ANSWER SET COUNTING AND ITS APPLICATIONS

by

MD MOHIMENUL KABIR

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2025

Supervisor:

Associate Professor Kuldeep S. Meel, Main Advisor
Assistant Professor Diptarka Chakraborty, Co-Advisor

Examiners:

Assistant Professor Umang Mathur
Associate Professor Roland Yap Hock Chuan

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Md Mohimenul Kabir

April 28, 2026

To my parents and my family

Acknowledgments

First and foremost, I am deeply grateful to the Almighty for granting me the strength, patience, and perseverance to undertake and complete this journey. Through moments of clarity and periods of doubt, it is by divine guidance and grace that I have been able to reach this milestone. This work is a testament to that unwavering support.

I would like to express my heartfelt gratitude to my advisor, Kuldeep S. Meel, for his unwavering support, exceptional guidance, and mentorship throughout my PhD journey. From the very beginning, he played an instrumental role in shaping my research direction, instilling in me the values of clarity, rigor, and ambition. His sharp insights, high standards, and constant encouragement pushed me to grow as a researcher and to strive for excellence. Beyond research, his generous support extended to every aspect of my academic and professional development, and I am truly fortunate to have worked under his supervision.

I would also like to thank Diptarka Chakraborty for stepping in to assist with administrative matters during the final phase of my PhD, especially after Kuldeep's departure from NUS. His help with institutional processes ensured a smooth transition and timely completion of my degree requirements, for which I am sincerely grateful.

I am also grateful to the members of my thesis committee for their valuable time and constructive comments. Their suggestions and critiques have helped improve the clarity and rigor of my work.

I am especially grateful to my collaborators — Supratik Chakraborty, Johannes K. Fichte, Markus Hecker, Ankit Shukla, Flavio Everardo, Van-Giang Trinh, and Samuel Pastva — for their insightful discussions, technical contributions, and shared enthusiasm for research. Working with each of them has been intellectually enriching, and their perspectives have significantly influenced the ideas and results presented in this thesis. I sincerely appreciate their generosity with time, ideas, and feedback throughout our collaborations.

I extend my appreciation to the Meel Group for providing a vibrant and intellectually stimulating research environment. The discussions, brainstorming sessions, and collective enthusiasm for solving hard problems have been a constant source of

motivation. I am especially thankful to Bishwa, Priyanka, Teodora, Yash, Suwei, Mate, Jiong, Pang, Anna, Tim, Izza, Yifan, Jack, Ashwin, Arijit, Uddalok, Tianguang, Paulius, Aashiq, Razo, Tareq, Neamul, Rafi, Rafeed, Haroon, Najeeb, Kaasif, Nasir, and Zubair for their support and friendship throughout this journey.

I would also like to thank the School of Computing, National University of Singapore, for the academic support, facilities, and funding that enabled me to carry out my research. I am particularly thankful for access to computing resources such as Aspire2A and Niagara, which were essential for many of the experiments conducted in this thesis.

To my friends and peers, thank you for making this journey less solitary and more rewarding. The shared moments of stress, laughter, and late-night debugging have left lasting memories.

Lastly, I am deeply indebted to my family and relatives for their unconditional love, patience, and belief in me. Their support has been a constant source of strength, and this accomplishment would not have been possible without them.

Md Mohimenul Kabir

Contents

List of Algorithms

List of Figures

List of Tables

Publications during PhD

1. Mohimenul Kabir, 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(5), 5755-5764.

2. Mohimenul Kabir, and Kuldeep S. Meel. (2023) A Fast and Accurate ASP Counting Based Network Reliability Estimator. In *LPAR*, pp. 270-287. 2023.

3. Mohimenul Kabir, Supratik Chakraborty, and Kuldeep S. Meel. (2024) Exact ASP Counting with Compact Encodings. Proceedings of the *AAAI Conference on Artificial Intelligence*, vol. 38, no. 9, pp. 10571-10580. 2024.

4. Mohimenul Kabir, and Kuldeep S. Meel. (2024) On Lower Bounding Minimal Model Count. *Theory and Practice of Logic Programming* 24, no. 4, 2024: 586-605.

5. Mohimenul Kabir, Supratik Chakraborty, and Kuldeep S. Meel. (2025) Counting Answer Sets of Disjunctive Logic Programs. *ICLP 2025*.

6. Mohimenul Kabir, Van-Giang Trinh, Samuel Pastva, and Kuldeep S. Meel. (2025) Scalable Counting of Minimal Trap Spaces and Fixed Points in Boolean Networks. *CP 2025*.

List of Released Tools

- ApproxASP [KES+22]
  - Source code: https://github.com/meelgroup/ApproxASP
  - Benchmarks: https://zenodo.org/records/19665703
- RelNet-ASP [KM23]

  - Source code: https://github.com/meelgroup/RelNet-ASP
  - Benchmarks: https://zenodo.org/records/19664400
- SharpASP [KCM24]
  - Source code: https://github.com/meelgroup/sharpASP
  - Benchmarks: https://zenodo.org/records/19665132
- MinLB [KM24]
  - Source code: https://github.com/meelgroup/MinLB
  - Benchmarks: https://zenodo.org/records/19757334
- SharpASP-SR [KCM25]
  - Source code: https://github.com/meelgroup/sharpASP-sr
  - Benchmarks: https://zenodo.org/records/19665845
- BN-counting [KTP+25]
  - Source code: https://github.com/meelgroup/bn-counting
  - Benchmarks: https://zenodo.org/records/19665913

Abstract

Answer Set Counting and its Applications

by

Md Mohimenul Kabir

Doctor of Philosophy in Computer Science

National University of Singapore

Answer Set Programming (ASP) has emerged as a promising paradigm in knowledge representation and automated reasoning owing to its ability to model hard combinatorial problems from diverse domains in a natural way. Building on advances in propositional SAT solving, the past two decades have witnessed the emergence of well-engineered systems for solving the answer set satisfiability problem, i.e., finding models or answer sets for a given answer set program. In recent years, there has been growing interest in problems beyond satisfiability, such as model counting, in the context of ASP. Importantly, a variety of probabilistic reasoning and probabilistic logic programming require counting answer sets. Akin to the early days of propositional model counting, state-of-the-art answer set counters do not scale well beyond small instances. While counting can be done by enumeration, simple enumeration becomes infeasible if the answer set count is high.

In this thesis, we propose efficient counting techniques for answer sets, overcoming the scalability issues faced by existing answer set systems. We propose two new ASP counting frameworks, called (i) sharpASP and (ii) sharpASP-$\mathcal{SR}$. Both frameworks count answer sets without translating the full problem into substantially larger propositional formulas. These techniques rely on an alternative way of defining answer sets. For sharpASP, these alternative definition allows for the lifting of key techniques developed in the context of propositional model counting to answer set counting; while for sharpASP-$\mathcal{SR}$, the definition facilitates an efficient reduction of answer set counting into projected model counting.

We present a scalable approach to approximate counting for ASP. Our approach is based on systematically adding parity constraints to ASP programs, which divide the search space. We prove that adding random parity constraints partitions the

answer sets of an ASP program. In practice, we use a Gaussian elimination-based approach by lifting ideas from SAT to ASP and integrate it into a state-of-the-art ASP solver, which we call ApproxASP. Finally, our experimental evaluation shows the scalability of our approach over existing ASP systems.

The thesis also demonstrates the applicability of answer set counting in several domains, including network reliability, systems biology, and minimal model counting. We propose RelNet-ASP, an ASP-based framework for approximating the reliability of systems by reducing reliability estimation to approximate answer set counting with formal guarantees. We also formulate several meaningful counting problems for Boolean Networks, a fundamental modeling framework for complex dynamical systems, and develop approximate counting techniques for these problems using ASP. Finally, we study the problem of counting minimal models of Boolean formulas and introduce two ASP-based techniques: one based on knowledge compilation and another based on hashing-based approximate counting. Across these applications, our empirical results show that the proposed ASP-counting-based techniques improve the scalability, runtime performance, and accuracy of existing state-of-the-art methods.

Chapter 1

Introduction

Answer Set Programming (ASP) [165] is a declarative problem-solving approach with a wide variety of applications ranging from decision support systems [173], exploration of feasible logic models in system biology [107], recommendation in e-tourism [119], diagnosis in dynamic remarketing ads [39], production configuration support tool [199], nurse scheduling [57], workforce management, and customer categorization in call center [153]. An ASP program consists of a set of *rules*, following the syntax illustrated in Listing 1.1. Each rule has a *body*, which is either true or a conjunction of literals, and a *head*, which is either false or a disjunction of atoms. Together, the body and the head express an implication relation. Under the answer set semantics [98], the rules of an ASP program encode domain knowledge in a natural way. An assignment to the propositional atoms that satisfies the answer set semantics is called an *answer set*.

ASP provides a powerful rule-based modelling language, featuring *variables*, *functions*, *recursions*, *aggregates*, and more [12, 95]. To evaluate an ASP program, existing ASP systems follow a two-step procedure: *grounding* [68, 89] and *solving* [11, 93]. During the *grounding* step, also known as *instantiation*, the program's variables are systematically replaced, yielding a *propositional-like* program with *negations*. This transformed program is free of variables while preserving the same answer sets as the original. This transformed version of the program is often known as the *grounded program*. After grounding, ASP solvers process the grounded program to compute the answer sets. The grounding is a crucial preprocessing step in efficient answer set solving [36, 52, 141]; since a naive grounding can be computationally expensive. However, significant advances have been made in grounding logic programs [19, 54, 174, 213]. Similar to the ASP literature, this thesis assumes that the programs are already grounded and presents our contributions on grounded ASP programs.

1 $\underbrace{a_1 \vee \ldots a_k}_{\text{Head}} \leftarrow \underbrace{b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n}_{\text{Body}}.$

Listing 1.1: A basic/standard ASP rule

While the most common/standard form of a rule is illustrated in Listing 1.1, there are other syntactic forms of ASP rules, such as the *choice* rule, *cardinality* rule, *weight* rule, *aggregate*, *weak constraint*, and *optimization* statements [42, 172]. It is well known that most extended rule forms can be translated into the common form efficiently [32, 33, 34, 35]. Following the approach in ASP literature, we formalize our contributions using the standard rule syntax, as demonstrated in Listing 1.1. To illustrate, we present an ASP encoding for the *graph reachability* problem [209] in Listing 1.2.

Given a graph G, the graph reachability problem seeks to know whether two nodes (referred to as source and destination nodes) of interest in G are reachable or not; that is, whether there exists a path from the source node to the destination node in G. The encoding consecutively examines subgraphs of graph G and uses the *predicate* $\mathsf{in}(\mathsf{X}, \mathsf{Y})$ to indicate if the edge (X, Y) belongs to a subgraph. Additionally, the encoding uses the predicate $\mathsf{reached}(\mathsf{X})$ to denote that the node X is reachable from the source node. Two other predicates, $\mathsf{source}(\mathsf{X})$ and $\mathsf{dest}(\mathsf{X})$, specify the source and destination nodes, respectively. The encoding in Listing 1.2 can be interpreted as follows: first it designates the source node as trivially reachable (line 2 of Listing 1.2), then transitively computes the reachable nodes from the source node (line 4 of Listing 1.2), and finally the last line (line 6 of Listing 1.2) enforces that the destination node must be reachable. If there is an answer set of the program in Listing 1.2, then the answer set corresponds to a subgraph that has a path from the source node to the destination node. Otherwise, there is no path in G from the source node to the destination node.

ASP programs can be categorized based on their expressiveness into two types: *disjunctive* and *normal* ASP programs. An ASP program is called disjunctive if at least one rule has a head containing two or more atoms; otherwise, the program is called normal. It is well-established that disjunctive programs are more expressive than normal ones [79, 152]. However, the expressiveness comes with

```
% source node is trivially reachable
reached(X) ← source(X).
% transitive definition of reachability
reached(Y) ← in(X,Y), reached(X).
% target node must be reachable
← dest(X), not reached(X).
```

Listing 1.2: Graph Reachability Problem

higher computational complexity. Specifically, determining whether a disjunctive program has an answer set is $\sum_2^P$-complete [64], while for normal programs, the complexity is reduced to NP-complete [166].

Although ASP programs are syntactically similar to prolog programs [212], ASP shares computational similarities with propositional satisfiability [101, 157]. Like satisfiability, an ASP program encodes the target problem specification and the goal is to compute an answer set of the program, which satisfies the problem specification (the direction is different from Prolog; the goal in Prolog is to evaluate whether a given *query* holds within a program). The answer sets of ASP programs are analogous to satisfying assignments (or models) of propositional formulas. Similarly, answer set programming can be viewed as a constraint programming paradigm, much like propositional satisfiability [157]. Furthermore, both ASP programs and propositional formulas are solved using techniques based on *conflict-driven clause learning* (CDCL) [167, 168, 217].

In general, given a set of constraints in a theory, model counting seeks to determine the number of models (or solutions) to the set of constraints. From a computational complexity perspective, this problem is significantly harder than deciding whether there exists a solution to the set of constraints, i.e. the satisfiability problem. Yet, in the context of propositional reasoning, compelling applications have driven substantial practical advancements in propositional model counting, also referred to as #SAT [188, 198], over the past decade. This, in turn, has ushered in new applications in quantified information flow [30], neural network verification [20], network reliability [59], probabilistic inference [182, 186], control improvisation [100] and the like. The success of practical propositional model counting in diverse domains has naturally led researchers to ask if practically efficient counting algorithms can

```
1  % a friend influences another friend
2  influence(P1, P2) ← friend(P1, P2).
3  % smoking due to being stressed
4  smokes(P) ← stress(P).
5  % smoking due to influenced by smoker friend
6  smokes(P2) ← smokes(P1), influence(P1,P2).
7  % given evidance
8  stress(a).
9  % given query
10 :- not smokes(b).
```

Listing 1.3: Smokers Problem

be developed for constraints beyond propositional logic.

The problem of counting answer sets to a given ASP program, known as #ASP, has garnered growing interest, motivated by its applications in probabilistic reasoning, navigation, plausibility reasoning, network reliability, and systems biology [78, 80, 83, 134, 136, 137, 138]. The computational complexity of #ASP for disjunctive programs is $\# \cdot$coNP-complete [82], which is #P-complete [125, 126] if the program is *normal*. We provide two examples to illustrate the answer set counting problem.

The first example is the *network reliability problem* [59, 134], which is the *quantitative* version of the graph reachability problem, as outlined in Listing 1.2. The network reliability problem seeks to determine the probability of two nodes being connected, i.e., the probability of a path existing from the source node to the destination node in graph G. The probability can be computed as the fraction of the number of subgraphs containing a path from the source to the destination to the total number of all subgraphs of G. While counting all subgraphs is trivial, answer set counting can be used to count only those subgraphs where a path exists from the source to the destination (the framework RelNet-ASP [134] presented a detailed analysis of the problem). Hence, answer set counting allows efficient computation of network reliability.

The second example is the *smokers* problem [56, 83]; the basic encoding is presented in Listing 1.3. In the smokers problem, a person may start smoking in two ways: (i) due to being stressed (Line 4 in Listing 1.3) or (ii) being influenced by a smoker (Line 6 in Listing 1.3). The smoker problem aims to compute the probability of an event (e.g. the person b is smoking), given some evidences (e.g. the person b

is stressed). This probability computation can be reduced to counting answer sets of an ASP program [83].

Early efforts to build answer set counters sought to work by enumerating answer sets of a given ASP program [82, 91]. While this works extremely well for answer set counts upto a certain threshold, enumeration cannot scale well for problem instances with too many answer sets. Therefore, subsequent approaches to answer set counting sought to leverage the significant progress made in #SAT techniques. Specifically, Aziz et al. [17] integrated a *component-caching* based propositional model counting technique with *unfounded set detection* to yield an answer set counter, called ASProblog. In another line of work, dynamic programming on a *tree decomposition* of the input problem instance has been proposed to achieve scalability for ASP instances with *low treewidth* [75, 82]. Yet another approach has been to translate a given normal logic program P into a propositional formula F, such that there is a one-to-one correspondence between answer sets of P and models of F [32, 66, 126, 127]. The answer sets of P can then be counted by invoking an off-the-shelf propositional model counter [188] on F. Though promising in principle, a naive application of this approach does not scale well in practice owing to a blowup in the size of the resulting formula F when the implications between propositional atoms encoded in the program P give rise to circular dependencies [161], which is a common occurrence when modeling numerous real-world applications. To address this, researchers have proposed techniques: e.g., *level numbering* [125], *level ranking* [127], *unfolding* [65], *mixed integer programming* [164], and enforcing *acyclicity* [86]. Thus, despite significant advances, state-of-the-art exact answer set counters are stymied by scalability bottlenecks, limiting their practical applicability.

Most existing answer set counters focus on normal logic programs - a restricted class of ASP. The thesis also targets the more expressive disjunctive logic programs [64]. The practical deployment of counters for disjunctive programs have received limited attention, creating a significant gap in the literature. This focus is well-motivated: complexity theory indicates that barring a collapse of the polynomial hierarchy, translation from disjunctive to normal programs must incur exponential overhead [63, 218]. Consequently, existing counters optimized for normal programs cannot efficiently handle disjunctive programs, unless *special properties* present [26, 79, 130]. While loop formula-based translation [151] theoretically enables counting,

the exponential overhead becomes prohibitive for programs with cyclic atom relationships [161]. Although disjunctive answer sets counting can be theoretically reduced to QBF counting [62], the practical implementation of efficient QBF counters is still in its nascent stages [44, 190].

Due to the higher computational complexity of exact model counting [209], the applicability of model counting is often limited within small to medium instances. In this context, the approximate model counting [47, 48] has shown to be successful in recent model counting competitions [76, 77]. Of particular interesting to us are *hashing-based frameworks* developed in the context of approximate model counting. The core idea is hashing-based frameworks, which partition the solution space or answer sets into *roughly equal small* cells of answer sets by employing *pairwise independent hash* functions [45] using XOR-constraints, and then the count is obtained by enumerating solutions or answer sets within one of the randomly chosen *cells*. The complexity of approximate model counting is known to be BPP^{NP}; in fact the approximate model counter calls a *spacialized oracle* polynomially many times [195]. While approximate model counting achieves scalability at the cost of accuracy, such inaccuracy is often acceptable in many real-world applications [48].

1.1 Thesis Contribution

The thesis contributions are going to be discussed in two directions. First, we present efficient and scalable answer set counting techniques — these counting techniques include both exact and approximate methods. These techniques address the scalability challenges faced by off-the-shelf counting techniques. The contributions from the development of these efficient answer set counters are as follows:

Contribution 1: sharpASP We present an alternative approach to exact answer set counting, called sharpASP, while alleviating a key bottleneck related to the *size* of CNF translation faced by earlier approaches. While a mere reduction in translation size does not inherently establish a scalable ASP counting technique for general scenarios, sharpASP enables the solving of larger and more instances of exact answer set counting than previously possible. sharpASP lifts component-caching based propositional model counting algorithms to ASP counting. The key

idea that facilitates this adaptation is an alternative yet correlated perspective on defining answer sets. This redefinition enables the application of concepts such as *decomposability* and *determinism* from propositional model counting to answer set counting. Our experimental analysis demonstrates that sharpASP, built using this approach, significantly outperforms the performance of previous state-of-the-art techniques across instances from diverse domains, underscoring the effectiveness of our approach over previous exact answer set counters.

Contribution 2: sharpASP-$\mathcal{SR}$ We present the design, implementation, and extensive evaluation of a novel counter for disjunctive programs, employing *subtractive reduction* [61], by transforming the problem into projected propositional model counting [16], while maintaining polynomial formula size growth. The approach first computes an overcount of the answer set count, then precisely subtracts the surplus using projected counting. This yields a #NP algorithm that leverages recent advances in projected model counting [150, 188]. This approach is both theoretically justified and provides a practical counting algorithm for disjunctive logic programs: since $\# \cdot \text{co-NP} = \# \cdot \text{P}^{\text{NP}} = \#\text{NP}$ [61, 113] (answer set counting for disjunctive programs is in $\# \cdot$ co-NP [82]). Our counter, sharpASP-$\mathcal{SR}$, employs an alternative definition of answer sets for disjunctive programs, extending the idea of sharpASP on normal programs [132]. This definition enables the use of off-the-shelf projected model counters without exponential formula growth.

Contribution 3: ApproxASP We present ApproxASP, the first scalable technique for #ASP that provides rigorous (ε, δ) guarantees. From the technical perspective, we lift the XOR-based hashing framework developed in the context of propositional model counting to ASP. As demonstrated in the development of ApproxMC [47], designing a scalable counter requires enhancements to the underlying solver to support XOR constraints. To this end, we present the first ASP solver that can natively handle XOR constraints via Gauss-Jordan elimination (GJE).

In the second part of this thesis, we demonstrate real-world applications of answer set counting. Specifically, we demonstrate how our proposed answer set counters provide efficient and effective solutions in the domains of network reliability, systems biology, and minimal model reasoning.

Contribution 4: RelNet-ASP We propose a framework, called RelNet-ASP, that reduces the problem of network reliability to answer set counting. The framework RelNet-ASP addresses more general network reliability scenarios, where each edge is active with a predefined probability and incorporates theories from weighted model counting [46] to model probability associated with each edge. Our empirical evaluation demonstrates that RelNet-ASP significantly outperforms prior state-of-the-art approaches when accounting for both accuracy and runtime performance.

Contribution 5: Counting on Boolean Networks *Boolean Networks* is a fundamental modeling framework for representing complex dynamical behaviour of system biology [187, 205]. We formulated meaningful counting and projected counting problems on BN, which aim to quantify core and meaningful concepts of BNs (e.g., *minimal trap space* and *fixed point*), while satisfying certain *properties*. More importantly, these counting problems have significance in studying system's *stability*, *attractor* structure, and probabilistic behaviour [115, 144]. We propose novel and efficient ASP-based methods to address those counting problems. We conducted an extensive experimental evaluation on a diverse set of real-world BNs. Our analysis demonstrates that ApproxASP [133] efficiently estimates the number of minimal trap spaces and fixed points. Importantly, ApproxASP overcomes exhaustive enumerations, significantly improving the feasibility of counting compared to enumeration and BDD-based methods used in off-the-shelf tools.

Contribution 6: MinLB We extended answer set counting principles to count solutions in other theories that are semantically similar to ASP. We developed methods for estimating a lower bound on the number of *minimal models* of a given propositional formula. More specifically, the minimal models are subset-minimal (with respect to $\subseteq$) models of a Boolean formula [14]. This is achieved by integrating *knowledge compilation* and *hashing-based techniques* with minimal model reasoning, thereby facilitating the estimation of lower bounds. The effectiveness of our proposed methods has been empirically validated on instances from model counting competitions and itemset mining. To assess the performance of our proposed methods, we introduce a new metric that considers both the quality of the lower bound and the computational time; our methods achieve the best score compared

to existing minimal model reasoning systems.

1.2 Thesis Organization

The thesis is organized as follows: Chapter 2 provides the background knowledge necessary to understand the technical contribution of the thesis. In Chapters 3 and 4, we present our exact answer set counters sharpASP and sharpASP-$\mathcal{SR}$, respectively. Chapter 5 describes our approximate answer set counter ApproxASP. We then explore applications of our proposed answer set counters. Chapter 6 demonstrates how RelNet-ASP efficiently addresses the network reliability problem invoking an answer set counter. Chapter 7 demonstrates how ASP counters can be utilized to address counting problems over Boolean Networks. Chapter 8 presents how existing answer set systems can be utilized to obtain lower bounds on minimal model counts. Finally, we conclude our thesis in Chapter 9.

Chapter 2

Preliminaries

This chapter introduces the necessary notations and preliminaries needed to understand the technical contribution.

2.1 Propositional Satisfiability

A propositional *variable* v takes one of two values: 0 (denoting false) or 1 (denoting true). A *literal* ℓ is either a variable (positive literal) or its negation (negated literal), and a *clause* C is a disjunction of literals. For convenience of exposition, we sometimes represent a clause as a set of literals, with the implicit understanding that all literals in the set are disjoined in the clause. A clause with a single literal is also called a *unit clause*. In general, the constraint represented by a clause $C \equiv (\neg v_1 \vee \ldots \vee \neg v_k \vee v_{k+1} \vee \ldots \vee v_{k+m})$ can be expressed as a logical *implication*: $(v_1 \wedge \ldots \wedge v_k) \rightarrow (v_{k+1} \vee \ldots \vee v_{k+m})$. If $k = 0$, the antecedent of the above implication is true, and if $m = 0$, the consequent is false. A *conjunctive normal form (CNF)* formula ϕ is a conjuction of clauses. When there is no confusion, a CNF formula is also sometimes represented as a set of clauses, with the implicit understanding that all clauses in the set are conjoined to give the formula. We denote the set of variables in ϕ as $\mathsf{Var}(\phi)$.

An assignment over a set X of propositional variables is a mapping $\tau : X \rightarrow \{0, 1\}$. 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).

We often consider an assignment τ as a set of literals it assigns and $\mathsf{Var}(\tau)$ denotes the set of variables assigned by τ. For two assignments τ_1 and τ_2, τ_1 satisfies τ_2, denoted as $\tau_1 \models \tau_2$, if $\tau_{1\downarrow \mathsf{Var}(\tau_2)} = \tau_2$, the notation $\tau_{\downarrow X}$ denotes the *projection* of τ onto variable set X. Otherwise, τ_1 does not satisfy τ_2, denoted as $\tau_1 \not\models \tau_2$.

2.1.1 Unit Propagation

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* $\{\neg\ell\}$, i.e. 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 $\phi|_\tau$, i.e. if $\{\ell\} \in \phi|_\tau$.

2.1.2 Model Counting Notations

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 $\#\exists X\phi$.

2.1.3 Logical Equivalence

Two clauses C_i and $C_{i'}$ are *logically equivalent*, denoted as $C_i \leftrightarrow C_{i'}$, if C_i and $C_{i'}$ have the same truth value for all assignments over $\mathsf{atoms}(C_i \wedge C_{i'})$. The logical relation between clauses C_i and $C_{i'}$ is known as *equivalence* and for notational convenience, we denote an equivalence $C_i \leftrightarrow C_{i'}$ as a tuple of C_i and $C_{i'}$. Given an equivalence $C_i \leftrightarrow C_{i'}$, if C_i ($C_{i'}$ resp.) consists of only one literal, then the atoms of $C_{i'}$ (C_i resp.) *define* the truth value of C_i ($C_{i'}$ resp.).

2.1.4 xor Constraints

An XOR constraint [104] or parity constraint over $\mathsf{Var}(\phi)$ is a Boolean "XOR" ($\oplus$) applied to the variables $\mathsf{Var}(\phi)$. A random XOR constraint over variables $\{x_1, \ldots, x_k\}$ is expressed as $a_1 \cdot x_1 \oplus \ldots a_k \cdot x_k \oplus b$, where all a_i and b follow the *Bernoulli* distribution with a probability of $1/2$. An XOR constraint $x_{i_1} \oplus \ldots x_{i_k} \oplus 1$ (or $x_{i_1} \oplus \ldots x_{i_k} \oplus 0$ resp.) is evaluated as true if an even (or odd resp.) number of variables from $\{x_{i_1}, \ldots, x_{i_k}\}$ are assigned to true.

2.1.5 Minimal Models

To define minimal models of a propositional formula ϕ, we introduce an *ordering operator* over models [129]. For two given models τ_1 and τ_2, τ_1 is considered *smaller* than τ_2, denoted as $\tau_1 \leq \tau_2$, if and only if for each $x \in \mathsf{Var}(\phi)$, $\tau_1(x) \leq \tau_2(x)$. We define τ_1 as *strictly smaller* than τ_2, denoted as $\tau_1 < \tau_2$, if $\tau_1 \leq \tau_2$ and $\tau_1 \neq \tau_2$. A model τ is a *minimal model* of ϕ if and only if τ is a model of ϕ and no model of ϕ is strictly smaller than τ. We use the notation $\mathsf{MinModels}(\phi)$ to denote minimal models of ϕ and for a set $X \subseteq \mathsf{Var}(\phi)$, $\mathsf{MinModels}(\phi)_{\downarrow X}$ denotes the minimal models of ϕ projected onto the variable set X. The minimal model counting problem seeks to determine the cardinality of $\mathsf{MinModels}(\phi)$, denoted $|\mathsf{MinModels}(\phi)|$.

We sometimes represent minimal models by listing the variables assigned as true. For example, suppose $\mathsf{Var}(\phi) = \{a, b, c\}$ and under minimal model $\tau = \{a, b\}$, $\tau(a) = \tau(b) = \mathsf{true}$ and $\tau(c) = \mathsf{false}$. The notation $\neg\tau$ denotes the negation of assignment τ; in fact, $\neg\tau$ is a clause or disjunction of literals (e.g., when $\tau = \{a, b\}$, $\neg\tau = \neg a \vee \neg b$). For each model $\sigma \in \mathsf{MinModels}(\phi)$, each of the variables assigned to true is *justified*; more specifically, for every literal $\ell \in \sigma$, there exists a clause $c \in \phi$ such that $\sigma \setminus \{\ell\} \not\models c$. Otherwise, $\sigma \setminus \{\ell\} \models \phi$ (which is $< \sigma$).

2.2 Answer Set Programming

An answer set program P expresses logical constraints between a set of propositional variables. In the context of answer set programming, such variables are also called *atoms*, and the set of atoms appearing in P is denoted $\mathsf{atoms}(P)$. For notational convenience, we will henceforth use the terms "variable" and "atom"

interchangeably. An *answer set program* is a set of rules of the following form:

$$\text{Rule } r\text{:}\ \ a_1 \vee \ldots a_k \leftarrow b_1, \ldots, b_m, \sim c_1, \ldots, \sim c_n \tag{2.1}$$

In the above rule, $\sim$ denotes *default negation*, signifying *negation as failure* [51]. For rule r shown above, the atom set $\{a_1, \ldots a_k\}$ is called the *head* of rule r and is denoted $\mathsf{Head}(r)$. Similarly, the set of literals $\{b_1, \ldots, b_m, \sim c_1, \ldots, \sim c_n\}$ is called the *body* of rule r. Specifically, $\{b_1, \ldots, b_m\}$ are the *positive body atoms*, denoted $\mathsf{Body}(r)^+$, and $\{c_1, \ldots, c_n\}$ are the *negative body atoms*, denoted $\mathsf{Body}(r)^-$. For purposes of the following discussion, we use $\mathsf{Body}(r)$ to denote the conjunction $b_1 \wedge \ldots \wedge b_m \wedge \neg c_1 \wedge \ldots \wedge \neg c_n$. Atoms that appear in the head of a rule (like $\{a_1, \ldots a_k\}$ in rule r above) have also been called *founded variables/atoms* in the literature [17]. A program P is called a *disjunctive logic program* if there exists a rule $r \in P$ such that $|\mathsf{Head}(r)| \geq 2$ [23]. Otherwise, the program P is called a *normal logic program*.

2.2.1 Answer Sets: Minimal Model Characterization

In answer set programming, an interpretation $M \subseteq \mathsf{atoms}(P)$ lists the true atoms, i.e., an atom x is true under M iff $x \in M$. An assignment M satisfies $\mathsf{Body}(r)$, denoted $M \models \mathsf{Body}(r)$, iff $\mathsf{Body}(r)^+ \subseteq M$ and $\mathsf{Body}(r)^- \cap M = \emptyset$, where $\sim$ is interpreted classically, i.e., $M \models \sim c_i$ iff $M \not\models c_i$. The rule r (see Equation 2.1) specifies that if all atoms in $\mathsf{Body}(r)^+$ *hold* and no atom in $\mathsf{Body}(r)^-$ holds, then $\mathsf{Head}(r)$ also holds. The assignment M satisfies rule r, denoted $M \models r$, if and only if $(\mathsf{Head}(r) \cup \mathsf{Body}(r)^-) \cap M \neq \emptyset$ or $\mathsf{Body}(r)^+ \setminus M \neq \emptyset$. Let $\mathsf{Rules}(P)$ denote the set of all rules in program P. Then, we say that an assignment M satisfies P, denoted $M \models P$, if and only if $M \models r$ for each $r \in \mathsf{Rules}(P)$.

Given an assignment (or set of atoms) M, the *Gelfond-Lifschitz (GL) reduct* of a program P w.r.t. M is defined as $P^M = \{\mathsf{Head}(r) \leftarrow \mathsf{Body}(r)^+ \mid r \in \mathsf{Rules}(P), \mathsf{Body}(r)^- \cap M = \emptyset\}$ [98]. A set of atoms M is an answer set of P if and only if $M \models P^M$, but $N \not\models P^M$ for every proper subset N of M. The set of all answer sets of program P is denoted by $\mathsf{AS}(P)$, and the answer set counting problem is to compute $|\mathsf{AS}(P)|$, which is denoted by $\mathsf{CntAS}(P)$.

It is folklore that we cannot obtain new answer sets from introducing integrity constraints.

Observation 1. *Let P be a program, X be a set of integrity constraint rules, and $M \subseteq \mathsf{atoms}(P)$. Moreover, let M satisfy P, but there is a set $N \subsetneq M$ such that N satisfies P^M. Then, if M satisfies $P \cup X$, there is also a set $N' \subsetneq M$ such that N' satisfies $(P \cup X)^M$.*

Proof. Let P, X, and M be as given above, in particular, assume that M satisfies both P and $P \cup X$, but $N \subsetneq M$ is a model of P^M. Since M also satisfies $P \cup X$ by assumption, for every rule $r \in (P \cup X)$ either (i) $\mathsf{Body}(r)^- \cap M \neq \emptyset$ and hence $r \notin (P \cup X)^M$ or (ii) $\mathsf{Body}(r)^- \cap M = \emptyset$ and $r \in (P \cup X)^M$. In Case (i) the rule is not relevant when considering whether N satisfies $(P \cup X)^M$, hence we can ignore that case. In Case (ii), if (iia) $r \in P^M$, we have that N satisfies such r by assumption. If (iib) $r \in X^M$, clearly it is true that $H(r) = \mathsf{Body}(r)^- = \emptyset$ as r is a constraint. Since M satisfies $(P \cup X)$ and in particular the rule $r \in X$, we have that $\mathsf{Body}(r)^+ \setminus M \neq \emptyset$. Since $N \subsetneq M$, we have in particular that $\mathsf{Body}(r)^+ \setminus N \neq \emptyset$. Hence, N also satisfies $r \in P^M$. Considering all cases, we can conclude that the observation is true. □

Corollary 2.1. *Let P be a program, X be a set of constraint rules, and $M \subseteq \mathsf{atoms}(P)$. Then, $\mathsf{AS}(P \cup X) \subseteq \mathsf{AS}(P)$.*

Proof. Since for every model was not minimal with respect to the GF-reduct of the program, we can still construct a set that prohibits M from being a minimal model of the GF-reduct. □

2.2.2 Clark's Completion

The *Clark Completion* [51, 151] translates an ASP program P to a propositional formula $\mathsf{Comp}(P)$. Note that Clark Completion was introduced for logic programs with negation as failure and it transformed Prolog programs into first-order theories [160]. The definition of answer sets provides an alternative interpretation of the meaning of Prolog rules with negation [160].

Given a program, the completion $\mathsf{Comp}(P)$ is defined as the conjunction of the following propositional implications:

1. (group 1) for each atom $a \in \mathsf{atoms}(P)$ s.t. $\nexists\, r \in \mathsf{Rules}(P)$ and $a \in \mathsf{Head}(r)$, add a unit clause $\neg a$ to $\mathsf{Comp}(P)$

2. (group 2) for each rule $r \in \mathsf{Rules}(P)$, 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 $a \in \mathsf{atoms}(P)$ 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 $\mathsf{Comp}(P)$:

$$a \longrightarrow \bigvee_{i \in [1,k]} \Big(\bigwedge_{\ell \in \mathsf{Body}(r_i)} \ell \wedge \bigwedge_{x \in \mathsf{Head}(r_i) \setminus \{a\}} \neg x \Big)$$

Finally, $\mathsf{Comp}(P)$ is obtained as the logical conjunction of all constraints added above. Note that, for normal logic programs, the implications in Groups 2 and 3 together yield an equivalence ($\longleftrightarrow$) — an atom is true if and only if the body of at least one of its defining rules evaluates to true. It has been shown in the literature that an answer set of P satisfies $\mathsf{Comp}(P)$ but not vice versa [71, 151, 162].

2.2.3 Loop Formula

The idea of *loop formula* was introduced in [162]. We outline the construction of a loop formula for normal programs below. Given a normal program P, we start by defining the *positive dependency graph* $\mathsf{DG}(P)$ of P as follows [139]. The vertices of $\mathsf{DG}(P)$ are simply $\mathsf{atoms}(P)$. For $a, b \in \mathsf{atoms}(P)$, there exists an edge from b to a in $\mathsf{DG}(P)$ if there is a rule $r \in \mathsf{Rules}(P)$ such that $a \in \mathsf{Body}(r)$ and $b = \mathsf{Head}(r)$. A set of atoms $L \subseteq \mathsf{atoms}(P)$ constitutes a *loop* in P if for every two atoms $x, y \in L$ there is a path from x to y in $\mathsf{DG}(P)$ such that all atoms (nodes) on the path are in L. An atom a is called a *loop atom* of P if there is a loop L in P such that $a \in L$. We use $\mathsf{Loops}(P)$ and $\mathsf{LA}(P)$ to denote the set of all loops and the set of all loop atoms of P, respectively. A program P is called *tight* if there is no loop in P; otherwise, P is called *non-tight*. Lin and Zhao [162] showed that atoms in a loop cannot be asserted true by themselves; instead they must be asserted by some atoms external to the loop. Specifically, a rule r is an *external support* of a loop L in P if $\mathsf{Head}(r) \in L$ and $\mathsf{Body}(r)^+ \cap L = \emptyset$. Let $\mathsf{ExtRule}(L)$ denote the set of all external supports of loop L in P. The loop formula $\mathsf{LF}(L, P)$ [151] of a loop L in program P

can now be defined as follows:

$$\mathsf{LF}(L,P) = (\bigwedge_{a \in L} a) \rightarrow \bigvee_{r \in \mathsf{ExtRule}(L)} \mathsf{Body}(r)$$

Finally, the loop formula $\mathsf{LF}(P)$ of program P is defined as the conjunction of loop formulas for all loops L in P, i.e. $\bigwedge_{L \in \mathsf{Loops}(P)} \mathsf{LF}(L,P)$. Let $M \subseteq \mathsf{atoms}(P)$ be a subset of atoms of P. We use $\tau^M : \mathsf{atoms}(P) \rightarrow \{0,1\}$ to denote the assignment corresponding to M, i.e. $\tau^M(v) = 1$ if $v \in M$ and $\tau^M(v) = 0$ otherwise, for all $v \in \mathsf{atoms}(P)$. Then M is an answer set of P if and only if τ^M satisfies the propositional formula $\mathsf{Comp}(P) \wedge \mathsf{LF}(P)$ [162].

2.2.4 Answer Sets: Unfounded Set Characterization

An alternative characterization of answer sets is based on so called *unfounded sets* [210], which is widely used in state-of-the-art ASP solvers [8, 94]. Moreover, let $I \subseteq \mathrm{lit}(P)$ (a literal is an atom or its negation and the notation $\mathrm{lit}(P)$ denotes all literals of P). A set $U \subseteq \mathsf{atoms}(P)$ is an *unfounded set* wrt. I if, for each rule $r \in P$, we have (i) $\mathsf{Head}(r) \notin U$ (ii) $\mathsf{Body}(r)^+ \cap I^- \neq \emptyset$ or $\mathsf{Body}(r)^- \cap I^+ \neq \emptyset$, or (iii) $\mathsf{Body}(r)^+ \cap U \neq \emptyset$.

Then, M is an *answer set* of a program P if (U1) M satisfies $\mathsf{Comp}(P)$ and (U2) no loop contained in M is unfounded. ASP solvers use a slightly varying characterization based on *nogoods* of unfounded sets; that express (ii) as a *nogood* [94].

2.2.5 Faceted Answer Set Navigation

We use some notations from faceted answer set navigation [4]. The faceted answer set navigation shows that for a given atom $a \in \mathsf{atoms}(P)$ and $f \in \{a, \mathsf{not}\ a\}$ $\mathsf{AS}(P \cup \mathsf{Rule}(\leftarrow\ f)) = \{\tau \in \mathsf{AS}(P) | \tau \models \mathsf{Rule}(\leftarrow\ f)\}$, i.e., adding an *integrity constraint* to a program P filters out answer sets of P that do not satisfy the integrity constraint. More specifically, $\mathsf{AS}(P \cup \mathsf{Rule}(\leftarrow a)) = \{\tau \in \mathsf{AS}(P) | a \notin \tau\}$ and $\mathsf{AS}(P \cup \mathsf{Rule}(\leftarrow \mathsf{not}\ a)) = \{\tau \in \mathsf{AS}(P) | a \in \tau\}$ (we often the notation $\mathsf{Rule}(.)$ to express a rule).

2.2.6 Independent Support of ASP Programs

Approximate counting and sampling widely use *independent support* [122, 192] of a theory. For an ASP program P, we say a set $I \subseteq \mathsf{atoms}(P)$ of atoms is an *independent support* if for any answer sets M_1, $M_2 \in \mathsf{AS}(P)$ we have that $M_1 \cap I = M_2 \cap I$, then $M_1 = M_2$. Intuitively, assigning atoms of independent support I uniquely defines an answer set. Moreover, $\mathsf{atoms}(P)$ is also an independent support, which is called *trivial independent support*.

Example 2.1. *Consider the program* $P = \{a_i \vee \bar{a}_i.\ b_i \leftarrow a_i.\ c_i \leftarrow\sim a_i.\}$, *where* $i = 1, \ldots, 10$. *Observe that some independent supports of program* P *are* $\{a_1, \ldots, a_{10}\}$, $\{\bar{a_1}, \ldots, \bar{a_{10}}\}$, $\{b_1, \ldots, b_{10}\}$, *and* $\{c_1, \ldots, c_{10}\}$.

2.3 Independent Hash Functions

2.3.1 *k-wise Independent* Hash Function

A special class of hash function h is called *k-wise independent* if for all distinct elements $x_1, \ldots, x_k$, the values $h(x_1), \ldots, h(x_k)$ are independent. Formally, let $\mathcal{H}(n, m) = \{h : \{0,1\}^n \to \{0,1\}^m\}$ be a family of hash functions. We call $\mathcal{H}$ the family of *k-wise independent* functions if for any distinct x_1, ..., $x_k \in \{0,1\}^n$, and any y_1, ..., $y_k \in \{0,1\}^m$, we have

$$\Pr_{h \in \mathcal{H}}[h(x_1) = y_1 \wedge \ldots \wedge h(x_k) = y_k] = (\frac{1}{2^m})^k \tag{2.2}$$

2.3.2 A Special Family of Hash Function

There is a special family of hash functions based on random XOR or parity constraints, denoted by $\mathcal{H}_{xor}(n, m)$ [47]. The family $\mathcal{H}_{xor}(n, m)$ can be defined as $\mathbf{Ax} + \mathbf{B}$, where $\mathbf{x}$ is one-dimensional matrix representation of $\mathsf{atoms}(P)$, $|\mathsf{atoms}(P)|= n$, $\mathbf{A} \in \{0,1\}^{m \times n}, \mathbf{B} \in \{0,1\}^{m \times 1}$, each entry of $\mathbf{A}$ and $\mathbf{B}$ are generated according to a *Bernoulli distribution* with a probability of 0.5. The family of hash function has been shown to be 3-wise independent [103].

2.4 Model Counting with Probabilistic Guarantee

Let assume that $\mathrm{Sol}(I)$ consists of the set of solutions for a problem instance I.

2.4.1 (ε, δ)-Approximate Model Counting

An approximate counting tries to compute the number of solutions using a probabilistic algorithm approximately [47, 140]. An approximation counting algorithm takes an instance I, $\varepsilon > 0$ called *tolerance*, and δ with $0 < \delta \leq 1$ called *confidence* as input. The output is a real number cnt, which estimates the cardinality of $\mathrm{Sol}(I)$ based on the parameters ε and δ following the inequality:

$$\Pr\left[\frac{|\mathrm{Sol}(I)|}{(1+\varepsilon)} \leq \mathrm{cnt} \leq (1+\varepsilon) \cdot |\mathrm{Sol}(I)|\right] \geq 1 - \delta.$$

2.4.2 Probabilistic Lower Bound

Let the real number cnt represents a lower bound estimate for the cardinality of $\mathrm{Sol}(I)$. We assert that cnt is a lower bound for the cardinality of $\mathrm{Sol}(I)$ with a *confidence* δ, when

$$\mathsf{Pr}[\mathrm{cnt} \leq |\mathrm{Sol}(I)|] \geq 1 - \delta$$

2.5 Graph Thoery

Let $G = (V, E)$ be a graph, where $V = \mathsf{Node}(G)$ is the set of the nodes, and $E = \mathsf{Edge}(G)$ is the set of edges. Each edge $e \in E$ is represented as a tuple $e = (a, b)$, where nodes $a, b \in \mathsf{Node}(G)$ are two endpoints of e. If there is an edge $(a, b) \in E$, then node a (b resp.) is *adjacent* to node b (a resp.). A graph G' is a *subgraph* of G, denoted as $G' = (V', E')$, if $V' \subseteq V$ and $E' \subseteq E$.

2.5.1 Two-terminal Network Reliability

Given two arbitrary nodes $s, t \in \mathsf{Node}(G')$, if there exists a set of edges in G' that connects nodes s and t or there is a path in G' with nodes s and t, then G' is referred to as (s, t)-connected subgraph where nodes s and t are the source and

target nodes, respectively. The nodes s and t are also called *terminal* nodes. We use the notation $\mathsf{Subgraph}(G, s, t)$ to denote all (s, t)-connected subgraphs of G.

In this work, our graphs are probabilistic and the probabilities are assigned to the edges. The probability of edge e is represented by $\mathsf{W}(\mathsf{e})$, which determines the likelihood that edge e is active and the likelihood of edge e failing is represented by $1 - \mathsf{W}(\mathsf{e})$. A graph is *unweighted* if $\forall e \in \mathsf{Edge}(G), \mathsf{W}(\mathsf{e}) = 1/2$; otherwise, the graph is *weighted*. Given a subgraph G', $\mathsf{Pr}[G']$ is defined as $\prod_{e_i \in \mathsf{Edge}(G')} \mathsf{W}(\mathsf{e_i}) \times \prod_{e_i \in \mathsf{Edge}(G) \setminus \mathsf{Edge}(G')} (1 - \mathsf{W}(\mathsf{e_i}))$, i.e., the probability of a subgraph G' is calculated as the product of the probabilities of its edges that are active and the complement of the probabilities of its edges that are inactive. The reliability of graph G w.r.t. source node s, target node t, and probability over edges W, is defined as $r(G, u, v, \mathsf{W}) = \sum_{G' \in \mathsf{Subgraph}(G,s,t)} \mathsf{Pr}[G']$, i.e., the reliability of a graph G, with respect to source node s, target node t and edge probability W, is defined as the sum of the probabilities of all (s, t)-connected subgraphs of G.

2.5.2 Two Operations on Graphs

We introduce two well-known operations on graphs. The *removal* of edge e on a graph $G = (V, E)$, denoted as $G \setminus e$, which is defined as $(V, E \setminus \{e\})$, i.e., deleting the edge e from graph G. The *contraction* of edge $e = (a, b)$ on a graph $G = (V, E)$, denoted as G/e, which is defined as (V', E'), where the node set $V' = V \setminus \{a, b\} \cup \{c\}$, the new node c is not present in V and the edge set $E' = E \cup \{(d, c) | d \notin \{a, b\}$ and node d is either adjacent to a or $b\} \setminus \{e' \in E |$ one of the endpoints of e' is either a or $b\}$, i.e., contraction of edge e merges two endpoints of e into a newly introduced node.

2.6 Chain Formula

The *chain formula* [46] is a restricted class of propositional formulas and has been found to be useful for reducing *weighted model counting* to *unweighted model counting*. Let we are interested in computing chain formula corresponding to the weight of $\frac{m}{2^k}$ (obtained after possible reduction), where $m > 0$ is a natural number, and $k < 2^m$ is a positive odd number. Let $c_1, \ldots, c_m$ be the binary representation of k, where c_m be the least significant bit. Then we can formulate the chain formula

$\phi_{k,m}$ over m propositional atoms $b_1, \ldots, b_m$ using the following notation:

$$\phi_{k,m}(b_1, \ldots, b_m) = (b_1 C_1 (b_2 C_2 \ldots (b_{m-1} C_{m-1} b_m) \ldots))$$

where $C_i = \vee$, if $c_i = 1$, otherwise $C_i = \wedge$. Chakraborty et al. [46] showed that the size of $\phi_{k,m}$ is linear with m and chain formula $\phi_{k,m}$ has k satisfying assignments.

Although the chain formula is not in conjunctive normal form (CNF), it can be converted into CNF by introducing fresh Boolean variables and *equivalence* (bi-implication). A standard method for this conversion is the *Tseitin transformation* [207], which produces an *equisatisfiable* CNF formula without an exponential increase in size. The transformation assigns a fresh Boolean variable t to each subformula ϕ, using t as a placeholder for subformula ϕ. The original formula is then rewritten by replacing occurrences of ϕ with t, and additional constraints of the form $t \leftrightarrow \phi$ are introduced to preserve logical consistency [169]. This process is applied recursively until the entire formula is expressed in CNF.

Tseitin Transformation of $\phi_{k,m}$. We demonstrate the chain formula transformation as follows (the transformation is used in our work in Chapter 6): the transformation first introduces an equivalence and a new atom for the innermost simple Boolean expression of $\phi_{k,m}$; the simple Boolean expression is $(b_{m-1} C_{m-1} b_m)$, and the equivalence is $t_{m-1} \leftrightarrow (b_{m-1} C_{m-1} b_m)$, where t_{m-1} is a new propositional atom. Then the transformation introduces another equivalence and a new atom for the second innermost simple Boolean expression of $\phi_{k,m}$ (if any), namely, the equivalence is $t_{m-2} \leftrightarrow (b_{m-2} C_{m-2} (b_{m-1} C_{m-1} b_m))$. However, a truth value of the Boolean expression $(b_{m-1} C_{m-1} b_m)$ defines the truth value of t_{m-1}. Thus, the new equivalence can be written as $t_{m-2} \leftrightarrow (b_{m-2} C_{m-2} t_{m-1})$. The transformation continues in this way until the transformation encounters the simple Boolean expression $b_1 C_1 t_2$. Thus, the transformation generates a total of $m-2$ propositional atoms $(t_2, \ldots, t_{m-1})$ and derives a total of $m-1$ equivalences. For simplification, we introduce one additional equivalence, $t_1 \leftrightarrow (b_1 C_1 t_2)$, to the set of equivalences. Given a chain formula $\phi_{k,m}$, let denote the transformation using the notation $\mathsf{T}(\phi_{k,m})$.

The transformation introduces a new set of propositional atoms, which are logically defined by the original set of atoms $\{b_1, \ldots, b_m\}$. As a result, an assignment over the atom set of $\{b_1, \ldots, b_m\}$ uniquely defines the truth value of $\{t_1, \ldots, t_{m-1}\}$.

For arbitrary assignment over atom set of $\{b_1, \ldots, b_m\}$, the truth values of t_{i-1} and $b_{i-1}C_{i-1}O_i$ are same, for $i \in [1, m-1]$, where O_i is the other operand of C_i except b_{i-1}. It follows that if an assignment τ over $\{b_1, \ldots, b_m\}$ satisfies $\phi_{k,m}$, then τ evaluates t_1 to be true. Thus, $\phi_{k,m}$ and $\mathsf{T}(\phi_{k,m}) \wedge \{t_1 \leftrightarrow 1\}$ have the same number of satisfying assignments. As a result, $\mathsf{T}(\phi_{k,m}) \wedge \{t_1 \leftrightarrow 1\}$ preserves the number of satisfying assignments of the original chain formula $\phi_{k,m}$.

Example 2.2. *Construct the chain formula for* $k = 5$ *and* $m = 3$.
The binary representation of 5 *using* 3 *bits is* 101. *Therefore, we have* $\phi_{k,m}(b_1, b_2, b_3) = (b_1 \vee (b_2 \wedge b_3))$, $\mathsf{T}(\phi_{k,m}) = \{t_1 \leftrightarrow (b_1 \vee t_2), t_2 \leftrightarrow (b_2 \wedge b_3)\}$. *Finally,* $\mathsf{T}(\phi_{k,m}) \wedge \{t_1 \leftrightarrow 1\}$ *has* 5 *satisfying assignments.*

2.7 Boolean Networks

A Boolean Network (BN) f is defined as a finite set of Boolean functions over a finite set of Boolean variables, denoted by $\mathsf{Var}(f)$. Each variable $v \in \mathsf{Var}(f)$ is associated with a Boolean function $f_v \colon \mathbb{B}^{|\mathsf{Var}(f)|} \to \mathbb{B}$. A function f_v is termed *constant* if it is always either 0 or 1 regardless of the values of its arguments. A variable v is considered a *source variable* if f_v is the identity function on v, i.e., $f_v = v$. A state s of f is a Boolean vector $s \in \mathbb{B}^{|\mathsf{Var}(f)|}$ that can be viewed as a mapping: $s \colon \mathsf{Var}(f) \to \mathbb{B}$; we denote the value of variable v in state s by s_v. For convenience, a state is often represented as a string of values (e.g., "0110" instead of $(0, 1, 1, 0)$).

2.7.1 Update scheme of Boolean Networks

At each discrete time step t, each variable v can update its state according to its Boolean function f_v; that is, v's state at time $t+1$ is given by $s'_v = f_v(s)$. An *update scheme* specifies how these state updates occur over time [187]. The two primary schemes are synchronous, in which all variables update simultaneously, and fully asynchronous, where a single variable is chosen non-deterministically to update. Under arbitrary update scheme, the BN transitions from one state to another — a process known as a *state transition*. The overall dynamics of the BN are captured by the *State Transition Graph* (STG), a directed graph whose nodes represent states

and edges represent transitions. We denote the STG under the synchronous update scheme as $\mathsf{sstg}(f)$ and that under the fully asynchronous scheme as $\mathsf{astg}(f)$.

2.7.2 Trap Set, Minimal Trap Space, and Fixed Point

A non-empty set A of states is a *trap set* if there is no transition from a state in A to a state outside A in the State Transition Graph (STG) of f (i.e., there is no pair $x \in A$ and $y \notin A$ such that (x, y) is an arc in the STG) [144]. A trap set that is minimal with respect to set inclusion is termed an *attractor*. In particular, an attractor containing a single state is called a *fixed point*, while one with two or more states is referred to as a *cyclic* attractor. A *sub-space* m of a BN f is a mapping $m\colon \mathsf{Var}(f) \to \mathbb{B}_\star$. A variable $v \in \mathsf{Var}(f)$ is said to be *fixed* (resp. *free*) in m if $m(v) \neq \star$ (resp. $m(v) = \star$). For convenience, a sub-space is often represented as a string of values (e.g., $0\star$ instead of $\{v_1 = 0, v_2 = \star\}$). The sub-space m represents a set of states, denoted by $\mathcal{S}[m]$, defined as

$$\mathcal{S}[m] = \{s \in \mathbb{B}^{|\mathsf{Var}(f)|} \mid s_v = m(v), \forall v \in \mathsf{Var}(f), m(v) \neq \star\}$$

For example, if $m = \star 11$, then $\mathcal{S}[m] = \{011, 111\}$. If a sub-space is also a trap set, it is a *trap space*. Unlike trap sets and attractors, trap spaces are independent of the update scheme employed [144]. Notably, a fixed point of f is a special trap space in which all variables are fixed. A trap space m is *minimal* if there is no trap space m' such that $\mathcal{S}[m'] \subset \mathcal{S}[m]$. Since an attractor is a subset-minimal trap set, a minimal trap space contains at least one attractor of the BN, regardless of the update scheme employed [144].

Example 2.3. *Let us consider BN f with $\mathsf{Var}(f) = \{a, b\}$, $f_a = a \wedge \neg b$, and $f_b = a$. The synchronous STG of f is shown in Figure 2.1a. The set $\{00, 01, 11\}$ is a trap set but not a trap space. It is easy to check that f has three trap spaces: $m_1 = 00$, $m_2 = 0\star$, and $m_3 = \star\star$. Among these, m_1 is a minimal trap space (also a fixed point) of f. In this case, m_1 is also the only synchronous attractor of f.*

2.8 Answer Sets and Minimal Models

There are similarities between minimal models and answer sets (discussed in Chapter 8). More specifically, we can reason minimal models through answer set solving.

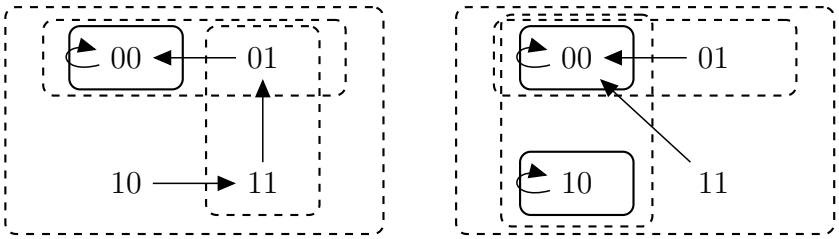


Figure 2.1: (a) Synchronous STG $\mathsf{sstg}(f)$ of BN f from Example 2.3. (b) Synchronous STG $\mathsf{sstg}(f)$ where variable b is subject to a *knockout* (i.e., its value is forced to 0). Trap spaces (resp. minimal trap spaces) are enclosed by dashed (resp. solid) rectangular frames.

2.8.1 From Minimal Models to Answer Sets.

Consider a Boolean formula, $F = \bigwedge_i C_i$, where each clause is of the form: $C_i = \ell_0 \vee \ldots \ell_k \vee \neg\ell_{k+1} \vee \ldots \neg\ell_m$. We can transform each clause C_i into a rule r of the form: $\ell_0 \vee \ldots \vee \ell_k \leftarrow \ell_{k+1}, \ldots, \ell_m$. Given a formula F, let us denote this transformation by the notation $\mathcal{DLP}(F)$. Each minimal model of F corresponds uniquely to an answer set of $\mathcal{DLP}(F)$.

Lemma 2.1. *Each minimal model of F corresponds to an answer set of $\mathcal{DLP}(F)$.*

2.8.2 Minimal Model Applications: Minimal Generators

We define transactions over a finite set of items, denoted by $\mathcal{I}$. A *transaction* t_i is an ordered pair of (i, I_i), where i is the unique identifier of the transaction and $I_i \subseteq \mathcal{I}$ represents the set of items involved in the transaction. A transaction database is a collection of transactions, where each uniquely identified by the identifier i, corresponding to the transaction t_i. A transaction (i, I_i) supports an itemset $J \subseteq \mathcal{I}$ if $J \subseteq I_i$. The *cover* of an itemset J within a database D, denoted as $C(J, D)$, is defined as: $\mathcal{C}(J, D) = \{i | (i, I_i) \in D \text{ and } J \subseteq I_i\}$. Given an itemset I and transaction database D, the itemset I is a *minimal generator* of D if, for every itemset J where $J \subset I$, it holds that $\mathcal{C}(I, D) \subset \mathcal{C}(J, D)$.

Encoding Minimal Generators as Minimal Models Given a transaction database D, we encode a Boolean formula $\mathsf{MG}(D)$ such that minimal models of $\mathsf{MG}(D)$ correspond one-to-one with the minimal generators of D. This encoding

introduces two types of variables: (i) for each item $a \in \mathcal{I}$, we introduce a variable p_a to denote that a is present in a minimal generator (ii) for each transaction t_i, we introduce a variable q_i to denote the presence of the itemset in the transaction t_i. Given a transaction database $D = \{t_i | i = 1, \ldots n\}$, consisting of the union of transactions t_i, consider the following Boolean formula:

$$\mathsf{MG}(D) = \bigwedge_{i=1}^{n} \left(\neg q_i \rightarrow \bigvee_{a \in \mathcal{I} \setminus I_i} p_a \right) \tag{2.3}$$

Lemma 2.2. *Given a transaction database D, σ is a minimal model of $\mathsf{MG}(D)$ if and only if the corresponding itemset $I_\sigma = \{a | p_a \in \sigma\}$ is a minimal generator of D.*

The encoding of $\mathsf{MG}(D)$ bears similarities to the encoding detailed in [123, 183]. However, our encoding achieves compactness by incorporating a *one-sided implication*, which enhances the efficiency of the representation.

The proofs of Lemma 2.1 and 2.2 is deferred to Chapter A.

2.9 Subtractive Reduction in Counting Problems

Borrowing notation from [61], 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) $|Q_1(x)| = |Q_2(f(x))| - |Q_2(g(x))|$.

Part I

Counting Answer Sets

In this part, we discuss about answer set counting techniques, drawing on the following three publications:

- **[KCM2024]** Mohimenul Kabir, Supratik Chakraborty, and Kuldeep S. Meel. "Exact ASP counting with compact encodings." *AAAI*, vol. 38, no. 9, pp. 10571-10580. 2024.
- **[KCM2025]** Mohimenul Kabir, Supratik Chakraborty, and Kuldeep S. Meel. "Counting Answer Sets of Disjunctive Answer Set Programs." *ICLP*, 2025.
- **[KES$^+$2022]** Mohimenul Kabir, Flavio O. Everardo, Ankit K. Shukla, Markus Hecher, Johannes Klaus Fichte, and Kuldeep S. Meel. "ApproxASP - A scalable approximate answer set counter." *AAAI*, vol. 36, no. 5, pp. 5755-5764. 2022.

In this part, we discuss about answer set counting techniques. First we present exact answer set counting techniques based on an alternative definition of answer sets in Chapter 3 (from [KCM2024]) and Chapter 4 (from [KCM2025]). Then we present a framework for approximate answer set counting in Chapter 5 (from [KES$^+$2022]).

Chapter 3

Exact Answer Set Counter: sharpASP

We present an answer set counter, named sharpASP, for exact answer set counting. In this work, we target only normal logic programs, which has been used in diverse applications (see for example [40, 57]). The core idea of the counter is an alternative, yet correlated way of defining answer sets. The alternative way of defining answer sets facilitates core ideas like *decomposability* and *determinism* from propositional model counters to our proposed answer set counter. Finally, we empirically evaluate the performance of sharpASP against state-of-the-art answer set counters.

3.1 Related Work

The decision version of normal logic programs is NP-complete; therefore, the ASP counting for normal logic programs is #P-complete [209]. Given the #P-completeness, a prominent line of work focused on ASP counting relies on translations from the ASP program to the CNF formula [65, 66, 125, 126, 127, 162]. Such translations often result in a large number of CNF clauses and thereby limit practical scalability for *non-tight* ASP programs.

Eiter et al. [65, 66] introduced $T_{\mathcal{P}}$-*unfolding* to break cycles and produce a tight program. They proposed an ASP counter called aspmc, that performs a *treewidth-aware* Clark completion from a cycle-free program to the CNF formula. Jakl, Pichler, and Woltran [124] extended the tree decomposition based approach for #SAT due to Samer and Szeider [184] to answer set programming and proposed a fixed-parameter tractable (FPT) algorithm for answer sets counting. Fichte et al. [75, 82] revisited the FPT algorithm due to Jakl et al. and developed an exact model counter, called DynASP, that performs well on instances with low treewidth. Aziz et al. [17] extended a propositional model counter to an answer set counter by

integrating unfounded set detection. Kabir et al. [133] (Chapter 5) focused on lifting hashing-based techniques to ASP counting, resulting in an approximate counter, called ApproxASP, with (ε, δ)-guarantees.

3.2 An Alternative Definition of Answer Sets

Our algorithm for answer set counting crucially relies on an alternative way of defining the answer sets of a normal program P. We first introduce an operation called $\mathsf{Copy}()$ that plays a central role in this alternative definition. Our $\mathsf{Copy}()$ operation is related to, but not the same as, a similar operation used in ASProblog. Specifically, founded variables (i.e. variables appearing at the head of a rule) were the focus of the copy operation used in ASProblog. In contrast, loop atoms in the program P are the focus of the $\mathsf{Copy}()$ operation in our approach. We elaborate more on this below.

3.2.1 $\mathsf{Copy}(P)$ for Normal Logic Programs

Given a normal program P, for every loop atom/variable v in $\mathsf{LA}(P)$, let v' be a fresh variable not present in $\mathsf{atoms}(P)$. We refer to v' as the *copy variable of* v. For $X \subseteq \mathsf{LA}(P)$, we denote the set of copy variables corresponding to atoms in X as X'.

Given a normal program P, the $\mathsf{Copy}(P)$ operation returns a set of (implicitly conjoined) implications, defined as follows:

1. (type 1) for every $v \in \mathsf{LA}(P)$, the implication $v' \rightarrow v$ is in $\mathsf{Copy}(P)$.
2. (type 2) for every rule $a \leftarrow b_1, \ldots b_m, \sim c_1, \ldots \sim c_n$ in P such that $a \in \mathsf{LA}(P)$, the implication $\psi(b_1) \wedge \ldots \psi(b_m) \wedge \neg c_1 \wedge \ldots \neg c_n \rightarrow \psi(a)$ is in $\mathsf{Copy}(P)$, where for each variable x, $\psi(x)$ 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 in $\mathsf{Copy}(P)$.

Note that in implications of type 2, copy variables are used exclusively for positive loop atoms in the body of the rule and for the loop atom in the head of the rule. Specifically, if the head of a rule is not a loop atom, we do not add any implication

of type 2 for that rule. As an extreme case, if P is a tight program or $\mathsf{LA}(P) = \emptyset$, then $\mathsf{Copy}(P) = \emptyset$.

An Alternative Definition of Answer Set We now present a key observation that provides the basis for an alternative definition of answer sets. Akin to the existing definitions of answer set [101, 126, 159], our definition seeks justification for atoms within an answer set. However, our definition seeks to justify only loop atoms belonging to an answer set, while the existing definitions, to the best of our knowledge, aim to justify each atom in an answer set. The alternative definition derives from the observation that under Clark's completion of a program, if the loop atoms of an answer set are justified, then the remaining atoms of the answer set are also justified. Thus, under Clark's completion, it suffices to seek justifications for loop atoms. Unlike existing definitions of answer sets, our definition of answer sets operates exclusively within the realm of Boolean formulas and employs unit propagation as a tool to decide whether an atom is justified or not.

Recall the definition of $\phi|_\tau$, i.e. unit propagation of an assignment τ on a CNF formula ϕ from Section 2.1. Recall also that a CNF formula can be viewed as a set of clauses, where each clause can be interpreted as an implication. Therefore, the set of implications $\mathsf{Copy}(P)$ can be thought of as representing a CNF formula. For an assignment $\tau : X \to \{0, 1\}$ where $X \subseteq \mathsf{atoms}(P)$, we use the notation $\mathsf{Copy}(P)|_\tau$ to denote the (implicitly conjoined) set of implications that remain after unit propagating τ on the CNF formula represented by $\mathsf{Copy}(P)$. Specifically, we say that $\mathsf{Copy}(P)|_\tau = \emptyset$ if τ unit propagates to only unit clauses on copy variables in the CNF formula represented by $\mathsf{Copy}(P)$.

Theorem 1. *For a normal program P, let $X \subseteq \mathsf{atoms}(P)$ and let $\tau : X \mapsto \{0, 1\}$ be an assignment. Let M^τ denote the set of atoms of P that are assigned 1 by τ. Then $M^\tau \in \mathsf{AS}(P)$ if and only if $\tau \models \mathsf{Comp}(P)$ and $\mathsf{Copy}(P)|_\tau = \emptyset$.*

Proof. (i) (proof of 'if part') **Proof By Contradiction**. Assume that $\tau \models \mathsf{Comp}(P)$ and $\mathsf{Copy}(P)|_\tau = \emptyset$, but $M^\tau \notin \mathsf{AS}(P)$. Since $M^\tau \notin \mathsf{AS}(P)$ and $\tau \models \mathsf{Comp}(P)$, it implies that $\tau \not\models \mathsf{LF}(P)$. Thus, there is a loop L in P such that $\tau \not\models \mathsf{LF}(L, P)$. Assume that L is comprised of the set of loop atoms $\{ x_1, \ldots, x_k \}$. Then $\tau \not\models x_1 \wedge \ldots \wedge x_k \to \bigvee_{r \in \mathsf{ExtRule}(L)} \mathsf{Body}(r)$. In other words, even if τ is augmented by

setting $x_1 = \ldots = x_k = 1$, the formula $\bigvee_{r \in \mathsf{ExtRule}(L)} \mathsf{Body}(r)$ evaluates to 0 under the augmented assignment. Now recall that τ itself is an assignment to a subset of $\mathsf{atoms}(P)$, and it does not assign any truth value to $x_1', \ldots, x_k'$. Therefore, there must be at least one type 2 implication in $\mathsf{Copy}(P)|_\tau$, specifically one arising from a rule $r \in \mathsf{ExtRule}(L)$ (see definition from Subsection 2.2.3), that does not unit propagate to a unit clause or to 1 under τ. This contradicts the premise that $\mathsf{Copy}(P)|_\tau = \emptyset$.

(ii) (proof of 'only if part') **Proof By Contradiction**. Suppose $M^\tau \in \mathsf{AS}(P)$. We know that this implies $\tau \models \mathsf{Comp}(P) \wedge \mathsf{LF}(P)$. We now show that in this case, we must also have $\mathsf{Copy}(P)|_\tau = \emptyset$. Suppose, if possible, $\mathsf{Copy}(P)|_\tau \neq \emptyset$. We ask if an implication of type 1, say $v' \rightarrow v$, can stay back in $\mathsf{Copy}(P)|_\tau$. If $v \in M^\tau$, then $\tau(v) = 1$, and clearly the implication $v' \rightarrow v$ doesn't stay back in $\mathsf{Copy}(P)|_\tau$. If $v \notin M^\tau$, then $\tau(v) = 0$, and in this case τ unit propagates to $\{\neg v'\}$, and hence the implication doesn't stay back in $\mathsf{Copy}(P)|_\tau$ either. Therefore, no implication of type 1 can stay back in $\mathsf{Copy}(P)|_\tau$. Next, we ask if any implication of type 2 can stay back in $\mathsf{Copy}(P)|_\tau$. Suppose this is possible. Note that for every $v \in \mathsf{atoms}(P)$, either $v \in M^\tau$ or $v \notin M^\tau$. Therefore, $\tau(v)$ is either 0 or 1 for all $v \in \mathsf{atoms}(P)$. Therefore, if $\mathsf{Copy}(P)|_\tau \neq \emptyset$, there must be some $x_1' \in \mathsf{Var}(\mathsf{Copy}(P)|_\tau)$ and there must a (potentially simplified) implication $x_2' \wedge C_1 \rightarrow x_1'$ in $\mathsf{Copy}(P)|_\tau$, where C_1 is either true or a conjunction of copy variables. The existence of copy variable x_2' in $\mathsf{Copy}(P)|_\tau$ implies the existence of another implication: $x_3' \wedge C_2 \rightarrow x_2'$ in $\mathsf{Copy}(P)|_\tau$. Continuing this argument, we find that there are two cases to handle: (i) there are an unbounded number of copy variables in $\mathsf{Copy}(P)|_\tau$, which contradicts the fact that there can be at most $|\mathsf{Var}(P)|$ copy variables. (ii) otherwise, there exists i, j such that $x_i' = x_j'$ and $i < j$, which implies that the set of variables $\{ x_i, \ldots x_{j-1} \}$ constitutes an *unfounded set*. However, this contradicts the fact that $M^\tau \in \mathsf{AS}(P)$. In either case, we reach a contradiction, thereby proving that $\mathsf{Copy}(P)|_\tau$ is empty. This completes the proof. □

Example 3.1. *Consider the normal program P given by the rules $\{r_1 = a \leftarrow\sim b.\ \ r_2 = b \leftarrow\sim a.\ \ r_3 = c \leftarrow a, b.\ \ r_4 = c \leftarrow d.\ \ r_5 = d \leftarrow a.\ \ r_6 = d \leftarrow b, c.\ \ r_7 = e \leftarrow\sim a, \sim b\}$.*

This program has a single loop L consisting of atoms c and d, i.e. $\mathsf{LA}(P) = \{c, d\}$.

Therefore, $\mathsf{Copy}(P)$ *consists of the conjunction of implications:* $\{\, c' \rightarrow c, d' \rightarrow d, a \wedge b \rightarrow c', d' \rightarrow c', a \rightarrow d', b \wedge c' \rightarrow d' \,\}$. *Note that there are no variables* a', b', e' *or constraints involving them in* $\mathsf{Copy}(P)$. *The followings are now easily verified.*

- *Consider* τ_1 *that assigns* 1 *to* b *and* 0 *to* a, c, d, e. *For the corresponding answer set* M^{τ_1}*:* $\{\, b \,\}$, $\mathsf{Copy}(P)|_{\tau_1} = \emptyset$
- *Consider* τ_2 *assigns* 1 *to* a, c, d *and* 0 *to* b, e. *For the corresponding answer set* M^{τ_2}*:* $\{\, a, c, d \,\}$, $\mathsf{Copy}(P)|_{\tau_2} = \emptyset$
- *Consider* τ_3 *that assigns* 1 *to* b, c, d *and* 0 *to* a, e. *For the corresponding non-answer set* M^{τ_3}*:* $\{\, b, c, d \,\}$, $\mathsf{Copy}(P)|_{\tau_3} \neq \emptyset$

3.3 Answer Set Counter: sharpASP

In this section, we first show how the alternative definition of answer sets provides a new way to counting all answer sets of a given normal program. Subsequently, we explore how off-the-shelf state-of-the-art propositional model counters can be easily adapted to correctly count answer sets by leveraging the alternative definition.

It is straightforward from Theorem 1 that the count of answer sets of a normal program P can be obtained simply by counting assignments $\tau \in 2^{|\mathsf{atoms}(P)|}$ such that $\tau \models \mathsf{Comp}(P)$ and $\mathsf{Copy}(P)|_{\tau} = \emptyset$. This motivates us to represent a normal program P using a pair (F, G), where $F = \mathsf{Comp}(P)$ and $G = \mathsf{Copy}(P)$. Further, we discuss below how key ideas in propositional model counters can be adapted to work with this pair representation of normal programs to yield exact answer set counters.

3.3.1 Decomposition

Propositional model counters often *decompose* the input CNF formula into *disjoint subformulas* to boost up the counting efficiency [22] – for two formulas ϕ_1 and ϕ_2, if $\mathsf{Var}(\phi_1) \cap \mathsf{Var}(\phi_2) = \emptyset$, then ϕ_1 and ϕ_2 are *decomposable*, i.e., we can count the number of models of ϕ_1 and ϕ_2 separately and multiply these two counts to get the number of models of $\phi_1 \wedge \phi_2$.

Given a normal program, our proposed definition involves a pair of formulas: F and G. Specifically, we define *component decomposition* with respect to (F, G) as follows:

Definition 1. *$(F_1 \wedge F_2, G_1 \wedge G_2)$ is decomposable to (F_1, G_1) and (F_2, G_2) if and only if $(\mathsf{Var}(F_1) \cup \mathsf{Var}(G_1)) \cap (\mathsf{Var}(F_2) \cup \mathsf{Var}(G_2)) = \emptyset$.*

Finally, Lemma 3.1 offers evidence supporting the correctness of our proposed definition of decomposition in computing the number of answer sets.

Lemma 3.1. *Let $(F_1 \wedge \ldots F_k, G_1 \wedge \ldots G_k)$ is decomposed to $(F_1, G_1), \ldots, (F_k, G_k)$ then $\mathsf{CntAS}(F_1 \wedge \ldots F_k, G_1 \wedge \ldots G_k) = \mathsf{CntAS}(F_1, G_1) \times \ldots \mathsf{CntAS}(F_k, G_k)$*

Proof. By definition of decomposition, we know that $(\mathsf{Var}(F_i) \cup \mathsf{Var}(G_i)) \cap (\mathsf{Var}(F_j) \cup \mathsf{Var}(G_j)) = \emptyset$ for $1 \leq i < j \leq k$. This, in turn, implies that $\mathsf{Var}(G_i) \cap \mathsf{Var}(G_j) = \emptyset$ for $1 \leq i < j \leq k$. Therefore, no variable (copy variable or otherwise) is common in G_i and G_j, if $i \neq j$. Hence, for every assignment $\tau : \mathsf{atoms}(P) \to \{0, 1\}$, unit propagation of τ on G_i and G_j must happen completely independent of each other, i.e. no unit literal obtained by unit propagation of τ on G_i affects unit propagation of τ on G_j, and vice versa. In other words, $G_i|_\tau \wedge G_j|_\tau = (G_i \wedge G_j)|_\tau$.

Let $F = F_1 \wedge \ldots F_k$ and $G = G_1 \wedge \ldots G_k$. In the following, we use the notation τ to denote an assignment $\mathsf{atoms}(P) \to \{0, 1\}$, and τ_i to denote an assignment $\mathsf{atoms}(P) \cap (\mathsf{Var}(F_i) \cup \mathsf{Var}(G_i)) \to \{0, 1\}$, for $1 \leq i \leq k$. By virtue of the argument in the previous paragraph, it is easy to see that the domains of τ_i and τ_j are disjoint for $1 \leq i < j \leq k$. We use the notation $\tau_1 \cup \ldots \tau_k$ to denote the assignment $\mathsf{atoms}(P) \to \{0, 1\}$ defined as follows: if $v \in \mathsf{atoms}(P) \cap (\mathsf{Var}(F_i) \cup \mathsf{Var}(G_i))$, then $(\tau_1 \cup \ldots \tau_k)(v) = \tau_i(v)$. The proof now consists of showing the following two claims:

1. $\mathsf{CntAS}(F_1, G_1) \times \cdots \mathsf{CntAS}(F_k, G_k) \geq \mathsf{CntAS}(F, G)$.
2. $\mathsf{CntAS}(F_1, G_1) \times \cdots \mathsf{CntAS}(F_k, G_k) \leq \mathsf{CntAS}(F, G)$.

Proof of part 1: Suppose $\tau \in \mathsf{AS}(F, G)$. By definition, $\tau \models F$ and $G|_\tau = \emptyset$. Since $F = F_1 \wedge \ldots F_k$, we know that $\tau \models F_i$ for $1 \leq i \leq k$. By the above definition of τ_i, it then follows that $\tau_i \models F_i$. Similarly, since unit propagation of τ on G_i and G_j happen independently for all $i \neq j$, and since unit propagation of τ on $G = G_1 \wedge \ldots G_k$ gives

$\emptyset$, we have $G_i|_{\tau_i} = \emptyset$ as well. It follows that $\tau_i \in \mathsf{AS}(F_i, G_i)$ for $1 \leq i \leq k$. Therefore, every $\tau \in \mathsf{AS}(F, G)$ yields a sequence of $\tau_i \in \mathsf{AS}(F_i, G_i)$, for $1 \leq i \leq k$. Since the domains of all τ_i's are distinct, it follows that $\mathsf{CntAS}(F_1, G_1) \times \cdots \mathsf{CntAS}(F_k, G_k) \geq \mathsf{CntAS}(F, G)$.

Proof of part 2: Suppose $\tau_i \in \mathsf{AS}(F_i, G_i)$ for $1 \leq i \leq k$. By definition, $\tau_i \models F_i$ and $G_i|_{\tau_i} = \emptyset$. Since the domains of τ_i and τ_j are disjoint for all $1 \leq i < j \leq k$, it follows that $(\tau_1 \cup \ldots \tau_k) \models (F_1 \wedge \ldots F_k)$ and hence $\tau \models F$. We have also seen that $(G_1 \wedge \cdots G_k)|_\tau = (G_1|_\tau \wedge \cdots G_k|_\tau)$. However, since $\mathsf{Var}(G_i)$ is a subset of the domain of τ_i, we have $(G_1 \wedge \cdots G_k)|_\tau = (G_1|_{\tau_1} \wedge \cdots G_k|_{\tau_k})$. Since $G_i|_{\tau_i} = \emptyset$ for $1 \leq i \leq k$, it follows that $(G_1 \wedge \cdots G_k)|_\tau = \emptyset$. Therefore $G|_\tau = \emptyset$. Since $\tau \models F$ as well, we have $\tau \in \mathsf{AS}(F, G)$. Therefore, every distinct sequence of $\tau_i, 1 \leq i \leq k$ such that $\tau_i \in \mathsf{AS}(F_i, G_i)$ yields a distinct $\tau \in \mathsf{AS}(F, G)$. It follows that $\mathsf{CntAS}(F_1, G_1) \times \cdots \mathsf{CntAS}(F_k, G_k) \leq \mathsf{CntAS}(F, G)$.

It follows from the above two claims that

$$\mathsf{CntAS}(F_1, G_1) \times \cdots \mathsf{CntAS}(F_k, G_k) = \mathsf{CntAS}(F, G).$$

□

3.3.2 Determinism

Propositional model counters utilize *determinism* [55], which involves assigning one of the variables in a formula to either false or true. The number of models of ϕ is then determined as the sum of the number of models in which a variable $x \in \mathsf{Var}(\phi)$ is assigned to false and true. A similar idea can be used for answer set counting using our pair representation as well. To establish the correctness of the determinism employed in our approach, we first introduce two helper propositions: Proposition 1 and 2.

Proposition 1. *For partial assignment τ and program P represented as $(\mathsf{Comp}(P), \mathsf{Copy}(P))$, if $\mathsf{Comp}(P)|_\tau = \emptyset$ and $\emptyset \subset \mathsf{Var}(\mathsf{Copy}(P)|_\tau) \subseteq \mathsf{CopyVar}(P)$, then $\exists L \in \mathsf{Loops}(P)$ s.t. $L \subseteq M^\tau$ and $\tau \not\models \mathsf{LF}(L, P)$.*

Proof. Since $\emptyset \subset \mathsf{Var}(\mathsf{Copy}(P)|_\tau) \subseteq \mathsf{CopyVar}(P)$, there exists a copy variable $x_{i_1}{}' \in \mathsf{Var}(\mathsf{Copy}(P)|_\tau)$ and an implication (simplified after unit propagation) of type 2 of the

form $C_1 \rightarrow x_{i_1}{}'$ in $\mathsf{Copy}(P)|_\tau$, where C_1 is a non-empty conjunction of copy variables. Let $x_{i_2}{}' \in \mathsf{Var}(C_1)$, then there must also exist another implication (simplified after unit propagation) $C_2 \rightarrow x_{i_2}{}'$ in $\mathsf{Copy}(P)|_\tau$, where C_2 is again a conjunction of copy variables. Accordingly, for $x_{i_3} \in \mathsf{Var}(C_2) \setminus \mathsf{Var}(C_1)$, we have another implication of the form $C_3 \rightarrow x_{i_3}{}'$ in $\mathsf{Copy}(P)|_\tau$. Since the number of atoms is bounded, it must be the case that there exists i_k such that there is an implication (simplified) of type 2 $C_k \rightarrow x_k{}'$ such that $C_k \setminus (C_1 \cup C_2 \ldots C_{k-1}) = \emptyset$.

Now, observation $C_k \setminus (C_1 \cup C_2 \ldots C_{k-1}) = \emptyset$ implies existence of an atom set $L = \{x_{i_1}, x_{i_2}, \ldots x_{i_j}\} \subseteq \{x_{i_1}, x_{i_2}, \ldots x_{i_k}\}$ that forms a loop in $\mathsf{DG}(P)$. Given that $\mathsf{Var}(\mathsf{Copy}(P)|_\tau) \subseteq \mathsf{CopyVar}(P)$, we also know that τ assigns a value to every $x \in \mathsf{Var}(\mathsf{Copy}(P)) \cap \mathsf{atoms}(P)$. Furthermore, each of the atoms $x_{i_1}, \ldots x_{i_k}$ must have been assigned 1 by τ. Otherwise, if any x_{i_l} was assigned 0 by τ, then τ would have unit propagated on $\mathsf{Copy}(P)|_\tau$ to $\neg x'_{i_l}$, which contradicts the observation that the copy variables $x'_{i_1}, \ldots x'_{i_k}$ stayed backed in antecedents of implications of type 2 in $\mathsf{Copy}(P)|_\tau$. It follows that atoms in loop L form a subset of atoms assigned 1 by τ.

We have shown above that $\{x_{i_1}, x_{i_2}, \ldots x_{i_j}\}$ constitutes a loop in the positive dependency graph. We now show by contradiction that $\tau \not\models \bigvee_{r \in \mathsf{ExtRule}(L)} \mathsf{Body}(r)$. Indeed, if $\tau \models \bigvee_{r \in \mathsf{ExtRule}(L)} \mathsf{Body}(r)$, let x_{i_t} be $\mathsf{Head}(r)$ for a rule r such that $\tau \models \mathsf{Body}(r)$. In this case, τ must have unit propagated to $\{x'_{i_t}\}$ in $\mathsf{Copy}(P)|_\tau$. This contradicts the fact that the copy variables $x'_{i_1}, \ldots x'_{i_k}$ stayed backed in antecedents of implications of type 2 in $\mathsf{Copy}(P)|_\tau$.

Therefore $\tau \models x_{i_1} \wedge \ldots x_{i_j}$ but $\tau \not\models \bigvee_{r \in \mathsf{ExtRule}(L)} \mathsf{Body}(r)$. This shows that $\tau \not\models \mathsf{LF}(L, P)$. □

Proposition 2. *For partial assignment τ and program P represented as $(\mathsf{Comp}(P), \mathsf{Copy}(P))$, suppose $\tau \not\models \mathsf{LF}(L, P)$, where $L = \{x_1, \ldots, x_k\}$. Then there exists τ^+ such that $\{x_1{}', \ldots, x_k{}'\} \subseteq \mathsf{Var}(\mathsf{Copy}(P)|_{\tau^+})$, $\mathsf{Comp}(P)|_{\tau^+} = \emptyset$ and $\tau \subseteq \tau^+$.*

Proof. As $\tau \not\models \mathsf{LF}(L, P)$, we have $\forall x_i \in L, \tau(x_i) = 1$ and $\forall r \in \mathsf{ExtRule}(L), \tau \not\models \mathsf{Body}(r)$. Let us denote by r' an implication of type 2 corresponding to a rule $r \in \mathsf{ExtRule}(L)$. Then we have $r'|_\tau \neq \emptyset$; moreover, if $\mathsf{Head}(r) = x_i$, then $x_i{}' \in \mathsf{Var}(r|_\tau)$. Since the above observation holds for all $r \in \mathsf{ExtRule}(L)$ and for $x_i \in L$, therefore, $\{x_1{}', \ldots, x_k{}'\} \subseteq \mathsf{Var}(\mathsf{Copy}(P)|_\tau)$. Observe that for every extension τ' of τ that does not assign values to variables in $\{x_1{}', \ldots, x_k{}'\}$, it must be the case that

$\{x_1', \ldots, x_k'\} \subseteq \mathsf{Var}(\mathsf{Copy}(P)|_{\tau'})$. Furthermore, since the set of variables in $\mathsf{Comp}(P)$ does not contain a variable from the set $\{x_1', \ldots, x_k'\}$, therefore, there exists an extension, τ^+, of τ such that $\mathsf{Comp}(P)|_{\tau^+} = \emptyset$ and $\{x_1', \ldots, x_k'\} \subseteq \mathsf{Var}(\mathsf{Copy}(P)|_{\tau'})$. □

We are now ready to state and prove the correctness of determinism employed in our ASP counter:

Proposition 3. *Let program P be represented as (F, G). Then*

$$\mathsf{CntAS}(F, G) = \mathsf{CntAS}(F|_{\neg x}, G|_{\neg x}) + \mathsf{CntAS}(F|_x, G|_x), \quad \textit{for all } x \in \mathsf{atoms}(P) \tag{3.1}$$

$$\mathsf{CntAS}(\bot, G) = 0 \tag{3.2}$$

$$\mathsf{CntAS}(\emptyset, G) = \begin{cases} 1 & \textit{if } G = \emptyset \\ 0 & \textit{if } \mathsf{Var}(G) \subseteq \mathsf{CopyVar}(P) \end{cases} \tag{3.3}$$

Note that if $\mathsf{Comp}(P) = \emptyset$ then either $G = \emptyset$ or $\emptyset \subset \mathsf{Var}(G) \subseteq \mathsf{CopyVar}(P)$.

Proof. The proof comprises the following three parts:

Equation 3.1 applies determinism by partitioning all answer sets of (F, G) into two parts – the answer sets where x is 0 and 1, respectively. Observe that performing unit propagation on (F, G) is valid since $\tau \in \mathsf{AS}(F|_\sigma, G|_\sigma)$ if and only if $\sigma \cup \tau \in \mathsf{AS}(F, G)$, where $\sigma \in 2^{|X|}$, $\tau \in 2^{|\mathsf{atoms}(P) \setminus X|}$, where $X \subseteq \mathsf{atoms}(P)$.

The proof of the first base case in Equation 3.2 is trivial. Each answer set of P conforms to the completion of the program $\mathsf{Comp}(P)$, where, according to the alternative definition of answer sets, $F = \mathsf{Comp}(P)$.

We utilize the helper propositions proved earlier to demonstrate the correctness of the second base case, as outlined in Equation 3.3, which appropriately selects answer sets from the models of completion. First, we show that if there is a copy variable in $\mathsf{Copy}(P)|_\tau$, where $\mathsf{Comp}(P)|_\tau = \emptyset$, then one of the loop formulas of the program is not satisfied by τ. The claim is proved in Proposition 1. Thus, τ cannot be extended to an answer set. Second, we demonstrate that if there is an unsatisfied loop formula under a partial assignment τ_1, then there exists τ_1^+ such that some copy variables are not propagated in $\mathsf{Copy}(P)|_{\tau_1^+}$, where $\mathsf{Comp}(P)|_{\tau_1^+} = \emptyset$ and $\tau_1 \subseteq \tau_1^+$. The claim is established in Proposition 2. Thus, through the method of contradiction,

we can infer that, for an assignment τ, if $\mathsf{Copy}(P)|_\tau = \emptyset$, then τ can be extended to an answer set.

□

3.3.3 Conjoin F and G

Until now, we have represented a program P as a pair of formulas F and G. However, in this subsection, we illustrate that rather than considering the pair, we can regard their conjunction $F \wedge G$, and all the subroutines of model counting algorithms work correctly. First, in Lemma 3.2, we demonstrate that $F \wedge G$ uniquely defines a program (F, G) under arbitrary partial assignments.

Lemma 3.2. *For two assignments τ_1 and τ_2, and given a normal program, $F|_{\tau_1} \wedge G|_{\tau_1} = F|_{\tau_2} \wedge G|_{\tau_2}$ if and only if $F|_{\tau_1} = F|_{\tau_2}$ and $G|_{\tau_1} = G|_{\tau_2}$*

Proof. (i) (proof of 'if part') The proof is trivial.

(ii) (proof of 'only if part') **Proof By Contradiction**. Assume that there is a clause $c \in F|_{\tau_1}$ and $c \notin F|_{\tau_2}$. As $F|_{\tau_1} \wedge G|_{\tau_1} = F|_{\tau_2} \wedge G|_{\tau_2}$ clause $c \in G|_{\tau_2}$. As $c \in F|_{\tau_1}$, c has no copy variable. Assume that clause c is derived from the unit propagation of $\mathsf{Copy}(r)$, i.e., $c = \mathsf{Copy}(r)|_{\tau_2} = a_1' \wedge \ldots \wedge a_k' \wedge b_1 \wedge \ldots \wedge b_m \wedge \neg c_1 \wedge \ldots \wedge \neg c_n \rightarrow x'|_{\tau_2}$, where $\forall i, a_i'$ propagates to 1 and x' propagates to 0, which follows that under assignment τ_2, the atom x is assigned to 0 and $\forall i, a_i$ is assigned to 1. The rule r also belongs to $\mathsf{Comp}(P)$ and both $F|_{\tau_1}$ and $F|_{\tau_2}$ are derived from $\mathsf{Comp}(P)$. Thus, under assignment τ_2, if x is assigned to 0 and each of the a_i's is assigned to 1, then the clause $c \in F|_{\tau_2}$, which must be derived from rule r, so contradiction. □

As a result, it is possible to perform unit propagation on $F \wedge G$ instead of performing unit propagation on F and G separately. Although both formulas F and G are necessary to check the base cases, we can still check base cases by considering the conjunction $F \wedge G$. Checking the first base case (Equation 3.2) is trivial because if an assignment τ *conflicts* on F, then τ conflicts on $F \wedge G$ as well. Additionally, calculating $\mathsf{Var}(F \wedge G)$ suffices to check the second base case (Equation 3.3). The component decomposition part also works with their conjunction because the component decomposition condition $(\mathsf{Var}(F_1) \cup \mathsf{Var}(G_1)) \cap (\mathsf{Var}(F_2) \cup \mathsf{Var}(G_2)) = \emptyset$ is equivalent to $(\mathsf{Var}(F_1 \wedge G_1)) \cap (\mathsf{Var}(F_2 \wedge G_2)) = \emptyset$. Moreover, as we

restrict our decision to $\mathsf{atoms}(P)$, the conjunction $F \wedge G$ does not introduce new conflicts — if a partial assignment τ conflicts on $F \wedge G$, then τ conflicts on F. To summarize, the model counting algorithm correctly computes the answer set count, even when processing the formula $F \wedge G$ instead of processing the two formulas F and G separately.

3.3.4 sharpASP: Putting It All Together

In this subsection, we aim to extend a propositional model counter to an exact answer set counter by integrating the alternative answer set definition, component decomposition (Lemma 3.1), and determinism (Equation 3.1).

Algorithm 1 sharpASP(P)

```
1:  function Counter(φ, CV)                          ▷ modified CNF counter
2:      if φ = ∅ then return 1
3:      else if Var(φ) ⊆ CV then return 0
4:      else if ∅ ∈ φ then return 0
5:      v ← PickNonCopyVar(φ)
6:      for ℓ ← {v, ¬v} do
7:          Count[ℓ] ← 1
8:          comps ← Decomposition(φ|ℓ)
9:          for each c ∈ comps do
10:             if c ∈ Cache then
11:                 Count[ℓ] ← Count[ℓ] × Cache[c]
12:             else
13:                 Count[ℓ] ← Count[ℓ] × Counter(c, CV)
14:             if Count[ℓ] = 0 then
15:                 break
16:     Cache[φ] ← Count[v] + Count[¬v]
17:     return Cache[φ]
18: F ← Comp(P), G ← Copy(P)                          ▷ Algorithm starts here
19: return Counter(F ∧ G, CopyVar(P))
```

The pseudocode for sharpASP is presented in Algorithm 1. Given a non-tight program P, sharpASP initially computes $\mathsf{Comp}(P)$ and $\mathsf{Copy}(P)$ (Line 18 of Algorithm 1) and then calls the adapted propositional model counter Counter, with $\mathsf{Comp}(P) \wedge \mathsf{Copy}(P)$ as the input formula, and $\mathsf{CopyVar}(P)$ as the set of copy variables (Line 19 of Algorithm 1). The model counting algorithm utilizes $\mathsf{CopyVar}(P)$ to check the base cases (Equations (3.2) and (3.3)) of the Equation 3.1.

The Counter differs from the existing propositional model counters mainly in two ways. Firstly, following Equation 3.3, the Counter returns 0 if it encounters a component consisting solely of copy variables (Line 3 of Algorithm 1). Secondly, during *variable branching*, Counter selects variables from Var(Comp(P)) (Line 5 of Algorithm 1). Apart from that, the subroutines of unit propagation, component decomposition (Line 8 of Algorithm 1), and caching[1] (Line 10 of Algorithm 1) within Counter and a propositional model counter remain unchanged.

While sharpASP uses copy variables and copy operations similar to ASProblog, there are notable distinctions between the two approaches. Firstly, sharpASP aims to justify only loop atoms, whereas the ASProblog algorithm aims to justify all founded variables. Our empirical findings underscore that loop atoms constitute a relatively small subset of the founded variables. Consequently, the copy operation of ASProblog introduces more copy variables and logical implications involving copy variables compared to ours. Secondly, the unit propagation techniques employed in ASProblog differ from those used in sharpASP. Specifically, ASProblog performs unit propagation by propagating only the justified literals from a program while leaving the unjustified literals in the residual program. In contrast, sharpASP adheres to the conventional unit propagation technique and employs copy variables to determine whether all atoms are justified.

3.4 Experimental Evaluation

We developed a prototype[2] of sharpASP on top of the existing state-of-the-art model counters (GANAK, D4, and SharpSAT-TD) [147, 149, 188]. We modified SharpSAT-TD by disabling all the preprocessing techniques, as they would no longer preserve answer sets. We use notations sharpASP(STD), sharpASP(G), and sharpASP(D) to represent sharpASP with underlying propositional model counters SharpSAT-TD, GANAK, and D4, respectively. Note that GANAK is a *probabilistic* exact model counter, and we ran GANAK in *non-probabilistic* mode.

We compared the performance of sharpASP with that of the prior state-of-the-art

[1]Model counter stores the count of previously solved subformulas by a caching mechanism to avoid recounting [185].

[2]The implementation is available at https://github.com/meelgroup/sharpASP

	clingo	ASProb	DynASP	aspmc+STD	lp2sat+STD	sharpASP(STD)
Hamil. (405)	230	0	0	167	112	300
Reach. (600)	318	149	2	421	167	463
aspben (465)	321	39	208	252	193	260
Total (1470)	869 (4285)	188 (8722)	210 (8571)	840 (4572)	776 (5082)	1023 (3372)

Table 3.1: The performance comparison of sharpASP vis-a-vis other ASP counters on different problems in terms of number of solved instances and PAR2 scores (within parenthesis).

exact ASP counters: Clingo[3] [91], ASProblog [17], and DynASP [82]. In addition, we utilized two translations from ASP to SAT: (i) lp2sat [32, 71, 127] (ii) aspmc [65, 66], followed by invoking off-the-shelf #SAT solvers. We use notations lp2sat+X and aspmc+X to denote lp2sat and aspmc followed by propositional model counter X, respectively. For sharpASP and #SAT-based tools, we excluded the CNF translation time from the reported runtimes, as it is less than 1 second on average per instance.

Our benchmark suite consists of non-tight programs from the domains of the Hamiltonian cycle and graph reachability problems [17, 133]. We also considered the benchmark set from [65] (designated as aspben). We gathered a total of 1470 graph instances from the benchmark set of [65, 133]. We evaluated only non-tight instances, as tight instances can be efficiently solved by employing Clark completion [51, 71]. The *choice/random* variables in the hamiltonian cycle and aspmc benchmark pertain to graph edges. While the choice variables are associated with graph nodes for the graph reachability problem. The benchmarks and experimental log files are available at https://zenodo.org/records/19665132.

All experiments were carried out on a high-performance computer cluster, where each node consists of AMD EPYC 7713 CPUs running with 128 real cores. The runtime and memory limit were set to 5000 seconds and 8 GB, respectively.

		clingo ($\leq 10^5$) +			
	clingo	ASProb	aspmc +STD	lp2sat +STD	sharpASP (STD)
Hamil. (405)	230	123	167	128	302
Reach. (600)	318	152	418	470	460
aspben (465)	321	278	284	297	300
Total (1470)	869 (4285)	553 (6239)	869 (4310)	895 (4205)	1062 (3082)

Table 3.2: The performance comparison of hybrid counters (combining clingo's enumeration with existing answer set counters) in terms of the number of solved instances and PAR2 scores.

3.4.1 Experimental Results

The performance of our considered counters varies across different computational problems. Our evaluation of their performance, considering both total solved instances and PAR2 scores[4], for each computational problem is detailed in Table 3.1. In the table, the numbers in parentheses in the first column indicate the number of instances of particular problems (e.g. there are 405 instances of hamiltonian cycle problem). The table demonstrates that sharpASP either outperforms or achieves performance on par with existing ASP counters, particularly for the Hamiltonian cycle and graph reachability problems. However, on aspben, the clingo enumeration outperforms other answer set counters.

We observed that clingo demonstrates superior performance, particularly on instances with a limited number of answer sets. Since this observation applies to all non-enumeration based counters in our repertoire, we devised a hybrid counter that combines the strengths of enumeration based counting with that of translation and propositional SAT based counting. Based on data collected from runs of clingo, there is a shift in the runtime performance of clingo when the count of answer sets exceeds 10^5 (within our benchmarks). To ensure that our experiments can be replicated on different platforms, we chose to use an answer set count-based threshold instead of a time-based threshold. Hence, our hybrid counter is structured as follows: it initiates

[3]Clingo counts answer sets via enumeration.

[4]PAR2 [21, 77] is a penalized average runtime that penalizes two times the timeout for each unsolved benchmarks.

enumeration with a maximum of 10^5 answer sets. In cases where not all answer sets are enumerated, the hybrid counter then employs an ASP counter with a time limit of $5000 - t$ seconds, where t is the time spent in clingo. In the latter case (when an ASP counter is employed), it starts from scratch and does not reuse any information obtained during enumeration. The performance of the hybrid counters is tabulated in Table 3.2. In the table, the hybrid counters correspond to last 4 columns that employ clingo enumeration followed by ASP counters. The clingo (2nd column) refers to clingo enumeration for 5000 seconds. The numbers in parentheses in the first column of the table indicate the number of instances of particular problems (e.g. there are 405 instances of hamiltonian cycle problem). The table also demonstrates that the hybrid counter based on sharpASP clearly outperforms competitors by a handsome margin.

A detailed experimental analysis is given in Chapter B.

Chapter 4

Exact Answer Set Counter: sharpASP-$\mathcal{SR}$

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. Like as sharpASP (Chapter 3), this approach introduces an alternative characterization of answer sets that enables efficient reduction while ensuring that intermediate representations remain of polynomial size. Unlike Chapter 3, sharpASP-$\mathcal{SR}$ focuses on disjunctive logic programs. This reduction allows sharpASP-$\mathcal{SR}$ to leverage recent advances in projected model counting. Through extensive experimental evaluation on diverse benchmarks, we demonstrate that sharpASP-$\mathcal{SR}$ significantly outperforms existing counters on instances with large answer set counts.

4.1 Related Work

Answer set counting exhibits distinct complexity characteristics across different classes of logic programs. For normal logic programs, the problem is #P-complete [209], while for disjunctive logic programs, it rises to $\# \cdot$ co-NP-complete [82]. This complexity gap between normal and disjunctive programs highlights that answer set counting for disjunctive logic programs is likely harder than that for normal ones, 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 [166], the same problem for disjunctive logic programs is Σ_2^p-complete [64]. 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 cannot exist unless the polynomial hierarchy collapses [128, 130, 218].

Much of the early research on answer set counting focused on normal logic programs [17, 65, 66, 132]. The methodologies for counting answer sets have evolved significantly over time. Initial approaches relied primarily on enumeration-based techniques [93]. More recent methods have adopted advanced algorithmic techniques, particularly tree decomposition and dynamic programming.

Jakl, Pichler, and Woltran [124] pioneered the application of tree decomposition techniques from #SAT [184] to ASP counting. Their work introduced a fixed-parameter tractable (FPT) algorithm for answer set counting. This approach was later refined by Fichte et al. [75, 82], who developed DynASP, an exact answer set counter optimized for instances with small treewidth.

Subtraction-based techniques have emerged as promising approaches for various counting problems, e.g., MUS counting [27]. In the context of answer set counting, subtraction-based methods were introduced in [74, 112]. 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 [112] developed a method utilizing projected model counting over propositional formulas with projection sets. In a different direction, Fichte et al. [74] 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.

4.2 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 sharpASP (Chapter 3) for normal logic programs. Before presenting the alternative definition of answer sets, we provide a formal definition of the notion of *justification*, which is crucial to understand our technical contribution.

4.2.1 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 [41, 72, 179].

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 [98, 159]. Given an interpretation M s.t. $M \models P$, ASP solvers determine whether some of the atoms in M can be set to false, while satisfying the reduct P^M [156]. We use the notation τ_M to denote the assignment over $\mathsf{atoms}(P)$ corresponding to interpretation M. Furthermore, we say that $x \in \tau_M^+$ (resp. τ_M^-) iff $\tau_M(x) = 1$ (resp. 0).

We define the notion of justification based on the reduct P^M, for each interpretation $M \models P$. While the existing literature typically formulates justification using rule-based or graph-based explanations [72], we propose a model-theoretic definition from the reduct. An atom $x \in M$ is *justified* in M if for each $M' \models P$ and $M' \subseteq M$, it holds that $x \in M'$. It implies that removing x from M would violate 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 $M' \models P^M$; i.e., each atom $x \in M$ is justified. Conversely, an atom $x \in M$ is *not justified* in M if there exists a proper subset $M' \subsetneq M$ 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* [156] — such solvers encode P^M as a set of implications: for each rule $r \in P^M$, $\mathsf{Body}(r) \longrightarrow \mathsf{Head}(r)$, and the check the satisfiability of

$$P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigvee_{x \in \tau_M^+} \neg x,$$

Proposition 4. *For a program P and 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 answer set 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.

Example 4.1. *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:* $\{(p_0 \longrightarrow \neg p_1), (p_1 \longrightarrow \neg p_0), (q_0 \longrightarrow (\neg q_1 \vee w)), (q_1 \longrightarrow (\neg q_0 \vee w)), (w \longrightarrow (p_0 \vee (p_1 \wedge q_1)))\}$ *(see notations from Subsection 2.2.2).*

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* $\{q_1, w\}$.

Consider the following two interpretations over $\mathsf{atoms}(P)$*:*

- $M_1 = \{p_0, w, q_0, q_1\}$*: 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*. We can find 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 the loop atoms of P in the interpretation M. Note that the ASP counter, sharpASP (Chapter 3), also checks justifications for loop atoms in the context of normal logic programs. Specifically, we establish that when $\tau_M \models \mathsf{Comp}(P)$, if any atom in M is not justified, then there must also be some loop atoms in M that is not justified. To verify justifications for 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 5. *For each* $M \subseteq \mathsf{atoms}(P)$ *and* $M \models P$*, 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. Both Propositions 4 and 5 are applied for each interpretation $M \subseteq \mathsf{atoms}(P)$ such that $\tau_M \models \mathsf{Comp}(P)$.

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 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 is not justified in M. For each interpretation $M \models P$, justifying all loop atoms of M is the necessary and sufficient condition to justify all atoms of M. The following lemma formalizes our claim:

Lemma 4.1. *For a given program P and interpretation $M \subseteq \mathsf{atoms}(P)$ such that $\tau_M \models \mathsf{Comp}(P)$,*

1. *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.*
2. *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 UNSAT, then $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigvee_{x \in \tau_M^+} \neg x$ is UNSAT*

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.
Proof of **'1'**:
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 4). Similarly, since B is UNSAT, we know that all loop atoms are justified in M (Proposition 5). Therefore, there must be a non-loop atom, say x_1, that is not justified in M. Since $x_1 \in M$ and $\tau_M \models \mathsf{Comp}(P)$, by 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 $\mathsf{atoms}(P)$. Case (ii) implies that some loop atom is not justified – a contradiction of our premise!

Proof of **'2'**:
This part is the contraposition of our previous claim. □

Example 4.1 (continued)**.** *Consider the following two interpretations over* $\mathsf{atoms}(P)$*:*

- $M_1 = \{p_0, w, q_0, q_1\}$*: 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*. We can find 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*.*

4.2.2 $\mathsf{Copy}(P)$ for Disjunctive Logic Programs

Towards establishing an alternative definition of answer sets for disjunctive logic programs, we now generalize the copy operation used in sharpASP (Chapter 3) 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 $x' \notin \mathsf{atoms}(P)$. We refer to x' as the *copy variable* of x. Similar to sharpASP, 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 $r = a_1 \vee \ldots a_k \leftarrow b_1, \ldots, b_m, \mathsf{not}\ c_1, \ldots, \mathsf{not}\ c_n \in \mathsf{Rules}(P)$ such that $\{a_1, \ldots a_k\} \cap \mathsf{LA}(P) \neq \emptyset$, the implication $\psi(b_1) \wedge \ldots \psi(b_m) \wedge \neg c_1 \wedge \ldots \neg c_n \longrightarrow \psi(a_1) \vee \ldots \psi(a_k)$ 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 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 4.1 (continued)**.** *For the given program P, we have $\mathsf{LA}(P) = \{q_1, w\}$. Thus, $\mathsf{Copy}(P)$ introduces two fresh copy variables $q_1{}'$ and w', and adds 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 in the context of normal programs, Lemma 4.2 formalizes this important relationship in the context of disjunctive logic programs.

Lemma 4.2. *For a given program P and interpretation $M \subseteq \mathsf{atoms}(P)$ 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*

Proof. **The proof of "part 1"**:
Recall that following SAT-based ASP solvers, each rule of P^M can be written as an implication. Thus, P^M can be thought of as a set of implications. First we discuss which implications are left in the subformula $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x$, following unit propagation.

Note that the subformula $\bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x$ includes a unit clause $\neg x$, for each atom $x \in \tau_M^-$ and a unit clause x, each atom $x \in \tau_M^+ \cap \mathsf{LA}(P)$. Since $\tau_M \models \mathsf{Comp}(P)$, the unit propagation does not result in any empty clause. After the unit propagation, all atoms that are assigned to false ($x \in \tau_M^-$) and non loop atoms that are assigned to true ($x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)$) are removed from implications of P^M. Thus, after unit propagation, each implication is of the form: $\beta \longrightarrow \alpha$, where β (antecedent) is either true or a conjunction of loop atoms, and α (consequent) is a disjunction of loop atoms, where those loop atoms are assigned to true in τ_M. Note that no implication left after unit propagation such that its consequent part

is false (or $\bot$); otherwise, we can say that $\tau_M \not\models \mathsf{Comp}(P)$. In summary, after unit propagation, the formula $P^M \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x$ is left with a set of unit clauses and implications, where the implications consists of only loop atoms that are assigned to true in τ_M.

Now we discuss how the implications of $\mathsf{Copy}(P)|_{\tau_M}$ relate to the implications of $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x$. According to the definition of the copy operation, $\mathsf{Copy}(P)$ introduces a type 2 implication for every rule whose head contains at least one loop atom; that is, $\mathsf{Copy}(P)$ introduces a type 2 implication for a rule r if $\mathsf{Head}(r) \cap \mathsf{LA}(P) \neq \emptyset$. Recall that these type 2 implications are similar to the implications discussed above for P^M, except that each of the loop atoms (i.e., x) of rule r is replaced by their corresponding copy atom (i.e., x'). Clearly, no loop atom of P exists in $\mathsf{Copy}(P)$.

The assignment τ_M does not assign the copy variables. The $\mathsf{Copy}(P)|_{\tau_M}$ unit propagates all variables of $\mathsf{Copy}(P)$ except copy variables. Since $\tau_M \models \mathsf{Comp}(P)$, no empty clause is introduced due to the unit propagation. If a loop atom $x \in \mathsf{LA}(P)$ is false in τ_M, then the corresponding copy atom x' will be unit propagated to false in $\mathsf{Copy}(P)|_{\tau_M}$, (type 1 implication). Additionally, no type 1 implication is left in $\mathsf{Copy}(P)|_{\tau_M}$ after the unit propagation. The formula $\mathsf{Copy}(P)|_{\tau_M}$ is a set of unit clauses and implications following unit propagation, where the implications consists of only copy atoms and their corresponding loop atoms are assigned to true in τ_M.

From the discussion so far, we can say that the implications of $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigwedge_{x \in \tau_M^+ \wedge x \notin \mathsf{LA}(P)} x$ left after unit propagation are identical to the implications left from $\mathsf{Copy}(P)|_{\tau_M}$ after unit propagation, except that each loop atom is replaced by their corresponding copy atom in $\mathsf{Copy}(P)|_{\tau_M}$. The satisfiability of both formulas depends on these implications, where the variables of unit clauses are seperated from the variables of those implications in both cases. From the relationship discussed above, following to Proposition 5, the satisfiability checking of $\mathsf{Copy}(P)|_{\tau_M} \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x'$ can be rephrased as to check justification of all loop atoms under the interpretation τ_M. Thus we can say that 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.

The proof of "part 2":

proof of "if part": Following the relationship established in **the proof of part 1** between the implications in $\mathsf{Copy}(P)|_{\tau_M}$ and implications in P^M, the proof of "if

part" follows Lemma 4.1 — the Lemma 4.1 proved that if some atoms of M are not justified (or $P^M \wedge \bigwedge_{x \in \tau_M^-} \neg x \wedge \bigvee_{x \in \tau_M^+} \neg x$ is SAT), then some loop atoms of M are not justified (or $\mathsf{Copy}(P)|_{\tau_M} \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x'$ is SAT). We have already shown that (**part 1**) the satisfiability checking of $\mathsf{Copy}(P)|_{\tau_M} \wedge \bigvee_{x \in \tau_M^+ \wedge x \in \mathsf{LA}(P)} \neg x'$ can be rephrased as to check justification of all loop atoms under the interpretation τ_M. So, the "if part" is proved.

proof of "only if part": Following the relationship established in **the proof of part 1** between the implications in $\mathsf{Copy}(P)|_{\tau_M}$ and implications in P^M, the proof is trivial. If some loop atoms of M are not justified, then some atoms of M are not justified. So, the "only if part" is proved. □

We now integrate Clark's completion, the copy operation introduced above, and the core idea from Lemma 4.2 to propose an alternative definition of answer sets.

Lemma 4.3. *For a given program P and interpretation $M \subseteq \mathsf{atoms}(P)$ 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 4.2, and from the definition of answer sets based on the Gelfond-Lifschitz reduct P^M.

Example 4.1 (continued)**.** *Consider two interpretations $M_1, M_2 \subseteq \mathsf{atoms}(P)$:*

- *$M_1 = \{p_0, w, q_0, q_1\}$, 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 $\mathsf{Copy}(P)|_{\tau_{M_2}} \wedge (\neg {q_1}' \vee \neg w') = (\neg w' \vee {q_1}') \wedge (\neg {q_1}' \vee w') \wedge (\neg w' \vee \neg {q_1}')$ is SAT.*

Our alternative definition of answer sets, formalized in Proposition 4.3, implies that the complexity of checking correctness of an answer set for disjunctive logic programs is in co-NP. In contrast, the definition in sharpASP (see Definition 1), 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 our $\mathsf{Copy}(P)$ has similarities with formulas introduced in [79, 112] for coNP checks.

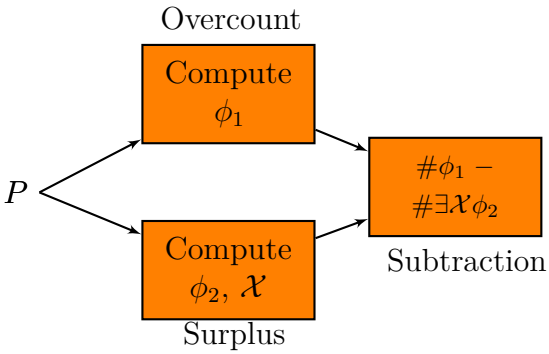


Figure 4.1: The architecture of sharpASP-$\mathcal{SR}$ for a program P.

In the following section, we utilize the definition in Definition 4.3 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.

4.3 Answer Set Counter: sharpASP-$\mathcal{SR}$

We now introduce a subtractive reduction-based technique for counting the answer sets of a disjunctive logic program. 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 [16]; hence reducing answer set counting (a $\# \cdot$ coNP-complete problem) to projected model counting makes sense (the classes #NP and $\# \cdot$ coNP are known to coincide). In contrast, answer set counting of normal logic programs is in #P, and is therefore easier.

At a high level, the proposed subtractive reduction-based counting approach is illustrated in Figure 4.1. 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 Definition 4.3. Finally, the count of answer sets of P is given by $\#\phi_1 - \#\exists\mathcal{X}\phi_2$.

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 $\mathsf{Comp}(P)$ [151]. However, for non tight programs, $\#\mathsf{Comp}(P)$ overcounts $|\mathsf{AS}(P)|$. Therefore, we use

$$\phi_1 = \mathsf{Comp}(P) \tag{4.1}$$

Note that the complexity of $\#\mathsf{Comp}(P)$ lies in $\#\mathsf{P}$.

Counting Surplus (ϕ_2) To count the surplus, we utilize the alternative answer set definition presented in Definition 4.3. 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; i.e., $\mathsf{CV}' = \{x' | x \in \mathsf{LA}(P)\}$ and $\mathsf{CV}^\star = \{x^\star | x \in \mathsf{LA}(P)\}$. To compute the surplus, we define the formula $\phi_2(\mathsf{atoms}(P), \mathsf{CV}', \mathsf{CV}^\star)$ as follows:

$$\begin{aligned}\phi_2(\mathsf{atoms}(P), \mathsf{CV}', \mathsf{CV}^\star) = \mathsf{Comp}(P) \wedge \mathsf{Copy}(P)' \wedge \mathsf{Copy}(P)^\star \\ \wedge \bigwedge_{x \in \mathsf{LA}(P)} (x' \longrightarrow x^\star) \wedge \bigvee_{x \in \mathsf{LA}(P)} (\neg x' \wedge x^\star)\end{aligned} \tag{4.2}$$

Lemma 4.4. *The number of models of $\mathsf{Comp}(P)$ that are not answer sets of P can be computed as $\#\exists \mathsf{CV}', \mathsf{CV}^\star\ \phi_2(\mathsf{atoms}(P), \mathsf{CV}', \mathsf{CV}^\star)$, where the formula ϕ_2 is defined in Equation 4.2.*

Proof. From ϕ_2 (Equation 4.2), 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 $\exists x \in \mathsf{LA}(P), \sigma(x') < \sigma(x^\star)$[1]. Let M be the corresponding interpretation over $\mathsf{atoms}(P)$ 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 $\sigma(x) = \mathsf{true}$, while after setting the copy variables x' to false, the formula $\mathsf{Copy}(P)_{|\tau_M}$ is still satisfied. According to Definition 4.3, we can conclude that $M \notin \mathsf{AS}(P)$. As a result, $\#\exists \mathsf{CV}', \mathsf{CV}^\star\ \phi_2(\mathsf{atoms}(P), \mathsf{CV}', \mathsf{CV}^\star)$ counts all interpretations that are not answer sets. □

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

Theorem 2. *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 4.1 and 4.2. Furthermore, both ϕ_1 and ϕ_2 can be computed in time polynomial in $|P|$.*

Proof. The proof of the part $|\mathsf{AS}(P)| = \#\phi_1 - \#\exists\mathcal{X}\phi_2$, where $\mathcal{X} = CV' \cup CV^\star$, follows from Lemma 4.4. Initially, $\#\phi_1$ overcounts the number of answer sets, while Lemma 4.4 establishes that $\#\exists\mathcal{X}\phi_2$ counts the surplus from $\#\phi_1$. Thus, the subtraction determines the number of answer sets of P.

For a program P, $\mathsf{Comp}(P)$, $\mathsf{Copy}(P)$, and $\mathsf{LA}(P)$ can be computed in time polynomial in the size of P. Additionally, for a program P, $|\mathsf{LA}(P)| \leq |\mathsf{atoms}(P)|$. Thus, we can compute both ϕ_1 and ϕ_2 in polynomial time in the size of P. □

Now recall to subtractive reduction definition (Section 2.9), 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. 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 2 as sharpASP-$\mathcal{SR}$. While sharpASP-$\mathcal{SR}$ shares similarities with the answer set counting approach outlined in [112], there are key differences between the two techniques. First, instead of counting the number of models of Clark completion, the technique in [112] 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 [112] introduces copy (referred to as *duplicate* variable) variables for every variable in the program. Third, sharpASP-$\mathcal{SR}$ focuses on generating 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 [112] 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 [111]. 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.

4.4 Experimental Evaluation

We developed a prototype of sharpASP-$\mathcal{SR}$ (available at https://github.com/meelgroup/SharpASP-SR), by leveraging existing projected model counters. Specifically, we employed GANAK [188] as the underlying projected model counter, given its competitive performance in model counting competitions.

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 [93], (ii) DynASP v2.0 [82], and (iii) Wasp v2 [11]. ASP solvers Clingo and Wasp count answer sets via enumeration. We were unable to baseline against ASP counters such as aspmc+#SAT [66], lp2sat+#SAT [126, 127], sharpASP (Chapter 3), and iascar [74], 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 [112], a comparison against their approach was not possible.

Our benchmark suite comprised non-tight disjunctive logic program instances previously used to evaluate disjunctive answer set solvers. We considered only non-tight instances because the tight instances can be counted via Clark Completion [7, 151]. Our benchmarks span diverse computational problems, including: (i) 2QBF [133], (ii) strategic companies [156], (iii) *preferred* extensions of *abstract argumentations* [85], (iv) pc configuration [80], and (v) minimal diagnosis [97]. The benchmarks were sourced from cs.engr.uky.edu[2], abstract argumentation competitions, ASP competitions [96] and from [133]. Following recent work on disjunctive logic programs [6], we generated additional non-tight disjunctive answer set programs using the generator implemented by [13]. We also incorporated non-tight disjunctive logic programs from research on computing *minimal trap spaces* [201]. The benchmarks and experimental log files are available at https://zenodo.org/records/19665845.

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 answer set counting.

[2]Benchmark Problems for ASP Systems: http://cs.engr.uky.edu/ai/benchmarks.html

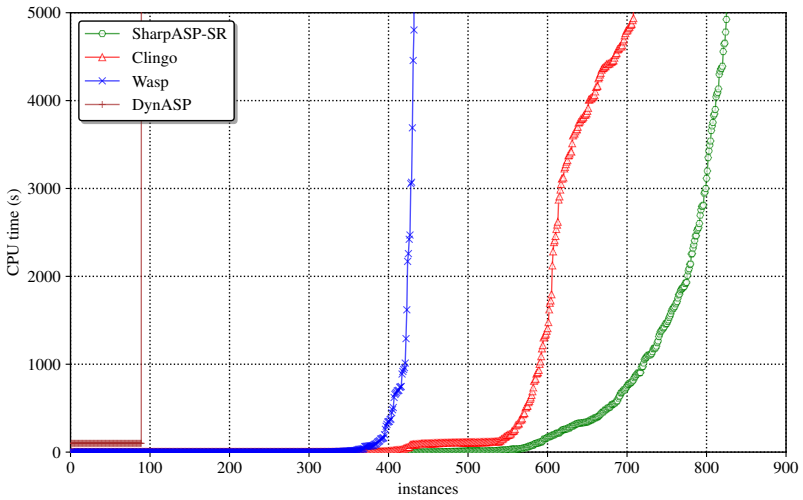


Figure 4.2: The runtime performance of sharpASP-$\mathcal{SR}$ vis-a-vis other ASP counters.

4.4.1 Experimental Results

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

Table 4.1: The performance of sharpASP-$\mathcal{SR}$ vis-a-vis existing disjunctive answer set counters, based on 1125 instances.

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

Table 4.2: The performance comparison of hybrid counters (combining clingo's enumeration with existing answer set counters), based on 1125 instances.

sharpASP-$\mathcal{SR}$ demonstrated significant performance improvement across the benchmark suite, as evidenced in Table 4.1. For comparative analysis, we present both the number of solved instances and PAR2 scores [21], 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.

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

Table 4.3: 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 shortforms SA, and cl+SA denote sharpASP-$\mathcal{SR}$, and the hybrid counter Clingo ($\leq 10^4$) + sharpASP-$\mathcal{SR}$, respectively.

The cactus plot in Figure 4.2 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.

4.4.2 Ablation Study

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 techniques, following the experimental evaluation of sharpASP. This hybrid approach first employs Clingo enumeration (maximum 10^4 answer sets) and switches to alternative counting techniques if all answer sets are not enumerated. In the latter case, the counter restarts from scratch and does not reuse any information obtained during enumeration. Within our benchmark instances, a noticeable shift was observed on Clingo's runtime performance when the number of answer sets exceeds 10^4. The results of hybrid counters are shown in Table 4.2. In the table, the hybrid counters correspond to last 3 columns that employ clingo enumeration followed by ASP counters. The clingo (2nd column) refers to clingo enumeration for 5000 seconds. The table demonstrates that the hybrid counter based on sharpASP-$\mathcal{SR}$ significantly outperforms baseline approaches.

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 in Table 4.3. In the table, the second column ($\sum$) indicates the number of instances within each range of $|\mathsf{LA}(P)|$. The results 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 marked decrease in the solved instances count.
Some detailed experimental analysis is given in Chapter C.

Chapter 5

Approximate Answer Set Counter: ApproxASP

Unlike sharpASP (Chapter 3) and sharpASP-$\mathcal{SR}$ (Chapter 4), which focus on exact counting, we now turn to the problem of approximate answer set counting. We introduce ApproxASP, the first scalable technique for approximate answer set counting, offering rigorous (ε, δ) guarantees. ApproxASP supports both normal and disjunctive answer set programs. Technically, ApproxASP extends the XOR-based hashing framework, originally developed for propositional model counting, to the ASP setting. As demonstrated in the development of approximate model counters, designing a scalable counter requires careful engineering of the underlying theory solver to handle XOR constraints. To address this, we present the first ASP solver capable of natively handling XOR constraints using Gauss-Jordan elimination (GJE). Our experimental results show that ApproxASP outperforms existing answer set counters, thereby positioning it as the tool of choice in answer set counting.

5.1 Related Work

The answer set counting is #P-complete for normal logic program [209], while for disjunctive logic programs, it rises to # $\cdot$ co-NP-complete [82]. Fichte et al. [82] established an implementation, called DynASP, for counting the number of answer sets. It is based on dynamic programming on tree decompositions and theoretically performs well if the treewidth is low. In practice, a decomposition heuristic is required that can find a tree decomposition of low width fast. Due to theoretical restrictions, an instance can easily have high treewidth simply if it contains a large rule. Since most encodings are not treewidth aware [111], the approach has certain

theoretical limitations. A line of work established compilation techniques that transform ASP programs into SAT instances [34, 65, 66, 126, 127]. However, unless the considered program is tight [23, 139], there can be an exponential overhead [151, 161]. Nonetheless, the compilation can be used to transform an input instance and use the existing model counter, e.g., [47].

ASP Solving with Parity Constraints. An extension to the ASP solver Clingo [67], called *xorro*, introduces a variety of parity constraints into ASP solving, relying on ASP encodings of parity constraints or theory propagators [87]. In contrast, our implementation relies on a dedicated high performant implementation of Gauss-Jordan elimination.

5.2 Partitioning and Sampling Counts

In the course of approximate model counting, we divide the search space by so-called parity constraints (see Subsection 2.3.2 for definition). Intuitively, a parity constraint makes sure that atoms occurring in it occur only in an even or odd number in an answer set of the program. Corollary 2.1 shows that adding simple constraints to a program will not introduce new answer sets, which is crucial to split the search space. We could use this property and include a counter for each of the model and integrity constraints. The integrity constraints ensure that the counter is even (resp. odd) for even (resp. odd) parity. In fact, the ASP-Core-2 language already allows for representing parity constraints using aggregate rules [42, 69]. The ASP-Core-2 language would allow to represent it in the form of:

`:-#count{1:  p(1)} = N, N\2 !=1.` and

`:-#count{X: p(X), X>1} = N, N\ 2 !=0.` resp.

Note that N\ 2 is N modulo 2. Unfortunately, such a construction is quite impractical for two reasons. First, it would require us to add far too many auxiliary variables for each new parity constraint, and second, we would suffer from the known inadequacy to scale due to grounding [67]. Hence, we aim for an incremental "propagator"-based implementation when solving parity constraints [87].

First, we formally define parity constraints and show basic properties needed for partitioning the search space. The meaning of parity constraints follows previous

works on the topic [67]. For atoms a_1 and a_2, we denote the *exclusive-or* between these two atoms by $a_1 \oplus a_2$. Let M be a set of atoms. Then, M *satisfies* $a_1 \oplus a_2$ if $M \cap \{a_1\} \cup M \cap \{a_2\} \neq \emptyset$ and $M \cap \{a_1, a_2\} \neq \{a_1, a_2\}$, i.e., *either* a_1 *or* a_2 is in M; but not both. Generalizing to n distinct atoms $a_1, \ldots, a_n$, we obtain an exclusive-or constraint r of the form $(((a_1 \oplus a_2) \ldots) \oplus a_n)$ by applying $\oplus$ consecutively. Then, r is satisfied if and only if $|M \cap \mathsf{atoms}(r)|$ is odd and we refer to r as *odd parity constraint*. Due to associativity, we simply write $a_1 \oplus a_2 \oplus \ldots \oplus a_n$. Analogously, an *even parity constraint*, XOR *constraint* for short, is of the form $a_1 \oplus \ldots \oplus a_n \oplus \mathsf{true}$ and satisfied if and only if $M \cap \mathsf{atoms}(r)$ is even. Parity constraints can in principle contain literals. But since we can easily rewrite them, we omit such definitions. For example, constraint $\neg a_1 \oplus a_2$ is equivalent to $a_1 \oplus a_2 \oplus \mathsf{true}$ and $\neg a_1 \oplus \neg a_2$ is equivalent to $a_1 \oplus a_2$, i.e., pairs of negated literals cancel parity inversion. We extend the definitions on answer set programs from the preliminaries above to include parity constraints. Following standard ASP syntax, one would expect that parity constraint rules are of the form $\leftarrow a_1 \oplus \ldots \oplus a_n \oplus \top$ and $\leftarrow a_1 \oplus \ldots \oplus a_n$, respectively. However, to simplify the presentation, we write parity constraint rules as above and call them simply *parity constraint*. We let a *parity-constrained program* P be a set of rules or parity constraints and we denote by $\mathsf{Rules}(P)$ the rules and by $C(P)$ the parity constraints of P.

Semantics for Search Space Partitioning. A straight-forward approach to include parity constraints in ASP would be to extend the GL-reduct [98] as follows. The *reduct* of a parity-constrained program P under a set M of atoms is the program $P_M := \{\, \mathsf{Head}(r) \leftarrow \mathsf{Body}(r)^+ \mid r \in \mathsf{Rules}(P), \mathsf{Body}(r)^- \cap M = \emptyset \,\} \cup C(P)$. However, our next example shows that extending the popular ASP-definition and including parity constraints into the GL-reduct from above is not enough.

Example 5.1. *Consider the program* $P = \{a \leftarrow \sim b; b \leftarrow \sim c; c \leftarrow \sim a; a \oplus b \oplus c\}$, *which contains an odd constraint. Furthermore, take the set* $M = \{a, b, c\}$. *It is easy to see that* M *satisfies each rule of* P_M, *in particular, the parity constraint. Now, while* $N = \emptyset$ *satisfies each rule* $r \in \mathsf{Rules}(P)_M$, *the set* N *does not satisfy the parity constraints. Hence,* M *would be an answer set of the parity-constrained program* P. *In other words, adding a parity constraint to a program might yield a*

new, counterintuitive answer set.

From the previous example, we can conclude that simply extending the GL-reduct is not enough when introducing parity-constraints into answer set programming (for approximate counting). Hence, we suggest the following definition:

Definition 2 (Answer Sets of Parity-Constrained Programs)**.** *A set* $M \subseteq \mathsf{atoms}(P)$ *is an* answer set *of* P *if (i)* M *satisfies each rule* $r \in \mathsf{Rules}(P)$ *(in particular, also the parity constraints) and (ii)* M *is the minimal model of* $(\mathsf{Rules}(P))^M$*. Again, we denote by* $\mathsf{AS}(P)$ *the set of all answer sets of* P*.*

Observation 2 shows that parity-constraints, according to our definition, do not introduce new answer sets, which is crucial to search space partitioning.

Observation 2. *Let* P *be a program and* X *be a set of parity constraints. Then,* $\mathsf{AS}(P \cup X) \subseteq \mathsf{AS}(P)$*.*

Proof. Assume that $M \subseteq \mathsf{atoms}(P)$ is an answer set of $P \cup X$. Since M satisfies every non-parity rule in $P \cup X$, M satisfies every rule $r \in \mathsf{Rules}(P)$. Since parity constraints do not occur in the program $(\mathsf{Rules}(P))^M$ by Definition 2, for every $M \subseteq \mathsf{atoms}(P)$ satisfying P and $N \subsetneq M$ we have that N satisfies $(\mathsf{Rules}(P))^M$. In other words, a set N that shows that M is not minimal with respect to the program $(\mathsf{Rules}(P))^M$, still shows that M is not minimal when parity constraints are added. In turn, we cannot accidentally introduce new answer sets by adding the constraints, which establishes the observation. □

Well-Defined for Unfounded Set Characterization. Since we have seen certain pitfalls above and we aim for using parity constraints in an ASP solver such as clasp, we quickly review the behavior of parity constraints under an unfounded set characterization (Subsection 2.2.4). Therefore, we extend the definitions for programs: a set $M \subseteq \mathsf{atoms}(P)$ is an *answer set* of a *parity-constrained program* P *under the unfounded set semantics* if (UP1) M satisfies $\mathsf{Comp}(P)$ (see subsection 2.2.2), (UP1a) M satisfies $C(P)$, and (UP2) no loop contained in M is unfounded. We denote by $\mathsf{AS}_{\mathsf{uf}}(P)$ the set of all answer sets of P according to the unfounded set definition from above. Since parity constrains only apply to the model part, we

observe that answer sets remain the same for both definitions. So, parity constraints only restrict answer sets, but never introduce new ones.

Observation 3. *Let P be a program and X a set of parity constraints. Then,* $\mathsf{AS}_{\mathsf{uf}}(P \cup X) \subseteq \mathsf{AS}_{\mathsf{uf}}(P)$.

Proof. The main argument is that propositional logic is monotonic and we conclude that Condition (UP1a) will only remove models. However, we can also show the following stronger statement $\mathsf{AS}_{\mathsf{uf}}(P \cup X) \subseteq \mathsf{AS}_{\mathsf{uf}}(P)$. Assume that M is an answer set of $P \cup X$. By Condition (UP1) we have that M satisfies $\mathsf{Comp}(P)$. Since $P = \mathsf{Rules}(P \cup X)$, M satisfies $\mathsf{Comp}(P)$ and Condition (U1) is trivially satisfied. Since for every rule $r \in C(X)$ we have that $\mathsf{Head}(r) = \emptyset$, we can conclude that the positive dependency digraphs remains the same, i.e., $\mathsf{DG}(P) = \mathsf{DG}(P \cup X)$. Thus, the loops of $P \cup X$ and P are the same. In consequence, since Condition (UP2) holds for M, it allows us to conclude that Condition (U2) is also satisfied. Finally, from the folkore that propositional logic is monotonic and hence Condition (UP1a) will only remove models from $P \cup X$, we conclude that our claim $\mathsf{AS}_{\mathsf{uf}}(P \cup X) \subseteq \mathsf{AS}_{\mathsf{uf}}(P)$, which in turn establishes the observation. □

Algorithm 2 $\mathsf{ApproxASP}(P, I, \varepsilon, \delta)$

Input: Program P, independent support I, tolerance ε, confidence δ

$C \leftarrow \{\}$ ▷ Sampled counts
$n_c \leftarrow 2$ ▷ Number of Cells
$\mathrm{p} \leftarrow 1 + \lceil 9.84 \times (\frac{\varepsilon}{1+\varepsilon}) \times (1 + \frac{1}{\varepsilon})^2 \rceil$ ▷ Pivot
▷ Try to enumerate $p + 1$ many answer sets projected to I
$S = \textbf{Enum-k-AS}(P, \mathrm{p} + 1, I)$
if $|S| \leq \mathrm{p}$ **then return** $|S|$
for $i \leftarrow 0$ to $\lceil 17 \cdot \log_2 \frac{3}{\delta} \rceil$ **do**
    ▷ Divide search space and sample at most $p + 1$ solutions
    $(n_c, s) \leftarrow \mathsf{DivideNSampleCell}(P, I, \mathrm{p} + 1, n_c)$
    ▷ Keep estimate if search space was actually divided
    if $n_c > 0$ **then**
        $C \leftarrow C \cup \{n_c \times s\}$
return median(C)

Algorithm 3 DivideNSampleCell

Input: Program P, Independent support I, pivot p, cell count n_c

1: ▷ Randomly choose $|I|$ many constraints $X \in \mathcal{H}_{xor}(|I|, |I|-1)$ of length $|I-1|$ over variables
2: $X \leftarrow$ **ChooseXOR**$(I, |I|, |I|-1)$
3: $S =$ **Enum-k-AS**$(P \cup X, \mathrm{p}, I)$ ▷ Enumerate p answer sets
4: **if** $|S| \geq$ p **then return** $(0, 0)$
5: $m \leftarrow$ **LogASPSearch**$(P, I, X, \mathrm{p}, \log_2 n'_c)$ ▷ Estimate size of cells
6: $Y \leftarrow$ **Choosek**(X, m) ▷ Pick k XOR constraints
7: $S \leftarrow$ **Enum-k-AS**$(P \cup Y, \mathrm{p}, I)$ ▷ Enum answer sets for one cell
8: **return** $(2^m, |S|)$

5.2.1 Approximate Counting

The central idea for approximate counting is to employ hash functions for sampling the search space uniformly [171]. First, one partitions the set Sol(I) (the solution set, see notations in Section 2.4) of an input instance I into roughly equally small cells. Then, one picks a random cell, counts the number s of solutions in the cell, and scales s by the number c of cells to obtain an ε-approximate estimate of the count c. A priori, we do not know the distribution of the set Sol(I) of solutions. Hence, we have to hash without knowledge of the distribution of the solutions, i.e., partition Sol(I) into cells uniformly and independently. Interestingly, this can be resolved by using a k-wise independent hash function [103] (see definitions in Section 2.3). In the context of approximate counting techniques, the most exploited hash family is $\mathcal{H}_{xor}$ (see Subsection 2.3.2) [191]. We use an already evolved algorithm from the case of propositional satisfiability [47, 48] and lift it to ASP.

The algorithm is given in Algorithms 2 and 3. The ApproxASP algorithm takes as input an ASP program P, an independent support I for P, a tolerance ε ($0 < \varepsilon \leq 1$), and a confidence δ ($0 < \delta \leq 1$). First, sampled counts and the number of cells are initialized. Then, in Line 3, we compute a pivot p that depends on ε to determine the chosen value of the size of a small cell. Then, it checks if the input program P has at least p+1 answer sets projected to I (**Enum-k-AS**); otherwise, we are trivially done and return the answer set count $|S|$. The function **Enum-k-AS**(P, p, I) enumerates up to p answer sets of P projected to I. Subsequently, the algorithm continues and calculates a value $r := 17 \cdot \log_2 {}^3\!/\!_\delta$ that determines how often we need to sample for the requested confidence δ. The value of r originates in a probabilistic

analysis [47]. Next, we divide the search space and sample a cell at most r times using the function DivideNSampleCell. If the attempt to split into at least 2 cells worked, represented by a non-zero number of cells, we store the count and estimate the total count by multiplying the number of cells by the answer set count within the sampled cell. The final estimate of the count returned by ApproxASP is the median of the estimates stored in C, computed in Line 13.

Function DivideNSampleCell takes as input an ASP program P, an independent support I, a pivot p, and the number n'_c of cells from the previous round. It returns the number of constructed cells from the chosen XOR constraints and an ε-approximate estimate of the count of the answer sets of program P. The function randomly chooses $|I|$ many constraints $X \in \mathcal{H}_{xor}(|I|, |I|-1)$ of length $|I-1|$ over variables I. Then it checks (lower bound) whether the program together with the XOR-constraint $(P \cup X)$ has at most p answer sets by simply enumerating at most p answer sets of $P \cup X$. Intuitively, if the cell contains more answer sets than the pivot, we have selected XOR-constraints unfavorable as we cannot count the number of answer sets in the considered cell. We proceed with finding the *right* number of XOR-constraints to take from the chosen XOR-constraints by the function **LogASPSearch**, which has an underlying idea that the partitioned cell is large enough, but not too large. The **LogASPSearch** function uses a binary search-like procedure to compute the right number m of XOR constraints, following prior work on approximate model counting [48]. Then, we choose m of the XOR-constraints from X in Line 6 using the function **Choosek**(X, m). We enumerate at most p answer sets projected to I for the program P together with the chosen m XOR-constraints. Finally, we return the number of cells, which is 2^m, and the number of answer sets of the sampled cell. Overall, the number of satisfiability calls is $\mathcal{O}(\mathrm{p} \times \log |C|)$. The seemingly magic constants — such as the pivot value p in Line 3, iteration count r, and the use of the median estimates (in Line 13) — originate in a probabilistic analysis using Chernoff and Chebyshev inequalities for counting sets [48].

Interestingly, our algorithm relies only on the property that adding constraints does not increase the number of solutions, which is needed for Line 5 to 7. In fact, in Observation 2, we already established the property and the remaining construction directly works due to results on propositional satisfiability where the construction is based on basics of probability theory that works for sets $S \subseteq \{0,1\}^n$ [48].

5.3 Implementation Details

Most of the computation of approximate model counting is involved in satisfiability checking. Similar to ApproxMC, the consistency checking of ApproxASP consists of two sub-routines, the answer set solving and the XOR solving. For the answer set solving, ApproxASP uses Clingo as the underlying solver. The ASP solver addresses the answer set computation while invoking the XOR solving subroutine.

Everardo et al. [67] discussed the impact of eagerly or lazily translating XOR constraints into ASP encodings. An eager translation is practically infeasible due to an exponential blowup in the number of constraints. Alternatively, lazy translation relies on non-trivial XOR solving techniques in ASP using *theory propagators*. We go beyond this approach and implement a dedicated, sophisticated theory propagator. We extend Clingo with full XOR solving using an implementation of Han and Jiang's Gauss-Jordan elimination (GJE) [109], which has been integrated into the state-of-the-art SAT solver CryptoMiniSat [194]. There, Clingo searches for an answer set while providing the XOR subroutine a (partial) assignment such that the GJE deduces its satisfiability. An answer set is reached if both sub-routines are satisfied.

5.3.1 Gauss-Jordan Elimination

Han and Jiang [109] proposed a framework for Gauss-Jordan elimination representing a set of XOR constraints as matrix $\mathbb{M} = [A|b]$, where A is an $m \times n$ matrix coupled with a parity constraint b, where m is the number of XOR constraints and $n = |\mathsf{atoms}(P)|$. Each row and column in A represents an XOR and a variable, respectively. This method suits perfectly to remove linearly dependent equations while incrementally updating $\mathbb{M}$.

The framework uses the *well-known* two-watched literals scheme per XOR constraint, where one watched literal is called *basic* and the other one is *non-basic*. Basic variables are on the diagonal of the matrix in *reduced row-echelon* form. Instead of removing a column when the corresponding variable is assigned, GJE computes the state of the XOR constraint after one of the watched literals is assigned. The state of an XOR constraint is exactly one of the following: (i) conflicted, (ii) propagated, (iii) satisfied, and (iv) new watch variable assigned.

An XOR constraint is propagating if all except one literal are assigned. It is

satisfied or conflicted, respectively, if all of its variables are assigned, and the truth value of the literals satisfies or dissatisfies, respectively, the XOR-constraint. If an XOR constraint has more than one unassigned literal, it must watch a new literal, i.e., there is no determination of the satisfiability of the XOR.

To compute a conflict and propagation clause, we start with an empty clause and scan forward the corresponding row of the matrix. For all set bits, we insert the corresponding variable into the clause with the negative or positive phase if it is assigned false or true, respectively. In the case of propagation, we insert the unassigned variable with the appropriate phase, such that the number of positive phases is odd or even if the parity of the XOR is even or odd, respectively.

Example 5.2. *Let*

$$\begin{bmatrix}1 & 0 & 1 & 1 & 1\end{bmatrix}$$

be the binary row-vector representation of an XOR constraint, where the columns correspond to variables x_1, x_2, x_3, x_4, x_5*, and the parity is* 1*.*

- *If the assignment is* $\begin{bmatrix}? & 0 & 1 & 1 & 0\end{bmatrix}$*, (? denotes an unassigned value), then the XOR constraint is propagating and the clause is* $x_1 \vee \neg x_3 \vee \neg x_4 \vee x_5$*.*
- *If the assignment is* $\begin{bmatrix}0 & 0 & 1 & 0 & 1\end{bmatrix}$*, then the XOR constraint is conflicting, and the conflict clause is* $x_1 \vee \neg x_3 \vee x_4 \vee \neg x_5$*.*

5.3.2 Further Optimizations in xor Solving

We utilize some heuristics further to speed up the XOR solving of ApproxASP.

Heuristic 3. *If the state of an XOR is satisfied, the state will be unchanged as long as the ASP solver does not backtrack.*

For each XOR, we keep a separate bit to store whether the XOR is satisfied or not. If the state of an XOR is satisfied, we set the bit. Later, we do not compute the state as long as the corresponding bit is set. However, if the ASP solver backtracks, we clear the corresponding bit.

Heuristic 4. *If both basic and non-basic variables of an XOR are unassigned, then the state of the XOR is undetermined.*

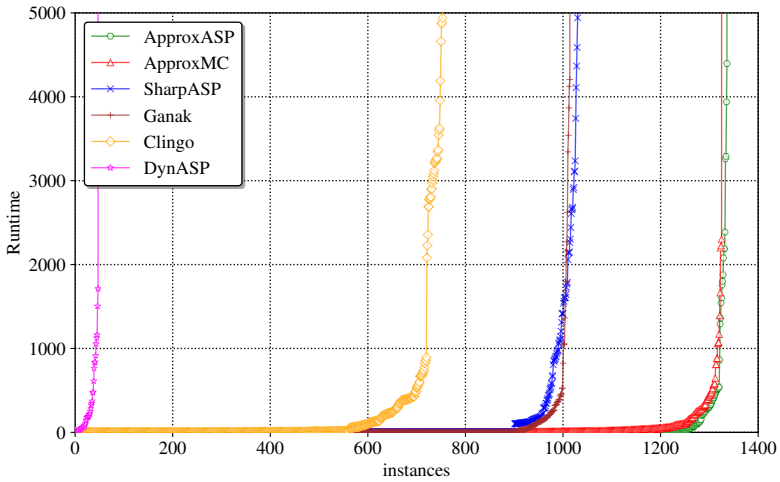


Figure 5.1: Runtime of various tools for normal programs.

If an XOR is undetermined, it is unnecessary to compute its state. To reduce the unnecessary computation, we skip computing the state of an XOR if both of its basic and non-basic variables are unassigned.

5.4 Experimental Evaluation

We conducted a preliminary experimental evaluation to assess the runtime performance of ApproxASP and the quality of its approximation. ApproxASP implementation is available at https://github.com/meelgroup/ApproxASP. For the evaluation, we consider the following questions:

RQ1 How does ApproxASP compare to existing systems?

RQ2 Does ApproxASP output approximate counts that are indeed between $(1+\varepsilon)^{-1}$ and $(1+\varepsilon)$ of the exact counts?

Environment. All experiments were carried out on a high-performance computer cluster, where each node consists of an 2xE5-2690v3 CPUs running with 2x12 real cores and 96GB of RAM. The runtime was limited to 5000 seconds, and the memory limit was 4 GB. We follow standard guidelines for empirical evaluations [81, 148].

Baselines. We selected existing systems that allow for counting answer sets, namely, DynASP [82], Clingo [91], Ganak [193], ApproxMC [191]. Clingo can count

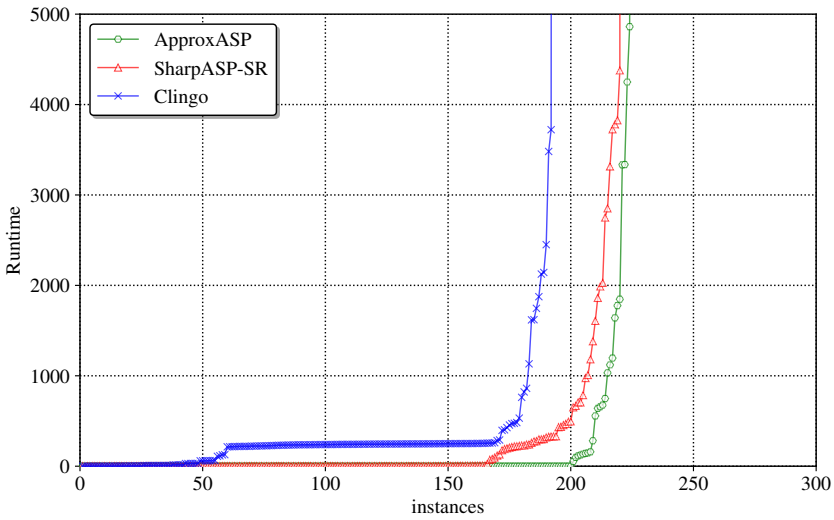


Figure 5.2: The runtime comparison of different counters on disjunctive instances.

answer sets by enumeration. We evaluated the exact answer set counter sharpASP (Chapter 3) only on normal programs. We also evaluated the subtractive reduction-based answer set counter, sharpASP-$\mathcal{SR}$ (Chapter 4) for disjunctive programs. Since sharpASP-$\mathcal{SR}$ employs projected model counters, we did not evaluate normal programs with sharpASP-$\mathcal{SR}$. Note that Ganak and ApproxMC take CNF formulas as input. We experimented with both CNF translations — aspmc [65] and lp2sat [32, 127], but the lp2sat-based translation appeared to perform better for approximate counting. We ran GANAK with cache size bounded to 2000 MB and compute the independent support [192] of the SAT instance for both ApproxMC and ApproxASP. In line with previous works in approximate counting, we set $\varepsilon = 0.8$ and $\delta = 0.2$, for both ApproxMC and ApproxASP.

Instances. We divide our benchmarks into (i) normal programs and (ii) disjunctive programs. For normal programs (i), we chose from different well-known graph problems encoded into ASP programs, where we selected counting variants for the vertex cover, independent set, dominating set, graph reachability, and r-arborescence problem. For disjunctive programs (ii), we used the projected model counting problem on 2QBFs and *strategic companies*, which were used in evaluating sharpASP-$\mathcal{SR}$ (Chapter 4). We used a benchmark set of 256 disjunctive programs. The benchmark set and experimental log files are available at https://zenodo.org/records/19665703.

		Clingo	DynASP	Ganak	ApproxMC	SA	ApproxASP
Normal	#Solved	753	47	1014	1325	1030	**1336**
	PAR-2	5082	9697	3274	1196	3213	**1129**
		Clingo	DynASP	Ganak	ApproxMC	SA-SR	ApproxASP
Disj.	#Solved	192	0	0	0	221	**224**
	PAR-2	2720	10000	10000	10000	1560	**1364**

Table 5.1: The runtime performance comparison of Clingo, DynASP, Ganak, ApproxMC, sharpASP(SA), sharpASP-$\mathcal{SR}$ (SA-SR) and ApproxASP on all considered instances. The numbers in parentheses indicate the number of instances.

5.4.1 Experimental Results

Analysis of RQ1. We demonstrate the experimental results for normal programs in Figure 5.1. The plot shows the runtimes for each counter, and a point (x, y) on the plot indicates that x instances took less than or equal to y seconds to solve. Out of the 1500 instances, ApproxASP solved 1336 instances, where ApproxMC managed to solve 1325 instances. The Figure 5.2 shows the cactus plot of ApproxASP and other counters for disjunctive problems.

Table 5.1 shows the performance evaluation of Clingo, DynASP, Ganak, ApproxMC, and ApproxASP on all instances. The first row shows the total number of instances, the second row shows the number of instances solved by each counter, and the third row presents the PAR-2 score[1]. The time spent in translating ASP to SAT and calculating the independent support is negligible, which is < 1s on average.

For normal programs, ApproxASP solved more instances than ApproxMC. For disjunctive problems, ApproxASP solved 224 instances and sharpASP-$\mathcal{SR}$ solved 221 instances, out of 256 instances. While ApproxASP solved more instances than sharpASP-$\mathcal{SR}$, we observe slightly different behavior than on normal programs. The gap between the runtime performance of ApproxASP and sharpASP-$\mathcal{SR}$ is small. However, this is not entirely surprising, since we use trivial independent support (all atoms of the program) for disjunctive problems.

[1]PAR-2 score [77] is penalized average runtime that assigns a runtime of two times the timeout (without "not solved" status) for each benchmark not solved by a tool.

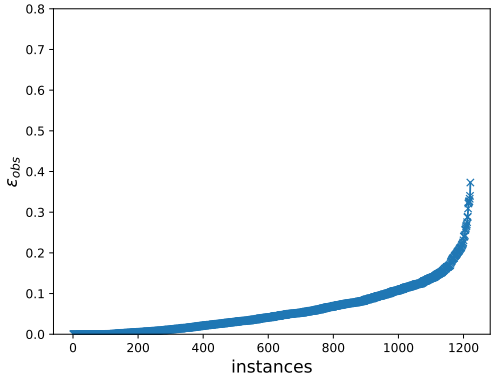


Figure 5.3: Visualization of the tolerance computed from the ApproxASP estimate.

Analysis of RQ2. We compare the number of solutions computed by ApproxASP with solutions returned by exact counters to assess the quality of the approximation. The results of our comparison are shown in Figure 5.3. An experimental evaluation with *xorro* is presented in Chapter D.

We observe that, for all instances, ApproxASP produces estimates within the tolerance of the counts returned by Clingo or exact answer set counters. Moreover, we compute the observed tolerance ε_{obs}, which is defined as $\max(\mathsf{Count}/|\mathsf{AS}(P)| - 1, |\mathsf{AS}(P)|/\mathsf{Count} - 1)$, where Count is the output given by ApproxASP. We observe a maximum value of $\varepsilon_{obs} = 0.3729$ and the arithmetic mean of $\varepsilon_{obs} = 0.0592$ across all instances, while the theoretical guarantee is 0.8. In summary, ApproxASP outputs counts within $[(1+\varepsilon)^{-1}, (1+\varepsilon)]$ ratio of the exact number of answer sets for all instances.

Part II

Answer Set Counting Applications

In this part, we explore the applications of efficient answer set counting techniques, based on the following three publications:

- **[KM2023]** Mohimenul Kabir and Kuldeep S. Meel. "A Fast and Accurate ASP Counting Based Network Reliability Estimator." *LPAR*, vol. 94, pp. 270-287. 2023
- **[KTP$^+$2025]** Mohimenul Kabir, Van-Giang Trinh, Samuel Pastva, and Kuldeep S. Meel. "Scalable Counting of Minimal Trap Spaces and Fixed Points in Boolean Networks." *CP 2025*
- **[KM2024]** Mohimenul Kabir and Kuldeep S. Meel. "On Lower Bounding Minimal Model Count." *Theory and Practice of Logic Programming*, no. 4, 586-605. 2024 (**Best Paper Award of ICLP 2024**)

First we discuss how answer set counting can be exploited in network reliability estimation in Chapter 6 (from [KM2023]). Subsequently, we demonstrate how answer set counting can be exploited to address counting problems in Boolean Networks in Chapter 7 (from [KTP$^+$2025]). Finally, we discuss how answer set counters and systems can be modified to count minimal models of a Boolean formula in Chapter 8 (from [KM2024]).

Chapter 6

Network Reliability Estimation

We present RelNet-ASP, an ASP counting based approach to network reliability estimation. RelNet-ASP takes in a graph $G = (V, E)$, a probability function over edges W, terminal nodes s and t, and computes an estimate of reliability $r(G, s, t, \mathsf{W})$ (ref. Subsection 2.5.1). Recall from Subsection 2.5.1, the reliability $r(G, s, t, \mathsf{W})$ is defined as the sum of the probabilities of all (s, t)-connected subgraphs of G. More specifically, RelNet-ASP reduces the network reliability estimation into answer set counting problem. Our empirical evaluation demonstrates that RelNet-ASP significantly outperforms prior state-of-the-art approaches when accounting for accuracy and runtime performance.

6.1 Related Work

Valiant [209] initiated the complexity study of the network reliability problem and showed that the problem is #P-complete. The exact techniques are often based on the enumeration of *cut sets* or *path sets* to account for disjoint events [1] or the usage of *recursive* or *online decompositions* of disjoint events [181, 208].

Monte Carlo (MC) techniques are often employed to design approximate techniques; a straightforward design of MC techniques struggles to scale due to *rare event situations*. To address the aforementioned limitation, researchers have proposed variations of MC, including multilevel splitting [102], generalized splitting [37], permutation MC-based Lomonosov's Turnip [99], splitting sequential MC [208], subset simulation [219], among others. The *stopping rule algorithm* [53] is an approximation algorithm offering PAC guarantees based on *Sequential analysis*. Dagum et al. [53] proposed a new algorithm based on the stopping rule algorithm by improving its sample size. The *gamma Bernoulli approximation scheme* [116] can calculate network

reliability with PAC guarantees and improve the running time of [53] by utilizing the *central limit theorem* and achieving a lower order on the running time. Subsequently, Huber [117] proposed a two-stage algorithm that combines the gamma Bernoulli approximation scheme and the algorithm of [53] to improve the sample size further.

Advanced sampling techniques, e.g., line sampling and variance reduction method [43], employing graphical models, offer guarantee-less approaches to quantify network reliability. Another line of research involves statistical learning theories with numerical simulation, which offers techniques for network reliability estimation [118]. Furthermore, specialized techniques borrowed from data mining, such as hierarchical and unsupervised spectral clustering, combined with sampling, provide valuable insights into reliability and risk assessment [73, 106, 216].

Another related line of work has focused on the problem of network unreliability, i.e., to estimate $1 - r(G, s, t, \mathsf{W})$. Paredes et al. [59] proposed a CNF model counting-based PAC-style framework for network unreliability estimation. It is worth remarking that PAC-approximation of $1 - r(G, s, t, \mathsf{W})$ cannot be used to derive PAC-approximation of $r(G, s, t, \mathsf{W})$. Akin to CNF-based techniques, the network reliability can be computed via *plausibility reasoning* [78] on ASP; however, this technique performs well particularly on lower treewidth encodings and most encodings do not offer a lower treewidth [111].

Answer Set Counters. The complexity study shows that ASP counting is #·co-NP-complete [78], while the complexity drops to #P for *normal* programs. Various techniques exist for ASP counting, including *unfounded set detection* in propositional model counters [17, 132], dynamic programming on *tree decomposition* [82], *level ranking* of loop atoms [127] and *cycle breaking* from positive dependency graphs [65]. Another line of work combines hashing-based techniques with ASP solving to yield PAC-style ASP counting [133] (Chapter 5).

6.2 Methodology

The high-level overview of RelNet-ASP is presented in Figure 6.1. The RelNet-ASP consists of three phases, which we discuss below:

- (Phase 1) Given a graph G and terminal nodes (s and t), RelNet-ASP generates

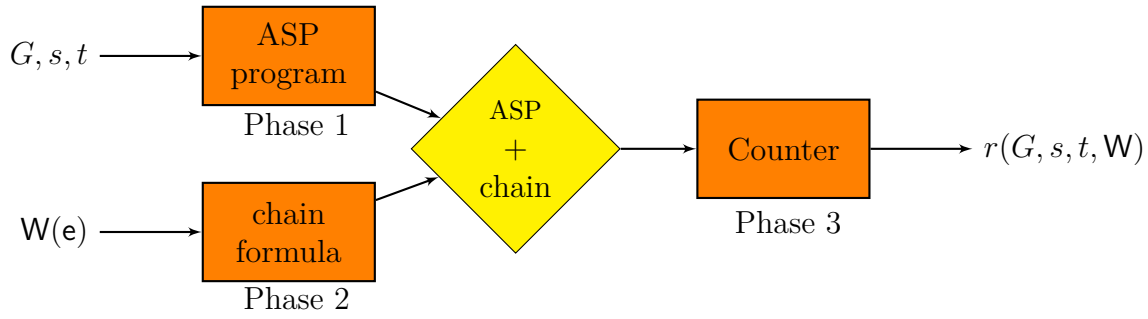


Figure 6.1: The architecture of RelNet-ASP.

an ASP program $\mathcal{P}_{G,s,t}$ such that the answer sets of $\mathcal{P}_{G,s,t}$ correspond one-to-one to the (s,t)-connected subgraphs of G.

- (Phase 2) We encode the arbitrary edge probabilities W into ASP rules by relying on the notion of chain formulas (see Section 2.6); the notation Ch^e denotes the resulting ASP rules for arbitrary edge $e \in E$.
- (Phase 3) RelNet-ASP relies on a state-of-the-art ASP counter to compute (exact/approximate) estimate of $|\mathsf{AS}(\mathcal{P}_{G,s,t} \wedge \bigwedge_{e \in E} \mathsf{Ch}^e)|$, which is normalized to return the desired estimate of $r(G, s, t, \mathsf{W})$.

Following earlier work [59], we assume the edge probability is of the form $\mathsf{W}(\mathsf{e}) = \frac{k}{2^m}$, where k and m are integers and $0 < k < 2^m$.

6.2.1 Generate ASP Program (Phase 1)

We assume that the input graph G is unweighted, i.e., each edge has the identical probability of $\frac{1}{2}$. RelNet-ASP generates a program $\mathcal{P}_{G,s,t}$ such that $|\mathsf{AS}(\mathcal{P}_{G,s,t})| = |\mathsf{Subgraph}(G, s, t)|$. In program $\mathcal{P}_{G,s,t}$, capital letters (e.g., X, Y) denote arbitrary objects and small letters denote specific objects (e.g., s, t). The program $\mathcal{P}_{G,s,t}$ uses a set of atoms to represent the nodes and edges of the graph, the activity of an edge, and the reachability of a node from the source node s. These atoms may include:

- node(X) and edge(X,Y) represent that X is a node and (X,Y) is an edge in the input graph, respectively
- source(X) and target(Y) represent X and Y are the source and target nodes, respectively

- in(X,Y) represents that edge(X,Y) is active
- reached(X) represents that node X is reachable from the source node s

Using these atoms described above, the program $\mathcal{P}_{G,s,t}$ uses a set of rules to represent the structure of the graph, the activity of edges, and the reachability of nodes from the source node s. The encoding of Listing 6.1 presents $\mathcal{P}_{G,s,t}$. In the encoding, the atoms in(X,Y) act as *choice variables*[1] of ASP [88]. The node s is the source node and the program starts graph traversal starting from node s. The atom reached(s) is true initially, i.e., the source node s is reachable from the s. Then the program transitively determines the reachable nodes from node s (Listing 6.1, lines 3 and 4). These rules state that if one of the endpoints of an active edge is reachable from s, then the other endpoint of the active edge is reachable from s. Finally, the program adds a constraint that target node t must be reachable from node s (Listing 6.1, line 6). The constraint ensures that only the connected subgraphs that include a path from node s to node t are considered as a part of the answer sets.

```
1  % transitive definition of reachability
2  reached(X) ← source(X).
3  reached(Y) ← in(X,Y), reached(X).
4  reached(X) ← in(X,Y), reached(Y).
5  % target node must be reachable
6  ← target(X), not reached(X).
```

Listing 6.1: Program $\mathcal{P}_{G,s,t}$

K-terminal Reliability The ASP program Listing 6.1 can be used if there are k terminal nodes, where $k > 2$. In this case, we consider one of the terminal nodes as the source node and the remaining $k-1$ nodes as target nodes. The constraint in line 6 of Listing 6.1 would validate that all $k-1$ target nodes will be reachable from the source node. The constraint would be represented by a rule that states that $\forall t, t$ is a target, and the atom reached(t) must be true.

[1] In unweighted cases, the atom in(X,Y) is true (i.e., edge (X,Y) is active) with 0.5 probability.

Algorithm 4 ChainASP($\phi_{k,m}$)

Input: $\phi_{k,m}$
Output: ASP program Ch

```
1: for each (a, b) ∈ T(φ_{k,m}) do
2:     if b is of form p ∨ q then
3:         Ch.add(Rule(a ← p))
4:         Ch.add(Rule(a ← q))
5:     else if b is of form p ∧ q then
6:         Ch.add(Rule(a ← p, q))
7: return Ch
```

6.2.2 Chain Formula in ASP (Phase 2)

In this subsection, RelNet-ASP encodes arbitrary edge probabilities relying on the concept of chain formula. Finally, RelNet-ASP generates an ASP program for a given chain formula such that the number of answer sets of the program is proportional to the corresponding weight of the chain formula.

From Chain Formula to ASP Recall that the chain formula is discussed in Chapter 2 and a chain formula $\phi_{k,m}$ can be viewed as a set of equivalences $\mathsf{T}(\phi_{k,m})$ (see section 2.6). Given a chain formula, Algorithm 4 of RelNet-ASP derives an ASP program Ch. For each equivalence of $\mathsf{T}(\phi_{k,m})$, Algorithm 4 introduces at most two ASP rules. Recall from the chain formula definition, for each equivalence $(a, b) \in \mathsf{T}(\phi_{k,m})$, the part b is either a conjunction or disjunction of two atoms. For each equivalence of form: $(a, p \vee q) \in \mathsf{T}(\phi_{k,m})$ (see notations from Section 2.6), Algorithm 4 introduces two rules: $\mathsf{Rule}(a \leftarrow p)$ and $\mathsf{Rule}(a \leftarrow q)$ to Ch. For each equivalence of form: $(a, p \wedge q) \in \mathsf{T}(\phi_{k,m})$, Algorithm 4 introduces one rule: $\mathsf{Rule}(a \leftarrow p, q)$ to Ch.

There are two interesting characteristics of the Ch program. First, program Ch is a tight program. More specifically, the positive dependency graph of Ch has directed edges from b_i to t_i and from t_{i+1} to t_i, where $i \in [1, m-2]$. It follows that the positive dependency graph of Ch is acyclic; thus, the rules of Ch form a tight logic program. Second, as per the construction of Algorithm 4, the Clark completion of Ch corresponds to $\mathsf{T}(\phi_{k,m})$. As $\phi_{k,m}$ has k satisfying assignments, $\mathsf{T}(\phi_{k,m}) \wedge \{t_1 \leftrightarrow 1\}$ has exactly k solutions. Conversely, $\mathsf{Rewrite}(\phi_{k,m}, m, e) \wedge \{t_1 \leftrightarrow 0\}$ has exactly $2^m - k$ solutions. As the program Ch is tight, the answer sets of Ch correspond to the

satisfying assignments of $\mathsf{T}(\phi_{k,m})$. It follows that $\mathsf{Ch} \wedge \mathsf{Rule}(\leftarrow \mathsf{not}\ t_1)$ has exactly k answer sets, whereas $\mathsf{Ch} \wedge \mathsf{Rule}(\leftarrow t_1)$ has $2^m - k$ answer sets.

Encoding Edge Probabilities in RelNet-ASP RelNet-ASP employs the chain formula concept in ASP (as outlined in Algorithm 4) to encode arbitrary edge probabilities. More specifically, RelNet-ASP constructs an ASP program Ch^e, for an edge $e \in \mathsf{Edge}(G)$. If an edge $e = (a, b)$ has a probability $\mathsf{W(e)} = \frac{k}{2^m}$, RelNet-ASP uses Algorithm 4 to generate an ASP program Ch^e, which deals with the edge probability $\frac{k}{2^m}$ by introducing k new answer sets when in(a,b) is true, i.e., edge(a,b) is active and $2^m - k$ new answer sets when in(a,b) is false, i.e., edge(a,b) is not active.

Example 6.1. *Consider an edge e such that* $\mathsf{W(e)} = \frac{5}{2^3}$. *Recall from Example 2.2:* $\phi_{k,m}(b_1, b_2, b_3) = (b_1 \vee (b_2 \wedge b_3))$. $\mathsf{T}(\phi_{k,m})$ *is the conjunction of the equivalences:* $\{t_2 \leftrightarrow (b_2 \wedge b_3),\ t_1 \leftrightarrow (b_1 \vee t_2)\}$. *From* $\mathsf{T}(\phi_{k,m})$, *the program* Ch^e *consists of the following rules:* $\{\mathsf{Rule}(t_2 \leftarrow b_2, b_3), \mathsf{Rule}(t_1 \leftarrow b_1), \mathsf{Rule}(t_1 \leftarrow t_2)\}$. *Finally,* $\mathsf{Ch}^e \wedge \mathsf{Rule}(\leftarrow \mathsf{not}\ t_1)$ *has* 5 *answer sets and* $\mathsf{Ch}^e \wedge \mathsf{Rule}(\leftarrow t_1)$ *has* 3 *answer sets.*

6.2.3 Reduction to ASP Counting (Phase 3)

Algorithm 5 of RelNet-ASP reduces the network reliability estimation problem to an ASP counting problem. The algorithm takes in a weighted graph G and terminal nodes s and t, and generates a chain formula augmented ASP program $\mathcal{Q}$. Initially, $\mathcal{Q}$ equals to $\mathcal{P}_{G,s,t}$ and num is 0. For each edge $e = (X, Y) \in \mathsf{Edge}(G)$ where $\mathsf{W(e)} = \frac{k}{2^m}$, the algorithm invokes Algorithm 4, which outputs an ASP program Ch^e (Algorithm 5, line 5). Then the algorithm adds a new rule: $\mathsf{Rule}(\mathsf{in(X,Y)} \leftarrow t_1^e)$ to Ch^e, augments $\mathcal{Q}$ with Ch^e, and increments num by m (Algorithm 5, lines 7 and 8). The program $\mathcal{Q}$ considers the probability of each of the edges; more specifically, $\mathcal{Q}$ is the conjunction of $\mathcal{P}_{G,s,t}$ and $\bigwedge_{e \in \mathsf{Edge}(G)} \mathsf{Ch}_e$. Finally, the algorithm returns the tuple ($\mathcal{Q}$, num). If Algorithm 5 returns ($\mathcal{Q}$, num), then $r(G, s, t, \mathsf{W})$ is $|\mathsf{AS}(\mathcal{Q})|$ divided by 2^{num}. As $r(G, s, t, \mathsf{W})$ is a constant factor of $|\mathsf{AS}(\mathcal{Q})|$, an ASP counter with (ε, δ) guarantees offers (ε, δ) guarantees on network reliability estimation.

Algorithm 5 ProcessProb(G, s, t, W)

Input: G, s, t, W
Output: Chain formula augmented ASP program

1: $\mathcal{Q} \leftarrow \mathcal{P}_{G,s,t}$
2: $\mathsf{num} \leftarrow 0$
3: **for each** $e = (X, Y) \in \mathsf{Edge}(G)$ **do**
4: compute k, m such that $\mathsf{W(e)} = \frac{k}{2^m}$
5: $\mathsf{Ch}^e \leftarrow \mathsf{ChainASP}(\phi_{k,m})$
6: $\mathsf{Ch}^e.\mathsf{add}(\mathsf{Rule}(\mathsf{in(X,Y)} \leftarrow t_1^e))$
7: $\mathcal{Q}.\mathsf{add}(\mathsf{Ch}^e)$
8: $\mathsf{num} \leftarrow \mathsf{num} + m$
9: **return** $(\mathcal{Q}, \mathsf{num})$

6.3 Theoretical Analysis

In this section, we establish the correctness of our methodology. To this end, Lemma 6.1 establishes the correctness of Phase 1. Finally, we combine Lemma 6.1 and Lemma 6.2 to prove the main theorem (Theorem 5).

Lemma 6.1. *Given a unweighted graph G, source and target nodes s and t, respectively,* $|\mathsf{Subgraph}(G, s, t)| = |\mathsf{AS}(\mathcal{P}_{G,s,t})|$.

Proof. As the probability of a graph is associated with its edges, we represent a subgraph using the set of active edges it contains. We use the notation $\tau_{\downarrow \mathsf{in}}$ to refer the set of $\mathsf{in}(X, Y)$ atoms such that $\mathsf{in}(X, Y) \in \tau$ or the set of active edges under answer set τ. We use the shortforms $\mathsf{in}/2$ and $\mathsf{reached}/1$ to refer to the set of active edges and reachable nodes from the source node s, respectively. More specifically, an arbitrary set of $\mathsf{in}/2$ atoms $\{\mathsf{in}(X, Y) | (X, Y) \in \mathsf{Edge}(G)\}$ refers to a subgraph of G where the corresponding edges $\mathsf{edge}(X, Y)$ are active. We consider the following statements to prove the lemma.

S1 If $\tau \in \mathsf{AS}(\mathcal{P}_{G,s,t},)$, then τ corresponds to a (s, t)-connected subgraph of G

S2 If $\tau_1, \tau_2 \in \mathsf{AS}(\mathcal{P}_{G,s,t},)$ and $\tau_{1\downarrow \mathsf{in}} = \tau_{2\downarrow \mathsf{in}}$, then $\tau_1 = \tau_2$

S3 Each subgraph has a unique set of active edges.

S4 Each (s, t)-connected subgraph of G corresponds to a unique answer set of $\mathcal{P}_{G,s,t}$

Proof of **S1**. For an answer set $\tau \in \mathsf{AS}(\mathcal{P}_{G,s,t},)$, we construct a unique $(s,\ t)$-connected subgraph G' of the input graph G. Let G' be the subgraph of G such that $\mathsf{Node}(G') = \mathsf{Node}(G)$ and $(X, Y) \in \mathsf{Edge}(G')$ if and only if $\mathsf{in}(X, Y) \in \tau$. More specifically, according to Listing 6.1, $\mathsf{reached}(s) \in \tau$ and $\mathsf{reached}(t) \in \tau$. For a node $X \in \mathsf{Node}(G)$, if $\mathsf{reached}(X) \in \tau$, then node X is reachable in subgraph G' from source node s because Listing 6.1 computes the reachable nodes from source node s transitively and the active edges of G' are same as the corresponding active edges of answer set τ. Thus, the subgraph G' is $(s,\ t)$-connected.

Proof of **S2**. Proof by contradiction. Assume that there are two answer sets τ_1 and τ_2 such that $\tau_{1\downarrow\mathsf{in}} = \tau_{2\downarrow\mathsf{in}}$ and $\tau_1 \neq \tau_2$. Without loss of generality, assume that $\mathsf{reached}(i_1) \in \tau_1 \setminus \tau_2$. Let's say $x_k = \mathsf{reached}(i_1)$ (initially $k = 1$). There is a rule r_k such that $\mathsf{Head}(r_k) = \mathsf{reached}(i_1)$, $\tau_1 \models \mathsf{Body}(r_k) = \mathsf{in}(i_1, i_2), \mathsf{reached}(i_2)$ and $x_{k+1} = \mathsf{reached}(i_2) \notin \tau_2$ because if $\mathsf{reached}(i_2) \in \tau_2$ then $\mathsf{reached}(i_1) \in \tau_2$. Note that the same conclusion $\mathsf{reached}(i_2) \in \tau_2$ holds if $\mathsf{Body}(r_k) = \mathsf{in}(i_2, i_1), \mathsf{reached}(i_2)$. Similarly, for x_{k+1}, there is another rule r_{k+1}. Thus, the atom set of $\{x_i\}$ is an infinite sequence of $\mathsf{reached}/1$ atoms. If each of x_i is distinct, then it contradicts that $\tau_1 \setminus \tau_2$ has at most $|\mathsf{atoms}(P)|$ atoms. Otherwise, $x_i = x_j$ for some $i < j$, which follows that there is an *unfounded set* between the atom set of $\{x_i, \ldots, x_j\}$, which is a contradiction. Thus, there is no such x_k atom.

Proof of **S3**. The statement follows from the definition of subgraph.

Proof of **S4**. We can prove this using construction. Recall that each subgraph is a set of $\mathsf{in}(X, Y)$ atoms. For each $(s,\ t)$-connected subgraph G', we have an assignment τ over $\mathsf{atoms}(\mathcal{P}_{G,s,t})$ and show that τ maps to a unique answer set of $\mathcal{P}_{G,s,t}$.

Given an $(s,\ t)$-connected subgraph G', initially we have $\tau = \{\mathsf{in}(X, Y) | (X, Y) \in \mathsf{Edge}(G')\} \cup \{\mathsf{source}(s), \mathsf{target}(t)\}$. We start graph traversal on G' from source node s and $\tau = \tau \cup \{\mathsf{reached}(s)\}$. We determine the reachable nodes from node s transitively, i.e., if one of the endpoints of an edge e of G' is reachable from node s, then the other endpoint of e is also reachable from node s. If a node $X \in \mathsf{Node}(G')$ is reachable from source node s, then we append $\mathsf{reached}(X)$ to τ. Upon finishing the graph traversal on G', we get all reachable nodes from source node s under subgraph G'.

Note that the subgraph G' is (s, t)-connected, so $\mathsf{reached}(t) \in \tau$.

The assignment τ satisfies all rules of $\mathcal{P}_{G,s,t}$ because we traverse G' transitively. The assignment τ satisfies the Clark completion of $\mathcal{P}_{G,s,t}$. Now there are two cases to consider: either τ satisfies $\mathsf{LF}(\mathcal{P}_{G,s,t})$ or τ does not satisfy the loop formula $\mathsf{LF}(\mathcal{P}_{G,s,t}, L)$, where L is one of the loops of $\mathsf{DG}(\mathcal{P}_{G,s,t})$. We show that τ is an answer set of $\mathcal{P}_{G,s,t}$ by proving that τ satisfies $\mathsf{LF}(\mathcal{P}_{G,s,t})$. For the purpose of contradiction, assume that there is a loop L in $\mathsf{DG}(\mathcal{P}_{G,s,t})$ such that $\tau \not\models \mathsf{LF}(\mathcal{P}_{G,s,t}, L)$. It is easy to verify that loop L consists of $\mathsf{reached}/1$ atoms. Let $L = \{\mathsf{reached}(i_1), \ldots \mathsf{reached}(i_k)\}$. Therefore, $\tau \models \bigwedge_{a \in L} a$ and $\tau \not\models \bigvee_{r \in \mathsf{ExtRule}(L)} \mathsf{Body}(r)$ (see definition from Subsection 2.2.3). But the condition contradicts to our graph traversal technique because we traverse the subgraph G' transitively. So, there is no such loop L, which follows that τ is an answer set of $\mathcal{P}_{G,s,t}$. □

To summarize, each (s, t) corresponds to a unique answer set of $\mathcal{P}_{G,s,t}$ and each answer set of $\mathcal{P}_{G,s,t}$ corresponds to a unique (s, t)-connected subgraph. Thus, the Lemma 6.1 is proved.
The following lemma is useful to prove the Theorem 5.

Lemma 6.2. *Given a graph G, terminal nodes s, t, edge probabilities W, if Algorithm 5 returns $(\mathcal{Q}_{G,s,t,\mathsf{W}}, \mathsf{num})$ and $e = (a, b) \in \mathsf{Edge}(G)$ is a weighted edge with $\mathsf{W}(\mathsf{e}) = \frac{k}{2^m}$, then*

$$|\mathsf{AS}(\mathcal{Q}_{G,s,t,\mathsf{W}})| = k \times |\mathsf{AS}(\mathcal{Q}_{G/e,s,t,\mathsf{W}} \setminus \mathsf{Ch}^e)| + (2^m - k) \times |\mathsf{AS}(\mathcal{Q}_{G\setminus e,s,t,\mathsf{W}} \setminus \mathsf{Ch}^e)|$$

Proof. There is *exactly one* rule $r = \mathsf{Rule}(\mathsf{in}(a, b) \leftarrow t_1^e) \in \mathcal{Q}$ such that $\mathsf{Head}(r) = \mathsf{in}(a, b)$. If M is an answer set of $\mathcal{Q}_{G,s,t,\mathsf{W}}$, from Clark completion semantics, we have that $t_1^e \in M$ implies $\mathsf{in}(\mathrm{a,b}) \in M$. Similarly, if t_1^e does not belong to an answer set M, then $\mathsf{in}(\mathrm{a,b})$ does not belong to M.

$\mathsf{atoms}(\mathsf{Ch}_e) \cap \mathsf{atoms}(\mathcal{Q}_{G,s,t,\mathsf{W}} \setminus \mathsf{Ch}_e) = \{\mathsf{in}(a, b)\}$. Following the above discussion on atoms $\mathsf{in}(a, b)$ and t_1^e, we can write the following equation on $|\mathsf{AS}(\mathcal{Q}_{G,s,t,\mathsf{W}})|$:

$$\begin{aligned}|\mathsf{AS}(\mathcal{Q}_{G,s,t,\mathsf{W}})| = {} & |\mathsf{AS}(\mathsf{Ch}_e \wedge \mathsf{Rule}(\leftarrow t_1^e))| \times |\mathsf{AS}(\mathcal{Q}_{G,s,t,\mathsf{W}} \setminus \mathsf{Ch}^e \wedge \mathsf{Rule}(\leftarrow \mathsf{in}(a, b)))| \\ & + |\mathsf{AS}(\mathsf{Ch}_e \wedge \mathsf{Rule}(\leftarrow \mathsf{not}\ t_1^e))| \times |\mathsf{AS}(\mathcal{Q}_{G,s,t,\mathsf{W}} \setminus \mathsf{Ch}^e \wedge \mathsf{Rule}(\leftarrow \mathsf{not}\ \mathsf{in}(a, b)))|\end{aligned}$$

The *set minus* operation considers a program as a set of rules and removes the corresponding rules from a program. $\mathcal{Q}_{G,s,t,\mathsf{W}} \wedge \mathsf{Rule}(\leftarrow \mathsf{in}(a, b))$ implies that $\mathsf{in}(a, b)$

is false, i.e., edge (a, b) is removed from the input graph G (see subsection 2.5.2). On the other hand, $\mathcal{Q}_{G,s,t,\mathsf{W}} \wedge \mathsf{Rule}(\leftarrow \mathsf{not}\ \mathsf{in}(a, b))$ implies that $\mathsf{in}(a, b)$ is true, i.e., edge (a, b) is active in the input graph G, which is known as edge contraction in graph theory (see subsection 2.5.2). Thus, we have the following equation:

$$|\mathsf{AS}(\mathcal{Q}_{G,s,t,\mathsf{W}})| = (2^m - k) \times |\mathsf{AS}(\mathcal{Q}_{G\setminus e,s,t,\mathsf{W}} \setminus \mathsf{Ch}^e)| + k \times |\mathsf{AS}(\mathcal{Q}_{G/e,s,t,\mathsf{W}} \setminus \mathsf{Ch}^e)|$$

□

We are now ready to state the main theorem of our work.

Theorem 5. *Given a graph G, terminal nodes s, t, edge probabilities W, if Algorithm 5 returns $(\mathcal{Q}_{G,s,t,\mathsf{W}}, \mathsf{num})$, then $r(G, s, t, \mathsf{W}) = \frac{|\mathsf{AS}(\mathcal{Q}_{G,s,t,\mathsf{W}})|}{2^{\mathsf{num}}}$.*

Proof. We use induction on the number of edges $|\mathsf{Edge}(G)|$.
Base case: The theorem holds for $|\mathsf{Edge}(G)| = 0$ because if there is no edge then there is no (s,t)-connected subgraph. If there is no edge, then $|\mathsf{AS}(\mathcal{Q}_{G,s,t,\mathsf{W}})| = |\mathsf{AS}(\mathcal{P}_{G,s,t})| = 0$. So, $r(G, s, t, \mathsf{W})$ is 0. For $|\mathsf{Edge}(G)| = 1$, assume that $e_1 = (a, b) \in \mathsf{Edge}(G)$ and $\mathsf{W}(\mathsf{e_1}) = \frac{k}{2^m}$. There are two cases to consider: (i) nodes s and t are the two endpoints of e_1 and (ii) otherwise. The case (ii) is trivial because the reliability is zero. In case (i), the two endpoints a and b are reachable when edge e_1 is active, i.e., in(a,b) is true; and according to the Clark completion, we have that $t_1^{e_1}$ is true. It follows that $|\mathsf{AS}(\mathcal{Q}_{G,s,t,\mathsf{W}})| = k$ and the network reliability is calculated as $r(G, s, t, \mathsf{W}) = \frac{|\mathsf{AS}(\mathcal{Q}_{G,s,t,\mathsf{W}})|}{2^m} = \frac{k}{2^m}$.
Induction step: Assume that the theorem is true for $|\mathsf{Edge}(G)| \leq i$ (induction hypothesis). We proof that the theorem holds for $|\mathsf{Edge}(G)| = i + 1$. Assume that $e_{i+1} = (a, b) \in \mathsf{Edge}(G)$ and $\mathsf{W}(\mathsf{e_{i+1}}) = \frac{k}{2^m}$. We use the following equation to compute the reliability of graph $r(G, s, t, \mathsf{W})$:

$$\begin{aligned} r(G, s, t, \mathsf{W}) &= \mathsf{W}(\text{edge } \mathsf{e_{k+1}} \text{ fails}) \times r(G \setminus e_{k+1}, s, t, \mathsf{W}) \\ &\quad + \mathsf{W}(\text{edge } \mathsf{e_{k+1}} \text{ active}) \times r(G/e_{k+1}, s, t, \mathsf{W}) \\ &= (1 - \frac{k}{2^m}) \times r(G \setminus e_{k+1}, s, t, \mathsf{W}) + \frac{k}{2^m} \times r(G/e_{k+1}, s, t, \mathsf{W}) \\ &= \frac{1}{2^m} \times (k \times r(G/e_{k+1}, s, t, \mathsf{W}) + (2^m - k) \times r(G \setminus e_{k+1}, s, t, \mathsf{W})) \end{aligned}$$

Both of the graphs $G \setminus e_{k+1}$ and G/e_{k+1} have at most i edges. So, we can apply induction hypothesis on both of them.

$$r(G,s,t,\mathsf{W}) = \frac{1}{2^m} \times (k \times \frac{|\mathsf{AS}(\mathcal{Q}_{G/e_{k+1},s,t,\mathsf{W}})|}{2^{\mathsf{num}-m}} + (2^m - k) \times \frac{|\mathsf{AS}(\mathcal{Q}_{G\setminus e_{k+1},s,t,\mathsf{W}})|}{2^{\mathsf{num}-m}})$$
$$= \frac{1}{2^{\mathsf{num}}} \times (k \times |\mathsf{AS}(\mathcal{Q}_{G/e_{k+1},s,t,\mathsf{W}})| + (2^m - k) \times |\mathsf{AS}(\mathcal{Q}_{G\setminus e_{k+1},s,t,\mathsf{W}})|)$$

Now we can apply Lemma 6.2.

$$r(G,s,t,\mathsf{W}) = \frac{1}{2^{\mathsf{num}}} \times (|\mathsf{AS}(\mathcal{Q}_{G,s,t,\mathsf{W}})|)$$

□

6.4 Experimental Evaluation

We implemented a prototype of RelNet-ASP that is configured to use hashing-based answer set counter ApproxASP (see Chapter 5). The RelNet-ASP implementation is available at https://github.com/meelgroup/RelNet-ASP. ApproxASP provides (ε, δ)-guarantees. Another approach for ASP counting is based on translation to CNF. To this end, we have incorporated two standard translations from ASP to CNF: (i) lp2sat and (ii) aspmc [65], and the resulting CNF formulas are fed to exact (#SAT [147]) and approximate (ApproxMC [191]) #SAT solvers. We also incorporated sharpASP (Chapter 3) as a counting tool in RelNet-ASP.

Baseline. We compare the performance of RelNet-ASP with the prior state-of-the-art tools for network reliability estimation. As exact methods for reliability estimation, we considered state space partition (SSP) [176], *Cut*-based (Cut) [180], a decomposition method based on the *Binary Decision Diagram* (BDD) [110], and ProbLog [83]. ProbLog computes the marginal probability of a query given a *probabilistic logic* program. In experiment, we ran ProbLog with *Sentinal Decision Diagram* as the underlying knowledge compilation tool. These methods were chosen as they are widely used in the literature and have been proven to be effective in estimating network reliability. In addition to these exact methods, we also compared RelNet-ASP with several approximate methods. In particular, we compared

DKLR [53] and GBAS [116], and 2-stage DKLR [117] (Huber22). In line with prior studies, we set $\varepsilon = 0.8$ and $\delta = 0.2$ for all techniques that provide (ε, δ)-guarantees. Since the baseline techniques require different inputs, we excluded the graph conversion time from the reported runtimes.

Instances. We conducted experiments on a set of real-world graph networks. We extracted these graph networks from power grids [214] and online social networks [211]. The dataset comprised 710 undirected graph networks, with a maximum of 1000 nodes and 1500 edges. In line with the previous research on network reliability [59], we assumed that each edge had a probability of $\frac{1}{8}$. For each graph network, we randomly chose two nodes as terminal nodes to estimate network reliability. The benchmarks and experimental log files are available at https://zenodo.org/records/19664400.

Objective. The objective of the experimental evaluation was to assess the performance of RelNet-ASP in terms of runtime and accuracy. We compare the performance of RelNet-ASP against several state-of-the-art techniques to compute network reliability: these techniques vary in the accuracy of their estimates. It is crucial to use a metric that captures both accuracy and runtime performance. Therefore, we used the Time Accuracy Penalty (TAP) score [3] as the metric for comparison.

The Time Accuracy Penalty (TAP) score utilizes a *reliable dataset*, which is a set of instances with known ground truth[2]. To compute the ground truth, we applied exact network reliability methods with a memory capacity of 16 GB and a time limit of 10000 seconds to each instance. Finally, we could construct a reliable dataset consisting of 171 instances. Our adapted κ-TAP score follows the definition of TAP score; for a given tool and an instance, the κ-TAP score is defined as follows:

$$\kappa\text{-TAP}(t,i) = \begin{cases} 2 \times \mathcal{T}, & \text{for timeout or memout or error} \\ t + \mathcal{T} \times \frac{\mathcal{R}}{\kappa}, & \text{for } \mathcal{R} < \kappa \\ 2 \times \mathcal{T} - (\mathcal{T} - t) \times \exp(\kappa - \mathcal{R}) & \text{for } \mathcal{R} \geq \kappa \end{cases}$$

where $\kappa = 1 + \varepsilon$, t is the time to estimate reliability, $\mathcal{T} = 3600$ is the timeout (in seconds), $\mathcal{R} = \mathsf{max}(\frac{\hat{r}}{r}, \frac{r}{\hat{r}})$ is the relative error, and r and $\hat{r}$ are ground truth and

[2]The TAP score of [3] relies on *conjectured* value of Z. However, our experimental setup has no *reliable algorithm*. Thus, we rely on instances having known ground truth.

estimated reliability, respectively. We use the term TAP to refer κ-TAP score in the following discussion.

The experimental results demonstrate that RelNet-ASP, with ApproxASP as the underlying ASP counter, outperforms existing reliability estimators in terms of both runtime and accuracy metrics. Current estimators typically prioritize either higher accuracy or runtime efficiency, but RelNet-ASP strikes a balance between the two, rendering it a promising technique for network reliability estimation.

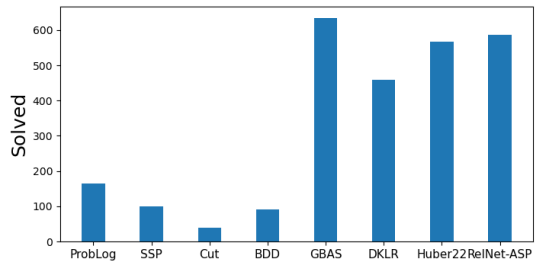
Environmental Settings. All experiments were carried out on a high-performance computer cluster, where each node consists of an 2xE5-2690v3 CPU running with 2x12 real cores and 96GB of RAM. The runtime was limited to 3600 seconds and the memory limit was to limited to 4 GB.

6.4.1 Experimental Results

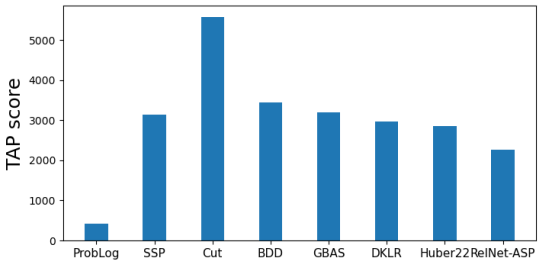
The Table 6.1 compares the performance of RelNet-ASP with baseline techniques. We compare the performance of the techniques in terms of three metrics: the number of solved instances, observed tolerance, and TAP score. The observed tolerance is defined as $\max(\frac{r}{\hat{r}} - 1, \frac{\hat{r}}{r} - 1)$, where r and $\hat{r}$ are the ground truth and estimated reliability, respectively. For better visualization, the performance comparison is shown in bar plots in Figure 6.2. In the table, the 1st, 2nd, and 3rd rows show the number of solved instances, average tolerances, and average TAP scores, respectively.

From Table 6.1, it is clear that while GBAS is able to count for more instances but the runtime performance comes at the cost of accuracy – this observation is consistent in the previous analysis of Monte Carlo methods. On the other hand, ProbLog achieves the lowest TAP score, but its scalability is limited due to the complexity of exact counting. However, among approximate methods, RelNet-ASP achieves the lowest TAP score. Therefore, RelNet-ASP achieves a good balance between the two when considering both runtime performance and accuracy.

Table 6.2 illustrates the performance of RelNet-ASP w.r.t. different underlying ASP counters. The table demonstrates that ApproxASP outperforms other underlying approximate answer set counters in terms of performance. Furthermore, RelNet-ASP, in conjunction with exact answer set counters (e.g., lp2sat+SharpSAT-TD and aspmc+SharpSAT-TD), is able to output the ground truth of network reliability,



(a) Number of solved instances; the higher the better.



(b) TAP score; the lower the better.

Figure 6.2: The performance comparison of RelNet-ASP with baselines.

	Exact methods				Approximate methods			
	ProbLog	SSP	Cut	BDD	GBAS	DKLR	Huber22	RelNet-ASP
Solved (710)	164	100	38	92	635	460	567	587
Tolerance (avg)	0	0	0	0	0.423	0.078	0.132	0.036
TAP (avg)	**428**	3133	5583	3442	3198	2972	2853	2262

Table 6.1: The performance of RelNet-ASP with different reliability estimators.

indicating the validity of RelNet-ASP as an estimator for network reliability.

	RelNet-ASP					
		aspmc		lp2sat		
	SharpASP	#SAT	ApproxMC	#SAT	ApproxMC	ApproxASP
Solved (710)	30	91	214	78	474	587
Tolerance (avg)	0	0	0.065	0	0.076	0.036
TAP (avg)	5372	3630	2568	3982	2452	**2262**

Table 6.2: The performance of RelNet-ASP w.r.t. different underlying ASP counters.

Chapter 7

Counting Problems on Boolean Networks

Our objective is to propose, formulate, and evaluate counting problems over Boolean Networks (BN). More specifically, we formulate six meaningful problems related to counting minimal trap spaces and fixed points in BNs (see Section 2.7 for definitions). These counting problems capture core tasks such as counting all minimal trap spaces or fixed points, counting those that satisfy a given *property* (e.g., a *phenotype*), and counting solutions under *perturbations*. We subsequently propose novel and efficient methods to solve the counting problems by exploiting the expressive power of Answer Set Programming. Following existing ASP counting literature, our ASP-based reduction leverages the approximate answer set counter ApproxASP. Our extensive experimental evaluation on a diverse benchmark dataset demonstrates that ApproxASP efficiently estimates the number of minimal trap spaces and fixed points, and ApproxMC efficiently estimates fixed point counts.

7.1 Related Work

ASP-based Computation of Fixed Points and Minimal Trap Spaces. Several ASP encodings have been proposed to characterize fixed points and minimal trap spaces in BNs. In many cases, an ASP encoding for fixed points is derived from its minimal trap space counterpart by adding *integrity constraints* to capture the specific properties of fixed points [202, 204]. The first ASP encoding that requires the computation of *prime implicants* for each Boolean function was proposed and implemented in [144]. A major bottleneck of this encoding is that computing even a single prime implicant is NP-hard and the total number of prime implicants can be

exponential in the number of function inputs [49]. Subsequent encodings [178, 202, 204] that were proposed to overcome this bottleneck still suffer from scalability and efficiency issues, particularly for very large and complex models, primarily because they require the *disjunctive normal forms* of all the Boolean functions of the original BN. For fixed points, the ASP encoding by Trinh et al. [203] (called `fASP`) uses a *negation normal form* for each Boolean function, which is much more efficient to obtain. This encoding was later generalized for minimal trap spaces [201] (called `tsconj`); however, when dealing with *unsafe formulas* that might yield incorrect solutions, it still requires a disjunctive normal form to ensure the correctness.

BN Encoding of Normal Logic Programs. The theoretical work by Inoue [120] was among the first to establish a connection between ASP and BNs. It defines a BN encoding for finite ground normal logic programs, which relies on the notion of the Clark's completion [51], and points out that the two-valued models of the Clark's completion of a finite ground normal logic program one-to-one correspond to the fixed points of the encoded BN. The subsequent work [121] points out that the strict supported classes of a finite ground normal logic program one-to-one correspond to the synchronous attractors of the encoded BN. Trinh et al. [205] related the regular models in a finite ground normal logic program and the minimal trap spaces in the respective BN, and further applied these theoretical results to explore graphical conditions for the existence, uniqueness, and number of regular models.

SAT Characterization of Fixed Points. The set of fixed points of a BN f can be characterized as the set of *satisfying assignments* of the propositional formula $\bigwedge_{v \in \mathsf{Var}(f)} (v \leftrightarrow f_v)$ [58]. Hence, we can apply #SAT tools [47, 188, 198] to counting the number of fixed points of a BN. To the best of our knowledge, to date, there is no SAT characterization for the set of minimal trap spaces of a BN.

Answer Set Counting. For general ASP programs, #ASP is #·coNP-complete [82], while #ASP is #·P-complete for normal ASP programs, which follows from standard reductions [126]. The projected answer set counting #PASP is simply Σ_2^p-complete if the set of projection atoms is empty; otherwise, the complexity is #·Σ_2^p-complete [75]. Fichte et al. [75, 82] exploited the *tree decomposition*-based technique for counting

answer sets. Kabir et al. [133] introduced the hashing-based approximate counting technique for answer set counting (Chapter 5).

7.2 Problem Formulations

In this section, we introduce several counting problems related to minimal trap spaces and fixed points in Boolean networks (BNs). We begin with a straightforward counting variant, then propose a specialized variant that requires solutions to satisfy a specific *property*, and finally present a more complex variant focused on phenotype measurement in BNs under perturbations. To highlight the biological utility of these theoretical problems, a brief case study is also given in subsection 7.4.2.

7.2.1 Counting Minimal Trap Spaces and Fixed Points

We introduce two fundamental counting problems for Boolean networks (BNs): one that counts the number of minimal trap spaces (Definition 3) and another one counts the number of fixed points (Definition 4). These problems address the basic question: How many minimal trap spaces or fixed points does a given BN have?

Definition 3 (C-MTS-1)**.** *Given a BN f, compute the number of minimal trap spaces of f.*

Definition 4 (C-FIX-1)**.** *Given a BN f, compute the number of fixed points of f.*

In BN research, both counting problems — C-MTS-1 and C-FIX-1 — are valuable when full enumeration is infeasible, as in gene regulatory network models with many source variables [2, 201, 203]. They are also useful when divergent solutions are sought [50]. Notably, the fixed point counting problem can enhance methods for enumerating asynchronous attractors by enabling the selection of a smaller candidate set, thereby potentially speeding up the filtering process [206]. Finally, both C-MTS-1 and C-FIX-1 can lay the groundwork for probabilistic reasoning in BNs [189].

7.2.2 Counting with Satisfying Properties

We examine a specialized variant of problems C-MTS-1 and C-FIX-1, focusing on counting minimal trap spaces (Definition 5) and fixed points (Definition 6) that

satisfy a specified property. This formulation addresses the question: How many minimal trap spaces (or fixed points) in a BN exhibit a given property or assumption?

In systems biology, BNs are used to model biological phenotypes, which reflect an organism's functional characteristics [154]. Several definitions of phenotype in BNs have been proposed [29, 145, 146]; in this work, we adopt one of the most widely used notions. Given a BN f, we define *trait* as a statement of the form $(v \leftrightarrow e)$, where $v \in \mathsf{Var}(f)$ and $e \in \mathbb{B}_\star$. Note that $v \leftrightarrow \star$ is evaluated true if and only if $v = \star$. A phenotype β is then defined as the conjunction of a set of traits. A sub-space m satisfies a phenotype β (denoted by $m \models \beta$) if, upon replacing each variable $v \in \beta$ with its value $m(v)$, the resulting formula evaluates to true under propositional semantics. Unlike minimal trap spaces, fixed points require that all variables take Boolean values ($e \in \mathbb{B}$), and hence phenotypes involving '$\star$' are not meaningful in this context.

In our problem formulation, the property of interest is a desirable phenotype. In systems biology, a minimal trap space satisfying a phenotype suggests the phenotype's potential emergence in vivo.

Definition 5 (C-MTS-2)**.** *Given a BN f and a phenotype β, compute the number of minimal trap spaces of f that satisfy β.*

Definition 6 (C-FIX-2)**.** *Given a BN f and a phenotype β, compute the number of fixed points of f that satisfy β.*

Example 7.1. *Consider again the BN f shown in Example 2.3. It has a unique minimal trap space $m_1 = 00$, which is also its only fixed point of f. Hence, the answer to C-MTS-1 (resp. C-FIX-1) is* 1*. Considering the phenotype $\beta = (b \leftrightarrow \star)$, the answer to C-MTS-2 is* 0*.*

7.2.3 Counting Under Perturbations and Robustness

We consider one of the main contributions of our work to be the identification of new, relevant counting problems (other than C-MTS-2 and C-FIX-2, which were previously known): C-MTS-3 and C-FIX-3. To formalize these problems, we introduce the concept of a *perturbation*.

Consider a BN f. A *perturbation* σ [196] on a set $\mathcal{X} \subseteq \mathsf{Var}(f)$ of *perturbable* variables is defined as a mapping from $\mathcal{X} \to \mathbb{B}_\star$. In practice, $\mathcal{X}$ can be any subset of variables whose perturbation is biologically meaningful. Since each variable in $\mathcal{X}$ can assume one of three values under perturbation, there are $3^{|\mathcal{X}|}$ possible perturbations. For each $v \in \mathcal{X}$, setting $\sigma(v) = 0$ forces $f_v = 0$ (*knockout perturbation*), setting $\sigma(v) = 1$ forces $f_v = 1$ (*over-expression perturbation*), and setting $\sigma(v) = \star$ leaves f_v unchanged. Consequently, the perturbed BN, denoted f^σ, is defined by $\mathsf{Var}(f^\sigma) = \mathsf{Var}(f)$ and, for every $v \in \mathsf{Var}(f^\sigma)$,

$$f_v^\sigma = \begin{cases} \sigma(v) & \text{if } v \in \mathcal{X} \text{ and } \sigma(v) \neq \star \\ f_v & \text{otherwise} \end{cases}$$

Note that, the value $\star$ is used to distinguish imperturbable variables and perturbable variables that are unchanged under a certain perturbation.

In biological systems, a perturbation refers to any disturbance that disrupts the normal functioning of a BN. Such disturbances may arise from *genetic mutations* [189], external factors such as *medications* [31], or other influences [170]. These perturbations can substantially alter the phenotypes exhibited by a BN, making it crucial to quantify their effects on network behavior. In systems biology, this impact is commonly assessed in terms of *robustness* [143], which motivates our definitions of problems C-MTS-3 and C-FIX-3.

Definition 7 (C-MTS-3)**.** *Given a BN f, a set of perturbable variables $\mathcal{X}$, and a target phenotype β, determine the number of perturbations σ on $\mathcal{X}$ such that the perturbed BN f^σ exhibits at least one minimal trap space that satisfies β.*

Definition 8 (C-FIX-3)**.** *Given a BN f, a set of perturbable variables $\mathcal{X}$, and a target phenotype β, determine the number of perturbations σ on $\mathcal{X}$ such that the perturbed BN f^σ exhibits at least one fixed point that satisfies β.*

Leveraging the results of C-MTS-3 (or C-FIX-3 if focusing solely on fixed points), we define the robustness of a phenotype as the fraction of perturbations that preserve the phenotype relative to the total number of perturbations ($3^{|\mathcal{X}|}$). In other words, robustness measures the probability that a phenotype remains active following a random, admissible perturbation is applied to the network. This measure of

phenotype robustness can inform the selection of specific perturbations and guide targeted treatment strategies [29, 200]. A small case study on an interferon model with 121 variables presents these concepts more practically in subsection 7.4.2.

Example 7.2. *Consider BN f given in Example 2.3. Consider $\mathcal{X} = \{b\}$ denote the set of perturbable variables and define the set of desirable phenotype as $\beta = (a \leftrightarrow 0 \wedge b \leftrightarrow 0)$. There are three possible perturbations: $\sigma_1 = \{b = 0\}$, $\sigma_2 = \{b = 1\}$, and $\sigma_3 = \{b = \star\}$. The perturbation σ_1 represents the knockout perturbation of variable b and $\mathsf{sstg}(f^{\sigma_1})$ is given in Figure 2.1b. It is straightforward to observe that f^{σ_1} has two minimal trap spaces 00 and 10. In constrast, the BN f^{σ_2} has one minimal trap space 01, which does not satisfy the given phenotype. The BN f^{σ_3} equals f and has one minimal trap space 00. Therefore, for BN f, phenotype β and perturbable variables $\mathcal{X}$, the answer to C-MTS-3 is 2 and the perturbation robustness of phenotype β in BN f w.r.t. $\mathcal{X}$ is $2/3$.*

On the impact of precision. While these problems are defined *exactly*, in practice their results mainly serve as a means of comparison. For example, the results of C-MTS-2 could be used to compare the abundance of two biological phenotypes. Then the exact count may not be important, as long as the two phenotypes can be compared reliably. As such, these problems are particularly suitable for approximate counting. This also impacts the choice of method parameters (ϵ and δ), as in practice, even low precision (as used in our experiments) can be sufficient to distinguish between significantly different phenotypes. For closely matched results, the precision can be then increased as needed.

7.3 Methodology

Given a BN f, our approach is to construct an ASP program P such that the answer sets of P one-to-one correspond to the minimal trap spaces (or fixed points) of f. This reduction allows us to leverage existing answer set counters to efficiently count the answer sets of P. For C-MTS-1 and C-FIX-1, we simply use the ASP encodings of `tsconj` and `fASP`, respectively. For C-MTS-2 and C-FIX-2, we complement the encoding of the given phenotype. For C-MTS-3 and C-FIX-3,

we propose a new perturbation encoding. Now, we begin by briefly reviewing the `tsconj` and `fASP` encodings.

7.3.1 `tsconj` and `fASP` Encodings

We first present the common components of the two encodings, followed by their differences. The ASP encodings represent sub-spaces using atoms $\mathsf{p}(v)$ and $\mathsf{n}(v)$, indicating whether variable v is fixed to 1, 0, or left free. The goal is to translate BN trap space and fixed point properties into ASP rules whose answer sets correspond to these configurations.

Given a BN f, the encodings compute an ASP program P as follows: for each variable $v \in \mathsf{Var}(f)$, two atoms $\mathsf{p}(v)$ and $\mathsf{n}(v)$ are introduced to indicate positive and negative assignments of the variable v, respectively. Additionally, for every $v \in \mathsf{Var}(f)$, one rule of the form: $\mathsf{p}(v) \vee \mathsf{n}(v) \leftarrow \top$ is added to ensure that each answer set corresponds to a sub-space of f. The translation from an answer set M to a sub-space m is defined as follows: for each $v \in \mathsf{Var}(f)$, we have (i) $m(v) = 1$ if and only if $\mathsf{p}(v) \in M \wedge \mathsf{n}(v) \notin M$, (ii) $m(v) = 0$ if and only if $\mathsf{p}(v) \notin M \wedge \mathsf{n}(v) \in M$, and (iii) $m(v) = \star$ if and only if $\mathsf{p}(v) \in M \wedge \mathsf{n}(v) \in M$. Recall that a trap space of f can be characterized by $\bigwedge_{v \in \mathsf{Var}(f)} (v \leftarrow f_v) \wedge (\neg v \leftarrow \neg f_v)$ [201]. For every $v \in \mathsf{Var}(f)$, two ASP rules — (i) $\gamma(v) \leftarrow \gamma(\mathsf{NNF}(f_v))$ and (ii) $\gamma(\neg v) \leftarrow \gamma(\mathsf{NNF}(\neg f_v))$ — are added to P to express the characterization, where $\mathsf{NNF}(\Phi)$ denotes a negation normal form of a Boolean formula Φ and γ is a procedure defined as follows:

$$\gamma(v) = \mathsf{p}(v), \gamma(\neg v) = \mathsf{n}(v), v \in \mathsf{Var}(f),$$
$$\gamma(\bigwedge_{1 \le j \le J} \alpha_j) = \gamma(\alpha_1) \wedge \ldots \wedge \gamma(\alpha_J), \gamma(\bigvee_{1 \le j \le J} \alpha_j) = \mathsf{aux}_k,$$

where aux_k is a new *auxiliary* atom, k is a global counter starting from 1 and shall be increased by 1 after each new auxiliary atom is created, and for each j, the rule $\mathsf{aux}_k \leftarrow \gamma(\alpha_j)$ is added to P.

For the correctness of the `tsconj` encoding, Trinh et al. [201] defined a syntactic safeness condition for a Boolean formula Φ: Φ is considered safe if it does not contain any conjunction of two subformulas Φ_1 and Φ_2 such that there exists a variable x appearing in Φ_1 with $\neg x$ appearing in Φ_2. When both f_v and $\neg f_v$ are safe for every $v \in \mathsf{Var}(f)$, then the set of answer sets of P one-to-one corresponds with the set

of minimal trap spaces of f. When a Boolean formula Φ (f_v or $\neg f_v$) is unsafe, the disjunctive normal form of Φ is used instead, which is always safe by definition.

There is no notion of "safeness" in the `fASP` encoding. Rather an additional rule $\bot \leftarrow \mathsf{p}(v), \mathsf{n}(v)$ is added to P, for every $v \in \mathsf{Var}(f)$ such that the variable $v \neq \star$. The answer sets of P one-to-one correspond to the fixed points of f.

Following on, given a BN f, we use the notations $\mathsf{P\text{-}tsconj}(f)$ and $\mathsf{P\text{-}fASP}(f)$ to denote the encoded ASP programs of f, according to the `tsconj` and `fASP` encodings, respectively. ASP offers a direct declarative way to characterize minimal trap spaces and fixed points through stable model semantics, giving it an advantage over other approaches for computing them.

7.3.2 Methods for Problems C-MTS-1 and C-FIX-1

We make use of `tsconj` and `fASP` encodings for C-MTS-1 and C-FIX-1, respectively. The choice of encodings is due to following two reasons: first, these encodings rely on less expensive representations — specifically, negation normal forms (ref. Section 7.1). Second, they establish a one-to-one correspondence between the minimal trap spaces (or fixed points) of the original BN and the answer sets of the encoded ASP program, whereas other encodings yield a one-to-one correspondence with the subset-minimal (or subset-maximal) answer sets [10], which prevents the direct use of existing ASP counters.

7.3.3 Methods for Problems C-MTS-2 and C-FIX-2

We add the encoding of phenotype to the encodings of `tsconj` and `fASP` to solve counting problems C-MTS-2 and C-FIX-2, respectively. Given a BN f and a phenotype β, we compute an ASP program $\mathsf{ToASP}(\beta)$ to capture the phenotype β by invoking Algorithm 6. The algorithm exploits atoms introduced in the `tsconj` encoding to interpret the phenotype β. The main idea of Algorithm 6 is exploiting *faceted answer set navigation* (see Subsection 2.2.5) to constrain the search space of answer sets [80]. We prove that the minimal trap spaces of f satisfying the phenotype β one-to-one correspond to the answer sets of $\mathsf{P\text{-}tsconj}(f) \cup \mathsf{ToASP}(\beta)$ (Theorem 6).

Algorithm 6 $\mathsf{ToASP}(\beta)$

Input: Phenotype β
Output: ASP program Q

1: $Q \leftarrow \emptyset$
2: **for each** $(v \leftrightarrow e) \in \beta$ **do**
3: **if** $e = 1$ **then**
4: $Q.\mathsf{add}(\bot \leftarrow {\sim}\mathsf{p}(v), \quad \bot \leftarrow \mathsf{n}(v))$
5: **else if** $e = 0$ **then**
6: $Q.\mathsf{add}(\bot \leftarrow \mathsf{p}(v), \quad \bot \leftarrow {\sim}\mathsf{n}(v))$
7: **else if** $e = \star$ **then**
8: $Q.\mathsf{add}(\bot \leftarrow {\sim}\mathsf{p}(v), \quad \bot \leftarrow {\sim}\mathsf{n}(v))$
9: **return** Q

Theorem 6. *Given a BN f and a phenotype β, the minimal trap spaces of f satisfying β one-to-one correspond to the answer sets of* $\mathsf{P\text{-}tsconj}(f) \cup \mathsf{ToASP}(\beta)$.

Proof. To prove the correctness of Theorem 6, we reuse the correctness proof of Theorem 2 of [201], which establishes that the answer sets of $\mathsf{P\text{-}tsconj}(f)$ one-to-one correspond to the minimal trap spaces of f. Let us recall the translation between an answer set M of $\mathsf{P\text{-}tsconj}(f)$ and its respective minimal trap space m of f: for each variable $v \in \mathsf{Var}(f)$, $m(v) = 1$ if and only if $\mathsf{p}(v) \in M \wedge \mathsf{n}(v) \notin M$, $m(v) = 0$ if and only if $\mathsf{p}(v) \notin M \wedge \mathsf{n}(v) \in M$, and $m(v) = \star$ if and only if $\mathsf{p}(v) \in M \wedge \mathsf{n}(v) \in M$.

We employ the theories of faceted answer set navigation [80] to prove the correctness of Theorem 6. According to faceted navigation, for a program P and an atom $a \in \mathsf{atoms}(P)$, adding the integrity constraint $\{\bot \leftarrow a\}$ to program P restricts the search space of P, where no answer set contains the atom a. Conversely, adding the integrity constraint $\{\bot \leftarrow {\sim}a\}$ to program P ensures that every answer set in the modified program contains the atom a.

We prove the correctness of our encoding for each trait $(v \leftrightarrow e) \in \beta$. We show that Algorithm 6 selects *facets* in such a way that the answer sets of $\mathsf{P\text{-}tsconj}(f) \cup \mathsf{ToASP}(\beta)$ one-to-one correspond to the minimal trap spaces satisfying β of f. When $(v \leftrightarrow 1) \in \beta$ (line 4 in Algorithm 6), two constraints are added to $\mathsf{ToASP}(\beta)$ to ensure that every answer set contains the atom $\mathsf{p}(v)$ and none contains the atom $\mathsf{n}(v)$. These constraints effectively assign the value 1 to variable v. When $(v \leftrightarrow 0) \in \beta$ (line 6 in Algorithm 6), two added constraints ensure that every answer set contains the atom $\mathsf{n}(v)$ and none contains $\mathsf{p}(v)$, thereby assigning the value 0 to variable v. When

$(v \leftrightarrow \star) \in \beta$ (line 8 in Algorithm 6), two added constraints ensure that every answer set contains both $\mathsf{p}(v)$ and $\mathsf{n}(v)$. These constraints effectively assign the value $\star$ to variable v.
Combining all the cases of Algorithm 6, we can claim the correctness of our encoding. □

For C-FIX-2, we can apply the `fASP` encoding and Algorithm 6 similarly. Note that in the counting problem C-FIX-2, the term e is either 0 or 1, for each $(v \leftrightarrow e) \in \beta$, since fixed points require all variables to be fixed. We formally prove the correctness of our proposed method for C-FIX-2 in Theorem 7.

Theorem 7. *Given a BN f and a phenotype β, the fixed points of f satisfying β one-to-one correspond to the answer sets of* $\mathsf{P\text{-}fASP}(f) \cup \mathsf{ToASP}(\beta)$.

Proof. The proof technique of Theorem 6 can be similarly extended for fixed point counting with the program $\mathsf{P\text{-}fASP}(f)$. □

Example 7.3. *Consider the BN f of Example 2.3. The program* $\mathsf{P\text{-}tsconj}(f)$ *is as follows:*

$$\mathsf{p}(a) \vee \mathsf{n}(a) \leftarrow \top \qquad \mathsf{p}(a) \leftarrow \mathsf{p}(a), \mathsf{n}(b) \qquad \mathsf{n}(a) \leftarrow \mathsf{aux}_1 \qquad \mathsf{aux}_1 \leftarrow \mathsf{n}(a) \quad \mathsf{aux}_1 \leftarrow \mathsf{p}(b)$$
$$\mathsf{p}(b) \vee \mathsf{n}(b) \leftarrow \top \qquad \mathsf{p}(b) \leftarrow \mathsf{p}(a) \qquad \mathsf{n}(b) \leftarrow \mathsf{n}(a)$$

The program $\mathsf{P\text{-}tsconj}(f)$ *has a unique answer set* $\{\mathsf{n}(a), \mathsf{n}(b), \mathsf{aux}_k\}$ *corresponding to the unique minimal trap space* 00 *of f. Consider the phenotype $\beta = (b \leftrightarrow \star)$, the BN f has no minimal trap space satisfying β (see Example 7.1), thus C-MTS-2 returns* 0. *Following the procedure outlined above, the ASP program* $\mathsf{ToASP}(\beta)$ *is as follows:*

$$\bot \leftarrow {\sim}\mathsf{p}(b) \qquad \bot \leftarrow {\sim}\mathsf{n}(b)$$

Indeed, the program $\mathsf{P\text{-}tsconj}(f) \cup \mathsf{ToASP}(\beta)$ *has no answer set.*

7.3.4 Methods for Problems C-MTS-3 and C-FIX-3

Given a BN f, a phenotype β, and a set of perturbable variables $\mathcal{X}$, one possible approach to solving these problems is to introduce new atoms to represent possible perturbations over perturbable variables, then correspondingly to intervene in the

encoded ASP program obtained by applying the encodings proposed for C-MTS-2 and C-FIX-2, and finally to apply projected counting restricted to these new atoms. Instead, we propose a more convenient approach that reduces C-MTS-3 (resp. C-FIX-3) to C-MTS-2 (resp. C-FIX-2) along with projected answer set counting.

Definition 9. *Consider a BN f and a set of perturbable variables $\mathcal{X} \subseteq \mathsf{Var}(f)$, we construct a new BN g such that for every $v \in \mathsf{Var}(f)$, if $v \in \mathsf{Var}(f) \setminus \mathcal{X}$, then the variable $v \in \mathsf{Var}(g)$ and $g_v = f_v$, and if $v \in \mathcal{X}$, then three variables $v, v^k, v^o \in \mathsf{Var}(g)$ and*

$$\begin{aligned} g_v &= \neg v^k \wedge (v^o \vee f_v), \\ g_{v^k} &= v^k, \\ g_{v^o} &= v^o \wedge \neg v^k. \end{aligned}$$

Instead of modifying the ASP encoding to model perturbations, we construct a perturbed version of the original BN, thereby preserving the semantics of the original encodings. For each perturbable variable $v \in \mathcal{X}$, we introduce two new variables in g: v^k, encoding whether v is knocked out, and v^o, encoding whether it is over-expressed. By construction:

- If $v^k = 1$, then $v^o = 0$ and $v = 0$, modeling a knockout of v.
- If $v^k = 0$ and $v^o = 1$, then $v = 1$, modeling over-expression.
- If $v^k = 0$ and $v^o = 0$, then v follows its update function f_v (v is unperturbed).
- The case $v^k = 1$ and $v^o = 1$ is infeasible due to the constraint $g_{v^o} = v^o \wedge \neg v^k$.

Since $\mathsf{Var}(f) \subseteq \mathsf{Var}(g)$, any phenotype β defined over f remains valid in g. The minimal trap spaces (resp. fixed points) of g correspond to those of f under all possible perturbations over $\mathcal{X}$. Thus, the minimal trap spaces (resp. fixed points) of g that satisfy the phenotype β represent the perturbed solutions of f that also satisfy β. Crucially, for each perturbation, multiple satisfying minimal trap spaces (or fixed points) are counted once, enabling us to count the number of satisfying perturbations. Hence, the counting problem C-MTS-3 (resp. C-FIX-3) reduces to the projected version of C-MTS-2 (resp. C-FIX-2) over the newly defined BN g.

We now discuss how we solve C-MTS-3 and C-FIX-3 by applying projected answer set counting. We focus on the case for minimal trap spaces, and the case of fixed points is trivially similar. Following Theorem 6, the set of answer sets of $\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta)$ represents the set of minimal trap spaces of g satisfying the phenotype β. Let $\Omega = \bigcup_{v \in \Delta} \{\mathsf{p}(v), \mathsf{n}(v)\}$ denote the set of perturbation-related variables, where $\Delta = \bigcup_{v \in \mathcal{X}} \{v^k, v^o\}$ be the set of ASP atoms used in the encoding. It follows that the number of answer sets of $\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta)$, projected onto the set Ω is equal to the number of perturbation settings (i.e., assignments to variables in $\mathcal{X}$) under which g admits a minimal trap space satisfying β.

Theorem 8. *Given a BN f, a set of perturbable variables $\mathcal{X} \subseteq \mathsf{Var}(f)$, a phenotype β, and $\Omega = \bigcup_{v \in \Delta} \{\mathsf{p}(v), \mathsf{n}(v)\}$, where $\Delta = \bigcup_{v \in \mathcal{X}} \{v^k, v^o\}$, then C-MTS-3 can be computed as $\#\mathsf{PASP}(\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta), \Omega)$, where g is the new BN according to Definition 9.*

Theorem 9. *Given a BN f, a set of perturbable variables $\mathcal{X} \subseteq \mathsf{Var}(f)$, a phenotype β, and $\Omega = \bigcup_{v \in \Delta} \{\mathsf{p}(v), \mathsf{n}(v)\}$, where $\Delta = \bigcup_{v \in \mathcal{X}} \{v^k, v^o\}$, then C-FIX-3 can be computed as $\#\mathsf{PASP}(\mathsf{P\text{-}fASP}(g) \cup \mathsf{ToASP}(\beta), \Omega)$, where g is the new BN following Definition 9.*

To prove Theorems 8 and 9, we prepare the following preliminaries.

Definition 10. *The total order $\leq_t$ on $\mathbb{B}_\star$ is defined by $0 <_t \star <_t 1$.*

Definition 11. *The partial order $\leq_s$ on $\mathbb{B}_\star$ is defined by $0 <_s \star$, $1 <_s \star$, and it contains no other relation.*

Definition 12. *Consider a BN f, a sub-space m of f, and a Boolean expression e over $\mathsf{Var}(f)$. The value of e under sub-space m w.r.t. the Kleene three-valued logic, denoted as $m(e)$, is recursively defined as follows:*

$$m(e) = \begin{cases} e & \text{if } e \in \mathbb{B}_\star \\ m(a) & \text{if } e = a, a \in \mathsf{Var}(f) \\ \neg m(e_1) & \text{if } e = \neg e_1 \\ min_{\leq_t}(m(e_1), m(e_2)) & \text{if } e = e_1 \wedge e_2 \\ max_{\leq_t}(m(e_1), m(e_2)) & \text{if } e = e_1 \vee e_2 \end{cases}$$

where $\neg 1 = 0, \neg 0 = 1, \neg\star = \star$*, and* $min_{\leq_t}$ *(resp.* $max_{\leq_t}$*) is the function to get the minimum (resp. maximum) value of two values w.r.t. the order* $\leq_t$*.*

Theorem 10 (Theorem 1 of [144])**.** *Consider a BN* f *and a sub-space* m *of* f*. A sub-space* m *is a trap space of* f *iff* $m(f_v) \leq_s m(v)$ *for every* $v \in \mathsf{Var}(f)$*.*

Corollary 7.1. *Consider a BN* f *and a sub-space* m *of* f*. A sub-space* m *is a minimal trap space of* f *iff* m *is a* $\leq_s$*-minimal trap space of* f*.*

Corollary 7.2. *Consider a BN* f *and a sub-space* m *of* f*. A sub-space* m *is a fixed point of* f *iff* m *is a trap space of* f *and* $m(v) \neq \star$ *for every* $v \in \mathsf{Var}(f)$*.*

Now, we show the formal proof of Theorem 8.

Lemma 7.1. *Consider a BN* f *and a set of perturbable variables* $\mathcal{X} \subseteq \mathsf{Var}(f)$*. Let* g *be the BN obtained from* f*, according to Definition 9. Let* Δ *be the set* $\bigcup_{v\in\mathcal{X}}\{v^k, v^o\}$*. If* m *is a minimal trap space of* g*, then* $m(v) \neq \star$ *for every* $v \in \Delta$*.*

Proof. Let m be a minimal trap space of g. Assume that there exists $v^k \in \Delta$ such that $m(v^k) = \star$ or $v^o \in \Delta$ such that $m(v^o) = \star$. We consider two cases as follows.

Case 1: $m(v^k) = \star$. Let m' be a sub-space of g such that $m'(u) = m(u)$, for every $u \in \mathsf{Var}(g) \setminus \{v^k\}$ and $m'(v^k) = 0$. We have $m'(g_{v^k}) = m'(v^k)$. The variable v^k only affects v^o and v. Regarding v^o, we have $m'(g_{v^o}) = m'(v^o \wedge \neg v^k) = \min_{\leq_t}(m'(v^o), \neg m'(v^k)) = \min_{\leq_t}(m'(v^o), 1) = m'(v^o)$. Regarding v, we have $m'(g_v) = m'(\neg v^k \wedge (v^o \vee f_v)) = \min_{\leq_t}(\neg m'(v^k), \max_{\leq_t}(m'(v^o), m'(f_v))) = \min_{\leq_t}(1, \max_{\leq_t}(m'(v^o), m'(f_v))) = \max_{\leq_t}(m'(v^o), m'(f_v)) = \max_{\leq_t}(m(v^o), m(f_v))$. Following Theorem 10, $m(g_v) \leq_s m(v)$. We have $m(g_v) = m(\neg v^k \wedge (v^o \vee f_v)) = \min_{\leq_t}(\neg m(v^k), \max_{\leq_t}(m(v^o), m(f_v))) = \min_{\leq_t}(\star, \max_{\leq_t}(m(v^o), m(f_v)))$. Since $m(v) <_s \max_{\leq_t}(m(v^o), m(f_v))$ implies $m(v) <_s m(g_v)$ which is a contradiction, we derive that $\max_{\leq_t}(m(v^o), m(f_v)) \leq_s m(v)$. This implies $m'(g_v) \leq_s m(v) = m'(v)$. For any $u \in \mathsf{Var}(g) \setminus \{v, v^k, v^o\}$, $m'(g_u) = m(g_u) \leq_s m(u) = m'(u)$. Hence, m' is trap space of g and $m' <_s m$, which is a contradiction.

Case 2: $m(v^o) = \star$. Let m' be a sub-space of g such that $m'(u) = m(u)$ for every $u \in \mathsf{Var}(g) \setminus \{v^o\}$ and $m'(v^o) = 0$. We have $m'(g_{v^o}) = m'(v^o \wedge \neg v^k) = \min_{\leq_t}(m'(v^o), \neg m'(v^k)) = \min_{\leq_t}(0, \neg m'(v^k)) = 0 = m'(v^o)$. The variable v^o only affects v. We have $m'(g_v) = m'(\neg v^k \wedge (v^o \vee f_v)) = \min_{\leq_t}(\neg m'(v^k), \max_{\leq_t}(m'(v^o), m'(f_v))) =$

$\min_{\leq_t}(\neg m'(v^k), \max_{\leq_t}(0, m'(f_v))) = \min_{\leq_t}(\neg m'(v^k), m'(f_v)) = \min_{\leq_t}(\neg m(v^k), m(f_v))$. Following Theorem 10, $m(g_v) \leq_s m(v)$. We have $m(g_v) = m(\neg v^k \wedge (v^o \vee f_v)) = \min_{\leq_t}(\neg m(v^k), \max_{\leq_t}(m(v^o), m(f_v))) = \min_{\leq_t}(\neg m(v^k), \max_{\leq_t}(\star, m(f_v)))$. Suppose that $m'(v) = m(v) <_s \min_{\leq_t}(\neg m(v^k), m(f_v))$. If $m(f_v) = 0$, then $m(v) <_s 0$ which contradicts to the definition of $\leq_s$. Hence $m(f_v) \neq 0$, leading to $\max_{\leq_t}(\star, m(f_v)) = m(f_v)$. Then $m(v) <_s \min_{\leq_t}(\neg m(v^k), m(f_v)) = m(g_v)$, which is a contradiction. Hence, $\min_{\leq_t}(\neg m(v^k), m(f_v)) \leq_s m'(v)$. This implies that $m'(g_v) \leq_s m'(v)$. For any $u \in \mathsf{Var}(g) \setminus \{v, v^o\}$, $m'(g_u) = m(g_u) \leq_s m(u) = m'(u)$. Hence, m' is trap space of g and $m' <_s m$, which is a contradiction.

Combining Case 1 and Case 2, we can conclude that for $m(v) \neq \star$ for every $v \in \Delta$. □

Lemma 7.2. *Consider a BN f and a set of perturbable variables $\mathcal{X} \subseteq \mathsf{Var}(f)$. Let g be the BN obtained from f, according to Definition 9 and Δ be the set $\bigcup_{v \in \mathcal{X}}\{v^k, v^o\}$. Let m be a trap space of g such that $m(v) \neq \star$ for every $v \in \Delta$. Let σ be the perturbation of f such that $\sigma(v) = 1$ if and only if $m(v^k) = 0$ and $m(v^o) = 1$, $\sigma(v) = 0$ if and only if $m(v^k) = 1$ and $m(v^o) = 0$, and $\sigma(v) = \star$ if and only if $m(v^k) = 0$ and $m(v^o) = 0$. Then m' is a trap space of f^σ where $m'(v) = m(v)$ for every $v \in \mathsf{Var}(f)$.*

Proof. Assume that there exists $v \in \mathcal{X}$ such that $m(v^k) = m(v^o) = 1$. Since m is a trap space of g, $m(g_{v^o}) \leq_s m(v^o)$. We have $m(g_{v^o}) = m(v^o \wedge \neg v^k) = 0$, whereas $m(v^o) = 1$, leading to $0 \leq_s 1$, which contradicts to the definition of $\leq_s$. Hence, the case of $m(v^k) = m(v^o) = 1$ is impossible for every $v \in \mathcal{X}$. Since $m(v) \neq \star$ for every $v \in \Delta$, σ is well specified.

Recall that $\mathsf{Var}(g) = \mathsf{Var}(f) \cup \Delta$ and $\mathsf{Var}(f) = \mathsf{Var}(f^\sigma)$. Consider $v \in \mathsf{Var}(f^\sigma)$. If $v \notin \mathcal{X}$, we have $m'(f^\sigma_v) = m'(f_v) = m'(g_v) = m(g_v) \leq_s m(v) = m'(v)$. If $v \in \mathcal{X}$, we have the following cases:

Case 1: $m(v^k) = 0$ and $m(v^o) = 0$. Then $\sigma(v) = \star$, thus $m'(f^\sigma_v) = m'(f_v)$. We have $m(g_v) = m(\neg v^k \wedge (v^o \vee f_v)) = m(f_v) \leq_s m(v) = m'(v)$. Since $m'(f_v) = m(f_v)$, it follows that $m'(f^\sigma_v) \leq_s m'(v)$.

Case 2: $m(v^k) = 1$ and $m(v^o) = 0$. Then $\sigma(v) = 0$, thus $m'(f^\sigma_v) = 0$. We have $m(g_v) = m(\neg v^k \wedge (v^o \vee f_v)) = 0 \leq_s m(v) = m'(v)$. Hence, $m'(f^\sigma_v) \leq_s m'(v)$.

Case 3: $m(v^k) = 0$ and $m(v^o) = 1$. Then $\sigma(v) = 1$, thus $m'(f_v^\sigma) = 1$. We have $m(g_v) = m(\neg v^k \wedge (v^o \vee f_v)) = 1 \leq_s m(v) = m'(v)$. Hence, $m'(f_v^\sigma) \leq_s m'(v)$.

Now we can conclude that $m'(f_v^\sigma) \leq_s m'(v)$ for every $v \in \mathsf{Var}(f^\sigma)$. Hence, m' is a trap space of f^σ. □

Proof of Theorem 8

Proof. First, we consider a perturbation $\sigma: \mathcal{X} \to \mathbb{B}_\star$ of BN f. Let $m_\Delta: \Delta \to \mathbb{B}$ be a mapping such that for every $v \in \mathcal{X}$, $\sigma(v) = 1$ if and only if $m_\Delta(v^k) = 0$ and $m_\Delta(v^o) = 1$, $\sigma(v) = 0$ if and only if $m_\Delta(v^k) = 1$ and $m_\Delta(v^o) = 0$, and $\sigma(v) = \star$ if and only if $m_\Delta(v^k) = 0$ and $m_\Delta(v^o) = 0$. Recall that $\mathsf{Var}(g) = \mathsf{Var}(f) \cup \Delta$ and $\mathsf{Var}(f) = \mathsf{Var}(f^\sigma)$. Let m be a minimal trap space of f^σ and $m \models \beta$. Let m' be a sub-space of g such that $m'(v) = m(v)$ if $v \in \mathsf{Var}(f)$ and $m'(v) = m_\Delta(v)$ if $v \in \Delta$. We show that m' is a minimal trap space of g and $m' \models \beta$ (1).

Consider $v \in \mathcal{X}$, we have $m'(g_{v^k}) = m'(v^k)$. If $m'(v^k) = 0$, then $m'(v^o \wedge \neg v^k) = m'(v^o)$. If $m'(v^k) = 1$, then $m'(v^o) = 0$ due to the definition of m_Δ, leading to $m'(v^o \wedge \neg v^k) = 0 = m'(v^o)$. Hence, we can derive that $m'(g_{v^o}) = m'(v^o \wedge \neg v^k) = m'(v^o)$. Regarding $m'(g_v) = m'(\neg v^k \wedge (v^o \vee f_v))$, we have the following cases:

Case 1: $\sigma(v) = \star$. Then $m'(v^k) = 0$ and $m'(v^o) = 0$. We have $m'(g_v) = m'(f_v) = m(f_v) = m(f_v^\sigma) \leq_s m(v) = m'(v)$.

Case 2: $\sigma(v) = 1$. Then $m'(v^k) = 0$ and $m'(v^o) = 1$. We have $m'(g_v) = 1 = m(f_v^\sigma) \leq_s m(v) = m'(v)$.

Case 3: $\sigma(v) = 0$. Then $m'(v^k) = 1$ and $m'(v^o) = 0$. We have $m'(g_v) = 0 = m(f_v^\sigma) \leq_s m(v) = m'(v)$. Consider $v \in \mathsf{Var}(f) \setminus \mathcal{X}$. We have $m'(g_v) = m'(f_v) = m(f_v) = m(f_v^\sigma) \leq_s m(v) = m'(v)$.

Now, we can conclude that m' is a trap space of g. Assume that m' is not minimal. Then there is a trap space n of g such that $n <_s m'$. Since $m'(v) \neq \star$ for every $v \in \Delta$, $n(v) = m'(v)$ for every $v \in \Delta$, leading to $n(v) \neq \star$ for every $v \in \Delta$. Following the Lemma 7.2, n' is a trap space of f^σ where $n'(v) = n(v)$ for every $v \in \mathsf{Var}(f)$. Since $n(v) = m'(v)$ for every $v \in \Delta$, we have $n' <_s m$, which contradicts to the $\leq_s$-minimality of m w.r.t. f^σ. Hence, m' is a minimal trap space of g. In addition, since β only contains the variables in $\mathsf{Var}(f)$, it is trivial that $m' \models \beta$.

Second, we consider a minimal trap space m of g such that $m \models \beta$. By Lemma 7.1, $m(v) \neq \star$ for every $v \in \Delta$. The case of $m(v^k) = m(v^o) = 1$ is impossible because if it holds, then $m(g_{v^o}) = m(v^o \wedge \neg v^k) = 0 \leq_s m(v^o) = 1$, which is a contradiction. Let σ be the perturbation of f such that $\sigma(v) = 1$ if and only if $m(v^k) = 0$ and $m(v^o) = 1$, $\sigma(v) = 0$ if and only if $m(v^k) = 1$ and $m(v^o) = 0$, and $\sigma(v) = \star$ if and only if $m(v^k) = 0$ and $m(v^o) = 0$. Let m' be a sub-space of f such that $m'(v) = m(v)$ for every $v \in \mathsf{Var}(f)$. We show that m' is a minimal trap space of f^σ and $m' \models \beta$ (2).

By Lemma 7.2, m' is a trap space of f^σ. Assume that m' is not minimal. Then there is a minimal trap space n of f^σ such that $n <_s m'$. Let n' be a sub-space of g such that $n'(v) = m(v)$ for every $v \in \Delta$ and $n'(v) = n(v)$ for every $v \in \mathsf{Var}(f)$. By following the same reasoning for (1), we have n' is a trap space of g. However, $n' <_s m$, which contradicts to the $\leq_s$-minimality of m w.r.t. g. Hence, m' is a minimal trap space of f^σ. In addition, since β only contains the variables in $\mathsf{Var}(f)$, it is trivial that $m' \models \beta$.

From (1) and (2), we can conclude that, given $f, \mathcal{X}$, and β, the result of the counting problem C-MTS-3 is equivalent to the number of minimal trap spaces of g that satisfy β where multiple minimal trap spaces with the same values on the variables in Δ are only counted once. By Theorem 6, the answer sets of $\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta)$ one-to-one correspond to the minimal trap spaces of g satisfying β. The set Ω includes the atoms of $\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta)$ corresponding to the variables in Δ of g. It follows that the number of answer sets of $\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta)$ projected to Ω is equal to the number of minimal trap spaces of g satisfying β projected to Δ. This implies that C-MTS-3 can be computed as the projected answer set counting query $\#\mathsf{PASP}(\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta), \Omega)$. □

Proof of Theorem 9

Proof. The proof technique of Theorem 8 can be similarly extended for C-FIX-3 with the program $\mathsf{P\text{-}fASP}(g)$. □

Example 7.4. *Consider again Example 7.2. Following Definition 9, we obtain the BN g:* $\mathsf{Var}(g) = \{a, b, b^k, b^o\}$, $g_a = a \wedge \neg b$, $g_{b^k} = b^k$, $g_{b^o} = b^o \wedge \neg b^k$, *and*

$g_b = \neg b^k \wedge (b^o \vee a)$.

The ASP program $\mathsf{P\text{-}tsconj}(f)$ *is as follows:*

$\mathsf{p}(a) \vee \mathsf{n}(a) \leftarrow \top \quad \mathsf{p}(a) \leftarrow \mathsf{p}(a), \mathsf{n}(b) \quad \mathsf{n}(a) \leftarrow \mathsf{aux}_1 \quad \mathsf{aux}_1 \leftarrow \mathsf{n}(a) \quad \mathsf{aux}_1 \leftarrow \mathsf{p}(b)$

$\mathsf{p}(b) \vee \mathsf{n}(b) \leftarrow \top$

$\mathsf{p}(b) \leftarrow \mathsf{n}(b^k), \mathsf{aux}_2 \quad \mathsf{aux}_2 \leftarrow \mathsf{p}(b^o) \quad \mathsf{aux}_2 \leftarrow \mathsf{p}(a)$

$\mathsf{n}(b) \leftarrow \mathsf{aux}_3 \quad \mathsf{aux}_3 \leftarrow \mathsf{p}(b^k) \quad \mathsf{aux}_3 \leftarrow \mathsf{n}(b^o), \mathsf{n}(a)$

$\mathsf{p}(b^k) \vee \mathsf{n}(b^k) \leftarrow \top \quad \mathsf{p}(b^k) \leftarrow \mathsf{p}(b^k) \quad \mathsf{n}(b^k) \leftarrow \mathsf{n}(b^k)$

$\mathsf{p}(b^o) \vee \mathsf{n}(b^o) \leftarrow \top \quad \mathsf{p}(b^o) \leftarrow \mathsf{p}(b^o), \mathsf{n}(b^k) \quad \mathsf{n}(b^o) \leftarrow \mathsf{aux}_4 \quad \mathsf{aux}_4 \leftarrow \mathsf{n}(b^o) \quad \mathsf{aux}_4 \leftarrow \mathsf{p}(b^k)$

The ASP program $\mathsf{ToASP}(\beta)$ *is as follows:*

$$\bot \leftarrow \mathsf{p}(a), \qquad \bot \leftarrow {\sim}\mathsf{n}(a)$$
$$\bot \leftarrow \mathsf{p}(b), \qquad \bot \leftarrow {\sim}\mathsf{n}(b)$$

Let $\Omega = \{\mathsf{p}(b^k), \mathsf{n}(b^k), \mathsf{p}(b^o), \mathsf{n}(b^o)\}$. *Then C-MTS-3 can be computed as* $\#\mathsf{PASP}(\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta), \Omega)$. *Indeed,* $\#\mathsf{PASP}(\mathsf{P\text{-}tsconj}(g) \cup \mathsf{ToASP}(\beta), \Omega)$ *returns* 2, *which is consistent with the result shown in Example 7.2.*

7.4 Experimental Evaluation

This section presents the experimental evaluation of the presented methods. We use existing minimal trap space and fixed point computation tools as baselines—namely, AEON [28], k++ADF (ADF) [163], and clingo [90]. The ADF tool is applicable here due to the equivalence between ADFs and BNs [18, 114], which supports only C-MTS-1 and C-FIX-1. Both k++ADF and Clingo count minimal trap spaces and fixed points via enumeration on the ADF and the encoded ASP respectively, and AEON via BDD-based encoding. For the fixed point problem variants, we further considered the propositional model counter [188] (GANAK) and the approximate model counter [215] (ApproxMC), using the translation to CNF (see Section 7.1). However, note that these techniques cannot be directly used for minimal trap space counting. For approximate answer set counting, we employed the

hashing-based approximate answer set counter ApproxASP [133] with parameters $\varepsilon = 0.8$ and $\delta = 0.2$. Following prior work on counting [135], we provided ApproxASP with an independent support (see Subsection 2.2.6) of a disjunctive ASP program exploiting Padoa theorem [175]. We computed the independent support of each program with a time limit of 1000 seconds. Since the baselines require different input formats, we excluded the BN conversion time from the reported runtimes. We also tested the tree decomposition-based answer set counter DynASP [82]; however, we did not include it in the final analysis because it has been significantly outperformed by the remaining baselines. We also compared with sharpASP-$\mathcal{SR}$ (Chapter 4) for C-MTS-1; however, its performance is only slightly better than that of clingo. Finally, note that we could not include #SAT-based ASP counters aspmc [66] and sharpASP [132] as baselines, since these counters are designed for normal logic programs, while `tsconj` and `fASP` encodings produce disjunctive ASP programs. The tool implementation is available at `https://github.com/meelgroup/bn-counting`.

Benchmark Information. We compiled our benchmark set from prior studies on minimal trap spaces and fixed points in BNs [177, 201, 203]. The set comprises 645 BN instances—245 real-world models and 400 randomly generated—with up to 5,000 variables. The real-world Boolean models range up to 1076 variables, out of which up to 223 are source variables. However, the median network size in this dataset is less than 100 variables. As such, we also consider larger random Boolean networks ranging between 1,000 and 5,000 variables. These 645 instances are used directly as inputs for benchmarking the C-FIX-1 and C-MTS-1 problems. All source variables across all networks are left *unrestricted*, meaning they can take the value 0 or 1, thereby maximizing the number of admissible trap spaces.

To evaluate counting problems C-MTS-2, C-FIX-2, C-MTS-3, and C-FIX-3, we pseudo-randomly fixed three variables to represent the target phenotype and selected up to 50 perturbable variables (yielding as many as 3^{50} possible perturbations). To evaluate C-FIX-2 and C-MTS-2, we augment each network with a pseudo-random phenotype specification. Here, the specification is chosen as follows: we first compute an arbitrary minimal trap space using `tsconj`. We then randomly select the values of three fixed variables — excluding all the source variables of the network. The conjunction of these values represents the tested phenotype. This process ensures

that for each Boolean network, problem C-MTS-2 always has at least one valid minimal trap space solution (existence of a fixed point solution cannot be guaranteed regardless of the chosen phenotype). Note that in real world scenario, phenotypes are typically not published in machine-readable format. Our benchmark preparation maintains the biological interpretability by deriving the phenotype from a known trap space, linking it to an existing biological feature of the network. Finally, to evaluate C-FIX-3 and C-MTS-3, we use the same phenotype but also pseudo-randomly select up to 50 perturbable variables, excluding both the source variables and those fixed by the phenotype. For networks with less than 50 such candidates, we simply select all viable variables as perturbable. We then use the transformation proposed in Definition 9 to construct a new variant of each benchmark network in which the selected variables can be perturbed. Each such perturbed network, together with the phenotype, represents the input for C-FIX-3 and C-MTS-3. As for the chosen size, only a few variables are sufficient to identify phenotypes: e.g., in [84], only 10-200 entities are needed out of 10,000. Since our tests are often significantly smaller, we scaled down the phenotypes accordingly. The benchmark set and experimental log files are available at `https://zenodo.org/records/19665913`.

Environmental Settings. All experiments were conducted on a high-performance computing cluster, with each node consisting of Intel Xeon Gold 6248 CPUs. Each benchmark instance was allocated one core, with runtime and memory limits set to 5000 seconds and 8 GB respectively, for all the tools considered.

7.4.1 Experimental Results

C-MTS-1 and C-FIX-1. The results for C-MTS-1 and C-FIX-1 are shown in Table 7.1 and Table 7.2, respectively. Each table reports the number of instances solved (i.e., instances for which a count was successfully returned) by each tool and their corresponding PAR2 scores [21] (PAR2 score is a runtime metric that also penalizes benchmark timeouts). Here, approximate counting (ApproxASP and ApproxMC) clearly outperform all existing solutions. Note that for C-FIX-1, ApproxASP and ApproxMC are roughly comparable, but ApproxASP is faster on simpler instances.

It is worth noting that unsafe formulas are quite rare in 245 real-world models,

which is consistent with the observation in [201]. All 400 randomly generated models have no unsafe formulas because of the nature of the generation [201].

	AEON	ADF	clingo	ApproxASP
#Solved	179	200	211	**364**
PAR2	7255	6923	6742	**4448**

Table 7.1: The performance comparison of different counters on C-MTS-1.

	AEON	ADF	clingo	GANAK	ApproxMC	ApproxASP
#Solved	247	217	227	317	**420**	413
PAR2	6172	6656	6493	5269	3801	**3760**

Table 7.2: The performance comparison of different counters on C-FIX-1.

C-MTS-2 and C-FIX-2. The results for C-MTS-2 and C-FIX-2 are shown in Table 7.3 and Table 7.4, respectively, using the same metric (solved instances and PAR2 score) as Table 7.1 and Table 7.2. Here, the relative performance of individual tools mirrors that observed for C-MTS-1 and C-FIX-1. However, all tools solved more instances overall, largely due to the inclusion of the phenotype property, which typically reduces the number of solutions. Since phenotype properties are generally much simpler than the update functions describing network dynamics, we expect them to simplify the counting problem—often substantially.

	AEON	clingo	ApproxASP
#Solved	231	308	**464**
PAR2	6428	5237	**2919**

Table 7.3: The performance comparison of different counters on C-MTS-2.

	AEON	clingo	GANAK	ApproxMC	ApproxASP
#Solved	252	236	333	**438**	429
PAR2	6099	6360	5030	3527	**3499**

Table 7.4: The performance comparison of different counters on C-FIX-2.

C-MTS-3 and C-FIX-3. The results for problems C-MTS-3 and C-FIX-3 are presented in Tables 7.5 and 7.6, respectively. This benchmark confirms the leading performance of ApproxASP in minimal trap space and fixed point counting as it was able to solve 644/645 and 645/645 problem instances for C-MTS-3 and C-FIX-3, respectively. Here, ApproxASP significantly outperforms even ApproxMC, and outperforms all other tools by a factor of 2× or more. In contrast to C-MTS-2 and C-FIX-2, where all tools benefited from a reduced number of solutions, the presence of perturbations in this setting generally increases both the number of solutions and the underlying complexity of BNs. The key performance differentiation lies in the tools' ability to handle *projected counting*. For this problem, the *independent support* for XOR constraints is derived from the BN perturbable variables. Since this is only a subset of the network variables, the independent support size is relatively small, reducing the size of the XORs. This results in the superior performance of ApproxASP and ApproxMC. Note that in these countings, the number of perturbable variables is at most 50 and the count is upper-bounded by 3^{50}.

	AEON	clingo	ApproxASP
#Solved	148	84	**644**
PAR2	7725	8711	**283**

Table 7.5: The performance comparison of different counters on C-MTS-3.

	AEON	clingo	GANAK	ApproxMC	ApproxASP
#Solved	248	99	286	600	**645**
PAR2	6176	8476	5757	2481	**150**

Table 7.6: The performance comparison of different counters on C-FIX-3.

A detailed experimental analysis is given in Chapter E.

7.4.2 Phenotype Robustness Analysis: A Case Study

Model no. 118 in the BBM dataset [177] represents an interaction network related to the activation of the so-called *Interferon 1*, a biochemical species closely tied to immune response present in T-cells. The model was initially derived using [2]

Table 7.7: Phenotype robustness analysis for the Interferon 1 model.

ISG	PCK	IFN	C-MTS-3	Robustness (r)
1	-	-	3486784401	1.000
-	1	-	2114072298	0.606
-	-	1	2313362673	0.663
0	0	0	478296900	0.137
1	0	0	621785970	0.178
0	1	0	478296900	0.137
0	0	1	813104730	0.233
1	1	0	782989740	0.224
1	0	1	1096362783	0.314
0	1	1	813104730	0.233
1	1	1	1409735826	0.404

and then later tuned by domain experts to correctly capture relevant biological phenotypes. It consists of 121 variables, of which 55 are "inputs", meaning they are not regulated by other variables.

The model defines three phenotype variables, *ISG* (*ISG expression antiviral response phenotype*), *PCK* (*Proinflammatory cytokine expression inflammation phenotype*), and *IFN* (*Type 1 IFN response phenotype*). As these are separate output variables of the network, each trap space can exhibit any combination of active and inactive phenotype variables.

Since the model defines response of T-cells to immunological stimuli and environmental factors, it is important to understand how these mechanisms respond to potential permanent perturbations, either due to genetic mutations or therapeutic treatments. Here, we provide an overview of the model phenotypes through the lens of phenotype robustness.

For simplicity, we selected 20 variables of the model as potential perturbation targets. In reality, this choice would be further influenced by known genetic risk factors or drug targets. Consequently, this choice results in $3^{20} = 3486784401$ admissible perturbations in our Boolean system. We then consider two phenotype variants: First, where a single phenotype variable is expected to be 1 and the remaining phenotype variables are unconstrained (i.e. they can be 0, 1, or $\star$), yielding three combinations. And second, more specific one, where each phenotype variable is fixed to either 1 or 0, yielding eight combinations.

The results for C-MTS-3 and the subsequent robustness computation are given

in Table 7.7. The first three columns describe the desired phenotype (− meaning the value is unconstrained). The C-MTS-3 column lists the number of perturbations for which said phenotype appears in the network. Finally, robustness r indicates what portion of the possible perturbations still enable the corresponding phenotype. Here, we can notice several biologically interesting outcomes:

- Regardless of perturbation, the network always exhibits a trap space with the *ISG* phenotype. The remaining phenotypes are usually also present ($r = 0.606$ and $r = 0.663$), but are significantly less robust than *ISG*. This indicates that (assuming favorable environmental conditions), expression of ISG as a response to viral activity is robust and cannot be disrupted by a perturbation.
- Among the fully defined phenotypes, 010 and 000 are the least robust while 111 is the most robust. This shows the general tendency of T-cells to reliably and consistently respond to immunological stimuli, as the 111 phenotype indicates the maximal level of immunological activity.
- Even though *ISG* is the most robust phenotype when taken in isolation, phenotypes with active *IFN* generally achieve higher robustness when other phenotypes are required to be inactive.

Such outcomes serve several functions: First, they can be used to validate (or refute) assumptions about the biological tendencies of the studied system. Second, if observations do not match model predictions, better quantitative understanding can provide possible sources of further model refinement. Third, this knowledge can enable us to better (more reliably or efficiently) select a phenotype that should be targeted by a treatment among several related but distinct cellular phenotypes. Finally, note that the number of solutions in each case is significantly higher than what would be countable using standard enumeration, underscoring the importance of dedicated counting methods, and approximate counting in particular.

Chapter 8

Lower Bounding Minimal Model Counts

Our objective is to develop methods to estimate a lower bound for the number of minimal models (see definitions in Subsection 2.1.5) of a given propositional formula. For this sake, we integrate knowledge compilation and hashing-based techniques with minimal model reasoning, thus facilitating the estimation of lower bounds. At the core, the proposed methods conceptualize minimal models of a formula as *answer sets* of a target ASP program. Additionally, our proposed methods depend on the efficiency of well-engineered ASP systems. Our approach utilizing knowledge compilation effectively counts the number of minimal models or provides a lower bound. Besides, our hashing-based method offers a lower bound with a probabilistic guarantee (see definitions in Subsection 2.4.2). The effectiveness of our proposed methods has been empirically validated on datasets from model counting competitions and itemset mining. Our methods perform better compared to existing minimal model reasoning systems.

8.1 Related Work

Given its significance in numerous reasoning tasks, minimal model reasoning has garnered considerable attention from the scientific community. Minimal models of a Boolean formula F can be computed using an iterative approach with a SAT solver [155]. The fundamental principle is as follows: for any model $\alpha \in \mathsf{MinModels}(F)$, no model of F can exist that is strictly smaller than α; thus, $F \wedge \neg\alpha$ yields no model. Conversely, if $F \wedge \neg\alpha$ returns a model, it is strictly smaller than α.

Minimal models can be efficiently determined using *unsatisfiable core*-based

MaxSAT algorithms [5]. This technique leverages the unsatisfiable core analysis commonly used in MaxSAT solvers and operates within an incremental solver to enumerate minimal models sorted by their size. In parallel, another line of research focuses on the enumeration of minimal models by applying cardinality constraints to calculate models of bounded size [70, 158]. Notably, Faber et al. [70] employed an algorithm that utilized an external solver for the enumeration of cardinality-minimal models of a given formula. Upon finding a minimal model, a blocking clause is integrated into the input formula, ensuring that these models are not revisited by the external solver.

There exists a close relationship between minimal models of propositional formula and answer sets of ASP program [23]. Beyond solving disjunctive logic programs (as detailed in Chapter 2), minimal models can also be effectively computed using specialized techniques within the context of ASP, such as *domain heuristics* [92] and *preference* relations [38].

Due to the intractability of minimal model finding, research has branched into exploring specific subclasses of positive CNF formulas where minimal models can be efficiently identified within polynomial time [14, 24]. Notably, a Horn formula possesses a singular minimal model, which can be derived in linear time using unit propagation [24]. Ben-Eliyahu and Palopoli [25] developed an *elimination algorithm* designed to find and verify minimal models for *head-cycle-free* formulas. Angiulli et al. [15] introduced the *Generalized Elimination Algorithm* (GEA), capable of identifying minimal models across any positive formula when paired with a suitably chosen *eliminating operator*. The efficiency of the GEA hinges on the complexity of the eliminating operator used. With an appropriate eliminating operator, the GEA can determine minimal models of *head-elementary-set-free* CNF formulas in polynomial time, which is a broader superclass of the head-cycle-free subclass.

Graph-theoretic properties have been effectively utilized in the reasoning about minimal models. Specifically, Angiulli et al. [15] demonstrated that minimal models of positive CNF formulas can be decomposed based on the structure of their dependency graph. Furthermore, they introduced an algorithm that leverages model decomposition, utilizing the underlying dependency graph to facilitate the discovery of minimal models. This approach underscores the utility of graph-theoretic concepts in enhancing the efficiency and understanding of minimal model reasoning.

To the best of our knowledge, the literature on minimal model counting is relatively sparse. The complexity of counting minimal models for specific structures of Boolean formulas, such as Horn, dual Horn, bijunctive, and affine, has been established as #P [60]. This complexity is notably lower than the general case complexity, which is $\# \cdot$ co-NP-complete [142].

8.2 Estimating Minimal Model Count

In this section, we introduce our proposed methods for determining a lower bound for the number of minimal models of a Boolean formula. We detail two specific approaches aimed at estimating this number. The first method is based on the *decomposition* of the input formula, whereas the second method utilizes a hashing-based approach of approximate model counting.

8.2.1 Formula Decomposition and Minimal Model Counting

Considering a Boolean formula $F = F_1 \wedge F_2$, we define the components F_1 and F_2 as *disjoint* if no variable of F is mentioned by both components F_1 and F_2 (i.e., $\mathsf{Var}(F_1) \cap \mathsf{Var}(F_2) = \emptyset$). Under this condition, the models of F can be independently derived from the models of F_1 and F_2 and the vice versa. Thus, if F_1 and F_2 are disjoint in the formula $F = F_1 \wedge F_2$, the total number of models of F is the product of the number of models of F_1 and F_2. This principle underpins the decomposition frequently applied in knowledge compilation [149].

Building on the concept of the knowledge compilation techniques, we introduce a strategy centered on formula decomposition to count minimal models. Unlike methods that count models for each disjoint component, we enumerate minimal models of F projected onto the variables of disjoint components. Our approach incorporates a level of enumeration that stops upon enumerating a specific count of minimal models, thereby providing a lower bound estimate of the total number of minimal models. Our method utilizes a straightforward "Cut" mechanism to facilitate formula decomposition.

Formula Decomposition by "Cut" A "cut" $\mathcal{C}$ within a formula F is identified as a subset of $\mathsf{Var}(F)$ such that for every assignment $\tau \in 2^{\mathcal{C}}$, $F|_\tau$ *effectively decomposes*

into disjoint components [149]. This concept is often used in context of model counting [147]. It is important to note that models of $F|_\tau$ can be directly expanded into models of F.

Challenges in Knowledge Compilation for Counting Minimal Models When it comes to counting minimal models, the straightforward application of unit propagation and the conventional decomposition approach are not viable [131]. More specifically, simple unit propagation does not preserve minimal models. Additionally, the count of minimal models cannot be simply calculated by multiplying the counts of minimal models of its disjoint components. An example provided below demonstrates these inconsistencies.

Example 8.1. *Consider a formula* $F = \{a \vee b \vee c, \neg a \vee \neg b \vee d, \neg a \vee \neg b \vee e\}$.
(i) With the assignment $\tau_1 = \{e\}$. *Then* $\{a\}$ *becomes a minimal model of* $F|_{\tau_1}$. *However, the extended assignment* $\tau_1 \cup \{a\}$ *is not a minimal model of* F.
(ii) Considering a cut $\mathcal{C} = \{a, b\}$ *and the partial assignment* $\tau_2 = \{a, b\}$, *then* $F|_{\tau_2}$ *is decomposed into two components, each containing the unit clauses* $\{d\}$ *and* $\{e\}$, *respectively. Despite this, the combined assignment* $\tau_2 \cup \{d, e\} = \{a, b, d, e\}$ *is not a minimal model of* F, *as a strictly smaller assignment* $\{a, d, e\}$ *also satisfies* F.

Traditional methods such as unit propagation and formula decomposition cannot be straightforwardly applied to minimal model counting. Importantly, every atom in a minimal model must be justified (this notion of justifiedness is similar to the notion of justification discussed in Chapter 4). In Example 8.1, (i) the variable e is assigned truth values without justification, leading to an incorrect minimal model when the assignment is extended with the minimal model of $F|_{\{e\}}$. (ii) The formula is decomposed without justifying the variables a and b, resulting in incorrect minimal model when combining assignments from the other two components of $F|_{\{a,b\}}$. Therefore, for accurate minimal model counting, operations such as unit propagation and formula decomposition must be applied only to assignments that are justified. Consequently, a knowledge compiler for minimal model counting must frequently verify the justification of assignments. It is worth noting that verifying the justification of an assignment is computationally intractable.

Minimal Model Counting using Justified Assignment We introduce the concept of a *justified assignment* $\tau^\star$, based on a given assignment τ. Within the minimal model semantics, any assignment of false is inherently justified. Therefore, we define justified assignment $\tau^\star$ as follows: $\tau^\star = \tau_{\downarrow\{v \in \mathsf{Var}(F) | \tau(v)=0\}}$ (the justification definition has similarity with the justification we have discussed in Chapter 4).

By applying unit propagation of $\tau^\star$, instead of τ, every minimal model derived from $F|_{\tau^\star}$ can be seamlessly extended into a minimal model of F. While $F|_\tau$ effectively decompose into multiple disjoint components, the justified assignment $\tau^\star$ does not necessarily lead to effectively $F|_{\tau^\star}$ decomposing into disjoint components.

A basic approach to counting minimal models involves enumerating all minimal models. When a formula is decomposed into multiple disjoint components, the number of minimal models can be determined by conducting a projected enumeration over these disjoint variable sets and subsequently multiplying the counts of projected minimal models. The following corollary outlines how projected enumeration can be employed across disjoint variable sets to accurately count the minimal models.

Corollary 8.1. *Let F be decomposed into disjoint components $F_1, \ldots, F_k$, with each component F_i having a variable set $V_i = \mathsf{Var}(F_i)$, for $i \in [1, k]$. Suppose $V = \mathsf{Var}(F) = \bigcup_i V_i$. Then, $|\mathsf{MinModels}(F)| = \prod_{i=1}^{k} |\mathsf{MinModels}(F)_{\downarrow V_i}|$.*

Algorithm: Counting Minimal Models by Projected Enumeration We introduce an enumeration-based algorithm, called Proj-Enum, that leverages justified assignments and projected enumeration to accurately count the number of minimal models of a Boolean formula F. The algorithm takes in as input a Boolean formula F and a set of variables $\mathcal{C}$, referred to as a "cut" in our context. To understand how the algorithm works, we introduce two new concepts: $\mathsf{MinModelswithBlocking}(F, \mathcal{B})$ and $\mathsf{ProjMinModels}(F, \tau, X)$. $\mathsf{MinModelswithBlocking}(F, \mathcal{B})$ finds $\sigma \in \mathsf{MinModels}(F)$ such that $\forall \tau \in \mathcal{B}, \sigma \not\models \tau$; here, $\mathcal{B}$ is a set of *blocking* clauses and each blocking clause is an assignment τ. $\mathsf{ProjMinModels}(F, \tau, X)$ enumerates the set $\{\sigma_{\downarrow X} | \sigma \in \mathsf{MinModels}(F), \sigma \models \tau\}$, where τ serves as the *conditioning* factor and X serves as the projection set. The algorithm iteratively processes minimal models of F (Line 2), starting with an initially empty set of blocking clauses ($\mathcal{B} = \emptyset$). Upon identifying a minimal model σ, the algorithm projects σ onto $\mathcal{C}$, denoting the projected set as τ.

Subsequently, Algorithm 7 enumerates all minimal models $\sigma \in \mathsf{MinModels}(F)$ that satisfy $\sigma \models \tau$.

To address the inefficiency associated with brute-force enumeration, the algorithm utilizes the concept of justified assignment and projected enumeration (Line 5). Corollary 8.1 establishes that the number of minimal models can be counted through multiplication. The notation $\mathsf{Components}(F)$ (Line 4) denotes all disjoint components of the formula F. It is important to note that if $F|_{\tau^\star}$ does not decompose into more than one component, then the projection variable set X defaults to $\mathsf{Var}(F)$, which leads to brute-force enumeration of non-projected minimal models. Finally, we add τ to $\mathcal{B}$ (Line 7) to prevent the re-enumeration of the same minimal models.

Algorithm 7 Proj-Enum(F, $\mathcal{C}$)

1: Count $\leftarrow 0$, $\mathcal{B} \leftarrow \emptyset$
2: **while** $\exists \sigma \in \mathsf{MinModelswithBlocking}(F, \mathcal{B})$ **do**
3: $\quad \tau \leftarrow \sigma_{\downarrow \mathcal{C}}$, $d \leftarrow 1$
4: $\quad$ **for each** comp $\in \mathsf{Components}(F|_{\tau^\star})$ **do**
5: $\quad\quad d = d \times |\mathsf{ProjMinModels}(F, \tau, \mathsf{Var}(\mathsf{comp}))|$
6: $\quad$ Count $\leftarrow$ Count $+ d$
7: $\quad \mathcal{B}.\mathsf{add}(\tau)$
8: **return** Count

Implementation Details of Proj-Enum. We implemented Proj-Enum using Python. To find minimal models using $\mathsf{MinModelswithBlocking}(F, \mathcal{B})$, the algorithm invokes an ASP solver on $\mathcal{DLP}(F)$ (see notations in Subsection 2.8.1). We used Clingo as the underlying ASP solver. Each assignment $\tau \in \mathcal{B}$ is incorporated as a constraint [9], which ensures that minimal models of F are preserved (see Observation 1). To compute $\mathsf{ProjMinModels}(F, \tau, X)$, Algorithm 7 employs an ASP solver on $\mathcal{DLP}(F)$, using X as the projection set. Additionally, it incorporates each literal from τ as a *facet* (see Subsection 2.2.5) into $\mathcal{DLP}(F)$ [4] to ensure that the condition τ is satisfied. We noted that the function $\mathsf{ProjMinModels}(F, \tau, X)$ requires more time to enumerate all minimal models. To leverage the benefits of decomposition, we enumerate upto a specific threshold number of minimal models (set the threshold to 10^6 in our experiment) invoking $\mathsf{ProjMinModels}(F, \tau, X)$. Consequently, our prototype either accurately counts the total number of minimal models or provides a lower bound. Employing a tree decomposition technique [108], we calculated a cut of the

formula that effectively decompose the input formula into several components. In our implementation, we computed the cut with a time limit of 100 seconds.

Analysis of Proj-Enum.

Lemma 8.1. *For Boolean formula F and a cut $\mathcal{C}$, the minimal models of F can be computed as follows:* $\mathsf{MinModels}(F) = \bigcup_{\tau \in 2^{\mathcal{C}}} \mathsf{ProjMinModels}(F, \tau, \mathsf{Var}(F))$.

Proof. Corollary 8.1 demonstrates that minimal models can be computed through component decompositions. By taking the union over $\tau \in 2^{\mathcal{C}}$, we iterate over all possible assignments of $\mathcal{C}$. Consequently, Proj-Enum algorithm computes all minimal models of F by conditioning over all possible assignments over $\mathcal{C}$. Therefore, the algorithm is correct. □

8.2.2 Hashing-based Minimal Model Counting

The number of minimal models can be approximated using a hashing-based model counting technique, which adds constraints that restrict the search space. Specifically, this method applies *uniform and random* (see Subsection 2.3.2) XOR constraints to a formula F, focusing the search on a smaller subspace [105]. A particular XOR-based model counter demonstrates that if t trials are conducted where s random and uniform XOR constraints are added each time, and the constrained formula of F is satisfiable in all t cases, then F has at least $2^{s-\alpha}$ models with high confidence, where α is the *precision slack* [104]. Each XOR constraint incorporates variables from $\mathsf{Var}(F)$. Similar to ApproxASP (Chapter 5), our approach to minimal model counting fundamentally derives from the strategy of introducing random and uniform XOR constraints to the formula. In the domain of approximate model counting, the XOR constraints consist of variables from a subset of $\mathsf{Var}(F)$, denoted as $\mathcal{X}$ within our algorithm, which is widely known as *independent support* [48, 192].

Algorithm 8 outlines a hashing-based algorithm, named HashCount, for determining the lower bound of minimal models of a Boolean formula F. This algorithm takes in a Boolean formula F, an independent support $\mathcal{X}$, and a confidence parameter δ. During its execution, the algorithm generates total $|\mathcal{X}|-1$ random and uniform XOR constraints, denoted as Q^i, where i ranges from 1 to $|\mathcal{X}|-1$. To better explain the op-

eration of the algorithm, we introduce a notation: $\mathsf{MinModels}(F^m)$ represents the minimal models of F satisfying first m XOR constraints, $Q^1, \ldots, Q^m$. Upon generating random and uniform XOR constraints, the algorithm finds the value of m such that $|\mathsf{MinModels}(F^m)| > 0$ (meaning that $\exists \sigma \in \mathsf{MinModels}(F), \sigma \models Q^1 \wedge \ldots \wedge Q^m$), while $|\mathsf{MinModels}(F^{m+1})| = 0$ (meaning that $\nexists\, \sigma \in \mathsf{MinModels}(F), \sigma \models Q^1 \wedge \ldots \wedge Q^{m+1}$) by iterating a loop (Line 7). The loop terminates either when a Timeout occurs or when it successfully identifies the value of m. If the Timeout happens, the algorithm assigns the maximum observed value of m (denoted as $\hat{\mathsf{m}}$) to $\mathsf{m}^\star$, ensuring that $|\mathsf{MinModels}(F^{\hat{\mathsf{m}}})| \geq 1$ (Line 8), which is sufficient to offer lower bounds. Finally, Algorithm 8 returns $2^{\mathsf{m}^\star - \alpha}$ as the probabilistic lower bound of $|\mathsf{MinModels}(F)|$.

Algorithm 8 $\mathsf{HashCount}(F, \mathcal{X}, \delta)$

1: $\alpha \leftarrow -\log_2(\delta) + 1$
2: generate $|\mathcal{X}|-1$ random constraints, namely $Q^1, \ldots, Q^{|\mathcal{X}|-1}$
3: $\mathsf{hasMinModels}[0] \leftarrow 1, \mathsf{hasMinModels}[|\mathcal{X}|] \leftarrow 0$
4: $\mathsf{loIndex} \leftarrow 0, \mathsf{hiIndex} \leftarrow |\mathcal{X}|, m \leftarrow 1$
5: $\hat{\mathsf{m}} \leftarrow \perp, \mathsf{m}^\star \leftarrow \perp$
6: **for** $i \leftarrow 1$ to $|\mathcal{X}|-1$ **do** $\mathsf{hasMinModels}[i] \leftarrow \perp$
7: **while** true **do**
8: **if** Timeout **then** $\mathsf{m}^\star \leftarrow \hat{\mathsf{m}}$ **break**
9: **if** $\exists \sigma \in \mathsf{MinModels}(F^m)$ **then**
10: $\hat{\mathsf{m}} \leftarrow \mathsf{Max}(\hat{\mathsf{m}}, m)$
11: **if** $\mathsf{hasMinModels}[m+1] = 0$ **then** $\mathsf{m}^\star \leftarrow m$ **break**
12: **for** $i \leftarrow 1$ to m **do** $\mathsf{hasMinModels}[i] \leftarrow 1$
13: $\mathsf{loIndex} \leftarrow m$
14: **if** $2 \times m < |\mathcal{X}|$ **then** $m \leftarrow 2 \times m$
15: **else** $m \leftarrow \frac{(\mathsf{hiIndex}+m)}{2}$
16: **else**
17: **if** $\mathsf{hasMinModels}[m-1] = 1$ **then** $\mathsf{m}^\star \leftarrow m-1$ **break**
18: **for** $i \leftarrow m$ to $|\mathcal{X}|-1$ **do** $\mathsf{hasMinModels}[i] \leftarrow 0$
19: $\mathsf{hiIndex} \leftarrow m$
20: $m \leftarrow \frac{(\mathsf{loIndex}+m)}{2}$
21: **return** $2^{\mathsf{m}^\star - \alpha}$

Analysis of HashCount. We adopt the following notation: $s^\star = \log_2 |\mathsf{MinModels}(F)|$. Each minimal model σ of F is an assignment over $\mathsf{Var}(F)$, and according to the definition of random XOR constraint [103], σ satisifies a random XOR constraint

with probability of $1/2$. Due to uniformity and randomness of XOR constraints, each minimal model of F satisfies m random and uniform XOR constraints with probability of $1/2^m$. In our theoretical analysis, we apply the Markov inequality: if Y is a non-negative random variable, then $\mathsf{Pr}[Y \geq a] \leq \frac{\mathbb{E}[Y]}{a}$, where $a > 0$.

Lemma 8.2. *For arbitrary s, $\mathsf{Pr}[|\mathsf{MinModels}(F^s) \geq 1|] \leq \frac{2^{s^\star}}{2^s}$.*

Proof. For each $\sigma \in \mathsf{MinModels}(F)$, we define a random variable $Y_\sigma \in \{0,1\}$ and $Y_\sigma = 1$ indicates that σ satisifies the first s XOR constraints $Q^1, \ldots, Q^s$, otherwise, $Y_\sigma = 0$. The random variable Y is the summation, $Y = \sum_{\sigma \in \mathsf{MinModels}(F)} Y_\sigma$. The expected value of Y can be calculated as $\mathbb{E}(Y) = \sum_{\sigma \in \mathsf{MinModels}(F)} \mathbb{E}(Y_\sigma)$.

Due to the nature of random and uniform XOR constraints, each minimal model $\sigma \in \mathsf{MinModels}(F)$ satisfies all s XOR constraints with probability $\frac{1}{2^s}$. It follows that the expected value $\mathbb{E}(Y_\sigma) = \frac{1}{2^s}$, and the expected value of Y is $\mathbb{E}(Y) = \frac{|\mathsf{MinModels}(F)|}{2^s} = \frac{2^{s^\star}}{2^s}$. According to the Markov inequality: $\mathsf{Pr}[|\mathsf{MinModels}(F^s) \geq 1|] \leq \frac{2^{s^\star}}{2^s}$. □

Lemma 8.3. *Given a formula F and confidence δ, if $\mathsf{HashCount}(F, \delta)$ returns $2^{s-\alpha}$, then $\mathsf{Pr}[2^{s-\alpha} \leq |\mathsf{MinModels}(F)|] \geq 1 - \delta$*

Proof. Given a input Boolean formula F and for each $m \in [1, |\mathcal{X}|-1]$, we denote the following two events: I_m denotes the event that Algorithm 8 invokes $\mathsf{MinModels}(F^m)$ and E_m denotes the event that $|\mathsf{MinModels}(F^m)| \geq 1$. The algorithm $\mathsf{HashCount}(F, \delta)$ returns an incorrect bound when $s - \alpha > s^\star$ and let use the notation $\mathcal{R}$ to denote that $\mathsf{HashCount}(F, \delta)$ returns an incorrect bound or $\mathsf{HashCount}(F, \delta) > 2^{s^\star}$. The upper bound of $\mathsf{Pr}[\mathcal{R}]$ can be calculated as follows:

$$
\begin{aligned}
\mathsf{Pr}[\mathcal{R}] &= \mathsf{Pr}[\mathsf{HashCount}(F, \delta) \text{ returns } 2^{s-\alpha} \text{ and } s > s^\star + \alpha] \\
&\leq \sum_{s > s^\star + \alpha} \mathsf{Pr}[I_s \cap E_s] \leq \sum_{s > s^\star + \alpha} \mathsf{Pr}[E_s] \\
&\leq \sum_{s > s^\star + \alpha} \mathsf{Pr}[|\mathsf{MinModels}(F^s)| \geq 1] \leq \sum_{s > s^\star + \alpha} \frac{2^{s^\star}}{2^s} \qquad \text{According to Lemma 8.2} \\
&\leq \frac{1}{2^\alpha} \times 2 \leq 2^{1-\alpha} \leq 2^{\log_2 \delta} \leq \delta
\end{aligned}
$$

Thus, $\mathsf{Pr}[\mathsf{HashCount}(F, \delta) \text{ returns } 2^{s-\alpha} \text{ and } s \leq s^\star + \alpha] \geq 1 - \delta$. □

Implementation Details of HashCount The effectiveness of an XOR-based model counter is dependent on the performance of a theory+XOR solver [191]. In our approach, we begin by transforming a given formula F into a disjunctive logic program $\mathcal{DLP}(F)$ and introduce random and uniform XOR constraints into $\mathcal{DLP}(F)$ to effectively partition the minimal models of F. To verify the presence of any models in the XOR-constrained ASP program, we leverage the ASP+XOR solver capabilities provided by ApproxASP [133]. To compute an independent support for HashCount, we implemented a prototype inspired by Arjun [192], which checks answer sets of a disjunctive logic program in accordance with Padoa's theorem [175]. In our implementation, we computed an independent support with a time limit of 200 seconds.

8.2.3 Putting It All Together

Algorithm 9 MinLB(F, δ)

1: **if** $\nexists\, \sigma \in \mathsf{MinModels}(F)$ **then return** 0
2: **if** $|\mathsf{Cut}(F)| \leq 50$ **then return** Proj-Enum$(F, \mathsf{Cut}(F))$
3: **else return** HashCount$(F, \mathsf{IndependentSupport}(F), \delta)$

We designed a hybrid solver MinLB, presented in Algorithm 9, that selects either Proj-Enum or HashCount depending on the decomposability of the input formula. The core principle of MinLB is that Proj-Enum effectively leverages decomposition and projected enumeration on *easily decomposable* formulas. We use $|\mathsf{Cut}(F)|$ as a proxy to measure the decomposability. Thus, if $|\mathsf{Cut}(F)|$ is small, then MinLB employs Proj-Enum on F (Line 2); otherwise, it employs HashCount (Line 3). In our implementation, we used a tree-decomposition-based technique to compute the cut, with a time limit of 200 seconds.

Theorem 11. $\Pr[\mathsf{MinLB}(F, \delta) \leq |\mathsf{MinModels}(F)|] \geq 1 - \delta$

Proof. The proof consists of two cases.
(i) When MinLB calls Proj-Enum (if $|\mathcal{C}| \leq 50$): in the case, the proof follows Lemma 8.1.
(ii) MinLB calls HashCount: in the case, the proof follows Lemma 8.3. □

8.3 Experimental Evaluation

Benchmarks and Baselines Our benchmark set is collected from two different domains: (i) model counting benchmarks from recent competitions [77] and (ii) minimal generators benchmark from itemset mining dataset CP4IM[1]. We used existing systems for minimal model reasoning as baselines. These included various approaches such as (i) repeated invocations of the SAT solver MiniSAT [155], (ii) the application of MaxSAT techniques [5], (iii) domain-specific heuristics [92], and (iv) solving $\mathcal{DLP}(F)$ with ASP solvers, all systems primarily count via enumeration. These systems either return the number of minimal models by enumerating all of them or a lower bound of the number of minimal models in cases where they run out of time or memory. We cannot count minimal models by #SAT-based ASP counters [132] because of their incapablity of handling disjunctive logic programs. Experimentally, we observed that, for enumerating all minimal models, the technique solving disjunctive ASP programs using clingo [91] surpassed the other techniques. Therefore, we have exclusively reported the performance of clingo in our experimental analysis. Additionally, we evaluated ApproxASP [133], which offers (ϵ, δ)-guarantees in counting minimal models. We attempted an exact minimal model counting tool using a subtractive reduction approach — subtracting the *non-minimal model* count from the total model count—we denote the implementation using the notation #MinModels in further analysis. In our experiment, we ran HashCount with a confidence δ value of 0.2 and ApproxASP with confidence $\delta = 0.2$ and tolerance $\epsilon = 0.8$. The prototype of MinLB is available at https://github.com/meelgroup/MinLB; the benchmarks and experimental logfiles are available at https://zenodo.org/records/19757334.

Environmental Settings All experiments were conducted on a high-performance computing cluster equipped with nodes featuring AMD EPYC 7713 CPUs, each with 128 real cores. Throughout the experiment, the runtime and memory limits were set to 5000 seconds and 16GB, respectively, for all considered tools.

[1] https://dtai-static.cs.kuleuven.be/CP4IM/datasets/

Evaluation Metric The goal of our experimental analysis is to evaluate various minimal model counting tools based on their runtime and the quality of their lower bounds. It is essential to employ a metric that encompasses both runtime performance and the quality of the lower bound. Consequently, following the TAP score [3, 134], we have introduced a metric, called the *Time Quality Penalty* (TQP) score, which is defined for each tool and instance as follows:

$$\mathsf{TQP}(t, C) = \begin{cases} 2 \times \mathcal{T}, & \text{if no lower bound is returned} \\ t + \mathcal{T} \times \frac{1+\log(\mathsf{C_{min}}+1)}{1+\log(C+1)}, & \text{otherwise} \end{cases}$$

In the metric, $\mathcal{T}$ represents the timeout for the experiment, t denotes the runtime of the tool, C is the lower bound returned by the tool, and $\mathsf{C_{min}}$ is the minimum lower bound returned by any of the tools under consideration for the instance. The TQP score is based on the following principle: lower runtime and higher lower bound yield a better score.

To facilitate the comparison of lower bounds returned by two tools, we have introduced another metric for comparative analysis. If tools A and B yield lower bounds C_A and C_B respectively, their *relative quality* is defined as follows:

$$r_{AB} = \frac{1 + \log(C_A + 1)}{1 + \log(C_B + 1)} \tag{8.1}$$

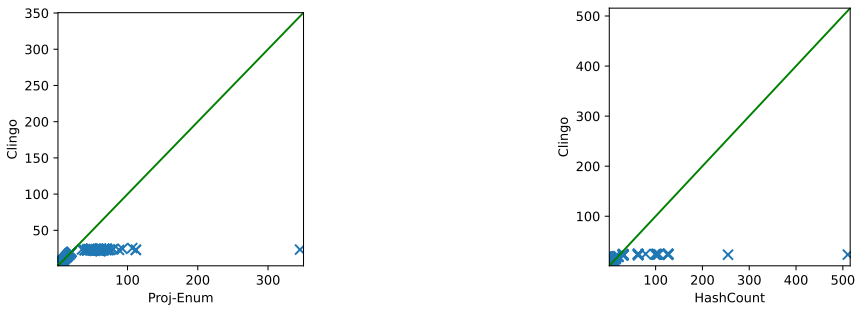
If $r_{AB} > 1$, then the lower bound returned by tool A is superior to that of tool B.

8.3.1 Experimental Results: Model Counting Benchmark

We present the TQP scores of MinLB alongside other tools in Table 8.1. This table indicates that MinLB achieves the lowest TQP scores. Among existing minimal model enumerators, clingo demonstrates the best performance in terms of TQP score. Additionally, in Figure 8.1, we graphically compare the lower bounds returned by Proj-Enum and HashCount against those returned by Clingo. Here, a point (x, y) indicates that, for an instance, the lower bounds returned by our prototypes and clingo are 2^x and 2^y, respectively. For an instance, if the corresponding point resides below the diagonal line, it indicates that Proj-Enum (HashCount, resp.) returns a better lower bound than Clingo. These plots illustrate that Proj-Enum and HashCount return better lower bounds compared to existing minimal model enumerators.

Clingo	ApproxASP	#MinModels	MinLB (our prototype)
6491	6379	7743	5599

Table 8.1: The TQP scores of MinLB and other tools on model counting benchmark.



(a) The lower bound returned by Proj-Enum (b) The lower bound returned by HashCount

Figure 8.1: The lower bound of Proj-Enum and HashCount vis-a-vis the lower bound returned by Clingo on minimal model counting benchmark. The axes are in log scale.

8.3.2 Experimental Results: Minimal Generator Benchmark

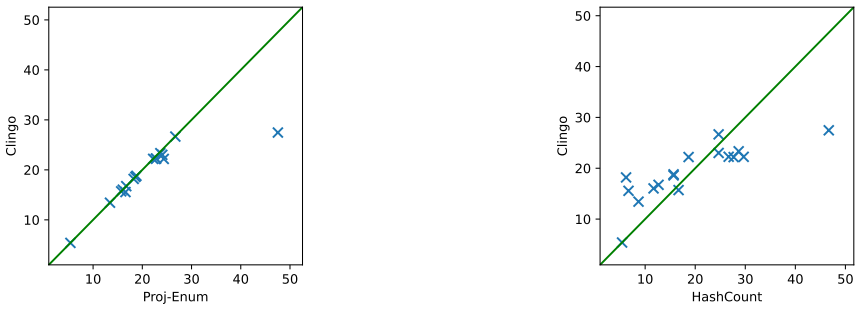
Table 8.2 showcases the TQP scores of MinLB alongside other tools on the minimal generator benchmark. Notably, HashCount achieves the most favorable TQP scores on the benchmark. Additionally, Figure 8.2 graphically compares the lower bounds returned by Proj-Enum and HashCount against those computed by Clingo. In the figures, the axes are in log scale.

Clingo	ApproxASP	#MinModels	MinLB
6944	5713	9705	5043

Table 8.2: The TQP scores of MinLB and baselines on minimal generator benchmark.

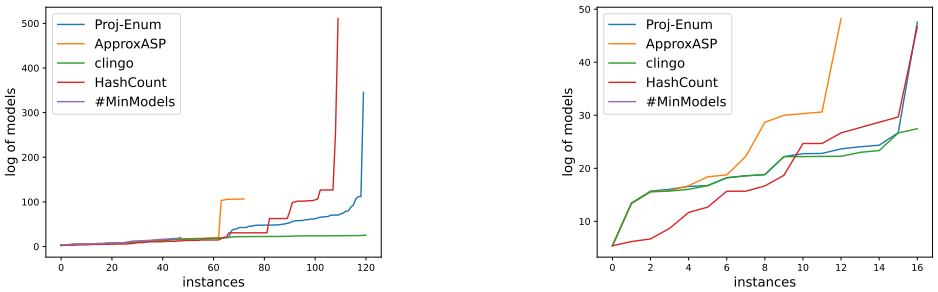
For a visual representation of the lower bounds returned by our HashCount and Proj-Enum, we illustrate them graphically in Figure 8.3. In the plot, a point (x, y) signifies that a tool returns a lower bound of at most 2^y for x instances. The plot demonstrates that the lower bounds returned by Proj-Enum and HashCount surpass those of existing systems.

Further experimental analysis is deferred to Chapter F.



(a) The lower bound returned by Proj-Enum (b) The lower bound returned by HashCount

Figure 8.2: The lower bound returned by Proj-Enum and HashCount vis-a-vis the lower bound given by Clingo on minimal generators benchmark.



(a) Model counting benchmark (b) Minimal generators benchmark

Figure 8.3: The lower bounds returned by Proj-Enum, HashCount, and existing minimal model counting tools. The y-axis show the log of the number of models.

Chapter 9

Conclusion

The thesis proposed efficient answer set counting techniques and solved real-world problems exploiting efficient answer set counters. In the first part, we introduce exact answer set counters sharpASP and sharpASP-$\mathcal{SR}$, leveraging an alternative definition of answer sets. Firstly, sharpASP adapts component caching-based propositional model counting to ASP counting without incurring a blowup in the size of the resulting formula. Secondly, sharpASP-$\mathcal{SR}$ is an answer set counter based on subtractive reduction, leveraging to its ability to rely on the scalability of state-of-the-art projected model counters. Our experimental evaluation reveals that sharpASP, sharpASP-$\mathcal{SR}$ and their corresponding hybrid counters can handle a greater number of instances, compared to other techniques.

We present ApproxASP, the first scalable approximate counter for ASP programs that employs pairwise independent hash functions, represented as XOR constraints, to partition the solution space, and then invokes an ASP solver on a randomly chosen cell. To achieve practical efficiency, we augment the state of the art ASP solver, Clingo, with native support for XORs. Our empirical evaluation clearly demonstrates that ApproxASP can tackle problems that lie beyond the reach of existing counting techniques. The empirical analysis, therefore, positions ApproxASP as the tool of choice in the context of counting for ASP programs.

In the second part, we demonstrate applications of answer set counting. We introduce a novel approach for estimating network reliability by combining ASP counting and theories derived from weighted model counting. The proposed tool, RelNet-ASP, leverages the expressive modeling capabilities of ASP and incorporates the latest advancements in ASP counting techniques. The experimental evaluation demonstrates the scalability of RelNet-ASP in terms of accuracy and efficiency, outperforming existing tools for network reliability estimation. These findings

highlight the potential of ASP as a powerful formalism for reliability analysis.

We demonstrate applications of answer set counting in systems biology and address the problem of counting minimal trap spaces and fixed points in Boolean networks. We propose novel methods for determining trap space and fixed point counts using approximate answer set counting, thus entirely avoiding costly enumeration. We apply this methodology to three biologically motivated problems: (a) general counting; (b) counting occurrences of a known biological phenotype, and (c) projected counting of perturbable variables that ensure the emergence of a known biological phenotype. Through extensive experiments on a diverse set of benchmarks, we show that approximate counting substantially improves the feasibility of counting in this domain, outperforming traditional enumeration-based and exact approaches whenever applicable.

Leveraging the expressive power of ASP semantics and efficiency of well-engineered ASP systems, we introduced two innovative methods for computing a lower bound on the number of minimal models. The first method, Proj-Enum, leveraged knowledge compilation techniques to provide improved lower bounds for easily decomposable formulas. The second method, HashCount, builds on recent advancements in ASP+XOR reasoning systems of ApproxASP and demonstrated performance that varies with the size of the independent support.

Bibliography

[1] J. A. Abraham, "An improved algorithm for network reliability", in *IEEE Transactions on Reliability*, vol. 28, IEEE, 1979, pp. 58–61.

[2] S. S. Aghamiri, V. Singh, A. Naldi, T. Helikar, S. Soliman, A. Niarakis, and J. Xu, "Automated inference of Boolean models from molecular interaction maps using CaSQ", *Bioinform.*, vol. 36, no. 16, pp. 4473–4482, 2020.

[3] D. Agrawal, Y. Pote, and K. S. Meel, "Partition function estimation: A quantitative study", in *IJCAI*, Aug. 2021.

[4] C. Alrabbaa, S. Rudolph, and L. Schweizer, "Faceted answer set navigation", in *RuleML+RR*, Springer, 2018, pp. 211–225.

[5] M. Alviano, "Model enumeration in propositional circumscription via unsatisfiable core analysis", *TPLP*, vol. 17, no. 5-6, pp. 708–725, 2017.

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

[7] M. Alviano and C. Dodaro, "Completion of disjunctive logic programs." In *IJCAI*, vol. 16, 2016, pp. 886–892.

[8] M. Alviano, C. Dodaro, W. Faber, N. Leone, and F. Ricca, "Wasp: A native ASP solver based on constraint learning", in *LPNMR*, 2013, pp. 54–66.

[9] M. Alviano, C. Dodaro, S. Fiorentino, A. Previti, and F. Ricca, "Enumeration of minimal models and MUSes in Wasp", in *LPNMR*, Springer, 2022, pp. 29–42.

[10] M. Alviano, C. Dodaro, S. Fiorentino, A. Previti, and F. Ricca, "ASP and subset minimality: Enumeration, cautious reasoning and MUSes", *Artif. Intell.*, vol. 320, p. 103 931, 2023.

[11] M. Alviano, C. Dodaro, N. Leone, and F. Ricca, “Advances in Wasp”, in *LPNMR*, Springer, 2015, pp. 40–54.

[12] M. Alviano, W. Faber, and M. Gebser, “Aggregate semantics for propositional answer set programs”, *Theory and Practice of Logic Programming*, vol. 23, no. 1, pp. 157–194, 2023.

[13] G. Amendola, F. Ricca, and M. Truszczynski, “Generating hard random boolean formulas and disjunctive logic programs.” In *IJCAI*, 2017, pp. 532–538.

[14] F. Angiulli, R. Ben-Eliyahu, F. Fassetti, and L. Palopoli, “On the tractability of minimal model computation for some CNF theories”, *Artificial Intelligence*, vol. 210, pp. 56–77, 2014.

[15] F. Angiulli, R. Ben-Eliyahu, F. Fassetti, and L. Palopoli, “Graph-based construction of minimal models”, *Artificial Intelligence*, vol. 313, p. 103 754, 2022.

[16] R. A. Aziz, G. Chu, C. Muise, and P. Stuckey, “#∃SAT: Projected model counting”, in *SAT*, Springer, 2015, pp. 121–137.

[17] R. A. Aziz, G. Chu, C. Muise, and P. J. Stuckey, “Stable model counting and its application in probabilistic logic programming”, in *AAAI*, 2015.

[18] E. Azpeitia, S. M. Gutiérrez, D. A. Rosenblueth, and O. Zapata, “Bridging abstract dialectical argumentation and Boolean gene regulation”, *CoRR*, vol. abs/2407.06106, 2024. arXiv: 2407.06106.

[19] M. Balduccini and Y. Lierler, “Constraint answer set solver EZCSP and why integration schemas matter”, *Theory and Practice of Logic Programming*, vol. 17, no. 4, pp. 462–515, 2017.

[20] T. Baluta, S. Shen, S. Shinde, K. S. Meel, and P. Saxena, “Quantitative verification of neural networks and its security applications”, in *CCS*, 2019, pp. 1249–1264.

[21] T. Balyo, M. J. Heule, and M. Järvisalo, “SAT competition 2017–solver and benchmark descriptions”,, pp. 14–15, 2017.

[22] R. J. Bayardo Jr and J. D. Pehoushek, "Counting models using connected components", in *AAAI/IAAI*, 2000, pp. 157–162.

[23] R. Ben-Eliyahu and R. Dechter, "Propositional semantics for disjunctive logic programs", *Annals of Mathematics and Artificial intelligence*, vol. 12, pp. 53–87, 1994.

[24] R. Ben-Eliyahu and R. Dechter, "On computing minimal models", *Annals of Mathematics and Artificial Intelligence*, vol. 18, no. 1, pp. 3–27, 1996.

[25] R. Ben-Eliyahu and L. Palopoli, "Reasoning with minimal models: Efficient algorithms and applications", *Artificial Intelligence*, vol. 96, no. 2, pp. 421–449, 1997.

[26] R. Ben-Eliyahu-Zohary, F. Angiulli, F. Fassetti, and L. Palopoli, "Modular construction of minimal models", in *LPNMR*, Springer, 2017, pp. 43–48.

[27] J. Bendík and K. S. Meel, "Counting minimal unsatisfiable subsets", in *CAV*, Springer, 2021, pp. 313–336.

[28] N. Benes, L. Brim, J. Kadlecaj, S. Pastva, and D. Safránek, "AEON: attractor bifurcation analysis of parametrised Boolean networks", in *CAV*, Springer, 2020, pp. 569–581.

[29] N. Benes, L. Brim, S. Pastva, D. Safránek, and E. Smijáková, "Phenotype control of partially specified Boolean networks", in *CMSB*, Springer, 2023, pp. 18–35.

[30] F. Biondi, M. A. Enescu, A. Heuser, A. Legay, K. S. Meel, and J. Quilbeuf, "Scalable approximation of quantitative information flow in programs", in *VMCAI*, Springer, 2018, pp. 71–93.

[31] P. Bloomingdale, V. A. Nguyen, J. Niu, and D. E. Mager, "Boolean network modeling in systems pharmacology", *J. Pharmacokinet. Pharmacodyn.*, vol. 45, pp. 159–180, 2018.

[32] J. Bomanson, "LP2NORMAL—a normalization tool for extended logic programs", in *LPNMR*, 2017, pp. 222–228.

[33] J. Bomanson, M. Gebser, and T. Janhunen, "Improving the normalization of weight rules in answer set programs", in *JELIA*, Springer, 2014, pp. 166–180.

[34] J. Bomanson, M. Gebser, and T. Janhunen, "Rewriting Optimization Statements in Answer-Set Programs", in *Tech. Comm. of ICLP*, vol. 52, 2016, 5:1–5:15.

[35] J. Bomanson and T. Janhunen, "Normalizing cardinality rules using merging and sorting constructions", in *LPNMR*, Springer, 2013, pp. 187–199.

[36] J. Bomanson, T. Janhunen, and A. Weinzierl, "Enhancing lazy grounding with lazy normalization in answer-set programming", in *AAAI*, vol. 33, 2019, pp. 2694–2702.

[37] Z. I. Botev, P. L'Ecuyer, G. Rubino, R. Simard, and B. Tuffin, "Static network reliability estimation via generalized splitting", in *INFORMS Journal on Computing*, vol. 25, INFORMS, 2013, pp. 56–71.

[38] G. Brewka, J. Delgrande, J. Romero, and T. Schaub, "Asprin: Customizing answer set preferences without a headache", in *AAAI*, vol. 29, 2015.

[39] A. Brik and J. Remmel, "Diagnosing automatic whitelisting for dynamic remarketing ads using hybrid ASP", in *LPNMR*, Springer, 2015, pp. 173–185.

[40] D. R. Brooks, E. Erdem, S. T. Erdoğan, J. W. Minett, and D. Ringe, "Inferring phylogenetic trees using answer set programming", *Journal of Automated Reasoning*, vol. 39, no. 4, p. 471, 2007.

[41] P. Cabalar and J. Fandinno, "Justifications for programs with disjunctive and causal-choice rules", *TPLP*, vol. 16, no. 5-6, pp. 587–603, 2016.

[42] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, and T. Schaub, "ASP-Core-2 input language format", *TPLP*, vol. 20, no. 2, pp. 294–309, 2020.

[43] H. Cancela, M. E. Khadiri, G. Rubino, and B. Tuffin, "Balanced and approximate zero-variance recursive estimators for the network reliability problem", in *TOMACS*, vol. 25, ACM New York, NY, USA, 2014, pp. 1–19.

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

[45] L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)", in *STOC*, J. E. Hopcroft, E. P. Friedman, and M. A. Harrison, Eds., ACM, 1977, pp. 106–112. [Online]. Available: https://doi.org/10.1145/800105.803400.

[46] S. Chakraborty, D. Fried, K. S. Meel, and M. Y. Vardi, "From weighted to unweighted model counting", in *IJCAI*, 2015, pp. 689–695.

[47] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter", in *CP*, Springer, 2013, pp. 200–216.

[48] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls." In *IJCAI*, 2016, pp. 3569–3576.

[49] A. K. Chandra and G. Markowsky, "On the number of prime implicants", *Discrete Mathematics*, vol. 24, no. 1, pp. 7–11, 1978.

[50] S. Chevalier, V. Noël, L. Calzone, A. Y. Zinovyev, and L. Paulevé, "Synthesis and simulation of ensembles of Boolean networks for cell fate decision", in *CMSB*, Springer, 2020, pp. 193–209.

[51] K. L. Clark, "Negation as failure", in *Logic and data bases*, Springer, 1978, pp. 293–322.

[52] B. Cuteri, C. Dodaro, F. Ricca, and P. Schüller, "Overcoming the grounding bottleneck due to constraints in asp solving: Constraints become propagators." In *IJCAI*, vol. 20, 2020, pp. 1688–1694.

[53] P. Dagum, R. Karp, M. Luby, and S. Ross, "An optimal algorithm for Monte Carlo estimation", *SIAM Journal on computing*, vol. 29, no. 5, pp. 1484–1496, 2000.

[54] A. Dal Palu, A. Dovier, E. Pontelli, and G. Rossi, "GASP: Answer set programming with lazy grounding", *Fundamenta Informaticae*, vol. 96, no. 3, pp. 297–322, 2009.

[55] A. Darwiche, "A compiler for deterministic, decomposable negation normal form", in *AAAI/IAAI*, 2002, pp. 627–634.

[56] L. De Raedt and K. Kersting, "Probabilistic inductive logic programming", in *Probabilistic inductive logic programming: theory and applications*, Springer, 2008, pp. 1–27.

[57] C. Dodaro and M. Maratea, "Nurse scheduling via answer set programming", in *LPNMR*, Springer, 2017, pp. 301–307.

[58] E. Dubrova and M. Teslenko, "A SAT-based algorithm for finding attractors in synchronous Boolean networks", *IEEE ACM Trans. Comput. Biol. Bioinform.*, vol. 8, no. 5, pp. 1393–1399, 2011.

[59] L. Duenas-Osorio, K. Meel, R. Paredes, and M. Vardi, "Counting-based reliability estimation for power-transmission grids", in *AAAI*, vol. 31, 2017.

[60] A. Durand and M. Hermann, "On the counting complexity of propositional circumscription", *Information Processing Letters*, vol. 106, no. 4, pp. 164–170, 2008.

[61] A. Durand, M. Hermann, and P. G. Kolaitis, "Subtractive reductions and complete problems for counting complexity classes",, vol. 340, no. 3, pp. 496–513, 2005, ISSN: 0304–3975.

[62] U. Egly, T. Eiter, H. Tompits, and S. Woltran, "Solving advanced reasoning tasks using quantified boolean formulas", in *AAAI/IAAI*, 2000, pp. 417–422.

[63] T. Eiter, M. Fink, H. Tompits, and S. Woltran, "On eliminating disjunctions in stable logic programming", *KR*, vol. 4, pp. 447–458, 2004.

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

[65] T. Eiter, M. Hecher, and R. Kiesel, "Treewidth-aware cycle breaking for algebraic answer set counting", in *KR*, vol. 18, 2021, pp. 269–279.

[66] T. Eiter, M. Hecher, and R. Kiesel, "aspmc: New frontiers of algebraic answer set counting", *Artificial Intelligence*, vol. 330, p. 104 109, 2024.

[67] F. Everardo, T. Janhunen, R. Kaminski, and T. Schaub, "The return of xorro", in *LPNMR'19*, 2019, pp. 284–297.

[68] W. Faber, N. Leone, and S. Perri, "The intelligent grounder of dlv", *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, pp. 247–264, 2012.

[69] W. Faber, G. Pfeifer, N. Leone, T. Dell'armi, and G. Ielpa, "Design and implementation of aggregate functions in the DLV system", *Theory Pract. Log. Program.*, vol. 8, no. 5-6, pp. 545–580, 2008, ISSN: 1471-0684.

[70] W. Faber, M. Vallati, F. Cerutti, and M. Giacomin, "Solving set optimization problems by cardinality optimization with an application to argumentation", in *ECAI*, IOS Press, 2016, pp. 966–973.

[71] F. Fages, "Consistency of Clark's completion and existence of stable models", *Journal of Methods of logic in computer science*, vol. 1, no. 1, pp. 51–60, 1994.

[72] J. Fandinno and C. Schulz, "Answering the "why" in answer set programming–a survey of explanation approaches", *TPLP*, vol. 19, no. 2, pp. 114–203, 2019.

[73] Y.-P. Fang and E. Zio, "Unsupervised spectral clustering for hierarchical modelling and criticality analysis of complex networks", *Reliability Engineering & System Safety*, vol. 116, pp. 64–74, 2013.

[74] J. K. Fichte, S. A. Gaggl, M. Hecher, and D. Rusovac, "IASCAR: Incremental answer set counting by anytime refinement", *TPLP*, vol. 24, no. 3, pp. 505–532, 2024.

[75] J. K. Fichte and M. Hecher, "Treewidth and counting projected answer sets", in *LPNMR*, Springer, 2019, pp. 105–119.

[76] J. K. Fichte and M. Hecher, "The model counting competitions 2021-2023", 2025. arXiv: 2504.13842 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2504.13842.

[77] J. K. Fichte, M. Hecher, and F. Hamiti, "The model counting competition 2020", *Journal of Experimental Algorithmics*, vol. 26, pp. 1–26, 2021.

[78] J. K. Fichte, M. Hecher, M. A. Nadeem, and T. Dresden, "Plausibility reasoning via projected answer set counting–a hybrid approach", in *IJCAI*, vol. 22, 2022, pp. 2620–2626.

[79] J. K. Fichte and S. Szeider, “Backdoors to normality for disjunctive logic programs”, *TOCL*, vol. 17, no. 1, pp. 1–23, 2015.

[80] J. K. Fichte, S. A. Gaggl, and D. Rusovac, “Rushing and strolling among answer sets–navigation made easy”, in *AAAI*, vol. 36, 2022, pp. 5651–5659.

[81] J. K. Fichte, M. Hecher, C. McCreesh, and A. Shahab, “Complications for computational experiments from modern processors”, in *CP*, vol. 210, 2021, 25:1–25:21.

[82] J. K. Fichte, M. Hecher, M. Morak, and S. Woltran, “Answer set solving with bounded treewidth revisited”, in *LPNMR*, 2017, pp. 132–145.

[83] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, and L. De Raedt, “Inference and learning in probabilistic logic programs using weighted boolean formulas”, *TPLP*, vol. 15, no. 3, pp. 358–401, 2015.

[84] S. Fischer and J. Gillis, “How many markers are needed to robustly determine a cell’s type?” *iScience*, vol. 24, no. 11, p. 103 292, Nov. 2021, ISSN: 2589-0042.

[85] S. A. Gaggl, N. Manthey, A. Ronca, J. P. Wallner, and S. Woltran, “Improved answer set programming encodings for abstract argumentation”, *TPLP*, vol. 15, no. 4-5, pp. 434–448, 2015.

[86] M. Gebser, T. Janhunen, and J. Rintanen, “Answer set programming as SAT modulo acyclicity”, in *ECAI*, IOS Press, 2014, pp. 351–356.

[87] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko, “Theory Solving Made Easy with Clingo 5”, in *Tech. Comm. of ICLP*, vol. 52, 2016, 2:1–2:15.

[88] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, “Answer set solving in practice”, *Synthesis lectures on artificial intelligence and machine learning*, vol. 6, no. 3, pp. 1–238, 2012.

[89] M. Gebser, R. Kaminski, A. König, and T. Schaub, “Advances in Gringo series 3”, in *LPNMR*, Springer, 2011, pp. 345–351.

[90] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. Schneider, "Potassco: The Potsdam answer set solving collection", *AI Commun.*, vol. 24, no. 2, pp. 107–124, 2011.

[91] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "Conflict-driven answer set enumeration", in *LPNMR*, vol. 4483, 2007, pp. 136–148.

[92] M. Gebser, B. Kaufmann, J. Romero, R. Otero, T. Schaub, and P. Wanko, "Domain-specific heuristics in answer set programming", in *AAAI*, vol. 27, 2013, pp. 350–356.

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

[94] M. Gebser, B. Kaufmann, and T. Schaub, "Advanced conflict-driven disjunctive answer set solving", in *IJCAI*, 2013, pp. 912–918.

[95] M. Gebser, N. Leone, M. Maratea, S. Perri, F. Ricca, and T. Schaub, "Evaluation techniques and systems for answer set programming: A survey." In *IJCAI*, vol. 18, 2018, pp. 5450–5456.

[96] M. Gebser, M. Maratea, and F. Ricca, "The seventh answer set programming competition: Design and results", *TPLP*, vol. 20, no. 2, pp. 176–204, 2020.

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

[98] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming." In *ICLP/SLP*, vol. 88, 1988, pp. 1070–1080.

[99] I. B. Gertsbakh and Y. Shpungin, "Models of network reliability: analysis, combinatorics, and Monte Carlo", CRC press, 2016.

[100] A. Gittis, E. Vin, and D. J. Fremont, "Randomized synthesis for diversity and cost constraints with control improvisation", in *CAV*, Springer, 2022, pp. 526–546.

[101] E. Giunchiglia, Y. Lierler, and M. Maratea, "Answer set programming based on propositional satisfiability", *Journal of Automated reasoning*, vol. 36, no. 4, pp. 345–377, 2006.

[102] P. Glasserman, P. Heidelberger, P. Shahabuddin, and T. Zajic, "Multilevel splitting for estimating rare event probabilities", in *Operations Research*, vol. 47, Informs, 1999, pp. 585–600.

[103] C. Gomes, A. Sabharwal, and B. Selman, "Near-uniform sampling of combinatorial spaces using XOR constraints", in *NIPS*, 2007, pp. 481–488.

[104] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting: A new strategy for obtaining good bounds", in *AAAI*, vol. 10, 2006, pp. 1 597 538–1 597 548.

[105] C. P. Gomes, A. Sabharwal, and B. Selman, "Model counting", in *Handbook of satisfiability*, IOS press, 2021, pp. 993–1014.

[106] C Gomez, M Sanchez-Silva, and L. Duenas-Osorio, "Clustering methods for risk assessment of infrastructure network systems", *Applications of statistics and probability in civil engineering*, pp. 1389–1397, 2011.

[107] C. Guziolowski, S. Videla, F. Eduati, S. Thiele, T. Cokelaer, A. Siegel, and J. Saez-Rodriguez, "Exhaustively characterizing feasible logic models of a signaling network using answer set programming", *Bioinformatics*, vol. 29, no. 18, pp. 2320–2326, 2013.

[108] M. Hamann and B. Strasser, "Graph bisection with pareto optimization", *Journal of Experimental Algorithmics (JEA)*, vol. 23, pp. 1–34, 2018.

[109] C.-S. Han and J.-H. R. Jiang, "When Boolean satisfiability meets Gaussian elimination in a simplex way", in *CAV*, Springer, 2012, pp. 410–426.

[110] G. Hardy, C. Lucet, and N. Limnios, "K-terminal network reliability measures with binary decision diagrams", in *IEEE Transactions on Reliability*, vol. 56, IEEE, 2007, pp. 506–515.

[111] M. Hecher, "Treewidth-aware reductions of normal ASP to SAT–is normal ASP harder than SAT after all?" *Artificial Intelligence*, vol. 304, p. 103 651, 2022.

[112] M. Hecher and R. Kiesel, "The impact of structure in answer set counting: Fighting cycles and its limits", in *KR*, vol. 19, 2023, pp. 344–354.

[113] L. A. Hemaspaandra and H. Vollmer, "The satanic notations: Counting classes beyond #P and other definitional adventures", *ACM SIGACT News*, vol. 26, no. 1, pp. 2–13, 1995.

[114] J. Heyninck, M. Knorr, and J. Leite, "Abstract dialectical frameworks are Boolean networks", in *LPNMR*, Springer, 2024, pp. 98–111.

[115] C. M. Homan and S. Kosub, "Dichotomy results for fixed point counting in Boolean dynamical systems", *Theor. Comput. Sci.*, vol. 573, pp. 16–25, 2015.

[116] M. Huber, "A Bernoulli mean estimate with known relative error distribution", in *Random Structures & Algorithms*, vol. 50, Wiley Online Library, 2017, pp. 173–182.

[117] M. Huber, "Tight relative estimation in the mean of Bernoulli random variables", *arXiv preprint arXiv:2210.12861*, 2022.

[118] J. E. Hurtado, "Structural reliability: statistical learning perspectives", Springer Science & Business Media, 2004, vol. 17.

[119] S. M. Ielpa, S. Iiritano, N. Leone, and F. Ricca, "An ASP-based system for e-tourism", in *LPNMR*, Springer, 2009, pp. 368–381.

[120] K. Inoue, "Logic programming for Boolean networks", in *IJCAI*, IJCAI/AAAI, 2011, pp. 924–930.

[121] K. Inoue and C. Sakama, "Oscillating behavior of logic programs", in *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, Springer, 2012, pp. 345–362.

[122] A. Ivrii, S. Malik, K. S. Meel, and M. Y. Vardi, "On computing minimal independent support and its applications to sampling and counting", *Constraints*, vol. 21, no. 1, pp. 41–58, 2016.

[123] S. Jabbour, L. Sais, and Y. Salhi, "Mining top-k motifs with a SAT-based framework", *Artificial Intelligence*, vol. 244, pp. 30–47, 2017.

[124] M. Jakl, R. Pichler, and S. Woltran, "Answer set programming with bounded treewidth." In *IJCAI*, vol. 9, 2009, pp. 816–822.

[125] T. Janhunen, "Representing normal programs with clauses", in *ECAI*, vol. 16, 2004, p. 358.

[126] T. Janhunen, "Some (in) translatability results for normal logic programs and propositional theories", *Journal of Applied Non-Classical Logics*, vol. 16, no. 1-2, pp. 35–86, 2006.

[127] T. Janhunen and I. Niemelä, "Compact translations of non-disjunctive answer set programs to propositional clauses", in 2011, pp. 111–130.

[128] T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J.-H. You, "Unfolding partiality and disjunctions in stable model semantics", *TOCL*, vol. 7, no. 1, pp. 1–37, 2006.

[129] M. Janota, R. Grigore, and J. Marques-Silva, "Counterexample guided abstraction refinement algorithm for propositional circumscription", in *European Workshop on Logics in Artificial Intelligence*, Springer, 2010, pp. 195–207.

[130] J. Ji, H. Wan, K. Wang, Z. Wang, C. Zhang, and J. Xu, "Eliminating disjunctions in answer set programming by restricted unfolding", in *IJCAI*, 2016, pp. 1130–1137.

[131] M. Kabir, "Minimal model counting via knowledge compilation", *arXiv preprint arXiv:2409.10170*, 2024.

[132] M. Kabir, S. Chakraborty, and K. S. Meel, "Exact ASP counting with compact encodings", in *AAAI*, vol. 38, 2024, pp. 10 571–10 580.

[133] M. Kabir, F. O. Everardo, A. K. Shukla, M. Hecher, J. K. Fichte, and K. S. Meel, "ApproxASP–a scalable approximate answer set counter", in *AAAI*, vol. 36, 2022, pp. 5755–5764.

[134] M. Kabir and K. S. Meel, "A fast and accurate ASP counting based network reliability estimator", in *LPAR*, vol. 94, 2023, pp. 270–287.

[135] M. Kabir and K. S. Meel, "On lower bounding minimal model count", *TPLP*, vol. 24, no. 4, pp. 586–605, 2024.

[136] M. Kabir and K. S. Meel, "A simple and effective ASP-based tool for enumerating minimal hitting sets", *arXiv preprint arXiv:2507.09194*, 2025.

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

[138] M. Kabir, V.-G. Trinh, S. Pastva, and K. S. Meel, "Scalable counting of minimal trap spaces and fixed points in boolean networks", *CP*, 2025.

[139] K. Kanchanasut and P. J. Stuckey, "Transforming normal logic programs to constraint logic programs",, vol. 105, no. 1, pp. 27–56, 1992, ISSN: 0304-3975.

[140] R. M. Karp, M. Luby, and N. Madras, "Monte-carlo approximation algorithms for enumeration problems", *J. Algorithms*, vol. 10, no. 3, pp. 429–448, 1989. [Online]. Available: https://doi.org/10.1016/0196-6774(89)90038-2.

[141] B. Kaufmann, N. Leone, S. Perri, and T. Schaub, "Grounding and solving in answer set programming", *AI magazine*, vol. 37, no. 3, pp. 25–32, 2016.

[142] L. M. Kirousis and P. G. Kolaitis, "The complexity of minimal satisfiability problems", *Information and Computation*, vol. 187, no. 1, pp. 20–39, 2003.

[143] H. Kitano, "Towards a theory of biological robustness", *Mol. Syst. Biol.*, vol. 3, no. 1, p. 137, 2007.

[144] H. Klarner, A. Bockmayr, and H. Siebert, "Computing maximal and minimal trap spaces of Boolean networks", *Nat. Comput.*, vol. 14, no. 4, pp. 535–544, 2015.

[145] H. Klarner, F. Heinitz, S. Nee, and H. Siebert, "Basins of attraction, commitment sets, and phenotypes of boolean networks", *IEEE/ACM TCBB*, vol. 17, no. 4, pp. 1115–1124, 2018.

[146] H. Klarner, E. Tonello, L. Fontanals, F. Janody, C. Chaouiya, and H. Siebert, "Detection of markers for discrete phenotypes", in *CSBio*, 2021, pp. 64–68.

[147] T. Korhonen and M. Järvisalo, "Integrating tree decompositions into decision heuristics of propositional model counters", in *CP*, 2021.

[148] E. van der Kouwe, D. Andriesse, H. Bos, C. Giuffrida, and G. Heiser, "Benchmarking crimes: An emerging threat in systems security", *arXiv preprint arXiv:1801.02381*, 2018.

[149] J.-M. Lagniez and P. Marquis, "An improved Decision-DNNF compiler." In *IJCAI*, vol. 17, 2017, pp. 667–673.

[150] J.-M. Lagniez and P. Marquis, "A recursive algorithm for projected model counting", in *AAAI*, vol. 33, 2019, pp. 1536–1543.

[151] J. Lee and V. Lifschitz, "Loop formulas for disjunctive logic programs", in *ICLP*, Springer, 2003, pp. 451–465.

[152] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The DLV system for knowledge representation and reasoning", *TOCL*, vol. 7, no. 3, pp. 499–562, 2006.

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

[154] Q. Li, A. Wennborg, E. Aurell, E. Dekel, J.-Z. Zou, Y. Xu, S. Huang, and I. Ernberg, "Dynamics inside the cancer cell attractor reveal cell heterogeneity, limits of stability, and escape", *Proceedings of the National Academy of Sciences*, vol. 113, no. 10, pp. 2672–2677, 2016.

[155] Z. Li, W. Yisong, X. Zhongtao, and F. Renyan, "Computing propositional minimal models: MiniSAT-based approaches", *Journal of Computer Research and Development*, vol. 58, no. 11, pp. 2515–2523, 2021, ISSN: 1000-1239.

[156] Y. Lierler, "Cmodels–SAT-based disjunctive answer set solver", in *LPNMR*, Springer, 2005, pp. 447–451.

[157] Y. Lierler, "What is answer set programming to propositional satisfiability", *Constraints*, vol. 22, pp. 307–337, 2017.

[158] M. H. Liffiton and K. A. Sakallah, "Algorithms for computing minimal unsatisfiable subsets of constraints", *Journal of Automated Reasoning*, vol. 40, pp. 1–33, 2008.

[159] V. Lifschitz, "Thirteen definitions of a stable model", *Fields of logic and computation*, pp. 488–503, 2010.

[160] V. Lifschitz, "Answer sets and the language of answer set programming", *AI Magazine*, vol. 37, no. 3, pp. 7–12, 2016.

[161] V. Lifschitz and A. Razborov, "Why are there so many loop formulas?" *TOCL*, vol. 7, no. 2, pp. 261–268, 2006.

[162] F. Lin and Y. Zhao, "ASSAT: Computing answer sets of a logic program by SAT solvers", *Artificial Intelligence*, vol. 157, no. 1-2, pp. 115–137, 2004.

[163] T. Linsbichler, M. Maratea, A. Niskanen, J. P. Wallner, and S. Woltran, "Advanced algorithms for abstract dialectical frameworks based on complexity analysis of subclasses and SAT solving", *Artif. Intell.*, vol. 307, p. 103 697, 2022.

[164] G. Liu, T. Janhunen, and I. Niemelä, "Answer set programming via mixed integer programming." *KR*, vol. 12, pp. 32–42, 2012.

[165] V. W. Marek and M. Truszczyński, "Stable models and an alternative logic programming paradigm", in *The Logic Programming Paradigm*, Springer, 1999, pp. 375–398.

[166] W. Marek and M. Truszczyński, "Autoepistemic logic", *Journal of the ACM (JACM)*, vol. 38, no. 3, pp. 587–618, 1991.

[167] J. Marques-Silva, I. Lynce, and S. Malik, "Conflict-driven clause learning sat solvers", in *Handbook of satisfiability*, ios Press, 2021, pp. 133–182.

[168] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability", *IEEE Transactions on computers*, vol. 48, no. 5, pp. 506–521, 2002.

[169] G. Masina, G. Spallitta, and R. Sebastiani, "On CNF conversion for disjoint SAT enumeration", in *SAT*, 2023, 15:1–15:16.

[170] A. Montagud, J. Béal, L. Tobalina, P. Traynard, V. Subramanian, B. Szalai, R. Alföldi, L. Puskás, A. Valencia, and E. Barillot, "Patient-specific Boolean models of signalling networks guide personalised treatments", *Elife*, vol. 11, e72626, 2022.

[171] R. Motwani and P. Raghavan, "Randomized algorithms", *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 33–37, 1996.

[172] I. Niemelä and P. Simons, "Extending the Smodels system with cardinality and weight constraints", *Logic-based artificial intelligence*, pp. 491–521, 2000.

[173] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry, "An A-Prolog decision support system for the space shuttle", in *PADL*, Springer, 2001, pp. 169–183.

[174] M. Ostrowski and T. Schaub, "ASP modulo CSP: The clingcon system", *TPLP*, vol. 12, no. 4-5, pp. 485–503, 2012.

[175] A. Padoa, "Essai d'une théorie algébrique des nombres entiers, précédé d'une introduction logique à une théorie déductive quelconque", *Bibliothèque du Congrès International de Philosophie*, vol. 3, pp. 309–365, 1901.

[176] R. Paredes, L. Duenas-Osorio, and I. Hernandez-Fajardo, "Decomposition algorithms for system reliability estimation with applications to interdependent lifeline networks",, 13, vol. 47, Wiley Online Library, 2018, pp. 2581–2600.

[177] S. Pastva, D. Šafránek, N. Beneš, L. Brim, and T. Henzinger, "Repository of logically consistent real-world Boolean network models", *bioRxiv*, 2023. [Online]. Available: https://www.biorxiv.org/content/early/2023/06/12/2023.06.12.544361.

[178] L. Paulevé, J. Kolčák, T. Chatain, and S. Haar, "Reconciling qualitative, abstract, and scalable modeling of biological networks", *Nat. Commun.*, vol. 11, no. 1, pp. 1–7, Aug. 2020.

[179] E. Pontelli, T. C. Son, and O. Elkhatib, "Justifications for logic programs under answer set semantics", *TPLP*, vol. 9, no. 1, pp. 1–56, 2009.

[180] J. S. Provan and M. O. Ball, "Computing network reliability in time polynomial in the number of cuts", *Operations Research*, vol. 32, no. 3, pp. 516–526, 1984.

[181] S. Rai and A. Kumar, "Recursive technique for computing system reliability", in *IEEE transactions on reliability*, vol. 36, IEEE, 1987, pp. 38–44.

[182] D. Roth, "On the hardness of approximate reasoning", *Artificial Intelligence*, vol. 82, no. 1-2, pp. 273–302, 1996.

[183] Y. Salhi, "On enumerating all the minimal models for particular CNF formula classes." In *ICAART (2)*, 2019, pp. 403–410.

[184] M. Samer and S. Szeider, "Algorithms for propositional model counting", in *LPAR*, Springer, 2007, pp. 484–498.

[185] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi, "Combining component caching and clause learning for effective model counting." *SAT*, vol. 4, p. 7, 2004.

[186] T. Sang, P. Beame, and H. A. Kautz, "Performing bayesian inference by weighted model counting", in *AAAI*, vol. 5, 2005, pp. 475–481.

[187] J. D. Schwab, S. D. Kühlwein, N. Ikonomi, M. Kühl, and H. A. Kestler, "Concepts in Boolean network modeling: What do they all mean?" *Comput. Struct. Biotechnol. J.*, vol. 18, pp. 571–582, 2020.

[188] S. Sharma, S. Roy, M. Soos, and K. S. Meel, "GANAK: A scalable probabilistic exact model counter." In *IJCAI*, vol. 19, 2019, pp. 1169–1176.

[189] I. Shmulevich, E. R. Dougherty, and W. Zhang, "From Boolean to probabilistic Boolean networks as models of genetic regulatory networks", *Proc. IEEE*, vol. 90, no. 11, pp. 1778–1792, 2002.

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

[191] M. Soos, S. Gocht, and K. S. Meel, "Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling", in *CAV*, Springer, 2020, pp. 463–484.

[192] M. Soos and K. S. Meel, "Arjun: An efficient independent support computation technique and its applications to counting and sampling", in *ICCAD*, 2022, pp. 1–9.

[193] M. Soos and K. S. Meel, "Engineering an efficient probabilistic exact model counter", in *CAV*, Springer, 2025, pp. 72–91.

[194] M. Soos, K. Nohl, and C. Castelluccia, "Extending sat solvers to cryptographic problems", in *SAT*, Springer, 2009, pp. 244–257.

[195] L. Stockmeyer, "The complexity of approximate counting", in *STOC*, 1983, pp. 118–126.

[196] C. Su and J. Pang, "Sequential temporary and permanent control of Boolean networks", in *CMSB*, Springer, 2020, pp. 234–251.

[197] R. Suzuki, K. Hashimoto, and M. Sakai, "Improvement of projected model-counting solver with component decomposition using SAT solving in components", JSAI Technical Report, Tech. Rep., 2017.

[198] M. Thurley, "sharpSAT–counting models with advanced component caching and implicit BCP", in *SAT*, Springer, 2006, pp. 424–429.

[199] J. Tiihonen, T. Soininen, I. Niemelä, and R. Sulonen, "A practical tool for mass-customising configurable products", in *ICED*, 2003.

[200] E. Tonello and L. Paulevé, "Phenotype control and elimination of variables in Boolean networks", *Peer Community Journal*, vol. 4, Aug. 2024, ISSN: 2804-3871.

[201] G. Trinh, B. Benhamou, S. Pastva, and S. Soliman, "Scalable enumeration of trap spaces in boolean networks via answer set programming", in *AAAI*, vol. 38, 2024, pp. 10 714–10 722.

[202] V. Trinh, B. Benhamou, and L. Paulevé, "mpbn: A simple tool for efficient edition and analysis of elementary properties of Boolean networks", *CoRR*, vol. abs/2403.06255, 2024. arXiv: 2403.06255.

[203] V. Trinh, B. Benhamou, and S. Soliman, "Efficient enumeration of fixed points in complex Boolean networks using answer set programming", in *CP*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 35:1–35:19.

[204] V.-G. Trinh, B. Benhamou, and S. Soliman, "Trap spaces of Boolean networks are conflict-free siphons of their Petri net encoding", *Theor. Comput. Sci.*, vol. 971, p. 114 073, Sep. 2023.

[205] V. Trinh, B. Benhamou, S. Soliman, and F. Fages, "Graphical conditions for the existence, unicity and number of regular models", in *ICLP*, 2024, pp. 175–187.

[206] V. Trinh, K. Hiraishi, and B. Benhamou, "Computing attractors of large-scale asynchronous Boolean networks using minimal trap spaces", in *ACM-BCB*, ACM, 2022, 13:1–13:10.

[207] G. S. Tseitin, "On the complexity of derivation in propositional calculus", *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pp. 466–483, 1983.

[208] R. Vaisman, D. P. Kroese, and I. B. Gertsbakh, "Splitting sequential Monte Carlo for efficient unreliability estimation of highly reliable networks", in *Structural Safety*, vol. 63, Elsevier, 2016, pp. 1–10.

[209] L. G. Valiant, "The complexity of enumeration and reliability problems", *SIAM Journal on Computing*, vol. 8, no. 3, pp. 410–421, 1979.

[210] A. Van Gelder, K. Ross, and J. S. Schlipf, "Unfounded sets and well-founded semantics for general logic programs", in *PODS*, 1988, pp. 221–230.

[211] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook", in *Proceedings of the 2nd ACM workshop on Online social networks*, 2009, pp. 37–42.

[212] D. S. Warren, "Introduction to prolog", in *Prolog: The Next 50 Years*, Springer, 2023, pp. 3–19.

[213] A. Weinzierl, "Blending lazy-grounding and CDNL search for Answer Set solving", in *LPNMR*, Springer, 2017, pp. 191–204.

[214] B. Wiegmans, "Gridkit: European and north-american extracts",, vol. 47317, 2016.

[215] J. Yang and K. S. Meel, "Rounding meets approximate model counting", in *CAV*, Springer, 2023, pp. 132–162.

[216] C. Yin and A. Kareem, "Computation of failure probability via hierarchical clustering", *Structural Safety*, vol. 61, pp. 67–77, 2016.

[217] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient conflict driven learning in a boolean satisfiability solver", in *ICCAD*, IEEE, 2001, pp. 279–285.

[218] Y. Zhou, "From disjunctive to normal logic programs via unfolding and shifting", in *ECAI 2014*, IOS Press, 2014, pp. 1139–1140.

[219] K. M. Zuev, S. Wu, and J. L. Beck, "General network reliability problem and its efficient solution by subset simulation", *Probabilistic Engineering Mechanics*, vol. 40, pp. 25–35, 2015.

Appendix A

Preliminaries: Appendix

Proof of 2.1

Proof. proof of "if" part: Proof idea: Proof by Contradication.
Assume that $\sigma \in \mathsf{AS}(\mathcal{DLP}(F))$ but $\sigma \notin \mathsf{MinModels}(F)$. As $\sigma \in \mathsf{AS}(\mathcal{DLP}(F))$, then $\sigma \models F$. Thus we only proof that there is no model $\sigma' \subset \sigma$ such that $\sigma' \models F$. For purpose of the contradiction, assume that there is a model $\sigma' \subset \sigma$ and $\sigma' \models F$. By consturction of program $\mathcal{DLP}(F)$, as $\sigma' \models F$, $\sigma' \models \mathcal{DLP}(F)$. Note that the reduct of $\mathcal{DLP}(F)$ w.r.t. σ and σ' are same, more specifically, $\mathcal{DLP}(F)^{\sigma} = \mathcal{DLP}(F)^{\sigma'} = \mathcal{DLP}(F)$ because there is no default negation in $\mathcal{DLP}(F)$. Thus, $\sigma' \subset \sigma$ and $\sigma' \models \mathcal{DLP}(F)^{\sigma}$, which contradicts that σ is an answer set of $\mathcal{DLP}(F)$.

proof of "only if" part: The proof is trivial. If $\sigma \in \mathsf{MinModels}(F)$ and $\sigma \notin \mathsf{AS}(\mathcal{DLP}(F))$, then there is another $\sigma' \subset \sigma$ and $\sigma' \models \mathcal{DLP}(F)$, which contradicts that σ is a minimal model because $\sigma' \models F$. □

Proof of Lemma 2.2

Proof. 'if' part proof: Let $I = \{a'_1, \ldots, a'_k\} \subseteq \mathcal{I}$ be a minimal generator of D and $\mathcal{C}(I, D) = \{t'_1, \ldots, t'_m\}$. We proof that $\sigma = \{p_{a'_1}, \ldots, p_{a'_k}, q_{t'_1}, \ldots, q_{t'_m}\}$ is a minimal model of $\mathsf{MG}(D)$. By definition of $\mathsf{MG}(D)$, σ is a model of $\mathsf{MG}(D)$. Now we show that σ is a minimal model of $\mathsf{MG}(D)$. We proof it by contradiction and assume that there is model $\sigma' \models \mathsf{MG}(D)$ and σ' is strictly smaller than σ. As I is a minimal generator, there is no $I' \subset I$ such that $\mathcal{C}(I, D) = \mathcal{C}(I', D)$. Thus, there is at least one $t'_u \in \sigma \setminus \sigma'$, i.e., $t'_u \in \mathcal{C}(\sigma, D)$ and $t'_u \notin \mathcal{C}(\sigma', D)$. By definition of $\mathsf{MG}(D)$, t'_u occurs exactly in one clause of $\mathsf{MG}(G)$ and let us denote the clause by notation $C_{t'_u}$. The

literal t'_u is justified in minimal model M, thus $\forall a \in (\mathcal{I} \setminus I_{t'_u}), p_a \notin M$, which follows that the clause $C_{t'_u}$ is not satisified by σ', which contradicts that $\sigma' \models \mathsf{MG}(D)$.

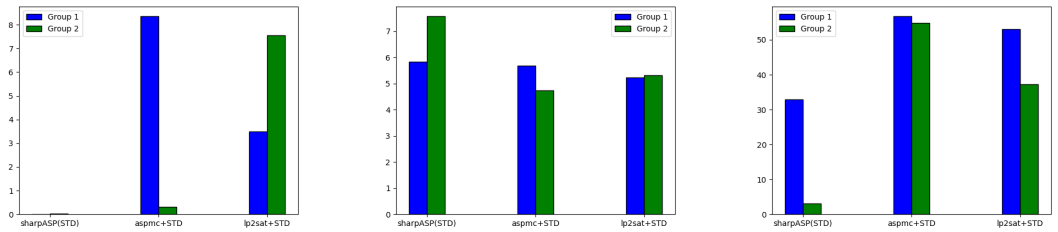
'only if' part proof: Let σ be a minimal model of $\mathsf{MG}(D)$ and assume that $I_\sigma = \{a | p_a \in \sigma\}$ is not a minimal generator of D i.e., there is another $I'_\sigma \subset I_\sigma$ such that $\mathcal{C}(I_\sigma, D) = \mathcal{C}(I'_\sigma, D)$. By construction of $\mathsf{MG}(D)$, $\mathcal{C}(I_\sigma, D) \subseteq \{q_t | q_t \in \sigma\}$. As $\sigma \in \mathsf{MinModels}(\mathsf{MG}(D))$, $\forall q_t \in \sigma$ has a justification. Note that the (positive) literal q_t occurs in exactly one clause $\mathsf{MG}(D)$, which implies that $\forall a \in (\mathcal{I} \setminus I_t), p_a \notin \sigma$. That turns out $\mathcal{C}(I_\sigma, D) = \{q_t | q_t \in \sigma\}$, which follows that $\mathcal{C}(I'_\sigma, D) = \mathcal{C}(I_\sigma, D) = \{q_t | q_t \in \sigma\}$. However, if $I'_\sigma \subset I_\sigma$, then there are two possible cases: (i) either $\sigma' = \{p_a | a \in I'_\sigma\} \cup \{q_t | q_t \in \sigma\} \models \mathsf{MG}(D)$, that contradicts that σ is a minimal model of $\mathsf{MG}(D)$, (ii) or $\sigma' = \{p_a | a \in I'_\sigma\} \cup \{q_t | q_t \in \sigma\} \not\models \mathsf{MG}(D)$ that contradicts that $\mathcal{C}(I_\sigma, D) = \mathcal{C}(I'_\sigma, D)$. □

Appendix B

Further Analysis: sharpASP

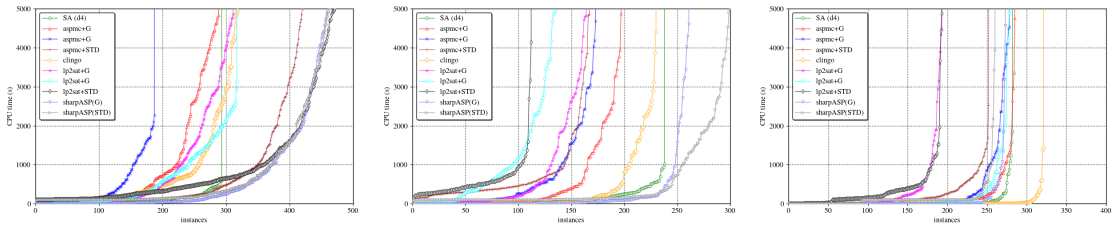
We now delve into the internals and form two groups of benchmarks:

- **Group** 1 instances where sharpASP(STD) runs faster than lp2sat+STD and aspmc+STD, which highlights the scenarios where the sharpASP(STD) algorithm is more efficient than lp2sat+STD and aspmc+STD
- **Group** 2 instances where lp2sat+STD and aspmc+STD run faster than sharpASP(STD), which shows the opposite scenario of Group 1.



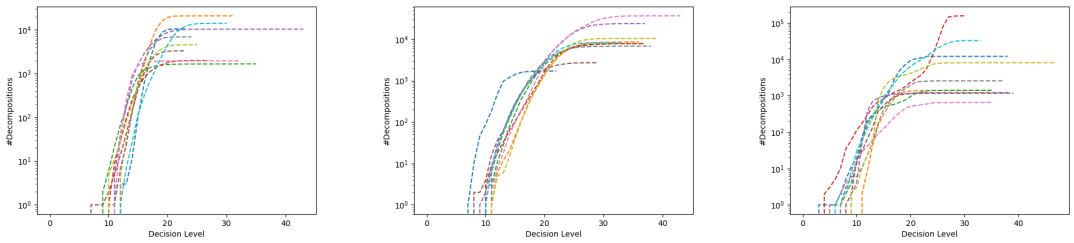
(a) BCP time (seconds) (b) #decisions (10-base log). (c) Cache hit (percentage).

Figure B.1: The ablation study of sharpASP(STD), lp2sat+STD, and aspmc+STD on Group 1 and Group 2 benchmarks.



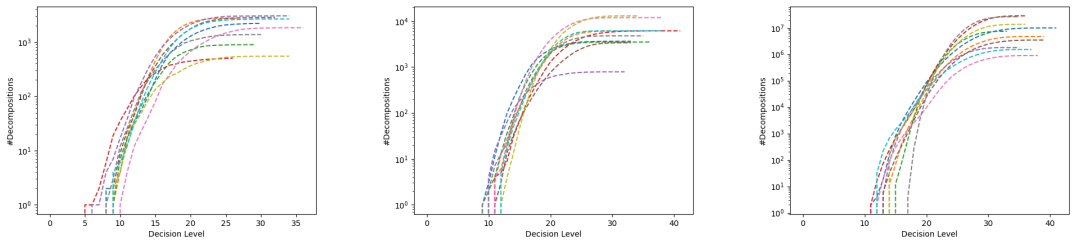
(a) Graph Reachability (b) Hamiltonian cycle (c) aspben

Figure B.2: The runtime performance of different counters on different computational problems.



(a) aspmc+STD (b) lp2sat+STD (c) sharpASP(STD)

Figure B.3: The number of decompositions upto certain decision levels for different variants of SharpSAT-TD (STD) on Group 1



(a) aspmc+STD (b) lp2sat+STD (c) sharpASP(STD)

Figure B.4: The number of decompositions upto certain decision levels for different variants of SharpSAT-TD (STD) on Group 2

Each group consists of 10 instances that had more than 10^5 answer sets, and therefore clingo could not enumerate all answer sets. By running the instances on all versions of SharpSAT-TD, we record the time spent on the procedure *binary constraint propagation* (BCP), number of decisions, and *cache hit rate* for each counter. Taking each group's average of each quantity provides a clear and concise way to see how sharpASP compares with others on average across all benchmarks. The statistical findings across all counters are visually summarized in Figure B.1.

The strength of sharpASP lies in its ability to minimize the time spent on binary constraint propagation (BCP) compared to other counters. The significantly large formula size increases the overhead for BCP in the case of lp2sat+STD and aspmc+STD. However, we also observe that sharpASP suffers from high overhead in the branching phase and high *cache misses* on Group 2 instances. To find out the reason for a higher number of decisions, we analyze the decomposibility of Group 1 and Group 2 instances.

Our investigation has shown that, on all variants of SharpSAT-TD, most in-

stances of Group 1 start decomposing at nearly the same decision levels. Thus, sharpASP(STD) outperforms on Group 1 instances due to spending less time on BCP. We observed that several instances of Group 1 took comparatively more decisions to make to count the number of answer sets on sharpASP(STD). One possible explanation is that aspmc+STD and lp2sat+STD assign auxiliary variables, which have higher *activity scores* compared to original ASP program variables. Assigning auxiliary variables facilitates lp2sat+STD and aspmc+STD by assigning fewer variables. However, sharpASP(STD) outperforms others due to structural simplicity and low-cost BCP.

Our investigation has also revealed that Group 2 instances are hard-to-decompose on sharpASP(STD) compared to other counters – necessitating more variable assignments to break down an instance into disjoint components. Since sharpASP(STD) assigns the original set of variables; it necessitates a larger number of decisions to count answer sets on hard-to-decompose instances compared to aspmc and lp2sat based counters. Moreover, the structure of hard-to-decompose instances also worsens the cache performance of sharpASP. However, lp2sat+STD and aspmc+STD effectively decompose the input formula by initially assigning auxiliary variables.

To explain why sharpASP can count Group 1 instances efficiently but not the Group 2 instances, we present a comparative study between sharpASP(STD), lp2sat+STD, aspmc+STD. For the comparative study, we visually present the number of decomposition nodes at different decision levels. Ideally, formula decomposition avoids model counters from enumerating solutions and lesser decomposability implies more number of decisions to make to count all solutions.

Figure B.3 shows the number of decompositions upto certain decision levels for counters lp2sat+STD, aspmc+STD, and sharpASP(STD) on Group 1 instances. The graphs are non-decreasing because the number of decompositions upto level $\ell + 1$ is the sum of the number of decompositions upto level ℓ and number of decomposition at level $\ell + 1$. These figures show that most of the instances of Group 1 start decomposing almost at the same decision levels on all variants of SharpSAT-TD. Thus, sharpASP(STD) outperforms on Group 1 instances due to spending lesser time on BCP. We observed that several instances of Group 1 took comparatively more decisions to make to count the number of answer sets on sharpASP(STD). The possible explanation is that aspmc+STD and lp2sat+STD assign auxiliary

variables, which has more *activity score* compared to original ASP program variables. Assigning auxiliary variables facilitates lp2sat+STD and aspmc+STD by assigning a lesser number of variables. However, sharpASP(STD) outperforms others due to structural simplicity and low-cost BCP.

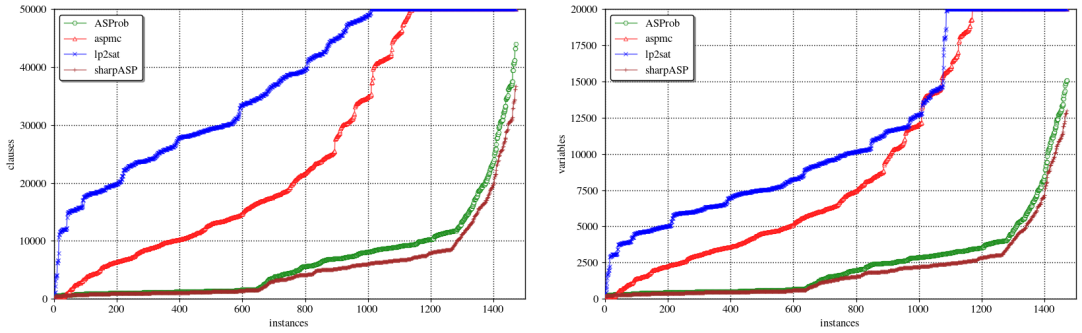
Figure B.4 shows the number of decompositions upto certain decision levels for counters lp2sat+STD, aspmc+STD, and sharpASP(STD) on Group 2 instances. These plots show that Group 2 instances are hard-to-decompose on sharpASP(STD) compared to other counters — requiring more variable assignments to make to decompose an instance into disjoint components. As sharpASP(STD) assigns the original set of variables, sharpASP(STD) makes a large number of decisions to count answer sets on hard-to-decompose instances compared to aspmc and lp2sat based counters. Moreover, the structure of hard-to-decompose instances also worsens the cache performance of sharpASP. However, lp2sat+STD and aspmc+STD effectively decompose the input formula by initially assigning auxiliary variables.

To conclude, we can say that the performance of sharpASP depends on the structural hardness and variable branching heuristics employed.

In light of these findings, it is evident that the performance of sharpASP is critically reliant on the decomposability of input instances and the variable branching heuristic employed. Notably, sharpASP demonstrates superior performance when applied to *structurally simpler* input instances. If a variable branching heuristic effectively decomposes the input formula by assigning variables within the ASP programs, sharpASP outperforms alternative ASP counters. Conversely, when the input formula's decomposability is hindered, alternative approaches involving the introduction of auxiliary variables prove to be more advantageous.

The runtime performance of each counter under different computational problems is shown as cactus plots in Figure B.2. A point (x, y) denotes a counter counts a total of x instances within a runtime of y seconds.

Compactness of Encodings. We analyze the size of the CNF formula on which the underlying counting algorithms operate since the size of the input formula can have a considerable impact on the performance of counting algorithms for problems of similar nature. Specifically, for sharpASP and ASProb, we measure the size of the formula as the conjunction of completion and copy operation of an



(a) the number of clauses (b) the number of variables

Figure B.5: Visualization of the size of input formulas of different ASP counters.

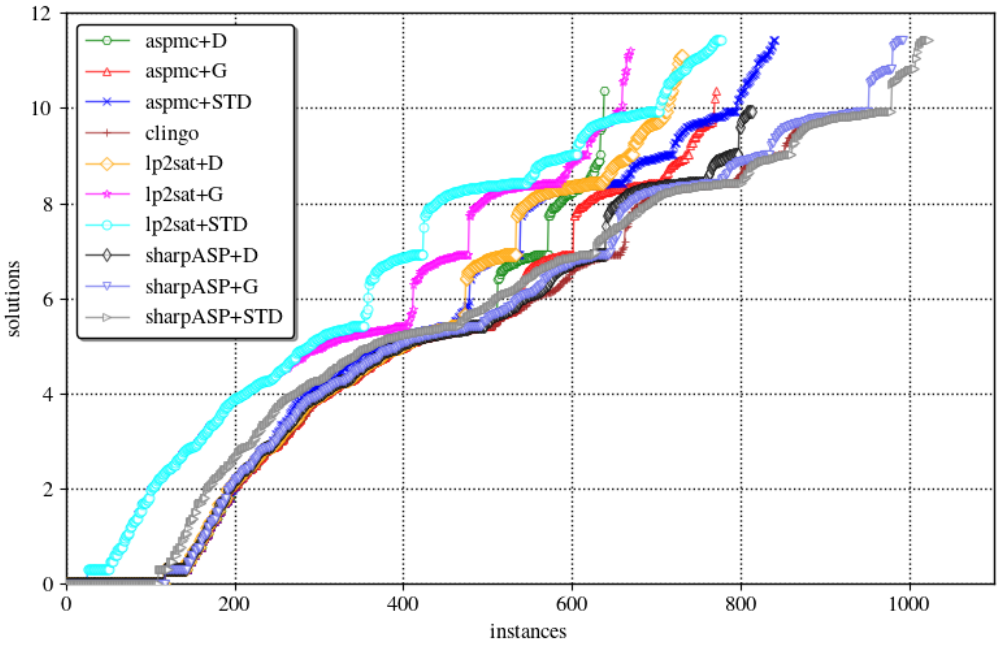


Figure B.6: The number of answer sets of instances solved by different ASP counters.

ASP program, whereas for aspmc and lp2sat, we measure the size of the formula obtained after lp2sat and aspmc translation, respectively. Figures B.5a and B.5b represent the comparison of CNF formulas in terms of the number of clauses and variables, respectively. Our analysis shows that the CNF formula for sharpASP is significantly smaller than that of aspmc and lp2sat, and the encoding of sharpASP is more compact than ASProb, both in terms of the number of variables and clauses.

Number of Answer Sets. We investigate whether sharpASP counts answer sets via enumerations. The investigation involves a comparative analysis of the number

of answer sets for each instance solved (i.e., returns the count) by sharpASP and existing answer set counters. A cactus plot shows the findings of this analysis in Figure B.6. In the plot, a point (x, y) denotes that a counter solves total x instances, of which each of the instances has at most 10^y answer sets. The plot demonstrates that sharpASP can solve instances that cannot be counted via enumeration-based techniques (e.g., Clingo) due to the large number of answer sets and illustrates the competitiveness of sharpASP compared to other ASP counters in terms of $|\mathsf{AS}(P)|$.

Appendix C

Further Analysis: sharpASP-$\mathcal{SR}$

Performance comparison across different computation problems. We present the Table C.1 showing the number of instances across different benchmark classes solved by different ASP counters. The second ($\sum$) and third columns ($\sum^{\geq 1000}$) represents the total number of instances and total number of instances having more that 1000 loop atoms in each benchmark class, respectively. We observe that there are two benchmark classes: preferred extension and diagnosis, where the performance of sharpASP-$\mathcal{SR}$ is surpassed by Clingo. Our observations reveal that these instances tend to have a significantly larger number of loop atoms. More specifically, around 66% of Preferred extension instances and 100% of Diagnosis instances contain ≥ 1000 loop atoms.

	$\sum$	$\sum^{\geq 1000}$	Clingo	DynASP	Wasp	sharpASP-$\mathcal{SR}$
2QBF	200	0	179	0	58	**181**
Strategic	226	0	53	0	0	**125**
Preferred	217	142	**208**	2	192	110
PC config	1	1	0	0	0	0
Diagnosis	11	11	**11**	0	8	6
Random	226	0	80	0	0	**213**
MTS	244	53	177	87	174	**190**
			708	89	432	825

Table C.1: The performance comparison of different ASP counters across different benchmark classes. The abbreviation MTS refers to minimal trap space benchmark, respectively.

	D4	GPMC	Ganak
#Solved	697	759	825
PAR2	3856	3463	2939

Table C.2: The performance comparison of alternative #∃SAT counting techniques.

Experiments with Alternative Projected Model Counters We conducted experiments with alternative projected model counters, including D4 [149] and GPMC [197]. The purpose of this experiment is to evaluate the performance of sharpASP-$\mathcal{SR}$ in combination with other projected model counters. The results of these experiments, comparing the performance of the alternative counting techniques, are summarized in Table C.2. The results reveal that sharpASP-$\mathcal{SR}$ with GANAK outperforms sharpASP-$\mathcal{SR}$ with alternative projected model counters.

Appendix D

Further Analysis: ApproxASP

We evaluate the ASP+XOR solver of ApproxASP with *xorro* [67]. In the evaluation, to generate ASP+XOR programs, we add as much XORs to the ASP program so that the number of answer sets in a randomly chosen cell is at most pivot p. The results on ASP+XOR programs are shown in Figure D.1. If a point is below the diagonal, then ApproxASP (XOR solver) solves it faster. The timeouted instances are shown beyond the 1000s axis. From the figure, it is clear that the XOR solver of ApproxASP is significantly faster than *xorro*.

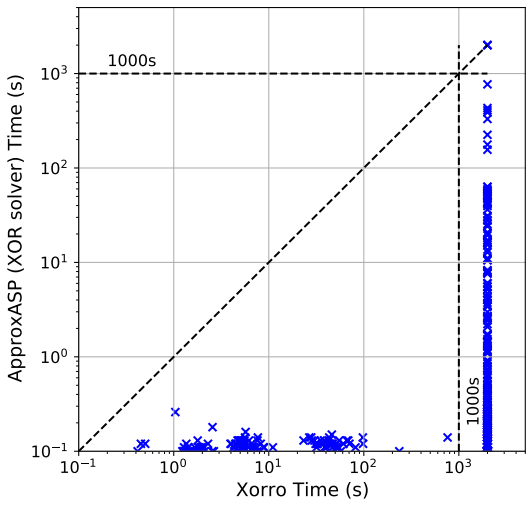


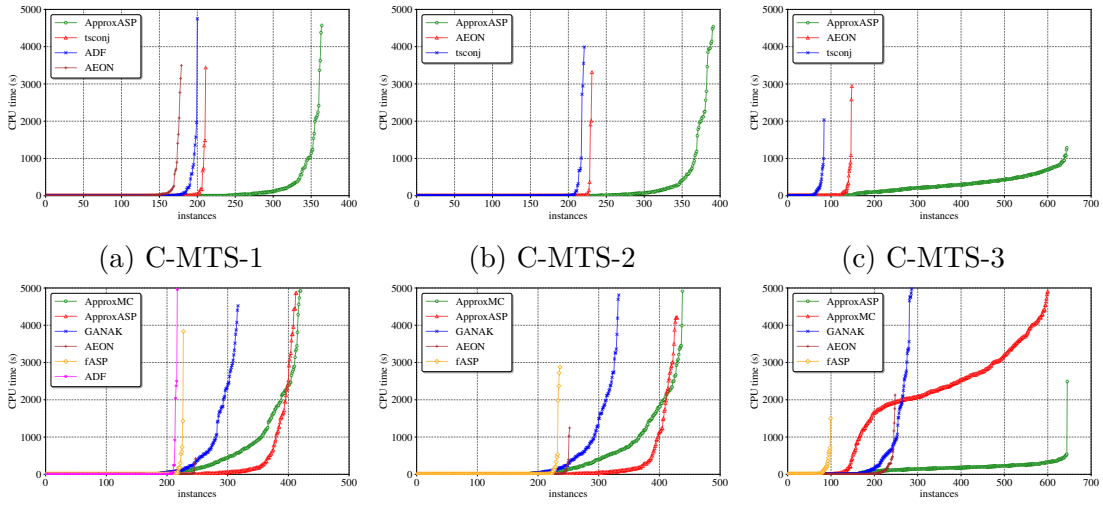
Figure D.1: *xorro* vs ApproxASP (XOR solver) on ASP+XOR programs.

Appendix E

Further Analysis: Counting BNs

The runtime performance of different tools is depicted in Figure E.1. We also compare their counts for each solved instance in Figure E.2. In these cactus plots, a point (x, y) indicates that a tool successfully completes x benchmark instances, with each instance taking at most y seconds. The plots highlight the superiority of the hashing-based counting techniques, ApproxASP and ApproxMC. Notably, even in cases where ApproxASP solves fewer instances than ApproxMC (e.g. C-FIX-1), it is typically faster on simpler instances, which is also reflected in its PAR2 score.

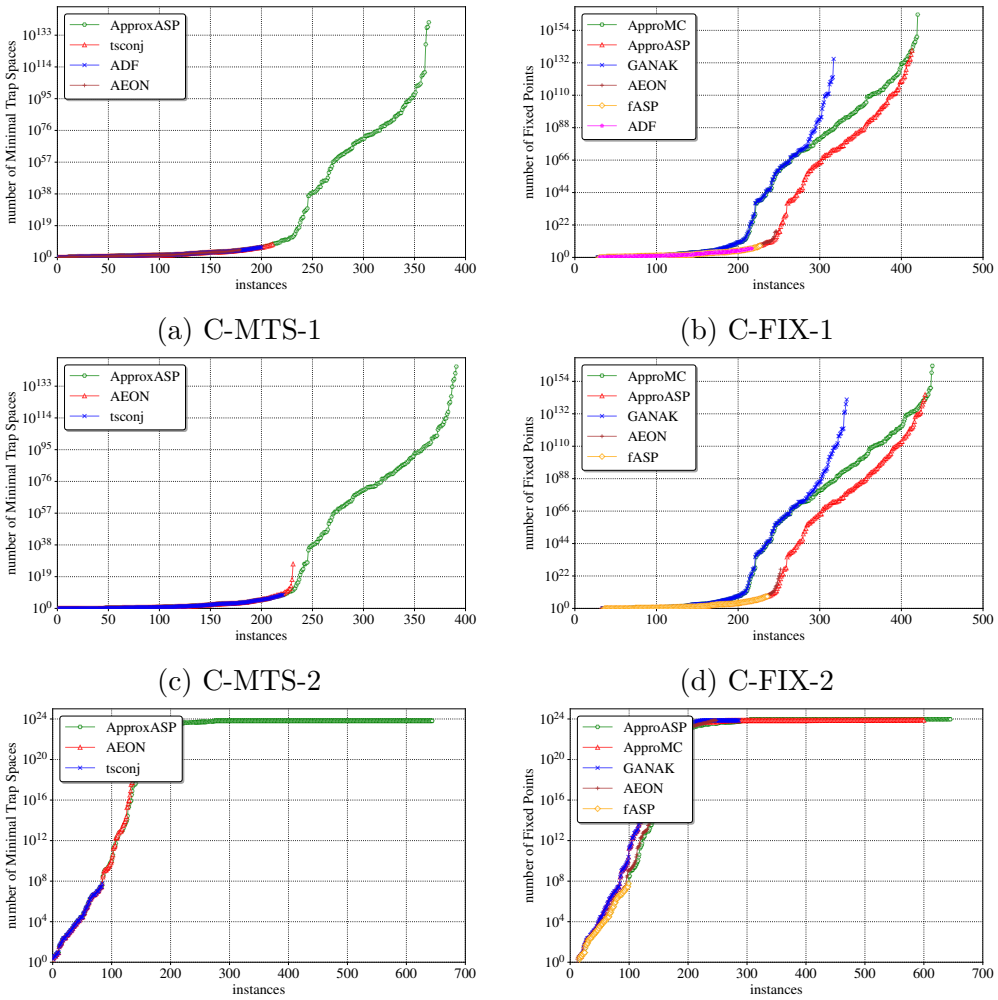
By examining the number of solutions successfully computed for different tasks, we observe that only ApproxASP, ApproxMC, and GANAK can reliably count instances having a large number of solutions (e.g. $\geq 10^{30}$). Here, BDD-based counters like AEON perform somewhat better on fixed point problems compared



(a) C-MTS-1 (b) C-MTS-2 (c) C-MTS-3

(d) C-FIX-1 (e) C-FIX-2 (f) C-FIX-3

Figure E.1: Performance comparison of different counters across all counting problems.



(a) C-MTS-1 (b) C-FIX-1

(c) C-MTS-2 (d) C-FIX-2

(e) C-MTS-3 (f) C-FIX-3

Figure E.2: The comparison of the number of solutions (minimal trap spaces or fixed points) for instances solved by different counters.

to tools using plain enumeration (ADF and clingo), but cannot compete in the (arguably more complex) trap space problems.

The performance of GANAK and ApproxMC is also severely affected by the time required to compute their input CNF formulas (ref. Section 7.1). Here, deriving the CNF problem representation is often considerably more time-consuming than computing the comparable ASP encoding. Our results reveal that, on average, it took about 545 seconds to compute the CNF formula for each BN. Moreover, for 39 BNs, the corresponding CNF could not be computed within the 5000-second timeout. In contrast, the ASP encodings for all BNs—including the more challenging ones with perturbable variables—were generated within seconds. This demonstrates the

superior flexibility of the ASP-based approach.

Note that the approximate counters ApproxMC and ApproxASP provide an (ε, δ)-guarantee. Thus for each solved instance, we computed the *observed tolerance*, which is defined as $\max(\mathsf{Count}/|\mathsf{AS}(P)|, |\mathsf{AS}(P)|/\mathsf{Count}) - 1$, where Count is the count returned by ApproxASP or ApproxMC, and $|\mathsf{AS}(P)|$ denotes the answer set count of program P. On average, ApproxMC and ApproxASP exhibit observed tolerances of 0.032 and 0.007, respectively. The maximum observed tolerances were 0.39 for ApproxASP and 0.07 for ApproxMC, both of which are well below the theoretical bound of $\varepsilon = 0.8$.

We evaluated ApproxMC and ApproxASP with a tighter guarantee, setting $\varepsilon = 0.01$ and $\delta = 0.05$. In these setting, ApproxASP solved $244, 243, 114, 258, 259, 130$ for C-MTS-1, C-MTS-2, C-MTS-3, C-FIX-1, C-FIX-2, and C-FIX-3, respectively, while ApproxMC solved $263, 259, 120$ for C-FIX-1, C-FIX-2, and C-FIX-3, respectively. Thus, the higher precision significantly reduces the number of solved instances.

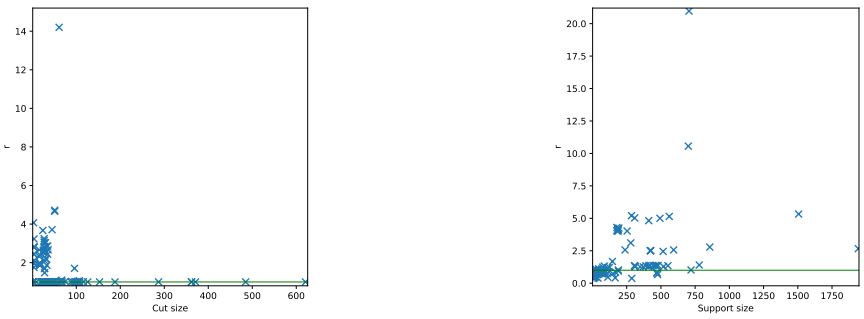
We compared the size of instances solved by different counting techniques across various counting problems. We observed that, for minimal trap spaces, ApproxASP solved instances with up to 4000 variables, while `tsconj` and AEON solved instances with up to 321 variables. For fixed points, although there were large instances (up to 4000 variables), these have no fixed points, making a direct comparison unfeasible.

Appendix F

Further Analysis: MinLB

The performance of Proj-Enum and HashCount contingent upon the size of the cut and independent support, respectively. In this analysis, we explore the strengths and weaknesses of Proj-Enum and HashCount by measuring their relative quality, as defined in Equation 8.1, across various sizes of cuts and independent supports, respectively. This comparative analysis is visually represented in Figure F.1, where clingo serves as the reference.

In the graphical representations, each point (x, y) corresponds to an instance where for the size of cut (independent support resp.) is x and the prototype Proj-Enum (HashCount resp.) achieves a relative quality of y. In the plots, the horizontal line is across $r = 1$ A relative quality exceeding 1 indicates that the lower bound returned by Proj-Enum or HashCount surpasses that of clingo. These plots reveal that Proj-Enum tends to perform well with smaller cut sizes, while HashCount demonstrates better performance across a range from small to medium sizes of independent support.



(a) The relative quality of Proj-Enum with varying cut size.

(b) The relative quality of HashCount with varying independent support size.

Figure F.1: The relative quality of Proj-Enum and HashCount vis-a-vis different cut and independent support size, where clingo is used as the reference baseline.