A Contraction-Recursive Algorithm for Treewidth

Hisao Tamaki
Meiji University, Kawasaki, Japan

Abstract

Let $\text{tw}(G)$ denote the treewidth of graph G. Given a graph G and a positive integer k such that $\text{tw}(G) \leq k + 1$, we are to decide if $\text{tw}(G) \leq k$. We give a certifying algorithm RTW ("R" for recursive) for this task: it returns one or more tree-decompositions of G of width $\leq k$ if the answer is YES and a minimal contraction H of G such that $\text{tw}(H) > k$ otherwise. Starting from a greedy upper bound on $\text{tw}(G)$ and repeatedly improving the upper bound by this algorithm, we obtain $\text{tw}(G)$ with certificates.

RTW uses a heuristic variant of Tamaki's PID algorithm for treewidth (ESA2017), which we call HPID. Informally speaking, PID builds potential subtrees of tree-decompositions of width $\leq k$ in a bottom up manner, until such a tree-decomposition is constructed or the set of potential subtrees is exhausted without success. HPID uses the same method of generating a new subtree from existing ones but with a different generation order which is not intended for exhaustion but for quick generation of a full tree-decomposition when possible. RTW, given G and k, interleaves the execution of HPID with recursive calls on G/e for edges e of G, where G/e denotes the graph obtained from G by contracting edge e. If we find that $\text{tw}(G/e) > k$, then we have $\text{tw}(G) > k$ with the same certificate. If we find that $\text{tw}(G/e) \leq k$, we "uncontract" the bags of the certifying tree-decompositions of G/e into bags of G and feed them to HPID to help progress. If the question is not resolved after the recursive calls are made for all edges, we finish HPID in an exhaustive mode. If it turns out that $\text{tw}(G) > k$, then G is a certificate for $\text{tw}(G') > k$ for every G' of which G is a contraction, because we have found $\text{tw}(G/e) \leq k$ for every edge e of G. This final round of HPID guarantees the correctness of the algorithm, while its practical efficiency derives from our methods of "uncontracting" bags of tree-decompositions of G/e to useful bags of G, as well as of exploiting those bags in HPID.

Experiments show that our algorithm drastically extends the scope of practically solvable instances. In particular, when applied to the 100 instances in the PACE 2017 bonus set, the number of instances solved by our implementation on a typical laptop, with the timeout of 100, 1000, and 10000 seconds per instance, are 72, 92, and 98 respectively, while these numbers are 11, 38, and 68 for Tamaki's PID solver and 65, 82, and 85 for his new solver (SEA 2022).

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases graph algorithm, treewidth, exact computation, BT dynamic programming, contraction, certifying algorithms

Digital Object Identifier 10.4230/LIPIcs.IPEC.2023.34

Related Version *Full Version*: `https://arxiv.org/abs/2307.01318`

Supplementary Material *Software (Source Code)*: `https://github.com/twalgor/RTW`

1 Introduction

Treewidth is a graph parameter introduced and extensively studied in the graph minor theory [14]. A *tree-decomposition* of graph G is a tree with each node labeled by a vertex set of G, called a *bag*, satisfying certain conditions (see Section 2) so that those bags form a tree-structured system of vertex-separators of G. The width $w(T)$ of a tree-decomposition T is the maximum cardinality of a bag in T minus one and the treewidth $\text{tw}(G)$ of graph G is the smallest k such that there is a tree-decomposition of G of width k.

The impact of the notion of treewidth on the design of combinatorial algorithms is profound: there are a huge number of NP-hard graph problems that are known to be tractable when parameterized by treewidth: they admit an algorithm with running time $f(k)n^{O(1)}$, where n is the number of vertices, k is the treewidth of the given graph, and f is some typically exponential function (see [10], for example). Those algorithms typically perform dynamic programming based on the system of separators provided by the tree-decomposition. To make such algorithms practically useful, we need to compute the treewidth, or a good approximation of the treewidth, together with an associated tree-decomposition.

Computing the treewidth $\mathrm{tw}(G)$ of a given graph G is NP-complete [2], but is fixed-parameter tractable [14, 4]. In particular, the algorithm due to Bodlaender [4] runs in time linear in the graph size with a factor of $2^{O(\mathrm{tw}(G)^3)}$. Unfortunately, this algorithm does not seem to run efficiently in practice.

In more practical approaches to treewidth computation, triangulations of graphs play an important role. A *triangulation* of graph G is a chordal graph H with $V(G) = V(H)$ and $E(G) \subseteq E(H)$. For every tree-decomposition T of G, filling every bag of T into a clique gives a triangulation of G. Conversely, for every triangulation H of G, there is a tree-decomposition of G in which every bag is a maximal clique of H. Through this characterization of tree-decompositions in terms of triangulations, we can enumerate all relevant tree-decompositions by going through the total orderings on the vertex set, as each total ordering defines a triangulation for which the ordering is a perfect elimination order (see [7], for example). Practical algorithms in the early stage of treewidth research performed a branch-and-bound search over these total orderings [7]. Dynamic programming on this search space results in a $2^n n^{O(1)}$ time algorithms [5], which works well in practice for graphs with a small number of vertices. It should also be noted that classical upper bound algorithms, such as min-deg or min-fill, which heuristically choose a single vertex ordering defining a tree-decomposition, are fast and often give a good approximation of the treewidth in a practical sense [7].

Another important link between chordal graphs and treewidth computation was established by Bouchitté and Todinca [9]. They introduced the notion of potential maximal cliques (PMCs, see below in ”Our approach” paragraph for a definition) and gave an efficient dynamic programming algorithm working on PMCs (BT dynamic programming) to find a minimal triangulation of the given graph that corresponds to an optimal tree-decomposition. They showed that their algorithm runs in polynomial time for many special classes of graphs. BT dynamic programming is also used in an exponential time algorithm for treewidth that runs in time $O(1.7549^n)$ [12].

BT dynamic programming had been considered mostly of theoretical interest until 2017, when Tamaki presented its positive-instance driven (PID) variant, which runs fast in practice and significantly outperforms previously implemented treewidth algorithms [18]. Further efforts on treewidth computation based on or around his approach have been made since then, with some incremental successes [17, 16, 19, 1].

In his most recent work [20], Tamaki introduced another approach to treewidth computation, based on the use of contractions to compute tight lower bounds on the treewidth. For edge e of graph G, the *contraction* of G by e, denoted by G/e, is a graph obtained from G by replacing e by a new single vertex v_e and let v_e be adjacent to all neighbors of the ends of e in $V(G) \setminus e$. A graph H is a *contraction* of G if H is obtained from G by zero or more successive contractions by edges. It is well-known and easy to see that $\mathrm{tw}(H) \leq \mathrm{tw}(G)$ for every contraction H of G. This fact has been used to quickly compute reasonably good lower bounds on the treewidth of a graph, typically to be used in branch-and-bound algorithms mentioned above [7, 8]. Tamaki [20] gave a heuristic method of successively improving contraction based lower bounds which, together with a separate heuristic method for upper bounds, quite often succeeds in computing the exact treewidth of instances that are hard to solve for previously published solvers.

Our approach

Our approach is based on the observation that contractions are useful not only for computing lower bounds but also for computing upper bounds. Suppose we have a tree-decomposition T of G/e of width k for some edge $e = \{u, v\}$ of k. Let v_e be the vertex to which e contracts. Replacing each bag X of T by X', where $X' = X \setminus \{v_e\} \cup \{u, v\}$ if $v_e \in X$ and $X' = X$ otherwise, we obtain a tree-decomposition T' of G of width $\leq k + 1$, which we call the *uncontraction* of T. In a fortunate case where every bag X of T with $v_e \in X$ has $|X| \leq k$, the width of T' is k. To increase the chance of having such fortunate cases, we deal with a set of tree-decompositions rather than a single tree-decomposition. We represent such a set of tree-decompositions by a set of potential maximal cliques as follows.

A vertex set of G is a *potential maximal clique* (PMC for short) if it is a maximal clique of some minimal triangulation of G. Let $\Pi(G)$ denote the set of all PMCs of G. For each $\Pi \subseteq \Pi(G)$, let $\mathscr{T}_\Pi(G)$ denote the set of all tree-decompositions of G whose bags all belong to Π. Let $\mathrm{tw}_\Pi(G)$ denote the smallest k such that there is a tree-decomposition in $\mathscr{T}_\Pi(G)$ of width k; we set $\mathrm{tw}_\Pi(G) = \infty$ if $\mathscr{T}_\Pi(G) = \emptyset$. Bouchitté and Todinca [9] showed that $\mathscr{T}_{\Pi(G)}(G)$ contains a tree-decomposition of width $\mathrm{tw}(G)$ and developed a dynamic programming algorithm (BT dynamic programming) to find such a tree-decomposition. Indeed, as Tamaki [17] noted, BT dynamic programming can be used for arbitrary $\Pi \subseteq \Pi(G)$ to compute $\mathrm{tw}_\Pi(G)$ in time linear in $|\Pi|$ and polynomial in $|V(G)|$.

A set of PMCs is a particularly effective representation of a set of tree-decompositions for our purposes, because BT dynamic programming can be used to work on $\Pi \subseteq \Pi(G)$ and find a tree-decomposition in $\mathscr{T}_\Pi(G)$ that minimizes a variety of width measures based on bag weights. In our situation, suppose we have $\Pi \subseteq \Pi(G/e)$ such that $\mathrm{tw}_\Pi(G/e) = k$. Using appropriate bag weights, we can use BT dynamic programming to decide if $\mathscr{T}_\Pi(G/e)$ contains T such that the uncontraction T' of T has width k and find one if it exists.

These observations suggest a recursive algorithm for improving an upper bound on treewidth. Given a graph G and k such that $\mathrm{tw}(G) \leq k+1$, the task is to decide if $\mathrm{tw}(G) \leq k$. Our algorithm certifies the YES answer by $\Pi \subseteq \Pi(G)$ with $\mathrm{tw}_\Pi(G) \leq k$. It uses heuristic methods to find such Π and, when this goal is hard to achieve, recursively solves the question if $\mathrm{tw}(G/e) \leq k$ for edge e of G. Unless $\mathrm{tw}(G/e) = k + 1$ and hence $\mathrm{tw}(G) = k + 1$, the recursive call returns $\Pi \subseteq \Pi(G/e)$ such that $\mathrm{tw}_\Pi(G/e) \leq k$. We use the method mentioned above to look for $T \in \mathscr{T}_\Pi(G/e)$ whose uncontraction has width $\leq k$. If we are successful, we are done for G. Even when this is not the case, the uncontractions of tree-decompositions in $\mathscr{T}_\Pi(G/e)$ may be useful for our heuristic upper bound method in the following manner.

In [17], Tamaki proposed a local search algorithm for treewidth in which a solution is a set of PMCs rather than an individual tree-decomposition and introduced several methods of expanding $\Pi \subseteq \Pi(G)$ into $\Pi' \supset \Pi$ in hope of having $\mathrm{tw}_{\Pi'}(G) < \mathrm{tw}_\Pi(G)$. His method compares favourably with existing heuristic algorithms but, like typical local search methods, is prone to local optima. To let the search escape from a local optimum, we would like to inject "good" PMCs to the current set Π. It appears that tree-decompositions in $\mathscr{T}_{\Pi'}(G/e)$ such that $\mathrm{tw}_{\Pi'}(G/e) \leq k$, where $k = \mathrm{tw}_\Pi(G) - 1$, are reasonable sources of such good PMCs: we uncontract $T \in \mathscr{T}_{\Pi'}(G/e)$ into a tree-decomposition T' of G and extract PMCs of G from T'. Each such PMC appears in a tree-decomposition of width $\leq k + 1$ and may appear in a tree-decomposition of width $\leq k$. It is also important that Π' is obtained, in a loose sense, independently of Π and not under the influence of the local optimum around which Π stays.

Our algorithm for deciding if $\mathrm{tw}(G) \leq k$ interleaves the execution of a local search algorithm with recursive calls on G/e for edges e of G and injects PMCs obtained from the results of the recursive calls. This process ends in either of the following three ways.

1. The local search succeeds in finding Π with $\mathrm{tw}_\Pi(G) \leq k$.
2. A recursive call on G/e finds that $\mathrm{tw}(G/e) = k+1$: we conclude that $\mathrm{tw}(G) = k+1$ on the spot.
3. Recursive calls G/e have been tried for all edges e and it is still unknown if $\mathrm{tw}(G) \leq k$. We invoke a conventional exact algorithm for treewidth to settle the question.

Note that, when the algorithm concludes that $\mathrm{tw}(G) = k+1$, there must be a contraction H of G somewhere down in the recursion path from G such that Case 3 applies and the exact computation shows that $\mathrm{tw}(H) = k+1$. In this case, H is a minimal contraction of G that certifies $\mathrm{tw}(G) = k+1$, as the recursive calls further down from H have shown $\mathrm{tw}(H/e) \leq k$ for every edge e of H.

As the experiments in Section 11 show, this approach drastically extends the scope of instances for which the exact treewidth can be computed in practice.

Organization

The rest of this paper is organized as follows. After the preliminaries in Section 2, the main algorithm in its basic form is described in Section 3. Sections 4, 6, 7, 8, 9, and 10 describe some details of the techniques used to make the algorithm run fast in practice. Section 11 presents experimental results and Section 12 offers some concluding remarks.

The source code of the implementation of our algorithm used in the experiments is available at `https://github.com/twalgor/RTW`.

2 Preliminaries

Graphs and treewidth

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: $N_G(U) = (\bigcup_{v \in U} N_G(v)) \setminus U$.

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. We denote by $\mathcal{C}(G)$ the set of all components of G. When the graph G is clear from the context, we denote $\mathcal{C}(G[U])$ by $\mathcal{C}(U)$. 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.

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.

For each pair (i, j) of adjacent nodes of a tree-decomposition $(T, \mathcal{X})$ of G, let $T(i, j)$ denote the subtree of T consisting of nodes of T reachable from i without passing j and let $V(i, j) = \bigcup_{k \in V(T(i,j))} X_k$. Then, it is well-known and straightforward to show that $X_i \cap X_j = V(i, j) \cap V(j, i)$ and there are no edges between $V(i, j) \setminus V(j, i)$ and $V(j, i) \setminus V(i, j)$; $X_i \cap X_j$ is a separator of G unless $V(i, j) \subseteq V(j, i)$ or $V(j, i) \subseteq V(j, i)$. We say that T *uses* separator S if there is an adjacent pair (i, j) such that $S = X_i \cap X_j$. In this paper, we assume G is connected whenever we consider a tree-decomposition of G.

In this paper, most tree-decompositions are such that $X_i = X_j$ only if $i = j$. Because of this, we use a convention to view a tree-decomposition of G as a tree T whose nodes are bags (vertex sets) of G.

Triangulations, 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 $E(H)$. It is known (see [13] for example) that if H is a minimal triangulation of G then every minimal separator of H is a minimal separator of G. In fact, the set of minimal separators of H is a maximal set of pairwise non-crossing minimal separators of G, where two separators S and R *cross each other* if at least two components of $G \setminus S$ intersects R.

Triangulations and tree-decompositions are closely related. For a tree-decomposition T of G, let $\mathrm{fill}(G, T)$ denote the graph obtained from G by filling every bag of T into a clique. Then, it is straightforward to see that $\mathrm{fill}(G, T)$ is a triangulation of G. Conversely, for each chordal graph H, consider a tree on the set $\mathcal{K}$ of all maximal cliques of H such that if $X, Y \in \mathcal{K}$ are adjacent to each other then $X \cap Y$ is a minimal separator of H. Such a tree is called a *clique tree* of H. It is straightforward to verify that a clique tree T of a triangulation H of G is a tree-decomposition of G and that $\mathrm{fill}(G, T) = H$.

We call a tree-decomposition T of G *minimal* if it is a clique tree of a minimal triangulation of G. It is clear that there is a minimal tree-decomposition of G of width $\mathrm{tw}(G)$, since for every tree-decomposition T of G, there is a minimal triangulation H of G that is a subgraph of $\mathrm{fill}(G, T)$ and every clique tree T' of H has $w(T') \leq w(T)$.

A vertex set $X \subseteq V(G)$ is a *potential maximal clique*, PMC for short, 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. By definition, every bag of a minimal tree-decomposition of G belongs to $\Pi(G)$.

Bouchitté-Todinca dynamic programming

For each $\Pi \subseteq \Pi(G)$, say that Π *admits* a tree-decomposition T of G if every bag of T belongs to Π. Let $\mathscr{T}_\Pi(G)$ denote the set of all tree-decompositions of G that Π admits and let $\mathrm{tw}_\Pi(G)$ denote the smallest k such that there is $T \in \mathscr{T}_\Pi(G)$ of width k; we set $\mathrm{tw}_\Pi(G) = \infty$ if $\mathscr{T}_\Pi(G) = \emptyset$. The treewidth algorithm of Bouchitté and Todinca [9] is based on the observation that $\mathrm{tw}(G) = \mathrm{tw}_{\Pi(G)}(G)$. Given G, their algorithm first constructs $\Pi(G)$ and then search through $\mathscr{T}_{\Pi(G)}(G)$ by dynamic programming (BT dynamic programming) to find T of width $\mathrm{tw}_{\Pi(G)}(G)$. As observed in [17], BT dynamic programming can be used to compute $\mathrm{tw}_\Pi(G)$ for an arbitrary subset Π of $\Pi(G)$ to produce an upper bound on $\mathrm{tw}(G)$. As we extensively use this idea, we describe how it works here.

Fix $\Pi \subseteq \Pi(G)$ such that $\mathscr{T}_\Pi(G)$ is non-empty. To formulate the recurrences in BT dynamic programming, we need some definitions. A vertex set B of G is a *block* if B is connected and either $N_G(B)$ is a minimal separator or is empty. As we are assuming that G is connected, $B = V(G)$ in the latter case. A *partial tree-decomposition* of a block B in G is a tree-decomposition of $G[B \cup N_G(B)]$ that has a bag containing $N_G(B)$, called the *root bag* of this partial tree-decomposition. Note that a partial tree-decomposition of block $V(G)$ is a tree-decomposition of G. For a graph G and a block B, let $\mathcal{P}_\Pi(B, G)$ denote the set of all partial tree-decompositions of B in G all of whose bags belong to Π and, when this set is non-empty, let $\mathrm{tw}_\Pi(B, G)$ denote the smallest k such that there is $T \in \mathcal{P}_\Pi(B, G)$ with $w(T) = k$; if $\mathcal{P}_\Pi(B, G)$ is empty we set $\mathrm{tw}_\Pi(B, G) = \infty$.

A PMC X of G is a *cap* of block B if $N_G(B) \subseteq X$ and $X \subseteq B \cup N_G(B)$. Note that a cap of B is a potential root bag of a partial tree-decomposition of B. For each block B, let $\mathcal{B}_\Pi(B)$ denote the set of all caps of B belonging to Π. Recall that, for each vertex set $U \subseteq V(G)$, $\mathcal{C}(U)$ denotes the set of components of $G[U]$. The following recurrence holds.

$$\mathrm{tw}_\Pi(B, G) = \min_{X \in \mathcal{B}_\Pi(B)} \max\{|X| - 1, \max_{C \in \mathcal{C}(B \setminus X)} \mathrm{tw}_\Pi(C, G)\} \tag{1}$$

BT dynamic programming evaluates this recurrence for blocks in the increasing order of cardinality and obtains $\mathrm{tw}_\Pi(G) = \mathrm{tw}_\Pi(V(G), G)$. Tracing back the recurrences, we obtain a tree-decomposition $T \in \mathscr{T}_\Pi(G)$ with $w(T) = \mathrm{tw}_\Pi(G)$.

Tamaki's PID algorithm [18], unlike the original algorithm of Bouchitté and Todinca [9], does not construct $\Pi(G)$ before applying dynamic programming. It rather uses the above recurrence to generate relevant blocks and PMCs. More precisely, PID is for the decision problem whether $\mathrm{tw}(G) \le k$ for given G and k and it generates all blocks C with $\mathrm{tw}(C, G) \le k$ using the recurrence in a bottom up manner. We have $\mathrm{tw}(G) \le k$ if and only if $V(G)$ is among those generated blocks.

Contractors and contractions

To extend the notation G/e of a contraction by an edge to a contraction by multiple edges, we define contractors. A *contractor* γ of G is a partition of $V(G)$ into connected sets. For contractor γ of G, the contraction of G by γ, denoted by G/γ, is the graph obtained from G by contracting each part of γ to a single vertex, with the adjacency inherited from G. For notational convenience, we also view a contractor γ as a mapping from $V(G)$ to $\{1, 2, \ldots, m\}$, the index set of the parts of the partition γ. In this view, the vertex set of G/γ is $\{1, 2, \ldots, m\}$ and $\gamma(v)$ for each $v \in V(G)$ is the vertex of G/γ into which v is contracted. For each $w \in V(G/\gamma)$, $\gamma^{-1}(w)$ is the part of the partition γ that contracts to w. For $U \subseteq V(G/\gamma)$, we define $\gamma^{-1}(U) = \bigcup_{w \in U} \gamma^{-1}(w)$.

3 Main algorithm

The pseudo code in Algorithm 1 shows the main iteration of our treewidth algorithm. It starts from a greedy upper bound and repeatedly improves the upper bound by algorithm RTW. The call $RTW(G, k, \Pi)$, where $\Pi \subseteq \Pi(G)$ and $\mathrm{tw}_{\Pi}(G) \leq k+1$, decides if $\mathrm{tw}(G) \leq k$. If $\mathrm{tw}(G) \leq k$, it returns YES with certificate $\Pi' \subseteq \Pi(G)$ such that $\mathrm{tw}_{\Pi'}(G) \leq k$; otherwise it returns NO with certificate H, a minimal contraction of G such that $\mathrm{tw}(H) = k+1$.

Algorithm 1 Main iteration for computing $\mathrm{tw}(G)$.

Ensure: compute $\mathrm{tw}(G)$ for given G

$T \leftarrow$ a minimal tree-decomposition of G obtained by a greedy algorithm
$\Pi \leftarrow$ the set of bags of T
$k \leftarrow w(T)$
while true **do**
  call $RTW(G, k-1, \Pi)$
  if the call returns NO with certificate H **then**
    stop: $\mathrm{tw}(G)$ equals k with $\mathrm{tw}(G) \leq k$ certified by Π and $\mathrm{tw}(G) \geq k$ certified by H
  else
    $k \leftarrow k-1$
    $\Pi \leftarrow$ the certificate of the YES answer
  end if
end while

The pseudo code in Algorithm 2 describes RTW in its basic form. We sketch here the functions of subalgorithms used in this algorithm. More details can be found in subsequent sections.

Our method of local search in the space of sets of PMCs is a heuristic variant, which we call HPID, of the PID algorithm due to Tamaki [18]. PID constructs partial tree-decompositions of width $\leq k$ using the recurrence of BT dynamic programming in a bottom up manner to exhaustively generate all partial tree-decompositions of width $\leq k$, so that we have a tree-decomposition of width $\leq k$ if and only if $\mathrm{tw}(G) \leq k$. HPID uses the same recurrence to generate partial tree-decompositions of width $\leq k$ but the aim is to quickly generate a tree-decomposition of G of width $\leq k$ and the generation order it employs does not guarantee exhaustive generation. The state of HPID computation is characterized by the set Π of root bags of the generated partial tree-decompositions. Recall that the bags of the set of partial tree-decompositions generated by the BT recurrence are PMCs, so $\Pi \subseteq \Pi(G)$. Using BT dynamic programming, we can reconstruct the set of partial tree-decompositions from Π, if needed, in time linear in $|\Pi|$ and polynomial in $|V(G)|$. Thus, we may view HPID as performing a local search in the space of sets of PMCs. This view facilitates communications between HPID and external upper bound heuristics. Those communications are done through the following operations.

We consider each invocation of HPID as an entity having a state. Let s denote such an invocation instance of HPID for G and k. Let $\Pi(s)$ denote the set of PMCs that are root bag of the partial tree-decompositions generated so far by s. The following operations are available.

s.**width()** returns $\mathrm{tw}_{\Pi(s)}(G)$.

s.**usefulPMCs()** returns the set of PMCs that are the root bags of the partial tree-decompositions of width $\leq s.width()$ generated so far by s.

Algorithm 2 Procedure $RTW(G, k, \Pi)$.

Require: $\Pi \subseteq \Pi(G)$ and $\mathrm{tw}_\Pi(G) \leq k+1$

Ensure: returns YES with $\Pi \subseteq \Pi(G)$ such that $\mathrm{tw}_\Pi(G) \leq k$ if $\mathrm{tw}(G) \leq k$; NO with a minimal contraction H of G such that $\mathrm{tw}(H) = k+1$ otherwise

create an HPID instance s for G and k
s.IMPORTPMCs(Π)
if s.width() $\leq k$ **then**
    return YES with s.USEFULPMCs()
end if
order the edges of G appropriately as e_1, e_2, $\ldots e_m$.
for $i = 1, \ldots, m$ **do**
    $\Theta \leftarrow$ CONTRACTPMCs(s.USEFULPMCs(), G, e_i)
    call $RTW(G/e_i, k, \Theta)$
    if the call returns NO with certificate H **then**
        return NO with certificate H
    else
        $\Psi \leftarrow$ the certificate for the YES answer
        $\Psi' \leftarrow$ UNCONTRACTPMCs(Ψ, G, e)
        s.IMPORTPMCs(Ψ')
        s.IMPROVE($UNIT\_BUDGET \times i$)
        if s.width() $\leq k$ **then**
            return YES with s.USEFULPMCs()
        end if
    end if
end for
s.FINISH()
if s.width() $\leq k$ **then**
    return YES with s.USEFULPMCs()
else
    return NO with certificate G
end if

s.importPMCs(Π) updates $\Pi(s)$ to $\Pi(s) \cup \Pi$ and updates the set of partial tree-decompositions by BT dynamic programming.

s.improve(budget) generates more partial tree-decompositions under the specified budget, in terms of the number of search steps spent for the generation.

s.finish() exhaustively generates remaining partial decompositions of width $\leq k$, thereby deciding if $\mathrm{tw}(G) \leq k$.

See Section 4 for some details of these procedures.

We use two additional procedures.

uncontractPMCs(Π, G, e) , where e is an edge of G and $\Pi \subseteq \Pi(G/e)$, returns $\Pi' \subseteq \Pi(G)$ such that $\mathrm{tw}_{\Pi'}(G) \leq \mathrm{tw}_\Pi(G/e) + 1$ and possibly $\mathrm{tw}_{\Pi'}(G) \leq \mathrm{tw}_\Pi(G/e)$

contractPMCs(Π, G, e) , where e is an edge of G and $\Pi \subseteq \Pi(G)$, returns $\Pi' \subseteq \Pi(G/e)$ such that $\mathrm{tw}_{\Pi'}(G/e) \leq \mathrm{tw}_\Pi(G)$ and possibly $\mathrm{tw}_{\Pi'}(G/e) \leq \mathrm{tw}_\Pi(G) - 1$

See Sections 6 and 7 for details of these procedures.

The correctness of this algorithm can be proved by straightforward induction and does not depend on the procedures EXPAND, CONTRACTPMCs, or UNCONTRACTPMCs except that the procedure CONTRACTPMCs(Π, G, e) must return Θ such that $\mathrm{tw}_\Theta(G/e) \leq \mathrm{tw}_\Pi(G)$

as promised. On the other hand, practical efficiency of this algorithm heavily depends on the performances of these procedures. If they collectively work really well, then we expect that the **for** loop would exit after trying only a few edges, assuming $\mathrm{tw}(G) \leq k$, and s.FINISH() would be called only if $\mathrm{tw}(G) = k+1$ and $\mathrm{tw}(G/e) \leq k$ for every edge e. On the other extreme of perfect incapability of these procedures, the **for** loop would always run to the end and s.FINISH() would be called in every call of $RTW(G, k, \Pi)$, making the recursion totally meaningless. Our efforts are devoted to developing effective methods for these procedures.

4 Heuristic PID

In this section, we give some details of the HPID algorithm. In particular, we describe in some details how the procedures IMPROVE(budget) and FINISH() work.

We first describe how we use Recurrence (1) to generate a new partial tree-decomposition from existing ones. The method basically follows that of PID [18] but differs in the way we view tree-decompositions as rooted-decompositions. The differences are motivated by the need of HPID to interact with external upper bound components through PMCs.

Fix G and k. We assume a total order $<$ on $V(G)$ and say that $U \subseteq V(G)$ is *larger* then $V \subseteq V(G)$ if $|U| > |V|$ or $|U| = |V|$ and U is lexicographically larger than V. We say a block B is *small* if there is some block B' with $N_G(B') = N_G(B)$ such that $B' > B$. We say that a block B of G is *feasible* if $\mathrm{tw}(B, G) \leq k$. We use Recurrence (1), with Π set to $\Pi(G)$, to generate feasible blocks that are small.

Each HPID instance s maintains a set set $\mathcal{F}$ of small feasible blocks. To generate a new feasible block to add to $\mathcal{F}$, it invokes a backtrack search procedure SEARCHNEWFEASIBLE(B) on a block $B \in \mathcal{F}$ which enumerates $\mathcal{B} \subseteq \mathcal{F}$ such that

1. $B \in \mathcal{B}$ and B is the largest block in $\mathcal{B}$ and
2. there is a block $B_{\mathcal{B}}$ that is either small or is equal to $V(G)$ and a PMC $X_{\mathcal{B}} \in \Pi(G)$ such that $\mathcal{C}(B_{\mathcal{B}} \setminus X_{\mathcal{B}}) = \mathcal{B}$.

For each such $\mathcal{B}$ found, we add $B_{\mathcal{B}}$ to $\mathcal{F}$ since the Recurrence (1) shows that $B_{\mathcal{B}}$ is feasible.

Procedure s.IMPROVE(budget) uses this search procedure as follows. It uses a priority queue Q of small feasible blocks, in which larger blocks are given higher priority. It first put all blocks in $\mathcal{F}$ to Q. Then, it dequeues a block B, call SEARCHNEWFEASIBLE(B), and add newly generated feasible blocks to Q. This is repeated until either Q is empty or the cumulative number of search steps exceeds budget. Because of the queuing policy, there is a possibility of $V(G)$ found feasible, when it is indeed feasible, even with a small budget.

The role of procedure s.FINISH() is to complete the PID computation by exhaustively generating partial tree-decompositions. The implementation used in our experiment uses another variant of PID called SemiPID [16] for this task.

5 Minimalizing tree-decompositions

Given a graph G and a triangulation H of G, *minimalizing* H means finding a minimal triangulation H' of G such that $E(H') \subseteq E(H)$. Minimalizing a tree-decomposition T of G means finding a minimal tree-decomposition T' of G whose bags are maximal cliques of the minimalization of $\mathrm{fill}(G, T)$. We want to minimalize a tree-decomposition for two reasons. One is our decision to represent a set of tree-decompositions by a set of PMCs. Whenever we get a tree-decomposition T by some method that may produce non-minimal tree-decompositions, we minimalize it to make all bags PMCs. Another reason is that minimalization may reduce the width. We have two procedures for minimalization. When the second reason is of no concern, we use MINIMALIZE(T) which is an implementation of one of the standard triangulation minimalization algorithm due to Blair *et al* [3]. When the second reason is important, we use MINIMALIZEOPTIMALLY(T), which finds a minimalization of T of the smallest width. This task is NP-hard, but the following algorithm works well in practice.

Say a minimal separator of G is *admissible for* T if it is a clique of $\mathrm{fill}(G, T)$. Observe that, for every minimalization T' of T, every separator used by T' is a minimal separator of G admissible for T. We first construct the set of all minimal separators of G admissible for T. Then we apply the SemiPID variant of BT dynamic programming, due to Tamaki [16], to this set and obtain a tree-decomposition of the smallest width, among those using only admissible minimal separators. Because of the admissibility constraint, the number of minimal separators is much smaller and both the enumeration part and the SemiPID part run much faster in practices than in the general case without such constraints.

6 Uncontracting PMCs

In this section, we develop an algorithm for procedure UNCONTRACTPMCS(G, Π, e). In fact, we generalize this procedure to UNCONTRACTPMCS(G, Π, γ), where the third argument is a general contractor of G.

Given a graph G, $\Pi \subseteq \Pi(G)$, and a contractor γ of G, we first find tree-decompositions $T \in \mathscr{T}_\Pi$ that minimize $w(\gamma^{-1}(T))$. This is done by BT dynamic programming over $\mathscr{T}_\Pi(G/\gamma)$, using bag weights defined as follows. For each weight function ω that assigns weight $\omega(U)$ to each vertex set U, define the width of tree-decomposition T with respect to ω, denoted by $\mathrm{tw}(G, \omega)$, to be the maximum of $\omega(X)$ over all bags of T. Thus, if ω is defined by $\omega(U) = |U| - 1$ then $\mathrm{tw}(G, \omega) = \mathrm{tw}(G)$. A natural choice for our purposes is to set $\omega(X) = |\gamma^{-1}(X)| - 1$. Then, the width of a tree decomposition T of G/γ with respect to this bag weight is $w(\gamma^{-1}(T))$. Therefore, BT dynamic programming with this weight function ω gives us the desired tree-decomposition in $\mathscr{T}_\Pi(G/r)$.

We actually use a slightly modified weight function, considering the possibility of reducing the weight of $\gamma^{-1}(T)$ by minimalization.

Let $T \in \mathscr{T}_\Pi(G/\gamma)$ and X a bag of T. If $X' = \gamma^{-1}(X)$ is a PMC of G, then every minimalization of $\gamma^{-1}(T)$ must contain X' as a bag. Therefore, if $|X'| > k + 1$ then it is impossible that the width of $\gamma^{-1}(T)$ is reduced to k by minimalization. On the other hand, if X' is not a PMC, then no minimalization of $\gamma^{-1}(T)$ has X' has a bag and there is a possibility that there is a minimalization of $\gamma^{-1}(T)$ of width k even if $|X'| > k + 1$. These considerations lead to the following definition of our weight function ω.

$$\omega(U) = 2|\gamma^{-1}(U)| \quad \text{if } \gamma^{-1}(U) \text{ is a PMC of } G \tag{2}$$

$$\omega(U) = 2|\gamma^{-1}(U)| - 1 \quad \text{otherwise} \tag{3}$$

Algorithm 3 describes the main steps of procedure UNCONTRACTPMCS(Π, G, γ).

7 Contracting PMCs

The algorithm for procedure CONTRACTPMCS is similar to that for UNCONTRACTPMCS. Given a graph G, $\Pi \subseteq \Pi(G)$, and a contractor γ of G, we first find tree-decompositions $T \in \mathscr{T}_\Pi(G/\gamma)$ that minimize $w(\gamma(T))$. This is done by BT dynamic programming with the following weight function ω.

$$\omega(U) = 2|\gamma(U)| \quad \text{if } \gamma(U) \text{ is a PMC of } G/\gamma$$

$$\omega(U) = 2|\gamma(U)| - 1 \quad \text{otherwise}$$

Algorithm 3 Procedure UNCONTRACTPMCS(Π, G, γ).

Require: $\Pi \subseteq \Pi(G/\gamma)$
Ensure: returns $\Pi' \subseteq \Pi(G)$ that results from uncontracting Π and then minimalizing

```
let ω be the weight function on 2^{V(G/γ)} defined by equations 2 and 3
use BT dynamic programming to obtain tree-decompositions T_i, 1 ≤ i ≤ m, of G/γ such
    that w(T_i, ω) = tw_Π(G, ω)
for each i, 1 ≤ i ≤ m do
    T'_i ← MINIMALIZEOPTIMALLY(γ^{-1}(T_i))
    Π_i ← the set of bags of T'_i
end for
return ⋃_i Π_i
```

Then, we minimalize those tree-decompositions and collect the bags of those minimalized tree-decompositions.

8 Safe separators

Bodlaender and Koster [6] introduced the notion of safe separators for treewidth. Let S be a separator of a graph G. We say that S is *safe for treewidth*, or simply safe, if $\mathrm{tw}(G) = \mathrm{tw}(G \cup K(S))$. As every tree-decomposition of $G \cup K(S)$ must have a bag containing S, $\mathrm{tw}(G)$ is the larger of $|S| - 1$ and $\max\{\mathrm{tw}(G[C \cup N_G(C)] \cup K(N_G(C))\}$, where C ranges over all the components of $G \setminus S$. Thus, the task of computing $\mathrm{tw}(G)$ reduces to the task of computing $\mathrm{tw}(G[C \cup N_G(C)] \cup K(N_G(C))\}$ for every component C of $G \setminus S$. The motivation for looking at safe separators of a graph is that there are sufficient conditions for a separator being safe and those sufficient conditions lead to an effective preprocessing method for treewidth computation. We use the following two sufficient conditions.

A vertex set S of G is an *almost-clique* if $S \setminus \{v\}$ is a clique for some $v \in S$. Let R be a vertex set of G. A contractor γ of G is *rooted on* R if, for each part C of γ, $|C \cap R| = 1$.

▶ **Theorem 1** (Bodlaender and Koster [6]).

1. *If S is an almost-clique minimal separator of G, then S is safe.*
2. *Let lb be a lower bound on $\mathrm{tw}(G)$. Let $C \subseteq V(G)$ be connected and let $S = N_G(C)$. Suppose (1) $\mathrm{tw}(G[C \cup S] \cup K(S)) \leq lb$ and (2) $G[C \cup S]$ has a contractor γ rooted on S such that $G[C \cup S]/\gamma$ is a complete graph. Then, S is safe.*

We use safe separators both for preprocessing and during recursion. For preprocessing, we follow the approach of [19]: to preprocess G, we fix a minimal triangulation H of G and test the sufficient conditions in the theorem for each minimal separator of H. Since deciding if the second condition holds is NP-complete, we use a heuristic procedure. Let $\mathcal{S}$ be the set of all minimal separators of H that are confirmed to satisfy the first or the second condition of the theorem. Let $\mathcal{A}$ be a tree-decomposition of G that uses all separators of $\mathcal{S}$ but no other separators. Then, $\mathcal{A}$ is what is called a *safe-separator decomposition* in [6]. A tree-decomposition of G of width $\mathrm{tw}(G)$ can be obtained from $\mathcal{A}$ by replacing each bag X of $\mathcal{A}$ by a tree-decomposition of $G[X] \cup \bigcup_{C \in \mathcal{C}(G \setminus X)} K(N_G(C))$, the graph obtained from the subgraph of G induced by X by filling the neighborhood of every component of $G \setminus X$ into a clique.

Safe separators are also useful during the recursive computation. Given G, we wish to find a contractor γ of G such that $\mathrm{tw}(G/\gamma) = \mathrm{tw}(G)$, so that we can safely recurse on G/γ. The second sufficient condition in Theorem 1 is useful for this purpose. Let C, S, and γ be as in the condition. We construct γ' such that $\mathrm{tw}(G/\gamma') = \mathrm{tw}(G)$ as follows. The proof of this sufficient condition is based on the fact that we get a clique on S when we apply the contractor γ on $G[C \cup S]$. Thus, we may define a contractor γ' on G such that $G/\gamma' = (G \setminus C) \cup K(S)$. As each tree-decomposition of $\mathrm{tw}(G/\gamma)$ can be extended to a tree-decomposition of G, using the tree-decomposition of $G[C \cup S] \cup K(S)$ of width at most $lb \leq \mathrm{tw}(G)$, we have $\mathrm{tw}(G/\gamma') = \mathrm{tw}(G)$ as desired. When the recursive call on $\mathrm{tw}(G/\gamma')$ returns a certificate $\Pi \subseteq \Pi(G/\gamma')$ such that $\mathrm{tw}_\Pi(G/\gamma') \leq k$, we need to "uncontract" Π into a $\Pi' \subseteq \Pi(G)$ such that $\mathrm{tw}_{\Pi'}(G) \leq k$. Fortunately, this can be done without invoking the general uncontraction procedure. Observe first that each PMC in Π naturally corresponds to a PMC of $(G \setminus C) \cup K(S)$, which in turn corresponds to a PMC of G contained in $V(G) \setminus C$. Let Π_1 be the set of those PMCs of G to which a PMC in Π corresponds in that manner. Let $\Pi_2 \subseteq \Pi(G[C \cup S] \cup K(S))$ be such that $\mathrm{tw}_{\Pi_2}(G[C \cup S] \cup K(S)) \leq lb$. Similarly as above, each PMC of Π_2 corresponds to a PMC of G contained $C \cup S$. Let Π_2' denote the set of those PMCs of G to which a PMC in Π_2 corresponds. As argued above, a tree-decomposition in $\mathscr{T}_\Pi((G \setminus C) \cup K(S))$ of $(G \setminus C) \cup K(S)$ and a tree-decomposition in $\mathscr{T}_{\Pi_2}(G[C \cup S] \cup K(S))$ of $G[C \cup S] \cup K(S)$ can be combined into a tree-decomposition belonging to $\mathscr{T}_{\Pi_2'}(G)$ of width $\leq k$. Thus, Π_2' is a desired certificate for $\mathrm{tw}(G) \leq k$.

9 Edge ordering

We want an edge e such that $\mathrm{tw}(G/e) = \mathrm{tw}(G)$, if such exists, to appear early in our edge order. Heuristic criteria for such an ordering have been studied in the classic work on contraction based lower bounds [8]. Our criterion is similar to those but differs in that it derives from a special case of safe separators. The following is a simple corollary of Theorem 1.

▶ **Proposition 2.** *Let $e = \{u, v\}$ be an edge of G and let $S = N_G(v)$. Suppose $S \setminus \{u\}$ is a clique of G. Then, we have $\mathrm{tw}(G/e) = \mathrm{tw}(G)$.*

If e satisfies the above condition, then we certainly put e first in the order. Otherwise, we evaluate e in terms of its closeness to this ideal situation. Define the *deficiency* of graph H, denoted by $\mathrm{defic}(H)$, to be the number of edges of its complement graph. For each ordered pair (u, v) of adjacent vertices of G, let $\mathrm{defic}_G(u, v)$ denote $\mathrm{defic}(G[N_G(v) \cup \{v\}]/\{u, v\})$. Note that $\mathrm{defic}_G(u, v) = 0$ means that the condition of the above proposition is satisfied with $S = N_G(v)$. Thus, we regard $e = \{u, v\}$ preferable if either $\mathrm{defic}_G(u, v)$ or $\mathrm{defic}_G(v, u)$ is small. We relativize the smallness with respect to the neighborhood size, so the *value* of edge $e = \{u, v\}$ is $\min\{defic_G(u, v)/|N_G(v)|, defic_G(v, u)/|N_G(u)|\}$. We order edges so that this value is non-decreasing.

10 Suppressed edges

Consider the recursive call on G/e from the call of RTW on G, where e is an edge of G. Suppose there is an ancestor call on G' such that $G = G'/\gamma$ and edge e' of G' such that γ maps the ends of e' to the ends of e. If the call on G'/e' has been made and it is known that $\mathrm{tw}(G'/e') \leq k$ then we know that $\mathrm{tw}(G/e) \leq k$, since G/e is a contraction of G'/e'. In this situation, we say that e is *suppressed* by the pair (G', e'). We may omit the recursive call on G/e without compromising the correctness if e is suppressed. For efficiency, however, it is preferable to obtain the certificate $\Pi \subseteq \Pi(G/e)$ for $\mathrm{tw}(G/e) \leq k$ and feed the uncontraction of Π to the HPID instance on G to help progress. Fortunately, this can be done without making the recursive call on G as follows. Suppose e is suppressed by (G', e') and let $\Pi' \subseteq \Pi(G'/e')$ such that $\mathrm{tw}_{\Pi'}(G'/e') \leq k$. Let γ' be the contractor of G'/e' such that $G'/e'/\gamma' = G/\gamma/e$: such γ' is straightforward to obtain from γ. Letting $\Pi = \textsc{contractPMCs}(\Pi, G'/e', \gamma')$, we obtain $\Pi \subseteq \Pi(G/e)$ such that $\mathrm{tw}_{\Pi}(G/e) \leq k$.

11 Experiments

We have implemented RTW and evaluated it 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 implementation uses a single thread except for additional threads that may be invoked for garbage collection by JVM.

Our primary benchmark is the bonus instance set of the exact treewidth track of PACE 2017 algorithm implementation challenge [11]. This set, consisting of 100 instances, is intended to be a challenge for future implementations and, as a set, are hard for the winning solvers of the competition. Using the platform of the competition, about half of the instances took more than one hour to solve and 15 instances took more than a day or were not solvable at all.

We have run our implementation on these instances with the timeout of 10000 seconds each. For comparison, we have run Tamaki's PID solver [18], which is one of the PACE 2017 winners, available at [15] and his new solver [20] available at [21]. Figure 1 summarizes the results on the bonus set. In contrast to PID solver which solves only 68 instances within the timeout, RTW solves 98 instances. Moreover, it solve 72 of them in 100 seconds and 92 of them in 1000 seconds. Thus, we can say that our algorithm drastically extends the scope of practically solvable instances. Tamaki's new solver also quickly solves many instances that are hard for PID solver and is indeed faster then RTW on many instances. However, its performance in terms of the number of instances solvable in practical time is inferior to RTW.

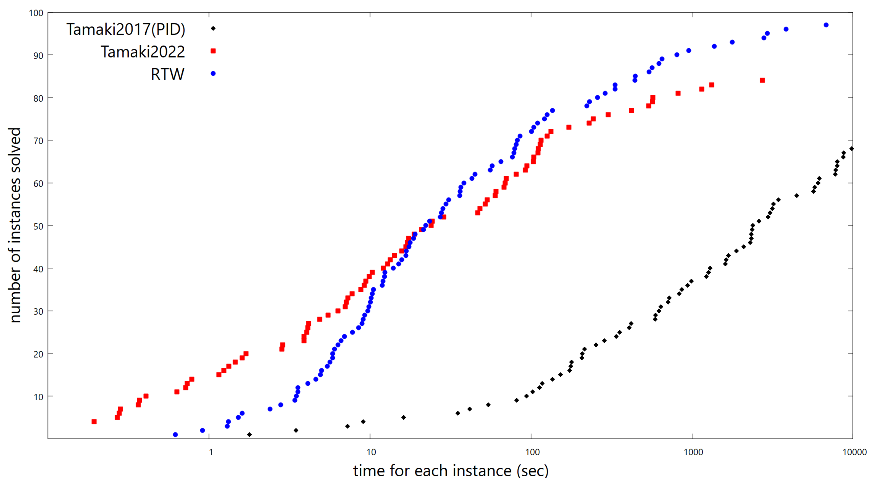

Figure 1 Number of bonus instances solved within a specified time.

We have also run the solvers on the competition set of the exact treewidth track of PACE 2017. This set, consisting of 200 instances, is relatively easy and the two winning solvers of the competition solved all of the instances within the allocated timeout of 30 minutes for each instance. Figure 2 summarizes the results on the competition set. Somewhat expectedly, PID performs the best on this instance set. It solves almost all instances in 200 seconds for each instance, while RTW fails to do so on about 30 instances. There are two instances that RTW fails to solve in 10000 seconds and one instance it fails to solve at all. Tamaki's new solver shows more weakness on this set, failing to solve about 50 instances in the timeout of 10000 seconds.

These results seem to suggest that RTW and PID should probably complement each other in a practical treewidth solver.

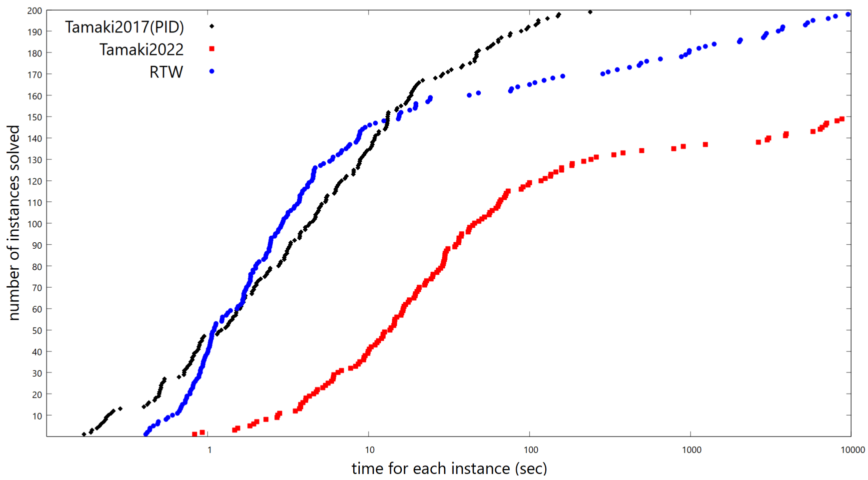

Figure 2 Number of competition instances solved within a specified time.

12 Conclusions and future work

We developed a treewidth algorithm RTW that works recursively on contractions. Experiments show that our implementation solves many instances in practical time that are hard to solve for previously published solvers. RTW, however, does not perform well on some instances that are easy for conventional solvers such as PID. A quick compromise would be to run PID first with an affordable timeout and use RTW only when it fails. It would be, however, interesting and potentially fruitful to closely examine those instances that are easy for PID and hard for RTW and, based on such observations, to look for a unified algorithm that avoids the present weakness of RTW.

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 Jean RS Blair, Pinar Heggernes, and Jan Arne Telle. A practical algorithm for making filled graphs minimal. *Theoretical Computer Science*, 250(1-2):125–141, 2001.

4 Hans L Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1996.

5 Hans L Bodlaender, Fedor V Fomin, Arie MCA Koster, Dieter Kratsch, and Dimitrios M Thilikos. On exact algorithms for treewidth. In *Algorithms–ESA 2006: 14th Annual European*

Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings, pages 672–683. Springer, 2006.

6 Hans L Bodlaender and Arie MCA Koster. Safe separators for treewidth. *Discrete Mathematics*, 306(3):337–350, 2006.

7 Hans L Bodlaender and Arie MCA Koster. Treewidth computations i. upper bounds. *Information and Computation*, 208(3):259–275, 2010.

8 Hans L Bodlaender and Arie MCA Koster. Treewidth computations ii. lower bounds. *Information and Computation*, 209(7):1103–1119, 2011.

9 Vincent Bouchitté and Ioan Todinca. Treewidth and minimum fill-in: Grouping the minimal separators. *SIAM Journal on Computing*, 31(1):212–232, 2001.

10 Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*. Springer, 2015.

11 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.

12 Fedor V Fomin and Yngve Villanger. Treewidth computation and extremal combinatorics. *Combinatorica*, 32(3):289–308, 2012.

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

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

15 Hisao Tamaki. PID. `https://github.com/TCS-Meiji/PACE2017-TrackA`, 2017. [github repository].

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

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

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

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

20 Hisao Tamaki. Heuristic Computation of Exact Treewidth. In Christian Schulz and Bora Uçar, editors, *20th International Symposium on Experimental Algorithms (SEA 2022)*, volume 233 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:16, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.SEA.2022.17`.

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