Informatics

Evaluation Techniques for Algebraic Answer Set Counting Over Idempotent Semirings

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Martin Ritter
Matrikelnummer 01635598

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter
Mitwirkung: Rafael Kiesel, MSc

Wien, 17. Mai 2022

Martin Ritter   Thomas Eiter

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Evaluation Techniques for Algebraic Answer Set Counting Over Idempotent Semirings

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Martin Ritter
Registration Number 01635598

to the Faculty of Informatics

at the TU Wien

Advisor: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter
Assistance: Rafael Kiesel, MSc

Vienna, 17th May, 2022

Martin Ritter Thomas Eiter

Erklärung zur Verfassung der Arbeit

Martin Ritter

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 17. Mai 2022

Martin Ritter

Acknowledgements

I would hereby like to express my gratitude towards everyone who had a part in this thesis. First an foremost, my supervisors Thomas Eiter and Rafael Kiesel for their advice, guidance, and expertise. Working on this thesis was a great experience, which surely had to do with the exceptional supervision. I am particularly thankful for the great amount of help I got when encountering technical difficulties while setting up the benchmark tests.

I would also like to thank my friends and my family, without whom this would not have been possible. Having friends and family you can always rely on makes everything in life easier.

Kurzfassung

Algebraic Answer Set Counting (AASC) eignet sich zum Lösen einer Vielzahl von Problemen wie z.B. probabilistisches Schließen, präferenzielles Schließen, und Optimisierungsprobleme. Dazu wird eine Erweiterung von Answer Set Programming (ASP) verwendet, in der über einem Semiring Gewichtungen für Answer Sets berechnet werden. Die Evaluierung von AASC erfordert einen hohen Rechenaufwand (Komplexitätsklasse #P/OptP/NP-schwer, je nach verwendetem Semiring). Daher besteht für AASC besonderes Interesse darin, effiziente Evaluierungsansätze zu finden.

Ausgehend von einem Evaluierungsansatz, in dem AASC auf Algebraic Model Counting (AMC) reduziert wird, präsentieren wir mehrere Modifikationen dieses Ansatzes mit dem Ziel, die Effizienz zu verbessern. Wir fokussieren uns dabei insbesondere auf den Spezialfall von AASC über idempotenten Semiringen, für den wir beweisen, dass zusätzliche Cycle-Breaking-Algorithmen anwendbar sind, die im allgemeinen Fall von beliebigen Semiringen nicht verwendet werden können. Für den allgemeinen Fall beweisen wir weiters die Anwendbarkeit einer Preprocessing-Technik für AMC namens `B+E`, bei der Defined-Variables durch Variable-Forgetting eliminiert werden. Zusätzlich beschreiben wir ein abgeänderte Version für idempotente Semiringe, bei der neben Defined-Variables noch weitere Variablen eliminiert werden.

Die theoretischen Resultate bezüglich des Preprocessings setzten wir in die Praxis um, indem wir den AASC-Solver `aspmc` um die beschriebenen Preprocessing-Techniken erweitern. Dafür adaptieren wir eine existierende Implementation von `B+E`. Unsere Experimente zur Evaluierung der Performance des Preprocessors zeigen, dass durch dieses Feature die benötigte Zeit für die nachfolgenden Evaluierungsschritte (Knowledge-Compilation und Counting) reduziert wird. Allerdings erhöht sich die Gesamtzeit für die Evaluierung, aufgrund der zusätzlichen Zeit, die für das Preprocessing selbst benötigt wird.

Abstract

Algebraic Answer Set Counting (AASC) is a reasoning task that has recently gained interest. It is defined over an extension of Answer Set Programming (ASP) with weights over semirings, called ASP with algebraic measures. What makes AASC particularly interesting is that it can be used to model a variety of different problems, such as probabilistic reasoning, preferential reasoning, and optimization problems. At the same time, AASC is a computationally hard task (#P/OptP/NP-hard, depending on the semiring). Therefore, one of the goals in current research is to find efficient evaluation techniques for this task.

Starting with an evaluation approach that reduces AASC to Algebraic Model Counting (AMC), we propose modifications intended to improve efficiency. We focus in particular on the special case of AASC over idempotent semirings, for which we show the applicability of alternative cycle breaking algorithms that are not applicable in the general case. We establish further theoretical results regarding the preprocessing of AMC instances. Here, we first show, for the general case of arbitrary semirings, the applicability of the preprocessor `B+E`, which uses variable forgetting to eliminate defined variables. Then we modify the technique for idempotent semirings so that further variables, i.e. not just defined ones, are eliminated.

We put our theoretical results regarding preprocessing to practical use by adding the proposed preprocessor to the AASC solver `aspmc`. For this we adapt an existing implementation of `B+E`. Our experiments for evaluating the performance of the preprocessor show that it reduces the time for the evaluation steps that follow, which are knowledge compilation and counting. However, the total time needed to solve instances may increase due to the additional time needed for the preprocessing itself.

Contents

CHAPTER 1
Introduction

In this introduction a short overview on the topic of this thesis is given. After some background information, we motivate the topic with a practical example. Then we define the scope of this thesis and state concrete objectives. Furthermore, we provide an overview on literature related to this thesis.

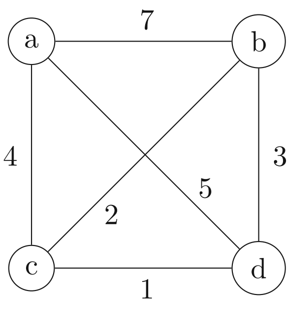
1.1 Background

Answer Set Programming (ASP) (Brewka et al. 2011; Eiter et al. 2009) is a declarative problem solving paradigm, where problems are encoded as non-monotonic logic programs. An answer set program consists of rules that describe the problem instance, rather than a concrete algorithm that computes the solutions. Such a program can be submitted to an ASP solver, which then outputs the solutions (answer sets) of the program.

Algebraic Answer Set Counting (AASC) (Eiter et al. 2021) is a task that generalizes a variety of problems, such as e.g. probabilistic reasoning, preferential reasoning, or optimization problems. AASC is defined on an extension of Answer Set Programming (ASP) (Eiter et al. 2009; Brewka et al. 2011), called ASP with algebraic measures. In this extension, a weight is calculated for each answer set of a given logic program; AASC then amounts to summing up the weights of some or all of the answer sets of the program. Depending on the problem that is modeled, the weights may represent different concepts, such as e.g. probabilities, cost, etc.

The semantics is highly general due to being defined for arbitrary semirings. We recall that a semiring $\mathcal{R} = (R, \oplus, \otimes, e_\oplus, e_\otimes)$ is a structure with operators for addition ($\oplus$) and multiplication ($\otimes$) over a set R, with the respective identity elements $e_\oplus$ and $e_\otimes$, satisfying a characteristic set of axioms. For the natural numbers $\mathbb{N}$, we get answer set counting as a special case of AASC.

Figure 1.1: A TSP instance



As the title of this thesis indicates, we intend to find efficient evaluation techniques for AASC over idempotent semirings. Idempotent semirings are a subclass of semirings where adding a value to itself results in the same value, i.e. $\forall r \in R : r \oplus r = r$ holds.

1.2 Motivation

AASC instances are in general hard to solve (Eiter and Kiesel 2021). Thus there is a rising interest in finding efficient evaluation techniques. We want to contribute to this with improved evaluation techniques for the special case of idempotent semirings. There are various idempotent semirings that are useful for modeling a variety of problems (Friesen and Domingos 2016; Kimmig et al. 2017; Larrosa et al. 2010).

For example, the semiring $(\mathbb{R}_+, \max, \cdot, 0, 1)$ can be used for Most Probable Explanation (MPE) inference (Pearl 1988), a probabilistic reasoning task. For MPE inference, one is given some evidence (truth values of some variables), and the task is to find the most likely interpretation of the non-evidence variables.

$\mathcal{R}_{\max} = (\mathbb{N} \cup \{-\infty\}, \max, +, -\infty, 0)$ and $\mathcal{R}_{\min} = (\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$, called the tropical semirings, are useful for modeling optimization problems, such as e.g. $\mathcal{R}_{max}$ for MAX-SAT. Another well-known example for an optimization problem is the Traveling Salesman Problem (TSP). This problem has as input a list of cities with distances for each pair of cities. The task is to find the shortest route through all cities, visiting no city twice and returning at the starting city. This amounts to finding the shortest path in an undirected weighted graph, with the first and last vertex being the same and every other vertex occurring exactly once. The following is an example for how AASC can be used to solve a TSP instance.

Example 1. Consider the TSP instance depicted in Figure 1.1. We can write a program Π where first the edges are guessed and then all models are eliminated, where not all vertices are visited exactly once.

Guessing the edges:

$$\begin{aligned}
&\mathrm{vertex}(a)\\
&\mathrm{vertex}(b)\\
&\mathrm{vertex}(c)\\
&\mathrm{vertex}(d)\\
&\{\mathrm{edge}(X,Y)\} \leftarrow \mathrm{vertex}(X), \mathrm{vertex}(Y), X \neq Y.
\end{aligned}$$

Eliminating unwanted models:

$$\begin{aligned}
&\mathrm{start}(a)\\
&\mathrm{visited}(Y) \leftarrow \mathrm{edge}(X,Y), \mathrm{start}(X)\\
&\mathrm{visited}(Y) \leftarrow \mathrm{edge}(X,Y), \mathrm{visited}(X)\\
&\leftarrow \mathrm{vertex}(X), \mathrm{not\ visited}(X)\\
&\leftarrow \mathrm{edge}(X,Y), \mathrm{edge}(X,Z), Y \neq Z\\
&\leftarrow \mathrm{edge}(X,Y), \mathrm{edge}(Z,Y), X \neq Z.
\end{aligned}$$

We then choose the semiring $\mathcal{R}_{\min} = (\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$ and define the following weights:

$$\begin{aligned}
\beta(\mathrm{edge}(a,b)) &= \beta(\mathrm{edge}(b,a)) = 7\\
\beta(\mathrm{edge}(a,c)) &= \beta(\mathrm{edge}(c,a)) = 4\\
\beta(\mathrm{edge}(a,d)) &= \beta(\mathrm{edge}(d,a)) = 5\\
\beta(\mathrm{edge}(b,c)) &= \beta(\mathrm{edge}(c,b)) = 2\\
\beta(\mathrm{edge}(b,d)) &= \beta(\mathrm{edge}(d,b)) = 3\\
\beta(\mathrm{edge}(c,d)) &= \beta(\mathrm{edge}(d,c)) = 1.
\end{aligned}$$

The weight of one single answer set is calculated by "multiplying" the weights of all edges. However, since in the case of $\mathcal{R}_{\min}$, $+$ is used as the multiplication operator, this actually amounts to summing up the edge weights. Therefore, the weight of one answer set is the distance of the route it represents. Similarly, "summing up" the weights of all answer sets amounts to taking the minimum, since *min* is used as the addition operator. This gives us as a result the distance of the smallest route.

As this example shows, there are still relevant applications of idempotent semirings. What makes this restricted variant interesting is that in terms of evaluation the restriction can be exploited in order to increase efficiency. Namely, for evaluation over idempotent semirings there are weaker requirements in terms of preserving models, as compared to the general case.

1.3 Research Focus

There are multiple approaches to evaluating AASC. We focus on one that reduces an AASC instance to an instance of Algebraic Model Counting (AMC) (Kimmig et al. 2017). The main difference between these two problems is that the AMC instance has a propositional theory as part of the input instead of an answer set program. Our overall aim is to modify this approach according to the weakened requirements for idempotent semirings.

We focus on two particular steps in the evaluation. The first one is cycle breaking, which has the purpose of removing cyclic dependencies in the input program. The second one is preprocessing of the AMC instance. For this step we first show the applicability of a preprocessing technique for general AASC. We then modify it to obtain an improved version for idempotent AASC.

1.4 Contributions

Concretely, we make the following contributions:

- We show that the cycle breaking algorithms by Hecher (2020) and Lin and Zhao (2003) are applicable in the evaluation of AASC over idempotent semirings. Both algorithms are not applicable in the general case.
- We show that the preprocessing technique `B+E` by Lagniez et al. (2020) is applicable in the evaluation of AASC in the general case.
- We modify `B+E` so that further variables are eliminated. We then show that this modified version is still applicable for idempotent semirings.
- Based on an existing implementation of `B+E`, we add both versions of the preprocessor (general and idempotent) to the `aspmc` solver (Eiter et al. 2021).
- We perform empirical experiments (benchmarks) to assess the impact of the preprocessor on the performance of `aspmc`.

The results of the benchmarks show that preprocessing does reduce the time needed for the subsequent evaluation steps (knowledge compilation and counting). However, preprocessing itself takes too much time and as a consequence enabling preprocessing increases the total time. Thus, while the preprocessor looks promising, the current implementation does not have a beneficial effect yet.

1.5 Related Work

The idea of AASC has been introduced by Eiter and Kiesel (2020) as weighted LARS in the context of stream reasoning. Eiter et al. (2021) then consider algebraic measures for just ASP, without the formalism for stream reasoning. Problog (De Raedt et al. 2007), and especially its algebraic extension (Kimmig et al. 2011), is a similar formalism. Problog itself is a probabilistic extension of the logic programming language Prolog. Compared to AASC as described by Eiter et al. (2021) there are a few differences. First, Problog does not support the full syntax of ASP (in essence there is no negation in Problog, only guesses). Second, it only allows so-called factorized measures, where answer set weights are computed as a product of the values of the literals.

The idea of the evaluation approach that we consider has been described by Fierens et al. (2015) for Problog. There, inference on Problog programs is reduced to Weighted Model Counting (WMC), of which AMC is a generalization. According to Eiter et al. (2021) the approach of reducing AASC to AMC has certain advantages as compared to some alternatives: first, with clingo (Gebser et al. 2014), answer set counting can be done by enumeration, which is only feasible if the number of answer sets is not too high. Second, the dynASP2.5 solver (Fichte et al. 2021) uses an algorithm that is only feasible for programs with very low treewidth.

One of the aspects that separates our work from the work mentioned above is that we specifically consider the case of idempotent semirings. Kimmig et al. (2017) mention some examples of AMC tasks over idempotent semirings and also consider idempotence in the evaluation, but only for AMC and not AASC.

In the translation to propositional logic, we primarily focus on cycle breaking. Cycle breaking has been studied for plain ASP and one can, in general, use existing algorithms from the literature (Janhunen 2003; Hecher 2020; Lin and Zhao 2003). An additional requirement for AASC is that models are preserved bijectively (like e.g. by Janhunen's method), which is not necessarily true for all cycle breaking algorithms. Examples for this are the two cycle breaking algorithms by Hecher (2020) and by Lin and Zhao (2003) described later in Chapter 4.

Eiter et al. (2021) did not use an existing algorithm but introduced a new cycle breaking method that is inspired by $T_{\mathcal{P}}$-compilation (Vlasselaer et al. 2016). Its advantage is that it only slightly increases the program's treewidth. The treewidth of a program is an important parameter that gives performance guarantees for AASC.

We are not aware of any work on preprocessing for AMC. The preprocessor that we make use of (Lagniez et al. 2020) has been designed for model counting and uses the concepts of definability and variable forgetting. Definability and variable forgetting in propositional logic have been extensively studied in the past, cf. Lin and Reiter (1994), Lang et al. (2003), Lang and Marquis (2008), and Su et al. (2009).

CHAPTER 2

Preliminaries

This chapter introduces basic concepts relevant to this thesis. Answer Set Programming (ASP) (Brewka et al. 2011; Eiter et al. 2009) is the foundation that we start with in Section 2.1. It is a declarative problem solving paradigm and is well-suited for modeling a multitude of different types of problems. There exist numerous extensions of the core language. They serve the purpose of increasing expressiveness and/or allowing for more convenient ways of representing certain problems. One such extension is ASP with algebraic measures (Eiter et al. 2021). In this extension a weight, defined over a semiring (Section 2.2), is assigned to each answer set. Section 2.3 introduces Algebraic Answer Set Counting (AASC) which is performed on ASP with algebraic measures. For all necessary definitions we follow the notation from Eiter et al. (2021).

We assume the reader is familiar with the basics of propositional logic. In Section 2.4 we present a brief recap of the concepts necessary for this thesis. For a more comprehensive overview see e.g. Kleine Büning and Lettmann (1999) and Enderton (2001).

2.1 Answer Set Programming

The idea of answer set programming is to encode problems as non-monotonic logic programs. The solutions of such a program are called answer sets. They represent the solutions of the described problem.

Definition 1 (Normal Answer Set Program)**.** A *(normal) answer set program* Π is a finite set of rules

$$a \leftarrow b_1, ..., b_m, \text{not } c_1, ..., \text{not } c_n$$

where $a, b_1, ..., b_m, c_1, ..., c_n$ are propositional variables (i.e. propositional atoms). The *head* of a rule is defined as $H(r) := a$, the *positive/negative body* as $B^+(r) := \{b_1, ..., b_m\}$ and $B^-(r) := \{c_1, ..., c_n\}$ respectively. The set of propositional variables occurring in Π is denoted by $\mathit{Var}(\Pi)$.

The negation "not" is called *default negation*. The negative body of a rule r is satisfied in the absence of evidence for any variable occurring in $B^-(r)$. The use of default negation can lead to non-monotonic behavior, i.e. solutions may be retracted under additional evidence.

Example 2. Let Π_1 be the answer set program consisting of the following rules:

$$a \leftarrow \text{not } b$$
$$b \leftarrow \text{not } a$$
$$c \leftarrow a$$

This program models a choice between a and b, where in the case of a being true, c is also true.

2.1.1 Semantics

For the semantics of answer set programs, we first define under what circumstances an interpretation satisfies a program.

Definition 2 (Interpretation)**.** An *interpretation* of a program Π is a subset of $\mathit{Var}(\Pi)$.

Definition 3 (Satisfaction)**.** An interpretation $\mathcal{I}$ *satisfies*

- the head $H(r)$ of a rule r if $H(r) \in \mathcal{I}$,
- the positive body $B^+(r)$ of a rule r if $B^+(r) \subseteq \mathcal{I}$,
- the negative body $B^-(r)$ of a rule r if $B^-(r) \cap \mathcal{I} = \emptyset$,
- a rule r if $\mathcal{I}$ either satisfies $H(r)$ or does not satisfy both $B^+(r)$ and $B^-(r)$, and
- a program Π if it satisfies every rule $r \in \Pi$.

If an interpretation $\mathcal{I}$ satisfies a program, for every rule whose body is satisfied, its head is already in $\mathcal{I}$. In other words, from the variables in $\mathcal{I}$ no further variables can be derived.

Example 3. The program Π_1 from Example 2 is satisfied by $\{a, b, c\}$, $\{a, c\}$, $\{b, c\}$, and $\{b\}$, but is not satisfied by $\{a, b\}$, $\{a\}$, and $\emptyset$.

We want models to only contain variables that are necessarily true. A positive (i.e. no default negation) answer set program Π has an unique $\subseteq$-minimal satisfying interpretation, called the *least model* of Π, or $LM(\Pi)$. For programs with default negation we need the concept of the Gelfond-Lifschitz reduct (Gelfond and Lifschitz 1988). The reduct of a program is a positive program. Negation is removed according to a candidate interpretation.

Definition 4 (Reduct)**.** The *reduct* $\Pi^{\mathcal{I}}$ of a program Π w.r.t. an interpretation $\mathcal{I}$ is obtained by

1. removing any rule r for which $B^-(r) \cap \mathcal{I} \neq \emptyset$
2. removing the negative body from the remaining rules

Example 4. Consider the program Π_1 from Example 2. Let $\mathcal{I}_1 := \{a, b\}$ and $\mathcal{I}_2 := \{a, c\}$. Then $\Pi_1^{\mathcal{I}_1}$:

$$c \leftarrow a$$

and $\Pi_1^{\mathcal{I}_2}$:

$$\begin{gathered} a \\ c \leftarrow a \end{gathered}$$

If the candidate interpretation coincides with the least model of the reduct then it is an answer set of the program.

Definition 5 (Answer Set)**.** An interpretation $\mathcal{I}$ is an *answer set* of Π if $\mathcal{I} = LM(\Pi^{\mathcal{I}})$. The set of all answer sets of Π is denoted by $\mathcal{AS}(\Pi)$.

Example 5. The least model of $\Pi_1^{\mathcal{I}_1}$ is $\{\}$, which is not equal to $\mathcal{I}_1$. The least model of $\Pi_1^{\mathcal{I}_2}$ is $\{a, c\}$, which is equal to $\mathcal{I}_2$. Therefore $\mathcal{I}_2$ is an answer set of Π_1 while $\mathcal{I}_1$ is not.

2.2 Semiring

We first recall the definition of a monoid.

Definition 6. A *monoid* $\mathcal{R} = (R, \circ, e)$ consists of a non-empty set R and a binary operator $\circ$, where

- $\circ$ is associative, i.e. $\forall a, b, c \in R : (a \circ b) \circ c = a \circ (b \circ c)$, and
- e is an identity element, i.e. $\forall r \in R : e \circ r = r = r \circ e$.

Furthermore, a monoid is *commutative* if $(R, \circ)$ is commutative, i.e. $\forall a, b \in R : a \circ b = b \circ a$.

A semiring is an algebraic structure defined as follows.

Definition 7 (Semiring)**.** A *semiring* $\mathcal{R} = (R, \oplus, \otimes, e_{\oplus}, e_{\otimes})$ consists of a non-empty set R and two binary operators $\oplus$ and $\otimes$, called *addition* and *multiplication* respectively, where

- $(R, \oplus, e_\oplus)$ is a commutative monoid,
- $(R, \otimes, e_\otimes)$ is a monoid,
- multiplication left and right distributes over addition, i.e. $\forall a, b, c \in R : a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$, and
- $e_\oplus$ annihilates R, i.e. $\forall r \in R : r \otimes e_\oplus = e_\oplus = e_\oplus \otimes r$.

Furthermore, a semiring is *commutative* if $(R, \otimes)$ is commutative, and is *idempotent* if $\forall r \in R : r \oplus r = r$.

Example 6. (Cf. Eiter et al. 2021). Some common examples of semirings:

- $\mathbb{F} = (\mathbb{F}, +, \cdot, 0, 1)$, for $\mathbb{F} \in \{\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$, the semiring of numbers in $\mathbb{F}$ with addition and multiplication,
- $\mathbb{B} = (\{0, 1\}, \vee, \wedge, 0, 1)$, the Boolean semiring,
- $\mathcal{R}_{\max} = (\mathbb{N} \cup \{-\infty\}, \max, +, -\infty, 0)$, the max-plus semiring, and
- $\mathcal{P} = ([0, 1], +, \cdot, 0, 1)$, the probability semiring.

Out of these examples, $\mathbb{B}$ and $\mathcal{R}_{\max}$ are idempotent semirings.

2.3 Algebraic Answer Set Counting

Algebraic Answer Set Counting (AASC)(Eiter et al. 2021) is a task that is performed on an extension of ASP, called ASP with algebraic measures. In this extension, a weight is assigned to each answer set of a given logic program; AASC then amounts to summing up the weights of the answer sets of the program. The weight of an answer set is determined by a weighted propositional formula. Calculation of the weights and summing up answer set weights are generalized from the natural numbers to arbitrary semirings.

Definition 8 (Weighted Propositional Logic). Let $\mathcal{R} = (R, \oplus, \otimes, e_\oplus, e_\otimes)$ be a commutative semiring. A *weighted formula* α over $\mathcal{R}$ is defined as

$$\alpha ::= k \mid v \mid \neg v \mid \alpha + \alpha \mid \alpha * \alpha$$

where $k \in R$ and v is a propositional variable. The semantics of α w.r.t. some interpretation $\mathcal{I}$ is denoted by $[\![\alpha]\!]_\mathcal{R}(\mathcal{I})$ and is defined as

$$\begin{aligned}
[\![k]\!]_\mathcal{R}(\mathcal{I}) &= k, \\
[\![l]\!]_\mathcal{R}(\mathcal{I}) &= \begin{cases} e_\otimes & l = v, v \in \mathcal{I} \text{ or } l = \neg v, v \notin \mathcal{I}, \\ e_\oplus & \text{otherwise,} \end{cases} \\
[\![\alpha_1 + \alpha_2]\!]_\mathcal{R}(\mathcal{I}) &= [\![\alpha_1]\!]_\mathcal{R}(\mathcal{I}) \oplus [\![\alpha_2]\!]_\mathcal{R}(\mathcal{I}), \\
[\![\alpha_1 * \alpha_2]\!]_\mathcal{R}(\mathcal{I}) &= [\![\alpha_1]\!]_\mathcal{R}(\mathcal{I}) \otimes [\![\alpha_2]\!]_\mathcal{R}(\mathcal{I}).
\end{aligned}$$

Definition 9 (Algebraic Measure)**.** An *algebraic measure* $\mu = \langle \Pi, \alpha, \mathcal{R} \rangle$ consists of an answer set program Π, together with a weighted formula α over a semiring $\mathcal{R}$. The weight of an answer set $\mathcal{I} \in \mathcal{AS}(\Pi)$ is denoted by $\mu(\mathcal{I})$ and defined as

$$\mu(\mathcal{I}) := [\![\alpha]\!]_{\mathcal{R}}(\mathcal{I}).$$

A query $\mu(a)$ for $a \in \mathit{Var}(\Pi)$ is defined as

$$\mu(a) := \bigoplus_{\mathcal{I} \in \mathcal{AS}(\Pi), a \in \mathcal{I}} \mu(\mathcal{I}).$$

The task of evaluating such a query is called *algebraic answer set counting*.

Example 7. Recall Π_1 from Example 2. The rules $a \leftarrow \text{not } b$ and $b \leftarrow \text{not } a$ represent a choice between either a or b. Using an algebraic measure, we can assign probabilities to these two variables. Let $\mu_1 = \langle \Pi_1, \alpha_1, \mathcal{P} \rangle$, where $\alpha_1 := 0.3 \cdot a + 0.7 \cdot \neg a$. Π_1 has two answer sets, $\{a, c\}$ and $\{b\}$, with weights $\mu_1(\{a, c\}) = 0.3$ and $\mu_1(\{b\}) = 0.7$. The result of the query $\mu_1(c)$, i.e. the probability of c being true, is 0.3.

One special case of algebraic measures are factorized measures, where the answer set weights can be defined as the product of literal weights.

Definition 10 (Factorized Measure)**.** Let $\mu = \langle \Pi, \alpha, \mathcal{R} \rangle$ be an algebraic measure and let $F \subseteq \mathit{Var}(\Pi)$. Then μ is *factorized* w.r.t. F if there is a weight function

$$\beta : F \cup \{\neg f | f \in F\} \to \mathcal{R}$$

s.t.

$$\mu(\mathcal{I}) = \bigotimes_{f \in \mathcal{F} \cap \mathcal{I}} \beta(f) \otimes \bigotimes_{f \in \mathcal{F} \setminus \mathcal{I}} \beta(\neg f)$$

for all $\mathcal{I} \in \mathcal{AS}(\Pi)$

Example 8. The measure μ_1 from Example 7 is factorized w.r.t. $\{a\}$, by letting $\beta(a) = 0.3$ and $\beta(\neg a) = 0.7$.

Eiter et al. (2021) showed that for any algebraic measure, an equivalent factorized algebraic measure, along with the corresponding weight function β, can be constructed in linear time. Therefore, we can in the context of this thesis assume that we are always dealing with factorized measures.

From now on we refer to AASC over idempotent semirings as *idempotent AASC*, to distinguish it from the general case with arbitrary semirings which we call *general AASC*.

We refer to all variables that contribute to the calculation of the model weights as *contributing variables*. These are

- the query variable and

- the variables occurring in the weighted formula.

For factorized measures we can further restrict the set of contributing variables to those that have unequal weights for the positive and negative literal. This also includes the query variable, since it can be thought of as having weight $e_\otimes$ for the positive, weight $e_\oplus$ for the negative literal.

Note that in order to determine the result of a query $\mu(a)$ to $\langle \Pi, \alpha, \mathcal{R} \rangle$ it is sufficient to know the values of all contributing variables for all models of Π.

Example 9. In Example 7, for the query $\mu_1(c)$, both a and c are contributing variables. For the query $\mu_1(a)$, only a is a contributing variables.

For the rest of this thesis we assume, for factorized measures $\langle \Pi, \beta, \mathcal{R} \rangle$, that the query variable is encoded into the weight function (using $e_\otimes$ and $e_\oplus$), and we refer to $\langle \Pi, \beta, \mathcal{R} \rangle$ as an *AASC instance*. The result is computed by summing up *all* answer set weights.

2.4 Propositional Logic

A *propositional formula* is built from *propositional variables* and the *logical connectives* $\wedge$, $\vee$, and $\neg$, called *conjunction*, *disjunction*, and *negation* respectively. A *literal* is a propositional variable v or its negation $\neg v$. A *clause* is a disjunction $l_1 \vee ... \vee l_k$, $k \geq 1$ of literals l_i. A formula is in *Conjunctive Normal Form (CNF)* if it is a conjunction $c_1 \wedge ... \wedge c_m$, $m \geq 1$ of clauses c_i.

Let φ be a propositional formula. The set of *immediate subformulas* of φ is defined as follows:

- if $\varphi = \neg\varphi_1$ then φ_1 is an immediate subformula,
- if $\varphi = \varphi_1 \wedge \varphi_2$ or $\varphi = \varphi_1 \vee \varphi_2$ then φ_1 and φ_2 are immediate subformulas.

The set of *subformulas* is defined as the reflexive transitive closure of the immediate subformulas.

A *substitution* is defined as simultaneously replacing in a formula φ every occurrence of a variable x by a formula ψ and the result is denoted by $\varphi[x := \psi]$.

A formula φ is defined over a set of variables, denoted $Var(\varphi)$. This includes, but is not limited to, the set of all variables that occur in the formula. An *interpretation* $\mathcal{I} \subseteq Var(\varphi)$ is a *model* of φ if it satisfies φ according to the classical truth table, where $\mathcal{I}$ satisfies an atom v if $v \in \mathcal{I}$. The set of all models of φ is denoted by $Mod(\varphi)$.

Let φ and ψ be propositional formulas. Then ψ is a logical consequence of φ, denoted $\varphi \models \psi$, if every model of φ is a model of ψ.

CHAPTER 3

Evaluation of Idempotent AASC

We start off this chapter with an evaluation approach for general AASC which amounts to reducing AASC to algebraic model counting. Then we describe how the restriction to idempotent semirings weakens some requirements for the evaluation algorithm. This offers attack points, where the presented general approach can be modified for idempotent AASC.

3.1 Reducing AASC to Algebraic Model Counting

There are multiple possible ways of evaluating AASC. The approach that we chose is used for inference on Problog programs (Kimmig et al. 2017). Its use for evaluating AASC has been described by Eiter et al. (2021). The main idea is to reduce AASC to a task called Algebraic Model Counting (AMC), for which available solvers can be used. Fierens et al. (2015) provided a detailed step-by-step description of the workflow. It is written in the context of Problog but the general idea remains the same in our setting.

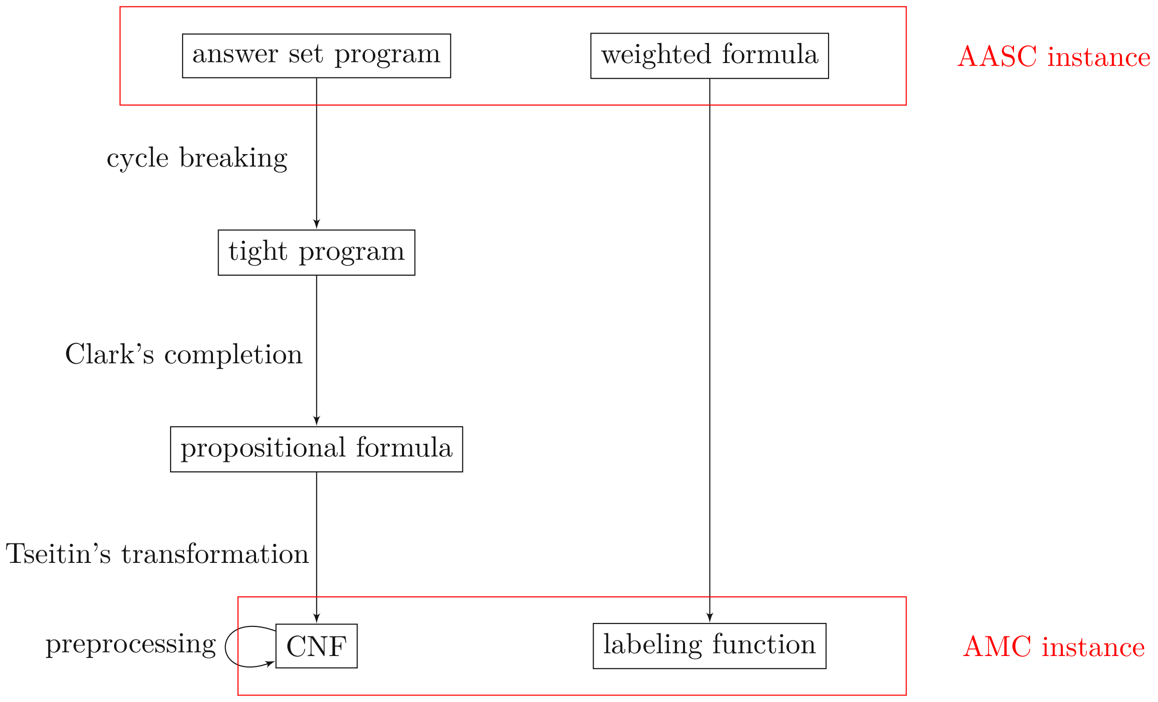
3.1.1 Algebraic Model Counting

Kimmig et al. (2017) introduced Algebraic Model Counting (AMC) as a generalization of Weighted Model Counting (WMC). In the context of this thesis, we can view AMC as a black box. We are thus primarily interested in how the input to AMC looks like.

Definition 11 (AMC Problem)**.** An *AMC instance* $\langle \Gamma, \beta, \mathcal{R} \rangle$ consists of

- a propositional logic theory Γ,
- a commutative semiring $\mathcal{R} = (R, \oplus, \otimes, e_{\oplus}, e_{\otimes})$, and
- a labeling function $\beta : \mathcal{L} \rightarrow R$, where $\mathcal{L}$ is the set of literals of the variables in Γ.

Figure 3.1: AASC evaluation by reduction to AMC



Algebraic model counting describes the task of computing the sum of the values of all models, where the value of a model is defined as the product of the values of its literals, i.e.

$$\bigoplus_{\mathcal{I} \in Mod(\Gamma)} \bigotimes_{a \in \mathcal{I}} \beta(a) \otimes \bigotimes_{a \notin \mathcal{I}} \beta(\neg a)$$

where $Mod(\Gamma)$ denotes the set of models of Γ.

3.1.2 Obtaining an AMC Instance

To obtain an AMC instance from an AASC instance, there are two main objectives, namely constructing a propositional logic theory and a labeling function. Moreover, we want the propositional theory to be a formula in CNF, so we can apply certain preprocessing techniques. Figure 3.1 visualizes the necessary steps in this approach. In the following we take a closer look at these steps.

As described in Section 2.3, we can assume that we are only dealing with factorized measures. Therefore, by definition we have a weight function β that can be used as the labeling function in the AMC instance.

The translation of an answer set program to propositional logic involves a bit more work. However, the task is not specific to AASC, as it does not involve the weighted formula and the semiring. This allows one to consider existing translations for plain ASP, as long as the necessary model-preserving properties are fulfilled. Namely, as opposed to some tasks like e.g. ASP consistency, in AASC in general every individual answer set is relevant to the result. This requires a one-to-one correspondence between the answer sets of the original program and the models of the CNF of the AMC instance. We capture this with a property called bijective faithfulness.

Definition 12 (Bijective Faithfulness)**.** Let Γ and Δ be propositional theories. A translation from Γ to Δ is *bijectively faithful* w.r.t. a set of variables $\mathcal{F}$, if there exists a bijection $\mu : \mathit{Mod}(\Gamma) \to \mathit{Mod}(\Delta)$, s.t. for every $\mathcal{I} \in \mathit{Mod}(\Gamma)$, $\mathcal{I} \cap \mathcal{F} = \mu(\mathcal{I}) \cap \mathcal{F}$ holds.

The property is defined for propositional theories, of which both answer set programs and propositional formulas are special cases. To ensure that the result of any query is preserved, we require for each step in the translation bijective faithfulness w.r.t. all contributing variables.

Theorem 1. If a translation from an AASC instance $\langle \Pi, \beta, \mathcal{R} \rangle$ to an AMC instance $\langle \{\varphi\}, \beta, \mathcal{R} \rangle$ is bijectively faithful w.r.t. all contributing variables, then the AASC result is preserved.

Proof. By definition of bijective faithfulness we know that each answer set of Π has exactly one corresponding model of φ that has the same truth values for all contributing variables. Therefore, these models have the same weights as their corresponding original answer sets. Since the result of an AASC instance with a factorized measure is computed in the same way as the result of an AMC instance, we conclude that the result is preserved. $\square$

In general the translation involves the following steps (as depicted in Figure 3.1):

1. *Cycle breaking.* The purpose of cycle breaking is to obtain a program without cycles in its positive dependency graph. The *positive dependency graph* G of a program Π is a directed graph, where $V(G) = \mathit{Var}(\Pi)$ and $(b, a) \in E(G)$ if for some $r \in \Pi$, $a \in H(r)$ and $b \in B^+(r)$. A program without such cyclic dependencies is also called a *tight program* (Lifschitz 1996).

   Cycle breaking is a well-studied topic in ASP. There are cycle breaking algorithms designed for plain ASP that are bijectively faithful, like e.g. by Janhunen (2003). There are also others, like by Eiter et al. (2021), that are designed with AASC in mind and are thus also bijectively faithful.

2. *Clark's completion* (Fages 1994) translates an tight program Π into a propositional theory Ψ, s.t. Π and Ψ have the same models, i.e. $\mathit{Mod}(\Pi) = \mathit{Mod}(\Psi)$.

3. *Tseitin's transformation* (Tseitin 1983), is used to obtain a CNF. The fact that Tseitin's transformation preserves models one-to-one is commonly used in the literature. However, as we could not find a full proof of this fact, we provide one here.

Definition 13 (Tseitin's transformation)**.** (Tseitin 1983) Let φ be a propositional formula over $\mathit{Var}(\varphi)$. The result of *Tseitin's transformation* is computed as follows.

a) Let ψ be a subformula of φ that has no further subformulas other than itself and simple variables. Let φ' be the formula obtained by replacing every occurrence of ψ in φ with a fresh variable l_ψ.
   Set $\varphi := \varphi' \land (l_\psi \leftrightarrow \psi)$, and $\mathit{Var}(\varphi) := \mathit{Var}(\varphi) \cup \{l_\psi\}$.

b) Repeat a) on subformulas of φ' until there are no more subformulas left to replace.

Since each ψ contains at most one logical operator, each $(l_\psi \leftrightarrow \psi)$ can easily be transformed into a CNF.

Theorem 2. Tseitin's transformation applied to a propositional formula φ is bijectively faithful w.r.t. $\mathit{Var}(\varphi)$.

Proof. We show that each step, transforming $\varphi = \varphi'[l_\psi := \psi]$ over $\mathit{Var}(\varphi)$ into $\varphi' \land (l_\psi \leftrightarrow \psi)$ over $\mathit{Var}(\varphi) \cup \{l_\psi\}$, is bijectively faithful w.r.t. $\mathit{Var}(\varphi)$.

Consider the mapping $\mu : \mathit{Mod}(\varphi) \to \mathit{Mod}(\varphi' \land (l_\psi \leftrightarrow \psi))$, where

$$\mu(\mathcal{I}) = \begin{cases} \mathcal{I} \cup \{l_\psi\} & \text{if } \mathcal{I} \models \psi \\ \mathcal{I} & \text{if } \mathcal{I} \not\models \psi. \end{cases}$$

If $\mathcal{I} \models \varphi$, then $\mu(\mathcal{I}) \models \varphi' \land (l_\psi \leftrightarrow \psi)$. Therefore, μ is a valid mapping. Also, μ is injective, i.e. maps distinct elements in the domain to distinct elements in the codomain.

For bijectivity, it remains to be shown that μ is surjective. For this, assume $\mathcal{J}$ is a model of $\varphi' \land (l_\psi \leftrightarrow \psi)$. This can only be the case if $\mathcal{J} \models l_\psi \iff \mathcal{J} \models \psi$. Therefore, it follows that $\mathcal{J} \setminus \{l_\psi\} \models \varphi$ and $\mu(\mathcal{J} \setminus \{l_\psi\}) = \mathcal{J}$.

□

These are the basic steps needed for the reduction to AMC. Additionally one can do some preprocessing on the resulting CNF in order to simplify the AMC instance. According to Lagniez et al. (2020), preprocessing is a valuable component in various automated reasoning tasks, like e.g. SAT solving (Eén and Biere 2005; Biere et al. 2021). To our knowledge, preprocessing techniques for algebraic model counting have not been extensively studied. It is, however, likely that some existing techniques for other reasoning tasks can be re-used for AMC. In this thesis we show in Chapter 5 that the preprocessing technique described by Lagniez et al. (2020) for model counting is bijectively faithful and thus applicable in evaluating general AASC.

3.2 Exploiting Idempotence

In the case of idempotent AASC, bijective faithfulness is a stronger requirement than we need. Therefore, in this section we describe a weaker (but sufficient) requirement. When summing up the weights of the answer sets, the idempotent addition operator is used. This means that multiplicity is ignored, i.e. multiple answer sets that have the same weight are only counted once.

Example 10. Assume we have two answer set programs, Π_1 and Π_2, that have the following weights, using the semiring $\mathcal{R}_{max}$:

- Π_1: $1, 1, 1, 1, 3, 3, 4$
- Π_2: $1, 3, 4, 4$

"Summing up" (using the *max*-operator) we get 4 as a result for both programs.

In the translation to an AMC instance we thus only need to make sure that each distinct model weight is preserved. For this we define faithfulness.

Definition 14 (Faithfulness)**.** Let Γ and Δ be propositional theories. A translation from Γ to Δ is *faithful* w.r.t. a set of variables $\mathcal{F}$, if for every model $\mathcal{I}$ of Γ there exists some model $\mathcal{J}$ of Δ s.t. $\mathcal{I} \cap \mathcal{F} = \mathcal{J} \cap \mathcal{F}$, and vice versa.

Similarly to general AASC, to ensure that query results are preserved in the translation from an AASC instance to an AMC instance, we require faithfulness for each step in the translation, w.r.t. all contributing variables.

Theorem 3. If a translation from an AASC instance $\langle \Pi, \beta, \mathcal{R} \rangle$, where $\mathcal{R}$ is idempotent, to an AMC instance $\langle \{\varphi\}, \beta, \mathcal{R} \rangle$ is faithful w.r.t. all contributing variables, then the AASC result is preserved.

Proof. By definition of faithfulness, we know that for each answer set of Π there is at least one corresponding model of φ that has the same truth values for all contributing variables, and vice versa. Therefore, each weight of an answer set also occurs as the weight of at least one model, and no models with new weights are introduced. Since idempotence lets us disregard multiplicities of weights when summing up, we conclude that the result of AASC is preserved. $\square$

This means that in the general approach presented in 3.1.2 one can replace the presented bijectively faithful algorithms for the individual steps by others that are only faithful.

CHAPTER 4

Non-bijective Cycle Breaking

In this chapter we show the applicability of two non-bijective cycle breaking algorithms (Hecher 2020; Lin and Reiter 1994) for idempotent AASC. As elaborated in the previous chapter, this requires a proof that they are faithful.

4.1 Algorithm 1

The first algorithm is by Hecher (2020) and is designed for deciding consistency of an answer set program, i.e. whether there exists an answer set. It is originally defined for head-cycle-free programs (Ben-Eliyahu and Dechter 1994), which generalize normal answer set programs. In head-cycle-free programs, the head of a rule can be a disjunction of variables, as long as in the positive dependency graph there is no cycle containing two variables from the same rule head. As the output of this cycle breaking algorithm already is a propositional formula, Clark's completion in a separate step is not needed here. In the following we refer to this algorithm as *Algorithm 1*.

```
Algorithm 4.1: Algorithm 1
    Input: a head-cycle-free program $\Pi$ and a tree decomposition $\mathcal{T} = (T, \chi)$ of $G_\Pi$
    Output: a propositional formula $\varphi$
```

The correctness of the translation (w.r.t. ASP consistency) is proven by the following theorem.

Theorem 4 (Correctness)**.** (Hecher 2020) Algorithm 1 is correct in that for each answer set of Π there is a model of φ and vice versa.

The proof of this theorem involves, in the one direction, extending an arbitrary model of Π to a model of φ. In the other direction, it is shown that the intersection of $\mathit{Var}(\Pi)$ and an arbitrary model of φ is a model of Π. This allows us to also conclude faithfulness.

Corollary 1. Algorithm 1 is faithful, w.r.t. the variables of the input program.

4.2 Algorithm 2

The second algorithm, which we will refer to as *Algorithm 2*, is by Lin and Zhao (2003).

For this algorithm we need the concept of a strongly connected component: a set S of variables in a program Π is called a *Strongly Connected Component (SCC)* of Π if there is a path in the dependency graph G_Π of Π from every node a to every node b, with $a, b \in S$, and S is $\subseteq$-maximal.

Definition 15 (Algorithm 2)**.** Let Π be an answer set program. Let the output program Ψ be empty initially.

1. Let $r \in \Pi$ be of the form

   $$a \leftarrow b_1, ..., b_n, d_1, ..., d_m, \text{not } c_1, ..., \text{not } c_s$$

   where $b_1, ..., b_n$ and a are in the same SCC while $d_1, ..., d_m$ are all not in a SCC with a. We add the following rules into Ψ:

   $$\begin{aligned}
   &a_r \leftarrow \text{not right}(a, b_1), ..., \text{not right}(a, b_n), \quad b_1, ..., b_n, d_1, ..., d_m, \text{not } c_1, ..., \text{not } c_s\\
   &a \leftarrow a_r\\
   &\text{right}(b_i, a) \leftarrow a_r\\
   &\text{right}(X, a) \leftarrow a_r, \text{right}(X, b_i)
   \end{aligned}$$

   where $1 \leq i \leq n$ and X is instantiated with all variables in the same SCC as a.

2. For every $r \in \Pi$ that is a constraint, add r to Ψ

The intuition behind $\text{right}(X, Y)$ is that it records X being used to derive Y. This predicate is used to block the application of a rule whose head variable has been used to derive a variable in the rule's positive body.

While an output program of Algorithm 2 is not necessarily tight, Lin and Zhao (2003) show that for each output program Π it still holds that Π and its completion have the same models.

It is also claimed that this translation is bijectively faithful w.r.t. the variables of the input program, which however is not true. The following is a counterexample.

Example 11. Let Π be defined as follows:

$$\begin{aligned}
&a\\
&b\\
&a \leftarrow b\\
&b \leftarrow a.
\end{aligned}$$

Applying Algorithm 2 results in the following program Π':

$$\begin{aligned}
&a_1\\
&a \leftarrow a_1\\
&b_2\\
&b \leftarrow b_2\\
&a_3 \leftarrow \text{not right}(a,b), b\\
&a \leftarrow a_3\\
&\text{right}(b,a) \leftarrow a_3\\
&\text{right}(a,a) \leftarrow a_3, \text{right}(a,b)\\
&\text{right}(b,a) \leftarrow a_3, \text{right}(b,b)\\
&b_4 \leftarrow \text{not right}(b,a), a\\
&b \leftarrow b_4\\
&\text{right}(a,b) \leftarrow b_4\\
&\text{right}(b,b) \leftarrow b_4, \text{right}(b,a)\\
&\text{right}(a,b) \leftarrow b_4, \text{right}(a,a).
\end{aligned}$$

Π has exactly one answer set, namely $\{a, b\}$, while Π' has two answer sets, namely $\{a, a_1, b, b_2, a_3, \text{right}(b,a)\}$ and $\{a, a_1, b, b_2, b_4, \text{right}(a,b)\}$.

4.2.1 Applicability in Idempotent AASC

While the translation is not bijectively faithful, we can still show faithfulness.

Theorem 5. Application of Algorithm 2 to an answer set program Π is faithful, w.r.t. $\mathit{Var}(\Pi)$.

Proof. Let Ψ be the result of applying Algorithm 2 to Π. We need to show that for every answer set $\mathcal{I}$ of Π there exists an answer set $\mathcal{I}_{ext}$ of Ψ, s.t. $\mathcal{I}_{ext} \cap \mathit{Var}(\Pi) = \mathcal{I}$, and vice versa.

" $\Longrightarrow$ "-direction: Assume $\mathcal{I}$ is an answer set of Π. We extend $\mathcal{I}$ to $\mathcal{I}_{ext}$ as follows.

1. We start with $\mathcal{I}_{ext} := \mathcal{I}$.

2. Consider some sequence $s \in (\Pi^{\mathcal{I}})^*$ of rules that computes $LM(\Pi^{\mathcal{I}})$, i.e. each $a \in LM(\Pi^{\mathcal{I}})$ is in the head of some rule in s and the body of each rule in s is satisfied by the head variables of the previous rules in the sequence. We assume our sequence contains all rules that are applicable (i.e. body is satisfied), except for any rule whose head variable has already been used to derive some variable in its body.

3. For each rule
$$a \leftarrow b_1, ..., b_n, d_1, ..., d_m$$
in this sequence, for every $1 \leq i \leq n$ and for every x with right$(x, b_i) \in \mathcal{I}_{ext}$, add a_r, right(b_i, a) and right(x, a) to $\mathcal{I}_{ext}$.

By construction of the sequence s, we know that for every rule
$$a \leftarrow b_1, ..., b_n, d_1, ..., d_m$$
in s, there is no right(a, b_i) in $\mathcal{I}_{ext}$. Therefore, for each such rule, the rules
$$a_r \leftarrow b_1, ..., b_n, d_1, ..., d_m$$
$$a \leftarrow a_r$$

are in $\Psi^{\mathcal{I}_{ext}}$. Therefore, $LM(\Psi^{\mathcal{I}_{ext}})$ will contain all $a \in LM(\Pi^{\mathcal{I}})$, and, by definition of Algorithm 2, all variables that we added to $\mathcal{I}$ in order to construct $\mathcal{I}_{ext}$. Therefore, $\mathcal{I}_{ext} \subseteq LM(\Psi^{\mathcal{I}_{ext}})$.

Now for the sake of contradiction assume $LM(\Psi^{\mathcal{I}_{ext}})$ contains additional variables that are not in $\mathcal{I}_{ext}$. W.l.o.g. we can assume this is a variable of the form a_r, derived by the rule
$$a_r \leftarrow \text{not right}(a, b_1), ..., \text{not right}(a, b_n), b_1, ..., b_n, d_1, ..., d_m, \text{not } c_1, ..., \text{not } c_s$$
in Ψ, with right$(a, b_i) \notin \mathcal{I}_{ext}$ for $1 \leq i \leq n$. But since $\mathcal{I}_{ext} \cap Var(\Pi) = \mathcal{I}$, the corresponding rule
$$a \leftarrow b_1, ..., b_n, d_1, ..., d_m$$
would be in $\Pi^{\mathcal{I}}$, and it would be applicable. Because we also know that a has not been used to derive any b_i, the rule has to be in s. Therefore, a_r also has to be in $\mathcal{I}_{ext}$.

" $\Longleftarrow$ "-direction: Assume $\mathcal{I}_{ext}$ is an answer set of Ψ. Let $\mathcal{I} := \mathcal{I}_{ext} \cap Var(\Pi)$.

Every $a \in \mathcal{I}$ is in $LM(\Psi^{\mathcal{I}_{ext}})$. Since for every two rules
$$a_r \leftarrow b_1, ..., b_n, d_1, ..., d_m$$
$$a \leftarrow a_r$$

in $\Psi^{\mathcal{I}_{ext}}$, the rule
$$a \leftarrow b_1, ..., b_n, d_1, ..., d_m$$
must be in $\Pi^{\mathcal{I}}$, every $a \in \mathcal{I}$ is also in $LM(\Pi^{\mathcal{I}})$.

Now for the sake of contradiction assume $LM(\Pi^{\mathcal{I}})$ contains an additional variable a that is not in $\mathcal{I}$. The rule used to derive a also has to be in $\Psi^{\mathcal{I}_{ext}}$, except if it got removed due to some right$(a, b) \in \mathcal{I}_{ext}$. However, this would mean that a is used to derive b. Therefore, a would be in $\mathcal{I}_{ext}$ and thus also in $\mathcal{I}$.

$\square$

CHAPTER 5

AMC Preprocessing

In this chapter we describe a preprocessing technique that simplifies the input CNF of AMC instances. Concretely, we modify an existing technique by Lagniez et al. (2020). The latter is designed for propositional model counting, where it shows promising empirical results. The general idea is to eliminate variables that are definable in terms of other variables. We transfer the technique to our setting and show that it is applicable in AASC (excluding contributing variables from elimination), even in the general case. Furthermore, we show that in the evaluation of idempotent AASC one can safely eliminate further non-contributing variables, using the same elimination technique.

5.1 Preliminaries

For this chapter we need some basic definitions concerning variable forgetting (Lang et al. 2003; Delgrande 2017) and definability in propositional logic (Lang and Marquis 2008). Notation-wise we mostly follow Lagniez et al. (2020).

5.1.1 Variable Forgetting

We want to be able to eliminate certain variables from a CNF formula, while preserving logical consequences that do not involve these variables. This is captured by the concept of variable forgetting.

Definition 16. A formula φ is *independent* of a set X of variables if φ is equivalent to a formula ψ s.t. $\mathit{Var}(\psi) \cap X = \emptyset$.

Definition 17 (Variable Forgetting)**.** Let $\varphi \in \mathcal{L}_{\mathcal{P}}$ and $X \subseteq \mathit{Var}(\varphi)$. The *forgetting* of X in φ, denoted as $\exists X.\varphi$, is defined over $\mathit{Var}(\varphi) \setminus X$ and is the strongest logical consequence of φ, that is independent of X. "Logically strongest" means that for every formula ψ independent of X it holds that $\varphi \models \psi$ implies $\exists X.\varphi \models \psi$.

If φ is a CNF, $\exists X.\varphi$ can be computed by recursively eliminating each $x \in X$ by applying the resolution principle, i.e. resolving every clause containing x with every clause containing $\neg x$.

Example 12. Let φ be the CNF consisting of the following clauses:

$$\begin{array}{rr} (a \lor b) & (b \lor c) \\ (\neg a \lor e \lor f) & (\neg b \lor \neg e) \end{array}$$

Then $\exists\{a\}.\varphi$:

$$\begin{array}{rr} (b \lor e \lor f) & (b \lor c) \\ (\neg b \lor \neg e) & \end{array}$$

and $\exists\{b, a\}.\varphi = \exists\{b\}.(\exists\{a\}.\varphi)$:

$$(c \lor \neg e)$$

We cannot always forget variables without changing the AASC result. This is where we need the concept of definability, which we can later use to define a necessary prerequisite.

5.1.2 Definability

We introduce a formal definition of definability, i.e. the ability to derive the value of a variable in a formula from the values of some of the other occurring variables.

Definition 18 (Definability)**.** Let $\varphi \in \mathcal{L}_\mathcal{P}$, $X \subseteq \mathit{Var}(\varphi)$, and $y \in \mathit{Var}(\varphi)$. Then φ *defines* y in terms of X if there exists a formula ψ_X over X s.t. $\varphi \models (\psi_X \leftrightarrow y)$.

For the preprocessing, we are interested in finding a set of variables defined in terms of the remaining variables in the formula.

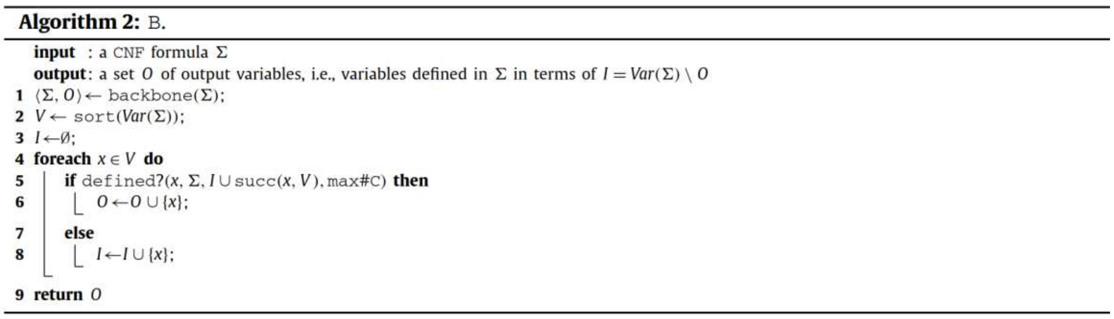
Definition 19 (Definability Partition)**.** Let $\varphi \in \mathcal{L}_\mathcal{P}$. A *definability partition* is a pair $\langle I, O\rangle$, s.t. $I \cup O = \mathit{Var}(\varphi)$, $I \cap O = \emptyset$, and φ defines every $x \in O$ in terms of I. A definability partition is *subset-minimal* if there exists no definability partition $\langle I', O'\rangle$ with $I' \subseteq I$.

If φ is given in the context, we refer to O as a *set of defined variables*, actually meaning that $\langle \mathit{Var}(\varphi) \setminus O, O\rangle$ is a definability partition.

5.2 Forgetting of Defined Variables

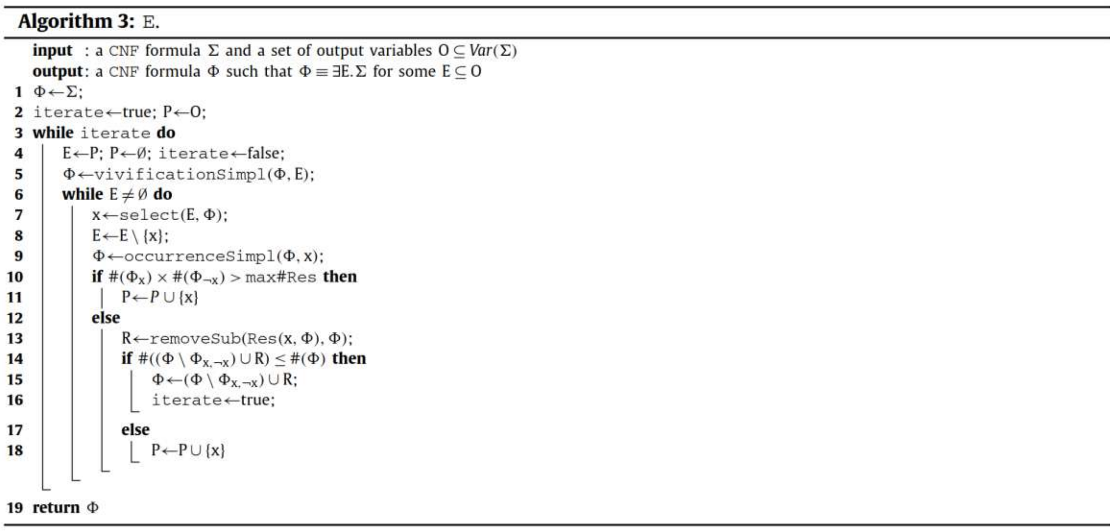
In this section we first present the existing preprocessing algorithm by Lagniez et al. (2020) which eliminates defined variables . In the second part we show that this algorithm is applicable in general AASC.

Figure 5.1: (Lagniez et al. 2020) Algorithm B



```
Algorithm 2: B.
input  : a CNF formula $\Sigma$
output: a set $O$ of output variables, i.e., variables defined in $\Sigma$ in terms of $I = Var(\Sigma) \setminus O$
$\langle \Sigma, O \rangle \leftarrow$ backbone($\Sigma$);
$V \leftarrow$ sort($Var(\Sigma)$);
$I \leftarrow \emptyset$;
foreach $x \in V$ do
    if defined?($x, \Sigma, I \cup$ succ($x, V$), max#C) then
        $O \leftarrow O \cup \{x\}$;
    else
        $I \leftarrow I \cup \{x\}$;
return $O$
```

Figure 5.2: (Lagniez et al. 2020) Algorithm E



```
Algorithm 3: E.
input  : a CNF formula $\Sigma$ and a set of output variables $O \subseteq Var(\Sigma)$
output: a CNF formula $\Phi$ such that $\Phi \equiv \exists E.\Sigma$ for some $E \subseteq O$
$\Phi \leftarrow \Sigma$;
iterate $\leftarrow$ true; $P \leftarrow O$;
while iterate do
    $E \leftarrow P$; $P \leftarrow \emptyset$; iterate $\leftarrow$ false;
    $\Phi \leftarrow$ vivificationSimpl($\Phi, E$);
    while $E \neq \emptyset$ do
        $x \leftarrow$ select($E, \Phi$);
        $E \leftarrow E \setminus \{x\}$;
        $\Phi \leftarrow$ occurrenceSimpl($\Phi, x$);
        if $\#(\Phi_x) \times \#(\Phi_{\neg x}) >$ max#Res then
            $P \leftarrow P \cup \{x\}$
        else
            $R \leftarrow$ removeSub(Res($x, \Phi$), $\Phi$);
            if $\#((\Phi \setminus \Phi_{x,\neg x}) \cup R) \leq \#(\Phi)$ then
                $\Phi \leftarrow (\Phi \setminus \Phi_{x,\neg x}) \cup R$;
                iterate $\leftarrow$ true;
            else
                $P \leftarrow P \cup \{x\}$
return $\Phi$
```

5.2.1 Algorithm **B+E**

Algorithm B+E consists of two parts. In the first part (Figure 5.1) a definability partition $\langle I, O \rangle$ to an input CNF formula φ is computed.

The second part (Figure 5.2) takes ϕ and the set of defined variables O as input. It then computes a formula that is equal to the forgetting of E in φ for some $E \subseteq O$.

Both algorithms are proven to be correct w.r.t. their input/output specifications (Lagniez et al. 2020). This allows us to view them as a black box.

Note that not all variables in O are forgotten. The reason for this is that for some variables, elimination would increase the number of clauses in the formula. After computing the set of resolvents, the clauses properly subsumed by other clauses are removed. Then there is a check for whether the resulting set of clauses of the formula is larger than before the elimination. For variables where this would be the case, their elimination is postponed. This ensures that as a result we get a CNF formula that has fewer (or not more) variables than the input formula, while at the same time the size is not increased (or only by a negligible factor) (Lagniez et al. 2020).

5.2.2 Applicability in General AASC

In propositional model counting, for which `B+E` is designed, the requirement to the preprocessor is that the number of models stays the same. This property is not sufficient in our case. Instead we need bijective faithfulness.

We start by showing that variable forgetting does not introduce new models.

Lemma 1. Let $\varphi \in \mathcal{L}_\mathcal{P}$ and $X \subseteq \mathit{Var}(\varphi)$. Then every model of $\exists X.\varphi$ can be extended to a model of φ.

Proof. Let $\mathcal{M} \in \mathit{Mod}(\exists X.\varphi)$. Let $\gamma := \bigwedge_{x \in \mathit{Var}(\exists E.\varphi) \cap \mathcal{M}} x \wedge \bigwedge_{x \in \mathit{Var}(\exists E.\varphi) \setminus \mathcal{M}} \neg x$. Clearly $\gamma \wedge \exists E.\varphi$ is consistent, thus

$$\exists X.\varphi \not\models \neg\gamma.$$

As $\exists X.\varphi$ is the strongest logical consequence of φ, and γ is independent of X, we have that

$$\varphi \not\models \neg\gamma.$$

This then implies that $\gamma \wedge \varphi$ is consistent. Thus, φ has a model that extends $\mathcal{M}$. □

Now we can show bijective faithfulness, given that the input is a set of defined variables.

Theorem 6. Let $\varphi \in \mathcal{L}_\mathcal{P}$ and let $\langle I, O\rangle$ be a definability partition of φ. Let $E \subseteq O$. Then a translation from φ to $\exists E.\varphi$ is bijectively faithful w.r.t. $\mathit{Var}(\exists E.\varphi)$.

Proof. We show the theorem by establishing that the mapping

$$\begin{aligned} \mu :& \mathit{Mod}(\varphi) \to \mathit{Mod}(\exists E.\varphi) \\ & \mathcal{N} \mapsto \mathcal{N} \cap \mathit{Var}(\exists E.\varphi) \end{aligned}$$

is a bijection.

First, μ is a valid mapping since by definition every model of φ is a model of $\exists E.\varphi$. For bijectivity we need to show injectivity and surjectivity:

- injectivity:
  Let $\mathcal{M} \in \mathit{Mod}(\exists E.\varphi)$. We know that φ defines every $x \in E$ in terms of I. Furthermore, $I \subseteq \mathit{Var}(\varphi) \setminus E = \mathit{Var}(\exists E.\varphi)$ Thus, if there exists an extension of $\mathcal{M}$ to a model of φ it is unique. This implies that μ maps distinct models of φ to distinct models of $\exists E.\varphi$.

- surjectivity:
  This follows from Lemma 1.

□

This result additionally implies that the number of models does not change, which is a weaker property proven by Lagniez et al. (2020).

Corollary 2. Let $\varphi \in \mathcal{L}_\mathcal{P}$ and let $\langle I, O\rangle$ be a definability partition of φ. Let $E \subseteq O$. Then $|Mod(\varphi)| = |Mod(\exists E.\varphi)|$

When implementing this preprocessing technique, one has to consider that for the variables that are eliminated their values will not be calculated in AMC. Since all contributing variables are needed to determine the model weights, these variables have to be excluded from elimination. One can do this by, after computing the definability bipartition $\langle I, O\rangle$, simply moving all contributing variables that are in O into I. Or one can modify $\mathtt{B}$ to not put contributing variables into O when computing the definability partition.

5.3 Forgetting Non-contributing Variables

In this section we propose a modification of the described preprocessing that is applicable for idempotent AASC and eliminates any variable that is non-contributing. Theorem 7 proves that the forgetting of any set of variables is faithful.

Theorem 7. Let $\varphi \in \mathcal{L}_\mathcal{P}$ and let $E \subseteq Var(\varphi)$. Then a translation from φ to $\exists E.\varphi$ is faithful w.r.t. $Var(\exists E.\varphi)$.

Proof. By definition every model of φ is a model of $\exists E.\varphi$. The other direction follows from Lemma 1. □

Again, for applying this technique to idempotent AASC, we require E to be a subset of the non-contributing variables. This gives us faithfulness w.r.t. all contributing variables. Therefore, since this is our only restriction on the input variables, our proposed preprocessing technique consists of running Algorithm $\mathtt{E}$ on the set of all non-contributing variables.

CHAPTER 6

Implementation of AMC Preprocessing

This chapter describes a way of using the two preprocessing techniques described in Chapter 5 in practice. We have incorporated them into the algebraic answer set counter `aspmc` (Eiter et al. 2021), by modifying an implementation of `B+E` accordingly. Our code is publicly available, both the preprocessor module [1] and its invocation in `aspmc` [2].

Furthermore, we present the results of benchmark tests designed to assess the impact of this preprocessing on the time needed to solve instances.

6.1 Implementation

Our first approach was to use the exact implementation[3] of `B+E` described in Lagniez et al. (2020). There, we encountered two problems. (1) When running `E` alone, the variables that were actually forgotten were not listed in the output. However, this information is needed to get the correct result in counting, as forgotten variables need special treatment. Theoretically they are no longer part of the set of variables over which the formula is defined. In practice one can to the same effect assume a fixed value for each of them, e.g. set all of them to be true. This problem of missing output regarding forgotten variables required only minor changes to the code and was resolved by the authors upon request.

(2) The second problem is that some variables seem to disappear during the preprocessing. The following is an example where we could observe this behavior.

[1] `https://github.com/martin5598/sharpsat-td-preprocessor`
[2] `https://github.com/raki123/aspmc`
[3] `https://github.com/crillab/b-plus-e`

Example 13. For the CNF over $\{a, b, c, d\}$ consisting of the following clauses

$$\begin{array}{cc} (\neg a) & (\neg c) \\ (\neg a \vee d) & (a \vee \neg d) \\ (d \vee \neg b) & (\neg d \vee b) \end{array}$$

one would expect as a result of forgetting c and d, the CNF

$$(\neg a) \qquad (\neg b)$$

over $\{a, b\}$. However, the result of running `E` on c and d as input variables is an empty clause set over the variables $\{a, b, c, d\}$ (i.e. no variables are forgotten, but still all clauses disappear).

We suspect that this behavior is due to some additional simplification technique. However, due to this the required output specifications, as described for Algorithm `E`, are not met and thus faithfulness is not guaranteed. This can be seen in the above example, where the input formula has one model $\mathcal{I}$, but for the output formula every interpretation is a model.

To resolve this second problem we estimated more extensive changes to be necessary. Thus, we decided to consider a different implementation of `B+E`, namely the one used in the preprocessor of SharpSAT-TD[4], which we estimated to be easier to adapt to our setting. It differs in some details from `B+E` as it is described in Lagniez et al. (2020), but the elimination of variables is implemented in the same way. Therefore, we get as output the forgetting of some of the input variables, which is consistent with the specifications of `B+E`. We added a mode for idempotent semirings that implements the modified version described in Section 5.3. The other changes necessary to incorporate the preprocessor into `aspmc` were mostly of a technical nature.

6.2 Experiments

In this section we describe how the benchmark tests were conducted and then present the results thereof.

6.2.1 Experimental Questions

With our experiments we intend to answer the following questions:

Q1 Does preprocessing decrease the time needed for knowledge compilation / counting?

Q2 Does the overall performance (wall clock time) improve with preprocessing, i.e. does the speedup of compilation/counting make up for the additional overhead of preprocessing?

[4]`https://github.com/Laakeri/sharpsat-td`

Q3 Regarding the overall performance impact of preprocessing, are there different results depending on instance size?

For Q1, we expected preprocessing to reduce the time needed for both compilation and counting. For Q2, we expected that it would improve the overall performance. For Q3, we expected instance size not to be a factor in whether preprocessing increases or decreases the total time.

6.2.2 Setup

Benchmarking Instances We use five different benchmarking sets, all publicly available[5]. The set (1) *simple_paths* contains the graph representations of public transport networks from several transport agencies over the world. This set has been used by Eiter et al. (2021) for benchmarking. The set (2) *smokers* contains instances of the well-known smokers example (Fierens et al. 2015). In this example, for a group of people each one (with a certain probability) either smokes or does not, and each person who smokes may (with a certain probability) influence their friends to start smoking too. We generated the the instances ourselves, using randomly generated power law graphs that are known to resemble social networks (Barabási and Bonabeau 2003). The sets (3) *gh* and (4) *gnb* contain artificially generated programs with an increasing number of head variables and negated body variables respectively. The set (5) *blood* contains instances that model the inheritance of blood types, with an increasing number of ancestors. All three of *gh*, *gnb*, and *blood* are from Bellodi et al. (2020).

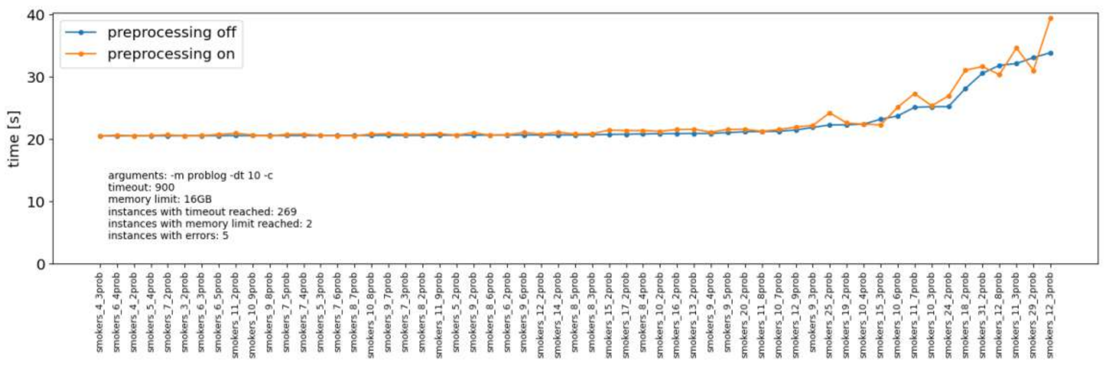
The *simple_paths* set consists of instances for answer set counting. The other sets consist of AASC instances over the probabilistic semiring. In addition *gh*, *gnb*, and *blood* are also suitable for Most Probable Explanation (MPE) inference (Pearl 1988), using the idempotent maxtimes semiring. Recall that MPE inference is finding the most likely interpretation of non-evidence variables, given some evidence (truth values of some variables).

Hardware Specifications We use a cluster consisting of 12 nodes. Each cluster has two Intel Xeon E5-2650 CPUs with 2.2 GHz clock speed and access to 256 GB shared RAM. For the benchmarking we limit each run of an instance to 8GB RAM and specify a timeout of 1800 seconds.

Run Configurations All instances were run with `aspmc` in Problog mode (i.e. using Problog syntax), using the arguments `-m problog -dt <t> -c`, except for the *simple_paths* instances which are run in ASP mode, using `-m asp -dt <t> -c`. The timeout `<t>` for computation of a tree decomposition was set to 10 seconds for the sets containing large instances, and to 1 second otherwise. The sets *gh*, *gnb*, and *blood* are

[5] `https://doi.org/10.5281/zenodo.6428320`

Figure 6.1: Wall clock times for the set *smokers*, part 1 of 2.



additionally run using the idempotent semiring $(\mathbb{R}_+, \max, \cdot, 0, 1)$, by adding the argument `-s maxtimes`.

Each instance is run with and without preprocessing, three repetitions each. To enable preprocessing, the additional argument `-p` is used. Some instances may fail due to reaching the specified timeout or memory limit. Furthermore, in some cases we get an error because no tree decomposition is found within the specified timeout for computation of a tree decomposition. For interpreting the results we discard instances that have less than two runs out of three completed for either mode (preprocessing disabled/enabled). The following values are obtained as results from the runs:

- *total wall clock time*,
- *knowledge compilation time*,
- *counting time*,
- *preprocessing time*.

The times for knowledge compilation, counting, and preprocessing are measured and printed out by `aspmc`. Thus, we were able to parse these values from the output.

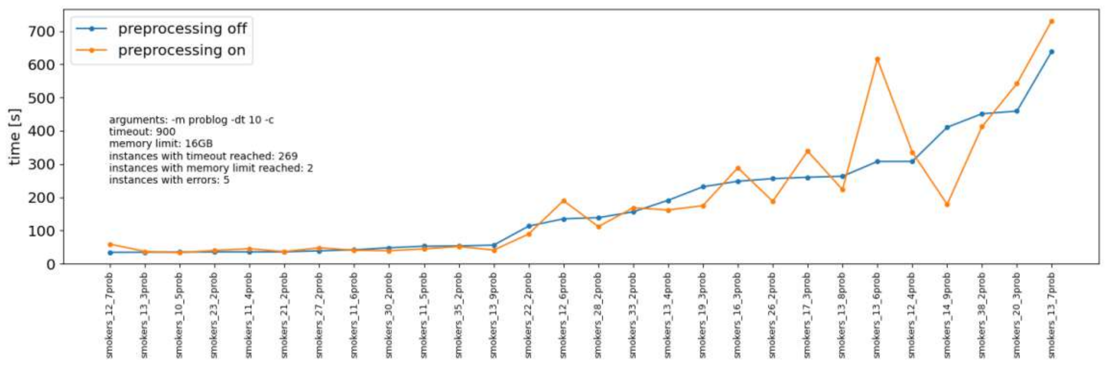
6.2.3 Experimental Results

For the sets *gh*, *gnb*, *blood* all runs were completed. For *smokers* and *simple_paths* some runs failed. Also, in the latter case, due to the large number of instances we ended the experiment after about 24 hours, before all runs could be started. A complete collection of all the results plus the scripts used to parse and plot the results are publicly available[6].

We first have a look at the results concerning Q2. An improvement in overall performance can not be observed in general. Our preprocessing more often seems to have a mixed (see

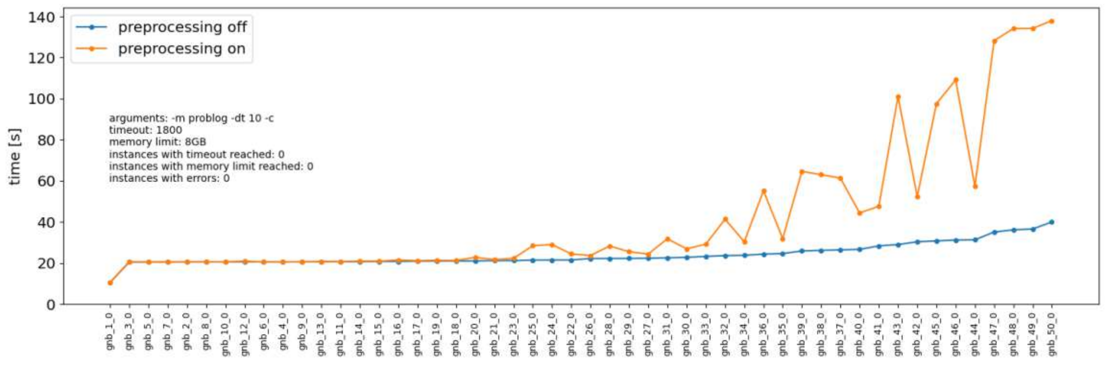
[6]`https://github.com/martin5598/benchmark-results`

Figure 6.2: Wall clock times for the set *smokers*, part 2 of 2.



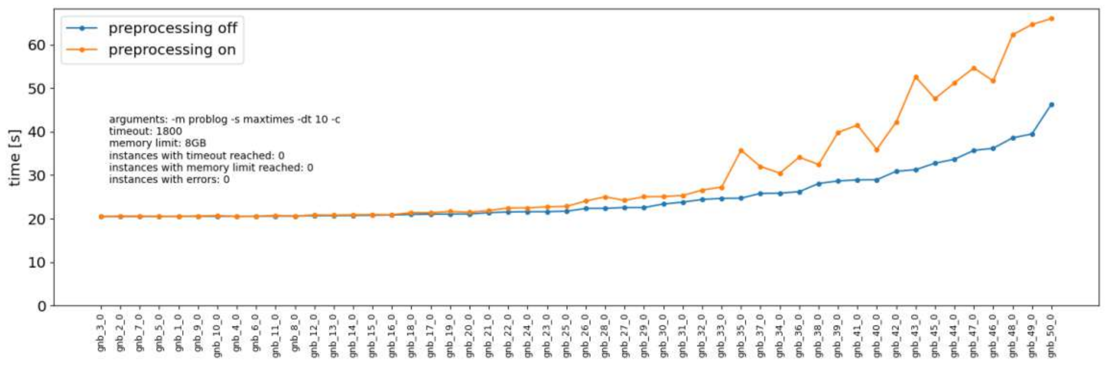
arguments: -m problog -dt 10 -c
timeout: 900
memory limit: 16GB
instances with timeout reached: 269
instances with memory limit reached: 2
instances with errors: 5

Figure 6.3: Wall clock times for the set *gnb*.



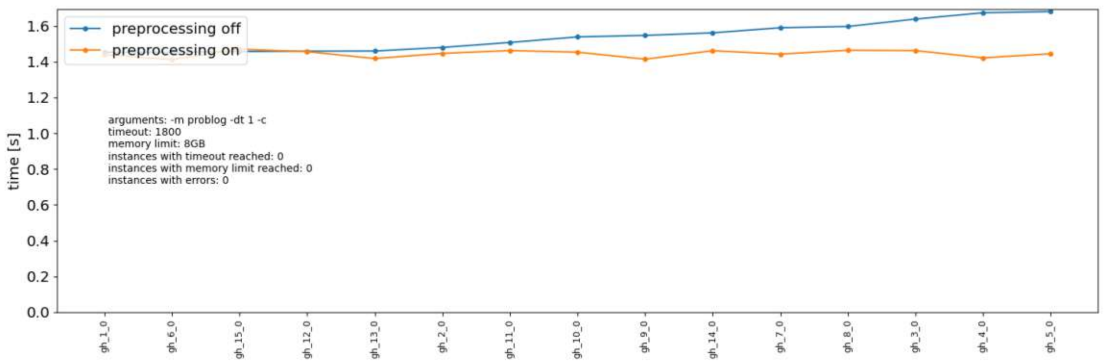
arguments: -m problog -dt 10 -c
timeout: 1800
memory limit: 8GB
instances with timeout reached: 0
instances with memory limit reached: 0
instances with errors: 0

Figure 6.4: Wall clock times for the set *gnb*, using an idempotent semiring.



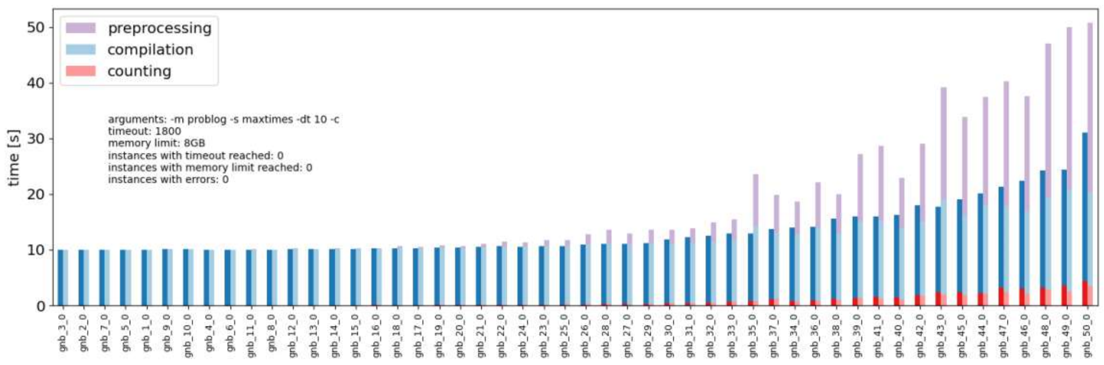
arguments: -m problog -s maxtimes -dt 10 -c
timeout: 1800
memory limit: 8GB
instances with timeout reached: 0
instances with memory limit reached: 0
instances with errors: 0

Figure 6.5: Wall clock times for the set *gh*.



```
arguments: -m problog -dt 1 -c
timeout: 1800
memory limit: 8GB
instances with timeout reached: 0
instances with memory limit reached: 0
instances with errors: 0
```

Figure 6.6: Detailed times for the set *gnb*, using an idempotent semiring.



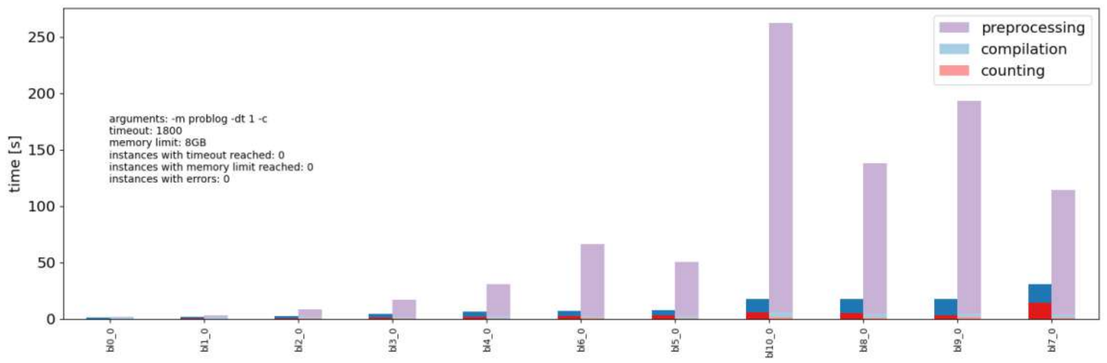
```
arguments: -m problog -s maxtimes -dt 10 -c
timeout: 1800
memory limit: 8GB
instances with timeout reached: 0
instances with memory limit reached: 0
instances with errors: 0
```

Figures 6.1 and 6.2) or even negative (see Figures 6.3 and 6.4) impact on performance. An example where the preprocessed runs are faster is depicted in Figure 6.5, but this is a rather small set with small instances.

To get an insight as to why the preprocessor is performing badly, we have a look at Q1. For this we plot the individual times for compilation, counting, and preprocessing as stacked bars. For each pair of columns, the first one represents times with preprocessing disabled, the second one with preprocessing enabled. Figures 6.6 and 6.7 show that preprocessing does decrease the time needed for compilation and counting in almost all of the depicted instances. Here the problem is that preprocessing itself takes too much time. In particular for the larger instances of these sets the preprocessing times seem to explode.

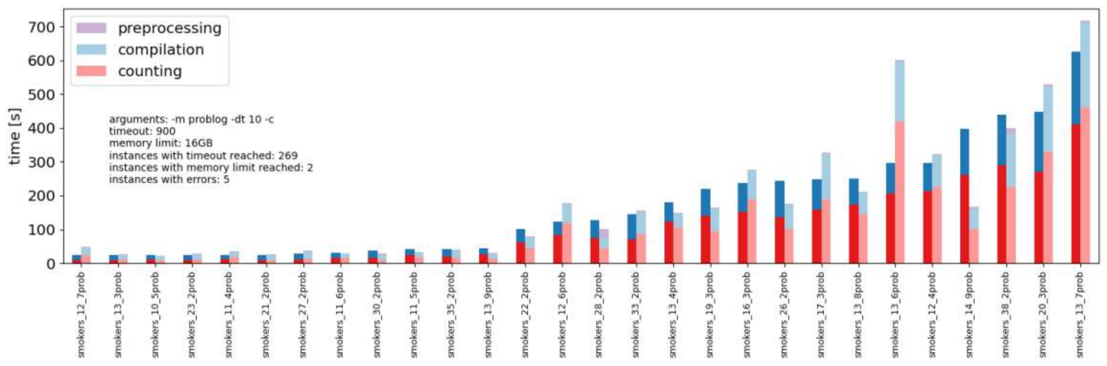
In the case of the *smokers* set (Figure 6.8) there seems to be another reason for some of the instances taking longer with preprocessing enabled. Here, preprocessing time accounts for only a small fraction of the total time. And while for some instances it is beneficial for performance, for others it increases compilation and counting time. We could not figure out an apparent reason for this behavior.

Figure 6.7: Detailed times for the set *blood*.



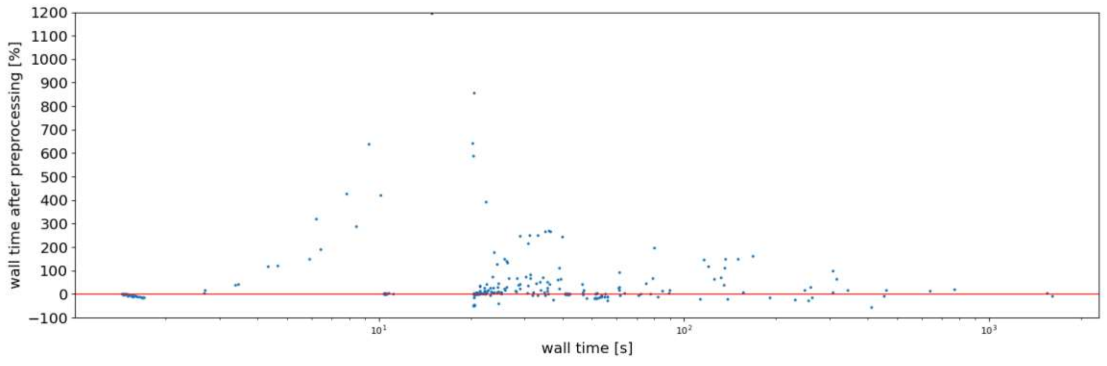
```
arguments: -m problog -dt 1 -c
timeout: 1800
memory limit: 8GB
instances with timeout reached: 0
instances with memory limit reached: 0
instances with errors: 0
```

Figure 6.8: Detailed times for the set *smokers*, part 2.



```
arguments: -m problog -dt 10 -c
timeout: 900
memory limit: 16GB
instances with timeout reached: 269
instances with memory limit reached: 2
instances with errors: 5
```

Figure 6.9: Relative change in wall time depending on instance size, using instances from all sets.



To answer Q3, we plot the instances of all sets together in a scatter plot (Figure 6.9). The X-axis represents the instance size, for which we use the wall time without preprocessing as a measure. The Y-axis represents the relative change in wall time caused by enabling preprocessing. Below the red line preprocessing decreases wall time, i.e. it improves performance. The plot shows no clear correlation with instance size. It holds for both small and large instances that our preprocessor implementation is sometimes beneficial but more often disadvantageous.

6.2.4 Summary

Considering these results we conclude that this implementation of the preprocessor `B+E` does not have the impact on performance that we hoped for, given that for a majority of instances the total wall time increases with preprocessing enabled. Therefore, at the moment we do not recommend using this feature in `aspmc`.

As for a possible cause of these long running times, we could not definitely make out one. The most likely cause is the generation of the resolvents and the removal of redundant clauses (that are subsumed by other clauses) afterward. This quickly becomes expensive for variables that appear in more than a few clauses. In the implementation that we used, there is a limit for the number of clauses a variable may appear in, in order to be considered for elimination. However, this is checked only once at the beginning, before any variable is eliminated.

A positive takeaway is that the preprocessor decreases (or does not increase) the time needed for compilation and counting (except for a few instances in the *smokers* set). Therefore if, hypothetically, the large preprocessing times are due to some efficiency issue with the preprocessor implementation that we use, one might by fixing that issue get a preprocessor that is actually beneficial. Another promising option might be to use a different implementation altogether.

CHAPTER 7

Conclusion

We can divide the contributions of this thesis into a theoretical and a practical part. In the theoretical part we established the applicability

- of two algorithms for cycle breaking in the evaluation of idempotent AASC,
- and of an algorithm for preprocessing in the evaluation of general AASC, with an extended version for idempotent AASC.

For the cycle breaking algorithm by Hecher (2020), applicability in our setting already follows from the original correctness proof. The second algorithm by Lin and Zhao (2003) is claimed to be bijectively faithful which would make it applicable for general AASC. However, we found this not to be true and provided a counterexample. Then we showed that despite not being bijectively faithful the algorithm can still be used for idempotent AASC.

Concerning the preprocessing step, we first established applicability of the `B+E` preprocessor (originally designed for model counting) in the evaluation of general AASC. We were able to prove applicability without modifications to the original algorithms. We also proposed a modified version that makes use of the restriction to idempotent semirings. Here, the restriction that only defined variables can be forgotten does not apply anymore and can thus be dropped. This allowed us to consider all non-contributing variables for forgetting. Contributing variables are excluded in both versions, so that the answers to AASC are preserved.

In the practical part of this thesis, we added a preprocessing feature to `aspmc`, based on our theoretical results. For this we adapted an existing implementation of `B+E`. Besides a few necessary technical adaptations, we also implemented the mode for idempotent AASC. This mode may turn out to be more efficient, since more variables can be eliminated and at the same time no computation of a definability bipartition is necessary.

To evaluate the impact on the time needed to solve instances, we performed benchmark tests on a computer cluster. The results indicate that while our preprocessor implementation reduces the time needed for the subsequent steps (knowledge compilation and counting), the additional overhead introduced by the preprocessing itself negates this effect. More often than not the time needed for solving increased with enabling preprocessing, which is the opposite of what we want. These observations apply to both versions (general and idempotent). As for possible causes for this inefficiency, we suspect that much of the preprocessing time can be attributed to the variable elimination part, more specifically the computation of resolvents and the removal of redundant clauses afterwards. However, to get to the bottom of this, a deeper investigation is necessary.

Future Work The next step would be to find out whether the inefficiency of our preprocessor is (at least partly) due to some inefficiency in the implementation that can be fixed. One option would be to examine the selection of variables that are actually eliminated. In the implementation that we used, this is dependent on the number of clauses a variable appears in. Furthermore, this is for all variables only checked once, before any variable is actually eliminated.

As an alternative to fixing the current implementation, it may be worthwhile considering a different implementation of `B+E` altogether, like the one[1] we initially attempted to adapt.

Besides the preprocessor that we considered, one can also explore other ways of AMC preprocessing. In general, to the best of our knowledge, there is not much previous work on preprocessing for AMC. Looking at further existing techniques for similar tasks such as model counting, weighted model counting, etc. might be promising.

In this thesis we presented cycle breaking algorithms that could be used as an alternative to currently used ones. It remains to be examined whether these alternatives have advantages in complexity and whether an implementation of them would outperform the currently used cycle breakings.

[1]`https://github.com/crillab/b-plus-e`

List of Figures

List of Tables

List of Algorithms

Bibliography

Albert-László Barabási and Eric Bonabeau. Scale-free networks. *Scientific American*, 288(5):60–69, 2003.

Elena Bellodi, Marco Alberti, Fabrizio Riguzzi, and Riccardo Zese. Map inference for probabilistic logic programming. *Theory and Practice of Logic Programming*, 20(5): 641–655, 2020. doi:10.48550/arXiv.2008.01394.

Rachel Ben-Eliyahu and Rina Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(1):53–87, 1994.

Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. *Handbook of Satisfiability*, pages 391–435, 2021.

Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, December 2011. ISSN 0001-0782. doi:10.1145/2043174.2043195.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, pages 2462–2467. Hyderabad, 2007.

James P. Delgrande. A knowledge level account of forgetting. *Journal of Artificial Intelligence Research*, 60:1165–1213, 2017.

Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 61–75. Springer, 2005.

Thomas Eiter and Rafael Kiesel. Weighted LARS for quantitative stream reasoning. In *ECAI 2020*, pages 729–736. IOS Press, 2020. doi:10.3233/FAIA200160.

Thomas Eiter and Rafael Kiesel. On the complexity of sum-of-products problems over semirings. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6304–6311, 2021.

Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. *Answer Set Programming: A Primer*, pages 40–110. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-03754-2. doi:10.1007/978-3-642-03754-2_2.

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*, volume 18, pages 269–279, 2021. doi:10.24963/kr.2021/26.

Herbert B. Enderton. *A Mathematical Introduction to Logic*. Elsevier, 2001.

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

Johannes K. Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Dynasp2. 5: Dynamic programming on tree decompositions in action. *Algorithms*, 14(3):81, 2021.

Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

Abram Friesen and Pedro Domingos. The sum-product theorem: A foundation for learning tractable models. In *International Conference on Machine Learning*, pages 1909–1918. PMLR, 2016.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo= ASP+ control: Preliminary report. *arXiv preprint arXiv:1405.3694*, 2014.

Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings Fifth Intl. Conference and Symposium Logic Programming*, pages 1070–1080. MIT Press, 1988.

Markus Hecher. Treewidth-aware reductions of normal ASP to SAT - is normal ASP harder than SAT after all? In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 17, pages 485–495, 2020. doi:10.24963/kr.2020/49.

Tomi Janhunen. A counter-based approach to translating normal logic programs into sets of clauses. In *Proceedings of the 2nd Intl. Answer Set Programming Workshop*, pages 166–180, 2003.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. An algebraic prolog for reasoning about possible worlds. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *Journal of Applied Logic*, 22:46–62, 2017. doi:10.1016/j.jal.2016.11.031.

Hans Kleine Büning and Theodor Lettmann. *Propositional Logic: Deduction and Algorithms*, volume 48. Cambridge University Press, 1999.

Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Definability for model counting. *Artificial Intelligence*, 281:103229, 2020. doi:10.1016/j.artint.2019.103229.

Jérôme Lang and Pierre Marquis. On propositional definability. *Artificial Intelligence*, 172(8-9):991–1017, 2008.

Jérôme Lang, Paolo Liberatore, and Pierre Marquis. Propositional independence-formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18: 391–443, 2003.

Javier Larrosa, Albert Oliveras, and Enric Rodríguez-Carbonell. Semiring-induced propositional logic: Definition and basic algorithms. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 332–347. Springer, 2010.

Vladimir Lifschitz. Foundations of logic programming. *Principles of Knowledge Representation*, 3:69–127, 1996.

Fangzhen Lin and Ray Reiter. Forget it. In *Working Notes of AAAI Fall Symposium on Relevance*, pages 154–159, 1994.

Fangzhen Lin and Jicheng Zhao. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 853–858, 2003.

Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

Kaile Su, Abdul Sattar, Guanfeng Lv, and Yan Zhang. Variable forgetting in reasoning about knowledge. *Journal of Artificial Intelligence Research*, 35:677–716, 2009.

Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, pages 466–483. Springer, 1983.

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*, 78:15–32, 2016.