On the Maximum Weight Minimal Separator

Tesshu Hanaka[1(✉)], Hans L. Bodlaender[2,3], Tom C. van der Zanden[2], and Hirotaka Ono[1]

[1] Department of Economic Engineering, Kyushu University, 6-19-1, Hakozaki, Higashi-ku, Fukuoka 812-8581, Japan
3EC15004S@s.kyushu-u.ac.jp, hirotaka@econ.kyushu-u.ac.jp

[2] Department of Computer Science, Utrecht University, PO Box 80.089, 3508 TB Utrecht, The Netherlands
{h.l.bodlaender,t.c.vanderzanden}@uu.nl

[3] Department of Mathematics and Computer Science, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. Given an undirected and connected graph $G = (V, E)$ and two vertices $s, t \in V$, a vertex subset S that separates s and t is called an s-t separator, and an s-t separator is called *minimal* if no proper subset of S separates s and t. In this paper, we consider finding a minimal s-t separator with maximum weight on a vertex-weighted graph. We first prove that this problem is NP-hard. Then, we propose an $\mathbf{tw}^{O(\mathbf{tw})}n$-time deterministic algorithm based on tree decompositions. Moreover, we also propose an $O^*(9^{\mathbf{tw}} \cdot W^2)$-time randomized algorithm to determine whether there exists a minimal s-t separator where W is its weight and $\mathbf{tw}$ is the treewidth of G.

Keywords: Parameterized algorithm · Minimal separator · Treewidth

1 Introduction

Given a connected graph $G = (V, E)$ and two vertices $s, t \in V$, a set $S \subseteq V$ of vertices is called an *s-t separator* if s and t belong to different connected components in $G \setminus S$, where $G \setminus S = (V \setminus S, E)$. If a set S is an s-t separator for some s and t, it is simply called a *separator*. If an s-t separator S is minimal in terms of set inclusion, that is, no proper subset of S separates s and t, it is called a *minimal s-t separator*. Similarly, if a separator is minimal in terms of set inclusion, it is called a *minimal separator*.

Separators and minimal separators have been considered important in several contexts and have been intensively studied indeed. For example, they are obviously related to the connectivity of graphs, which is an important notion

This study is partially supported by NWO Gravity grant "Networks" (024.002.003), JSPS KAKENHI Grant Numbers JP26540005, 26241031, and Asahi Glass Foundation.

in many practical applications, such as network design, supply chain analysis and so on. From a theoretical point of view, minimal separators are related to treewidth or potential maximal cliques, which play key roles in designing fast algorithms [4,6].

In this paper, we consider the problem of finding a maximum weight minimal separator of a given weighted graph. More precisely, the problem is defined as follows: Given a connected graph $G = (V, E)$, vertices $s, t \in V$ and a weight function $w : V \to \mathbb{N}^+$, find a minimal s-t separator whose weight $\sum_{v \in S} w(v)$ is maximum. The decision version of the problem is to decide the existence of minimal s-t separator with weight W. We name the problems Maximum Weight Minimal s-t Separator.

This problem is motivated in the context of supply chain network analysis. When a weighted network represents a supply chain where a vertex represents an industry, s and t are virtual vertices respectively representing source and sink, and the weight of a vertex represents its financial importance, the maximum weight minimal s-t separator is interpreted as the most important set of industries that is influential or vulnerable in the supply chain network.

Unfortunately, the problem is shown to be NP-hard, and we then design an FPT algorithm with respect to treewidth. It should be noted that since the condition of s-t connectivity can be written in Monadic Second Order Logic, it can be solved in $f(\mathbf{tw}) \cdot n$ time by Courcelle's meta-theorem, where f is a computable function and $\mathbf{tw}$ is treewidth of the graph. However, the function f forms a tower of exponentials; the existence of an FPT algorithm with better running time is not obvious.

In this paper, we propose two parameterized algorithms with respect to treewidth. One is a $\mathbf{2}^{O(\mathbf{tw} \log \mathbf{tw})} n$-time deterministic algorithm and the other is an $O^*(c^{\mathbf{tw}} \cdot W^2)$-time randomized algorithm for the decision version, where c is a constant and O^* is the order notation omitting the polynomial factor. The former algorithm is based on a standard dynamic programming approach, whereas the latter utilizes two techniques recently developed. The first technique is called *Cut & Count*, and by using this, the running time is bounded by a single exponential of treewidth. Furthermore, by applying the second technique called *fast convolution*, we improve the running time by reducing the base of the exponent from $c = 21$ to $c = 9$; the total running time of the resulting algorithm is $O^*(9^{\mathbf{tw}} \cdot W^2)$, which can be further improved when the graph is unweighted.

1.1 Related Work

The Number of Minimal Separators. Minimal separators have been investigated for a long time in many aspects. As mentioned above, they are related to treewidth or potential maximal cliques, for example [4,6]. In general, a graph has exponentially many minimal separators, and in fact there exists a graph with $\Omega(3^{n/3})$ minimal separators [9]. Recently, this bound was improved to $\Omega(1.4521^n)$ [10]. On the other hand, some graph classes have only polynomially (even linearly) many minimal separators. For example, Bouchitté showed that weakly triangulated (weakly chordal) graphs have a polynomial number of separators [5]. As examples of other graph classes with polynomial minimal separators, there are circular arc graphs [12], and polygon circle graphs, which is a superclass of circle graphs [16,17].

On the other hand, Berry et al. presented an $O(n^3 \cdot R_{sep})$-time algorithm that enumerates all the minimal separators where R_{sep} is the number of these [2]. By combining these results, we know that Maximum Weight Minimal s-t Separator can be solved in polynomial time for the graph classes mentioned above. That is, we just enumerate all the minimal separators and evaluate the weights of these for such graphs.

Proposition 1. Maximum Weight Minimal s-t Separator *can be solved in polynomial time for a graph that has a polynomial number of minimal separators.*

The Relationship Between Minimal Separators and Treewidth. Minimal separators and treewidth are strongly related. As for the number of minimal separators, if a graph has a polynomial minimal separators, we can compute its treewidth in polynomial time [5,6]. Such graph classes include circular-arc ($O(n^2)$ [11,12]), polygon circle ($O(n^2)$ [16]), weakly triangulated ($O(n^2)$ [5]) and so on. On the other hand, computing treewidth is fixed parameter tractable with respect to the maximum size of minimal separators [15]. This parameter corresponds to the solution size of Maximum Weight Minimal s-t Separator on unweighted graphs. In this sense, this paper focuses on the converse relation of two parameters: maximum size of minimal separators and treewidth. For treewidth as the parameter, we consider the fixed parameter tractability of Maximum Weight Minimal s-t Separator.

The remainder of this paper is organized as follows. In Sect. 2, we first give basic terminology, basic notions of algorithm design and NP-hardness for the problem. In Sect. 3, we design a standard dynamic programming algorithm based on tree decompositions. In Sect. 4, we propose randomized algorithms based on the *Cut & Count* technique.

2 Preliminaries

In this section, we give notations, definitions, and some basic concepts. Let $G = (V, E)$ be an undirected and vertex-weighted graph. We assume that G does not have an edge (s, t), that is, $(s, t) \notin E$ because if not then there is no s-t separator. For $V' \subseteq V$, let $G[V']$ denote the subgraph of G induced by V'. Furthermore, we denote the set of neighbors of a vertex v by $N(v)$. We define the function $[p]$ as follows: if p is true, then $[p] = 1$, otherwise $[p] = 0$.

2.1 Tree Decomposition

Our algorithms proposed in Sects. 3 and 4 are based on dynamic programming on tree decompositions. In this subsection, we give the definition of tree decomposition.

Definition 1. *A* tree decomposition *of a graph $G = (V, E)$ is defined as a pair $\langle \mathcal{X}, T \rangle$, where $\mathcal{X} = \{X_1, X_2, \ldots, X_N \subseteq V\}$, and T is a tree whose nodes are labeled by $I \in \{1, 2, \ldots, N\}$, such that*

1. $\bigcup_{i \in I} X_i = V$.
2. *For all $\{u, v\} \in E$, there exists an X_i such that $\{u, v\} \subseteq X_i$.*
3. *For all $i, j, k \in I$, if j lies on the path from i to k in T, then $X_i \cap X_k \subseteq X_j$.*

In the following, we call T a decomposition tree, and we use term "nodes" (not "vertices") for the elements of T to avoid confusion. Moreover, we call a subset of V corresponding to a node $i \in I$ a *bag* and denote it by X_i. The *width* of a tree decomposition $\langle \mathcal{X}, T \rangle$ is defined by $\max_{i \in I} |X_i| - 1$, and the *treewidth* of G, denoted by $\mathbf{tw}(G)$, is the minimum width over all tree decompositions of G. We sometimes use the notation $\mathbf{tw}$ instead of $\mathbf{tw}(G)$ for simplicity.

In general, computing $\mathbf{tw}(G)$ of a given graph G is NP-hard [1], but fixed-parameter tractable with respect to itself and there exists a linear time algorithm if treewidth is fixed [3]. In the following, we assume that a decomposition tree with the minimum treewidth is given.

Kloks introduced a very useful type of tree decomposition for some algorithms, called *nice tree decomposition* [11]. More precisely, it is a special binary tree decomposition which has four types of nodes, named *leaf*, *introduce vertex*, *forget* and *join*. A variant of the notion, using a new type of node named *introduce edge*, was introduced by Cygan et al. [7].

Definition 2. *A tree decomposition $\langle \mathcal{X}, T \rangle$ is called* nice tree decomposition *if it satisfies the following:*

1. *T is rooted at a designated node $X_r \in \mathcal{X}$ satisfying $|X_r| = 0$, called the* root node.
2. *Every node of the tree T has at most two children.*
3. *Each node in T has one of the following five types:*
   – *A* leaf *node i which has no children and its bag X_i satisfies $|X_i| = 0$.*
   – *An* introduce vertex *node i has one child j with $X_i = X_j \cup \{v\}$ for a vertex $v \in V$.*
   – *An* introduce edge *node i has one child j and labeled with an edge $(u, v) \in E$ where $u, v \in X_i = X_j$.*
   – *A* forget *node i has one child j and satisfies $X_i = X_j \setminus \{v\}$ for a vertex $v \in V$.*
   – *A* join *node i has two children nodes j_1, j_2 and satisfies $X_i = X_{j_1} = X_{j_2}$.*

We can transform any tree decomposition to a nice tree decomposition with $O(n)$ bags in linear time [8]. Given a tree decomposition $\langle \mathcal{X}, T \rangle$, we define a subgraph $G_i = (V_i, E_i)$ for each node i where V_i is the union of all bags X_j with $j = i$ or j a descendant of i in T, and $E_i \subseteq E$ is the set of edges with both endpoints in V_i.

2.2 NP-hardness

In this subsection, we mention NP-hardness for Maximum Weight Minimal s-t Separator. The proof is omitted from this extended abstract.

Theorem 1. Maximum Weight Minimal s-t Separator *is NP-hard even if all the vertex weights are identical.*

3 Dynamic Programming on Tree Decompositions

In this section, we give an FPT algorithm with respect to treewidth. It is a standard dynamic programming algorithm based on tree decompositions, and the running time is $\mathbf{tw}^{O(\mathbf{tw})}$. The running time $\mathbf{tw}^{O(\mathbf{tw})}$ appears in some connectivity problems, for example Steiner Tree, Feedback Vertex Set and Connected Vertex Cover [7,8].

We first discuss how Maximum Weight Minimal s-t Separator can be viewed as a connectivity problem. To show this, we define connected partitions.

Definition 3. *A* connected partition *of weight W is a partition (S, A, B, Q) of V such that: (1) $s \in A, t \in B$, (2) $G[A]$ is connected, (3) $G[B]$ is connected, (4) $\sum_{v\in S} w(v) = W$, (5) for $\forall v \in S$, there exist vertices $a \in A, b \in B$ such that $(a, v) \in E, (v, b) \in E$ and (6) for sets A, B, Q, there does not exist an edge (u, v) such that u and v are in different sets.*

Note that a connected partition represents a structure of separators. In fact, it corresponds to a minimal separator and we can show the following theorem, which plays a key role of designing dynamic programming algorithms. The proof is omitted from this extended abstract.

Theorem 2. *There exists a minimal s-t separator of weight W if and only if there exists a connected partition (S, A, B, Q) of weight W.*

Using connected partitions, we design an $\mathbf{tw}^{O(\mathbf{tw})}$-time algorithm for Maximum Weight Minimal s-t Separator. First, we partition S into $S_\emptyset, S_A, S_B$ and S_{AB}. (See Fig. 1). They are needed for the updating process in the dynamic programming. Set $S_\emptyset$ consists of the vertices in S that have no neighbor in A and B, but may have neighbors in S_A, S_B, S_{AB}, Q. Set S_A (resp., S_B) consists of the vertices in S that has at least one neighbor in A (resp., B), but no neighbor of B (resp., A). They may have neighbors in S_A, S_B, S_{AB}, Q. Set S_{AB} consists of the vertices in S that have neighbors in A and in B and may have neighbors in S_A, S_B, S_{AB}, Q. With these sets, we define a *partial solution* as follows.

Definition 4. *Given a node i of the tree decomposition of G, a* partial solution *for node i is a partition $(S_\emptyset, S_A, S_B, S_{AB}, A, B, Q)$, such that:*

- *$S_\emptyset \cup S_A \cup S_A \cup S_{AB} \cup A \cup B \cup Q = V_i$,*
- *$\forall v \in S_\emptyset$, $N(v) \cap (A \cup B) = \emptyset$,*

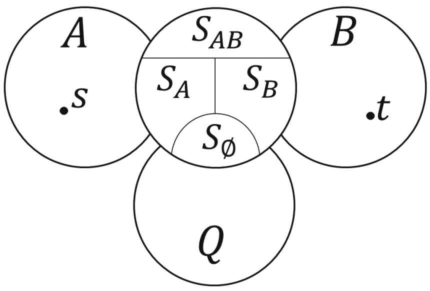

Fig. 1. Connection between vertex sets

- $\forall v \in S_A$, $N(v) \cap B = \emptyset$ *and* $N(v) \cap A \neq \emptyset$,
- $\forall v \in S_B$, $N(v) \cap B \neq \emptyset$ *and* $N(v) \cap A = \emptyset$,
- $\forall v \in S_{AB}$, $N(v) \cap B \neq \emptyset$ *and* $N(v) \cap A \neq \emptyset$,
- $s \in V_i \Rightarrow s \in A$ *and*
- $t \in V_i \Rightarrow t \in B$.

We prepare DP tables for each node. For a representation of the state of v, we define the *coloring* function $c : V \to \{s_\emptyset, s_A, s_B, s_{AB}, a, b, q\}$. Each element of $\{s_\emptyset, s_A, s_B, s_{AB}, a, b, q\}$ is called a *state*. For a coloring c, we denote the state of the coloring of v by $c(v)$. The states of a coloring c represent which set a vertex is in, for example, v is in $S_\emptyset$ if $c(v) = s_\emptyset$.

To consider the connectivity of sets A and B, we use all partitions of these in a bag. That is, we define two partitions $\mathcal{P}^A = \{P_1^A, P_2^A, \ldots, P_\alpha^A\}$ of $X_i \cap A$ and $\mathcal{P}^B = \{P_1^B, P_2^B, \ldots, P_\beta^B\}$ of $X_i \cap B$, where α and β are the number of partitioned sets of $X_i \cap A$ and $X_i \cap B$, respectively, that is, α and β are at most $|X_i \cap A|$ and $|X_i \cap B|$. We call each element of a partition P_ℓ a *block*. They correspond to connected components of $G[A]$(resp., $G[B]$). Note that there are $|X_i|^{O(|X_i|)}$ partitions for each node X_i. Intuitively, just one block $\{\{v\}\}$ is added to $\mathcal{P}$ in each introduce vertex v node; then blocks are merged in the updating process of introduce edge nodes and join nodes. For forget nodes, we have to consider the relationship between connectivity and partitions.

Suppose that introduce vertex nodes, introduce edge nodes and forget nodes have one child node j respectively, and join nodes have two child nodes j_1, j_2. We sometimes denote a coloring in parent node i by c_i and in child node j by c_j to emphasize that we deal with two different nodes. Moreover, we denote the coloring of vertex v by $c_i(v)$, $c_j(v)$, respectively.

Now, we transform a nice tree decomposition by adding $\{s, t\}$ to all bags; thus we can suppose that the root bag X_r contains only two vertices s, t. The width of this tree decomposition is at most $\mathbf{tw} + 2$. We can transform any tree decomposition into such a tree decomposition in polynomial time.

We then define function $f_i(c, \mathcal{P}^A, \mathcal{P}^B)$ as the possible maximum weight of vertices in $S \cap V_i$ under the following conditions: (1) c defines a partial solution $(S_\emptyset, S_A, S_B, S_{AB}, A, B, Q)$ and (2) each block of $\mathcal{P}^A$ and $\mathcal{P}^B$ forms a connected component in $X_i \cap A$ and $X_i \cap B$, respectively. If $c, \mathcal{P}^A, \mathcal{P}^B$ do not satisfy the conditions, let f_i be $-\infty$.

We now define recursive formulas for each node. In a root node, $f_r(\{a\} \times \{b\}, \{\{a\}\}, \{\{b\}\})$ is an optimal value because $X_r = \{s, t\}$.

Leaf node: In a leaf node, we define $f_i(\{a\} \times \{b\}, \{\{a\}\}, \{\{b\}\}) := 0$, otherwise $f_i(c, \mathcal{P}^A, \mathcal{P}^B) := -\infty$ since there are only two vertices s, t in X_i.

Introduce vertex v node: In an introduce vertex node, we consider three cases for colorings. If $c(v) = s_\emptyset$, we add $w(v)$ to $f_j(c, \mathcal{P}^A, \mathcal{P}^B)$ because v is added in S. If $c(v) \in \{a, b, q\}$, the value of f_i does not change since $v \notin S$. Moreover, we add a block $\{\{v\}\}$ to $\mathcal{P}^A$ or $\mathcal{P}^B$ depending on if $c(v) = a$ or $c(v) = b$, respectively. Finally, if $c(v) \in \{s_A, s_B, s_{AB}\}$, a partial solution is invalid by the definition because v has no incident edge and hence no neighbor in A and B. Therefore, we define f_i as follows:

$$f_i(c, \mathcal{P}^A, \mathcal{P}^B) := \begin{cases} f_j(c \setminus \{c(v)\}, \mathcal{P}^A, \mathcal{P}^B) + w(v) & \text{if } c(v) = s_\emptyset \\ f_j(c \setminus \{c(v)\}, \mathcal{P}^A \setminus \{\{v\}\}, \mathcal{P}^B) & \text{if } c(v) = a \\ f_j(c \setminus \{c(v)\}, \mathcal{P}^A, \mathcal{P}^B \setminus \{\{v\}\}) & \text{if } c(v) = b \\ f_j(c \setminus \{c(v)\}, \mathcal{P}^A, \mathcal{P}^B) & \text{if } c(v) = q \\ -\infty & \text{otherwise.} \end{cases}$$

Introduce edge (u, v) node: In an introduce edge node, we define f_i for the following cases of $c(u)$, $c(v)$.

- If $c(u) = a$ and $c(v) = a$, the vertices u, v are in A. Because edge (u, v) is added, u and v are in the same block of partition $\mathcal{P}^A$. Hence, if not, we set $f_i(c, \mathcal{P}^A, \mathcal{P}^B) := -\infty$. Then, there are two cases: the partitions in $A \cap X_i$ (parent) and $A \cap X_j$ (child) are same or not. In the former case, u and v is in the same block in the partition of $A \cap X_j$, and we then set $f_i(c, \mathcal{P}^A, \mathcal{P}^B) := f_j(c, \mathcal{P}^A, \mathcal{P}^B)$. In the latter case, let $\mathcal{P}'^A$ be a partition of $A \cap X_j$ such that $\mathcal{P}^A \neq \mathcal{P}'^A$ but $\mathcal{P}'^A$ changes to $\mathcal{P}^A$ by merging two blocks of $\mathcal{P}'^A$ including u and v respectively with edge (u, v). Therefore, we take a $\mathcal{P}'^A$ that maximizes $f_i(c, \mathcal{P}'^A, \mathcal{P}^B)$. Then, we set f_i as follows:

$$f_i(c, \mathcal{P}^A, \mathcal{P}^B) := \max\{f_j(c, \mathcal{P}^A, \mathcal{P}^B), \max_{\mathcal{P}'^A} f_j(c, \mathcal{P}'^A, \mathcal{P}^B)\}.$$

- The case that $c(u) = b$ and $c(v) = b$ is almost the same as the case that $c(u) = a$ and $c(v) = a$. If u and v are not in the same block of partition $\mathcal{P}^B$, we then set $f_i(c, \mathcal{P}^A, \mathcal{P}^B) := -\infty$. Let $\mathcal{P}'^B$ be a partition of $B \cap X_j$ such that $\mathcal{P}^B \neq \mathcal{P}'^B$ but $\mathcal{P}'^B$ changes to $\mathcal{P}^B$ by merging two blocks of $\mathcal{P}'^B$ including u and v respectively with edge (u, v). Then, we define f_i as follows:

$$f_i(c, \mathcal{P}^A, \mathcal{P}^B) := \max\{f_j(c, \mathcal{P}^A, \mathcal{P}^B), \max_{\mathcal{P}'^B} f_j(c, \mathcal{P}^A, \mathcal{P}'^B)\}.$$

– If $c(u), c(v) \in \{s_\emptyset, s_A, s_B, s_{AB}, q\}$, we define f_i as follows:
$$f_i(c, \mathcal{P}^A, \mathcal{P}^B) = f_j(c, \mathcal{P}^A, \mathcal{P}^B).$$
In this case, (u, v) is indifferent to the partitions and the value is not changed because only one edge (u, v) is added.
– If $(c(u), c(v)) = (s_A, a), (a, s_A)$, we consider two cases. One case is that $u \in S_A$ and $v \in A$ in a child node and the other case is that $u \in S_\emptyset$ and $v \in A$ in a child node. In the other case, u is moved from $S_\emptyset$ into S_A by adding (u, v), because u has a neighbor v in A. Thus, we define f_i as follows:
$$\begin{aligned} f_i(c \times \{s_A\} \times \{a\}, \mathcal{P}^A, \mathcal{P}^B) := \max \{\ & f_j(c \times \{s_A\} \times \{a\}, \mathcal{P}^A, \mathcal{P}^B), \\ & f_j(c \times \{s_\emptyset\} \times \{a\}, \mathcal{P}^A, \mathcal{P}^B)\}. \end{aligned}$$
– If $(c(u), c(v)) = (s_B, b), (b, s_B)$, we consider almost the same cases as above; that is, $u \in S_B$, $v \in B$ and $u \in S_\emptyset$ and $v \in B$ in a child node.
$$\begin{aligned} f_i(c \times \{s_B\} \times \{b\}, \mathcal{P}^A, \mathcal{P}^B) := \max \{\ & f_j(c \times \{s_B\} \times \{b\}, \mathcal{P}^A, \mathcal{P}^B), \\ & f_j(c \times \{s_\emptyset\} \times \{b\}, \mathcal{P}^A, \mathcal{P}^B)\}. \end{aligned}$$
– If $(c(u), c(v)) = (s_{AB}, a), (a, s_{AB})$, there are two cases: (1) $u \in S_{AB}$ and $v \in A$ in a child node and (2) $u \in S_B$ and $v \in A$ in a child node. In the latter case, u is moved from S_B to S_{AB} by adding (u, v), because u has a neighbor v in B. Therefore, we define f_i as follows:
$$\begin{aligned} f_i(c \times \{s_{AB}\} \times \{a\}, \mathcal{P}^A, \mathcal{P}^B) := \max \{\ & f_j(c \times \{s_{AB}\} \times \{a\}, \mathcal{P}^A, \mathcal{P}^B), \\ & f_j(c \times \{s_B\} \times \{a\}, \mathcal{P}^A, \mathcal{P}^B)\}. \end{aligned}$$
– If $(c(u), c(v)) = (s_{AB}, b), (b, s_{AB})$, we consider almost the same cases as above; that is, $u \in S_{AB}$, $v \in B$ and $u \in S_A$ and $v \in B$ in a child node.
$$\begin{aligned} f_i(c \times \{s_{AB}\} \times \{b\}, \mathcal{P}^A, \mathcal{P}^B) := \max \{\ & f_j(c \times \{s_{AB}\} \times \{b\}, \mathcal{P}^A, \mathcal{P}^B), \\ & f_j(c \times \{s_A\} \times \{b\}, \mathcal{P}^A, \mathcal{P}^B)\}. \end{aligned}$$
– Otherwise, we define $f_i(c, \mathcal{P}^A, \mathcal{P}^B)) := -\infty$ because the rest of cases is invalid. Recall the definition of connected partition and the meaning of states.

Forget v node: In a forget v node, if $c_j(v) \in \{s_\emptyset, s_A, s_B\}$, vertex v never has neighbors both in A and in B and hence such case is invalid because of the definition of connected partition. If $c_j(v) \in \{s_{AB}, q\}$, we need not consider the connectivity of these. In the case that $c_j(v) = a$, we only consider partitions such that there exists at least one vertex u in A included the same block as v. If not, the block including v is never merged. Consequently, $G[A]$ would not be connected in the root node. The case that $c_j(v) = b$ is almost the same. Let $\mathcal{P}'^A, \mathcal{P}'^B$ be a partition satisfying such conditions, then we define f_i as follows:
$$\begin{aligned} f_i(c, \mathcal{P}^A, \mathcal{P}^B)) := \max \{\ & f_j(c \times \{s_{AB}\}, \mathcal{P}^A, \mathcal{P}^B)), f_j(c \times \{q\}, \mathcal{P}^A, \mathcal{P}^B), \\ & \max_{\mathcal{P}'^A} f_j(c \times \{a\}, \mathcal{P}'^A, \mathcal{P}^B)), \max_{\mathcal{P}'^B} f_j(c \times \{b\}, \mathcal{P}^A, \mathcal{P}'^B))\}. \end{aligned}$$

Table 1. This table represents combinations of states of two children nodes j_1, j_2 for each vertex in $X_i = X_{j_1} = X_{j_2}$. The row and column correspond to states of j_1, j_2 respectively and inner elements correspond to states of x. For example, if $c_i(v) = s_A$, there are three combinations such that $(c_{j_1}(v), c_{j_2}(v)) = (s_A, s_\emptyset)$, $(c_{j_1}(v), c_{j_2}(v)) = (s_\emptyset, s_A)$ and $(c_{j_1}(v), c_{j_2}(v)) = (s_A, s_A)$.

	$s_\emptyset$	s_A	s_B	s_{AB}	a	b	q
$s_\emptyset$	$s_\emptyset$	s_A	s_B	s_{AB}			
s_A	s_A	s_A	s_{AB}	s_{AB}			
s_B	s_B	s_{AB}	s_B	s_{AB}			
s_{AB}	s_{AB}	s_{AB}	s_{AB}	s_{AB}			
a					a		
b						b	
q							q

Join node: For a parent node i and two children nodes j_1, j_2, we denote each coloring by c_i, c_{j_1}, c_{j_2} and each partition by $\mathcal{P}^A_{j_1}, \mathcal{P}^B_{j_1}, \mathcal{P}^A_{j_2}, \mathcal{P}^B_{j_2}$. We then define the subset D of tuples of $((c_{j_1}, \mathcal{P}^A_{j_1}, \mathcal{P}^B_{j_1}), (c_{j_2}, \mathcal{P}^A_{j_2}, \mathcal{P}^B_{j_2}))$ such that the combinations of colorings for c_{j_1}, c_{j_2} satisfy the following conditions. (See Table 1):

- $\forall v \in c_i^{-1}(\{s_\emptyset, a, b, q\}), (c_{j_1}(v), c_{j_2}(v)) = (c_i(v), c_i(v))$,
- $\forall v \in c_i^{-1}(\{s_A\}, (c_{j_1}(v), c_{j_2}(v)) = (s_A, s_\emptyset), (s_\emptyset, s_A), (s_A, s_A)$,
- $\forall v \in c_i^{-1}(\{s_B\}), c_{j_1}(v), c_{j_2}(v)) = (s_B, s_\emptyset), (s_\emptyset, s_B), (s_B, s_B)$, and
- $\forall v \in c_i^{-1}(\{s_{AB}\}), (c_{j_1}(v), c_{j_2}(v)) = (s_{AB}, s_\emptyset), (s_{AB}, s_A), (s_{AB}, s_B)$, (s_{AB}, s_{AB}), $(s_\emptyset, s_{AB})$, (s_A, s_{AB}), (s_B, s_{AB}), (s_A, s_B), (s_B, s_A),

and the partition caused by merging $\mathcal{P}^A_{j_1}$ and $\mathcal{P}^A_{j_2}$ equals to $\mathcal{P}^A$ and the partition caused by merging $\mathcal{P}^B_{j_1}, \mathcal{P}^B_{j_2}$ equals to $\mathcal{P}^B$. If $D = \emptyset$ for $c_i, \mathcal{P}^A, \mathcal{P}^B$, we set $f_i(c_i, \mathcal{P}^A, \mathcal{P}^B) := -\infty$. Otherwise, we set $S^* := c_i^{-1}(\{s_\emptyset, s_A, s_B, s_{AB}\})$. Then we define f_i as follows:

$$f_i(c_i, \mathcal{P}^A, \mathcal{P}^B) := \max_{((c_{j_1}, \mathcal{P}^A_{j_1}, \mathcal{P}^B_{j_1}), (c_{j_2}, \mathcal{P}^A_{j_2}, \mathcal{P}^B_{j_2})) \in D} \{ f_{j_1}(c_{j_1}, \mathcal{P}^A_{j_1}, \mathcal{P}^B_{j_1}) + f_{j_2}(c_{j_2}, \mathcal{P}^A_{j_2}, \mathcal{P}^B_{j_2}) - w(S^*)\}.$$

The subtraction in the right hand side of the equation above is because the weight $w(S^*)$ is counted twice; once in each child node.

We recursively calculate f_i on the decomposition tree. Note that all bags have $|X_i|$ vertices and the number of combinations of colorings and partitions $(c, \mathcal{P}^A, \mathcal{P}^B)$ in each node is $|X_i|^{O(|X_i|)} = \mathbf{tw}^{O(\mathbf{tw})}$. The running time to compute all f_i's in X_i is dominated by join nodes and it is roughly $(\mathbf{tw}^{O(\mathbf{tw})})^3 = \mathbf{tw}^{O(\mathbf{tw})}$ since we scan every coloring and partition in two children nodes X_{j_1} and X_{j_2} for each coloring c_i and each partition $\mathcal{P}^A, \mathcal{P}^B$ and then check all combinations. Therefore, the total running time is $\mathbf{tw}^{O(\mathbf{tw})}n$ and we conclude with the following theorem.

Theorem 3. *For graphs of treewidth at most* $\mathbf{tw}$*, there exists an algorithm that solves* MAXIMUM WEIGHT MINIMAL *s-t* SEPARATOR *in time* $\mathbf{tw}^{O(\mathbf{tw})}n$*.*

4 Algorithms Using *Cut & Count*

In this section, we give an algorithm that solves the decision version of MAXIMUM WEIGHT MINIMAL *s-t* SEPARATOR to decide the existence of minimal *s-t* separator with weight W in time $O^*(9^{\mathbf{tw}} \cdot W^2)$ for graphs of treewidth at most $\mathbf{tw}$. This algorithm is based on the *Cut & Count* technique.

4.1 Isolation Lemma

In this subsection, we explain the Isolation Lemma introduced by Mulmuley et al. [13]. The main idea of the *Cut & Count* technique is to obtain a single solution with high probability; we count modulo 2, and the Isolation Lemma guarantees the existence of such a single solution.

Definition 5 ([13]). *A function* $w' : U \to \mathbb{Z}$ isolates *a set family* $\mathcal{F} \subseteq 2^U$ *if there is a unique* $S' \in \mathcal{F}$ *with* $w'(S') = \min_{S \in \mathcal{F}} w'(S)$ *where* $w'(X) = \sum_{u \in X} w'(u)$*.*

Lemma 1 (Isolation Lemma [13]). *Let* $F \subseteq 2^U$ *be a set family over a universe* U *with* $|F| > 0$*. For each* $u \in U$*, choose a weight* $w'(u) \in \{1, 2, \dots N\}$ *uniformly and independently at random. Then* $Pr[w' \text{ isolate } \mathcal{F}] \geq 1 - |U|/N$*.*

4.2 Cut & Count

The *Cut & Count* technique was introduced by Cygan et al. for solving connectivity problems [7]. The concept of *Cut & Count* is counting the number of relaxed solutions such that we do not consider whether they are connected or disconnected. Then we compute the number of relaxed solutions modulo 2 and we determine whether there exists a connected solution by cancellation tricks. Now, we define a *consistent cut* to explain the detail of *Cut & Count*.

Definition 6 ([7]). *A cut* (V_1, V_2) *of* $V' \subseteq V$ *such that* $V_1 \cup V_2 = V'$ *and* $V_1 \cap V_2 = \emptyset$ *is* consistent *if* $v_1 \in V_1$ *and* $v_2 \in V_2$ *implies* $(v_1, v_2) \notin E$*.*

This means that a consistent cut (V_1, V_2) of V' has no edge between V_1 and V_2. We fix an arbitrary vertex v in V_1. Then, if $G[V]$ is connected, then there only exists one consistent cut, that is, $(V_1, V_2) = (V, \emptyset)$. Therefore, the number of consistent cuts is odd. By this fact, we only compute the number of consistent cuts modulo 2 on decomposition tree and return yes if the number of consistent cuts is odd, otherwise no in a root node. The Isolation Lemma is useful for us as it implies that when the number of solutions is odd, there is a unique solution with high probability; and hence we can use the modulo 2 trick.

Let $\mathcal{S} \subseteq 2^U$ be a set of solutions. According to [7,8], the *Cut & Count* technique is divided into two parts as follows.

- **The Cut part:** Relax the connectivity requirement by considering the set $\mathcal{R} \supseteq \mathcal{S}$ of possibly connected or disconnected candidate solutions. Moreover, consider the set $\mathcal{C}$ of pairs $(X; C)$ where $X \in \mathcal{R}$ and C is a consistent cut of X.
- **The Count part:** Isolate a single solution by sampling weights of all elements in U with high probability by the Isolation Lemma. Then, compute $|\mathcal{C}|$ modulo 2 using a sub-procedure. Disconnected candidate solutions $X \in \mathcal{R} \setminus \mathcal{S}$ cancel since they are consistent with an even number of cuts. If the only connected candidate $x \in \mathcal{S}$ exists, we obtain the odd number of cuts.

Given a set U and a tree decomposition $\langle \mathcal{X}, T \rangle$, the general scheme of *Cut & Count* is as follows:

Step 1. Set the integer weight for every vertex uniformly and independently at random by $w' : U \to \{1, \ldots, 2|U|\}$.
Step 2. For each integer weight $0 \leq W' \leq 2|U|^2$, compute the number of relaxed solutions of weight W' with consistent cuts modulo 2 on a decomposition tree. Then return yes if it is odd, otherwise no in the root node.

We use the *Cut & Count* technique to determine whether there exists a connected partition (S, A, B, Q) of weight W so that A and B are connected. To apply the above scheme, we newly give the following definition of a *partial solution*. Note that we have to consider two consistent cuts of A and B.

Definition 7. *Given a node i of the tree decomposition of G, a* partial solution *for that node is a tuple $(S_\emptyset, S_A, S_B, S_{AB}, A_l, A_r, B_l, B_r, Q, w)$, such that:*

- $V_i = S_\emptyset \cup S_A \cup S_A \cup S_{AB} \cup A_l \cup A_r \cup B_l \cup B_r \cup Q$,
- (A_l, A_r) *is a consistent cut: there exists no edge* $(u, v) \in E$ *such that* $u \in A_l$ *and* $v \in A_r$,
- (B_l, B_r) *is a consistent cut: there exists no edge* $(u, v) \in E$ *such that* $u \in B_l$ *and* $v \in B_r$,
- $w = \Sigma_{v \in S} w(v)$,
- $\forall v \in S_\emptyset$, $N(v) \cap (A_l \cup A_r \cup B_l \cup B_r) = \emptyset$,
- $\forall v \in S_A$, $N(v) \cap (B_l \cup B_r) = \emptyset$ *and* $N(v) \cap (A_l \cup A_r) \neq \emptyset$,
- $\forall v \in S_B$, $N(v) \cap (B_l \cup B_r) \neq \emptyset$ *and* $N(v) \cap (A_l \cup A_r) = \emptyset$ *and*
- $\forall v \in S_{AB}$, $N(v) \cap (B_l \cup B_r) \neq \emptyset$ *and* $N(v) \cap (A_l \cup A_r) \neq \emptyset$.
- $s \in V_i \Rightarrow s \in A_l$
- $t \in V_i \Rightarrow t \in B_l$

For each vertex v, we set another weight $w'(v)$ by choosing from $\{1, \ldots, 2|V|\}$ and independently at random. We also set the *coloring* $c : V \to \{s_\emptyset, s_A, s_B, s_{AB}, a_l, a_r, b_l, b_r, q\}$. Now, we give a dynamic programming algorithm that computes the number of partial solutions. To count the number of relaxed solutions with consistent cuts, for each c, w and w' we define the counting function $h_i : \{s_\emptyset, s_A, s_B, s_{AB}, a_l, a_r, b_l, b_r, q\}^{|X_i|} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ in each node i on a nice tree decomposition as follows.

Leaf node: In a leaf node, we define $h_i(\emptyset, 0, 0) = 1$, if $S_\emptyset = S_A = S_B = S_{AB} = A_l = A_r = B_l = B_r = \emptyset$ and $w, w' = 0$. Otherwise, $h_i(c, w, w') = 0$.

Introduce vertex v node: The function h_i has five cases in an introduce vertex node. Note that we only add one vertex v without edges. Thus, if $c(v) \in \{s_A, s_B, s_{AB}\}$, a partial solution is invalid by the definition because v has no neighbor. If $c(v) = s_\emptyset$, vertex v is chosen as a vertex of S, and we hence add each weight $w(v)$, $w'(v)$ to w, w', respectively. Moreover, v must not be s, t because s (resp., t) should be in A_l (resp., B_l). If not, it is not a connected partition. Similarly, if $c(v) = a_l$ (resp., b_l), we check whether v is not t (resp., s). As for $c(v) \in \{a_r, b_r, q\}$, we also check whether v is neither s nor t. Therefore, we define h_i in introduce vertex nodes as follows:

$$h_i(c \times \{c(v)\}, w, w') := \begin{cases} [v \neq s, t] h_j(c, w - w(v), w' - w'(v)) & \text{if } c(v) = s_\emptyset \\ [v \neq t] h_j(c, w, w') & \text{if } c(v) = a_l \\ [v \neq s] h_j(c, w, w') & \text{if } c(v) = b_l \\ [v \neq s, t] h_j(c, w, w') & \text{if } c(v) \in \{a_r, b_r, q\} \\ 0 & \text{otherwise.} \end{cases}$$

Introduce edge (u, v) node: In an introduce edge node, we check each state of endpoints of the edge (u, v) and define f_i for some cases.

- If $c(u) = s_\emptyset$, vertex u has no vertices in A, B. Hence, we define the function h_i in this case as follows:

$$h_i(c \times \{c(u)\} \times \{c(v)\}, w, w') := [c(v) \notin \{a_l, a_r, b_l, b_r\}] \\ \cdot h_j(c \times \{s_\emptyset\} \times \{c(v)\}, w, w').$$

- If $c(u) = s_A$, vertex u has neighbors of A but no neighbor of B. In this case, we have two cases. The other case is that $u \in S_\emptyset$ and $v \in A$ in a child node, because adding edge (u, v) in the introduce edge (u, v) node, vertex u is moved from $S_\emptyset$ to S_A. The other case is that $u \in S_A$ and $v \notin B$ in a child node. If $v \in B$, vertex u is in S_{AB} in the parent node. We define h_i as follows. Note that only if $c(v) \in \{a_l, a_r\}$, we sum up two cases. If $c(v) \in \{b_l, b_r\}$, $h_i(c \times \{c(u)\} \times \{c(v)\}, w, w') := 0$, otherwise $h_i(c \times \{c(u)\} \times \{c(v)\}, w, w') := h_j(c \times \{s_A\} \times \{c(v)\}, w, w')$.

$$h_i(c \times \{c(u)\} \times \{c(v)\}, w, w') := [c(v) \in \{a_l, a_r\}] h_j(c \times \{s_\emptyset\} \times \{c(v)\}, w, w') \\ + [c(v) \notin \{b_l, b_r\}] h_j(c \times \{s_A\} \times \{c(v)\}, w, w').$$

- If $c(u) = s_B$ is almost the same as above case, that is, we replace A(resp., B) to B(resp., A).

$$h_i(c \times \{c(u)\} \times \{c(v)\}, w, w') := [c(v) \in \{b_l, b_r\}] h_j(c \times \{s_\emptyset\} \times \{c(v)\}, w, w') \\ + [c(v) \notin \{a_l, a_r\}] h_j(c \times \{s_B\} \times \{c(v)\}, w, w').$$

- If $c(u) = s_{AB}$, we consider three cases: $u \in S_A$ and $v \in B$, $u \in S_B$ and $v \in A$, and $u \in S_{AB}$ and v is in arbitrary set in the children node. For first and second cases, vertex u is moved from S_A(resp., S_B) into S_{AB} by adding edge (u, v). If $u \in S_{AB}$, v is allowed to be in any set because a vertex in S_{AB} could connect to all sets. Therefore, we define f_i as follows:

$$\begin{aligned} h_i(c \times \{c(u)\} \times \{c(v)\}, w, w') := &[c(v) \in \{b_l, b_r\}] h_j(c \times \{s_A\} \times \{c(v)\}, w, w') \\ &+ [c(v) \in \{a_l, a_r\}] h_j(c \times \{s_B\} \times \{c(v)\}, w, w') \\ &+ h_j(c \times \{s_{AB}\} \times \{c(v)\}, w, w'). \end{aligned}$$

- If $c(u) \in \{a_l, a_r\}$, then $c(v) \notin \{b_l, b_r, q\}$ since there is no edge between A, B and Q by the definition of connected partition. There is also no edge between A_l and A_r because (A_l, A_r) is a consistent cut. Therefore, if u is in A_l or A_r, then v are in the same set of u or separator sets S_A, S_{AB}. Note that because u is in A, v is not in $S_\emptyset$, S_B.

$$\begin{aligned} h_i(c \times \{c(u)\} \times \{c(v)\}, w, w') := &[c(v) = c(u)] h_j(c \times \{c(u)\} \times \{c(v)\}, w, w') \\ &+ [c(v) \in \{s_A, s_{AB}\}] h_j(c \times \{c(u)\} \times \{c(v)\}, w, w'). \end{aligned}$$

- The case that $c(u) \in \{b_l, b_r\}$ is almost the same as above case, that is, we replace A by B.

$$\begin{aligned} h_i(c \times \{c(u)\} \times \{c(v)\}, w, w') := &[c(v) = c(u)] h_j(c \times \{c(u)\} \times \{c(v)\}, w, w') \\ &+ [c(v) \in \{s_B, s_{AB}\}] h_j(c \times \{c(u)\} \times \{c(v)\}, w, w'). \end{aligned}$$

- If $c(u) = q$, vertex u is in Q. Hence, v must be in $S_\emptyset, S_A, S_B, S_{AB}$, or Q because a vertex in Q has no neighbor of A and B by the definition of connected partition.

$$\begin{aligned} h_i(c \times \{c(u)\} \times \{c(v)\}, w, w') := &[c(v) \in \{s_\emptyset, s_A, s_B, s_{AB}, q\}] \\ &\cdot h_j(c \times \{c(u)\} \times \{c(v)\}, w, w'). \end{aligned}$$

Forget v node: For a forget v node, the state of v would never change forward. Thus, if $c_j(v) \in \{s_\emptyset, s_A, s_B\}$, a partial solution does not satisfy the condition of connected partition because any $v \in S$ must have neighbors of both A and B. For this reason, we only sum up for each state $c_j(v) \in \{s_{AB}, a_l, a_r, b_l, b_r, q\}$. The function h_i in forget nodes is defined as follows:

$$h_i(c, w, w') := \sum_{c_j(v) \in \{s_{AB}, a_l, a_r, b_l, b_r, q\}} h_j(c \times \{c_j(v)\}, w, w').$$

Join node: We denote each coloring and weights of partial solutions in i, j_1, j_2 by c_i, c_{j_1}, c_{j_2} and w_i, w_{j_1}, w_{j_2}, w'_i, w'_{j_1}, w'_{j_2}, respectively. Moreover, for a state subset $L \subseteq \{s_\emptyset, s_A, s_B, s_{AB}, a_l, a_r, b_l, b_r, q\}$, we define $c^{-1}(L)$ as the vertex set such that all vertices satisfy $c(v) \in L$. For a coloring c_i, we also define the subset

D of tuples of (c_{j_1}, c_{j_2}) as the combinations of colorings of c_{j_1}, c_{j_2} like Sect. 3 such that:

- $\forall v \in c_i^{-1}(\{s_\emptyset, a_l, a_r, b_l, b_r, q\})$, $(c_{j_1}(v), c_{j_2}(v)) = (c_i(v), c_i(v))$,
- $\forall v \in c_i^{-1}(\{s_A\}$, $(c_{j_1}(v), c_{j_2}(v)) = (s_A, s_\emptyset), (s_\emptyset, s_A), (s_A, s_A)$,
- $\forall v \in c_i^{-1}(\{s_B\})$, $c_{j_1}(v), c_{j_2}(v)) = (s_B, s_\emptyset), (s_\emptyset, s_B), (s_B, s_B)$, and
- $\forall v \in c_i^{-1}(\{s_{AB}\})$, $(c_{j_1}(v), c_{j_2}(v)) = (s_{AB}, s_\emptyset)$, (s_{AB}, s_A), (s_{AB}, s_B), (s_{AB}, s_{AB}), $(s_\emptyset, s_{AB})$, (s_A, s_{AB}), (s_B, s_{AB}), (s_A, s_B), (s_B, s_A).

Let S^* be the vertex subset $c_i^{-1}(\{s_\emptyset, s_A, s_B, s_{AB}\})$. To sum up all combinations of vertex states and weights for counting, we define the function h_i. If $D = \emptyset$, we define $h_i(c_i, w_i, w'_i) := 0$. Otherwise,

$$h_i(c_i, w_i, w'_i) := \sum_{\substack{w_{j_1}+w_{j_2} \\ =w_i+w(S^*)}} \sum_{\substack{w'_{j_1}+w'_{j_2} \\ =w'_i+w'(S^*)}} \sum_{(c^*_{j_1}, c^*_{j_2}) \in D} h_{j_1}(c^*_{j_1}, w_{j_1}, w'_{j_1}) h_{j_2}(c^*_{j_2}, w_{j_2}, w'_{j_2}).$$

From now, we analyze the running time of this algorithm. In each leaf, introduce vertex, introduce edge, and forget node, we can compute f_i for each coloring c and weight w, w' in $O(1)$-time because we only use $O(1)$-operations. Therefore, the total running time in them is $O^*(9^{\mathbf{tw}} \cdot W \cdot W')$. However, in a join node, we sum up all weight combinations and coloring combinations satisfying some conditions. There are 21 coloring's combinations for each vertex and $W \cdot W'$ weight's combinations. Therefore, we compute all f_i's in a join node in time $O^*(21^{\mathbf{tw}} \cdot W^2)$. Note that by the definition, $O(W'^2)$ is a polynomial factor.

Theorem 4. *For graphs of treewidth at most* $\mathbf{tw}$*, there exists a Monte-Carlo algorithm that solves the decision version of* Maximum Weight Minimal *s-t* Separator *in time* $O^*(21^{\mathbf{tw}} \cdot W^2)$*. It cannot give false positives and may give false negatives with probability at most* 1/2.

Using the convolution technique [14], we can obtain a faster Monte-Carlo algorithm. The technique helps to speed up the computation for join nodes. The details are omitted from this extended abstract.

Theorem 5. *For graphs of treewidth at most* $\mathbf{tw}$*, there exists a Monte-Carlo algorithm that solves the decision version of* Maximum Weight Minimal *s-t* Separator *in time* $O^*(9^{\mathbf{tw}} \cdot W^2)$*. It cannot give false positives and may give false negatives with probability at most* 1/2*. If the input graph is unweighted, the running time is* $9^{\mathbf{tw}} \cdot |V|^{O(1)}$.

As usual for this type of algorithms, the probability of a false negative can be made arbitrarily small by repeating the algorithm.

Acknowledgments. We are grateful to Dr. Jesper Nederlof for helpful discussions.

References

1. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Algebraic Discrete, Methods **8**(2), 277–284 (1987)
2. Berry, A., Bodat, J.P., Cogis, O.: Generating all the minimal separators of a graph. Int. J. Found. Comput. Sci. **11**(3), 397–404 (2000)
3. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. **25**(6), 1305–1317 (1996)
4. Bodlaender, H.L., Kloks, T., Kratsch, D.: Treewidth and pathwidth of permutation graphs. SIAM J. Discrete Math. **8**(4), 606–616 (1995)
5. Bouchitté, V., Todinca, I.: Treewidth and minimum fill-in: grouping the minimal separators. SIAM J. Comput. **31**(1), 212–232 (2001)
6. Bouchitté, V., Todinca, I.: Listing all potential maximal cliques of a graph. Theoret. Comput. Sci. **276**(1–2), 17–32 (2002)
7. Cygan, M., Nederlof, J., Pilipczuk, M., Pilipczuk, M., van Rooij, J.M.M., 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), pp. 150–159 (2011)
8. Cygan, M., Fomin, F.V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S.: Parameterized Algorithms. Springer International Publishing, Switzerland (2015)
9. Fomin, F.V., Kratsch, D., Todinca, I., Villanger, Y.: Exact algorithms for treewidth and minimum fill-in. SIAM J. Comput. **38**(3), 1058–1079 (2008)
10. Gaspers, S., Mackenzie, S.: On the number of minimal separators in graphs. arXiv:1503.01203v2 (2015)
11. Kloks, T. (ed.): Treewidth: Computations and Approximations. LNCS, vol. 842. Springer, Heidelberg (1994)
12. Kloks, T.: Treewidth of circle graphs. Int. J. Found. Comput. Sci. **7**(2), 111 (1996)
13. Mulmuley, K., Vazirani, U.V., Vazirani, V.V.: Matching is as easy as matrix inversion. Combinatorica **7**(1), 105–113 (1987)
14. Rooij, J.M.M., Bodlaender, H.L., Rossmanith, P.: Dynamic programming on tree decompositions using generalised fast subset convolution. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 566–577. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04128-0_51
15. Skodinis, K.: Efficient analysis of graphs with small minimal separators. In: Widmayer, P., Neyer, G., Eidenbenz, S. (eds.) WG 1999. LNCS, vol. 1665, pp. 155–166. Springer, Heidelberg (1999). doi:10.1007/3-540-46784-X_16
16. Suchan, K.: Minimal separators in intersection graphs. Masters thesis, Akademia Gorniczo- Hutnicza im. Stanislawa Staszica w Krakowie, Cracow (2003)
17. Sundaram, R., Singh, K.S., Rangan, C.P.: Treewidth of circular-arc graphs. SIAM J. Discrete Math. **7**(4), 647–655 (1994)