Efficient Implementation of Dynamic Programming with Representative Sets

Master of Science Graduation Thesis

Stefan Fafianie

November 11, 2013

Supervisors:

dr. H.L. Bodlaender

dr. G. Tel

dr. J. Nederlof

Department of Computing Science,
Utrecht University,
Utrecht, The Netherlands

Preface

This is my thesis for the conclusion of my master program Computing Science at the Utrecht University. During my two years of study I have specialized in algorithm design and complexity. At the very start of my master phase I have attended the course Algorithms & Networks thaught by Hans Bodlaender. This course further ignited a pre-existing interest in algorithms and complexity resulting from my bachelor program studies.

At the start of this year I have completed an Experimentation Project as part of the master track under supervision of Hans Bodlaender and with support by Jesper Nederlof. In this project I have performed an experimental evaluation of a recent result in which the powerful notion of representative sets is used to speed up dynamic programming algorithms. The results of this evaluation where very positive and we have submitted a paper to the 8th International Symposium on Parameterized and Exact Computation (IPEC 2013) which I was also fortunate enough to attend.

This thesis is the result of further study into the implementation of dynamic programming algorithms using representative sets. The first contribution is the result of a successful effort to improve a computational step that was observed to be expensive in practice during the aforementioned evaluation. This first part of the thesis is presented in an extended version of the IPEC 2013 paper in which the algorithms described in the previous version of the paper are discussed in some more detail and the new improvement is described and evaluated. In the second part of this thesis a method is discussed that is alternative to the usual bottom-up calculation of tables in dynamic programming algorithms.

The past year in which I have worked on the experimentation project and master thesis has been exhilarating and I would like to thank both Hans and Jesper for their unyielding support.

Abstract

In the rank based approach introduced by Bodlaender et al. [5] the notion of representative sets is applied to dynamic programming algorithms for connectivity problems parameterized by treewidth. In this approach the tables in which partial solutions are stored are intermittently reduced in size. This results in a speed-up of these algorithms yielding running times single exponential in treewidth. We build this thesis upon earlier work, given an experimental study of these algorithms. In particular, in [13], we compared the performance of a classic dynamic programming algorithm for Steiner Tree with the performance of its counterpart in which the rank-based approach is applied.

The first contribution of the thesis is an alternative representation of the partial solutions generated during the dynamic programming, with which we can eliminate a computational step of the reductions that is expensive in practice. We discuss the adaption of operators introduced in the framework by Bodlaender et al. in order to apply them to this new representation. This allows us to use the new representation for any of the connectivity problems for which a dynamic programming algorithm can be described using these operators. In an experimental evaluation for Steiner Tree we find that this representation yields very positive results.

The second contribution of the thesis is an algorithm which calculates partial solutions in order of optimality instead of calculating entire tables at a time. We do this in order to avoid calculating partial solutions that do not contribute to the optimal solution of the full problem. We give a detailed description of this algorithm for Steiner Tree and perform an experimental evaluation. We find some mixed results, where the algorithm performs well in a few instances and badly in others.

Contents

Chapter 1

Introduction

1.1 Contributions of this Thesis

In computational complexity theory it is assumed that a polynomial time algorithm with which we can solve a problem that is classified as $\mathcal{NP}$-hard can not be found. One way to deal with some of these problems is the study of fixed parameter tractable algorithms. Such an algorithm spends time exponential in the size of a fixed, preferably small, parameter of the input and polynomial in the size of the input.

For problems on graphs, treewidth can be a very useful parameter. The treewidth of a graph measures how much it resembles a tree. This parameter is typically used in dynamic programming algorithms in which the input graph is first transformed into a tree decomposition. These algorithms then calculate tables of partial solution for each node in the decomposion in a bottom-up fashion. The partial solutions in a table represent the choices that where made in the subgraph induced by the decendents of a node. Ultimately, a full solution for the graph problem can be found in the table calculated at the root of the tree decomposition.

For many local graph problems, e.g. Vertex Cover, Dominating Set, exact deterministic algorithms running in single exponential time in treewidth are known. However, such algorithms have only recently been found for connectivity problems, e.g. Steiner Tree, Traveling Salesman by Bodlaender et al. [5]. The rank based approach described in their paper applies the notion of representative sets in order to intermittently reduce the size of tables calculated during dynamic programming.

In an experimental evaluation [13] we compared the performance of the classic dynamic programming algorithm for Steiner Tree on tree decompositions to its counterpart in which the rank based approach is applied to reduce the size of tables. During this evaluation we found that the rank based approach works very well in practice, giving significant speed-ups even for instances of relatively small treewidth. Building upon this earlier work, we further explore the implementation of dynamic programming algorithms with representative sets.

The first contribution of this thesis is the direct result of an unanticipated observation during the aforementioned experiments. In the reduction steps of the rank based approach a special matrix is used in which rows correspond to partial solutions. A representative table of partial solutions is found by finding a basis of minimum weight in this matrix using Gaussian elimination. While the asymptotic worst case running time of the Gaussian elimination step is the bottleneck of the reduction step in theory, we have observed that in practice a significant portion of the time is spent calculating the entries of these special matrices. Inspired by this observation, we have designed a version of the algorithms in which we identify partial solutions directly by their row elements of its matrix. This new representation of partial solutions in which we use weighted bit strings allows us to calculate the matrices for parent nodes directly from rows in matrices obtained from child nodes. We show that we can use this representation for any of the connectivity problems that may defined using the framework presented in [5]. We then perform an experimental evaluation in which the performance of algorithms using the original and new representation are compared.

The second contribution of this thesis is an algorithm which evaluates partial solutions computed during straightforward dynamic programming in order of optimality. We do this in order to avoid calculating partial solutions that do not contribute to the optimal solution of the full problem. Instead of calcuting tables in a bottom-up fashion we allow each node in the tree decomposition to request partial solutions from its children in order of optimality. It can then use these to calculate partial solutions for its parent. Once a solution is found at the root node we can terminate the algorithm since this solution is optimal. We give a detailed description for Steiner Tree and provide an experimental evaluation.

1.2 Thesis Overview

The results of the experimental evaluation peformed in [13] where to submitted in a paper to the 8th International Symposium on Parameterized and Exact Computation (IPEC 2013). This paper was co-authored by Hans L. Bodlander and Jesper Nederlof. The first contribution of the thesis is presented in an extended version of this paper which we will submit to the special issue of Algorithmica dedicated to IPEC 2013. We give an introduction to the rank based approach and provide a detailed description of the weighted bit string representation of partial solutions. We then provide implementation details and show results of the experimental evaluation. The contents of this extended paper are presented in Chapter 2.

We present the second contribution of the thesis in Chapter 3. We show a detailed description of the algorithm for Steiner Tree in which partial solutions are calculated in ascending order of weight. We then provide some pointers for the implementation and discuss results of the experimental evaluation for this algorithm.

Chapter 2

Speeding Up Dynamic Programming with Representative Sets

2.1 Introduction

The notion of treewidth provides us with a method of solving many $\mathcal{NP}$-hard problems by means of dynamic programming algorithms on tree decompositions of graphs, resulting in algorithmic solutions which are fixed-parameter tractable in the treewidth of the input graph. For many problems, this gives algorithms that are linear in the number of vertices n, but at least exponential in the width of the tree decomposition on which the dynamic programming algorithm is executed. The dependency of the running time on the width of the tree decomposition has been a point of several investigations. For many problems, algorithms were known whose running time is single exponential on the width, see e.g., [25]. A recent breakthrough was obtained by Cygan et al. [11] who showed for several *connectivity* problems, including Hamiltonian Circuit, Steiner Tree, Connected Dominating Set (and many other problems) that these can be solved in time, single exponential in the width, but at the cost of introducing randomization and an additional factor in the running time that is polynomial in n. Very recently, Bodlaender et al. [5] introduced a new technique (termed the *rank based approach*) that allows algorithms for connectivity problems that are (i) deterministic, (ii) can handle weighted vertices, and (iii) have a running time of the type $O(c^k n)$ for graphs with a given tree decomposition of width k and n vertices, i.e., the running time is single exponential in the width, and linear in the number of vertices.

The main ideas of the rank based approach are the following. (Many details are abstracted away in the discussion below. See [5] for more details.) Suppose we store during dynamic programming a table T with each entry giving the characteristic of a partial solution. If we have an entry s in T, such that for each extension t of s to a 'full solution', $s \cdot t$, there is an other entry $s' \neq s$ in T, that can be extended in the same way to a full solution $s' \cdot t$, and solution $s' \cdot t$ has a value that is as least as good as the value of $s \cdot t$, then s is not needed for obtaining an optimal solution, and we can delete s from T. This idea leads to the notion of *representativity*, pioneered by Monien in 1985 [23]. Consider the matrix M with rows indexed by partial solutions, and columns indexed by manners to extend partial solutions, with a 1 if the combination gives a full solution, and a 0 otherwise. A table T corresponds to a set of rows in M, with a value associated to each row. (E.g., for the Steiner Tree problem, a row corresponds to the characteristic of a forest in a subgraph, and the value is the sum of the edges in the forest.) It is not hard to see that a maximal subset of linear independent rows of minimal cost (in case of minimization problems, and of maximal value in case of maximization problems) forms a representative set. Now, if we have an explicit basis of M (the characteristics of the columns in a maximal set of independent columns in T) and M has 'small' rank, then we can find a 'small' representative set efficiently, just by performing Gaussian elimination on a submatrix of M. Now, for many connectivity problems, including Steiner Tree, Feedback Vertex Set, Long Path, Hamiltonian Circuit, Connected Dominating Set, the rank of this matrix M when solving these problems on a tree decomposition is single exponential in the width of the current bag. This leads to the improved dynamic programming algorithm: interleave the steps of the existing DP algorithm with computing representative sets by computing the submatrix of M and then carrying out Gaussian elimination on this submatrix.

The notion of representative sets was pioneered by Monien in 1985 [23]. Using the well known two families theorem by Lovász [21], it is possible to obtain efficient FPT algorithms for several other problems [22, 14]. Cygan et al. [10] give an improved bound on the rank as a function of the width of the tree decomposition for problems on finding cycles and paths in graphs of small treewidth, including TSP, Hamiltonian Circuit, Long Path.

In this paper, we perform an *experimental evaluation* of the rank based approach, targeted at the Steiner Tree problem, i.e., we discuss an implementation of the algorithm, described by Bodlaender et al. [5] for the Steiner Tree problem and its performance. We test the algorithm on a number of graphs from a benchmark for Steiner Tree, and some randomly

generated graphs. The results of our experiments are very positive: the new algorithm is considerably faster compared to the classic dynamic programming algorithm, i.e., the time that is needed to reduce the tables with help of Gaussian elimination is significantly smaller than the gain in time caused by the fact that tables are much smaller. Furthermore, we propose an alternative representation for partial solutions using weighted bit strings. This allows us to avoid a computational step in the table reductions that is expensive in practice. Again, the experimental evaluation of this bit string representation shows very positive results.

The Steiner Tree problem (of which Minimum Spanning Tree is a special case) is a classic $\mathcal{NP}$-hard problem which was one of Karp's original 21 $\mathcal{NP}$-complete problems [17]. Extensive overviews on this problem and algorithms for it can be found in [16, 30]. Applications of Steiner Tree include electronic design automation, very large scale integration (VSLI) of circuits and wire routing. In this paper we consider the weighted variant, i.e., edges have a weight, and we want to find a Steiner tree of minimum weight. It is well known that Steiner Tree can be solved in linear time for graphs of bounded treewidth. In 1983, Wald and Colbourn [27] showed this for graphs of treewidth two. For larger fixed values of k, polynomial time algorithms are obtained as consequence of a general characterization by Bodlaender [4] and linear time algorithms are obtained as consequence of extensions of Courcelles theorem, by Arnborg et al. [2] and Borie et al. [7]. In 1990, Korach and Solel [20] gave an explicit linear time algorithm for Steiner Tree on graphs of bounded treewidth. Inspection shows that the running time of this algorithm is $O(2^{O(k \log k)}n)$; k denotes the width of the tree decomposition. We call this algorithm the *classic* algorithm. Recently, Chimani et al. [8] gave an improved algorithm for Steiner Tree on tree decompositions that uses $O(B_{k+1}^2 \cdot k \cdot n)$ time, where the Bell number B_i denotes the number of partitions of an i element set. Our description of the classic algorithm departs somewhat from the description in Korach and Solel [20], but the underlying technique is essentially the same. We have chosen not to use the coloring schemes from Chimani et al. [8], but instead use hash tables to store information. While the coloring schemes give a better worst case running time, we also spend time with these on 'non-existing table entries', and thus we expect faster computations when using hash tables. Wei-Kleiner [29] gives a tree decomposition based algorithm for Steiner tree, that particularly aims at instances with a small set of Steiner vertices.

In this paper, we compare five different algorithms:

- The *classic* dynamic programming algorithm (CDP), see the discussion above. On a nice tree decomposition, we build for each node i a table. Tables map partitions of subsets of X_i to values, characterizing the minimum weight of a 'partial solution' that has this partition of a subset as 'fingerprint'.

- RBA: To the classic dynamic programming algorithm, we add a step where we apply the *reduce* algorithm from [5]. With help of Gaussian elimination on a specific matrix (with rows corresponding to entries in the DP table, columns corresponding to a 'basis of the fingerprints of ways of extending partial solutions to Steiner trees', and values 1, if the extension of the column applied to the entry of the row gives a Steiner tree and 0 otherwise), we delete some entries from the table. It can be shown that deleted entries are not needed to obtain an optimal solution, i.e., the step does not affect optimality of the solution. This elimination step is performed each time after the DP algorithm has computed a table for a node of the nice tree decomposition.

- RBC: Similar to RBA, but now the elimination step is only performed for 'large' tables, i.e., tables where the theory tells us that we will delete at least one entry when we perform the elimination step.

- BSA: Similar to RBA, but here we use a weighted bit string representation for partial solutions. These bit strings directly represent the rows of the matrix on which Gaussian

elimination is applied during the reduction step. The entries in this matrix are thus acquired implicitly during the building of tables in the dynamic programming algorithm.

- BSC: Similar to BSA, but again the elimination step is only performed for 'large' tables.

Our software is publicly available, can be used under a GNU Lesser General Public Licence, and can be downloaded at:
http://www.staff.science.uu.nl/~bodla101/java/steiner.zip

This paper is organized as follows. Some preliminary definitions are given in Section 2.2. In Section 2.3, we briefly describe both the classic dynamic programming algorithm for Steiner Tree on nice tree decompositions, as well as the improvement with the rank based approach as presented in [5]. We then show how the operators used to define dynamic programming formulations in [5] can be applied to sets of weighted bit strings as opposed to sets of weighted partitions. In Section 2.4, we describe the setup of our experiments, and in Section 2.5, we discuss the results of the experiments. Some final conclusions are given in Section 2.6.

2.2 Preliminaries

We use standard graph theory notation and quite some additional notation from [5]. For a subset of edges $X \subseteq E$ of an undirected graph $G = (V, E)$, we let $G[X]$ denote the subgraph induced by edges and endpoints of X, i.e. $G[X] = (V(X), X)$.

For two partitions p and q of a set W, we say that p is a coarsening of q (or, q is a refinement of p) if every block of q is contained in a block of p, and we let $p \sqcap q$ denote the finest partition that is a coarsening of p and of q. (In graph terms: take an edge between $v \in W$ and $w \in W$ iff $v \neq w$ and v and w belong to the same block in p or to the same block in q. Now, the classes of $p \sqcap q$ are the connected components of this graph.)

The Steiner Tree problem can be defined as follows.

Steiner Tree
Input: A graph $G = (V, E)$, weight function $\omega : E \to \mathbb{N} \setminus \{0\}$, a terminal set $K \subseteq V$ and a nice tree decomposition $\mathbb{T}$ of G of width $\mathtt{tw}$.
Question: The minimum of $\omega(X)$ over all subsets $X \subseteq E$ of G such that $G[X]$ is connected and $K \subseteq V(G[X])$.

Definition 1 (Tree decomposition, [24])**.** *A* tree decomposition *of a graph G is a tree $\mathbb{T}$ in which each node x has an assigned set of vertices $B_x \subseteq V$ (called a* bag*) such that $\bigcup_{x \in \mathbb{T}} B_x = V$ with the following properties:*

- *for any $e = (u, v) \in E$, there exists an $x \in \mathbb{T}$ such that $u, v \in B_x$.*
- *if $v \in B_x$ and $v \in B_y$, then $v \in B_z$ for all z on the (unique) path from x to y in $\mathbb{T}$.*

The treewidth $tw(\mathbb{T})$ of a tree decomposition $\mathbb{T}$ is the size of the largest bag of $\mathbb{T}$ minus one, and the treewidth of a graph G is the minimum treewidth over all possible tree decompositions of G.

Definition 2 (Nice tree decomposition)**.** *A* nice tree decomposition *is a tree decomposition with one special bag z called the* root *and in which each bag is one of the following types:*

- leaf bag*: a leaf x of $\mathbb{T}$ with $B_x = \emptyset$.*
- introduce vertex bag*: an internal vertex x of $\mathbb{T}$ with one child vertex y for which $B_x = B_y \cup \{v\}$ for some $v \notin B_y$. This bag is said to* introduce *v.*
- introduce edge bag*: an internal vertex x of $\mathbb{T}$ labelled with an edge $e = (u, v) \in E$ with one child bag y for which $u, v \in B_x = B_y$. This bag is said to* introduce *e.*

- forget vertex bag*: an internal vertex x of $\mathbb{T}$ with one child bag y for which $B_x = B_y \setminus \{v\}$ for some $v \in B_y$. This bag is said to* forget v.
- join bag*: an internal vertex x with two child vertices y and y' with $B_x = B_y = B_{y'}$.*

We additionally require that every edge in E is introduced exactly once.

Nice tree decompositions were introduced in the 1990s by Kloks [18]. We use here a more recent version that distinguishes *introduce edge* and *introduce vertex* bags [11]. To each bag x we associate the graph $G_x = (V_x, E_x)$, with V_x the union of all B_y with $y = x$ or y a descendant of x, and E_x the set of all edges introduced at bags y with $y = x$ or y a descendant of x. There are also many heuristics for finding a tree decomposition of small width; see [6] for a recent overview. Given a tree decomposition $\mathbb{T}$ of G, a nice tree decomposition rooted at a forget bag can be computed in $n \cdot \mathtt{tw}^{\mathcal{O}(1)}$ time by following the arguments given in [18], with the following modification: between a forget bag X_i where we 'forget vertex v' and its child bag X_j, we add a series of introduce edge bags for each edge $e = \{v, w\} \in E$ and $w \in X_j$. We also assume that root bag z is a forget node with $B_x = \emptyset$ and that the vertex that is forgotten at the root bag is a terminal.

A collection of operators on sets of weighted partitions is presented in [5]. It is shown that we can apply the rank based approach to any dynamic programming algorithm that can be formulated using these operators and maintain correctness. Let $\Pi(U)$ denote the set of all partitions of some set U. Let $\mathcal{A} \subseteq \Pi(U) \times \mathbb{N}$ denote a *set of weighted partitions*, i.e. pairs $(p, w) \in \mathcal{A}$ consist of a partition p of U and a non-negative integer weight w. The operators are then defined as follows.

Definition 3 (Operators on sets of weighted partitions)**.**

- **Union.** *For $\mathcal{B} \subseteq \Pi(U) \times \mathbb{N}$, define $\mathcal{A} \uplus \mathcal{B} = \mathtt{rmc}(\mathcal{A} \cup \mathcal{B})$.* Combine two sets of weighted partitions and discard dominated partitions.
- **Insert.** *For $X \cap U = \emptyset$, define $\mathtt{ins}(X, \mathcal{A}) = \{(p_{\uparrow U \cup X}, w) | (p, w) \in \mathcal{A}\}$.* Insert additional elements into U and add them as singletons in each partition.
- **Shift.** *For $w' \in \mathbb{N}$ define $\mathtt{shft}(w', \mathcal{A}) = \{(p, w + w') | (p, w) \in \mathcal{A}\}$.* Increase the weight of each partition by w'.
- **Glue.** *For u, v, let $\hat{U} = U \cup \{u, v\}$ and define $\mathtt{glue}(uv, \mathcal{A}) \subseteq \Pi(\hat{U}) \times \mathbb{N}$ as*

$$\mathtt{glue}(uv, \mathcal{A}) = \mathtt{rmc}(\Big\{ (\hat{U}[uv] \sqcap p_{\uparrow \hat{U}}, w) \Big| (p, w) \in \mathcal{A} \Big\}).$$

  Also, if $\omega : \hat{U} \times \hat{U} \to \mathbb{N}$, let $\mathtt{glue}_\omega(uv, \mathcal{A}) = \mathtt{shft}(\omega(u, v), \mathtt{glue}(uv, \mathcal{A}))$. In each partition combine the sets containing u and v into one; add u and v to the base set if needed.
- **Project.** *For $X \subseteq U$ let $\overline{X} = U \setminus X$, and define $\mathtt{proj}(X, \mathcal{A}) \subseteq \Pi(\overline{X}) \times \mathbb{N}$ as*

$$\mathtt{proj}(X, \mathcal{A}) = \mathtt{rmc}(\Big\{ (p_{\downarrow \overline{X}}, w) \Big| (p, w) \in \mathcal{A} \wedge \forall e \in X : \exists e' \in \overline{X} : p \sqsubseteq U[ee'] \Big\}).$$

  Remove all elements of X from each partition, but discard partitions where this would reduce the number of blocks/sets.
- **Join.** *For $\mathcal{B} \subseteq \Pi(U') \times \mathbb{N}$ let $\hat{U} = U \cup U'$ and define $\mathtt{join}(\mathcal{A}, \mathcal{B}) \subseteq \Pi(\hat{U}) \times \mathbb{N}$ as*

$$\mathtt{join}(\mathcal{A}, \mathcal{B}) = \mathtt{rmc}(\Big\{ (p_{\uparrow \hat{U}} \sqcap q_{\uparrow \hat{U}}, w_1 + w_2) \Big| (p, w_1) \in \mathcal{A} \wedge (q, w_2) \in \mathcal{B} \Big\}).$$

  Extend all partitions to the same base set. For each pair of partitions return the outcome of the meet operation $\sqcap$, with weight equal to the sum of the weights.

Here $\mathtt{rmc}(\mathcal{A}) = \{(p, w) \in \mathcal{A} | \nexists (p, w') \in \mathcal{A} \wedge w' < w\}$ denotes the set obtained by removing non-minimal weight copies. The partition that is the same as p but with sets containing a and b merged is obtained by $p \sqcap U[ab]$ and $p \sqsubseteq U[ab]$ is true when a and b are in the same set in p.

2.3 Dynamic Programming Algorithms for Steiner Tree Parameterized by Treewidth

In this section we briefly sketch both the classic dynamic programming algorithm on (nice) tree decompositions for (the edge weighted version of) Steiner Tree and its variant with the rank based approach. Finally we will show how the operators from [5] can be modified to work on weighted bit strings. We will not present full proofs of correctness for the algorithms but will show the fundemantal intuition behind these proofs.

2.3.1 Classic Dynamic Programming

The classic dynamic programming algorithm computes for each bag x a function A_x. This function is represented by a table, with only trivial entries (e.g., partitions mapping to infinity, as there are no forests corresponding to the partition) not stored.

The function A_x maps a subset $W \subseteq B_x$ to a collection of pairs. Each pair consists of a partition p of W and a weight w. If (p, w) is in the collection associated to W, then w is the minimum weight over all forests F in G_x with the following properties: (1) For all $v \in B_x$, $v \in W$ iff v belongs to F; (2) each terminal in V_x belongs to F; (3) each tree in F contains at least one vertex in W; and (4) two vertices in W belong to the same class in the partition p, iff they belong to the same tree in F. I.e., for each partition p, we store at most one weight; if for set W and a partition p, no forest exists that fulfills the properties, then we have no pair in $A_x(W)$ of the form $(p, \ldots)$.

In the full paper [13] we can find a more formal description with slightly different notation (based on the notation in [5]), and recurrences for A for each of the types (leaf, introduce vertex, introduce edge, join, forget) of nodes in nice tree decompositions. As an example let us consider an introduce edge $e = uv$ node x with child node y. For each partial solution in the child table we can choose whether or not to use the edge. Given the child table A_y we can build table A_x using the following recurrence.

$$A_x(W) = \begin{cases} A_y(W) & \text{if } u \notin W \vee v \notin W \\ A_y(W) \uplus \texttt{glue}_{\omega}(uv, A_y(W)) & \text{otherwise.} \end{cases}$$

We include each partial solution from A_y in A_x and where possible we include partial solutions from A_y modified to use the edge. This may yield multiple solutions with the same partition, in which case we keep the one with minimum weight. In bottom-up order, we compute for each node x in the nice tree decomposition a table for A_x. The minimum value of a Steiner tree in G can be directly observed given the table for the root node.

In our implementation, we use two levels of hash tables: one with keys the different subsets W of B_x, and for each W with at least one partial solution, we have a hash table storing for each p the value z of the pair $(p, z) \in A_x(W)$, in case such a pair exists.

2.3.2 Rank Based Table Reductions

The main idea of the rank based approach from [5] is that after we have computed a table for a bag x in the nice tree decomposition, we can carry out a reduction step and possibly remove a number of entries from the table without affecting optimality. A table is transformed thus to a (possibly smaller) table whose weighted partitions are *representative* for the collection of weighted partitions in the earlier table. Let us consider partitions p representing the connectivity of partial solutions in the subgraph G_x induced by edges introduced in descendents of bag x. Then for partitions q that are extensions, i.e. in the case of Steiner Tree, forests containing the other terminals using edges not in G_x and some vertex in bag x for each tree, we have a full solution if p and q together form a single connected component, $p \sqcap q = U$. Since we do not explicitly know all extensions q, the goal is to find representative sets $A'_x(W)$ where for each

pair $(p, w) \in A_x(W)$, for any partition q on W, if $p \sqcap q = W$ then there is a $(p', w') \in A'_x(W)$ such that $p' \sqcap q = W \wedge w' \leq w$. This ensures that if a partial solution can be extended to a full solution, then a partial solution in the representative table can be extended to a full solution of at most the same weight in the same way, thus maintaining optimality.

For each $W \subseteq B_x$ we consider a matrix $\mathcal{M}$ with a row for each partition p appearing in a pair in $A_x(W)$, and a column for each partition q of W, with $\mathcal{M}(p, q) = 1$ if and only if $p \sqcap q = W$. In order to get a good rank bound the matrix $\mathcal{M}$ can be written as the product of two cutmatrices $\mathcal{C}$ which are defined as follows. Let $\texttt{cuts}(W) := \{(V_1, V_2) | V_1 \cup V_2 = W \wedge w \in V_1\}$ for some arbitrary fixed $w \in W$. Then $\mathcal{C}$ is a matrix with a row for each partition p and with a column for each cut in $\texttt{cuts}(W)$, with $\mathcal{C}[p, (V_1, V_2) = 1$ if $(V_1, V_2) \sqsubseteq q$. For $z = p \sqcap q$ we have $(V_1, V_2) \sqsubseteq z$ if and only if $(V_1, V_2) \sqsubseteq p \wedge (V_1, V_2) \sqsubseteq q$. Furthermore, the cuts that z is a refinement of can be enumerated by fixing an arbitrary block from z in V_1, and then choosing for every other block in z whether to include it in V_1 or in V_2. Then z is a refinement of $2^{\#\texttt{blocks}(z)-1}$ cuts, which is odd when $z = W$ and even if z consists of multiple blocks. The product $\mathcal{C}\mathcal{C}^T$ counts the number of cuts of which both p and q are a refinement. Therefore, it follows that $\mathcal{M} \equiv \mathcal{C}\mathcal{C}^T$ in arithmetic modulo two.

Now, from [5], it follows that it is sufficient to keep a minimum weight basis of rows in $\mathcal{C}$. The rank of $\mathcal{C}$ is bounded by the number of cuts of W, i.e. $2^{|W|-1}$. The reduction step is performed as follows: for each $W \subseteq B_x$, we calculate cutmatrix $\mathcal{C}$. With help of Gaussian elimination, we compute such a minimum weight basis (after first sorting the rows with respect to their weights), and then delete all other entries from the table. Correctness follows from the analysis in [5]. In our experiments, we consider both the case where we always apply the reduction step, and the case where we only apply it when $|A_x(W)| \geq 2^{|W|-1}$. Both cases give the same guarantees on the size of tables and worst case upper bound on the running time, but the actual running times in experiments differ, as we discuss in later sections.

2.3.3 Representing partial solutions with weighted bit strings.

When we first performed our experimental evaluation [13] we have found that during the reduction steps most time is spent calculating the entries of cutmatrices. While the asymptotic worst case running time of the Gaussian elimination step dominates this time for the calculation of the cut matrices, in our experiments, we observed that the actual time for the latter is significantly larger than the actual time for Gaussian elimination. Inspired by this observation, we designed a version of the algorithms where we avoid most of the work to compute the entries of the cutmatrices. More precisely, we *identify* partial solutions not with help of a partitions, but directly by the row elements of its cut matrix. The partition thus is implictly represented by this row. This new representation allows us to calculate rows in cutmatrices for parent nodes directly from rows in cutmatrices obtained from child nodes.

We will now formally introduce the weighted bit string representation for partial solutions. For each of the operators used in the framework introduced by Bodleander et al. [5] we show an adaptation for weighted bit strings. The effects that each of these operators have for partial solutions on entries in a cutmatrix should now be captured directly as manipulations on these weighted bit strings. Thus, we show that this alternative representation can be used for any of the connectivity problems presented in [5], as well as any other connectivity problem that can be represented with recurrences using these operators.

Let $(s, w) \in A_x(W)$ be pair consisting of a bit string s directly representing a row in a cutmatrix and w be its weight. Let $l(s) = 2^{|W|-1}$ denote the length of this bit string and let $s_i \in \{0, 1\}$ denote the value of the bit at index $i \in \{0..l-1\}$. In order to capture the effects that the operators have on this bit string we should first make a strict assumption about which specific cut corresponds to entry s_i. Without loss of generality let us assume an arbitrary fixed ordering $W = \{v_0, \ldots, v_{|W|-1}\}$ of vertices in W. Now let $\texttt{cuts}(W) = \{c_0, \ldots, c_{l-1}\}$ be cuts corresponding to index i in the bit string. Intuitively, at some point during the dynamic

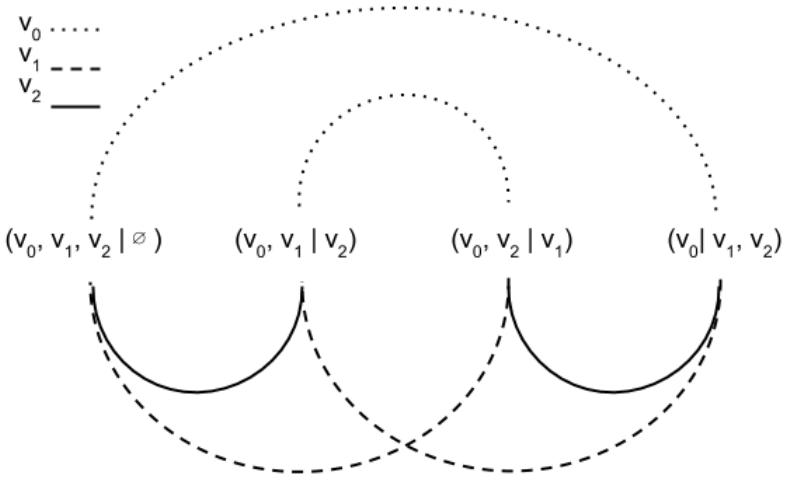

Figure 2.1: Emerging pattern in ordered cuts. The edges depict cuts that are identical if the corresponding vertex is left out.

programming algorithm we have a set $W = \{v_0\}$ where $\texttt{cuts}(W)$ contains a single cut $(v_0|\emptyset)$. This set of cuts is gradually expanded when introducing other vertices by fixing the new vertex to the left- and righthand side of the cuts represented by columns in the previous table, i.e.

$$\texttt{cuts}(\{v_0\}) = \{(v_0|\emptyset)\}$$
$$\texttt{cuts}(\{v_0, v_1\}) = \{(v_0, v_1|\emptyset), (v_0|v_1)\}$$
$$\texttt{cuts}(\{v_0, v_1, v_2\}) = \{(v_0, v_1, v_2|\emptyset), (v_0, v_1|v_2), (v_0, v_2|v_1), (v_0|v_1, v_2)\}$$
$$\texttt{etc.}$$

As an invariant we will assume that for any given pair $(s, w) \in A_x(W)$ the indices of s correspond to cuts ordered this way. We can now proceed with the adaption of the operators on sets of weighted partitions (see Section 2.2). First let us trivially adapt the definition of $\texttt{rmc}(\mathcal{A})$ and the union operator where $\mathcal{A}$ is now a set of weighted bit strings.

$$\texttt{rmc}(A) = \left\{(s, w) \in A \middle| \nexists (s, w') \in A \wedge w' < w\right\}$$

- **Union.** *For a table of weighted bit strings $\mathcal{B}$, define* $\mathcal{A} \Cup \mathcal{B} = \texttt{rmc}(\mathcal{A} \cup \mathcal{B})$. Combine two sets of weighted bit strings and discard dominated bit strings.

The insert operator is more involved. Suppose we have a bit string s based on cuts of set W and extend this set with a single vertex v, i.e. $W' = W \cup \{v\}$. We then want to capture the effect of adding this vertex as singleton in our partial solution. The resulting bit string s' will have length $l(s') = 2 \cdot l(s)$ since we have twice as many cuts. If we have a cut $(V_1, V_2) \in \texttt{cuts}(W)$ where $V_1 \cup V_2 = W$ then $(V_1 \cup \{v\}, V_2), (V_1, V_2 \cup \{v\}) \in \texttt{cuts}(W')$. If a partial solution is a refinement of the old cut then it must be a refinement of the two new cuts once we add a vertex as singleton since no change in connectivity is introduced. Likewise, if a partial solution is not a refinement of the old cut then it cannot be a refinement of the new cuts when we add a vertex as singleton.

When we have a bit s_i we are left with finding the position for two copies of this bit in s' such that the invariant holds. Suppose $v_j \in W'$ is the inserted vertex. Then we need the position of cuts $(V_1 \cup \{v_j\}, V_2), (V_1, V_2 \cup \{v_j\}) \in \texttt{cuts}(W')$. If $v_j \neq v_0$ then according to our invariant we have pairs of cuts that are next to each other in $\text{cuts}(v_0, \ldots, v_j)$ which are identical except for

the side on which v_j is fixed. When we expand cuts$(v_0, \ldots, v_j)$ to cuts(W') these pairs are at a distance of $d = 2^{|W'|-1-w_j}$ appart since we expand for $|W'| - 1 - w_j$ more vertices, each time fixing a vertex left or right. These pairs are packed in blocks of size $b = 2^{|W'|-w_j}$ (see Fig. 1). We calculate the new bit string by iterating over indices i of string s. The block containing the new bits corresponding to s_i starts at index $p \cdot b$ where $p = i/d$ indicates in which of the blocks we are currently working. Note that we use integer division for p. In this block we find the first bit after $k = i \mod d$ more indices and the second bit d indices later. So we have the following.

- *For a single element $v_j \in W' = W \cup v_j$ where $v_j \neq v_0$, define*
  $\texttt{ins}(v_j, \mathcal{A}) = \left\{(s', w) \middle| (s, w) \in \mathcal{A} \wedge s'_{p \cdot b+k} = s'_{p \cdot b+k+d} = s'_i\right\}$ where
  $b = 2^{|W'|-w_j}, d = \frac{b}{2}, p = i/d$ and $k = i \mod d$, $\forall i \in \{0..l(s) - 1\}$

In the case that $v_j = v_0$ we have pairs of cuts that are identical except for the side on which v_0 is fixed. These cuts are pushed to opposite sides at every expansion since cuts$(\{v_0\}) = (v_0|\emptyset)$ starts out asymetrically (see Fig. 2.1), i.e.

- *For a single element $v_j \in W' = W \cup v_j$ where $v_j = v_0$, define*
  $\texttt{ins}(v_j, \mathcal{A}) = \left\{(s', w) \middle| (s, w) \in \mathcal{A} \wedge s'_i = s'_{l(s')-i-1} = s_i\right\}$, $\forall i \in \{0..l(s) - 1\}$

We now have an adaptation of the insert operator for bit strings where we insert a single vertex. Finally, in order to insert a set of vertices we can insert them one at a time, i.e.

- **Insert.** *For $X \cap W = \emptyset$ and $x \in X$, define*
  $\texttt{ins}(X, \mathcal{A}) = \left\{\texttt{ins}(X \backslash x, \texttt{ins}(x, \mathcal{A})\right\}$

The project operator is somewhat similar, but here the length of a bit string decreases by half. In this case, if we project for a single vertex v, we have pairs of bits corresponding to $(V_1 \cup \{v\}, V_2), (V_1, V_2 \cup \{v\}) \in \texttt{cuts}(W)$ and end up with a single bit corresponding to $(V_1, V_2) \in \texttt{cuts}(W')$ where $W' = W \cup \{v\}$. Now, if a partial solution is a refinement of either of the old cuts then it must be a refinement of the new cut since connectivity with v is lost. Likewise, if a partial solution is a refinement of neither of the old cuts then it cannot be a refinement of the new cut since there must be some other connectivity between vertices in V_1 and vertices in V_2. Now we must make sure that the partial solution is removed if removing v would have reduced the number of blocks in the original partition. We can do this by finding out if v is a singleton, which we can achieve by checking if the partial solution is a refinement of the cut $(W \backslash v|v)$. Suppose we project $v_j \in W$. Assuming our invariant holds we can find the bit corresponding to this particular cut at index $2^{|W|-v_j-1}$ if $v_j \neq v_0$ and at index $l(s) - 1$ otherwise. The project operator for bit strings is then as follows.

- *For a single element $v_j \in W$ where $v_j \neq v_0$, define*
  $\texttt{proj}(v_j, \mathcal{A}) = \texttt{rmc}(\left\{(s', w) \middle| (s, w) \in \mathcal{A} \wedge \neg\texttt{singleton}(v_j, s) \wedge s'_i = s_{p \cdot b+k} \,\mathrm{OR}\, s_{p \cdot b+k+d}\right\})$
  where $b = 2^{|W|-w_j}, d = \frac{b}{2}, p = i/d$ and $k = i \mod d, \forall i \in \{0..l(s) - 1\}$

- *For a single element $v_j \in W$ where $v_j = v_0$, define*
  $\texttt{proj}(v_j, \mathcal{A}) = \texttt{rmc}(\left\{(s', w) \middle| (s, w) \in \mathcal{A} \wedge \neg\texttt{singleton}(v_j, s) \wedge s'_i = s_i \,\mathrm{OR}\, s_{l(s)-i-1}\right\})$,
  $\forall i \in \{0..l(s) - 1\}$

- *For a single element v_j and bit string s define*
  $$\texttt{singleton}(v_j, s) = \begin{cases} true & v_j \neq v_0 \wedge s_{2^{|W|-v_j-1}} = 1 \\ true & v_j = v_0 \wedge s_{l-1} = 1 \\ false & \text{otherwise.} \end{cases}$$

- **Project.** *For $X \subseteq W$ and $x \in X$, define*
  $\texttt{proj}(X, \mathcal{A}) = \left\{\texttt{proj}(X \backslash x, \texttt{proj}(x, \mathcal{A}))\right\}$

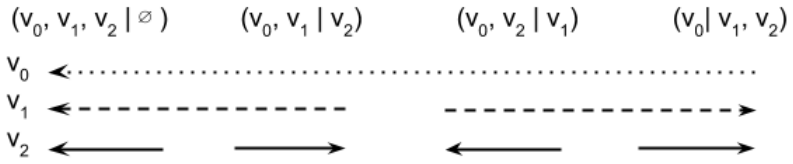

Figure 2.2: Emerging pattern in ordered cuts. The arrows depict on which side of the cuts the corresponding vertex is fixed.

Let us now consider the join operator. Suppose we have some cut c and join the connectivity of partitions p and q. If either p or q is not a refinement of c then there is at least one block b in either partition with vertices in both the left- and righthand side of the cut. When we join the connectivity of these partitions a block in the resulting partition $z = p \sqcap q$ will contain all vertices in b and therefore $c \not\sqsubseteq z$. Vice versa, if $c \sqsubseteq p$ and $c \sqsubseteq q$ then each block in p and q is contained either completely in the left- or righthand side of the cut. Joining the connectivity would not result in blocks containing vertices from both sides. Therefore the z is a refinement of c, i.e. $c \sqsubseteq z$ if and only if $c \sqsubseteq p$ and $c \sqsubseteq q$. Assuming our invariant holds for $(s^a, w^a) \in \mathcal{A}$ and $(s^b, w^b) \in \mathcal{B}$ where $\mathcal{A}$ and $\mathcal{B}$ are based on the same set of vertices W, we know that s_i^a and s_i^b correspond to the same cut c_i. If $\mathcal{A}$ and $\mathcal{B}$ are not based on the same set of vertices we can extend them using the insert operator. We can then adapt the join operator as follows.

- **Join.** *For a table of weighted bit strings $\mathcal{B}$ corresponding to a set of vertices W', define* $\texttt{join}(\mathcal{A}, \mathcal{B}) = \texttt{rmc}(\{(s, w^a + w^b) \big| s_i = (s_i^a \text{ AND } s_i^b) \wedge (s^a, w^a) \in \text{ins}(W' \backslash W, \mathcal{A}) \wedge (s^b, w^b) \in \text{ins}(W \backslash W', \mathcal{B})\})$

For the glue operator, combining sets with vertices $v_j, v_k \in W$ in a partial solution is equal to performing the meet operator with a partition which contains a single class $\{v_j, v_k\}$ and all other vertices as singletons. This partition is a refinement of a cut if v_j and v_k are fixed on the same side. In $\texttt{cuts}(W)$ we have alternating blocks of $2^{|W|-v_j}$ cuts where vertex v_j is fixed to the left side and then on the right (see Fig. 2.2). Using $l_i^{v_j}$ to indicate if v_j is contained in the left side of the cut corresponding to bit i we can then build a bit string $s(v_j v_k)$ for the partition as follows.

- *For vertices v_j and v_k, define* $s(v_j v_k)_i = l_i^{v_j} \text{ XNOR } l_i^{v_k}$ where
$$l_i^{v_j} = \begin{cases} 1, & i \bmod b_{v_k} < \frac{b_{v_k}}{2} \\ 0, & \text{otherwise.} \end{cases},$$
$$l_i^{v_k} = \begin{cases} 1, & i \bmod b_{v_j} < \frac{b_{v_j}}{2} \\ 0, & \text{otherwise.} \end{cases},$$
$b_{v_j} = 2^{|W|-v_j}$ and $b_{v_k} = 2^{|W|-v_k}$.

This gives us a bit string where a bit is set to 1 if both v_j and v_k are completely contained in either the left- or righthand side of the corresponding cut. We then use this bit string in the adaptation of the glue operator.

- **Glue.** *For v_j, $v_k \in W$, define* $\texttt{glue}(v_j v_k, \mathcal{A}) = \texttt{rmc}(\{(s', w) \big| (s, w) \in \mathcal{A} \wedge s'_i = s_i \text{ AND } s(v_j v_k)_i\})$

Finally we trivially adapt the shift operator.

- *For $w' \in \mathbb{N}$, define*
  $\texttt{shft}(w', \mathcal{A}) = \big\{(s, w + w') \big| (s, w) \in \mathcal{A}\big\})$

And the glue with weight operator.

- *For $\omega : W' \times W' \to \mathbb{N}$ where $W' = W \cup \{v_j, v_k\}$, define*
  $\texttt{glue}_\omega(v_j v_k, \mathcal{A}) = \texttt{shft}(\omega(v_j, v_k), \texttt{glue}(v_j v_k, \mathcal{A}))$

This concludes the introduction of the representation of partial solutions using weighted bit strings. For each of the operators defined for weighted partitions we have shown an adaption for the weighted bit string representation. We can now use this representation in any of the connectivity problems for which we can apply the rank based approach. By implicitly representing the partition by its row in the cut-matrix we can compute entries of the cutmatrices more efficiently.

2.4 Implementation

In this section, we give some details on our implementation of the algorithms described in the previous section. We have implemented the algorithms in Java. For each of the test graphs, we used the well known (and quite simple and effective, see e.g., [6]) *Greedy Degree* heuristic to find a tree decomposition. These tree decompositions were subsequently transformed into nice tree decompositions, using the procedure which was previously described in Section 2.2. The algorithms were executed on the thus obtained nice tree decompositions.

The recursions for the different types of nodes were implemented such that we spend linear time per generated entry (before removing double entries, and before the reduction step). For most types, this is trivial. The computation for join bags contains a step, where we are given two partitions, and must compute the partition that is the closure of the combination of the two (i.e., the finest partition that is a coarsening of both). We implemented this step with a breadth first search on the vertices in the bag, with the children of a vertex v all not yet discovered vertices that are in the same block as v in either of the partitions.

Sets $W \subseteq B_x$ are represented by a bitstring. In the computations of join, introduce edge, and forget nodes, it is possible that we generate two or more entries for the same W and partition p of W. Of these duplicate partial solutions, we need to keep only the one with the smallest weight. In order to find such duplicate partial solutions we have represented the partial solution tables in a nested hash-map structure. First we use sets of vertices that where not used in a partial solution as keys, pointing to tables of weighted partitions, effectively grouping partitions consisting of the same base set of vertices together. These weighted partition tables are then represented by another hash-map where the partitions, which are represented as nested sets, are used as keys, pointing to the minimum weight corresponding to the partial solution. This allows us to find and replace any duplicate partial solution in amortized constant time. Java provides hash-codes for sets by adding the hash-codes for all objects contained within a set, which works well enough for the outer hash-table used in our structure. This standard approach breaks down when we use it to calculate hash-codes for partitions however, as it effectively adds all hash-codes of vertices used in the partition together. This results in the same hash-code for all partitions used in the same inner hash-map. To resolve this problem we disrupt this commutative effect by multiplying indexes of vertices contained in each block, and then taking the sum of these values of blocks in order to calculate hash-codes for partitions. We apply the multiplications modulo a prime number to avoid integer overflows. In our experiments, we observed that this approach results in approximately 3% collisions for large tables. In the implementation using weighted bit strings we can directly use the value of these strings as hash codes.

In the implementation of the rank based approach, for each bag, we first compute a table as in the classic algorithm, and then compute the corresponding matrix $\mathcal{C}$, as discussed above.

When we use the weighted bit string representation we fill rows in this matrix by directly copying values from the strings stored in the table. We perform the steps of Gaussian elimination with rows in order of nondecreasing weight. I.e., first we order the rows of $\mathcal{C}$ in order of nondecreasing weight. find the first 1 in the row, and now add the values in this row to all later rows with a 1 in the same column (modulo 2). (This is precisely one step of Gaussian elimination). When a row consists of only 0's, it is linearly dependent on previous processed rows (of smaller weight), and thus safely eliminated. We stop when all partial solutions have been processed, or when we have processed $2^{|W|}$ rows, since all remaining partial solutions are linearly dependent on solutions in $\mathcal{A}$. Any time a partial solution is processed we can eliminate the column containing its leading 1, since all elements in this column are 0.

Chimani et al. [8] give an efficient algorithm for Steiner tree for graphs given with a tree decomposition, that runs in $O(B_{k+2}^2 kn)$ time, with k the width of the tree decomposition. We have chosen not to use the coloring scheme from Chimani et al. [8], but instead use hash tables (as discussed above) to store the tables. Of course, our choice has the disadvantage that we lose a guarantee on the worst case running time (as we cannot rule out scenarios where many elements are hashed to the same position in the hash table), but gives a simple mechanism which works in practice very well. In fact, if we assume that the expected number of collisions of an element in the hash table is bounded by a constant (which can be observed in practice), then the expected running time of our implementation matches asymptotically the worst case running time of Chimani et al.

2.5 Experimental Results

In this section, we will report the results for experiments with the algorithms discussed in Section 2.3. We will denote the classic dynamic programming algorithm as CDP. With RBA, we denote the algorithm where we always apply the reduction step, whereas RBC denotes the algorithm which only applies the reduction step when we have a table whose size is larger than the bound guaranteed by reduction. Similarly, we denote the algorithms using the bit string representation with BSA and BSC where we always apply the reduction step in the former and conditionally in the latter. We will compare the runtime of these five algorithms. Furthermore we will compare the number of partial solutions generated during the execution of CDP, RBA and RBC algorithms to illustrate how much work is being saved by reducing the tables. The number of partial solutions generated for BSA and BSC are comparable to RBA and RBC respectively.

Each of the five algorithms receives as input the same nice tree decomposition of the input graph; this nice tree decomposition is rooted at a forget bag of a terminal vertex. The experiments where performed on sets of graphs of different origin, spanning a range of treewidth sizes of their tree decompositions, and where possible diversified on the number of vertices, edges and terminals. Our graphs come from benchmarks for algorithms for the Steiner Tree problem and for Treewidth. The graphs from Steiner tree benchmarks can be found in Steinlib [19], a repository for Steiner Tree problems. These are prefixed by *b*, *i080* or *es*. Graph instances prefixed by *b* are randomly generated sparse graphs with edge weights between 1 and 10; these were introduced in [3] and were generated following a scheme outlined in [1]. The *i080* graph instances are randomly generated sparse graphs with incidence edge weights, introduced in [12]. We have grouped these sparse graphs together in the results. The next set of instances, prefixed by *es*, were generated by placing random points on a two-dimensional grid, which serve as terminals. By building the grid outlined in [15] they where converted to rectilinear graphs with L1 edge weights and preprocessed with GeoSteiner [28]. The last collection of graphs are often used as benchmarks for algorithms for Treewidth. These come from Bayesian network and graph colouring applications. We transformed these to Steiner Tree instances by adding random edge weights between 1 and 1000, and by selecting randomly a subset of the vertices

as terminals (about 20% of the original vertices). These graphs can be found in [26].

All algorithms have been implemented in Java and the computations have been carried out on a Windows-7 operated PC with an Intel Core i5-3550 processor and 16.0 GB of available main memory. We have given each of the algorithms a maximum time of two hours to find a solution for a given instance; in the tables, we marked instances halted due to the use of the maximum time by a *.

instance	$\mathtt{tw}(\mathbb{T})$	$\|V\|$	$\|E\|$	$\|T\|$	CDP	RBA	RBC	BSA	BSC
b01.stp	4	50	63	9	63	55	19	26	22
b02.stp	4	50	63	13	12	30	12	9	8
b08.stp	6	75	94	19	592	122	73	10	7
b09.stp	6	75	94	38	88	55	38	6	6
b13.stp	7	100	125	17	1552	548	892	95	240
b14.stp	7	100	125	25	2001	515	336	43	32
b15.stp	8	100	125	50	15860	1695	1503	161	169
i080-001.stp	9	80	120	6	477716	13386	9279	1571	1251
i080-003.stp	9	80	120	6	1996394	21598	19250	3077	3019
i080-004.stp	10	80	120	6	2283606	74845	74464	14464	18197
b06.stp	10	50	100	25	1449534	36041	28389	6021	5379
I080-005.stp	11	80	120	6	*	815457	723720	236683	293567
b05.stp	11	50	100	13	*	341862	275824	137917	118226

Table 2.1: Runtime in milliseconds for instances from Steinlib (1)

instance	$\mathtt{tw}(\mathbb{T})$	$\|V\|$	$\|E\|$	$\|T\|$	CDP	RBA	RBC	BSA	BSC
es90fst12.stp	5	207	284	90	76	130	65	19	11
es100fst10.stp	5	229	312	100	116	177	93	20	16
es80fst06.stp	6	172	224	80	308	329	185	30	22
es100fst14.stp	6	198	253	100	133	179	93	19	14
es90fst01.stp	7	181	231	90	684	351	201	29	20
es100fst13.stp	7	254	361	100	1594	1351	804	112	84
es100fst15.stp	8	231	319	100	2069	1470	826	120	101
es250fst03.stp	8	543	727	250	3320	2343	1484	206	162
es100fst08.stp	9	210	276	100	5088	2588	2165	309	321
es250fst05.stp	9	596	832	250	35961	14521	8322	1550	1109
es250fst07.stp	10	585	799	250	127681	60701	37042	7508	5942
es500fst05.stp	10	1172	1627	500	145408	51504	34684	5972	4933
es250fst12.stp	11	619	872	250	*	138073	99427	23311	20045
es100fst02.stp	12	339	522	100	*	365800	299014	150013	143582
es250fst01.stp	12	623	876	250	*	395694	288476	105810	91650
es250fst08.stp	13	657	947	250	*	2469463	2208040	1257730	1236192
es250fst13.stp	13	713	1053	250	*	2725460	2416867	1684224	1557617

Table 2.2: Runtime in milliseconds for instances from Steinlib (2)

In Tables 3.1 – 3.3, we have gathered the results for the runtimes of the five algorithms for the aforementioned graph instances. We immediately notice that RBC outperforms RBA in all cases. In Table 3.4 –3.6 we give the number of partial solutions (table entries) computed for each of the CDP, RBA and RBC algorithms. If we investigate these tables we notice that the number of partial solutions computed during RBA is not significantly smaller compared to the

instance	$\mathtt{tw}(\mathbb{T})$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	CDP	RBA	RBC	BSA	BSC
myciel3.stp	5	11	20	2	8	9	5	1	<1
BN_28.stp	5	24	49	4	7	15	8	2	2
pathfinder.stp	6	109	211	21	599	281	157	26	18
csf.stp	6	32	94	6	1135	254	165	19	15
oow-trad.stp	7	33	72	6	803	601	371	50	36
mainuk.stp	7	48	198	9	10040	3925	2444	291	214
ship-ship.stp	8	50	114	10	6015	3929	2465	352	254
barley.stp	8	48	126	9	3000	1836	1248	168	142
miles250.stp	9	128	387	25	37745	14099	8444	1761	1291
jean.stp	9	80	254	16	17988	20404	9231	1907	1175
huck.stp	10	74	301	14	18652	37696	20376	3829	2657
myciel4.stp	11	23	71	4	1602408	86183	83358	16385	23824
munin1.stp	11	189	366	37	*	521081	501164	162717	227469
pigs.stp	12	441	806	88	*	1130537	1071765	1469987	1791343
anna.stp	12	138	493	27	*	5515952	4822620	2357740	2398758

Table 2.3: Runtime in milliseconds for instances on graphs from TreewidthLib

instance	$\mathtt{tw}(\mathbb{T})$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	CDP	RBA	RBC
b01.stp	4	50	63	9	1921	1654	1654
b02.stp	4	50	63	13	1948	1628	1638
b08.stp	6	75	94	19	99740	11654	12005
b09.stp	6	75	94	38	18615	5302	5302
b13.stp	7	100	125	17	279852	47032	58717
b14.stp	7	100	125	25	318744	37406	38146
b15.stp	8	100	125	50	2248833	76681	93161
i080-001.stp	9	80	120	6	65460491	570132	571425
i080-003.stp	9	80	120	6	249390279	1279544	1282358
i080-004.stp	10	80	120	6	256761016	2687590	3507987
b06.stp	10	50	100	25	151246080	723392	754926
I080-005.stp	11	80	120	6	*	25194893	29825246
b05.stp	11	50	100	13	*	6827459	6955686

Table 2.4: Number of generated partial solutions for instances of Steinlib (1)

number computed during RBC. From these results and their running times we can conclude that it is preferable to use the reductions more sparingly in order to decrease runtime, since applying the reductions when the tables are already smaller than their size guarantee does not seem to have a noteworthy effect. In the case of BSA and BSC the prefered strategy is less clear, since we inherently perform part of the reduction step, i.e. the filling of cutmatrices, during the table calculations.

We also notice that, while RBA outperforms CDP in numerous cases, RBC outperforms CDP in all but one (discussed below). For example, in the case of *i080-004* we see a significant speed-up: the classic DP uses 38 minutes to find the optimal solution, but RBC uses just 74 seconds. Furthermore we see a strong increase in the runtime difference when the width of the tree decompositions increases. This is further reflected in Table 3.4, where we see that when the width of the tree decompositions increases, the difference in the number of of generated partial solutions grows significantly. Again, for algorithms BSA and BSC we see further significant speed-ups compared to RBA and RBC for all but the smallest instances. In the case of *i080-*

instance	$\mathtt{tw}(\mathbb{T})$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	CDP	RBA	RBC
es90fst12.stp	5	207	284	90	25817	17693	17706
es100fst10.stp	5	229	312	100	34612	22181	22204
es80fst06.stp	6	172	224	80	73436	31721	32301
es100fst14.stp	6	198	253	100	35664	21947	21971
es90fst01.stp	7	181	231	90	137705	30097	30139
es100fst13.stp	7	254	361	100	323259	99203	99420
es100fst15.stp	8	231	319	100	388118	100469	100487
es250fst03.stp	8	543	727	250	593651	151722	151802
es100fst08.stp	9	210	276	100	724207	84869	90006
es250fst05.stp	9	596	832	250	5283073	739953	740698
es250fst07.stp	10	585	799	250	15397120	1664352	1665205
es500fst05.stp	10	1172	1627	500	17953689	1790843	1791361
es250fst12.stp	11	619	872	250	*	3771954	3772893
es100fst02.stp	12	339	522	100	*	4909388	4909500
es250fst01.stp	12	623	876	250	*	4715125	4715631
es250fst08.stp	13	657	947	250	*	18954259	19509166
es250fst13.stp	13	713	1053	250	*	15870380	16101777

Table 2.5: Number of generated partial solutions for instances of Steinlib (2)

instance	$\mathtt{tw}(\mathbb{T})$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	CDP	RBA	RBC
myciel3.stp	5	11	20	2	2382	1295	1347
BN_28.stp	5	24	49	4	2346	1670	1700
pathfinder.stp	6	109	211	21	128163	21206	22073
csf.stp	6	32	94	6	206434	21111	21215
oow-trad.stp	7	33	72	6	164723	39318	39327
mainuk.stp	7	48	198	9	1691584	202454	210694
ship-ship.stp	8	50	114	10	1093800	144493	144682
barley.stp	8	48	126	9	472223	77799	84125
miles250.stp	9	128	387	25	5524562	273711	278717
jean.stp	9	80	254	16	2932817	292577	302644
huck.stp	10	74	301	14	3238678	526947	531597
myciel4.stp	11	23	71	4	203990952	1876695	3482635
munin1.stp	11	189	366	37	*	19289467	23535116
pigs.stp	12	441	806	88	*	13488332	16814404
anna.stp	12	138	493	27	*	82060857	99551566

Table 2.6: Number of generated partial solutions for instances on graphs from TreewidthLib

004 we now see that BSA uses just 14 seconds and BSC uses 17 seconds.

The *huck* instance is the only example where using a straightforward implementation of the rank based approach does not pay off. Upon further inspection we found that the tree decomposition for this instance has only one bag of size 11, while most of the other bags are of size 7 and below. This is also reflected by the difference in the number of generated partial solutions, where the improvement factor is not comparable to the other cases. Conversely we found that the *i080-004* case included 18 bags of treewidth 11 of which 6 where join bags, which explains the extreme difference. In practice, when we run dynamic programming algorithms on tree decompositions, the underlying structure of the decomposition has a large influence on the performance, which is not always properly reflected by the treewidth of a graph. In general however, the rank based approach is more and more advantageous as the treewidth increases,

even allowing us to find solutions where CDP does not find any within the time limit. The implementation of the rank based approach using bit strings gives us an even better performance. However, when comparing the proportion of decrease in running times between straightforward and bit string implementations we see slight diminishing returns as the treewidth increases. As treewidth increases the Guassian elimination step which is the bottleneck of the algorithm in theory starts to have more influence on the running time of the algorithms. Nevertheless, in a practical setting the bit string representation seems to be very advantageous.

2.6 Discussion and Concluding Remarks

In this paper, we presented an experimental evaluation of the rank based approach by Bodlaender et al. [5], comparing the classic dynamic programming for Steiner Tree and the new versions based on Gaussian elimination. The results are very promising: even for relatively small values of the width of the tree decompositions, the new approach shows a notable speed-up in practice. The theoretical analysis of the algorithm already predicts that the new algorithms are asymptotically faster, but it is good to see that the improvement already is clearly visible at small size benchmark instances. Furthermore, we have presented an implementation of the rank based approach using weighted bit strings to directly identify rows in $\mathcal{C}$. This implementation yields even further significant improvements on the running time.

Overall, the rank based approach is an example of the general technique of representativity: a powerful but so far underestimated paradigmatic improvement to dynamic programming. A further exploration of this concept, both in theory (improving the asymptotic running time for problems) as in experiment and algorithm engineering seems highly interesting. Our current paper gives a clear indication of the practical relevance of this concept.

We end this paper with a number of specific points for further study:

- The rank based approach also promises faster algorithms on tree decompositions for several other problems. The experimental evaluation can be executed for other problems. In particular, for Hamiltonian Circuit and similar problems, it would be interesting to compare the use of the basis from [5] with the smaller basis given by Cygan et al. [10].
- How well does the *Cut and Count* method perform? As remarked in [11], it seems advantageous to use polynomial identity testing rather then the isolation lemma to optimize the running time.
- In what extent do results change if we use normal (instead of nice) tree decompositions?
- What is the effect of the ratio between the number of terminals and the number of vertices on the running times?
- Are running time improvements possible by other forms of reduction of tables (without affecting optimality)? If we exploit the two families theorem by Lovász [21], we obtain a variant of our algorithm, with a somewhat different reduce algorithm [14] (see also [22]); how does the running time of this version compare with the running time of the algorithm we studied?
- Can we use the rank based approach to obtain a faster version of the *tour merging* heuristic for TSP by Cook and Seymour [9]? Also, it would be interesting to try a variant of tour merging for other problems, e.g., 'tree merging' as a heuristic for Steiner Tree.
- For what other problems does the rank based approach give faster algorithms in practical settings?
- Are there good heuristic ways of obtaining small representative sets, even for problems where theory tells us that representative sets are large in the worst case?

Chapter 3

Evaluating Partial Solutions in Order of Optimality

3.1 Introduction

In a straightforward implementation of dynamic programming algorithms on tree decompositions complete tables of partial solutions are calculated for nodes in the decomposition in a bottom-up fashion. In the weighted case of connectivity problems such as those discussed in the paper by Bodlaender et al. [5] we are only interested in the optimal solution. In an attempt to exploit this we will now consider an algorithm that calculates partial solutions in order of optimality.

In the case of Weighted Steiner Tree, which we will again use as the running example in the second part of this thesis, this means that we will calculate partial solutions as follows. A node in the tree decomposition is able to request the next smallest partial solution calculated for any of its children. It may need to do this multiple times in order to ensure that it can compute the next smallest partial solution which it will then be able to pass along to its parent node. As soon as the root node has acquired its first partial solution we have a solution for Weighted Steiner Tree since it is also the solution of minimal weight.

While this approach does not improve the running time of the algorithm in the worst case, many partial solutions which would be generated when we compute entire tables may possibly be skipped. For example, any non-minimum weight solution in the root bag and many of the partial solutions generated in descendent nodes that were at some point used to compute this non-minimum weight solution will not be considered. In general, we save work at any node after it has calculated a partial solution which at some point can be extended to a full solution of minimal weight. The overall effectiviness of this algorithm is instance specific.

An obvious obstacle for this algorithm would be its space efficiency. This is because we need to keep internal state at every node in the tree decomposition. In the worst case scenario where we still need to consider every partial solution that would otherwise be generated during the bottom-up dynamic programming algorithm this could cause us to run out of memory since we simultaneously store sets of partial solutions at every node. However, we can still apply the rank based reductions by rank based table reductions by Bodlaender et al. [5] to eliminate linearly dependent partial solutions. This allows us to keep the amount of generated partial solutions in check which, besides the decrease in time complexity, results in a more efficient use of space. Furthermore, we can use the bit string representation from the first part of this thesis which, besides allowing us to eliminate linearly dependent solutions faster, is also very space efficient for low treewidth.

In this part of the thesis we will give a detailed description of this new approach applied to Weighted Steiner Tree. First we will show the internal logic used to calculate partial solutions in ascending order for each type of node in a *nice* tree decomposition. We will then show how to apply the rank based reductions to eliminate linearly dependent partial solutions piecemeal as they are generated. Finally, we will perform an experimental evaluation of this algorithm, where we compare its performance to the performance of the algorithms from the first part of the thesis which use the bit string representation to calculate tables in a bottom-up fashion.

For brevity we will skip the preliminaries and refer the reader to the first part of the thesis. The sequel is organized as follows. A detailed description of the new algorithm for Weighted Steiner Tree will be given in Section 3.2. In Section 3.3, we will discuss the implementation after which we will present the experimental results in Section 3.4. Finally, conclusions are given in section 3.5.

3.2 Algorithm for Steiner Tree

As an initialization step we will split each bag in the nice tree composition given in the input for Weighted Steiner Tree into nodes for every subset of vertices. We do this because a set

of partial solutions based on some subset W for a given bag is dependent on partial solutions based on at most two subsets from children of this bag. Splitting the bags into these more localized nodes will simplify the procedure to find the next minimal weight partial solution in each node.

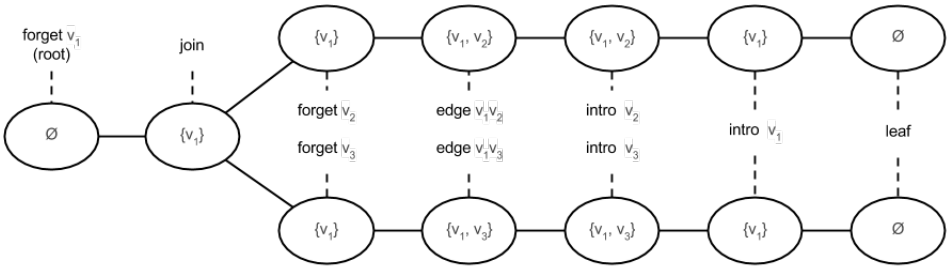

Figure 3.1: An example nice tree decomposition containing every type of bag. For each bag its set of vertices is shown.

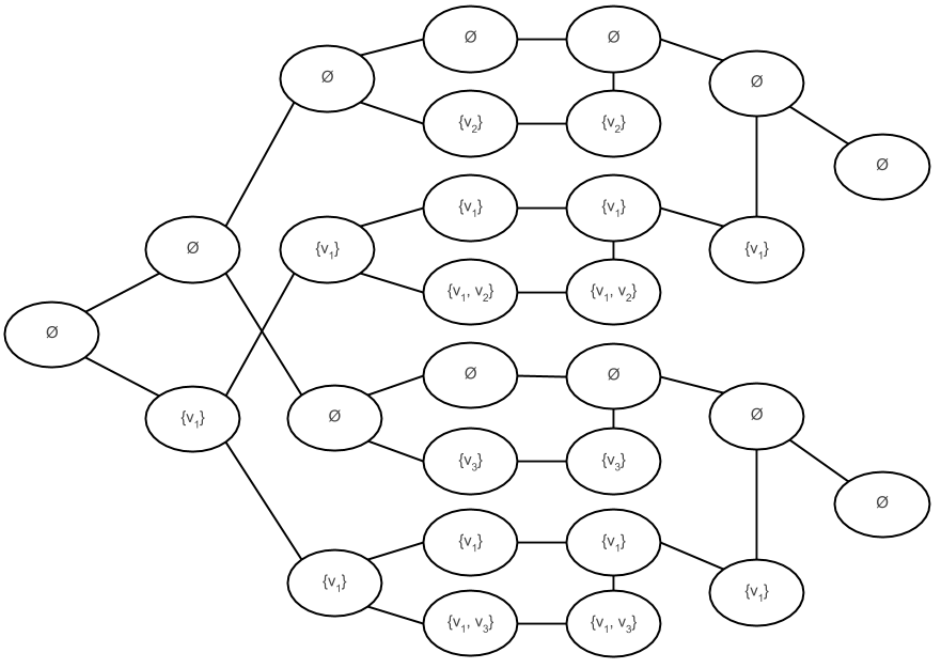

Figure 3.2: Structure after splitting bags from the nice tree decomposition in Fig. 3.1 into nodes for every subset of vertices.

Let us denote the node for some subset $W \subseteq B_x$ in bag x as $n_x(W)$. We split the different types of bags in a nice tree decomposition (see Fig. 3.1 for an example) as follows. For a leaf bag x we have a single node $n_x(\emptyset)$ for the empty set. For an introduce vertex v bag x we spawn two nodes for each node $n_y(W)$ in child bag y. The first node $n_x(W)$ will have $n_y(W)$ as a child and will be able to request partial solutions from this node. The second node $n_x(W \cup \{v\})$ will also have access to these partial solutions through $n_x(W)$. This ensures that for each node there is only a single parent for which it has to generate partial solutions. For an introduce edge $e = uv$ bag x with child bag y we spawn node $n_x(W)$ for each node $n_y(W)$. For a forget vertex v bag x with child bag y we have pairs of nodes $n_y(W), n_y(W \setminus \{v\})$ for which we spawn node $n_x(W)$ which has this pair of nodes as children. Finally, for join bag x with children y and z

we have pairs of nodes $n_y(W), n_z(W)$ for which we spawn node $n_x(W)$ which has this pair as children. Now that we have this new structure of nodes (see Fig. 3.2 for an example) we can proceed to define the internal procedures of each type of node in such a way that they generate partial solutions in ascending order of weight.

3.2.1 Evaluating Partial Solutions in Ascending Order of Weight

We will now revisit the recurrence of the classic dynamic programming algorithm for Weighted Steiner Tree which uses the special set of operators introduced in [5]. As discussed in the first part of this thesis, this ensures that we can apply the rank based approach and we can use either the weighted partition representation or the weighted bit string representation of partial solutions. As we reiterate the recurrence relations for each type of bag in a nice tree decomposition we will show the procedure of the corresponding nodes in the formerly introduced structure. In the forget, introduce edge and join bags we may generate multiple partial solutions that represent the same connectivity between their vertices. In this case we only need to keep the partial solution of minimum weight. We will ignore this for now and deal with it in Sect. 3.2.2.

For each node $n_x(W)$ corresponding to bag x with subset $W \subseteq B_x$ we define the function nextPartialSolution($n_x(W)$) which returns the next partial solution of minimum weight in this node. Nodes keep some internal state and may call this function from their children in order to generate these partial solutions. We will first introduce some global variables which nodes may use to for decision making. Every node $n_x(W)$ has a boolean $\texttt{hasNext}(n_x(W))$ indicating if it can generate any more partial solutions. If a node has a child $n_y(W')$, then it may inquire about $\texttt{hasNext}(n_y(W'))$. The starting condition of this boolean is $\texttt{hasNext}(n_x(W)) \leftarrow$ true for each bag $n_x(W)$ unless stated otherwise. Furthermore, nodes may use a boolean $\texttt{request}(n_y(W'))$ for each child $n_y(W')$ indicating if it needs a new partial solution from this child. The starting condition for these booleans is also $\texttt{request}(n_y(W')) \leftarrow$ true for each child.

We will start with the trivial procedure of leaf nodes. The recurrence for a leaf bag x is defined as.

$$A_x(\emptyset) = \{(\emptyset, 0)\}$$

A leaf bag has a single partial solution of weight 0 in which there are no vertices to connect. In leaf nodes we only return this partial solution, i.e. the procedure for leaf nodes (Alg. 1) is as follows.

Algorithm 1: nextPartialSolution($n_x(\emptyset)$) for leaf nodes.

$\texttt{hasNext}(n_x(\emptyset)) \leftarrow$ false
return $\{(\emptyset, 0)\}$

For an introduce vertex v bag x with child y the recurrence is defined as.

$$A_x(W) = \begin{cases} \texttt{ins}(\{v\}, A_y(W \setminus \{v\})) & \text{if } v \in W \\ A_y(W) & \text{if } v \notin W \wedge v \notin K \\ \emptyset & \text{if } v \notin W \wedge v \in K \end{cases}$$

For each partial solution in the child table we can decide whether or not to use v. However, if v is a terminal then we are forced to use it. For each case we will include a partial solution in the table for bag x. The procedure for introduce vertex nodes (Alg. 2) is as follows. For a child node $n_y(W \setminus \{v\})$ we now have two introduce vertex nodes $n_x(W \setminus \{v\})$ and $n_x(W)$. Since there is no change in weight, the former will return partial solutions from $n_y(W \setminus \{v\})$ in the order it receives them. Likewise, the latter will return partial solutions from $n_y(W \setminus \{v\})$ in

the same order while inserting v. Now we must make sure that at any time a partial solution is returned by $n_y(W \setminus \{v\})$ it is recieved by both $n_x(W \setminus \{v\})$ and $n_x(W)$.

To this end, both introduce vertex nodes will keep a `list` in which partial solutions can be stored. If either node requests a partial solution from the child it is added to both lists. Now either node can process partial solutions in its list or request more if it is empty. Finally, an introduce vertex node can not return partial solutions if its child has no more partial solutions and its list is empty. In the case that v is terminal we set $\texttt{hasNext}(n_x(W \setminus \{v\})) \leftarrow$ false since we cannot return any valid partial solutions in this node.

Algorithm 2: nextPartialSolution($n_x(W)$) for introduce vertex v nodes.

```
if v ∈ W then
    if request(n_y(W \ {v})) then
        p ← nextPartialSolution(n_y(W \ {v}))
        add p to list
        add p to list of node n_x(W \ {v})
        request(n_y(W \ {v})) ← false
    r ← first element of list
    remove r from list
    if list is empty then
        if hasNext(n_y(W \ {v})) then
            request(n_y(W \ {v})) ← true
        else
            hasNext(n_x(W)) ← false
    r ← ins({v}, r)
else
    if request(n_y(W)) then
        p ← nextPartialSolution(n_y(W))
        add p to list
        add p to list of node n_x(W ∪ {v})
        request(n_y(W)) ← false
    r ← first element of list
    remove r from list
    if list is empty then
        if hasNext(n_y(W)) then
            request(n_y(W)) ← true
        else
            hasNext(n_x(W)) ← false
return r
```

For a forget vertex v bag x with child y the recurrence is defined as.

$$A_x(W) = A_y(W) \uplus \texttt{proj}(v, A_y(W \setminus \{v\}))$$

Partial solutions in the child table either use v or do not use v. In the former case we can simply include it in the table of bag x. In the latter case we only insert it if v is connected to some other vertex in the partial solution and discard it otherwise. The procedure (Alg. 3) for forget vertex node $n_x(W)$ with children $n_y(W)$ and $n_y(W \cup \{v\})$ is as follows. We consider partial solutions from both children. If the partial solution from $n_y(W)$ is smaller we return it. Otherwise, we return the partial solution from $n_y(W \cup \{v\})$ after removing v from it. In the case that this partial solution is invalid we recursively call nextPartialSolution($n_x(W)$). Finally,

if either child can no longer return partial solutions we return the remaining partial solutions of the other child. Once this child is also depleted we can no longer return anything.

Algorithm 3: nextPartialSolution($n_x(W)$) for forget vertex v nodes.

```
if ¬hasNext(n_y(W)) ∧ request(n_y(W)) then
    if request(n_y(W ∪ {v})) then
        r ← nextPartialSolution(n_y(W ∪ {v}))
        request(n_y(W ∪ {v})) ← false
    else
        r ← p_2
    if ¬hasNext(n_y(W ∪ {v})) then
        hasNext(n_x(W)) ← false
    r ← proj({v}, r)
else if ¬hasNext(n_y(W ∪ {v})) ∧ request(n_y(W ∪ {v})) then
    if request(n_y(W)) then
        r ← nextPartialSolution(n_y(W))
        request(n_y(W)) ← false
    else
        r ← p_1
    if ¬hasNext(n_y(W)) then
        hasNext(n_x(W)) ← false
else
    if request(n_y(W)) then
        p_1 ← nextPartialSolution(n_y(W))
        request(n_y(W)) ← false
    if request(n_y(W ∪ {v})) then
        p_2 ← nextPartialSolution(n_y(W ∪ {v}))
        request(n_y(W ∪ {v})) ← false
    if w(p_1) < w(p_2) then
        r ← p_1
        request(n_y(W)) ← true
    else
        r ← proj({v}, p_2)
        request(n_y(W ∪ {v})) ← true
if r is invalid then
    r ← nextPartialSolution(n_x(W))

return r
```

The recurrence for an introduce edge $e = uv$ bag x with child y is defined as.

$$A_x(W) = \begin{cases} A_y(W) & \text{if } u \notin W \vee v \notin W \\ A_y(W) \uplus \texttt{glue}_\omega(uv, A_y(W)) & \text{otherwise.} \end{cases}$$

For each partial solution in the child table we can either ignore the edge or include it. We can only include the edge in the partial solution if it uses both u and v. We insert partial solutions for each case in the table for bag x. The procedure (Alg. 4) for introduce edge nodes $n_x(W)$ with child node $n_y(W)$ is as follows. If $u \notin W \wedge v \notin W$ we simply return partial solutions from the child node in the order we recieve them. In the other case we return partial solutions from the child node, until at some point we have to return partial solutions in which the edge

is included. To keep a correct order we will use a `list` in which we store partial solutions from the child node. When we recieve a new partial solution p_2 we add it to this list. The other partial solutions in this list, of which p_1 is the first and smallest, have already been returned to the parent. We then consider either including the edge in p_1 and returning it or returning p_2. In the former case we remove p_1 from the list since we are done with this particular partial solution. In the latter case we need to request a new partial solution from the child. Finally, when the child has no more partial solutions we finish processing the list until it is empty, after which we can no longer return anything.

Algorithm 4: nextPartialSolution($n_x(W)$) for introduce edge uv nodes.

```
if u ∉ W ∨ v ∉ W then
    r ← nextPartialSolution(n_y(W))
    if ¬hasNext(n_y(W)) then
        hasNext(n_x(W)) ← false
else
    if hasNext(n_y(W)) ∨ ¬request(n_y(W)) then
        if request(n_y(W)) then
            add nextPartialSolution(n_y(W)) to list
        p_1 ← first element of list
        p_2 ← last element of list
        if w(p_1) + ω(uv) < w(p_2) then
            r ← glue_ω(uv, p_1)
            remove p_1 from list
            request(n_y(W)) ← false
        else
            r ← p_2
            request(n_y(W)) ← true
    else
        p ← first element of list
        r ← glue_ω(uv, p)
        remove p from list
        if list is empty then
            hasNext(n_x(W)) ← false
return r
```

Lastly, the recurrence for a join bag x with children y and z is defined as.

$$A_x(W) = \texttt{join}(A_y(W), A_z(W))$$

We have partial solutions from bag y in which vertices are connected using edges introduced in its decendents. Likewise, we have partial solutions from bag z. For every combination of partial solutions from bags y and z which share the same base set of vertices W we add a partial solution in the table of x in which the connectivity is joined and weight is combined. The procedure (Alg. 5 for join nodes $n_x(W)$ with children $n_y(W)$ and $n_z(W)$ is as follows. We keep a $\texttt{list}_y$ of partial solutions from $n_y(W)$, a $\texttt{list}_z$ of partial solutions from $n_z(W)$, and a `queue` of pairs of partial solutions which we have yet to return. When we recieve a partial solution p from $n_y(W)$ or q from $n_z(W)$ we add it to $\texttt{list}_y, \texttt{list}_z$ respectively. We then add every new pair (p, q) to the queue. We then return the pair of smallest combined weight and remove it from the queue. If a partial solution from this pair is the last partial solution that we

have recieved from either child we must request a new partial solution from this child. We do this to make sure that there is no undiscovered pair of partial solutions of smaller weight than anything in the queue.

Algorithm 5: nextPartialSolution($n_x(W)$) for join nodes.

```
if request(n_y(W)) then
    p ← nextPartialSolution(n_y(W))
    add p to list_y
    request(n_y(W)) ← false
    for every q in list_z do
        add pair (p, q) to queue
if request(n_z(W)) then
    q ← nextPartialSolution(n_z(W))
    add q to list_z
    request(n_z(W)) ← false
    for every p in list_y do
        add pair (p, q) to queue
(p, q) ← pair from queue with minimal w(p) + w(q)
r ← join(p, q)
if p is last element of list_y ∧ hasNext(n_y(W)) then
    request(n_y(W)) ← true
if q is last element of list_z ∧ hasNext(n_z(W)) then
    request(n_z(W)) ← true
if queue is empty ∧¬hasNext(n_y(W)) ∧ ¬hasNext(n_z(W)) then
    hasNext(n_x(W)) ← false
return r
```

This concludes the recurrence for Weighted Steiner Tree. We have shown procedures in which partial solutions are generated in ascending order of weight for any type of node which capture the recurrence given in any type of bag in a nice tree decomposition.

3.2.2 Piecemeal Elimination of Linearly Dependent Partial Solutions

We will now show how to apply the rank based reductions in order to eliminate new partial solutions that are linearly dependent on previously returned partial solutions. We will also deal with partial solutions that represent the same connectivity between vertices in which case we only want to return the partial solution of minimum weight. Once we return a partial solution we know that we can eliminate any further partial solution that represents the same connectivity since they are returned in ascending order of weight. The same goes for linearly dependent partial solutions. Any partial solution that is linearly dependent on a previously generated partial solution can be eliminated since it cannot be part of a minimum weight basis.

We will eliminate partial solutions as follows. Each node $n_x(W)$ will have a procedure eliminate(p) which determines if we should eliminate partial solution (p, w). If this procedure (Alg. 6) returns *no* we will return (p, w) to the parent node. If it returns *yes* we will not return (p, w) but continue until we generate a partial solution which is not eliminated. Each node will keep a $\texttt{list}_p$ of partitions (or bit strings) of partial solutions it has previously returned. We will eliminate the partial solution if p is in this list and add it to the list otherwise. We also keep a collection of `rows` from matrix $\mathcal{C}$ on which elementary row operations have already been applied. We then calculate the row r_1 corresponding to p in $\mathcal{C}$ and apply row operations on it

for every row $r_2 \in$ `rows`. We eliminate (p, w) if it is linearly dependent, i.e. r_1 is all zeros, or add r_1 to `rows` and return it otherwise.

Algorithm 6: eliminate(p)

```
if p ∈ list_p then
    KwRet yes
else
    add p to list_p
r_1 ← row of p in C
for row r_2 ∈ rows do
    l ← leading one in r_2
    if bit in r_1 at position l = 1 then
        r_1 ← r_1 + r_2  mod 2
if every bit in r_1 is 0 then
    return yes
else
    add r_1 to rows
    return no
```

This concludes the description of the algorithm which evaluates partial solutions for Weighted Steiner Tree in ascending order of weight, for which we have shown that we can apply the rank based approach to eliminate linearly dependent partial solutions as they are generated.

3.3 Implementation

In this section, we will discuss the implementation of the algorithm described in the previous section and show some pitfalls which should be avoided. We have implemented the algorithm in Java using the bit string representation for partial solutions discussed in the first part of the thesis.

The queue in join nodes is implemented using a priority queue. While insertion in this queue takes time logarithmic in the queue size we have observed that adding pairs takes no significant part in the runtime since the queue stays relatively small.

We keep an extra list at each node containing the position of leading ones in the collection of rows. We do this in order to save time since we need these positions every time a row is considered for elimination. Partial solutions will never be eliminated in leaf nodes, introduce vertex nodes, and the introduce edge nodes in which edges can not be introduced. In these nodes we directly return partial solutions to the parent node, without considering them for elimination using the procedure from Sect. 3.2.2, in order to save work.

Some care should be taken when we eliminate partial solutions. In some cases the node may not be able to generate any more partial solutions after a partial solution has been eliminated. This can happen in the procedure of the forget vertex nodes as well when the project operator returns an invalid partial solution. Therefore, the procedure in each node should be able to deal with the event that a child does not return a partial solution when one is requested. Since full description of handling this requirement is somewhat tedious and uninstructive we invite the reader to view our code for more details.

Our software is publicly available, can be used under a GNU Lesser General Public Licence, and can be downloaded at:
http://www.staff.science.uu.nl/ bodla101/java/steiner.zip

3.4 Experimental Results

In this section, we will report the results for experiments with the algorihm discussed in Section 3.2 which we will denote as ORDER. We will compare the performance of this algorithm to the performance of BSA and BSC from the first part of the thesis. We will compare the runtime of these algorithms. Furthermore, we will compare the number of partial solutions generated during the execution of these algorithms to illustrate how much work is being saved by generating partial solutions in ascending order of weight. We will use the same graph instances from Steinlib [19] and Treewidthlib [26] that were used in the first part of the thesis.

All algorithms have been implemented in Java and the computations have been carried out on a Windows-7 operated PC with an Intel Core i5-3550 processor and 16.0 GB of available main memory. The results of the ORDER algorithm where gathered at the same time as the

instance	$\mathtt{tw}(\mathbb{T})$	$\|V\|$	$\|E\|$	$\|T\|$	BSA	BSC	ORDER
b01.stp	4	50	63	9	26	22	31
b02.stp	4	50	63	13	9	8	20
b08.stp	6	75	94	19	10	7	13
b09.stp	6	75	94	38	6	6	7
b13.stp	7	100	125	17	95	240	198
b14.stp	7	100	125	25	43	32	41
b15.stp	8	100	125	50	161	169	337
i080-001.stp	9	80	120	6	1571	1251	2259
i080-003.stp	9	80	120	6	3077	3019	2369
i080-004.stp	10	80	120	6	14464	18197	22040
b06.stp	10	50	100	25	6021	5379	10831
I080-005.stp	11	80	120	6	236683	293567	234546
b05.stp	11	50	100	13	137917	118226	156883

Table 3.1: Runtime in milliseconds for instances from Steinlib (1)

instance	$\mathtt{tw}(\mathbb{T})$	$\|V\|$	$\|E\|$	$\|T\|$	BSA	BSC	ORDER
es90fst12.stp	5	207	284	90	19	11	17
es100fst10.stp	5	229	312	100	20	16	19
es80fst06.stp	6	172	224	80	30	22	26
es100fst14.stp	6	198	253	100	19	14	15
es90fst01.stp	7	181	231	90	29	20	37
es100fst13.stp	7	254	361	100	112	84	134
es100fst15.stp	8	231	319	100	120	101	147
es250fst03.stp	8	543	727	250	206	162	264
es100fst08.stp	9	210	276	100	309	321	548
es250fst05.stp	9	596	832	250	1550	1109	2461
es250fst07.stp	10	585	799	250	7508	5942	13197
es500fst05.stp	10	1172	1627	500	5972	4933	9580
es250fst12.stp	11	619	872	250	23311	20045	37366
es100fst02.stp	12	339	522	100	150013	143582	253664
es250fst01.stp	12	623	876	250	105810	91650	109069
es250fst08.stp	13	657	947	250	1257730	1236192	1806706
es250fst13.stp	13	713	1053	250	1684224	1557617	1617842

Table 3.2: Runtime in milliseconds for instances from Steinlib (2)

instance	$\mathtt{tw}(\mathbb{T})$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	BSA	BSC	ORDER
myciel3.stp	5	11	20	2	1	< 1	< 1
BN_28.stp	5	24	49	4	2	2	1
pathfinder.stp	6	109	211	21	26	18	31
csf.stp	6	32	94	6	19	15	15
oow-trad.stp	7	33	72	6	50	36	32
mainuk.stp	7	48	198	9	291	214	271
ship-ship.stp	8	50	114	10	352	254	372
barley.stp	8	48	126	9	168	142	203
miles250.stp	9	128	387	25	1761	1291	1677
jean.stp	9	80	254	16	1907	1175	460
huck.stp	10	74	301	14	3829	2657	863
myciel4.stp	11	23	71	4	16385	23824	3208
munin1.stp	11	189	366	37	162717	227469	99843
pigs.stp	12	441	806	88	1469987	1791343	513948
anna.stp	12	138	493	27	2357740	2398758	509029

Table 3.3: Runtime in milliseconds for instances on graphs from TreewidthLib

instance	$\mathtt{tw}(\mathbb{T})$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	BSA	BSC	ORDER
b01.stp	4	50	63	9	1653	1653	1284
b02.stp	4	50	63	13	1629	1639	1364
b08.stp	6	75	94	19	11655	12006	6726
b09.stp	6	75	94	38	5292	5292	3840
b13.stp	7	100	125	17	47004	58682	29813
b14.stp	7	100	125	25	37395	37535	21299
b15.stp	8	100	125	50	76692	90363	44150
i080-001.stp	9	80	120	6	570135	571433	284126
i080-003.stp	9	80	120	6	1279586	1282396	339357
i080-004.stp	10	80	120	6	2687824	3507831	1382932
b06.stp	10	50	100	25	723512	750390	602203
I080-005.stp	11	80	120	6	25194937	31197986	6102838
b05.stp	11	50	100	13	6827453	6974959	2741485

Table 3.4: Number of generated partial solutions for instances of Steinlib (1)

results of the first part of the thesis, i.e. under the same circumstances, using the same nice tree decompositions for the graph instances.

In Tables 3.1 - 3.3, we have gathered the results for the runtimes of the three algorithms for the graph instances. For the Steinlib instances we see that ORDER is outperformed by BSA and BSC in all but a few instances, e.g. *i080-003* and *i080-005* for which we see slight improvements. This is further reflected in Tables 3.4 - 3.5 in which we see only small differences in the amount of generated partial solutions except for these instances. In the worst case ORDER takes about twice as much time. This shows that we lose too much time with the extra overhead used to generate partial solutions in ascending order of weight in the ORDER algorithm. Generally, for these instances, we do not see enough pay-off from the decrease in generated partial solution.

The results in Table 3.3 for the Treewidthlib instances seem more positive. For instances of treewidth 9 and up we see some significant improvement in running times for the ORDER algorithm. This is further reflected in Table 3.6 where we see a substantial disparity in the

amount of partial solutions that are generated. This shows that in some instances we can avoid a lot of work by generating partial solutions in ascending order of weight.

instance	$\mathtt{tw}(\mathbb{T})$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	BSA	BSC	ORDER
es90fst12.stp	5	207	284	90	17691	17704	15654
es100fst10.stp	5	229	312	100	22181	22207	19645
es80fst06.stp	6	172	224	80	31727	32307	27767
es100fst14.stp	6	198	253	100	21945	21969	19594
es90fst01.stp	7	181	231	90	30083	30124	23606
es100fst13.stp	7	254	361	100	99199	99415	86671
es100fst15.stp	8	231	319	100	100472	100490	86538
es250fst03.stp	8	543	727	250	151732	151810	120721
es100fst08.stp	9	210	276	100	84869	90006	69233
es250fst05.stp	9	596	832	250	739927	740659	622536
es250fst07.stp	10	585	799	250	1664403	1665253	1394474
es500fst05.stp	10	1172	1627	500	1790537	1791036	1486215
es250fst12.stp	11	619	872	250	3772577	3773396	3215275
es100fst02.stp	12	339	522	100	4909234	4909360	3818320
es250fst01.stp	12	623	876	250	4714990	4715219	4065129
es250fst08.stp	13	657	947	250	18954303	19507901	16379865
es250fst13.stp	13	713	1053	250	15870409	16101784	13893935

Table 3.5: Number of generated partial solutions for instances of Steinlib (2)

instance	$\mathtt{tw}(\mathbb{T})$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert T\rvert$	BSA	BSC	ORDER
myciel3.stp	5	11	20	2	1294	1346	555
BN_28.stp	5	24	49	4	1669	1699	1237
pathfinder.stp	6	109	211	21	21205	22072	12253
csf.stp	6	32	94	6	21110	21214	9322
oow-trad.stp	7	33	72	6	39317	39326	21203
mainuk.stp	7	48	198	9	202412	210410	103686
ship-ship.stp	8	50	114	10	144494	144683	88549
barley.stp	8	48	126	9	77798	84124	45446
miles250.stp	9	128	387	25	273710	278716	118480
jean.stp	9	80	254	16	292576	302643	58935
huck.stp	10	74	301	14	526946	531596	101938
myciel4.stp	11	23	71	4	1876694	3482634	188023
munin1.stp	11	189	366	37	19289463	23534527	5288937
pigs.stp	12	441	806	88	13488331	16814403	5251439
anna.stp	12	138	493	27	82060856	99551997	6436710

Table 3.6: Number of generated partial solutions for instances on graphs from TreewidthLib

3.5 Discussion and Concluding Remarks

In this paper, we have presented a new algorithm for Weighted Steiner Tree in which we evaluate partial solutions from the dynamic programming recurrence in ascending order of weight. We did this in an attempt to avoid part of the work of a straightforward dynamic

programming algorithm which calculates tables in a bottom-up fashion. We have shown that we can still use the rank based approach and bit set representation which proved to be very effective in the first part of the thesis. This also allowed us to limit the amount of space used by the algorithm. During the experimentation we have not encountered any complications caused by lack of memory for the graph instances.

In practice, when we compare the ORDER algorithm to BSA and BSC we see some mixed results. For the Steinlib instances we see mostly negative results while for the instances of Treewidthlib of sufficient treewidth we see a very postive pay-off. We end this paper with a number of suggestions for further study:

- We see a disparity in results when we consider different types of graphs. What causes the ORDER algorithm to perform well in some instances and badly in others? Can we find some properties of graph instances for which this approach works well?
- At first glance it looks like graph density and the number of terminals in a Steiner Tree instance could have some effect on the effectiveness of the algorithm. If this is the case, can we find some general rule which tells us when the ORDER algorithm performs well?
- How well would a similar algorithm for different connectivity problems, e.g. Traveling Salesman, Feedback Vertex Set perform?
- Can we increase the performance in join bags in the bottom-up dynamic programming algorithms in practice by calculating partial solutions in order of weight? For every base set of vertices we have to consider every combination of partial solutions from both children. Many of these combinations will be eliminated when we find a representative set. We could generate partial solutions in ascending order of weight until we reach the rank bound and skip all other combinations.

Bibliography

[1] Y. P. Aneja. An integer linear programming approach to the Steiner problem in graphs. *Networks*, 10:167–178, 1980.

[2] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

[3] J. E. Beasley. An algorithm for the Steiner problem in graphs. *Networks*, 14:147–159, 1984.

[4] H. L. Bodlaender. Dynamic programming algorithms on graphs with bounded tree-width. In T. Lepistö and A. Salomaa, editors, *Proceedings of the 15th International Colloquium on Automata, Languages and Programming, ICALP'88*, volume 317 of *Lecture Notes in Computer Science*, pages 105–119. Springer Verlag, 1988.

[5] H. L. Bodlaender, M. Cygan, S. Kratsch, and J. Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. In *Proceedings of the 40th International Colloquium on Automata, Languages and Programming, ICALP 2013, Part I*, volume 7965 of *Lecture Notes in Computer Science*, pages 196–207. Springer Verlag, 2013.

[6] H. L. Bodlaender and A. M. C. A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208:259–275, 2010.

[7] R. B. Borie, R. G. Parker, and C. A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7:555–581, 1992.

[8] M. Chimani, P. Mutzel, and B. Zey. Improved Steiner tree algorithms for bounded treewidth. *Journal of Discrete Algorithms*, 16:67–78, 2012.

[9] W. Cook and P. D. Seymour. Tour merging via branch-decomposition. *INFORMS Journal on Computing*, 15(3):233–248, 2003.

[10] M. Cygan, S. Kratsch, and J. Nederlof. Fast Hamiltonicity checking via bases of perfect matchings. In *Proceedings of the 45th Annual Symposium on Theory of Computing, STOC 2013*, pages 301–310, 2013.

[11] M. Cygan, J. Nederlof, M. Pilipczuk, M. Pilipczuk, J. van Rooij, and J. O. Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *Proceedings of the 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011*, pages 150–159, 2011.

[12] C. Duin. *Steiner Problems in Graphs*. PhD thesis, University of Amsterdam, Amsterdam, the Netherlands, 1993.

[13] S. Fafianie, H. L. Bodlaender, and J. Nederlof. Speeding-up dynamic programming with representative sets — an experimental evaluation of algorithms for Steiner tree on tree decompositions. Report on arXiv 1305.7448, 2013.

[14] F. V. Fomin, D. Lokshtanov, and S. Saurabh. Efficient computation of representative sets with applications in parameterized and exact algorithms. Report on arXiv 1304.4626, 2013.

[15] M. Hanan. On Steiner's problem with rectilinear distance. *SIAM J. Applied Math.*, 14:255–265, 1966.

[16] F. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*. Elsevier, 1992.

[17] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85 – 104. Plenum Press, 1972.

[18] T. Kloks. *Treewidth. Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1994.

[19] T. Koch, A. Martin, and S. Voß. Steinlib, an updated library on Steiner tree problems in graphs. Technical Report ZIB-Report 00-37, Konrad-Zuse Zentrum für Informationstechnik Berlin, 2000. http://elib.zib.de/steinlib.

[20] E. Korach and N. Solel. Linear time algorithm for minimum weight Steiner tree in graphs with bounded treewidth. Technical Report 632, Technion, Haifa, Israel, 1990.

[21] L. Lovász. Flats in matroids and geometric graphs. In *Combinatorial Surveys. Proceedings 6th Britisch Combinatorial Conference*, pages 45–86. Academic Press, 1977.

[22] D. Marx. A parameterized view on matroid optimization problems. *Theoretical Computer Science*, 410:4471–4479, 2009.

[23] B. Monien. How to find long paths efficiently. *Annals of Discrete Mathematics*, 25:239–254, 1985.

[24] N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.

[25] J. Telle and A. Proskurowski. Efficient sets in partial k-trees. *Discrete Applied Mathematics*, 44:109–117, 1993.

[26] Treewidthlib. `http://www.cs.uu.nl/people/hansb/treewidthlib`, 2004– ....

[27] J. A. Wald and C. J. Colbourn. Steiner trees, partial 2-trees, and minimum IFI networks. *Networks*, 13:159–167, 1983.

[28] D. Warme, P. Winter, and M. Zachariasen. GeoSteiner, software for computing Steiner trees. http://www.diku.dk/hjemmesider/ansatte/martinz/geosteiner/.

[29] F. Wei-Kleiner. Tree decomposition based Steiner tree computation over large graphs. Report on arXiv 1305.5757, 2013.

[30] P. Winter. Steiner problem in networks: A survey. *Networks*, 17:129–167, 1987.