FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Solving the *k*-Node Minimum Label Spanning Arborescence Problem with Exact and Heuristic Methods

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Mag. Karin Oberlechner

Matrikelnummer 9303191

an der
Fakultät für Informatik der Technischen Universität Wien

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

Wien, 14.08.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Solving the *k*-Node Minimum Label Spanning Arborescence Problem with Exact and Heuristic Methods

Karin Oberlechner

14.08.2010

Erklärung zur Verfassung der Arbeit

Karin Oberlechner, Rotensterngasse 31/19 1020 Wien

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

Wien, 14.08.2010

Kurzfassung

In dieser Diplomarbeit werden exakte und heuristische Methoden entwickelt um das *k-node Minimum Label Spanning Arborescence (k-MLSA)* Problem zu lösen. Dieses Problem ist eine Kombination des *Minimum Label Spanning Tree (MLST)* Problems und des *k-cardinality Tree* Problems, die beide NP-vollständig sind. Beim *MLST* Problem soll in einem gerichteten Graphen, dessen Kanten ein oder mehrere Label zugeordnet sind, die minimale Teilmenge an Labels gefunden werden, sodass die entsprechenden Kanten einen gerichteten Baum mit zumindest k Knoten des Graphen enthalten. Dieses Problem ergibt sich im Kontext eines laufenden Projektes am Institut für Computer Graphik und Algorithmen (Technische Universität Wien), in dem ein neuer graphenbasierter Ansatz zur Datenkompression entwickelt wurde. Neben den exakten und heuristischen Methoden zur Lösung des *k-MLSA* Problems wird in dieser Diplomarbeit ein neuer *Preprocessing* Algorithmus entwickelt um die initialen Labels zu erzeugen.

Die entwickelte Heuristik ist ein *memetischer Algorithmus (MA)*, d.h. die Kombination eines genetischen Algorithmus mit lokalen Suchmethoden, die dazu dienen die mittels evolutionärer Operatoren gefundenen Kandidatenlösungen weiter zu verbessern. Der exakte Algorithmus basiert auf Methoden der mathematischen Programmierung. Zur Lösung des zugrundeliegenden ganzzahligen linearen Modells werden *Branch-and-Cut* und Spalten Generierung zu einem *Branch-and-Cut-and-Price (BCP)* Algorithmus verbunden. Dieser Algorithmus erzeugt neue (Label) Variablen und Nebenbedingungen dynamisch während des *Branch-and-Bound* Prozesses, und fügt diese zu einem ursprünglich unvollständigen Modell hinzu. Die neuen Variablen werden erzeugt indem das *Pricing Problem* basierend auf den Werten der dualen Variablen der jeweiligen Lösung berechnet wird. In diesem Ansatz müssen die (Label) Variablen nicht mehr in einem Preprocessing Schritt generiert werden. Da das *Pricing Problem* im Verlauf des *BCP* Prozesses häufig gelöst werden muss, erfordert es besonders effiziente Algorithmen. Zu diesem Zweck wird eine Methode vorgestellt, die auf der Suche von nicht dominierten Intervallen basiert und effiziente Lösungen sowohl für das *Pricing Problem* als auch für den Preprocessing Schritt generieren kann.

Obwohl die in dieser Diplomarbeit vorgestellte *BCP* Methode grössere Instanzen in kürzerer Zeit lösen kann als die schon entwickelten *Branch-and-Cut* und *Branch-and-Price* Algorithmen ist für praktische Anwendungen der *MA* geeigneter, da dieser fast optimale Lösungen für grössere Instanzen in sehr kurzer Zeit findet. Die neue Preprocessing Methode reduziert die Berechnungskomplexiät im Vergleich zur früheren Methode, die einen Flaschenhals im Verfahren bildete, signifikant.

Abstract

In this thesis, exact and heuristic methods for solving the *k-node minimum label spanning arborescence (k-MLSA)* problem are developed. The *k-MLSA* problem is a combination of the *minimum label spanning tree* problem and the *k-cardinality tree* problem, which are both NP-complete. Given a complete, directed graph in which one or more labels are associated with each arc, the goal is to derive a minimum cardinality subset of labels such that their corresponding arcs contain a directed tree connecting at least k nodes of the graph. The problem arises in the context of a data-compression model, which has been developed as part of a project at the Institute of Computer Graphics and Algorithms (Vienna University of Technology). In addition to exact and heuristic methods for solving the *k-MLSA* problem, this thesis contributes a new preprocessing algorithm for constructing the initial labels.

The heuristic method is a *memetic algorithm (MA)*, i.e. a combination of a genetic algorithm with local search methods, which are used to further improve candidate solutions derived by the evolutionary operators. The exact algorithm is based on mathematical programming. It solves the underlying integer programming formulation by combining *branch-and-cut* with column generation in a *branch-and-cut-and-price (BCP)* approach. In this approach, new (label) variables and constraints are added dynamically to an incomplete initial model during the *branch-and-bound* process. The new variables are generated by solving the pricing problem, which is based on the values of the dual variables of the current solution. Hence, the label variables no longer need to be constructed in a preprocessing step. As the pricing problem needs to be solved frequently as part of the overall *BCP* process, efficient algorithms are required. For this purpose, a method based on non-dominated interval search is proposed. It provides an efficient solution to the pricing problem as well as to the preprocessing step.

The *BCP* approach proposed in this thesis can solve larger instances than previously developed *branch-and-cut* and *branch-and-price* algorithms. However, for practical applications, the *MA* is more appropriate as it is able to find near-optimal solutions for larger instances within short running times. The new preprocessing method significantly reduces computational complexity compared to the previous method, which has been the bottleneck in the overall procedure.

Acknowledgments

I want to thank my advisors Günther Raidl and Andreas Chwatal for making this thesis possible as well as for providing many interesting ideas and being a great help overall. Special thanks go to Andreas Chwatal for his valuable advice, for his thorough reviews and for taking so much time for many interesting tips and discussions. I would also like to thank Aksel Filipovic, the helpful sys-admin of the institute, for coping so well with the relentless onslaught of my testing procedures. Furthermore, I want to thank my friends and family for their support and my partner Alexander Jerusalem for being inspiring and patient.

Contents

1. Introduction

In this thesis, both exact and heuristic methods are developed to solve the *k-node minimum label spanning arborescence (k-MLSA)* problem. For the *k-MLSA* problem we are given a complete directed graph with (multiple) labels associated with its arcs. The goal is to derive a minimum cardinality subset of the labels such that their corresponding arcs contain a directed tree connecting at least k nodes of the initial graph.

The application background is the compression of fingerprints for embedding into images using digital watermarking techniques. For instance, biometric passports could contain two fingerprints of the passport holder embedded into the photo as an additional security feature. Since the size of the photograph is just $6 - 20$ kilobytes, it allows for the storage of only one fingerprint. Hence, new compression methods are required in order to embed both fingerprints [22]. These methods are obviously compression methods for very small datasets. A comparison of our approach to other compression techniques is given in [22]. Fingerprint images can be represented by their so called minutiae, i.e. distinctive features like ridge endings, crossover, bifurcations, islands and pores. Our model uses four attributes to describe minutiae: type, x and y coordinates as well the angle θ between x-axis and the tangent of the ridge that ends in the respective point. This angle indicates the orientation of the minutiae.

The input data comprises n minutiae, which are interpreted as four-dimensional points. These points are represented by the nodes of a complete directed graph $G = (V, A)$. For the compression a subset k of all points is selected and connected by a directed spanning tree. Thus, the arcs can be represented relative to each other. Furthermore we use a dictionary. This dictionary contains template arcs and each arc is represented as a reference to the most similiar template arc and a correction vector. Considering these template arcs as edge labels allows us to transform the problem into the problem of finding a directed *k-node minimum label spanning arborescence (k-MLSA)*.

The *k-MLSA* problem is an extension of the *minimum label spanning tree (MLST)* problem and the *k-cardinality* tree problem, which are both NP-complete. The input to the *MLST* problem is an undirected graph with labeled arcs. The goal is to find a spanning tree that contains the minimum number of labels. An application of the *MLST* problem is, for example, the design of communication networks, where uniformity is often required. The *k-MLSA* problem extends the *MLST* problem insofar as the graph is directed and only a subset of k nodes is connected. Moreover, in our approach, the labels have geometric properties. Even though no proof exists, it is assumed that the *k-MLSA* problem is also NP-hard. Insofar as only a subset k of the nodes is selected, the problem is related to the *k-cardinality* problem, also known as *k-minimum spanning tree* problem, which is also NP-hard. That problem receives an undirected graph $G(V, A)$, an edge weight function and a number k as input and tries to find the minimum subgraph $G' \subset G$ that contains k nodes.

This master's thesis is part of an ongoing project at the Institute of Computer Graphics and Algorithms (Vienna University of Technology) where the described graph based compression method has been developed [41, 18]. Two problems that were raised in previous work are solved in this thesis:

1. Instances with a large number of variables could not be solved by the existing *branch-and-cut (BC)* and *branch-and-price (BP)* approaches.

2. The template arcs (labels) used in our problem are not provided as input. They have to be

constructed in a preprocessing step, which turned out to be a bottleneck. On the one hand, the existing methods were too slow to meet the practical requirement of a running time of merely a few minutes. On the other hand, the preprocessing of large data sets was not possible at all, due to high memory usage.

The contribution of this work is the development of new exact and heuristic methods for solving the *k-MLSA* problem as well as a new approach to solving the related preprocessing problem. The memetic algorithm *MA* is a combination of a genetic algorithm and local search methods, which are used to further improve candidate solutions derived by the evolutionary operators.

The exact algorithm combines branch-and-cut and column-generation to *branch-and-cut-and-price (BCP)* to solve the underlying integer programming formulation. In this approach, new (label) variables and constraints are dynamically added to an incomplete initial model during the branch-and-bound process. For the preprocessing step, two approaches where conceived. In the first approach, the template arcs are not constructed in advance. Instead, they are constructed on demand in the pricing step of the *BCP* method. The second approach is a completely new preprocessing algorithm called *non-dominated interval search (NIS)*. It constructs all candidate template arcs from the input dataset very quickly and with low memory usage.

1.1 Related Work

The *MLST* problem was introduced by Chang and Leu [10]. It has been solved with heuristic (*Maximum Vertex Covering Algorithm (MVCA)*, *Edge Replacement*) and exact (*A*-Algorithm*) algorithms. Chang and Leu also proved that the *MLST* problem is NP-complete by reducing the set covering problem to it. Their proof works as follows: A decision version of the *MLST* problem, the *bounded labeling spanning tree problem (BLST)* is defined and its NP-completeness is shown.

Definition 1. (BLST problem) Given a graph $G = (V, A)$, a positive integer J and a labeling function $L(a)$ $\forall a \in A$, is there a spanning tree T for G, such that $|L_T| \leq J$ [10]?

It is easy to see that $BLST \in NP$. First, a subset of edges is guessed. Then it is checked in polynomial time whether this subset connects all vertices and whether L_T is smaller than or equal to J. It is proved by transforming the problem to a minimum set cover problem, which is known to be NP-complete. Given a set $S = \{x_1, ..., x_n\}$ and the subsets $C_1 = \{x_{i_1}, x_{j_1}, x_{k_1}\}, ..., C_m = \{x_{i_m}, x_{j_m}, x_{k_m}\}$ of S, the minimum set cover problem asks for a minimum number of subsets that can cover all elements of S. The next step in the proof is to construct a graph from S that contains a cover of $k - 1$ subsets, if and only if it has a minimum label spanning tree with j labels. It can be shown that this graph contains a *MLST* with J labels if and only if a minimum cover with $J - 1$ subsets exists and that the construction of *MLST* can be done in polynomial time. The *MVCA*, introduced by Chang and Leu [10], is a polynomial time heuristic which constructs a *MLST*. It starts with an empty graph and successively adds the label that covers the most uncovered nodes until all nodes are covered. This heuristic has the drawback that the constructed graph is not necessarily connected, even though all nodes are covered.

The *MVCA* version introduced by Krumke and Wirth [31] solves this problem by reducing greedily the number of connected components instead of the number of nodes. The algorithm starts with an empty graph and adds the label that minimizes the number of connected components most, until only

one component is left. If more than one label minimizes the components equally, one of them is selected randomly. This algorithm has a performance guarantee of $(1 + \log(n - 1))$. This bound was further improved by Wan et al. [48] to a $(1 + \log(n - 1))$-approximation. In 2005, Xiong et al. [51] showed that the worst case bound of *MVCA* is the b^{th}-harmonic number hb, if the label frequency is bounded by b. Another strategy is to use local search techniques. Brüggemann et al. [7] applied local search techniques that are based on *k-switch* neighborhoods to a restricted version of *MLST* problem, the so called $MLST_r$ problem. In the $MLST_r$ problem, every label occurs at most r times ($r \geq 2$) on the edges of G. With the *k-switch* neighborhood search technique, each *k-switch* replaces up to k colors from a feasible solution by other colors. Formally, the *k-switch* neighborhood is defined as follows:

Definition 2. Let $k \leq 1$ be an integer, and let $C1$, $C2$ be two feasible color sets for some instance of MLST. Then the set $C2$ is in the *k-switch* neighborhood of the set $C1$, if and only if $|C1 - C2| \leq k$ and $|C2 - C1| \leq k$ [7].

Thus the color set $C2$ can be derived from the color set $C1$ through the removal of up to k colors from $C1$, and then the addition of up to k colors. In the same paper, a proof is given showing that there exists an instance G of *MLST* and a spanning tree T for G that is a local optimum with respect to a *k-switch* neighborhood, such that the value of this local optimum is a $r/2 + \epsilon$ approximation of the optimal objective value. Other heuristic approaches to the *MLST* problem are *Genetic Algorithms (GA)*. These approaches outperform the *MVCA* in many cases. Xiong et al. [52] proposed a *GA* which encodes a candidate solution as a set of labels. A characteristic feature of this *GA* is that it uses only a single parameter p, which determines the population size. The fitness of an individual is the number of labels. The objective function minimizes the number of labels. Encoding only the labels is much easier than encoding the spanning tree. It is also sufficient, since all spanning trees induced by the labels are solutions for the *MLST* problem. The reason is that the structure of the tree does not matter; only the number of labels is significant.

For the initial population, the chromosomes are built by adding randomly selected labels (genes) until the solution is feasible. The crossover operator selects the labels in an offspring solution from the union of the parents. First, the union of the parents is built, then the labels are sorted in descending order of their frequency. The offspring is created by adding labels according their sort order until the solution is feasible. In the mutation step, one label is added and redundant labels are removed. This removal of redundant labels can also be seen as local search step. A time consuming operation in this algorithm is to validate a solution using *depth first searches (DFSs)*. A single *DFS* has the running time $O(m + n)$, where m denotes the number of edges and n denotes the number of nodes. Since *DFSs* are performed after each addition of a label, the running time for crossover and mutation is $O(l(m+n))$ each, where l denotes the number of labels. If p denotes the number of generations and the population size, p crossover and p mutation steps are performed in each generation. Thus, the worst case running time of the algorithm is $O(p^2 l(m + n))$.

Another, very similiar, *GA* was developed by Numella and Julstrom [39]. A characteristic of this *GA* is that a chromosome encodes all labels, and the fitness of a chromosome is the number of labels needed to build a feasible solution. Thus, the labels at the beginning of the chromosome encode a feasible solution. This feasible solution is called feasible set of the chromosome. In the crossover operation, called alternating crossover, the labels are alternatingly taken from the parent chromosomes and added to the offspring. If there are duplicates, the later occurrence of a label is eliminated. In the

mutation step, either two randomly chosen labels are swapped, or one label from the feasible set is swapped with a label from outside this set. A local search step tries to reduce the number of labels needed for a feasible solution by reordering the labels. The *GA* is 1−elitist, i.e. it always preserves the best chromosome.

Cerulli et al. applied the so called *Pilot* method to the *MLST* problem [9]. The *Pilot* method is a tempered greedy heuristic with additional lookahead results, so called pilots. It was developed by Duin and Voss [23]. Compared to other meta-heuristics like simulated annealing, variable neighborhood search and reactive tabu search, the *Pilot* method achieves better results in most cases. However, the running times are quite long. A *modified genetic algorithm (MGA)*, developed by Xiong et al.[53], lead to the best results for the *MLST* problem, regarding the solution quality and the running time.

A new approach, which compared favorably with existing heuristics, is to use ant colony optimization for the *MLST* [15]. There are few approaches to solving the *MLST* problem with mathematical programming techniques. A *MIP* formulation, which is based on a single commodity flow, was developed by Captivo [8]. A branch-and-cut-and-price approach was developed in [14], and several mathematical programming techniques where developed in [16].

As part of this project, the *k-MLSA* problem was solved using the exact methods *branch-and-cut (BC)* [18] and *branch-and-price (BP)* [46] as well as a heuristic method *GRASP* [22]. The *BCP* aproach integrates the *BC* and the *BP* approach. It uses a similiar *ILP* model and the same cuts as the *BC* method. This is described in detail in Chapter 5. The *GRASP* algorithm is compared with the delveloped *MA* in Chapter 4.

Regarding the application background the compression of fingerprints, Dietzel [22] gives a a detailed comparison of our approach to other compression techniques. A survey of the current state of the art in fingerprint recognition is given in [36]. Jain and Uludag present an application of watermarking and steganography where fingerprint data are hidden in an image [28]. A review of general data compression techniques is given in [43].

1.2 Outline of the Thesis

In the first part of this thesis, an algorithm for constructing the labels is developed. In the second part, the *k-MLSA* problem is solved with exact and heuristic methods. For the exact approach, the problem is modeled as a mixed integer program (MIP) and solved using a *branch-and-cut-and-price (BCP)* method. Extending the *BC* approach towards a *BCP* algorithm was motivated by the large number of labels in our problem. The *BCP* algorithm starts with a small subset of labels and adds new labels on demand in the pricing step. The *BCP* method is superior to *BC* and a *BP* because it can solve much larger instances.

The second chapter describes the compression approach in detail and introduces the datasets that were used for the tests. The third chapter deals with the construction of candidate template arcs T^c. After a short introduction to the structure of the problem, two different approaches for the generation of template arcs are presented. In the first approach, the set of all template arcs T^c is generated in advance, using an algorithm called *non-dominated interval search (NIS)*, which is explained in detail. In the second approach, template arcs are generated on demand to be used within the *BCP* framework presented in Chapter 5. The construction of template arcs on demand is done using a variant of *NIS* called *dynamic non-dominated interval search tree (DNIST)*. As *DNIST* is invoked very frequently in the pricing step, it has to be very efficient. It is also explained in detail. In the final part of the chapter

the results are presented and discussed. The fourth chapter presents a memetic algorithm *MA* for the solution of the *k-MLSA* problem. Besides giving a short general introduction to genetic algorithms and known techniques for solving the related *MLST* problem, the newly developed memetic algorithm is described in detail, and the results are presented and compared with a *GRASP* approach. In the menetic algorithm, feasible arborescences are searched for very frequently using depth first searches (DFS).

A technique to reduce the number of DFS calls is also introduced in Chapter 4. The fifth chapter presents a *BCP* approach to solving the *k-MLSA* problem. It gives a short general introduction to *BCP* and presents the ILP model as well as the cutting plane separation with cycle elimination cuts and directed connection inequalities. The directed connection cuts are separated by computing the maximum flow and can be improved by back-cuts and creep flow. After presenting the pricing model, the different variants of branch-and-cut-and-price (BCP) algorithms are described. Furthermore, preprocessing strategies with the *MA* as an initial heuristic and the computation of a lower bound for a reduced version of the problem are discussed. Also in this chapter, different *BCP* variants are compared based on computational experiments. Chapter 6 discusses some implementation details such as third party libraries, frameworks and solvers as well as parameter settings and the software design. In Chapter 7, results and possible further developments are discussed.

2. Formal Description of the Problem

This chapter describes the compression model as well as the structure of the input dataset. We give only a short illustration of the compression model, since a more detailed description is already given in [18].

2.1 Compression Model

Our input data set consists of n minutiae. Each minutia is interpreted as a d-dimensional point, i.e. as a vector $\vec{v} = \{v_1, \ldots, v_n\}$ from a discrete domain $\mathbb{D} = \{0, \ldots, \tilde{v}^1 - 1\} \times \ldots \times \{0, \ldots, \tilde{v}^d - 1\}$, $\mathbb{D} \subseteq \mathbb{N}^d$. The compression can be lossy or lossless. In the first case, a subset k of the n minutiae is selected, whilst in the second case all minutiae are used, i.e. $k = n$. Our compression approach is based on the following two ideas:

1. Select k points from n and connect them by a k-node arborescence: For this purpose we start with a complete directed graph $G = (V, A)$ with $A = \{(u, v) \mid u, v \in V, u \neq v\}$. Each node V of this graph corresponds to one of the n minutiae. In this graph a k-node arborescence is constructed, so each arc in the arborescence represents the relative geometric position of its endpoint relative to its starting point [18].

2. Use a codebook: We use a small codebook of template arcs, which are selected from the set of candidate template arcs T^c (see Chapter 4), and encode each arc relative to the most similiar template arc. The idea behind this strategy is that it should require less space to store the arcs as they can be encoded by their difference to their corresponding template arc. The difference of each arc to its template arc is expressed by a so called correction vector $\vec{\delta} \in \mathbb{D}'$ from a prespecified, small domain $\mathbb{D}' \subseteq \mathbb{D}$ with $\mathbb{D}' = \{0, \ldots, \tilde{\delta}^1 - 1\} \times \ldots \times \{0, \ldots, \tilde{\delta}^d - 1\}$ [18].

The objective of our optimization is to minimize the number of candidate template arcs by solving the resulting *k-MLSA* problem, while the size of the correction vector domain is prespecified. Note that other objectives are also possible and could be subject for further research: E.g., the size of the correction vector could be minimized while the number of candidate template arcs stays the same. Or, in a multi-objective optimization, the size of the correction vector as well as the number of template arcs (i.e. the size of the codebook) could be minimized. The result of our optimization is a minimum labeled spanning arborescence, and a solution consists of:

1. The codebook that contains the candidate template arcs T^c.

2. A rooted, outgoing tree $G_T = (V_T, A_T)$ with $V_T \subseteq V$ and $A_T \subseteq A$ connecting precisely $|V_T| = k$ nodes. Each arc of this tree is associated with a candidate template arc index $\kappa_{i,j} \in \{1, \ldots, m\}$ and a correction vector $\delta_{i,j} \in \mathbb{D}'$ [18].

Starting at the root of the tree we can derive every point from the relation of the source and target node v_i and v_j of each arc. So for each arc the relation

$$v_j = (v_i + t_{\kappa_{i,j}} + \delta_{i,j}) \bmod \tilde{v}, \qquad \forall (i, j) \in A_T, \tag{2.1.1}$$

holds, i.e. each arc v_j can be derived from v_i by adding a small correction vector $\delta_{i,j}$ to the index of the template arc $\kappa_{i,j}$ that corresponds to it [18]. We use a modulo-calculation so we do not have to deal with negative values and we avoid considering domain boundaries explicitly. For the modulo-calculation the domain border $\vec{\tilde{v}}$ is defined as $\tilde{v}^l = \max_{i=1,\ldots,n} v_i^l,\ l = 1,\ldots,d$. From this specific domain border, which is the maximum from one input instance, we have to distinguish the overall domain $\vec{\tilde{\xi}}^l$, $\tilde{\xi}^l = \max_I \tilde{v}_I^l,\ l = 1,\ldots,d$, which is the maximum over all input instances I.

The compressed information we have to store consists of the codebook, the correction vector $\vec{\tilde{\delta}}$ and the tree. We encode the structure of the tree as a bit string by traversing the arborescence depth first and storing 1 if an arc has been traversed in forward direction and 0 if it is traversed backwards. The following Equation (see Equation (2.1.2)) shows the complete compressed information on binary level.

$$\begin{aligned}\lambda(m,\vec{\delta},k,\vec{\tilde{v}},\vec{\tilde{\xi}},\vec{\chi}) \quad = \quad & size\,(\textsc{ConstData}') + \underbrace{2\cdot 7}_{\text{values } k,m} + \underbrace{2\cdot\sum_{l=1}^{d}\lceil \chi^l \operatorname{ld} \tilde{\xi}^l\rceil}_{\text{root node, domain } \vec{\tilde{v}}} \\ & + \underbrace{2\cdot(k-1)}_{\text{encoding of tree structure}} + \underbrace{\lceil m\cdot\sum_{l=1}^{d}\chi^l \operatorname{ld} \tilde{v}^l\rceil}_{\text{template arcs}} \\ & + (k-1)\cdot\lceil \underbrace{\operatorname{ld} m}_{\text{index to template arc}} + \underbrace{\sum_{l=1}^{d}\chi^l \operatorname{ld} \tilde{\delta}^l}_{\tilde{\delta}-\text{values}} + \underbrace{\sum_{l=1}^{\tilde{d}}(1-\chi^l)\operatorname{ld}\tilde{v}^l}_{\text{remaining dimensions}}\rceil\end{aligned} \tag{2.1.2}$$

Data of constant size are designated by ConstData, this variable contains for example the offset values for each dimension. For the number of nodes in the arborescence k and the number of template arcs m we reserve 2×7 bits. Not all dimensions of the vector are necessarily represented by a template arc and a correction vector. To exclude these dimensions we define the following function:

$$\chi^l = \begin{cases} 1 & \text{dimension } l \text{ is considered by the compression method} \\ 0 & \text{otherwise} \end{cases} \tag{2.1.3}$$

The third term of Equation (2.1.2) contains the bits reserved for the root node and the respective domain borders $\tilde{v}^l$. Term four contains the bits reserved for the bit-string to store the tree structure. Term five contains the bits that hold the template arcs, and term six contains the bits that hold the arcs. To represent each arc we need the index of the template arc. The space required to store these indices is ld m. Furthermore we have to store the correction vector and the remaining dimensions, which are not encoded by the correction vector. Note that we round up the whole fifth term of Equation (2.1.2), which saves bits compared to rounding up each term individually. To demonstrate the encoding we quote the complete encoding example from [18]. This example shows the encoding for two dimensions with $\vec{\tilde{\delta}} = (5,5)^T$, $k = 9$ and the following input:

$$\left\{\begin{pmatrix}200\\200\\21\end{pmatrix},\begin{pmatrix}208\\304\\30\end{pmatrix},\begin{pmatrix}211\\386\\97\end{pmatrix},\begin{pmatrix}261\\356\\210\end{pmatrix},\begin{pmatrix}313\\330\\293\end{pmatrix},\begin{pmatrix}314\\409\\22\end{pmatrix},\begin{pmatrix}503\\252\\268\end{pmatrix},\begin{pmatrix}608\\280\\157\end{pmatrix},\begin{pmatrix}414\\356\\77\end{pmatrix},\begin{pmatrix}662\\332\\104\end{pmatrix},\begin{pmatrix}702\\676\\78\end{pmatrix}\right\}.$$

The offsets are 200 for the first and second coordinate and the domain borders are $\vec{\tilde{v}} = (503, 477, 294)^T$.

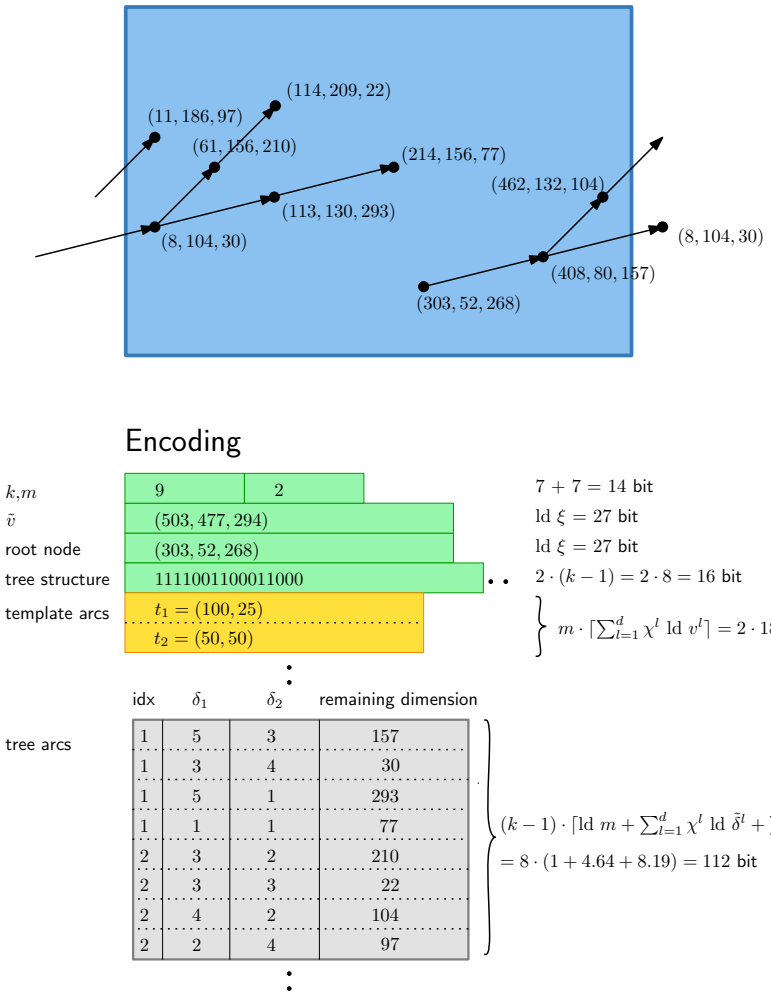

Figure 1: This figure shows a concrete encoding example. The first block basically contains information to be able to process the following blocks. It is followed by the list of the template arcs. This can be compared to a dictionary or codebook of traditional compression methods. The block on the bottom contains the actual tree information, i.e. a list of arcs encoded by an index to one of the template arcs, the respective correction vectors, and finally the values of the dimensions which are not considered for compression. The black dots indicate that the size of the respective (sub)blocks is not known in advance, because it depends on output values of the compression algorithm (like the number of template arcs m). Caption and image source [18].

2.1.1 The Set of Candidate Template Arcs T^c

The set of candidate template arcs T^c is now described in more detail, since its construction will play an important role in this thesis. Each arc $(i, j) \in A$ represents the geometric information of a d-dimensional vector a, where a^l denotes the coordinate of dimension l of that vector and $l = 1, \ldots, d$.

Let $A' \subseteq A,\ A' \neq \emptyset$ be some subset of arcs from A. A set A' is dominated by a set A'' if $A' \subset A''$. We want to determine all different non-dominated subsets of A that can be represented together by a common template arc $t \in \mathbb{D}$. A template arc t can represent all arcs within the domain of the predefined correction vector $\mathbb{D}' \subseteq \mathbb{D}$ (see Figure 2). Recall that we use a modulo structure instead of subtraction, so each arc can be represented by adding t to the respective correction vector $\tilde{\delta}$. Thus, each t can represent the following subspace $D(t) \subseteq \mathbb{D}$:

$$D(t) = \{t^1, \ldots, (t^1 + \tilde{\delta}^1 - 1) \bmod \tilde{v}^1\} \times \ldots \times \{t^d, \ldots, (t^d + \tilde{\delta}^d - 1) \bmod \tilde{v}^d\}. \tag{2.1.4}$$

The set $A' \subset A$ may be representable by several different template arcs t, one of which is chosen as the so called standard template arc $\tau(A')$ by selecting the smallest coordinate a^l from $E \subset A'$ in each dimension l where E comprises only arcs in A' that are reachable without crossing the domain border.

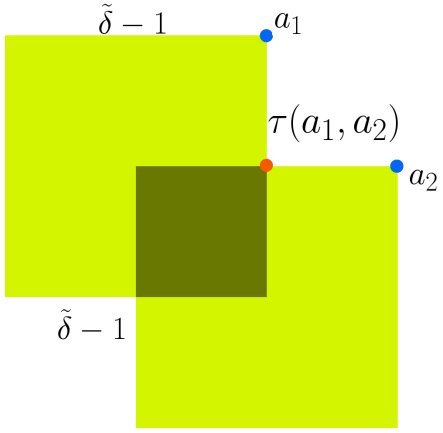

Figure 2: Each point within the light-green areas anchored at a_1 and a_2 can represent a_1 and a_2 respectively. Each point within the dark green area can represent both a_1 and a_2 together. We designate the point with the greatest dimension values within the dark green area as $\tau^l(a_1, a_2)$ (the red dot in the upper right corner of the dark green area). That point is also the one whose dimension values coincide with the smallest dimension values from the set of the points it represents (a_1 and a_2 in this case). This figure does not show the special case of crossing the domain border.

Figure 2 shows how the position of the standard template arc is chosen. Although all template arcs in the dark green area can represent both arcs the standard template arc is set in the upper right corner. A standard template arc $\tau(A')$ is dominated by another standard template arc $\tau(A'')$ if $A' \subset A''$. The number of this non-dominated template arcs can become very high. An upper bound of the number of non-dominated template arcs $|T^c|$ is given by $O(|A|^d)$, as illustrated in [18]. Figure 2.1.1 demonstrates the construction of this upper bound. Bold dots represent the vector set A, small dots the non-dominated standard template arcs T^c. Obviously, $|T^c| = (|A|/4 + 1)^2 = \Theta(|A|^2)$ [18].

2.2 Test Instances

We used test instances from the Fraunhofer Institute Berlin (*Fraunhofer Templates*) and the U.S. National Institute of Standards and Technology [24] (*NIST* data). The test instances are described in more detail in [18].

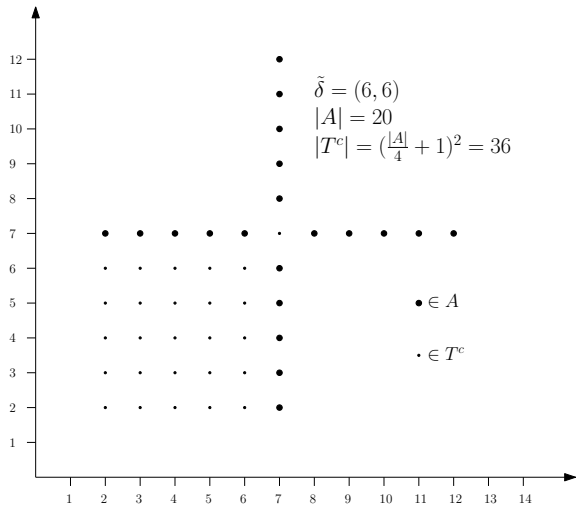

Figure 3: Example for $|T^c| = \Theta(|A|^d)$ with $d = 2$. Image credits to [18].

Table 1: Overview about the *Fraunhofer* test instances used for our experiments.

short name	**full name**	$\|V\|$	**short name**	**full name**	$\|V\|$
ft-01	P0001_F00_R00	31	ft-11	P0001_F03_R00	38
ft-02	P0001_F00_R01	38	ft-12	P0001_F03_R01	28
ft-03	P0001_F00_R02	35	ft-13	P0001_F03_R02	25
ft-04	P0001_F00_R03	20	ft-14	P0001_F03_R03	33
ft-05	P0001_F00_R04	39	ft-15	P0001_F03_R04	29
ft-06	P0001_F01_R00	15	ft-16	P0014_F00_R00	37
ft-07	P0001_F01_R01	28	ft-17	P0014_F00_R01	31
ft-08	P0001_F01_R02	27	ft-18	P0014_F00_R02	40
ft-09	P0001_F01_R03	27	ft-19	P0014_F00_R03	35
ft-10	P0001_F01_R04	31	ft-20	P0014_F00_R04	28

The Fraunhofer templates are multiple scans of four different fingers from two individuals, and each of them contains 15 to 40 minutiae. Each minutia is encoded as $\vec{\tilde{\xi}} = (\xi_x, \xi_y, \xi_\theta, \xi_{\text{type}})^T = (2^9, 2^9, 2^9, 2^1)^T$. The size of ConstData is 14 bits, i.e. 7 bits for the offset value of the respective spatial dimension. Group *NIST* contains 12 exemplary templates from each of the the three classification categories *ugly*, *bad* and *good*. Each minutia is encoded with resolution $\vec{\tilde{\xi}} = (\xi_x, \xi_y, \xi_\theta, \xi_{\text{type}})^T = (2^{12}, 2^{12}, 2^9, 2^1)^T$. The size of ConstData is 33, which is the offset of the respective dimensions x, y and θ. Each instance contains between 53 and 120 minutiae. The compression ratio ρ is computed as follows:

$$\rho\,[\%] = 100 - \frac{100 \cdot \lambda_{\text{raw}}}{\lambda_{\text{enc}}}, \tag{2.2.1}$$

where λ_{raw} is the size (in bits) of the original raw data:

$$\lambda_{\text{raw}} = \textit{size}(\text{ConstData}) + n \cdot \sum_{l=1}^{d} \lceil \text{ld}\, \tilde{\xi}^l \rceil. \tag{2.2.2}$$

Table 2: Overview about the *NIST* test instances used for our experiments.

short name	**full name**	$\|V\|$	**short name**	**full name**	$\|V\|$	**short name**	**full name**	$\|V\|$
nist-u-01-t	u201t6i	99	nist-b-01-t	b101t9i	106	nist-g-01-t	g001t2i	99
nist-u-02-t	u202t8i	93	nist-b-02-t	b102t0i	94	nist-g-02-t	g002t3i	101
nist-u-03-t	u204t2i	100	nist-b-03-t	b104t8i	107	nist-g-03-t	g003t8i	102
nist-u-04-t	u205t4i	84	nist-b-04-t	b105t2i	81	nist-g-04-t	g004t8i	120
nist-u-05-t	u206t3i	72	nist-b-05-t	b106t8i	93	nist-g-05-t	g005t8i	80
nists-u-06-t	u299t8i	70	nists-b-06-t	b117t0i	77	nists-g-06-t	g013t4i	87
nists-u-07-t	u298t2i	63	nists-b-07-t	b118t8i	82	nists-g-07-t	g012t4i	74
nists-u-08-t	u277t9i	74	nists-b-08-t	b119t0i	76	nists-g-08-t	g030t1i	99
nists-u-09-t	u274t9i	69	nists-b-09-t	b120t9i	76	nists-g-09-t	g033t7i	53
nists-u-10-t	u267t3i	76	nists-b-10-t	b124t5i	63	nists-g-10-t	g042t2i	76
nists-u-11-t	u249t3i	63	nists-b-11-t	b129t7i	80	nists-g-11-t	g045t9i	67
nists-u-12-t	u232t4i	80	nists-b-12-t	b132t7i	85	nists-g-12-t	g050t7i	84

The size (in bits) of the compressed data λ_{enc} is computed as shown before (see Section 2.1.2). Note that ρ is the compression ratio of the k selected points, not that of all points.

3. Creating the Candidate Template Arcs T^c

This chapter deals with the construction of candidate template arcs T^c. After a short introduction to the structure of the problem, two different approaches for the generation of template arcs are presented. In the first approach, the set of all template arcs T^c is generated in advance, using an algorithm called *non dominated interval search (NIS)*, which is explained in detail. In the second approach, template arcs are generated on demand to be used within the *BCP* framework presented in Chapter 5. The construction of template arcs on demand is done using a variant of *NIS* called *dynamic non-dominated interval search tree (DNIST)*. As *DNIST* is invoked very frequently in the pricing step, it has to be very efficient. It is also explained in detail. In the final part of the chapter the results are presented and discussed.

3.1 Introduction

Prior to running the optimization algorithm candidate template arcs T^c have to be generated from the input dataset, i.e. from the minutiae. The optimization algorithm then computes a subset T of T^c, called the codebook, by solving a *k-node minimum label spanning arborescence* problem. Recall the following from Chapter 2.1.1: The set T^c contains all non-dominated template arcs. A template arc t can represent all arcs within the domain of the predefined correction vector $\vec{\delta}$, i.e. within the domain $(t^l + \tilde{\delta}^l - 1) \bmod \tilde{v}$ where $l = 1, \ldots, d$. Each arc $(i, j) \in A$ represents the geometric information of a d-dimensional vector a, where a^l denotes dimension l of that vector and $l = 1, \ldots, d$. Constructing the set of all non-dominated template arcs can also be seen as the determination of all different non-dominated subsets of A that can be represented together by a common template arc t.

Two different approaches are used for the generation of T^c. The first approach is to create the whole set T^c in advance in a preprocessing step. The second approach is to generate the candidate template arcs on demand by solving a pricing problem. Our first approach to compute T^c, called *non-dominated interval search (NIS)*, replaces an older method that was slow and could not generate more than about 40000 candidate template arcs (see [18]). To be applicable in practice, the entire compression run, including preprocessing as well as the heuristic or exact optimization procedure, should not take more than a few minutes. The geometric structure of our problem, in which each candidate template arc represents a particular interval in each dimension, seems to suggest that the well known interval tree [20] might be applicable. However, our problem does not benefit from a balanced tree since we do not perform range search. We have two uses cases. In the first one, *NIS* is used to construct all template arcs as part of the preprocessing step. A similarity to the interval tree is that we take all points reachable to the left and right from a so called center point. However *NIS* chooses the position of the center points quite differently as will be explained shortly. The second use case is to generate template arcs on demand, to be used within the *BCP* framework already mentioned in Chapter 1 and to be discussed in more detail in Chapter 5. For this purpose, we have developed a variant of *NIS*, which we call *dynamic non dominated interval search tree (DNIST)*. Contrary to *NIS*, *DNIST* partially retains the state of the recursion in the form of a tree. On-demand retrieval does not benefit from a balanced tree either, because what is requested is the template arc with the greatest value (see Section 3.3) and not template arcs within a particular interval.

3.2 Non-dominated Interval Search (NIS)

The algorithm *NIS* computes all non-dominated candidate template arcs T^c. It avoids building dominated arc sets from the outset, and therefore only has to deal with a small subset of all possible template arcs. The algorithm comprises two phases which are not executed strictly one after the other but interleaved as part of a recursive procedure that can be seen as a depth first tree traversal. We start with the set of all arcs. Each phase cycles through all requested dimensions once and divides the sets in non-disjoint subsets according to the respective dimension. A distinctive feature of the algorithm is that it can decide whether a set is dominated using the geometric position of the arcs. If a set is dominated, it is not built at all.

In phase one, sets are built that contain at least one element representable together with all other elements in respect of each dimension. These sets overlap only their neighboring sets. In phase two, the sets are divided further, so that all elements contained in one set are representable together regarding the respective dimension. In phase two, this is done by traversing the arcs, which are sorted in ascending order in descending order and collecting all arcs that lie within the domain of the correction vector $\vec{d}$ minus one. Phase two also divides the sets according to each dimension, so the elements are representable together regarding each of the requested dimensions at the end of phase two. These elements can therefore be represented by the same template arc. Since we never build dominated sets, we do not have to decide if a template arc is dominated at the end of phase two.

The goal of phase one is to reduce the number of possible combinations of arcs before building all sets of arcs that contain only elements that are representable together, of which there is a much greater number. More formally, we can describe the construction of subsets as follows:

Let S_x^r denote an arc set where $r = 0, \ldots, 2d$ denotes the depth of the recursion, d denotes the number of requested dimensions l and x denotes the arc set number. The set of all arcs is $S_1^0 = A$. In each recursion the set of arcs S_x^r is divided in up to $|S_x^r|$ further not disjoint subsets $S_y^{r+1} \subset S_x^r$, where $y = 1 \cdots |S_x^r|$ such that no subset is dominated by any other subset. Note the difference between recursion depth and dimension: Since we cycle through all requested dimensions twice the maximum recursion depth is $2d$ In the following Subsections 3.2.1 and 3.2.2 phase one and phase two are described in detail. After that, we describe how dominated sets are detected (see Subsection 3.2.3). At the end of Subsection 3.2.3 algorithms as well as examples for the whole procedure are presented.

3.2.1 Phase One

The size of each interval I in phase one is chosen as follows: We choose an initial arc on the left (assuming arcs are arranged left to right in ascending order of the respective dimension) and call it center arc $a_{c_1}^l$. The interval is chosen so that the center arc can potentially be represented by a candidate template arc together with either all arcs on its left or all arcs on its right.

A center arc can be represented all arc within the interval $[\max(0, (a_{c_1}^l - \tilde{\delta}^l + 1)), (a_{c_1}^l + \tilde{\delta}^l - 1) \bmod \tilde{v}^l]$, $\forall l = 1, \ldots, d$. The first arc outside this interval $a^l > (a_{c_1}^l + \tilde{\delta}^l - 1) \bmod \tilde{v}^l$ is chosen as the next center arc $a_{c_2}^l$ and we build the next interval around it. The advantage of this partitioning approach is that only neighboring intervals can overlap and hence only arcs within such neighboring intervals can possibly be covered by the same template arc. Note that due to the ring structure the first and the last intervals are neighbors and can overlap. We start in the first dimension, $l = 1, r = 1$, with the set of all arcs, $S_1^0 = A$, sorted in ascending order of their value in that dimension: $a_1^l \leq a_2^l \leq \cdots \leq a_{|A|}^l$. Now this set is divided into overlapping intervals, I_x^l, which differ in at least one arc. However, the algorithm

proceeds depth first, i.e. before the second interval of the current dimension is computed, the first interval is processed recursively, going through both phases.

The first arc set $S_1^1 \subset A$ is built from arcs within the first interval in dimension 1, I_1^1, and the arcs it contains are sorted according to the next dimension ($(l+1) \bmod d$). Then the arc set $S_1^2 \subset S_1^1$ is built from arcs within the first interval I_1^2 in dimension $l+1$ and sorted according to the following dimension. This routine is repeated until we reach the last dimension, d, at which point the first set $S_1^r \subset S_1^{r-1}$ enters phase two where the actual template arcs are built. Note that the arcs collected during the final recursion of phase one are sorted by dimension 1 in preparation for phase two (see below).

Running time in phase one: Since only neighboring intervals overlap, the greatest number of arcs we may have to traverse in each arc set is $2 \cdot |S_x^{r-1}|$. Consequently, the number of arcs can only double in each recursion step. Using balanced binary search trees for traversal and sorting of arcs generally leads to logarithmic complexity. Thus, the running time for traversal and sorting is $O(|A|\log(|A|))$ and an upper bound for the running time of phase one is $O(2^d \cdot |A|\log(A))$.

3.2.1.1 Determining the First and the Last Center Arc

As a consequence of the ring structure the first and the last center arc have to be chosen differently, depending on whether the first and the last interval are overlapping (Case 2) or not (Case 1).

Definition 3. (First and last center arc) Case 1: Arcs are not representable across the domain border $((a^l_{|S^{r-1}|} + \tilde{\delta}^l - 1) \bmod \tilde{v}^l) < a^l_1$, regarding the respective dimension l. The first center arc $a^l_{c_1}$ is the greatest arc that can be represented together with all smaller arcs $(a^l_{c_1} - \tilde{\delta}^l + 1) \leq a^l_1$. Let C denote the set of center arcs from the respective dimension l. The last center arc $a^l_{c_{|C|}}$ is the smallest arc that can be represented together with all greater arcs $(a^l_{c_{|C|}} + \tilde{\delta}^l - 1) \geq a^l_{|S^{r-1}|}$. Case 2: Arcs are representable across the domain border $((a^l_{|S^{r-1}|} + \tilde{\delta}^l - 1) \bmod \tilde{v}^l) \geq a^l_1$, regarding the respective dimension l. The first center arc is the first arc that cannot be represented together with the last center arc $a^l_{c_1} > (a^l_{c_{|C|}} + \tilde{\delta}^l - 1) \bmod \tilde{v}^l$. The last center arc is the greatest arc $a^l_{|S^{r-1}|}$.

In Case 2 we always choose the greatest arc $a^l_{|S^{r-1}|}$ as the last center arc, because for reasons explained in Section 3.2.3, we have to know how far the last interval reaches when we process the first interval. Due to this choice, it is possible that not only the last, but also the penultimate interval overlaps with the first interval, which is the single exception to the rule that only neighboring intervals can overlap.

3.2.2 Phase Two

In the second phase we also build sets using depth first traversal. In contrast to phase one, we now generate all possible non-dominated intervals of arcs that can be represented together, not just intervals around a particular center arc. All arcs have to be representable together in terms of all dimensions, because in phase two the actual candidate template arcs are generated. Therefore we start in dimension 1 and iterate through all requested dimensions again.

Phase two starts with the arc set S_x^r constructed in phase one, where $r = d$. The greatest interval between two arcs of this set is $2 \cdot (\tilde{\delta}^l - 1)$ for all dimensions l. Again beginning with dimension 1, this set is further divided into subsets where the greatest interval between two arcs contained in a subset is

$(\tilde{\delta}^l - 1)$. Note the relationship between the dimension l and the depth of the recursion r. The dimension counter l runs from $1, ..., d$ in each phase whereas r is increased throughout the entire recursion taking on the values $0, ..., 2d$ and $r = 1$ in dimension 1. Hence, at the beginning of phase two, $l = 1, r = d + 1$ and the current arc set is sorted by dimension 1.

In phase two the arcs are traversed in descending order. We choose an initial arc, which we call start arc $a^l_{s_1}$ and add all arcs within the interval $(a^l_{s_1} - \tilde{\delta}^l + 1) \bmod \tilde{v}^l$ to the set S^r_1. Unlike before, we choose as the next start arc $a^l_{s_2}$ the greatest arc that can reach at least one arc a^l_i not covered by the previous interval $[(a^l_i < (a^l_{s_1} - \tilde{\delta}^l + 1) \bmod \tilde{v}^l), (a^l_i \geq (a^l_{s_2} - \tilde{\delta}^l + 1) \bmod \tilde{v}^l)]$. We repeat this depth first traversal for all dimensions. Once the last dimension is reached, all arcs within a set S^r_x are representable by a common template arc $\tau(S^r_x)$ with regard to all dimensions. The interval construction in phase two is shown in Figure 6 for dimension 1 and in Figure 7 for dimension 2.

Now the purpose of phase one becomes obvious. In phase one, the number of intervals in each dimension is much smaller than in phase two, because the intervals are only built in reference to the center arc. Phase one divided s many sets of arcs into subsets that are within an interval in terms of their previous but not their later dimensions. So these sets never make it to phase two where all possible intervals are built, which is a much greater number.

If, for example, some arcs are within interval one in dimensions 1 and 2, and therefore in the same set S^2_1, but each of them is in a separate interval in dimension 3 , the set S^2_1 is further divided into $|S^2_1|$ different sets in dimension 3. Assume the number of dimensions is three and we now enter phase two. Phase two does not get one large set of size $|S^2_1|$, but $|S^2_1|$ different sets, each of size one, so no further sets are built. Thus, as a result of phase one, we have to build up to $|S^2_1|^{d-1}$ fewer sets in phase two.

Without phase one a worst case scenario would be to build $|S^2_1|^{d-1}$ sets in dimension 1 (recursion level 4) and 2 (recursion level 5) before they are finally all divided in subsets, each containing one element in dimension 3 (recursion level 6). Running time in phase two: Let $\mathcal{I}$ denote the set of intervals within a dimension. An upper bound for the traversal and insertion into each arc set is $O(|\mathcal{I}| \cdot |A| \log(|A|))$.

How many intervals can be built in phase two? The intervals in phase two have the size $2 \cdot (\tilde{\delta}^l - 1)$ for all dimensions l. Since each interval must differ in at least one arc, and the arc values in each dimension are integers, the maximal number of intervals in each dimension is $\tilde{\delta}^l - 1$. Let δ_{rmmax} be the maximal correction vector $\delta_{rmmax} = \max\ \tilde{\delta}^l,\ l = 1, \ldots, d$. Hence, the running time for phase two is $O(\delta^d_{\max} \cdot |A| \log |A|)$.

3.2.2.1 Determining the First Start Arc

Recall that in phase two the arcs are traversed in descending order. In contrast to phase one we do not generate intervals around one center arc, but collect all arcs that are representable together beginning with a so called start arc.

Definition 4. (First start Arc) Case 1: Arcs are not representable across the domain border $((a^l_{|S^{r-1}|} + \tilde{\delta}^l - 1) \bmod \tilde{v}^l) < a^l_1$. The first start arc $a^l_{s_1}$ is simply the last arc $a^l_{|S^{r-1}|}$. Case 2: Arcs are representable across the domain border $((a^l_{|S^{r-1}|} + \tilde{\delta}^l - 1) \bmod \tilde{v}^l) \geq a^l_1$. The first start arc is the greatest arc that cannot be represented together with the smallest arc $a^l_{s_1} < (a^l_1 - \tilde{\delta}^l + 1 + \tilde{v}^l)$.

We do not have to define a last start arc explicitly, since we build intervals as long as we can reach arcs not covered by the previous interval.

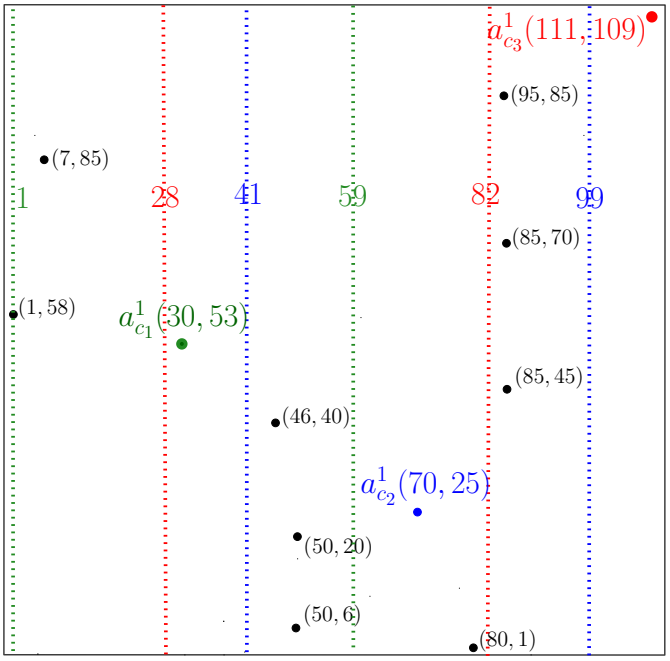

Figure 4: Phase one: interval construction in dimension 1.

3.2.2.2 Example

In the following example, phase one and two are demonstrated. Since this example contains no new information it can be skipped. We demonstrate the construction of the intervals on the basis of the following example which we will use throughout this chapter: Let $\vec{v} = (112, 110)^T$, $\vec{\delta} = (30, 30)^T$.

$$A = \left\{a_0 = \begin{pmatrix}1\\58\end{pmatrix}, a_1 = \begin{pmatrix}7\\85\end{pmatrix}, a_2 = \begin{pmatrix}30\\53\end{pmatrix}, a_3 = \begin{pmatrix}46\\40\end{pmatrix}, a_4 = \begin{pmatrix}50\\20\end{pmatrix}, a_5 = \begin{pmatrix}50\\6\end{pmatrix}, a_6 = \begin{pmatrix}70\\25\end{pmatrix}, a_7 = \begin{pmatrix}80\\1\end{pmatrix}, a_8 = \begin{pmatrix}85\\45\end{pmatrix}, a_9 = \begin{pmatrix}85\\70\end{pmatrix}, a_{10} = \begin{pmatrix}95\\85\end{pmatrix}, a_{11} = \begin{pmatrix}111\\109\end{pmatrix}\right\}.$$

In the following examples, arcs are denoted by their index number.

The interval construction in phases one and two is demonstrated on the basis of this example and shown in figures 4, 5, 6 and 7. Interval construction in phase one: In phase one sets of arcs are built that contain at least one element (the center arc) that is representable together with each other arc in the interval. Figure 4 shows the construction of intervals in dimension 1. The first center arc $a^1_{c_1} = 30$ since this is the first arc outside the previous interval $I^1_{last} = [82, 28]$, which is delimited by the red dotted lines. As explained before, if the last interval overlaps the first interval, we always designate the final arc as last center arc (see Paragraph 3.2.1.1 case 2). The first center arc $a^1_{c_1} = 30$ can be represented together with all arcs within the interval $I^1_1 = [a^1_{c_1} - \tilde{\delta}^1 + 1, a^1_{c_1} + \tilde{\delta}^1 - 1] = [1, 59]$. This interval is delimited by the green dotted lines and contains the arc set $\{0, 1, 2, 3, 4, 5\}$. The second center arc $a^1_{c_2} = 70$ is the first arc located outside the first interval. The second interval $I^1_2 = [41, 99]$ is delimited by the blue dotted lines and contains the arc set $\{3, 4, 5, 6, 7, 8, 9, 10\}$. The first arc outside the second interval is also the final arc $a^1_{c_1} = 111$. The third interval $I^1_2 = [82, 28]$ is delimited by the red dotted lines and contains the arc set $\{0, 1, 10, 11, 9, 8\}$.

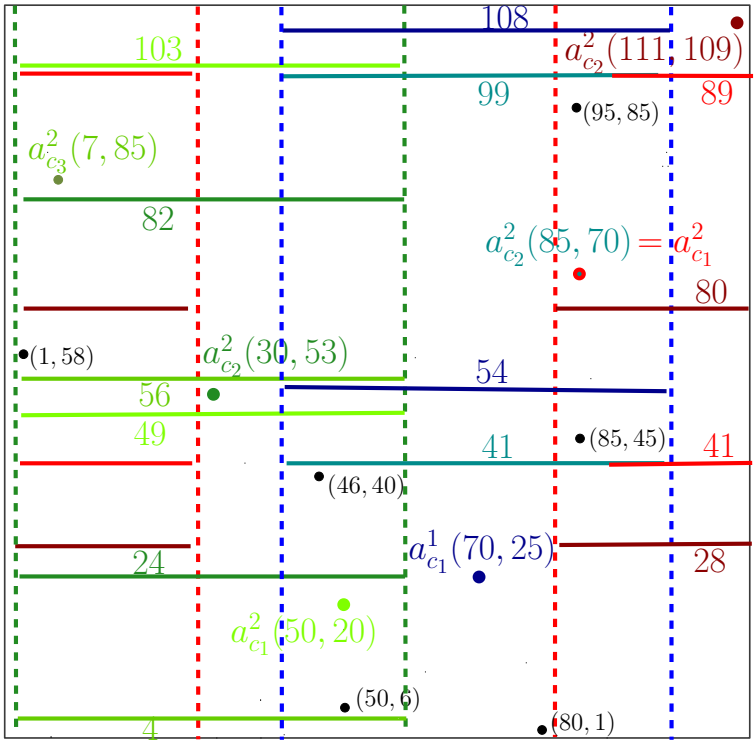

Figure 5: Phase one: interval construction in dimension 2

The intervals built in dimension 1 are further partitioned in dimension 2. Figure 5 shows the construction of intervals in dimension 2.

We start with the first interval from dimension 1, delimited by the green dashed lines, which contains the arc set $\{0, 1, 2, 3, 4, 5\}$. The first center arc within this interval is $a^2_{c_1} = 20$. It spans the interval $I^2_1 = [103, 49]$, delimited by the light green lines. This interval contains the arc set $\{3, 4, 5\}$. The first arc outside this interval is $a^2_{c_2} = 53$. So the second interval is $I^2_2 = [24, 82]$. It contains the arc set $\{2, 0, 3\}$ and is delimited by the dark green lines. The next center arc $a^2_{c_3} = 85$ so the third interval is $I^2_3 = [56, 4]$ it contains the arc set $\{1\}$ and is delimited by the green lines. The next interval constructed in dimension 1 is delimited by the blue dashed lines and contains the arc set $\{3, 4, 5, 6, 7, 8, 9, 10\}$. It is further divided into interval $I^2_1 = [108, 54]$, delimited by the dark blue lines. It has the center arc $a^2_{c_1} = 25$ and contains the arc set $\{3, 4, 5, 6, 7, 8\}$. The second interval $I^2_2 = [41, 99]$ is delimited by the cyan lines. It has the center arc $a^2_{c_2} = 70$ and contains the arc set $\{8, 9, 10\}$. The next interval from dimension 1 is delimited by the red dashed lines and contains the arc set $\{8, 9, 10, 11, 1, 0\}$. The first center arc is $a^2_{c_1} = 70$. It spans the interval $I^2_1 = [41, 89]$, delimited by the red lines. This interval contains the arc set $\{8, 9, 10, 1, 0\}$. The second center arc is $a^2_{c_2} = 109$. It spans interval $I^2_2 = [111, 90]$, which is delimited by the dark red lines and contains the arc set $\{0, 11, 1\}$.

Interval construction in phase two: In phase two sets that contain only elements that are all representable together are constructed. We cycle again through all dimensions and at the end of phase two all elements from a set are representable by one template arc. Figures 6 and 7 do not show all

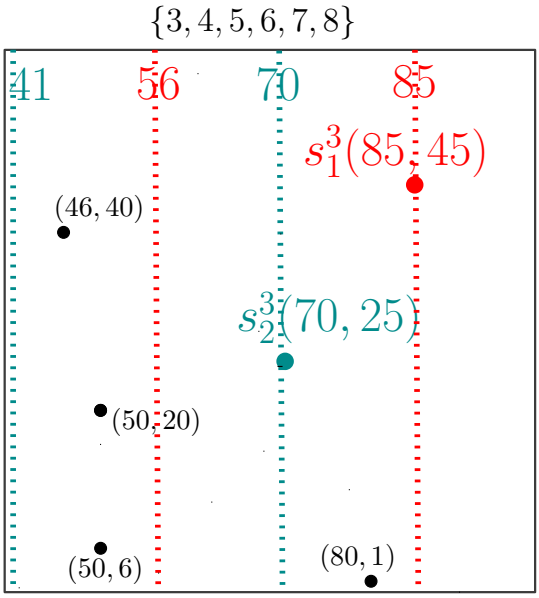

Figure 6: Phase Two: interval construction in dimension 1

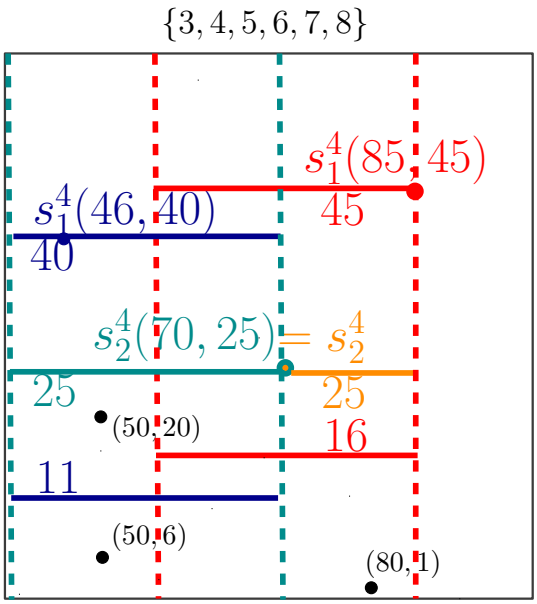

Figure 7: Phase Two: interval construction in dimension 2

intervals but demonstrate phase two with the arc set $\{3, 4, 5, 6, 7, 8\}$, i.e. the intervals $I_2^1 = [41, 99]$ and $I_1^2 = [108, 54]$. Note that we do not show the part of interval $I_1^2 = [108, 54]$ across the domain border because within this part $[108, 110]$ lie no arcs. The arcs are now traversed in descending order.

Figure 6 shows the interval construction in dimension 1. The first start arc is $a_{s_1}^3 = 85$ (see Section 3.2.2.1 case 1). This start arc can represent the arcs within the interval $I_1^3 : [a_{s_1}^1, a_{s_1}^1 - \tilde{\delta}^1 + 1] = [85, 56]$ which are the arcs $\{6, 7, 8\}$. This interval is delimited by the red dotted lines. The next arc outside $I_1^3 = [85, 56]$ is arc $5 = (50, 6)$. The next arc that can reach it and therefore the second start arc is $a_{s_2}^3 = 70$ with the interval $I_2^3 : [70, 41]$. This interval is delimited by the blue doted lines and contains the arc set $\{3, 4, 5, 6\}$. Regarding the first dimension all arcs within one interval are now representable by one template arc.

Figure 7 shows the interval construction in dimension 2. In dimension 2 we start with interval $I_1^3 = [85, 56]$ from dimension 1 which is delimited by the red dashed lines and contains the arc set $\{6, 7, 8\}$. The first start arc is $a_{s_1}^4 = 45$ with interval $I_1^4 = [45, 16]$. The interval is delimited by the red lines and contains arc set $\{8, 6\}$. The arcs in this set can be represented together regarding all dimension, so we construct the template arc $t(70, 25)$. The next start arc is $a_{s_2}^4 = 25$, since it can also represent arc $7 = (80, 1)$. It spans the interval $I_2^4 = [106, 25]$ which is deliminated by the orange line. The constructed template arc is $t(60, 1)$. The second interval from dimension 1 is $I_2^3 = [70, 41]$. It is delimited by the cyan dashed lines and contains the arc set $\{3, 4, 5, 6\}$. Start arc is $a_{s_1}^4 = 40$, it spans the interval $I_1^4 = [40, 11]$, delimited by the dark blue lines and contains the arc set $\{3, 6, 4\}$. The constructed template arc is $t(46, 20)$. The next start arc $a_{s_2}^4 = 25$, since it can also represent arc $5 = (50, 6)$. The interval is $I_2^4 = [106, 25]$ delimited by the cyan lines and contains the arc set $\{4, 5, 6\}$. The constructed template arc is $t(50, 6)$.

3.2.3 How to Avoid Dominated Sets

Avoiding to build dominated sets reduces the complexity of the algorithm significantly.

Definition 5. (Dominance of an interval) Let $A(I_x^l)$ denote the set of arcs A an interval I_x^l can represent. The interval I_x^l is dominated by interval I_{x+1}^l if and only if $A(I_{x+1}^l) \subset A(I_x^l)$.

Intervals are never dominated in the dimension in which they are constructed as every interval must differ in at least one arc in order to be constructed at all. Therefore, a set is never dominated in the dimension in which it is built. The problem arises on later recursion levels when previously unique sets become dominated because arcs that defined their difference are cut off.

The following example illustrates this situation. In dimension 1 we have the sets $A_1^1 = \{1, 2, 3, 4\}$ and $B_1^1 = \{2, 3, 4, 5\}$. Neither is dominated. In dimension 2, the sets are divided further into $A_1^2 = \{1, 2\}$, $A_2^2 = \{3, 4\}$, $B_1^2 = \{2, 3, 4\}$ and $B_2^2 = \{4, 5\}$. As we can see, set A_2^2 has become dominated and should not be built. Obviously, a set is dominated by its ancestors as each set is a subset of its parent. This form of dominance does not matter, since our goal is to construct non dominated template arcs in the last recursion step. If we associate each set with the node of a tree, the sets corresponding to a node n or the descendants of n must not be dominated by sets corresponding to siblings or descendants of siblings of n. Since sets are never dominated in the dimension in which they are built, sets corresponding to a node n are never dominated by sets corresponding to siblings of n. Thus we only have to check for the sets corresponding to descendants of a node n if they have become dominated by sets associated to nodes in a branch of the tree that has its origin in a sibling of n.

3.2.3.1 Dominated Sets in Phase One

In order to detect which sets have become dominated in phase one, we store in each recursion r, where $r = 1, ..., d$ for the respective dimension l, where $l = 1, ...d$ the successor s^r and the predecessor p^r. The successor s^r is the smallest integer that can be reached from the next center arc $a_{c_{i+1}}$. The predecessor p^r is the greatest integer that can be reached form the previous center arc $a_{c_{i-1}}$.

Note that the recursion level denotes the depth of the tree, while the dimension value of the arc is compared to the predecessor and the successor. Since in phase one all dimensions are traversed for the first time, the recursion level and the dimension value are the same. Assuming we are currently in interval I_x^r, with center arc a_{c_x}, we can define predecessor and successor as follows:

Definition 6. (Predecessor and Successor)

1. $p^r = (a^r_{c_{x-1}} + \tilde{\delta}^r - 1) \bmod \tilde{v}^r$

2. $s^r = (a^r_{c_{x+1}} - \tilde{\delta}^r + 1)$

Lemma 1. *If a set contains at least one arc $a^r < s^r$ and at least one arc $a^r > p^r$, it is not dominated with regard to recursion level r, where $r = 1, \ldots, d$.*

It is not necessarily the same arc that has to be greater than the predecessor and smaller than the successor. If it is the same arc, that arc lies in the interval $a^r \in (p^r, s^r)$ and we know for sure that this arc a^r cannot be represented by any other set (except of course ancestors of the set. In the rest of this

text we will not mention this trivial fact again). We call the condition $a^r \in (p^r, s^r)$ a strong condition and the condition from Lemma 1 a weak condition. If only the weak condition is fulfilled, the interval contains a combination of arcs, with one arc $a^r < s^r$ and one arc $a^r > p^r$ that can not be represented by any other set.

E.g., In recursion 1, we have the sets $A_1^1 = \{1, 2\}$ and $B_1^1 = \{2, 3, 4, 5\}$ and $C_1^1 = \{4, 5\}$. Assume that in recursion 2 the sets are divided further and set $B_1^2 = \{2, 3, 4\}$. Set B_1^2 fulfills the strong condition, since it contains arc 3, which is in interval (p^1, s^1). Arc 3 is not represented by any other set since it is neither reachable from the preceding nor from the subsequent interval. Assume on the other hand that B_1^2 does not contain arc 3 but $B_1^2 = \{2, 4\}$. In that case, the weak condition is fulfilled and B_1^2 contains a combination of arcs that is not represented by any other set.

3.2.3.2 Dominated Sets in Phase Two

Recall that in phase two we traverse the arcs in descending order. The names successor and predecessor refer to the order in which the arcs are traversed and not the sort order of the arcs. In phase two we traverse all dimensions for the second time, i.e. we traverse the recursion levels $r = d + 1, ..., 2d$. Since we always compare dimension values, we have to normalize the recursion value r by setting the variable l, which denotes the dimension, to $l = r \bmod d$.

In order to detect which sets have become dominated, we store in each recursion the successor s^r, i.e. the next start arc $a_{s_{i+1}}$ and the predecessor p^r, i.e. the smallest integer that can be reached form the previous start arc $a_{s_{i-1}}$. Assuming we are currently in interval I_x^r, with start arc a_{s_x}, we can define predecessor and successor as follows:

Definition 7. (Predecessor and Successor)

1. $s^r = a^l_{s_{x+1}}$

2. $p^r = (\tilde{v}^l + (a^l_{s_{x-1}} - \tilde{\delta}^l + 1)) \bmod \tilde{v}^l$

Thus, a non-dominated interval must contain at least one arc $a^l > s^r$ and at least one arc $a^l < p^r$.

Lemma 2. *If a set contains at least one arc $a^l > s^r$ and at least one arc $a^r < p^r$, it is not dominated with regard to recursion level r, where $r = d + 1, \ldots, 2d$.*

3.2.3.3 Non-dominated Sets Over All Recursion Levels

In the following proof of Lemma 1 and 2 we first show that an interval that does not overlap its direct neighbors does not overlap any other interval either. We then show that this condition is met if the set built from the arcs within this interval contains an arc or a combination of arcs not contained in any of the neighboring intervals.

Proof. All intervals differ in at least one arc and the arcs are sorted in ascending order of the respective dimension. In phase one, the successor s^r is defined as the smallest integer reachable by the subsequent interval in recursion level r. The predecessor p^r is defined as the greatest integer reachable by the preceding interval. So if an interval does not overlap its preceding and subsequent interval on recursion

level r, where $r = 1, \ldots, d$, it cannot overlap any other interval on this recursion level. Therefore a set that contains one arc such that $a^r > p^r$ and one arc such that $a^r < s^r$ cannot overlap any other set with regard to recursion level r.

The same is true for phase two. If an interval in phase two contains one arc such that $a^{r \bmod d} < p^r$ and one arc such that $a^{r \bmod d} > s^r$, it does not overlap its preceding and subsequent interval on recursion level r, where $r = d + 1, \ldots, 2d$, and it can therefore not overlap any other interval on this recursion level. Thus, if a set contains one arc such that $a^{r \bmod d} < p^r$ and one arc such that $a^{r \bmod d} > s^r$, it is not dominated with regard to recursion level r. □

In the following steps we extend the definitions of Lemma 1 and 2 to all recursion levels. Let $A(n^r)$ denote the set of arcs corresponding to a tree node n^r in recursion depth r.

Lemma 3. *The sets corresponding to the descendants of node n^r, where $r = 1, \ldots, d$, which contain one $a^r \in A(n^r)$ such that $a^r > p^r$ and one $a^r \in A(n^r)$ such that $a^r < s^r$, cannot be dominated by sets corresponding to siblings or descendants of siblings of n^r.*

For phase two the corresponding properties are defined by Lemma 4.

Lemma 4. *The sets corresponding to the descendants of node n^r, where $r = d + 1, \ldots, 2d$, which contain one $a^l \in A(n^r)$, where $l = r \bmod d$, such that $a^l < p^r$ and one $a^l \in A(n^r)$ such that $a^l > s^r$ cannot be dominated by sets corresponding to siblings or the descendants of siblings of n^r.*

Proof. Lemma 3 and 4 associate each set of a recursion level r with a tree node n^r and state that a set corresponding to a node n^r, which fulfills the condition of Lemma 1 and 2 and is therefore not dominated by siblings of n^r, is also not dominated by the descendants of siblings of n^r. This is easy to show. Since in each recursion step a set is divided further into subsets, it holds that $A(n^{t+1}) \subseteq A(n^r)$, $\forall t = r + 1, \ldots, 2d$. Thus, a set corresponding to a node n^r, which is not dominated by sets corresponding to siblings of this node, cannot be dominated by sets corresponding to descendants of the siblings. □

The next step is to extend this principles to all preceding recursion levels of a node n^r. Since the set corresponding to n^r can still be dominated by sets corresponding to siblings of its ancestors n^t, where $t = 0, \ldots, r - 1$.

Theorem 5. *If a set corresponding to node n^r fulfills the condition of both Lemma 3 and Lemma 4 regarding all previous recursion levels $t = 1, \ldots, r - 1$, it can not be dominated by any set* [1].

Proof. In Lemma 1 and Lemma 2 we showed that an arc set is not dominated on recursion level r if it contains at least one arc not representable by the preceding interval and one arc not representable by the subsequent interval. We then showed in Lemma 3 and Lemma 4 that an arc set $A(n^r)$ corresponding to a node n^r cannot be dominated by sets corresponding to siblings or descendants of siblings of n^r. In a third step we have to show that if $A(n^r)$ is not dominated on any of the preceding recursion levels, it is not dominated by sets corresponding to siblings of its ancestors $A(n^t)$, where $t = 1, \ldots, r - 1$ or by their descendants. If a set corresponding to node n^r contains at least one arc such that $a^t < s^t$, at least one arc such that $a^t > p^t$, $\forall t = 1, \ldots,$ min(r-1,d), at least one arc such that $a^{t \bmod d} > s^t$ and at least one arc such that $a^{t \bmod d} < p^t$, $\forall t = d + 1, ..., r - 1$, it cannot be dominated by any set. Note that whenever a new set is built, we have to check again if it is not dominated on any preceding recursion level. □

[1] We imply the trivial fact that a set corresponding to n^r is dominated by the sets corresponding to the ancestors of n^r.

3.2.3.4 Normalization of the Arc Comparisons

In the following, we list the cases where the comparison of the arc value with its predecessor or successor has to be adapted to the ring structure. Even though it should be clear implicitly where comparisons have to be normalized in order to account for the ring structure, we will list these special cases here.

3.2.3.4.1 Phase One In phase one, the last or the penultimate interval can overlap the first interval. In this case, we define the greatest arc $a^l_{|S^{r-1}|}$ as the last center arc (see Section 3.2.1) and the last interval as predecessor of the first interval. If intervals go across the domain border, the following two cases can occur.

Case 1: The predecessor p^l belongs to an interval that goes across the domain border (i.e. it belongs to the last or the penultimate interval). To meet the condition $a^l > p^l$, the compared arc a^l must also not lie in the last interval. Thus, it has to meet the additional condition $a^l < \tilde{v} - \tilde{\delta}^l + 1$. Case 2: Inversely, an arc also meets the condition $a^l > p^l$ if $(a^l < \tilde{\delta}^l - 1) \wedge (p^l \geq \tilde{\delta}^l - 1)$. This case can occur if the predecessor is in the penultimate interval, and a^l is in the last interval.

3.2.3.4.2 Phase Two In phase two we traverse the arcs in descending order. Thus, the first start arc $a^l_{s_1}$ is defined as the first arc not reachable from the smallest arc $a^l_{s_1} < a^l_1 - \tilde{\delta}^l + 1 + \tilde{v}^l$ (see Section 3.2.2.1). If intervals go across the domain border, we have to normalize the arcs in the following two cases. Case 1: The condition $a^l < p^l$ is not met if $(a^l \leq \tilde{\delta}^l - 1) \wedge (p^l \geq \tilde{v}^l - \tilde{\delta}^l + 1)$. Case 2: The condition $a^l > s^l$ is not met if $(s^l \leq \tilde{\delta}^l - 1) \wedge (a^l \geq \tilde{v}^l - \tilde{\delta}^l + 1)$.

3.2.3.5 Recursive Detection of Dominance

The following part describes how dominated sets are detected and avoided from the outset. If a set is not dominated, its intervals have to be non-dominated in all preceding recursion levels. To improve the efficiency of the dominance checks we use two flags, *ndp* and *nds*, which indicate whether or not the current interval is dominated by its predecessor or successor in any of the preceding dimensions (*ndp*/*s* means not dominated by predecessor/successor).

During the collection of the arcs within an interval, a comparison of each arc with its predecessors and successors in the former dimensions is performed by procedure `arcSubsumption` (Algorithm 1), which receives the current arc and the recursion depth − 1 as input parameters. Table 3 shows the symbols used in the following algorithms.

Procedure `arcSubsumption` (Algorithm 1) sets the two flags *ndp* and *nds*.

Algorithm 1: `arcSubsumption(`a`,`x`)`

1 $$\text{ndp[r]} = \begin{cases} a^{r \bmod d} > p^r & \forall r = 0, ..., x \\ a^{r \bmod d} < p^r & \forall r = d+1, ..., x \end{cases}$$

2 $$\text{nds[r]} = \begin{cases} a^{r \bmod d} < s^r & \forall r = 0, ..., x \\ a^{r \bmod d} > s^r & \forall r = d+1, ..., x \end{cases}$$

Table 3: Symbols used in `buildRange` (Algorithm 4), `buildTAs` (Algorithm 5), `noTAExists` (Algorithm 3), `arcSubsumption` (Algorithm 1) and `checkDomination` (Algorithm 2.)

Symbol	Purpose
T^c	Set of template arcs
L	Ordered set for arc lookup where the arc position corresponds to the arc identity on each position $L[a]$ the template arcs that correspond to the arc a are stored
$A(t)$	Arcs a specific template arc t can represent
p	Vector containing one predecessor for each recursion level
s	Vector containing one successor for each recursion level
ndp	Vector for predecessor lookup containing a domination flag for each recursion level
nds	Vector for successor lookup containing a domination flag for each recursion level
B	newly constructed ordered set of template arcs
S	ordered set of template arcs

In phase one $ndp[r]$ is set to *true* if $a^r > p^r$ and $nds[r]$ is set to *true* if $a^r < s^r$. In phase two $ndp[r]$ is set to *true* if $a^{r \bmod d} < p$ and $nds[r]$ is set to *true* if $a^{r \bmod d} > s^r$.

The direction is reversed for phase two because the traversal is performed in descending order and the subsequent arc is therefore smaller than the current arc. Note that only the predecessor from the directly preceding interval and the successor from the immediately following interval have to be stored, for only neighboring intervals can overlap.

At any point during the entire procedure no more than $4d - 2$ predecessor-successor pairs have to be stored, i.e two pairs for each dimension for each of the two phases, except for the last dimension in phase two. An exception to this rule is the first and the last interval, provided there are arcs representable across the domain border. Because of the ring structure, the last interval is the predecessor of the first interval and the first interval is the successor of the last interval. As mentioned before, p^r in the first interval is known before all arcs have been traversed, because the last arc is always designated as the last center arc.

The procedure `checkDomination` always starts on recursion level one and makes use of the previously collected predecessor-successor pairs in order to speed up the operation. If the predecessor-successor pair $ndp[j]$ and $nds[j]$ are both *true* the interval is not dominated on recursion level j (line 2) and *build* is set to *true*.

A special case arises when two intervals are exactly the same. Since we want to build the set only once, each interval I_n^1 must be processed on its last possible occurrence, because otherwise it may be dominated by a set built from a subsequent interval I_{n+1}^1. If $nds[j] = false$ the subsequent interval overlaps the current interval completely, i.e. it is not the last occurrence of the interval and $false$ is returned (line 7-9).

We know that a particular occurrence of an interval is the last one if it does not overlap its successor, since in this case it represents arcs that are no longer represented by the succeeding interval. Obviously, the first occurrence of intervals that go across the domain border ($across[j] = true$) is an exception, since their predecessor is built later. So if $ndp[j] = false$, $nds[j] = true$ and $across[j] = true$ (line 3) it is the last occurrence of the interval in recursion level j, and we set *build* to *true*. We also set *checkTA* to *true* (line 5), to indicate that we have to check later if the set already exists, provided *build* is still *true*. Note that the interval could overlap with its successor on a later recursion level, in

which case False is returned.

Whenever a decision is made about whether or not to build a new set, we check if it is dominated on any of the preceding recursion levels. Procedure `checkDomination` (Algorithm 2) checks each recursion level preceding the current one to see if it is dominated (line 1 - 10) and returns *true* if the interval is not dominated on any recursion level.

If it is the last occurrence of an interval with regard to all preceding recursions and $checkTA = true$ we have to check if a template arc exists that covers all arcs from this interval. Note that we have to check if a template arc already exists only in special cases. Because it has to be the last occurrence of an interval regarding all recursion levels and this interval must not be dominating.

Whenever a candidate template arc $t \subset T^c$ is constructed the arcs it represents, $A(t) \subset A$, are associated with it. Conversely, each arc contains a set of pointers to candidate template arcs, $T(a) \subset T^c$, by which it can be represented. The arcs represented by the newly constructed template arc are stored in a global array L. The arc position in this array corresponds to the arc number. Thus, arc a_i has position i. L is initially empty.

Algorithm 2: `checkDomination(`S `,` r`)`

```
1  for j = 1,...,r do  //check all previous recursion levels
2      build ← ndp[j] ∧ nds[j]
3      if build = false ∧ nds[j] = true ∧ across[j] = false then
4          build ← true      //last occurrence of interval
5          checkTA ← true
6      end
7      if build = false then
8          return false
9      end
10 end
11 if checkTA then
12     build ← noTaExists(S)      //returns false if S ⊆ A(t)
13 end
14 return build
```

The procedure `noTaExists` (Algorithm 3) checks whether a template arc that covers all arcs within an interval already exists. It first performs a lookup to see if each arc within the interval I is represented by at least one template arc (line 1-5). The complexity of this lookup is $O(|A(I)|)$.

Note that it is only checked whether all arcs within the current set S are represented at all, and therefore the position that corresponds to the arc index in the global array $L[a]$ is not empty. These arcs may be represented by different template arcs.

Thus if all arcs are represented, we check if they are represented by a single template arc (line 6-12). We take an arbitrary arc from our interval and look it up in the global arc vector. Then we iterate over the set of template arcs associated with it, which can be done in $O(|T(a)|)$. We check, for each template arc, if it can represent all arcs within the interval (line 8). This is done in $O(|A(I)|)$. If a template arc can be found that represents all arcs contained in the interval $A(i)$ `noTAExists` returns *false* and *build* is set to *true*, because the set must not be built. If none of the template arcs represents all of them, build is set to *true* (line 12).

Algorithm 3: `noTAExists(`S`)`

```
1  foreach a ∈ S do //are all arcs represented
2      if L[a] = 0 then
3          return true
4      end
5  end
6  let a be an arbitrary arc from S
7  foreach t ∈ L[a] do //check if S represented by a template arc associated with a
8      if S ⊆ A(t) then
9          return false
10     end
11 end
12 return true
```

Now we can describe the whole *NIS* procedure. Procedure `buildRange` (Algorithm 4) gets the ordered set of all arcs $S_1^0 = A$ as input. The arcs are ordered by dimension 1. On each recursion level r new ordered arc sets B are built, so we construct an empty set in line 1. The position of the first center arc a_c^r (line 1) is either the first arc that can represent all smaller arcs (see Section 3.2.1.1 case 1) or, if arcs are representable across the domain border, the first arc not reachable from the last center arc (see Section 3.2.1.1 case 2). Variable m stores the value of the predecessor from the first interval, which we assign to the predecessor variable $p[r]$ in line 4. Note that the value of m can be negative, in which case the condition that the interval has to contain one variable greater than the predecessor is always true. The value of the predecessor of the first interval is either the value of the last arc reached by the last interval if arcs are representable across the domain border or negative (line 2). Starting with the first center arc we iterate over all arcs (line 3) and collect the arcs within the right (line 5-11) and the left (line 12-18) range of that center arc. The arcs that are representable across the domain border are collected on lines 19-28. The collected arcs are sorted according to the next dimension. Once the requested dimension has been reached, the arcs are once again sorted by dimension 1 in preparation for the invocation of `buildTA`, which starts phase two. While collecting the arcs the procedure `arcSubsumption` (Algorithm 1) also collects the predecessor-successor pairs of the preceding recursion levels u = r − 1...1 (line 8,15,23). If the interval is not dominated on any of the previous recursion levels, which is checked by `checkDomination` (Algorithm 2), the new set is built (line 31-37). The procedure `buildRange` is called recursively (line 32-34) until the requested dimension d is reached. When d is reached `buildTA` (Algorithm 5) is called (line 35). At this point `buildRange` has constructed a group of arcs where each arc is representable together with at least one other arc regarding each dimension.

The value of the predecessor from the next interval is the last arc reached by the current center arc (line 29). The successor of this interval is the smallest arc reached by the next center arc (line 30). The procedure `buildTA` builds the sets that can be represented by one candidate template arc. It starts again in dimension 1, but the arcs are now traversed in descending order. The position of the first start arc is either the last arc (see Section 3.2.2.1 case 1) or, if arcs are representable across the domain border, the first arc not representable together with the smallest arc (see Section 3.2.2.1 case 2).

Algorithm 4: `buildRange(`S `,` r`)`

```
1  B ← ∅
2  pos ← position of first center arc
3  m ← S[|S|]^r + δ̃^r − 1 − ṽ^r   //last arc reached by previous center arc [a]
4  while pos ≤ |S| do //the arcs in S are sorted by r
5      a_c^r ← S[pos]^r; i ← pos; j ← 1; p[r] ← m
6      while S[pos]^r ≤ (a_c^r + δ̃^r − 1) do
7          B ← B ∪ S[pos]
8          if r > 1 then
9              arcSubsumption(S[pos], r − 1)   //set global variables nds and ndp
10         end
11         pos ← pos + 1
12     end
13     while S[i]^r ≥ (a_c^r − δ̃^r + 1) ∧ (i ≥ 0) do
14         B ← B ∪ S[i]
15         if r > 1 then
16             arcSubsumption(S[i], r − 1)
17         end
18         i ← i − 1
19     end
20     if (a_c^r + δ̃^r − 1) ≥ ṽ^r then
21         while S[j]^r ≤ (a_c^r + δ̃^r − 1) mod ṽ^r do
22             B ← B ∪ S[j]
23             if r > 1 then
24                 arcSubsumption(S[j], r − 1)
25             end
26             j ← j + 1
27         end
28         pos ← |S|    //get position of next center arc across the domain border
29     end
30     m ← (a_c^r + δ̃^r − 1) mod ṽ^r     //value of predecessor from next interval
31     s[r] ← S[pos]^r − δ̃^r + 1     //successor from this interval
32     if r < 2 ∨ checkDomination(B, r − 1) then
33         sort B in next dimension (r mod d) + 1
34         if r < d then
35             buildRange(B, r + 1)
36         else
37             buildTA(B, r + 1)
38         end
39     end
40 end
```

[a] Because of the ring structure the last interval overlaps with the first interval.

Algorithm 5: `buildTA(`S`,`r`)`

```
1  B ← ∅ ; pos ← position of first start arc
2  h ← (r − 1) mod d + 1      //dimension in which arcs are sorted
3  m ← S[1]^h − δ̃^h + 1 + ṽ^h      //first interval overlaps last interval
4  while pos > 0 do      //traverse in descending order
5      p[r] ← m
6      bound ← (S[pos]^h − δ̃^h + 1)
7      while S[pos]^h ≥ bound do      //arcs representable together
8          B ← B ∪ S[pos]
9          arcSubsumption(S[pos], r − 1)      //set global variables nds and ndp
10         pos ← pos − 1
11     end
12     if bound ≤ 0 then      //arcs representable together across the domain border
13         pos ← |S|
14         while S[pos]^h ≥ (ṽ^h + bound) do
15             B ← B ∪ S[pos]
16             arcSubsumption(S[pos], r − 1)
17             pos ← pos − 1
18         end
19         a_s ← max(a ∈ S : (a^h − δ̃^h + 1) ≤ S[pos]^h ∧ (a^h − δ̃^h + 1) ≤ δ̃^h)      //next start arc
20     else
21         a_s ← max(a ∈ S : (a^h − δ̃^h + 1) ≤ S[pos]^h)      //next start arc
22     end
23     if checkDomination(B, r − 1) then
24         if h < d then
25             sort B in next dimension h + 1
26             buildTA(B,r + 1)
27         else
28             t ← τ(B)      //set candidate template arc
29             T^c ← T^c ∪ t
30             L ← L ∪ B      //insert represented arcs in global arc lookup
31         end
32     end
33     pos ← position of next start arc a_s
34     s[r] ← S[pos]^r      //successor of this interval is the next start arc
35     m ← bound      //value of predecessor from next interval
36 end
```

The procedure `buildTA` iterates over all arcs until the smallest arc is reached (line 4-35). All arcs within the range of the respective start arc are collected (line 4-11). The arcs that are representable across the domain border are collected in line 12-19. The collected arcs are sorted according to

the next dimension (line 8,15). While collecting the arcs `arcSubsumption` (Algorithm 1) is called (line 9,16). If the interval is not dominated in any of the previous recursions, which is verified by `checkDomination`, the new set is built (line 22-29).

The procedure `buildTA` is called recursively (line 23-25) until the requested recursion depth $2d$ is reached. If $2d$ is reached we are finished and the new candidate template arc, which can represent the newly built arc set, is constructed by selecting the smallest dimension value a^l from $E \subset B$ in each dimension (line 27), where E comprises only arcs in B that are reachable without crossing the domain border. The candidate template arc is stored in the global arc set T^c (line 28). All arcs within the interval are stored in the global arc lookup table L, where each position corresponds to the arc identity. Furthermore, each arc is associated with the template arcs $T(a)$ that can represent it (line 29). The next start arc is the greatest arc that can represent the first arc ($S[pos]^h$) outside the current range (line 16, 19).

3.2.3.6 Space Complexity of the Entire Procedure

During the entire procedure, only T^c, the global array of arcs and the directly preceding recursion steps $2d \cdot |A|$ are stored. Let $t^{\max}$ denote the template arc that represents the greatest number of arcs, and let $a^{\max}$ denote the arc that is represented by the greatest number of candidate template arcs. Therefore the space complexity is $\mathrm{O}(|T|^c \cdot |A(t^{\max})| + |A| \cdot |T^c|(a^{\max}) + 2d \cdot |A|)$. Note that we have to represent both arrays, since a smaller $t^{\max}$ leads to a greater $a^{\max}$ and in the extreme case one template arc represents all arcs.

3.2.3.7 Example: Demonstration of *NIS*

Figure 8 illustrates the functionality of *NIS* on the basis of the previous example (see Section 3.2.2.2). Note that the impact of pruning the tree is much greater in a realistic application when we typically deal with thousands of arcs and three dimensions. Note that even though a tree is used in this illustration, the algorithm avoids storing tree nodes in order to be more memory efficient. We describe the first branch of the tree shown in 8 in the order of its construction. At the beginning the arcs are sorted in ascending order by dimension 1.

Recursion depth 1, interval number 1: The tree traversal starts in dimension 1. Arc 2 is chosen as first center arc $a^1_{c_1} = 30$ (2), since this is the first arc not reachable by the last center arc. As stated before (see Section 3.2.1.1 case 2) we designate the final arc as last center arc. Therefore we know that the last interval is $I^1_{last} : [82, 28]$. The arcs representable together with $a^1_{c_1}$ are within $I^1_1 : [1, 59]$ because $(a^1_{c_1} + \tilde{\delta}^1 - 1) \bmod \tilde{v} = 59$ and $(a^1_{c_1} - \tilde{\delta}^1 + 1) \bmod \tilde{v} = 1$. The last interval overlaps the first interval in $[0, 28]$, so the predecessor $p^1 = 28$. The next center arc is $a^1_{c_2} = 70$ (6) since this is the the smallest arc outside the current interval. The smallest integer representable by the next center arc $a^1_{c_2}$ is 41, and hence the successor of the current interval is $s^1 = 41$.

The predecessor p^1 and successor s^1 become relevant when the set is divided in later recursions. The children of the set built from the first interval must always contain at least one arc greater than $p^1 = 28$ and one arc smaller than $s^1 = 41$ regarding dimension 1 (see Section 3.2.3.3).

Recursion depth 2, interval number 1: As the tree is traversed depth first, the next step is to divided the set along dimension 2, and the first center arc $a^2_{c_1} = 20$ (4) in dimension 2 is arc 4. Although the interval $I^2_1 : [103, 49]$ goes across the domain border, no arcs are representable across the domain border, since the greatest arc regarding dimension 2 is 85. Thus, the first interval has no preceding

interval and the predecessor p^2 is therefore negative . In the Figure we set it to the dummy value -1. Since the next center arc $a^2_{c_2} = 53$ (2), the successor is set to $s^2 = 24$.

While the arcs $\{3, 4, 5\}$ contained in this interval are collected, we check whether one of them has a dimension 1 value greater than the predecessor $p^1 = 28$ and wether one of them has a dimension 1 value smaller than the successor $s^1 = 41$. Since arc 0 is 50 in dimension 1, it is greater than $p^1 = 28$. Thus the interval does not overlap its preceding interval regarding dimension 1. The outcome of the successor check is that none of the arcs is smaller than $s^1 = 41$. Thus the interval overlaps the subsequent interval regarding dimension 1, so we must not build the set and the whole branch of the tree is pruned.

Recursion depth 2, interval number 2: The next center arc is $a^2_{c_2} = 53$ (2), the interval is $I^2_1 : [24, 82]$ and the arc set is $\{2, 3, 0\}$. The interval does not overlap regarding dimension 1, since arc 2 is 30 in dimension 1 and therefore greater than the predecessor $p^1 = 28$ and also smaller than the successor $s^1 = 41$. Thus the new set is built. The predecessor in dimension 2 is $p^2 = 49$ and the successor in dimension 2 is $s^2 = 56$, because the first arc not reachable by this interval, and hence the next center arc, is $a^2_{c_3} = 85$ (1).

Recursion depth 3, interval number 1: At this point the algorithm enters phase two. In phase two, we collect all arcs that are representable together. We start again in dimension 1 (note that recursion depth 3 equals dimension 1) and build, in each dimension, intervals of size $\tilde{\delta}^r - 1$. Note that we now proceed in descending order, i.e. we start with the greatest arc. Consequently, predecessor and successor are also reversed in phase two, so each interval must contain an arc smaller than its predecessor and one arc greater than its successor. The greatest arc, and therefore the start arc, is $a^3_{s_1} = 46$ (3), the interval is $I^3_1 : [46, 7]$ and the arc set is $\{3, 2\}$. Now, two preceding dimensions have to be checked for dominance. Since the coordinate value of arc 2 is 30 in dimension 1, it is greater than $p^1 = 28$ and also smaller than $s^1 = 41$. The coordinate value of arc 2 is 53 in dimension 2. Thus, it is greater than $p^2 = 49$ and it is also smaller than $s^2 = 56$. Because the dominance check is negative for both dimensions a new set is built. Since no arcs are representable across the domain border, the predecessor of this interval is set to $p^3 = \infty$. Note that in contrast to phase one the next start arc is the arc that can reach an arc not reachable by the previous start arc, because we want to build all groups of arcs that are representable together now. The next start arc is $a^3_{s_1} = 30$ and therefore $s^3 = 30$.

Recursion depth 4, interval number 1: The first start arc on recursion level 4 is $a^4_{s_1} = 53$ (2), the interval is $I^4_1 : [53, 24]$, and the arc set is $\{3, 2\}$. Since this set is exactly the same as in dimension 3 the dominance checks are redundant. For the sake of clarity we describe them anyway: We have to check three preceding dimensions for dominance. The coordinate value of arc 2 is 30 in dimension 1, thus it is greater than $p^1 = 28$ and also smaller than $s^1 = 41$. The coordinate value of arc 2 is 53 in dimension 2, so it is greater than $p^2 = 49$ and also smaller than $s^2 = 56$. In phase two predecessor and successor are reversed. The interval is automatically smaller than the predecessor because $p^3 = \infty$ and arc 3 is 46, so it is greater than its successor $s^3 = 30$. So a new template arc is built, which has the coordinates $(30, 40)$, i.e. the smallest dimension value a^l from the arc set $\{3, 2\}$ in each dimension l.

Recursion depth 3, interval number 2: The second start arc on recursion level 3 is the first arc that can reach an arc outside the current interval. This is $a^3_{s_2} = 30$ (2), the interval is $I^3_2 : [30, 1]$ and the arc set is $\{2, 0\}$. Since the coordinate of value of arc 2 is 30 in dimension 1, it is greater than $p^1 = 28$ and also smaller than $s^1 = 41$. The coordinate value of arc 2 is 53 in dimension 2, so it is greater than $p^2 = 49$ and also smaller than $s^2 = 56$. Since the dominance check is negative for both dimensions a new set is built.

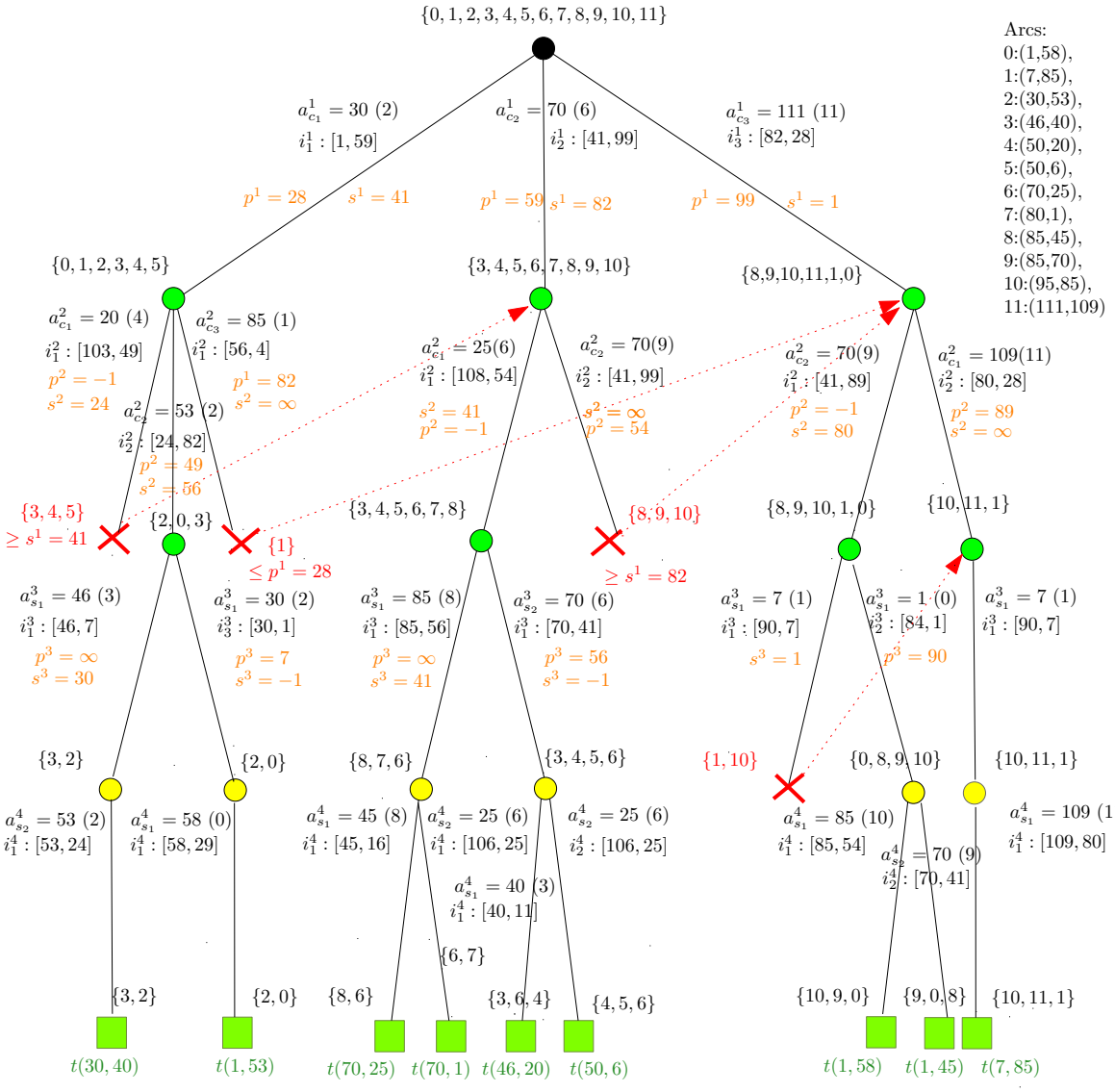

Figure 8: The upper right of the figure shows the set of arcs with their respective numbers and coordinates. In the tree diagram only the arc numbers are quoted. The caption of the ingoing arc of each node describes the specific data of this node, i.e. the center arc $a^l_{c_x}$ or in phase two the start arc $a^l_{s_x}$, the representable interval $I^l_x : [..., ...]$, the predecessor p^r and successor s^r of the interval (in orange), and in curly braces the set of arcs that the node can represent. The superscript always denotes the recursion level r, and the subscript x always denotes the number of the respective interval. Let us look at the leftmost child of the root node, for instance: $a^1_{c_1} = 30(2)$ means the center arc is the first center arc and it is in recursion level 1. The value of this center arc is 30 and it has arc number 2. The first interval in dimension 1 goes from 11 to 69 ($I^1_1 : [11, 69]$). The predecessor of this interval is 11 ($p^1 = 11$), the successor of this interval is 31 ($s^1 = 31$) and $\{0, 1, 2, 3, 4, 5\}$ is the represented arc set. The dominated sets (which are not built) are marked red, where $\geq s^r$ means the set contains only arcs greater than the successor and is therefore dominated by the subsequent interval and $\leq p^r$ means that the set contains only arcs smaller than the predecessor and the interval is therefore dominated by the preceding interval. E.g., all elements from the set $\{3, 4, 5\}$ are $\geq 41 = s^1$ regarding dimension 1 and therefore dominated by the subsequent arc set $\{3, 4, 5, 6, 7, 8, 9, 10\}$ in interval $I^1_2 : [41, 99]$. The pruned branches are marked with a red cross, and the red dotted arrow points from the dominated set to its dominating set, e.g from $\{3, 4, 5\}$ to $\{3, 4, 5, 6, 7, 8, 9, 10\}$.

Recursion depth 4, interval number 1: The first start arc on recursion level 4 is $a^4_{s_1} = 58$ (0), the interval is $I^4_1 : [58, 29]$, and the arc set is $\{2, 0\}$. Since this set is exactly the same as in dimension 3 the dominance checks are redundant. A new template arc is built, which has the coordinates $(1, 53)$.

Recursion depth 2, interval number 3: The next center arc in dimension 2 is $a^2_{c_3} = 85$ (1), the interval is $I^2_3 : [56, 4]$, and the arc set is $\{1\}$. This interval overlaps the last interval (which is its preceding interval) regarding dimension 1, because arc 1 is 7 in dimension 1, so it is smaller than $p^1 = 28$ and the set is not built. We can see that arc 1 is contained in the arc set $\{8, 9, 10, 11, 1, 0\}$ of the rightmost node. In this case arc 1 is represented across the domain border.

3.3 Dynamic Non-Dominated Interval Search Tree (DNIST)

The algorithm *DNIST* is a modified variant of *NIS*. It is used for solving the pricing problem in a *BCP* process described in Chapter 5.3.3. In the pricing approach, values are assigned to the arcs. More precisely, in each pricing step, each arc is assigned the value of its corresponding dual variable. We denote these values as $\overline{c}_{i,j}$. The value of a candidate template arc is the sum of the arc values it can represent. The pricing problem is to find the most valuable candidate template arc in terms of its arc values. The pricing-problem needs to be solved frequently within the overall *BCP* process, where the arc values change in each request while the arcs stay the same.

The procedure *DNIST* returns the candidate template arc with the greatest value whenever a pricing request is made. Instead of storing only the sets generated in the directly preceding recursion steps, as in *NIS*, we now store all generated sets in a tree structure. Every node n^r_x of the tree represents the corresponding set of arcs S^r_x in the respective recursion depth r.

The value of each node is the sum of the arc values it represents. This sum is a local upper bound $ub(n^r_x)$ of this node, since it can only become smaller or stay the same in later recursion steps. Whenever a candidate template arc is requested the previously stored parts of the tree are searched. If necessary new branches are created and stored for further requests. The tree is built using preorder traversal as described before.

The best template arc found in a request obviously constitutes a global lower bound lb^* for this pricing step. Whenever the local upper bound of a node is smaller than or equal to the global lower bound $ub(n^r_x) \leq lb^*$ the algorithm does not need to traverse or build the children of the current node as it is impossible to find a better candidate template arc down that path. Recall that in each request a new dual value is assigned to each arc. Hence, different branches of the tree may be built and traversed, depending on the arc values.

The search starts with the root node. If the root node has children they are traversed depth first in order of their creation, as long as their upper bound is greater than lb^*. If no children exists the tree is built further as described before for algorithm *NIS*.

The one difference to *NIS* is that the nodes are stored now and branches are only built from nodes n^r_x where $ub(n^r_x > lb^*)$. Thus the reason that a branch is not built further is either that it is dominated or that th value of $ub(n^r_x$ is to low. Dominated arc sets in the final recursion step need to be treated specially, because although we do not build these arc sets, we know at this point that a candidate template arc with at least this value must exist. Whenever an arc set in the final recursion step is dominated and the sum of its arc values is greater than lb^*, we store this sum minus a tiny ϵ value as the new lb^* . We have to subtract this tiny value as we also want to find dominating template arcs that are equal to lb^*.

The algorithm performs even better if we store the number of arcs a node can represent ($|A(n)|$). If this number times the greatest assigned dual value ($c_{\max}$) is smaller than or equal to the lower bound ($|A(n)| \cdot c_{\max} \leq lb^*$) we do not have to add the arc values within that node, since their sum will never be greater than lb^*.

Note that whenever a branch of the tree is not built further because the local upper bound of the node is too small, we have to store the predecessor p^l and successor s^l vectors as well as the lookup tables ndp and nds within this node, because we need to know its dominance information in case we want to build this branch further in a later call. Note that an alternative strategy would be to traverse the branches *best-first* along the best local upper bound $ub(n_x^r)$. This strategy was used for a *segmentation tree* developed for the same pricing purpose in [14, 46]. However, we did not pursue this approach, since it did not seem to be promising regarding the structure of our tree as its depth is only $2 \cdot d$ and the branches do not contain dominated arcs.

The sets which are built in phase one are relatively big. Therefore, a large upper bound in this phase does not say much about the value of the template arc. Whereas the recursion depth in phase two is only d. Furthermore since dominated branches are never built, we do not need to avoid them with the strategy only to visit branches with greater lower bounds. The idea behind the later is, that if a node n_x^r is dominated by node n_y^r it has always a smaller upper bound than its dominating node $ub(n_x^r) \leq ub(n_Y^r)$.

3.4 Results

All tests were run on a Dual Core Opteron 270 CPU (2.0 GHz.) with 4GB RAM under Linux kernel 2.6.62. The algorithms are implemented in C++ using the Standard Template Library (STL) and Boost (http://www.boost.org/). We used g++-4.1 to compile the code. For set and graph data structures as well as graph algorithms we used LEDA (Library of Efficient Data Types and Algorithms) version 5 [33]. Our algorithms are embedded in a framework developed by Dietzel, see [22] for a detailed description.

3.4.1 NIS Results

Table 4 shows the running time of *NIS* for all *NIST* instances. Table 5 shows the running time of *NIS* for large $\vec{\delta}$ values. For the *NIST* instances that could be solved. Missing values are indicated by a '-' character. The procedure *NIS* takes only a few seconds to generate of up to 100000 candidate template arcs. It generates half a million candidate template arcs in just a few minutes and can be used to generate up to 1.5 million candidate template arcs.

3.4.2 DNIST Results

Table 6 shows the running time of *DNIST* for 1000 iterations, where the arcs are initialized with random values, to simulate the pricing problem. We see that *DNIS* can generate template arcs on demand very fast, especially in relation to the *BCP* method presented in Chapter 5. The developed *BCP* method can successfully solve instances from the *NIST* dataset containing up to 9000 template arcs and 90 nodes with a correction vector size of $\vec{\delta} = (50, 50, 50)^T$ in a reasonable amount of time. For instances of this size the runtime of $DNIST$ is less than a second.

Table 4: Number of candidate template arcs $|T^c|$ and running times of *NIS* for the NIST data.

$\vec{\delta}$	$(40,40,40)^T$		$(80,80,80)^T$		$(120,120,120)^T$		$(130,130,130)^T$	
inst.	$\|T^c\|$	t[s]	$\|T^c\|$	t[s]	$\|T^c\|$	t[s]	$\|T^c\|$	t[s]
nist-u-01-t	7339	0.19	25032	1.05	106756	10.42	156137	21.37
nist-u-02-t	6924	0.17	25560	1.23	108946	12.85	155000	28.73
nist-u-03-t	9260	0.28	44834	2.55	239528	49.66	366274	132.73
nist-u-04-t	5238	0.11	14490	0.57	54618	4.47	79124	9.38
nist-u-05-t	4002	0.10	10565	0.43	38467	3.08	55091	6.25
nist-b-01-t	9559	0.25	39870	1.99	191019	27.84	283884	69.77
nist-b-02-t	8278	0.25	38826	2.20	194689	34.85	289420	89.77
nist-b-03-t	10142	0.28	42600	2.33	206608	32.72	309440	79.33
nist-b-04-t	4986	0.12	13611	0.55	51060	4.00	69336	7.67
nist-b-05-t	7150	0.18	27287	1.36	120638	15.06	168582	31.00
nist-g-01-t	7996	0.20	26776	1.28	112871	13.50	163900	30.87
nist-g-02-t	7820	0.18	25548	1.16	109406	10.48	150206	22.06
nist-g-03-t	9618	0.26	42035	2.29	198267	34.47	288411	85.17
nist-g-04-t	12657	0.36	52759	2.81	248842	37.51	355886	88.02
nist-g-05-t	4764	0.10	13130	0.52	50303	3.87	70240	8.15
$\vec{\delta}$	$(140,140,140)^T$		$(150,150,150)^T$		$(160,160,160)^T$		$(170,170,170)^T$	
inst.	$\|T^c\|$	t[s]	$\|T^c\|$	t[s]	$\|T^c\|$	t[s]	$\|T^c\|$	t[s]
nist-u-01-t	220914	45.67	315322	106.72	451665	272.99	636632	814.78
nist-u-02-t	220970	64.39	312336	158.01	457700	431.09	647866	1048.10
nist-u-03-t	542574	390.49	820830	1227.64	1169763	4682.64	1657232	10504.00
nist-u-04-t	111073	19.48	155297	40.33	224364	97.27	319632	245.46
nist-u-05-t	77374	12.59	107045	26.17	157276	61.95	212538	111.32
nist-b-01-t	402073	178.27	589198	477.87	869852	1543.89	1226542	4418.00
nist-b-02-t	410884	236.00	591532	609.38	874004	2329.24	1244119	6718.76
nist-b-03-t	465228	222.30	668902	639.24	958216	2170.07	1376313	7032.33
nist-b-04-t	99113	15.83	135304	31.81	189207	71.31	270754	160.87
nist-b-05-t	245192	74.03	350845	186.98	487907	489.50	682166	1266.62
nist-g-01-t	232484	70.19	343638	177.74	507706	506.84	742149	1442.22
nist-g-02-t	217917	49.07	306137	112.78	433021	275.88	594481	560.14
nist-g-03-t	428795	215.40	627226	597.52	915942	2161.18	1370066	7327.17
nist-g-04-t	514112	227.85	779460	767.36	1187806	2845.4	-	-
nist-g-05-t	96446	15.6	133290	33.91	189306	77.82	263606	166.65

3.4.3 Comparison with Other Preprocessing Methods and Earlier Approaches

In an earlier version of *NIS* we did not avoid building dominated branches of the tree through predecessor and successor comparisons. We only checked whether a template arc already exists by calling `noTaExists` (Algorithm 3) in the last recursion step. Obviously, this leads to a much greater number of sets being created as dominance is only detected in the last recursion step. The following Table 8 shows the speedup of the candidate template arc construction achieved by the new approach. We call the earlier version of *NIS EVNIS*. Table 7 shows a comparison of the previous preprocessing algorithm *PP* and *NIS*.

Finally, we can say that the two phase interval construction with *NIS* is a very efficient strategy and that the further improvement of pruning dominated branches reduces the space and time complexity of the algorithm significantly.

Table 5: Number of candidate template arcs $|T^c|$ and running times of *NIS* for large $\vec{\delta}$ values.

$\vec{\delta}$	$(180,180,180)^T$		$(190,190,190)^T$		$(200,200,200)^T$	
inst.	$\|T^c\|$	t[s]	$\|T^c\|$	t[s]	$\|T^c\|$	t[s]
nist-u-01-t	894186	1394.10	1176418	3153.79	1551236	7660.54
nist-u-02-t	915069	2130.18	1226780	4641.15	1532387	9972.58
nist-u-04-t	445778	447.92	583266	915.45	730458	1683.56
nist-u-05-t	279507	214.49	364574	386.49	458466	738.74
nist-b-01-t	1696518	8295.49	-	-	-	-
nist-b-04-t	358524	272.83	451805	487.06	583386	919.53
nist-b-05-t	970713	2531.70	1280962	5278.04	1657834	11602.10
nist-g-01-t	1037960	2811.48	1320905	5778.04	1614272	10174.10
nist-g-02-t	810362	1112.06	1059842	2767.13	1371250	5123.53
nist-g-05-t	361926	285.00	461576	561.35	580814	915.33

Table 6: Running times for 1000 iterations of DNIST for the *NIST* data where the arcs are initialized with random values between 1 and 100.

$\vec{\delta}$	$(40,40)^T$	$(40,40,40)^T$	$(80,80)^T$	$(80,80,80)^T$	$(120,120)^T$	$(120,120,120)^T$
nist-u-01	3.94	4.17	18.27	14.35	81.03	114.09
nist-u-02	3.74	3.48	16.94	12.36	68.92	85.08
nist-u-03	5.77	5.33	37.11	26.08	192.55	277.49
nist-u-04	2.57	2.44	9.54	6.30	34.24	40.68
nist-u-05	1.78	1.95	6.86	6.37	22.39	33.07
nist-b-01	5.36	4.84	31.11	21.69	143.80	183.80
nist-b-02	5.59	4.45	38.80	20.77	166.97	221.68
nist-b-03	5.63	5.41	29.51	17.68	125.41	159.84
nist-b-04	1.99	2.39	8.44	7.11	35.30	38.76
nist-b-05	3.97	3.94	21.31	13.68	105.87	104.42
nist-g-01	3.74	3.84	16.15	11.59	53.59	75.97
nist-g-02	3.55	4.05	16.29	13.17	63.73	87.97
nist-g-03	5.37	5.20	32.56	23.43	133.19	200.13
nist-g-04	7.53	6.67	45.66	28.46	203.00	277.67
nist-g-05	2.08	2.39	7.91	7.83	30.68	40.05

Table 7: Comparison of *PP* and *NIS*.

inst.	$\vec{\delta}$	$\|T^c\|$	PP t[s]	NIS t[s]
ft-01	$(30,30,30)^T$	1693	51.70	0.07
ft-18	$(30,30,30)^T$	3963	325.61	0.17
ft-03	$(40,40,40)^T$	6464	1153.62	0.62
ft-11	$(45,45,45)^T$	17137	13338.20	2.18
nist-u-04-t	$(40,40,40)^T$	5238	2395.28	0.11
nist-u-01-t	$(80,80,80)^T$	25032	11583.10	1.05
nist-u-01-t	$(120,120,120)^T$	106756	-	10.24
nist-u-05-t	$(120,120,120)^T$	38467	37112.10	3.08
nist-b-01-t	$(120,120,120)^T$	191019	-	27.84
nist-b-03-t	$(120,120,120)^T$	206608	-	32.72
nist-u-03-t	$(120,120,120)^T$	239528	-	49.66

Table 8: Number of candidate template arcs $|T^c|$ and running times of the early (EVNIS) and the improved version of NIS.

$\vec{\delta}$	$(40,40)^T$			$(40,40,40)^T$			$(80,80)^T$		
inst.	$\|T^c\|$	EVNIS t[s]	NIS t[s]	$\|T^c\|$	EVNIS t[s]	NIS t[s]	$\|T^c\|$	EVNIS t[s]	NIS t[s]
nist-u-01-t	10310	0.40	0.19	7339	0.72	0.19	25990	3.34	0.88
nist-u-02-t	9667	0.42	0.20	6924	0.74	0.17	26398	4.30	0.93
nist-u-03-t	13656	0.78	0.29	9260	1.29	0.28	37326	10.60	1.66
nist-u-04-t	6405	0.24	0.12	5238	0.47	0.11	16098	1.48	0.48
nist-u-05-t	4692	0.20	0.08	4002	0.37	0.10	11075	1.05	0.31
nist-b-01-t	13674	0.77	0.29	9559	1.12	0.25	37124	7.15	1.42
nist-b-02-t	12517	0.74	0.25	8278	1.26	0.25	33910	10.31	1.54
nist-b-03-t	14764	0.72	0.32	10142	1.17	0.28	39073	9.28	1.73
nist-b-04-t	5895	0.22	0.10	4986	0.51	0.12	14243	1.26	0.42
nist-b-05-t	10040	0.55	0.20	7150	0.81	0.18	27118	4.28	0.96
nist-g-01-t	10386	0.44	0.21	7996	0.87	0.20	28375	4.31	0.98
nist-g-02-t	9845	0.39	0.20	7820	0.90	0.18	26316	2.88	0.82
nist-g-03-t	13804	0.73	0.29	9618	1.26	0.26	36329	9.31	1.62
nist-g-04-t	18215	0.99	0.39	12657	1.50	0.36	49874	11.02	2.13
nist-g-05-t	5834	0.21	0.09	4764	0.50	0.10	14158	1.24	0.40
$\vec{\delta}$	$(80,80,80)^T$			$(120,120)^T$			$(120,120,120)^T$		
inst.	$\|T^c\|$	EVNIS t[s]	NIS t[s]	$\|T^c\|$	EVNIS t[s]	NIS t[s]	$\|T^c\|$	EVNIS t[s]	NIS t[s]
nist-u-01-t	25032	6.44	1.05	49754	26.19	2.92	106756	81.13	10.42
nist-u-02-t	25560	7.84	1.22	50033	38.79	3.17	108946	145.43	12.85
nist-u-03-t	44834	18.91	2.55	65888	113.54	5.75	239528	832,56	49.66
nist-u-04-t	14490	3.58	0.57	30482	11.09	1.50	54618	34.80	4.47
nist-u-05-t	10565	2.22	0.43	20791	6.18	0.96	38467	25.03	3.08
nist-b-01-t	39870	13.90	1.99	68219	72.26	4.96	191019	392.56	27.84
nist-b-02-t	38826	17.85	2.2	58296	107.66	4.95	194689	640.82	34.85
nist-b-03-t	42600	16.31	2.33	71028	98.76	5.98	206608	524.83	32.72
nist-b-04-t	13611	3.07	0.55	27264	8.83	1.25	51060	35.56	4.00
nist-b-05-t	27287	8.26	1.36	50953	41.95	3.48	120638	184.82	15.06
nist-g-01-t	26776	10.00	1.28	52880	38.02	3.26	112871	175.18	13.50
nist-g-02-t	25548	6.67	1.16	50035	25.99	2.64	109406	102.52	10.48
nist-g-03-t	42035	18.13	2.29	65499	113.93	5.34	198267	613.28	34.47
nist-g-04-t	52759	18.67	2.81	90508	125.90	6.86	248842	564.38	37.51
nist-g-05-t	13130	2.71	0.52	26763	9.45	1.31	50303	32.88	3.87

4. A Memetic Algorithm for Solving the *k*-MLSA Problem

This chapter presents a memetic algorithm for the solution of the *k-node minimum labeled spanning arborescence (k-MLSA)* problem. After a short general introduction to genetic algorithms and known technics for solving the related *MLST* problem, the newly developed *memetic algorithm (MA)* is described in detail. In the menetic algorithm, feasible arborescences are searched very frequently using depth first searches (DFS). A technique to reduce the number of DFS calls is introduced in the next part of the chapter. At the end of the chapter the results are presented and compared with a *GRASP* approach.

4.1 Theoretical Background

Evolutionary algorithms are stochastic search methods that imitate the principles of evolution theory, selection, recombination and mutation. The term genetic algorithm was first used by Holland [26], although in the 1960s similiar ideas where developed in Germany by Rechenberg [42] and Schwefel [44] (evolution strategies). In recent decades, various different types of evolutionary algorithms have been developed. The best known are *evolution strategy (ESs)*, *genetic algorithms (GAs)*, *genetic programming (GPs)* and *evolutionary programming (EP)*. Although there is no strict definition of what exactly falls under the name *GA*, most *GAs* start from an initial population, i.e. a set of solutions, where each solution is represented by a chromosome, which is then evolved by applying the three operators, selection, mutation and recombination.

In the original form of *GA*, the population is represented as a set of strings, which are called chromosomes in analogy to evolution theory. Each chromosome represents an individual. The elements of a chromosome are called genes. Each individual has a so-called fitness, a mapping of the value of its objective function. The selection operator selects individuals from the population according to their fitness. From these indiviudals the next generation is built. The encoding of a chromosome is called its genotype. Genetic algorithms operate on the genotype and evaluate the phenotype. Thus, search and solution space are distinct. A good genotype to phenotype mapping is essential for the efficacy of the algorithm.

At the start of the algorithm, a set of initial solutions is constructed, which may or may not be feasible. The goal is to find better solutions in the following generations. In each generation, new solutions are generated by pairwise recombination of individuals, selected according to their fitness values. The class of methods used for recombination is called crossover. The newly generated individuals are then slightly modified in the mutation step [37, 4]. The purpose of the mutation step is to maintain the diversity of the population by introducing new or previously lost genetic material. If local improvement methods are applied as part of the *GA*, the resulting hybrid algorithm is often called *memetic algorithm MA*. In the following, we describe the basic elements of a genetic algorithm in more detail. The genetic algorithm starts with a randomly distributed initial population of chromosomes selected from the search space. This initial population should cover the whole search space and can either be uniformly distributed or use a bias usually torwards the global optimum. In the selection

step chromosomes with high fitness values are selected as parent chromosomes. This selection can either be stochastic or deterministic.

An example for stochastic selection is tournament selection, where k solutions are selected random uniformly. Then a tournament is held over the k solutions and the winner of it is chosen as a parent chromosome. The tournament size k goes from two (binary tournament) to the size of the population.

Probably the most common stochastic selection technique is stochastic sampling with replacement, also called roulette wheel or fitness proportional selection. The selection probability is based on the relative fitness of each chromosome, and the fittest members of the population are selected with the highest probability. The name roulette wheel has its origin in the analogy to a roulette wheel, where the amount of space a chromosome occupies reflects its fitness. Another technique is elitist selection, where the best or the best and some good chromosomes are always selected for the next generation, while the rest of the chromosomes is selected using different criteria. This technique increases the performance of the *GA*, because good chromosomes are not lost and the best solution is guaranteed to survive the evolutionary process.

In the selection of parent chromosomes, there is a danger of converging to a local optimum resulting from a population that becomes too uniform. We denote the selection of fitter parents as selection bias. In order to avoid local optima, the factor that determines the selection bias in favor of good solutions must not be set too high. If, on the other hand, a very low selection bias is chosen, the algorithm does not converge quickly enough and degrades into a random search.

In the recombination step the selected chromosomes are combined to a new chromosome through crossover operations. There are many types of crossover. Some of the more important ones are single-point crossover, two-point crossover, multi-point crossover, uniform crossover and arithmetic crossover. In single-point crossover, a crossover point is chosen randomly. Everything before this crossover point, including the point itself, is copied from the first parent into the first newly generated chromosome. Everything after the crossover point is copied from the second parent into the first new chromosome. The reverse is done for the second offspring. In two-point crossover, the same operation is performed with two crossover points, and the string is interpreted as ring structure. Multi-point crossover is a generalization of two-point crossover.

Uniform crossover is a generalization of multi-point crossover insofar as multi-point crossover defines places where a chromosome can be split while uniform crossover decides for each gene from which parent it is taken. Besides this basic crossover operation, there are many more problem specific crossover types. In the mutation step a small part of the chromosome is changed either randomly or according to some criteria. In this way, lost material is brought back into the population.

The solution can be further improved by local search methods. Hybridizing local and genetic search can lead to large performance improvements. Combining these two features exploits complementary properties of both strategies. Global exploration of the search space is performed by the genetic search, while local search is used for local exploitation around the chromosomes [37]. The type of local search used is extremly problem dependent.

There are different strategies to combine the GA with a local search method. One strategy is to use local search only once after the GA proper has finished. Another strategy is to include it in each generation, either unconditionally or with some probability. Usually, local search is invoked at least a few times during the run of the GA.

Local search may affect only the fitness function or both the fitness function and the genotype of the indivudual. This is the difference between Lamarckian evolution, where learning (i.e. local search)

also affects the genotype, and the Baldwin effect, where only the fitness of an individual is affected.

In the Lamarckian case, the changed individual is copied back into the population, while in the Baldwin case only the fitness function is changed. The Lamarckian search is much faster but can have the disatvantage to converge towards a local optimum [25]. A genetic algorithm combined with local search methods is often called *memetic algorithm (MA)*.

There are two main types of genetic algorithm in terms of offspring construction. Stationary, also called steady-state GAs, and generational GAs. In a generational GA, a new offspring is generated from the members of the old population and placed in a new population. The new population has the same size as the old population and replaces it completely.

In a steady-state GA, only a single offspring is generated and replaces either the worst, a relatively bad (tournament selection) or a randomly chosen chromosome from the existing population. Duplicates are eliminated. The steady-state GA converges faster than the generational GA, but it can have the disadvantage to explore the landscape less thoroughly than the generational GA as its population is less diverse. It is still debated why and how genetic algorithms work and several attempts have been made to explain their functionality theoretically.

4.2 Related Work

The *memetic algorithm* for solving the *k-MLSA* problem, which is described in the following, is mainly based on Xiong et al. [52] as well as Nummela and Julstrom [39] who solved the similiar, NP-hard *MLST* problem with genetic algorithms. Xiong et al. [52] proposed a *GA* which encodes a candidate solution as a set of labels. A characteristic feature of this *GA* is that it uses only a single parameter p, which determines the population size. The fitness of an individual is the number of labels. The objective function minimizes the number of labels. Encoding only the labels is much easier than encoding the spanning tree. It is also sufficient, since all spanning trees induced by the labels are solutions of the same quality as the structure of the tree does not matter but only the number of labels has an impact on the objective function.

For the initial population, the chromosomes are built by adding randomly selected labels (genes) until the solution is feasible. The crossover operator selects the labels in an offspring solution from the union of the parents. First, the union of the parents is built, then the labels are sorted in descending order of their frequency. The offspring is created by adding labels according their sort order until the solution is feasible. In the mutation step, one label is added and redundant labels are removed. This removal of redundant labels can also be seen as local search step.

A time consuming operation in this algorithm is to validate a solution using *depth first searches (DFSs)*. A single *DFS* has the running time $O(m + n)$, where m denotes the numer of edges and n denotes the number of nodes. Since *DFSs* are performed after each addition of a label, the running time for crossover and mutation is $O(l(m+n))$ each, where l denotes the number of labels. If p denotes the number of generations and the population size, p crossover and p mutation steps are performed in each generation. Thus, the worst case running time of the algorithm is $O(p^2 l(m + n))$.

Another, very similiar, *GA* was developed by Numella and Julstrom [39]. A characteristic of this *GA* is that a chromosome encodes all labels, and the fitness of a chromosome is the number of labels needed to build a feasible solution.Thus, the labels at the beginning of the chromosome encode a feasible solution. This feasible solution is called feasible set of the chromosome.

In the crossover operation, called alternating crossover, the labels are alternatingly taken from the

parent chromosomes and added to the offspring. If there are duplicates, the later occurrence of a label is eliminated. In the mutation step, either two randomly chosen labels are swapped, or one label from the feasible set is swapped with a label from outside this set. A local search step tries to reduce the number of labels needed for a feasible solution by reordering the labels. The *GA* is 1−elitist, i.e. it always preserves the best chromosome.

The following text is mostly a repetition of the description of the *MA* in [18], which was my contribution to the paper. For the same problem, a *GRASP* method was developed by Dietzel as part of the same project, see [22] and [18].

4.3 Steady-State MA

The *MA* is based on a steady-state framework, where in each iteration a single offspring solution is derived and locally improved.

Algorithm 6: k-MLSA-MA()

```
1 randomly create initial population
2 t ← 0
3 while t < t_max do
4     select parents T' and T'' by tournament selection
5     T ← crossover(T', T'')
6     mutation(T)
7     local improvement(T)
8     t ← t + 1
9 end
```

It replaces a randomly chosen candidate solution from the population, to retain diversity. The algorithm uses tournament selection, and local improvement steps are performed for each new candidate solution after the application of the evolutionary operators, i.e. recombination and mutation. Algorithm 6 shows the overall framework where T denotes a chromosome.

4.3.1 Encoding

To create a feasible solution, it must be possible to build a directed rooted spanning tree from the edges represented by the chosen labels. The spanning tree has to include either all nodes or a subset of nodes whose size is determined by a given size constraint k. In other words a feasible solution is a solution that contains a $k - node$ arborescence [17].

During the iterations of the memetic algorithm it is sufficient to compute solutions that are known to contain a k-node arborescence. Redundant edges (and vertices if $k < n$) are eliminated for the best chromosome only after the memetic algorithm has finished.

If local improvement is not executed in every generation, redundant labels also have to be removed in the postprocessing. A feasible solution contains many subgraphs. Xiong et al. [52] pointed out that this structure has two important properties: 1. If we have a feasible solution with n labels and G' is a subgraph contained in this solution, every spanning tree of G' has at most n labels. 2. If G' is the

subgraph of an optimal solution then any spanning tree of G' is a minimum labeling spanning tree. These properties also apply to a directed graph, as in our case.

Since the solution is derivable from the permutation of the labels, a chromosome only needs to encode this permutation. Thus following the ideas presented in [52] we encode a candidate solution as an ordered subset of labels. In our case the template arcs correspond to these labels and the chromosome of a candidate solution is therefore denoted by T, $T[i]$ denotes the i-th template arc of candidate solution T.

If these template arcs induce a k-node arborescence we have a feasible solution, otherwise further template arcs need to be added to the candidate solution in order to make the solution feasible. Note however, that a feasible solution may contain redundant template arcs, which are not necessarily part of an optimal solution induced by the other template arcs of the ordered set.

For candidate solutions of the initial population we ensure that they are feasible. To create a randomized candidate solution, all template arcs are shuffled and then added as long as the candidate solution remains infeasible.

The MA then tries to minimize the number of template arcs required for a feasible solution by iterative application of the genetic operators and local improvement. As many candidate solutions have the same number of template arcs, the total number of induced arcs is also considered in the fitness function $f(T)$, which is going to be minimized.

4.3.2 Fitness Function

The fitness function which is minimized evaluates the number of template arcs $|T|$ which are required for a feasible solution and is given by:

$$f(T) = |T| + \left(1 - \frac{|A'|}{|A|}\right). \tag{4.3.1}$$

Again, A' denotes the set of induced tree arcs. This accounts for the fact that candidate solutions whose template arcs cover many arcs are more likely to produce good offspring and result in successful mutations.

4.3.3 Initialization

To get a randomized initial population the labels are shuffled with a simple shuffle algorithm, first published by Fisher and Yates [29].

Algorithm 7: Shuffling

1. 1. initialization $j \leftarrow n$;
2. 2. Generate a random number U, uniformly distributed between zero and one;
3. 3. set $k = \lfloor j \cdot U \rfloor + 1$;
4. Swap $X_k \leftrightarrow X_j$;
5. 4. Decrease j by one.;
6. If $j > 1$ return to step 2;

The initial population consists only of chromosomes which encode feasible solutions. To create a feasible solution, labels from the shuffled permutation are added to the chromosome until the graph (we call it permutation-graph) built from the edges represented by those labels contains a k-node arborescence.

4.3.4 Crossover

Since the order of the template arcs does not need to be preserved, we use a crossover operator introduced in [39], which takes the template arcs for the child candidate solution alternatingly from the parents until a feasible solution is obtained.

Algorithm 8: `crossover(`T'`, `T''`)`

```
1  T ← ∅  // new offspring initialized with empty set
2  i ← 0, j ← 0  // counter variables
3  while T contains no k-MLSA do
4      if i mod 2 = 0 then
5          t ← T'[⌊i/2⌋]
6      else
7          t ← T''[⌊i/2⌋]
8      end
9      if t ∉ T then
10         T[j] ← t
11         j ← j + 1
12     end
13     i ← i + 1
14 end
15 return T
```

4.3.5 Mutation

In addition to recombination we use three different types of mutation:

1. A randomly selected template arc $t \notin T$ is appended. This increases the likelihood for the ability to remove some redundant template arc by a subsequent local improvement.

2. A randomly selected template arc $t \notin T$, replaces either a random or the worst $t' \in T$. The worst template arc is the one inducing the minimal number of arcs. If the solution is not feasible, further randomly selected template arcs are added until a feasible solution is reached.

3. The permutation is sorted by the number of edges the labels represent. A label from outside the permutation is randomly selected. Beginning with the worst label of the permutation an attempt is made to replace each label with the new label. Once an attempt results in a feasible solution, the substitution is performed and the mutation ends. If no replacement leads to a feasible solution the mutation leaves the chromosome unchanged. A variation of this algorithm

tries to remove a randomly selected label. In the tests mutation type 1 performed considerably worse than type 2 and type 3.

Algorithm 9: Mutation

```
1  if mutation type 1 then
2      add randomly selected label not in perm;
3  end
4  else if mutation type 2 then
5      replace worst (random) label with randomly selected label not in perm;
6      add labels until perm contains an k-node arborescence;
7  else if mutation type 3 then
8      if replace worst label then
9          sort labels in descending order of edge frequency;
10         l = perm.size;
11         while l > 0 do
12             replace l with randomly selected label not in perm;
13             if G is k-node arborescence then
14                 l = 0;
15             else
16                 undo replacement;
17                 l = l − 1;
18             end
19         end
20     end
21     else if replace random label then
22         i = perm.size();
23         while i > 0 do
24             replace randomly selected label from perm with randomly selected label not in
               perm ;
25             if G is k-node arborescence then
26                 i = 0;
27             else
28                 undo replacement;
29                 i = i − 1;
30             end
31         end
```

4.3.6 Local Improvement

The subsequent local improvement method `local-improvement(`T`)` (Algorithm 10), following the one presented in [39], uses the idea that a reordering of the template arcs could make some of them redundant. In contrast to the local improvement method used in the GRASP algorithm this method

can only remove template arcs from a current solution if some of them are actually redundant. As the MA continuously modifies the candidate solutions from the population and also further template arcs are added to a candidate solution by mutation, there is no need to use a more expensive neighborhood search, which also considers currently unused template arcs.

Algorithm 10: `local-improvement(`T`)`

```
1  i ← 0  // counter variable
2  while i < |T| do
3      remove all arcs only labeled by T[i]
4      if T contains k-MLSA then
5          T ← T \ T[i]
6      else
7          restore respective arcs
8          i ← i + 1
9      end
10 end
```

Postprocessing is performed on the best chromosome after the MA has finished. Redundant labels, edges and nodes (in the case of $k <$ number of nodes) are deleted from the solution and the root node of the tree is set.

4.4 Check if the Tree Contains an Arborescence

Obviously, the algorithm frequently has to check if a partial solution already contains a feasible arborescence. This task can be achieved by performing depth first search (DFS) using each node as start node (time complexity $O(k^3)$). To achieve a speedup of this method we try to avoid or reduce the number of time consuming DFS calls. Let G' denote the graph containing just the edges and nodes induced by some template arc set T, i.e. if $(i, j) \in A$ is represented by template arc $t \in T$ we add the nodes i, j and the arc (i, j) to $G' = (V', A')$. Let further $\delta^-(v)$ denote the in-degree of a node v, i.e. the number of incoming arcs. Furthermore let $\delta_0^i(V')$ denote the subset of nodes from V' with $\delta^-(V') = 0$, and let us assume that the current partial solution consists of the template arcs (labels) T. Following the idea of Dietzel [22] we first check the degree of each node to see if a sufficient number of nodes v with in-degree $\delta^-(v) > 0$ is available. If $|V'| - \delta_0^i(V') + 1 < k$ then G' cannot represent a valid solution, and we do not have to perform the DFS. If a solution is possible we distinguish the following two cases. In the first case, where $k = |V|$, there can be at most one node with in-degree zero. If there is such a node it has to be the root node and we perform the DFS starting from this node. Otherwise, if all nodes $v \in V'$ have $\delta^-(v) > 0$ we have no choice but to perform DFS starting from all nodes. In the more general second case $k < |V|$, if $|V'| - \delta_0^i(V') + 1 = k$, one of the nodes with in-degree zero has to be the root of the tree, otherwise the tree would not contain the required k nodes. So it is sufficient to perform the DFS starting at just these $\delta_0^i(V')$ nodes. Otherwise we again have to perform DFS starting from all nodes.

Table 9: Runtime comparison of unimproved and improved DFS elimination methods

inst	$\|V\|$	$\vec{\delta}$	k	m	I t[s]	II t[s]	III t[s]
ft-01	31	$(15,15,15)^T$	31	11	19.00	6.67	2.66
ft-01	31	$(5,5)^T$	31	9	17.18	7.07	2.31
ft-02	28	$(5,5)^T$	28	10	12.30	5.14	2.32
ft-03	35	$(5,5)^T$	35	10	29.65	9.43	3.30
ft-04	20	$(5,5)^T$	20	9	6.11	2.57	1.36
ft-05	39	$(15,15,15)^T$	39	11	35.37	13.02	3.73
ft-05	39	$(5,5)^T$	39	11	33.12	11.76	3.68
ft-06	15	$(5,5)^T$	15	7	3.24	1.53	0.90
ft-07	28	$(5,5)^T$	28	9	16.31	5.92	2.52
ft-08	27	$(5,5)^T$	27	10	13.01	5.35	2.19
ft-09	27	$(5,5)^T$	27	10	13.56	4.98	2.27
ft-10	31	$(5,5)^T$	31	11	18.40	6.60	2.76
ft-11	38	$(5,5)^T$	38	11	32.50	10.07	3.57
ft-12	28	$(5,5)^T$	28	8	11.91	4.79	2.05
ft-13	25	$(5,5)^T$	25	9	10.99	4.50	2.05
ft-14	33	$(5,5)^T$	33	10	25.33	8.91	3.22
ft-15	29	$(5,5)^T$	29	9	15.31	4.73	1.96
ft-16	37	$(5,5)^T$	37	11	33.89	11.12	2.91
ft-17	31	$(5,5)^T$	31	11	18.81	6.24	2.64
ft-18	40	$(5,5)^T$	40	12	38.53	11.28	4.03
ft-18	40	$(15,15,15)^T$	40	10	37.50	12.31	3.85
ft-19	35	$(5,5)^T$	35	12	29.32	10.26	3.21
ft-20	28	$(5,5)^T$	28	10	15.94	7.05	2.15
ft-01	31	$(5,5)^T$	25	7	6.91	4.31	3.66
ft-01	31	$(5,5)^T$	30	9	12.16	5.14	2.48
ft-01	31	$(15,15,15)^T$	25	8	6.88	4.41	2.89
ft-01	31	$(15,15,15)^T$	30	10	13.30	5.24	2.44
ft-05	39	$(5,5)^T$	30	8	13.07	7.69	5.32
ft-05	39	$(15,15,15)^T$	30	8	14.14	5.61	4.08
ft-18	40	$(5,5)^T$	30	8	13.87	8.60	6.36
ft-18	40	$(15,15,15)^T$	30	7	12.00	6.96	6.11
ft-18	40	$(15,15,15)^T$	35	9	17.54	7.54	3.76

4.5 Results

All tests where performed on a Pentium 4 with 2GB memory under Linux kernel 2.4.21. The programming language, libraries, framework and compiler version are the same as specified before in Section 3.4. For the memetic algorithm we used EAlib2, developed at the Institute for Computer Graphics and Algorithms of the Vienna University of Technology. Table 9 shows the much lower runtime of the *MA* if DFS calls are avoided (see Section 4.4). We differentiate the basis (I, II) and the improved versions (III) of the algorithm. Version I performs all DFS without any improvement with regard to the running time. Version II avoids some DFS calls (If there are not enough edges or to many nodes with in-degree zero no DFS is performed). This version adapts Dietzels *containsArborescence* function [22]. Version III implements all the improvements described before. The table also lists the instance *inst*, the number of nodes $|V|$, the size of the domain $\vec{\delta}$, the size of the computed tree k and the number of labels in the solution $|m|$. We can see that both improvements reduce the running time by half.

The results of the memetic algorithm are presented in Table 10. The first three columns show the instance names and parameters k and $\vec{\delta}$. Then, in the first part of the table, we list the results from the exact branch-and-cut method, the second part contains the objective value of the currently best

Table 10: Results and running times of the memetic algorithm

inst	k	$\vec{\delta}$	m	$t[s]$	m_{best}	m_{avg}	σ_m	#b.s. [%]	$t_{\text{avg}}[s]$	m_{best}	m_{avg}	σ_m	#b.s. [%]	$t_{\text{avg}}[s]$	$\rho_{\text{b.s.}}[\%]$
			B&C results		MA (it = 10000, $size_{pop}$ = 100)					MA (it = 30000, $size_{pop}$ = 100)					
ft-01			3	79	3	3.70	0.47	30	1.84	3	3.23	0.42	77	5.58	6.10
ft-02			3	0	3	3.04	0.18	97	1.38	3	3.00	0.00	100	4.29	6.10
ft-03			3	85	3	3.00	0.00	100	2.45	3	3.00	0.00	100	7.51	6.10
ft-04			4	10	4	4.00	0.18	100	1.01	4	4.00	0.00	100	3.08	-0.87
ft-05			3	327	3	3.00	0.00	100	2.23	3	3.00	0.00	100	6.48	6.10
ft-07			4	14	4	4.00	0.00	100	1.49	4	4.00	0.00	100	4.62	-1.57
ft-08			4	13	4	4.00	0.00	100	1.50	4	4.00	0.00	100	4.08	-1.57
ft-09			4	24	4	4.00	0.00	100	1.35	4	4.00	0.00	100	4.54	-1.57
ft-10			3	36	3	3.43	0.50	57	1.53	3	3.17	0.38	84	5.79	6.10
ft-11	20	$\begin{pmatrix}30\\30\\30\end{pmatrix}$	3	853	3	3.00	0.00	100	2.01	3	3.00	0.00	100	6.50	6.10
ft-12			3	52	3	3.00	0.00	100	2.26	3	3.00	0.00	100	4.86	6.10
ft-13			3	21	3	3.03	0.18	97	1.59	3	3.00	0.00	100	3.60	6.10
ft-14			3	275	3	3.16	0.37	83	1.15	3	3.00	0.00	100	7.04	6.10
ft-15			3	15	3	3.00	0.00	100	2.33	3	3.00	0.00	100	4.88	6.10
ft-16			3	282	3	3.00	0.00	100	1.69	3	3.00	0.00	100	7.50	6.10
ft-17			3	235	3	3.46	0.48	67	1.82	3	3.17	0.38	84	5.36	6.10
ft-18			3	823	3	3.00	0.00	100	2.75	3	3.00	0.00	100	8.50	6.10
ft-19			3	97	3	3.06	0.18	97	2.69	3	3.00	0.00	100	8.05	6.10
ft-20			3	0	3	3.00	0.00	100	1.58	3	3.00	0.00	100	4.79	6.10
			best known solution		MA (it = 10000, $size_{pop}$ = 100)					MA (it = 60000, $size_{pop}$ = 200)					
nist-b-01			4	n/a	**5**	5.10	0.31	0	24.66	**5**	5.00	0.00	0	98.90	13.75
nist-b-02			4	n/a	4	4.74	0.44	27	15.16	4	4.20	0.40	80	89.95	18.90
nist-b-03			4	n/a	4	4.60	0.50	40	17.94	4	4.10	0.31	90	104.43	18.90
nist-b-04			5	n/a	5	4.94	0.37	10	16.32	5	5.60	0.50	40	91.75	13.75
nist-b-05			4	n/a	**5**	5.94	0.51	0	16.86	4	5.00	0.26	3	93.73	18.90
nist-g-01			4	n/a	4	4.87	0.35	14	17.57	4	4.64	0.49	33	101.82	18.90
nist-g-02			5	n/a	**6**	6.17	0.38	0	21.03	5	5.97	0.18	3	117.78	13.38
nist-g-03	40	$\begin{pmatrix}80\\80\\80\end{pmatrix}$	4	n/a	4	4.80	0.41	46	16.36	4	4.30	0.47	70	94.29	18.90
nist-g-04			4	n/a	**5**	5.38	0.49	0	21.38	**5**	5.00	0.00	0	115.99	13.38
nist-g-05			5	n/a	5	6.57	0.57	3	16.97	5	5.74	0.45	26	92.97	13.75
nist-u-01			5	n/a	**6**	6.90	0.31	0	21.17	**6**	6.10	0.31	0	117.19	11.03
nist-u-02			5	n/a	5	5.60	0.49	40	17.54	5	5.00	0.00	100	100.68	13.75
nist-u-03			4	n/a	**5**	5.04	0.18	0	17.41	4	4.50	0.50	50	98.18	18.90
nist-u-04			5	n/a	5	5.84	0.38	17	17.75	5	5.24	0.43	76	98.87	13.75
nist-u-05			5	n/a	**6**	6.37	0.49	0	15.30	**6**	6.04	0.18	0	83.56	11.47

known solution. The column m_{best} shows the best result of 30 runs of the algorithm, column m_{avg} shows the average value. By σ_x we denote the standard deviation of the entity x. The column $\#b.s.$ shows the percentage of runs, where the solution listed in m_{best} has been found. Average running times are listed in column t_{avg} and $\rho_{\text{b.s.}}$ shows the achieved compression ratio. We used a population size $size_{\text{pop}} \in \{100, 200\}$, and a group size of four for the tournament selection. The crossover and mutation probability is set to one, i.e. each offspring is created by crossover and subsequent mutation. In each iteration a randomly selected candidate solution from the population was replaced by the newly generated one. Local improvement is performed for each newly created candidate solution. As mutation type 2 produced better overall results than mutation type 1, the former was used to create the results listed in Table 10. Replacing a randomly selected $t \in T$ turned out to be advantageous over replacing the worst one. Table 10 shows the results of 30 runs with 10000 and 30000 iterations for the Fraunhofer templates (population size 100); for the NIST templates we list the results for 10000

and 60000 iterations with a population size of 100 and 200, respectively. Again, the presented results are not essentially different to the ones for any other parameter settings of k and $\vec{\delta}$. The Fraunhofer data can be compressed within 10000 iterations, which takes an average running time of roughly 2 seconds. Due to the larger number of points, the compression of the NIST data is computationally more expensive. At least 60000 iterations must be used in order to be able to produce good results. The respective running times are roughly 100 seconds.

Table 11 shows the results of the GRASP algorithm developed by Dietzel [22] for the same parameter settings. The average running time to find good solutions w.r.t. our application background (i.e. to find the optimal solution in most of the cases) is roughly less than ten seconds for the Fraunhofer templates. Due to their larger size it is much more expensive to solve the NIST data. In this case the running times range from less than one minute to slightly more than three minutes.

In the case of the Fraunhofer data the MA is clearly superior to the GRASP, as it produces better solutions in less time. For the NIST data GRASP clearly outperforms the MA, if we allow higher running times of up to five minutes.

Table 11: Results and running times of the GRASP

instance	k	$\vec{\delta}$	m	$t[s]$	m_{best}	m_{avg}	σ_m	#b.s. [%]	$t_{\text{avg}}[s]$	m_{best}	m_{avg}	σ_m	#b.s. [%]	$t_{\text{avg}}[s]$	$\rho_{\text{b.s.}}[\%]$
			B&C results		GRASP $it = 10, it_{ls} = 5, rcl_{\max} = 5, imp_{\max} = 0$					GRASP $it = 10, it_{ls} = 20, rcl_{\max} = 10, imp_{\max} = 10$					
ft-01			3	79	**4**	4.00	0.00	100	1.57	3	3.87	0.34	13	7.97	6.10
ft-02			3	0	**4**	4.00	0.00	100	1.00	3	3.67	0.47	33	6.20	6.10
ft-03			3	85	3	3.00	0.00	100	2.03	3	3.00	0.00	100	10.03	6.10
ft-04			4	10	4	4.70	0.46	30	0.15	4	4.30	0.46	70	2.03	-0.87
ft-05			3	327	3	3.00	0.00	100	4.82	3	3.00	0.00	100	16.90	6.10
ft-07			4	14	4	4.00	0.00	100	0.33	4	4.00	0.00	100	4.00	-1.57
ft-08			4	13	4	4.00	0.00	100	0.93	4	4.00	0.00	100	3.93	-1.57
ft-09			4	24	4	4.00	0.00	100	0.33	4	4.00	0.00	100	4.00	-1.57
ft-10			3	36	3	3.13	0.34	87	1.00	3	3.00	0.00	100	6.30	6.10
ft-11	20	$\begin{pmatrix}30\\30\\30\end{pmatrix}$	3	853	3	3.00	0.00	100	4.27	3	3.00	0.00	100	23.07	6.10
ft-12			3	52	3	3.00	0.00	100	1.00	3	3.00	0.00	100	5.93	6.62
ft-13			3	21	3	3.00	0.00	100	0.20	3	3.00	0.00	100	2.97	6.62
ft-14			3	275	3	3.00	0.00	100	2.00	3	3.00	0.00	100	8.73	6.10
ft-15			3	15	3	3.00	0.00	100	1.00	3	3.00	0.00	100	5.93	6.62
ft-16			3	282	3	3.00	0.00	100	2.97	3	3.00	0.00	100	16.53	6.10
ft-17			3	235	3	3.00	0.00	100	1.03	3	3.00	0.00	100	5.97	6.10
ft-18			3	823	3	3.00	0.00	100	5.57	3	3.00	0.00	100	21.93	6.10
ft-19			3	97	3	3.00	0.00	100	1.90	3	3.00	0.00	100	8.10	6.10
ft-20			3	0	3	3.00	0.00	100	0.30	3	3.00	0.00	100	3.00	6.10
			best known solution		$it = 5, it_{ls} = 5, rcl_{\max} = 10, imp_{\max} = 0$					$it = 5, it_{ls} = 10, rcl_{\max} = 10, imp_{\max} = 5$					
nist-b-01			4	n/a	4	4.83	0.37	17	91.60	4	4.50	0.50	50	189.97	18.90
nist-b-02			4	n/a	4	4.37	0.48	63	78.47	4	4.00	0.00	100	180.17	18.90
nist-b-03			4	n/a	4	4.43	0.50	57	89.57	4	4.03	0.18	97	190.67	18.90
nist-b-04			5	n/a	5	5.27	0.44	73	24.70	5	5.00	0.00	100	53.97	13.75
nist-b-05			4	n/a	4	4.93	0.25	7	73.37	4	4.77	0.42	23	157.47	18.90
nist-g-01			4	n/a	4	4.83	0.37	17	71.23	4	4.37	0.48	63	159.57	18.90
nist-g-02			5	n/a	5	5.73	0.44	27	67.50	5	5.40	0.49	60	151.13	13.38
nist-g-03	40	$\begin{pmatrix}80\\80\\80\end{pmatrix}$	4	n/a	4	4.17	0.37	83	94.50	4	4.00	0.00	100	189.57	18.90
nist-g-04			4	n/a	**5**	5.00	0.00	0	85.77	4	4.97	0.18	3	201.70	18.60
nist-g-05			5	n/a	5	5.13	0.34	87	40.13	5	5.03	0.18	97	64.57	13.75
nist-u-01			5	n/a	**6**	6.03	0.18	0	68.90	5	5.90	0.30	10	156.70	18.38
nist-u-02			5	n/a	5	5.00	0.00	100	69.87	5	5.00	0.00	100	152.37	13.75
nist-u-03			4	n/a	4	4.47	0.50	53	82.20	4	4.13	0.34	87	193.13	18.90
nist-u-04			5	n/a	5	5.17	0.37	86	44.53	5	5.00	0.00	100	74.83	13.75
nist-u-05			5	n/a	5	5.93	0.25	6	20.90	5	5.70	0.46	30	42.77	13.75

5. Branch-and-Cut-and-Price

This chapter presents an exact method to solve the *k-MLSA* problem. It gives a short general introduction to *BCP* and presents the ILP model as well as the cutting plane separation with cycle elimination cuts and directed connection inequalities. The separation of the directed connection cuts utilizes the max-flow min-cut theorem, which states that the maximum flow in a network equals the value of the minimum cut. The separation of these cuts can be improved by back-cuts and creep flow. New variables are generated by solving the pricing-problem which is based on the values of the dual variables of the current solution and added dynamically to the ILP model. After presenting the pricing model, the different variants of branch-and-cut-and-price (BCP) algorithms are described. Furthermore, preprocessing strategies with the emphMA as an initial heuristic and the computation of a lower bound for a reduced version of the problem are discussed. Finally different *BCP* variants are compared based on computational experiments.

5.1 Theoretical Background

The foundation of the mathematical discipline of linear programming dates back to 1947, when G.B. Dantzig developed the simplex algorithm [13]. The term Linear Programming (LP) denotes the class of optimization problems where the optimization criterion as well as the constraints are linear functions [19]. Thus, in its standard form an LP problem is a linear function, that has to be maximized (or minimized) and is subject to linear problem specific constraints, restricting the space of feasible solutions. In matrix form an LP looks as follows:

$$\text{maximize} \quad c^T x \tag{5.1.1}$$

$$\text{s.t.} \quad Ax \leq b, x \geq 0, \tag{5.1.2}$$

where $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $c \in \mathbb{R}^n$ are given, and $x \in \mathbb{R}^m$ has to be found. If the domain of the variables is restricted to integers the problem is called a Integer Linear Program (ILP). If only some of the variables are integers, it is called a Mixed Integer Program (MIP). Wolsey [49] gives a comprehensive introduction in integer programming.

5.1.1 Duality

Every LP called primal program has a corresponding dual program. The dual form of the LP given by (5.1.1) and (5.1.2) is:

$$\text{minimize} \quad b^T y \tag{5.1.3}$$

$$\text{s.t.} \quad A^T y \leq c, y \geq 0, \tag{5.1.4}$$

where $y \in \mathbb{R}^m$ has to be found. Every feasible solution of the dual program provides an upper bound for the optimal solution value of the primal program. The reverse is also true, i.e. every feasible solution to the primal program gives a lower bound for the optimal solution of the dual program. Thus, if feasible solutions for the primal and dual programs are found, and the solutions have the same

objective value, we have found the optimal solution. This insight is the central theorem of Linear Programming, the so called *LP-Duality Theorem*, which can be formalized as:

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} c_i y_i^* \tag{5.1.5}$$

where $x^* = (x_1^*, ..., x_n^*)$ and $y^* = (y_1^*, ..., y_n^*)$ are optimal solutions for the primal and the dual program [47].

5.1.2 Column Generation

The idea of column generation is to start with a small set of variables, the so-called Restricted Master Problem (RMP), and add new variables (columns) on demand. The following description is mainly based on the introduction to column generation by Desrosiers et al. [21]. Column generation has its origin in the simplex algorithm. In the simplex algorithm, in each iteration a new variable that can potentially improve the solution enters the basis.

Column generation follows the idea that only a small subset of the variables that can potentially enter the basis has to be considered. If we have a minimization problem, the next variable to enter the basis is the variable with the highest negative reduced costs. This variable can be found by solving an optimization problem, the so called pricing problem. Let us call the following problem the master problem *MP*:

$$\text{minimize} \sum_{i \in I} c_i \lambda_i \tag{5.1.6}$$

$$\text{s.t.} \sum_{i \in I} a_i \lambda_i \quad \geq \quad b, \qquad \lambda \geq 0, i \in I. \tag{5.1.7}$$

We are looking for a variable to enter the basis in each iteration of the simplex algorithm. As shown before (5.1.3) each constraint of the primal program has a corresponding dual variable. Let $\mathbf{u}$ denote the vector of non-negative variables of the dual problem. Then the reduced costs are given by

$$\overline{c}_i = c_i - \mathbf{u}^T A \tag{5.1.8}$$

and we want to find an $i \in I$ that minimizes $\overline{c}_i$. If we have a large $|I|$, this so called explicit pricing is costly. Thus we work only with a subset $I' \subset I$ of the columns. The reduced costs are evaluated implicitly be enumeration. The subset I' is called the restricted master problem (RMP). Let $\mathbf{u}$ be the optimal dual solution of the RMP. Assume that the columns a_i, $i \in I$ are elements of a set A and the cost coefficient is computed by applying function c to a_i. Then the pricing problem is the following subproblem: $\overline{c}^* = \min\{c(a) - \mathbf{u}^T a \mid a \in A\}$. If the value of $\overline{c}^* \geq 0$, the solution to the RMP also solves the MP optional. Otherwise the column is added to the RMP and the RMP is resolved [21]. Since the pricing problem will always identify a column with negative reduced costs, if one exists, the optimal solution to the linear program will be found [5]. A comprehensive survey of column generation methods is given in [35]. An outstandingly clear introduction to column generation is given in [13, 6].

5.1.3 Branch-and-Cut-and-Price

Branch-and-cut-and-price has its origin in the class of branch-and-bound algorithms. Branch-and-bound follows a divide and conquer strategy by partitioning a problem into smaller subproblems and optimizing each subproblem individually. When the bounding is achieved by LP techniques, the method is called LP-based branch-and-bound. In the usual case of LP-based branch-and-bound the integrality constraints of an IP are omitted and the so called LP-relaxation of the problem is solved. This solution provides a upper or lower bound to the objective function value of the considered problem [32].

This elementary idea was improved by branch-and-cut and branch-and-price, which are complementary techniques for solving linear integer programs. Both techniques are suitable for problems with a large number of variables and constraints. The idea of branch-and-cut is to omit inequalities from the LP-Relaxation in order to reduce the number of constraints, because most of them will not influence the optimal solution anyway. If an optimal solution is found, a subproblem, called separation problem, is solved in order to find violated inequalities. If violated inequalities are found, they are added to the relaxation in order to cut off infeasible solutions. The term cut has its origin in a graphical interpretation of the LP. Each LP can be represented as polyhedron, where the corner points or extreme points describe the optimal solution of the LP. Parts of this polyhedron are cut away when a valid inequality is added. Thus, the inequalities are called *cutting-planes* or simply *cuts*. After adding the inequalities the LP is resolved.

This operation takes place in each node of the branch-and-bound tree. If no more violated inequalities are found, the branching operation takes place [5]. The idea of branch-and-price is to start with a small set of variables and add new variables (columns) that can improve the solution on demand in the pricing process. If we have a minimization problem the next variable to enter the basis is a variable with negative reduced costs. To identify variables that can improve the solution and therefore enter the basis the so called pricing problem is solved. Note that the pricing problem is the separation problem for the dual LP. If new variables enter the basis the LP is resolved. When no further variables are found, branching is performed. The idea of branch-and-cut-and-price is to generate variables and constraints dynamically during a LP-based branch-and-bound process [32]. Since we deal with a large number of labels, and only a few of them are contained in a feasible solution, it seemed promising expand the branch-and-cut approach described in [18] to a branch-and-cut-and-price approach.

5.2 Related Work

The *MLST* problem is solved with mathematical programming techniques in [16, 14]. In [14] a branch-and-cut framework is presented to solve the MLST instances exactly. A polyhedral and a computational comparison of an underlying flow-formulation and a formulation based on directed connectivity cuts is given. Also odd-hole inequalities are applied to this problem for the first time. For the separation of cutting planes with odd-hole inequalities a heuristic that is based on Miller-Tucker-Zemlin inequalities is presented. The odd-hole inequalities improved the running time significantly for some classes of instances. The presented framework can solve benchmark instances within a significantly shorter running time than other approaches, and can solve new larger instances for the first time. A branch-and-cut-and-price approach which different variants of connectivity cuts is also developed in [14]. Other exact approaches based on mathematical programming are developed by Chen et al. [11] and Captivo et al. [8]. Chen et al. [11] presented the first MIP formulation which is based on Miller-

Tucker-Zemlin inequalities. Captivo et al. [8] proposed a mixed integer formulation for undirected graphs which is based on a single commodity flows.

As part of this project, the *k-MLSA* problem was solved using the exact methods *branch-and-cut (BC)* [18] and *branch-and-price (BP)* [46]. The *BCP* approach integrates the *BC* and the *BP* approach. It uses a similiar *ILP* model and the same cuts as the *BC* method. The pricing problem is the same as the one used in the *BP* algorithm.

The branch-and-price approach [46] presents a single-commodity flow as well as a multi-commodity flow formulation. Furthermore a static and a dynamic k-d tree based segmentation algorithm for the preprocessing is developed in [46]. The general idea of *BC* and *BCP* (referring to the work being topic of [46] and this thesis is also discussed briefly in [14].

5.3 Branch-and-Cut-and-Price for Solving the *k-MLSA* Problem

For the separation of cuts we mainly use the branch-and-cut approach developed in [18, 17]. The next part is mostly an repetition of the model described in [18]. The ILP model is the same, except for Equation (5.3.11) and Equation (5.3.8).

5.3.1 ILP Formulation

The set of nodes V is extended to V^+ by adding an artificial root node 0. Also, the set of arcs A is extended to A^+ by adding $(0, i),\ \forall i \in V$. The following variables are used in the ILP model of the problem:

- Variables $y_t \in \{0, 1\}$ indicate for each candidate template arc $t \in T^c$ whether it is part of the dictionary T.
- Variables $x_{ij} \in \{0, 1\},\ \forall (i, j) \in A^+$ indicate which arcs belongs to the tree.
- Variables $z_i \in \{0, 1\},\ \forall i \in V$ indicate whether a node is part of the tree.

Let $A(t) \subset A$ denote the set of tree arcs a template arc $t \in T^c$ can represent, and $T(a)$ denote the set of template arcs that can represent an arc $a \in A$, i.e. $T(a) = \{t \in T^c \mid a \in A(t)\}$. The k-MLSA problem is modeled as follows:

$$\text{minimize} \quad m = \sum_{t \in T^c} y_t \tag{5.3.1}$$

$$\text{s.t.} \sum_{t \in T(a)} y_t \geq x_a, \qquad \forall a \in A \tag{5.3.2}$$

$$\sum_{i \in V} z_i = k \tag{5.3.3}$$

$$\sum_{a \in A} x_a = k - 1 \tag{5.3.4}$$

$$\sum_{i \in V} x_{(0,i)} = 1 \tag{5.3.5}$$

$$\sum_{(j,i) \in A^+} x_{ji} = z_i \qquad \forall i \in V \tag{5.3.6}$$

$$x_{ij} \leq z_i \qquad \forall (i,j) \in A \tag{5.3.7}$$

$$x_{ij} + x_{ji} \leq z_i \qquad \forall (i,j) \in A \tag{5.3.8}$$

$$\sum_{a \in C} x_a \leq |C| - 1 \qquad \forall \text{ cycles } C \text{ in } G,\ |C| > 2 \tag{5.3.9}$$

$$\sum_{a \in \delta^-(S)} x_a \geq z_i \qquad \forall i \in V,\ \forall S \subseteq V,\ i \in S,\ 0 \notin S \tag{5.3.10}$$

$$\sum_{t \in T(\delta^-(i))} y_t \geq z_i - x_{0,i} \qquad \forall i \in V \tag{5.3.11}$$

Inequalities (5.3.2) enforce that each used tree arc $a \in A$ is represented by at least one valid template arc t. Equalities (5.3.3) and (5.3.4) enforce that the requested number of nodes and arcs are selected. Equations (5.3.5) ensure that there is exactly one arc from the artificial root to one of the tree nodes. Equations (5.3.6) enforce that the selected nodes have in-degree one. Inequalities (5.3.7) enforce that an arc can only be selected if its source node is selected. Inequalities (5.3.8) forbid cycles of length two. We improved the earlier version by using z_i instead of 1 to make the inequalities tighter. The value of z_i can be less than 1 when only a subset of k nodes is selected. Thus, by using z_i instead of 1 more cycles of length two are excluded. This is feasible because of Equations (5.3.6) and inequalities (5.3.7), which enforce that $x_{i,j}$ as well $x_{j,i}$ have to be less than or equal to z_i. Inequalities (5.3.9) forbid all further cycles C with $|C| > 2$. Directed connectivity-constraints can be added to strengthen the ILP. They lead to a tighter characterization of the spanning tree polyhedron, i.e. to better theoretical bounds compared to the cycle elimination cuts, but their separation is computationally more expensive. The directed connectivity-constraints are represented by inequalities (5.3.10), where $\delta^-(S)$ represents the ingoing cut of node set S. These constraints ensure the existence of a path from the root 0 to any node $i \in V$ contained in the solution. Although Equations (5.3.10) make Equations (5.3.6), (5.3.7), (5.3.8) and (5.3.9) redundant, using them together can be beneficial.

Optionally, we can add inequalities (5.3.11), which associate nodes with template arcs. They ensure that for each node, except the root node, the sum over the template arc variables that correspond to ingoing arcs of the node is equal or greater than one.

5.3.2 Cut Separation

Since the cut separation is described in [18], we give only a short recapitulation of this description here. The number of the cycle elimination inequalities (5.3.9) and connectivity inequalities (5.3.10) is exponential. Therefore, we use a branch-and-cut approach [49, 50], i.e. we only use the constraints (5.3.2) to (5.3.8) at the beginning and add the cycle elimination and connectivity inequalities on demand. The cycle elimination cuts are computed using Dijkstra's shortest path algorithm: First, each arc is assigned the weight $1 - x_{ij}^{LP}$, where x_{ij}^{LP} is the arc value of the respective LP-relaxation. Then each arc is iteratively selected as the start arc $(i, j) \in A$ and the shortest path from j to i is computed. If the value of all arcs contained in the path plus $1 - x_{ij}^{LP}$ is smaller than one, the path is a cycle for which inequalities (5.3.9) are violated. If a violated inequality is found, it is added to the LP, and the LP is resolved. This procedure is repeated until no further violated inequalities are found. To separate the directed connection cuts, we make use of the max-flow min-cut theorem [40, 3], which states that the maximum flow in a network equals the value of the minimum cut. Thus, we can separate the cuts

by computing the maximum flow from the root node to each node z_i in the graph. If the maximum flow is smaller than z_i^{LP}, we know from the Max-Flow Min-Cut theorem that the minimum cut, and therefore inequality (5.3.10) is violated. Again, the violated inequality is added to the LP, and the LP is resolved again. This procedure is repeated until no further inequalities are found. In the following, we describe two enhancements of the original cut generation described in [18].

With the original cut separation we get at most $|V^+| - 1$ cuts, but in most cases many of these cuts are the same. To generate a larger amount of different cuts in each separation step we also use *back-cuts* [45]. The idea of back-cuts is to flip the graph G to G'. That means we reverse the direction of each arc while preserving the arc capacity, i.e. (j, i) is in $A(G')$ if and only if (i, j) is in $A(G)$ and the capacities of (j, i) and (i, j) are the same. When the max-flow algorithm is performed on $A(G')$, a node from $|V^+| - r$ is used as the new source node and r is used as the sink. This is repeated for all $|V^+| - r$ nodes. If the capacity of the back-cut is less than the max-flow in G the corresponding cut in G is obtained by reversing all arcs in G' and added to the LP. For the separation of cuts we use Cherkassky and Goldberg's implementation of the push-relabel method for the maximum flow problem [12]. The implementation of Cherkassky and Goldberg is advantageous if cuts and back-cuts are to be computed, since the max-flow algorithm returns the minimum cut closest to the sink as well as the minimum cut closest to the source [34, 38]. A speedup is also achieved through Creep-Flow, introduced in [30]. The idea of Creep-Flow is to add a tiny capacity ϵ to each arc, to get not only a weight minimal cut but also an arc minimal cut. Although this increases the running time for computing a minimal cut, since more arcs must be considered, the number of cutting plane iterations is reduced significantly [30]. Using cycle elimination cuts and directed connection cuts together guarantees that the optimal solution is always found.

5.3.3 The Pricing Problem

Our objective for the k-MLSA problem is to minimize the number of labels, i.e. we search for a combination of labels that forms a feasible solution (i.e. the represented arcs can form a $k-$node arborescence) and is minimal. We price over the labels, since we we have a large number of labels and only a small number of labels is contained in every feasible solution.

We start with a near optimal, feasible solution generated by the *MA*, which consists of a small set of labels (template arcs), and add new labels that can potentially improve the solution. Since we deal with a minimization problem, every column with negative reduced costs can improve the solution and is therefore a candidate to enter the basis. The pricing problem is to find a column with negative reduced cost. We call the dual variables $\pi_{i,j}$ if they correspond to the arc-label constraints (5.3.2) and μ_j if they correspond to the node-label constraints (5.3.11). The reduced costs of a label (i.e. a template arc) are given by the following formula:

$$\bar{c}_t = 1 - \Big(\sum_{(i,j)\in A(t)} \pi_{i,j} + \sum_{j\in\{v|(u,v)\in A(t)\}} \mu_j \Big) \tag{5.3.12}$$

Consequently, we can define the pricing problem as:

Definition 8. (Pricing Problem)

$$t^* = \text{argmin}_{t\in T}\{\bar{c}_t\} \tag{5.3.13}$$

Whenever we find a template arc t with negative value $\overline{c}_t$, it may improve the solution and is therefore added to the model. We differentiate between a basic and an extended case of the pricing problem. In the basic case, we only use the arc-label constraints. In this case, μ_j is set to zero. The drawback of using only the arc-label constraints is that they cause the preference of labels that can represent many different arcs, even though the represented nodes may be the same. In the extended case additional node-label constraints (5.3.11) associating nodes with template arcs are used. Due to the node-label constraint, template arcs that represent poorly covered nodes are preferred.

5.3.4 Constructing all Candidate Template Arcs in Advance

For the construction of candidate template arcs (labels) we use two different approaches. The first approach is to construct all candidate template arcs in advance with *NIS* (as described before in Section 3.2) and select the best candidate template arc from this set in every pricing step. The second approach is to use *DNIS* (as described before in Section 3.3) to construct candidate template arcs just on demand.

The advantage of constructing all candidate template arcs in advance is that we can use a heuristic method to improve the solution, which is described in the following. Let $\overline{c}_{i,j} = \pi_{i,j} + \mu_j$ ($\overline{c}_a$) denote the costs of one arc, i.e the dual values assigned to it. Let $c(t)$ denote the value of a template arc t, i.e. the sum of the costs of the arcs represented by the template arc: $c(t) = \sum_{(i,j)\in A(t)} \overline{c}_{i,j}$. It is possible that some of the arcs represented by a template arc cannot be used together in a k-node arborescence, so the value $c(t)$ assigned to it by summing up its arc costs may be too high. Consequently, we have to decrease $c(t)$ by counting only the costs of the arcs that can be used together in a valid solution. Let t_s^* denote the template arc with the greatest value where only arcs that are usable in a valid solution are counted. Note, however, that we use this heuristics only for the selection of the best template arc from the set of template arcs with negative reduced costs. If after decreasing the value $c(t)$ of all template arcs with $c(t) > 1$ and no template arc with $c(t) > 1$ is left, we return the adjusted template arc with the greatest c_{ts^*} anyway. Two or more arcs cannot be used together in a valid solution in the following two cases:

1. The in-degree $d^-(v)$ of each node v, except for the root node, has to be exactly one. So, if a node is the target of more than one arc, only one of these arcs is usable in the k-node arborescence. Let $A(t, v)$ denote the set of ingoing arcs of the node v represented by template arc t. From this set, we count only the costs of one arc, namely the ingoing arc with the highest costs i.e. $\bigcup_{v\in V} \text{argmax}_{a\in A(t,v)}\{\overline{c}_a\}$. Figure 9 shows this situation. The arcs in the picture are represented by a template arc. Node 1 has in-degree $d^-(1) > 1$, so only one of its ingoing arcs is usable in a feasible arborescence.

2. If n of the m arcs represented by a template arc form a cycle, we can only use $n - 1$ of these arcs. Hence, from the n arcs contained in the cycle we do not count the costs of the arc with the lowest costs. Since it is not important to obtain the exact value of a template arc, we only consider cycles of length two, because they can be found very efficiently. Let $A(t, c)$ denote the set of two tree arcs that form a cycle and are represented by template arc t. From $A(t, c)$ we count only the arc with the higher costs i.e. $\bigcup_{a\in A} \text{argmax}_{a\in A(t,c)}\{\overline{c}_a\}$.

We first apply the rule defined in case 1 and count only the ingoing arc with the highest costs if a node has more ingoing arcs. If the reduced value $c(t)$ of the template arc is still greater than one we search

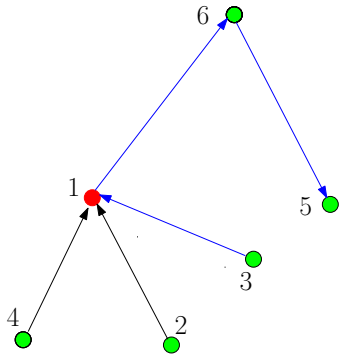

Figure 9: Arcs represented by a template arc t. Node 1 has in-degree $d^-(1) = 3$. The blue arcs belong to an arborescence.

for cycles. It is possible that the value of one or more of the arcs that form a cycle has already been subtracted, because one of the nodes v in the cycle may have in-degree $d^-(v) > 1$. In this case, the value $c(t)$ of the template arc remains the same. Figure 10 shows this situation. Node 1 is contained in a cycle and has also in-degree $d^-(1) > 1$.

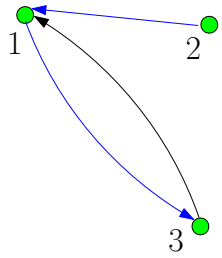

Figure 10: Arcs represented by a template arc t. The arcs form a cycle, and node 1 has in-degree $d^-(1) = 2$. The blue arcs belong to an arborescence.

If we find a cycle of length two, and there is a node with in-degree greater than one $d^-(v) > 1$ in the cycle, the value of one arc has already been subtracted in step one. Otherwise we subtract the value of the smaller arc. We must also consider the case where nodes with $d^-(v) > 1$ are involved in more than one cycle. Consider one node with $d^-(v) = 4$ which is involved in three cycles. In this case we would not have to subtract values for the cycles, because only the value of one of the incoming arcs is counted. To process this case we use an in-degree counter for each node v and set it to $d^-(v)$. Whenever the value of an arc has already been subtracted from $c(t)$, because there is a node with $d^-(v) > 1$ in the cycle, we decrease the in-degree counter of v by one. Only if the in-degree counter is greater than one, we do not have to subtract the value of one of the arcs in the cycle. This procedure is described by the following two algorithms.

The method, `getBestLabel` (Algorithm 11), returns the adjusted template arc with the highest value c_{ts^*} in terms of the values of the arcs $\overline{c}_{i,j}$ that can be used together in a feasible solution. The algorithm selects from the set of unused template arcs, i.e. T^u, the template arcs that can possibly have a value $c(t)$ greater than one.

Table 12: Symbols used in `getBestLabel` (Algorithm 11) and `checkCycleTwo` (Algorithm 12)

Symbol	Purpose
$T^u \subset T^c$	set of template arcs (labels) not in the RMP
$A(t)$	set of tree arcs a template arc can represent
$A^c(t)$	set of tree arcs a template arc can represent that form a cycle of size two
T^s	multimap of selected template arcs (insert O(log n))
b	array of target node values (initially all values are set to 0)
$c(t)$	value of template arc t
c_{ts^*}	value of optimal adjusted template arc
t_s^*	optimal adjusted template arc
$\overline{c}_{i,j}$	arc costs $\overline{c}_{i,j} = \pi_{i,j} + \mu_j$
$c_{\max}$	greatest arc costs assigned in a pricing step

Algorithm 11: `getBestLabel()`

```
1  b[i] ← 0, ∀1 ≤ i ≤ n
2  forall t ∈ T^u do
3      if (|A(t)| · c_max) > 1 then
4          forall a_{i,j} ∈ A(t) do
5              if b[j] < c̄_{i,j} then  //count only greatest target node
6                  b[j] ← c̄_{i,j}
7              end
8          end
9          c(t) ← Σ_{j=1}^{n} b[j]
10         set all elements from b to 0
11         if c(t) > 1 then
12             T^s[c(t)] ← t //template arcs are sorted in descending order
13         end
14     end
15 end
16 t_s^* ← checkCycleTwo(T^s)   //t_s^* = best adjusted label
17 if c_{ts^*} > 1 then
18     T^u ← T^u \ t_s^*
19     return t_s^*
20 end
```

A template arc can have a value greater than one only if $(|A(t)| \cdot c_{\max}) > 1$ (line 2). From each of these template arcs the sum of the values of the arcs (i, j) that have different target nodes j is computed (line 3-6). If this sum is greater than one (line 9), a template arc qualifies as a candidate label (line 10) and is checked for cycles of size two (line 14). Procedure `checkCycle2` (Algorithm 12) reduces the value of a template arc by the values of the arcs that form a cycle of size two and returns the template

arc with the greatest value if this value is still greater 1 (line 15, 17). The returned template arc is removed from the set of unused template arcs (line 16).

Algorithm 12: `checkCycleTwo`(T^s)

```
1  c_ts* ← 0
2  t*_s ← NULL
3  target[i] ← 0, ∀1 ≤ i ≤ n
4  forall t ∈ T^s do
5      if c_ts* ≥ c(t) then  //no greater value possible
6          return t*_s
7      end
8      A^c(t) ← ∅
9      forall a_i,j ∈ A(t) do
10         if marked[a_j,i] = true then
11             A^c(t) ← A^c(t) ∪ a_i,j   //collect arcs that form a cycle
12         else
13             marked[a_i,j] ← true
14         end
15         target[j] ← target[j] + 1   //count how often node j is target
16     end
17     forall a_i,j ∈ A^c(t) do
18         minValue ← 0
19         if target[j] > 1 then //c(t) already decreased
20             target[j] ← target[j] − 1
21         end
22         else if target[i] > 1 then  //c(t) already decreased
23             target[i] ← target[i] − 1
24         else
25             minValue ← min(c̄_i,j, c̄_j,i)
26             c(t) ← c(t) − minValue
27             if c(t) ≤ c_ts* then
28                 break
29             end
30         end
31     end
32     if c(t) > c_ts* then
33         c_ts* ← c(t)
34         t*_s ← t
35     end
36 end
37 return t*_s
```

Procedure `checkCycleTwo` (Algorithm 12) gets the set of candidate labels T^s, i.e. the labels with a value greater than 1, as input. Its purpose is to reduce the value of a label if a cycle is found as

described before. Note that the only purpose of this algorithm is to reduce the values of the labels given as input parameter and to return the best adjusted label t_s^*. The candidate labels are sorted by their value in descending order. We loop over these labels until a label with a value smaller than the greatest value of an adjusted label $c(t_s^*)$ known thus far is found. In this case, the algorithm returns the best label t_s^* it has found (lines 1 to 4) as no better label is possible. We know that no better label is possible because the candidate labels are sorted by their values in descending order. If a label has a value greater than the best found label, all arcs represented by the label are checked for cycles of size two. To do that, we iterate over each arc (i, j) of a label (lines 5 to 10) and check if its reverse arc (j, i) (line 6) is marked. If this is the case, we have found a cycle and insert it in the set of cycles (line7). Otherwise we mark the current arc (i, j) (line 9). We also count how often a node is a target node (line 11). Then we loop over all arcs that are members of the cycle set $A^c(t)$ (lines 13 to 27). If one of the nodes from the arcs that form a cycle of size two is also a target node more than once, we know that we have only counted one of the arcs in the cycle anyway and therefore do not have to reduce the value of the label. In this case, we only reduce the target node counter (lines 15-18). Otherwise we reduce the value of the label by the value of the smaller arc (lines 21 to 22). If the value of the label is not greater than the greatest value we stop (lines 23 to 25). If after the loop the adjusted label is greater than the greatest value found so far, we have found a new best label and store this value as the greatest value (lines 28 to 30).

5.3.4.1 Start Heuristic

At the beginning of the *BCP* process we have to determine a feasible start solution for the initial *RMP*. Since it is desirable to begin with a near optimal feasible solution, we use the memetic algorithm described in Chapter 4 to create the initial solution. The following Section 5.3.4.2 describes how a lower bound (*LB*) can be constructed for the problem. If the gap between LB and MA is 0%, the MA has found the optimal solution, and BCP algorithm does not have to be applied at all.

5.3.4.2 Determining a Tight Lower Bound

In a preprocessing step we try to construct a lower bound. What is the minimum number of template arcs required to build a feasible solution? We can always determine whether the number of template arcs has to be greater than or equal to three in the following way. First we reduce our problem, the search for a k-node arborescence, to the following simpler problem.

How many template arcs are required to cover the requested k nodes. To represent k nodes we need at least $k-1$ different target nodes, i.e. nodes with $d^-(v) > 0$, since only one node can be the root of the tree and therefore have $d^-(v) = 0$. Consequently considering only the target nodes gives a better lower bound. Note that previously we looked at the arcs a template arc can represent. In contrast to that, we now examine the target nodes a template arc can represent. Obviously, the lower bound has to be at least one, since we need one template arc to represent any nodes at all.

The second step is to look at the maximal number of target nodes a template arc t can represent. We denote this number as $t_{\max}$. The number of required template arcs is at least $\lceil \frac{k-1}{t_{\max}} \rceil$. If $\lceil \frac{k-1}{t_{\max}} \rceil$ is greater than two we set the lower bound to the computed value and stop here, because any further computation would be too expensive. Otherwise, in a third step, we examine all pairs of templates arcs, which contain at least $k-1$ target nodes. Since some of the nodes represented by a pair of template arcs may be duplicates, the next step is to eliminate the duplicates and check if a pair still

contains $k-1$ different target nodes. If a pair of template arcs contains $k-1$ different target nodes, we check if the arcs represented by these template arcs form a k-node arborescence. If no pair of template arcs contains a k-node arborescence the lower bound has to be at least three. Otherwise we have found the best solution. Note that the number of comparisons is reduced significantly by comparing only the pairs of template arcs that together cover at least $k-1$ different target nodes. However, the third step is only applicable if T^c is not too large. With this lower bound we can verify whether our primal heuristic, the memetic algorithm described before, has already found the optimal solution.

5.3.5 Constructing Candidate Template Arcs on Demand

In this approach we use *DNIST* to construct candidate template arcs on demand as described before in Chapter 3.3. Since we must not price the same candidate template arc twice, we simply mark the already used candidate template arcs tabu. Note that in this case we cannot use the improvement described in the presentation of *DNIST* (see Chapter 3.3) to store the value of a dominated template arc in the final recursion step as a new lower bound if the sum of its arc values is greater than lb^*, since we do not know if the dominating candidate template arc is marked tabu.

As a feasible initial solution we simply construct the candidate template arcs that can represent $k-1$ outgoing arcs from node zero. This can be done by setting the value of each arc $(0, j)$ where $j = 1, \ldots, k$ to one, while the value of all other arcs is set to zero. We call this type of initialization star initialization because of its star shape.

At the start of each pricing step we set lb^* to $1-\epsilon$ so we do not have to traverse branches with a value smaller than one, since only labels that have a value greater than one will be priced into the model.

5.3.6 Branching

Branching is performed if the LP solution is fractional in order to enforce the integrality condition. In the branching process two subproblems (child nodes) are created, by selecting one integral variable x_j with fractional value x_j and creating the two branches with subproblems $x_j \leq \lfloor \overline{x}_j \rfloor$ and $x_j \geq \lceil \overline{x}_j \rceil$ respectively. For a detailed description see [1].

5.4 Results

The tests were performed on a Dual Core AMD Opteron 270, 1.9 GHz with 8GB RAM. Libraries, programming language, framework and compiler version are the same as for the tests presented in Chapter 3 and 4. We used SCIP 1.2.0 (Solving Constraint Integer Programs) as a Branch-and-Cut-and-Price framework [2] with CPLEX version 11.2 [27] as the underlying LP-Solver.

We tested six variants of the branch-and-cut-and-price algorithm, which are listed in Table 13. The variants where all template arcs T^c are constructed in advance with *NIS* have the prefix *BCP*, while the variant where the template arcs are constructed on demand with *DNIS* is called *Dynamic BCP (DBCP)*. In *DBCP* we always use the arc-label constraints and the node-label constraints.

The basic version, which uses only the arc-label constraints, is called *BCP*. The variant which uses the additional node-label constraint is called *BCP/NL*. The basic version with the additional improvement to count only arcs that are usable in a valid solution is labeled *BCP/UA*. If we use the additional node-label constraint in this variant, it called *BCP/NL/UA*. The variant of *BCP* that does

not use back-cuts is denoted as BCP/NB. Using directed connection cuts and cycle elimination cuts lead to the best results. So we used this configuration for all tests. We also used creep-flow for all tests. If a variant uses the *MA* as an initial heuristics, it has the postfix *MA*. If a lower bound is computed, the version has the postfix *LB*.

Lower bound computation (*LB*): Section 5.3.4.2 describes how a lower bound for the *k-MLSA* problem can be computed. For our particular data sets we can always decide if the lower bound has to be at least three, since for the *Fraunhofer Templates* $|T^c|$ is small enough to compare all pairs of template arcs, and for the *NIST* data, which have a larger $|T^c|$, we always need at least three template arcs to represent all requested target nodes. With this lower bound we can verify whether our primal heuristic, the *MA*, has already found the optimal solution. For the *Fraunhofer Templates*, the gap between the lower bound and the result of the *MA* is 0% wherever the number of required template arcs does not exceed three, because for these instances the *MA* always found the optimal solution.

In the tables of this chapter that show the running time for single instances we use the following column headers. The first four columns show the instance name *inst*, the number of nodes $|V|$, the number of candidate template arcs $|T^c|$ and the running time for the candidate template arc construction $tt[s]$. The column *m* shows the number of template arcs (labels) contained in the optimal solution. The three columns *col*, *cut* and *nod* show the number of priced variables (*col*), the number of added cuts (*cuts*) and the number of nodes (*nod*), generated in the branch-and-cut-and-price process. Column $t[s]$ shows the running time of the branch-and-cut-and-price algorithm. The column $t_p[s]$ shows the running time of the preprocessing step, i.e. the running time of the *MA* and the time for computing the lower bound (*LB*).

Table 14 shows the results of different *BCP* versions for the *Fraunhofer Templates* with $\vec{\delta} = (40, 40, 40)^T$ and $k = 30$. Table 15 shows the results of the dynamic generation *DBCP* of template arcs for the *Fraunhofer Templates* and the same parameter setting. Since the *Fraunhofer Templates* are relatively small, the best solution is often found at the start as the lower bound equals the solution found by the *MA*. If the gap between *LB* and *MA* is 0% we know that our *MA* has found the optimal solution, and we do not need to run the BCP algorithm at all. For this reason only one column ($BCP/NL/UA + MA + LB$) is shown where the instances are initialized by the *MA* and the lower bound (*LB*) is computed. In the same way we can compare how the other variants of *BCP* work for the *Fraunhofer Templates*. The sign $\emptyset$ means that the *MA* computed a solution equal to the lower bound and no *BCP* is performed. The variant $BCP/NL/UA + MA + LB$ clearly outperforms the other variants since it can solve many instances immediately. The performance of the different *BCP* variants strongly depends on the particular instances. There is also the strong suspicion that for these small datasets the performance depends heavily on the random initialization. Thus, the small single instances from the *Fraunhofer Templates* do not support a conclusive judgment on the quality of the different methods, except for the evident superiority of $BCP/NL/UA + MA + LB$. The computation of average values over all instances and especially over the larger instances from the $NIST$ dataset, will allow a better analysis of the different *BCP* variants. In all other test cases throughout this section we always computed the lower bound and used the *MA* as an initial heuristic. An exception to this rule is of course the construction of template arcs on demand in variant *DBCP*.

As seen before, the *MA* found the optimal solution for the Fraunhofer data in all all cases. Also, for the *NIST* data the optimal or a near optimal solution is found in many cases. So it is obvious that for most *Fraunhofer Templates* the *MA* in combination with the lower bound computation (*LB*) is the best method. Furthermore for many combinations of $\vec{\delta}$ and k values, the solution is found instantaneously

Table 13: Different variants of *BCP* and *BP*.

Symbol	Purpose
BCP	basic version with *arc-label* constraint (see 5.3.11)
BCP/UA	*BCP* and count only arcs that are usable (*UA*) in a valid solution (see 5.3.4)
BCP/NL	*BCP* with additional *node-label* constraint (see 5.3.2)
BCP/NL/UA	*BCP* + *NL* and count only arcs that are usable in a valid solution
BCP/NB	*BCP* without back-cuts
DBCP	dynamic *BCP* with both constraints
BP	branch-and-price approach

by applying *MA* and *LB*. In all tables of this chapter that show average values of a set of instances, the additional column $t_{SD}[s]$ shows the standard deviation of the running time. Unsolved instances are indicated by *ns*.

Table 16 shows the average values of all *BCP* variants and the *BP* approach, developed by Thöni [46], for all *Fraunhofer Templates*. The *DBCP* variant does not generate the template arcs at the beginning, so the template arc generation time $tt[s]$ as well as $t_p[s]$ (the running time of the *MA* and the *LB* computation time) does not apply for this variant. Missing values are generally indicated by a '-' character. As initialization for this variant the star initialization described before (see Section 5.3.5) was used, and a lower bound computation is not possible. The *MA* performed 10000 iterations and the population size was 200. Column $t_{\text{best}}[s]$ shows the time in which the best solution was found. The best result for each configuration is marked bold. Again, we can see a significant reduction of the running time, if the *MA* is the initial heuristics and a lower bound is computed. For a comparison with the *BP* method, we use variant *DBCP*, since both variants generate template arcs on demand. Both variants also use the same star initialization. In three of four cases *DBCP* has a significantly lower running time than *BP*. The reason for this is that in variant *DBCP*, fewer variables are priced in, due to the cuts. Here we can see that the dominant reason for high running times is a large number of columns rather than a large number of cuts or nodes. This shows especially for the configurations $\vec{\delta} = (40, 40, 40)^T$ and $k = 30$, where the running time of *BP* is almost three times as high as the running time of *DBCP*, although the node number of *DBCP* is twice as high and cuts are used, but *DBCP* has only half the number of columns. The only instance where *BP* performed better than *DBCP* was $\vec{\delta} = (30, 30, 30)^T$ and $k = 30$. For this configuration the number of columns is nearly the same for both methods but the number of nodes is ten times higher for *DBCP*. The low average running time of *DNIS* regarding $\vec{\delta} = (45, 45, 45)^T$ is due to its higher performance in solving instance ft-01 which is an extreme outlier. We can see that ft-01 is an outlier for all methods except *DNIS* on the much lower standard deviation of *DNIS*. We present instance ft-01 in Table 17. Note that although the running time strongly depends on the number of nodes and template arcs, some instances are much more difficult to solve than others. E.g. instance ft-18 is more difficult to solve than instance ft-11, even though it contains the same number of nodes and fewer template arcs. Table 18 presents a further comparison of variant *DBCP* with the *BP* approach, developed by Thöni [46], for all *Fraunhofer Templates*. For $\vec{\delta}$ values greater or equal than 40 only one template arc is requested if $k = 10$ or $k = 20$. For $k = 30$ never more than two template arcs are requested, so we do not represent these cases. Also for $k = 10$

Table 14: Running times of different *BCP* variants for the *Fraunhofer Templates* with $\vec{\delta} = (40, 40, 40)^T$ and $k = 30$.

					BCP				BCP/NL				BCP/NL/UA			
inst.	$\|V\|$	$\|T^c\|$	$tt[s]$	m	*col*	*cut*	*nod*	$t[s]$	*col*	*cut*	*nod*	$t[s]$	*col*	*cut*	*nod*	$t[s]$
ft-01	30	3897	0.24	4	818	2248	788	81.11	792	293	840	88.61	165	169	162	10.83
ft-02	28	3353	0.21	4	1212	62	2116	133.95	1059	65	1579	104.58	230	27	209	10.7
ft-03	30	6464	0.56	3	614	1883	227	69.05	220	171	94	43.12	199	86	393	21.93
ft-04	20	1223	0.06	3	181	203	85	6.81	157	78	63	4.99	25	37	7	0.37
ft-05	30	8669	0.80	3	2648	8315	1717	435.64	649	250	297	140.13	412	832	670	54.12
ft-06	15	439	0.02	3	142	31	42	2.02	86	58	33	1.32	26	14	4	0.15
ft-07	28	2237	0.09	4	432	106	175	27.29	344	165	173	24.07	123	44	46	5.0
ft-08	27	2164	0.10	4	521	398	293	33.87	381	126	185	24.67	283	892	230	9.54
ft-09	27	2091	0.10	4	594	190	657	47.81	474	237	521	35.67	157	30	157	5.73
ft-10	30	2930	0.14	4	593	172	681	81.20	860	1085	738	87.97	207	29	139	13.41
ft-11	30	11497	0.94	3	3439	1673	10831	2141.04	3492	3937	14761	2808.65	1039	1760	5205	377.41
ft-12	28	4224	0.28	3	497	280	123	29.65	653	532	277	32.00	111	377	49	3.31
ft-13	25	2573	0.16	3	1355	249	2715	97.14	1346	512	1729	90.82	653	228	1652	30.17
ft-14	30	5150	0.30	3	1408	1835	1434	175.34	1121	1601	891	131.48	1262	1154	5243	193.59
ft-15	29	3980	0.26	3	1081	604	662	71.82	642	285	293	42.08	89	244	28	5.14
ft-16	30	6095	0.36	3	884	263	1590	270.17	930	1158	1947	329.68	282	692	599	51.27
ft-17	30	3847	0.19	3	849	1082	566	84.03	610	461	159	62.38	400	96	187	18.86
ft-18	30	8889	0.53	3	4584	129460	33717	5205.28	3420	3781	20756	4065.17	1645	265	16240	1452.02
ft-19	30	4538	0.21	3	1306	6998	1093	141.02	838	2610	1059	127.41	360	754	250	23.87
ft-20	28	2021	0.09	4	629	268	539	48.49	466	134	396	33.44	88	436	119	8.07

					BCP/NB				BCP/UA				BCP/NL/UA + LB + MA				
inst.	$\|V\|$	$\|T^c\|$	$tt[s]$	m	*col*	*cut*	*nod*	$t[s]$	*col*	*cut*	*nod*	$t[s]$	*col*	*cut*	*nod*	$t[s]$	$t_p[s]$
ft-01	30	3897	0.17	4	728	439	725	47.67	183	112	113	12.23	108	0	51	5.43	1.26
ft-02	28	3353	0.14	4	1261	280	2653	96.21	224	27	145	11.92	168	27	109	7.46	1.03
ft-03	30	6464	0.37	3	1289	615	1446	105.36	334	563	208	20.52	∅	∅	∅	∅	1.96
ft-04	20	1223	0.03	3	184	103	85	3.30	31	75	19	0.7	∅	∅	∅	∅	0.80
ft-05	30	8669	0.53	3	1339	2046	707	107.47	1387	3656	2329	231.73	∅	∅	∅	∅	2.38
ft-06	15	439	0.01	3	66	41	13	0.56	32	0	3	0.19	∅	∅	∅	∅	0.41
ft-07	28	2237	0.07	4	367	97	125	14.61	154	11	61	6.41	105	0	37	2.31	0.96
ft-08	27	2164	0.07	4	486	250	273	20.31	189	169	88	7.23	157	26	71	3.98	0.87
ft-09	27	2091	0.06	4	635	49	676	29.77	158	198	135	8.71	145	0	121	5.46	0.95
ft-10	30	2930	0.11	4	629	230	544	39.97	360	306	314	24.06	215	57	137	12.03	1.01
ft-11	30	11497	0.65	3	3283	224	10349	1101.99	1187	222	9073	634.72	∅	∅	∅	∅	2.02
ft-12	28	4224	0.19	3	136	34	19	4.82	90	76	26	3.48	∅	∅	∅	∅	0.93
ft-13	25	2573	0.11	3	993	1397	1167	34.55	188	79	122	7.75	∅	∅	∅	∅	0.89
ft-14	30	5150	0.22	3	315	43	117	26.67	390	162	645	36.27	∅	∅	∅	∅	1.35
ft-15	29	3980	0.16	3	137	71	27	8.94	46	168	54	9.75	557	2055	2624	42.3	0.9
ft-16	30	6095	0.23	3	1028	582	1674	185.55	543	1371	1652	121.6	∅	∅	∅	∅	2.06
ft-17	30	3847	0.12	3	904	55	605	51.20	173	58	46	8.01	173	58	46	8.01	1.04
ft-18	30	8889	0.38	3	3522	1272	20695	2367.16	1948	4619	15767	1591.69	∅	∅	∅	∅	2.47
ft-19	30	4538	0.14	3	166	219	47	18.09	1273	11728	2174	124.68	∅	∅	∅	∅	1.74
ft-20	28	2021	0.07	4	572	144	433	29.32	206	157	83	10.16	114	66	31	3.32	0.92

Table 15: Running times of *DBCP* for the *Fraunhofer Templates* with $\vec{\delta} = (40, 40, 40)^T$ and $k = 30$.

inst.	$\|V\|$	m	*col*	*cut*	*nod*	$t[s]$	**inst.**	$\|V\|$	m	*col*	*cut*	*nod*	$t[s]$
ft-01	30	4	634	196	612	32.83	ft-11	30	3	2998	822	9837	1050.35
ft-02	28	4	1112	273	1703	57.62	ft-12	28	3	296	217	54	14.16
ft-03	30	3	1233	1471	933	60.23	ft-13	25	3	1446	1022	2937	63.38
ft-04	20	3	151	186	103	3.03	ft-14	30	3	80	67	9	5.87
ft-05	30	3	411	111	148	51.07	ft-15	29	3	1283	361	1269	45.49
ft-06	15	3	104	50	48	0.80	ft-16	30	3	1147	393	1807	179.03
ft-07	28	4	354	3	176	11.00	ft-17	30	3	887	60	548	42.43
ft-08	27	4	363	35	185	13.60	ft-18	30	3	2953	545	15481	1706.89
ft-09	27	4	475	145	490	19.69	ft-19	30	3	965	2459	819	55.99
ft-10	30	4	536	40	623	25.38	ft-20	28	4	456	167	341	17.46

Table 16: Average runtimes of *BCP* variant 1 – 6 and *BP* for the *Fraunhofer Templates*. Missing values are indicated by a '-'.

k		20							30						
method	$\vec{\delta}$	*col*	*cut*	*nod*	$t_{best}[s]$	$t[s]$	$t_{SD}[s]$	$t_p[s]$	*col*	*cut*	*nod*	$t_{best}[s]$	$t[s]$	$t_{SD}[s]$	$t_p[s]$
BCP	$\binom{30}{30}$	118.90	35.85	202.60	6.65	9.20	17.37	1.78	408.05	297.95	2558.15	34.59	89.61	141.36	1.43
BCP/NL		105.90	57.70	424.70	10.18	12.64	38.61	1.77	395.65	356.20	2500.85	20.98	71.74	122.13	1.42
BCP/UA		57.15	12.80	152.95	4.77	6.15	14.31	1.76	318.94	335.11	2396.00	44.85	86.98	156.20	1.50
BCP/NL/UA		31.25	15.40	25.95	0.98	**2.53**	4.35	1.76	250.56	156.06	1558.00	26.37	**57.51**	101.64	1.51
DBCP		557.35	443.55	1630.95	34.56	146.60	259.56	-	470.45	537.10	2752.45	63.70	115.39	226.11	-
BCP/NB		136.85	39.05	528.50	18.66	21.39	57.95	1.80	400.10	370.90	2411.60	34.14	94.40	153.36	1.44
BP		772.00	-	540.00	-	288.40	-	-	477.00	-	219.00	-	98.90	-	-
BCP	$\binom{40}{40}$	9.00	0.00	0.50	0.00	0.11	0.13	0.63	333.15	98.80	544.90	8.98	21.22	33.62	1.35
BCP/NL		9.80	0.00	0.50	0.00	0.10	0.14	0.63	293.95	28.85	379.85	3.73	13.23	16.34	1.35
BCP/UA		0.00	0.00	0.00	0.00	**0.00**	0.00	0.63	148.22	75.33	201.39	6.35	**10.01**	14.51	1.42
BCP/NL/UA		0.00	0.00	0.00	0.00	**0.00**	0.00	0.63	87.10	114.45	161.35	2.52	4.52	9.32	1.33
DBCP		773.75	310.80	4143.80	37.18	168.33	250.00	-	894.20	431.15	1906.15	22.35	172.82	416.83	-
BCP/NB		9.00	0.00	0.50	0.00	0.12	0.14	0.64	374.00	167.55	908.70	23.42	37.34	90.75	1.38
BP		982.00	-	444.00	-	278.70	-	-	1855.00	-	927.00	-	431.70	-	-
BCP	$\binom{45}{45}$	1.75	0.00	0.10	0.00	0.01	0.02	0.07	476.90	1596.10	15397.65	12.62	384.98	1618.75	1.04
BCP/NL		2.05	0.00	0.10	0.00	0.00	0.01	0.07	506.40	1587.65	15504.45	29.98	357.44	1422.68	1.05
BCP/UA		0.00	0.00	0.00	0.00	0.00	0.00	0.07	292.94	1345.11	15772.06	2.00	371.48	1522.92	1.08
BCP/NL/UA		0.00	0.00	0.00	0.00	0.00	0.00	0.07	264.78	1796.83	14330.50	0.95	324.04	1331.92	1.09
DBCP		596.50	202.85	467.75	25.20	58.84	82.69	-	1424.10	702.20	3549.65	87.44	**255.58**	492.41	-
BCP/NB		1.75	0.00	0.10	0.00	0.01	0.02	0.07	478.25	1686.40	14603.65	14.85	450.54	1894.15	1.07

Table 17: Solution time of instance ft-01 for *BCP* variant 1 – 6 with $\vec{\delta} = (45, 45, 45)^T$ and $k = 30$

inst	**method**	*col*	*cut*	*nod*	$t_{best}[s]$	$t[s]$	$t_p[s]$
ft-01	*BCP*	5382	30647	295967	0.00	7439.47	1.36
ft-01	*BCP/NL*	5173	24355	270641	0.00	6544.30	1.36
ft-01	*BCP/UA*	4468	22583	282981	0.00	6650.60	1.67
ft-01	*BCP/NL/UA*	4384	30992	257519	0.00	5815.69	1.66
ft-01	*DBCP*	3923	5233	42483	8.89	**1471.19**	-
ft-01	*BCP/NB*	5409	32453	280087	0.00	8705.08	1.41

and $\vec{\delta} = (30, 30)^T$ only one template arc is needed, we indicate this trivial solution by a 't' character. The branch-and-cut-and-price algorithm performs better in almost all cases, but its main quality is that it can solve the larger *nist* instances.

For the tests of the *NIST* data we have combined two groups of data, mainly for computing the averages. In the first group, called *nist*, each instance contains between 70 and 120 minutiae and comprises 15 instances, five from each of the categories good, bad and ugly (see Section 2.2). This group was used for the tests in the previous chapters. In the second group $nist_{small}$, each instance contains between 53 and 93 minutiae. This group comprises 9 instances from each category: nist-u-04-t to nist-u-12-t, nist-b-04-t to nist-b-12-t and nist-g-05-t to nist-g-13-t (see Section 2.2). The following tables show the average values of the different *BCP* variants over instances from $nist_{small}$ and *nist* as well as values for individual instances. With regard to the test instances, the determining factors for the running time were the number of nodes V and the number of template arcs T^c, but also the structure of the individual instances. With regard to the parameter settings, the determining factors for the running time were the size of the correction vector $\vec{\delta}$ and the size of k. Note that those

Table 18: Comparison of *DBCP* and *BP*. If the solution is trivial i.e. consists of only one template arc this is indicated by a 't' character.

k		10				20				30			
method	$\vec{\delta}$	*col*	*cut*	*nod*	$t[s]$	*col*	*cut*	*nod*	$t[s]$	*col*	*cut*	*nod*	$t[s]$
DBCP	$\binom{10}{10}$	391	16	1221	129.3	217	81	517	49.1	167	64	230	17.7
BP	$\binom{10}{10}$	508	-	1240	407.6	235	-	384	115.3	143	-	83	38.0
DBCP	$\binom{20}{20}$	560	1	1634	183.4	281	33	1476	92.8	308	49	376	31.4
BP	$\binom{20}{20}$	989	-	1669	487.5	368	-	401	90.6	296.0	-	94	42.0
DBCP	$\binom{30}{30}$	t	t	t	t	634	23	1268	109.9	547	31	571	39.2
BP	$\binom{30}{30}$	t	t	t	t	767	-	142	183.0	177	-	170	15.9
DBCP	$\binom{10}{10}$	111	17	120	12.4	92	23	85	7.82	62	42	50	3.2
BP	$\binom{10}{10}$	114	-	88	31.5	107	-	84	31.3	45	-	66	10.5

two are related, since a greater size of the correction vector $\vec{\delta}$ leads to a higher number of template arcs. The two groups were tested with different combinations of parameter settings for $\vec{\delta}$ and k. The instances from group $nist_{small}$ were tested with different three- and two-dimensional correction vectors $\vec{\delta}$ combined with different k values. The instances from group *nist* were more difficult to solve, and were therefore only tested with different three-dimensional correction vectors combined with different k values. The time limit for the computation of the average values is always four hours. The average values are computed only over the instances that were solved by all *BCP* variants within the time limit. The instances that could not be solved by one of the *BCP* variants, and are therefore not included in the computation of the average values, nn separate tables. Since the number of instances a particular method could not solve is probably the most important information, all tables that list average values have an additional column *ns* where the number of instances that could not be solved by that particular *BCP* variant is listed. The results of *BCP* variant $1-6$ for the instances that could be solved within 4 hours are presented in Table 19 and Table 20. The correction vector domain is $\vec{\delta} = (50, 50, 50)^T$ and $k = |V|$. We see that different methods solve different instances within the time limit. The performance of the methods differs greatly between the various instances. Instances with more than 7000 template arcs (nist-b-05 and nist-g-12) can no longer be solved by all methods. The overall best methods are $BCP/NL/UA$ and BCP/NL. Method *DBCP* also performed good for this instances.

The average values over instances from $nist_{small}$ that where solved within a time limit of 4 hours, using *BCP* variant $1-6$ and 3 dimensional $\vec{\delta}$ values, are presented in Table 22. The best solution for each configuration is highlighted with a bold typeface. Since no preprocessing step (i.e. *MA* and *LB* computation) is required for *DNIS* the column $t_p[s]$ is set to '-'. Note that due to the exclusion of instances that could not be solved by any of the methods, the average values are not sufficient for the evaluation of a particular method. The number of solved instances and the performance regarding a particular instance have to be considered as well. The variant $BCP/NL/UA$ solved most instances and performed best in terms of solution time. Using back-cuts was mostly beneficial. A parameter configuration with $k = 40$ is much more difficult to solve. This can be seen by comparing $k = 40$ and $k = |V|$ with $\vec{\delta} = (50, 50, 50)^T$. The number of unsolved instances at least doubles if $k = 40$. For the parameter configuration $\tilde{\delta} = (50, 50, 50)^T$, around 1000 variables were priced into the model. The

Table 19: Running times of *BCP* variant 1 – 4 for the *NIST* data from group $nist_{small}$ with $\vec{\delta} = (50, 50, 50)^T$ and $k = |V|$. Unsolved instances are indicated by *ns*.

					BCP						BCP/NL					
inst.	$\|V\|$	$\|T^c\|$	$tt[s]$	m	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_p[s]$	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_p[s]$
nist-b-04-t	81	5743	0.11	17	1289	352	1630	1292.08	1292.08	8.42	1322	7289	2250	1157.67	1157.67	8.54
nist-b-06-t	77	5366	0.08	16	1242	3174	1124	361.22	757.64	6.96	1069	2464	1505	574.43	857.69	6.88
nist-b-07-t	82	6232	0.10	16	2221	2249	12613	8066.70	8066.71	7.88	1606	2942	6839	3360.03	3360.03	7.88
nist-b-08-t	76	5852	0.10	14	1866	2329	3328	1815.47	1874.76	7.23	1649	1590	5153	1935.19	1954.88	7.17
nist-b-09-t	76	6303	0.12	13	1325	3360	937	597.50	713.64	6.39	2394	22092	19407	6153.10	6153.11	6.37
nist-b-10-t	63	3271	0.06	15	856	3783	1711	165.87	487.30	5.47	608	627	1394	77.16	278.67	5.52
nist-b-11-t	80	5912	0.10	15	1300	13907	1202	845.05	928.18	7.72	2226	4826	30310	13372.00	13372.98	7.78
nist-g-05-t	80	5526	0.09	17	2223	10996	27479	1346.61	12317.08	9.23	1663	3814	28801	252.73	10199.48	9.37
nist-g-07-t	74	5038	0.10	14	1250	9887	4341	308.91	2021.95	6.85	1258	5034	5540	530.87	1707.15	6.81
nist-g-08-t	82	6815	0.11	16	1821	4707	5317	257.46	3567.00	7.53	1705	5293	6675	878.11	3332.21	7.39
nist-g-09-t	53	2315	0.05	11	379	647	179	45.2	51.97	3.65	430	1431	375	51.73	55.08	3.42
nist-g-10-t	76	5333	0.09	16	1776	17382	5581	736.14	2684.14	7.32	1391	4822	5816	719.87	2341.21	7.65
nist-g-11-t	67	3966	0.07	14	841	252	902	219.05	387.98	5.80	738	2889	1138	91.86	332.72	5.77
nist-g-12-t	84	7474	0.14	15	ns	ns	ns	ns	ns	ns	2017	1800	22789	402.00	12257.31	7.41
nist-g-13-t	55	2207	0.03	15	387	1733	569	48.09	102.38	4.35	332	924	496	28.08	61.51	4.30
nist-u-04-t	84	6114	0.10	17	1133	941	2003	256.24	1838.54	8.98	1148	4564	2206	295.06	1373.64	8.85
nist-u-05-t	73	4460	0.06	16	1691	4729	17835	908.48	5479.13	6.58	1184	3318	16642	282.01	4837.68	6.70
nist-u-06-t	70	4086	0.06	15	818	858	323	236.71	259.41	6.02	720	2709	524	152.40	168.84	6.23
nist-u-07-t	63	3342	0.05	14	656	1266	310	140.47	140.48	5.26	384	204	121	72.29	72.30	5.30
nist-u-08-t	74	4694	0.08	16	1448	3622	1704	710.43	748.09	6.55	1256	19613	2446	690.39	700.08	6.57
nist-u-09-t	69	3822	0.06	15	903	1871	907	340.54	340.54	5.88	706	1136	796	216.67	216.67	5.85
nist-u-10-t	76	5390	0.10	14	1012	677	698	566.91	578.33	7.45	857	3724	641	364.90	377.48	7.47
nist-u-11-t	63	3265	0.06	14	1008	2172	902	309.33	319.19	4.83	650	1377	731	148.77	173.02	4.87
nist-u-12-t	80	6702	0.14	15	2156	3395	13875	416.35	8349.83	6.68	1592	79	16994	116.71	8078.58	6.65

					BCP/UA						BCP/NL/UA					
inst.	$\|V\|$	$\|T^c\|$	$tt[s]$	m	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_p[s]$	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_p[s]$
nist-b-04-t	81	5743	0.1	17	1611	2565	2603	1718.02	1718.02	8.09	795	1518	421	296.6	299.05	8.26
nist-b-05-t	93	9355	0.15	16	ns	ns	ns	ns	ns	ns	2440	3529	11867	5611.09	8575.54	9.41
nist-b-06-t	77	5366	0.09	16	1385	7672	1342	268.39	711.21	6.58	801	730	984	110.17	539.34	6.73
nist-b-07-t	82	6232	0.1	16	2480	6587	27589	12906.7	12906.68	7.51	1800	1411	20174	7198.73	7198.73	7.62
nist-b-08-t	76	5852	0.08	14	1718	2164	3707	1769.6	1842.25	6.85	1132	2536	1566	607.54	697.61	6.93
nist-b-09-t	76	6303	0.11	13	2580	5804	15476	6656.97	6656.98	6.09	1925	9426	7347	3148.72	3148.73	6.23
nist-b-10-t	63	3271	0.06	15	704	501	1228	72.54	357.94	5.29	639	1129	1562	98.28	343.37	5.40
nist-b-11-t	80	5912	0.11	15	1682	3472	2324	1681.08	1730.5	7.35	1398	4765	2883	1636.22	1683.06	7.48
nist-g-05-t	80	5526	0.09	17	2099	2776	26524	234.31	12545.29	8.9	1713	2180	32789	148.06	11041.25	9.16
nist-g-07-t	74	5038	0.07	14	1412	2113	3519	724.37	1808.15	6.57	1082	917	3840	169.82	1339.38	6.62
nist-g-08-t	82	6815	0.12	16	1999	1705	6375	903.86	3920.13	7.24	1530	798	7889	388.41	4011.42	7.26
nist-g-09-t	53	2315	0.03	11	194	56	47	6.27	36.85	3.61	233	456	89	22.47	28.68	3.58
nist-g-10-t	76	5333	0.1	16	1265	1033	4416	229.56	2181.62	6.88	1037	2622	4621	146.69	1848.35	7.11
nist-g-11-t	67	3966	0.06	14	1192	5420	1708	307.68	527.67	5.36	746	1005	1259	151.75	324.05	5.44
nist-g-13-t	55	2207	0.04	15	295	175	232	24.42	73.33	4.33	282	298	252	17.49	47.75	4.20
nist-u-04-t	84	6114	0.11	17	1744	27783	2955	863.32	2015.4	8.52	1574	24077	3607	907.7	1805.58	8.58
nist-u-05-t	73	4460	0.07	16	1362	699	9832	186.39	3717.19	6.24	1039	204	11148	84.6	3173.91	6.33
nist-u-06-t	70	4086	0.06	15	698	2524	289	176.19	204.28	5.92	725	2975	458	149.16	164.51	5.81
nist-u-07-t	63	3342	0.04	14	559	376	259	138.68	138.68	5.10	590	3116	629	141.61	141.61	5.08
nist-u-08-t	74	4694	0.08	16	1474	9263	1793	976.82	993.93	6.16	898	4004	918	331.27	374.08	6.26
nist-u-09-t	69	3822	0.05	15	620	5217	333	126.24	126.4	5.37	765	4639	754	177.27	177.27	5.53
nist-u-10-t	76	5390	0.1	14	1073	2291	371	379.58	392.29	7.17	459	343	95	149.5	164.63	7.06
nist-u-11-t	63	3265	0.03	14	712	3464	353	139.0	163.05	4.42	740	5359	582	125.75	134.59	4.63
nist-u-12-t	80	6702	0.11	15	2164	819	12909	367.54	7819.09	6.42	1961	20078	15375	549.83	6280.11	6.39

Table 20: Running times of *BCP* variant 5 and 6 for the *NIST* data from group $nist_{small}$ with $\vec{\delta} = (50, 50, 50)^T$ and $k = |V|$.

					BCP/NB						DBCP				
inst.	$\|V\|$	$\|T^c\|$	$tt[s]$	m	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_p[s]$	col	cut	nod	$t_{best}[s]$	$t[s]$
nist-b-04-t	81	5743	0.11	17	1543	1916	2716	2075.22	2075.23	8.56	1861	6729	20558	7991.03	7991.03
nist-b-06-t	77	5366	0.10	16	1276	4835	1056	340.46	673.59	6.84	1218	3575	1768	565.58	876.04
nist-b-07-t	82	6232	0.09	16	2146	1866	9757	6079.25	6079.25	7.71	1873	2488	9582	4814.69	4814.69
nist-b-08-t	76	5852	0.10	14	2245	9194	7364	3214.83	3229.49	7.02	1562	7106	3210	1252.32	1331.22
nist-b-09-t	76	6303	0.11	13	2479	27143	7008	2790.82	2832.24	6.29	2442	59866	12183	3360.08	3360.08
nist-b-10-t	63	3271	0.05	15	1101	6962	2376	321.63	576.43	5.53	791	4332	2396	79.48	461.86
nist-b-11-t	80	5912	0.11	15	1700	8632	2217	1744.26	1769.06	7.72	1701	25063	3302	1585.57	1615.42
nist-g-05-t	80	5526	0.09	17	2126	1671	27239	434.24	12871.36	9.29	1766	6867	32248	219.36	12537.63
nist-g-07-t	74	5038	0.08	14	1345	9916	4411	384.49	1963.73	6.73	1187	12659	3382	248.94	1166.82
nist-g-08-t	82	6815	0.10	16	1896	4935	6120	362.84	3992.23	7.38	1710	6613	5917	477.85	3749.41
nist-g-09-t	53	2315	0.02	11	379	639	179	45.31	52.10	3.63	598	13682	908	87.58	90.63
nist-g-10-t	76	5333	0.07	16	1637	13174	5671	516.00	2536.25	7.29	1303	6981	4687	433.47	1843.29
nist-g-11-t	67	3966	0.06	14	841	252	902	213.27	378.38	5.71	643	243	989	93.79	297.91
nist-g-12-t	84	7474	0.14	15	ns	ns	ns	ns	ns	ns	2398	8059	22574	1213.87	13546.30
nist-g-13-t	55	2207	0.03	15	379	835	669	32.65	110.29	4.43	401	976	560	24.52	74.25
nist-u-04-t	84	6114	0.12	17	1164	920	1917	245.85	1699.12	8.85	1493	8900	4148	1079.95	2680.98
nist-u-05-t	73	4460	0.09	16	1794	7171	21975	1159.07	6216.73	6.64	1410	9040	19404	656.50	5894.67
nist-u-06-t	70	4086	0.07	15	936	2969	532	274.26	303.27	6.04	427	328	115	64.77	104.87
nist-u-07-t	63	3342	0.04	14	565	417	153	120.04	120.04	5.26	398	220	168	92.04	92.04
nist-u-08-t	74	4694	0.07	16	1632	5377	3227	1269.49	1286.78	6.56	1180	2917	1371	459.02	489.72
nist-u-09-t	69	3822	0.06	15	676	890	409	175.00	176.31	5.87	748	484	693	226.54	226.54
nist-u-10-t	76	5390	0.08	14	1012	670	698	542.62	553.57	7.37	1099	1515	682	556.59	561.54
nist-u-11-t	63	3265	0.05	14	790	1439	460	178.61	194.96	4.79	781	12994	751	145.10	162.03
nist-u-12-t	80	6702	0.10	15	2434	11676	16519	1043.42	10102.57	6.58	1996	11595	14537	304.60	6233.43

Table 21: Unsolved instances of $nist_{small}$ for 3 dimensional $\vec{\delta}$. These instances could not be solved by at least one variant of *BCP* and are therefore not included in the average computation.

$\vec{\delta}$	k	not solved
$(20, 20, 20)^T$	40	nist-b-05-t
$(30, 30, 30)^T$	60	nist-b-05-t
$(30, 30, 30)^T$	40	nist-b-05-t, nist-b-12-t
$(40, 40, 40)^T$	$\|V\|$	nist-b-05-t
$(40, 40, 40)^T$	60	nist-b-05-t, nist-b-08-t, nist-b-07-t
$(40, 40, 40)^T$	40	nist-b-05-t, nist-b-07-t, nist-b-12-t, nist-g-12-t
$(50, 50, 50)^T$	$\|V\|$	nist-b-05-t, nist-b-12-t, nist-g-06-t, nist-g-12-t
$(50, 50, 50)^T$	60	nist-b-05-t, nist-b-09-t, nist-b-12-t, nist-g-12-t, nist-g-06-t, nist-g-12-t, nist-u-10-t, nist-u-12-t
$(50, 50, 50)^T$	40	nist-b-05-t, nist-b-07-t, nist-b-08-t, nist-b-11-t, nist-b-12-t, nist-g-05-t, nist-g-06-t nist-g-08-t, nist-g-12-t, nist-u-04-t, nist-u-12-t

instances from $nist_{small}$ that could not be solved within the time limit of four hours by at least one method and are therefore excluded from the computation of the average values are listed in Table 21.

The charts in Figure 11 visualize the average running time of the different *BCP* variants. Note that the excluded instances have an impact on the overall performance. Therefore, the different $\vec{\delta}$ values are not comparable to each other. Only the methods within one particular $\vec{\delta}$ can be compared.

The proportion of cycle elimination cuts (CEC)and directed connection cuts (DEC) for the best variant, $BCP/NL/UA$, is shown in Table 24. The average values over the solved instances from $nist_{small}$ for all variants of *BCP* with two-dimensional $\vec{\delta}$ are presented in Table 23. Again, the time

Table 22: Average runtimes of *BCP* variant 1 − 6 for the *NIST* data from group $nist_{small}$ for 3 dimensional $\vec{\delta}$.

$\vec{\delta}$		$(20,20,20)^T$								$(30,30,30)^T$							
method	k	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_{SD}[s]$	$t_p[s]$	ns	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_{SD}[s]$	$t_p[s]$	ns
BCP	$\|V\|$	186	1034	58	47.83	48.05	45.71	10.89	0	518	4758	1073	269.21	455.82	582.38	9.47	0
BCP/NL		164	1004	45	37.99	**38.54**	28.37	11.05	0	494	8655	1260	303.57	472.27	670.45	9.53	0
BCP/UA		174	760	37	44.90	46.69	41.49	10.72	0	447	3214	1010	311.60	489.94	872.97	9.26	0
BCP/NL/UA		169	1272	59	42.10	42.79	40.69	10.93	0	439	4432	903	193.91	**354.89**	430.67	9.51	0
DBCP		205	2320	69	54.40	54.71	74.44	0.00	0	498	7754	1120	277.42	429.06	558.42	0.00	0
BCP/NB		185	1069	54	49.45	49.63	44.90	10.99	0	485	3114	989	249.99	442.83	640.44	9.50	0
BCP	60	233	3046	322	130.82	185.47	313.82	38.79	0	493	3664	772	821.00	859.21	1462.96	32.90	0
BCP/NL		227	815	100	138.59	**144.89**	195.68	38.93	0	407	2879	561	538.48	578.49	1008.84	32.94	1
BCP/UA		209	5508	429	126.06	191.57	357.58	39.50	0	445	1681	540	531.73	**574.34**	781.46	33.97	0
BCP/NL/UA		260	3447	319	218.18	256.07	419.76	40.46	0	418	2003	840	885.68	926.53	2042.42	34.09	0
DBCP		230	851	107	144.64	147.77	232.36	0.00	0	447	2663	777	814.99	854.54	2129.73	0.00	1
BCP/NB		254	5655	403	139.92	218.69	408.44	39.03	0	514	2909	831	929.88	966.27	2373.64	33.14	0
BCP	40	463	683	478	380.52	**498.77**	546.85	28.31	0	682	2437	1301	1218.92	1648.03	2293.06	25.01	1
BCP/NL		442	1243	539	487.68	591.71	680.41	29.35	0	643	3667	1172	984.36	1427.28	2232.51	25.05	1
BCP/UA		488	720	492	421.34	531.62	615.30	29.52	0	666	2760	1369	1641.64	2044.12	3454.82	26.19	2
BCP/NL/UA		475	1493	741	580.73	688.10	1034.30	29.69	0	609	2354	974	920.24	**1357.47**	2163.28	26.12	1
DBCP		498	2728	854	624.38	724.40	873.11	0.00	1	683	3384	1433	1386.27	1839.48	2971.81	0.00	1
BCP/NB		467	922	549	446.14	552.57	674.35	29.02	0	655	1397	1012	1023.25	1389.43	1981.50	25.59	1

$\vec{\delta}$		$(40,40,40)^T$								$(50,50,50)^T$							
method	k	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_{SD}[s]$	$t_p[s]$	ns	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_{SD}[s]$	$t_p[s]$	ns
BCP	$\|V\|$	1045	9663	2677	747.13	1292.36	1483.13	7.62	1	1287	4100	4586	869.17	2317.67	3156.10	6.65	4
BCP/NL		843	6877	2415	486.17	**927.73**	898.01	7.74	1	1169	4468	6817	1370.52	2659.20	3527.76	6.67	3
BCP/UA		986	5108	2296	677.33	1185.37	1443.47	7.79	1	1349	4108	5486	1341.63	2721.17	3682.45	6.35	4
BCP/NL/UA		778	4935	2494	519.53	1015.61	1264.22	7.76	1	1038	4112	5184	728.59	**1955.09**	2750.46	6.42	3
DBCP		919	7561	2740	572.15	1091.55	1170.37	0.00	1	1243	8921	6242	1079.10	2463.31	3104.58	0.00	3
BCP/NB		1045	13249	3100	902.21	1431.58	2002.34	7.61	1	1395	5370	5373	1024.51	2599.69	3270.28	6.61	4
BCP	60	844	5276	1758	1282.71	2081.47	2173.21	27.31	1	1002	6990	2004	1305.83	2128.10	2088.75	22.47	7
BCP/NL		736	6550	1765	1047.47	1985.19	2493.17	27.73	1	807	3250	1746	1060.15	**1754.25**	1589.46	22.06	6
BCP/UA		774	3058	1564	1247.94	2045.06	2410.25	28.48	1	951	6511	2179	1734.74	2473.32	2773.96	22.72	7
BCP/NL/UA		684	4590	1689	954.26	**1924.07**	2682.55	28.47	1	867	6827	2510	1745.47	2454.85	3138.23	22.14	6
DBCP		815	6325	1919	1339.80	2267.32	3089.63	0.00	2	892	8634	2285	1200.90	2048.13	2035.49	0.00	7
BCP/NB		814	4520	1542	1212.23	2009.21	2512.46	27.84	2	962	4493	2002	1268.86	2072.27	2166.19	22.89	7
BCP	40	744	792	1462	826.41	**2014.83**	2047.65	21.31	3	985	3089	1334	1169.60	**1441.06**	1119.42	17.22	9
BCP/NL		759	1746	1593	996.13	2187.14	2268.18	21.81	4	1033	3747	1824	1736.24	1955.35	1739.28	17.55	11
BCP/UA		846	2587	1806	1350.37	2489.34	2792.57	21.96	4	1010	2257	1699	1859.37	2050.74	1932.81	17.80	6
BCP/NL/UA		871	1938	2122	1499.02	2873.88	3251.85	22.10	3	1189	10178	2654	2503.51	2724.96	3115.72	17.44	7
DBCP		710	1643	1504	740.58	2076.92	2162.41	0.00	4	1167	9631	2769	2481.44	2680.10	2833.68	0.00	8
BCP/NB		952	9295	2049	1490.55	2669.82	3361.66	21.84	4	1240	6960	2312	2024.12	2346.31	1870.87	17.53	9

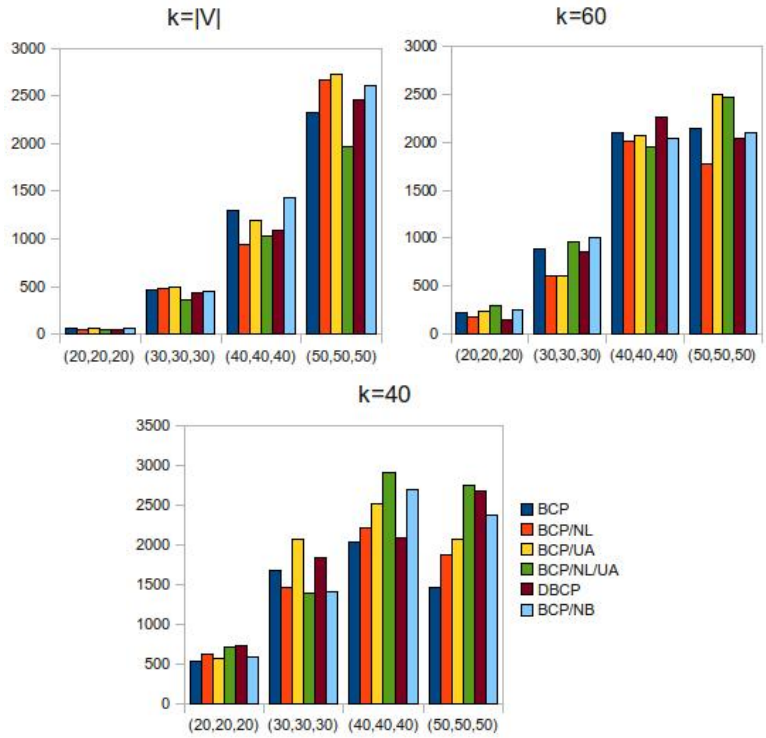

Figure 11: Average runtime of *BCP* variant 1 − 6 for $nist_{small}$.

limit was four hours. The best solution for each configuration is marked bold. The instances that could not be solved by one of the variants and are therefore not included in the computation of the averages are listed in Table 25. As we can see, the problem becomes much more difficult to solve if we use a 2-dimensional $\vec{\delta}$. Fewer instances could be solved with the parameter configuration $k = 40$ and $\vec{\delta} = (40, 40)^T$ than with the three-dimensional configuration of $\vec{\delta} = (50, 50, 50)^T$ and $k = 40$. Again, variant $BCP/NL/UA$ was able solve most instances within the time limit and BCP/NL performed best regarding the solution time. Table 26 shows the results of different *BCP* variants for instances of group *nist* that could be solved within 4 hours. The correction vector domain is $\vec{\delta} = (40, 40, 40)^T$ and $k = |V|$. For this instances the *MA* performed 500000 iterations and the population size was 10000. Generally we can say that instances with a greater number of nodes are much more difficult to solve. We can see that the instance nist-g-01-t, which contains 99 nodes, the instance nist-g-03-t, which contains 101 nodes, the instance nist-g-04-t, which contains 120 nodes, the instance nist-b-01-t, which contains 106 nodes and the instance nist-b-05-t, which contains 93 nodes could not be solved by any of the methods. It is not possible to determine an exact number of nodes and template arcs for which instances can no longer be solved, because the structure of the instance influences the solution time as well. Especially, the instance nist-b-05-t shows the relevance of the instance structure, since it contains fewer nodes and fewer template arcs than instances that could be solved within the time limit, like the instance nist-b-03-t with 107 nodes and 10142 template arcs. The greatest number of variables priced into the model was 3081 for the instance nist-u-03-t and the method BCP/UA. The greatest number of instances

Table 23: Average runtimes of *BCP* variant 1 – 6 for the *NIST* data from group $nist_{small}$ for 2 dimensional $\vec{\delta}$.

$\vec{\delta}$		$(20,20)^T$								$(30,30)^T$							
method	k	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_{SD}[s]$	$t_p[s]$	ns	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_{SD}[s]$	$t_p[s]$	ns
BCP	$\|V\|$	840	3369	3054	682.33	1310.34	2208.95	7.94	0	1140	2719	4205	696.07	1962.02	2383.33	5.73	6
BCP/NL		710	6298	2489	374.87	**812.18**	1341.92	7.88	0	994	7406	4308	426.13	**1504.17**	1809.27	5.69	5
BCP/UA		819	3401	3063	732.82	1392.31	2536.95	7.79	0	1069	1857	3334	430.25	1691.98	2308.84	5.73	8
BCP/NL/UA		655	2421	2490	444.60	968.55	2056.08	7.83	0	911	1539	4004	548.42	1648.88	2331.27	5.71	3
DBCP		722	6023	2506	326.47	888.47	1837.77	0.00	0	1043	3228	4459	635.92	1732.00	2131.04	0.00	5
BCP/NB		811	3795	2647	516.92	1192.42	1984.68	8.12	0	1197	2442	4561	946.45	2250.82	2901.79	5.80	6
BCP	60	655	4707	1794	1008.94	**1966.16**	2217.85	30.07	1	922	3799	2133	1532.38	2203.02	1960.88	21.46	8
BCP/NL		665	3728	2168	1386.18	2300.91	2244.84	30.03	1	748	6182	1603	849.42	**1558.25**	1828.17	21.46	8
BCP/UA		657	6951	2143	1254.03	2201.59	2471.39	30.06	1	819	1825	1859	1359.34	2010.71	2666.49	22.00	8
BCP/NL/UA		621	5759	1951	1141.09	1987.78	2075.75	30.06	1	773	3770	2103	1496.93	2196.82	2700.63	21.92	6
DBCP		691	3828	2126	1087.58	2208.98	2420.22	0.00	1	766	3648	1639	1002.67	1692.29	2091.33	0.00	6
BCP/NB		636	3470	1919	1016.09	2034.18	2192.22	31.20	1	888	5410	1957	1527.17	2148.64	2168.96	22.30	8
BCP	40	817	1986	2199	1504.44	2543.32	2224.15	21.93	3	883	1359	3724	1695.88	4184.07	4306.55	17.91	10
BCP/NL		732	3090	2198	1465.81	2562.72	2169.92	21.92	4	833	1590	3595	1178.06	3834.75	3640.34	17.87	11
BCP/UA		870	4205	2871	2093.29	3123.96	2988.07	21.97	3	975	2536	3437	1094.12	**3817.76**	4108.09	17.51	11
BCP/NL/UA		795	5215	3138	2128.86	3337.46	3196.23	22.13	4	723	1371	3167	998.26	3598.62	4109.07	17.39	10
DBCP		824	2472	2621	1752.33	3080.79	2961.06	0.00	3	848	1395	3615	1105.98	3795.85	3997.14	0.00	11
BCP/NB		716	705	1821	1427.25	**2447.25**	2373.94	22.86	4	794	610	3610	1094.31	3986.35	4391.52	17.45	11

$\vec{\delta} = (40,40)^T$									
method	k	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_{SD}[s]$	$t_p[s]$	ns
BCP	$\|V\|$	1625	3718	7583	1613.96	3781.41	3884.83	4.70	13
BCP/NL		1379	2037	7082	952.43	2539.11	2752.30	4.61	9
BCP/UA		1405	2603	5360	870.50	2488.28	3171.14	4.44	10
BCP/NL/UA		1206	1657	5231	493.46	**2159.60**	3130.94	4.46	9
DBCP		1388	4586	5762	794.02	2374.83	3302.58	0.00	13
BCP/NB		1508	3179	5680	780.92	2620.26	3087.99	4.53	10
BCP	60	879	798	1746	869.38	2018.78	1954.94	16.38	9
BCP/NL		929	2654	3186	1321.52	2525.20	2512.41	15.42	11
BCP/UA		1086	2476	3539	1862.43	3009.89	2943.89	15.87	9
BCP/NL/UA		1028	3244	2687	1277.60	2285.69	1874.81	16.36	7
DBCP		825	1724	1900	668.78	**1959.96**	2150.35	0.00	7
BCP/NB		966	1729	1963	1044.72	2071.92	1912.48	15.90	7
BCP	40	905	917	3833	929.94	3399.71	3898.54	12.70	15
BCP/NL		905	724	3858	1197.45	3408.48	3623.22	12.97	13
BCP/UA		835	683	3498	669.59	**3039.99**	3488.43	12.67	16
BCP/NL/UA		1031	2221	3853	1264.15	3489.05	3972.65	13.10	13
DBCP		768	504	3560	360.83	3221.77	4157.38	0.00	15
BCP/NB		883	459	4103	1556.38	3959.41	5088.31	13.05	16

Table 24: Number of directed connection cuts (DCC) and cycle elimination cuts (CEC) for the *NIST* data from group $nist_{small}$ and variant $BCP/NL/UA$.

	$\vec{\delta}$				$(20,20,20)^T$		$(30,30,30)^T$		$(40,40,40)^T$		$(50,50,50)^T$	
inst.	$\|V\|$	$\|T^c\|$	$tt[s]$	m	DCC	CEC	DCC	CEC	DCC	CEC	DCC	CEC
nist-b-04-t	81	5954	0.07	37	316	0	906	112	3955	383	1278	240
nist-b-06-t	77	5306	0.05	33	2711	320	9611	1151	2140	301	680	50
nist-b-07-t	82	5988	0.04	35	992	73	2117	365	6472	1061	1279	132
nist-b-08-t	76	5068	0.05	32	729	51	3854	431	756	87	2220	316
nist-b-09-t	76	4978	0.05	30	742	67	1010	142	12195	1750	8725	701
nist-b-10-t	63	3664	0.04	31	62	0	1076	144	5707	878	955	174
nist-b-11-t	80	5598	0.06	33	238	1	6573	834	4735	732	4005	760
nist-g-05-t	80	5772	0.05	34	362	29	8260	1563	2244	347	1881	299
nist-g-07-t	74	4706	0.06	29	289	17	475	42	2683	354	810	107
nist-g-08-t	82	5846	0.05	33	162	0	14535	2346	790	118	728	70
nist-g-09-t	53	2500	0.02	22	358	12	61	1	733	95	356	100
nist-g-10-t	76	5112	0.06	32	393	32	733	64	440	27	2306	316
nist-g-11-t	67	4026	0.04	31	137	1	1894	255	1933	396	898	107
nist-g-13-t	55	2842	0.04	30	1951	290	581	62	218	4	272	26
nist-u-04-t	84	6250	0.07	34	3355	437	964	73	2851	599	20629	3448
nist-u-05-t	73	4829	0.05	32	395	42	852	75	435	38	199	5
nist-u-06-t	70	4414	0.05	32	2823	421	149	18	2128	375	2583	392
nist-u-07-t	63	3654	0.04	30	288	2	1132	197	396	75	2635	481
nist-u-08-t	74	4952	0.04	35	89	0	445	42	1935	223	3471	533
nist-u-09-t	69	4340	0.05	32	341	74	2155	272	399	55	3919	720
nist-u-10-t	76	5056	0.05	31	462	64	8851	1544	22397	3885	322	21
nist-u-11-t	63	3574	0.04	30	373	1	1504	285	1602	333	4615	744
nist-u-12-t	80	5544	0.06	33	373	31	6188	761	3925	586	17490	2588

Table 25: Unsolved instances of $nist_{small}$ for 2 dimensional $\vec{\delta}$. These instances could not be solved by at least one variant of BCP and are therefore not included in the computation of the average values.

$\vec{\delta}$	k	not solved
$(20,20)^T$	60	nist-b-05-t
$(20,20)^T$	40	nist-b-05-t, nist-b-12-t, nist-g-06-t, nist-g-12-t, nist-u-04-t, nist-u-10-t, nist-u-12-t
$(30,30)^T$	$\|V\|$	nist-b-05-t, nist-b-07-t, nist-b-11-t, nist-b-12-t, nist-g-06-t, nist-g-07-t, nist-g-08-t, nist-g-12-t
$(30,30)^T$	60	nist-b-05-t, nist-b-07-t, nist-b-09-t, nist-g-06-t nist-g-05-t, nist-g-12-t, nist-u-04-t, nist-u-10-t
$(30,30)^T$	40	nist-b-04-t, nist-b-05-t, nist-b-06-t, nist-b-07-t, nist-b-09-t, nist-b-11-t, nist-b-12-t, nist-g-06-t nist-g-08-t, nist-g-10-t, nist-g-12-t, nist-u-04-t, nist-u-06-t
$(40,40)^T$	$\|V\|$	nist-b-05-t, nist-b-06-t, nist-b-07-t, nist-b-08-t, nist-b-11-t, nist-b-12-t, nist-g-05-t, nist-g-06-t nist-g-07-t, nist-g-08-t, nist-g-10-t, nist-g-12-t, nist-u-04-t, nist-u-11-t, nist-u-12-t
$(40,40)^T$	60	nist-b-05-t, nist-b-06-t, nist-b-08-t, nist-b-07-t, nist-b-11-t, nist-b-12-t, nist-g-06-t, nist-g-10-t, nist-g-12-t, nist-u-04-t, nist-u-06-t, nist-u-10-t, nist-u-12-t
$(40,40)^T$	40	nist-b-04-t, nist-b-05-t, nist-b-06-t, nist-b-07-t, nist-b-08-t, nist-b-09-t, nist-b-11-t, nist-b-12-t, nist-g-05-t, nist-g-06-t nist-g-07-t, nist-g-08-t, nist-g-12-t, nist-u-04-t, nist-u-05-t, nist-u-06-t, nist-u-08-t, nist-u-11-t, nist-u-12-t

could once again be solved by BCP/NL and $BCP/NL/UA$. Table 27 shows the average values for solved instances from group *nist* for all BCP variants within a time limit of 4 hours. Again, the number of instances that a specific method could not solve within the time limit is denoted by ns and the best result is printed bold. Compared to the results of group $nist_{small}$, we can see that much fewer instances could be solved. For the parameter setting $\vec{\delta} = (40,40,40)^T$ and $k = |V|$ all but one instance from group $nist_{small}$ could be solved within the time limit by all methods. From group *nist* about half of the instances could not be solved with the same parameter configuration. The variant BCP/NL

Table 26: Running times of *BCP* variant 1 − 6 for the *NIST* data from group *nist* with $\vec{\delta} = (40, 40, 40)^T$ and $k = |V|$. If $t_{best}[s] = 0.0$ the *MA* found the best solution.

					BCP						BCP/NL					
inst.	$\|V\|$	$\|T^c\|$	$tt[s]$	m	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_p[s]$	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_p[s]$
nist-b-02-t	94	8278	0.13	17	2451	1507	6034	5468.9	5617.38	553.48	2367	2096	13722	8849.26	8885.05	530.78
nist-b-03-t	107	10142	0.15	22	ns	ns	ns	ns	ns	ns	1942	8551	5723	2910.77	5819.95	767.84
nist-b-04-t	81	4986	0.05	22	889	1088	3446	102.61	1378.44	504.86	745	1983	2786	58.42	897.91	504.45
nist-g-02-t	101	7820	0.10	24	1812	10289	3721	3541.09	4339.43	750.58	1524	25069	3524	2165.19	3109.88	748.98
nist-g-05-t	80	4764	0.07	21	834	469	1983	0.00	878.66	485.02	715	390	1875	0.00	553.41	477.93
nist-u-01-t	99	7339	0.12	24	ns	ns	ns	ns	ns	ns	1874	12559	18250	1838.54	11998.04	674.90
nist-u-02-t	93	6924	0.11	20	1062	1253	1054	404.65	1439.17	641.79	1533	4237	3540	2044.86	2447.48	637.15
nist-u-04-t	84	5238	0.06	21	834	757	2008	180.79	1296.16	569.97	895	2042	2317	628.02	1255.4	578.33
nist-u-05-t	73	4002	0.06	19	668	717	881	270.05	328.19	478.32	457	898	349	96.14	156.55	469.18

					BCP/UA						BCP/NL/UA					
inst.	$\|V\|$	$\|T^c\|$	$tt[s]$	m	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_p[s]$	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_p[s]$
nist-b-03-t	107	10142	0.18	22	ns	ns	ns	ns	ns	ns	2375	13434	8179	5062.64	7455.26	757.29
nist-b-04-t	81	4986	0.07	22	1133	9851	3739	386.85	1478.27	491.34	722	1207	2796	92.30	1285.75	508.71
nist-g-02-t	101	7820	0.12	24	1775	3683	4475	3626.92	4629.27	724.8	1176	758	2534	1586.45	2407.27	746.00
nist-g-05-t	80	4764	0.06	21	821	449	2025	0.00	1017.04	488.38	742	551	2413	0.00	709.98	476.18
nist-u-01-t	99	7339	0.11	24	ns	ns	ns	ns	ns	ns	1835	5254	18314	2101.19	12369.19	699.87
nist-u-02-t	93	6924	0.12	20	1121	1101	1213	543.9	1361.54	619.51	1179	3470	1963	1167.96	1643.86	645.62
nist-u-03-t	100	9260	0.18	19	3081	1117	9755	9574.46	11277.34	584.47	2580	2937	11765	9394.14	10697.54	610.48
nist-u-04-t	84	5238	0.06	21	1050	4836	2899	756.24	1634.46	554.51	743	1082	2746	206.61	1401.51	579.58
nist-u-05-t	73	4002	0.06	19	442	298	230	101.46	183.17	441.15	567	873	672	260.61	296.02	462.34

					BCP/NB						DBCP				
inst.	$\|V\|$	$\|T^c\|$	$tt[s]$	m	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_p[s]$	col	cut	nod	$t_{best}[s]$	$t[s]$
nist-b-02-t	94	8278	0.15	17	2489	785	6873	7698.98	7835.0	690.51	2071	537	6779	5105.36	5172.37
nist-b-04-t	81	4986	0.07	22	923	1054	3252	126.25	1724.4	633.89	1145	18587	6074	299.61	2085.72
nist-g-02-t	101	7820	0.14	24	1449	18443	1920	2454.63	3686.64	902.73	1656	12906	5398	3807.41	4815.42
nist-g-05-t	80	4764	0.09	21	834	469	1983	0.00	1067.96	622.67	776	469	2013	19.13	830.70
nist-u-02-t	93	6924	0.12	20	1048	1340	1049	559.08	1733.91	738.89	907	1652	880	287.58	841.51
nist-u-03-t	100	9260	0.16	19	ns	ns	ns	ns	ns	ns	3063	23221	11171	6996.61	8701.43
nist-u-04-t	84	5238	0.09	21	862	1817	2161	277.32	1598.9	713.69	1051	5495	2944	423.19	1620.72
nist-u-05-t	73	4002	0.05	19	668	717	881	305.35	369.25	591.24	814	13411	1417	378.67	445.50

performed best in terms of solved instances, while variant $BCP/NL/UA$ has the lowest running time. Note, however, that for this comparison the long preprocessing time is not considered. If we consider it, $DBCP$ is the fastest method. Table 28 shows some instances with $|V| > 100$ and a difficult to solve instance structure.

We see that the time required to process instances that where previously excluded due to the time limit is often much longer than the time limit. The running time ist mostly determined by the number of template arcs and the number of nodes, but it also depends on the structure of the respective instance. Method $BCP/NL/UA$ for larger $\vec{\delta}$ values is presented in Table 29. We chose a high time limit of 2000 minutes to see how much we can enlarge the correction vector. Recall that larger correction vectors lead to more template arcs, and thus to a greater repertory of variables that can be priced into the equation. Larger correction vectors $\vec{\delta}$ generally lead to better compression ratios. The best compression rate (18.9%) is achieved with $\tilde{\delta} = (80, 80, 80)^T$ and $k = 40$. This combination cannot be solved by the BCP approach. But even if we set the parameter k to $|V|$, which makes the instance much easier to solve, the limitation of our method is a correction vector size around $\vec{\delta} = (70, 70, 70)^T$. Except, of course, if we use instances with a very small number of nodes. Instances with a very large $\vec{\delta}$ value do not become easier to solve. Either all instances are solved immediately by the *MA* and the

Table 27: Average runtimes of *BCP* variant 1 – 5 for the *NIST* data from group *nist*.

k		80								$\|V\|$							
method	$\vec{\delta}$	*col*	*cut*	*nod*	$t_{best}[s]$	$t[s]$	$t_{SD}[s]$	$t_p[s]$	*ns*	*col*	*cut*	*nod*	$t_{best}[s]$	$t[s]$	$t_{SD}[s]$	$t_p[s]$	*ns*
BCP	(20)	318	1052	241	632.88	641.35	900.52	3023.13	1	305	5252	160	270.71	271.77	323.71	887.72	0
BCP/NL	(20)	249	1348	189	468.43	**482.39**	864.97	2987.77	0	369	6750	191	199.68	202.06	187.44	884.37	0
BCP/UA	(20)	320	814	311	788.90	797.49	1256.02	2973.00	0	221	927	36	159.47	160.07	150.32	884.09	0
BCP/NL/UA		256	946	391	809.79	826.52	1897.23	3104.36	0	273	2139	102	137.37	**138.35**	126.95	908.24	0
DBCP		416	3396	366	828.76	830.82	1146.33	-	0	364	3940	139	181	182.01	134.97	-	0
BCP	(30)	410	3244	383	195.36	195.44	68.47	573.64	2	940	6480	3523	1707.07	2444.14	1983.18	723.88	1
BCP/NL	(30)	409	5417	313	141.23	141.76	54.44	606.61	1	876	12521	3207	1424.98	**2148.68**	1661.85	762.99	1
BCP/UA	(30)	426	4596	648	256.03	258.56	185.25	557.26	2	957	8217	3107	1588.30	2350.91	2051.90	709.75	1
BCP/NL/UA		347	1536	333	113.45	**114.53**	63.54	585.12	2	849	6854	2996	1507.58	2346.05	2110.38	740.20	1
DBCP		515	10228	484	188.14	190.85	121.73	-	2	1002	12750	3006	1484.97	2181.63	1612.64	-	1
BCP	(40)	941	9177	1193	1583.16	**1883.95**	1667.22	1533.80	8	1017	2429	2182	749.87	1610.01	1278.60	571.76	6
BCP/NL	(40)	840	8086	1426	2092.38	2311.15	2515.08	1554.92	6	978	5770	2399	832.11	1403.44	1045.36	569.34	6
BCP/UA	(40)	978	5154	2057	2897.36	3225.33	3106.47	1507.38	7	1057	3370	2430	902.56	1717.29	1385.28	553.28	6
BCP/NL/UA		814	5826	1460	1667.39	1980.31	2151.84	1568.64	6	855	1324	2187	552.32	**1290.73**	672.11	569.74	7
DBCP		1238	35851	2756	3442.15	3690.90	4327.02	-	7	1058	753	3121	869.27	1773.26	1466.01	-	7

Table 28: Running times for instances with $|V| > 100$ and $k = |V|$.

inst.	method	$\vec{\delta}$	V	$\|T^c\|$	$tt[s]$	m	*col*	*cut*	*nod*	$t[s]$	$t_p[s]$
nist-b-01-t	*BCP/NL*	$(40, 40, 40)^T$	106	9559	0.17	22	2839	12287	78026	71863.76	13.59
nist-g-03-t			102	9618	0.28	19	3296	11971	47239	96387.83	18.13

lower bound computation, because only one, two or three template arcs are contained in a feasible solution, or the solution time increased sharply in comparison to $\vec{\delta} = (70, 70, 70)^T$. All instances from the *nist* group were solved immediately for $\vec{\delta} = (300, 300)^T$ and $\vec{\delta} = (200, 200)^T$ with $k = 80$, $k = 40$, $k = 20$ and $k = 10$. For $\vec{\delta} = (150, 150)^T$ with $k = 80$, $k = 40$, $k = 20$ and $k = 10$ the instances from the *nist* group were solved immediately for all k values except $k = 80$. For $k = 80$ a few instances could not be solved immediately. None of the instances that were not solved immediately could be solved within the time limit of four hours. Some runs that were performed with a higher time limit allow the conclusion that the solution time increases a lot for very large correction vectors, so the instances can no longer be solved within a reasonable amount of time. E.g. the instance nist-u-05 was solved in 402.25 seconds for $k = |V|$ and $\vec{\delta} = (70, 70, 70)^T$ and could not be solved within 1100 minutes for $k = 73$ and $\vec{\delta} = (150, 150)^T$.

5.4.1 Conclusions

The *BCP* approach could solve *NIST* data up to a correction vector size of $\vec{\delta} = (50, 50, 50)^T$ for instances with up to 80 nodes within a reasonable amount of time and was therefore an improvement over the *BC* approach, which could not solve *NIST* data at all, and over the *BP* approach, which was also developed in this project. If only a subset $k = 40$ of all nodes is selected the instances become much more difficult to solve. Moreover, problems with a 2-dimensional correction vector require much more time and can solve a much smaller number of instances within the time limit. The

Table 29: Running times of $BCP/NL/UA$ for large $\vec{\delta}$ and a time limit of 2000 minutes.

$k = \|V\|$, $BCP/NL/UA$											
inst.	$\vec{\delta}$	V	$\|T^c\|$	$tt[s]$	m	col	cut	nod	$t_{best}[s]$	$t[s]$	$t_p[s]$
nist-b-04-t	$\binom{60}{60}$	81	7331	0.13	14	878	1106	786	186.59	509.28	6.96
nist-b-05-t		93	13010	0.29	ns	ns	ns	ns	ns	ns	ns
nist-b-06-t		77	7632	0.14	13	2424	5133	7274	2733.40	2733.40	6.19
nist-b-07-t		82	8459	0.17	14	2609	4069	79605	301.11	33203.89	6.80
nist-b-08-t		76	7716	0.16	12	2259	33446	31669	879.26	13420.56	6.23
nist-b-09-t		76	8612	0.17	11	2295	1091	18746	742.21	7627.27	5.33
nist-b-10-t		63	4106	0.08	12	602	542	467	142.61	151.50	4.45
nist-b-11-t		80	8117	0.13	13	2389	3858	18767	2337.86	8868.76	6.73
nist-b-12-t		85	11660	0.25	12	3291	1213	41450	4131.32	29401.65	7.36
nist-g-05-t		80	7293	0.13	14	1638	5858	2913	1089.89	1551.36	6.64
nist-g-06-t		87	10480	0.21	14	3955	982	196026	2417.59	101773.77	7.49
nist-g-07-t		74	6578	0.11	12	2721	10026	113515	218.90	39918.70	5.68
nist-g-08-t		82	9420	0.17	13	4063	5182	32687	12962.30	12992.34	6.42
nist-g-09-t		53	3007	0.05	9	443	679	245	68.75	68.76	3.21
nist-g-10-t		76	7319	0.15	13	1623	434	11316	113.91	4277.32	6.13
nist-g-11-t		67	5201	0.11	12	1264	3005	7700	99.95	1805.54	4.92
nist-g-12-t		84	9974	0.2	12	1737	171	5542	216.21	3767.10	7.10
nist-g-13-t		55	2528	0.04	13	300	667	104	37.28	45.89	3.78
nist-u-04-t		84	7992	0.15	14	1416	4728	3235	597.42	2119.70	7.08
nist-u-05-t		73	5775	0.11	13	1384	1717	2546	647.53	822.38	6.02
nist-u-06-t		70	5200	0.08	13	1268	30	9725	61.20	2323.65	5.59
nist-u-07-t		63	4192	0.07	13	1696	9397	89725	143.18	12657.94	4.64
nist-u-08-t		74	5977	0.12	14	1644	363	18663	222.50	5291.75	6.13
nist-u-09-t		69	4686	0.07	13	1426	1523	31825	75.62	7677.18	5.19
nist-u-10-t		76	7002	0.14	12	1425	969	5832	120.95	2712.12	6.05
nist-u-11-t		63	4173	0.08	12	902	1996	1405	106.19	328.42	4.08
nist-u-12-t		80	9520	0.21	12	1578	3637	3341	918.76	2279.28	6.08
nist-b-04-t	$\binom{70}{70}$	81	10110	0.21	12	3990	2304	50170	20077.20	20077.21	6.36
nist-b-05-t		93	18417	0.48	ns	ns	ns	ns	ns	ns	ns
nist-b-06-t		77	10429	0.24	11	4017	3020	23515	8975.49	8975.49	5.0
nist-b-07-t		82	11296	0.27	ns	ns	ns	ns	ns	ns	ns
nist-b-08-t		76	11177	0.2	9	1827	1585	8528	102.49	3939.55	5.22
nist-b-09-t		76	11783	0.29	9	2283	622	8610	1488.91	4525.65	4.82
nist-b-10-t		63	5358	0.09	11	1846	193	42181	52.46	7147.75	3.99
nist-b-11-t		80	10848	0.23	11	2080	2443	8046	220.42	4495.57	5.52
nist-b-12-t		85	16747	0.41	10	7714	6541	95576	32518.70	56471.05	6.21
nist-g-05-t		80	9553	0.21	12	3430	15619	16758	3711.54	7673.07	6.69
nist-g-06-t		87	14873	0.37	ns	ns	ns	ns	ns	ns	ns
nist-g-07-t		74	8886	0.18	9	2340	738	20453	127.22	8419.46	4.83
nist-g-08-t		82	12614	0.31	11	3300	596	13576	3110.64	8919.61	5.65
nist-g-09-t		53	3994	0.08	8	383	325	182	31.78	52.63	3.21
nist-g-10-t		76	10036	0.21	11	3012	3626	35960	1774.31	13960.82	5.65
nist-g-11-t		67	6995	0.14	10	1234	753	2139	346.83	831.51	4.23
nist-g-12-t		84	13896	0.32	ns	ns	ns	ns	ns	ns	ns
nist-g-13-t		55	3079	0.06	11	591	396	471	77.61	99.40	3.23
nist-u-04-t		84	10594	0.21	12	2850	1739	27054	2492.28	15576.72	6.45
nist-u-05-t		73	7563	0.15	11	948	1158	880	342.03	402.25	5.11
nist-u-06-t		70	6787	0.14	11	1281	2100	2191	247.85	829.05	4.59
nist-u-07-t		63	5334	0.12	11	1650	755	24730	37.04	4081.77	4.16
nist-u-08-t		74	7937	0.16	12	2085	1322	18578	265.45	5979.78	5.17
nist-u-09-t		69	5950	0.11	12	2016	4012	28780	154.05	6937.01	4.81
nist-u-10-t		76	9352	0.2	10	2445	10388	7377	1852.13	2954.88	5.00
nist-u-11-t		63	5435	0.08	10	842	203	1172	83.54	381.70	3.78
nist-u-12-t		80	13660	0.32	9	5610	1286	55303	24829.70	24829.69	4.84

reason for this is that more template arcs are generated for 2-dimensional correction vectors up to a correction vector size of $\tilde{\delta} = (120, 120, 120)^T$ ($\tilde{\delta} = (80, 80)^T$). The opposite effect can be observed at the correction vector size of $\tilde{\delta} = (120, 120, 120)^T$ ($\tilde{\delta} = (120, 120)^T$), where many more template arcs are generated for 3-dimensional correction vectors. Overall we can conclude that instances with many nodes V and template arcs t, where only a small subset of nodes k is selected are more difficult to solve. The number of columns has the greatest impact on the running time. Regarding the different variants of *BCP*, we we can say in summary that using back-cuts is an improvement in many cases, especially if $k = 40$. From all methods, $BCP/NL/UA$ performed best in terms of solved instances and solution time. Generally, the *BCP* variants show good performance for different instances.

6. Implementation

All algorithms have been implemented in C++ using the Standard Template Library (STL) and the Boost Library (http://www.boost.org/) and are embedded in a framework developed by Dietzel [22]. The code was compiled with g++-4.1. For graph algorithms as well as set and graph data structures we used version 5.1 of LEDA (Library of Efficient Data Types and Algorithms) [33]. In Chapter 4 we used EAlib2, developed at the Institute for Computer Graphics and Algorithms of the Vienna University of Technology for the memetic algorithm. As a Branch-and-Cut-and-Price framework we used SCIP 1.2.0 (Solving Constraint Integer Programs)[2] with version 11.2 of CPLEX [27] as the underlying LP-Solver.

6.1 Module Structure

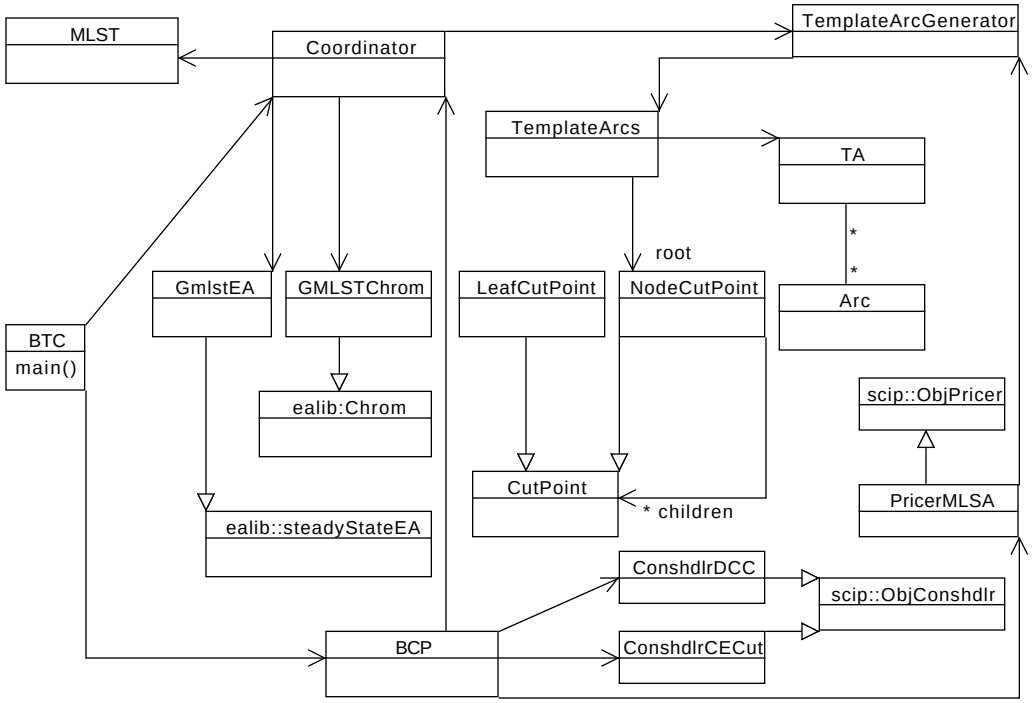

Figure 12: Overview of the main classes.

The *Coordinator* class is the connection to the existing framework. It fetches the arcs, which are generated from the input data, from the class MLST and is responsible for the encoding of the solution. It also configures and runs the *MA*.

Table 30: Symbols used in `getBestLabel` (Algorithm 11) and `checkCycleTwo` (Algorithm 12)

a	the algorithm: the MA (gen_mlst), BCP (BCP) or if we only want to create all template arcs (dFirt).
e	path to a file containing settings specific to the *MA*
s	the numbers of starts of the *MA*.
b	the *BCP* initialization method: initialization using the *MA* (0), initialization with all TAs (1), random initialization (2), initialization using *DNIS* (3). Note that (1) is only there for testing purposes.
z	the variant of pricing as shwon in table 30 : *BCP* (0), *BCP*/*NL* (1), *BCP*/*UA* (4), *BCP*/*NL*/*UA* (5), *DBCP* (7).
u	back-cuts: activate back-cuts (1) - this is the default setting, deactivate back-cuts *BCP*/*NB* (0).
w	lower bound: compute lower bound (1), do not compute lower bound (0).

The class *TemplateArcGenerator* is responsible for managing the template arcs. It initializes the generation of all template arcs if all template arcs are generated in advance and computes a lower bound if requested (see Chapter 5.3.4.2). Whenever the pricer requests a template arc, it returns the template arc with the highest value. This template arc is either selected from the set of pre-existing template arcs or generated from scratch *DNIS*.

The class *TemplateArcs* is responsible for creating the template arcs either using the *NIS* algorithm, if all template arcs are created in advance, or using *DNIST* if one template arc per call is created on demand. The classes *CutPoint*, *NodeCutPoint* and *LeafCutpoint* contain the implementation of *NIS* and *DNIST*. The class *TA* represents a single template arc.

The class *BCP* solves the MIP by calling the appropriate functions of SCIP. Besides the linear constraints included in SCIP, users can configure their own constraint handlers. The constraint handler *ConshdlrCECut* generates the cycle elimination cuts, and the constraint handler *ConshdlrDCCut* generates the directed connection cuts. For the invocation of the `main()` function the algorithms *MA* and *BCP* require the following command line parameters:

The parameters that are not specific to our problem are already described in [22].

6.2 SCIP

In the following, we describe how some of the features of SCIP were used. SCIP uses a plugin based architecture that can be extended and customized via user-defined callback objects. The function *SCIPcreateVar* is used to generate the variables and the function *SCIPaddVar* adds the variables. The most important SCIP plugins are constraint handlers. Each class of constraints is represented by a constraint handler. The constraint handler checks the feasibility of a solution with respect to its associated constraints. The constraint handlers for the linerar constraints included in SCIP are created with *SCIPcreateConsLinear* and added with *SCIPaddCons*. They were implemented according to our ILP model and used the recommended default values for the parameters, except for the parameters *enforce* and *check*, which are set to true, since there are no redundant constraints. Since we use a pricing approach, the parameter *modifiable*, which determines whether a variable is subject to column generation, must be true for the arc-label and node-label constraints. Custom contraint handlers are

added using *SCIPincludeObjConshdlr*, which makes the constraint handler available to the model. Individual constraints are created by the callback function *SCIPcreateConsX*.

We added two custom constraint handlers, *ConshdlrDCC* for the directed connection cuts and *ConshdlrCECut* for the cycle elimination cuts. These constraint handlers can also be seen in Figure 12. The two constraint handler classes inherit from *scip::ObjConshdlr*. The following constraint handler properties were configured: The separation priority *CONSHDLR_SEPAPRIORITY*, i.e. the priority of the separation methods of the respective constraint is set to 1000000 for both constraints. This has to be seen in relation to the linear constraint handlers, which are much easier to separate, and whose priority is set to 100000. The enforcement priorities *CONSHDLR_ENFOPRIORITY*, i.e. the priority of the enforcement methods, as well as the priority of constraint checking *CONSHDLR_CHECKPRIORITY* is set to -201000 for *ConshdlrDCC* and to -200000 for *ConshdlrCECut*. The separation frequency *CONSHDLR_SEPAFREQ* defines the frequency of the cut separation, i.e. at which depth level of the tree the separation methods are called. For both constraint handlers the separation frequency is set to 40. The pricer class inherits from *scip::ObjPricer*. It must implement the virtual callback function *scip_redcosts*, which searches for new variables and adds them to the equation. We also implemented the optional virtual function *scip_farkas*, which generates additional variables that make an infeasible variable feasible again [2].

6.3 Test Setups

For our tests, we use two different datasets. The first dataset containing 20 templates was provided by the Fraunhofer Institute Berlin *Fraunhofer Templates*. The second test set comes from the U.S. National Institute of Standards and Technology [24]. All tests where performed on a Dual Core AMD Opteron 270, 1.9 GHz with 8GB RAM under Linux kernel 2.6.62, or on a Pentium 4 with 2GB memory under Linux kernel 2.4.21. The time limit for the tests was usually four hours. The specific test enviorments and time limits are given in the results section of the respective chapters.

7. Conclusion and Further Work

In this thesis, exact and heuristic methods for the compression of fingerprint templates as well as a new algorithm *NIS* for the construction of template arcs have been proposed. Since the number of variables (labels) in our problem is huge we decided not to use all variables but add them on demand in the pricing approach of a Branch-an-Cut-and-Price framework. This strategy was effective since the *BCP* approach can deal with much larger datasets than the *BC* approach (see [18]) as the pricing approach only deals with a subset of all variables. Combining row and column generation in *BCP* was also more efficient than the *BP* approach [46] in terms of running time and the size of the exactly solvable instances.

For cut separation, cycle elimination cuts and directed connection cuts were used. Not separating cuts in every node yields better results. The developed *BCP* method is able to successfully solve instances from the *NIST* dataset containing up to 9000 template arcs and 90 nodes with a correction vector size of $\vec{\delta} = (50, 50, 50)^T$ in a reasonable amount of time. Larger correction vectors $\vec{\delta}$ generally lead to better compression ratios. However, larger correction vectors also cause the generation of a much greater number of candidate template arcs, so that the problem can no longer be solved using the *BCP* approach. If we use lossy compression and select only a subset k of all nodes, the performance of the *BCP* algorithm decreases dramatically.

The developed heuristic method is a memetic algorithm (*MA*). As part of the *MA*, feasible arborescences are searched for very frequently using depth first search (DFS). We have developed a technique to significantly reduce the number of these searches. The *MA* can deal with a large number of correction vectors in combination with a subset k of the nodes. The best compression rate (18.9%) is achieved with $\tilde{\delta} = (80, 80, 80)^T$ and $k = 40$. This combination cannot be solved by the *BCP* because of the high running time. Therefore, even though the *BCP* method is able to deal with a greater number of variables than existing exact approaches, it is outperformed by the *MA* in terms of the compression rate.

To improve the efficiency of the *BCP* process, we construct a near optimal, feasible start solution with a memetic algorithm, and compute a lower bound using a reduced version of the *k-MLSA* problem. The lower bound computation significantly improves the solution time, because with the BCP approach, verifying whether a particular solution is indeed the best solution is often more time consuming than generating the solution. Preprocessing the data can now be done in a very short time with *NIS*. Thus, the entire compression procedure can be completed within a few minutes using the *MA*.

The procedure *NIS* constructs the set of all non-dominated template arcs, i.e. all different non-dominated sets of arcs that can be represented together by a common template arc t. It is a two phase algorithm, which uses a divide and conquer approach. In phase one, sets are built that contain at least one element representable with all other elements. In phase two, these sets are divided further, so that all elements contained in one set are representable together. The main of the algorithm is that dominated sets are never built, since the algorithm can determine if an arc set is dominated by considering the geometric position of the arcs. The procedure *NIS* takes only a few seconds to generate up to 100000 candidate template arcs. It generates half a million candidate template arcs in just a few minutes and can be used to generate up to 1.7 million candidate template arcs. A variant of *NIS*, called

DNIST, can find the solution to the pricing-problem efficiently.

Developing other objective functions could be the subject of further research. For instance, the size of the correction vector domain could be optimized while the number of candidate template arcs remains the same. Also, a multi-objective optimization approach could be considered to minimize the size of the correction vector, the number of selected nodes and the number of template arcs (i.e. the size of the codebook) all at the same time. It may also be beneficial to hybridize the *BCP* algorithm with meta-heuristics to cope with a larger number of variables. However, these approaches are beyond the scope of this thesis.

List of Tables

List of Figures

List of Algorithms

Bibliography

[1] T. Achterberg. SCIP - a framework to integrate constraint and mixed integer programming. Technical Report 04-19, Zuse Institute Berlin, 2004. `http://www.zib.de/Publications/abstracts/ZR-04-19/`.

[2] T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, February 1993.

[4] T. Bäck, D. Fogel, and Z. Michalewicz. *Handbook of evolutionary computation*. Oxford Univ. Press, 1997.

[5] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1996.

[6] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization (Athena Scientific Series in Optimization and Neural Computation, 6)*. Athena Scientific, February 1997.

[7] T. Brüggemann, J. Monnot, and G. J. Woeginger. Local search for the minimum label spanning tree problem with bounded color classes. *Operations Research Letters*, 31(3):195 – 201, 2003.

[8] M. Captivo, J. ao C. N. Clímaco, and M. M. B. Pascoal. A mixed integer linear formulation for the minimum label spanning tree problem. *Comput. Oper. Res.*, 36(11):3082–3085, 2009.

[9] R. Cerulli, A. Fink, M. Gentili, and S. Voß. Metaheuristics comparison for the minimum labelling spanning tree problem. In R. Sharda, S. Voß, B. Golden, S. Raghavan, and E. Wasil, editors, *The Next Wave in Computing, Optimization, and Decision Technologies*, volume 29 of *Operations Research/Computer Science Interfaces Series*, pages 93–106. Springer US, 2005.

[10] R.-S. Chang and S.-J. Leu. The minimum labeling spanning trees. *Inf. Process. Lett.*, 63(5):277–282, 1997.

[11] Y. Chen, N. Cornick, A. O. Hall, R. Shajpal, J. Silberholz, I. Yahav, and B. L. Golden. Comparison of heuristics for solving the gmlst problem. In R. Sharda, S. Voß, S. Raghavan, B. Golden, and E. Wasil, editors, *Telecommunications Modeling, Policy, and Technology*, volume 44 of *Operations Research/Computer Science Interfaces Series*, pages 191–217. Springer US, 2008.

[12] B. V. Cherkassky and A. V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[13] V. Chvatal. *Linear Programming (Series of Books in the Mathematical Sciences)*. W. H. Freeman, September 1983.

[14] A. M. Chwatal. *On the Minimum Label Spanning Tree Problem. Solution Methods and Applications.* PhD thesis, "Vienna University of Technology, Institute of Computer Graphics and Algorithms", 2010.

[15] A. M. Chwatal and G. R. Raidl. Solving the minimum label spanning tree problem by ant colony optimization.

[16] A. M. Chwatal and G. R. Raidl. Solving the minimum label spanning tree problem by mathematical programming techniques. Technical report, Vienna University of Technology, 2010.

[17] A. M. Chwatal, G. R. Raidl, and O. Dietzel. Compressing fingerprint templates by solving an extended minimum label spanning tree problem. In *Proceedings of the Seventh Metaheuristics International Conference (MIC)*, Montreal, Canada, 2007.

[18] A. M. Chwatal, G. R. Raidl, and K. Oberlechner. Solving a k-node minimum label spanning arborescence problem to compress fingerprint templates. *Journal of Mathematical Modelling and Algorithms*.

[19] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill Science/Engineering/Math, 1 edition, September 2006.

[20] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications (2nd ed.)*. Springer-Verlag, 2000.

[21] J. Desrosiers and M. E. Lübbecke. A primer in column generation. In G. Desaulniers, J. Desrosiers, and M. M. Solomon, editors, *Column Generation*, pages 1–32. Springer US, 2005.

[22] O. Dietzel. Combinatorial Optimization for the Compression of Biometric Templates. Master's thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, May 2008.

[23] C. Duin and S. Voß. The pilot method: A strategy for heuristic repetition with application to the steiner problem in graphs. *Networks*, 34:181–191, 1999.

[24] Garris M. D. and McCabe R. M. NIST special database 27: Fingerprint minutiae from latent and matching tenprint images. Technical report, National Institute of Standards and Technology, 2000.

[25] M. Gen and R. Cheng. *Genetic Algorithms and Engineering Optimization (Engineering Design and Automation)*. Wiley-Interscience, December 1999.

[26] H. Holland. *Adaptation in Natural and Aritificial Systems*. The University of Michigan Press, 1975.

[27] ILOG Concert Technology, CPLEX. ILOG. http://www.ilog.com. Version 11.0.

[28] A. Jain and U. Uludag. Hiding fingerprint minutiae in images. In *Proceedings of Third Workshop on Automatic Identification Advanced Technologies*, pages 97–102, 2002.

[29] D. E. Knuth. *The art of computer programming, volume 2: (3rd ed.) Seminumerical Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[30] T. Koch and A. Martin. Solving steiner tree problems in graphs to optimality. *Networks*, 32:207–232, 1998.

[31] S. O. Krumke and H.-C. Wirth. On the minimum label spanning tree problem. *Information Processing Letters*, 66(2):81–85, 1998.

[32] L. Ladányi, T. K. Ralphs, and J. L. E. Trotter. Branch, cut, and price: Sequential and parallel. In *Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions [based on a Spring School]*, pages 223–260, London, UK, 2001. Springer-Verlag.

[33] Library for Efficient Datastructures and Algorithms (LEDA). Algorithmics Solutions Software GmbH. http://www.algorithmic-solutions.com/. Version 5.1.

[34] I. Ljubic. *Exact and Memetic Algorithms for Two Network Design Problems.* PhD thesis, Faculty of Computer Science, Vienna University of Technology, November 2004.

[35] M. E. Lübbecke and J. Desrosiers. Selected topics in column generation. *Operations Research*, 53:1007–1023, 2002.

[36] D. Maltoni, D. Maio, A. K. Jain, and S. Prabhakar. *Handbook of fingerprint recognition*. Springer, 2003.

[37] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2004.

[38] A. Moser. Finding Provably Optimal Solutions for the (Prize Collecting) Steiner Tree Problem. Master's thesis.

[39] J. Nummela and B. A. Julstrom. An effective genetic algorithm for the minimum-label spanning tree problem. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 553–558, New York, NY, USA, 2006. ACM.

[40] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization; Algorithms and Complexity*. Dover Publications, 1998.

[41] G. R. Raidl and A. Chwatal. Fingerprint template compression by solving a minimum label k-node subtree problem. In *Numerical Analysis and Applied Mathematics, volume 936 of AIP Conference Proceedings*, volume 936, pages 444–447. American Institute of Physics, 2007.

[42] I. Rechenberg. *Evolutionsstrategie: Optimierung technisher Systeme nach Prinzipien. der biologischen Evolution*. Frommann-Holzboog, 1973.

[43] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers Inc., USA, third edition, 2006.

[44] H.-P. Schwefel. *Numerische Optimierung von Computer-modellen mittels der Evolutionsstrategie*. Birkhäuser Verlag, 1981.

[45] M. R. R. Sunil Chopra. Edgar R. Gorres. Solving the steiner tree problem on a graph using branch and cut. *ORSA J COMPUT*, 4:320–335, 1992.

[46] C. Thöni. Compressing fingerprint templates by solving the k-node minimum label spanning arborescence problem by branch-and-price. Diploma thesis, Vienna University of Technology, Institute of Computer Graphics and Algorithms, 2009.

[47] V. V. Vazirani. *Approximation Algorithms*. Springer, 2004.

[48] Y. Wan, G. Chert, and Y. Xu. A note on the minimum label spanning tree. *Inf. Process. Lett.*, 84(2):99–101, 2002.

[49] L. A. Wolsey. *Integer Programming*. Wiley-Interscience, September 1998.

[50] L. A. Wolsey and G. L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience, November 1999.

[51] Y. Xiong, B. Golden, and E. Wasil. Worst-case behavior of the mvca heuristic for the minimum labeling spanning tree problem. *Operations Research Letters*, 33(1):77 – 80, 2005.

[52] Y. Xiong, B. L. Golden, and E. A. Wasil. A one-parameter genetic algorithm for the minimum labeling spanning tree problem. *IEEE Trans. Evolutionary Computation*, 9(1):55–60, 2005.

[53] Y. Xiong, B. L. Golden, and E. A. Wasil. Improved heuristics for the minimum label spanning tree problem. *IEEE Trans. Evolutionary Computation*, 10(6):700–703, 2006.