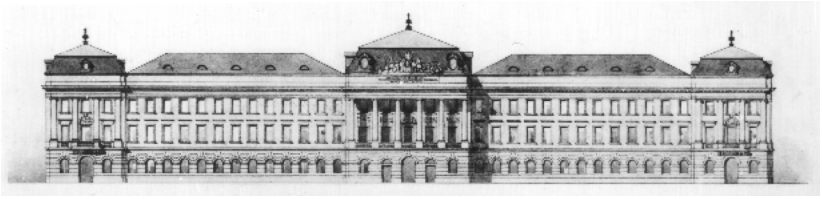
dbai

TECHNICAL REPORT



INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND ARTIFICIAL INTELLIGENCE

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

DBAI-TR-2016-101, Revised Version

Johannes K. Fichte **Markus Hecher**
Michael Morak **Stefan Woltran**

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

DBAI TECHNICAL REPORT

2016-12-15

TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DBAI TECHNICAL REPORT
DBAI TECHNICAL REPORT DBAI-TR-2016-101, REVISED VERSION, 2016-12-15

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

Johannes K. Fichte [1] **Markus Hecher** [1] **Michael Morak** [1]
Stefan Woltran [1]

Abstract. In this paper, we present novel algorithms for propositional answer set programming (ASP). In particular, we employ dynamic programming on tree decompositions which leads to algorithms that directly exploit the structure of the given ASP program. It is well-known that for tree-like ASP programs (that is, those programs whose treewidth is bounded), the complexity of solving drops from the second level of the polynomial hierarchy to polynomial time. Based on the idea of a dynamic programming algorithm previously introduced by Jakl, Pichler, and Woltran at IJCAI 2009, we propose several graph representations for ASP programs amenable for tree decomposition-based algorithms, and we present multiple ASP solving algorithms for these that can handle the full, state-of-the-art *lparse* ASP syntax that is used by most of the currently available ASP solvers. Finally, we present two prototype systems with these algorithms, using advanced implementation techniques. We show that these implementations exhibit favorable runtime behaviour especially for the problem of answer set counting.

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

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

Contents

1 Introduction

Answer set programming (ASP) [21, 45, 48, 72] is a well-established logic programming paradigm based on the stable model semantics of logic programs. Its main advantage is an intuitive, declarative language, and the fact that generally, each answer set of a given logic program describes a valid answer to the original question. Moreover, ASP solvers—see e.g. [5, 6, 38, 47]—have made huge strides in efficiency and are now readily available for anyone to use. Because of these factors, ASP has found great success in the field of artificial intelligence and has become widely used both in research and industry.

Solving a ground (i.e. propositional), disjunctive answer set program (also called ASP program) is Σ_2^P-hard in general [37]. However, ASP programs often exhibit an underlying structure that may make solving them easier. One such parameter is called treewidth. Roughly, it measures how close a given graph structure is to a tree. Following from a seminal result by Bruno Courcelle [25], the authors of [51] show the following: it turns out that, for suitable graph representations, ASP programs can actually be solved in linear time given that the treewidth is low (i.e. bounded). A corresponding linear-time algorithm using this result was presented in [61] and a prototype implementation in [74]. However, with both being preliminary works, they only treat a minimal subset of the common propositional ASP language that modern solvers use (called the lparse syntax, [96]).

The general method these algorithms use is dynamic programming on tree decompositions. A tree decomposition arranges a graph into a labelled tree in such a way that each tree node contains a subset of the vertices of the original graph. The treewidth of a given graph G provides an optimal upper bound on the size of the tree nodes in such a decomposition of G. For ASP programs, a rough overview of the mode of operation of such an algorithm could be stated as follows: given a suitable graph representation of the ASP program, the contents of each tree node identify a sub-program of the full ASP program. A tree decomposition of an ASP program can then be evaluated in a bottom-up manner, solving the partial program at each tree node. In a dynamic programming fashion, partial solutions to partial problems can be discarded as soon as it becomes clear that such a partial solution cannot be extended to a full solution. Once all nodes have been evaluated, the relevant partial solutions can then be combined to form complete solutions for the entire ASP program.

Based on these ideas, in this paper we propose and evaluate novel dynamic programming-based ASP solving algorithms that exploit, using the notion of tree decompositions, the underlying graph structure of the given (ground) input ASP program. In contrast to the algorithms and systems presented in [61, 74], we present several different graph representations that can be used for tree decomposition-based dynamic programming algorithms (namely, the primal, semi-incidence, and incidence graph representations). We then present corresponding, novel algorithms and variations that are specifically tailored to these graph representations and, furthermore, we handle the full, state-of-the-art propositional ASP syntax [96] including weight constraints and choice rules. This requires the design of more sophisticated algorithms and data structures. Further, we make use of advanced techniques for dynamic programming on tree decompositions originally proposed in previous work [13]. In additional, our solver also supports optimization statements that allow us not

only to find or count arbitrary models but also do the same for optimal models.

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:*

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

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

$$edge(v, w). \qquad (\textit{for every } vw \in E)$$
$$vertex(v). \qquad (\textit{for every } v \in V)$$

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 $color(v, red)$ and $color(v, blue)$ of an answer set of the program represent a 2-coloring of the input graph and vice versa. Correspondingly, 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 Σ_2^P-complete [37]. 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.

Related Work. Gutin [55] has recently stressed on the importance of implementing and evaluating algorithms that take certain structural features of an instance into account. Multiple papers have investigated theoretical runtime bounds for different reasoning problems when parameterized by treewidth; see e.g. [92] for SAT or [84] for constraint satisfaction problems. Several works have proposed algorithms for #SAT (that is, the model counting problem for SAT), guaranteeing favorable theoretical runtime bounds [42] and providing prototypical implementations [67, 82]. Samer and Szeider [83] 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. Backdoors to bounded treewidth SAT instances were investigated in [43].

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. As already mentioned above, the authors of [61, 74] propose algorithms for evaluating ASP programs also based on the idea of using tree decompositions. However, their algorithms cannot treat choice and weight rules and support only one graph representation (the incidence graph). Fichte and Szeider [41] have shown that tree decompositions of the dependency graph cannot be exploited to decide answer set existence more efficiently.

1.1 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 [96]. The central difference between these algorithms is the underlying graph representation of the input program. Furthermore, we improve these algorithms with

2. We show that the algorithms run in polynomial time in the input program size when the weight lower bounds and 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 based on ideas from other domains [13], 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.

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) [40]. 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.

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 [21, 47, 68]. Our algorithms treat the full smodels input format [96]. Note that state-of-the-art ASP grounders output the smodels internal format and support the full ASP-Core-2 language [23].

In the following, we provide definitions for the most important rule types. Let ℓ, 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

$$\{a_1; \ldots; a_\ell\} \leftarrow a_{\ell+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n.$$

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:

$$a_1 \vee \cdots \vee a_\ell \leftarrow a_{\ell+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n.$$

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

$$a_1, \cdots, a_\ell \leftarrow w \leqslant \{a_{\ell+1} = w_{\ell+1}, \ldots, a_m = w_m,\ \neg a_{m+1} = w_{m+1}, \ldots, \neg a_n = w_n\}.$$

where $\ell \leq 1$ with the intuitive meaning that a_1 must be true (or the rule is unsatisfied if $\ell = 0$) 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

$$\leftsquigarrow l_1[w_1].$$

with the intuitive meaning that setting atom a of literal $l_1 = a$ ($l_1 = \neg a$) to true (false) counts as penalty of weight w_1 if optimization rules are subject to minimization.

A *rule* is either a disjunctive, or a choice, a weight, or an optimization rule. For a rule r, we write $\text{at}(r) := H(r) \cup B^+(r) \cup B^-(r)$ (*atoms* of r) and $B(r) := B^+(r) \cup [\bigcup_{b \in B^-(r)}(\neg b)]$ (*body* of r) where $H(r)$ (*head* of r), $B^+(r)$ (*positive body* of r) and $B^-(r)$ (*negative body* of r) will be defined in

the following. For a choice or disjunctive rule r, we write $\{a_1, \ldots, a_\ell\} = H(r)$, $\{a_{\ell+1}, \ldots, a_m\} = B^+(r)$, and $\{a_{m+1}, \ldots, a_n\} = B^-(r)$. For a weight rule r, we write $\{a_1, \ldots, a_\ell\} = H(r)$, $\{a_{\ell+1}, \ldots, a_m\} = B^+(r)$, $\{a_{m+1}, \ldots, a_n\} = B^-(r)$, $\mathrm{wght}(r, a)$ maps an atom a in $B^+(r) \cup B^-(r)$ to its corresponding weight and 0 otherwise, $\mathrm{wght}(r, A) = \sum_{a \in A} \mathrm{wght}(r, a)$ for a set $A \subseteq B^+(r) \cup B^-(r)$, and $\mathrm{bnd}(r) = w$. For an optimization rule r, we write $\mathrm{wght}(r) = w_1$ and if $l_1 = a$ ($l_1 = \neg a$), $\{a\} = B^+(r)$ and $\emptyset = B^-(r)$ ($\{a\} = B^-(r)$ and $\emptyset = B^+(r)$). A rule r is called *negation-free* if $B^-(r) = \emptyset$ and a disjunctive rule r is *normal* if $|H(r)| \leq 1$. A *ground answer set program* (or, simply, *program*) is a set Π of rules. Terminology about rules naturally extends to programs. Further, for a program Π we let $\mathrm{CH}(\Pi) \subseteq \Pi$ consist of all choice rules of Π, $\mathrm{DISJ}(\Pi) \subseteq \Pi$ consist of all disjunctive rules of Π, $\mathrm{OPT}(\Pi) \subseteq \Pi$ consist of all optimization rules, and $\mathrm{WGT}(\Pi) \subseteq \Pi$ consist of all weight rules of Π. We denote the set of atoms occurring in Π by $\mathrm{at}(\Pi)$.

We follow standard definitions of answer sets [45]. Let M be a set $M \subseteq \mathrm{at}(\Pi)$. We say M *satisfies* r if (i) $(H(r) \cup B^-(r)) \cap M \neq \emptyset$ or $B^+(r) \not\subseteq M$ for disjunctive rule $r \in \Pi$, (ii) $H(r) \cap M \neq \emptyset$ or $\Sigma_{a_i \in M \cap B^+(r)}\ w_i + \Sigma_{a_i \in B^-(r) \setminus M}\ w_i < w$ for every weight rule $r \in \Pi$, and (iii) if r is a choice or an optimization rule it is always satisfied.

We say M is a model of Π, $M \vDash \Pi$ for short, if M satisfies every rule $r \in \Pi$. For convenience, we use $\mathrm{Mod}(\mathcal{C}, \Pi) := \{\, C : C \in \mathcal{C}, C \vDash \Pi \,\}$ (set of models) for a set of sets $C \subseteq \mathrm{at}(\Pi)$ and $\mathrm{SatRules}(\Pi, M) := \{\, r : r \in \Pi, M \vDash \{r\} \,\}$ (satisfied rules) for a set $M \subseteq \mathrm{at}(\Pi)$.

The *reduct* r^M of (i) a choice rule r is a set $\{\, a \leftarrow B^+(r) : a \in H(r) \cap M \,\}$ of normal rules, (ii) a disjunctive rule r is the disjunctive rule $H(r) \leftarrow B^+(r)$, and (iii) a weight rule r is the rule $H(r) \leftarrow w' \leqslant [a = W(a) : a \in B^+(r)]$, where $w' = w - \Sigma_{a_i \in B^-(r) \setminus M}\ w_i$. The *GL reduct* (Gelfond-Lifschitz reduct) Π^M is the program that consists of (i) reducts r^M where $B^-(r) \cap M = \emptyset$ for all choice and disjunctive rules $r \in \Pi$ and (ii) reducts r^M for all weight rules $r \in \Pi$.

A set $M \subseteq \mathrm{at}(\Pi)$ is an *answer set* of a program Π if M is a subset minimal model of the reduct Π^M.

We call $\mathrm{o}(\Pi, M, A) := \sum_{r \in \mathrm{OPT}(\Pi) : (B^+(r) \cap M) \cup ([A \cap B^-(r)] \setminus M) \neq \emptyset} \mathrm{wght}(r)$ the *objective* of Π and M with respect to set of atoms $A \subseteq \mathrm{at}(\Pi)$. A set $M \subseteq \mathrm{at}(\Pi)$ is an *optimal answer set* if M is an answer set of Π and the objective $\mathrm{o}(\Pi, M, \mathrm{at}(\Pi))$ is minimal over all answer sets of Π.

Example 2.1. *Consider the following program* Π*:*

$$a \leftarrow 2 \leqslant 2\{b = 2, c = 1\}.$$
$$b \vee d \leftarrow .$$

This program has two answer sets: $\{a, b\}$ *and* $\{d\}$. □

ASP Problems

We consider the following fundamental ASP problems.

Problem:	CHECK
Input:	A program Π and a set $M \subseteq \mathrm{at}(\Pi)$.
Task:	Decide whether M is an (optimal) answer set of Π.

Problem:	CONSISTENCY
Input:	A program Π.
Task:	Decide whether Π has an (optimal) answer set.

Problem:	BRAVE REASONING
Input:	A program Π and an atom $a \in \text{at}(\Pi)$.
Task:	Decide whether a belongs to some (optimal) answer set of Π.

Problem:	COMPUTEAS
Input:	A program Π.
Task:	Output an (optimal) answer set of Π if one exists, otherwise output No.

Problem:	#ASP
Input:	A program Π.
Task:	Output the number of (optimal) answer sets of Π.

Problem:	ENUMASP
Input:	A program Π.
Task:	List all (optimal) answer sets of Π.

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 [8, 58, 78].

We use the asymptotic notation $\mathcal{O}(\cdot)$ in the standard way. Let Σ and Σ' 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 $I \notin L$ 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 $\mathcal{O}(\|I\|^c)$ 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 $\mathcal{O}(\|I\|^c)$ 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 $\mathcal{O}(\|I\|^c)$. 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, 90, 91, 97] up to the second level. The Polynomial Hierarchy consists of complexity classes Σ_i^p for $i \geq 0$ based on the following definitions: $\Sigma_0^p := P$ and $\Sigma_{i+1}^p = \mathrm{NP}^{\Sigma_i^p}$ 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 $L \in \mathrm{C}$, i.e., a decision problem $L' \in \mathrm{NP}^{\mathrm{C}}$ is non-deterministically polynomial-time solvable using any problem $L \in \mathrm{C}$ as an oracle. A decision problem $L' \in \mathrm{P}^{\mathrm{C}[\log]}$ is deterministically polynomial-time solvable doing $\mathcal{O}(\log n)$ calls to any problem $L \in \mathrm{C}$ as an oracle. Moreover, $\Pi_k^p :=$ co-Σ_k^p. Note that $\mathsf{NP} = \Sigma_1^p$, co-$\mathsf{NP} = \Pi_1^p$, $\Sigma_2^p = \mathrm{NP}^{\mathrm{NP}}$, and $\Pi_2^p =$ co-$\mathsf{NP}^{\mathrm{NP}}$.

A *witness function* is a function $\mathcal{W}\colon \Sigma^* \to 2^{\Sigma'^*}$ that maps an instance $I \in \Sigma^*$ to a finite subset of Σ'^*. 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 $\mathcal{O}(\|I\|^c)$ for some constant c and (ii) for every instance $I \in \Sigma^*$ the decision problem $\mathcal{W}(I)$ belongs to the complexity class $\mathcal{C}$. Then, $\#\cdot$ 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 $I \in \Sigma^*$, 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 from 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 $L' \in$ C to L. If in addition $L \in$ 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 Σ_2^p-complete [39], and remains Σ_2^p-complete when the input is restricted to disjunctive programs [37] or weight programs where negative weights are allowed [39]. However, the complexity of the problem CONSISTENCY drops to NP-complete when the input is restricted to normal programs [12, 71], or choice [89], or weight programs with non-negative weights [89]. The problem BRAVE REASONING is $P^{\Sigma_2^p[\log n]}$-complete [22, 36], BRAVE REASONING is Σ_2^p-complete when the input is restricted to disjunctive programs without optimization rules [37], or weight programs where negative weights but no optimization rules are allowed [39]. BRAVE REASONING is $P^{\mathrm{NP}[\log n]}$-complete when the input is restricted to normal or choice programs with optimization rules [36], CHECK of ASP is Π_2^P-complete in general, is co-NP-complete when the input is restricted to disjunctive programs [37], or weight programs where negative weights are allowed [39], or normal programs with optimization rules [36]. The complexity of the problem CHECK drops to P when the input is restricted to normal programs [71], choice [89], or weight programs with non-negative weights [89]. #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.

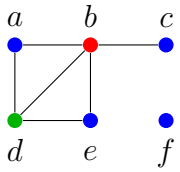


Figure 1 A graph and a 3-coloring of the graph.

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 $E \subseteq \{\, \{u, v\} \subseteq V : u \neq v \,\}$ 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 $v_i v_{i+1}$ 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 $v_1v_2, \ldots, v_{k-1}v_k, v_kv_1$. 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 $u, v \in U$ or $u, v \in W$. G is a $(k \times \ell)$-grid if $V = \{\, v_{1,1}, \ldots, v_{1,\ell}, v_{2,\ell}, \ldots, v_{k,\ell} \,\}$ and $E = \{\, v_{i,j}v_{i',j'} : |i - i'| + |j - j'| = 1 \,\}$. 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 $E \subseteq \{\, (u, v) \in V \times V : u \neq v \,\}$ 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* u and a vertex $u \in \{\, w : (v, w) \in E \,\}$ 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 $c : V \to \{1, \ldots, k\}$ *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 CDS 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 CVC asks to give a minimum (cardinality-minimal) vertex cover of a given graph. Then,

the counting problem SVC 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 [20, 31].

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 decommissions of low width. In Section 3.2, 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 dynamic programming algorithms and variants thereof based on two different graph representations to solve the problems CONSISTENCY, BRAVE REASONING, and COMPUTEAS from a theoretical perspective. Section 3.4 provides considerations on the correctness and the runtime of the algorithms. Then, we adapt the algorithms to compute answer sets of programs with optimization rules, solve the counting problem #ASP, and provide complexity result for the problem #ASP in Section 3.5. Finally, we conclude with a comparison in Section 3.6.

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, 14, 17]. 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 [34, 51, 80]. In particular, instances from practical settings often exhibit small treewidth (see, e.g., [4, 54, 60, 66, 73, 94]).

Definition 3.1 ([81]). *Let $G = (V, E)$ be graph, $T = (N, E_T, r)$ a rooted tree, and $\chi : N \to 2^V$ a function that maps to each node $t \in T$ a set of vertices. We call the sets $\chi(\cdot)$* bags *and N the set of nodes. Then, the pair $\mathcal{T} = (T, \chi)$ 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.*

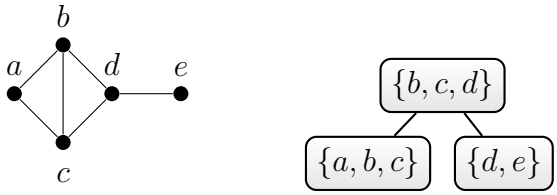


Figure 2 A graph G (left) and a tree decomposition of G width 2 (right).

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, χ) 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 ([63])**.** *A tree decomposition (T, χ) 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 $|\chi(t) \triangle \chi(t')| = 1$, 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, χ) 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 ([63])**.** *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 at 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 [25] 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 [16], logic, and artificial intelligence [52, 53, 80]. For a comprehensive introduction and examples we refer to other sources [26].

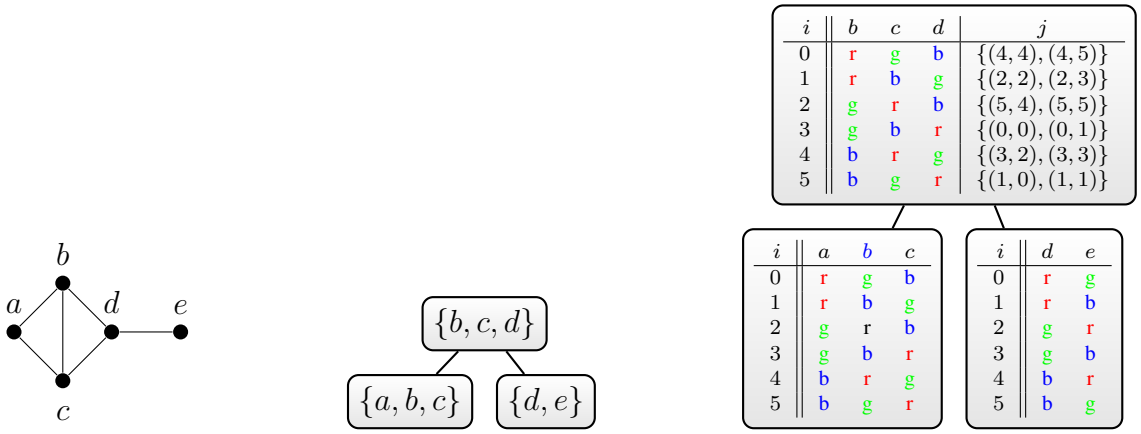
Dynamic algorithms on tree decompositions all share a common structure. The tree decomposition is traversed from the leaf nodes to the root node. At each node a sub-problem is solved, which consists of the part of the problem instance that is induced by the content of the bag of the node. This results 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 of its induced sub-problem. 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 $\mathcal{T}$ of G of width* 2*. Figure 3c illustrates the tables, which are computed when running dynamic programming on the tree decomposition $\mathcal{T}$. 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 sub-problem represented by the bag (to the sub-problem).*

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.



i	b	c	d	j
0	r	g	b	$\{(4,4),(4,5)\}$
1	r	b	g	$\{(2,2),(2,3)\}$
2	g	r	b	$\{(5,4),(5,5)\}$
3	g	b	r	$\{(0,0),(0,1)\}$
4	b	r	g	$\{(3,2),(3,3)\}$
5	b	g	r	$\{(1,0),(1,1)\}$

i	a	b	c
0	r	g	b
1	r	b	g
2	g	r	b
3	g	b	r
4	b	r	g
5	b	g	r

i	d	e
0	r	g
1	r	b
2	g	r
3	g	b
4	b	r
5	b	g

(a) A graph G. (b) A tree decomposition $\mathcal{T}$ of G of width 2. (c) The computed dynamic programming tables.

Figure 3 Dynamic programming for the problem 3-COL on the tree decomposition $\mathcal{T}$. We represent the colors by "r", "g", and "b".

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 and 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 $2^{\mathcal{O}(k^3)} \cdot |V|$ [15] 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) [32, 33, 87]. Further, several exact algorithms and algorithms to compute upper bounds have been proposed [10, 18, 49, 85, 88].

Since the running times of these algorithms are far from practical on large graphs, we often use efficient heuristics [30, 50] that produce a tree decomposition of reasonably low, but not necessarily optimal, width[2]. Various heuristics have been proposed: greedy heuristic algorithms include Maximum Cardinality Search (MCS) [93], Min-Fill heuristic [29], and Minimum Degree heuristic [11]. Metaheuristic techniques have been provided in terms of genetic algorithms [65, 76], ant colony optimization [56], and local search based techniques [24, 62, 75]. For a detailed description, we refer to recent surveys [18, 57].

In this paper, we use the graph decomposition library htd [1] to heuristically construct tree decompositions of a given graph. The library relies on the so-called bucket elimination algorithm [28], which works as follows: Let the pair (T, χ) 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 $\chi(n) := \{\, u : u \textit{ is a neighbor of } v \,\}$, (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 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. 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 ("local witnesses") together with sets of sets of atoms ("local counterwitnesses"). For a node $t \in N$ of a tree decomposition $\mathcal{T}$ and a mapping σ that maps nodes of $\mathcal{T}$ to tables, we write *corresponding table* for the table $\sigma(t)$ that is assigned to the node t.

1. We first of all construct a graph representation (given in the next subsection) of the given input program.

2. Compute a tree decomposition of the problem instance by means of heuristics, 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.

[2]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.

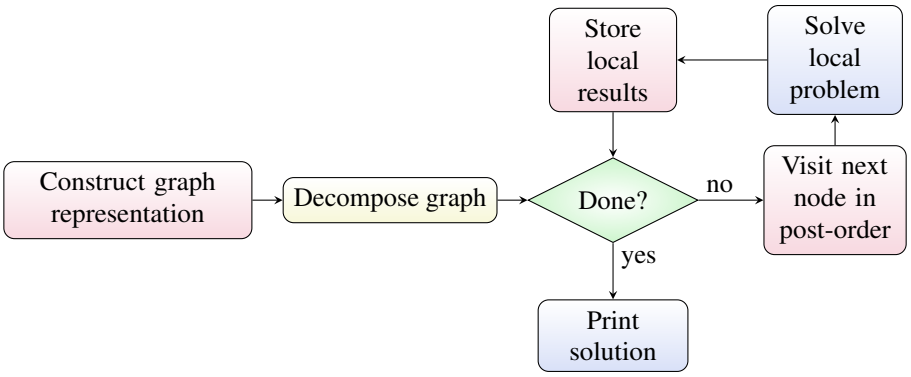


Figure 4 Control flow in DynASP 2

3. For every node n in the tree decomposition (in a bottom-up traversal) compute assignments M forming local witnesses corresponding to n, compute proper subsets $\mathcal{C}$ of M, forming local counterwitnesses contradicting the optimality of M, and store the tuple $\langle M, \mathcal{C} \rangle$ in the corresponding table. For more details of this intuitive, informal description we refer to our algorithms discussed in Section 3.3.

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.[3]

Among these tasks, the one that is entirely problem-specific and graph representation specific is the solving of 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.* [13] 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 assignments M forming local witnesses corresponding to the node and store M in the corresponding table ("Pass I"). Then, for every node in the tree decomposition (in a top-down traversal) remove local witnesses, which do not extend to a local witness in the table for the parent node, from the corresponding table ("Purge non-witnesses").

II. For every node in the tree decomposition construct a second table, which will contain "local counterwitnesses" for a local witness not being subset-minimal. For every node in the tree decomposition (in a bottom-up traversal) compute subsets $\mathcal{C}$ of the local witnesses obtained in Pass I from the corresponding table ("Pass II") and store $\mathcal{C}$ in the second table. Then, for

[3]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.

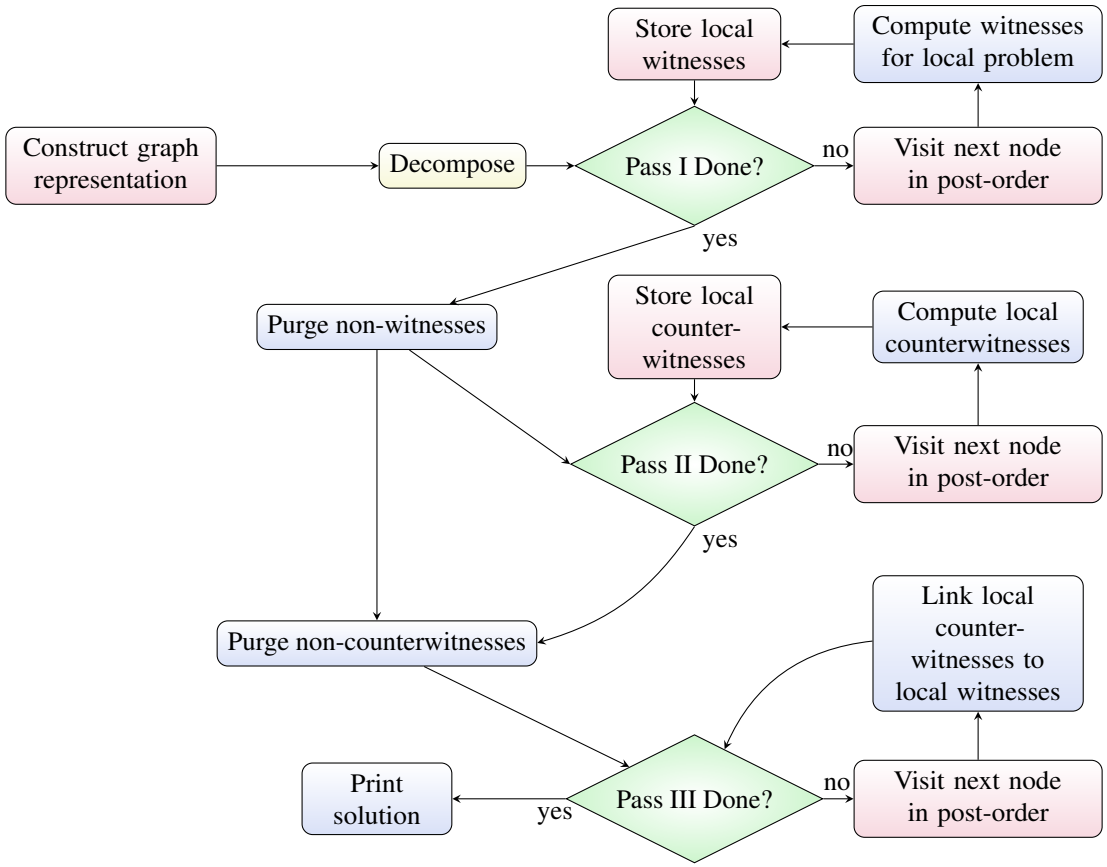


Figure 5 Adapted control flow in DynASP 2.5

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-counterwitnesses").

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., local witnesses to local counterwitnesses. 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.* [13]. In fact, results of the advanced approach are equivalent. However, practical advantages are (i) we can reuse certain local counterwitnesses for various local witnesses and every local counterwitness is only computed once among all local witnesses and (ii) using three passes allows for purging local witnesses and counterwitnesses early.

3.3 Solving CONSISTENCY, BRAVE REASONING, 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 (see forthcoming Definition 3.6), Algorithm SINC uses the incidence graph, but in addition atoms appearing in weight or choice rules form a clique (see forthcoming Definition 3.8), whereas INC uses only the incidence graph (see forthcoming Definition 3.14). Presenting the algorithms in this order allows us to incrementally explain the added difficulties and the technical necessities implied by solving ASP with dynamic programming on tree decompositions on these three graph types. Note that in this section, we restrict ourselves to the description of the algorithms for the problem CONSISTENCY since the nature of our algorithms for the problem CONSISTENCY actually can be used to solve problem COMPUTEAS (ENUMASP) via an according postprocessing step, which targets reconstructing the answer set(s) for the input program via combining certain local assignments of the decomposition nodes' tables accordingly. Moreover, note that the problem BRAVE REASONING with respect to a program Π and an atom a can be solved via the consistency (co-)problem (IN-)CONSISTENCY and the modified program $\Pi \cup \{\leftarrow a\}$.

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 τ_t is derived from the tuples of the tables $\tau_{t'}$ and $\tau_{t''}$.

3.3.1 Algorithms PRIM and INVPRIM

The first algorithm (PRIM) to solve CONSISTENCY is based on the primal graph. In order to describe this algorithm, we require a suitable definition of the primal graph and fixed notation first. The following definition extends concepts of Samer and Szeider [83] from the propositional setting to answer set programming and parts that have already been introduced by Jakl, Pichler, and Woltran [61].

Definition 3.6. *Let Π be a program. The* primal graph $P(\Pi)$ *of Π 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 $a, b \in \text{at}(r)$. For an atom a and a vertex v_a, we call the vertex v_a a corresponding vertex of atom a, and vice versa.*

For a tree decomposition $\mathcal{T} = (T, \chi)$ of the primal graph $P(\Pi)$ for program Π where $T = (N, \cdot, n)$, and a node $t \in N$, we let A_t contain all atoms $\text{at}(\Pi)$ that have a corresponding vertex in $\chi(t)$ and R_t contain all rules $r \in \Pi$ such that $\text{at}(r) \subseteq A_t$[4]. Furthermore, the *local (primal) program* Π_t *in* t consists of all rules $r \in \Pi$ such that $r \in R_{t'}$ for some $t' \in N'$ of the induced

[4]Since the primal graph contains a clique on all corresponding vertices to atoms that participate in a rule r, it follows from the properties of a tree decomposition that there will be at least one node whose bag contains all the corresponding vertices to atoms of rule r.

subtree $T' = (N', \cdot)$ of T rooted at t. We omit t if it is clear from the context to which t we refer. Note that it is easy to see that $\Pi_n = \Pi$ for the empty root n.

In the algorithm, we will construct the table τ_t separately for each node depending on the type of the node, which are defined in the following. Let $\mathcal{T}$ be a normalized tree decomposition of the primal graph $P(\Pi)$ of Π. 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 or removal node, respectively, where the bag of node and its child differ in a corresponding atom $a \in \mathrm{at}(\Pi)$. We denote the set of all atom introduce nodes of $\mathcal{T}$ by $\mathrm{AI}(\mathcal{T})$, the set of all atom removal nodes of $\mathcal{T}$ by $\mathrm{AR}(\mathcal{T})$, the set of all join nodes of $\mathcal{T}$ by $\mathrm{JOIN}(\mathcal{T})$, and the set of all leaf nodes of $\mathcal{T}$ by $\mathrm{LEAF}(\mathcal{T})$.

Tuples for the PRIM algorithm are of the form $\langle M, \mathcal{C} \rangle$, where $M \subseteq A_t$ represents a model of R_t in t and $\mathcal{C} \subset 2^M$ forming models $C \subsetneq M$ of the reduct $(R_t)^M$ and therefore representing counterwitnesses to M being an answer set of R_t. The existence of such a tuple $\langle M, \emptyset \rangle$ in table τ_t witnesses the existence of an answer set $M' \supseteq M$ for the local program in t, i.e., M is a model of R_t and can be extended to a model M' of the local program Π_t. Therefore M is sometimes referred to as *local (primal) witness* witnessing the existence of $M' \supseteq M$ with M' being an answer set of Π_t. On the other hand, since each $C \in \mathcal{C}$ fulfills $C \subsetneq M$ and can be extended to a model $C' \subsetneq M'$ of the program $\Pi_t^{M'}$, it is referred to as *local (primal) counterwitness* to M. Then, a model of the input program Π exists if the (empty) root contains a tuple v, and v is an answer set of Π, if $v = \langle \emptyset, \emptyset \rangle$.

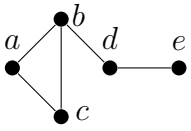
Algorithm 1: PRIM algorithm.

Data: Program Π, Node t and its children t' and t'', if exist(s), of the fixed tree decomposition $\mathcal{T}$ of $P(\Pi)$.
Result: Modified table τ_t for node t.

if $t \in \mathrm{LEAF}(\mathcal{T})$ **then**
  $\tau_t := \left\{ \left\langle \emptyset, \emptyset \right\rangle \right\}$
if $t \in \mathrm{AI}(\mathcal{T})$ *and let* $a \in A_t \setminus A_{t'}$ **then**
  $\tau_t := \left\{ \left\langle M, \mathrm{Mod}(\mathcal{C}, R_t^M) \right\rangle \;\middle|\; \langle M, \mathcal{C} \rangle \in \tau_{t'}, M \vDash R_t \right\} \bigcup$
  $\left\{ \left\langle M \cup \{a\}, \mathrm{Mod}(\mathcal{C} \cup [\mathcal{C} \sqcup \{a\}] \cup \{M\}, R_t^{(M \cup \{a\})}) \right\rangle \;\middle|\; \langle M, \mathcal{C} \rangle \in \tau_{t'}, M \cup \{a\} \vDash R_t \right\}$
if $t \in \mathrm{AR}(\mathcal{T})$ *and let* $a \in A_{t'} \setminus A_t$ **then**
  $\tau_t := \left\{ \left\langle M \setminus \{a\}, \{C \setminus \{a\} \mid C \in \mathcal{C}\} \right\rangle \;\middle|\; \langle M, \mathcal{C} \rangle \in \tau_{t'} \right\}$
if $t \in \mathrm{JOIN}(\mathcal{T})$ **then**
  $\tau_t := \left\{ \left\langle M,\ (\mathcal{C}' \cap \mathcal{C}'') \cup (\{M\} \cap \mathcal{C}'') \cup (\mathcal{C}' \cap \{M\}) \right\rangle \;\middle|\; \langle M, \mathcal{C}' \rangle \in \tau_{t'}, \langle M, \mathcal{C}'' \rangle \in \tau_{t''} \right\}$

Algorithm 1 provides the description of how the tables τ_t are computed by our PRIM algorithm[5]. Therefore, 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. For an atom introduce node $t \in \mathrm{AI}(\mathcal{T})$ each local witness M of the child node t' and the "introduced" atom $a \in A_t \setminus A_{t'}$ is extended either by taking M or $M \cup \{a\}$. We verify whether M or $M \cup \{a\}$ is a model of the

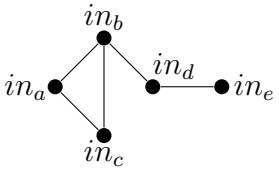
[5]We use for program Π, set of sets $\mathcal{S}$ and element e the abbreviation $\mathcal{S} \sqcup \{e\} := \{ S \cup \{e\} : S \in \mathcal{S} \}$.



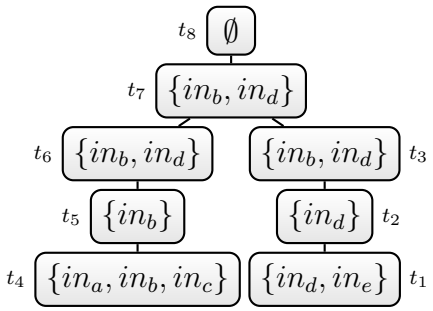
(a) A graph G

$$in_a \vee in_b.\ in_a \vee in_c.\ in_b \vee in_c.\ in_b \vee in_d.$$
$$in_d \vee in_e.$$

(b) Program Π which encodes the problem VC for G



(c) Primal graph $P(\Pi)$



(d) A tree decomposition $\mathcal{T}$ of $P(\Pi)$

Figure 6 Example graph and tree decomposition for the problem VC.

program 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 R_t into account. Intuitively, R_t consists of each rule where vertices in a bag correspond entirely to atoms that occur in the rule. Since we can verify (without additional information) whether M or $M \cup \{a\}$ is a model of R_t, we can immediately discard all tuples where M or $M \cup \{a\}$, respectively, is not a model of R_r. All possible subsets of M or $M \cup \{a\}$, respectively, are considered as local counterwitnesses. A local counterwitness of a local witness can be obtained directly from extending a local counterwitness C in the table of the child node by taking C or $C \cup \{a\}$, respectively. For a join node $t \in \text{JOIN}(\mathcal{T})$, we enforce that left and right child tuples agree on local witnesses and counterwitnesses.

Example 3.7. *Consider the problem* VC *and the graph $G = (V, E)$ from Figure 6a. The program $\{\ in_x \vee in_y : xy \in E\ \}$ encodes the problem* VC *in ASP and is given in Figure 6b. Further, Figure 6c illustrates the primal graph $P(\Pi)$ and Figure 6d visualizes a decomposition of the primal graph $P(\Pi)$. In Figure 7 we present the result of the* PRIM *algorithm on the decomposition of Figure 6d 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 8 represents running the same instance with the improved algorithm of* PRIM *using DynASP 2.5. Comparing these two figures leads to the observation that algorithm DynASP 2.5 compactly represents the resulting computation tree and does not recompute local counterwitnesses for different local witnesses.*

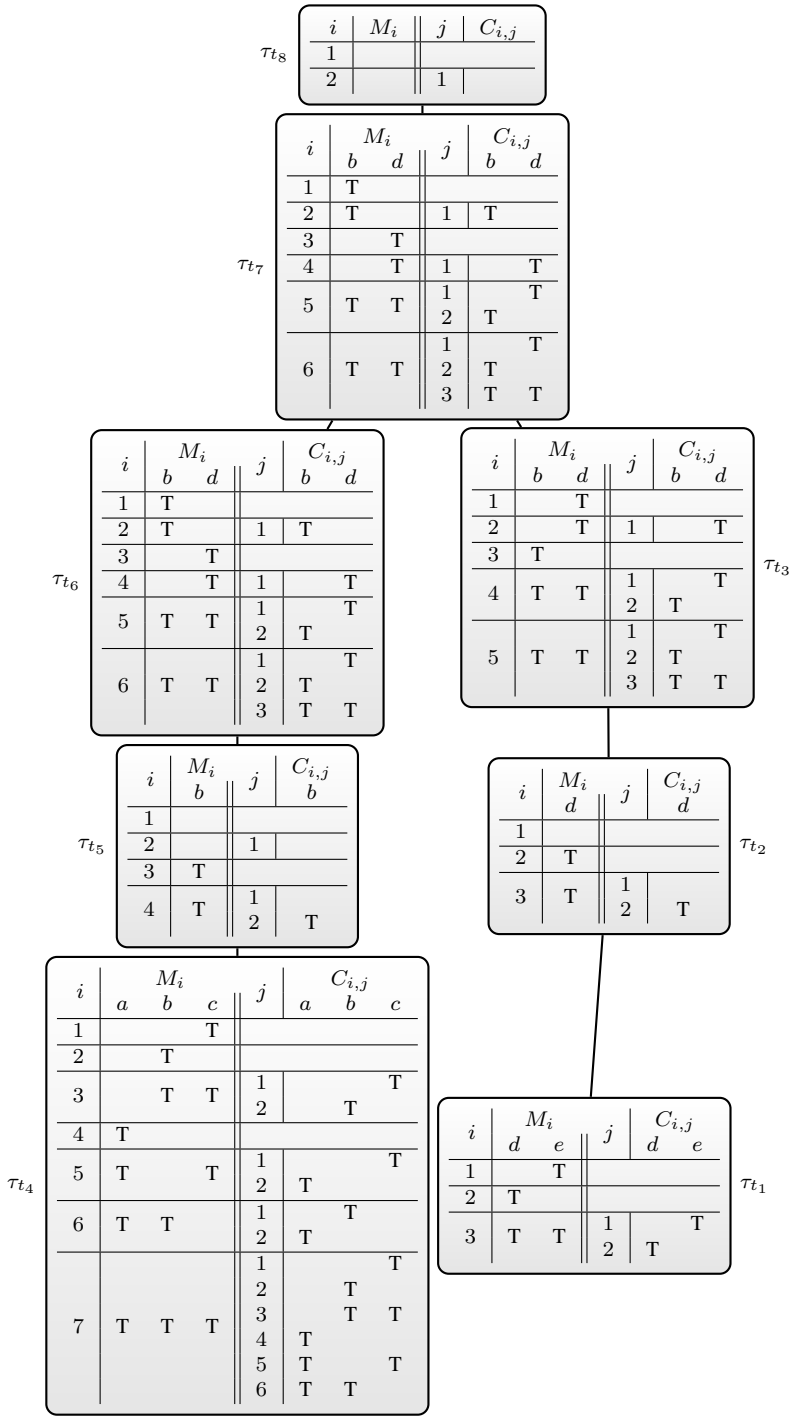
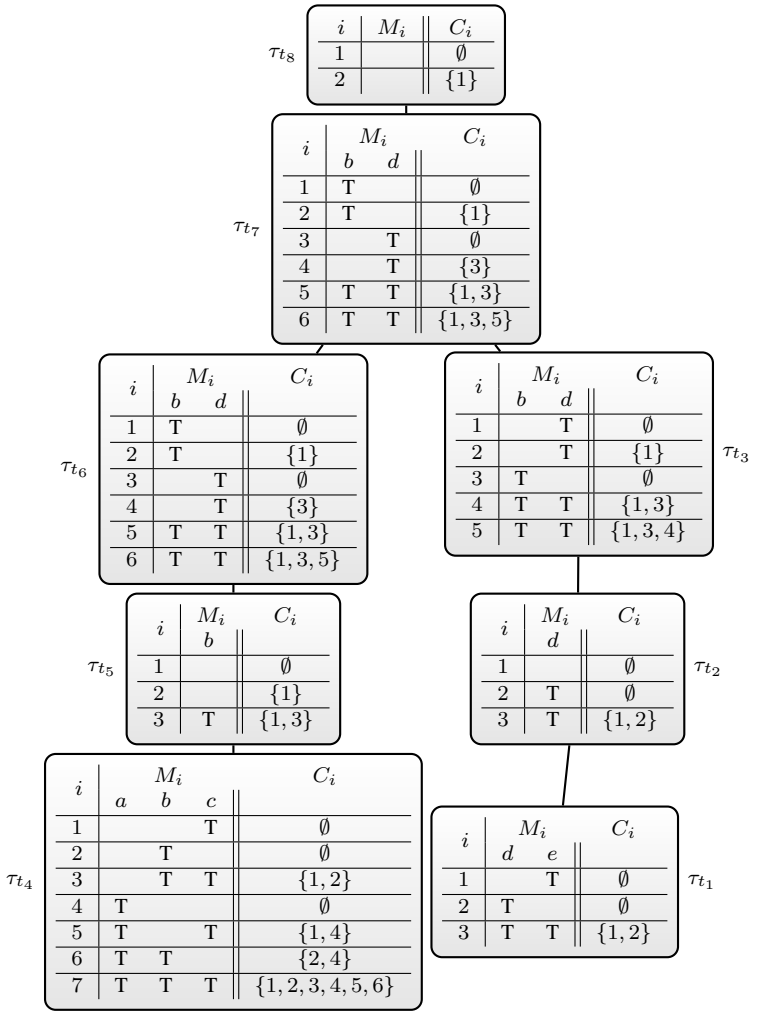


Figure 7 Dynamic programming tables for the nodes of decomposition $\mathcal{T}$ of Figure 6d using PRIM with the DynASP 2 approach.



i	M_i	C_i
1		$\emptyset$
2		$\{1\}$

i	M_i b	M_i d	C_i
1	T		$\emptyset$
2	T		$\{1\}$
3		T	$\emptyset$
4		T	$\{3\}$
5	T	T	$\{1,3\}$
6	T	T	$\{1,3,5\}$

i	M_i b	M_i d	C_i
1	T		$\emptyset$
2	T		$\{1\}$
3		T	$\emptyset$
4		T	$\{3\}$
5	T	T	$\{1,3\}$
6	T	T	$\{1,3,5\}$

i	M_i b	M_i d	C_i
1		T	$\emptyset$
2		T	$\{1\}$
3	T		$\emptyset$
4	T	T	$\{1,3\}$
5	T	T	$\{1,3,4\}$

i	M_i b	C_i
1		$\emptyset$
2		$\{1\}$
3	T	$\{1,3\}$

i	M_i d	C_i
1		$\emptyset$
2	T	$\emptyset$
3	T	$\{1,2\}$

i	M_i a	M_i b	M_i c	C_i
1			T	$\emptyset$
2		T		$\emptyset$
3		T	T	$\{1,2\}$
4	T			$\emptyset$
5	T		T	$\{1,4\}$
6	T	T		$\{2,4\}$
7	T	T	T	$\{1,2,3,4,5,6\}$

i	M_i d	M_i e	C_i
1		T	$\emptyset$
2	T		$\emptyset$
3	T	T	$\{1,2\}$

Figure 8 Dynamic programming tables for the nodes of decomposition $\mathcal{T}$ of Figure 6d using PRIM with the DynASP 2.5 approach.

A slightly modified implementation of the algorithm PRIM is the INVPRIM algorithm, where we store in each tuple the set $\overline{\mathcal{C}}$ of "inverse" local counterwitnesses, instead of the set $\mathcal{C}$ of local counterwitnesses, in order to see whether this leads to a speed-up. More precisely, for a set M, a program Π, and a tree decomposition $\mathcal{T}$ of $P(\Pi)$, the set $\mathcal{C}$ consists of sets $C \subsetneq M$ such that C is a model of $(R_t)^M$. However, the set $\overline{\mathcal{C}}$ consists of sets $\overline{C} \subsetneq M$ such that $\overline{C}$ is not a model of $(R_t)^M$. Hence, we have $\overline{\mathcal{C}} = 2^M \setminus \mathcal{C}$. Since the adaption of the conditions and results in Algorithm 1 to the set $\overline{\mathcal{C}}$ of inverse local counterwitnesses is straight forward, we omit full definitions here.

3.3.2 Algorithm SINC

Our SINC algorithm is based on the semi-incidence graph of a given program, defined as follows and its decision version (restricted to disjunctive programs) restates an algorithm by Jakl, Pichler, and Woltran [61].

Definition 3.8. *Let Π be a program. The* semi-incidence graph $S(\Pi)$ *is the graph where the set V consists of a vertex v_a for every atom $a \in \mathrm{at}(\Pi)$ 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 Π, an edge $v_a v_b$ if atoms a, b appear together in a choice head or weighted body of a rule in Π. For an atom a and 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.*

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.

Observation 3.9. *Let Π be a program. The treewidth of the semi-incidence graph is at most the treewidth of its primal graph plus 1.*

In order to describe algorithms based on the semi-incidence graph, we fix the following notation. Again, for a tree decomposition $\mathcal{T} = (T, \chi)$ where $T = (N, \cdot, n)$ of the semi-incidence graph of a program Π and a node $t \in N$, we let A_t contain all atoms $\mathrm{at}(\Pi)$ that have a corresponding vertex in $\chi(t)$ and R_t contain all rules $r \in \Pi$ that have a corresponding vertex in $\chi(t)$. The *local program* Π_t *in* t consists of all rules $r \in \Pi \setminus R_t$ such that $r \in R_{t'}$ for some $t' \in N'$ of the induced subtree $T' = (N', \cdot)$ of T rooted at t. We might also omit t if it is clear from the context. It is also easy to see that $\Pi_n = \Pi$ for the empty root n.

Let $\mathcal{T}$ be a normalized tree decomposition of the semi-incidence graph $S(\Pi)$ of Π. We further 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 or removal node, respectively, where the bag of node and its child differ in a corresponding atom $a \in \mathrm{at}(\Pi)$. 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 $r \in \Pi$. We denote the set of all atom introduce nodes of $\mathcal{T}$ by $\mathrm{AI}(\mathcal{T})$, the set of all atom removal nodes of $\mathcal{T}$ by $\mathrm{AR}(\mathcal{T})$, the set of all join nodes of $\mathcal{T}$ by $\mathrm{JOIN}(\mathcal{T})$, the set of all leaf nodes of $\mathcal{T}$ by $\mathrm{LEAF}(\mathcal{T})$, the set of all rule introduce nodes of $\mathcal{T}$ by $\mathrm{RI}(\mathcal{T})$, and the set of all rule removal nodes of $\mathcal{T}$ by $\mathrm{RR}(\mathcal{T})$.

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 heads and weighted bodies). Hence, we additionally need to keep track of already satisfied rules by local witnesses and counterwitnesses, respectively. Therefore, we need a "local view" on the program depending on the node t and we define the following notation.

Definition 3.10. *Let Π be a program, $\mathcal{T}$ be a tree decomposition of the semi-incidence graph of Π, t a node of $\mathcal{T}$, and $\mathcal{P} \subseteq R_t$ be a subprogram of R_t. The* (semi-incidence) bag program $\mathcal{P}^t$ *in node t is obtained from $\mathcal{P} \cup \{\leftarrow B(r) : H(r) \not\subseteq A_t, r \in \mathrm{CH}(\mathcal{P})\}$ by*

1. *removing from the heads and bodies of every remaining choice and disjunctive rules all literals $a, \neg a$ with $a \in \mathrm{at}(\Pi) \setminus A_t$; and*
2. *removing from the heads and bodies of every remaining weight rule r all literals $a, \neg a$ with $a \in \mathrm{at}(\Pi) \setminus A_t$ and set the bound $w - \min\{\, w, \mathrm{wght}(r, (B^-(r) \cup B^+(r)) \cap (\mathrm{at}(\Pi) \setminus A_t))\,\}$.*

We denote the (semi-incidence) local bag program $\mathcal{P}^t = R_t^t$ where $\mathcal{P} = R_t$ by $\hat{R}_t$.

Let Π be a program, $r \in \Pi$, t a node, and $M \subseteq A_t$. Then, a tuple in table τ_t for node t of the tree decomposition is a triple $\langle M, S, \mathcal{C}\rangle$ where M again witnesses the existence of a model $M' \supseteq M$ of the local program Π_t (and is therefore referred to as *local [semi-incidence] witness*) and the additional set S consists of all rules that correspond to vertices in $\chi(t)$ and are satisfied by some superset of M (i.e., in the end there exists $M'' \supseteq M$ with $M'' \vDash (\Pi_t \cup S)$). The set $\mathcal{C}$ consists of pairs (C, R) of sets of atoms and rules, where C represents a counterwitness to M being an answer set (i.e., C witnesses the existence of $C' \supseteq C$ satisfying $C' \vDash \Pi_t^{M''}$ with $C' \subsetneq M''$ and is called *local [semi-incidence] counterwitness*) and R consists of all rules that correspond to vertices in $\chi(t)$ and are satisfied in the GL reduct under M by a superset of C. Thus, in the end C witnesses the existence of $C'' \supseteq C$ satisfying $C'' \vDash (\Pi_t \cup R)^{M''}$ with $C'' \subsetneq M''$. The algorithm guarantees that a surviving tuple $\langle M, S, \emptyset\rangle$ witnesses that the set M can be extended to a model M'' satisfying the program $(\Pi_t \cup S)$ in 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 contains the tuple $\langle\emptyset, \emptyset, \emptyset\rangle$.

The SINC algorithm[6] is given in Algorithm 2. Let $\mathcal{T}$ be a tree decomposition of the semi-incidence 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 vertices that appear together in at least one bag. For an atom introduction node $t \in \mathrm{AI}(\mathcal{T})$, tuples of child nodes are computed very similar as before. Every local witness 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 bag program $\Pi^{(t,S)}$ that are satisfied by M or $M \cup \{a\}$, respectively. Again, all subsets of M or $M \cup \{a\}$, respectively, are considered as local counterwitnesses. Further, the set R of the child node t' for a local counterwitness is extended by each rule r of the GL reduct of the bag program $\Pi^{(t,S)}$ under M such that r is satisfied by M or $M \cup \{a\}$, respectively. For a rule removal node $t \in \mathrm{RR}(\mathcal{T})$ 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 $t \in \mathrm{JOIN}(\mathcal{T})$, we ensure that tuples of the left and right children are merged that agree on the sets M.

[6]For set S and element e we write $S_e^+ := S \cup \{e\}$ and $S_e^- := S \setminus \{e\}$.

Algorithm 2: SINC Algorithm.

Data: Program Π, Node t and its children t' and t'', if exist(s), of the fixed tree decomposition $\mathcal{T}$ of $S(\Pi)$.
Result: Modified table τ_t for node t.

if $t \in \mathrm{LEAF}(\mathcal{T})$ **then**
  $\tau_t := \left\{ \left\langle \emptyset, \emptyset, \emptyset \right\rangle \right\}$
if $t \in \mathrm{AI}(\mathcal{T})$ *and let* $a \in A_t \setminus A_{t'}$ **then**
  $\tau_t := \Big\{ \Big\langle M,\ S \cup \mathrm{SatRules}(\hat{R}_t, M),$
    $\{\langle C, R \cup \mathrm{SatRules}(\hat{R}_t^M, C)\rangle \mid \langle C, R\rangle \in \mathcal{C}\} \Big\rangle \quad \Big| \ \langle M, S, \mathcal{C}\rangle \in \tau_{t'} \Big\} \bigcup$
    $\Big\{ \Big\langle M_a^+,\ S \cup \mathrm{SatRules}(\hat{R}_t, M_a^+)\},$
    $\{\langle C, R \cup \mathrm{SatRules}(\hat{R}_t^{M_a^+}, C)\rangle \mid \langle C, R\rangle \in \mathcal{C}\} \cup$
    $\{\langle C_a^+, R \cup \mathrm{SatRules}(\hat{R}_t^{M_a^+}, C_a^+)\rangle \mid \langle C, R\rangle \in \mathcal{C}\} \cup$
    $\{\langle M, S \cup \mathrm{SatRules}(\hat{R}_t^{M_a^+}, M)\rangle\} \Big\rangle \quad \Big| \ \langle M, S, \mathcal{C}\rangle \in \tau_{t'} \Big\}$
if $t \in \mathrm{RI}(\mathcal{T})$ *and let* $r \in R_t \setminus R_{t'}$ **then**
  $\tau_t := \Big\{ \Big\langle M,\ S \cup \mathrm{SatRules}(\{r\}^t, M),$
    $\{\langle C, R \cup \mathrm{SatRules}([\{r\}^t]^M, C)\rangle \mid \langle C, R\rangle \in \mathcal{C}\} \Big\rangle \quad \Big| \ \langle M, S, \mathcal{C}\rangle \in \tau_{t'} \Big\}$
if $t \in \mathrm{AR}(\mathcal{T})$ *and let* $a \in A_{t'} \setminus A_t$ **then**
  $\tau_t := \Big\{ \Big\langle M_a^-,\ S, \{\langle C_a^-, R\rangle \mid \langle C, R\rangle \in \mathcal{C}\} \Big\rangle \quad \Big| \ \langle M, S, \mathcal{C}\rangle \in \tau_{t'} \Big\}$
if $t \in \mathrm{RR}(\mathcal{T})$ *and let* $r \in R_{t'} \setminus R_t$ **then**
  $\tau_t := \Big\{ \Big\langle M,\ S_r^-, \{\langle C, R_r^-\rangle \mid \langle C, R\rangle \in \mathcal{C}, r \in R\} \Big\rangle \quad \Big| \ \langle M, S, \mathcal{C}\rangle \in \tau_{t'}, r \in S \Big\}$
if $t \in \mathrm{JOIN}(\mathcal{T})$ **then**
  $\tau_t := \Big\{ \Big\langle M,\ S' \cup S'', \{\langle C, R' \cup R''\rangle \mid \langle C, R'\rangle \in \mathcal{C}', \langle C, R''\rangle \in \mathcal{C}''\} \cup$
    $\{\langle M, R \cup S''\rangle \mid \langle M, R\rangle \in \mathcal{C}'\} \cup$
    $\{\langle M, R \cup S'\rangle \mid \langle M, R\rangle \in \mathcal{C}''\} \Big\rangle \quad \Big| \ \langle M, S', \mathcal{C}'\rangle \in \tau_{t'}, \langle M, S'', \mathcal{C}''\rangle \in \tau_{t''} \Big\}$

Remark 3.11. *We would like to mention that choice rules are conceptually tricky to handle. In some sense these rules conditionally (depending on the evaluation of the body) break the subset minimality. Due to the nature of dynamic programming on tree decompositions, the information concerning choice rules is not evident in every tree decomposition node, in fact, it might turn out quite late for some rules during the bottom-up evaluation.*

Example 3.12. *To be more concrete, look at Figure 9 containing a simple choice rule r, its semi-incidence graph $S(\Pi)$, a corresponding tree decomposition and the computation obtained after bottom-up traversing the tree decomposition, which we will now focus on.*

Let us start with table t_1. Note that although a is part of a choice rule, line 2 of the table τ_{t_1} for node t_1 containing $M_2 = \{a\}$ still requires a local counterwitness $C_{2,1} = \emptyset$ (i.e., a is set to false) since $\emptyset \subsetneq \{a\}$. According to Definition 3.10, the bag program under M_2 is

$\hat{R}^{M_2}_{t_1} = \{\{a\} \leftarrow\} \cup \{a\} \cup \{\leftarrow\}$ which is unsatisfiable and therefore $r \notin R_{2,1}$. However, keep in mind that storing local counterwitness $C_{2,1}$ is mandatory since we do not know anything about the truth value of c.

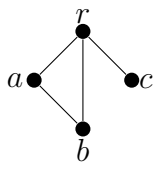
When we progress to the table τ_{t_2} for node t_2, we observe that r is satisfied in all the local witnesses, which is because of the corresponding bag program(s). To be more concrete, the first condition of Definition 3.10 does not include additional rules for choice rule r since $H(r) \subseteq A_{t_2}$ (remember that a node containing all the head atoms has to exist by construction of $S(\Pi)$) and moreover, a choice rule is vacuously satisfied. Note that none of the local counterwitnesses satisfy the corresponding bag reduct of r since the local counterwitnesses form a strict inclusion with respect to the corresponding local witness. For example, take $M_4 = \{a, b\}$ and $C_{4,2} = \{a\}$. We observe that $C_{4,2} \subsetneq M_4$ and $C_{4,2} \not\models \hat{R}^{M_4}_{t_2}$ where $\hat{R}^{M_4}_{t_2} = \{\{a; b\} \leftarrow\} \cup \{a \leftarrow\} \cup \{b \leftarrow\}$. Mind that for different tree decompositions there might occur situations involving choice rules r' and some node t' where $C_{i,j} \cap H(r) = M_i \cap H(r)$ in table $\tau_{t'}$ and therefore $r' \in R_{i,j}$ since $C_{i,j} \models \hat{R}^{M_i}_{t'}$.

We let the reader think about node t_3 and immediately jump to the more interesting node t_4. Observe that local counterwitnesses of table τ_{t_3} do not include c, i.e., $c \notin C_{i,j}$ and therefore vacuously satisfy the bag corresponding reducts $\{\{\} \leftarrow c\} \cup \{\leftarrow c\}\}$. So, in the end it turned out that for these local counterwitnesses, the corresponding local counterwitnesses of table τ_{t_1} were mandatory, since none of them includes c leaving r satisfied in these cases. Mind again that for different input programs there might occur tree decomposition nodes during the bottom-up traversal which introduce some rule(s) enforcing that c is in every answer set.

We end up with the empty answer set consisting of local witnesses M_1 for nodes t_5 to t_1.

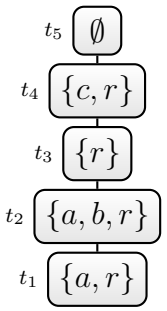
Example 3.13. *Consider again the problem* VC *and the graph $G = (V, E)$ from Figure 10a. Remember that the program $\{\, in_x \vee in_y : xy \in E \,\}$ encodes the problem* VC *in ASP given in Figure 10b. Further, Figure 10c illustrates the semi-incidence graph $S(\Pi)$ and Figure 10d visualizes a decomposition of the semi-incidence graph $S(\Pi)$.*

In Figure 11, we present the result of the SINC *algorithm on the decomposition of Figure 10d using DynASP 2. Figure 12 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.*



$\{a; b\} \leftarrow c.$

(a) Program $\Pi = \{r\}$ with semi-incidence graph $S(\Pi)$



(b) A tree decomposition $\mathcal{T}$ of $S(\Pi)$

τ_{t_5}

i	M_i	S_i	j	$C_{i,j}$	$R_{i,j}$
1					
2			1		

τ_{t_4}

i	M_i c	S_i r	j	$C_{i,j}$ c	$R_{i,j}$ r
1		sat			
2		sat	1		sat
3	T	sat	1		sat

τ_{t_3}

i	M_i	S_i r	j	$C_{i,j}$	$R_{i,j}$ r
1		sat			
2		sat	1		

τ_{t_2}

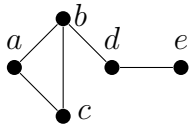
i	M_i a	M_i b	S_i r	j	$C_{i,j}$ a	$C_{i,j}$ b	$R_{i,j}$ r
1			sat				
2		T	sat	1			
3	T		sat	1			
4	T	T	sat	1 2 3	T	T	

τ_{t_1}

i	M_i a	S_i r	j	$C_{i,j}$ a	$R_{i,j}$ r
1					
2	T		1		

(c) Dynamic programming tables for $\mathcal{T}$

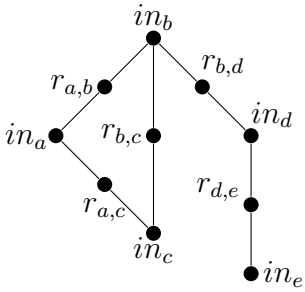
Figure 9 Dynamic programming tables for choice rule r using SINC (DynASP 2 approach).



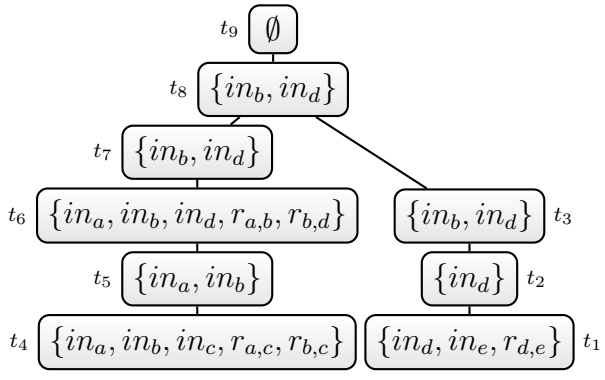
(a) A graph G

$in_a \vee in_b.\ in_a \vee in_c.\ in_b \vee in_c.\ in_b \vee in_d.$
$in_d \vee in_e.$

(b) Program Π which encodes the problem VC for G



(c) Semi-incidence graph $S(\Pi)$



(d) A tree decomposition $\mathcal{T}$ of $P(\Pi)$

Figure 10 Example graph and tree decomposition for the problem VC.

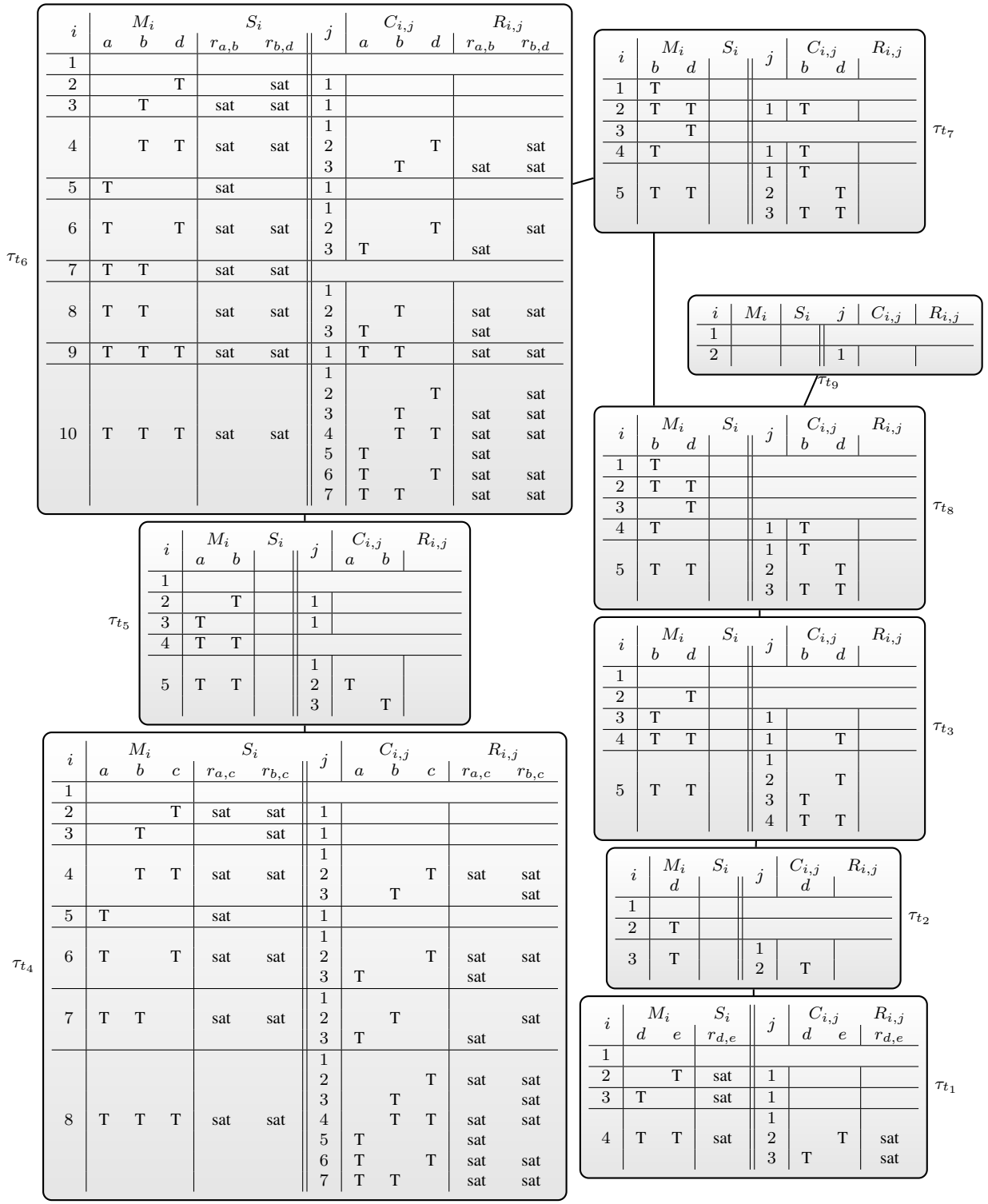
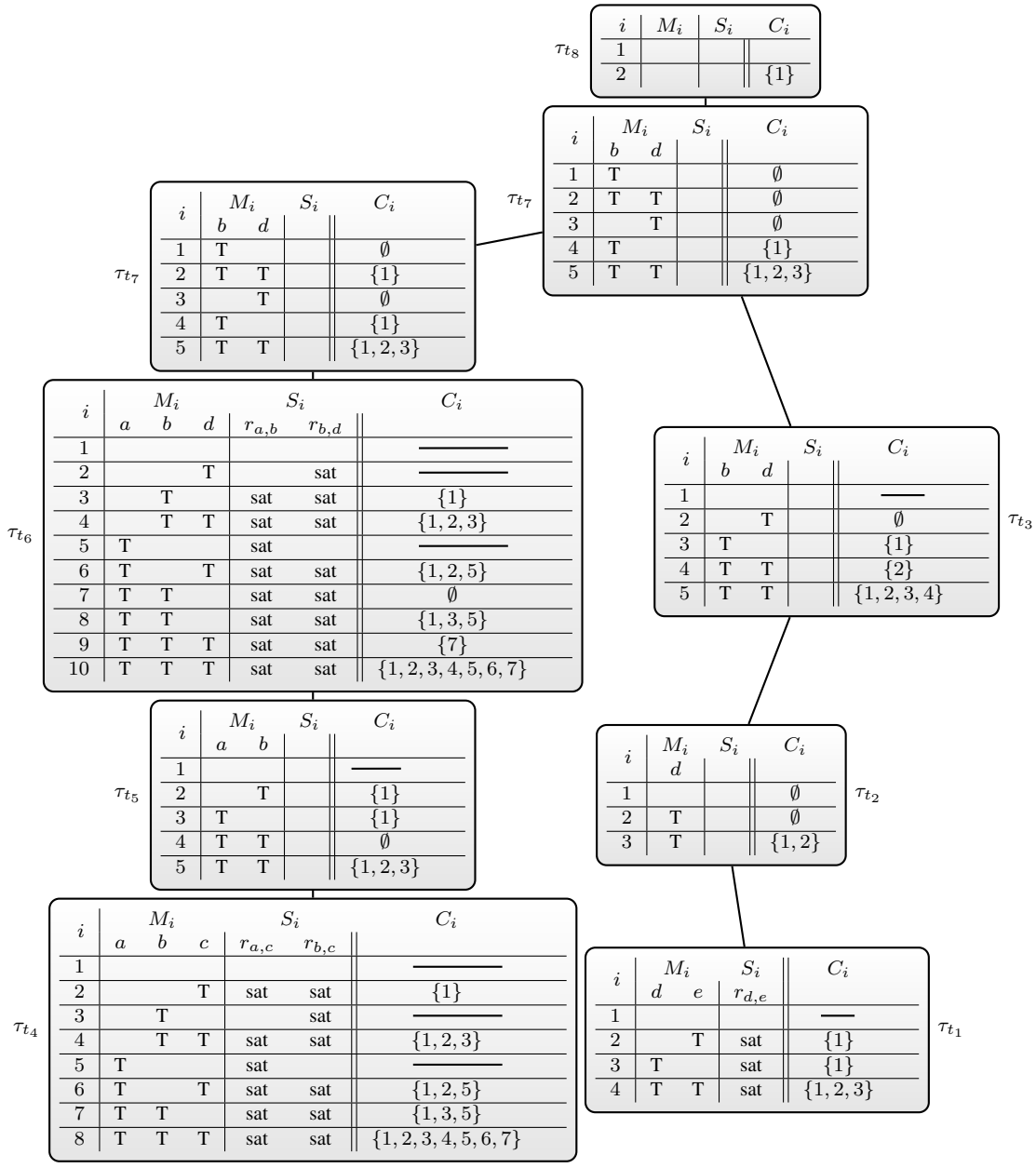


Figure 11 Dynamic programming tables for the nodes of decomposition $\mathcal{T}$ of Figure 10 using SINC with the DynASP 2 approach.



τ_{t_8}

i	M_i	S_i	C_i
1			
2			$\{1\}$

τ_{t_7}

i	M_i b	M_i d	S_i	C_i
1	T			$\emptyset$
2	T	T		$\emptyset$
3		T		$\emptyset$
4	T			$\{1\}$
5	T	T		$\{1,2,3\}$

τ_{t_7}

i	M_i b	M_i d	S_i	C_i
1	T			$\emptyset$
2	T	T		$\{1\}$
3		T		$\emptyset$
4	T			$\{1\}$
5	T	T		$\{1,2,3\}$

τ_{t_6}

i	M_i a	M_i b	M_i d	S_i $r_{a,b}$	S_i $r_{b,d}$	C_i
1						—
2			T		sat	—
3		T		sat	sat	$\{1\}$
4		T	T	sat	sat	$\{1,2,3\}$
5	T			sat		—
6	T		T	sat	sat	$\{1,2,5\}$
7	T	T		sat	sat	$\emptyset$
8	T	T		sat	sat	$\{1,3,5\}$
9	T	T	T	sat	sat	$\{7\}$
10	T	T	T	sat	sat	$\{1,2,3,4,5,6,7\}$

τ_{t_3}

i	M_i b	M_i d	S_i	C_i
1				—
2		T		$\emptyset$
3	T			$\{1\}$
4	T	T		$\{2\}$
5	T	T		$\{1,2,3,4\}$

τ_{t_5}

i	M_i a	M_i b	S_i	C_i
1				—
2		T		$\{1\}$
3	T			$\{1\}$
4	T	T		$\emptyset$
5	T	T		$\{1,2,3\}$

τ_{t_2}

i	M_i d	S_i	C_i
1			$\emptyset$
2	T		$\emptyset$
3	T		$\{1,2\}$

τ_{t_4}

i	M_i a	M_i b	M_i c	S_i $r_{a,c}$	S_i $r_{b,c}$	C_i
1						—
2			T	sat	sat	$\{1\}$
3		T			sat	—
4		T	T	sat	sat	$\{1,2,3\}$
5	T			sat		—
6	T		T	sat	sat	$\{1,2,5\}$
7	T	T		sat	sat	$\{1,3,5\}$
8	T	T	T	sat	sat	$\{1,2,3,4,5,6,7\}$

τ_{t_1}

i	M_i d	M_i e	S_i $r_{d,e}$	C_i
1				—
2		T	sat	$\{1\}$
3	T		sat	$\{1\}$
4	T	T	sat	$\{1,2,3\}$

Figure 12 Dynamic programming tables for the nodes of decomposition $\mathcal{T}$ of Figure 10 using SINC with the DynASP 2.5 approach.

3.3.3 Algorithm INC

The INC algorithm is based on the incidence graph of a given program. However, before we start with its description, we require the forthcoming definition of the incidence graph. The incidence graph is similar to the semi-incidence graph, but corresponding vertices of atoms appearing in choice or weight rules do not form a clique.

Definition 3.14. *Let Π be a program. The* incidence graph $I(\Pi)$ *of Π 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 and 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.*

In the following, before describing algorithms based on the incidence graph, we fix notation. Analogously to Section 3.3.2, for a tree decomposition $\mathcal{T} = (T, \chi)$ (where $T = (N, \cdot, \cdot)$) of the incidence graph of a program Π and a node $t \in N$, we denote by A_t the set of atoms in t, R_t denotes the set of rules in t and Π_t refers to the *local program* in t. Similarly, the *local atoms* $\text{at}^t(\Pi)$ *in* t consist of all atoms $a \in \text{at}(\Pi)$ such that $a \in A_{t'}$ for some $t' \in N'$ of the induced subtree $T' = (N', \cdot)$ of T rooted at t. Note that as already introduced in Section 3.3.2, for a tree decomposition $\mathcal{T}$, we refer to the sets $\text{AI}(\mathcal{T})$, $\text{AR}(\mathcal{T})$, $\text{JOIN}(\mathcal{T})$, $\text{LEAF}(\mathcal{T})$, $\text{RI}(\mathcal{T})$ and $\text{RR}(\mathcal{T})$.

The intuition is similar to the one of SINC. However, since the atoms that occur in choice heads or weighted bodies might not have corresponding vertices that occur together in one bag, we require in addition to SINC to remember *satisfaction states* for rules, which are not yet known to be satisfied.

A tuple for the INC algorithm in the table τ_t for a node t of the tree decomposition is a triple $\langle M, \sigma, \mathcal{C} \rangle$, where again M witnesses the existence of a model $M' \supseteq M$ of the local program Π_t (and is therefore referred to as *local [incidence] witness*). The *satisfaction state* σ for M maps from rules R_t to $\mathbb{N}_0 \cup \{+\infty\}$ and represents both rules r that are satisfied in the bag program ($\sigma(r) = +\infty$) and rules r', which are not yet satisfied ($\sigma(r') \neq +\infty$). Rules r with a vertex of the current bag $\chi(t)$ and $\sigma(r) = +\infty$ are satisfied by some superset of M, i.e., in the end there exists $M'' \supseteq M$ with $M'' \models (\Pi_t \cup \{r : r \in R_t, \sigma(r) = +\infty\})$. Intuitively, a satisfaction state $\sigma(r') \neq +\infty$ for a local witness M and rule r' represents for a weight rule the sum of weights of literals in the weight rule that have been removed (in some descendant rule removal node) and been taken or not taken, respectively, into the (local) witness in the table of some descendant node. We can think of this "satisfaction state" as an aiding tool that incorporates information from descendant nodes for weight rules. The set $\mathcal{C}$ consists of tuples (C, ρ) where C again represents a counterwitness to M being an answer set (i.e., C witnesses the existence of $C' \supseteq C$ satisfying $C' \models \Pi_t^{M''}$ with $C' \subsetneq M''$ and is called *local [incidence] counterwitness*). Again, ρ is referred to as *satisfaction state* for C under M and maps from rules R_t to $\mathbb{N}_0 \cup \{+\infty\}$ and represents both rules r that are satisfied in the bag reduct under M ($\rho(r) = +\infty$) and rules r', which are not yet satisfied in said reduct ($\rho(r') \neq +\infty$). Rules r with $\sigma(r) = +\infty$ are satisfied in the GL reduct under M by a superset of C. Thus, in the end again C witnesses the existence of $C'' \supseteq C$ satisfying $C'' \models (\Pi_t \cup \{r : r \in R_t, \rho(r) = +\infty\})^{M''}$ with $C'' \subsetneq M''$. Intuitively, a satisfaction state $\rho(r') \neq +\infty$ for local counterwitness C under M and rule r' represents for a choice rule

the number of atoms in the choice head that have been removed (in some descendant rule remove nodes) and taken into the (local) witness, but not into the corresponding (local) counterwitness 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 removed (in some descendant rule removal nodes) and been taken or not taken, respectively, into the (local) counterwitness in the table of some descendant node. The algorithm guarantees that a tuple $\langle M, \sigma, \mathcal{C}\rangle$ in τ_t for node t witnesses that the set M can be extended to a model M'' satisfying the program $(\Pi_t \cup \{r : r \in R_t, \sigma(r) = +\infty\})$. Again, there exists an answer set of the input program Π if the (empty) root contains tuple $\langle \emptyset, \sigma_\emptyset, \emptyset\rangle$[7].

Definition 3.15. *Let Π be a program, $\mathcal{T}$ be a tree decomposition of the incidence graph of Π, t and t' nodes of $\mathcal{T}$, $\mathcal{P} \subseteq R_t$ be a subprogram of R_t, and $\sigma_{t'} : R_{t'} \to \mathbb{N}_0 \cup \{+\infty\}$ a mapping. The* (incidence) bag program *$\mathcal{P}^{(t,\sigma_{t'})}$ is obtained from $\mathcal{P} \cup \{\leftarrow\ B(r) : |H(r)| > |\text{at}^t(\Pi) \cap H(r)| - \sigma_{t'}(r), r \in \text{CH}(\mathcal{P})\}$ by*

1. *removing from the heads and bodies of every choice and disjunctive rule all literals $a, \neg a$ with $a \in \text{at}(\Pi) \setminus A_t$; and*
2. *removing from the heads and bodies of every weight rule r all literals $a, \neg a$ with $a \in \text{at}(\Pi) \setminus A_t$ and set the bound $w - \min\{\, w, \sigma_{t'}(r) + \text{wght}(r, (B^-(r) \cup B^+(r)) \cap [A_t \cup (\text{at}(\Pi) \setminus \text{at}^t(\Pi))])\,\}$.*

If $\mathcal{P} = R_t$, then we denote the (incidence) local bag program $\mathcal{P}^{(t,\sigma)} = R_t^{(t,\sigma)}$ by R_t^σ and the (incidence) local bag reduct $[\mathcal{P}^{(t,\sigma)}]^M = [R_t^{(t,\sigma)}]^M$ by $R_t^{(\sigma,M)}$.

We define the following notation for our algorithm. For mappings σ and σ', we intuitively require an operator $(\sigma \uplus \sigma')$ designed to add the function values if the argument belongs to both domains, and to otherwise take the function value where the argument belongs to either one domain[8]. Moreover, for $R \subseteq \sigma^{-1}$, we define σ_R^- as for every $x \in \sigma^{-1} \setminus R$ we have $\sigma_R^-(x) := \sigma(x)$. This operation potentially shrinks the domain of σ.

Let Π be a program, t be a node of some tree decomposition $\mathcal{T}$ of $I(\Pi)$ and $M, C \subseteq A_t$. Then, for $r \in R_t$, we define $\text{SatStates}(\Pi, M) := \sigma$ where $\sigma(r) := +\infty$ if $r \in \text{SatRules}(\Pi, M)$, and $\sigma(r) := 0$ otherwise. Further, we let $\text{StateValues}(R_t, M, C) := \sigma$ where

$$\sigma(r) := \begin{cases} \text{wght}(r, \{a\} \cap [(B^-(r) \setminus M) \cup (B^+(r) \cap C)]), & \text{if } r \in \text{WGT}(R_t) \\ |\{a\} \cap H(r) \cap (M \setminus C)|, & \text{if } r \in \text{CH}(R_t) \\ 0, & \text{if } r \in \text{DISJ}(R_t) \end{cases} \quad \text{for every } r \in R_t.$$

The INC algorithm is given in Algorithm 3.

[7] We denote by $\sigma_\emptyset : \emptyset \to \mathbb{N}_0 \cup \{+\infty\}$ an empty satisfaction state.

[8] More formally, using mappings σ and σ', we define $(\sigma \uplus \sigma')(x)$ for every $x \in (\sigma^{-1} \cup \sigma'^{-1})$ where

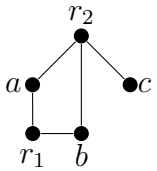
$$(\sigma \uplus \sigma')(x) := \begin{cases} \sigma(x) + \sigma'(x), & \text{if } x \in \sigma^{-1} \text{ and } x \in \sigma'^{-1} \\ \sigma(x), & \text{if } x \in \sigma^{-1} \text{ and } x \notin \sigma'^{-1} \\ \sigma'(x), & \text{if } x \notin \sigma^{-1} \text{ and } x \in \sigma'^{-1} \end{cases}$$

Algorithm 3: INC Algorithm.

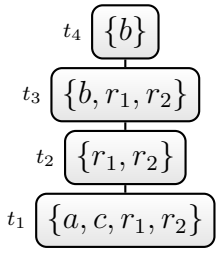
Data: Program Π, Node t and its children t' and t'', if exist(s), of the fixed tree decomposition $\mathcal{T}$ of $I(\Pi)$.
Result: Modified table τ_t for node t.

if $t \in \mathrm{LEAF}(\mathcal{T})$ **then**
  $\tau_t := \left\{ \left\langle \emptyset, \sigma_\emptyset, \emptyset \right\rangle \right\}$
if $t \in \mathrm{AI}(\mathcal{T})$ *and let* $a \in A_t \setminus A_{t'}$ **then**
  $\tau_t := \Big\{ \Big\langle M, \sigma \uplus \mathrm{SatStates}(R_t^\sigma, M),$
    $\{\langle C, \rho \uplus \mathrm{SatStates}(R_t^{(\rho,M)}, C)\rangle \mid \langle C, \rho\rangle \in \mathcal{C}\} \Big\rangle \;\Big|\; \langle M, \sigma, \mathcal{C}\rangle \in \tau_{t'} \Big\} \bigcup$
    $\Big\{ \Big\langle M_a^+, \sigma \uplus \mathrm{SatStates}(R_t^\sigma, M_a^+)\},$
    $\{\langle C, \rho \uplus \mathrm{SatStates}(R_t^{(\rho,M_a^+)}, C)\rangle \mid \langle C, \rho\rangle \in \mathcal{C}\} \cup$
    $\{\langle C_a^+, \rho \uplus \mathrm{SatStates}(R_t^{(\rho,M_a^+)}, C_a^+)\rangle \mid \langle C, \rho\rangle \in \mathcal{C}\} \cup$
    $\{\langle M, \sigma \uplus \mathrm{SatStates}(R_t^{(\sigma,M_a^+)}, M)\rangle\} \Big\rangle \;\Big|\; \langle M, \sigma, \mathcal{C}\rangle \in \tau_{t'} \Big\}$
if $t \in \mathrm{RI}(\mathcal{T})$ *and let* $r \in R_t \setminus R_{t'}$ **then**
  $\tau_t := \Big\{ \Big\langle M, \sigma \uplus \mathrm{SatStates}(\{r\}^{(t,\sigma)}, M),$
    $\{\langle C, \rho \uplus \mathrm{SatStates}([\{r\}^{(t,\rho)}]^M, C)\rangle \mid \langle C, \rho\rangle \in \mathcal{C}\} \Big\rangle \;\Big|\; \langle M, \sigma, \mathcal{C}\rangle \in \tau_{t'} \Big\}$
if $t \in \mathrm{AR}(\mathcal{T})$ *and let* $a \in A_{t'} \setminus A_t$ **then**
  $\tau_t := \Big\{ \Big\langle M_a^-, \sigma \uplus \mathrm{StateValues}(R_t, M, M),$
    $\{\langle C_a^-, \rho \uplus \mathrm{StateValues}(R_t, M, C)\rangle \mid \langle C, \rho\rangle \in \mathcal{C}\} \Big\rangle \;\Big|\; \langle M, \sigma, \mathcal{C}\rangle \in \tau_{t'} \Big\}$
if $t \in \mathrm{RR}(\mathcal{T})$ *and let* $r \in R_{t'} \setminus R_t$ **then**
  $\tau_t := \Big\{ \Big\langle M, \sigma_{\{r\}}^-, \{\langle C, \rho_{\{r\}}^-\rangle \mid \langle C, \rho\rangle \in \mathcal{C}, \rho(r) = +\infty\} \Big\rangle \;\Big|\; \langle M, \sigma, \mathcal{C}\rangle \in \tau_{t'}, \sigma(r) = +\infty \Big\}$
if $t \in \mathrm{JOIN}(\mathcal{T})$ **then**
  $\tau_t := \Big\{ \Big\langle M,\ \sigma' \uplus \sigma'', \{\langle C, \rho' \uplus \rho''\rangle \mid \langle C, \rho'\rangle \in \mathcal{C}', \langle C, \rho''\rangle \in \mathcal{C}''\} \cup$
    $\{\langle M, \rho \uplus \sigma''\rangle \mid \langle M, \rho\rangle \in \mathcal{C}'\} \cup$
    $\{\langle M, \sigma' \uplus \rho\rangle \mid \langle M, \rho\rangle \in \mathcal{C}''\} \Big\rangle \;\Big|\; \langle M, \sigma', \mathcal{C}'\rangle \in \tau_{t'}, \langle M, \sigma'', \mathcal{C}''\rangle \in \tau_{t''} \Big\}$

$r_1 : \{a; b\}.$
$r_2 : c \leftarrow 1 \leq \{b = 1, \neg a = 1\}.$



(a) Program $\Pi = \{r_1, r_2\}$ with incidence graph $I(\Pi)$



(b) A tree decomposition $\mathcal{T}$ of $I(\Pi)$

τ_{t_4}

i	M_i b	σ_i	j	$C_{i,j}$ b	$\rho_{i,j}$
1					
2	T				
3			1		

τ_{t_3}

i	M_i b	σ_i r_1	σ_i r_2	j	$C_{i,j}$ b	$\rho_{i,j}$ r_1	$\rho_{i,j}$ r_2
1		$+\infty$	1				
2	T	$+\infty$	1	1		0	1
3		$+\infty$	$+\infty$	1		$+\infty$	1
4	T	$+\infty$	$+\infty$	1		0	1
				2	T	$+\infty$	1
				3		0	$+\infty$
5		$+\infty$	$+\infty$	1		1	1
6	T	$+\infty$	0	1		1	1
				2	T	1	1
				3		0	$+\infty$
7		$+\infty$	$+\infty$	1		1	1
				2		1	$+\infty$
				3		$+\infty$	$+\infty$
8	T	$+\infty$	$+\infty$	1		1	1
				2	T	1	1
				3		1	$+\infty$
				4	T	1	$+\infty$
				5		1	$+\infty$
				6	T	$+\infty$	0

τ_{t_2}

i	M_i	σ_i r_1	σ_i r_2	j	$C_{i,j}$	$\rho_{i,j}$ r_1	$\rho_{i,j}$ r_2
1		0	1				
2		0	$+\infty$	1		0	1
3		0	0	1		1	1
4		0	$+\infty$	1		1	1
				2		1	$+\infty$
				3		0	0

τ_{t_1}

i	M_i a	M_i c	σ_i r_1	σ_i r_2	j	$C_{i,j}$ a	$C_{i,j}$ c	$\rho_{i,j}$ r_1	$\rho_{i,j}$ r_2
1			0	0					
2		T	0	$+\infty$	1			0	0
3	T		0	0	1			0	0
4	T	T	0	$+\infty$	1			0	0
					2		T	0	$+\infty$
					3	T		0	0

(c) Dynamic programming tables for $\mathcal{T}$

Figure 13 Dynamic programming tables for program Π using INC (DynASP 2 approach).

Example 3.16. *Look at Figure 13 containing two simple rules* $\Pi = \{r_1, r_2\}$*, its incidence graph* $I(\Pi)$*, a corresponding tree decomposition and the computation obtained after bottom-up traversing the tree decomposition. Note that the the tables for* $\mathcal{T}$ *encode the answer sets* $\{c\}, \{a\}, \{b, c\}, \{a, b, c\}$ *forming exactly the answer sets of* Π*.*

3.4 Correctness and Runtime

In the following, we provide insights on the correctness of our Algorithm 3 and reason later on that it is fixed-parameter tractable and provide a sufficient parameter.

3.4.1 Soundness and Completeness

The correctness proof of these algorithms is rather tedious, as each node type needs to be investigated separately. However, it has to be shown that a tuple at a node t guarantees that there exists a model for the local program in t, proving soundness. Conversely, it can be shown that each candidate answer set is indeed evaluated while traversing the tree decomposition, which proves completeness. In the following, we sketch this idea even further using both the notion of *partial solutions* consisting of *partial models* and the notion of *local partial solutions*. Note that this section is dedicated to the most demanding INC algorithm provided in Algorithm 3, since the other algorithms discussed in this paper can be shown by basic simplifications.

Definition 3.17. *Let Π be a program and $\mathcal{T} = (T, \chi)$ where $T = (N, \cdot, \cdot)$ be a tree decomposition of the incidence graph $I(\Pi)$. Furthermore, let $t \in N$ be a node. A* partial model *for t under M is a tuple (C, σ) where $M, C \subseteq \text{at}^t(\Pi)$ and $\sigma : (\Pi_t \cup R_t) \to \mathbb{N}_0 \cup \{+\infty\}$ such that*

1. *$C \vDash \Pi_t^M$,*
2. *$\sigma(r) = 0$ or $\sigma(r) = +\infty$ for any $r \in (\Pi_t \cup R_t)$,*
3. *$C \cap H(r) \neq \emptyset$ or $(M \cap B^-(r)) \neq \emptyset$ or $(B^+(r) \cap \text{at}^t(\Pi)) \not\subseteq C$ if $\sigma(r) = +\infty$ for any $r \in \text{DISJ}(\Pi_t \cup R_t)$,*
4. *$\text{bnd}(r) > \text{wght}(r, [(\text{at}(\Pi) \setminus \text{at}^t(\Pi)) \cap (B^+(r) \cup B^-(r))] \cup (B^+(r) \cap C) \cup (B^-(r) \setminus M))$ or $C \cap H(r) \neq \emptyset$ if $\sigma(r) = +\infty$ for any $r \in \text{WGT}(\Pi_t \cup R_t)$, and*
5. *$M \cap B^-(r) \neq \emptyset$ or $(B^+(r) \cap \text{at}^t(\Pi)) \not\subseteq C$ or both $H(r) \subseteq \text{at}^t(\Pi)$ and $(M \setminus C) \cap H(r) = \emptyset$ if $\sigma(r) = +\infty$ for any $r \in \text{CH}(\Pi_t \cup R_t)$.*

Definition 3.18. *Let Π be a program and $\mathcal{T} = (T, \chi)$ where $T = (N, \cdot, n)$ be a tree decomposition of the incidence graph $I(\Pi)$. Furthermore, let $t \in N$ be a node. A* partial solution *for t is a tuple $(M, \sigma, \mathcal{C})$ where (M, σ) is a partial model (under M) and $\mathcal{C}$ is a set of partial models (C, ρ) under M with $C \subsetneq M$.*

Note that without loss of generality we may assume that the root of our tree decompositions always has an empty bag, which can be achieved by introducing intermediate nodes. With the following proposition, one can see the correspondence between answer sets and partial solutions.

Proposition 3.19. *Let Π be a program and $\mathcal{T} = (T, \chi)$ where $T = (\cdot, \cdot, n)$ be a tree decomposition of the incidence graph $I(\Pi)$ and let $\chi(n) = \emptyset$. Then there exists an answer set M for Π if and only if there exists a partial solution $v = (M, \sigma, \emptyset)$ with $\sigma^{-1}(+\infty) = \Pi$ for node n.*

Proof. Given an answer set M of Π it is easy to construct $v = (M, \sigma, \emptyset)$ with $\sigma(r) := +\infty$ for $r \in \Pi$ and to check that v is indeed a partial solution for n (according to Definition 3.18). For the other direction, one can check (see again Definition 3.18) that indeed the existence of v suffices to guarantee that M is an answer set. □

In order to proof the correctness of our algorithm, we require the following notion of local partial solutions which correspond to the tuples obtained in Algorithm 3. The equivalence is obvious for leaf nodes $t \in \mathrm{LEAF}(\cdot)$ for the tree decomposition and will be shown in Theorem 3.23 proving soundness.

Definition 3.20. *Let Π be a program and $\mathcal{T} = (T, \chi)$ where $T = (N, \cdot, n)$ be a tree decomposition of the incidence graph $I(\Pi)$. Furthermore, let $t \in N$ be a node, $M, C \subseteq \mathrm{at}(\Pi)$ and $\sigma : \Pi \to \mathbb{N}_0 \cup \{+\infty\}$ be a mapping. We define the* local (satisfaction) state $\sigma^{t,M,C} := (\sigma \uplus \sigma')^{-}_{\Pi_t}$ *for C under M of node t where $\sigma' : R_t \to \mathbb{N}_0 \cup \{+\infty\}$ by*

1. $\sigma'(r) = \mathrm{wght}(r, (\mathrm{at}^t(\Pi) \setminus A_t) \cap [(B^-(r) \setminus M) \cup (B^+(r) \cap C)])$ *for any* $r \in \mathrm{WGT}(R_t)$,
2. $\sigma'(r) = |(\mathrm{at}^t(\Pi) \setminus A_t) \cap H(r) \cap (M \setminus C)|$ *for any* $r \in \mathrm{CH}(R_t)$, *and*
3. $\sigma'(r) = 0$ *for any* $r \in \mathrm{DISJ}(R_t)$.

Definition 3.21. *Let Π be a program and $\mathcal{T} = (T, \chi)$ where $T = (N, \cdot, n)$ be a tree decomposition of the incidence graph $I(\Pi)$. Furthermore, let $t \in N$ be a node. A* local (partial) solution *for t is a tuple $\langle M, \sigma, \mathcal{C} \rangle$ if there exists a partial solution $(\hat{M}, \hat{\sigma}, \hat{\mathcal{C}})$ such that*

1. $M = \hat{M} \cap A_t$,
2. $\sigma = \hat{\sigma}^{t,\hat{M},\hat{M}}$, *and*
3. $\mathcal{C} = \{\, \langle \hat{C} \cap A_t, \hat{\rho}^{t,\hat{M},\hat{C}} \rangle : (\hat{C}, \hat{\rho}) \in \hat{\mathcal{C}} \,\}$.

We denote the construction of the local partial solution $v = \langle M, \sigma, \mathcal{C} \rangle$ out of partial solution $\hat{v} = (\hat{M}, \hat{\sigma}, \hat{\mathcal{C}})$ by $v = \hat{v}^t$.

The following proposition now provides justification that indeed it suffices to store local partial solutions, i.e., we do not need to completely store partial solutions per node.

Proposition 3.22. *Let Π be a program and $\mathcal{T} = (T, \chi)$ where $T = (\cdot, \cdot, n)$ be a tree decomposition of the incidence graph $I(\Pi)$ and let $\chi(n) = \emptyset$. Then there exists an answer set M for Π if and only if there exists a local (partial) solution of the form $v = \langle M, \sigma_\emptyset, \emptyset \rangle$ for node n.*

Proof. Since $\chi(n) = \emptyset$, every partial solution for n is an extension of v according to Definition 3.21 and therefore the claim follows from Proposition 3.19. □

Next, we show our two main theorems of this section concluding the correctness result.

Theorem 3.23 (Soundness)**.** *Let Π be a program and $\mathcal{T} = (T, \chi)$ where $T = (N, \cdot, \cdot)$ be a tree decomposition of the incidence graph $I(\Pi)$. Furthermore, let $t \in N$ be a node and $t' \in N$ be its successor (or $t', t'' \in N$ be its successors, respectively). Given a local partial solution v' of table $\tau_{t'}$ (or local partial solutions v' of table $\tau_{t'}$ and v'' of table $\tau_{t''}$, respectively), each tuple v of table τ_t for node t constructed using* INC *algorithm specified in Algorithm 3 is also a local partial solution.*

Proof. Let v' be a local partial solution for $t' \in N$ and let v be a tuple for node $t \in N$ such that v was derived from v' using INC algorithm. Hence, t' is the only child of t and either $t \in \mathrm{AI}(\mathcal{T}), t \in \mathrm{AR}(\mathcal{T}), t \in \mathrm{RI}(\mathcal{T})$ or $t \in \mathrm{RR}(\mathcal{T})$. Assume that t is a rule removal node $t \in \mathrm{RR}(\mathcal{T})$ and $r \in R_{t'} \setminus R_t$ for some rule r. Note that $v = \langle M, \sigma, \mathcal{C} \rangle$ and $v' = \langle M', \sigma', \mathcal{C}' \rangle$ only differ in the satisfaction states (σ, σ') and the satisfaction states occurring within the set of local counterwitnesses ($\mathcal{C}, \mathcal{C}'$), since only r is removed during transition from t' to t. According to Algorithm 3 and since v has to be derived from v', we require that either r is a choice rule (trivially satisfied) or $\sigma'(r) = +\infty$. Similarly, we require for any $\langle C', \rho' \rangle \in \mathcal{C}'$ that $\rho'(r) = +\infty$ resulting in the fact that C' satisfies r under M'. Since v' is a local partial solution, there has to exist a partial solution $\hat{v}'$ of t', satisfying the conditions of Definition 3.21. We now claim that $\hat{v}'$ is also a partial solution for node t, which can be easily verified checking conditions of Definition 3.18. Finally, note that $v = \hat{v'}^t$ since the projection of $\hat{v}'$ to the bag $\chi(t)$ is v itself, which results in the conclusion that v is indeed a local partial solution. Analogously, the theorem can be checked for the three remaining node types above.

Now assume for the remaining case that t is a join node and therefore let v' and v'' be local partial solutions for $t', t'' \in N$ and v be a tuple for node $t \in N$ such that v can be derived using both v' and v'' in accordance with INC algorithm. Since v' and v'' are local partial solutions, there exist partial solutions $\hat{v}' = (\hat{M}', \hat{\sigma}', \hat{\mathcal{C}}')$ for node t' and $\hat{v}'' = (\hat{M}'', \hat{\sigma}'', \hat{\mathcal{C}}'')$ for node t''. Using these two partial solutions, we can construct $\hat{v} = (\hat{M}' \cup \hat{M}'', \hat{\sigma}' \uplus \hat{\sigma}'', \hat{\mathcal{C}}' \bowtie \hat{\mathcal{C}}'')$ where $\bowtie(\cdot, \cdot)$ is defined in accordance with Algorithm 3 as follows:

$$\begin{aligned}\hat{\mathcal{C}}' \bowtie \hat{\mathcal{C}}'' := {} & \{(\hat{C}' \cup \hat{C}'', \hat{\rho}' \uplus \hat{\rho}'') : (\hat{C}', \hat{\rho}') \in \hat{\mathcal{C}}', (\hat{C}'', \hat{\rho}'') \in \hat{\mathcal{C}}'', \hat{C}' \cap A_t = \hat{C}'' \cap A_t\} \cup \\ & \{(\hat{C}' \cup \hat{M}'', \hat{\rho}' \uplus \hat{\sigma}'') : (\hat{C}', \hat{\rho}') \in \hat{\mathcal{C}}', \hat{C}' \cap A_t = \hat{M}'' \cap A_t\} \cup \\ & \{(\hat{M}' \cup \hat{C}'', \hat{\sigma}' \uplus \hat{\rho}'') : (\hat{C}'', \hat{\rho}'') \in \hat{\mathcal{C}}'', \hat{M}' \cap A_t = \hat{C}'' \cap A_t\}\end{aligned}$$

Then, we check all conditions of Definition 3.18 in order to verify that $\hat{v}$ is a partial solution for t. Moreover, the projection $\hat{v}^t$ of $\hat{v}$ to the bag $\chi(t)$ is exactly v by construction and hence, $v = \hat{v}^t$ is a local partial solutions. □

Theorem 3.24 (Completeness)**.** *Let Π be a program and $\mathcal{T} = (T, \chi)$ where $T = (N, \cdot, \cdot)$ be a tree decomposition of the incidence graph $I(\Pi)$. Furthermore, let $t \in N$ be a node and $t' \in N$ be its successor (or $t', t'' \in N$ be its successors, respectively). Given a local partial solution v of table τ_t, either $t \in \mathrm{LEAF}(\mathcal{T})$ or there exists a local partial solution v' of table $\tau_{t'}$ (or two local partial solutions, namely v' of table $\tau_{t'}$ and v'' of table $\tau_{t''}$) such that v can be constructed by v' (or v' and v'', respectively) using* INC *algorithm specified in Algorithm 3.*

Proof. Again one can distinguish the node types of node t. Let $t \in N$ be a rule removal node $t \in \mathrm{RR}(\mathcal{T})$ and $r \in R_{t'} \setminus R_t$ with successor node $t' \in N$. We have to show that there exists a bag

model v' of table $\tau_{t'}$ for node t' such that v can be constructed using v' with the help of the INC algorithm specified in Algorithm 3. Since v is a local partial solution, there exists a partial solution $\hat{v} = (\hat{M}, \hat{\sigma}, \hat{\mathcal{C}})$ for node n, satisfying the conditions of Definition 3.21. Since r is the removed rule, $\hat{\sigma}(r) = +\infty$ or $r \in \mathrm{CH}(\Pi)$ is a choice rule by Definition 3.18. Hence, $\hat{M}$ satisfies r either way by similar arguments for any tuple $(\hat{C}, \hat{\rho}) \in \hat{\mathcal{C}}$, and we define $v' = \hat{v}^t$, which is, the projection of $\hat{v}$ onto the bag of t'. The resulting v' is a local partial solution for node t' according to Definition 3.21. But then it is easy to check, that v can be derived using INC algorithm and v'. Finally, the theorem can be checked for the remaining (five) node types. □

3.4.2 Running time

In the following we show a theoretical runtime bound for the more demanding INC algorithm. Results based on the primal and semi-incidence graph can be obtained analogously via according simplifications.

Theorem 3.25. *Given a program Π, a tree decomposition $\mathcal{T} = (T, \chi)$ with $T = (N, \cdot, \cdot)$ of the incidence program $I(\Pi)$ and node $t \in N$ where k is the width of $\mathcal{T}$ and $\ell = max\{bnd(r) : r \in \mathrm{WGT}(\Pi)\}$ is the maximal bound among the weight rules $\mathrm{WGT}(\Pi)$. Then the size of table τ_t, i.e., the number of tuples for node t, using INC algorithm (see Algorithm 3) is at most $2^{(2^k+k)} \cdot (\ell+3)^{k\cdot(2^k+1)}$.*

Proof. (Sketch) The size bound is due to the fact that we can have 2^k local witness and for each local witness a subset of the set of local witnesses ($2^{(2^k)}$ many) forming the set of local counterwitnesses. Moreover, we observe that Algorithm 3 can be easily modified such that a state $\sigma : R_t \to \mathbb{N}_0 \cup \{+\infty\}$ for a node $t \in N$ assigns each weight rule $r \in \mathrm{WGT}(\Pi)$ a non-negative integer $\sigma(r) \leq \mathrm{bnd}(r) + 1$, each choice rule $r \in \mathrm{CH}(\Pi)$ a non-negative integer $\sigma(r) \leq 2$ and each disjunctive rule $r \in \mathrm{DISJ}(\Pi)$ a non-negative integer $\sigma(r) \leq 1$. This is the case since we need to model $\sigma(r) = 0$ and $\sigma(r) = +\infty$ for each rule r. Moreover, for choice rules r, it suffices to additionally model whether $1 \leq \sigma(r) < +\infty$, and for weight rules r, we require to remember any weight $1 \leq \sigma(r) \leq \mathrm{bnd}(r)$. Hence, we finally observe that for each of the $2^k \cdot 2^{(2^k)}$ many lines we have to store in the worst case $(\ell+3)^k$ different states for each certificate, which yields $(\ell+3)^{k\cdot(2^k+1)}$ many combinations. Consequently, the number of tuples per node is at most $2^k \cdot 2^{(2^k)} \cdot (\ell+3)^{k(2^k+1)}$. Hence, the claim sustains. □

Theorem 3.26. *Given a program Π, the approaches* DynASP 2 *and* DynASP 2.5 *(see Section 3.2) using algorithm* INC *run in time $O(2^{(2^k+k)} \cdot (\ell+3)^{[k\cdot(2^k+1)]} \cdot \|I(\Pi)\|^c)$ for some constant c where k is the treewidth of the incidence graph $I(\Pi)$ and $\ell = max\{bnd(r) : r \in \mathrm{WGT}(\Pi)\}$ is the maximal bound among the weight rules $\mathrm{WGT}(\Pi)$.*

Proof. (Sketch) Let w and c be constants. Then we can compute a tree decomposition of size at most w or decide that no such decomposition exists in time $g(w) \cdot \|I(\Pi)\|^c$ [15] for some computable function g where $\|I(\Pi)\|$ is the size of the incidence graph of Π. Hence, we can compute a tree decomposition $\mathcal{T} = (T, \chi)$ where $T = (N, \cdot, \cdot)$ of width k in time $k \cdot g(k) \cdot \|I(\Pi)\|^c$. The result

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 $2^k \cdot 2^{(2^k)} \cdot (\ell+3)^{k(2^k+1)}$ shown in Theorem 3.25. □

Corollary 3.27. *Given a program* Π*, the approaches* DynASP 2 *and* DynASP 2.5 *(see Section 3.2) using algorithm* INC *are fixed-parameter tractable (see ,e.g., [33]) with respect to the treewidth* k *of* $I(\Pi)$ *and the maximal bound* ℓ *among* $\mathrm{WGT}(\Pi)$*, i.e., the runtimes correspond to* $f(k+\ell) \cdot \|I(\Pi)\|^c$ *for some computable function* f *and constant* c*.*

Proof. The corollary immediately follows from Theorem 3.26, since $f(k+\ell) = k \cdot g(k) \cdot 2^k \cdot 2^{(2^k)} \cdot (\ell+3)^{k(2^k+1)}$. □

Note that in practice we assume bounded lower bounds for weight rules and bounded choice rules such that both the maximal bound among weight rules and the biggest cardinality of the heads among choice rules falls below the range of an integer number in real computer systems. In other words $(\ell+3)$ of Theorem 3.26 can be considered a system-dependant constant s, leading to the worst-case runtime $O(2^{(2^k+k)} \cdot s^{[k\cdot(2^k+1)]} \cdot \|I(\Pi)\|^c)$ for some constant c in practice.

3.5 Extensions

For simplicity, we note the following assumptions of this section. Without further notice, we let $\mathcal{A} \in \{$ PRIM, INVPRIM SINC, INC, $\}$, Π be the given program, and $\mathcal{T} = (T, \chi)$ with $T = (N, \cdot, \cdot)$ be a tree decomposition of the respective graph representation of Π (the primal, or semi-incidence, or incidence graph depending on $\mathcal{A}$). We also assume for $t \in N$ that $t' \in N$ and $t'' \in N$ are the successors of t if t is a join node and $t' \in N$ is the only successor of t if t is not a join node and not a leaf node. For convenience, we denote by $v \in \mathcal{A}_t(v')$ ($v \in \mathcal{A}_t(v', v'')$) for given tuple(s) v' (v' and v'') that tuple v is in the resulting table τ_t for node $t \in (N \setminus \mathrm{LEAF}(\mathcal{T}))$ using algorithm $\mathcal{A}$ and the table(s) $\{v'\}$ ($\{v'\}$, $\{v''\}$) for node t' (nodes t', t'').

3.5.1 Extensions to Optimization

The algorithms presented in Section 3.3 cannot handle optimization rules. In the following, we introduce the extension of our three proposed ASP dynamic programming algorithms such that we can handle optimization rules.

Recall definition of the objective $\mathrm{o}(\Pi, M, A)$ for program Π and sets A and M of atoms of Π (see Section 2). In order to compute the objective of the local witnesses in the table τ_t for node $t \in N$, we need to extend every tuple $v \in \tau_t$ by a value describing the current value of the objective. Recall that tuples are of the form $v = \langle M, \cdot \rangle$ or $v = \langle M, \cdot, \cdot \rangle$ depending on the algorithm $\mathcal{A}$. Hence, we obtain tuples of the form $v = \langle M, \cdot, c \rangle$ or $v = \langle M, \cdot, \cdot, c \rangle$, respectively, for some integer c. We call the value c the *cost* of v up to t and write c_v for the cost c of tuple v. Before we specify the algorithm, we let $\mathrm{vmin}_{t'}(v) := \{v' : v' \in \tau_{t'}, v \in \mathcal{A}_t(v')\rangle, c_{v'} = \mathrm{cmin}_{t'}(v)\}$ be the tuples of minimum cost where $\mathrm{cmin}_{t'}(v) := min\{c_{v'} : v' \in \tau_{t'}, v \in \mathcal{A}_t(v')\}$. Similarly, we define $\mathrm{vmin}_{(t',t'')}(v) := \{(v', v'') : v' \in \tau_{t'}, v'' \in \tau_{t''}, v \in \mathcal{A}_t(v', v''), c_{t'} + c_{t''} = \mathrm{cmin}_{(t',t'')}(v)\}$ for $t \in \mathrm{JOIN}(\mathcal{T})$ where $\mathrm{cmin}_{(t',t'')}(v) := min\{c_{t'} + c_{t''} : v' \in \tau_{t'}, v'' \in \tau_{t''}, v \in \mathcal{A}_t(v', v'')\}$.

The algorithm extension for minimal models is presented in Algorithm 4.

Algorithm 4: OPT-$\mathcal{A}$ algorithm extension.

Data: Algorithm $\mathcal{A}$, Program Π, Node t and its children t' and t'', if exist(s), of fixed tree decomposition $\mathcal{T}$.
Result: Modified table τ_t for node t. For tuple $\langle M, \cdots, c\rangle$, we only state cost c as the remaining part is unaffected and the local witness M to indicate, which local witness is affected. Therefore, we abuse the symbol "$\cdots$". The actual arity of the tuple depends on the algorithm $\mathcal{A}$.

if $t \in \mathrm{LEAF}(\mathcal{T})$ **then**
  $\tau_t := \left\{ \left\langle \emptyset, \cdots, 0 \right\rangle \right\}$
if $t \in \mathrm{AI}(\mathcal{T})$ *and let* $a \in A_t \setminus A_{t'}$ **then**
  $\tau_t := \left\{ \left\langle M, \cdots, \mathrm{o}(\Pi, \emptyset, \{a\}) + \mathrm{cmin}_{t'}(v) \right\rangle \;\middle|\; \langle M, \dots\rangle \in \tau_{t'} \right\} \bigcup$
  $\left\{ \left\langle M_a^+, \cdots, \mathrm{o}(\Pi, \{a\}, \{a\}) + \mathrm{cmin}_{t'}(v) \right\rangle \;\middle|\; \langle M, \dots\rangle \in \tau_{t'} \right\}$
if $t \in [\mathrm{RI}(\mathcal{T}) \cup \mathrm{RR}(\mathcal{T})]$ **then**
  $\tau_t := \left\{ \left\langle M, \cdots, \mathrm{cmin}_{t'}(v) \right\rangle \;\middle|\; \langle M, \dots\rangle \in \tau_{t'} \right\}$
if $t \in \mathrm{AR}(\mathcal{T})$ *and let* $a \in A_{t'} \setminus A_t$ **then**
  $\tau_t := \left\{ \left\langle M_a^-, \cdots, \mathrm{cmin}_{t'}(v) \right\rangle \;\middle|\; \langle M, \dots\rangle \in \tau_{t'} \right\}$
if $t \in \mathrm{JOIN}(\mathcal{T})$ **then**
  $\tau_t := \left\{ \left\langle M, \cdots, \mathrm{cmin}_{(t',t'')}(v) - \mathrm{o}(\Pi, M, A_t) \right\rangle \;\middle|\; \langle M, \dots\rangle \in \tau_{t'}, \langle M, \dots\rangle \in \tau_{t''} \right\}$

Algorithm 5: CNTOPT-$\mathcal{A}$ algorithm extension.

Data: Algorithm $\mathcal{A}$, Program Π, Node t and its children t' and t'', if exist(s), of fixed tree decomposition $\mathcal{T}$.
Result: Modified table τ_t for node t. For tuple $\langle M, \cdots, n\rangle$, we only state counter n as the remaining part is unaffected and the local witness M to indicate, which local witness is affected. Therefore, we abuse the symbol "$\cdots$". The actual arity of the tuple depends on the algorithm $\mathcal{A}$.

if $t \in \mathrm{LEAF}(\mathcal{T})$ **then**
  $\tau_t := \left\{ \left\langle \emptyset, \cdots, 1 \right\rangle \right\}$
if $t \in \mathrm{AI}(\mathcal{T})$ *and let* $a \in A_t \setminus A_{t'}$ **then**
  $\tau_t := \left\{ \left\langle M, \cdots, \Sigma_{v' \in \mathrm{vmin}_{t'}(v)}\, n_{v'} \right\rangle \;\middle|\; \langle M, \dots\rangle \in \tau_{t'} \right\} \bigcup$
  $\left\{ \left\langle M_a^+, \cdots, \Sigma_{v' \in \mathrm{vmin}_{t'}(v)}\, n_{v'} \right\rangle \;\middle|\; \langle M, \dots\rangle \in \tau_{t'} \right\}$
if $t \in [\mathrm{RI}(\mathcal{T}) \cup \mathrm{RR}(\mathcal{T})]$ **then**
  $\tau_t := \left\{ \left\langle M, \cdots, \Sigma_{v' \in \mathrm{vmin}_{t'}(v)}\, n_{v'} \right\rangle \;\middle|\; \langle M, \dots\rangle \in \tau_{t'} \right\}$
if $t \in \mathrm{AR}(\mathcal{T})$ *and let* $a \in A_{t'} \setminus A_t$ **then**
  $\tau_t := \left\{ \left\langle M_a^-, \cdots, \Sigma_{v' \in \mathrm{vmin}_{t'}(v)}\, n_{v'} \right\rangle \;\middle|\; \langle M, \dots\rangle \in \tau_{t'} \right\}$
if $t \in \mathrm{JOIN}(\mathcal{T})$ **then**
  $\tau_t := \left\{ \left\langle M, \cdots, \Sigma_{(v',v'') \in \mathrm{vmin}_{(t',t'')}(v)}\, (n_{v'} \cdot n_{v''}) \right\rangle \;\middle|\; \langle M, \dots\rangle \in \tau_{t'}, \langle M, \dots\rangle \in \tau_{t''} \right\}$

3.5.2 Extensions to Counting

Next, we extend our algorithms by counters on the seen solutions to solve the #ASP problem. Therefore, we extend every tuple $v \in \tau_t$ in the table τ_t for node $t \in N$ by an integer describing the number of potential answer sets, which the tuple gives rise to. Recall that tuples are of the form $v = \langle M, \cdot, c\rangle$ or $v = \langle M, \cdot, \cdot, c\rangle$ depending on the algorithm $\mathcal{A}$. Hence, we obtain tuple of the form $v = \langle M, \cdot, c, n\rangle$ or $v = \langle M, \cdot, \cdot, c, n\rangle$, respectively, for some integer n. We call the value n the *counter* of v up to t and write n_v for the counter n of tuple v.

The algorithm extension for counting minimal models is presented in Algorithm 5.

3.5.3 Runtime results

Note that one is able to derive similar results as in Theorem 3.26 for algorithms using the extensions presented above, but for the extension to optimization, we additionally require to fix parameter $m := max\{\mathrm{wght}(r) : r \in \mathrm{OPT}(\Pi)\}$ (maximum weight among the optimization rules) leading to at most $2^{(2^k+k)} \cdot (\ell + m + 3)^{k\cdot(2^k+1)}$ tuples per table τ_t for each node t using algorithm INC (compare to Theorem 3.25).

3.6 Comparison of the Algorithms

The algorithms presented are all based on tree decompositions of different graph representations of the ASP program.

Observation 3.28. *Let Π be a program. The treewidth of the incidence graph is at most the treewidth of its semi-incidence program, which treewidth is at most the treewidth of its primal graph plus 1.*

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 states 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.

A final 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 [83].

Counting Complexity

In the following, we show that (under standard complexity-theoretic assumptions) the counting problem we consider here is strictly harder than #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 3.29. *The problem* #ASP *for programs without optimization rules is* $\#\cdot$ coNP*-complete.*

Proof. Observe that programs containing choice and weight rules can be compiled to disjunctive ones (normalization) without these rule types (see [19]) using a polynomial number (in the original program size) of disjunctive rules. Membership follows from the fact that, given such a normalized program Π and an interpretation I, checking whether I is an answer of Π is coNP-complete, see e.g., [64]. Hardness is a direct consequence of $\#\cdot$ coNP-hardness for the problem of counting subset minimal models of a CNF formula [35], 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 rules (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 [59].

4 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 3.3. 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 [96]. 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 [44, 46] 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 quide to DynASP 2

After downloading, compiling and installing htd (library for computing tree decompositions via heuristics), sharp (framework for supporting algorithms based on dynamic programming on tree decompositions) and DynASP 2 at 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 (used in htd)
  -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 3.3). 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 stdinput or a file (FILE) and is expected to be in SModels syntax as defined in the manual [96]. Since the grounder Gringo [44, 46] 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.

$$gringo\ [\text{FILE}]\ \mid\ dynasp\ [\text{OPTION}]\ ...$$

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 $gringo\ 3\text{col\_ex.lp}\ \mid\ dynasp\ -\text{bb}$. 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

usr,sys,cpu,wall,description
0.00,0.00,0.00,0.00,program start
0.00,0.00,0.00,0.00,parsing time
0.00,0.00,0.00,0.00,tree decomposition time
0.03,0.00,0.03,0.04,PASS 1
0.00,0.00,0.00,0.00,solving time
0.00,0.00,0.00,0.00,solution extraction time
```

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.

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. Note that 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-FLAT^2 [13][9], an extension of the dynamic programming framework D-FLAT [2][10] 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 3.2) 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.

4.3.1 A quick hands-on quide 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).

[9] D-FLAT^2 is open source and available at https://www.github.com/hmarkus/dflat-2.

[10] D-FLAT allows rapid prototyping of algorithms exploiting dynamic programming on tree decompositions and can be downloaded at https://www.github.com/bbliem/dflat.

```
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 (used in htd)
  -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).

```
Parsing...
Initializing solver...
Decomposing...
TREEWIDTH: 6
Solving...
PASS 1 finished

PASS 2 finished

PASS 3 finished
SIZE: 7
done.
OPTIMAL WEIGHT: 0
SOLUTION COUNT: 36

usr,sys,cpu,wall,description
0.00,0.00,0.00,0.00,program start
0.00,0.00,0.00,0.00,parsing time
0.00,0.00,0.00,0.00,tree decomposition time
0.01,0.00,0.01,0.00,PASS 1
0.00,0.00,0.00,0.01,PASS 2
0.01,0.00,0.01,0.01,PASS 3
0.00,0.00,0.00,0.00,solving time
0.00,0.00,0.00,0.00,solution extraction time
LEAF: 36
EXCHANGE: 27
JOIN: 35

FIRSTS: 783
REUSES: 696
SPLITUPS: 767
```

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], DepQBF0[11], Clasp 3.1.4 [47], and SharpSAT 12.08 [95].

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[12] 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.

[11] Since DepQBF [70] does not support counting by default, we implemented a naive counting approach into DepQBF using methods described in [69], which we call DepQBF0.

[12] 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 ℓ 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 [27] and splited by transportation type, e.g., train, metro, tram, combinations. We heuristically computed tree decompositions [30] 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.

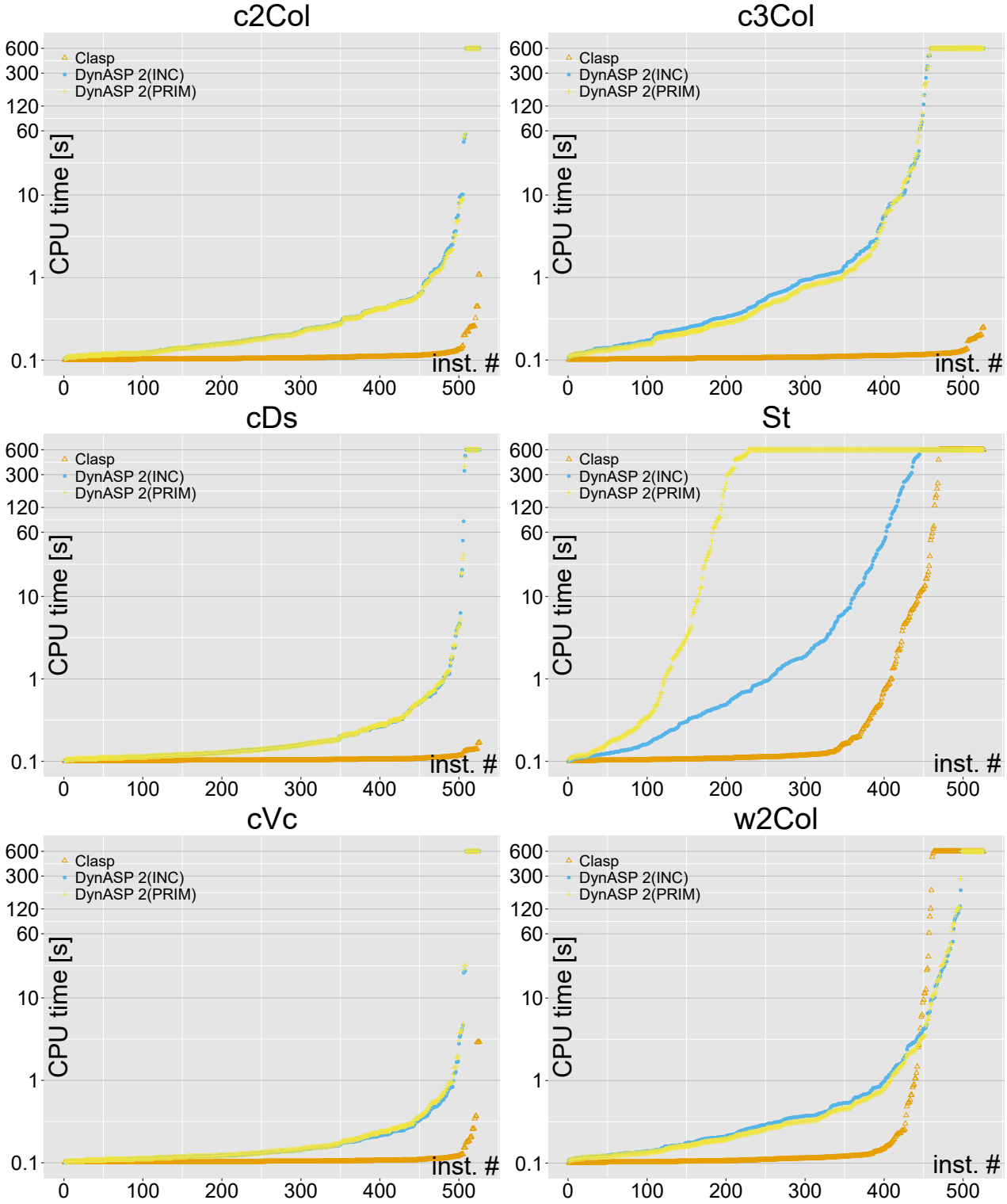


Figure 14 COMPUTEAS: Result visualization of structured instance sets.

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 theses circumstances. However, it is quite surprising on the other hand that DynASP 2 seems to work better solving the weighted 2-colorability variant (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.

	C2COL	C3COL	CDS	ST	CVC	W2COL	SDS	SVC
Clasp	**0.12 (0)**	**0.11 (0)**	**0.11 (3)**	**69.00 (57)**	**0.13 (0)**	76.39 (64)	**53.48 (20)**	**0.10 (0)**
INC	21.29 (18)	85.54(73)	22.74 (19)	115.11 (91)	20.88 (18)	**37.02 (29)**	571.62 (498)	26.70 (21)
PRIM	21.27 (18)	85.20(72)	22.63 (19)	368.31 (320)	20.91 (18)	37.37 **(29)**	574.14 (505)	26.66 (21)

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

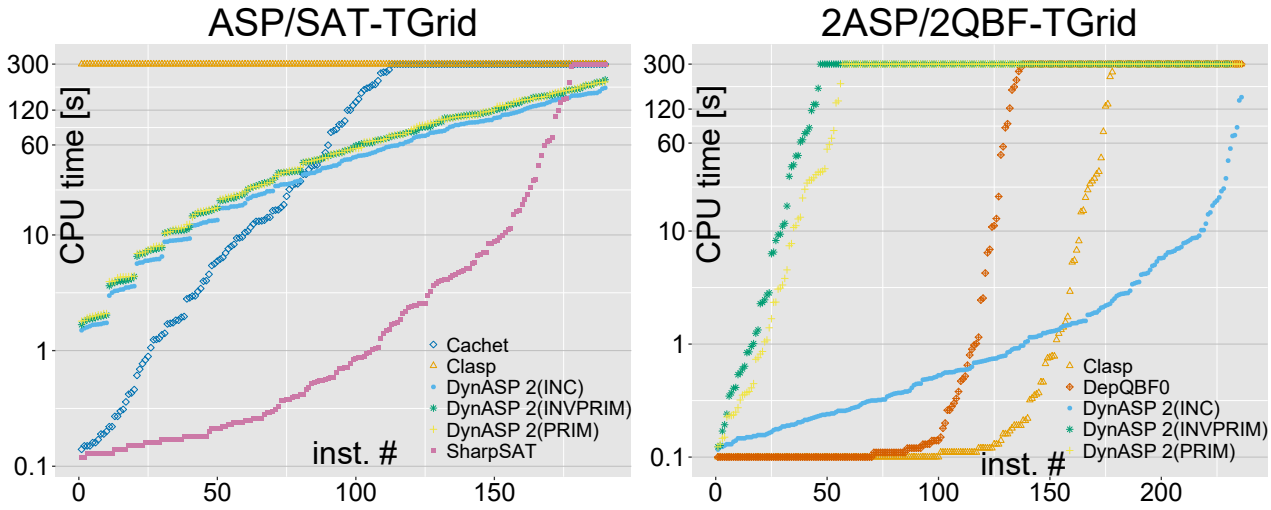


Figure 15 #ASP: Result visualization of randomly generated instance sets.

	C2COL		C3COL		CDS		ST		CVC		SVC	
CLASP	31.72	(21)	**0.10**	**(0)**	8.99	(3)	**0.21**	**(0)**	29.88	(21)	98.34	(71)
INC	**1.43**	**(0)**	0.58	**(0)**	**0.54**	**(0)**	115.02	(498)	**0.68**	**(0)**	**0.78**	**(0)**
INVP	1.50	**(0)**	0.47	**(0)**	0.79	**(0)**	91.92	(248)	0.99	**(0)**	1.15	**(0)**
PRIM	1.54	**(0)**	0.53	**(0)**	0.68	**(0)**	79.36	(221)	0.99	**(0)**	1.30	**(0)**

Table 2 #ASP: Runtime results (given in sec.) on real-world graph instance sets and number of timeouts in brackets.

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-TGRID and ASP-TGRID: Cachet solved 125 instances. Clasp always timed out for both configurations (branch and bound; and unsatisfiable core). A reason could be the high number of solutions as Clasp counts the models by enumerating them. DynASP 2$(\cdot)$ solved each instance within at most 270 seconds (on average 67 seconds). The best configuration with respect to runtime was PRIM. However, the running times of the different configurations were close. We observed as expected a sub-polynomial growth in the runtime with an increasing number of solutions. SharpSAT timed out on 3 instances and ran into a memory out on 7 instances, but solved most of the instances quite fast. Half of the instances were solved within 1 seconds and more than 80% of the instances within 10 seconds, and about 9% of the instances took more than 100 seconds. The number of solutions does not have an impact on the runtime of SharpSAT. SharpSAT was the fastest solver in total, however, DynASP 2$(\cdot)$ solved all instances. The results are illustrated in the two graphs of Figure 15.

2QBF-TGRID and 2ASP-TGRID: Clasp solved more than half of the instances in less than 1 second, however, timed out on 59 instances. DepQBF0 shows a similar behavior as Clasp, which is not surprising as both solvers count the number of solutions by enumerating them and hence the number of solutions has a significant impact on the runtime of the solver. However, Clasp is throughout faster than DepQBF0. DynASP 2(INC) solved half of the instances within less than 1 second, about 92% of the instances within less than 10 seconds, and provided solutions also 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$(\cdot)$ solved for each set but the set ST more than 80% of the instances in less than 1 second and the remaining instances in less than 100 seconds. For ST the situation was different. Half of the instances were solved in less than 10 seconds and a little less than the other half timed out. Similar to the random setting, DynASP 2$(\cdot)$ ran still fast on instances with a large number of solutions. Whenever the instance had relatively

	C2COL	C3COL	CDS	ST	CVC	W2COL	SDS	SVC
INC	21.29 **(18)**	85.54(73)	22.74 **(19)**	115.11 (91)	**20.88 (18)**	**37.02 (29)**	571.62 **(498)**	26.70 (21)
PRIM	**21.27 (18)**	**85.20(72)**	**22.63 (19)**	368.31 (320)	20.91 **(18)**	37.37 **(29)**	574.14 (505)	26.66 (21)
INC 2.5	21.69 **(18)**	102.39(82)	24.30 (21)	**74.76 (62)**	21.65 **(18)**	53.01 (41)	569.70 **(498)**	24.96 (21)
	-r -n -l 2	-r -n -l 2	-r -n -l 2	-r -l 2	-r -n -l 2	-r -n -l 2	-r -l 2	-r -l 2
PRIM 2.5	21.95 **(18)**	108.49(86)	23.49 (20)	317.11 (275)	22.67 (19)	59.69 (46)	**568.56 (498)**	**20.93 (18)**
	-r -l 5	-r -n -l 2	-r -n -l 2	-n -l 2	-r -n -l 2	-r -n -l 2	-n -l 2	-l 2

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)

few solutions Clasp was faster, otherwise DynASP 2$(\cdot)$ (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$(\cdot)$ is still reasonably fast on structured instances with few solutions as it yields the number of solutions mostly within less than 10 seconds. We observed that DynASP 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.

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.

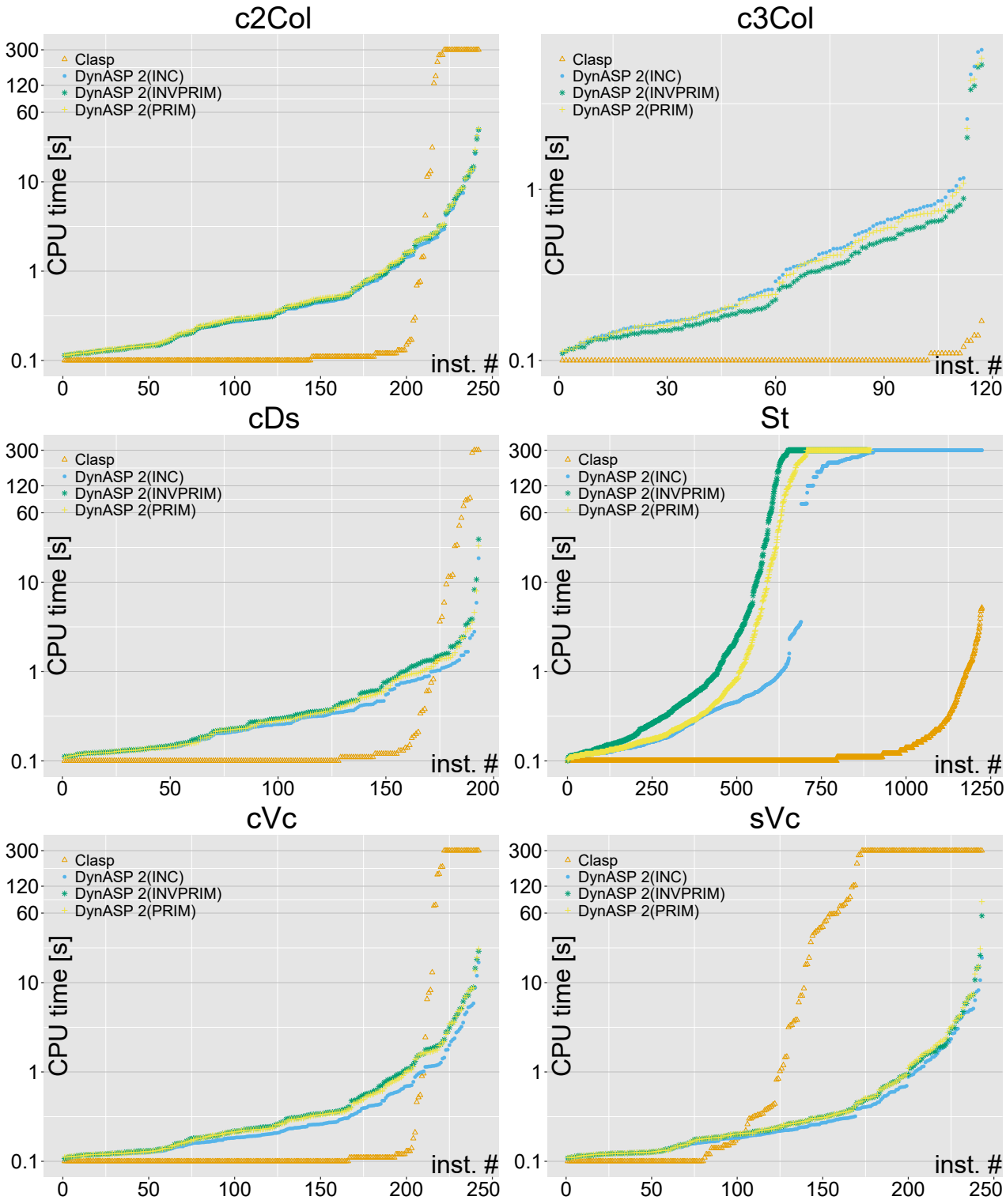


Figure 16 #ASP: Result visualization of structured instance sets.

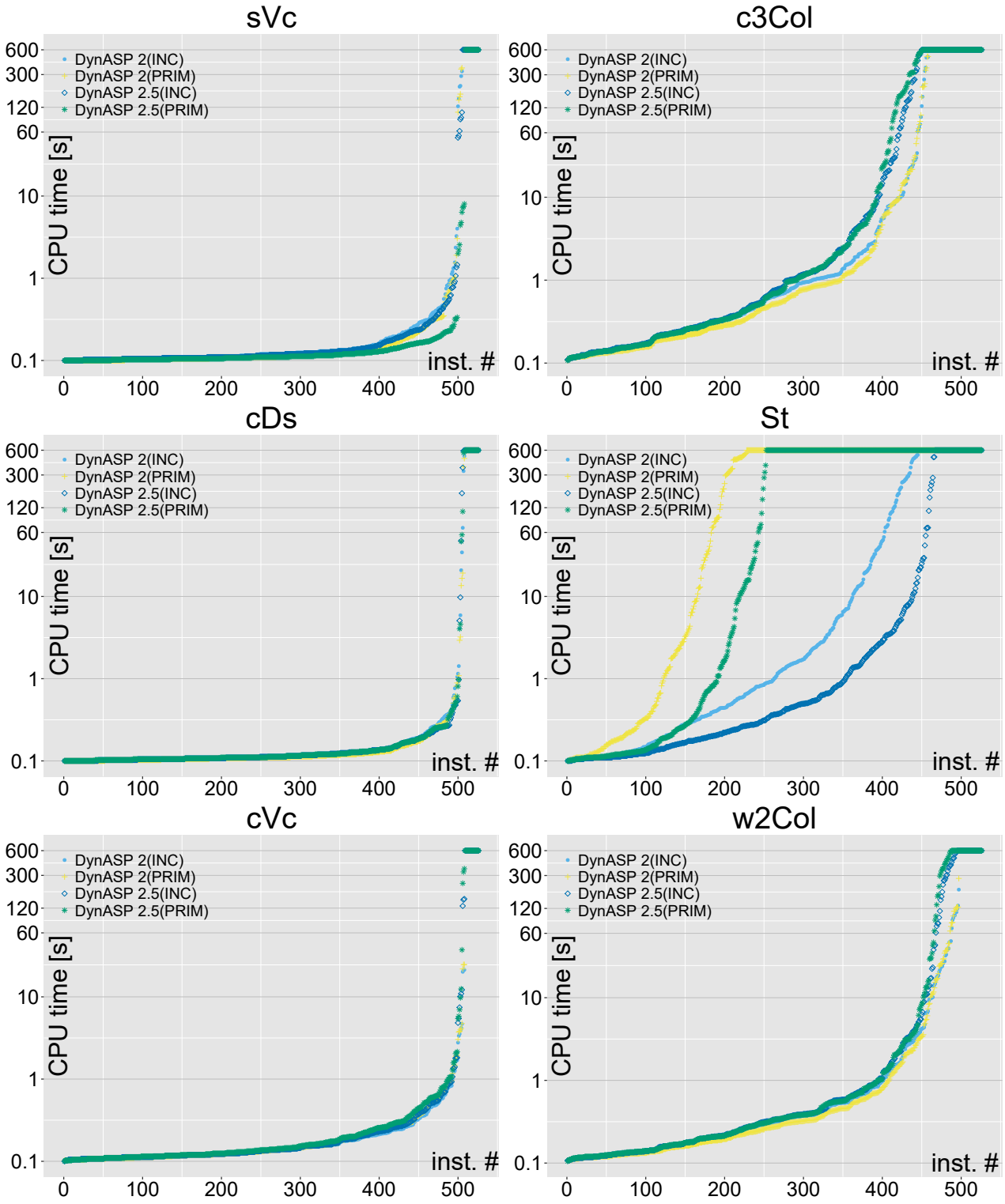


Figure 17 DynASP 2 vs. DynASP 2.5: Result visualization of structured instance sets.

6 Conclusion

In this paper, we have presented several dynamic programming algorithms for evaluating answer sets of logic programs based on tree decompositions, and compared two prototype implementations to existing solvers. Generally, our algorithms are not built for all-purpose ASP solving, since their runtime depends heavily on the treewidth of the given ASP instance. However, our intention is to show that they can perform well on instances of low treewidth. For the COMPUTEAS problem, our algorithms also have a significant disadvantage since dynamic programming is not well-suited to finding only one solution. However, interestingly, in the case of optimization (i.e. the problem of finding the best answer set) we found that there are instances—especially instances with non-uniform weights—where our algorithms are able to outperform existing solvers on instances of low treewidth.

On the other hand, for aggregation problems that require computing all solutions, our algorithms perform very well when compared to state-of-the-art solvers. In particular, we found that the #ASP problem is an area where our algorithms perform very well. On instances of low treewidth, we were able to consistently outperform state-of-the-art SAT model counters, ASP solvers, and QBF solvers.

Compared to each other, our optimized DynASP 2.5 implementation payed off especially where optimization problems with non-uniform weights were considered.

Future Work. Since model counting seems to be an area well-suited for evaluation by our algorithms, we plan to extend them to weighted model counting algorithms that can then be used to solve the Bayesian Inference problem (that is, what is the probability that a given atom is true in an answer set). This problem is important in mathematical statistics and particularly in the evaluation and analysis of sequences of data. It finds use in expert systems, pattern recognition, statistical classification (e.g. for email spam filters), or even phylogenetics.

References

[1] Michael Abseher. htd: A small but efficient c++ library for computing (customized) tree and hypertree decompositions. https://github.com/mabseher/htd, 2016.

[2] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. D-FLAT: Progress report. Technical Report DBAI-TR-2014-86, DBAI, Vienna University of Technology, 2014.

[3] Michael Abseher, Frederico Dusberger, Nysret Musliu, and Stefan Woltran. Improving the efficiency of dynamic programming on tree decompositions via machine learning. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 275–282. The AAAI Press, 2015.

[4] Rachit Agarwal, Philip Brighten Godfrey, and Sariel Har-Peled. Approximate distance queries and compact routing in sparse graphs. In *INFOCOM 2011. 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 10-15 April 2011, Shanghai, China*, pages 1754–1762. IEEE, 2011.

[5] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native ASP solver based on constraint learning. In Pedro Cabalar and Tran Cao Son, editors, *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, volume 8148 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2013.

[6] Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer, and Giorgio Terracina. The disjunctive datalog system DLV. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*, pages 282–301. Springer, 2011.

[7] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Algebraic Discrete Methods*, 8(2):277–284, 1987.

[8] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, Cambridge, United Kingdom, 2009.

[9] Markus Aschinger, Conrad Drescher, Georg Gottlob, Peter Jeavons, and Evgenij Thorstensen. Structural decomposition methods and what they are good for. In Thomas Schwentick and Christoph Dürr, editors, *28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011, March 10-12, 2011, Dortmund, Germany*, volume 9 of *LIPIcs*, pages 12–28. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[10] Emgad H. Bachoore and Hans L. Bodlaender. A branch and bound algorithm for exact, upper, and lower bounds on treewidth. In Siu-Wing Cheng and Chung Keung Poon, editors, *Algorithmic Aspects in Information and Management, Second International Conference, AAIM 2006, Hong Kong, China, June 20-22, 2006, Proceedings*, volume 4041 of *Lecture Notes in Computer Science*, pages 255–266. Springer, 2006.

[11] Anne Berry, Pinar Heggernes, and Geneviève Simonet. The minimum degree heuristic and the minimal triangulation process. In Hans L. Bodlaender, editor, *Graph-Theoretic Concepts in Computer Science, 29th International Workshop, WG 2003, Elspeet, The Netherlands, June 19-21, 2003, Revised Papers*, volume 2880 of *Lecture Notes in Computer Science*, pages 58–70. Springer, 2003.

[12] Nicole Bidoít and Christine Froidevaux. Negation by default and unstratifiable logic programs. *Theoretical Computer Science*, 78(1):86–112, 1991.

[13] Bernhard Bliem, Günther Charwat, Markus Hecher, and Stefan Woltran. D-flat^2: Subset minimization in dynamic programming on tree decompositions made easy. *Fundamenta Informaticae*, 147(1):27–61, 2016.

[14] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2):1–21, 1993.

[15] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[16] Hans L. Bodlaender. Treewidth: Algorithmoc techniques and results. In Igor Prívara and Peter Ruzicka, editors, *Mathematical Foundations of Computer Science 1997, 22nd International Symposium, MFCS'97, Bratislava, Slovakia, August 25-29, 1997, Proceedings*, volume 1295 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 1997.

[17] Hans L. Bodlaender. Discovering treewidth. In Peter Vojtás, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *SOFSEM 2005: Theory and Practice of Computer Science, 31st Conference on Current Trends in Theory and Practice of Computer Science, Liptovský Ján, Slovakia, January 22-28, 2005, Proceedings*, volume 3381 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005.

[18] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. upper bounds. *Information and Computation*, 208(3):259–275, 2010.

[19] Jori Bomanson, Martin Gebser, and Tomi Janhunen. Rewriting optimization statements in answer-set programs. In Manuel Carro, Andy King, Neda Saeedloei, and Marina De Vos, editors, *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016, October 18-21, 2016, New York, USA*, volume 52 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[20] Adrian Bondy and U.S.R. Murty. *Graph Theory*, volume 244 of *Graduate Texts in Mathematics*. Springer, London, United Kingdom, 2008.

[21] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[22] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing disjunctive datalog by constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

[23] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-core-2 input language format. https://www.mat.unical.it/aspcomp2013/ASPStandardization, 2015.

[24] François Clautiaux, Aziz Moukrim, Stéphane Nègre, and Jacques Carlier. Heuristic and metaheuristic methods for computing graph treewidth. *RAIRO - Operations Research*, 38(1):13–26, 2004.

[25] Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, Formal Models and Semantics, pages 193–242. The MIT Press, Cambridge, MA, USA, 1990.

[26] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer International Publishing, Cham, Switzerland, 2015.

[27] Jehiah Czebotar. GTFS data exchange. http://www.gtfs-data-exchange.com/, 2016.

[28] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[29] Rina Dechter. *Constraint processing*. Elsevier Morgan Kaufmann, Amsterdam, The Netherlands, 2003.

[30] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In Alexander F. Gelbukh and Eduardo F. Morales, editors, *MICAI 2008: Advances in Artificial Intelligence, 7th Mexican International Conference on Artificial Intelligence, Atizapán de Zaragoza, Mexico, October 27-31, 2008, Proceedings*, volume 5317 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2008.

[31] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer Verlag, Berlin Heidelberg, Germany, 4th edition, 2010.

[32] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, New York, NY, USA, 1999.

[33] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, London, United Kingdom, 2013.

[34] Paul E. Dunne. Computational properties of argument systems satisfying graph-theoretic constraints. *Artificial Intelligence*, 171(10-15):701–729, 2007.

[35] Arnaud Durand, Miki Hermann, and Phokion G. Kolaitis. Subtractive reductions and complete problems for counting complexity classes. *Theoretical Computer Science*, 340(3):496–513, 2005.

[36] Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):123–165, 2007.

[37] Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):289–323, 1995.

[38] Islam Elkabani, Enrico Pontelli, and Tran Cao Son. Smodels^{a} - A system for computing answer sets of logic programs with aggregates. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 427–431. Springer, 2005.

[39] Wolfgang Faber. Decomposition of nonmonotone aggregates in answer set programming. In Michael Fink, Hans Tompits, and Stefan Woltran, editors, *20th Workshop on Logic Programming, Vienna, Austria, February 22–24, 2006*, volume 1843-06-02 of *INFSYS Research Report*, pages 164–171. Technische Universität Wien, Austria, 2006.

[40] Johannes Klaus Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Counting answer sets via dynamic programming. In Johannes Klaus Fichte and Christoph Redl, editors, *Informal Proceedings of the First Workshop on Trends and Applications of Answer Set Programming, TAASP 2016, Klagenfurt, Austria, September 26, 2016*, 2016.

[41] Johannes Klaus Fichte and Stefan Szeider. Backdoors to tractable answer set programming. *Artificial Intelligence*, 220:64–103, 2015.

[42] Eldar Fischer, Johann A. Makowsky, and Elena V. Ravve. Counting truth assignments of formulas of bounded tree-width or clique-width. *Discrete Applied Mathematics*, 156(4):511–529, 2008.

[43] Serge Gaspers and Stefan Szeider. Strong backdoors to bounded treewidth SAT. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 489–498. IEEE Computer Society, 2013.

[44] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Marius Lindauer, Max Ostrowski, Javier Romero, Torsten Schaub, and Sven Thiele. Potassco user guide. https://sourceforge.net/projects/potassco/files/guide/, 2015.

[45] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, Williston, VT, USA, 2012.

[46] Martin Gebser, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Sven Thiele. On the input language of ASP grounder gringo. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 502–508. Springer, 2009.

[47] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52–89, 2012.

[48] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988*, volume 2, pages 1070–1080. The MIT Press, 1988.

[49] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. In David Maxwell Chickering and Joseph Y. Halpern, editors, *UAI '04, Proceedings of the 20th Conference in Uncertainty in Artificial Intelligence, Banff, Canada, July 7-11, 2004*, pages 201–208. AUAI Press, 2004.

[50] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[51] Georg Gottlob, Reinhard Pichler, and Fang Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artificial Intelligence*, 174(1):105–132, 2010.

[52] Georg Gottlob, Francesco Scarcello, and Martha Sideri. Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artif. Intell.*, 138(1-2):55–86, 2002.

[53] Georg Gottlob and Stefan Szeider. Fixed-parameter algorithms for artificial intelligence, constraint satisfaction and database problems. *The Computer Journal*, 51(3):303–325, 2008.

[54] Jens Gramm, Arfst Nickelsen, and Till Tantau. Fixed-parameter algorithms in phylogenetics. *The Computer Journal*, 51(1):79–101, 2008.

[55] Gregory Gutin. Should we care about huge imbalance in parameterized algorithmics? *Parameterized Complexity Newsletter*, 11(2), December 2015.

[56] Thomas Hammerl and Nysret Musliu. Ant colony optimization for tree decompositions. In Peter I. Cowling and Peter Merz, editors, *Evolutionary Computation in Combinatorial Optimization, 10th European Conference, EvoCOP 2010, Istanbul, Turkey, April 7-9, 2010. Proceedings*, volume 6022 of *Lecture Notes in Computer Science*, pages 95–106. Springer, 2010.

[57] Thomas Hammerl, Nysret Musliu, and Werner Schafhauser. Metaheuristic algorithms and tree decomposition. In Janusz Kacprzyk and Witold Pedrycz, editors, *Handbook of Computational Intelligence*, pages 1255–1270. Springer, New York, NY, USA, 2015.

[58] Lane A. Hemaspaandra and Heribert Vollmer. The satanic notations: counting classes beyond #p and other definitional adventures. *SIGACT News*, 26(1):2–13, 1995.

[59] Miki Hermann and Reinhard Pichler. Complexity of counting the optimal solutions. *Theoretical Computer Science*, 410(38-40):3814–3825, 2009.

[60] Xiuzhen Huang and Jing Lai. Parameterized graph problems in computational biology. In *Proceeding of the Second International Multi-Symposium of Computer and Computational Sciences, IMSCCS 2007, August 13-15, 2007, The University of Iowa, Iowa City, Iowa, USA*, pages 129–132. IEEE Computer Society, 2007.

[61] Michael Jakl, Reinhard Pichler, and Stefan Woltran. Answer-set programming with bounded treewidth. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 816–822. The AAAI Press, 2009.

[62] Uffe Kjærulff. Optimal decomposition of probabilistic networks by simulated annealing. *Statistics and Computing*, 2(1):7–17, 1992.

[63] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, New York, NY, USA, 1994.

[64] Christoph Koch and Nicola Leone. Stable model checking made easy. In Thomas Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI'99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 70–75. Morgan Kaufmann, 1999.

[65] Pedro Larrañaga, Cindy M. H. Kuijpers, Mikel Poza, and Roberto H. Murga. Decomposing bayesian networks: triangulation of the moral graph with genetic algorithms. *Statistics and Computing*, 7(1):19–34, 1997.

[66] Matthieu Latapy and Clémence Magnien. Measuring fundamental properties of real-world complex networks. *Computing Research Repository (CoRR)*, abs/cs/0609115, 2006.

[67] Wei Li, Pascal Poupart, and Peter van Beek. Exploiting structure in weighted model counting approaches to probabilistic inference. *Journal of Artificial Intelligence Research (JAIR)*, 40:729–765, 2011.

[68] Vladimir Lifschitz. What is answer set programming? In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1594–1597. The AAAI Press, 2008.

[69] Florian Lonsing. Personal communication, 2015.

[70] Florian Lonsing and Armin Biere. Depqbf: A dependency-aware QBF solver. *Journal on Satisfiability, Boolean Modelin and Computation*, 7(2-3):71–76, 2010.

[71] V. Wiktor Marek and Miroslaw Truszczynski. Autoepistemic logic. *Journal of the ACM*, 38(3):588–619, 1991.

[72] Victor W. Marek and Mirosław Truszczyński. Stable models – an alternative logic programming paradigm. In Krzysztof R. Apt, Victor W. Marek, Mirek Truszczynski, and David S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, New York, NY, USA, 2011.

[73] Guy Melançon. Just how dense are dense graphs in the real world?: a methodological note. In Enrico Bertini, Catherine Plaisant, and Giuseppe Santucci, editors, *Proceedings of the 2006 AVI Workshop on BEyond time and errors: novel evaluation methods for information visualization, BELIV 2006, Venice, Italy, May 23, 2006*, pages 1–7. ACM Press, 2006.

[74] Michael Morak, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. A dynamic-programming based asp-solver. In Tomi Janhunen and Ilkka Niemelä, editors, *Logics in Artificial Intelligence - 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings*, volume 6341 of *Lecture Notes in Computer Science*, pages 369–372. Springer, 2010.

[75] Nysret Musliu. An iterative heuristic algorithm for tree decomposition. In Carlos Cotta and Jano I. van Hemert, editors, *Recent Advances in Evolutionary Computation for Combinatorial Optimization*, volume 153 of *Studies in Computational Intelligence*, pages 133–150. Springer, New York, NY, USA, 2008.

[76] Nysret Musliu and Werner Schafhauser. Genetic algorithms for generalized hypertree decompositions. *European Journal of Industrial Engineering*, 1(3):317–340, 2007.

[77] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*, volume 31 of *Oxford Lecture Series in Mathematics and its Applications*. Oxford University Press, Oxford, United Kingdom, 2006.

[78] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, USA, 1994.

[79] Reinhard Pichler, Stefan Rümmele, Stefan Szeider, and Stefan Woltran. Tractable answer-set programming with weight constraints: bounded treewidth is not enough. *Theory and Practice of Logic Programming*, 14(2):141–164, 2014.

[80] Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. Belief revision with bounded treewidth. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September*

14-18, 2009. Proceedings, volume 5753 of *Lecture Notes in Computer Science*, pages 250–263. Springer, 2009.

[81] Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

[82] Sigve Hortemo Sæther, Jan Arne Telle, and Martin Vatshelle. Solving #SAT and MAXSAT by dynamic programming. *Journal of Artificial Intelligence Research (JAIR)*, 54:59–82, 2015.

[83] Marko Samer and Stefan Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010.

[84] Marko Samer and Stefan Szeider. Constraint satisfaction with bounded treewidth revisited. *Journal of Computer and System Sciences*, 76(2):103–114, 2010.

[85] Marko Samer and Helmut Veith. Encoding treewidth into SAT. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 45–50. Springer, 2009.

[86] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In Holger H. Hoos and David G. Mitchell, editors, *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada. Proceedings*, volume 3542 of *Lecture Notes in Computer Science*. Springer, 2004.

[87] Claus-Peter Schnorr. On self-transformable combinatorial problems. In H. König, Bernhard Korte, and Klaus Ritter, editors, *Mathematical Programming at Oberwolfbach*, volume 14 of *Mathematical Programming Studies*, pages 225–243. Springer Verlag, Berlin Heidelberg, Germany, 1981.

[88] Kirill Shoikhet and Dan Geiger. A practical algorithm for finding optimal triangulations. In Benjamin Kuipers and Bonnie L. Webber, editors, *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island*, pages 185–190. The AAAI Press / The MIT Press, 1997.

[89] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[90] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[91] Larry J. Stockmeyer and Albert R. Meyer. Word problems requiring exponential time: Preliminary report. In Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong, editors, *Proceedings of the*

5th Annual ACM Symposium on Theory of Computing, STOC, April 30 - May 2, 1973, Austin, Texas, USA, pages 1–9. Association for Computing Machinery, 1973.

[92] Stefan Szeider. On fixed-parameter tractable parameterizations of SAT. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2003.

[93] Robert Endre Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[94] Mikkel Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.

[95] Marc Thurley. sharpsat - counting models with advanced component caching and implicit BCP. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429. Springer, 2006.

[96] Tommi Syrjänen Tommi. Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps, 2002.

[97] Celia Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):23–33, 1976.

A Appendix

A.1 Used Encodings

For completion, we add the encodings for our graph problems (C2COL, W2COL, C3COL, CDS, SDS, 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, 45, 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

```
{ in(X) } ← vertex(X).

dominated(Y) ← in(X), edge(X,Y).
← vertex(X), not in(X), not dominated(X).

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

Listing 9 Encoding for CDS

```
{ in(X) } ← vertex(X).
dominated(X) ← in(Y), edge(Y,X).

%not dominated
← vertex(X), not in(X), not dominated(X).

%guess potential counter candidate
cin(X) ; ncin(X) ← vertex(X).
spoil ← cin(X), not in(X).

cin(X) ← spoil, vertex(X).
ncin(X) ← spoil, vertex(X).

← not spoil.

%SUB
%some X is not dominated, nor in
spoil ← ncin(X), C1 #count{Y: ncin(Y), edge(Y,X)}, C1 = #count{Y: edge(Y,X)}.
```

Listing 10 Encoding for SDS

```
0 { p(X,Y) } 1 ← edge(X,Y).

% For all partitions into two sets there must be a crossing edge.
s1(X) ∨ s2(X) ← vertex(X).
s1(X) ← saturate, vertex(X).
s2(X) ← saturate, vertex(X).
← not saturate.

% Found crossing edge?
saturate ← p(X,Y), s1(X), s2(Y).
saturate ← p(Y,X), s1(X), s2(Y).

% Not a partition due to a set being terminal empty?
% This is the case if the other set contains all terminal elements.
numVertices(N) ← N = #count{ X : terminalVertex(X) }.
saturate ← N #count{ X : s1(X), terminalVertex(X) }, numVertices(N).
saturate ← N #count{ X : s2(X), terminalVertex(X) }, numVertices(N).

#minimize { 1,X,Y : p(X,Y) }.
```

Listing 11 Encoding for ST

```
{ in(X) } ← vertex(X).
← edge(X,Y), not in(X), not in(Y).

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

Listing 12 Encoding for cVC

```
in(X) ∨ in(Y) ← edge(X,Y).
```

Listing 13 Encoding for sVC