TPLP: Page 1–19.

doi:10.1017/S1471068425100124

Counting Answer Sets of Disjunctive Answer Set Programs

MD MOHIMENUL KABIR
School of Computing, National University of Singapore, Singapore, Singapore
(*e-mail:* mahibuet045@gmail.com)

SUPRATIK CHAKRABORTY
Department of Computer Science, Indian Institute of Technology Bombay, Mumbai, India
(*e-mail:* supratik@cse.iitb.ac.in)

KULDEEP S. MEEL
Georgia Institute of Technology, Atlanta, USA
(*e-mail:* meel@cs.toronto.edu)

submitted 19 July 2025; revised 19 July 2025; accepted 27 July 2025

Abstract

Answer Set Programming (ASP) provides a powerful declarative paradigm for knowledge representation and reasoning. Recently, counting answer sets has emerged as an important computational problem with applications in probabilistic reasoning, network reliability analysis, and other domains. This has motivated significant research into designing efficient ASP counters. While substantial progress has been made for normal logic programs, the development of practical counters for disjunctive logic programs remains challenging. We present sharpASP-$\mathcal{SR}$, a novel framework for counting answer sets of disjunctive logic programs based on subtractive reduction to projected propositional model counting. Our approach introduces an alternative characterization of answer sets that enables efficient reduction while ensuring the intermediate representations remain polynomial in size. This allows sharpASP-$\mathcal{SR}$ to leverage recent advances in projected model counting technology. Through extensive experimental evaluation on diverse benchmarks, we demonstrate that sharpASP-$\mathcal{SR}$ significantly outperforms existing counters on instances with large answer set counts. Building on these results, we develop a hybrid counting approach that combines enumeration techniques with sharpASP-$\mathcal{SR}$ to achieve state-of-the-art performance across the full spectrum of disjunctive programs. The extended version of the paper is available at: https://arxiv.org/abs/2507.11655.

KEYWORDS: answer set counting, disjunctive programs, subtractive reduction, projected model counting

1 Introduction

Answer Set Programming (ASP) (Marek and Truszczyński 1999) has emerged as a powerful declarative problem-solving paradigm with applications across diverse application

domains. These include decision support systems (Nogueira *et al.* 2001), systems biology (Gebser *et al.* 2008), and diagnosis and repair (Leone and Ricca 2015). In the ASP paradigm, domain knowledge and queries are expressed through rules defined over propositional atoms, collectively forming an ASP program. Solutions manifest as answer sets – assignments to these atoms that satisfy program rules according to ASP semantics. Our work focuses on the fundamental challenge of answer set counting #ASP: determining the total number of valid answer sets for a given ASP program.

Answer set counting shares conceptual similarities with propositional model counting (#SAT), in which we count satisfying assignments of Boolean formulas (Valiant 1979). While #SAT is #P-complete (Valiant 1979), its practical significance has driven substantial research, yielding practically efficient propositional model counters that combine strong theoretical guarantees with impressive empirical performance. This, in turn, has motivated research in counting techniques beyond propositional logic. Specifically, there has been growing interest in answer set counting, spurred by applications in probabilistic reasoning (Lee *et al.* 2017), network reliability analysis (Kabir and Meel 2023), answer set navigation (Rusovac *et al.* 2024), system biology (Kabir *et al.* 2025), and others (Kabir and Meel 2024, 2025).

Early approaches to answer set counting relied primarily on exhaustive enumeration (Gebser *et al.* 2012). Recent methods have made significant progress by leveraging #SAT techniques (Janhunen 2006; Janhunen and Niemelä 2011; Aziz *et al.* 2015b; Eiter *et al.* 2024; Kabir *et al.* 2024; Fichte *et al.* 2024). Complementing these approaches, dynamic programming on tree decompositions has shown promise for programs with bounded treewidth (Fichte *et al.* 2017; Fichte and Hecher 2019). Most existing answer set counters focus on normal logic programs – a restricted class of ASP. Research on counters for the more expressive class of disjunctive logic programs (Eiter and Gottlob 1995) has received relatively less attention over the years. Our work attempts to bridge this gap by focusing on practically efficient counters for disjunctive logic programs. Complexity theoretic arguments show that barring a collapse of the polynomial hierarchy, translation from disjunctive to normal programs must incur exponential overhead (Eiter *et al.* 2004; Zhou 2014). Consequently, counters optimized for normal programs cannot efficiently handle disjunctive programs, unless the programs themselves have *special properties* (Fichte and Szeider 2015; Ji *et al.* 2016; Ben-Eliyahu-Zohary *et al.* 2017). While loop formula-based translation (Lee and Lifschitz 2003) enables counting in theory, the exponential overhead becomes practically prohibitive for programs with many cyclic atom relationships (Lifschitz and Razborov 2006). Similarly, although disjunctive answer set counting can be reduced to QBF counting in principle (Egly *et al.* 2000), this doesn't yield a practically scalable counter since QBF model counting still does not scale as well in practice as propositional model counting (Shukla *et al.* 2022; Capelli *et al.* 2024). This leads to our central research question: *Can we develop a practical answer set counter for disjunctive logic programs that can scale effectively to handle large answer set counts?*

Our work provides an affirmative answer to this question through several key contributions. We present the design, implementation, and extensive evaluation of a novel counter for disjunctive programs, employing subtractive reduction (Durand *et al.* 2005) to projected propositional model counting (Aziz *et al.* 2015a), while maintaining polynomial formula size growth. The approach first computes an over-approximation of the

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

answer set count and then subtracts the surplus computed using projected counting. This yields a #NP algorithm that leverages recent advances in projected propositional counting (Sharma *et al.* 2019). This approach is theoretically justified: answer set counting for normal programs is in #P (Janhunen and Niemelä 2011; Eiter *et al.* 2021), while for disjunctive programs, it lies in $\# \cdot$ co-NP (Fichte *et al.* 2017). Since $\# \cdot$ co-NP $= \# \cdot \mathsf{P}^{\mathsf{NP}} = \#\mathsf{NP}$ (Hemaspaandra and Vollmer 1995; Durand *et al.* 2005), our reduction is complexity-theoretically sound and yields a practical counting algorithm.

While subtractive reduction for answer set counting has been proposed earlier (Hecher and Kiesel 2023), our work makes several novel contributions beyond the theoretical framework. We develop a complete implementation with careful algorithm design choices and provide comprehensive empirical evaluation across diverse benchmarks. A detailed comparison with the prior approach is presented in Section 5.

Our counter, sharpASP-$\mathcal{SR}$, employs an alternative definition of answer sets for disjunctive programs, extending earlier work on normal programs (Kabir *et al.* 2024). This definition enables the use of off-the-shelf projected model counters without exponential formula growth. Extensive experiments on standard benchmarks demonstrate that sharpASP-$\mathcal{SR}$ significantly outperforms existing counters on instances with large answer set counts. This motivates our development of a hybrid counter combining enumeration and sharpASP-$\mathcal{SR}$ to consistently exceed state-of-the-art performance.

The remainder of the paper is organized as follows. Section 2 covers essential background. Section 3 reviews prior work. Section 4 presents our alternative answer set definition for disjunctive programs. Section 5 details our counting technique sharpASP-$\mathcal{SR}$. Section 6 provides experimental results, and Section 7 concludes the paper with future research directions.

2 Preliminaries

We now introduce some notations and preliminaries needed in subsequent sections.

2.1 Propositional satisfiability

A propositional *variable* v takes value from the domain $\{0, 1\}$ ($\{\mathit{false}, \mathit{true}\}$ resp.). A *literal* ℓ is either a variable or its negation.

A *clause* C is a *disjunction* ($\vee$) of literals. For clarity, we often represent a clause as a set of literals, implicitly meaning that all literals in the set are disjoined in the clause. A *unit clause* is a clause with a single literal. The constraint represented by a clause $C \equiv (\neg x_1 \vee \ldots \vee \neg x_k \vee x_{k+1} \vee \ldots \vee x_{k+m})$ can be expressed as a logical *implication* as follows: $(x_1 \wedge \ldots \wedge x_k) \longrightarrow (x_{k+1} \vee \ldots \vee x_{k+m})$, where the conjunction of literals $x_1 \wedge \ldots \wedge x_k$ is known as the *antecedent* and the disjunction of literals is known as the *consequent*. If $k = 0$, the antecedent of the implication is *true*, and if $m = 0$, the consequent is *false*.

A formula ϕ is said to be in *conjunctive normal form (CNF)* if it is a conjunction ($\wedge$) of clauses. For convenience of exposition, a CNF formula is often represented as a set of clauses, implicitly meaning that all clauses in the set are conjoined in the formula. We denote the set of variables of a propositional formula ϕ as $\mathsf{Var}(\phi)$.

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

An assignment over a set X of propositional variables is a mapping $\tau : X \to \{0, 1\}$. For a variable $x \in X$, we define $\tau(\neg x) = 1 - \tau(x)$. An assignment τ over $\mathsf{Var}(\phi)$ is called a *model* of ϕ, represented as $\tau \models \phi$, if ϕ evaluates to *true* under the assignment τ, as per the semantics of propositional logic. A formula ϕ is said to be *SAT* (resp. *UNSAT*) if there exists a model (resp. no model) of ϕ. Given an assignment τ, we use the notation τ^+ (resp. τ^-) to denote the set of variables that are assigned 1 or *true* (resp. 0 or *false*).

Given a CNF formula ϕ (as a set of clauses) and an assignment $\tau : X \to \{0, 1\}$, where $X \subseteq \mathsf{Var}(\phi)$, the *unit propagation* of τ on ϕ, denoted $\phi|_\tau$, is another CNF formula obtained by applying the following steps recursively: (a) remove each clause C from ϕ that contains a literal ℓ s.t. $\tau(\ell) = 1$; (b) remove from each clause C in ϕ all literals ℓ s.t. either $\tau(\ell) = 0$ or there exists a *unit clause* $\{\neg\ell\}$, that is, a clause with a single literal $\neg\ell$; and (c) apply the above steps recursively to the resulting CNF formula until there are no further syntactic changes to the formula. As a special case, the unit propagation of an empty formula is the empty formula. It is not hard to show that unit propagation of τ on ϕ always terminates or reaches *fixed point*. We say that τ *unit propagates* to literal ℓ in ϕ, if $\{\ell\}$ is a unit clause in $\phi|_\tau$, that is, if $\{\ell\} \in \phi|_\tau$.

Given a propositional formula ϕ, we use $\#\phi$ to denote the count of models of ϕ. If $X \subseteq \mathsf{Var}(\phi)$ is a set of variables, then $\#\exists X\phi$ denotes the count of models of ϕ after disregarding assignments to the variables in X. In other words, two different models of ϕ that differ only in the assignment of variables in X are counted as one in $\#\exists X\phi$.

2.2 Answer set programming

An *answer set program* P consists of a set of rules, where each rule is structured as follows:

$$\text{Rule } r\text{: } a_1 \vee \ldots \vee a_k \leftarrow b_1, \ldots, b_m, \mathsf{not}\ c_1, \ldots, \mathsf{not}\ c_n \tag{1}$$

where $a_1, \ldots, a_k, b_1, \ldots, b_m, c_1, \ldots, c_n$ are propositional variables or *atoms*, and k, m, n are non-negative integers. The notations P and $\mathsf{at}(P)$ refer to the rules and atoms of the program P, respectively. In rule r above, the operator "not" denotes *default negation* (Clark 1978). For each such rule r, we use the following notation: the set of atoms $\{a_1, \ldots, a_k\}$ constitutes the *head* of r, denoted by $\mathsf{Head}(r)$, the set of atoms $\{b_1, \ldots, b_m\}$ is referred to as the *positive body atoms* of r, denoted by $\mathsf{Body}(r)^+$, and the set of atoms $\{c_1, \ldots, c_n\}$ is referred to as the *negative body atoms* of r, denoted by $\mathsf{Body}(r)^-$. We use $\mathsf{Body}(r)$ to denote the set of literals $\{b_1, \ldots, b_m, \neg c_1, \ldots, \neg c_n\}$. For notational convenience, we sometimes use $\bot$ on the left (resp. $\top$ on the right) of $\leftarrow$ in a rule r to denote that $\mathsf{Head}(r)$ (resp. $\mathsf{Body}(r)$) is empty. A program P is called a *disjunctive logic program* if $\exists r \in P$ such that $|\mathsf{Head}(r)| \geq 2$ (Ben-Eliyahu and Dechter 1994); otherwise, it is a *normal logic program*. Our focus in this paper is on disjunctive logic programs.

Following standard ASP semantics, an interpretation M over the atoms $\mathsf{at}(P)$ specifies which atoms are present in M, or equivalently assigned *true* in M. Specifically, atom a is *true* in M if and only if $a \in M$. An interpretation M satisfies a rule r, denoted by $M \models r$, if and only if $(\mathsf{Head}(r) \cup \mathsf{Body}(r)^-) \cap M \neq \emptyset$ or $\mathsf{Body}(r)^+ \setminus M \neq \emptyset$. An interpretation M is a *model* (though not necessarily an answer set) of P, denoted by $M \models P$, if M satisfies every rule in P, that is, $\forall_{r \in P} M \models r$. The *Gelfond–Lifschitz (GL) reduct* of a program

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

P with respect to an interpretation M is defined as $P^M = \{\mathsf{Head}(r) \leftarrow \mathsf{Body}(r)^+ \mid r \in P, \mathsf{Body}(r)^- \cap M = \emptyset\}$ (Gelfond and Lifschitz 1991). An interpretation M is an *answer set* of P if $M \models P$ and $\nexists M' \subset M$ such that $M' \models P^M$. In general, an ASP P may have multiple answer sets. The notation $\mathsf{AS}(P)$ denotes the set of all answer sets of P.

2.3 Clark completion

The *Clark Completion* (Lee and Lifschitz 2003) translates an ASP program P to a propositional formula $\mathsf{Comp}(P)$. The formula $\mathsf{Comp}(P)$ is defined as the conjunction of the following propositional implications:

1. (group 1) for each atom $a \in \mathsf{at}(P)$ s.t. $\nexists r \in P$ and $a \in \mathsf{Head}(r)$, add a unit clause $\neg a$ to $\mathsf{Comp}(P)$
2. (group 2) for each rule $r \in P$, add the following implication to $\mathsf{Comp}(P)$:
$$\bigwedge_{\ell \in \mathsf{Body}(r)} \ell \longrightarrow \bigvee_{x \in \mathsf{Head}(r)} x$$
3. (group 3) for each atom $a \in \mathsf{at}(P)$ occuring in the head of at least one of the rules of P, let $r_1, \ldots, r_k$ be precisely all rules containing a in the head, and add the following implication to $\mathsf{Comp}(P)$:
$$a \longrightarrow \bigvee_{i \in [1,k]} \Big(\bigwedge_{\ell \in \mathsf{Body}(r_i)} \ell \wedge \bigwedge_{x \in \mathsf{Head}(r_i) \setminus \{a\}} \neg x \Big)$$

It is known that every answer set of P satisfies $\mathsf{Comp}(P)$, although the converse is not necessarily true (Lee and Lifschitz 2003).

Given a program P, we define the *positive dependency graph* $\mathsf{DG}(P)$ of P as follows. Each atom $x \in \mathsf{at}(P)$ corresponds to a vertex in $\mathsf{DG}(P)$. For $x, y \in \mathsf{at}(P)$, there is an edge from y to x in $\mathsf{DG}(P)$ if there exists a rule $r \in P$ such that $x \in \mathsf{Body}(r)^+$ and $y \in \mathsf{Head}(r)$ (Kanchanasut and Stuckey 1992). A set of atoms $L \subseteq \mathsf{at}(P)$ forms a *loop* in P if, for every $x, y \in L$, there is a path from x to y in $\mathsf{DG}(P)$, and all atoms (equivalently, nodes) on the path belong to L. An atom x is called a *loop atom* of P if there is a loop L in $\mathsf{DG}(P)$ such that $x \in L$. We use the notation $\mathsf{LA}(P)$ to denote the set of all loop atoms of the program P. If there is no loop in P, we call the program *tight*; otherwise, it is said to be *non-tight* (Fages 1994).

Example 1.
Consider the program $P = \{r_1 : p_0 \vee p_1 \leftarrow \top;\ r_2 : q_0 \vee q_1 \leftarrow \top;\ r_3 : q_0 \leftarrow w;\ r_4 : q_1 \leftarrow w;\ r_5 : w \leftarrow p_0;\ r_6 : w \leftarrow p_1, q_1;\ r_7 : \bot \leftarrow \mathsf{not}\ w;\}$.

The group 2 clauses in $\mathsf{Comp}(P)$ *are:* $\{(p_0 \vee p_1), (q_0 \vee q_1), (\neg w \vee q_0), (\neg w \vee q_1), (\neg p_0 \vee w), (\neg p_1 \vee \neg q_1 \vee w), (w)\}$; *and the group 3 clauses are:* $\{(p_0 \longrightarrow \neg p_1), (p_1 \longrightarrow \neg p_0), (q_0 \longrightarrow (\neg q_1 \vee w)), (q_1 \longrightarrow (\neg q_0 \vee w)), (w \longrightarrow (p_0 \vee (p_1 \wedge q_1)))\}$.

Since each atom occurs in at least one rule's head, there are no group 1 clauses. Thus, $\mathsf{Comp}(P)$ *consists of only group 2 and group 3 clauses. In this program, the set of loop atoms is* $\{q_1, w\}$.

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

2.4 Subtractive reduction

Borrowing notation from Durand *et al.* (2005), suppose Σ and Γ are alphabets, and $Q_1, Q_2 \subseteq \Sigma^* \times \Gamma^*$ are binary relations such that for each $x \in \Sigma^*$, the sets $Q_1(x) = \{y \in \Gamma^* \mid Q_1(x, y)\}$ and $Q_2(x) = \{y \in \Gamma^* \mid Q_2(x, y)\}$ are finite. Let $\#Q_1$ and $\#Q_2$ denote counting problems that require us to find $|Q_1(x)|$ and $|Q_2(x)|$, respectively, for a given $x \in \Sigma^*$. We say that $\#Q_1$ strongly reduces to $\#Q_2$ via a subtractive reduction, if there exist polynomial-time computable functions f and g such that for every string $x \in \Sigma^*$, the following hold: (a) $Q_2(g(x)) \subseteq Q_2(f(x))$, and (b) $|Q_1(x)| = |Q_2(f(x))| - |Q_2(g(x))|$. As we will see in Section 5, in our context, $\#Q_1$ is the answer set counting problem for disjunctive logic programs, and $\#Q_2$ is the projected model counting problem for propositional formulas.

3 Related Work

Answer set counting exhibits distinct complexity characteristics across different classes of logic programs. For normal logic programs, the problem is #P-complete (Valiant 1979), while for disjunctive logic programs, it rises to $\# \cdot \mathsf{co\text{-}NP}$ (Fichte *et al.* 2017). This complexity gap between normal and disjunctive programs highlights that answer set counting for disjunctive logic programs is likely harder than that for normal logic programs, under standard complexity theoretic assumptions.

This complexity distinction is also reflected in the corresponding decision problems as well. While determining the existence of an answer set for normal logic programs is NP-complete (Marek and Truszczyński 1991), the same problem for disjunctive logic programs is Σ_2^p-complete (Eiter and Gottlob 1995). This fundamental difference in complexity has important implications for translations between program classes. Specifically, a polynomial-time translation from disjunctive to normal logic programs that preserves the count of answer sets does exist unless the polynomial hierarchy collapses (Janhunen *et al.* 2006; Zhou 2014; Ji *et al.* 2016).

Much of the early research on answer set counting focused on normal logic programs (Aziz *et al.* 2015b; Eiter *et al.* 2021, 2024; Kabir *et al.* 2024). The methodologies for counting answer sets have evolved significantly over time. Initial approaches relied primarily on enumerations (Gebser *et al.* 2012). More recent methods have adopted advanced algorithmic techniques, particularly tree decomposition and dynamic programming. Fichte *et al.* (2017) developed DynASP, an exact answer set counter optimized for instances with small treewidth. Kabir *et al.* (2022) explored a different direction with ApproxASP, which implements an approximate counter providing (ε, δ)-guarantees, with the adaptation of hashing-based techniques.

Subtraction-based techniques have emerged as promising approaches for various counting problems, for example, MUS counting (Bendík and Meel 2021). In the context of answer set counting, subtraction-based methods were introduced in Hecher and Kiesel (2023) and Fichte *et al.* (2024). These methods employ a two-phase strategy: initially overcounts the answer set count, subsequently subtracts the surplus to obtain the exact count. Hecher and Kiesel (2023) developed a method utilizing projected model counting over propositional formulas with projection sets. A detailed comparison of our work with

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

their approach is provided at the end of Section 5. In a different direction, Fichte *et al.* (2024) proposed iascar, specifically tailored for normal programs. Their approach iteratively refines the overcount count by enforcing *external support* for each loop and applying the *inclusion-exclusion principle*. The key distinction of iascar lies in its comprehensive consideration of external supports for all cycles in the counting process.

4 An Alternative Definition of Answer Sets

In this section, we present an alternative definition of answer sets for disjunctive logic programs, that generalizes the work of Kabir *et al.* (2024) for normal logic programs. Before presenting the alternative definition of answer sets, we provide a definition of *justification*, which is crucial to understand our technical contribution.

4.1 Checking Justification in ASP

Intuitively, justification refers to a *structured explanation* for *why* a literal (atom or its negation) is *true* or *false* in a given answer set (Pontelli *et al.* 2009; Fandinno and Schulz 2019). Recall that the classical definition of answer sets requires that each *true* atom in an interpretation, that also appears at the head of a rule, must be justified (Gelfond and Lifschitz 1988; Lifschitz 2010). More precisely, given an interpretation M s.t. $M \models P$, ASP solvers check whether some of the atoms in M can be set to *false*, while satisfying the reduct program P^M (Lierler 2005). We use the notation τ_M to denote the assignment of propositional variables corresponding to the interpretation M. Furthermore, we say that $x \in \tau_M^+$ (resp. τ_M^-) iff $\tau_M(x) = 1$ (resp. 0).

While the existing literature typically formulates justification using *rule-based* or *graph-based* explanations (Fandinno and Schulz 2019), we propose a model-theoretic definition from the reduct P^M, for each interpretation $M \models P$. An atom $x \in M$ is *justified* in M if for every $M' \models P^M$ such that $M' \subseteq M$, it holds that $x \in M'$. In other words, removing x from M violates the satisfaction of P^M. The definition is compatible with the standard characterization of answer sets, since M is an answer set, when no $M' \subsetneq M$ exists such that $M' \models P^M$; that is, each atom $x \in M$ is justified. Conversely, an atom $x \in M$ is *not justified* in M if there exists a proper subset $M' \subset M$ such that $M' \models P^M$ and $x \notin M'$. This notion of justification also aligns with how SAT-based ASP solvers perform *minimality checks* (Lierler 2005) – such solvers encode P^M as a set of implications (see definition of P^M in Section 2) and check the satisfiability of the formula: $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigvee_{x \in \tau_M^+} \neg x$.

Proposition 1.
For a program P and each interpretation M such that $M \models P$, if the formula $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigvee_{x \in \tau_M^+} \neg x$ is satisfiable, then some atoms in M are not justified.

The proposition holds by definition. In the above formula, the term $\bigwedge_{x \in \tau_M^-} \neg x$ encodes the fact that variables assigned *false* in M need no justification. On the other hand, the term $\bigvee_{x \in \tau_M^+} \neg x$ verifies whether any of the variables assigned *true* in M is not justified.

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

Example 1 (continued).
Consider the following two interpretations over $\mathsf{at}(P)$:

- $M_1 = \{p_0, w, q_0, q_1\}$*: Clearly,* $\tau_{M_1} = \{p_0, w, q_0, q_1, \neg p_1\}$*. As no strict subset of* M_1 *satisfies* P^{M_1}*, each atom of* M_1 *is justified.*
- $M_2 = \{p_1, w, q_0, q_1\}$*: Here,* $\tau_{M_2} = \{p_1, w, q_0, q_1, \neg p_0\}$*. Note that* $\tau_{M_2} \models \mathsf{Comp}(P)$*. The program* P^{M_2} *includes all rules of* P *except rule* r_7*. There is an interpretation* $\{p_1, q_0\} \subset M_2$ *that satisfies* P^{M_2}*. It indicates that atoms* q_1 *and* w *are not justified in* M_2*.*

We now show that under the Clark completion of a program, or when $\tau_M \models \mathsf{Comp}(P)$, then it suffices to check justification of only loop atoms of P in the interpretation M. Note that the ASP counter, sharpASP (Kabir *et al.* 2024), also checks justifications for loop atoms in the context of normal logic programs. Our contribution lies in proving the sufficiency of checking justifications for loop atoms even in the context of disjunctive logic programs – a non-trivial generalization. Specifically, we establish that when $\tau_M \models \mathsf{Comp}(P)$, if some atoms in M are not justified, then there must also be some loop atoms in M that is not justified. To verify justifications for only loop atoms, we check the satisfiability of the formula: $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x$.

Proposition 2.
For each $M \subseteq \mathsf{at}(P)$ *such that* $M \models P$*, if the formula* $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x$ *is satisfiable, then some of the loop atoms in* M *are not justified; otherwise, each loop atom in* M *is justified.*

Proof.
Since $\tau_M \models \mathsf{Comp}(P)$, it implies that $\tau_M \models P^M$. Thus, the formula $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x$ is satisfiable.

If $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x$ is satisfiable, then there are some loop atoms from $\tau_M^+ \cup \mathsf{LA}(P)$ that can be set to *false*, while satisfying the formula $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x$. It indicates that some of the loop atoms of M are not justified; otherwise, each loop atom of M is justified. □

In this above formula, the term $\bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x$ ensures that we are not concerned with justifications for non-loop atoms. On the other hand, the term $\bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x$ specifically verifies whether any of the loop atoms assigned to *true* in M is not justified.

For every interpretation $M \models P$, checking justification of all loop atoms of M suffices to check justification all atoms of M. The following lemma formalizes our claim:

Lemma 1.
For a given program P *and each interpretation* $M \subseteq \mathsf{at}(P)$ *such that* $\tau_M \models \mathsf{Comp}(P)$*, if* $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigvee_{x \in \tau_M^+} \neg x$ *is SAT then* $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x$ *is also SAT.*

Proof.
For notational clarity, let A and B denote the formulas $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigvee_{x \in \tau_M^+} \neg x$ and $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x$, respectively.

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

We use proof by contradiction. Suppose, if possible, A is SAT but B is UNSAT. Given that A is SAT, we know that some atoms are not justified in M (Proposition 1). Similarly, since B is UNSAT, we know that all loop atoms are justified in M (Proposition 2). Therefore, there must be a non-loop atom, say x_1, that is not justified in M. Since $x_1 \in M$ and $\tau_M \models \mathsf{Comp}(P)$, according to group 3 implications in the definition of Clark completion, there exists a rule $r_1 \in P$ such that $\mathsf{Body}(r_1) \wedge \bigwedge_{x \in \mathsf{Head}(r_1) \setminus \{x_1\}} \neg x$ is *true* under τ_M. It follows that there exists an atom $x_2 \in \mathsf{Body}(r_1)^+$ that is not justified; otherwise, the atom x_1 would have no other option but be justified. Now, we can repeat the same argument we presented above for x_1, but in the context of the non-justified atom x_2 in M. By continuing this argument, we obtain a sequence of not justified atoms $\{x_1, x_2, \ldots\}$, such that the underlying set is a subset of M. There are two possible cases to consider: either (i) the sequence $\{x_i\}$ is unbounded, or (ii) for some $i < j$, $x_i = x_j$. Case (i) contradicts the finiteness of $\mathsf{at}(P)$. Case (ii) implies that some loop atoms are not justified – a contradiction of our premise! □

Example 1 (continued).
Consider the following two interpretations over $\mathsf{at}(P)$:

- $M_1 = \{p_0, w, q_0, q_1\}$*: Clearly,* $\tau_{M_1} = \{p_0, w, q_0, q_1, \neg p_1\}$*. Note that* $M_1 \in \mathsf{AS}(P)$*, as no strict subset of* M_1 *satisfies* P^{M_1}*.*
- $M_2 = \{p_1, w, q_0, q_1\}$*: Here,* $\tau_{M_2} = \{p_1, w, q_0, q_1, \neg p_0\}$*. While* $\tau_{M_2} \models \mathsf{Comp}(P)$*, it can be shown that* $M_2 \notin \mathsf{AS}(P)$*. The program* P^{M_2} *includes all rules of* P *except rule* r_7*. There is an interpretation* $\{p_1, q_0\} \subset M_2$ *that satisfies* P^{M_2}*. This means that the atoms* q_1 *and* w *in* M_2 *are not justified. Note that both* q_1 *and* w *are loop atoms in program* P*.*

4.2 $\mathsf{Copy}(P)$ *for disjunctive logic programs*

Toward establishing an alternative definition of answer sets for disjunctive logic programs, we now generalize the copy operation used in Kabir *et al.* (2024) in the context of normal logic programs. Given an ASP program P, for each loop atom $x \in \mathsf{LA}(P)$, we introduce a fresh variable x' such that $x' \notin \mathsf{at}(P)$. We refer to x' as the *copy variable* of x. Similar to Kabir *et al.* (2024) and Kabir (2024), the operator $\mathsf{Copy}(P)$ returns the following set of implicitly conjoined implications.

1. (type 1) for each loop atom $x \in \mathsf{LA}(P)$, the implication $x' \longrightarrow x$ is included in $\mathsf{Copy}(P)$.
2. (type 2) for each rule $r = a_1 \vee \ldots \vee a_k \leftarrow b_1, \ldots, b_m, \mathsf{not}\ c_1, \ldots, \mathsf{not}\ c_n \in P$ such that $\{a_1, \ldots a_k\} \cap \mathsf{LA}(P) \neq \emptyset$, the implication $\psi(b_1) \wedge \ldots \wedge \psi(b_m) \wedge \neg c_1 \wedge \ldots \wedge \neg c_n \longrightarrow \psi(a_1) \vee \ldots \vee \psi(a_k)$ is included in $\mathsf{Copy}(P)$, where $\psi(x)$ is a function defined as follows: $\psi(x) = \begin{cases} x' & \text{if } x \in \mathsf{LA}(P) \\ x & \text{otherwise} \end{cases}$
3. No other implication is included in $\mathsf{Copy}(P)$.

Note that we do not introduce any type 2 implication for a rule r if $\mathsf{Head}(r) \cap \mathsf{LA}(P) = \emptyset$. In a type 2 implication, each loop atom in the head and each

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

positive body atom is replaced by its corresponding copy variable. As a special case, if the program P is tight then $\mathsf{Copy}(P) = \emptyset$.

Example 1(continued).
For the given program P, we have $\mathsf{LA}(P) = \{q_1, w\}$. Therefore, $\mathsf{Copy}(P)$ introduces two fresh copy variables $q_1{}'$ and w', and adds the the following implications: $\{q_1{}' \longrightarrow q_1, w' \longrightarrow w, q_0 \vee q_1{}', w' \longrightarrow q_1{}', p_0 \longrightarrow w', p_1 \wedge q_1{}' \longrightarrow w'\}$.

We now demonstrate an important relationship between P^M and $\mathsf{Copy}(P)|_{\tau_M}$, for a given interpretation M. Specifically, we show that we can use $\mathsf{Copy}(P)|_{\tau_M}$, instead of P^M, to check the justification of loop atoms in M. While sharpASP also utilizes a similar idea for normal programs, the following lemma (Lemma 2) formalizes this important relationship in the context of the more general class of disjunctive logic programs.

Lemma 2.
For a given program P and each interpretation $M \subseteq \mathsf{at}(P)$ such that $\tau_M \models \mathsf{Comp}(P)$,

1. *the formula $\mathsf{Copy}(P)|_{\tau_M} \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x'$ is SAT if and only if $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x$ is SAT*
2. *the formula $\mathsf{Copy}(P)|_{\tau_M} \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x'$ is SAT if and only if $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigvee_{x \in \tau_M^+} \neg x$ is SAT*

The proof is deferred to the extended version of the paper.[1]

We now integrate Clark's completion, the copy operation introduced above, and the core idea from Lemma 2 to propose an alternative definition of answer sets.

Lemma 3.
For a given program P and each interpretation $M \subseteq \mathsf{at}(P)$ such that $\tau_M \models \mathsf{Comp}(P)$, $M \in \mathsf{AS}(P)$ if and only if the formula $\mathsf{Copy}(P)|_{\tau_M} \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x'$ is UNSAT.

The proof follows directly from the correctness of Lemma 2 and from the definition of answer sets based on the GL reduct P^M (see Section 2).

Example 1(continued).
Consider two interpretations $M_1, M_2 \subseteq \mathsf{at}(P)$:

- *$M_1 = \{p_0, w, q_0, q_1\}$, where $\tau_{M_1} = \{p_0, w, q_0, q_1, \neg p_1\}$. Note that $M_1 \in \mathsf{AS}(P)$ and we can verify that $\mathsf{Copy}(P)|_{\tau_{M_1}} \wedge (\neg q_1{}' \vee \neg w')$ is UNSAT.*
- *$M_2 = \{p_1, w, q_0, q_1\}$, where $\tau_{M_2} = \{p_1, w, q_0, q_1, \neg p_0\}$. Here, $M_2 \notin \mathsf{AS}(P)$. While $\tau_{M_2} \models \mathsf{Comp}(P)$, we can see that $\mathsf{Copy}(P)|_{\tau_{M_2}} \wedge (\neg q_1{}' \vee \neg w')$ is SAT.*

Our alternative definition of answer sets, formalized in Lemma 3, implies that the complexity of checking answer sets for disjunctive logic programs is in co-NP. In contrast, the definition in Kabir *et al.* (2024), which applies only to normal logic programs, allows answer set checking for this restricted class of programs to be accomplished in polynomial time. Note that the $\mathsf{Copy}(P)$ has similarities with formulas introduced in Fichte and Szeider (2015) and Hecher and Kiesel (2023) for co-NP checks.

[1] The extended version of the paper is available at: https://arxiv.org/abs/2507.11655

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

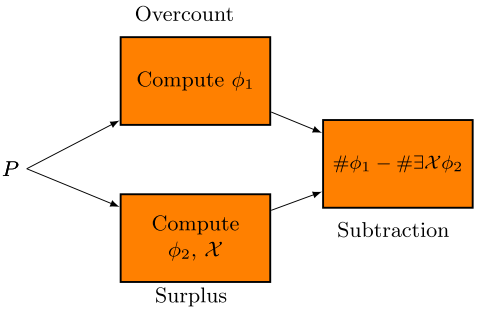

Fig. 1. The high-level architecture of sharpASP-$\mathcal{SR}$ for a program P.

In the following section, we utilize the definition in Lemma 3 to count of models of $\mathsf{Comp}(P)$ that are not answer sets of P. This approach allows us to determine the number of answer sets of P via subtractive reduction.

5 Answer Set Counting: sharpASP-$\mathcal{SR}$

We now introduce a subtractive reduction-based technique for counting the answer sets of disjunctive logic programs. This approach reduces answer set counting to projected model counting for propositional formulas. Note that projected model counting for propositional formulas is known to be in #NP (Aziz *et al.* 2015a); hence reducing answer set counting (a $\# \cdot$ coNP-complete problem) to projected model counting makes sense.[2] In contrast, answer set counting of normal logic programs is in #P, and is therefore easier.

At a high level, the proposed subtractive reduction approach is illustrated in Figure 1. For a given ASP program P, we overcount the answer sets of P by considering the satisfying assignment of an appropriately constructed propositional formula ϕ_1 (Overcount). The value $\#\phi_1$ counts all answer sets of P, but also includes some interpretations that are not answer sets of P. To account for this surplus, we introduce another Boolean formula ϕ_2 and a projection set $\mathcal{X}$ such that $\#\exists\mathcal{X}\phi_2$ counts the surplus from the overcount of answer sets (Surplus). To correctly count the surplus, we employ the alternative answer set definition outlined in Lemma 3. Finally, the count of answer sets of P is determined by $\#\phi_1 - \#\exists\mathcal{X}\phi_2$.

5.1 Counting overcount (ϕ_1)

Given a program P, the count of models of $\mathsf{Comp}(P)$ provides an overcount of the count of answer sets of P. In the case of tight programs, the count of answer sets is equivalent to the count of models of $\mathsf{Comp}(P)$ (Lee and Lifschitz 2003). However, for non-tight programs, the count of models of $\mathsf{Comp}(P)$ overcounts $|\mathsf{AS}(P)|$. Therefore, we use

$$\phi_1 = \mathsf{Comp}(P) \tag{2}$$

[2] The classes #NP and $\# \cdot$ coNP are known to coincide.

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

5.2 Counting surplus (ϕ_2)

To count the surplus, we utilize the alternative answer set definition presented in Lemma 3. We use a propositional formula ϕ_2, in which for each loop atom x, there are two fresh copy variables: x' and $x^\star$. We introduce two sets of copy operations of P, namely, $\mathsf{Copy}(P)'$ and $\mathsf{Copy}(P)^\star$, where for each loop atom x, the corresponding copy variables are denoted as x' and $x^\star$, respectively. We use the notations CV' and $\mathsf{CV}^\star$ to refer to the copy variables of $\mathsf{Copy}(P)'$ and $\mathsf{Copy}(P)^\star$, respectively; that is, $\mathsf{CV}' = \{x' | x \in \mathsf{LA}(P)\}$ and $\mathsf{CV}^\star = \{x^\star | x \in \mathsf{LA}(P)\}$. To compute the surplus, we define the formula $\phi_2(\mathsf{at}(P), \mathsf{CV}', \mathsf{CV}^\star)$ as follows:

$$\phi_2(\mathsf{at}(P), \mathsf{CV}', \mathsf{CV}^\star) = \mathsf{Comp}(P) \wedge \mathsf{Copy}(P)' \wedge \mathsf{Copy}(P)^\star \wedge \bigwedge_{x \in \mathsf{LA}(P)} (x' \longrightarrow x^\star) \wedge \bigvee_{x \in \mathsf{LA}(P)} (\neg x' \wedge x^\star) \tag{3}$$

Lemma 4.
The number of models of $\mathsf{Comp}(P)$ *that are not answer sets of* P *can be computed as* $\#\exists \mathsf{CV}', \mathsf{CV}^\star\ \phi_2(\mathsf{at}(P), \mathsf{CV}', \mathsf{CV}^\star)$, *where the formula* ϕ_2 *is defined in* Equation (3).

Proof.
From the definition of ϕ_2 (Equation (3)), we know that for every model $\sigma \models \phi_2$, the assignment to CV' and $\mathsf{CV}^\star$ is such that $\forall x \in \mathsf{LA}(P), \sigma(x') \leq \sigma(x^\star)$ and $\exists x \in \mathsf{LA}(P), \sigma(x') < \sigma(x^\star)$.[3] Let M be the corresponding interpretation over $\mathsf{at}(P)$ of the satisfying assignment σ. Since $\sigma \models \phi_2$, $\tau_M \models \mathsf{Comp}(P)$ and some of the copy variables $x' \in \mathsf{CV}'$ can be set to *false* where $\sigma(x) = true$, while after setting the copy variables x' to *false*, the formula $\mathsf{Copy}(P)_{|\tau_M}$ is still satisfied. According to Lemma 3, we can conclude that $M \notin \mathsf{AS}(P)$. As a result, $\#\exists \mathsf{CV}', \mathsf{CV}^\star\ \phi_2(\mathsf{at}(P), \mathsf{CV}', \mathsf{CV}^\star)$ counts all interpretations that are not answer sets of P. □

Theorem 1.
For a given program P, *the number of answer sets:* $|\mathsf{AS}(P)| = \#\phi_1 - \#\exists \mathcal{X} \phi_2$, *where* $\mathcal{X} = CV' \cup CV^\star$, *and* ϕ_1 *and* ϕ_2 *are defined in Equations (2) and (3). Furthermore, both* ϕ_1 *and* ϕ_2 *can be computed in time polynomial in* $|P|$.

The proof is deferred to the extended version of the paper.

Now recall to subtractive reduction definition (ref. Section 2), for a given ASP program P, $f(P)$ computes the formula ϕ_1, and $g(P)$ computes the formula $\exists \mathsf{CV}', \mathsf{CV}^* \phi_2$.

We refer to the answer set counting technique based on Theorem 1 as sharpASP-$\mathcal{SR}$. While sharpASP-$\mathcal{SR}$ shares similarities with the answer set counting approach outlined in Hecher and Kiesel (2023), there are key differences between the two techniques. First, instead of counting the number of models of Clark completion, the technique in Hecher and Kiesel (2023) counts *non-models* of the Clark completion. Second, to count the surplus, sharpASP-$\mathcal{SR}$ introduces copy variables only for loop variables, whereas the approach of Hecher and Kiesel (2023) introduces copy (referred to as *duplicate* variable) variables for every variable in the program. Third, sharpASP-$\mathcal{SR}$ focuses on generating

[3] We use $0 < 1$ for this discussion.

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

a copy program over the cyclic components of the input program, while their approach duplicates the entire program. A key distinction is that the size of Boolean formulas introduced by Hecher and Kiesel (2023) depends on the tree decomposition of the input program and its treewidth, assuming that the treewidth is small. However, most natural encodings that result in ASP programs are not treewidth-aware (Hecher 2022). Importantly, their work focused on theoretical treatment and, as such, does not address algorithmic aspects. It is worth noting that there is no accompanying implementation. Our personal communication with authors confirmed that they have not yet implemented their proposed technique.

6 Experimental Results

We developed a prototype of sharpASP-$\mathcal{SR}$,[4] by leveraging existing projected model counters. Specifically, we employed GANAK (Sharma *et al.* 2019) as the underlying projected model counter, given its competitive performance in model counting competitions. All counters are sourced from the model counting competition 2024.

6.1 Baseline and benchmarks

We evaluated sharpASP-$\mathcal{SR}$ against state-of-the-art ASP systems capable of handling disjunctive answer set programs: (i) clingo v5.7.1 (Gebser *et al.* 2012), (ii) DynASP v2.0 (Fichte *et al.* 2017), and (iii) Wasp v2 (Alviano *et al.* 2015). ASP solvers clingo and Wasp count answer sets via enumeration. We were unable to baseline against existing ASP counters such as aspmc+#SAT (Eiter *et al.* 2024), lp2sat+#SAT (Janhunen 2006; Janhunen and Niemelä 2011), sharpASP (Kabir *et al.* 2024), and iascar (Fichte *et al.* 2024), as these systems are designed exclusively for counting answer sets of normal logic programs. Since no implementation is available for the counting techniques outlined by Hecher and Kiesel (2023), a comparison against their approach was not possible. We also considered ApproxASP (Kabir *et al.* 2022) for comparison purposes and the result is provided in the extended version.

Our benchmark suite comprised non-tight disjunctive logic program instances previously used to evaluate disjunctive answer set solvers. These benchmarks span diverse computational problems, including (i) 2QBF (Kabir *et al.* 2022), (ii) strategic companies (Lierler 2005), (iii) *preferred* extensions of *abstract argumentation* (Gaggl *et al.* 2015), (iv) pc configuration (Fichte *et al.* 2022), (v) minimal diagnosis (Gebser *et al.* 2008), and (vi) *minimal trap spaces* (Trinh *et al.* 2024). The benchmarks were sourced from abstract argumentation competitions, ASP competitions (Gebser *et al.* 2020) and from Kabir *et al.* (2022) and Trinh *et al.* (2024). Following recent work on disjunctive logic programs (Alviano *et al.* 2019), we generated additional non-tight disjunctive answer set programs using the generator implemented by Amendola *et al.* (2017). The complete benchmark set comprises 1125 instances.

[4] https://github.com/meelgroup/SharpASP-SR

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

Table 1. *The performance of* sharpASP-$\mathcal{SR}$ *vis-a-vis existing disjunctive answer set counters, based on* 1125 *instances*

	clingo	DynASP	Wasp	sharpASP-$\mathcal{SR}$
#Solved (1125)	708	89	432	**825**
PAR2	4118	9212	6204	**2939**

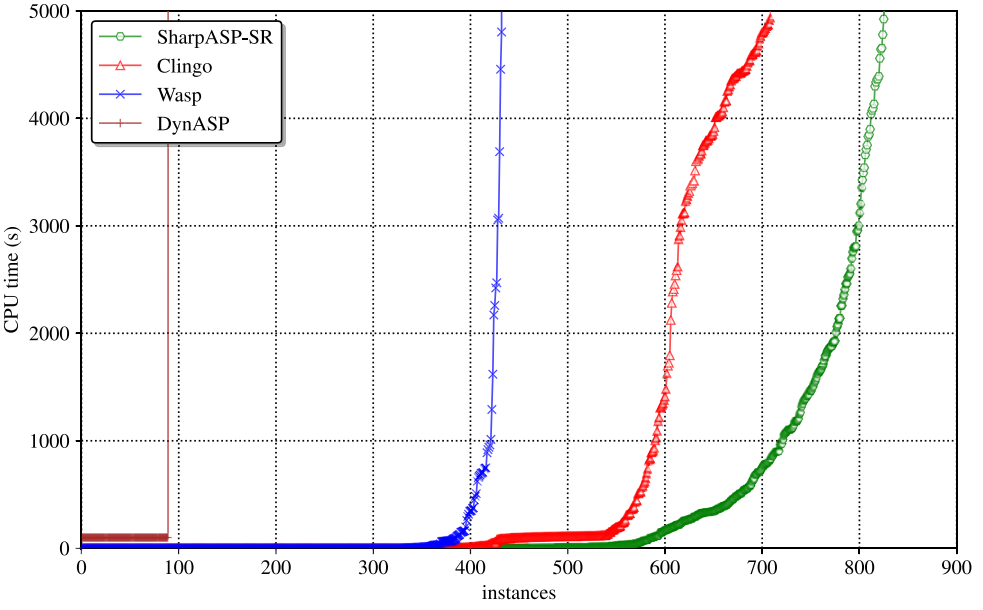

Fig. 2. The runtime performance of sharpASP-$\mathcal{SR}$ vis-a-vis other ASP counters.

6.2 Environmental settings

All experiments were conducted on a computing cluster equipped with AMD EPYC 7713 processors. Each benchmark instance was allocated one core, with runtime and memory limits set to 5000 seconds and 8 GB, respectively, for all tools, which is consistent with prior works on model counting and answer set counting.

6.1.1 Experimental results

sharpASP-$\mathcal{SR}$ demonstrated significant performance improvement across the benchmark suite, as evidenced in Table 1. For comparative analysis, we present both the number of solved instances and PAR2 scores (Balyo *et al.* 2017), for each tool. sharpASP-$\mathcal{SR}$ achieved the highest solution count while maintaining the lowest PAR2 score, indicating superior scalability compared to existing systems capable of counting answer sets of disjunctive logic programs. The comparative performance of different counters is shown in a cactus plot in Figure 2.

Given clingo's superior performance on instances with few answer sets, we developed a hybrid counter integrating the strengths of clingo's enumeration and other counting

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

Table 2. *The performance comparison of hybrid counters, based on* 1125 *instances. The hybrid counters correspond to last* 3 *columns that employ clingo enumeration followed by ASP counters. The clingo (*2*nd column) refers to clingo enumeration for* 5000 *seconds*

	clingo	DynASP	clingo ($\leq 10^4$) + Wasp	sharpASP-$\mathcal{SR}$
#Solved (1125)	708	377	442	**918**
PAR2	4118	4790	4404	**1600**

Table 3. *The performance comparison of* sharpASP-$\mathcal{SR}$ *(SA) vis-a-vis existing disjunctive answer set counters across instances with varying numbers of loop atoms. The second column ($\sum$) indicates the number of instances within each range of* $|\mathsf{LA}(P)|$

$\lvert\mathsf{LA}(P)\rvert$	$\sum$	clingo	DynASP	Wasp	sharpASP-$\mathcal{SR}$	clingo+sharpASP-$\mathcal{SR}$
$[1, 100]$	399	248	87	165	**386**	388
$[101, 1000]$	519	316	2	142	**398**	401
> 1000	207	**144**	0	125	41	129

techniques, following the experimental evaluation of Kabir *et al.* (2024). This hybrid approach first employs clingo enumeration (maximum 10^4 answer sets) and switches to alternative counting techniques if needed. Within our benchmark instances, a noticeable shift was observed on clingo's runtime performance when the number of answer sets exceeds 10^4. As shown in Table 2, the hybrid counter based on sharpASP-$\mathcal{SR}$ significantly outperforms baseline approaches.

The cactus plot in Figure 2 illustrates the runtime performance of the four tools, where a point (x, y) indicates that a tool can count x instances within y seconds. The plot shows sharpASP-$\mathcal{SR}$'s clear performance advantage over state-of-the-art answer set counters for disjunctive logic programs.

Since clingo and Wasp employ enumeration-based techniques, their performance is inherently constrained by the answer set count. Our analysis revealed that clingo (resp. Wasp) timed out on nearly all instances with approximately 2^{30} (resp. 2^{24}) or more answer sets, while sharpASP-$\mathcal{SR}$ can count instances upto 2^{127} answer sets. However, the performance of sharpASP-$\mathcal{SR}$ is primarily influenced by the hardness of the projected model counting, which is related to the cyclicity of the program. The cyclicity of a program is quantified using the measure $|\mathsf{LA}(P)|$.

To analyze sharpASP-$\mathcal{SR}$'s performance relative to $|\mathsf{LA}(P)|$, we compared different ASP counters across varying ranges of loop atoms. The results in the Table 3 indicate that while sharpASP-$\mathcal{SR}$ performs exceptionally well on instances with fewer loop atoms, its performance deteriorates significantly for instances with a higher number of loop atoms (e.g., those with $|\mathsf{LA}(P)| > 1000$), leading to a decrease in the solved instances count.

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

To further analyze the performance of sharpASP-$\mathcal{SR}$, we compare the number of answer sets for each instance solved by different ASP counters. Since both clingo and Wasp count using enumeration, clingo and Wasp can handle instances with up to 2^{30} and 2^{24} answer sets (roughly), respectively, whereas sharpASP-$\mathcal{SR}$ is capable of counting instances having 2^{127} answer sets. Due to its use of projected model counting, sharpASP-$\mathcal{SR}$ demonstrates superior scalability on instances with a large number of answer sets.
Further experimental evaluation of sharpASP-$\mathcal{SR}$ *are provided in the extended version.*

7 Conclusion

In this paper, we introduced sharpASP-$\mathcal{SR}$, a novel answer set counter based on subtractive reduction. By leveraging an alternative definition of answer sets, sharpASP-$\mathcal{SR}$ achieves significant performance improvements over baseline approaches, owing to its ability to rely on the scalability of state-of-the-art projected model counting techniques. Our experimental results demonstrate the effectiveness and efficiency of our approach across a range of benchmarks.

The use of subtractive reductions for empirical efficiency opens up potential avenues for future work. In particular, an interesting direction would be to categorize problems that can be reduced to #SAT via subtractive methods, which would allow us to utilize existing #SAT model counters.

References

Alviano, M., Amendola, G., Dodaro, C., Leone, N., Maratea, M. and Ricca, F. 2019. Evaluation of disjunctive programs in Wasp. In *LPNMR*, Springer, 241–255.

Alviano, M., Dodaro, C., Leone, N. and Ricca, F. 2015. Advances in Wasp. In LPNMR, Springer, 40–54.

Amendola, G., Ricca, F. And Truszczynski, M. 2017. Generating hard random boolean formulas and disjunctive logic programs. In *IJCAI*, 532–538.

Aziz, R. A., Chu, G., Muise, C. and Stuckey, P. 2015a. #∃SAT: Projected model counting. In SAT, Springer, 121–137.

Aziz, R. A., Chu, G., Muise, C. and Stuckey, P. J. 2015b. Stable model counting and its application in probabilistic logic programming. In *AAAI*.

Balyo, T., Heule, M. J. and Järvisalo, M. 2017. SAT competition 2017–solver and benchmark descriptions, pp. 14–15.

Ben-Eliyahu, R. and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence* 12, 53–87.

Ben-Eliyahu-Zohary, R., Angiulli, F., Fassetti, F. and Palopoli, L. 2017. Modular construction of minimal models. In *LPNMR*, Springer, 43–48.

Bendík, J. and Meel, K. S. 2021. Counting minimal unsatisfiable subsets. In *CAV*, Springer, 313–336.

Capelli, F., Lagniez, J.-M., Plank, A. and Seidl, M. 2024. A top-down tree model counter for quantified boolean formulas. In *IJCAI*.

Clark, K. L. 1978. Negation as failure. In *Logic and Data Bases*, 293–322.

Durand, A., Hermann, M. and Kolaitis, P. G. 2005. Subtractive reductions and complete problems for counting complexity classes. *Theoretical Computer Science* 340, 3, 496–513.

Egly, U., Eiter, T., Tompits, H. and Woltran, S. 2000. Solving advanced reasoning tasks using quantified boolean formulas. In *AAAI/IAAI*, 417–422.

Eiter, T., Fink, M., Tompits, H. and Woltran, S. 2004. On eliminating disjunctions in stable logic programming. In KR, 4, 447–458.

Eiter, T. and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence* 15, 289–323.

Eiter, T., Hecher, M. and Kiesel, R. 2021. Treewidth-aware cycle breaking for algebraic answer set counting. In KR, 269–279.

Eiter, T., Hecher, M. and Kiesel, R. 2024. New frontiers of algebraic answer set counting. *Artificial Intelligence* 330, 104109.

Fages, F. 1994. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science* 1, 1, 51–60.

Fandinno, J. and Schulz, C. 2019. Answering the "why" in answer set programming–a survey of explanation approaches. *Theory and Practice of Logic Programming* 19, 2, 114–203.

Fichte, J. K., Gaggl, S. A., Hecher, M. and Rusovac, D. 2024. IASCAR: Incremental answer set counting by anytime refinement. *Theory and Practice of Logic Programming* 24, 3, 505–532.

Fichte, J. K., Gaggl, S. A. and Rusovac, D. 2022. Rushing and strolling among answer sets–navigation made easy. In *AAAI*, Vol. 36, 5651–5659.

Fichte, J. K. and Hecher, M. (2019) Treewidth and counting projected answer sets. In *LPNMR*, Springer, 105–119.

Fichte, J. K., Hecher, M., Morak, M. and Woltran, S. 2017. Answer set solving with bounded treewidth revisited. In *LPNMR*, 132–145.

Fichte, J. K. and Szeider, S. 2015. Backdoors to normality for disjunctive logic programs. *ACM Transactions on Computational Logic* 17, 1, 1–23.

Gaggl, S. A., Manthey, N., Ronca, A., Wallner, J. P. and Woltran, S. 2015. Improved answer set programming encodings for abstract argumentation. *Theory and Practice of Logic Programming* 15, 4-5, 434–448.

Gebser, M., Kaufmann, B. and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence* 187, 52–89.

Gebser, M., Maratea, M. and Ricca, F. 2020. The seventh answer set programming competition: Design and results. *Theory and Practice of Logic Programming* 20, 2, 176–204.

Gebser, M., Schaub, T., Thiele, S., Usadel, B. and Veber, P. 2008. Detecting inconsistencies in large biological networks with answer set programming. In *ICLP*, Springer, 130–144.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In ICLP/SLP, Vol. 88, 1070–1080.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9, 365–385.

Hecher, M. 2022. Treewidth-aware reductions of normal ASP to SAT – Is normal ASP harder than SAT after all? *Artificial Intelligence* 304, 103651.

Hecher, M. and Kiesel, R. 2023. The impact of structure in answer set counting: fighting cycles and its limits. In KR, 344–354.

Hemaspaandra, L. A. and Vollmer, H. 1995. The satanic notations: counting classes beyond #P and other definitional adventures. *ACM SIGACT News* 26, 1, 2–13.

Janhunen, T. 2006. Some (in) translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics* 16, 1-2, 35–86.

Janhunen, T. and Niemelä, I. 2011. *Compact Translations of Non-disjunctive Answer Set Programs to Propositional Clauses*, 111–130.

Janhunen, T., Niemelä, I., Seipel, D., Simons, P. and You, J.-H. 2006. Unfolding partiality and disjunctions in stable model semantics. *ACM Transactions on Computational Logic* 7, 1, 1–37.

Ji, J., Wan, H., Wang, K., Wang, Z., Zhang, C. and Xu, J. 2016. Eliminating disjunctions in answer set programming by restricted unfolding. In IJCAI, 1130–1137.

Kabir, M., Chakraborty, S. and Meel, K. S. 2024. Exact ASP counting with compact encodings. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 10571–10580.

Kabir, M., Everardo, F. O., Shukla, A. K., Hecher, M., Fichte, J. K. and Meel, K. S. 2022. ApproxASP–a scalable approximate answer set counter. *Proceedings of the AAAI Conference on Artificial Intelligence* 36, 5755–5764.

Kabir, M. and Meel, K. S. 2023. A fast and accurate ASP counting based network reliability estimator. In *LPAR*, 270–287.

Kabir, M. and Meel, K. S. 2024. On lower bounding minimal model count. *Theory and Practice of Logic Programming* 24, 4, 586–605.

Kabir, M. and Meel, K. S. 2025. An ASP-based framework for MUSes, arXiv preprint arXiv: 2507.03929

Kabir, M. 2024. Minimal model counting via knowledge compilation, arXiv preprint arXiv: 2409.10170

Kabir, M., Trinh, V.-G., Pastva, S. and Meel, K. S. 2025. Scalable counting of minimal trap spaces and fixed points in boolean networks, arXiv preprint arXiv: 2506.06013

Kanchanasut, K. and Stuckey, P. J. 1992. Transforming normal logic programs to constraint logic programs. *Theoretical Computer Science* 105, 1, 27–56.

Lee, J. and Lifschitz, V. 2003. Loop formulas for disjunctive logic programs, *ICLP* 2003, Springer, 451–465.

Lee, J., Talsania, S. and Wang, Y. 2017. Computing LPMLN using ASP and MLN solvers. *Theory and Practice of Logic Programming* 17, 5-6, 942–960.

Leone, N. and Ricca, F. 2015. Answer set programming: A tour from the basics to advanced development tools and industrial applications. In *Reasoning Web International Summer School*, Springer, 308–326.

Lierler, Y. 2005. Cmodels–SAT-based disjunctive answer set solver. In LPNMR, Springer, 447–451.

Lifschitz, V. 2010. Thirteen definitions of a stable model. In *Fields of Logic and Computation*, 488–503.

Lifschitz, V. and Razborov, A. 2006. Why are there so many loop formulas? *ACM Transactions on Computational Logic* 7, 2, 261–268.

Marek, V. W. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm*, Springer, 375–398.

Marek, W. and Truszczyński, M. 1991. Autoepistemic logic. *Journal of the ACM (JACM)* 38, 3, 587–618.

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R. and Barry, M. 2001. An A-Prolog decision support system for the space shuttle. In PADL, Springer, 169–183.

Pontelli, E., Son, T. C. and Elkhatib, O. 2009. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming* 9, 1, 1–56.

Rusovac, D., Hecher, M., Gebser, M., Gaggl, S. A. and Fichte, J. K. 2024. Navigating and querying answer sets: how hard is it really and why? In *KR*, Vol. 21, 642–653.

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press

Sharma, S., Roy, S., Soos, M. and Meel, K. S. 2019. GANAK: A scalable probabilistic exact model counter. In *IJCAI*, Vol. 19, 1169–1176.

Shukla, A., Möhle, S., Kauers, M. and Seidl, M. 2022. Outercount: A first-level solution-counter for quantified boolean formulas. In *CICM*, Springer, 272–284.

Trinh, G., Benhamou, B., Pastva, S. and Soliman, S. 2024. Scalable enumeration of trap spaces in boolean networks via answer set programming. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 10714–10722.

Valiant, L. G. 1979. The complexity of enumeration and reliability problems. *SIAM Journal on Computing* 8, 3, 410–421.

Zhou, Y. (2014) From disjunctive to normal logic programs via unfolding and shifting. In *ECAI* 2014, IOS Press, 1139–1140.

https://doi.org/10.1017/S1471068425100124 Published online by Cambridge University Press