# Counting Answer Sets via Dynamic Programming

Johannes Fichte, Markus Hecher, Michael Morak, and Stefan Woltran

TU Wien, Vienna, Austria
lastname@dbai.tuwien.ac.at

**Abstract.** While the solution counting problem for propositional satisfiability (#SAT) has received renewed attention in recent years, this research trend has not affected other AI solving paradigms like answer set programming (ASP). Although ASP solvers are designed to enumerate all solutions, and counting can therefore be easily done, the involved materialization of all solutions is a clear bottleneck for the counting problem of ASP (#ASP). In this paper we propose dynamic programming-based #ASP algorithms that exploit the structure of the underlying (ground) ASP program. Experimental results for a prototype implementation show promise when compared to existing solvers.

## 1 Introduction

With the rise of efficient solvers, model counting for the propositional satisfiability problem (#SAT) [36,4] has received renewed attention in recent years (cf. e.g., [7,3]). Knowing the number of models of a propositional formula is a useful measurement, and can be used, inter alia, in the areas of machine learning, probabilistic reasoning, statistics, and combinatorics [30,34,10,37]. Various systems have been implemented that solve the #SAT problem; see e.g., [35,33].

Similar strides in efficiency have also been made in answer set programming (ASP) regarding the model existence problem [17,6], where efficient solvers are now readily available, e.g., [16,2]. ASP is a rule-based language that has found great success as it allows users to specify intuitive, fully-declarative problem descriptions, and is used in both industry and research. When using SAT solvers to evaluate a problem, the problem usually has to be rewritten into a SAT formula. While such SAT rewritings require a specialized algorithm for each particular problem, ASP as a rule-based formalism allows for the declarative specification of problem statements. The actual problem instance can then simply be given as an input database of ground facts.

For instance, a simple graph 2-colorability problem can be stated, using two rules, as follows:

$$color(V, red) \vee color(V, blue) \leftarrow vertex(V)$$
$$\bot \leftarrow edge(V1, V2), color(V1, C), color(V2, C)$$

Together with a graph, given as a set of facts of the form $edge(x, y)$ and $vertex(v)$, each answer set represents exactly one valid coloring of the graph. Evaluating ASP programs like the one above is usually a two-step process. First, a *grounder* instantiates the program, replacing all variables by domain constants, and then a *solver* evaluates

the ground program and computes the answer sets. While for SAT the model existence problem is NP-complete, the problem of evaluating ground (disjunctive) ASP programs is located on the second level of the polynomial hierarchy. Thus, ASP allows for efficient encodings of problems of higher complexity that typically arise in AI, like circumscription or diagnosis. Opposite to standard SAT solvers which simply decide the problem or deliver one (counter-)model, ASP systems are tailored to enumerate all answer sets. Due to this fact, the answer set counting problem (#ASP) has received far less attention than the #SAT problem. However, materializing all answer sets can be expensive and is not necessarily required for counting.

It is the aim of this paper to propose and evaluate a dynamic programming-based answer set counting algorithm that exploits the structure of the given (ground) input ASP program and avoids the costly materialisation of all answer sets. The importance of evaluating such parameterized algorithms in practice has recently been stressed [18]. Several works have shown that such an approach works well for #SAT, guaranteeing both favorable theoretical runtime bounds [15,32], as well as good practical performance in prototype implementations [24,31]. Jakl et al. [21] and Morak et al. [27] have introduced dynamic programming algorithms for deciding the answer set existence problem in linear time in the size of the input, where the enumeration of answer sets can be done with linear delay. This is accomplished by exploiting the structure of the given program via tree decompositions of its incidence graph. A tree decomposition of a graph (roughly) tries to arrange the graph into a tree by combining cyclic parts of the graph into single tree nodes. If the size of these tree nodes can be bounded by a (small) constant, then the problem can be evaluated efficiently by traversing the tree decomposition in a bottom-up manner, evaluating the answer set existence problem only locally for each node and combining the partial solutions. In this paper, we investigate how this idea can be used for counting without answer set materialization, and propose additional variations of the algorithm based on different graph representations of the program.

*Contributions.* The main contributions of this paper are:

1. We use dynamic programming on tree decompositions to solve the answer set counting problem for ASP. Three versions of the algorithms are proposed, based in part on different graph representation of the input.
2. We show that the algorithm exhibits favorable theoretical runtime bounds.
3. We provide a prototype implementation of our proposed algorithms, and give an experimental performance analysis and evaluation, comparing the solver to several existing solutions.

The remainder of the paper is structured as follows: In Section 2, we give some preliminaries on ASP and tree decompositions. Section 3 gives an overview of the general principles of dynamic programming algorithms on tree decompositions and then proceeds to give the proposed answer set counting algorithms. Finally, a prototype implementation is evaluated via experiments in Section 4. We close with some concluding remarks in Section 5.

## 2 Preliminaries

*Answer Set Programs.* A *ground answer set program* (or 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 $n \geq m \geq l$ and $a_i \in \mathcal{A}$ for all $1 \leq i \leq n$. A rule $r \in \mathcal{R}$ of form (1) consists of a head $H(r) = \{a_1, \ldots, a_l\}$ and a body 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 $B^+(r) \subseteq M$ together with $B^-(r) \cap M = \emptyset$ implies that $H(r) \cap M \neq \emptyset$. We denote the set of models of $r$ by $\mathit{models}(r)$ and the models of a program $\Pi = (\mathcal{A}, \mathcal{R})$ are given by $\mathit{models}(\Pi) = \bigcap_{r \in \mathcal{R}} \mathit{models}(r)$.

The reduct $\Pi^I$ of a program $\Pi$ with respect to a set of atoms $I \subseteq \mathcal{A}$ is the program $\Pi^I = \big(\mathcal{A}, \{r^I \mid r \in \mathcal{R}, B^-(r) \cap I = \emptyset)\}\big)$, where the reduct $r^I$ of a rule $r$ is the same rule without negative body, i.e., $H(r^I) = H(r)$, $B^+(r^I) = B^+(r)$, and $B^-(r^I) = \emptyset$. Following [17], $M \subseteq \mathcal{A}$ is an *answer set* of a program $\Pi = (\mathcal{A}, \mathcal{R})$ if $M \in \mathit{models}(\Pi)$ and for no $N \subsetneq M$, we have $N \in \mathit{models}(\Pi^M)$. The *consistency problem* of ASP (decide whether there exists an answer set for a given program $\Pi$) is $\Sigma_2^P$-complete [13].

*Fixed-Parameter Tractability.* We briefly recall the basic notions of fixed-parameter tractability. For more detailed information we refer to other sources, e.g. [11]. A *parameterized problem* $L$ is a subset of $\Sigma^* \times \mathbb{N}$ for some finite alphabet $\Sigma$. For an instance $(I, k) \in \Sigma^* \times \mathbb{N}$ we call $I$ the *main part* and $k$ the *parameter*. A problem $L$ is *fixed-parameter tractable (FPT)* if there exists a computable function $f$ and a constant $c$, such that there exists an algorithm that decides whether $(I, k) \in L$ in time $\mathcal{O}(f(k) \cdot \|I\|^c)$ where $\|I\|$ denotes the size of $I$. Such an algorithm is called an *fpt-algorithm*. A well studied parameter is the so-called treewidth.

*Tree Decomposition and Treewidth.* A *tree decomposition* of a graph $G = (V, E)$ is a pair $\mathcal{T} = (T, \chi)$, where $T$ is a rooted tree and $\chi$ is a labelling function with $\chi(t) \subseteq V$—we call $\chi(t)$ the *bag* of $t$—such that the following holds: (i) for each $v \in V$, there exists a $t \in T$, such that $v \in \chi(t)$; (ii) for each $\{v, w\} \in E$, there exists a $t \in T$, such that $\{v, w\} \subseteq \chi(t)$; and (iii) for each $r, s, t \in T$, such that $s$ lies on the path from $r$ to $t$, we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. The *width* of a tree decomposition is defined as the cardinality of its largest bag minus one. The *treewidth* of a graph $G$, denoted by $tw(G)$, is the minimum width over all tree decompositions of $G$. For arbitrary but fixed $w \geq 1$, it is feasible in linear time to decide if a graph has treewidth $\leq w$ and, if so, to compute a tree decomposition of width $w$.

A tree decomposition $(T, \chi)$ is called *normalized* (or *nice*) [22], if (i) each $t \in T$ has at most two children; (ii) for each $t \in T$ with two children $t'$ and $t''$, it holds that $\chi(t) = \chi(t') = \chi(t'')$; and (iii) for each $t \in T$ with exactly one child $t'$, the bags $\chi(t)$ and $\chi(t')$ differ in exactly one element, that is, $|\chi(t) \cap \chi(s)| = 1$. Every tree decomposition can be normalized in linear time without increasing its width [22]. We assume w.l.o.g. that normalized tree decompositions have root and leaf nodes whose bags are empty.

*Dynamic Programming on Tree Decompositions.* Dynamic programming is an established technique in the toolkit of parameterized complexity [28]. Especially, for problems parameterized by treewidth, dynamic programming algorithms on tree decompositions have been applied to many graph problems [5], as well as problems in Logic and Artificial Intelligence [29]. Dynamic algorithms on tree decompositions all share a common structure. The tree decomposition is traversed from the leaf nodes to the root node. At each node a subproblem is solved which consists of the part of the problem instance that is induced by the content of the bag of the current node. This results in a set of partial solutions (called *tuples*) which is propagated from the child nodes to the parent node. From these, the parent node then calculates the partial solutions its induced subproblem. Finally, at the root node there is a correspondence between the partial solutions of the root node and the solutions of the whole problem instance. An appropriate data structure to represent the partial solutions must be devised: this data structure must contain sufficient information to compute the representation of the partial solutions at each node from the corresponding representation at the child node(s). In addition, to ensure efficiency, the size of the data structure should only depend on the size of the bag (and not on the size of the entire problem instance).

*Tree Decompositions of Logic Programs.* To build tree decompositions for ground answer set programs $\Pi = (\mathcal{A}, \mathcal{R})$, we use two types of graph representations: (a) The *incidence graph* $G_{inc}(\Pi)$ of $\Pi$ is an undirected, bipartite graph $(\mathcal{A} \cup \mathcal{R}, E)$, where $E$ contains an edge $(a, r)$, iff atom $a$ occurs in rule $r$ of $\Pi$; and (b) the *primal graph* $G_{prim}(\Pi)$ of $\Pi$ is an undirected graph $(\mathcal{A}, E)$, where $E$ contains an edge $(a_i, a_j)$, iff there exists an $r \in \mathcal{R}$, such that both $a_i$ and $a_j$ appear in $r$. A tree decomposition of such a graph representation of a program $\Pi$ is called a tree decomposition of $\Pi$. The treewidth of an incidence graph of a program is at most the treewidth of its primal graph plus 1.

For normalized tree decompositions of programs, we can distinguish between six types of nodes: *leaf* (LEAF), *join* (JOIN), *atom introduction* (AI), *atom removal* (AR), *rule introduction* (RI), and *rule removal* (RR) node. A node is a leaf node if it has no child node. A node is a join node if it has two child nodes. A node is an atom introduction node if its bag contains one additional atom compared to the bag of its child node. Similarly, a node is an atom removal node, if its bag contains one less atom compared to the bag of its child node. Rule introduction and rule removal nodes are defined analogously to AI and AR nodes, where the difference between bags is an added or removed rule instead of an atom. The last four types, AI, AR, RI, and RR, will often be augmented with the element $e$ (either an atom or a rule) which is removed or added compared to the bag of the child node. For example AI($a$) denotes that atom $a$ is added and RR($r$) denotes that rule $r$ is removed. Notice that primal graph tree decompositions cannot contain RI and RR nodes.

When we refer to tree decompositions of programs, we will use the following notation: Let $\mathcal{T} = (T, \chi)$ be a tree decomposition of program $\Pi = (\mathcal{A}, \mathcal{R})$ and let $t \in T$ be a node. We denote the atoms occurring in $\chi(t)$ with $A_t$ and the rules occurring in $\chi(t)$ with $R_t$. These notions naturally extend from nodes $t$ to subtrees $T_t$ of $T$ (rooted at $t$). We say that a set of atoms $M$ entails a rule $r$ w.r.t. a tree node $t$, denoted $M \vDash_t r$, iff $M \in models(r|_t)$, where $r|_t$ is the rule obtained from $r$ by removing all literals

formed by atoms that are not contained in $\chi(t)$. For example, a rule where all literals are removed thus becomes $\bot \leftarrow \top$, that is, a contradiction.

*Counting Complexity.* Investigating the complexity of counting problems was initiated in [36]. Formally, a *counting problem* is presented using a *witness* function which for every input $x$ returns a set of *witnesses* for $x$. A *witness* function is a function $w\colon \Sigma^* \to \Gamma^{*,<\infty}$, where $\Sigma$ and $\Gamma$ are two alphabets; $\Gamma^{*,<\infty}$ are all finite strings in $\Gamma^*$. A *counting problem* is then defined as follows: given $x \in \Sigma^*$, find the cardinality $|w(x)|$. For a standard complexity class $\mathcal{C}$, define $\#\mathcal{C}$ as the class of all counting problems whose witness function $w$ satisfies (a) there is a polynomial $p(n)$ such that for every $x \in \Sigma^*$ and every $y \in w(x)$ we have $|y| \leq p(|x|)$; and (b) the problem "given $x$ and $y$, is $y \in w(x)$?" is in $\mathcal{C}$; see also [19].

## 3 Counting Answer Sets

In this section, we introduce our #ASP dynamic programming algorithms. Before doing so, we give a brief complexity-theoretic discussion of the #ASP problem.

**Complexity of #ASP** Before introducing the actual algorithms, we provide the following straight-forward result that shows that (under standard complexity-theoretic assumptions) the counting problem we consider here is strictly harder that #SAT. In fact the following result is a corollary of existing results that deal with the complexity of evaluating logic programs and the complexity of counting subset-minimal models of CNF formulas:

**Theorem 1.** *The counting problem #ASP is* #coNP*-complete*

*Proof.* Membership follows from the fact that, given a program $P$ and an interpretation $I$, checking whether $I$ is an answer of $P$ is coNP-complete, see e.g. [23]. Hardness is a direct consequence of #coNP-hardness for the problem of counting subset minimal models of a CNF formula [12], since answer sets of negation-free programs and subset-minimal models of CNF formulas are essentially the same objects.

We note that the counting complexity of ASP programs including optimization statements (i.e., where only optimal answer sets are counted w.r.t. a cost function) is slightly higher; exact results can be established employing hardness results from [20].

**Counting Algorithms for ASP** In order to simplify the presentation, we start by giving only the decision version of the algorithms, and extend them to counting algorithms later. We assume that, as input, the algorithm is given an answer set program $\Pi$, and a normalized tree decomposition $\mathcal{T}$ of $\Pi$. At each node $t$ of the tree decomposition, the algorithm will compute a set $\tau_t$ of tuples that represent the partial solutions. Given a node $t$, we denote by $t'$ its first child, and by $t''$ its second child. Thus, a full specification of the algorithm is given by describing how the set $\tau_t$ is derived from the sets $\tau_{t'}$ and $\tau_{t''}$.

We next present three dynamic programming algorithms. The first algorithm, INC, works on a tree decomposition of the incidence graph $G_{inc}(\Pi)$. The other two algorithms, PRIM and INVPRIM, use the primal graph $G_{prim}(\Pi)$.

*INC Algorithm.* The first algorithm, based on the incidence graph, is given in Figure 1. It contains a specification of how, for each node type of the tree decomposition, the set $\tau_t$ at tree node $t$ can be derived from the set $\tau_{t'}$ and $\tau_{t''}$ of its child nodes. A tuple in such a set is a triple $\langle M, S, C\rangle$, where $M$ represents a truth assignment of the atoms in $\chi(t)$, $S$ is a set of rules in $\chi(t)$ which are already satisfied, and $C$ is a set of certificates, that is, pairs $(A, R)$ of sets of atoms and rules, where $A \subset M$ represents a potential counter-model with respect to $\Pi^M$. The existence of such a tuple in $\tau_t$ witnesses the existence of a partial answer set for the program induced by the subtree rooted at $t$. Note that the decision version of the INC algorithm below restates the algorithm from [21].

Intuitively, the algorithm works as follows: In an atom introduction node $t$ for atom $a$, partial solutions of child nodes are extended in two ways: $a$ is either set to true and added to the set $M$ in the tuple, or set to false and not added to $M$. For each rule in $\chi(t)$, we verify whether the rule is satisfied by the choice on $a$. Also, all possible subsets of the new set $M$ are considered as certificates. By connectedness of tree decompositions, once a rule removal node removes a rule $r$, we can discard all tuples where $r$ is not yet satisfied, since $r$ will never appear again in an ancestor node. In join nodes, on agreeing sets $M$, the satisfied rules of the left child tuples and right child tuples are merged, since all of them are already satisfied. The certificates are updated in a similar manner. Continuing all the way up to the (empty) root node, this process guarantees that a surviving tuple witnesses that there is an assignment $M$ to all atoms of $\Pi$ that satisfies all the rules. Further, if the set of certificates of such a tuple is empty, then this witnesses that there does not exist a subset of $M$ that is a model of the reduct $\Pi^M$. For each node type of the tree decomposition, the construction of the set of tuples $\tau_t$ of node $t$ from the tuple sets $\tau_{t'}$ and $\tau_{t''}$ of its child nodes $t'$ and $t''$, which embodies the intuitive idea above, is given above. Note that we specify the set $\tau_t$ directly. Checking whether there exists an answer set is equivalent to checking whether, after a bottom-up traversal of the tree decomposition, the root node contains the tuple $\langle\emptyset, \emptyset, \emptyset\rangle$.

*PRIM and INVPRIM Algorithms.* A simplification of the above INC algorithm can be achieved if one considers the primal graph $G_{prim}(\Pi)$. Recall that by definition, a tree decomposition must contain each edge of the original graph in some node bag. Since the primal graph contains a clique between all atoms that participate in a rule $r$, it follows from the connectedness condition of a tree decomposition that there will be at least one node whose bag contains all the atoms of rule $r$. We denote by $\hat{R}_t$ all rules induced in that way by the bag $\chi(t)$ of node $t$. Thus, rule satisfaction can be checked immediately, and separate sets to keep track of satisfied rules are no longer needed in the tuple structure. For a given tree node $t$, the set $\tau_t$ thus contains simplified tuples of the form $\langle M, C\rangle$, where $M$ is the same as for INC, and $C$ is a set of sets $A \subset M$, that are again the same as for INC. Otherwise, the intuition of the algorithm below is similar to the one of INC. If primal and incidence graph have similar treewidth, the PRIM algorithm should benefit from this simplified logic. The PRIM algorithm is given

$$
\begin{aligned}
AI(a): \{\langle\ & M, S \cup \{r \mid r \in R_t, M \vDash_t r\}, \\
& \{(A, R \cup \{r \mid r \in R_t, A \vDash_t r^M\}) \mid (A,R) \in C\}\rangle \\
\mid\ & \langle M, S, C\rangle \in \tau_{t'}\} \cup \\
\{\langle\ & M, S \cup \{r \mid r \in R_t, M \vDash_t r\}, \\
& \{(A, R \cup \{r \mid r \in R_t, A \vDash_t r^M\}) \mid (A,R) \in C\} \cup \\
& \{(A, R \cup \{r \mid r \in R_t, A \vDash_t r^M\}) \\
& \qquad \mid (A', R) \in C, A = A' \cup \{a\}\} \cup \\
& \{(M', \{r \mid r \in R_t, M' \vDash_t r^M\})\}\rangle \\
\mid\ & \langle M', S, C\rangle \in \tau_{t'}, M = M' \cup \{a\}\}
\end{aligned}
$$

$$
\begin{aligned}
AR(a): \{\langle\ & M \setminus \{a\}, S, \\
& \{(A \setminus \{a\}, R) \mid (A,R) \in C\}\rangle \\
\mid\ & \langle M, S, C\rangle \in \tau_{t'}\}
\end{aligned}
$$

$$
\begin{aligned}
RI(r): \{\langle\ & M, S \cup \{s \mid s = r, M \vDash_t s\} \\
& \{(A, R \cup \{s \mid s = r, A \vDash_t s^M\}) \mid (A,R) \in C\}\rangle \\
\mid\ & \langle M, S, C\rangle \in \tau_{t'}\}
\end{aligned}
$$

$$
\begin{aligned}
RR(r): \{\langle\ & M, S \setminus \{r\} \\
& \{(A, R \setminus \{r\}) \mid (A,R) \in C, r \in R\}\rangle \\
\mid\ & \langle M, S, C\rangle \in \tau_{t'}, r \in S\}
\end{aligned}
$$

$$
\begin{aligned}
JOIN: \{\langle\ & M, S' \cup S'' \\
& \{(A, R' \cup R'') \mid (A, R') \in C', (A, R'') \in C''\} \cup \\
& \{(A, R \cup S'') \mid (A, R) \in C', A = M\} \cup \\
& \{(A, R \cup S') \mid (A, R) \in C'', A = M\}\rangle \\
\mid\ & \langle M, S', C'\rangle \in \tau_{t'}, \langle M, S'', C''\rangle \in \tau_{t''}\}
\end{aligned}
$$

$$
LEAF: \quad \{\langle \emptyset, \emptyset, \emptyset\rangle\}
$$

Fig. 1: INC Algorithm.

in Figure 2. Again, checking whether an answer set exists is equivalent to checking whether the tuple $\langle\emptyset,\emptyset\rangle$ exists at the root node of the tree decomposition.

$$
\begin{aligned}
AI(a):\ & \{\langle\ M, \\
& \quad\ \{A \mid A \in C, \forall r \in \hat{R}_t : A \vDash_t r^M\}\rangle, \\
& \ \mid\ \langle M, C\rangle \in \tau_{t'}, \forall r \in \hat{R}_t : M \vDash_t r\} \cup \\
& \{\langle\ M, \\
& \quad\ \{A \mid A \in C, \forall r \in \hat{R}_t : A \vDash_t r^M\} \cup \\
& \quad\ \{A \cup \{a\} \mid A \in C, \forall r \in \hat{R}_t : A \cup \{a\} \vDash_t r^M\} \cup \\
& \quad\ \{M'' \mid M'' = M', \forall r \in \hat{R}_t : M'' \vDash_t r^M\}\rangle \\
& \ \mid\ \langle M', C\rangle \in \tau_{t'}, M = M' \cup \{a\}\} \\
AR(a):\ & \{\langle\ M \setminus \{a\}, \{A \setminus \{a\} \mid A \in C\}\rangle \\
& \ \mid\ \langle M, C\rangle \in \tau_{t'}\} \\
JOIN:\ & \{\langle\ M, C' \cap C''\rangle \\
& \ \mid\ \langle M, C'\rangle \in \tau_{t'}, \langle M, C''\rangle \in \tau_{t''}\} \\
LEAF:\ & \quad \{\langle\emptyset,\emptyset\rangle\}
\end{aligned}
$$

Fig. 2: PRIM Algorithm.

The idea underlying the INVPRIM algorithm is to save in each tuple, instead of the set $C$ the set $\overline{C}$ of *inverse* certificates, i.e. sets $A \subset M$ that are surely not countermodels w.r.t. the reduct $\Pi^M$. We leave the straightforward adaptation of the above PRIM algorithm to the interested reader.

*Counting.* The three algorithms presented above, as-is, do not allow for model counting. However, a simple modification allows this. To this end, associate with each tuple $\bar{t}$ a number $n(\bar{t})$. For tuples in leaf nodes, set this number to $1$. For tuples in join nodes, let $n(\bar{t}) = n(\overline{t'}) \cdot n(\overline{t''})$, where $t'$ ($t''$) is the left (right) child's tuple that gave rise to $\bar{t}$. For introduction and removal nodes, let $f$ be the surjective function that maps a child tuple $\overline{t'}$ to tuple $\bar{t}$, according to the algorithms given above. Then, let $n(\bar{t}) = sum_{\overline{t'} \in f^{-1}(\bar{t})} n(\overline{t'})$ (i.e. if two tuples map to the same tuple, their counts are summed up).

*Correctness and Runtime.* The correctness proof of these algorithms is rather tedious, as each node type needs to be investigated separately. However, it is not difficult to see that a tuple at a node $t$ guarantees that there exists a model for the ASP sub-program induced by the subtree rooted at $t$, proving soundness. Conversely, it can be shown that each candidate answer set is indeed evaluated while traversing the tree decomposition, which proves completeness. Regarding the theoretical runtime bounds, the algorithms all work in time $O(2^{2^w} \cdot n)$, where $w$ is the width of the underlying tree decomposition, and $n$ is the size of $\Pi$.

An interesting observation is that, by dropping all the logic concerning the certificates from the above algorithms, one obtains a pure satisfiability checking algorithm, similar to those proposed in [32].

## 4 Experimental Evaluation

We performed experiments to evaluate the efficiency of our approach and its various algorithm configurations (PRIM, INVPRIM, INC) on programs where we can heuristically find a decomposition of small width reasonably fast. In fact, programs of small width exist in practice as real-world graphs often admit tree decompositions of small width. Further, we compared our approach with a modern ASP solver, recent #SAT solvers, and a QBF solver. The solvers tested include our own prototype implementation, which we refer to as DynASP, and the existing solvers Cachet 1.21 [33] (a SAT model counter), DepQBF0[1] (a QBF solver), Clasp 3.1.4 [16] (an ASP solver), and SharpSAT 12.08 [35] (a SAT model counter).

We used both random and structured instances for benchmark sets, of which we give a description below. The random instances (SAT-TGRID, 2QBF-TGRID, ASP-TGRID, 2ASP-TGRID) were designed to have a high number of variables and solutions, but with certain probability a treewidth larger than some fixed $k$. Therefore, let $k$ and $\ell$ be some positive integers and $p$ a rational number such that $0 < p \leq 1$. An instance $F$ of SAT-TGRID$(k, l, p)$ consists of the set $V = \{ (1,1), \ldots, (1,\ell), (2,\ell), \ldots, (k,\ell) \}$ of variables and with probability $p$ for each variable $(i,j)$ such that $1 < i \leq k$ and $1 < j \leq \ell$ a clause $s_1(i,j)$, $s_2(i-1,j)$, $s_3(i,j-1)$, a clause $s_4(i,j)$, $s_5(i-1,j)$, $s_6(i-1,j-1)$, and a clause $s_7(i,j)$, $s_8(i-1,j-1)$, $s_9(i,j-1)$ where $s_i \in \{-,+\}$ is selected with probability one half. In that way, such an instance has an underlying dependency graph that consists of various triangles forming for probability $p = 1$ a graph that has a grid as subgraph. Let $q$ be a rational number such that $0 < q \leq 1$. An instance of the set 2QBF-TGRID$(k, l, p, q)$ is of the form $\exists V_1.\forall V_2.F$ where a variable belongs to $V_1$ with probability $q$ and to $V_2$ otherwise. Instances of the sets ASP-TGRID or 2ASP-TGRID have been constructed in a similar way, however, as an ASP program instead of a formula. Note that the number of answer sets and the number of satisfiable assignments correspond. We fixed the parameters to $p = 0.85$, $k = 3$, and $l \in \{40, 80, \ldots, 400\}$ to obtain instances that have with high probability a small fixed width, a high number of variables and solutions. Further, we took fixed random seeds and generated 10 instances to ensure a certain randomness. The structured instances model various graph problems (2COL, 3COL, DS, ST CVC, SVC) on real world mass transit graphs of 82 cities, metropolitan areas, or countries (e.g., Beijing, Berlin, Shanghai, and Singapore). The graphs have been extracted from publicly available mass transit data feeds [8] using gtfs2graphs [14] and split by transportation type, e.g., train, metro, tram, combinations. We heuristically computed tree decompositions [9] and obtained relatively fast decompositions of small width unless detailed bus networks were present. The encoding for 2COL counts all minimal sets $S$ of vertices such that there are two sets $F$ and $S$ where no two neighboring vertices $v$ and $w$ belong to $F$ for a given

[1] Since DepQBF [26] does not support counting by default, we implemented a naive counting approach into DepQBF using methods described in [25], which we call DepQBF0.

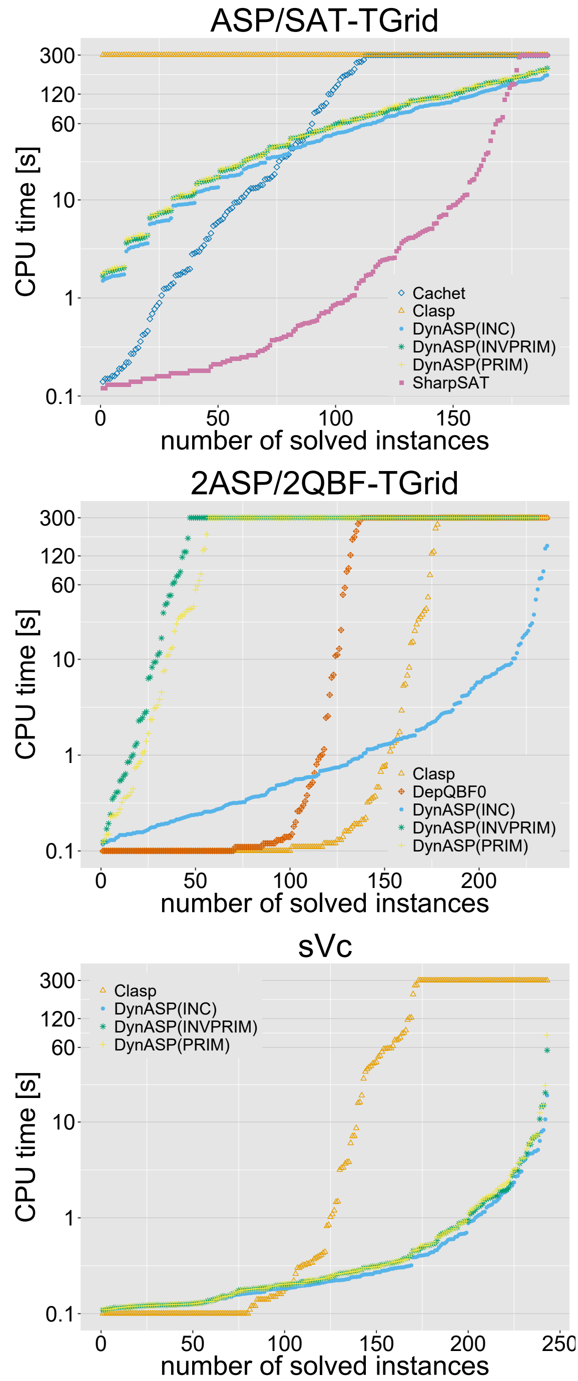


Fig. 3: Visualization of benchmark results of randomly generated instance sets and a selected real-world instance set.

input graph. The encoding for 3COL models to count all 3-colorings. The encoding for DS models to count all minimal dominating sets. The encoding for ST models to count all Steiner trees. The encoding for CVC asks to count all minimal vertex covers. The encoding for SVC models to count all subset-minimal vertex covers.

We ran the experiments on an Ubuntu 12.04 Linux cluster of 3 nodes with two AMD Opteron 6176 SE CPUs of 12 physical cores each at 2.3Ghz clock speed and 128GB RAM. Input instances were given to the solvers via a shared memory file system. During a run we limited the available memory to 4GB RAM and the CPU time to 300 seconds. We used default options for cachet and SharpSAT, option "–qdc" for DepQBF0, option "–stats=2 –opt-mode=optN -n 0 –opt-strategy=bb/usc -q" for clasp, and will refer to the different variants of our prototype implementation as DynASP(PRIM), DynASP(INVPRIM) and DynASP(INC). Since we cannot expect to solve instances of high treewidth efficiently, we restricted the instances such that we were able to heuristically find a decomposition of width smaller than 20 within 60 seconds.

In order to draw conclusions about the efficiency of our approach, we mainly inspected the (total cpu) running time[2] and number of timeouts on the random and structured benchmark sets. Figure 3 illustrates the running times of the solvers on the various random instance sets and a selected structured instance set as a cactus plot. Table 1 reports on the average running times, number of solved instances, and number of timeouts of the solvers on the considered structured instance sets.

SAT-TGRID and ASP-TGRID: Cachet solved 125 instances. Clasp always timed out for both configurations (branch and bound; and unsatisfiable core). A reason could be the high number of solutions as Clasp counts the models by enumerating them. DynASP($\cdot$) solved each instance within at most 270 seconds (on average 67 seconds). The best configuration with respect to runtime was PRIM. However, the running times of the different configurations were close. We observed as expected a sub-polynomial growth in the runtime with an increasing number of solutions. SharpSAT timed out on 3 instances and ran into a memory out on 7 instances, but solved most of the instances quite fast. Half of the instances were solved within 1 seconds and more than 80% of the instances within 10 seconds, and about 9% of the instances took more than 100 seconds. The number of solutions does not have an impact on the runtime of SharpSAT. SharpSAT was the fastest solver in total, however, DynASP($\cdot$) solved all instances. The results are illustrated in the two left graphs of Figure 3.

2QBF-TGRID and 2ASP-TGRID: Clasp solved more than half of the instances in less than 1 second, however, timed out on 59 instances. DepQBF0 shows a similar behavior as Clasp, which is not surprising as both solvers count the number of solutions by enumerating them and hence the number of solutions has a significant impact on the runtime of the solver. However, Clasp is throughout faster than DepQBF0. DynASP(INC) solved half of the instances within less than 1 second, about 92% of the instances within less than 10 seconds, and provided solutions also if the instance had a large number of

---

[2] The runtime for DynASP($\cdot$) includes decomposition times. Note that we randomly generated three fixed seeds for the decomposition computation to allow a certain variance in decomposition features [1]. When evaluating the results, we constructed the average on the seeds per instance.

| | 2COL | 3COL | DS | ST | CVC | SVC |
|---|---|---|---|---|---|---|
| Clasp | 31.72 (21) | 0.10(0) | 8.99 (3) | 0.21 (0) | 29.88 (21) | 98.34 (71) |
| INC | 1.43 (0) | 0.58(0) | 0.54 (0) | 115.02 (498) | 0.68 (0) | 0.78 (0) |
| INVP | 1.50 (0) | 0.47(0) | 0.79 (0) | 91.92 (248) | 0.99 (0) | 1.15 (0) |
| PRIM | 1.54 (0) | 0.53(0) | 0.68 (0) | 79.36 (221) | 0.99 (0) | 1.30 (0) |

Table 1: Runtime results on real-world graph instance sets and number of timeouts in brackets. Runtimes given in sec.

answer sets. DynASP(INVPRIM) and DynASP(PRIM) quickly produced timeouts due to a significantly larger width of the computed decompositions.

Structured instances: Clasp solved most of the structured instances reasonably fast. However, the number of solutions has again, similar to the random setting, a significant impact on its performance. If the instance has a small number of solutions, then Clasp yields the number almost instantly. If the number of solutions was very high, then Clasp timed out. DynASP($\cdot$) solved for each set but the set ST more than 80% of the instances in less than 1 second and the remaining instances in less than 100 seconds. For ST the situation was different. Half of the instances were solved in less than 10 seconds and a little less than the other half timed out. Similar to the random setting, DynASP($\cdot$) ran still fast on instances with a large number of solutions. Whenever the instance had relatively few solutions Clasp was faster, otherwise DynASP($\cdot$) (e.g., SVC) was faster.

The empirical results of the benchmarks confirm that our DynASP prototype works reasonably fast under the assumption that the input instance has small treewidth. The comparison to state-of-the-art ASP and QBF solvers shows that our solver has an advantage if we have to count many solutions, whereas Clasp and DepQBF0 perform well if the number of instances is relatively small. However, DynASP($\cdot$) is still reasonably fast on structured instances with few solutions as it yields the number of solutions mostly within less than 10 seconds. We observed that DynASP(INC) seems to be the overall best solving algorithm in our setting, which indicates that the smaller treewidth obtained by decomposing the incidence graph generally outweighs the benefits of simpler solving algorithms for the primal graph. A comparison to recent #SAT solvers suggests that dedicated #SAT algorithms are somewhat faster on random SAT formulas of small treewidth than our decomposition based approach, which is, however, not particularly surprising since our implementation is equipped to handle the full ASP semantics. The results indicate that our approach seem to be suitable for practical use, at least for certain classes of instances with low treewidth, and hence could fit into a portfolio-based solver.

## 5 Conclusions

In this paper, we have presented several dynamic programming algorithms for counting answer sets of logic programs, and compared a prototype implementation to existing solvers. For large instances of low treewidth, our implementation proved to be com-

petitive both against classical ASP solvers that need to materialize all answer sets in order to count them, as well as specific counting algorithms developed for SAT. These promising results confirm the importance of evaluating parameterized algorithms in practice [18]. Future work includes extending our algorithms to weighted model counting, to solve e.g., the Bayesian inference problem.

**Acknowledgements**

The authors gratefully acknowledge support by the Austrian Science Fund (FWF), Grant Y698. The first author is also affiliated with the Institute of Computer Science and Computational Science at University of Potsdam, Germany.

## References

1. Abseher, M., Dusberger, F., Musliu, N., Woltran, S.: Improving the efficiency of dynamic programming on tree decompositions via machine learning. In: IJCAI'15 (2015)
2. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A native ASP solver based on constraint learning. In: LPNMR'13 (2013)
3. Aziz, R.A., Chu, G., Muise, C.J., Stuckey, P.J.: #(∃)SAT: Projected Model Counting. In: SAT'15 (2015)
4. Bayardo Jr., R.J., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: AAAI'97 (1997)
5. Bodlaender, H.L.: Treewidth: Algorithmoc techniques and results. In: MFCS'97 (1997)
6. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM 54(12) (2011)
7. Chakraborty, S., Fried, D., Meel, K.S., Vardi, M.Y.: From weighted to unweighted model counting. In: IJCAI'15 (2015),
8. Czebotar, J.e.a.: GTFS data exchange. http://www.gtfs-data-exchange.com (2016)
9. Dermaku, A., Ganzow, T., Gottlob, G., McMahan, B., Musliu, N., Samer, M.: Heuristic methods for hypertree decomposition. In: MICAI'08 (2008)
10. Domshlak, C., Hoffmann, J.: Probabilistic planning via heuristic forward search and weighted model counting. J. Artif. Intell. Res. 30 (2007)
11. Downey, R.G., Fellows, M.R.: Fundamentals of Parameterized Complexity. Springer Verlag (2013)
12. Durand, A., Hermann, M., Kolaitis, P.G.: Subtractive reductions and complete problems for counting complexity classes. Theoretical Computer Science 340(3) (2005)
13. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. AMAI 15(3–4) (1995)
14. Fichte, J.K.: daajoe/gtfs2graphs – a GTFS transit feed to graph format converter (2016), https://github.com/daajoe/gtfs2graphs
15. Fischer, E., Makowsky, J.A., Ravve, E.V.: Counting truth assignments of formulas of bounded tree-width or clique-width. Discr. Appl. Math. 154(4) (2008), http://dx.doi.org/10.1016/j.dam.2006.06.020
16. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artificial Intelligence 187–188 (2012)
17. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP'88 (1988)

18. Gutin, G.: Should we care about huge imbalance in parameterized algorithmics? The Parameterized Complexity Newsletter (Dec 2015)
19. Hemaspaandra, L.A., Vollmer, H.: The satanic notations: Counting classes beyond #P and other definitional adventures. SIGACT News (1995)
20. Hermann, M., Pichler, R.: Complexity of counting the optimal solutions. Theoretical Computer Science 410(38–40) (2009)
21. Jakl, M., Pichler, R., Woltran, S.: Answer-set programming with bounded treewidth. In: IJCAI'09 (2009)
22. Kloks, T.: Treewidth, computations and approximations, LNCS, vol. 842. Springer (1994)
23. Koch, C., Leone, N.: Stable model checking made easy. In: IJCAI'99 (1999)
24. Li, W., Poupart, P., van Beek, P.: Exploiting structure in weighted model counting approaches to probabilistic inference. J. Artif. Intell. Res. 40 (2011)
25. Lonsing, F.: Personal communication (2015)
26. Lonsing, F., Biere, A.: DepQBF: A dependency-aware QBF solver system description. J. Sat., Bool. Model. and Comp. 7 (2010)
27. Morak, M., Pichler, R., Rümmele, S., Woltran, S.: A dynamic-programming based asp-solver. In: JELIA'10 (2010)
28. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford University Press (2006)
29. Pichler, R., Rümmele, S., Woltran, S.: Belief revision with bounded treewidth. In: LPNMR'09 (2009)
30. Roth, D.: On the hardness of approximate reasoning. Artificial Intelligence 82(1–2) (1996)
31. Sæther, S.H., Telle, J.A., Vatshelle, M.: Solving #SAT and MAXSAT by dynamic programming. J. Artif. Intell. Res. 51 (2015)
32. Samer, M., Szeider, S.: Algorithms for propositional model counting. J. Discrete Algorithms 8(1) (2010)
33. Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: SAT'04 (2004)
34. Sang, T., Beame, P., Kautz, H.A.: Performing bayesian inference by weighted model counting. In: AAAI'05 (2005)
35. Thurley, M.: sharpSAT – counting models with advanced component caching and implicit BCP. In: SAT'06 (2006)
36. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM J. Comput. 8(3) (1979)
37. Xue, Y., Choi, A., Darwiche, A.: Basing decisions on sentences in decision diagrams. In: AAAI'12 (2012)