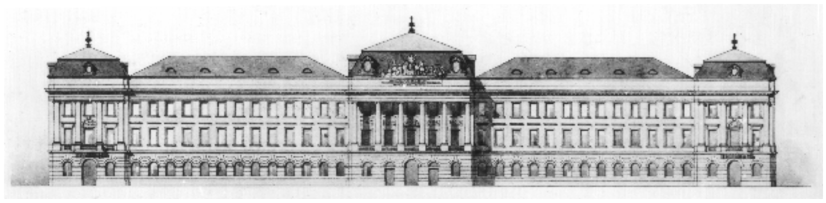
dbai

TECHNICAL REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning

DBAI-TR-2016-94

Michael Abseher, Nysret Musliu, Stefan Woltran

Institut für Informationssysteme
Abteilung Datenbanken und
Artificial Intelligence
Technische Universität Wien
Favoritenstr. 9
A-1040 Vienna, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18493
sek@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

DBAI TECHNICAL REPORT
2016

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DBAI TECHNICAL REPORT

DBAI TECHNICAL REPORT DBAI-TR-2016-94, 2016

Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning

Abseher Michael [1] **Nysret Musliu** [1] **Stefan Woltran** [1]

Abstract. Dynamic Programming (DP) over tree decompositions is a well-established method to solve problems – that are in general NP-hard – efficiently for instances of small treewidth. Experience shows that (i) heuristically computing a tree decomposition has negligible runtime compared to the DP step; (ii) DP algorithms exhibit a high variance in runtime when using different tree decompositions; in fact, given an instance of the problem at hand, even decompositions of the same width might yield extremely diverging runtimes. We thus propose here a novel and general method that is based on selection of the best decomposition from an available pool of heuristically generated ones. For this purpose, we require machine learning techniques that provide automated selection based on features of the decomposition rather than on the actual problem instance. As the good characterization of a tree decomposition is crucial in this process, we propose new original features. We report on extensive experiments in different problem domains which show a significant speedup when choosing the tree decomposition according to this concept over simply using an arbitrary one of the same width.

[1]TU Wien. E-mail: {abseher, musliu, woltran}@dbai.tuwien.ac.at

Acknowledgements: This work has been supported by the Austrian Science Fund (FWF): P25607-N23, P24814-N23, Y698-N23.
The authors thank Frederico Dusberger for helpful advice and excellent support during the early stages of the project.

Contents

1 Introduction

The notion of treewidth – and, as basic underlying concept, tree decompositions – was introduced by Bertelè and Brioschi [1973], Halin [1976] and Robertson and Seymour [1984]. Many NP-hard problems become tractable for instances whose treewidth is bounded by some constant k [Arnborg and Proskurowski, 1989; Niedermeier, 2006; Bodlaender and Koster, 2008]. A problem exhibiting tractability by bounding some problem-inherent constant is also called fixed-parameter tractable (FPT) [Downey and Fellows, 1999]. While constructing an optimal tree decomposition, i.e. a decomposition with minimal width, is intractable [Arnborg *et al.*, 1987], researchers proposed several exact methods for small graphs and efficient heuristic approaches that usually construct tree decompositions of almost optimal width for larger graphs. Examples of exact algorithms for tree decompositions are [Shoikhet and Geiger, 1997], [Gogate and Dechter, 2004], and [Bachoore and Bodlaender, 2006]. Greedy heuristic algorithms include Maximum Cardinality Search (MCS) [Tarjan and Yannakakis, 1984], Min-fill heuristic, Minimum Degree heuristic, to mention just a few. Metaheuristic techniques have been provided in terms of genetic algorithms ([Larranaga *et al.*, 1997; Musliu and Schafhauser, 2007]), ant colony optimization [Hammerl and Musliu, 2010], and local search based techniques ([Kjaerulff, 1992; Clautiaux *et al.*, 2004; Musliu, 2008]). A more detailed description of tree decomposition techniques is given in the recent surveys [Bodlaender and Koster, 2010; Hammerl *et al.*, 2015]).

A promising technique for solving problems using this concept is the computation of a tree decomposition followed by a dynamic programming (DP) algorithm that traverses the nodes of the decomposition and consecutively solves the respective sub-problems [Niedermeier, 2006]. For problems that are FPT w.r.t. treewidth, the general runtime of such algorithms for an instance of size n is $f(k) \cdot n^{\mathcal{O}(1)}$, where f is an arbitrary function of width k of the used tree decomposition. In fact, tree decompositions have been used for several applications including inference problems in probabilistic networks [Lauritzen and Spiegelhalter, 1988], frequency assignment [Koster *et al.*, 1999], computational biology [Xu *et al.*, 2005], and Answer-Set Programming [Morak *et al.*, 2012]. However, as recently stressed by Gutin [2015], to turn the concept of FPT to practical success, more empirical work is required. In terms of FPT algorithms for treewidth, experience shows that even decompositions of the same width lead to significant differences in the runtime of DP algorithms and recent results confirm that the width is indeed not the only important parameter that has a significant influence on the runtime. Morak *et al.* [2012], for instance, suggested that the consideration of further properties of tree decompositions is important for the runtime of DP algorithms for answer set programming. In another paper, Jégou and Terrioux [2014] observed that the existence of multiple connected components in the same tree node (bag) may have a negative impact on the efficiency of solving constraint satisfaction problems.

In this paper we want to gain a deeper understanding of the impact of tree decompositions on the runtime of DP algorithms by facilitaing machine learning technology. Recently, researchers have successfully used machine learning for runtime prediction and algorithm selection on several problem domains. Such problems include SAT [Xu *et al.*, 2008; Hutter *et al.*, 2014], combinatorial auctions [Leyton-Brown *et al.*, 2009], TSP [Smith-Miles *et al.*, 2010; Kanda *et al.*, 2011; Mersmann *et al.*, 2013; Pihera and Musliu, 2014; Hutter *et al.*, 2014], Graph Coloring [Smith-

Miles *et al.*, 2013; Musliu and Schwengerer, 2013], etc. For surveys on this domain, see e.g. [Smith-Miles, 2008; Hutter *et al.*, 2014; Kotthoff, 2014].

Our research gives new contributions in this area, as to the best of our knowledge, this is the first application of machine learning techniques towards the optimization of tree decompositions in DP algorithms. To this aim we propose new original features for tree decompositions that allows for a reliable prediction of the influence of a given tree decomposition on the performance of DP algorithms. Further, to select the most promising tree decomposition from a pool of generated ones using those features we propose an approach that applies machine learning techniques. We create the appropriate training sets by conducting extensive experiments on different problem domains, instances and tree decompositions. Moreover, we run our experiments on two inherently different prominent systems that apply DP algorithms on tree decompositions, namely D-FLAT [Abseher *et al.*, 2014] and SEQUOIA [Kneis *et al.*, 2011]. The two systems are problem-independent general-purpose frameworks which allow for (relatively) easy prototyping of various DP algorithms. Without loss of generality one can apply our new approach also on specialized DP algorithms.

The complete picture of our evaluation shows a significant benefit of selecting a decomposition that is promising according to our prediction in contrast to simply choosing an arbitrary one. Furthermore, the results confirm that our approach is generally applicable, independent from the particular solver and problem domain. Hence, relying only on the width as a measure for the quality of a tree decomposition appears to be a too narrow approach, and we see the strong need for new, enhanced notions which allow for a better discrimination between different tree decompositions of the same instance. In our experimental evaluation we show that our proposed features are indeed promising candidates for these new quality measures. Finally, the results provide valuable insights for laying the foundation to construct customized decompositions optimizing the relevant features.

This work is a significantly extended version of the conference paper [Abseher *et al.*, 2015]. Compared to our previous work this extended version proposes many additional features of tree decompositions and a full, formal characterization of all features. Furthermore, we completely rearranged the experiments, allowing for much deeper investigation. In this work we consider five problem domains instead of three used in our original paper. A pool of sixteen machine learning algorithms replaces the five algorithms used in the previous paper and the evaluation on real-world instances is now done on the problem of STEINER TREE, a problem with high practical relevance. We apply our approach in five concrete scenarios and each of the problem instances is investigated separately. Additionally, we also give a thorough evaluation of the sixteen models and also provide the results of an inter-domain evaluation of our approach.

The remainder of this article is organized as follows. In Section 2 we give the background of our work consisting of an introduction to tree decomposition and dynamic programming on tree decompositions. In Section 3 we propose a novel approach on how to improve the performance and robustness of dynamic programming on tree decompositions and in Section 4 we provide an extensive experimental evaluation of the new approach both on random input data and real-world instances. Following the experiments, Section 5 gives a detailed recipe on how to apply our approach in practice and Section 6 finally concludes our article.

2 Background

In the following we give a formal definition of tree decompositions and treewidth and illustrate the principle of DP on such decompositions. Furthermore, we provide a short overview on D-FLAT [Abseher *et al.*, 2014] and SEQUOIA [Kneis *et al.*, 2011], highlighting how each of them realizes DP on tree decompositions.

Tree decomposition is a technique often applied for solving NP-hard problems. The underlying intuition is to obtain a tree from a (potentially cyclic) graph by subsuming multiple vertices in one node and thereby isolating the parts responsible for the cyclicity. Formally, the notions of tree decomposition and treewidth are defined as follows [Robertson and Seymour, 1984; Bodlaender and Koster, 2010].

Definition 1. *Given a graph $G = (V, E)$, a* tree decomposition *of G is a pair (T, χ) where $T = (N, F)$ is a tree and $\chi : N \rightarrow 2^V$ assigns to each node a set of vertices (called the node's* bag*), such that the following conditions hold:*

1. *For each vertex $v \in V$, there exists a node $i \in N$ such that $v \in \chi_i$.*
2. *For each edge $(v, w) \in E$, there exists an $i \in N$ with $v \in \chi_i$ and $w \in \chi_i$.*
3. *For each $i, j, k \in N$: If j lies on the path between i and k then $\chi_i \cap \chi_k \subseteq \chi_j$.*

The width *of a given tree decomposition is defined as* $\max_{i \in N}|\chi_i| - 1$ *and the* treewidth *of a graph is the minimum width over all its tree decompositions.*

Note that the tree decomposition of a graph is in general not unique. In the following we consider rooted tree decompositions, for which additionally a root $r \in N$ is defined. Figure 1 gives an example of a graph and a possible (non-normalized) tree decomposition of it.

Definition 2. *Given a graph $G = (V, E)$, a* normalized *(or* nice*) tree decomposition of G is a rooted tree decomposition T where each node $i \in N$ is of one of the following types:*

1. Leaf*: i has no child nodes.*
2. Introduce Node*: i has one child j with $\chi_j \subset \chi_i$ and $|\chi_i| = |\chi_j| + 1$*
3. Forget Node*: i has one child j with $\chi_j \supset \chi_i$ and $|\chi_i| = |\chi_j| - 1$*
4. Join Node*: i has two children j, k with $\chi_i = \chi_j = \chi_k$*

Each tree decomposition can be transformed into a normalized one in linear time without increasing the width [Kloks, 1994].

For graph problems and problems that can be formulated on a graph, tree decompositions permit a natural way of applying DP by traversing the tree from the leaf nodes to its root. For each node $i \in N$ solutions for the subgraph of the instance graph induced by the vertices in χ_i are computed. When traversing to the next node the (partial) solutions computed for its children are taken into account, such that only consistent solutions are computed. Thus the partial solutions computed in

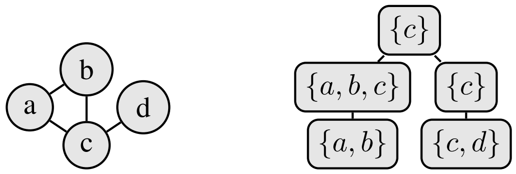
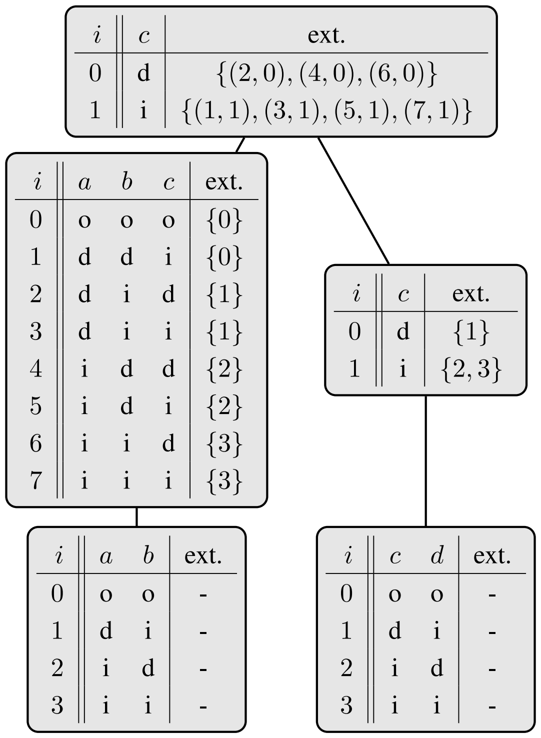


Figure 1 Example graph and a possible tree decomposition.



i	c	ext.
0	d	$\{(2,0),(4,0),(6,0)\}$
1	i	$\{(1,1),(3,1),(5,1),(7,1)\}$

i	a	b	c	ext.
0	o	o	o	$\{0\}$
1	d	d	i	$\{0\}$
2	d	i	d	$\{1\}$
3	d	i	i	$\{1\}$
4	i	d	d	$\{2\}$
5	i	d	i	$\{2\}$
6	i	i	d	$\{3\}$
7	i	i	i	$\{3\}$

i	c	ext.
0	d	$\{1\}$
1	i	$\{2,3\}$

i	a	b	ext.
0	o	o	-
1	d	i	-
2	i	d	-
3	i	i	-

i	c	d	ext.
0	o	o	-
1	d	i	-
2	i	d	-
3	i	i	-

Figure 2 Solving Dominating Sets via DP for the problem instance in Figure 1.

the root node are consistent to the solutions for the whole graph. Finally, the complete solutions are obtained in a reverse traversal from the root node to the leaves combining the computed partial solutions. Figure 2 shows the tables computed in a DP algorithm for solving the DOMINATING SET[1] problem on the graph given in Figure 1. The central columns of each table list the possible values for the vertices in the node constituting the partial solutions. Here, *i*, *d* and *o* stand for the respective vertex being *in* the selected set, *dominated* by being adjacent to a vertex from the set or simply *out*. The last column stores the possible partial solutions from the child node(s) that can consistently be extended to these values. Note that infeasibilities in the partial solutions can already lead to an early removal of solution candidates. The partial solution $\{c = \text{o}, d = \text{o}\}$ can, for instance, already be discarded when d is forgotten during the traversal to the next node, since d cannot appear again in any further node and can thus not become dominated or a part of the solution set anymore. The solution for the example graph can then be simply extracted by starting at the root of the tree de-

[1]The definition of this problem is given in Section 4.

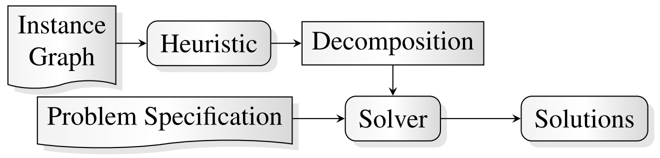
composition and following the extension pointers down to the leaves. One solution is for instance formed by selecting the vertices b and d. This solution is represented in the dynamic programming tables by the first line of the root table, row 2 of its left-hand and row 0 of its right-hand child together with row 1 of the left leaf and row 1 of the right leaf. For the sake of simplicity it is common practice to define DP algorithms on normalized tree decompositions [Bodlaender and Koster, 2008; Kneis *et al.*, 2011]. Thus, in the following, we refer to normalized decompositions unless stated otherwise.

In our experiments we use two systems that apply DP on tree decompositions, D-FLAT and SEQUOIA. D-FLAT [Abseher *et al.*, 2014] is a general framework capable of solving any problem expressible in monadic second-order logic (MSO) in FPT time w.r.t. the parameter treewidth [Bliem *et al.*, 2013]. The D-FLAT system combines DP on tree decompositions with answer set programming (ASP) [Brewka *et al.*, 2011]. Given an ASP encoding Π of the DP algorithm and an instance graph G, D-FLAT first constructs a tree decomposition from G in polynomial time using internal heuristics. If desired, this decomposition can be left untouched or be normalized in a preprocessing step. The decomposition is then traversed bottom-up in the manner described above, i.e. at each node the program specified by Π and the currently known facts for the vertices in that node's bag is solved. Notice that due to its internal structure D-FLAT always stores all partial solutions in its DP tables and thus generally solves the problem of enumerating all solutions to a given instance.

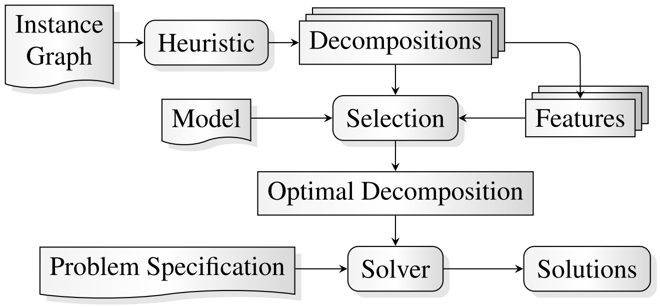
An alternative approach also based on DP on normalized tree decompositions is realized in the SEQUOIA framework [Kneis *et al.*, 2011]. Similarly to D-FLAT, SEQUOIA is a general solver that can be applied to any problem expressible in MSO. One general difference to D-FLAT is that it expects the problem definition directly in terms of an MSO formula, rather than an ASP program and that this formula has to describe the problem in a monolithic manner. In other words, the "decomposition" of the formula into the parts relevant to the separate tree decomposition nodes, as well as the special considerations on how partial solutions from child nodes are considered, are done internally. The system is based on the model checking game [Hintikka, 1973; Langer *et al.*, 2014] in which the verifier tries to prove that a given formula holds on the input graph while the falsifier tries to refute it. Further major differences between the two systems are the fact that SEQUOIA always outputs one (optimal) solution (for satisfiable instances) and does not enumerate all (optimal) possibilities and furthermore, SEQUOIA works only on normalized tree decomposition while D-FLAT can handle arbitrary ones. Also for this reason, we restrict ourselves to normalized tree decompositions in this article.

3 Improving the Efficiency of DP Algorithms

Systems such as SEQUOIA and D-FLAT follow a straight-forward approach for using tree decompositions: a single decomposition is generated heuristically and then fed into the DP algorithm used in the system (see Figure 3a). However, experiments have shown that the "quality" of such tree decompositions varies, leading to differing runtimes for the same problem instance. Most interestingly, "quality" does not necessarily mean low width. Even tree decompositions of the same width lead to huge differences in the runtime of a DP algorithm when applied to the same problem



(a) Standard Approach



(b) Improved Approach

Figure 3 Comparison of Approaches

instance.

The approach we propose in this work is illustrated in Figure 3b. The main idea is to generate a pool of tree decompositions for the given input instance and then to select, based on features of the decomposition, the one which promises best performance. The key aspects of the approach are as follows:

- Generating a number of tree decompositions for a given input graph can be usually done very efficiently by employing sophisticated heuristics for tree decompositions (e.g. min-fill heuristic), thus the runtime overhead will be negligible in most of the cases.

- Models allowing to predict the runtime behavior of a tree decomposition for a given DP algorithm are required. These models can be obtained in an off-line training-phase by running several instances with different tree decompositions and by storing the runtime and the feature values which are then processed by machine learning algorithms. For our purposes, machine learning techniques need to predict a good *ranking* of tree decompositions based on the predicted runtime for these decompositions. We note that a very accurate prediction of runtime is not crucial in our case, but rather predicting a correct order of tree decompositions. For example if the actual runtime of the DP algorithm using tree decomposition $TD1$ is faster than with $TD2$, it is important that machine learning algorithms predict that the runtime for $TD1$ is shorter than the runtime for $TD2$. In that case the order of tree decompositions is correct even if the runtime prediction for both tree decompositions is not completely exact.

- The main challenge and novel aspect of the approach is given by the fact that the features used to obtain these rankings need to be defined on the tree-decomposition not on the given problem instance. This is because instance features only help to distinguish instances but they do not help us to choose a proper decomposition as they are the same for each of the generated decompositions. To successfully exhibit machine learning techniques we need to find powerful features that characterize well the quality of tree decompositions. Moreover, the computation of these features needs to be done efficiently.

In other words, our approach works as follows: First, a number (which can be arbitrarily large) of tree decompositions of the given problem instance is computed and stored in a pool. Second, the features (acting as explanatory variables) of these decompositions are extracted and used to predict the runtime as the response variable. This gives us a ranking from which we select the decomposition with minimal predicted runtime (in case of existing ties, we choose randomly one of the decompositions with minimal predicted runtime). We apply several regression algorithms, such as linear regression, regression trees, nearest-neighbor, multi-layer perceptrons and support-vector machines to generate the models for prediction. Finally, the selected decomposition is handed over to the actual system to run the DP algorithm.

3.1 Tree Decomposition Features

In what follows we address one of the main contributions of the work, namely the identification of new tree decomposition features. We group them into three main categories: *decomposition size features*, *node-type specific features* and *structural features*.

Subsequently we will give for each feature a short description and formal specificaton. Providing multiple statistical key figures like minimum, maximum, mean and median leads us to a total of 144 features.

3.1.1 Notation

Before we present the collection of tree decomposition features, let us fix the formal notation we will use in the corresponding formulae. We assume a given tree decomposition TD (T, χ) with $T = (N, F)$ of a graph $G = (V, E)$. Each node $i \in N$ in a has associated a type $t_i \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\}$. Furthermore, we define the sets Leaf, $\mathit{Introduce}$, Forget and Join to contain exactly the nodes from TD where t_i matches the name of the set. The bag content of a node i is denoted by χ_i. The set $\mathit{NonLeaf}$ is defined as $N \setminus \mathit{Leaf}$ and the set $\mathit{NonEmpty}$ covers all nodes $i \in N$ where $|\chi_i| > 0$. The distance between two nodes i and j (in T) is given by the function $distance(i, j)$. By l_i we denote the level (also called depth) of a node i, given by the distance between the root and i. The level of the root r is thus 0. The set of children of node i is denoted by $\mathit{Children}_i$. The set N_v of a vertex $v \in V$ is the set of decomposition nodes $i \in N$ such that $\chi_i \cap \{v\} \neq \emptyset$. Some of the more elaborate features require information about the neighborhood and the reachability relation. For this reason we define the following functions:

- $neighbors(v)$

  Returns the set of neighbors of vertex v (excluding v) in the original graph.

- $adjacent(u, v)$

  Returns 1 in case that vertices u and v are adjacent in the original graph and 0 otherwise.

- $reachable(u, v)$

  Returns 1 in case that vertex v can be reached from u in the original graph and 0 otherwise.

Most of the features we present and use in this paper rely heavily on the use of aggregates. In order to avoid redundancies we employ two slightly different sets of aggregates in the actual computations shown below. Note that we give here just the simple enumeration of the aggregates we use for our experiments. When implementing our approach one can use (almost) all possible subsets and every superset of the following sets.

- $\mathit{Agg1} = \{\mathit{count}, \mathit{min}, \mathit{max}, \mathit{mean}, \mathit{median}, \mathit{sd}$ (Standard Deviation) $\}$
- $\mathit{Agg2} = \mathit{Agg1} \setminus \{\mathit{count}\}$

3.1.2 Decomposition Size Features

The set of features dealing with *decomposition size* measures the general complexity of a given tree decomposition, like the number of nodes in the tree decomposition or statistics on the bag size over all nodes.

BagSize, NonLeafNodeBagSize, NonEmptyNodeBagSize These feature shall capture the complexity of the tree decomposition by recording the size of the bags. Because of the fact that leaf nodes are sometimes treated different in a DP algorithm than the other nodes in the decomposition, we also measure the bag size of non-leaf nodes separately. Furthermore, we take care of empty nodes. This seems natural because the presence of empty nodes might change the outcome of some of the aggregates over the plain bag size drastically but empty nodes often do not influence the actual runtime. This behavior is explained by the fact that a DP algorithm will only spend constant time in these nodes. To deal with this kind of nodes we therefore measure the bag size of non-empty nodes separately. We have for each $\alpha \in \mathit{Agg1}$

$$\begin{aligned} \mathit{BagSize}^{\alpha} &= \alpha(\{|\chi_i| : i \in N\}) \\ \mathit{BagSize}^{\alpha}_{\mathit{NonLeaf}} &= \alpha(\{|\chi_i| : i \in \mathit{NonLeaf}\}) \\ \mathit{BagSize}^{\alpha}_{\mathit{NonEmpty}} &= \alpha(\{|\chi_i| : i \in \mathit{NonEmpty}\}) \end{aligned} \tag{1}$$

CumulativeBagSize This feature is created based on the assumption that tree decompositions containing a multitude of vertices of the input graphs are bad for the performance. The value for this feature is computed by just summing up all the bag sizes of the decomposition at hand.

$$\mathit{CumulativeBagSize} = \sum_{i \in N} |\chi_i| \tag{2}$$

DecompositionOverheadRatio This feature which is closely related to *CumulativeBagSize* represents the size increase between the original graph and the corresponding tree decomposition. The value for this feature is computed by dividing *CumulativeBagSize* by the number of vertices in the input graph.

$$DecompositionOverheadRatio = \left(\sum_{i \in N} |\chi_i|\right) / |V| \tag{3}$$

ContainerCount By the container count of a vertex v we refer to the number of bags a vertex v of the original graph appears in. For each $\alpha \in Agg2$ we have

$$ContainerCount^{\alpha} = \alpha(\bigcup_{v \in V} |\{i : v \in \chi_i\}|) \tag{4}$$

ItemLifetime This feature is very similar to *ContainerCount*, but this time we only count the number of distinct levels of the tree decomposition the vertex appears in. For each $\alpha \in Agg2$ we have

$$ItemLifetime^{\alpha} = \alpha(\bigcup_{v \in V} |\{l_i : v \in \chi_i\}|) \tag{5}$$

NodeDepth, NonLeafNodeDepth, NonEmptyNodeDepth These three features measure the distance from the root node to the node under focus. The idea behind these features is the possibility that trees of low height could provide benefits (or disadvantages) compared to trees with a multitude of levels. Again we distinguish between the complete set of tree decomposition nodes, the non-leaf nodes and the non-empty nodes in order to capture the most important cases. For each $\alpha \in Agg2$ we have

$$\begin{aligned} NodeDepth^{\alpha} &= \alpha(\{l_i : i \in N\}) \\ NodeDepth^{\alpha}_{NonLeaf} &= \alpha(\{l_i : i \in NonLeaf\}) \\ NodeDepth^{\alpha}_{NonEmpty} &= \alpha(\{l_i : i \in NonEmpty\}) \end{aligned} \tag{6}$$

3.1.3 Node Type Features

The node type specific features contain statistics which are computed separately for each node type (leaf, introduce, forget and join). This can be important as the computational work that has to be done in a node during the execution of a DP algorithm depends on both the implementation of the DP algorithm and the node type.

Percentage This feature records the overall percentage of the respective node type in the tree decomposition at hand.

$$Percentage_t = |\{i : i \in t\}| / |N| \qquad \text{for } t \in \{Leaf, Introduce, Forget, Join\} \tag{7}$$

BagSize Analogous to the feature *BagSize* in the section dedicated to the decomposition size features, but this time for each type of node (leaf, introduce, forget and join) separately. For each $\alpha \in \mathit{Agg1}$ we have

$$\mathit{BagSize}_t^\alpha = \alpha(|\{i : i \in t\}|) \qquad \text{for } t \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\} \tag{8}$$

CumulativeBagSize To capture the overall number of vertices in introduce and forget nodes, we also measure the sum of bag sizes for each corresponding node type. Furthermore, we measure the total number of vertices of the original graph participating in joins and the number of vertices being part of leaf nodes.

$$\mathit{CumulativeBagSize}_t = \sum_{i \in t} |\chi_i| \qquad \text{for } t \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\} \tag{9}$$

Depth Analogous to the depth measurements proposed in the previous section, this feature captures the distance from the root node for every node of the respective type in the decomposition. The intuition behind this feature is that the higher the depth of a node, the longer the decisions made in this node have to be propagated and so it can strongly influence the performance of the DP algorithm. For each $\alpha \in \mathit{Agg2}$ we have

$$\mathit{Depth}_t^\alpha = \alpha(\{l_i : i \in t\}) \qquad \text{for } t \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\} \tag{10}$$

JoinNodeDistance Join nodes often have a strong influence on the runtime of DP algorithms as they have the potential to increase (or decrease) the number of valid solution candidates drastically. This feature keeps track of the distance between join nodes and it is 0 in the case that not more than a single join node is present in the decomposition.

In case that two or more join nodes are present, the distance is measured for each pair i and j of join nodes separately by taking the length of the path between i and j in the decomposition. In case that not more than one join node is present, all these values are set to 0. For each $\alpha \in \mathit{Agg2}$ we have

$$\mathit{JoinNodeDistance}^\alpha = \alpha(\{\mathit{distance}(i,j) : i,j \in \mathit{Join}, i \neq j\}) \tag{11}$$

3.1.4 Structural Features

Finally, to characterize the shape of the tree decomposition we propose several *structural features*. Most of the features presented here take care of the bag content and measure relations between them, like connectedness.

BranchingFactor This feature measures the number of children for each node within the tree decomposition. The intuition is that it most probably makes a big difference if one deals with lots of join nodes (having two or potentially also more children) or with simple exchange nodes, having only a single child. For each $\alpha \in \mathit{Agg2}$ we have

$$\mathit{BranchingFactor}^\alpha = \alpha(\{\mathit{Children}_i : i \in N\}) \tag{12}$$

BagAdjacencyFactor This feature measures the ratio of the number of pairs of vertices in the bag that are adjacent in the original graph G and the total number of vertex pairs in the bag. The intuition behind this feature is the possibility that seeing the a big part of the vertices' neighborhood at once might speedup DP algorithms. For each $\alpha \in \mathit{Agg2}$ we have

$$BAF^{\alpha} = \alpha\left(\left\{\frac{|\{(u,v) : u,v \in \chi_i, u \neq v, \mathit{adjacent}(u,v)\}|}{\mathit{max}(1, |\chi_i| * (|\chi_i| - 1))} : i \in N\right\}\right) \tag{13}$$

BagConnectednessFactor This feature relates the number of pairs of vertices in the bag that are connected in the original graph G to the total number of vertex pairs in the bag. The value for a single bag i is computed by averaging over all values for the vertices in i. For each $\alpha \in \mathit{Agg2}$ we have

$$BCF^{\alpha} = \alpha\left(\left\{\frac{|\{(u,v) : u,v \in \chi_i, u \neq v, \mathit{reachable}(u,v)\}|}{\mathit{max}(1, |\chi_i| * (|\chi_i| - 1))} : i \in N\right\}\right) \tag{14}$$

BagNeighborhoodCoverageFactor For each vertex in the bag the ratio between the number of neighbors in the bag to the number of neighbors in the original graph is computed. The value for a single bag i is computed by averaging over all values for the vertices in i. For each $\alpha \in \mathit{Agg2}$ we have

$$BNCF^{\alpha} = \alpha\left(\left\{\mathit{mean}\left(\left\{\frac{|\mathit{neighbors}(v) \cap \chi_i|}{|\mathit{neighbors}(v)|} : v \in \chi_i\right\}\right) : i \in N\right\}\right) \tag{15}$$

IntroducedVertexNeighborCount, ForgottenVertexNeighborCount Experience shows that propagating information is in many cases not the bottleneck for DP algorithms. In fact, most of the "real" work has to be done when vertices are introduced or forgotten and the algorithm has to evaluate rules and check constraints between the new (forgotten) vertex and the neighbors in the current bag. These two features are dedicated exactly to this issue. For each $\alpha \in \mathit{Agg2}$ we have

$$IVNC^{\alpha} = \alpha\left(\{|\mathit{neighbors}(v) \cap \chi_i| : v \in V, i \in N_v\}\right) \tag{16}$$

(Note that v denotes an introduced vertex for bag i in Formula (16). For all other vertices in the bag associated with node i, the feature value is not computed to avoid a incorrect results in the aggregation phase which is needed to get the actual, atomic, statistical measurements. The values for feature ForgottenVertexNeighborCount are computed analogously for all forgotten vertices v.)

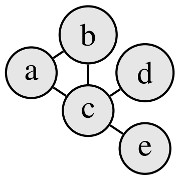
IntroducedVertexConnectednessFactor, ForgottenVertexConnectednessFactor Very closely related to the proposed feature of the neighbor count for introduced and forgotten vertices is the connectedness factor. For the last two features used in this work we measure the ratio between the number of vertices in the bag connected to a introduced (forgotten) vertex v in the original graph and the total number of all possible connections between all nodes in the bag. For each $\alpha \in \mathit{Agg2}$ we have

$$IVCF^{\alpha} = \alpha\left(\left\{\frac{|\{(u,v) : u,v \in \chi_i, u \neq v, \mathit{reachable}(u,v)\}|}{\mathit{max}(1, |\chi_i| * (|\chi_i| - 1))} : v \in V, i \in N_v\right\}\right) \tag{17}$$

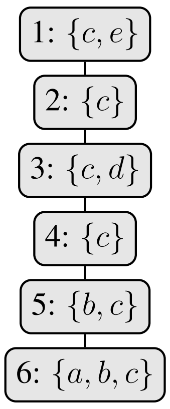
(Note that v denotes an introduced vertex for bag i in Formula (17). For all other vertices in the bag associated with node i, the feature value is not computed to avoid incorrect results in the aggregation phase which is needed to get the actual, atomic, statistical measurements. The values for feature ForgottenVertexConnectednessFactor are computed analogously for all forgotten vertices v.)

3.1.5 Features By Example

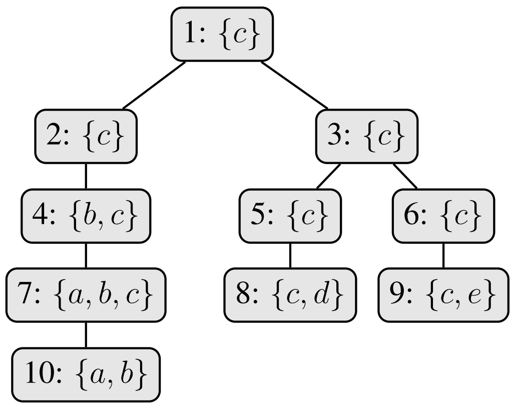
After presenting the feature set we will now give a short example in order to illustrate how the difference between tree decompositions (of the same width) is captured via these featuers. The input graph G we will base the computations on is shown in Figure 4a. Below, in Figure 4b and Figure 4c, we provide two different, normalized tree decompositions for the input graph G.



(a) Example graph G



(b) *TD1*



(c) *TD2*

Figure 4 Two normalized tree decompositions *TD1* and *TD2* for graph G

We can see that both decompositions *TD1* and *TD2* have a maximum bag size of 3. This means that they have exactly the same width – namely 2 – but they most probably will lead to a completely different runtime behavior when they are processed in a dynamic programming algorithm. We will use these example decompositions to show that our proposed features will allow use to clearly distinguish them. In Table 1 we give a selection of the features we will obtain for each of the two decompositions. For brevity we only provide the median for those features where multiple aggregates are applied.

Feature	$TD1$	$TD2$
$BagSize^{Median}$	2	1.5
$NonLeafNodeBagSize^{Median}$	2	1
$CumulativeBagSize$	11	16
$DecompositionOverheadRatio$	2.2	3.2
$ContainerCount^{Median}$	1	2
$ItemLifetime^{Median}$	1	2
$NodeDepth^{Median}$	2.5	2
$LeafNodePercentage$	16.67 %	30.00 %
$IntroduceNodePercentage$	33.33 %	10.00 %
$ForgetNodePercentage$	50.00 %	40.00 %
$JoinNodePercentage$	0.00 %	20.00 %
$LeafNodeBagSize^{Median}$	3	2
$IntroduceNodeBagSize^{Median}$	2	3
$ForgetNodeBagSize^{Median}$	1	1
$JoinNodeBagSize^{Median}$	0	1
$CumulativeLeafNodeBagSize$	3	6
$CumulativeIntroduceNodeBagSize$	4	3
$CumulativeForgetNodeBagSize$	4	5
$CumulativeJoinNodeBagSize$	0	3
$LeafNodeDepth^{Median}$	5	3
$IntroduceNodeDepth^{Median}$	1	3
$ForgetNodeDepth^{Median}$	3	3
$JoinNodeDepth^{Median}$	0	0.5
$JoinNodeDistance^{Median}$	0	1
$BranchingFactor^{Median}$	1	1
$BagAdjacencyFactor^{Median}$	1	1
$BagConnectednessFactor^{Median}$	1	1
$BagNeighborhoodCoverageFactor^{Median}$	0.5	0.19
$IntroducedVertexNeighborCount^{Median}$	1	2
$ForgottenVertexNeighborCount^{Median}$	1	1
$IntroducedVertexConnectednessFactor^{Median}$	1	1
$ForgottenVertexConnectednessFactor^{Median}$	1	1

Table 1   Subset of extracted features for decompositions $TD1$ and $TD2$ of graph G.

4 Experimental Evaluation

In this section, we experimentally evaluate the proposed method. All our experiments were performed on a single core of an Intel Xeon E5-2637@3.5GHz processor running Debian GNU/Linux 8.3 and each test run was limited to a runtime of at most six hours and 64 GB of main memory.

We evaluate our approach using two recently developed DP solvers, D-FLAT (v. 1.0.1) and SEQUOIA (v. 0.9). The subsequent machine learning tasks were carried out with WEKA 3.6.13 [Hall *et al.*, 2009].

The full benchmark setup including instances, programs, configurations, problem encodings and all results can, together with a description on how to reproduce the results of the paper, be downloaded under the following link:

`www.dbai.tuwien.ac.at/research/project/dflat/features_2016_03.zip`

4.1 Methodology

Problems In our analysis we considered the following set of problems defined on an undirected graph $G = (V, E)$.

1. MINIMUM DOMINATING SET (MDS): Find all sets $S \subseteq V$ of minimal cardinality, such that for all $u \in V$ either $u \in S$ or there is an edge $(u, v) \in E$ with $v \in S$.

2. 3-COLORABILITY (COL): Is G 3-colorable?

3. PERFECT DOMINATING SET (PDS): Find all sets $S \subseteq V$ of minimum size meeting the following requirements:

   - S is a dominating set of G.
   - $\forall x \in S : x$ dominates at most one $y \in (V \setminus S)$.
   - $\forall y \in (V \setminus S) : y$ is dominated by exactly one $x \in S$.

4. CONNECTED VERTEX COVER (CVC): Find all sets $S \subseteq V$, such that for all $(u, v) \in E$, $u \in S$ or $v \in S$, and the vertices in S form a connected subgraph of G.

Furthermore, we considered the following problem defined on undirected graph $G = (V, E)$ with edge weights $E \to \mathbb{N}$.

5. STEINER TREE (ST): Given a set of terminal vertices $T \subseteq V$, find all sets of edges $X \subseteq E$ of minimum total weight which meet the following requirements:

   - For every $t \in T$ there is an $e \in X$ containing t.
   - The graph formed by the edges in X is connected.

We note that the goal of this paper is not to outperform existing, specialized state-of-the-art solvers for the respective problem domains but to improve the performance and robustness of dynamic programming algorithms on tree decompositions. Indeed the methods based on tree decomposition are exact techniques and currently can usually solve only problems of limited size.

Machine Learning Algorithms In our experiments we apply 16 models which are computed using WEKA's regression algorithms which have been used successfully in different application domains. For each of the five problems and for each solver for the respective problem at hand, the following machine learning algorithms were considered (The first line within each item shows the regression algorithm used and the below of each algorithm we provide the exact command line parameters applied in our experiments.):

1. GaussianProcesses (weka.classifiers.functions.GaussianProcesses)

   -L 1.0 -N 0 -K "weka.classifiers.functions.supportVector.PolyKernel -C 250007 -E 1.0"

2. IsotonicRegression (weka.classifiers.functions.IsotonicRegression)

   (Algorithm is not configurable.)

3. LeastMedSq (weka.classifiers.functions.LeastMedSq)

   -S 4 -G 0

4. LinearRegression (weka.classifiers.functions.LinearRegression)

   -S 1 -R 1.0E-8

5. MultilayerPerceptron (weka.classifiers.functions.MultilayerPerceptron)

   -L 0.3 -M 0.2 -N 500 -V 0 -S 0 -E 20 -H a

6. PaceRegression (weka.classifiers.functions.PaceRegression)

   -E pace6

7. PLSClassifier (weka.classifiers.functions.PLSClassifier)

   -filter "weka.filters.supervised.attribute.PLSFilter -C 20 -M -A PLS1 -P none"

8. SMOreg (weka.classifiers.functions.SMOreg)

   -C 1.0 -N 2 -I "weka.classifiers.functions.supportVector.RegSMOImproved -L 0.001 -P 1.0E-12 -W 1 -T 0.001 -V" -K "weka.classifiers.functions.supportVector.RBFKernel -C 250007 -G 0.01"

9. IBk (weka.classifiers.lazy.IBk)

   -K 9 -W 0 -I -A "weka.core.neighboursearch.LinearNNSearch -A "weka.core.EuclideanDistance -R first-last""

10. KStar (weka.classifiers.lazy.KStar)

    -B 20 -M a

11. LWL (weka.classifiers.lazy.LWL)

    -U 0 -K -1 -A ”weka.core.neighboursearch.LinearNNSearch
    -A ”weka.core.EuclideanDistance -R first-last”” -W weka.classifiers.functions.SMOreg -C 1.0 -N 2
    -I ”weka.classifiers.functions.supportVector.RegSMOImproved -L 0.001 -P 1.0E-12 -T 0.001 -V” -K
    ”weka.classifiers.functions.supportVector.RBFKernel -C 250007 -G 0.01”

12. AdditiveRegression (weka.classifiers.meta.AdditiveRegression)

    -S 1.0 -I 10 -W weka.classifiers.functions.SMOreg
    -C 1.0 -N 2 -I ”weka.classifiers.functions.supportVector.RegSMOImproved -L 0.001 -P 1.0E-12
    -W 1 -T 0.001 -V” -K ”weka.classifiers.functions.supportVector.RBFKernel -C 250007 -G 0.01”

13. Bagging (weka.classifiers.meta.Bagging)

    -P 100 -S 1 -I 10 -W weka.classifiers.functions.SMOreg
    -C 1.0 -N 2 -I ”weka.classifiers.functions.supportVector.RegSMOImproved -L 0.001 -P 1.0E-12
    -W 1 -T 0.001 -V” -K ”weka.classifiers.functions.supportVector.RBFKernel -C 250007 -G 0.01”

14. CVParameterSelection (weka.classifiers.meta.CVParameterSelection)

    -X 10 -S 1 -W weka.classifiers.trees.M5P -M 4.0

15. M5Rules (weka.classifiers.rules.M5Rules)

    -M 2.0

16. M5P (weka.classifiers.trees.M5P)

    -M 2.0

The exact configuration for each algorithm is given in the above enumeration and was chosen by taking the configuration with highest correlation between the actual, standardized runtime and the predicted, standardized runtime.

This initial evaluation considers all parameters provided by WEKA and was done on a separate benchmark set consisting of 500 tree decompositions (50 instances of 3-COLORABILITY with 10 decompositions for each instance). For each parameter we experimented with different values and we used cross validation to determine the performance of each configuration. The fixed parameters were used problem domains that we investigate in this paper. This underlines the generality and flexibility of our approach, but one could also consider using a tailored algorithm setup for each problem domain.

Training Set All the aforementioned machine learning algorithms were trained separately for each problem using a training set consisting of 800 benchmark runs. These runs are obtained by running 20 satisfiable[2] problem instances with 40 different tree decompositions. In addition

[2]Not all generated instances are satisfiable for 3-COL or CVC. In our experiments for these problems, we consider only those that are, because for unsatisfiable instances, a large part of the tree decomposition might not even be visited

to restricting the training set to satisfiable instances, we also ensured that the training set contains no benchmark runs which exceeded the allowed time or the memory limit to definitively rule out biased results.

The problem instances we used in our experiments are of different size and also the probability of whether an edge exists between two vertices of the input graph varies for different instances. While these variations are automatically present in the real-world instances we investigated, we applied the Erdős-Rényi random graph model to achieve an appropriate level of randomness for the constructed instances. For these random instances, we used three graph sizes and three different edge probabilities per problem to construct the corresponding training set.

After termination of a test run we extracted all of our proposed tree decomposition features and stored the outcome together with the runtime achieved by the dynamic programming algorithm. When all test runs for a problem instance were finished, we had to normalize the results in order to make sure that each instance contributes equally to the computation of the machine learning models. This normalization step is done by standardizing these values X feature-wise based on the formula $(X - \mu)/\sigma$.

Evaluation Set The evaluation set for the computed models consists of 2000 benchmark runs per problem. These runs are obtained by running 50 problem instances with 40 different tree decompositions. Again, we ensured that unsatisfiable instances and such that violate the limits are excluded. In general, all the steps for generating the evaluation set are identical to the steps used to generate the training set. The only exception is the fact that at this point we are not allowed to normalize the runtime because we clearly want to know the actual time needed to solve the respective instance and not only its normalized abstraction.

For a given instance, the actual evaluation is done by predicting the normalized runtime the problem-specific DP algorithm will need to solve the problem. We do this for each model and for each of the 40 tree decompositions. Afterwards, we select for each model the tree decomposition with the minimum runtime where ties are broken randomly. All that remains is to simply lookup the real, non-normalized runtime and compare it with the median runtime (the runtime the "average" decomposition would lead to) over all the tree decomposition for the problem instance.

The value for the runtime improvement which we will investigate in the subsequent sections is computed by subtracting the quotient of the selected decomposition's actual runtime and the median runtime from 1. This means that a result of 0 implies that no improvement could be made and the (unlikely) result of 1 means that we were able to save 100%. Every value less than 0 means that we observe a deterioration of the performance using the respective model as runtime predictor and we clearly would not choose this model in practice.

As the runtime improvement is strongly dependent on the size of the problem at hand and as it is not the only interesting observation we can make, we also investigate the predicted rank. This measurement refers to the rank the tree decomposition predicted as the optimal one achieves within the pool of 40 decompositions when the pool is ranked by the actual runtime. The closer the

by a DP algorithm. This is due to the fact that the algorithm terminates as soon as it is evident that no solution exists for the instance at hand. Therefore, unsatisfiable problem instances do not allow us to investigate the effect of decomposition selection on the runtime of DP algorithms and we thus omit them in our comparisons.

predicted rank is to 1, the better. One can expect a runtime improvement whenever the predicted rank is less than the median rank, which is for our pool of 40 tree decompositions between 20 and 21. It is important to mention that although rank 1 is not achieved one can still significantly improve the performance if the selected tree decomposition is ranked better than the "average" tree decomposition.

Evaluation Process Indeed, the strict separation in training set and evaluation set makes the experiments prone to potential bias. To overcome this issue, we use a randomized adaption of the well-established technique of 10-fold cross-validation throughout our whole experimental evaluation. This means that the complete experimental set consists for a problem and a solver consists of 2800 tree decompositions (70 problem instances with 40 tree decompositions for each of the instances).

For each of the ten iterations we select randomly 20 problem instances (leading to 800 tree decompositions) for the training set and the remainder of the pool is put into the evaluation set. The analysis then proceeds as described in the paragraph dedicated to the evaluation set. This process is repeated ten times to rule out bias as good as possible. By doing so, we obtain for each problem, solver and model a total of 500 measurements from which we can draw precise conclusions about the runtime improvement obtained by using the tree decomposition predicted as the optimal one and the same holds also for conclusion about the predicted rank.

4.2 Experiments on Random Instances

Subsequently we provide a thorough investigation of experiments on random instances to show the potential of our approach. For every problem and each of the sixteen machine learning algorithms in our experimental setup we will present the predicted rank and the runtime improvement via boxplots. We also give aggregated performance measurements based on all computed models to underline the advantages our approach of selecting the optimal decomposition from a pool provides compared to the standard way of computing only one decomposition for a given problem instance.

4.2.1 Minimum Dominating Set

The first problem we look at is the well-known problem of selecting a subset S of the input graph's vertices such that each vertex of the input graph is either contained in S or adjacent to at least one vertex that is contained in S. The results we obtained in our experiments are summarized in Figures 5 and 6. Before we go into the details of the figures, we first want to introduce their structure as it is crucial for interpreting the expected performance gain and therefore it will follow us throughout the remainder of the paper.

In the header of each figure the problem name, the solver, the origin of the input instances as well as aggregated measurements for the performance improvement are given. The values are computed based on the median improvement obtained by using each Model 1–16. Furthermore, in the last row of the header we provide the results of our analysis for statistical significance. The value gives the probability that our approach leads to an improvement and is computed by

MINIMUM DOMINATING SET
D-FLAT Random Instances

Minimum Improvement:	7.39 %	Average Improvement:	21.80 %
Maximum Improvement:	31.15 %	Median Improvement:	24.25 %

Statistical Significance:	$\geq$ 99.95 %

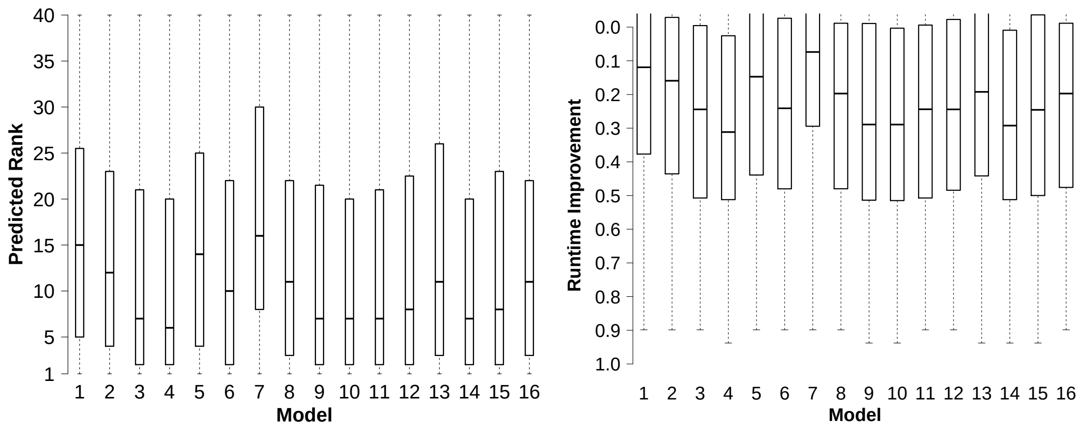


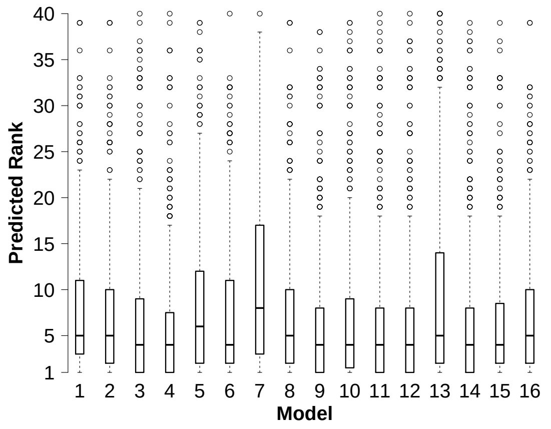
Figure 5 Performance Characteristics for MINIMUM DOMINATING SET (D-FLAT)

taking the median significance of the one-sided t-test with the null-hypothesis that the observed performance improvement for the model is 0. In other words, this last value in the header gives the probability that the average model will indeed lead to an performance improvement.

After this short introduction, let's have a look at the concrete values for the problem at hand. The figure headers for MINIMUM DOMINATING SET tell us that the improvement for any of the sixteen models is between 7.39% and 31.15% for D-FLAT and between 11.39% and 19.65% for SEQUOIA while both median and average improvement are relatively close to the maximum. Please note that this is the net runtime improvement we would achieve in practice, hence we immediately see that the approach indeed pays off and that we can easily save a large portion of the total runtime. The very high statistical significance of not less than 99.95 percent – quite close to absolute certainty – for our benchmark setup at hand finally tells us that the results are not a lucky strike and that we can also expect performance improvements in future experiments, at least in the same problem domain.

The two box-plots in Figures 5 and 6 are constructed on the basis of the 500 evaluations (50 instances with 10 iterations each) for each solver and computed Model 1–16 (see Section 4.1). On the left-hand side the predicted rank is illustrated and on the right-hand side we provide the box-plot of the distribution of the runtime improvement. Due to the fact that box-plots show the statistical distribution of values we gain even more insights into the capabilities of our proposed approach: By looking at the quartiles and outliers we can directly reason about the potential of our approach depending on the problem instances at hand. To allow for a uniform presentation

SEQUOIA Random Instances			
Minimum Improvement:	11.39 %	Average Improvement:	16.24 %
Maximum Improvement:	19.65 %	Median Improvement:	17.39 %
Statistical Significance:	≥ 99.95 %		



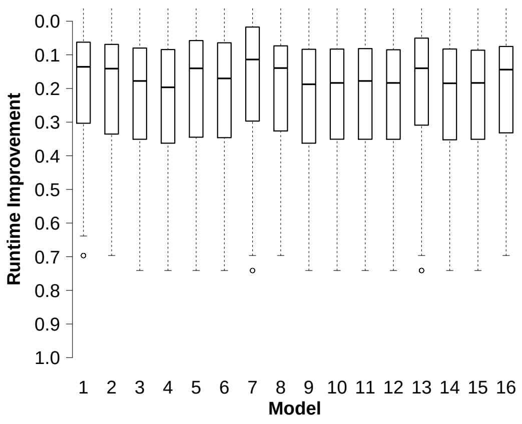


Figure 6 Performance Characteristics for MINIMUM DOMINATING SET (SEQUOIA)

and because models leading to performance deteriorations would never be selected in practice, the box-plot for the performance improvement only shows the interesting range between 0 and 1.

Before we move on, note that the importance of providing both the predicted rank and the performance improvement becomes apparent when we compare the two plots: When we focus on the left-hand side only, we do not obtain any insights what performance gain was achieved by a good selection, because the maximum possible improvement strongly depends on the performance differences imposed by the different tree decompositions. When we only look at the right-hand-side, we have no possibility to reason about the performance in future experiments as we don't see which rank was selected and hence we cannot map it to upcoming situations. Hence, the two figures complement each other.

The fact that predicted rank and runtime improvement are different, yet related dimensions in our evaluation is underlined when we finally compare the two solvers. We can see that D-FLAT and SEQUOIA behave totally different and that there is appearing a strange situation: On the one hand we have SEQUOIA where the prediction is very good but the median performance gain is only moderate. On the other hand we have D-FLAT where the predicted rank is about twice as high but where we still save significantly more time. This can be explained by the fact that SEQUOIA is more robust with respect to the runtime (at least in our setup), hence different tree decompositions do not influence the solving time as much as they do in the case of D-FLAT. For the remainder of this paper we will concentrate our analysis on D-FLAT because there the solving times for a given instance are much more diverse and so we expect more insights in the quality for each of the models.

4.2.2 3-Colorability

After introducing the capabilities of our approach by means of MINIMUM DOMINATING SET we will now present our experiments on 3-COLORABILITY. Compared to the first problem we investigated in this paper, this one is a less "complex" because one does not have to keep track of additional, global information like the size of the dominating set in order to minimize it. In the problem at hand it is sufficient to look at each vertex in the input graph separately and simply check for each introduced neighbor if it is assigned the same color as the vertex under focus. Hence, there is almost no propagation of information needed, except for keeping track of the vertex color within the current tree decomposition node. Therefore, we could expect that machine learning for this second problem is somewhat easier than for MINIMUM DOMINATING SET and that the performance improvement is higher. Indeed these assumptions are confirmed in this case when we compare the figures for the two problems. Although this works in our setup, we note that this implication most probably will not hold in general because all the results depend on the DP algorithms' implementations for the two problems, the input instances and, clearly, the pool of tree decompositions computed for each instance.

Apart from this small side remark we have for the problem domain at hand the situation that each of the computed models has a good selection quality (compared to selecting a decomposition randomly) in most of the cases, as we can see in both figures. An interesting fact visualized in the figures is the one that many models select in average a rank less than 10 out of 40 available tree decompositions for a problem instance while Models 7 and 13 (PLSClassifier and Bagging) are still good but significantly worse than the others.

4.2.3 Perfect Dominating Set

An extension to the problem of finding minimum dominating sets in a given input graph is the problem of finding minimum perfect dominating sets in a graph. The only difference between the two problems is the fact that in the latter, a dominated vertex must have exactly one dominator. This allows for much less solutions and so we expect a higher impact of the tree decomposition features and therefore a much better predicted rank than in the case of MINIMUM DOMINATING SET.

Figure 8 shows that not only the predicted rank is almost perfect for most of the models and also the runtime is cut in half in almost any of the investigated cases. Interestingly, most of the models predict rank 5 or better for the majority of the input instances. Again we observe that Models 7 and 13 (PLSClassifier and Bagging) show a worse outcome than the remaining models, but they still lead to an optimized runtime behavior.

4.2.4 Connected Vertex Cover

The next problem we want to focus on is CONNECTED VERTEX COVER. In practical situations, the connectedness of a solution often is a crucial feature and so it is important to show that our proposed approach also works in these scenarios. The requirement for connectedness makes the prediction of runtime even harder because before solving the problem at hand there is no chance

3-COLORABILITY			
D-FLAT Random Instances			
Minimum Improvement:	16.90 %	Average Improvement:	29.74 %
Maximum Improvement:	36.67 %	Median Improvement:	30.99 %
Statistical Significance:		≥ 99.95 %	

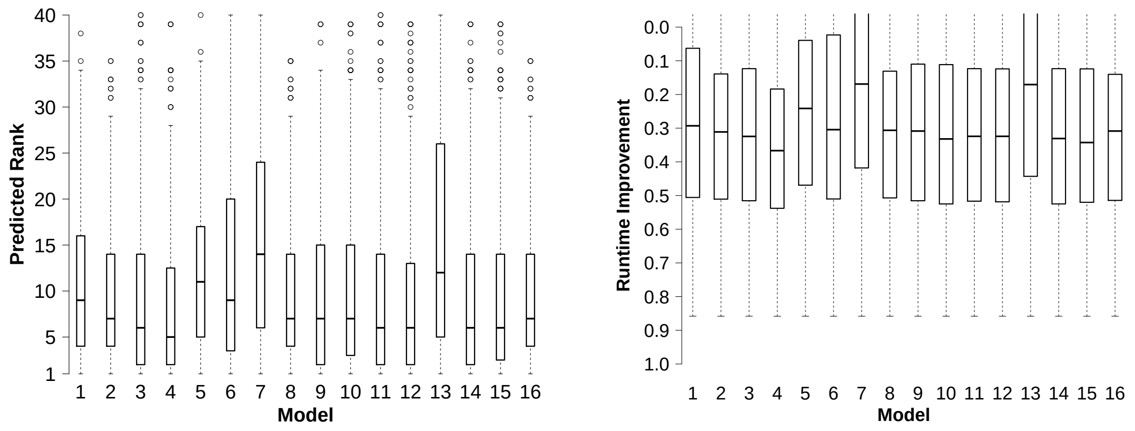


Figure 7 Performance Characteristics for 3-COLORABILITY (D-FLAT)

to maintain the solution's property of connectedness and to keep track of it only by looking at the tree decomposition features.

Figure 9 shows that also scenarios of this kind can be handled by our approach. Although the prediction is less accurate than in the case of PERFECT DOMINATING SET – a fact that was expected, as mentioned above. – we save one third of the overall runtime in the median case, which is indeed surprising. Even the Models 7 and 13 (PLSClassifier and Bagging) which again are performing worst allow us to save a significant portion of the runtime in most of the cases. This time, we also have with model number 6 (PaceRegression) a dedicated "winner" of the comparison as it is able to predict a rank between 1 and 10 in 75% of the cases and it selects rank 4 out of 40 in the majority of the cases.

4.2.5 Steiner Tree

The final problem domain we investigate in this paper is the problem of STEINER TREE. Given a graph with positive edge weights and a subset of the the graph's vertices – the so-called terminals – the goal is to determine a minimum-weight, cycle-free subgraph of the input graph which connects all terminals.

We will have a look at the performance characteristics on real-world instances in the subsequent section. At this point, we first want to analyze the impact of our approach by means of randomly generated instances. We fix the number of terminals for each instance to ten randomly chosen ones and we use the same terminal vertices in each tree decomposition generated for an instance.

PERFECT DOMINATING SET
D-FLAT Random Instances

Minimum Improvement:	46.22 %	Average Improvement:	58.91 %
Maximum Improvement:	64.72 %	Median Improvement:	60.69 %

Statistical Significance:	$\geq$ 99.95 %

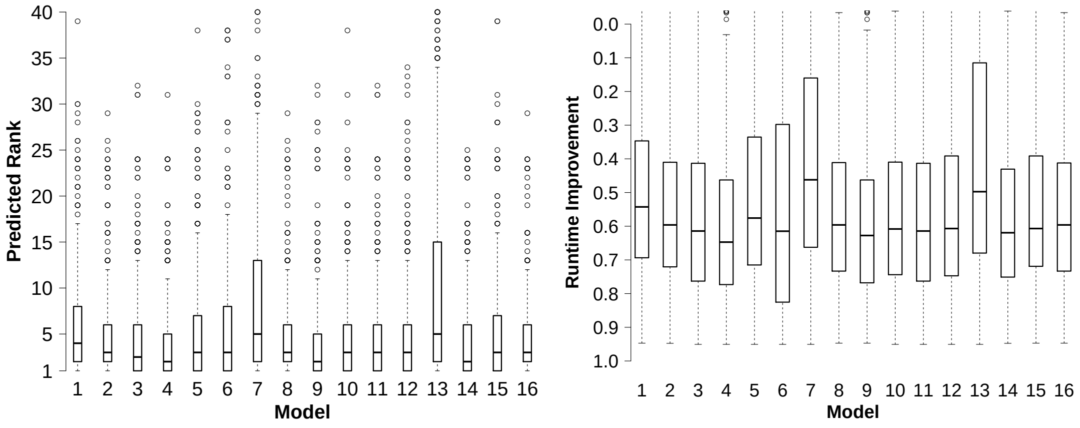


Figure 8 Performance Characteristics for PERFECT DOMINATING SET (D-FLAT)

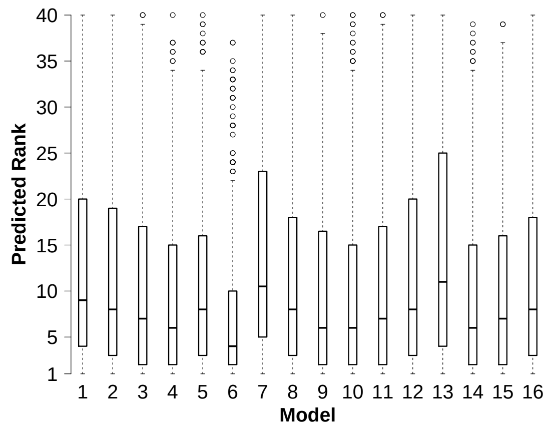
The predicted rank and the runtime savings achieved during our experiments indicate that our approach works well also for “hidden” information like the actual terminals which are in no way distinguished from the other vertices in the generated tree decompositions. Hence, the tree decomposition features are completely unaffected by this information and one could be tempted to think that this fact would make learning much harder or even impossible. That this is absolutely not the case is shown in Figure 10: Even the worst models – again Model 7 (PLSClassifier) holds the red lantern, while Model 13 (Bagging) is only slightly better – lead to runtime savings of about a third.

4.3 Experiments on Real-World Instances

Until now we only considered random instances in our experiments. With the goal of strengthening our findings, we additionally conducted a series of tests for STEINER TREE on real-world graphs.

The problem has many real-world applications like minimizing the effort (distance, time, etc.) to connect different terminals. While in some contexts the terminals are fixed for an input graph – one could for instance think of train stations – there are also situations where the set of terminal vertices changes frequently for the same input graph. One such example can be found in the area of predictive policing: In many cases the regions where crime is occurring more frequently is known but depending on the daytime and events taking place in the city these problematic regions can change rapidly. This is no problem as long as we can send officers to all the places, but often not enough personnel is available to do so. Therefore, there is the tendency to split the available

CONNECTED VERTEX COVER			
D-FLAT Random Instances			
Minimum Improvement:	22.38 %	Average Improvement:	32.91 %
Maximum Improvement:	42.78 %	Median Improvement:	34.49 %
Statistical Significance:	$\geq$ 99.95 %		



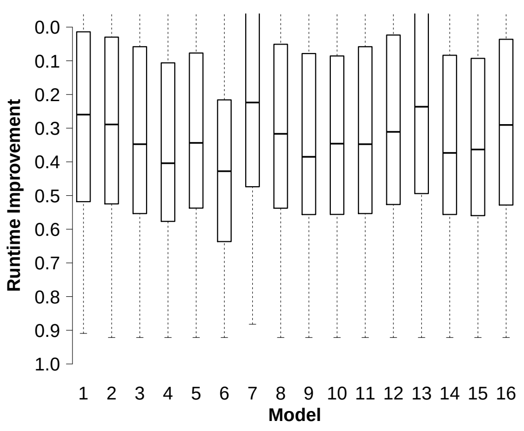


Figure 9 Performance Characteristics for CONNECTED VERTEX COVER (D-FLAT)

City	# Vertices	# Edges	Tree Decomposition Width			
			Min.	Max.	Avg.	Med.
Tokyo (JPN)	143	162	4	5	4.010	4
Osaka (JPN)*	145	160	4	5	4.334	4
Singapore (SGP)	101	114	4	5	4.487	4
Santiago (CHL)	127	138	4	6	4.218	4
Vienna (AUT)*	138	160	5	6	5.073	5

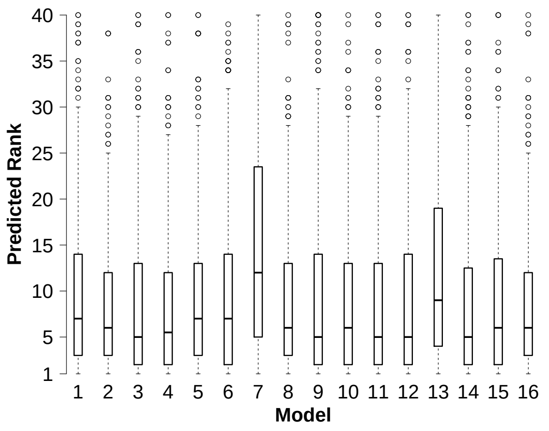
Table 2 Investigated Metro Systems (* ... Metro and Interurban Train)

officers into groups. One of the crucial problems to prepare for the case of an emergency is to maintain the ability to combine the forces with the least possible effort and this is exactly the point where the STEINER TREE problem comes into play.

The graphs chosen for these experiments are shown in Table 2 and represent the metro systems of some cities around the world. Compared to the random instances we have seen before, metro systems are much more structured. Often they have a more or less complex central region (covering the city center) and the remainder of the network is formed by simple paths (which are of width 1).

Apart from the name of the selected cities, Table 2 also shows the size of the corresponding network in terms of vertices and edges. Furthermore, the last four columns contain the minimum,

STEINER TREE - 10 TERMINALS			
D-FLAT Random Instances			
Minimum Improvement:	31.56 %	Average Improvement:	59.05 %
Maximum Improvement:	66.33 %	Median Improvement:	62.02 %
Statistical Significance:	≥ 99.95 %		



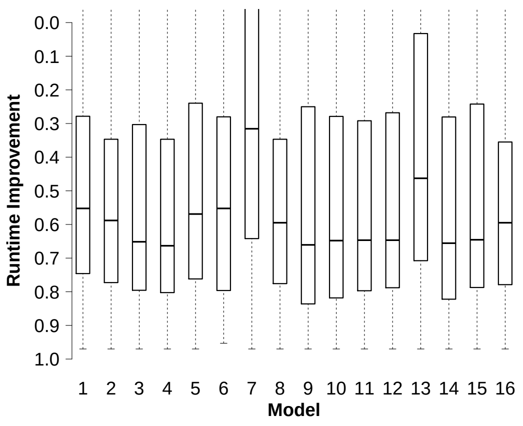


Figure 10 Performance Characteristics for STEINER TREE (D-FLAT)

maximum, mean and median value for the width over the 2800 benchmark runs for each of the cities. Note that the metro networks contain a higher number of vertices than the random graphs investigated in the previous section. Therefore a different configuration for D-FLAT is used at this point in order to avoid problems due to main memory limitations. We will see in Section 4.5 that this modified configuration does no harm to the generality of our proposed approach.

Finally, note that Table 2 also highlights the fact that most of the tree decompositions for the cities are of the same width and hence, the huge runtime differences we will observe especially on the metro systems of Singapore and Vienna cannot be explained by considering the width only.

The first city we have a look at is Tokyo. Although the capital of Japan is one of the metropolitan areas with the highest population in the whole world, the metro system (without the various interurban train lines) is decomposable into a tree decomposition of width 4. As distance measure we used the distance between adjacent stations rounded to full 100 meters.

D-FLAT was able to solve the Steiner Tree Problem with 10 (varying) terminal vertices in a time between 4 and 25 seconds for Tokyo. Although the absolute values are not overwhelming, Figure 11 tells us that we can easily save about a fifth of the total runtime using our approach. This is on the first glance an objection to the 60% runtime improvement we achieved for the random instances, but note that the runtime variance is low for Tokyo while in our random experiments we had differences (for the same instance) between some seconds and much more than one hour. When we look at the predicted rank we see that our findings from the previous sections indeed carry over to the evaluation of the Tokyo Metro at hand.

STEINER TREE - 10 TERMINALS			
D-FLAT Metro System of Tokyo			
Minimum Improvement:	15.38 %	Average Improvement:	21.45 %
Maximum Improvement:	24.44 %	Median Improvement:	22.40 %
Statistical Significance:		$\geq$ 99.95 %	

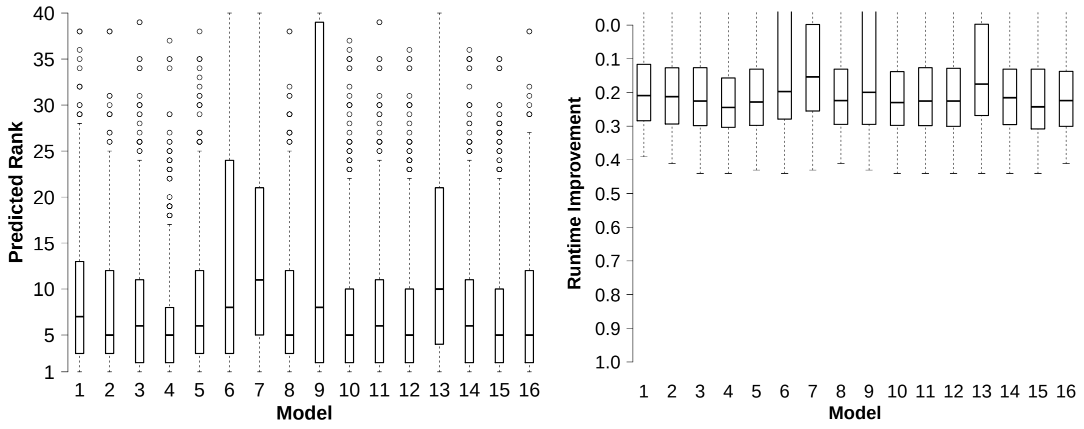


Figure 11 Performance Characteristics for STEINER TREE (Metro Tokyo)

The second city we want to present in our evaluation is Osaka, the third-largest city of Japan. Again, we used the distance between adjacent stations, rounded to full 100 meters, as the distance measure. For the metro system of Osaka together with the urban train network of the city, D-FLAT took between about 4 seconds and 10 minutes to compute the most optimal solutions for the given decomposition. Although the predicted rank is worse than for Tokyo – see especially Model 13 (Bagging) which actually performs worse than a random selection would do in this case – the higher variance in terms of runtime still allows us to achieve savings of about a quarter of the total runtime with the average model.

The next city on our list is Singapore. For this city we decided to use the travel time in minutes as the distance measure. As there are now much more partial solution candidates of equal cost present in each node of the tree decomposition, the runtime requirements for computing the optimal solutions are between 4 seconds and about half an hour. Figure 13 shows that Model 4 (LinearRegression) achieves high savings of 63% while Model 13 (Bagging) again fails a good prediction. Viewing the complete picture we can still conclude that the approach works very well.

The fourth city we investigated is Santiago, the capital of Chile. For this city, we took unit weights as the distance measure: Each hop from a station to another is penalized with the weight 1. Note that the Steiner Tree stays NP-complete also in the case of unit weights, as long as not all vertices are terminals. In this special case, the problem would collapse to the polynomial-time task

STEINER TREE - 10 TERMINALS			
D-FLAT Metro System of Osaka			
Minimum Improvement:	-31.68 %	Average Improvement:	19.92 %
Maximum Improvement:	29.71 %	Median Improvement:	24.37 %
Statistical Significance:	$\geq$ 99.95 %		

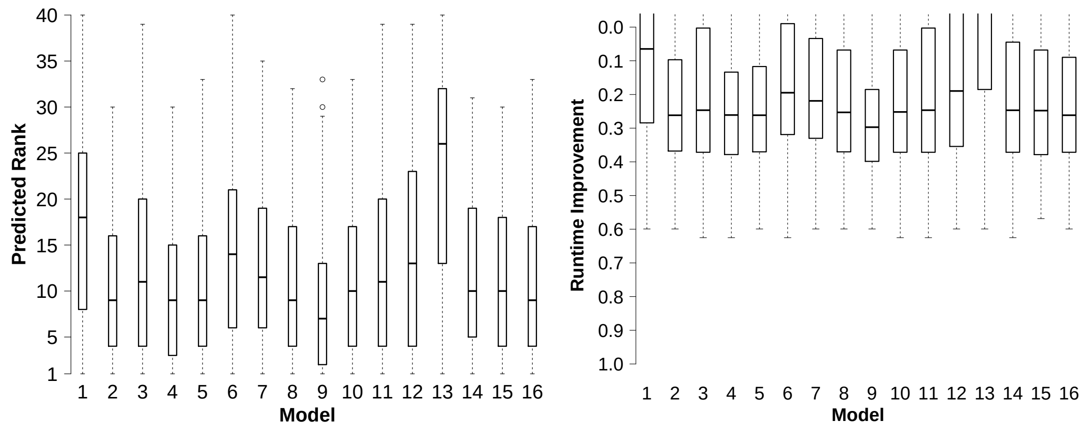


Figure 12 Performance Characteristics for STEINER TREE (Metro Osaka)

of computing the (minimum) spanning tree of the graph.

During our experiments, we observed runtimes between 3 seconds and 12.5 minutes. Another observation was the fact that the weights indeed influence the number of partial solution candidates for each bag as this number is relatively high for the unit-weight network at hand. This confirms our assumptions we made in the previous cases where the chances for equal weights are small. As we have seen in the section on the evaluation of random instances, this “hidden” information which is not accessible via the tree decomposition features makes learning potentially harder. In Figure 14 we can see that this theory is confirmed for most models, but again our approach pays off in most cases.

The last city in our evaluation is Vienna, the capital of Austria. Like for Osaka, we enhanced the metro network by combining it with the urban train system in order to increase the complexity. The distance measure we used in this case is the travel time in minutes. Not surprisingly, Vienna as the example of highest width leads to the highest runtime: The solving times underlying the evaluation at hand are distributed between 45 seconds and 12000 seconds.

Figure 15 presents the results we obtained for the metro and urban train system of the largest city in Austria. Even in the worst case of the 16 models we are able to save more than 47% of the total runtime while the average model leads to savings of about two thirds. Moreover, via this example we can see that also the Models 7 and 13 (PLSClassifier and Bagging), which are again under-performing in comparison to most of the other models, can lead to an extreme advantage

STEINER TREE - 10 TERMINALS			
D-FLAT Metro System of Singapore			
Minimum Improvement:	-213.81 %	Average Improvement:	31.49 %
Maximum Improvement:	63.68 %	Median Improvement:	50.00 %
Statistical Significance:		$\geq$ 99.95 %	

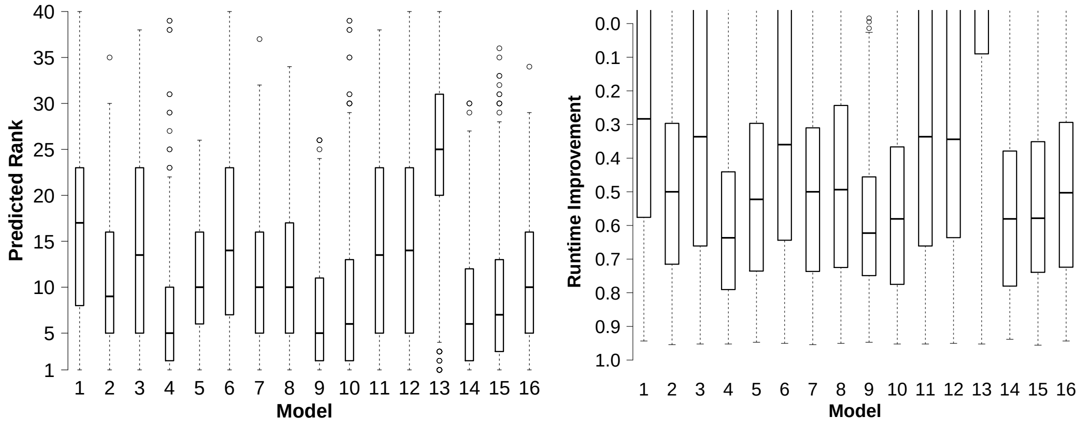


Figure 13 Performance Characteristics for STEINER TREE (Metro Singapore)

compared to taking a completely random tree decomposition.

By comparing our findings for Vienna with those for the other cities allowing a smaller width (and runtime), we can conclude that our proposed approach is most effective whenever the runtime differences are high. On the one hand, this seems to be by no means surprising because a small runtime variation implies a small potential for improvement. On the other hand, and this is what was indeed surprising for us, we observe that (most) machine learning algorithms can deal very well with "hidden" information like the number of partial solutions per tree decomposition node.

4.4 Model Evaluation

After we presented the thorough investigation of our approach on both random instances and real-world networks, we also want to present the results of our performance analysis separately for each model. Table 3 summarizes these outcomes. The table shows for each of the 16 models under investigation the median predicted rank over all evaluation runs for each of the problems. Note that for the column "MDS" we merge the results for D-FLAT and SEQUOIA, so that the cell content represents the median over 1000 evaluations. The column "ST (Real)" contains the median predicted rank over 2500 evaluations as we merge the results from the five cities under investigation. For all other problems, the cell content is computed by taking the median over 500 evaluations. The numbers in boldface highlight the best-performing models.

The last two rows (columns) then provide the mean and median of all preceding rows

STEINER TREE - 10 TERMINALS			
D-FLAT Metro System of Santiago			
Minimum Improvement:	-7.56 %	Average Improvement:	20.69 %
Maximum Improvement:	35.83 %	Median Improvement:	22.65 %
Statistical Significance:		$\geq$ 99.95 %	

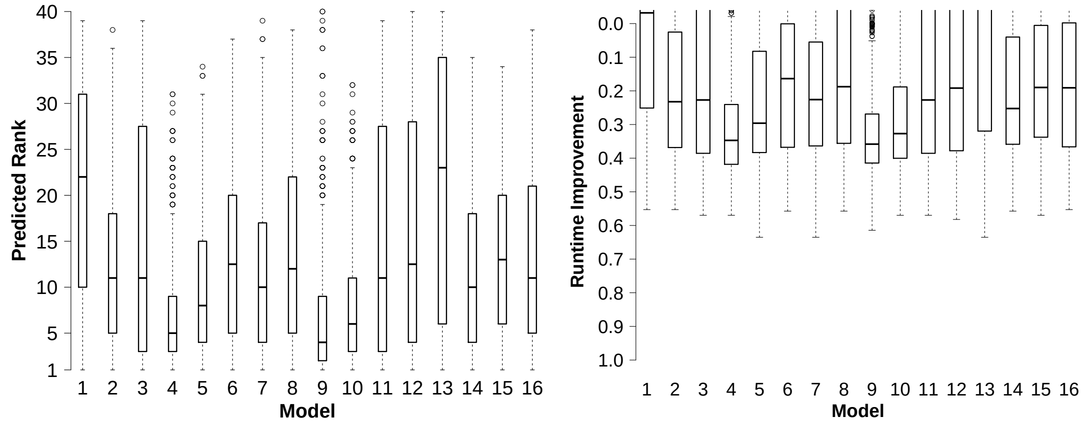


Figure 14 Performance Characteristics for STEINER TREE (Metro Santiago)

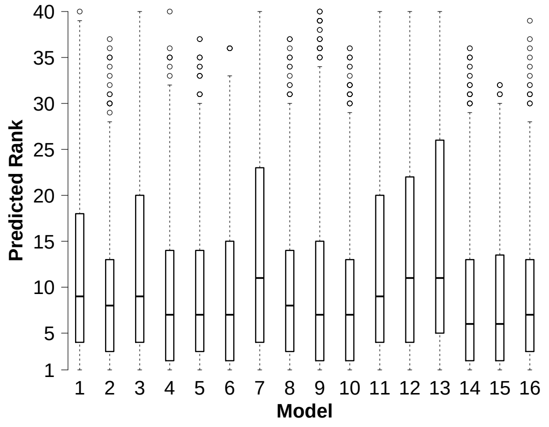
(columns). Note that the three cells in the bottom right corner are left empty because in these cases, the results will differ depending on whether one computes the median of means or the mean of medians. The same applies for the median of the medians, which also differs depending on whether we start with the column-wise or the row-wise median. What we can compute easily is the average predicted rank over the average model and all problems and this value is shown in the highlighted cell in the bottom right corner. We can see that in the average case we predict rank 7 out of 40, which is much better than the median rank of 20.5 and hence we can expect a important gain in terms of performance.

Even the machine learning algorithms holding the red lantern, Models 7 and 13 (PLSClassifier and Bagging), predict well in most cases. In fact, the only case in our whole experimental evaluation where our approach does not lead to an improvement in the average case is Model 13 (Bagging) on the real-world Steiner Tree Problem. In all other cases we actually achieve quite impressive prediction results. Especially noteworthy is Model 4 (LinearRegression) which is able to select a Top-5 rank in the average case of our experiments.

4.5 Inter-Domain Evaluation

Until this point of the paper it was the case that we investigated the applicability of our approach for each problem domain separately. For practical application scenarios it might be of importance (or at least of interest) to be able to adapt algorithms or change the application domain without

STEINER TREE - 10 TERMINALS			
D-FLAT Metro System of Vienna			
Minimum Improvement:	47.31 %	Average Improvement:	62.68 %
Maximum Improvement:	72.10 %	Median Improvement:	66.64 %
Statistical Significance:	$\geq$ 99.95 %		



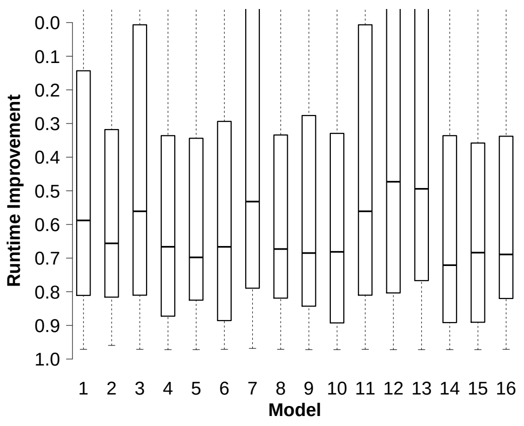


Figure 15 Performance Characteristics for STEINER TREE (Metro Vienna)

having to re-train the models one has already computed as this can be a time-consuming task.

Therefore we now want to have a deeper look at the inter-domain applicability of our approach. All the results are summarized in Table 4. The rows refer to the problem and (where applicable) solver that was used to generate the training data for the models and the columns stand for the evaluation dataset. The respective dataset are the same as for the domain-dependent experiments. The cells of the table then show the median value of the predicted rank over all 16 models. The rightmost two columns and the last two rows illustrate the mean and median over all problem domains, analogous to Table 3. The only difference is the fact that this time we give two results: The number on the top of the cells is the respective outcome over all domains while the second number represents the outcome computed with the diagonal excluded. This means that the first number gives the overall performance over all domains while the second one is the performance we observed on average in our setting when we switched the problem domain or solver.

We can see that in almost all cases (48 out of 49) we observe improved results. There is only one problematic situation, namely the case where we use the dataset obtained from solving STEINER TREE on random instances to predict the outcomes for 3-COLORABILITY. In this case we observed a slight deterioration of the predicted rank – On average, our models predicted rank 23.75 compared to the median rank 20.5. – compared to a random selection of the tree decomposition. As mentioned before, in all other cases we observed improvements which are statistically highly significant with a confidence level of over 99.95% and we have to keep in mind that these are the values for the average model, not for the best one. In the complete picture, summarizing all the

Model	Col	MDS	PDS	CVC	ST	ST (Real)	Average	Median
1 (GaussianProc.)	9	10	4	9	7	13	8.67	9.00
2 (IsotonicReg.)	7	8.5	3	8	6	8	6.75	7.50
3 (LeastMedSq)	6	5.5	2.5	7	**5**	10	6.00	5.75
4 (LinearReg.)	**5**	**5**	**2**	6	5.5	**6**	**4.92**	**5.25**
5 (ML-Perceptron)	11	10	3	8	7	8	7.83	8.00
6 (PaceReg.)	9	7	3	**4**	7	11	6.83	7.00
7 (PLSClassifier)	14	12	5	10.5	12	11	10.75	11.50
8 (SMOreg)	7	8	3	8	6	9	6.83	7.50
9 (IBk)	7	5.5	**2**	6	**5**	**6**	5.25	5.75
10 (KStar)	7	5.5	3	6	6	7	5.75	6.00
11 (LWL)	6	5.5	3	7	**5**	10	6.08	5.75
12 (AdditiveReg.)	6	6	3	8	**5**	10	6.33	6.00
13 (Bagging)	12	8	5	11	9	20	10.83	10.00
14 (CVPSelection)	6	5.5	**2**	6	**5**	7	5.25	5.75
15 (M5Rules)	6	6	3	7	6	7	5.83	6.00
16 (M5P)	7	8	3	8	6	8	6.67	7.50
Average	7.81	7.25	3.09	7.47	6.41	9.44	6.91	—
Median	7.00	6.50	3.00	7.50	6.00	8.5	—	—

Table 3 Predicted Ranks (Median) for Computed Models

positive impact of our approach, we have the fact that we were able to select rank 11 out of 40 on average (result shown in the highlighted cell in the bottom right corner of the table) which gives us important performance improvements compared to a random selection of the tree decomposition.

4.6 Discussion

As the evaluation underlines, our machine learning approach shows great potential for improving the performance of DP algorithms. As the width of the tree decompositions is the same for almost all decomposition for a given instance, this shows that the approach of minimizing only the width of the tree decompositions is not always sufficient and one needs a better way to select and/or to customize tree decompositions in order to improve the overall performance and especially the robustness of dynamic programming algorithms.

We can see that in general there is no "perfect" model which performs best in every case and that there exist differences between the problems. Our experiments also show that it is hard to

	CoL	MDS (D)	MDS (S)	PDS	CVC	ST	ST (Real)	Average	Median
CoL	7	11.75	13.5	12	11	19	11.5	12.25 13.13	11.75 11.88
MDS (D)	12.5	9	9.25	7	8	14	9.5	9.89 10.04	9.25 9.38
MDS (S)	10	9	4	5.5	6	13	9	8.07 8.75	9.00 9.00
PDS	20	10.75	13	3	6	6	12	10.11 11.29	10.75 11.38
CVC	16	8	8	5	7.5	8	11	9.07 9.33	8.00 8.00
ST	23.75	17.25	14	5	9.5	6	16	13.07 14.25	14.00 15.00
ST (Real)	16.5	15.5	14	10	13.5	16	8.5	13.43 14.25	14.00 14.75
Average	15.11 16.46	11.61 12.04	10.82 11.96	6.79 7.42	8.79 9.00	11.71 12.67	11.07 11.50	10.84 11.58	—
Median	16.00 16.25	10.75 11.25	13.00 13.25	5.50 6.25	8.00 8.75	13.00 13.50	11.00 11.25	—	—

Table 4 Predicted Ranks (Median) for Computed Models (Inter-Domain Evaluation)

predict the very best rank, but in many cases this is not needed. In general, every rank better than the median rank will increase the performance and the advantage grows with the runtime variance. Furthermore, in Section 4.5 we showed that one does not need to train models for each new problem and that one can achieve good results also by applying models trained in a completely different setting. This makes our approach even more applicable in real-world application scenarios as one does not necessarily have to re-train the model(s) when the problem domain changes or new constraints are added.

In our experiments it was the case that Model 4 (Linear Regression) performed best and that the Models 7 (PLSClassifier) and 13 (Bagging) showed the worst – but in many cases still relatively good – performance characteristics. We note that these are just the results for our evaluation scenarios at hand and that there might be other situations where these models would perform very different. This means that the selection of the regression algorithm and its corresponding parameter settings is of utmost importance when one aims at the maximum possible optimization quality.

5 The Approach in Practice

In this section we will provide a short sketch how our proposed approach can be applied in practice. We will do this by providing an example scenario where we want to solve some arbitrary problem, let's call it X, using the solver D-FLAT. For computing the regression model(s) we use the machine learning toolkit provided by WEKA. Subsequently one can find the recipe on which also the evaluation was based.

1. Write a D-FLAT problem encoding for the DP of problem X or take an existing one. As a starting point, some examples for problem encodings can be found under the following link: `https://github.com/bbliem/dflat/tree/master/applications`

2. Take a representative set of instances of X and convert them to a format appropriate for the previously created encoding. The size of this training set is dependent on the future application area, e.g., in the case that one only deals with a fixed instance (like it was the case in our real-world scenario), a set consisting of only this single instance completely suffices.

3. Compute the solutions and store the runtime together with the random seed. Note that a crucial requirement for our approach is the presence of multiple runs for each of the instances, because otherwise computing the standardized results is rendered useless and no learning is possible. As a rough guideline, make sure that the full training set obtained using this process contains at least 50 to 100 benchmark runs and at least 10 runs with different tree decompositions for each training instance.

   Note that the actual solutions are not relevant in this step. For D-FLAT we selected counting mode which can be enabled via the parameter `--depth 0` and outputs only the number of solutions. Actually, taking this mode is required in D-FLAT for the learning phase because printing the solutions often consumes a significant portion of time and would bias the measurements.

4. Take the recorded random seeds to compute again the corresponding tree decompositions. This time, we don't want to solve the problem. Instead we extract the complete feature set and store each of the values in a table where each feature is assigned to a separate column. Additionally, the generated table contains for each tree decomposition the runtime of the DP algorithm.

5. Clean the result table obtained in the previous step. This means that we have to get rid of all unsatisfiable instances and we have to delete benchmark runs leading to timeouts or which exceeded the memory limit. Furthermore, we have to ensure that each instance finally has the same number of benchmark runs. This is most easily done by introducing a fixed threshold: Every instance with a number of successful runs less than the threshold is deleted and for those instances with more successful than the threshold we successively delete a randomly selected benchmark run until the threshold is met exactly.

6. Standardize the values column-wise and grouped by the problem instance.

7. Choose one (or more) of WEKA's regression algorithms and train it using the table obtained from the above procedure. The class attribute must be set appropriately as we clearly want to predict the runtime. During the computation of the model, WEKA will try to maximize the prediction accuracy in terms of several parameters, like correlation ratio or different error measures. Make sure that you try out algorithms and different parameter combinations because it is by no means trivial to find the algorithm with optimal prediction quality.

8. For a new instance of problem X, generate some tree decompositions, extract their features, standardize them and use the generated model to predict the runtime. All that remains is to select the decomposition with minimal predicted runtime and you will very likely end up with a significantly reduced actual runtime compared to a randomly chosen decomposition of the same width.

6 Conclusion

In this work we studied the applicability of machine learning techniques to improve the runtime behavior of DP algorithms based on tree decompositions. To this end we identified a variety of tree decomposition features, beside the width, that strongly influence the runtime behavior. Machine learning models using those features for the selection of the optimal decomposition have been validated by means of an extensive experimental analysis including real-world instances. In our experiments we considered five different problems and our approach showed a remarkable, positive effect on the performance with a high statistical significance. We thus conclude that turning the huge body of theoretical work on tree decompositions and dynamic programing into efficient systems highly depends on the quality of the chosen tree decomposition, and that advanced selection mechanisms for finding good decompositions are indeed crucial.

The presented work, however, is only a first step of a larger research project. In a next step, we have to investigate whether the models we obtained will give further insights about those features of tree decompositions that are most influential in order to reduce the runtime of DP algorithms. Such insights then will be used to design and implement new heuristics for constructing tree decompositions that optimize the relevant features. Therefore, the ultimate goal of this research perspective is to achieve the potential speed-up we have observed in our experiments by directly obtaining tree decompositions of higher quality and thus *without* the initial training step our method requires.

References

[Abseher *et al.*, 2014] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. The D-FLAT System for Dynamic Programming on Tree Decompositions. In *European Conference On Logics In Artificial Intelligence (JELIA)*, volume 8761 of *Lecture Notes in Artificial Intelligence*, pages 558–572. Springer, 2014.

[Abseher *et al.*, 2015] Michael Abseher, Frederico Dusberger, Nysret Musliu, and Stefan Woltran. Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 275–282. AAAI Press, 2015.

[Arnborg and Proskurowski, 1989] 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.

[Arnborg *et al.*, 1987] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[Bachoore and Bodlaender, 2006] Emgad H. Bachoore and Hans L. Bodlaender. A Branch and Bound Algorithm for Exact, Upper, and Lower Bounds on Treewidth. In *Algorithmic Aspects in Information and Management: Second International Conference, AAIM 2006*, volume 4041 of *Lecture Notes in Computer Science*, pages 255–266. Springer, 2006.

[Bertelè and Brioschi, 1973] Umberto Bertelè and Francesco Brioschi. On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A*, 14(2):137–148, 1973.

[Bliem *et al.*, 2013] Bernhard Bliem, Reinhard Pichler, and Stefan Woltran. Declarative Dynamic Programming as an Alternative Realization of Courcelle's Theorem. In *International Symposium on Parameterized and Exact Computation (IPEC)*, volume 8246 of *Lecture Notes in Computer Science*, pages 28–40. Springer, 2013.

[Bodlaender and Koster, 2008] Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial Optimization on Graphs of Bounded Treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[Bodlaender and Koster, 2010] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[Brewka *et al.*, 2011] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[Clautiaux *et al.*, 2004] François Clautiaux, Aziz Moukrim, Stèfane Négre, and Jaques Carlier. Heuristic and meta-heurisistic methods for computing graph treewidth. *RAIRO Operational Research*, 38:13–26, 2004.

[Downey and Fellows, 1999] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[Gogate and Dechter, 2004] Vibhav Gogate and Rina Dechter. A Complete Anytime Algorithm for Treewidth. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence, UAI 2004*, pages 201–208. AUAI Press, 2004.

[Gutin, 2015] Gregory Gutin. Should we care about huge imbalance in parameterized algorithmics? *The Parameterized Complexity Newsletter*, December 2015.

[Halin, 1976] Rudolf Halin. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976.

[Hall *et al.*, 2009] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Exploration Newsletters*, 11(1):10–18, 2009.

[Hammerl and Musliu, 2010] Thomas Hammerl and Nysret Musliu. Ant colony optimization for tree decompositions. In *Proceedings of the 10th European Conference on Evolutionary Computation in Combinatorial Optimization, EvoCOP 2010*, Lecture Notes in Computer Science, pages 95–106. Springer, 2010.

[Hammerl *et al.*, 2015] Thomas Hammerl, Nysret Musliu, and Werner Schafhauser. Metaheuristic Algorithms and Tree Decomposition. In *Handbook of Computational Intelligence*. Springer, 2015.

[Hintikka, 1973] Jaakko Hintikka. *Logic, Language-games and Information: Kantian Themes In the Philosophy of Logic*. Oxford: Clarendon Press, 1973.

[Hutter *et al.*, 2014] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014.

[Jégou and Terrioux, 2014] Philippe Jégou and Cyril Terrioux. Bag-Connected Tree-Width: A New Parameter for Graph Decomposition. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2014*, pages 12–28, 2014.

[Kanda *et al.*, 2011] Jorge Kanda, André Carvalho, Eduardo Hruschka, and Carlos Soares. Selection of algorithms to solve traveling salesman problems using meta-learning. *International Journal of Hybrid Intelligent Systems*, 8(3):117–128, 2011.

[Kjaerulff, 1992] Uffe Kjaerulff. Optimal decomposition of probabilistic networks by simulated annealing. *Statistics and Computing*, 2(1):2–17, 1992.

[Kloks, 1994] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

[Kneis *et al.*, 2011] Joachim Kneis, Alexander Langer, and Peter Rossmanith. Courcelle's Theorem - A Game-Theoretic Approach. *Discrete Optimization*, 8(4):568–594, 2011.

[Koster *et al.*, 1999] Arie M. C. A. Koster, Stan P. M. van Hoesel, and Antoon W. J. Kolen. Solving Frequency Assignment Problems via Tree-Decomposition 1. *Electronic Notes in Discrete Mathematics*, 3:102–105, 1999.

[Kotthoff, 2014] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, 35(3):48–60, 2014.

[Langer *et al.*, 2014] Alexander Langer, Felix Reidl, Peter Rossmanith, and Somnath Sikdar. Practical algorithms for MSO model-checking on tree-decomposable graphs. *Computer Science Review*, 13–14:39–74, 2014.

[Larranaga *et al.*, 1997] Pedro Larranaga, Cindy M.H. Kujipers, Mikel Poza, and Roberto H. Murga. Decomposing bayesian networks: Triangulation of the moral graph with genetic algorithms. *Statistics and Computing*, 7 (1):19–34, 1997.

[Lauritzen and Spiegelhalter, 1988] Steffen L. Lauritzen and David J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society, Series B*, 50:157–224, 1988.

[Leyton-Brown *et al.*, 2009] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical Hardness Models: Methodology and a Case Study on Combinatorial Auctions. *Journal of the ACM*, 56(4):1–52, 2009.

[Mersmann *et al.*, 2013] Olaf Mersmann, Bernd Bischl, Heike Trautmann, Markus Wagner, Jakob Bossek, and Frank Neumann. A novel feature-based approach to characterize algorithm performance for the traveling salesperson problem. *Annals of Mathematics and Artificial Intelligence*, 69(2):151–182, 2013.

[Morak *et al.*, 2012] Michael Morak, Nysret Musliu, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. Evaluating Tree-Decomposition Based Algorithms for Answer Set Programming. In *Learning and Intelligent Optimization Conference, LION 2012*, volume 7219 of *Lecture Notes in Computer Science*, pages 130–144. Springer, 2012.

[Musliu and Schafhauser, 2007] Nysret Musliu and Werner Schafhauser. Genetic algorithms for generalized hypertree decompositions. *European Journal of Industrial Engineering*, 1(3):317–340, 2007.

[Musliu and Schwengerer, 2013] Nysret Musliu and Martin Schwengerer. Algorithm Selection for the Graph Coloring Problem. In *Learning and Intelligent Optimization Conference, LION 2013*, volume 7997 of *Lecture Notes in Computer Science*, pages 389–403. Springer, 2013.

[Musliu, 2008] Nysret Musliu. An iterative heuristic algorithm for tree decomposition. *Studies in Computational Intelligence, Recent Advances in Evolutionary Computation for Combinatorial Optimization*, 153:133–150, 2008.

[Niedermeier, 2006] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford Lecture Series in Mathematics And Its Applications. Oxford University Press, 2006.

[Pihera and Musliu, 2014] Josef Pihera and Nysret Musliu. Application of machine learning to algorithm selection for TSP. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014*, pages 47–54, 2014.

[Robertson and Seymour, 1984] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Ser. B*, 36(1):49–64, 1984.

[Shoikhet and Geiger, 1997] Kirill Shoikhet and Dan Geiger. A Practical Algorithm for Finding Optimal Triangulations. In *AAAI/IAAI*, pages 185–190. AAAI Press / The MIT Press, 1997.

[Smith-Miles *et al.*, 2010] Kate Smith-Miles, Jano I. van Hemert, and Xin Yu Lim. Understanding TSP Difficulty by Learning from Evolved Instances. In *Learning and Intelligent Optimization Conference, LION 2010*, volume 6073 of *Lecture Notes in Computer Science*, pages 266–280. Springer, 2010.

[Smith-Miles *et al.*, 2013] Kate Smith-Miles, Brendan Wreford, Leo Lopes, and Nur Insani. Predicting Metaheuristic Performance on Graph Coloring Problems Using Data Mining. In *Hybrid Metaheuristics*, volume 434 of *Studies in Computational Intelligence*, pages 417–432. Springer, 2013.

[Smith-Miles, 2008] Kate Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys*, 41(1):6:1–6:25, 2008.

[Tarjan and Yannakakis, 1984] R.E. Tarjan and M. Yannakakis. Simple linear-time algorithm to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13:566–579, 1984.

[Xu *et al.*, 2005] Jinbo Xu, Feng Jiao, and Bonnie Berger. A tree-decomposition approach to protein structure prediction. In *Computational Systems Bioinformatics Conference, CSB 2005*, pages 247–256, 2005.

[Xu *et al.*, 2008] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.