Speeding up dynamic programming with representative sets: An experimental evaluation of algorithms for Steiner tree on tree decompositions

DOI:
10.1007/s00453-014-9934-0

Document status and date:
Published: 01/01/2015

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Algorithmica (2015) 71:636–660
DOI 10.1007/s00453-014-9934-0

Speeding Up Dynamic Programming with Representative Sets: An Experimental Evaluation of Algorithms for Steiner Tree on Tree Decompositions

Stefan Fafianie · Hans L. Bodlaender · Jesper Nederlof

Received: 25 November 2013 / Accepted: 16 August 2014 / Published online: 29 August 2014

Abstract Dynamic programming on tree decompositions is a frequently used approach to solve otherwise intractable problems on instances of small treewidth. In recent work by Bodlaender et al. (Proceedings of the 40th international colloquium on automata, languages and programming, ICALP 2013, part I, volume 7965 of Lecture Notes in Computer Science. Springer, Berlin, pp 196–207, 2013), it was shown that for many connectivity problems, there exist algorithms that use time linear in the number of vertices and single exponential in the width of the tree decomposition that is used. The central idea is that it suffices to compute representative sets, and that these can be computed efficiently with help of Gaussian elimination. In this paper, we give an experimental evaluation of this technique for the STEINER TREE problem. Our comparison of the classic dynamic programming algorithm and the improved dynamic programming algorithm that employs table reduction shows that the new approach gives significant improvements on the running time of the algorithm and the size of the tables computed by the dynamic programming algorithm. Thus, the rank-based approach from Bodlaender et al. (2013) does not only give significant theoretical improvements but also is a viable approach in a practical setting, and showcases the potential of exploiting the idea of representative sets for speeding up dynamic programming algorithms. Furthermore, we propose an alternative represen-

The third author is supported by the NWO Project 'Space and Time Efficient Structural Improvements of Dynamic Programming Algorithms'.

S. Fafianie · H. L. Bodlaender (✉) · J. Nederlof
Utrecht University, Utrecht, The Netherlands
e-mail: h.l.bodlaender@uu.nl

S. Fafianie
e-mail: s.fafiani@gmail.com

J. Nederlof
e-mail: jespernederlof@hotmail.com

tation of partial solutions using weighted bit strings in order to circumvent a part of the reduction step that is computationally expensive in practice. In the experimental evaluation we find that this representation yields further significant improvements. We show that the representation can also be used for the other problems fitting in the framework of Bodlaender et al. (2013).

Keywords Experimental evaluation · Algorithm engineering · Steiner tree · Treewidth · Dynamic programming · Exact algorithms

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 algorithms which are fixed-parameter tractable in the treewidth of the input graph, i.e., they run in $f(k)n^c$ time for some function f and constant c if we are given an n-vertex graph along with a tree decomposition of it of width at most k. For many problems, this gives algorithms that are linear in the number of vertices n (i.e., $c = 1$), but where $f(k)$ is 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 in the width, see e.g., [25]. A recent breakthrough was obtained by Cygan et al. [11] who showed for several *connectivity* problems like Hamiltonian Circuit, Steiner Tree, Connected Dominating Set, and many other problems, that these can be solved in time single exponential in the width. However, this is at the cost of introducing randomization and an additional factor in the running time that is polynomial in n. More 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 efficiently handle weighted extensions of the problems, 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 idea of the rank-based approach is that each step where a table for a bag of the tree decomposition is computed is followed by an step where several entries of this table are deleted. These entries can be seen to be unnecessary for finding an optimal solution. This latter step is an application of Gaussian elimination, and the size of the resulting table is bounded by the rank of a certain matrix, hence the name *rank-based* approach.

In this paper, we give an experimental evaluation of the rank-based approach for the Steiner Tree problem. We will see that it gives a significant speed-up; we expect that similar speed-ups will be achieved when the approach is used for other problems, like Hamiltonian Circuit or Connected Dominating Set.

Dynamic programming and the rank-based approach: a high-level description We will now give a high level description of the standard dynamic programming algorithm for Steiner Tree on tree decompositions, and the improvement with the rank-based approach.

In the Steiner Tree problem, we are given an undirected graph $G = (V, E)$, a subset of the vertices $K \subseteq V$ (called the *terminals*), and a weight for each edge. A subtree of G that includes all the terminals is called a Steiner tree; the problem is to find the minimum weight of a Steiner tree.

In our algorithms, we assume that we have a nice tree decomposition of G. (Details on nice tree decompositions can be found in Sect. 2.) To each bag i of the nice tree decomposition (with vertex set X_i), we associate a graph G_i; G_i is the subgraph of G, formed by all vertices that belong to a bag that is a descendant of i (including i itself), and all edges that are introduced in a bag that is a descendant of i (again, including i itself). A Steiner tree T restricted to a subgraph G_i describes a *partial solution*, which can be defined as the intersection of T and G_i, i.e., all vertices and edges of T that belong to G_i. Thus, we can associate with each bag i a collection of partial solutions. A partial solution thus must be a forest in G_i fulfilling a number of conditions (discussed in detail later in the paper.) If Steiner tree T contains partial solution F, we also say that T extends F. The set of edges in T that do not belong to F is said to be an *extension* of F.

To partial solutions, we associate their weight (the total sum of the weights of the edges in the forest) and a *characteristic*. The characteristic describes which vertices in bag X_i belong to the forest and how these are connected in the forest. Now, if we have two partial solutions with the same characteristic with costs c_1 and c_2, then if we can extend the first to a Steiner tree with cost $c_1 + \alpha$, the same extension can be applied to the second partial solution to obtain a Steiner tree with cost $c_2 + \alpha$. This gives the main idea of the standard dynamic programming algorithm: it is sufficient and possible to compute for each bag a table, with for each characteristic the minimum weight over all partial solutions with that characteristic. We compute these table bottom-up in the tree (e.g., in postorder): to compute a table for a bag, we use the information local in the bag and the tables of its children.

In the rank based approach, each step where we compute a table for a bag is followed by a *reduce* step. The main observation is the following. Suppose we have a characteristic c in our table, representing some partial solution s. Suppose that for each extension e of s to a Steiner tree T, there is another characteristic c' for a partial solution s' with equal or smaller cost compared to s, such that e applied to s' also gives a Steiner tree, say T'. We directly observe that the cost of T' is equal to or smaller than the cost of T. Thus, we see that we do not need c for finding the optimum weight of a Steiner tree, and can delete it from the table computed by dynamic programming. (For formal proofs, we refer to the paper by Bodlaender et al. [5].) This idea leads to the notion of *representativity*, pioneered by Monien [23] (see also Sect. 3.2).

To find small representative sets, we consider the matrix M with rows indexed by partial solutions, and columns indexed by extensions of partial solutions, with a 1 if the combination gives a full solution (i.e., Steiner tree), and a 0 otherwise. Rows of partial solutions with the same characteristic can be seen to be equal. Similarly, we can define characteristics of extensions, such that extensions with the same characteristic correspond to equal columns. The following two facts form the basis of the rank-based approach. First, a maximal subset of linear independent rows of minimal cost forms a representative set. Second, the rank of this matrix M is bounded by $3^{|X_i|}$, for the Steiner tree problem, i.e., single exponential in the width of the tree decomposition. (Similar bounds, single exponential in the bag size / treewidth can be shown for other problems, see [5] and [10].) In both cases, we compute in $GF(2)$.

Now, if we have an explicit basis of M (the characteristics of the columns in a maximal set of independent columns in T), we can find a representative set of size bounded by the rank of M, just by performing Gaussian elimination on a submatrix of M, with the rows corresponding to the entries in the table, and the columns corresponding to the elements of the basis.

It has been shown [5,10] that the approach sketched above for the Steiner Tree problem gives for several connectivity problems on graphs deterministic algorithms that are single exponential in the treewidth of the graph. In this paper, we report on an experimental evaluation of this approach for the Steiner Tree problem.

Previous work Concerning previous work related to the rank-based approach, as mentioned before, the notion of representative sets was pioneered by Monien [23]. Using the well-known two families theorem by Lovász [21], it is possible to obtain efficient FPT algorithms for several other problems in a way that is similar to ours [14,22]. 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 and Long Path.

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 a consequence of a general characterization by Bodlaender [4] and linear time algorithms are obtained as a 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, the tables are typically sparse, 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.

This paper As said, 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 reduction of the running 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, namely the construction of certain matrices, that is expensive in practice. Again, the experimental evaluation of this bit string representation shows very positive results.

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, representing the minimum weight of a partial solution that has this partition of a subset as characteristic.
- The rank-based approach (RBA): To the classic dynamic programming algorithm, we add a step where we apply the *reduce* algorithm from [5]. This elimination step is performed each time after the DP algorithm has computed a table for a node of the nice tree decomposition.
- The rank-based combined approach (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.
- The rank-based bit string approach (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.
- The rank-based bit string combined approach (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 Sect. 2. In Sect. 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 Sect. 4, we describe the setup of our experiments, and in Sect. 5, we discuss the results of the experiments. Some final conclusions are given in Sect. 6.

2 Preliminaries

We use standard graph theory notation and additional notation taken 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. $G[X] = (V(X), X)$ and $V(G)$ denote the vertex set of G. Let $\Pi(U)$ denote the set of all partitions of some set U. Given $p \in \Pi(U)$ we let $\#\texttt{blocks}(p)$ denote the number of blocks of p. If $X \subseteq U$ we let $p_{\downarrow X} \in \Pi(X)$ be the partition obtained by removing all elements not in X from it, and analogously we let for $U \subseteq X$ denote $p_{\uparrow X} \in \Pi(X)$ for the partition obtained by adding singletons for every element in $X \setminus U$ to p. Also, for $X \subseteq U$, we let $U[X]$ be the partition of U where one block is $\{X\}$ and all other blocks are singletons. If $a, b \in U$ we shorthand $U[ab] = U[\{a, b\}]$. If $\omega : U \to \mathbb{N}$ and $X \subseteq U$, we let $\omega(X)$ denote $\sum_{e \in X} \omega(e)$.

Formally a partition of a ground set S is a family of pair-wise disjoint subsets of S whose union equals S. In this paper, we will shorthand the trivial partition in one set (i.e., $\{S\}$, if the ground set is S), with S itself (i.e., the name of the ground set in general). 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. We will shorthand this by $p \sqsubseteq q$. We let $p \sqcap q$ denote the finest partition that is a coarsening of p and of q.[1]

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.

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.

[1] The notation is motivated from lattice theory; it can be observed that the set of all partitions of a set partially ordered by $\sqsubseteq$ is a lattice (i.e., the inverse of the usual *partition lattice*), and that $\sqcap$ is the meet operation of this lattice.

– *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 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 an empty forget bag can be computed in $n \cdot \texttt{tw}^{\mathcal{O}(1)}$ 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 can also assume that the 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.

The Steiner Tree problem studied in this paper 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 $\texttt{tw}$ forgetting a terminal.
Question: The minimum of $\omega(X)$ over all subsets $X \subseteq E$ of G such that $G[X]$ is connected and $K \subseteq V(G[X])$.

As outline above, the requirements on the tree decomposition can be easily relaxed.

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

$$\texttt{glue}(uv, \mathcal{A}) = \texttt{rmc}\left(\left\{(\hat{U}[uv] \sqcap p_{\uparrow \hat{U}}, w) \;\middle|\; (p, w) \in \mathcal{A}\right\}\right).$$

Also, if $\omega : \hat{U} \times \hat{U} \to \mathbb{N}$, let $\texttt{glue}_\omega(uv, \mathcal{A}) = \texttt{shft}(\omega(u, v), \texttt{glue}(uv, \mathcal{A}))$. *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 $\texttt{proj}(X, \mathcal{A}) \subseteq \Pi(\overline{X}) \times \mathbb{N}$ as

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

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 $\texttt{join}(\mathcal{A}, \mathcal{B}) \subseteq \Pi(\hat{U}) \times \mathbb{N}$ as

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

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 $\texttt{rmc}(\mathcal{A}) = \{(p, w) \in \mathcal{A} \mid \nexists (p, w') \in \mathcal{A} : w' < w\}$ 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.

3 Dynamic Programming Algorithms for Steiner Tree Parameterized by Treewidth

In this section we describe 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. We then give a detailed description of the weighted bit string representation of partial solutions.

3.1 Classic Dynamic Programming

We follow the description from [5]. In order to facilitate the correctness proof and the description of the algorithms we will use the operators from Definition 3 here, and thus obtain compact descriptions of the recurrences (for the different types of nodes in the nice tree decomposition) that shape the dynamic programming algorithm.

For each node in the nice tree decomposition, we compute a table. Each table entry maps a partition of a subset of the bag to an integer value. We now will introduce the notation, and give the corresponding recurrences (with just brief sketches for their correctness).

We will denote the weighted partition tables as $A_x(\cdot)$ and sets of partial solutions as $\mathcal{E}_x(\cdot)$ where x denotes the current bag. The classic dynamic programming algorithm computes for each bag x the function A_x. This function is stored in a table, with only trivial entries (e.g., partitions mapping to infinity, as there are no forests corresponding to the partition) not stored.

We use $S \subseteq B_x$ to describe which vertices belong to the tree. For a bag x and $S \subseteq B_x$ define:

$$A_x(S) = \left\{ \left(p, \min_{X \in \mathcal{E}_x(p,S)} \omega(X) \right) \middle| p \in \Pi(S) \wedge \mathcal{E}_x(p,S) \neq \emptyset \right\}, \text{ where}$$

$$\begin{aligned}\mathcal{E}_x(p,S) = \Big\{ X \subseteq E_x \;\Big|\; & V(G[X]) \cap B_x = S \wedge V_x \cap K \subseteq V(G[X]) \\ & \wedge \forall v_1, v_2 \in S : v_1, v_2 \text{ are in same block in } p \leftrightarrow v_1, v_2 \text{ connected in } G[X] \\ & \wedge \#\texttt{blocks}(p) = \texttt{cc}(G[X]) \Big\}\end{aligned}$$

In words, $A_x(S)$ is all pairs (p, w) where p is a partition of S, and w is the minimum weight to connect the terminals $K \cap V_x$ and the vertices in S according to p using only edges in E_x.

In the definition of $\mathcal{E}_x(p, S)$, partial solutions have (a subset of) S as incident vertices in B_x connected according to the partition p. The blocks in p represent connected components in the partial solution. When two vertices are in the same block they belong to the same connected component. Naturally, any terminal $v \in K$ has to be used in a partial solution and we are allowed to use other vertices to connect these terminals. Connected components are formed by using subsets $X \subseteq E_x$ of edges and multiple different subsets can form the same partition in a partial solution. In the partial solution tables $A_x(S)$ we only consider minimum weight partial solutions and discard any other partial solutions that are dominated. If we have a tree decomposition $\mathbb{T}$ such that its root is a forget vertex bag for $v \in K$ as input for Steiner Tree, then this root has one child y with one entry in $A_y(S)$ where $v \in S$. There are no other vertices in this bag since the root bag is empty. Therefore $A_y(S)$ only contains the partition $p = \{\{v\}\}$ in which the single block represents a single connected component containing all terminals with minimum weight over all possible subsets of edges, thus yielding the solution.

We proceed with the recurrence for $A_x(S)$ which is used by the classic dynamic programming algorithm. In order to simplify the notation, let v denote the vertex introduced and contained in an introduce bag, and let y, z denote the left and right children of x in $\mathbb{T}$, if present. We let U respectively U' denote the base set of vertices present in y and z. We distinguish on the type of bag in $\mathbb{T}$. For a leaf bag x let:

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

This is the trivial case, where $\mathcal{E}_x(p, S)$ only contains the empty set, which does not contain or connect any vertices and has weight 0.

For an introduce vertex v bag x with child y let:

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

For each partial solution in $A_y(S)$ we consider whether or not to use v and add both cases (when feasible) to A_x to fill our table for introduce vertex bag x. Using

v corresponds to $v \in S$, and because v was just introduced and thus is currently an isolated vertex, we insert it as a singleton into each partition. If we do not use v, i.e, $v \notin S$ then we do not insert v and preserve the same partial solution as in the child bag. If v is a terminal, then not inserting v is not feasible.

For a forget vertex v bag x with child y let:

$$A_x(S) = A_y(S) \Cup \mathtt{proj}(v, A_y(S \cup v))$$

We assume that x is not the root. The procedure basically does two steps: if v is forgotten, then any partition in which v is used and is a singleton gives more than one connected component. (Recall here that the root bag forgets a terminal, and here v cannot be connected to that terminal vertex.) All such entries are deleted. All other entries are 'projected', i.e., v is removed from the partitions. Possibly, multiples entries have the same projection; then we keep the one with the smallest value.

For an introduce edge $e = (u, v)$ bag x with child y let:

$$A_x(S) = \begin{cases} A_y(S), & \text{if } \{u, v\} \nsubseteq S \\ A_y(S) \Cup \mathtt{glue}_\omega(uv, A_y(S)), & \text{otherwise.} \end{cases}$$

For each partial solution in $A_y(S)$ we consider whether or not to include e and add both cases (when feasible) to A_x to fill our table for introduce edge bag x. If we include an edge in a partial solution then we must ensure that u and v are used in the partition i.e. $u, v \in S$. Including the edge increases the weight of the partial solution by $\omega(u, v)$ and connects the connected components containing v respectively u, and thus, we combine their blocks in the new partial solution. Again, if we do not include e, the partial solution remains the same. Because v and u may already have been part of the same connected component we must eliminate dominated partial solutions.

For join bag x with children y and z let:

$$A_x(S) = \mathtt{join}(A_y(S), A_z(S))$$

Here we combine choices previously made in the subtree of y with choices made in the subtree of z, by combining pairs of partial solutions. We account for the weight by adding their respective weights. Using edges from both partial solutions may merge connected components, so we join their connectivity. This may again result in multiple partitions of different weight, of which we keep the minimum weight. This concludes the formulation of the recurrence for the classic dynamic programming algorithm.

The algorithm now can be expressed as follows: in bottom-up order for each bag x we compute A_x, and finally computes the minimum weight of a Steiner Tree by inspection the information for the root bag, as discussed above.

3.2 Rank-Based Table Reductions

In this section, we describe the rank-based approach from [5]. The main idea is that after we have computed a table for a bag 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. If a set of partitions extends to an optimal solution then we should also be able to extend to an optimal solution from the representative set. Representation is formally defined as:

Definition 4 (*Representation*) Given a set of weighted partitions $\mathcal{A} \subseteq \Pi(U) \times \mathbb{N}$ and a partition $q \in \Pi(U)$, define:

$$\texttt{opt}(q, \mathcal{A}) = \min\{w \mid (p, w) \in \mathcal{A} \wedge p \sqcap q = \{U\}\}$$

For another set of weighted partitions $\mathcal{A}' \subseteq \Pi(U) \times \mathbb{N}$, we say that $\mathcal{A}'$ *represents* $\mathcal{A}$ if for all $q \in \Pi(U)$ it holds that $\texttt{opt}(q, \mathcal{A}') = \texttt{opt}(q, \mathcal{A})$.

Intuitively, partitions store connectivity of partial solutions and in order for two partial solutions to combine to a global solutions, the meet of the two corresponding partitions need to be the trivial partition. Then $\texttt{opt}(q, \mathcal{A})$ is the minimum weight over all partial solutions from $\mathcal{A}$ that combine with q.

Although this definition is symmetric, we will only be interested in finding $\mathcal{A}'$ where $\mathcal{A}' \subseteq \mathcal{A}$ and where we have a size guarantee such that $\mathcal{A}'$ is small. Omitting the formal proof (see [5]), we now state that the functions describing the formulation of the recurrence in Sect. 3.1 preserve representation:

Definition 5 (*Preserving representation*) A function $f : 2^{\Pi(U)\times\mathbb{N}} \times Z \to 2^{\Pi(U')\times\mathbb{N}}$ is said to *preserve representation* if for every $\mathcal{A}, \mathcal{A}' \subseteq \Pi(U) \times \mathbb{N}$ and $z \in Z$ it holds that if $\mathcal{A}'$ represents $\mathcal{A}$ then $f(\mathcal{A}', z)$ represents $f(\mathcal{A}, z)$.

We consider possible extensions of partial solutions, e.g., for Steiner Tree we consider forests in $G \setminus G_x$ that can extend partial solutions for bag x into a spanning tree. Similar to partial solutions, the connectivity of these extensions can be denoted with a partition.

At the core of the rank-based approach, the key to obtaining a small representative set is to find for partitions q the minimum weight of partial solutions $(p, w) \in \mathcal{A}$ such that $p \sqcap q = \{U\}$. So if we can find a set cover of partitions p with minimum weight for every q with this property, then we have a representative set, since when they can all extend to the unit partition, then one must also extend to the optimal solution. We can achieve this by finding a basis of minimum weight in the matrix $\mathcal{M} \in \mathbb{Z}_2^{\Pi(U)\times\Pi(U)}$ where $\mathcal{M}[p, q] = 1$ if $p \sqcap q = \{U\}$ and $\mathcal{M}[p, q] = 0$ otherwise. In arithmetic modulo two we can rewrite this matrix as a product of two cut-matrices $\mathcal{C}$ defined as:

Definition 6 Define $\texttt{cuts}(U) := \{(V_1, V_2) \mid V_1 \cup V_2 = U \wedge 1 \in V_1\}$, where 1 stands for an arbitrary but fixed element of U. Define $\mathcal{C} \in \mathbb{Z}_2^{\Pi(U)\times\texttt{cuts}(t)}$ by $\mathcal{C}[p, (V_1, V_2)] = 1$ if $(V_1, V_2) \sqsubseteq p$ and $\mathcal{C}[p, (V_1, V_2)] = 0$ otherwise.

We now can see that $\mathcal{M} \equiv \mathcal{C}\mathcal{C}^T$ and because of linear dependencies we are allowed to use the lightest (i.e., with minimal weights) basis of the cut-matrix $\mathcal{C}$ as the representative subset $\mathcal{A}' \subseteq \mathcal{A}$ where $|\mathcal{A}'| \leq 2^{|U|}$. We can find this basis via straightforward Gaussian elimination in $\mathcal{C}[\mathcal{A}, \cdot]$ after we order its rows by weight.

This yields the improved algorithm for solving Steiner Tree: for each node in the tree of the nice tree decomposition, in bottom-up order, we compute a table and then reduce the size of this intermediate table by the reduce algorithm.[2] The computation of the table uses the same recurrences as for A_x, but as inputs we use the reduced tables for the children, i.e., we restrict the domains—in this way, we obtain for each node a table whose entries are 'representative' for A_x since the recurrence only use the operators from Definition 3, which preserve representation as mentioned above. For a formal proof, we refer the reader to [5].

We have two variants: we can choose to always apply the reductions, or to apply them only in some cases. 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 $|\mathcal{A}| \geq 2^{|U|}$. 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.

3.3 Representing Partial Solutions with Weighted Bit Strings

When we first performed our experimental evaluation [13], we found that during the reduction steps most time is spent calculating the entries of cut matrices. 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 cut matrices. More precisely, we *identify* partial solutions not with help of partitions, but directly by the row elements of cut matrices. Every partial solution is part of a set of partial solutions with partitions that are based on the same set W. During the reduction step a cut-matrix is calculated for W in which each partial solution has a corresponding row. The partition can thus be implicitly represented by this row. This new representation allows us to calculate rows in cut matrices for parent nodes directly from rows in cut matrices 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 Bodlaender 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 cut matrix 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.

Consider a set of weighted bit strings $A_x(W)$ for bag x that represent partial solutions using vertices in W. Let $(s, w) \in A_x(W)$ be a pair consisting of a bit string s directly representing a row in a cut matrix (i.e. corresponding to a partition of W) and w be an integer (referred to as its weight). Let $l(s) = 2^{|W|-1}$ denote the length of

[2] See the proof of Theorem 3.7 in the arXiv report of [5].

this bit string and let $s_i \in \{0, 1\}$ denote the value of the bit at index $i \in [0..l-1]$. If the partial solution represented by (s, w) does not use any vertices, i.e. $W = \emptyset$, then s is an empty bit string. 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 $W = \{v_0, \dots, v_{|W|-1}\}$ on the vertices in W. Now let $\texttt{cuts}(W) = \{c_0, \dots, c_{l-1}\}$ be cuts corresponding to index i in the bit string. Intuitively, at some point during the dynamic 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 right-hand side of the cuts represented by columns in the previous table, i.e.

$$\begin{aligned}
\texttt{cuts}(\{v_0\}) &= ((v_0 \mid \emptyset)) \\
\texttt{cuts}(\{v_0, v_1\}) &= ((v_0, v_1 \mid \emptyset), (v_0 \mid v_1)) \\
\texttt{cuts}(\{v_0, v_1, v_2\}) &= ((v_0, v_1, v_2 \mid \emptyset), (v_0, v_1 \mid v_2), (v_0, v_2 \mid v_1), (v_0 \mid v_1, v_2)) \\
&\texttt{etc.}
\end{aligned}$$

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 Sect. 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 $\mathcal{A} \uplus \mathcal{B} = \texttt{rmc}(\mathcal{A} \cup \mathcal{B})$. 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 cuts$(v_0, \dots, v_j)$ which are identical except for the side on which v_j is fixed. When we expand cuts$(v_0, \dots, v_j)$ to cuts(W') these pairs are at a distance of $d = 2^{|W'|-1-j}$ apart since we expand for $|W'| - 1 - j$ more vertices, each time fixing a vertex left or right. These pairs are packed in blocks of size $b = 2^{|W'|-j}$ (see Fig. 1). We calculate

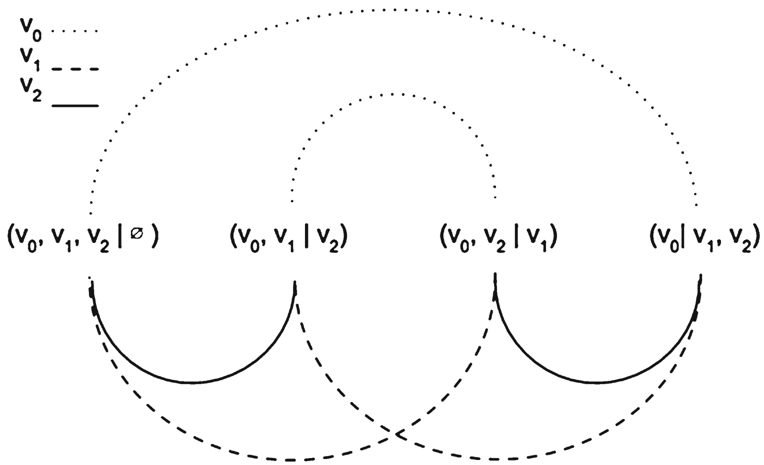

Fig. 1 Emerging pattern in ordered cuts. The *edges* depict cuts that are identical if the corresponding vertex is left out

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*

$$\texttt{ins}(v_j, \mathcal{A}) = \left\{(s', w) \mid (s, w) \in \mathcal{A} \wedge s'_{p \cdot b+k} = s'_{p \cdot b+k+d} = s_i\right\} \text{where}$$
$$b = 2^{|W'|-j}, d = \frac{b}{2}, p = i/d \,\text{and}\, k = i \mod d, \ \forall i \in [0..l(s) - 1]$$

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 $\text{cuts}(\{v_0\}) = (v_0 \mid \emptyset)$ starts out asymmetrically (see Fig. 1), i.e.

– *For a single element* $v_j \in W' = W \cup \{v_j\}$ *where* $v_j = v_0$, *define*

$$\texttt{ins}(v_j, \mathcal{A}) = \left\{(s', w) \mid (s, w) \in \mathcal{A} \wedge s'_i = s'_{l(s')-i-1} = s_i\right\}, \ \forall i \in [0..l(s) - 1]$$

When we insert v_0 while $W = \emptyset$ there is a single cut, i.e. $\texttt{cuts}(W') = (v_0 \mid \emptyset)$. Partial solutions based on $W' = \{v_0\}$ are always a refinement of this cut.

– *For a single element* $v_j \in W' = W \cup \{v_j\} = \{v_j\}$, *define*

$$\texttt{ins}(v_j, \mathcal{A}) = \left\{(s', w) \mid (s, w) \in \mathcal{A} \wedge s'_0 = 1\right\}$$

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*

$$\texttt{ins}(X, \mathcal{A}) = \left\{\texttt{ins}(X \setminus \{x\}, \texttt{ins}(x, \mathcal{A})\right\}$$

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 $W' = W \setminus \{v\}$. 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 $(W\setminus\{v\} \mid v)$. Suppose we project $v_j \in W$. Assuming our invariant holds we can find the bit corresponding to this particular cut at index $2^{|W|-j-1}$ if $v_j \neq v_0$ and at index $l(s) - 1$ otherwise. Note that partial solutions will always be eliminated in the case that $W = \{v_j\}$. Therefore we will not see new empty bit strings as the result of the project operator. 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*

$$\begin{aligned}\texttt{proj}(v_j, \mathcal{A}) = \texttt{rmc}\,&\left(\left\{(s', w) \;\middle|\; (s, w) \in \mathcal{A} \wedge \neg\texttt{singleton}(v_j, s)\right.\right.\\ &\left.\left.\wedge s'_i = s_{p\cdot b+k} \text{ OR } s_{p\cdot b+k+d}\right\}\right) \text{ where } b = 2^{|W|-j},\, d = \frac{b}{2},\, p = i/d\\ &\text{and } k = i \mod d, \forall i \in [0..l(s) - 1]\end{aligned}$$

– *For a single element $v_j \in W$ where $v_j = v_0$, define*

$$\begin{aligned}\texttt{proj}(v_j, \mathcal{A}) = \texttt{rmc}\,&\left(\left\{(s', w) \;\middle|\; (s, w) \in \mathcal{A} \wedge \neg\texttt{singleton}(v_j, s)\right.\right.\\ &\left.\left.\wedge s'_i = s_i \text{ OR } s_{l(s)-i-1}\right\}\right),\ \forall i \in [0..l(s) - 1]\end{aligned}$$

– *For a single element v_j and bit string s define*

$$\texttt{singleton}(v_j, s) = \begin{cases}\text{true} & v_j \neq v_0 \wedge s_{2^{|W|-j-1}} = 1\\ \text{true} & v_j = v_0 \wedge s_{l-1} = 1\\ \text{false} & \text{otherwise.}\end{cases}$$

– **Project** *For $X \subseteq W$ and $x \in X$, define*

$$\texttt{proj}(X, \mathcal{A}) = \left\{\texttt{proj}(X \setminus \{x\}, \texttt{proj}(x, \mathcal{A}))\right\}$$

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

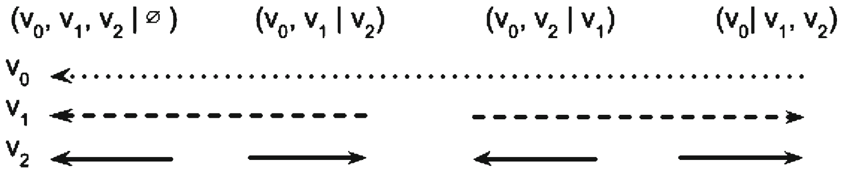

Fig. 2 Emerging pattern in ordered cuts. The *arrows* depict on which side of the cuts the corresponding vertex is fixed

at least one block b in either partition with vertices in both the left- and right-hand 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 $c \not\sqsubseteq z$. Conversely, if $c \sqsubseteq p$ and $c \sqsubseteq q$ then each block in p and q is contained either completely in the left- or right-hand side of the cut. Joining the connectivity would not result in blocks containing vertices from both sides. Therefore z is a refinement of c, i.e. $c \sqsubseteq z$ if and only if $c \sqsubseteq p$ and $c \sqsubseteq q$.[3] 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*

$$\begin{aligned}\texttt{join}(\mathcal{A}, \mathcal{B}) = \texttt{rmc}\Big(\Big\{(s, w^a + w^b) \;\Big|\; s_i = (s_i^a \text{ AND } s_i^b) \\ \wedge (s^a, w^a) \in \text{ins}(W' \setminus W, \mathcal{A}) \wedge (s^b, w^b) \in \text{ins}(W \setminus W', \mathcal{B})\Big\}\Big)\end{aligned}$$

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 $2^{|W|-j}$ cuts where vertex v_j is fixed to the left side and then on the right (see Fig. 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*

$$\begin{aligned} s(v_j v_k)_i &= l_i^{v_j} \text{ XNOR } l_i^{v_k} \text{ where} \\ l_i^{v_j} &= \begin{cases} 1, & i \mod b_{v_k} < \frac{b_{v_k}}{2} \\ 0, & \text{otherwise.} \end{cases}, \\ l_i^{v_k} &= \begin{cases} 1, & i \mod b_{v_j} < \frac{b_{v_j}}{2} \\ 0, & \text{otherwise.} \end{cases}, \\ b_{v_j} &= 2^{|W|-j} \text{ and } b_{v_k} = 2^{|W|-k}. \end{aligned}$$

[3] In fact, for any lattice we have $c \sqsubseteq p \sqcap q$ if and only if $c \sqsubseteq p$ and $c \sqsubseteq q$.

This gives us a bit string where the bit $s(v_j v_k)$ is set to 1 if both v_j and v_k are completely contained in either the left- or right-hand side of the corresponding cut indexed by i. We then use this bit string in the adaptation of the glue operator.

– **Glue** *For* $v_j, v_k \in W$, *define*

$$\mathtt{glue}(v_j v_k, \mathcal{A}) = \mathtt{rmc}\left(\left\{(s', w) \mid (s, w) \in \mathcal{A} \wedge s'_i = s_i \text{ AND } s(v_j v_k)_i\right\}\right)$$

Finally we trivially adapt the shift operator.

– *For* $w' \in \mathbb{N}$, *define*

$$\mathtt{shft}(w', \mathcal{A}) = \left\{(s, w + w') \mid (s, w) \in \mathcal{A}\right\}$$

And the glue with weight operator.

– *For* $\omega : W' \times W' \to \mathbb{N}$ *where* $W' = W \cup \{v_j, v_k\}$, *define*

$$\mathtt{glue}_\omega(v_j v_k, \mathcal{A}) = \mathtt{shft}(\omega(v_j, v_k), \mathtt{glue}(v_j v_k, \mathcal{A}))$$

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 cut matrices more efficiently.

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 Sect. 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 bit string. 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 were 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. For a new partial solution (p, w) we use the outer hash-map to find one of the inner hash-maps, in which we can check if a partial solution that has the same partition p is already present, and if so, what its weight is. We then decide whether or not (p, w) should be inserted into this inner hash-map as (key, value) pair. 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 of this hash code 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). Note that 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 it 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.

5 Experimental Results

In this section, we will report the results for experiments with the algorithms discussed in Sect. 3. We will compare the runtime of the five earlier introduced algorithms CDP, RBA, RBC, BSA and BSC. 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 coloring applications. We transformed these to STEINER TREE instances by adding random edge weights between 1 and 1,000, 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 2 h 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 *.

In Tables 1, 2 and 3, we have gathered the results for the run-times of the five algorithms for the aforementioned graph instances. We immediately notice that RBC outperforms RBA in all cases. In Tables 4, 5 and 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 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

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

Instances	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	1,552	548	892	95	240
b14.stp	7	100	125	25	2,001	515	336	43	32
b15.stp	8	100	125	50	15,860	1,695	1,503	161	169
i080-001.stp	9	80	120	6	477,716	13,386	9,279	1,571	1,251
i080-003.stp	9	80	120	6	1,996,394	21,598	19,250	3,077	3,019
i080-004.stp	10	80	120	6	2,283,606	74,845	74,464	14,464	18,197
b06.stp	10	50	100	25	1,449,534	36,041	28,389	6,021	5,379
I080-005.stp	11	80	120	6	*	815,457	723,720	236,683	293,567
b05.stp	11	50	100	13	*	341,862	275,824	137,917	118,226

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

Instances	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	1,594	1,351	804	112	84
es100fst15.stp	8	231	319	100	2,069	1,470	826	120	101
es250fst03.stp	8	543	727	250	3,320	2,343	1,484	206	162
es100fst08.stp	9	210	276	100	5,088	2,588	2,165	309	321
es250fst05.stp	9	596	832	250	35,961	14,521	8,322	1,550	1,109
es250fst07.stp	10	585	799	250	127,681	60,701	37,042	7,508	5,942
es500fst05.stp	10	1,172	1,627	500	145,408	51,504	34,684	5,972	4,933
es250fst12.stp	11	619	872	250	*	138,073	99,427	23,311	20,045
es100fst02.stp	12	339	522	100	*	365,800	299,014	150,013	143,582
es250fst01.stp	12	623	876	250	*	395,694	288,476	105,810	91,650
es250fst08.stp	13	657	947	250	*	2,469,463	2,208,040	1,257,730	1,236,192
es250fst13.stp	13	713	1,053	250	*	2,725,460	2,416,867	1,684,224	1,557,617

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 preferred strategy is less clear, since we inherently perform part of the reduction step, i.e. the filling of cut matrices, during the table calculations.

Table 3 Runtime in milliseconds for instances on graphs from TreewidthLib

Instances	tw($\mathbb{T}$)	$\|V\|$	$\|E\|$	$\|T\|$	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	1,135	254	165	19	15
oow-trad.stp	7	33	72	6	803	601	371	50	36
mainuk.stp	7	48	198	9	10,040	3,925	2,444	291	214
ship-ship.stp	8	50	114	10	6,015	3,929	2,465	352	254
barley.stp	8	48	126	9	3,000	1,836	1,248	168	142
miles250.stp	9	128	387	25	37,745	14,099	8,444	1,761	1,291
jean.stp	9	80	254	16	17,988	20,404	9,231	1,907	1,175
huck.stp	10	74	301	14	18,652	37,696	20,376	3,829	2,657
myciel4.stp	11	23	71	4	1,602,408	86,183	83,358	16,385	23,824
munin1.stp	11	189	366	37	*	521,081	501,164	162,717	227,469
pigs.stp	12	441	806	88	*	4,155,602	3,808,347	2,835,576	2,878,242
anna.stp	12	138	493	27	*	5,515,952	4,822,620	2,357,740	2,398,758

Table 4 Number of generated partial solutions for instances of Steinlib (1)

Instances	tw($\mathbb{T}$)	$\|V\|$	$\|E\|$	$\|T\|$	CDP	RBA	RBC
b01.stp	4	50	63	9	1,921	1,654	1,654
b02.stp	4	50	63	13	1,948	1,628	1,638
b08.stp	6	75	94	19	99,740	11,654	12,005
b09.stp	6	75	94	38	18,615	5,302	5,302
b13.stp	7	100	125	17	279,852	47,032	58,717
b14.stp	7	100	125	25	318,744	37,406	38,146
b15.stp	8	100	125	50	2,248,833	76,681	93,161
i080-001.stp	9	80	120	6	65,460,491	570,132	571,425
i080-003.stp	9	80	120	6	249,390,279	1,279,544	1,282,358
i080-004.stp	10	80	120	6	256,761,016	2,687,590	3,507,987
b06.stp	10	50	100	25	151,246,080	723,392	754,926
I080-005.stp	11	80	120	6	*	25,194,893	29,825,246
b05.stp	11	50	100	13	*	6,827,459	6,955,686

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 min to find the optimal solution, but RBC uses just 74 s. Furthermore we see a strong increase in the runtime difference when the width of the tree decompositions increases. This is further reflected in Table 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,

Table 5 Number of generated partial solutions for instances of Steinlib (2)

Instances	tw($\mathbb{T}$)	$\|V\|$	$\|E\|$	$\|T\|$	CDP	RBA	RBC
es90fst12.stp	5	207	284	90	25,817	17,693	17,706
es100fst10.stp	5	229	312	100	34,612	22,181	22,204
es80fst06.stp	6	172	224	80	73,436	31,721	32,301
es100fst14.stp	6	198	253	100	35,664	21,947	21,971
es90fst01.stp	7	181	231	90	137,705	30,097	30,139
es100fst13.stp	7	254	361	100	323,259	99,203	99,420
es100fst15.stp	8	231	319	100	388,118	100,469	100,487
es250fst03.stp	8	543	727	250	593,651	151,722	151,802
es100fst08.stp	9	210	276	100	724,207	84,869	90,006
es250fst05.stp	9	596	832	250	5,283,073	739,953	740,698
es250fst07.stp	10	585	799	250	15,397,120	1,664,352	1,665,205
es500fst05.stp	10	1,172	1,627	500	17,953,689	1,790,843	1,791,361
es250fst12.stp	11	619	872	250	*	3,771,954	3,772,893
es100fst02.stp	12	339	522	100	*	4,909,388	4,909,500
es250fst01.stp	12	623	876	250	*	4,715,125	4,715,631
es250fst08.stp	13	657	947	250	*	18,954,259	19,509,166
es250fst13.stp	13	713	1,053	250	*	15,870,380	16,101,777

Table 6 Number of generated partial solutions for instances on graphs from TreewidthLib

Instances	tw($\mathbb{T}$)	$\|V\|$	$\|E\|$	$\|T\|$	CDP	RBA	RBC
myciel3.stp	5	11	20	2	2,382	1,295	1,347
BN_28.stp	5	24	49	4	2,346	1,670	1,700
pathfinder.stp	6	109	211	21	128,163	21,206	22,073
csf.stp	6	32	94	6	206,434	21,111	21,215
oow-trad.stp	7	33	72	6	164,723	39,318	39,327
mainuk.stp	7	48	198	9	1,691,584	202,454	210,694
ship-ship.stp	8	50	114	10	1,093,800	144,493	144,682
barley.stp	8	48	126	9	472,223	77,799	84,125
miles250.stp	9	128	387	25	5,524,562	273,711	278,717
jean.stp	9	80	254	16	2,932,817	292,577	302,644
huck.stp	10	74	301	14	3,238,678	526,947	531,597
myciel4.stp	11	23	71	4	203,990,952	1,876,695	3,482,635
munin1.stp	11	189	366	37	*	19,289,467	23,535,116
pigs.stp	12	441	806	88	*	164,037,075	169,483,545
anna.stp	12	138	493	27	*	82,060,857	99,551,566

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 *i080-004* we now see that BSA uses just 14 s and BSC uses 17 s.

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 were 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 Gaussian 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.

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 is already 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 the cut matrix $\mathcal{C}$. This implementation yields even further significant improvements on the running time. In addition, as we have shown in Sect. 3.3, this new representation of partial solutions using weighted bit strings can not only be used for the STEINER TREE problem, but also for the other problems that fit in the framework given by Bodlaender et al. [5],

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]. It follows from our results (Sect. 3.3) that we can use the representation with bit strings when working with the basis from [5]. As an open problem, we pose if a smaller representation with bit strings is possible when using the basis of 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 than the isolation lemma to optimize the running time.
- To what extent do results change if we use normal (instead of nice) tree decompositions?
- We notice that the underlying structure of a decomposition can have a large influence on the performance of the algorithms. What is the payoff for further optimizing a decomposition (i.e. minimizing the number of large bags and join bags) after one of small width has been found?
- What is the effect of the ratio between the number of terminals and the number of vertices on the running times and space usages?
- 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 to 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?

References

1. Aneja, Y.P.: An integer linear programming approach to the Steiner problem in graphs. Networks **10**, 167–178 (1980)
2. Arnborg, S., Lagergren, J., Seese, D.: Easy problems for tree-decomposable graphs. J. Algorithms **12**, 308–340 (1991)
3. Beasley, J.E.: An algorithm for the Steiner problem in graphs. Networks **14**, 147–159 (1984)
4. Bodlaender, H.L.: Dynamic programming algorithms on graphs with bounded tree-width. In: Lepistö, T., Salomaa, A. (eds.) Proceedings of the 15th International Colloquium on Automata, Languages and Programming, ICALP'88, Volume 317 of Lecture Notes in Computer Science, pp. 105–119. Springer, Berlin (1988)
5. Bodlaender, H.L., Cygan, M., Kratsch, S., Nederlof, J.: 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, pp. 196–207. Springer, Berlin (2013)
6. Bodlaender, H.L., Koster, A.M.C.A.: Treewidth computations I. Upper bounds. Inf. Comput. **208**, 259–275 (2010)

7. Borie, R.B., Parker, R.G., Tovey, C.A.: Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. Algorithmica **7**, 555–581 (1992)
8. Chimani, M., Mutzel, P., Zey, B.: Improved Steiner tree algorithms for bounded treewidth. J. Discret. Algorithms **16**, 67–78 (2012)
9. Cook, W., Seymour, P.D.: Tour merging via branch-decomposition. INFORMS J. Comput. **15**(3), 233–248 (2003)
10. Cygan, M., Kratsch, S., Nederlof, J.: Fast Hamiltonicity checking via bases of perfect matchings. In: Proceedings of the 45th Annual Symposium on Theory of Computing, STOC 2013, pp. 301–310 (2013)
11. Cygan, M., Nederlof, J., Pilipczuk, M., Pilipczuk, M., van Rooij, J., Wojtaszczyk, J. O.: Solving connectivity problems parameterized by treewidth in single exponential time. In: Proceedings of the 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, pp. 150–159 (2011)
12. Duin, C.: Steiner problems in graphs. Ph.D. thesis, University of Amsterdam, Amsterdam, The Netherlands (1993)
13. Fafianie, S., Bodlaender, H.L., Nederlof, J.: 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. Fomin, F.V., Lokshtanov, D., Saurabh, S.: Efficient computation of representative sets with applications in parameterized and exact algorithms. In: Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, pp. 142–151
15. Hanan, M.: On Steiner's problem with rectilinear distance. SIAM J. Appl. Math. **14**, 255–265 (1966)
16. Hwang, F., Richards, D.S., Winter, P.: The Steiner Tree Problem, Volume 53 of Annals of Discrete Mathematics. Elsevier, Amsterdam (1992)
17. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations, pp. 85–104. Plenum Press, New York (1972)
18. Kloks, T.: Treewidth. Computations and Approximations, Volume 842 of Lecture Notes in Computer Science. Springer, Berlin (1994)
19. Koch, T., Martin, A., Voß, S.: Steinlib, an updated library on Steiner tree problems in graphs. Technical Report ZIB-Report 00–37, Konrad-Zuse Zentrum für Informationstechnik Berlin. http://elib.zib.de/steinlib (2000)
20. Korach, E., Solel, N.: Linear time algorithm for minimum weight Steiner tree in graphs with bounded treewidth. Technical Report 632, Technion, Haifa, Israel (1990)
21. Lovász, L.: Flats in matroids and geometric graphs. In: Combinatorial Surveys. Proceedings 6th Britisch Combinatorial Conference, pp. 45–86. Academic Press, London (1977)
22. Marx, D.: A parameterized view on matroid optimization problems. Theoret. Comput. Sci. **410**, 4471–4479 (2009)
23. Monien, B.: How to find long paths efficiently. Ann. Discret. Math. **25**, 239–254 (1985)
24. Robertson, N., Seymour, P.D.: Graph minors. II. Algorithmic aspects of tree-width. J. Algorithms **7**, 309–322 (1986)
25. Telle, J., Proskurowski, A.: Efficient sets in partial k-trees. Discret. Appl. Math. **44**, 109–117 (1993)
26. Treewidthlib. http://www.cs.uu.nl/people/hansb/treewidthlib (2004)
27. Wald, J.A., Colbourn, C.J.: Steiner trees, partial 2-trees, and minimum IFI networks. Networks **13**, 159–167 (1983)
28. Warme, D., Winter, P., Zachariasen, M.: GeoSteiner, software for computing Steiner trees. http://www.diku.dk/hjemmesider/ansatte/martinz/geosteiner/
29. Wei-Kleiner, F.: Tree decomposition based Steiner tree computation over large graphs. Report on arXiv:1305.5757 (2013)
30. Winter, P.: Steiner problem in networks: a survey. Networks **17**, 129–167 (1987)