Positive-Instance Driven Dynamic Programming for Treewidth*

Hisao Tamaki

Department of Computer Science, Meiji University, Kawasaki, Japan
tamaki@cs.meiji.ac.jp

Abstract

Consider a dynamic programming scheme for a decision problem in which all subproblems involved are also decision problems. An implementation of such a scheme is *positive-instance driven* (PID), if it generates positive subproblem instances, but not negative ones, building each on smaller positive instances.

We take the dynamic programming scheme due to Bouchitté and Todinca for treewidth computation, which is based on minimal separators and potential maximal cliques, and design a variant (for the decision version of the problem) with a natural PID implementation. The resulting algorithm performs extremely well: it solves a number of standard benchmark instances for which the optimal solutions have not previously been known. Incorporating a new heuristic algorithm for detecting safe separators, it also solves all of the 100 public instances posed by the exact treewidth track in PACE 2017, a competition on algorithm implementation.

We describe the algorithm and prove its correctness. We also perform an experimental analysis counting combinatorial structures involved, which gives insights into the advantage of our approach over more conventional approaches and points to the future direction of theoretical and engineering research on treewidth computation.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases treewidth, dynamic programming, minimal separators, potential maximal cliques, positive instances

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.68

1 Introduction

Suppose we design a dynamic programming algorithm for some decision problem, formulating subproblems, which are decision problems as well, and recurrences among those subproblems. A standard approach is to list all subproblem instances from “small" ones to “large" and scan the list, deciding the answer, positive or negative, to each instance by means of these recurrences. When the number of positive subproblem instances are expected to be much smaller than the total number of subproblem instances, a natural alternative is to generate positive instances only, using recurrences to combine positive instance to generate a “larger" positive instance. We call such a mode of dynamic programming execution *positive-instance driven* or *PID* for short. One goal of this paper is to demonstrate that PID is not simply a low-level implementation strategy but can be a paradigm of algorithm design for some problems.

The decision problem we consider is that of deciding, given graph G and positive integer k, if the treewidth of G is at most k. This graph parameter was introduced by Robertson and

* Full paper available as an arXiv preprint, https://arxiv.org/abs/1704.05286.

Seymour [17] and has had a tremendous impact on graph theory and on the design of graph algorithms (see, for example, a survey [7].) The treewidth problem is NP-complete [1] but fixed-parameter tractable: it has an $f(k)n^{O(1)}$ time algorithm for some fixed function $f(k)$ as implied by the graph minor theorem of Robertson and Seymour [18], and explicit $O(f(k)n)$ time algorithm is given by Bodlaender [3]. A classical dynamic programming algorithm due to Arnborg, Corneil, and Proskurowsky (ACP algorithm) [1] runs in $n^{k+O(1)}$ time. Bouchitté and Todinca [9] developed a more refined dynamic programming algorithm (BT algorithm) based on the notions of minimal separators and potential maximal cliques, which lead to algorithms running in $O(1.7549^n)$ time or in $O(n^5\binom{\lceil (2n+k+8)/3 \rceil}{k+2})$ time [11, 12].

Another important approach to treewidth computation is based on the perfect elimination order (PEO) of a minimal chordal completion of the given graph. PEO-based dynamic programming algorithms run in $O^*(2^n)$ time with exponential space and in $O^*(4^n)$ time with polynomial space [5], where $O^*(f(n))$ means $O(n^c f(n))$ for some constant c.

There has been a considerable amount of effort on implementing treewidth algorithms to be used in practice and, prior to this work, the most successful implementations for exact treewidth computation are all based on PEO. The authors of [5] implemented the $O^*(2^n)$ time dynamic programming algorithm and experimented on its performance, showing that it works well for small instances. For larger instances, PEO-based branch-and-bound algorithms are known to work well in practice [14]. Recent proposals for reducing treewidth computation to SAT solving are also based on PEO [19, 2].

From the PID perspective, this situation is somewhat surprising, for the following reasons. Let us first review the PEO approach. See [5], for example, for details. Let G be the input graph. Recall that a PEO of G is a total order $v_1, \ldots, v_n$ on $V(G)$ such that, for $1 \leq i \leq n$, v_i is simplicial in $G[V_i]$, where $V_i = \{v_i, \ldots, v_n\}$ and a vertex is *simplicial* in a graph if its neighbors form a clique. A graph is *chordal* if it has no induced cycle of length four or greater. A *chordal completion* of G is a chordal supergraph of G with vertex set $V(G)$. The above PEO-based algorithms utilizes two facts: that every chordal graph has a PEO and that, for chordal graphs, the optimal tree-decomposition consists of all maximal cliques as bags. Thus, these algorithms look for a total order of $V(G)$ that is a PEO of a chordal completion of G whose optimal tree-decomposition is an optimal tree-decomposition of G. The dynamic programming algorithm reduces the search space size from the naive $O(n!)$ to $O(2^n)$ applying the Held-Karp paradigm for sequencing problems [4]. In the decision problem version, it consists in defining the "feasibility" of each subset of $V(G)$, to be inductively decided by dynamic programming. Informally, $S \subseteq V(G)$ is *feasible* if it has a total ordering that qualifies as a prefix of a total ordering of $V(G)$ that gives a chordal completion with the clique number k or smaller. This feasibility notion, however, has a more direct interpretation in terms of tree-decompositions: S is feasible if each connected component of $G[S]$ is feasible and each connected vertex set C is feasible if $G[C \cup N(C)]$, where $N(C)$ is the open neighborhood of S, has a tree-decomposition of width k or smaller that has a bag containing $N(C)$. This feasibility of connected sets is nothing but the feasibility considered in the classical ACP algorithm. Thus, each positive subproblem instance in the PEO-based dynamic programming scheme corresponds to a combination of an indefinite number of positive subproblem instances in the ACP algorithm, and hence the number of positive subproblem instances can be exponentially larger than that in the ACP algorithm. Indeed, a PID variant of the ACP algorithm was implemented by the present author and has won the first place in the exact treewidth track of PACE 2016 [10], a competition on algorithm implementations, outperforming other submissions based on PEO. Given this success, a natural next step is to design a PID variant of the BT algorithm, which is tackled in this paper.

The resulting algorithm performs extremely well, as reported in Section 7. It is tested on DIMACS graph-coloring instances [15], which have been used in the literature as standard benchmark instances [14, 8, 16, 19, 5, 2]. Our implementation of the algorithm solves all the instances that have been previously solved (that is, with matching upper and lower bounds known) within 10 seconds per instance on a typical desktop computer and solves 13 out of the 42 previously unsolved instances. For nearly half of the instances which it leaves unsolved, it significantly reduces the gap between the lower and upper bounds. It is interesting to note that this is done by improving the lower bound. Since the number of positive subproblem instances are much smaller when k is below the treewidth than when k equals the treewidth, the PID approach is particularly good at establishing strong lower bounds.

We also adopt the notion of safe separators due to Bodlaender and Koster [6] in our preprocessing and design a new heuristic algorithm for detecting safe separators. With this preprocessing, our implementation also solves all of the 100 public instances posed by PACE 2017 [21], the successor of PACE 2016. It should be noted that these test instances of PACE 2017 are much harder than those of PACE 2016: the winning implementation of PACE 2016 mentioned above, which solved 199 of the 200 instances therein, solves only 62 of these 100 instances of PACE 2017 in the given time of 30 minutes per instance.

Adapting the BT algorithm to work in PID mode has turned out non-trivial. It requires concepts and observations not present in [9]. We describe these concepts and observations, formulate our variant in full details, and prove its correctness.

We also perform an experimental analysis in which we count combinatorial structures involved in both PID and non-PID approaches, namely minimal separators, potential maximal cliques, and related objects. The analysis reveals that the practical bottleneck of the original BT algorithm lies in listing potential maximal cliques. Let $\mathcal{P}_k(G)$ denote the set of all potential maximal cliques of cardinality of $k+1$ or smaller of graph G. Although there are theoretical upper bounds of $O(1.7549^n)$ and $n^{O(1)}\binom{\lceil(2n+k+8)/3\rceil}{k+2}$ on the time to compute $\mathcal{P}_k(G)$ [12], where n is the number of vertices, huge gaps between these bounds and $|\mathcal{P}_k(G)|$ are observed in the experiments. This motivates the need of output sensitive algorithms that run fast when $|\mathcal{P}_k(G)|$ is small. Our PID algorithm is a first step in this direction. Although it does not compute $|\mathcal{P}_k(G)|$ in an output sensitive manner, it does compute the set of positive subproblem instances, whose size is empirically comparable to $|\mathcal{P}_k(G)|$, in an output sensitive manner.

Due to the space limitation, we omit proofs of lemmas and theorems, all of which can be found in the full paper. Our implementation in source code is available at our GitHub repository [13].

2 Preliminaries

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. For each $v \in V(G)$, $N_G(v)$ denote 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 neighbor set 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 neighbor set of* U *in* G, denoted by $N_G[U]$, is defined by $N_G[U] = U \cup N_G(U)$. We also write $N_G[v]$ for $N_G[\{v\}] = N_G(v) \cup \{v\}$. We denote by $G[U]$ the subgraph of G induced by U: $V(G[U]) = U$ and $E(G[U]) = \{\{u, v\} \in E(G) \mid u, v \in U\}$. In the above notation, as well as in the notation further introduced below, we will often drop the subscript G when the graph is clear from the context.

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* of G if it is connected and is inclusion-wise maximal subject to this condition. A vertex set C in G is a *component associated with* $S \subseteq G$, if C is a connected component of $G[V(G) \setminus S]$. For each $S \subseteq V(G)$, we denote by $\mathcal{C}_G(S)$ (or $\mathcal{C}(S)$ when G is clear from the context) the set of all components associated with S. A vertex set $S \subseteq V(G)$ is a *separator* of G if $|\mathcal{C}_G(S)| \geq 2$. A component C is a *full component* associated with separator S if $N(C) = S$. A separator S is a *minimal separator* if there are at least two full components associated with S. This term is justified by this fact: if S is a minimal separator and a, b vertices belonging to two distinct full components associated with S, then for every proper subset S' of S, a and b belong to the same component associated with S'; S is a minimal set of vertices that separates a from b.

Graph H is *chordal* if every induced cycle of H has length exactly three. H is a *minimal chordal completion of* G if it is chordal, $V(H) = V(G)$, $E(G) \subseteq E(H)$, and $E(H)$ is minimal subject to these conditions. A vertex set $\Omega \subseteq V(G)$ is a potential maximal clique of G, if Ω is a clique in some minimal chordal completion of G.

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 such that the following three conditions are satisfied. We call members of $V(T)$ *nodes* of T and 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 $I_v = \{i \in V(T) \mid v \in X_i\}$ of $V(T)$ induces a connected subtree of 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 minimum width of all tree-decompositions of G. We may assume that the bags X_i and X_j are distinct from each other for $i \neq j$ and, under this assumption, we will often regard a tree-decomposition as a tree T in which each node is a bag.

We call a tree-decomposition T of G *canonical* if each bag of T is a potential maximal clique of G and, for every pair X, Y of adjacent bags in T, $X \cap Y$ is a minimal separator of G. The following fact is well-known. It easily follows, for example, from Proposition 2.4 in [9].

▶ **Lemma 1.** *Let G be an arbitrary graph. There is a tree-decomposition T of G of width $\mathrm{tw}(G)$ that is canonical.*

The following local characterization of a potential maximal clique is crucial. We say that a vertex set $S \subseteq V(G)$ is *cliquish* in G if, for every pair of distinct vertices u and v in S, either u and v are adjacent to each other or there is some $C \in \mathcal{C}(S)$ such that $u, v \in N(C)$. In other words, S is cliquish if completing $N(C)$ for every $C \in \mathcal{C}(S)$ into a clique makes S a clique.

▶ **Lemma 2** (Theorem 3.15 in [9]). *A separator S of G is a potential maximal clique of G if and only if* (1) *S has no full-component associated with it and* (2) *S is cliquish.*

It is also shown in [9] that if Ω is a potential maximal clique of G and S is a minimal separator contained in Ω, then there is a unique component C_S associated with S that contains $\Omega \setminus S$. We need an explicit way of forming C_S from Ω and S.

Let $K \subseteq V(G)$ be an arbitrary vertex set and S an arbitrary proper subset of K. We say that a component $C \in \mathcal{C}(K)$ is *confined to* S if $N(C) \subseteq S$; otherwise it is *unconfined to* S. Let $\mathrm{unconf}(S, K)$ denote the set of components associated with K that are unconfined to S. Define $\mathrm{crib}(S, K) = (K \setminus S) \cup \bigcup_{C \in \mathrm{unconf}(S,K)} C$. The following lemma relies only on

the second property of potential maximal cliques, namely that they are cliquish, and will be applied not only to potential maximal cliques but also to separators with full components, which are trivially cliquish.

▶ **Lemma 3.** *Let $K \subseteq V(G)$ be a cliquish vertex set. Let S be an arbitrary proper subset of K. Then,* $\mathrm{crib}(S, K)$ *is a full component associated with S.*

▶ Remark. As $\mathrm{crib}(S, K)$ contains $K \setminus S$, it is clearly the only component associated with S that intersects K. Therefore, the above mentioned assertion on potential maximal cliques is a corollary of this Lemma.

3 Recurrences on oriented minimal separators

In this section, we fix graph G and positive integer k that are given in the problem instance: we are to decide if the treewidth of G is at most k.

For connected set $C \subseteq V(G)$, we denote by $G\langle C\rangle$ the graph obtained from $G[N[C]]$ by completing $N(C)$ into a clique: $V(G\langle C\rangle) = N[C]$ and $E(G\langle C\rangle) = E(G[N[C]]) \cup \{\{u, v\} \mid u, v \in N(C), u \neq v\}$. We say C is *feasible* if $\mathrm{tw}(G\langle C\rangle) \leq k$. Equivalently, C is feasible if $G[N[C]]$ has a tree-decomposition of width k or smaller that has a bag containing $N(C)$.

Let us first review the BT algorithm [9] adapting it to our decision problem. We first list all minimum separators of cardinality k or smaller and all potential maximal cliques of cardinality $k + 1$ or smaller. Then, for each pair of a potential maximal clique Ω and a minimal separator S such that $S \subset \Omega$, place a link from S to Ω. To understand the difficulty of formulating a PID variant of the algorithm, it is important to note that the pair (Ω, S) to be linked is easy to find from the side of Ω, but not the other way round. Then, we scan the full blocks $(N(C), C)$ of minimal separators in the increasing order of $|C|$ to decide if C is feasible, using the following recurrence: C is feasible if and only if there is some potential maximal clique Ω such that $N(C) \subset \Omega$, $C = \mathrm{crib}(N(C), \Omega)$, and every component $D \in \mathrm{unconf}(N(C), \Omega)$ is feasible. Finally, we have $\mathrm{tw}(G) \leq k$ if and only if there is a potential maximal clique Ω with $|\Omega| \leq k + 1$ such that every component associated with Ω is feasible.

To facilitate the PID construction, we orient minimal separators as follows. We assume a total order $<$ on $V(G)$. For each vertex set $U \subseteq V(G)$, the *minimum element* of U, denoted by $\min(U)$, is the smallest element of U under $<$. For vertex sets U and W, we say U *precedes* W and write $U \prec W$ if $\min(U) < \min(W)$.

We say that a connected set C is *inbound* if there is some full block associated with $N(C)$ that precedes C; otherwise, it is *outbound*. Observe that if C is inbound then $N(C)$ is a minimal separator, since $N(C)$ has another full component associated with it. Contrapositively, if $N(C)$ is not a minimal separator then C is necessarily outbound. We say a full block $(N(C), C)$ is *inbound* (*outbound*) if C is inbound (outbound, respectively).

▶ **Lemma 4.** *Let K be a cliquish vertex set and let A_1, A_2 be two components associated with K. Suppose that A_1 and A_2 are outbound. Then, either $N(A_1) \subseteq N(A_2)$ or $N(A_2) \subseteq N(A_1)$.*

Let K be a cliquish vertex set. Based on the above lemma, we define the *outlet* of K, denoted by $\mathrm{outlet}(K)$, as follows. If no non-full component associated with K is outbound, then we let $\mathrm{outlet}(K) = \emptyset$. Otherwise, $\mathrm{outlet}(K) = N(A)$, where A is a non-full component associated with K that is outbound, chosen so that $N(A)$ is maximal. We define $\mathrm{support}(K) = \mathrm{unconf}(\mathrm{outlet}(K), K)$, the set of components associated with K that are not confined to $\mathrm{outlet}(K)$. By Lemma 4, every member of $\mathrm{support}(K)$ is inbound.

We call a full block $(N(C), C)$ an *I-block* if C is inbound and $|N(C)| \leq k$. We call it an *O-block* if C is outbound and $|N(C)| \leq k$.

We say that an I-block $(N(C), C)$ is *feasible* if C is feasible. We say that an O-block $(N(A), A)$ is feasible if $N(A) = \bigcup_{C \in \mathcal{C}} N(C)$ for some set $\mathcal{C}$ of feasible inbound components. Note that this definition of feasibility of an O-block is somewhat weak in the sense that we do not require every inbound component associated with $N(A)$ to be feasible.

Let Ω be a potential maximal clique with $|\Omega| \leq k+1$. For each $C \in \text{support}(\Omega)$, block $(N(C), C)$ is an I-block, since C is inbound as observed above and we have $|N(C)| \leq k$ by our assumption that $|\Omega| \leq k+1$. We say that Ω is *feasible* if $|\Omega| \leq k+1$ and either

1. $\Omega = N[v]$ for some $v \in V(G)$,
2. there is some subset $\mathcal{C}$ of $\text{support}(\Omega)$ such that $\Omega = \bigcup_{D \in \mathcal{C}} N(D)$ and every member of $\mathcal{C}$ is feasible, or
3. $\Omega = N(A) \cup (N(v) \cap A)$ for some feasible O-block $(N(A), A)$ and a vertex $v \in N(A)$.

We say that Ω is *strongly feasible* if $|\Omega| \leq k+1$ and every $C \in \text{support}(\Omega)$ is feasible. It will turn out that every strongly feasible potential maximal clique is feasible (Lemma 9). This implication, however, is not immediate from the definitions.

▶ **Lemma 5.** *We have* $\text{tw}(G) \leq k$ *if and only if* G *has a strongly feasible potential maximal clique* Ω *with* $\text{outlet}(\Omega) = \emptyset$.

▶ **Lemma 6.** *Let* C *be a connected set of* G *such that* $N(C)$ *is a minimal separator. Let* Ω *be a potential maximal clique of* $G\langle C \rangle$*. Then,* Ω *is a potential maximal clique of* G.

The following is our oriented version of the recurrence in the BT algorithm described in the beginning of this section.

▶ **Lemma 7.** *An I-block* $(N(C), C)$ *is feasible if and only if there is some strongly feasible potential maximal clique* Ω *with* $\text{outlet}(\Omega) = N(C)$ *and* $\bigcup_{D \in \text{support}(\Omega)} D = C$.

▶ **Lemma 8.** *Let* K *be a cliquish vertex set,* $\mathcal{C}$ *a non-empty subset of* $\text{support}(K)$*, and* $S = \bigcup_{C \in \mathcal{C}} N(C)$*. If* S *is a proper subset of* K *then* $\text{crib}(S, K)$ *is outbound.*

The following lemma is crucial for our PID result: the algorithm described in the next section generates all feasible potential maximal cliques and we need to guarantee all strongly feasible maximal cliques to be among them.

▶ **Lemma 9.** *Let* Ω *be a strongly feasible potential maximal clique. Then,* Ω *is feasible.*

4 Algorithm

Given graph G and positive integer k, our algorithm generates all I-blocks, O-blocks, and potential maximal cliques that are feasible. In the following algorithm, variable $\mathcal{I}$ is used for listing feasible I-blocks, $\mathcal{O}$ for feasible O-blocks, $\mathcal{P}$ for feasible potential maximal cliques, and $\mathcal{S}$ for strongly feasible potential maximal cliques.

Algorithm PID-BT
Input Graph G and positive integer k
Output "YES" if $\text{tw}(G) \leq k$; "NO" otherwise
Procedure

1. Let $\mathcal{I}_0 = \emptyset$ and $\mathcal{O}_0 = \emptyset$.
2. Initialize $\mathcal{P}_0$ and $\mathcal{S}_0$ to $\emptyset$.
3. Set $j = 0$.

4. For each $v \in V(G)$, if $N[v]$ is a potential maximal clique with $|N[v]| \leq k+1$ then add $N[v]$ to $\mathcal{P}_0$ and if, moreover, $\mathrm{support}(N[v]) = \emptyset$ then do the following.
   a. Add $N[v]$ to $\mathcal{S}_0$.
   b. If $\mathrm{outlet}(N[v]) \neq \emptyset$ then let $C = \mathrm{crib}(\mathrm{outlet}(N[v]), N[v])$ and, provided that $C \neq C_h$ for $1 \leq h \leq j$, increment j and let $C_j = C$.
5. Set $i = 0$.
6. Repeat the following and stop repetition when j is not incremented during the iteration step.
   a. While $i < j$, do the following.
      i. Increment i and let $\mathcal{I}_i$ be $\mathcal{I}_{i-1} \cup \{C_i\}$.
      ii. Initialize $\mathcal{O}_i$ to $\mathcal{O}_{i-1}$, $\mathcal{P}_i$ to $\mathcal{P}_{i-1}$, and $\mathcal{S}_i$ to $\mathcal{S}_{i-1}$.
      iii. For each $B \in \mathcal{O}_{i-1}$ such that $C_i \subseteq B$ and $|N(C_i) \cup N(B)| \leq k+1$, let $K = N(C_i) \cup N(B)$ and do the following.
         A. If K is a potential maximal clique, then add K to $\mathcal{P}_i$.
         B. If $|K| \leq k$ and there is a full component A associated with K (which is unique), then add A to $\mathcal{O}_i$.
      iv. Let A be the full component associated with $N(C_i)$ and add A to $\mathcal{O}_i$.
      v. For each $A \in \mathcal{O}_i \setminus \mathcal{O}_{i-1}$ and $v \in N(A)$, let $K = N(A) \cup (n(v) \cap A)$ and if $|K| \leq k+1$ and K is a potential maximal clique then add K to $\mathcal{P}_i$.
      vi. For each $K \in \mathcal{P}_i \setminus \mathcal{S}_{i-1}$, if $\mathrm{support}(K) \subseteq \mathcal{I}_i$ then add K to $\mathcal{S}_i$ and do the following: if $\mathrm{outlet}(K) \neq \emptyset$ then let $C = \mathrm{crib}(\mathrm{outlet}(K), K)$ and, provided that $C \neq C_h$ for $1 \leq h \leq j$, increment j and let $C_j = C$.
7. If there is some $K \in \mathcal{S}_j$ such that $\mathrm{outlet}(K) = \emptyset$, then answer "YES"; otherwise, answer "NO".

▶ **Theorem 10.** *Algorithm PID-BT, given G and k, answers "YES" if and only if* $\mathrm{tw}(G) \leq k$.

5 Experimental analysis

To identify the practical bottleneck in the BT algorithm, we have performed some experiments. We are interested in the number of combinatorial objects involved in the treewidth computation: minimal separators, potential maximal cliques, and feasible objects used in our PID algorithm. In the case of minimal separators and potential maximal cliques, we count the total numbers of those as well as of those *relevant* in our decision problem: minimal separators with cardinality k or smaller and potential maximal cliques with cardinality $k+1$ or smaller.

Table 1 shows the results on some random instances, with k set to the treewidth of the graph: we are not interested in larger k and, for smaller k, the numbers in the columns dependent on k are smaller. The full paper contains results for more graphs with varying number of edges. The total number of minimal separators and that of potential maximal cliques grow much faster than the number of feasible objects in our algorithm, as the size of the graph grows. However, the growth in the numbers of relevant minimal separators and relevant potential maximal cliques is similar to the growth in the number of feasible objects. For example, the number of relevant potential maximal cliques grows only slightly faster than the number of feasible potential maximal cliques and is within 1.2 times the latter for the graph with 40 vertices.

Thus, scanning all relevant minimal separators and all relevant potential maximal cliques as in the original BT algorithm may not be an immediate disadvantage. The bottleneck lies rather in the time to list all relevant potential maximal cliques. Table 2 shows the number of

Table 1 The numbers of principal objects in treewidth computation.

			minimal separators		potential maximal cliques		feasible objects		
$\|V\|$	$\|E\|$	tw	all	$\leq$ tw	all	$\leq$ tw $+ 1$	I-blocks	O-blocks	PMCs
20	60	8	191	48	796	96	46	108	93
30	90	11	2983	247	20154	682	228	708	618
40	120	14	164773	2356	1740644	10372	2080	8637	8577

Table 2 The number of objects involved in generating principal objects.

			$\leq$ tw $+ 1$	vertex	feasible objects			pairs to be
$\|V\|$	$\|E\|$	tw	PMCs	representations	I-blocks	O-blocks	PMCs	examined
20	60	8	96	25263	46	108	93	206
30	90	11	682	3480559	228	708	618	1351
40	120	14	10372	167700496	2080	8637	8577	17906

additional combinatorial objects, called vertex representations, which needs to be generated in the algorithm in [12] in order to list all relevant potential maximal cliques.

The figures in the table suggests that a more output sensitive algorithm for listing relevant potential maximal cliques is desirable and that some method not relying on vertex representation is needed to achieve this goal.

In our PID approach, each feasible potential maximal clique, except in the base case, is generated from a combination of a feasible O-block and a feasible I-block. Each feasible O-block in turn is generated also from a combination of a feasible O-block and a feasible I-block. Let $\mathcal{I}$ be the set of feasible I-blocks and $\mathcal{O}$ the set of feasible O-blocks of the given graph. The crucial fact to our advantage is that most of the pairs in $\mathcal{I} \times \mathcal{O}$ are easily seen not to generate a new O-block or a potential maximal clique. The last column in Table 2 shows that the number of pairs in $\mathcal{I} \times \mathcal{O}$ that remain to be examined seriously is quite small. The data structure we called block sieves, described in the next section, is used to quickly filter out those simply rejectable pairs.

Algorithm PID-BT has a trivial output-sensitive upper bound of $n^{O(1)}|\mathcal{I}| \cdot |\mathcal{O}|$ on the time to generate necessary objects. A tighter analysis of our algorithm would be of great interest. It is also interesting to study if our approach can be applied to the problem of listing relevant potential maximal cliques.

6 Implementation

In this section, we sketch two important ingredients of our implementation. Although both are crucial in obtaining the result reported in Section 7, our work on this part is preliminary and improvements are the subject of future research.

6.1 Data structures

The crucial elementary operation in our algorithm is the following. We have a set $\mathcal{O}$ of feasible O-blocks obtained so far and, given a new feasible I-block $(N(C), C)$, need to find all members $(N(A), A)$ of $\mathcal{O}$ such that $C \subseteq A$ and $|N(C) \cup N(A)| \leq k + 1$. As the experimental analysis in the previous section shows, there is only a few such A on average for the tested instances even though $\mathcal{O}$ is usually huge. To support an efficient query processing, we introduce an abstract data structure we call block sieve.

Let G be a graph and k a positive integer. A *block sieve* for graph G and width k is a data structure storing vertex sets of $V(G)$ which supports the following operations.

store(U) : store vertex set U in in the block sieve.

supersets(U) : return the list of entries W stored in the block sieve such that $U \subseteq W$ and $|N(U) \cup N(W)| \leq k+1$.

Data structures for superset query have been studied [20]. The second condition above on the retrieved sets, however, appears to make this data structure new. For each $U \subseteq V(G)$, we define the *margin* of U to be $k+1-|N(U)|$. Our implementation of block sieves described below exploits an upper bound on the margins of vertex sets stored in the sieve.

We first describe how such block sieves with upper bounds on margins are used in our algorithm. Let $\mathcal{O}$ be the current set of O-blocks. We use t block sieves $\mathcal{B}_1, \ldots, \mathcal{B}_t$, each $\mathcal{B}_i$ having a predetermined upper bound m_i on the margins of the sets stored. We have $0 < m_1 < m_2 < \ldots < m_t = k$. We set $m_0 = 0$ for notational ease below. In our implementation, we choose roughly $t = \log_2 k$ and $m_i = 2^i$ for $0 < i < t$. For each $(N(A), A)$ in $\mathcal{O}$, A is stored in $\mathcal{B}_i$ such that the margin $k+1-|N(A)|$ is m_i or smaller but larger than m_{i-1}. When we are given an I-block $(N(C), C)$ and are to list relevant blocks in $\mathcal{O}$, we query all of the t blocks with the operations supersets(C). These queries as a whole return the list of all vertex sets A such that $(N(A), A) \in \mathcal{O}$, $C \subseteq A$, and $|(N(A) \cup N(C))| \leq k+1$.

We implement a block sieve by a trie $\mathcal{T}$. The upper bound m on margin is not used in the construction of the sieve; it is used in the query time. In the following, we assume $V(G) = \{1, \ldots, n\}$ and, by an interval $[i, j]$, $1 \leq i \leq j \leq n$, we mean the set $\{v : i \leq v \leq j\}$ of vertices. Each non-leaf node p of $\mathcal{T}$ is labelled with a non-empty interval $[s_p, f_p]$, such that $s_r = 0$ for the root r, $s_p = f_q + 1$ if p is a child of q, and $f_p = n$ if p is a parent of a leaf. Each edge (p, q) which connects node p and a child q of p, is labelled with a subset $S_{(p,q)}$ of the interval $[s_p, f_p]$. Thus, for each node p, the union of the labels of the edges along the path from the root to p is a subset of the interval $[1, s_p - 1]$, or $[1, n]$ when p is a leaf, which we denote by S_p. The choice of interval $[s_p, f_p]$ for each node p is heuristic. It is chosen so that the number of descendants of p is not too large or too small. In our implementation, the interval size is adaptively chosen from 8, 16, 32, and 64.

Each leaf q of trie $\mathcal{T}$ represents a single set stored at this leaf, namely S_q as defined above. We denote by $S(\mathcal{T})$ the set of all sets stored in $\mathcal{T}$. Then, for each node p of $\mathcal{T}$, the set of sets stored under p is $\{U \in S(\mathcal{T}) \mid U \cap [1, p] = S_p\}$.

We now describe how a query is processed against this data structure. Suppose query U is given. The goal is to visit all leaves q such that $U \subseteq S_q$ and $|N(U) \cup N(S_q)| \leq k+1$. This is done by a depth-first traversal of the trie $\mathcal{T}$. When we visit node p, we have the invariant that $U \cap [1, f_p] \subseteq S_p$, since otherwise no leaf in the subtree rooted at p stores a superset of U. Therefore, we descend from p to a child p' of p only if this invariant is maintained. Moreover, we keep track of the quantity $i(p, U) = |N(U) \cap S_p|$ in order to make further pruning of search possible. For each leaf q below p such that $U \subseteq S_q$, we have $i(q, U) \geq i(p, U)$. Combining this with eauality $|N(U) \setminus N(S_q)| = |N(U) \cap S_q| = i(q, U)$, we have $|N(U) \cup N(S_q)| \geq |N(S_q)| + i(p, U)$. Since we know an upper bound m on the margin $k+1-|N(S_q)|$ of S_q, or lower bound $k+1-m$ on $|N(S_q)|$, we may prune the search under node p if $i(p, U) > m$, since this inequality implies $|N(U) \cup N(S_q)| > k+1$ for every leaf q under p. When we reach a leaf q, we test if $|N(U) \cup N(S_q)| \leq k+1$ indeed holds.

6.2 Safe separators

The notion of safe separators for tree width was introduced by Bodlaender and Koster [6]: a separator S of G is *safe* if completing S into a clique does not change the treewidth of G. If

we find a safe separator S then the problem of deciding tree width of G reduces to that of deciding the treewidth of $G\langle C\rangle$ for each component C associated with S. Preprocessing G into such independent subproblems is highly desirable whenever possible.

The above authors observed that a powerful sufficient condition for safeness can be formulated based on graph minors. A *labelled minor* of G is a graph obtained from G by zero or more applications of the following operations. (1) Edge contraction: choose an edge $\{u, v\}$, replace u and v by a single new vertex and let all neighbors of u and v be adjacent to this new vertex; name the new vertex as either u or v. (2) Vertex deletion: delete a vertex together with all incident edges. (3) Edge deletion.

▶ **Lemma 11** (Bodlaender and Koster [6]). *A separator S of G is safe if, for every component C associated with S, $G[V(G) \setminus C]$ contains clique S as a labelled minor.*

Call a separator *minor-safe* if it satisfies the sufficient condition for safeness stated in this lemma. Bodlaender and Koster [6] showed that if S is a minimal separator and is an almost clique (deleting some single vertex makes it a clique) then S is minor-safe and moreover that the set of all almost clique minimal separators can be found in $O(n^2m)$ time, where n is the number of vertices and m is the number of edges.

We aim at capturing as many minor-safe separators as possible, at the expense of theoretical running time bounds on the algorithm for finding them. Thus, in our approach, both the algorithm for generating candidate separators and the algorithm for deciding minor-safeness are heuristic. For candidate generation, we use greedy heuristic for treewidth such as min-fill and min-degree: the separators in the resulting tree-decomposition are all candidates for safe separators.

When we apply our heuristic decision algorithm for minor-safeness to candidate separator S, one of the following occurs.

1. The algorithm answers "YES". In this case, the required labelled clique minor has been found for every component associated S and hence S is minor-safe.
2. The algorithm answers "DON'T KNOW". In this case, the algorithm has failed to find a labelled clique minor for at least one component, and hence it is not known if S is minor-safe or not.
3. The algorithm aborts, after reaching the prescribed number of execution steps.

Our heuristic decision algorithm works in two phases. Let S be a separator, C a component associated with S, and $R = V(G) \setminus (S \cup C)$. In the first phase, we contract edges in R and obtain a graph B on vertex set $S \cup R'$, where each vertex of R' is a contraction of some vertex set of R and B has no edge between vertices in R'. For each pair u, v of distinct vertices in S, let $N(u, v)$ denote the common neighbors of u and v in graph B. The contractions are performed with the goal of making $|N(u, v) \cap R'|$ large for each missing edge $\{u, v\}$ in S. In the second phase, for each missing edge $\{u, v\}$, we choose a common neighbor $w \in N(u, v) \cap R'$ and contract either $\{u, w\}$ or $\{v, w\}$. The choice of the next missing edge to be processed and the choice of the common neighbor are done as follows. Suppose the contractions in the second phase are done for some missing edges in S. For each missing edge $\{u, v\}$ not yet "processed", let $N'(u, v)$ be the set of common neighbors of u and v that are not yet contracted with any vertex in S. We choose $\{u, v\}$ with the smallest $|N'(u, v) \cap R'|$ to be processed next. Tie-breaking when necessary as well as the choice of the common neighbor w in $N'(u, v) \cap R'$ to be contracted with u or v is done in such a way that the minimum of $|(N'(x, y) \cap R') \setminus \{w\}|$ is maximized over all remaining missing edges $\{x, y\}$ in S.

The performance of these heuristics strongly depends on the instances. For PACE 2017 public instances, they work quite well. Table 3 shows the preprocessing result on the last 10

Table 3 Safe separator preprocessing on PACE 2017 instances.

name	$\|V\|$	$\|E\|$	$tw(G)$	safe separators found	max subproblem	time(secs)
ex181	109	732	18	18	89	0.078
ex183	265	471	11	173	76	0.031
ex185	237	793	14	142	52	0.046
ex187	240	453	10	138	81	0.031
ex189	178	4517	70	6	161	0.062
ex191	492	1608	15	184	132	0.171
ex193	1391	3012	10	791	119	3.17
ex195	216	382	10	114	84	0.015
ex197	303	1158	15	176	56	0.062
ex199	310	537	9	157	131	0.046

of those instances. For each instance, the number of safe separators found and the maximum subproblem size in terms of the number of vertices, after the graph is decomposed by the safe separators found, are listed. The results show that these instances, which are deemed the hardest among all the 100 public instances, are quickly decomposed into manageable subproblems by our preprocessing.

On the other hand, these heuristics have turned out useless for most of the DIMACS graph coloring instances: no safe separators are found for those instances. We suspect that this is not the limitation of the heuristics but is simply because those instances lack minor-safe separators.

7 Performance results

We used our implementation of the PID-BT algorithm to determine the treewidth of benchmark instances. For a given instance, we use our decision procedure with k being incremented one by one, starting from the obvious lower bound, namely the minimum degree of the graph. Binary search is not used because the cost of overshooting the exact treewidth is huge.

The computing environment for the experiment is as follows. CPU: Intel Core i7-7700K, 4.20GHz; RAM: 32GB; Operating system: Windows 10, 64bit; Programming language: Java 1.8; JVM: jre1.8.0_121. The maximum heap size is 6GB by default and is 24GB where it is stated so. The implementation is single threaded, except that multiple threads may be invoked for garbage collection by JVM. The time measured is the CPU time, which includes the garbage collection time.

Table 4 lists the DIMACS graph coloring instances that are newly solved: the previously known upper and lower bounds did not match. For all but three of them, the previous best upper bound has turned out optimal: only the lower bound was weaker. In this experiment, however, no knowledge of previous bounds are used and our algorithm independently determines the exact treewidth.

The results on “queen" instances illustrate how far our algorithm has extended the practical limit of exact treewidth computation. Queen7_7 with 49 vertices is the largest instance previously solved, while queen10_10 with 100 vertices is now solved.

Our implementation also solves all previously solved DIMACS graph coloring instances within 10 seconds per instance and many of them within a second. Moreover, for many of the test instances which it leaves unsolved, it significantly improves the previously best known lower bounds. The details can be found in the full paper.

Table 5 summarizes the result on PACE 2017 public instances. More details can be found in the full paper. The instance which took the longest time (530 seconds) was “ex169” which

Table 4 Newly solved DIMACS graph coloring instances.

name	$\|V\|$	$\|E\|$	tw	time(secs)	prev UB	prev LB
DSJC125.5	**125**	**3891**	**108**	**459**	**108**	56
DSJC250.9	**250**	**27897**	**243**	**0.44**	**243**	212
DSJC500.9	**500**	**112437**	**492**	**14**	**492**	433
DSJR500.5	**500**	**58862**	**246**	**546**	-	**-**
games120†	**120**	**638**	**32**	**94738**	**32**	24
homer†	**561**	**1628**	**30**	**2765**	31	26
miles750	**128**	**2113**	**36**	**0.23**	**36**	35
myciel6	**95**	**755**	**35**	**419**	**35**	29
queen8_8	**64**	**728**	**45**	**4.16**	**45**	25
queen9_9	**81**	**1056**	**58**	**274**	**58**	35
queen8_12	**96**	**1368**	**65**	**649**	**-**	39
queen10_10	**100**	**1470**	**72**	**20934**	**72**	39

Previous upper bounds from [14] and [16]; previous lower bounds from [14] and [8].
† 24GB heap space is used for these instances.

Table 5 Summary of the results on PACE 2017 public instances.

	$t \leq 1$ sec	1 sec $< t \leq 1$ min	1 min $< t \leq 10$ min
the number of instances solved in time t	25	68	7

has 3706 vertices, 42236 edges, and treewidth 22. Considering the fact that this test set has been designed to be challenging for the second competition on treewidth in PACE and that the time allocated for each instance is 30 minutes, we can say that our implementation performs quite well.

Acknowledgment. The author thanks Hiromu Ohtsuka for his help in implementing the block sieve data structure. He also thanks Yasuaki Kobayashi for helpful discussions and especially for drawing the author's attention to the notion of safe separators. This work would have been non-existent if not motivated by the timely challenges of PACE 2016 and 2017. The author is deeply indebted to their organizers, especially Holger Dell, for their dedication and excellent work.

References

1 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.

2 Jeremias Berg and Matti Järvisalo. Sat-based approaches to treewidth computation: an evaluation. In *Tools with Artificial Intelligence (ICTAI), 2014 IEEE 26th International Conference on*, pages 328–335. IEEE, 2014.

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

4 Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. A note on exact algorithms for vertex ordering problems on graphs. *Theory of Computing Systems*, 50(3):420–432, 2012.

5 Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On exact algorithms for treewidth. *ACM Transactions on Algorithms (TALG)*, 9(1):12, 2012.

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

7 Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2007.

8 Hans L. Bodlaender, Thomas Wolle, and Arie M. C. A. Koster. Contraction and treewidth lower bounds. *J. Graph Algorithms Appl.*, 10(1):5–49, 2006.

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 Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. The first parameterized algorithms and computational experiments challenge. In *LIPIcs – Leibniz International Proceedings in Informatics*, volume 63. Schloss Dagstuhl-Leibniz – Zentrum fürr Informatik, 2017.

11 Fedor V. Fomin, Dieter Kratsch, Ioan Todinca, and Yngve Villanger. Exact algorithms for treewidth and minimum fill-in. *SIAM Journal on Computing*, 38(3):1058–1079, 2008.

12 Fedor V. Fomin and Yngve Villanger. Treewidth computation and extremal combinatorics. *Combinatorica*, pages 1–20, 2012.

13 GitHub repository for TCS-Meiji on PACE2017 Track A submission. `https://github.com/TCS-Meiji/PACE2017-TrackA`. Public since: 2017-05-01.

14 Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 201–208. AUAI Press, 2004.

15 David S. Johnson and Michael A. Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

16 Nysret Musliu. An iterative heuristic algorithm for tree decomposition. In *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, pages 133–150. Springer, 2008.

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

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

19 Marko Samer and Helmut Veith. Encoding treewidth into SAT. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 45–50. Springer, 2009.

20 Iztok Savnik. Index data structure for fast subset and superset queries. In *International Conference on Availability, Reliability, and Security*, pages 134–148. Springer, 2013.

21 The Parameterized Algorithms and Computational Experiments Challenge. `https://pacechallenge.wordpress.com`. Accessed: 2017-06-30.