Journal of Artificial Intelligence Research 58 (2017) 829-858 Submitted 08/16; published 04/17

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

Michael Abseher ABSEHER@DBAI.TUWIEN.AC.AT
Nysret Musliu MUSLIU@DBAI.TUWIEN.AC.AT
Stefan Woltran WOLTRAN@DBAI.TUWIEN.AC.AT
Institute of Information Systems 184/2
TU Wien
Favoritenstraße 9–11, 1040 Vienna, Austria

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; and (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. Thus, one main contribution of this work is to propose novel features for tree decompositions. Moreover, 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. Introduction

The notion of treewidth – and, as basic underlying concept, tree decompositions – was introduced in the work of 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 & Proskurowski, 1989; Niedermeier, 2006; Bodlaender & Koster, 2008). A famous result by Courcelle (1990), which became known as Courcelle's theorem, states that any graph property expressible in monadic second-order logic can be decided in linear time with respect to the parameter treewidth. A similar result was developed independently by Borie et al. (1992). A problem exhibiting tractability by bounding some problem-inherent constant is also called fixed-parameter tractable (FPT) (Downey & Fellows, 1999). While constructing an optimal tree decomposition, i.e. a decomposition with minimal width, is intractable (Arnborg, Corneil, & Proskurowski, 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. The approaches by Shoikhet and Geiger (1997), Gogate and Dechter (2004) as well as by Bachoore and Bodlaender (2006) are examples of exact algorithms for computing tree decompositions. Greedy heuristic algorithms include Maximum Cardinality Search (MCS) (Tarjan & Yannakakis, 1984), Min-fill heuristic (Dechter, 2003), and Minimum Degree heuristic (Berry, Heggernes, & Simonet, 2003), to mention just a few. Metaheuristic techniques have been provided in terms of genetic algorithms (Larranaga, Kujipers, Poza, & Murga, 1997; Musliu & Schafhauser, 2007), ant colony optimization (Hammerl & Musliu, 2010), and local search based techniques (Kjaerulff, 1992; Clautiaux, Moukrim, Négre, & Carlier, 2004; Musliu, 2008). A more detailed description of tree decomposition techniques is given in the recent surveys (Bodlaender & Koster, 2010; Hammerl, Musliu, & Schafhauser, 2015).

A promising technique for solving problems using graph decompositions 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 subproblems (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 over width k of the tree decomposition used. In fact, this approach has been used for several applications, including inference in probabilistic networks (Lauritzen & Spiegelhalter, 1988), frequency assignment (Koster, van Hoesel, & Kolen, 1999), computational biology (Xu, Jiao, & Berger, 2005), logic programming (Morak, Musliu, Pichler, Rümmele, & Woltran, 2012) and the Steiner Tree problem (Fafianie, Bodlaender, & Nederlof, 2015).

From a theoretical point of view, the actual width k is the crucial parameter towards efficiency for FPT algorithms that use tree decompositions. 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.

Due to the fact that algorithms based on dynamic programming on tree decompositions are often very sensitive to the shape of the actually used tree decomposition, Bodlaender and Fomin (2005) introduced the concept of tree decompositions of small cost. The cost associated to a node of the tree decomposition is defined based on a function f which maps the bag size of a tree decomposition node to a real number according to the assumed time (or memory) complexity of the problem at hand. The total f-cost of a tree decomposition is then the sum over all evaluations of the formula f for the nodes of the given tree decomposition. Hence, the work by Bodlaender and Fomin allows to distinguish tree decompositions in a more fine-grained way than just by the width and in their article an extensive theoretical analysis of the problem of finding tree decompositions of minimum f-cost is provided. In the considerations made by Bodlaender and Fomin, the bag size is the only input for a function that estimates the costs of a given tree decomposition node in the context of a DP algorithm. Although no experimental evaluation is provided in the paper by Bodlaender and Fomin, it is assumed that considering all bags (and not just the largest one, i.e., the bag from which the width is derived) allows to better estimate the runtime of DP algorithms.

In our paper at hand we now go one step further by considering much more features of tree decompositions for the estimation function for the runtime. More precisely, in this article we want to gain a deeper understanding of the impact of tree decompositions on the actual runtime of DP algorithms by employing machine learning techniques. Recently, researchers have successfully used machine learning for runtime prediction and algorithm selection on several problem domains including SAT (Xu, Hutter, Hoos, & Leyton-Brown, 2008; Hutter, Xu, Hoos, & Leyton-Brown, 2014), combinatorial auctions (Leyton-Brown, Nudelman, & Shoham, 2009), TSP (Smith-Miles, van Hemert, & Lim, 2010; Kanda, Carvalho, Hruschka, & Soares, 2011; Mersmann, Bischl, Trautmann, Wagner, Bossek, & Neumann, 2013; Pihera & Musliu, 2014; Hutter et al., 2014), graph coloring (Smith-Miles, Wreford, Lopes, & Insani, 2013; Musliu & Schwengerer, 2013), etc. Surveys on this topic are provided, for instance, by Smith-Miles (2008), Hutter et al. (2014) or 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 allow 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 a state-of-the-art system that applies DP algorithms on tree decompositions, namely D-FLAT (Abseher, Bliem, Charwat, Dusberger, Hecher, & Woltran, 2014). D-FLAT is a problem-independent general-purpose framework designed for (relatively) easy prototyping of various DP algorithms.

The complete picture of our evaluation shows a significant benefit of selecting a decomposition that is promising according to the prediction in contrast to simply choosing an arbitrary one. Furthermore, the results confirm that our approach is generally applicable and independent from the particular 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 a conference paper (Abseher, Dusberger, Musliu, & Woltran, 2015). In particular, we consider here additional features and completely rearrange the experiments. We now 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. Finally, we also give a thorough evaluation of the sixteen models and provide the results of an inter-domain evaluation of our approach. An additional evaluation conducted with another tree-decomposition based system, namely SEQUOIA (Kneis, Langer, & Rossmanith, 2011), can be found in an accompanying technical report (Abseher, Musliu, & Woltran, 2016). The results for the SEQUOIA system are very similar to those we observe in the article at hand. Although in the case of SEQUOIA, the variation in terms of solving time between different random seeds (and hence, different tree decompositions) for the same problem instance is relatively small, predicting the runtime of DP algorithms based on tree decomposition features works very well. As the prediction seems to be more complicated in the case of D-FLAT and also due to the fact that the edge weights used in the context of the Steiner Tree problem are not handled by SEQUOIA, we focus in this paper on D-FLAT as here the learning task seems to be more involved.

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 on random input data and real-world instances. Section 5 finally concludes our article.

2. Background

In the following we give a formal definition of tree decompositions and treewidth, and we illustrate the principle of DP on such decompositions. Furthermore, we provide a short overview on D-FLAT (Abseher et al., 2014).

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 & Seymour, 1984; Bodlaender & 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 $\chi : N \to 2^V$ 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 a root $r \in N$ is explicitly defined. Figure 1 shows a graph and one of its (non-normalized) tree decompositions.

Definition 2 *Given a graph $G = (V, E)$, a* normalized *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 $\chi_i = \chi_j = \chi_k$*

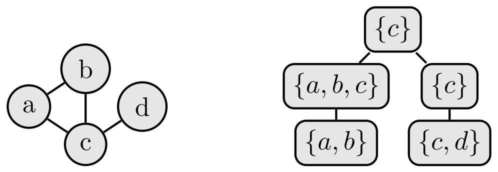
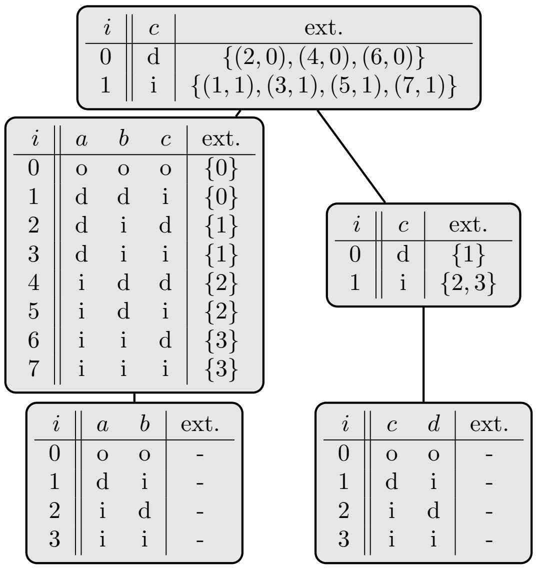


Figure 1: Example Graph and a Possible Tree Decomposition



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 Set via DP for the Problem Instance in Figure 1

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 the root node are consistent to the solutions for the whole problem. The key aspect of FPT algorithms is to bound the costs for computing these solutions by the width of the given tree decomposition. The complete solutions can be obtained (with polynomial delay) in a reverse traversal from the root node to the leaves combining the computed partial solutions.

Example 1 *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*

1. Given a graph (V, E), find all sets $S \subseteq V$ such that for all $u \in V$ either $u \in S$ or there is an edge $(u, v) \in E$ with $v \in S$.

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 $\{c = o, d = o\}$ 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 decomposition and following the extension pointers down to the leaves. One solution here is given 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.

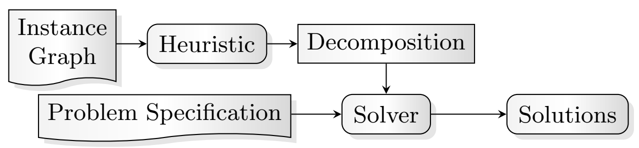
As mentioned before, in our experiments we use a state-of-the-art system that applies DP on tree decompositions, namely D-FLAT. D-FLAT (Abseher et al., 2014) is a general framework capable of solving any problem expressible in monadic second-order logic (MSO) in FPT time w.r.t. the parameter treewidth (Bliem, Pichler, & Woltran, 2013). The D-FLAT system combines DP on tree decompositions with answer set programming (Brewka, Eiter, & Truszczyński, 2011) which is used to specify the DP algorithm. Given an encoding Π of the DP algorithm and an instance graph G, D-FLAT first constructs a tree decomposition of G in polynomial time using internal heuristics. If desired, this decomposition can be left untouched or be normalized before running the DP algorithm (it is common practice to define DP algorithms on normalized tree decompositions (Bodlaender & Koster, 2008; Kneis et al., 2011); thus, in the following, we refer to normalized decompositions unless stated otherwise). 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.

3. Improving the Efficiency of DP Algorithms

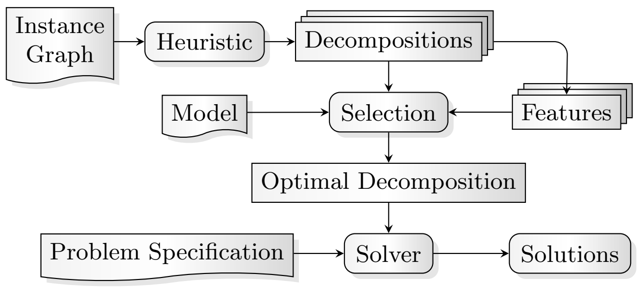
Systems such as D-FLAT follow a straight-forward approach for using tree decompositions: a single decomposition is generated heuristically and then fed into the DP algorithm used in the system (see Figure 3a). However, experiments have shown that the "quality" of such tree decompositions varies, leading to significantly differing runtimes for the same problem instance. Most interestingly, "quality" in this context does not necessarily mean low width. Even tree decompositions of exactly the same width lead to huge differences in the observed solving times. For instance, in our experiments for the Steiner Tree problem with real-world instances (see Section 4.3) we observe runtimes between 67 seconds and around two hours for the problem instance `vienna/metro_10terminals_46` for which all the tree decompositions used in our evaluation are of width 5.

3.1 Automated Selection of Tree Decompositions

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:



(a) Standard Approach



(b) Improved Approach

Figure 3: Comparison of Approaches

- Generating a number of tree decompositions for a given input graph can be usually done very efficiently by employing sophisticated heuristics for tree decompositions like, e.g., Min-Fill heuristic (Dechter, 2003), thus the runtime overhead will be negligible in most 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*.

- 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 apply learning techniques we need to find powerful features that characterize 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 to generate the models for prediction. Finally, the selected decomposition is handed over to the actual system to run the DP algorithm.

Note at this point that the model(s) as well as the features are crucial ingredients for the applicability of our approach in practice. Indeed, it should be possible to compute each feature efficiently. Furthermore, it can be a time-consuming task to train such a regression model. Fortunately, as we will show in Section 4.5, it seems that models which were trained for some specific problem domain can often be re-used in different application scenarios. In the following section, we will propose a set of several tree decomposition features which are all computable in polynomial time. In many cases, this is possible even in linear time.

3.2 Tree Decomposition Features

In what follows we address one of the main contributions of the work, namely the identification of new tree decomposition features. Subsequently we will give for each feature a short description and formal specification. Providing multiple statistical key figures like minimum, maximum, mean and median leads us to a total of 144 features.

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 $i \in N$ has associated a type $t_i \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\}$. Furthermore, we define the sets *Leaf*, *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 *NonLeaf* is defined as $N \setminus \mathit{Leaf}$ and the set *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 $\mathit{distance}(i, j)$. 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 $v \in \chi_i$. Some of the more elaborate features require information about the neighborhood and the reachability relation. For this reason we define the following functions:

- $\mathit{neighbors}(v)$ returns the set of neighbors of vertex v (excluding v) in G;
- $\mathit{adjacent}(u, v)$ returns 1 whenever vertices u and v are adjacent in G and 0 otherwise;
- $\mathit{reachable}(u, v)$ returns 1 whenever v can be reached from u in G 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) $\}$
- $\mathit{Agg2} = \mathit{Agg1} \setminus \{\mathit{count}\}$

The proposed features are described below.

3.2.1 BagSize, NonLeafNodeBagSize, NonEmptyNodeBagSize

These feature shall capture the complexity of the tree decomposition by recording the size of the bags. We have for each $\alpha \in \mathit{Agg1}$

$$\begin{aligned} \mathit{BagSize}^{\alpha} &= \alpha(\{|\chi_i| : i \in N\}) \\ \mathit{BagSize}^{\alpha}_{t} &= \alpha(\{|\chi_i| : i \in t\}) \qquad \text{for } t \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\} \\ \mathit{BagSize}^{\alpha}_{\mathit{NonLeaf}} &= \alpha(\{|\chi_i| : i \in \mathit{NonLeaf}\}) \\ \mathit{BagSize}^{\alpha}_{\mathit{NonEmpty}} &= \alpha(\{|\chi_i| : i \in \mathit{NonEmpty}\}) \end{aligned}$$

Additionally, to cover the overall size of the decomposition, we have

$$\begin{aligned} \mathit{CumulativeBagSize} &= \sum_{i \in N} |\chi_i| \\ \mathit{CumulativeBagSize}_t &= \sum_{i \in t} |\chi_i| \qquad \text{for } t \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\} \\ \mathit{DecompositionOverheadRatio} &= \left(\sum_{i \in N} |\chi_i|\right) / |V| \end{aligned}$$

3.2.2 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 $\alpha \in \mathit{Agg2}$ we have

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

Note that $\mathit{ContainerCount}^{mean} = \mathit{CumulativeBagSize}/|V| = \mathit{DecompositionOverheadRatio}$. In this special case, the features are indeed equivalent. In practice, one can avoid redundancy by using only a single one of these measures.

3.2.3 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 $\alpha \in \mathit{Agg2}$ we have

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

3.2.4 NodeDepth, NonLeafNodeDepth, NonEmptyNodeDepth

These features measure the distance from the root node to the node under focus. For each $\alpha \in \mathit{Agg2}$ we have

$$\begin{aligned} \mathit{NodeDepth}^{\alpha} &= \alpha(\{l_i : i \in N\}) \\ \mathit{NodeDepth}^{\alpha}_{t} &= \alpha(\{l_i : i \in t\}) \qquad \text{for } t \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\} \\ \mathit{NodeDepth}^{\alpha}_{\mathit{NonLeaf}} &= \alpha(\{l_i : i \in \mathit{NonLeaf}\}) \\ \mathit{NodeDepth}^{\alpha}_{\mathit{NonEmpty}} &= \alpha(\{l_i : i \in \mathit{NonEmpty}\}) \end{aligned}$$

3.2.5 Percentage

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

$$\mathit{Percentage}_t = |\{i : i \in t\}|/|N| \qquad \text{for } t \in \{\mathit{Leaf}, \mathit{Introduce}, \mathit{Forget}, \mathit{Join}\}$$

3.2.6 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

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

3.2.7 BranchingFactor

This feature measures the number of children for each node within the tree decomposition. For each $\alpha \in \mathit{Agg2}$ we have

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

3.2.8 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. For each $\alpha \in \mathit{Agg2}$ we have

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

3.2.9 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

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

3.2.10 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

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

3.2.11 [Introduced | Forgotten]VertexNeighborCount

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, based on the introduced (forgotten) vertices X_i for bags i

$$NC^{\alpha} = \alpha\left(\{|\mathit{neighbors}(v) \cap \chi_i| : v \in X_i, i \in N_v\}\right)$$

3.2.12 [Introduced | Forgotten]VertexConnectednessFactor

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, based on the introduced (forgotten) vertices X_i for bags i

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

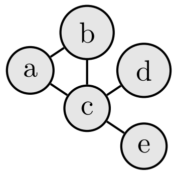
Example 2 *Consider graph G as shown in Figure 4. That figure also provides two different, normalized tree decompositions for G. We can see that both decompositions TD1 and TD2 have a maximum bag size of 3; hence they have exactly the same width – namely 2 – but their actual difference is clearly reflected by the proposed features, see Table 1. For brevity we only provide the median for those features where multiple aggregates are applied.*

Note that the set of tree decomposition features we present here shall act as a starting point. It contains various features which quantify the structural parameters of a given tree decomposition, but it also includes several measurements which are related to the general runtime behavior of dynamic programming algorithms, like, e.g., the neighborhood-related features. Although we focus solely on tree decomposition features in this work, considering also problem-specific features may improve prediction quality in concrete scenarios. In the following experiments we will use the complete set of our proposed features, but one can also try to drop some of them, e.g., by using well-established feature selection techniques from the area of machine learning (for an overview of feature selection approaches see, e.g., Guyon and Elisseeff, 2003, or Chandrashekar and Sahin, 2014).

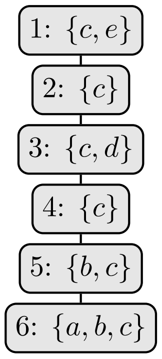
4. Experimental Evaluation

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.

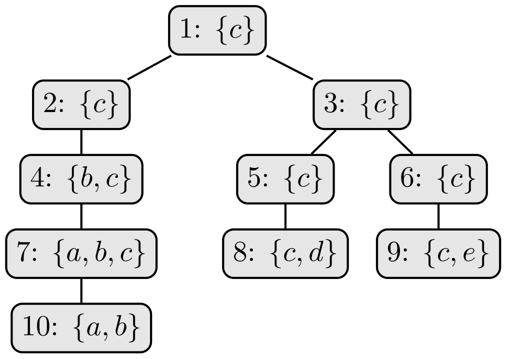
We evaluate our approach using a recently developed DP solver, D-FLAT (v. 1.0.1). The machine learning tasks were carried out with WEKA 3.6.13 (Hall, Frank, Holmes,



(a) Example graph G



(b) *TD1*



(c) *TD2*

Figure 4: Two Normalized Tree Decompositions *TD1* and *TD2* for Graph G

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
$JoinNodeDistance^{median}$	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

Pfahringer, Reutemann, & Witten, 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

Subsequently, we give details of the setup on which the experimental evaluation is based.

4.1.1 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.

4.1.2 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.

1. GaussianProcesses (weka.classifiers.functions.GaussianProcesses)
2. IsotonicRegression (weka.classifiers.functions.IsotonicRegression)
3. LeastMedSq (weka.classifiers.functions.LeastMedSq)
4. LinearRegression (weka.classifiers.functions.LinearRegression)
5. MultilayerPerceptron (weka.classifiers.functions.MultilayerPerceptron)
6. PaceRegression (weka.classifiers.functions.PaceRegression)
7. PLSClassifier (weka.classifiers.functions.PLSClassifier)
8. SMOreg (weka.classifiers.functions.SMOreg)
9. IBk (weka.classifiers.lazy.IBk)
10. KStar (weka.classifiers.lazy.KStar)
11. LWL (weka.classifiers.lazy.LWL)
12. AdditiveRegression (weka.classifiers.meta.AdditiveRegression)
13. Bagging (weka.classifiers.meta.Bagging)
14. CVParameterSelection (weka.classifiers.meta.CVParameterSelection)
15. M5Rules (weka.classifiers.rules.M5Rules)
16. M5P (weka.classifiers.trees.M5P)

The initial evaluation which was used to find the exact configuration for the regression algorithms 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 available in WEKA we experimented with different values and used 10-fold cross validation to determine the performance of each configuration. The fixed parameters that can be found in the report by Abseher et al. (2016) were used for all problem domains investigated in this paper.

4.1.3 Training Set

All the aforementioned machine learning algorithms were trained separately for each problem using a training set consisting of 800 independent benchmark runs. These runs are obtained by investigating 20 satisfiable[2] instances and by considering 40 different tree decompositions (generated using the Min-Fill heuristic, see Dechter, 2003) for each of the

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 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.

problem instances. In addition 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 the termination of a test run we extracted all of our proposed tree decomposition features (in our experiments, this took at most two seconds even for the largest instances) 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 $(X - \mu)/\sigma$. An example for this normalization is the following: Assume that $2, 4$ and 6 are three evaluations for a feature X, obtained from three different tree decompositions of a problem instance. Obviously, the mean μ of these values is 4 and the standard deviation σ is 2. Hence, after standardization, we obtain the values $-1, 0$ and 1. When we consider another problem instance where feature X takes the values $1, 2$ and 3 (when given three tree decompositions of the instance), we again end up with the normalized values $-1, 0$ and 1. In this way, the previously different domains of the feature values for the two instances become comparable.

4.1.4 Evaluation Set

The evaluation set for the computed models consists of 2000 benchmark runs per problem domain. 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.

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 predicted by the respective model (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 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. For the utopistic case that we are able to save 100% of the runtime, i.e., when the dynamic programming algorithm needs no time to solve the problem instance using the selected decomposition, we would obtain a result of 1. Every value less than 0 means that we observe a deterioration of the performance using the respective model as runtime predictor.

As the runtime improvement is strongly dependent on the size of the problem at hand, 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. The tree decomposition which led to the fastest solving time is ranked first. Hence, the closer the 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. In this context 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" one.

4.1.5 Evaluation Process

Indeed, the strict separation of the input data set into training and evaluation set makes the experiments prone to potential bias. To overcome this issue, we use random sub-sampling with 10 splits throughout our whole experimental evaluation. This approach constitutes a randomized adaption of the well-established technique of 10-fold cross-validation. The complete experimental setup for a problem therefore consists of 2800 benchmark runs based on different 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 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 box-plots. 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 results we obtained for this problem in our experiments are summarized in Figure 5. Before we go into the details of the figure, we first want to introduce its 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 the figure the problem name as well as information about the minimum and maximum runtime variation for the given instances are provided. These two ranges refer to the span between minimum and maximum solving time for a given problem instance

Minimum Dominating Set			
Minimum Runtime Variation:		2.0 s – 5.2 s	
Maximum Runtime Variation:		41.1 s – 2877.6 s	
Minimum Improvement:	7.39 %	Average Improvement:	21.80 %
Maximum Improvement:	31.15 %	Median Improvement:	24.25 %
Statistical Significance:		$\geq$ 99.95 %	

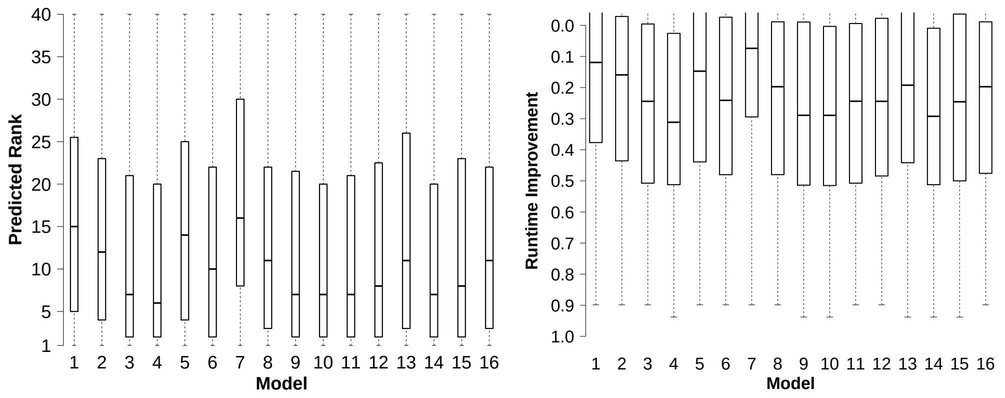


Figure 5: Performance Characteristics for Minimum Dominating Set

when considering all random seeds which were used. In the case at hand we observe that the instance of Minimum Dominating Set with minimum runtime variation needed a solving time between 2.0 and 5.2 seconds (leading to a variation of 3.2 seconds) and that the instance with maximum variation requires solving times between 41.1 and 2877.6 seconds. The second pair of values, indicating a runtime variation of more than half an hour for the very same instance, impressively illustrates that there are huge differences in terms of runtime and so there is a significant potential for improvements which we try to exploit.

The aggregated measurements for the performance improvements achieved by using our approach are given in the subsequent rows of the header. 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 taking the median significance of the one-sided t-test with the null-hypothesis that the observed performance improvement for a given model is 0. In other words, this last value in the header gives the probability that the average model, i.e., the hypothetical model ranked at the median position 8.5 among the 16 models, will indeed lead to a statistically significant 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 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

3-Colorability	
Minimum Runtime Variation:	1.7 s – 3.1 s
Maximum Runtime Variation:	3.2 s – 1351.7 s

Minimum Improvement:	16.90 %	Average Improvement:	29.74 %
Maximum Improvement:	36.67 %	Median Improvement:	30.99 %

Statistical Significance:	$\geq$ 99.95 %

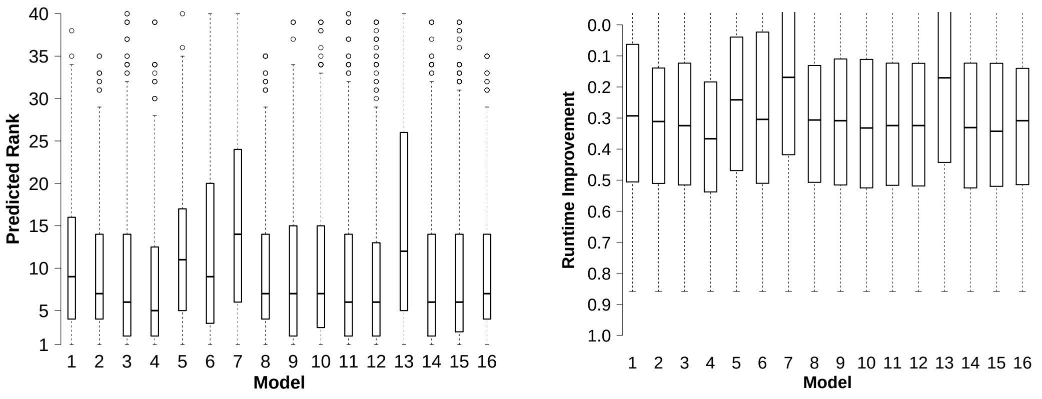


Figure 6: Performance Characteristics for 3-Colorability

certainty – for our benchmark setup 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 Figure 5 are constructed on the basis of the 500 evaluations (50 instances with 10 iterations each) for each computed Model 1–16 (see Section 4.1.2). 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 actual problem instances. To allow for a uniform presentation 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.

4.2.2 3-Colorability

We will now present our experiments on 3-Colorability, depicted in Figure 6. Compared to the first problem we investigated in this paper, this one is 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

Perfect Dominating Set			
Minimum Runtime Variation:		2.3 s – 4.3 s	
Maximum Runtime Variation:		47.3 s – 6470.8 s	
Minimum Improvement:	46.22 %	Average Improvement:	58.91 %
Maximum Improvement:	64.72 %	Median Improvement:	60.69 %
Statistical Significance:		$\geq$ 99.95 %	

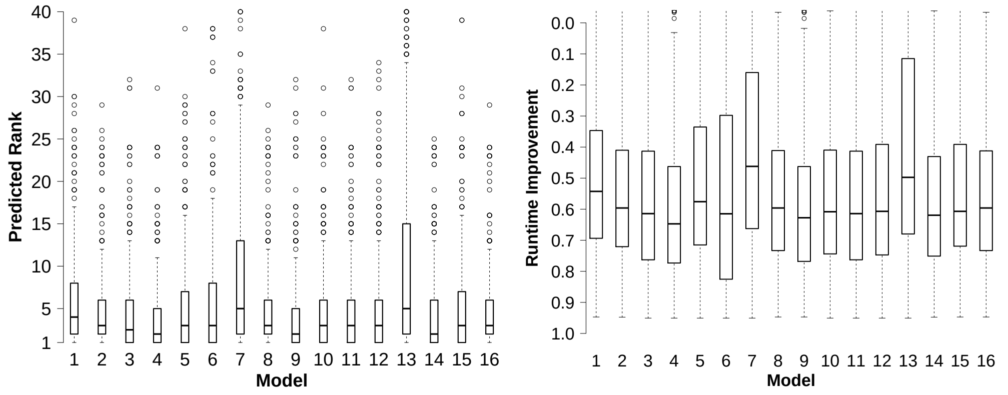


Figure 7: Performance Characteristics for Perfect Dominating Set

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. Again, we observe a remarkable difference between the minimum and maximum solving time for the very same problem instance, especially when we look at the maximum runtime variation depicted in the header of Figure 6. In our experiments with 3-Colorability, selecting a "good" tree decomposition can make the difference between solving an instance within seconds or having to wait more than twenty minutes.

Apart from this small side remark we have for this problem domain 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 fewer solutions and so we expect a higher impact of the tree decomposition features on the solving time and therefore a better predicted rank than in the case of Minimum Dominating Set.

Figure 7 shows that 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

Connected Vertex Cover			
Minimum Runtime Variation:		4.0 s – 10.0 s	
Maximum Runtime Variation:		111.7 s – 2417.6 s	
Minimum Improvement:	22.38 %	Average Improvement:	32.91 %
Maximum Improvement:	42.78 %	Median Improvement:	34.49 %
Statistical Significance:		≥ 99.95 %	

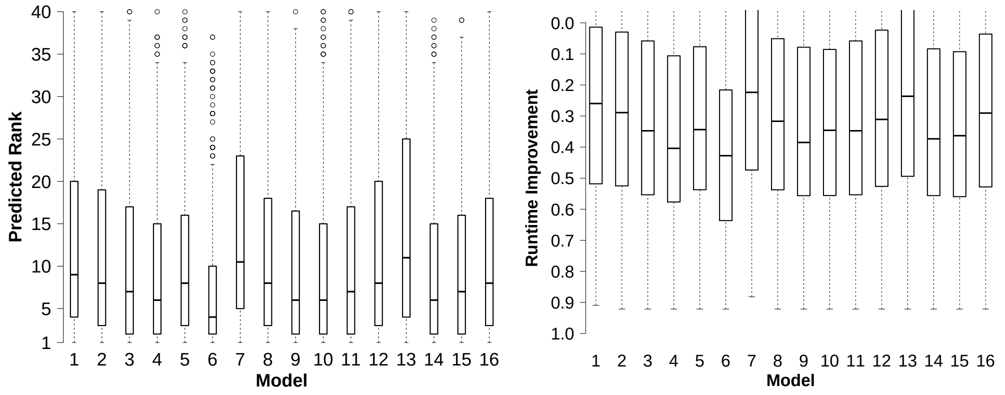


Figure 8: Performance Characteristics for Connected Vertex Cover

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.

Also in this case, the extremely diverging runtimes become apparent when interpreting the minimum and maximum runtime depicted in Figure 7. Solving a problem instance in less than a minute or waiting for the same result more than 1.5 hours can make a huge difference and also for the easiest instances the runtime needed to solve the respective instance is approximately cut in half.

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

Figure 8 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. 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.

Steiner Tree - 10 Terminals			
Minimum Runtime Variation:		1.8 s – 8.1 s	
Maximum Runtime Variation:		191.3 s – 10618.6 s	
Minimum Improvement:	31.56 %	Average Improvement:	59.05 %
Maximum Improvement:	66.33 %	Median Improvement:	62.02 %
Statistical Significance:		≥ 99.95 %	

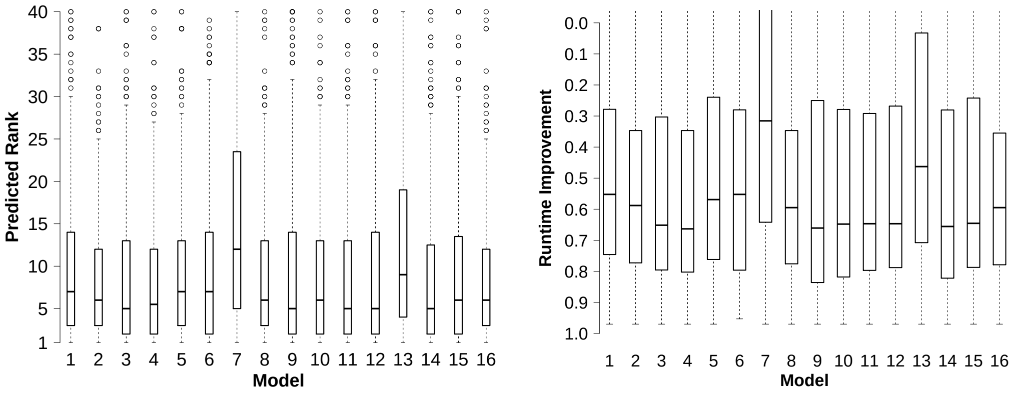


Figure 9: Performance Characteristics for Steiner Tree

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.

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. Still, as shown in Figure 9, 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. The fact that even the models performing worst achieve significant savings of around a third of the total runtime becomes even more important when we look at the fact that these savings can be in the magnitude of hours when considering the maximum runtime variation depicted in Figure 9.

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 a tendency to split the available 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).

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)

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, 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 tree decompositions for the cities are of the same width and hence, the huge runtime differences we observe on the different metro systems cannot be explained by considering the width only.

Figure 10 shows the aggregated outcome for our experiments on the metro systems, hence each box-plot is constructed from 14000 benchmark runs. A separate discussion for each of the cities is given in the accompanying technical report (Abseher et al., 2016). In the figure we can see that each model leads to runtime savings and that the majority of the models helps us saving more than a quarter of the total runtime in average. For the metro system of Tokyo this saves us “only” a few seconds while in the case of Vienna we sometimes save hours using our approach.

Steiner Tree - 10 Terminals			
Minimum Runtime Variation:		3.7 s – 8.4 s	
Maximum Runtime Variation:		35.9 s – 21528.0 s	
Minimum Improvement:	1.55 %	Average Improvement:	27.33 %
Maximum Improvement:	35.61 %	Median Improvement:	29.66 %
Statistical Significance:		$\geq$ 99.95 %	

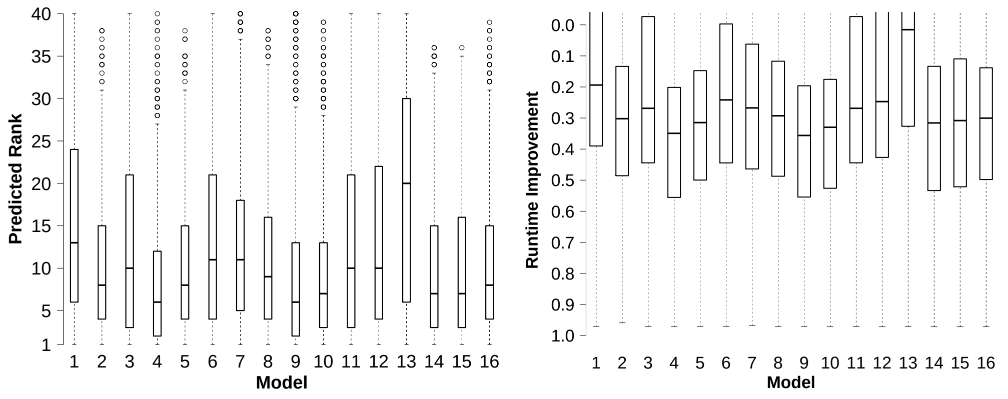


Figure 10: Performance Characteristics for Steiner Tree (Real-World)

4.4 Model Evaluation

After we presented the thorough investigation of our approach on random/real-world instances, 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. 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 (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 an 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 results. Especially noteworthy is Model 4 (LinearRegression) which is able to select a Top-5 rank in the average case of our experiments.

Model	Col	MDS	PDS	CVC	ST	ST (Real)	Avg.	Med.
1 (Gauss.-Proc.)	9	15	4	9	7	13	9.50	9.00
2 (IsotonicReg.)	7	12	3	8	6	8	7.33	7.50
3 (LeastMedSq)	6	7	2.5	7	**5**	10	6.25	6.50
4 (LinearReg.)	**5**	**6**	**2**	6	5.5	**6**	**5.08**	**5.75**
5 (ML-Perc.)	11	14	3	8	7	8	8.50	8.00
6 (PaceReg.)	9	10	3	**4**	7	11	7.33	8.00
7 (PLSClassifier)	14	16	5	10.5	12	11	11.42	11.50
8 (SMOreg)	7	11	3	8	6	9	7.33	7.50
9 (IBk)	7	7	**2**	6	**5**	**6**	5.50	6.00
10 (KStar)	7	7	3	6	6	7	6.00	6.50
11 (LWL)	6	7	3	7	**5**	10	6.33	6.50
12 (AdditiveReg.)	6	8	3	8	**5**	10	6.67	7.00
13 (Bagging)	12	11	5	11	9	20	11.33	11.00
14 (CVPSel.)	6	7	**2**	6	**5**	7	5.50	6.00
15 (M5Rules)	6	8	3	7	6	7	6.17	6.50
16 (M5P)	7	11	3	8	6	8	7.17	7.50
Average	7.81	9.81	3.09	7.47	6.41	9.44	7.34	—
Median	7.00	9.00	3.00	7.50	6.00	8.50	—	—

Table 3: Predicted Ranks (Median) for Computed Models

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 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 that was used to generate the training data for the models and the columns stand for the evaluation dataset. The respective datasets 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. Cases in which we do not observe improvements are enclosed by brackets. 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 observe on average in our setting when we switch the problem domain.

We can see that in almost all cases (34 out of 36) we observe improved results and 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

	Col	MDS	PDS	CVC	ST	ST (Real)	Avg.	Med.
Col	7	11.75	12	11	19	11.5	12.04 13.05	11.63 11.75
MDS	12.5	9	7	8	14	9.5	10.00 10.20	9.25 9.50
PDS	(20)	10.75	3	6	6	12	9.63 10.95	8.38 10.75
CVC	16	8	5	7.5	8	11	9.25 9.60	8.00 8.00
ST	(23.75)	17.25	5	9.5	6	16	12.92 14.30	12.75 16.00
ST (Real)	16.5	15.5	10	13.5	16	8.5	13.33 14.30	14.25 15.5
Average	15.96 17.75	12.04 12.65	7.00 7.80	9.25 9.60	11.50 12.60	11.42 12.00	11.19 12.07	—
Median	16.25 16.50	11.25 11.75	6.00 7.00	8.75 9.50	11.00 14.00	11.25 11.50	—	—

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

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 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 decompositions for a given instance, just minimizing 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. In our experimental evaluation 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 assume that the poor performance of PLSClassifier originates from an overly restrictive filter being used. In the case of Bagging, the underlying regression algorithm (the algorithm which is used in our experiments employs support-vector machines for regression) may be too strong and choosing a weaker base classifier may help to improve the prediction quality, as suggested by Breiman (1996).

Our experiments show that it is hard to 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.

5. 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 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.

Indeed, due to the fact that algorithms based on tree decompositions are an area of intensive research, there also arise performance improvements for specialized algorithms, as studied by Fafianie et al. (2015) who showed that the efficiency of solving the Steiner Tree problem can be significantly improved by combining tree decompositions with methods from linear algebra. It may be an interesting task in future work to investigate the impact of tree decomposition selection also in this context. Furthermore, we expect that problem-specific features are a promising enhancement of our approach. In the paper at hand, we only used features of the tree decomposition in order to establish the problem-independent, general applicability of our approach. In practical situations in which efficiency is crucial, it may be worth trying to find some kind of problem-specific tuning. For instance, one could aim to develop a good approximation function that estimates the time of filling the dynamic programming tables, similar to the idea of the f-cost in the work by Bodlaender and Fomin (2005), and then use this function to obtain a new feature.

Acknowledgments

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. Furthermore, we thank the anonymous reviewers of our paper for their valuable comments which helped to significantly improve our work.

References

Abseher, M., Bliem, B., Charwat, G., Dusberger, F., Hecher, M., & Woltran, S. (2014). The D-FLAT System for Dynamic Programming on Tree Decompositions. In *European Conference On Logics In Artificial Intelligence, JELIA 2014*, Vol. 8761 of *Lecture Notes in Artificial Intelligence*, pp. 558–572. Springer.

Abseher, M., Dusberger, F., Musliu, N., & Woltran, S. (2015). 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*, pp. 275–282. AAAI Press.

Abseher, M., Musliu, N., & Woltran, S. (2016). Improving the Efficiency of Dynamic Programming on Tree Decompositions via Machine Learning. Tech. rep. DBAI-TR-2016-94, TU Wien. Available under `http://www.dbai.tuwien.ac.at/research/report/dbai-tr-2016-94.pdf`.

Arnborg, S., Corneil, D. G., & Proskurowski, A. (1987). Complexity of finding embeddings in a k-tree. *Journal on Algebraic Discrete Methods*, *8*(2), 277–284.

Arnborg, S., & Proskurowski, A. (1989). Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, *23*(1), 11–24.

Bachoore, E. H., & Bodlaender, H. L. (2006). 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*, Vol. 4041 of *Lecture Notes in Computer Science*, pp. 255–266. Springer.

Berry, A., Heggernes, P., & Simonet, G. (2003). The Minimum Degree Heuristic and the Minimal Triangulation Process. In *Graph-Theoretic Concepts in Computer Science: 29th International Workshop, WG 2003*, Vol. 2880 of *Lecture Notes in Computer Science*, pp. 58–70. Springer.

Bertelè, U., & Brioschi, F. (1973). On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A*, *14*(2), 137–148.

Bliem, B., Pichler, R., & Woltran, S. (2013). Declarative Dynamic Programming as an Alternative Realization of Courcelle's Theorem. In *International Symposium on Parameterized and Exact Computation, IPEC 2013*, Vol. 8246 of *Lecture Notes in Computer Science*, pp. 28–40. Springer.

Bodlaender, H. L., & Fomin, F. V. (2005). Tree decompositions with small cost. *Discrete Applied Mathematics*, *145*(2), 143–154.

Bodlaender, H. L., & Koster, A. M. C. A. (2008). Combinatorial Optimization on Graphs of Bounded Treewidth. *The Computer Journal*, *51*(3), 255–269.

Bodlaender, H. L., & Koster, A. M. C. A. (2010). Treewidth computations I. Upper bounds. *Information and Computation*, *208*(3), 259–275.

Borie, R. B., Parker, R. G., & Tovey, C. A. (1992). Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, *7*(1), 555–581.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, *24*(2), 123–140.

Brewka, G., Eiter, T., & Truszczyński, M. (2011). Answer set programming at a glance. *Communications of the ACM*, *54*(12), 92–103.

Chandrashekar, G., & Sahin, F. (2014). A survey on feature selection methods. *Computers & Electrical Engineering*, *40*(1), 16–28.

Clautiaux, F., Moukrim, A., Négre, S., & Carlier, J. (2004). Heuristic and meta-heurisistic methods for computing graph treewidth. *RAIRO Operational Research*, *38*, 13–26.

Courcelle, B. (1990). The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, *85*(1), 12–75.

Dechter, R. (2003). *Constraint Processing.* The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann.

Downey, R. G., & Fellows, M. R. (1999). *Parameterized Complexity.* Monographs in Computer Science. Springer.

Fafianie, S., Bodlaender, H. L., & Nederlof, J. (2015). Speeding up dynamic programming with representative sets: An experimental evaluation of algorithms for Steiner tree on tree decompositions. *Algorithmica*, *71*(3), 636–660.

Gogate, V., & Dechter, R. (2004). A Complete Anytime Algorithm for Treewidth. In *Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence, UAI 2004*, pp. 201–208. AUAI Press.

Gutin, G. (2015). Should we care about huge imbalance in parameterized algorithmics?. *Parameterized Complexity Newsletter*, *11*(2).

Guyon, I., & Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research*, *3*, 1157–1182.

Halin, R. (1976). S-functions for graphs. *Journal of Geometry*, *8*, 171–186.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA Data Mining Software: An Update. *SIGKDD Exploration Newsletters*, *11*(1), 10–18.

Hammerl, T., & Musliu, N. (2010). 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, pp. 95–106. Springer.

Hammerl, T., Musliu, N., & Schafhauser, W. (2015). Metaheuristic Algorithms and Tree Decomposition. In *Handbook of Computational Intelligence*, pp. 1255–1270. Springer.

Hutter, F., Xu, L., Hoos, H. H., & Leyton-Brown, K. (2014). Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, *206*, 79–111.

Jégou, P., & Terrioux, C. (2014). Bag-Connected Tree-Width: A New Parameter for Graph Decomposition. In *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2014*, pp. 12–28.

Kanda, J., Carvalho, A., Hruschka, E., & Soares, C. (2011). Selection of algorithms to solve traveling salesman problems using meta-learning. *International Journal of Hybrid Intelligent Systems*, *8*(3), 117–128.

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

Kloks, T. (1994). *Treewidth, Computations and Approximations*, Vol. 842 of *Lecture Notes in Computer Science*. Springer.

Kneis, J., Langer, A., & Rossmanith, P. (2011). Courcelle's Theorem - A Game-Theoretic Approach. *Discrete Optimization*, *8*(4), 568–594.

Koster, A. M. C. A., van Hoesel, S. P. M., & Kolen, A. W. J. (1999). Solving Frequency Assignment Problems via Tree-Decomposition 1. *Electronic Notes in Discrete Mathematics*, *3*, 102–105.

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

Larranaga, P., Kujipers, C. M., Poza, M., & Murga, R. H. (1997). Decomposing bayesian networks: Triangulation of the moral graph with genetic algorithms. *Statistics and Computing*, *7 (1)*, 19–34.

Lauritzen, S. L., & Spiegelhalter, D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society, Series B*, *50*, 157–224.

Leyton-Brown, K., Nudelman, E., & Shoham, Y. (2009). Empirical Hardness Models: Methodology and a Case Study on Combinatorial Auctions. *Journal of the ACM*, *56*(4), 1–52.

Mersmann, O., Bischl, B., Trautmann, H., Wagner, M., Bossek, J., & Neumann, F. (2013). A novel feature-based approach to characterize algorithm performance for the traveling salesperson problem. *Annals of Mathematics and Artificial Intelligence*, *69*(2), 151–182.

Morak, M., Musliu, N., Pichler, R., Rümmele, S., & Woltran, S. (2012). Evaluating Tree-Decomposition Based Algorithms for Answer Set Programming. In *Learning and Intelligent Optimization Conference, LION 2012*, Vol. 7219 of *Lecture Notes in Computer Science*, pp. 130–144. Springer.

Musliu, N. (2008). An iterative heuristic algorithm for tree decomposition. *Studies in Computational Intelligence, Recent Advances in Evolutionary Computation for Combinatorial Optimization*, *153*, 133–150.

Musliu, N., & Schafhauser, W. (2007). Genetic algorithms for generalized hypertree decompositions. *European Journal of Industrial Engineering*, *1*(3), 317–340.

Musliu, N., & Schwengerer, M. (2013). Algorithm Selection for the Graph Coloring Problem. In *Learning and Intelligent Optimization Conference, LION 2013*, Vol. 7997 of *Lecture Notes in Computer Science*, pp. 389–403. Springer.

Niedermeier, R. (2006). *Invitation to Fixed-Parameter Algorithms.* Oxford Lecture Series in Mathematics And Its Applications. Oxford University Press.

Pihera, J., & Musliu, N. (2014). Application of machine learning to algorithm selection for TSP. In *26th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2014*, pp. 47–54. IEEE Computer Society.

Robertson, N., & Seymour, P. D. (1984). Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Ser. B*, *36*(1), 49–64.

Shoikhet, K., & Geiger, D. (1997). A Practical Algorithm for Finding Optimal Triangulations. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference, AAAI 1997/IAAI 1997*, pp. 185–190. AAAI Press / The MIT Press.

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

Smith-Miles, K., van Hemert, J. I., & Lim, X. Y. (2010). Understanding TSP Difficulty by Learning from Evolved Instances. In *Learning and Intelligent Optimization Conference, LION 2010*, Vol. 6073 of *Lecture Notes in Computer Science*, pp. 266–280. Springer.

Smith-Miles, K., Wreford, B., Lopes, L., & Insani, N. (2013). Predicting Metaheuristic Performance on Graph Coloring Problems Using Data Mining. In *Hybrid Metaheuristics*, Vol. 434 of *Studies in Computational Intelligence*, pp. 417–432. Springer.

Tarjan, R. E., & Yannakakis, M. (1984). 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.

Xu, J., Jiao, F., & Berger, B. (2005). A tree-decomposition approach to protein structure prediction. In *Computational Systems Bioinformatics Conference, CSB 2005*, pp. 247–256. IEEE Computer Society.

Xu, L., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2008). SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, *32*, 565–606.