5.1 Checking for Overlaps

At this point only the determination, if regions *overlap*, remains. This we need in algorithm `DynamicAppendNodes`, at lines 10 and 11. The hard part derives from the fact that regions may

overlap across the domain $\mathbb{D}$.
If in `DynamicAppendNodes` at node n we split up n_R into the subregions n_{left_R} and n_{right_R} by (c, d), both these subregions have to be tested if they contain the elements from the upper bound set of n. We check this by generating a region R_j for each element from n_{UB}, transform it into our domain and crop it to R'_j against the subspace enclosed by the respective subnode with $R'_j = R_j \cap n_{left_R}$ and $R'_j = R_j \cap n_{right_R}$ respectively. Then, we check if $R'_j \cap n_{left_R} \neq \emptyset$ and $R_j \cap n_{right_R} \neq \emptyset$. If the intersection is not empty for the left subnode, we add j to the left nodes upper bound set, the right bound set likewise. If R_j overlaps both left and right successors region, j is added to both upper bound sets. We have to decide this only the elements of UB, and not for all node differences b_i.

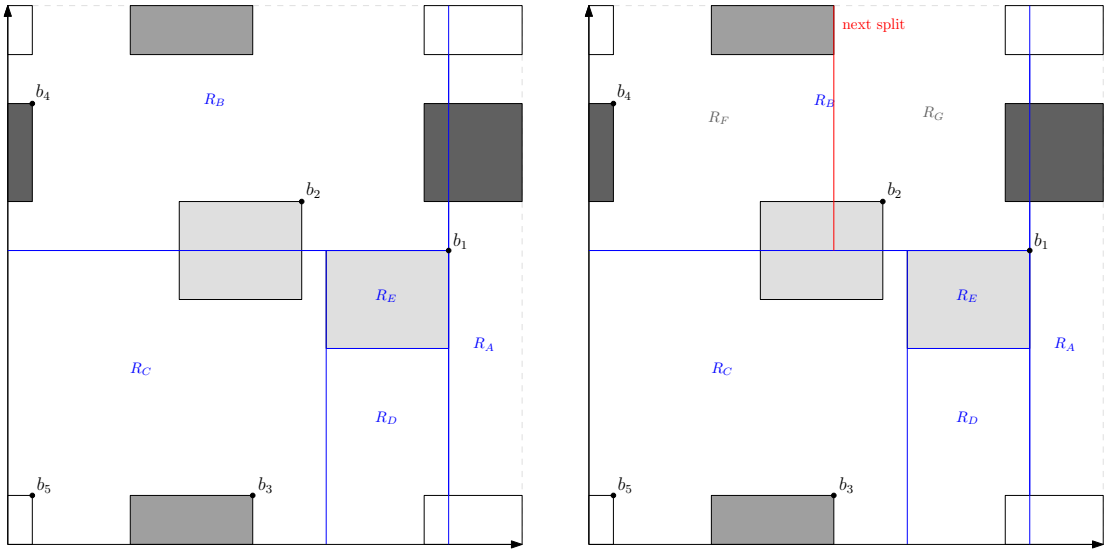
In the static tree, we always know exactly, where our b_i has to be inserted and expand the upper bound sets in the passing. In the dynamic version, we have to explicitly determine if an overlap occurs for each element in UB. As the subspaces defined by n_R are always regular, but we newly generate regions $R_j, j \in UB$ at each such bound set determination, the R_j may be negative. Thus we have to take into account negative parts of R_j, transform R_j into our domain and perform the overlap check on the transformed part.

We speed up the procedure by leaving out the crop part, and perform the check with adjusted intervals, w.r.t. the modulo transformation into $\mathbb{D}$. The check is performed for each dimension. We simply check for a dimension d, if the interval $[R^{(j)}_{min}, R^{(j)}_{max}]$ (adjusted for negative regions) lies in the interval defined by the actual node subspace $[n_{R_{min}}, n_{R_{max}}]$.
For regions crossing the domain border, two interval checks are performed separately for the dimensions having the value < 0. We perform one interval check for the regular part normally and one for the interval of the transformed negative part. Likewise, if R_j crosses the domain border in more dimensions. If no overlap occurs in one dimension, no overlap occurs for R_j and n_R at all, and we can just break the loop which iterates the dimensions d.

Figure 6.13a shows some example segmentation encoding the node subspaces R_A, R_B, R_C, R_D, R_E. We list now the overlaps for regions R_1, R_2, R_3, R_4, R_5, derived from b_1, b_2, b_3, b_4, b_5. The overlaps for node subspaces and regions for b_i found in this figure are:

- Node subspace R_A: The elements that overlap the subspace defined through this branch are the regions for b_4, b_5. The regions R_4, R_5 reach into subspace R_A across the domain border. The left branch counterpart node holds the subspace $R_B \cup R_C \cup R_D \cup R_E$ which overlaps with all R_j. The upper bound at this node contains all b_i.
- Node subspace R_B: The subspace defined by this branch overlaps with regions for node differences b_2, b_3, b_4, b_5. The left branch counterpart node hold subspace $R_C \cup R_D \cup R_E$ and overlaps with regions for b_1, b_2, b_3, b_5.
- Node subspace R_C: The subspace defined by this branch overlaps with regions for b_2, b_3, b_5. Its right branch counterpart node holds subspace $R_D \cup R_E$, which overlaps with b_1, b_5. The remaining segments work likewise.

Figure 6.13b shows an example of a possible next split. In the iteration before, b_1 was inserted, the bounds were generated for each node. The green node is an atomic leaf and needs not to be expanded further. Next element to be inserted is b_3, determined to have the greatest dual value. The region $R2$ is divided by the splitting hyperplane (red line) into $R6$ and $R7$. We regard the upper bound set at $R2$ which is b_2, b_3, b_4, b_5, For each element we decide if it overlaps $R6$ respective $R7$. The new upper bounds are then: $R6_{UB} = \{b_2, b_3, b_4, b_5\}$ since b_3, b_5 "look into" $R6$ across the domain border. $R7_{UB} = \{b_2, b_4, b_5\}$. Figure 6.14 illustrates the resulting segmentation tree.



(a) Some cases of overlaps.

(b) Specific example of a possible next split.

Figure 6.13: Some cases of overlaps. In figure 6.13a, the case for b_4 is interesting since it reaches into the node subspace $R2$ again across the domain border. Parts of subspace b_5 reach into even more node subspaces. When having big $\tilde{\delta}$ such overlaps occur very often. All other cases are trivial. Figure 6.13b shows a specific example of a possible next split. The figure shows the segmentation after insertion of b_1. The region for b_3 will be inserted next. The next split is drawn in red.

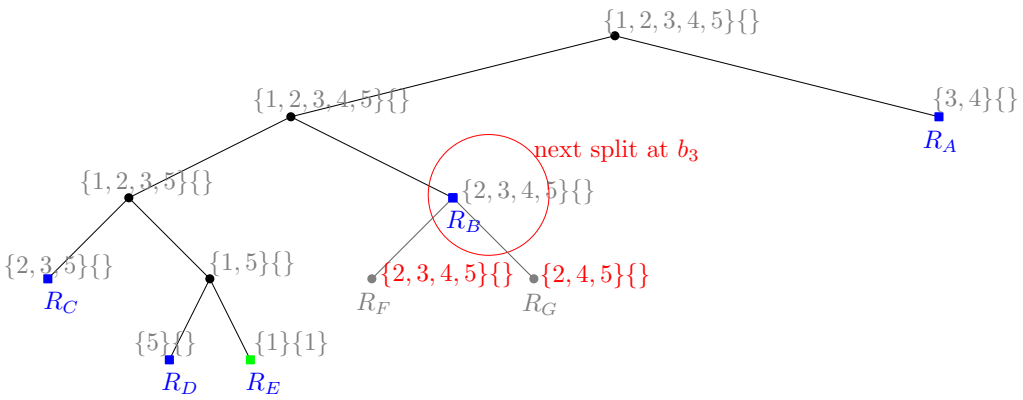


Figure 6.14: Dynamic segmentation tree resulting from 6.13b.

6.5.2 Simulating `DynamicInsertAndSearch-`τ^*

The dynamic segmentation tree is again verified by simulating 1000 random value based iterations. This test works the same as described in subsection 6.4.5 and uses random values to simulate pricing based on the set T^c. The only difference is that we do not build the segmentation tree in advance, but gradually in each simulated pricing/search iteration. We can reset the dynamic tree before each pricing simulation iteration, or let it grow gradually. In our tests, all simulation iterations succeeded, we found each time $\tau^*_{T^c} = \tau^*_{dynseg}$ and the pricing in the dynamic segmentation tree thus was showed to work correctly.

6.6 Pricing Algorithms

Since the pricing procedure is generic, both the single and multi commodity commodity flow employ the same pricing algorithm. The algorithm `StaticTreeSearch-`τ^*, which determines τ^* in a previously constructed static tree was presented in section 6.4.3 and is the basis for the *static tree pricer*. The *dynamic tree pricer* employs the algorithms `DynamicInsertAndSearch-`τ^*, `DynamicInsertNode` and `DynamicAppendNodes` from section 6.5. Additionally, we use both pricing algorithms to determine variables with negative Farkas coefficients.
Either algorithm `StaticPricer` or algorithm `DynamicPricer` is called as PricerAlgorithm in algorithm 5.1 from the previous chapter at line 10.

A pseudo code overview for the `StaticPricer` is listed in algorithm 6.8. The set of minutiae data points V and $\tilde{\delta}$ are assumed to be available as global variables. Input for each pricing iteration are the actual dual values u_{ij} for the template constraints and μ_j for node-label constraints, if these are employed by the model. The entire segmentation tree is built only at the first call (lines 1-2). After this, the pricer determines the lower bound set defining τ^* with the according procedure (line 3) and generates the new variable t_{new} (line 5), which is added in the end to the current LP after having returned it to the branch-and-price framework (line 6).
If we allow no variable duplicates, the algorithm marks the newly priced variable t_{new}, so that he pricer does not find the same variable again (line 4). When doing so, the algorithm `Static TreeSearch-`τ^* has to be altered, so that it finds only unmarked result values.

Algorithm 6.8: StaticPricer($u_{ij}, [\mu_j]$)

Data: Dual variables for arc-label constraints u_{ij}, optional node-label constraints duals μ_j. At this point, the input minutiae data set V and $\tilde{\delta}$ are given.
Result: Returns the new column(s) t_{new} with $\arg\max\{\bar{c}_t\}$.

```
1 if segtree = ∅ then
2    segtree ← StaticSegmentation(V, δ̃);
3 LB* ← StaticTreeSearch-τ*(n_root);
4 mark node for LB* as found; /* optional */
5 t_new ← generate variable for the standard template arc from LB*;
6 return t_new;
```

The algorithm `DynamicPricer` is listed in pseudo code 6.9. Again, V and $\tilde{\delta}$ are given. Input are again the actual dual values u_{ij} and, if employed, μ_j. While searching for the lower bound set defining τ^*, the algorithm builds the dynamic tree (line 1). The result is again the variable to be added to our current LP (lines 4-5).

Algorithm 6.9: DynamicPricer($u_{ij}, [\mu_j]$)

Data: Dual variables for arc-label constraints u_{ij}, optional node-label constraints duals μ_j.
Result: Returns the new column(s) t_{new} with $\arg\max\{\bar{c}_t\}$.

```
1 LB* ← DynamicTreeSearch-τ*(n_root);
2 dyntree = ∅; /* optional */
3 mark node for LB* as found; /* optional */
4 t_new ← generate variable for the standard template arc from LB*;
5 return t_new;
```

For the dynamic tree pricer, two alternatives exist. Either we let it expand the segmentation tree in each pricing iteration, thus leaving out the step in line 2, or we clean up the entire tree before

each pricing iteration (line 2) and let it build in each such pricing iteration only the tree segments that are needed to solve the actual pricing problem. Again, variable duplicates may be excluded from search by marking found t_{new}. The according parts in `DynamicInsertAndSearch-`τ^* have to be altered.

6.6.0.1 Outlook

The following chapter 7 describes how to extract the set of candidate template arcs T^c out of the presented segmentation tree, and regards the static as well es dynamic version. Thus, we create an alternative to the preprocessing method from [ChwRai09]. The framework, where branch-and-price is implemented, as well as implementation details, will be subject in chapter 8.

Chapter 7

Extracting T^C from the Segmentation Tree

As already hinted at in chapter 6, besides τ^* another result can be extracted from the segmentation tree. This option arose with the need for some test or verification algorithm in order to determine if the segmentation tree is correct. So we extracted all non empty template arcs from the tree and noted, that by omitting dominated ones, we actually get the set of non-dominated template arcs T^c, a set until now determined by the preprocessing method from [ChwRai09], in the following named preprocessing. All k-MLSA solution strategies presented in 1.2.2 need this set, the only exception is branch-and-price. By implementing a strategy that intelligently traverses the segmentation tree and extracts the set T^c, we obtain an efficient alternative to the existing the preprocessing step.

7.1 Overview

This chapter describes, how to extract the set T^c from an already constructed static or dynamic segmentation tree. Basically we traverse the tree by means of the bound sets UB and LB, which at each node encode information about the node differences b_i. At a leaf, the bound sets $UB = LB$ represent the set of b_i, that encodes a template arc. Thus we denote such a leafs LB or UB as bound set l. In the traversing process, we use a multiset L to save the non-dominated bound sets l. This set L in the end holds all bound sets where for each such bound set we can extract[1] the standard template arc forming T^c. When searching T^c, we do not have and require dual values, but the bound sets themselves are of interest. At each leaf, that encodes an atomic region, the upper bound set equals the lower bound set $UB = LB$. As we search for template arcs, the leafs containing $UB = LB = \emptyset$ are irrelevant, since these leafs encode areas, where no b_i lie and therefore lead to no template arc. Now, the concept of domination arises for varying purposes, e.g. we must determine for such a bound set, if it is dominated or not.

The domination concept arises for multiple applications. One use is, that we want to decide, which bound set LB from some leaf is added to the multiset L of non-dominated bound sets. For this, we use the domination concept, defined in the following as *domination of template arcs*. As the name implies, we compare bound sets representing some template arcs, and determine dominated and dominating bound sets.

Further, we want to identify at each visited node, which one of the successors to visit and if one or both branches may be omitted. For this we need a more specific domination concept. Each node in the segmentation tree holds an upper bound set UB. Based on this UB, we want to decide, if the respective subtree having UB must be visited or can be omitted. We want to

[1] Described in sections 4.2.1 and 6.4.3.1.

visit only non-dominated subtrees. For this, we look again at the multiset L and extend our domination concept to *domination of subtrees*. By using this extended domination concept, we determine if such a subtree is dominated or not, by comparing the elements from L with our actual subtrees UB. Dominated subtrees are excluded from the traversing process. Further, at a node n, we can deduce some further information: If we additionally compare the left subnodes upper bound $n_{left_{UB}}$ with the right subnodes upper bound $n_{right_{UB}}$, we can identify additional dominated branches and leave them out in the search phase. In the following, we define and describe in detail the presented concepts.

7.1.0.2 Domination of Template Arc defining Bound Sets

We need this concept, when constructing the multiset L of non-dominated template arc defining bound sets. As we traverse the segmentation tree, we encounter leafs having $UB = LB \neq \emptyset$, with LB being a bound set, out of which a standard template arc is extracted. Having some leaf with a LB, we have to decide, if we add it to our multiset L or not. This we do by determining if LB is *dominated* by some element in L or if in turn LB dominates one or more element from L. As the elements of L are nothing other than LB from previously found leafs, we define *domination of template arc defining bound sets*, or *domination of template arcs* as follows. This definition adheres to definition 9, chapter 4, for convenience we rephrase it in terms of UB and LB.

Definition 12 (**Domination of Template Arcs**). *Given two leafs having $UB = LB \neq \emptyset$. A template arc defining bound set LB_A* dominates *another template arc defining bound set LB_B, if $LB_B \subseteq LB_A$. Sets $LB = \emptyset$ are always dominated.*

Thus, at each found leaf, we can decide, if our leafs LB is dominated by some element in L or not, or if we have to remove some $l \in L$ because they are dominated by LB.

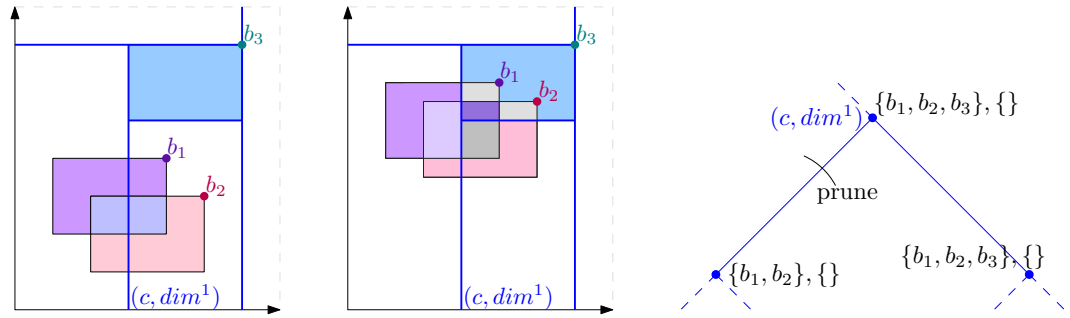
For example, we have some intermediate multiset L that holds $l_1 = \{b_1, b_2, b_3\}$ and $l_2 = \{b_3, b_4\}$. In the traversing process we find an $LB = \{b_1, b_2\}$. This template arc defining bound set is not added to L, since $LB \subseteq l_1$. The same would be for an $LB = \{b_1, b_2, b_3\}$. If instead we find a leaf having $LB = \{b_1, b_2, b_3, b_4\}$, all elements in L that are dominated by this LB are removed, and LB added to L instead of l_1, l_2. If we find a leaf having $LB = \{b_1, b_2, b_5\}$, we add it to L, since it is not dominated by any l.

7.1.0.3 Domination of Subtrees

By looking at the upper bound sets of the nodes in the segmentation tree, we use the enhanced domination concept for identifying dominated branches, it acts thus as *bounding strategy*.

Definition 13 (**Domination of Subtrees**). *Given a node n, with two successors having non-empty upper bound sets UB_A and UB_B. The branch having UB_A* dominates *the branch UB_B, if $UB_B \subseteq UB_A$, but only if there are no domain crossing elements in both sets that may lead to differing template arcs somewhere in the subtree. In order to determine this, we partition each UB into two disjoint sets $UB = UB' \cup UB''$, where UB'' contains all elements representable across the domain border regarding the ring structure, whereas $UB' = UB \setminus UB''$. Then, UB_A dominates UB_B, iff $UB_B \subseteq UB_A \wedge UB''_B \subseteq UB''_A$. A branch having $UB = \emptyset$ is always dominated.*

If at node n, the two successors result in $UB_A \subseteq UB_B$ and none of the elements from both bounds cross the domain border, we can omit the subtree having UB_A, since it is dominated.
Thus we omit subtrees that are dominated, but only if no elements from both UB do not cross the domain border. No matter what template arcs we finally will find in a dominated subtree, they are always dominated by the template arcs found in the other subtree (or by some element in L). If some elements cross the domain border with respect to the ring structure, no information about



(a) Example 1a. (b) Example 2a. (c) Bounds for Ex. 1a and 1b.

Figure 7.1: Illustration for example 1a and 1b depicting some bound sets in the segmentation tree.

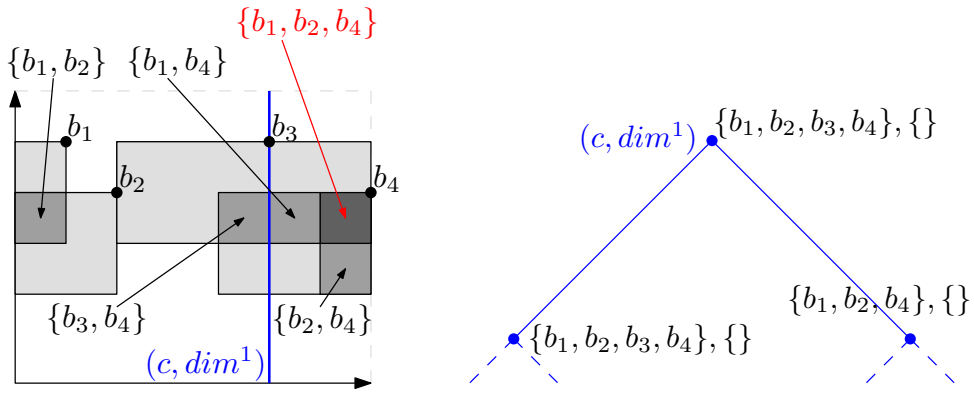
domination can be extracted. The following examples clarify the concept.

Example 1a. The segmentation for example 1a is depicted in figure 7.1a, the discussed segmentation subtree in figure 7.1c. We are at some intermediate node n with $UB = \{b_1, b_2, b_3\}$ and the splitting hyperplane (c, dim). The elements in UB may be $<$ and/or $>$ than (c, dim) at n. Let's assume that (c, dim) splits the area defined by n into two parts producing $UB_{left} = \{b_1, b_2\}$ and $UB_{right} = \{b_1, b_2, b_3\}$, the lower bounds are empty. All elements b_1, b_2, b_3 are regular regions and do not cross the domain border. Both upper bounds encode the points that maximal can be represented by a template arc lying in the respective subspace defined by this subnode, the lower bound indicates all template arcs that can be represented by a template arc lying in the respective subspace defined by this node. Thus, the subtree n_{left} may produce the template arcs $\{b_1\}$, $\{b_2\}$ or $\{b_1, b_2\}$, since LB_{left} is empty and we do not yet know anything about the definitive template arcs. A template arc $\{b_1, b_2\}$ would only be given if additionally $LB_{left} = \{b_1, b_2\}$. The right counterpart subtree works similarly. In subtree n_{right} the following template arcs are possible: $\{b_1\}$, $\{b_2\}$, $\{b_3\}$, $\{b_1, b_2\}$, $\{b_1, b_3\}$, $\{b_2, b_3\}$ or $\{b_1, b_2, b_3\}$.
A relation exists between both subtrees: If in the left subtree b_1 overlaps b_2, such an overlap exists also in the right subtree for sure and vice versa. In the right subtree we have additionally b_3. At this point it does not matter, if it overlaps b_1 and/or b_2, but it is immanent, that in the right subtree, also the correct template arc for b_1, b_2 is formed.
In this example 1a, b_1 overlaps b_2, but b_3 overlaps none of these two, the according left subtree is $UB_{left} = \{b_1, b_2\}$. Somewhere in this left subtree, the template arc $\{b_1, b_2\}$ lies. The right subtree has $UB_{right} = \{b_1, b_2, b_3\}$, the template arcs will result somewhere in this subtree as $\{b_1, b_2\}$ and $\{b_3\}$. Thus UB_{right} dominates UB_{left}.

Example 1b. The segmentation for example 1b is depicted in figure 7.1b, the resulting segmentation subtree part is again figure 7.1c. In this case b_1 overlaps b_2, and b_3 overlaps both sets, but only in the right subtree. The left subtree will produce the template arc $\{b_1, b_2\}$, which is dominated by the template arc produced in the right subtree $\{b_1, b_2, b_3\}$.

Now we *combine both domination concepts* and enhance our *bounding strategy*. By comparing an upper bound set UB for some subtree with the elements from multiset L, we can prune subtrees that would lead to template arcs dominated by the elements in L. If $\forall l \in L : UB \subseteq l$ we can safely omit this subtree.

For example, if at node n, the upper bound is $UB = \{b_1, b_2\}$, and we have an $l \in L$ consisting of $\{b_1, b_2, b_3\}$, then $UB \subseteq l$ and we omit this branch, since no permutation better than the already found l can be found in this subtree. Thus we check at each node n if one or both n_{left} and n_{right} are dominated by some $l \in L$.
Example 2. This example is depicted in figure 7.2. The upper bound set at n is $UB = \{b_1, b_2, b_3, b_4\}$



(a) Example 2. (b) Bounds for Ex. 2.

Figure 7.2: Illustration for example 2 depicting some domain crossing regions in 7.2a and corresponding segmentation tree in 7.2b.

and $L =$. The left subtree has $UB_{left} = \{b_1, b_2, b_3, b_4\}$, whereas the right subtree has $UB_{right} = \{b_1, b_2, b_4\}$. In b_1 and b_2 are irregular regions crossing the domain border w.r.t. the ring structure. Although the left UB set dominates the right UB set, we must search both subtrees, since there are two elements crossing the domain border, namely b_1, b_2. At this point we know not for sure, if the left subtree will produce a $\{b_1, b_2, b_3, b_4\}$ that would dominate the bound set of the right subtree. In the example it turns out, that in the left subtree we find the template arc $\{b_3, b_4\}$ and in the right subtree, we find $\{b_1, b_2, b_4\}$. All other possible bound set representing a template arc are dominated by these two template arcs.
When anticipating the traversing process, it results as follows. Since $L =$ and we have at n elements that cross the domain border we search both subtrees. We first descend left, and find the bound set $\{b_1, b_2\}$ and add it to L. Next, the bound set $\{b_3, b_4\}$ is found and added to L. The recursions unwind, and we are again at n, where we descend right. Here, we find first $\{b_1, b_2, b_4\}$, since the algorithm will search in branches having a greater bound set $|UB|$. This template arc dominates $\{b_1, b_2\} \in L$, so we remove this template arc from L and insert $\{b_1, b_2, b_4\}$ instead. The other branches containing $\{b_1, b_4\}$ and $\{b_2, b_4\}$ are then pruned, since they will result in no better template arcs.

All the relations of the template arc defining bound sets are already encoded into the segmentation tree. We utilize now both domination concepts, when searching for the non-dominated bounds sets that define them and omit the irrelevant branches containing dominated bound sets. In the end of the traversing process, we extract the standard template arcs from the non-dominated bound sets in L and get T^c.

In the following we describe three variants for the bounding strategy. All three strategies base on best first search.

- Best first search for determining T^c,
- UB-driven traversing, using a simple bounding strategy for determining T^c,
- Advanced bounding traversal for determining T^c. This traversing strategy prunes *all* branches that can be identified as dominated.

All strategies work for the static segmentation tree as well for the dynamic segmentation tree. For convenience, we explain all developed algorithms for the static tree. All algorithms are easily adapted for the dynamic segmentation tree. In the following, we shortly describe all strategies.

7.2 Traversing the Static Segmentation Tree

The three traversing strategies are ordered by complexity of the adopted bounding technique. They differ in the used domination concepts as well as the number of checked $l \in L$. Basis is standard best first search. When using a good bounding strategy as advanced bounding traversal, presented in the following, the algorithm checks entire bound sets at many tree leafs. So, the algorithm is estimated to get slower the more accurate bounding is, but in contrast to this, the number of visited nodes should decrease by a big factor the better the bounding strategy.

7.2.1 Best First Search for Determining T^c

This strategy uses only domination of template arc defining bound sets. We traverse the segmentation tree recursively by best first search and observe the cardinality of the upper bound at each node. Found lower bound sets are saved in multiset L. Thus, the algorithm finds big bound sets first. Hopefully, a lot of dominated leafs are then pruned on the basis of these big bound sets. At each recursion we decide for the actual node the successors processing order. We always follow the branch with the greater bound size $|UB|$ first, thus finding the leafs having large bound sets first. If we reach a leaf n with $UB = LB \neq \emptyset$, we check if we have to save the LB set by comparing it to the set of lower bounds L found until now. If no element from L dominates LB, we insert LB to L. If the set LB is dominated by some element from L, we do not save it to L. If LB dominates one or more elements in L, we remove all dominated elements and insert LB. After having completed the traversing process and filled L, in the end we determine for each element $l \in L$ the standard template arc τ.

By recalling `StaticAppendNodes`, at an intermediate node we have always two successors, since `StaticAppendNodes` appends exactly two successors, never one.

Naturally, in practice this strategy consumes a lot of processing time, since we do not prune any subtrees except empty ones. As we have to compare many LBs, this is the main bottleneck, especially if the bound sets are big. The result set was proved to be always equal to T^c. Pseudo code for `TraverseBFS` is listed in algorithm 7.1. At line 2 we decide again if we are at a leaf or not. We save the actual bound set LB to L if it is not dominated by some $l \in L$, or remove dominated l from L and insert LB (lines 3-7). Lines 9-14 select the branch having a greater $|UB|$ to visit first performing thus best first search.

Algorithm 7.1: TraverseBFS(Node n)

Data: Found non-dominated bound sets are saved to the multiset L.
Result: Fills L with the set of non-dominated bound sets. Out of L we extract the non-dominated template arcs set T^c.

```
1  if n ≠ ∅ then
2      if n_UB = n_LB then
3          save ← true;
4          for l ∈ L do
5              if n_LB ⊆ l then save ← false; break; /* Do not save n_LB */
6              else if l ⊂ n_LB then L ← L \ l;
7          if save then L ← L ∪ n_LB;
8      else
9          if |n_left_UB| ≥ |n_right_UB| then
10             /* Follow the left subtree first. */
11             if |n_left_UB| ≠ 0 then TraverseBFS (n_left);
12             if |n_right_UB| ≠ 0 then TraverseBFS (n_right);
13         else
14             As lines 10-12, reverse order. First right, then left branch.
```

7.2.2 UB-Driven Traversing for Determining T^c

Again we traverse the segmentation tree with a best first search, and drive the search toward promising regions with the aid of the cardinality of the upper bound. If we arrive at a leaf, we proceed as in `TraverseBFS`, subsection 7.2.1. We save dominating LB to L and do not save and/or remove dominated LBs. At intermediate nodes, we use the following bounding strategy:

- If a left or right successor has an upper bound size $|UB| = 0$, we do not follow this branch, thus pruning it.
- If the left successors upper bound set is a subset of the right successors upper bound set $n_{left_{UB}} \subseteq n_{right_{UB}}$, we additionally check if in the elements contained in both sets have bounding boxes that cross the domain border. Such elements have parts on both sides of the actual splitting hyperplane and we check, if the intersection set is not empty by $n_{left_{UB}} \cap n_{right_{UB}} \neq \emptyset$. So, we have to determine only for the elements in this intersection set if they cross the domain border. If no crossing elements exist in the intersection set, we prune the left branch since it is dominated. If such elements exist, we have to visit this branch. We do the same for $n_{right_{UB}} \subseteq n_{left_{UB}}$.
- If $n_{left_{UB}} \not\subset n_{right_{UB}}$ we check if $n_{left_{UB}}$ was already found in L. If $n_{left_{UB}} \in L$, we do not follow this branch, else we follow it. The case $n_{right_{UB}} \not\subset n_{left_{UB}}$ works the same.

The pseudo code for UB-`DrivenTraversing` is listed in algorithm 7.2. The differing lines are 6-19, where an additional check was inserted, if a subnodes UB dominates its counterpart node (line 11, 15). A recursive call to UB-`DrivenTraversing` is only executed, if a subnodes UB contains elements crossing the domain border (lines 12, 16) or if the actual UB is not part of L (lines 13, 17).

Algorithm 7.2: UB-DrivenTraversing(Node n)

Data: Found non-dominated bound sets are saved to the multiset L.
Result: Fills L with the set of non-dominated bound sets. Out of L we extract the non-dominated template arcs set T^c.

```
1  if n ≠ ∅ then
2      if n_UB = n_LB then
3          Lines 3-7 from algorithm TraverseBFS, listing 7.1.
4      else
5          /* If left or right branch are empty, prune.  */
6          if |n_left_UB| = ∅ then UB-DrivenTraversing (n_right);
7          else if |n_right_UB| = ∅ then UB-DrivenTraversing (n_left);
8          else
9              if |n_left_UB| ≥ |n_right_UB| then
10                 /* Check, if we have to visit the left subtree */
11                 if n_left_UB ⊆ n_right_UB then
12                     if in n_left_UB ∃ elements crossing 𝔻 then UB-DrivenTraversing (n_left);
13                 else if n_left_UB ∉ L then UB-DrivenTraversing (n_left);
14                 /* Check, if we have to visit the right subtree */
15                 if n_right_UB ⊆ n_left_UB then
16                     if in n_right_UB ∃ elements crossing 𝔻 then UB-DrivenTraversing (n_right);
17                 else if n_right_UB ∉ L then UB-DrivenTraversing (n_right);
18             else
19                 As lines 10-17, reverse order.  First right, then left branch.
```

After having completed the traversing process, we determine again for each $l \in L$ the standard template arc τ. The result set was again shown to be equal to T^c by computational experiments. This bounding strategy is not "complete", since we only determine for a subtree's upper bound set if it is exactly contained in the multiset L. This check is efficient, when L is implemented

as a binary tree using the smallest element as discriminator, where finding a specific element takes logarithmic time. The present bounding strategy is a "structural property exploit", it takes advantage of the fact, that similar bound sets are located in the same subtrees.

7.2.3 Advanced Bounding Traversal for Determining T^c

This algorithm differs from listing 7.2 only in the bounding strategy. The lines 10-17 from 7.2 become 10-26 in listing 7.3. We enhance the bounding strategy as follows: Here, when deciding if to visit a branch or not, instead of checking only if $n_{left_{UB}} \in L$, we check the entire multiset L, if there exists a bound that dominates $n_{left_{UB}}$. If we find such a dominating element in L we do not have to follow the actual branch, else we follow it. We call this bounding approach "complete", since we cut away all dominated branches, by processing all the bound sets found until now. But, it may happen that in the process first some dominated bound sets are found (as in UB-`DrivenTraversing`), which later are replaced by the dominating bound set.

Again, from L we finally extract T^c. This was confirmed by our tests to work correctly. As for the bounding strategy, in the worst case we search at each node through the entire multiset L, which can be huge. This behaviour is estimated to be inferior to UB-`DrivenTraversing`, especially when having big bound sets. Only *visiting* $2-10$ millions of tree nodes with depth first search was measured to perform more or less in a very small time. The UB-driven traversing has the advantage of performing only a simple check, if the actual $UB \in L$, at each node. It may visit more irrelevant branches, but we have a lot fewer overall comparisons of bound sets to each other. Advanced bounding traversal performs the expensive test, if $UB \in L$ and thus searches at every node in the multiset L. Consequently, for each node (and therefore a branch) we can determine if it is dominated by the actual L or not, thus pruning more branches and leafs as the UB-`DrivenTraversing`. So, the amount of visited nodes for advanced bounding traversal is significantly smaller. But we have to perform much more comparisons of elements in L and actual UB for determining this. In chapter 9, where we present computational experiments, it will be evaluated, if it is better to visit less branches, or to use a cheaper bounding strategy. We estimate advanced bounding traversal to be advantageous when having small bound sets and big segmentation trees, whereas the cheaper pricing strategy is estimated to be better when having big bound sets.

Pseudo code `AdvBoundingTraversal` is listed in algorithm 7.3. The new flag `found`, initialized to `false` in lines 10 and 19, is used for iterating through L (lines 14, 23) and determining if an actual UB is not dominated by any $l \in L$ (lines 15, 24). We visit a branch only if its UB was determined to eventually bring forth a non-dominated bound set (lines 12, 17, 21 and 26).

7.3 Traversing the Dynamic Segmentation Tree

All presented strategies also work for the dynamic tree. Since the dynamic tree merges traversing and building, we now have to incorporate the relevant bounding parts. We present only UB-driven traversing for dynamic segmentation tree, the remaining two traversing strategies are constructed accordingly.

7.3.1 UB-Driven Traversing for the Dynamic Segmentation Tree

Basis is the algorithm `DynamicInsertAndSearch-`τ^*, listing (6.5). We replace the tree traversing part of algorithm (6.5), lines 30-40, and get the algorithm `DynamicSearch-`T^c, having the bounding strategy from UB-`DrivenTraversing` at lines 30-41. Further we must alter the part where unconstructed subtrees are expanded such that at leafs we save the non-dominated LB to L. The

Algorithm 7.3: AdvBoundingTraversal(Node n)

Data: Found non-dominated bound sets are saved to the multiset L.
Result: Fills L with the set of non-dominated bound sets. Out of L we extract T^c.

```
1  if n ≠ ∅ then
2      if n_UB = n_LB then
3          Lines 3-7 from algorithm TraverseBFS, listing 7.1.
4      else
5          if |n_left_UB| = ∅ then AdvBoundingTraversal (n_right);
6          else if |n_right_UB| = ∅ then AdvBoundingTraversal (n_left);
7          else
8              if |n_left_UB| ≥ |n_right_UB| then
9                  /* Check if we have to follow the left subtree */
10                 found ← false;
11                 if n_left_UB ⊆ n_right_UB then
12                     if in n_left_UB ∃ elements crossing 𝔻 then AdvBoundingTraversal (n_left);
13                 else
14                     forall ∀l ∈ L do
15                         if n_left_UB ⊆ l then found ← true; break;
16                     /* n_left_UB was already found, prune this branch. */
17                     if ¬ found then AdvBoundingTraversal (n_left);
18                 /* Check if we have to follow the right subtree */
19                 found ← false;
20                 if n_right_UB ⊆ n_left_UB then
21                     if in n_right_UB ∃ elem. crossing 𝔻 then AdvBoundingTraversal (n_right);
22                 else
23                     forall ∀l ∈ L do
24                         if n_right_UB ⊆ l then found ← true; break;
25                     /* n_right_UB was already found, prune this branch. */
26                     if ¬ found then AdvBoundingTraversal (n_left);
27             else
28                 As lines 9-26, reverse order. First right, then left branch.
```

according new lines are 15-20 in `DynamicSearch-`T^c replacing lines 15-20 from `DynamicInsert-AndSearch-`τ^*. Additionally, at incomplete nodes having $UB \neq LB$, we replace the lines 8 and 24 from `DynamicInsertAndSearch-`τ^*, where we select the best b_i from the set $\Phi = UB \setminus LB$, that were not yet segmented. Since we have no dual values and each b_i must be segmented anyway (since we search T^c), in `DynamicSearch-`T^c at lines 8 and 23 we simply segment the first element from Φ. The resulting algorithm `DynamicSearch-`T^c is listed in pseudo code 7.4. The algorithms `DynamicAppendNodes` (listing 6.7) and `DynamicInsertNode` (listing 6.6) remain unchanged.

7.4 A Non-Dominated Segmentation Tree

At this point the question, if a segmentation tree containing only non-dominated branches can be built, remains as future work. The non-dominated segmentation tree would be built by inserting all b_i sequentially into an empty tree. But at the moment it is unclear, if for each inserted b_i can be decided based on the incomplete construction status, if a subtree containing b_i can be definitely identified as as dominated a priori (and thus must be built) or not.

An algorithm sketch mainly based on checking overlaps of b_i at the nodes n was developed. We only outline the algorithm, since the work was not yet finished. Basis is again the algorithm `StaticInsert`, listing (6.2). By inserting $b_i \in B$ one by one, we construct only non-dominated branches, and mark the dominated ones with a flag. If a b_i generates no new non-dominated

Algorithm 7.4: DynamicSearch-T^c(Node n)

Data: Found bound sets are saved to the set L.
Result: Fills L with the set of non-dominated bound sets. Out of L we extract T^c.

```
1  if n ≠ NULL then
2     if (n_left = ∅) AND (n_right = ∅) then
3        /* Leaf, determine whether to save n_LB or if to expand the tree. */
4        if (|n_UB| = 0) AND (|n_LB| = 0) then
5           /* Don't save this bound, since it is empty. */
6        else if (|n_UB| > 0) AND (|n_LB| = 0) then
7           /* n_LB is empty, expand this branch with first unsegmented b_i. */
8           b_i ← UB_1; R_i ← (b_i, δ̃);
9           /* Crop and transform negative regions R_i into 𝔻. */
10          R'_i ← R_i ∩ R_n;
11          DynamicInsertNode (R'_i, n);
12          DynamicSearch-T^c (n);
13       else if (|n_UB| > 0) AND (|n_LB| > 0) then
14          if n_UB = n_LB then
15             /* Do not expand n. Save n_LB to L if not dominated. */
16             save ← true;
17             for l ∈ L do
18                if n_LB ⊆ l then save ← false; break; /* Do not save n_LB */
19                else if l ⊂ n_LB then L = L \ l;
20             if save then L = L ∪ n_LB;
21          else
22             /* If n_UB ≠ n_LB, expand branch. Get b_i ∉ LB. */
23             Φ = UB \ LB; b_i = Φ_1;
24             R_i ← (b_i, δ̃);
25             R'_i ← R_i ∩ R_n;
26             DynamicInsertNode (R'_i, n);
27             DynamicSearch-T^c (n);
28    else
29       /* If node n is no leaf, descend with UB-driven traversing strategy.
            */
30       if |n_left_UB| = 0 then DynamicSearch-T^c (n_right);
31       else if |n_right_UB| = 0 then DynamicSearch-T^c (n_left);
32       else
33          if |n_left_UB| ≥ |n_right_UB| then
34             if n_left_UB ⊆ n_right_UB then
35                if in n_left_UB ∃ el. crossing 𝔻 then DynamicSearch-T^c (n_left);
36             else if n_left_UB ∉ L then DynamicSearch-T^c (n_left);
37             if n_right_UB ⊆ n_left_UB then
38                if in n_right_UB ∃ el. crossing 𝔻 then DynamicSearch-T^c (n_right);
39             else if n_right_UB ∉ L then DynamicSearch-T^c (n_right);
40          else
41             As lines 33-39, reverse order. First right, then left branch.
```

template arc at some dominated branch, we only add its index to the actual upper bound set. A dominated branch is unmarked and expanded, if a new b_i comes across, that generates a new template arc somewhere in this subtree. Having such a case, elements from the dominated branch, that were not yet segmented, must be re-segmented.

If such a tree can be built, it is an improvement for both the search for T^c and the pricing problem. In both cases, performance may be improved by only creating the (hopefully much smaller) non-dominated branches instead of a full segmentation tree.

Chapter 8

Implementation

This chapter summarizes all implementation specific details. Basis for our implementation is the framework where [ChwRai09] and [Dietzel08] implemented all previous solution strategies. All algorithms have been integrated into this existing framework.

The algorithms have been implemented in `C++`, compiled and linked with `g++-4.1`. The existing code makes use of the following libraries: The `Standard Template Library (STL)` from *Silicon Graphics (SGI)*, the `Library of Efficient Data types and Algorithms (LEDA)` from *Algorithmic Solutions GmbH*[1] which collects a variety of algorithms for graph theory and computational geometry, the library `Boost` which extends `C++`, and `ILOG CPLEX`© `11.2`, an optimization software package for solving (among other) linear and integer programs. `CPLEX` is a state of the art commercial solver software that is able to solve linear and integer programs with millions of variables by using linear and non-linear methods.

8.1 Branch-and-Price Framework

A specialized branch-and-price framework has been integrated into the existing code. Our choice fell upon `Solving Constraint Integer Programs (SCIP)` [SCIP] developed by the *Division Scientific Computing, Department Optimization* at the *Konrad-Zuse-Zentrum für Informationstechnik Berlin* in cooperation with *TU Braunschweig*, *TU Darmstadt* and *Siemens AG*. We used `SCIP 1.2.0`.
This framework implements and provides most algorithms, tools and diagnostics that we need for branch-and-price. `SCIP` is highly flexible, implemented in `C` and provides `C++` wrapper classes which we used for integrating it into the existing code. Most of its features are realized with easily extensible plugins, and include all sorts of constraint handlers, presolvers, branching rules, relaxators and primal heuristics. Interfaces for implementing variable pricers and cut separators exist. `SCIP` provides facility for constraint integer programming and branch-and-cut-and-price. The authors note that "`SCIP` is currently one of the fastest non-commercial mixed integer programming solvers on the market".

An introduction to `SCIP` is [Schwa08]. A detailed description to the framework is [AcBeKoWo08] and [Achter07], who developed the original framework. The used documentation [Pfe07, Ber07, Ach07, Wolt07, Ach07-2] as well as implementation examples can be found at the website [SCIP]. `SCIP` also includes the open LP solver `SoPlex` which is, compared to other freeware solvers, very fast, but unfortunately not competitive enough for our purposes. As `SCIP` integrates support and interfaces for other solvers, we chose `ILOG CPLEX`© `11.2` as LP solver.

[1] Originally by *Max Planck Institute for Informatics Saarbrücken.*

8.2 Implementing SCF and MCF formulation in SCIP

SCIP saves variables and constraints by the data types SCIP_VAR* and SCIP_CONS*, managed in an arbitrary container. The SCF and MCF formulations are loaded into the framework with provided interface functions. Each problem was initialized with SCIPcreateProb, the objective sense set by SCIPsetObjsense to SCIP_OBJSENSE_MINIMIZE. Our solution was constrained to be integral by SCIPsetObjIntegral and to be less than k by SCIPsetObjlimit.

All variables were created by SCIPcreateVar (where we can specify name, variable type, limits, coefficients and so on) and added to the problem by SCIPaddVar. The variables were assembled into constraints by using SCIPcreateConsLinear and SCIPaddCons. For each constraint a lot of parameters can be specified: We defined all constraints to be part of the initial LP and disabled separating in the solving process. Enforcing of constraint was enabled, since we have no redundant constraints. Feasibility checking for each constraint in each branch-and-bound node was enabled. Additionally we defined constraints to be valid globally, rather than locally at some node. The arc-label constraints and node-label constraints[2] were set to be modifiable, this is needed for pricing. The option for dynamic constraints (needed when cuts are separated as constraints) was disabled. The flags for a constraint to be removable from the relaxation was disabled and the option, if constraints may stick at the BB node where they were added, was enabled.

Our pricer classes inherit from the C++ wrapper object interface scip::ObjPricer, and thus implement the virtual functions scip_init, scip_redcost and scip_farkas. By providing these functions, SCIP is able to call back the according user implemented methods from an arbitrary branch-and-bound node or state. SCIP is notified about own pricers by SCIPincludeObjPricer and SCIPactivatePricer, after having constructed the pricer object.

We started the solution process by SCIPsolve. SCIP uses presolvers and relaxators for each node to determine feasibility and objective function values, saves actual bounds and in the process calls our functions scip_redcost and scip_farkas respectively. Dual variables are provided by the framework through the constraint handler by using SCIPgetDualsolLinear. Farkas coefficients are extracted by SCIPgetDualfarkasLinear. Finally, we added the determined column to the according constraints in the global model by first doing SCIPcreateVar and then SCIPaddPricedVar.

8.2.1 SCIP Plugins

The solution process depends heavily on the plugins, that were defined[3] before loading the model. In the following we only name the plugins that have been used. A detailed plugin description would get to long and can be looked up in the SCIP documentation, or in [Achter07, AcBeKoWo08].

Constraint handlers influence the way the constraints are relaxed and checked for validity. We used SCIPincludeConshdlrLinear. The framework is able to transform constraints into a tighter form. In our case some constraints were tightened automatically by enabling ConshdlrIntegral, ConshdlrVarbound and ConshdlrSetppc. Many more constraint handlers exist.
As we perform branch-and-price, [SCIP] recommends to deactivate some presolver plugins, mainly PresolDualfix. We deactivated all presolvers, including PresolBoundshift, PresolImplics, PresolInttobinary, PresolProbing and PresolTrivial.

The branch-and-bound traversing mode is selected by NodeselBfs or NodeselDfs, whereby BFS needs a good node selection strategy as well as good branching strategy, which are the

[2] Constraints (5.2) and (5.14) in the SCF model 5.2, constraints (5.16) and (5.28) in MCF model 5.3.
[3] All the plugins described in the following are enabled by using the function SCIPincludeXXX, with XXX the plugin name. If plugins are not included such, they are not used by SCIP.

main bottleneck. DFS produces a much greater amount of BB nodes, but in our tests no significant run time difference were evaluated. Branch-and-bound nodes are selected automatically by the plugins `NodeselEstimate`, `NodeselHybridestim` and `NodeselRestartdfs`. Provided branch rules are `BranchruleAllfullstrong`, `BranchruleFullstrong`, `BranchruleInference`, `BranchruleMostinf`, `BranchruleLeastinf`, `BranchrulePscost`, `BranchruleRandom` and `BranchruleRelpscost`, which heavily impact on the running time. The document [Ach07] describes these plugins and their working principle. We let `SCIP` determine by its own an appropriate branching rule.

We do not separate any constraints, in this case [SCIP] recommends to not include any separator plugins. Furthermore, many heuristics plugins are at disposal, which we do not list here, since they were not included.

8.2.2 Module Structure

Figures 8.2 and 8.1 outline all implemented modules and how they work together. The remaining framework is described in [Dietzel08].

Figure 8.1 shows all classes relevant to segmentation and traversing. Basically we distinguish a static segmentation, implemented in `Segmentation`, and a dynamic segmentation, implemented in `SegmentationDyn`.

The static algorithms presented in chapters 6 and 7 have been implemented as functions in the class `SegTree`, which manages nodes of the type `SegNode`. Basically, each node contains the actual upper and a lower bound set, the splitting coordinate, the split discriminator and a flag if the actual node contains a null subspace. The tree may may be traversed by `TraverseBFS`, UB-`Driven Traversing` or `AdvBoundingTraversal`, when determining T^c. All such traversing algorithms implement an interface `Traverser`. The τ^* is determined by `PricingTraverser` (algorithm 6.4). The according `StaticTreePricer` from figure 8.2 uses this class in combination with `SegTree`.

The dynamic version is realized by the classes `DynTree` and `DynNode`. Here the traversing and generation of new nodes is done by `PreprocExpander` (for getting T^c) and `PricingExpander`, all implementing the abstract class `Expander`. The according `DynamicTreePricer` from figure 8.2 uses `PricingExpander`, which searches in a `DynTree`.

The classes `SimulatePricingTraverser` and `SimulPricingExpander` simulate the pricing problem based on random values as described in sections 6.4.5 and 6.5.2. The *Comparer* determines if single template arcs or entire sets T^c equal, by comparing coordinates, expressed node differences and sums of dual/random values for equality. The remaining classes are containers for T^c, result set of template arcs and node differences and implement some conversion functionality, as well es template arc determination. Further classes draw 2-dimensional segmentation trees either as bitmap (BMP) or scalable vector graphic (SVG).

Figure 8.2 shows the classes used to realize branch-and-price. The class `BranchPrice` loads the appropriate model, each one implementing the abstract class `Model`. Both, the SCF and MCF model may be solved entirely with `CompleteSCFModel` and `CompleteMCFModel`. In the classes `SCFModel` and `MCFModel` we load only an RMP, including either a starting solution or Farkas priced values, and expand the RMP by performing branch-and-price. In the SCF case, the `SCFPricer` prices a new variable by using either `StaticTreePricer`, `DynamicTreePricer` or the testing pricer `TcPricer`. Pricing for MCF works accordingly.

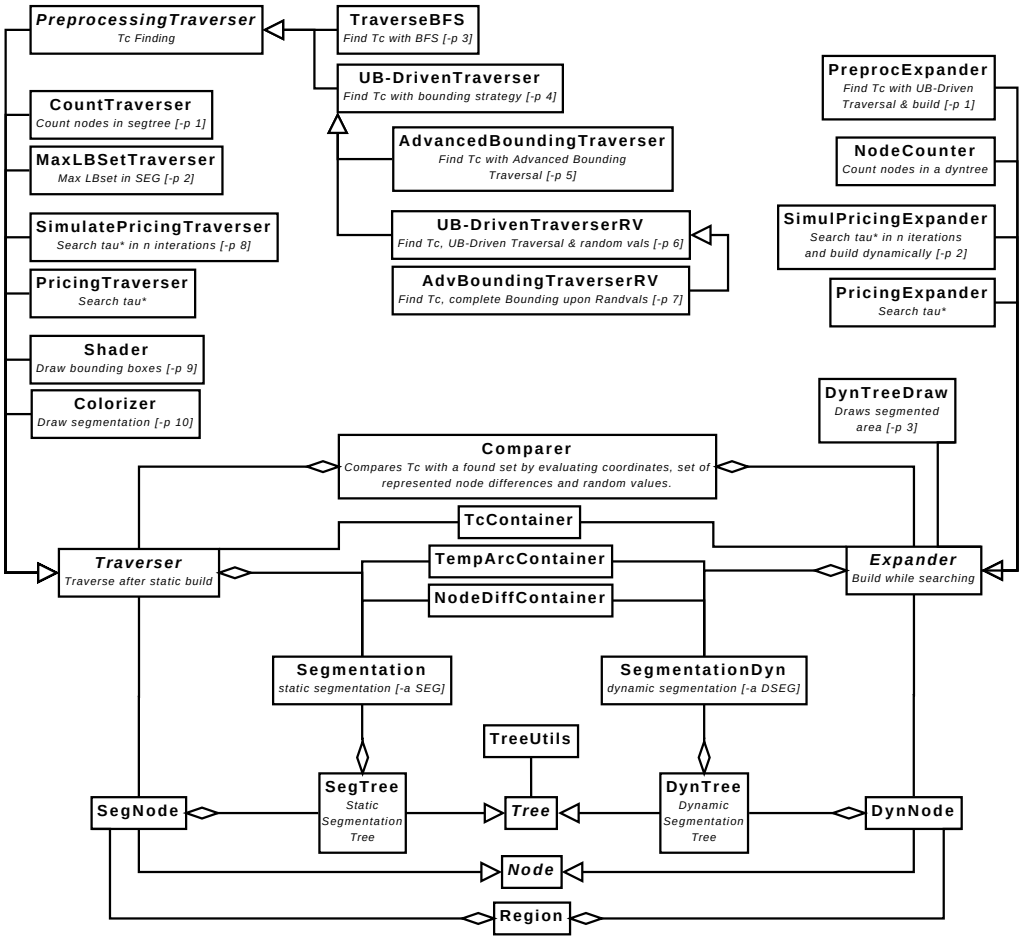


Figure 8.1: Class diagram for the parts concerning segmentation and traversing.

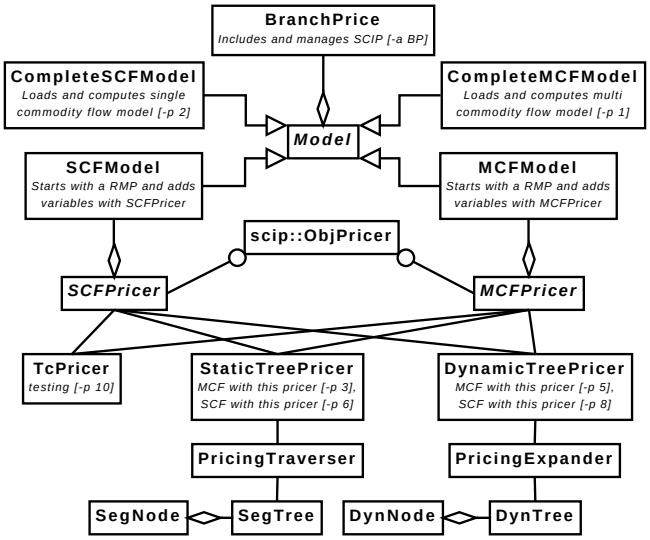


Figure 8.2: Class diagram for the parts concerning branch-and-price.

Chapter 9

Experimental Results

In this chapter we present the computational results for the algorithms presented in the previous chapters. Here, input data for the computational experiments is described. First, we evaluate run times and visited nodes for the algorithms that replace the preprocessing by [ChwRai09]. Then, computational results for the algorithms that solve the pricing problem are presented. The SCF and MCF formulation are compared, as well as pricer alternatives (static versus dynamic). Additionally, we evaluate how the run times differ when using node-label constraints and when not using them, as well as runs employing starting solutions versus Farkas pricer runs. All test runs have been executed on the following machines running `Linux`:

- `G1`: Grid of Dual Core AMD$^{\text{TM}}$Opteron$^{\text{TM}}$270, 1.993 GHz processor, 8 GB RAM, amd64.
- `L1`: Intel$^{\text{TM}}$Pentium$^{\text{TM}}$M processor 1.73GHz, 1 GB RAM, ix86.
- `O1`: AMD$^{\text{TM}}$Opteron$^{\text{TM}}$2.4 GHz processor, 4 GB RAM, amd64.

9.1 Input Data Files

Test data are 20 files provided by the *Fraunhofer Institute* [Fraun] in binary format ('`fhg`'), and selected 15 files from the *U.S. National Institute of Standards and Technology (NIST)* data set [NIST]. Table 9.1 lists files and amount of contained minutiae data points $|V|$, as well as the number of implied node differences is $|B|$. We name the Fraunhofer test set 'fhg', the second test set 'nist'.

All fhg and nist data is 4-dimensional and have varying numbers of minutiae data points. The files are multiple scans of four different fingers of two persons, indicated in the name by `P` (person), `F` (finger), `R` (release). The used forensic algorithm extracted a different number of minutiae in the different scans. In the Fraunhofer templates the minimum number of data points is $\min(|V|) = 15$, the maximum $\max(|V|) = 40$. These sets have an average number of $\text{avg}(|V|) = 30.75$ data points. Here, the domain has an average of $\tilde{v}_{\text{avg}} = (286, 383, 358, 2)^\top$, the smallest domain $\tilde{v}_{\min} = (129, 191, 252, 2)^\top$ and the maximal $\tilde{v}_{\max} = (224, 287, 312, 2)^\top$.
The NIST templates data points are a subset of a large test set. We selected 5 instances from the templates classified as *good*, *bad* and *ugly*. The NIST data points range from $\min(|V|) = 72$ to $\max(|V|) = 120$, with an average of $\text{avg}(|V|) = 96.47$ data points. The average domain is $\tilde{v}_{\text{avg}} = (3993, 3368, 359, 2)^\top$, the minimal domain is $\tilde{v}_{\min} = (2936, 2281, 359, 2)^\top$, the maximum $\tilde{v}_{\max} = (3293, 2788, 353, 2)^\top$.

Table 9.2 shows for which of the selected $\tilde{\delta}$ the set T^c is already available by the application of the preprocessing method from [ChwRai09]. We used these files to verify our segmentation tree results and for simulating the pricing problem. We point out that when *counting* nodes in a big segmentation tree (2–10 millions of nodes) the run time is always very small and the runs take 0.02–3.00 seconds. When running the actual tests, deviations of some milliseconds occur,

Table 9.1: Fraunhofer Institute and NIST sample minutiae data files. Column '$|V|$' shows the number of contained minutiae data points, '$|B|$' the number of resulting node differences.

short name	file name	file type	$\|V\|$	$\|B\|$
ft-01	P0001_F00_R00_L01_S00_C.fpt	fhg	31	930
ft-02	P0001_F00_R01_L01_S00_C.fpt	fhg	28	756
ft-03	P0001_F00_R02_L01_S00_C.fpt	fhg	35	1190
ft-04	P0001_F00_R03_L01_S00_C.fpt	fhg	20	380
ft-05	P0001_F00_R04_L01_S00_C.fpt	fhg	39	1482
ft-06	P0001_F01_R00_L01_S00_C.fpt	fhg	15	210
ft-07	P0001_F01_R01_L01_S00_C.fpt	fhg	28	756
ft-08	P0001_F01_R02_L01_S00_C.fpt	fhg	27	702
ft-09	P0001_F01_R03_L01_S00_C.fpt	fhg	27	702
ft-10	P0001_F01_R04_L01_S00_C.fpt	fhg	31	930
ft-11	P0001_F03_R00_L01_S00_C.fpt	fhg	28	1406
ft-12	P0001_F03_R01_L01_S00_C.fpt	fhg	38	756
ft-13	P0001_F03_R02_L01_S00_C.fpt	fhg	25	600
ft-14	P0001_F03_R03_L01_S00_C.fpt	fhg	33	1056
ft-15	P0001_F03_R04_L01_S00_C.fpt	fhg	29	812
ft-16	P0014_F00_R00_L01_S00_C.fpt	fhg	37	1332
ft-17	P0014_F00_R01_L01_S00_C.fpt	fhg	31	930
ft-18	P0014_F00_R02_L01_S00_C.fpt	fhg	40	1560
ft-19	P0014_F00_R03_L01_S00_C.fpt	fhg	35	1190
ft-20	P0014_F00_R04_L01_S00_C.fpt	fhg	28	756
nist-g-01	g001t2i.txt	txt	99	9702
nist-g-02	g002t3i.txt	txt	101	10100
nist-g-03	g003t8i.txt	txt	102	10302
nist-g-04	g004t8i.txt	txt	120	14280
nist-g-05	g005t8i.txt	txt	80	6320
nist-b-01	b101t9i.txt	txt	106	11130
nist-b-02	b102t0i.txt	txt	94	8742
nist-b-03	b104t8i.txt	txt	107	11342
nist-b-04	b105t2i.txt	txt	81	6480
nist-b-05	b106t8i.txt	txt	93	8556
nist-u-01	u201t6i.txt	txt	99	9702
nist-u-02	u202t8i.txt	txt	93	8556
nist-u-03	u204t2i.txt	txt	100	9900
nist-u-04	u205t4i.txt	txt	84	6972
nist-u-05	u206t3i.txt	txt	73	5256

in very big instances up to some seconds. This is due to memory fetching, swapping and other computational factors. We selected following delta values for tests:

- 2D: $\tilde{\delta} = \big((10,10)^\top, (20,20)^\top, (30,30)^\top, (40,40)^\top, (80,80)^\top, (120,120)^\top\big)$.
- 3D: $\tilde{\delta} = \big((10,10,10)^\top, (20,20,20)^\top, (30,30,30)^\top, (40,40,40)^\top, (80,80,80)^\top\big)$.

Table 9.2: Precomputed files, containing candidate template arcs (T^c) determined by the preprocessing from [ChwRai09].

2D $\tilde{\delta}$	$(10,10)$	$(20,20)$	$(30,30)$	$(40,40)$	$(80,80)$	$(120,120)$
fhg files	√	√	√	√		
nist files		√		√	√	√
3D $\tilde{\delta}$	$(10,10,10)$	$(20,20,20)$		$(40,40,40)$	$(80,80,80)$	
fhg files	√	√		√		
nist files				√	√	

9.2 Static and Dynamic Segmentation

The verification of the correctness of the segmentation tree has been done by performing the following steps:

- We compared the running times of the static segmentation build procedure, presented in section 6.4.1 and 6.4.2 and tested its correctness by extracting T^c, which then was compared to the set T^c from the preprocessing by [ChwRai09].
- We compared the running times of the strategies that determine T^c from chapter 7, namely UB-driven traversing (section 7.2.2), advanced bounding traversal (section 7.2.3), and best first search (section 7.2.1), in order to determine the quickest way for getting T^c. Also, we examined the overall number of created nodes as well as the percentage of visited nodes for each traversing strategy. The conclusion is that UB-driven traversing performs best.
- We checked the dynamic segmentation, presented in 6.5, for correctness by extracting again T^c with the algorithm UB-driven traversing for dynamic trees, namely `DynamicSearch-`T^c (section 7.3.1), and compared the set to the set T^c extracted by the previous preprocessing.
- We simulated the pricing problem for both the static and dynamic variant in order to determine, if both strategies to find τ^*, presented in section 6.4.3 and 6.5, find the correct τ^* in each simulated pricing iteration, and if they are quick enough for being used in the pricer routine.

This data is summarized in the following tables. The tests have been run on grid `G1`. Tables 9.3, 9.4 and 9.5, 9.6 show the run times of the static and dynamic segmentation in comparison to the run times of the preprocessing from [ChwRai09], once for 2-dimensional and once for 3-dimensional parameters $\tilde{\delta}$. For each $\tilde{\delta}$ the number of found candidate template arcs is listed (column '$|T^c|$'), followed by static tree building time in seconds (column 'b[s]'). As during the tests UB-driven traversing was found to be the quickest variant for determining T^c, in these tables we compare its traversing time and total time for building and traversing with the dynamic version total run time. The super-column 'static/UBD' lists static segmentation tree data, and column 'dyn' dynamic segmentation tree data. The columns 't[s]' list run times in seconds needed for traversing the static tree with UBD, the columns 'tot[s]' list total run time in seconds for determining T^c with UBD. The last column 'PP' in each $\tilde{\delta}$ block is the run time in seconds of the preprocessing by [ChwRai09]. Note that these run times have been determined on machine `O1` and we used them, since a new series of tests would take too long.

Tables 1, 2 (2-dimensional $\tilde{\delta}$) and 3, 4 (3-dimensional $\tilde{\delta}$) in the appendix list the run times of all three strategies that find T^c as well as the amount of visited nodes in the static tree, or created nodes for the dynamic tree. Examined strategies are:

- UB-driven traversing for determining T^c in a static tree. Column 'UBD' lists run times.
- Advanced bounding traversal for determining T^c in a static tree. Column 'ABT'.
- Best first search for determining T^c in a static tree. Column 'BFS' lists run times.

These tests have been run on grid `G1`. We compare this data again to the dynamic segmentation. The respective columns are again entitled 'static' for the static tree and 'dyn' for the dynamic tree. For 'static', the column '$|n|$' lists the total amount of nodes contained in the static tree. This tree size is later used as basis for the computation of the percentage of visited nodes. The columns 'UBD', 'ABT' and 'BFS' list the run times of each traversing strategy in seconds. For each traversing strategy the respective amount of visited nodes is listed in the columns called '%n'. For the dynamic version 'dyn' we listed in column '$|n|$' the overall amount of produced nodes when searching for T^c, and how they correlate with the number of nodes in a static tree. The column '%n' indicates the percentage, which the dynamic tree is smaller than the static tree. The amount of nodes in the dynamic tree is the smallest number of nodes needed for finding T^c. When searching only for τ^* in the dynamic tree, the number of nodes decreases massively, especially when emptying the tree before each pricing iteration. In contrast, when searching τ^* in the static tree, the number of nodes remains equal, since this tree is always built completely. In the following we aggregate the gained insights, subdivided into the main topics.

Table 9.3: Two dimensional $\tilde{\delta} = (10,10)^\top, (20,20)^\top, (30,30)^\top$: The building ('b[s]'), traversing ('t[s]') and total ('tot[s]') run times (on `G1`) of the best static tree traversing strategy UBD (UB-driven Traversing) compared to the total run times of the dynamic segmentation ('dyn/tot[s]'). The run times (on `01`) of the preprocessing from [ChwRai09] is listed in column 'PP'. Column '$|T^c|$' shows the extracted (and correct) number of candidate template arcs by all three strategies.

2d/fhg	$\tilde{\delta} = (10,10)^\top$						$\tilde{\delta} = (20,20)^\top$						$\tilde{\delta} = (30,30)^\top$					
		static/UBD			dyn	PP		static/UBD			dyn	PP		static/UBD			dyn	PP
file	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]
ft-01	797	0.07	0.05	0.12	0.13	7	1863	0.27	0.27	0.54	0.58	74	3747	0.78	0.87	1.65	2.17	526
ft-02	610	0.04	0.03	0.07	0.09	4	1443	0.18	0.17	0.35	0.37	30	2666	0.49	0.51	1.00	1.21	185
ft-03	1002	0.08	0.10	0.18	0.19	12	2684	0.35	0.59	0.94	1.07	146	4644	1.02	1.71	2.73	3.65	902
ft-04	296	0.02	0.01	0.03	0.02	1	510	0.06	0.04	0.10	0.10	4	1035	0.17	0.12	0.29	0.34	23
ft-05	1401	0.12	0.15	0.27	0.27	23	3398	0.52	0.71	1.23	1.42	297	7044	1.46	2.47	3.93	5.53	2052
ft-06	145	0.00	0.01	0.01	0.01	1	248	0.02	0.01	0.03	0.03	1	354	0.06	0.02	0.08	0.09	3
ft-07	517	0.04	0.03	0.07	0.08	3	1019	0.14	0.12	0.26	0.28	19	2035	0.39	0.43	0.82	0.89	102
ft-08	533	0.04	0.03	0.07	0.07	3	1030	0.13	0.11	0.24	0.24	19	1898	0.37	0.31	0.68	0.73	92
ft-09	506	0.03	0.03	0.06	0.07	3	1022	0.14	0.11	0.25	0.24	17	1828	0.35	0.30	0.65	0.73	94
ft-10	767	0.06	0.04	0.10	0.11	6	1721	0.22	0.25	0.47	0.50	46	3132	0.62	0.70	1.32	1.62	283
ft-11	1565	0.12	0.18	0.30	0.32	28	3751	0.57	0.97	1.54	1.65	377	7561	1.62	2.99	4.61	8.20	3261
ft-12	699	0.05	0.05	0.10	0.10	5	1548	0.22	0.22	0.44	0.50	51	3268	0.68	0.80	1.48	1.95	356
ft-13	498	0.04	0.03	0.07	0.06	3	1010	0.14	0.11	0.25	0.25	19	2201	0.38	0.38	0.76	0.98	122
ft-14	1002	0.08	0.09	0.17	0.16	10	2292	0.31	0.41	0.72	0.80	109	4458	0.97	1.28	2.25	2.93	991
ft-15	742	0.06	0.05	0.11	0.12	6	1768	0.25	0.26	0.51	0.57	63	3487	0.76	0.90	1.66	2.21	442
ft-16	1144	0.09	0.11	0.20	0.20	14	2501	0.36	0.54	0.90	1.00	145	4984	1.03	1.82	2.85	3.99	919
ft-17	800	0.05	0.05	0.10	0.10	5	1633	0.22	0.23	0.45	0.46	42	3317	0.62	0.85	1.47	1.69	301
ft-18	1288	0.10	0.15	0.25	0.26	22	2858	0.43	0.66	1.09	1.20	238	5632	1.23	2.33	3.56	4.38	1577
ft-19	1017	0.08	0.09	0.17	0.17	10	2161	0.30	0.38	0.68	0.73	87	3857	0.81	1.11	1.92	2.21	531
ft-20	622	0.04	0.02	0.06	0.09	3	1086	0.14	0.13	0.27	0.30	20	1649	0.37	0.31	0.68	0.78	107
AVG	**797**	**0.1**	**0.1**	**0.1**	**0.1**	**8.5**	**1777**	**0.2**	**0.3**	**0.6**	**0.6**	**90.2**	**3440**	**0.7**	**1.0**	**1.7**	**2.3**	**643.5**
stdev		0.0	0.1	0.1	0.1	7.8		0.1	0.3	0.4	0.4	103.8		0.4	0.8	1.3	2.0	821.8

2d/nist	$\tilde{\delta} = (10,10)^\top$						$\tilde{\delta} = (20,20)^\top$						$\tilde{\delta} = (30,30)^\top$					
		static/UBD			dyn	PP		static/UBD			dyn	PP		static/UBD			dyn	PP
file	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]
nist-g-01	7944	0.29	1.88	2.17	2.46	-	6418	0.53	2.67	3.20	4.03	-	7992	0.89	4.70	5.59	5.75	-
nist-g-02	8514	0.24	1.93	2.17	2.63	-	6787	0.41	2.40	2.81	4.11	-	7811	0.71	3.99	4.70	5.46	-
nist-g-03	7853	0.25	1.99	2.24	2.70	-	7151	0.55	3.18	3.73	4.17	-	9869	1.06	6.76	7.82	9.74	-
nist-g-04	11080	0.41	4.00	4.41	5.29	-	9938	0.86	6.47	7.33	8.43	-	13216	1.50	13.04	14.54	21.54	-
nist-g-05	5460	0.16	0.78	0.94	1.09	-	4309	0.25	0.99	1.24	1.31	-	4627	0.42	1.58	2.00	2.01	-
nist-b-01	8758	0.31	2.45	2.76	3.19	-	7686	0.64	3.84	4.48	6.84	-	10255	1.12	7.70	8.82	10.77	-
nist-b-02	6648	0.24	1.51	1.75	2.02	-	6401	0.49	2.71	3.20	3.32	-	9055	0.96	5.71	6.67	7.29	-
nist-b-03	8880	0.32	2.53	2.85	3.25	-	8023	0.76	4.14	4.90	4.94	-	10706	1.18	8.13	9.31	11.35	-
nist-b-04	5618	0.15	0.82	0.97	1.13	-	4360	0.32	1.06	1.38	1.41	-	4696	0.44	1.64	2.08	2.12	-
nist-b-05	6974	0.23	1.44	1.67	1.96	-	5938	0.46	2.19	2.65	3.27	-	7376	0.80	4.04	4.84	5.11	-
nist-u-01	8112	0.24	1.81	2.05	2.46	-	6560	0.46	2.40	2.86	3.86	-	8058	0.77	4.39	5.16	5.79	-
nist-u-02	7024	0.22	1.42	1.64	1.92	-	5858	0.44	2.04	2.48	2.64	-	7248	0.74	3.71	4.45	4.73	-
nist-u-03	7526	0.27	1.92	2.19	2.51	-	7102	0.62	3.31	3.93	5.18	-	9766	1.06	6.76	7.82	9.02	-
nist-u-04	5974	0.17	0.96	1.13	1.33	-	4616	0.32	1.24	1.56	1.7	-	5124	0.50	1.86	2.36	2.44	-
nist-u-05	4457	0.10	0.52	0.62	0.75	-	3501	0.25	0.65	0.90	0.96	-	3782	0.35	1.05	1.40	1.39	-
AVG	**7388**	**0.2**	**1.7**	**2.0**	**2.3**		**6309**	**0.5**	2.6	**3.1**	**3.7**		**7972**	**0.8**	**5.0**	**5.8**	**7.0**	
stdev		0.1	0.9	0.9	1.1			0.2	1.5	1.7	2.1			0.3	3.2	3.5	5.1	

Table 9.4: Two dimensional $\tilde{\delta} = (40, 40)^\top, (80, 80)^\top, (120, 120)^\top$: The building ('b[s]'), traversing ('t[s]') and total ('tot[s]') run times (on `G1`) of the best static tree traversing strategy UBD (UB-Driven Traversing) compared to the total run times of the dynamic segmentation ('dyn/tot[s]'). The run times (on `O1`) of the preprocessing from [ChwRai09] is listed in column 'PP'. Column '$|T^c|$' shows the extracted (and correct) number of candidate template arcs by all three strategies.

2d/fhg	$\tilde{\delta} = (40, 40)^\top$						$\tilde{\delta} = (80, 80)^\top$					
		static/UBD			dyn	PP		static/UBD			dyn	PP
file	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]
ft-01	5802	1.75	1.88	3.63	6.27	2810	16873	11.81	21.80	33.61	124.33	-
ft-02	4374	1.07	1.28	2.35	3.66	911	11954	8.99	13.54	22.53	91.75	-
ft-03	7786	2.27	4.36	6.63	10.95	4799	22686	16.83	68.17	85.00	271.14	-
ft-04	31582	0.36	0.26	0.62	0.91	101	4045	3.01	2.32	5.33	14.68	-
ft-05	11276	3.25	6.20	9.45	20.71	12144	29516	22.78	103.74	126.52	419.99	-
ft-06	574	0.12	0.06	0.18	0.24	11	$\tilde{\delta} > \tilde{v}/2$					
ft-07	3280	0.88	1.12	2.00	2.57	426	11186	7.98	18.04	26.02	59.13	-
ft-08	3402	0.90	0.87	1.77	2.41	463	11452	7.92	11.06	18.98	67.54	-
ft-09	3429	0.80	0.84	1.64	2.22	382	11404	6.84	10.98	17.82	55.95	-
ft-10	5365	1.44	1.94	3.38	4.78	1334	18162	11.58	37.96	49.54	151.41	-
ft-11	11542	3.74	7.07	10.81	23.52	15580	32418	24.90	123.44	148.34	567.63	-
ft-12	5064	1.54	1.86	3.40	5.47	1836	16012	11.29	30.88	42.17	169.66	-
ft-13	3980	0.88	1.03	1.91	3.01	617	10950	7.38	8.64	16.02	71.05	-
ft-14	7130	2.23	3.12	5.35	8.21	3468	23253	15.81	64.27	80.08	243.10	-
ft-15	5720	1.72	2.22	3.94	6.33	2549	17756	12.36	38.17	50.53	171.39	-
ft-16	7330	2.43	3.85	6.28	9.81	4431	26221	18.00	91.64	109.64	336.23	-
ft-17	5585	1.38	2.10	3.48	4.87	1365	18549	11.73	36.85	48.58	177.10	-
ft-18	8950	2.75	5.22	7.97	13.21	6106	29762	21.37	122.68	144.05	396.29	-
ft-19	6247	1.86	2.70	4.56	6.50	2515	19188	14.39	47.32	61.71	190.01	-
ft-20	3022	0.77	0.80	1.57	2.21	395	9417	7.19	7.83	15.02	53.35	-
AVG	**7072**	**1.6**	**2.4**	**4.0**	**6.9**	**3112.2**	**17937**	**12.7**	**45.2**	**58.0**	**191.1**	
stdev		1.0	2.0	2.9	6.2	4091.9		5.9	39.6	45.4	148.9	

2d/nist	$\tilde{\delta} = (40, 40)^\top$						$\tilde{\delta} = (80, 80)^\top$					
		static/UBD			dyn	PP		static/UBD			dyn	PP
file	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]
nist-g-01	10386	1.46	8.31	9.77	23.90	-	28375	8.21	79.34	87.55	234.06	-
nist-g-02	9845	1.15	6.92	8.07	21.64	-	26316	6.43	57.49	63.92	262.14	-
nist-g-03	13804	1.84	12.62	14.46	31.37	-	36329	12.75	185.15	197.90	523.29	-
nist-g-04	18215	2.60	24.37	26.97	66.01	-	49874	16.92	488.56	505.48	1012.72	-
nist-g-05	5834	0.66	2.61	3.27	3.24	-	14158	3.34	14.30	17.64	38.31	-
nist-b-01	13674	1.97	14.18	16.15	30.05	-	37124	12.62	225.14	237.76	705.72	-
nist-b-02	12517	1.72	10.94	12.66	20.61	-	33910	12.23	173.32	185.55	347.06	-
nist-b-03	14764	2.08	14.90	16.98	34.22	-	39073	14.14	237.95	252.09	481.17	-
nist-b-04	5895	0.71	2.73	3.44	4.05	-	14243	3.42	14.67	18.09	68.22	-
nist-b-05	10040	1.49	7.59	9.08	22.07	-	27118	8.49	96.94	105.43	297.27	-
nist-u-01	10310	1.42	7.79	9.21	12.22	-	25990	7.13	87.67	94.80	201.67	-
nist-u-02	9667	1.29	6.89	8.18	10.54	-	26398	7.69	78.18	85.87	283.43	-
nist-u-03	13656	2.08	13.16	15.24	49.12	-	37326	14.78	333.14	347.92	466.92	-
nist-u-04	6405	0.82	3.15	3.97	5.25	-	16098	4.46	31.59	36.05	103.35	-
nist-u-05	4692	0.58	1.71	2.29	2.49	-	11075	2.98	10.19	13.17	19.66	-
AVG	**10646**	**1.5**	**9.2**	**10.6**	**22.5**		**28227**	**9.0**	**140.9**	**149.9**	**336.3**	
stdev		0.6	6.0	6.6	18.0			4.6	135.7	140.0	270.0	

2d/nist	$\tilde{\delta} = (120, 120)^\top$					
		static/UBD			dyn	PP
file	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]
nist-g-01	52880	30.25	601.33	631.58	1177.04	-
nist-g-02	50035	23.41	465.29	488.70	1316.57	-
nist-g-03	65499	50.06	1036.14	1086.20	1615.35	-
nist-g-04	90508	memory overflow			2816.71	-
nist-g-05	26763	11.28	98.28	109.56	396.21	-
nist-b-01	68219	46.90	1198.02	1244.92	1867.14	-
nist-b-02	58296	48.09	966.37	1014.46	1253.60	-
nist-b-03	71028	84.30*	1330.78*	1415.08*	1710.77	-
nist-b-04	27264	11.89	179.48	191.37	289.93	-
nist-b-05	50953	32.93	757.66	790.59	1209.40	-
nist-u-01	49754	28.04	578.95	606.99	1017.48	-
nist-b-02	50033	29.34	778.36	807.70	1173.88	-
nist-b-03	65888	86.34*	1736.33*	1822.67*	1604.14	-
nist-b-04	30482	14.62	190.28	204.90	428.79	-
nist-b-05	20791	9.56	57.79	67.35	250.44	-
AVG	**51892**	**28.0**	**575.7**	**603.7**	**1208.5**	
stdev	14.6	386.2	400.5	690.6		

* indicates data determined on `O1` instead of `G1`. This machine has more memory at disposition. The average 'AVG' does not include `O1` data.

Table 9.5: Three dimensional $\tilde{\delta} = (10, 10, 10)^\top, (20, 20, 20)^\top, (30, 30, 30)^\top$: The building ('b[s]'), traversing ('t[s]') and total ('tot[s]') run times (on `G1`) of the best static tree traversing strategy UBD (UB-Driven Traversing) compared to the total run times of the dynamic segmentation ('dyn/tot[s]'). The run times (on `01`) of the preprocessing from [ChwRai09] are listed in column 'PP'. Column '$|T^c|$' shows the extracted (and correct) number of candidate template arcs by all three strategies.

3d/fhg	$\tilde{\delta} = (10, 10, 10)^\top$						$\tilde{\delta} = (20, 20, 20)^\top$						$\tilde{\delta} = (30, 30, 30)^\top$					
		static/UBD		dyn	PP		static/UBD		dyn	PP		static/UBD		dyn	PP		static/UBD		dyn	PP
file	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]
ft-01	826	0.05	0.02	0.07	3	860	0.24	0.13	0.37	0.34	6	1693	0.91	0.90	1.81	1.73	34			
ft-02	664	0.05	0.02	0.07	2	715	0.22	0.10	0.32	0.28	4	1560	0.82	0.83	1.65	2.49	23			
ft-03	1042	0.08	0.05	0.13	5	1152	0.38	0.25	0.63	0.56	11	2620	1.55	2.00	3.55	3.29	82			
ft-04	356	0.02	0.01	0.03	1	352	0.10	0.04	0.14	0.12	1	695	0.46	0.23	0.69	1.03	8			
ft-05	1246	0.11	0.07	0.18	8	1486	0.47	0.41	0.88	1.36	20	3546	2.23	3.90	6.13	5.82	237			
ft-06	202	0.01	0.00	0.01	1	156	0.03	0.00	0.03	0.05	1	231	0.08	0.02	0.10	0.12	1			
ft-07	680	0.04	0.02	0.06	2	614	0.16	0.07	0.23	0.32	3	1129	0.54	0.41	0.95	0.78	11			
ft-08	630	0.03	0.02	0.05	2	570	0.13	0.05	0.18	0.19	3	1064	0.49	0.30	0.79	1.36	10			
ft-09	626	0.04	0.02	0.06	2	578	0.16	0.07	0.23	0.30	3	1061	0.51	0.30	0.81	0.72	11			
ft-10	812	0.06	0.03	0.09	3	812	0.21	0.12	0.33	0.51	5	1548	0.82	0.65	1.47	2.18	28			
ft-11	1168	0.10	0.07	0.17	7	1677	0.69	0.60	1.29	1.78	25	4446	3.08	5.71	8.79	8.44	305			
ft-12	632	0.06	0.02	0.08	2	752	0.24	0.13	0.37	0.57	5	1718	1.05	0.85	1.90	3.42	40			
ft-13	517	0.04	0.01	0.05	1	531	0.16	0.06	0.22	0.35	3	1143	0.61	0.37	0.98	1.51	15			
ft-14	842	0.08	0.04	0.12	4	1024	0.32	0.19	0.51	0.83	8	2286	1.25	1.36	2.61	3.00	58			
ft-15	656	0.06	0.02	0.08	2	786	0.25	0.12	0.37	0.35	5	1676	0.94	0.78	1.72	2.77	30			
ft-16	1142	0.09	0.06	0.15	6	1457	0.45	0.37	0.82	1.27	21	3125	1.91	2.73	4.64	9.26	190			
ft-17	814	0.06	0.02	0.08	3	878	0.24	0.13	0.37	0.57	6	1878	0.91	0.93	1.84	1.80	31			
ft-18	1298	0.13	0.08	0.21	10	1710	0.65	0.58	1.23	1.21	31	3963	2.94	4.34	7.28	15.78	270			
ft-19	1000	0.09	0.05	0.14	5	992	0.36	0.20	0.56	0.81	10	1944	1.17	1.04	2.21	3.50	49			
ft-20	658	0.04	0.02	0.06	2	672	0.14	0.07	0.21	0.25	4	990	0.55	0.33	0.88	0.96	20			
AVG	**791**	**0.1**	**0.0**	**0.1**	**3.6**	**889**	**0.3**	**0.2**	**0.5**	**0.6**	**8.8**	**1916**	**1.1**	**1.4**	**2.5**	**3.5**	**72.7**			
stdev		0.0	0.0	0.1	2.5		0.2	0.2	0.3	0.5	8.6		0.8	1.6	2.4	3.8	95.2			

3d/nist	$\tilde{\delta} = (10, 10, 10)^\top$						$\tilde{\delta} = (20, 20, 20)^\top$						$\tilde{\delta} = (30, 30, 30)^\top$					
		static/UBD		dyn	PP		static/UBD		dyn	PP		static/UBD		dyn	PP		static/UBD		dyn	PP
file	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]
nist-g-01	9538	0.40	1.93	2.33	2.72	-	8336	0.97	2.68	3.65	3.70	-	7466	2.02	5.58	7.60	6.75	-		
nist-g-02	10002	0.34	2.05	2.39	2.97	-	9062	0.66	2.34	3.00	3.72	-	7866	1.19	3.61	4.80	6.22	-		
nist-g-03	10053	0.42	2.16	2.58	3.07	-	8745	0.94	2.86	3.80	4.34	-	7993	2.02	6.36	8.38	8.87	-		
nist-g-04	13956	0.62	4.21	4.83	6.67	-	12006	1.51	5.81	7.32	8.47	-	10852	2.87	11.89	14.76	16.01	-		
nist-g-05	6240	0.22	0.82	1.04	1.22	-	5772	0.41	0.95	1.36	1.51	-	5022	0.72	1.40	2.12	2.55	-		
nist-b-01	10880	0.47	2.54	3.01	4.03	-	9496	1.01	3.46	4.47	4.74	-	8399	2.06	6.90	8.96	8.48	-		
nist-b-02	8530	0.36	1.57	1.93	2.24	-	7260	0.80	2.22	3.02	3.18	-	6642	1.66	4.77	6.43	6.65	-		
nist-b-03	11120	0.50	2.65	3.15	3.66	-	9722	1.12	3.53	4.65	7.38	-	8634	2.29	7.53	9.82	10.49	-		
nist-b-04	6426	0.24	0.86	1.10	1.27	-	5954	0.46	1.05	1.51	1.56	-	5230	0.80	1.72	2.52	2.48	-		
nist-b-05	8448	0.33	1.50	1.83	2.46	-	7472	0.76	1.92	2.68	3.29	-	6508	1.51	3.82	5.33	5.58	-		
nist-u-01	9560	0.35	1.90	2.25	2.71	-	8732	0.63	2.13	2.76	3.35	-	7491	1.21	3.42	4.63	5.58	-		
nist-u-02	8432	0.32	1.47	1.79	2.37	-	7580	0.67	1.79	2.46	2.78	-	6538	1.32	3.22	4.54	5.01	-		
nist-u-03	9656	0.35	1.99	2.34	2.85	-	8358	0.86	2.62	3.48	3.98	-	7476	1.75	5.68	7.43	8.44	-		
nist-u-04	6866	0.24	0.99	1.23	1.44	-	6250	0.49	1.18	1.67	1.80	-	5408	0.92	1.86	2.78	2.76	-		
nist-u-05	5187	0.18	0.56	0.74	0.86	-	4829	0.33	0.64	0.97	1.08	-	4189	0.62	1.00	1.62	1.62	-		
AVG	**8993**	**0.4**	**1.8**	**2.2**	**2.7**		**7972**	**0.8**	**2.3**	**3.1**	**3.7**		**7048**	**1.5**	**4.6**	**6.1**	**6.5**			
stdev		0.1	0.9	1.0	1.4			0.3	1.3	1.6	2.1			0.6	2.9	3.5	3.7			

Table 9.6: Three dimensional $\tilde{\delta} = (40, 40, 40)^\top, (80, 80, 80)^\top$: The building ('b[s]'), traversing ('t[s]') and total ('tot[s]') run times (on `G1`) of the best static tree traversing strategy UBD (UB-Driven Traversing) compared to the total run times of the dynamic segmentation ('dyn/tot[s]'). The run times (on `01`) of the preprocessing from [ChwRai09] is listed in column 'PP'. Column '$|T^c|$' shows the extracted (and correct) number of candidate template arcs by all three strategies.

3d/fhg		$\tilde{\delta} = (40, 40, 40)^\top$				
		static/UBD			dyn	PP
file	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]
ft-01	3897	3.01	5.10	8.11	8.90	272
ft-02	3353	2.75	4.28	7.03	14.06	241
ft-03	6464	5.58	13.93	19.51	49.82	1019
ft-04	1223	1.31	0.93	2.24	2.69	55
ft-05	8669	8.04	26.91	34.95	41.04	2578
ft-06	439	0.21	0.08	0.29	0.40	2
ft-07	2237	1.74	2.10	3.84	3.54	63
ft-08	2164	1.60	1.55	3.15	5.95	57
ft-09	2091	1.50	1.45	2.95	3.02	58
ft-10	2930	2.58	3.06	5.64	5.36	165
ft-11	11497	12.04	39.93	51.97	86.04	3748
ft-12	4224	3.72	5.25	8.97	21.28	341
ft-13	2573	2.17	2.23	4.40	8.51	143
ft-14	5150	4.33	7.83	12.16	34.57	522
ft-15	3980	3.63	5.04	8.67	20.77	333
ft-16	6095	7.07	12.97	20.04	24.96	2858
ft-17	3847	3.06	5.10	8.16	16.61	273
ft-18	8889	9.87	22.27	32.14	122.60	3435
ft-19	4538	3.87	5.72	9.59	10.39	399
ft-20	2021	1.70	1.45	3.15	6.06	104
AVG	**4314**	**4.0**	**8.4**	**12.3**	**24.3**	**833.3**
stdev		3.1	10.3	13.3	31.1	1230.1

3d/nist		$\tilde{\delta} = (40, 40, 40)^\top$				
		static/UBD			dyn	PP
file	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]
nist-g-01	7996	3.84	13.19	17.03	27.50	-
nist-g-02	7820	2.44	7.12	9.56	17.45	-
nist-g-03	9618	3.93	15.08	19.01	36.57	-
nist-g-04	12657	5.74	29.14	34.88	60.34	-
nist-g-05	4764	1.34	2.62	3.96	4.55	-
nist-b-01	9559	4.13	16.33	20.46	39.83	-
nist-b-02	8278	3.90	12.82	16.72	26.15	-
nist-b-03	10142	4.31	16.92	21.23	21.53	-
nist-b-04	4986	1.49	3.31	4.80	4.47	-
nist-b-05	7150	2.92	8.34	11.26	16.47	-
nist-u-01	7339	2.21	6.59	8.80	15.72	-
nist-u-02	6924	2.48	7.09	9.57	13.28	-
nist-u-03	9260	3.99	15.14	19.13	40.32	-
nist-u-04	5238	1.69	3.63	5.32	5.27	-
nist-u-05	4002	1.20	1.93	3.13	2.96	-
AVG	**7715**	**3.0**	**10.6**	**13.7**	**22.2**	
stdev		1.3	7.3	8.7	16.5	

3d/nist		$\tilde{\delta} = (80, 80, 80)^\top$				
		static/UBD			dyn	PP
file	$\|T^c\|$	b[s]	t[s]	tot[s]	tot[s]	tot[s]
nist-g-01	26776	memory overflow			876.04	-
nist-g-02	25548	18.62	154.3	172.92	692.06	-
nist-g-03	42035	memory overflow			2780.34	-
nist-g-04	52759	memory overflow			3267.13	-
nist-g-05	13130	9.36	47.6	56.96	193.03	-
nist-b-01	39870	memory overflow			1832.18	-
nist-b-02	38826	memory overflow			1648.16	-
nist-b-03	42600	memory overflow			2258.74	-
nist-b-04	13611	10.39	60.62	71.01	206.71	-
nist-b-05	27287	26.51	261.88	288.39	991.52	-
nist-u-01	25032	18.92	136.9	155.82	789.52	-
nist-u-02	25560	22.59	190.51	213.1	883.76	-
nist-u-03	44834	memory overflow			2642.19	-
nist-u-04	14490	12.08	64.44	76.52	216.44	-
nist-u-05	10565	7.86	30.28	38.14	83.42	-
AVG	**29528**	**15.8**	**118.3**	**134.1**	**1290.7**	
stdev		6.8	81.5	88.2	1046.7	

9.2.1 Implications for the Pricing Algorithms

As tables 9.3, 9.4 and 9.5, 9.6 show in column 'b[s]', building the static segmentation tree takes a small amount of time. The average run times increase moderately with the size of $\tilde{\delta}$. As we can see, the building needs an average time less than 1 second for small and medium $\tilde{\delta}$. Big $\tilde{\delta}$, especially for the NIST files (which have bigger domain borders $\tilde{v}$ and as double data points as the fhg files) up to 10–28 seconds. Such building times are acceptable to be basis for a static pricing algorithm. Figure 9.1 shows the growth rate of the average run times for the static tree with 2- and 3-dimensional $\tilde{\delta}$ parameters.
Generally, also the run times of the dynamic tree are well suited for being used as basis for a dynamic pricing algorithm. Since here we do not build the entire segmentation tree in advance, but only the parts needed for the solution of the pricing problem, a significantly less amount of time is used for building and searching in each pricing iteration.

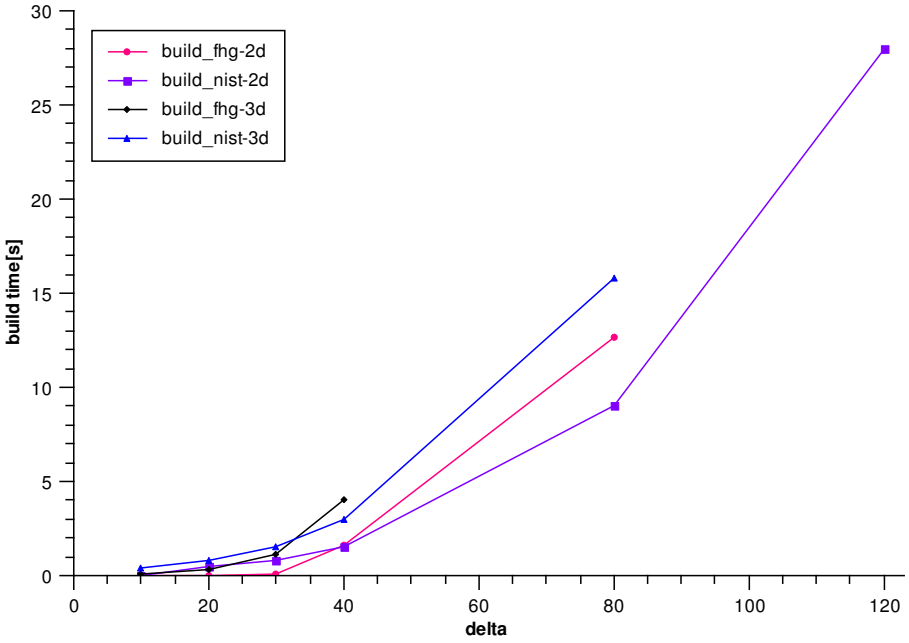


Figure 9.1: Growth rates of static tree building time, distinguished by dimensionality of $\tilde{\delta}$ as well as by data set (fhg and nist).

9.2.2 Implications for the Determination of T^c

Tables 9.3, 9.4 and 9.5, 9.6 compare the strategies for determining T^c. The tables list data for the static segmentation tree (column 'static/tot[s]') and for the dynamic segmentation tree (column 'dyn/tot[s]'). For this evaluation, we selected the best static approach, namely UB-driven traversing. Both approaches, static and dynamic, determined the correct set T^c. When we compare the total (for static tree build and traverse, columns 'b[s]', 't[s]') run times in seconds, for small $\tilde{\delta}$ the run times for the dynamic tree are nearly equal to the run times for the static tree. The dynamic approach is outperformed, the bigger $\tilde{\delta}$ gets. Again, the total run time increases with the size of $\tilde{\delta}$. The total time needed for building and traversing is decisively better than for the preprocessing from [ChwRai09] and both approaches clearly outperform the preprocessing (column 'PP') and replace it efficiently.

We examined the strategies more closely in tables 1, 2, 3 and 4, listed in the appendix. For all three tested strategies UBD, ABT and BFS, the respective column '%n' lists the percentage of

visited nodes. The columns '$|n|$' show that the static approach produces a much greater number of nodes than the dynamic approach. This is due to the dynamic tree expands only branches determined to be relevant. This ratio was embedded in the tables in columns 'dyn/%n' which lists the percentage of nodes created in the dynamic tree with respect to the nodes in the static tree. Figure 9.2 shows average percentages of visited nodes for UBD, ABT and BFS. As we can see, UBD visits approximately 10% more nodes than ABT, but its average run times are significantly slower. Further, as we can see, ABT comes close to the optimal path through the static tree. This optimal path was determined by visiting with best first search only branches that contain elements of T^c, for each element of T^c only one branch is visited.

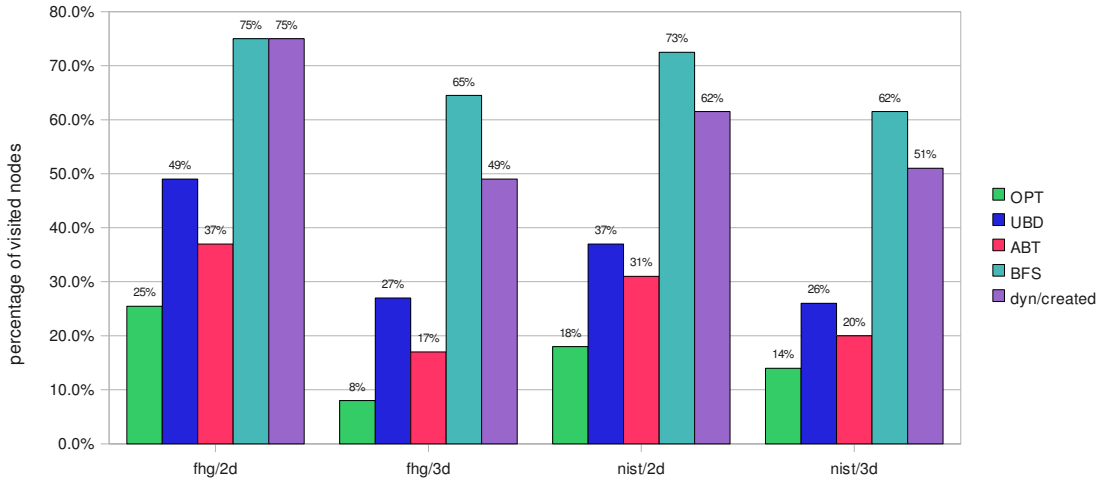


Figure 9.2: Averages percentage of visited nodes for the static UBD, ABT and BFS. Created nodes percentage (OPT) in a dynamic tree in relation to the minimal number of nodes for determining T^c, when visiting branches containing an element from T^c only once.

The ABT strategy has running times inferior to UBD, since a big amount of time is consumed by determining, if a branch is visited or not. Its run time heavily increases when having big UB sets, which are immanent for big $\tilde{\delta}$. Sometimes even BFS outperforms this strategy, despite visiting 63%–73% of the overall nodes in the static tree. When comparing the run times of all three traversing strategies, UB-driven traversing performs best. In the 2-d case it visits an average of 42.5% of the nodes in a static segmentation tree, in the 3-d case only 26.5%. Table 9.7 lists the average traversing times for UBD, ABT and BFS. Figures 9.3 illustrate, how the average run times for traversing increase with the size of $\tilde{\delta}$. For 3-dimensional $\tilde{\delta}$, ABT performs better when having 2-dimensional $\tilde{\delta}$.

Another trend is visible in appendix tables 1–4. For fhg-files the number of visited nodes increases (static and dynamic version), when $\tilde{\delta}$ increases. This is induced by the fact, that the minutiae in fhg instances are relatively dense. Here, with our $\tilde{\delta}$, we approach very fast $\frac{\tilde{v}}{2}$, and this results in very many big UB sets, which differ in only very few expressed template arcs and thus produce many non-dominated template arcs, increasing the number of visited branches. In contrast, the minutiae in the NIST files are relatively sparse. When calculating these instances, the number of visited nodes decreases, when $\tilde{\delta}$ increases. When having small $\tilde{\delta}$, there are fewer overlaps of template arcs and the tree contains many very small upper bound sets of size one or two, which must all be visited. When $\tilde{\delta}$ increases, more overlaps occur, and the bounding strategy becomes effective.

When determining T^c, the dynamic tree produces an average of 50% (2-d case) up to 67.5% (3-d case) of nodes when comparing the number of created dynamic nodes to the total amount of

Table 9.7: Average traversing times for fhg and nist files.

AVG run times	$\tilde{\delta}$	$(10,10)^\top$ trav[s]	$(20,20)^\top$ trav[s]	$(30,30)^\top$ trav[s]	$(40,40)^\top$ trav[s]	$(80,80)^\top$ trav[s]	$(120,120)^\top$
fhg	UBD	0.07	0.31	1.01	2.44	45.23	
	ABT	0.16	0.87	2.59	5.88	76.23	$\tilde{\delta} > \tilde{v}/2$
	BFS	0.15	0.68	1.78	3.74	49.77	
nist	UBD	1.73	2.62	5.00	9.19	140.91	575.66
	ABT	4.05	6.84	16.16	35.38	795.96	1340.93
	BFS	2.65	6.65	15.71	33.92	1205.75	2292.39

AVG run times	$\tilde{\delta}$	$(10,10,10)^\top$ trav[s]	$(20,20,20)^\top$ trav[s]	$(30,30,30)^\top$ trav[s]	$(40,40,40)^\top$ trav[s]	$(80,80,80)^\top$ trav[s]
fhg	UBD	0.03	0.18	1.40	8.36	-
	ABT	0.08	0.46	3.89	23.04	-
	BFS	0.08	0.67	5.89	34.92	-
nist	UBD	1.81	2.35	4.58	10.62	118.32
	ABT	4.92	7.12	13.46	30.27	424.13
	BFS	2.52	7.01	18.23	45.91	612.04

nodes in the static tree. It must be annotated, that the traversing process, that determines T^c, visits approximately 75%–107% of nodes. This is because of the nature of the algorithm, which sometimes backtracks while segmenting because it found a more promising branch than the branch that it just created on the basis of some dual value.

Nonetheless all three strategies clearly outperform the preprocessing. When taking into account for UBD, ABT and BFS the respective build time from the tables 9.3, 9.4 and 9.5, 9.6, the running times for all three bounding strategies are extremely shorter that the preprocessing from [ChwRai09]. UB-driven traversing is the strategy that performs best, despite the simple bounding strategy.

Figure 9.3 shows the relations of the run time (including building and traverse) for UBD, ABT, BFS and the dynamic tree algorithm `DynamicSearch-`T^c from section 7.3.1. The figures are again distinguished by fhg, nist, 2– and 3-dimensional $\tilde{\delta}$. In order to bring out more clearly the differences for the run times, two scales were used. In the diagrams on the left side of the figure 9.3 the run times are scaled linearly by run time in seconds, on the right side we used a logarithmic scale.

Table 9.8 lists the run times for the segmentation algorithms in comparison to the run times of the preprocessing by [ChwRai09] in percent. Overall, UBD performs in 1.91% (with a standard deviation of 1.76%) of the time needed by the preprocessing. ABT performs in 3.39% (2.96%), BFS in 4.14% (4.1%) and the dynamic tree in 2.62% (2.37%) of time.

Table 9.8: Percentage of run time in comparison to the preprocessing by [ChwRai09].

fhg/2d	$(10,10)^\top$	$(20,20)^\top$	$(30,30)^\top$	$(40,40)^\top$	**AVG**	stdev
UBD	1.53%	0.62%	0.27%	0.13%	**0.64%**	0.63%
ABT	2.59%	1.24%	0.51%	0.24%	**1.17%**	1.10%
BFS	2.47%	1.03%	0.39%	0.17%	**1.02%**	1.04%
dyn	1.53%	0.68%	0.36%	0.22%	**0.70%**	0.59%
fhg/3d	$(10,10,10)^\top$	$(20,20,20)^\top$	$(30,30,30)^\top$	$(40,40,40)^\top$	**AVG**	stdev
UBD	2.50%	5.23%	3.49%	1.48%	**3.18%**	1.60%
ABT	3.89%	8.41%	9.93%	3.24%	**5.62%**	2.46%
BFS	3.89%	10.80%	9.67%	4.67%	**7.26%**	3.48%
dyn	3.61%	6.82%	4.81%	2.92%	**4.54%**	1.71%

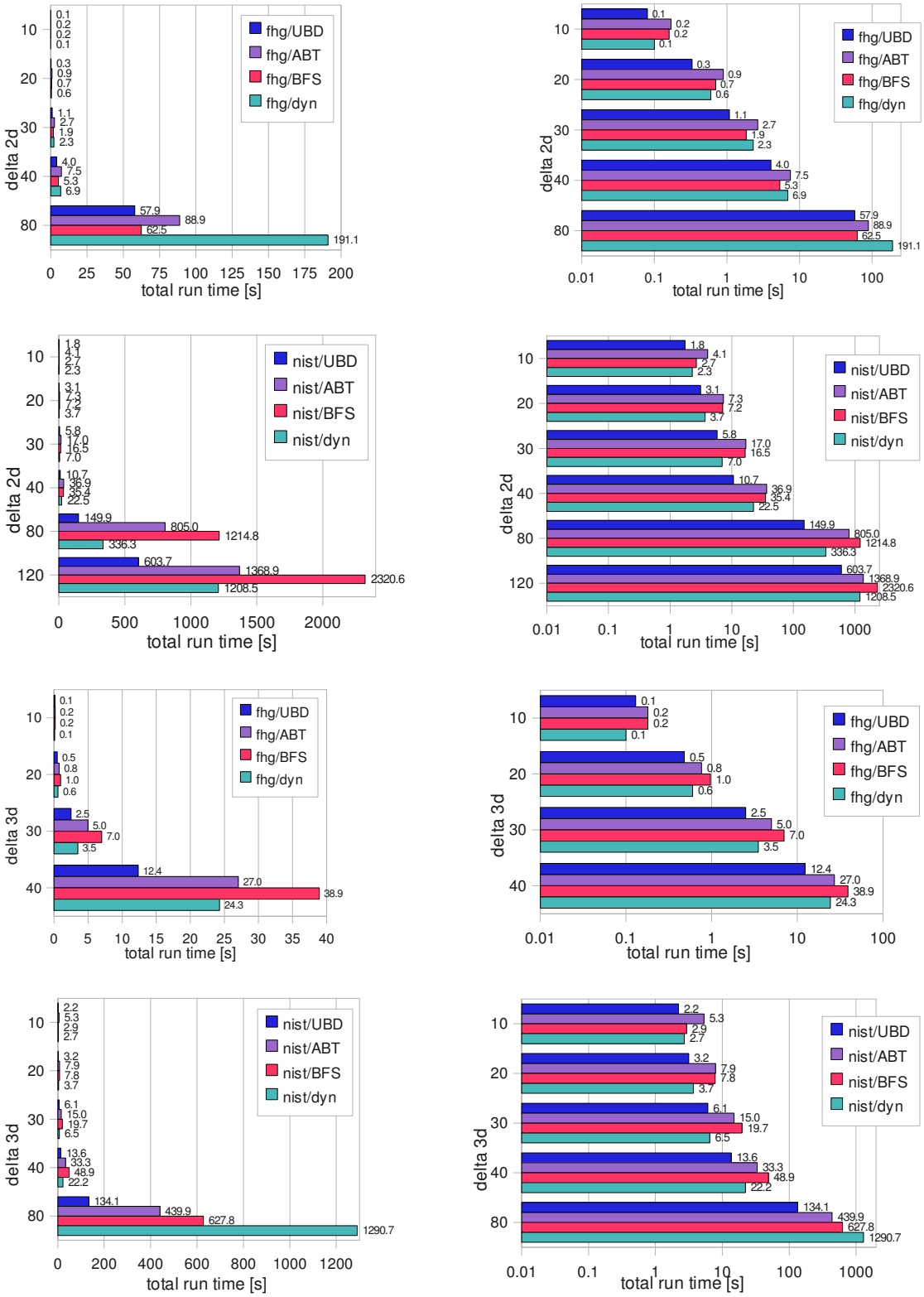


Figure 9.3: Fhg, nist instances running times for each traversing strategy, including the dynamic tree for 2– and 3-dimensional $\tilde{\delta}$. On the left side we used a linear scale, on the right side a logarithmic scale in order to make visible the differences for run times with small $\tilde{\delta}$.

9.2.2.1 Memory Usage of the Segmentation Tree

The segmentation tree has a main drawback. For some instances, when using a big $\tilde{\delta}$, like $\tilde{\delta} = (120, 120)^\top$ or $\tilde{\delta} = (80, 80, 80)^\top$ value, a memory overflow occurs. The cause of excessive memory usage are mainly the upper and lower bound sets at each node, used to store information about expressed template arcs. The greater $\tilde{\delta}$, the more elements are contained in these bound sets. When using a precision of 4 bytes for each bound set element, at the root node for a big instance, for example `nist-g-03` with $|V| = 102$ and thus $|B| = 10302$ uses already 330KB of storage. Segmentation trees for such big NIST instances with 2–10 millions of nodes use already more than 3.5 GB of memory. As memory on `G1` is limited, some instances terminated with a memory overflow. An improvement would be to decrease segmentation tree size. This could be done by omitting bound sets at nodes, where no difference regarding the parent bound set occurs. Another possibility is to save the differences for each bound set at each node.

9.2.3 Simulating the Pricing Problem

With the basis of random values that act as dual values we performed 100 and 1000 pricing iterations, as described in subsections 6.4.5 and 6.5.2. The test have been run on machine `L1`, with a clearly inferior processor as the other machines. Nonetheless, both algorithms perform very fast. The tests were performed based on the files, preprocessed by [ChwRai09] and listed in table 9.2. All iterations succeeded when comparing the result τ^*_{seg} and τ^*_{dynseg} to the $\tau^*_{T^c}$ from the preprocessed file, for the static as well for the dynamic tree. The run time increases moderately when $\tilde{\delta}$ increases. So, we can assess the robustness and quickness of the pricing problem algorithms. A comparison for average run times in seconds is listed in table 9.9. The static averages include once the build time, 100, 1000 iterations each for traversing (to get τ^*_{seg} and τ^*_{dynseg}) and searching $\tau^*_{T^c}$, and in the end comparing the result. The dynamic averages include traversing process, as searching $\tau^*_{T^c}$ in T^c. The dynamic tree may grow slowly in each pricing iteration, and is extended in each simulated pricing iteration by the part that is needed for completing the simulation. The average run times for the dynamic segmentation are significantly lower than the static segmentation tree average run times. Despite we build the static tree only once and then perform all the simulating iterations, the dynamic tree is much more effective. When searching for τ^*, both segmentation trees perform very well, and we derive, that the dynamic segmentation tree is better suited for solving the pricing problem and used as pricer.

Table 9.9: Average run times ('AVG t[s]') for fhg, nist data when simulating the pricing problem with random values and 100, 1000 pricing iterations ('pit'). Tests run on a single processor mobile machine `L1`.

100 pit	fhg AVG t[s]		nist AVG t[s]		1000 pit	fhg AVG t[s]		nist AVG t[s]	
$\tilde{\delta}$	static	dynamic	static	dynamic	$\tilde{\delta}$	static	dynamic	static	dynamic
$(10, 10)^\top$	0.90	0.22	-	-	$(10, 10)^\top$	7.65	1.66	-	-
$(20, 20)^\top$	2.05	0.46	9.19	2.13	$(20, 20)^\top$	14.67	3.40	88.09	19.20
$(30, 30)^\top$	4.22	0.89	-	-	$(30, 30)^\top$	26.97	6.43	-	-
$(40, 40)^\top$	7.91	1.62	15.78	3.10	$(40, 40)^\top$	44.58	11.36	137.57	28.04
$(80, 80)^\top$	-	-	45.11	-	$(80, 80)^\top$	-	-	281.82	-
$(10, 10, 10)^\top$	0.81	-	-	-	$(10, 10, 10)^\top$	7.68	-	-	-
$(20, 20, 20)^\top$	2.06	-	-	-	$(20, 20, 20)^\top$	16.67	-	-	-
$(40, 40, 40)^\top$	13.97	-	24.94	-	$(40, 40, 40)^\top$	65.62	-	217.60	-

9.3 Branch-and-Price Results for Fraunhofer Data

In the following, we evaluate the branch-and-price algorithm test runs in comparison to branch-and-cut test runs, and how node-label constraints impact on the overall performance. Basis test set are the 20 'fhg' files from the Fraunhofer institute. All test results have been run on machines `G1`. For reasons of space, we list the averages over each instance data set since listing

results on instance level would require too much space. In the following we list experimental results separated by solution approach.

9.3.1 Tested Parameters and Limits

For branch-and-price and branch-and-cut tests we need a small, a middle and a big value for k. As we first run tests with fhg-files, the selected values for k are $k = 10, 20, \max\{26, \ldots, 30\}$. Tested values for $\tilde{\delta}$ are $\tilde{\delta} = (10,10)^\top$, $(20,20)^\top$, $(30,30)^\top$, $(40,40)^\top$, $(80,80)^\top$, $(10,10,10)^\top$, $(30,30,30)^\top$, $(40,40,40)^\top$ and $(80,80,80)^\top$. The small $\tilde{\delta}$ values (10,20) generate small bounding boxes, the other middle (30,40) and big (80) bounding boxes. For the instance `ft-06` the big $\tilde{\delta} = (80,80)^\top, (80,80,80)^\top$ value exceeds the guideline $\tilde{\delta} < \frac{\tilde{v}}{2}$. Instances `ft-04`, `ft-06` and `ft-13` have a smaller number of data points than $\max\{26, \ldots, 30\}$, so they are omitted when averaging. For parameter $\tilde{\delta} = (80,80,80)^\top$ some sets T^c are not calculable neither with preprocessing by [ChwRai09] nor with a static tree pricing algorithm and become only computable with branch-and-price using a dynamic tree pricing algorithm. The solution approaches presented in the following in sections 9.3.2, 9.3.3 and 9.3.4 do not include this parameter value $\tilde{\delta} = (80,80,80)^\top$, the averages for the approach in section 9.3.5 do include it. Tables 9.11, 9.12, 9.13, 9.14 and 9.15 summarize average data for all tested solution approaches. In the following we describe the tables and the tested approaches in detail.
In preliminary tests we determined for fhg instances an algorithm run time limit of 14400 seconds, which are 4 hours. For these instances, we will not let branch-and-price tests run longer than this value. Also a limit of 26000 for pricing iterations was established and we end a test run if it exceeds this pricing iteration limit. We needed these limits, especially when allowing variable duplicates, to break the execution at some time point to prevent the tailing off effect (section 3.4.7).

9.3.2 Branch-and-Cut Reference Run Times

Basis for comparison are test runs with the branch-and-cut algorithm employing the directed cut model. This algorithm has been implemented by [ChwRai09], where a profound analysis of it can be found. We introduced the model shortly in section 4.2.4. All tests have been re-run on machines `G1` for better comparison of the run times. Parameters and time limit have been set as described in the previous section.
Tables 9.11, 9.12, 9.13, 9.14 and 9.15 list in rows 'BC/UBD' and 'BC/PP' run time data for branch-and-cut tests. Rows 'BC/UBD' lists average algorithm run time in seconds (column 'alg[s]'). Column 'tot[s]' shows average total run time in seconds when UB-driven traversing for a static tree (the fastest algorithm for determining T^c) is used for the preprocessing step. Rows 'BC/PP' lists average run times[1] with preprocessing by [ChwRai09]. Columns 'mi' show the amount of instances that are missing because of either exceeding the time or pricing iterations limit or because of memory usage. Tables 6 and 7 in the appendix show the standard deviations for 'BC/UBD', standard deviations for 'BC/PP' are much higher.

The branch-and-cut algorithm failed to complete two instances with 2-dimensional $\tilde{\delta}$ within the time limit. For 3-dimensional $\tilde{\delta}$ all test runs have been completed, except all $\tilde{\delta} = (80,80,80)^\top$ data for which no T^c could be determined. As we can see in these tables, branch-and-cut using UB-driven traversing for determining T^c outperforms branch-and-cut using the preprocessing by [ChwRai09]. Branch-and-cut works extremely good for 3-dimensional $\tilde{\delta}$. For small $\tilde{\delta}$ branch-and-cut run times improve with increasing k. When $\tilde{\delta}$ gets bigger than $(30,30)^\top$, $(30,30,30)^\top$ the opposite effect is noted. When regarding the compression rate in practical application, result sets having a small number of template arcs are favorable. This implies a big $\tilde{\delta}$ and therefore a big number of candidate template arcs T^c. Based on experimental data, [ChwRai09] derived that branch-and-cut works better with small sets T^c and big sets T^c have a disadvantageous effect. This disadvantage is remedied by branch-and-price, since here we do not need the set T^c anymore.

[1] Note that these preprocessing run times have been determined on machine `01`.

In the following we use the 'BC/UBD' run times as reference. The branch-and-cut result values for m, the minimal number of template arcs needed for encoding a template of size k, were used for determining correctness of branch-and-price. The minimal codebook sizes for tested parameters are summarized in table 5 in the appendix.

9.3.3 Complete SCF and MCF Formulation

As the single and multi commodity flow model presented in chapter 5 can be solved entirely with T replaced by T^c, extracted with UB-driven traversing from a static segmentation tree. Tests have been run for determining if the models are correct. Parameters k and $\tilde{\delta}$ have been set as discussed in section 9.3.1.
Tables 9.11, 9.12, 9.13, 9.14 and 9.15 summarize data for a single commodity flow formulation solved entirely. The tables list in row 'SCF/com' average algorithm run times in seconds (column 'alg[s]') and total run times including initialization step (column 'tot[s]') in seconds for the single commodity flow formulation. Column 'bbn' shows the average amount of branch-and-bound nodes. Averages for the multi commodity flow formulation are listed in rows 'MCF/com'. Column 'mi' lists again the amount of instances that failed to be calculated within the time limit. Tables 6 and 7 in the appendix show the standard deviations for 'SCF/com', standard deviation for 'MCF/com' is very high and has an average of 7199.

With the SCF and MCF solved entirely, all codebooks have been determined correctly. As with branch-and-cut here also no data could be collected for all $\tilde{\delta} = (80, 80, 80)^\top$ because the set T^c is not computable (memory usage) for this parameter value. For the MCF, already with 2-dimensional $\tilde{\delta}$ many instances surpass the time limit. For the tested parameters $\tilde{\delta} = (10, 10)^\top$, $(20, 20)^\top$, $(40, 40)^\top$, $(80, 80)^\top$, already 36% of the instances failed to be calculated within the time limit. With 3-dimensional $\tilde{\delta}$ even more instances are missing and hence they are not listed. We deduce that the SCF outperforms the MCF. Further experiments with pricing algorithms have shown, that the multi commodity flow formulation is not competitive enough for our purposes. The algorithms work, but run times for many instances lie wide beyond our time limit. In the following we will concentrate on the single commodity flow formulation.
When comparing the appropriate values (rows 'SCF/com') with the corresponding branch-and-cut values (columns 'BC/UBD'), the SCF performs relatively good, despite the weaker LP relaxation. For the SCF model, the average amount of branch-and-bound nodes (searched with BFS) is 862 (standard deviation 1218). The strength of the SCF can already be seen in these tests. As the size of T^c increases with increasing $\tilde{\delta}$ the run times also increase with branch-and-cut. With the SCF model the opposite effect occurs. Also, for small and middle $\tilde{\delta}$ the amount of possible solutions is very high and there exist many template arcs with the same size. This induces large plateaus of equally good solutions and the algorithm spends most of the time for determining if the found solution is optimal. When $\tilde{\delta}$ gets bigger the amount of expressed points for each template arc increases the optimal solution becomes unambiguous and is found very fast. Thus, the run times for $(40, 40)^\top$, $k = 10$ and $(80, 80)^\top$, $k = 30$ are relatively small. Because of this characteristic also very many branch-and-bound nodes $(1 - 38507)$ are created for small and middle 2-dimensional $\tilde{\delta}$, whereas very few nodes $(1 - 18)$ are needed when $\tilde{\delta}$ is big. Further effect can be seen for 3-dimensional and small $\tilde{\delta}$ where the amount of branch-and-bound nodes gets tinier when compared to 2-dimensional data.

9.3.4 Static Tree Pricing Algorithm for SCF

The static tree pricing algorithm was presented in section 6.4.3 and listed as pseudo code in section 6.6. Limits and parameters k and $\tilde{\delta}$ have been set as described in section 9.3.1. For the tests a huge amount of pricing possibilities arise, the main test subject was the impact on performance when using node-label constraints:

- We test how the static tree pricing algorithm performs when the SCF model uses only arc-label constraints and the pricing bases on the dual variables u_{ij}. In contrast to it we test the

performance, when the model additionally includes node-label constraints combined with the according pricing strategy described in section 5.5 based on the dual variables u_{ij} and μ_j.
- How do run times differ, when either a starting solution (star shaped spanning tree) or a Farkas pricing algorithm for determining a starting solution is used instead.
- How are run times affected, when pricing is restricted only to non-dominated template arcs or dominated arcs are allowed as well.
- How does the algorithm perform when only one arc with negative reduced cost per pricing step is determined, confronted to pricing all arcs with reduced cost < 0 in one pricing step.
- How does disabling or enabling variable duplicates perform. For the former, in each pricing step we check if we already have found the actual template arc. If the actual arc was already priced, we add another variable, if existing, with equal or inferior reduced cost instead. When doing this the problem turns out to be efficiently solved. The counterpart is to allow equal template arcs to be multiply priced.
- For the branch-and-bound decision tree we determine how depth first search (DFS) and best first search (BFS) perform.
- How does using continuous values for arcs variables x_{ij} affect performance.

All the presented options may be combined. As listing each such test combination would take too much space, we present the fastest variants. All tested variants determined the correct minimal codebook. Test runs using DFS for the branch-and-bound tree have been showed not to be competitive for our purposes. So for all presented test runs BFS has been used. Also tests with a pricing algorithm that allows variable duplicates could not reach not in the least the run times of the pricing algorithm that does not permit variable duplicates and marks the found variable after each pricing iteration and hence is not listed here.

Tables 9.11, 9.12, 9.13, 9.14 and 9.15 show test data for the static tree pricing algorithm based on a SCF model. As already described, in these tables columns 'alg[s]' lists average algorithm run time in seconds as well as average total run time (columns 'tot[s]') in seconds. Columns 'pit' shows the average amount of pricing iterations, columns 'pvar' the average number of priced variables. Columns 'bbn' shows the average amount of nodes created in the branch-and-bound decision tree. Rows 'BP/static' shows data for a static tree pricing algorithm in standard configuration that employs only arc-label constraints, uses a starting solution, determines in each pricing iteration only one arc with maximal negative reduced cost and allows no variable duplicates. The branch-and-bound decision tree uses BFS and we restricted the search to non-dominated template arcs. For rows 'BP/static/nlc' additionally node-label constraints have been used. For rows 'BP/static/fark' and 'BP/static/nlc/fark' a starting solution determined with Farkas pricing algorithm has been used instead of a star shaped spanning tree. Rows 'BP/static/dom' and 'BP/static/nlc/dom' show data for static tree pricing algorithms that allow dominated template arcs to be priced. The systematic should be clear for 'BP/static/nlc/fark/dom'. Rows 'BP/static/allarcs' lists run times for a static tree pricing algorithm that prices all arcs with reduced costs < 0 in one pricing iteration. Rows 'BP/static/nlc/cont' show average run times of the static tree pricing algorithm using an SCF, where arcs variables x_{ij} have been implemented as continuous values. This approach was considered because of the following. As described in [CaClPa09] it is possible to solve the MLST formulated as a flow network formulations with real valued arcs variables x. The result values for x may be fractional, but the results for the labels are correct. We assumed that this may be the case also for our extended variant (selected nodes, directed). Computational tests support this assumption, and codebooks were determined correctly. Averages for the runs (rows 'BP/static/nlc/cont') are presented in tables 9.11–9.15. Tables 6 and 7 in the appendix show in row 'BP/static/[AVG]' the standard deviation as an average over 'BP/static/[fark,dom,nlc,nlc/fark,nlc/dom,nlc/fark/dom]' since values for these static tree pricing algorithm variants are very similar.

In test runs employing only arc-label constraints and 2-dimensional $\tilde{\delta}$ one value (`ft-19`, $\tilde{\delta} = (20, 20)^\top$, $k = 20$)) exceeds the pricing iterations limit[2]. The same test runs employing additionally

[2]When more elaborate tests are done, time and pricing iterations limit should be increased.

node-label constraints were all computed. For 3-dimensional $\tilde{\delta} = (80, 80, 80)^\top$ again the static segmentation tree could not be build because of memory usage. This must be regarded when comparing values. Mainly data for branch-and-cut, the SCF solved completely and static tree pricing algorithm are easily compared.

Branch-and-Price Initialization Times The static tree pricing algorithm has to perform an initialization step. It has to load the SCF model, generate variables and constraints, build the entire static tree, and determine the starting solution, a star shaped spanning tree (naturally, if Farkas pricing was not enabled). The building of such a starting solution is described in 5.6. Average times for this step are listed in table 9.10 and correlate with the build time of a static tree. As we can see, this step increases the run times for big $\tilde{\delta} = (80, 80)^\top$ with an average of 20.0 seconds, which often is much more than the effective branch-and-price run time. For small $\tilde{\delta}$, the effect is almost negligible.

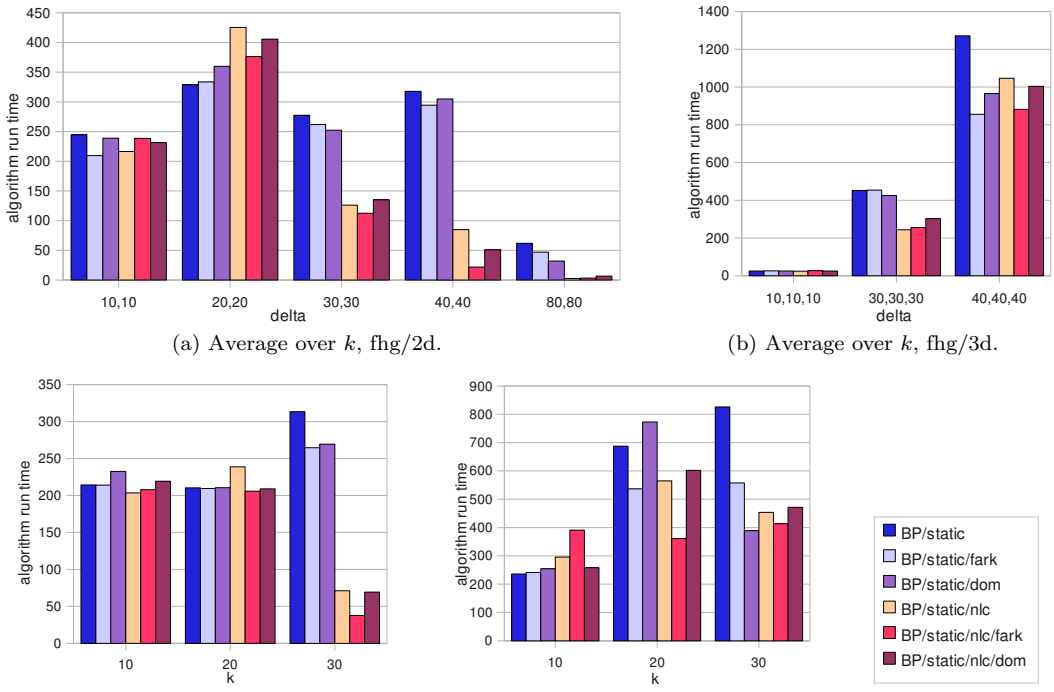
Table 9.10: Average initialization times in seconds for static tree pricing algorithms, 2-dimensional $\tilde{\delta}$.

fhg/2d						fhg/3d			
k	$(10,10)^\top$	$(20,20)^\top$	$(30,30)^\top$	$(40,40)^\top$	$(80,80)^\top$	k	$(10,10,10)^\top$	$(30,30,30)^\top$	$(40,40,40)^\top$
10	0.11	0.41	1.12	2.19	15.92	10	0.13	1.96	6.01
20	0.17	0.63	1.67	2.92	17.44	20	0.22	2.79	9.06
30	0.25	0.87	2.36	3.81	20.05	30	0.31	3.87	12.10
AVG	**0.18**	**0.64**	**1.72**	**2.97**	**17.80**	**AVG**	**0.22**	**2.87**	**9.05**
stdev	0.07	0.23	0.62	0.81	2.09	stdev	0.09	0.96	3.05

9.3.4.1 Results

All variants determined a correct minimal codebook. As we can see in tables 9.11, 9.12, 9.13, 9.14 and 9.15, branch and price performs very good for big 2-dimensional $\tilde{\delta} = (40, 40)^\top, (80, 80)^\top$ values. The number of priced variables with these parameters is very small when compared to the set T^c (discussed in section 9.4). So is the number of branch-and-bound nodes. For smaller $\tilde{\delta}$, and as k increases, more pricing iterations, as well as branch-and-bound nodes are needed to solve the problem. Accordingly, the run times are higher. For 3-dimensional $\tilde{\delta}$ the run times can not reach the excellent branch-and-cut run times, the best values were achieved with small $\tilde{\delta} = (10, 10, 10)^\top$. As we can see in the tables concerning 2-dimensional data, the run time decreases when k increases and $\tilde{\delta}$ is small. In case $\tilde{\delta}$ is big, the opposite effect occurs, and run times increase with the size of k. For the amount of pricing iterations, the same effect is noted, the amount of pricing iterations directly influence the run times. Best values for big $\tilde{\delta} = (80, 80)^\top$ are result when using node-label constraints, for big $\tilde{\delta} = (40, 40)^\top$ Farkas pricing is more effective. The effects of having a better starting set take effect the more $\tilde{\delta}$ increases from $(10, 10)^\top$ to $(40, 40)^\top$. When $\tilde{\delta}$ is very big, the best template arcs can be found very quickly and a starting set built by Farkas pricing or a star shaped spanning tree makes very few difference.
In the according tables we see, that the static tree pricing algorithm test runs which include node-label constraints as well as the according pricing strategy perform best. The algorithms using node-label constraints are faster than the variants using only arc-label constraints. Branch-and-bound nodes as well as pricing iterations and priced variables are significantly smaller, especially the more k increases. If node-label constraints are used in combination with Farkas pricing, a very good starting solution (which may be different from the one described in 5.6) is found and results mostly in smaller run times and less priced variables. Interestingly, when allowing the pricing of dominated template arcs the branch-and-price algorithm performs well too. Pricing all arcs with a reduced cost < 0 in one pricing iteration (using no node-label constraints) performs relatively good, but this approach is outperformed by the other tested variants. The smallest average amount of branch-and-bound nodes with an average of 289 (2-dimensional $\tilde{\delta}$) 292 (3-dimensional $\tilde{\delta}$) is achieved with the variant using node-label constraints. We can say that using node-label constraints reduce significantly the amount of branch-and-bound nodes. The overall

average of branch-and-bound nodes for the variants using node-label constraints is half of the ones needed for solving instances using no node-label constraints, but the average is influenced somewhat by relatively big branch-and-bound node values for $k = 20$, $\tilde{\delta} = (20, 20)^\top$. The variant using Farkas pricing in combination with node-label constraints has the smallest average amount of priced variables, which were averagely 299 with 2-dimensional $\tilde{\delta}$ and 638 for 3-dimensional $\tilde{\delta}$. Here, the amount of priced variables is small, since Farkas pricing determines better starting solutions with more template arcs that may be part of the solution. As fastest static tree pricing algorithm was determined 'BP/static/nlc/fark', the approaches 'BP/static/nlc' and 'BP/static/nlc/dom' are also relatively fast.
Figures 9.4 shows a comparison of run times for selected options. In figures 9.4a,9.4b the average was taken over k, thus depicting the performance for each $\tilde{\delta}$ parameter. In images 9.4c, 9.4d the average was taken over $\tilde{\delta}$ values, thus depicting the performance for different k. In these images we see that approaches that use node-label constraints perform better, the more k increases. For small $\tilde{\delta}$, the node-label constraints are not very meaningful and the brought in information becomes effective, when $\tilde{\delta}$ increases.



(a) Average over k, fhg/2d. (b) Average over k, fhg/3d.

(c) Average over $\tilde{\delta}$, fhg/2d. (d) Average over $\tilde{\delta}$, fhg/3d. (e) Legend.

Figure 9.4: Performance of selected options for the static tree pricing algorithm, figures 9.4a, 9.4b in the first row show the average over k, figures 9.4c, 9.4d in the second row the average over $\tilde{\delta}$.

Branch-and-price works correctly when arcs variables may hold continuous values (rows 'BP/static/nlc/cont'). The approach could not reach the average run times of the other approaches that use integer arcs variables, but for some parameter values good results for the amount of pricing iterations, priced variables and branch-and-bound nodes could be achieved. Higher run times are consequence of the SCIP framework that strongly utilizes heuristics, which are beneficial when having integer and binary variables.
The variant allowing variable duplicates was found to be the weakest and is not listed. For some instances a very big amount of variables (>100000) are multiply priced and increase needlessly the size of the LP. SCIP is able to cope with such multiple variables. All tested instances with this option enabled have run times exceeding greatly the run times presented here.

9.3.5 Dynamic Tree Pricing Algorithm for SCF

The dynamic tree pricing algorithm was presented in section 6.5 and listed as pseudo code in section 6.6. Parameters, run time and pricing iterations limit have been set as in 9.3.1. All tested variants determined a correct minimal codebook.
Again we have a huge amount of possibilities, the tested configurations were mainly the same as for the static tree pricing algorithm. Tables 9.11, 9.12, 9.13, 9.14 and 9.15 list data (columns 'alg[s]', 'tot[s]', 'pit', 'pvar', 'bbn') for the best variants using a dynamic tree pricing algorithm. Rows 'BP/dyn' list data for the standard configuration (uses only arc-label constraints, star shaped spanning tree as starting solution, price only non-dominated arcs, use BFS for the branch-and-bound decision tree and allow no variable duplicates). Runs with further options are denominated as previously: Farkas pricing ('fark'), allow dominated template arcs ('dom'), using additionally node-label constraints ('nlc'), implement arcs variables x_{ij} as continuous values ('cont'). Tables 6 and 7 in the appendix show in row 'BP/dyn/[AVG]' the standard deviation as an average over 'BP/dyn/[fark,dom,nlc,nlc/fark,nlc/dom,nlc/fark/dom]' since the values for all dynamic tree pricing algorithm are very similar.

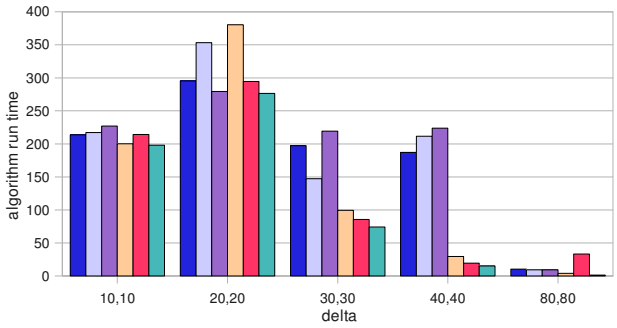
For the dynamic tree, an additional option for saving memory arises. We may delete the dynamic segmentation tree after each pricing operation and in each pricing iteration build anew exclusively the parts that we need for solving the pricing problem. In this manner only a small tree is built for each pricing step. Although there may be parts that are often built anew, but as the tree requires much storage, this is a simple option for coping with very big $\tilde{\delta}$. When variable duplicates are not allowed, we have to save the found template arcs separately from the tree. Also the pricing behaviour may be different because the tree behaviour changes. When we start in each pricing iteration with an empty tree and build each time straightforwardly in direction of the solution template arc, it may be a different one than a template arc found in a tree part that was built in some previous pricing iteration. Thus the pricing order may be influenced somewhat, an effect that also occurs when comparing the behaviour of static and dynamic tree pricing algorithm. The row 'BP/dyn/del/nlc/fark' lists data for the best such run.

For almost all variants and 2-dimensional parameters $\tilde{\delta}$, the number of completed instances was exactly the same, this eases comparison for the dynamic tree pricer options. As in the static tree pricing algorithm, all variants employing no node-label constraints failed to determine instance `ft-19`, $\tilde{\delta} = (20, 20)^\top$, $k = 20$ within the time limit. The test runs using node-label constraints finished all instances. For 3-dimensional parameters $\tilde{\delta}$ more instances could not be computed, mainly with $\tilde{\delta} = (80, 80, 80)^\top$ because of the storage requirements for the corresponding dynamic tree. In tables 9.11–9.15 we listed the amount of missing instances in column 'mi'. Note that instances for $\tilde{\delta} = (80, 80, 80)^\top$ became now calculable.

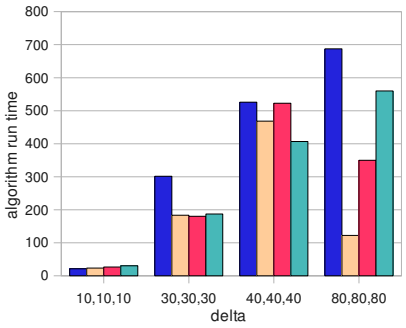
9.3.5.1 Results

Regarding k and $\tilde{\delta}$ the branch-and-price behaviour of the dynamic tree pricing algorithm is similar to the static version. The average amount of branch-and-bound nodes and priced variables varies, since the search for τ^* is a little bit different. The static tree pricing algorithm adds the first unpriced variable that is found in the tree, regardless if other template arcs may be found later with the same sum of dual values. Whereas the dynamic tree builds in direction of the biggest sum of dual values and thus some other template arc may be found and added first instead. In our tests, this behaviour resulted to be beneficial, since the amount of produced branch-and-bound nodes as well as priced variables decreases.
The initialization time for the dynamic tree pricing algorithm using node-label constraints or no node-label constraints performs in an average time slightly smaller than in the static tree, with exception of very big $\tilde{\delta}$ where the initialization step may take slightly more time. First we must regard that for the static tree pricing algorithm there is no data for $\tilde{\delta} = (80, 80, 80)^\top$, so these times seem different because the average value includes also these values that previously could

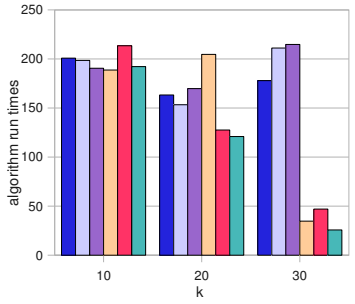
not be calculated. One minor cause is the dynamic tree behaviour which has a more complex handling for the next bounding box to be segmented, and that there may be segmented some small tree parts that hold no element from the starting solution and so increase the initialization time. Thus, variants using Farkas pricing perform better for the dynamic tree pricing algorithm. When a starting solution is priced based on Farkas coefficients, the initial template arcs set may already be smaller than the star shaped spanning tree (section 5.6) and less tree nodes required to find this starting set. With this option enabled the initialization step for fhg-data takes an average of 0.01 seconds, regardless of the tested parameters $k, \tilde{\delta}$. Also, finding a star shaped spanning tree starting solution, with dominated template arcs allowed, is relatively fast and takes an average of 0.52 seconds for all $k, \tilde{\delta}$. Farkas pricing is the better choice in the most of cases.
When regarding k and $\tilde{\delta}$ more closely, the same trends as with the static tree pricing algorithms are observed. For 2-dimensional parameters $\tilde{\delta}$ the run times decrease with increasing $\tilde{\delta}$. For big $\tilde{\delta}$ and small k, very good run times can be achieved. When k increases to 30 the run time decreases. For 3-dimensional $\tilde{\delta}$ again the opposite effect occurs and the branch-and-price run times can not quite reach the branch-and-cut run times. Like for 2-dimensional parameters $\tilde{\delta}$ the most time intensive runs are the ones with a medium k. Here also, using node-label constrains is beneficial and a good option for sure. When using them, the average amount of priced variables and branch-and-bound nodes decreases massively. The best approach for small and middle $\tilde{\delta}$ parameters is branch-and-cut, as tables 9.11–9.15 show. With branch-and-cut the 3-dimensional fhg-files test instances produced excellent run times. But the advantage of branch-and-price is that the instances for $\tilde{\delta} = (80, 80, 80)^\top$ are now calculable.
Figures 9.5 show a comparison of run times of the dynamic tree pricing algorithm with selected options. Figures 9.5a and 9.5b show the average over k, thus depicting the performance for the $\tilde{\delta}$ parameters. For figures 9.5c and 9.5d the average was taken over $\tilde{\delta}$ values, showing thus performance for different k values. As for the static tree pricing algorithm the utilization of node-label constraints performs the better the greater k, respective $\tilde{\delta}$.



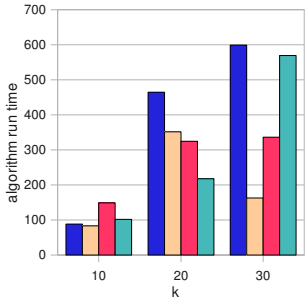
(a) Average over k, fhg/2d.



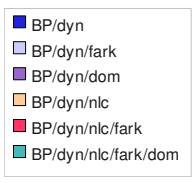
(b) Average over k, fhg/3d.



(c) Average over $\tilde{\delta}$, fhg/2d.



(d) Average over $\tilde{\delta}$, fhg/3d.



(e) Legend.

Figure 9.5: Performance of selected options for the dynamic tree pricing algorithm, figures 9.5a, 9.5b in the first row show the average over k, figures 9.5c, 9.5d in the second row the average over $\tilde{\delta}$.

Table 9.11: Average algorithm run time (column 'alg[s]'), total run time (column 'tot[s]'), pricing iterations (column 'pit'), priced variables (column 'pvar') and branch-and-bound nodes (column 'bbn'). Parameter $\tilde{\delta}$ 2-dimensional. Rows description in sections 9.3.2–9.3.5.

	fhg/2d	$(10,10)^\top$					$(20,20)^\top$					$(30,30)^\top$				
k	options	alg[s]	tot[s]	pit	pvar	bbn	alg[s]	tot[s]	pit	pvar	bbn	alg[s]	tot[s]	pit	pvar	bbn
10	BC/UBD	149.5	149.6	-	-	-	715.8	716.4	-	-	-	120.4	122.2	-	-	-
	BC/PP	149.5	158.0	-	-	-	715.8	806.0	-	-	-	120.4	763.9	-	-	-
	SCF/com	438.4	438.7	-	-	1286	479.1	480.0	-	-	1081	37.9	40.6	-	-	23
	MCF/com	10564.6	n/a	-	-	-	9496.5	n/a	-	-	-	n/a	n/a	-	-	-
	BP/sta	477.6	477.7	2057	491	1566	558.1	558.5	2704	960	1744	24.5	25.6	211	186	25
	BP/sta/fark	383.7	383.7	1853	499	1357	659.9	660.2	3129	968	2163	19.2	20.0	159	138	22
	BP/sta/dom	494.0	494.2	1962	539	1423	629.1	629.5	3077	1004	2072	35.4	36.5	286	257	29
	BP/sta/nlc	457.7	457.8	1757	471	1286	539.9	540.3	1790	1009	781	15.2	16.3	112	99	12
	BP/sta/nlc/fark	506.8	506.9	1977	548	1433	509.4	509.7	1732	915	818	16.9	17.7	124	113	12
	BP/sta/nlc/dom	492.1	492.2	1926	543	1384	568.5	568.9	1953	1031	921	32.0	33.0	255	230	26
	BP/sta/nlc/cont	571.0	571.1	2492	582	1910	810.3	810.7	2354	1226	1127	245.4	246.6	1136	693	442
	BP/sta/allarcs	587.7	587.8	1593	2953	1565	782.3	782.8	1694	6183	1544	47.4	48.4	54	6749	35
	BP/dyn	407.6	407.6	1747	508	1240	579.6	579.7	2975	1116	1859	11.8	12.0	147	131	16
	BP/dyn/fark	398.7	398.7	2102	504	1602	575.6	575.6	2952	910	2044	13.4	13.4	149	129	22
	BP/dyn/dom	444.5	444.5	1837	550	1288	487.5	487.5	2658	989	1669	17.3	17.3	206	182	23
	BP/dyn/nlc	403.3	403.4	1547	461	1086	522.2	522.3	1847	996	851	13.8	14.0	155	139	16
	BP/dyn/nlc/fark	479.4	479.4	2034	511	1526	543.5	543.5	2056	999	1058	10.6	10.6	85	76	10
	BP/dyn/nlc/fark/dom	408.5	408.5	1607	496	1114	538.7	538.7	2070	1040	1031	11.1	11.1	113	100	13
	BP/dyn/nlc/cont	980.3	980.3	3652	645	3007	714.4	714.5	2257	1185	1071	602.9	603.1	1881	773	1108
	BP/dyn/del/nlc/fark	591.3	591.3	2008	539	1472	544.2	544.2	1658	892	767	9.7	9.7	52	47	6
20	BC/UBD	51.4	51.5	-	-	-	387.6	388.1	-	-	-	309.6	311.4	-	-	-
	BC/PP	51.4	60.2	-	-	-	387.6	482.5	-	-	-	309.6	986.7	-	-	-
	SCF/com	129.7	130.0	-	-	334	408.7	409.7	-	-	1432	681.3	684.2	-	-	1854
	MCF/com	7178.6	n/a	-	-	-	3391.8	n/a	-	-	-	n/a	n/a	-	-	-
	BP/sta	156.0	156.2	804	307	496	200.3	200.9	1186	645	541	511.7	513.3	2835	1217	1618
	BP/sta/fark	178.3	178.4	910	299	616	90.3	90.6	766	368	401	540.4	541.2	2997	1118	1881
	BP/sta/dom	161.0	161.2	914	329	585	182.2	182.8	1443	683	761	506.8	508.2	2831	1208	1622
	BP/sta/nlc	154.2	154.4	679	258	421	628.3	629.0	2116	935	1180	284.4	286.1	1052	877	175
	BP/sta/nlc/fark	155.5	155.6	757	274	488	574.3	574.5	2103	633	1474	259.3	260.1	1096	941	158
	BP/sta/nlc/dom	158.5	158.7	720	280	440	495.4	496.0	1746	591	1156	301.6	303.0	1093	959	134
	BP/sta/nlc/cont	222.7	223.0	991	353	637	258.3	259.0	1497	571	925	1088.2	1089.9	1841	1268	573
	BP/sta/allarcs	258.5	258.6	646	1767	585	864.9	865.8	2119	3859	1961	879.8	881.3	1745	7527	1582
	BP/dyn	162.3	162.4	862	315	547	139.4	139.5	1028	585	445	394.2	394.5	2699	1333	1366
	BP/dyn/fark	188.7	188.7	1034	369	671	165.7	165.7	1123	547	579	302.3	302.3	2297	1000	1299
	BP/dyn/dom	174.0	174.0	1024	329	694	113.1	113.2	1081	656	426	419.4	419.4	3450	1594	1856
	BP/dyn/nlc	158.4	158.5	751	284	467	544.9	545.1	1948	806	1142	252.3	252.8	1080	930	150
	BP/dyn/nlc/fark	115.3	115.3	613	235	384	226.5	226.5	1197	467	733	219.5	219.5	1002	841	163
	BP/dyn/nlc/fark/dom	146.3	146.3	729	254	481	248.4	248.4	1351	571	784	183.0	183.0	906	767	142
	BP/dyn/nlc/cont	168.3	168.3	842	311	531	339.8	340.0	1972	661	1311	673.7	674.2	1527	1104	423
	BP/dyn/del/nlc/fark	168.6	168.6	755	291	469	477.1	477.1	1753	740	1016	134.8	134.9	644	547	99
30	BC/UBD	27.8	28.0	-	-	-	153.6	154.2	-	-	-	354.5	356.5	-	-	-
	BC/PP	27.8	37.5	-	-	-	153.6	258.3	-	-	-	354.5	1102.8	-	-	-
	SCF/complete	91.7	92.0	-	-	208	220.9	221.9	-	-	350	165.6	168.6	-	-	496
	MCF/com	5646.4	n/a	-	-	-	6139.4	n/a	-	-	-	n/a	n/a	-	-	-
	BP/sta	100.0	100.2	572	260	311	227.7	228.6	1178	600	577	296.2	298.4	1895	1168	727
	BP/sta/fark	66.6	66.6	425	252	183	250.7	251.0	1312	654	663	226.4	227.2	1599	856	746
	BP/sta/dom	61.6	61.9	381	213	167	268.3	269.0	1421	662	758	214.5	216.2	1557	821	735
	BP/sta/nlc	37.7	38.0	226	143	83	107.6	108.6	611	444	167	78.8	81.2	545	391	153
	BP/sta/nlc/fark	52.9	53.0	313	170	151	45.9	46.2	371	284	92	61.2	62.0	468	336	135
	BP/sta/nlc/dom	43.5	43.8	275	169	106	153.0	153.8	919	520	399	71.6	73.4	517	403	113
	BP/sta/nlc/cont	83.7	84.0	481	207	274	155.5	156.6	697	391	306	39.8	42.3	476	268	207
	BP/sta/allarcs	98.8	98.9	251	1433	193	524.9	526.0	968	4883	832	400.6	402.7	1090	9621	958
	BP/dyn	71.6	71.7	440	237	203	166.8	167.0	1138	688	450	185.7	186.2	1789	984	805
	BP/dyn/fark	63.9	63.9	419	227	201	318.4	318.4	1764	628	1141	126.4	126.4	1400	661	742
	BP/dyn/dom	62.3	62.3	410	227	183	237.4	237.5	1571	839	732	221.3	221.4	1706	1037	669
	BP/dyn/nlc	38.6	38.7	252	162	90	73.8	74.1	487	313	174	32.7	33.6	380	278	101
	BP/dyn/nlc/fark	48.6	48.6	281	156	133	113.3	113.3	792	396	402	26.4	26.4	298	182	119
	BP/dyn/nlc/fark/dom	39.3	39.3	243	168	85	42.0	42.0	386	296	94	28.8	28.8	321	216	108
	BP/dyn/nlc/cont	126.3	126.5	666	217	448	186.7	187.0	978	439	539	15.0	15.9	348	177	170
	BP/dyn/del/nlc/fark	67.0	67.0	327	179	156	48.4	48.5	331	236	99	52.3	52.3	366	273	96

Table 9.12: Average algorithm run time (column 'alg[s]'), total run time (column 'tot[s]'), pricing iterations (column 'pit'), priced variables (column 'pvar') and branch-and-bound nodes (column 'bbn'). Parameter $\tilde{\delta}$ 2-dimensional. Rows description in sections 9.3.2–9.3.5.

	fhg/2d	$(40,40)^\top$					$(80,80)^\top$					
k	options	alg[s]	tot[s]	pit	pvar	bbn	alg[s]	tot[s]	pit	pvar	bbn	mi
10	BC/UBD	126.9	130.9	-	-	-	695.6	753.5	-	-	-	0
	BC/PP	126.9	3239.1	-	-	-	695.6	n/a	-	-	-	0
	SCF/com	6.8	12.8	-	-	1	77.8	155.8	-	-	1	0
	MCF/com	723.2	n/a	-	-	-	n/a	n/a	-	-	-	15, all 30, 80
	BP/static	8.3	10.5	64	60	5	3.3	19.3	4	3	1	0
	BP/static/fark	5.1	6.8	35	32	4	2.3	15.9	2	2	1	0
	BP/static/dom	2.0	4.1	5	4	1	2.8	17.7	3	2	1	0
	BP/static/nlc	2.6	4.9	8	7	1	2.3	18.4	3	2	1	0
	BP/static/nlc/fark	3.4	5.1	6	5	2	3.3	16.8	1	1	1	0
	BP/static/nlc/dom	2.5	4.6	7	6	1	1.8	16.8	3	2	1	0
	BP/static/nlc/cont	84.6	87.0	327	257	70	62.4	79.1	109	52	58	1
	BP/static/allarcs	16.4	19.5	4	8213	2	41.0	56.9	2	10109	1	0
	BP/dyn	3.2	3.6	17	15	1	2.4	15.0	4	3	1	0
	BP/dyn/fark	2.7	2.7	8	7	2	2.3	2.3	2	2	1	0
	BP/dyn/dom	1.7	1.7	5	4	1	2.0	2.3	3	2	1	0
	BP/dyn/nlc	2.0	2.4	8	7	1	2.0	15.2	3	2	1	0
	BP/dyn/nlc/fark	3.0	3.0	6	5	1	30.6	30.6	1	1	1	0
	BP/dyn/nlc/fark/dom	1.9	1.9	5	5	1	0.8	0.8	1	1	1	0
	BP/dyn/nlc/cont	30.0	30.4	205	158	47	151.2	164.4	957	474	482	0
	BP/dyn/del/nlc/fark	3.2	3.2	6	5	1	33.0	33.0	1	1	1	0
20	BC/UBD	112.3	116.6	-	-	-	1020.5	1078.5	-	-	-	0
	BC/PP	112.3	3387.7	-	-	-	1020.5	n/a	-	-	-	0
	SCF/com	338.6	345.2	-	-	1221	67.7	145.1	-	-	2	0
	MCF/com	1166.9	n/a	-	-	-	n/a	n/a	-	-	-	25, all 30, 80
	BP/static	176.1	179.0	1384	844	540	7.7	25.2	14	12	3	1
	BP/static/fark	231.5	233.3	1513	967	548	6.3	19.7	12	8	5	1
	BP/static/dom	194.7	197.4	1361	955	406	8.0	24.0	17	14	3	1
	BP/static/nlc	125.2	128.4	528	473	55	1.9	19.9	3	2	1	0
	BP/static/nlc/fark	36.2	38.0	181	162	20	3.5	17.1	1	1	1	0
	BP/static/nlc/dom	88.1	90.8	530	492	38	1.7	17.9	3	2	1	0
	BP/static/nlc/cont	366.5	369.8	673	536	137	108.0	126.5	114	98	16	3
	BP/static/allarcs	505.3	509.4	726	9086	620	102.6	123.4	19	15359	14	0
	BP/dyn	114.8	115.7	1488	910	578	6.1	22.3	65	51	14	0
	BP/dyn/fark	106.4	106.4	1395	930	467	4.0	4.0	21	18	4	1
	BP/dyn/dom	140.2	140.3	1487	1041	447	2.8	3.3	11	10	2	1
	BP/dyn/nlc	65.7	66.9	621	572	48	2.4	19.6	3	2	1	0
	BP/dyn/nlc/fark	41.2	41.2	271	235	38	35.8	35.8	12	9	4	0
	BP/dyn/nlc/fark/dom	26.0	26.0	267	240	28	1.3	1.3	3	2	2	0
	BP/dyn/nlc/cont	20.8	22.0	237	201	35	3.3	20.6	6	3	4	5
	BP/dyn/del/nlc/fark	30.9	30.9	172	155	18	36.8	36.8	2	2	2	0
30	BC/UBD	556.2	561.2	-	-	-	2542.3	2605.8	-	-	-	2
	BC/PP	556.2	4600.8	-	-	-	2542.3	n/a	-	-	-	2
	SCF/com	2201.7	2208.7	-	-	4647	91.0	188.0	-	-	1	0
	MCF/com	5668.0	n/a	-	-	-	n/a	n/a	-	-	-	21, all 30, 80
	BP/static	768.8	772.6	2905	1671	1233	175.1	195.2	290	277	13	0
	BP/static/fark	647.0	648.9	2835	1369	1469	132.4	147.0	214	199	16	0
	BP/static/dom	717.8	721.0	2836	1570	1265	84.7	102.3	149	137	12	0
	BP/static/nlc	127.1	131.6	393	379	14	4.1	25.2	11	7	4	0
	BP/static/nlc/fark	25.8	27.7	106	97	11	3.0	17.5	1	1	1	0
	BP/static/nlc/dom	61.8	65.2	253	244	9	16.5	34.5	42	37	5	0
	BP/static/nlc/cont	36.0	40.5	151	146	5	680.9	702.7	405	396	9	1
	BP/static/allarcs	1325.3	1329.6	1067	13353	945	188.2	208.4	3	28017	1	0
	BP/dyn	443.2	444.5	2850	1577	843	22.4	45.1	137	126	11	0
	BP/dyn/fark	525.9	525.9	3320	1656	1667	21.5	21.5	141	132	10	0
	BP/dyn/dom	529.4	529.5	3215	1926	1289	23.6	24.1	273	258	14	0
	BP/dyn/nlc	20.6	22.6	219	210	8	7.7	31.7	26	21	5	0
	BP/dyn/nlc/fark	13.8	13.8	128	119	11	33.2	33.2	1	1	1	0
	BP/dyn/nlc/fark/dom	17.5	17.5	157	152	8	1.5	1.5	21	20	2	0
	BP/dyn/nlc/cont	22.0	23.9	190	183	7	4.4	28.4	15	13	1	1
	BP/dyn/del/nlc/fark	18.9	18.9	109	106	5	33.9	33.9	1	1	1	0

Table 9.13: Average algorithm run time (column 'alg[s]'), total run time (column 'tot[s]'), pricing iterations (column 'pit'), priced variables (column 'pvar') and branch-and-bound nodes (column 'bbn'). Parameter $\tilde{\delta}$ 3-dimensional. Rows description in sections 9.3.2–9.3.5.

	fhg/3d	$(10,10,10)^\top$					$(30,30,30)^\top$				
k	options	alg[s]	tot[s]	pit	pvar	bbn	alg[s]	tot[s]	pit	pvar	bbn
10	BC/UBD	11.2	11.3	-	-	-	32.1	34.6	-	-	-
	BC/PP	11.2	14.8	-	-	-	32.1	104.7	-	-	-
	SCF/com	12.5	12.7	-	-	76	65.3	68.6	-	-	132
	BP/static	37.7	37.8	197	110	87	228.5	230.4	919	593	326
	BP/static/fark	32.3	32.4	285	137	156	291.2	292.4	1310	720	593
	BP/static/dom	32.6	32.7	204	108	96	189.6	191.5	791	521	270
	BP/static/nlc	32.3	32.5	203	124	79	156.7	158.7	582	464	118
	BP/static/nlc/fark	35.6	35.7	199	126	80	154.3	155.5	574	455	121
	BP/static/nlc/dom	29.1	29.3	192	119	73	177.6	179.5	653	505	148
	BP/static/nlc/cont	72.8	73.0	420	153	267	270.6	272.6	901	702	198
	BP/static/allarcs	46.1	46.2	202	965	200	262.7	264.6	308	16804	289
	BP/dyn	31.5	31.5	202	114	88	133.3	133.4	894	570	324
	BP/dyn/nlc	31.9	31.9	209	128	81	95.5	95.6	627	503	123
	BP/dyn/nlc/fark	34.9	34.9	188	125	70	95.5	95.5	657	524	136
	BP/dyn/nlc/fark/dom	41.8	41.8	239	129	118	106.2	106.2	734	593	143
	BP/dyn/nlc/cont	65.1	65.1	402	148	254	179.9	180.0	974	751	222
	BP/dyn/del/nlc/fark	50.6	50.6	227	131	104	170.5	170.5	666	497	171
20	BC/UBD	4.5	4.6	-	-	-	92.7	95.3	-	-	-
	BC/PP	4.5	8.1	-	-	-	92.7	169.1	-	-	-
	SCF/com	14.0	14.2	-	-	81	475.1	478.6	-	-	1262
	BP/static	27.7	27.9	242	115	127	641.7	644.4	2503	878	1626
	BP/static/fark	31.8	31.8	226	122	117	624.2	625.5	2715	887	1835
	BP/static/dom	31.1	31.3	191	107	84	626.1	628.9	2452	870	1582
	BP/static/nlc	27.7	27.9	165	100	65	451.5	454.4	1327	830	497
	BP/static/nlc/fark	31.1	31.2	179	120	72	411.7	413.0	1426	812	618
	BP/static/nlc/dom	36.1	36.3	197	106	91	480.5	483.3	1515	896	619
	BP/static/nlc/cont	49.4	49.7	211	105	105	810.7	813.5	2752	888	1863
	BP/static/allarcs	41.1	41.3	238	708	229	1101.8	1104.6	1660	12768	1546
	BP/dyn	22.5	22.5	191	102	88	511.9	512.1	2741	917	1827
	BP/dyn/nlc	25.4	25.5	161	96	65	356.0	356.3	1543	918	625
	BP/dyn/nlc/fark	31.4	31.4	175	115	72	288.4	288.4	1308	772	540
	BP/dyn/nlc/fark/dom	33.1	33.1	195	122	84	388.3	388.3	2017	1113	908
	BP/dyn/nlc/cont	41.9	42.0	203	114	88	690.4	690.7	2698	841	1856
	BP/dyn/del/nlc/fark	37.4	37.4	158	108	61	491.9	491.9	1226	713	517
30	BC/UBD	5.0	5.1	-	-	-	82.6	85.5	-	-	-
	BC/PP	5.0	9.0	-	-	-	82.6	166.6	-	-	-
	SCF/com	7.8	8.1	-	-	20	371.3	375.0	-	-	1815
	BP/static	10.4	10.7	111	67	45	486.8	490.6	2145	947	1198
	BP/static/fark	14.6	14.7	128	90	58	446.1	447.5	2283	879	1413
	BP/static/dom	11.2	11.4	114	71	43	461.1	464.8	2065	840	1224
	BP/static/nlc	13.2	13.5	80	63	17	123.2	127.1	570	407	162
	BP/static/nlc/fark	15.9	16.0	79	75	22	203.7	205.1	815	453	370
	BP/static/nlc/dom	11.8	12.2	74	60	14	248.6	252.2	907	532	375
	BP/static/nlc/cont	15.7	16.1	106	73	33	458.3	462.3	2105	724	1380
	BP/static/allarcs	11.9	12.2	96	241	87	530.5	534.4	1062	11569	913
	BP/dyn	10.4	10.5	110	66	45	259.7	260.0	1898	871	1027
	BP/dyn/nlc	13.1	13.2	82	63	19	98.4	98.9	696	477	219
	BP/dyn/nlc/fark	14.0	14.0	68	70	16	157.1	157.1	913	456	464
	BP/dyn/nlc/fark/dom	16.4	16.4	96	80	34	67.4	67.5	575	397	186
	BP/dyn/nlc/cont	14.5	14.6	101	70	30	150.5	151.0	1428	576	851
	BP/dyn/del/nlc/fark	18.5	18.5	88	74	31	117.8	117.9	470	329	148

Table 9.14: Average algorithm run time (column 'alg[s]'), total run time (column 'tot[s]'), pricing iterations (column 'pit'), priced variables (column 'pvar') and branch-and-bound nodes (column 'bbn'). Parameter $\tilde{\delta}$ 3-dimensional. Rows description in sections 9.3.2–9.3.5.

	fhg/3d	$(40,40,40)^\top$					$(80,80,80)^\top$					
k	options	alg[s]	tot[s]	pit	pvar	bbn	alg[s]	tot[s]	pit	pvar	bbn	mi
10	BC/UBD	75.1	87.5	-	-	-			n/a			0
	BC/PP	75.1	908.4	-	-	-			n/a			0
	SCF/com	183.2	197.7	-	-	570			n/a			0, all 80
	BP/static	442.0	448.1	1414	904	509			n/a			0, all 80
	BP/static/fark	401.1	405.4	1343	774	571			n/a			0, all 80
	BP/static/dom	541.5	547.6	1779	1057	723			n/a			0, all 80
	BP/static/nlc	698.7	704.6	1671	1248	423			n/a			1, all 80
	BP/static/nlc/fark	983.2	987.5	1898	1432	468			n/a			0, all 80
	BP/static/nlc/dom	566.4	572.1	1372	1091	281			n/a			1, all 80
	BP/static/nlc/cont	1008.7	1014.7	2446	1838	608			n/a			1, all 80
	BP/static/allarcs	1079.9	1086.2	714	50430	629			n/a			0, all 80
	BP/dyn	184.2	184.6	1551	975	575	4.8	43.6	9	8	1	0
	BP/dyn/nlc	202.5	203.1	1286	1060	225	3.8	48.7	6	5	1	0
	BP/dyn/nlc/fark	461.8	461.8	1966	1481	487	3.5	3.6	4	4	1	0
	BP/dyn/nlc/fark/dom	255.0	255.0	1629	1248	383	4.0	4.0	13	11	3	0
	BP/dyn/nlc/cont	453.5	454.0	2261	1756	505	184.0	223.1	681	496	184	2
	BP/dyn/del/nlc/fark	462.1	462.1	1488	1144	346	5.4	5.4	4	4	1	1
20	BC/UBD	252.0	264.9	-	-	-			n/a			0
	BC/PP	252.0	1129.0	-	-	-			n/a			0
	SCF/com	571.3	586.8	-	-	3053			n/a			0, all 80
	BP/static	1393.4	1402.3	5585	1866	3720			n/a			1, all 80
	BP/static/fark	954.7	959.4	4057	1352	2709			n/a			2, all 80
	BP/static/dom	1661.5	1669.9	6654	2114	4541			n/a			0, all 80
	BP/static/nlc	1216.2	1225.4	2408	1741	666			n/a			0, all 80
	BP/static/nlc/fark	639.8	644.3	1836	1206	632			n/a			0, all 80
	BP/static/nlc/dom	1289.4	1297.9	2731	1990	741			n/a			0, all 80
	BP/static/nlc/cont	2807.1	2816.6	6460	2747	3712			n/a			3, all 80
	BP/static/allarcs	2270.4	2279.4	3577	38867	3315			n/a			1, all 80
	BP/dyn	484.8	485.6	4303	1761	2542	839.7	885.2	2520	2274	246	8
	BP/dyn/nlc	744.3	745.4	3150	2245	904	280.7	331.1	802	771	31	6
	BP/dyn/nlc/fark	658.8	658.8	3129	1819	1313	319.2	319.2	810	775	36	3
	BP/dyn/nlc/fark/dom	431.7	431.7	2780	1855	927	17.3	17.3	149	135	16	5
	BP/dyn/nlc/cont	1264.7	1265.8	7182	2318	4864	134.5	191.1	448	432	16	7
	BP/dyn/del/nlc/fark	680.7	680.8	2252	1466	789	613.1	613.1	597	571	28	1
30	BC/UBD	263.4	277.5	-	-	-			n/a			0
	BC/PP	263.4	1231.9	-	-	-			n/a			0
	SCF/com	1308.9	1325.5	-	-	2753			n/a			0, all 80
	BP/static	1980.8	1992.5	4458	2098	2360			n/a			1, all 80
	BP/static/fark	1212.3	1216.8	2889	1517	1378			n/a			2, all 80
	BP/static/dom	694.6	703.5	2010	1420	590			n/a			3, all 80
	BP/static/nlc	1224.3	1236.7	1841	1240	601			n/a			0, all 80
	BP/static/nlc/fark	1021.6	1026.4	1511	1061	456			n/a			0, all 80
	BP/static/nlc/dom	1154.3	1165.2	1718	1359	359			n/a			0, all 80
	BP/static/nlc/cont	667.6	679.0	1238	859	378			n/a			1, all 80
	BP/static/allarcs	1613.4	1623.9	1310	40866	1108			n/a			2, all 80
	BP/dyn	908.4	909.7	4162	1974	2189	1217.5	1257.9	4776	4086	689	11
	BP/dyn/nlc	458.5	460.4	1590	976	613	82.1	147.3	236	224	12	7
	BP/dyn/nlc/fark	447.0	447.1	1300	985	319	727.1	727.1	964	946	21	4
	BP/dyn/nlc/fark/dom	534.1	534.1	1596	1251	352	1658.5	1658.5	1266	1249	19	2
	BP/dyn/nlc/cont	276.9	278.7	1427	982	444	77.5	152.6	203	194	9	6
	BP/dyn/del/nlc/fark	679.7	679.7	1503	1069	441	153.9	153.9	236	228	11	5

Table 9.15: Averages over k, $\tilde{\delta}$ for the tested algorithms branch-and-cut (row 'BC'), SCF and MCF solved entirely (rows 'SCF/com' and 'MCF/com'), branch-and-price using a static tree pricing algorithm (all rows 'BP/static') and a dynamic tree pricing algorithm (rows 'BP/dyn').

AVG over k, $\tilde{\delta}$						
fhg/2d	alg[s]	tot[s]	pit	pvar	bbn	mi
BC/UBD	488.3	501.6	-	-	-	2
BC/PP	488.3	1323.6	-	-	-	2, PP t[s] for $(80,80)^\top$
SCF/com	362.5	381.4	-	-	862	0
MCF/com	5552.8	n/a	-	-	-	61, all $(30,30)^\top$, $(80,80)^\top$
BP/static	246.1	250.7	1207	580	627	1
BP/static/fark	229.3	232.7	1184	515	672	1
BP/static/dom	237.5	241.7	1216	560	656	1
BP/static/nlc	171.1	176.0	656	366	289	0
BP/static/nlc/fark	150.5	153.9	616	299	320	0
BP/static/nlc/dom	165.9	170.2	683	367	315	0
BP/static/nlc/cont	320.9	325.9	916	470	446	5
BP/static/allarcs	441.6	446.6	799	8607	723	0
BP/dyn	180.7	184.4	1159	572	559	1
BP/dyn/fark	187.7	187.7	1209	515	697	1
BP/dyn/dom	191.8	191.9	1263	643	620	1
BP/dyn/nlc	142.7	146.7	622	346	276	0
BP/dyn/nlc/fark	129.4	129.4	585	282	306	0
BP/dyn/nlc/fark/dom	113.0	113.0	545	289	260	0
BP/dyn/nlc/cont	269.3	273.3	1049	436	612	6
BP/dyn/del/nlc/fark	150.0	150.0	546	268	281	0
fhg/3d	alg[s]	tot[s]	pit	pvar	bbn	mi
BC/UBD	90.9	96.2	-	-	-	0, all $(80,80,80)^\top$.
BC/PP	90.9	415.7	-	-	-	0, all $(80,80,80)^\top$.
SCF/com	334.4	340.8	-	-	1085	0, all $(80,80,80)^\top$.
BP/static	583.2	587.2	1953	842	1111	2, all $(80,80,80)^\top$.
BP/static/fark	445.4	447.3	1693	720	981	4, all $(80,80,80)^\top$.
BP/static/dom	472.1	475.7	1807	790	1017	3, all $(80,80,80)^\top$.
BP/static/nlc	438.2	442.3	983	691	292	1, all $(80,80,80)^\top$.
BP/static/nlc/fark	388.5	390.5	946	638	315	0, all $(80,80,80)^\top$.
BP/static/nlc/dom	443.7	447.6	1040	740	300	1, all $(80,80,80)^\top$.
BP/static/nlc/cont	684.6	688.6	1849	899	949	5, all $(80,80,80)^\top$.
BP/static/allarcs	773.1	777.0	1018	19247	924	3, all $(80,80,80)^\top$.
BP/dyn	384.1	394.7	1946	1143	803	19
BP/dyn/nlc	199.3	213.1	866	622	243	13
BP/dyn/nlc/fark	269.9	269.9	957	673	290	7
BP/dyn/nlc/fark/dom	296.2	296.2	941	682	264	7
BP/dyn/nlc/cont	294.5	309.1	1501	723	777	15
BP/dyn/del/nlc/fark	290.1	290.1	743	528	221	7

The overall average run times for branch-and-price using a dynamic tree pricing algorithm are smaller than the ones determined with a static tree pricing algorithm, so it is the better option. We conclude that the best pricing strategy for branch-and-price is a dynamic tree pricing algorithm using node-label constraints and Farkas pricing. Also allowing the pricing of dominated template arcs has good run times.

Finally, we shortly analyze the variant that deletes the dynamic tree after each pricing iteration. Naturally, all runs that delete the tree have run times inferior the the original variants. Focus was the applicability to the bigger data sets, for example the nist data set. As these instances have many more data points and bigger domain sizes, building an entire segmentation tree is extremely memory intensive, up to the extent that for very big $\tilde{\delta}$ no segmentation tree can be built because of a memory overflow. Thus we tested, how much slower the versions are that delete the dynamic tree in each pricing iteration. For all such variants, the memory usage for big $\tilde{\delta}$ is generally very good. The overall run time average increases especially when the deletion is often performed (many pricing iterations). When $\tilde{\delta}$ is small this variant has good run times since the according dynamic tree has few levels. If $\tilde{\delta}$ gets bigger, many nodes have to be created and deleted again for each pricing iteration, thus slowing down the algorithm. When we compare for the best variant 'BP/dyn/deltree/nlc/fark' (listed in the results tables) the run times for both versions, the one that deletes the dynamic tree and the other being standard dynamic tree pricing without deletion process, can be said: The version that deletes the tree performs in an average 110.2% of the time needed by the original version, needs 83.6% of the amount pricing iterations, prices 83.4% of the variables and creates 84.2% of the branch-and-bound nodes. The different pricing behaviour (described in section 9.3.5) is beneficial. For the variant 'BP/dyn/deltree' that uses no additional options, the version that deletes the tree has run times that take 239.8% of the run time (pit: 97.0%, pvar: 102.1%, bbn: 92.7%) for the respective version that does not delete the tree, with 'BP/dyn/deltree/nlc' run times need 173.4% (pit: 108.9%, pvar: 107.6%, bbn; 111.4%) of the time needed by 'BP/dyn/nlc'.
The best option for big instances is to use node-label constraints in combination with Farkas pricing. If memory has to be saved starting in each pricing iteration with an empty dynamic tree is an option.

9.3.6 Summary for Pricing Algorithms with Fraunhofer Data Tests

Table 9.15 lists total average run times for all tested variants. In tables 9.11, 9.12, 9.13, 9.14 we saw that the strength of branch-and-price are big $\tilde{\delta}$ values. The algorithm is very fast for 2-dimensional such parameters, when run times are compared to the ones of the branch-and-cut algorithm. When k increases, the run times for small $\tilde{\delta}$ values diminish, and increase somewhat for middle and big $\tilde{\delta}$. In comparison to branch-and-cut for most middle, big and very big $\tilde{\delta}$ a huge improvement is achieved. For 3-dimensional $\tilde{\delta}$ parameters, the run times can not reach branch-and-cut run times, but are relatively good. The same characteristics for k and $\tilde{\delta}$ are seen with 3-dimensional $\tilde{\delta}$ parameters.

The comparison of average values for 3-dimensional data was not as easy as for 2-dimensional data since some test instances were not calculated (mostly from $(80, 80, 80)^\top$, for smaller $\tilde{\delta}$ almost no instance is missing) either because of a memory overflow or because of surpassing our pricing iterations or time limit. For a more profound analysis, these limits as well memory size have to be increased. Another option would be to decrease the size of the segmentation trees in order to find a solution more quickly. The variants that delete the dynamic tree after each pricing iteration would surely need a bigger time limit to complete all test instance runs. When regarding the amount of calculated instances the dynamic tree pricing algorithm 'BP/dyn/nlc/fark/(dom)' is the best. Big 3-dimensional $\tilde{\delta}$ were only calculable with this algorithm.

As for node-label constraints, when added to the problem, a huge run time improvement can be noted for all $\tilde{\delta}$ in combination with all k, except $k = 20, \tilde{\delta} = (20, 20)^\top$. The improvement

increases with k and $\tilde{\delta}$. Node-label constraints are advantageous with big k and $\tilde{\delta}$ parameters, where the amount of overall labels is very high and the constraints become effective by making the algorithm prefer labels that contribute to form a spanning tree. A improvement in the average amount of branch-and-bound nodes as well priced variables is achieved.
Finally, a note on the LP relaxation: If $k = |V|$, the LP relaxation takes the following form. All variables z_i are reduced to 1 since all nodes in the directed k-MLSA are part of the solution, thus for corresponding k values a speedup can be noted.

Figures 9.6 show the average over all tested parameters and dimensions for two selected pricing algorithm variants. As best pricing alternative was determined the dynamic tree pricing algorithm that uses node-label constraints. Using additionally Farkas pricing is beneficial in most cases for most parameter configurations. For instances with a great number of possible template arcs these options are likely to produce good results. Allow the pricing of dominated template arcs has relatively small run time averages.

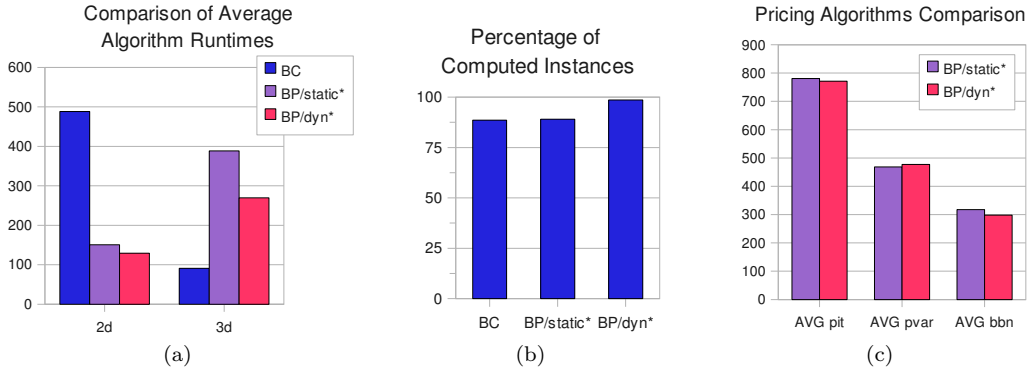


Figure 9.6: Average run times, completed instances, pricing data for 'BC' and the best branch-and-price variants 'BP/static/nlc/fark' (abbreviated by 'BP/static*') and 'BP/dyn/nlc/fark' (abbreviated by 'BP/dyn*'). Run time averages over $\tilde{\delta}$, k.

9.4 Percentage of Priced Variables

As branch-and-price should eliminate the issues of branch-and-cut with big sets T^c, the amount of priced variables confronted to the size of T^c is of interest. Optimally, branch-and-price needs to price less variables than the set of non-dominated template arcs T^c has. Table 9.16 shows a comparison of the average amount of candidate template arcs T^c (row '$|T^c|$') compared to the average amount of priced variables (listed in columns 'pvar'). The columns '%' display the relation in percents. The average over each k separately are shown in column 'AVG', the overall averages over all k in column 'AVGk'.

When comparing the size of T^c for 2– and 3-dimensional $\tilde{\delta}$ we see that for 3-dimensional $\tilde{\delta}$ the amount of non-dominated template arcs is relatively smaller than for 2-dimensional $\tilde{\delta}$, since in the latter case less points may lie in the $\tilde{\delta}$-sized bounding box of a template arc. This is the cause, why branch-and-cut performs better on 3-dimensional $\tilde{\delta}$ than branch-and-price. The best variant when regarding the average number of priced variables is 'BP/dyn/deltree/nlc/fark', but the average run times of this version are not of the best. Fewest variables priced by a dynamic tree pricing algorithm are result for version 'BP/dyn/nlc/fark', but its average amount of priced variables is outperformed by the static version 'BP/static/nlc/fark'.

Table 9.16: A comparison for the average size of set T^c, the set of non-dominated template arcs, with the average amount of priced variables 'pvar' for each tested variant.

		$(10,10)^\top$		$(20,20)^\top$		$(30,30)^\top$		$(40,40)^\top$		$(80,80)^\top$		$(10,10,10)^\top$		$(30,30,30)^\top$		$(40,40,40)^\top$		$(80,80,80)^\top$		AVG		
$\|T^c\|$		798		1777		3440		7072		17937		791		889		1916		4314		4326		AVGk
algorithm	k	pvar	%	pvar	%	pvar	%	pvar	%	pvar	%	pvar	%	pvar	%	pvar	%	pvar	%	AVG	%	over k
BP/static	10	491	61.5	960	54.0	186	5.4	60	0.8	3	0.0	110	14.0	593	66.7	904	47.2	-	-	413	9.6	678 15.7
	20	307	38.5	645	36.3	1217	35.4	844	11.9	12	0.1	115	14.5	878	98.8	1866	97.4	-	-	735	17.0	
	30	260	32.6	600	33.8	1168	34.0	1671	23.6	277	1.5	67	8.5	947	106.6	2098	109.5	-	-	886	20.5	
BP/static/fark	10	499	62.6	968	54.4	138	4.0	32	0.5	2	0.0	137	17.3	720	81.0	774	40.4	-	-	409	9.4	592 13.7
	20	299	37.5	368	20.7	1118	32.5	967	13.7	8	0.0	122	15.4	887	99.8	1352	70.6	-	-	640	14.8	
	30	252	31.6	654	36.8	856	24.9	1369	19.4	199	1.1	90	11.3	879	98.9	1517	79.2	-	-	727	16.8	
BP/static/dom	10	539	67.6	1004	56.5	257	7.5	4	0.1	2	0.0	108	13.7	521	58.6	1057	55.1	-	-	436	10.1	646 14.9
	20	329	41.2	683	38.4	1208	35.1	955	13.5	14	0.1	107	13.6	870	97.9	2114	110.3	-	-	785	18.1	
	30	213	26.7	662	37.3	821	23.9	1570	22.2	137	0.8	71	9.0	840	94.6	1420	74.1	-	-	717	16.6	
BP/static/nlc	10	471	59.0	1009	56.8	99	2.9	7	0.1	2	0.0	124	15.7	464	52.2	1248	65.1	-	-	428	9.9	488 11.3
	20	258	32.3	935	52.6	877	25.5	473	6.7	2	0.0	100	12.7	830	93.4	1741	90.9	-	-	652	15.1	
	30	143	17.9	444	25.0	391	11.4	379	5.4	7	0.0	63	8.0	407	45.8	1240	64.7	-	-	384	8.9	
BP/static/nlc/fark	10	548	68.7	915	51.5	113	3.3	5	0.1	1	0.0	126	16.0	455	51.2	1432	74.7	-	-	450	10.4	426 9.8
	20	274	34.4	633	35.6	941	27.3	162	2.3	1	0.0	120	15.1	812	91.3	1206	63.0	-	-	519	12.0	
	30	170	21.4	284	16.0	336	9.8	97	1.4	1	0.0	75	9.5	453	50.9	1061	55.4	-	-	310	7.2	
BP/static/nlc/dom	10	543	68.0	1031	58.0	230	6.7	6	0.1	2	0.0	119	15.1	505	56.8	1091	56.9	-	-	441	10.2	507 11.7
	20	280	35.1	591	33.2	959	27.9	492	7.0	2	0.0	106	13.4	896	100.8	1990	103.9	-	-	665	15.4	
	30	169	21.2	520	29.3	403	11.7	244	3.4	37	0.2	60	7.6	532	59.9	1359	70.9	-	-	415	9.6	
BP/static/nlc/cont	10	582	73.0	1226	69.0	693	20.2	257	3.6	52	0.3	153	19.3	702	79.0	1838	96.0	-	-	688	15.9	631 14.6
	20	353	44.3	571	32.1	1268	36.8	536	7.6	98	0.5	105	13.3	888	99.9	2747	143.4	-	-	821	19.0	
	30	207	25.9	391	22.0	268	7.8	146	2.1	396	2.2	73	9.2	724	81.5	859	44.8	-	-	383	8.9	
BP/dyn	10	508	63.6	1116	62.8	131	3.8	15	0.2	3	0.0	114	14.4	570	64.1	975	50.9	8	0.2	382	8.8	826 19.1
	20	315	39.5	585	32.9	1333	38.8	910	12.9	51	0.3	102	12.9	917	103.2	1761	91.9	2274	52.7	916	21.2	
	30	237	29.7	688	38.7	984	28.6	1577	22.3	126	0.7	66	8.3	871	98.0	1974	103.0	4086	94.7	1179	27.2	
BP/dyn/nlc	10	461	57.8	996	56.1	139	4.0	7	0.1	2	0.0	128	16.2	503	56.6	1060	55.3	5	0.1	367	8.5	469 10.8
	20	284	35.6	806	45.3	930	27.0	572	8.1	2	0.0	96	12.1	918	103.3	2245	117.2	771	17.9	736	17.0	
	30	162	20.4	313	17.6	278	8.1	210	3.0	21	0.1	63	7.9	477	53.7	976	50.9	224	5.2	303	7.0	
BP/dyn/nlc/fark	10	511	64.1	999	56.2	76	2.2	5	0.1	1	0.0	125	15.8	524	58.9	1481	77.3	4	0.1	414	9.6	456 10.5
	20	235	29.4	467	26.3	841	24.4	235	3.3	9	0.0	115	14.5	772	86.9	1819	94.9	775	18.0	585	13.5	
	30	156	19.6	396	22.3	182	5.3	119	1.7	1	0.0	70	8.9	456	51.3	985	51.4	946	21.9	368	8.5	
BP/dyn/nlc/fark/dom	10	496	62.2	1040	58.5	100	2.9	5	0.1	1	0.0	129	16.3	593	66.8	1248	65.1	11	0.3	403	9.3	463 10.7
	20	254	31.9	571	32.1	767	22.3	240	3.4	2	0.0	122	15.5	1113	125.3	1855	96.8	135	3.1	562	13.0	
	30	168	21.0	296	16.7	216	6.3	152	2.1	20	0.1	80	10.1	397	44.6	1251	65.3	1249	29.0	425	9.8	
BP/dyn/nlc/cont	10	645	80.8	1185	66.7	773	22.5	158	2.2	474	2.6	148	18.7	751	84.6	1756	91.7	496	11.5	710	16.4	564 13.0
	20	311	39.0	661	37.2	1104	32.1	201	2.8	3	0.0	114	14.4	841	94.6	2318	121.0	432	10.0	665	15.4	
	30	217	27.2	439	24.7	177	5.2	183	2.6	13	0.1	70	8.9	576	64.8	982	51.3	194	4.5	317	7.3	
BP/dyn/del/nlc/fark	10	539	67.6	892	50.2	47	1.4	5	0.1	1	0.0	131	16.5	497	55.9	1144	59.7	4	0.1	362	8.4	383 8.9
	20	291	36.5	740	41.6	547	15.9	155	2.2	2	0.0	108	13.6	713	80.2	1466	76.5	571	13.2	510	11.8	
	30	179	22.5	236	13.3	273	7.9	106	1.5	1	0.0	74	9.3	329	37.0	1069	55.8	228	5.3	277	6.4	

9.5 Branch-and-Price Results for NIST Data

Until now, instances from the nist data set were not solvable with branch-and-cut, since the determination of the set T^c has very high run times. We attempted to solve the nist instances with the dynamic tree pricing algorithm, as we determined it previously to be the fastest and because of the lesser memory requirements. The most promising option is to use node-label constraints with the according pricing (section 5.5) and a starting solution determined with Farkas pricing. As these instances have a greater amount of data points and bigger sets T^c, we require an adjusted time limit of 86400 seconds (24 hours). Pricing iterations limit has been removed. For nist test runs the $\tilde{\delta}$ parameters have to be selected carefully. For small $\tilde{\delta}$, at the moment no significant parameter combinations could be found, but as the fhg tests showed, small and middle $\tilde{\delta}$ are solved by branch-and-price with relatively high run times. Experimental tests for the nist test set included $k = 10, 20, 40, 80$ and $\tilde{\delta} = (80, 80)^\top$, $(120, 120)^\top$, $(150, 150)^\top$, $(200, 200)^\top$, $(300, 300)^\top$ values. As the average domain for nist-data is inhomogeneous ($\tilde{v}_{\text{avg}} = (3993, 3368, 359, 2)^\top$) only 2-dimensional parameter settings have been evaluated. Many tested parameters may be unsuited for compression and were selected in order to show which settings are solvable and which not. Table 9.17 shows average run times ('alg[s]'), total run times ('tot[s]'), pricing iterations ('pit'), priced variables ('pvar') and branch-and-bound nodes ('bbn'). Each test run included selected 15 nist instance files, as described in section 9.1. We tested branch-and-price versions 'BP/dyn/nlc/fark' and 'BP/dyn/del/nlc/fark'. Averages are listed for instances where at least 26% of instances could be computed.

Table 9.17: Branch-and-price results for nist data and varying k, $\tilde{\delta}$. Columns 'alg[s]', 'tot[s]', 'pit', 'pvar' and 'bbn' list average algorithm run time and average total run time (including Farkas priced starting solution) in seconds, average amount of pricing iterations, priced variables, branch-and-bound nodes.

		$(80,80)^\top$					$(120,120)^\top$				
options	k	alg[s]	tot[s]	pit	pvar	bbn	alg[s]	tot[s]	pit	pvar	bbn
	10	13764.6	13764.7	2191	2074	119		n/a			
BP/dyn/	20		> 1 day					> 1 day			
nlc/fark	40		> 1 day					> 1 day			
	80		> 1 day					n/a			
	10	47187.4	47187.5	4712	4557	157	16204.3	16204.4	3456	3285	172
BP/dyn/del/	20		> 1 day					> 1 week			
nlc/fark	40		> 1 day					> 1 day			
	80		> 1 day					n/a			
		$(150,150)^\top$					$(200,200)^\top$				
options	k	alg[s]	tot[s]	pit	pvar	bbn	alg[s]	tot[s]	pit	pvar	bbn
	10	1181.6	1181.6	327	315	13	710.7	710.8	74	70	5
BP/dyn/	20	237.4	237.5	12	12	1	7698.0	7699.1	1590	1569	21
nlc/fark	40		> 1 day				366.4	366.5	12	12	1
	80		> 1 day				517.5	517.6	42	40	3
	10	1147.6	1147.7	481	469	14	246.1	246.1	9	9	1
BP/dyn/del	20	13789.5	13789.6	3721	3666	56	5327.9	5327.9	734	708	27
nlc/fark	40		> 1 day				382.2	382.3	11	11	1
	80		> 1 day				3469.7	3469.8	981	961	21
		$(300,300)^\top$									
options	k	alg[s]	tot[s]	pit	pvar	bbn					
	10	114.8	114.9	5	5	1					
BP/dyn/	20	440.0	440.1	35	29	7					
nlc/fark	40	185.5	185.6	12	12	1					
	80	589.2	589.3	162	158	5					
	10	130.4	130.5	5	5	1					
BP/dyn/del/	20	371.8	371.9	10	8	3					
nlc/fark	40	894.2	894.3	96	94	3					
	80	1286.5	1286.6	184	177	8					

9.5.1 Results

For very big $\tilde{\delta} = (150, 150)^\top$, $(200, 200)^\top$, $(300, 300)^\top$ almost all instances were solved relatively fast and are computed within some hundred seconds. Small $k = 10, 20$, relevant to compression, have good results. The bigger k and the smaller $\tilde{\delta}$, the less instances could be completed within the time limit. Instances with very many data points are amongst the solved instances and the size of $|V|$ does not influence solvability.

Generally, the tailing off effect becomes extremely visible for this data set. For instances, where after 24 hours no result could be determined, no result will be computed even after 1 week. For k and $\tilde{\delta}$, the behaviour of branch-and-price is very similar to the one determined with fhg-data. For big $\tilde{\delta} = (150, 150)^\top$, $(200, 200)^\top$, $(300, 300)^\top$ test parameters, the absolute compression rates range between 91–96% because the codebook size for such parameters is one or two. Combined with small k almost all instances could be computed. The bigger k and the smaller $\tilde{\delta}$, the more instances exceed the time limit.
We expect that smaller $\tilde{\delta}$, more suited for compression, become solvable with fine tuning of the `SCIP` framework, for example by enabling a setting with an emphasis on feasibility. The tested parameters cover solution possibilities of branch-and-price with nist data and indicates the magnitude of the resulting run times.

9.6 Branch-and-Price Summary

All developed static and dynamic segmentation tree algorithms UB-driven traversing, advanced bounding traversal, best first search, are a massive improvement for the branch-and-cut preprocessing strep. All variants greatly outperform the preprocessing from [ChwRai09]. The run times for determining T^c with the best variant UB-driven traversing are reduced to 1.91% of the previous preprocessing, and is considered to be very beneficial, despite a high memory usage. An improvement for this memory usage would be not to save the upper and lower bound set at each node, to either leave out equal bound sets for predecessor and successor nodes or to save only the differences between predecessor and successor.

All presented run times for branch-and-price are the total time needed by `SCIP` for determining the solution and determine that no better solution can be found in the branch-and-bound decision tree. It lies in the nature of our problem, that more than one combination of template arcs may be an optimal solution. As the possibilities for combinations are high we have as consequence large plateaus of equally good solutions. The times needed by `SCIP` to find the first best solutions may be relatively small, the most of time is needed to determine that the actual found solution is minimal. `SCIP` may find many more optimal solution possibilities in the branching and bounding process. If the result codebook size is already one, which is often the case with big $\tilde{\delta}$, there is no need to further determine optimality and the branch-and-price process can be terminated earlier.

The best pricing algorithm was determined to be the dynamic tree pricing algorithm. Best run time results have been achieved with this pricing algorithm when combined with node-label constraints and the according pricing as well with Farkas pricing. For this, the average amount of branch-and-bound nodes was smallest as well as the average number of priced variables.
As for branch-and-cut run times, the time for searching T^c must be added, we added the fast UB-driven traversing times. Even when regarding the new, better preprocessing times for branch-and-cut, there are parameter combinations for which branch-and-price outperforms branch-and-cut, despite the directed cut model used by branch-and-cut has the better LP relaxation.
As the Farkas pricing algorithm showed beneficial effects when compared to the star shaped spanning tree starting solution, we expect that branch-and-price would receive a speed up, if an even better starting solution determined with some heuristic performs better.

Another characteristic for our problem is that the solution is ambiguous. Thus, much needless pricing is done. A further consequence of this ambiguity were the template arcs duplicates, which we reduced by not pricing same template arcs again. For branch-and-price we saw that the more the correction vector domain $\tilde{\delta}$ increases, the more the optimal solution becomes unambiguous and is found very fast. For such big $\tilde{\delta}$ values, branch-and-price greatly outperforms branch-and-cut and the run times for big $\tilde{\delta}$ are good and we can calculate the solution for an fhg instance in a couple of seconds. Good results have been determined with small and big k. When $\tilde{\delta}$ and k have middle values, branch-and-price performs not so well.

The disadvantage when pricing non-dominated template arcs is again ambiguous optimum in the tree, searched by regarding the sums of dual values. Many branches having sums of dual values equal to our actual found value have to be searched, since one element in such a branch may dominate the actual one. As one or more dual values in such a template arc defining bound set may be 0, which is the case very often, dominated and non-dominated bound sets have often the same sum of dual values and must be checked for dominance. This may occur also for differing bound sets, so we find often sets with equal sums of dual values so we can not unambiguously decide which one to price. Much time for searching would be saved if these sums of dual values would be unambiguous. Branch-and-price would be extremely faster with an unique template arc for each pricing iteration. The effect is the main slowing cause for small and middle $\tilde{\delta}$: Here many template arcs have same sums of dual values and many tree branches must be searched. If $\tilde{\delta}$ increases, the more the template arcs have unambiguous sums of dual values.

The NIST test set, which was not calculable with branch-and-cut, becomes now computable for selected parameter values with branch-and-price and a dynamic tree pricing algorithm. Although most working parameters may be inappropriate for compression, this is a strong indication that a branch-and-cut-and-price algorithm may be able to solve these instances.

When regarding the application results, the codebooks with a small amount of template arcs are favorable, since thus higher compression rates are result. This implies big $\tilde{\delta}$ and thus big T^c. If $|T^c|$ is big, branch-and-cut performs not so well and is outperformed by branch-and-price, since it effectively bypasses the need for this preprocessed set T^c. Thus, with small $\tilde{\delta}$, branch-and-price works rather not so good as for big $\tilde{\delta}$ it performs very well. Table 5 in the appendix lists an overview of the minimal codebooks.

Chapter 10

Conclusions

This thesis investigates compression of fingerprint minutiae data for watermarking. It represents a continuation of the work of [ChwRai09] who analyzed various algorithms, both approximating and exact, for compressing such fingerprint templates.
In general a fingerprint template is an unordered set of d-dimensional data points. All approaches based on an encoding of k of these data points as a directed spanning tree for the modeling of which [ChwRai09] developed the k-MLSA. By means of this arborescence, in the end a minimal set of labels or template arcs is extracted, which are used in combination with a correction vector to encode the data points in order to achieve compression. As exact method [ChwRai09] analyzed branch-and-cut. The k-MLSA approach outperforms several well known compression techniques on the tested data sets and for reasonably large values of k ($k \geq 20$) (in order to keep small the false non-match-rates) the authors achieve good average absolute compression ratios. So, this method is suitable for watermarking fingerprint minutiae with its compressed data. Main drawback of all developed methods was the need for a run time intensive preprocessing step for the determination of candidate template arcs.

In this thesis we analyzed another exact algorithm and developed branch-and-price for solving the k-MLSA. As in branch-and-price we start with a small MIP and add variables or columns on demand. For this approach, no preprocessing is needed anymore. Yet, with branch-and-price arises the pricing problem, the solution of which was one of the main topics. We developed a static as well as a dynamic k-d tree based segmentation algorithm. At each node we associate an upper and lower bound with the help of which we build and extract the solution to the pricing problem efficiently, by searching the maximal sum of dual values for each such bound set.

By developing these segmentation algorithms an efficient alternative to the preprocessing by [ChwRai09], which determines also the candidate template arcs set, was discovered. The bound sets at each node are vital for this approach and with these bounds we conduct the search process for searching T^c, the set of non-dominated candidate template arcs. If an element from this set is found, in the tree represented by a non-dominated bound set, we extract the according template arc from it and save it. We developed three traversing strategies for the static tree (UB-driven traversing, advanced bounding traversal, best first search) and one for the dynamic tree (dynamic UB-driven traversing). All variants determined the correct set T^c. The best variant for searching the candidate template arcs set resulted to be a static segmentation tree and using UB-driven traversing. This algorithm takes an average of 1.91% of the time needed by the previous preprocessing step and speeds up the process with a factor of 50. As all developed variants are very fast in comparison to the preprocessing, both static and dynamic tree, combined with an arbitrary traversing strategy, are a great improvement for the preprocessing step. The only disadvantage for the segmentation trees is a high memory usage for the upper and lower bound sets. Very nice visualizations can be extracted from the segmentation trees, some are presented in the appendix.

Next, for solving branch-and-price, the k-MLSA was modeled as a flow network. The thesis presents a single commodity flow and a multi commodity flow formulation. Solvability and efficiency of these two k-MLSA formulations have been investigated. Branch-and-price has been realized by employing a non-commercial framework for solving constraint integer problems, namely `SCIP` in combination with the commercial state of the art solver `ILOG CPLEX`.

For both models, a static tree pricing algorithm and a dynamic tree pricing algorithm have been developed. The behaviour of both pricing algorithms can be directed by using multiple options. All options have been tested extensively. The multi commodity flow formulation resulted to be not competitive to the single commodity flow formulation.

When regarding the new preprocessing times for branch-and-cut, determined now with UB-driven traversing, for many parameter configurations branch-and-price outperforms branch-and-cut, despite for the latter a directed cut model was used which has the better LP relaxation. Our problem has the property, that more than one combination of template arcs may be an optimal solution. As the possibilities for combinations are high, large plateaus of equally good solutions are consequence. Another characteristic for our problem is that the solution is ambiguous and much pricing is done needlessly.

Generally branch-and-price performs best when the correction vector domain $\tilde{\delta}$ is big. When using such big $\tilde{\delta} = (40, 40)^\top$,$(80, 80)^\top$ values ($\frac{1}{4}$ - $\frac{1}{10}$ of domain border) the amount of optimal solutions becomes small and branch-and-price greatly outperforms branch-and-cut. The run time averages are very low and we can calculate the solution for a smaller fhg-instance (from [Fraun]) in a couple of seconds. Good results were determined with small and big k. When $\tilde{\delta}$ and k have middle values of 20, branch-and-price can not reach branch-and-cut run times. For 3-dimensional $\tilde{\delta}$, branch-and-cut performed extremely good and with branch-and-price run times twice as high could be achieved.

As best pricing algorithm the dynamic tree pricing algorithm was determined. For this pricing algorithm the average run times for tested instances were smallest. Best results were determined with this algorithm when the corresponding single commodity flow formulation employs additionally node-label constraints, and the pricing algorithm uses the according pricing strategy. Here, the average amount of branch-and-bound nodes was smallest as well as the average number of priced variables. When Farkas pricing is used for determining a good starting solution the run times may improve. The main disadvantage when pricing non-dominated template arcs is again the ambiguity for the optimum template arcs, determined based on the sums of dual values. As one or more template arc's dual values may be 0, which occurs very often, the according non-dominated template arc encoding bound set may have an equal sum of dual values as a dominated bound set sum and we have to check for dominance. This may occur also for differing bound sets. As we may have many sets in the segmentation tree with equal sums of dual values, we may not unambiguously decide which one to price. Branch-and-price would be extremely faster with an unique template arc for each pricing iteration. The effect is the main slowing factor for small and middle $\tilde{\delta}$: Here many template arcs have same sums of dual values and many tree branches must be searched. If $\tilde{\delta}$ increases, the more the template arcs have unambiguous sums of dual values. Nonetheless, when allowing dominated template arcs to be priced, branch-and-price performs well also and computes a correct codebook.

The nist-testset (from [NIST]) was not computable with branch-and-cut because of a higher amount of possible template arcs. Branch-and-price performs good, but only for well-chosen $\tilde{\delta}$ values. Now, with such carefully selected parameters we can compute solutions for these instances.

For test runs, were branch-and-cut data can be computed, the following results are summarized. When comparing the amount of priced variables to the set of non-dominated candidate template arcs, branch-and-price performs very good and prices an amount being only 12.8% (average over all

tested fhg-instances and parameters) of the size determined by preprocessing. When considering the time limit set branch-and-cut completes 88.6% of the test instances, whereas with the best dynamic tree pricing algorithm we can compute 98.6% . Branch-and-price performs very good for most model parameters and the run time averages are smaller than with branch-and-cut.

10.1 Further Work

As future work remains branch-and-cut-and-price. It has to be evaluated, if such an approach performs quicker and faster and more efficiently that branch-and-price.
As for the segmentation trees the memory consumption is very high, especially when using the static variant in combination with very big $\tilde{\delta}$, a further improvement would be to reduce the size of the segmentation trees, by building only non-dominated branches. Another possibility for reducing segmentation tree size would be not to save upper and lower bound set at each node, by either interleave sets that do not change from predecessor node to successor nodes or to save only the difference to the predecessor node.

Appendix

Glossary, Often Used Terms and Notations

Common Terms in Optimization Theory	
LP	Linear program
ILP	Integer linear program
IP	Integer program
BIP	Binary integer program
MIP	Mixed integer program
COP	Combinatorial optimization problem
SEP	Separation problem
PP	Pricing problem
CF	Compact formulation
EF	Extended formulation
MP	Master program
RMP	Restricted master program
P	Problem
P'	Relaxation of P
z^*	Optimal solution, best possible solution
$z^*_{P'}$	Optimal objective function value of P
BB	Branch-and-bound
BC	Branch-and-cut
BP	Branch-and-price
BCP	Branch-and-cut-and-price
Terms Specific to this Thesis	
k-MLSA	k-Node Minimum Label Spanning Tree
MLST	Minimum Label Spanning Tree
MVCA	Maximum Vertex Covering Algorithm
k-CT	k-Cardinality
SCF	Single commodity flow formulation
MCF	Multi commodity flow formulation
PP	Preprocessing
SEG	Segmentation tree
UBD	UB-driven traversing
ABT	Advanced bounding traversal
BFS	Best first search
DFS	Depth first search

Additional Experimental Results Tables

The following pages list additional tables for the experimental results determined to be too big for including in the chapter. The tables are described in chapter 9, sections 9.2 and 9.3.

Table 1: Two dimensional input parameter $\tilde{\delta} = (10,10)^\top, (20,20)^\top, (30,30)^\top$. Run times on `G1` and percentages of visited nodes for the static segmentation for all three traversing variants UBD (UB-Driven Traversing), ABT (Advanced Bounding Traversal) and BFS (Best First Search). The percentage '%n' indicates visited nodes w.r.t. the number of total nodes '$|n|$' in percent. Column 'dyn' lists number and percentage of created nodes (w.r.t. the static variant) for the dynamic segmentation.

2d	$\tilde{\delta} = (10,10)^\top$								$\tilde{\delta} = (20,20)^\top$								$\tilde{\delta} = (30,30)^\top$							
	static						dyn		static						dyn		static						dyn	
file	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n
ft-01	31213	0.05	40	0.14	31	0.13	74	21081	68	63369	0.27	43	0.74	33	0.60	76	43577	69	98621	0.87	48	2.27	36	1.60	75	73105	74
ft-02	25605	0.03	40	0.07	31	0.08	74	16985	66	50925	0.17	43	0.46	33	0.37	76	35441	70	73377	0.51	48	1.39	37	0.98	75	54553	74
ft-03	42189	0.10	41	0.25	31	0.22	74	27737	66	81721	0.59	46	1.73	36	1.31	76	57881	71	127037	1.71	47	4.73	36	3.56	75	92013	72
ft-04	11657	0.01	41	0.01	32	0.01	71	7217	62	25221	0.04	40	0.08	28	0.07	76	16157	64	37665	0.12	47	0.24	33	0.19	76	26381	70
ft-05	52301	0.15	42	0.42	33	0.37	75	34029	65	105153	0.71	44	1.89	35	1.48	76	73121	70	151165	2.47	49	5.97	40	3.89	75	113089	75
ft-06	4945	0.01	41	0.01	31	0.00	70	3505	71	9665	0.01	40	0.02	30	0.02	76	6269	65	17033	0.02	43	0.04	28	0.03	76	10921	64
ft-07	24513	0.03	39	0.06	29	0.05	72	16237	66	49209	0.12	39	0.32	29	0.27	76	31953	65	76073	0.43	43	1.06	33	0.79	75	52277	69
ft-08	21529	0.03	39	0.06	31	0.05	73	14777	69	45837	0.11	39	0.29	29	0.25	76	29317	64	71185	0.31	43	0.80	32	0.63	75	48925	69
ft-09	22121	0.03	38	0.06	30	0.04	73	14353	65	44925	0.11	40	0.29	30	0.25	76	28525	63	70661	0.30	43	0.78	32	0.61	75	47897	68
ft-10	30077	0.04	41	0.11	31	0.11	74	20165	67	63645	0.25	41	0.67	32	0.52	76	43385	68	95041	0.70	45	1.78	34	1.32	75	68325	72
ft-11	52685	0.18	42	0.51	33	0.43	76	36509	69	102313	0.97	46	2.70	37	1.86	75	72801	71	136185	2.99	52	7.56	43	4.34	75	110061	81
ft-12	27141	0.05	40	0.10	31	0.09	75	17609	65	53173	0.22	43	0.57	33	0.42	76	36513	69	81533	0.80	49	1.98	37	1.30	75	61561	76
ft-13	20745	0.03	41	0.06	31	0.05	74	13141	63	39849	0.11	43	0.28	32	0.22	76	27061	68	60093	0.38	48	0.91	37	0.62	75	45109	75
ft-14	36685	0.09	42	0.23	33	0.20	75	24925	68	74365	0.41	44	1.12	35	0.85	76	51673	69	115717	1.28	48	3.43	37	2.30	75	86261	75
ft-15	28477	0.05	41	0.12	31	0.10	75	19337	68	58877	0.26	44	0.67	33	0.51	76	41441	70	87665	0.90	50	2.06	37	1.37	75	66793	76
ft-16	44805	0.11	41	0.28	32	0.25	74	29941	67	93809	0.54	41	1.50	32	1.24	76	62925	67	137437	1.82	46	4.55	36	3.24	75	99229	72
ft-17	28969	0.05	41	0.12	32	0.11	74	19705	68	62673	0.23	41	0.66	31	0.54	76	41885	67	95305	0.85	47	2.17	37	1.54	75	68657	72
ft-18	52713	0.15	40	0.38	31	0.34	74	35829	68	104741	0.66	41	1.96	32	1.57	76	70741	68	154285	2.33	47	6.40	37	4.36	75	113965	74
ft-19	39881	0.09	39	0.20	31	0.20	74	26353	66	81285	0.38	42	1.11	32	0.92	76	53565	66	121649	1.11	44	3.04	34	2.27	75	83949	69
ft-20	22861	0.02	39	0.06	31	0.07	73	16853	74	45257	0.13	41	0.33	31	0.27	76	30965	68	73357	0.31	41	0.71	29	0.62	76	49425	67
AVG		0.07	40.4	0.16	31.3	0.15	73.7		67.0		0.31	42.1	0.87	32.2	0.68	76.0		67.6		1.01	46.4	2.59	35.3	1.78	75.2		72.2
stdev		0.05		0.14		0.12					0.26		0.73		0.54					0.84		2.18		1.39			

2d	$\tilde{\delta} = (10,10)^\top$								$\tilde{\delta} = (20,20)^\top$								$\tilde{\delta} = (30,30)^\top$							
	static						dyn		static						dyn		static						dyn	
file	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n
nist-g-01	160365	1.88	40	4.43	38	2.86	65	109781	68	274249	2.67	39	6.79	31	6.78	70	171269	62	389885	4.70	38	14.22	29	14.53	74	235189	60
nist-g-02	142449	1.93	42	4.36	40	2.56	65	106129	75	242069	2.40	38	5.98	32	5.88	69	160809	66	348137	3.99	37	11.95	29	12.40	73	221985	64
nist-g-03	160197	1.99	41	4.62	38	3.13	66	119377	75	276197	3.18	40	8.47	32	8.14	72	184457	67	416693	6.76	39	23.14	31	21.15	75	266437	64
nist-g-04	236541	4.00	41	9.61	38	6.49	65	158257	67	410569	6.47	39	17.66	31	16.82	71	248729	61	600757	13.04	38	44.21	30	40.85	75	354777	59
nist-g-05	92769	0.78	42	1.79	40	1.04	64	65225	70	151841	0.99	38	2.41	32	2.27	68	95909	63	218017	1.58	37	4.32	29	4.46	72	132225	61
nist-b-01	189017	2.45	40	5.77	37	3.93	65	128545	68	324617	3.84	39	10.16	31	9.97	71	203181	63	475089	7.70	38	25.46	30	24.37	75	288713	61
nist-b-02	141081	1.51	42	3.52	38	2.46	66	97397	69	246209	2.71	40	7.27	32	6.91	72	157333	64	363493	5.71	39	19.28	31	17.54	75	225269	62
nist-b-03	186445	2.53	41	5.95	38	4.09	65	125609	67	324005	4.14	39	10.91	32	10.81	71	200237	62	471885	8.13	38	26.49	30	26.41	75	283477	60
nist-b-04	94141	0.82	42	1.89	40	1.07	64	67905	72	159577	1.06	39	2.57	32	2.44	68	100497	63	222649	1.64	38	4.44	29	4.64	72	135245	61
nist-b-05	139321	1.44	41	3.43	38	2.25	65	96429	69	238929	2.19	39	5.62	31	5.55	70	152149	64	353109	4.04	37	12.84	29	13.06	74	214933	61
nist-u-01	147717	1.81	41	4.19	39	2.61	65	105829	72	248353	2.40	38	6.06	31	6.01	70	163721	66	366413	4.39	37	13.35	29	13.96	74	227381	62
nist-u-02	134701	1.42	41	3.34	39	2.10	65	91189	68	228693	2.04	39	5.22	32	5.12	70	142229	62	331589	3.71	37	11.43	30	11.96	74	199601	60
nist-u-03	159361	1.92	42	4.44	38	3.10	66	110945	70	279549	3.31	40	8.86	32	8.54	72	177509	63	411629	6.76	38	23.01	31	21.37	75	253033	61
nist-u-04	106761	0.96	41	2.25	39	1.34	64	75465	71	178597	1.24	38	3.05	32	2.99	69	110989	62	246993	1.86	36	5.34	29	5.85	73	148685	60
nist-u-05	75521	0.52	41	1.17	40	0.68	64	53469	71	126413	0.65	38	1.56	32	1.52	68	79785	63	182997	1.05	37	2.88	29	3.06	72	107937	59
AVG		1.73	41.2	4.05	38.7	2.65	64.9		70.1		2.62	38.9	6.84	31.7	6.65	70.1		63.4		5.00	37.6	16.16	29.7	15.71	73.9		61.0
stdev		0.87		2.08		1.47					1.49		4.12		3.95					3.17		11.08		10.10			

Table 2: Two dimensional input parameter $\tilde{\delta} = (40,40)^\top, (80,80)^\top, (120,120)^\top$. Run times on `G1` and percentages of visited nodes for the static segmentation for all three traversing variants UBD (UB-Driven Traversing), ABT (Advanced Bounding Traversal) and BFS (Best First Search). The percentage '%n' indicates visited nodes w.r.t. the number of total nodes '$|n|$' in percent. Column 'dyn' lists number and percentage of created nodes (w.r.t. the static variant) for the dynamic segmentation.

2d	$\tilde{\delta} = (40,40)^\top$ static									$\tilde{\delta} = (80,80)^\top$ static								
file	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	dyn $\|n\|$	dyn %n	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	dyn $\|n\|$	dyn %n
ft-01	127154	1.88	52	4.89	39	3.12	75	100357	79	193957	21.80	65	38.00	50	24.46	75	178101	92
ft-02	98109	1.28	50	3.40	39	2.27	75	77489	79	175781	13.54	64	29.07	46	15.99	75	161345	92
ft-03	159949	4.36	51	11.32	41	7.95	75	121741	76	251817	68.17	65	117.39	52	76.06	75	228077	91
ft-04	51753	0.26	49	0.53	34	0.40	75	36969	71	111161	2.32	62	3.88	36	3.34	75	96517	87
ft-05	190025	6.20	54	14.44	43	8.55	75	151441	80	288317	103.74	64	194.96	53	133.17	75	261525	91
ft-06	22205	0.06	48	0.10	32	0.07	75	16093	72				$\delta > \bar{v}/2$					
ft-07	106561	1.12	46	2.62	34	1.93	75	76673	72	205337	18.04	61	36.35	42	18.13	75	179837	88
ft-08	100557	0.87	47	2.11	34	1.53	75	75785	75	188005	11.06	63	22.89	44	12.11	75	168545	90
ft-09	94053	0.84	48	2.11	36	1.45	75	68453	73	182241	10.98	62	23.83	45	12.38	75	161725	89
ft-10	132589	1.94	50	4.51	38	3.13	75	101913	77	222897	37.96	65	61.29	50	36.44	75	204673	92
ft-11	170529	7.07	57	16.36	47	9.15	75	145385	85	217025	123.44	71	192.88	63	135.15	75	211481	97
ft-12	104237	1.86	53	4.20	40	2.61	75	82993	80	152905	30.88	70	51.61	55	43.13	75	147373	96
ft-13	83353	1.03	54	2.34	40	1.44	75	66893	80	143969	8.64	68	14.45	48	9.04	75	134813	94
ft-14	151593	3.12	52	7.33	41	4.54	75	119341	79	215765	64.27	68	93.84	55	57.97	75	203485	94
ft-15	110869	2.22	56	4.84	42	2.83	75	89905	81	160377	38.17	70	58.75	55	28.15	75	155389	97
ft-16	184477	3.85	49	9.17	38	6.12	75	138989	75	284957	91.64	64	141.44	52	95.68	75	258541	91
ft-17	125821	2.10	51	5.00	40	3.18	75	98081	78	221989	36.85	65	51.11	50	34.82	75	206461	93
ft-18	198101	5.22	50	13.59	40	8.49	75	150637	76	316473	122.68	65	221.36	53	149.40	75	291573	92
ft-19	162609	2.70	48	7.00	37	4.77	75	120121	74	267517	47.32	63	79.76	47	50.17	75	241473	90
ft-20	93521	0.80	45	1.75	33	1.34	75	67241	72	188325	7.83	58	15.45	40	10.02	75	159513	85
AVG		2.44	50.5	5.88	38.4	3.74	75.0		76.7		45.23	64.9	76.23	49.3	49.77	75.0		91.5
stdev		1.97		4.76		2.85					39.56		66.84		46.51			

2d	$\tilde{\delta} = (40,40)^\top$ static									$\tilde{\delta} = (80,80)^\top$ static								
file	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	dyn $\|n\|$	dyn %n	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	dyn $\|n\|$	dyn %n
nist-g-01	533937	8.31	37	29.32	29	28.86	75	313965	59	1346441	79.34	35	449.21	29	806.60	75	770861	577
nist-g-02	478181	6.92	36	24.23	28	24.47	76	290709	61	1235317	57.49	34	550.63	28	519.77	76	711357	58
nist-g-03	570581	12.62	38	50.96	31	42.05	76	355629	62	1528281	185.15	36	1330.05	30	1785.51	75	906237	59
nist-g-04	825441	24.37	37	108.72	30	112.80	76	483641	59	2136701	488.56	35	2696.64	30	4085.80	75	1227005	57
nist-g-05	288601	2.61	36	8.35	28	8.62	75	169941	59	715277	14.30	35	73.18	28	63.67	76	404633	57
nist-b-01	650361	14.18	37	55.12	29	49.60	76	385893	59	1678097	225.14	35	1361.97	29	2188.09	75	965741	58
nist-b-02	502805	10.94	38	42.20	31	33.74	76	309701	62	1333249	173.32	36	1072.60	31	1348.94	75	799289	60
nist-b-03	653093	14.90	37	54.29	30	56.35	76	389389	60	1705301	237.95	36	1366.33	29	2312.99	75	990593	58
nist-b-04	293929	2.73	37	8.56	29	8.92	75	173177	59	722033	14.67	34	73.52	28	71.99	76	410441	57
nist-b-05	478973	7.59	36	27.57	29	26.11	76	285373	60	1213301	96.94	35	573.81	29	949.62	75	717701	59
nist-u-01	501461	7.79	36	28.33	28	27.53	76	299661	60	1269141	87.67	34	461.46	28	801.19	75	720125	57
nist-u-02	454561	6.89	37	24.31	29	23.57	76	269569	59	1174905	78.18	35	521.35	29	866.85	75	683709	58
nist-u-03	563313	13.16	38	52.69	31	49.33	76	346189	61	1503109	333.14	36	1256.44	31	2104.42	75	893781	59
nist-u-04	328861	3.15	36	10.42	28	11.08	75	195909	60	822953	31.59	34	109.24	28	142.88	76	465017	57
nist-u-05	245201	1.71	36	5.60	28	5.83	75	141681	58	598581	10.19	34	42.97	27	37.88	76	331329	55
AVG		9.19	36.8	35.38	29.2	33.92	75.7		59.8		140.9	34.9	795.9	28.9	1205.7	75.3		57.7
stdev		6.05		26.82		26.96					135.7		723.7		1124.6			

2d	$\tilde{\delta} = (120,120)^\top$ / static							dyn	
file	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n
nist-g-01	2442949	601.33	35	2775.10	29	[t]		1402625	57
nist-g-02	2277469	465.29	34	2986.23	29	4625.13	75	1314141	58
nist-g-03	2770253	1036.14	36	[t]		[t]		1648553	60
nist-g-04				memory overflow				2205945	
nist-g-05	1313201	98.28	35	618.99	28	1068.14	75	744473	57
nist-b-01	3093905	1198.02	35	[t]		[t]		1781381	58
nist-b-02	2356737	966.37	36	[t]		[t]		1404877	60
nist-b-03	3067877	1330.78^{01}	35	[t]		[t]		1789985	58
nist-b-04	1297557	179.48	35	668.84	28	1198.12	75	740957	57
nist-b-05	2253093	757.66	35	[t]		[t]		1327061	59
nist-u-01	2363901	578.95	34	[t]		[t]		1338369	57
nist-u-02	2139293	778.36	35	[t]		4612.82	75	1259401	59
nist-u-03	2680809	1736.33^{01}	35	[t]		[t]		1586813	59
nist-u-04	1510005	190.28	34	572.58	28	1673.20	75	855589	57
nist-u-05	1081601	57.79	34	423.82	27	576.95	75	610097	56
AVG		712.50	34.8	1340.93	28.2	2292.39	75.0		57.9
stdev		386.20		1197.35		1835.67			

Table 3: Three dimensional input parameter $\tilde{\delta} = (10,10,10)^\top, (20,20,20)^\top, (30,30,30)^\top$. Run times on `G1` and percentages of visited nodes for the static segmentation for all three traversing variants UBD (*UB*-Driven Traversing), ABT (Advanced Bounding Traversal) and BFS (Best First Search). The percentage '%n' indicates visited nodes w.r.t. the number of total nodes '$|n|$' in percent. Column 'dyn' lists number and percentage of created nodes (w.r.t. the static variant) for the dynamic segmentation.

3d	$\tilde{\delta} = (10,10,10)^\top$								$\tilde{\delta} = (20,20,20)^\top$								$\tilde{\delta} = (30,30,30)^\top$							
	static						dyn		static						dyn		static						dyn	
file	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n
ft-01	44911	0.02	25	0.06	23	0.04	59	23353	52	175297	0.13	26	0.29	17	0.42	64	84055	48	509911	0.90	28	2.18	14	3.39	67	249151	49
ft-02	41167	0.02	26	0.05	22	0.04	59	20797	51	156073	0.10	27	0.22	16	0.32	64	71011	45	454819	0.83	31	2.07	16	2.71	67	216559	48
ft-03	69067	0.05	24	0.11	22	0.11	59	32773	47	270523	0.25	26	0.63	16	0.90	64	120295	44	798919	2.00	29	5.38	15	8.49	67	368917	46
ft-04	20539	0.01	24	0.02	22	0.02	59	10033	49	79573	0.04	27	0.06	15	0.08	64	35509	45	244231	0.23	33	0.45	15	0.64	67	111223	46
ft-05	88639	0.07	25	0.18	21	0.18	60	43537	49	341659	0.41	26	1.06	16	1.54	65	166135	49	1073017	3.90	29	10.99	15	16.40	67	510691	48
ft-06	7669	0.00	25	0.00	25	0.00	58	4345	57	21847	0.00	25	0.01	17	0.01	62	10123	46	57679	0.02	27	0.03	15	0.04	65	24133	42
ft-07	36481	0.02	25	0.04	23	0.04	59	18811	52	127453	0.07	27	0.16	17	0.20	63	59587	47	335035	0.41	28	0.94	16	1.34	66	146215	44
ft-08	31579	0.02	26	0.03	23	0.03	59	19801	63	106051	0.05	26	0.11	16	0.16	64	57055	54	304507	0.30	26	0.70	14	1.14	67	162241	53
ft-09	34465	0.02	25	0.04	23	0.03	59	17329	50	125611	0.07	26	0.15	16	0.18	63	54685	44	319837	0.30	26	0.76	14	1.21	67	139429	44
ft-10	47959	0.03	25	0.07	22	0.06	59	24109	50	166789	0.12	27	0.25	16	0.36	64	80341	48	469111	0.65	26	1.56	14	2.58	67	209263	45
ft-11	95269	0.07	25	0.17	20	0.17	60	45349	48	447937	0.60	27	1.69	16	2.46	66	192079	43	1380379	5.71	28	18.08	16	25.75	68	627649	45
ft-12	44167	0.02	26	0.04	21	0.07	60	23653	54	176443	0.13	27	0.27	16	0.41	65	88801	50	538279	0.85	28	2.12	14	3.43	68	281605	52
ft-13	30073	0.01	26	0.03	22	0.02	60	17041	57	118339	0.06	27	0.12	15	0.18	65	60433	51	333061	0.37	28	0.93	15	1.33	67	162193	49
ft-14	63535	0.04	26	0.09	21	0.09	60	35413	56	224899	0.19	26	0.44	16	0.68	65	114211	51	677059	1.36	28	3.72	14	5.71	67	346435	51
ft-15	47317	0.02	26	0.04	21	0.05	60	24775	52	172465	0.12	26	0.26	16	0.39	65	82885	48	495139	0.78	27	1.97	15	3.05	68	245713	50
ft-16	75529	0.06	26	0.14	22	0.13	60	38947	52	301213	0.37	27	0.92	17	1.35	65	139243	46	891415	2.73	28	6.95	15	10.77	67	433903	49
ft-17	47299	0.02	26	0.06	22	0.05	59	25393	54	176335	0.13	26	0.29	16	0.43	65	84463	48	505639	0.93	27	2.17	15	3.45	67	242455	48
ft-18	103723	0.08	26	0.21	21	0.23	60	53731	52	428689	0.58	27	1.53	16	2.36	65	209809	49	1342069	4.34	28	13.15	15	20.52	67	629761	47
ft-19	71689	0.05	26	0.12	21	0.11	60	35977	50	253567	0.20	26	0.49	15	0.74	64	116509	46	653701	1.04	26	3.04	14	4.54	67	299875	46
ft-20	34915	0.02	27	0.04	23	0.03	59	20485	59	120895	0.07	26	0.16	16	0.24	64	62323	52	322195	0.33	26	0.69	13	1.23	67	163003	51
AVG		0.03	25.5	0.08	22.0	0.08	59.4		52.6		0.18	26.4	0.46	16.1	0.67	64.3		47.7		1.40	27.9	3.89	14.7	5.89	67.0		47.5
stdev		0.02		0.06		0.06					0.17		0.48		0.72					1.57		4.84		7.14			

3d	$\tilde{\delta} = (10,10,10)^\top$								$\tilde{\delta} = (20,20,20)^\top$								$\tilde{\delta} = (30,30,30)^\top$							
	static						dyn		static						dyn		static						dyn	
file	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n
nist-g-01	297085	1.93	29	8.26	29	2.85	58	181399	61	719257	2.68	24	8.24	21	8.59	59	329923	46	1472215	5.58	25	16.35	18	23.16	61	612337	42
nist-g-02	251017	2.05	33	5.00	33	2.45	58	183079	73	511417	2.34	25	6.62	23	5.71	59	306823	60	918229	3.61	25	9.79	19	12.85	60	537253	59
nist-g-03	303871	2.16	30	5.54	30	3.01	58	197395	65	705037	2.86	24	8.59	21	8.87	60	383437	54	1442461	6.36	26	17.86	18	25.59	62	724075	50
nist-g-04	455953	4.21	29	11.53	29	6.39	58	273475	60	1108333	5.81	24	18.63	20	19.61	59	508609	46	2074825	11.89	25	34.70	18	50.55	62	942145	45
nist-g-05	166303	0.82	32	2.08	32	1.03	58	110179	66	314611	0.95	25	2.78	23	2.24	59	176227	56	565207	1.40	25	3.97	19	4.94	60	296059	52
nist-b-01	343921	2.54	29	6.84	29	3.74	58	210445	61	779281	3.46	24	10.94	21	10.75	59	383089	49	1524397	6.90	26	19.55	18	27.95	61	693007	45
nist-b-02	265033	1.57	30	4.14	30	2.37	58	161563	61	604711	2.22	25	6.43	21	6.91	60	311977	52	1203961	4.77	26	13.04	18	19.10	62	605329	50
nist-b-03	360211	2.65	29	7.12	29	4.06	58	204499	57	840943	3.53	24	11.33	21	11.48	59	372163	44	1665505	7.53	26	21.39	18	30.44	61	671605	40
nist-b-04	170011	0.86	32	2.18	32	1.08	58	110029	65	351085	1.05	24	3.12	23	2.62	59	173035	49	596575	1.72	25	4.52	19	5.68	60	274525	46
nist-b-05	246523	1.50	30	3.88	30	2.06	58	156391	63	560203	1.92	24	5.83	21	5.62	59	290005	52	1117501	3.82	25	16.71	18	14.64	61	524053	47
nist-u-01	251317	1.90	32	4.78	32	2.37	58	174367	69	495763	2.13	25	6.24	22	5.35	59	296977	60	919531	3.42	25	10.18	19	12.90	61	515287	56
nist-u-02	230479	1.47	31	3.72	31	1.91	58	153205	66	503269	1.79	24	5.35	22	5.03	59	268093	53	982669	3.22	25	9.29	19	13.24	61	493555	50
nist-u-03	264379	1.99	32	4.93	31	2.60	58	184693	70	618937	2.62	25	7.40	21	7.72	60	345109	56	1290457	5.68	26	15.01	18	22.06	62	678961	53
nist-u-04	175225	0.99	33	2.41	33	1.21	58	115729	66	381463	1.18	24	3.43	22	3.09	59	189151	50	678271	1.86	24	5.19	19	6.85	60	320203	47
nist-u-05	137695	0.56	32	1.39	32	0.71	58	91435	66	265675	0.64	24	1.84	23	1.52	59	144145	54	490645	1.00	24	4.29	19	3.43	60	236965	48
AVG		1.81	30.9	4.92	30.8	2.52	58.0		64.7		2.35	24.3	7.12	21.7	7.01	59.2		52.1		4.58	25.2	13.46	18.5	18.23	60.9		48.8
stdev		0.91		2.69		1.44					1.30		4.23		4.61					2.88		8.26		12.42			

Table 4: Three dimensional input parameter $\tilde{\delta} = (40, 40, 40)^\top, (80, 80, 80)^\top$. Run times on `G1` and percentages of visited nodes for the static segmentation for all three traversing variants UBD (*UB*-Driven Traversing), ABT (Advanced Bounding Traversal) and BFS (Best First Search). The percentage '%n' indicates visited nodes w.r.t. the number of total nodes '$|n|$' in percent. Column 'dyn' lists number and percentage of created nodes (w.r.t. the static variant) for the dynamic segmentation.

3d	$\tilde{\delta} = (40, 40, 40)^\top$									
			static					dyn		
file	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n	
ft-01	1211869	5.10	31	13.23	15	19.81	68	628129	52	
ft-02	1058251	4.28	32	10.82	15	15.64	67	556549	53	
ft-03	1937617	13.93	32	37.57	16	56.82	68	956137	49	
ft-04	532459	0.93	34	1.59	13	2.84	67	268093	50	
ft-05	2541433	26.91	31	75.54	15	115.49	67	1304257	51	
ft-06	125827	0.08	29	0.14	15	0.18	68	57331	46	
ft-07	817909	2.10	28	4.80	15	7.69	68	373387	46	
ft-08	750067	1.55	27	3.89	14	6.21	68	388723	52	
ft-09	717181	1.45	28	3.85	14	6.06	68	338419	47	
ft-10	1107115	3.06	27	7.21	14	11.90	68	487915	44	
ft-11	3399523	39.93	29	119.63	16	170.55	67	1660903	49	
ft-12	1293493	5.25	30	13.16	15	19.83	68	670921	52	
ft-13	831433	2.23	30	5.74	14	8.82	68	439621	53	
ft-14	1605067	7.83	29	21.58	15	31.46	68	843499	53	
ft-15	1325143	5.04	29	11.98	14	19.23	68	663373	50	
ft-16	2144569	12.97	29	30.02	14	50.53	68	1034167	48	
ft-17	1209067	5.10	28	10.67	14	17.01	68	570127	47	
ft-18	3033139	22.27	28	68.80	15	106.86	67	1428277	47	
ft-19	1564423	5.72	27	17.58	15	26.10	68	744397	48	
ft-20	713365	1.45	28	3.06	13	5.41	68	362899	51	
AVG		8.36	29.3	23.04	14.6	34.92	67.8		49.3	
stdev		10.32		30.86		45.30				

3d	$\tilde{\delta} = (40, 40, 40)^\top$									
			static					dyn		
file	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n	
nist-g-01	2511715	13.19	26	36.60	17	54.49	63	1029031	41	
nist-g-02	1545889	7.12	26	18.76	17	28.69	62	898765	58	
nist-g-03	2530663	15.08	26	45.85	16	67.87	64	1230583	49	
nist-g-04	3693943	29.14	25	87.97	16	133.14	64	1557961	42	
nist-g-05	919051	2.62	26	6.93	17	9.98	62	479845	52	
nist-b-01	2686135	16.33	26	46.18	17	71.78	64	1179229	44	
nist-b-02	2259883	12.82	26	36.69	16	56.54	64	1048873	46	
nist-b-03	2637925	16.92	26	49.21	17	74.17	64	1161265	44	
nist-b-04	1000693	3.31	26	8.19	18	12.13	62	456103	46	
nist-b-05	1934743	8.34	26	23.38	17	35.65	63	895801	46	
nist-u-01	1563925	6.59	26	18.57	17	27.63	62	865915	55	
nist-u-02	1712041	7.09	26	19.89	17	31.05	63	814375	48	
nist-u-03	2390287	15.14	26	41.33	16	63.90	64	1167967	49	
nist-u-04	1128943	3.63	26	9.48	17	14.32	62	517843	46	
nist-u-05	811339	1.93	25	4.96	17	7.32	62	380377	47	
AVG		10.62	25.9	30.27	16.8	45.91	63.0		47.5	
stdev		7.34		22.23		33.75				

3d	$\tilde{\delta} = (80, 80, 80)^\top$								
			static					dyn	
file	$\|n\|$	UBD	%n	ABT	%n	BFS	%n	$\|n\|$	%n
nist-g-01	14089453	[t]	[t]	[t]	[t]	[t]	[t]	5565961	40
nist-g-02	9337897	154.30	24	769.50	14			4630843	50
nist-g-03				memory overflow				7365505	
nist-g-04				memory overflow				[t]	[t]
nist-g-05	4875211	47.60	25	147.88	14	251.51	68	2281333	47
nist-b-01				memory overflow				7018759	
nist-b-02				memory overflow				6517957	
nist-b-03				memory overflow				7111615	
nist-b-04	5472625	60.62	25	177.49	14	300.00	67	2362807	43
nist-b-05	11648653	261.88	23	783.24	14	1468.98	68	4927951	42
nist-u-01	9335785	136.90	24	548.46	14			4136515	44
nist-u-02	10272037	190.51	24	656.35	14	1113.85	68	4412641	43
nist-u-03				memory overflow				7477975	
nist-u-04	6108757	64.44	25	205.65	14	366.05	67	2509591	41
nist-u-05	4192669	30.28	24	104.46	14	171.82	67	1752499	42
AVG		118.32	24.3	424.13	14.0	612.04	67.5		43.5
stdev		81.46		293.91		541.81			

Minimal Codebook Sizes

Table 5: Minimal codebook sizes m for fhg-files and all tested parameters.

		$k = 10$					$k = 10$			
file	$\|V\|$	$(10,10)^\top$	$(20,20)^\top$	$(30,30)^\top$	$(40,40)^\top$	$(80,80)^\top$	$(10,10,10)^\top$	$(30,30,30)^\top$	$(40,40,40)^\top$	$(80,80,80)^\top$
ft-01	31	2	1	1	1	1	4	2	1	1
ft-02	38	2	2	1	1	1	4	2	1	1
ft-03	35	2	1	1	1	1	4	2	1	1
ft-04	20	3	2	1	1	1	5	2	1	1
ft-05	39	2	1	1	1	1	4	2	1	1
ft-06	15	3	2	1	1	-	7	2	2	-
ft-07	28	3	2	1	1	1	5	2	2	1
ft-08	27	3	2	1	1	1	5	2	2	1
ft-09	27	3	2	1	1	1	4	2	2	1
ft-10	31	3	2	1	1	1	4	2	2	1
ft-11	38	2	1	1	1	1	4	2	1	1
ft-12	28	2	1	1	1	1	4	2	1	1
ft-13	25	2	2	1	1	1	4	2	1	1
ft-14	33	2	2	1	1	1	4	2	1	1
ft-15	29	2	1	1	1	1	4	2	1	1
ft-16	37	3	2	1	1	1	4	2	1	1
ft-17	31	3	2	1	1	1	4	2	2	1
ft-18	40	3	2	1	1	1	4	2	1	1
ft-19	35	2	2	1	1	1	4	2	2	1
ft-20	28	3	2	1	1	1	4	2	1	1

		$k = 20$					$k = 20$			
file	$\|V\|$	$(10,10)^\top$	$(20,20)^\top$	$(30,30)^\top$	$(40,40)^\top$	$(80,80)^\top$	$(10,10,10)^\top$	$(30,30,30)^\top$	$(40,40,40)^\top$	$(80,80,80)^\top$
ft-01	31	4	2	1	1	1	9	3	3	1
ft-02	38	4	2	2	1	1	8	3	3	1
ft-03	35	4	2	2	1	1	7	3	2	1
ft-04	20	6	3	2	2	1	11	4	3	2
ft-05	39	3	2	1	1	1	7	3	2	1
ft-06	15	-	-	-	-	-	-	-	-	-
ft-07	28	4	3	2	2	1	9	4	3	1
ft-08	27	4	3	2	1	1	10	4	3	1
ft-09	27	4	3	2	2	1	8	4	3	1
ft-10	31	4	2	2	1	1	8	3	3	1
ft-11	38	4	2	1	1	1	7	3	2	1
ft-12	28	4	2	1	1	1	7	3	2	1
ft-13	25	4	3	2	1	1	9	3	3	1
ft-14	33	4	2	2	1	1	7	3	2	1
ft-15	29	4	2	2	1	1	8	3	2	1
ft-16	37	4	2	2	1	1	7	3	2	1
ft-17	31	4	2	2	1	1	9	3	2	1
ft-18	40	4	2	2	1	1	7	3	2	1
ft-19	35	4	3	2	1	1	8	3	2	1
ft-20	28	5	3	2	2	1	8	3	3	1

		$k = 30$					$k = 30$			
file	$\|V\|$	$(10,10)^\top$	$(20,20)^\top$	$(30,30)^\top$	$(40,40)^\top$	$(80,80)^\top$	$(10,10,10)^\top$	$(30,30,30)^\top$	$(40,40,40)^\top$	$(80,80,80)^\top$
ft-01	31	6	3	2	2	1	14	6	4	2
ft-02	38	6	3	2	2	1	12	5	4	2
ft-03	35	5	3	2	2	1	11	4	3	2
ft-04	20	-	-	-	-	-	-	-	-	-
ft-05	39	5	2	2	1	1	11	4	3	
ft-06	15	-	-	-	-	-	-	-	-	-
ft-07	28	7	4	3	2	1	14	5	4	2
ft-08	27	6	3	2	1	1	14	5	4	2
ft-09	27	6	4	3	2	1	12	5	4	2
ft-10	31	6	3	2	2	1	13	5	4	2
ft-11	38	5	2	2	1	1	11	4	3	
ft-12	28	5	3	2	1	1	11	4	3	2
ft-13	25	-	-	-	-	-	-	-	-	-
ft-14	33	5	3	2	2	1	11	5	3	1
ft-15	29	5	3	2	2	1	11	5	3	1
ft-16	37	6	3	2	2	1	11	4	3	1
ft-17	31	6	3	3	2	1	14	5	3	2
ft-18	40	5	3	2	2	1	11	3	3	1
ft-19	35	6	3	2	2	1	12	4	3	2
ft-20	28	7	4	3	2	1	13	5	4	2

Standard Deviations for Branch-and-Price

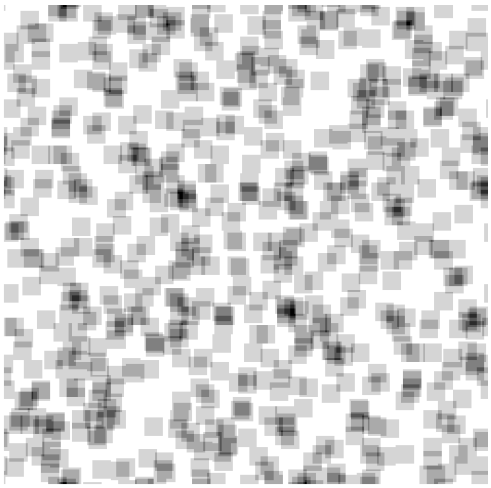
Table 6: Standard deviations for 2-dimensional $\tilde{\delta}$: Rows 'BP/static/[AVGopt]' ('BP/dyn/[AVGopt]') list the average standard deviation for all static (dynamic) tree pricing algorithm variants since the according values were very similar. Rows 'AVERAGE[BP]', 'MIN[BP]', 'MAX[BP]' list the average standard deviation for all tested branch-and-price variants.

k	options	$(10,10)^\top$				$(20,20)^\top$				$(30,30)^\top$				$(40,40)^\top$				$(80,80)^\top$			
		tot stdev	pit stdev	pvar stdev	bbn stdev	tot stdev	pit stdev	pvar stdev	bbn stdev	tot stdev	pit stdev	pvar stdev	bbn stdev	tot stdev	pit stdev	pvar stdev	bbn stdev	tot stdev	pit stdev	pvar stdev	bbn stdev
10	BC/UBD	396	-	-	-	2434	-	-	-	198	-	-	-	168	-	-	-	1131	-	-	-
	SCF/com	822	-	-	2215	1223	-	-	2268	116	-	-	56	10	-	-	0	160	-	-	0
	BP/static/nlc/cont	972	3606	421	3317	1416	2601	1097	1575	591	2021	1072	968	154	587	466	134	243	419	210	209
	BP/dyn/nlc/cont	2079	6340	457	6006	1315	2505	1074	1479	2245	5177	1340	3938	32	218	182	49	502	3181	1590	1591
	BP/dyn/del/nlc/fark	1146	3025	407	2718	1132	2408	1117	1320	14	116	104	12	6	7	5	2	53	0	0	0
	BP/static/[AVG]	1006	3122	390	2815	1173	3389	1044	2407	42	346	310	38	9	66	61	5	9	1	1	0
	BP/dyn/opt[AVG]	937	2997	384	2703	1126	3434	1049	2473	19	269	236	35	3	10	9	1	15	1	1	0
	AVERAGE[BP]	1057	3304	489	3019	1177	3208	1315	2272	215	696	780	341	18	80	559	14	63	226	515	113
	MIN[BP]	815	2672	368	2353	1032	2408	884	1320	14	116	104	12	1	2	2	0	0	0	0	0
	MAX[BP]	2079	6340	1897	6006	1416	4668	5197	3857	2245	5177	6683	3938	154	587	7866	134	502	3181	6432	1591
20	BC/UBD	97	-	-	-	1044	-	-	-	571	-	-	-	147	-	-	-	1998	-	-	-
	SCF/com	325	-	-	725	1069	-	-	4727	1323	-	-	3753	494	-	-	2504	115	-	-	4
	BP/static/nlc/cont	342	935	217	767	617	2580	721	1947	3130	3527	1970	1751	1471	2320	1824	496	457	468	415	54
	BP/dyn/nlc/cont	278	940	192	789	715	3060	781	2351	1817	2654	1679	1093	34	419	354	66	13	6	2	5
	BP/dyn/del/nlc/fark	365	1213	216	1021	1061	3272	900	2596	258	1062	906	159	86	458	433	31	55	5	1	4
	BP/static/[AVG]	389	1454	263	1218	830	2750	725	2147	659	2734	1296	1492	246	1419	975	595	9	11	9	4
	BP/dyn/opt[AVG]	364	1418	261	1183	532	2056	623	1563	500	2467	1272	1279	153	1510	1054	597	18	50	37	14
	AVERAGE[BP]	367	1359	317	1148	704	2726	888	2180	811	2583	1610	1399	266	1383	1317	561	46	58	502	14
	MIN[BP]	278	935	192	767	115	727	382	456	258	1062	906	159	34	419	354	31	2	1	1	0
	MAX[BP]	409	1657	1309	1399	1421	5864	3712	5729	3130	4172	5805	2971	1471	2320	6284	1277	457	468	7339	56
30	BC/UBD	47	-	-	-	251	-	-	-	844	-	-	-	951	-	-	-	3832	-	-	-
	SCF/com	189	-	-	419	348	-	-	404	253	-	-	1432	4150	-	-	10940	167	-	-	0
	BP/static/nlc/cont	152	743	197	560	475	1412	579	855	60	1059	358	783	80	240	229	10	2368	1258	1239	20
	BP/dyn/nlc/cont	275	1180	239	946	535	1945	672	1282	29	921	272	666	44	276	263	13	19	26	25	1
	BP/dyn/del/nlc/fark	124	460	150	318	56	390	232	178	103	569	348	254	49	269	261	9	56	1	1	0
	BP/static/[AVG]	103	445	152	306	366	1475	583	925	227	1542	646	1001	729	2240	1236	1069	126	198	188	11
	BP/dyn/opt[AVG]	85	380	146	245	345	1643	552	1132	191	1489	601	958	457	2232	1208	1050	40	189	178	14
	AVERAGE[BP]	693	2430	507	2080	945	2941	1409	2425	1146	2426	1881	1593	308	857	1631	454	286	589	1296	246
	MIN[BP]	97	935	192	725	115	727	382	456	14	116	104	12	1	2	2	0	0	0	0	0
	MAX[BP]	2079	6340	1897	6006	1421	5864	5197	5729	3130	5177	6683	3938	1471	2320	7866	2504	1998	3181	7339	1591

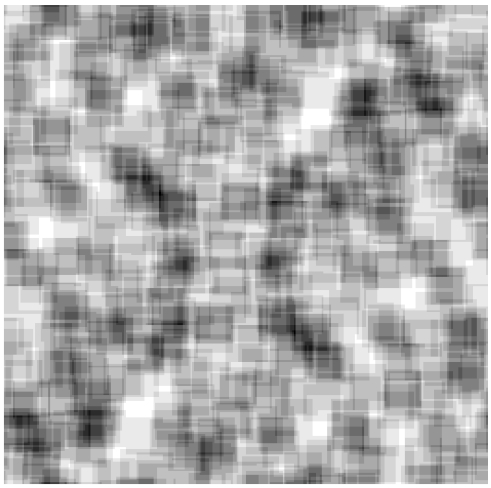
Table 7: Standard deviations 3-dimensional $\tilde{\delta}$: Rows 'BP/static/[AVG]' ('BP/dyn/opt[AVG]') list the average standard deviation over all static (dynamic) tree pricing algorithm variants, 'AVERAGE[BP]', 'MIN[BP]', 'MAX[BP]' list the average standard deviation for all tested branch-and-price variants.

k	options	$(10,10,10)^\top$				$(30,30,30)^\top$				$(40,40,40)^\top$				$(80,80,80)^\top$			
		tot stdev	pit stdev	pvar stdev	bbn stdev	tot stdev	pit stdev	pvar stdev	bbn stdev	tot stdev	pit stdev	pvar stdev	bbn stdev	tot stdev	pit stdev	pvar stdev	bbn stdev
10	BC/+UBD	26	-	-	-	55	-	-	-	99	-	-	-	-	-	-	-
	SCF/com	13	-	-	138	100	-	-	250	357	-	-	1550	-	-	-	-
	BP/static/nlc/cont	121	499	119	396	433	949	761	215	1085	2192	1553	661	-	-	-	-
	BP/dyn/nlc/cont	101	481	119	384	237	993	777	279	527	1853	1403	511	272	1074	906	280
	BP/dyn/del/nlc/fark	65	202	99	121	202	628	438	221	779	2143	1548	618	3	2	2	0
	BP/static/[AVG]	41	184	88	107	297	809	477	347	886	1976	1243	775	-	-	-	-
	BP/dyn/opt[AVG]	49	196	95	111	145	677	490	201	462	2063	1419	682	25	12	10	2
	AVERAGE[BP]	59	248	129	171	259	753	1481	285	771	1943	4606	714	62	187	158	48
	MIN[BP]	25	134	78	63	114	409	412	119	259	905	775	355	2	2	2	0
	MAX[BP]	121	499	559	403	469	1451	13931	866	1890	2981	46852	961	272	1074	906	280
20	BC/+UBD	5	-	-	-	146	-	-	-	291	-	-	-	-	-	-	-
	SCF/com	14	-	-	136	1037	-	-	2228	575	-	-	5090	-	-	-	-
	BP/static/nlc/cont	101	322	106	222	2022	5149	1017	4182	4048	6899	2474	5445	-	-	-	-
	BP/dyn/nlc/cont	82	245	92	157	1801	5240	999	4290	1690	8029	1866	6585	269	800	776	25
	BP/dyn/del/nlc/fark	48	175	96	82	1006	1978	972	1018	706	2022	1278	957	2068	1778	1723	57
	BP/static/[AVG]	39	213	89	136	986	2590	828	1813	1341	4335	1341	3334	-	-	-	-
	BP/dyn/opt[AVG]	31	177	89	96	738	2564	1002	1599	794	3420	1490	2244	801	1956	1841	185
	AVERAGE[BP]	47	222	115	145	1137	2932	1712	2109	1421	4455	4475	3398	924	1734	1644	137
	MIN[BP]	22	155	79	72	556	1602	767	846	477	1754	1042	826	50	458	436	25
	MAX[BP]	101	385	430	375	2227	5240	12001	4290	4048	8029	43026	6818	2068	3586	3260	591
30	BC/+UBD	7	-	-	-	129	-	-	-	397	-	-	-	-	-	-	-
	SCF/com	11	-	-	37	732	-	-	3955	3245	-	-	6363	-	-	-	-
	BP/static/nlc/cont	20	95	51	48	873	3155	754	2466	1643	1989	1185	848	-	-	-	-
	BP/dyn/nlc/cont	19	92	50	46	204	1902	467	1497	647	2199	1366	906	96	150	130	23
	BP/dyn/del/nlc/fark	18	90	36	59	101	365	197	196	1625	2569	1644	944	172	312	302	15
	BP/static/[AVG]	15	105	47	62	520	1731	528	1281	2395	3395	1560	1961	-	-	-	-
	BP/dyn/opt[AVG]	17	104	49	58	245	1190	456	811	1587	4013	1902	2190	1890	2011	1918	105
	AVERAGE[BP]	16	110	58	69	424	1566	1274	1166	2006	3248	4037	1809	1304	1418	1351	76
	MIN[BP]	11	55	35	19	76	365	197	196	647	978	728	476	96	150	130	15
	MAX[BP]	24	215	197	203	873	3155	11422	2466	3588	6370	35358	4648	4320	3303	3268	318

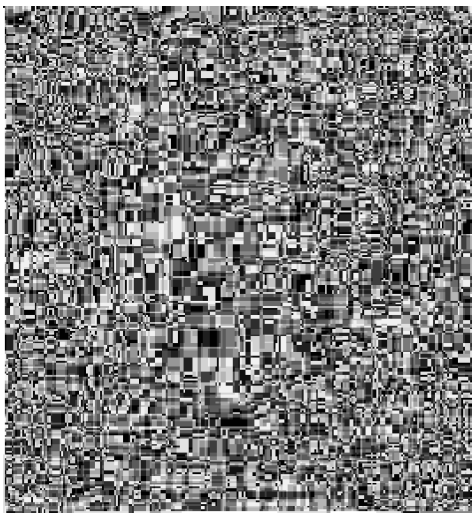
Making Art with the Segmentation Tree by Visualization



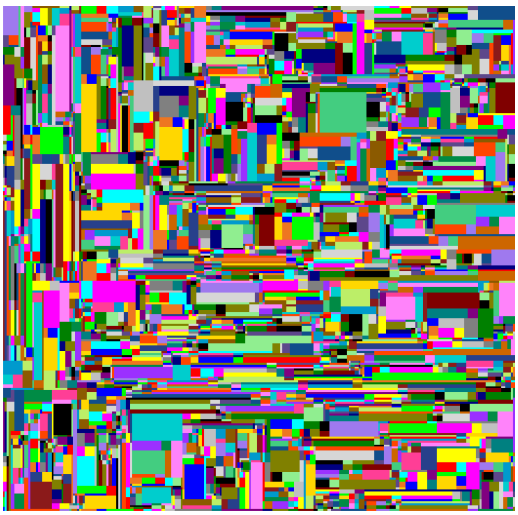
Visualization of bounding boxes, 2-d static segtree, instance `ft-09`, $\tilde{\delta} = (10, 10)^\top$



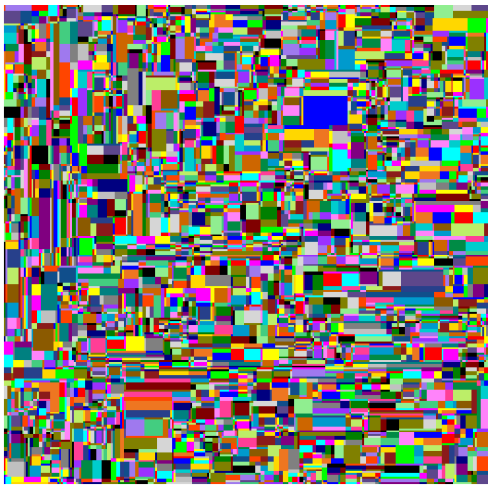
Visualization of bounding boxes, 2-d static segtree, instance `ft-09`, $\tilde{\delta} = (20, 20)^\top$



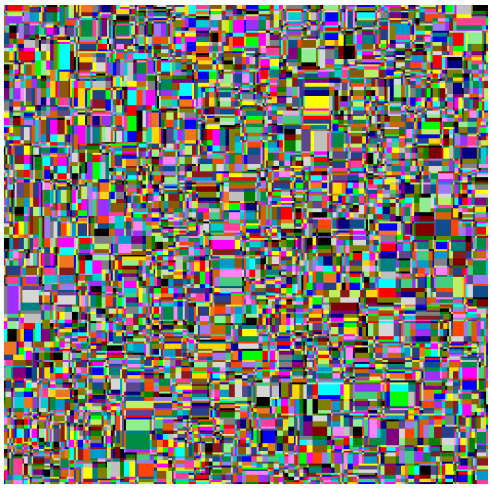
Black and white segmentation visualization, 2-d static tree, instance `ft-03`, $\tilde{\delta} = (40, 40)^\top$



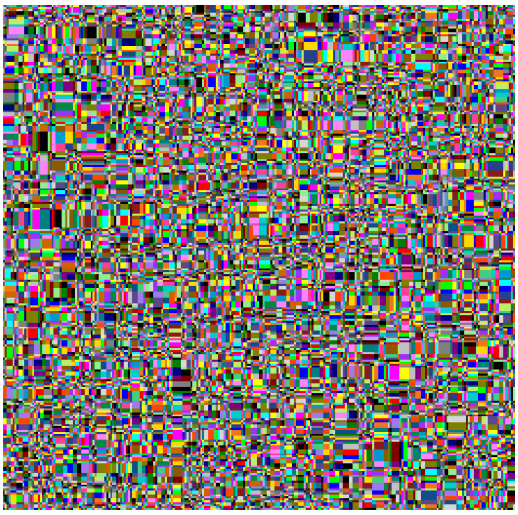
Color segmentation visualization, 2-d static tree, instance `ft-09`, $\tilde{\delta} = (5, 5)^\top$



Color segmentation visualization, 2-d static tree, instance `ft-09`, $\tilde{\delta} = (10, 10)^\top$



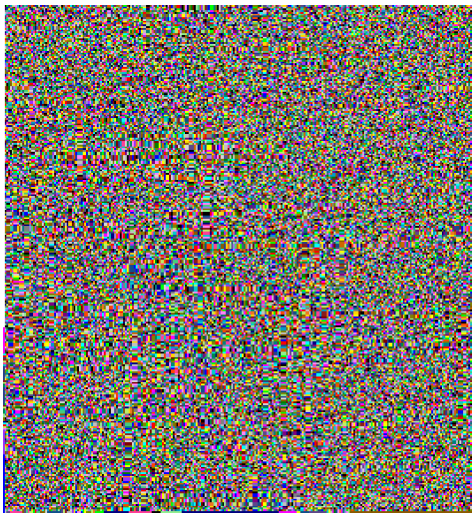
Color segmentation visualization, 2-d static tree, instance `ft-09`, $\tilde{\delta} = (20, 20)^\top$



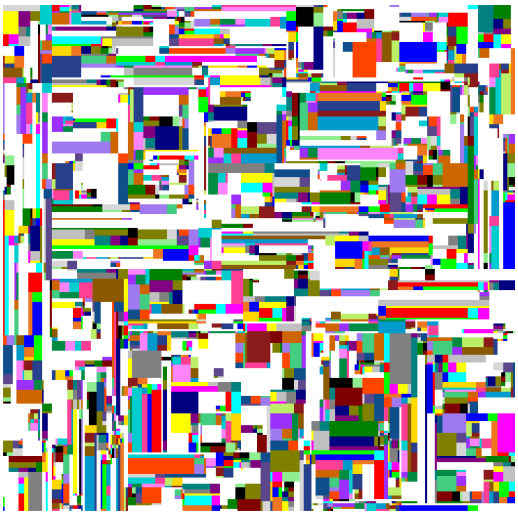
Color segmentation visualization, 2-d static tree, instance `ft-09`, $\bar{\delta} = (40, 40)^\top$

Color segmentation visualization, 2-d static tree, instance `ft-06`, $\bar{\delta} = (10, 10)^\top$

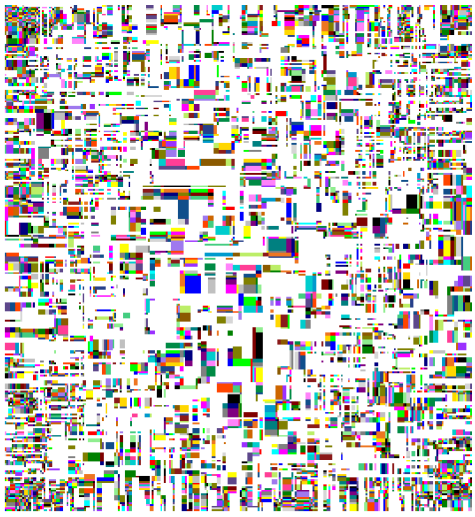
Color segmentation visualization, 2-d static tree, instance `ft-03`, $\bar{\delta} = (40, 40)^\top$



Color segmentation visualization, 2-d static tree, instance `ft-03`, $\bar{\delta} = (80, 80)^\top$



Visualization of the segmented parts of a 2-d dynamic tree, instance `ft-09`, $\bar{\delta} = (5, 5)^\top$



Visualization of the segmented parts of a 2-d dynamic tree, instance `ft-03`, $\bar{\delta} = (20, 20)^\top$

List of Algorithms

List of Figures

List of Tables

Bibliography

[Jain08] Jain, Anil K. [Hrsg.]; Flynn, Patrick; Ross, Arun A. (Eds.): *Handbook of biometrics*, New York, NY : Springer Science + Business Media, (2008). ISBN: 978-0-387-71040-2.

[Heum06] Heumann, Björn: *Whitepaper Biometrie*, `http://www.biometrie-online.de`, `http://www.heumann-webdesign.de/pages/biometrie/Whitepaper_Biometrie.pdf`, (2006).

[BasSchu05] Basler, Georg; Schutt, Andreas: *Fingerabdrucksysteme*, Humboldt Universität, Institut für Informatik, `http://www2.informatik.hu-berlin.de/Forschung_Lehre/algorithmenII/Lehre/SS2004/Biometrie/04Fingerprint/fingerabdrucksysteme.pdf`, Technical Report, (2005).

[MalJai05] Maltoni, Davide; Jain, Anil K.; Maio, Dario; Prabhakar, Salil: *Handbook of fingerprint recognition*, Edition 2, New York, NY [u.a.], Springer, (2005). ISBN 0-387-95431-7.

[JaRoPr01] Jain, Anil K.; Ross, Arun; Prabhakar, Salil: *Fingerprint Matching using minutiae and texture features* In: International Conference on Image Processing (ICIP), pp. 282-285, (2001).

[webWong] Wong, Kevin: *Fingerprint identification*, `http://www.lems.brown.edu/vision/courses/computer-vision-2002/projects/Wong/mid.htm`, (2002).

[Seitz05] Seitz, Juergen: *Digital Watermarking for Digital Media*, Hershey, Pa. : Information Science Publ., (2005). ISBN 1-591-40518-1.

[Fraun08] *Mediensicherheit*, `http://watermarking.sit.fraunhofer.de`, (Februar 2008).

[Sayood06] Sayood, K.: *Introduction to Data Compression*, 3rd edition, Morgan Kaufmann Publishers Inc., San Mateo, (2006).

[ChwRai09] Chwatal, Andreas M.; Raidl, Günther R.; Oberlechner, Karin: *Solving a k-Node Minimum Label Spanning Arborescence Problem to Compress Fingerprint Templates*, In: Journal of Mathematical Modelling and Algorithms (JMMA), Springer Netherlands, Vol. 8, No. 3, pp. 293-334, (August 2009).

[ChwRai07] Chwatal, Andreas M.; Raidl, Günther R.: *Applying branch-and-cut for compressing fingerprint templates*, short abstract, In: Proceedingsof the European Conference on Operational Research (EURO) XXII, Prague, (2007).

[RaiChw07] Raidl, Günther; Chwatal, Andreas: *Fingerprint Template Compression by Solving a Minimum Label k-Node Subtree Problem*, In: Simos, E. (editor), Numerical Analysis and Applied Mathematics, AIP Conference Proceedings, Vol. 936, pp. 444-447. American Institute of Physics, New York, (2007).

[ChwMIC07] Chwatal, Andreas M.; Raidl, Günther R.; Dietzel, Olivia: *Compressing Fingerprint Templates by Solving an Extended Minimum Label Spanning Tree Problem*, In: Proceedings of MIC 2007, the 7th Metaheuristics International Conference, pp. 105/1-3, Montreal, Canada, (2007).

[Dietzel08] Dietzel, Olivia: *Combinatorial Optimization for the Compression of Biometric Templates*, Master's thesis, Vienna University of Technology, Institute for Computer Graphics and Algorithms, Supervised by G. Raidl and A. Chwatal, (May 2008).

[SalAdh01] Saleh, A.; Adhami, R.: *Curvature-based matching approach for automatic fingerprint identification*, In: Proceedings of the Southeastern Symposium on System Theory, pp. 171175, (2001).

[CormLei07] Cormen, Th. H.; Leiserson, Ch. E. (Editors); Rivest, R.; Stein, C.: *Algorithmen - Eine Einführung*, 2. Auflage, Oldenbourg, München, Wien, (2007). ISBN: 978-3-486-58262-8.

[CormLei01] Cormen, Th. H.; Leiserson, Ch. E. (Editors); Rivest, R.; Stein, C.: *Introduction to Algorithms*, 2nd edition, MIT Press, Cambridge, (2001). ISBN: 0-07-013151-1.

[Sedgew01] Sedgewick, Robert: *Algorithms in C*, 3rd edition, Addison-Wesley Professional, (2001).

[Math99] Stöcker, Prof. Dr. Horst (Hrsg.): *Taschenbuch mathematischer Formeln und moderner Verfahren*, 4., korrigierte Auflage, Verlag Harri Deutsch, Frankfurt am Main, Thun, (1999).

[webCaldwell05] Caldwell, Chris K.: *Graph Theory Tutorials*, `http://www.utm.edu/departments/math/graph/`, (2005).

[Evans78] Evans, James R.: *On equivalent representations of certain multicommodity networks as single commodity flow problems*, Springer Berlin / Heidelberg, Mathematical Programming, Vol. 15, No. 1, pp. 92-99, (December 1978).

[Ev78] Evans, J.: *A single commodity transform for certain multi commodity networks*, Operations Research, Vol. 26, No. 4, pp. 673-680, (1978).

[Bentley75] Bentley, Jon Louis: *Multidimensional Binary Search Trees Used for Associative Searching*, Communications of the ACM, Vol. 18, No. 9, pp. 509-517, (September 1975).

[webGraphAlgo] Emden-Weinert, Thomas; Hougardy, Stefan; Kreuter, Bernd; Prömel, Hans Jürgen; Steger, Angelika: *Einführung in Graphen und Algorithmen*, `http://www2.informatik.hu-berlin.de/alkox/lehre/skripte/ga/`, Online Skriptum von Matroids Matheplanet, (1996).

[ChiKa08] Chimani, M.; Kandyba, M.; Ljubic I.; Mutzel, P.: *Obtaining Optimal k-Cardinality Trees Fast*, Proceedings of Workshop on Algorithm Engineering and Experiments (ALENEX'08), San Francisco, (2008). `http://ls11-www.cs.uni-dortmund.de/people/kandyba/kcard.html`.

[ChLe97] Chang, Ruay-Shiung; Leu, Shing-Jiuan: *The Minimum Labeling Spanning Trees*, In: Information Processing Letters Vol. 63, No. 5, pp. 277-282, (1997).

[Kru98] Krumke, Sven O.; Wirth, Hans Christoph: *On The Minimum Label Spanning Tree Problem*, In: Information Processing Letters, Vol. 66, No. 2, pp. 81-85, (1998).

[XiGolWas05] Xiong, Yupei; Golden, Bruce; Wasil, Edward: *A One-Parameter Genetic Algorithm for the Minimum Labeling Spanning Tree Problem*, In: IEEE Transactions on Evolutionary Computation, Vol. 9, No. 1, pp. 55-60, (February 2005).

[Xioung05] Xiong, Yupei: *The Minimum Label Spanning Tree Problem and some Variants*, Dissertation, Faculty of the Graduate School of the University of Maryland, (2005).

[Cons06] Consoli, S.; Moreno José, A.; Mladenovic, N.; Darby-Dowman, K.: *Constructive Heuristics for the Minimum Labelling Spanning Tree Problem: a preliminary comparison*, DEIOC Technical Report, September 2006.

[XiGolWas06] Xiong, Yupei; Golden, Bruce; Wasil, Edward: *The Minimum Label Spanning Tree Problem: Some Genetic Algorithm Approaches*, Lunteren Conference on the Mathematics of Operations Research, The Netherlands, (January 2006).

[Wol98] Wolsey, Laurence A.: *Integer Programming*, New York, NY, Wiley, John Wiley & Sons, Chichester, (1998). ISBN: 978-0-471-28366-9.

[Van98] Vanderbei, Robert J.: *Linear Programming: Foundations and extensions*, Boston, Mass., Kluver, (1998). ISBN: 0-7923-8141-6.

[webDesSoum98] Desrosiers, Jacques; Soumis, Franois: *Column Generation*, `http://www.gerad.ca/~gencol/gceng.html`, `http://www.gerad.ca/~gencol/Column/`, Team GÉNération de COLonnes, Various Universities, Online since 1998.

[BoydVan04] Boyd, Stephen; Vandenberghe, Lieven: *Convex Optimization*, Cambridge Univ. Press, (2004). ISBN: 0-521-83378-7.

[DesLu05] Lübbecke, Marco E.; Desrosiers, Jacques: *Selected Topics in Column Generation*, In: Operations Research, Vol 53, pp. 1007-1023, (November, December 2005).

[Luebbe01] Lübbecke, Marco; Zimmermann, Prof. Dr. Uwe T.; Desrosiers, Prof. Dr. Jacques; Fekete, Prof. Dr. Sándor P.: *Engine Scheduling by Column Generation*, Dissertation, Fachbereich f ür Mathematik und Informatik der Technischen Universität Braunschweig, (Juli 2001).

[VanWol96] Vanderbeck, Francois; Wolsey, Laurence A.: *An Exact Algorithm for IP Column Generation*, Elsevier Science, Operations Research Letters, Vol. 19, Iss. 4, pp. 151 - 159, (November 1996).

[GuzRal07] Guzelsoy, Menal; Ralphs, Ted K.: *Duality for Mixed-Integer Linear Programs*, Technical Report, COR@L Lab, Industrial and Systems Engineering, Lehigh University, (2007).

[webTrick97] Trick, Michael A.: *A Tutorial on Integer Programming*, `http://mat.gsia.cmu.edu/orclass/integer/integer.html`, (1998).

[Vanbk94] Vanderbeck, Francois; (Wolsey, L.A.): *Decomposition and Column Generation for Integer Programs*, PhD Thesis, Université catholique de Louvain (UCL), `http://www.math.u-bordeaux.fr/~fv/publications.html`, (1994).

[NemWol88] Wolsey, Laurence A., Nemhauser, George L.: *Integer and Combinatorial Optimization* Wiley, New York, (1988).

[Schrij99] Schrijver, Alexander: *Theory of Linear and Integer Programming*, Wiley Interscience Series in Discrete Mathematics and Optimization, Chichester, (1999).

[AvKa04] Avis, David; Kaluzny, Bohdan: *Solving Inequalities and Proving Farkass Lemma Made Easy*, Amer. Math. Monthly, The Mathematical Association of America, Vol. 111, No. 2, pp. 152-157. (2004).

[MacPlLi03] Maculan, Nelson; Plateau, Gérard; Lisser, Abdel: *Integer Linear Models with a Polynomial Number of Variables and Constraints for some Classical Combinatorial Optimization Problems*, Pesquisa Operacional, Vol. 23, No. 1, pp. 161-168, (2003).

[MagWol94] Magnanti, Thomas L.; Wolsey Laurence A.: *Optimal Trees*, Handbooks in Operations Research and Management Science, Volume entitled *Networks*, OR 209-94, (1994).

[CaClPa09] Captivo M. Eugnia; Clmaco, Joo C.N.; Pascoal, Marta M.B.; *A mixed integer linear formulation for the minimum label spanning tree problem* Computers & Operations Research 36, pp. 3082–3085, (2009).

[Schwa08] Schwarz, Cornelius: *An Introduction to SCIP*, Univerity of Bayreuth, `http://scip.zib.de/download/files/scip_intro_01.pdf` (2008).

[Pfe07] Pfetsch, Marc: *Introduction to SCIP*, DFG Research Center Matheion and Zuse Institute Berlin, SCIP Workshop 2007, Berlin, `http://scip.zib.de/download/slides/SCIP-introduction.pdf` (2007).

[Ber07] Berthold, Timo: *Primal Heuristics in SCIP*, DFG Research Center Matheion and Zuse Institute Berlin, SCIP Workshop 2007, Berlin, `http://scip.zib.de/download/slides/SCIP-primalHeuristics.pdf` (2007).

[Ach07] Achterberg, Tobias: *Branching - SCIP Workshop at ZIB*, DFG Research Center Matheion and Zuse Institute Berlin, SCIP Workshop 2007, Berlin, `http://scip.zib.de/download/slides/SCIP-branching.ppt` (2007).

[Wolt07] Wolter, Kati: *Cutting Plane Separators in SCIP*, DFG Research Center Matheion and Zuse Institute Berlin, SCIP Workshop 2007, Berlin, `http://scip.zib.de/download/slides/SCIP-cuttingPlanes.pdf` (2007).

[Ach07-2] Achterberg, Tobias: *Conflict Analysis - SCIP Workshop at ZIB*, DFG Research Center Matheion and Zuse Institute Berlin, SCIP Workshop 2007, Berlin, `http://scip.zib.de/download/slides/SCIP-conflictAnalysis.ppt` (2007).

[Achter07] Achterberg, Tobias: *Constraint Integer Programming*, Ph.D. thesis, Technische Universität Berlin, `http://opus.kobv.de/tuberlin/volltexte/2007/1611/` (July 2007).

[AcBeKoWo08] Achterberg, Tobias; Berthold, Timo; Koch, Thorsten; Wolter, Kati: *Constraint Integer Programming: a New Approach to Integrate CP and MIP*, Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems CPAIO 2008, LNCS 5015, (2008).

[Achter04] Achterberg, Tobias: *SCIP - a framework to integrate Constraint and Mixed Integer Programming*, Technical Report ZR-04-19, Zuse Institute Berlin ZIB, `http://www.zib.de/Publications/abstracts/ZR-04-19/` (June 2004).

[NIST] Garris, M.D.; McCabe, R.M.: NIST special database 27: fingerprint minutiae from latent and matching tenprint images, Technical Report, U.S. National Institute of Standards and Technology, `http://www.nist.gov`. (2000).

[ILOG] ILOG Concert Technology, CPLEX: ILOG, `http://www.ilog.com`, Version 11.0, (2009).

[OptOn] Optimization Online,
`http://www.optimization-online.org`,
Optimization Online, supported by the Mathematical Programming Society and by the Optimization Technology Center, Since 2000.

[Fraun] Fraunhofer Gesellschaft zur Förderung der angewandten Forschung.
`http://www.fraunhofer.de`.

[SCIP] Framework for Solving Constraint Integer Programs (SCIP), From website of ZIB, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Department Scientific Computing, Department Optimization, `http://scip.zib.de`.

[ZIBOpt] ZIB Optimization Suite Website, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Department Scientific Computing, Department Optimization, `http://zibopt.zib.de`.