Blending Lazy-Grounding and CDNL Search for Answer-Set Solving*

Antonius Weinzierl

TU Wien
Institute of Information System, Knowledge-based Systems Group
Favoritenstr. 9-11, 1040 Vienna, Austria
weinzierl@kr.tuwien.ac.at

Abstract. Efficient state-of-the-art answer-set solvers are two-phased: first grounding the input program, then applying search based on conflict-driven nogood learning (CDNL). The latter provides superior search performance but the former causes exponential memory requirements for many ASP programs. Lazy-grounding avoids this grounding bottleneck but exhibits poor search performance. The approach here aims for the best of both worlds: grounding and solving are interleaved, but there is a solving component distinct from the grounding component. The solving component works on (ground) nogoods, employs conflict-driven first-UIP learning and enables heuristics. Guessing is on atoms that represent applicable rules, atoms may be one of true, false, or must-be-true, and nogoods have a distinguished head literal. The lazy-grounding component is loosely coupled to the solver and may yield more ground instances than necessary, which avoids re-grounding whenever the solver moves from one search branch to another. The approach is implemented in the new ASP solver Alpha.

1 Introduction

This work presents an approach at blending lazy-grounding and search procedures based on conflict-driven nogood learning (CDNL) for Answer-Set Programming (ASP). The most efficient state-of-the-art ASP solvers (e.g., Clasp [7] or Wasp [1]) are two-phased: first, the input program is grounded, then the answer-sets of the resulting propositional ground program are searched using advanced search techniques like CDNL, showing superior search performance. Since the pre-grounding increases the program size exponentially, however, there are many real-world applications where ASP cannot be used due to this so-called *grounding bottleneck* (cf. [2]). Intelligent grounding [3] mitigates this issue by excluding those parts of the grounding that, in advance, can be determined to be unnecessary, constraint ASP approaches (cf. [10, 16]) try escaping the bottleneck by adding constraint programming techniques, while incremental ASP [6] gradually expands certain parts of the program step-by-step. They are all inherently restricted, however, so users of ASP encounter the grounding bottleneck quickly.

* This research has been funded by the Austrian Science Fund (FWF): P27730.

Lazy-grounding on the other hand promises to avoid the grounding bottleneck in general by interleaving grounding and solving, i.e., rules of an ASP program are grounded during solving and only when needed. Indeed, lazy-grounding ASP systems like Gasp [15], Asperix [13, 12, 11], or Omiga [4, 17] show little trouble with grounding size, but their search performance is quite bad compared to pre-grounding ASP systems. Lazy-grounding ASP systems do not profit from (up-to exponential) search-space reducing techniques like conflict-driven learning and backjumping, since lazy-grounding ASP solving differs significantly from CDNL-based ASP solving in the following ways.

Lazy-grounding is based on so-called computation sequences which correspond to sequences of firing rules. Consequently, guessing is on whether an applicable rule fires, while CDNL-based search guesses on whether an atom is *true* or *false*. Lazy-grounding solvers make use of *must-be-true* as a third truth value to distinguish whether an atom was derived by a rule firing or by a constraint. CDNL-based solvers, on the other hand, have no equivalent of *must-be-true* and propagation on nogoods must be adapted. Furthermore, ordinary nogoods (i.e., the input to CDNL-based ASP solvers) cannot distinguish between a rule and a constraint. Therefore, combining lazy-grounding with CDNL-based search is challenging and necessitates intricate adaptions in the search procedures as well as on the underlying data structures.

The contributions of this work are:

- an extension of nogoods to faithfully represent rules in the presence of *must-be-true*, allowing choices to correspond to the firing of applicable rules.
- lazy-grounding that is only loosely coupled to the search state in order to avoid re-grounding of the same rules in different parts of the search space.
- a CDNL-based algorithm, *AlphaASP*, employing lazy-grounding, conflict-driven learning following the first-UIP schema, backjumping, and heuristics.
- an implementation, the Alpha ASP system, combining the best of both worlds: lazy-grounding to avoid the grounding bottleneck and CDNL-based search procedures for good search performance.
- finally, by the above it also works out the differences between lazy-grounding and CDNL-based search and demonstrates their commonalities.

The remainder of this work is structured as follows. In Section 2 preliminary concepts are introduced and in Section 3 the novel nogood representation for rules is presented. Section 4 describes the loosely-coupled lazy-grounding and Section 5 presents the Alpha algorithm blending lazy-grounding and CDNL search. Discussion is in Section 6 and Section 7 concludes.

2 Preliminaries

Let $\mathcal{C}$ be a finite set of constants, $\mathcal{V}$ be a set of variables, and $\mathcal{P}$ be a finite set of predicates with associated arities, i.e., elements of $\mathcal{P}$ are of the form p/k where p is the predicate name and k its arity. We assume each predicate name occurs only with one arity. The set $\mathcal{A}$ of (non-ground) atoms is then given by

$\{p(t_1, \ldots, t_n) \mid p/n \in \mathcal{P}, t_1, \ldots, t_n \in \mathcal{C} \cup \mathcal{V}\}$. An atom $at \in \mathcal{A}$ is ground if no variable occurs in at and the set of variables occurring in at is denoted by $var(at)$. The set of all ground atoms is denoted by $\mathcal{A}_{grd}$. A (normal) rule is of the form:

$$at_0 \leftarrow at_1, \ldots, at_k, not\ at_{k+1}, \ldots, not\ at_n.$$

where each $at_i \in \mathcal{A}$ is an atom, for $0 \leq i \leq n$. For such a rule r the head, positive body, negative body, and body are defined as $H(r) = \{at_0\}$, $B^+(r) = \{at_1, \ldots, at_k\}$, $B^-(r) = \{at_{k+1}, \ldots, at_n\}$, and $B(r) = \{at_1, \ldots, at_n\}$, respectively. A rule r is a constraint if $H(r) = \emptyset$, a fact if $B(r) = \emptyset$, and ground if each $at \in B(r) \cup H(r)$ is ground. The variables occurring in r are given by $var(r) = \bigcup_{at \in H(r) \cup B(r)} var(at)$. A literal l is positive if $l \in \mathcal{A}$, otherwise it is negative. A rule r is *safe* if all variables occurring in r also occur in its positive body, i.e., $var(r) \subseteq \bigcup_{a \in B^+(r)} var(a)$.

A program P is a finite set of safe rules. P is ground if each $r \in P$ is. A (Herbrand) interpretation I is a subset of the Herbrand base wrt. P, i.e., $I \subseteq \mathcal{A}_{grd}$. An interpretation I satisfies a literal l, denoted $I \models l$ if $l \in I$ for positive l and $l \notin I$ for negative l. I satisfies a ground rule r, denoted $I \models r$ if $B^+(r) \subseteq I \wedge B^-(r) \cap I = \emptyset$ implies $H(r) \subseteq I$ and $H(r) \neq \emptyset$. Given an interpretation I and a ground program P, the FLP-reduct P^I of P wrt. I is the set of rules $r \in P$ whose body is satisfied by I, i.e., $P^I = \{r \in P \mid B^+(r) \subseteq I \wedge B^-(r) \cap I = \emptyset\}$. I is an *answer-set* of a ground program P if I is the subset-minimal model of P^I; the set of all answer-sets of P is denoted by $AS(P)$.

A substitution $\sigma : \mathcal{V} \to \mathcal{C}$ is a mapping of variables to constants. Given an atom at the result of applying a substitution σ to at is denoted by $at\sigma$; this is extended in the usual way to rules r, i.e., $r\sigma$ for a rule of the above form is $at_0\sigma \leftarrow at_1\sigma, \ldots, not\ at_n\sigma$. Then, the grounding of a rule is given by $grd(r) = \{r\sigma \mid \sigma \text{ is a substitution for all } v \in var(r)\}$ and the grounding $grd(P)$ of a program P is given by $grd(P) = \bigcup_{r \in P} grd(r)$. Elements of $grd(P)$ and $grd(r)$ are called ground instances of P and r, respectively. The answer-sets of a non-ground program P are given by the answer-sets of $grd(P)$.

CDNL-based ASP solving takes a ground program, translates it into nogoods and then runs a SAT-inspired (i.e., a DPLL-style) model building algorithm to find a solution for the set of nogoods. A nogood is comprised of Boolean signed literals, which are of the form $\mathbf{T}at$ and $\mathbf{F}at$ for $at \in \mathcal{A}$; a nogood $ng = \{s_1, \ldots, s_n\}$ is a set of Boolean signed literals s_i, $1 \leq i \leq n$, which intuitively states that a solution cannot satisfy all literals s_1 to s_n. Nogoods are interpreted over assignments, which are sets A of Boolean signed literals, i.e., an assignment is a (partial) interpretation where false atoms are represented explicitly. A solution for a set Δ of nogoods then is an assignment A such that $\{at \mid \mathbf{T}at \in A\} \cap \{at \mid \mathbf{F}at \in A\} = \emptyset$, $\{at \mid \mathbf{T}at \in A\} \cup \{at \mid \mathbf{F}at \in A\} = \mathcal{A}$, and no nogood $ng \in \Delta$ is violated, i.e., $ng \not\subseteq A$. A solution thus corresponds one-to-one to an interpretation that is a model of all nogoods. For more details and algorithms, see [7–9]. Also note that CDNL-based solvers for ASP employ additional checks to ensure that the constructed model is supported and unfounded-free, but these checks are not necessary in the approach presented.

Lazy-grounding, also called grounding on-the-fly, is built on the idea of a computation, which is a sequence $(A_0, \ldots, A_\infty)$ of assignments starting with the empty set and adding at each step the heads of applicable rules (cf. [15, 11, 4]). A ground rule r is *applicable* in a step A_k, if its positive body already has been derived and its negative body is not contradicted, i.e., $B^+(r) \subseteq A_k$ and $B^-(r) \cap A_k = \emptyset$. A computation $(A_0, \ldots, A_\infty)$ has to satisfy the following conditions besides $A_0 = \emptyset$, given the usual immediate-consequences operator T_P:

1. $\forall i \geq 1 : A_i \subseteq T_P(A_{i-1})$ (the computation contains only consequences),
2. $\forall i \geq 1 : A_{i-1} \subseteq A_i$ (the computation is monotonic),
3. $A_\infty = \bigcup_{i=0}^{\infty} A_i = T_P(A_\infty)$ (the computation converges), and
4. $\forall i \geq 1 : \forall at \in A_i \setminus A_{i-1}, \exists r \in P$ such that $H(r) = at$ and $\forall j \geq i-1 : B^+(r) \subseteq A_j \wedge B^-(r) \cap A_j = \emptyset$ (applicability of rules is persistent through the computation).

It has been shown that A is an *answer-set* of a normal logic program P iff there is a computation $(A_0, \ldots, A_\infty)$ for P such that $A = A_\infty$ [12]. Observe that A is finite, i.e., $A_\infty = A_n$ for some $n \in \mathbb{N}$, because $\mathcal{C}, \mathcal{P}$, and P are finite.

3 Nogood Representation of Rules

The goal of the Alpha approach is to combine lazy-grounding with CDNL-search in order to obtain an ASP solver that avoids the grounding bottleneck and shows very good search performance. The techniques of CDNL-based ASP solvers require a fully grounded input, i.e., their most basic ingredients are nogoods over ground literals. Alpha provides this by having a grounder component responsible for generating ground nogoods, and a solving component executing the CDNL-search. This separation is common for CDNL-based ASP solvers but differently from these, the grounding component in Alpha is not just used once but it is called every time after the solver has finished deriving new truth assignments. Hence, there is a cyclic interplay between the solving component and the grounding component. Different from CDNL-solving, however, the result of this interplay is a computation sequence in the style of lazy grounding. Most importantly, the solver does not guess on each atom whether it is *true* or *false*, but it guesses on ground instances of rules whether they fire or not.

Taking inspiration from other ASP solvers (e.g. DLV [14], Asperix, or Omiga), Alpha introduces *must-be-true* as a possible truth value for atoms. If an atom a is *must-be-true*, denoted $\mathbf{M}a$, then a must be true but there currently is no rule firing with a in its head. This allows to distinguish the case that a is true due to a constraint, from the case that a is derived because a rule fired. In Alpha an atom a therefore may be assigned one of *must-be-true*, *true*, or *false*, denoted by $\mathbf{M}$, $\mathbf{T}$, or $\mathbf{F}$, respectively. Nogoods however, are still over Boolean signed literals, i.e., only literals of form $\mathbf{T}a$ and $\mathbf{F}a$ occur in nogoods. For convenience, the complement of such a literal s is denoted by $\overline{s}$, formally $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. In order to capture propagation to $\mathbf{T}a$, nogoods may have a specifically indicated head. Usually a nogood propagates to $\mathbf{M}a$ or $\mathbf{F}a$, but a nogood with head may propagate to $\mathbf{T}a$ if certain conditions are met. Formally:

Definition 1. *A* nogood with head *is a nogood* $ng = \{s_1, \ldots, s_n\}$ *with one distinguished negative literal* s_i *such that* $s_i = \mathbf{F}a$ *for some* $a \in \mathcal{A}$ *and* $1 \leq i \leq n$. *A nogood with head is written as* $\{s_1, \ldots, s_n\}_i$ *where* i *is the index of the head; we write* $hd(ng)$ *to denote the head of* ng, *i.e.,* $hd(ng) = s_i$.

In the remainder of this work, the term nogood may denote a nogood with head or an ordinary nogood (without head). Next, we formally define assignments over $\mathbf{M}$, $\mathbf{T}$, and $\mathbf{F}$.

Definition 2. *An* Alpha assignment*, in the following just* assignment*, is a sequence* $A = (s_1, \ldots, s_n)$ *of signed literals* s_i, $1 \leq i \leq n$, *over* $\mathbf{T}$, $\mathbf{M}$, *and* $\mathbf{F}$.

In slight abuse of notation, we consider an assignment A to be a set, i.e., $s \in A$ for $A = (s_1, \ldots, s_n)$ holds if there exists $1 \leq i \leq n$ such that $s = s_i$. Furthermore, let $A^{\mathcal{B}}$ denote the *Boolean projection* of A defined as:

$$A^{\mathcal{B}} = \{\mathbf{T}a \mid \mathbf{T}a \in A \text{ or } \mathbf{M}a \in A\} \cup \{\mathbf{F}a \mid \mathbf{F}a \in A\}.$$

An assignment A is *consistent* if for every atom $a \in \mathcal{A}$ the following holds:

(i) $\mathbf{T}a \in A^{\mathcal{B}}$ implies $\mathbf{F}a \notin A$ and $\mathbf{F}a \in A$ implies $\mathbf{T}a \notin A^{\mathcal{B}}$ (*false* and *true* exclude each other),
(ii) if $\mathbf{M}a \in A$ and $\mathbf{T}a \in A$ with $\mathbf{M}a = s_i$ and $\mathbf{T}a = s_j$ for $1 \leq i, j \leq n$, then $i < j$ (*must-be-true* precedes *true*), and
(iii) if $\mathbf{X}a \in A$ for any $\mathbf{X} \in \{\mathbf{M}, \mathbf{T}, \mathbf{F}\}$ then there exists exactly one $1 \leq i \leq n$ with $\mathbf{X}a = s_i$ (every occurrence is unique).

Due to the nature of lazy-grounding, A needs not be complete with respect to $\mathcal{A}$, i.e., there may be $a \in \mathcal{A}$ such that $\mathbf{X}a \notin A$ holds for any $\mathbf{X} \in \{\mathbf{T}, \mathbf{F}, \mathbf{M}\}$. In the remainder of this work only consistent assignments are considered.

Example 1. Consider the assignment $A = (\mathbf{F}a, \mathbf{M}b, \mathbf{M}c, \mathbf{T}b)$. A is consistent and $A^{\mathcal{B}} = \{\mathbf{F}a, \mathbf{T}b, \mathbf{T}c\}$. The assignment $B = (\mathbf{M}b, \mathbf{F}b, \mathbf{T}a, \mathbf{M}a, \mathbf{F}b)$, however, is inconsistent, because it violates every single condition for consistency.

In order to establish unit-propagation on nogoods which propagates to *true*, *false*, or *must-be-true*, we use two different notions of being unit.

Definition 3. *Given a nogood* $ng = \{s_1, \ldots, s_n\}$ *and an assignment* A*:*

- ng *is* weakly-unit *under* A *for* s *if* $ng \setminus A^{\mathcal{B}} = \{s\}$ *and* $\overline{s} \notin A^{\mathcal{B}}$,
- ng *is* strongly-unit *under* A *for* s *if* ng *is a nogood with head,* $ng \setminus A = \{s\}$, $s = hd(ng)$, *and* $\overline{s} \notin A$.

If a nogood is strongly-unit, it also is weakly-unit but not the other way round.

Let ng be a nogood and A be an assignment, ng is *satisfied* by A if there exists $s \in ng$ with $\overline{s} \in A^{\mathcal{B}}$ and additionally for ng being a nogood with head $s = hd(ng)$, it holds that ng being strongly-unit under $A \setminus \{\overline{s}\}$ implies that $\overline{s} \in A$. The nogood ng is *violated* by the assignment A if $ng \subseteq A^{\mathcal{B}}$.

Definition 4. *An assignment* A *is a* solution *to a set* Δ *of nogoods if for every* $ng \in \Delta$ *it holds that* ng *is satisfied by* A.

Example 2. Let $n_1 = \{\mathbf{F}a, \mathbf{T}b, \mathbf{F}c\}_1$ and $n_2 = \{\mathbf{F}c, \mathbf{T}d\}$ be nogoods and $A = (\mathbf{M}b, \mathbf{F}c, \mathbf{F}d)$, $A' = (\mathbf{T}b, \mathbf{F}c, \mathbf{T}d)$ be assignments. Then, n_1 is weakly-unit under A but not strongly-unit, intuitively because b is assigned *must-be-true* and not *true*. However, n_1 is strongly-unit under A'. The nogood n_2 is satisfied by A while it is violated by A'.

In order to represent bodies of whole ground rules, Alpha introduces atoms representing such, similar as Clasp. Given a rule r and a variable substitution σ, the atom representing the body of $r\sigma$ is denoted by $\beta(r, \sigma)$. For convenience, we assume $\beta(r, \sigma) \in \mathcal{A}$ and β not occurring in any program, i.e., $\beta(r, \sigma)$ is a fresh and unique ground atom.

Definition 5. *Given a rule r and a substitution σ such that $r\sigma$ is ground, let $B^+(r\sigma) = \{a_1, \ldots, a_k\}$, $B^-(r\sigma) = \{a_{k+1}, \ldots, a_n\}$ and $H(r) = \{a_0\}$, the* nogood representation *$ng(r)$ of r is the following set of nogoods:*

$$\begin{aligned} ng(r) = \big\{ &\{\mathbf{F}\beta(r,\sigma), \mathbf{T}a_1, \ldots, \mathbf{T}a_k, \mathbf{F}a_{k+1}, \ldots, \mathbf{F}a_n\}_1, \{\mathbf{F}a_0, \mathbf{T}\beta(r,\sigma)\}_1, \\ &\{\mathbf{T}\beta(r,\sigma), \mathbf{F}a_1\}, \ldots, \{\mathbf{T}\beta(r,\sigma), \mathbf{F}a_k\} \\ &\{\mathbf{T}\beta(r,\sigma), \mathbf{T}a_{k+1}\}, \ldots, \{\mathbf{T}\beta(r,\sigma), \mathbf{T}a_n\}\big\} \end{aligned}$$

Intuitively, the first nogood of the nogood representation $ng(r)$ expresses that whenever the body of $r\sigma$ holds, so does the atom representing the body, the second nogood expresses that whenever this atom is *true* then the head also is *true*. Note that both nogoods have heads, indicating that $\beta(r, \sigma)$ for the first and a_0 for the second nogood may be assigned to $\mathbf{T}$ upon unit-propagation. The nogoods of the last two rows express that $\beta(r, \sigma)$ is *true* only if the body of $r\sigma$ is satisfied. It is easy to see that $\beta(r, \sigma)$ is *true* iff the body of $r\sigma$ holds, formally:

Proposition 1. *Let r be a rule, σ be a substitution such that $r\sigma$ is ground, and let A be a consistent assignment such that A is a solution to $ng(r\sigma)$. Then, $\mathbf{T}\beta(r, \sigma) \in A$ iff $B^+(r\sigma) \subset A \wedge B^-(r\sigma) \cap A = \emptyset$.*

Note that $ng(r\sigma)$ contains no nogoods establishing support, because it is in general hard to determine the set of all ground rules having the same head; e.g. in $p(X) \leftarrow q(X, Y)$. the head $p(a)$ may be derived by many ground rules.

The following proposition shows that the nogood representation is correct, i.e., if an assignment, free of *must-be-true*, is a solution to $ng(r\sigma)$ then the corresponding Boolean interpretation satisfies the ground rule $r\sigma$.

Proposition 2. *Given a consistent assignment A with $A = A^{\mathcal{B}}$, a rule r, and a substitution σ s.t. $r\sigma$ is ground. Then, $A \models r\sigma$ iff A is a solution to $ng(r\sigma)$.*

Since the solver component of Alpha does not guess on every atom, but only on those representing applicable ground rules, i.e., it guesses on $\beta(r, \sigma)$ if $r\sigma$ is applicable under the current assignment, additional nogoods are added for the solver to detect this. Recall that a rule is applicable, if its positive body holds and its negative body is not contradicted, formally: $B^+(r\sigma) \subseteq A$ and $B^-(r\sigma) \cap A = \emptyset$. For a rule r' without a negative body, i.e., $B^-(r') = \emptyset$, no additional guessing

is necessary, because r' is applicable only if r' fires, hence the rule cannot be guessed to not fire. Consequently, no choices need to be done for rules without negative body. For rules that may possibly serve as a choice point to guess on, the following nogoods are added.

Definition 6. *Given a ground rule r with $B^-(r) \neq \emptyset$, and a substitution σ such that $r\sigma$ is ground. Let $B^+(r\sigma) = \{a_1, \ldots, a_k\}$, $B^-(r\sigma) = \{a_{k+1}, \ldots, a_n\}$, and let $cOn(r,\sigma)$ and $cOff(r,\sigma)$ be fresh ground atoms not occurring in any input program. Then, the set $ng_{ch}(r\sigma)$ of* choice nogoods *is:*

$$ng_{ch}(r\sigma) = \big\{\{\mathbf{F}\,cOn(r,\sigma), \mathbf{T}\,a_1, \ldots, \mathbf{T}\,a_k\}_1, \\ \{\mathbf{F}\,cOff(r,\sigma), \mathbf{T}\,a_{k+1}\}, \ldots, \{\mathbf{F}\,cOff(r,\sigma), \mathbf{T}\,a_n\}\big\}$$

The choice nogoods encode applicability of the ground rule $r\sigma$ as follows: if the positive body holds, $cOn(r,\sigma)$ becomes *true* due to the first nogood, and if one of the negative body atoms becomes *true* or *must-be-true*, i.e., the rule is no longer applicable, then $cOff(r,\sigma)$ becomes *true* or *must-be-true*. The solver can then identify valid choices, i.e., applicable ground rules, by checking that for $\beta(r,\sigma)$, $cOn(r,\sigma)$ is *true* and $cOff(r,\sigma)$ does not hold (is *false* or unassigned).

Proposition 3. *Let r be a rule with $B^-(r) \neq \emptyset$, σ be a substitution such that $r\sigma$ is ground, and let A be an assignment such that A is a solution to $ng_{ch}(r,\sigma)$. Then, $r\sigma$ is applicable under $\{a \in \mathcal{A} \mid \mathbf{T}a \in A\}$ iff $\mathbf{T}\,cOn(r,\sigma) \in A$, $\mathbf{M}\,cOff(r,\sigma) \notin A$, and $\mathbf{T}\,cOff(r,\sigma) \notin A$.*

Example 3. Consider the rule $r = p(X) \leftarrow q(X), not\ s(X).$ with substitution $\sigma : X \mapsto d$. Then, $ng_{ch}(r,\sigma) = \big\{\{\mathbf{F}\,cOn(r,\sigma), \mathbf{T}q(d)\}_1, \{\mathbf{F}\,cOff(r,\sigma), \mathbf{T}s(d)\}\big\}$. The assignment $A = (\mathbf{T}q(d))$ is no solution to $ng_{ch}(r,\sigma)$ but its extension $A' = (\mathbf{T}q(d), \mathbf{T}\,cOn(r,\sigma), \mathbf{F}s(d))$ is and $r\sigma$ is applicable in A', which is also reflected by $\mathbf{T}\,cOn(r,\sigma) \in A'$, $\mathbf{M}\,cOff(r,\sigma) \notin A'$, and $\mathbf{T}\,cOff(r,\sigma) \notin A'$ all holding.

4 Loosely Coupled Lazy-Grounding

How the lazy grounder generates the nogoods presented in the previous section is shown next. The main task of the lazy grounder is to construct substitutions σ for all rules $r \in P$. Necessary are not all possible substitutions, but only those that yield ground rules potentially applicable under the current assignment.

Definition 7. *Given an assignment A, a substitution σ for a rule $r \in P$ with $r\sigma$ being ground* is of interest *if $B^+(r\sigma) \subseteq A^{\mathcal{B}}$.*

It is easy to see that substitutions not of interest can be ignored for the computation of answer-sets, since they do not fire and do not lead to any guesses:

Proposition 4. *Given an assignment A, a rule $r \in P$, and a substitution σ not of interest wrt. A. Then, $r\sigma$ is not applicable and it does not fire under $A^{\mathcal{B}}$.*

The lazy grounder therefore only needs to find substitutions of interest.

Algorithm 1: *lazyGround*

Input: A program P, an assignment A, and a state S.
Output: A set Δ of ground nogoods for substitutions of interest wrt. A.

$A_{new} \leftarrow \{a \mid \mathbf{T}a \in A^{\mathcal{B}}\} \setminus S$.
$S \leftarrow S \cup A_{new}$
$R_{subst} \leftarrow \{r\sigma \mid r \in P, r\sigma \text{ is ground}, B^+(r\sigma) \subseteq S, \text{ and } B^+(r\sigma) \cap A_{new} \neq \emptyset\}$ **(a)**
foreach $r\sigma \in R_{subst}$ **do**
  $\Delta \leftarrow \Delta \cup ng(r\sigma) \cup ng_{ch}(r\sigma)$ **(b)**
return Δ

Definition 8. *Let P be a (non-ground) program and A be an assignment, the* lazy-grounding *wrt. A, denoted by $lgr_P(A)$, is the set of all ground rules that are of interest wrt. A. Formally, $lgr_P(A) = \big\{r\sigma \mid r \in P, B^+(r\sigma) \subseteq \{a \mid \mathbf{T}a \in A^{\mathcal{B}}\}\big\}$.*

Since each rule $r \in P$ is safe, $B^+(r\sigma) \subseteq \{a \mid \mathbf{T}a \in A^{\mathcal{B}}\}$ implies that $B^-(r\sigma)$ is ground. Consequently, all $r \in lgr_P(A)$ are ground.

Example 4. Consider the program $P = \{b \leftarrow not\, a. \quad p(X) \leftarrow q(X), not\, a.\}$ and the assignment $A = \big(\mathbf{M}q(c), \mathbf{F}q(d), \mathbf{T}q(e)\big)$. The lazy-grounding wrt. A is $lgr_P(A) = \big\{b \leftarrow not\, a. \quad p(c) \leftarrow q(c), not\, a. \quad p(e) \leftarrow q(e), not\, a.\big\}$.

In Alpha the grounder is only loosely coupled with the state of the solver, i.e., the grounder returns all nogoods of interest, but it may report more nogoods. Although this looks inefficient at first, it can avoid a lot of re-grounding. In other lazy-grounding approaches the grounder is strongly bound to the current branch of the search and whenever the solver changes to another branch, all previously grounded rules are eliminated. In Alpha the lazy-grounder does not need to eliminate ground instances and therefore can re-use those ground instances for any future branch explored by the solver. Accumulating already derived ground substitutions therefore leads to an increasing number of substitutions of interest that need not be computed again. In the extreme case, the complete grounding is derived early on, leaving no work for the lazy-grounder.

In order to construct $lgr_P(A)$, the lazy-grounder maintains a state $S \subseteq \mathcal{A}_{grd}$, which is an accumulator of positive assignments previously seen. Substitutions of interest, which have not been computed before, yield at least one positive ground atom contained in $A \setminus S$, i.e., with increasing S the potentially relevant substitutions decrease. The lazy-grounding procedure is given by Algorithm 1:

- In (a) all relevant substitutions are computed. The restriction to substitutions $r\sigma$ with $B^+(r\sigma) \cap A_{new} \neq \emptyset$ enables the grounder to consider only rules that are relevant and this is a starting point for the construction of σ.
- In (b) the corresponding nogoods to be returned in the last step are generated. A semi-naive grounding algorithm is used here in the implementation.

Proposition 5. *For a program P and an assignment A, Algorithm 1 computes all ground rules of interest wrt. A, i.e., $lazyGround_P(A) = lgr_P(A)$ for a state $S = \emptyset$. Moreover, consecutive calls are such that: $\bigcup lazyGround_P(A) \subseteq lgr_P(A)$.*

Note that the accumulation of ground substitutions may exhaust all available memory. In such a situation, Alpha may simply forget all derived substitutions, except those in the current branch of the search space. Likewise, the solver can also forget previously obtained nogoods and thus free memory again.

5 The Alpha Approach

Based on the concepts above, Alpha blends lazy-grounding with CDNL search. At a glance, Alpha is a modified CDNL search algorithm using the lazy-grounder of the previous section and assignments with *must-be-true* in combination with nogoods with heads. The main algorithm is given by Algorithm 2, like for CDNL-based solvers it is comprised of one big loop where the search space is explored.

At a glance, the algorithm is as follows. After initialization, nogoods resulting from facts are requested from the grounder in (a) and the main loop is entered:

- First, unit propagation is applied to the known nogoods potentially extending the current assignment in (b) and then each iteration does one of:
- analyzing and learning from a conflict in (c),
- querying the lazy-grounding component for more nogoods in (d),
- introducing a choice in (e),
- assigning all unassigned atoms to false after the cycle of propagation, choice, and lazy-grounding reached a fixpoint in (f),
- reporting a found answer-set if it is free of *must-be-true* in (g), and finally
- backtracking if some assignment to *must-be-true* remained and the search branch thus yields no answer-set in (h).

For lazy-grounding, the function *lazyGround* of the previous section is used; for readability, however, the state S is assumed to be implicitly given. The *propagate* function applies unit-resolution to compute all inferences from the current assignment A and the set of nogoods Δ. Formally, let the immediate unit-propagation be given by:

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

Then, $propagate(\Delta, A) = (A', d)$ where $A' = lfp\big(\Gamma_\Delta(A)\big)$ is the least fixpoint of the immediate unit-propagation and $d = 1$ if $A' \neq A$ and $d = 0$ otherwise indicates whether some new assignment has been derived by the propagation. For space reasons, the algorithm computing $propagate(\Delta, A)$ is omitted.

In (c) *AlphaASP* checks for a conflict caused by some nogood δ being violated in A. If so, $analyze(\delta, \Delta, A)$ analyzes the conflict employing ordinary conflict-driven nogood learning following the first-UIP schema. On an abstract level, the process applies resolution on certain nogoods from Δ, starting at δ and resolving them until a stop criterion is reached. The nogoods to resolve are obtained from the so-called implication graph, storing information about which assignments in

Algorithm 2: *AlphaASP*

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

```
AS ← ∅                                   // Set of found answer-sets.
A ← ∅                                    // Current assignment.
Δ ← ∅                                    // Set of nogoods.
dl ← 0                                   // Current decision level.
exhausted ← 0                            // Stop condition.
Δ ← lazyGround_P(A)                                                  (a)
while exhausted = 0 do
    (A', didPropagate) ← propagate(Δ, A)                             (b)
    A ← A'
    if ∃δ ∈ Δ : δ ⊆ A then                                           (c)
        (δ_l, dl_bj) ← analyze(δ, Δ, A)
        A ← backjump(dl_bj)
        dl ← dl_bj
        Δ ← Δ ∪ {δ_l}
    else if didPropagate = 1 then                                    (d)
        Δ ← Δ ∪ lazyGround_P(A)
    else if acp(Δ, A) ≠ ∅ then                                       (e)
        dl ← dl + 1
        s ← select(acp(Δ, A))
        A ← A ∪ {s}
    else if atoms(Δ) \ atoms(A) ≠ ∅ then                             (f)
        A ← A ∪ {Fa | a ∈ atoms(Δ) \ atoms(A)}
    else if {Ma ∈ A} = ∅ then                                        (g)
        AS ← AS ∪ {A}
        δ_enum ← enumNg(A)
        Δ ← Δ ∪ {δ_enum}
        A ← backtrack()
        dl ← dl − 1
    else                                                             (h)
        A ← backtrack()
        dl ← dl − 1
        if dl = 0 then
            exhausted ← 1
return AS
```

A stem from which nogood by unit-resolution done during an earlier propagation. The so-called first-UIP is used to stop, which is a certain graph dominator in the implication graph. First-UIP nogood learning is a well-established technique used here basically unaltered, for details see [9]. Backjumping is done by $backjump(dl_{bj})$ which removes from A all entries of a decision $dl > dl_{bj}$ higher than the decision level to backjump to and returns the resulting assignment; entries of the decision level dl_{bj} are kept.

In (d) new nogoods are requested from the grounder using *lazyGround*, given that the previous propagation derived new assignments. In the implementation, only the recently added assignments of A have to be transferred to the lazy-grounding component, since it has its internal state S and reports only those nogoods from new ground instances of a rule.

In (e) guessing takes place. For this, a list of active choice points is maintained, where each such choice point corresponds to an applicable rule. Formally, let the *atoms occurring* in a set Δ of nogoods be given by $atoms(\Delta) = \{a \mid \mathbf{X}a \in \delta, \delta \in \Delta, \mathbf{X} \in \{\mathbf{T}, \mathbf{F}\}\}$ and let the atoms occurring in an assignment A be $atoms(A) = \{a \mid \mathbf{X}a \in A, \mathbf{X} \in \{\mathbf{T}, \mathbf{F}, \mathbf{M}\}\}$. The set of *active choice points* wrt. a set of nogoods Δ and an assignment A is: $acp(\Delta, A) = \{\beta(r,\sigma) \in atoms(\Delta) \mid \mathbf{T}cOn(r,\sigma) \in A \wedge \mathbf{T}cOff(r,\sigma) \notin A \wedge \mathbf{M}cOff(r,\sigma) \notin A\}$. The *select* function simply takes an element from the set of active choice points. Heuristics can be employed to select a good guessing candidate.

When (f) is reached, the interplay between propagation, grounding, and guessing has reached a fixpoint, i.e., there are no more applicable ground rule instances and nothing can be derived by propagation or from further grounding. Since the guessing does not guess on all atoms, there may be atoms in A not having an assigned truth value. Due to the fixpoint, these atoms must be *false* and the algorithm assigns all to *false*. The propagation at the following iteration then guarantees that (potentially) violated nogoods are detected.

In (g) the solver checks whether there is an atom assigned to *must-be-true* but could not be derived to be *true*, i.e., no rule with the atom as its head fired. If there is an atom assigned *must-be-true*, then the assignment A is no answer set. If no such atom exists, then A is an answer-set and it is recorded. Furthermore, the function $enumNg(A)$ generates a nogood that excludes the current assignment. This is necessary to avoid finding the same answer-set twice. Following [8], this enumeration nogood only needs to contain the decision literals of A, formally:

$$enumNg(A) = \{\mathbf{X}a \in A \mid a = \beta(r,\sigma) \text{ for some } r, \sigma, \text{ and } \mathbf{X} \in \{\mathbf{T}, \mathbf{F}\}\}$$

In (h) backtracking is done by *backtrack*, which removes from the assignment A all entries done in the current decision level and returns an assignment A' containing only elements of A that have not been assigned in the last decision level. For readability, the details of this are not made explicit (i.e., the assignment associating to each literal a decision level on which it was assigned). If the current decision level dl reaches 0, the search space exhausted.

The *AlphaASP* algorithm computes the answer-sets of normal logic programs.

Theorem 1. *Given a normal logic program P, then $AlphaASP(P) = AS(P)$.*

Proof. Ignoring branches not leading to an answer-set, *AlphaASP* constructs a computation sequence $A_s = (A_0, \ldots, A_\infty)$ where the assignment A in *AlphaASP* at the i-th iteration of the main loop corresponds to the Boolean assignment A_i in A_s by $A_i = A \setminus \{\mathbf{M}a \mid a \in \mathcal{A}\}$, i.e., assignments to *must-be-true* are removed. First observe that at the last step of *AlphaASP* the assignment A is free of *must-be-true* because otherwise the algorithm would have backtracked in (g),

i.e., $A_\infty = A$. The sequence $(A_0, \ldots, A_\infty)$ is a computation, because: (1) by *propagate* and *acp* only containing consequences, propagation and guessing only add consequences, hence $\forall i \geq 1 : A_i \subseteq T_P(A_{i-1})$. (2) the current assignment only increases, i.e., $\forall i \geq 1 : A_{i-1} \subseteq A_i$. (3) due to the main loop only stopping after the search space is completely explored, the computation converges, i.e., $A_\infty = \bigcup_{i=0}^{\infty} A_i = T_P(A_\infty)$. (4) To show that $\forall i \geq 1 : \forall at \in A_i \setminus A_{i-1}, \exists r \in P$ such that $H(r) = at$ and $\forall j \geq i-1 : B^+(r) \subseteq A_j \wedge B^-(r) \cap A_j = \emptyset$ holds, observe that: (a) Guessing is solely on auxiliary atoms $\beta(r, \sigma)$ s.t. $\beta(r, \sigma) \in acp(\Delta, A)$ holds, i.e., $\mathbf{T}cOn(r, \sigma) \in A$ and by $ng_{ch}(r\sigma)$ it holds for each $a \in B^+(r\sigma)$ that $\mathbf{T}a \in A$, i.e., if $\beta(r, \sigma)$ is an active choice point then its positive body is wholly *true*. Additionally, nogoods in $ng(r\sigma)$ ensure that all atoms in $B^-(r\sigma)$ are *false* once $\beta(r, \sigma)$ is *true*. (b) Non-auxiliary atoms a become *true* only by *propagate* if there is a nogood δ strongly-unit, i.e., δ has head atom a and all atoms positively occurring in δ are *true* in A. For each ground $r\sigma$, there is exactly one nogood with head in $ng(r\sigma)$, thus $H(r\sigma)$ is *true* iff $r\sigma$ fires under A. In summary, *AlphaASP* is sound. Completeness follows from *AlphaASP* exploring the whole search space except for those parts excluded by learned nogoods; since *analyze* learns by resolution, completeness follows. Consequently, $AlphaASP(P) = AS(P)$.

6 Discussion

The Alpha approach is implemented in the Alpha system, which is freely available at: https://github.com/alpha-asp. Early benchmarks showed a much improved performance on non-trivial search problems (e.g. graph 3-colorability) compared to other lazy-grounding ASP systems. Detailed benchmark results can be found at www.kr.tuwien.ac.at/research/systems/alpha. In Alpha, support nogoods (constraints) are not added in general, because they are not necessary and constructing them is hard for rules like $p(X) \leftarrow q(X, Y)$ which may have infinitely many ground instances all deriving the same ground atom $p(a)$. The solver may now guess that all rules with $p(a)$ in the head do not fire while also deriving that $p(a)$ *must-be-true*, which cannot yield an answer-set. The solver detects this but only after all atoms are assigned. Eventual unrelated guesses then make the search exponential where for traditional ASP systems it is not. Alpha mitigates this by adding support nogoods for a very limited class of atoms.

Similar to Alpha, in [5] a computation sequence is constructed using nogoods and conflict-driven learning. The ASP program, however, is grounded upfront, hence traditional techniques for conflict-driven ASP solving can be used directly. Integrating lazy-grounding into [5] very likely necessitates similar techniques as presented here, because the established CDNL techniques require support nogoods and loop nogoods, which are in general hard to construct (see above).

7 Conclusion

This work presented the Alpha approach that combines lazy-grounding with CDNL-based search techniques to create a novel ASP solver that avoids the

grounding bottleneck and provides good search performance. For that, nogoods with heads and assignments with *must-be-true* are introduced such that the *AlphaASP* algorithm constructs answer-sets similar to other lazy-grounding systems, i.e., it guesses whether a rule fires and not whether an atom is *true*. *AlphaASP* makes use of a number of techniques for efficient answer-set solving: conflict-driven nogood learning following the first-UIP schema, backjumping, and heuristics. The Alpha approach is implemented in the freely available Alpha system. Early benchmarks show it avoids the grounding bottleneck and provides good search performance. Topics for future work are forgetting of nogoods and a lifting of learning to the first-order case akin to [17].

References

1. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A native ASP solver based on constraint learning. In: LPNMR. pp. 54–66. Springer (2013)
2. Balduccini, M., Lierler, Y., Schüller, P.: Prolog and ASP inference under one roof. In: LPNMR. LNCS, vol. 8148, pp. 148–160. Springer (2013)
3. Calimeri, F., Fuscà, D., Perri, S., Zangari, J.: *I* -dlv: The new intelligent grounder of dlv. In: AI*IA. pp. 192–207 (2016)
4. Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., Weinzierl, A.: Omiga : An open minded grounding on-the-fly answer set solver. In: JELIA. pp. 480–483 (2012)
5. Dovier, A., Formisano, A., Pontelli, E., Vella, F.: A GPU implementation of the ASP computation. In: PADL. LNCS, vol. 9585, pp. 30–47. Springer (2016)
6. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an incremental ASP solver. In: ICLP. LNCS, vol. 5366, pp. 190–205. Springer (2008)
7. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: *clasp* : A conflict-driven answer set solver. In: LPNMR. pp. 260–265 (2007)
8. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. In: LPNMR 2007. LNCS, vol. 4483, pp. 136–148. Springer (2007)
9. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. 187, 52–89 (2012)
10. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: ICLP. LNCS, vol. 5649, pp. 235–249. Springer (2009)
11. Lefèvre, C., Beatrix, C., Stephan, I., Garcia, L.: Asperix, a first-order forward chaining approach for answer set computing. TPLP p. 145 (Jan 2017)
12. Lefèvre, C., Nicolas, P.: A first order forward chaining approach for answer set computing. In: LPNMR. LNCS, vol. 5753, pp. 196–208 (2009)
13. Lefèvre, C., Nicolas, P.: The first version of a new ASP solver : Asperix. In: LPNMR. LNCS, vol. 5753, pp. 522–527 (2009)
14. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7, 499–562 (2002)
15. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: Gasp: Answer set programming with lazy grounding. Fundam. Inform. 96(3), 297–322 (2009)
16. Teppan, E.C., Friedrich, G.: Heuristic constraint answer set programming. In: ECAI. FAIA, vol. 285, pp. 1692–1693. IOS Press (2016)
17. Weinzierl, A.: Learning non-ground rules for answer-set solving. In: Grounding and Transformation for Theories with Variables. pp. 25–37 (2013)