A New Tree-Decomposition Based Algorithm for Answer Set Programming

Michael Morak, Nysret Musliu, Reinhard Pichler, Stefan Rümmele, Stefan Woltran
Institute of Information Systems, Vienna University of Technology
Contact: see http://dbai.tuwien.ac.at/staff

***Abstract*—A promising approach to tackle intractable problems is given by combining decomposition methods with dynamic programming algorithms. One such decomposition concept is tree decomposition. In this paper, we provide a new algorithm using this combined approach for solving reasoning problems in propositional answer set programming.**

I. 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 computed 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 (we use the term "dynamic algorithm" as a synonym for "dynamic *programming* algorithm") 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 II), but can also be used in general provided that 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.

Tree-decomposition based algorithms have been used in several applications, e.g., constraint satisfaction problems such as MAX-SAT [2]. The application area we shall focus on here is propositional Answer-Set Programming (ASP, for short) [3], [4] which is nowadays a well acknowledged paradigm for declarative problem solving as witnessed by many successful applications in the areas of AI and KR (see http://www.cs.uni-potsdam.de/~torsten/asp/ for a collection). 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 [5] for programs having an incidence graph of bounded treewidth. In this paper, we consider a certain subclass of programs, namely head-cycle free programs (HCFPs, for short) [6] (for more formal definitions, we again refer to Section II); 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 (either red, green or blue) for each node in the graph, and then check in the remaining three rules whether adjacent vertices have indeed different colors:

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

In fact, the above program is head-cycle free. Many NP-complete problems can be succinctly represented using HCFPs. We refer to [7] (Section 3) for a collection of problems which can be represented with HCFPs 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 algorithm we discuss in this paper work on such variable-free programs. 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 II 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 [8]. We provide here a new algorithm specifically designed for HCFPs. 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 [6] which applies to HCFPs 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.

II. 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: $a_1 \vee \cdots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$ where "$\neg$" is default negation, $n \geq 1$, $n \geq m \geq l$ and $a_i \in \mathcal{A}$ for all $1 \leq$

$i \leq n$. We omit strong negation as considered in [6]; our result easily extend to programs with strong negation. A rule $r \in \mathcal{R}$ of the above form consists of a head $H(r) = \{a_1, \dots, a_l\}$ and a body $B(r) = B^+(r) \cup B^-(r)$, given by $B^+(r) = \{a_{l+1}, \dots, a_m\}$ and $B^-(r) = \{a_{m+1}, \dots, a_n\}$. A set $M \subseteq \mathcal{A}$ is 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 [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 $N \subset M$, $N \in Mod(\Pi^M)$.

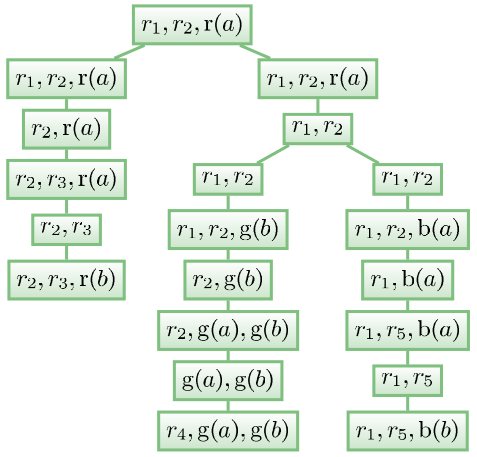
We consider here the class of *head-cycle free programs* (HCFPs) as introduced in [6]. A *dependency graph* of a program $\Pi = (\mathcal{A}, \mathcal{R})$ is given by $\mathcal{G} = (V, E)$, where $V = \mathcal{A}$ and $E = \{(p, q) \mid r \in \mathcal{R}, p \in B^+(r), q \in H(r)\}$. A program Π is 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 1: We provide the fully instantiated (i.e. ground) version of our introductory example from Section I over an input database with facts $\mathrm{vertex}(a)$, $\mathrm{vertex}(b)$, and $\mathrm{edge}(a, b)$.
$r1 : \mathrm{r}(a) \vee \mathrm{g}(a) \vee \mathrm{b}(a) \leftarrow \top$; $r2 : \mathrm{r}(b) \vee \mathrm{g}(b) \vee \mathrm{b}(b) \leftarrow \top$;
$r3 : \bot \leftarrow \mathrm{r}(a), \mathrm{r}(b)$; $r4 : \bot \leftarrow \mathrm{g}(a), \mathrm{g}(b)$;
$r5 : \bot \leftarrow \mathrm{b}(a), \mathrm{b}(b)$;

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*) [11], 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 [11]. The *treewidth* of a graph $\mathcal{G}$, denoted by $tw(\mathcal{G})$, is the minimum width over all tree decompositions of $\mathcal{G}$.

Tree Decompositions of Logic Programs. To build tree decompositions for programs, we use incidence graphs. 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 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 augmented with either an atom or a rule which is removed or added compared to the bag of the child node. Figure 1 shows a tree decomposition of Example 1.


$r_1, r_2, \mathrm{r}(a)$
$r_1, r_2, \mathrm{r}(a)$
$r_1, r_2, \mathrm{r}(a)$
$r_2, \mathrm{r}(a)$
r_1, r_2
$r_2, r_3, \mathrm{r}(a)$
r_1, r_2
r_1, r_2
r_2, r_3
$r_1, r_2, \mathrm{g}(b)$
$r_1, r_2, \mathrm{b}(a)$
$r_2, r_3, \mathrm{r}(b)$
$r_2, \mathrm{g}(b)$
$r_1, \mathrm{b}(a)$
$r_2, \mathrm{g}(a), \mathrm{g}(b)$
$r_1, r_5, \mathrm{b}(a)$
$\mathrm{g}(a), \mathrm{g}(b)$
r_1, r_5
$r_4, \mathrm{g}(a), \mathrm{g}(b)$
$r_1, r_5, \mathrm{b}(b)$
Fig. 1. A normalized tree decomposition of Example 1.

III. Algorithm for Head-Cycle Free Programs

Tree-decomposition based dynamic algorithms start at the leaf nodes and traverse the tree to the root. 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, it 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 program). For HCFPs we consider the following graph as a main ingredient for such a data structure.

Definition 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 $E' = \{(b, r) : r \in \rho, b \in B^+(r) \cap M\} \cup \{(r, a) : r \in \rho, a \in H(r) \cap M\}$.

We can now show a characterization of answer sets for HCFPs, which adapts the characterization in [6] to our needs.

Theorem 1: 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 $|H(r) \cap M| = 1$.

Based on this characterization, we now present a dynamic algorithm for checking if an HCFP has an answer set. Given an HCFP $\Pi = (\mathcal{A}, \mathcal{R})$ together with a tree decomposition $\mathcal{T} = (T, \chi)$ of Π. 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 Definition 2 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 2: 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. Following the characterization in Theorem 1, 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. By structural induction this algorithm can be shown to correctly compute all pairs at each 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. 1 and 2. If the resulting graph contains a cycle, we discard the tuple.

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. 1 and 2. If the resulting graph contains a cycle, we discard the tuple.

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 $|H(r) \cap M| = 1$.

IV. Conclusion

In this paper we have presented a new dynamic algorithm for head-cycle free programs. This new algorithm provides better theoretical runtime than previous tree-decomposition based algorithms for ASP. Preliminary experimental results are very promising and we plan to do an extensive comparison with state-of-the-art ASP-solvers in the future.

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

References

[1] N. Robertson and P. D. Seymour, "Graph minors. II. Algorithmic aspects of tree-width," *J. Algorithms*, vol. 7, no. 3, pp. 309–322, 1986.

[2] A. M. Koster, S. P. van Hoesel, and A. W. Kolen, "Solving partial constraint satisfaction problems with tree decomposition," *Networks*, vol. 40, no. 3, pp. 170–180, 2002.

[3] 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.

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

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

[6] R. Ben-Eliyahu and R. Dechter, "Propositional semantics for disjunctive logic programs," *Ann. Math. Artif. Intel.*, vol. 12, no. 1–2, pp. 53–87, 1994.

[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 *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] 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.

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