Evaluating Tree-Decomposition Based Algorithms for Answer Set Programming

Michael Morak, Nysret Musliu, Reinhard Pichler,
Stefan Rümmele, and Stefan Woltran

Institute of Information Systems, Vienna University of Technology
{surname}@dbai.tuwien.ac.at

Abstract. A promising approach to tackle intractable problems is given by a combination of decomposition methods with dynamic algorithms. One such decomposition concept is tree decomposition. However, several heuristics for obtaining a tree decomposition exist and, moreover, also the subsequent dynamic algorithm can be laid out differently. In this paper, we provide an experimental evaluation of this combined approach when applied to reasoning problems in propositional answer set programming. More specifically, we analyze the performance of three different heuristics and two different dynamic algorithms, an existing standard version and a recently proposed algorithm based on a more involved data structure, but which provides better theoretical runtime. The results suggest that a suitable combination of the tree decomposition heuristics and the dynamic algorithm has to be chosen carefully. In particular, we observed that the performance of the dynamic algorithm highly depends on certain features (besides treewidth) of the provided tree decomposition. Based on this observation we apply supervised machine learning techniques to automatically select the dynamic algorithm depending on the features of the input tree decomposition.

1 Introduction

Many instances of constraint satisfaction problems and other NP-hard problems can be solved in polynomial time if their treewidth is bounded by a constant. This suggests two-phased implementations where first a tree decomposition [25] of the given problem is obtained which is then used in the second phase to solve the problem under consideration by a (usually, dynamic) algorithm traversing the tree decomposition. The running time of the dynamic algorithm[1] mainly depends on the width of the provided tree decomposition. Hence, the overall process performs well on instances of small treewidth (formal definitions of tree decompositions and treewidth are given in Section 2), but can also be used in general in case the running time for finding a tree decomposition remains low. Thus, instead of complete methods for finding a tree decomposition, heuristic methods are often employed. In other words, to gain a good performance for this combined tree-decomposition dynamic-algorithm (TDDA, in the following) approach we require efficient tree decomposition techniques which still provide results for which the running time of the dynamic algorithm is feasible.

[1] We use – throughout the paper – the term "dynamic algorithm" as a synonym for "dynamic *programming* algorithm" to avoid confusion with the concept of Answer-Set *programming*.

Y. Hamadi and M. Schoenauer (Eds.): LION 6, LNCS 7219, pp. 130–144, 2012.

Tree-decomposition based algorithms have been used in several applications including probabilistic networks [18] or constraint satisfaction problems such as MAX-SAT [17]. The application area we shall focus on here is propositional Answer-Set Programming (ASP, for short) [20,23] which is nowadays a well acknowledged paradigm for declarative problem solving with many successful applications in the areas of AI and KR.[2] The problem of deciding ASP consistency (i.e. whether a logic program has at least one answer set) is Σ_2^P-complete in general but has been shown tractable [12] for programs of bounded treewidth. In this paper, we consider a certain subclass of programs, namely head-cycle free programs (for more formal definitions, we again refer to Section 2); for such programs the consistency problem is NP-complete.

Let us illustrate here the functioning of ASP on a typical example. Consider the problem of 3-colorability of an (undirected) graph and suppose the vertices of a graph are given via the predicate $\mathrm{vertex}(\cdot)$ and its edges via the predicate $\mathrm{edge}(\cdot,\cdot)$. We employ a disjunctive rule to guess a color for each node in the graph, and then check in the remaining three rules whether adjacent vertices have indeed different colors:

$$\mathrm{r}(X) \vee \mathrm{g}(X) \vee \mathrm{b}(X) \leftarrow \mathrm{vertex}(X);$$
$$\bot \leftarrow \mathrm{r}(X), \mathrm{r}(Y), \mathrm{edge}(X,Y);$$
$$\bot \leftarrow \mathrm{g}(X), \mathrm{g}(Y), \mathrm{edge}(X,Y);$$
$$\bot \leftarrow \mathrm{b}(X), \mathrm{b}(Y), \mathrm{edge}(X,Y);$$

Assume a simple input database with facts $\mathrm{vertex}(a)$, $\mathrm{vertex}(b)$ and $\mathrm{edge}(a,b)$. The above program (together with the input database) yields six answer sets. In fact, the above program is head-cycle free. Many NP-complete problems can be succinctly represented using head-cycle free programs (in particular, the disjunction allows for a direct representation of the guess; in our example the guess of a coloring); see [19] (Section 3) for a collection of problems which can be represented with head-cycle free programs as opposed to problems which require the full power of ASP. However, the above program contains variables and thus has to be grounded yet. So-called grounders turn such programs into variable-free (i.e., propositional) ones which are then fed into ASP-solvers. The algorithms discussed in this paper work on variable-free programs. We emphasize at this point a valuable side-effect. For our example above, it turns out that if the input graph has small treewidth, then the grounded variable-free program has small treewidth as well (see Section 2 for a continuation of the example). This not only holds for the encoding of the 3-colorability problem, but for many other ASP programs (in particular, programs without recursive rules). Thus the class of propositional programs with low treewidth is indeed important also in the context of ASP with variables.

A dynamic algorithm for general propositional ASP has already been presented in [15]. Recently, a new algorithm was proposed for the fragment of head-cycle free programs [21]. Their main differences are as follows: the algorithm from [15] is based on ideas from dynamic SAT algorithms [26] and explicitly takes care of the minimality checks following the standard definition of answer sets; thus it requires double-exponential time in the width of the provided tree decomposition. The algorithm

[2] See `http://www.cs.uni-potsdam.de/~torsten/asp/` for a collection.

proposed in [21] follows a more involved characterization [5] which applies to head-cycle free programs and thus calls for a more complex data structure and operations. However, it runs in single-exponential time wrt. the width of the provided tree decomposition. Both algorithms have been integrated into a novel TDDA system for ASP, which we call dynASP[3]. For the tree-decomposition phase, dynASP offers three different heuristics, namely Maximum Cardinality Search (MCS) [29], Min-Fill and Minimum Degree (see [7] for a survey on such heuristics). According to [11], the min-fill heuristic usually produces tree decompositions of lower width than the other heuristics.

By the above considerations, one would naturally expect that computing a tree decomposition with the min-fill heuristic (which usually yields the lowest width) and applying the dedicated dynamic algorithm from [21] for head-cycle free ASPs (which is single-exponential wrt. to the width of the tree decomposition) yields the best two-phased algorithm for head-cycle free ASPs. Surprisingly, extensive testing with our dynASP system has by no means confirmed these expectations: First, the TDDA algorithm is not always most efficient when the best heuristic for tree decomposition is used. Second, the specialized algorithm for head-cycle free programs does not always perform better than the general algorithm, although the worst-case running time of the latter is double-exponential in the treewidth while the running time of the former is only single-exponential.

The goal of this paper is to get a deeper understanding of the interplay between tree decompositions and dynamic algorithms and to arrive at an optimal configuration of the two-phased dynamic algorithm. The above mentioned experimental results suggest that the width of the tree decomposition is not the only significant parameter for efficiency of our dynamic algorithms. Therefore, we identify other important features of tree decompositions that influence the running time of the dynamic algorithms. Based on these observations, we propose the application of machine learning techniques to automatically select the best dynamic algorithm for the given input instance. We successfully apply classification techniques for algorithm selection in this domain. Additionally, we exploit regression techniques that are used to predict the runtime of our dynamic algorithms based on input instance features.

Note that the proposed features of tree decompositions are independent of the application domain of ASP. We therefore expect that our insights into the influence of various characteristics of tree decompositions on the performance of TDDAs are generally applicable to tree-decomposition based algorithms and that they are by no means restricted to ASPs. The same holds true for the methodology developed here in order to arrive at an optimal algorithm configuration of such two-phased algorithms.

2 Preliminaries

Answer Set Programming. A (propositional) disjunctive logic program (program, for short) is a pair $\Pi = (\mathcal{A}, \mathcal{R})$, where $\mathcal{A}$ is a set of propositional atoms and $\mathcal{R}$ is a set of rules of the form:

[3] A preliminary version of this system has been presented in [22], see `http://dbai.tuwien.ac.at/proj/dynasp`.

$$a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \quad (1)$$

where "¬" is default negation[4] $n \geq 1$, $n \geq m \geq l$ and $a_i \in \mathcal{A}$ for all $1 \leq i \leq n$. A rule $r \in \mathcal{R}$ of the form (1) consists of a head $H(r) = \{a_1, \ldots, a_l\}$ and a body $B(r) = B^+(r) \cup B^-(r)$, given by $B^+(r) = \{a_{l+1}, \ldots, a_m\}$ and $B^-(r) = \{a_{m+1}, \ldots, a_n\}$. A set $M \subseteq \mathcal{A}$ is a called a model of r, if $B^+(r) \subseteq M \wedge B^-(r) \cap M = \emptyset$ implies that $H(r) \cap M \neq \emptyset$. We denote the set of models of r by $Mod(r)$ and the models of a program $\Pi = (\mathcal{A}, \mathcal{R})$ are given by $Mod(\Pi) = \bigcap_{r \in \mathcal{R}} Mod(r)$.

The reduct Π^I of a program Π w.r.t. an interpretation $I \subseteq \mathcal{A}$ is given by $(\mathcal{A}, \{r^I : r \in \mathcal{R}, B^-(r) \cap I = \emptyset)\})$, where r^I is r without the negative body, i.e., $H(r^I) = H(r)$, $B^+(r^I) = B^+(r)$, and $B^-(r^I) = \emptyset$. Following [10], $M \subseteq \mathcal{A}$ is an *answer set* of a program $\Pi = (\mathcal{A}, \mathcal{R})$ if $M \in Mod(\Pi)$ and for no $N \subset M$, $N \in Mod(\Pi^M)$.

We consider here the class of *head-cycle free programs* (HCFPs) as introduced in [5]. We first recall the concept of *(positive) dependency graphs*. A dependency graph of a program $\Pi = (\mathcal{A}, \mathcal{R})$ is given by $\mathcal{G} = (V, E)$, where $V = \mathcal{A}$ and $E = \{(p, q) \mid r \in \mathcal{R}, p \in B^+(r), q \in H(r)\}$. A program $\Pi = (\mathcal{A}, \mathcal{R})$ is called head-cycle free if its dependency graph does not contain a directed cycle going through two different atoms which jointly occur in the head of a rule in $\mathcal{R}$.

Example 1. We provide the fully instantiated (i.e. ground) version of our introductory example from Section 1, which solves the 3-colorability for the given input database $\mathrm{vertex}(a)$, $\mathrm{vertex}(b)$ and $\mathrm{edge}(a, b)$, yielding five rules (taking straight forward simplifications as performed by state-of-the-art grounders into account):

$r1 : \mathrm{r}(a) \vee \mathrm{g}(a) \vee \mathrm{b}(a) \leftarrow \top;$ $\quad r2 : \mathrm{r}(b) \vee \mathrm{g}(b) \vee \mathrm{b}(b) \leftarrow \top;$
$r3 : \bot \leftarrow \mathrm{r}(a), \mathrm{r}(b);$ $\quad r4 : \bot \leftarrow \mathrm{g}(a), \mathrm{g}(b);$
$r5 : \bot \leftarrow \mathrm{b}(a), \mathrm{b}(b);$

Tree Decomposition and Treewidth. A *tree decomposition* of a graph $\mathcal{G} = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$, where T is a tree and χ maps each node t of T (we use $t \in T$ as a shorthand below) to a *bag* $\chi(t) \subseteq V$, such that (1) for each $v \in V$, there is a $t \in T$, s.t. $v \in \chi(t)$; (2) for each $(v, w) \in E$, there is a $t \in T$, s.t. $\{v, w\} \subseteq \chi(t)$; (3) for each $r, s, t \in T$, s.t. s lies on the path from r to t, $\chi(r) \cap \chi(t) \subseteq \chi(s)$.

A tree decomposition (T, χ) is called *normalized* (or *nice*) [16], if (1) each $t \in T$ has ≤ 2 children; (2) for each $t \in T$ with two children r and s, $\chi(t) = \chi(r) = \chi(s)$; and (3) for each $t \in T$ with one child s, $\chi(t)$ and $\chi(s)$ differ in exactly one element, i.e. $|\chi(t) \Delta \chi(s)| = 1$.

The *width* of a tree decomposition is defined as the cardinality of its largest bag minus one. Every tree decomposition can be normalized in linear time without increasing the width [16]. The *treewidth* of a graph $\mathcal{G}$, denoted by $tw(\mathcal{G})$, is the minimum width over all tree decompositions of $\mathcal{G}$.

For a given graph and integer k, deciding whether the graph has treewidth at most k is NP-complete [2]. For computing tree decompositions, different complete [27,11,3] and heuristic methods have been proposed in the literature. Heuristic techniques are mainly

[4] We omit strong negation as considered in [5]; our results easily extend to programs with strong negation.

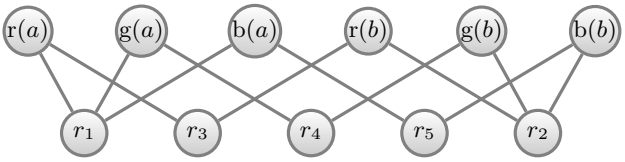

Fig. 1. The incidence graph of the ground program of Example 1

based on searching for a good elimination ordering of graph nodes. Several heuristics that run in polynomial time have been proposed for finding a good elimination ordering of nodes. These heuristics select the ordering of nodes based on different criteria, such as the degree of the nodes, the number of edges to be added to make the node simplicial (a node is simplicial if its neighbors form a clique) etc. We briefly mention three of them: (i) Maximum Cardinality Search (MCS) [29] initially selects a random vertex of the graph to be the first vertex in the elimination ordering (the elimination ordering is constructed from right to left). The next vertex will be picked such that it has the highest connectivity with the vertices previously selected in the elimination ordering. The ties are broken randomly. MCS repeats this process iteratively until all vertices are selected. (ii) The min-fill heuristic first picks the vertex which adds the smallest number of edges when eliminated (the ties are broken randomly). The selected vertex is made simplicial and it is eliminated from the graph. The next vertex in the ordering will be any vertex that adds the minimum number of edges when eliminated from the graph. This process is repeated iteratively until the whole elimination ordering is constructed. (iii) The minimum degree heuristic picks first the vertex with the minimum degree. The selected vertex is made simplicial and it is removed from the graph. Further, the vertex that has the minimum number of unselected neighbors will be chosen as the next node in the elimination ordering. This process is repeated iteratively. MCS, min-fill, and min-degree heuristics run in polynomial time and usually produce a tree decomposition of reasonable width. For other types of heuristics and metaheuristic techniques based on the elimination ordering of nodes, see [7].

Tree Decompositions of Logic Programs. To build tree decompositions for programs, we use incidence graphs.[5] Thus, for program $\Pi = (\mathcal{A}, \mathcal{R})$, such a graph is given by $\mathcal{G} = (V, E)$, where $V = \mathcal{A} \cup \mathcal{R}$ and E is the set of all pairs (a, r) with an atom $a \in \mathcal{A}$ appearing in a rule $r \in \mathcal{R}$. Thus the resulting graphs are bipartite.

For normalized tree decompositions of programs, we thus distinguish between six types of nodes: *leaf* (L), *join* or *branch* (B), *atom introduction* (AI), *atom removal* (AR), *rule introduction* (RI), and *rule removal* (RR) node. The last four types will be often augmented with the element e (either an atom or a rule) which is removed or added compared to the bag of the child node.

Figures 1 and 2 show the incidence graph of Example 1 and a corresponding tree decomposition.

[5] See [26] for justifications why incidence graphs are favorable over other types of graphs.

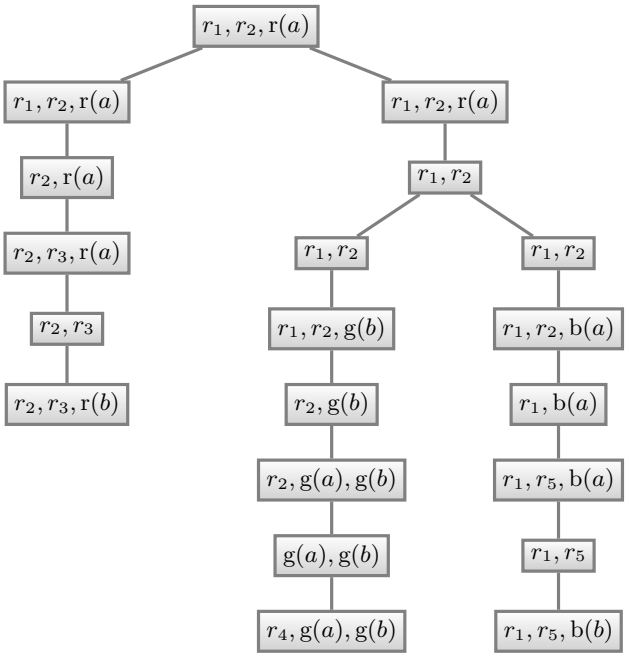

Fig. 2. A normalized tree decomposition of the graph shown in Figure 1

3 Dynamic Algorithms for ASP

Tree-decomposition based dynamic algorithms start at the leaf nodes and traverse the tree to the root. Thereby, at each node a set of partial solutions is generated by taking those solutions into account that have been computed for the child nodes. The most difficult part in constructing such an algorithm is to identify an appropriate data structure to represent the partial solutions at each node: on the one hand, this data structure must contain sufficient information so as to compute the representation of the partial solutions at each node from the corresponding representation at the child node(s). On the other hand, the size of the data structure must only depend on the size of the bag (and not on the size of the entire answer set program).

In this section we review two completely different realizations of this data structure, leading to algorithms which we will call Dyn-ASP1 and Dyn-ASP2.

Dyn-ASP1. The first algorithm was presented in [15]. It was proposed for propositional disjunctive programs Π which are not necessarily head-cycle free. Its data structure, called tree interpretation, follows very closely the characterization of answer sets presented in Section 2. A tree interpretation for tree decomposition $\mathcal{T}$ is a tuple $(t, M, \mathcal{C})$, where t is a node of $\mathcal{T}$, $M \subseteq \chi(t)$ is called assignment, and $\mathcal{C} \subseteq 2^{\chi(t)}$ is called certificate. The idea is that M represents a partial solution limited to what is visible in the bag $\chi(t)$. That means it contains parts of a final answer set as well as all those rules which are already satisfied. The certificate $\mathcal{C}$ takes care of the minimality criteria for

answer sets. It is a list of those partial solutions which are smaller than M together with the rules which are satisfied by them. This means when reaching the root node of $\mathcal{T}$, assignment M can only represent a real answer set if the associated certificate is empty or contains only entries which do not satisfy all rules.

It turns out that due to the properties of tree decompositions it is indeed enough to store only the information of the partial solution which is still visible in the current bag of the tree decomposition. Hence, for each node the number of different assignments M is limited single exponential in the treewidth. Together with the possible exponential size of the certificate this leads to an algorithm with a worst case running time linear in the input size and double exponential in the treewidth.

Dyn-ASP2. We recently proposed the second algorithm in [21]. In contrast to Dyn-ASP1 it is limited to head-cycle free programs. Its data structure is motivated by a new characterization of answer sets for HCFPs:

Theorem 1 ([21]). *Let $\Pi = (\mathcal{A}, \mathcal{R})$ be an HCFP. Then, $M \subseteq \mathcal{A}$ is an answer set of Π if and only if the following holds:*

- $M \in Mod(\Pi)$, *and*
- *there exists a set $\rho \subseteq \mathcal{R}$ such that, $M \subseteq \bigcup_{r \in \rho} H(r)$; the derivation graph induced by M and ρ is acyclic; and for all $r \in \rho$: $B^+(r) \subseteq M$, $B^-(r) \cap M = \emptyset$, and $|H(r) \cap M| = 1$.*

Here the derivation graph induced by M and ρ is given by $V = M \cup \rho$ and E is the transitive closure of the edge set $E' = \{(b, r) : r \in \rho, b \in B^+(r) \cap M\} \cup \{(r, a) : r \in \rho, a \in H(r) \cap M\}$.

Hence, the data structure used in Dyn-ASP2 is a tuple (G, S), where G is a derivation graph (extended by a special node due to technical reasons) and S is the set of satisfied rules used to test the first condition in Theorem 1. Again it is enough to limit G and S to the elements of the current bag $\chi(t)$. Therefore the number of possible tuples (G, S) in each node is at most single exponential in the treewidth. This leads to an algorithm with a worst case running time linear in the input size and single exponential in the treewidth.

4 Evaluation of Tree Decompositions for ASP

In this section we give an extensive evaluation of dynamic algorithms based on tree decompositions for solving benchmark problems in answer set programming. In Figure 3 our solver based on tree decompositions and dynamic algorithms is presented, where Dyn-ASP1 and Dyn-ASP2 refers to the two algorithms described Section 3. Moreover, note that tree decompositions have to be normalized to be amenable to the two dynamic algorithms. The efficiency of our solver depends on the tree decomposition module and the applied dynamic algorithm. Regarding the tree decomposition we evaluated three heuristics which produce different tree decompositions. Furthermore, we analyzed the impact of tree decomposition features on the efficiency of the dynamic algorithms. Observing that neither dynamic algorithm dominates the other on all instances, we propose an automated selection of a dynamic algorithm during the solving process based on the features of the produced tree decomposition.

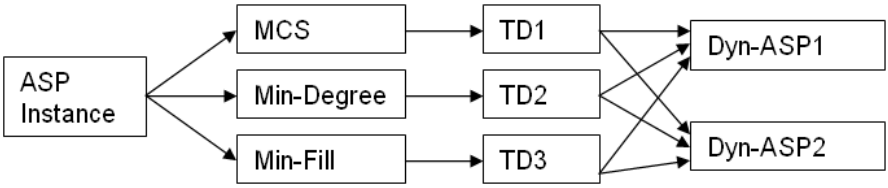

Fig. 3. Architecture of the TDDA-based ASP solver

Benchmark Description: To identify tree decomposition features that impact the runtime of our Dyn-ASP1 and Dyn-ASP2, different logic programs were generated and different tree decompositions were computed for these programs.

Programs were generated in two ways: Firstly, by generating a random SAT instance using MKCNF[6]. These CNF formulas were then encoded as a logic program and passed to the dynASP program. MKCNF was called with the following parameters: Number of clauses ranging from 150 to 300, clause-size ranging from 3 to 13 and number of variables calculated by $10 \times$ number of clauses $\times$ clause-size.

The second method used for program generation closely follows the one described in [31]. For rule-length n, from a set $\mathcal{A}$ of atoms, a head atom and $n-1$ body atoms are randomly selected. Each of the body atoms is negated with a probability of 0.5. Here the rule-length ranges from 3 to 7 and the number of rules ranges from 20 to 50. The number of atoms is always $\frac{1}{5}$ of the number of rules, which is, according to [31], a hard region for current logic program solvers.

For each of these programs, three different tree decompositions are computed using the three heuristics described below. Each of these tree decompositions is then normalized, as both algorithms currently only handle "nice" tree compositions.

Applied Tree-Decomposition Algorithms: As we described in Section 2 different methods have been proposed in the literature for constructing of tree decompositions with small width. Although complete methods give the exact treewidth, they can be used only for small graphs, and were not applicable for our problems which contains up to 20000 nodes. Therefore, we selected three heuristic methods (MCS, min-fill, and min-degree) which give a reasonable width in a very short amount of time. We have also considered using and developing new metaheuristic techniques. Although such an approach slightly improves the treewidth produced by the previous three heuristics, they are far less efficient compared to the original variants. In our experiments we have observed that a slightly improved treewidth does not have a significant impact on the efficiency of the dynamic algorithm for our problem domain and therefore we decided to use the three heuristics directly. We initially used an implementation of these heuristics available in a state-of-the-art libraries [8] for tree/hypertree decomposition. Further, we implemented new data structures that store additional information about vertices, their adjacent edges and neighbors to find the next node in the ordering faster. With these new data structures the performance of Min-fill and MCS heuristics was improved by factor 2–3.

[6] `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC`

4.1 Algorithm Selection

In our experiments we have noted that neither dynamic algorithm dominates the other in all problem instances. Therefore, we have investigated the idea of automated selection of the dynamic algorithm based on the features of the decomposition. Automated algorithm selection is an important research topic and has been investigated by many researchers in the literature (c.f. [28] for a survey). However, to the best of our knowledge, algorithm selection has not yet been investigated for tree decompositions.

To achieve our goal we identified important features of tree decompositions and applied supervised machine learning techniques to select the algorithm that should be used on the particular tree decomposition. We have provided training sets to the machine learning algorithms and analyzed the performance of different variants of these algorithms on the testing set. The detailed performance results of the machine learning algorithm are presented in the next section.

Structural Properties of Tree Decompositions: For every tree decomposition, a number of features are calculated to identify the properties that make them particularly suitable for one of the algorithms (or conversely, particularly unsuitable). The following features (besides treewidth) were used:

- Percentage of join nodes in the normalized tree decomposition (*jpct*)
- Percentage of join nodes in the non-normalized decomposition (*tdbranchpct*)
- Percentage of leaf nodes in the non-normalized decomposition (*tdleafpct*)
- Average distance between two join nodes in the decomposition (*jjdist*)
- Relative size increase of the decomposition during normalization (*nsizeinc*)
- Average bag size of join nodes (*jwidth*)
- Relative size of the tree decomposition (i.e. number of tree nodes) compared to the size (vertices + edges) of the incidence graph (*reltdsize*)

We note that our data set also includes features of the graph from which the tree decomposition is constructed. These features include number of edges of the graph, number of vertices, minimum degree, maximum degree etc. Because the graph features had a minor impact on the machine learning algorithms, the discussion in this paper is concentrated on tree decomposition features.

Experiments: All experiments were performed on a 64bit Gentoo Linux machine with an Intel Core2Duo P9500 2.53GHz processor and 4GB of system RAM. For each generated head-cycle free logic program, 50 tree decompositions were computed with each of the three heuristics available. For each of these 150 decompositions, the two algorithms described in Section 3 were run in order to determine which one works best on the given tree decomposition. Thus, a tuple in the benchmark dataset consists of the generated program and a tree decomposition, and for each tuple it is stored which algorithm performed better and its corresponding runtime.

Based on this generated dataset, using the WEKA toolkit [13], a machine learning approach was used to try to automatically select the best dynamic algorithm for an already computed tree decomposition. Trying to select the best combination of both tree decomposition heuristic and dynamic algorithm unfortunately seems impractical, as the

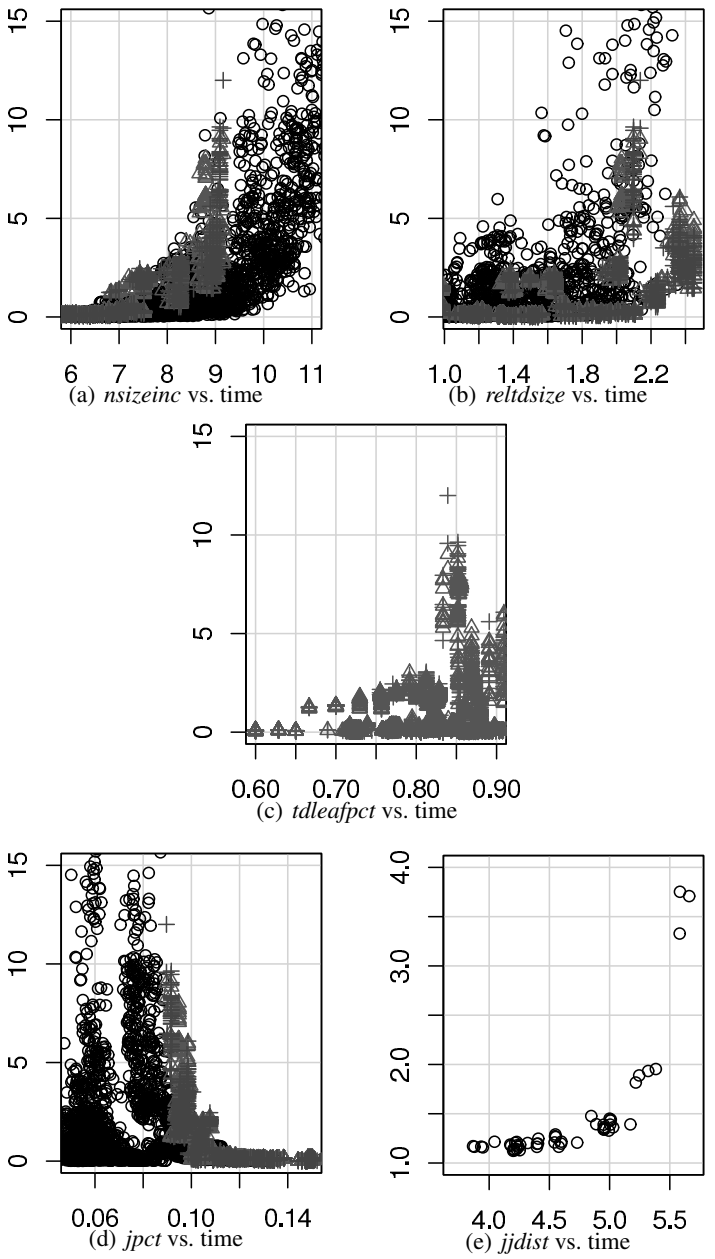

Fig. 4. Every benchmark instance (i.e. each calculated tree decomposition) contributes one datapoint to the plots above. Usage of the MCS, Min-Degree and Min-Fill heuristics are represented by black circles, grey triangles and light grey crosses respectively. Note that the latter two almost always overlap. The Y scale measures overall running time of the best algorithm in seconds. Plots (a)–(d) use the full benchmark set, (e) uses MKCNF 21000 300 7.

underlying graph structure does not seem to provide enough information for a machine learning approach and calculating multiple tree decompositions is not feasible, as it is an expensive process.

Algorithm selection based on classification techniques: Based on the performance of the two algorithms, each tuple in the dataset was either labelled "Dyn-ASP1" or "Dyn-ASP2". Given the differences in runtime as shown in extracts in Table 1, the overall runtime can be improved notably if the better-performing algorithm is run.

Table 1. Exemplary performance differences that can occur in our two algorithms when working on the same tree decomposition

Heuristic	Algorithm	TD width	Runtime (sec)
Min-Degree	Dyn-ASP1	11	53.1629
Min-Degree	Dyn-ASP2	11	7.4058
MCS	Dyn-ASP1	10	6.2420
MCS	Dyn-ASP2	10	268.2940
Min-Fill	Dyn-ASP1	10	9.8325
Min-Fill	Dyn-ASP2	10	2.6030

By using the well-known CFS subset evaluation approach implemented in WEKA (see [14] for details), the *jjdist* and *jpct* properties were identified to correlate strongly with the best algorithm, indicating that they are tree decomposition features which have a high impact on the performance of the dynamic algorithms. When ranked by information gain (see Table 2), the *reltdsize* property ranks second, followed by *tdleafpct*, *tdbranchpct* and *jwidth* indicating that all of these tree decomposition features bear some influence on the dynamic algorithms' runtimes. These outcomes can also be seen in Figure 4, which shows the relationship between runtime and these tree decomposition properties. Interestingly, a direct influence of the *jjdist* feature on the overall running could only be found for the MCS heuristics (see Figure 4(e)). Both other heuristics produced tree decompositions with almost constant *jjdist* value. Conversely, for the *tdleafpct* feature, MCS was the only heuristic not producing direct results (Figure 4(c)).

In order to test the feasibility of a machine learning approach in this setting, a number of machine learning algorithms were run to compare their performance. Three such classifiers were tested: Random decision trees, k-nearest neighbor and a single rule algorithm. The latter serves as a reference point, it always returns the class that occurs most often (in this case "Dyn-ASP2"). For training, the dataset was split tenfold and ten training- and validation runs were done, always training on nine folds and validating with the 10th (10-fold cross-validation). Table 3 shows the classifier performance in detail. It shows for each classifier, how many tuples of each class (the "correct" class) were incorrectly classified, e.g. for all training-tuples on which the Dyn-ASP1 algorithm performed better, the k-NN classifier (wrongly) chose the Dyn-ASP2 algorithm in only 10.8% of the cases.

Algorithm selection based on regression techniques: The second approach that we applied for selection of the best dynamic algorithm on the particular tree decomposition is

Table 2. Feature ranking based on Information Gain, using 10-fold cross-validation

Average merit	Average rank	Attribute
0.436 ± 0.002	1 ± 0	jpct
0.422 ± 0.004	2 ± 0	reltdsize
0.386 ± 0.004	3.2 ± 0.4	jjdist
0.372 ± 0.012	4.2 ± 0.87	tdleafpct
0.357 ± 0.006	5.2 ± 0.6	tdbranchpct
0.354 ± 0.01	5.4 ± 0.8	jwidth

Table 3. Different classifiers and percentages of incorrectly classified instances

Classifier	Correct class	Incorrectly classified
Single-rule	Dyn-ASP1	23.1%
Single-rule	Dyn-ASP2	18.4%
k-NN, k=10	Dyn-ASP1	10.8%
k-NN, k=10	Dyn-ASP2	18.5%
Random forest	Dyn-ASP1	10.6%
Random forest	Dyn-ASP2	18.4%

based on regression techniques. The main idea is to use machine learning algorithms to first predict the runtime of each dynamic algorithm in a particular instance, and then select the algorithm that has better predicted runtime. To learn the model for runtime prediction we provide for each dynamic algorithm a training set that consists of instances that include features of input tree decomposition (and the input graph). Additionally, for each example are given the information for the time needed to construct the tree decomposition and the running time of the particular dynamic algorithm.

We experimented with several machine learning algorithms for regression available in WEKA, and compared their performance regarding the selection accuracy of the fastest dynamic algorithm for the given input instance. For each machine learning algorithm we provided a training set consisting of 6090 examples. The testing set contained 3045 examples.

The algorithm k-NN (k=5) gave best results among these machine learning algorithms regarding runtime prediction for both dynamic algorithms. To illustrate the performance of k-NN algorithm regarding the runtime prediction we present the actual runtime and the predicted runtime for both dynamic algorithms in Figure 5. Results for the first 30 examples in the testing set are given.

Regarding the algorithm selection based on the runtime prediction, we present in Table 4 the best current results that we could obtain with two machine learning algorithms k-NN (k-nearest neighbors, see [1]) and M5P (Pruned regression tree, see [24] and [30]). As we can see the accuracy of selecting the right (fastest) dynamic algorithm for a particular instance is good. In particular, the k-NN algorithm selects the best algorithm for the 88% of the test instances.

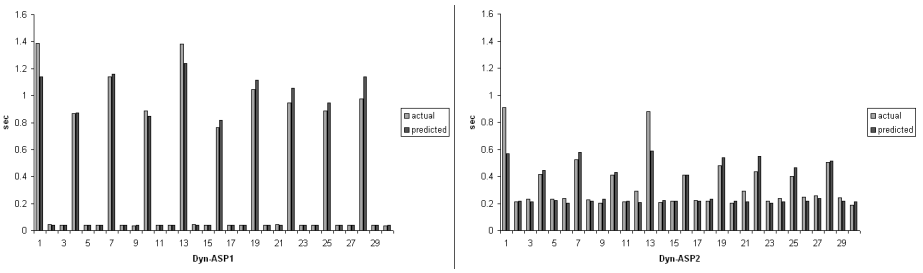

Fig. 5. Actual and predicted time with k-NN for first 30 test examples

Table 4. Two regression algorithms and their accuracy regarding the selection of better dynamic algorithm

Algorithm	Dynamic algorithm selection accuracy
M5P	80.2%
k-NN, k=5	88.1%

5 Discussion

In our experiments, we have identified several important tree decomposition features. As these features can have a high impact on the performance of a subsequent dynamic algorithm, heuristics should try to create "good" decompositions also with respect to these features and not only with respect to the width. It has become apparent that a higher width can be compensated by such a decomposition, e.g. in our benchmarks, the MCS heuristic always produced the worst width, but actually speeds up our dynamic algorithms. Moreover, these features have turned out to be well suitable for classification and regression methods.

The good results that were obtained by our machine learning approach clearly suggest that two-phased algorithms like our dynASP system significantly profit from an automatic selection of the dynamic algorithm in the second phase – based on the tree decomposition features identified here. Given the effectiveness of the single-rule classifier, by simply implementing this rule (which is equivalent to a simple if-statement), the dynamic algorithm can be effectively selected once the tree decomposition has been computed. By utilizing a k-nearest neighbor or random decision tree approach, on average more than 85% of the decisions made are correct, yielding further improvements. Machine learning approaches (like portfolio solvers) are already in use for ASP (see e.g. [4,9]), however these are specific to ASP, whereas our approach, using tree decomposition features for decisions, can generally be used for all TDDA approaches.

6 Conclusion

In this paper we have studied the interplay between three heuristics for the computation of tree decompositions and two different dynamic algorithms for head-cycle free

programs, an important subclass of disjunctive logic programs. We have identified features beside the width of tree decompositions that influence the running time of our dynamic algorithms. Based on these observations, we have proposed and evaluated algorithm selection via different machine learning techniques. This will help to improve our prototypical TDDA system dynASP.

For future work, we plan to study the possibilities to not only perform algorithm selection for the dynamic algorithm but also for the heuristic to compute the tree decomposition. Furthermore, our results suggest that heuristic methods for tree decompositions should not only focus on minimizing the width but should also take some other features as objectives into account. Finally, we expect that our observations are independent of the domain of answer set programming. We therefore plan to evaluate tree-decomposition based algorithms for further problems from various other areas [6].

Acknowledgments. The work was supported by the Austrian Science Fund (FWF): P20704-N18, and by the Vienna University of Technology program "Innovative Ideas".

References

1. Aha, D.W., Kibler, D.F., Albert, M.K.: Instance-based learning algorithms. Machine Learning 6, 37–66 (1991)
2. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Alg. Disc. Meth. 8, 277–284 (1987)
3. Bachoore, E.H., Bodlaender, H.L.: A Branch and Bound Algorithm for Exact, Upper, and Lower Bounds on Treewidth. In: Cheng, S.-W., Poon, C.K. (eds.) AAIM 2006. LNCS, vol. 4041, pp. 255–266. Springer, Heidelberg (2006)
4. Balduccini, M.: Learning and using domain-specific heuristics in ASP solvers. AI Commun. 24(2), 147–164 (2011)
5. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Ann. Math. Artif. Intell. 12, 53–87 (1994)
6. Bodlaender, H.L.: A tourist guide through treewidth. Acta Cybern. 11(1-2), 1–22 (1993)
7. Bodlaender, H.L., Koster, A.M.C.A.: Treewidth computations I. Upper Bounds. Inf. Comput. 208(3), 259–275 (2010)
8. Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B., Musliu, N., Samer, M.: Heuristic Methods for Hypertree Decomposition. In: Gelbukh, A., Morales, E.F. (eds.) MICAI 2008. LNCS (LNAI), vol. 5317, pp. 1–11. Springer, Heidelberg (2008)
9. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M.T., Ziller, S.: A Portfolio Solver for Answer Set Programming: Preliminary Report. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 352–357. Springer, Heidelberg (2011)
10. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. 9(3/4), 365–386 (1991)
11. Gogate, V., Dechter, R.: A complete anytime algorithm for treewidth. In: Proc. UAI 2004, pp. 201–208. AUAI Press (2004)
12. Gottlob, G., Pichler, R., Wei, F.: Bounded treewidth as a key to tractability of knowledge representation and reasoning. In: Proc. AAAI 2006, pp. 250–256. AAAI Press (2006)
13. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. SIGKDD Explorations 11(1), 10–18 (2009)
14. Hall, M.A., Smith, L.A.: Practical feature subset selection for machine learning. In: Proc. ACSC 1998, pp. 181–191. Springer (1998)

15. Jakl, M., Pichler, R., Woltran, S.: Answer-set programming with bounded treewidth. In: Proc. IJCAI 2009, pp. 816–822. AAAI Press (2009)
16. Kloks, T.: Treewidth, computations and approximations. LNCS, vol. 842. Springer, Heidelberg (1994)
17. Koster, A., van Hoesel, S., Kolen, A.: Solving partial constraint satisfaction problems with tree-decomposition. Networks 40(3), 170–180 (2002)
18. Lauritzen, S., Spiegelhalter, D.: 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)
19. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. 7(3), 499–562 (2006)
20. Marek, V.W., Truszczyński, M.: Stable Models and an Alternative Logic Programming Paradigm. In: The Logic Programming Paradigm – A 25-Year Perspective, pp. 375–398. Springer (1999)
21. Morak, M., Musliu, N., Pichler, R., Rümmele, S., Woltran, S.: A new tree-decomposition based algorithm for answer set programming. In: Proc. ICTAI, pp. 916–918 (2011)
22. Morak, M., Pichler, R., Rümmele, S., Woltran, S.: A Dynamic-Programming Based ASP-Solver. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 369–372. Springer, Heidelberg (2010)
23. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. Ann. Math. Artif. Intell. 25(3-4), 241–273 (1999)
24. Quinlan, R.J.: Learning with continuous classes. In: 5th Australian Joint Conference on Artificial Intelligence, Singapore, pp. 343–348 (1992)
25. Robertson, N., Seymour, P.D.: Graph minors II: Algorithmic aspects of tree-width. Journal Algorithms 7, 309–322 (1986)
26. Samer, M., Szeider, S.: Algorithms for propositional model counting. J. Discrete Algorithms 8(1), 50–64 (2010)
27. Shoikhet, K., Geiger, D.: A practical algorithm for finding optimal triangulations. In: Proc. AAAI 1997, pp. 185–190. AAAI Press/The MIT Press (1997)
28. Smith-Miles, K.: Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Comput. Surv. 41(1) (2008)
29. Tarjan, R., Yannakakis, M.: Simple linear-time algorithm to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SIAM J. Comput. 13, 566–579 (1984)
30. Wang, Y., Witten, I.H.: Induction of model trees for predicting continuous classes. In: Poster Papers of the 9th European Conference on Machine Learning (1997)
31. Zhao, Y., Lin, F.: Answer Set Programming Phase Transition: A Study on Randomly Generated Programs. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, pp. 239–253. Springer, Heidelberg (2003)