Techniques for Efficient Lazy-Grounding ASP Solving*

Lorenz Leutgeb[1] and Antonius Weinzierl[1,2]

[1] Knowledge-Based Systems Group
Institute of Information Systems, TU Wien
Vienna, Austria
[2] Helsinki Institute for Information Technology HIIT
Department of Computer Science, Aalto University
Espoo, Finland
lorenz@leutgeb.xyz weinzierl@kr.tuwien.ac.at

Abstract. Answer-Set Programming (ASP) is a well-known and expressive logic programming paradigm based on efficient solvers. State-of-the-art ASP solvers require the ASP program to be variable-free, they thus ground the program upfront at the cost of a potential exponential explosion of the space required. Lazy-grounding, where solving and grounding are interleaved, circumvents this grounding bottleneck, but the resulting solvers lack many important search techniques and optimizations. The recently introduced ASP solver Alpha combines lazy-grounding with conflict-driven nogood learning (CDNL), a core technique of efficient ASP solving. This work presents how techniques for efficient propagation can be lifted to the lazy-grounding setting. The Alpha solver and its components are presented and detailed benchmarks comparing Alpha to other ASP solvers demonstrate the feasibility of this approach.

1 Introduction

Answer-Set Programming (ASP) is an expressive logic programming paradigm where non-monotonic rules are used to formalize problem descriptions. The semantics of such rules are given in terms of answer sets, which represent solutions to the specified problem (see [4] for a detailed introduction). For example the following rules encode that for nodes N of a graph a color C may be chosen.

$$chosenColor(N, C) \leftarrow node(N), color(C), not\ notChosenColor(N, C).$$
$$notChosenColor(N, C) \leftarrow node(N), color(C), not\ chosenColor(N, C).$$

Rules allow to easily encode complex problems like graph coloring. Finding the answers to such a problem, however, is hard and requires advanced techniques.

ASP solvers are traditionally based on a two-phase computation. First, the variables are removed from the input program by grounding and second, the

* This work has been supported by the Austrian Science Fund (FWF) project P27730 and the Academy of Finland, project 251170.

ground program is solved by highly optimized algorithms for propositional problems. Prominent such ground-and-solve systems are DLV [11] and Clingo [5]. The ground program, however, is in the worst case exponential in the size of the non-ground program. This makes many real-world programs simply too big to fit in memory and therefore referred to as the *grounding bottleneck* of ASP.

Lazy-grounding on the other hand interleaves the grounding and solving phases and thus overcomes the grounding bottleneck (cf. GASP [13], Asperix [10], and Omiga [3]). Due to this interleaving, such solvers explore the (exponential) search space fundamentally different from CDNL-based solvers, making them very inefficient at solving problems that are trivial for ground-and-solve ASP systems. The Lazy-MX system [2] for the language of $FO(ID)$ follows a different approach and achieves lazy-grounding with efficient solving, but it is restricted to (some) subclass of ASP and requires manual translation.

The recently introduced ASP solver Alpha, however, combines CDNL-based search procedures with lazy-grounding to get the best of both worlds: fast search space exploration and avoidance of the grounding bottleneck at the same time.

Example 1. Consider the following program which selects from a domain exactly one element:

$$dom(1). \quad \ldots \quad dom(12). \qquad sel(X) \leftarrow dom(X), not\ nsel(X).$$
$$\leftarrow sel(Y), sel(X), X \neq Y. \qquad nsel(X) \leftarrow dom(X), not\ sel(X).$$

Adding to this program one rule that forms a large cross-product over selected elements, already exhibits the grounding bottleneck.

$$p(X_1, X_2, X_3, X_4, X_5, X_6) \leftarrow sel(X_1), sel(X_2), sel(X_3), sel(X_4), sel(X_5), sel(X_6).$$

For solvers like Clingo, the amount of required memory increases dramatically when domain elements are added to *dom*. A domain size of 20 already requires several gigabytes of memory to ground, while the same program can be solved by lazy-grounding almost immediately and without such memory consumptions.

Blending lazy-grounding and CDNL solving is challenging for a number of reasons. First, usual CDNL solvers guess truth assignments for atoms while lazy-grounding solvers guess whether rules satisfying certain conditions fire or not. Second, atoms may only become *true* due to a rule that fires and must not become *true* due to constraints, since e.g. the constraint $\leftarrow$ *not a*. is no justification for a being *true*. Therefore unit-propagation on nogoods may not simply set atoms to *true*. Introducing *must-be-true* as a third truth value fixes this problem, but requires intricate adaptions on the data structures for unit-propagation. Specifically, the 2-watched-literals schema for nogoods is no longer sufficient. A solution to the first challenge is described in detail in [15]. Here, we provide an overview to that solution and are otherwise concerned with the second challenge.

The contributions (after preliminary Section 2) of this work are as follows:

– presenting the novel architecture of the Alpha ASP solver (in Section 3) followed by an overview of the Alpha approach for blending lazy-grounding and CDNL-based search,
– an enhancement of the two-watched literals schema to obtain efficient propagation performance in the presence of a third truth value and nogoods that are extended with heads (in Section 4), and
– benchmarks of the resulting ASP solver Alpha (in Section 5), showing impressive improvements but also directions for future work (in Section 6).

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 $vars(at)$. 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 $vars(r) = \bigcup_{at \in H(r) \cup B(r)} vars(at)$. 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., $vars(r) \subseteq \bigcup_{a \in B^+(r)} vars(a)$.

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 $grd(r) = \{r\sigma \mid \sigma \text{ is a substitution for all } v \in vars(r)\}$ 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. Following established notation, a Boolean signed literal is 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. For example, the nogood $ng = \{\mathbf{T}a, \mathbf{F}b\}$ states that it cannot be the case that a is true and b is false at the same time. 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 [5–7]. The complement of a Boolean signed literal s, denoted $\overline{s}$, is $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. 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 heads of applicable rules (cf. [13, 8, 3]). 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 $B^-(r) \cap A_k = \emptyset$. Observe that finding applicable ground rules, i.e., finding a non-ground rule r and a grounding substitution σ such that $r\sigma$ is applicable, is the task of the (lazy) grounder. A computation $(A_0, \ldots, A_\infty)$ then 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 $A = A_\infty$ [9, 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 The Alpha Approach

Alpha is a combination of lazy-grounding and CDNL-search to obtain an ASP solver that avoids the grounding bottleneck and shows good search performance.

Architecture. On an abstract level, Alpha achieves this by utilizing a grounder component and a solver component, where the solver is a modified CDNL-search

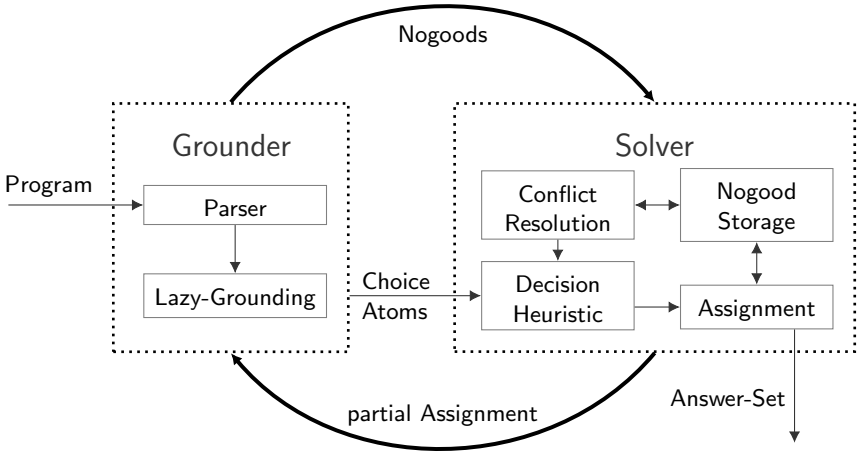

Fig. 1. Architecture of the Alpha system. Data flow is indicated by arrows. Grounder (left) and CDNL-based solver (right) interact cyclically for lazy-grounding.

algorithm, but both components interact cyclically in the style of lazy-grounding ASP systems. The architecture of the Alpha solver is depicted in Figure 1. The grounder is composed of a parser and a semi-naive grounder that, given a partial assignment, computes all ground rules that potentially fire, transforms the ground rules into nogoods, and sends these to the solver. The solving component is a modified CDNL solver trying to find a satisfying assignment to the set of nogoods presented. It contains a nogood store for unit-propagation on nogoods, conflict resolution implementing conflict-driven nogood learning following the first-UIP schema to learn new nogoods, and a decision heuristic. The most important difference to an ordinary CDNL solver is that guessing is restricted to atoms representing applicable ground rules, i.e., rules whose positive body is satisfied in the current assignment. By that, Alpha prevents unfounded sets from becoming *true*, thus the assignments constructed by the solver are guaranteed to be unfounded-free. Another difference is that the partial assignments of Alpha contain truth values *true*, *false*, and *must-be-true* where the latter indicates that an atom must be true (e.g. due to a constraint) but no firing rule derives/justifies the atom yet.

Core Algorithm. The remainder of this section provides a summary of the Alpha algorithm and its fundamentals while full details can be found in [15]. The Alpha algorithm at a glance is given by Algorithm 1 which is similar to the main algorithm of CDNL solvers. There is one loop in which the search space is explored and each iteration begins with propagating from the already derived knowledge. If a conflict occurs, it is analyzed in line (a) and a new nogood is derived following the first-UIP schema for conflict-driven learning. In (b) the grounder is requested to derive new nogoods from the assignment derived so far. This is the lazy-grounding part and it is usually absent in CDNL solvers. In

Algorithm 1: The Alpha Algorithm (simplified).

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

Initialize $\mathcal{AS} = \emptyset$, assignment A, and nogood storage Δ.
Run lazy grounder, obtain initial nogoods Δ from facts.
while *search space not exhausted* **do**
  Propagate on Δ extending A.
  if *there exists conflicting nogood* **then**
    Analyze conflict, learn new nogood, and backjump. **(a)**
  else if *propagation extended* A **then**
    Run lazy grounder wrt. A and extend Δ. **(b)**
  else if *exists an applicable rule* **then** **(c)**
    Guess as chosen by heuristic.
  else if *exists unassigned atom* **then** **(d)**
    Assign all unassigned atoms to false.
  else if *no atom in* A *assigned to must-be-true* **then** **(e)**
    $\mathcal{AS} \leftarrow \mathcal{AS} \cup \{A\}$
    Add enumeration nogood and backtrack.
  else **(f)**
    Backtrack.
return $\mathcal{AS}$

(c) a heuristic decides which atom to guess on. This way of guessing has been newly developed for Alpha and it ensures that the atom guessed on corresponds exactly to an applicable ground rule, i.e., the positive body of the ground rule is already in the assignment and the negative body is not (yet) contradicted by the assignment. When (d) is reached, the interplay of propagation, grounding, and guessing has reached a fixpoint: there are no more applicable ground rule instances and nothing can be derived by propagation or from further grounding. However, there may still be some atoms with unassigned truth value, because the guessing is restricted and does not guess on all atoms. Therefore in (d) all unassigned atoms are explicitly assigned to *false* and the propagation at the beginning of the following iteration ensures that no nogood is violated. Finally, in (e) the solver checks whether there is an atom assigned to *must-be-true* but could not be derived by some rule firing to become *true*. If there is no *must-be-true*, the current assignment corresponds to an answer-set and it is recorded as such. If the check fails, the current assignment is no answer-set and backtracking occurs in (f).

In order to represent rules using nogoods, Alpha introduces the notion of a *nogood with head*, that is, a nogood $ng = \{s_1, \ldots, s_n\}_i$ with one distinguished negative literal s_i, $1 \leq i \leq n$, such that $s_i = \mathbf{F}a$ for some $a \in \mathcal{A}$. The head of a nogood is denoted by $hd(ng) = s_i$. The head literal, intuitively, captures the idea of the head of a logic programming rule: if the nogood is unit on the head, it is assigned to *true* and not just *must-be-true*.

The full representation of a rule by nogoods is as follows: let r be a rule and σ be a substitution such that $r\sigma$ is ground, let the positive body be $B^+(r\sigma) = \{a_1, \ldots, a_k\}$ and the negative body be $B^-(r\sigma) = \{a_{k+1}, \ldots, a_n\}$ while the head is $H(r\sigma) = \{a_0\}$, then the *nogood representation* is given by the following nogoods:

$$\{\mathbf{F}\beta(r,\sigma), \mathbf{T}a_1, \ldots, \mathbf{T}a_k, \mathbf{F}a_{k+1}, \ldots, \mathbf{F}a_n\}_1, \{\mathbf{F}a_0, \mathbf{T}\beta(r,\sigma)\}_1,$$
$$\{\mathbf{T}\beta(r,\sigma), \mathbf{F}a_1\}, \ldots, \{\mathbf{T}\beta(r,\sigma), \mathbf{F}a_k\}, \{\mathbf{T}\beta(r,\sigma), \mathbf{T}a_{k+1}\}, \ldots, \{\mathbf{T}\beta(r,\sigma), \mathbf{T}a_n\}$$

The new atom $\beta(r,\sigma)$, intuitively, represents the body of the ground rule $r\sigma$. Notice that the first and second nogood each has a head (as indicated by the subscript 1, the head is the first literal). Despite similarities, this nogood representation differs from the one used by Clingo: first, Alpha uses nogoods with heads and second, there are no nogoods establishing support of ground atoms, because that would require full grounding.

Example 2. Consider from Example 1 the rule r as follows:

$$sel(X) :- dom(X), not\ nsel(X).$$

From an assignment A where $dom(3)$ holds, i.e., $\mathbf{T}dom(3) \in A$, the grounder generates the substitution $\sigma : X \mapsto 3$ for r and it introduces the new atom $\beta(r,3)$ representing the body of the ground rule $r\sigma$. It then yields the following nogoods:

n_1 : $\{\mathbf{F}\beta(r,3), \mathbf{T}dom(3), \mathbf{F}nsel(3)\}_1$ $\quad$ n_2 : $\{\mathbf{T}\beta(r,3), \mathbf{F}dom(3)\}$

n_3 : $\{\mathbf{T}\beta(r,3), \mathbf{T}dom(3)\}$ $\quad$ n_4 : $\{\mathbf{F}sel(3), \mathbf{T}\beta(r,3)\}_1$

Nogoods n_1 to n_3 establish that $\beta(r,3)$ holds if and only if the body of the ground rule holds. Nogood n_4 ensures that the head atom is *true* whenever $\beta(r,3)$ holds. Observe that n_1 and n_4 have their first literal indicated as head, i.e., the solver will not set them to *must-be-true* but to *true* whenever the nogood is unit and all other positively occurring literals are *true*. This enables the nogoods to represent rules in the presence of two truth values, *must-be-true* and *true*.

4 Efficient Propagation: 3-Watched-Literals

This section provides details on efficient propagation realized in Alpha. Our approach extends the state-of-the-art propagation technique from SAT and CDNL-based ASP solving known as the 2-watched literals (2WL) schema (cf. [1]). A direct use of 2WL in lazy-grounding ASP solving, however, is not possible due to such solvers using *must-be-true* as a third truth value requiring special treatment. Since *must-be-true* allows propagation to *true*, but no other truth value may be changed once it is assigned, this requires a different propagation mechanism than 2WL, which is designed for propagation to *true* and *false* only.

Formally, propagation is the task of identifying nogoods that are unit, i.e., nogoods violated except for one yet unassigned literal whose truth value then is set in order to avoid violating the nogood, and subsequently assigning this unassigned literal. In Alpha, a nogood with head may propagate to the truth

value *true*, *false*, and *must-be-true* while a nogood without head may only propagate to *false* and *must-be-true*. Subsequently, there are two notions of being unit: weakly-unit and strongly-unit. Formally, an *assignment* A in Alpha is over truth values $\mathbf{T}$, $\mathbf{F}$, and $\mathbf{M}$; the *Boolean-projection* $A^{\mathcal{B}}$ maps $\mathbf{M}$ to $\mathbf{T}$, i.e., $A^{\mathcal{B}} = \{\mathbf{T}a \mid \mathbf{T}a \in A \text{ or } \mathbf{M}a \in A\} \cup \{\mathbf{F}a \mid \mathbf{F}a \in A\}$. 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 $\overline{s} \notin A$. By this definition a nogood with head is strongly-unit only if all its positively occurring literals are assigned to *true*. Also note that only a nogood with head can be strongly-unit and if a nogood is strongly-unit, it also is weakly-unit.

Propagation is the least fixpoint of the *immediate unit-propagation*, i.e., $propagate(A) = lfp\big(\Gamma_\Delta(A)\big)$ s.t. for a set Δ of nogoods and an assignment A:

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

In order to compute the propagation efficiently, we extend the concept of two-watched literals to our setting where nogoods may have a head literal and a nogood can be unit in two different ways. Two-watched literals, intuitively is based on the following observations: if a nogood δ contains more than two literals $s_1, s_2, s_3 \in \delta$ that are unassigned in some assignment A and one of these, say s_3, becomes assigned in $A' \supset A$, then δ is still not unit. Hence for as long as there are at least two unassigned literals, the nogood need not be checked for being unit. Therefore each nogood only requires two of its unassigned literals to be watched for being assigned in order to detect when the nogood is unit.

For our setting where a nogood may be weakly-unit or strongly-unit, intuitively, two-watched literals are required twice, 2WL for each type of being unit. Since the literal that will be propagated by a strongly-unit nogood always is the head literal of the nogood, it need not be watched explicitly. Therefore, three watches are sufficient. These watches are organized such that each atom is assigned one list per polarity and unit-type. Notice that each nogood requires only three watches but for each atom there are four types of watches.

Definition 1. *A* watch structure *W for an assignment A and a set of nogoods Δ is a mapping $W : \mathcal{A} \to \Delta^4$ of atoms to quadruples of lists (sets) of nogoods in Δ. For a watch structure W, each atom $a \in \mathcal{A}$ is associated a quadruple of lists*

$$W(a) = \big(watch^+(a), watch^-(a), watch^+_\alpha(a), watch^-_\alpha(a)\big).$$

The list $watch^+(a)$ (resp. $watch^-(a)$) contains all nogoods δ where a watch is on a positive literal $\mathbf{T}a \in \delta$ (res. negative literal $\mathbf{F}a \in \delta$) for detecting whether δ is *weakly-unit*. The list $watch^+_\alpha(a)$, resp. $watch^-_\alpha(a)$, contains all nogoods δ where a watch is on a positive literal $\mathbf{T}a \in \delta$, resp. negative literal $\mathbf{F}a \in \delta$, for detecting whether δ is *strongly-unit*.

A visualization of this data structure is given in Figure 2.

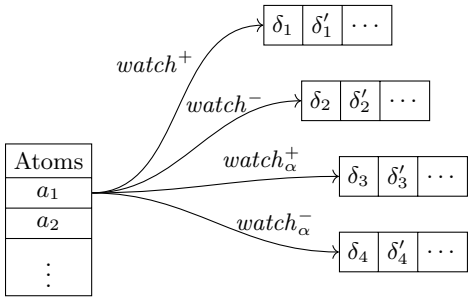

Fig. 2. Data structure for accessing watched nogoods.

For convenience, in the following we denote for a signed literal $s = \mathbf{X}a$ by $watch(s)$ the list $watch^+(a)$ if $\mathbf{X} \in \{\mathbf{T}, \mathbf{M}\}$ and $watch^-(a)$ otherwise. Similarly, $watch_\alpha(s)$ denotes $watch^+_\alpha(a)$ if a $\mathbf{X} \in \{\mathbf{T}, \mathbf{M}\}$ and $watch^-_\alpha(a)$ if $\mathbf{X} = \mathbf{F}$.

In order to obtain correctly watched literals also after backtracking and subsequent assignments (where some assigned atoms may become unassigned and subsequently being propagated), the watches for satisfied nogoods have to point at those literals that were assigned in the highest decision level.

Given an assignment A and an atom a, we denote by $dl^w(A, a)$ the decision level on which a is assigned to *must-be-true* or *false* in A. Similarly, $dl^s(A, a)$ denotes the decision level at which a is assigned to *true* or *false* in A. Furthermore, for a signed literal $s = \mathbf{X}a$ with $\mathbf{X} \in \{\mathbf{F}, \mathbf{T}, \mathbf{M}\}$, we denote by $at(s)$ the atom of the literal, i.e., $at(s) = a$.

Intuitively, the watches of a nogood have to point at either (1) two unassigned literals, or (2) one of these literals atoms is assigned such that the nogood is satisfied and the other literal is either unassigned or assigned at an equal-or-higher decision level. The latter condition ensures that if backtracking removes the satisfying assignment then the second watched literal is guaranteed to be unassigned, i.e., even in case of backtracking the nogood is guaranteed to be either satisfied or contain two unassigned and watched literals.

Definition 2. *Let δ be a nogood and A be an assignment, then $s, s' \in \delta$ are* potential watches *if one of the following holds.*

(i) $at(s)$ and $at(s')$ are both unassigned in A.
(ii) The atom of s is complementary assigned, i.e., $\overline{s} \in A^{\mathcal{B}}$, and either s' is unassigned in A or $dl^w(A, at(s')) \geq dl^w(A, at(s))$.

For a nogood with head there is only one watch, which is not the head itself, and it obeys a similar condition; the main difference being that an atom assigned to *must-be-true* is treated like it were unassigned.

Definition 3. *Let δ be a nogood with head and A be an assignment, then $s_\alpha \in \delta$ with $hd(\delta) \neq s_\alpha$ is a* potential α-watch *if one of the following holds.*

(i) $at(s_\alpha)$ is unassigned in A, assigned to must-be-true in A, or s_α is complementary assigned in A.

(ii) $dl^s(A, at(s_\alpha)) \geq dl^s(A, at(hd(\delta)))$ *and the head is true, i.e.,* $\mathbf{T}at(hd(\delta)) \in A$.

Example 3. Consider the assignment $A = \{\mathbf{M}c, \mathbf{F}d\}$ with $dl^w(A, c) \leq dl^w(A, d)$, i.e., $\mathbf{M}c$ was assigned at lower decision level than $\mathbf{F}d$, and the nogoods $\delta_1 = \{\mathbf{F}a, \mathbf{T}b, \mathbf{T}c, \mathbf{F}d, \mathbf{F}e\}_1$, $\delta_2 = \{\mathbf{F}c, \mathbf{F}d\}$, and $\delta_3 = \{\mathbf{F}a, \mathbf{T}c\}_1$, where δ_1 and δ_3 are nogoods with a head. For δ_1 any two literals from $\{\mathbf{F}a, \mathbf{T}b, \mathbf{F}e\}$ are potential watches since they are all unassigned and any literal in $\{\mathbf{T}b, \mathbf{T}c, \mathbf{F}e\}$ is a potential α-watch. The nogood δ_2 has the potential watches $\mathbf{F}c$ and $\mathbf{F}d$ since A assigns c complementary to its occurrence in δ_2 and d has higher decision level than c. Since δ_2 has no head, there is no potential α-watch. For δ_3 the literal $\mathbf{T}c$ is a potential α-watch since c is assigned *must-be-true* in A, but δ_3 has no potential watches, intuitively, because δ_3 is weakly-unit under A and propagates $\mathbf{F}a$.

Intuitively, a watch structure is consistent for an assignment and a set of nogoods, if each nogood is watched correctly.

Definition 4. *A watch structure W for a set of nogoods Δ is* consistent *with an assignment A if for each nogood $\delta \in \Delta$ there exist potential watches s, s' and, for δ being a nogood with head, a potential α-watch s_α such that $\delta \in watch(s)$, $\delta \in watch(s')$, and $\delta \in watch_\alpha(s_\alpha)$ all hold.*

Example 4 (continued). Let A be the same as in Example 3 and let $\Delta = \{\delta_1, \delta_2\}$. One watch structure W consistent with Δ and A is as follows:

$$W(a) = \big(\emptyset, \{\delta_1\}, \emptyset, \emptyset\big) \quad W(b) = \big(\{\delta_1\}, \emptyset, \{\delta_1\}, \emptyset\big) \quad W(c) = \big(\emptyset, \{\delta_2\}, \emptyset, \emptyset\big)$$
$$W(d) = \big(\emptyset, \{\delta_2\}, \emptyset, \emptyset\big) \quad W(e) = \big(\emptyset, \emptyset, \emptyset, \emptyset\big)$$

Thus W watches δ_1 on $\mathbf{F}a$, $\mathbf{T}b$, and α-watches it on $\mathbf{T}b$. Furthermore, it watches δ_2 on $\mathbf{F}c$ and $\mathbf{F}d$ while there exists no α-watch for δ_2 since it has no head. W is consistent because all watched literals in W are also potential (α-)watches in A. Note that for $\Delta' = \{\delta_1, \delta_2, \delta_3\}$ and A there exists no consistent watch structure since δ_3 has no potential watches (it is weakly-unit in A).

Computing *propagate*(A) is possible using Algorithm 2 where a watch structure W consistent with the current assignment A and set of nogoods Δ is maintained. Notice that the algorithm receives as input a set Σ of new assignments, i.e., assignments done by Algorithm 1 outside of propagation (for example by guessing or backtracking). Intuitively, Algorithm 2 iterates over all new assignments (including those it derives itself during propagation) until all new assignments have been processed. For each new assignment the two lists of watched nogoods fitting to the polarity of the assignment are considered, e.g., if $\mathbf{F}d$ is a new assignment then only nogoods δ with $\mathbf{F}d \in \delta$ are considered. Each of those lists is then checked whether one of its nogoods is violated, weakly-unit, or strongly-unit. If one of the latter two is the case, a new assignment is recorded. Afterwards, the watch structure is adapted such that consistency (with regard to the currently processed assignment) is restored.

The following properties can be shown:

Algorithm 2: *propagate*

Input: An assignment A, a set Σ of new assignments, and a watch structure W consistent with A and Δ.

Output: An (extended) assignment A' or a pair of extended assignment A' and a violated nogood d.

$A' \leftarrow A$
while $\Sigma \neq \emptyset$ **do**
  $\Sigma \leftarrow \Sigma \setminus \{\mathbf{X}a\}$ for some $\mathbf{X}a \in \Sigma$. `// Process each new assignment.`
  $(\Delta, \Delta_\alpha) \leftarrow \begin{cases} (watch^+(a), watch^+_\alpha(a)) & \text{if } \mathbf{X} \in \{\mathbf{T}, \mathbf{M}\}, \text{ and} \\ (watch^-(a), watch^-_\alpha(a)) & \text{otherwise.} \end{cases}$
  foreach $\delta \in \Delta$ **do** `// Propagation to` $\mathbf{M}, \mathbf{F}$.
    if δ *is violated* **then**
      return (A', δ)
    else if δ *is weakly-unit for* s **then**
      Let $s' = \mathbf{M}b$ if $s = \mathbf{F}b$ and $s' = \mathbf{F}b$ otherwise.
      $A' \leftarrow A \cup \{s'\}$
      $\Sigma \leftarrow \Sigma \cup \{s'\}$
    Remove δ from Δ. `// Update ordinary watches.`
    Let s, s' be potential watches of δ
    $watch(at(s)) \leftarrow watch(at(s)) \cup \{\delta\}$
    $watch(at(s')) \leftarrow watch(at(s')) \cup \{\delta\}$
  foreach $\delta \in \Delta_\alpha$ **do** `// Propagation to` $\mathbf{T}$.
    if δ *is strongly-unit* **then**
      $A' \leftarrow A \cup \{\mathbf{T}at(hd(\delta))\}$
      $\Sigma \leftarrow \Sigma \cup \{\mathbf{T}at(hd(\delta))\}$
    Remove δ from Δ_α. `// Update alpha watch.`
    Let s be a potential α-watch of δ
    $watch_\alpha(at(s)) \leftarrow watch_\alpha(at(s)) \cup \{\delta\}$
return A'

Proposition 1. *Let W be a watch structure W for a set of nogoods Δ that is consistent with an assignment A and let $A' \supseteq A$ be a larger assignment with $\Sigma = A' \setminus A$. Then, Algorithm 2 running on A, Σ, and W returns either*

1. *a pair (A'', δ) such that A'' is a consequence of A' and Δ and $\delta \in \Delta$ is violated by A'', or*
2. *an assignment $A'' = propagate(A')$ and the modified watch structure is consistent with A'' and Δ.*

5 Evaluation

We evaluated the Alpha solver on four benchmarks, that exercise different parts of a solver, comparing Alpha to the lazy-grounding solvers Omiga and AsPeRiX as well as to Clingo. All benchmarks were performed on a Linux machine with

two 12-core AMD Opteron 6176 SE CPUs and 128 GB RAM. The timeout for each run was 300 secs and the memory limit 8 GB. The *HTCondor* system[3] was used for load distribution to minimize runtime variations for different runs. Since Java restricts itself to use only parts of the available system memory, the JVM was instructed that 8 GB of RAM are available and that it can use up to 3.5 GB for heap allocations, i.e., Java was called with the following command-line arguments: `-XX:MaxRAM=8000M -Xmx3500M`

We report the average runtimes in seconds on 10 randomly generated instances for each benchmark problem, except for one benchmark where only one instance per size exists. The compared solver versions were: Clingo version 5.2.0, AsPeRiX version 2.5, Omiga built from source using Git commit 037b3f9 and Alpha from source using Git commit a65421f.

Ground Explosion. This benchmark is the program of Example 1, i.e., given some domain, select exactly one element from the domain and derive a new atom containing the selected element six times. Table 1 shows the runtimes for domain sizes from 8 up to 1.000 where each solver is requested to compute 10 answer sets.

All lazy-grounding ASP solvers compute the answer sets within seconds for all instances, while Clingo runs out of 8GB memory with a domain of size 18 already. Comparing Alpha with Omiga and AsPeRiX one can observe that Alpha is slower than the other two. This is likely caused by Alpha having to maintain the data structures of a CDNL solver (e.g., creation of nogoods, watch structures, etc.) while Omiga and AsPeRiX use a more direct representation of rules. One surprising result is that AsPeRiX takes more than 5 seconds for the instance with domain of size 8, which is much higher than for larger instances. A closer investigation revealed that AsPeRiX needs a lot of time to detect when no more answer sets exist for this particular problem. This only shows for this particular instance where there exist less than the requested 10 answer sets. Requesting 14 answer sets from AsPeRiX for the instance with domain size 12 already results in a timeout. Alpha, in contrast, does not exhibit the same problem.

Cutedge benchmarks. This problem was first introduced in [3] and is as follows: given a graph $G = (V, E)$, choose one edge $e \in E$ and compute reachability on the graph G' where e is cut, i.e., $G' = (V, E \setminus \{e\})$. This problem is hard for ASP systems that are based on grounding the program upfront, while it is significantly easier for lazy-grounding ASP solvers.

We ran this problem on graphs with 100 to 500 vertices and 3.000 to 125.000 edges, instructing the solvers to compute 10 answer sets each. The results are given in Table 2. As expected, Clingo is only able to solve small instances and starting from graphs with 12.000 edges Clingo always hits the timeout of 300 seconds. Surprisingly, Clingo hits the timeout and does not run out of memory within 300 seconds. Closer inspection revealed, that Clingo indeed runs out of memory when given more time. It seems, however, that grounding in Clingo is

[3] http://research.cs.wisc.edu/htcondor

Instance size	Alpha	Omiga	AsPeRiX	Clingo
8	1.37(0)	0.42(0)	5.54(0)	1.74(0)
10	1.48(0)	0.43(0)	0.02(0)	7.00(0)
12	1.46(0)	0.44(0)	0.02(0)	22.47(0)
14	1.64(0)	0.47(0)	0.02(0)	56.39(0)
16	1.60(0)	0.51(0)	0.03(0)	145.28(0)
18	1.64(0)	0.45(0)	0.03(0)	memout
20	1.83(0)	0.46(0)	0.05(0)	memout
22	1.69(0)	0.46(0)	0.05(0)	memout
24	1.57(0)	0.47(0)	0.06(0)	memout
26	1.24(0)	0.44(0)	0.06(0)	memout
28	1.29(0)	0.47(0)	0.06(0)	memout
30	1.52(0)	0.49(0)	0.07(0)	memout
50	1.31(0)	0.53(0)	0.11(0)	memout
100	1.60(0)	0.81(0)	0.21(0)	memout
300	1.58(0)	0.93(0)	0.63(0)	memout
500	2.19(0)	1.41(0)	1.06(0)	memout
1000	2.30(0)	1.66(0)	2.21(0)	memout

Table 1. Grounding explosion benchmark results. Instance size is the overall number of constants in the domain. Shown is runtime in seconds; out of memory is indicated by memout.

not fast enough to run out of 8GB memory within 300 seconds time. Table 2 further shows that Alpha is comparable to AsPeRiX and both are slower than Omiga. This may be rooted in the fact that Omiga uses a Rete network for efficient grounding while Alpha uses a semi-naive grounding procedure similar to that of AsPeRiX.

Graph Colorability. This problem is inspired by the problem with the same name from the ASP competition. The task is to color a given graph with 5 available colors. This problem poses no grounding problem but requires efficient search procedures. The benchmark was run on randomly generated instances with 10 to 1.000 vertices and 40 to 4.000 edges. For each setting 10 random graphs were constructed. The average runtimes in seconds is reported in Table 3.

As expected, this benchmark is very easy for Clingo, while the lazy-grounding solvers Omiga and AsPeRiX struggle for all but the trivial instances. AsPeRiX performs better than Omiga, even solving instances with 100 vertices and 200 edges. These graphs are very sparse, however, and nearly each coloring yields an answer set. For less-trivial instances with more edges per vertex, like those with 30 vertices and 120 edges, Omiga and AsPeRiX time out on all of them. Alpha on the other hand, is able to solve also the harder instances where search is non-trivial. Comparing Alpha with Clingo we observe that there still is a significant gap in terms of search performance. This is rooted in the fact that Clingo employs numerous efficient search techniques (heuristics, nogood forgetting, nogood minimization, etc.) that are largely lacking in Alpha. There is some progress on implementing heuristics in Alpha (cf. [14]), but due to the specifics

Instance size	Alpha	Omiga	AsPeRiX	Clingo
100/30	12.59(0)	4.25(0)	0.78(0)	27.64(0)
100/50	11.87(0)	6.22(0)	1.79(0)	79.50(0)
200/30	22.90(0)	13.46(0)	13.29(0)	300.00(10)
200/50	45.95(0)	24.20(0)	35.18(0)	300.00(10)
300/10	16.92(0)	10.08(0)	8.54(0)	291.35(4)
300/30	59.58(0)	32.36(0)	72.09(0)	300.00(10)
400/10	40.97(0)	20.72(0)	27.61(0)	300.00(10)
400/30	300.00(10)	84.73(0)	284.71(4)	300.00(10)
500/10	62.46(0)	32.01(0)	70.38(0)	300.00(10)
500/30	300.00(10)	122.16(0)	300.00(10)	300.00(10)
500/50	300.00(10)	215.01(0)	300.00(10)	300.00(10)

Table 2. Cutedge benchmark results. Instance is number of number of vertices / average percentage of edge being present. Shown is the average runtime in seconds over 10 instances with number of timeouts in parentheses.

Instance size	Alpha	Omiga	AsPeRiX	Clingo
10/40	1.41(0)	14.33(0)	31.10(1)	0.02(0)
20/80	1.53(0)	234.93(6)	128.79(4)	0.02(0)
30/120	1.59(0)	300.00(10)	230.23(7)	0.03(0)
40/160	2.54(0)	300.00(10)	217.17(7)	0.04(0)
50/200	2.31(0)	300.00(10)	300.00(10)	0.04(0)
100/400	4.24(0)	300.00(10)	300.00(10)	0.06(0)
400/1600	22.54(0)	300.00(10)	300.00(10)	0.45(0)
500/2000	33.85(0)	300.00(10)	300.00(10)	0.68(0)
750/3000	67.22(0)	300.00(10)	300.00(10)	1.46(0)
1000/4000	119.94(0)	300.00(10)	300.00(10)	2.66(0)

Table 3. Graph 5-colorability benchmark results. Instance is number of vertices / number of edges. Shown is the average runtime in seconds over 10 instances with number of timeouts in parentheses.

of lazy-grounding (restricted guessing, etc.) the techniques of Clingo cannot be adapted directly.

In order to more precisely compare the lazy-grounding solvers, Table 4 shows their runtimes on graphs with a fixed number of 50 vertices and an increasing number of edges. Omiga has timeouts even for 50 edges while AsPeRiX is able to handle 100 edges. With more than 100 edges Alpha is the only lazy-grounding solver that returned the requested answer sets in time.

Reachability. This benchmark is comprised of a simple positive program computing reachability in a large graph. The task is: given some start vertex of a graph, compute the set of all vertices reachable from the start vertex. The tests were run on 10 randomly generated graphs for each instance size, with 1.000 and 10.000 vertices and 4.000 to 80.000 edges. Since the resulting ASP program contains no negation, Clingo only uses its intelligent grounder while the solver

Instance size	Alpha	Omiga	AsPeRiX	Clingo
50/50	1.88(0)	290.47(9)	0.24(0)	0.03(0)
50/100	2.05(0)	300.00(10)	0.45(0)	0.03(0)
50/200	2.31(0)	300.00(10)	300.00(10)	0.04(0)
50/300	74.39(2)	300.00(10)	300.00(10)	0.07(0)
50/400	253.80(8)	300.00(10)	300.00(10)	0.06(0)
50/500	168.76(4)	300.00(10)	300.00(10)	0.04(0)

Table 4. Graph 5-colorability benchmark on graphs with 50 vertices varying edges. Instance is number of vertices / number of edges. Shown is the average runtime in seconds over 10 instances with number of timeouts in parentheses.

has no work left to do. The benchmark therefore allows to compare the speed (and overhead) of lazy-grounding with a highly-optimized grounder.

Table 5 shows the results of this benchmark. On large instances, Alpha is the fastest of all lazy-grounding solvers while for smaller instances Omiga and AsPeRiX are faster. Since this purely positive program can be fully solved by an intelligent grounder, Clingo is fastest here.

Instance size	Alpha	Omiga	AsPeRiX	Clingo
1000/4	2.13(0)	1.21(0)	0.77(0)	0.11(0)
1000/8	3.19(0)	1.63(0)	2.57(0)	0.21(0)
10000/2	10.95(0)	7.82(0)	31.11(0)	0.52(0)
10000/4	13.06(0)	22.55(0)	130.00(0)	1.09(0)
10000/8	16.62(0)	56.93(0)	300.00(10)	2.27(0)

Table 5. Reachability benchmark results. Instance size is number of vertices / multiple of edges of the random graph.

Summary. We observe that Alpha is comparable in speed to the other lazy-grounding solvers for problems where lazy-grounding avoids the grounding bottleneck. In addition to that, Alpha provides much better search performance, making search-intense problems solvable using lazy-grounding. There are, however, many efficient solving techniques not yet available for lazy-grounding ASP solving, making it slower than state-of-the-art ASP solvers on problems where grounding is not an issue. For ASP programs where grounding is problematic, however, Alpha is the best choice as it provides a good compromise between grounding performance and solving performance.

Note that Alpha is freely available at: https://github.com/alpha-asp/Alpha

6 Conclusion

We presented the novel ASP solver Alpha which combines lazy-grounding and CDNL-search to obtain a system that is both, avoiding the grounding bottleneck

and efficiently exploring the search space. An overview of Alpha and its architecture was given. To provide an efficient propagation the well-known 2-watched literals schema was enhanced to 3-watched literals in order to cope with nogoods being unit in two distinct ways.

Benchmarks showed that Alpha now is on-par with other lazy-grounding solvers on problems where grounding is an issue, while it provides a significant improvement for problems where search is dominating. Since Alpha is very recent, it lacks several important optimizations for search, making it noticeably slower than Clingo. Contrary to Clingo, however, Alpha does not suffer from the grounding bottleneck.

Topics for future work, among many others, are forgetting of learned nogoods, and using dependency information like strongly-connected-components for faster solving.

References

1. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
2. de Cat, B., Denecker, M., Bruynooghe, M., Stuckey, P.J.: Lazy model expansion: Interleaving grounding with search. J. Artif. Intell. Res. 52, 235–286 (2015)
3. 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)
4. Eiter, T., Ianni, G., Krennwallner, T.: Answer set programming: A primer. In: Reasoning Web 2009. LNCS, vol. 5689, pp. 40–110. Springer (2009)
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: *clasp* : A conflict-driven answer set solver. In: LPNMR. pp. 260–265 (2007)
6. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. In: LPNMR 2007. LNCS, vol. 4483, pp. 136–148. Springer (2007)
7. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. 187, 52–89 (2012)
8. Lefèvre, C., Beatrix, C., Stephan, I., Garcia, L.: Asperix, a first-order forward chaining approach for answer set computing. TPLP p. 145 (Jan 2017)
9. Lefèvre, C., Nicolas, P.: A first order forward chaining approach for answer set computing. In: LPNMR. pp. 196–208 (2009)
10. Lefèvre, C., Nicolas, P.: The first version of a new ASP solver : Asperix. In: LPNMR. pp. 522–527 (2009)
11. 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)
12. Liu, L., Pontelli, E., Son, T.C., Truszczynski, M.: Logic programs with abstract constraint atoms: The role of computations. In: ICLP. LNCS, vol. 4670, pp. 286–301. Springer (2007)
13. Palù, A.D., Dovier, A., Pontelli, E., Rossi, G.: Gasp: Answer set programming with lazy grounding. Fundam. Inform. 96(3), 297–322 (2009)
14. Taupe, R., Weinzierl, A., Schenner, G.: Introducing heuristics for lazy-grounding ASP solving. In: PAoASP (2017), to appear
15. Weinzierl, A.: Blending lazy-grounding and CDNL search for answer-set solving. In: LPNMR. pp. 191–204 (2017)