Plausibility Reasoning via Projected Answer Set Counting – A Hybrid Approach

Johannes K. Fichte\(^1\), Markus Hecher\(^1\), Mohamed A. Nadeem\(^2\)

\(^1\) TU Wien, Vienna, Austria
\(^2\) TU Dresden, Dresden, Germany
jfichte@dbai.tuwien.ac.at, hecher@dbai.tuwien.ac.at, mohamed.nadeem@mailbox.tu-dresden.de

Abstract

Answer set programming is a form of declarative programming widely used to solve difficult search problems. Probabilistic applications however require to go beyond simple search for one solution and need counting. One such application is plausibility reasoning, which provides more fine-grained reasoning mode between simple brave and cautious reasoning. When modeling with ASP, we often introduce auxiliary atoms in the program. If these atoms are functionally independent of the atoms of interest, we need to hide the auxiliary atoms and project the count to the atoms of interest resulting in the problem projected answer set counting. In practice, counting becomes quickly infeasible with standard systems such as clasp. In this paper, we present a novel hybrid approach for plausibility reasoning under projections, thereby relying on projected answer set counting as basis. Our approach combines existing systems with fast dynamic programming, which in our experiments shows advantages over existing ASP systems.

1 Introduction

Answer set programming (ASP) [Gelfond and Leone, 2002; Lifschitz, 2002; Marek and Truszczynski, 1999; Niemelä, 1999] is a declarative framework that is well-known in the area of knowledge representation and non-monotonic reasoning [Baral and Chitta, 2003; van Harmelen et al., 2008]. It is widely used to solve difficult search problems while allowing compact modeling [Gebser et al., 2012]. Traditionally, when considering reasoning, the ASP community focuses on the modes cautious and brave, where brave and cautious reasoning ask whether an atom is contained in at least one answer set or all answer sets, respectively. This decision-based direction (yes/no-answers) to symbolic reasoning is indirectly motivated by convenient theoretical results and practical solving (existing systems).

However, these simple reasoning modes are insufficient if we are interested in more fine-grained reasoning, for example, the probability of an atom occurring in an answer set. More generally, we say plausibility reasoning asks whether a set \( Q \) of literals matches at least \( p \cdot 100\% \) of the answer sets for \( p \in [0,1] \). Figure 1 illustrates this more fine-grained reasoning mode between brave and cautious reasoning. It asks whether a set \( Q \) of literals matches \( \geq p \cdot 100\% \) of the answer sets for a program \( \Pi \).

![Figure 1: Plausibility reasoning](image)

Our main contributions are as follows.
1. We formally introduce the fine-grained reasoning mode plausibility reasoning under projection to ASP.
2. We combine existing systems with fast dynamic programming into a hybrid solver for plausibility reasoning.
3. We provide an implementation and initial empirical evaluation. In our experiments, we see feasibility and advantages of our approach over existing ASP systems.
2 Preliminaries

Answer Set Programming (ASP). We follow standard definitions of propositional ASP [Marek and Truszczynski, 1999; Niemelä, 1999]. Let \( t, m, n \) be non-negative integers such that \( \ell \leq m \leq n, a_1, \ldots, a_n \) be distinct atoms. We refer by \( \text{literal} \) to an atom or the negation thereof. A program \( \Pi \) is a finite set of rules of the form \( a_1 \lor \cdots \lor a_m \rightarrow \neg a_{m+1}, \ldots, \neg a_n \). For a rule \( r \), we let \( H_r := \{a_1, \ldots, a_k\}, B_r^+ := \{a_{k+1}, \ldots, a_m\}, \) and \( B_r^- := \{a_{m+1}, \ldots, a_n\} \). We denote the sets of atoms occurring in a rule \( r \) or in a program \( \Pi \) by \text{at}(r) \) and \text{at}(\Pi) := \bigcup_{r \in \Pi} \text{at}(r) \). Let \( \Pi \) be a program and \( Q \) be a set of literals (query). Then, \( \Pi \sqcup Q := \Pi \cup \{a \to \neg a, \neg b \to a \mid a \in Q \} \backslash \text{at}(\Pi), \neg b \in Q \} \) is the program \( \Pi \) under \( Q \). An interpretation \( I \) is a set of \( \{\neg a \mid a \in \Pi \} \) and say that \( Q \) matches \( I \), if \( Q \cap \text{at}(\Pi) = \Pi \) and \( Q \cap \text{at}(\Pi) = \emptyset \). \( I \) satisfies a rule \( r \) if \( H_r \cup B_r^- \cap \text{at}(r) = \emptyset \) or \( B_r^+ \cap \text{at}(r) = \emptyset \). \( I \) is a model of \( \Pi \) if it satisfies all rules of \( \Pi \). The Gelfond-Lifschitz (GL) reduct of \( \Pi \) under \( I \) is the program \( \Pi_I \) obtained from \( \Pi \) by first removing all rules \( r \) with \( B_r^+ \cap \text{at}(r) = \emptyset \) and then removing all \( \neg z \) where \( z \in B_r^- \) from the remaining rules \( r \) [Gelfond and Lifschitz, 1991]. \( I \) is an answer set of a program \( \Pi \), in symbols \( I \models \Pi \), if \( I \) is a minimal model of \( \Pi \). The set of answer sets are given by \( \mathcal{AS}(\Pi) := \{I \subseteq \text{at}(\Pi) \mid I \models \Pi \} \). Deciding whether a disjunctive program has an answer set is \( \Sigma^p_2 \)-complete [Eiter and Gottlob, 1995]. The problem is called \text{consistency} of an ASP program. For restricted forms, the complexity drops to np-complete [Marek and Truszczynski, 1991]. If \( \mathcal{AS}(\Pi) = \emptyset \), we say that \( \Pi \) is inconsistent.

Example 1. Consider the program \( \Pi := \{a \lor b \leftarrow c \lor d \land \neg d; c \lor d \leftarrow b; \} \). Answer sets of \( \Pi \) are \( \{a, c\}, \{b, c\}, \) and \( \{a, d\} \).

Projected Answer Set Counting (#PA). Let \( \Pi \) be a program and \( P \) be a set of atoms, called projection. Then, projected answer set counting \#PA on \( \Pi \) and \( P \) is the number of subsets \( I \subseteq P \) s.t. \( I \uplus J \) is an answer set of \( \Pi \) for some set \( J \subseteq \text{at}(\Pi) \setminus P \), i.e., \#PA(\Pi, P) := \{|I \cap P \mid I \models \Pi \} \). Problem \#PA(\Pi, A) is \( \Sigma^p_2 \)-complete.

Example 2. Consider \( \Pi \) from Example 1 and its three answer sets as well as the projection \( P := \{a, b\} \). When we project the answer sets to \( P \), we only have the two answer sets \( \{a\} \) and \( \{b\} \), i.e., the projected answer set count \#PA(\Pi, P) is 2.

Tree Decompositions (TDs). We follow standard terminology on graphs and digraphs. For a tree \( T \) with root \( root(T) \) and a node \( t \) of \( T \), we let \( children(t, T) \) be the set of all child nodes \( t' \) of \( t \). Let \( G = (V, E) \) be a graph. A tree decomposition (TD) of graph \( G \) is a pair \( T = (T, \chi) \), where \( T \) is a rooted tree, and \( \chi \) a mapping that assigns to each node \( t \) of \( T \) a set \( \chi(t) \subseteq V \) called a bag, such that the following conditions hold: (i) \( V = \bigcup_{t \in T} \chi(t) \) and \( E \subseteq \bigcup_{t \in T} \chi(t) \times \chi(t') \); 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, \( \text{width}(T) := \max_{t \in T} |\chi(t)| - 1 \). The treewidth \( tw(G) \) of \( G \) is the minimum width \( T \) over all TDs \( T \) 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 \( T \) of width \( w \) [Bodlaender and Kloks, 1996].

For the ease of presentation, we use nice TDs, which can be computed in linear time without increasing the width [Bodlaender and Kloks, 1996] and are defined as follows. For a node \( t \in V \), we say that \( \text{type}(t) \) is \text{leaf} if \( children(t, T) = \emptyset \); \text{join} if \( children(t, T) = \langle t', t'' \rangle \) and \( \chi(t) = \chi(t') \cup \chi(t'') \); \text{irr} \text{’} (“introduce”) if \( children(t, T) = \langle t' \rangle \) and \( \chi(t) = \chi(t') \); \text{rem} (‘’”removal”) if \( children(t, T) = \langle t', t'' \rangle \) and \( \chi(t) \subseteq \chi(t') \cup \chi(t'') \). If every node \( t \in V \), \text{type}(t) \in \{	ext{leaf, join, irr, rem} \}, and \( \chi(t') = \emptyset \) for root and leaf \( t' \), the TD is nice.

In order to use TDs for ASP, we define the primal graph \( G_{\Pi} \) of a program \( \Pi \) [Jakl et al., 2009], which has vertices the atoms of \( \Pi \) with an edge \( \{a, b\} \) whenever there exists a rule \( r \in \Pi \) s.t. \( a,b \in at(r) \).

Example 3. Figure 2 illustrates the primal graph \( G_{\Pi} \) of \( \Pi \) from Example 2 as well as a tree decomposition of \( G_{\Pi} \) of width 2, which is also the treewidth of \( G_{\Pi} \).

Dynamic Programming (DP). Tree Decompositions enable \textit{dynamic programming} algorithms, whose effort is (exponentially) bounded by the width of a TD. These algorithms take a TD \( T = (T, \chi) \) of a given instance (graph), and iterate over every node of \( T \) in \textit{post-order}, i.e., they perform a bottom-up traversal of \( T \). Thereby, for each node \( t \) of \( T \) during the traversal, an algorithm is executed that computes a table \( \tau_t \) for \( t \), which is a set of sequences that are partial solutions to a local problem restricted to the bag \( \chi(t) \). For ASP, we define these local problem by \( DP \text{ program } \Pi_t := \{r \in \Pi \mid at(r) \subseteq \chi(t)\} \). Such DP algorithms are exponential only in the largest bag size, which allow for efficient counting without explicitly enumerating all solutions.

3 Plausibility Reasoning via Projection

Recall that the ASP literature distinguishes the traditional reasoning modes, cautious (skeptical) and brave (credulous) reasoning. Both modes consider a given program \( \Pi \) as well as a set \( Q \) of literals. \textit{Cautious reasoning} then asks whether query \( Q \) matches every answer set of \( \Pi \), whereas \textit{brave reasoning}...
soning focuses only on determining whether \( Q \) matches some answer set of \( \Pi \). Interestingly, both ASP reasoning modes are extreme forms of its own, since cautious reasoning is rather strict or unlikely to hold in general, whereas brave reasoning is very easy to be answered affirmatively, cf. Figure 1.

**Example 4.** Recall \( \Pi \) from Example 1 and observe that brave reasoning on \( \{ a \}, \{ b \}, \{ c \}, \) or \( \{ d \} \) is answered affirmatively, which is not the case for cautious reasoning.

Towards providing more fine-grained reasoning modes, we measure the proportion of answer sets matching query \( Q \). Formally, we define the plausibility of \( Q \) as the proportion of answer sets matching \( Q \).

**Definition 1 (Projected Plausibility).** Let \( \Pi \) be a program, \( Q \) be a query, and \( P \) be projection. Then, \((P-)\)projected plausibility of \( \Pi \) under \( Q \) is defined as
\[
P[\Pi, Q]_P := \sup_{P} \left( \frac{\#PA(\Pi \cup Q, P)}{\#PA(\Pi, P)} \right)
\]

Note that the usage of \( \sup \) prevents division by zero, i.e., in case of consistency plausibility is zero as well. Interestingly, with the help of plausibility, one can define a more fine-grained reasoning mode that generalizes and subsumes both cautious and brave reasoning.

**Definition 2 (Plausibility Reasoning).** Let \( \Pi \) be a program, \( Q \) be a set of literals, \( P \) be a set of atoms, and \( 0 \leq p \leq 1 \). Then, plausibility reasoning on \( Q \) and \( P \) asks if \( P[\Pi, Q]_P \geq p \).

**Example 5.** Plausibility reasoning on a program \( \Pi \) provides more fine-grained reasoning between the two extreme cases of cautious reasoning (\( p = \frac{1}{\at(\Pi)} \)) and brave reasoning (\( p = 1 \)), where \( P = \at(\Pi) \). In our running example, \( P[\Pi, \{ b \}]_{\at(\Pi)} = \frac{2}{3} \), which could show that \( b \) and \( c \) are very likely (compare to, e.g., \( P[\Pi, \{ a \}]_{\at(\Pi)} = \frac{1}{2} \)). This might suffice in drawing conclusions based on quantitative arguments. However, assuming that \( a \) and \( b \) are atoms of interest, indicating that plausibility should be considered using projection \( P = \{ a, b \} \) of Example 2, it turns out that \( P[\Pi, \{ a \}]_P = \frac{1}{2} \). Interestingly, \( P[\Pi, \{ c \}]_P = 1 \), since the answer sets matching \( \{ c \} \) when projected to \( P \) are identical to \( \AS(\Pi) \) restricted to \( P \). As a consequence, \( P[\Pi, \{ c \}]_P \) is more likely than \( P[\Pi, \{ b \}]_P \), since \( \Pi \) under query \( \{ c \} \) leads to every projected answer set of \( \Pi \) and \( P \).

In order to efficiently perform plausibility reasoning, we need a fast implementation of projected answer set counting, which we pursue in the following.

### 3.1 Towards a Hybrid Approach for Plausibility

While existing answer set solvers showed a tremendous performance increase in the last decades, counting requires different techniques, especially if the number of solutions is beyond the limit of enumeration. Over the time, some approaches have been lifted from Boolean satisfiability to ASP (e.g. [Jaikl et al., 2009; Pichler et al., 2010; Eiter et al., 2021; Besin et al., 2021]). However, those are not available for counting with respect to projection (\( \#PA \)), which, as indicated in the preliminaries, is probably harder than plain counting. We propose a new method, that is hybrid in the sense that it combines ideas from parameterized complexity with standard ASP solving, thereby providing a flexible balance between those. Inspired by existing works [Eiben et al., 2021; Dell et al., 2019; Hecher et al., 2020], we define a graph representation that will be used to combine these ideas.

**Definition 3.** Let \( \Pi \) be a program and \( A \subseteq \at(\Pi) \). A projection path in \( G^A_{\Pi} \) is a path in \( G^A_{\Pi} \) of the form \( a, c_1, \ldots, c_\ell, b \) s.t. \( \{ a, b \} \subseteq A, \ell \geq 0 \), and \( \{ c_1, \ldots, c_\ell \} \cap A = \emptyset \). The projection primal graph \( G^A_{\Pi} \) has \( A \) as vertices with an edge between two \( a, b \in A \), whenever there is a projection path from \( a \) to \( b \).

**Example 6.** Recall \( \Pi \) and projection \( P \) from Example 2. Figure 3 shows projection primal graphs \( G^A_{\Pi} \) (left), \( G^{\{b,c\}}_{\Pi} \) (middle), and \( G^{\{a,c,d\}}_{\Pi} \) (right) with an edge \( \{ a, c \} \) due to projection path \( a, b, c \) and an edge \( \{ a, d \} \) due to projection path \( a, b, d \).

Interestingly, this projection graph provides the flexibility to decide the balance between dynamic programming and standard solving. If we choose for \( A = \emptyset \), we obtain by \( G^A_{\Pi} \) the empty graph, i.e., all the effort goes to standard solving with no focus on dynamic programming, whereas \( A = \at(\Pi) \) yields the primal graph \( G^A_{\Pi} = G^A \) for full dynamic programming. We make this precise, by formalizing “effort” in terms of ASP programs that are needed to be solved.

**Definition 4.** Let \( \Pi \) be a program and \( A \subseteq \at(\Pi) \). Further, assume a TD \( T = (T, \chi) \) of \( G^A_{\Pi} \), and a node \( t \) of \( T \). Then, the sub atom (at \( t \)) is given by \( \chi^A_t := \{ x \in \at(\Pi) \setminus A \mid x \notin \bigcup_{\ell \neq t, t \neq T} \chi^A_{\ell} \}, \{ a, b \} \subseteq \chi^A(t), x \) is in a projection path of \( G^A_{\Pi} \) from \( a \) to \( b \). The sub program (at \( t \)) is defined as \( \Pi^A_t := \{ r \in \Pi \mid \Pi_{\ell} \at(r) \subseteq \chi^A_t \} \).

Note that there are many possibilities to fulfill sub atoms of Definition 4, which can be treated differently in actual implementations. Observe that for the primal graph \( G^A_{\Pi} = G^A \), the sub program at any node is empty, whereas for the empty graph \( G^A_{\Pi} \), and a TD thereof consisting only of one node \( t \) has \( \Pi^A_t = \Pi \). Intuitively, the set \( A \) and these definitions provide us a “handle” to share and shift complexity between dynamic programming (DP program \( \Pi_t \)) and standard solving (sub program \( \Pi_{\ell} \)). As a result, we refer to crucial set \( A \) by (projection) abstraction.

**Example 7.** Recall projection primal graphs \( G^{\{b,c\}}_{\Pi} \) and \( G^{\{a,c,d\}}_{\Pi} \) of Figure 3. Compare primal graph vs. projection graph. Figure 4 shows a TD \( T_1 \) of \( G^{\{b,c\}}_{\Pi} \) and a TD \( T_2 \) of \( G^{\{a,c,d\}}_{\Pi} \). The sub programs for \( T_1 \) could be defined by \( \Pi^{\{b,c\}}_{\Pi} := \{ a \lor b \} \) and then \( \Pi^{\{b,c\}}_{\Pi} := \{ c \lor d, c \lor d \} \). Alternatively, one can also fulfill Definition 4 by setting \( \Pi^{\{b,c\}}_{\Pi} \) to the empty set and \( \Pi^{\{b,c\}}_{\Pi} \) to \( \Pi \). In any case, for \( T_1 \) the main effort goes into the sub programs, since the DP programs are empty. This is in contrast to \( T_2 \), where we
The algorithm for hybrid solving is outlined in Listing 1, taking a given depth $dpth$ (initially 0), a program $\Pi$, and a query $Q$. The algorithm assumes that the used TD $T$ is nice, i.e., it gives a clear case distinction between nodes of types $\text{leaf}$, $\text{intrans}$, $\text{rem}$, and $\text{join}$. Note that the usage of nice TDs is not a restriction, since one can overlap these cases accordingly in order to obtain an algorithm for a node of an arbitrary TD. For (empty) leaf nodes, in Line 1 we construct the empty interpretation and set both counters to 1. Then, when an atom $a \in \chi$ is introduced in Line 2, in the current branch of the tree, $a$ is encountered for the first time. Consequently, interpretation $J$ extends $I$ by either including $a$ or not, and we ensure in Line 3 that $J$ is an answer set of $\Pi$. Line 4 (recursively) calls algorithm $\text{HybPA}$ on the increased depth, for obtaining the projected count on the sub program $\Pi'$ under $Q$. Line 5 computes the regular projected count on $\Pi'$, through recursion via algorithm $\text{HybPA}$. Since, intuitively, the computed counts form "sub problems", we merge the resulting counts by multiplication and only keep positive numbers. Further, whenever an atom $a$ gets removed in Line 6, it is guaranteed by the properties of a TD that every rule containing $a$ has been processed. As a result, we remove $a$ from the interpretation and potentially merge counts of interpretations that then coincide by taking the sum (cf. Line 7). Finally, when joining nodes of different (independent) branches, the counters of matching

Finally, the last block performs the actual dynamic programming on the projection graph $G^3_1$, which spans Lines 10–12. Thereby, Line 10 takes the current TD $T = (T, \chi)$ and iterates though every node of $T$ in post-order. For each node $t$ of $T$ during the traversal, an algorithm $\text{PA}_t$ is executed, computing a table $\tau_t$ for $t$, which is a set of sequences of the form $\langle I, q, c \rangle$, where $I \subseteq \chi(t)$ is an interpretation restricted to $\chi(t)$ and $q, c$ are integers used for counting (counters). Precisely, $q$ is the projected answer set count of the program $\Pi'$ under $Q$, where $\Pi'$ contains all DP programs and sub programs for every node below $t$ under $I$. So, $q = \#\text{PA}(\Pi' \cup (\chi(t) \setminus I))$, and $c$ is the projected answer set count of $\Pi'$, i.e., $c = \#\text{PA}(\Pi')$. Observe that therefore these counters for the root table lead to the projected plausibility of the whole program. In Line 12, $\text{PA}_t$ gets as parameters $dpth$, the bag $D$, program $\Pi_t$, sub program $\Pi'_t$, projection $Q$, and tables for all child nodes of $t$.

The algorithm $\text{PA}_t$ is given in Listing 2, which we briefly discuss. For the ease of presentation, $\text{PA}_t$ assumes that the used TD $T$ is nice, i.e., it gives a clear case distinction between nodes of types $\text{leaf}$, $\text{intrans}$, $\text{rem}$, and $\text{join}$. Note that the usage of nice TDs is not a restriction, since one can overlap these cases accordingly in order to obtain an algorithm for a node of an arbitrary TD. For (empty) leaf nodes, in Line 1 we construct the empty interpretation and set both counters to 1. Then, when an atom $a \in \chi$ is introduced in Line 2, in the current branch of the tree, $a$ is encountered for the first time. Consequently, interpretation $J$ extends $I$ by either including $a$ or not, and we ensure in Line 3 that $J$ is an answer set of $\Pi$. Line 4 (recursively) calls algorithm $\text{HybPA}$ on the increased depth, for obtaining the projected count on the sub program $\Pi'$ under $Q$. Line 5 computes the regular projected count on $\Pi'$, through recursion via algorithm $\text{HybPA}$. Since, intuitively, the computed counts form "sub problems", we merge the resulting counts by multiplication and only keep positive numbers. Further, whenever an atom $a$ gets removed in Line 6, it is guaranteed by the properties of a TD that every rule containing $a$ has been processed. As a result, we remove $a$ from the interpretation and potentially merge counts of interpretations that then coincide by taking the sum (cf. Line 7). Finally, when joining nodes of different (independent) branches, the counters of matching
interpretations are multiplied, as shown in Line 9.

**Theorem 1 (Correctness).** Algorithm HybPA is correct. Precisely, for every safe program $\Pi$ and projection $P$, and query $Q$, we have that $\text{HybPA}(0, \Pi, P, Q)$ returns $\mathbb{P}(\Pi, Q)|_{P}$.

**Proof (Sketch).** We present the proof outline for a slightly stronger statement, where for any query $R$ we show correctness: $\text{HybPA}(\text{dpth}, \Pi \sqcup R, P, Q) = \mathbb{P}(\Pi \sqcup R, Q)|_{P}$. For showing correctness of $\text{HybPA}(\text{dpth}, \Pi \sqcup R, P, Q)$ for any $\text{dpth} \geq 0$, we perform structural induction, where we assume induction hypothesis $H$: Correctness of $\text{HybPA}(\text{dpth} + 1, \Pi \sqcup R', P, Q)$ for any $\Pi \subseteq \Pi'$ and any query $R'$.

We show that $\text{HybPA}(\text{dpth}, \Pi, P, Q)$ always returns $\mathbb{P}(\Pi, Q)|_{P}$. Observe that in case of empty projections $P$, $cn \cdot (\mathbb{AS}(\Pi \sqcup Q) \neq \emptyset) = \frac{\mathbb{AS}(\Pi \sqcup Q)}{\max(1, \mathbb{AS}(\Pi))}$ since $cn = 0$ (inconsistency of $\Pi$) implies that also $\Pi \sqcup Q$ is inconsistent ($Q$ only adds constraints, i.e., rules with empty heads). Consequently, Line 3 of Listing 1 returns the projected plausibility. Line 6 fulfills Definition 1 as well. For the soundness of dynamic programming, assume a set $A \subseteq P$ and a TD $T = (T, \chi)$ of $G_{P}$. By Definition 4, each atom $x \in at(\Pi) \setminus A$ appears among sub atoms $\chi_{t}^{A}$ for at most one node $t$ of $T$. Further, by construction of $G_{P}^{\Theta}$ (cf. Definition 3) there always exists a projection path containing $x$. Consequently, there is a unique node $t$ s.t. $x \in \chi_{t}^{A}$ and we follow that for every rule $r \in \Pi$ there is a unique node $t$ of $T$ with either $r \in \Pi$, or $r \in \Pi_{t}^{A}$.

For showing soundness of $\text{PA}_{k}$, we define the following invariant $I$ for any node $t$ of $T$: The sequence $(I, q, c)$ returned by $\text{PA}_{k}$ amounts to $\#\mathbb{PA}(\Pi \sqcup Q, P, c)$ and amounts to $\#\mathbb{PA}(\Pi', P)$, where $\Pi'$ is the whole program of the subtree below $t$, i.e., $\Pi' := \bigcup_{t' \in T, t'} \text{at}(\Pi_{t'} \cup \Pi_{t}^{A})$. Observe that therefore if $I$ holds, Line 13 returns $\mathbb{P}(\Pi, Q)|_{P}$.

The invariant $I$ can be shown by structural induction on $T$, whereby we prove it for any node $t$ of $T$ of type $\text{type}(t) \in \{\text{leaf, intr, rem, join}\}$, thereby assuming $I$ for every child node of $t$. The remainder is a technical case distinction for $\text{type}(t) = \text{intr}$ and Lines 4 and 5. We apply induction hypothesis $H$, thereby concluding correctness of the calls to $\text{HybPA}$.

4 Implementation and Experimental Results

We implemented algorithm HybPA, as given in Listings 1 and 2, into the system HybPA, which is publicly available at https://github.com/maliabd-al-majid/dpdbASP. Our system builds upon clingo 5.5.1 [Gebser et al., 2009] and the tool dpdb [Fichte et al., 2021b] for handling table manipulations during dynamic programming via database management system PostgreSQL version 12.9. HybPA is written in Python3 and uses the decomposition tool htd [Abeher et al., 2017] to efficiently obtain TDs (not necessarily nice) via heuristics.

**Implementation Details.** We briefly discuss implementation specifics along the lines of Listing 1. In Line 2, we call clingo. Since we anyways need the solving call to decide consistency, we extend this in practice and let clingo run for 60 seconds. If the plausibility is already solved by clingo, we stop and output the result. Lines 4 and 9 use htd to decompose heuristically. The constant $\text{threshold}_{\text{abstract}}$ is set to 1000 and $\text{threshold}_{\text{abstract}}$ is set to 8, allowing for aggressive abstractions. Then, for solving projected counting, Line 6 invokes clingo with options “--q” and “--project”, where projection $P$ is set via statements “#show” in the encoding. The computation of suitable abstractions $A \subseteq P$ in Line 8 is implemented using two ASP encodings that are solved by means of clingo. In details, we developed a two-phase approach via heuristics, due to some large instances: First, we estimate among $P$ up to 95 elements of smallest degree, for which we use up to 10 seconds and interrupt afterwards, taking the best result found. Then, among those 95 elements, we invest up to 35 seconds for finding up to 64 elements for the final abstraction. Thereby, we approximate those elements, which lead to a projection primal graph of small width. Both encodings were implemented using weak constraints (optimization), interrupting the computation after the respective timeouts. For details, we refer to the encodings of the supplemental material. The table algorithm of Line 12, i.e., Algorithm PAk of Listing 2, is implemented using PostgreSQL, which is based on relational algebra. Thereby, the construction of the sub program for a node (cf. Definition 4) in Line 12 is implemented such that TD nodes lower in the tree get precedence, i.e., we prefer larger sub programs lower than higher in the tree.

**Experimental Setting and Instances.** Our main interest are more fine-grained reasoning modes between simple brave and cautious reasoning. In order to evaluate practical feasibility, we conducted a series of practical experiments. Computing projected plausibility for a query mainly relies on projected answer set counting for a program and a given projection. Since $\#\mathbb{PA}(\Pi, P)$ is the computationally hardest part being $\#\mathbb{\Sigma}_{2}$-complete, we focus on this part in our experimental evaluation without explicitly assuming queries. In the following, we state two hypotheses that we aim to study and verify in the course of our experimental analysis.

**H1:** We can compute projected answer sets for a large part of crucial problems in ASP allowing for projected plausibility reasoning in ASP.

**H2:** Hybrid solving using HybPA outperforms clingo, in particular, on hard instances.

No dedicated projected answer set counter exist. But the system clingo can solve $\#\mathbb{PA}$ [Gebser et al., 2009], which we take for comparison. As instances, we take two sets (S1) ASP instances from the abstract argumentation competition and (S2) a prototypical ASP domain with reachability and use of transitive closure on real-world graphs. In more details, S1 contains instances of the ICCMA’17 competition [Lagniez et al., 2021]. We focused on the 2017 instances, since the ICCMA’19 instances are known to be on the easier side and the ICCMA’21 can be tackled well by non-hybrid dynamic programming [Lagniez et al., 2021]. The ASP encoding for admissible extensions was taken from the ASPARTIX system [Dvorák et al., 2020]. On the ICCMA’17 instances, we create 4 subsets by randomly selecting 25, 50, 75, and 100 projection atoms. For S2, we used real-world graphs of public transport networks from all over the world, which were used in the PACE’16 and ’17 challenges [Dell et al., 2017]. In total, the transit instances consist of 561 full networks and 2553 subgraphs with different transportation modes. For each
instance, we assume the station with the smallest and largest index to be the start and end stations, respectively. We randomly selected 25 atoms for projection.

Platform, Measure, and Restrictions. All our solvers ran on a cluster consisting of 12 nodes. Each node of the cluster is equipped with two Intel Xeon E5-2650 CPUs, where each of these 12 physical cores runs at 2.2 GHz clock speed and has access to 256 GB shared RAM. Results are gathered on Ubuntu 16.04.1 LTS powered on kernel 4.4.0-139 with hyperthreading disabled using version 3.7.6 of Python3. We mainly compare wall clock time. Run times larger than 1,800 seconds, respectively, count as timeout and main memory (RAM) was restricted to 60GB. We ran jobs exclusively on one machine, where solvers were executed sequentially with exclusive access and no parallel execution of other runs.

Experimental Results and Summary. Figure 5 (left) and (middle) illustrates the number of solved instances for the sets S1 (argumentation) and S2 (reachability) as cactus plot. In addition, Figure 5 (right) visualizes the solved instances for clingo vs HybPA in a direct comparison on set S2. Notably, HybPA solved instances for which the decomposer constructed a decomposition of up to width 100. The detailed results in Table 1 show that a notable number of instances can be solved, which confirms H1. In fact, HybPA completes clasp by allowing to solve a notable number of instances more, both on S1 and S2. Unsurprisingly, the total runtime is quite high for this computationally very hard problem. clingo solves instances with few solutions fast. Average runtime indicates that HybPA tackles harder instances. In particular, we see that HybPA works well on instances with a large number of projected answer sets. Indeed, nested solving outperforms clingo (H2) on both instance sets. The performance gain of HybPA is especially visible in Figure 5. There the rightmost subfigure illustrates a scatter plot. In fact, all instances solved below the dashed line can already be solved by clingo, which hybrid solving can easily take advantage of by appropriate parameter tuning. Instead, we see that a notable number of instances is above the identity line meaning that HybPA can solve those instances while clingo solves only few instances that cannot be solved by hybrid solving.

5 Conclusion and Future Work

This work introduced a more fine-grained reasoning mode for answer set programming (ASP) that is between cautious and brave reasoning. In addition to these, plausibility reasoning allows to draw conclusion based on the the actual (quantitative) proportion, a given query holds. This is visualized in Figure 1 and can be seen as an alternative for drawing conclusions in logic programs, if both cautious and brave reasoning are too weak and too strict, respectively.

In order to provide the flexibility to pull focus or concentrate on variables $V$ of interest (unaffected by encodings involving auxiliary variables), we base plausibility reasoning on the framework of projected model counting, reasoning over answer sets with respect to $V$. Due to the lack of existing (projected) counting systems for ASP, we developed a novel hybrid approach that interleaves ideas from parameterized algorithms (dynamic programming on tree decompositions) and standard CDCL-based solving. This concept uses dedicated abstractions in order to speed-up CDCL-based search via DP algorithms suitable for counting. Our experimental results confirm that this is indeed promising.

For future work, we plan to tightly integrate our method into ASP solvers. We expect that multi-shot extensions of ASP enable advanced learning, by unifying solver entities and sharing learned constraints (nogoods) among different tree decompositions nodes. Further, we are convinced that approximate variants of plausibility reasoning that do not precisely compute plausibility [Kabir et al., 2022], could provide a fruitful alternative for computationally hard applications.

<table>
<thead>
<tr>
<th>Set</th>
<th>System</th>
<th>$n$</th>
<th>$t_{[h]}$</th>
<th>$t_{[s]}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>clingo</td>
<td>1797</td>
<td>683.6</td>
<td>49.17</td>
</tr>
<tr>
<td></td>
<td>HybPA</td>
<td>1913</td>
<td>628.1</td>
<td>90.46</td>
</tr>
<tr>
<td></td>
<td>vbest</td>
<td>1929</td>
<td>622.0</td>
<td>53.53</td>
</tr>
<tr>
<td>S2</td>
<td>clingo</td>
<td>1087</td>
<td>117.2</td>
<td>28.66</td>
</tr>
<tr>
<td></td>
<td>HybPA</td>
<td>1146</td>
<td>89.5</td>
<td>32.88</td>
</tr>
<tr>
<td></td>
<td>vbest</td>
<td>1150</td>
<td>87.5</td>
<td>32.79</td>
</tr>
</tbody>
</table>

Table 1: Detailed results over instances of sets S1 and S2 for the considered systems. $n$ lists the number of solved instances, $t_{[h]}$ contains the elapsed time over all instances in hours, and $t_{[s]}$ states the average elapsed time over the solved instances.

Figure 5: Runtime of systems HybPA and clingo on sets S1 (left) and S2 (middle). The x-axis shows the number $n$ of instances; the y-axis depicts runtimes in seconds. Instances are ordered for each system individually in ascending order of their runtimes. The plot legend lists solvers from best to worst (“right” to “left” in the plot). The right plot is a scatter plot comparing performance of the two systems HybPA and clingo on set S2, omitting instances that cannot be solved by any solver. Each dot compares the runtime of an instance for HybPA vs. clingo. The dashed line illustrates identity. If a value is located above identity HybPA runs faster, otherwise clingo performs better on that instance.
Acknowledgments
This research has been supported by the Vienna Science and Technology Fund (WWTF) grant ICT19-065, and by the Austrian Science Fund (FWF) grants P32830 and Y698.

References
[Lagniez et al., 2021] Jean-Marie Lagniez, Emmanuel Lonca, Jean-Guy Mailly, and Julien Rossit. Results of the fourth international competition on computational models of argumentation, 2021.