ASP and Projected Counting Meets Bounded Treewidth*

Johannes K. Fichte[1] and Markus Hecher[2]

[1] TU Dresden, Fakultät Informatik, Germany, johannes.fichte@tu-dresden.de
[2] TU Wien, Logic and Computation, Austria, hecher@dbai.tuwien.ac.at

Abstract. In this paper, we introduce a novel algorithm to solve the problem *projected answer set counting (#PAs)* for head-cycle-free programs. #PAs interests in outputting the number of distinct projected answer sets, which are answer sets that have been restricted to a given set of *projected atoms*. Our algorithms exploit small treewidth of the primal graph of the input instance by dynamic programming (DP). Therefore, we first establish a new algorithm to compute answer sets of head-cycle-free programs and then apply very recent results for projected counting. Our algorithms run in time double exponential of the treewidth and polynomial in the input size of the instance. Finally, we take the exponential time hypothesis (ETH) into account and establish lower bounds. In particular, one cannot expect (under ETH) to solve #PAs in polynomial time in the instance size while being single exponential in the treewidth.

Introduction

Answer Set Programming (ASP) is an important framework for declarative modeling and problem solving for hard combinatorial problems in artificial intelligence (AI) [7, 15]. In ASP, one typically solves a given problem by encoding the problem into a logic program whose solutions (answer sets) correspond to the solutions of the problem. Then, by computing the answer sets using a solver one obtains the solution to the original problem. If the problem involves a counting question, the corresponding answer set problem is #As, which recently received increasing attention and asks to output the number of answer sets of a given program. A natural abstraction of #As is to consider projected counting where we ask to count the answer sets of a program when the answer sets are restricted to a given set of projected atoms (#PAs) [1].

For disjunctive programs, the computational complexity of the plain counting problem is #·coNP-complete [13], which is even harder than counting the models of a Boolean formula (#SAT). Note that #SAT is already a considerably challenging task in practice. The problem #PAs for disjunctive programs is even harder, more precisely, complete for $\#\cdot\Sigma_2 P$ [9]. A well-established way in theory to solve such computationally hard problems is to exploit structural parameters of the instance such as bounded treewidth [8]. Intuitively, for treewidth

* This paper completes an extended abstract [9].

we consider a graph representation of a program and then assess how far this graph representation is from being a tree. The underlying idea is that problems on trees often allow for a monotonicity in the evaluation, which makes solving computationally easier. In ASP, we can employ graph representations such as the *primal graph*.

For disjunctive programs and the problem #PAs, a very recent abstract stated a triple exponential runtime in the treewidth and showed that under reasonable assumptions similar lower bounds hold [9]. However, when modeling in ASP one rarely requires the full power of disjunctive programs [15]. Hence, additional algorithms that work fast on restricted classes of programs can be a big step towards practical solving. A well-established fragment of programs is the class of *head-cycle-free* (hcf) programs [2], which intuitively requires the absence of cycles in a certain graph representation of the program. Hcf programs are widely exploited in practical ASP solving. Theoretically, finding an answer set for a given hcf program is of lower complexity than for disjunctive programs. Similar for projected counting, we can show that the problem #PAs is #·coNP when the input is restricted to hcf programs. If there are no projected atoms, then #PAs(hcf) is NP-complete. If all atoms are projected atoms, then #PAs(hcf) is #P-complete. Then, a challenging question for #PAs(hcf) is whether we can also significantly improve the runtime when exploiting treewidth. Therefore, we establish the following contributions:

- We present the classical complexity for #PAs(hcf).
- We establish a novel algorithm that finds an answer set of an hcf program in time single exponential in the treewidth.
- By exploiting treewidth, we introduce a formal framework that allows for counting projected answer sets. Therefore, we lift recent results from projected model counting in the domain of Boolean formulas to counting projected answer sets. We establish an algorithm that is double exponential in the treewidth if the input is restricted to hcf programs.
- Using the exponential time hypothesis (ETH) for lower bounds, we establish that #PAs(hcf) *cannot* be solved in time single exponential in the treewidth.

Related Work. Gebser et al. [16] considered projected enumeration for ASP. Aziz [1] introduced techniques to modify modern ASP solvers in order to count projected answer sets. Jakl et al. [20] presented DP algorithms that solve counting in time double exponential in the treewidth. Pichler et al. [28] investigated the complexity of extended programs, which contain for example weights, and other constructs and also presented DP algorithms for it. We employ ideas from their algorithms to solve hcf programs. Fichte et al. [13] presented algorithms to solve counting for the full standard syntax of modern solvers. Recently DP algorithms for projected #SAT and lower bounds were established [14].

Preliminaries

Basics and Combinatorics. For a set X, let 2^X be the *power set of* X consisting of all subsets Y with $\emptyset \subseteq Y \subseteq X$. Let $\boldsymbol{s}$ be a sequence of elements of X. When

we address the i-th element of the sequence $\boldsymbol{s}$ for a given positive integer i, we simply write $\boldsymbol{s}_{(i)}$. The sequence $\boldsymbol{s}$ *induces* an *ordering* $<_{\boldsymbol{s}}$ on the elements in X by defining the relation $<_{\boldsymbol{s}} := \{(\boldsymbol{s}_{(i)}, \boldsymbol{s}_{(j)}) \mid 1 \leq i < j \leq |\boldsymbol{s}|\}$. Given some integer n and a family of finite subsets $X_1, X_2, \ldots, X_n$. Then, the generalized combinatorial inclusion-exclusion principle [18] states that the number of elements in the union over all subsets is $\left|\bigcup_{j=1}^{n} X_j\right| = \sum_{I \subseteq \{1,\ldots,n\}, I \neq \emptyset} (-1)^{|I|-1} \left|\bigcap_{i \in I} X_i\right|$.

Computational Complexity. We assume familiarity with standard notions in computational complexity [27] and use counting complexity classes as defined by Durand et al. [10]. For parameterized complexity, we refer to standard texts [8]. Let Σ and Σ' be some finite alphabets. We call $I \in \Sigma^*$ an *instance* and $\|I\|$ denotes the size of I. Let $L \subseteq \Sigma^* \times \mathbb{N}$ and $L' \subseteq \Sigma'^* \times \mathbb{N}$ be parameterized problems. An *fpt-reduction* r from L to L' is a many-to-one reduction from $\Sigma^* \times \mathbb{N}$ to $\Sigma'^* \times \mathbb{N}$ such that for all $I \in \Sigma^*$ we have $(I, k) \in L$ if and only if $r(I, k) = (I', k') \in L'$ with $k' \leq g(k)$ for a fixed computable function $g : \mathbb{N} \to \mathbb{N}$, and r is computable in time $O(f(k)\|I\|^c)$ for a computable function f and constant c.

Answer Set Programming (ASP). We follow standard definitions of propositional disjunctive ASP. For comprehensive foundations, we refer to introductory literature [21]. Let ℓ, m, n be non-negative integers such that $\ell \leq m \leq n$, $a_1, \ldots, a_n$ be distinct propositional atoms. Moreover, we refer by *literal* to an atom or the negation thereof. A *program* Π is a set of *rules* of the form $a_1 \vee \cdots \vee a_\ell \leftarrow a_{\ell+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$. For a rule r, we let $H_r := \{a_1, \ldots, a_\ell\}$, $B_r^+ := \{a_{\ell+1}, \ldots, a_m\}$, and $B_r^- := \{a_{m+1}, \ldots, a_n\}$. We denote the sets of *atoms* occurring in a rule r and in a program Π by $\mathrm{at}(r) := H_r \cup B_r^+ \cup B_r^-$ and $\mathrm{at}(\Pi) := \bigcup_{r \in \Pi} \mathrm{at}(r)$. The program Π is *normal*, if $|H_r| \leq 1$ for every $r \in \Pi$. The *positive dependency digraph* D_Π of Π is the directed graph defined on the set of atoms from $\bigcup_{r \in \Pi} H_r \cup B_r^+$, where for every rule $r \in \Pi$ two atoms $a \in B_r^+$ and $b \in H_r$ are joined by an edge (a, b). A head-cycle of D_Π is an $\{a, b\}$-cycle[3] for two distinct atoms a, $b \in H_r$ for some rule $r \in \Pi$. Program Π is head-cycle-free if D_Π contains no head-cycle [2].

An *interpretation* I is a set of atoms. I *satisfies* a rule r if $(H_r \cup B_r^-) \cap I \neq \emptyset$ or $B_r^+ \setminus I \neq \emptyset$. I is a *model* of Π if it satisfies all rules of Π, in symbols $I \models \Pi$. The *Gelfond-Lifschitz (GL) reduct* of Π under I is the program Π^I obtained from Π by first removing all rules r with $B_r^- \cap I \neq \emptyset$ and then removing all $\neg z$ where $z \in B_r^-$ from the remaining rules r [17]. I is an *answer set* of a program Π if I is a minimal model of Π^I. Deciding whether a disjunctive program has an answer set (*consistency* problem) is Σ_2^{P}-complete [12]. If the input is restricted to normal programs, the complexity drops to NP-complete [3, 26]. A head-cycle-free (hcf) program Π can be translated into a normal program in polynomial time [2]. The following well-known characterization of answer sets is often invoked when considering normal programs [25]. Given a model I of a normal program Π and an ordering σ of atoms over I. An atom $a \in I$ is *proven* if there is a rule $r \in \Pi$

[3] Let $G = (V, E)$ be a digraph and $W \subseteq V$. Then, a cycle in G is a W-cycle if it contains all vertices from W.

with $a \in H_r$ where (i) $B_r^+ \subseteq I$, (ii) $I \cap B_r^- = \emptyset$, (iii) $I \cap (H_r \setminus \{a\}) = \emptyset$, and (iv) $b <_\sigma a$ for every $b \in B_r^+$.

Then, I is an *answer set* of Π if (i) I is a model of Π, and (ii) every atom $a \in I$ is proven. This characterization vacuously extends to hcf programs by applying the results of Ben-Eliyahu et al. [2].

Counting Projected Answer Sets. An instance is a pair (Π, P), where Π is a program and $P \subseteq \mathrm{at}(\Pi)$ is the set of *projection atoms*. The *projected answer set count* of Π with respect to P is the number of subsets $I \subseteq P$ such that $I \cup J$ is an answer set of Π for some set $J \subseteq \mathrm{at}(\Pi) \setminus P$. The *counting projected answer set problem (*#PAS*)* asks to output the projected answer set count of Π, i.e., $|\{I \cap P \mid I \in S\}|$ where S is the set of all answer sets of Π.

Example 1. Consider $\Pi := \{\overbrace{a \vee b \leftarrow}^{r_1}; \overbrace{c \vee e \leftarrow}^{r_2}; \overbrace{d \vee e \leftarrow b}^{r_3}; \overbrace{b \leftarrow e, \neg d}^{r_4}; \overbrace{d \leftarrow \neg b}^{r_5}\}$. It is easy to see that Π is head-cycle-free. The set $A = \{b, c, d\}$ is an answer set of Π. Consider the ordering $\sigma = \langle b, c, d\rangle$, from which we can prove atom b by rule r_1, atom c by rule r_2, and atom d by rule r_3. Further answer sets are $B = \{a, c, d\}$, $C = \{b, e\}$, and $D = \{a, d, e\}$.

Consider the set $P := \{d, e\}$ of projection atoms. When we project the answer sets to the set P, we only have the three answer sets $\{d\}$, $\{e\}$, and $\{d, e\}$. Hence, while Π has 4 answer sets, the projected answer set count of (Π, P) is 3.

Proposition 1 ($\star$[4]). *The problem* #PAS *is* $\#\cdot$NP*-complete when the input is restricted to head-cycle-free programs.*

Tree Decompositions (TDs). For basic terminology on graphs and digraphs, we refer to standard texts [6]. For a tree $T = (N, A, n)$ with root n and a node $t \in N$, we let $\mathsf{children}(t, T)$ be the sequence of all nodes t' in arbitrarily but fixed order, which have an edge $(t, t') \in A$. Let $G = (V, E)$ be a graph. A *tree decomposition (TD)* of graph G is a pair $\mathcal{T} = (T, \chi)$, where $T = (N, A, n)$ is a rooted tree, $n \in N$ the root, and χ a mapping that assigns to each node $t \in N$ a set $\chi(t) \subseteq V$, called a *bag*, such that the following conditions hold: (i) $V = \bigcup_{t \in N} \chi(t)$ and $E \subseteq \bigcup_{t \in N} \{\{u, v\} \mid u, v \in \chi(t)\}$; and (ii) for each r, s, t, such that s lies on the path from r to t, we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. Then, $\mathsf{width}(\mathcal{T}) := \max_{t \in N} |\chi(t)| - 1$. The *treewidth* $\mathsf{tw}(G)$ of G is the minimum $\mathsf{width}(\mathcal{T})$ over all tree decompositions $\mathcal{T}$ of G. For arbitrary but fixed $w \geq 1$, it is feasible in linear time to decide if a graph has treewidth at most w and, if so, to compute a TD of width w [4]. In order to simplify case distinctions in the algorithms, *we always use so-called nice TDs*, which can be computed in linear time without increasing the width [22] and are defined as follows. For a node $t \in N$, we say that type(t) is *leaf* if $\mathsf{children}(t, T) = \langle\rangle$; *join* if $\mathsf{children}(t, T) = \langle t', t''\rangle$ where $\chi(t) = \chi(t') = \chi(t'') \neq \emptyset$; *int* ("introduce") if $\mathsf{children}(t, T) = \langle t'\rangle$, $\chi(t') \subseteq \chi(t)$ and $|\chi(t)| = |\chi(t')| + 1$; *rem* ("remove") if $\mathsf{children}(t, T) = \langle t'\rangle$, $\chi(t') \supseteq \chi(t)$ and $|\chi(t')| = |\chi(t)| + 1$. If for every node $t \in N$, type$(t) \in \{\mathit{leaf}, \mathit{join}, \mathit{int}, \mathit{rem}\}$ and bags of leaf nodes and the root are empty, then the TD is called *nice*.

[4] Proofs of statements marked with "$\star$" are omitted for space reasons.

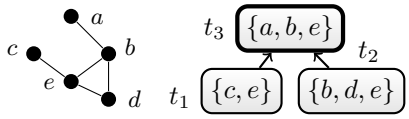

Fig. 1: Graph G_1 and a tree decomposition of G_1.

Example 2. Figure 1 illustrates a graph G_1 and a tree decomposition of G_1 of width 2. By a basic property[5] of TDs [22], the treewidth of G_1 is 2.

Dynamic Programming on TDs

In order to use TDs for ASP solving, we need a dedicated graph representation of ASP programs [20]. The *primal graph* G_Π of program Π has the atoms of Π as vertices and an edge $\{a, b\}$ if there exists a rule $r \in \Pi$ and $a, b \in \mathrm{at}(r)$.

Example 3. Recall program Π from Example 1 and observe that graph G_1 in Figure 1 is the primal graph G_Π of Π.

Let $\mathcal{T} = (T, \chi)$ be a TD of primal graph G_Π of a program Π. Further, let $T = (N, \cdot, n)$ and $t \in N$. The *bag-program* is defined as $\Pi_t := \{r \mid r \in \Pi, \mathrm{at}(r) \subseteq \chi(t)\}$, the *program below* t as $\Pi_{\leq t} := \{r \mid r \in \Pi_{t'}, t' \in \mathsf{post\text{-}order}(T, t)\}$, and the *program strictly below* t as $\Pi_{<t} := \Pi_{\leq t} \setminus \Pi_t$, where $\mathsf{post\text{-}order}(T, n)$ provides a sequence of nodes for tree T rooted at n in *post-order*. It holds that $\Pi_{\leq n} = \Pi_{<n} = \Pi$ [13]. Analogously, we define the *atoms below* t by $\mathrm{at}_{\leq t} := \bigcup_{t' \in \mathsf{post\text{-}order}(T,t)} \chi(t')$, and the *atoms strictly below* t by $\mathrm{at}_{<t} := \mathrm{at}_{\leq t} \setminus \chi(t)$.

Algorithms that decide consistency or solve #As [13, 20] proceed by dynamic programming along the tree decomposition (in post-order) where at each node of the tree information is gathered [5] in a table by a *local algorithm* $\mathbb{A}$. More generally, a *table* is a set of rows, where a *row* $\boldsymbol{u}$ is a sequence of fixed length. Similar as for sequences when addressing the i-th element, for a set U of rows (table) we let $U_{(i)} := \{\boldsymbol{u}_{(i)} \mid \boldsymbol{u} \in U\}$. The actual length, content, and meaning of the rows depend on the algorithm $\mathbb{A}$. Since we later traverse the tree decomposition repeatedly running different algorithms, we explicitly state $\mathbb{A}$-*row* if rows of this *type* are syntactically used for algorithm $\mathbb{A}$ and similar $\mathbb{A}$-*table* for tables. In order to access tables computed at certain nodes after a traversal as well as to provide better readability, we attribute tree decompositions with an additional mapping to store tables. Formally, a *tabled tree decomposition (TTD)* of graph G is a pair $\mathcal{T} = (T, \chi, \tau)$, where (T, χ) is a TD of G and τ maps nodes t of T to tables. If not specified otherwise, we assume that $\tau(t) = \{\}$ for every node t of T. When a TTD has been computed using algorithm $\mathbb{A}$ after traversing the entire decomposition, we call the decomposition the $\mathbb{A}$-TTD of the given input instance. Then, DP for ASP performs the following steps for a given program Π:

1. Compute a tree decomposition of primal graph G_Π of Π.

[5] The vertices e,b,d are all neighbors to each other in G_1.

Listing 1: Algorithm $\mathtt{DP}_{\mathbb{A}}((\Pi, P), \mathcal{T})$: Dynamic programming on TTD $\mathcal{T}$, c.f., [13].

In: Problem instance (Π, P), TTD $\mathcal{T} = (T, \chi, \iota)$ of G_Π such that n is the root of T and $\mathsf{children}(t, T) = \langle t_1, \ldots, t_\ell \rangle$.
Out: $\mathbb{A}$-TTD (T, χ, o) with $\mathbb{A}$-table mapping o

```
1 o ← empty mapping
2 for iterate t in post-order(T,n) do
3   o(t) ← 𝔸(t, χ(t), ι(t), (Π_t, P), ⟨o(t_1), ..., o(t_ℓ)⟩)
4 return (T, χ, o)
```

2. Run algorithm $\mathtt{DP}_{\mathbb{A}}$ (see Listing 1). It takes a tabled TD $\mathcal{T} = (T, \chi, \iota)$ with $T = (N, \cdot, n)$ and traverses T in post-order. At each node $t \in N$ it computes a new $\mathbb{A}$-table $o(t)$ by executing the algorithm $\mathbb{A}$. The algorithm $\mathbb{A}$ has a "local view" on the computation and can access only t, the atoms in the bag $\chi(t)$, the bag-program Π_t, and child $\mathbb{A}$-table $o(t')$ for any child t' of t.[6]
3. Print the solution by interpreting table $o(n)$ for root n of the resulting $\mathbb{A}$-tabled tree decomposition (T, χ, o).

Then, the actual computation of algorithm $\mathbb{A}$ is a somewhat technical case distinction of the types $\mathrm{type}(t)$ we can have when considering node t. Algorithms for counting answer sets of disjunctive programs [20] and its extensions have already been established. Implementations of these algorithms can be useful also for solving [13], but the running time is clearly double exponential time in the treewidth in the worst case. We, however, establish an algorithm ($\mathbb{PHC}$) that is restricted to hcf programs. The runtime of our algorithm is factorial in the treewidth and therefore faster than previous algorithms. Our constructions are inspired by ideas used in previous dynamic programming algorithms [28]. In the following, we first present the (local) algorithm for the problem of deciding whether an hcf program has an answer set (consistency). In the end, this algorithm outputs a new tabled tree decomposition, which we later reuse to solve our actual counting problem. Note that the tree decomposition itself remains the same, but for readability, we keep the computed tables and nodes aligned.

Consistency of Hcf Programs

We can use algorithm $\mathtt{DP}_{\mathbb{PHC}}$ to decide the consistency problem for hcf programs and simply specify our new local algorithm ($\mathbb{PHC}$) that "transforms" tables from one node to another. As graph representation we use the primal graph. The idea is to implicitly apply along the tree decomposition the characterization of answer sets by Lin et al. [25] extended to hcf programs [2]. To this end, we store in table $o(t)$ at each node t rows of the form $\langle I, \mathcal{P}, \sigma \rangle$. The first position consists of an interpretation I restricted to the bag $\chi(t)$. For a sequence $\boldsymbol{u}$, we write $\mathcal{I}(\boldsymbol{u}) := \boldsymbol{u}_{(1)}$ to address the *interpretation part*. The second position

[6] Note that in Listing 1, $\mathbb{A}$ takes in addition as input the set P and table ι_t. We will later reuse this listing. Then, P represents the set of projected atoms and ι_t is a table at t from an earlier traversal.

Listing 2: Local algorithm $\mathbb{PHC}(t, \chi_t, \cdot, (\Pi_t, \cdot), \langle \tau_1, \ldots \rangle)$.

In: Node t, bag χ_t, bag-program Π_t, $\langle \tau_1, \ldots \rangle$ is the sequence of $\mathbb{PHC}$-tables of children of t. **Out:** $\mathbb{PHC}$-table τ_t.

1 **if** type$(t) =$ *leaf* **then** $\tau_t \leftarrow \{\langle \emptyset, \emptyset, \langle\rangle \rangle\}$
2 **else if** type$(t) =$ *int and* $a \in \chi_t$ *is the introduced atom* **then**
3 $\quad \tau_t \leftarrow \{\langle J, \mathcal{P} \cup \mathsf{proven}(J, \sigma', \Pi_t), \sigma' \rangle \quad | \ \langle I, \mathcal{P}, \sigma \rangle \in \tau_1,\ J \in \{I, I_a^+\},\ J \models \Pi_t,\ \sigma' \in \mathsf{ords}(\sigma, \{a\} \cap J)\}$
4 **else if** type$(t) =$ *rem and* $a \notin \chi_t$ *is the removed atom* **then**
5 $\quad \tau_t \leftarrow \{\langle I_a^-, \mathcal{P}_a^-, \sigma_a^\sim \rangle \quad | \ \langle I, \mathcal{P}, \sigma \rangle \in \tau_1, a \in \mathcal{P} \cup (\{a\} \setminus I)\}$
6 **else if** type$(t) =$ *join* **then**
7 $\quad \tau_t \leftarrow \{\langle I, \mathcal{P}_1 \cup \mathcal{P}_2, \sigma \rangle \quad | \ \langle I, \mathcal{P}_1, \sigma \rangle \in \tau_1, \langle I, \mathcal{P}_2, \sigma \rangle \in \tau_2\}$
8 **return** τ_t

$\sigma_{\sigma_i}^\sim := \langle \sigma_1, \ldots, \sigma_{i-1}, \sigma_{i+1}, \ldots, \sigma_k \rangle$ with $\sigma = \langle \sigma_1, \ldots, \sigma_k \rangle$, $S_e^+ := S \cup \{e\}$, and $S_e^- := S \setminus \{e\}$.

consists of a set $\mathcal{P} \subseteq I$ that represents atoms in I for which we know that they have already been proven. The third position σ is a sequence of the atoms I such that there is a super-sequence σ' of σ, which induces an ordering $<_{\sigma'}$. Our local algorithm $\mathbb{PHC}$ stores interpretation parts always restricted to bag $\chi(t)$ and ensures that an interpretation can be extended to a model of program $\Pi_{\leq t}$. More precisely, it guarantees that interpretation I can be extended to a model $I' \supseteq I$ of $\Pi_{\leq t}$ and that the atoms in $I' \setminus I$ (and the atoms in $\mathcal{P} \subseteq I$) have already been *proven*, using some induced ordering $<_{\sigma'}$ where σ is a sub-sequence of σ'. In the end, an interpretation $\mathcal{I}(\boldsymbol{u})$ of a row $\boldsymbol{u}$ of the table $o(n)$ at the root n proves that there is a superset $I' \supseteq \mathcal{I}(\boldsymbol{u})$ that is an answer set of $\Pi = \Pi_{\leq n}$.

Listing 2 presents the algorithm $\mathbb{PHC}$. Intuitively, whenever an atom a is introduced (*int*), we decide whether we include a in the interpretation, determine bag atoms that can be proven in consequence of this decision, and update the sequence σ accordingly. To this end, we define for a given interpretation I and a sequence σ the set $\mathsf{proven}(I, \sigma, \Pi_t) := \bigcup_{r \in \Pi_t, a \in H_r} \{a \mid B_r^+ \subseteq I, I \cap B_r^- = \emptyset, I \cap (H_r \setminus \{a\}) = \emptyset, B_r^+ <_\sigma a\}$ where $B_r^+ <_\sigma a$ holds if $b <_\sigma a$ is true for every $b \in B_r^+$. Moreover, given a sequence $\sigma = \langle \sigma_1, \ldots, \sigma_k \rangle$ and a set A of atoms, we compute the potential sequences involving A. Therefore, we let $\mathsf{ords}(\sigma, A) := \{\sigma \mid A = \emptyset\} \cup \bigcup_{a \in A} \{\langle a, \sigma_1, \ldots, \sigma_k \rangle, \ldots, \langle \sigma_1, \ldots, \sigma_k, a \rangle\}$. When removing (*rem*) an atom a, we only keep those rows where a has been proven (contained in $\mathcal{P}$) and then restrict remaining rows to the bag (not containing a). In case the node is of type *join*, we combine two rows in two different child tables, intuitively, we are enforced to agree on the interpretations I and sequences σ. However, concerning individual proofs $\mathcal{P}$, it suffices that an atom is proven in one row.

Example 4. Recall program Π from Example 1. Figure 2 depicts a TD $\mathcal{T} = (T, \chi)$ of the primal graph G_1 of Π. Further, the figure illustrates a snippet of tables of the TTD (T, χ, τ), which we obtain when running $\mathtt{DP}_{\mathbb{PHC}}$ on program Π and TD $\mathcal{T}$ according to Listing 2. In the following, we briefly discuss some selected rows of those tables. Note that for simplicity and space reasons, we write τ_j instead of $\tau(t_j)$ and identify rows by their node and identifier i in the figure. For example, the row $\boldsymbol{u}_{13.3} = \langle I_{13.3}, \mathcal{P}_{13.3}, \sigma_{13.3} \rangle \in \tau_{13}$ refers to the third row of table τ_{13} for

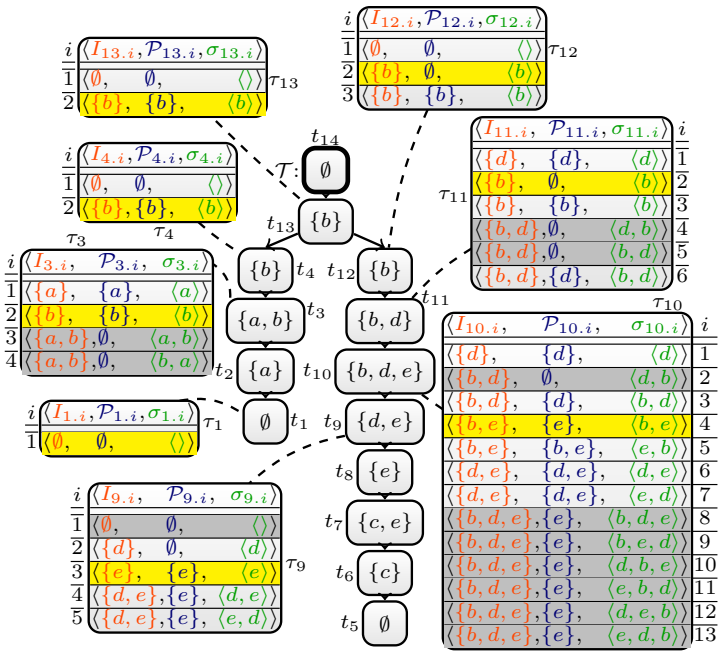

Fig. 2: Selected tables of τ obtained by $\mathtt{DP}_{\mathbb{PHC}}$ on TD $\mathcal{T}$.

node t_{13}. Node t_1 is of type *leaf*. Table τ_1 has only one row, which consists of the empty interpretation, empty set of proven atoms, and the empty sequence (Line 1). Node t_2 is of type *int* and introduces atom a. Executing Line 3 results in $\tau_2 = \{\langle\emptyset, \emptyset, \langle\rangle\rangle, \langle\{a\}, \emptyset, \langle a\rangle\rangle\}$. Node t_3 is of type *int* and introduces b. Then, bag-program at node t_3 is $\Pi_{t_3} = \{a \vee b \leftarrow\}$. By construction (Line 3) we ensure that interpretation $I_{3.i}$ is a model of Π_{t_3} for every row $\langle I_{3.i}, \mathcal{P}_{3.i}, \sigma_{3.i}\rangle$ in τ_3. Node t_4 is of type *rem*. Here, we restrict the rows such that they contain only atoms occurring in bag $\chi(t_4) = \{b\}$. To this end, Line 5 takes only rows $\boldsymbol{u}_{3.i}$ of table τ_3 where atoms in $I_{3.i}$ are also proven, i.e., contained in $\mathcal{P}_{3.i}$. In particular, every row in table τ_4 originates from at least one row in τ_3 that either proves $a \in \mathcal{P}_{3.i}$ or where $a \notin I_{3.i}$. Basic conditions of a TD ensure that once an atom is removed, it will not occur in any bag at an ancestor node. Hence, we also encountered all rules where atom a occurs. Nodes t_5, t_6, t_7, and t_8 are symmetric to nodes t_1, t_2, t_3, and t_4. Nodes t_9 and t_{10} again introduce atoms. Observe that $\mathcal{P}_{10.4} = \{e\}$ since $\sigma_{10.4}$ does not allow to prove b using atom e. However, $\mathcal{P}_{10.5} = \{b, e\}$ as the sequence $\sigma_{10.5}$ allows to prove b. In particular, in row $\boldsymbol{u}_{10.5}$ atom e is used to derive b. As a result, atom b can be proven, whereas ordering $\sigma_{10.4} = \langle b, e\rangle$ does not serve in proving b. We proceed similar for nodes t_{11} and t_{12}. At node t_{13} we join tables τ_4 and τ_{12} according to Line 7. Finally, $\tau_{14} \neq \emptyset$. Hence, Π has an answer set, namely, $\{b, e\}$, which we obtain by combining interpretation parts I of the yellow marked rows of Figure 2.

Next, we provide a notion to reconstruct answer sets from a computed TTD, which allows for computing for a given row its predecessor rows in the

corresponding child tables, c.f., [14]. Let Π be a program, $\mathcal{T} = (T, \chi, \tau)$ be an $\mathbb{A}$-TTD of G_Π, and t be a node of T where $\mathsf{children}(t, T) = \langle t_1, \ldots, t_\ell \rangle$. Given a sequence $\boldsymbol{s} = \langle s_1, \ldots, s_\ell \rangle$, we let $\langle\!\{\boldsymbol{s}\}\!\rangle := \langle \{s_1\}, \ldots, \{s_\ell\} \rangle$. For a given $\mathbb{A}$-row $\boldsymbol{u}$, we define the originating $\mathbb{A}$-rows of $\boldsymbol{u}$ in node t by $\mathbb{A}\text{-}\mathsf{origins}(t, \boldsymbol{u}) := \{\boldsymbol{s} \mid \boldsymbol{s} \in \tau(t_1) \times \cdots \times \tau(t_\ell), \boldsymbol{u} \in \mathbb{A}(t, \chi(t), \cdot, (\Pi_t, \cdot), \langle\!\{\boldsymbol{s}\}\!\rangle)\}$. We extend this to an $\mathbb{A}$-table ρ by $\mathbb{A}\text{-}\mathsf{origins}(t, \rho) := \bigcup_{\boldsymbol{u} \in \rho} \mathbb{A}\text{-}\mathsf{origins}(t, \boldsymbol{u})$.

Example 5. Consider program Π and $\mathbb{PHC}$-tabled tree decomposition (T, χ, τ) from Example 4. We focus on $\boldsymbol{u_{1.1}} = \langle \emptyset, \emptyset, \langle\rangle \rangle$ of table τ_1 of leaf t_1. The row $\boldsymbol{u_{1.1}}$ has no preceding row, since $\mathrm{type}(t_1) = \mathit{leaf}$. Hence, we have $\mathbb{PHC}\text{-}\mathsf{origins}(t_1, \boldsymbol{u_{1.1}}) = \{\langle\rangle\}$. The origins of row $\boldsymbol{u_{11.1}}$ of table τ_{11} are given by $\mathbb{PHC}\text{-}\mathsf{origins}(t_{11}, \boldsymbol{u_{11.1}})$, which correspond to the preceding rows in table τ_{10} that lead to row $\boldsymbol{u_{11.1}}$ of table τ_{11} when running algorithm $\mathbb{PHC}$, i.e., $\mathbb{PHC}\text{-}\mathsf{origins}(t_{11}, \boldsymbol{u_{11.1}}) = \{\langle \boldsymbol{u_{10.1}} \rangle, \langle \boldsymbol{u_{10.6}} \rangle, \langle \boldsymbol{u_{10.7}} \rangle\}$. Origins of row $\boldsymbol{u}_{12.2}$ are given by $\mathbb{PHC}\text{-}\mathsf{origins}(t_{12}, \boldsymbol{u_{12.2}}) = \{\langle \boldsymbol{u_{11.2}} \rangle, \langle \boldsymbol{u_{11.6}} \rangle\}$. Note that $\boldsymbol{u_{11.4}}$ and $\boldsymbol{u_{11.5}}$ are not among those origins, since d is not proven. Observe that $\mathbb{PHC}\text{-}\mathsf{origins}(t_j, \boldsymbol{u}) = \emptyset$ for any row $\boldsymbol{u} \notin \tau_j$. For node t_{13} of type *join* and row $\boldsymbol{u_{13.2}}$, we obtain $\mathbb{PHC}\text{-}\mathsf{origins}(t_{13}, \boldsymbol{u_{13.2}}) = \{\langle \boldsymbol{u_{4.2}}, \boldsymbol{u_{12.2}} \rangle, \langle \boldsymbol{u_{4.2}}, \boldsymbol{u_{12.3}} \rangle\}$.

Theorem 1 ($\star$). *The algorithm $\mathtt{DP}_{\mathbb{PHC}}$ is correct. In other words, given an hcf program Π and a TTD $\mathcal{T} = (T, \chi, \cdot)$ of G_Π where $T = (N, \cdot, n)$ with root n. Then, algorithm $\mathtt{DP}_{\mathbb{PHC}}((\Pi, \cdot), \mathcal{T})$ returns the $\mathbb{PHC}$-TTD (T, χ, τ) such that Π has an answer set if and only if $\langle \emptyset, \emptyset, \langle\rangle \rangle \in \tau(n)$. Further, we can construct all the answer sets of Π from transitively following the origins of $\tau(n)$.*

Proof (Idea). For soundness, we state an invariant and establish that this invariant holds for every node $t \in N$. For each row $\boldsymbol{u} = \langle I, \mathcal{P}, \sigma \rangle \in \tau(t)$, we have $I \subseteq \chi(t), \mathcal{P} \subseteq I$, and σ is a sequence over atoms in I. Intuitively, we ensure that $I \models \Pi_{\leq t}$ and that exactly the atoms in $\mathrm{at}_{<t}$ and $\mathcal{P}$ can be proven using a super-sequence σ' of σ. By construction, we guarantee to decide consistency if row $\langle \emptyset, \emptyset, \langle\rangle \rangle \in \tau(n)$. Further, we can reconstruct answer sets, by following $\mathbb{PHC}\text{-}\mathsf{origins}$ of this row to the leaves. For completeness, we show that we obtain all rows required to output all answer sets of Π.

Theorem 2. *Given an hcf program Π and a TD $\mathcal{T} = (T, \chi)$ of G_Π of width k with g nodes. Algorithm $\mathtt{DP}_{\mathbb{PHC}}$ runs in time $\mathcal{O}(3^k \cdot k! \cdot g)$.*

Proof (Proof (Sketch).). Let $d = k + 1$ be maximum bag size of the TD $\mathcal{T}$. The table $\tau(t)$ has at most $3^d \cdot d!$ rows, since for a row $\langle I, \mathcal{P}, \sigma \rangle$ we have $d!$ many sequences σ, and by construction of algorithm $\mathbb{PHC}$, an atom can be either in I, both in I and $\mathcal{P}$, or neither in I nor in $\mathcal{P}$. In total, with the help of efficient data structures, e.g., for nodes t with $\mathrm{type}(t) = \mathit{join}$, one can establish a runtime bound of $\mathcal{O}(3^d \cdot d!)$. Then, we apply this to every node t of the TD, which resulting in running time $\mathcal{O}(3^d \cdot d! \cdot g) \subseteq \mathcal{O}(3^k \cdot k! \cdot g)$.

[7] ν contains rows obtained by recursively following origins of $\tau(n)$.

[8] Later we use (among others) $\mathtt{PCNT}_{\mathbb{PHC}}$ where $\mathbb{A} = \mathbb{PHC}$.

A natural question is whether we can significantly improve this algorithm for fixed k. To this end, we take the *exponential time hypothesis (ETH)* into account, which states that there is some real $s > 0$ such that we cannot decide satisfiability of a given 3-CNF formula F in time $2^{s \cdot |F|} \cdot \|F\|^{\mathcal{O}(1)}$.

Proposition 2 ($\star$). *Unless ETH fails, we cannot decide consistency of a given hcf program Π in time $2^{o(k)} \cdot \|\Pi\|^{o(k)}$ where k is the treewidth of primal graph G_Π.*

Dynamic Programming for #PAs(hcf)

In this section, we present our dynamic programming algorithm[8] $\texttt{PCNT}_{\mathbb{A}}$, which allows for solving the projected answer set counting problem #PAs(hcf). $\texttt{PCNT}_{\mathbb{A}}$ is based on an approach of projected counting for Boolean formulas [14] where TDs are traversed multiple times. We show that ideas from that approach can be fruitfully extended to answer set programming. First, we construct the primal graph G_Π of the input program Π and compute a TD of Π. Then, we traverse the TD a first time by running $\texttt{DP}_{\mathbb{A}}$ (Step 2a), which outputs a TTD $\mathcal{T}_{\text{cons}} = (T, \chi, \tau)$. Afterwards, we traverse $\mathcal{T}_{\text{cons}}$ in pre-order and remove all rows from the tables that cannot be extended to an answer set (*"Purge non-solutions"*). In other words, we keep only rows $\boldsymbol{u}$ of table $\tau(t)$ at node t, if $\boldsymbol{u}$ is involved in those rows that are used to construct an answer set of Π, and let the resulting TTD be $\mathcal{T}_{\text{purged}} = (T, \chi, \nu)$[7]. We refer to ν as *purged table mapping*. In Step 2b ($\texttt{DP}_{\mathbb{PROJ}}$), we traverse $\mathcal{T}_{\text{purged}}$ to count interpretations with respect to the projection atoms and obtain $\mathcal{T}_{\text{proj}} = (T, \chi, \pi)$. From the table $\pi(n)$ at the root n of T, we can then read the projected answer set count of the input instance. In the following, we only describe the local algorithm ($\mathbb{PROJ}$), since the traversal in $\texttt{DP}_{\mathbb{PROJ}}$ is the same as before. For $\mathbb{PROJ}$, a row at a node t is a pair $\langle \rho, c \rangle \in \pi(t)$, where $\rho \subseteq \nu(t)$ is an $\mathbb{A}$-table and c is a non-negative integer. In fact, integer c stores the number of *intersecting (overlapping) answer sets (*ipasc*)* restricted to $P \cap \text{at}_{\leq t}$, to which *all* the rows in ρ can be extended. However, we ultimately aim for the *projected answer set count (*pasc*)*, whose computation requires additional definitions, which we hence widen from recent work [14].

In the remainder, we assume (Π, P) to be an instance of #PAs, (T, χ, τ) to be an $\mathbb{A}$-TTD of G_Π and the mappings τ, ν, and π as used above. Further, let t be a node of T with $\mathsf{children}(t, T) = \langle t_1, \ldots, t_\ell \rangle$ and let $\rho \subseteq \nu(t)$. The relation $=_{\text{P}} \subseteq \rho \times \rho$ considers equivalent rows with respect to the projection of its interpretations by $=_{\text{P}} := \{(\boldsymbol{u}, \boldsymbol{v}) \mid \boldsymbol{u}, \boldsymbol{v} \in \rho, \mathcal{I}(\boldsymbol{u}) \cap P = \mathcal{I}(\boldsymbol{v}) \cap P\}$. Let $\mathsf{buckets}_P(\rho)$ be the set of equivalence classes induced by $=_{\text{P}}$ on ρ, i.e., $\mathsf{buckets}_P(\rho) := (\rho / =_{\text{P}}) = \{[\boldsymbol{u}]_P \mid \boldsymbol{u} \in \rho\}$, where $[\boldsymbol{u}]_P = \{\boldsymbol{v} \mid \boldsymbol{v} =_{\text{P}} \boldsymbol{u}, \boldsymbol{v} \in \rho\}$ [29]. Further, $\mathsf{sub\text{-}buckets}_P(\rho) := \{S \mid \emptyset \subsetneq S \subseteq B, B \in \mathsf{buckets}_P(\rho)\}$.

Example 6. Consider program Π, set P of projection atoms, TTD (T, χ, τ), and table τ_{10} from Example 6 and Figure 2. Note that during purging rows $\boldsymbol{u_{10.2}}$ and $\boldsymbol{u_{10.8}}, \ldots, \boldsymbol{u_{10.13}}$ are removed (highlighted gray), since they are not involved in any answer set, resulting in table ν_{10}. Then, $\boldsymbol{u_{10.4}} =_P \boldsymbol{u_{10.5}}$ and $\boldsymbol{u_{10.6}} =_P \boldsymbol{u_{10.7}}$. The set $\nu_{10} / =_{\text{P}}$ of equivalence classes of ν_{10} is $\mathsf{buckets}_P(\nu_{10}) = \{\{\boldsymbol{u_{10.1}}\}, \{\boldsymbol{u_{10.3}}\}, \{\boldsymbol{u_{10.4}}, \boldsymbol{u_{10.5}}\}, \{\boldsymbol{u_{10.6}}, \boldsymbol{u_{10.7}}\}\}$.

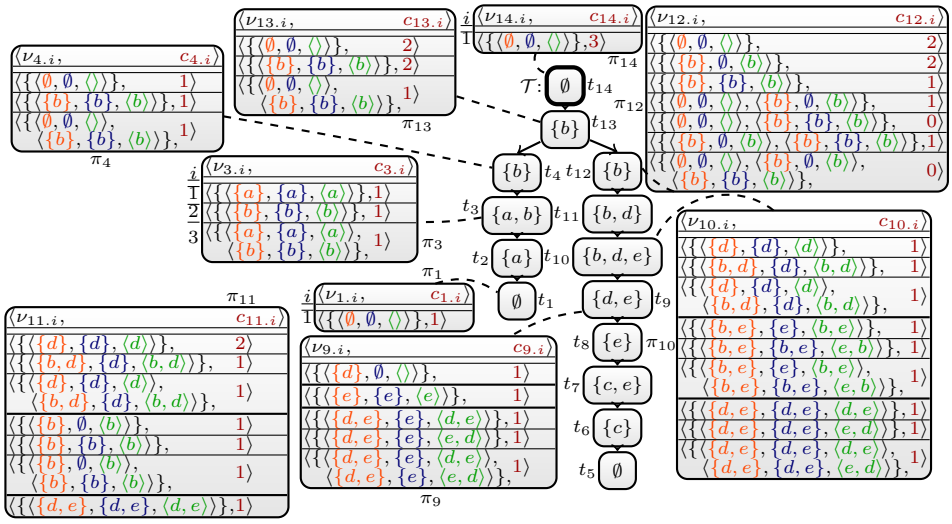

Fig. 3: Selected tables of π obtained by $\mathtt{DP}_{\mathbb{PROJ}}$ on TD $\mathcal{T}$ and purged table mapping ν (obtained by purging on τ, c.f, Figure 2).

Later, we require to reuse already computed projected answer set counts for tables of children of a given node t. Therefore, we define the *stored* ipasc of a table $\rho \subseteq \nu(t)$ in table $\pi(t)$ by $\mathsf{s\text{-}ipasc}(\pi(t), \rho) := \sum_{\langle \rho, c\rangle \in \pi(t)} c$. We extend this to a sequence $s = \langle \pi(t_1), \ldots, \pi(t_\ell)\rangle$ of tables of length ℓ and a set $O = \{\langle \rho_1, \ldots, \rho_\ell\rangle, \langle \rho'_1, \ldots, \rho'_\ell\rangle, \ldots\}$ of sequences of ℓ tables by $\mathsf{s\text{-}ipasc}(s, O) = \prod_{i\in\{1,\ldots,\ell\}} \mathsf{s\text{-}ipasc}(s_{(i)}, O_{(i)})$. This allows to select the i-th position of the sequence together with sets of the i-th positions from the set of sequences.

Intuitively, when we are at a node t in algorithm $\mathtt{DP}_{\mathbb{PROJ}}$ we have already computed $\pi(t')$ of $\mathcal{T}_{\text{proj}}$ for every node t' below t. Then, we compute the projected answer set count of $\rho \subseteq \nu(t)$. Therefore, we apply the inclusion-exclusion principle to the stored projected answer set count of origins. We define $\mathsf{pasc}(t, \rho, \langle \pi(t_1), \ldots\rangle) := \sum_{\emptyset \subsetneq O \subseteq \mathbb{A}\text{-}\mathsf{origins}(t,\rho)} (-1)^{(|O|-1)} \cdot \mathsf{s\text{-}ipasc}(\langle \pi(t_1), \ldots\rangle, O)$. Vaguely speaking, pasc determines the $\mathbb{A}$-origins of table ρ, goes over all subsets of these origins and looks up the stored counts (s-ipasc) in the $\mathbb{PROJ}$-tables of the children t_i of t in order to count the answer sets restricted to $P \cap \mathrm{at}_{\leq t}$, to which rows in ρ can be extended.

Example 7. Consider again program Π and TD $\mathcal{T}$ from Example 1 and Figure 2. First, we compute the projected count $\mathsf{pasc}(t_4, \{\boldsymbol{u_{4.1}}\}, \langle \pi(t_3)\rangle)$ for row $\boldsymbol{u_{4.1}}$ of table $\nu(t_4)$, where $\pi(t_3) := \{\langle \{\boldsymbol{u_{3.1}}\}, 1\rangle, \langle \{\boldsymbol{u_{3.2}}\}, 1\rangle, \langle \{\boldsymbol{u_{3.1}}, \boldsymbol{u_{3.2}}\}, 1\rangle\}$ with $\boldsymbol{u_{3.1}} = \langle \emptyset, \emptyset, \langle\rangle\rangle$ and $\boldsymbol{u_{3.2}} = \langle \{a\}, \emptyset, \langle a\rangle\rangle$. Note that t_5 has only the child t_4 and therefore the product in s-ipasc consists of only one factor. Since $\mathbb{PHC}\text{-}\mathsf{origins}(t_4, \boldsymbol{u_{4.1}}) = \{\langle \boldsymbol{u_{3.1}}\rangle\}$, only the value of s-ipasc for set $\{\langle \boldsymbol{u_{3.1}}\rangle\}$ is non-zero. Hence, we obtain $\mathsf{pasc}(t_4, \{\boldsymbol{u_{4.1}}\}, \langle \pi(t_3)\rangle) = 1$. Next, we compute $\mathsf{pasc}(t_4, \{\boldsymbol{u_{4.1}}, \boldsymbol{u_{4.2}}\}, \langle \pi(t_3)\rangle)$. Observe that $\mathbb{PHC}\text{-}\mathsf{origins}(t_4, \{\boldsymbol{u_{4.1}}, \boldsymbol{u_{4.2}}\}) = \{\langle \boldsymbol{u_{3.1}}\rangle, \langle \boldsymbol{u_{3.2}}\rangle\}$. We sum up the values of s-ipasc for sets $\{\boldsymbol{u_{4.1}}\}$ and $\{\boldsymbol{u_{4.2}}\}$ and subtract the one for set $\{\boldsymbol{u_{4.1}}, \boldsymbol{u_{4.2}}\}$. Hence, we obtain $\mathsf{pasc}(t_4, \{\boldsymbol{u_{4.1}}, \boldsymbol{u_{4.2}}\}, \langle \pi(t_3)\rangle) = 1 + 1 - 1 = 1$.

Listing 3: Local algorithm $\mathbb{PROJ}(t, \cdot, \nu_t, (\cdot, P), \langle \pi_1, \ldots \rangle)$ for projected counting.

In: Node t, purged table mapping ν_t, set P of projection atoms, $\langle \pi_1, \ldots \rangle$ is the sequence of $\mathbb{PROJ}$-tables of children of t.
Out: $\mathbb{PROJ}$-table π_t of pairs $\langle \rho, c \rangle$, where $\rho \subseteq \nu_t$, and $c \in \mathbb{N}$.
1 $\pi_t \leftarrow \big\{ \langle \rho, \mathsf{ipasc}(t, \rho, \langle \pi_1, \ldots \rangle) \rangle \big| \, \rho \in \mathsf{sub\text{-}buckets}_P(\nu_t) \big\}$
2 **return** π_t

Next, we provide a definition to obtain ipasc, which can be determined at a node t for given table $\rho \subseteq \nu(t)$ by computing the pasc for children t_i of t using stored ipasc values from tables $\pi(t_i)$, subtracting and adding ipasc values for subsets $\emptyset \subsetneq \varphi \subsetneq \rho$ accordingly. Formally, $\mathsf{ipasc}(t, \rho, s) := 1$ if $\mathrm{type}(t) = \mathit{leaf}$ and otherwise $\mathsf{ipasc}(t, \rho, s) := \big| \, \mathsf{pasc}(t, \rho, s) \; + \quad \sum_{\emptyset \subsetneq \varphi \subsetneq \rho} (-1)^{|\varphi|} \cdot \mathsf{ipasc}(t, \varphi, s) \big|$ where $s = \langle \pi(t_1), \ldots \rangle$. In other words, for (empty) nodes of type *leaf* the ipasc is one. Otherwise, we compute the "non-overlapping" count of given table $\rho \subseteq \nu(t)$ with respect to $P \cap \mathrm{at}_{\leq t}$, by exploiting the inclusion-exclusion principle on $\mathbb{A}$-origins of ρ such that we count every projected answer set *only once*. Then we subtract and add ipasc ("all-overlapping" counts) for strict subsets φ of ρ, accordingly.

Finally, Listing 3 presents the local algorithm $\mathbb{PROJ}$, which stores $\pi(t)$ consisting of every sub-bucket of the given table $\nu(t)$ together with its ipasc. For the only row at root node n, ipasc and pasc values coincide and solve #PAs.

Example 8. Recall instance (Π, P), TD $\mathcal{T}$, and tables $\tau_1, \ldots, \tau_{14}$ from Examples 6, 4, and Figure 2. Figure 3 depicts selected tables of $\pi_1, \ldots, \pi_{14}$ obtained after running $\mathtt{DP}_{\mathbb{PROJ}}$ for counting projected answer sets. We assume that row i in table π_t corresponds to $\boldsymbol{v_{t.i}} = \langle \rho_{t.i}, c_{t.i} \rangle$ where $\rho_{t.i} \subseteq \nu(t)$. Recall that for some nodes t, there are rows among different $\mathbb{PHC}$-tables that are removed (highlighted gray in Figure 2) during purging. Purging avoids correcting counters (backtracking) whenever a row has no "succeeding row".

Next, we discuss selected rows obtained by $\mathtt{DP}_{\mathbb{PROJ}}((\Pi, P), (T, \chi, \nu))$. Tables π_1, …, π_{14} are shown in Figure 3. Since $\mathrm{type}(t_1) = \mathit{leaf}$, we have $\pi_1 = \langle \{ \langle \emptyset, \emptyset, \langle \rangle \rangle \}, 1 \rangle$. Intuitively, at t_1 the row $\langle \emptyset, \emptyset, \langle \rangle \rangle$ belongs to 1 bucket. Node t_2 introduces atom a, which results in table $\pi_2 := \big\{ \langle \{ \boldsymbol{u_{2.1}} \}, 1 \rangle, \langle \{ \boldsymbol{u_{2.2}} \}, 1 \rangle, \langle \{ \boldsymbol{u_{2.1}}, \boldsymbol{u_{2.2}} \}, 1 \rangle \big\}$, where $\boldsymbol{u_{2.1}} = \langle \emptyset, \emptyset, \langle \rangle \rangle$ and $\boldsymbol{u_{2.2}} = \langle \{a\}, \emptyset, \langle a \rangle \rangle$ (derived similarly to table π_4 as in Example 7). Node t_{10} introduces projected atom e, and node t_{11} removes e. For row $\boldsymbol{v_{11.1}}$ we compute the count $\mathsf{ipasc}(t_{11}, \{ \boldsymbol{u_{11.1}} \}, \langle \pi_{10} \rangle)$ by means of pasc. Therefore, take for φ the singleton set $\{ \boldsymbol{u_{11.1}} \}$. We simply have $\mathsf{ipasc}(t_{11}, \{ \boldsymbol{u_{11.1}} \}, \langle \pi_{10} \rangle) = \mathsf{pasc}(t_{11}, \{ \boldsymbol{u_{11.1}} \}, \langle \pi_{10} \rangle)$. To compute $\mathsf{pasc}(t_{11}, \{ \boldsymbol{u_{11.1}} \}, \langle \pi_{10} \rangle)$, we take for O the sets $\{ \boldsymbol{u_{10.1}} \}$, $\{ \boldsymbol{u_{10.6}} \}$, $\{ \boldsymbol{u_{10.7}} \}$, and $\{ \boldsymbol{u_{10.6}}, \boldsymbol{u_{10.7}} \}$ into account, since all other non-empty subsets of origins of $\boldsymbol{u_{11.1}}$ in ν_{10} do not occur in π_{10}. Then, we take the sum over the values $\mathsf{s\text{-}ipasc}(\langle \pi_{10} \rangle, \{ \boldsymbol{u_{10.1}} \}) = 1$, $\mathsf{s\text{-}ipasc}(\langle \pi_{10} \rangle, \{ \boldsymbol{u_{10.6}} \}) = 1$, $\mathsf{s\text{-}ipasc}(\langle \pi_{10} \rangle, \{ \boldsymbol{u_{10.7}} \}) = 1$ and subtract $\mathsf{s\text{-}ipasc}(\langle \pi_{10} \rangle, \{ \boldsymbol{u_{10.6}}, \boldsymbol{u_{10.7}} \}) = 1$. This results in $\mathsf{pasc}(t_{11}, \{ \boldsymbol{u_{11.1}} \}, \langle \pi_{10} \rangle) = c_{10.1} + c_{10.7} + c_{10.8} - c_{10.9} = 2$. We proceed similarly for row $v_{11.2}$, resulting in $c_{11.2} = 1$. Then for row $v_{11.3}$, $\mathsf{ipasc}(t_{11}, \{ \boldsymbol{u_{11.1}}, \boldsymbol{u_{11.6}} \}, \langle \pi_{10} \rangle) = |\, \mathsf{pasc}(t_{11}, \{ \boldsymbol{u_{11.1}}, \boldsymbol{u_{11.6}} \}, \langle \pi_{10} \rangle) - \mathsf{ipasc}(t_{11}, \{ \boldsymbol{u_{11.1}} \}, \langle \pi_{10} \rangle) - \mathsf{ipasc}(t_{11}, \{ \boldsymbol{u_{11.6}} \}, \langle \pi_{10} \rangle)| = |2 - c_{11.1} - c_{11.2}| = |2 - 2 - 1| = |-1| = 1 = c_{11.3}$. Hence, $c_{11.3} = 1$ represents the number of projected answer

sets, both rows $\boldsymbol{u_{11.1}}$ and $\boldsymbol{u_{11.6}}$ have in common. We then use it for table t_{12}. Node t_{12} removes projection atom d. For node t_{13} where $\text{type}(t_{13}) = \textit{join}$ one multiplies stored s-ipasc values for $\mathbb{A}$-rows in the two children of t_{13} accordingly. The projected answer set count of Π results in $\text{s-ipasc}(\langle \pi_{14} \rangle, \boldsymbol{u_{14.1}}) = 3$.

Runtime Analysis and Correctness

Next, we present asymptotic upper bounds on the runtime of our Algorithm $\texttt{DP}_{\mathbb{PROJ}}$. We assume $\gamma(n)$ to be the number of operations required to multiply two n-bit integers, which can be achieved in time $n \cdot \log n \cdot \log\log n$ [23, 19].

Theorem 3. *Given a #PAs(hcf) instance (Π, P) and a TTD $\mathcal{T}_{purged} = (T, \chi, \nu)$ of G_Π of width k with g nodes. Then, $\texttt{DP}_{\mathbb{PROJ}}$ runs in time $\mathcal{O}(2^{4m} \cdot g \cdot \gamma(\|\Pi\|))$ where $m := \max(\{\nu(t) \mid t \in N\})$.*

Proof. For each node t of T, we consider the table $\nu(t)$ of $\mathcal{T}_{\text{purged}}$. Let TDD (T, χ, π) be the output of $\texttt{DP}_{\mathbb{PROJ}}$. In worst case, we store in $\pi(t)$ each subset $\rho \subseteq \nu(t)$ together with exactly one counter. Hence, we have at most 2^m many rows in ρ. In order to compute ipasc for ρ, we consider every subset $\varphi \subseteq \rho$ and compute pasc. Since $|\rho| \leq m$, we have at most 2^m many subsets φ of ρ. Finally, for computing pasc, we consider in the worst case each subset of the origins of φ for each child table, which are at most $2^m \cdot 2^m$ because of nodes t with $\text{type}(t) = \textit{join}$. In total, we obtain a runtime bound of $\mathcal{O}(2^m \cdot 2^m \cdot 2^m \cdot \gamma(\|\Pi\|)) \subseteq \mathcal{O}(2^{4m} \cdot \gamma(\|\Pi\|))$ due to multiplication of two n-bit integers for nodes t with $\text{type}(t) = \textit{join}$ at costs $\gamma(n)$. Then, we apply this to every node of T resulting in runtime $\mathcal{O}(2^{4m} \cdot g \cdot \gamma(\|\Pi\|))$.

From the theorem we immediately obtain that $\texttt{PCNT}_{\mathbb{PHC}}$ runs in time $\mathcal{O}(2^{3^{k+1.27} \cdot k!} \cdot \|\Pi\| \cdot \gamma(\|\Pi\|))$. Then, the next result establishes lower bounds.

Theorem 4. *Unless ETH fails, #PAs(hcf) cannot be solved in time $2^{2^{o(k)}} \cdot \|\Pi\|^{o(k)}$ for a given instance (Π, P), where k is the treewidth of primal graph G_Π.*

Proof. Assume for proof by contradiction that there is such an algorithm. We show that this contradicts a very recent result [24, 14], which states that one cannot decide the validity of a QBF $\forall V_1 . \exists V_2 . E$ in time $2^{2^{o(k)}} \cdot \|E\|^{o(k)}$, where E is in CNF. Let $(\forall V_1 . \exists V_2 . E, k)$ be an instance of $\forall\exists$-SAT parameterized by the treewidth k. Then, we define an *fpt-reduction* to an instance $((\Pi, P), 2k)$ of the decision version #PAs(hcf)-exactly-$2^{|V_1|}$ when parameterized by treewidth of G_Π such that $P = V_1$, the number of solutions is exactly $2^{|V_1|}$, and Π is as follows. For each $v \in V_1 \cup V_2$, program Π contains rule $v \vee nv \leftarrow$. Each clause $x_1 \vee \ldots \vee x_i \vee \neg x_{i+1} \vee \ldots \vee \neg x_j$ results in one additional rule $\leftarrow \neg x_1, \ldots, \neg x_i, x_{i+1}, \ldots, x_j$. It is easy to see that the reduction is correct and therefore instance $((\Pi, P), 2k)$ is a yes instance of #PAs(hcf)-exactly-$2^{|V_1|}$ if and only if $(\forall V_1 . \exists V_2 . E, k)$ is a yes instance of problem $\forall\exists$-SAT. In fact, since the stated reduction *at most doubles* the treewidth of Π, the result of $\forall\exists$-SAT carries over.

Proposition 3 ($\star$). *Algorithm $\texttt{PCNT}_{\mathbb{PHC}}$ is correct and gives for any instance of #PAs(hcf) its projected answer set count.*

Conclusions

We introduced a novel algorithm to count projected answer sets (#PAs) of hcf programs, which employs dynamic programming and exploits small treewidth of the primal graph of the input program. Moreover, we presented a formal framework to solve projected answer set counting, which lifts results from the setting of Boolean formulas to ASP and also applies to disjunctive programs. Finally, we established complexity lower bounds using ETH, showing that #PAs(hcf) cannot be solved in time significantly better than double exponential in the treewidth and still stay polynomial. Our results extend previous work to ASP and we believe that it is also applicable to other hard combinatorial problems, such as argumentation [11].

References

1. Aziz, R.A.: Answer Set Programming: Founded Bounds and Model Counting. Ph.D. thesis, Department of Computing and Information Systems, The University of Melbourne (2015)
2. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Ann. Math. Artif. Intell. **12**(1), 53–87 (1994)
3. Bidoít, N., Froidevaux, C.: Negation by default and unstratifiable logic programs. Theoretical Computer Science **78**(1), 85–112 (1991)
4. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. **25**(6), 1305–1317 (1996)
5. Bodlaender, H.L., Kloks, T.: Efficient and constructive algorithms for the pathwidth and treewidth of graphs. J. Algorithms **21**(2), 358–402 (1996)
6. Bondy, J.A., Murty, U.S.R.: Graph theory, Graduate Texts in Mathematics, vol. 244. Springer Verlag, New York, USA (2008)
7. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM **54**(12), 92–103 (2011)
8. Cygan, M., Fomin, F.V., Kowalik, Ł., Lokshtanov, D., Dániel Marx, M.P., Pilipczuk, M., Saurabh, S.: Parameterized Algorithms. Springer Verlag (2015)
9. Doe, J., Bloggs, J.: Disjunctive #PAs and treewidth. (2018), extended abstract. To appear.
10. Durand, A., Hermann, M., Kolaitis, P.G.: Subtractive reductions and complete problems for counting complexity classes. Theoretical Computer Science **340**(3), 496–513 (2005)
11. Dvořák, W., Ordyniak, S., Szeider, S.: Augmenting tractable fragments of abstract argumentation. Artif. Intell. **186**, 157–173 (2012)
12. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: Propositional case. Ann. Math. Artif. Intell. **15**(3–4), 289–323 (1995)
13. Fichte, J.K., Hecher, M., Morak, M., Woltran, S.: Answer set solving with bounded treewidth revisited. In: Balduccini, M., Janhunen, T. (eds.) Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17). Lecture Notes in Computer Science, vol. 10377, pp. 132–145. Springer Verlag, Espoo, Finland (2017)
14. Fichte, J.K., Hecher, M., Morak, M., Woltran, S.: Exploiting treewidth for projected model counting and its limits. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) Proceedings on the 21th International Conference on Theory and Applications of

Satisfiability Testing (SAT'18). Lecture Notes in Computer Science, vol. 10929, pp. 165–184. Springer Verlag, Oxford, UK (2018)
15. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan & Claypool (2012)
16. Gebser, M., Kaufmann, B., Schaub, T.: Solution enumeration for projected boolean search problems. In: van Hoeve, W.J., Hooker, J.N. (eds.) Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09). Lecture Notes in Computer Science, vol. 5547, pp. 71–86. Springer Verlag, Berlin, Germany (2009)
17. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. **9**(3/4), 365–386 (1991)
18. Graham, R.L., Grötschel, M., Lovász, L.: Handbook of combinatorics, vol. I. Elsevier Science Publishers, North-Holland (1995)
19. Harvey, D., van der Hoeven, J., Lecerf, G.: Even faster integer multiplication. J. Complexity **36**, 1–30 (2016)
20. Jakl, M., Pichler, R., Woltran, S.: Answer-set programming with bounded treewidth. In: Proceedings of the 21st International Joint Conference on Artificial intelligence (IJCAI'09). vol. 2, pp. 816–822 (2009)
21. Janhunen, T., Niemelä, I.: The answer set programming paradigm. AI Magazine **37**(3), 13–24 (2016), `http://www.aaai.org/ojs/index.php/aimagazine/article/view/2671`
22. Kloks, T.: Treewidth. Computations and Approximations, Lecture Notes in Computer Science, vol. 842. Springer Verlag (1994)
23. Knuth, D.E.: How fast can we multiply? In: The Art of Computer Programming, Seminumerical Algorithms, vol. 2, chap. 4.3.3, pp. 294–318. Addison-Wesley, 3 edn. (1998)
24. Lampis, M., Mitsou, V.: Treewidth with a quantifier alternation revisited. In: Lokshtanov, D., Nishimura, N. (eds.) Proceedings of the 12th International Symposium on Parameterized and Exact Computation (IPEC'17). Dagstuhl Publishing (2017)
25. Lin, F., Zhao, J.: On tight logic programs and yet another translation from normal logic programs to propositional logic. In: Gottlob, G., Walsh, T. (eds.) Proceedings of the 18th International Joint Conference on Artificial intelligence (IJCAI'03). pp. 853–858. Morgan Kaufmann, Acapulco, Mexico (2003)
26. Marek, W., Truszczyński, M.: Autoepistemic logic. J. of the ACM **38**(3), 588–619 (1991)
27. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1994)
28. Pichler, R., Rümmele, S., Szeider, S., Woltran, S.: Tractable answer-set programming with weight constraints: bounded treewidth is not enough. Theory Pract. Log. Program. **14**(2) (2014)
29. Wilder, R.L.: Introduction to the Foundations of Mathematics. John Wiley & Sons, 2nd edn. (1965)