DynASP2.5: Dynamic Programming on Tree Decompositions in Action[*][†]

Johannes K. Fichte[1], Markus Hecher[2], Michael Morak[3], and Stefan Woltran[4]

1 **Institut für Informationssysteme, TU Wien, Vienna, Austria and Universität Potsdam, Potsdam, Germany**
`fichte@dbai.tuwien.ac.at`

2 **Institut für Informationssysteme, TU Wien, Vienna, Austria and Universität Potsdam, Potsdam, Germany**
`hecher@dbai.tuwien.ac.at`

3 **Institut für Informationssysteme, TU Wien, Vienna, Austria**
`morak@dbai.tuwien.ac.at`

4 **Institut für Informationssysteme, TU Wien, Vienna, Austria**
`woltran@dbai.tuwien.ac.at`

Abstract

Efficient, exact parameterized algorithms are a vibrant theoretical research area. Recent solving competitions, such as the PACE challenge, show that there is also increasing practical interest in the parameterized algorithms community. An important research question is whether such algorithms can be built to efficiently solve specific problems in practice, that is, to be competitive with established solving systems. In this paper, we consider Answer Set Programming (ASP), a logic-based declarative modeling and problem solving framework. State-of-the-art ASP solvers generally rely on SAT-based algorithms. In addition, DynASP2, an ASP solver that is based on a classical dynamic programming on tree decompositions, has recently been introduced. DynASP2 outperforms modern ASP solvers when the goal is to count the number of solutions of programs that have small treewidth. However, for quickly finding one solutions, DynASP2 proved uncompetitive. In this paper, we present a new algorithm and implementation, called DynASP2.5, that shows competitive behavior compared to state-of-the-art ASP solvers on problems like Steiner tree for low-treewidth graphs, even when the task is to find just one solution. Our implementation is based on a novel approach that we call multi-pass dynamic programming.

1998 ACM Subject Classification I.2.8 Problem Solving, Control Methods, and Search, D.1.6 Logic Programming, F.4.1 Mathematical Logic, I.2.3 Deduction and Theorem Proving, I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Parameterized algorithms, Fixed-parameter linear time, Tree decompositions, Multi-pass dynamic programming

Digital Object Identifier 10.4230/LIPIcs.IPEC.2017.17

1 Introduction

Answer set programming (ASP) is a logic-based declarative modeling language and problem solving framework [13], where a program is defined by a set of rules over propositional atoms

[*] Research was supported by the Austrian Science Fund (FWF), Grant Y698.
[†] For an extended version see [9], `https://arxiv.org/abs/1706.09370`.

and is interpreted under an extended stable model semantics [15]. Problems are usually modeled in ASP in such a way that the solutions (called *answer sets*) of a program directly correspond to the solutions of the considered problem instance. Computational problems for disjunctive, propositional ASP, such as deciding whether a program has an answer set, are complete for the second level of the polynomial hierarchy [8]. In consequence, finding answer sets usually involves a Sat part (finding a model of the program) and an Unsat part (minimality check). A variety of ASP solvers based on techniques of Sat solvers are readily available [3, 11] and have proven to be very successful in solving competitions.

Recently, a dynamic programming based solver (*DynASP2*) that builds upon ideas from parameterized algorithms has been proposed [10]. The running time of the underlying algorithm is double exponential in the treewidth of the input program and linear in its size (a so-called *fixed-parameter linear* algorithm). DynASP2 roughly works as follows. First, it computes a tree decomposition of the incidence graph of the given input program. Second, it solves the program via dynamic programming (DP) on the tree decomposition, traversing the tree exactly once bottom-up. Both the Sat and Unsat tasks are considered in a single pass. Once the root node has been reached, complete solutions for the input program can be constructed, if any exist. The exhaustive nature of DP algorithms, where all potential solutions are computed locally for each node of the tree decomposition, works well when all solutions are indeed needed, e.g., for counting answer sets. However, this approach is not competitive when the task is to construct just one answer set, since space requirements can be quite extensive, resulting in long running times. Another downside is the fact that DP algorithms on tree decompositions may exhibit running times that vary considerably, even on tree decompositions of the exact same width [2].

In this paper, we propose a multi-pass algorithm, called $\texttt{M-DP}_{\mathbb{SINC}}$, for dynamic programming on tree decompositions, as well as a new implementation (DynASP2.5). In contrast to classical DP algorithms for problems on the second level of the polynomial hierarchy, $\texttt{M-DP}_{\mathbb{SINC}}$ traverses the given tree decomposition multiple times in a bottom-up fashion. During the first pass, it computes and stores sets of atoms that are relevant for the Sat part (finding a model of the program) up to the root. In the second pass, it computes and stores sets of atoms that are relevant for the Unsat part (checking for minimality). Finally, in the third pass, it links those sets from past two passes together that might lead to an answer set. This allows us to discard candidates that do not lead to answer sets early on.

In addition, we present technical improvements (including working on non-nice tree decompositions) and employ dedicated customization techniques for selecting tree decompositions. These improvements are the main ingredients to speed up the solving process for DP algorithms. Experiments indicate that DynASP2.5 is competitive, even for quickly finding some answer set, when solving the Steiner tree problem on graphs of low treewidth. In particular, DynASP2.5 are able to solve instances that have an upper bound on the incidence treewidth of 14 (whereas DynASP2 could solve instances of treewidth at most 9) on our benchmark set[1].

Contributions. Our main contributions can be summarized as follows:

1. We establish a novel fixed-parameter linear algorithm ($\texttt{M-DP}_{\mathbb{SINC}}$), which works in multiple passes and computes Sat and Unsat parts separately.
2. We present an implementation (DynASP2.5)[2] and an experimental evaluation.

[1] The set is available at `https://github.com/daajoe/dynasp_experiments/tree/ipec2017`.
[2] The sources of our solver are available at `https://github.com/daajoe/dynasp/releases/tag/v2.5.0`.

Related Work. An ASP solver (DynASP2) that is based on single pass DP on tree decompositions was recently introduced [10]. The solver is dedicated to counting answer sets for the full ground ASP language. The present paper extends these results by a multi-pass DP algorithm. In contrast to systems that use encodings in Monadic Second-Order (MSO) [14], our approach directly treats ASP. Bliem et al. [5] introduced a multi-pass approach and an implementation (D-FLATˆ2) for DP on tree decompositions solving subset minimization tasks. Their approach allows to specify DP algorithms by means of ASP. In D-FLATˆ2 one can see ASP as a meta-language to describe what needs to be done at each node of the tree decomposition, whereas our work presents an algorithm dedicated to find some answer set of a program. Further, we require specialized adaptations to the ASP problem semantics, including three valued evaluation of atoms, handling of non-nice tree decompositions, and optimizations in join nodes to be competitive. We use in our solver the heuristic tree decomposition library htd [1]. For other systems we refer to the PACE challenge [7].

2 Formal Background

Tree Decompositions. Let $G = (V, E)$ be a graph, $T = (N, F, n)$ a rooted tree, and $\chi : N \to 2^V$ a function that maps each node $t \in N$ to 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 (TD)* of G if the following conditions hold: (i) for every vertex $v \in V$ there is a node $t \in N$ with $v \in \chi(t)$; (ii) for every edge $e \in E$ there is a node $t \in N$ with $e \subseteq \chi(t)$; and (iii) 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 $\chi(t_1) \cap \chi(t_3) \subseteq \chi(t_2)$. We call $\max\{|\chi(t)| - 1 \mid t \in N\}$ the *width* of the TD. The *treewidth* $tw(G)$ of a graph G is the minimum width over all possible TDs of G. For some arbitrary but fixed integer k and a graph of treewidth at most k, we can compute a TD of width $\leqslant k$ in time $2^{\mathcal{O}(k^3)} \cdot |V|$ [6]. Given a TD (T, χ) with $T = (N, \cdot, \cdot)$, for a node $t \in N$ we say that type(t) is *leaf* if t has no children; *join* if t has children t' and t'' with $t' \neq t''$ and $\chi(t) = \chi(t') = \chi(t'')$; *int* ("introduce") if t has a single child t', $\chi(t') \subseteq \chi(t)$ and $|\chi(t)| = |\chi(t')| + 1$; *rem* ("removal") if t has a single child t', $\chi(t') \supseteq \chi(t)$ and $|\chi(t')| = |\chi(t)| + 1$. If every node $t \in N$ has at most two children, type$(t) \in \{$*leaf*, *join*, *int*, *rem*$\}$, and bags of leaf nodes and the root are empty, then the TD is called *nice*. For every TD, we can compute a nice TD in linear time without increasing the width [6]. Later, we traverse a TD bottom up. Therefore, let post-order(T, t) be the sequence of nodes in post-order of the induced subtree $T' = (N', \cdot, t)$ of T rooted at t.

Answer Set Programming (ASP). ASP is a declarative modeling and problem solving framework that combines techniques of knowledge representation and database theory. Two of the main advantages of ASP are its expressiveness and, when using non-ground programs, its advanced declarative problem modeling capability. Prior to solving, non-ground programs are usually compiled into ground ones by a grounder. There are classes of non-ground programs that preserve the treewidth of the input instance after grounding [4]. In this paper, we restrict ourselves to ground ASP programs. For a comprehensive introduction, see, e.g., [13]. Let ℓ, m, n be non-negative integers such that $\ell \leq m \leq n$, $a_1, \ldots, a_n$ distinct propositional atoms, and $l \in \{a_1, \neg a_1\}$. 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 atom of $a_{m+1}, \ldots, a_n$ is true. A *disjunctive rule* is of the form $a_1 \vee \cdots \vee a_\ell \leftarrow a_{\ell+1}, \ldots, a_m, \neg\, a_{m+1}$, $\ldots$, $\neg a_n$, which, intuitively, means 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 atom of $a_{m+1}, \ldots, a_n$ is true. A *rule* r

is either a disjunctive or a choice rule. Let $H_r := \{a_1, \dots, a_\ell\}$, $B_r^+ := \{a_{\ell+1}, \dots, a_m\}$, and $B_r^- := \{a_{m+1}, \dots, a_n\}$. Usually, for a rule r, if $B_r^- \cup B_r^+ = \emptyset$, we simply write H_r instead of $H_r \leftarrow$. For a rule r, let $\text{at}(r) := H_r \cup B_r^+ \cup B_r^-$ denote its *atoms* and $B_r := B_r^+ \cup \{\neg b \mid b \in B_r^-\}$ its *body*. A *program* P is a set of rules, where $\text{at}(P) := \bigcup_{r \in P} \text{at}(r)$ denotes its atoms. A set $M \subseteq \text{at}(P)$ *satisfies* a rule r if (i) r is a disjunctive rule and $(H_r \cup B_r^-) \cap M \neq \emptyset$ or $B_r^+ \not\subseteq M$ or (ii) r is a choice rule. Note that choice rules are always satisfied. M is a (classical) model of P, denoted by $M \vDash P$, if M satisfies every rule $r \in P$. The *reduct* of a rule r with respect to M, denoted by r^M, is defined (i) for a choice rule r as the set $\{a \leftarrow B_r^+ \mid a \in H_r \cap M, B_r^- \cap M = \emptyset\}$ of rules and (ii) for a disjunctive rule r as the singleton $\{H_r \leftarrow B_r^+ \mid B_r^- \cap M = \emptyset\}$. $P^M := \bigcup_{r \in P} r^M$ is called the *(GL) reduct* of P with respect to M. A set $M \subseteq \text{at}(P)$ is an *answer set* of a program P, if $M \vDash P$ and there does not exist a proper subset $M' \subsetneq M$, such that $M' \vDash P^M$.

▶ **Example 1.** Consider program P, consisting of the following nine rules: $P = \{\overbrace{\{e_{ab}\}}^{r_{ab}};\ \overbrace{\{e_{bc}\}}^{r_{bc}};\ \overbrace{\{e_{cd}\}}^{r_{cd}};\ \overbrace{\{e_{ad}\}}^{r_{ad}};\ \overbrace{a_b \leftarrow e_{ab}}^{r_b};\ \overbrace{a_d \leftarrow e_{ad}}^{r_d};\ \overbrace{a_c \leftarrow a_b, e_{bc}}^{r_{c1}};\ \overbrace{a_c \leftarrow a_d, e_{cd}}^{r_{c2}};\ \overbrace{\leftarrow \neg a_c}^{r_{\neg c}}\}$. The set $A = \{e_{ab}, e_{bc}, a_b, a_c\}$ is an answer set of P, as $\{e_{ab}, e_{bc}, a_b, a_c\}$ is a minimal model of the reduct $P^A = \{e_{ab} \leftarrow;\ e_{bc} \leftarrow;\ a_b \leftarrow e_{ab};\ a_d \leftarrow e_{ad};\ a_c \leftarrow a_b, e_{bc};\ a_c \leftarrow a_d, e_{cd}\}$. Now, consider program $R = \{a \vee c \leftarrow b;\ b \leftarrow c, \neg g;\ c \leftarrow a;\ b \vee c \leftarrow e;\ h \vee i \leftarrow g, \neg c;\ a \vee b;\ g \leftarrow \neg i;\ c;\ \{d\} \leftarrow g\}$. The set $B = \{b, c, d, g\}$ is an answer set of R, since B is a minimal model of the reduct $R^B = \{a \vee c \leftarrow b;\ c \leftarrow a;\ b \vee c \leftarrow e;\ a \vee b;\ g;\ c;\ d \leftarrow g\}$.

In this paper, we mainly consider the *output answer set problem*, that is, output an answer set for an ASP program. The decision version of this problem is Σ_2^p-complete.

Graph Representations of Programs. In order to use TDs for ASP solving, we need dedicated graph representations of programs. The *incidence graph* $I(P)$ of P is the bipartite graph that has the atoms and rules of P as vertices and an edge $a\,r$ if $a \in \text{at}(r)$ for some rule $r \in P$ [10]. The *semi-incidence graph* $S(P)$ of P is a graph that has the atoms and rules of P as vertices and (i) an edge $a\,r$ if $a \in \text{at}(r)$ for some rule $r \in P$ as well as (ii) an edge $a\,b$ for distinct atoms $a, b \in H_r$ where $r \in P$ is a choice rule. Since, for every program P, the incidence graph $I(P)$ is a subgraph of the semi-incidence graph, we have that $tw(I(P)) \leq tw(S(P))$. Further, by definition of TDs and the construction of a semi-incidence graph, head atoms of each choice rule occur together in at least one bag of the TD.

Sub-programs. Let $\mathcal{T} = (T, \chi)$ be a nice TD of the semi-incidence graph $S(P)$ of a program P. Also, let $T = (N, \cdot, n)$ and $t \in N$. The *bag-program* is defined as $P_t := P \cap \chi(t)$. The set $\text{at}_{\leq t} := \{a \mid a \in \text{at}(P) \cap \chi(t'), t' \in \text{post-order}(T, t)\}$ is called *atoms below* t, the *program below* t is defined as $P_{\leq t} := \{r \mid r \in P_{t'}, t' \in \text{post-order}(T, t)\}$, and the *program strictly below* t is $P_{<t} := P_{\leq t} \setminus P_t$. Since $\chi(n) = \emptyset$, it holds that $P_{\leq n} = P_{<n} = P$ and $\text{at}_{\leq n} = \text{at}(P)$.

3 A Single Pass DP Algorithm

DynASP2 [10], a dynamic programming-based ASP solver, splits the input program P into bag-programs based on the structure of a given nice tree decomposition for P and evaluates each bag-program in turn, storing results in tables for each TD node. The algorithm works as shown in Figure 1 (following the red DynASP2 arrow) and executes the following steps:

1. Construct a graph representation $G(P)$ of the given input program P.
2. Compute a TD $\mathcal{T}$ of the graph $G(P)$ by means of some heuristic, thereby decomposing P into several smaller bag-programs and fixing an ordering in which P will be evaluated.

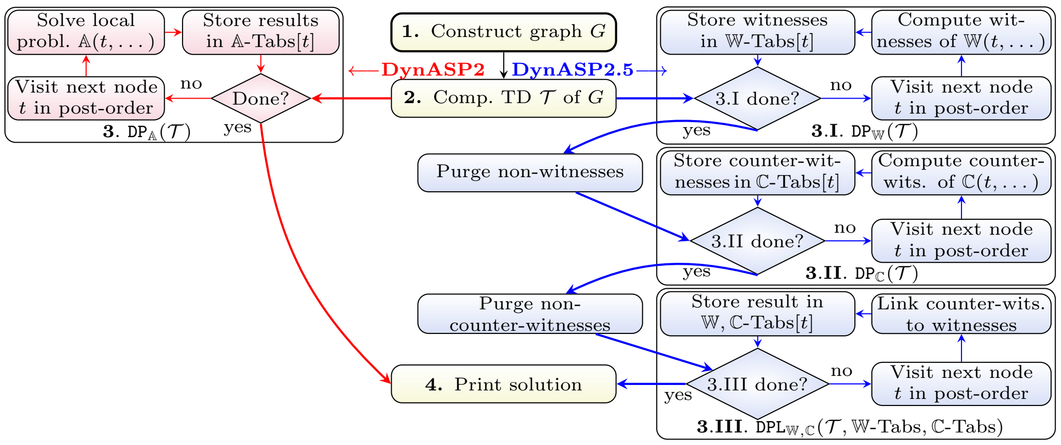

Solve local probl. $\mathbb{A}(t,\ldots)$
Store results in $\mathbb{A}$-Tabs[t]
Visit next node t in post-order
no
Done?
yes
3. $\mathtt{DP}_{\mathbb{A}}(\mathcal{T})$
1. Construct graph G
DynASP2
DynASP2.5
2. Comp. TD $\mathcal{T}$ of G
Purge non-witnesses
Purge non-counter-witnesses
4. Print solution
Store witnesses in $\mathbb{W}$-Tabs[t]
Compute witnesses of $\mathbb{W}(t,\ldots)$
3.I done?
Visit next node t in post-order
3.I. $\mathtt{DP}_{\mathbb{W}}(\mathcal{T})$
Store counter-witnesses in $\mathbb{C}$-Tabs[t]
Compute counter-wits. of $\mathbb{C}(t,\ldots)$
3.II done?
3.II. $\mathtt{DP}_{\mathbb{C}}(\mathcal{T})$
Store result in $\mathbb{W}$, $\mathbb{C}$-Tabs[t]
Link counter-wits. to witnesses
3.III done?
3.III. $\mathtt{DPL}_{\mathbb{W},\mathbb{C}}(\mathcal{T}, \mathbb{W}\text{-Tabs}, \mathbb{C}\text{-Tabs})$

Figure 1 Control flow for DP-based ASP solvers DynASP2 (left) and DynASP2.5 (right).

Listing 1: Algorithm $\mathtt{DP}_{\mathbb{A}}(\mathcal{T})$ for Dynamic Programming on TD $\mathcal{T}$ for ASP [9].

```
In: Table algorithm $\mathbb{A}$, nice TD $\mathcal{T} = (T, \chi)$ with $T = (N, \cdot, n)$ of $G(P)$ according to $\mathbb{A}$.
Out: $\mathbb{A}$-Tabs: maps each TD node $t \in T$ to some computed table $\tau_t$.
for iterate $t$ in post-order$(T,n)$ do
    Child-Tabs$_t$ := $\{\mathbb{A}\text{-Tabs}[t'] \mid t' \text{ is a child of } t \text{ in } T\}$
    $\mathbb{A}$-Tabs[$t$] ← $\mathbb{A}(t, \chi(t), P_t, \mathrm{at}_{\leq t}, \text{Child-Tabs}_t)$
```

3. Algorithm $\mathtt{DP}_{\mathbb{A}}(\mathcal{T})$ (see Listing 1 above) specifies the general scheme for this step, assuming that an algorithm $\mathbb{A}$, which strongly depends on the graph representation, is given. Such an algorithm $\mathbb{A}$ is called a *table algorithm* that specifies, how the individual bag-programs for each tree node are evaluated. $\mathtt{DP}_{\mathbb{A}}(\mathcal{T})$ works as follows: Traverse the tree decomposition $\mathcal{T}$ in post-order. For every node $t \in T$ in the tree decomposition $\mathcal{T} = ((T, E, n), \chi)$, run $\mathbb{A}$ to compute the table for node t (denoted $\mathbb{A}$-Tabs[t]). Intuitively, each tuple, which we refer to as *row*, in the table represents a witness for the existence of a solution for the bag-program at node t. $\mathbb{A}$-Tabs[t] is computed by taking, as input, the tables of the child nodes of t, and extending them according to the bag-program P_t. Each row in $\mathbb{A}$-Tabs[t] consists of a *witness set* (a set of atoms relevant for the SAT part of the problem), and a family of *counter-witness sets* (sets of atoms relevant for the UNSAT part) [10]. This directly follows the definition of answer sets, namely, being models of P and subset-minimal with respect to P^M.
4. For root node n, check if a "solution row" exists in table $\mathbb{A}$-Tabs[n] and print the solution to the output ASP problem.

With the above general algorithm in mind, we are now ready to propose $\mathbb{SINC}$, a new table algorithm for solving ASP on the semi-incidence graph (see Listing 2). $\mathtt{DP}_{\mathbb{SINC}}$ merges two earlier algorithms for the primal and incidence graph [10].

As in the general approach, $\mathbb{SINC}$ computes and stores witness sets, and their corresponding counter-witness sets. However, in addition, for each witness set and counter-witness set, respectively, we need to store so-called *satisfiability states* (or *sat-states*, for short), since the atoms of a rule may no longer be contained in one single bag of the TD of the semi-incidence graph. Therefore, we need to remember in each TD node, how much of a rule is already satisfied. The following describes this in more detail.

Listing 2: Table algorithm $\mathbb{SINC}(t, \chi_t, P_t, \mathrm{at}_{\leq t}, \text{Child-Tabs}_t)$.

```
   In: Bag $\chi_t$, bag-program $P_t$, atoms-below $\mathrm{at}_{\leq t}$, child tables $\text{Child-Tabs}_t$ of $t$. Out: Tab. $\tau_t$.
1  if $\mathrm{type}(t) = \mathit{leaf}$ then  $\tau_t \leftarrow \{\langle \emptyset, \emptyset, \emptyset \rangle\}$ ;                /* For Abbreviations see below.  */
2  else if $\mathrm{type}(t) = \mathit{int}$, $a \in \chi_t \setminus P_t$ is introduced and $\tau' \in$ Child-Tabs$_t$ then
3      $\tau_t \leftarrow \{\langle M_a^+, \sigma \cup \mathrm{SatPr}(\dot{P}_t^{(t)}, M_a^+), \{\langle C_a^+, \rho \cup \mathrm{SatPr}(\dot{P}_t^{(t,M_a^+)}, C_a^+)\rangle \mid \langle C, \rho\rangle \in \mathcal{C}\} \cup$
4        $\{\langle C, \rho \cup \mathrm{SatPr}(\dot{P}_t^{(t,M_a^+)}, C)\rangle \mid \langle C, \rho\rangle \in \mathcal{C}\} \cup \{\langle M, \sigma \cup \mathrm{SatPr}(\dot{P}_t^{(t,M_a^+)}, M)\rangle\}\rangle \mid \langle M, \sigma, \mathcal{C}\rangle \in \tau'\}$
5        $\cup\ \{\langle M, \sigma \cup \mathrm{SatPr}(\dot{P}_t^{(t)}, M), \{\langle C, \rho \cup \mathrm{SatPr}(\dot{P}_t^{(t,M)}, C)\rangle \mid \langle C, \rho\rangle \in \mathcal{C}\}\ \rangle \mid \langle M, \sigma, \mathcal{C}\rangle \in \tau'\}$
6  else if $\mathrm{type}(t) = \mathit{int}$, $r \in \chi_t \cap P_t$ is introduced and $\tau' \in$ Child-Tabs$_t$ then
7      $\tau_t \leftarrow \{\langle M, \sigma \cup \mathrm{SatPr}(\{\dot{r}\}^{(t)}, M), \{\langle C, \rho \cup \mathrm{SatPr}(\{\dot{r}\}^{(t,M)}, C)\rangle \mid \langle C, \rho\rangle \in \mathcal{C}\}\rangle \mid \langle M, \sigma, \mathcal{C}\rangle \in \tau'\}$
8  else if $\mathrm{type}(t) = \mathit{rem}$, $a \notin \chi_t$ is removed atom and $\tau' \in$ Child-Tabs$_t$ then
9      $\tau_t \leftarrow \{\langle M_a^-, \sigma, \{\langle C_a^-, \rho\rangle \mid \langle C, \rho\rangle \in \mathcal{C}\}\rangle \mid \langle M, \sigma, \mathcal{C}\rangle \in \tau'\}$
10 else if $\mathrm{type}(t) = \mathit{rem}$, $r \notin \chi_t$ is removed rule and $\tau' \in$ Child-Tabs$_t$ then
11     $\tau_t \leftarrow \{\langle M, \sigma_r^-, \big\{\langle C, \rho_r^-\rangle \mid \langle C, \rho\rangle \in \mathcal{C}, r \in \rho\big\}\rangle \mid \langle M, \sigma, \mathcal{C}\rangle \in \tau', r \in \sigma\}$
12 else if $\mathrm{type}(t) = \mathit{join}$ and $\tau', \tau'' \in$ Child-Tabs$_t$ with $\tau' \neq \tau''$ then
13     $\tau_t \leftarrow \{\langle M, \sigma' \cup \sigma'', \{\langle C, \rho' \cup \rho''\rangle \mid \langle C, \rho'\rangle \in \mathcal{C}', \langle C, \rho''\rangle \in \mathcal{C}''\} \cup \{\langle M, \rho \cup \sigma''\rangle \mid \langle M, \rho\rangle \in \mathcal{C}'\} \cup$
14       $\{\langle M, \sigma' \cup \rho\rangle \mid \langle M, \rho\rangle \in \mathcal{C}''\}\rangle \mid \langle M, \sigma', \mathcal{C}'\rangle \in \tau', \langle M, \sigma'', \mathcal{C}''\rangle \in \tau''\}$
```

For set S and element s, we denote $S_s^+ \leftarrow S \cup \{s\}$ and $S_s^- \leftarrow S \setminus \{s\}$.

By definition of TDs and the semi-incidence graph, for every atom a and every rule r of a program, it is true that if atom a occurs in rule r, then a and r occur together in at least one bag of the TD. As a consequence, the table algorithm encounters every occurrence of an atom in any rule. In the end, on removal of r, we have to ensure that r is among the rules that are already satisfied. However, we need to keep track of whether a witness set satisfies a rule, because not all atoms that occur in a rule occur together in a bag. Hence, when our algorithm traverses the TD and an atom is removed we still need to store this sat-state, as setting this atom to a certain truth value influences the satisfiability of the rule. Since the semi-incidence graph contains a clique on every set A of atoms that occur together in a choice rule head, those atoms A occur together in a bag in every TD of the semi-incidence graph. For that reason, we do not need to incorporate choice rules into the satisfiability state, in contrast to the algorithm for the incidence graph [10].

In Algorithm $\mathbb{SINC}$ (detailed in Listing 2), a row u in the table τ_t is a triple $\langle M, \sigma, \mathcal{C}\rangle$. The set $M \subseteq \mathrm{at}(P) \cap \chi(t)$ represents a witness set. The family $\mathcal{C}$ of rows represents counter-witnesses, which we will discuss in more detail below. The sat-state σ for M represents rules of $\chi(t)$ satisfied by a superset of M. Hence, M witnesses a model $M' \supseteq M$ where $M' \vDash P_{<t} \cup \sigma$. For that reason, a witness set together with its sat-state is called a *witness*. We use the binary operator $\cup$ to combine sat-states, which ensures that rules satisfied in at least one operand remain satisfied. For a node t, our algorithm considers a local-program depending on the bag $\chi(t)$. Intuitively, this provides a local view on the program.

▶ **Definition 2.** Let P be a program, $\mathcal{T} = (\cdot, \chi)$ a TD of $S(P)$, t a node of $\mathcal{T}$ and $R \subseteq P_t$. The *local-program* $R^{(t)}$ is obtained from $R \cup \{\leftarrow B_r \mid r \in R$ is a choice rule, $H_r \subsetneq \mathrm{at}_{<t}\}$[3] by removing all literals a and $\neg a$ from every rule where $a \notin \chi(t)$.

▶ **Example 3.** Observe $P_{t_4}^{(t_4)} = \{\leftarrow e_{bc}, r_b\}$ and $P_{t_5}^{(t_5)} = \{c \leftarrow\}$ for P_{t_4} and P_{t_5} of Figure 2.

Since the local-program $P^{(t)}$ depends on the considered node t, we may have different local-programs for for node t and its child t'. In particular, the programs $\{r\}^{(t)}$ and $\{r\}^{(t')}$ might already differ for a rule $r \in \chi(t) \cap \chi(t')$. In consequence for satisfiability with respect

[3] We require to add $\{\leftarrow B_r \mid r \in R$ is a choice rule, $H_r \subsetneq \mathrm{at}_{<t}\}$ in order to decide satisfiability for corner cases of choice rules involving counter-witnesses of Line 3 in Listing 2.

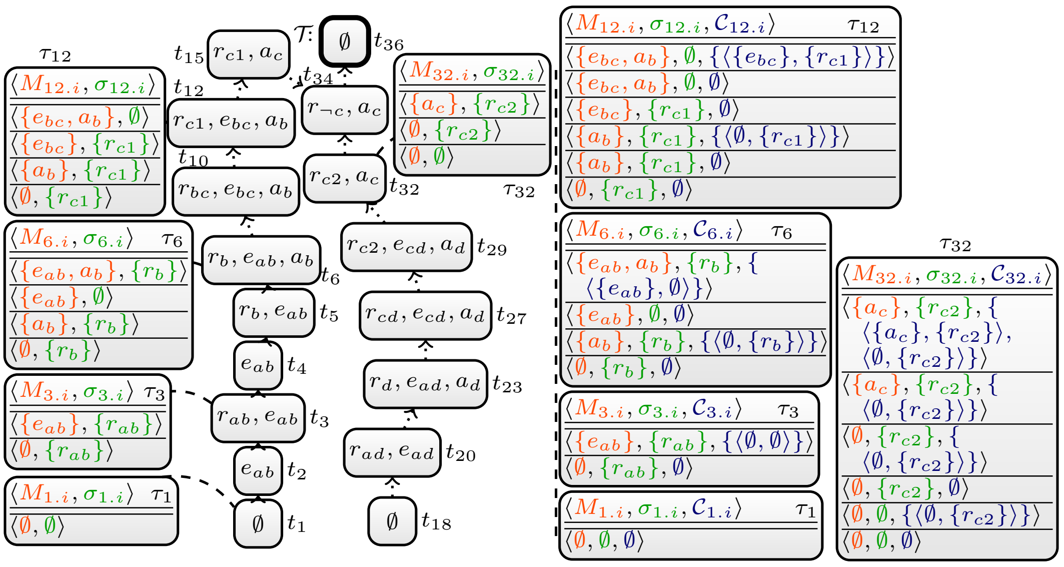

$\mathcal{T}$:
$\emptyset$ t_{36}
r_{c1}, a_c t_{15}
$r_{\neg c}, a_c$ t_{34}
r_{c1}, e_{bc}, a_b t_{12}
r_{bc}, e_{bc}, a_b t_{10}
r_{c2}, a_c t_{32}
r_b, e_{ab}, a_b t_6
r_{c2}, e_{cd}, a_d t_{29}
r_b, e_{ab} t_5
r_{cd}, e_{cd}, a_d t_{27}
e_{ab} t_4
r_d, e_{ad}, a_d t_{23}
r_{ab}, e_{ab} t_3
r_{ad}, e_{ad} t_{20}
e_{ab} t_2
$\emptyset$ t_1
$\emptyset$ t_{18}
τ_{12}
$\langle M_{12.i}, \sigma_{12.i}\rangle$
$\langle\{e_{bc}, a_b\}, \emptyset\rangle$
$\langle\{e_{bc}\}, \{r_{c1}\}\rangle$
$\langle\{a_b\}, \{r_{c1}\}\rangle$
$\langle\emptyset, \{r_{c1}\}\rangle$
$\langle M_{6.i}, \sigma_{6.i}\rangle$ τ_6
$\langle\{e_{ab}, a_b\}, \{r_b\}\rangle$
$\langle\{e_{ab}\}, \emptyset\rangle$
$\langle\{a_b\}, \{r_b\}\rangle$
$\langle\emptyset, \{r_b\}\rangle$
$\langle M_{3.i}, \sigma_{3.i}\rangle$ τ_3
$\langle\{e_{ab}\}, \{r_{ab}\}\rangle$
$\langle\emptyset, \{r_{ab}\}\rangle$
$\langle M_{1.i}, \sigma_{1.i}\rangle$ τ_1
$\langle\emptyset, \emptyset\rangle$
$\langle M_{32.i}, \sigma_{32.i}\rangle$ τ_{32}
$\langle\{a_c\}, \{r_{c2}\}\rangle$
$\langle\emptyset, \{r_{c2}\}\rangle$
$\langle\emptyset, \emptyset\rangle$
$\langle M_{12.i}, \sigma_{12.i}, \mathcal{C}_{12.i}\rangle$ τ_{12}
$\langle\{e_{bc}, a_b\}, \emptyset, \{\langle\{e_{bc}\}, \{r_{c1}\}\rangle\}\rangle$
$\langle\{e_{bc}, a_b\}, \emptyset, \emptyset\rangle$
$\langle\{e_{bc}\}, \{r_{c1}\}, \emptyset\rangle$
$\langle\{a_b\}, \{r_{c1}\}, \{\langle\emptyset, \{r_{c1}\}\rangle\}\rangle$
$\langle\{a_b\}, \{r_{c1}\}, \emptyset\rangle$
$\langle\emptyset, \{r_{c1}\}, \emptyset\rangle$
$\langle M_{6.i}, \sigma_{6.i}, \mathcal{C}_{6.i}\rangle$ τ_6
$\langle\{e_{ab}, a_b\}, \{r_b\}, \{\langle\{e_{ab}\}, \emptyset\rangle\}\rangle$
$\langle\{e_{ab}\}, \emptyset, \emptyset\rangle$
$\langle\{a_b\}, \{r_b\}, \{\langle\emptyset, \{r_b\}\rangle\}\rangle$
$\langle\emptyset, \{r_b\}, \emptyset\rangle$
$\langle M_{3.i}, \sigma_{3.i}, \mathcal{C}_{3.i}\rangle$ τ_3
$\langle\{e_{ab}\}, \{r_{ab}\}, \{\langle\emptyset, \emptyset\rangle\}\rangle$
$\langle\emptyset, \{r_{ab}\}, \emptyset\rangle$
$\langle M_{1.i}, \sigma_{1.i}, \mathcal{C}_{1.i}\rangle$ τ_1
$\langle\emptyset, \emptyset, \emptyset\rangle$
τ_{32}
$\langle M_{32.i}, \sigma_{32.i}, \mathcal{C}_{32.i}\rangle$
$\langle\{a_c\}, \{r_{c2}\}, \{\langle\{a_c\}, \{r_{c2}\}\rangle, \langle\emptyset, \{r_{c2}\}\rangle\}\rangle$
$\langle\{a_c\}, \{r_{c2}\}, \{\langle\emptyset, \{r_{c2}\}\rangle\}\rangle$
$\langle\emptyset, \{r_{c2}\}, \{\langle\emptyset, \{r_{c2}\}\rangle\}\rangle$
$\langle\emptyset, \{r_{c2}\}, \emptyset\rangle$
$\langle\emptyset, \emptyset, \{\langle\emptyset, \{r_{c2}\}\rangle\}\rangle$
$\langle\emptyset, \emptyset, \emptyset\rangle$

Figure 2 A TD $\mathcal{T}$ of the semi-incidence graph $S(P)$ for program P from Example 1 (center). Selected DP tables after $\texttt{DP}_{\mathbb{MOD}}$ (left) and after $\texttt{DP}_{\mathbb{SINC}}$ (right) for nice TD $\mathcal{T}$.

to sat-states, we need to keep track of a representative of a rule. We achieve this by a function $\dot{R}^{(t)} : R \rightarrow 2^{R^{(t)}}$ that maps a rule in $R \subseteq P_t$ for bag-program P_t to its local-program, i.e., $\dot{R}^{(t)}(r) := \{r\}^{(t)}$ for $r \in R$. When we compute newly satisfied rules for witness set M and a set R of rules, we use the function $\dot{R}^{(t)}$. Formally, $\text{SatPr}(\dot{R}^{(t)}, M) := \{r \mid (r, S) \in \dot{R}^{(t)}, M \models S\}$ for $M \subseteq \chi(t) \setminus P_t$ using program $S = \dot{R}^{(t)}(r)$ constructed by $\dot{R}^{(t)}$ and $r \in R$.

Example 4 provides an explanation of the part of $\mathbb{SINC}$ that deals with witnesses only. Therefore, the resulting algorithm $\mathbb{MOD}$ computes only models and is obtained from $\mathbb{SINC}$, by taking only the first two row positions into account (red and green text in Listing 2). The remaining position (blue text) can be seen as an algorithm $\mathbb{CMOD}$ that computes counter-witnesses (see [9, Ex. 4]). Note that we discuss selected cases, and we assume that each row in a table τ_t is identified by a number, i.e., row i corresponds to $u_{t.i} = \langle M_{t.i}, \sigma_{t.i}\rangle$.

▶ **Example 4.** Consider program P from Example 1, TD $\mathcal{T} = (\cdot, \chi)$ in Figure 2, and the tables $\tau_1, \ldots, \tau_{34}$, which illustrate computation results obtained during post-order traversal of $\mathcal{T}$ by $\texttt{DP}_{\mathbb{MOD}}$. Figure 2 (left) does not show every intermediate node of TD $\mathcal{T}$. Table $\tau_1 = \{\langle\emptyset, \emptyset\rangle\}$ as type$(t_1) = $ *leaf* (see Line 1 in Listing 2). Table τ_3 is obtained via introducing rule r_{ab}, after introducing atom e_{ab} (type$(t_2) = $ type$(t_3) = $ *int*). It contains two rows due to two possible truth assignments using atom e_{ab} (Line 3–5). Observe that rule r_{ab} is satisfied in both rows $M_{3.1}$ and $M_{3.2}$, since the head of choice rule r_{ab} is in $\text{at}_{\leq t_3}$ (see Line 7 and Definition 2). Intuitively, whenever a rule r is proven to be satisfied, sat-state $\sigma_{t.i}$ marks r as *satisfied* since an atom of a rule of $S(P)$ might only occur in one TD bag. Consider table τ_4 with type$(t_4) = $ *rem* and $r_{ab} \in \chi(t_3) \setminus \chi(t_4)$. By definition of TDs of $S(P)$, we have encountered every occurrence of any atom in r_{ab}. Thus, $\mathbb{MOD}$ enforces that only rows where r_{ab} is marked satisfied in τ_3, are considered for table τ_4. The resulting table τ_4 consists of rows of τ_3 with $\sigma_{4.i} = \emptyset$, where rule r_{ab} is proven satisfied ($r_{ab} \in \sigma_{3.1}, \sigma_{3.2}$, see Line 11). Note that between nodes t_6 and t_{10}, an atom and rule remove as well as an atom and rule introduce node is placed. Observe that the second row $u_{6.2} = \langle M_{6.2}, \sigma_{6.2}\rangle \in \tau_6$ does not have a "successor row" in τ_{10}, since $r_b \notin \sigma_{6.2}$. Intuitively, join node t_{34} joins only common witness sets in τ_{17} and τ_{33} with $\chi(t_{17}) = \chi(t_{33}) = \chi(t_{34})$. In general, a join node marks rules satisfied, which are marked satisfied in at least one child (see Line 13–14).

Listing 3: Algorithm $\texttt{DPL}_{\mathbb{W},\mathbb{C}}(\mathcal{T}, \mathbb{W}\text{-Tabs}, \mathbb{C}\text{-Tabs})$ for linking counter-witnesses to witnesses.

```
In:  Nice TD $\mathcal{T} = (T, \chi)$ with $T = (N, \cdot, n)$ of a graph $S(P)$, mappings $\mathbb{W}$-Tabs[·], $\mathbb{C}$-Tabs[·].
Out: $\mathbb{W}, \mathbb{C}$-Tabs: maps $t \in T$ to some pair $(\tau_t^{\mathbb{W}}, \tau_t^{\mathbb{C}})$ with $\tau_t^{\mathbb{W}} \in \mathbb{W}\text{-Tabs}[t], \tau_t^{\mathbb{C}} \in \mathbb{C}\text{-Tabs}[t]$.
$\text{Child-Tabs}_t := \{\mathbb{W}, \mathbb{C}\text{-Tabs}[t'] \mid t' \text{ is a child of } t \text{ in } T\}$
/* Get for a node $t$ tables of (preceding) combined child rows (CCR)           */
$\text{CCR}_t := \hat{\Pi}_{\tau' \in \text{Child-Tabs}_t} \tau'$                      /* For Abbreviations see below.  */
/* Get for a row $\vec{u}$ its combined child rows (origins)                         */
$\text{orig}_t(\vec{u}) := \{S \mid S \in \text{CCR}_t, \vec{u} \in \tau, \tau = \mathbb{W}(t, \chi(t), P_t, \text{at}_{\leq t}, f_w(S))\}$
/* Get for a table $S$ of combined child rows its successors (evolution)            */
$\text{evol}_t(S) := \{\vec{u} \mid \vec{u} \in \tau, \tau = \mathbb{C}(t, \chi(t), P_t, \text{at}_{\leq t}, \tau'), \tau' \in S\}$
for iterate $t$ in post-order(T,n) do
    /* Compute counter-witnesses ($\prec$-smaller rows) for a witness set $M$          */
    $\text{subs}_{\prec}(f, M, S) := \{\vec{u} \mid \vec{u} \in \mathbb{C}\text{-Tabs}[t], \vec{u} \in \text{evol}_t(f(S)), \vec{u} = \langle C, \cdots \rangle, C \prec M\}$
    /* Link each witness $\vec{u}$ to its counter-witnesses and store the results       */
    $\mathbb{W}, \mathbb{C}\text{-Tabs}[t] \leftarrow \{(\vec{u}, \text{subs}_{\subsetneq}(f_w, M, S) \cup \text{subs}_{\subseteq}(f_{cw}, M, S))$
                                      $\mid \vec{u} \in \mathbb{W}\text{-Tabs}[t], \vec{u} = \langle M, \cdots \rangle, S \in \text{orig}_t(u)\}$
```

For set $I = \{1, \ldots, n\}$ and sets S_i, we define $\Pi_{i \in I} S_i := S_1 \times \cdots \times S_n = \{(s_1, \ldots, s_n) : s_i \in S_i\}$. Moreover, for $\Pi_{i \in I} S_i$, let $\hat{\Pi}_{i \in I} S_i := \{\{\{s_1\}, \ldots, \{s_n\}\} \mid (s_1, \ldots, s_n) \in \Pi_{i \in I} S_i\}$. If for each $S \in \hat{\Pi}_{i \in I} S_i$ and $\{s_i\} \in S$, s_i is a pair with a witness and a counter-witness part, let $f_w(S) := \bigcup_{\{(W_i, C_i)\} \in S} \{\{W_i\}\}$ and $f_{cw}(S) := \bigcup_{\{(W_i, C_i)\} \in S} \{\{C_i\}\}$ restrict S to the witness or counter-witness parts, respectively.

Since we already explained how to obtain models, we only briefly describe how to compute counter-witnesses. Family $\mathcal{C}$ consists of rows (C, ρ), where $C \subseteq \text{at}(P) \cap \chi(t)$ is a counter-witness set to M. Similar to the sat-state σ, the sat-state ρ for C under M represents whether rules of the GL reduct P_t^M are satisfied by a superset of C. A counter-witness set together with its sat-state is called a *counter-witness*. Thus, C witnesses the existence of $C' \subsetneq M'$ satisfying $C' \models (P_{<t} \cup \rho)^{M'}$ since M witnesses a model $M' \supseteq M$ where $M' \models P_{<t}$. In consequence, there exists an answer set of P if the root table contains $\langle \emptyset, \emptyset, \emptyset \rangle$.

In order to decide the satisfiability of counter-witness sets, we require local-reducts similar to local-programs (see Definition 2 and below).

▶ **Definition 5.** Let P be a program, $\mathcal{T} = (\cdot, \chi)$ be a TD of $S(P)$, t be a node of $\mathcal{T}$, $R \subseteq P_t$ and $M \subseteq \text{at}(P)$. We define *local-reduct* $R^{(t,M)}$ as $[R^{(t)}]^M$ and $\dot{R}^{(t,M)} : R \to 2^{R^{(t,M)}}$ as $\dot{R}^{(t,M)}(r) := \{r\}^{(t,M)}$ for any $r \in R$.

Note that one can now easily refine $\text{SatPr}(\cdot, \cdot)$ such that it takes as first argument arbitrary functions mapping from rules to programs. In particular, one can then pass the function $\dot{R}^{(t,M)}$ for a set R of rules, and a witness set $M \subseteq \chi(t) \setminus P_t$, as used in Listing 2.

▶ **Proposition 6** ($\star$, c.f. [10]). *Let P be a program and $k := tw(S(P))$. Then, the algorithm* $\texttt{DP}_{\mathbb{SINC}}$ *runs in time $\mathcal{O}(2^{2^{k+2}} \cdot \|S(P)\|)$ and is correct.*

4 DynASP2.5: Implementing a III Pass DP Algorithm

The classical DP algorithm $\texttt{DP}_{\mathbb{SINC}}$ (Step 3 of Figure 1) follows a single pass approach. It computes both witnesses and counter-witnesses in one step by traversing the given TD exactly once. In particular, it exhaustively stores all potential counter-witnesses, even those where the associated witness does not lead to a solution at the root node. In addition, there

[5] Due to space limitations, proofs of statements marked with "$\star$" have been omitted.

can be a high number of duplicates among the counter-witnesses, which are stored separately. In this section, we propose a multi-pass algorithm, $\texttt{M-DP}_{\mathbb{SINC}}$, for DP on TDs and a new implementation (DynASP2.5), which tackles this issue by adapting and extending concepts for DP on TDs presented in [5]. Our novel algorithm allows for an early cleanup (purging) of witnesses that do not lead to answer sets. As a consequence, this (i) avoids to construct expendable counter-witnesses. Moreover, multiple passes enable us to store witnesses and counter-witnesses separately which, in turn, (ii) avoids storing duplicates of counter-witnesses and (iii) allows for highly space-efficient data structures (pointers) in practice when linking witnesses and counter-witnesses together. Figure 1 (following the blue arrows) presents the control flow of the new multi-pass approach *DynASP2.5*, where $\texttt{M-DP}_{\mathbb{SINC}}$ introduces a much more elaborate computation in Step 3 (cf. Figure 1).

4.1 The Algorithm

Algorithm $\texttt{M-DP}_{\mathbb{SINC}}$ executed as Step 3 runs $\texttt{DP}_{\mathbb{MOD}}$, $\texttt{DP}_{\mathbb{CMOD}}$ and new Algorithm $\texttt{DPL}_{\mathbb{MOD},\mathbb{CMOD}}$ in three respective passes (3.I, 3.II, and 3.III) as follows:

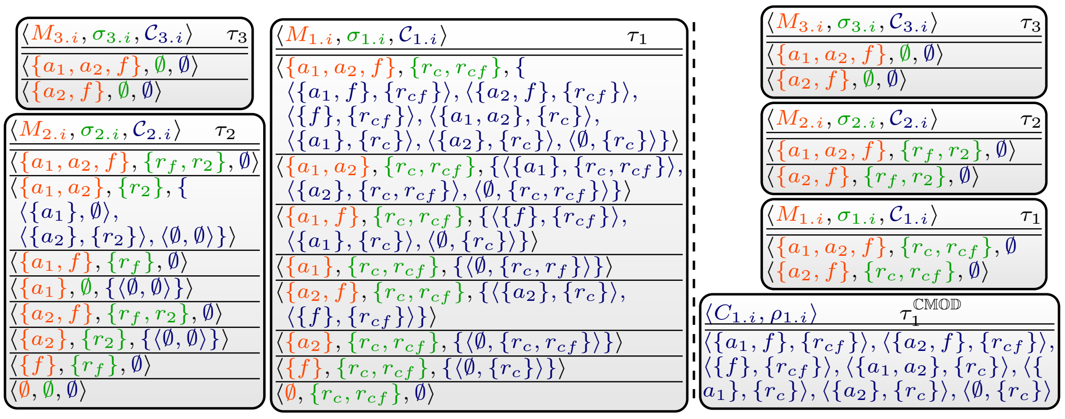
3.I. First, we run the algorithm $\texttt{DP}_{\mathbb{MOD}}$, which computes, via a bottom-up traversal, a table $\mathbb{MOD}$-Tabs$[t]$ of witnesses for every node t in the tree decomposition. Then, via a top-down traversal, it purges those witnesses from tables $\mathbb{MOD}$-Tabs$[t]$ that do not extend to a witness in the table for the parent node; these witnesses can never be used to construct a model (nor answer set) of the program.

3.II. For this step, let $\mathbb{CMOD}$ be a table algorithm computing only counter-witnesses of $\mathbb{SINC}$ (blue parts of Listing 2). We execute $\texttt{DP}_{\mathbb{CMOD}}$, which computes all counter-witnesses for all the witnesses at once and stores the resulting tables in $\mathbb{CMOD}$-Tabs$[\cdot]$. For every node t, table $\mathbb{CMOD}$-Tabs$[t]$ contains possible counter-witnesses for subset-minimality. Again, irrelevant rows are purged.

3.III. Finally, via a bottom-up traversal, for every node t in the TD, witnesses and counter-witnesses are linked using algorithm $\texttt{DPL}_{\mathbb{MOD},\mathbb{CMOD}}$ (see Listing 3). $\texttt{DPL}_{\mathbb{MOD},\mathbb{CMOD}}$ takes previous results and maps rows in $\mathbb{MOD}$-Tabs$[t]$ to a set of rows in $\mathbb{CMOD}$-Tabs$[t]$.

We already explained the table algorithms $\texttt{DP}_{\mathbb{MOD}}$ and $\texttt{DP}_{\mathbb{CMOD}}$ in the previous section. The main part of our multi-pass algorithm is the algorithm $\texttt{DPL}_{\mathbb{MOD},\mathbb{CMOD}}$ based on the general algorithm $\texttt{DPL}_{\mathbb{W},\mathbb{C}}$ (Listing 3) with $\mathbb{W} = \mathbb{MOD}$, $\mathbb{C} = \mathbb{CMOD}$, which links those separate tables together. Before we quickly discuss the core of $\texttt{DPL}_{\mathbb{W},\mathbb{C}}$ in Line 5–7, note that Line 2–4 introduce auxiliary definitions. Line 2 combines rows of the child nodes of given node t, which is achieved by a product over sets where we drop the order and keep sets only. For a row $\vec{u}$, Line 3 determines its preceding combined rows that lead to $\vec{u}$, using table algorithm $\mathbb{W}$. Via algorithm $\mathbb{C}$, Line 4 derives the succeeding rows (called *evolution* rows) of a certain child row combination τ' (called *origin* row). In the actual implementation, origin and evolution rows are not computed, but represented via pointer data structures, directly linking to $\mathbb{W}$-Tabs$[\cdot]$ and $\mathbb{C}$-Tabs$[\cdot]$, respectively. Then, the table algorithm $\texttt{DPL}_{\mathbb{W},\mathbb{C}}$ applies a post-order traversal and links witnesses to counter-witnesses in Line 7. $\texttt{DPL}_{\mathbb{W},\mathbb{C}}$ searches for origins (orig) of a witness $\vec{u}$, uses the counter-witnesses (f_{cw}) linked to these origins, and then determines the evolution (procedure evol) in order to derive counter-witnesses (procedure subs) of $\vec{u}$.

▶ **Example 7.** Let k be some integer and P_k be the program that consists of the rules $r_c := \{a_1, \cdots, a_k\} \leftarrow f$, $r_2 := \leftarrow \neg a_2$, …, $r_k := \leftarrow \neg a_k$, and $r_f := \leftarrow \neg f$ and $r_{cf} := \{f\} \leftarrow$. The rules $r_2, \ldots, r_k$ simulate that only specific subsets of $\{a_1, \cdots, a_k\}$ are allowed. Rules r_f and r_{cf} enforce that f is set to true. Let $\mathcal{T} = (T, \chi, t_3)$ be a TD of the semi-incidence graph $S(P_k)$ of program P_k where $T = (V, E)$ with $V = \{t_1, t_2, t_3\}$, $E = \{(t_1, t_2), (t_2, t_3)\}$,



$\langle M_{3.i}, \sigma_{3.i}, \mathcal{C}_{3.i}\rangle$ τ_3
$\langle\{a_1, a_2, f\}, \emptyset, \emptyset\rangle$
$\langle\{a_2, f\}, \emptyset, \emptyset\rangle$

$\langle M_{2.i}, \sigma_{2.i}, \mathcal{C}_{2.i}\rangle$ τ_2
$\langle\{a_1, a_2, f\}, \{r_f, r_2\}, \emptyset\rangle$
$\langle\{a_1, a_2\}, \{r_2\}, \{\langle\{a_1\}, \emptyset\rangle, \langle\{a_2\}, \{r_2\}\rangle, \langle\emptyset, \emptyset\rangle\}\rangle$
$\langle\{a_1, f\}, \{r_f\}, \emptyset\rangle$
$\langle\{a_1\}, \emptyset, \{\langle\emptyset, \emptyset\rangle\}\rangle$
$\langle\{a_2, f\}, \{r_f, r_2\}, \emptyset\rangle$
$\langle\{a_2\}, \{r_2\}, \{\langle\emptyset, \emptyset\rangle\}\rangle$
$\langle\{f\}, \{r_f\}, \emptyset\rangle$
$\langle\emptyset, \emptyset, \emptyset\rangle$

$\langle M_{1.i}, \sigma_{1.i}, \mathcal{C}_{1.i}\rangle$ τ_1
$\langle\{a_1, a_2, f\}, \{r_c, r_{cf}\}, \{\langle\{a_1, f\}, \{r_{cf}\}\rangle, \langle\{a_2, f\}, \{r_{cf}\}\rangle, \langle\{f\}, \{r_{cf}\}\rangle, \langle\{a_1, a_2\}, \{r_c\}\rangle, \langle\{a_1\}, \{r_c\}\rangle, \langle\{a_2\}, \{r_c\}\rangle, \langle\emptyset, \{r_c\}\rangle\}\rangle$
$\langle\{a_1, a_2\}, \{r_c, r_{cf}\}, \{\langle\{a_1\}, \{r_c, r_{cf}\}\rangle, \langle\{a_2\}, \{r_c, r_{cf}\}\rangle, \langle\emptyset, \{r_c, r_{cf}\}\rangle\}\rangle$
$\langle\{a_1, f\}, \{r_c, r_{cf}\}, \{\langle\{f\}, \{r_{cf}\}\rangle, \langle\{a_1\}, \{r_c\}\rangle, \langle\emptyset, \{r_c\}\rangle\}\rangle$
$\langle\{a_1\}, \{r_c, r_{cf}\}, \{\langle\emptyset, \{r_c, r_f\}\rangle\}\rangle$
$\langle\{a_2, f\}, \{r_c, r_{cf}\}, \{\langle\{a_2\}, \{r_c\}\rangle, \langle\{f\}, \{r_{cf}\}\rangle\}\rangle$
$\langle\{a_2\}, \{r_c, r_{cf}\}, \{\langle\emptyset, \{r_c, r_{cf}\}\rangle\}\rangle$
$\langle\{f\}, \{r_c, r_{cf}\}, \{\langle\emptyset, \{r_c\}\rangle\}\rangle$
$\langle\emptyset, \{r_c, r_{cf}\}, \emptyset\rangle$

$\langle M_{3.i}, \sigma_{3.i}, \mathcal{C}_{3.i}\rangle$ τ_3
$\langle\{a_1, a_2, f\}, \emptyset, \emptyset\rangle$
$\langle\{a_2, f\}, \emptyset, \emptyset\rangle$

$\langle M_{2.i}, \sigma_{2.i}, \mathcal{C}_{2.i}\rangle$ τ_2
$\langle\{a_1, a_2, f\}, \{r_f, r_2\}, \emptyset\rangle$
$\langle\{a_2, f\}, \{r_f, r_2\}, \emptyset\rangle$

$\langle M_{1.i}, \sigma_{1.i}, \mathcal{C}_{1.i}\rangle$ τ_1
$\langle\{a_1, a_2, f\}, \{r_c, r_{cf}\}, \emptyset$
$\langle\{a_2, f\}, \{r_c, r_{cf}\}, \emptyset\rangle$

$\langle C_{1.i}, \rho_{1.i}\rangle$ $\tau_1^{\mathbb{CMOD}}$
$\langle\{a_1, f\}, \{r_{cf}\}\rangle, \langle\{a_2, f\}, \{r_{cf}\}\rangle, \langle\{f\}, \{r_{cf}\}\rangle, \langle\{a_1, a_2\}, \{r_c\}\rangle, \langle\{a_1\}, \{r_c\}\rangle, \langle\{a_2\}, \{r_c\}\rangle, \langle\emptyset, \{r_c\}\rangle$

Figure 3 Selected DP tables after $\mathtt{DP}_{\mathbb{SINC}}$ (left) and after $\mathtt{M\text{-}DP}_{\mathbb{SINC}}$ (right) for TD $\mathcal{T}$.

$\chi(t_1) = \{a_1, \cdots, a_k, f, r_c, r_{cf}\}$, $\chi(t_2) = \{a_1, \cdots, a_k, r_2, \cdots, r_k, r_f\}$, and $\chi(t_3) = \emptyset$. Figure 3 (left) illustrates the tables for program P_2 after $\mathtt{DP}_{\mathbb{SINC}}$, whereas Figure 3 (right) presents tables after $\mathtt{M\text{-}DP}_{\mathbb{SINC}}$ was run, which, mainly due to cleanup, are exponentially smaller in k. Observe that in Pass 3.II, $\mathtt{M\text{-}DP}_{\mathbb{SINC}}$ "temporarily" materializes counter-witnesses for τ_1 only, presented in table $\tau_1^{\mathbb{CMOD}}$. Hence, using multi-pass algorithm $\mathtt{M\text{-}DP}_{\mathbb{SINC}}$ results in an exponential speedup. Note that we can extend the program such that we have the same effect for a TD of minimum width and even if we take the incidence graph. The program P_k and the TD $\mathcal{T}$ also reveal that a different TD of the same width, where f occurs very early in the bottom-up traversal, would result in a smaller table τ_1 even when running $\mathtt{DP}_{\mathbb{SINC}}$.

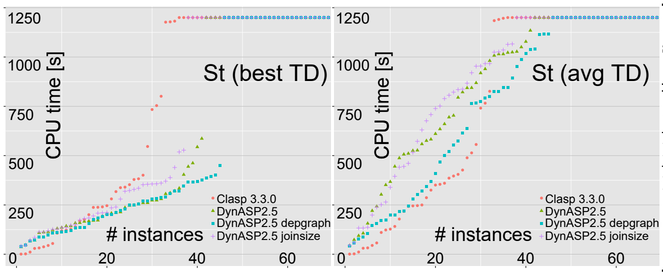
▶ **Theorem 8** ($\star$). *For a program P of semi-incidence treewidth $k = tw(S(P))$, the algorithm $\mathtt{M\text{-}DP}_{\mathbb{SINC}}$ is correct and runs in time $\mathcal{O}(2^{2^{k+2}} \cdot \|P\|)$.*

4.2 Implementation Details

Efficient implementations of dynamic programming algorithms on TDs are not a by-product of computational complexity theory and involve tuning and sophisticated algorithm engineering. For that reason, we present additional details about implementing the $\mathtt{M\text{-}DP}_{\mathbb{SINC}}$ algorithm into our prototypical multi-pass solver DynASP2.5.

Even though *normalizing* a TD (computing a nice TD) can be achieved without increasing its width, a normalization may artificially introduce additional atoms. Normalization causes several additional intermediate join nodes among such artificially introduced atoms requiring a significant amount of total unnecessary computation in practice. That is why, we use non-nice tree decompositions. In order to still ensure the theoretical runtime bounds, we limit the number of children per node to a constant. Moreover, linking counter-witnesses to witnesses efficiently is crucial. The main challenge is to deal with situations where a witness might be linked to a different family of counter-witnesses depending on different predecessors of the row (hidden in set notation of Line 9 in Listing 3). In these cases, DynASP2.5 eagerly creates a "clone" in form of a very light-weighted proxy to the original row and ensures that only the original row (if at all required) serves as a counter-witness during Pass 3. Together with efficient caches of counter-witnesses, DynASP2.5 reduces the overhead caused by clones in practice.

Dedicated data structures are vital. In DynASP2.5, sets of witnesses and satisfied rules are represented via constant-size bit vectors. We use 32-bit integers to represent whether



solver\time	# TO	Avg *solved* runtime TD	3.I	3.II	3.III	Σ
Clasp 3.3.0	34	-	-	-	-	334.5
DynASP2	68	-	-	-	-	-
DynASP2.5	28	0.5	3.5	16.6	264.4	285.0
...depgraph	24	10.9	3.4	15.3	251.3	280.9
...joinsize	32	0.7	3.5	16.5	278.3	299.0

Figure 4 Cactus plots showing best and average runtime among five TDs (left). Number of Timeouts (TO) and average runtime among solved instances (right).

an atom is set to true or a rule is satisfied in the respective bit positions according to the bag. A restriction to 32-bit integers seems reasonable as we assume, because of practical memory limitations, that our approach works well on TDs of width ≤ 20. Since state-of-the-art computers handle constant-sized integers extremely efficiently, DynASP2.5 allows for efficient projections and joins of rows, as well as subset checks. In order to not recompute counter-witnesses (in Pass 3.II) for different witnesses, we use a three-valued notation of counter-witness sets consisting of atoms set to true (T), false (F), or false but true in the witness set (TW) to build the reduct. Note that only atoms occurring in negations or choice rules are among the (TW)-atoms, since only these atoms "affect" the corresponding reducts.

Minimum width is not the only optimization goal when computing TDs by means of heuristics. Instead, using customized TDs that not only optimize the width, but also some other, relevant feature, works seemingly well in practice [2]. While DynASP2.5 ($\texttt{M-DP}_{\mathbb{SINC}}$) does not take additional TD features into account, we also implemented a variant (*DynASP2.5 depgraph*), which prefers one out of ten TDs that intuitively speaking avoids to introduce head atoms of some rule r in node t, without having encountered every body atom of r below t, similar to atom dependencies in the program [12]. The variant *DynASP2.5 joinsize* minimizes bag sizes of child nodes of join nodes, c.f. [1].

4.3 Experimental Evaluation

We performed experiments to investigate the runtime behavior of DynASP2.5 and its variants, in order to evaluate whether our multi-pass approach can be beneficial and has practical advantages over the classical single pass approach (DynASP2). Further, we considered the dedicated ASP solver clasp 3.3.0. Clearly, we *cannot* hope to solve programs with graph representations of high treewidth. However, programs involving real-world graphs such as graph problems on transit graphs admit TDs of acceptable width to perform DP on TDs. To get a first intuition, we focused on the Steiner tree problem (St) for our benchmarks.

We mainly inspected the CPU time using the average over five runs per instance (five fixed seeds allow for some variance in the heuristic TD computation). For each run, we limited the environment to 16 GB RAM and 1200 seconds CPU time. We used clasp with improvements for unsatisfiable cores [3] enabled and solution printing/recording disabled. We also benchmarked clasp with branch-and-bound, which, however, timed out on almost every instance. The left plot in Figure 4 shows the result of always selecting the best among five TDs, whereas the right plot shows the average running time. The table in Figure 4 reports average running times (TD computation and Passes 3.I, 3.II, 3.III) among the *solved* instances and the total number of timeouts (TO). We consider an instance to time out when all five TDs exceeded the limit. For the variants depgraph and joinsize, runtimes for computing and selecting among ten TDs are included. Our empirical benchmark results confirm that DynASP2.5 exhibits competitive runtime behavior even for TDs of treewidth around 14. Compared to clasp, DynASP2.5 is capable of additionally delivering the number of optimal solutions. In particular, the depgraph variant shows promising runtimes.

5 Conclusion

In this paper, we presented a novel approach for ASP solving based on ideas from parameterized complexity. Our algorithms run in linear time assuming bounded treewidth of the input program. Our solver applies DP in three passes, thereby avoiding redundancies. Experimental results indicate that our ASP solver is competitive for certain classes of instances with small treewidth, where the latest version of the well-known solver clasp hardly keeps up. An interesting question for future research is whether a linear amount of passes (incremental DP) can improve the runtime behavior.

References

1 Michael Abseher, Nysret Musliu, and Stefan Woltran. htd - A free, open-source framework for (customized) tree decompositions and beyond. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, pages 376–386. Springer, 2017. `doi:10.1007/978-3-319-59776-8_30`.

2 Michael Abseher, Nysret Musliu, and Stefan Woltran. Improving the efficiency of dynamic programming on tree decompositions via machine learning. *J. Artif. Intell. Res.*, 58:829–858, 2017. `doi:10.1613/jair.5312`.

3 Mario Alviano and Carmine Dodaro. Anytime answer set optimization via unsatisfiable core shrinking. *TPLP*, 16(5-6):533–551, 2016. `doi:10.1017/S147106841600020X`.

4 B. Bliem, M. Moldovan, M. Morak, and S. Woltran. The impact of treewidth on ASP grounding and solving. In *IJCAI'17*, The AAAI Press, pages 852–858, 2017.

5 Bernhard Bliem, Günther Charwat, Markus Hecher, and Stefan Woltran. D-flat^2: Subset minimization in dynamic programming on tree decompositions made easy. *Fundam. Inform.*, 147(1):27–61, 2016. `doi:10.3233/FI-2016-1397`.

6 Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *Comput. J.*, 51(3):255–269, 2008. `doi:10.1093/comjnl/bxm037`.

7 Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The pace 2017 parameterized algorithms and computational experiments challenge: The second iteration. In *IPEC'17*, LIPIcs, pages 30:1—-30:13, 2017. `doi:10.4230/LIPIcs.IPEC.2017.30`.

8 Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323, 1995. `doi:10.1007/BF01536399`.

9 J. K. Fichte, M. Hecher, and S. Woltran. DynASP2.5: Dynamic Programming on Tree Decompositions in Action. *CoRR*, arXiv:1706.09370, 2017.

10 Johannes Klaus Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Answer set solving with bounded treewidth revisited. In Marcello Balduccini and Tomi Janhunen, editors, *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, volume 10377 of *Lecture Notes in Computer Science*, pages 132–145. Springer, 2017. `doi:10.1007/978-3-319-61660-5_13`.

11 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012. `doi:10.1016/j.artint.2012.04.001`.

12 Georg Gottlob, Francesco Scarcello, and Martha Sideri. Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artif. Intell.*, 138(1-2):55–86, 2002. `doi:10.1016/S0004-3702(02)00182-0`.

13 T. Janhunen and I. Niemelä. The answer set programming paradigm. *AI Magazine*, 37(3):13–24, 2016. `doi:10.1609/aimag.v37i3.2671`.

14 Joachim Kneis, Alexander Langer, and Peter Rossmanith. Courcelle's theorem - A game-theoretic approach. *Discrete Optimization*, 8(4):568–594, 2011. `doi:10.1016/j.disopt.2011.06.001`.

15 Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002. `doi:10.1016/S0004-3702(02)00187-X`.