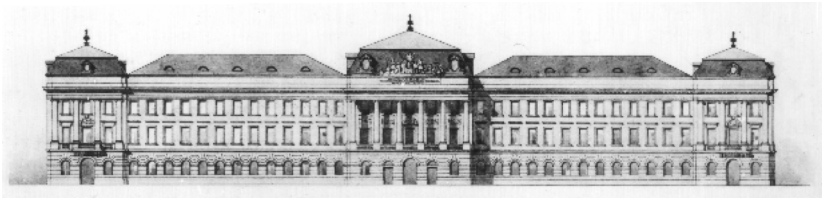
dbai

TECHNICAL REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

Evaluating Tree-Decomposition Based Algorithms for Answer Set Programming

DBAI-TR-2011-73

Michael Morak **Nysret Musliu** **Stefan Rümmele**
Stefan Woltran **Reinhard Pichler**

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

DBAI TECHNICAL REPORT
2011-09-30

TU WIEN
TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DBAI TECHNICAL REPORT
DBAI TECHNICAL REPORT DBAI-TR-2011-73, 2011-09-30

Evaluating Tree-Decomposition Based Algorithms for Answer Set Programming

Michael Morak **Nysret Musliu** **Reinhard Pichler**
Stefan Rümmele **Stefan Woltran** [1]

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 novel 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]Institute for Information Systems 184/2, Vienna University of Technology, Favoritenstrasse 9-11, 1040 Vienna, Austria. E-mail: [surname]@dbai.tuwien.ac.at

Acknowledgements: This work was supported by special fund ”Innovative Projekte 9006.09/008” of TU Vienna

1 Introduction

Many instances of constraint satisfaction problems and in general 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 [1] 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.

Tree-decomposition based algorithms have been used in several applications including probabilistic networks [2] or constraint satisfaction problems such as MAX-SAT [3]. The application area we shall focus on here is propositional Answer-Set Programming (ASP, for short) [4, 5] which is nowadays a well acknowledged paradigm for declarative problem solving as witnessed by 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 [6] for programs having an incidence graph 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:

$$\begin{aligned}
&\mathrm{r}(X) \vee \mathrm{g}(X) \vee \mathrm{b}(X) \leftarrow \mathrm{vertex}(X);\\
&\quad \bot \leftarrow \mathrm{r}(X), \mathrm{r}(Y), \mathrm{edge}(X,Y);\\
&\quad \bot \leftarrow \mathrm{g}(X), \mathrm{g}(Y), \mathrm{edge}(X,Y);\\
&\quad \bot \leftarrow \mathrm{b}(X), \mathrm{b}(Y), \mathrm{edge}(X,Y);
\end{aligned}$$

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); we refer to [7] (Section 3) for a collection of problems which can

[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*.

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

be represented with head-cycle free programs as opposed to problems which require the full power of ASP. However, above program contains variables and thus still has to be grounded. So-called grounders turn such programs into variable-free (i.e., propositional) ones which are then fed into ASP-solvers. In fact, the algorithms we discuss in this paper work on such variable-free programs.

A dynamic algorithm for general propositional ASP has already been presented in [8]. We provide here a new algorithm for the concept of head-cycle free programs. Their main differences are as follows: the algorithm from [8] is based on ideas from dynamic SAT algorithms [9] and explicitly takes care of the minimality checks required for ASP; thus it requires double-exponential time in the width of the provided tree decomposition. Our novel algorithm follows a more involved characterization [10] 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) [12], Min-Fill and Minimum Degree (see [13] for a survey on such heuristics). The system can be run with any combination of heuristics and dynamic algorithm, but it is up to the user to select them.

Although we focus here on algorithms for propositional programs, we emphasize at this point a valuable side-effect. For our example above, it turns out that in case 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.

In this paper we will present several experimental results that shed light on the combination of tree-decomposition heuristics and dynamic algorithms. In particular, we note that the TDDA algorithm is not always most efficient when the best heuristic for tree decomposition is used. This implies that the width of the tree decomposition is not always the most significant parameter for efficiency of our dynamic algorithms. Based on this observation we shall identify other important features of the tree decomposition and propose to use machine learning techniques for selection of the most promising dynamic algorithm during the solving process.

To summarize, the main contributions of this paper are as follows. We provide a novel algorithm for head-cycle free programs. We have integrated this algorithm, together with an existing standard algorithm, into a system for ASP where the tree decomposition can be obtained via three different heuristics. Experimental results show that the interplay between these two parts is crucial to obtain good performance and we give suggestions which features of tree decompositions (besides their width) are crucial. Based on these features we propose and evaluate the automated selection of the dynamic algorithm based on the input tree decomposition.

[3] A preliminary version of this system has been presented in [11], see `http://dbai.tuwien.ac.at/proj/dynasp`.

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:

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

where "$\neg$" 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 Π 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 [14], $M \subseteq \mathcal{A}$ 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 [10]. 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 Π 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 Π.

Example 2.1. *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, yielding five rules for the two vertices and their edge:*

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

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*) [15], 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 [15]. The *treewidth* of a graph $\mathcal{G}$, denoted by $tw(\mathcal{G})$, is the minimum width over all tree decompositions of $\mathcal{G}$.

[4] We omit strong negation as considered in [10]; our result easily extend to programs with strong negation.

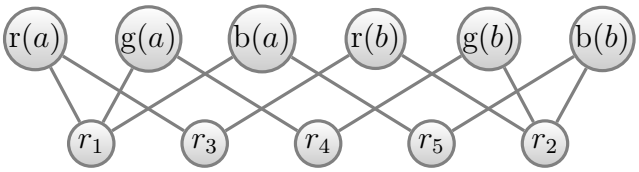


Figure 1: The incidence graph of the ground program of Example 2.1.

For a given graph and integer k, deciding whether the graph has treewidth at most k is NP-complete [16]. For computing tree decompositions, different complete [17, 18, 19] and heuristic methods have been proposed in the literature. Heuristic techniques are mainly 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) [12] 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 (a vertex of a graph is simplicial if its neighbors form a clique) 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. According to [18] the min-fill heuristic performs better than MCS and min-degree heuristic. For other types of heuristics and metaheuristic techniques based on the elimination ordering of nodes see [13].

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.

[5] See [9] for justifications why incidence graphs are favorable over other types of graphs.

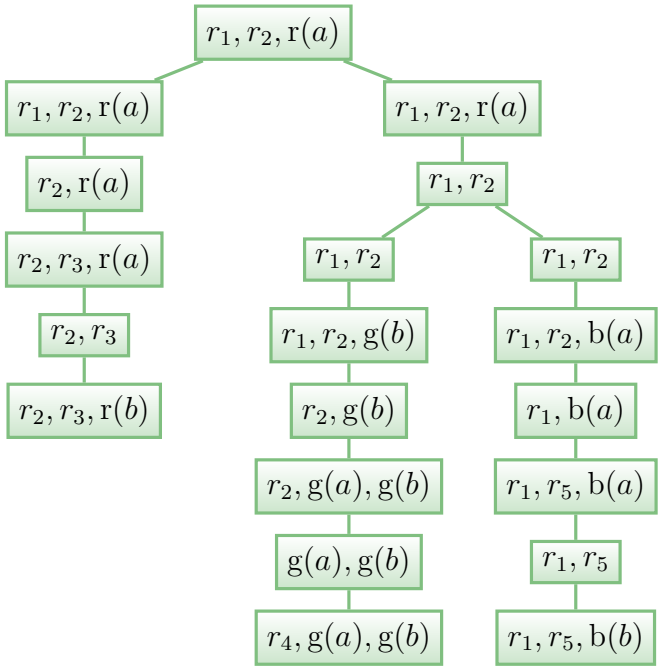


Figure 2: A normalized tree decomposition of the graph shown in Figure 1.

Figures 1 and 2 show the incidence graph of Example 2.1 and a corresponding tree decomposition.

3 Characterization of Answer Sets of Head-Cycle Free Programs

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). For HCFPs we consider the following graph as a main ingredient for such a data structure.

Definition 3.1. *Let $\Pi = (\mathcal{A}, \mathcal{R})$ be an HCFP, let $M \subseteq \mathcal{A}$, and let $\rho \subseteq \mathcal{R}$. Then the* derivation graph *$G = (V, E)$ induced by M and ρ is given by $V = M \cup \rho$ and E is the transitive closure of the edge set $E' = \{(b, r) : r \in \rho, b \in B^+(r) \cap M\} \cup \{(r, a) : r \in \rho, a \in H(r) \cap M\}$.*

The following theorem gives an alternative characterization of answer sets for HCFPs utilizing the derivation graph.

Theorem 3.2. *Let $\Pi = (\mathcal{A}, \mathcal{R})$ be an HCFP. Then, $M \subseteq \mathcal{A}$ is an answer set of Π iff the following holds:*

1. $M \in Mod(\Pi)$, *and*
2. *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 $|H(r) \cap M| = 1$.*

Before we prove this theorem, we recall the following characterization of answer sets for HCFPs due to Ben-Eliyahu and Dechter [10].

Theorem 3.3. *[10] Let $\Pi = (\mathcal{A}, \mathcal{R})$ be an HCFP. A set $M \subseteq \mathcal{A}$ is an answer set of Π iff the following conditions hold:*

1. $M \in Mod(\Pi)$, *and*
2. *there exists a function $f : \mathcal{A} \to \mathbb{N}^+$ such that for each atom $a \in M$, there exists a rule $r \in \mathcal{R}$ where*
   (a) $B^+(r) \subseteq M$, $B^-(r) \cap M = \emptyset$,
   (b) $H(r) \cap M = \{a\}$, *and*
   (c) $\forall b \in B^+(r) : f(b) < f(a)$.

By transforming the function f into a partial order and adding some rules in $\mathcal{R}$ to it, we get:

Corollary 3.4. *Let $\Pi = (\mathcal{A}, \mathcal{R})$ be an HCFP. A set $M \subseteq \mathcal{A}$ is an answer set of Π iff the following conditions hold:*

1. $M \in Mod(\Pi)$, *and*
2. *there exists a partial order $\trianglelefteq \subseteq (\mathcal{A} \cup \mathcal{R})^2$ such that for each atom $a \in M$, there exists a rule $r \in \mathcal{R}$ where*
   (a) $B^+(r) \subseteq M$, $B^-(r) \cap M = \emptyset$,
   (b) $H(r) \cap M = \{a\}$,
   (c) $\forall b \in B^+(r)$: $b \neq a \wedge b \trianglelefteq r$, *and*
   (d) $r \trianglelefteq a$.

Proof. Let $M \subseteq \mathcal{A}$ satisfy both conditions above and let $\trianglelefteq$ be the partial order from Condition 2. By removing all rules from $\trianglelefteq$, we construct a new partial order $\trianglelefteq' \subseteq \mathcal{A} \times \mathcal{A}$. Then, M satisfies the new condition

2') there exists a partial order $\trianglelefteq' \subseteq \mathcal{A} \times \mathcal{A}$ such that, for each atom $a \in M$, there exists a rule $r \in \mathcal{R}$ where
   (a) $B^+(r) \subseteq M$, $B^-(r) \cap M = \emptyset$,

(b) $H(r) \cap M = \{a\}$, and

(c) $\forall b \in B^+(r)$: $b \neq a \wedge b \trianglelefteq' a$.

Now let the total order $\leq$ on set $\mathcal{A}$ be any linear extension of $\trianglelefteq'$. It is then possible to find a function $f : \mathcal{A} \to \{1, \ldots, |\mathcal{A}|\}$ such that $\forall a, b \in \mathcal{A} : f(a) < f(b) \Leftrightarrow a < b$. Then f satisfies Condition 2 of Theorem 3.3 and therefore M is an answer set of Π.

Conversely, suppose that M is an answer set of Π. By Theorem 3.3 there exists a function f satisfying Condition 2. This function gives rise to a total order $\trianglelefteq'$ on $\mathcal{A}$. Since every total order is a partial order, $\trianglelefteq'$ satisfies Condition 2' above.

We now construct the partial order $\trianglelefteq \subseteq (\mathcal{A} \cup \mathcal{R})^2$ by extending $\trianglelefteq'$ in the following way: For each $a \in M$ let $r \in \mathcal{R}$ be one of the rules satisfying Condition 2'. Then we add (r, a) and for all $b \in B^+(r)$ the pair (b, r) to $\trianglelefteq$. Now $\trianglelefteq$ satisfies Condition 2 of this corollary and therefore M satisfies all conditions. □

Given this corollary, we easily get yet another corollary.

Corollary 3.5. *Let $\Pi = (\mathcal{A}, \mathcal{R})$ be an HCFP. A set $M \subseteq \mathcal{A}$ is an answer set of Π iff the following conditions hold:*

1. *$M \in Mod(\Pi)$, and*
2. *there exists a set $\rho \subseteq \mathcal{R}$ and a partial order $\trianglelefteq \subseteq (M \cup \rho)^2$, such that $M \subseteq \bigcup_{r \in \rho} H(r)$ and for all $r \in \rho$*
   - *(a) $B^+(r) \subseteq M$, $B^-(r) \cap M = \emptyset$,*
   - *(b) $|H(r) \cap M| = 1$,*
   - *(c) $\forall b \in B^+(r)$: $b \neq a \wedge b \trianglelefteq r$, and*
   - *(d) $\forall a \in H(r) \cap M$: $r \trianglelefteq a$.*

Proof. Let $M \subseteq \mathcal{A}$ satisfy both conditions above and let ρ be the set of rules and $\trianglelefteq$ be the partial order of Condition 2. $\trianglelefteq$ is also a partial order over the set $\trianglelefteq \subseteq (\mathcal{A} \cup \mathcal{R})^2$ which is needed in Condition 2 of Corollary 3.4. Since $M \subseteq \bigcup_{r \in \rho} H(r)$ there exists for each $a \in M$ at least one rule $r \in \rho$ such that $a \in H(r)$. From $|H(r) \cap M| = 1$ it follows that $H(r) \cap M = \{a\}$ and therefore r satisfies Condition 2 of Corollary 3.4. Hence, M is an answer set of Π.

Now suppose that M is an answer set of Π. By Corollary 3.4 there exists a partial order $\trianglelefteq$ satisfying Condition 2 of Corollary 3.4. Furthermore, for each atom $a \in M$ there exists a rule $r \in \mathcal{R}$ satisfying this condition. Let $\rho \subseteq \mathcal{R}$ be the set containing exactly these rules (i.e. $|\rho| = |M|$). Hence, for each $a \in M$ there exists exactly one $r \in \rho$ with $H(r) \cap M = \{a\}$ and therefore $M \subseteq \bigcup_{r \in \rho} H(r)$. Now let $\trianglelefteq'$ be the partial order obtained by restricting $\trianglelefteq$ to the set $M \cup \rho$. Then ρ and $\trianglelefteq'$ satisfy Condition 2 of Corollary 3.5 and hence, M satisfies all conditions of this corollary. □

We are now ready to prove Theorem 3.2.

Proof of Theorem 3.2. Let $M \subseteq \mathcal{A}$ satisfy the conditions of Theorem 3.2. Moreover, let ρ be the set of rules of Condition 2 of Theorem 3.2 and let $G = (V, E)$ be the derivation graph induced by M and ρ. Since G is acyclic and E is transitively closed, it gives rise to a partial order $\trianglelefteq \subseteq V^2 = (M \cup \rho)^2$ with $(a, b) \in E \Rightarrow a \trianglelefteq b$ and $a \trianglelefteq a$ for all $a \in V$. From the construction of E in Definition 3.1, it immediately follows that ρ and $\trianglelefteq$ satisfy Condition 2 of Corollary 3.5. Hence, M satisfies both conditions of that corollary and is therefore an answer set of Π.

Now suppose that M is an answer set. By Corollary 3.5 there exists a set ρ and a partial order $\trianglelefteq$ satisfying Condition 2 of that corollary. Let $G' = (V', E')$ be given by $V' = M \cup \rho$ and $E' = \trianglelefteq \setminus \{(a, a) : a \in V'\}$. Since $\trianglelefteq$ is a partial order, G' is acyclic. Let $G = (V, E)$ be the derivation graph induced by M and ρ. Then $V = V'$ and $E \subseteq E'$, therefore G is acyclic as well. Hence, ρ satisfies Condition 2 of Theorem 3.2 and M, by Corollary 3.5, satisfies both conditions. □

4 Dynamic Algorithm for Head-Cycle Free Programs

We now present a dynamic algorithm for checking if an HCFP has an answer set. This algorithm is based on the characterization of answer sets presented in Theorem 3.2. Given an HCFP $\Pi = (\mathcal{A}, \mathcal{R})$ together with a tree decomposition $\mathcal{T} = (T, \chi)$ of Π, Theorem 3.2 can be utilized in the following way. Instead of just guessing a solution candidate $M \subseteq \mathcal{A}$ and then checking whether it is an answer set (cf. [8]), we can now also guess the set $\rho \subseteq \mathcal{R}$. This makes the checking part easier and reduces the amount of information we need to store at each node. More precisely, given a node $t \in T$, a partial solution is represented by a tuple (G', S'), where G' is a derivation graph induced by $M = V(G') \cap \mathcal{A}$ and $\rho = V(G') \cap \mathcal{R}$, and $S' \subseteq \mathcal{R}$ represents the satisfied rules. Due to the connectedness condition of tree decompositions it is sufficient to restrict G' and S' to those atoms and rules visible in the bag $\chi(t)$, i.e., we store $G = G'[\chi(t)]$ and $S = S' \cap \chi(t)$. Note that by storing $G'[\chi(t)]$ we might lose the information that a rule $r \in V(G)$ already derived some atom a, i.e., the edge (r, a) is present in G' but not in G. Therefore the following definition introduces a special node which keeps track of this information. The same holds for the information whether a given node a has already been derived by some rule r.

Definition 4.1. *We extend derivation graphs $G = (V, E)$ by a special node $\blacklozenge$. For each rule $r \in V$ having at least one outgoing edge, we add an edge $(r, \blacklozenge)$. For each atom $a \in V$ having at least one incoming edge, we also add an edge $(\blacklozenge, a)$.*

W.l.o.g. we assume that the bags at L nodes and at the root node are empty, since this can always be achieved by adding the corresponding AI and RI nodes for leaves, respectively AR and RR nodes for the root. Following the characterization in Theorem 3.2, the root node has a partial solution (G', S') as defined above, if and only if Π has an answer set. Therefore, our new algorithm calculates all possible pairs (G, S) in a bottom-up traversal over the tree decomposition. When arriving at the empty root node, we can derive the tuple $G = (\{\blacklozenge\}, \emptyset)$ and $S = \emptyset$, if and only if Π has an answer set, i.e. the algorithm is sound and complete. In order to compute the pairs (G, S), the following operations are performed depending on the type of the considered node.

L nodes: Since these nodes are empty, the only tuple generated is the one with $G = (\{\blacklozenge\}, \emptyset)$ and $S = \emptyset$.

a-AI nodes: For each tuple at the child node, we generate two new tuples; (1) one where we guess $a \notin M$ and (2) one where we guess $a \in M$. In case of (1), we discard the tuple if $\neg a$ violates $B(r)$ for any rule $r \in V(G)$. Otherwise, we add those rules r' to S which are now satisfied because $a \in B^+(r)$. In case of (2), we discard the tuple if for any rule $r \in V(G)$, a violates $B(r)$; or if $(r, \blacklozenge) \in E(G)$ and $a \in H(r)$, i.e., $|H(r) \cap M| > 1$. Otherwise, we add those rules r' to S which are now satisfied because $a \in B^-(r)$. Additionally, we add a to $V(G)$ together with the appropriate edges according to Defs. 3.1 and 4.1. If the resulting graph contains a cycle, we discard the tuple.

For example, in node $\{r_2, r_3\}$ from our example in Figure 2, we encounter the tuple $(G, S) = (\{\blacklozenge, r_2\}, \{(r_2, \blacklozenge)\}), \emptyset)$. In node $\{r_2, r_3, \mathrm{b}(b)\}$ we thus derive the tuple $((\{\blacklozenge, r_2\}, \{(r_2, \blacklozenge)\}), \emptyset)$ for case (1), but no tuple for case (2), as $(r_2, \blacklozenge)$ exists in G.

r-RI nodes: For each tuple at the child node, we generate two new tuples; (1) one where we guess $r \notin \rho$ and (2) one where we guess $r \in \rho$. In case of (1), we add r to S if it is already satisfied, i.e., either there is an atom $a \in V(G)$ with $a \in B^-(r)$ or $a \in H(r)$, or there is an atom $a' \notin V(G)$ with $a' \in B^+(r)$. In case of (2), we discard the tuple if $B(r)$ is violated, i.e., either there is an atom $a \in V(G)$ with $a \in B^-(r)$ or there is an atom $a' \notin V(G)$ with $a' \in B^+(r)$. The tuple is also discarded if there are two different atoms $a, b \in V(G)$ that occur both in $H(r)$. Otherwise, we add r to S if it is already satisfied, i.e., there is an atom $a \in V(G)$ with $a \in H(r)$. Additionally, we add r to $V(G)$ together with the appropriate edges according to Defs. 3.1 and 4.1. If the resulting graph contains a cycle, we discard the tuple.

In node $\{\mathrm{g}(a), \mathrm{g}(b)\}$ of our example, we have a tuple $(G, S) = ((\{\blacklozenge\}, \emptyset), \emptyset)$, resulting in two tuples in the parent node, one in which r_2 is added to $V(G)$ and one where it is not. The remainder of the tuple is left unchanged.

a-AR nodes: For each tuple at the child node, we generate a new tuple. Note that we identify tuples that can now no longer be distinguished. We discard the tuple if $a \in V(G)$ but $(\blacklozenge, a) \notin E(G)$, i.e., a was guessed as part of the solution but was not derived by any rule. Otherwise, we remove a together with all incident edges from G. If a was not part of G, the tuple is left unchanged.

r-RR nodes: For each tuple at the child node, we generate a new tuple. Again we identify tuples that can now no longer be distinguished. We discard the tuple if $r \notin S$, i.e., r was not satisfied. Otherwise, we remove r from S and if applicable, we also remove r together with all incident edges from G.

B nodes: For each pair of a tuple (G_l, S_l) from the left child with a tuple (G_r, S_r) from the right child, we generate a new tuple. We discard the tuple if $V(G_l) \neq V(G_r)$, i.e., the guesses were different. Otherwise, we generate a new graph $G = (V(G_l), E(G_l) \cup E(G_r))$ and a new set $S = S_l \cup S_r$, i.e., a rule is satisfied if it was satisfied in at least one child. We discard the tuple if G contains a cycle or if there exists a rule $r \in V(G)$ with outgoing edges to two different atoms, i.e., r violates $|H(r) \cap M| = 1$.

It can be shown by structural induction that this algorithm correctly computes all pairs (G, S) at each node in the tree decomposition.

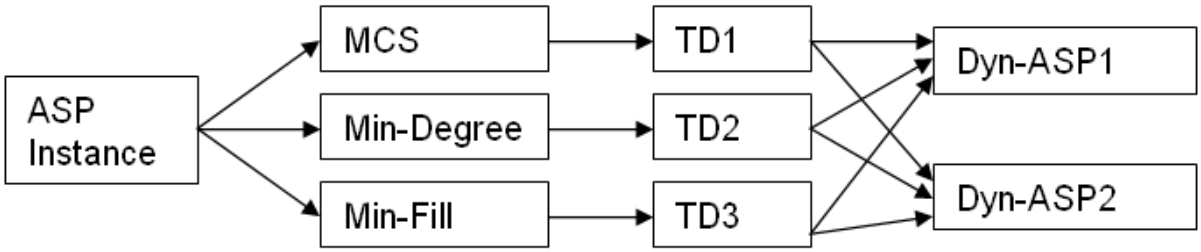


Figure 3: Architecture of the TDDA-based ASP solver

5 Evaluation of Tree Decompositions for ASP

In this section we give an extensive evaluation of dynamic algorithms based on tree decomposition for solving benchmark problems in answer set programming. In Figure 3 our solver based on tree decomposition and dynamic algorithm is presented, where Dyn-ASP1 refers to the standard algorithm from [8] and Dyn-ASP2 refers to the novel algorithm presented in the previous section. 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.

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 [20]. 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 [20], 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.

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

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 [21] 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.

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. [22] 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 classify 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*)

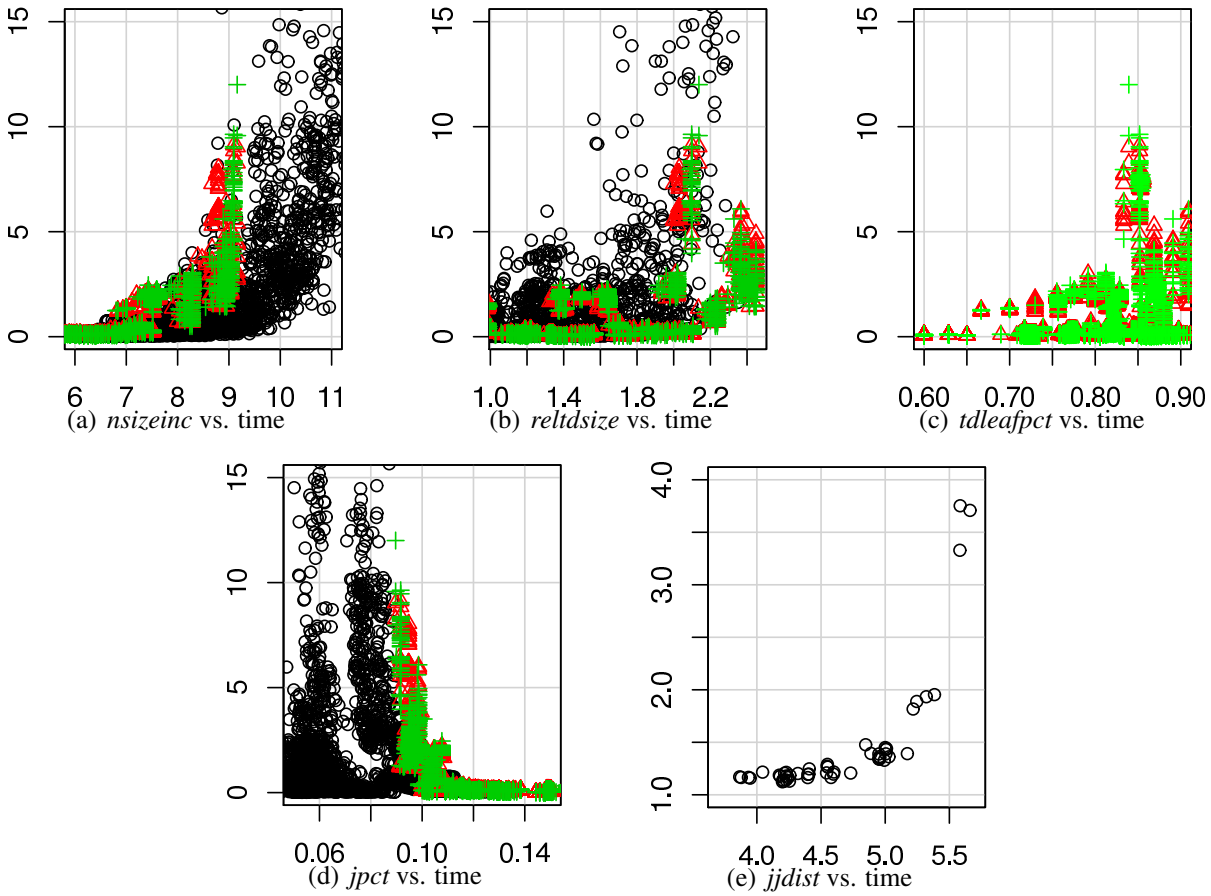


Figure 4: Every benchmark instance (i.e. each calculated tree decomposition) contributes one datapoint to the plots above. Usage of the MCS, Min-Degree and Min-Fill heuristics are represented by black circles, grey triangles and light grey crosses respectively. Note that the latter two almost always 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.

- Relative size of the tree decomposition (i.e. number of tree nodes) compared to the size (vertices + edges) of the incidence graph (*reltdsize*)

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, our new algorithm as described in Section 4 as well as the standard algorithm was 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 the thus generated dataset, using the WEKA toolkit [23], 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 underlying graph structure does not seem to provide enough information for a machine learning approach and calculating multiple tree decompositions is infeasible, as it is an expensive process. Based on the performance of the two algorithms, each tuple in the dataset was either labelled "Dyn-ASP1" or "Dyn-ASP2", respectively representing the standard dynamic algorithm and our new algorithm for HCFPs. Given the differences in runtime as shown in extracts in Table 1, 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 [24] 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)).

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

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 kNN classifier (wrongly) chose the Dyn-ASP2 algorithm in only 10.8% of the cases.

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%
kNN, k=10	Dyn-ASP1	10.8%
kNN, k=10	Dyn-ASP2	18.5%
Random forest	Dyn-ASP1	10.6%
Random forest	Dyn-ASP2	18.4%

6 Discussion

The new algorithm presented in this paper overall performs significantly better than the standard algorithm, which we are able to beat in almost 70% of our benchmark instances. A comparison to existing ASP systems can be found in [25], where it is shown that the presented TDDA approach indeed outperforms those systems on programs of small treewidth. However, in 30% of the cases, our new HCFP algorithm performs worse than the standard approach. Algorithm selection can be used to close this gap further, by trying to automatically select the better of the two algorithms at runtime.

Given the good results that were obtained by our machine learning approach, it is worth considering to implement such a method directly into our dynASP system. 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. [26, 27]), however these are specific to ASP, whereas our approach, using tree decomposition features for decisions, can generally be used for all TDDA approaches.

Furthermore, in our experiments several important tree decomposition features have been identified. 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 treewdith. For the domain of answer-set programming, a higher width can be compensated by such a decomposition, e.g. in our benchmarks, the MCS heuristic always produced the worst width, but (for small widths) actually speeds up our dynamic algorithms.

7 Conclusion

In this paper we have presented a new dynamic algorithm for head-cycle free programs, an important concept in the area of answer-set programming (ASP). By experimental evaluation we have studied the interplay between this new algorithm, an existing dynamic algorithm for ASP and three heuristics for the computation of tree decompositions. 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 conjecture that our observations are independent of the domain of answer set programming. To give an affirmative answer, we plan to evaluate tree-decomposition based algorithms for further problems from various areas.

References

[1] N. Robertson and P. D. Seymour, "Graph minors II: Algorithmic aspects of tree-width," *Journal Algorithms*, vol. 7, pp. 309–322, 1986.

[2] S. Lauritzen and D. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Journal of the Royal Statistical Society, Series B*, vol. 50, pp. 157–224, 1988.

[3] A. Koster, S. van Hoesel, and A. Kolen, "Solving partial constraint satisfaction problems with tree-decomposition," *Networks*, vol. 40(3), pp. 170–180, 2002.

[4] V. W. Marek and M. Truszczyński, "Stable Models and an Alternative Logic Programming Paradigm," in *The Logic Programming Paradigm – A 25-Year Perspective*. Springer, 1999, pp. 375–398.

[5] I. Niemelä, "Logic programming with stable model semantics as a constraint programming paradigm," *Ann. Math. Artif. Intell.*, vol. 25, no. 3–4, pp. 241–273, 1999.

[6] G. Gottlob, R. Pichler, and F. Wei, "Bounded treewidth as a key to tractability of knowledge representation and reasoning," in *Proc. AAAI'06*. AAAI Press, 2006, pp. 250–256.

[7] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The dlv system for knowledge representation and reasoning," *ACM Trans. Comput. Log.*, vol. 7, no. 3, pp. 499–562, 2006.

[8] M. Jakl, R. Pichler, and S. Woltran, "Answer-set programming with bounded treewidth," in *Proc. IJCAI'09*. AAAI Press, 2009, pp. 816–822.

[9] M. Samer and S. Szeider, "Algorithms for propositional model counting," *J. Discrete Algorithms*, vol. 8, no. 1, pp. 50–64, 2010.

[10] R. Ben-Eliyahu and R. Dechter, "Propositional semantics for disjunctive logic programs," *Ann. Math. Artif. Intell.*, vol. 12, pp. 53–87, 1994.

[11] M. Morak, R. Pichler, S. Rümmele, and S. Woltran, "A dynamic-programming based ASP-solver," in *Proc. JELIA'10*, 2010, pp. 369–372.

[12] R. Tarjan and M. Yannakakis, "Simple linear-time algorithm to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs," *SIAM J. Comput.*, vol. 13, pp. 566–579, 1984.

[13] H. L. Bodlaender and A. M. C. A. Koster, "Treewidth computations I. upper bounds," *Inf. Comput.*, vol. 208, no. 3, pp. 259–275, 2010.

[14] M. Gelfond and V. Lifschitz, "Classical negation in logic programs and disjunctive databases," *New Generation Comput.*, vol. 9, no. 3/4, pp. 365–386, 1991.

[15] T. Kloks, *Treewidth, computations and approximations*, ser. LNCS. Springer, 1994, vol. 842.

[16] S. Arnborg, D. G. Corneil, and A. Proskurowski, "Complexity of finding embeddings in a k-tree," *SIAM J. Alg. Disc. Meth.*, vol. 8, pp. 277–284, 1987.

[17] K. Shoikhet and D. Geiger, "A practical algorithm for finding optimal triangulations," in *Proc. AAAI'97*. AAAI Press / The MIT Press, 1997, pp. 185–190.

[18] V. Gogate and R. Dechter, "A complete anytime algorithm for treewidth," in *Proc. UAI 2004*. AUAI Press, 2004, pp. 201–208.

[19] E. Bachoore and H. Bodlaender, "A branch and bound algorithm for exact, upper, and lower bounds on treewidth," in *Proc. AAIM'06*, ser. LNCS, vol. 4041. Springer, 2006, pp. 255–266.

[20] Y. Zhao and F. Lin, "Answer set programming phase transition: A study on randomly generated programs," in *Proc. ICLP'03*, ser. LNCS, vol. 2916. Springer, 2003, pp. 239–253.

[21] A. Dermaku, T. Ganzow, G. Gottlob, B. J. McMahan, N. Musliu, and M. Samer, "Heuristic methods for hypertree decomposition," in *Proc. MICAI*, ser. LNCS, vol. 5317. Springer, 2008, pp. 1–11.

[22] K. Smith-Miles, "Cross-disciplinary perspectives on meta-learning for algorithm selection," *ACM Comput. Surv.*, vol. 41, no. 1, 2008.

[23] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009.

[24] M. A. Hall and L. A. Smith, "Practical feature subset selection for machine learning," in *Proc. ACSC'98*. Springer, 1998, pp. 181–191.

[25] M. Morak, "A dynamic programming-based answer set programming solver," Master's thesis, Vienna University of Technology, 2011.

[26] M. Balduccini, "Learning and using domain-specific heuristics in ASP solvers," *AI Commun.*, vol. 24, no. 2, pp. 147–164, 2011.

[27] M. Gebser, B. Kaufmann, and T. Schaub, "The conflict-driven answer set solver clasp: Progress report," in *Proc. LPNMR*, 2009, pp. 509–514.