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

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

Abstract: Efficient exact parameterized algorithms are an active research area. Such algorithms exhibit a broad interest in the theoretical community. In the last few years, implementations for computing various parameters (parameter detection) have been established in parameterized challenges, such as treewidth, treedepth, hypertree width, feedback vertex set, or vertex cover. In theory, instances, for which the considered parameter is small, can be solved fast (problem evaluation), i.e., the runtime is bounded exponential in the parameter. While such favorable theoretical guarantees exists, it is often unclear whether one can successfully implement these algorithms under practical considerations. In other words, can we design and construct implementations of parameterized algorithms such that they perform similar or even better than well-established problem solvers on instances where the parameter is small. Indeed, we can build an implementation that performs well under the theoretical assumptions. However, it could also well be that an existing solver implicitly takes advantage of a structure, which is often claimed for solvers that build on SAT-solving. In this paper, we consider finding one solution to instances of answer set programming (ASP), which is a logic-based declarative modeling and solving framework. Solutions for ASP instances are so-called answer sets. Interestingly, the problem of deciding whether an instance has an answer set is already located on the second level of the polynomial hierarchy. An ASP solver that employs treewidth as parameter and runs dynamic programming on tree decompositions is DynASP2. Empirical experiments show that this solver is fast on instances of small treewidth and can outperform modern ASP when one counts answer sets. It remains open, whether one can improve the solver such that it also finds one answer set fast and shows competitive behavior to modern ASP solvers on instances of low treewidth. Unfortunately, theoretical models of modern ASP solvers already indicate that these solvers can solve instances of low treewidth fast, since they are based on SAT-solving algorithms. In this paper, we improve DynASP2 and construct the solver DynASP2.5, which uses a different approach. The new solver shows competitive behavior to state-of-the-art ASP solvers even for finding just one solution. We present empirical experiments where one can see that our new implementation solves ASP instances, which encode the Steiner tree problem on graphs with low treewidth, fast. Our implementation is based on a novel approach that we call multi-pass dynamic programming (M-DP). In the paper, we describe the underlying concepts of our implementation (DynASP2.5) and we argue why the techniques still yield correct algorithms.

Keywords: parameterized algorithms; fixed-parameter linear time; semi-incidence graph; tree decompositions; multi-pass dynamic programming
1. Introduction

Answer set programming (ASP) is a logic-based declarative modeling language and problem solving framework [1], where a program consists of sets of rules over propositional atoms and is interpreted under extended stable model semantics [2]. In ASP, problems are usually modeled in such a way that the stable models answer sets of a program directly form a solution to 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 [3] and we can solve ASP problems by encoding it into solving quantified Boolean formulas (QBF) [4,5]. A standard encoding usually involves an existential part (guessing a model of the program) and a universal part (minimality check).

Interestingly, when considering a more fine-grained complexity analysis one can take a structural parameter such as treewidth into account. On that account, one usually defines a graph representation of an input program such as the primal graph or the incidence graph and can show that solving ASP is double exponential (upper bound) in the treewidth of the graph representation and linear in the number of atoms of the input instance [6]. For a tighter conditional lower runtime bound, we can take a standard assumption in computational complexity, such as the Exponential Time Hypothesis (ETH), into account. Unsurprisingly, given its classical computational complexity, one can establish that the runtime of various computational ASP problems is double exponential in the treewidth (lower bound) in the worst case if we believe that ETH is true [7,8].

Despite the seemingly bad results in classical complexity, researchers implemented a variety of successful CDCL-based ASP solvers, among them Clasp [9] and WASP [10]. When viewing practical results from a parameterized perspective, one can find a solver (DynASP2) that build upon ideas from parameterized algorithmics and solves ASP problems by dynamic programming along tree decompositions [11]. The solver DynASP2 performs according to the expected theoretical worst case runtime guarantees and runs practically well if the input program has low treewidth. If we consider counting problems on instances of low treewidth, runtimes of DynASP2 are even faster than those of Clasp. However, conditional lower bounds seem to forbid significant improvement under theoretical aspects unless we design a multi-variate algorithm that takes additional structure into account. In the following of this paper, we outline how to construct a novel algorithm that additionally takes a “second parameter” into account.

The solver DynASP2 (i) takes a tree decomposition of a graph representation of the given input instance and (ii) solves the program via dynamic programming (DP) on the tree decomposition by traversing the tree exactly once. Both finding a model and checking minimality are considered at the same time. Once the root node has been reached, complete solutions (if exist) for the input program can be constructed. Due to the exhaustive nature of dynamic programming, all potential values are computed locally for each node of the tree decomposition. In consequence, space requirements can be quite extensive resulting in long running times. Moreover, in practice, dynamic programming algorithms on tree decompositions may yield extremely diverging run-times on tree decompositions of the exact same width [12].

In this paper, we propose a multi-traversal approach ($\Pi\text{-DP}_{\text{SINC}}$) for dynamic programming on tree decompositions as well as a new implementation (DynASP2.5). (The source code of our solver is available at https://github.com/daajoe/dynasp/releases/tag/v2.5.0.) In contrast to the approach described above, $\Pi\text{-DP}_{\text{SINC}}$ traverses the given tree decomposition multiple times. Starting from the leaves, we compute and store (i) sets of atoms that are relevant for the existential part (finding a model of the program) up to the root. Then we go back again to the leaves and compute and store (ii) sets of atoms that are relevant for the universal part (checking for minimality). Finally, we go once again back to the leaves and (iii) link sets from past Traversals (i) and (ii) that might lead to an answer set in the future. As a result, we allow for early cleanup of candidates that do not lead to answer sets. Theoretically, such an algorithm resembles a multi-variate view on instances, even
though we have a dynamic parameter, which we cannot simply compute before running the solvers.

Furthermore, we present technical improvements (including working on tree decompositions that do not have to be nice) and employ dedicated customization techniques for selecting tree decompositions. Our improvements are main ingredients to speedup the solving process for dynamic programming algorithms. Experiments indicate that DynASP2.5 is competitive even for finding one answer set using the Steiner tree problem on graphs with low treewidth. In particular, we are able to solve instances that have an upper bound on the incidence treewidth of 14 (whereas DynASP2 solved instances of treewidth at most 9).

1.1. Contributions

Our main contributions can be summarized as follows:

1. We present a new dynamic programming algorithm on a graph representation that is somewhat in-between the incidence and primal graph and show its correctness.
2. We establish a novel fixed-parameter linear algorithm (\(M\)-DP_{SINC}), which works in multiple traversals and computes existential and universal parts separately.
3. We present an implementation (DynASP2.5) and an experimental evaluation.

1.2. Related Work

Jakl, Pichler, and Woltran [6] have considered ASP solving when parameterized by the treewidth of a graph representation and suggested fixed-parameter linear algorithms. Fichte et al. [11] have established additional algorithms and presented empirical results on an implementation that is dedicated to counting answer sets for the full ground ASP language. The present paper extends their work by a multi-traversal dynamic programming algorithm. Algorithms for particular fragments of ASP [13] and for further language extensions of ASP [14] were proposed, including lower bounds [15]. Compared to general systems [16], our approach directly treats ASP. Bliem et al. [17] have introduced a general multi-traversal approach and an implementation (D-FLAT2) for dynamic programming on tree decompositions solving subset minimization tasks. Their approach allows to specify dynamic programming algorithms by means of ASP. In a way, one can see ASP in their approach as a meta-language to describe table algorithms (See Algorithm 1 for the concept of table algorithms.), whereas our work presents a dedicated algorithm to find an answer set of a program. In fact, our implementation extends their general ideas for subset minimization (disjunctive rules) to also support weight rules. However, due to space constraints we do not report on weight rules in this paper. Beyond that, we require specialized adaptions to the ASP problem semantics, including three valued evaluation of atoms, handling of tree decompositions that can be not nice, and optimizations in join nodes to be competitive. Abseher, Musliu, and Woltran [18] have presented a framework that computes tree decompositions via heuristics, which is also used in our solver. Other tree decomposition systems can be found on the PACE challenge track A website [19]. Note that improved heuristics for finding a tree decomposition of smaller width (if possible) directly yields faster results for our solver. Other parameters than treewidth have been considered in the literature, however, mostly in a theoretical context [20,21], some also consider just improving on solving the universal part [22], or take multiple parameters into account [23].

1.3. Journal Version

This paper is an extended and updated version of a paper that appeared in the proceedings of the 12th International Symposium on Parameterized and Exact Computation (IPEC’17). The present paper provides a higher level of detail, in particular regarding proofs and examples. We extend its previous versions in the following way. Additionally, we provide comprehensive details on the entire setting of our empirical work, present full listings of encodings, and include results for several runs with varying tree decompositions.
2. Preliminaries

2.1. Tree Decompositions

We assume that the reader is familiar with basic notions in graph theory [24,25], but we will fix some basic terminology below. An undirected graph or simply a graph is a pair \( G = (V, E) \) where \( V \neq \emptyset \) is a set of vertices and \( E \subseteq \{ \{u, v\} \subseteq V \mid u \neq v \} \) is a set of edges. We denote an edge \( \{v, w\} \) by \( vw \). 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 \( 1 \leq i \leq k \). A path is called simple if all its vertices are distinct. A graph \( G \) is called a tree if any two vertices in \( G \) can be connected by a unique simple path.

Let \( G = (V, E) \) be a graph; \( T = (N, F, n) \) a rooted tree with a set \( N \) of nodes, a set \( F \) of edges, and a root \( n \in N \); and \( \chi : N \to 2^V \) a function that maps each node \( t \in N \) to a set of vertices. We call the sets \( \{ \chi(t) \} \) bags. Then, the pair \( T = (T, \chi) \) is a tree decomposition (TD) of \( G \) if the following conditions hold:

1. For every vertex \( v \in V \) there is a node \( t \in N \) with \( v \in \chi(t) \);
2. For every edge \( e \in E \) there is a node \( t \in N \) with \( e \subseteq \chi(t) \); and
3. For any three nodes \( t_1, t_2, t_3 \in N \) whenever \( t_2 \) lies on the unique path from \( t_1 \) to \( t_3 \), then we have \( \chi(t_1) \cap \chi(t_3) \subseteq \chi(t_2) \).

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 \( G \). Note that each graph has a trivial TD \( (T, \chi) \) consisting of the tree \( (\{n\}, \emptyset, n) \) and the mapping \( \chi(n) = V \). It is well known that the treewidth of a tree is 1, and a graph containing a clique of size \( k \) has at least treewidth \( k - 1 \). For some arbitrary but fixed integer \( k \) and a graph of treewidth at most \( k \), we can compute a TD of width \( \leq k \) in time \( 2^{O(k)} \cdot |V| \) [26]. Given a TD \( (T, \chi) \) with \( T = (N, r, \cdot) \), for a node \( t \in N \) we say that \( \text{type}(t) = \text{leaf} \) if \( t \) has no children; \( \text{join} \) if \( t \) has children \( t' \) and \( t'' \) with \( t' \neq t'' \) and \( \chi(t) = \chi(t') = \chi(t'') \); \( \text{int} \) (“introduce”) if \( t \) has a single child \( t' \), \( \chi(t') \subseteq \chi(t) \) and \( |\chi(t')| = |\chi(t)| + 1 \); \( \text{forget} \) (“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, \( \text{type}(t) \in \{\text{leaf}, \text{join}, \text{int}, \text{forget}\} \), 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 [26]. Later, we traverse a TD bottom up, therefore, let post-order \((T, t)\) be the sequence of nodes in post-order of the induced sub-tree \( T' = (N', \cdot, t) \) of \( T \) rooted at \( t \).

2.2. Answer Set Programming (ASP)

Answer Set Programming, or ASP for short, is a declarative modeling and problem solving framework that combines techniques of knowledge representation and database theory. In this paper, we restrict ourselves to so-called ground ASP programs. For a comprehensive introduction, we refer the reader to standard texts [1,27–29].

2.2.1. Syntax

We consider a universe \( U \) of propositional atoms. A literal is an atom \( a \in U \) or its negation \( \neg a \). Let \( \ell, m, n \) be non-negative integers such that \( \ell \leq m \leq n \), \( a_1, \ldots, a_n \) distinct propositional atoms, and \( l \) a literal. A choice rule is an expression of the form \( \{a_1; \ldots; a_l\} \leftarrow a_{l+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \). A disjunctive rule is of the form \( a_1 \lor \cdots \lor a_l \leftarrow a_{l+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n \). An optimization rule is an expression of the form \( \leftarrow l \). A rule is either a disjunctive, a choice, or an optimization rule. For a choice or disjunctive rule \( r \), let \( \mathcal{H}_r := \{a_1, \ldots, a_l\} \) and \( \mathcal{B}_r^+ := \{a_{l+1}, \ldots, a_m\} \) and \( \mathcal{B}_r^- := \{a_{m+1}, \ldots, a_n\} \). Usually, if \( \mathcal{B}_r^- \cup \mathcal{B}_r^+ = \emptyset \) we write for a rule \( r \) simply \( \mathcal{H}_r \) instead of \( \mathcal{H}_r \leftarrow \emptyset \). For an optimization rule \( r \), if \( l = a_j \), let \( \mathcal{B}_r^+ := \{a_1\} \) and \( \mathcal{B}_r^- := \emptyset \); and if \( l = \neg a_j \), let \( \mathcal{B}_r^- := \{a_1\} \) and \( \mathcal{B}_r^+ := \emptyset \). For a rule \( r \), let \( \text{at}(r) := \mathcal{H}_r \cup \mathcal{B}_r^+ \cup \mathcal{B}_r^- \) denote its atoms and \( \mathcal{B}_r := \mathcal{B}_r^+ \cup \{-b \mid b \in \mathcal{B}_r^-\} \) its body.

Let a program \( P \) be a set of rules, and \( \text{at}(P) := \bigcup_{r \in P} \text{at}(r) \) denote its atoms. Furthermore, let \( \text{CH}(P) \), \( \text{DISJ}(P) \), and \( \text{OPT}(P) \) denote the set of all choice, disjunctive, and optimization rules in \( P \), respectively.
2.2.2. Semantics

A set $M \subseteq \text{at}(P)$ satisfies a rule $r$ if (i) $(H_r \cup B_r^+) \cap M \neq \emptyset$ or $B_r^+ \subseteq M$ for $r \in \text{DIS}(P)$ or (ii) $r \in \text{CH}(P) \cup \text{OPT}(P)$. Hence, choice and optimization rules are always satisfied. $M$ is a model of $P$, denoted by $M \models P$, if $M$ satisfies every rule in $P$. The reduct $r^M$ (i) of a choice rule $r$ is the set $\{a \leftarrow B_r^+ \mid a \in H_r \cap M, B_r^- \cap M = \emptyset\}$ of rules, and (ii) of a disjunctive rule $r$ is the singleton $\{H_r \leftarrow B_r^+ \mid B_r^- \cap M = \emptyset\}$. $P^M := \bigcup_{r \in P} r^M$ is called GL reduct of $P$ with respect to $M$. A set $M \subseteq \text{at}(P)$ is an answer set of $P$ if (i) $M \models P$ and (ii) there is no $M' \subseteq M$ such that $M' \models P^M$, that is, $M$ is subset minimal with respect to $P^M$. We call $\text{cst}(P, M) := \{r \mid r \in P, r$ is an optimization rule, $(B_r^+ \cap M) \cup (B_r^- \setminus M) \neq \emptyset\}$ the cost of answer set $M$ of $P$. An answer set $M$ of $P$ is optimal if its cost is minimal over all answer sets.

Intuitively, a choice rule of the form $a_1; \ldots; a_t \leftarrow a_{t+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$ allows us to conclude that if all atoms $a_{t+1}, \ldots, a_m$ are in an answer set $M$ and there is no evidence that any atom of $a_{m+1}, \ldots, a_n$ is in $M$ (so subset-minimality with respect to the reduct forces that $a_{m+1}, \ldots, a_n \notin M$), we can conclude that any subset of the atoms $a_1, \ldots, a_t$ can be in $M$. In other words, the atoms $a_1, \ldots, a_t$ are “excluded” from subset minimization. A disjunctive rule of the form $a_1 \lor \cdots \lor a_t \leftarrow a_{t+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$ allows us to conclude that if all atoms $a_{t+1}, \ldots, a_m$ are in $M$ and there is no evidence that any atom of $a_{m+1}, \ldots, a_n$ is in $M$, we can conclude that a subset of the atoms $a_1, \ldots, a_t$ can be in $M$, however, remains subject to subset minimization with respect to the reduct.

The meaning of optimization rules is simply that we minimize the cardinality of a set of atoms with respect to the literals in $M$ that occur in any minimization rule positively and not in $M$ for those that occur in any minimization rules negatively.

Example 1. Consider program

$$P = \{e_{ab}; e_{bc}; e_{cd}; e_{ad}; e_{db}; a \leftarrow e_{ab}; d \leftarrow e_{db}; c \leftarrow a; b \leftarrow c; \neg b; c \leftarrow a; b \leftarrow c; \neg b; g \leftarrow \neg a; i \leftarrow a; d \leftarrow g\}.$$

The set $A = \{e_{ab}, e_{bc}, a, c\}$ is an answer set of $P$, since $\{e_{ab}, e_{bc}, a, c\}$ is the only minimal model of $P^A = \{e_{ab} \leftarrow; e_{bc} \leftarrow; a \leftarrow e_{ab}; d \leftarrow e_{db}; c \leftarrow a; b \leftarrow c; \neg b; c \leftarrow a; b \leftarrow c; \neg b; g \leftarrow \neg a; i \leftarrow a; d \leftarrow g\}$. Then, consider program $R = \{a \lor c \leftarrow b; b \leftarrow c; \neg g; c \leftarrow a; b \lor c \leftarrow e; h \lor i \leftarrow g; \neg c; a \lor b; g \leftarrow \neg i; c; \{d\} \leftarrow g\}$. The set $B = \{b, c, d, g\}$ is an answer set of $R$ since $\{b, c, d, g\}$ and $\{a, c, d, g\}$ are the minimal models of $R^B = \{a \lor c \leftarrow b; c \leftarrow a; b \lor c \leftarrow e; a \lor b; g; c; d \leftarrow g\}$.

In this paper, we mainly consider the following output answer set problem, that is, output an answer set for an ASP program. The decision version of this problem is $\Sigma_2^P$-complete. When sketching correctness, we also consider the following two problems: listing all optimal answer sets of $P$, ENUMASP for short; and we the entailment problem of listing every subset-minimal model $M$ of $P$ with $\text{sol} \in M$, or ENUMMINSAT, for a given propositional formula $F$ and an atom sol.

2.3. 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 $ar$ if $a \in \text{at}(r)$ for some rule $r \in P$ [11]. 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 $ar$ if $a \in \text{at}(r)$ for some rule $r \in P$ as well as (ii) an edge $ab$ for disjoint 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 $tw(I(P)) \leq tw(S(P))$. Furthermore, by definition of a TD and the construction of a semi-incidence graph that head atoms of choice rules, respectively, occur in at least one common bag of the TD.
2.4. Sub-Programs

Let $T = (T, \chi)$ be a nice TD of graph representation $S(P)$ of a program $P$. Furthermore, let $T = (N, n)$ and $t \in N$. The bag-program is defined as $P_t := P \cap \chi(t)$. Further, 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 $P_{\leq t} := P_t \setminus P_t$. It holds that $P_{\leq t} = P_{< t} = P$ and $\text{at}_{\leq t} = \text{at}_{t} = \text{at}(P)$.

Example 2. Figure 1 (upper) illustrates the semi-incidence graph of program $P$ from Example 1. Figure 1 (lower) shows a tree decomposition of this graph. Intuitively, the tree decomposition enables us to evaluate $P$ by analyzing sub-programs and combining results agreeing on $a_c$. Indeed, for the given TD of Figure 1 (left), $P_{\leq t_6} = \{r_{ab}, r_b\}$ and $\text{at}_{\leq t_6} = \{e_{ab}, a_b\}$, $P_{\leq t_{23}} = \{r_{ad}, r_d\}$ and $\text{at}_{\leq t_{23}} = \{e_{ad}, a_d\}$, as well as $P_{< t_{29}} = \{r_{ad}, r_{cd}, r_d\}$.

![Figure 1](image_url)
3. For every node \( t \in T \) in the tree decomposition \( T = ((T, E, n), \chi) \) (in a bottom-up traversal), run an algorithm \( A \), which we call table algorithm, and compute a "table" \( A\)-Tabs\([t]\), which is a set of tuples (or rows for short). Intuitively, algorithm \( A \) transforms tables of child nodes of \( t \) to the current node, and solves a "local problem" using bag-program \( P_t \). The algorithm thereby computes (i) sets of atoms called (local) witness sets and (ii) for each local witness set \( M \) subsets of \( M \) called counter-witness sets [11], and directly follows the definition of answer sets being (i) models of \( P \) and (ii) subset minimal with respect to \( P_M \).

4. For root \( n \) interpret the table \( A\)-Tabs\([n]\) (and tables of children, if necessary) and print the solution to the considered ASP problem.

**Algorithm 1:** Algorithm DP\(_A\)(\(T\)) for Dynamic Programming on TD \( T \) for ASP [11].

<table>
<thead>
<tr>
<th>In:</th>
<th>Table algorithm ( A ), nice tree TD ( T = (\mathcal{T}, \chi) ) with ( \mathcal{T} = (N, E, n) ) of ( G(P) ) according to ( A ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out:</td>
<td>( A)-Tabs: maps each TD node ( t \in T ) to some computed table ( \tau_t ).</td>
</tr>
<tr>
<td>1</td>
<td>for iterate ( t ) in post-order((T, n)) do</td>
</tr>
<tr>
<td>2</td>
<td>( \text{Child-Tabs}_t := { A)-Tabs([t']</td>
</tr>
<tr>
<td>3</td>
<td>( A)-Tabs([t] \leftarrow A(t, \chi(t), P_t, \text{at} \leq, \text{Child-Tabs}_t) )</td>
</tr>
</tbody>
</table>

An Algorithm on the Semi-Incidence Graph

Next, we propose a new table algorithm (SINC) for programs without optimization rules. Since our algorithm trivially extends to counting and optimization rules by earlier work [11], we omit such rules. The table algorithm SINC employs the semi-incidence graph and is depicted in Algorithm 2. DP\(_{\text{SINC}}\) merges two earlier algorithms for the primal and incidence graph [11] resulting in slightly different worst case runtime bounds (c.f., Proposition 1).

**Algorithm 2:** Table algorithm SINC\((t, \chi_t, P_t, \text{at} \leq, \text{Child-Tabs}_t)\).

<table>
<thead>
<tr>
<th>In:</th>
<th>Bag ( \chi_t ), bag-program ( P_t ), atoms-below ( \text{at} \leq ), child tables ( \text{Child-Tabs}_t ) of ( t ). Out:</th>
<th>Tab. t.</th>
</tr>
</thead>
<tbody>
<tr>
<td>/* We use the following abbreviations */</td>
<td>For set ( S ) and element ( s ), we let ( S^+_s := S \cup { s } ) and ( S^-_s := S \setminus { s } ).</td>
<td>*/</td>
</tr>
<tr>
<td>1</td>
<td>if type((t) = \text{leaf} ) then ( t := { (\emptyset, \emptyset, \emptyset) } )</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>else if type((t) = \text{int} ), ( a \in \chi_t ) &amp; ( P_t ) is introduced and ( t' \in \text{Child-Tabs}_t ) then</td>
<td></td>
</tr>
</tbody>
</table>
| 3 | \( \tau_t := \{ (M^+_t, \sigma \cup \text{SatPr}(P_t^{(+M)}, M^+_t)), \} \)
| 4 | \( \{ (C^+_t, \rho \cup \text{SatPr}(P_t^{(+M)}, C^+_t)) | (C, \rho) \in C \} \cup \)
| 5 | \( \{ (C, \rho \cup \text{SatPr}(P_t^{(+M)}, C)) | (C, \rho) \in C \} \cup \{ (M, \sigma \cup \text{SatPr}(P_t^{(\sigma M)}, M)) \} \)
| 6 | \( \cup \{ (M, \sigma \cup \text{SatPr}(P_t^{(\sigma)}, M), \} \)
| 7 | \( \{ (C, \rho \cup \text{SatPr}(P_t^{(\rho)}, M), | (C, \rho) \in C \} \} \)
| 8 | \( \cup \{ (\sigma, C, \rho) \in \text{sat} | (\sigma, C) \in \text{at} \} \)
| 9 | else if type\((t) = \text{int} \), \( r \in \chi_t \) \& \( P_t \) is introduced and \( t' \in \text{Child-Tabs}_t \) then |
| 10 | \( \tau_t := \{ (M, \sigma \cup \text{SatPr}(\{r\}^{(+M)}, M), \} \)
| 11 | \( \{ (C, \rho \cup \text{SatPr}(\{r\}^{(\rho)}, C)) | (C, \rho) \in C \} \} \)
| 12 | else if type\((t) = \text{forget} \), \( a \notin \chi_t \) \& \( t' \in \text{Child-Tabs}_t \) then |
| 13 | \( \tau_t := \{ (\sigma, C, \rho) \in C | (\sigma, C) \in \text{sat} \} \)
| 14 | else if type\((t) = \text{forget} \), \( r \notin \chi_t \) \& \( t' \in \text{Child-Tabs}_t \) then |
| 15 | \( \tau_t := \{ (M, \sigma, C, \rho) \in \text{sat} | (\sigma, C, \rho) \in \text{at} \} \)
| 16 | else if type\((t) = \text{join} \) \& \( t', t'' \in \text{Child-Tabs}_t \) with \( t' \neq t'' \) then |
| 17 | \( \tau_t := \{ (M, \sigma', \rho), \} \)
| 18 | \( \{ (C, \rho') \in C', (C, \rho'' ) \in C'' \} \cup \)
| 19 | \( \{ (M, \rho \cup \rho'') | (M, \rho) \in C' \} \cup \)
| 20 | \( \{ (M, \rho \cup \rho) | (M, \rho) \in C'' \} \}

The solution to the considered ASP problem.
Our table algorithm SINC computes and stores (i) sets of atoms (witnesses) that are relevant for the SAT part (finding a model of the program) and (ii) sets of atoms (counter-witnesses) that are relevant for the UNSAT part (checking for minimality). In addition, we need to store for each set of witnesses as well as its set of counter-witnesses satisfiability states (sat-states for short). For the following reason: By Definition of TDs and the semi-incidence graph, it is true for every atom \( a \) and every rule \( r \) of a program that if atom \( a \) occurs in rule \( r \), then \( a \) and \( r \) occur together in at least one bag of the TD. In 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 whether a witness satisfies a rule, because not all atoms that occur in a rule occur together in exactly one bag. Hence, when our algorithm traverses the TD and an atom is forgotten we still need to store this sat-state, as setting the forgotten 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 choice rule head, those atoms \( A \) are not stored in a common bag of any 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 [11]. We can see witness sets together with its sat-state as witness. Then, in Algorithm 2 (SINC) a row in the table \( \tau \) is a triple \( (M, \sigma, C) \). The set \( M \subseteq \text{at}(P) \cap \chi(t) \) represents a witness set. The family \( C \) of sets concerns counter-witnesses, which we will discuss in more detail below. The sat-state \( \sigma \) for \( M \) represents rules of \( \chi(t) \) satisfied by a superset of \( M \). Hence, \( M \) witnesses a model \( M' \supseteq M \) where \( M' \models P_{cl} \cup \sigma \). We use binary operator \( \cup \) to combine sat-states, which ensures that rules satisfied in at least one operand remain satisfied. We compute a new sat-state \( \sigma \) from a sat-state and satisfied rules, formally, \( \text{SatPr}(R, M) := \{ r \mid (r, R) \in R, M \models R \} \) for \( M \subseteq \chi(t) \setminus P_{cl} \) and program \( R(r) \) constructed by \( R \), mapping rules to local-programs as given in the following definition.

**Definition 1.** Let \( P \) be a program, \( T = (\cdot, \chi) \) be a TD of \( S(P) \), \( t \) be a node of \( T \) and \( R \subseteq P_t \). The local-program \( R^{(i)} \) is obtained from \( R \cup \{ \leftarrow B_r \mid r \in R \text{ is a choice rule}, H_r \subseteq \text{at}_{\leq 1} \} \) (We require to add \( \{ \leftarrow B_r \mid r \in R \text{ is a choice rule}, H_r \subseteq \text{at}_{\leq 1} \} \) in order to decide satisfiability for corner cases of choice rules involving counter-witnesses of Line 3 in Algorithm 2.) by removing from every rule all literals \( a, \neg a \) with \( a \notin \chi(t) \). We define \( R^{(i)} : R \rightarrow 2^{R^{(i)}} \) by \( R^{(i)}(r) := \{ r \}^{(i)} \) for \( r \in R \).

After we explained how to obtain models, we describe how to compute counter-witnesses. Family \( C \) consists of rows \( (C, \rho) \) where \( C \subseteq \text{at}(P) \cap \chi(t) \) is a counter-witness set in \( t \) to \( M \). Similar to the sat-state \( \sigma \), the sat-state \( \rho \) for \( C \) under \( M \) represents whether rules of the GL reduct \( P_t^M \) are satisfied by a superset of \( C \). We can see counter-witness sets together with its sat-state as counter-witnesses. Thus, \( C \) witnesses the existence of \( C' \subseteq M' \) satisfying \( C' \models (P_{cl} \cup \rho)^M \) since \( M \) witnesses a model \( M' \subseteq M \) where \( M' \models P_{cl} \). In consequence, there exists an answer set of \( P \) if the root table contains \( (\emptyset, \emptyset, \emptyset) \). We require local-reducts for deciding satisfiability of counter-witness sets.

**Definition 2.** Let \( P \) be a program, \( T = (\cdot, \chi) \) be a TD of \( S(P) \), \( t \) be a node of \( T \), \( R \subseteq P_t \) and \( M \subseteq \text{at}(P) \). We define local-reduct \( R^{(i,M)} \) by \( [R^{(i)}]^M \) and \( \bar{R}^{(i,M)} : R \rightarrow 2^{R^{(i,M)}} \) by \( \bar{R}^{(i,M)}(r) := \{ r \}^{(i,M)} \), \( r \in R \).

Definitions 1 and 2 together with Algorithm 2 provides the formal description of our algorithm to solve ASP instances by means of dynamic programming on tree decompositions of the semi-incidence graph of the input instance. In the following section, we argue for correctness of our algorithm.

### 3.1. Correctness and Runtime of Algorithm 2 (SINC)

Next, we provide insights into the correctness and runtime of Algorithm 2 (SINC).
Proposition 1. Let $P$ be a program and $k := \text{tw}(S(P))$. Then, the algorithm $\mathcal{DP}_{\text{SINC}}$ is correct and runs in time $O(2^{k+2} \cdot \|S(P)\|)$.

Proof (Idea/Main Arguments). Previous works [6,11] established the correctness and runtime guarantees on the primal and incidence graph, respectively. Since the semi-incidence graph introduces a clique only for choice rule and the algorithm on the semi-incidence graph follows the algorithm on the incidence graph for non-choice rules and the one for the primal graph for choice rules, we immediately obtain correctness and runtime results as the cases never overlap. □

The remainder of this sections provides details on proving the statement above. The correctness proof investigates each node type separately. We have to show that a tuple at a node $t$ guarantees existence of an answer set for the program $P_{\leq t}$, proving soundness. Conversely, we have to show that each answer set is indeed evaluated while traversing the tree decomposition, which provides completeness. We employ this idea using the notions of (i) partial solutions consisting of partial models and the notion of (ii) local partial solutions.

Definition 3. Let $P$ be a program, $\mathcal{T} = (T, \chi)$ be a tree decomposition of the semi-incidence graph $S(P)$ of $P$, where $T = (N, \cdot, \cdot)$, and $t \in N$ be a node. Further, let $M, C \subseteq \text{at}_{\leq t}$ be sets and $\sigma \subseteq P_{\leq t}$ be a set of rules. The tuple $(C, \sigma)$ is a partial model for $t$ under $M$ if the following conditions hold:

1. $C \models (P_{\leq t})^M$,
2. for $r \in P_{\leq t}$ we require:
   (a) for disjunctive rule $r \in P_{\leq t}$, we have $B_r^- \cap M \neq \emptyset$ or $B_r^+ \cap \text{at}_{\leq t} \not\subseteq C$ or $H_r \cap C \neq \emptyset$ if and only if $r \in \sigma$;
   (b) for choice rule $r \in P_t$, we have $B_r^- \cap M \neq \emptyset$ or $B_r^+ \cap \text{at}_t \not\subseteq C$ or both $H_r \subseteq \text{at}_t$ and $H_r \cap (M \setminus C) = \emptyset$ if and only if $r \in \sigma$.

Definition 4. Let $P$ be a program, $\mathcal{T} = (T, \chi)$ where $T = (N, \cdot, n)$ be a tree decomposition of $I(P)$, and $t \in N$ be a node. A partial solution for $t$ is a tuple $(M, \sigma, C)$ where $(M, \sigma)$ is a partial model under $M$ and $C$ is a set of partial models $(C, \rho)$ under $M$ with $C \subseteq M$.

The following lemma establishes correspondence between answer sets and partial solutions.

Observation 1. Let $P$ be a program, $\mathcal{T} = (T, \chi)$ be a tree decomposition of the semi-incidence graph $S(P)$ of program $P$, where $T = (\cdot, \cdot, n)$, and $\chi(n) = \emptyset$. Then, there exists an answer set $M$ for $P$ if and only if there exists a partial solution $u = (M, \sigma, \emptyset)$ with $\sigma P$ for root $n$.

Proof. Given an answer set $M$ of $P$ we construct $u = (M, \sigma, \emptyset)$ with $\sigma := P$ such that $u$ is a partial solution for $n$ according to Definition 4. For the other direction, Definition 3 and Definition 4 guarantee that $M$ is an answer set if there exists some tuple $u$. In consequence, the observation holds. □

Next, we define local partial solutions to establish a notion that corresponds to the tuples obtained in Algorithm 2 but on in relation to solutions that we are interested in.

Definition 5. Let $P$ be a program, $\mathcal{T} = (T, \chi)$ a tree decomposition of the semi-incidence graph $S(P)$, where $T = (N, \cdot, n)$, and $t \in N$ be a node. A tuple $u = (M, \sigma, C)$ is a local partial solution for $t$ if there exists a partial solution $\hat{u} = (\hat{M}, \hat{\sigma}, \hat{C})$ for $t$ such that the following conditions hold:

1. $M = \hat{M} \cap \chi(t)$,
2. $\sigma = \hat{\sigma}$, and
3. $C = \{ (\hat{C} \cap \chi(t), \hat{\rho}^{M, \hat{C}}) \mid (\hat{C}, \hat{\rho}) \in \hat{C} \}$.

We denote by $\bar{u}^t$ the local partial solution $u$ for $t$ given partial solution $\hat{u}$.
The following observation provides justification that it suffices to store local partial solutions instead of partial solutions for a node \( t \in N \).

**Observation 2.** Let \( P \) be a program, \( T = (T, \chi) \) a tree decomposition of \( S(P) \), where \( T = (N, \cdot, n) \), and \( \chi(n) = \emptyset \). Then, there exists an answer set for \( P \) if and only if there exists a local partial solution of the form \( \langle \emptyset, \emptyset, \emptyset \rangle \) for the root \( n \in N \).

**Proof.** Since \( \chi(n) = \emptyset \), every partial solution for the root \( n \) is an extension of the local partial solution \( u \) for the root \( n \in N \) according to Definition 5. By Observation 1, we obtain that the observation is true. \( \square \)

In the following, we abbreviate atoms occurring in bag \( \chi(t) \) by \( a_t \), i.e., \( a_t := \chi(t) \setminus P_t \).

**Lemma 1** (Soundness). Let \( P \) be a program, \( T = (T, \chi) \) a tree decomposition of semi-incidence graph \( S(P) \), where \( T = (N, \cdot, \cdot) \), and \( t \in N \) a node. Given a local partial solution \( u' \) of child table \( \tau' \) (or local partial solution \( u'' \) of table \( \tau'' \) and local partial solution \( u'' \) of table \( \tau'' \)), each tuple \( u \) of table \( \tau_t \) constructed using table algorithm \( \text{SINC} \) is also a local partial solution.

**Proof.** Let \( u' \) be a local partial solution for \( t' \in N \) and \( u \) a tuple for node \( t \in N \) such that \( u \) was derived from \( u' \) using table algorithm \( \text{SINC} \). Hence, node \( t' \) is the only child of \( t \) and \( t \) is either removal or introduce node.

Assume that \( t \) is a removal node and \( r \in P_t \setminus P_t \) for some rule \( r \). Observe that \( u = \langle M, \sigma, C \rangle \) and \( u' = \langle M, \sigma', C' \rangle \) are the same in witness \( M \). According to Algorithm 2 and since \( u \) is derived from \( u' \), we have \( r \in \sigma' \). Similarly, for any \( \langle C', \rho' \rangle \in C', r \in \rho' \). Since \( u' \) is a local partial solution, there exists a partial solution \( \hat{u}' \) of \( t' \), satisfying the conditions of Definition 5. Then, \( \hat{u}' \) is also a partial solution for node \( t \), since it satisfies all conditions of Definitions 3 and 4. Finally, note that \( u = (\hat{u}')_t \) since the projection of \( \hat{u}' \) to the bag \( \chi(t) \) is \( u \) itself. In consequence, the tuple \( u \) is a local partial solution.

For a \( a \in at \setminus at_t \) as well as for introduce nodes, we can analogously check the lemma.

Next, assume that \( t \) is a join node. Therefore, let \( u' \) and \( u'' \) be local partial solutions for \( t' \), \( t'' \in N_r \), respectively, and \( u \) a tuple for node \( t \in N \) such that \( u \) can be derived using both \( u' \) and \( u'' \) in accordance with the \( \text{SINC} \) algorithm. Since \( u' \) and \( u'' \) are local partial solutions, there exists partial solution \( \hat{u}' \) of node \( t' \) and partial solution \( \hat{u}'' \) of node \( t'' \). Using these two partial solutions, we can construct \( \hat{u} = (\hat{u}'' \cup \hat{u}''', \hat{u}' = \hat{u}'' \cup \hat{u}'', \hat{u}'' = \hat{u}'' \cup \hat{u}'') \) where \( = (\cdot, \cdot) \) is defined in accordance with Algorithm 2 as follows:

\[
\hat{C}' \cup \hat{C}'' := \left\{ (\hat{C}', \hat{C}'') \mid (\hat{C}', \hat{C}') \in \hat{C}', \hat{C}' \cap at_t = \hat{C}' \cap at_t \right\} \cup
\left\{ (\hat{C}, \hat{C}') \mid (\hat{C}, \hat{C}') \in \hat{C}, \hat{C} \cap at_t = \hat{C} \cap at_t \right\} \cup
\left\{ (\hat{C}, \hat{C}'') \mid (\hat{C}, \hat{C}'') \in \hat{C}, \hat{C} \cap at_t = \hat{C} \cap at_t \right\}.
\]

Then, we check all conditions of Definitions 3 and 4 in order to verify that \( \hat{u} \) is a partial solution for \( t \). Moreover, the projection \( \hat{u}' \) of \( \hat{u} \) to the bag \( \chi(t) \) is exactly \( u \) by construction and hence, \( u = \hat{u}' \) is a local partial solution.

Since we have provided arguments for each node type, we established soundness in terms of the statement of the lemma. \( \square \)

**Lemma 2** (Completeness). Let \( P \) be a program, \( T = (T, \chi) \) where \( T = (N, \cdot, \cdot) \) be a tree decomposition of the semi-incidence graph \( S(P) \) and \( t \in N \) be a node. Given a local partial solution \( u \) of table \( t \), either \( t \) is a leaf node, or there exists a local partial solution \( u' \) of child table \( \tau' \) (or local partial solution \( u'' \) of table \( \tau'' \) and local partial solution \( u'' \) of table \( \tau'' \)) such that \( u \) can be constructed by \( u' \) (or \( u' \) and \( u'' \), respectively) and using table algorithm \( \text{SINC} \).

**Proof.** Let \( t \in N \) be a removal node and \( r \in P_t \setminus P_t \) with child node \( t' \in N \). We show that there exists a tuple \( u' \) in table \( \tau_t \) for node \( t' \) such that \( u \) can be constructed using \( u' \)
by SINC (Algorithm 2). Since \( u \) is a local partial solution, there exists a partial solution \( \hat{u} = (M, \sigma, \hat{C}) \) for node \( t \), satisfying the conditions of Definition 5. Since \( r \) is the removed rule, we have \( r \in \sigma \). By similar arguments, we have \( r \in \rho \) for any tuple \((\hat{C}, \rho) \in \hat{C} \). Hence, \( \hat{u} \) is also a partial solution for \( t' \) and we define \( u' := \hat{u}' \), which is the projection of \( \hat{u} \) onto the bag of \( t' \). Apparently, the tuple \( u' \) is a local partial solution for node \( t' \) according to Definition 5. Then, \( u \) can be derived using SINC algorithm and \( u' \). By similar arguments, we establish the lemma for \( a \in atv \setminus atv \) and the remaining (three) node types. Hence, the lemma sustains. \( \square \)

Now, we are in situation to prove the correctness statement in Proposition 1.

**Proof of Proposition 1 (Correctness).** We first show soundness. Let \( T = (T, \chi) \) be the given tree decomposition, where \( T = (N, \cdot, n) \). By Observation 2 we know that there is an answer set for \( P \) if and only if there exists a local partial solution for the root \( n \). Note that the tuple is of the form \( \langle \emptyset, \emptyset, \emptyset \rangle \) by construction. Hence, we proceed by induction starting from the leaf nodes. In fact, the tuple \( \langle \emptyset, \emptyset, \emptyset \rangle \) is trivially a partial solution by Definitions 3 and 4 and also a local partial solution of \( \langle \emptyset, \emptyset, \emptyset \rangle \) by Definition 5. We already established the induction step in Lemma 1. Hence, when we reach the root \( n \), when traversing the tree decomposition in post-order by algorithm \( DP_{SINC} \), we obtain only valid tuples in between and a tuple of the form \( \langle \emptyset, \emptyset, \emptyset \rangle \) in the table of the root \( n \) witnesses an answer set. Next, we establish completeness by induction starting from the root \( n \). Let therefore, \( M \) be an arbitrary answer set of \( P \). By Observation 1, we know that for the root \( n \) there exists a local partial solution of the form \( \langle \emptyset, \emptyset, \emptyset \rangle \) for partial solution \( \langle M, \sigma, \emptyset \rangle \) with \( r \in \sigma \) for \( r \in P \). We already established the induction step in Lemma 2. Hence, we obtain some (corresponding) tuples for every node \( t \). Finally, stopping at the leaves \( n \). In consequence, we have shown both soundness and completeness resulting in the fact that the correctness statement made in Proposition 1 is true. \( \square \)

Next, we turn our attention to the worst-case runtime bounds claimed in Proposition 1. First, we give a lemma on worst-case space requirements in tables for the nodes of our algorithm.

**Lemma 3.** Given a program \( P \), a tree decomposition \( T = (T, \chi) \) with \( T = (N, \cdot, n) \) of the semi-incidence graph \( S(P) \), and a node \( t \in N \). Then, there are at most \( 2^{k+1} \cdot 2^{k+1} \cdot 2^{2k+1} \cdot 2^{k+1} \) tuples in \( \tau_t \) using algorithm \( DP_{SINC} \) for width \( k \) of \( T \).

**Proof (Sketch).** Let \( P \) be the given program, \( T = (T, \chi) \) a tree decomposition of the semi-incidence graph \( S(P) \), where \( T = (N, \cdot, \cdot) \), and \( t \in N \) a node of the tree decomposition. Then, by definition of a decomposition of the primal graph for each node \( t \in N \), we have \( |\chi(t)| \leq k \). In consequence, we can have at most \( 2^{k+1} \) many witnesses, and for each witness a subset of the set of witnesses consisting of at most \( 2^{k+1} \) many counter-witnesses. In total, we need to distinguish \( 2^{k+1} \) two states for the sat-states \( \sigma \) and for each witness of a tuple in the table \( \tau_t \) for node \( t \). Since for each witness in the table \( \tau_t \) for node \( t \in N \) we remember rule-states for at most \( k + 1 \) rules, we store up to \( 2^{k+1} \) many combinations per witness. In total we end up with at most \( 2^{2k+1} \cdot 2^{k+1} \) many counter-witnesses for each witness and rule-state in the worst case. Thus, there are at most \( 2^{k+1} \cdot 2^{k+1} \cdot 2^{2k+1} \) tuples in table \( \tau_t \) for node \( t \). In consequence, we established the lemma. \( \square \)

**Proof of Proposition 1 (Runtime).** Let \( P \) be a program, \( S(P) = (V, \cdot) \) its semi-incidence graph, and \( k \) be the treewidth of \( P(P) \). Then, we can compute in time \( 2^{O(k^2)} \cdot |V| \) a tree decomposition of width at most \( k \) [30]. We take such a tree decomposition and compute in linear time a nice tree decomposition with \( T = (N, \cdot, \cdot) \). Since the number of nodes in \( N \) is linear in the graph size and since for every node \( t \in N \) the table \( \tau_t \) is bounded by \( 2^{k+1} \cdot 2^{k+1} \cdot 2^{2k+1} \cdot 2^{k+1} \) according to Lemma 3,
we obtain a running time of $O(2^{2^l+2} \cdot ||S(P)||)$. Consequently, the claimed runtime bounds in the proposition hold. □

3.2. An Extended Example

In Example 3 we give an idea how we compute models of a given program using the semi-incidence graph. The resulting algorithm $\text{MOD}$ is obtained from $\text{SINC}$, by taking only the first two row positions (red and green parts). The remaining position (blue part), can be seen as an algorithm $(\text{C} \text{MOD})$ that computes counter-witnesses. We assume that the $i$th-row in each table $\tau_i$ corresponds to $u_{i,t} = (M_{t,i}, e_{i,t})$.

Example 3. Consider program $P$ from Example 1, which was given by

$$P = \{ \{e_{ab}\}; \{e_{bc}\}; \{e_{cd}\}; \{e_{ad}\}; a_b \leftarrow e_{ab}; a_d \leftarrow e_{cd}; a_c \leftarrow a_b, e_{bc}; a_c \leftarrow a_d, e_{cd}; \}$$. 

TD $T = (\cdot, \chi)$ in Figure 2, and the tables $\tau_{12}, \ldots, \tau_{34}$, which illustrate computation results obtained during post-order traversal of $T$ by $\text{DP}_{\text{MOD}}$. Note that Figure 2 does not show every intermediate node of TD $T$. Table $\tau_1 = \{ (\emptyset, \emptyset) \}$ as type($t_1$) = leaf (see Algorithm 2 L1). Table $\tau_3$ is obtained via introducing rule $r_{ab}$ after introducing atom $e_{ab}$ (type($t_2$) = type($t_1$) = int). It contains two rows due to two possible truth assignments using atom $e_{ab}$ (L3–5). 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 $a_t \in \tau_3$ (see L7 and Definition 1).

Intuitively, whenever a rule $r$ is proven to be satisfiable, sat-state $t_{1r}$ marks $r$ satisfiable since an atom of a rule of $S(P)$ might only occur in one TD bag. Consider table $\tau_4$ with type($t_4$) = forget and $r_{ab} \in \chi(t_4) \setminus \chi(t_3)$. By definition (TDs and semi-incidence graph), we have encountered every occurrence of any atom in $r_{ab}$. Consequently, $\text{MOD}$ enforces that only rows where $r_{ab}$ is marked satisfiable in $t_3$, are considered for table $\tau_4$. The resulting table $\tau_4$ consists of rows of $\tau_3$ with $\sigma_{3,1} = \emptyset$, where rule $r_{ab}$ is proven satisfied ($r_{ab} \notin \sigma_{2,2}$, see L11). 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 $u_{6,2} = (M_{6,2}, e_{6,2}) \in \emptyset$ does not have a “successor row” in $\tau_{10}$, since $r_b \notin \sigma_{6,2}$. Intuitively, join node $i_{34}$ joins only common witness sets in $\tau_{17}$ and $\tau_{33}$ 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 L13–14).

![Figure 2. The tree decomposition $T$ of the semi-incidence graph $S(P)$ for program $P$ from Example 1 and Figure 1. Selected DP tables after $\text{DP}_{\text{MOD}}$. We explain details of the figure in Example 3.](image-url)
We assume again row numbers per table $\tau_i$, i.e., $u_{i,j} = \langle M_{i,j}, \sigma_{i,j}, C_{i,j} \rangle$ is the $i^{th}$-row. Further, for each counter-witness $\langle C_{i,j}, p_{i,j} \rangle \in C_{i,j}$, $j$ marks its “order” (as depicted in Figure 3 (right)) in set $C_{i,j}$.

**Example 4.** Again, we consider program $P$ from Example 1, and $T = (\cdot, \chi)$ of Figure 3 as well as tables $\tau_1, \ldots, \tau_{34}$ of Figure 3 (right) using $\text{DP}_{\text{SINC}}$. We only discuss certain tables. Table $\tau_1 = \{\langle \emptyset, \emptyset, \emptyset \rangle \}$ as type $t_1 = \text{leaf}$. Node $t_2$ introduces atom $e_{ab}$, resulting in table $\{\langle e_{ab}, \emptyset, \{\emptyset, \emptyset, \emptyset \} \rangle \}$ (compare to Algorithm 2 L3-5). Then, node $t_3$ introduces rule $r_{bc}$, which is forgotten in node $t_4$. Note that $C_{3,1,1} = \{\langle \emptyset, \emptyset \rangle \} \in C_{3,1,1}$ does not have a “successor row” in table $\tau_4$ since $r_{bc}$ is not satisfied (see L11 and Definition 2). Table $\tau_5$ is then the result of a chain of introduce nodes, and contains for each witness set $M_{6,i}$ every possible counter-witness set $C_{6,i,j} \subseteq M_{6,i}$. We now discuss table $\tau_{12}$, intuitively containing a (projection of (counter-))witnesses of $\tau_{10}$, which satisfy rule $r_{bc}$ after introducing rule $r_{c_1}$. Observe that there is no succeeding witness set for $M_{6,2} = \{e_{ab}\}$ in $\tau_{10}$ (nor $\tau_{12}$), since $e_{ab} \in M_{6,2}$, but $a_0 \notin M_{6,2}$ (required to satisfy $r_5$). Rows $u_{12,1}, u_{12,4}$ form successors of $u_{6,3}$, while rows $u_{12,2}, u_{12,5}$ succeed $u_{6,1}$, since counter-witness set $C_{6,1,1}$ has no succeeding row in $\tau_{10}$ because it does not satisfy $r_5$. Remaining rows $u_{12,3}, u_{12,6}$ have “origin” $u_{6,4}$ in $\tau_6$.

![Figure 3](image-url) Extends Figure 2. The figure illustrates a tree decomposition $T$ of the semi-incidence graph $S(P)$ for program $P$ from Example 1 (center). Selected DP tables after $\text{DP}_{\text{MOD}}$ (left) and after $\text{DP}_{\text{SINC}}$ (right) for nice tree decomposition $T$. We explain details of the figure in Example 4.

4. **DynASP2.5: Towards a III Traversal DP Algorithm**

The classical DP algorithm $\text{DP}_{\text{SINC}}$ (Step 3 of Figure 4) follows a single traversal approach. It computes both witnesses and counter-witnesses by traversing the given TD exactly once. In particular, it stores exhaustively all potential counter-witnesses, even those counter-witnesses where the witnesses in the table of a node cannot be extended in the parent node. In addition, there can be a high number of duplicates among the counter-witnesses, which are stored repeatedly.
In this section, we propose a multi-traversal approach (\(\text{DPL}\)) for DP on TDs and a new implementation (DynASP2.5), which fruitfully adapts and extends ideas from a different domain [17]. From a theoretical perspective, our novel algorithm \(\text{DPL}\) is multi-variate practically, considering treewidth together with the maximum number of witnesses in a table that lead to a solution at the root. Intuitively, this subtle difference can yield practical advantages, especially if the number of relevant witnesses in a table that lead to a solution is subexponential in the treewidth. Therefore, our novel algorithm is designed to allow for an early cleanup (purging) of witnesses that do not lead to answer sets, which in consequence (i) avoids to construct expendable counter-witnesses. Moreover, multiple traversals enable us to store witnesses and counter-witnesses separately, which in turn (ii) avoids storing counter-witnesses duplicately and (iii) allows for highly space efficient data structures (pointers) in practice when linking witnesses and counter-witnesses together. Figure 4 (right, middle) presents the control flow of the new multi-traversal approach DynASP2.5, where \(\text{DPL}\) introduces a much more elaborate computation in Step 3.

4.1. The Algorithm

Our algorithm (\(\text{DPL}\)) executed as Step 3 runs \(\text{DP}_{\text{MOD}}\), \(\text{DP}_{\text{CMOD}}\), and \(\text{DPL}_{\text{MOD,CMod}}\) in three traversals (3.I, 3.II, and 3.III) as follows:

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

3.II. For this step, let \(\text{CMOD}\) be a table algorithm computing only counter-witnesses of \(\text{SINC}\) (blue parts of Algorithm 2). We execute \(\text{DP}_{\text{CMOD}}\), for all witnesses, compute counter-witnesses at once and store the resulting tables in \(\text{CMOD-Tabs}[\cdot]\). For every node \(t\), table \(\text{CMOD-Tabs}[t]\) contains counter-witnesses to witness being \(C\)-minimal. Again, irrelevant rows are removed ("Purge non-counter-witnesses").

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

We already explained the table algorithms \(\text{DP}_{\text{MOD}}\) and \(\text{DP}_{\text{CMOD}}\) in the previous section. The main part of our multi-traversal algorithm is the algorithm \(\text{DPL}_{\text{MOD,CMod}}\) based on the general algorithm \(\text{DPL}_{W,C}\) (Algorithm 3) with \(W = \text{MOD}, C = \text{CMOD}\), which links those separate tables together. Before we quickly discuss the core of \(\text{DPL}_{W,C}\) in Lines 6–10, note that Lines 3–5 introduce auxiliary definitions. Line 3 combines rows of the child
Algorithm 3: Algorithm DPL$W,C(T, W\text{-Tabs}, C\text{-Tabs})$ for linking counter-witnesses to witnesses.

1. Nice TD $T = (T, \chi)$ with $T = (N, r, n)$ of a graph $S(P)$, and mappings $W\text{-Tabs}[\cdot], C\text{-Tabs}[\cdot]$. 
   **Out**: $W,C\text{-Tabs}$: maps node $t \in T$ to some pair $(W_t, C_t)$ with $W_t \in W\text{-Tabs}[t], C_t \in C\text{-Tabs}[t]$.

2. Child-Tables: $:= \{W_t, C\text{-Tabs}[t'] \mid t' \text{ is a child of } t \text{ in } T\}$
   /* We use the following technical abbreviations below
   For set $l = \{1, \ldots, n\}$ and sets $S_i$:
   $\prod_{i \in l} S_i := S_1 \times \cdots \times S_n = \{(s_1, \ldots, s_n) : s_i \in S_i\}$.
   For $\prod_{i \in l} S_i$: $\prod_{i \in l} S_i := \{\{(s_1) \ldots, s_n\} : (s_1, \ldots, s_n) \in \prod_{i \in l} S_i\}$
   If for each $S \in \prod_{i \in l} S_i$ and $\{s_i\} \in S$, the element $s_i$ is a pair
   (witness,counter-witness),
   $f_w(S) := \bigcup_{(W, C) \in S}\{\{W\} \}$ restricts $S$ to the witness parts and
   $f_cw(S) := \bigcup_{(W, C) \in S}\{\{C\} \}$ restricts $S$ to the counter-witness parts.
   */
   /* Get for a node $t$ tables of (preceding) combined child rows (CCR)
   */

3. CCR$_t := \prod_{i \in \text{Child-Tabs}[t]}$
   /* Get for a row $\bar{u}$ its combined child rows (origins) */

4. orig($\bar{u}$) := $\{S \mid S \in \text{CCR}_t, \bar{u} \in S, = W(t, \chi(t), P_t, at, f_w(S))\}$ /* Get for a
   table $S$ of combined child rows its successors (evolution) */

5. evol$_t(S) := \{\bar{u} \mid \bar{u} \in S, = C(t, \chi(t), P_t, at, f_cw(S))\}

6. for $t$ iterate in post-order($T, n$) do

   /* Compute counter-witnesses ($\prec$-smaller rows) for a witness set $M$
   */

   7. subs$_t(f, M, S) := \{\bar{u} \mid \bar{u} \in C\text{-Tabs}[t], \bar{u} \in \text{evol}_t(f(S)), \bar{u} = (C, \cdots), C \prec M\}$

   8. /* Link each witness $\bar{u}$ to its counter-witnesses and store the
   results */

   9. $W_t, C\text{-Tabs}[t] := \{(\bar{u}, \text{subs}_t(f_w, M, S) \cup \text{subs}_t(f_cw, M, S)) \mid \bar{u} \in W\text{-Tabs}[t], \bar{u} =$

   10. $\langle M, \cdots\rangle, S \in \text{orig}_t(u)\}$

nodes of given node $t$, which is achieved by a product over sets, where we drop the
order and keep sets only. Line 4 concerns determining for a row $\bar{u}$ its origins (finding
preceding combined rows that lead to $\bar{u}$ using table algorithm $W$). Line 5 covers deriving
succeeding rows for a certain child row combination its evolution rows via algorithm $C$.
In an implementation, origin as well as evolution are not computed, but represented
via pointer data structures directly linking to $W\text{-Tabs}[\cdot]$ or $C\text{-Tabs}[\cdot]$, respectively. Then,
the table algorithm $DPL_{W,C}$ applies a post-order traversal and links witnesses to counter-
witnesses in Line 10. $DPL_{W,C}$ searches for origins (orig) of a certain witness $\bar{u}$, uses
the counter-witnesses ($f_cw$) linked to these origins, and then determines the evolution (evol) in
order to derive counter-witnesses (using subs) of $\bar{u}$.

4.2. An Elaborated Example

Example 5. Let $k$ be some integer and $P_k$ be some program that contains the following rules
$r_c := \{a_1, \ldots, a_k\} \prec f, r_2 := \neg a_2, \ldots, r_k := \neg a_k$, and $r_f := \neg f$ and $r_{cf} :=$
$\{f\} \prec \cdot$. The rules $r_1, \ldots, r_k$ simulate that only certain subsets of $\{a_1, \ldots, a_k\}$ are allowed.
Rules $r_f$ and $r_{cf}$ enforce that $f$ is set to true. Let $T = (T, \chi, t_s)$ be a TD of the semi-incidence
graph $S(P_k)$ where $T = (V, E)$ with $V = \{t_1, t_2, t_3\}, E = \{(t_1, t_2), (t_2, t_3)\}$,
$\chi(t_1) = \{a_1, \ldots, a_k, f, r_c, r_{cf}\}, \chi(t_2) = \{a_1, \ldots, a_k, r_2, \ldots, r_k, r_f\}$, and $\chi(t_3) = \emptyset$. Figure 5
(left) illustrates the tables for program $P_k$ after $DPL_{\text{SINC}}$, whereas Figure 5 (right) presents tables
using $M\text{-DP}_{\text{SINC}}$, which are exponentially smaller in $k$, mainly due to cleanup. Observe that Traversal
3.II $M\text{-DP}_{\text{SINC}}$, “temporarily” materializes counter-witnesses only for $t_1$, presented in table $t_1$\textsuperscript{CMOD}
Hence, using multi-traversal algorithm $M\text{-DP}_{\text{SINC}}$ results in an exponential speedup. Note that we
can trivially extend the program such that we have the same effect for a TD of minimum width and
even if we take the incidence graph. In practice, programs containing the rules above frequently occur when encoding by means of saturation [3]. The program \( P_3 \) and the TD \( T \) also reveal that a different TD of the same width, where \( f \) occurs already very early in the bottom-up traversal, would result in a smaller table \( \tau_1 \) even when running \( \text{DP}_{\text{SINC}} \).

![Figure 5. Selected DP tables after \( \text{DP}_{\text{SINC}} \) (left) and after \( \text{M-DP}_{\text{SINC}} \) (right) for TD \( T \).](image)

4.3. Correctness and Runtime of DynASP2.5

**Theorem 1.** For a program \( P \) of semi-incidence treewidth \( k := tw(S(P)) \), the algorithm \( \text{M-DP}_{\text{SINC}} \) is correct and runs in time \( O(2^{3k+2} \cdot ||P||) \).

We first sketch the outline of the proof.

**Proof (Sketch).** We sketch the proof idea for enumerating answer sets of disjunctive ASP programs by means of \( \text{M-DP}_{\text{SINC}} \). Below we present a more detailed approach to establish the correctness. Let \( P \) be a disjunctive program and \( k := tw(S(P)) \). We establish a reduction \( R(P,k) \) of \( \text{ENUMASP} \) to \( \text{ENUMMINSAT1} \), such that there is a one-to-one correspondence between answer sets and models of the formula, more precisely, for every answer set \( M \) of \( P \) and for the resulting instance \((F,k') = R(P,k) \) the set \( M \cup \{\text{sol}\} \) is a subset-minimal model of \( F \) and \( k' = tw(I(F)) \) with \( k' \leq 7k + 2 \). We compute in time \( 2^O(k^3) \cdot ||F|| \) a TD of width at most \( k' \) [26] and add \( \text{sol} \) to every bag. Using a table algorithm designed for \( \text{SAT} \) [32] we compute witnesses and counter-witnesses. Observe that witnesses and counter-witnesses are essentially the same objects in a table algorithm for computing subset-minimal models of a propositional formula, since witnesses and counter-witnesses of traversals 3.I and 3.II, respectively, are essentially the same objects in a table algorithm for computing subset-minimal models of a propositional formula. Conceptually, one could also modify \( \text{MOD} \) for this task. In order to finally show correctness of linking counter-witnesses to witnesses as presented in \( \text{DP}_{\text{MOD-MOD}} \), we have to extend earlier work [17], (Theorem 3.25 and 3.26). Therefore, we enumerate subset-minimal models of \( F \) by following each witness set containing \( \text{sol} \) at the root having counter-witnesses \( \emptyset \) back to the leaves. This runs in time \( O(2^{n(3k+2)} \cdot ||P||) \), c.f., [11,17]. A more involved (direct) proof, allows to decrease the runtime to \( O(2^{3k+2} \cdot ||P||) \) (even for choice rules).

In the following of this section, we go into more details to provide more depth arguments to support our statement above. Bliem et al. [17] have shown that augmentable \( \mathcal{W} \)-Tabs can be transformed into \( \mathcal{W}, \mathcal{W} \)-Tabs, which easily allows reading off subset-minimal solutions starting at the table \( \mathcal{W}, \mathcal{W} \)-Tabs[\( n \)] for TD root \( n \). We follow their concepts and define a slightly extended variant of augmentable tables. Therefore, we reduce the problem of enumerating disjunctive programs to \( \text{ENUMMINSAT1} \) and show that the resulting tables of algorithm \( \text{MOD} \) (see Algorithm 2) are augmentable. In the end, we apply an earlier theorem [17] transforming \( \text{MOD} \)-Tabs obtained by \( \text{DP}_{\text{MOD}} \) into \( \text{MOD}, \text{MOD} \)-Tabs.
via the augmenting function \( \text{aug}(\cdot) \) proposed in their work. To this extent, we use auxiliary definitions \( \text{Child-Tabs}_t \), \( \text{orig}_t(\cdot) \) and \( \text{evol}_t(\cdot) \) specified in Algorithm 3.

**Definition 6.** Let \( T = (\mathcal{T}, \chi) \) be a TD where \( \mathcal{T} = (\mathcal{N}, \cdot, \cdot) \), \( \mathcal{W} \) be a table algorithm, \( t \in \mathbb{N} \), and \( \tau \in \mathcal{W}\text{-Tabs}[t] \) be the table for node \( t \). For tuple \( \vec{u} = (\mathcal{M}, \sigma, \cdot, \cdot) \in \tau \), we define \( \alpha(\vec{u}) := \mathcal{M}, \beta(\vec{u}) := \sigma \). We inductively define

\[
\alpha^*(\tau) := \bigcup_{\vec{u} \in \mathcal{T}} \alpha(\vec{u}) \cup \bigcup_{\tau' \in \text{Child-Tabs}_t} \alpha^*(\tau'), \quad \text{and}
\beta^*(\tau) := \bigcup_{\vec{u} \in \mathcal{T}} \beta(\vec{u}) \cup \bigcup_{\tau' \in \text{Child-Tabs}_t} \beta^*(\tau').
\]

Moreover, we inductively define the extensions of a row \( \vec{u} \in \tau \) as

\[
E(\vec{u}) := \left\{ \{\vec{u}\} \cup U \mid U \in \bigcup_{\{\vec{u}'_1, \ldots, \vec{u}'_k\} \in \text{orig}_t(\vec{u})} \{\tau_1 \cup \cdots \cup \tau_k \mid \tau_i \in E(\vec{u}_i) \text{ for all } 1 \leq i \leq k \right\}.
\]

**Remark 1.** Any extension \( U \in E(\vec{u}) \) contains \( \vec{u} \) and exactly one row from each table that is a descendant of \( \tau \). If \( \vec{u} \) is a row of a leaf table, \( E(\vec{u}) = \{\{\vec{u}\}\} \) since \( \text{orig}_t(\vec{u}) = \{\emptyset\} \) assuming \( \prod_{i \in \emptyset} S_i = \{()\} \).

**Definition 7.** Let \( \tau_u \) be the table in \( \mathcal{W}\text{-Tabs} \) for TD root \( n \). We define the set sol(\( \mathcal{W}\text{-Tabs} \)) of solutions of \( \mathcal{W}\text{-Tabs} \) as \( \text{sol}(\mathcal{W}\text{-Tabs}) := \{\alpha^*(U) \mid \vec{u} \in \tau_u, U \in E(\vec{u})\} \).

**Definition 8.** Let \( \tau \) be a table in \( \mathcal{W}\text{-Tabs} \) such that \( \tau_1', \ldots, \tau_k' \) are the child tables \( \text{Child-Tabs}_t \), and let \( \vec{u}_i, \vec{v}_i, \in \tau \). We say that \( x \in X(\vec{u}) \) has been \( X \)-illegally introduced at \( \vec{u} \) if there are \( \{\vec{u}_1', \ldots, \vec{u}_k'\} \in \text{orig}_t(\vec{u}) \) such that for some \( 1 \leq i \leq k \) it holds that \( x \notin X(\vec{u}_i) \) while \( x \in X^*(\tau_i') \). Moreover, we say that \( x \in X(\vec{u}) \setminus X(\vec{u}) \) has been \( X \)-illegally removed at \( \vec{u} \) if there is some \( U \in E(\vec{u}) \) such that \( x \in X(U) \).

**Definition 9.** We call a table \( \tau \) augmentable if the following conditions hold:

1. For all rows of the form \( (\mathcal{M}, \cdot, \cdot, \cdot) \), we have \( \mathcal{C} = \emptyset \).
2. For all \( \vec{u}_i, \vec{v}_i, \in \tau \) with \( \vec{u} \neq \vec{v} \) it holds that \( \alpha(\vec{u}_i) \cup \beta(\vec{u}_i) \neq \alpha(\vec{v}) \cup \beta(\vec{v}) \).
3. For all \( \vec{u} = (\mathcal{M}, \sigma, \cdot, \cdot) \in \tau \), \( \{\vec{u}'_i, \ldots, \vec{u}'_k\} \in \text{orig}_t(\vec{u}) \), \( 1 \leq i < j \leq k \), \( I \in E(\vec{u}_i') \) and \( J \in E(\vec{u}_j') \) it holds that \( \alpha^*(I) \cap \alpha^*(J) \subseteq \mathcal{M} \) and \( \beta^*(I) \cap \beta^*(J) \subseteq \sigma \).
4. No element of \( \alpha^*(\tau) \) has been \( \alpha \)-illegally introduced and no element of \( \beta^*(\tau) \) has been \( \beta \)-illegally introduced.
5. No element of \( \alpha^*(\tau) \) has been \( \alpha \)-illegally removed and no element of \( \beta^*(\tau) \) has been \( \beta \)-illegally removed.

call \( \mathcal{W}\text{-Tabs} \) augmentable if all its tables are augmentable.

It is easy to see that \( \text{MOD}\text{-Tabs} \) are augmentable, that is, Algorithm 2 (\( \text{DP}_{\text{MOD}}(\cdot) \)) computes only augmentable tables.

**Observation 3.** \( \text{MOD}\text{-Tabs} \) are augmentable, since \( \text{DP}_{\text{MOD}}(\cdot) \) computes augmentable tables.

The following theorem establishes that we can reduce an instance of ENUMASp (restricted to disjunctive input programs) when parameterized by semi-incidence treewidth to an instance of ENUMMINSAT1 when parameterized by the treewidth of its incidence graph.

**Lemma 4.** Given a disjunctive program \( P \) of semi-incidence treewidth \( k = \text{tw}(S(P)) \). We can produce in time \( O(||P||) \) a propositional formula \( F \) such that the treewidth \( k' \) of the incidence graph \( I(F) \) of the incidence graph \( I(F) \) of a propositional formula \( F \) in CNF is the bipartite graph
that has the variables and clauses of $F$ as vertices and an edge $v \in c$ if $v$ is a variable that occurs in $c$ for some clause $c \in F$. If $F$ is not $k$-CNF and the answer sets of $P$ and subset-minimal models of $F^*$ are in a particular one-to-one correspondence. More precisely, $M$ is an answer set of $P$ if and only if $M \cup M_{\text{aux}} \cup \{\text{sol}\}$ is a subset-minimal model of $F$ where $M_{\text{aux}}$ is a set of additional variables occurring in $F$, but not in $P$ and variables introduced by Tseitin normalization.

**Proof.** Let $P$ be a disjunctive program of semi-incidence treewidth $k = tw(S(P))$. First, we construct a formula $\mathcal{F}$ consisting of a conjunction over formulas $\mathcal{F}_r$, $\mathcal{F}_{\text{imp}}$, $\mathcal{F}_{\text{sol}}$, $\mathcal{F}_{\text{min}}$ followed by Tseitin normalization of $\mathcal{F}$ to obtain $\mathcal{F}^*$. Among the atoms (Note that we do not distinguish between atoms and propositional variables in terminology here.) of our formulas will the atoms $\text{at}(P)$ of the program. Further, for each atom $a$ such that $a \in B_r^-(r)$ for some rule $r \in P$, we introduce a fresh atom $a'$. In the following, we denote by $Z'$ the set $\{z' : z \in Z\}$ for any set $Z$ and by $B_p := \bigcup_{r \in P} B_r^-$. Hence, $(B_p^-)'$ denotes a set of fresh atoms for atoms occurring in any negative body. Then, we construct the following formulas:

$$
\begin{align*}
F^*(r) := H_r & \lor \neg B_r^+ & \lor (B_r^-)''' & \quad \text{for } r \in P \\
F_{\text{imp}}(a) := & a \rightarrow a' & \quad \text{for } a \in B_p^- \\
F_{\text{sol}}(a) := & \text{sol} \rightarrow (a' \rightarrow a) & \quad \text{for } a \in B_p^- \\
F_{\text{min}} := & \neg \text{sol} & \quad \quad \bigvee_{a' \in (B_p^-)'} \\
F := & \bigwedge_{r \in P} F(r) & \bigwedge_{a \in B_p^-} F_{\text{imp}}(a) & \bigwedge_{a \in B_p^-} F_{\text{sol}}(a) & \bigwedge_{a \in B_p^-} F_{\text{min}}
\end{align*}
$$

Next, we show that $M$ is an answer set of $P$ if and only if $M \cup (\{N \cap B_p^\prime\}') \cup \{\text{sol}\}$ is a subset-minimal model of $\mathcal{F}$.

$(\Rightarrow)$: Let $M$ be an answer set of $P$. We transform $M$ into $Y := M \cup (\{N \cap B_p^\prime\}') \cup \{\text{sol}\}$. Observe that $Y$ satisfies all subformulas of $\mathcal{F}$ and therefore $\mathcal{F} \models F$. It remains to show that $Y$ is a minimal model of $\mathcal{F}$. Assume towards a contradiction that $Y$ is not a minimal model. Hence, there exists $X$ with $X \subs Y$. We distinguish the following cases:

1. $\text{sol} \in X$: By construction of $\mathcal{F}$ we have $X \models a' \leftrightarrow a$ for any $a' \in (B_p^-)'$, which implies that $X \cap \text{at}(P) \models p^M$. However, this contradicts our assumption that $M$ is an answer set of $P$.

2. $\text{sol} \not\in X$: By construction of $\mathcal{F}$ there is at least one atom $a \in B_p^-$ with $a' \in X$, but $a \not\in X$. Consequently, $X \cap \text{at}(P) \models p^M$. This contradicts again that $M$ is an answer set of $P$.

$(\Leftarrow)$: Given a formula $\mathcal{F}$ that has been constructed from a program $P$ as given above. Then, let $Y$ be a subset-minimal model of $\mathcal{F}$ such that $\text{sol} \in Y$. By construction we have for every $a' \in Y \cap (B_p^-)'$ that $a \in Y$. Hence, we let $M = \text{at}(P) \cap Y$. Observe that $M$ satisfies every rule $r \in P$ according to (A1) and is in consequence a model of $P$. It remains to show that $M$ is indeed an answer set. Assume towards a contradiction that $M$ is not an answer set. Then there exists a model $N \subs M$ of the reduct $p^M$. We distinguish the following cases:

1. $N$ is not a model of $P$: We construct $X := N \cup [Y \cap (B_p^-)']$ and show that $X$ is indeed a model of $F$. For this, for every $r \in P$ where $B^-_r \cap M \not\equiv \emptyset$ we have $X \models F^*(r)$, since $(Y \cap (B_p^-)') \subs X$ by definition of $X$. For formulas (A1) constructed by $F^*(r)$ using remaining rules $r$, we also have $X \models F^*(r)$, since $N \models \{r\}^M$. In conclusion, $X \models F$ and $X \subs Y$, and therefore $X$ contradicts $Y$ is a subset-minimal model of $\mathcal{F}$.

2. $N$ is also a model of $P$: Observe that then $X := N \cup [N \cap B_p^\prime]' \cup \{\text{sol}\}$ is also a model of $F$, which contradicts optimality of $Y$ since $X \subs Y$.

By Tseitin normalization, we obtain $F^\prime$, thereby introducing fresh atoms $l_{\phi}$ for each $a' \in (B_p^-)'$.
We can easily see that where \( \chi \). For any minimization. Moreover, by the correctness of the table algorithm Definition 10. Let \( \chi' \) be defined as follows. For each TD node \( t \),

\[
\chi'(t) := \bigcup_{u \in B_P \cap \chi(t)} \{a', l_d'\} \cup [\chi(t) \cap \text{at}(P)] \cup \{\text{sol}\} \cup \text{cl}(t)
\]

where

\[
\text{cl}(t) := \bigcup_{u \in B_P \cap \chi(t)} \{F^{\text{impb}}(a), F^{\text{sol}}(a), F^{\text{min}}(a), F^{\text{min}}(a), F^{\text{min}}(a)\} \cup \{F^{\text{min}}(a)\} \cup \bigcup_{r \in P, \chi(t)} \{F^r(r)\}.
\]

Observe that the Tseitin normalization is correct and that there is a bijection between models of formula \( F^r \) and formula \( F \).

Observe that our transformations runs in linear time and that the size of \( F^r \) is linear in \( \|P\| \). It remains to argue that \( tw(I(F^r)) \leq 7k + 2 \). For this, assume that \( T = (T, \chi, n) \) is an arbitrary but fixed TD of \( S(P) \) of width \( w \). We construct a new TD \( T' := (T, \chi', n) \) where \( \chi' \) is defined as follows. For each TD node \( t \),

\[
\chi'(t) := \bigcup_{u \in B_P \cap \chi(t)} \{a', l_d'\} \cup [\chi(t) \cap \text{at}(P)] \cup \{\text{sol}\} \cup \text{cl}(t)
\]

where

\[
\text{cl}(t) := \bigcup_{u \in B_P \cap \chi(t)} \{F^{\text{impb}}(a), F^{\text{sol}}(a), F^{\text{min}}(a), F^{\text{min}}(a), F^{\text{min}}(a)\} \cup \{F^{\text{min}}(a)\} \cup \bigcup_{r \in P, \chi(t)} \{F^r(r)\}.
\]

We can easily see that \( T' \) is indeed a tree decomposition for \( I(F^r) \) and that the width of \( T' \) is at most \( 7w + 2 \).

**Definition 10.** We inductively define an augmenting function \( \text{aug}(W\text{-Tabs}) \) that maps each table \( \tau \in W\text{-Tabs}[t] \) for node \( t \) from an augmentable table to a table in \( W, W\text{-Tabs}[t] \). Let the child tables of \( t \) be called \( t_1', \ldots, t_k' \). For any \( 1 \leq i \leq k \) and \( \bar{u} \in t_i \), we write \( \text{res}_{t_i}(\bar{u}) \) to denote \( \{\bar{v} \in \text{aug}(t_i') \mid a(\bar{v}) = a(\bar{u})\} \). We define \( \text{aug}(\tau) \) as the smallest table that satisfies the following conditions:

1. For any \( \bar{u} \in \tau \), \( \{\{\bar{u}_1\}, \ldots, \{\bar{u}_k\}\} \in \text{orig}_{t_i}(\bar{u}) \) and \( \{\{\bar{v}_1\}, \ldots, \{\bar{v}_k\}\} \in \prod_{1 \leq i \leq k} \text{res}_{t_i}(\bar{u}) \), there is a row \( \bar{v} \in \text{aug}(\tau) \) with \( a(\bar{v}) = a(\bar{u}) \) and \( \{\{\bar{v}_1\}, \ldots, \{\bar{v}_k\}\} \in \text{orig}_{t_i}(\bar{v}) \).

2. For any \( \bar{u}, \bar{v} \in \text{aug}(\tau) \) with \( \bar{u} = \langle \cdots, C \rangle \) such that \( a(\bar{v}) \subseteq a(\bar{u}) \), \( \{\{\bar{u}_1\}, \ldots, \{\bar{u}_k\}\} \in \text{orig}_{t_i}(\bar{u}) \) and \( \{\{\bar{v}_1\}, \ldots, \{\bar{v}_k\}\} \in \text{orig}_{t_i}(\bar{v}) \) the following holds: Let \( 1 \leq i \leq k \) with \( \bar{u}'_i = (\cdots, C_i, \bar{c}_i) \in (C_i \cup \{\bar{u}_i'\}) \) with \( C_i \subseteq a(\bar{v}) \), and \( 1 \leq j \leq k \) with \( \bar{c}_i \neq \bar{u}_i' \) or \( a(\bar{v}) \subseteq a(\bar{u}) \). Then, there is a row \( \bar{c} \in C \) with \( a(\bar{c}) \subseteq a(\bar{v}) \) if and only if \( \bar{c} \) is in \( \tau \) and \( \{\{\bar{c}_1\}, \ldots, \{\bar{c}_k\}\} \in \text{orig}_{t_i}(\bar{c}) \).

For \( W\text{-Tabs} \), we write \( \text{aug}(W\text{-Tabs}) \) to denote the result isomorphic to \( W, W\text{-Tabs} \) where each table \( \tau \) in \( W\text{-Tabs} \) corresponds to \( \text{aug}(\tau) \).

**Proposition 2.** Let \( W\text{-Tabs} \) be augmentable. Then,

\[
\text{sol}(\text{aug}(W\text{-Tabs})) = \{M \in \text{sol}(W\text{-Tabs}) \mid \exists M' \in \text{sol}(W\text{-Tabs}) : M' \subseteq M\}.
\]

**Proof (Sketch).** The proof follows previous work [17]. We sketch only differences from their work. Any row \( \bar{u} \in \tau \) of any table \( \tau \) not only consists of set \( a(\bar{u}) \) being subject to subset-minimization and relevant to solving ENUMASP. In addition, our definitions presented above also allow “auxiliary” sets \( \beta(\bar{u}) \) per row \( \bar{u} \), which are not subject to the minimization. Moreover, by the correctness of the table algorithm \( \text{SINC} \) above, we only
Theorem 2. EnumASP when the input is restricted to disjunctive programs can be solved in time $2^{2(k+4)} \cdot |P|$ computing $\text{aug}(\text{DP}_{\text{MOD}}(\cdot))$, where $k$ refers to the treewidth of $S(P)$.

Proof. First, we use reduction $R(P, k) = (F^*, k')$ defined in Lemma 4 to construct an instance of SAT given our disjunctive ASP program $P$. Note that $k' = \text{tw}(I(F^*)) \leq 7k + 2$.

Then, we can compute in time $2^{O(k^3)} \cdot |I(F^*)|$ a tree decomposition of width at most $k'$ [26]. Note that since we require to look for solutions containing $\text{sol}$ at the root, we modify each bag of $\mathcal{T}$ such that it contains $\text{sol}$. We call the resulting tree decomposition $\mathcal{T}'$. We compute $\text{aug}(\text{DP}_{\text{MOD}}(\mathcal{T}'))$ using formula $F^*$ as in Algorithm 2. Finally, by Proposition 2 and Lemma 4, we conclude that answer sets of $P$ correspond to $\{ M \in \text{sol}(\text{aug}(\text{DP}_{\text{MOD}}(\mathcal{T}'))) \mid \text{sol} \in M, \exists M' \in \text{sol}(\text{DP}_{\text{MOD}}(\mathcal{T}')) : M' \subset M \}$.

The complexity proof sketched in work by Bliem et al. [17] only cares about the runtime being polynomial. In fact, the algorithm can be carried out in linear time, following arguments from above, which leads to a worst-case runtime of $2^{2(k+4)} \cdot |P|$.

We can now even provide a “constructive definition” of the augmenting function $\text{aug}(\cdot)$.

Proposition 3. The resulting table $\text{aug}(\mathcal{W}\text{-Tabs})$ obtained via $\text{DP}_{\mathcal{W}}(\cdot)$ for any TD $\mathcal{T}$ is equivalent to the table $\text{DPL}_{\mathcal{W},\mathcal{W}}(\mathcal{T})$ as given in Algorithm 3.

Proof (Idea). Intuitively, Condition 1 of Definition 10 concerns completeness, i.e., ensures that no row is left out during the augmentation, and is ensured by Line 7 of Algorithm 3 since each $\vec{u} \in \mathcal{W}\text{-Tabs}$ is preserved. Condition 2 enforces that there is no missing counter-witness for any witness, and the idea is that whenever two witnesses $\vec{u}, \vec{v}$ are in a subset relation $(\alpha(\vec{v}) \subseteq \alpha(\vec{u}))$ and their corresponding linked counter-witnesses $(f_{\text{cw}})$ of the corresponding origins (orig) are in a strict subset relation, then there is some counter-witness $\vec{c}$ for $\vec{u}$ if and only if $\vec{c} \in \tau$ is the successor (evol) of this corresponding linked counter-witnesses. Intuitively, we cannot miss any counter-witnesses in $\text{DPL}_{\mathcal{W},\mathcal{W}}(\mathcal{T})$ required by Condition 2, since this required that there are two rows $\vec{u}, \vec{v} \in \tau$ with $\alpha(\vec{v}) = \alpha(\vec{u})$ for one table $\tau'$. Now, let the corresponding succeeding rows $\vec{u}, \vec{v} \in \tau$ (i.e., $\vec{u} \in \text{evol}(\{\vec{u}'\}), \vec{v} \in \text{evol}(\{\vec{v}'\}$), respectively) with $\alpha(\vec{v}) \not\subseteq \alpha(\vec{u}), \beta(\vec{v}) \not\subseteq \beta(\vec{u})$ and $\beta(\vec{v}) \not\subseteq \beta(\vec{u})$, mark the first encounter of a missing counter-witness. Since $\beta(\vec{v})$ is incomparable to $\beta(\vec{u})$, we conclude that the first encounter has to be in a table preceding $\tau$. To conclude, one can show that $\text{DPL}_{\mathcal{W},\mathcal{W}}(\mathcal{T})$ does not contain “too many” rows, which do not fall under Conditions 1 and 2.

Theorem 2 works not only for disjunctive ASP via reduction to EnumMinSAT1, where witnesses and counter-witnesses are derived with the same table algorithm MOD. In fact, one can also link counter-witnesses to witnesses by means of $\text{DPL}_{\mathcal{W},\mathcal{C}}(\cdot)$, thereby using table algorithms $\mathcal{W}, \mathcal{C}$ for computing witnesses and counter-witnesses, respectively. In order to show correctness of algorithm $\text{DPL}_{\text{MOD},\text{CMOD}}(\cdot)$ (Theorem 1) working for any ASP program, it is required to extend the definition of the augmenting function $\text{aug}(\cdot)$ such that it is capable of using two different tables.

Corollary 1. Problem EnumASP can be solved in time $f(k) \cdot |P|$ computing $\text{DPL}_{\text{MOD},\text{CMOD}}(\cdot)$, where $k$ refers to the treewidth of $S(P)$ and $f$ is a computable function.
5. Experimental Evaluation

5.1. 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. Therefore, we present additional implementation details of algorithm $M_{\text{DP}}$ into our prototypical multi-traversal solver DynASP2.5, including two variations (depgraph, joinsize TDs).

Even though one can construct a nice TD from a TD without increasing its width, one may artificially introduce additional atoms in a nice TD. This results in several additional intermediate join nodes among such artificially introduced atoms requiring a significant amount of total unnecessary computation in practice. On that account, we use tree decompositions that do not have to be nice. In order to still obtain a fixed-parameter linear algorithm, 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 row (witness) might be linked to different set of counter-witnesses depending on different predecessors of the row (hidden in set notation of the last line in Algorithm 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 counter-witness during traversal three. Together with efficient caches of counter-witnesses, DynASP2.5 reduces overhead due to clones in practice.

Dedicated data structures are vital. Sets of witnesses and satisfied rules are represented in the DynASP2.5 system via constant-size bit vectors. 32-bit integers are used to represent by value 1 whether 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 for now (practical memory limitations) that our approach works well on TDs of width $\leq 20$. Since state-of-the-art computers handle such constant-sized integers extremely efficient, DynASP2.5 allows for efficient projections and joins of rows, and subset checks in general. In order to not recompute counter-witnesses (in Traversal 3.II) for different witnesses, we use a three-valued notation of counter-witness sets consisting of atoms set to true (T) or false (F) or false but true in the witness set (TW) used to build the reduct. Note that the algorithm enforces that only (TW)-atoms are relevant, i.e., an atom has to occur in a default negation or choice rule.

Minimum width is not the only optimization goal when computing TDs by means of heuristics. Instead, using TDs where a certain feature value has been maximized in addition (customized TDs) works seemingly well in practice [12,33]. While DynASP2.5 ($M_{\text{DP}}$) 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 [34]. The variant DynASP2.5 joinsize minimizes bag sizes of child nodes of join nodes, c.f. [18].

5.2. Benchmark Set

In the following empirical evaluation, we consider the uniform Steiner tree problem (ST), which is a variant of a decision problem among the first well-known NP-hard problems [35]. We take public transport networks as input graphs. The instance graphs have been extracted from publicly available mass transit data feeds and split by transportation type, e.g., train, metro, tram, combinations [36]. We heuristically computed tree decompositions [18] and obtained decompositions of small width relatively fast unless detailed bus networks were present. Among the graphs considered were public transit networks of the cities London, Bangladesh, Timisoara, and Paris. Benchmarks, encodings, and results are available online at https://github.com/daajoe/dynasp_experiments/tree/ipec2017. We assumed for simplicity, that edges have unit costs, and randomly generated a set of terminals.
5.2.1. Steiner Tree Problem

The considered version of the uniform Steiner tree problem can be formalized as follows:

**Definition 11** (Uniform Steiner Tree Problem ($S_T$)). Let $G = (V, E)$ be a graph and $V_T \subseteq V$ be a set of vertices, called terminal vertices. Then, a uniform Steiner tree on $G$ is a tree $S_G = (V_S, E_S)$ with $V_S \subseteq V$ and $E_S \subseteq E$ such that

1. all terminal vertices are covered: $V_T \subseteq V_S$, and
2. the number of edges in $E_S$ forms a minimum: there is no tree, where all terminal vertices are covered that consists of no more than $|E_S| - 1$ many edges.

The uniform Steiner tree problem ($S_T$) asks to compute the uniform Steiner trees for $G$ and $V_T$.

**Example 6** (Encoding $S_T$ into an ASP program). Let $G = (V, E)$ be a graph and $V_T \subseteq V$ be a set of terminal vertices. We encode the problem $S_T$ into an ASP program as follows: Among the atoms of our program will be an atom $a_v$ for each vertex $v \in V_T$, and an atom $e_{vw}$ for each edge $vw \in E$ assuming $v < w$ for an arbitrary, but fixed total ordering $<$ among $V$. Let $s$ be an arbitrary vertex $s \in V_T$. We generate program $P(G, V_T) := \{ e_{vw} \leftarrow ; \leftarrow e_{vw} \mid vw \in E \} \cup \{ a_v \leftarrow a_w, e_{vw}; a_w \leftarrow a_v, e_{vw} \mid vw \in E, v < w \} \cup \{ \leftarrow \neg a_v \mid vs. \in V_T \} \cup \{ a_s \leftarrow \}$. It is easy to see that the answer sets of the program and the uniform Steiner trees are in a one-to-one correspondence.

5.2.2. Considered Encodings

We provide the encoding as non-ground ASP program, which allows to state first-order variables that are instantiated by present constants. This allows us to provide a readable encoding and avoid stating an encoder as an imperative program. Note that for certain non-ground programs, treewidth guarantees are known [37]. In the experiment, we used gringo [38] as grounder. An encoding for the problem $S_T$ is depicted in Listing 1 and assumes a specification of the graph (via edge) and the terminal vertices (terminalVertex) as well as the number (numVertices) of vertices. Note that this encoding is quite compact and non-ground and therefore it contains first-order variables. Prior to solving, these first-order variables are instantiated by a grounder, which results in a ground program. Interestingly, the encoding in Listing 1 is based on the saturation technique [3] and in fact outperformed a different encoding presented in Listing 2 on all our instances using both solvers, Clasp and DynASP2.5. At first sight, this observation seems quite surprising, however, we benchmarked on more than 60 graphs with 10 varying decompositions for each solver variant and additional configurations and different encodings for Clasp.

**Listing 1.** Encoding for $S_T$.

```
vertex(X) ← edge(X,_).
vertex(Y) ← edge(_,Y).
edge(X,Y) ← edge(Y,X).

0 { selectedEdge(X,Y) } 1 ← edge(X,Y), X < Y.
s1(X) ∨ s2(X) ← vertex(X).
saturate ← selectedEdge(X,Y), s1(X), s2(Y), X < Y.
saturate ← selectedEdge(X,Y), s2(X), s1(Y), X < Y.
saturate ← N #count{ X : s1(X), terminalVertex(X) }, numVertices(N).
saturate ← N #count{ X : s2(X), terminalVertex(X) }, numVertices(N).
s1(X) ← saturate, vertex(X).
s2(X) ← saturate, vertex(X).
← not saturate.
#minimize{ 1,X,Y : selectedEdge(X,Y) }.
```
5.3. Experimental Evaluation

We performed experiments to investigate the runtime behavior of DynASP2.5 and its variants, in order to evaluate whether our multi-traversal approach can be beneficial and has practical advantages over the classical single traversal approach (DynASP2). Further, we considered the dedicated ASP solver Clasp 3.3.0 (Clasp is available at https://github.com/potassco/clasp/releases/tag/v3.3.0.). 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. Note that we support the most frequently used SModels input format [39] for our implementation. For our experiments, we used gringo [38] for both Clasp and DynASP2.5 and therefore do not compare the time needed for grounding, since it is exactly the same for both solvers.

5.3.1. Benchmark Environment

The experiments presented ran on an Ubuntu 16.04.1 LTS Linux cluster of 3 nodes with two Intel Xeon E5-2650 CPUs of 12 physical cores each at 2.2 GHz clock speed and 256 GB RAM. All solvers have been compiled with GCC version 4.9.3 and executed in single core mode.

5.3.2. Runtime Limits

We mainly inspected the CPU time using the average over five runs per instance (five fixed seeds allow certain variance for heuristic TD computation). For each run, we limited the environment to 16 GB RAM and 1200 s CPU time. We used Clasp with options “--stats=2 --opt-strategy=usc,pmres,disjoint,stratify --opt-usc-shrink=min -q”, which enable improvements for unsatisfiable cores [40], and disabled solution printing/recording. We also benchmarked Clasp with branch-and-bound, which was, however, outperformed by the unsatisfiable core options on all our instances. In fact, without using unsatisfiable core advances, Clasp timed out on almost every instance.

5.3.3. Summary of the Results

The upper plot in Figure 6 shows the result of always selecting the best among five TDs, whereas the lower plot concerns about the median runtime among those runs.
Figure 6. Runtime results illustrated as cumulated solved instances. The y-axis labels consecutive integers that identify instances. The x-axis depicts the runtime. The instances are ordered by running time, individually for each solver. The first plot (top) shows the best results among five TDs and the lower plot (bottom) depicts the median runtime among five TDs.

The upper table of Table 1 reports on the minimum running times (TD computation and Traversals 3.I, 3.II, 3.III) among the solved instances and the total PAR2 scores. We consider an instance as solved, if the minimum and median PAR2 score is below the timeout of 1200 s. The lower table of Table 1 depicts the median running times over solved instances and the total median PAR2 scores. 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 state-of-the-art ASP solver Clasp, DynASP2.5 is capable of additionally delivering the number of optimal solutions. In particular, variant “depgraph” shows promising runtimes.
Table 1. The table illustrates detailed benchmark results, where data for the best possible among all five runs is depicted (top) and results for the median run are shown (bottom). All columns illustrate values in seconds, except the first two columns. Column “Solver” indicates a solving system or configuration and “Solved” refers to the number of solved instances. Then, columns “TD”, “3.I”, “3.II”, and “3.III” list the total runtime in seconds over all solved instances for computation involving obtaining TDs, Pass 3.I, Pass 3.II, and Pass 3.III, respectively. Column “Σ” refers to the total runtime in seconds over all solved instances. The last column depicts the PAR2 score over all instances, where timeouts are treated as two times the timeout (“PAR2 score”). Bold-face text indicates the best result in the corresponding column.

<table>
<thead>
<tr>
<th>Solver</th>
<th>Solved</th>
<th>Runtimes (Best) among Solved Instances</th>
<th>PAR2 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TD</td>
<td>3.I</td>
</tr>
<tr>
<td>Clasp 3.3.0</td>
<td>35</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DynASP2</td>
<td>13</td>
<td>7.96 (0.2%)</td>
<td>-</td>
</tr>
<tr>
<td>DynASP2.5</td>
<td>41</td>
<td>21.68 (0.2%)</td>
<td>130.10 (1.4%)</td>
</tr>
<tr>
<td>“depgraph”</td>
<td>45</td>
<td>408.72 (4.0%)</td>
<td>138.21 (1.4%)</td>
</tr>
<tr>
<td>“joinsize”</td>
<td>37</td>
<td>22.82 (0.3%)</td>
<td>120.19 (1.3%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solver</th>
<th>Solved</th>
<th>Runtimes (median) among solved instances</th>
<th>PAR2 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>TD</td>
<td>3.I</td>
</tr>
<tr>
<td>Clasp 3.3.0</td>
<td>34</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DynASP2</td>
<td>8</td>
<td>8.74 (0.2%)</td>
<td>-</td>
</tr>
<tr>
<td>DynASP2.5</td>
<td>32</td>
<td>21.91 (0.2%)</td>
<td>140.08 (1.3%)</td>
</tr>
<tr>
<td>“depgraph”</td>
<td>38</td>
<td>473.12 (4.1%)</td>
<td>146.53 (1.3%)</td>
</tr>
<tr>
<td>“joinsize”</td>
<td>32</td>
<td>25.47 (0.2%)</td>
<td>129.41 (1.3%)</td>
</tr>
</tbody>
</table>

5.4. A Brief Remark on Instances without Optimization

Our algorithm (M-DP SINC) is a multi-variate algorithm. The idea of avoiding to construct counter-witnesses focuses on situations were many models will be removed in the first traversal. This happens when (i) we have instances with only few models or (ii) we have optimization statements in our instances (minimizing cardinality of answer sets with respect to certain literals). The section above confirmed this expectation on Steiner tree instances. Technically, one might ask whether DynASP2.5 also improves over its predecessor when solving common graph problems that do not include optimization on the solution size such as variants of graph coloring, dominating set, and vertex cover. Clearly, the theoretical expectation would be that a multiple traversal algorithm (DynASP2.5) would not benefit much over a single traversal algorithm (DynASP2). This is indeed the case on graph problems. However, we omit details as it is not the focus of this paper and refer to a technical report [41] for benchmarks into this direction. There, DynASP2.5 was not able to significantly improve the running time required for problem solving, i.e., the additional overhead due to multiple traversals of DynASP2.5 did not pay off. One might ask a similar question for a comparison between Clasp and DynASP2.5. When taking Cases (i) and (ii) from above into account, we would expect the following. If we have few instances, Clasp does not have to run many steps with standard optimization strategies, which use branch-and-bound or unsatisfiable-based optimization running in multiple rounds of CDCL-search. If we have no optimization, Clasp will run only one round of CDCL-search and hence runs much faster than with optimization.

6. Conclusions

In this paper, we presented a novel approach for ASP solving based on ideas from parameterized complexity. Our algorithms runs in linear time assuming bounded treewidth of the input program. Our solver applies DP in three traversals, 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 traversals (incremental dynamic programming) can improve the runtime behavior. It might also be worth investigating if more general parameters than treewidth can be used [42]. Furthermore, it might be interesting to apply high level approaches on dynamic programming that include the use of existing technology to answer set programming [43,44] or whether such an approach pays off for other domains such as default logic [45], argumentation [46], or description logics [47].

Author Contributions: Authors contributed equally. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the Vienna Science and Technology Fund (WWTF) project ICT19-065 and the Austrian Science Fund (FWF) projects P30168 and P32830.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The source code of our solver is available at https://github.com/daajoe/dynasp/releases/tag/v2.5.0. Benchmarks, encodings, and results are available online at https://github.com/daajoe/dynasp_experiments/tree/ipec2017.

Acknowledgments: We would like to thank Arne Meier (Leibniz University Hannover, Germany) for organizing the Special Issue on Parameterized Complexity and Algorithms for Nonclassical Logics and a waiver for our submission. We also thank the two reviewers for their valuable comments and detailed feedback.

Conflicts of Interest: The authors declare no conflict of interest.

References


29. Schaub, T.; Woltran, S. Special Issue on Answer Set Programming. KI 2018, 32, 101–103. [CrossRef]


32. Samer, M.; Szeider, S. Algorithms for propositional model counting. J. Discrete Algorithms 2010, 8. [CrossRef]


