Jdrasil: A Modular Library for Computing Tree Decompositions

Max Bannach[1], Sebastian Berndt[2], and Thorsten Ehlers[3]

1 **Institute for Theoretical Computer Science, Universität zu Lübeck, Lübeck, Germany**
bannach@tcs.uni-luebeck.de
2 **Institute for Theoretical Computer Science, Universität zu Lübeck, Lübeck, Germany**
berndt@tcs.uni-luebeck.de
3 **Department of Computer Science, Kiel University, Kiel, Germany**
the@informatik.uni-kiel.de

Abstract

While the theoretical aspects concerning the computation of tree width – one of the most important graph parameters – are well understood, it is not clear how it can be computed *practically*. We present the open source Java library Jdrasil that implements several different state of the art algorithms for this task. By experimentally comparing these algorithms, we show that the default choices made in Jdrasil lead to an competitive implementation (it took the third place in the first PACE challenge) while also being easy to use and easy to extend.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases tree width, algorithmic library, experimental evaluation

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.28

1 Introduction

The concept of the *tree width* of a graph – the similarity of the graph to a tree – has seen an enormous amount of research in the last few years due to its theoretical and pratical importance. *Google Scholar*[1] lists more than 6.000 papers concerning this subject written in the last five years and more than 16.000 papers in total. More than half of the papers in the proceedings of the 10th International Symposium on Parameterized and Exact Computation (IPEC 2015) mention this important graph notion [25]. Tree width as a measure of the complexity of a graph has shown to be helpful in a wide range of applications ranging from the analysis of genome structure (e. g. [32]) to the learning of probablistic network from a given dataset (e. g. [27]). It has also been shown to be very useful for the theoretical investigation of the computational complexity of several graph problems, as many problems that are intractable (i. e. NP-hard) become efficiently solvable on graphs with bounded tree width. Due to this fact, tree width plays a major part in the development of fixed-parameter algorithms in the field of parameterized complexity.

A wide range of algorithms is known to compute tree decompositions, ranging from experimental heuristics over to exact exponential-time algorithms. However, they usually suffer from at least one of the following problems:

[1] https://scholar.google.com

1. The running time of the algorithm is too high even for medium-sized instances (e. g. graphs with $n \approx 100$ vertices) that arise in typical applications;
2. The value of the computed solution may be arbitrarily bad compared to the value of an optimal solution;
3. The algorithm itself may be quite complicated, which prevents an useful implementation of the algorithm.

Due to this problems, the first Parameterized Algorithms and Computational Experiments (PACE) challenge [28, 21] decided to choose the fast computation of tree decompositions as one of its tracks. The organizers wrote[2]:

> *The ambition of this track is to turn tree width, a concept that has been tremendously successful in theoretical work, into a practically useful tool. Many algorithms in parameterized complexity rely on the existence of tree decompositions of small width, and yet in practice we don't have a good understanding for how to actually compute such a decomposition. This has to change.*

This lack of practicality hinders research in theoretical and in practical investigations:

Case Example A: A researcher in bioinformatics has used very sophisticated algorithms to learn the causal graph describing the behaviour of the norovirus. This graphs consists of roughly 600 vertices. To speed up the following combinatorically complex algorithms, she would like to have an optimal tree decomposition of this graph. Unfortunately, the graph is to big to find an optimal decomposition with simple algorithms, and she tries, unsuccessfully, to find implementation of the more advanced algorithms on the internet.

Case Example B: A professor in parameterized complexity has developed a new algorithm that solves the graph 3-coloring problem using tree decompositions. He is aware of the classic algorithm by Arnborg and Proskurowski [2], but believes that his algorithm is practically more feasible. In order to evaluate his claim, he wishes to give out a bachelor thesis to a promising student, who should implement and compare both algorithms. However, the first step of these algorithms — computing a tree decomposition -– almost busts the scale of bachelor thesis, and the new algorithm ends up added to the fast growing list of never implemented algorithms.

Case Example C: A Ph. D. student has developed a new parallel algorithm to compute optimal tree decompositions, and she now wishes to implement and test this algorithm. However, before she can actually start with the "real" implementation, she has to take care of a lot of other things: data structures for graphs, and more involved ones for tree decompositions. She also has to implement all kinds of difficult graph parameters used by her new algorithm. Finally, in order to be competitive, she also has to implement all the known pre- and post-processing algorithms for tree width. Before she can actually start, a long time has passed.

1.1 Our Contributions

In this work, we aim to provide solutions for these use cases, and to broaden the understanding of the behaviour of several different approaches for the computation of tree width.

1. We provide the Java library Jdrasil that implements several different tree width algorithms (exact and heuristic). This library is designed to be both easy to use and easy to extend.

[2] `https://pacechallenge.wordpress.com/pace-2016/track-a-treewidth/`

2. We compare several different algorithms on a wide range of graphs. The results of these comparisons were used to create a competitor for the first PACE challenge, where it scored third place in the exact sequential track and third place in the heuristic parallel track.
3. We show that two quite different practical algorithmic approaches – SAT solvers and FPT algorithms – work very well together for a wide range of instances. We thus propose to stimulate the exchange of techniques between those fields.

In Section 2, we give two equivalent formulations of tree width that we will use within this work. In the next section – Section 3 – we look at different algorithmic paradigms (constraint satisfaction problems, exact exponential algorithms, heuristics, and FPT algorithms) and compare several algorithms within those paradigms. We compare those algorithms on a wide range of different graphs and combine the best of them. The combined algorithm was submitted to the first PACE challenge, where it took the third place. Experimental comparison between the winners of the PACE challenge on almost 2.000 graphs are presented in Section 4. The results show that the developed approach is competitive. The appendix contains a compact overview on the graph sets used in our experimental comparisons.

1.2 Experimental Comparisons

Our experimental comparisons were designed in such a way that a *global overview* on the behaviour of the different algorithms is given. Our plots thus show *trends* that we have observed in our computational experiments. For example, a single algorithm may always beat another algorithm, two algorithms are largely incomparable, or an algorithm either terminates within a few seconds or never. While Jdrasil contains implementation of all of the discussed algorithms, this global overviews allowed us to decide upon a standard behaviour of the library. More detailed experiments on specific algorithms and their concrete evaluations can be found in the referenced papers.

2 Preliminaries

In this paper all graphs are undirected, simple, and connected unless stated otherwise. A *tree decomposition* of a graph G is a pair (T, ι) consisting of a tree T and a mapping ι from the nodes of T to subsets of vertices of G (called *bags*), such that (1) for every edge $\{u, v\} \in E(G)$ there is a node $x \in V(T)$ with $\{u, v\} \subseteq \iota(x)$, and (2) for all nodes $x, y, z \in V(T)$ we have $\iota(y) \subseteq \iota(x) \cap \iota(z)$ whenever y lies on the unique path between x and z in T.

The *width* of a tree decomposition is the maximum size of its bags minus 1, i. e., $\text{width}(T, \iota) = \max_x\{\, |\iota(x)| - 1 \,\}$. The *tree width* of a graph G is the smallest width of any tree decomposition of G and is denoted as $\text{tw}(G)$. Deciding whether a given graph G has tree width at most k is an NP-complete problem [1].

Note that the above definition of tree width does not immediately give rise to an algorithm that computes the optimal tree decomposition. It is, hence, useful to look at alternative characterizations of this concept:

An *elimination ordering* π of a graph $G = (V, E)$ is a bijection $\pi\colon V \to \{1, 2, \ldots, |V|\}$. The *filled graph* $G_\pi = (V, E_\pi)$ of the elimination ordering π is a directed graph with edges E_π that are constructed via the following process:

- The first edge set E_π^0 simply equals E, where the edges are directed from the "lower" vertex (according to π) to the "higher" vertex, i. e.,

$$E_\pi^0 = \{\, (u, v) \mid \pi(u) < \pi(v) \wedge \{u, v\} \in E \,\}.$$

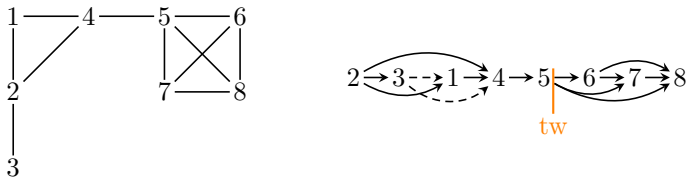

Figure 1 An example of a graph G and the corresponding filled graph G_π for the elimination ordering $\pi = 2, 3, 1, 4, 5, 6, 7, 8$. Here, the solid edges represent the edges of the original graph, while the dashed edges are the edges created by eliminating vertex 2.

- The next edge set E_π^{i+1} is generated by connecting all vertices u, v with $\pi(u) > i$ and $\pi(v) > i$ if both u and v are connected with the vertex $\pi^{-1}(i)$, i. e.,

$$E_\pi^i = E_\pi^{i-1} \cup \{\, (u, v) \mid \pi(v) > \pi(u) > i \wedge (\pi^{-1}(i), u) \in E_\pi^{i-1} \wedge (\pi^{-1}(i), v) \in E_\pi^{i-1} \,\}.$$

Finally, E_π is equal to $E_\pi^{|V|}$. Figure 1 shows an example of a graph G and the corresponding filled graph G_π.

The *width* of an elimination ordering π is the largest number of direct successors of a vertex in G_π, i. e., $\text{width}(\pi) = \max_i\{\, |\{(u_i, v) \in E_\pi\}| \,\}$. The (optimal) width of the example on the right hand side is 3, as there exist three outgoing arcs from vertex 5.

The following fact is well known and allows us to characterize the tree width of a graph either via a suitable tree decomposition or via an elimination ordering.

▶ **Fact 1** (e. g. [14]). $\text{tw}(G) = \max_\pi\{\text{width}(\pi)\}$.

3 Computing Tree Decompositions

3.1 Point of View: Constraint Satisfaction Problem

A very common (theoretical and practical) approach to solve intractable problems is to first represent them as *constraint satisfaction problems* (CSP) and then solve those problems via specialized solvers. The most widely used solvers are SAT solvers that work on Boolean formulas or ILP solvers that work on linear inequalities. As it is typically not clear from the problem alone, which of those approaches leads to a better result, the next subsection compares experimental evaluations of both approaches on a certain formulation for the elimination ordering problem. The formulation we will use is based on the work of Berg and Järvisalo [7], which in turn is an improved version of a formulation of Samer and Veith [30].

3.1.1 The CSP formulation

If $G = (V, E)$ is a graph on $|V| = n$ vertices, our CSP first contains $n(n-1)/2$ variables $\text{ord}_{i,j}$ for each $i \in \{1, \ldots, n\}$ and each $j > i$, indicating that the vertex v_i appears before v_j in the elimination ordering. To simplify notation, for two integers i and j, let $\text{ord}^*_{i,j}$ be either $\text{ord}_{i,j}$ if $i < j$ or $\neg\text{ord}_{j,i}$ if $j < i$. To ensure that these variables encode a linear ordering of the vertices, it is sufficient to enforce the transitivity: For all distinct $i, j, k \in \{1, \ldots, n\}$, we need to ensure that if $\text{ord}^*_{i,j}$ and $\text{ord}^*_{j,k}$ are true, $\text{ord}^*_{i,k}$ is also true.

To encode the directed edges of the filled graph G_π, another n^2 variables $\text{arc}_{i,j}$ are introduced. As all original edges of G are present in G_π, for each $\{v_i, v_j\} \in E$, either $\text{arc}_{i,j}$ or $\text{arc}_{j,i}$ must be set. To be consistent with the ordering implied by $\text{ord}_{i,j}$, we need to enforce that $\text{ord}^*_{i,j}$ implies that $\text{arc}_{j,i}$ is not set.

To describe the elimination process, note that if v_i and v_k are adjacent and v_i and v_j are adjacent with $\pi(i) < \pi(j)$ and $\pi(i) < \pi(k)$, the filled graph G_π contains either the arc (v_j, v_k) or the arc (v_k, v_j). Hence, if $\text{arc}_{i,j}$ and $\text{arc}_{i,k}$ are set and $\text{ord}^*_{j,k}$ is also set, we need to set $\text{arc}_{j,k}$ as well. To ensure that the width of the produced elimination does not exceed a value $t \in \mathbb{N}$, we also need to make sure that for each v_i, at most t edges (v_i, v_j) exist in G_π.

$\forall i,j,k \in \{1,\dots,n\}$	SAT formulation	ILP formulation
$i \neq j, i \neq k, j \neq k$	$\text{ord}^*_{i,j} \wedge \text{ord}^*_{j,k} \implies \text{ord}^*_{i,k}$	$\text{ord}^*_{i,k} - \text{ord}^*_{i,j} - \text{ord}^*_{j,k} \geq -1$
$\{v_i, v_j\} \in E$	$\text{arc}_{i,j} \vee \text{arc}_{j,i}$	$\text{arc}_{i,j} + \text{arc}_{j,i} \geq 1$
$i \neq j$	$\text{ord}^*_{i,j} \implies \neg\text{arc}_{j,i}$	$\text{ord}^*_{i,j} + \text{arc}_{j,i} \leq 2$
$i \neq j, i \neq k, j \neq k$	$\text{ord}^*_{j,k} \wedge \text{arc}_{i,j} \wedge \text{arc}_{i,k} \implies \text{arc}_{j,k}$	$\text{ord}^*_{j,k} + \text{arc}_{i,j} + \text{arc}_{i,k} - \text{arc}_{j,k} \leq 2$
	$\sum_{j=1}^{n} \text{arc}_{i,j} \leq t$	$\sum_{j=1}^{n} \text{arc}_{i,j} \leq t$
		$\text{arc}_{i,j}, \text{ord}_{i,j} \in \{0,1\}$

We extend this formulation by a trick observed in [12]: If $C \subseteq V$ is a clique in $G = (V, E)$, then there is an optimal elimination order π that eliminates C at last. Therefore, if we know a clique C, we can fix it at the end of the permutation. Of course, finding a clique of large cardinality is a difficult problem as well. We find them either with a CSP formulation as well, or, if this it not feasible, with a greedy heuristic. We noticed, however, that CSP solver performed very well on finding maximal cliques in graphs of small tree with – this is not surprising, as the cardinality of the largest clique is bounded by tree width of the graph. Given the clique C, we can extend the formulation as shown in the following table.

$\forall i,j \in \{1,\dots,n\}$	SAT	ILP
$i \in V \setminus C, j \in C$	$\text{ord}^*_{i,j}$	$\text{ord}^*_{i,j}$
$v_i \in C, v_j \in C$	$\text{ord}^*_{i,j}$	$\text{ord}^*_{i,j}$

The SAT formulation $\varphi(t)$ encodes a fixed value $t \in \mathbb{N}$. In order to determine $\text{tw}(G)$, the above encoding would be used for $t = n, n-1, \dots$ until the system does not have any solution, while the ILP would be able to minimize this quantity directly. We make use of the *iterative* abilities of modern SAT solvers that allows to add clauses to an already solved formula and thus does not require resets between the calls – this technique was also recommended by Berg and Järvisalo [7]. The SAT solver is thus able to reuse some of its already computed knowledge. We can thus solve $\varphi(n)$, add the constraints $\sum_{j=1}^{n} \text{arc}_{i,j} \leq n-1$ for each i, solve this new formula (which is equivalent to $\varphi(n-1)$, and repeat this process until $\varphi(t)$ is not satisfiable. Note that the last formula of the SAT formulation is a so called *cardinality constraint* and can be expressed, e. g., via sequential counters or sorting networks. See [4] for a discussion about this topic.

3.1.2 Experimental Evaluations

In order to determine whether a SAT solver or an ILP solver is more suited for finding the solution of our CSP, we performed a number of experimental evaluations. To solve the SAT formula, we made use of the SAT solver lingeling by Biere [8]. To solve the ILP, we used CPLEX of IBM [26]. Our test set was the set of 50 *easy instances* provided by the PACE challenge [28, 21]. The experimental results with a timeout of 5 minutes for each graph were very clear: While lingeling was able to solve 47 of the 50 instances within 14 seconds, CPLEX only managed to solve 23 of the graphs within 5 minutes. Furthermore, lingeling was faster on all of the provided graphs. A graphical representation of the experimental results can be found in Figure 2, where the graphs are sorted by the increasing running time of CPLEX. Here, the running time is shown on the y-axis in seconds.

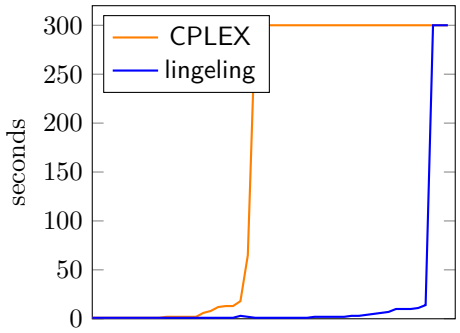

Figure 2 Comparing CPLEX and lingeling on the 50 easy instances of the PACE challenge with a timeout of 5 minutes.

3.2 Point of View: Exact Exponential Algorithms

One of the first algorithms to compute tree decompositions was presented by Arnborg et al. and is based on a $n^{O(\mathrm{tw}(G))}$ brute-force search [1]. Note that this algorithm is polynomial for constant tree width, but uses $O^*(2^n)$ time[3] and memory for non-constant tree width. A similar result can be achieved by combing a result of Seymour and Thomas [31], who have showed a connection of the tree width of the graph and a cops-and-robber search game, together with an algorithm by Berarducci and Intrigila [6], who provided an $n^{O(\mathrm{tw}(G))}$ algorithm to evaluate such games.

The characterization of tree width over elimination orders, as shown in the preliminaries, actually provides a much more rough brute-force approach: simply check all $n!$ possible permutations. It turns out that this strategy, combined with dynamic programming and some heuristics, can lead to $O^*(2^n)$ branch-and-bound algorithms. An example is QuickBB [24].

Finally, Bodlaender et al. have introduced a collection of Held-Karp like dynamic programs to compute optimal tree decompositions [12]. The practical feasible algorithms of this kind have time and space complexity $O^*(2^n)$ as well.

In the design of Jdrasil, it was interesting to study exact exponential time algorithms for two reasons: (1) to evaluate how competitive the SAT approach is against more direct approaches, and (2) to improve the SAT approach on certain instances. As usual for NP-hard problems, we can design instances on which certain algorithms do fail horribly. In our case, we noticed that the SAT approach fails on very symmetric instances. For example, it was not feasible to solve the McGee graph, which has only 24 vertices. We have implemented three different exact algorithms: A version of the cops-and-robber game, a QuickBB inspired branch-and-bound algorithm, and the dynamic program of [12]. Figure 3 shows the running time of one of the algorithms (from left to right: cops-and-robber (CAR), branch-and-bound (BAB), dynamic program (DP)) in blue, against the running time of a SAT solver (its time is shown in orange). The graphs are sorted by the running time of the SAT solver, therefore we see a phase transition in the orange plot (from feasible to unsolved within the time limit of 100 seconds). As we wish to improve the SAT approach, we are interested in the blue plot after the phase transition, i. e., in instances where the SAT solver fails. One can see in the very right plot that the dynamic program does not solve any of these instances and is thus not really useful for us. On the other hand, the cops-and-robber game (plot on the very left) and the branch-and-bound algorithm (center plot) do solve some instances on which the SAT

[3] The O^* notation does not only suppress constants, but also polynomial factors.

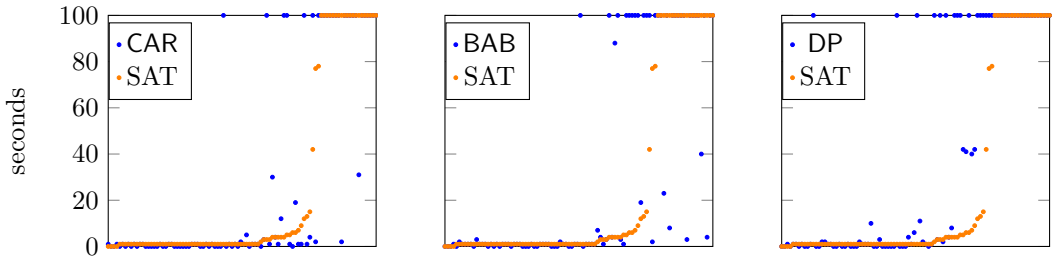

Figure 3 Comparison of cops-and-robber, branch-and-bound, and the dynamic program against the SAT solver on the 193 medium instances of the PACE challenge with a timeout of 100 seconds. The graphs are sorted by the running time of the SAT solver.

solver fails (whenever there is a blue peek after the phase transition). But on the down site, there are also a lot of instances where these algorithm fail, but SAT succeeds (blue peeks before the phase transition).

We conclude the following from the above experiment: First of all, the SAT approach is competitive, as it solves a couple of instances which are unsolved by all other algorithms. Second, there are instances that can be solved much quicker by the cops-and-robber game or the branch-and-bound algorithm and we face, therefore, the new problem of deciding when to use which algorithm.

3.3 Point of View: Upper and Lower Bounds

Research on heuristics for tree width is long standing and multi-faceted, including many different upper and lower bound algorithms. For an overview, we refer to the detailed survey papers by Bodleander and Koster [14, 15].

3.3.1 Upper bounds

The characterization of tree width over elimination orderings gives access to very simple, but still powerful, heuristics running in low-order polynomial running time (ranging from $\mathcal{O}(n^2)$ to $\mathcal{O}(n^4)$ depending on the concrete heuristic). Recall that for graph $G = (V, E)$, every permutation π_V of V corresponds to a tree decomposition of G, and there is always one corresponding to an optimal decomposition. Many heuristics try to find such a permutation greedily: let π_S be a permutation of $S \subseteq V$, i. e., a partial permutation of V; the greedy algorithm selects a vertex $v \in V \setminus S$ that minimize some value function $\gamma(v)$ in the current graph $H = (V, E_{\pi_S})$ and appends v to the partial permutation π_S obtaining an updated permutation $\pi_{S \cup \{v\}}$. While many value functions γ are possible, an overview of six reasonable ones is given in [14]. They are summarized in Table 1, where $\psi_H(v) = |\{\, \{u, w\} \mid \{v, u\} \in E(H), \{v, w\} \in E(H), \{u, w\} \notin E(H) \,\}|$ equals the number of so called *fillin edges* and $\delta_H(v)$ is the degree of v in H.

We have implemented all of them to compare their quality. In the following plots, we sorted the graphs by their tree width (shown in orange), and have plotted the upper bounds produced by the heuristics in black. On the left picture of Figure 4, the heuristics with the six value functions from [14] are shown. We have omitted a labeling, because the message of this plot is not that a certain heuristic a is better on some instance x, but rather that they are all solid and that there is a lot of noise about which heuristic is better on which instance.

We used the result of the experiment to derive the heuristic that we now actually use. Note that, if we greedily select a vertex v that minimizes $\gamma(v)$, we may end up in a situation

Table 1 Value functions used by the upper bounds.

Name	$\gamma(v)$
Degree	$\delta_H(v)$
FillIn	$\psi_H(v)$
Degree+FillIn	$\delta_H(v) + \psi_H(v)$
SparsestSubgraph	$\psi_H(v) - \delta_H(v)$
FillInDegree	$\delta_H(v) + \psi_H(v)/n^2$
DegreeFillIn	$\delta_H(v) + \psi_H(v)/n$

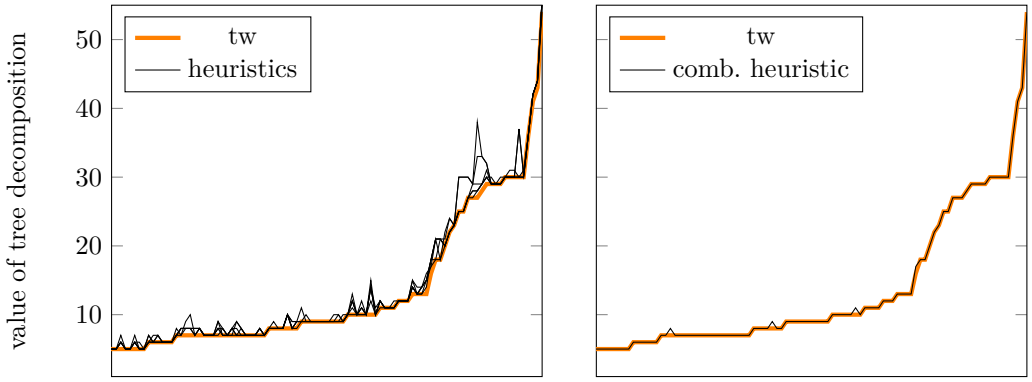

Figure 4 Comparison of the six upper bound heuristics against the optimal tree width on the 193 medium instances of the PACE challenge. The graphs are sorted by their tree width.

where we have a tie of more then one vertex. By breaking these ties randomly, we obtain a *randomized* algorithm. Already on very small test sets one can observe that the quality of this algorithm improves if we run it multiple times. On the other hand, if we repeat the algorithm multiple times, we do not have to fix a function γ. We have obtained very good results by running the heuristic $O(\sqrt{n})$ times and by selecting the value function γ in each run at random (from the pool of the six functions). An further improvement we did is a *look-ahead*: instead of choosing the vertex that minimizes γ, we take the vertex that minimizes the sum over the next c choices (for a constant c). With this extension (already for $c = 2$) we obtain the right plot from above. Note that of the 193 graphs of the test set, there are only 3 graphs on which the heuristic did not find the optimum.

3.3.2 Lower Bounds

There are a couple of very different approaches to compute lower bounds for the tree width of a graph. We refer to the second paper of Bodleander and Koster for an overview [15]. A promising approach is based on the fact that the tree width of every minor of a graph is bounded by the tree width of the host graph. Gogate and Dechter have developed a lower bound algorithm that greedily tries to find a minor with high tree width [24]. It repeatedly chooses a vertices v of minimum degree and one of its neighbors w and contracts the edge $\{v, w\}$. The largest minimum degree encountered in this process then yields a lower bound on the tree width. This algorithm can be used with different strategies concerning the greedy selection of the neighbour w that minimizes the value function $\gamma_v(w)$ in the current contracted graph H [15]. They are summarized in Table 2, where $N_H(v)$ denotes the neighbourhood of v in H.

Table 2 Value functions used by the lower bounds.

Name	$\gamma_v(w)$
min-d	$\delta_H(w)$
max-d	$-\delta_H(w)$
least-c	$\lvert N_H(v) \cap N_H(w)\rvert$

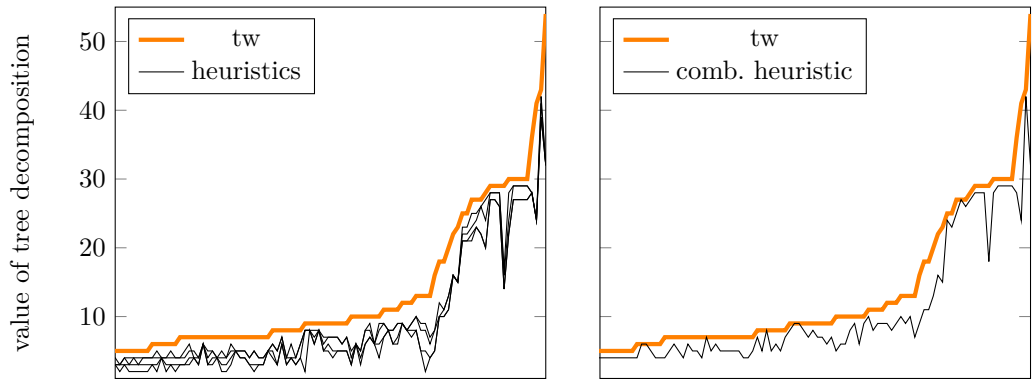

Figure 5 Comparison of the three lower bound heuristics against the optimal tree width on the 193 medium instances of the PACE challenge. The graphs are sorted by their tree width.

We have implemented the algorithm with the three strategies discussed in [15]. The results are shown in the following plots in Figure 5. The graphs are sorted by their tree width, which is plotted in orange. In the left plot, the lower bounds produced by the heuristic with the three different strategies is shown. We again omit the labels, as we wish to show the trend. One can see that the lower bounds have less quality then the upper bounds and that one strategy – least-c – actually dominates the others.

In [15] it is surveyed how the performance of a lower bound algorithm $\mathbb{A}$ can be boosted. The key idea is to work in the *k-neighbor improved graph*, which is obtained from the input graph by adding edges between all vertices that share k common neighbors. Starting with $k = \text{low}$, where low is the lower bound produced by $\mathbb{A}$ on the input graph G, we obtain a new graph G'. Then we can run $\mathbb{A}$ on this graph and eventually increase low allowing us to compute a new neighbor improved graph. Combined with the contraction idea of the algorithm of Gogate and Dechter, this yield a powerful lower bound algorithm (in [15] it is called LBN+). The quality of the produced lower bounds can be seen in the right plot above.

3.4 Point of View: Parameterized Complexity

The concept of tree width plays a central role in the field of parameterized complexity theory and has thus obtained a lot of attention in this area. While we typically model problems as languages $L \subseteq \Sigma^*$ over some fixed alphabet Σ, we define a *parameterized problem* as a tuple (Q, κ) with $Q \subseteq \Sigma^*$ and $\kappa\colon \Sigma^* \to \mathbb{N}$. The intuition is that the language Q models, as before, the problem, while κ (the parameter) highlights some special property of the instance. One can now analyze the running time of an algorithm for (Q, κ) with respect to both, the instance size and its parameter. We say a parameterized problem (Q, κ) is *fixed-parameter tractable* if there is an algorithm that decides for every $w \in \Sigma^*$ whether or not $w \in Q$ holds in time $f(\kappa(w)) \cdot \text{poly}(|w|)$, where $f\colon \mathbb{N} \to \mathbb{N}$ is some computable function. Note that we

can parameterize a problem in many different ways (number of vertices, maximum degree, solution size, girth, ...), and that some of the resulting parameterized problems may be fixed-parameter tractable while others may not.

Many NP- or even PSPACE-hard graph problems are fixed-parameter tractable with respect to the parameter tree width. A prime example is Courcell's Theorem [19], which states that all problems definable in monadic second-order logic can be solved in linear time, if a small tree decomposition is presented together with the input. This leads to the requirement of an algorithm that, given a graph $G = (V, E)$ with small tree width, computes an optimal tree decomposition of G. Such an algorithm was found by Bodlaender [9] and runs in time $f(\mathrm{tw}(G)) \cdot n$. Although this algorithms solves the problem theoretically, it can not be used in practice due to its huge constants [14]. Therefore, the parameterized complexity community has continued its search for a fast algorithm.

An important concept in parameterized complexity is *preprocessing*. Given an instance w of a parameterized problem (Q, κ), we wish to reduce it in polynomial time to a new instance w' with $|w'| \leq h(\kappa(w))$ for a computable function h. That is, we wish to reduce the problems to its hard core (the *kernel*), whose size may only depend on the parameter. This process is called *kernelization*. A positive result is that every fixed-parameter tractable problem has such a kernelization [18], and so does tree width. On the other hand, there are problems which probably do not have a kernel of polynomial size. Unfortunately, tree width is one of these problems [10]. The seek for good kernelization algorithms for tree width has lead to very efficient *heuristic reduction rules*, which safely reduce the graph but do not give any guarantees on their effectiveness [16]. A refined analysis of these reduction rules can be used to find polynomial kernels for tree width with respect to other parameters such as the vertex cover number or the size of a feedback vertex set [13].

Beside the effort of introducing good preprocessing algorithms, the parameterized complexity community has complemented the theoretically fast algorithm by Bodlaender [9] with actual fast *constant size approximation algorithms*. The first $4k + 3$ approximation algorithm running in time $O(3^{3k} \cdot n^2)$ was introduced by Robertson and Seymour during their quest to prove the graph minor theorem [29]. This was constantly improved by various authors (see [11] for an survey). The latest result is a $5k + 4$ algorithm running in time $2^{O(k)} \cdot n$.

For an implementation that should compute an optimal tree decomposition, an approximation algorithm as the one by Robertson and Seymour can be interesting in two ways: it produces lower *and* upper bounds at the same time. We have implemented an algorithm that is inspired by the one of Robertson and Seymour (see the textbook [20] for details), and have analyzed its practical running time and the quality of the produced lower and upper bounds. The following two plots in Figure 6 show the results of our experiments. On the left, we have plotted the running time (in seconds) of the approximation algorithm in blue against the running time of the SAT approach in orange. Here, the graphs are again sorted by the running time of the SAT solver and, hence, we have a phase transition in the orange plot from feasible to not feasible. The algorithm performs quite well and, in particular, does only fail on very few instances that can be handled by the SAT solver. The plot on the right shows the computed lower and upper bounds (in blue) against the exact tree width of the graphs (in orange). Here, the graphs are sorted by their tree width. While these bounds are reasonable – as expected from an approximation algorithm – they come short in comparison to the lower and upper bounds described in the last section.

From the experiments we conclude that the approximation algorithm delivers, besides its theoretically beauty, a practical access to the computation of tree decompositions. However, it falls short against the other algorithms in our tool box and is thus not used in our final version.

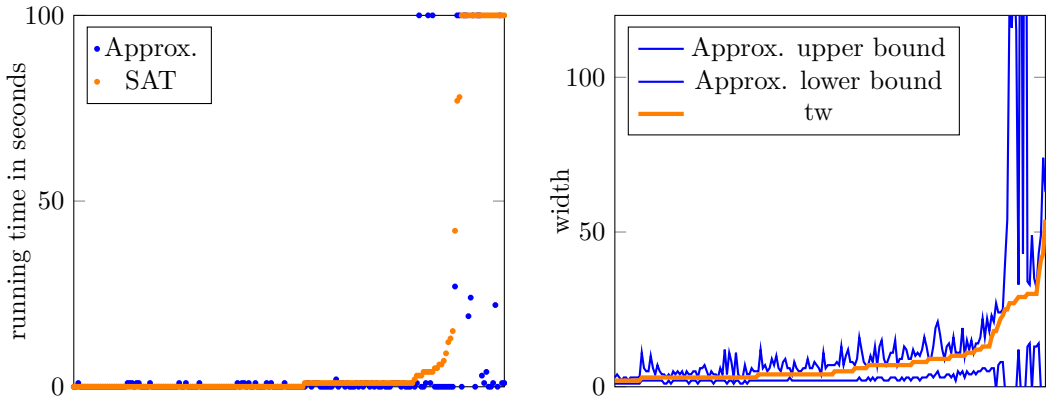

Figure 6 Comparison of the approximation algorithm and the SAT solver on the 193 medium instances of the PACE challenge with a timeout of 100 seconds. On the left, the graphs are sorted by the running time of the SAT solver and by their tree width on the right.

3.5 Cherry-Pick the Best from each World

While Jdrasil is designed as library and provides access to all implemented algorithms mentioned above, we have do decide which of these we actually wish to use if we want to compute an exact tree decomposition. The core of our algorithm is the SAT approach and we try to use it whenever possible. It is known that different SAT solvers behave differently on specific formulas and we, thus, tested different solvers on our formulas. It turned out that the SAT solver glucose of Audemard and Simon [3] – based on the classical solver MiniSat by Eén and Sörensson [23] – outperforms lingeling in our setting. We thus chose this solver for use in our final version.

We use the preprocessing techniques developed by the FPT community as reducing the instance almost always makes sense. This is in particular important from the view point of using a SAT solver, as the SAT solver does handle a huge tree in the same way it handles every other graph. The parameterized point of view also influenced the design of the formula, which uses many cardinality constraints. These constraints are usually implemented at the cost of $O(n \log n)$ auxiliary variables. However, we can also implement them using $O(kn)$ auxiliary variables, where k is the tree width of the graph, i. e., we craft the formula with respect to a parameter.

We use the lower and upper bound heuristics presented to prune the search space and to reduce the number of formulas that we have to consider. From the different exact exponential time algorithms, we choose to use the cops-and-robber game and the SAT approach. While the cops-and-robber game *always* runs in time $n^{\mathcal{O}(\mathrm{tw}(G))}$, the behaviour of the SAT solver is much more diverse. While it is able to solve `contiki_lpp_send_probe.gr` – a graph containing 92 vertices – within 54 seconds, it is not able solve the McGee graph containing 24 vertices. In contrast to this, the cops-and-robber game solve the McGee graph within a second. Our computational experiments showed that the cops-and-robber outperformed the other approaches for those graphs with treewidth at most 8 or at most 25 vertices. We thus use the cops-and-robber game, if our computed upper bound is at most 8 or if the graph has at most 25 vertices. On the other graphs, we use the SAT solver. Note that, of course, this decision is made after preprocessing, i. e., depending on the size of the "hard core" of the graph. The performance of the complete algorithm is analyzed in the next section.

4 Experimental Results

4.1 PACE 2016

As noted in the introduction, the Parameterized Algorithms and Computational Experiments (PACE) challenge was started in 2016 to "investigate the applicability of algorithmic ideas studied and developed in the subfields of multivariate, fine-grained, parameterized, or fixed-parameter tractable algorithms" [28, 21]. The challenge consisted of two tracks: one for tree width and one for feedback vertex set. We submitted the algorithm described in Section 3.5 to the the exact sequential competition of the tree width track (i. e. algorithms needed to output the optimal tree decomposition without the use of parallelism). The programs were given 200 graphs with predetermined timeouts ranging between 100 seconds an 3600 seconds. Our program (Jdrasil [5]) took the third place and solved 166 of the graphs. The second place was awarded to the program BZTreewidth [17] of Hans Bodlaender and Tom van der Zanden that solved 173 instances. They used a combination of dynamic programming and balanced separators. Finally, the first place was awarded to the program Exact treewidth [33] by Hisao Tamaki that solved 199 instances and used an improved version of the algorithm of Arnborg et al. [1].

4.2 Graph Benchmarks

The authors of this work later found a subtle bug in script configuring their implementation of Jdrasil that scheduled all simultaneously running instance of Jdrasil on the same processor. To test the feasibility of our approach, we ran the three winners of the PACE challenge on a benchmark set of 1813 graphs with a timeout of 300 seconds on each graph. Note that these experiments were performed in order to test the feasibility of our cherry-picking approach on a wide range of graphs within a reasonable time frame. They are *not* intended to be a replacement of the experiments determining the winner of the PACE challenge (where sophisticated voting rules were used to determine the winner). This yielded the following results:

	number of solved instances	average running time
Jdrasil [5]	1188	5,48 seconds
BZTreewidth [17]	1004	7,83 seconds
Exact treewidth [33]	1307	4,53 seconds

In order to understand the different behaviours of the algorithms, we have also looked at all combinations (p_1, p_2) of programs and the instances they were not able to solve. In the following table, an entry $x/y/z$ in the row labeled with p_1 and in the column labeled with p_2 denotes that the number of instances that p_1 and p_2 did both not solve was x, the number of instances that p_1 did not solve, but p_2 did, was y, and the number of instances that p_2 did not solve, but p_1 did, was z. For example, the colored entry shows that there were 505 instances, which neither Jdrasil nor BZTreewidth solved, only 30 instances that Jdrasil did not solve, but BZTreewidth solved, and 188 instances that Jdrasil solved, but BZTreewidth did not.

	Jdrasil	BZTreewidth	Exact treewidth
Jdrasil	–	505/30/188	329/206/96
BZTreewidth	505/188/30	–	397/296/28
Exact treewidth	329/96/209	397/28/296	–

Besided the average running time of the programs, another important aspect is the number of graphs where p_1 was faster than p_2. In the following table, an entry $x/y/z$ in the row labeled with p_1 and in the column labeled with p_2 denotes that p_1 was faster than p_2 on x graphs, while p_1 was faster than p_2 on y graphs, and z denotes the seconds p_1 was faster, minus the seconds p_2 was faster, i. e., if it is positive, p_1 was faster in total. This is an important information: It could be the case that p_1 is a second faster on half of the instances, but that p_2 is multiple minutes faster on a quarter of the remaining instances. While p_1 would outperform p_2 concerning the number of instances it solved faster, p_2 is clearly preferable. For example, the colored entry shows that Jdrasil was faster than BZTreewidth on 280 instances, while BZTreewidth outperformed Jdrasil on 236 instances. In total, Jdrasil outperformed BZTreewidth by 10231 seconds.

	Jdrasil	BZTreewidth	Exact treewidth
Jdrasil	–	280/236/10231	16/590/ − 6077
BZTreewidth	236/280/ − 10231	–	15/387/ − 12214
Exact treewidth	590/16/6077	387/15/12214	–

In summary, we believe that the results of this section show that the "cherry-picking"-approach described in the last section is competitive even to very specialized implementations. Our program Jdrasil solved more instances than the second place winner BZTreewidth and is substantially faster in total.

5 Handle the Use Cases

Case Example A: In order to compute the tree width of the causal graph, the researcher downloads Jdrasil from its homepage [5] and enters the command `$ ./gradlew exact` to build the scripts `tw-exact` (for Unix) or `tw-exact.bat` (for Windows). If her graph is stored in the file `graph.gr` (either in the PACE format [28, 21] or in the DIMACS graph format [22]), she can compute an optimal tree decomposition with the following command: `$ ./tw-exact < graph.gr`

Case Example B: The professor tells the student to look into the Jdrasil manual, which can be generated by the command `$ ./gradlew manual` that produces the manual in the directory `build/docs/manual`. After careful reading, he writes the following Java code that uses Jdrasil to compute an exact tree decomposition:

```
import jdrasil.algorithms.ExactDecomposer;
import jdrasil.graph.Graph;
import jdrasil.graph.TreeDecomposition;

public class Algorithm {
    public int computeFirstAlgorithm(Graph<Integer> g) {
        TreeDecomposition<Integer> decomposition = null;
        try {
          ExactDecomposer<Integer> ex = new ExactDecomposer<>(g);
          decomposition = ex.call();
        }
        ...
```

He can compile and run this code by including the file `Jdrasil.jar` produced by `$ ./gradlew jar` which can be found in the directory `build/jars`.

Case Example C: The Ph. D. student looks at the documentation of the java code generated by `$ ./gradlew javadoc` and finds the generated HTML files in the directory `build/docs/javadoc`. After considering which classes she needs, she decides to make use of the class `graph.Graph` (to use an efficient implementation of the underlying graph), the class `algorithms.lowerbounds.MinorMinWidthLowerbound` (to compute a lower bound) and the class `algorithms.preprocessing.GraphReducer` (to reduce the graph by using the reduction rules of [16]).

6 Conclusion

In this paper we have presented our Java library Jdrasil for the computation of tree decompositions. The goals we have achieved with the library are threefold: first of all we hope that the library gives algorithmic engineers, who wish to work on tree decompositions, an easy access to these complex graph theoretic structures (Section 5). On the other hand, Jdrasil implements a broad range of tools that can be used by theorists or engineers that wish to implement new algorithms for computing tree decompositions (Section 3). Our computational results imply that different algorithms are needed for different graphs and we show that combining several of those algorithms allows us to be competitive against other optimized implementations (Section 4). This "cherry-picking" can be done easily due to the highly modular design of Jdrasil.

All together, we hope that Jdrasil will be helpful for studying tree decompositions both in a theoretical and practical domain; and we look towards to further improve and extend the implementation.

References

1 Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of Finding Embeddings in a k-Tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987. doi:10.1137/0608024.

2 Stefan Arnborg and Andrzej Proskurowski. Linear Time Algorithms for NP-hard Problems Restricted to Partial k-Trees. *Discrete applied mathematics*, 23(1):11–24, 1989. doi:10.1016/0166-218X(89)90031-0.

3 Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proc. IJCAI*, pages 399–404, 2009.

4 Olivier Bailleux and Yacine Boufkhad. Efficient CNF Encoding of Boolean Cardinality Constraints. In *Proc. CP*, pages 108–122. Springer, 2003. doi:10.1007/978-3-540-45193-8_8.

5 Max Bannach, Sebastian Berndt, and Thorsten Ehlers. Jdrasil, 2016. URL: https://github.com/maxbannach/Jdrasil.

6 Alessandro Berarducci and Benedetto Intrigila. On the Cop Number of a Graph. *Advances in Applied Mathematics*, 14(4):389–403, 1993. doi:10.1006/aama.1993.1019.

7 Jeremias Berg and Matti Järvisalo. SAT-Based Approaches to Treewidth Computation: An Evaluation. In *Proc. ICTAI*, pages 328–335. IEEE Computer Society, 2014. doi:10.1109/ICTAI.2014.57.

8 Armin Biere. Lingeling, Plingeling, Picosat and Precosat at SAT Race 2010. *FMV Report Series Technical Report*, 10(1), 2010.

9 Hans L. Bodlaender. A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth. In *Proc. STOC*, pages 226–234. ACM, 1993. doi:10.1145/167088.167161.

10 Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Danny Hermelin. On Problems Without Polynomial Kernels. *Journal of Computer and System Sciences*, 75(8):423–434, 2009. doi:10.1016/j.jcss.2009.04.001.

11 Hans L. Bodlaender, Pal Gronas Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. An $O(c^k n)$ 5-Approximation Algorithm for Treewidth. In *Proc. FOCS*, pages 499–508, Oct 2013. doi:10.1109/FOCS.2013.60.

12 Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On Exact Algorithms for Treewidth. *ACM Trans. Algorithms*, 9(1):12:1–12:23, 2012. doi:10.1145/2390176.2390188.

13 Hans L. Bodlaender, Bart M. P. Jansen, and Stefan Kratsch. Preprocessing for Treewidth: A Combinatorial Analysis through Kernelization. In *Proc. ICALP*, volume 6755 of *Lecture Notes in Computer Science*, pages 437–448. Springer, 2011. doi:10.1137/120903518.

14 Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth Computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010. doi:10.1016/j.ic.2009.03.008.

15 Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations II. Lower Bounds. *Information and Computation*, 209(7):1103–1119, 2011. doi:10.1016/j.ic.2011.04.003.

16 Hans L. Bodlaender, Arie M. C. A. Koster, and Frank van den Eijkhof. Pre-Processing for Triangulation of Probabilistic Networks. *Computational Intelligence*, 21(3):286–305, 2005. doi:10.1111/j.1467-8640.2005.00274.x.

17 Hans L. Bodlaender and Tom van der Zanden. BZTreewidth, 2016. URL: https://github.com/TomvdZanden/BZTreewidth.

18 Liming Cai, Jianer Chen, Rodney G. Downey, and Michael R. Fellows. Advice Classes of Parameterized Tractability. *Ann. Pure Appl. Logic*, 84(1):119–138, 1997. doi:10.1016/S0168-0072(95)00020-8.

19 Bruno Courcelle. Graph Rewriting: An Algebraic and Logic Approach. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 193–242. Elsevier, Amsterdam, Netherlands and MIT Press, Cambridge, Massachusetts, 1990. doi:10.1016/B978-0-444-88074-1.50010-X.

20 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Daniel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2015.

21 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 *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *LIPIcs*, pages 30:1–30:9. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.IPEC.2016.30.

22 DIMACS Graph Format. Accessed: 2017-01-26. URL: http://prolland.free.fr/works/research/dsat/dimacs.html.

23 Niklas Eén and Niklas Sörensson. An Extensible SAT-Solver. In *Proc. SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.

24 Vibhav Gogate and Rina Dechter. A Complete Anytime Algorithm for Treewidth. In *Proc. UAI*, pages 201–208. AUAI Press, 2004.

25 Thore Husfeldt and Iyad A. Kanj, editors. *Proc. IPEC*, volume 43 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015.

26 IBM. *IBM ILOG CPLEX Optimization Studio CPLEX User's Manual*. URL: https://www.ibm.com/support/knowledgecenter/SSSA5P_12.6.1/ilog.odms.studio.help/pdf/usrcplex.pdf.

27 David R. Karger and Nathan Srebro. Learning Markov Networks: Maximum Bounded Tree-Width Graphs. In *Proc. SODA*, pages 392–401. ACM/SIAM, 2001.

28 The Parameterized Algorithms and Computational Experiments Challenge (PACE). Accessed: 2017-01-26. URL: https://pacechallenge.wordpress.com/.

29 Neil Robertson and Paul D. Seymour. Graph Minors. XIII. The Disjoint Paths Problem. *Journal of Combinatorial Theory*, 63(1):65–110, 1995. doi:10.1007/978-3-540-24605-3_37.

30 Marko Samer and Helmut Veith. Encoding Treewidth into SAT. In *Proc. SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 45–50. Springer, 2009. doi:10.1007/978-3-642-02777-2_6.

31 Paul D. Seymour and Robin Thomas. Graph Searching and a Min-Max Theorem for Tree-Width. *Journal of Combinatorial Theory, Series B*, 58(1):22–33, 1993. doi:10.1006/jctb.1993.1027.

32 Yinglei Song, Chunmei Liu, Russell L. Malmberg, Fangfang Pan, and Liming Cai. Tree Decomposition Based Fast Search of RNA Structures Including Pseudoknots in Genomes. In *Proc. CSB*, pages 223–234. IEEE Computer Society, 2005.

33 Hisao Tamaki. Exact treewidth, 2016. URL: https://github.com/TCS-Meiji/treewidth-exact.

A Graph Benchmarks

The 50 easy instances used in Section 3.1 are taken from the easy instances provided by the PACE challenge. The notation $(n/m/t)$ means that the graph has n vertices, m edges and tree width t.

- BalancedTree_3,5.gr (364/363/1)
- contiki_collect_send_next_packet.gr (26/25/1)
- contiki_ctk_ctk_menu_add.gr (25/27/2)
- contiki_cxmac_input_packet.gr (90/97/3)
- contiki_dhcpc_dhcpc_init.gr (34/34/2)
- contiki_dhcpc_dhcpc_request.gr (27/27/2)
- contiki_httpd-cfs_send_file.gr (44/48/3)
- contiki_ifft_ifft.gr (172/180/2)
- contiki_ircc_list_channel.gr (70/76/3)
- contiki_lpp_send_packet.gr (116/120/2)
- contiki_nullrdc_packet_input.gr (28/30/3)
- contiki_polite-announcement_send_timer.gr (31/31/2)
- contiki_powertrace_add_stats.gr (46/47/2)
- contiki_powertrace_powertrace_print.gr (323/323/2)
- contiki_profile_profile_episode_start.gr (31/32/2)
- contiki_psock_psock_generator_send.gr (61/68/4)
- contiki_ringbuf_ringbuf_put.gr (29/29/2)
- contiki_rudolph0_send_nack.gr (27/26/1)
- contiki_rudolph1_rudolph1_send.gr (30/29/1)
- contiki_runicast_runicast_open.gr (24/23/1)
- contiki_shell-collect-view_process_thread_collect_view_data_process.gr (61/62/2)
- contiki_shell-rime-ping_recv_mesh.gr (47/47/2)
- contiki_shell-rime_recv_collect.gr (62/64/2)
- contiki_shell-text_process_thread_shell_echo_process.gr (25/25/2)
- contiki_shell_shell_register_command.gr (42/45/2)
- contiki_tcpip_eventhandler.gr (98/112/2)
- contiki_uip-neighbor_uip_neighbor_add.gr (67/71/3)
- contiki_uip-over-mesh_recv_data.gr (85/88/2)
- contiki_uip_uip_init.gr (26/27/2)
- contiki_webclient_senddata.gr (108/109/2)

- fuzix_clock_settime_clock_settime.gr (20/21/2)
- fuzix_devf_fd_transfer.gr (119/129/3)
- fuzix_devio_bfind.gr (27/29/3)
- fuzix_devio_kprintf.gr (69/78/3)
- fuzix_difftime_difftime.gr (74/73/1)
- fuzix_fgets_fgets.gr (53/58/3)
- fuzix_filesys_filename.gr (45/48/3)
- fuzix_filesys_getinode.gr (52/57/3)
- fuzix_filesys_i_open.gr (129/143/3)
- fuzix_filesys_newfstab.gr (20/21/2)
- fuzix_getpass__gets.gr (31/35/3)
- fuzix_malloc___insert_chunk.gr (104/116/3)
- fuzix_process_getproc.gr (32/35/2)
- fuzix_ran_rand.gr (46/48/2)
- fuzix_regexp_regcomp.gr (118/129/2)
- fuzix_se_ycomp.gr (83/96/3)
- fuzix_stat_statfix.gr (52/51/1)
- fuzix_syscall_fs2_chown_op.gr (27/28/2)
- fuzix_tty_tty_read.gr (123/137/4)

The 193 instances used in Section 3.2, Section 3.3, and in Section 3.4 are taken from the 100 second instances of the PACE challenge. The notation (n/m) means that the graph has n vertices and m edges.

- AhrensSzekeresGeneralizedQuadrangleGraph_3.gr (27/135)
- BalancedTree_3,5.gr (364/363)
- BlanusaSecondSnarkGraph.gr (18/27)
- ChvatalGraph.gr (12/24)
- ClebschGraph.gr (16/40)
- CycleGraph_100.gr (100/100)
- DesarguesGraph.gr (20/30)
- DodecahedralGraph.gr (20/30)
- DorogovtsevGoltsevMendesGraph.gr (3282/6561)
- DoubleStarSnark.gr (30/45)
- DyckGraph.gr (32/48)
- ErreraGraph.gr (17/45)
- FibonacciTree_10.gr (143/142)
- FlowerSnark.gr (20/30)
- FolkmanGraph.gr (20/40)
- FriendshipGraph_10.gr (21/30)
- GNP_20_10_0.gr (20/28)
- GNP_20_10_1.gr (20/24)
- GNP_20_20_0.gr (20/46)
- GNP_20_20_1.gr (20/48)
- GNP_20_30_0.gr (20/56)
- GNP_20_30_1.gr (20/63)
- GNP_20_40_0.gr (20/78)
- GNP_20_40_1.gr (20/71)
- GNP_20_50_0.gr (20/91)
- GNP_20_50_1.gr (20/106)
- GeneralizedPetersenGraph_10_4.gr (20/30)
- GoethalsSeidelGraph_2_3.gr (16/72)
- GoldnerHararyGraph.gr (11/27)
- GrayGraph.gr (54/81)
- GrotzschGraph.gr (11/20)

- HararyGraph_6_15.gr (15/45)
- HeawoodGraph.gr (14/21)
- HoffmanGraph.gr (16/32)
- HyperStarGraph_10_2.gr (45/72)
- IcosahedralGraph.gr (12/30)
- KneserGraph_10_2.gr (45/630)
- LadderGraph_20.gr (40/58)
- MarkstroemGraph.gr (24/36)
- McGeeGraph.gr (24/36)
- MeredithGraph.gr (70/140)
- NauruGraph.gr (24/36)
- NonisotropicOrthogonalPolarGraph_3_5.gr (15/60)
- NonisotropicUnitaryPolarGraph_3_3.gr (63/1008)
- OddGraph_4.gr (35/70)
- OrthogonalArrayBlockGraph_4_3.gr (9/36)
- PaleyGraph_17.gr (17/68)
- PappusGraph.gr (18/27)
- PoussinGraph.gr (15/39)
- RKT_20_40_10_0.gr (20/87)
- RKT_20_40_10_1.gr (20/87)
- RKT_20_50_10_0.gr (20/73)
- RKT_20_50_10_1.gr (20/73)
- RKT_20_60_10_0.gr (20/58)
- RKT_20_60_10_1.gr (20/58)
- RKT_20_70_10_0.gr (20/44)
- RKT_20_70_10_1.gr (20/44)
- RKT_20_80_10_0.gr (20/29)
- RKT_20_80_10_1.gr (20/29)
- RandomBarabasiAlbert_100_2.gr (100/196)
- RandomBipartite_10_50_3.gr (60/138)
- RandomGNM_100_100.gr (100/100)
- RingedTree_6.gr (63/123)
- SchlaefliGraph.gr (27/216)
- ShrikhandeGraph.gr (16/48)
- SierpinskiGasketGraph_3.gr (15/27)
- SquaredSkewHadamardMatrixGraph_2.gr (49/588)
- StarGraph_100.gr (101/100)
- SylvesterGraph.gr (36/90)
- SzekeresSnarkGraph.gr (50/75)
- TaylorTwographDescendantSRG_3.gr (27/135)
- TaylorTwographSRG_3.gr (28/210)
- Toroidal6RegularGrid2dGraph_4_6.gr (24/72)
- WheelGraph_100.gr (100/198)
- WorldMap.gr (166/323)
- contiki_calc_input_to_operand1.gr (31/33)
- contiki_collect_enqueue_dummy_packet.gr (46/46)
- contiki_collect_received_announcement.gr (52/59)
- contiki_collect_send_ack.gr (53/52)
- contiki_collect_send_next_packet.gr (26/25)
- contiki_collect_send_queued_packet.gr (95/99)
- contiki_contikimac_input_packet.gr (116/127)
- contiki_contikimac_powercycle.gr (166/194)
- contiki_ctk_ctk_menu_add.gr (25/27)
- contiki_cxmac_input_packet.gr (90/97)
- contiki_dhcpc_dhcpc_init.gr (34/34)

- contiki_dhcpc_dhcpc_request.gr (27/27)
- contiki_dhcpc_handle_dhcp.gr (276/313)
- contiki_httpd-cfs_send_file.gr (44/48)
- contiki_httpd-cfs_send_headers.gr (106/116)
- contiki_ifft_ifft.gr (172/180)
- contiki_ircc_handle_connection.gr (138/161)
- contiki_ircc_list_channel.gr (70/76)
- contiki_lpp_dutycycle.gr (102/114)
- contiki_lpp_init.gr (22/21)
- contiki_lpp_send_packet.gr (116/120)
- contiki_lpp_send_probe.gr (92/94)
- contiki_nullrdc_packet_input.gr (28/30)
- contiki_polite-announcement_send_timer.gr (31/31)
- contiki_powertrace_add_stats.gr (46/47)
- contiki_powertrace_powertrace_print.gr (323/323)
- contiki_process_exit_process.gr (72/82)
- contiki_profile_profile_episode_start.gr (31/32)
- contiki_psock_psock_generator_send.gr (61/68)
- contiki_psock_psock_readto.gr (56/61)
- contiki_ringbuf_ringbuf_put.gr (29/29)
- contiki_route-discovery_route_discovery_discover.gr (20/20)
- contiki_rudolph1_rudolph1_open.gr (27/26)
- contiki_rudolph1_write_data.gr (35/36)
- contiki_serial-line_process_thread_serial_line_process.gr (72/81)
- contiki_shell-base64_base64_add_char.gr (70/74)
- contiki_shell-collect-view_process_thread_collect_view_data_process.gr (61/62)
- contiki_shell-netperf_memcpy_misaligned.gr (30/32)
- contiki_shell-ps_process_thread_shell_ps_process.gr (45/46)
- contiki_shell-rime-debug_recv_broadcast.gr (24/23)
- contiki_shell-rime-ping_recv_mesh.gr (47/47)
- contiki_shell-rime_process_thread_shell_send_process.gr (89/95)
- contiki_shell-rime_recv_collect.gr (62/64)
- contiki_shell-sendtest_read_chunk.gr (30/32)
- contiki_shell-text_process_thread_shell_echo_process.gr (25/25)
- contiki_shell_process_thread_shell_server_process.gr (76/85)
- contiki_shell_shell_register_command.gr (42/45)
- contiki_tcpip_eventhandler.gr (98/112)
- contiki_uip-neighbor_uip_neighbor_add.gr (67/71)
- contiki_uip-neighbor_uip_neighbor_periodic.gr (20/21)
- contiki_uip-over-mesh_recv_data.gr (85/88)
- contiki_uip_uip_connect.gr (111/120)
- contiki_uip_uip_init.gr (26/27)
- contiki_uip_uip_unlisten.gr (19/20)
- contiki_webclient_senddata.gr (108/109)
- contiki_webclient_webclient_appcall.gr (98/111)
- dimacs_anna.gr (138/260)
- dimacs_fpsol2.i.3.gr (206/2645)
- dimacs_inithx.i.2.gr (299/5162)
- dimacs_inithx.i.2-pp.gr (220/4165)
- dimacs_inithx.i.3-pp.gr (196/2185)
- dimacs_jean.gr (77/184)
- dimacs_miles1000.gr (128/1594)
- dimacs_miles250.gr (125/241)
- dimacs_miles750.gr (128/1252)
- dimacs_mulsol.i.1.gr (100/1725)

- dimacs_mulsol.i.2.gr (101/1233)
- dimacs_mulsol.i.3.gr (102/1233)
- dimacs_mulsol.i.4-pp.gr (78/1062)
- dimacs_mulsol.i.5.gr (102/1224)
- dimacs_mulsol.i.5-pp.gr (77/974)
- dimacs_myciel5.gr (46/139)
- dimacs_queen5_5.gr (25/106)
- dimacs_queen6_6.gr (36/217)
- dimacs_queen7_7.gr (49/388)
- dimacs_zeroin.i.2.gr (85/951)
- dimacs_zeroin.i.3.gr (83/917)
- dimacs_zeroin.i.3-pp.gr (49/651)
- fuzix_abort_abort.gr (21/20)
- fuzix_bankfixe_pagemap_alloc.gr (21/22)
- fuzix_clock_gettime_clock_gettime.gr (39/40)
- fuzix_clock_gettime_div10quickm.gr (30/29)
- fuzix_clock_settime_clock_settime.gr (20/21)
- fuzix_devf_fd_transfer.gr (119/129)
- fuzix_devio_bfind.gr (27/29)
- fuzix_devio_kprintf.gr (69/78)
- fuzix_difftime_difftime.gr (74/73)
- fuzix_fgets_fgets.gr (53/58)
- fuzix_filesys_filename.gr (45/48)
- fuzix_filesys_getinode.gr (52/57)
- fuzix_filesys_i_open.gr (129/143)
- fuzix_filesys_newfstab.gr (20/21)
- fuzix_filesys_srch_mt.gr (31/33)
- fuzix_gethostname_gethostname.gr (30/31)
- fuzix_getpass__gets.gr (31/35)
- fuzix_inode_rwsetup.gr (77/83)
- fuzix_malloc___insert_chunk.gr (104/116)
- fuzix_nanosleep_clock_nanosleep.gr (110/121)
- fuzix_process_getproc.gr (32/35)
- fuzix_qsort__lqsort.gr (89/94)
- fuzix_ran_rand.gr (46/48)
- fuzix_readdir_readdir.gr (60/65)
- fuzix_regexp_regcomp.gr (118/129)
- fuzix_se_ycomp.gr (83/96)
- fuzix_setbuffer_setbuffer.gr (43/44)
- fuzix_setenv_setenv.gr (122/131)
- fuzix_stat_statfix.gr (52/51)
- fuzix_syscall_fs2__fchdir.gr (22/22)
- fuzix_syscall_fs2_chown_op.gr (27/28)
- fuzix_syscall_proc__time.gr (48/49)
- fuzix_sysconf_sysconf.gr (142/162)
- fuzix_tty_tty_read.gr (123/137)
- fuzix_usermem_ugets.gr (24/25)
- fuzix_vfscanf_vfscanf.gr (587/668)
- stdlib_gmtime.gr (117/123)
- stdlib_mktime.gr (93/97)
- stdlib_print_format.gr (544/609)
- stdlib_sincoshf.gr (110/117)

The 1813 graphs used in Section 4 come from a wide range of sources: All of the graphs (exact and heuristic) from the PACE challenge, randomly generated partial k-trees, large grids, coloring instances[4], and classical test instances for tree width[5].

B Technical Specifications

All of the experiments were performed on a machine with 64 cores, where each core is a 2.1 Gigahertz processor. Note that we only used a single core for all experiments in order to prevent parallel programs from having an unfair advantage. The machine has 128 Gigabyte RAM and runs openSUSE 13.1 (Bottle) with kernel 3.11.10-29-desktop.

[4] Found at `http://mat.gsia.cmu.edu/COLOR/instances.html`.
[5] Found at `https://github.com/FrankvH/BooleanWidth/tree/master/Graphs/tw-lib`.