Heuristic Computation of Exact Treewidth

Hisao Tamaki
Department of Computer Science, Meiji University, Tokyo, Japan

Abstract

We are interested in computing the treewidth $\mathrm{tw}(G)$ of a given graph G. Our approach is to design heuristic algorithms for computing a sequence of improving upper bounds and a sequence of improving lower bounds, which would hopefully converge to $\mathrm{tw}(G)$ from both sides. The upper bound algorithm extends and simplifies the present author's unpublished work on a heuristic use of the dynamic programming algorithm for deciding treewidth due to Bouchitté and Todinca. The lower bound algorithm is based on the well-known fact that, for every minor H of G, we have $\mathrm{tw}(H) \leq \mathrm{tw}(G)$. Starting from a greedily computed minor H_0 of G, the algorithm tries to construct a sequence of minors H_0, H_1, $\ldots H_k$ with $\mathrm{tw}(H_i) < \mathrm{tw}(H_{i+1})$ for $0 \leq i < k$ and hopefully $\mathrm{tw}(H_k) = \mathrm{tw}(G)$.

We have implemented a treewidth solver based on this approach and have evaluated it on the bonus instances from the exact treewidth track of PACE 2017 algorithm implementation challenge. The results show that our approach is extremely effective in tackling instances that are hard for conventional solvers. Our solver has an additional advantage over conventional ones in that it attaches a compact certificate to the lower bound it computes.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases graph algorithm, treewidth, heuristics, BT dynamic programming, contraction, obstruction, minimal forbidden minor, certifying algorithms

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.17

Acknowledgements I thank Holger Dell for posing the challenging bonus instances, which have kept defying my "great ideas", showing how they fail, and pointing to yet greater ideas.

1 Introduction

Treewidth is a graph parameter which plays an essential role in the graph minor theory [15, 16, 17] and is an indispensable tool in designing graph algorithms (see, for example, a survey [6]). See Section 2 for the definition of treewidth and tree-decompositions. Let $\mathrm{tw}(G)$ denote the treewidth of graph G. Deciding if $\mathrm{tw}(G) \leq k$ for given G and k is NP-complete [2], but admits a fixed-parameter linear time algorithm [5].

Practical algorithms for treewidth have also been actively studied [8, 9, 3, 20, 18, 1], with recent progresses stimulated by PACE 2016 and 2017 [13] algorithm implementation challenges. The modern treewidth solvers use efficient implementations of the dynamic programming algorithm due to Boudhitté and Todinca (BT) [10]. After a first leap in that direction [20], some improvements have been reported [18, 1], but those improvements are incremental.

In this paper, we pursue a completely different approach. We develop a heuristic algorithm for the upper bound as well as one for the lower bound. These algorithms iteratively improve the bounds in hope that they converge to the exact treewidth from both sides.

Our upper bound algorithm is based on the following idea. For $\mathcal{B} \subseteq 2^{V(G)}$, we say that $\mathcal{B}$ admits a tree-decomposition of G if every bag of this tree-decomposition belongs to $\mathcal{B}$. The treewidth of G with respect to $\mathcal{B}$, denoted by $\mathrm{tw}_{\mathcal{B}}(G)$, is the smallest k such that $\mathcal{B}$ admits a tree-decomposition of G of width k. If $\mathcal{B}$ admits no tree-decomposition of G, then $\mathrm{tw}_{\mathcal{B}}(G)$ is undefined. A vertex set $X \subseteq V(G)$ is a *potential maximal clique* of G if it is a maximal clique of some minimal triangulation of G. We denote by $\Pi(G)$ the set of all potential maximal

20th International Symposium on Experimental Algorithms (SEA 2022).
Editors: Christian Schulz and Bora Uçar; Article No. 17; pp. 17:1–17:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

cliques of G. Bouchitté and Todinca [10] observe that $\Pi(G)$ admits a tree-decomposition of G of width $\mathrm{tw}(G)$ and present a dynamic programming algorithm (BT dynamic programming) to compute $\mathrm{tw}(G)$ based on this fact. Indeed, BT dynamic programming can be applied to an arbitrary set Π of potential maximal cliques to compute $\mathrm{tw}_\Pi(G)$. This allows us to work in a solution space where each solution is a set of potential maximal cliques rather than an individual tree-decomposition. A solution Π encodes a potentially exponential number of tree-decompositions it admits and offers rich opportunities of improvements in terms of $\mathrm{tw}_\Pi(G)$. This approach has been proposed by the present author in his unpublished work [19], where he presents several *ad hoc* operations to enrich Π in hope of reducing $\mathrm{tw}_\Pi(G)$. We extend and simplify this approach by replacing those operations with a single merging operation: given two sets Π_1 and Π_2 of potential maximal cliques, we construct a new set Π that includes $\Pi_1 \cup \Pi_2$ together with some additional potential maximal cliques potentially useful for making $\mathrm{tw}_\Pi(G)$ smaller than both $\mathrm{tw}_{\Pi_1}(G)$ and $\mathrm{tw}_{\Pi_2}(G)$. See Section 3 for more details.

The lower bound algorithm is based on the well-known fact that, for every minor H of G, we have $\mathrm{tw}(H) \leq \mathrm{tw}(G)$. Starting from a greedily computed minor H_0 of G, the algorithm tries to construct a sequence of minors H_0, H_1, $\ldots H_k$ with $\mathrm{tw}(H_i) < \mathrm{tw}(H_{i+1})$ for $0 \leq i < k$ and hopefully $\mathrm{tw}(H_k) = \mathrm{tw}(G)$. Although minors have been used to compute lower bounds on the treewidth [9], the goal has been to quickly obtain a lower bound of reasonable quality to be used, say, in branch-and-bound procedures. There seems to be no attempt in the literature to develop an algorithm which, given a minor H of G with $\mathrm{tw}(H) < \mathrm{tw}(G)$, construct a minor H' of G with an improved lower bound $\mathrm{tw}(H') > \mathrm{tw}(H)$. In view of this lack of attempts, our finding that this task can be performed with reasonable efficiency in practice might be somewhat surprising. See Section 4 for details.

We have implemented a treewidth solver based on this approach and evaluated it on the bonus instance set from the PACE 2017 algorithm implementation challenge for treewidth. This set is designed to remain challenging for solvers to be developed after the challenge. It consists of 100 instances and, according to the summary provided with the set, the time spent to compute the exact treewidth by the winning solvers of PACE 2017 is longer than an hour for 57 instances and longer than 12 hours for 23 instances, including 9 instances which fail to be solved at all. The results of applying our solver on the 91 solved instances are summarized as follows. With a timeout of 30 minutes using two threads (one for the upper bound and the other for the lower bound), 62 instances are exactly solved; for 20 of the other instances, the upper bound equals the exact treewidth and the lower bound is off by one. Moreover, with a timeout of 6 hours, our solver exactly solves 2 of the 9 unsolved instances. These results suggest that our approach is extremely effective in coping with instances that are hard for conventional solvers. See Section 6 for details.

The source code of the solver used in our experiments is available at [22].

2 Preliminaries

Graph notation

In this paper, all graphs are simple, that is, without self loops or parallel edges. Let G be a graph. We denote by $V(G)$ the vertex set of G and by $E(G)$ the edge set of G. As G is simple, each edge of G is a subset of $V(G)$ with exactly two members that are adjacent to each other in G. The *complete graph* on V, denoted by $K(V)$, is a graph with vertex set V in which every vertex is adjacent to all other vertices. The subgraph of G induced by $U \subseteq V(G)$ is denoted by $G[U]$. We sometimes use an abbreviation $G \setminus U$ to stand for $G[V(G) \setminus U]$.

A vertex set $C \subseteq V(G)$ is a *clique* of G if $G[C]$ is a complete graph. For each $v \in V(G)$, $N_G(v)$ denotes the set of neighbors of v in G: $N_G(v) = \{u \in V(G) \mid \{u, v\} \in E(G)\}$. For $U \subseteq V(G)$, the *open neighborhood of* U *in* G, denoted by $N_G(U)$, is the set of vertices adjacent to some vertex in U but not belonging to U itself: $N_G(U) = (\bigcup_{v \in U} N_G(v)) \setminus U$. The *closed neighborhood of* U *in* G, denoted by $N_G[U]$, is defined by $N_G[U] = U \cup N_G(U)$.

We say that vertex set $C \subseteq V(G)$ is *connected in* G if, for every $u, v \in C$, there is a path in $G[C]$ between u and v. It is a *connected component* or simply a *component* of G if it is connected and is inclusion-wise maximal subject to this condition. A vertex set $S \subseteq V(G)$ is a *separator* of G if $G \setminus S$ has more than one component. A graph is a *cycle* if it is connected and every vertex is adjacent to exactly two vertices. A graph is a *forest* if it does not have a cycle as a subgraph. A forest is a *tree* if it is connected.

Tree-decompositions

A *tree-decomposition* of G is a pair $(T, \mathcal{X})$ where T is a tree and $\mathcal{X}$ is a family $\{X_i\}_{i \in V(T)}$ of vertex sets of G, indexed by the nodes of T, such that the following three conditions are satisfied. We call each X_i the *bag* at node i.

1. $\bigcup_{i \in V(T)} X_i = V(G)$.
2. For each edge $\{u, v\} \in E(G)$, there is some $i \in V(T)$ such that $u, v \in X_i$.
3. For each $v \in V(G)$, the set of nodes $I_v = \{i \in V(T) \mid v \in X_i\} \subseteq V(T)$ is connected in T.

The *width* of this tree-decomposition is $\max_{i \in V(T)} |X_i| - 1$. The *treewidth* of G, denoted by $\mathrm{tw}(G)$ is the smallest k such that there is a tree-decomposition of G of width k.

It is well-known that, for each pair (i, j) of adjacent nodes of a tree-decomposition $\mathcal{T} = (T, \mathcal{X})$, the intersection $X_i \cap X_j$ is a separator of G. We say that $\mathcal{T}$ *induces* separator S if there is an adjacent pair (i, j) such that $S = X_i \cap X_j$.

Minimal separators and potential maximal cliques

Let G be a graph and S a separator of G. For distinct vertices $a, b \in V(G)$, S is an *a-b separator* if there is no path between a and b in $G \setminus S$; it is a *minimal a-b separator* if it is an a-b separator and no proper subset of S is an a-b separator. A separator is a *minimal separator* if it is a minimal a-b separator for some $a, b \in V(G)$.

Graph H is *chordal* if every induced cycle of H has exactly three vertices. H is a *triangulation of graph* G if it is chordal, $V(G) = V(H)$, and $E(G) \subseteq E(H)$. A triangulation H of G is *minimal* if it there is no triangulation H' of G such that $E(H')$ is a proper subset of $E(H)$. A vertex set $X \subseteq V(G)$ is a *potential maximal clique* of G, if X is a maximal clique in some minimal triangulation of G. We denote by $\Pi(G)$ the set of all potential maximal cliques of G and by $\Pi_k(G)$ the set of all potential maximal cliques of G of cardinality at most k.

Bouchitté-Todinca dynamic programming

The treewidth algorithm of Bouchitté and Todinca [10] is based on the fact that every graph G has a minimal triangulation H such that $\mathrm{tw}(H) = \mathrm{tw}(G)$ (see [14] for a clear exposition). This fact straightforwardly implies that $\Pi(G)$ admits an optimal tree-decomposition of G. Their algorithm consists of an algorithm for constructing $\Pi(G)$ and a dynamic programming algorithm (BT dynamic programming) to compute $\mathrm{tw}_{\Pi(G)}(G)$. As noted in the introduction, BT dynamic programming can be applied to compute $\mathrm{tw}_{\Pi}(G)$ for an arbitrary $\Pi \subseteq \Pi(G)$.

The most time-consuming part of their treewidth algorithm is the construction of $\Pi(G)$. Empirically observing that $\Pi_{k+1}(G)$ is substantially smaller than $\Pi(G)$ for $k \leq \mathrm{tw}(G)$, authors of modern implementations of the BT algorithm [20, 18, 1] use BT dynamic programming with $\Pi = \Pi_{k+1}(G)$ to decide if $\mathrm{tw}(G) \leq k$. Moreover, they try to avoid the full generation of $\Pi_{k+1}(G)$, by being lazy and generating a potential maximal clique only when it becomes absolutely necessary in evaluating the recurrence.

Both our upper and lower bound algorithms use, as a subprocedure, such an implementation of the BT algorithm for treewidth, in particular an implementation of the version proposed in [18]. In addition, our upper bound algorithm uses BT dynamic programming in its fully general form, to evaluate each solution Π in our solution space as described in the introduction. The efficiency of BT dynamic programming, which runs in time linear in $|\Pi|$ with a factor polynomial in $|V(G)|$, is crucial in our upper bound algorithm.

Contractions and minors

Let $F \subseteq E(G)$ be a forest on $V(G)$. The *contraction* of G by F, denoted by G/F is a graph whose vertices are the connected components of F and two components C_1 and C_2 are adjacent to each other if and only if there is $v_1 \in C_1$ and $v_2 \in C_2$ such that v_1 and v_2 are adjacent to each other in G. A graph H is a *minor* of G if it is a subgraph of some contraction of G. It is well-known and is easy to verify that $\mathrm{tw}(H) \leq \mathrm{tw}(G)$ if H is a minor of G.

Minimal triangulation algorithms

There are many algorithms for minimal triangulation of a graph (see [14] for a survey). For purposes in the current work, we are interested in algorithms that produce a minimal triangulation of small treewidth. Although a minimal triangulation H of G such that $\mathrm{tw}(H) = \mathrm{tw}(G)$, hence of the smallest treewidth, can be computed by the BT algorithm for treewidth, we need a faster heuristic algorithm. The MMD (Minimal Minimum Degree) algorithm [4] is known to perform well, in terms of the treewidth of the resulting triangulation. We use a variant MMAF (Minimal Minimum Average Fill) [21] of MMD which performs slightly better than the original MMD on benchmark instances.

Safe separators and almost-clique separators

Bodlaender and Koster [7] introduced the notion of safe separators for treewidth. Let S be a separator of a graph G. We say that S is *safe for width* k, if S is induced by some tree-decomposition of G of width k. It is simply safe if it is safe for width $\mathrm{tw}(G)$. The motivation of looking at safe separators is the fact that there are easily verifiable sufficient conditions for S being safe and a safe separator detected by those sufficient conditions can be used to reduce the problem of deciding if $\mathrm{tw}(G) \leq k$ to smaller subproblems. A trivial sufficient condition is that S is a clique. Bodlaender and Koster observed that this condition can be relaxed to S being an *almost-clique*, where S is an almost-clique if $S \setminus \{v\}$ is a clique for some $v \in S$. More precisely, a minimal separator that is an almost-clique is safe. They showed that this observation leads to a powerful preprocessing method of treewidth computation. An *almost-clique separator decomposition* of graph G is a tree-decomposition $\mathcal{A}$ of G such that every separator induced by $\mathcal{A}$ is an almost-clique minimal separator. For each bag A_i of $\mathcal{A}$, let G_i be a graph on A_i obtained from $G[A_i]$ by adding edges of $K(N(C))$ for every component C of $G \setminus A_i$. The following proposition holds [7].

► **Proposition 1.**

1. $\mathrm{tw}(G)$ *is the maximum of* $\mathrm{tw}(G_i)$*, where* i *ranges over the nodes of the decomposition, and a tree-decomposition of* G *of width* $\mathrm{tw}(G)$ *is obtained by combining tree-decompositions of* G_i *as prescribed by* $\mathcal{A}$*.*
2. G_i *is a minor of* G *for each* i*.*

Unpublished work of the present author [21] shows that this preprocessing approach is effective for instances that are much larger than those tested in [7], using a heuristic method for constructing almost-clique separator decompositions. We use his implementation in the current work.

We also make an unconventional use of safe separators in our lower bound algorithm. When we have a lower bound of k on $\mathrm{tw}(G)$, we wish to evaluate a minor H of G for the possibility of leading to a stronger lower bound. We use the set of all minimal separators of H that are safe for width k in this evaluation. Note that the computation of such a set is possible because H is small.

3 The upper bound algorithm

Recall that $\Pi(G)$ denotes the set of all potential maximal cliques of G. In our upper bound algorithm, a *solution* for G is a subset Π of $\Pi(G)$ that admits at least one tree-decomposition of G and the *value* of solution Π is $\mathrm{tw}_\Pi(G)$.

Our algorithm starts from a greedy solution and iteratively improves the solution. To improve a solution Π, we *merge* it with another solution Ω into another solution Π' in hope of having $\mathrm{tw}_{\Pi'}(G) < \min\{\mathrm{tw}_\Pi(G), \mathrm{tw}_\Omega(G)\}$. This merged solution Π' contains $\Pi \cup \Omega$ together with some other potential maximal cliques so that Π' would admit a tree-decomposition that contains some bags in Π, some bags in Ω, and some bags belonging to this additional set of potential maximal cliques. We describe below how these additional potential maximal cliques are computed.

Let $X \in \Pi$ and $Y \in \Omega$ be distinct and not crossing each other. Then, there is a unique component C of $G \setminus X$ such that $Y \subseteq N[C]$ and a unique component D of $G \setminus Y$ such that $X \subseteq N[D]$. Let $U = N[C] \cap N[D]$ and let H be a graph on U obtained from $G[U]$ by adding edges of $K(N(B))$ for each component B of $G \setminus U$. Let $\hat{H}$ be a minimal triangulation of H with small treewidth: if $|U|$ does not exceed a fixed threshold BASE_SIZE (=60), then we use the BT algorithm for treewidth to compute $\hat{H}$; otherwise we use MMAF to compute $\hat{H}$. Here, BASE_SIZE is a parameter of the algorithm represented as a constant in the implementation. The parenthesized number is the value of this parameter used in our experiment. In the following, we use the same convention for citing algorithm parameters. Note that each maximal clique of $\hat{H}$ is either a potential maximal clique of G or a minimal separator of G. If $\mathrm{tw}(\hat{H}) \leq \mathrm{tw}_\Pi(G)$, then we add all potential maximal cliques of G that are maximal cliques of $\hat{H}$ to Π'. When this happens, then Π' admits a tree-decomposition of width at most $\max\{\mathrm{tw}_\Pi(G), \mathrm{tw}_\Omega(G)\}$ consisting of some bags in Π, some bags in Ω, and some bags that are maximal cliques of $\hat{H}$. In this way, Π' would admit tree-decompositions that are not admitted by the simple union $\Pi \cup \Omega$ and, with some luck, some of the newly admitted tree-decomposition may have width smaller than $\mathrm{tw}_\Pi(G)$.

The procedure MERGE(Π, Ω) merges Π with Ω, applying the above operation to some pairs $X \in \Pi$ and $Y \in \Omega$ chosen as follows. We first pick $X \in \Pi$ uniformly at random. Then, let C be the largest component of $G \setminus X$. We choose $Y \in \Omega$ such that $Y \subseteq N[C]$ and $|Y| \leq \mathrm{tw}_\Pi(G)$. The first condition ensures that the method in the previous paragraph can be applied to this pair of X and Y. The second condition is meant to increase the chance of newly admitted tree-decompositions to have width smaller than $\mathrm{tw}_\Pi(G)$. We sort the candidates of such Y in the nondecreasing order of $|N[C] \cap N[D]|$, where D is the component of Y such that $X \subseteq N[D]$, and use the first N_TRY (=50) elements of this sorted list. We prefer Y such that $U = N[C] \cap N[D]$ is small, because that would increase the chance of the minimal triangulation $\hat{H}$, described in the previous paragraph, to have small treewidth. All the resulting potential maximal cliques from these trials are added to Π'.

In addition to procedure Merge, our algorithm uses two more procedures InitialSolution() and Improve(Π) described below. The input graph G is fixed in these procedures.

InitialSolution**()** generates an initial solution Π. We use a randomize version of MMAF to generate N_INITIAL_GREEDY(=10) minimal triangulations of G and take H with the smallest treewidth. The solution Π returned by the call InitialSolution() is the set of maximal cliques of H.

Improve**(Π)** returns a solution Π' with $\Pi \subseteq \Pi'$, where efforts are made to make $\mathrm{tw}_{\Pi'}(G)$ strictly smaller than $\mathrm{tw}_{\Pi}(G)$. We proceed in the following steps.

1. Let $\Omega =$ InitialSolution().
2. While $\mathrm{tw}_{\Omega}(G)) > \mathrm{tw}_{\Pi}(G)$, replace Ω by Improve(Ω).
3. Return Merge(Π, Ω).

Note the solution Ω to be merged with Π is generated independently of Π and improved so that $\mathrm{tw}_{\Omega}(G) \leq \mathrm{tw}_{\Pi}(G)$ before being merged with Π.

Given these procedures, the main iteration of our algorithm proceeds as follows. It is supposed that the algorithm has an access to lower bounds provided by the lower bound algorithm.

1. Let $\Pi =$ InitialSolution(). Report $\mathrm{tw}_{\Pi}(G)$ as the initial upper bound on $\mathrm{tw}(G)$, together with a tree-decomposition of G of width $\mathrm{tw}_{\Pi}(G)$ admitted by Π.
2. While $\mathrm{tw}_{\Pi}(G)$ is greater than the current lower bound, replace Π by Improve(Π). When this replacement reduces $\mathrm{tw}_{\Pi}(G)$, report this new upper bound on $\mathrm{tw}(G)$, together with a tree-decomposition of G of width $\mathrm{tw}_{\Pi}(G)$ admitted by Π. We also shrink Π, removing all members of cardinality greater than $k+2$, whenever $\mathrm{tw}_{\Pi}(G)$ is improved to k.

4 The lower bound algorithm

In our lower bound algorithm, we use a procedure we call Lift, which, given a graph G and a forest F on $V(G)$, finds another forest F' such that $\mathrm{tw}(G/F') > \mathrm{tw}(G/F)$; it inevitably fails if $\mathrm{tw}(G/F) = \mathrm{tw}(G)$. Given this procedure, the overall lower bound algorithm proceeds in the following steps. Let G be given. It is supposed that the algorithm has an access to the upper bound being computed by the upper bound algorithm.

1. Construct a contraction G/F of G, using a greedy heuristic for contraction-based lower bounds on treewidth.
2. While $\mathrm{tw}(G/F)$ is smaller than the current upper bound on $\mathrm{tw}(G)$, replace F by Lift(G, F) unless this call fails.

When the current upper bound is larger than $\mathrm{tw}(G)$, it is possible that the call Lift(G, F) is made for F such that $\mathrm{tw}(G/F) = \mathrm{tw}(G)$. In such an event, the call would eventually fail but the time it takes would be at least as the time taken by conventional solvers to compute $\mathrm{tw}(G)$. Our solver implements a mechanism to let such a call terminate as soon as the upper bound is updated to be equal to the current lower bound $\mathrm{tw}(G/F)$.

The design of procedure Lift is described in the following subsections.

4.1 Contraction lattice

First consider looking for the result of Lift(G, F) among the subsets of F. Assuming that $\mathrm{tw}(G/F) < \mathrm{tw}(G)$, a subset F' of F such that $\mathrm{tw}(G/F') > \mathrm{tw}(G/F)$ certainly exists.

For each $A \in 2^F$, let H_A denote the contraction $G/(F \setminus A)$. Then, $\Lambda(G, F) = \{H_A \mid A \in 2^F\}$ is a lattice isomorphic to the power set lattice 2^F, with top G and bottom G/F. Brute force searches for H with $\mathrm{tw}(H) > \mathrm{tw}(G/F)$ in this lattice are hopeless as $|F|$ can be large: we typically have $|F| > 100$ for graph instances we target.

We need to understand the terrain of this lattice to design an effective search method. In the remainder of this subsection and subsequent subsections, let k denote $\mathrm{tw}(G/F)$. Call $H \in \Lambda(G, F)$ *lifted* if $\mathrm{tw}(H) > k$; otherwise call it *unlifted*. Let $\mathrm{ML}(G, F)$ denote the set of minimal lifted elements of $\Lambda(G, F)$. Call an unlifted element H_A *covered* if there is some $H_B \in \mathrm{ML}(G, F)$ with $A \subset B$. Let $\mathrm{COV}(G, F)$ denote the set of all covered unlifted elements of $\Lambda(G, F)$. Ideally, we wish to confine our search in $\mathrm{COV}(G, F) \cup \mathrm{ML}(G, F)$. This would be possible if there is a way of knowing, for each covered element H_A and $e \in F \setminus A$, if $H_{A \cup \{e\}}$ is still covered. Then, we would start with the clearly covered element $H_\emptyset$ and greedily ascend the lattice staying in $\mathrm{COV}(G, F)$ until we hit an element in $\mathrm{ML}(G, F)$. Although such an ideal scenario is unlikely to be possible, we still aim at something close to it in the following sense. For each $A \in 2^F$ such that H_A is unlifted, call $S \subseteq A$ an *excess* in A if $H_{A \setminus S}$ is covered. We wish to confine our search among elements with a small excess. We employ a strategy that works only for pairs (G, F) with a special property, which is described in the following subsections.

4.2 Critical fills

Call a pair $\{u, v\}$ of distinct vertices of G a *fill* of G if u and v are not adjacent to each other in G. For a fill e of G, let $G + e$ denote the graph on $V(G)$ with edge set $E(G) \cup \{e\}$. We say that a fill e of G is *critical for F* if $\mathrm{tw}((G + e)/F) > \mathrm{tw}(G/F)$.

Suppose G has a critical fill for F. Observe the following.

1. Since adding a single edge to G/F increases its treewidth, a small number of uncontractions applied to G/F may suffice to increase its treewidth. Thus, we may expect that there is a lifted element H_A in the lattice $\Lambda(G, F)$ such that $|A|$ is smaller than the value that is expected in a general case (without a critical fill). This would make our search for a lifted element easier.
2. The fact that e is a critical fill could be used to guide our search for a lifted element.

The next section describes how we exploit the existence of a critical fill to guide our search.

4.3 Breaking a critical fill

Assuming that G has a fill $e = \{u, v\}$ critical for F, we look for a lifted element H_A in the lattice $\Lambda(G, F)$. We call the procedure for this operation $\textsc{BreakFill}(F, e)$. The reason of this naming is that, informally speaking, we remove the fill e from $G + e$ by uncontracting some edges in F maintaining the treewidth. We need some preparations before we describe this procedure.

▶ **Proposition 2.** *If H_A is unlifted then e is critical for $F \setminus A$.*

Proof. Since H_A is unlifted, we have $\mathrm{tw}(G/(F \setminus A)) = k$. So, it suffices to show that $\mathrm{tw}((G + e)/(F \setminus A)) > k$. We have $\mathrm{tw}((G + e)/F) > k$, since e is critical for F. We also have $\mathrm{tw}((G + e)/(F \setminus A)) \geq \mathrm{tw}((G + e)/F)$, since $(G + e)/F$ is a contraction of $(G + e)/(F \setminus A)$. Therefore we have $\mathrm{tw}((G + e)/(F \setminus A)) > k$. ◀

Let u_A (v_A) denote the vertex of H_A into which u (v, respectively) is contracted. We say that a separator S of H_A *crosses* e if u_A and v_A belong to two distinct components of $H_A \setminus S$. Define $\text{ncs}_k(A, e)$ to be the number of minimal separators of H_A that are safe for width k and moreover cross e. Observe the following.

1. We have $\text{ncs}_k(A, e) > 0$ if H_A is unlifted. To see this, suppose otherwise that H_A is unlifted but $\text{ncs}_k(A) = 0$. Then, H_A has a tree-decomposition $\mathcal{T}$ of width k such that none of the minimal separators induced by $\mathcal{T}$ crosses e. Then, $\mathcal{T}$ is a tree-decomposition of $(G + e)/(F \setminus A)$ as well, contradicting the assumption that e is critical for F and hence for $F \setminus A$ by Proposition 2.
2. We have $\text{ncs}_k(A, e) = 0$ if H_A is lifted. This is simply because H_A does not have any minimal separator that is safe for treewidth k if $\text{tw}(H_A) > k$.

We empirically observe a tendency that $\text{ncs}_k(A, e)$ decreases as H_A approaches a lifted element from below. Based on this observation, we use this function ncs_k to guide our search for lifted elements. We are ready to describe our procedure BreakFill(F, e). It involves two parameters UNC_CHUNK (=5) and N_TRY (=100) and proceeds as follows.

1. Let $A = \emptyset$.
2. While H_A is unlifted, repeat the following:
   a. Pick N_TRY random supersets A' of A with cardinality $|A|$ + UNC_CHUNK (or $|F|$ if this exceeds $|F|$) and let A_{best} be A' such that $\text{ncs}_k(A', e)$ is the smallest.
   b. Replace A by A_{best}.
3. Return A.

This procedure is correct in a purely theoretical sense: since $H_F = G$ is always lifted, provided $\text{tw}(G/F) < \text{tw}(G)$, it eventually returns some A such that H_A is lifted. The time required for this to happen, however, can be prohibitively long since we are supposing that $|F|$ is fairly large. The success of this procedure hinges on the effectiveness of our heuristic relying on the critical fill.

4.4 Procedure Lift

Our procedure Lift(G, F) is recursive and works in the following steps.

1. Choose a fill e of G with the following heuristic criterion. For each $v \in V(G)$, let $d_F(v)$ denote the degree of v' in G/F, where v' is the vertex of G/F into which v contracts. Then we choose $e = \{u, v\}$ so as to maximize the pair $(d_F(u), d_F(v))$ in the lexicographic ordering, where the order of u and v is chosen so that $d_F(u) \leq d_F(v)$.
2. Let F_1 =Lift($(G + e)/F$). If $\text{tw}(G/F_1) > \text{tw}(G/F)$ then return F_1; otherwise, observing that e is critical for F_1, call BreakFill(e, F_1) to find $F_2 \subseteq F_1$ such that $\text{tw}(G/F_2) > \text{tw}(G/F_1)$.
3. Greedily compute a maximal forest F_3 on $V(G)$ such that $F_2 \subseteq F_3$ and $\text{tw}(G/F_3) = \text{tw}(G/F_2)$. Return F_3.

The criterion for choosing e in the first step is based on the following heuristic reasoning. Let u'' (v'') be the vertex of G/F_1 into which u (v, respectively) contracts. We expect that if the size of the minimum vertex cut between u'' and v'' in G/F_1 is large, then the minimum cardinality of A such that H_A is lifted in $\Lambda(G, F_1)$ would have a tendency to be small. Indeed, if the size of this cut is as large as k, then no separator of cardinality at most k crosses e and therefore we have $\text{tw}(H_\emptyset) > 0$. Although we cannot predict the size of the minimum u''-v''

cut in G/F_1 when we are choosing e, having larger degrees of u' and v' in G/F could have some positive influence toward this goal. This criterion, however, has not been evaluated with respect to this goal: further experimental studies are needed here.

We emphasize that Step 3 above is crucial in allowing us to work on relatively small contractions throughout the entire computation. Note also that, due to this step, the result of Lift(G, F) is not a subset of F in general.

5 The overall algorithm

The upper bound algorithm and the lower bound algorithm, together with the preprocessing algorithm, are combined in the following manner. Fix the graph G given.

1. We compute an almost-clique separator decomposition $\mathcal{A}$ of G using the method in [21].
2. For each bag A_i of $\mathcal{A}$, let G_i denote the graph on A_i obtained from $G[A_i]$ by adding edges of $K(N(C))$ for each component C of $G \setminus A_i$. By Proposition 1, the task of computing $\mathrm{tw}(G)$ reduces to the tasks of computing $\mathrm{tw}(G_i)$ for i, for all nodes i of $\mathcal{A}$. Moreover, G_i is a minor of G.
3. Let i^* be such that $|A_{i^*}| \geq |A_i|$ for every node i of $\mathcal{A}$. The lower bound algorithm works on G_{i^*}. When it finds a new lower bound $\mathrm{tw}(G_{i^*}/F)$ on $\mathrm{tw}(G_{i^*})$, this is also a lower bound on $\mathrm{tw}(G)$ since G_{i^*} is a minor of G; we record this new lower bound $\mathrm{tw}(G_{i^*}/F)$ together with the minor G_{i^*}/F of G certifying it.
4. The upper bound algorithm works on G_i for each i, keeping the current solution Π_i of G_i for each i. After initializing Π_i for each i, we start iterations. In each iteration, we choose i_0 to be i such that the current upper bound $\mathrm{tw}_{\Pi_i}(G_i)$ on $\mathrm{tw}(G_i)$ is the largest and replace Π_{i_0} by Improve(Π_{i_0}), where the implicit graph it works on is set to G_i. When the maximum of the upper bounds $\mathrm{tw}_{\Pi_i}(G_i)$ decreases, we record the new upper bound on G together with the tree-decomposition of G combining $\mathcal{A}$ with the currently best tree-decomposition of G_i for all nodes i of $\mathcal{A}$.
5. As described in the previous sections, the upper bound algorithm and the lower bound algorithm have access to the current bound computed by each other and terminate when they match.

6 Experiments

We have evaluated our solver by experiments. The computing environment for our experiments is as follows. CPU: Intel Core i7-8700K, 3.70GHz; RAM: 64GB; Operating system: Windows 10Pro, 64bit; Programming language: Java 1.8; JVM: jre1.8.0_271. The maximum heap size is set to 60GB. The solver uses two threads, one for the upper bound and the other for the lower bound, although more threads may be invoked for garbage collection by JVM.

As described in the previous sections, both of the upper and lower bound algorithms use BT dynamic programming for deciding the treewidth, and enumerating the safe separators, of small graphs. Our solver uses an implementation of the semi-PID version of this algorithm [18], which is available at the same github repository [22] in which the entire source code of our solver is posted.

The upper bound computation uses a single sequence of pseudo-random numbers and the lower bound computation uses another independent single sequence. The initial seed is set to a fixed value of 1 for both of these sequences, for the sake of reproducibility. With this setting, our solver can be considered deterministic.

We use the bonus instance set from the exact treewidth track of PACE 2017 algorithm implementation challenge [13]. This set of instances are available at [12]. We quote the note by Holgar Dell, the PACE 2017 organizer, explaining his intention to compile these instances.

> The instance set used in the exact treewidth challenge of PACE 2017 is now considered to be too easy. Therefore, this bonus instance set has been created to offer a fresh and difficult challenge. In particular, solving these instances in five minutes would require a 1000x speed improvement over the best exact treewidth solvers of PACE 2017.

The set consists of 100 instances and their summary, available at [12] in csv format, lists each instance with the time spent for solving it and its exact treewidth if the computation is successful. According to this summary, the exact treewidth is known for 91 of those instances; the remaining 9 instances are unsolved.

Tables 1 and 2 show the list of those 91 solved instances. We number them in the increasing order of the computation time provided in the summary. Columns “n”, “m”, and “tw” give the number of vertices, the number of edges, and the treewidth, respectively, of each instance.

We see that some of these instances are indeed challenging. Even though they have been solved, they require more than a day to solve. We also note that, to date, no new solvers has been published that have overcome the challenge posed by this instance set.

We have run our solver on these instances with the timeout of 30 minutes. Figure 1 (instance No.1 – No.45) and Figure 2 (instance No.46 – No.91) show the results. In each column representing an instance G, the box with a blue number plots the time for obtaining the best upper bound computed before timeout, where the non-negative number d in the box indicates that this bound is $\mathrm{tw}(G) + d$. Similarly, the box with a red non-positive number d plots the time for obtaining the best lower bound, which is $\mathrm{tw}(G) + d$.

We see from these figures that the bounds computed by our solver are quite tight for most instances. Let us say that the result for instance G is of type (d_1, d_2) if the lower bound (upper bound) obtained by our solver is $\mathrm{tw}(G) + d_1$ ($\mathrm{tw}(G) + d_2$, respectively). Then, the results are of type $(0, 0)$ for 62 instances, $(-1, 0)$ for 20 instances, $(-2, 0)$ for 3 instances, $(-3, 0)$ for one instance, $(0, 1)$ for 4 instances, and $(-2, 1)$ for one instance.

We also see from these figures that the performance of our solver on an instance is not so strongly correlated to the hardness of the instance as measured by the time taken by conventional solvers. For example, of the last 6 instances, each of which took more than a day to solve by the PACE 2017 solvers, 5 are exactly solved by our solver and the remaining one has a result of type $(-1, 0)$. Most of the instances with poorer results, with the gap of 2 or 3 between the upper and lower bounds, occur much earlier in the list.

We have also run our solver on the 9 unsolved instances of the bonus set, with 6 hour timeout. It solved 2 of them and, for other 7 instances, the gap between the upper/lower bounds is 2 for 2 instances, 3 for 4 instances, and 7 for one instance.

The certificates of the lower bounds computed by our algorithm are small and easily verified. For each G, let $\mathrm{lb}(G)$ denote the best lower bound computed by our algorithm before the timeout of 30 minutes and let $\mathrm{nc}(G)$ denote the number of vertices of the certificate for this lower bound. Figures 3 and 4 show $\mathrm{lb}(G)$, $\mathrm{nc}(G)$, and the time to verify the certificate using our implementation of the BT algorithm, for each instance G. The average, the minimum, and the maximum of the ratio $\mathrm{nc}(G)$ / $\mathrm{lb}(G)$ over the 91 solved instances are 2.32, 1.25, and 3.9 respectively. The maximum of $\mathrm{nc}(G)$ over these instances is 49 and the time for verifying each certificate is at most 400 milliseconds.

Table 1 bonus instances with known treewidth (first half).

no.	name	n	m	tw	time
1	Sz512_15127_1.smt2-stp212.gaifman_3	175	593	14	4.74 seconds
2	MD5-32-1.gaifman_4	225	705	12	7.45 seconds
3	Promedas_69_9	133	251	9	11.5 seconds
4	GTFS_VBB_EndeApr_Dez2016.zip_train+metro_12	103	212	11	18.5 seconds
5	Promedas_56_8	155	299	10	28.3 seconds
6	Promedas_48_5	134	278	11	46.1 seconds
7	minxor128.gaifman_2	231	606	4	1.24 minutes
8	Promedas_49_8	184	367	10	1.92 minutes
9	FLA_14	266	423	8	3.06 minutes
10	post-cbmc-aes-d-r2.gaifman_10	263	505	11	4.32 minutes
11	Pedigree_11_7	202	501	14	4.64 minutes
12	countbitsarray04_32.gaifman_10	331	843	13	4.96 minutes
13	mrpp_4x4#8_8.gaifman_3	106	589	24	5.00 minutes
14	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_9	143	303	13	5.24 minutes
15	Promedus_38_15	208	398	10	5.35 minutes
16	Promedas_50_7	175	362	12	7.40 minutes
17	Promedus_34_11	157	289	11	7.61 minutes
18	GTFS_VBB_EndeApr_Dez2016.zip_train+metro_15	124	250	13	7.73 minutes
19	GTFS_VBB_EndeApr_Dez2016.zip_train+metro_14	123	248	13	7.88 minutes
20	Promedas_43_13	197	354	10	8.09 minutes
21	Promedas_46_8	175	318	11	8.36 minutes
22	Promedus_14_9	173	357	12	9.27 minutes
23	jgiraldezlevy.2200.9086.08.40.41.gaifman_2	95	568	34	9.76 minutes
24	modgen-n200-m90860q08c40-14808.gaifman_2	112	686	35	10.0 minutes
25	Promedus_38_14	242	462	10	11.1 minutes
26	Pedigree_11_6	205	503	14	14.9 minutes
27	Promedas_27_8	165	323	12	15.4 minutes
28	Promedas_45_7	159	313	12	17.4 minutes
29	jgiraldezlevy.2200.9086.08.40.46.gaifman_2	105	658	33	20.4 minutes
30	jgiraldezlevy.2200.9086.08.40.22.gaifman_2	111	675	33	22.2 minutes
31	Promedas_25_8	204	378	11	22.5 minutes
32	Pedigree_12_8	217	531	14	22.8 minutes
33	Promedus_34_12	210	389	11	26.1 minutes
34	Promedas_22_6	200	415	12	28.0 minutes
35	aes_24_4_keyfind_5.gaifman_3	104	380	23	31.5 minutes
36	Promedus_18_8	195	411	13	35.2 minutes
37	6s151.gaifman_3	253	634	14	36.8 minutes
38	LKS_15	220	385	10	39.4 minutes
39	Promedas_23_6	253	500	12	45.3 minutes
40	Promedus_28_14	193	351	11	47.8 minutes
41	Promedas_21_9	253	486	11	50.9 minutes
42	Promedas_59_10	209	396	11	56.3 minutes
43	Promedas_60_11	216	387	11	58.2 minutes
44	Promedas_69_10	194	379	12	1.08 hours
45	newton.3.3.i.smt2-stp212.gaifman_2	119	459	19	1.18 hours

Table 2 bonus instances with known treewidth (second half).

no.	name	n	m	tw	time
46	jgiraldezlevy.2200.9086.08.40.167.gaifman_2	92	552	34	1.22 hours
47	Promedus_34_14	188	352	12	1.30 hours
48	modgen-n200-m90860q08c40-22556.gaifman_2	135	855	33	1.33 hours
49	jgiraldezlevy.2200.9086.08.40.93.gaifman_2	100	593	36	1.33 hours
50	Promedas_61_8	156	305	13	1.48 hours
51	Promedas_30_7	164	320	13	1.62 hours
52	FLA_13	280	456	9	1.66 hours
53	am_7_7.shuffled-as.sat03-363.gaifman_6	189	424	14	1.72 hours
54	LKS_13	293	484	9	1.78 hours
55	SAT_dat.k80-24_1_rule_1.gaifman_3	130	698	22	1.83 hours
56	Promedas_28_10	333	605	11	1.84 hours
57	smtlib-qfbv-aigs-lfsr_004_127_112-tseitin.gaifman_6	316	669	13	1.89 hours
58	Promedas_11_7	191	385	13	2.64 hours
59	Promedus_20_13	193	353	12	3.39 hours
60	Pedigree_13_12	264	646	15	3.50 hours
61	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_10	187	385	14	3.64 hours
62	Promedas_22_8	224	441	13	3.77 hours
63	FLA_15	325	522	9	4.68 hours
64	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_15	198	406	14	4.99 hours
65	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_13	190	390	14	5.35 hours
66	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_12	197	404	14	5.46 hours
67	Promedas_63_8	181	374	14	5.53 hours
68	GTFS_VBB_EndeApr_Dez2016.zip_train+metro+tram_11	192	395	14	5.64 hours
69	Pedigree_12_14	284	703	15	6.18 hours
70	Promedus_12_15	293	533	11	6.19 hours
71	Promedus_12_14	272	494	11	6.63 hours
72	Promedas_44_9	276	534	12	6.65 hours
73	Promedas_32_8	238	487	13	7.05 hours
74	NY_13	283	448	9	7.16 hours
75	Promedus_18_10	187	397	14	8.98 hours
76	Promedas_34_8	174	348	14	9.11 hours
77	Promedas_62_9	217	427	13	9.67 hours
78	Promedus_17_13	180	349	13	11.1 hours
79	Promedus_11_15	247	497	13	12.5 hours
80	Promedas_24_11	273	494	12	13.4 hours
81	Promedus_14_8	199	417	14	15.1 hours
82	am_9_9.gaifman_6	212	480	15	15.7 hours
83	NY_11	226	369	10	17.9 hours
84	mrpp_8x8#24_14.gaifman_3	140	856	28	18.5 hours
85	Pedigree_12_10	286	712	16	22.1 hours
86	Promedas_55_9	221	425	13	27.0 hours
87	Pedigree_12_12	277	683	16	27.9 hours
88	Promedas_46_15	227	416	13	28.4 hours
89	Pedigree_13_9	268	665	16	38.9 hours
90	Promedas_51_12	230	431	13	40.2 hours
91	Promedus_27_15	189	353	13	41.6 hours

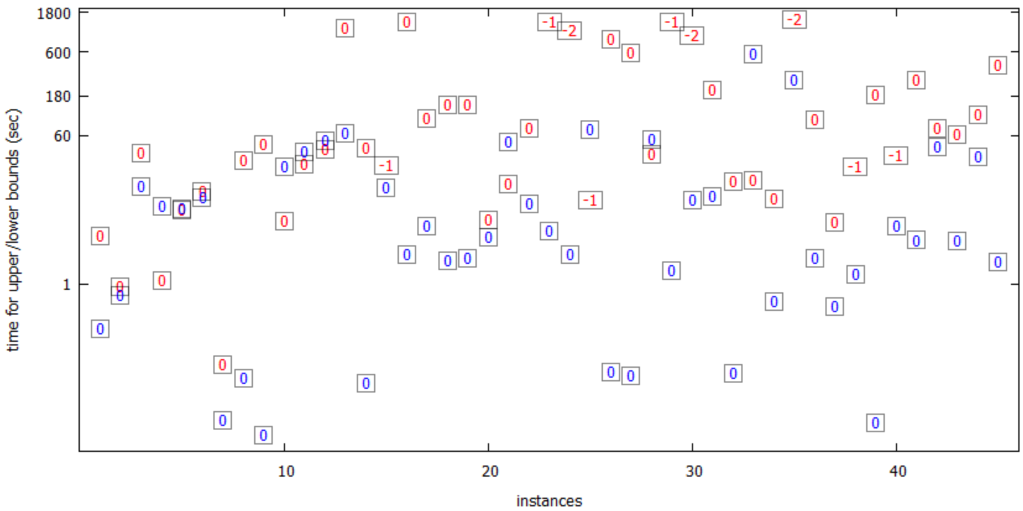

Figure 1 Time for computing upper/lower bounds for instances 1–45.

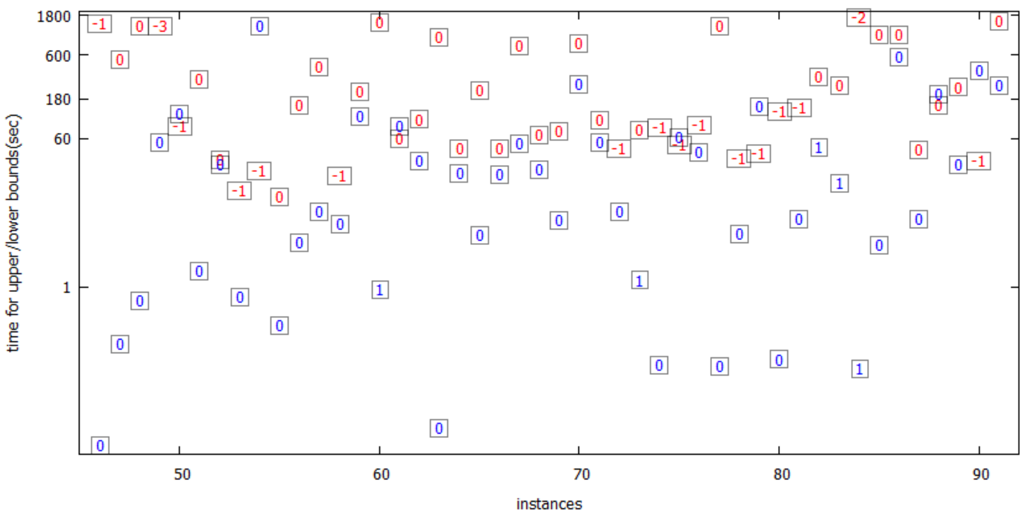

Figure 2 Time for computing upper/lower bounds for instances 46–91.

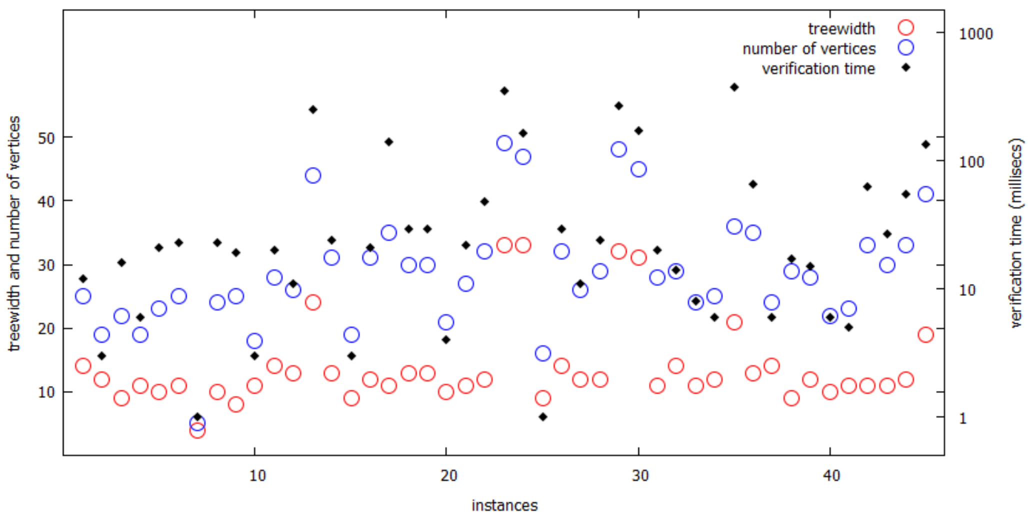

Figure 3 Lower bound certificates for instances 1–45.

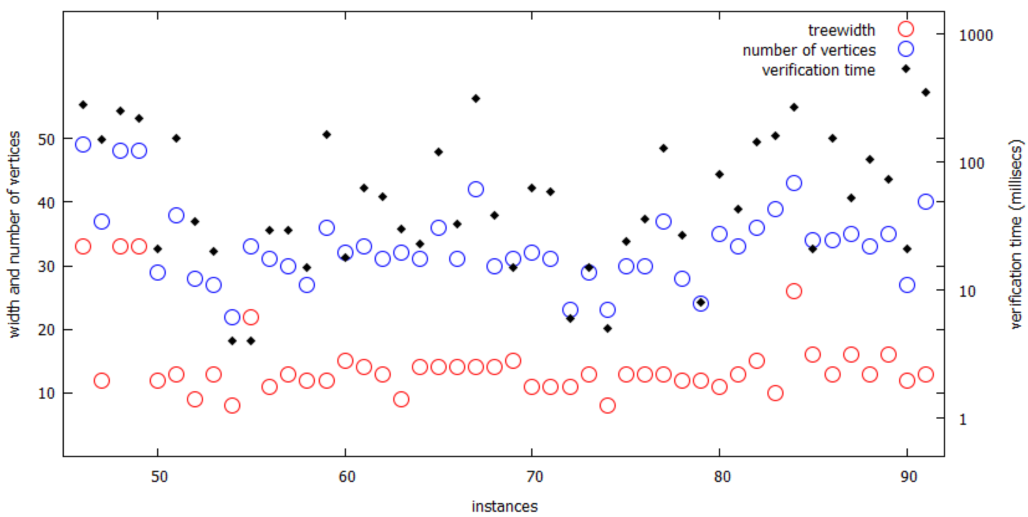

Figure 4 Lower bound certificates for instances 46–91.

7 Conclusions and future work

Our experiments using the bonus instance set from PACE 2017 algorithm implementation challenge have revealed that our approach to treewidth computation is extremely effective in tackling instances that are hard for conventional treewidth solvers. Even when it fails to give the exact treewidth, it produces a lower bound very close to the upper bound. In many applications, such a pair of tight bounds would be satisfactory, since it shows that further search for a better tree-decomposition could only result in a small improvement if at all.

To examine the strength and the weakness of our approach more closely, evaluation on more diverse sets of instances is necessary. For the upper bound part, there are several implementations of heuristic algorithms publicly available, such as the submissions to the heuristic treewidth track of PACE 2017 [11]. Although they are primarily intended for large instances for which exact treewidth appears practically impossible to compute, some of them are nonetheless potential alternatives to our upper bound algorithm. Comparative studies would be needed to determine which algorithm is most suitable for our purposes. On the other hand, it would also be interesting to evaluate our upper bound algorithm on large instances that are the principal targets of those algorithms.

Since our lower approach is new, there are several potential improvements that have not been tried out yet. More work could result in better performances.

We may also ask several theoretical questions regarding our lower bound approach. For example, it would be interesting to ask if the lower bound algorithm can be turned into a fixed parameter tractable algorithm for treewidth. It would also be interesting and useful to identify parameters or structures of graph instances that make them difficult for our lower bound algorithm.

References

1. Ernst Althaus, Daniela Schnurbusch, Julian Wüschner, and Sarah Ziegler. On tamaki's algorithm to compute treewidths. In *19th International Symposium on Experimental Algorithms (SEA 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.
2. Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.
3. Max Bannach, Sebastian Berndt, and Thorsten Ehlers. Jdrasil: A modular library for computing tree decompositions. In *16th International Symposium on Experimental Algorithms (SEA 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
4. Anne Berry, Pinar Heggernes, and Genevieve Simonet. The minimum degree heuristic and the minimal triangulation process. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 58–70. Springer, 2003.
5. Hans L Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1996.
6. Hans L Bodlaender. Treewidth: characterizations, applications, and computations. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 1–14. Springer, 2006.
7. Hans L Bodlaender and Arie MCA Koster. Safe separators for treewidth. *Discrete Mathematics*, 306(3):337–350, 2006.
8. Hans L Bodlaender and Arie MCA Koster. Treewidth computations i. upper bounds. *Information and Computation*, 208(3):259–275, 2010.
9. Hans L Bodlaender and Arie MCA Koster. Treewidth computations ii. lower bounds. *Information and Computation*, 209(7):1103–1119, 2011.
10. Vincent Bouchitté and Ioan Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM Journal on Computing*, 31(1):212–232, 2001.

11 Holgar Dell. PACE-challenge/Treewidth. `https://github.com/PACE-challenge/Treewidth`, 2017. [github repository, accessed January 12, 2022].

12 Holgar Dell. Treewidth-PACE-2017-bonus-instances. `https://github.com/PACE-challenge/Treewidth-PACE-2017-bonus-instances/`, 2017. [github repository, accessed January 12, 2022].

13 Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration. In *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

14 Pinar Heggernes. Minimal triangulations of graphs: A survey. *Discrete Mathematics*, 306(3):297–317, 2006.

15 Neil Robertson and Paul D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

16 Neil Robertson and Paul D Seymour. Graph minors. xiii. the disjoint paths problem. *Journal of combinatorial theory, Series B*, 63(1):65–110, 1995.

17 Neil Robertson and Paul D Seymour. Graph minors. xx. wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2):325–357, 2004.

18 Hisao Tamaki. Computing treewidth via exact and heuristic lists of minimal separators. In *International Symposium on Experimental Algorithms*, pages 219–236. Springer, 2019.

19 Hisao Tamaki. A heuristic use of dynamic programming to upperbound treewidth. *arXiv preprint*, 2019. `arXiv:1909.07647`.

20 Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. *Journal of Combinatorial Optimization*, 37(4):1283–1311, 2019.

21 Hisao Tamaki. A heuristic for listing almost-clique minimal separators of a graph. *arXiv preprint*, 2021. `arXiv:2108.07551`.

22 Hisao Tamaki. twalgor/tw. `https://github.com/twalgor/`, 2022. [github repository].