Non-incremental knowledge compilation of probabilistic answer set programs

Jonas Rodrigues Lima Gonçalves

Thesis presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
in partial fulfillment
of the requirements
for the degree of
Master of Science

Program: Computer Science
Advisor: Prof. Dr. Denis Deratani Mauá

São Paulo
October, 2025

Non-incremental knowledge compilation of probabilistic answer set programs

Jonas Rodrigues Lima Gonçalves

This is the original version of the thesis prepared by candidate Jonas Rodrigues Lima Gonçalves, as submitted to the Examining Committee.

Ficha catalográfica elaborada com dados inseridos pelo(a) autor(a)
Biblioteca Carlos Benjamin de Lyra
Instituto de Matemática e Estatística
Universidade de São Paulo

Lima Gonçalves, Jonas Rodrigues
Non-Incremental Knowledge Compilation of Probabilistic Answer Set Programs / Jonas Rodrigues Lima Gonçalves; orientador, Denis Maua. - São Paulo, 2025.
98 p.: il.

Dissertação (Mestrado) - Programa de Pós-Graduação em Ciência da Computação / Instituto de Matemática e Estatística / Universidade de São Paulo.
Bibliografia
Versão original

1. Inteligência Artificial. 2. Representação de Conhecimento. 3. Programação Lógico Probabilística. 4. Compilação de Conhecimento. 5. Circuitos Probabilísticos. I. Maua, Denis. II. Título.

Bibliotecárias do Serviço de Informação e Biblioteca
Carlos Benjamin de Lyra do IME-USP, responsáveis pela
estrutura de catalogação da publicação de acordo com a AACR2:
Maria Lúcia Ribeiro CRB-8/2766; Stela do Nascimento Madruga CRB 8/7534.

Acknowledgements

People who know me more intimately know well that I have a tendency for taking on many challenges at once. Over the last three years, I (perhaps ambitiously, if not dubiously) chose to concurrently pursue my Master's thesis, a Bachelor's in Computer Science, and a Specialization in Artificial Intelligence. This demanding series of endeavors proved to be an invaluable learning experience, due in large part to the steadfast support of my family, friends, and the wonderful community of professors and students at the Institute of Mathematics and Statistics of the University of São Paulo (IME-USP), especially the LIAMF laboratory.

I would like to express my deepest gratitude to my advisor, Professor Denis Deratani Mauá. Thank you not only for the usual guidance and insightful support expected of a Master's thesis, but also for continuously instigating my curiosity about the world of Probabilistic Logic Programming and its relation to Probabilistic Circuits. Your patience, expertise in explaining complex concepts, sound advice, and overall dedication to every piece of work related to yours, including this thesis and the articles we co-authored, have been truly foundational to my growth. I am especially grateful for your constant availability and the stimulating discussions that often ranged from early in the morning until late at night.

A special note of thanks goes to the members of both the Qualifying and Defense boards. Your incisive feedback and constructive discussions significantly helped me refine and improve this work.

To my friends and colleagues at IME-USP, and especially within the LIAMF lab, thank you for the support, the laughs, and all the interesting moments we shared over the years. This journey was made infinitely more enjoyable because of our community. A special acknowledgement is reserved for Renato Lui Geh, who I consider not only a dear friend but also a key mentor and inspiration alongside Denis, capturing my attention in the fields of Probabilistic Circuits, Neuro-Symbolic Artificial Intelligence, and beyond.

Lastly, and most importantly, I thank my family for their unwavering support and

encouragement throughout this journey: my mother Ana, my father André, and my brother Diogo. And, of course, a special and warm thank you to both my grandmother Fátima and my grandfather Chico, to whom this thesis is dedicated. Thank you all for everything.

Abstract

Jonas Rodrigues Lima Gonçalves. **Non-incremental knowledge compilation of probabilistic answer set programs**. Thesis (Master's). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2025.

Advances in Probabilistic Logic Programming (PLP) and Probabilistic Inference (PI) have demonstrated that Knowledge Compilation (KC) techniques, which compile probabilistic logic programs into Answer Set Programs (ASP), are essential for fast and exact inference. Despite the close relationship between Circuit Theory and the complexity of logical inference, it is possible to exploit the structure of PLPs to construct more succinct circuits that can efficiently answer probabilistic queries.

Although previous research has shown that the compilation of stratified programs can be achieved using various techniques, such as T_P-consequence or Loop Formulas, little has been explored in the context of PASP compilation, apart from top-down compilation approaches that translate the program into a Conjunction Normal Form (CNF) and subsequently apply knowledge compilation techniques. Decomposable Negation Normal Form (DNNF) compilation-based methods have also been investigated.

Drawing inspiration from successful works on the compilation of stratified PLP languages, this research aims to leverage the tractability and bottom-up nature of a special class of Arithmetic Circuits (AC), called Probabilistic Sentential Decision Diagrams (PSDDs), to compile PASP programs into PSDDs under the *Max Entropy* and *Credal* semantics. Furthermore, this bottom-up approach enables structural optimizations through modifications to the variable tree that governs the structure of the circuits, resulting in more succinct representations.

Finally, PSDDs are capable of performing exact inference in polynomial time and support numerous algorithms for learning and regularization, specifically designed for use in Probabilistic Circuits (PCs). This further underscores the potential of PSDDs as a tractable representation for PLPs and highlights the need to explore efficient heuristics for dynamically modifying the variable tree structure when compilingPASP programs into PSDDs.

Keywords: Probabilistic Logic Programming. Knowledge Compilation. Probabilistic Circuits.

Resumo

Jonas Rodrigues Lima Gonçalves. **Compilação de conhecimento não-incremental de programas de conjuntos de respostas probabilísticas**. Dissertação (Mestrado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2025.

Avanços em Programação Lógica Probabilística e Inferência Probabilística demonstraram que técnicas de Compilação de Conhecimento, as quais compilam programas lógicos probabilísticos em Programas de Conjunto de Respostas, são essenciais para inferência rápida e exata. Apesar da estreita relação entre a Teoria de Circuitos e a complexidade da inferência lógica, é possível explorar a estrutura dos Programas Lógicos Probabilísticos para construir circuitos mais concisos que podem responder eficientemente a consultas probabilísticas.

Embora pesquisas anteriores tenham mostrado que a compilação de programas estratificados pode ser alcançada usando várias técnicas, como a consequência do operador T_P ou *Loop Formulas*, pouco foi explorado no contexto da compilação de Programas de Conjunto de Respostas Probabilísticos, à parte de abordagens de compilação top-down (de cima para baixo) que traduzem o programa em uma Forma Normal Conjuntiva e subsequentemente aplicam técnicas de compilação de conhecimento. Métodos baseados na compilação para Forma Normal Negativa Decomponível também foram investigados.

Inspirada em trabalhos bem-sucedidos sobre a compilação de linguagens de Programação Lógica Probabilística estratificada, esta pesquisa visa alavancar a tratabilidade e a natureza bottom-up (de baixo para cima) de uma classe especial de Circuitos Aritméticos, chamados Diagramas de Decisão Sentencial Probabilísticos, para compilar Programas de Conjunto de Respostas Probabilísticos nestes diagramas sob a semântica de Entropia Máxima e Credal. Além disso, essa abordagem bottom-up permite otimizações estruturais por meio de modificações na árvore de variáveis que governa a estrutura dos circuitos, resultando em representações mais concisas.

Finalmente, os Diagramas de Decisão Sentencial Probabilísticos são capazes de realizar inferência exata em tempo polinomial e suportam inúmeros algoritmos para aprendizado e regularização, especificamente projetados para uso em Circuitos Probabilísticos. Isso reforça ainda mais o potencial dos Diagramas de Decisão Sentencial Probabilísticos como uma representação tratável para Programação Lógica Probabilística e destaca a necessidade de explorar heurísticas eficientes para modificar dinamicamente a estrutura da árvore de variáveis ao compilar Programas de Conjunto de Respostas Probabilísticos nos Diagramas de Decisão Sentencial Probabilísticos.

Palavras-chave: Programação Lógico Probabilística. Compilação de Conhecimento. Circuito Probabilísticos.

Lista of Abbreviations

2AMC	Second Level Algebraic Model Counting
AASC	Algebraic Answer Set Counting
AC	Arithmetic Circuit
AMC	Algebraic Model Counting
ASC	Answer Set Counting
BDD	Binary Decision Diagram
DNNF	Decomposable Negation Normal Form
d-DNNF	Deterministic Decomposable Negation Normal Form
EDLP	Extended Disjunctive Logic Programming
ELP	Extended Logic Programming
FOL	First Order Logic
KC	Knowledge Compilation
KR	Knowledge Representation
LC	Logic Circuit
LP	Logic Programming
NNF	Negation Normal Form
NP	Nondeteriministic Polynomial Time
OBDD	Ordered Binary Decision Diagram
PASP	Probabilistic Answer Set Programming
PC	Probabilistic Circuit
PLP	Probabilistic Logic Programming
PSDD	Probabilistic Sentential Decision Diagram
SDD	Sentential Decision Diagram
sd-DNNF	Smooth Decomposable Negation Normal Form
SDNNF	Structured Decomposable Negation Normal Form
WMC	Weighted Model Counting

List of Symbols

$X, Y, Z, \ldots$	Sets of variables
$x, y, z, \ldots$	Sets of assignments
$\langle f \rangle$	Semantics of Boolean formula f
$f \equiv g$	Equivalence between Boolean formulae f and g (i.e. $\langle f \rangle = \langle g \rangle$)
N, S, P, L	Sets of nodes
$Ch(N)$	Set of all children of node N
$Pa(N)$	Set of all parents of node N
$Desc(N)$	Set of all descendants of node N
$head(r)$	Head of a rule r
$body(r)$	The set of atoms inside the body of a rule r

List of Figures

List of Tables

List of Programs

Contents

Chapter 1

Introduction

Artificial Intelligence (AI) has achieved unprecedented prominence in both research and public discourse, largely driven by remarkable progress in large Neural Networks (NNs). From breakthroughs in Computer Vision with *AlexNet* (Alom *et al.*, 2018) in 2012, to mastering complex strategic games with *AlphaGo* (F.-Y. Wang *et al.*, 2016) in 2016, advancing biological discovery through *AlphaFold* (Ruff and Pappu, 2021) in 2018, and revolutionizing Natural Language Processing (NLP) with Large Language Models (LLMs) (Radford, Narasimhan, *et al.*, 2018; Radford, J. Wu, *et al.*, 2019; Brown *et al.*, 2020; OpenAI *et al.*, 2024), deep learning has reshaped what the field can accomplish.

While the pace of innovation continues to accelerate (Chang *et al.*, 2023; Gunasekar *et al.*, 2023; Sean Wu *et al.*, 2023), an equally pressing concern has emerged: the lack of transparency and trustworthiness in these purely data-driven systems (Sun *et al.*, 2024; X. Huang *et al.*, 2023; Y. Liu *et al.*, 2023). Such models excel at recognizing patterns from massive datasets but often fail to reason about them in a human-understandable way. As a result, their decisions are frequently opaque, difficult to verify, and prone to biases inherited from data.

This growing gap between performance and interpretability has reignited interest in *Neuro-Symbolic Artificial Intelligence (NeSy)* (Sarker *et al.*, 2021; Garcez *et al.*, 2022), an approach that seeks to combine the statistical learning capabilities of Neural Networks with the structured reasoning and explainability of symbolic systems. By bridging these two paradigms, Neuro-Symbolic AI aspires to enable more reliable, transparent, and generalizable forms of intelligence.

In order to tackle these issues, recent research has been studying several different approaches, such as the utilization of Logical Neural Networks (LNNs) (Riegel *et al.*, 2020; Qu and Tang, 2019), a Neuro-symbolic framework that aims to combine the best of both worlds: the expressiveness of logic and the learning capabilities of Neural Networks. In this type of model, every neuron has a meaning as a component of a formula in a weighted real-valued logic, yielding a highly interpretable disentangled representation (Riegel *et al.*, 2020). Inference is omnidirectional rather than focused on predefined target variables and corresponds to logical reasoning, including classical first-order logic theorem proving as a special case (Riegel *et al.*, 2020).

While this approach is promising, due to its one-to-one correspondence to systems of logical formulae and specification through logical constraints, it requires not only adaptation to the Neuro-symbolic framework but also a deep understanding of how to encode the logical constraints into the Neural Networks.

Another approach that has been gaining attention is the use of Probabilistic Logic Programming (PLP): logic programming languages, like Prolog, that leverage probabilistic inference to handle uncertainty and complexity in data. By integrating the symbolic capabilities of these languages with Neural Networks, one is capable of creating hybrid systems that use Neural Networks to estimate the probability of predicates and their relationships. Notable deep PLP systems are: DeepProbLog (Manhaeve, Dumancic, *et al.*, 2018) and NeurASP (Z. Yang *et al.*, 2023).

An advantage of using a PLP for deep learning tasks is that it allows the user to specify the logical constraints in a more natural way, with well-established syntax from languages such as Prolog or Answer Set Programming (ASP), and to leverage the highly optimized logical inference engines for these languages. Moreover, it not only provides a more explainable framework but also has been shown to be a more performant method for a wide range of tasks (Manhaeve, Dumancic, *et al.*, 2018; Z. Yang *et al.*, 2023).

1.1 Current Limitations of PLPs

Although there are promising results in the field of PLPs, the addition of logical constraints to neural systems is a double-edged sword. While logic is a powerful tool for representing knowledge and can be used to audit the reasoning of a program, it is also known to be intractable in many cases, especially when dealing with more expressive logics, such as first-order logic (Levesque, 1986; Levesque and Brachman, 1987). Thus, the logical inference of this type of model suffers from the same problems.

On the other hand, Probabilistic Inference is also known to be difficult, and some traditional approximate sampling methods, such as Gibbs sampling, do not achieve a desirable level of accuracy (Pajunen and Janhunen, 2021).

Therefore, the advantages of PLPs cannot be explored while these major limitations for their scalability are not addressed. One possible solution for the logical inference problem is to use Knowledge Compilation (KC), a well-studied method for translating propositional logic formulae into tractable logical circuits (A. Darwiche and P. Marquis, 2002). This field of study has seen many advancements in the last decades, with theoretical results that show relations between different classes of circuits and their capability of encoding logical formulae for tractable inference depending on the type of queries that are to be answered (A. Darwiche and P. Marquis, 2002). For the probabilistic counterpart, Variational Inference (VI) methods using circuits or adapting Answer Set Enumeration in the order of Optimality (ASEO) to work with circuits are promising approaches to be studied in the future.

1.2 Objective of this Work

It is well known that learning PLP models is a difficult and slow task (Eiter, Hecher, *et al.*, 2024; Vlasselaer, Van den Broeck, *et al.*, 2016). This is due to the fact that the logical constraints of PLPs must be taken into account during the computation of the gradients (Geh *et al.*, 2024), which can be a major bottleneck for the training process. This work aims to study the use of KC to address the computational bottleneck of probabilistic logical inference tasks within PLPs, with a specific focus on Probabilistic Answer Set Programming (PASP), a PLP language focused on expressing contradictory or vague knowledge. While the proposed techniques were tailored specifically for tractable probabilistic inference in PASP, they present a promising approach for general PLPs through the use of heuristics that leverage program structure or advancements in Non-Incremental compilation

Although the literature for methods for compiling stratified PLP programs is vast, contemplating techniques such as T_P-compilation (Vlasselaer, Van den Broeck, *et al.*, 2016) and *Loop Formulas* (Lee and Lifschitz, 2003), less is known about the more general class of PASP programs, where one has to take into account for both the logic and probabilistic semantics of the programs. In order to create tractable structures that correctly represent the desired semantics, one needs to impose constraints on the structure of the circuits (B. Wang *et al.*, 2025). The main approach for PASP in recent years of research consists of using a class of circuits called deterministic decomposable Negation Normal Form (d-DNNF) (Eiter, Hecher, *et al.*, 2024; Totis, De Raedt, *et al.*, 2023), using a procedure that is capable of compiling the program into a circuit, but at the cost of translating the program into a Conjunction Normal Form (CNF) formula, which involves the creation of auxiliary variables and clauses, which may lead to an exponential blow-up in the target representation, even for small programs (Eiter, Hecher, *et al.*, 2024).

Therefore, a desirable approach for this problem would be to directly compile the program into a circuit, without the need to translate the program into a CNF formula. The possibility of using a bottom-up approach is what allows for the compilation of PASP directly into a circuit, as the program can be grounded (turned into a propositional formula) and then one can build the circuit from the ground up, using operations similar to logical $\wedge$ and $\vee$.

Sentential Decision Diagrams (SDDs) have been shown to be able to compile (stratified) PLP programs in a bottom-up manner with great success (Vlasselaer, Van den Broeck, *et al.*, 2016). Even though theoretical results have shown that d-DNNF are a more succinct class of target language than SDD, i.e. they can represent the same Boolean functions with a more compact representation, the lack of constraints in their structure does not allow for the same direct bottom-up compilation approach as SDD. Hence, the trade-off of using a less succinct target representation, as SDDs, has been shown to be a more effective approach than the costly translation to CNF that many top-down approaches (usually via d-DNNF) suffer from (Vlasselaer, Renkens, *et al.*, 2014).

Thus, the main goal of this work is to study the use of KC for PASP programs for tractable inference under its more common semantics (MaxEnt and *Credal*).

Chapter 2

Probabilistic Answer Set Programming

We choose ASP as the object of study within the context of Knowledge Representation (KR) and Prolog because it allows for a purely declarative representation of problems, which contrasts with Prolog (Kowalski, 1979). This declarative nature means that the order of program rules and the order of sub-goals in a rule body does not matter to the processing, eliminating the need for explicit control knowledge (Eiter, Ianni, *et al.*, 2009). This differs from a principle like *Logic + Control*, characteristic of Prolog.

In the following sections, we present the foundational concepts of PASP, along with the properties and semantics necessary to understand the KC problem that is the focus of this work.

2.1 Notation and Definitions

Logic Programming Languages (LPs) are built from fundamental concepts, such as logical variables, constants, predicates, atoms, terms, and rules. Constants will be represented by lowercase starting letters or natural numbers, such as *turing*, *vonNeumann*, 0, or 1. Variables will be represented by uppercase starting letters, e.g., X, Y, $City$, $Name$. A term is defined either as a constant or a logical variable. Functional terms can be formed by combining terms with functions, e.g., $isNeighbour(jonas, Name)$, where $isNeighbour$ acts as a boolean function.

We define predicates as boolean functions that are used to represent properties or relations. An atom is written as $r(t_1, \ldots, t_n)$, where r is a predicate of arity n and each t_i is a term. We say that an atom is *ground* if it only contains constants; therefore, it does not contain any logical variables. If an atom is not *ground*, we say that it is *non-ground*.

Consider the following two atoms:

$$neighbour(jonas, brazilian) \qquad isNeighbour(jonas, Name),$$

the former expresses a property of *jonas* and is *ground*, while the latter expresses the

relationship *isNeighbour* between the constant *jonas* and the logical variable *Name*.

2.2 Normal Logic Programs

We introduce the concept of programs that include negation in their *body* rules, called Normal:

Definition 2.2.1 (Normal Logic Program)**.** A Normal Logic Program is a finite set of rules of the form:

```
a :- b1, ..., bM, not c1, ..., not cN.
```

where $M, N \geq 0$; a, b_i, c_i are atoms of a function- free First Order Logic (FOL); and NOT c represents the negation as failure of an atom c ($\neg c$). We also introduce the terminology of *subgoals* in the body, which are either atoms of the form b_i (also called *positive subgoals*) or $not\ c_i$ (*negative subgoals*).

2.2.1 Herbrand Universe, Base and Interpretation

Now, we define the concept of *Herbrand Universe* and *Base*:

Definition 2.2.2 (Herbrand Universe)**.** The *Herbrand Universe* $HU(P)$ of a logic program P is the set of all terms that can be formed from constants and functions in P (w.r.t. a predefined vocabulary L). Moreover, the *Hebrand Base* $HB(P)$ of P is the set of all ground atoms that can be formed from terms and predicates occurring $HU(P)$. Finally, ta *Herbrand Interpretation* is a subset of $HB(P)$, an interpretation I (a set denoting ground *truths*) over $HU(P)$.

Example 2.2.1. *Consider the following logic program P (Eiter, Ianni, et al., 2009):*

```
h(0, 0).
t(a, b, r).
p(0, 0, b).
p(f(X), Y, Z) :- p(X, Y, Z'), h(X, Y), t(Z, Z', r).
h(f(X), f(Y)) :- p(X, Y, Z'), h(X, Y), t(Z, Z', r).
```

Then, the Herbrand Universe $HU(P)$ is the union of the set containing all constants of P, $\{0, a, b, r\}$, and the set of terms that can be formed from these constants $\{f(0), f(a), f(b), f(r), f(f(0)), f(f(a)), ...\}$. Whereas, the Herbrand Base, $HB(P)$ is given by the set of all ground atoms assertions:

$$\{p(0,0,0), p(a,a,a), \ldots, h(0,0), \ldots, t(0,0,0), t(a,a,a), \ldots\}$$

Finally, we list a few Herbrand Interpretations over $HU(P)$:

- $I_1 = \emptyset$;
- $I_2 = HB(P)$;
- $I_3 = \{h(0,0), t(a,b,r), p(0,0,b)\}$.

Note that not all interpretations are consistent with the program P. For instance, the interpretation I_1 is contradictory, because it does not contain any of the facts of P.

2.2.2 Grounding of a Logic Program

With the notion of Herbrand Universe and Base, we can now have a moral formal definition of *grounding* (Eiter, Ianni, *et al.*, 2009):

Definition 2.2.3 (Grounding)**.** We define a ground instance of a clause C, of a logic program P, as any clause C' obtained from C by applying a substitution

$$\theta : Var(C) \rightarrow HU(P),$$

where $Var(C) \in \mathcal{V}(P)$ is the set of variables in C. Moreover, the grounding of a program P is the set of all possible ground instances of the clauses in P and is denoted by $ground(P) = \bigcup_{C \in P} ground(C)$ (the union of all ground instances for all the clauses in P).

2.2.3 Interpretation of a Logic Program

Following the formalization of a concept like grounding, we have the definition of an interpretation of a logic program:

Definition 2.2.4 (Interpretation)**.** The interpretation I of a logic program P is a model of P that is compatible with the assertions in P. That is, I is a model of

- A ground clause $C =$ `prologa :- b1, ..., bM, not c1, ..., cN`, denoted $I \models C$, if either $\{ a, c_1, \dots, c_N \} \cap I \neq \emptyset$ or $\{ b_1, \dots, b_M \} \nsubseteq I$;
- A clause C, denoted $I \models C$, if $I \models C'$ for every $C' \in ground(C)$;
- A program P, denoted $I \models P$, if $I \models C$ for every clause $C \in P$.

That is: an interpretation I is a model of a program P if it is compatible with all the ground instances of the clauses of P.

We call a model I of a program P a *minimal model*, if there is no other model J of P such that $J \subset I$. Although, Normal Logic Programs can have multiple minimal models for a program, it is true that Definite Logic Programs only have one minimal model (Eiter, Ianni, *et al.*, 2009).

2.2.4 Immediate Consequence Operator

To establish the semantics of normal logic programs, we define the notion of the *immediate consequence operator* T_P:

Definition 2.2.5 (T_P Operator)**.** Let P be a Normal Logic Program. The immediate consequence $T_P : 2^{HB(P)} \rightarrow 2^{HB(P)}$ of a set of ground atoms I is defined as:

$$T_P(I) = \left\{ a \;\middle|\; \begin{array}{l} \texttt{a :- b1, ..., bM, not c1, ..., cN} \in ground(P), \\ \text{such that } \{b_1, \dots, b_M\} \subseteq I \text{ and } \{c_1, \dots, c_N\} \cap I = \emptyset \end{array} \right\},$$

where $ground(P)$ denotes the set of all ground atoms in P.

The most important property of the immediate consequence operator is that it is *monotone* (Bogaerts and Van den Broeck, 2015) and thus, by the Knaster-Tarki Theorem, has a least fixed point, called $lfp(T_P)$, which is the least model of P (Eiter, Ianni, *et al.*, 2009). Moreover, the sequence obtained by iterating T_P starting from the empty set converges to the fixed point $lfp(T_P)$.

2.2.5 Negation and Stratification

Even though different classes of Logic Programs, such as Definite and Normal programs, try to capture the intrinsic duality of expressiveness versus efficiency in the field of KR and LPs, these definitions only take into account the different types of rules that can be present in a program, but not their structure.

However, understanding the impact of the negation operator's introduction to program rules is of utmost importance, as it directly creates the possibility of having a collection of multiple minimal models. This change creates the necessity of defining a proper semantic for attributing meaning to negation in logic problems, where there are two main approaches:

- Define a single model for a program, which possibly tries to capture problematic classes of programs due to negation possibly creating multiple minimal models. This approach has success when dealing with *stratified* programs, which we define later. For general normal programs, the most popular semantics following this approach is based on the *well-founded semantics* (that we do not define here for the sake of brevity) (Eiter, Ianni, *et al.*, 2009; Van Gelder *et al.*, 1991).

- Define a collection of models for a program, abandoning the necessity of only one model. This approach gives rise to ASP and its semantics, such as the *stable model semantics*, which we also define later.

One possible way of determining whether a program is *stratified* or not is by finding an ordering for the evaluation of its rules, such that the value of negative literals can be predicted by the values of positive literals (Eiter, Ianni, *et al.*, 2009). But we can also define a more general approach by analyzing the structure of the program, that is, the dependency graph:

Definition 2.2.6 (Dependency Graph)**.** Let P be a ground program. Then, its dependency (multi) graph $dep(P)$ is a tuple of the form (V, E), where

- A set of nodes V, where each node $v \in V$ is a ground atom occurring in P; and

- A set of edges E, where each edge $(v, w) \in E$ (v pointing to w, $v \rightarrow w$) occurs if and only if v is the head atom of a rule whose body contains a literal w. If the literal is negative, then the edge is marked as $*$ (v, w) ($v \rightarrow^* w$).

We call a graph *acyclic* if its grounded dependency graph does not contain any (directed) cycles.

By utilizing the notion of a dependency graph, we can define the concept of stratification:

Definition 2.2.7 (Stratification)**.** Let P be a logic program, and let $G = (V, E)$ be its dependency graph. A stratification of P is a partition $\Sigma = \{ S_i \mid i \in \{ 1, \dots, n \} \}$ of the predicates of P into n non-empty pairwise disjoint sets, such that:

1. If $v \in S_i$, $w \in S_j$, and $(v, w) \in E$, then $i \leq j$ (positive dependency must be within the same stratum or to a higher/later stratum); and
2. If $v \in S_i$ and $* \ (v, w) \in E$, then $i > j$ (negative dependency must be to a strictly lower/earlier stratum).

A program is called *stratified* if it has some stratification Σ, and its respective partitions S_i are called *strata*.

2.2.6 Stable Model Semantics

As stated before, Normal Logic Programs can have multiple minimal models, which is a consequence of the introduction of negation in the program rules. Following, we have a program where the introduction of negation creates multiple minimal models:

Example 2.2.2. *Let P be the following logic program:*

```
researcher(computability).
machine(X) :- researcher(X), not lambda(X).
lambda(X) :- researcher(X), not machine(X).
```

Then, note that the program P is not stratified. Furthermore, there is not only one minimal model for P, but two:

1. $M_1 = \{ researcher(computability), machine(computability) \}$*; and*
2. $M_2 = \{ researcher(computability), lambda(computability) \}$.

Hence, to attribute meaning to negation, we first introduce the concept of *reduction* of a program, and then introduce one of the most important semantics for logic programs, the Stable Model semantics.

Definition 2.2.8 (Reduct (short GL-reduct))**.** Let P be a Normal Logic Program and I be an interpretation. Then, the *reduct* of P w.r.t I, denoted P^I, is obtained by:

1. Grounding the program P;
2. Removing rules with $notc$ in the body, for each $c \in I$; and
3. Removing all negative literals $notc$ from the remaining rules.

Hence, a Gelfond-Lifschitz reduction produces a program that enforces the truth of the atoms in the interpretation I. The first condition, when a literal $c \in I$, makes the negative subgoal $notc$ false. Consequently, any ground rule containing $notc$ in its body is removed, as its body cannot be satisfied, and it will not contribute to the least model. When the second condition occurs, if a literal $c \notin I$, it is possible to assume that $notc$ is indeed true; therefore, it can be removed from the (body of the) rule.

A direct consequence of the definition of the reduct is that the reduction product, P^I, is a positive program. Using this property, it is easy to see that P^I has a least model $LM(P^I)$; and, if P^I is consistent (if it does not present a contradiction), then the least model of P^I is I itself. This bijective relation between P^I and I, of one being able to obtain I from P^I due to it being a least model of the program, and P^I being the reduct of P w.r.t. I, is the basis of the Stable Model semantics:

Definition 2.2.9 (Stable Model)**.** Let P be a normal logic ground program. An interpretation I is a stable model of P if I is a minimal model of the reduct P^I of P w.r.t. I.

Since a normal program may have several stable models, it is possible to have multiple interpretations that are stable models. Therefore, a normal program may have several stable models (or even none, as we stated earlier). For instance, the example above has two stable models, M_1 and M_2.

An interesting property of stable models is that they are fixed points of the immediate consequence operator T_P. Formally, a stable model I of P satisfies $T_P(I) = I$ (but the converse is not necessarily true) (Eiter, Ianni, *et al.*, 2009).

2.2.7 Reasoning

There are three types of reasoning with logic programs under the stable model semantics. We enumerate them in order of complexity, where the former can be reduced as an instance of the latter:

1. **Consistency** (Satisfiability): decide whether a program has at least one answer set (consistent);
2. **Brave reasoning**: given a ground literal Q, decide whether some (at least one) answer set satisfies Q. If it does, Q is called a *brave consequence* of the program; and
3. **Cautious reasoning**: given a ground literal Q, decide whether all answer sets satisfy Q. If all of them do, Q is called a *cautious consequence*.

2.3 Probabilistic Logic Programming

Before defining PLPs, we first need to establish the semantics for programs incorporating probabilistic distributions. One of the first works that supports the semantics we are going to define is the work by (Sato, 1995), where the idea behind a distribution semantics is to have probability associated with a set of *independent* events in such a way that a unique probability measure is induced over all interpretations of ground atoms (Cozman and Mauá, 2020).

As one could imagine from the date of Sato's work, the semantics was not specially designed for ASP. One of the consequences is that it is not possible to apply Sato's semantics to general ASP programs; his idea of distribution semantics fails to guarantee the specification of a single probability distribution over ground atoms under the usual two-valued logic (Cozman and Mauá, 2020). Thus, the extension of Sato's distribution semantics becomes necessary in order to specify probability measures by ASP programs.

Hence, we define PLPs as a pair (P, PF), where P is a **logic program** and PF is a set of **probabilistic facts**:

Definition 2.3.1 (Probabilistic Fact)**.** A probabilistic fact is a pair consisting of an atom A and a probability value α, which we denote by $\alpha :: A$ (Cozman and Mauá, 2017).

Furthermore, atoms of probabilistic facts may contain logical variables ($\alpha :: r(X_1, \ldots, X_n)$), and are therefore unground. When considering this case, we interpret such a probabilistic fact as the set of all grounded probabilistic facts, obtained after grounding the variables in the atom.

Properties from logic programs, such as *stratification*, *acyclicity*, and *definiteness*, are reflected in PLPs as well. For instance, a PLP (P, PF) is said to be acyclic if the dependency graph of P is acyclic.

The definition of PLPs does not directly extend the class of programs that we were used to; it only associates probabilities through the use of probabilistic facts. A more in-depth definition would be to extend the notion of Extended Disjunctive Logic Programming (EDLP) by adding probabilistic facts to the heads of the rules:

Definition 2.3.2 (Annotated Disjunctive Rules)**.** A probabilistic extension of an EDLP is called Annotated Disjunctive Rules (ADR), a set of rules of the form:

```
1    p1::a1; ...; pK::aK :- b1, ..., bM, not c1, ..., not cN,
```

where $p_1, \ldots, p_k$ are nonnegative real values whose sum is smaller than or equal to 1. In this sense, a probabilistic fact of the form

```
1    p::a
```

is a special case (and a syntactic sugar) of an ADR, where `prologp::a` is equivalent to `prologp::a (1-p):-f`, where f is an atom that is never true (Geh *et al.*, 2024).

2.3.1 Probabilistic Answer Set Programming

Although we have briefly defined Sato's distribution semantics, there is a need for a more formal description. Furthermore, this distribution semantics that Sato proposed is not well adapted to consider ASP programs.

It is important to note that we are defining a probability distribution over PLPs, and not logical ones. This makes probabilistic semantics over PLPs agnostic w.r.t. the logical semantics of the program, which is particularly useful when considering non-stratified programs that may use a *stable model* or *well-founded* semantics (Eiter, Ianni, *et al.*, 2009; Mauá and Cozman, 2020).

Definition 2.3.3 (Probabilistic Choice)**.** A PLP (P, PF) with n probabilistic facts can generate a total of 2^n different logic programs: one for each possible assignment of the probabilistic facts (when considering binary probabilistic facts). The assignment of a probabilistic fact $\alpha :: A$ can be seen as choosing to keep or erase the fact A from the program (Cozman and Mauá, 2017).

As commented before, these *probabilistic choices* are assumed to be independent.

Definition 2.3.4 (Total Choice)**.** A *total choice* θ for a PLP (P, PF) is a subset of the set of grounded probabilistic facts. Thus, θ can be interpreted as a set of ground facts that are probabilistically selected to be included in P, while all other ground facts obtained from probabilistic facts are discarded (Cozman and Mauá, 2017).

Because the probabilistic choices are assumed to be independent, the probability of a total choice can be easily computed by the product over the grounded probabilistic facts, where a probabilistic fact $\alpha :: A$ that is chosen to be included contributes with a factor of α, and, if it is discarded, it contributes with a factor of $(1 - \alpha)$ (Cozman and Mauá, 2017).

An important property of total choices is that, for a total choice θ, the induced logic program (w.r.t. P), denoted by $P \cap PF^{\downarrow\theta}$, is a normal logic program (Cozman and Mauá, 2017; Geh *et al.*, 2024). Therefore, for an ADR, a total choice θ induces a normal logic program, where each rule r from the ADR is transformed into a rule of the form

```
1    a_i :- b1, ..., bM, not c1, ..., not cN,
```

where i is the corresponding index of the probabilistic choice in θ (Geh *et al.*, 2024).

Whenever we have a total choice θ for a PLP such that $P \cup PF^{\downarrow\theta}$ is stratified, the resulting induced program has only one *stable model*, as we have seen in the earlier sections. Hence, Sato's distribution semantics can be naturally extended to this class of programs, and does not create any controversial interpretation of the probabilistic semantics.

A more interesting case is when we have a total choice θ for a PLP that induces a (non-stratified) logical program with multiple *stable models*. In this case, we have the need to define semantics capable of handling this situation: the Credal and MaxEnt semantics, which we define next.

First, we define the notion of *consistency* for PLPs:

Definition 2.3.5 (Consistency of Probabilistic Logic Programs)**.** A PLP (P, PF) is *consistent* if, for each total choice θ, the induced logic program $P \cup PF^{\downarrow\theta}$ has at least one stable model (Cozman and Mauá, 2017).

Given that we have a definition to express PLPs that are able to have stable models, independent of the total choice, we are capable of formalizing the notion of a *probability model* for PLPs:

Definition 2.3.6 (Probability Model for Probabilistic Logic Programs)**.** A *probability model* for a consistent PLP (P, PF) is a probability measure $\mathbb{P}$ over interpretations of P, such that (Cozman and Mauá, 2017):

1. Every interpretation I with $\mathbb{P}(I) > 0$ is a stable model of (the normal logic program) $P \cup PF^{\downarrow\theta}$ w.r.t. the total choice θ that agrees with I on the probabilistic facts; and
2. The probability of each total choice θ is the product of the probabilities for all individual choices in θ:

$$\mathbb{P}(\{\, I \mid I \cap C = C \,\}) = \prod_{\alpha :: A \mid A \in C} \alpha \prod_{\alpha :: A \mid A \notin C} (1 - \alpha).$$

Credal Semantics

As the first PASP semantics to be described, we introduce the *credal semantics*, which is based on the idea of using intervals (containing *upper* and *lower*) to represent the uncertainty of the probability (Lukasiewicz, 2005; Lukasiewicz, 2007).

Definition 2.3.7 (Credal Semantics)**.** The *credal semantics* for a consistent PLP (P, PF) is the set of all probability models for (P, PF) (Cozman and Mauá, 2017).

Moreover, the credal semantics is a *closed* and *convex* set of probability measures, and corresponds to the set of all probability measures that dominate an infinitely monotone Choquet capacity (Cozman and Mauá, 2017) (more about Choquet capacities can be found in Cozman and Mauá, 2020). As such, the credal semantics of any set of models $\mathcal{M}$ is characterized by an interval $[\underline{\mathbb{P}}(\mathcal{M}), \overline{\mathbb{P}}(\mathcal{M})]$:

$$\underline{\mathbb{P}}(\mathcal{M}) = \sum_{\theta \in \Theta \,:\, \Gamma(\theta) \subseteq \mathcal{M}} \mathbb{P}(\theta), \qquad \overline{\mathbb{P}}(\mathcal{M}) = \sum_{\theta \in \Theta \,:\, \Gamma(\theta) \cap \mathcal{M} \neq \emptyset} \mathbb{P}(\theta), \tag{2.1}$$

where Θ is the set of all total choices, and $\Gamma(\theta)$ is the set of stable models associated with the total choice (Cozman and Mauá, 2017; Cozman and Mauá, 2020).

Given a PLP, this interval can be computed for a set of assignments $\mathbf{Q}$ for ground atoms by the following algorithm (Cozman and Mauá, 2017):

1. Given a PLP (P, PF) and $\mathbf{Q}$, initialize variables a and b with 0;
2. For each total choice θ, compute the set S of all stable models of $P \cap PF^{\downarrow\theta}$, and:
   (a) If $\mathbf{Q}$ is *true* in every stable model in S, then $a \leftarrow a + P(\theta)$; and
   (b) If $\mathbf{Q}$ is *true* in some stable model of S, then $b \leftarrow b + P(\theta)$.
3. Return $[a, b]$ as the interval $[\underline{\mathbb{P}}(\mathbf{Q}), \overline{\mathbb{P}}(\mathbf{Q})]$.

An interesting relation between the credal semantics and reasoning over Answer Sets is that the computation of the upper probability, $\overline{\mathbb{P}}(\mathbf{Q})$, involves *brave reasoning*, and the computation of the lower probability, $\underline{\mathbb{P}}(\mathbf{Q})$, involves *cautious reasoning* (Cozman and Mauá, 2017).

Maximum-Entropy Semantics

To conclude this chapter, we define one of the most famous semantics for PASP: the *Maximum Entropy* semantics, also known as MaxEnt semantics:

Definition 2.3.8 (Maximum Entropy (MaxEnt) Semantics)**.** This semantics is based on the principle of maximum entropy: evenly dividing the probability mass among all stable models of a program (induced by a total choice). Formally, for a stable model $\mathcal{M}$, the probability $\mathbb{P}(\mathcal{M})$ is given by:

$$\mathbb{P}(\mathcal{M}) = \sum_{\theta \,:\, \mathcal{M} \in \Gamma(\theta)} \frac{\mathbb{P}(\theta)}{n},$$

where n is the number of stable models associated with the total choice θ (Cozman and Mauá, 2017).

Chapter 3

Knowledge Compilation

The study of Logic Circuits (LCs) in the field of Circuit Theory (CT) predates the creation of Logic Programming languages and other topics covered in this thesis. In greater detail, Logic Circuits are integral components in one of the most fundamental theorems of computer theory: the Cook-Levin Theorem, which states that SAT is NP-complete (Cook, 1971; Levin, 1973). Moreover, the expressiveness of these SAT-like problems in modeling real-world scenarios, such as Enumerative Combinatorial problems (valiant, 1979) or interdisciplinary applications (e.g., Bioinformatics (Bacchus *et al.*, 2021)), motivates the development of efficient structures and algorithms to solve these problems.

Therefore, with the preliminary definitions of PASP in the previous chapter 2, we have sufficient motivation to proceed with a formal definition of Logic Circuits, needed to perform the Knowledge Compilation for PASP proposed in this work. We define the Negation Normal Form Language (NNF), a language capable of unifying well-known succinct data structures for representing Boolean formulas, such as Binary Decision Diagrams (BDDs), Sentential Decision Diagrams (SDDs), and Decomposable Negation Normal Form (DNNFs) (A. Darwiche and P. Marquis, 2002; Adnan Darwiche, 2011).

3.1 Negation Normal Form Language

We begin with the formal definition of the NNF, a language capable of unifying well-known efficient data structures for representing Boolean formulas, such as CNF, DNF, OBDD, DNNF, SDD, and MODS (A. Darwiche and P. Marquis, 2002).

The notation and definitions in this section are heavily influenced by the work of (A. Darwiche and P. Marquis, 2002; Adnan Darwiche, 2011) and (Kisa *et al.*, 2014).

Definition 3.1.1 (Negation Normal Form Language)**.** Let V be a set of propositional variables. Then, we say that a sentence is in NNF_V if it is a *rooted* Direct Acyclic Graph (DAG) where each leaf is labeled as one of the following:

- *True* or *False*, or
- X or $\neg X$, with $X \in V$;

and each internal node is labeled with $\wedge$ or $\vee$ and can have arbitrarily many children.

The *size* of a sentence Σ in NNF_V, denoted $|\Sigma|$, is the number of edges of its DAG; and the *height* is the depth, the longest path from the root to a leaf of the DAG.

An important note is that the NNF language is not a *target compilation* language (unless $P \neq NP$) (Papadimitriou, 2003), as it is not tractable enough to permit a non-trivial set of polynomial queries (A. Darwiche and P. Marquis, 2002). However, many of its subsets are target compilation languages, as they impose structural constraints that enable succinct representations and allow for efficient algorithms to perform queries on them.

Another important class of languages is called *representation languages*, which we loosely define as human-interpretable languages (A. Darwiche and P. Marquis, 2002). Both CNFs and Horn Clauses (or Rules) are examples of representation languages.

3.1.1 Flat Normal Form Language

In addition to our earlier definitions of *target compilation* and *representation* languages, we can also categorize NNFs subsets as *flat* or *nested* languages. We describe the former first as follows:

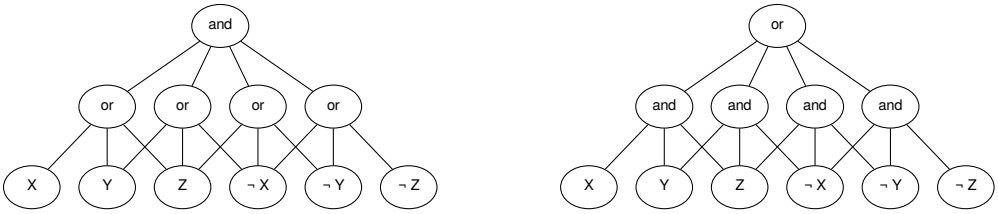
Definition 3.1.2 (Flat Normal Form Language)**.** A *flat* NNF is an NNF that satisfies the following three properties:

- **Flatness:** The height of the sentence's DAG is at most 2;
- **Simple-Conjunction:** All conjunction nodes (**and**-nodes) have children that are leaves sharing no variables; and
- **Simple-Disjunction:** All disjunction nodes (**or**-nodes) have children that are leaves sharing no variables.

Note that since the leaves are either literals or constants, **and**-nodes of a flat NNF are conjunctive clauses. Similarly, the **or**-nodes of a flat NNF are disjunctive clauses.

We define $f-\text{NNF}$ as the subset of NNFs that satisfies the *flatness* property.

Figure 3.1 shows examples of a CNF and a DNF, respectively, using the NNF language. Note that Figure 3.1a satisfies both the *flatness* and *simple-disjunction* properties, while Figure 3.1b satisfies the *flatness* and *simple-conjunction* properties.



(a) *Conjunction Normal Form (CNF).* **(b)** *Disjunction Normal Form (DNF).*

Figure 3.1: *Examples of CNF (Figure 3.1a) and DNF (Figure 3.1b) formulas using the NNF language.*

Remark. A simple observation is that CNFs and DNFs are both flat NNFs. Moreover, CNFs are $f-$NNF and satisfy the *simple-disjunction* property, while DNFs are $f-$NNF and satisfy the *simple-conjunction* property.

3.1.2 Nested Normal Form Language

Now, we introduce the *nested* subset of the NNF. Unlike the *flat* subset, the *nested* does not impose restrictions on the height of a sentence. Instead, it imposes more structurally complex constraints, called *decomposability*, *determinism*, *smoothness*, *decisiveness*, and *order*.

Before defining each of these properties, we introduce some auxiliary notation. Let Σ be a sentence in NNF_V (for some set of variables V). Then, for a node C in Σ, we define $Vars(C)$ as the set of variables that appear in the leaves of the subtree rooted at C. Moreover, we define $Var(\Sigma)$ as $Vars(R)$, where R is the root of Σ.

For the following definitions, we consider a sentence Σ as an NNF in NNF_V.

Definition 3.1.3 (Decomposability)**.** For each conjunction node C in Σ, the conjuncts of C (its children) do not share variables. Formally, for two children C_i and C_j of the **and**-node C, we have that $Vars(C_i) \cap Vars(C_j) = \emptyset$; thus, the children of C are pairwise disjoint with respect to the variables they contain.

Definition 3.1.4 (Determinism)**.** For each disjunction node D in Σ, any two disjuncts of D are logically contradictory. This means that, for two children D_i and D_j of the **or**-node D, we have that $C_i \wedge C_j \models \bot$; thus, the children of D are pairwise contradictory.

Definition 3.1.5 (Smoothness)**.** For each disjunction node D in Σ, each disjunct of C mentions the same variables. Hence, for two children D_i and D_j of the **or**-node D, we have that $Vars(D_i) = Vars(D_j)$.

Definition 3.1.6 (Decisiveness)**.** A node C in Σ is decisive if it is labeled as *True* or *False*, or if it is an **or**-node of the form $(X \wedge \alpha) \vee (\neg X \wedge \beta)$, where X is a variable, α and β are decisive (also called **decision** nodes). Then, we say that C decides on X, and denote this decision as $dVars(C) = X$.

Definition 3.1.7 (Order)**.** Let $<$ be a total order on the variables in V. Then, we say that Σ respects the order $<$ if, for two distinct decisive **or**-nodes N and M in Σ, where N is an ancestor of M, we have that $dVars(N) < dVars(M)$.

Similarly to *flat* NNFs, we can categorize distinct subsets of NNFs that satisfy some of these properties. For instance, we can define the subsets DNNF, deterministic decomposable NNF (d-DNNF), smooth d-DNNF (sd-DNNF), and others as follows:

Definition 3.1.8 (Nested Subsets of Negation Normal Form)**.** The language DNNF is the subset of NNF that satisfies the *decomposability* property. Moreover, we define $s-$NNF and $d-$NNF as the subsets of NNF that satisfy the *smoothness* and *determinism* properties, respectively.

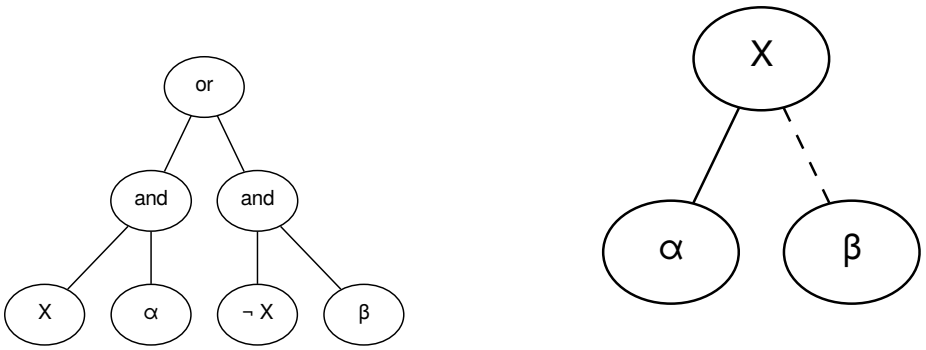
Finally, we define d-DNNF as the subset of DNNF that satisfies the *determinism* property, and sd-DNNF as the subset of d-DNNF that satisfies the smoothness property.

3.2 Binary Decision Diagrams

As mentioned in the introduction chapter 1, the BDD language is well-known for its efficiency in representing Boolean formulas. For many years, BDss were considered the most efficient language for representing general Boolean formulas in real-world applications (Franco and Martin, 2009). However, the introduction of more general languages, such as smooth (sd-DNFFs), which are capable of representing formulas more succinctly, has shifted this belief (a result that we discuss later in this chapter).

Definition 3.2.1 (Binary Decision Diagrams)**.** The BDD language is the subset of NNF where the root of each sentence is a decision node.

It is important to note that the BDD language corresponds to Binary Decision Diagrams as described in (Bryant, 1986). One of the main differences between the BDD language and the data structure is that the latter uses a more compact notation, where *true* and *false* are denoted by 1 and 0, respectively. Decision nodes, of the form $(X \wedge \alpha) \vee (\neg X \wedge \beta)$, are represented by a node X with a left child α and a right child β. A common convention is that the edge to the left child is represented by a solid line, while the edge to the right child is represented by a dashed line. For a graphical representation of BDss, refer to Figure 3.2.



(a) *Sentence in BDD.* **(b)** *BDD for the sentence in Figure 3.2a.*

Figure 3.2: *Example of the same Boolean function represented as an NNF structure adhering to the BDD language constraints (Figure 3.2a) and as the corresponding canonical BDD data structure (Figure 3.2b).*

There are two subsets of BDss that are particularly important: the Free Binary Decision Diagrams (FBDDs) and the Ordered Binary Decision Diagrams (OBDDs). The former has a property called *read-once*, which means that for each path from the root to a leaf, each variable appears at most once. The latter has a property called *ordering*, which enables it to have canonical representations. In this context, the order of the variables is fixed and consistent for all OBDDs that represent the same function. This property also enables operations such as *conjunction* and *disjunction*, making it possible to construct OBDDs from the bottom up by applying these operations (Akers, 1978; Bryant, 1986).

Definition 3.2.2 (Free Binary Decision Diagrams)**.** The FBDD language is the subset of the BDD language that intersects with the DNNF language.

Remark. By constraining a BDD to follow the **decomposability** property, one obtains the **read-once** property, as mentioned earlier.

Definition 3.2.3 (Ordered Binary Decision Diagrams)**.** The OBDD language is the subset of the FBDD language that follows the **order** property.

Remark. By constraining an FBDD to follow the **order** property, one enables the use of the **apply** operation and many other efficient algorithms to manipulate OBDDs (Bryant, 1986).

Although they are no longer considered state-of-the-art in Circuit Theory for succinctly representing Boolean formulas, their historical importance, combined with the possibility of bottom-up construction, still makes BDDs a relevant topic. In particular, this language had a significant influence on the development of SDDs, a subset of DNNFs that enables bottom-up construction and efficient algorithms for performing a diverse range of logical queries (Adnan Darwiche, 2011).

3.3 Sentential Decision Diagrams

Sentential Decision Diagrams are a subset of DNNFs that aim to achieve the succinctness of more general d-DNNF, which are capable of reasoning over a diverse range of probabilistic applications, while still allowing for bottom-up construction and efficient operations (Adnan Darwiche, 2011). Another desirable property that SDDs aim to achieve is *canonicity*, which allows OBDDs to optimize their succinctness by searching for the best variable ordering (instead of a process of searching for an optimal compilation) (Adnan Darwiche, 2011).

To define this language, we need to introduce two new properties: *structured decomposability* and *strong determinism*, both generalizations of the homonymous properties described in subsection 3.1.2.

It is worth noting that these two concepts, *structured decomposability* and *strong determinism*, are based on generalizations of *variable ordering* and *Shannon decomposition*, respectively, which are both used in OBDDs to achieve canonicity and succinctness. These generalizations are referred to as *v-trees* and *strong deterministic decompositions*, respectively (Adnan Darwiche, 2011).

Moreover, by introducing weaker constraints than OBDDs, SDDs are capable of achieving more succinct representations than OBDDs, while still preserving canonicity and allowing for efficient algorithms to perform Boolean operations (Adnan Darwiche, 2011).

3.3.1 Structured Decomposability

Before introducing the set of constraints that enable a more structured version of d-DNNF, by requiring a structured version of the decomposability property, we need to introduce the concept of *v-trees*, as follows:

Definition 3.3.1 (V-tree)**.** A *v-tree* for a set of variables V is a full rooted binary tree whose leaves are in one-to-one correspondence with the variables in V.

For a node n in a *v-tree* T, we define $\text{vars}(n)$ as the set of variables in the leaves of the subtree rooted at n. Moreover, for each non-leaf node n, we define n^l and n^r as the left and right children of n, respectively.

Lastly, if V includes the variables of a formula Δ, we say that T is a *v-tree* for Δ.

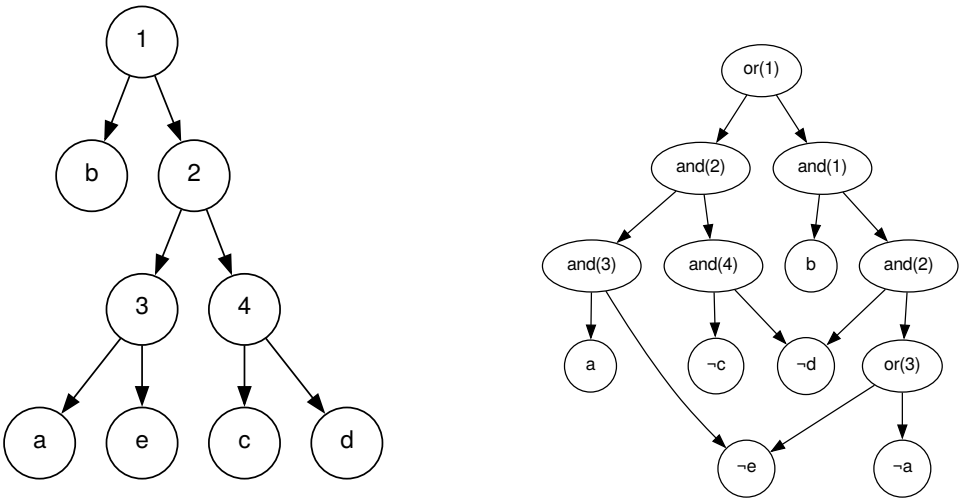
Definition 3.3.2 (DNNF w.r.t. a V-tree)**.** We say that a DNNF Σ respects a *v-tree* T if, for each conjunction $\alpha \wedge \beta$ in Σ, there is a node n in T such that $\text{vars}(\alpha) \subseteq \text{vars}(n^l)$ and $\text{vars}(\beta) \subseteq \text{vars}(n^r)$.

The *decomposition node* (*d-node*) of a sub-formula γ of Σ is defined as the deepest node n in T such that $\text{vars}(\gamma) \subseteq \text{vars}(n)$.

With the two previous definitions, we can now define the subset of DNNFs that respects a *v-tree* T:

Definition 3.3.3 (Smooth Decomposable NNF (SDNNF))**.** We denote the set of all DNNFs that respect a *v-tree* T as DNNF_T. Moreover, we define SDNNF as the set containing all DNNF_T, for any *v-tree* T.

Example 3.3.1. *Figure 3.3 depicts a v-tree T for variables $V = \{a, b, c, d, e\}$ and a* DNNF_T Σ.



(a) *A V-tree T with all internal nodes labeled with their indices.*

(b) *A structured DNNF formula for $(\neg a \vee \neg e) \wedge (a \vee b) \wedge (b \vee \neg c) \wedge (c \vee \neg d) \vee (\neg b \wedge \neg d)$.*

Figure 3.3: *A V-tree T (Figure 3.3a) and a DNNF Σ (Figure 3.3b) that respects T. Each internal node of Σ is labeled with the index of its decomposition node (d-node) in T (Pipatsrisawat and Adnan Darwiche, 2008).*

The most interesting property of SDNNF is that it allows the application of the polynomial-time *Conjoin* operator (Pipatsrisawat and Adnan Darwiche, 2008). More

specifically, given two DNNFs Σ_1 and Σ_2 that respect the same V-tree T, the *Conjoin* operator constructs a new DNNF$_T$ Σ that represents the conjunction $\alpha \wedge \beta$ of the formulas represented by Σ_1 and Σ_2; and this operation can be performed in polynomial time with respect to the product of the sizes of Σ_1 and Σ_2.

3.3.2 Strong Determinism

When defining *strong determinism*, we must also introduce a fundamental concept for this property, referred to as an (X, Y)-decomposition of a Boolean function f. This decomposition allows f to be expressed in terms of functions on X and Y (Adnan Darwiche, 2011).

Definition 3.3.4 ((X, Y)-Decomposition)**.** Let f be a Boolean function of the form $f = (p_1(X) \wedge s_1(Y)) \vee \ldots \vee (p_k(X) \wedge s_k(Y))$. Then, f has an (X, Y)-decomposition $\{(p_1, s_1), \ldots, (p_k, s_k)\}$. Each ordered pair (p_i, s_i) is called an *element* of the decomposition, and both p_i and s_i are referred to as *prime* and *sub*, respectively.

The size of a decomposition is defined as the number of its elements.

Furthermore, if $p_i \wedge p_j = false$ for all $i \neq j$, then the decomposition is said to be *strongly deterministic* on X.

Sentential Decision Diagramsare a proper subset of SDNNFs that adhere to a more structured decomposition (Adnan Darwiche, 2011), defined as follows:

Definition 3.3.5 (X-partition)**.** Let $\alpha = \{(p_1, s_1), \ldots, (p_k, s_k)\}$ be an (X, Y)-decomposition of a Boolean function f, such that this decomposition is strongly deterministic on X. Then, α is called an X-partition of f if and only if its primes form a partition. Specifically, the primes of α are consistent, mutually exclusive, and the disjunction of all primes is valid.

Moreover, α is said to be *compressed* if and only if its subs are all distinct.

These types of partitions exhibit some interesting properties. For example, one can always convert an uncompressed partition into a compressed one by replacing elements (p, s) and (q, s) with $(p \vee q, s)$, which does not alter the function represented. Furthermore, it is clear by definition that *false* can never be a prime of an X-partition; and if *true* is a prime, then it is the only prime. Lastly, **primes determine subs**. Hence, primes of different X-partitions of the same function f must form different partitions (Adnan Darwiche, 2011).

A more notable observation is that OBDDs are based on the following X-partition of a function f: $\{(X, f \mid X), (\neg X, f \mid \neg X)\}$, where $f \mid X$ is the result of *conditioning* f on X (Adnan Darwiche, 2011). This specific decomposition is called a **Shannon decomposition**, and the *decisions* of OBDDs are binary, as they are based on the value of literal primes X and $\neg X$.

The main idea behind SDDs is to generalize this concept, where the decisions are not binary because X corresponds to a set of variables instead of a single variable. In this way, the decisions are based on the values of sentential primes (Adnan Darwiche, 2011).

3.3.3 Formal Definition

As mentioned earlier, SDDs are a proper subset of SDNNFs, structured decomposable NNFs, that also respect strong determinism via (compressed) X-partitions.

Definition 3.3.6 (Sentential Decision Diagrams (SDDs))**.** We say that α is an SDD that respects a v-tree T rooted at v if and only if:

- $\alpha = \top$ or $\alpha = \bot$;
- $\alpha = X$ or $\alpha = \neg X$, and T is a leaf labeled with X; or
- $\alpha = \{(p_1, s_1), \ldots, (p_k, s_k)\}$, v is an internal node, $p_1, \ldots, p_k$ are SDDs that respect subtrees of v^l, and $s_1, \ldots, s_k$ are SDDs that respect subtrees of v^r, and $\langle p_1 \rangle, \ldots, \langle p_k \rangle$ is a partition,

where $\langle p \rangle$ denotes a mapping from SDDs into Boolean functions.

In the third case, if the primes of α respect v^l and the subs respect v^r (instead of respecting the subtrees v^l and v^r), then the SDD α is said to be *normalized*.

The semantics of an SDD α is defined, with respect to the items above, as follows:

- $\langle \top \rangle = true$ and $\langle \bot \rangle = false$;
- $\langle X \rangle = X$ and $\langle \neg X \rangle = \neg X$;
- $\langle \alpha \rangle = \bigvee_{i=1}^{k} \langle p_i \rangle \wedge \langle s_i \rangle$.

Finally, the size of an SDD α is denoted by $|\alpha|$ and is the sum of the sizes of all its decompositions.

Example 3.3.2. *Figure 3.4 shows an example of an SDD, but using a different notation than the one that is typically used. In this case, the notation of the SDD follows the NNF language, aiming to present this structure in a unified framework.*

One last important property of SDDs concerns their canonicity. More specifically, if two SDDs α and β are both compressed and trimmed (with the definitions of these properties as follows), respecting nodes in the same V-tree, then they represent the same Boolean function if and only if they are equal (Adnan Darwiche, 2011).

Definition 3.3.7 (Compressed and Trimmed SDDs)**.** We say that an SDD α is *compressed* if and only if, for all decompositions $\{(p_1, s_1), \ldots, (p_k, s_k)\}$ of α, all subs are pairwise inequivalent ($s_i \not\equiv s_j$, when $i \neq j$). Moreover, we say that α is *trimmed* if and only if it does not have any decompositions of the form $\{(\top, \beta)\}$ or $\{(\beta, \top), (\neg\beta, \bot)\}$.

3.3.4 SDD Operations

The importance of the properties defined above is not only related to *canonicity* but also to the efficient computation of Boolean functions. By imposing the *trimming* and *compression* properties, it is possible to assert that:

(i) Trivial SDDs $\bot$ and $\top$ are the only ones capable of representing the trivial Boolean functions, $false$ and $true$, respectively;

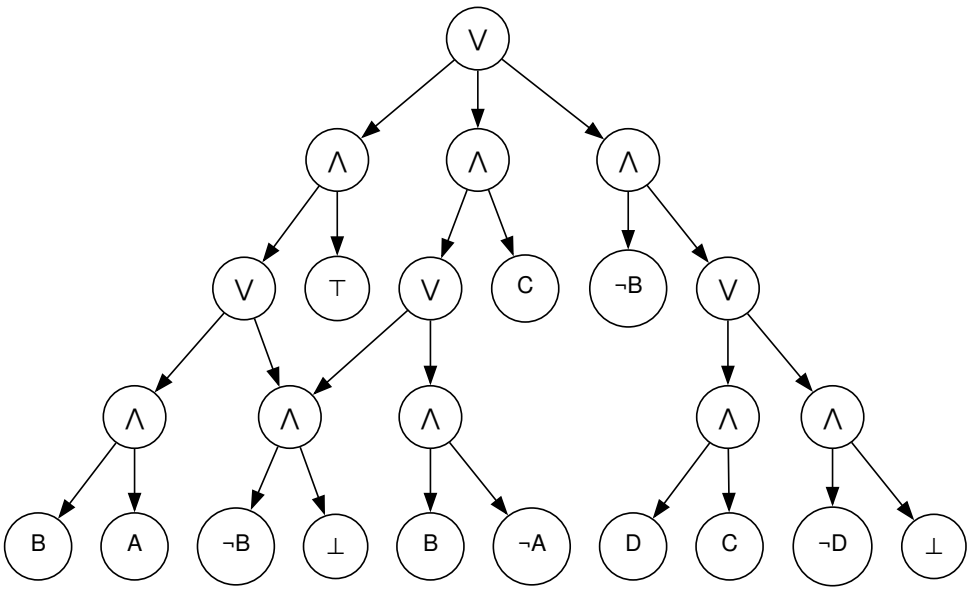


Figure 3.4: *A SDD (in NNF) representing a Boolean function $f = (A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$, following the variable order $\langle B, A, C, D \rangle$, with the V-tree described in more detail in (Adnan Darwiche, 2011).*

(ii) Non-trivial SDDs respect unique *V-tree* nodes, which depend on the functions they represent.

Another way to support this assertion is by imposing the *compression*, *light trimming*, and *normalization* properties, where *light trimming* corresponds to the exclusion of decompositions of the form $\{\top, \top\}$ or $\{\top, \bot\}$ by replacing them with $\top$ and $\bot$, respectively (Adnan Darwiche, 2011).

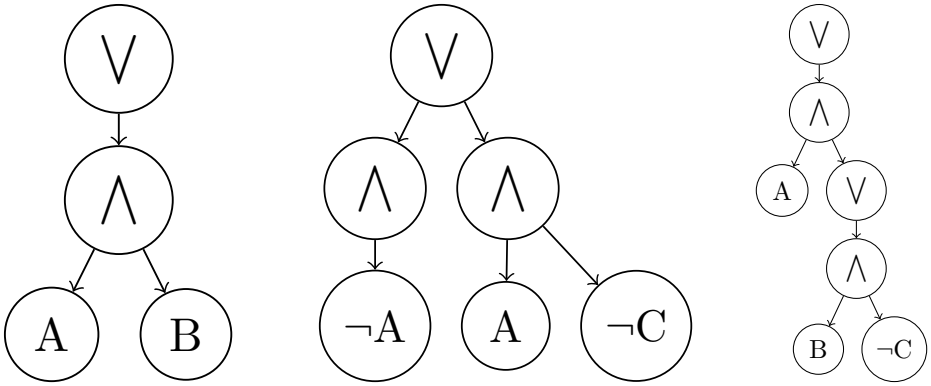
By ensuring these three properties, it is possible to support the claim that a Boolean function f can be uniquely represented by an SDD, given a fixed V-tree T.

Moreover, two normalized and lightly trimmed SDDs for the same V-tree, say α and β, can perform the application of any binary operator $\circ$, resulting in $\langle \alpha \rangle \circ \langle \beta \rangle$, in polynomial time $O(|\alpha| \cdot |\beta|)$, by following the *Apply* operator described in (Adnan Darwiche, 2011). This SDD *Apply* algorithm closely resembles the algorithm for the *Apply* operator in BDDs (Bryant, 1986), with the key difference being the use of V-trees instead of variable orderings.

Example 3.3.3. *Figure 3.5 illustrates a conjoin operation between two SDDs α and β, each representing $A \wedge B$ and $\neg A \vee \neg C$, following a balanced V-tree T.*

Comparison between SDDs and BDss

The two main characteristic properties of OBDDs are their canonicity and efficient computation of Boolean functions such as *conjunction*, *disjunction*, and *negation*. As mentioned earlier, the imposition of *structured decomposability* enables one to perform a



(a) *An SDD α representing $A \wedge B$.*

(b) *An SDD β representing $\neg A \vee \neg C$.*

(c) *The result of the conjoin operation between α and β.*

Figure 3.5: *Two SDDs α and β (Figures 3.5a and 3.5b, respectively) and the result of the conjoin operation between them (Figure 3.5c).*

polynomial-time *conjoin* operation, whereas the *strong determinism* property (together with *partitioning*) enables one to perform the *disjoin* binary Boolean operation in polynomial time, retrieving a new SDD that represents the disjunction of two functions represented by two SDDs (Adnan Darwiche, 2011). Furthermore, SDDs are also capable of performing the *negation* operation in polynomial time, which enables their use in a wide range of applications that OBDDs are used for (Adnan Darwiche, 2011; A. Darwiche and P. Marquis, 2002).

On the other hand, *compressed* and *trimmed* SDDs ensure the *canonicity* property, similar to OBDDs.

As stated at the start of this section, SDDs are a proper subset of SDNNF, which are a proper subset of DNNF (Pipatsrisawat and Adnan Darwiche, 2008). Another interesting relation is that OBDDs are a subset of SDDs, where the corresponding V-trees are *right-linear*, i.e., each left child is a leaf (Adnan Darwiche, 2011). Both of these relations together render the following hierarchy of succinctness:

$$\text{OBDD} \leq \text{SDD} < \text{d-DNNF}, \tag{3.1}$$

where all of these languages are capable of efficiently performing *model counting* inferences (A. Darwiche and P. Marquis, 2002); whereas d-DNNF are only capable of performing *conjunction* and *disjunction* operations if $P = NP$ (A. Darwiche and P. Marquis, 2002).

Chapter 4

Literature Review

The problem of efficiently computing probabilistic inferences over a propositional ASP theory is of great importance for Neuro-Symbolic systems. This is due to the inherent complexity of computing Weighted Model Counting (WMC) queries and the necessity of iterating over Answer Sets (Geh *et al.*, 2024; Z. Yang *et al.*, 2023; Eiter, Hecher, *et al.*, 2024). Furthermore, by representing the output of Neural Networks as *Annotated Disjunctions* (Geh *et al.*, 2024; Z. Yang *et al.*, 2023), one can integrate domain knowledge through ASP rules with the representational capabilities of Neural Networks for tasks such as *Computer Vision* and *Natural Language Processing*.

This combination of logic and neural networks is not new, as it has been explored in the literature for a variety of applications, such as *Inductive Logic Programming* (Avila Garcez and Zaverucha, 1999). Consequently, many techniques have been developed to perform inference over PLPs more efficiently, such as *lifted inference* (Van den Broeck *et al.*, 2021; Totis, Davis, *et al.*, 2023), solution enumeration by optimality (Pajunen and Janhunen, 2021; Manhaeve, Marra, *et al.*, 2021), and Knowledge Compilation.

With this in mind, we present in this chapter a review of the current literature on PLP Knowledge Compilation. We first focus on the ProbLog system and subsequently present a new approach for PASP, which reduces the problem of computing WMC queries to an extension of the Algebraic Model Counting (AMC) problem (Kimmig *et al.*, 2017).

4.1 Clark's Completion

Before delving into the details of the Knowledge Compilation methods in the literature, it is important to note that many top-down compilation methods work by translating a PLP program into a CNF formula, which is then compiled into a circuit. This final step is usually performed by a knowledge compiler, such as c2d or d-Sharp, both of which are based on the sd-DNNF class of circuits (Adnan Darwiche *et al.*, 2004; Muise *et al.*, 2012). For now, we will pause this discussion about knowledge compilers and begin by describing the translation of a PLP into a CNF.

To perform such a translation of a program into a CNF formula, the most common

approach is to use *Clark's Completion*, one of the main methods for attributing meaning to negation in logic, similar to the Stable Models semantics presented in Chapter 2. This approach differs from the Stable Models semantics in that the latter is based on the idea of selecting some models of the program to define its semantics, whereas the former focuses on translating programs into a first-order theory, referred to as a *completion* of the program (Cozman and Mauá, 2017).

Definition 4.1.1 (Clark's Completion)**.** Given a PLP program P, the Clark's completion of P is defined as the following propositional formula:

$$\textsc{Clark}(P) = \bigwedge_{a \in \mathcal{A}(P)} \left[a \iff \bigvee_{r \in \mathcal{R}(P,a)} \bigwedge_{b \in \textsc{body}(r)} b \right], \tag{4.1}$$

where $\mathcal{A}(P)$ is the set of propositional atoms that appear in P, $\mathcal{R}(P, a)$ is the set of rules in P that have a as the head, and $\textsc{body}(r)$ is the set of literals in the body of rule r.

This method can be transformed into a CNF formula, often with the addition of auxiliary variables aux_r for each rule r in a program P. The resulting CNF is the conjunction of the clauses described by Equations (4.2) and (4.3):

$$\left[\bigwedge_{a \in \mathcal{A}(P)} \neg a \vee \left(\bigvee_{r \in \mathcal{R}(P,a)} aux_r \right) \right] \wedge \left[\bigwedge_{a \in \mathcal{A}(P)} \bigwedge_{r \in \mathcal{R}(P,a)} a \vee \neg aux_r \right] \tag{4.2}$$

$$\bigwedge_{r \in P} aux_r \wedge \left[\bigwedge_{r \in P} \bigvee_{b \in \textsc{body}(r)} \neg b \right] \wedge \left[\bigwedge_{r \in P} \bigvee_{b \in \textsc{body}(r)} b \vee \neg aux_r \right] \tag{4.3}$$

It is important to note that, in order to control the number of auxiliary variables created, a Treewidth-Aware method was developed for Clark's completion (Fandinno and Hecher, 2023). Together with another Treewidth-Aware algorithm for (positive) cycle-breaking (Eiter, Hecher, *et al.*, 2021), these two methods are able to reduce the number of auxiliary variables created when translating a program into a CNF formula (Eiter, Hecher, *et al.*, 2024; Eiter, Hecher, *et al.*, 2021).

4.2 ProbLog Compilation

The Problog is one of the most well-known Probabilistic Logic Programming systems in the literature and is based on the widely recognized Prolog language. One of its main characteristics, and the most significant difference in relation to PASP, is that it follows the notion of *stratified* programs, where the program is not permitted to have *negative* cycles in the dependency graph.

With nearly two decades of research (De Raedt *et al.*, 2007), this PLP system has more than one version (De Raedt *et al.*, 2007; Fierens *et al.*, 2015) and several extensions (Totis, De Raedt, *et al.*, 2023; Manhaeve, Dumancic, *et al.*, 2018). Thus, we focus on describing the second version of Problog, two of its compilation techniques, and one of its most recent extensions.

4.2.1 ProbLog 2

The main contribution of the second version of the ProbLog system was the development of different KC algorithms for exact and approximate inference. Since the approximate inference relies on Monte Carlo Markov Chain methods, we focus on the exact ones, which are based on solving the WMC using Knowledge Compilation.

Among the two algorithms proposed, there were: *top-down* and *bottom-up* compilation of CNFs, using d-DNNFs and BDDs, respectively. Both were based on the following algorithm:

1. First, the input program is parsed and grounded;
2. Then, the grounded program is converted into a CNF;
3. Finally, a *knowledge compiler* is used to compile the CNF into a circuit.

The main difference between the two algorithms was that the former utilized the C2D compiler, which is based on sd-DNNFs, while the latter relied on BDDs, hence its capability of performing bottom-up compilation of a CNF formula through compositions of the $apply$ operator.

4.2.2 Bottom-Up Compilation

To avoid the creation of auxiliary variables through Clark's completion, the authors proposed a bottom-up algorithm for compiling a PLP program (Vlasselaer, Renkens, *et al.*, 2014), where one iteratively compiles the formula representing (4.1) by conjoining the circuits representing each *if and only if*.

As we will see in more detail in Chapter 5, by compiling all bodies of rules that share the same head and *disjoining* their respective circuits, the authors are able to avoid the creation of the aux_i variables in Equations (4.2) and (4.3).

This is a similar approach to the work done in Problog's second version, where bottom-up compilation performed by BDDs is achieved by *disjoining* and *conjoining* atoms. The main difference between these two methods is that bottom-up compilation is able to skip the translation to a CNF formula and is performed using SDDs, a more succinct representation of circuits than BDDs (as seen in (3.1)).

4.2.3 T_P Compilation

An advancement in the compilation of PLP for stratified programs is the T_P compilation, proposed by (Vlasselaer, Van den Broeck, *et al.*, 2016), where program compilation is performed iteratively rather than in its entirety.

When compiling a program P using the algorithm described in the previous subsection, one would need to ground the program and apply a cycle-breaking algorithm to compile the program in its entirety, rule by rule. The T_P compilation, however, relaxes this constraint by allowing partial compilation of a program or enabling exact compilation when a fixed point of the iterative process is reached. This approximate compilation has a wide range of applications in cases where an exact answer to the WMC task is not necessary, but a

quick computation is beneficial (within an acceptable error margin, which the algorithm can control through lower and upper bounds).

The core idea of this algorithm lies in a generalization of the previously described T_P operator in Chapter 2, called the T_{C_P} operator, which is defined as follows:

Definition 4.2.1 (Parameterized Interpretation)**.** Let P be a *ground* PLP with probabilistic facts $\mathcal{F}$ and atoms $\mathcal{A}$. We say that I is a parameterized interpretation of P if $I = \{(a, \lambda_a) \mid a \in \mathcal{A}\}$, where λ_a is a propositional formula over $\mathcal{F}$ expressing in which interpretations a is true.

Definition 4.2.2 (T_{C_P} Operator)**.** Let P be a *ground* definite Probabilistic Logic Programming with probabilistic facts F, atoms $\mathcal{A}$, and a parameterized interpretation $I = \{(a, \lambda_a)\}$. Then, the T_{C_P} operator is of the form $T_{C_P}(P) = \{(a, \lambda'_a) \mid a \in \mathcal{A}\}$, where

$$\lambda'_a = \begin{cases} a & \text{if } a \text{ is a probabilistic fact} \\ \bigvee_{r \in \mathcal{R}_a} \bigwedge_{b \in \mathcal{B}_r} \lambda_b & \text{otherwise} \end{cases}$$

$\mathcal{R}_a$ is the set of rules of P where a is the head, and $\mathcal{B}_r$ is the set of atoms in the body of a rule r.

After defining the T_{C_P} operator above, the authors generalize it for Normal Logic Programs and describe a simple algorithm based on the idea of computing, for each iteration of the algorithm, the operator T_{C_P} over one specific atom a, only considering the rules where a is the head. Hence, for each iteration, only the formula for the atom a is updated, and the algorithm stops when a fixed point is reached.

One way to represent these formulas in the interpretation I, which is iteratively updated, is through the use of PSDDs, due to their ability to perform Boolean operations efficiently (Adnan Darwiche, 2011; Kisa *et al.*, 2014). Therefore, one could replace the propositional formulas λ_a with PSDDs to perform a tractable computation of the T_P compilation.

4.2.4 smProblog Compilation

Although ProbLog (De Raedt *et al.*, 2007; Fierens *et al.*, 2015) is a well-known **stratified** Probabilistic Logic Programming system, it was recently extended to **non-stratified** programs under the Stable Model semantics through the development of smProblog (Totis, De Raedt, *et al.*, 2023). With this novel approach, the authors consider both the *Stable Model* and MaxEnt semantics (closely tied to the problem this work aims to address) and propose an algorithm for performing both Knowledge Compilation and inference over these extended Problog programs.

The need for a novel algorithm arises from the conceptual difference between the evaluation of the $SUCC$ and $SUCC^{\text{Max-Ent}}$ queries, which we define as follows. The $SUCC$ problem for a query q and a propositional theory T is defined as follows (Kiesel, Totis, *et al.*, 2022):

$$SUCC(q) = \sum_{I \in \mathcal{M}(T|q)} \mathbb{P}(I) = \sum_{I \in \mathcal{M}(T|q)} \prod_{i \in I} \alpha(i),$$

where $\mathcal{M}(T|q)$ is the set of models of T that satisfy q, and $P(I)$ is the probability of the models I where q *succeeds*. Thus, the $SUCC$ problem describes the sum of the probabilities of the models where the query q succeeds.

On the other hand, when considering probabilistic inference with *Stable Models* and the MaxEnt semantics, the $SUCC$ problem cannot be modeled only by considering the models of a given theory, but rather the *Stable Models* of the program. Therefore, the $SUCC^{\text{Max-Ent}}$ problem is defined as follows (Kiesel, Totis, *et al.*, 2022):

$$SUCC^{\text{Max-Ent}}(q) = \sum_{f \in \mathcal{A}(F)} \sum_{I \in \mathcal{M}(T|f)} \frac{|\mathcal{M}(T|f,q)|}{|M(T|f)|} \prod_{i \in I} \alpha(i),$$

where F corresponds to the set of probabilistic facts of the program.

To address this normalization by the number of *Stable Models*, characteristic of the MaxEnt semantics, the authors developed a normalization step after performing the Knowledge Compilation process. In more detail, the authors used a *top-down* compilation approach, similar to the ProbLog 2 system, where the program is translated into a CNF and then compiled through the use of a knowledge compiler (in this case, d-Sharp). They then perform a normalization step by computing the number of *Stable Models* per total choice.

This normalization operation is particularly costly because it needs to replace conjunctions and disjunctions of the circuit with Cartesian Products and Unions to enumerate all the possible *Stable Models* of the program and group them by the total choices. Furthermore, both the Cartesian Product and the Union operations are known to exhibit a higher computational cost than the former (which can be computed in linear time), as they entail set manipulations, which are more costly than manipulations of Boolean values. The Cartesian Product is especially costly due to the need to compute the product of each element of the sets involved in the operation.

4.3 Second-Level Algebraic Model Counting

To address this probabilistic inference problem via Knowledge Compilation in the context of PASP, the current state-of-the-art approach consists of reducing a PASP inference task to an instance of the Second Level Algebraic Model Counting (2AMC) problem (B. Wang *et al.*, 2025; Kiesel, Totis, *et al.*, 2022).

Formally, the AMC problem is a generalization of different (counting) logic problems, such as SAT, MAX-SAT, or WMC, where one describes a *semiring* over the set of models of a propositional theory, and the task is to compute the value of a given formula under this semiring. For example, the *Tropical Semiring* is a well-known semiring that generalizes the MAX-SAT as an AMC instance.

The 2AMC acts as an extension of the AMC, where a solution of an *inner* AMC is fed into an *outer* AMC problem, which is closely related to the notion of inference in PASP, as one needs to compute an inference over *Stable Models* of *total choices*.

4.3.1 Algebraic Answer Set Counting

Under this general framework for Algebraic Answer Set Counting (AASC) (instances of the 2AMC applied to PASP inference), inference can be tractably performed when imposing two properties: X-*firstness* and X-*determinism* (which coincide in SDDs, in a property called X-constraindness (Oztok *et al.*, 2016)) (B. Wang *et al.*, 2025), defined as follows:

Definition 4.3.1 (X-firstness)**.** Let σ be a NNF n on variables V partitioned into sets X and Y. Then, we say that an internal node n_i of Σ is *pure* if its variables are present exclusively in X or Y, $\text{vars}(n_i) \subseteq X$ or $\text{vars}(n_i) \subseteq Y$; and are called *mixed* otherwise. Furthermore, we say that n is X-*first* if, for each of its *and-nodes* n_i, all of its children are pure; or exactly one child is mixed and the remaining children n_j are pure with $\text{vars}(n_j) \subseteq X$.

Definition 4.3.2 (X-determinism)**.** Let σ be a NNF n on variables V partitioned into sets X and Y. Then, we say that an OR-node n_i of Σ is X-deterministic if, for any partial assignment a of variables in X, we have that at most one of the children of n_i has a nonzero output. If all of the OR-nodes in Σ follow this property, then we say that Σ is X-deterministic.

With such properties, (Eiter, Hecher, *et al.*, 2024) combine the 2AMC top-down compilation via sd-DNNFs with treewidth-aware methods, resulting in the following algorithm:

1. Given a propositional program, a (positive) cycle-breaking algorithm is applied, which has an upper bound based on the treewidth (Eiter, Hecher, *et al.*, 2021).
2. Next, the cycle-free program is converted into a CNF formula via another treewidth-based algorithm, this time based on Clark's completion (Fandinno and Hecher, 2023).
3. Finally, the CNF formula is compiled through the use of a knowledge compiler (c2d), and inference can be performed directly via model counting, which involves an inner model counting over answer sets and an outer weighted model counting over total choices.

Chapter 5

Contribution and Discussion

While stratified PLPs systems, such as Problog, have been explored since the early 2000s, research on PASP is a more recent focus of study, with early works outside Sato's probabilistic semantics dating back to the end of the first decade of the 21st century.

This historical context and the hardness of probabilistic inference under the Stable Models semantics are the main reasons why more efficient KC algorithms for compiling PASP programs require development.

For the reasons mentioned above, the contribution that this work aims to propose is the compilation of PASP programs under *Credal* and MaxEnt semantics that avoids introducing auxiliary variables during compilation, taking inspiration from different works, as the bottom-up compilation (Vlasselaer, Renkens, *et al.*, 2014) of Problog, the 2AMC framework for probabilistic inference (Kiesel, Totis, *et al.*, 2022; B. Wang *et al.*, 2025) and efficient encoding of Pseudo-Boolean constraints (which are the case of cardinality constraints in PASP) (Abío *et al.*, 2012). Furthermore, this work also proposes a new compilation paradigm for PLPs, called Non-Incremental compilation, where one exploits the structure of the program to identify intertwined subsets of rules that, when compiled independently, can be combined to form a more efficient compilation, by taking inspiration from recent works on bottom-up compilation (De Colnet, 2023).

5.1 Formalization of the Proposal

Previous studies in the field of KC suggest that, even though SDDs are indeed less succinct than d-DNNFs, the possibility of performing Boolean operations in polynomial time is more than a sufficient trade-off in flexibility. This is due to the fact that the *Apply* operator in SDDs enables direct compilation of program rules, avoiding the introduction of auxiliary variables that may be inherent to the Clark's completion procedure when using top-down compilers (Vlasselaer, Renkens, *et al.*, 2014).

In this sense, this work proposes bottom-up compilation algorithms for compiling PASP programs into SDDs, in order to perform both *Credal* and MaxEnt inference, described in Chapter 2. This bottom-up compilation is described in more depth in the following

sections, where structural constraints of the 2AMC problems are discussed.

Moreover, this work also claims that naive compilation of circuits through the use of trivial initialization of the *V-tree* may not be sufficient to achieve satisfactory results, as preliminary results suggest. Therefore, the adaptation of a heuristic for initialization of X-constrained V-tree is of great importance for the Answer Set Counting task, which is the task of performing both *Credal* and MaxEnt inference in PASP. Although a heuristic may seem quite simple, it is still relevant, since most state-of-the-art heuristics are focused on CNF compilation; and do not leverage the structure of Probabilistic Logic Programs.

Finally, to further enhance the compilation process, we propose an algorithm that identifies disjoint components of the program (subsets of the program that do not share any variables) in order to leverage theoretical properties that ensure that the compilation process is linearly bounded by the size of the intermediary representations of each one of these components. This approach allows for more efficient compilation by focusing on smaller, independent subproblems, which can be solved more quickly and with less memory overhead. If the program cannot be decomposed into disjoint components, we reduce the problem to a combinatorial optimization problem that can be solved efficiently, in order to find a compromise that still enables us to leverage the theoretical results of (De Colnet, 2023). This process is called Non-Incremental (bottom-up) compilation and the algorithm described in (De Colnet, 2023) assumed that the decomposition of the formula was given, so this is the first work to describe an algorithm that can decompose a program into disjoint components, in such a way that the algorithm proposed in this work could be applied to any other PLP; or even other target languages, such as And-Sum circuits (Onaka *et al.*, 2025), because they are also structured decomposable and share the *Apply* operator with SDDs.

5.1.1 Class of Considered Programs

This work focuses on ground PASP programs. This is a reasonable consideration due to recent advancements in *grounding* techniques for Answer Set Programming. Thus, we do not consider the task of performing grounding in this work, since it is a well-studied and optimized task in the literature (Gebser, Kaminski, Kaufmann, and Schaub, 2014; Gebser, Kaminski, Kaufmann, Ostrowski, *et al.*, 2016; Kaminski *et al.*, 2017), so we assume that the input program is already grounded.

There are also different considerations that could be made, such as assuming that the input program is *tight* (i.e. does not contain positive cycles in the program dependency graph), because every *non-tight* program can be transformed into a tight one through the application of a technique called *cycle-breaking*, which is also a well studied topic in the literature (Eiter, Hecher, *et al.*, 2024). Nevertheless, one interesting property of the bottom-up compilation that has not been explored in depth in the literature is the fact that it can leverage *Loop Formulas* (Lee and Lifschitz, 2003) to avoid the need of using cycle-breaking to cope with positive cycles; and we will explore this property in more detail in the following sections.

Finally, another consideration that is usually assumed when working with PLPs that we will not assume is that the input program does have *annotated disjunctions*, i.e. usually

it is assumed that the input program only has *probabilistic facts*. This assumption is usually associated with the idea that the annotated disjunctions present in the original program will undergo a transformation into probabilistic facts (Shterionov *et al.*, 2015). This transformation will not be covered in depth in this work, since we will show how to cope with annotated disjunctions in a similar way to cardinality constraints, as one can imagine annotated disjunction as a cardinality constraint that imposes a *exactly one* condition.

5.2 Motivation

As a first consideration before approaching the PASP compilation topic in more detail, it is important to note that an enumerative approach to this problem can be consistently better than a KC approach, but only if the number of instances of the 2AMC problem to be solved is small (Kiesel, Totis, *et al.*, 2022). More specifically, when considering few probabilistic facts, the overhead of compiling the program into a circuit representing its distribution is not amortized over time, which can lead to a slower computation time when compared to an enumerative approach. On the other hand, as one can see by the work of (Kiesel, Totis, *et al.*, 2022), the KC approach outperforms an enumerative approach when the number of instances is significant.

This difference in performance is largely due to optimizations of enumerative approaches that are not present in KC, such as dynamic programming and caching (Gebser, Kaminski, Kaufmann, and Schaub, 2014; Gebser, Kaminski, Kaufmann, Ostrowski, *et al.*, 2016; Kaminski *et al.*, 2017) and other techniques characteristic of ASP and SAT solvers (Gong and Zhou, 2017; Toda and Soh, 2016).

Since this work revolves around the Neuro-Symbolic topic, which encompasses the integration of Neural Networks with Probabilistic Logic Programming systems through inference and learning algorithms (Geh *et al.*, 2024; Manhaeve, Dumancic, *et al.*, 2018; Z. Li *et al.*, 2023), the KC approach is best suited for the task at hand. The reason for this is that learning algorithms are usually trained on large datasets and with a high number of iterations, which would make the overhead of computing the 2AMC problem through a KC approach amortized over time.

One great example to support this claim is the *NeurASP* system, which uses an enumerative approach to compute the *Stable Models* of a program. As the authors of the system stated, although this naive approach of enumerating is highly competitive for small instances of programs (few probabilistic facts), it is not scalable (Z. Yang *et al.*, 2023).

5.3 Bottom-Up Compilation of PASP Programs

In this first introduction to the compilation of PASP with a bottom-up approach, we will describe how this algorithm relates to the procedure of Clark's Completion, with examples of how the compilation of rules that have the same head can be performed in a more efficient way.

Furthermore, we present a naive approach to PASP compilation, in order to hint at how an indiscriminate choice of *V-tree* initialization can lead to a circuit that does not

capture the structure of the program, if other considerations (explained in the following sections) are not taken into account.

5.3.1 Relation to Clark's Completion

In the previous literature review chapter 4, the *Clark's Completion* was described in detail as a procedure to transform a Logic Program into a CNF formula through the introduction of auxiliary variables for heads of rules that appear more than once.

It is important to note that, even though this translation of program to CNF formula is not particularly a difficult task (since it only requires the applications of Equations (4.2) and (4.3), it does not take advantage of the structure of the program. However, an obvious but still a particularly interesting remark is that Logic Programs are not CNF formulas. Since the CNF are a well-studied topic for many applications, using it to encode knowledge can be viewed as a rather straightforward approach, but it comes at the cost of losing the structure of the program and introducing unnecessary variables during compilation.

Finally, we note that we use a slightly different version of Clark's completion, in order to obtain the boolean representation of disjunctive programs, in Equation (5.1). This completion can be seen as the result of applying the *Shifting* technique to the original PASP program (Lee and Lifschitz, 2003).

$$\textsc{Clark}(P) = \bigwedge_{a \in \mathcal{A}(P)} \left[a \iff \bigvee_{r \in \mathcal{R}(P,a)} \bigwedge_{b \in \textsc{body}(r)} b \bigwedge_{a' \in \textsc{head}(r) \setminus \{a\}} \neg a' \right]. \tag{5.1}$$

Note that Shifting can be viewed as a preprocessing step that transforms a disjunctive program of the form Program 5.1 into Program 5.2. Hence, we will consider (from now on) that all input programs to the compiler are already shifted; and, thus, their completion is given by Clark's completion discussed in 4.

Program 5.1 Example of a disjunctive program.

```
1    a; b :- c, d.
```

Program 5.2 Example of a disjunctive program after shifting.

```
1    a :- c, d, not b.
2    b :- c, d, not a.
```

5.3.2 Bottom-Up Compilation

The bottom-up compilation is rather straightforward, leveraging the *Apply* operator to compile a succinct representation of the program by following Equation (5.1). The core idea of this approach consists in checking if an atom appears as the head of multiple rules. If it is not the case, then the compilation of the rule consists in applying an *if*

and only if operation between the head atom and the conjunction of the literals in the body of this unique rule. This process can be seen in program 5.3, where the resulting bottom-compilation is shown in Fig. 5.1.

Program 5.3 Example of a program with an atom that appears as head only in one rule.

```
1    turing :- godel, church.
```

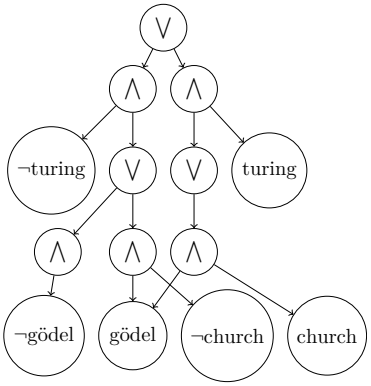


Figure 5.1: *Compilation of Program 5.3 to an SDD with balanced V-tree (with variable order as they appear in the program).*

In greater detail, the process of compiling the program 5.3 consists of the following steps:

(i) Initializing a balanced V-tree, that the SDD will respect;

(ii) Compiling each atom into an SDD representing the observation of the atom itself;

(iii) Compiling the body of the rule into an SDD α by performing a conjunction of the literals in the body: $godel \wedge church$ (where $godel$ and $church$ here represent their respective SDD and $\wedge$ the conjoin operation);

(iv) Performing an *if and only if* operation between the previously obtained SDD $alpha$, from the conjoin operation, and the SDD of the head of the rule: $turing \iff \alpha$ (where $turing$ represents the SDD of the atom *turing* and $\iff$ the Boolean *if and only if* operation).

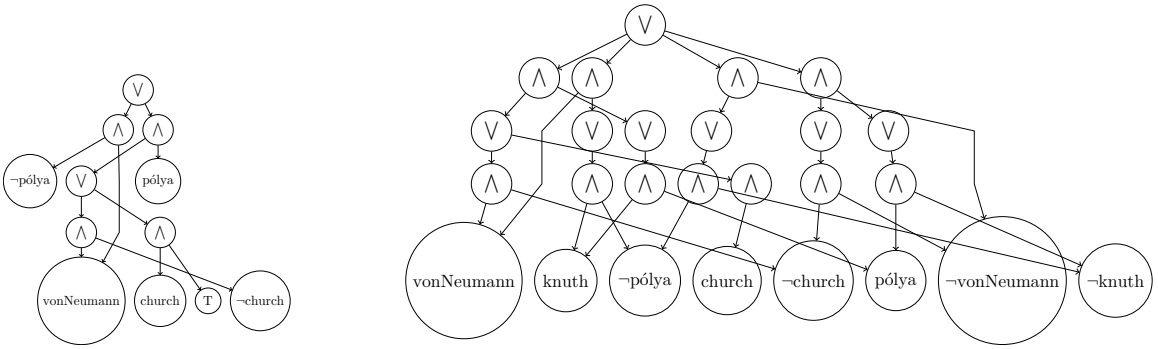
If we have a program where the same atom appears as the head of multiple rules, the process is slightly more complex, but still intuitive. Differently from the CNF approach, where an auxiliary variable would be introduced, the bottom-up proceeds by compiling each body of the rules that have the same head and performing a disjunction between the resulting circuits. This can be seen as computing the $\vee$ operator from Equation (4.1).

Consider the following program 5.4, where the atom $knuth$ appears as the head of two rules. To compile such a program we first compile the bodies of the rules, with the same approach as before, and then perform a disjunction between all different bodies, resulting in a single circuit. Then, we apply the *if and only if* operation between the head of rules and the circuit of the bodies, as we see in Figure 5.2.

Therefore, the single difference between the compilation of a rule with a unique head and the compilation of a set of rules with the same head is the disjunction operation between the bodies of the rules. By applying this approach, we can avoid the introduction of auxiliary variables, by utilizing the *apply* operation of SDDs.

Program 5.4 Example of a program with an atom that appears as head in multiple rules.

```
1    knuth :- polya, church.
2    knuth :- vonNeumann.
```



(a) *An SDD representing* $polya \wedge church$.

(b) *The result of the iff operation between* $knuth \iff (polya \wedge church) \vee vonNeumann$.

Figure 5.2: *Compilation of Program 5.4 into an SDD with balanced V-tree (with variable order as they appear in the program).*

Summarizing, the bottom-up compilation of PASP programs is defined as the following algorithm. Let P be a PLP and consider a predefined *V-tree* initialization for the SDD iterative compilation defined as follows:

(i) Compile each atom of the propositional program P into an SDD representing the observation of the atom itself;

(ii) For each atom a that appears as a head of the program P:

  (a) For each rule r that has a as the head, compile the conjunction (by applying the *conjoin* operator) of each literal in the body of r (if an atom appears negated in the body, apply the $\neg$ operator to its respective SDD initialized in the first step);

  (b) Perform a disjunction (by applying the *disjoin* operator) between the resulting SDDs from the previous step.

(iii) Apply an *if and only if* operation between the SDD of the head of the rules and the resulting circuit from the disjunction of the bodies, from the previous step.

(iv) Conjoin the resulting circuits of the previous step to obtain the circuit representing the program P.

By analyzing the algorithm above, one can see that it also correctly handles integrity constraints, as these can be seen as an immediate consequence of the Clark's completion

(Equation (4.1)). By considering the empty head of integrity constraints as $false$; and solely following the steps described in the bottom-up algorithm, we are able to correctly compile the integrity constraints in a similar manner as other normal rules.

Now, consider the Program5.5, where we combine rules from Programs 5.3 and 5.4. In this case, the compilation of the program, followed by compiling first the rule where $turing$ is the head and then the rules where $knuth$ is the head, would result in the circuit described by Figure 5.3 (when considering a balanced *V-tree*, with variables appearing in the order as they are presented in the program).

Program 5.5 Example of a program with rules both from Programs 5.3 and 5.4.

```
1    turing :- godel, church.
2    knuth :- polya, church.
3    knuth :- vonNeumann.
```

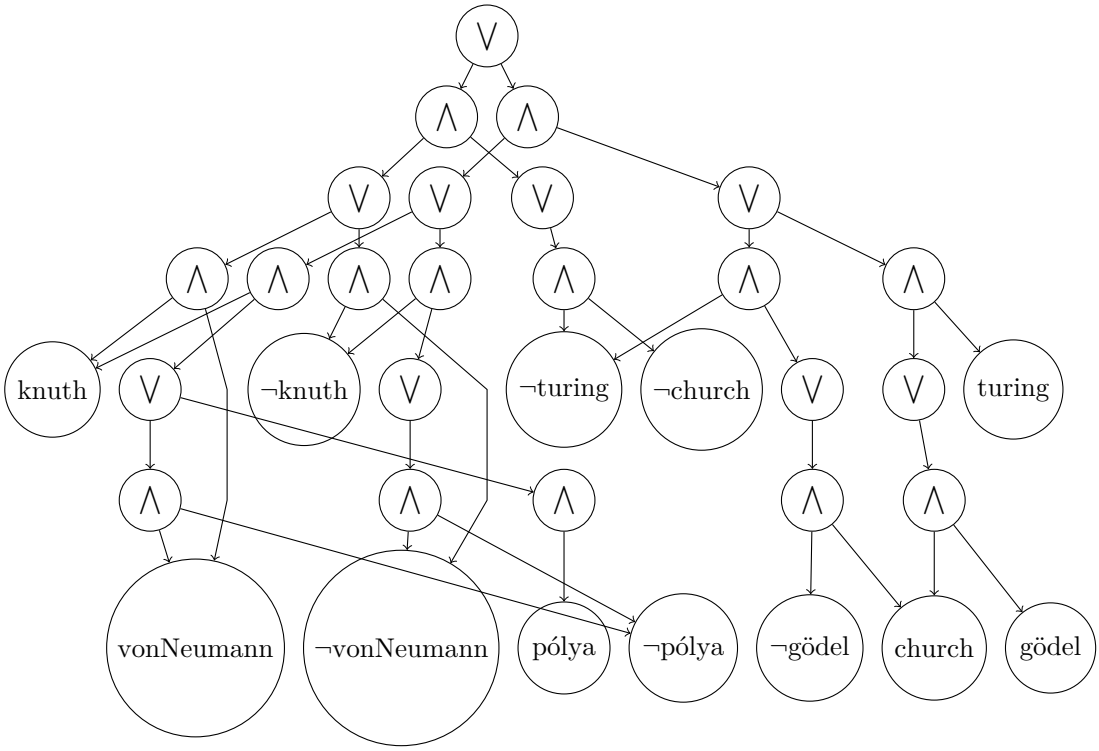


Figure 5.3: *Compilation of the Program 5.5, obtained by application of the conjoin SDD operator between the circuits of Figures 5.1 and 5.2b. Moreover, the V-tree of the circuit is balanced, and the variables appear in the order they are presented in Program 5.5.*

A final, but extremely relevant, consideration is that, until now, the bottom-up algorithm described only correctly represents acyclic programs (programs without any positive cycles). This is enough if one applies cycle-breaking as a pre-processing technique in order to break positive cycles in the dependency graph of the program. On the other hand, due to the flexibility of the *Apply* operator, it is easy to directly compile *Loop Formulas* without introducing auxiliary variables (as it is usually done to avoid costly translations to CNF).

More specifically, for each loop L in the positive dependency graph of a program P, we have that its completion is given by:

$$\text{Loop}(L) \equiv \bigvee_{l \in L} l \implies \bigwedge_{r \in R(L)} \bigwedge_{b \in body(r)} b \bigwedge_{a \in head(r) a \notin L} \neg a, \tag{5.2}$$

where $R(L)$ is the set of rules that share a head with an atom inside the loop L; and that do not share any body atoms with the atoms in L (Lee and Lifschitz, 2003). This consideration concludes the bottom-up compilation of disjunctive programs, thus leaving the compilation of cardinality constraints as the only remaining challenge to address ASP compilation.

5.3.3 Constraint Compilation

Another key challenge in PASP compilation is the representation of cardinality constraints. While there are well-studied methods in the literature capable of encoding such constraints more succinctly, such as Sequential Counters (Marques-Silva and Lynce, 2007) and Totalizers (Bailleux and Boufkhad, 2003), they introduce auxiliary variables, which our approach aims to mitigate. Thus, we propose a method to compile an upper cardinality constraint by calling $\text{Upper}(A, 0, u)$. This is an adaptation of (Abío *et al.*, 2012) that leverages bottom-up compilation to avoid introducing auxiliary variables:

$$\text{Upper}(A, c, t) = \begin{cases} \text{false}, & \text{if } c > t; \\ \text{true}, & \text{if } |A| \leq t - c; \\ \text{ITE}(a, \text{Upper}(A \setminus \{a\}, c+1, t), \text{Upper}(A \setminus \{a\}, c, t)), & \text{otherwise.} \end{cases} \tag{5.3}$$

where A represents the set of atoms inside the cardinality constraint, c is a counter of the current number of *true* atoms at the time of the function call, and u is the upper value of the constraint. Even though Equation (5.3) only encodes an upper cardinality constraint, it is fairly straightforward to generalize it for a lower bound cardinality constraint:

$$\text{Lower}(A, c, t) = \begin{cases} \text{true}, & \text{if } c \geq t; \\ \text{false}, & \text{if } |A| + c < t \\ \text{ITE}(a, \text{Lower}(A \setminus \{a\}, c+1, t), \text{Lower}(A \setminus \{a\}, c, t)), & \text{otherwise.} \end{cases} \tag{5.4}$$

With the above Equations (5.4) and (5.3), we can encode any cardinality constraint efficiently, whether it is an upper bound, a lower bound, or a combination of both. On the other hand, "exactly k" constraints are of special importance, particularly because annotated disjunctions can be seen as "exactly one" cardinality constraints. Usually, annotated disjunctions are encoded using a method proposed by (Shterionov *et al.*, 2015), which essentially encodes a Sequential Counter (Marques-Silva and Lynce, 2007) by introducing auxiliary variables. However, by exploiting simple symmetries, we can combine both upper and lower bounds into a single function that computes an "exactly k" constraint directly:

$$\text{Exactly}(A, c, t) = \begin{cases} \text{false}, & \text{if } c > t \\ \bigwedge_{a \in A} \neg a, & \text{if } c = t \\ \text{false}, & \text{if } |A| + c < t \\ \text{ITE}(a, \text{Exactly}(A \setminus \{a\}, c+1, t), \text{Exactly}(A \setminus \{a\}, c, t), \text{false}) & \end{cases} \tag{5.5}$$

Note that all of the above Equations (5.3), (5.4), and (5.5) can be easily generalized for the case of aggregators, when each atom is associated with a weight, since they are quite flexible to encode arbitrary pseudo-Boolean constraints.

In order to combine the compilation of the cardinality constraints into our bottom-up approach, we just conjoin the results of the bottom-up compilation with each circuit representing a cardinality constraint.

5.3.4 Consideration About Succinctness

Note that this approach is indeed more efficient than the one that uses the Clark's completion, but only when considering the number of variables in the circuit. The number of nodes (or edges) produced by the SDD compiler is not necessarily better than the one produced by the CNF compiler that uses d-DNNFs, because the latter are able to compile formulas more succinctly than the former Equation (3.1). Therefore, programs with a great number of rules with the same head may not benefit greatly from this approach, but the converse is also true, which means that the best approach largely depends on the structure of the program.

5.3.5 Limitations of Unconstrained Compilation

As stated in the literature review 4, this algorithm is based on the work of (Vlasselaer, Renkens, *et al.*, 2014), in which the authors only considered the compilation of *stratified* programs, a task that requires fewer constraints than the one proposed in this work, since PASP programs are intrinsically related to the notion of Answer Sets, as the name suggests.

This semantic difference between *stratified* and *non-stratified* (PASP) programs is the main reason for the development of this work, since the techniques proposed for the former cannot be directly applied to the latter without caution. The main reason for this is that the *stratified* does not encompass the *Stable Model* semantic, which is the main focus of the PASP programs. When taking into account the *Stable Model* semantics, we will see (in the following sections) that a naive compilation of a *non-stratified* program. For example, Program 5.6, with inference performed by a circuit produced by the bottom-up algorithm, can lead to erroneous results.

For instance, Figure 5.4 represents the results of two different initializations of the *V-tree* for the bottom-up algorithm, using the program 5.6, where Figure 5.4a represents a right-linear initialization and Figure 5.4b a left-linear initialization. Note that the results are different in structure, number of nodes and edges, but both of them retrieve the exact same number for Model Counting (MC). On the other hand, this does not imply that both

Program 5.6 Example of a *non-stratified* PASP program, representing a Hidden Markov Model.

```
1    0.5::b(1).
2    0.5::a(1).
3    x(1) :- a(1).
4    y(1,0); y(1,1) :- x(1).
5    y(1,0) :- b(1), not x(1).
6    y(1,1) :- not b(1), not x(1).
```

of them are able to correctly compute the desired inferences for *Credal* or MaxEnt, since the structure of the circuit does not impose any constraints on the variables, which can lead to erroneous results for WMC tasks, if this is not taken into account (a more detailed explanation of this problem is given in the following sections).

Therefore, we delve into the two main approaches to solve this WMC problem when considering inference KC of PASP programs.

5.3.6 Preliminary Results

In order to elucidate the impact of translating a program into a CNF representing, Figure 5.5 compares how many auxiliary variables are needed to compile a program via its Clark's completion, when one uses top-down compilation. The datasets that are considered here are explained in detail in Section 5.9.

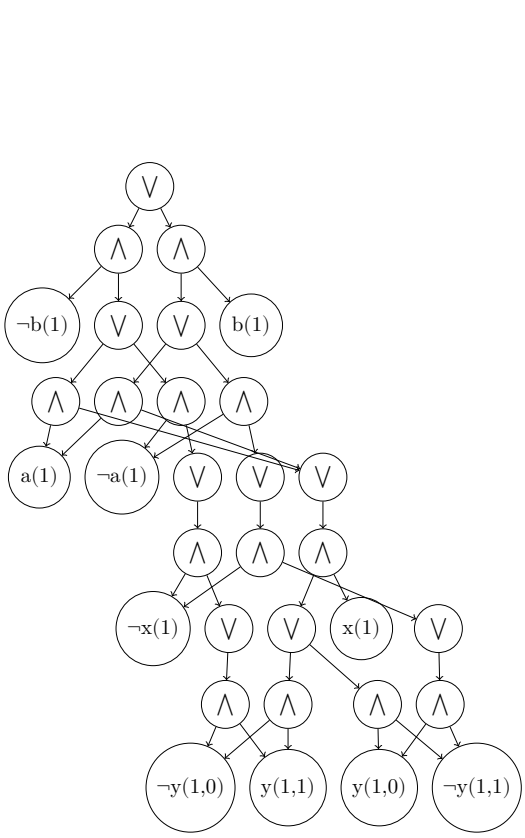
As one can see in Figure 5.5, in almost all datasets (besides IRL), the number of auxiliary variables introduced during Clark's completion significantly increases as the instance size grows. This suggests that the motivation behind this work is a fair one: even though SDDs are proven to be less succinct than sd-DNNFs, the large number of auxiliary variables introduced during Clark's completion suggests that the bottom-up compilation can be a fair alternative to the state-of-the-art PASP compilation.

5.4 PASP as a 2AMC problem

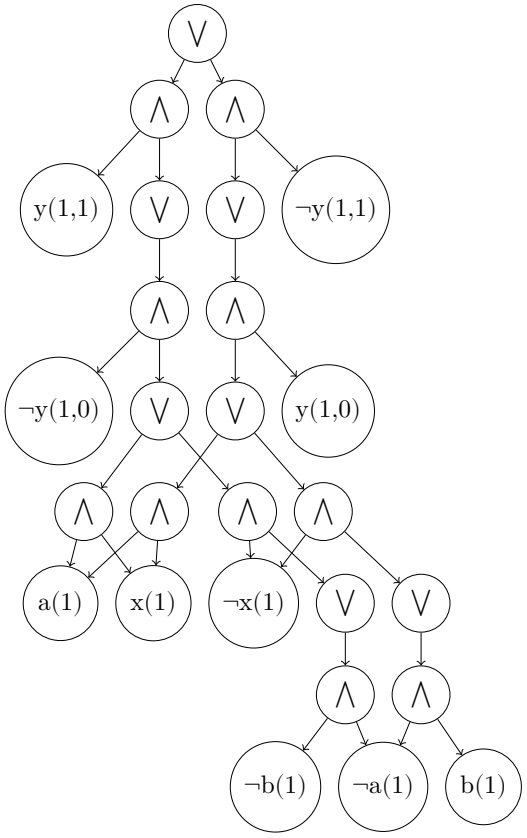
The first approach that we consider is the 2AMC problem. This is a generalization of the AMC problem, superficially described in the literature review 4 and described in more detail in the (Kimmig *et al.*, 2017; B. Wang *et al.*, 2025). We highly recommend the reader not familiarized with the WMC problem in general to read (Kimmig *et al.*, 2017) before continuing this section. In summary, WMC is a framework that generalizes probabilistic inference over propositional formulas with weighted atoms.

5.4.1 Credal Semantics as an Instance of 2AMC

As described in the preliminary Chapter 2, the Credal semantics is deeply related to the notion of *brave* and *cautious reasoning*, in the sense that we want to compute a probability interval $[\underline{\mathbb{P}}(\mathcal{M}), \overline{\mathbb{P}}(\mathcal{M})]$ (Equation (2.1)), for a set of models $\mathcal{M}$, such that



(a) *Bottom-Up compilation of the Program 5.6 following a right-linear V-tree.*



(b) *Bottom-Up compilation of the Program 5.6 following a left-linear V-tree.*

Figure 5.4: *Bottom-Up compilation of the program 5.6 by using bottom-up algorithm, using two different v-tree initializations: right-linear 5.4a and left-linear 5.4b (with variable order as they appear in the program).*

$$\overline{\mathbb{P}}(\mathcal{M}) = \sum_{\theta\in\Theta\,:\,\Gamma(\theta)\cap\mathcal{M}\neq\emptyset} \mathbb{P}(\theta), \qquad \underline{\mathbb{P}}(\mathcal{M}) = \sum_{\theta\in\Theta\,:\,\Gamma(\theta)\subseteq\mathcal{M}} \mathbb{P}(\theta),$$

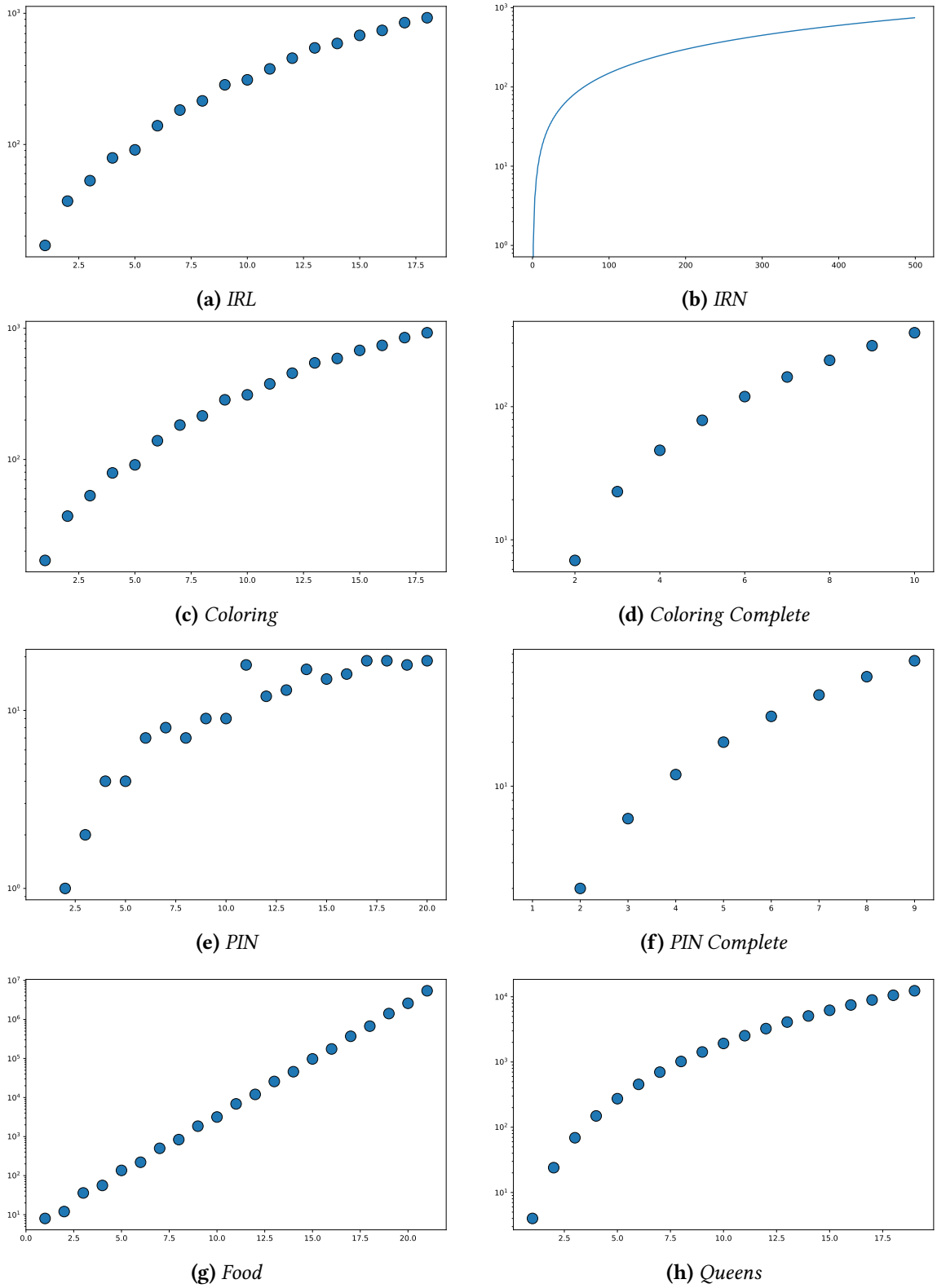
where Θ is the set of all total choices, and $\Gamma(\theta)$ the set of stable models associated with the total choice (Cozman and Mauá, 2017; Cozman and Mauá, 2020).

Although both equations above are not immediately recognizable as an instance of the 2AMC problem, its characterization as:

$$\overline{\mathbb{P}}(\mathcal{M}) = \sum_{\theta\in\Theta}\left[\mathbb{P}(\theta)\cdot\bigwedge_{\gamma\in\Gamma(\Theta)}[\![\gamma\in\mathcal{M}]\!]\right],$$

$$\underline{\mathbb{P}}(\mathcal{M}) = \sum_{\theta\in\Theta}\left[\mathbb{P}(\theta)\cdot\bigvee_{\gamma\in\Gamma(\Theta)}[\![\gamma\in\mathcal{M}]\!]\right].$$

Formally, one can see that the two semirings corresponding to the upper probability interval in a 2AMC problem are:



(a) *IRL*

(b) *IRN*

(c) *Coloring*

(d) *Coloring Complete*

(e) *PIN*

(f) *PIN Complete*

(g) *Food*

(h) *Queens*

Figure 5.5: *Comparison of number of auxiliary variables introduced during Clark's completion across eight different datasets. The x-axis and y-axis represent, respectively, the size of the number of auxiliary variables and the instance size.*

- The inner semiring $\mathcal{S}_I = (\{0, 1\}, \wedge, \wedge, \text{true}, \text{true})$, with labeling function α_I being constant as *true* for all elements of the semiring;
- The outer semiring $\mathcal{S}_O = (\mathbb{R}_{\geq 0}, +, \cdot, 0, 1)$, with labeling function $\alpha_O(o) = \mathbb{P}(o)$ for all elements of the semiring, which corresponds to the probability of a probabilistic fact o.

Furthermore, the weight transformation function can be considered as the identity function, since it corresponds elements with value 0 from the inner AMC to 0 in the outer AMC, and the same is true for an inner AMC with value 1.

A similar approach can be taken for the lower probability, since the main difference between both *upper* and *lower* probabilities concerns the notion of *brave* and *cautious* reasoning, respectively. Moreover, by applying the "tuple trick" (B. Wang *et al.*, 2025), we can create semiring elements capable of representing the computation of both *upper* and *lower* probabilities at the same time, as an instance of the 2AMC problem.

5.4.2 A Correct Property for 2AMC

One of the main results of (B. Wang *et al.*, 2025) regarding the 2AMC problem is that both X-firstness and X-determinism are required to efficiently compute inference in Answer Set Counting tasks (e.g., both MaxEnt and the lower of *Credal* semantics). In the context of SDDs, we have that both properties coincide, rendering a property called *X-constrainedness*, defined as follows:

Definition 5.4.1 (X-constrained)**.** Let V be a *V-tree* and v one of its nodes. We say that v is X-constrained if and only if v appears on the rightmost path of the *V-tree* and X is the set of variables outside v. Moreover, we say that a *V-tree* is X-constrained if and only if it has an X-constrained.

Finally, we say that an SDD is X-constrained if and if it is normalized for an X-constrained *V-tree*. An SDD node is said to be X-constrained if and only if it is normalized for the X-constrained *V-tree* node.

Formally, the notion of *X-constrained* is characteristic of SDDs (more generally, SDNNFs), since its definition is based on the notion of a *V-tree* respecting this order between partitions of variables. In the case of sd-DNNFs this notion is not present, since they do not need to respect the *structured decomposability* property 3. However, one (strong) property that can be imposed on sd-DNNFs that ensures tractability of the 2AMC problem would be the variable decomposition used in its construction. For example, the *c2d* compiler utilized the Shannon Decomposition, with some dynamic reordering of the variables, which renders the resulting sd-DNNF the desired property (Adnan Darwiche *et al.*, 2004).

An important observation is that (Kiesel, Totis, *et al.*, 2022) claims that the *X-firstness* property 4 by itself is indeed sufficient for the tractable computation of the 2AMC problem over sd-DNNFs, which is not true (B. Wang *et al.*, 2025), because such property is neither *necessary* nor *sufficient* for a tractable computation of the 2AMC problem. This mistake was probably induced by the decomposition performed by the *c2d* compiler, which does not impose the *X-firstness*, but rather stronger properties (Adnan Darwiche *et al.*, 2004) when performing variable decompositions.

On the other hand, *X-determinism* is a property that can be by itself *sufficient* for efficient computation of some of the desired inferences, such as the upper of the *Credal* semantics, more commonly known as *MaxCredal*.

5.4.3 2AMC Compilation via SDDs

Note that, since the 2AMC task can be computed in polynomial time through the use of an X-constrained V-trees, by modifying the bottom-up algorithm to take into account this type of structural constraint, we can compile a PASP into a (smooth) SDD that is capable of computing the 2AMC task in polynomial time. Hence, we are capable of performing a bottom-up compilation of a PASP program into a circuit that is capable of computing both *Credal* and MaxEnt queries in polynomial time, by restricting the V-tree used in the bottom-up algorithm to be X-constrained (i.e. the probabilistic variables are on the left of the V-tree; while the logical variables are on the right).

5.4.4 Comments on Other Compilers

The most popular approach to solve the 2AMC via KC in recent works, such as (Eiter, Hecher, *et al.*, 2024; Kiesel, Totis, *et al.*, 2022), encompasses the use of c2d (Adnan Darwiche *et al.*, 2004), d4 (Lagniez and Pierre Marquis, 2017) and SharpSAT (Korhonen and Järvisalo, 2021) as knowledge compilers. Indeed, the performance of these compilers can be rather remarkable when compared to other top-down approaches (Eiter, Hecher, *et al.*, 2024), specially when considering the compilers that do not use d-DNNFs, such as OBDDs-based compilers (such as *CUDD* (Somenzi, 1998), or *BDDC* (Lai, D. Liu, *et al.*, 2017)), or even SDDs-based compilers (for example, *miniC2D*, a variant of *c2d* that compiles formulas into SDDs instead of d-DNNFs) (Lai, Meel, *et al.*, 2022).

More recent work even proposes a new language for representing circuits, Constrained Conjunction and Decision Diagram (CCDD), and gives birth to a new compiler, called *Panini* (Lai, Meel, *et al.*, 2022), which surpasses the performance of the previously mentioned compilers in a variety of tests (Lai, Meel, *et al.*, 2022).

An important note is that the results of (Kiesel, Totis, *et al.*, 2022) describe the ability of sd-DNNFs of computing the 2AMC problem tractably. Although this result immediately implies that SDDs (and consequently, OBDDs) are capable of performing the same task in polynomial time w.r.t the size of the circuit, it is not clear if the same results follow for CCDDs.

Moreover, even though the 2AMC problem can be solved by sd-DNNF, *smoothness* is not the only property that circuits inside the d-DNNF family must have. As mentioned before, the circuit must exhibit a structure alike the *X-constraint* property of SDDs in order to perform such computational tasks tractably. With this problem in mind, *c2d* is the usual choice for solving 2AMC tasks for PASP applications, due to the software's ability ensuring structural constraints over the circuits it produces (a task that is not trivial to ensure for other compilers). On the other hand, both d4 and SharpSAT-TD were adapted by (Eiter, Hecher, *et al.*, 2024) to ensure the necessary structural constraints over the circuits they produce.

This implementation and algorithmic problem (concerning the variable decomposition) are major bottlenecks for the usage of sd-DNNF for compilation; and not only for PASP, but also for CNF formulas in general. As we are going to see in later sections, the possibility of dynamically reordering the variables during a bottom-up compilation is a major advantage of SDDs over sd-DNNFs, and can lead to a more succinct circuit representation of the program (even though the succinctness of the former is in a lower hierarchy than the latter (3.1)). This approach is of special interest when the input of the compilation is not a program, where the structure can be exploited, but a CNF, which is a common input for knowledge compilers.

5.4.5 Preliminary Results

Now, we delve into the first experimental results that indicate how the bottom-up compilation via SDDs can be an interesting alternative to the current state-of-the-art top-down compilation. In Figure 5.6, we can see that even without using any clever V-tree initialization, reordering or other techniques, some of the most complex classes of programs, PIN-Complete and Queens, are leveraged by the bottom-up compilation. This are two classes of programs that are heavily impacted by the increase of auxiliary variables (Figure 5.5). Moreover, the bottom-up compilation had a fair performance on the IRL dataset, largely surpassing SharpSAT-TD and achieving comparable performance w.r.t. both C2D and D4. In the remaining datasets, performance of the bottom-up compilation was poor, but we investigate, in the following sections, improvements to the bottom-up compilation process that will flip this situation, making it more competitive with the current state-of-the-art.

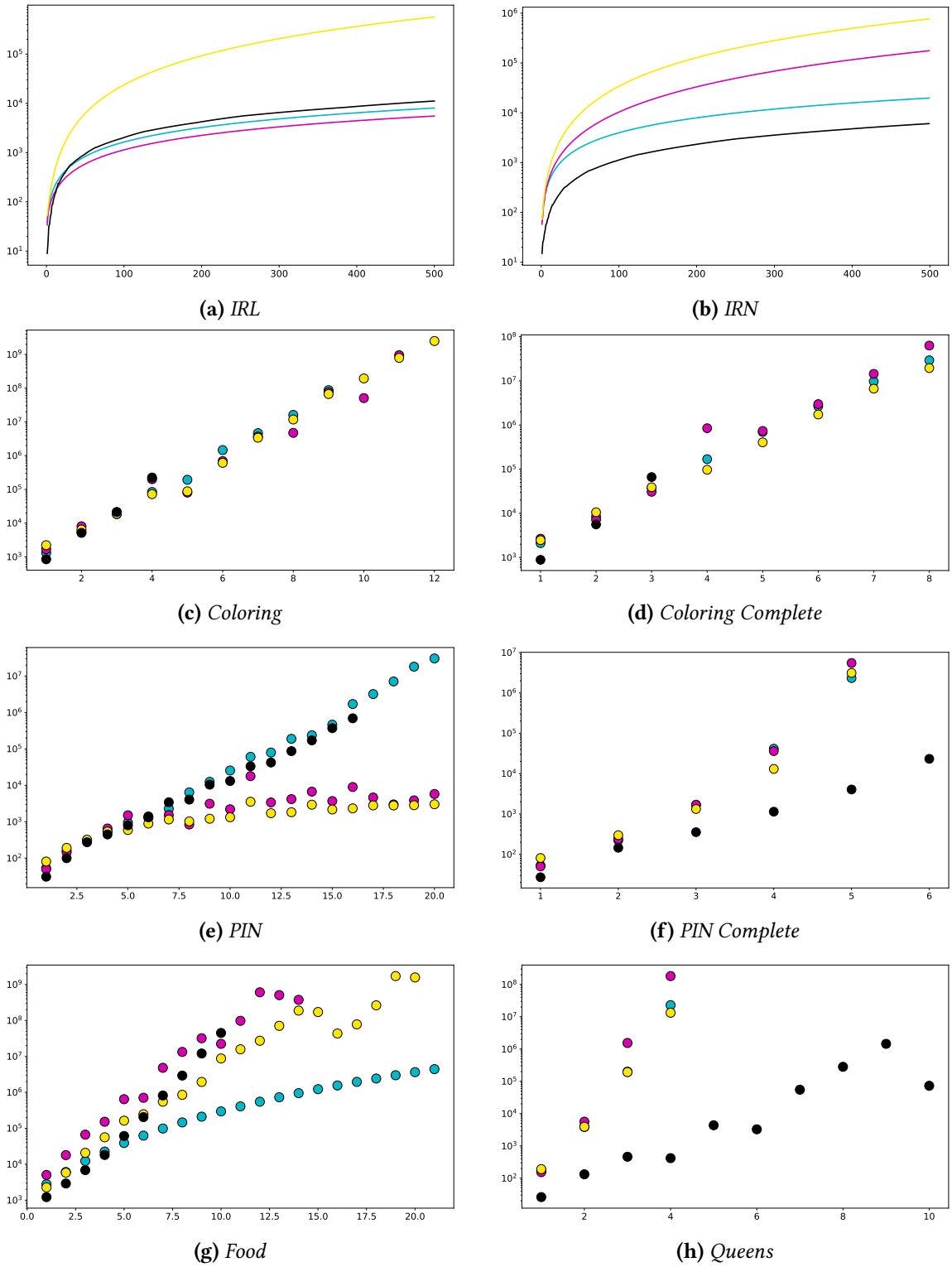
5.5 V-tree Optimization

Before delving into the second approach for PASP compilation, which uses unrestricted V-trees, we discuss the impact of V-tree initialization and reordering during the compilation process.

5.5.1 Static Initialization of V-trees

Problems related to *structured decomposability* were known even before the work of Darwiche when he proposed SDDs (Pipatsrisawat and Adnan Darwiche, 2008; Adnan Darwiche, 2011). With this in mind, Darwiche proposed some heuristics to initialize SDDs *V-trees* in a way that tries to minimize the growth of the circuit during the compilation process, after many conjoin and disjoin operations.

Hence, two methods for better initialization of *V-trees* were proposed in the paper that introduced SDDs: *minfill* and *MINCE* (Adnan Darwiche, 2011). The former adapts another heuristic proposed by Darwiche for DNNFs (Adnan Darwiche, 2001), exchanging the roles of clauses and variables, and use the *minfill* heuristic for variable elimination, while still ensuring that the leftmost child of a *V-tree* node is always the one with the smallest number of children. The latter, on the other hand, is a heuristic that generates a variable ordering, which is not as desirable as the one generated by *minfill*, since the



(a) *IRL* **(b)** *IRN*

(c) *Coloring* **(d)** *Coloring Complete*

(e) *PIN* **(f)** *PIN Complete*

(g) *Food* **(h)** *Queens*

Figure 5.6: *Comparison of circuit size for the different (top-down and bottom-up) compilers across eight different datasets. The x-axis and y-axis represent, respectively, the size of the circuit produced by a compiler and the instance size. Cyan, magenta, yellow, and black represent, respectively: C2D, D4, SharpSAT-TD and the bottom-up SDD compiler. Datasets where the black dots are consistently under the other colors representing scenarios where the bottom-up SDD compiler is more efficient; and vice versa.*

constraint of using a right-linear *V-tree* is basically enforcing a BDD structure. A possible modification to this method, proposed by Darwiche, would be to use a balanced or random *V-tree*, obtained from the variable ordering generated by *MINCE*; but the results were not as promising as the ones obtained by *minfill* (Adnan Darwiche, 2011).

Although both of these methods are interesting, they do not take advantage of the program structure, i.e. they were developed for CNF-based (top-down) approaches. By cleverly exploiting the program structure, we can achieve better results, as we show soon.

The heuristic proposed in this work is quite simple: represent the dependency graph of the program, count the number of descendants of each node (reachable nodes, using a depth-first search), and use this information to guide the construction of the *V-tree*. We expect that nodes with a high number of descendants "imply" in many other nodes, and thus they should appear early in the *V-tree*. For example, we have Figure 5.7.

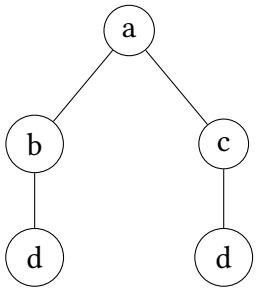


Figure 5.7: *Dependency graph of the program.*

Using this topological ordering, we can construct the *V-tree* that has on its left nodes with a high number of descendants and on its right leaves of the dependency graph. As we show in Section 5.9, this approach vastly improves the performance of the compilation process, outperforming both MinFill and MinDegree.

5.5.2 Dynamic Reordering of V-trees

Alone, the approach of initializing the *V-tree* with a good heuristic is indeed promising, but it is well known that compilation is indeed a hard task. If not, many well-known hard Computer Theoretical problems could be associated with different complexity classes.

Due to the hardness of the compilation task, it is expected that eventual blow-ups in the size of the circuit will happen, so it is only natural to think about a dynamic reordering of the circuit, in order to keep the circuit as small as possible, while not wasting too much time in the reordering process itself.

With this in mind, the work of (Choi and Adnan Darwiche, 2013) proposes two different heuristics for dynamic re-ordenation, based on the concept of reordering the variables of an OBDD (Rudell, 1993). The proposal of (Choi and Adnan Darwiche, 2013) is to reorder the structure of an SDDs by the application of *rotations* and children *swap*, but initially did not cover X-constrained V-trees. This extension was proposed soon after in (Oztok *et al.*, 2016).

This approach has achieved great success in the context of compiling PLPs, since the

work of (Vlasselaer, Renkens, *et al.*, 2014), when compiling rules directly into SDDs, achieved a better performance with dynamic reordering of the *V-tree*. The circuit obtained by this approach did not only have a considerably smaller size, but also did not take too much time to be compiled, when compared to their static counterpart.

5.5.3 Heuristic-Based PASP Compilation

By combining both static and dynamic approaches heuristics to initialize and reorder the variable tree of the SDD, during the compilation process, we can obtain algorithms for enhancing the previously described methods for PASP compilation. For instance, compilation like the one of *smProblog* is able to take advantage of an unrestricted search space for the *V-tree* initialization or reordering, which could render a more succinct circuit, analogous to the results of (Vlasselaer, Renkens, *et al.*, 2014) (that took advantage of the dynamic reordering of the variable tree to obtain considerably smaller circuits for PLP programs, when compared to methods without dynamic reordering). On the other hand, the 2AMC approach with X-*constraint* (or even X-determinism) needs the development of new heuristics, that preserve a good variable ordering, while not disrespecting the desired constraints over the circuit.

In order to combine these two improvements for the V-tree during the bottom-up compilation, one simply needs to call the heuristic when selecting the V-tree that will be fed to the bottom-up algorithm and to apply dynamic reordering of the variable tree when the SDD's size surpasses a certain threshold, as described (Choi and Adnan Darwiche, 2013). For example, one could call the dynamic reordering when the SDDs size doubles.

5.5.4 Preliminary Results

In this section, we focus on two different aspects: the impact of using the proposed heuristic for V-tree initialization and the impact of dynamic reordering on the bottom-up compilation. For the V-tree initialization, we considered a slightly different version of the Coloring dataset, where random graphs increase at an lower rate, in order to capture the behavior of the different heuristics across different graph sizes. Accordingly, Table 5.1 shows that the proposed heuristic outperforms the other two heuristics, both in terms of memory usage and circuit size, highlighting the importance of considering program structure when compiling PLPs, as opposed to other top-down CNF-based approaches.

#Nodes	Ours		MinDegree		MinFill	
	mb	s	mb	s	mb	s
13	**374**	**2.42**	10020	229	805	4.07
14	**387**	**5.48**	-	-	754	7.21
15	**997**	32.2	-	-	2939	**31.99**
16	**9640**	**279**	-	-	-	-
17	**10411**	**588**	-	-	-	-

Table 5.1: *Comparison of memory (**mb**) and time (seconds) between the proposed heuristic, MinDegree and MinFill for V-tree initialization in the **Coloring** dataset for bottom-up compilation.*

While V-tree initialization has its importance in order to provide a good start for generating succinct circuits, dynamic reordering is one of the most interesting aspects of bottom-up compilation, since it provides the possibility of exploring different variable orderings during compilation. Usually, when using top-down compilation, one chooses a variable ordering and stays with it during all compilation steps; thus, dynamic reordering provides a sound way to circumvent possible issues with the chosen ordering. Figure 5.8 shows how this dynamic reordering can be used to surpass top-down approaches on some of the datasets that the previous section had poor performance, such as IRN and Food. In the Food dataset, dynamic reordering was essential to significantly improve the performance of bottom-up compilation.

It is also important to note that dynamic reordering comes at a cost, as it requires additional computation time and resources to explore better V-trees during compile time. Although this can be more granularly fine-tuned during compilation, we observed that this approach hurts performance on some cases, such as PIN Complete, where this reordering overhead broke the time wall of 30 minutes and made it impossible to compile the instance of size 6 under the time limit.

5.6 Relaxing Constraints for PASP Compilation

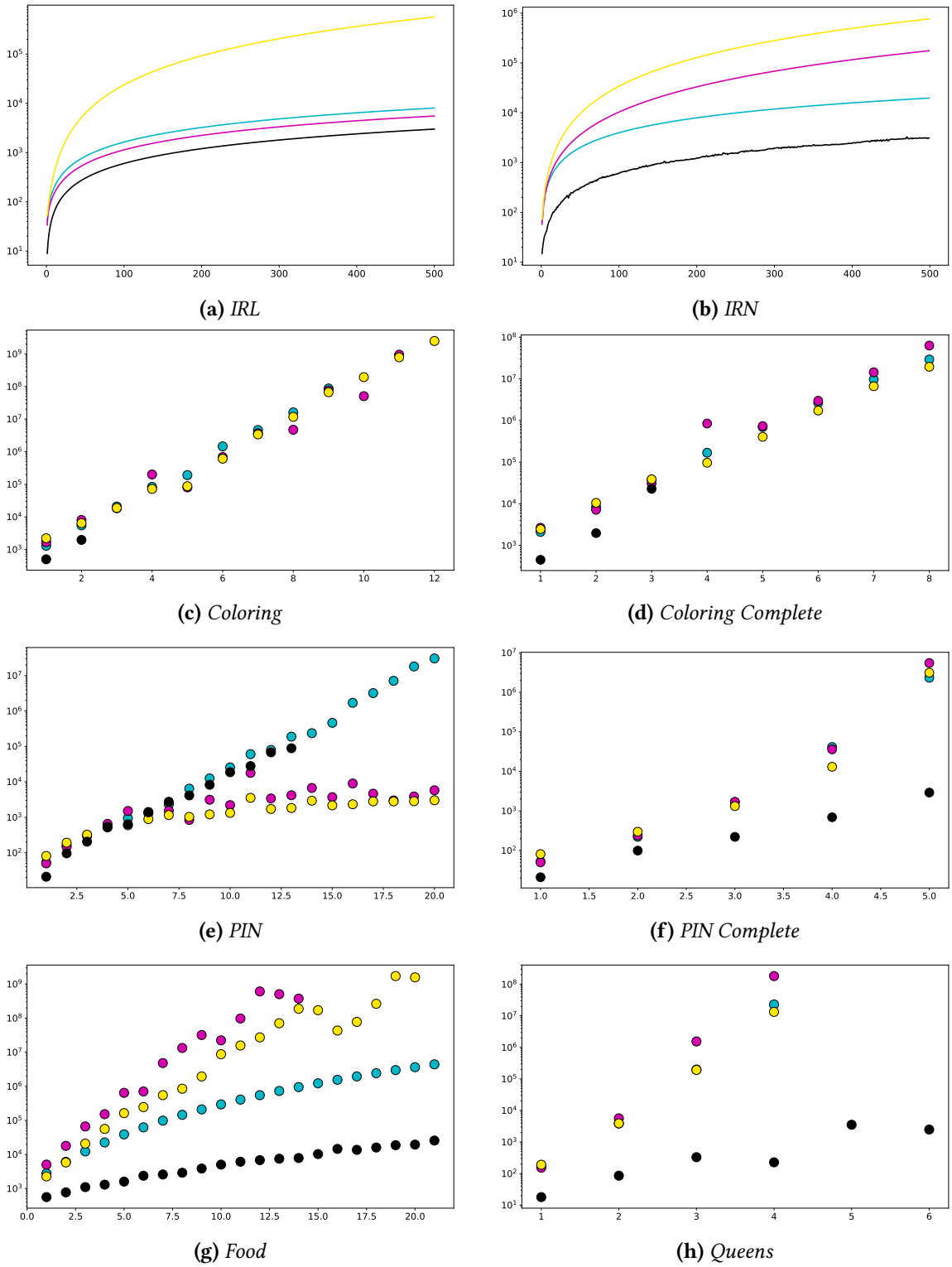
In this section, we comment on the work of (Totis, De Raedt, *et al.*, 2023) for the *smProblog* system, since both PASP and *smProblog* share the *Stable Model* and MaxEnt semantics, but with one key difference: the *smProblog* allows for *inconsistencies* in the program, while this do not consider this possibility in PASP. This difference is crucial, since the enumeration step of the *Stable Models* will not be the same, as all total choices will have at least one model in PASP, while this is not necessarily true for *smProblog*.

By taking inspiration from this unconstrained approach of (Totis, De Raedt, *et al.*, 2023), we propose an alternative compilation method that leverages recent discoveries in V-tree restructuring (H. Zhang *et al.*, 2025).

5.6.1 Particularities of the smProblog System

Another key detail of the *smProblog* system is the usage of the *d-Sharp* knowledge compiler, described in the previous section. While this may seem as like a minor detail, the implementation of *d-Sharp* does not allow for the imposition of the *X-firstness* or *X-determinism* over the circuit. Thus, need for an intractable normalization step by enumerating all *Stable Models* of the program is necessary.

This choice, although small, is one of the key arguments for the usage of SDDs, since one is not locked by specific design implementations of the knowledge compiler, and can both choose the *V-tree* initialization and check for properties as *X-constraindness* in polynomial time.



(a) *IRL* **(b)** *IRN*

(c) *Coloring* **(d)** *Coloring Complete*

(e) *PIN* **(f)** *PIN Complete*

(g) *Food* **(h)** *Queens*

Figure 5.8: *Comparison of circuit size for the different (top-down and bottom-up with dynamic reordering) compilers across eight different datasets. The x-axis and y-axis represent, respectively, the size of the circuit produced by a compiler and the instance size. Cyan, magenta, yellow, and black represent, respectively:* C2D, D4, SHARPSAT-TD *and the bottom-up SDD compiler with dynamic reordering. Datasets where the black dots are consistently under the other colors representing scenarios where the bottom-up SDD compiler is more efficient; and vice versa.*

5.6.2 Embracing the Enumeration

At a first glance, choosing to enumerate all *Stable Models* of a program may seem as like a poor choice, since the number of possible total choices increases exponentially with the number of probabilistic facts in the program. However, by abiding the constraints imposed by the 2AMC, there is a trade-off of being able to search for better *V-trees* without any structural constraints, which can lead to exponentially more succinct circuits than the ones obtained by the 2AMC approach.

Although it is certainly not guaranteed that there are exponentially better *V-trees* for a given program, or that we are capable of finding them in a reasonable amount of time, it is safe to say that, for programs with few probabilistic facts, the enumeration of all *Stable Models* becomes a viable step. Moreover, by drawing a parallel with the previous discussion of trade-offs between enumerative approaches and KC, we can see that an approach that resembles the one presented by *smProblog* can be seen as optimizing even more enumerative approaches, as one is able to encode the results of the enumeration in a circuit and, after this, all further inferences are amortized.

5.6.3 Advantages of the PASP Semantics

As stated in the *smProblog* paper (Totis, De Raedt, *et al.*, 2023), the main advantage of using some PASP semantics, such as the ones from *P-log* Baral *et al.*, 2009 or LP^{MLN}, when compared to *smProblog* semantic, is the fact that these two PASP do not consider different normalization constants per total choice. Hence, there is no need to enumerate an exponential number of models to compute different normalization constants, since these two semantics normalize the probability distribution globally, and not locally.

Therefore, besides the fact that the PASP languages encompass the same *Stable Model* semantics, by allowing negative cycles, there are also ASP probabilistic semantics capable of performing the same task, but more efficiently.

5.6.4 Naive Unconstrained Bottom-Up Compilation

A naive approach to PASP compilation with an unconstrained V-tree would be to follow the steps described in the bottom-up algorithm; and then perform the same normalization step as in (Totis, De Raedt, *et al.*, 2023). The shortcomings of this approach are immediate: the normalization step is not efficient, as it requires an evaluation of the circuit in order to compute the normalization weights (Totis, De Raedt, *et al.*, 2023). To compute such weights, we first describe the algorithm proposed by (Totis, De Raedt, *et al.*, 2023):

(i) First, replace each circuit *leaf* for a literal l by a set of partial models $\{\{l\}\}$; replace each *disjunction*by the *Union* of its children, and each *conjunction* by the *Cartesian Product* of its children.

(ii) Perform a bottom-up traversal of the circuit, which results in a set of models.

(iii) From the previous set of models, compute a function $\# : \Omega \to \mathbb{N}$, where Ω is the set containing all total choices of the program, that maps total choices to the number of models that satisfy it.

(iv) By taking the inverse of the function $\#(\omega)$, for a total choice ω, we can compute the normalization constant that appears in the definition of the MaxEnt semantics 2.

Because the authors consider inconsistent programs, there is the possibility that a total choice does not appear during this (enumerative) step and, thus, is inconsistent. On the other hand, the authors state that the normalization step can be simplified when considering PASP programs (as commented earlier), since there is no need to consider different normalization constants for each total choice, because it is the same for all models, the weight of all possible worlds (Totis, De Raedt, *et al.*, 2023). Hence, it is possible to obtain the normalization constant through a single evaluation of the weight of the root of the circuit.

5.6.5 Abandoning Enumeration

The choice for the V-tree is a known hard problem, and bad initializations or hard constraints (such as X-determinism) can lead to poor performance during compilation. To address this issue, we propose an alternative approach for bottom-up compilation: compiling an SDD without imposing any constraints on the structure of the circuit, in order to relax constraints and ease the compilation process; and then restructure the circuit by following the algorithm proposed by (H. Zhang *et al.*, 2025).

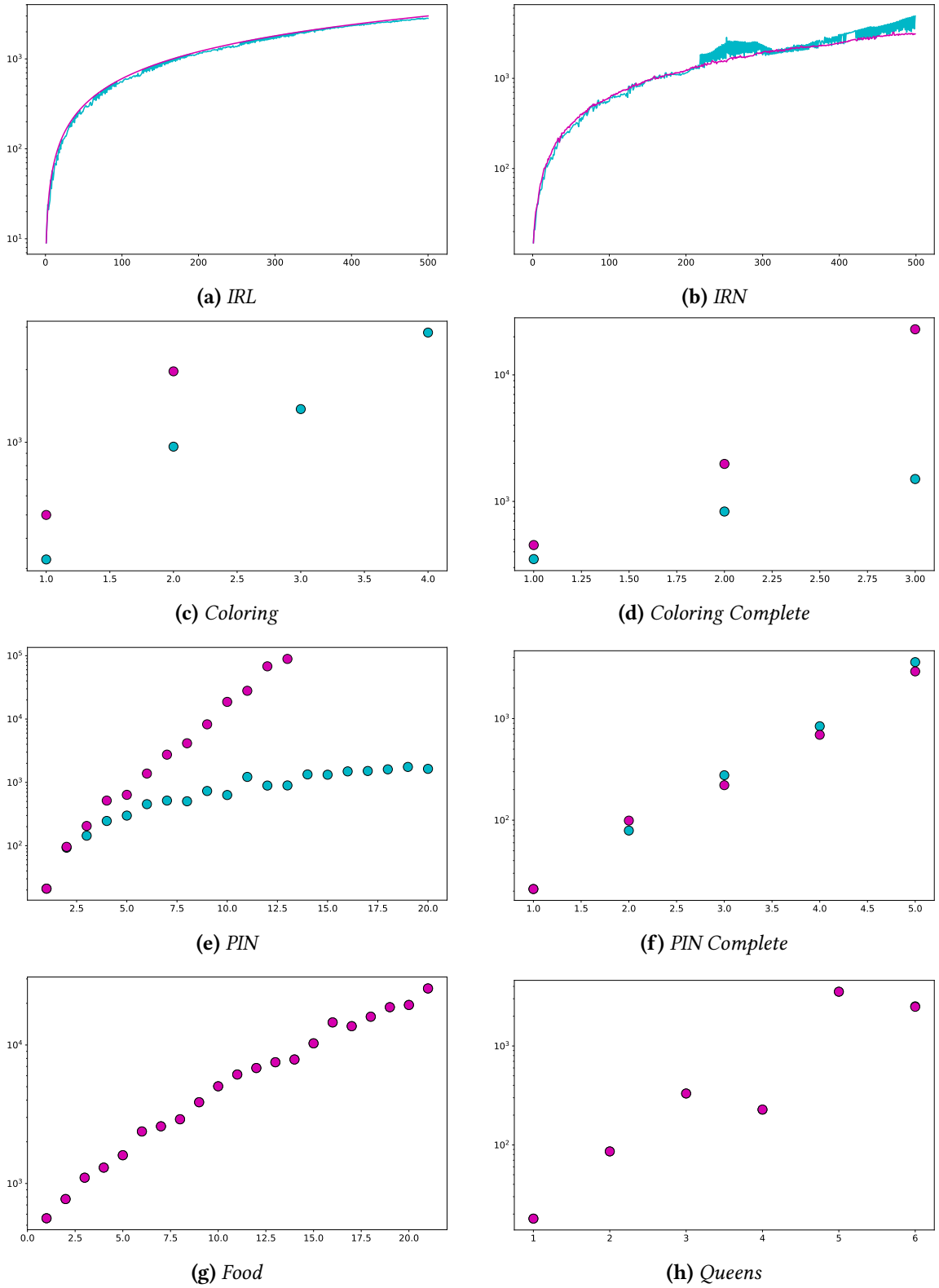
This algorithm is fairly complex, bridging techniques from Causality with Logic Circuits, providing different approaches to restructuring circuits. Thus, we do not delve into the details of this algorithm in this work. However, assuming that such an algorithm exists, we can see a fairly straightforward way to apply it to our problem. First, compile an SDD using dynamic reordering without imposing any structural constraints to its V-tree. After the compilation is finished, apply the algorithm proposed by (H. Zhang *et al.*, 2025) to restructure the circuit by choosing a balanced V-tree with the same variable ordering as the one obtained during compilation, but imposing X-constrainedness (probabilistic variables appear on the leftmost side).

5.6.6 Preliminary Results

In order to highlight how imposing X-constrainedness from the start can be detrimental to compilation process. Figure 5.9 shows in datasets Coloring and PIN, where previous bottom-up approaches had poor performance, and using an unconstrained compilation can lead not only to considerably more succinct circuits, but also enable compilation of much larger instances (2 more instance sizes for the Coloring dataset and 7 more instance sizes for the PIN dataset). This difference in size between constrained and unconstrained compilation can be seen as the "price" of imposing X-constrainedness from the start.

5.7 Non-Incremental Compilation

Non-Incremental compilation is a novel paradigm for compilation proposed by (Colnet, 2023). The main idea is to take into consideration the order of the operations that we apply to the circuit during compilation. When proposing such approach, (Colnet, 2023) assumes that one wants to compile disjoint formulas φ and ψ (that do not share any variables)



(a) *IRL* **(b)** *IRN*

(c) *Coloring* **(d)** *Coloring Complete*

(e) *PIN* **(f)** *PIN Complete*

(g) *Food* **(h)** *Queens*

Figure 5.9: *Comparison of circuit size for the unconstrained and X-constrained bottom-up compilation across eight different datasets. The x-axis and y-axis represent, respectively, the size of the circuit produced by a compiler and the instance size. Cyan and magenta represent, respectively, constrained and unconstrained compilation with dynamic reordering. Datasets where the cyan dots are consistently over the magenta representing scenarios imposing X-constrainedness had an heavy impact on circuit size.*

using *structure decomposable circuits*. By exploiting this disjointness, the author noted that the circuit increases linearly in size (a theoretical result), but does not provide any practical means to achieve such disjointness.

In our work, we propose a method for achieving such disjointness: by modeling the problem of finding disjoint components as a combinatorial optimization problem. We use flux algorithms to find disjoint components of the program we want to compile, if there are more than one; or we use a heuristic approach to break down a program into smaller disjoint components and, then, combine them to achieve the desired representation.

As we show in Section 5.9, this approach was not only able to greatly surpass the bottom-up approach described until now, but this idea of cleverly rearranging the order of the rules we use to compile the program can be applied to the standard incremental bottom-up described in previous sections.

5.7.1 Motivation

A key challenge in Knowledge Compilation are the intermediary circuits that are generated during the compilation process. Although certain formulas can be represented in a compact form, with polynomial size, the compilation process itself can lead to an exponential blow-up in the size of the circuit when compiling the program (Colnet, 2023). Thus, we present a theorem that shows that PASP non-incremental compilation can lead to more efficient circuits, especially in cases where others approaches would lead to exponential blow-up when compiling intermediate circuits. The core idea behind this approach is illustrated in Figure 5.10, where the compilation of $\Delta \wedge \Sigma$ is performed by dividing the compilation task into different *clusters*, that are independently compiled and then conjoined; as opposed to the standard incremental compilation, which linearly conjoins.

In more detail, we create an undirected dependency graph of Clark's completion of the program, where nodes are indexed by all the heads that are present in the program, and we have edges between nodes u and v iff they have an atom in common, either as a head or in the body. With this graph, we are able to detect disjoint subsets $c_1, \ldots, c_m$ of the Clark Completion rules by applying a connected components algorithm, such as Union-Find; and then compile each component c_i into a circuit Δ_i by using the bottom-up algorithm. Finally, we conjoin all Δ_i to obtain the representation of Equation (4.1).

(a) *Incremental* **(b)** *Non-incremental*

Figure 5.10: *Example of incremental and non-incremental approaches for compiling* $\Delta \wedge \Sigma$.

5.7.2 Preliminaries

Before we present the Non-Incremental compilation algorithm, we contextualize the results presented in the work of (Colnet, 2023). First, we define what we mean by *Incremental Compilation* and *Disjoint Subsets* of a program.

Definition 5.7.1 (Incremental Compilation)**.** A bottom-up compilation of a boolean formula φ is asequence $(\Sigma_1, I_1), \ldots, (\Sigma_N, I_N)$, where $\Sigma_1, \ldots, \Sigma_N$ are elements such that $\Sigma_N \equiv \varphi$ and where, for every $i \in \{1, \ldots, N\}$, I_i is an instruction telling how Σ_i is obtained. Three types of instructions are possible:

- $\Sigma_i = \text{Compile}(c)$ for some constraint c of φ. Then $\Sigma_i \equiv c$.
- $\Sigma_i = \text{Apply}(\Sigma_j, \Sigma_k, \wedge)$ for some $j, k < i$ such that $\Sigma_i, \Sigma_j, \Sigma_k$ respect the same vtree. Then $\Sigma_i \equiv \Sigma_j \wedge \Sigma_k$.
- $\Sigma_i = \text{Restructure}(\Sigma_j)$ for some $j < i$. Then $\Sigma_i \equiv \Sigma_j$ and their V-trees may differ.

Lemma 5.7.1 (Lemma 13 (Colnet, 2023))**.** *Let ϕ_1 and ϕ_2 be such that $var(\phi_1) \cap var(\phi_2) = \emptyset$. Given $(D_1^1, I_1^1), \ldots, (D_s^1, I_s^1)$ an incremental $strDNNF(\wedge, r)$ compilation of ϕ_1, and $(D_1^2, I_1^2), \ldots, (D_t^2, I_t^2)$ an incremental $strDNNF(\wedge, r)$ compilation of ϕ_2, there is an incremental $strDNNF(\wedge, r)$ compilation of $\phi_1 \wedge \phi_2$ whose largest element has size at most* $\max_i |D_i^1| + \max_j |D_j^2| + 1$.

Since SDDs are a subset of *strDNNF* (Adnan Darwiche, 2011), it follows that Lemma 5.7.1 can be applied to *SDDs*. Note that this Lemma 5.7.1 presents an upper bound on the size of the largest SDD obtained through the non-incremental compilation. In detail, by choosing incremental compilations $\Delta = (D_1, I_1), \ldots, (D_n, I_n)$ and $\Sigma = (S_1, J_1), \ldots, (S_m, J_m)$ of ϕ_1 and ϕ_2, respectively, we can build a non-incremental compilation $(F_1, K_1), \ldots, (F_{n+m}, K_{n+m})$ of $\phi_1 \wedge \phi_2$ as follows:

- $K_i = I_i$ and $F_i = D_i$ for $1 \leq i \leq n$.
- $K_{n+j} = J_j$ where every S_j is replaced by $F_{n+j} = D_n \wedge S_j$. Thus, the root F_{n+j} is and $\wedge$-node with children D_n and S_j (hence the size of F_{n+j} is equal to $1 + |D_n| + |S_j|$).

Since $var(D_n) \cap var(S_j) \subseteq var(\Delta) \cap var(\Sigma) = \emptyset$, each F_{n+j} is a *structured decomposable* circuit. Moreover, it is a bottom-up compilation of $\phi_1 \wedge \phi_2$ whose size is at most $\max(\max_i |D_i|, 1 + |D_n| + \max_j |S_j|)$.

5.7.3 Non-Incremental Algorithm

Suppose that a program P can be decomposed into two disjoint sets of rules, ϕ_1 and ϕ_2, that do not share any variables. Incremental compilations for both ϕ_1 and ϕ_2 exist, since we can compile both of them using the bottom-up algorithm. Therefore, we can apply Lemma 5.7.1 to obtain a non-incremental compilation of P.

This is the core idea regarding the non-incremental compilation of P. If the program P is disjoint by nature, i.e. the dependency graph of P has more than one connected component, then we can identify such components through an algorithm such as Union-Find. On the other hand, if P is not disjoint by nature, we can apply a min-vertex-cut algorithm to find a set of nodes that, when removed from the previous graph, will render the

graph disconnected into disjoint components, allowing application of the non-incremental compilation. The only special consideration is that, after compiling the disjoint components into a circuit Δ, one also needs to compile the logical constraints that were removed from the graph into another circuit Σ. The final circuit representing the program P is then obtained by conjoining $\Delta \wedge \Sigma$.

This process is summarized by the following theorem:

Theorem 5.7.2. *Given a program P, we can determine m disjoint subsets of the program in polynomial time, that can be non-incrementally compiled into a circuit of size at most $(m-1) + \sum_{i=1}^{m} |S_i|$, where S_i is the size of the largest circuit obtained by compiling each subset using the bottom-up KC.*

The result of this theorem is of special interest because usual conjoin (multiplication) of SDDs is polynomial w.r.t. the size of the SDDs, but not linear. This multiplication is indeed quadratic, bounded by the multiplication of the sizes of the two SDDs. Therefore, having an upper bound that guarantees that the size of intermediary circuits will not exceed the sum of the sizes of the individual circuits is highly desirable to avoid exponential blow-up of intermediate circuits.

Proof of Theorem 5.7.2. First, it is possible to obtain m disjoint subsets of head rules $\{c_1, \ldots, c_m\}$ of a program P in polynomial time, by using an algorithm for k-vertex-connectivity (such as a min-vertex-cut algoritm). Next, we can compile the first set of head rules $r_i \in c_1$ into an SDD Δ by applying the bottom-up KC algorithm. Then we conjoin Δ with the SDD Σ_i representing the bottom-up compilation of each head rule $r_i \in c_2$, creating k different "clusters" $\Delta \wedge \Sigma_i$ of SDDs 5.10.

Finally, we conjoin all these different clusters, obtaining an SDD with size at most $|S_1| + |S_2| + 1$ 5.7.1, where $|S_i|$ represents the size of the largest SDD obtained through an incremental compilation of the set of head rules in c_i, when following the bottom-up KC algorithm.

This process can be repeated with the remaining components $c_3, \ldots, c_m$, resulting in an SDD of size at most $(m-1) + \sum_{i=1}^{m} |D_i|$. □

The above Proof 5.7.3 only regards the compilation of the disjoint subsets of rules. As commented before, to obtain the correct semantics of the original PASP program, it still is necessary to retrieve the rules removed after the application of the min-vertex-cut algorithm, and conjoin them with the circuit to obtain the desired compilation.

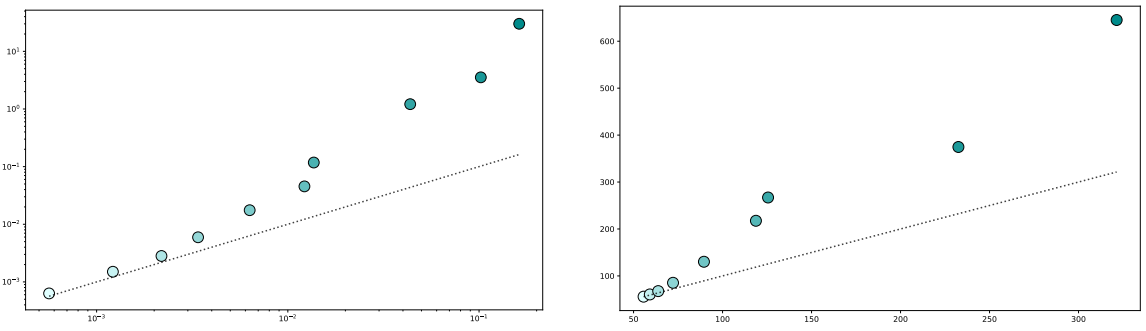
An interesting remark is that this non-incremental compilation could be applied to the compilation of PLPs, since we only assume the structure of a dependency graph. In reality, this approach is even more general due to Non-Incremental compilation being a relatively novel concept, this is the first work that proposes a principled approach to obtaining such disjoint components in order to utilize this approach in practice.

5.7.4 Preliminary Results

Finally, we present a result similar to the one in Table 5.1, in the Coloring dataset with slower growth rate in graph sizes, but comparing Non-Incremental and Incremental compilation. Figure 5.11 highlights how the non-incremental compilation performed better not only in terms of memory usage during compilation, but also in terms of compilation time. This results, although unexpected at first glance, is due to the Non-Incremental approach being less succeptible to the order of the rules that are being compiled, i.e. the order in which we compile the Clark's completion has a great impact on the compilation time and memory usage.

When first exploring both the Non-Incremental and Incremental compilation, the only optimization that we proposed to the order in which rules of the Clark's completion are compiled was to always prioritize the compilation of facts, as they are the most basic elements of the knowledge base. After seeing the results of Fig 5.11, we realized that the heuristic proposed to generate good V-tree initializations should also be used to guide the order in which rules are compiled. After applying the heuristic to the order of the rules, we observed a significant improvement in the Incremental compilation time, achieving similar performance as the Non-Incremental approach; and reducing the memory usage considerably. Although the Non-Incremental approach still used less memory overall, we could see that the memory usage difference for the largest instance of the Coloring dataset fell from $0.6GB$ to $0.3GB$, still noticeable, but not as large as before.

Therefore, the Non-Incremental approach had three main impacts in this work: first, further improving the incremental approach memory and time usage due to the aforementioned optimizations; second, providing a more robust, less rule-ordering susceptible compilation; and third, providing a sound approach for splitting a complex compilation problem into smaller, more manageable parts, that can be easily parallelized without sacrificing performance due to encountering bad rule ordering.



(a) *Compilation time (seconds).* **(b)** *Peak memory usage (mb).*

Figure 5.11: *Comparison between incremental (y-axis) and non-incremental (x-axis) compilation as we increase the number of nodes (darker colors) of the graph Coloring program. The dotted black line represents the baseline; points above it indicate that the incremental approach performed worse.*

5.8 Discussion About Circuits

Before advancing into the experimental results (Section 5.9), we highlight the importance of compilation for Neuro-Symbolic systems and briefly cover what remains to be addressed with respect to approximate compilation techniques.

5.8.1 Learning the Circuit

Another major advantage of using KC is the ability to differentiate the weights of a circuit, which can be used as the structure for a learning algorithm. This is particularly useful when considering Neuro-Symbolic approaches, that combine the Knowledge Representation of the program to define complex loss functions over Neural Networks.

Therefore, the bottom-up compilation of the program through the use of SDDs is not the last step for an efficient PASP KC algorithm, as we discussed in this section. It could also require a normalization step to obtain the MaxEnt semantics (when using the *smProblog* approach); and, after that, differentiation can be done, to learn both parameters of the circuit and Neural Networks acting as *annotated disjunctions*, through the use of gradient descent (Geh *et al.*, 2024; Z. Yang *et al.*, 2023).

Withal, advancements in the field of Probabilistic Circuits can be applied in the learning of the circuit, such as regularization techniques specifically designed for probabilistic circuits (A. Liu and Van den Broeck, 2021).

5.8.2 Partial Observations

With inspiration from Problog 2 paper (Fierens *et al.*, 2015), we also claim that there is the possibility of compiling circuits w.r.t. certain queries. The idea behind this approach is to only compile facts that are associated with the query, and not the whole program.

This is a recurrent theme when only partial observations are considered, and can act as a considerable *speed-up* in the overall inference routine. This is particularly useful when considering the *Neuro-Symbolic* approach, where not only many queries are made, but they are repeated each epoch of the learning algorithm.

Note that this approach can be particularly useful when considering the *smProblog* approach, since the removal of facts that are not associated with the query can lead to a more succinct circuit representation, which can alleviate the enumerative step of the normalization algorithm.

5.8.3 Approximate Compilation

Approximate methods for compiling PLPs are a well-studied topic, since the complexity of exact inference for these languages is known to be intractable (Mauá and Cozman, 2020). Thus, the possibility of using approximate methods for inference (with or without KC is of great importance for the development of the Neuro-Symbolic field.

Within this context, the work of (Pajunen and Janhunen, 2021) in ASEO and (Manhaeve, Marra, *et al.*, 2021) in *DPLA* are both different methods, applied to PASP and

Problog programs, respectively, that aim to circumvent the intractability associated with the exact compilation of PLPs, by providing approximate methods for inference (and, consequently, learning) of the parameters of the program, with the tradeoff of not considering all possible proofs of the query, but only the one with the highest probability (Pajunen and Janhunen, 2021; Manhaeve, Marra, *et al.*, 2021).

The first of these two methods, called ASEO, is based on the concept of enumerating the k most probable proofs of a query, while the remaining ones are discarded and not considered in the inference process (Pajunen and Janhunen, 2021). This process is done through the resolution of the homonymous problem, ASEO, since it does not enumerate solutions, and yet enumerate the most probable Answer Sets for a query (for an example, the k most probable that satisfy a literal q and evidence e) (Pajunen and Janhunen, 2021). On the other hand, the second method, *DPLA*, is based on combining the Selective Linear Definite (SLD) tree for best proof with the A^* search, in order to find the best proof (since the work only considered *stratified* programs, and, therefore, there is no need to consider Answer Sets), the k best proofs of a query or even all proofs (Manhaeve, Marra, *et al.*, 2021).

In this sense, the work present in the literature contemplates approximate methods for inference, with different approaches to compilation in *stratified* PLPs (Vlasselaer, Van den Broeck, *et al.*, 2016; Manhaeve, Marra, *et al.*, 2021). However, approximate methods for PASP are still to be explored, as only enumerative methods have been presented in the literature (Pajunen and Janhunen, 2021). Therefore, we believe that a generalization of the T_{C_P} operator for general Normal Logic Programs (instead of *stratified* ones, as the original work proposes) can be a promising approach for future research in this area.

For example, by splitting a PASP program, through the methods described in (Lifschitz and Turner, 1994; Ben-Eliyahu-Zohary, 2021), into a *stratified* part and a *non-stratified* part, it would be possible to apply the T_{C_P} operator on the former and PASP KC methods on the latter, creating a hybrid approach for approximate inference by KC in PASP. However, this is only a suggestion, and further investigation is needed to determine the effectiveness of this approach and to pursue further refinements.

5.9 Experimental Results

This section wraps up the experimental results of our proposed methods for Non-Incremental PASP KC, describing in further detail the datasets chosen for evaluation, research questions, and some relevant implementation considerations.

5.9.1 Datasets

Here, we describe each class of programs used in our experiments. As opposed to other works on PASP KC, we focus on the programs structure, and not their CNF representation. Although this may seem quite intuitive, many previous works, such as (Eiter, Hecher, *et al.*, 2021; Eiter, Hecher, *et al.*, 2024; Kiesel and Eiter, 2023), use CNF benchmarks as a proxy for PASP KC programs. However, this approach has limitations, as it does not offer any means to exploit program structure, which is crucial for our proposed methods,

both bottom-up KC or the V-tree heuristics.

In order to cope with these limitations, we propose using an extension to the benchmark proposed by (Azzolini and Riguzzi, 2024). This benchmark has two major drawbacks: the Coloring dataset is not consistent, i.e. there are total choices without any models, which makes the whole program probability less than 1; and there are no cardinality constraints. To address these issues, we introduce a new dataset called Food, which models the problem of selecting an item from a menu, given a distribution of preferences, subject to a majority vote on the selected item.

Coloring

The Coloring dataset in Listing 5.7, which models the problem of coloring a graph with three colors is consistent, since we added atoms $uncolorable$ and $colorable$, instead of using integrity constraints that would throw away models. This dataset appears in two different versions in our experiments:

1. Coloring: where the edges were sampled from a random $G_{n,p}$ graph; and
2. Coloring Complete: where the graph is complete.

Program 5.7 Definition of the Coloring Dataset (class of programs).

```
% Grounded Coloring Program
0.5::edge(X, Y). % Probability of edge between nodes X and Y
node(X). % Fact also derived from the dataset
% A node can have only one of three colors
red(X) :- node(X), not green(X), not blue(X).
green(X) :- node(X), not red(X), not blue(X).
blue(X) :- node(X), not red(X), not green(X).
% Symmetry between edges
e(X, Y) :- edge(X, Y).
e(X, Y) :- edge(Y, X).
% 3 graph coloring codification as
uncolorable :- e(X, Y), red(X), red(Y).
uncolorable :- e(X, Y), green(X), green(Y).
uncolorable :- e(X, Y), blue(X), blue(Y).
colorable :- not uncolorable.
```

PIN - Probabilistic Influence Network

The PIN dataset in Listing 5.8 models the dynamics of a disease spread across contact network. Individuals can get infected either by contact with other *infected* individual of the network or by an external event (indicated by the probabilistic predicate *contaminated*). An infected individual might be *symptomatic* or not; non-symptomatic individuals are called *vectors* of the disease.

This type of network displays the typical transitivity closure often used to evaluate logic program inferences (like the Smokers dataset (Vlasselaer, Renkens, *et al.*, 2014)). Relative to the Smokers program, this program also contains challenges relative to non-stratified

negation and cyclic dependencies, as the vector and symptomatic contradictory nature increase considerably the complexity of the inference process, requiring X-determinism to ensure tractability.

Program 5.8 Definition of the PIN (Probabilistic Influence Network) Dataset (class of programs).

```
% Probabilistic Interaction Network (PIN)
0.5::contaminated(1..N).
0.5::friend(1..N, 1..N).
infected(X) :- contaminated(X).
infected(X) :- friend(X, Y), infected(Y).
healthy(X) :- not infected(X).
symptomatic(X) :- infected(X), not vector(X).
vector(X) :- infected(X), not symptomatic(X).
```

Similarly to the Coloring dataset, the PIN dataset has two versions:

1. PIN: where the edges were sampled via snowball sampling on the Bitcoin OTC dataset (Kumar, Spezzano, *et al.*, 2016; Kumar, Hooi, *et al.*, 2018), a methodology similar to that of (S. Wang *et al.*, 2020).
2. PIN Complete: a complete graph.

IRL

The IRL dataset represents a sequence of probabilistic facts and logical rules. It is designed to test the scalability of encoding techniques with respect to the size of the body of the rules, with a fixed number of rules in the program. It is a fairly simple dataset that acts a baseline and should, in theory, behave especially well for top-down compilers, since there is no need to introduce auxiliary variables in the Clark completion.

Program 5.9 Definition of the IRL Dataset (class of programs).

```
% IRL Problem
0.5::a(1..N).
qr :- a(X0), a(X2), a(X4), ..., a(Xeven).
qr :- a(X1), a(X3), ..., a(Xodd), not nqr.
nqr :- a(X1), a(X2), ..., a(Xn), not qr.
```

IRN

For the IRN dataset (Azzolini and Riguzzi, 2024), we employed a similar strategy, varying the parameter N from 1 to 500.

N-Queens

The N-Queens dataset models a probabilistic version of the classical N-Queens problem, where queens must not attack each other. It demonstrates the effectiveness of encoding

Program 5.10 Definition of the IRN Dataset (class of programs).

```
% IRN Problem
0.5::a(1..N).
qr :- a(Xeven).
qr :- a(Xodd), not nqr.
nqr :- a(Xodd), not qr.
```

techniques in handling spatial constraints. Due to the high number of possible conflicts, it is expected that representing this problem via a top-down approach will lead to quick blowups in circuit sizes.

For the N-Queens dataset in Listing 5.11, we have a probabilistic version of the classical N-Queens problem, where each cell block can have a queen with a certain probability. It demonstrates the effectiveness of encoding techniques in handling spatial constraints. Due to the high number of possible conflicts, it is expected that representing this problem via a top-down approach will lead to quick blowups in circuit sizes.

Program 5.11 Definition of the N-Queens Dataset (class of programs).

```
rows(1..N). columns(1..N).
% N-Queens Problem
% For each cell block, there is a queen with a random
% distribution over the columns.
0.5::queen(R, C) :- rows(R), columns(C).
% We encode both satisfiable and unsatisfiable instances
% so there should always be a model
conflict :- queen(R1, C1), queen(R2, C2), abs(R1 - R2) == abs(C1 - C2).
conflict :- queen(R, C1), queen(R, C2), C1 != C2.
conflict :- queen(R1, C), queen(R2, C), R1 != R2.
```

Food

The Food dataset represents a preference selection problem, where the majority needs to decide on an item to be selected (the type of food they will have). This dataset is used to evaluate the impact of different encodings of cardinality constraints including constraints other than “exactly-one-of”.The number of food items can be fixed in order to vary the number of people and, thus, increasing only the cardinality constraints for the majority vote and annotated disjunctions.

When creating instances of the Food dataset in Listing 5.12, we kept the number of food items, M, constant as 4. This effectively fixed the number of voting options and allowed us to study the effects of varying the number of voters, N.

5.9.2 Research Questions

Finally, we conclude the chapter by revisiting all previous experiments in a more unified manner, highlighting the strengths and limitations of our proposed methods. We

Program 5.12 Definition of the Food Dataset (class of programs).

```
% Food Preference Problem
% Define the domains of people and food
person(1..N). food(1..M).
% Annotated disjunctions encode preferences
1/M::prefers(P,1); ...; 1/M::prefers(P,M) :- person(P).
% Exactly one food type must be chosen
1 { chosen(F) : food(F) } 1.
% Someone agrees if their prefered food is chosen
agrees(P) :- person(P), prefers(P,F), chosen(F).
% Constraint: More than half must agree
:- { P : agrees(P) } n//2.
```

do this by summarizing the results obtained from each experiment alongside its respective research question.

1. **Research Question 1:** How the number of auxiliary atoms used to represent a program in CNF form scale?
2. **Research Question 2:** Without any optimization, is the bottom-up KC algorithm competitive with other state-of-the-art algorithms?
3. **Research Question 3:** Is the V-tree initialization heuristic better than other state-of-the-art heuristics?
4. **Research Question 4:** Does the V-tree dynamic reordering offer a good trade-off between compilation time and circuit succinctness?
5. **Research Question 5:** The proposed method for encoding cardinality constraints is competitive with other state-of-the-art methods constraint encoding techniques?
6. **Research Question 6:** Is the price of imposing X-constrainedness too high?
7. **Research Question 7:** Is the proposed non-incremental approach capable of avoiding large intermediate representations?
8. **Research Question 8:** If we combine every possible optimization, can we achieve a significant improvement in compilation time and circuit succinctness, when compared to the state-of-the-art algorithms?

First, we have that **Q1** is answered by Figure 5.5, which shows that, for every class of programs besides IRL, we have a great increase in the number of auxiliary atoms as we increase instance size. This is due to programs with transitive dependencies (or other types of relationships) having a quadratic or even higher increase in the number of auxiliary atoms as we increase instance size.

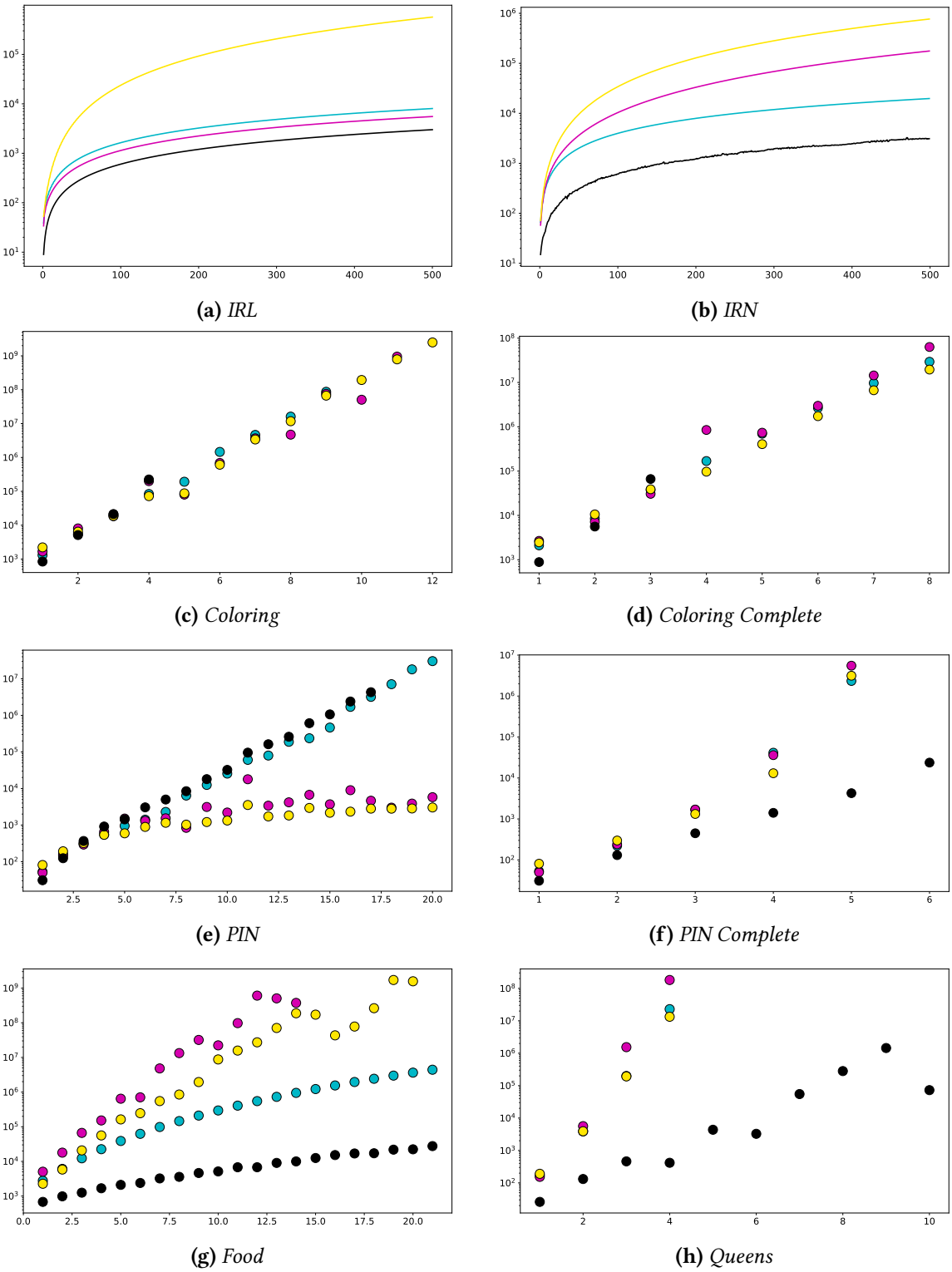
For **Q2**, we observe in Figure 5.6 that the bottom-up, although competitive in some scenarios, such as the IRN, PIN Complete and Queens datasets, still suffers on the majority of datasets, having a small loss on the IRL dataset. We can see that this change for datasets IRL and Food when we consider dynamic minimization, as shown in Figure 5.8 (**Q4**). This is

further enhanced by the V-tree initialization heuristic, which significantly surpasses other CNF-based approaches, such as MinFill or MinDegree (as shown in Table 5.1 - **Q3**).

Regarding different encodings of cardinality constraints, we observe that they were essential to leverage the bottom-up compilation. When using methods such as *Sequential Counters* or *Totalizers*, the maximum instance sizes that the bottom-up approach was able to compile were 13 and 7, respectively (even when using dynamic reordering). On the other hand, by using the proposed method of this work, we were able to not only avoid introducing auxiliary variables, but also were able to compile up to 25 instances. Larger instances could be compiled, but generating such programs consumed more than the time wall of 30 minutes. The displayed results in Figures 5.6, 5.8, 5.9 and 5.12 use the best encoding of cardinality constraints for each compiler, i.e. *Sequential Counters* for c2d and *Totalizers* for both d4 and SharpSAT-TD.

The limitations of imposing X-constrainedness were quite interesting, as Figure 5.9 shows (**Q6**). By imposing X-constrainedness, the most impacted datasets were the ones where the bottom-up approach had significantly worse performance than the top-down approaches, Coloring and PIN. These datasets are characterized by having a slower increase in the number of auxiliary variables introduced as the problem size grows (as opposed to other classes of programs such as Food or Queens, where the growth is much faster). Thus, not imposing X-constrainedness can be an interesting alternative for balancing compilation time, being able to compile larger instances, but requiring post-processing to ensure the correctness of the resulting circuit. On the other hand, an important remark is that many KC libraries (such as the SDD package, used for our approach) are not designed to handle more general Probabilistic Circuits operations, which make it far from trivial to implement the complex algorithm from (H. Zhang *et al.*, 2025) (which does not have any implementation available online). Therefore, the circuit sizes in Figure 5.9 are of unconstrained SDDs, and do not display the size of the resulting circuit after applying the algorithm described in (H. Zhang *et al.*, 2025).

Finally, Figure 5.11 shows how the Non-Incremental can be important to avoid larger intermediate circuit representations, which can lead to increased compilation time and memory usage; or even incapacitate compilation of certain instance sizes (**Q7**). This is further highlighted byFigure 5.12, which shows a more consistent performance of the bottom-up algorithm, alongside the Non-Incremental approach. By combining all methods described in this thesis, the Non-Incremental method was capable of producing smaller circuits than the top-down approach in most of the considered datasets, even though we are imposing hard constraints, such as X-constrainedness, during compile time (**Q8**).



(a) *IRL* **(b)** *IRN*

(c) *Coloring* **(d)** *Coloring Complete*

(e) *PIN* **(f)** *PIN Complete*

(g) *Food* **(h)** *Queens*

Figure 5.12: *Comparison of circuit size for the Non-Incremental bottom-up compilation across eight different datasets. The x-axis and y-axis represent, respectively, the size of the circuit produced by a compiler and the instance size. Cyan and magenta represent, respectively, constrained and Non-Incremental compilation with dynamic reordering. Datasets where the cyan dots are consistently over the magenta representing scenarios imposing X-constrainedness had an heavy impact on circuit size.*

Chapter 6

Conclusion

We've presented novel methods for Non-Incremental Probabilistic Answer Set Programming Knowledge Compilation, alongside theoretical results demonstrating their potential for efficient compilation. A key innovation in our approach lies in its Non-Incremental nature: we decompose the original program into disjoint subsets, compile each independently, and then conjoin their respective circuits to form the program's final representation. Furthermore, we've adapted bottom-up KC to effectively handle PASP-specific constraints, including cardinality constraints and probabilistic semantics.

Overall, our methods demonstrate potential for significant improvements in the efficiency and scalability of PASP inference, because it avoids introducing auxiliary variables during compilation. By moving beyond standard top-down pipelines and exploring alternative circuit compilation, we enable the generation of considerably more intricate circuits. This enhanced compilation capability, in turn, allows Neuro-Symbolic systems to encode more complex real-world constraints. This deep integration of neural perception with robust PASP reasoning can be further leveraged by recent advances in the encoding of circuits on GPUs (Maene *et al.*, 2024).

Possible future works include experimental validation of the unconstrained compilation approach proposed in this work, using recent techniques for re-structuring a circuit, (H. Zhang *et al.*, 2025), to trade-off flexibility during compilation with constraint enforcement after compilation for more efficient inference. Another possibly interesting direction is to explore the relation of the Non-Incremental methods with compositional ones, (Dal *et al.*, 2021), where the latter approaches the Weighted Model Counting (WMC) tasks by dividing the problem into smaller subproblems, similar in spirit to the Non-Incremental approach.

References

[Abío *et al.* 2012] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. "A new look at bdds for pseudo-boolean constraints". *Journal of Artificial Intelligence Research* 45 (2012), pp. 443–480 (cit. on pp. 31, 38).

[Akers 1978] Akers. "Binary decision diagrams". *IEEE Transactions on computers* 100.6 (1978), pp. 509–516 (cit. on p. 18).

[Alom *et al.* 2018] Md Zahangir Alom *et al.* "The history began from alexnet: a comprehensive survey on deep learning approaches". *arXiv preprint arXiv:1803.01164* (2018) (cit. on p. 1).

[Avila Garcez and Zaverucha 1999] Artur S Avila Garcez and Gerson Zaverucha. "The connectionist inductive learning and logic programming system". *Applied Intelligence* 11 (1999), pp. 59–77 (cit. on p. 25).

[Azzolini and Riguzzi 2024] Damiano Azzolini and Fabrizio Riguzzi. "Inference in probabilistic answer set programs with imprecise probabilities via optimization". In: *Proceedings of the Fortieth Conference on Uncertainty in Artificial Intelligence*. 2024, pp. 225–234 (cit. on pp. 60, 61).

[Bacchus *et al.* 2021] Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. "Maximum satisfiabiliy". In: *HandBook of satisfiability*. IOS Press, 2021, pp. 929–991 (cit. on p. 15).

[Bailleux and Boufkhad 2003] Olivier Bailleux and Yacine Boufkhad. "Efficient cnf encoding of boolean cardinality constraints". In: *International conference on principles and practice of constraint programming*. Springer. 2003, pp. 108–122 (cit. on p. 38).

[Baral *et al.* 2009] Chitta Baral, Michael Gelfond, and Nelson Rushton. "Probabilistic reasoning with answer sets". *Theory and Practice of Logic Programming* 9.1 (2009), pp. 57–144 (cit. on p. 51).

[Ben-Eliyahu-Zohary 2021] Rachel Ben-Eliyahu-Zohary. "How to split a logic program". *arXiv preprint arXiv:2109.08284* (2021) (cit. on p. 59).

[Bogaerts and Van den Broeck 2015] Bart Bogaerts and Guy Van den Broeck. "Knowledge compilation of logic programs using approximation fixpoint theory". *Theory and Practice of Logic Programming* 15.4-5 (2015), pp. 464–480 (cit. on p. 8).

[Brown *et al.* 2020] Tom B. Brown *et al. Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL] (cit. on p. 1).

[Bryant 1986] Randal E Bryant. "Graph-based algorithms for boolean function manipulation". *Computers, IEEE Transactions on* 100.8 (1986), pp. 677–691 (cit. on pp. 18, 19, 23).

[Chang *et al.* 2023] Yupeng Chang *et al.* "A survey on evaluation of large language models". *ACM Transactions on Intelligent Systems and Technology* (2023) (cit. on p. 1).

[Choi and Adnan Darwiche 2013] Arthur Choi and Adnan Darwiche. "Dynamic minimization of sentential decision diagrams". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 27. 1. 2013, pp. 187–194 (cit. on pp. 47, 48).

[Colnet 2023] Alexis de Colnet. "Separating Incremental and Non-Incremental Bottom-Up Compilation". In: *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*. Ed. by Meena Mahajan and Friedrich Slivovsky. Vol. 271. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 7:1–7:20. isbn: 978-3-95977-286-0. doi: 10.4230/LIPIcs.SAT.2023.7. url: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SAT.2023.7 (cit. on pp. 52, 54, 55).

[Cook 1971] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. New York, NY, USA: Association for Computing Machinery, 1971, pp. 151–158. isbn: 9781450374644. doi: 10.1145/800157.805047. url: https://doi.org/10.1145/800157.805047 (cit. on p. 15).

[Cozman and Mauá 2017] Fabio Gagliardi Cozman and Denis Deratani Mauá. "On the semantics and complexity of probabilistic logic programs". *Journal of Artificial Intelligence Research* 60 (2017), pp. 221–262 (cit. on pp. 11–14, 26, 41).

[Cozman and Mauá 2020] Fabio Gagliardi Cozman and Denis Deratani Mauá. "The joy of probabilistic answer set programming: semantics, complexity, expressivity, inference". *International Journal of Approximate Reasoning* 125 (2020), pp. 218–239 (cit. on pp. 10, 13, 41).

[Dal *et al.* 2021] Giso H Dal, Alfons W Laarman, Arjen Hommersom, and Peter JF Lucas. "A compositional approach to probabilistic knowledge compilation". *International Journal of Approximate Reasoning* 138 (2021), pp. 38–66 (cit. on p. 67).

[A. Darwiche and P. Marquis 2002] A. Darwiche and P. Marquis. "A knowledge compilation map". *Journal of Artificial Intelligence Research* 17 (Sept. 2002), pp. 229–264. ISSN: 1076-9757. DOI: 10.1613/jair.989. URL: http://dx.doi.org/10.1613/jair.989 (cit. on pp. 2, 15, 16, 24).

[Adnan Darwiche 2001] Adnan Darwiche. "Decomposable negation normal form". *Journal of the ACM (JACM)* 48.4 (2001), pp. 608–647 (cit. on p. 45).

[Adnan Darwiche *et al.* 2004] Adnan Darwiche *et al.* "New advances in compiling cnf to decomposable negation normal form". In: *Proc. of ECAI*. Citeseer. 2004, pp. 328–332 (cit. on pp. 25, 43, 44).

[Adnan Darwiche 2011] Adnan Darwiche. "Sdd: a new canonical representation of propositional knowledge bases". In: *Twenty-Second International Joint Conference on Artificial Intelligence*. 2011 (cit. on pp. 15, 19, 21–24, 28, 45, 47, 55).

[De Colnet 2023] Alexis De Colnet. "Separating incremental and non-incremental bottom-up compilation". In: *# PLACEHOLDER_PARENT_METADATA_VALUE#*. Vol. 271. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2023 (cit. on pp. 31, 32).

[De Raedt *et al.* 2007] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. "Problog: a probabilistic prolog and its application in link discovery". In: *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence*. IJCAI-INT JOINT CONF ARTIF INTELL. 2007, pp. 2462–2467 (cit. on pp. 26, 28).

[Eiter, Hecher, *et al.* 2021] Thomas Eiter, Markus Hecher, and Rafael Kiesel. "Treewidth-aware cycle breaking for algebraic answer set counting". In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*. Vol. 18. 1. 2021, pp. 269–279 (cit. on pp. 26, 30, 59).

[Eiter, Hecher, *et al.* 2024] Thomas Eiter, Markus Hecher, and Rafael Kiesel. "Aspmc: new frontiers of algebraic answer set counting". *Artificial Intelligence* 330 (2024), p. 104109. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2024.104109. URL: https://www.sciencedirect.com/science/Article/pii/S0004370224000456 (cit. on pp. 3, 25, 26, 30, 32, 44, 59).

[Eiter, Ianni, *et al.* 2009] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. *Answer set programming: A primer*. Springer, 2009 (cit. on pp. 5–8, 10, 11).

[Fandinno and Hecher 2023] Jorge Fandinno and Markus Hecher. "Treewidth-aware complexity for evaluating epistemic logic programs". In: *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*. 2023, pp. 3203–3211 (cit. on pp. 26, 30).

[Fierens *et al.* 2015] Daan Fierens *et al.* "Inference and learning in probabilistic logic programs using weighted boolean formulas". *Theory and Practice of Logic Programming* 15.3 (2015), pp. 358–401 (cit. on pp. 26, 28, 58).

[Franco and Martin 2009] John Franco and John Martin. "A history of satisfiability." *HandBook of satisfiability* 185 (2009), pp. 3–74 (cit. on p. 18).

[Garcez *et al.* 2022] Artur d'Avila Garcez *et al.* "Neural-symbolic learning and reasoning: a survey and interpretation". *Neuro-Symbolic Artificial Intelligence: The State of the Art* 342.1 (2022), p. 327 (cit. on p. 1).

[Gebser, Kaminski, Kaufmann, Ostrowski, *et al.* 2016] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, *et al.* "Theory solving made easy with clingo 5". In: *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2016 (cit. on pp. 32, 33).

[Gebser, Kaminski, Kaufmann, and Schaub 2014] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. "Clingo= asp+ control: preliminary report". *arXiv preprint arXiv:1405.3694* (2014) (cit. on pp. 32, 33).

[Geh *et al.* 2024] Renato Lui Geh, Jonas Gonçalves, Igor C Silveira, Denis D Mauá, and Fabio G Cozman. "Dpasp: a probabilistic logic programming environment for neurosymbolic learning and reasoning". In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*. Vol. 21. 1. 2024, pp. 731–742 (cit. on pp. 3, 11, 12, 25, 33, 58).

[Gong and Zhou 2017] Weiwei Gong and Xu Zhou. "A survey of sat solver". In: *AIP Conference Proceedings*. Vol. 1836. 1. AIP Publishing. 2017 (cit. on p. 33).

[Gunasekar *et al.* 2023] Suriya Gunasekar *et al. TextBooks Are All You Need*. 2023. arXiv: 2306.11644 [cs.CL] (cit. on p. 1).

[X. Huang *et al.* 2023] Xiaowei Huang *et al.* "A survey of safety and trustworthiness of large language models through the lens of verification and validation". *arXiv preprint arXiv:2305.11391* (2023) (cit. on p. 1).

[Kaminski *et al.* 2017] Roland Kaminski, Torsten Schaub, and Philipp Wanko. "A tutorial on hybrid answer set solving with clingo". *Reasoning Web. Semantic Interoperability on the Web: 13th International Summer School 2017, London, UK, July 7-11, 2017, Tutorial Lectures 13* (2017), pp. 167–203 (cit. on pp. 32, 33).

[Kiesel and Eiter 2023] Rafael Kiesel and Thomas Eiter. "Knowledge compilation and more with sharpsat-td". In: *Proceedings of the 20th IInternational Conference on Principles of Knowledge Representation and Reasoning*. IJCAI Organization. 2023, pp. 406–416 (cit. on p. 59).

[Kiesel, Totis, *et al.* 2022] Rafael Kiesel, Pietro Totis, and Angelika Kimmig. "Efficient knowledge compilation beyond weighted model counting". *Theory and Practice of Logic Programming* 22.4 (2022), pp. 505–522 (cit. on pp. 28, 29, 31, 33, 43, 44).

[Kimmig *et al.* 2017] Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. "Algebraic model counting". *Journal of Applied Logic* 22 (2017), pp. 46–62 (cit. on pp. 25, 40).

[Kisa *et al.* 2014] Doga Kisa, Guy Van den Broeck, Arthur Choi, and Adnan Darwiche. "Probabilistic sentential decision diagrams". In: *Fourteenth International Conference on the Principles of Knowledge Representation and Reasoning*. 2014 (cit. on pp. 15, 28).

[Korhonen and Järvisalo 2021] Tuukka Korhonen and Matti Järvisalo. "Integrating tree decompositions into decision heuristics of propositional model counters". In: *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2021, pp. 8–1 (cit. on p. 44).

[Kowalski 1979] Robert Kowalski. "Algorithm= logic+ control". *Communications of the ACM* 22.7 (1979), pp. 424–436 (cit. on p. 5).

[Kumar, Hooi, *et al.* 2018] Srijan Kumar, Bryan Hooi, *et al.* "Rev2: fraudulent user prediction in rating platforms". In: *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. ACM. 2018, pp. 333–341 (cit. on p. 61).

[Kumar, Spezzano, *et al.* 2016] Srijan Kumar, Francesca Spezzano, VS Subrahmanian, and Christos Faloutsos. "Edge weight prediction in weighted signed networks". In: *Data Mining (ICDM), 2016 IEEE 16th International Conference on*. IEEE. 2016, pp. 221–230 (cit. on p. 61).

[Lagniez and Pierre Marquis 2017] Jean-Marie Lagniez and Pierre Marquis. "An improved decision-dnnf compiler." In: *IJCAI*. Vol. 17. 2017, pp. 667–673 (cit. on p. 44).

[Lai, D. Liu, *et al.* 2017] Yong Lai, Dayou Liu, and Minghao Yin. "New canonical representations by augmenting obdds with conjunctive decomposition". *Journal of Artificial Intelligence Research* 58 (2017), pp. 453–521 (cit. on p. 44).

[Lai, Meel, *et al.* 2022] Yong Lai, Kuldeep S Meel, and Roland HC Yap. "Ccdd: a tractable representation for model counting and uniform sampling". *arXiv preprint arXiv:2202.10025* (2022) (cit. on p. 44).

[Lee and Lifschitz 2003] Joohyung Lee and Vladimir Lifschitz. "Loop formulas for disjunctive logic programs". In: *Logic Programming: 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003. Proceedings 19*. Springer. 2003, pp. 451–465 (cit. on pp. 3, 32, 34, 38).

[Levesque 1986] Hector J Levesque. "Knowledge representation and reasoning". *Annual review of computer science* 1.1 (1986), pp. 255–287 (cit. on p. 2).

[Levesque and Brachman 1987] Hector J Levesque and Ronald J Brachman. "Expressiveness and tractability in knowledge representation and reasoning 1". *Computational intelligence* 3.1 (1987), pp. 78–93 (cit. on p. 2).

[Levin 1973] Leonid Anatolevich Levin. "Universal sequential search problems". *Problemy peredachi informatsii* 9.3 (1973), pp. 115–116 (cit. on p. 15).

[Z. Li *et al.* 2023] Ziyang Li, Jiani Huang, and Mayur Naik. "Scallop: a language for neurosymbolic programming". *Proceedings of the ACM on Programming Languages* 7.PLDI (2023), pp. 1463–1487 (cit. on p. 33).

[Lifschitz and Turner 1994] Vladimir Lifschitz and Hudson Turner. "Splitting a logic program" (1994) (cit. on p. 59).

[A. Liu and Van den Broeck 2021] Anji Liu and Guy Van den Broeck. "Tractable regularization of probabilistic circuits". *Advances in Neural Information Processing Systems* 34 (2021), pp. 3558–3570 (cit. on p. 58).

[Y. Liu *et al.* 2023] Yang Liu *et al.* "Trustworthy llms: a survey and guideline for evaluating large language models' alignment". *arXiv preprint arXiv:2308.05374* (2023) (cit. on p. 1).

[Lukasiewicz 2005] Thomas Lukasiewicz. "Probabilistic description logic programs". In: *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*. Springer. 2005, pp. 737–749 (cit. on p. 13).

[Lukasiewicz 2007] Thomas Lukasiewicz. "Probabilistic description logic programs". *International Journal of Approximate Reasoning* 45.2 (2007), pp. 288–307 (cit. on p. 13).

[Maene *et al.* 2024] Jaron Maene, Vincent Derkinderen, and Pedro Zuidberg Dos Martires. "Klay: accelerating neurosymbolic ai". *arXiv preprint arXiv:2410.11415* (2024) (cit. on p. 67).

[Manhaeve, Dumancic, *et al.* 2018] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. "Deepproblog: neural probabilistic logic programming". *Advances in neural information processing systems* 31 (2018) (cit. on pp. 2, 26, 33).

[Manhaeve, Marra, *et al.* 2021] Robin Manhaeve, Giuseppe Marra, and Luc De Raedt. "Approximate inference for neural probabilistic logic programming". In: *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*. IJCAI Organization. 2021, pp. 475–486 (cit. on pp. 25, 58, 59).

[Marques-Silva and Lynce 2007] Joao Marques-Silva and Inês Lynce. "Towards robust cnf encodings of cardinality constraints". In: *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*. Springer. 2007, pp. 483–497 (cit. on p. 38).

[Mauá and Cozman 2020] Denis Deratani Mauá and Fabio Gagliardi Cozman. "Complexity results for probabilistic answer set programming". *International Journal of Approximate Reasoning* 118 (2020), pp. 133–154 (cit. on pp. 11, 58).

[Muise *et al.* 2012] Christian Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric Hsu. "DSHARP: Fast d-DNNF Compilation with sharpSAT". In: *Canadian Conference on Artificial Intelligence*. 2012 (cit. on p. 25).

[Onaka *et al.* 2025] Ryoma Onaka, Kengo Nakamura, Masaaki Nishino, and Norihito Yasuda. "An and-sum circuit with signed edges that is more succinct than sdd". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 39. 14. 2025, pp. 15100–15108 (cit. on p. 32).

[OpenAI *et al.* 2024] OpenAI *et al. GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL] (cit. on p. 1).

[Oztok *et al.* 2016] Umut Oztok, Arthur Choi, and Adnan Darwiche. "Solving pppp-complete problems using knowledge compilation." In: *KR*. 2016, pp. 94–103 (cit. on pp. 30, 47).

[Pajunen and Janhunen 2021] Jukka Pajunen and Tomi Janhunen. "Solution enumeration by optimality in answer set programming". *Theory and Practice of Logic Programming* 21.6 (2021), pp. 750–767 (cit. on pp. 2, 25, 58, 59).

[Papadimitriou 2003] Christos H Papadimitriou. "Computational complexity". In: *Encyclopedia of computer science*. 2003, pp. 260–265 (cit. on p. 16).

[Pipatsrisawat and Adnan Darwiche 2008] Knot Pipatsrisawat and Adnan Darwiche. "New compilation languages based on structured decomposability." In: *AAAI*. Vol. 8. 2008, pp. 517–522 (cit. on pp. 20, 24, 45).

[Qu and Tang 2019] Meng Qu and Jian Tang. "Probabilistic logic neural networks for reasoning". *Advances in neural information processing systems* 32 (2019) (cit. on p. 1).

[Radford, Narasimhan, *et al.* 2018] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, *et al.* "Improving language understanding by generative pre-training" (2018) (cit. on p. 1).

[Radford, J. Wu, *et al.* 2019] Alec Radford, Jeffrey Wu, *et al.* "Language models are unsupervised multitask learners". *OpenAI blog* 1.8 (2019), p. 9 (cit. on p. 1).

[Riegel *et al.* 2020] Ryan Riegel *et al.* "Logical neural networks". *arXiv preprint arXiv:2006.13155* (2020) (cit. on p. 1).

[Rudell 1993] Richard Rudell. "Dynamic variable ordering for ordered binary decision diagrams". In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. IEEE. 1993, pp. 42–47 (cit. on p. 47).

[Ruff and Pappu 2021] Kiersten M Ruff and Rohit V Pappu. "Alphafold and implications for intrinsically disordered proteins". *Journal of molecular biology* 433.20 (2021), p. 167208 (cit. on p. 1).

[Sarker *et al.* 2021] Md Kamruzzaman Sarker, Lu Zhou, Aaron Eberhart, and Pascal Hitzler. "Neuro-symbolic artificial intelligence". *AI Communications* 34.3 (2021), pp. 197–209 (cit. on p. 1).

[Sato 1995] Taisuke Sato. "A statistical learning method for logic programs with distribution semantics" (1995) (cit. on p. 10).

[Shterionov *et al.* 2015] Dimitar Shterionov *et al.* "The most probable explanation for probabilistic logic programs with annotated disjunctions". In: *Inductive Logic Programming: 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers*. Springer. 2015, pp. 139–153 (cit. on pp. 33, 38).

[Somenzi 1998] Fabio Somenzi. "Cudd: cu decision diagram package release 2.3. 0". *University of Colorado at Boulder* 621 (1998) (cit. on p. 44).

[Sun *et al.* 2024] Lichao Sun *et al.* "Trustllm: trustworthiness in large language models". *arXiv preprint arXiv:2401.05561* (2024) (cit. on p. 1).

[Toda and Soh 2016] Takahisa Toda and Takehide Soh. "Implementing efficient all solutions sat solvers". *Journal of Experimental Algorithmics (JEA)* 21 (2016), pp. 1–44 (cit. on p. 33).

[Totis, Davis, *et al.* 2023] Pietro Totis, Jesse Davis, Luc De Raedt, and Angelika Kimmig. "Lifted reasoning for combinatorial counting". *Journal of Artificial Intelligence Research* 76 (2023), pp. 1–58 (cit. on p. 25).

[Totis, De Raedt, *et al.* 2023] Pietro Totis, Luc De Raedt, and Angelika Kimmig. "Smproblog: stable model semantics in problog for probabilistic argumentation". *Theory and Practice of Logic Programming* 23.6 (2023), pp. 1198–1247 (cit. on pp. 3, 26, 28, 49, 51, 52).

[valiant 1979] l.g. valiant. "The complexity of computing the permanent". *theoretical computer science* 8.2 (1979), pp. 189–201. issn: 0304-3975. doi: https://doi.org/10.1016/0304-3975(79)90044-6. url: https://www.sciencedirect.com/science/Article/pii/0304397579900446 (cit. on p. 15).

[Van den Broeck *et al.* 2021] Guy Van den Broeck, Kristian Kersting, Sriraam Natarajan, and David Poole. *An Introduction to Lifted Probabilistic Inference*. MIT Press, 2021 (cit. on p. 25).

[Van Gelder *et al.* 1991] Allen Van Gelder, Kenneth A Ross, and John S Schlipf. "The well-founded semantics for general logic programs". *Journal of the ACM (JACM)* 38.3 (1991), pp. 619–649 (cit. on p. 8).

[Vlasselaer, Renkens, *et al.* 2014] Jonas Vlasselaer, Joris Renkens, Guy Van den Broeck, and Luc De Raedt. "Compiling probabilistic logic programs into sentential decision diagrams". In: *Proceedings Workshop on Probabilistic Logic Programming (PLP)*. 2014, pp. 1–10 (cit. on pp. 3, 27, 31, 39, 48, 60).

[Vlasselaer, Van den Broeck, *et al.* 2016] Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. "Tp-compilation for inference in probabilistic logic programs". *International Journal of Approximate Reasoning* 78 (2016), pp. 15–32 (cit. on pp. 3, 27, 59).

[B. Wang *et al.* 2025] Benjie Wang, Denis Deratani Mauá, Guy Van den Broeck, and YooJung Choi. *A Compositional Atlas for Algebraic Circuits*. 2025. arXiv: 2412.05481 [cs.AI]. url: https://arxiv.org/abs/2412.05481 (cit. on pp. 3, 29–31, 40, 43).

[F.-Y. Wang *et al.* 2016] Fei-Yue Wang *et al.* "Where does alphago go: from church-turing thesis to alphago thesis and beyond". *IEEE/CAA Journal of Automatica Sinica* 3.2 (2016), pp. 113–120 (cit. on p. 1).

[S. Wang *et al.* 2020] Shaobo Wang *et al.* "Provenance for probabilistic logic programs". In: *International Conference on Extending Database Technology (EDBT)*. 2020 (cit. on p. 61).

[Sean Wu *et al.* 2023] Sean Wu *et al.* "A comparative study of open-source large language models, gpt-4 and claude 2: multiple-choice test taking in nephrology". *arXiv preprint arXiv:2308.04709* (2023) (cit. on p. 1).

[Z. Yang *et al.* 2023] Zhun Yang, Adam Ishay, and Joohyung Lee. "Neurasp: embracing neural networks into answer set programming". *arXiv preprint arXiv:2307.07700* (2023) (cit. on pp. 2, 25, 33, 58).

[H. Zhang *et al.* 2025] Honghua Zhang, Benjie Wang, Marcelo Arenas, and Guy Van den Broeck. *Restructuring Tractable Probabilistic Circuits*. 2025. arXiv: 2411.12256 [cs.AI]. url: https://arxiv.org/abs/2411.12256 (cit. on pp. 49, 52, 64, 67).