Answer Set Solving using Tree Decompositions and Dynamic Programming
— The DynASP2 System —

Johannes Fichte  Markus Hecher  Michael Morak  Stefan Woltran

DBAI-TR-2016-101
Answer Set Solving using Tree Decompositions and Dynamic Programming
— The DynASP2 System —

Markus Hecher 1        Johannes Fichte 1        Michael Morak 1
Stefan Woltran 1

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 clearly a bottleneck for the counting problem of Answer Set Programming (#ASP).

In this paper, we propose dynamic programming algorithms for #ASP that exploit the structure of the underlying (ground) ASP program via the notion of tree decompositions. Experimental results for a prototype implementation show that our counting algorithms are very competitive when compared to existing solvers under the assumption that the tree width of the considered instance is sufficiently low.

1TU Wien. E-mail: {hecher, jfichte, morak, woltran}@dbai.tuwien.ac.at

Acknowledgements: The authors gratefully acknowledge support by the Austrian Science Fund (FWF), Grant Y698.

Copyright © 2016 by the authors
Contents

1 Introduction ................................................. 4
  1.1 Related Work ........................................... 5
  1.2 Contributions .......................................... 6
  1.3 Prior Work and Paper Organization ...................... 6

2 Background ..................................................... 7
  2.1 Answer Set Programming ................................. 7
  2.2 Computational Complexity ................................ 9
  2.3 Graphs .................................................... 11

3 Decomposition-based #ASP Solving ......................... 13
  3.1 Tree Decompositions .................................... 13
  3.2 Tree Decompositions of ASP Programs .................. 17
  3.3 Dynamic Programming Based Answer Set Solving ....... 18
  3.4 Solving CONSISTENCY, BRAVE REASONING, and COMPUTEAS ... 20
    3.4.1 Algorithms PRIM and INVPRIM ...................... 21
    3.4.2 Algorithms SINC and INVSINC ...................... 25
    3.4.3 Algorithms INC and INVINC ......................... 30
  3.5 Correctness and Runtime ................................ 31
  3.6 Extensions ................................................ 32
    3.6.1 Extensions to Optimization ......................... 32
    3.6.2 Extensions to Counting .............................. 32
  3.7 Comparison of the Algorithms .......................... 33

4 Towards Implementation: DynASP 2 and DynASP 2.5 .......... 35
  4.1 A quick hands-on quide to DynASP 2 .................... 35
  4.2 Implementation Tricks ................................... 36
  4.3 Extended Implementation: DynASP 2.5 ................. 37
    4.3.1 A quick hands-on quide to DynASP 2.5 .......... 38

5 Experimental Evaluation ..................................... 40
  5.1 Setup / Experimental Environment ....................... 40
  5.2 Instances ................................................ 41
  5.3 Results .................................................. 43
    5.3.1 COMPUTEAS ........................................ 43
    5.3.2 #Asp ................................................ 44
    5.3.3 Counting ASP models – DynASP 2 vs. DynASP 2.5 .... 47

6 Conclusion .................................................... 49
1 Introduction

With the rise of efficient solvers, model counting for the propositional satisfiability problem (\#SAT) \cite{50, 97} has received renewed attention in recent years (cf., e.g., \cite{10, 24}). Knowing the number of models of a propositional formula can be used, for example, in the areas of machine learning, probabilistic reasoning, statistics, and combinatorics \cite{33, 82, 87, 99}. Various systems have been implemented that solve the \#SAT problem; see, e.g., \cite{86, 96}. Answer set programming (ASP) is a rule-based language that allows users to specify intuitive, fully-declarative problem descriptions \cite{22, 48}. ASP has found great success in AI and is widely used in research as well as in industry. Significant efficiency improvements have also been made in solving the model existence problem of ASP, where efficient solvers are now readily available \cite{6, 44}. In contrast to standard SAT solvers which simply decide the problem or deliver one (counter-)model, however, ASP systems are built to enumerate all answer sets. Due to this, 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 necessary for counting.

In this paper, we propose and evaluate novel dynamic programming-based answer set counting algorithms that exploit, using the notion of tree decompositions, the underlying graph structure of the given (ground) input ASP program and avoid the costly materialization of all answer sets. We use ideas by Jakl, Pichler, and Woltran \cite{62} and Morak et al. \cite{74}, where algorithms that exploit certain tree decompositions for a similar but very restricted setting are presented. In fact, these algorithms can handle only ASP programs that consist of disjunctive rules. In contrast, we support the full ground ASP language, including choice rules and weight rules, which requires more sophisticated algorithms and novel data-structures. Further, we make use of advanced techniques for dynamic programming on tree decompositions originally proposed in previous work \cite{15}. Our solver also supports optimization statements, that allow us not only to find or count arbitrary models, but also to find and count optimal models only. To this end, we introduce different graph representations of ground ASP programs and compute tree decompositions of them. Intuitively, the graph is arranged into a labelled tree by combining cyclic parts of the graph into tree nodes. If the minimum size of these tree nodes (called \textit{treewidth}) can be bounded by a (small) constant, then the problem can be evaluated very efficiently by traversing the tree decomposition in a bottom-up manner, computing answer sets locally for each tree node, and counting the number of solutions in the process, without materializing individual answer sets. In fact, our algorithms solve the \#ASP problem in linear time in the size of the input for programs with bounded treewidth.

The general idea of our dynamic programming algorithms is as follows: First, we obtain a tree decomposition. Each tree node of the tree decomposition represents a sub-problem of the entire ASP program. In a bottom-up manner, we compute partial solutions for the given sub-problem. We hand these partial solutions to the parent node, and there, extend them to partial solutions of the parent node’s sub-problem. In the spirit of a dynamic programming algorithm, we can discard partial solutions that cannot be extended to a full solution of the entire ASP program very early.
Why ASP?

When using SAT solvers to evaluate a problem, the problem usually has to be encoded into a SAT formula. Such SAT encodings thus require specialized encoding algorithms for each particular problem. On the other hand, ASP, as a rule-based formalism, allows the declarative specification of problem statements. The actual problem instance can then simply be given as ground facts. With efficient ASP model counters, the intuitive, rule-based ASP language can be used in all application areas mentioned above.

Example 1.1. Consider the non-ground program consisting of the following rules:

\[
\begin{align*}
    \text{color}(V, \text{red}) \lor \text{color}(V, \text{blue}) & \leftarrow \text{vertex}(V). \\
    \bot & \leftarrow \text{edge}(V_1, V_2), \text{color}(V_1, C), \text{color}(V_2, C), V_1 \neq V_2.
\end{align*}
\]

The graph \( G = (V, E) \) will be given as a set of rules:

\[
\begin{align*}
    \text{edge}(v, w). & \quad (\text{for every } vw \in E) \\
    \text{vertex}(v). & \quad (\text{for every } v \in V)
\end{align*}
\]

It is easy to see that the program encodes the problem GRAPH 2-COLORING (see Section 2.3 for definition). For every \( v \in V \) the atoms \( \text{color}(v, \text{red}) \) and \( \text{color}(v, \text{blue}) \) of an answer set of the program represent a 2-coloring of the input graph and vice versa. Consequently, the number of answer sets equals the number of 2-colorings.

Evaluating (non-ground) programs 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 answer sets. The algorithms we present in this paper deal with the latter step. While for SAT the model existence problem is NP-complete, checking whether an answer set of a given ground (disjunctive) ASP program exists is \( \Sigma^P_2 \)-complete [39]. Thus, ASP allows for compact encodings of problems of higher complexity than SAT. Such problems typically arise in artificial intelligence domains like circumscription or diagnosis.

1.1 Related Work

Gutin et al. [56] have recently stressed on the importance of implementing and evaluating algorithms that take certain structural features of an instance into account. Several works have proposed algorithms for #SAT, guaranteeing favorable theoretical runtime bounds [43] and provided prototypical implementations [68, 83]. Pichler et al. [79] have established an algorithm for deciding answer set existence of disjunction-free programs with weight constraints that runs in linear time and exploits a small incidence treewidth together together with bounded weights. Samer and Szeider [84] have presented decomposition-based algorithms for #SAT, which work on various notions of graph representations. Two of those (the primal and incidence graph) are related to the graph representations we use. Fichte and Szeider [42] have shown that tree decompositions of the dependency graph cannot be exploited to decide answer set existence more efficiently.
1.2 Contributions

The main contributions of this paper are:

1. We propose several tree decomposition-based dynamic programming algorithms to solve the answer set counting problem for the full ground ASP language [93]. The central difference between these algorithms is the underlying graph representation of the input program. Further, we practically improve these algorithms inspired by ideas from other domains [15].

2. We show that the algorithms run in linear time in the input program size when the treewidth of the respective graph representation is bounded.

3. We provide two prototypical solvers implementing our proposed algorithms. We give an experimental performance analysis and evaluation, which shows that our algorithms are highly competitive on instances of low treewidth compared to state-of-the-art counting solutions.

   (a) Our first implementation DynASP 2 works in a single pass on the tree decompositions, i.e., the decomposition is traversed from bottom to top exactly once.

   (b) Our second implementation DynASP 2.5 operates in several passes (derive partial solutions, derive certificates, link certificates and partial solutions) based on ideas from other domains [15], which allows for removing non-solution tuples at an early stage during the bottom-up traversal of the tree decomposition. In particular, a several pass algorithm employs much more sophisticated data structures.

1.3 Prior Work and Paper Organization

This paper is an extended and updated version of a paper that appeared in the informal proceedings of the Workshop on Trends and Applications of Answer Set Programming (TAASP 2016) [41]. The present paper provides a higher level of detail and presents a novel algorithm and implementation (DynASP 2.5), which operates in several passes on the tree decomposition.

The remainder of the paper is structured as follows. In Section 2 we give some preliminaries on ASP. Section 3 deals with decomposition-based #ASP solving, which first of all provides an overview of the general principles of dynamic programming algorithms on tree decompositions and then proceeds to give the proposed answer set counting algorithms and discusses its different variations. In Section 4 we note implementation issues concerning our prototype dynasp. Finally, dynasp is evaluated via experiments in Section 5. We close with some concluding remarks in Section 6.
2 Background

In this section, we provide definitions, abbreviations, and explain the underlying concepts of the DynASP system. Section 2.1 is devoted to Answer Set Programming. Section 2.2 recalls basics on computational complexity. Section 2.3 provides basic graph theoretical terminology including the problem definitions we use for.

2.1 Answer Set Programming

Answer Set Programming (ASP) is a declarative problem modelling and solving framework, which allows for the description of a problem by means of a logic program consisting of rules over propositional atoms. Answer sets are the solutions to such a logic program and represent the solutions of the encoded problem. For a full introduction to ASP we refer to other sources [22, 44, 69]. Our algorithms treat the full smodels input format [93]. Note that state-of-the-art ASP grounders output the smodels internal format. Hence, we support the full ASP-Core-2 language [23].

In the following, we provide definitions for the most important rule types. Let \( \ell, m, n \) be non-negative integers such that \( \ell \leq m \leq n \), \( a_1, \ldots, a_n \) propositional atoms, and \( w, w_1, \ldots, w_n \) non-negative integers. A choice rule is an expression of the form

\[
\{ a_1; \ldots; a_\ell \} :- a_{\ell+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n.
\]

with the intuitive meaning that some subset of \( \{ a_1, \ldots, a_\ell \} \) is true if all atoms \( a_{\ell+1}, \ldots, a_m \) are true and there is no evidence that any atoms of \( a_{m+1}, \ldots, a_n \) are true.

A disjunctive rule is an expression of the form:

\[
a_1 \lor \cdots \lor a_\ell :- a_{\ell+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n.
\]

with the intuitive meaning that at least one atom of \( a_1, \ldots, a_\ell \) must be true if all atoms \( a_{\ell+1}, \ldots, a_m \) are true and there is no evidence that any atoms of \( a_{m+1}, \ldots, a_n \) are true.

A weight rule is an expression of the form

\[
a_1 :- w \leq \{ a_2 = w_2, \ldots, a_m = w_m, \neg a_{m+1} = w_{m+1}, \ldots, \neg a_n = w_n \}.
\]

with the intuitive meaning that \( a_1 \) must be true if the sum of all weights assigned to literals that are satisfied is at least \( w \).

Finally, an optimization rule (weak constraint) is an expression of the form

\[
\sim a_1, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n [w].
\]

with the intuitive meaning that setting atoms \( a_1 \) to \( a_m \) to true counts as penalty of weight \( w \) if there is no evidence that any atoms of \( a_{m+1}, \ldots, a_n \) are true and optimization rules are subject to minimization.

A rule is either a disjunctive, or a choice, a weight, or an optimization rule. For a choice or disjunctive rule \( r \), we write \( \{ a_1, \ldots, a_\ell \} = H(r) \) (the head of \( r \)), \( \{ a_{\ell+1}, \ldots, a_m \} = B^+(r) \).
(the positive body of r), and \( \{a_{m+1}, \ldots, a_n\} = B^-(r) \) (the negative body of r), and at(r) := H(r) ∪ B^+(r) ∪ B^-(r). For sake of presentation of our algorithms let \( \text{wght}(r, a) = 1 \) for an atom \( a \in B^+(r) \cup B^-(r) \), \( \text{wght}(r, A) = |A| \) for a set \( A \subseteq B^+(r) \cup B^-(r) \), and \( \text{bnd}(r) = |B^+(r) \cup B^-(r)| \). For a weight rule \( r \), we write \( \{a_1\} = H(r), \{a_2, \ldots, a_m\} = B^+(r), \{a_{m+1}, \ldots, a_n\} = B^-(r), \) \( \text{wght}(r, a) \) maps an atom \( a \) in \( B^+(r) \cup B^-(r) \) to its corresponding weight and 0 otherwise, \( \text{wght}(r, A) = \sum_{a \in A} \text{wght}(r, a) \) for a set \( A \subseteq B^+(r) \cup B^-(r) \), and \( \text{bnd}(r) = w \). For an optimization rule \( r \), we write \( \{a_1, \ldots, a_m\} = B^+(r), \{a_{m+1}, \ldots, a_n\} = B^-(r), \) and \( \text{wght}(r) = w \). We sometimes use the representation \( H(r) \leftarrow B^+(r), B^-(r) \) and write \( \top \) instead of \( \emptyset \) or even omit it. A rule \( r \) is called negation-free if \( B^-(r) = \emptyset \) and \( r \) is normal if \( |H(r)| \leq 1 \). A ground answer set program (or, simply, program) is a set \( \Pi \) of rules. Terminology about rules naturally extends to programs. Further, for a program \( \Pi \) we let \( \text{CH}(\Pi) \subseteq \Pi \) consist of all choice rules of \( \Pi \), \( \text{DISJ}(\Pi) \subseteq \Pi \) consist of all disjunctive rules of \( \Pi \), and \( \text{WGT}(\Pi) \subseteq \Pi \) consist of all weight rules of \( \Pi \). We denote the set of atoms occurring in \( \Pi \) by at(\( \Pi \)).

We follow standard definitions of answer sets [46]. Let \( M \) be a set \( M \subseteq \text{at}(\Pi) \). We call \( M \) a model of \( \Pi \), if (i) \( (H(r) \cup B^-(r)) \cap M \neq \emptyset \) or \( B^+(r) \setminus M \neq \emptyset \) for disjunctive rule \( r \in \Pi \) and (ii) \( H(r) \cap M \neq \emptyset \) or \( \sum_{a_i \in M \cap B^+(r)} w_i + \sum_{a_i \in B^-(r) \setminus M} w_i < w \) for every weight rule \( r \in \Pi \). The reduct \( r^M \) of (i) a choice rule \( r \) is the choice rule \( H(r) \cap M \leftarrow B^+(r) \), (ii) a disjunctive rule \( r \) is the disjunctive rule \( H(r) \leftarrow B^+(r) \), (iii) a weight rule \( r \) is the rule \( a_i \leftarrow w_i \leq [a = W(a) : a \in B^+(r)] \), where \( w'_i = w - \sum_{a_i \in B^+(r) \setminus M} w_i \), and (iv) an optimization rule \( r \) is the optimization rule \( : \sim B^+(r) \cap M[wght(r)] \). The reduct \( \Pi^M \) is the program that consists of (i) reduces \( r^M \) where \( B^-(r) \cap M = \emptyset \) and \( H(r) \cap M \neq \emptyset \) for all choice rules \( r \in \Pi \), (ii) reduces \( r^M \) where \( B^-(r) \cap M = \emptyset \) for all disjunctive rules \( r \in \Pi \), (iii) reduces \( r^M \) for all weight rules \( r \in \Pi \) if \( w'_i \leq \sum_{a_i \in B^+(r)} w_i \), and (iv) reduces \( r^M \) where \( B^+(r) \subseteq M \) and \( B^-(r) \cap M = \emptyset \) for all optimization rules \( r \in \Pi \). If \( M \) is a model of \( r \), we also say \( M \) satisfies \( r \). The GL reduct (Gelfond-Lifschitz reduct) \( \Pi^M \) of program \( \Pi \) consists of each rule \( r^M \) for \( r \in \Pi \) where \( B^-(r) \neq \emptyset \) for every choice or disjunctive rule \( r \in \Pi \) and \( r^M \) for every weight rule \( r \in \Pi \). We \( M \) a model of \( \Pi^M \) if \( M \) satisfies every disjunctive rule \( r \in \Pi^M \) and weight rule and if \( H(r) \subseteq M \) or \( B^-(r) \setminus M \neq \emptyset \) for every choice rule \( r \in \Pi^M \). We call \( a(\Pi, M) := \sum_{r \in \text{OPT}(\Pi^M)} \text{wght}(r) \) the objective of \( \Pi \) and \( M \).

A set \( M \subseteq \text{at}(\Pi) \) is an answer set of a program \( \Pi \) if \( M \) is a subset minimal model of the reduct \( \Pi^M \). A set \( M \subseteq \text{at}(\Pi) \) is an optimal answer set if \( M \) is an answer set of \( \Pi \) and the objective of \( \Pi \) and \( M \) is minimal over all answer sets of \( \Pi \).

**Example 2.1.** Consider the following program \( \Pi \):

\[
a : -2 \leq 2\{b = 2, c = 1\}.\]
\[
b \vee d : -\top.
\]

This program has two answer sets: \( \{a, b\} \) and \( \{d\} \).

**ASP Problems**

We consider the following fundamental ASP problems.


<table>
<thead>
<tr>
<th>Problem:</th>
<th>CHECK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>A program $\Pi$ and a set $M \subseteq \text{at}(\Pi)$.</td>
</tr>
<tr>
<td>Task:</td>
<td>Decide whether $M$ is an (optimal) answer set of $\Pi$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem:</th>
<th>CONSISTENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>A program $\Pi$.</td>
</tr>
<tr>
<td>Task:</td>
<td>Decide whether $\Pi$ has an (optimal) answer set.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem:</th>
<th>BRAVE REASONING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>A program $\Pi$ and an atom $a \in \text{at}(\Pi)$.</td>
</tr>
<tr>
<td>Task:</td>
<td>Decide whether $a$ belongs to some (optimal) answer set of $\Pi$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem:</th>
<th>COMPUTEAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>A program $\Pi$.</td>
</tr>
<tr>
<td>Task:</td>
<td>Output an (optimal) answer set of $\Pi$ if one exists, otherwise output No.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem:</th>
<th>#ASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>A program $\Pi$.</td>
</tr>
<tr>
<td>Task:</td>
<td>Output the number of (optimal) answer sets of $\Pi$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem:</th>
<th>ENUMASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input:</td>
<td>A program $\Pi$.</td>
</tr>
<tr>
<td>Task:</td>
<td>List all (optimal) answer sets of $\Pi$.</td>
</tr>
</tbody>
</table>

### 2.2 Computational Complexity

We assume that the reader is familiar with the main concepts of computational complexity theory, especially, algorithms, decision, counting, and search problems, and complexity classes. An overview over this topic can be found in standard works \cite{8, 59, 78}.

We use the asymptotic notation $O(\cdot)$ in the standard way. Let $\Sigma$ and $\Sigma'$ be some finite alphabets. We call $I \in \Sigma^*$ an instance and $\|I\|$ denotes the size of $I$. Let $L \subseteq \Sigma^*$ and $L' \subseteq \Sigma'^*$ be decision problems. We sometimes call an instance $I \in L$ a yes-instance and an instance $I \not\in L$ a no-instance. Usually, we identify $L$ with the set of all yes-instances $I$. A (non-deterministic) polynomial-time Turing reduction from $L$ to $L'$ is an (non-deterministic) algorithm that decides in time $O(\|I\|^c)$ for some constant $c$ whether $I \in L$ using $L'$ as an oracle. A polynomial-time (many-to-one) reduction from $L$ to $L'$ is a function $r : \Sigma^* \to \Sigma'^*$ such that for all $I \in \Sigma^*$ we have $I \in L$ if and only if $r(I) \in L'$ and $r$ is computable in time $O(\|I\|^c)$ for some constant $c$. In other words, a polynomial-time reduction transforms instances of decision problem $L$ into instances of decision problem $L'$ in polynomial time.

A decision problem $L$ is (non-deterministically) polynomial-time solvable if there exists a constant $c$ such that we can decide by an (non-deterministic) algorithm whether $I \in L$ in time $O(\|I\|^c)$. 
P is the class of all polynomial-time solvable decision problems. NP is the class of all non-deterministically polynomial-time solvable decision problems. Let C be a decision complexity class, e.g., NP. Then co-C denotes the class of all decision problems whose complement (the same problem with yes and no answers swapped) is in C. We are also interested in the Polynomial Hierarchy \([78, 91, 92, 98]\) up to the second level. The Polynomial Hierarchy consists of complexity classes \(\Sigma_i^p\) for \(i \geq 0\) based on the following definitions: \(\Sigma_0^p := P\) and \(\Sigma_{i+1}^p = \text{NP}^{\Sigma_i^p}\) for all \(i \geq 0\) where NP\(^C\) denotes the class of all decision problems such that there is a polynomial-time Turing reduction to any decision problem \(L \in \text{C}\), i.e., a decision problem \(L' \in \text{NP}^C\) is non-deterministically polynomial-time solvable using any problem \(L \in \text{C}\) as an oracle. Moreover, \(\Pi_k^p := \text{co-}\Sigma_k^p\). Note that \(\text{NP} = \Sigma_1^p\), co-NP = \(\Pi_1^p\), \(\Sigma_2^p = \text{NP}^{\text{NP}}\), and \(\Pi_2^p = \text{co-NP}^{\text{NP}}\).

A witness function is a function \(\mathcal{W} : \Sigma^* \to 2^{\Sigma^*}\) that maps an instance \(I \in \Sigma^*\) to a finite subset of \(\Sigma^*\). We call the set \(\mathcal{W}(I)\) the witnesses. Let \(L : \Sigma^* \to \mathbb{N}_0\) be a counting problem, more precisely, a function that maps a given instance \(I \in \Sigma^*\) to the cardinality of its witnesses \(|\mathcal{W}(I)|\). Let C be a decision complexity class. Then, \(\# \cdot C\) denotes the class of all counting problems whose witness function \(\mathcal{W}\) satisfies (i) there is a function \(f : \mathbb{N}_0 \to \mathbb{N}_0\) such that for every instance \(I \in \Sigma^*\) and every \(W \in \mathcal{W}(I)\) we have \(|W| \leq f(|I|)|\) and \(f\) is computable in time \(O(|I|^c)\) for some constant \(c\) and (ii) for every instance \(I \in \Sigma^*\) the decision problem \(\mathcal{W}(I)\) belongs to the complexity class \(C\). Then, \(\# \cdot \text{P}\) is the complexity class consisting of all counting problems associated with decision problems in NP. Let \(L\) and \(L'\) be counting problems with witness functions \(\mathcal{W}\) and \(\mathcal{W}'\). A parsimonious reduction from \(L\) to \(L'\) is a polynomial-time reduction \(r : \Sigma^* \to \Sigma^*\) such that for all \(I \in \Sigma^C\), we have \(|\mathcal{W}(I)| = |\mathcal{W}'(r(I))|\). It is easy to see that the counting complexity classes \(\# \cdot C\) defined above are closed under parsimonious reductions. It is clear for counting problems \(L\) and \(L'\) that if \(L \in \# \cdot C\) and there is a parsimonious reduction’s \(L'\) to \(L\), then \(L' \in \# \cdot C\).

We say that a problem \(L\) is C-hard if there is a polynomial-time reduction or parsimonious reduction, respectively, for every problem \(L' \in \text{C}\) to \(L\). If in addition \(L \in \text{C}\), then \(L\) is C-complete. For instance, a decision problem is NP-complete if it belongs to NP and all decision problems in NP have polynomial-time reductions to it.

### Complexity of the main ASP problems

The computational complexity of various decision problems arising in answer set programming has been the subject of extensive studies. The decision problem CONSISTENCY is \(\Sigma_2^p\)-complete \([40]\), and remains \(\Sigma_2^p\)-complete when the input is restricted to disjunctive programs \([39]\) or weight programs where negative weights are allowed \([40]\). However, the complexity of the problem CONSISTENCY drops to NP-complete when the input is restricted to normal programs \([13, 72]\), or choice \([90]\), or weight programs with non-negative weights \([90]\). The problem BRAVE REASONING is \(\Delta_2^p\)-complete \([38]\), BRAVE REASONING is \(\Sigma_2^p\)-complete when the input is restricted to disjunctive programs without optimization rules \([39]\), or weight programs where negative weights but no optimization rules are allowed \([40]\). BRAVE REASONING is \(\Delta_2^p\)-complete when the input is restricted to normal or choice programs with optimization rules \([38]\). CHECK of ASP is \(\Pi_2^p\)-complete in general, is co-NP-complete when the input is restricted to disjunctive programs \([39]\), or weight programs where negative weights are allowed \([40]\), or normal programs with optimization rules \([38]\). The complexity of the problem CONSISTENCY drops to P when the input is restricted to normal programs \([72]\), choice \([90]\), or weight programs with non-negative...
weights \cite{90}. \#ASP of normal ASP is easily seen to be \#P-hard. Several fragments of programs where ASP problems are non-deterministically polynomial-time solvable or even polynomial-time solvable have been identified.

## 2.3 Graphs

We recall some graph theoretical notations. An undirected graph or simply a graph is a pair $G = (V, E)$ where $V \neq \emptyset$ is a set of vertices and $E \subseteq \{ \{u, v\} : u \neq v \}$ is a set of edges. We denote an edge $\{u, v\}$ by $uv$ or $vu$. For a vertex $v \in V$, we call any $u \in V$ such that there is an edge $uv \in E$ a neighbor of $v$. A graph $G' = (V', E')$ is a subgraph of $G$ if $V' \subseteq V$ and $E' \subseteq E$ and an induced subgraph if additionally for any $u, v \in V'$ and $uv \in E$ also $uv \in E'$. A path of length $k$ is a graph with $k + 1$ pairwise distinct vertices $v_1, \ldots, v_{k+1}$, and $k$ distinct edges $v_iv_{i+1}$ where $1 \leq i \leq k$ (possibly $k = 0$). A cycle of length $k$ is a graph that consists of $k$ distinct vertices $v_1, v_2, \ldots, v_k$ and $k$ distinct edges $v_1v_2, v_2v_3, \ldots, v_{k-1}v_k, v_kv_1$. A chord of cycle $c$ of length $l$ in $G$ is an edge $v_iv_j \in E$ where $v_i$ and $v_j$ are not connected by an edge in $c$ (non-consecutive vertices). $G$ is chordal (triangulated) if every cycle in $G$ of length at least 4 has a chord. Let $G = (V, E)$ be a graph. $G$ is bipartite if the set $V$ of vertices can be divided into two disjoint sets $U$ and $W$ such that there is no edge $uv \in E$ with $u \in U$ or $v \in W$. $G$ is a $(k \times \ell)$-grid if $V = \{v_1, \ldots, v_{k\ell}\}$ and $E = \{v_{i,j}v_{i',j'} : \max(|i - i'|, |j - j'|) = 1\}$. $G$ is complete if for any two vertices $u, v \in V$ there is an edge $uv \in E$. $G$ contains a clique on $V' \subseteq V$ if the induced subgraph $(V', E')$ of $G$ is a complete graph. A connected component $C$ of $G$ is an inclusion-maximal subgraph $C = (V_C, E_C)$ of $G$ such that for any two vertices $u, v \in V_C$ there is a path in $C$ from $u$ to $v$. We say $G$ is a tree if it is a connected component $C = G$ and $G$ contains no cycles. We usually call the vertices of a tree nodes. A directed graph or simply a digraph is a pair $G = (V, E)$ where $V \neq \emptyset$ is a set of vertices and $E \subseteq \{(u, v) : u \in V : u \neq v\}$ is a set of directed edges. A digraph $G' = (V', E')$ is a subdigraph of $G$ if $V' \subseteq V$ and $E' \subseteq E$ and an induced subdigraph if additionally for any $u, v \in V'$ and $(u, v) \in E$ also $(u, v) \in E'$. For a vertex $v \in V$ we call a vertex $v' \in \{w : (w, u) \in E\}$ a predecessor of $v$ and a vertex $v \in \{w : (v, w) \in E\}$ a successor of $v$. A rooted tree $T = (V, E, r)$ consists of (i) a digraph $(V, E)$ whose underlying graph is a tree and (ii) a designated vertex $r$ which has no predecessor. We call a vertex $v \in V$ node of $T$, a successor of a node child, the vertex $r$ the root of $T$, and a vertex $v$ that has no child a leaf of $T$.

Let $G = (V, E)$ be a graph and $k$ a positive integer. We call a function $c : V \to \{1, \ldots, k\}$ a $k$-coloring of $G$ if $c(v) \neq c(w)$ for all $vw \in E$. Then, the counting problem $k$-COL asks to give the number of minimal $k$-colorings of a given graph. Figure 1 visualizes a graph and a 3-coloring of the graph. Dominating Set A dominating set of a graph $G = (V, E)$ is a set $S \subseteq V$ such that every vertex $v \notin S$ there is some $w \in S$ such that $vw \in E$. Then, the counting problem $k$-DS asks to give the number of minimal dominating set of a given graph. A vertex cover of a graph $G = (V, E)$ is a set $S \subseteq V$ such that for every edge $uv \in E$ we have $\{u, v\} \cap S \neq \emptyset$. Then, the counting problem $k$-VC asks to give a minimum (cardinality-minimal) vertex cover of a given graph. Then, the counting problem $k$-VC asks to give a minimal (subset-minimal) vertex cover of a given graph. Let $V$ be a set of nodes. A Steiner Tree is a tree on the nodes $V$. Then, the counting problem $ST$ asks to give a Steiner Tree $T = (V, N)$ of a given set $V$ of nodes of minimum number of nodes.
Background and Related Work

For further basic terminology on graphs and digraphs, we refer to standard texts [21] [32].
3 Decomposition-based #ASP Solving

In this section, we develop definitions and algorithms for dynamic programming on tree decompositions of various graph representations of a program. In Section 3.1, we recall basic notions on tree decompositions of graphs, how to find tree decompositions of graphs, and how in general dynamic programming algorithms exploit tree decompositions of low width. Then, in Section 3.2, we provide an overview and definitions on graph representations of answer set programs as well as required notation for our dynamic programming algorithms. In Section 3.3, we discuss principles of the architecture of our dynamic programming approaches (DynASP2/DynASP2.5) and give an overview on how the internal components interconnect. Subsequently, we present the underlying algorithms to solve the problems CONSISTENCY, BRAVE REASONING, and COMPUTEAS from a theoretical perspective. Section 3.5 provides considerations on the correctness and the runtime of the algorithms. Then, we adapt the algorithms to compute stable models of programs with optimization rules, solve the counting problem #ASP, and provide complexity result for the problem #ASP in Section 3.6. Finally, we conclude with a comparison in Section 3.7.

3.1 Tree Decompositions

Many computationally hard problems on graphs are easy if the input graph is a tree. Hence, it seems desirable to exploit a structural property of a graph that is “almost” a tree to solve a computational problem more efficiently. A structural property (parameter) that allows for measuring in a certain sense the “tree-likeness” of a graph is the treewidth. The underlying concept of tree decompositions and dynamic programming of tree decompositions provide us often with a powerful tool to exploit a small treewidth of a graph and solve problems more efficiently. Tree decompositions and treewidth have received a great deal of attention in the theoretical computer science community starting from the initial work by Robertson and Seymour [81]. For a comprehensive background and examples on treewidth we refer to a survey [9, 16, 19]. Since then, it has been widely acknowledged that treewidth represents a very useful parameter, which has been applied to a broad range of problems in knowledge representation, reasoning, and artificial intelligence [36, 52, 80]. In particular, instances from practical settings often exhibit small treewidth (see, e.g., [5, 55, 61, 67, 73, 95]).

Definition 3.1 ([81]). Let $G = (V, E)$ be graph, $T = (N, E_T, r)$ a rooted tree, and $\chi : N \rightarrow 2^V$ a function that maps to each node $t \in T$ a set of vertices. We call the sets $\chi(\cdot)$ bags. Then, the pair $T = (T, \chi)$ is a tree decomposition of $G$ if the following conditions are satisfied:

1. for every vertex $v \in V$, there exists a node $n \in N$ such that $v \in \chi(n)$ ("vertices covered")
2. for every edge $e \in E$, there exists a node $n \in N$ such that $e \subseteq \chi(n)$ ("edges covered")

and

3. for any three nodes $t_1, t_2, t_3 \in N$, if $t_2$ lies on the unique path from $t_1$ to $t_3$, then $\chi(t_1) \cap \chi(t_3) \subseteq \chi(t_2)$ ("connectedness").

We call $\max\{|\chi(t)| - 1 : t \in N\}$ the width of the decomposition. The treewidth of the graph $G$ is the minimum width over all possible tree decompositions of $G$. 
A tree decomposition of a graph is a rooted labeled tree obtained from a given graph. The idea of a tree decomposition is that each bag for each node of the tree subsumes multiple vertices of the given graph, thereby isolating the parts responsible for a graph of not being a tree. When we thus want to turn a graph into a tree, we can think of contracting vertices until we end up with a tree whose nodes represent subgraphs of the original graph. Our sought-for measure of a graph’s “tree-likeness” can thereby be determined as “how extensive” such contractions must be.

Note that each graph has a trivial decomposition \((T, \chi)\) consisting of the tree \((\{n\}, \emptyset, n)\) and the mapping \(\chi(n) = V\). It is well known that a tree has treewidth 1 and a cycle has treewidth 2. Further, if a graph \(G\) contains a clique on the set \(\{v_1, \ldots, v_k\}\), then any tree decompositions of \(G\) contains node \(n\) where \(\{v_1, \ldots, v_k\} \subseteq \chi(n)\). Therefore, the treewidth of a graph containing a \(k\)-clique is at least \(k - 1\). Besides, if the treewidth of a \((k \times k)\)-grid graph is \(k\). Hence, if a graph has a large clique or grid as a subgraph it implies large treewidth.

**Example 3.2.** Figure 2 shows a graph \(G\) together with a tree decomposition of \(G\) that is of width 2. The decomposition is optimal because the graph contains a clique on the vertices \(\{a, b, c\}\).

**Definition 3.3 (64).** A tree decomposition \((T, \chi)\) where \(T = (N, E_T, r)\) is called normalized, if the following conditions hold:

1. every node \(t \in N\) has at most two children,
2. for every node \(t \in N\), if \(t\) has two children \(t'\) and \(t''\), then \(\chi(t) = \chi(t') = \chi(t'')\), and
3. for every node \(t \in N\), if \(t\) has exactly one child \(t'\), then \(|\chi(t) \cap \chi(s)| = 1\), in other words, the bags \(\chi(t)\) and \(\chi(t')\) differ in exactly one element.

Let \(t\) be a node \(t \in N\) of a normalized tree decomposition \((T, \chi)\) where \(T = (N, E_T, r)\). We call \(t\) a leaf node if \(t\) has no children, an introduce node if \(t\) has exactly one child \(t' \in N\) and \(|\chi(t)| = |\chi(t')| + 1\), a removal node if \(t\) has exactly one child \(t'\) and \(|\chi(t)| = |\chi(t')| - 1\), a join node if \(t\) has two children.

**Proposition 3.4 (64).** For every tree decomposition, there is a normalized tree decomposition, which can be obtained in linear time without increasing the width of the tree decomposition.

In the subsequent of the paper, we assume according to Proposition 3.4 that tree decompositions are normalized and have a root. Further, for better readability of our algorithms unless otherwise stated we let bags of leaf nodes and the root be empty, which obviously does not increase the width and introduces has most linear many new nodes in the tree.
Exploiting Low Treewidth: Algorithms on Tree Decompositions

Many decision problems, which are NP-hard on arbitrary input graphs, are solvable in polynomial time when the treewidth of the input graph is bounded by a fixed (preferably low) constant. A generic result by Courcelle [26] states that every problem that is definable in a certain logic (monadic second-order logic\(^1\)) can be solved in time \(f(k) \cdot n^c\) for some linear-time function \(f\), which depends only on the treewidth of the input graph, and some fixed constant \(c\). Such problems are also known as fixed-parameter tractable problems when parameterized by treewidth [77]. A formula (in monadic second-order logic) for a given problem and input graph, can then be translated into a finite tree automaton. However, the algorithms resulting from such a translation are mostly impractical due to large constants [77]. Hence, special-tailored algorithms that work directly on tree decompositions are mostly employed in practice instead. These algorithms usually apply dynamic programming on tree decompositions and have been widely applied in graph theory [18], logic, and artificial intelligence [53, 54, 80]. For a comprehensive introduction and examples we refer to other sources [27].

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 sub-problem is solved, which consists of the part of the problem instance that is induced by the content of the bag of the 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).

In practice, it is usually not necessary work on an optimal tree decomposition in order to take advantage of low treewidth of an instance. In particular, having a non-optimal tree decomposition will typically imply higher runtime and memory consumption than necessary, but does not effect the computed solutions even when we are searching for optimal solutions.

Example 3.5. Consider the graph \(G\) from Figure 3a and the problem 3-COL (see Section 2.3 for the problem definition). Figure 3b illustrates a tree decomposition \(T\) of \(G\) of width 2. Figure 3c illustrates the tables, which are computed when running dynamic programming on the tree decomposition \(T\). Each of the tree decomposition node in Figure 3b has a corresponding table in Figure 3c and there is a column for each bag element. Additionally, we have a column \(i\) that is used to store an identifier for each row such that an entry in the column \(j\) of a potential parent table can refer to the respective row. Eventually, each row will describe a proper coloring of the subproblem represented by the bag (to the subproblem).

The tables in Figure 3c are computed in a bottom-up way. First, all 3-colorings for the leaf

\(^1\)Monadic second order logic enables us to express graph properties. It extends classical first-order logic by allowing quantification over sets of vertices and edges. In contrast to general second-order logic, we may only quantify over unary relational variables in monadic second order logic.
nodes are constructed and stored in the respective table. Therefore, we simply guess a color such that no vertex and its neighbor have the same color. For each non-leaf node with already computed child tables, we then look at all combinations of child rows and combine those rows that coincide on the colors of common bag elements, in other words, we join the rows. Since the leaves have no common bag elements, each pair of child rows is joined. However, we must eliminate all results in the table of the join node that violate a constraint, i.e., where a vertex an its neighbor have the same color associated. For instance, the combination of row 0 from the left child with row 2 from the right child cannot yield a 3-coloring because both $b$ and its neighbor $d$ are colored with “g”; the left row 0 combined with the right row 0 yields a 3-coloring and gives rise to the row 3 in the root table. We store the identifiers of these child rows as a pair in the $j$ column. Note that the entry of $j$ in row 3 not only contains $(0,0)$ but also $(0,1)$, because joining these rows produces the same row as we project onto the current bag elements $b$, $c$ and $d$. Note that in general to solve 3-COL we must not only decide which child rows to join but also extend partial solutions, which is however not necessary in our example. Storing all predecessors of a row like this allows us to enumerate all 3-colorings with a final top-down traversal.

It is easy to see that the presented algorithm runs in linear space and time when the treewidth is fixed by some constant.

### Finding Tree Decompositions

In general, determining the treewidth of a given graph and constructing an optimal tree decomposition are unfortunately intractable. More precisely, given a graph $G = (V, E)$ and a non-negative integer $k$, deciding whether $G$ has treewidth at most $k$ is NP-complete \[7\]. However, for arbitrary but fixed $k$ we can decide the problem in time $2^{O(k^3)} \cdot |V|$ \[17\] and thus solvable in linear time. In fact, the algorithm can also be used to construct a tree decomposition by means of standard
constructions (self-transformation) \cite{34, 35, 88}. Further, several exact algorithms and algorithms to compute upper bounds have been proposed \cite{11, 20, 49, 85, 89}.

Since the running times of these algorithms are far from practical on large graphs, we often use efficient heuristics \cite{31, 51} that produce a tree decomposition of reasonably low, but not necessarily optimal, width\footnote{By optimal width we mean that for a given graph \( G \) the width of the tree decomposition of \( G \) equals the treewidth of the graph \( G \).}. Various heuristics have been proposed: greedy heuristic algorithms include Maximum Cardinality Search (MCS) \cite{94}, Min-Fill heuristic \cite{30}, and Minimum Degree heuristic \cite{12}. Metaheuristic techniques have been provided in terms of genetic algorithms \cite{66, 76}, ant colony optimization \cite{57}, and local search based techniques \cite{25, 63, 75}. For a detailed description, we refer to recent surveys \cite{20, 58}.

In this paper, we use the graph decomposition library htd \cite{1, 4} to heuristically construct tree decompositions of a given graph. The library relies on the so-called bucket elimination algorithm \cite{29}, which works as follows: Let the pair \((T, \chi)\) consist of an empty tree and an empty labeling, i.e., \( T = (N, E_T, \varepsilon) \) with \( N = E_T = \emptyset \). For a given graph \( G = (V, E) \) (i) select a vertex \( v \in V \) according some heuristic, (ii) construct a clique on all neighbors of \( v \) in \( G \), remove \( v \) from \( G \) and the ordering \( o \), (iii) add a fresh node \( n \) to \( N \) and let \( \chi(n) := \{ u : u \text{ is a neighbor of } v \} \), (iv) add an edge \( e \) to each node in \( T \) in whose bag \( v \) appears as well, (v) proceed with Step (i) unless \( V = \emptyset \). To select a vertex the library htd supports two heuristics. The heuristic Min-fill selects the vertex whose “elimination” adds the smallest number of edges to \( E \). The heuristic Maximum Cardinality Search selects the first vertex randomly. Then, the vertex with the highest number of neighbors, which occur in bags of \( T \), in the input graph is selected. Ties are broken uniformly at random in both heuristics.

3.2 Tree Decompositions of ASP Programs

In order to exploit small treewidth and decompositions of small with in the domain of answer set programming, we first need to a suitable concept of graph representation of a program. The following definition extends concepts of Samer and Szeider \cite{84} from the propositional setting to answer set programming and parts have already been introduced by Jakl, Pittler, and Woltran \cite{62}.

**Definition 3.6.** Let \( \Pi \) be a program. The *primal graph* \( P(\Pi) \) of \( \Pi \) is the graph \( G = (V, E) \) where the set \( V \) consists of a vertex \( v_a \) for every atom \( a \in \text{at}(\Pi) \) and the set \( E \) contains an edge \( v_a v_b \) if there exists an a rule \( r \in \Pi \) and \( a, b \in \text{at}(r) \). The *semi-incidence graph* \( S(\Pi) \) is the graph where the set \( V \) consists of a vertex \( v_a \) for every atom \( a \in \text{at}(\Pi) \) and a vertex \( v_r \) for every rule \( r \in \Pi \) and the set \( E \) contains an edge \( v_r v_a \) if an atom \( a \) appears in rule \( r \) of \( \Pi \), an edge \( v_a v_b \) if atoms \( a, b \) appear together in a choice head or weighted body of a rule in \( \Pi \). The *incidence graph* \( I(\Pi) \) of \( \Pi \) is the bipartite graph \( G = (V, E) \) where the set \( V \) consists of a vertex \( v_a \) for every atom \( a \in \text{at}(\Pi) \) and a vertex \( v_r \) for every rule \( r \in \Pi \) and the set \( E \) contains an edge \( v_a v_r \) if \( a \in \text{at}(r) \) for some rule \( r \in \Pi \). For an atom \( a \) an a vertex \( v_a \), we call the vertex \( v_a \) a corresponding vertex of atom \( a \), and vice versa, and for a rule and a vertex \( v_r \) we call the vertex \( v_a \) a corresponding vertex of rule \( r \), and vice versa.
We call a tree decomposition of the incidence or primal graph, respectively of a program \( \Pi \) a tree decomposition of \( \Pi \). For \( T = (T, \chi) \) a tree decomposition of program \( \Pi \) and a node \( t \in T \), we let \( A_t \) contain all atoms \( \text{at}(\Pi) \) that have a corresponding vertex in \( \chi(t) \), \( R_t \) contain all rules \( r \in \Pi \) that have a corresponding vertex in \( \chi(t) \), and \( \hat{R}_t \) contain all rules \( r \in \Pi \) such that \( \text{at}(r) \subseteq A_t \).

Let \( \Pi \) be a program, \( (T, \chi) \) be a tree decomposition of the primal graph of \( \Pi \) where \( T = (N, E_T, r) \), and \( t \in N \) be a node. Then, the program \( \Pi_t \) consists of all rules \( r \in \Pi \) such that \( r \in \hat{R}_{t'} \) for some \( t' \in N' \) of the induced subtree \( T' = (N', E') \) of \( T \) rooted at \( t \). We call the sub-program \( \Pi_t \) the primal program induced by \( t \). It is easy to see that \( \Pi_r = \Pi \) for the empty root \( r \).

Let \( T \) be a normalized tree decomposition of \( \Pi \). We extend the notion from Definition 3.3 by atom and rule specific notations. More precisely, an atom introduce or atom removal node is an introduce ore removal node, respectively, where the bag of node and its child differ in a corresponding atom \( a \in \text{at}(\Pi) \). A rule introduce or rule removal node is an introduce or removal node, respectively, where the bag of a node and its child differ in a corresponding rule \( r \in \Pi \). Note that a tree decompositions of a primal graph cannot contain rule introduce or rule removal nodes.

We denote the set of all atom introduce nodes of \( T \) by \( \text{AI}(T) \), of all atom removal nodes of \( T \) by \( \text{AR}(T) \), of all join nodes of \( T \) by \( \text{JOIN}(T) \), of all leaf nodes of \( T \) by \( \text{LEAF}(T) \), of all rule introduce nodes of \( T \) by \( \text{AI}(T) \) of all rule removal nodes of \( T \) by \( \text{AR}(T) \).

Note that in the context of dynamic programming on tree decompositions the term table is commonly used for sets of atoms or a set of tuples, where a tuple \( \langle \cdot \rangle \) consists of sets of atoms (“model candidates”) together with sets of sets of atoms (“counter candidate”). For a node \( t \in N \) of a tree decomposition \( T \) and a mapping \( \sigma \) that maps nodes of \( T \) to tables, we write corresponding table for the table \( \sigma(t) \) that is assigned to the node \( t \).

**Observation 3.7.** Let \( \Pi \) be a program. The treewidth of an incidence graph is at most the treewidth of its primal graph plus 1.

### 3.3 Dynamic Programming Based Answer Set Solving

In this subsection, we explain the underlying approach to solve an ASP problem by means of dynamic programming on tree decompositions from a conceptual perspective. First, we quickly recall the general architectural principles. A classical dynamic programming approach, which we call the DynASP 2 approach, typically encompasses the steps as illustrated in Figure 4.

1. Construct a graph representation of the given input program.

2. Compute a tree decomposition of the problem instance by means of a heuristic, thereby decomposing the instance into several smaller parts and fixing an ordering in which the program will be evaluated. For every node in the tree decomposition construct an empty table.

---

3Since the primal graph contains a clique on all corresponding vertices to 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 \( r \). Hence, \( \hat{R}_t \) is well-defined.
3. For every node in the tree decomposition (in a bottom-up traversal) compute every model \( M \) of the sub-program corresponding to the node, compute all proper subsets \( C \) of \( M \), and store the tuple \( \langle M, C \rangle \) in the corresponding table.

4. Every tuple \( \langle M, \emptyset \rangle \) in the corresponding table of the root can be extended to an answer set (by means of a top-down traversal). Depending on the original problem print the solution.\(^4\)

Among these tasks, the one that is entirely problem-specific and graph representation specific is the solving the sub-problems in Step 3. When faced with a particular problem, algorithm designers typically focus on this step.

An advanced dynamic programming approach, which we call the DynASP 2.5 approach, follows ideas by Bliem et al. [14] and typically encompasses the steps as illustrated in Figure 5. The approach works similar as the DynASP 2 approach, however, Step 3 is split into three passes as follows:

I. For every node in the tree decomposition (in bottom-up traversal) compute every model \( M \) of the sub-program corresponding to the node and store \( M \) in the corresponding table (“Pass [1]”). However, do not check whether the model is also a minimal model of the reduct of the sub-program corresponding to the node. Then, for every node in the tree decomposition (in a top-down traversal) remove models, which do not extend to a model in the table for the parent node, from the corresponding table (“Purge non-models”).

II. For every node in the tree decomposition construct a second table, which will contain “counter candidates” for a model not being subset-minimal. For every node in the tree decomposition (in a bottom-up traversal) compute all subsets \( C \) of the models obtained in Pass I from the corresponding table (“Pass [II]”) and store \( C \) in the second table. Then, for every node in the tree decomposition (in a top-down traversal) remove subsets, which do not extend to a subset in the table for the parent node, from the corresponding second table (“Purge non-certs”).

---

\(^4\)Note that, depending on the problem, printing all solutions may not be required. Often we just want, e.g., to decide whether a solution exists, to count the number of solutions, or to find the optimum. Then the tables additionally contain dedicated counters. Our algorithms especially target these cases.
III. For every node in the tree decomposition (in a bottom-up traversal) construct a mapping of elements in the first table to a set of elements of the second table, i.e., models to counter-candidates. Then, the resulting mapping can be used to construct the solutions in Step 4.

All passes are carried out in accordance to an adapted version of the theory defined in earlier work by Bliem et al. [14]. In fact, results of the advanced approach are equivalent. However, practical advantages are (i) we can reuse certain counter candidates for various models and every counter candidate is only computed once among all models and (ii) using three passes allows for purging models and counter candidates early.

### 3.4 Solving CONSISTENCY, BRAVE REASONING, and COMPUTEAS

In the following subsection, we present three dynamic programming algorithms to solve the problems CONSISTENCY, BRAVE REASONING, and COMPUTEAS. We have already described the general approach to solve such problems by means of dynamic programming on tree decompositions in the previous section. Here we focus only on Step 3 of the DynASP 2 approach. The first two algorithms, PRIM and INVPRIM, work on a tree decomposition of the primal graph. Algorithm INC uses the incidence graph, but in addition atoms appearing in weight or choice rules form a clique, whereas INC uses only the incidence graph. Presenting the algorithms in this order allows us to
Input Node $t$ and tables $\tau_{t'}$ and $\tau_{t''}$ of children $t'$ and $t''$, if exist, of $t$, and $a \in A_t \setminus A_{t'}$ and $a \in A_{t'} \setminus A_t$, respectively.

Output Tuple set $\tau_t$ of node $t$

$t \in \text{LEAF}(T) \quad \{ \langle \emptyset, \emptyset \rangle \} $

$t \in \text{AI}(T) \quad \{ \langle M, \{ A \mid A \in C, \forall r \in \hat{R}_t : A \models r^M \} \rangle \quad \{ \langle M, C \rangle \in \tau_{t'}, \forall r \in \hat{R}_t : M \models r \} \cup $

\[
\{ \langle M, \{ A \mid A \in C, \forall r \in \hat{R}_t : A \models r^M \} \cup
\{ A \cup \{ a \} \mid A \in C, \forall r \in \hat{R}_t : A \cup \{ a \} \models r^M \} \cup
\{ M'' \mid M'' = M', \forall r \in \hat{R}_t : M'' \models r^M \} \rangle \quad \{ \langle M', C \rangle \in \tau_{t'}, M = M' \cup \{ a \}, \forall r \in \hat{R}_t : M \models r \}
\]

$t \in \text{AR}(T) \quad \{ \langle M \setminus \{ a \}, \{ A \setminus \{ a \} \mid A \in C \} \rangle \quad \{ \langle M, C \rangle \in \tau_{t'} \} $

$t \in \text{JOIN}(T) \quad \{ \langle M, (C' \cap C'') \cup (\{ M \} \cap C'') \cup (C' \cap \{ M \}) \rangle \quad \{ \langle M, C' \rangle \in \tau_{t'}, \langle M, C'' \rangle \in \tau_{t''} \} $

Figure 6 The algorithm PRIM.

incrementally explain the added difficulties and the technical necessities implied by solving ASP with dynamic programming on tree decompositions on these three graph types. We start with the description of the algorithms for the problem CONSISTENCY.

Given a node $t$, we denote its children by $t'$ and $t''$ (if it exists). A full specification of the dynamic programming algorithm can thus be given by describing how the table $\tau_t$ is derived from the tuples of the tables $\tau_{t'}$ and $\tau_{t''}$.

3.4.1 Algorithms PRIM and INVPRIM

The first algorithm (PRIM) to solve CONSISTENCY is based on the primal graph. We construct the table $\tau_t$ separately for each node depending on the type of the node. Tuples for the PRIM algorithm are of the form $\langle M, C \rangle$, where $M \subseteq A_t$ represents a model of the sub-program induced by $t$ and $C \subseteq 2^M$ forming models of the reduct $(\Pi_t)^M$ and therefore representing counter candidates to $M$. The existence of such a tuple $\langle M, \emptyset \rangle$ in table $\tau_t$ witnesses the existence of an answer set for the program induced by the subtree rooted at $t$. Then, a model of the input program $\Pi$ exists if the root contains a tuple. An answer set of $\Pi$ exists if the root contains the tuple $\langle \emptyset, \emptyset \rangle$, in other words, in addition to some model $M$ of $\Pi$ there is no counter model $N \subset M$ of $\Pi^M$.

Figure 6 provides the description of how the tables $\tau_t$ are computed by our PRIM algorithm. Therefore, let $T$ be a tree decomposition of the primal graph of the input program. Below we provide only descriptions for types of nodes where the result is non-trivial. For an atom introduce node $t \in \text{AI}(T)$ each local answer set $M$ of the child node $t'$ and the “introduced” atom $a \in A_t \setminus A_{t'}$

\footnote{For a definition of the node types see paragraphs below Definition 3.6}
Figure 7  Example graph and tree decomposition for the problem VC.

is extended either by taking \( M \) or \( M \cup \{a\} \). We verify whether \( M \) or \( M \cup \{a\} \) is a model of the program \( \hat{R}_t \). Recall that by Definition 2 a tree decomposition must contain each edge of the original graph in some node bag and by Definition 3.6 for a tree decomposition of a primal graph there is a bag that contains all corresponding vertices of atoms in a rule. Thus, our algorithm can immediately decide for a node \( t \) of the tree decomposition whether a rule is satisfied by taking the rules in the set \( \hat{R}_t \) into account. Intuitively, \( \hat{R}_t \) consists of each rule where vertices in a bag correspond entirely to \( t \). Since we can verify (without additional information) whether \( M \) or \( M \cup \{a\} \) is a model of \( \hat{R}_t \), we can immediately discard all tuples where \( M \) or \( M \cup \{a\} \), respectively, is not a model of \( \hat{R}_t \). All possible subsets of \( M \) or \( M \cup \{a\} \), respectively, are considered as counter candidates. A counter candidate of a model can be obtained directly from extending a counter candidate \( C \) (corresponding to the model) in the table of the child node by taking \( C \) or \( C \cup \{a\} \), respectively. For a join node \( t \in \text{JOIN}(T) \), we enforce that left and right child tuples agree on model candidates and the counter candidates.

A slightly modified implementation of the PRIM is the INVPRIM algorithm, where we store in each tuple the set \( \overline{C} \) of “inverse” counter models, instead of the set \( C \) of counter models, in order to see whether this leads to a speed-up. More precisely, for a set \( M \), a program \( \Pi \), and a tree decomposition \( T \), the set \( \overline{C} \) consists of all sets \( C \) such that \( C \) is a model of \( (\hat{R}_t)^M \). However, the set \( \overline{C} \) consists of all sets \( \overline{C} \) such that \( \overline{C} \) is not a model of \( \hat{R}_t \). Hence, we have \( \overline{C} = 2^M \setminus C \). Since the adaption of the conditions and results in Figure 6 to the set \( \overline{C} \) of inverse

\footnote{For a definition of \( \hat{R}_t \) see paragraphs below Definition 3.6}
counter models is straightforward, we omit full definitions here.

**Example 3.8.** Consider the problem VC and the graph \( G = (V, E) \) from Figure 7a. The program \( \{ in_x \lor in_y : xy \in E \} \) encodes the problem VC in ASP and is given in Figure 7b. Further, Figure 7c illustrates the primal graph \( P(\Pi) \) and Figure 7d visualizes a decomposition of the primal graph \( P(\Pi) \). In Figure 8 we present the result of the PRIM algorithm on the decomposition of Figure 7d using DynASP 2. Whenever in a row a certain atom \( in_i \) is set to true, we put “T” in the respective column labelled by \( i \) and satisfied rules are marked by “sat”. Figure 9 represents running the same instance with the improved algorithm of PRIM using DynASP 2.5. Comparing these two figures leads to the observation that the improved algorithm DynASP 2.5 quite compactly represents the resulting computation tree and moreover, does not recompute counter candidates for different local answer sets.
Figure 8  Dynamic programming tables for the nodes of decomposition $T$ of Figure 7d using PRIM with the DynASP 2 approach.
Figure 9  Dynamic programming tables for the nodes of decomposition $\mathcal{T}$ of Figure 7d using PRIM with the DynASP 2.5 approach.

3.4.2 Algorithms SINC and INVSINC

Our SINC algorithm is based on the semi-incidence graph of a given program. Since the width of the semi-incidence graph is smaller than the width of the primal graph, we expect the SINC algorithm to be faster than the PRIM algorithm. However, the SINC algorithm requires a considerable overhead in storage and computations as we will see below. Hence, if the used tree decompositions for PRIM and SINC are of similar width, we expect PRIM to be faster in practice. Note that the decision version of the SINC algorithm (restricted to disjunctive programs) restates an algorithm by Jakl, Pichler, and Woltran [62].

The intuition for tables and tuples is similar to the PRIM algorithm. However, for a node $t$ of the tree decomposition we do not necessarily know which corresponding rules to vertices in the bag are already satisfied (in contrast to the primal graph there is no property that ensures that all atoms
that occur together in a rule have a bag that contains all corresponding vertices, we can ensure this only for choice and weight rules). Hence, we additionally need to keep track of already satisfied rules by model candidates and counter candidates, respectively. Therefore, we need a “local view” on the program depending on the node \( t \) and we define the following notation.

**Definition 3.9.** Let \( \Pi \) be a program, \( \mathcal{T} \) be a tree decomposition of the semi-incidence graph of \( \Pi \), where \( \mathcal{T} = (T, \chi) \) and \( T = (N, E, r) \), and \( S \subseteq \Pi \) a set of rules. The local program in node \( t \) under the set \( S \) is the program \( \Pi^{(t, S)} \) obtained from \( \Pi \) by

1. keeping all rules in \( \left( R_t \setminus S \right) \) and removing all other rules;

2. removing from the heads and bodies of the remaining choice and disjunctive rules all literals \( a, \neg a \) with \( a \in \text{at}(\Pi) \setminus A_t \); and

3. removing from the heads and bodies of the remaining weight rules all literals \( a, \neg a \) with \( a \in \text{at}(\Pi) \setminus A_t \) and set the bound \( w - \text{wght}(r, B^+(r) \cap (\text{at}(\Pi) \setminus A_t) \cup B^-(r) \cap (\text{at}(\Pi) \setminus A_t)) \).

Then, a tuple in table \( \tau_t \) for node \( t \) of the tree decomposition is a triple \( (M, S, C) \) where \( M \) again represents a model, the additional set \( S \) consists of all rules that correspond to vertices in \( \chi(t) \) and are already satisfied in the local program, and \( C \) consists of pairs \( (C, R) \) of sets of atoms and rules, where \( C \) represents a counter model and \( R \) consists of all rules that correspond to vertices in \( \chi(t) \).
Input Node $t$ and tables $\tau_{t'}$ and $\tau_{t''}$ of children $t'$ and $t''$, if exist, of $t$, and $a \in A_t \setminus A_{t'}$ or $a \in A_t \setminus A_{t''}$ as well as if $r \in R_t \setminus R_{t'}$ or $r \in R_t \setminus R_{t''}$, respectively.

Output Tuple set $\tau_t$ of node $t$

Let $M$ be a set and $e$ be an element. We write $M^+_e := M \cup \{e\}$ and $M^-_e := M \setminus \{e\}$.

$t \in \text{LEAF}(\mathcal{T}) \left\{\langle \emptyset, \emptyset, \emptyset \rangle\right\}$

$t \in \text{AI}(\mathcal{T}) \left\{\langle M, S \cup \text{SatRules}_t(\Pi, S, M), \right.$

$\left\{\langle C, R \cup \text{SatRules}_t(\Pi^M, R, C) \rangle \mid \langle C, R \rangle \in C\rangle \cup \right.$

$\left\{\langle M^+_a, S \cup \text{SatRules}_t(\Pi, S, M^+_a)\rangle, \right.$

$\left\{\langle C, R \cup \text{SatRules}_t(\Pi^M^+, R, C) \rangle \mid \langle C, R \rangle \in C\rangle \cup \right.$

$\left\{\langle C^+_a, R \cup \text{SatRules}_t(\Pi^M^+, R, C^+_a)\rangle \mid \langle C, R \rangle \in C\rangle \cup \right.$

$\left\{\langle M, \text{SatRules}_t(\Pi^M^+, S, M)\rangle\rangle \mid \langle M, S, C \rangle \in \tau_t\right\}$

$t \in \text{RI}(\mathcal{T}) \left\{\langle M, S \cup \text{SatRules}(\{r\}, S, M), \right.$

$\left\{\langle C, R \cup \text{SatRules}(\{r\}^M, R, C) \rangle \mid \langle C, R \rangle \in C\rangle \mid \langle M, S, C \rangle \in \tau_t\right\}$

$t \in \text{AR}(\mathcal{T}) \left\{\langle M^+_a, S, \{C^+_a, R\} \mid \langle C, R \rangle \in C\rangle \left\mid \langle M, S, C \rangle \in \tau_t\right\}$

$t \in \text{RR}(\mathcal{T}) \left\{\langle M, S^-_r, \{C, R^-_r\} \mid \langle C, R \rangle \in C, r \in R\rangle \right\mid \langle M, S, C \rangle \in \tau_t, r \in S\right\}$

$t \in \text{JOIN}(\mathcal{T}) \left\{\langle M, S' \cup S'', \right.$

$\left\{\langle C, R' \cup R''\rangle \mid \langle C, R' \rangle \in C', \langle C, R'' \rangle \in C''\rangle \cup \right.$

$\left\{\langle M, R \cup S''\rangle \mid \langle M, R \rangle \in C'\rangle \cup \right.$

$\left\{\langle M, R \cup S'\rangle \mid \langle M, R \rangle \in C''\rangle \right\mid \langle M, S', C' \rangle \in \tau_t, \langle M, S'', C'' \rangle \in \tau_t\right\}$

Figure 11 SINC Algorithm.

and are already satisfied in the GL reduct of the local program under $M$. The algorithm guarantees that a surviving tuple $\langle M, S, \emptyset \rangle$ witnesses that the set $M$ satisfies the program induced by the subtree rooted at $t$. Again, 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 $\langle \emptyset, \emptyset, \emptyset \rangle$.

The SINC algorithm is given in Figure 11. Let $\mathcal{T}$ be a tree decomposition of the primal graph of the input program. Below we provide only descriptions for types of nodes where the result is non-trivial. Recall that we force atoms occurring in choice or weight rules, to have a corresponding

---

6 Let $\Pi$ be a program and $r \in \Pi$. Then, the rule $r' \in \{r\}^{(t, S)}$ is the corresponding rule of $r$ in $\Pi^{(t, S)}$, if $\{r\}^{(t, S)} \neq \emptyset$. 

---
vertices that appear together in at least on bag. For an atom introduction node \( t \in AI(T) \), tuples of
cchild nodes are computed very similar as before. Every local answer set \( M \) of the child node \( t' \) and
the “introduced” atom \( a \in A_t \setminus A_{t'} \) is extended either by taking \( M \) or \( M \cup \{a\} \). Further, the set \( S \)
of the child node \( t' \) is extended by all rules of the local program \( \Pi^{(t,S)} \) that are satisfied by \( M \) or
\( M \cup \{a\} \), respectively. Again, all possible subsets of \( M \) or \( M \cup \{a\} \), respectively, are considered
as counter candidates. Further, the set \( R \) of the child node \( t' \) for a counter candidate is extended
by each rule \( r \) of the GL reduct of the local program \( \Pi^{(t,S)} \) under \( M \) such that \( r \) is satisfied by \( M \)
or \( M \cup \{a\} \), respectively. For a rule removal node \( t \in RR(T) \) and the “removed” rule \( r \), we can
discard all tuples where some \( r \) is not yet satisfied (\( r \notin S \) or \( r \notin R \), respectively), since \( r \) will never
appear again in an ancestor node, which is ensured by Definition of a tree decomposition due to the
connectedness condition (Condition 3). By a similar argument, we can simply remove the rule \( r \)
from the tuples. For a join node \( t \in JOIN(T) \), we ensure that tuples of the left and right children
are merged that agree on the sets \( M \).

A slightly modified implementation of the SINC is the INVSINC algorithm, where we again
store in each tuple the set \( \overline{C} \) of “inverse” counter models, instead of the set \( C \) of counter models,
in order to see whether this leads to a speed-up. More precisely, for a set \( M \), a program \( \Pi \), and a
tree decomposition \( T \), the set \( C \) consists of all sets \( C \subseteq M \) such that \( C \) is a model of \( (\Pi^{(t,S)})^M \).
However, the set \( \overline{C} \) consists of all sets \( \overline{C} \subseteq M \) such that \( \overline{C} \) is not a model of \( (\Pi^{(t,S)})^M \). Since the
adaptation of the conditions and results in Figure 11 to the set \( \overline{C} \) of inverse counter models is straight
forward, we omit full definitions here.

**Example 3.10.** Consider again the problem VC and the graph \( G = (V, E) \) from Figure 10a.
Remember that the program \( \{ \text{in}_x \lor \text{in}_y : xy \in E \} \) encodes the problem VC in ASP given in
Figure 10b. Further, Figure 10c illustrates the semi-incidence graph \( S(\Pi) \) and Figure 10d visualizes
a decomposition of the semi-incidence graph \( S(\Pi) \).

In Figure 12 we present the result of the SINC algorithm on the decomposition of Figure 10d
using DynASP 2. Figure 13 shows running the same instance with the improved algorithm of SINC
using DynASP 2.5. Comparing these two computations by visual aspects again leads to favouring
DynASP 2.5.
Figure 12 Dynamic programming tables for the nodes of decomposition $T$ of Figure 10 using SINC with the DynASP 2 approach.
Figure 13 Dynamic programming tables for the nodes of decomposition $T$ of Figure 10 using SINC with the DynASP 2.5 approach.

### 3.4.3 Algorithms INC and INVINC

The INC is based on the incidence graph of a given program. The intuition is similar to the one of SINC. However, since the atoms that occur in choice or weight rules might not have corresponding vertices that occur together in one bag, we require in addition SINC to remember satisfiability.
information” for rules, which are not yet known to be satisfied.

A tuple for the INC algorithm in the table $\tau_t$ for a node $t$ of the tree decomposition is a tuple $\langle M, S, U, C \rangle$. Again $M$ represents a model and $S$ represents rules that are satisfied in the local program. In addition, $U$ consists of a set of pairs $(r, s)$ where $r$ is a rule in $R_t$ not yet satisfied and $s$ is a non-negative integer representing for a choice rule the number of atoms in the choice rule that have been taken into the (local) model in the table of some descendant node and for a weight rule the sum of weights of literals in the weight rule that have been taken or not taken, respectively, into the (local) model in the table of some descendant node. We can think of $s$ as a “local satisfaction state” that incorporates information from descendant nodes for choice and weight rules. Then, the set $U$ represents rules together with their “satisfaction states”. The set $C$ consists of tuples $(C, R, T)$ where $C$ again represents a counter model, $R$ again consists of all rules that correspond to vertices in $\chi(t)$ and are already satisfied in the GL reduct of the local program under $M$, and $T$ similar to $U$ consists of “local satisfaction states” with respect to the GL reduct. This is needed in order to check, on assignment/introduction of a new atom, whether rule $r$ is now satisfied, since this now depends on the weight of atoms previously assigned. Once $w_l$ reaches the bound of some weight rule $r$, it can be concluded that the body of $r$ is satisfied. Otherwise, if $w_h$ drops below this bound, it can be concluded that the body of $r$ is false and hence that $r$ is satisfied.

As we will see later, this is the reason, why both the SINC and INC algorithm require to keep track of satisfied rules in the tuple structure.

### 3.5 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 $O(2^{2w} \cdot n)$, where $w$ is the width of the underlying tree decomposition, and $n$ is the size of $\Pi$; such algorithms are also known as fixed-parameter tractable; see e.g., [35].

We omit the proof, which is lengthy but straightforward and follows from the fact that tree decompositions are linear in size of the underlying graph and the number of tuples in each node is bounded by $2^{2w}$. Finally, let us note that for INC, assuming bounded weights is non-restrictive in practice, since weights are generally small in practice, and real-world computer systems use numbers of bounded size.

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 by Samer and Szeider [84].
3.6 Extensions

3.6.1 Extensions to Optimization

In this section, we introduce the extension of our three proposed ASP dynamic programming algorithms such that we can handle minimization. For this purpose, each literal \( l \) gets assigned a nonnegative weight \( w \) denoted by \( \text{cst}(l) = w \). Now for each tree decomposition node \( t \), let \( \text{cst}_t(I) := \sum_{i \in I \cap \chi(t)} \text{cst}(i) + \sum_{i \in I \cap \gamma(t)} \text{cst}(\hat{i}) \) be the costs of the corresponding literal set \( I \) restricted to node \( t \). In order to compute models of minimum weight for node \( t \), we need to extend every tuple \( t \in \tau_t \) of our algorithms PRIM, SINC, INC, and INV by a cost value describing its weight (referenced by \( c(t) \)). The basic idea concerning modifying the corresponding algorithm description is conceptually simple. When an atom \( a \) gets introduced \( (AI_t(a)) \), one has to store for each tuple \( t \) of \( t \), the corresponding cost value of \( a \), i.e. \( c(t) = \text{cst}(a) \) if \( a \) is true in the first component of \( t \) and \( c(t) = \text{cst}(\bar{a}) \) otherwise. Concerning join nodes \( t \) \((JOIN_t)\) we store for each tuple \( t \) of \( t \) the cost value of the first component \( M \) of \( t \), i.e. \( c(t) = \text{cst}_t(M) \). With this preparation, we can compute the minimum weight as described in the next paragraph.

**Computing minimum weight.** The three algorithms presented above, as-is, do not allow for computing the minimum weight of minimal models. However, a simple modification allows this. To this end, associate with each tuple \( t \) a number \( tc(t) \) (total cost). For tuples in leaf nodes, set this number to 0. For introduction and removal nodes, let \( f \) be the surjective function that maps a child tuple \( \overline{t} \) to tuple \( \vec{t} \), according to the algorithms given above. Then, let \( \vec{t} \) be one child tuple with \( tc(\vec{t}) = \min_{\overline{t} \in f^{-1}(\vec{t})} tc(\overline{t}) \). We define \( tc(t) = tc(\vec{t}) + c(t) \) (i.e. if two tuples map to the same tuple, only the minimum weight over both counts). For tuples in join nodes, let \( g \) be the surjective function that maps two joined child tuples \( (\overline{t}, \overline{t'}) \) to tuple \( \vec{t} \), according to the algorithms given above. Again, let \( \vec{t} \) be one child tuple with \( tc(\vec{t}) = \min_{(\overline{t}, \overline{t'}) \in g^{-1}(\vec{t})} (tc(\overline{t}) + tc(\overline{t}')) \), where \( \overline{t} \) \((\overline{t'})\) is the left \((\right)\) child’s tuple that gave rise to \( t \). We define \( tc(t) = tc(\vec{t}) - c(t) \).

3.6.2 Extensions to Counting

In the following, we transform our three proposed algorithms into \#ASP dynamic programming algorithms.

**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 \( t \) a number \( n(t) \). For tuples in leaf nodes, set this number to 1. For introduction and removal nodes, let \( f \) be the surjective function that maps a child tuple \( \overline{t} \) to tuple \( \vec{t} \), according to the algorithms given above. Then, let \( n(\vec{t}) = \sum_{\overline{t} \in f^{-1}(\vec{t})} n(\overline{t}) \) (i.e. if two tuples map to the same tuple, their counts are summed up). For tuples in join nodes, again let \( g \) be the surjective function that maps two joined child tuple \( (\overline{t}, \overline{t'}) \) to tuple \( \vec{t} \), according to the algorithms given above. Now we define \( n(\vec{t}) = \sum_{(\overline{t}, \overline{t'}) \in g^{-1}(\vec{t})} (n(\overline{t}) \cdot n(\overline{t'})) \), where \( \overline{t} \) \((\overline{t'})\) is the left \((\right)\) child’s tuple that gave rise to \( t \).
Counting minimal models. In addition, one can also count minimum-weight models via a simple modification. To this end, associate with each tuple $\vec{t}$ a number $m(\vec{t})$. For tuples in leaf nodes, set this number to 1. For introduction and removal nodes, let $f$ be the surjective function that maps a child tuple $\vec{t}'$ to tuple $\vec{t}$, according to the algorithms given above. Then, let $MC(\vec{t}) = \{ \vec{t}' \in f^{-1}(\vec{t}) | \exists \vec{t}'' \in f^{-1}(\vec{t}) : tc(\vec{t}'') < tc(\vec{t}) \}$. This helps us in defining $m(\vec{t}) = \sum_{\vec{t}' \in MC(\vec{t})} m(\vec{t}')$ (i.e., if two tuples of minimum weight map to the same tuple, their counts are summed up). For tuples in join nodes, let once more $g$ be the surjective function that maps two joined child tuple $(\vec{t}', \vec{t}'')$ to tuple $\vec{t}$, according to the algorithms given above. Now let $MCJ(\vec{t}) = \{ (\vec{t}', \vec{t}'') \in g^{-1}(\vec{t}) | \exists (\vec{d}', \vec{d}'') \in g^{-1}(\vec{t}) : tc(\vec{d}') + tc(\vec{d}'') < tc(\vec{t}') + tc(\vec{t}'') \}$. We finally define $m(\vec{t}) = \sum_{(\vec{t}', \vec{t}'') \in MCJ(\vec{t})} (m(\vec{t}').m(\vec{t}''))$, where $\vec{t}'$, $\vec{t}''$ is the left (right) child’s tuple that gave rise to $\vec{t}$.

Next, 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 3.11. The problem #ASP is \#·coNP-complete.

Proof. Membership follows from the fact that, given a program $\Pi$ and an interpretation $I$, checking whether $I$ is an answer of $\Pi$ is coNP-complete, see e.g., [65]. Hardness is a direct consequence of \#·coNP-hardness for the problem of counting subset minimal models of a CNF formula [37], 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 with respect to a cost function) is slightly higher; exact results can be established employing hardness results from other sources [60].

3.7 Comparison of the Algorithms

The algorithms presented are all based on tree decompositions of different graph representations of the ASP program. All of them have an asymptotic runtime upper bound which is double-exponential in the corresponding treewidth, and in linear time in the program size. However, incidence treewidth is a more general parameter than primal treewidth. In particular, the treewidth of the primal graph of an ASP program is clearly lower-bounded by the size of the largest rule in the program (since all atoms in a rule form a clique in the primal graph), while for the incidence graph this is not the case. Therefore, programs containing large rules may perform badly when evaluated with the PRIM algorithm, since the theoretical runtime is double-exponential in the rule size. On the other hand, the INC algorithm runtime bound is not influenced by this, but data structures of the algorithm are much more complex than those of PRIM (since satisfaction-info tuples have to be remembered for each weight and choice rule). When rule size is small (and, therefore, incidence and primal treewidth are similar), we expect PRIM to perform better in practice since there is less overhead. However, for large rules, we expect INC to outperform PRIM substantially. INVPRIM is similar to PRIM, and SINC is only sensitive to large weight rules or large choice heads, instead of large...
rules in general (since cliques only exist between atoms appearing in these constructs). Thus, our different algorithms represents a classical trade-off between using a more general parameter which decreases the theoretical runtime bound, and algorithms of lower complexity which decreases the practical overhead when implementing the algorithm.
Towards Implementation: DynASP 2 and DynASP 2.5

We implemented the system DynASP 2 based on the architecture of the **DynASP 2 approach** introduced in Section 3 and the algorithms (and variants thereof) presented in Section 3.4. The result is a fully automatic software tool targeted for evaluating ASP programs using dynamic programming on tree decompositions. The input of DynASP 2 is assumed to be SModels intermediate format [93]. However, this is not a limitation since the complete ASP-Core-2 standard can be reduced to rules of only six types specified using this format. Therefore, DynASP 2 requires a grounder such as Gringo [45, 47] in order to reduce programs of ASP-Core-2 standard to rules of appropriate type.

This section now provides deeper insight on parts of the system requiring careful engineering and concerning implementation. At first, we provide a brief overview on working with DynASP 2 in Section 4.1. There we provide a short hands-on guide concerning the application of DynASP 2 in practice. In Section 4.2, we reveal performance-critical part, followed by a system based on the **DynASP 2.5 approach** in Section 4.3 dedicated to efficient counter candidate derivation.

4.1 A quick hands-on guide to DynASP 2

After downloading, compiling and installing htd, sharp and DynASP 2 at [https://github.com/daajoe/dynasp](https://github.com/daajoe/dynasp) the software prints its command line option flags by typing `dynasp -h` (see Listing 1).

Usage: `dynasp [OPTION]... [FILE]`

Arguments to options are always mandatory. Valid options:

- `-v` output version information and exit
- `-h` display this help message and exit
- `-d` perform decomposition and print treewidth
- `-t ALG` use ALG for decomposition, where ALG is one of:
  - `mcs`: maximum cardinality search bucket elimination
  - `mf`: minimum fill edge count bucket elimination (default)
- `-b` display timing information (use twice for CSV format)
- `-s NUM` set NUM as seed for the random number generator
- `-c NUM` set algorithm configuration to NUM:
  1: primal graph, full certificates
  2: primal graph, optimized certificates
  3: primal graph, inverse certificates
  4: incidence graph, full certificates (default)
  5: incidence graph/primal constraints, full certificates
  6: incidence graph/primal constraints, optimized certificates

Listing 1  Usage description of DynASP 2

The most important option of DynASP 2 probably is `-c`, which allows for selecting the algorithm (see Section 3.4). The first two variants cover PRIM variants and the third algorithm works as **INVPRIM**. The default selection is the fourth algorithm and corresponds to INC, whereas the last two configurations are variants of **SINC**. Flag `-t` allows to change the heuristics for finding elimination orders as explained in 3.1 and `-s` allows to pass the initial seed. With the option `-d`, one can enable DynASP 2 to perform the decomposition only, i.e., DynASP 2 can in fact also be
used as a decomposer.

Note that the input can stem from stdin or a file (FILE) and is expected to be in SModels syntax as defined in the manual [93]. Since the grounder Gringo [45, 47] for instance is capable of transforming non-ground ASP programs into this format, one can call gringo together with dynasp in the following UNIX-style pipe chain.

```
gringo [FILE] | dynasp [OPTION] ...
```

If one wants to compute the number of possibilities to color the graph of Figure 1 with 3 colors such that the two end points of an edge do not have the same color, one can store the corresponding encoding of Listing 7 in Section A.1 together with the facts of Figure 3a in a file “3col_ex.lp” and run DynASP 2 with `gringo 3col_ex.lp | dynasp -bb`. The result (there are 36 possibilities) might be the similar to the following output (see Listing 2).

```
Parsing...
Initializing solver...
Decomposing...
TREEWIDTH: 6
Solving...
PASS 1 finished
done.
OPTIMAL WEIGHT: 0
SOLUTION COUNT: 36
```

Listing 2  Result of running DynASP 2 with example of Figure 3

4.2 Implementation Tricks

For performance reasons, the actual implementation of DynASP 2 differs in its behavior from the algorithm descriptions given in Section 3. The following parts provide room for optimizations, which will be quickly discussed in the remainder of this subsection: (i) Derivation of Counter Candidates, (ii) Counting, (iii) Counting Models of minimum weight, (iv) Data structure for sets of atoms, and (v) Cleaning up tuples of child nodes.

Derivation of Counter Candidates. Concerning derivation of counter candidates, there are several improvements possible. In Section 4.3 for instance, we provided details concerning an improved version based on the DynASP 2.5 approach, which tries to automatically derive the sets of counter candidates via reusing already computed tuples. Note that there might be counter candidates,
which are not among the computed tuples due to the semantics of ASP, i.e. reusing derived tuples does not suffice in general. Similarly there might be models of some program reduct, which are not among the models of the program.

One could even try to use KR techniques in order to represent and maintain the set of counter candidates in a compact way. However, DynASP 2 does not apply this method of keeping counter candidates concise at the moment.

**Counting.** Counting solutions in DynASP 2 is not the result of a different pass (as described in Section 3.6.2), the corresponding counters are rather computed during the bottom-up traversal of the first pass.

**Counting Models of minimum weight.** In the actual implementation, we compute the minimum weight of models during the bottom-up traversal, which also allows DynASP 2 to immediately count the number of minimum weight models in a similar fashion already in the first pass.

**Data structure for sets of atoms.** In Section 4.3 we present a modification of the system improving counter candidate derivation, which uses bit vectors as data structure for sets of atoms, hence allowing efficient set comparisons, subset checks, computation of set unions, intersections, inverses and many more.

**Cleaning up tuples of child nodes.** Since the number of solutions is sufficient in the end, we do not require to keep intermediate results, which allows for cleaning computed tuples for child nodes during the bottom-up traversal immediately after deriving the corresponding tuples of the current node.

### 4.3 Extended Implementation: DynASP 2.5

The implementation of the solver DynASP 2.5 is an improvement influenced by the success of D-FLAT² [15], an extension of the dynamic programming framework D-FLAT [2] targeting problems located on the second level of the polynomial hierarchy. DynASP 2.5 operates in several passes while deriving partial solutions and the corresponding counter candidates (i.e. tuples as defined and used in Section 3.3) based on the *DynASP 2.5 approach* – thereby allowing removal of non-solution tuples at an early stage during the bottom-up traversal of the tree decomposition.

The architecture of DynASP 2.5 is adapted such that the system works in three passes as seen in the flowchart of Figure 5.

---

7 D-FLAT² is open source and available at [https://www.github.com/hmarkus/dflat-2](https://www.github.com/hmarkus/dflat-2)
8 D-FLAT allows rapid prototyping of algorithms exploiting dynamic programming on tree decompositions and can be downloaded at [https://www.github.com/bbliem/dflat](https://www.github.com/bbliem/dflat)
4.3.1 A quick hands-on guide to DynASP 2.5

The syntax of DynASP 2.5 is the same as in DynASP 2 (see Section 3), but extended accordingly. In particular, the extended system allows the following flags (see Listing 3).

Usage: dynasp2.5 [OPTION]... [FILE]

Arguments to options are always mandatory. Valid options:
- v output version information and exit
- h display this help message and exit
- d perform decomposition and print treewidth
- n do not use weak normalization
- r do not use reduct speedup optimization
- l NCH enforce join nodes with at most NCH children (default: 3)
- t ALG use ALG for decomposition, where ALG is one of:
  - mcs: maximum cardinality search bucket elimination
  - mf: minimum fill edge count bucket elimination (default)
- b display timing information (use twice for CSV format)
- s NUM set NUM as seed for the random number generator
- c NUM set algorithm configuration to NUM:
  - 2: primal graph, optimized certificates (default)
  - 6: incidence graph/primal constraints, optimized certificates

Listing 3 Usage description of DynASP 2.5

First of all, the extended implementation DynASP 2.5 was only provided for two algorithms (-c), namely 2 and 6. Concerning the particular implementation we added flag -r, which disables a certain optimization based on counter candidate handling leading to performance increases in some cases.

Moreover, the internal details and in particular the way how the idea based on three passes works, forced use to enable a flag (-l), which limits the number of children per join node. In addition, per default we enabled an efficient implementation of our two algorithms based on a weak form of normalized tree decompositions.

Besides the default setting, the following configurations might be worth to try for each of the two algorithms (more details in Section 5): (i) -n -l 2, (ii) -n -l 2 -r, and (iii) -l 3 -r.

Following our running example from Figure 3, we could use the extended implementation to compute the number of 3colorings for the graph of Figure 1. Again we store the corresponding ASP program (as described in Section 3) in a file “3col_ex.lp” and run DynASP 2.5 with gringo 3col_ex.lp | dynasp2.5 -bb. The result might be the similar to the following output (see Listing 4).
Listing 4  Result of running DynASP 2.5 with example of Figure 3
5 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 2 and DynASP 2.5, and the existing solvers Cachet 1.21 [86], DepQBF [71], Clasp 3.1.4 [44], and SharpSAT 12.08 [96].

5.1 Setup / Experimental Environment

The experiments were ran 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.

In order to draw conclusions about the efficiency of our approach, we mainly inspected the (total cpu) running time\(^{10}\) and number of timeouts on the random and structured benchmark sets.

We conducted three different benchmark modes as explained in the following.

**COMPUTEAS.** Concerning finding one answer set if it exists, we compared Clasp to our approach using structured instance sets and we did not limit the treewidth of the input instances. This mode shall present the runtime results without taking care of the disadvantages (concerning high treewidth) of DynASP 2 due to its underlying technique. We used parameters “–stats=2 –opt-strategy=usc -q” for Clasp, and will refer to the following two different variants of our prototype implementation as DynASP 2(PRIM) and DynASP 2(INC). Since this mode contains also instances of treewidth larger than 20, our runs were limited to 600 seconds of CPU time.

**#ASP.** Since this mode covers the main results of this section, we conducted random and structured benchmarks using the full set of solvers already mentioned. 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 2(PRIM), DynASP 2(INVPRIM) and DynASP 2(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. Note that during a run we limited the CPU time to 300 seconds.

\(^{9}\)Since DepQBF [71] does not support counting by default, we implemented a naive counting approach into DepQBF using methods described in [70], which we call DepQBF0.

\(^{10}\)The runtime for DynASP 2(·) includes decomposition times. Note that we randomly generated three fixed seeds for the decomposition computation to allow a certain variance in decomposition features [3]. When evaluating the results, we constructed the average on the seeds per instance.
Counting – DynASP 2 vs. DynASP 2.5. We compared our two variants of DynASP using structured sets without limiting the treewidth of the instances. Again, we will refer to our two basic variants of the prototype implementation DynASP 2 resp. DynASP 2.5 as DynASP 2(PRIM) and DynASP 2(INC) resp. DynASP 2.5(PRIM) and DynASP 2.5(INC). Since also this mode contains instances of treewidth ≥ 20, our runs were limited to 600 seconds of CPU time.

5.2 Instances

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 the set 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 (see Section A.1 c2COL, w2COL, c3COL, cDS, sDS, 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 and splited by transportation type, e.g., train, metro, tram, combinations. We heuristically computed tree decompositions and obtained relatively fast decompositions of small width unless detailed bus networks were present. The encoding for c2COL 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 input graph. We also added a modified variant called w2COL involving weights of the vertices. The encoding for c3COL models to count all 3-colorings where we minimize the number of assignments to color “red”. The encoding for cDS resp. sDS models to count all minimum resp. 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.
Figure 14 COMPUTEAS: Result visualization of structured instance sets.
5.3 Results

5.3.1 ComputeAs

In Figure 14 we show cactus plots of selected structured instances concerning the consistency problem. Since the technique of DynASP 2 is based on dynamic programming on tree decompositions and not optimized for determining only one answer set, it is not a secret that the solver does not perform best under these circumstances. However, it is quite surprising on the other hand that DynASP 2 seems to work better solving the weighted 2-colorability variant (\textit{w2Col}, see Listing 6) compared to Clasp. Note that at the moment we are still trying to determine the reason for this unexpected result and in particular are conducting a series of benchmark runs where we explore different solver configurations of Clasp. For more details we refer the reader to Table 1 reporting on the average running times and number of timeouts of the solvers on the considered structured instance sets.

<table>
<thead>
<tr>
<th></th>
<th>c2Col</th>
<th>c3Col</th>
<th>cDS</th>
<th>ST</th>
<th>cVc</th>
<th>w2Col</th>
<th>sDS</th>
<th>sVc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clasp</td>
<td>0.12</td>
<td>0.11</td>
<td>0.11</td>
<td>69.00</td>
<td>0.13</td>
<td>76.39</td>
<td>53.48</td>
<td>0.10</td>
</tr>
<tr>
<td>INC</td>
<td>21.29</td>
<td>85.54</td>
<td>22.74</td>
<td>115.11</td>
<td>20.88</td>
<td>37.02</td>
<td>571.62</td>
<td>26.70</td>
</tr>
<tr>
<td>PRIM</td>
<td>21.27</td>
<td>85.20</td>
<td>22.63</td>
<td>368.31</td>
<td>20.91</td>
<td>37.37</td>
<td>574.14</td>
<td>26.66</td>
</tr>
</tbody>
</table>

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

Figure 15 #Asp: Result visualization of randomly generated instance sets.
Table 2 reports on the average running times and number of timeouts of the solvers on the considered structured instance sets. Figure 15 illustrates the running times of the solvers on the various random instance sets whereas Figure 16 shows selected structured instance set as a cactus plot.

5.3.2 #ASP

Table 2 reports on the average running times and number of timeouts of the solvers on the considered structured instance sets. Figure 15 illustrates the running times of the solvers on the various random instance sets whereas Figure 16 shows selected structured instance set as a cactus plot.

The results can be summarized as follows:

SAT-TG\RID and ASP-TG\RID: 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 2 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 2 solved all instances. The results are illustrated in the two left graphs of Figure 15.

2QBF-TG\RID and 2ASP-TG\RID: 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 2 solved half of the instances within less than 1 second, about 92% of the instances within less than 10 seconds, and provided solutions also for a large number of answer sets. The other configurations quickly produced timeouts.

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 2 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 2 ran still fast on instances with a large number of solutions. Whenever the instance had relatively
few solutions Clasp was faster, otherwise DynASP 2(·) (e.g., sVc) was faster.

The empirical results of the benchmarks confirm that our DynASP 2 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 2(·) 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 2(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.
Figure 16  #ASP: Result visualization of structured instance sets.
### 5.3.3 Counting ASP models – DynASP 2 vs. DynASP 2.5

Results of our findings comparing DynASP 2 with DynASP 2.5 are visualized in Figure 17 and summarized in Table 3. Note that the differences concerning runtime are not significant for some problems and DynASP 2.5 does not look convincing in all categories, either. However, to this end we want to point the reader’s attention to the Steiner tree problem. Algorithm INC 2.5 shows quite good results using the configuration specified in Table 3. In particular, if we compare the runtime of INC 2.5 to the one of Clasp in Table 1 where we used the same instance sets as argued at the beginning of this Section, we notice that in about the time Clasp requires for proving the existence of a Steiner tree, DynASP 2.5 is capable of not only solving the corresponding consistency problem, but also counting the number of Steiner trees. Hence, the improved algorithms using DynASP 2.5 provide room for improvements, especially since DynASP 2.5 is still a prototype implementation and contains several unlocked potentials.

#### Table 3: DynASP 2 vs. DynASP 2.5: Runtime results (given in sec.) on real-world graph instance sets and number of timeouts in brackets. Best configurations given for INC 2.5 and PRIM 2.5 (see Section 4.3.1)

<table>
<thead>
<tr>
<th></th>
<th>C2COL</th>
<th>C3COL</th>
<th>CD</th>
<th>ST</th>
<th>CV</th>
<th>W2COL</th>
<th>SD</th>
<th>SV</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC</td>
<td>21.29 (18)</td>
<td>85.54 (73)</td>
<td>22.74 (19)</td>
<td>115.11 (91)</td>
<td><strong>20.88 (18)</strong></td>
<td><strong>37.02 (29)</strong></td>
<td>571.62 (498)</td>
<td>26.70 (21)</td>
</tr>
<tr>
<td>PRIM</td>
<td><strong>21.27 (18)</strong></td>
<td><strong>85.20 (72)</strong></td>
<td><strong>22.63 (19)</strong></td>
<td><strong>368.31 (320)</strong></td>
<td>20.91 (18)</td>
<td><strong>37.37 (29)</strong></td>
<td><strong>574.14 (505)</strong></td>
<td><strong>26.66 (21)</strong></td>
</tr>
<tr>
<td>INC 2.5</td>
<td>21.69 (18)</td>
<td>102.39 (82)</td>
<td>24.30 (21)</td>
<td><strong>74.76 (62)</strong></td>
<td>21.65 (18)</td>
<td>53.01 (41)</td>
<td>569.70 (498)</td>
<td>24.96 (21)</td>
</tr>
<tr>
<td>PRIM 2.5</td>
<td>-r -n -l 2</td>
<td>-r -n -l 2</td>
<td>-r -n -l 2</td>
<td>-r -l 2</td>
<td>-r -n -l 2</td>
<td>-r -n -l 2</td>
<td>-r -l 2</td>
<td>-r -l 2</td>
</tr>
<tr>
<td></td>
<td>21.95 (18)</td>
<td>108.49 (86)</td>
<td>23.49 (20)</td>
<td>317.11 (275)</td>
<td>22.67 (19)</td>
<td>59.69 (46)</td>
<td><strong>568.56 (498)</strong></td>
<td><strong>20.93 (18)</strong></td>
</tr>
<tr>
<td></td>
<td>-r -l 5</td>
<td>-r -n -l 2</td>
<td>-r -n -l 2</td>
<td>-n -l 2</td>
<td>-r -n -l 2</td>
<td>-r -n -l 2</td>
<td>-n -l 2</td>
<td>-l 2</td>
</tr>
</tbody>
</table>
Figure 17  DynASP 2 vs. DynASP 2.5: Result visualization of structured instance sets.
6 Conclusion

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 competitive 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 [56]. Future work includes extending our algorithms to weighted model counting, to solve e.g., the Bayesian inference problem.
References


A Appendix

A.1 Used Encodings

For completion, we add the encodings for our graph problems (c2COL, w2COL, c3COL, CDSDS, ST, CVC and SVC) used in Section 5 denoted in the language of gringo. For details concerning syntax and semantics, we refer the reader to the literature [23, 44, 46, 47]. Listings 5 and 6 covers our 2-colorability variants, followed by Listing 7 (Listing 8) for our 3-colorability (variant thereof). Then we show an encoding for the dominating set variants in Listings 9 and 10 and one for the steiner tree problem in Listing 11. Finally, we present the two variants of vertex cover (see Listings 13 and 12).

```
color(r;g).
1 { map(X,C) : color(C) } 1 ← vertex(X).
← edge(X,Y), map(X,g), map(Y,g).
#minimize { 1,X : map(X,r) }.
```

Listing 5 Encoding for c2COL

```
color(r;g).
1 { map(X,C) : color(C) } 1 ← vertex(X).
← edge(X,Y), map(X,g), map(Y,g).
#minimize { W,X : map(X,r), weight(X,W) }.
```

Listing 6 Encoding for w2COL

```
color(r;g;b).
1 { map(X,C) : color(C) } 1 ← vertex(X).
← edge(X,Y), map(X,C), map(Y,C).
```

Listing 7 Encoding for 3COL

```
color(r;g;b).
1 { map(X,C) : color(C) } 1 ← vertex(X).
← edge(X,Y), map(X,C), map(Y,C).
#minimize { 1,X : map(X,r) }.
```

Listing 8 Encoding for c3COL
Listing 9  Encoding for CDS

\[
\begin{align*}
\{ \text{in}(X) \} & \leftarrow \text{vertex}(X). \\
\text{dominated}(Y) & \leftarrow \text{in}(X), \text{edge}(X,Y). \\
& \leftarrow \text{vertex}(X), \not \text{in}(X), \not \text{dominated}(X). \\
\end{align*}
\]

\[
\#\text{minimize } \{ 1, X : \text{in}(X) \}.
\]

Listing 10  Encoding for SDS

\[
\begin{align*}
\{ \text{in}(X) \} & \leftarrow \text{vertex}(X). \\
\text{dominated}(X) & \leftarrow \text{in}(Y), \text{edge}(Y,X). \\
\not \text{dominated} & \leftarrow \text{vertex}(X), \not \text{in}(X), \not \text{dominated}(X). \\
\% \text{guess potential counter candidate} \\
\text{cin}(X); \text{ncin}(X) & \leftarrow \text{vertex}(X). \\
\text{spoil} & \leftarrow \text{cin}(X), \not \text{in}(X). \\
\text{cin}(X) & \leftarrow \text{spoil}, \text{vertex}(X). \\
\text{ncin}(X) & \leftarrow \text{spoil}, \text{vertex}(X). \\
& \leftarrow \not \text{spoil}. \\
\% \text{SUB} \\
\% \text{some } X \text{ is not dominated, nor in} \\
\text{spoil} & \leftarrow \text{ncin}(X), C_1 \#\text{count}(Y: \text{ncin}(Y), \text{edge}(Y,X)), C_1 = \#\text{count}(Y: \text{edge}(Y,X)).
\end{align*}
\]

Listing 11  Encoding for ST

\[
\begin{align*}
0 \{ \text{p}(X,Y) \} 1 & \leftarrow \text{edge}(X,Y). \\
\% \text{For all partitions into two sets there must be a crossing edge.} \\
\text{s1}(X) \lor \text{s2}(X) & \leftarrow \text{vertex}(X). \\
\text{s1}(X) & \leftarrow \text{saturate}, \text{vertex}(X). \\
\text{s2}(X) & \leftarrow \text{saturate}, \text{vertex}(X). \\
& \leftarrow \not \text{saturate}. \\
\% \text{Found crossing edge?} \\
\text{saturate} & \leftarrow \text{p}(X,Y), \text{s1}(X), \text{s2}(Y). \\
\text{saturate} & \leftarrow \text{p}(Y,X), \text{s1}(X), \text{s2}(Y). \\
\% \text{Not a partition due to a set being terminal empty?} \\
\% \text{This is the case if the other set contains all terminal elements.} \\
\text{numVertices}(N) & \leftarrow N = \#\text{count}(X: \text{terminalVertex}(X)). \\
\text{saturate} & \leftarrow N \#\text{count}(X: \text{s1}(X), \text{terminalVertex}(X)), \text{numVertices}(N). \\
\text{saturate} & \leftarrow N \#\text{count}(X: \text{s2}(X), \text{terminalVertex}(X)), \text{numVertices}(N). \\
\#\text{minimize } \{ 1, X, Y : \text{p}(X,Y) \}.
\end{align*}
\]
\{ \text{in}(X) \} \leftarrow \text{vertex}(X).
\leftarrow \text{edge}(X,Y), \textbf{not} \text{in}(X), \textbf{not} \text{in}(Y).

\#\text{minimize}\{ 1, X : \text{in}(X) \}.

\textbf{Listing 12} \hspace{1cm} \text{Encoding for CVC}

\text{in}(X) \lor \text{in}(Y) \leftarrow \text{edge}(X,Y).

\textbf{Listing 13} \hspace{1cm} \text{Encoding for SVC}