# Practical Access to Dynamic Programming on Tree Decompositions

## Max Bannach
Institute for Theoretical Computer Science, Universität zu Lübeck, Lübeck, Germany
bannach@tcs.uni-luebeck.de
 https://orcid.org/0000-0002-6475-5512

## Sebastian Berndt
Department of Computer Science, Kiel University, Kiel, Germany
seb@informatik.uni-kiel.de
 https://orcid.org/0000-0003-4177-8081

### Abstract

Parameterized complexity theory has lead to a wide range of algorithmic breakthroughs within the last decades, but the practicability of these methods for real-world problems is still not well understood. We investigate the practicability of one of the fundamental approaches of this field: dynamic programming on tree decompositions. Indisputably, this is a key technique in parameterized algorithms and modern algorithm design. Despite the enormous impact of this approach in theory, it still has very little influence on practical implementations. The reasons for this phenomenon are manifold. One of them is the simple fact that such an implementation requires a long chain of non-trivial tasks (as computing the decomposition, preparing it,...). We provide an easy way to implement such dynamic programs that only requires the definition of the update rules. With this interface, dynamic programs for various problems, such as 3-COLORING, can be implemented easily in about 100 lines of structured Java code.

The theoretical foundation of the success of dynamic programming on tree decompositions is well understood due to Courcelle's celebrated theorem, which states that every MSO-definable problem can be efficiently solved if a tree decomposition of small width is given. We seek to provide practical access to this theorem as well, by presenting a lightweight model-checker for a small fragment of MSO. This fragment is powerful enough to describe many natural problems, and our model-checker turns out to be very competitive against similar state-of-the-art tools.

## 1 Introduction

Parameterized algorithms aim to solve intractable problems on instances where some parameter tied to the complexity of the instance is small. This line of research has seen enormous growth in the last decades and produced a wide range of algorithms [9]. More formally, a problem is *fixed-parameter tractable* (in FPT), if every instance $I$ can be solved in time $f(\kappa(I)) \cdot \text{poly}(|I|)$ for a computable function $f$, where $\kappa(I)$ is the *parameter* of $I$. While the impact of parameterized complexity to the theory of algorithms and complexity cannot be overstated, its practical component is much less understood. Very recently, the investigation of the practicability of fixed-parameter tractable algorithms for real-world problems has started to become an important subfield (see e.g. [18, 11]). We investigate the practicability of dynamic programming on tree decompositions – one of the most fundamental techniques of parameterized algorithms. A general result explaining the usefulness of tree decompositions was given by Courcelle in [8], who showed that *every* property that can be expressed in monadic second-order logic is fixed-parameter tractable if it is parameterized by tree width.

By combining this result (known as Courcelle's Theorem) with the $f(\text{tw}(G)) \cdot |G|$ algorithm of Bodlaender [7] to compute an optimal tree decomposition in FPT-time, a wide range of graph-theoretic problems is known to be solvable on these tree-like graphs. Unfortunately, both ingredients of this approach are very expensive in practice.

One of the major achievements concerning practical parameterized algorithms was the discovery of a practically fast algorithm for treewidth due to Tamaki [19]. Concerning Courcelle's Theorem, there are currently two contenders concerning efficient implementations of it: D-Flat, an Answer Set Programming (ASP) solver for problems on tree decompositions [1]; and Sequoia, an MSO solver based on model checking games [17]. Both solvers allow to solve very general problems and the corresponding overhead might, thus, be large compared to a straightforward implementation of the dynamic programs for specific problems.

**Our Contributions**  In order to study the practicability of dynamic programs on tree decompositions, we expand our tree decomposition library Jdrasil with an easy to use interface for such programs: The user only needs to specify the *update rules* for the different kind of nodes within the tree decomposition. The remaining work – computing a suitable optimized tree decomposition and performing the actual run of the dynamic program – are done by Jdrasil. This allows users to implement a wide range of algorithms within very few lines of code and, thus, gives the opportunity to test the practicability of these algorithms quickly. This interface is presented in Section 3.

While D-Flat and Sequoia solve very general problems, the experimental results of Section 5 show that naïve implementations of dynamic programs might be much more efficient. In order to balance the generality of MSO solvers and the speed of direct implementations, we introduce a small MSO fragment, that avoids quantifier alternation, in Section 4. By concentrating on this fragment, we are able to build a model-checker, called Jatatosk, that runs nearly as fast as direct implementations of the dynamic programs. To show the feasibility of our approach, we compare the running times of D-Flat, Sequoia, and Jatatosk for various problems. It turns out that Jatatosk is competitive against the other solvers and, furthermore, its behaviour is much more consistent (i.e. it does not fluctuate greatly on similar instances). We conclude that concentrating on just a small fragment of MSO gives rise to practically fast solvers, which are still able to solve a large class of problems on graphs of bounded treewidth.

## 2    Preliminaries

All graphs considered in this paper are undirected, that is, they consists of a set of vertices $V$ and of a symmetric edge-relation $E \subseteq V \times V$. We assume the reader to be familiar with basic graph theoretic terminology, see for instance [10]. A *tree decomposition* of a graph $G = (V, E)$ is a tuple $(T, \iota)$ consisting of a rooted tree $T$ and a mapping $\iota$ from nodes of $T$ to sets of vertices of $G$ (which we call *bags*) such that (1) for all $v \in V$ there is a node $n$ in $T$ with $v \in \iota(n)$, (2) for every edge $\{v, w\} \in E$ there is a node $m$ in $T$ with $\{v, w\} \subseteq \iota(m)$, and (3) the set $\{x \mid v \in \iota(x)\}$ is connected in $T$ for every $v \in V$. The *width* of a tree decomposition is the maximum size of one of its bags minus one, and the *treewidth* of $G$, denoted by $\text{tw}(G)$, is the minimum width any tree decomposition of $G$ must have.

In order to describe dynamic programs over tree decompositions, it turns out be helpful to transform a tree decomposition into a more structured one. A *nice tree decomposition* is a triple $(T, \iota, \eta)$ where $(T, \iota)$ is a tree decomposition and $\eta \colon V(T) \to \{\text{leaf}, \text{introduce}, \text{forget}, \text{join}\}$ is a labeling such that (1) nodes labeled "leaf" are exactly the leaves of $T$, and the bags of

these nodes are empty; (2) nodes $n$ labeled "introduce" or "forget" have exactly one child $m$ such that there is exactly one vertex $v \in V(G)$ with either $v \notin \iota(m)$ and $\iota(n) = \iota(m) \cup \{v\}$ or $v \in \iota(m)$ and $\iota(n) = \iota(m) \setminus \{v\}$, respectively; (3) nodes $n$ labeled "join" have exactly two children $x, y$ with $\iota(n) = \iota(x) = \iota(y)$. A *very nice tree decomposition* is a nice tree decomposition that also has exactly one node labeled "edge" for every $e \in E(G)$, which virtually introduces the edge $e$ to the bag – i.e., whenever we introduce a vertex, we assume it to be "isolated" in the bag until its incident edges are introduced. It is well known that any tree decomposition can efficiently be transformed into a very nice one without increasing its width (essentially traverse through the tree and "pull apart" bags) [9]. Whenever we talk about tree decompositions in the rest of the paper, we actually mean very nice tree decompositions. However, we want to stress out that all our interfaces also support "just" nice tree decompositions.

We assume the reader to be familiar with basic logic terminology and give just a brief overview over the syntax and semantic of monadic second-order logic (MSO), see for instance [13] for a detailed introduction. A *vocabulary* (or *signature*) $\tau = (R_1^{a_1}, \ldots, R_n^{a_n})$ is a set of *relational symbols* $R_i$ of arity $a_i \geq 1$. A $\tau$-*structure* is a set $U$ – called *universe* – together with an *interpretation* $R_i^U \subseteq R^{a_i}$ of the relational symbols. Let $x_1, x_2, \ldots$ be a sequence of *first-order variables* and $X_1, X_2, \ldots$ be a sequence of *second-order variables* $X_i$ of arity $\mathrm{ar}(X_i)$. The atomic $\tau$-formulas are $x_i = x_j$ for two first-order variables and $R(x_{i_1}, \ldots, x_{i_k})$, where $R$ is either a relational symbol or a second-order variable of arity $k$. The set of $\tau$-formulas is inductively defined by (1) the set of atomic $\tau$-formulas; (2) Boolean connections $\neg\phi$, $(\phi \vee \psi)$, and $(\phi \wedge \psi)$ of $\tau$-formulas $\phi$ and $\psi$; (3) quantified formulas $\exists x \phi$ and $\forall x \phi$ for a first-order variable $x$ and a $\tau$-formula $\phi$; (4) quantified formulas $\exists X \phi$ and $\forall X \phi$ for a second-order variable $X$ of arity 1 and a $\tau$-formula $\phi$. The set of *free variables* of a formula $\phi$ consists of the variables that appear in $\phi$ but are not bounded by a quantifier. We denote a formula $\phi$ with free variables $x_1, \ldots, x_k, X_1, \ldots, X_\ell$ as $\phi(x_1, \ldots, x_k, X_1, \ldots, X_\ell)$. Finally, we say a $\tau$-structure $\mathcal{S}$ with an universe $U$ is a *model* of an $\tau$-formula $\phi(x_1, \ldots, x_k, X_1, \ldots, X_\ell)$ if there are elements $u_1, \ldots, u_k \in U$ and relations $U_1, \ldots, U_\ell$ with $U_i \subseteq U^{\mathrm{ar}(X_i)}$ with $\phi(u_1, \ldots, u_k, U_1, \ldots, U_\ell)$ being true in $\mathcal{S}$. We write $\mathcal{S} \models \phi(u_1, \ldots, u_k, U_1, \ldots, U_\ell)$ in this case.

▶ **Example 1.** Graphs can be modeled as $\{E^2\}$-structures with a symmetric interpretation of $E$. Properties such as "is 3-colorable" can then be described by formulas as:

$$\tilde{\phi}_{3\mathrm{col}} = \exists R \exists G \exists B \; (\forall x \; R(x) \vee G(x) \vee B(x)) \wedge (\forall x \forall y \; E(x, y) \rightarrow \bigwedge_{C \in \{R, G, B\}} \neg C(x) \vee \neg C(y)).$$

For instance, we have $\boxtimes \models \tilde{\phi}_{3\mathrm{col}}$ and $\boxtimes \not\models \tilde{\phi}_{3\mathrm{col}}$. We write $\tilde{\phi}$ whenever a more refined version of $\phi$ will be given later on.

The *model-checking* problem asks, given a logical structure $\mathcal{S}$ and a formula $\phi$, if $\mathcal{S} \models \phi$ holds. A *model-checker* is a program that solves this problem and outputs an assignment to its free and bounded variables if $\mathcal{S} \models \phi$ holds.

## 3 An Interface for Dynamic Programming on Tree Decompositions

It will be convenient to recall a classical viewpoint of dynamic programming on tree decompositions to illustrate why our interface is designed the way it is. We will do so by the guiding example of 3-COLORING: Is it possible to color vertices of a given graph with three colors such that adjacent vertices never share the same color? Intuitively, a dynamic program for 3-COLORING will work bottom-up on a very nice tree decomposition and manages a

set of possible colorings per node. Whenever a vertex is introduced, the program "guesses" a color for this vertex; if a vertex is forgotten we have to remove it from the bag and identify configurations that become eventually equal; for join bags we just have to take the configurations that are present in both children; and for edge bags we have to reject colorings in which both endpoints of the introduced edge have the same color. To formalize this vague algorithmic description, we view it from the perspective of automata theory.

## 3.1     The Tree Automaton Perspective

Classically, dynamic programs on tree decompositions are described in terms of tree automata [13]. Recall that in a very nice tree decomposition the tree $T$ is rooted and binary; we assume that the children of $T$ are ordered. The mapping $\iota$ can then be seen as a function that maps the nodes of $T$ to symbols from some alphabet $\Sigma$. A naïve approach to manage $\iota$ would yield a huge alphabet (depending on the size of the graph). We thus define the so called *tree-index*, which is a map $\mathrm{idx}\colon V(G) \to \{0, \ldots, \mathrm{tw}(G)\}$ such that no two vertices that appear in the same bag share a common tree-index. The existence of such an index follows directly from the property that every vertex is forgotten exactly once: We can simply traverse $T$ from the root to the leaves and assign a free index to a vertex $V$ when it is forgotten, and release the used index once we reach an introduce bag for $v$. The symbols of $\Sigma$ then only contain the information for which tree-index there is a vertex in the bag. From a theoreticians perspective this means that $|\Sigma|$ depends only on the treewidth; from a programmers perspective the tree-index makes it much easier to manage data structures that are used by the dynamic program.

▶ **Definition 2** (Tree Automaton). A nondeterministic bottom-up *tree automaton* is a tuple $A = (Q, \Sigma, \Delta, F)$ where $Q$ is a set of *states* with a subset $F \subseteq Q$ of *accepting states*, $\Sigma$ is an *alphabet*, and $\Delta \subseteq (Q \cup \{\bot\}) \times (Q \cup \{\bot\}) \times \Sigma \times Q$ is a *transition relation* in which $\bot \notin Q$ is a special symbol to treat nodes with less than two children. The automaton is *deterministic* if for every $x, y \in Q \cup \{\bot\}$ and every $\sigma \in \Sigma$ there is exactly one $q \in Q$ with $(x, y, \sigma, q) \in \Delta$.

▶ **Definition 3** (Computation of a Tree Automaton). The *computation of a tree automaton* $A = (Q, \Sigma, \Delta, F)$ on a labeled tree $(T, \iota)$ with $\iota\colon V(T) \to \Sigma$ and root $r \in V(T)$ is an assignment $q\colon V(T) \to Q$ such that for all $n \in V(T)$ we have (1) $(q(x), q(y), \iota(n), q(n)) \in \Delta$ if $n$ has two children $x$, $y$; (2) $(q(x), \bot, \iota(n), q(n)) \in \Delta$ or $(\bot, q(x), \iota(n), q(n)) \in \Delta$ if $n$ has one child $x$; (3) $(\bot, \bot, \iota(n), q(n)) \in \Delta$ if $n$ is a leaf. The computation is *accepting* if $q(r) \in F$.

**Simulating Tree Automata**   A dynamic program for a decision problem can be formulated as a nondeterministic tree automaton that works on the decomposition, see the left side of Figure 1 for a detailed example. Observe that a nondeterministic tree automaton $A$ will process a labeled tree $(T, \iota)$ with $n$ nodes in time $O(n)$. When we simulate such an automaton deterministically, one might think that a running time of the form $O(|Q| \cdot n)$ is sufficient, as the automaton could be in any potential subset of the $Q$ states at some node of the tree. However, there is a pitfall: For every node we have to compute the set of potential states of the automaton depending on the sets of potential states of the children of that node, leading to a quadratic dependency on $|Q|$. This can be avoided for transitions of the form $(\bot, \bot, \iota(x), p)$, $(q, \bot, \iota(x), p)$, and $(\bot, q, \iota(x), p)$, as we can collect potential successors of every state of the child and compute the new set of states in linear time with respect to the cardinality of the set. However, transitions of the form $(q_i, q_j, \iota(x), p)$ are difficult, as we now have to merge two sets of states. In detail, let $x$ be a node with children $y$ and $z$ and let $Q_y$ and $Q_z$ be the set of potential states in which the automaton eventually is in at these

nodes. To determine $Q_x$ we have to check for every $q_i \in Q_y$ and every $q_j \in Q_z$ if there is a $p \in Q$ such that $(q_i, q_j, \iota(x), p)$. Note that the number of states $|Q|$ can be quite large even for moderately sized parameters $k$, as $|Q|$ is typically of size $2^{\Omega(k)}$, and we will thus try to avoid this quadratic blow-up.

▶ **Observation 4.** *A tree automaton can be simulated in time $O(|Q|^2 \cdot n)$.*

Unfortunately, the quadratic factor in the simulation cannot be avoided in general, as the automaton may very well contain a transition for all possible pairs of states. However, there are some special cases in which we can circumnavigate the increase in the running time.

▶ **Definition 5** (Symmetric Tree Automaton). A *symmetric* nondeterministic bottom-up tree automaton is a nondeterministic bottom-up tree automaton $A = (Q, \Sigma, \Delta, F)$ in which all transitions $(l, r, \sigma, q) \in \Delta$ satisfy either $l = \bot$, $r = \bot$, or $l = r$.

Assume as before that we wish to compute the set of potential states for a node $x$ with children $y$ and $z$. Observe that in a symmetric tree automaton it is sufficient to consider the set $Q_y \cap Q_z$ and that the intersection of two sets can be computed in linear time if we take some care in the design of the underlying data structures.

▶ **Observation 6.** *A symmetric tree automaton can be simulated in time $O(|Q| \cdot n)$.*

The right side of Figure 1 illustrates the deterministic simulation of a symmetric tree automaton. The massive time difference in the simulation of tree automata and symmetric tree automata significantly influenced the design of the algorithms in Section 4, in which we try to construct an automaton that is 1) "as symmetric as possible" and 2) allows to take advantage of the "symmetric parts" even if the automaton is not completely symmetric.

## 3.2 The Interface

We introduce a simple Java-interface to our library Jdrasil, which originally was developed for the computation of tree decompositions only. The interface is build up from two classes: `StateVectorFactory` and `StateVector`. The only job of the factory is to generate `StateVector` objects for the leaves of the tree decomposition, or with the terms of the previous section: "to define the initial states of the tree automaton". The `StateVector` class is meant to model a vector of potential states in which the nondeterministic tree automaton is at a specific node of the tree decomposition. Our interface does not define at all what a "state" is, or how a collection of states is managed (although most of the times, it will be a set). The only thing the interface requests a user to implement is the behaviour of the tree automaton when it reaches a node of the tree-decomposition, i.e., given a `StateVector` (for some unknown node in the tree decomposition) and the information that the automaton reaches a certain node, how does the `StateVector` for this node look like? To this end, the interface contains the methods shown in Listing 1.

■ **Listing 1** The four methods of the interface describe the behaviour of the tree automaton. Here "T" is a generic type for vertices. Each function obtains as parameter the current bag and a tree-index "idx". Other parameters correspond to bag-type specifics, e.g. the introduced or forgotten vertex $v$.

```
StateVector<T> introduce(Bag<T> b, T v, Map<T, Integer> idx);
StateVector<T> forget(Bag<T> b, T v, Map<T, Integer> idx);
StateVector<T> join(Bag<T> b, StateVector<T> o, Map<T, Integer> idx);
StateVector<T> edge(Bag<T> b, T v, T w, Map<T, Integer> idx);
```
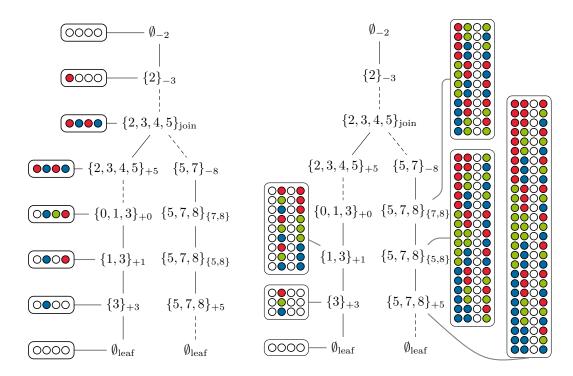
**Figure 1** The *left* picture shows a part of a tree decomposition of the grid graph with vertices $\{0, \ldots, 9\}$. The index of a bag shows the type of the bag: a positive sign means "introduce", a negative one "forget", a pair represents an "edge"-bag, and text is self explanatory. Solid lines represent real edges of the decomposition, while dashed lines illustrate a path (i.e., there are some bags skipped). On the left branch of the decomposition a run of a nondeterministic tree automaton with tree-index $\left(\begin{smallmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 3 & 0 & 1 & 2 & 3 & 0 & 1 & 0 \end{smallmatrix}\right)$ for 3-COLORING is illustrated. To increase readability, states of the automaton are connected to the corresponding bags with gray lines, and for some nodes the states are omitted. In the *right* picture the same automaton is simulated deterministically.

This already rounds up the description of the interface, everything else is done by Jdrasil. In detail, given a graph and an implementation of the interface, Jdrasil will compute a tree decomposition[1], transform this decomposition into a very nice tree decomposition, potentially optimize the tree decomposition for the following dynamic program, and finally traverse through the tree decomposition and simulate the tree automaton described by the implementation of the interface. The result of this procedure is the `StateVector` object assigned to the root of the tree decomposition.

## 3.3 Example: 3-Coloring

Let us illustrate the usage of the interface with our running example of 3-COLORING. A `State` of the automaton can be modeled as a simple integer array that stores a color (an integer) for every vertex in the bag. A `StateVector` stores a set of `State` objects, i.e., essentially a set of integer arrays. Introducing a vertex $v$ to a `StateVector` therefore means that three duplicates of each stored state have to be created, and for every duplicate a different color has to be assigned to $v$. Listing 2 illustrates how this operation could be realized in Java.

---

[1] See [6] for the concrete algorithms used by Jdrasil.

■ **Listing 2** Exemplary implementation of the `introduce` method for 3-COLORING.

```
StateVector<T> introduce(Bag<T> b, T v, Map<T, Integer> idx) {
  Set<State> newStates = new HashSet<>();
  for (State state : states) { // 'states' is the set of states
    for (int color = 1; color <= 3; color++) {
      State newState = new State(state); // copy the state
      newState.colors[idx.get(v)] = color;
      newStates.add(newState);
    }
  }
  states = newStates;
  return this;
}
```

The three other methods can be implemented in a very similar fashion: in the `forget`-method we set the color of $v$ to 0; in the `edge`-method we remove states in which both endpoints of the edge have the same color; and in the `join`-method we compute the intersection of the state sets of both `StateVector` objects. Note that when we forget a vertex $v$, multiple states may become identical, which is handled here by the implementation of the Java `Set`-class, which takes care of duplicates automatically.

A reference implementation of this 3-COLORING solver is publicly available [4], and a detailed description of it can be found in the manual of Jdrasil [5]. Note that this implementation is only meant to illustrate the interface and that we did not make any effort to optimize it. Nevertheless, this very simple implementation (the part of the program that is responsible for the dynamic program only contains about 120 lines of structured Java-code) performs surprisingly well, as the experiments in Section 5 indicate.

## 4  A Lightweight Model-Checker for a Small MSO-Fragment

Experiments with the coloring solver of the previous section have shown a huge difference in the performance of general solvers as D-Flat and Sequoia against a concrete implementation of a tree automaton for a specific problem (see Section 5). This is not necessarily surprising, as a general solver needs to keep track of way more information. In fact, a MSO-model-checker can probably (unless P = NP) not run in time $f(|\phi| + \text{tw}) \cdot \text{poly}(n)$ for any elementary function $f$ [14]. On the other hand, it is not clear (in general) what the concrete running time of such a solver is for a concrete formula or problem (see e.g. [16] for a sophisticated analysis of some running times in Sequoia). We seek to close this gap between (slow) general solvers and (fast) concrete algorithms. Our approach is to concentrate only on a small fragment of MSO, which is powerful enough to express many natural problems, but which is restricted enough to allow model-checking in time that matches or is close to the running time of a concrete algorithm for the problem. As a bonus, we will be able to derive upper bounds on the running time of the model-checker directly from the syntax of the input formula.

Based on the interface of Jdrasil, we have implemented a publicly available prototype called Jatatosk [3]. In Section 5, we perform various experiments on different problems on multiple sets of graphs. It turns out that Jatatosk is competitive against the state-of-the-art solvers D-Flat and Sequoia. Arguably these two programs solve a more general problem and a direct comparison is not entirely fair. However, the experiments do reveal that it seems very promising to focus on smaller fragments of MSO (or perhaps any other description language) in the design of treewidth based solvers.

## 4.1 Description of the Fragment

We only consider vocabularies $\tau$ that contain the binary relation $E^2$, and we only consider $\tau$-structures with a symmetric interpretation of $E^2$, i.e., we only consider structures that contain an undirected graph (but may also contain further relations). The fragment of MSO that we consider is constituted by formulas of the form $\phi = \exists X_1 \ldots \exists X_k \bigwedge_{i=1}^n \psi_i$, where the $X_j$ are second-order variables and the $\psi_i$ are first-order formulas of the form

$$\psi_i \in \{\, \forall x \forall y \; E(x, y) \to \chi_i, \; \forall x \exists y \; E(x, y) \wedge \chi_i, \; \exists x \forall y \; E(x, y) \to \chi_i,$$
$$\exists x \exists y \; E(x, y) \wedge \chi_i, \; \forall x \; \chi_i, \; \exists x \; \chi_i \,\}.$$

Here, the $\chi_i$ are quantifier-free first-order formulas in canonical normal form. It is easy to see that this fragment is already powerful enough to encode many classical problems as 3-COLORING ($\tilde{\phi}_{3\text{col}}$ from the introduction is part of the fragment), or VERTEX-COVER (we will discuss how to handle optimization in Section 4.4): $\tilde{\phi}_{\text{vc}} = \exists S \forall x \forall y \; E(x, y) \to S(x) \vee S(y)$.

## 4.2 A Syntactic Extension of the Fragment

Many interesting properties, such as connectivity, can easily be expressed in MSO, but not directly in the fragment that we study. Nevertheless, a lot of these properties can directly be checked by a model-checker if it "knows" what kind of properties it actually checks. We present a *syntactic extension* of our MSO-fragment which captures such properties. The extension consist of three new second order quantifiers that can be used instead of $\exists X_i$.

The first extension is a *partition quantifier*, which quantifies over partitions of the universe:

$$\exists^{\text{partition}} X_1, \ldots, X_k \equiv \exists X_1 \exists X_2 \ldots \exists X_k \big( \forall x \bigvee_{i=1}^k X_i(x) \big) \wedge \big( \forall x \bigwedge_{i=1}^k \bigwedge_{j \neq i} \neg X_i(x) \wedge \neg X_j(x) \big).$$

This quantifier has two advantages. First, formulas like $\tilde{\phi}_{3\text{col}}$ can be simplified to

$$\phi_{3\text{col}} = \exists^{\text{partition}} R, G, B \; \forall x \forall y \; E(x, y) \to \bigwedge_{C \in \{R, G, B\}} \neg C(x) \vee \neg C(y),$$

and second, the model-checking problem for them can be solved more efficiently: the solver directly "knows" that a vertex must be added to exactly one of the sets.

We further introduce two quantifiers that work with respect to the symmetric relation $E^2$ (recall that we only consider structures that contain such a relation). The $\exists^{\text{connected}} X$ quantifier guesses an $X \subseteq U$ that is connected with respect to $E$ (in graph theoretic terms), i.e., it quantifies over connected subgraphs. The $\exists^{\text{forest}} F$ quantifier guesses a $F \subseteq U$ that is acyclic with respect to $E$ (again in graph theoretic terms), i.e., it quantifies over subgraphs that are forests. These quantifiers are quite powerful and allow, for instance, to express that the graph induced by $E^2$ contains a triangle as minor:

$$\phi_{\text{triangle-minor}} = \exists^{\text{connected}} R \, \exists^{\text{connected}} G \, \exists^{\text{connected}} B \, .$$
$$\big( \forall x \, (\neg R(x) \vee \neg G(x)) \wedge (\neg G(x) \vee \neg B(x)) \wedge (\neg B(x) \vee \neg R(x)) \big)$$
$$\wedge \big( \exists x \exists y \; E(x, y) \wedge R(x) \wedge G(y) \big) \wedge \big( \exists x \exists y \; E(x, y) \wedge G(x) \wedge B(y) \big)$$
$$\wedge \big( \exists x \exists y \; E(x, y) \wedge B(x) \wedge R(y) \big).$$

We can also express problems that usually require more involved formulas in a very natural way. For instance, the FEEDBACK-VERTEX-SET problem can be described by the following formula (again, optimization will be handled in Section 4.4): $\tilde{\phi}_{\text{fvs}} = \exists S \, \exists^{\text{forest}} F \, \forall x \; S(x) \vee F(x)$.

## 4.3 Description of the Model-Checker

We describe our model-checker in terms of a nondeterministic tree automaton that works on a tree decomposition of the graph induced by $E^2$ (note that, in contrast to other approaches in the literature, we do not work on the Gaifman graph). We define any state of the automaton as bit-vector, and we stipulate that the initial state at every leaf is the zero-vector. For any quantifier or subformula, there will be some area in the bit-vector reserved for that quantifier or subformula and we describe how state transitions effect these bits. The "algorithmic idea" behind the implementation of these transitions is not new, and a reader familiar with folklore dynamic programs on tree decompositions (for instance for VERTEX-COVER or STEINER-TREE) will probably recognize them. An overview over common techniques can be found in the standard textbooks [9, 13].

**The Partition Quantifier**   We start with a detailed description of the partition quantifier $\exists^{\mathrm{partition}} X_1, \ldots, X_q$ (we do not implement an additional $\exists X$ quantifier, as we can easily state $\exists X \equiv \exists^{\mathrm{partition}} X, \bar{X}$): Let $k$ be the maximum bag-size of the tree decomposition. We reserve $k \cdot \log_2 q$ bit in the state description, where each block of length $\log_2 q$ indicates in which set $X_i$ the corresponding element of the bag is. On an introduce-bag (e.g. for $v \in U$), the nondeterministic automaton guesses an index $i \in \{1, \ldots, q\}$ and sets the $\log_2 q$ bits that are associated with the tree-index of $v$ to $i$. Equivalently, the corresponding bits are cleared when the automaton reaches a forget-bag. As the partition is independent of any edges, an edge-bag does not change any of the bits reserved for the partition quantifier. Finally, on join-bags we may only join states that are identical on the bits describing the partition (as otherwise the vertices of the bag would be in different partitions) – meaning this transition is symmetric with respect to these bits (in terms of Section 3.1).

**The Connected Quantifier**   The next quantifier we describe is $\exists^{\mathrm{connected}} X$ which has to overcome the difficulty that an introduced vertex may not be connected to the rest of the bag in the moment it got introduced, but may be connected to it when further vertices "arrive". The solution to this dilemma is to manage a partition of the bag into $k' \leq k$ connected components $P_1, \ldots, P_{k'}$, for which we reserve $k \cdot \log_2 k$ bit in the state description. Whenever a vertex $v$ is introduced, the automaton either guesses that it is not contained in $X$ and clears the corresponding bits, or it guesses that $v \in X$ and assigns some $P_i$ to $v$. Since $v$ is isolated in the bag in the moment of its introduction (recall that we work on a very nice tree decomposition), it requires its own component and is therefore assigned to the smallest empty partition $P_i$. When a vertex $v$ is forgotten, there are four possible scenarios: 1) $v \notin X$, then the corresponding bits are already cleared and nothing happens; 2) $v \in X$ and $v \in P_i$ with $|P_i| > 1$, then $v$ is just removed and the corresponding bits are cleared; 3) $v \in X$ and $v \in P_i$ with $|P_i| = 1$ and there are other vertices $w$ in the bag with $w \in X$, then the automaton rejects the configuration, as $v$ is the last vertex of $P_i$ and may not be connected to any other partition anymore; 4) $v \in X$ is the last vertex of the bag that is contained in $X$, then the connected component is "done", the corresponding bits are cleared and one additional bit is set to indicate that the connected component cannot be extended anymore. When an edge $\{u, v\}$ is introduced, components might need to be merged. Assume $u, v \in X$, $u \in P_i$, and $v \in P_j$ with $i < j$ (otherwise, an edge-bag does not change the state), then we essentially perform a classical union-operation from the well-known union-find data structure. Hence, we assign all vertices that are assigned to $P_j$ to $P_i$. Finally, at a join-bag we may join two states that agree locally on the vertices that are in $X$ (i.e., they have assigned the same vertices to some $P_i$), however, they do not have to agree in the way the different vertices are assigned to

$P_i$ (in fact, there does not have to be an isomorphism between these assignments). Therefore, the transition at a join-bag has to connect the corresponding components analogous to the edge-bags – in terms of Section 3.1 this transition is not symmetric. The description of the remaining quantifiers and subformulas is very similar and presented in Appendix A.

## 4.4 Extending the Model-Checker to Optimization Problems

As the example formulas from the previous section already indicate, performing model-checking alone will not suffice to express many natural problems. In fact, every graph is a model of the formula $\tilde{\phi}_{\mathrm{vc}}$ if $S$ simply contains all vertices. It is therefore a natural extension to consider an optimization version of the model-checking problem, which is usually formulated as follows [9, 13]: Given a logical structure $\mathcal{S}$, a formula $\phi(X_1, \ldots, X_p)$ of the MSO-fragment defined in the previous section with free unary second-order variables $X_1, \ldots, X_p$, and weight functions $\omega_1, \ldots, \omega_p$ with $\omega_i \colon U \to \mathbb{Z}$; find $S_1, \ldots, S_p$ with $S_i \subseteq U$ such that $\sum_{i=1}^{p} \sum_{s \in S_i} \omega_i(s)$ is *minimized* under $\mathcal{S} \models \phi(S_1, \ldots, S_p)$, or conclude that $\mathcal{S}$ is not a model for $\phi$ for any assignment of the free variables. We can now correctly express the (actually *weighted*) optimization version of VERTEX-COVER as follows: $\phi_{\mathrm{vc}}(S) = \forall x \forall y\ E(x,y) \to \big(S(x) \vee S(y)\big)$. Similarly we can describe the optimization version of DOMINATING-SET if we assume the input does not have isolated vertices (or is reflexive), and we can also fix the formula $\tilde{\phi}_{\mathrm{fvs}}$:

$$\phi_{\mathrm{ds}}(S) = \forall x \exists y\ E(x,y) \wedge \big(S(x) \vee S(y)\big), \quad \phi_{\mathrm{fvs}}(S) = \exists^{\mathrm{forest}} F\ \forall x\ \big(S(x) \vee F(x)\big).$$

We can also *maximize* the term $\sum_{i=1}^{p} \sum_{s \in S_i} \omega_i(s)$ by multiplying all weights with $-1$ and, thus, express problems such as INDEPENDENT-SET: $\phi_{\mathrm{is}}(S) = \forall x \forall y\ E(x,y) \to \big(\neg S(x) \vee \neg S(y)\big)$. The implementation of such an optimization is straightforward: essentially there is a partition quantifier for every free variable $X_i$ that partitions the universe into $X_i$ and $\bar{X}_i$. We assign a current value of $\sum_{i=1}^{p} \sum_{s \in S_i} \omega_i(s)$ to every state of the automaton, which is adapted if elements are "added" to some of the free variables at introduce nodes. Note that, since we optimize an affine function, this does not increase the state space: even if multiple computational paths lead to the same state with different values at some node of the tree, it is well defined which of these values is the optimal one. Therefore, the cost of optimization only lies in the partition quantifier, i. e., we pay with $k$ bits in the state description of the automaton per free variable – independently of the weights.

## 4.5 Handling Symmetric and Non-Symmetric Joins

In Section 4.3 we have defined the states of our automaton with respect to a formula, the left side of Table 1 gives an overview of the number of bits we require for the different parts of the formula. Let $\mathrm{bit}(\phi, k)$ be the number of bits that we have to reserve for a formula $\phi$ and a tree decomposition of maximum bag size $k$, i. e., the sum over the required bits of each part of the formula. By Observation 4 this implies that we can simulate the automaton (and hence, solve the model-checking problem) in time $O^*\big((2^{\mathrm{bit}(\phi,k)})^2 \cdot n\big)$; or by Observation 6 in time $O^*\big(2^{\mathrm{bit}(\phi,k)} \cdot n\big)$ if the automaton is symmetric[2]. Unfortunately, this is not always the case, in fact, only the quantifier $\exists^{\mathrm{partition}} X_1, \ldots, X_q$, the bits needed to optimize over free variables, as well as the formulas that do not require any bits, yield an symmetric tree automaton. That means that the simulation is wasteful if we consider a mixed formula (for instance, one that contains a partition and a connected quantifier). To overcome this issue, we partition

---

[2] The notation $O^*$ supresses polynomial factors.

■ **Table 1** The left table shows the precise number of bit we reserve in the description of a state of the tree automaton for different quantifier and formulas. The values are with respect to a tree decomposition with maximum bag size $k$. The right table gives an overview of example formulas $\phi$ used here, together with values symmetric$(\phi, k)$ and asymmetric$(\phi, k)$, as well as the precise time our algorithm will require for that particular formula.

| Quantifier / Formula | Number of Bit | | $\phi$ | symmetric$(\phi, k)$ / asymmetric$(\phi, k)$ | Time |
|---|---|---|---|---|---|
| free variables $X_1, \ldots, X_q$ | $q \cdot k$ | | $\phi_{3\text{col}}$ | $k \cdot \log_2(3)$ $0$ | $O^*(3^k)$ |
| $\exists^{\text{partition}} X_1, \ldots, X_q$ | $k \cdot \log_2 q$ | | | | |
| $\exists^{\text{connected}} X$ | $k \cdot \log_2 k + 1$ | | $\phi_{\text{vc}}(S)$ | $k$ $0$ | $O^*(2^k)$ |
| $\exists^{\text{forest}} X$ | $k \cdot \log_2 k$ | | | | |
| $\forall x \forall y\ E(x, y) \rightarrow \chi_i$ | $0$ | | $\phi_{\text{ds}}(S)$ | $k$ $k$ | $O^*(8^k)$ |
| $\forall x \exists y\ E(x, y) \wedge \chi_i$ | $k$ | | | | |
| $\exists x \forall y\ E(x, y) \rightarrow \chi_i$ | $k + 1$ | | $\phi_{\text{triangle-minor}}$ | $0$ $3k \cdot \log_2(k) + 3$ | $O^*(k^{6k+6})$ |
| $\exists x \exists y\ E(x, y) \wedge \chi_i$ | $1$ | | | | |
| $\forall x\ \chi_i$ | $0$ | | $\phi_{\text{fvs}}(S)$ | $k$ $k \cdot \log_2(k)$ | $O^*(2^k k^{2k})$ |
| $\exists x\ \chi_i$ | $1$ | | | | |

the bits of the state description into two parts: first the "symmetric" bits of the quantifiers $\exists^{\text{partition}} X_1, \ldots, X_q$ and the bits required for optimization, and in the "asymmetric" ones of all other elements of the formula. Let symmetric$(\phi, k)$ and asymmetric$(\phi, k)$ be defined analogously to bit$(\phi, k)$. We implement the join of states as in the following lemma, allowing us to deduce the running time of the model-checker for concrete formulas. The right side of Table 1 provides an overview for formulas presented here.

▶ **Lemma 7.** *Let $x$ be a node of $T$ with children $y$ and $z$, and let $Q_y$ and $Q_z$ be sets of states in which the automaton may be at $y$ and $z$. Then the set $Q_x$ of states in which the automaton may be at node $x$ can be computed in time $O^*\big(2^{\text{symmetric}(\phi, k) + 2 \cdot \text{asymmetric}(\phi, k)}\big)$.*

**Proof.** To compute $Q_x$, we first split $Q_y$ into $B_1, \ldots, B_q$ such that all elements in one $B_i$ share the same "symmetric bits". This can be done in time $|Q_y|$ using bucket-sort. Note that we have $q \leq 2^{\text{symmetric}(\phi, k)}$ and $|B_i| \leq 2^{\text{asymmetric}(\phi, k)}$. With the same technique we identify for every elements $v$ in $Q_z$ its corresponding partition $B_i$. Finally, we compare $v$ with the elements in $B_i$ to identify those for which there is a transition in the automaton. This yields a running time of $|Q_z| \cdot \max_{i=1}^{q} |B_i| \leq 2^{\text{bit}(\phi, k)} \cdot 2^{\text{asymmetric}(\phi, k)} = 2^{\text{symmetric}(\phi, k) + 2 \cdot \text{asymmetric}(\phi, k)}$. ◀
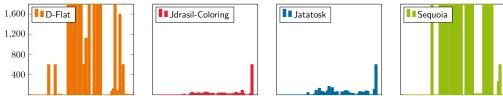
## 5 Applications and Experiments

In order to show the feasibility of our approach, we have performed experiments for widely investigated graph problems: 3-COLORING, VERTEX-COVER, DOMINATING-SET, INDEPENDENT-SET, and FEEDBACK-VERTEX-SET. All experiments where performed on an Intel Core processor containing four cores of 3.2 GHz each and 8 Gigabyte RAM. The machine runs Ubuntu 17.10. Jdrasil was used with Java 1.8 and both Sequoia and D-Flat were compiled with gcc 7.2. The implementation of Jatatosk uses hashing to realize Lemma 7, which has no constant-time worst case guarantee but works well in practice. We use a data set that was assembled from three different sources and that contains graphs with 18 to 956 vertices and treewidth 3 to 13. The first source is a collection of publicly available transit graphs from GTFS-transit feeds [15] that was also used for experiments in [12], the second source are
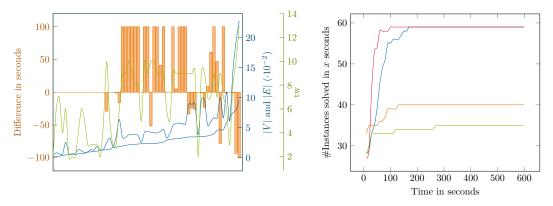
■ **Experiment 1** 3-COLORING

**(a)** Average, standard deviation, and median of the time (in seconds) each solver needed to solve 3-COLORING over all instances of the data set. The best values are highlighted.

|                    | D-Flat | Jdrasil-Coloring | Jatatosk | Sequoia |
|--------------------|--------|------------------|----------|---------|
| Average Time       | 478.19 | **36.52**        | 42.63    | 714.73  |
| Standard Deviation | 733.90 | **77.8**         | 81.82    | 866.34  |
| Median Time        | **3.5** | 21              | 24.5     | 20.5    |

**(b)** Comparison of solvers for the 3-COLORING problem on the complete data set.



**(c)** The left picture shows the difference of Jatatosk against D-Flat and Sequoia. A positive bar means that Jatatosk is faster by this amount in seconds, and a negative bar means that either D-Flat or Sequoia is faster by that amount. The bars are capped at 100. On every instance, Jatatosk was compared against the solver that was faster on this particular instance. The image also shows for every instance the size and the treewidth of the input. The right image shows the number of instances that can be solved by each of the solvers in $x$ seconds, i.e., faster growing functions are better. The colors in this image are as in **(b)**.



real-world instances collected in [2], and the last one are the publicly available graphs used in the PACE challenge [18] (we selected the ones with treewidth at most 11). For 3-COLORING the results can be found in Experiment 1, and for the other problems in Appendix B.

## 6 Conclusion and Outlook

We investigated the practicability of dynamic programming on tree decompositions, which is arguably one of the corner stones of parameterized complexity theory. We implemented a simple interface for such programs and demonstrated how it can be used to build a competitive graph coloring solver with just a few lines of code. We hope that the interface allows other researchers to implement and explore various dynamic programs on tree decompositions. The whole power of such dynamic programs is well captured by Courcelle's Theorem, which essentially states that there is an efficient version of such a program for every problem definable in monadic second-order logic. We took a step towards practice here as well, by

implementing a "lightweight" version in the form of a model-checker for a small fragment of the logic. By clever syntactic extensions, the fragment turns out to be powerful enough to express many natural problems such as 3-COLORING, FEEDBACK-VERTEX-SET, and more.

────── **References** ──────

**1** Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. D-flat: progress report. *DBAI, TU Wien, Tech. Rep. DBAI-TR-2014–86*, 2014.

**2** Michael Abseher, Frederico Dusberger, Nysret Musliu, and Stefan Woltran. Improving the efficiency of dynamic programming on tree decompositions via machine learning. In *Proc. IJCAI*, pages 275–282, 2015.

**3** M. Bannach. Jatatosk. `https://github.com/maxbannach/Jatatosk`, 2018. [Online; accessed 22-04-2018].

**4** M. Bannach. Jdrasil for Graph Coloring. `https://github.com/maxbannach/Jdrasil-for-GraphColoring`, 2018. [Online; accessed 22-04-2018].

**5** M. Bannach, S. Berndt, and T. Ehlers. Jdrasil. `https://github.com/maxbannach/Jdrasil`, 2017. [Online; accessed 22-04-2018].

**6** Max Bannach, Sebastian Berndt, and Thorsten Ehlers. Jdrasil: A modular library for computing tree decompositions. In *16th International Symposium on Experimental Algorithms, SEA 2017, June 21-23, 2017, London, UK*, pages 28:1–28:21, 2017. `doi:10.4230/LIPIcs.SEA.2017.28`.

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

**8** Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and computation*, 85(1):12–75, 1990.

**9** Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. `doi:10.1007/978-3-319-21275-3`.

**10** Reinhard Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

**11** M. R. Fellows. Parameterized complexity for practical computing. `http://www.mrfellows.net/wordpress/wp-content/uploads/2017/11/FellowsToppforsk2017.pdf`, 2018. [Online; accessed 22-04-2018].

**12** Johannes Klaus Fichte, Neha Lodha, and Stefan Szeider. Sat-based local improvement for finding tree decompositions of small width. In *Theory and Applications of Satisfiability Testing - SAT*, pages 401–411, 2017.

**13** J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, 2006. `doi:10.1007/3-540-29953-X`.

**14** Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. *Annals of pure and applied logic*, 130(1-3):3–31, 2004.

**15** gtfs2graphs - A Transit Feed to Graph Format Converter. `https://github.com/daajoe/gtfs2graphs`. Accessed: 2018-04-20.

**16** Joachim Kneis, Alexander Langer, and Peter Rossmanith. Courcelle's theorem—a game-theoretic approach. *Discrete Optimization*, 8(4):568 – 594, 2011. URL: `http://www.sciencedirect.com/science/article/pii/S157252861100034X`, `doi:https://doi.org/10.1016/j.disopt.2011.06.001`.

**17** Alexander Langer. *Fast algorithms for decomposable graphs*. PhD thesis, RWTH Aachen, 2013.

**18** The Parameterized Algorithms and Computational Experiments Challenge (PACE). `https://pacechallenge.wordpress.com/`. Accessed: 2018-04-20.

**19** Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. In *Proc. ESA*, pages 68:1–68:13, 2017.

## A    Technical Appendix: Description of the Fragment

---

$\exists^{\text{forest}} X$

| | |
|---|---|
| #Bit: | $k \cdot \log_2 k$ |
| Introduce: | As for $\exists^{\text{connected}} X$. |
| Forget: | Just clear the corresponding bits. |
| Edge: | As for $\exists^{\text{connected}} X$, but reject if two vertices of the same component are connected. |
| Join: | As for $\exists^{\text{connected}} X$, but track if the join introduces a cycle. |

---

$\forall x \forall y\ E(x,y) \rightarrow \chi_i$

| | |
|---|---|
| #Bit: | 0 |
| Introduce: | - |
| Forget: | - |
| Edge: | Reject if $\chi_i$ is not satisfied for the vertices of the edge. |
| Join: | - |

---

$\forall x \exists y\ E(x,y) \wedge \chi_i$

| | |
|---|---|
| #Bit: | $k$ |
| Introduce: | - |
| Forget: | Reject if the bit corresponding to $v$ is not set. |
| Edge: | Set the bit of $v$ if $\chi_i$ is satisfied. |
| Join: | Compute the logical-or of the bits of both states. |

---

$\exists x \forall y\ E(x,y) \rightarrow \chi_i$

| | |
|---|---|
| #Bit: | $k + 1$ |
| Introduce: | Set the corresponding bit. |
| Forget: | If the corresponding bit is set, set the additional bit. |
| Edge: | If $\chi_i$ is not satisfied, clear the corresponding bit. |
| Join: | Compute the logical-and of all but the last bit, for the last bit use a logical-or. |

---

$\exists x \exists y\ E(x,y) \wedge \chi_i$

| | |
|---|---|
| #Bit: | 1 |
| Introduce: | - |
| Forget: | - |
| Edge: | Set the bit if $\chi_i$ is satisfied. |
| Join: | Compute logical-or of the bit in both states. |

---

$\forall x\ \chi_i\ (\exists x\ \chi_i)$

| | |
|---|---|
| #Bit: | 0 (1) |
| Introduce: | Test if $\chi_i$ is satisfied and reject if not (set the bit if so). |
| Forget: | - |
| Edge: | - |
| Join: | - (Compute logical-or of the bit in both states.) |

## B    Technical Appendix: Further Experiments
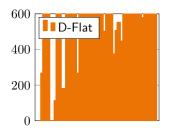
We perform the experiments from Section 5 for further problems. We did run every solver for a maximum of 600 seconds on every instance for every problem. It can be seen that Jatatosk is competitive (though not superior) against its competitors, as it is faster then the faster of the two on many instances and its average running time is at most twice the time of the corresponding fastest algorithm. Jatatosk outperforms the others for 3-COLORING, but gets outperformed for VERTEX-COVER by Sequoia. The same holds for INDEPENDENT-SET, also the difference is much smaller in this case. For DOMINATING-SET the situation is more complex, as Jatatosk outperforms the others on about half of the instances, and gets outperformed on the other half. Interestingly, the difference is quit high in both halves in both directions.
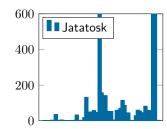
**Experiment 2** VERTEX-COVER
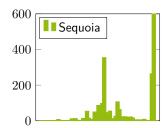
**(a)** Average, standard deviation, and median of the time (in seconds) each solver needed to solve VERTEX-COVER over all instances of the data set from Section 5. The best values are highlighted.

|                    | D-Flat | Jatatosk | Sequoia |
|--------------------|--------|----------|---------|
| Average Time       | 451.68 | 59.02    | **33.95** |
| Standard Deviation | 213.08 | 128.45   | **92.45** |
| Median Time        | 597.5  | 30       | **6**   |
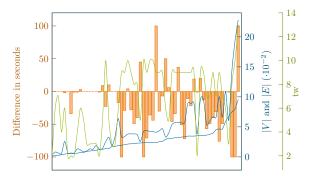
**(b)** Comparison of solvers for the VERTEX-COVER problem on the complete data set of Section 5. The time is measured in seconds and the instances are sorted by the number of vertices, see also (c).
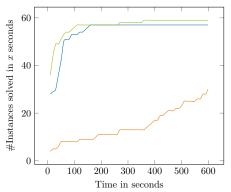


**(c)** The left picture shows the difference of Jatatosk against D-Flat and Sequoia. A positive bar means that Jatatosk is faster by this amount in seconds, and a negative bar means that either D-Flat or Sequoia is faster by that amount. The bars are capped at 100. On every instance, Jatatosk was compared against the solver that was faster on this particular instance. The image also shows for every instance the size and the treewidth of the input. The right image shows the number of instances that can be solved by each of the solvers in $x$ seconds, i.e., faster growing functions are better. The colors in this image are as in (b).
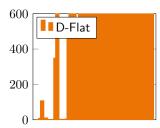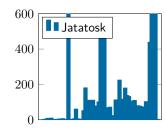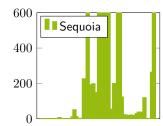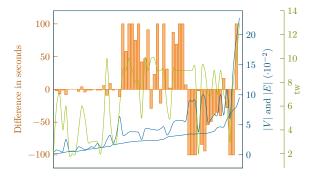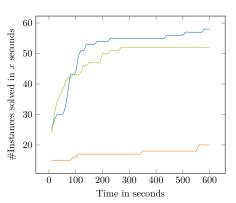
■ **Experiment 3** DOMINATING-SET

**(a)** Average, standard deviation, and median of the time (in seconds) each solver needed to solve DOMINATING-SET over all instances of the data set from Section 5. The best values are highlighted.

|  | D-Flat | Jatatosk | Sequoia |
|---|---|---|---|
| Average Time | 420.14 | **102.48** | 114.92 |
| Standard Deviation | 265.14 | **157.80** | 196.67 |
| Median Time | 600 | 44.5 | **20.5** |

**(b)** Comparison of solvers for the DOMINATING-SET problem on the complete data set of Section 5. The time is measured in seconds and the instances are sorted by the number of vertices, see also (c).



**(c)** The left picture shows the difference of Jatatosk against D-Flat and Sequoia. A positive bar means that Jatatosk is faster by this amount in seconds, and a negative bar means that either D-Flat or Sequoia is faster by that amount. The bars are capped at 100. On every instance, Jatatosk was compared against the solver that was faster on this particular instance. The image also shows for every instance the size and the treewidth of the input. The right image shows the number of instances that can be solved by each of the solvers in $x$ seconds, i. e., faster growing functions are better. The colors in this image are as in (b).
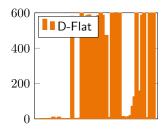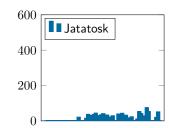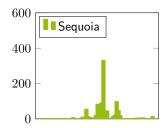
■ **Experiment 4** INDEPENDENT-SET

**(a)** Average, standard deviation, and median of the time (in seconds) each solver needed to solve INDEPENDENT-SET over all instances of the data set from Section 5. The best values are highlighted.

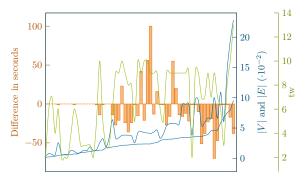|                    | D-Flat | Jatatosk | Sequoia |
|--------------------|--------|----------|---------|
| Average Time       | 229.18 | 16.98    | **15.32** |
| Standard Deviation | 272.64 | **18.17** | 45.53   |
| Median Time        | 13     | 14       | **1**   |

**(b)** Comparison of solvers for the INDEPENDENT-SET problem on the complete data set of Section 5. The time is measured in seconds and the instances are sorted by the number of vertices, see also (c).
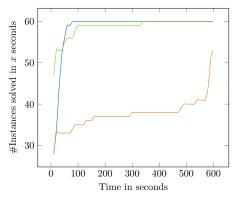


**(c)** The left picture shows the difference of Jatatosk against D-Flat and Sequoia. A positive bar means that Jatatosk is faster by this amount in seconds, and a negative bar means that either D-Flat or Sequoia is faster by that amount. The bars are capped at 100. On every instance, Jatatosk was compared against the solver that was faster on this particular instance. The image also shows for every instance the size and the treewidth of the input. The right image shows the number of instances that can be solved by each of the solvers in $x$ seconds, i. e., faster growing functions are better. The colors in this image are as in (b).
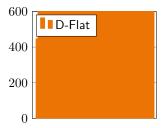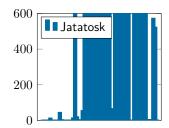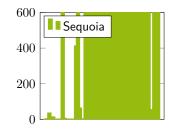
**Experiment 5** FEEDBACK-VERTEX-SET

**(a)** Average, standard deviation, and median of the time (in seconds) each solver needed to solve
FEEDBACK-VERTEX-SET over all instances of the data set from Section 5. The best values are
highlighted. To solve the problem in Sequoia, we used the well-known fact that $F \subseteq V(G)$ is a
feedback vertex set if for all non-empty subsets $W \subseteq V(G) \setminus F$, the induced graph $G[W]$ contains a
vertex of degree at most one.

|                    | D-Flat | Jatatosk | Sequoia |
|--------------------|--------|----------|---------|
| Average Time       | 587.84 | **303.6** | 384.16  |
| Standard Deviation | **77.72** | 292.76 | 282.28  |
| Median Time        | 600    | **548**  | 600     |

**(b)** Comparison of solvers for the FEEDBACK-VERTEX-SET problem on the complete data set of
Section 5. The time is measured in seconds and the instances are sorted by the number of vertices,
see also **(c)**.



**(c)** The left picture shows the difference of Jatatosk against D-Flat and Sequoia. A positive bar
means that Jatatosk is faster by this amount in seconds, and a negative bar means that either D-Flat
or Sequoia is faster by that amount. The bars are capped at 100. On every instance, Jatatosk
was compared against the solver that was faster on this particular instance. The image also shows
for every instance the size and the treewidth of the input. The right image shows the number of
instances that can be solved by each of the solvers in $x$ seconds, i. e., faster growing functions are
better. The colors in this image are as in **(b)**.