TU WIEN Informatics

Reboots in Lazy-Grounding ASP-Solving

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Paul Behofsics, BSc.
Matrikelnummer 11777756

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter
Mitwirkung: Univ.Ass. Dipl.-Inf. Dr.techn. Antonius Weinzierl

Wien, 14. Juli 2023

Paul Behofsics	Thomas Eiter

TU WIEN Informatics

Reboots in Lazy-Grounding ASP-Solving

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Paul Behofsics, BSc.
Registration Number 11777756

to the Faculty of Informatics

at the TU Wien

Advisor: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter
Assistance: Univ.Ass. Dipl.-Inf. Dr.techn. Antonius Weinzierl

Vienna, 14th July, 2023

Paul Behofsics | Thomas Eiter

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

Erklärung zur Verfassung der Arbeit

Paul Behofsics, BSc.

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, 14. Juli 2023

Paul Behofsics

Acknowledgements

First, I would like to thank my advisor Thomas Eiter for providing feedback regarding direction and methodology. Also, I would like to thank my co-advisor Antonius Weinzierl. Our discussions helped me overcome many problems during the project and his expertise significantly shaped the ideas behind this thesis.

Furthermore, I would like to thank the Knowledge-Based Systems group at the TU Vienna, that provided the computational resources for the performed experiments and benchmarks.

Finally, I'd like to thank my parents, friends and family for supporting me throughout the whole of my studies.

Kurzfassung

Answer Set Programming (ASP) ist ein deklarativer Problemlösungsansatz aus dem Bereich der Wissensrepräsentations und -verarbeitung. Hierbei werden logische Regeln verwendet, um Wissensverarbeitung zu ermöglichen. Typische moderne Implementierungen von Problemlösern für ASP durchlaufen, bevor sie mit dem eigentlichen Lösen einer Probleminstanz beginnen, eine Grundierungsphase, welche zu einem exponentiellen Blowup des erforderlichen Speichers führen kann. Dieser Effekt wird Grounding Bottleneck genannt. Der sogenannte Lazy-Grounding Ansatz verschränkt die Grundierungs- und Lösungsphasen, um den Grounding Bottleneck zu vermeiden. Bei Lazy-Grouding werden Regeln inkrementell grundiert, sobald sie für den Lösungsprozess von Interesse sind. Derzeit arbeitet der Lazy-Grounding ASP Problemlöser ALPHA mit einer monoton wachsenden Menge von Regeln, die aus dem Grundierungsprozess resultieren. Das führt potentiell dazu, dass nicht mehr benötigte grundierte Regeln die Performance des Problemlösers negativ beeinträchtigen, nachdem diese weiterhin gespeichert und verarbeitet werden. Diese Arbeit präsentiert eine Technik mit dem Namen Reboots, die es erlaubt, grundierte Regeln wieder zu entfernen. Reboots werden definiert und ihre Korrektheit wird formal bewiesen. Weiters werden mehrere Strategien für die Anwendung von Reboots vorgeschlagen. Reboots und zugehörige Strategien werden für den ALPHA Problemlöser definiert und implementiert. Experimentelle Ergebnisse zeigen, dass mittels Reboots für bestimmte Probleme größere Instanzen gelöst und Performance-Verbesserungen erzielt werden können.

Abstract

Answer set programming (ASP) is a declarative problem solving approach from the area of knowledge representation and reasoning. It uses logical rules for the purpose of knowledge processing. The state-of-the-art solvers for answer set programs initially, before starting the solving process, perform a grounding phase that potentially leads to an exponential blowup in memory. This effect is called the grounding bottleneck. The lazy-grounding approach interleaves the grounding and solving phases to avoid this grounding bottleneck. In lazy-grounding, rules are grounded incrementally as soon as they are relevant to the solving process. So far the lazy-grounding ASP solver Alpha considers a monotonically growing set of rules obtained from grounding. This potentially leads to ground rules becoming detrimental to the search performance as they are stored and processed for longer than they remain of interest to the search. This thesis proposes a technique, called reboots, that allows the removal of ground rules obtained during the search. Reboots are defined and their correctness is proven formally. Several strategies for the decision when to perform reboots are proposed. Furthermore, reboots and proposed strategies are implemented for the Alpha solver. Experimental results show that, using this technique, larger instances can be solved and performance improvements can be achieved for some problems.

Contents

CHAPTER 1

Introduction

Answer set programming [EIK09, Lif19, GKKS12, GL88], or ASP for short, is a declarative problem solving approach based on formal logic. It provides a rich specification language and is used in academia as well as in industry, where examples of its application include diagnosis, configuration, design and planning [FFS+18]. Encodings in ASP use logical facts and rules. Similarly, encodings can be split up into an instance-specific and a problem-specific part respectively. Rules in ASP describe a form of logical implication and possibly contain variables. A rule containing variables represents a set of rules and the step of translating rules with variables to rules without is called grounding. An example of a rule with variables would be the following:

$$option(A, M, T) \leftarrow task(A), machine(M), time(T).$$

This rule could be used to formalize the options for some assignment problem, where tasks are assigned to machines at certain time slots. It represents rules for all combinations of tasks, machines and time slots defined in the remaining program, resulting in a number of rules equal to the number of these combinations.

This demonstrates that the result of grounding is potentially quite large. In the worst case the grounding step can lead to an exponential blowup of the encoding. Most state-of-the-art solvers for ASP compute the whole grounding upfront, resulting in a grounding and a solving phase. These systems are further referred to as ground-and-solve systems. Results of the grounding step, that are too large to fit into memory, lead to the so-called grounding bottleneck of ASP [GLM+18].

One approach to tackle the grounding bottleneck involves grounding rules incrementally only when necessary, i.e., interleaving the grounding and solving phases. This approach is called lazy-grounding and was implemented in multiple ASP solvers (cf. GASP [PDPR09], ASPeRiX [LN09a] and [DEF+12]). These early lazy-grounding solvers, while allowing to solve problems that could not be solved before, exhibited significantly worse

solving performance than ground-and-solve systems. The lazy-grounding solver Alpha [Wei17, LW17, WTF20] incorporates several techniques that are used in ground-and-solve systems and originate from the related field of satisfiability solving, to improve performance. The Alpha solver combines lazy-grounding with conflict-driven learning, a technique originating from satisfiability solving [SS96].

While lazy-grounding typically works with less necessary memory than ground-and-solve systems, there is still room for improvement in this regard. Once a rule is grounded in Alpha, the result is stored until the end of the search. In case this result is only relevant for part of the search, it could negatively impact performance after it becomes obsolete. This leads to the question, whether it is possible to avoid monotonic growth of the set of grounding results, and thus solve more problems by reducing memory consumption.

In this thesis a novel technique is proposed to remove grounding results in lazy-grounding and we call it reboots. This is possible since lazy-grounding does not make the assumption that all grounding results are known upfront [LW17]. A challenge for the definition and implementation of reboots is to find a balance between removing potentially obsolete information and retaining still relevant information. Implementing this technique in the Alpha solver provides another challenge. Since removing grounding results could potentially undo progress made, it is necessary to make sure that this does not lead the solver to explore the same part of the search space indefinitely. Furthermore, it leads to the question of when to use the reboot technique during the search process.

Our contributions include a definition of reboots as well as a modified algorithm of Alpha that incorporates reboots. Furthermore, a formal soundness and completeness proof is presented based on the proposed notion of a computation tree, which is used to formally describe a run of the modified algorithm using a tree. Since the modified algorithm is a generalization of the original, the proofs in this thesis are also applicable to the original Alpha algorithm. Specifically the provided completeness proof goes into more detail than previously existing proofs for Alpha [Wei17]. Furthermore, several reboot strategies, for when to execute a reboot within the modified solving algorithm, are proposed and an implementation of reboots in Alpha [LLTW] combined with the proposed reboot strategies is provided. Three hypotheses regarding reboots are investigated:

- H_1: The decision when to reboot can have significant impact on performance.
- H_2: Reboots can significantly improve solving performance.
- H_3: Reboots are detrimental for problems where nogoods from grounding do not become obsolete.

Finally, evidence towards the hypotheses is presented in the form of experimental analysis.

The remainder of this thesis is structured as follows. Section 2 contains definitions from the literature, that are used throughout the thesis. In Section 3 reboots are defined and a modification of the algorithm employed by Alpha is presented. Section 3 also contains

formal soundness and completeness proofs for the presented algorithm. Furthermore, multiple possible strategies for the decision, when to perform a reboot, are proposed. Finally, arguments for termination of the presented algorithm, using specific types of ASP programs and reboot strategies, are given. Section 4 contains a description of reboots within the ALPHA system, the three examined hypotheses and experimental results. Section 5 gives an overview over important related work and Section 6 concludes the thesis.

CHAPTER 2

Preliminaries

2.1 Logical Basics

Consider a *language* $\mathcal{L} = \langle Const, Func, Pred, Vars \rangle$ as a quadruple of pairwise disjoint sets, containing a set $Const$ of all constant symbols of the language, a set $Func$ of all function symbols of the language together with their arity (e.g. $f/3 \in Func$ for some ternary function symbol f), a set $Pred$ of all predicate symbols of the language together with their arity (e.g. $p/2 \in Pred$ for some binary predicate symbol p) and a set $Vars$ contains all variables of the language.

The set of terms $\text{TERMS}_{\mathcal{L}}$ over some language $\mathcal{L} = \langle Const, Func, Pred, Vars \rangle$ is inductively defined as the smallest set such that:

- Every constant symbol is a term, i.e., $\forall c \in Const : c \in \text{TERMS}_{\mathcal{L}}$.
- Every variable is a term, i.e., $\forall v \in Vars : v \in \text{TERMS}_{\mathcal{L}}$.
- Let $t_1, \ldots, t_n \in \text{TERMS}_{\mathcal{L}}$ be terms and let $f/n \in Func$ be an n-ary function symbol, then $f(t_1, \ldots, t_n)$ is also a term, i.e., $f(t_1, \ldots, t_n) \in \text{TERMS}_{\mathcal{L}}$.

The set of ground terms is defined analogously but without the second case. Thus a *ground* term is a term that contains no variables.

An atom is defined as an expression $p(t_1, \ldots, t_n)$ constructed from an n-ary predicate symbol $p/n \in Pred$ and n terms $t_1, \ldots, t_n$. More formally the set of atoms $\text{ATOMS}_{\mathcal{L}}$ over some language $\mathcal{L} = \langle Const, Func, Pred, Vars \rangle$ is defined as follows:

$$\text{ATOMS}_{\mathcal{L}} = \{p(t_1, \ldots, t_n) \mid p/n \in Pred,\ \{t_1, \ldots, t_n\} \subseteq \text{TERMS}_{\mathcal{L}}\}.$$

An atom defined this way is called *ground* if all its terms $t_1, \ldots, t_n$ are ground.

A binary literal can be an atom or its negation and the set of literals $\mathrm{LIT}_{\mathcal{L}}$ over a language $\mathcal{L}$ is defined as:

$$\mathrm{LIT}_{\mathcal{L}} = \{\mathbf{T}a \mid a \in \mathrm{ATOMS}_{\mathcal{L}}\} \cup \{\mathbf{F}a \mid a \in \mathrm{ATOMS}_{\mathcal{L}}\}.$$

A binary literal is called *ground* if its corresponding atom is ground. Let $\bar{l} \in \mathrm{LIT}_{\mathcal{L}}$ denote the negation of a binary literal $l \in \mathrm{LIT}_{\mathcal{L}}$. More formally:

$$\bar{l} = \begin{cases} \mathbf{F}a & \text{if } l = \mathbf{T}a; \\ \mathbf{T}a & \text{if } l = \mathbf{F}a. \end{cases}$$

A *nogood* δ is defined as a conjunction of $n \geq 0$ binary literals $\delta = l_1 \wedge \ldots \wedge l_n$. Nogoods are called *ground* if they contain only ground literals. From here on, nogoods are represented as sets of binary literals. The set of all atoms occurring in a nogood δ is denoted with $atoms(\delta)$ and defined as follows:

$$atoms(\{\mathbf{T}a_1, \ldots, \mathbf{T}a_k, \mathbf{F}a_{k+1}, \ldots, \mathbf{F}a_m\}) = \{a_1, \ldots, a_m\}.$$

Furthermore, the *Herbrand universe* $\mathcal{HU}(\mathcal{L})$ and the *Herbrand base* $\mathcal{HB}(\mathcal{L})$ of a language $\mathcal{L}$ are defined as follows:

$$\mathcal{HU}(\mathcal{L}) = \{t \in \mathrm{TERMS}_{\mathcal{L}} \mid t \text{ is ground}\} \quad \text{and}$$
$$\mathcal{HB}(\mathcal{L}) = \{a \in \mathrm{ATOMS}_{\mathcal{L}} \mid a \text{ is ground}\}.$$

A *Herbrand interpretation* $I \subseteq \mathcal{HB}(\mathcal{L})$, sometimes referred to just as an *interpretation*, over a language $\mathcal{L}$ is then defined as a subset of the Herbrand base of $\mathcal{L}$. An *interpretation sequence* is defined as an infinite sequence $(X_0, \ldots, X_\infty)$ of Herbrand interpretations. In the remainder of this thesis, if not specified otherwise, a finite sequence $(X_0 \ldots, X_n)$ is used to represent an infinite interpretation sequence where the last element X_n is repeated. This means that a sequence $(X_0 \ldots, X_n)$ represents the interpretation sequence $(X_0 \ldots, X_n, X_n, X_n, \ldots)$.

2.2 ASP Syntax and Semantics

2.2.1 ASP Syntax

An ASP program consists of rules. Let $h, b_1, \ldots b_n$ be atoms. Then a rule r over some language $\mathcal{L}$ is defined as a logical expression of the form:

$$h \leftarrow b_1, \ldots, b_m, \mathrm{not}\, b_{m+1}, \ldots, \mathrm{not}\, b_n. \tag{2.1}$$

The left side of the implication arrow "$\leftarrow$" is called the *head* of the rule r. It is defined by $H(r)$ for the rule r as follows:

$$H(r) = \begin{cases} \{h\} & \text{if } r \text{ contains a head atom } h; \\ \emptyset & \text{if } r \text{ contains no head atom.} \end{cases}$$

The right side is called the *body* of the rule r and can be divided into the positive body $B^+(r) = \{b_1, \ldots, b_m\}$ and the negative body $B^-(r) = \{b_{m+1}, \ldots, b_n\}$. Either head or body (but not both) of a rule can be empty. A rule r_c with $H(r_c) = \emptyset$ is a syntactic shorthand, called a *constraint*. Let $a \in \text{ATOMS}_\mathcal{L}$ be a new atom, that does not occur in $\mathcal{P}$ otherwise. Then consider a constraint of the form:

$$\leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n.$$

It represents the following rule:

$$a \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n, \text{not } a.$$

Conversely a rule r_f with $B^+(r_f) \cup B^-(r_f) = \emptyset$ is called a *fact*. The set $atoms(r)$ of atoms of a rule r is defined as $atoms(r) = H(r) \cup B^+(r) \cup B^-(r)$.

A rule r is called *ground* if all atoms $a \in atoms(r)$ are ground. Furthermore, a rule r is called *safe* if every variable that occurs in r occurs in $B^+(r)$. For the remainder of this thesis, every considered program is assumed to be safe.

An ASP program $\mathcal{P}$ is defined as a set of rules over some language $\mathcal{L}$. The set of atoms $atoms(\mathcal{P})$ of a program $\mathcal{P}$ is defined as:

$$atoms(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} atoms(r).$$

If all rules of a program $\mathcal{P}$ are ground, then $\mathcal{P}$ is also called ground. If it holds for every rule r of a program $\mathcal{P}$ that $B^-(r) = \emptyset$, then $\mathcal{P}$ is called *positive*. Whenever a program $\mathcal{P}$ is considered in the remainder of this thesis without the mention of a language, it is assumed that the program is defined over some suitable language $\mathcal{L}$.

2.2.2 Substitutions and Grounding

Given some language $\mathcal{L} = \langle Const, Func, Pred, Vars \rangle$, a substitution σ is defined as a function $\sigma : TERMS_\mathcal{L} \to TERMS_\mathcal{L}$, written in postfix notation, such that:

- σ is homomorphous, i.e., $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots t_n\sigma)$ for function terms and $c\sigma = c$ for constants.
- σ is identical almost everywhere, i.e., $\{x \mid x \text{ is a variable and } x\sigma \neq x\}$ is finite.

A substitution σ is represented by the finite set $\{x_1 \mapsto x_1\sigma, \ldots, x_k \mapsto x_k\sigma\}$ where $\{x_1, \ldots, x_k\}$ is its domain and $\{x_1\sigma, \ldots, x_k\sigma\}$ is its codomain. Given some predicate $p/n \in Pred$, in slight abuse of notation, define the application of a substitution to an atom $p(t_1, \ldots, t_n) \in ATOMS_\mathcal{L}$ as follows:

$$p(t_1, \ldots, t_n)\sigma = p(t_1\sigma, \ldots, t_n\sigma).$$

Then let some rule r be of the form shown in Equation 2.1 and define the result $r\sigma$ of applying the substitution σ to the rule r as:

$$h\sigma \leftarrow b_1\sigma, \ldots, b_m\sigma, \text{not } b_{m+1}\sigma, \ldots, \text{not } b_n\sigma.$$

The *grounding* $\text{grd}(r)$ of a rule r is defined based on substitutions as follows:

$$\text{grd}(r) = \{r\sigma \mid \sigma \text{ is a substitution s.t. } r\sigma \text{ is ground}\}.$$

Further the grounding $\text{grd}(\mathcal{P})$ of a program $\mathcal{P}$ is defined as:

$$\text{grd}(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} \text{grd}(r).$$

2.2.3 ASP Semantics

Given a program $\mathcal{P}$ over a language $\mathcal{L}$, a Herbrand interpretation $I \subseteq \mathcal{HB}(\mathcal{L})$ is called a *Herbrand model*, sometimes referred to just as a *model*, of $\mathcal{P}$ if the following holds:

$$\forall r \in \text{grd}(\mathcal{P}): \quad (B^+(r) \subseteq I \ \wedge \ B^-(r) \cap I = \emptyset) \ \Rightarrow \ H(r) \subseteq I.$$

A Herbrand model M of a program $\mathcal{P}$ is *minimal*, if there does not exist another Herbrand model M' of $\mathcal{P}$ s.t. $M' \subset M$.

Given a partially ordered set $(S, \leq)$, an *operator* is a mapping $f : S \to S$. An operator f is called *monotone* if $f(x) \leq f(y)$ whenever $x \leq y$. A *fixpoint* of an operator $f : S \to S$ is an element $x \in S$ s.t. $f(x) = x$. Furthermore, a fixpoint x^* of $f : S \to S$ is called the *least fixpoint* $\text{lfp}(f)$ if for every fixpoint $x \in S$ of f it holds that $x^* \leq x$. It can be computed by fixpoint iteration as follows: $\text{lfp}(f) = f \uparrow \omega$, where $f \uparrow 0(I) = I$, $f \uparrow (n+1)(I) = f(f \uparrow n(I))$ and $f \uparrow \omega(I) = \bigcup_{n \in \mathbb{N}} f \uparrow n(I)$. If not specified otherwise, the initial set I is assumed to be the empty set.

A *complete lattice* is a partially ordered set $(S, \leq)$ such that each subset $S' \subseteq S$ has a least upper bound $\sup(S')$ and a greatest lower bound $\inf(S')$. The Knaster-Tarski theorem states the following:

Theorem 1 (Knaster-Tarski [Tar55])**.** *Let $(S, \leq)$ be a complete lattice, let $f : S \to S$ be a monotonic operator (w.r.t. $\leq$) and let P be the set of fixpoints of f. Then P is non-empty and $(P, \leq)$ is a complete lattice.*

The direct consequence operator $T_{\mathcal{P}} : 2^{\mathcal{HB}(\mathcal{L})} \to 2^{\mathcal{HB}(\mathcal{L})}$ for some program $\mathcal{P}$, defined over language $\mathcal{L}$, is defined as follows:

$$T_{\mathcal{P}}(X) = \{a \mid \exists r\sigma \in \text{grd}(\mathcal{P}),\ a \in H(r\sigma),\ B^+(r\sigma) \subseteq X,\ B^-(r\sigma) \cap X = \emptyset\}.$$

The partially ordered set $(2^{\mathcal{HB}(\mathcal{L})}, \subseteq)$ is a complete lattice where for every subset $S' \subseteq 2^{\mathcal{HB}(\mathcal{L})}$ it holds:

$$\sup(S') = \bigcup_{I \in S'} I \quad \text{and} \quad \inf(S') = \bigcap_{I \in S'} I.$$

Note that for a positive ground program $\mathcal{P}$ the $T_\mathcal{P}$ operator is monotone. By the Knaster-Tarski theorem, it follows for a positive ground program $\mathcal{P}$ that there exists a least fixpoint $\text{lfp}(T_\mathcal{P})$ that is the unique minimal Herbrand model of $\mathcal{P}$.

The semantics of ASP programs was originally defined based on the stable model semantics proposed by Gelfond and Lifschitz in 1988 [GL88]. They described the Gelfond-Lifschitz-reduct (GL-reduct) to obtain solutions, called *answer sets*, of an ASP program. Let $\mathcal{P}$ be a ground ASP program and let $I \subseteq \mathcal{HB}(\mathcal{L})$ be a interpretation. Then the GL-reduct $GL_I(\mathcal{P})$ is defined as a function that takes an ASP program $\mathcal{P}$ and returns a new set of rules obtained by:

- removing every rule r from $\mathcal{P}$ where $B^-(r) \cap I \neq \emptyset$, and
- setting $B^-(r) = \emptyset$ for each remaining rule r.

Note that $GL_I(\mathcal{P})$ is a positive program. Then an interpretation $I \subseteq \mathcal{HB}(\mathcal{L})$ is an answer set (originally called *stable model*) of a ground program $\mathcal{P}$, if I is the unique minimal Herbrand model of $GL_I(\mathcal{P})$. Further an interpretation $I \subseteq \mathcal{HB}(\mathcal{L})$ is an answer set of a non-ground program $\mathcal{P}$, if I is an answer set of the grounding $\text{grd}(P)$. Finally, $AS(\mathcal{P})$ is defined as the set of all answer sets of a program $\mathcal{P}$.

2.3 Fundamentals of Alpha

2.3.1 Ternary Assignments

Assignments in the algorithm of the ALPHA solver [Wei17] are ternary, meaning that they involve three possible truth values: the classical boolean values `true` (represented by **T**) and `false` (represented by **F**) as well as a third value `must-be-true` (represented by **M**). Intuitively $\mathbf{M}a \in A$ for some atom a represents, that a needs to be true (according to some nogood), but there is no justification for its truth, i.e., a rule that fired, under assignment A.

The set of ternary literals $\text{LIT}^3_\mathcal{L}$ over some language $\mathcal{L}$ is defined as:

$$\text{LIT}^3_\mathcal{L} = \{\mathbf{T}a \mid a \in \text{ATOMS}_\mathcal{L}\} \cup \{\mathbf{F}a \mid a \in \text{ATOMS}_\mathcal{L}\} \cup \{\mathbf{M}a \mid a \in \text{ATOMS}_\mathcal{L}\}.$$

The *boolean projection* $l^\mathcal{B}$ of a ternary literal l maps a ternary to a binary literal and is defined as:

$$l^\mathcal{B} = \begin{cases} \mathbf{T}a & \text{if } l = \mathbf{T}a; \\ \mathbf{T}a & \text{if } l = \mathbf{M}a; \\ \mathbf{F}a & \text{if } l = \mathbf{F}a. \end{cases}$$

A ternary assignment A over language $\mathcal{L}$ is a subset of all ternary literals over $\mathcal{L}$. Formally, the set of ternary assignments $\text{ASSIGN}^3_\mathcal{L}$ is defined as:

$$\text{ASSIGN}^3_\mathcal{L} = 2^{\text{LIT}^3_\mathcal{L}}.$$

A ternary assignment A is considered to be *consistent* if there does not exist an atom $a \in \text{ATOMS}_\mathcal{L}$ s.t. both $\mathbf{T}a \in A \vee \mathbf{M}a \in A$ and $\mathbf{F}a \in A$ hold. In the remainder of this thesis only consistent ternary assignment are considered.

Further the *boolean projection* $A^\mathcal{B}$ of a ternary assignment A is defined as:

$$A^\mathcal{B} = \{l^\mathcal{B} \mid l \in A\}.$$

The set of all atoms occurring in a ternary assignment $A \in \text{ASSIGN}^3_\mathcal{L}$ is denoted with $atoms(A)$ and defined as follows:

$$atoms(\{\mathbf{T}a_1, \ldots, \mathbf{T}a_j, \mathbf{F}a_{j+1}, \ldots, \mathbf{F}a_k, \mathbf{M}a_{k+1}, \ldots, \mathbf{M}a_m\}) = \{a_1, \ldots a_m\}.$$

2.3.2 Nogood Representation

In the solving process of Alpha there exist special nogoods with a designated head literal. Nogoods with head allow making justified inferences and result in propagation to truth value **T**. Intuitively this corresponds to some rule firing under the current assignment. Other propagation steps only propagate to truth value **M** or **F**. For nogoods with head, the head literal is underlined. For example the nogood $\{\underline{\mathbf{T}a}, \mathbf{T}b, \mathbf{T}c\}$ has the head literal $\mathbf{T}a$. For a nogood δ with head let $hd(\delta)$ denote the head literal of δ.

Based on nogoods without and with head, there are two possible types of propagation of a nogood in the Alpha solver: weak and strong propagation. They occur when a nogood δ is weakly- or strongly-unit respectively.

- A nogood δ is *weakly-unit* under assignment A for literal l if $\delta \setminus A^\mathcal{B} = \{l\}$ and $\bar{l} \notin A^\mathcal{B}$.
- A nogood δ is *strongly-unit* under assignment A for literal l if δ is a nogood with head, $\delta \setminus A = \{l\}$, $hd(\delta) = l$ and $\bar{l} \notin A$.

A nogood δ is considered to be *violated* under some ternary assignment A if $\delta \subseteq A$.

For some ground rule $r\sigma$ obtained from rule $r \in \mathcal{P}$ and substitution σ, let the *rule atom* $\beta(r, \sigma)$ be defined as a fresh atom with $\beta(r, \sigma) \notin \text{ATOMS}_\mathcal{L}$ where $\mathcal{L}$ is the language of the program $\mathcal{P}$. Intuitively $\beta(r, \sigma)$ represents whether the rule $r\sigma$ fires under some assignment.

Ground rules can be represented by a set of nogoods as defined by Weinzierl [Wei17] as follows: Given a ground rule $r\sigma$ of the form shown in Equation 2.1, the set of representation nogoods $ng_{re}(r\sigma)$ is defined as:

$$\begin{aligned} ng_{re}(r\sigma) = \{&\{\underline{\mathbf{F}\beta(r,\sigma)}, \mathbf{T}b_1, \ldots, \mathbf{T}b_k, \mathbf{F}b_{k+1}, \ldots, \mathbf{F}b_n\}, \{\underline{\mathbf{F}h}, \mathbf{T}\beta(r,\sigma)\}, \\ &\{\mathbf{T}\beta(r,\sigma), \mathbf{F}b_1\}, \ldots, \{\mathbf{T}\beta(r,\sigma), \mathbf{F}b_k\}, \\ &\{\mathbf{T}\beta(r,\sigma), \mathbf{T}b_{k+1}\}, \ldots, \{\mathbf{T}\beta(r,\sigma), \mathbf{T}b_n\}\}. \end{aligned}$$

Furthermore, for each such ground rule $r\sigma$ with $B^-(r\sigma) \neq \emptyset$ the *choice atoms* $cOn(r,\sigma)$ and $cOff(r,\sigma)$ are also defined as fresh atoms with $cOn(r,\sigma) \notin \text{ATOMS}_\mathcal{L}$ and $cOff(r,\sigma) \notin \text{ATOMS}_\mathcal{L}$. A rule $r\sigma$ is considered *applicable* under some set of atoms $At \subseteq \text{ATOMS}_\mathcal{L}$ if it holds:

$$B^+(r\sigma) \subseteq At \quad \text{and} \quad B^-(r\sigma) \cap At = \emptyset.$$

Further $r\sigma$ is considered *applicable* under some assignment $A \in \text{ASSIGN}^3_\mathcal{L}$ if it is applicable under the set $\{a \mid \mathbf{T}a \in A\}$. The set of choice nogoods $ng_{ch}(r\sigma)$ encodes the applicability of rules and is defined as:

$$\begin{aligned} ng_{ch}(r\sigma) = \{\{&\mathbf{F}\,cOn(r,\sigma), \mathbf{T}b_1, \ldots, \mathbf{T}b_k\}, \\ &\{\mathbf{F}\,cOff(r,\sigma), \mathbf{T}b_{k+1}\}, \ldots, \{\mathbf{F}\,cOff(r,\sigma), \mathbf{T}b_n\}\}. \end{aligned}$$

The full nogood representation $ng(r\sigma)$ of a rule $r\sigma$ is then defined as $ng(r\sigma) = ng_{re}(r\sigma) \cup ng_{ch}(r\sigma)$. More details on this encoding can be found in the original definition by Weinzierl [Wei17].

The set of nogoods $\text{ngREP}_\mathcal{P}$ resulting from the whole nogood representation of a program $\mathcal{P}$ is defined as:

$$\text{ngREP}_\mathcal{P} = \bigcup_{r\sigma \in \text{grd}(\mathcal{P})} ng(r\sigma).$$

The set of *auxiliary atoms* $\text{ATOMS}^{\text{aux}}_\mathcal{P}$ for a program $\mathcal{P}$ is defined as:

$$\text{ATOMS}^{\text{aux}}_\mathcal{P} = \mathit{atoms}(\text{ngREP}_\mathcal{P}) \setminus \text{ATOMS}_\mathcal{L}.$$

The atoms contained in a program $\mathcal{P}$, i.e. $a \in \mathit{atoms}(\mathcal{P})$, are referred to as *non-auxiliary atoms*.

2.3.3 Lazy Grounding

The lazy-grounding in Alpha is based on the notion of a grounder memory and a lazy-grounding strategy [TWF19]. A *grounder memory* is defined as a subset $G \subseteq \mathcal{HB}(\mathcal{L})$ of the Herbrand base. The set $GM = 2^{\mathcal{HB}(\mathcal{L})}$ denotes all possible grounder memories. To allow different degrees of permissiveness in grounding, a lazy-grounding strategy defines which rules are grounded depending on the current assignment and grounder memory.

Consider an ASP program $\mathcal{P}$ and let $R \subseteq \mathcal{P}$ be the set of non-ground rules in $\mathcal{P}$. Then a *lazy-grounding strategy* $gs : \text{ASSIGN}^3_\mathcal{L} \times GM \times R \to GM \times 2^{\text{grd}(\mathcal{P})}$ maps a ternary assignment $A \in \text{ASSIGN}^3_\mathcal{L}$, a grounder memory $G \in GM$ and a non-ground rule $r \in R$ to a new grounder memory G' and a set of new ground rules $R' \subseteq \text{grd}(\mathcal{P})$.

Taupe et al. [TWF19] presented multiple lazy-grounding strategies with the default grounding strategy being the most conservative one, i.e., grounding the smallest set of rules. This strategy is based on two notions, namely inactive rules and rules of interest. A ground rule $r\sigma \in \text{grd}(\mathcal{P})$ is *inactive* if (at least) one of the following holds:

- $B^+(r\sigma)$ contains an atom over a predicate p s.t. there does not exist a rule $r'\sigma'$ where the atom $a \in H(r'\sigma')$ is over p.
- $B^-(r\sigma)$ contains an atom a s.t. there exists a fact $r_f \in \mathcal{P}$ in the program with $a \in H(r_f)$.

A ground rule $r\sigma \in \text{grd}(\mathcal{P})$ is *of interest* w.r.t. the assignment $A \in \text{ASSIGN}^3_{\mathcal{L}}$ if it holds:

$$\{\mathbf{T}a \mid a \in B^+(r\sigma)\} \subseteq A^{\mathcal{B}}.$$

Then the *default grounding strategy* for a program $\mathcal{P}$ is defined as a lazy-grounding strategy gs_{def} where $gs_{def}(A, G, r) = (G', R')$ s.t. $G' = \{a \mid \mathbf{T}a \in A\}$ and:

$$R' = \{r\sigma \in \text{grd}(\mathcal{P}) \mid r\sigma \text{ is of interest w.r.t. } A \text{ and not inactive}\}.$$

Taupe et al. showed that a lazy-grounding ASP solver, that is sound and complete for the default grounding strategy gs_{def}, is also sound and complete for their other proposed (more permissive) grounding strategies. This thesis considers gs_{def} as the grounding strategy used in the solving algorithm.

2.3.4 The Alpha Algorithm

The algorithm employed by the Alpha solver [Wei17], further called *AlphaASP*, is based on the concept of a computation sequence originally defined by Liu et al. [LPST10]. Let a *computation sequence* be defined over some program $\mathcal{P}$ as an interpretation sequence $(X_0, \ldots, X_\infty)$ with $X_0 = \emptyset$ that satisfies the following properties:

(1) Persistence of beliefs:
$$\forall i \geq 1 : \; X_{i-1} \subseteq X_i$$

(2) Revision:
$$\forall i \geq 1 : \; X_i \subseteq T_{\mathcal{P}}(X_{i-1})$$

(3) Convergence:
$$X_\infty = \bigcup_{i=0}^{\infty} X_i = T_{\mathcal{P}}(X_\infty)$$

(4) Persistence of reasons:
$$\forall i \geq 1 \; \forall a \in X_i \setminus X_{i-1} \; \exists r_a \in \text{grd}(\mathcal{P}) : \\ a \in H(r_a) \; \wedge \; \forall j \geq i-1 : \Big(B^+(r_a) \subseteq X_j \wedge B^-(r_a) \cap X_j = \emptyset\Big)$$

Since an interpretation sequence is defined as a sequence of Herbrand interpretations, this definition is equivalent to what Liu et al. [LPST10] refer to as a *persistent computation*. Furthermore, the following Lemma was proven by them.

Lemma 1 (Proposition 3 in [LPST10]). *Let $\mathcal{P}$ be an ASP program. A set X is an answer set of $\mathcal{P}$ if and only if there exists a computation sequence $(X_0, \ldots, X_\infty)$ where $X_\infty = X$.*

The *AlphaASP* algorithm was originally defined by Weinzierl [Wei17]. Note that the version presented in Algorithm 2.1 splits up the set of nogoods Δ into static nogoods Δ_S and learned nogoods Δ_L. Static nogoods are those obtained from grounding and represent the static problem description. Learned nogoods are those learned from conflicts based on resolution. The technique of learning nogoods from conflicts in this way was originally developed in the field of satisfiability solving [SS96].

For some run of the *AlphaASP* algorithm, let the *decision level* $dl(l)$ of a ternary literal $l \in A$ be defined as the value of variable *dlev* at the point where l was added to A.

The algorithm begins with an initial request to the grounding component, i.e., $lazyGround_{\mathcal{P}}(A, G)$ for A and G both being empty. Here the grounding strategy is applied to the input program to obtain a set of new nogoods that represents the newly grounded rules. This step depends on the current assignment and grounder memory since a lazy-grounding strategy also takes both of these as inputs.

The main solving loop depends on the condition defined by function *wasExhausted*. Intuitively the function returns whether the search space was fully explored. Formally, it returns 1 if:

- the empty nogood was learned or obtained from grounding, i.e., $\emptyset \in \Delta_S \cup \Delta_L$, or
- the current decision level is negative, i.e. $dlev < 0$ holds.

The function *wasExhausted* returns 0 otherwise, indicating that further search is needed. The two cases where *wasExhausted* returns 1 are further referred to as *termination conditions*.

The loop begins with exhaustive unit-propagation in the *propagate* function. The semantics of *propagate* is defined based on immediate unit-propagation. Immediate unit-propagation in *AlphaASP* is defined as:

$$\begin{aligned} Pr_\Delta(A) = A \cup \{\mathbf{T}a \mid \exists \delta \in \Delta,\ \delta \text{ is strongly-unit under } A \text{ for } s = \mathbf{F}a\} \\ \cup \{\mathbf{M}a \mid \exists \delta \in \Delta,\ \delta \text{ is weakly-unit under } A \text{ for } s = \mathbf{F}a\} \\ \cup \{\mathbf{F}a \mid \exists \delta \in \Delta,\ \delta \text{ is weakly-unit under } A \text{ for } s = \mathbf{T}a\}. \end{aligned}$$

Performing $n \geq 1$ steps of immediate unit-propagation $Pr^n_\Delta(A)$ is defined inductively as:

$$Pr^n_\Delta(A) = \begin{cases} A & \text{if } n = 0; \\ Pr_\Delta(Pr^{n-1}_\Delta(A)) & \text{otherwise.} \end{cases} \tag{2.2}$$

Let $\mathrm{lfp}_A(f)$ denote the least fixpoint of f where A is used as the initial set instead of $\emptyset$. The function *propagate* consists of exhaustive application of immediate unit-propagation. More formally it is defined as the least fixpoint using the initial set A:

$$propagate(\Delta, A) = \mathrm{lfp}_A(Pr_\Delta).$$

Next the loop splits into multiple different cases with the following meaning:

- In case `(conflict)`, a conflict occurred, meaning a nogood δ was violated. Then the function *analyze* is used to analyze the conflict and learn a new nogood δ_l by resolution. Furthermore, a new decision level $dlev_{bj}$ is computed at which no nogood is violated. Before adding the new nogood, the function *backjump* leads to an unassignment of all assignments in A with decision levels above $dlev_{bj}$. More formally, the *backjump* function is defined as follows:

$$backjump(A, dlev_{bj}) = \{l \in A \mid dl(l) \leq dlev_{bj}\}.$$

- In case `(ground)`, the function *wasExtended* is used to check whether the assignment was extended in the previous iteration. The function *wasExtended* takes an assignment and returns 1 if a new literal was added since the last call to *wasExtended*, and 0 otherwise. The first time the function is called, it returns 1. In case the assignment was extended, a request to the grounding component is performed to potentially obtain new nogoods representing ground rules. The default grounding strategy gs_{def} is assumed to be used.

- In case `(choice)`, a choice is made. This means that an atom from the active choice points is chosen and assigned to `true`. The *active choice points* acp depend on the current set of nogoods Δ and the current assignment A. They represent the options for firing rules and are defined formally as:

$$\begin{aligned} acp(\Delta, A) = \Big\{ \beta(r,\sigma) \in atoms(\Delta) \;\Big|\; & \mathbf{T}cOn(r,\sigma) \in A \\ & \wedge\ \mathbf{T}cOff(r,\sigma) \notin A \wedge \mathbf{M}cOff(r,\sigma) \notin A \\ & \wedge\ \mathbf{T}\beta(r,\sigma) \notin A \wedge \mathbf{F}\beta(r,\sigma) \notin A \Big\}. \end{aligned}$$

  How an atom is selected from the active choice points does not influence the correctness of the solving process, but can have practical impact on performance.

- In case `(close)`, the algorithm has reached a point where no propagation, grounding or guessing needs to be performed. Here all unassigned atoms are assigned `false` since other atoms do not have any justification to be in the potential answer set. This is done to ensure that every atom is assigned some ternary truth-value. There is still the possibility that some atom is assigned to `must-be-true`.

- In case `(answer)`, an answer set is found since no unjustified atoms, i.e., atoms assigned `must-be-true`, remain. The answer set is added to the set $\mathcal{AS}$. Let A be the current assignment. Then a new nogood δ_{enum} is computed by the function *enumNG* as follows:

$$\delta_{enum} = \{\mathbf{T}\beta(r, \sigma) \in A\}.$$

  Nogoods of this type are called *enumeration nogoods* and ensure that the algorithm does not reach the same answer set multiple times. Afterwards the function *backtrack* is used to backtrack to, and flip, the most recent decision. Given some assignment $l \in A$, let the function *decision* indicate whether a literal was assigned by a decision as follows:

$$\mathit{decision}(l) = \begin{cases} 1 & \text{if } l \text{ was assigned in case } \texttt{(choice)} \text{ of the algorithm;} \\ 0 & \text{otherwise.} \end{cases}$$

  The set of decisions D_A is defined as:

$$D_A = \{a \in \mathit{atoms}(A) \mid \mathbf{T}a \in A,\ \mathit{decision}(\mathbf{T}a) = 1\}.$$

  If $D_A \neq \emptyset$ holds for some assignment A, then the decision atom a_A to flip is defined such that:

$$\{a_A\} = \operatorname*{argmax}_{a \in D_A} dl(\mathbf{T}a).$$

  The new assignment $\hat{A}$ after backtracking is defined as:

$$\hat{A} = \begin{cases} \{l \in A \mid dl(l) < dl(\mathbf{T}a_A)\} \cup \{\mathbf{F}a_A\} & \text{if } \quad D_A \neq \emptyset; \\ \emptyset & \text{otherwise.} \end{cases}$$

  Then the *backtrack* function is defined as:

$$\mathit{backtrack}(A) = \begin{cases} (dl(\mathbf{T}a_A), \hat{A}) & \text{if } \quad D_A \neq \emptyset; \\ (-1, \hat{A}) & \text{otherwise.} \end{cases}$$

- In case `(backtrack)`, a backtrack is performed since there are unjustified atoms remaining in the assignment.

When the loop terminates, all answer sets have been found and the set $\mathcal{AS}$ of all answer sets is returned.

Algorithm 2.1: The *AlphaASP* algorithm

Input: A (non-ground) normal logic program $\mathcal{P}$
Output: The set $AS(\mathcal{P})$ of all answer sets of $\mathcal{P}$

$\mathcal{AS} \leftarrow \emptyset$
$A \leftarrow \emptyset$
$\Delta_S \leftarrow \emptyset$
$\Delta_L \leftarrow \emptyset$
$G \leftarrow \emptyset$
$dlev \leftarrow 0$
$wasExtended(A)$
$(\Delta_S, G) \leftarrow lazyGround_{\mathcal{P}}(A, G)$
while $wasExhausted() = 0$ **do**
  $A \leftarrow propagate(\Delta_S \cup \Delta_L, A)$ (propagate)
  if $\exists \delta \in \Delta_S \cup \Delta_L : \delta \subseteq A$ **then** (conflict)
    $(\delta_l, dlev_{bj}) \leftarrow analyze(\delta, \Delta_S \cup \Delta_L, A)$
    $A \leftarrow backjump(A, dlev_{bj})$
    $dlev \leftarrow dlev_{bj}$
    $\Delta_L \leftarrow \Delta_L \cup \{\delta_l\}$
  else if $wasExtended(A) = 1$ **then** (ground)
    $(\Delta'_S, G) \leftarrow lazyGround_{\mathcal{P}}(A, G)$
    $\Delta_S \leftarrow \Delta_S \cup \Delta'_S$
  else if $acp(\Delta_S \cup \Delta_L, A) \neq \emptyset$ **then** (choice)
    $dlev \leftarrow dlev + 1$
    $s \leftarrow select(acp(\Delta_S \cup \Delta_L, A))$
    $A \leftarrow A \cup \{\mathbf{T}s\}$
  else if $atoms(\Delta_S \cup \Delta_L) \setminus atoms(A) \neq \emptyset$ **then** (close)
    $A \leftarrow A \cup \{\mathbf{F}a \mid a \in atoms(\Delta_S \cup \Delta_L) \setminus atoms(A)\}$
  else if $\{a \mid \boldsymbol{M}a \in A,\ \boldsymbol{T}a \notin A\} = \emptyset$ **then** (answer)
    $\mathcal{AS} \leftarrow \mathcal{AS} \cup \{\{a \in \text{ATOMS}_{\mathcal{L}} \mid \mathbf{T}a \in A\}\}$
    $\delta_{enum} \leftarrow enumNg(A)$
    $\Delta_L \leftarrow \Delta_L \cup \{\delta_{enum}\}$
    $(dlev, A) \leftarrow backtrack(A)$
  else (backtrack)
    $(dlev, A) \leftarrow backtrack(A)$
  end
end
return $\mathcal{AS}$

CHAPTER 3

Reboots in Lazy-Grounding

The set of static nogoods maintained by lazy-grounding ASP solving in ALPHA grows monotonically. This means that after a nogood has been obtained from grounding, it will be stored and processed until the search finishes. The proposed concept of a reboot eliminates this property and allows removal of stored static nogoods.

Lazy-grounding ASP solving is divided conceptually into the solving and grounding component, which need to have synchronized information about which ground rules have been instantiated, i.e., which nogoods the grounder has already passed to the solver. The grounder also indirectly stores this information to avoid passing the same nogoods to the solver repeatedly. If the solver removes parts of the grounding, this information needs to be propagated between the two components.

3.1 Reboots

The proposed definition of reboots uses a restricted set of nogoods, which is referred to as the learnable nogoods of a program. The *learnable nogoods* $\Delta^{\mathcal{P}}$ of a program $\mathcal{P}$ are defined as:

$$\Delta^{\mathcal{P}} = \{\delta \mid \delta \text{ is a nogood with } \mathit{atoms}(\delta) \subseteq \text{ATOMS}_{\mathcal{L}} \cup \text{ATOMS}_{\mathcal{P}}^{\text{aux}}\}.$$

The learnable nogoods represent all nogoods that the solver could theoretically obtain based on the non-auxiliary as well as auxiliary atoms of the input program $\mathcal{P}$. The current set of learned nogoods in the solver is thus a subset of the learnable nogoods. To retain learned nogoods containing some rule atom $\beta(r, \sigma)$, a reboot takes the set of learned nogoods as one of its inputs and adds $r\sigma$ to the rules in its output. The intuitive goal of this is to preserve learned information.

A reboot incorporates the chosen grounding strategy and a subset of the learnable nogoods $\Delta_L \subseteq \Delta^{\mathcal{P}}$ to obtain a set of ground rules by applying the grounding strategy to all rules of the program and adding additional rules based on Δ_L. More formally:

Definition 1. *Let $\mathcal{P} = \{r_1, ..., r_n\}$ be an ASP program with n (non-ground) rules, $\mathcal{A}$ the set of partial assignments, $Strat_{gr}$ the set of possible grounding strategies, GM the set of possible grounder memories, $\Delta_L \subseteq \Delta^{\mathcal{P}}$ a subset of the learnable nogoods and $grd(\mathcal{P})$ the grounding of program program $\mathcal{P}$. A* reboot $\mathrm{rbt} : \mathcal{A} \times Strat_{gr} \times 2^{\Delta^{\mathcal{P}}} \to GM \times 2^{grd(\mathcal{P})}$ *is then defined as:*

$$\mathrm{rbt}(A, strat_{gr}, \Delta_L) = \left(g_n, \left(\bigcup_{0 \leq i \leq n} R_i \right) \cup R_{\Delta_L} \right).$$

based on the sequence $((g_0, R_0), (g_1, R_1), \ldots, (g_n, R_n))$, which is inductively defined as:

$$g_0 = \emptyset, \quad R_0 = \emptyset,$$
$$\forall i \in \{1, \ldots, n\} : (g_i, R_i) = strat_{gr}(A, g_{i-1}, r_i)$$

and the set R_{Δ_L}, which is defined as:

$$R_{\Delta_L} = \{r\sigma \mid \beta(r, \sigma) \in atoms(\Delta_L)\}.$$

For a reboot rbt, the resulting grounder memory $\mathrm{rbt}_{gm} : \mathcal{A} \times Strat_{gr} \times 2^{\Delta^{\mathcal{P}}} \to GM$ and ground rules $\mathrm{rbt}_{gr} : \mathcal{A} \times Strat_{gr} \times 2^{\Delta^{\mathcal{P}}} \to 2^{grd(\mathcal{P})}$ are defined as:

$$\mathrm{rbt}_{gm}(A, strat_{gr}, \Delta_L) = g_n \quad \text{and} \quad \mathrm{rbt}_{gr}(A, strat_{gr}, \Delta_L) = R'$$

where $\mathrm{rbt}(A, strat_{gr}, \Delta_L) = (g_n, R')$.

To obtain the nogood representation instead of the ground rules themselves, the function $\mathrm{rbt}_{ng} : \mathcal{A} \times Strat_{gr} \times 2^{\Delta^{\mathcal{P}}} \to GM \times 2^{\Delta^{\mathcal{P}}}$ maps the resulting rules to their respective nogoods as follows:

$$\mathrm{rbt}_{ng}(A, strat_{gr}, \Delta_L) = \left(\mathrm{rbt}_{gm}(A, strat_{gr}, \Delta_L), \bigcup_{r\sigma \in \mathrm{rbt}_{gr}(A, strat_{gr}, \Delta_L)} ng(r\sigma) \right).$$

Performing a reboot at some point during the solving process, where $A \in \mathcal{A}$ is the current partial assignment, $strat_{gr} \in Strat_{gr}$ is the chosen grounding strategy and Δ'_L is the current set of learned nogoods, amounts to setting the grounder memory to $g' \in GM$ and setting the static nogoods to $\Delta'_S \subseteq \Delta^{\mathcal{P}}$, where $\mathrm{rbt}_{ng}(A, strat_{gr}, \Delta'_L) = (g', \Delta'_S)$.

Algorithm 3.1, further referred to as *AlphaRebootASP*, incorporates reboots into the solving algorithm employed by Alpha [Wei17], which was presented in Section 2.3.4. All parts of the algorithm not mentioned explicitly behave the same way as in the *AlphaASP* algorithm. The function *rebootAdvised* represents a heuristic that decides when reboots are performed. This decision and corresponding heuristics are covered in Section 3.5.

Note that each nogood in Δ_S is obtained from some ground rule of the program. Further note that when a ground rule $r\sigma$ is instantiated by the grounder, all nogoods $\delta \in ng(r\sigma)$ in its representation are passed to the solver. The set of *synchronized ground rules* at some point in the algorithm is defined as the set of all ground rules $r\sigma \in \mathrm{grd}(\mathcal{P})$ s.t. it holds $ng(r\sigma) \subseteq \Delta_S$.

Algorithm 3.1: The *AlphaRebootASP* algorithm

Input: A (non-ground) normal logic program $\mathcal{P}$ and a grounding strategy $strat_{gr}$
Output: The set $AS(\mathcal{P})$ of all answer sets of $\mathcal{P}$

$\mathcal{AS} \leftarrow \emptyset$
$A \leftarrow \emptyset$
$\Delta_S \leftarrow \emptyset$
$\Delta_L \leftarrow \emptyset$
$G \leftarrow \emptyset$
$dlev \leftarrow 0$
$wasExtended(A)$
$(\Delta_S, G) \leftarrow lazyGround_{\mathcal{P}}(A, G)$
while $wasExhausted() = 0$ **do**
  $A \leftarrow propagate(\Delta_S \cup \Delta_L, A)$ (propagate)
  if $\exists \delta \in \Delta_S \cup \Delta_L : \delta \subseteq A$ **then** (conflict)
    $(\delta_l, dlev_{bj}) \leftarrow analyze(\delta, \Delta_S \cup \Delta_L, A)$
    $A \leftarrow backjump(A, dlev_{bj})$
    $dlev \leftarrow dlev_{bj}$
    $\Delta_L \leftarrow \Delta_L \cup \{\delta_l\}$
  else if $wasExtended(A) = 1$ **then** (ground)
    $(\Delta'_S, G) \leftarrow lazyGround_{\mathcal{P}}(A, G)$
    $\Delta_S \leftarrow \Delta_S \cup \Delta'_S$
  else if $rebootAdvised()$ **then** (reboot)
    $(G, \Delta_S) \leftarrow \mathrm{rbt}_{ng}(A, strat_{gr}, \Delta_L)$
  else if $acp(\Delta_S \cup \Delta_L, A) \neq \emptyset$ **then** (choice)
    $dlev \leftarrow dlev + 1$
    $s \leftarrow select(acp(\Delta_S \cup \Delta_L, A))$
    $A \leftarrow A \cup \{\mathbf{T}s\}$
  else if $atoms(\Delta_S \cup \Delta_L) \setminus atoms(A) \neq \emptyset$ **then** (close)
    $A \leftarrow A \cup \{\mathbf{F}a \mid a \in atoms(\Delta_S \cup \Delta_L) \setminus atoms(A)\}$
  else if $\{a \mid \boldsymbol{M}a \in A,\ \boldsymbol{T}a \notin A\} = \emptyset$ **then** (answer)
    $\mathcal{AS} \leftarrow \mathcal{AS} \cup \{\{a \in \mathrm{ATOMS}_{\mathcal{L}} \mid \mathbf{T}a \in A\}\}$
    $\delta_{enum} \leftarrow enumNg(A)$
    $\Delta_L \leftarrow \Delta_L \cup \{\delta_{enum}\}$
    $(dlev, A) \leftarrow backtrack(A)$
  else (backtrack)
    $(dlev, A) \leftarrow backtrack(A)$
  end
end
return $\mathcal{AS}$

3.2 The Computation Tree

To show soundness and completeness of the presented algorithm, we introduce the notion of a computation tree. Such a computation tree represents the parts of the search space, i.e., the potential answer sets, explored by the algorithm. The root of the tree corresponds to the empty set and atoms are added along paths from the root.

For some run of the *AlphaRebootASP* algorithm (as previously for *AlphaASP*), let the *decision level* $dl(l)$ of a ternary literal $l \in A$ be defined as the value of the variable *dlev* at the point where l was added to A. This happens within *AlphaRebootASP* for example in these cases: propagation in `(propagate)`, making a choice in `(choice)` and closing the assignment in `(close)`. In the third case only negative literals are added. Note that in *AlphaRebootASP* literals are only removed from A in cases `(conflict)`, `(answer)` and `(backtrack)`.

In the remainder of this thesis, the term *computation point* is used to refer to the state, including the current variable values, at one of the following points during the execution of the *AlphaRebootASP* algorithm:

- the point at the end of an iteration
- the point after any step of immediate unit-propagation performed in `(propagate)` (as defined in Equation 2.2 of Section 2.3.4)
- the point after `(propagate)`

In the following, nodes in the constructed tree will represent computation points. Note that cases `(conflict)`, `(ground)`, `(reboot)`, `(choice)`, `(close)`, `(answer)`, `(backtrack)` are disjoint and exhaustive for a single iteration. Thus each iteration has a unique case that is chosen during the algorithm run.

The first n solving loop iterations of a *AlphaRebootASP* run on some program $\mathcal{P}$ over language $\mathcal{L}$ can be viewed as the traversal of a rooted computation tree of ternary assignments, where leafs correspond to either successes, failures or the point after the n-th iteration in the search for answer sets. Recall that $Pr^s_\Delta(A)$ is defined as the result of s steps of immediate unit-propagation (see Section 2.3.4).

Example 3.2.1. *Figure 3.1 provides an example for such a computation tree. The tree contains nodes representing propagation steps and iteration cases, which are denoted with P and I respectively in the figure. Intuitively the tree represents the possibilities to extend the assignment based on currently applicable rules.*

Definition 2. *The* ***computation tree*** $T^n = (V^n, E^n)$ *of some n-iteration run of the AlphaRebootASP algorithm is defined as a rooted tree where edges point away from the root. Each node* $v \in V^n$ *is labeled with a ternary assignment* $assn(v)$*, a decision level* $dl(v)$*, an iteration number* $iterNum(v)$ *and an iteration case* $iterCase(v)$*. The tree is defined inductively (over the number i of iterations) as follows:*

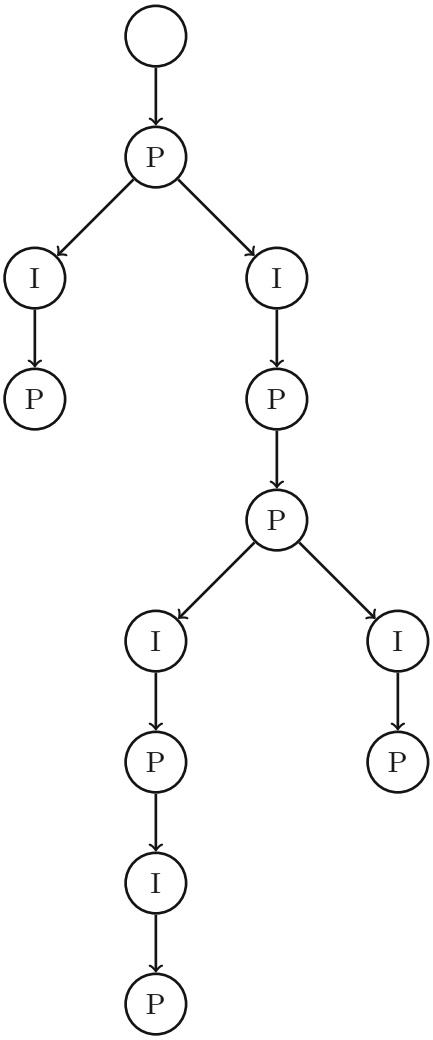

Figure 3.1: Example of a computation tree

- ***Base case*** $i = 0$*: In this case the tree only consists of the root node* v_0 *representing the beginning of the algorithm run.*

$$V^0 = \{v_0\}, \quad E^0 = \emptyset, \quad assn(v_0) = \emptyset, \quad dl(v_0) = 0.$$

- ***Inductive case****: Let* $T^{i-1} = (V^{i-1}, E^{i-1})$*, with* $V^{i-1} = \{v_0, \ldots, v_{k-1}\}$*, be the computation tree after* $i-1$ *iterations. Further let* $(case_i)$ *be the iteration case in iteration* i *and let* A_i *be the value of variable* A *at the computation point at the end of iteration* i*. Then let:*

$$m = \begin{cases} k & \text{if no steps of unit-propagation are performed in iteration } i; \\ k+p-1 & \text{if } p \geq 1 \text{ is the number of steps of unit-propagation in iteration } i; \end{cases}$$

$$V' = V^{i-1} \cup \{v_k, \ldots, v_m\},$$
$$E' = E^{i-1} \cup \{(v_{j-1}, v_j) \mid k \leq j \leq m\},$$
$$\forall j \in \{k, \ldots, m\} : assn(v_j) = Pr_{\Delta}^{(1+j-k)}(assn(v_{k-1})), \quad dl(v_j) = dl(v_{k-1}),$$
$$iterNum(v_j) = i \quad and \quad iterCase(v_j) = (case_i),$$

as well as:

$$assn(v_{m+1}) = A_i, \quad iterNum(v_{m+1}) = i, \quad iterCase(v_{m+1}) = (case_i).$$

Recall that a termination condition holds if the current decision level is negative or the empty nogood was learned or obtained from grounding, i.e., if wasExhausted() returns 1. There are several possible cases:

- *If a termination condition holds after iteration i, then $T^i = (V^i, E^i)$ is defined as:*
$$V^i = V', \quad E^i = E'.$$
- *If no termination condition holds and $(case_i)$ is `(choice)`, then $T^i = (V^i, E^i)$ is defined as:*
$$V^i = V' \cup \{v_{m+1}\}, \quad E^i = E' \cup \{(v_m, v_{m+1})\}, \quad dl(v_{m+1}) = dl(v_{k-1}) + 1.$$
- *If no termination condition holds and $(case_i)$ is `(ground)`, `(reboot)` or `(close)`, then $T^i = (V^i, E^i)$ is defined as:*
$$V^i = V' \cup \{v_{m+1}\}, \quad E^i = E' \cup \{(v_m, v_{m+1})\}, \quad dl(v_{m+1}) = dl(v_{k-1}).$$
- *If no termination condition holds and $(case_i)$ is `(conflict)`, let v' be the last node on the path from the root v_0 to v_{k-1} in T^{i-1} where $dl(v') = dlev$ after iteration i. Then $T^i = (V^i, E^i)$ is defined as:*
$$V^i = V' \cup \{v_{m+1}\}, \quad E^i = E' \cup \{(v', v_{m+1})\}, \quad dl(v_{m+1}) = dl(v').$$
- *If no termination condition holds and $(case_i)$ is `(answer)` or `(backtrack)`, let v' be the last node on the path from the root v_0 to v_{k-1} in T^{i-1} where $dl(v') = dlev - 1$ holds after iteration i. Then $T^i = (V^i, E^i)$ is defined as:*
$$V^i = V' \cup \{v_{m+1}\}, \quad E^i = E' \cup \{(v', v_{m+1})\}, \quad dl(v_{m+1}) = dl(v') + 1.$$

*For all cases where a node v_{m+1} is added, this node v_{m+1} is called an **iteration node**. All other nodes are called **propagation nodes**.*

Example 3.2.2. *Figure 3.2 illustrates the two options for the position of the iteration node. The left side is an example of an iteration node added as the child of the last propagation node in the same iteration. In this case the assignment is simply extended or left unchanged. The right side is an example of an iteration node added as the child of some node further up the tree. This node is determined by the decision level that backjumping or backtracking ends up at. For a backjump, unassignment of literals above the backtrack level is performed. For a backtrack, additionally the previous decision is flipped and added as a new decision.*

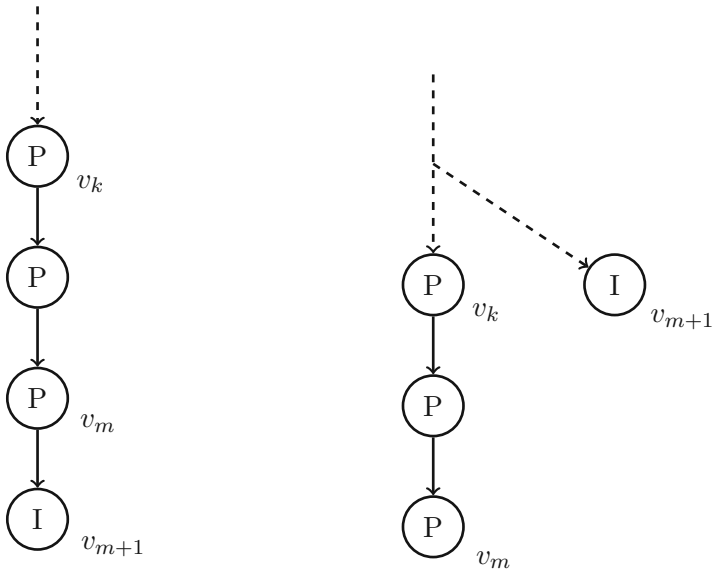

Figure 3.2: Options for the position of the iteration node v_{m+1}

Example 3.2.3. *Figure 3.3 illustrates the computation tree, with all defined node labels, after 6 iterations of AlphaRebootASP on the program $\mathcal{P}_{ex}$, where $\mathcal{P}_{ex}$ is defined as follows:*

$$a \leftarrow \text{not } b.$$
$$b \leftarrow \text{not } a.$$
$$c \leftarrow a.$$
$$d \leftarrow b.$$
$$e \leftarrow d.$$

The rules $r_1, \ldots, r_5$ refer to the rules of $\mathcal{P}_{ex}$ in the same order. The labels for some node v in the example represent $assn(v)$, $dl(v)$, $iterNum(v)$ and $iterCase(v)$ respectively. The left subtree represents the algorithm exploring potential answer sets where r_1 fires and the right subtree where it does not fire.

In the following we consider an inductively defined computation tree T^n. Note that the constructed tree T^n has $k \geq n$ nodes. Each node represents a specific computation point. An iteration node $v_t \in V^n$ represents the computation point at the end of iteration $iterNum(v_t)$. A propagation node $v_s \in V^n$ represents the computation point after a specific amount of immediate unit-propagation steps during its respective iteration $iterNum(v_s)$. If $v_s = v_m$ for some inductive case, then v_s represents the computation point after `(propagate)`. Otherwise it represents the computation point after $1 + s - k$ steps of unit propagation during `(propagate)`. In the remainder of this thesis, a node is also used to refer to the respective computation point it represents.

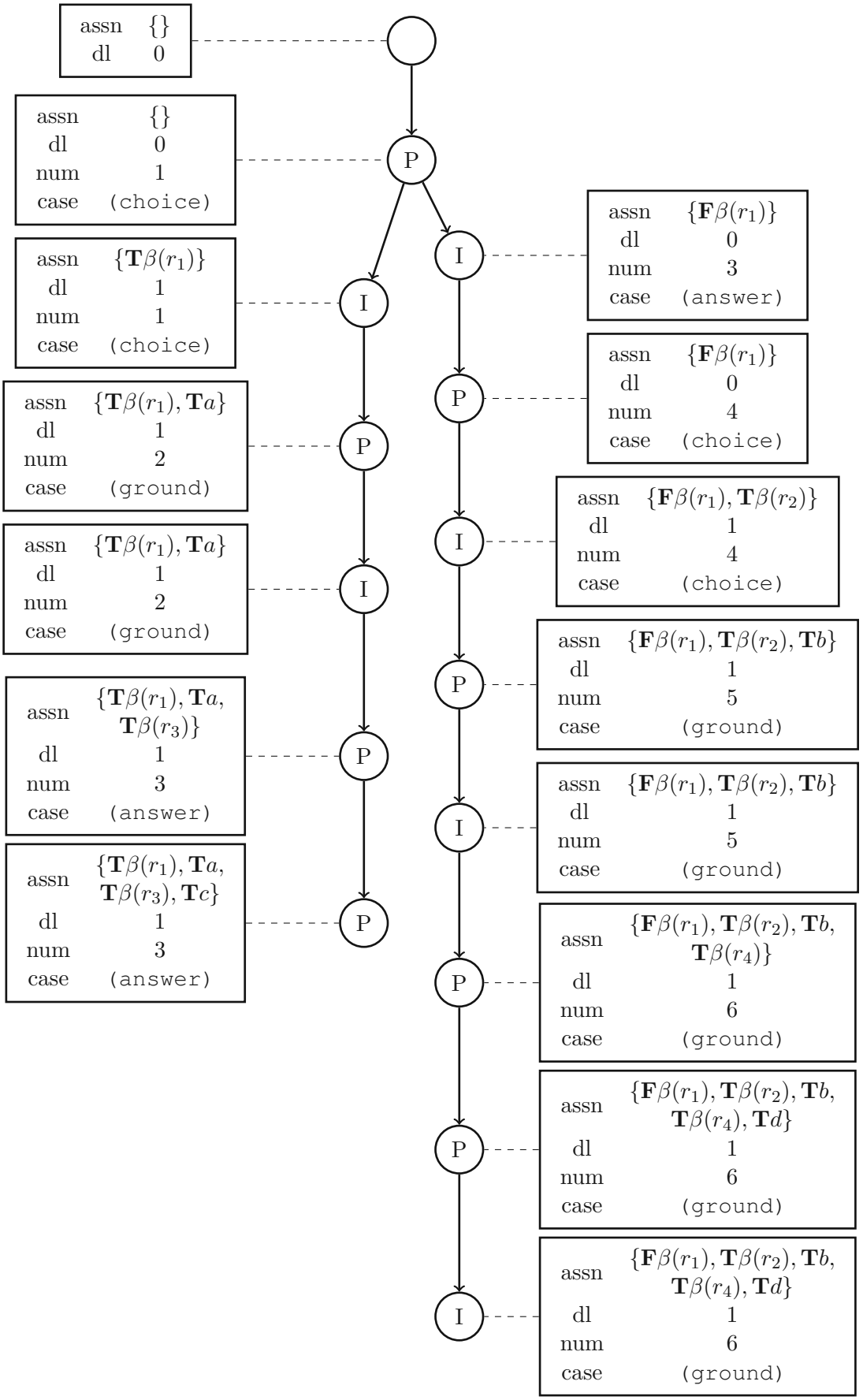

Figure 3.3: Computation tree after 6 iterations of *AlphaRebootASP* on $\mathcal{P}_{ex}$

A propagation node $v_s \in V^n$ is called a *failed* node if $\mathit{iterCase}(v_s) = \texttt{(conflict)}$. An iteration node $v_t \in V^n$ is called *failed* if some nogood $\delta \in \Delta_S \cup \Delta_L$ is violated at v_t. All other nodes in V^n are called *non-failed*. Intuitively a failed node represents an assignment that leads to a conflict after exhaustive propagation. Note that at some non-failed propagation node there also cannot be a violated nogood as otherwise the respective iteration would lead to a conflict. Further note that no non-failed node can be reached by a directed path from a failed node. Failed propagation nodes lead to backjumps from conflicts and failed iteration nodes lead to the end of the search.

The interpretation $\mathit{inter}(v_j)$ is defined for all nodes $v_j \in V^n$ as:

$$\mathit{inter}(v_j) = \{a \in \mathrm{ATOMS}_{\mathcal{L}} \mid \mathbf{T}a \in \mathit{assn}(v_j)\}.$$

Furthermore, an interpretation sequence $\mathit{seq}(v_j)$ is associated with each node $v_j \in V^n$ of the tree, where d is its distance from the root node, as follows. Consider the (unique) path $u_0, \ldots, u_d$ from the root node v_0 to the node v_j in T^n (i.e. $u_d = v_j$). Then $\mathit{seq}(v_j)$ is defined as:

$$\mathit{seq}(v_j) = (\mathit{inter}(u_0), \ldots, \mathit{inter}(u_d)).$$

Parent, child and descendant relations are defined based on the directed edges of T^n. The parent $\mathit{par}(v_j)$ of a node $v_j \in V^n$ is defined as follows:

$$\mathit{par}(v_j) = \begin{cases} v_{par} & \text{if } (v_{par}, v_j) \in E^i; \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Note that v_0 is the only node that does not have a defined parent while for all other nodes there is exactly a single incoming edge by construction of T^n.

The children $\mathit{chl}(v_j)$ of a node $v_j \in V^n$ are defined as follows:

$$\mathit{chl}(v_j) = \{v'_j \mid (v_j, v'_j) \in E^n\}.$$

The descendant relation intuitively represents the reflexive and transitive closure of the child relation or alternatively, which nodes are reachable from some starting node. Formally, the set of descendants $\mathit{dsc}(v_j)$ of a node $v_j \in V^n$ is defined as:

$$\mathit{dsc}(v_j) = \{v' \mid \text{there exists a directed path from } v_j \text{ to } v'\}.$$

Note that every node is reachable from itself by the empty path. The set of non-failed descendants $\mathit{dsc}_{nf}(v_j)$ of a node $v_j \in V^n$ is simply defined as the set of non-failed nodes in $\mathit{dsc}(v_j)$.

Note that for some iteration node $v_{it} \in V^n$ the parent $\mathit{par}(v_{it})$ must be a propagation node. The node $\mathit{par}(v_{it})$ represents the computation point after `(propagate)`. This holds since for each iteration node $v_{it} \in V^n$ there exists a following propagation node as child of v_{it} with the same decision level.

3.3 Soundness

The proof of soundness as well as the used lemmas assume that *AlphaRebootASP* runs on some program $\mathcal{P}$ over language $\mathcal{L}$ for n iterations.

Looking at the computation tree, the nodes of a subtree are all consecutive in the enumeration resulting from the tree construction. Lemma 2 describes this property.

Lemma 2. *Given a computation tree $T^n = (V^n, E^n)$ and nodes $v_i, v_j, v_k \in V^n$ where $i < j < k$, if $v_k \in dsc(v_i)$ then it follows $v_j \in dsc(v_i)$.*

Proof. Towards a contradiction assume that $v_j \notin dsc(v_i)$. Then, to show by induction that for $j' \geq j$ it holds $v_{j'} \notin dsc(v_i)$, consider that the base case $v_j \notin dsc(v_i)$ holds by assumption. In the induction step consider $v_{\mu+1}$, where $\mu \geq j$, and use the induction hypothesis that $v_\mu \notin dsc(v_i)$.

- If $par(v_{\mu+1}) = v_\mu$, then $v_{\mu+1} \notin dsc(v_i)$ follows from the induction hypothesis.
- If $par(v_{\mu+1}) \neq v_\mu$, then by definition $par(v_{\mu+1})$ is a node on the path from the root v_0 to v_μ. Thus it follows $v_\mu \in dsc(par(v_{\mu+1}))$. Then since $v_\mu \notin dsc(v_i)$ it follows $par(v_{\mu+1}) \notin dsc(v_i)$. Further it follows $v_{\mu+1} \notin dsc(v_i)$.

From the inductive argument it follows that $v_k \notin dsc(v_i)$ since $k \geq j$. This leads to a contradiction. □

When considering the computation tree nodes along a path from the root to some leaf, the current assignment only grows. Lemma 3 shows this property formally.

Lemma 3. *Given a computation tree $T^n = (V^n, E^n)$ and a (non-root) computation tree node $v_i \in V^n \setminus \{v_0\}$, it holds:*

$$assn(par(v_i)) \subseteq assn(v_i).$$

Proof. Unassignments only happen during *backjump* and *backtrack*, i.e. in cases `(conflict)`, `(answer)` and `(backtrack)`. There are four possible cases:

- If v_i is a propagation node, it follows $assn(par(v_i)) \subseteq assn(v_i)$ since no unassignments are performed during propagation.
- If v_i is an iteration node corresponding to case `(ground)`, `(choice)` or `(close)`, then consider the changes made to the assignment in these cases. No unassignment is performed in any of these cases. It thus follows $assn(par(v_i)) \subseteq assn(v_i)$.
- If v_i is an iteration node corresponding to case `(reboot)`, then recall that a reboot does not modify the current assignment. It thus follows $assn(par(v_i)) \subseteq assn(v_i)$.

- If v_i is an iteration node corresponding to case `(conflict)`, `(answer)` or `(backtrack)`, then by definition of the computation tree, no unassignment happened at decision levels up to $dl(par(v_i))$. Furthermore, all assignments in $assn(par(v_i))$ were done at decision levels up to $dl(par(v_i))$. Thus no literals were unassigned compared to the parent node and it follows $assn(par(v_i)) \subseteq assn(v_i)$.

□

In the remainder of this thesis, a node v_i is considered to represent a *computation point after exhaustive propagation* if v_i is a propagation node and has no child, that is also a propagation node.

At any non-failed computation tree node v_i during a run of the *AlphaRebootASP* algorithm the nogood representation ensures, that if a rule atom is assigned `true`, then no atom in the negative body of the corresponding rule will be assigned `true`. Lemma 4 captures this.

Lemma 4. *Given a computation tree $T^n = (V^n, E^n)$ and a non-failed computation tree node $v_i \in V^n$, if it holds $\boldsymbol{T}\beta(r,\sigma) \in assn(v_i)$ and $ng(r\sigma) \subseteq \Delta_S$ at v_i, then it follows for all non-failed descendants $v_d \in dsc_{nf}(v_i)$ that $\{\boldsymbol{T}b \mid b \in B^-(r\sigma)\} \cap assn(v_d) = \emptyset$.*

Proof. There are three possible cases for the node v_i:

- If v_i is a propagation node and $\forall b \in B^-(r\sigma) : \mathbf{F}b \in assn(v_i)$ holds, then by Lemma 3 it follows for all non-failed descendants $v_d \in dsc_{nf}(v_i)$:

  $$\forall b \in B^-(r\sigma) : \mathbf{F}b \in assn(v_d).$$

  It further follows for these non-failed descendants:

  $$\{\mathbf{T}b \mid b \in B^-(r\sigma)\} \cap assn(v_d) = \emptyset.$$

- If v_i is a propagation node and $\exists b' \in B^-(r\sigma) : \mathbf{F}b' \notin assn(v_i)$, then a nogood $\delta \in ng(r\sigma)$ of the form $\delta = \{\mathbf{T}\beta(r,\sigma), \mathbf{T}b'\}$ is weakly-unit under $A = assn(v_i)$. Thus unit-propagation was not performed exhaustively yet at node v_i. Note that nodes representing intermediate unit-propagation steps (i.e. all but the final step) have exactly a single child node. Starting from v_i and repeatedly traversing the tree towards the single child node then eventually leads to some node v_e, representing the computation point after exhaustive propagation. Further note that since v_i is a non-failed node, there is no conflict at v_e. It then follows for node v_e and further, by Lemma 3, for all non-failed descendants $v_d \in dsc_{nf}(v_e)$:

  $$\forall b \in B^-(r\sigma) : \mathbf{F}b \in assn(v_d).$$

It further follows for these non-failed descendants:

$$\{\mathbf{T}b \mid b \in B^-(r\sigma)\} \cap assn(v_d) = \emptyset.$$

Note that $v_e \in dsc(v'_d)$ holds for each node v'_d on the path from v_i to v_e. Thus it follows for v'_d, by Lemma 3, and further for all non-failed descendants $v_d \in dsc(v_i)$ of node v_i:

$$\{\mathbf{T}b \mid b \in B^-(r\sigma)\} \cap assn(v_d) = \emptyset.$$

- If v_i is an iteration node, then by construction of the computation tree, v_i has at most one child v'_i. Further if such a child exists, then it is a propagation node. Since propagation does not modify the set of nogoods, one of the previous two cases applies for v'_i, if v'_i is non-failed. Furthermore, v_i is non-failed and thus no nogood δ of the form $\delta = \{\mathbf{T}\beta(r,\sigma), \mathbf{T}b\}$ for some $b \in B^-(r\sigma)$ is violated. Thus it holds for all non-failed descendants $v_d \in dsc_{nf}(v_i)$:

$$\{\mathbf{T}b \mid b \in B^-(r\sigma)\} \cap assn(v_d) = \emptyset.$$

□

At any non-failed computation tree node v_i during a run of the *AlphaRebootASP* algorithm the nogood representation ensures, that if a rule atom is assigned `true` in the assignment of v_i, then the positive body of the respective rule is assigned `true` and no negative body atom is assigned `true` at any descendant of v_i. This is captured by Lemma 5.

Lemma 5. *Given a computation tree $T^n = (V^n, E^n)$, a non-failed computation tree node $v_i \in V^n$ and an arbitrary rule atom $\beta(r,\sigma)$, if $\boldsymbol{T}\beta(r,\sigma) \in assn(v_i) \setminus assn(par(v_i))$ holds, then it follows for all non-failed descendants $v_d \in dsc_{nf}(v_i)$:*

$$\forall b \in B^+(r\sigma): \ \boldsymbol{T}b \in assn(v_d) \quad \textit{and} \quad \forall b' \in B^-(r\sigma): \ \boldsymbol{T}b' \notin assn(v_d).$$

Proof. There are two possible cases for how $\mathbf{T}\beta(r,\sigma)$, for ground rule $r\sigma$ of the form in Equation 2.1, could have been added to $assn(par(v_i))$:

- If $\mathbf{T}\beta(r,\sigma)$ was added by strong propagation on some nogood $\delta \in ng(r\sigma)$ where $ng(r\sigma) \subseteq \Delta_S$ holds at both $par(v_i)$ and v_i (since a propagation step does not modify the set of nogoods), then this nogood must have been of the form:

$$\delta = \{\underline{\mathbf{F}\beta(r,\sigma)}, \mathbf{T}b_1, \ldots, \mathbf{T}b_k, \mathbf{F}b_{k+1}, \ldots \mathbf{F}b_m\}.$$

Then since δ was strongly-unit under $assn(par(v_i))$, it follows:

$$\forall b \in B^+(r\sigma): \ \mathbf{T}b \in assn(par(v_i)) \quad \text{and} \quad \forall b' \in B^-(r\sigma): \ \mathbf{T}b' \notin assn(par(v_i)).$$

Since this means that v_i is a propagation node where $\mathbf{T}\beta(r,\sigma)$ was added through unit-propagation, it follows:

$$assn(v_i) = assn(par(v_i)) \cup \{\mathbf{T}\beta(r,\sigma)\}.$$

Then it further follows:

$$\forall b \in B^+(r\sigma) : \mathbf{T}b \in assn(v_i) \quad \text{and} \quad \forall b' \in B^-(r\sigma) : \mathbf{T}b' \notin assn(v_i).$$

Thus since $ng(r\sigma) \subseteq \Delta_S$ at v_i, by Lemma 4 it follows for all non-failed descendants $v_d \in dsc_{nf}(v_i)$:

$$\{\mathbf{T}b \mid b \in B^-(r\sigma)\} \cap assn(v_d) = \emptyset.$$

- If $\mathbf{T}\beta(r,\sigma)$ was added during a choice in case `(choice)`, then at the point it was added $\beta(r,\sigma) \in acp(\Delta_S \cup \Delta_L, A)$ did hold, where $A = assn(par(v_i))$.

  Thus by definition of acp, it follows that $\mathbf{T}cOn(r,\sigma) \in A$. The choice atom $cOn(r,\sigma)$ can only be assigned `true` through strong propagation on a nogood δ of the form:

  $$\delta = \{\underline{\mathbf{F}cOn(r,\sigma)}, \mathbf{T}b_1, \ldots, \mathbf{T}b_k\}.$$

  Then there exists some node v_a on the path from the root v_0 to v_i s.t. $\mathbf{T}cOn(r,\sigma) \in assn(v_a) \setminus assn(par(v_a))$. This means that $\mathbf{T}cOn(r,\sigma)$ was assigned at this propagation node v_a. Note that this means that $ng(r\sigma) \subseteq \Delta_S$ at v_a. It follows $\{\mathbf{T}b_1, \ldots, \mathbf{T}b_k\} \subseteq assn(v_a)$ and since $v_i \in dsc(v_a)$, by Lemma 3, further:

  $$\forall b \in B^+(r\sigma) : \mathbf{T}b \in assn(v_i).$$

  Next, to show that $ng(r\sigma) \subseteq \Delta_S$ holds at v_i, consider when $ng(r\sigma)$ could be removed from Δ_S before node v_i. This could only happen through a reboot performed earlier at some iteration node v_r where $a < r < i$. By Lemma 2 it follows that $v_r \in dsc(v_a)$. Thus by Lemma 3 it follows $\{\mathbf{T}b_1, \ldots, \mathbf{T}b_k\} \subseteq assn(v_r)$. Then rule $r\sigma$ is of interest w.r.t. $assn(v_r)$. Further $r\sigma$ is not inactive since it has already been a result of grounding. Thus for the reboot $\mathrm{rbt}(A, strat_G, \Delta_L)$ at v_r it holds $r\sigma \in \mathrm{rbt}_{gr}(A, strat_G, \Delta_L)$. Thus $ng(r\sigma) \subseteq \Delta_S$ holds after an arbitrary such reboot and it follows that $ng(r\sigma) \subseteq \Delta_S$ holds at v_i.

  Since it holds $ng(r\sigma) \subseteq \Delta_S$ at v_i and $\mathbf{T}\beta(r,\sigma) \in assn(v_i)$, by Lemma 4 it follows for all non-failed descendants $v_d \in dsc_{nf}(v_i)$:

  $$\{\mathbf{T}b \mid b \in B^-(r\sigma)\} \cap assn(v_d) = \emptyset.$$

Then in both cases it holds for all non-failed descendants $v_d \in dsc_{nf}(v_i)$:

$$\forall b' \in B^-(r\sigma) : \mathbf{T}b' \notin assn(v_d).$$

In both cases it also holds $\forall b \in B^+(r\sigma) : \mathbf{T}b \in assn(v_i)$. By Lemma 3 it follows for all non-failed descendants $v_d \in dsc_{nf}(v_i)$:

$$\forall b \in B^+(r\sigma) : \mathbf{T}b \in assn(v_d).$$

□

In the *AlphaRebootASP* algorithm, as in the *AlphaASP* algorithm, every non-auxiliary atom is supported. This means that whenever a non-auxiliary atom a is assigned `true`, there exists a rule $r\sigma$ that is considered to have fired. For a non-failed propagation node v_i, that represents the assignment of $\mathbf{T}a$, let this be formally described by Lemma 6.

Lemma 6. *Given a computation tree $T^n = (V^n, E^n)$ for a program $\mathcal{P}$ and the current assignment $assn(v_i)$ at an arbitrary non-failed node $v_i \in V^n$, let $a \in \text{ATOMS}_{\mathcal{L}}$ be an arbitrary non-auxiliary atom s.t. $\boldsymbol{T}a \in assn(v_i) \setminus assn(par(v_i))$. Then it follows for every non-failed descendant $v_d \in dsc(par(v_i))$:*

$$\exists r\sigma \in \text{grd}(\mathcal{P}) : \Big(a \in H(r\sigma) \ \wedge \ \forall b \in B^+(r\sigma) : \ \boldsymbol{T}b \in assn(v_d) \\ \wedge \ \forall b' \in B^-(r\sigma) : \ \boldsymbol{T}b' \notin assn(v_d)\Big).$$

Proof. A non-auxiliary atom is only added to the assignment through strong propagation. By construction of the nogood representation, the only nogoods with a non-auxiliary atom a in the head literal are of the form $\delta = \{\underline{\mathbf{F}a}, \mathbf{T}\beta(r,\sigma)\}$. By construction of these nogoods (see Section 2.3.2) it follows that $a \in H(r\sigma)$. By definition of strongly-unit nogoods, it then follows that, for a positive literal $\mathbf{T}a$ assigned through propagation of nogood δ under assignment $assn(par(v_i))$, there exists a rule $r\sigma$ such that $\mathbf{T}\beta(r,\sigma) \in assn(par(v_i))$. Then there exists some non-failed node v_a on the path from the root v_0 to $par(v_i)$ s.t. $\mathbf{T}\beta(r,\sigma) \in assn(v_a) \setminus assn(par(v_a))$. Since it holds $par(v_i) \in dsc(v_a)$ and thus $dsc(par(v_i)) \subseteq dsc(v_a)$, by Lemma 5 it follows for all non-failed descendants $v_d \in dsc_{nf}(par(v_i))$:

$$\forall b \in B^+(r\sigma) : \ \mathbf{T}b \in assn(v_d) \quad \textit{and} \quad \forall b' \in B^-(r\sigma) : \ \mathbf{T}b' \notin assn(v_d).$$

□

When an answer set is found during a run of *AlphaRebootASP*, then the solver knows about all applicable rules, i.e. all such rules are part of the synchronized ground rules. Lemma 7 describes this property.

Lemma 7. *Given a computation tree $T^n = (V^n, E^n)$ for a program $\mathcal{P}$, a propagation node $v_i \in V^n$ after exhaustive propagation with $iterCase(v_i) =$ `(answer)` after which an answer set $A' = \{\boldsymbol{T}a \mid a \in assn(v_i)\}$ is found and a ground rule $r\sigma \in \text{grd}(\mathcal{P})$ s.t. $\{\boldsymbol{T}b \mid b \in B^+(r\sigma)\} \subseteq assn(v_i)$ and $\{\boldsymbol{T}b \mid b \in B^-(r\sigma)\} \cap assn(v_i) = \emptyset$, then $ng(r\sigma) \subseteq \Delta_S$ holds at v_i.*

Proof. Assume, towards a contradiction, that $ng(r\sigma) \not\subseteq \Delta_S$ holds at v_i. This means that $r\sigma$ is not part of the synchronized ground rules at v_i. Consider that $r\sigma$ is of interest w.r.t. $assn(v_i)$. To show that $r\sigma$ is not inactive, assume towards a contradiction that it is inactive. Then there are two (not necessarily disjoint, but exhaustive) cases:

- If there exists an atom $p(t_1, \ldots, t_z) \in B^+(r\sigma)$ over a predicate p s.t. there does not exist a rule $r'\sigma'$ where the atom $a \in H(r'\sigma')$ is over p, then consider that $\{\mathbf{T}b \mid b \in B^+(r\sigma)\} \subseteq assn(v_i)$. Thus $\mathbf{T}p(t_1, \ldots, t_z) \in assn(v_i)$ holds. Further it follows by Lemma 6 that there exists some rule $r'\sigma' \in \text{grd}(\mathcal{P})$ with $p(t_1, \ldots, t_z) \in H(r'\sigma')$, which is a contradiction to the definition of an inactive rule.

- If $B^-(r\sigma)$ contains an atom a s.t. there exists a fact $r_f \in \mathcal{P}$ with $a \in H(r_f)$ in the program, then consider the fact r_f. Since grounding was applied exhaustively before the answer set was added in case `(answer)`, it follows that r_f was already returned by grounding. Also r_f would have been returned by a reboot at any point since it is always of interest and never inactive. Thus $ng(r_f) \subseteq \Delta_S$ holds. Consider that it holds $\{\mathbf{T}b \mid b \in B^-(r\sigma)\} \cap assn(v_i) = \emptyset$ as well as $a \in atoms(\Delta_S \cup \Delta_L)$ and cases `(close)` and `(backtrack)` do not apply after v_i. From this it follows $\mathbf{F}a \in assn(v_i)$. Then nogood $\delta = \{\underline{\mathbf{F}a}\} \in ng(r_f)$ is violated under $assn(v_i)$. Thus case `(conflict)` would be applied after v_i, which is a contradiction since case `(answer)` was applied.

Thus $r\sigma$ is not inactive.

Further, the assignment did not change since the most recent grounding step as otherwise it would hold $iterCase(v_i) =$ `(ground)`. Thus $r\sigma$ was a result of the most recent grounding step. A reboot at any point since then would have also returned $r\sigma$ since $r\sigma$ would have still been of interest and not inactive. Then $r\sigma$ is still part of the synchronized ground rules at v_i, which is a contradiction and therefore $ng(r\sigma) \subseteq \Delta_S$ holds at v_i. □

Theorem 2 (Soundness)**.** *The algorithm AlphaRebootASP is sound, i.e., for every program $\mathcal{P}$ over some language $\mathcal{L}$, it holds AlphaRebootASP$(\mathcal{P}) \subseteq AS(\mathcal{P})$.*

Proof. By construction, the *AlphaRebootASP* algorithm only adds answer sets to the set $\mathcal{AS}$ at leaf nodes of the tree T^n in case `(answer)`, since a backtrack happens after adding an answer set to $\mathcal{AS}$. Each leaf node v_l of the tree has an associated interpretation sequence $seq(v_l)$. More specifically, some leaf nodes have an associated computation sequence $seq(v_l)$. In the following it will be shown that each leaf node v of the tree, where *AlphaRebootASP* adds the positive atoms of the corresponding assignment $assn(v)$ to $\mathcal{AS}$, contains a computation sequence $seq(v)$.

Assume that v_m is an arbitrary leaf node, after which *AlphaRebootASP* adds $\{a \mid \mathbf{T}a \in assn(v_m)\}$ to $\mathcal{AS}$. Note that this means that v_m is a propagation node after exhaustive propagation. Let $(X_0, \ldots, X_k)$ be the sequence resulting from $seq(v_m)$. Then $(X_0, \ldots, X_k)$ is a computation sequence which can be verified by checking the four properties as follows:

(1) By Lemma 3 it holds $\forall i \in \{1, \ldots, n\} : assn(par(v_i)) \subseteq assn(v_i)$. Thus by definition of $seq(v_i)$, it follows $\forall j \in \{1, \ldots, k\} : X_{j-1} \subseteq X_j$.

(2) Consider an arbitrary interpretation X_j, with $j \geq 1$, in the sequence $seq(v_m)$. Then by definition of $seq(v_m)$, there exists some (non-failed) node v_i with $X_j = \{a \mid \mathbf{T}a \in assn(v_i)\}$. Further consider an arbitrary atom $a \in X_j$ in this interpretation. Then consider the node v_a along the path from the root v_0 to v_i where $\mathbf{T}a \in assn(v_a) \setminus assn(par(v_a))$. Since a is non-auxiliary and $par(v_i) \in dsc_{nf}(par(v_a))$, it follows by Lemma 6 that:

$$\exists r\sigma \in grd(\mathcal{P}) : \Big(a \in H(r\sigma) \ \wedge \ \forall b \in B^+(r\sigma) : \ \mathbf{T}b \in assn(par(v_i)) \\ \wedge \ \forall b' \in B^-(r\sigma) : \ \mathbf{T}b' \notin assn(par(v_i))\Big).$$

Thus by definition of $seq(v_m)$ and $T_\mathcal{P}$, it follows $X_j \subseteq T_\mathcal{P}(X_{j-1})$. Further by the choice of X_j it follows $\forall j \in \{1, \ldots, k\} : \ X_j \subseteq T_\mathcal{P}(X_{j-1})$.

(3) By the choice of v_m it follows that an answer set is added in case `(answer)` after propagation node v_m. Thus $wasExtended(A) = 0$ holds at v_m since otherwise case `(ground)` would apply instead. Towards a contradiction, assume that $X_k \neq T_\mathcal{P}(X_k)$. Then there are two possible (not necessarily disjoint, but exhaustive) cases:

- If it holds $\exists a \in X_k \setminus T_\mathcal{P}(X_k)$, then consider that $X_k = \{a \mid \mathbf{T}a \in assn(v_m)\}$. Further consider the node v_a along the path from the root v_0 to v_m where $\mathbf{T}a \in assn(v_a) \setminus assn(par(v_a))$. Since a is non-auxiliary and $v_m \in dsc(par(v_a))$, it follows by Lemma 6 that:

$$\exists r\sigma \in grd(\mathcal{P}) : \Big(a \in H(r\sigma) \ \wedge \ \forall b \in B^+(r\sigma) : \ \mathbf{T}b \in assn(v_m) \\ \wedge \ \forall b' \in B^-(r\sigma) : \ \mathbf{T}b' \notin assn(v_m)\Big).$$

  Then by definition of $seq(v_m)$ and $T_\mathcal{P}$, it follows $a \in T_\mathcal{P}(X_k)$, resulting in a contradiction.

- If it holds $\exists a \in T_\mathcal{P}(X_k) \setminus X_k$, then there exists some rule $r\sigma \in \mathrm{grd}(\mathcal{P})$ such that:

$$a \in H(r\sigma), \quad B^+(r\sigma) \subseteq X_k, \quad B^-(r\sigma) \cap X_k = \emptyset.$$

  It follows by Lemma 7 that $ng(r\sigma) \subseteq \Delta_S$ holds at v_m. Then there exists a nogood $\delta \in ng(r\sigma)$ of the form $\delta = \{\underline{\mathbf{F}\beta(r,\sigma)}, \mathbf{T}b_1, \ldots, \mathbf{T}b_{l'}, \mathbf{F}b_{l'+1}, \ldots \mathbf{F}b_l\}$. Since propagation was performed exhaustively at v_m, it follows $\mathbf{T}\beta(r,\sigma) \in assn(v_m)$. Because of a nogood of the form $\delta' = \{\underline{\mathbf{F}a}, \mathbf{T}\beta(r,\sigma)\} \in ng(r\sigma)$ it then also follows $\mathbf{T}a \in assn(v_m)$. Since $a \notin X_k$, it follows $\mathbf{T}a \notin assn(v_m)$. This is a contradiction.

It thus follows $X_k = T_\mathcal{P}(X_k)$. Further by condition (1) it follows $X_k = \bigcup_{0 \leq i \leq k} X_i$. Then the sequence converges and it holds $X_k = \bigcup_{0 \leq i \leq k} X_i = T_\mathcal{P}(X_k)$.

(4) Consider an arbitrary interpretation X_j, with $j \geq 1$, in the sequence $seq(v_m)$. Then by definition of $seq(v_m)$, there exists some (non-failed) node v_i with $X_j = \{a \mid \mathbf{T}a \in assn(v_i)\}$. Further consider an arbitrary non-auxiliary atom $a \in X_j \setminus X_{j-1}$ that was newly added in this interpretation. Note that it then holds $a \in assn(v_i) \setminus assn(par(v_i))$. Since a is non-auxiliary, by Lemma 6 it follows for every $v_d \in dsc(par(v_i))$ that:

$$\exists r\sigma \in grd(\mathcal{P}) : \Big(a \in H(r\sigma) \ \wedge \ \forall b \in B^+(r\sigma) : \ \mathbf{T}b \in assn(v_d) \\ \wedge \ \forall b' \in B^-(r\sigma) : \ \mathbf{T}b' \notin assn(v_d)\Big).$$

Notably this includes all nodes corresponding to interpretations $X_{j-1}, \dots X_k$ of the sequence. Thus by definition of $seq(x)$ and the choice of X_j, it follows:

$$\forall 1 \leq j \leq k \ \forall a \in X_j \setminus X_{j-1} \ \exists r_a \in \mathrm{grd}(\mathcal{P}) : \\ a \in H(r_a) \ \wedge \ \forall j' \in \{j-1, \dots, k\} : \Big(B^+(r_a) \subseteq X_{j'} \wedge B^-(r_a) \cap X_{j'} = \emptyset\Big).$$

Thus since for every answer set A added to $\mathcal{AS}$ by *AlphaRebootASP*, there exists a computation sequence $(X_0, \dots, X_k)$ where $X_k = A$. It follows from Lemma 1 that $AlphaRebootASP(\mathcal{P}) \subseteq AS(\mathcal{P})$. □

3.4 Completeness

The theoretical proof of completeness and its auxiliary lemmas use the assumption that the *AlphaRebootASP* algorithm terminates when run on some input program $\mathcal{P}$, i.e., *AlphaRebootASP* terminates after some $n \geq 0$ steps. The computation tree T^n is defined as the resulting computation tree for the whole n-iteration run of *AlphaRebootASP* on $\mathcal{P}$.

To prove completeness of the algorithm, an invariant is used. It intuitively states, that at no inner node of the computation tree an arbitrary answer set is prevented from being found. To show this invariant, the possible complete assignments that correspond to an answer set are considered. More formally, for some program $\mathcal{P}$ over language $\mathcal{L}$ the set of *complete assignments* $\mathrm{allASSIGN}^3_{\mathcal{P},\mathcal{L}}$ is defined as:

$$\mathrm{allASSIGN}^3_{\mathcal{P},\mathcal{L}} = \{X \in \mathrm{ASSIGN}^3_{\mathcal{L}} \mid atoms(X) = \mathrm{ATOMS}_{\mathcal{L}} \cup \mathrm{ATOMS}^{\mathrm{aux}}_{\mathcal{P}}\}.$$

This set is then restricted to only assignments that assign exactly the non-auxiliary atoms from some answer set to `true` and other non-auxiliary atoms to `false`, while not violating any nogoods in $\mathrm{ngREP}_{\mathcal{P}}$. Formally, for some answer set A of program $\mathcal{P}$ the set of *A-compatible assignments* is defined as:

$$\mathrm{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}} = \Big\{X \in \mathrm{allASSIGN}^3_{\mathcal{P},\mathcal{L}} \ \Big| \ \{a \in \mathrm{ATOMS}_{\mathcal{L}} \mid \mathbf{T}a \in X\} = A, \\ \{a \mid \mathbf{M}a \in X, \mathbf{T}a \notin X\} = \emptyset, \\ \{\delta \in \mathrm{ngREP}_{\mathcal{P}} \mid \delta \subseteq X\} = \emptyset\Big\}.$$

Lemma 8. *Given an answer set A for program $\mathcal{P}$, there exists some A-compatible assignment $X \in \text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$.*

Proof. An A-compatible assignment X can be constructed as follows (consider rules to be of the form in Equation 2.1):

$$\begin{aligned} X = \{\mathbf{T}a \mid a \in A\} \cup \{\mathbf{M}a \mid a \in A\} \cup \{\mathbf{F}a \mid a \in \text{ATOMS}_\mathcal{L} \setminus A\} \\ \cup \{\mathbf{T}\beta(r,\sigma) \mid r\sigma \in \text{grd}(\mathcal{P}),\ B^+(r\sigma) \subseteq A,\ B^-(r\sigma) \cap A = \emptyset\} \\ \cup \{\mathbf{F}\beta(r,\sigma) \mid r\sigma \in \text{grd}(\mathcal{P}),\ B^+(r\sigma) \not\subseteq A \text{ or } B^-(r\sigma) \cap A \neq \emptyset\} \\ \cup \{\mathbf{T}cOn(r,\sigma) \mid r\sigma \in \text{grd}(\mathcal{P}),\ B^+(r\sigma) \subseteq A\} \\ \cup \{\mathbf{F}cOn(r,\sigma) \mid r\sigma \in \text{grd}(\mathcal{P}),\ B^+(r\sigma) \not\subseteq A\} \\ \cup \{\mathbf{T}cOff(r,\sigma) \mid r\sigma \in \text{grd}(\mathcal{P}),\ B^-(r\sigma) \cap A \neq \emptyset\} \\ \cup \{\mathbf{F}cOff(r,\sigma) \mid r\sigma \in \text{grd}(\mathcal{P}),\ B^-(r\sigma) \cap A = \emptyset\}. \end{aligned}$$

Note that for each auxiliary atom a' either $\mathbf{T}a' \in X$ or $\mathbf{F}a' \in X$ (but not both) holds. Furthermore, note that for each non-auxiliary atom a either $\{\mathbf{T}a, \mathbf{M}a\} \subseteq X$ or $\mathbf{F}a \in X$ (but not both) holds. Thus $X \in \text{allASSIGN}^3_{\mathcal{P},\mathcal{L}}$ holds. To show that $X \in \text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$ holds, consider the conditions for an A-compatible assignment:

- Consider that only the first line in the definition of X adds non-auxiliary atoms to X. It then holds:

$$\{a \in \text{ATOMS}_\mathcal{L} \mid \mathbf{T}a \in X\} = A \quad \text{and} \quad \{a \mid \mathbf{M}a \in X,\ \mathbf{T}a \notin X\} = \emptyset.$$

- To show that $\{\delta \in \text{ngREP}_\mathcal{P} \mid \delta \subseteq X\} = \emptyset$ holds, further consider that for each ground rule $r\sigma \in \text{grd}(\mathcal{P})$ it holds $ng(r\sigma) = ng_{re}(r\sigma) \cup ng_{ch}(r\sigma)$. Then consider these two sets separately:
  - For $\delta \in ng_{re}(r\sigma)$ recall:

$$\begin{aligned} ng_{re}(r\sigma) = \{\{\underline{\mathbf{F}\beta(r,\sigma)}, \mathbf{T}b_1, \ldots, \mathbf{T}b_k, \mathbf{F}b_{k+1}, \ldots, \mathbf{F}b_n\}, \{\underline{\mathbf{F}h}, \mathbf{T}\beta(r,\sigma)\}, \\ \{\mathbf{T}\beta(r,\sigma), \mathbf{F}b_1\}, \ldots, \{\mathbf{T}\beta(r,\sigma), \mathbf{F}b_k\}, \\ \{\mathbf{T}\beta(r,\sigma), \mathbf{T}b_{k+1}\}, \ldots, \{\mathbf{T}\beta(r,\sigma), \mathbf{T}b_n\}\}. \end{aligned}$$

    Then check that $\delta \not\subseteq X$ for each nogood separately:
    - For a nogood of the form $\delta = \{\underline{\mathbf{F}\beta(r,\sigma)}, \mathbf{T}b_1, \ldots, \mathbf{T}b_k, \mathbf{F}b_{k+1}, \ldots, \mathbf{F}b_n\}$, if $\mathbf{F}\beta(r,\sigma) \in X$, then by definition of X it holds $B^+(r\sigma) \not\subseteq A$ or $B^-(r\sigma) \cap A \neq \emptyset$. Thus either $\mathbf{T}b_i \notin X$ holds for some i with $1 \leq i \leq k$ or $\mathbf{F}b_j \notin X$ holds for some j with $k+1 \leq j \leq n$ and it follows $\delta \not\subseteq X$.
    - For a nogood of the form $\delta = \{\underline{\mathbf{F}h}, \mathbf{T}\beta(r,\sigma)\}$, if $\mathbf{T}\beta(r,\sigma) \in X$, then by definition of X it holds $B^+(r\sigma) \subseteq A$ and $B^-(r\sigma) \cap A = \emptyset$. Thus further by definition of an answer set it follows that $H(r\sigma) \neq \emptyset$ and $H(r\sigma) \subseteq A$. Then it follows $\mathbf{F}h \notin X$ by definition of X and further $\delta \not\subseteq X$.

* For a nogood of the form $\{\mathbf{T}\beta(r,\sigma), \mathbf{F}b_i\}$ with $1 \leq i \leq k$, if $\mathbf{T}\beta(r,\sigma) \in X$, then by definition of X it holds $B^+(r\sigma) \subseteq A$. Thus $\mathbf{F}b_i \notin X$ holds and it follows $\delta \not\subseteq X$.
* For a nogood of the form $\{\mathbf{T}\beta(r,\sigma), \mathbf{T}b_i\}$ with $k+1 \leq i \leq n$, if $\mathbf{T}\beta(r,\sigma) \in X$, then by definition of X it holds $B^-(r\sigma) \cap A = \emptyset$. Thus $\mathbf{T}b_i \notin X$ holds and it follows $\delta \not\subseteq X$.

– For $\delta \in ng_{ch}(r\sigma)$ recall:

$$ng_{ch}(r\sigma) = \{\{\underline{\mathbf{F}\,cOn(r,\sigma)}, \mathbf{T}b_1, \ldots, \mathbf{T}b_k\},$$
$$\{\mathbf{F}\,cOff(r,\sigma), \mathbf{T}b_{k+1}\}, \ldots, \{\mathbf{F}\,cOff(r,\sigma), \mathbf{T}b_n\}\}.$$

Then check that $\delta \not\subseteq X$ for each nogood separately:

* For a nogood of the form $\delta = \{\underline{\mathbf{F}\,cOn(r,\sigma)}, \mathbf{T}b_1, \ldots, \mathbf{T}b_k\}$, if $\mathbf{F}\,cOn(r,\sigma) \in X$, then by definition of X it holds $B^+(r\sigma) \not\subseteq A$. Thus $\mathbf{T}b_i \notin X$ holds for some $1 \leq i \leq k$ and it follows $\delta \not\subseteq X$.
* For a nogood of the form $\delta = \{\mathbf{F}\,cOff(r,\sigma), \mathbf{T}b_i\}$ with $k+1 \leq i \leq n$, if $\mathbf{F}\,cOff(r,\sigma) \in X$, then by definition of X it holds $B^-(r\sigma) \cap A = \emptyset$. Thus $\mathbf{T}b_i \notin X$ holds and it follows $\delta \not\subseteq X$.

□

Note that the definition of A-compatible assignments is based only on the nogood representation of the program. The algorithm works with static nogoods in $\text{ngREP}_\mathcal{P}$ as well as with nogoods learned by resolution from $\text{ngREP}_\mathcal{P}$. Learned nogoods are not violated under an A-compatible assignment. This property is described formally by Lemma 9.

Lemma 9. *Given an answer set A for program $\mathcal{P}$ and an A-compatible assignment $X \in \text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$, then it holds for every nogood δ learned by resolution from the set $\text{ngREP}_\mathcal{P}$ that $\delta \not\subseteq X$.*

Proof. To show that $\delta \notin X$ by induction, consider n as the minimum number of resolution steps needed to derive a nogood δ from $\text{ngREP}_\mathcal{P}$. For the base case $n = 0$ consider a nogood $\delta \in \text{ngREP}_\mathcal{P}$. By definition of an A-compatible assignment it holds $\delta \not\subseteq X$. As the induction hypothesis consider the property that $\delta \not\subseteq X$ holds for all nogoods δ that can be derived by resolution from $\text{ngREP}_\mathcal{P}$ in $n-1$ steps.

In the induction step assume, towards a contradiction, that $\delta \subseteq X$ holds for some nogood δ that can be derived in n steps by resolution from $\text{ngREP}_\mathcal{P}$. Then there exist two nogoods δ_1 and δ_2 and some atom a s.t. $\delta = (\delta_1 \setminus \{\mathbf{T}a\}) \cup (\delta_2 \setminus \{\mathbf{F}a\})$ and $\mathbf{T}a \in \delta_1$ as well as $\mathbf{F}a \in \delta_2$. Note that the nogoods δ_1, δ_2 can be derived by resolution from $\text{ngREP}_\mathcal{P}$ in at most $n-1$ steps. Then since $X \in \text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$, it follows that either $\mathbf{T}a \in X$ or $\mathbf{F}a \in X$ by definition of an A-compatible assignment. There are two possible cases:

- If $\mathbf{T}a \in X$, then $\delta_1 \subseteq X$ follows. By induction hypothesis $\delta_1 \not\subseteq X$ holds, since δ_1 can be derived by resolution from $\text{ngREP}_\mathcal{P}$ in $n-1$ steps. This is a contradiction.
- If $\mathbf{F}a \in X$, then $\delta_2 \subseteq X$ follows. By induction hypothesis $\delta_2 \not\subseteq X$ holds, since δ_2 can be derived by resolution from $\text{ngREP}_\mathcal{P}$ in $n-1$ steps. This is a contradiction.

Thus $\delta \not\subseteq X$ holds. □

Consider any nogood δ obtained during the *AlphaRebootASP* algorithm and an A-compatible assignment X for some answer set A of a program $\mathcal{P}$. Then δ is not violated under X or A is found by *AlphaRebootASP* (or both). This is captured by Lemma 10.

Lemma 10. *Given a computation tree $T^n = (V^n, E^n)$ for a program $\mathcal{P}$ and an answer set A for $\mathcal{P}$, an A-compatible assignment $X \in \text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$ and a nogood δ where $\delta \in \Delta_S \cup \Delta_L$ holds at some $v \in V^n$, then at least one of the following holds: there exists a propagation node $v_z \in V^n$ after exhaustive propagation with $\mathit{iterCase}(v_z) =$ `(answer)` after which A is found, or $\delta \not\subseteq X$ holds.*

Proof. The *AlphaRebootASP* algorithm stores only the following types of nogoods: static nogoods from the nogoods representation $\text{ngREP}_\mathcal{P}$, nogoods learned by resolution from $\text{ngREP}_\mathcal{P}$ and enumeration nogoods obtained when an answer set is found. Consider these two possible cases for some nogood $\delta \in \Delta_S \cup \Delta_L$:

- If $\delta \in \text{ngREP}_\mathcal{P}$ is a static nogood or a nogood learned by resolution from $\text{ngREP}_\mathcal{P}$, then by Lemma 9 it follows $\delta \not\subseteq X$ and the lemma holds.
- If δ is an enumeration nogood obtained when finding an answer set after some propagation node v_i with $\mathit{iterCase}(v_i) =$ `(answer)`, then consider which answer set is found after v_i. If A is the answer set found after v_i, then the lemma holds.

  Thus consider the case where some other $A' \neq A$ is the answer set found after v_i, where $A' = \{a \in \text{ATOMS}_\mathcal{L} \mid \mathbf{T}a \in \mathit{assn}(v_i)\}$. The constructed enumeration nogood δ (see Section 2.3.4) is of the form $\delta = \{\mathbf{T}\beta(r,\sigma) \in \mathit{assn}(v_i)\}$.

  Assume, towards a contradiction, that $\delta \subseteq X$. Next, a set A^*, with the heads of all rules considered to have fired at v_i, is constructed from the positively assigned rule atoms at v_i. The set A^* is then used to arrive at a contradiction by showing $A' = A^*$ and further $A' \subset A$.

  Consider an arbitrary literal $\mathbf{T}\beta(r',\sigma') \in \mathit{assn}(v_i)$ with $h \in H(r'\sigma')$. By Lemma 3 there exists a unique node v' on the path from the root v_0 to v_i where $\mathbf{T}\beta(r',\sigma') \in \mathit{assn}(v') \setminus \mathit{assn}(\mathit{par}(v'))$. Since $v_i \in \mathit{dsc}_{nf}(v')$ it follows by Lemma 5:

  $$\forall b \in B^+(r'\sigma') : \mathbf{T}b \in \mathit{assn}(v_i) \quad \text{and} \quad \forall b' \in B^-(r'\sigma') : \mathbf{T}b' \notin \mathit{assn}(v_i).$$

  Then it follows by Lemma 7 that $ng(r'\sigma') \subseteq \Delta_S$ holds at v_i. This means that $r'\sigma'$ is part of the synchronized ground rules at v_i. Consider the nogood $\delta' \in ng(r'\sigma')$

of the form $\delta' = \{\underline{\mathbf{F}h}, \mathbf{T}\beta(r', \sigma')\}$. Since v_i is a propagation node after exhaustive propagation and $\mathbf{T}\beta(r', \sigma') \in assn(v_i)$, it follows that $\mathbf{T}h \in assn(v_i)$.

By the choice of $\mathbf{T}\beta(r', \sigma')$, it follows:

$$\{\mathbf{T}h \mid h \in H(r\sigma),\ \mathbf{T}\beta(r, \sigma) \in assn(v_i)\} \subseteq assn(v_i).$$

Further let $A^* = \{h \mid h \in H(r\sigma),\ \mathbf{T}\beta(r, \sigma) \in \delta\}$. Then, since δ is the enumeration nogood for the answer set A' found at v_i, it holds:

$$A^* = \{h \mid h \in H(r\sigma),\ \mathbf{T}\beta(r, \sigma) \in assn(v_i)\} \subseteq A'.$$

To show that $A' \subseteq A^*$, consider an arbitrary atom $a \in A'$ and recall that A' is an answer set. Let $(Y_0, Y_1, \ldots, Y_\infty)$ be a computation sequence with $Y_\infty = A'$. Then by property (4) of a computation sequence, there exists some ground rule $r_a\sigma \in \text{grd}(\mathcal{P})$ such that:

$$a \in H(r_a\sigma), \quad B^+(r_a\sigma) \subseteq A', \quad B^-(r_a\sigma) \cap A' = \emptyset.$$

It follows by Lemma 7 that $ng(r_a\sigma) \subseteq \Delta_S$ holds at v_i. Consider that there exists some nogood $\delta^* \in \text{ngREP}_\mathcal{P}$ where $r_a\sigma$ is of the form shown in Equation 2.1 and δ^* is of the form:

$$\delta^* = \{\underline{\mathbf{F}\beta(r_a, \sigma)}, \mathbf{T}b_1, \ldots, \mathbf{T}b_k, \mathbf{F}b_{k+1}, \ldots, \mathbf{F}b_n\}.$$

Then since propagation was performed exhaustively at v_i, it follows $\mathbf{T}\beta(r_a, \sigma) \in assn(v_i)$. Then by definition of A^* it holds $a \in A^*$. By the choice of a it follows $A' \subseteq A^*$. Further since $A^* \subseteq A'$, it follows $A' = A^*$.

Recall that $\delta \subseteq X$ holds by assumption and no nogood of the form $\{\underline{\mathbf{F}h}, \mathbf{T}\beta(r, \sigma)\} \in \text{ngREP}_\mathcal{P}$ with $\mathbf{T}\beta(r, \sigma) \in \delta$ is violated under X. Then since X is A-compatible and $A^* \subseteq \text{ATOMS}_\mathcal{L}$, it follows $A' = A^* \subseteq \{a \mid \mathbf{T}a \in X\} = A$. Since $A' \neq A$ it thus holds $A' \subset A$. This is a contradiction to A being a subset minimal model of $GL_A(\mathcal{P})$. It follows $\delta \not\subseteq X$ and the lemma holds.

□

Furthermore, a *partial computation sequence* is defined as a finite sequence (as opposed to an infinite computation sequence) with the persistence of beliefs and revision properties of a computation sequence. More formally, a *partial computation sequence* is defined as a finite sequence $(X_0, \ldots, X_k)$ of Herbrand interpretations over some program $\mathcal{P}$ with $X_0 = \emptyset$ and satisfying the following properties:

(1) Persistence of beliefs:

$$\forall i \in \{1, \ldots, k\} :\ X_{i-1} \subseteq X_i$$

(2) Revision:

$$\forall 1 \in \{1, \ldots, k\} : \; X_i \subseteq T_{\mathcal{P}}(X_{i-1})$$

The notion of an A-compatible node intuitively describes that an answer set A of program $\mathcal{P}$ can still be found from that node. Formally a node v_i is an *A-compatible node* if it satisfies the following properties:

(I) There exists a partial computation sequence $\mathit{nodeSeq}(v_i) = (X_0, \ldots, X_k)$ where $X_0 = \emptyset$ and $X_k = \mathit{inter}(v_i)$.

(II) The assignment $\mathit{assn}(v_i)$ is a subset of some A-compatible assignment, i.e.:

$$\exists X^*_A \in \text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}} : \; \mathit{assn}(v_i) \subseteq X^*_A.$$

(III) For every atom a added to a set in the partial computation sequence $\mathit{nodeSeq}(v_i)$, there exists some rule that has a in its head. Formally, it holds:

$$\forall j \in \{1, \ldots, k\} : \Big(X_{j-1} \subset X_j \Rightarrow \exists r\sigma \in \text{grd}(\mathcal{P}) : \; (X_j \setminus X_{j-1} = H(r\sigma) \\ \wedge \; \mathbf{T}\beta(r, \sigma) \in \mathit{assn}(v_i))\Big).$$

Given an an A-compatible propagation node for some answer set A, then the node is non-failed or A is found by *AlphaRebootASP* (or both). This is described by Lemma 11.

Lemma 11. *Given a computation tree $T^n = (V^n, E^n)$ for a program $\mathcal{P}$, an answer set A for $\mathcal{P}$ and an A-compatible propagation node $v_i \in V^n$, then at least one of the following holds: there exists a propagation node $v_z \in V^n$ after exhaustive propagation with $\mathit{iterCase}(v_z) =$ `(answer)` after which A is found, or v_i is a non-failed node.*

Proof. Consider the unique node $v_e \in \mathit{dsc}(v_i)$ that represents the first computation point after exhaustive propagation starting from v_i and traversing T^n away from the root. Note that there is a unique (possibly empty) path from v_i to v_e in T^n. Since v_i is A-compatible, there exists some A-compatible assignment X^*_A s.t. $\mathit{assn}(v_i) \subseteq X^*_A$.

Consider the literal l added by the first propagation step from v_i up to v_e. Let $\delta \in \Delta_S \cup \Delta_L$ be the nogood that triggered the propagation of l. Then by Lemma 10 one of the following holds: there exists a propagation node v_z after exhaustive propagation with $\mathit{iterCase}(v_z) =$ `(answer)` after which A is found, or $\delta \not\subseteq X^*_A$ holds. In the first case the lemma holds. Thus consider the case where $\delta \not\subseteq X^*_A$ holds. Thus, by definition of an A-compatible assignment, it follows $l \in X^*_A$.

This argument can be applied for all literals added by propagation steps from v_i up to v_e (in order of propagation) and it follows that $\mathit{assn}(v_e) \subseteq X^*_A$. Then, again by Lemma 10, it follows for arbitrary $\delta' \in \Delta_S \cup \Delta_L$ at v_e that either A is found after some propagation node v_z or $\delta' \not\subseteq \mathit{assn}(v_e)$ holds. If the first case holds for any such nogood, then the lemma holds. If it holds for none of the nogoods, then $\mathit{iterCase}(v_i)$, which is the same as $\mathit{iterCase}(v_e)$, is not `(conflict)` and the lemma also holds. □

At any point in the algorithm immediately before iteration case `(close)` or `(backtrack)` it holds that there cannot be an applicable rule left under the currently considered set of atoms. This is captured by Lemma 12.

Lemma 12. *Given a computation tree $T^n = (V^n, E^n)$ for a program $\mathcal{P}$, an answer set A for $\mathcal{P}$ and a propagation node $v_i \in V^n$ with $X = inter(v_i)$, where v_i represents a computation point after exhaustive propagation, and $iterCase(v_i) \in \{$ `(close)`, `(backtrack)` $\}$, then it holds:*

$$\nexists r\sigma \in \mathrm{grd}(\mathcal{P}) : \exists a \in \mathrm{ATOMS}_{\mathcal{L}} \setminus X : \Big(a \in H(r\sigma) \wedge B^+(r\sigma) \subseteq X \wedge B^-(r\sigma) \cap X = \emptyset\Big).$$

Proof. Assume, towards a contradiction, that there exists some ground rule $r\sigma \in \mathrm{grd}(\mathcal{P})$ of the form in Equation 2.1 and some atom $a \in \mathrm{ATOMS}_{\mathcal{L}} \setminus X$ s.t.:

$$a \in H(r\sigma) \ \wedge \ B^+(r\sigma) \subseteq X \ \wedge \ B^-(r\sigma) \cap X = \emptyset.$$

Note that $r\sigma$ is of interest w.r.t. $assn(v_i)$. Consider an arbitrary atom $p(t_1, \ldots, t_z) \in B^+(r\sigma)$. Note that $B^+(r\sigma) \subseteq X$ holds and thus $\mathbf{T}p(t_1, \ldots, t_z) \in assn(v_i)$ follows. Note that v_i is non-failed since $iterCase(v_i) \neq$ `(conflict)`. Then by Lemma 3 there exists a unique node v' on the path from the root v_0 to v_i where $\mathbf{T}p(t_1, \ldots, t_z) \in assn(v') \setminus assn(par(v'))$. It follows by Lemma 6, that there exists some rule $r'\sigma' \in \mathrm{grd}(\mathcal{P})$ with $p(t_1, \ldots, t_z) \in H(r'\sigma')$. By the choice of $p(t_1, \ldots, t_z)$, this follows for every atom $a \in B^+(r\sigma)$. Thus $r\sigma$ does not satisfy the first case in the definition of an inactive rule.

Consider an arbitrary fact $r_f \in \mathcal{P}$ in the program $\mathcal{P}$. Since grounding was applied exhaustively, it follows that r_f was already returned by grounding. Also r_f would have been returned by a reboot at any point since it is always of interest and never inactive. Thus $ng(r_f) \in \Delta_S$ holds at v_i. Since $iterCase(v_i) \neq$ `(conflict)`, no nogood of the form $\delta = \{\underline{\mathbf{F}a}\}$, with $a \in H(r_f)$, is violated under $assn(v_i)$. By the choice of r_f, it follows that $B^-(r\sigma)$ does not contain an atom a s.t. there exists a fact r_f with $a \in H(r_f)$. Thus $r\sigma$ is not inactive.

The assignment did not change since the most recent grounding step since $iterCase(v_i) \neq$ `(ground)`. Thus $r\sigma$ was part of the synchronized ground rules after the most recent grounding step. A reboot at any point since then would have also returned $r\sigma$, since $r\sigma$ would have still been of interest and not inactive. Thus $r\sigma$ is still part of the synchronized ground rules at v_i.

Then consider that propagation was applied exhaustively at v_i. Because there exists a nogood $\delta_1 \in ng(r\sigma)$ of the form $\delta_1 = \{\underline{\mathbf{F}cOn(r, \sigma)}, \mathbf{T}b_1, \ldots, \mathbf{T}b_k\}$ and v_i is non-failed, it follows that $\mathbf{T}cOn(r, \sigma) \in assn(v_i)$.

Next, note that the assignment $\mathbf{T}cOff(r, \sigma)$ can only be added by strong propagation on some nogood $\delta_2 \in ng(r\sigma)$ of the form $\delta_2 = \{\mathbf{F}cOff(r, \sigma), \mathbf{T}b_j\}$, where $b_j \in B^-(r\sigma)$.

Then consider an arbitrary atom $b_j \in B^-(r\sigma)$. Further consider an arbitrary node $\hat{v}$ on the path from the root v_0 to v_i. Since $\mathbf{T}b_j \notin \mathit{assn}(v_i)$ it follows by Lemma 3 that $\mathbf{T}b_j \notin \mathit{assn}(\hat{v})$. Thus $\mathbf{T}\mathit{cOff}(r,\sigma)$ cannot be added at node $\hat{v}$ since $\mathbf{T}b_j \notin \mathit{assn}(v_i)$. By the choice of $\hat{v}$, it follows that $\mathbf{T}\mathit{cOff}(r,\sigma) \notin \mathit{assn}(v_i)$.

Then $\beta(r,\sigma)$ is an element of the active choice points at v_i. Further v_i does not represent a computation point before iteration case `(close)` or `(backtrack)` since the active choice points are not empty. This is a contradiction. □

Based on the notion of A-compatible nodes, Lemma 13 states that the intermediate steps up to some A-compatible node can be continued up to the answer set A, resulting in a computation sequence for A.

Lemma 13. *Given an answer set A for program $\mathcal{P}$ and an A-compatible computation tree node $v_i \in V^n$ with partial computation sequence $\mathit{nodeSeq}(v_i) = (X_0, \ldots, X_k)$, then there exists a computation sequence $(X_0, \ldots, X_k, X_{k+1}, \ldots, X_n)$ s.t. $X_n = A$.*

Proof. Since A is an answer set, it follows by Lemma 1, that there exists some computation sequence $S = (Y_0, \ldots, Y_m)$ where $Y_m = A$ and $Y_0 = \emptyset$. A computation sequence $\hat{S} = (X_0, \ldots, X_k, X_{k+1}, \ldots, X_n)$, where $\mathit{nodeSeq}(v_i) = (X_0, \ldots, X_k)$ and $n = k + m$, can be constructed inductively as follows:

$$\forall j \in \{1, \ldots, m\} : \; X_{k+j} = X_{k+j-1} \cup Y_j.$$

Note that $X_k = \mathit{inter}(v_i)$ holds. Since v_i is A-compatible, it follows:

$$\exists X_A^* \in \mathrm{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}} : \; \mathit{assn}(v_i) \subseteq X_A^*.$$

Thus it follows $X_k \subseteq A$ by definition of $\mathrm{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$. Further it holds:

$$\forall j \in \{0, \ldots, m\} : \; Y_j \subseteq Y_m \quad \text{and}$$
$$\forall j \in \{0, \ldots, k\} : \; X_j \subseteq X_k.$$

Thus it follows $X_n \subseteq Y_m$ by construction of $\hat{S}$ since $Y_m = A$. It also holds $Y_m \subseteq X_n$ by construction of $\hat{S}$ and thus it follows $X_n = Y_m = A$.

Then $\hat{S}$ is a computation sequence, which can be verified by checking the four properties as follows (note that property (2) is intentionally listed last):

(1) Since persistence of beliefs holds for the partial computation sequence $\mathit{nodeSeq}(v_i)$, persistence of beliefs follows for $\hat{S}$ by the inductive definition of $\hat{S}$.

(3) Since convergence holds for $(Y_0, \ldots, Y_m)$ and $Y_m = X_n$, convergence follows for $\hat{S}$.

(4) Consider X_t as an arbitrary interpretation with $1 \leq t \leq n$ in the sequence $\hat{S}$. Further consider an arbitrary non-auxiliary $a \in X_t \setminus X_{t-1}$ that was newly added in this interpretation. There are two possible cases:

- If $1 \leq t \leq k$, then by property (III) of the A-compatible node v_i it holds:

$$\exists r\sigma \in \text{grd}(\mathcal{P}) : \Big(X_t \setminus X_{t-1} = H(r\sigma) \ \wedge \ \mathbf{T}\beta(r,\sigma) \in assn(v_i)\Big).$$

  By property (II) of the A-compatible node v_i it holds:

$$\exists X_A^* \in \text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}} : \ assn(v_i) \subseteq X_A^*.$$

  It follows that $\mathbf{T}\beta(r,\sigma) \in X_A^*$. Further by definition of $\text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$ it follows $B^-(r\sigma) \cap A = \emptyset$. Thus by revision for $nodeSeq(v_i)$ and persistence of beliefs for $\hat{S}$, it follows:

$$a \in H(r\sigma) \ \wedge \ \forall j \geq t-1 : \Big(B^+(r\sigma) \subseteq X_j \ \wedge \ B^-(r\sigma) \cap X_j = \emptyset\Big).$$

- If $k+1 \leq t \leq n$, then by persistence of reasons for S, there exists some rule $r\sigma \in \text{grd}(\mathcal{P})$ s.t.:

$$a \in H(r\sigma) \ \wedge \ B^+(r\sigma) \subseteq X_{t-1} \ \wedge \ B^-(r\sigma) \cap A = \emptyset.$$

  By persistence of beliefs for $\hat{S}$, it then follows:

$$a \in H(r\sigma) \ \wedge \ \forall j \geq t-1 : \Big(B^+(r\sigma) \subseteq X_j \ \wedge \ B^-(r\sigma) \cap X_j = \emptyset\Big).$$

  By the choice of X_t and a, persistence of reasons follows for $\hat{S}$.

(2) The revision property for $\hat{S}$ follows from persistence of beliefs for $\hat{S}$ by the following argument. An arbitrary atom $a \in X_j$ in some arbitrary element X_j of the sequence $\hat{S}$ with $j \geq 1$ has some earliest X_a where $a \in X_a \setminus X_{a-1}$. Then by persistence of beliefs for $\hat{S}$ it holds $a \in T_{\mathcal{P}}(X_{j-1})$. Thus by the choice of a and X_j the revision property for $\hat{S}$ follows.

□

Given some answer set A of a program $\mathcal{P}$, when traversing the computation tree downwards from the root, an A-compatible node, that is not a leaf, has a child node that is also A-compatible. Lemma 14 describes this property formally.

Lemma 14. *Given an answer set A for program $\mathcal{P}$ and an A-compatible computation tree node $v_i \in V^n$ with $chl(v_i) \neq \emptyset$, then at least one of the following holds: there exists a propagation node $v_z \in V^n$ after exhaustive propagation with $iterCase(v_z) =$ `(answer)` after which A is found, or there exists an A-compatible child node $v_c \in chl(v_i)$.*

Proof. Consider an arbitrary A-compatible node $v_i \in V^n$ with $chl(v_i) \neq \emptyset$. Then consider the child node $v_c \in chl(v_i)$ where the index c from the computation tree construction (see Section 3.2) is minimal. This represents the first child node reached by the algorithm. There are five possible cases for the node v_c:

- If v_c is a propagation node, then consider the assignment $assn(v_i)$ and the literal $l \in assn(v_c) \setminus assn(v_i)$ added by the propagation step. By definition of propagation (see Sections 2.3.2 and 2.3.4) there exists some nogood $\delta \in \Delta_S \cup \Delta_L$ at v_i, that caused the propagation step at v_i.

  Consider that there exists some A-compatible assignment $X^*_A \in \text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$ with $assn(v_i) \subseteq X^*_A$ since v_i is an A-compatible node. Then by Lemma 10 at least one of the following holds: there exists a propagation node v_z after exhaustive propagation with $iterCase(v_z) = \texttt{(answer)}$ after which A is found, or $\delta \not\subseteq X$ holds. In the first case the lemma holds. Thus assume that $\delta \not\subseteq X$ holds. Then since $\delta \setminus \{\bar{l}\} \subseteq assn(v_i) \subseteq X^*_A$, it follows that $l \in X^*_A$. Thus it further follows $assn(v_c) = assn(v_i) \cup \{l\} \subseteq X^*_A$. Then property (II) of an A-compatible node holds for v_c.

  Further there are three possible cases for the literal $l \in assn(v_c) \setminus assn(v_i)$ added by the propagation step:

  - If the atom in l is auxiliary, then $inter(v_i) = inter(v_c)$ holds and thus there exists a partial computation sequence $nodeSeq(v_c) = nodeSeq(v_i)$.
  - If $l = \mathbf{F}a$ or $l = \mathbf{M}a$ and a is non-auxiliary, then again $inter(v_i) = inter(v_c)$ holds and thus there exists a partial computation sequence $nodeSeq(v_c) = nodeSeq(v_i)$.
  - If $l = \mathbf{T}a$, and a is non-auxiliary, then consider that there exists a partial computation sequence $nodeSeq(v_c) = (X_0, \ldots, X_k, X_{k+1})$ where $(X_0, \ldots, X_k) = nodeSeq(v_i)$ and $X_{k+1} = X_k \cup \{a\}$. Consider the properties of a partial computation sequence. Note that property (1) holds for $nodeSeq(v_i)$. Thus property (1) immediately follows for $nodeSeq(v_c)$.

    To show that property (2) also holds for $nodeSeq(v_c)$, consider an arbitrary atom $a' \in X_{k+1}$. Note that $\mathbf{T}a' \in assn(v_c)$ holds, since $X_k = inter(v_i)$ and $assn(v_c) = assn(v_i) \cup \{\mathbf{T}a\}$. Consider the node $v_{a'}$ along the path from the root v_0 to v_c where $a' \in assn(v_{a'}) \setminus assn(par(v_{a'}))$. By Lemma 3 the assignment only grows along this path and it follows that $v_{a'}$ is unique.

    By Lemma 11, one of the following two options holds: there exists a propagation node v_z after exhaustive propagation with $iterCase(v_z) = \texttt{(answer)}$ after which A is found, or v_i is non-failed. In the first case the lemma holds. Thus consider the case where v_i is non-failed. Since it holds $v_i \in dsc_{nf}(v_{a'})$ and a' is non-auxiliary, it follows by Lemma 6 that there exists some ground rule $r\sigma \in grd(\mathcal{P})$ such that:

$$a' \in H(r\sigma) \ \wedge \ \forall b \in B^+(r\sigma) : \ \mathbf{T}b \in assn(v_i) \\ \wedge \ \forall b' \in B^-(r\sigma) : \ \mathbf{T}b' \notin assn(v_i).$$

    It follows $B^+(r\sigma) \subseteq inter(v_i)$ and $B^-(r\sigma) \cap inter(v_i) = \emptyset$. Thus since $X_k = inter(v_i)$, it holds $a' \in T_{\mathcal{P}}(X_k)$.

Then, by the choice of a', it follows $X_{k+1} \subseteq T_{\mathcal{P}}(X_k)$. It follows that property (2) of a partial computation sequence also holds for $nodeSeq(v_c)$. Then property (I) of an A-compatible node holds for v_c since $X_{k+1} = inter(v_c)$.

Next consider that $\mathbf{T}a$ was added by strong propagation on some nogood $\delta = \{\underline{\mathbf{F}a}, \mathbf{T}\beta(r', \sigma')\}$ for some rule $r'\sigma'$ with $a \in H(r'\sigma')$. Thus it also holds $\mathbf{T}\beta(r', \sigma') \in assn(v_i)$ and, by Lemma 3, further $\mathbf{T}\beta(r', \sigma') \in assn(v_c)$. Then consider that $X_{k+1} \setminus X_k = \{a\}$. Thus it holds:

$$X_{k+1} \setminus X_k = H(r'\sigma') \ \wedge \ \mathbf{T}\beta(r', \sigma') \in assn(v_c).$$

Since property (III) of an A-compatible node holds for v_i, it follows that property (III) also holds for v_c and thus v_c is A-compatible.

- If v_c is an iteration node for case `(ground)`, then the current assignment is not modified. This means that $assn(v_i) = assn(v_c)$. Since property (II) of an A-compatible node holds for v_i, it also holds for v_c. Further there exists a partial computation sequence $nodeSeq(v_c) = nodeSeq(v_i)$ and property (I) holds for v_c since it holds for v_i. Then since property (III) holds for v_i and $assn(v_i) = assn(v_c)$ as well as $nodeSeq(v_i) = nodeSeq(v_c)$ hold, it follows that property (III) holds for v_c and thus v_c is an A-compatible node.

- If v_c is an iteration node for case `(reboot)`, then the current assignment is not modified. Thus, by the same argument as for the previous case, it follows that v_c is an A-compatible node.

- If v_c is an iteration node for case `(choice)`, then $assn(v_c) = assn(v_i) \cup \{\mathbf{T}\beta(r, \sigma)\}$ for some rule $r\sigma \in \mathrm{grd}(\mathcal{P})$ where $\beta(r, \sigma) \in acp(\Delta, assn(v_i))$. Consider the A-compatible assignment $X^*_A \in \mathrm{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$ where $assn(v_i) \subseteq X^*_A$. There are two possible cases:

  - If $\mathbf{T}\beta(r, \sigma) \in X^*_A$, then $assn(v_c) \subseteq X^*_A$ and thus property (II) of an A-compatible node holds for v_c. Since $inter(v_i) = inter(v_c)$ holds, there exists the partial computation sequence $nodeSeq(v_c) = nodeSeq(v_i)$ and thus properties (I) and (III) also hold for v_c. Then v_c is A-compatible.
  - If $\mathbf{T}\beta(r, \sigma) \notin X^*_A$, then note that by Lemma 2 all nodes in the subtree rooted at v_c have consecutive indices in the computation tree construction and describe a consecutive sequence of iterations during the execution of the algorithm. Then consider the highest iteration number $iterNum(v_m)$ of a node v_m in this subtree. There are four possible cases for the end of the subtree, i.e., the iteration case $iterCase(v_m)$:
    - If the subtree ends with a backtrack, then another child node $v'_c \in chl(v_i)$ exists with $assn(v'_c) = assn(v_i) \cup \{\mathbf{F}\beta(r, \sigma)\}$. Note that since $\mathbf{T}\beta(r, \sigma) \notin X^*_A$ it follows $\mathbf{F}\beta(r, \sigma) \in X^*_A$ from the definition of $\mathrm{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$ since $\beta(r, \sigma) \in \mathrm{ATOMS}^{\mathrm{aux}}_{\mathcal{P}}$. Thus it holds $assn(v'_c) \subseteq X^*_A$. Then by the same argument as for the case where $\mathbf{T}\beta(r, \sigma) \in X^*_A$, it follows that v'_c is A-compatible.

  - If the subtree ends with a backjump to decision level $dl(v_i)$, then another child node $v'_c \in chl(v_i)$ exists with $assn(v'_c) = assn(v_i)$. Then by the same argument as for the case where v_c is an iteration node for case `(ground)`, it follows that v'_c is A-compatible.
  - If the subtree ends with a backjump to some decision level lower than $dl(v_i)$, then there exists some nogood δ where $\delta \subseteq assn(v_i)$. Thus it holds:

$$\delta \subseteq assn(v_i) \subseteq X^*_A.$$

    By Lemma 10 at least one of the following two options holds: there exists a propagation node v_z after exhaustive propagation with $iterCase(v_z) =$ `(answer)` after which A is found, or $\delta \not\subseteq X^*_A$ holds. Since $\delta \subseteq X^*_A$ holds, the first option follows and thus the lemma holds.
  - If the subtree ends with the empty nogood $\delta = \emptyset$ being learned or returned from grounding, then $\delta \subseteq X^*_A$ holds. This is a contradiction since, by Lemma 9, no nogood resulting from $\text{ngREP}_\mathcal{P}$ by resolution is violated under X^*_A.

- If v_c is an iteration node for case `(close)`, then consider the A-compatible assignment $X^*_A \in \text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$ where $assn(v_i) \subseteq X^*_A$. There are two possible cases for the set $X_{cls} = \{\mathbf{F}a \mid a \in atoms(\Delta_S \cup \Delta_L) \setminus atoms(A)\}$:
  - If $X_{cls} \subseteq X^*_A$, then it holds:

$$assn(v_c) = assn(v_i) \cup X_{cls} \subseteq X^*_A.$$

    Thus property (II) of an A-compatible node holds for v_c. Further consider that $inter(v_i) = inter(v_c)$ holds. Thus there exists the partial computation sequence $nodeSeq(v_c) = nodeSeq(v_i)$ and properties (I) and (III) also hold for v_c. Then v_c is A-compatible.
  - If $X_{cls} \not\subseteq X^*_A$, then there exists some atom a s.t. $\mathbf{T}a \in X^*_A$ and $\mathbf{T}a \notin assn(v_i)$. Further it then holds $inter(v_i) \subset A$. By Lemma 13 there exists a computation sequence $(X_0, \ldots, X_k, X_{k+1}, \ldots, X_n)$ s.t. $X_n = A$ and $nodeSeq(v_i) = (X_0, \ldots, X_k)$ where $X_k = inter(v_i)$. Then it follows that there exists some smallest j where $k+1 \leq j \leq n$ and $X_{j-1} \subset X_j$. Note that by the choice of j and the persistence of beliefs property of a computation sequence, it follows that $X_{j-1} = X_k$. By the revision property of a computation sequence $X_j \subseteq T_\mathcal{P}(X_{j-1})$ holds. Thus there exist some rule $r'\sigma' \in \text{grd}(\mathcal{P})$ and some atom $a \in X_j \setminus X_{j-1}$ with:

$$a \in H(r'\sigma') \ \wedge \ B^+(r'\sigma') \subseteq X_k \ \wedge \ B^-(r'\sigma') \cap X_k = \emptyset.$$

    By Lemma 12 it holds that:

$$\nexists r\sigma \in \text{grd}(\mathcal{P}) : \ \exists a \in \text{ATOMS}_\mathcal{L} \setminus X_k : \ \Big(a \in H(r\sigma) \ \wedge \ B^+(r\sigma) \subseteq X_k \ \wedge \ B^-(r\sigma) \cap X_k = \emptyset\Big).$$

This is a contradiction.

Recall that an iteration node represents the computation point at the end of an iteration, specifically after backtracking and backjumping in the respective cases. An iteration node for case `(conflict)`, `(answer)` or `(backtrack)` cannot be the first child node since there needs to be another child node with a lower index for a backtrack or backjump to this decision level to be possible. □

Theorem 3 (Completeness)**.** *The algorithm AlphaRebootASP is complete, i.e., for every program $\mathcal{P}$ over some language $\mathcal{L}$, it holds $AlphaRebootASP(\mathcal{P}) \supseteq AS(\mathcal{P})$.*

Proof. Consider an arbitrary answer set $A \in AS(\mathcal{P})$. By Lemma 8 there exists some A-compatible assignment $X_A^* \in \text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$. Then consider the root node $v_0 \in V^n$. There exists the partial computation sequence $nodeSeq(v_0) = (X_0)$ with $X_0 = \emptyset$. Furthermore, it holds $assn(v_0) = \emptyset \subseteq X_A^*$. Thus v_0 is A-compatible.

Since the computation tree is finite, by assumption that the algorithm terminates on program $\mathcal{P}$, it follows by repeated use of Lemma 14, that at least one of the following holds: there exists a propagation node v_z after exhaustive propagation with $iterCase(v_z) =$ `(answer)` after which A is found, or there exists an A-compatible leaf node v_l. If the first option holds, it follows $A \in AlphaRebootASP(\mathcal{P})$. Thus consider the case where the second option holds. The possible iteration cases for this (propagation) leaf node, representing the computation point after exhaustive propagation, are: `(conflict)`, `(answer)`, `(backtrack)`. Consider these three cases:

- If $iterCase(v_l) =$ `(conflict)` holds, then there exists some violated nogood δ with $\delta \subseteq assn(v_l)$. Since v_l is A-compatible, there exists some A-compatible assignment $X_A^* \in \text{compatASSIGN}^3_{A,\mathcal{P},\mathcal{L}}$ s.t. $assn(v_l) \subseteq X_A^*$. Thus it holds $\delta \subseteq assn(v_l) \subseteq X_A^*$.

  By Lemma 10 at least one of the following holds: there exists a propagation node v_z after exhaustive propagation with $iterCase(v_z) =$ `(answer)` after which A is found, or $\delta \not\subseteq X_A^*$. Since $\delta \subseteq X_A^*$ holds, it follows $A \in AlphaRebootASP(\mathcal{P})$.

- If $iterCase(v_l) =$ `(backtrack)` holds, then there exists some atom a s.t. $\mathbf{M}a \in assn(v_l)$ and $\mathbf{T}a \notin assn(v_l)$. Since $assn(v_l) \subseteq X_A^*$ it also holds $\mathbf{T}a \in X_A^*$ by definition of an A-compatible assignment. Further it then holds $inter(v_l) \subset A$. By Lemma 13 there exists a computation sequence $(X_0, \ldots, X_k, X_{k+1}, \ldots, X_n)$ s.t. $X_n = A$ and $nodeSeq(v_l) = (X_0, \ldots, X_k)$ where $X_k = inter(v_l)$. Then it follows that there exists some smallest j where $k+1 \leq j \leq n$ and $X_j \supset X_{j-1}$. Note that $X_{j-1} = X_k$ holds by the choice of j. By the revision property of a computation sequence $X_j \subseteq T_{\mathcal{P}}(X_{j-1})$ holds. Thus there exist some rule $r'\sigma' \in \text{grd}(\mathcal{P})$ and some atom $a \in X_j \setminus X_{j-1}$ with:

$$a \in H(r'\sigma') \ \wedge \ B^+(r'\sigma') \subseteq X_k \ \wedge \ B^-(r'\sigma') \cap X_k = \emptyset.$$

By Lemma 12 it holds that:

$$\nexists r\sigma \in \text{grd}(\mathcal{P}) : \exists a \in \text{ATOMS}_{\mathcal{L}} \setminus X_k : \Big(a \in H(r\sigma) \wedge B^+(r\sigma) \subseteq X_k \wedge B^-(r\sigma) \cap X_k = \emptyset\Big).$$

This is a contradiction.

- If $\mathit{iterCase}(v_l) = \texttt{(answer)}$ holds, then it follows by Theorem 2 that $\mathit{inter}(v_l)$ is an answer set of $\mathcal{P}$. Furthermore, $A = \{a \mid \mathbf{T}a \in X^*_A\}$ and $\mathit{assn}(v_l) \subseteq X^*_A$ hold. Thus it follows that $\mathit{inter}(v_l) \subseteq A$. Assume, towards a contradiction, that $\mathit{inter}(v_l) \subset A$. Then A is not a subset minimal model of $GL_A(\mathcal{P})$, resulting in a contradiction. Thus it follows that $\mathit{inter}(v_l) = A$. This means that the algorithm finds the answer set A, i.e., $A \in \mathit{AlphaRebootASP}(\mathcal{P})$.

By the choice of A it follows $\mathit{AlphaRebootASP}(\mathcal{P}) \supseteq \mathit{AS}(\mathcal{P})$. □

Since this thesis considers gs_{def} as the grounding strategy used in the solving algorithm, the soundness and completeness results presented in this chapter hold for all the grounding strategies presented by Taupe et al. [TWF19].

3.5 Reboot Strategies

Since the presented proof of completeness assumes termination of *AlphaRebootASP*, reboots should not prevent termination for input programs where *AlphaASP* would terminate. This is especially relevant, if more than one reboot is performed during a solving run. A *reboot strategy* is defined as a decision procedure that decides, based on information gathered during the solving process, whether to perform a reboot. The most simple examples of reboot strategies would be to always decide for a reboot or always decide against it. The second strategy would result in the *AlphaASP* algorithm. The first is an example of a strategy that prevents termination for all input programs. Two possible ways to address the issue of termination in regard to reboots are using some kind of progress measure and increasing intervals between reboots.

Using a progress measure is a way of trying to ensure that in each interval between two reboots at least some progress is made towards termination. As one possible progress measure the number of learned clauses can be used. Note that it is important for reboots to not remove parts of this progress.

An alternative, for trying to ensure termination, is continually increasing the interval size between reboots. The idea here is that if *AlphaASP* terminates for some input program, then it only explores a finite part of the (potentially infinite) search space. Then by increasing the interval size between reboots continually, it will at some point be large enough to explore the whole of this finite part.

Based on these approaches, several reboot strategies are proposed in the following.

3.5.1 FIXED

Using the idea of a progress measure, an option is to simply choose a fixed value for this measure and reboot after the specified amount of progress was achieved. Using learned clauses as the progress measure, this strategy is called `FIXED` and implemented in ALPHA. It counts the number of enumeration nogoods and nogoods learned from conflicts. Both of these nogood types represent, that part of the search space was explored and thus constitute a form of progress. The single parameter of this strategy is the fixed number of nogoods after which to decide for a reboot.

3.5.2 ANSWER and ASSIGN

Reboot strategies can also be based on when answer sets are found or complete assignments are encountered. The `ANSWER` strategy decides for a reboot if an answer set was found, i.e., loop case `(answer)` of *AlphaRebootASP* was executed, but no reboot was performed since then. The `ASSIGN` strategy behaves the same except it considers both loop cases `(answer)` and `(backtrack)` instead of only case `(answer)`. This corresponds to finding a complete assignment, that can be an answer set or no answer set, before deciding for the next reboot.

3.5.3 GEOM and LUBY

In satisfiability (SAT) solving, infinite sequences have been used to obtain interval sizes [Hua07] for a technique called restarts (see Section 5). These sequences contain arbitrarily large values and can thus be used to allow intervals of arbitrary size between reboots. Two such types of sequences are geometric and Luby sequences. The latter were originally proposed by Luby et al. in the context of Las Vegas algorithms [LSZ93]. For these sequences, reboot strategies with the names `GEOM` and `LUBY` are defined respectively and implemented in ALPHA.

A geometric sequence is defined as an infinite sequence $\langle t_i \rangle = t_1, t_2, t_3, \ldots$, parametrized by initial value a and scale factor r, where $t_i = a \cdot r^{i-1}$. The strategy `GEOM` is then defined with the same two parameters a and r. Assuming n reboots have been performed so far, strategy `GEOM` decides for a reboot if at least t_{n+1} nogoods have been learned in total (including enumeration nogoods), where $\langle t_i \rangle$ is a geometric sequence with parameters a and r.

A Luby sequence [LSZ93] is defined as an infinite sequence $\langle t_i \rangle = t_1, t_2, t_3, \ldots$ where:

$$t_i = \begin{cases} 2^{k-1} & \text{if } i = 2^k - 1; \\ t_{i-2^{k-1}+1} & \text{if } 2^{k-1} \leq i < 2^k - 1. \end{cases}$$

The strategy `LUBY` is then defined with the single parameter s. The scaling factor s represents the length of the smallest interval. Assuming n reboots have been performed so far, strategy `LUBY` decides for a reboot if at least $s \cdot t_{n+1}$ nogoods have been learned in total (including enumeration nogoods), where $\langle t_i \rangle$ is a Luby sequence.

3.6 Termination

When considering the general case for an ASP program, the solving algorithms *AlphaASP* and *AlphaRebootASP* do not necessarily terminate. This can be observed with a simple example. The following program $\mathcal{P}$ has exactly one infinite answer set:

$$p(a).$$
$$p(f(X)) \leftarrow p(X).$$

Thus the algorithms cannot terminate for $\mathcal{P}$ within a finite amount of steps since in each step only a finite part of the answer set is potentially computed. Termination can thus only be ensured under additional assumptions.

To theoretically argue about termination of *AlphaRebootASP* with a wider range of reboot strategies, $\texttt{LIMITED}^{\texttt{n}}$ is defined as the class of strategies that perform at most `n` reboots before a new answer set is found. This restriction ensures that reboots and other cases of the algorithm cannot alternate indefinitely. Note that the only strategy proposed in Section 3.5 that belongs to the category $\texttt{LIMITED}^{\texttt{n}}$ is the `ANSWER` strategy, but other strategies could easily be adapted by adding the limit for `n` reboots per answer set as an additional parameter. In the following, it is assumed that a reboot strategy in the category $\texttt{LIMITED}^{\texttt{n}}$ is used by the *AlphaRebootASP* algorithm.

Consider the assumption that for input program $\mathcal{P}$ over language $\mathcal{L}$, the Herbrand base $\mathcal{HB}(\mathcal{L})$, and thus also the grounding $\text{grd}(\mathcal{P})$, are finite. From this assumption it follows that $\Delta^{\mathcal{P}}$ is also finite. Then the number of possible conflict cases, i.e. iteration case `(conflict)`, is finite since every conflict results in a new learned nogood discovered by the solver. Furthermore, the number of possible answer sets of $\mathcal{P}$ is finite since $\mathcal{HB}(\mathcal{L})$ is finite.

Note that an atom only becomes eligible for a choice again after the backjump during conflict handling. The number of possible atoms for a choice in iteration case `(choice)` until the next conflict is then finite. A backtrack in iteration case `(backtrack)` can happen at most once for each choice made. Thus the number of backtracks until the next conflict is also finite.

Consider that the number of atoms that can be assigned through propagation until the next conflict is finite. Thus the number of times grounding, i.e. iteration case `(ground)`, can be performed before the next conflict or reboot is finite as well. Note that the number of reboots until the next answer set is at most n by assumption of a reboot strategy in the class $\texttt{LIMITED}^{\texttt{n}}$.

Closing the assignment in iteration case `(close)` is then limited to a finite number of times since it cannot be performed multiple times in immediate succession and all other cases are limited to a finite number. It follows that the *AlphaRebootASP* algorithm over the given input program $\mathcal{P}$ terminates within a finite number of iterations.

CHAPTER 4

Practical Evaluation

4.1 Implementation

Intuitively the solver performs the search and the grounder provides ground rules in the form of nogoods to the solver along the way. Reboots modify the set of rules that is considered already grounded. This set is shared conceptually between the solving and grounding component. Thus both grounder and solver are affected by the implementation of reboots.

Figure 4.1 shows the resulting interactions of a reboot within the architecture of the ALPHA system presented by Leutgeb and Weinzierl [LW17].

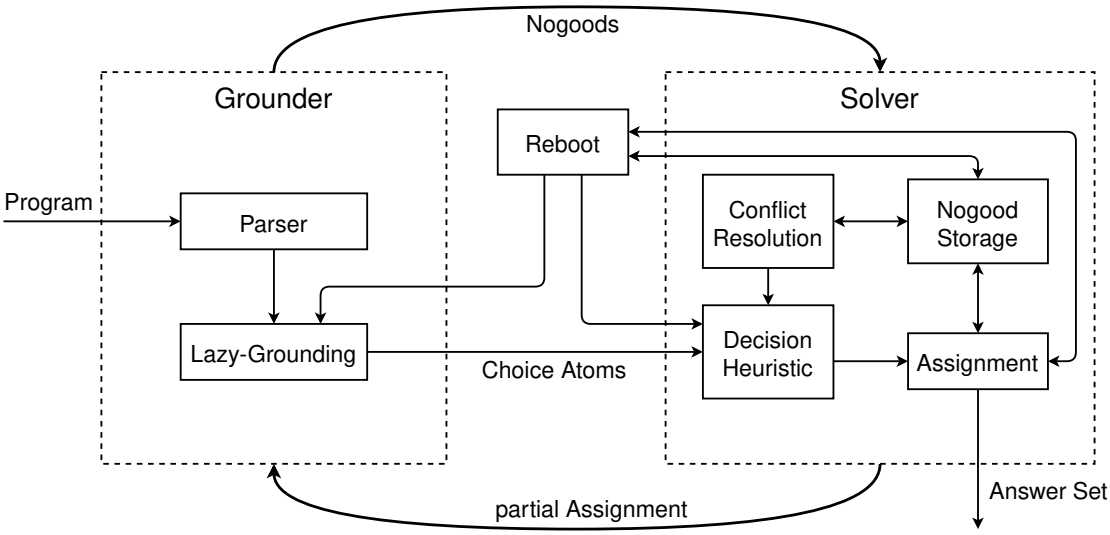

Figure 4.1: Reboots within the ALPHA system

Propagation and choices constitute changes to the assignment. Reboots can be triggered based on such changes and the decision of the reboot strategy. The reboot strategy is part of the reboot component and decides based on information gathered so far, whether a reboot should be performed. Possible strategies for this decision are described in Section 3.5.

When a reboot is performed, the nogood store component is modified such that parts of the static nogoods are removed. Furthermore, the grounder memory, located in the lazy-grounding component of the grounder, is modified according to Definition 1. This is accomplished by the following steps within the reboot component:

- First a backup of all learned nogoods (including enumeration nogoods) in the nogood store component, as well as a backup of the choices in the current assignment, i.e., the choice stack, is created.
- Then decision heuristic and lazy-grounding components are reset to their initial state, followed by the nogood store and assignment being cleared.
- In the next step the nogood backup is added to the nogood store and each rule $r\sigma$ is grounded where $\beta(r\sigma)$ is contained in some stored nogood.
- In the last step the old assignment is reconstructed based on the choice stack backup by making choices and performing propagation and grounding after each choice. These steps correspond to a modification of the set of static nogoods according to the function rbt_{ng} as described in Section 3.1.

In theory a reboot could be performed at any point during the search, including after a choice step but before the following propagation and grounding steps. This would allow reboots before exhaustive grounding was applied, which leads to the possibility that a reboot actually increases the set of synchronized ground rules. To avoid this possibility, reboots are restricted to points in the Alpha algorithm where a choice could be made. This restriction also simplifies the application of reboots in the algorithm of Alpha.

The implementation of the Alpha solver uses a translation between logical rules and their internal representation. Nogoods are stored in their internal representation within the nogood store while a separate component, the atom store, stores the translation. We decided to implement reboots in the presented way to fully reset the atom store and ensure that logical atoms are not stored multiple times.

The current implementation of reboots does not preserve information learned by the branching heuristic. Branching decisions have been observed to have a high impact on performance of conflict-driven learning SAT solvers [KSS11]. Since Alpha is also based on conflict-driven learning and uses similar heuristics to those of modern SAT solvers, preventing a complete reset of the branching heuristic might be beneficial. Thus in some way reconstructing the old heuristic state after a reboot could improve performance, especially if reboots are performed frequently. This idea of reconstruction is the challenge,

that it is specific to each decision heuristic and provides an open issue resulting from this thesis.

4.2 Experiments

Our practical examination involves the study of three hypotheses:

H_1: The decision when to reboot can have significant impact on performance.

H_2: Reboots can significantly improve solving performance.

H_3: Reboots are detrimental for problems where nogoods from grounding do not become obsolete.

The first important question in regard to reboots is whether they can actually improve performance of the solver. This is captured by hypothesis H_2 and includes the amount of memory necessary to solve problem instances as well as the runtime. An improvement in at least one aspect of performance is a prerequisite for any practical relevance of the technique.

On the other hand there are some problems for which reboots are expected to result in performance losses. This intuitively should be the case for problems where the encoding contains no potential for rules and their nogood representation to become obsolete during the search. More specifically, it should include problems where the vast majority of rules in the encoding are relevant for finding a single answer set. Hypothesis H_3 is the result of this consideration.

The decision when to perform a reboot is made by a heuristic that needs to balance computational effort with the performance improvement achieved by a better heuristic choice. To better inform the design of such heuristics it is helpful to know how much of an impact the heuristic decision has on overall performance. In case the impact is insignificant, a naive heuristic with minimal computational overhead might be sufficient. According to hypothesis H_1 this is not the case.

Benchmark Instances. To examine the performance of reboots in the Alpha solver, we use the benchmark sets from the original evaluation of Alpha by Leutgeb and Weinzierl [LW17, Wei17]. Among them are the Ground Explosion, Graph 5-Colorability, Cutedge and Reachability benchmark sets. Additionally we use a modified version of the Ground Explosion encoding to obtain a separate instance, further referred to as the Selection instance, and the Selection Explosion benchmark set where the domain size of the Selection instance is scaled up further in increments of 100 up to 2000, resulting in 20 instances. This benchmark set can be viewed as a harder version of the Ground

Explosion benchmark set. The Selection instance is constructed as follows:

$$
\begin{aligned}
dom(X) &\leftarrow X = 1..100.\\
sel(X) &\leftarrow dom(X),\ \text{not}\, nsel(X).\\
nsel(X) &\leftarrow dom(X),\ \text{not}\, sel(X).\\
&\leftarrow sel(X),\ sel(Y),\ X \neq Y,\ dom(X),\ dom(Y).\\
p(X1, X2, X3, X4, X5, X6) &\leftarrow sel(X1),\ sel(X2),\ sel(X3),\ sel(X4),\ sel(X5),\ sel(X6).
\end{aligned}
$$

The syntax 1..100 is a common shorthand for 100 instances of the same rule with one of the values $x \in \{1, \ldots, 100\}$ at this position in the rule. The remaining syntax can be found in the ASP-Core-2 language format definition [CFG+20].

Intuitively the Selection instance involves a selection of up to 100 values and derives atoms over a predicate p for combinations of selected values. A constraint prevents multiple values from being selected. Note that adding $nsel(X)$ in the head of the constraint would not change the semantics of the program but might lead to different computational behavior. The last rule leads to a large grounding size since it involves a predicate with high arity. There exists a constraint instantiation for every combination of values in the domain, but after an answer set for some selected value x is discovered, the solver should be able to learn to not select x anymore, leading to the constraint combinations with x becoming redundant. From this arises the expectation that reboots lead to improved performance, especially in necessary memory, for the Selection instance since large numbers of obsolete nogoods might be removed.

Benchmark Setup. We compared Alpha with reboots to Alpha without reboots and Clingo [GKKS19], one of the most efficient ground-and-solve systems. For these comparisons Alpha was built from source using Git commit 3dfad44 and Clingo version 5.4.0 was used. Benchmarks were run on a machine cluster with each machine featuring two Intel® Xeon® E5-2650 v4 CPUs, 252 GB of memory and Ubuntu 16.04.1 LTS Linux. Benchmarks were scheduled on the cluster using HTCondor™. For measurements of time and memory consumption as well as enforcing time and memory limits, version 3.4.0 of the runsolver tool [Rou11] was used. Timeout was set to 300 seconds and memory limit to 12000 MiB. Each benchmark configuration was run five times and median performance was used for the evaluations.

4.2.1 Benchmark – Single Reboot

For the first experiment, the impact of a single reboot is examined. The Selection instance is used in combination with the `FIXED` reboot strategy, where values 0, 10, 20, 30, ..., 200 were used for the strategy parameter to obtain 21 total configurations. Recall that the parameter of strategy `FIXED` controls the number of nogoods learned before a reboot is performed. The results of this experiment are depicted in Figure 4.2. In the run with parameter value 200, no reboot was performed while for all other runs exactly one reboot was performed. The results show a speedup by about a factor of two for the best

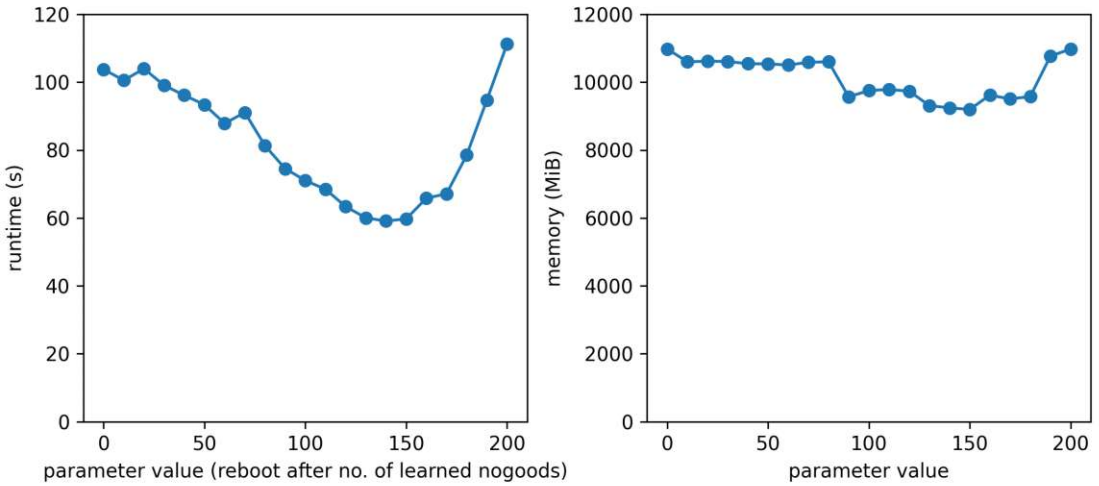

Figure 4.2: Single Reboot benchmark results

case of the parameter value compared to when no reboot was performed. The speedup decreases the more the parameter deviates from the best case. The memory results are less significant compared to runtime improvements as only about 10% of the memory consumption is avoided with the best choice of the parameter.

4.2.2 Benchmark – Multiple Reboots

In the second experiment, the impact of multiple reboots is examined, again using the Selection instance and the `FIXED` reboot strategy. The values $\{1, 10, 20, 30, \ldots, 200\}$ are used to parametrize the strategy. Note that the parameter value 0 for strategy `FIXED` with repeated reboots would necessarily lead to non-termination. The results of this experiment are visualized in Figure 4.3. Numerical median results, including the number of reboots performed for each parameter value, are shown in Table 4.1. The results show improvements in runtime performance by up to a factor of five and in memory performance by up to a factor of two. Furthermore, rebooting more frequently tends to lead to better runtime and memory performance for this instance except for parameter value 1, where runtime does not decrease further, but instead increases. We assume that this is the effect of the high amount of reboots that results in a significant overhead for executing the reboots. Additionally, frequently resetting the branching heuristic might lead to significantly worse branching decisions in this case.

The experiments investigating a single as well as multiple reboots provide evidence for hypotheses H_1 and H_2. Furthermore, these experiments show more significant improvement for runtime than for memory consumption. Our initial expectation was that the main upside of reboots would be lower memory consumption. This is investigated further in the following benchmarks.

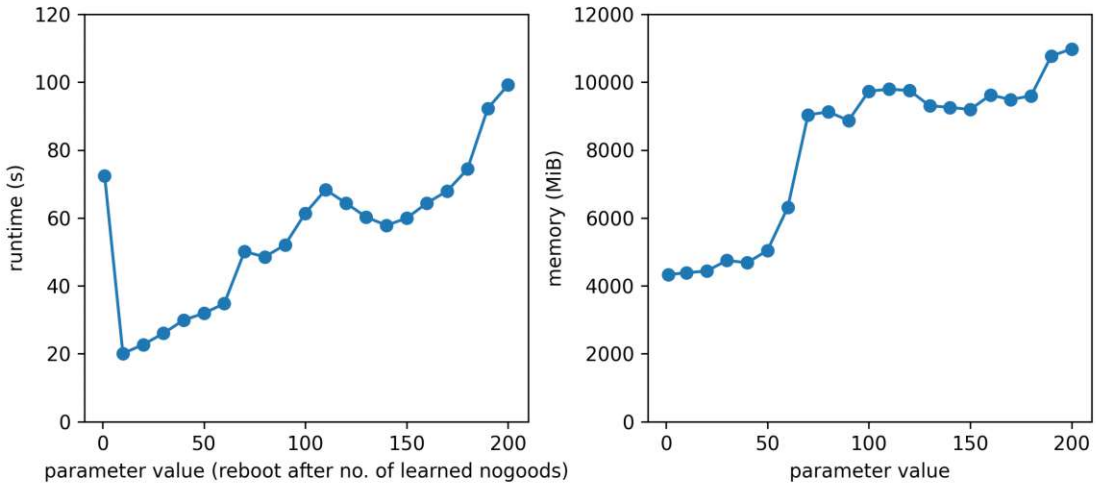

Figure 4.3: Multiple Reboots benchmark results

parameter value	runtime (s)	memory (MiB)	reboots
1	72.4169	4328.57	640
10	20.0693	4385.04	33
20	22.6616	4434.99	13
30	25.9946	4747.97	8
40	29.9122	4679.23	5
50	31.8925	5044.55	4
60	34.7577	6311.42	3
70	50.1080	9035.75	3
80	48.5278	9125.72	2
90	52.0672	8870.92	2
100	61.3979	9735.97	2
110	68.3188	9792.46	1
120	64.3945	9754.82	1
130	60.2662	9306.02	1
140	57.8219	9257.57	1
150	59.9178	9195.67	1
160	64.3954	9621.98	1
170	67.9484	9488.44	1
180	74.3969	9597.52	1
190	92.2684	10772.70	1
200	99.1725	10979.70	0

Table 4.1: Multiple Reboots benchmark results

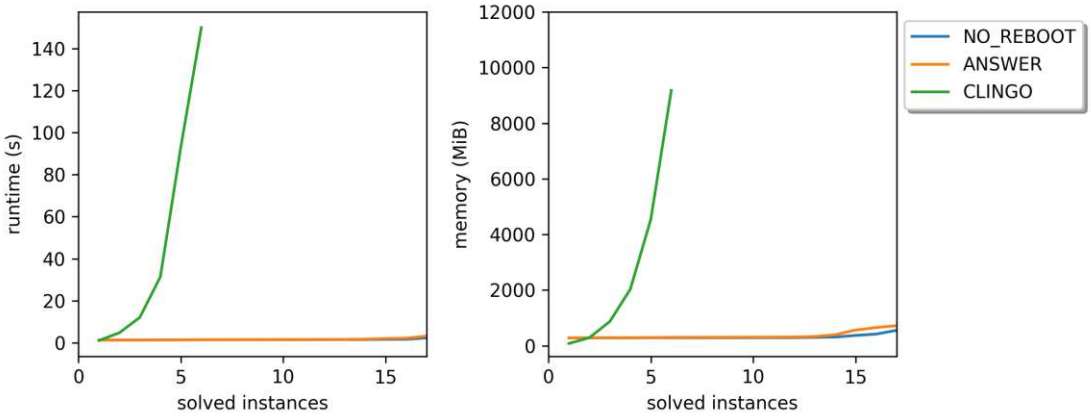

Figure 4.4: Ground Explosion benchmark results

4.2.3 Further Benchmarks

The benchmarks presented in the following use ALPHA without reboots (NO_REBOOT), ALPHA with the ANSWER reboot strategy (ANSWER) and CLINGO (CLINGO). For every instance 20 answer sets were requested from the solver. The specific benchmark sets were chosen to exercise different parts of an ASP system [LW17]. Both Reachability and Graph 5-Colorability encodings contain information, that is intuitively relevant for all answer sets, and don't present any obvious candidates for rules that might become obsolete. Since the encoding in the Reachability benchmark set is a positive program, these instances have exactly one answer set and the ANSWER reboot strategy is not expected to lead to any performance improvements.

Benchmark results are presented in Figures 4.4–4.8.

Ground Explosion. Both configurations of ALPHA solve all 17 instances of the Ground Explosion benchmark set and show very similar runtime and memory performance. CLINGO only solves 6 instances and shows significantly worse performance. Since this benchmark set is grounding intensive, this result is not unexpected.

Selection Explosion. For the Selection Explosion benchmark set, ALPHA without reboots solves 7 instances while ALPHA with reboots solves all 20 instances. Furthermore, the performance with reboots starts out similar or slightly worse and significantly improves with larger instances. CLINGO does not solve any instances of this benchmark set.

Reachability. Both ALPHA configurations and CLINGO solve all 50 instances of the Reachability benchmark set. The ALPHA configurations perform very similar but noticeably worse than CLINGO. Considering that the use of reboots is not expected to improve performance for this problem, it also does not show significantly worse performance.

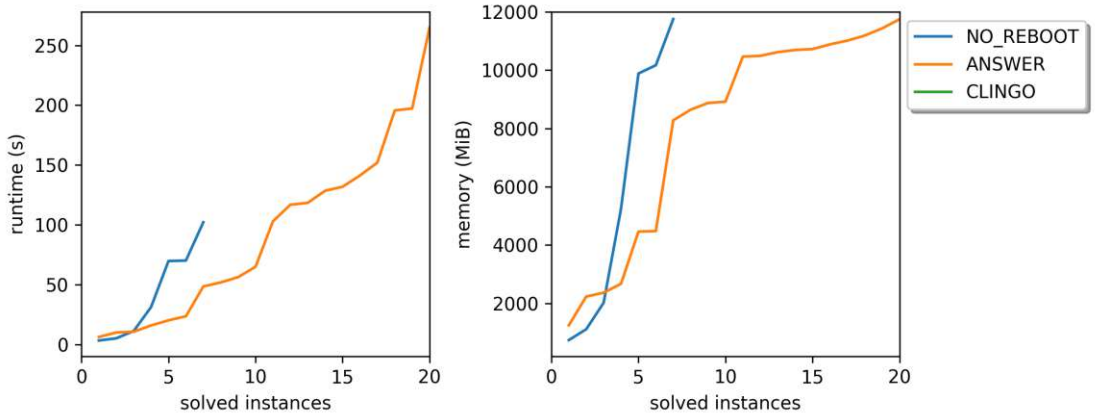

Figure 4.5: Selection Explosion benchmark results

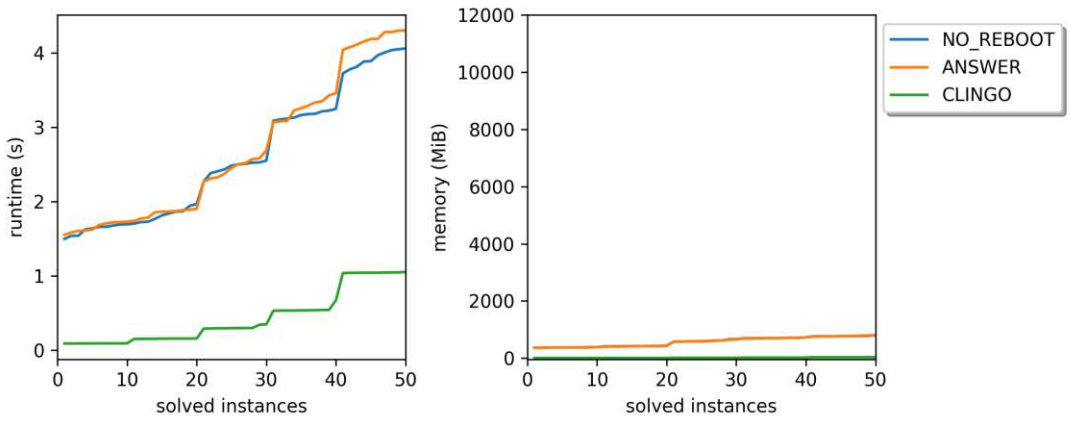

Figure 4.6: Reachability benchmark results

This might be a product of small instance sizes, considering that all instances are solved within less than five seconds.

Graph 5-Colorability. Both Alpha configurations and Clingo solve all 180 instances of the Graph 5-Colorability benchmark set and Clingo shows significantly better performance. Furthermore, while Alpha with reboots solves all instances, it exhibits worse time and memory performance for a large part of the instances. This is expected since the Graph 5-Colorability encoding intuitively does not provide potential for rules becoming obsolete, meaning reboots would remove only relevant rules.

Cutedge. For the Cutedge benchmark set, Clingo solves 40 instances, Alpha without reboots solves 80 instances and Alpha with reboots solves 100 of the 130 total instances.

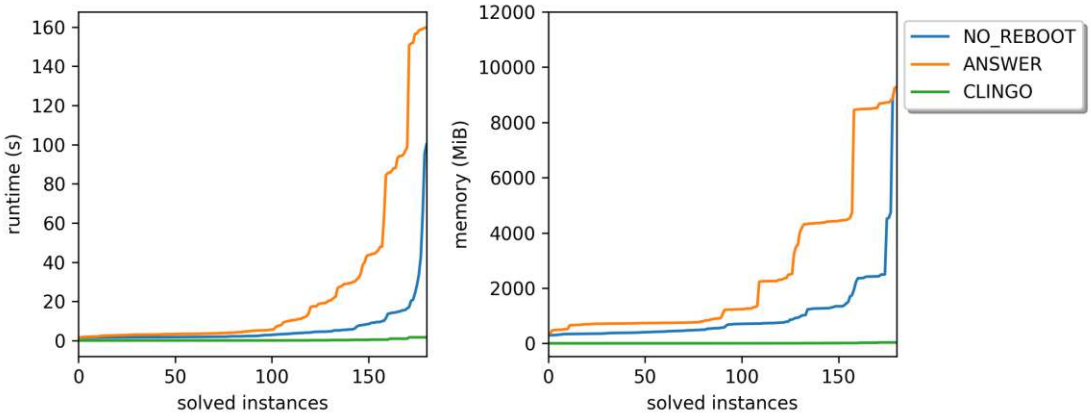

Figure 4.7: Graph 5-Colorability benchmark results

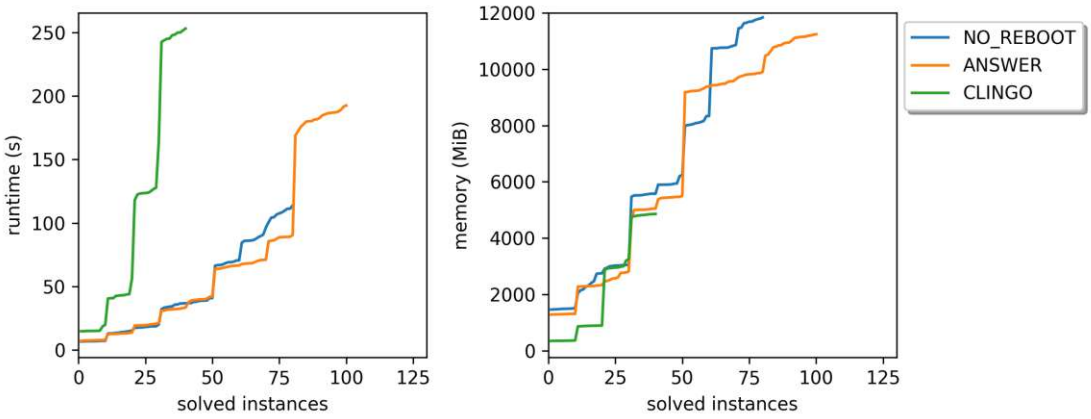

Figure 4.8: Cutedge benchmark results

Furthermore, using reboots shows better runtime and memory performance than not using them for most of the instances. While CLINGO performs worse in regard to runtime overall, it performs similar in regard to memory consumption for the instances it solves. The Cutedge encoding contains potential for rules to become obsolete for specific parts of the search. Thus reboots are expected to have a positive impact for this benchmark set.

4.2.4 Summary

The experiments using the Selection instance and `FIXED` reboot strategy show significant performance improvements that depend on when and how frequent reboots are performed, providing evidence for hypotheses H_1 and H_2. Further investigations using the `ANSWER` strategy on the Ground Explosion, Selection Explosion, Graph 5-Colorability, Cutedge and Reachability benchmark sets provide more evidence in the following ways. For

the Selection Explosion and Cutedge benchmark sets, Alpha with reboots solves more instances and leads to better performance than without reboots or using Clingo. This constitutes evidence towards hypothesis H_2. Since the 5-colorability encoding does not provide potential for obsolete nogoods, the exhibited worse performance is evidence towards hypothesis H_3. Experiments on the remaining benchmark sets show very similar performance of Alpha with and without reboots.

Compared to the alternatives proposed in Section 3.5, the `ANSWER` and `ASSIGN` strategies avoid the need for parametrization. The `ANSWER` strategy takes the most conservative approach and cannot lead to excessive reboots like the `FIXED` strategy in particular. Obtaining parameter values that provide reasonable results on instances of varying size for the proposed parametrized strategies is an open issue.

CHAPTER 5

Related Work

Related Techniques. The development of ASP solvers was influenced significantly by the related field of satisfiability (SAT) solving. Numerous techniques, originally developed for SAT solvers, have been adopted for ground-and-solve ASP solvers [GKK+11, ADF+13, LPF+02]. Some of these techniques have also been adapted for lazy grounding and implemented in ALPHA [Wei17, LW17, WTF20]. Examples include conflict-driven learning, restarts, phase-saving and propagation based on watched-literals. Some techniques used in ASP solvers work under the assumption that the whole problem representation is known. This assumption holds for SAT solving and upfront grounding in ASP solving. It does not hold in general for lazy grounding ASP systems, since there the problem representation is constructed incrementally during the search.

Learned Nogood Deletion. Another technique implemented in ALPHA that originates from SAT solving is learned nogood deletion [WTF20]. This technique, similarly to the proposed reboots, deletes nogoods. It differs from reboots in the type of nogoods that are deleted. While learned nogood deletion removes nogoods learned by resolution from the solvers memory, reboots remove static nogoods obtained from grounding.

Restarts. Gomes et al. [GSK98, GSCK00] observed specific combinations of problem instances and deterministic search algorithms resulting in significantly higher runtime. This so-called heavy-tailed behavior was observed for SAT and constraint satisfaction problems (CSP). The researchers proposed the introduction of a controlled degree of randomness into the otherwise deterministic SAT and CSP solvers; the resulting technique was called a restart. When performing a restart the solver backjumps to decision level 0, but retains all learned information. This includes learned clauses (in case of SAT) or nogoods (in case of ASP) as well as information gathered by the decision heuristic. Different strategies for when to perform restarts have been developed. Some of them use an infinite increasing sequence to define the size of intervals between restarts.

Geometric and Luby [LSZ93] sequences (described in Section 3.5) have been used for this purpose [Hua07]. Modern SAT solvers perform rapid restarts [AS12] combined with phase-saving [PD07], which reuses the most recent polarity of variables for decisions.

The main factor that distinguishes both of these techniques from reboots is that they don't remove parts of the static problem encoding. Reboots accomplish this by deleting nogoods obtained from grounding. Since SAT solving and upfront grounding in ASP solving assume the static problem representation to be fully known at any point of the search, no part of it can be removed without violating this assumption. In lazy-grounding it is possible to delete static nogoods since they can be obtained again from the grounder when needed.

Lazy-Grounding. The first lazy-grounding ASP solvers were GASP [PDPR09] and ASPeRiX [LN09b, LN09a, LBSG17]. They are both based on the notion of a computation sequence as described in Section 2.3.4, but were developed independently. GASP was implemented in Prolog while ASPeRiX was implemented in C++. Another lazy-grounding system is the OMiGA [DEF+12] solver that was built to improve grounding performance. It was developed in Java and features a Rete network [For82], that stores partial matches to speed up grounding.

GASP, ASPeRiX and OMiGA address the problem of the grounding bottleneck, but are not based on conflict-driven learning. Thus they do not profit from many of the techniques developed for conflict-driven clause learning (CDCL) solvers in the SAT solving field.

Alternatives to Lazy-Grounding. Other approaches to avoid the grounding bottleneck have also been investigated. The intelligent grounding in $\mathcal{I}$-DLV [CFPZ17] prevents grounding unnecessary parts of the program and is based on semi-naive database techniques. Incremental ASP [GKK+08] reuses grounding results from smaller sizes for larger ones in bounded problems.

Eiter et al. [EFM10] proposed methods for evaluating ASP programs, that contain predicates of bounded arity, in polynomial space. Two of their methods also require programs to be head-cycle-free. A program $\mathcal{P}$ is considered head-cycle-free if the positive dependency graph does not contain any cycle with at least two atoms in rule heads of $\mathcal{P}$. The positive dependency graph contains the predicates occuring in $\mathcal{P}$ as nodes and the following set of edges: $E = \{(p_1, p_2) \mid \exists r \in \mathcal{P} : \ p_1 \text{ occurs in } H(r) \text{ and } p_2 \text{ in } B^+(r)\}$.

Another approach is the rewriting of rules during a preprocessing step. Bichler et al. [BMW20] presented the preprocessing algorithm `lpopt`. The `lpopt` algorithm splits up rules based on tree decompositions over the variables in a rule. This splitting is done in a static fashion, meaning that a rule is always decomposed according to the obtained tree decomposition if possible. This is a disadvantage in cases where keeping the original rule instead would be preferable. Calimeri et al. [CPZ19] proposed a dynamic rewriting algorithm that estimates the cost of grounding for possible rule decompositions and chooses the best decomposition (or the original rule) based on these estimates.

An approach proposed by Besin et al. [BHW22], called body-decoupled grounding, rewrites normal programs into disjunctive programs with grounding size bounded by predicate arities instead of the number of variables in a rule. This technique results in a set of rules for each rewritten rule and transfers part of the effort, otherwise contained in grounding, to the solving part instead.

Cuteri et al. [CDRS20] investigated the case where the grounding bottleneck is caused by constraints. They proposed and implemented a technique (on top of the ground-and-solve system WASP [ADLR15]) that converts these constraints into propagators within the conflict-driven learning algorithm. This way problem instances can be solved without problematic constraints being grounded explicitly.

Most of these approaches are restricted in their applicability or only have limited impact on the grounding bottleneck. Lazy grounding with reboots can be applied to arbitrary ASP programs and the frequency of reboots can be heuristically controlled to significantly reduce memory consumption for instances exhibiting the grounding bottleneck.

CHAPTER 6

Conclusion

Reboots in lazy-grounding ASP solving were defined and a modification of the algorithm in Alpha, that incorporates reboots, was presented. The formal notion of a computation tree was defined to formalize the search process of the modified Alpha algorithm. Detailed formal proofs of soundness and completeness, based on the notion of the computation tree, were provided for the modified algorithm. Several reboot strategies, using different approaches to the decision of when to reboot, were proposed. Furthermore, arguments for termination of the modified algorithm under restricted sets of programs and reboot strategies were given. Reboots, as well as the proposed reboot strategies, were implemented in Alpha and their interactions within the Alpha system were illustrated. Finally, three hypotheses about the impact of reboots on performance were presented and examined. Experimental results reinforce the presented hypotheses. More specifically, reboots show potential for performance improvements on grounding intensive benchmarks. Not only is memory consumption reduced, leading to more instances being solved, but reboots also result in lower runtime for these benchmarks.

Open Issues. One of the hypotheses that was reinforced by performed benchmarks suggests that reboots are detrimental for specific problems, where they remove mostly or exclusively promising nogoods. Dynamic strategies could prevent reboots from being performed repeatedly for such problems. The presented reboot strategies provide a possible starting point for a more fine-grained examination of the decision when to reboot. A quality measurement for static nogoods, similar to the literals blocks distance in SAT solving [AS09], could allow for a dynamic approach to reboot strategies. Another possible approach would be allowing users to provide problem-specific information to the reboot strategy by developing a specification language, similar to declarative specifications of domain-specific decision heuristics [GKR$^+$13, CFSW23].

The provided implementation of reboots in Alpha resets the state of the decision heuristic. As heuristic decisions are an important part of the conflict-driven learning solver Alpha,

retaining as much information learned by the heuristic might be beneficial. Development of heuristic-specific ways to accomplish this provides potential for future improvement of reboots.

The proposed definition of reboots aims at removing as many obsolete nogoods as possible. Using a measurement for the quality of static nogoods could allow an informed decision about which nogoods appear to be more promising. Based on this a more fine-grained technique than reboots could be developed, that removes less nogoods but keeps promising ones. Furthermore, since reboots have the goal of removing unnecessary information, the question arises whether it is possible to transform programs accordingly. Decreasing dependence between subproblems of a program during a preprocessing step could allow better exploitation of reboots.

List of Figures

List of Tables

List of Algorithms

Bibliography

[ADF+13] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native ASP solver based on constraint learning. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2013*, volume 8148 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2013.

[ADLR15] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2015*, volume 9345 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2015.

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence, IJCAI 2009*, pages 399–404, 2009.

[AS12] Gilles Audemard and Laurent Simon. Refining restarts strategies for SAT and UNSAT. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming, CP 2012*, volume 7514 of *Lecture Notes in Computer Science*, pages 118–126. Springer, 2012.

[BHW22] Viktor Besin, Markus Hecher, and Stefan Woltran. Body-decoupled grounding via solving: A novel approach on the ASP bottleneck. In *Proceedings of the 31st International Joint Conference on Artificial Intelligence, IJCAI 2022*, pages 2546–2552. ijcai.org, 2022.

[BMW20] Manuel Bichler, Michael Morak, and Stefan Woltran. lpopt: A rule optimization tool for answer set programming. *Fundamenta Informaticae*, 177(3-4):275–296, 2020.

[CDRS20] Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1688–1694. ijcai.org, 2020.

[CFG+20] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format. *Theory and Practice of Logic Programming*, 20(2):294–309, 2020.

[CFPZ17] Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale*, 11(1):5–20, 2017.

[CFSW23] Richard Comploi-Taupe, Gerhard Friedrich, Konstantin Schekotihin, and Antonius Weinzierl. Domain-specific heuristics in answer set programming: A declarative non-monotonic approach. *Journal of Artificial Intelligence Research*, 76:59–114, 2023.

[CPZ19] Francesco Calimeri, Simona Perri, and Jessica Zangari. Optimizing answer set computation via heuristic-based decomposition. *Theory and Practice of Logic Programming*, 19(4):603–628, 2019.

[DEF+12] Minh Dao-Tran, Thomas Eiter, Michael Fink, Gerald Weidinger, and Antonius Weinzierl. Omiga: An open minded grounding on-the-fly answer set solver. In *Proceedings of the 13th European Conference on Logics in Artificial Intelligence, JELIA 2012*, volume 7519 of *Lecture Notes in Computer Science*, pages 480–483. Springer, 2012.

[EFM10] Thomas Eiter, Wolfgang Faber, and Mushthofa Mushthofa. Space efficient evaluation of ASP programs with bounded predicate arities. In *Proceedings of the 24th Conference on Artificial Intelligence, AAAI 2010*. AAAI Press, 2010.

[EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web. Semantic Technologies for Information Systems, 5th International Summer School 2009*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

[FFS+18] Andreas A. Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich Christian Teppan. Industrial applications of answer set programming. *Künstliche Intelligenz*, 32(2-3):165–176, 2018.

[For82] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[GKK+08] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental ASP solver. In *Proceedings of the 24th International Conference on Logic Programming, ICLP 2008*, volume 5366 of *Lecture Notes in Computer Science*, pages 190–205. Springer, 2008.

[GKK+11] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011.

[GKKS12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[GKKS19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019.

[GKR+13] Martin Gebser, Benjamin Kaufmann, Javier Romero, Ramón Otero, Torsten Schaub, and Philipp Wanko. Domain-specific heuristics in answer set programming. In Marie desJardins and Michael L. Littman, editors, *Proceedings of the 27th Conference on Artificial Intelligence, AAAI 2013*. AAAI Press, 2013.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[GLM+18] Martin Gebser, Nicola Leone, Marco Maratea, Simona Perri, Francesco Ricca, and Torsten Schaub. Evaluation techniques and systems for answer set programming: a survey. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI 2018*, pages 5450–5456. ijcai.org, 2018.

[GSCK00] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning, JAR*, 24(1/2):67–100, 2000.

[GSK98] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98*, pages 431–437. AAAI Press / The MIT Press, 1998.

[Hua07] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 2007*, pages 2318–2323, 2007.

[KSS11] Hadi Katebi, Karem A. Sakallah, and João P. Marques Silva. Empirical study of the anatomy of modern sat solvers. In *Theory and Applications of Satisfiability Testing, SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 343–356. Springer, 2011.

[LBSG17] Claire Lefèvre, Christopher Béatrix, Igor Stéphan, and Laurent Garcia. Asperix, a first-order forward chaining approach for answer set computing. *Theory and Practice of Logic Programming*, 17(3):266–310, 2017.

[Lif19] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019.

[LLTW] Michael Langowski, Lorenz Leutgeb, Richard Taupe, and Antonius Weinzierl. Alpha. `https://github.com/alpha-asp/alpha`. Accessed: 2023-06-26.

[LN09a] Claire Lefèvre and Pascal Nicolas. A first order forward chaining approach for answer set computing. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2009*, volume 5753 of *Lecture Notes in Computer Science*, pages 196–208. Springer, 2009.

[LN09b] Claire Lefèvre and Pascal Nicolas. The first version of a new ASP solver: Asperix. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2009*, volume 5753 of *Lecture Notes in Computer Science*, pages 522–527. Springer, 2009.

[LPF+02] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2002.

[LPST10] Lengning Liu, Enrico Pontelli, Tran Cao Son, and Miroslaw Truszczyński. Logic programs with abstract constraint atoms: The role of computations. *Artificial Intelligence*, 174(3):295–315, 2010.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. In *Proceedings of the Second Israel Symposium on Theory of Computing Systems, ISTCS 1993*, pages 128–133. IEEE Computer Society, 1993.

[LW17] Lorenz Leutgeb and Antonius Weinzierl. Techniques for efficient lazy-grounding ASP solving. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming, DECLARE 2017*, volume 10997 of *Lecture Notes in Computer Science*, pages 132–148. Springer, 2017.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing, SAT 2007*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[PDPR09] Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. GASP: answer set programming with lazy grounding. *Fundamenta Informaticae*, 96(3):297–322, 2009.

[Rou11] Olivier Roussel. Controlling a solver execution with the runsolver tool. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 7(4):139–144, 2011.

[SS96] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996*, pages 220–227. IEEE Computer Society / ACM, 1996.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285 – 309, 1955.

[TWF19] Richard Taupe, Antonius Weinzierl, and Gerhard Friedrich. Degrees of laziness in grounding - effects of lazy-grounding strategies on ASP solving. In *Proceedings of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2019*, volume 11481 of *Lecture Notes in Computer Science*, pages 298–311. Springer, 2019.

[Wei17] Antonius Weinzierl. Blending lazy-grounding and CDNL search for answer-set solving. In *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2017*, volume 10377 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2017.

[WTF20] Antonius Weinzierl, Richard Taupe, and Gerhard Friedrich. Advancing lazy-grounding ASP solving techniques - restarts, phase saving, heuristics, and more. *Theory and Practice of Logic Programming*, 20(5):609–624, 2020.