Introducing Heuristics for Lazy-Grounding ASP Solving[*]

RICHARD TAUPE[1,2], ANTONIUS WEINZIERL[3], and GOTTFRIED SCHENNER[1]

[1] *Siemens AG Österreich, Corporate Technology, Vienna, Austria,*
(*e-mail:* `firstname.lastname@siemens.com`)

[2] *Alpen-Adria-Universität, Klagenfurt, Austria,*
(*e-mail:* `rtaupe@edu.aau.at`)

[3] *Institute of Information Systems, TU Wien, Vienna, Austria,*
(*e-mail:* `weinzierl@kr.tuwien.ac.at`)

submitted 3 April 2017; revised 12 June 2017; accepted 22 May 2017

Abstract

Most ASP systems suffer from the so-called *grounding bottleneck*, i.e. they cannot solve problems whose propositional grounding exceeds given memory limits. Lazy-grounding solvers mitigate this issue by interleaving grounding with search but they cannot compete with state-of-the-art solvers in terms of runtime performance. The latter draw their power from conflict-driven nogood learning (CDNL) and from sophisticated domain-independent or domain-specific heuristics. The direct transfer of traditional heuristics to lazy-grounding solvers is not possible because these solvers do not guess the truth value of atoms but whether a rule fires or not. In this paper, we introduce the first heuristics for lazy-grounding ASP solving, using the CDNL-based lazy-grounding solver Alpha. We develop a family of novel branching heuristics inspired by traditional ones, implement them in Alpha, and investigate their performance on a set of benchmark problems. We observe considerable performance gains in many cases.

KEYWORDS: answer set programming, lazy grounding, heuristics

1 Introduction

The so-called *grounding bottleneck* is a major problem of traditional approaches to answer set programming (ASP), which follow the *ground-and-solve* paradigm. They transform an input program containing variables into its propositional counterpart, which might result in an exponential blow-up in space. For example, the constraint

$$\leftarrow \; sp(SP, P1),\; sp(SP, P2),\; P1 \neq P2.$$

has to be replaced by up to $|SP| \cdot |P1| \cdot |P2|$ ground constraints, where $|SP|$, $|P1|$ and $|P2|$ are the number of values the respective variable may take. Many of these ground constraints may be irrelevant because they cannot be satisfied in a given problem instance anyway. Problems that are actually easy to solve thus become

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

prohibitive as soon as their grounding does not fit into working memory. This makes ASP, an otherwise powerful and versatile approach, unsuitable for large-scale problem instances frequently occurring in real-world settings.

Example 1.1
For illustration, consider a simple configuration problem:

$$\left\{\begin{array}{l}
\mathit{part}(N) \leftarrow n(N),\ \mathit{not}\ \mathit{no\_part}(N). \\
\mathit{no\_part}(N) \leftarrow n(N),\ \mathit{not}\ \mathit{part}(N). \\
\mathit{spid}(SP, ID) \leftarrow \mathit{sp}(SP, P),\ n(ID),\ \mathit{not}\ \mathit{no\_spid}(SP, ID). \\
\mathit{no\_spid}(SP, ID) \leftarrow \mathit{sp}(SP, P),\ n(ID),\ \mathit{not}\ \mathit{spid}(SP, ID). \\
\mathit{sp}(SP, P) \leftarrow \mathit{part}(P),\ \mathit{part}(SP),\ P \neq SP,\ \mathit{not}\ \mathit{no\_sp}(SP, P). \\
\mathit{no\_sp}(SP, P) \leftarrow \mathit{part}(P),\ \mathit{part}(SP),\ P \neq SP,\ \mathit{not}\ \mathit{sp}(SP, P). \\
\leftarrow \mathit{sp}(SP, P1),\ \mathit{sp}(SP, P2),\ P1 \neq P2. \\
\leftarrow \mathit{sp}(SP1, P),\ \mathit{sp}(SP2, P),\ SP1 \neq SP2,\ \mathit{spid}(SP1, ID),\ \mathit{spid}(SP2, ID).
\end{array}\right\}$$

Together with a problem instance of the form $\{n(1).,\ldots,n(N).\}$, the grounding of this program quickly exceeds usual memory limits. For example, for an instance as little as $N = 75$, GRINGO 5.1.0 (Gebser et al. 2011) produces a grounding of roughly 1GB and takes more than one minute to do so on an average computer, even though the resulting ground program can be solved by CLASP (Gebser et al. 2012) in less than one second subsequently. Therefore, traditional ASP systems cannot be used for problems of this size, which is still small compared to industrial application scenarios.

Sophisticated grounding techniques employed by state-of-the-art systems omitting irrelevant ground rules do not help, because they can only reduce and not eliminate the exponential blow-up in the worst case. *Lazy grounding* interleaves grounding and search to avoid storing the entire ground program in memory.

Approaches to lazy grounding for full ASP contain ASPeRiX (Lefèvre and Nicolas 2009; Lefèvre et al. 2017), GASP (Dal Palù et al. 2009), and OMIGA (Dao-Tran et al. 2012; Weinzierl 2013). They all get lost during search, however. The recently proposed ALPHA solver (Weinzierl 2017) blends lazy grounding with conflict-driven nogood learning (CDNL), a key technique in state-of-the-art pre-grounding systems like CLASP. It does not, however, reach the performance of traditional solvers yet.

A key issue in lazy grounding following the above approaches is that, in order to obtain unfounded-free models, choices have to correspond to rules that are *applicable*, a property that depends on the current state of the search and may change even in the same search branch. Equivalent criteria can be guaranteed by loop nogoods and nogoods guaranteeing support, as in Dovier et al. (2016). Procuring such nogoods, however, relies on knowing all ground rules that possibly derive an atom, i.e., it requires the full grounding, which conflicts with lazy grounding.

Another approach to interleaving grounding with search is presented by de Cat et al. (2015). It relies on a semantic totality property, which is not guaranteed for ASP programs with cycles over negation. The authors propose manual rewriting, an unsatisfying option for practical use of ASP.

We propose branching heuristics to steer the solver of CDNL-based lazy-grounding systems into the right parts of the search space while also respecting applicability.

A major challenge is that traditional heuristics are not directly transferable: they do not respect the variable applicability condition and would pick invalid choices.

The main contributions of this work are the following:

- an adaption of domain-independent heuristics tried and trusted by traditional solvers to a lazy-grounding solver,
- the first account of heuristics specifically designed for lazy-grounding solvers,
- a prototypical design of a domain-specific heuristic for such a solver,
- implementations of these heuristics within ALPHA[1], a CDNL-based lazy-grounding ASP system capable of solving normal logic programs,
- and an early experimental evaluation of how these heuristics perform, revealing considerable performance gains.

This paper is organized as follows: Section 2 recalls preliminaries of ASP, CDNL, and lazy grounding. After briefly outlining the Alpha approach to lazy grounding in Section 3, we present in Section 4 a short survey of established heuristics and then our newly developed heuristics. Our experiments and results are described in Section 5 and discussed in Section 6. The paper concludes in Section 7.

2 Preliminaries

Let $\mathcal{V}$ be a set of variables, $\mathcal{C}$ a finite set of constants, and $\mathcal{P}$ a finite set of predicates with associated arities, i.e. structures of the form p/k where p is the predicate's name and k its arity. $\mathcal{P}$ contains built-in predicates like $</2$, $>/2$, $\neq/2$, etc. The set $\mathcal{A}$ of (non-ground) atoms is then given by $\{p(t_1, \dots, t_n) \mid p/n \in \mathcal{P}, t_1, \dots, t_n \in \mathcal{C} \cup \mathcal{V}\} \cup \{\bot\}$, where $\bot$ is a special ground atom that can never become true. An atom $a \in \mathcal{A}$ is ground if no variable occurs in it. The set of ground atoms is denoted by $grd(\mathcal{A})$. A literal l is either an atom a or a negated atom *not* a, where *not* denotes default negation. An answer-set program P over $\mathcal{A}$ is a finite set of rules of the form

$$a_0 \leftarrow a_1, \dots, a_m,\ not\ a_{m+1}, \dots, not\ a_n.$$

where $0 \leq m \leq n$ and each $a_i \in \mathcal{A}$ for $0 \leq i \leq n$ is an atom. For a rule $r \in P$, let $H(\mathrm{r})$ denote its head a_0, $B^+(\mathrm{r})$ its positive body $\{a_1, \dots, a_m\}$ and $B^-(\mathrm{r})$ its negative body $\{a_{m+1}, \dots, a_n\}$. The entire rule body is denoted by $B(r) = \{a_1, \dots, a_m, not\ a_{m+1}, \dots, not\ a_n\}$. Rule r is a fact if $B(r) = \emptyset$, a constraint if $H(r) = \bot$, and it is ground if $H(r) \cup B(r)$ contains only ground atoms. Program P is ground if each $r \in P$ is ground.

A substitution $\sigma : \mathcal{V} \to \mathcal{C}$ is a mapping of variables to constants. Given an atom a, the result of applying a substitution σ to a is denoted by $a\sigma$. This is extended in the usual way to rules, i.e., $r\sigma$ for a rule r is $a_0\sigma \leftarrow a_1\sigma, \dots, not\ a_n\sigma$. The grounding of a rule is given by $grd(r) = \{r\sigma \mid \sigma \text{ is a substitution}\}$ and the grounding $grd(P)$ of a program P is given by $grd(P) = \bigcup_{r \in P} grd(r)$. Most modern ASP systems employ sophisticated grounding algorithms to produce smaller groundings than that, cf. GRINGO (Gebser et al. 2011) and $\mathcal{I}$-DLV (Faber et al. 2012; Calimeri et al. 2016).

The semantics of a ground program P are given by its *answer sets*, or *stable*

[1] The source code of ALPHA is freely available at `https://github.com/alpha-asp`.

models (Gelfond and Lifschitz 1988). A set $X \subseteq grd(\mathcal{A}) \setminus \{\bot\}$ is a *model* of P if for all $r \in P$ it holds that $H(r) \in X$ if $B^+(r) \subseteq X$ and $B^-(r) \cap X = \emptyset$. X is an answer set of P if it is the $\subseteq$-minimal model of P's *reduct* P^X relative to X, which is defined by: $P^X = \{H(r) \leftarrow B^+(r) \mid r \in P, B^-(r) \cap X = \emptyset\}$. An assignment A is a set of boolean signed literals, which are of the form $\mathbf{T}a$ and $\mathbf{F}a$ for $a \in grd(\mathcal{A})$. Thereby, an assignment is a (partial) interpretation where false atoms are represented explicitly.

Conflict-driven nogood learning (CDNL). CDNL-based ASP solving takes a ground program, translates it into nogoods and then runs a SAT-inspired (i.e. DPLL-style) model-building algorithm to find a solution for the set of nogoods. A nogood $\delta = \{s_1, \ldots, s_n\}$ is a set of boolean signed literals, which intuitively states that a solution cannot satisfy all literals $s_i \in \delta$. Nogoods are interpreted over assignments. A solution for a set Δ of nogoods then is an assignment A such that $\{a \mid \mathbf{T}a \in A\} \cap \{a \mid \mathbf{F}a \in A\} = \emptyset$, $\{a \mid \mathbf{T}a \in A\} \cup \{a \mid \mathbf{F}a \in A\} = grd(\mathcal{A})$, and no nogood is violated, i.e. for all $\delta \in \Delta$ it holds that $\delta \not\subseteq A$. A solution thus corresponds directly to an interpretation that is a model of all nogoods. For more details and algorithms, see (Gebser et al. 2012; Alviano et al. 2013; Dodaro et al. 2016). CDNL-based solvers use loop nogoods, source pointers, and nogoods establishing support to ensure that the constructed model is supported and unfounded-free.

Lazy grounding. Lazy grounding (or grounding on the fly), uses the notion of 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 (Dal Palù et al. 2009; Dao-Tran et al. 2012; Lefèvre et al. 2017). 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 $(A_0, \ldots, A_\infty)$ satisfies the following conditions, given the usual immediate-consequences operator T_P:

1. $A_0 = \emptyset$,
2. $\forall i \geq 1 : A_i \subseteq T_P(A_{i-1})$ (the computation contains only consequences),
3. $\forall i \geq 1 : A_{i-1} \subseteq A_i$ (the computation is monotonic),
4. $A_\infty = T_P(A_\infty)$ (the computation converges), and
5. $\forall i \geq 1 : \forall a \in A_i \setminus A_{i-1}, \exists r \in P$ such that $H(r) = a$ and $\forall j \geq i-1 : B^+(r) \subseteq A_j \wedge B^-(r) \cap A_j = \emptyset$ (persistence of reason).

Lefèvre and Nicolas (2009) showed 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 $A = A_\infty$. Thus, A is supported and unfounded-free without the need of employing loop nogoods and nogoods establishing support.

3 Lazy Grounding with CDNL Search

Alpha combines lazy grounding with CDNL search to avoid the grounding bottleneck of ASP and obtain very good search performance. In this section we briefly sketch the Alpha approach. CDNL-based ASP solvers require a fully grounded input, usually in the form of nogoods. Alpha provides this by having two dedicated components: a lazy grounder and a modified CDNL solver. This separation is common for pre-grounding ASP solvers, but for Alpha these components interact cyclically: whenever the solving component derives new truth assignments to atoms, the grounding component is queried for new ground nogoods obtainable by the new assignments. In contrast to traditional CDNL-based solving, the result of this interplay is a computation sequence as described above. 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. This is realised by creating a unique atom for each ground body and then guessing on these body-representing atoms (henceforth called *choice points*). These are of the form $\beta(r, \sigma)$, where r is a rule and σ is a substitution of the variables occurring in r by constants from $\mathcal{C}$.

Example 3.1
We illustrate the translation of rules to nogoods by means of the following ground rule r from Example 1.1: $\mathit{part}(\mathit{1}) \leftarrow n(\mathit{1}), \mathit{not}\ \mathit{no\_part}(\mathit{1})$. Let γ denote the new atom $\beta(r, \sigma)$ representing the choice point $B(r)$ with the substitution $\sigma = \{N \mapsto 1\}$. Then, the following nogoods are generated:

$$\{\mathbf{F}\mathit{part}(1), \mathbf{T}\gamma\} \qquad\qquad \{\mathbf{F}\gamma, \mathbf{T}n(1), \mathbf{F}\mathit{no\_part}(1)\}$$
$$\{\mathbf{T}\gamma, \mathbf{F}n(1)\} \qquad\qquad \{\mathbf{T}\gamma, \mathbf{T}\mathit{no\_part}(1)\}$$

The core algorithm of Alpha, presented in Algorithm 1, is generally in good agreement with the core algorithm of CDNL solvers. The search space is explored by one main loop, each iteration of which begins with the propagation of knowledge derived previously. If a conflict occurs, it is analyzed in (a) and a new nogood is derived following the first-UIP schema for conflict-driven learning (cf. Gebser et al. 2012). In (b) the grounder is requested to derive new nogoods from the assignment derived so far. In the current implementation, the lazy grounder computes all ground rules $r\sigma$ where the positive body is already derived, i.e., $B^+(r\sigma) \subseteq A$. The use of heuristics influencing the grounding component is a future issue as of now. In (c) a heuristic picks an atom and a truth value to assign. This guessing, which has been newly developed for Alpha, 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, i.e., there are no more applicable ground rule instances and nothing can be derived by propagation or from further grounding. There may be, however, still some atoms with unassigned truth value, because the guessing is restricted to choice points. Therefore in (d) all unassigned atoms are assigned false explicitly and the propagation at the beginning of the following iteration ensures that no nogood is violated. Finally, the solver checks in (e) whether there is an atom assigned to must-be-true, which is the case if an atom has not yet been derived to be true by some firing rule, but already decided to become true as otherwise some constraint would be violated. If there is no such atom, the current assignment corresponds to an answer set and is recorded. If the check fails

Algorithm 1: The Alpha algorithm (simplified pseudocode).

```
Input: A (non-ground) program $P$.
Output: The answer sets $\mathcal{AS}(P)$ of $P$.
$\mathcal{AS} \leftarrow \emptyset$
Initialize $A$ (assignment) and $\Delta$ (nogood storage).
Run lazy grounder (obtain inital nogoods $\Delta$ from facts).
while search space not exhausted do
    Propagate on $\Delta$ extending $A$.
    if exists conflicting nogood then                              (a)
        Analyze conflict, learn new nogood, and backjump.
    else if propagation extended $A$ then                          (b)
        Run lazy grounder wrt. $A$, extend $\Delta$.
        Get choice information from lazy grounder.
    else if exists an applicable rule then                         (c)
        Guess as chosen by heuristics.
    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 $\mathcal{AS}$
```

and some must-be-true assigned atom remains, the assignment is no answer set and backtracking occurs in (f). More details on Alpha are available in Weinzierl (2017).

4 Heuristics for Lazy-Grounding ASP Solving

Alpha takes ideas from state-of-the-art pre-grounding ASP solvers. Therefore it is natural to adopt heuristics from such systems as well. Note that we focus here on heuristics for the *solver* component only, while heuristics for the *grounder* in a lazy-grounding system are subject of future work. Heuristics for answer-set solving can roughly be classified as follows: *domain-independent heuristics* do not take the nature of the problem at hand into account, whereas *domain-specific heuristics* have to be tailored to a specific problem.

Domain-independent heuristics. For the first class, a prominent example is *BerkMin* (Goldberg and Novikov 2002), which is a domain-independent heuristic originally developed for SAT but also successfully employed by ASP solvers (such as CLASP (Gebser et al. 2012) and WASP (Alviano et al. 2013)). It organizes the set of conflict clauses as a chronologically ordered stack, thereby preferring variables in recent conflicts. This is done to regard the fact that the set of variables responsible for conflicts may change very quickly. Additionally, a so-called *activity* counter is assigned to every variable that counts the number of clauses involving this variable that are responsible for at least one conflict. These counters are divided by a constant ("decayed") periodically to reduce the influence of "aged" clauses. When BerkMin is asked for an atom, it chooses the most active unassigned atom in the nogood nearest to the top of the stack that is not yet fully assigned. Other counters are maintained as well to enable the choice which truth value to assign.

BerkMin is an improvement over *VSIDS* (Variable State Independent Decaying Sum, cf. Moskewicz et al. 2001), which introduced the idea of periodically reducing the impact of older conflict clauses.

Domain-specific heuristics. One representative of this class of heuristics is HCLASP (Gebser et al. 2013), a declarative framework to incorporate domain-specific heuristics into ASP solving. This is achieved by providing a dedicated predicate which is grounded in the same way as other predicates but which is then processed by the solver in a special way, thereby modifying the built-in solver heuristics. This heuristic language is tailored towards CDNL and allows to influence both the heuristic to choose unassigned literals and the heuristic to choose the truth value to assign. Experiments on various planning benchmarks showed promising results, but also that each problem class needs a customized heuristic to achieve good performance.

Another approach is HWASP (Dodaro et al. 2016), an extension of WASP providing an interface for external heuristics specified in languages like `Python` or `C++`.

BerkMin in a lazy-grounding solver. A direct application of *BerkMin* to a lazy-grounding ASP solver like Alpha seems unnatural, because such a solver differs in many important ways from a solver adhering to the classical ground-and-solve paradigm. One major difference is that not all ground rules, and consequently not all ground literals and atoms, are known at any time to a lazy-grounding solver. Because of this, a *BerkMin*-like heuristic applied to lazy grounding can only incorporate atoms that are already known to the solver.

Another major difference lies in the solving mechanism: while a traditional ASP solver can choose any atom to guess on, Alpha is restricted to atoms representing rule bodies. In other words, Alpha only guesses whether a certain rule fires or not, but it does not guess whether an atom in a rule's head or body is true or not.

4.1 Domain-Independent Heuristics in Alpha

In the main algorithm of Alpha, shown in Algorithm 1, branching heuristics come into play in (c), where they are used to choose both a body-representing atom and a truth value to assign to this atom. In the following paragraphs we describe branching heuristics implemented in Alpha so far. Abbreviations in paragraph headings will be used later to identify the heuristics in the experiments in Section 5.

A naive heuristic (N). The naive heuristic implemented in Alpha, to which all other heuristics will be compared to, is implemented as follows: Active choice points are organized as a FIFO queue, i.e. the earlier a choice point is grounded the earlier it is chosen to be guessed on. The truth value is always chosen to be true.

A BerkMin-inspired heuristic (BMI). Our initial application of BerkMin to Alpha looks as follows: To choose an atom, iterate through the stack of nogoods until one is encountered that contains literals that are not yet assigned, which are then transformed into atoms (by removing their sign). From these atoms, the most active choice point is then chosen (i.e., the atom with the highest activity counter that represents a choice point). To choose the truth value, we implemented a slightly modified version of BerkMin's procedure: If the atom is already assigned *must-be-true*, we assign true. Otherwise, if the atom occurs more often positively in nogoods, we assign false, else we assign true. The goal of this is to violate as few nogoods as possible. If this does not lead to a decision, we choose the sign randomly.

BMI with a bounded queue of choice points (BBQ). BMI is still inefficient because it maintains a stack of nogoods even though only a small subset of the atoms in these nogoods can be considered as valid choice points. In fact, choosing the most active choice point in a nogood is mostly useless because most if not all nogoods will contain only a single literal representing a rule body. Therefore, *BBQ* manages a bounded FIFO queue (of length 32) of choice points occurring in recent nogoods. When new nogoods arrive (by grounding or learning), the oldest choice points leave the queue. The branching heuristic then chooses the most active atom from this queue, disregarding the order in which they appear. The truth value is chosen as in *BMI*.

A dependency-driven heuristic (DD). The BerkMin variants *BMI* and *BBQ* described so far suffer from the fact that choice points comprise only a small portion of all the literals occuring in nogoods and therefore do not influence activity and sign counters as much as other atoms. The basic idea of *DD* is therefore to find *dependent* atoms that can enrich the information available for a choice point. Intuitively, all atoms occurring in $\{H(r\sigma)\} \cup B(r\sigma)$ depend on a choice point $\gamma = \beta(r, \sigma)$.

DD manages a stack of nogoods similar as *BMI* and starts by looking at the most active atom a in the nogood currently at the top of the stack. If a is an active choice point (i.e. representing the body of an applicable rule), it is immediately chosen; else the most active choice point dependent on a is. If there is no such atom, *DD* continues further down the stack. Several notions of activity for dependent atoms can be employed to define the activity of a choice point γ: by default, *DD* compares choice points by their activity as computed by BerkMin. *DD-Sum* sums up the activities of all the atoms contained by the body represented by γ. *DD-Avg* computes the average, *DD-Max* the maximum and *DD-Min* the minimum of the same activity values. *DD* uses the *BMI* method to choose a sign, but employs it on the head of the rule whose body is chosen, not on the choice atom itself.

A generalized dependency-driven heuristic (GDD). DD needs to know a lot about the structure of nogoods, i.e. the role their members play in rules. However, the solving component of Alpha usually deals only with nogoods and cannot naturally access this information. Therefore, *GDD* both generalizes and simplifies *DD* by redefining the set of atoms dependent on a choice point γ. This set is constructed by adding to it, every time a new nogood containing a choice point γ is added to the stack, all other atoms in the nogood. To choose an atom, *GDD* then proceeds as *DD*. Variants like *GDD-Sum*, *GDD-Avg*, *GDD-Max* and *GDD-Min* are constructed similarly. Because *GDD* does not know the head belonging to γ, it just uses the *BMI* method to choose a truth value.

Pyromaniacal variants (DD-Pyro and GDD-Pyro). One weakness of Alpha that has to be addressed in future work is its lack of support nogoods (except in one special case, cf. Weinzierl 2017). This means that the solver cannot recognize when an atom occurring negatively in a rule body cannot be satisfied anymore, thus exploring large portions of the search space in search for a witness. Therefore, Alpha currently benefits in many cases from heuristics that assign true to choice points whenever they can, as will be shown in the experiments in Section 5. We apply this modification to *DD* and *GDD*, which then always assign true first and try false only when backtracking, and call these variants *pyromaniacal* since they prefer the firing of a rule over its non-firing.

4.2 Domain-Specific Heuristics in Alpha

Often the problem at hand contains subproblems for which well-studied domain-specific heuristics are known. For instance, a well-known heuristic for graph-colouring problems is to explore vertices with higher degrees first; for bin-packing problems, it is better to pack large items first; etc. In CSP solvers these heuristics can be expressed by supplying specific variable and value orderings for a selected set of variables (van Beek 2006). In Alpha we can implement such heuristics by influencing the order in which the solver chooses applicable rules.

Domain-specific heuristics are not only a means for improving the performance of a solver but also for controlling in which order answer sets will be found by the solver. In that respect domain-specific heuristics can be used to express preferences. For example in the case of product configuration problems it is often preferable to find solutions that contain as few parts as possible, therefore a suitable domain-specific heuristic would be to reuse existing parts whenever possible.

The heuristics to use depend also on the capabilities of the solver. Most traditional heuristics lead a solver in the direction of a solution. A solver with conflict-driven nogood learning on the other hand may benefit from a heuristic that leads to a conflict first, because this will prune a large portion of the search space.

We have implemented domain-specific heuristics in ALPHA using a hybrid approach involving ASP and `Java`, the language that ALPHA is implemented in. For every domain-specific heuristic a new `Java` class must be implemented. Additional ASP code specific to the heuristic can be written to derive further information supporting which decision to make (similarly to the *_heuristic* predicate in HCLASP (Gebser et al. 2013)). When the solver decides which rule to fire next, it delegates

Algorithm 2: Connected graph colouring heuristic

```
Data: Set $R$ of applicable rules.
Result: A chosen rule $r \in R$.
$VCR \leftarrow$ rules from $R$ with $H(r) = vertex\_colour(V, C)$
foreach $r$ in $VCR$ do use neighbour for usable color:
    if $V$ has neighbor $N$ with colour(N)=C then
        return $r$
foreach $r$ in $VCR$ do use any vertex for new color:
    if $C$ is unused then
        return $r$
return default_heuristic(R)
```

the decision to the domain-specific class. Based on syntactic features the heuristic then decides which rule to fire next, taking into consideration the current state of the search. For example, in the case of a classical graph colouring problem the heuristic will identify all rules deriving a decision about the colour of a vertex (e.g. $vertex\_colour/2$) and choose the preferred one, e.g. the rule deriving a colour for a vertex with high degree (variable order in CSP) and with the smallest possible value (value order in CSP). As a proof of concept we implemented a heuristic to solve the connectedness problem of the *Combined Configuration Problem* (cf. Gebser et al. 2015). The problem consists of colouring the vertices of a given graph such that there is always a path between two vertices with the same colour, i.e. the subgraph of same-coloured vertices must be connected. There is an upper limit on the number of vertices that can be coloured with the same colour.

Without any domain-specific heuristics, Alpha currently can solve only small instances of this problem (10 vertices). One limiting factor is that cardinality constraints, which could be used to efficiently check the upper bound on the number of used colours, are not supported by Alpha yet. To remedy this, we removed these constraints from the ASP program and added them to the domain-specific heuristic. With the heuristic in Algorithm 2, Alpha can solve graphs with more than 1,000 vertices within a minute on a standard personal computer. This comes at no surprise as the heuristic in this case simulates a greedy algorithm within an ASP solver. As has been shown by Gebser et al. (2015), this combination of a greedy heuristic and a complete solver can improve the performance of an ASP solver significantly.

5 Experiments

We investigated the feasibility of our domain-independent heuristics experimentally on an Intel Core i5-4300M CPU at 2.6 GHz, with 8 GB of RAM and Windows 7. The following problems were used in the experiments:

Configuration is the configuration problem presented in Example 1.1. Instances are denoted by the value of N.

Pigeon is the problem of assigning p pigeons to h different holes so that there is at most one pigeon in a hole, using the encoding given by Niemelä (1999). We

Table 1. Experimental results (median number of choices)

	Instance	N	BMI	BBQ	DD	DD-Pyro	GDD	GDD-Pyro
configuration	2	2	2	2	2	6	2	6
	4	32	∞	∞	21	4	∞	4
	8	8	8	8	8	86	8	102
	16	512	∞	131,104	1,872	16	∞	16
	32	1,024	1,024	∞	12,037	32	∞	32
	75	5,625	5,625	∞	∞	75	∞	75
pigeon	$10 \mid 10$	∞	∞	2,700	546	482	546	482
	$19 \mid 20$	∞	∞	∞	2,310	2,469	2,315	2,760
	$28 \mid 30$	866	5,223	∞	9,553	9,065	9,553	9,065
wheel	3	6	37	37	8	7	37	7
	7	18	2,009	663	492	25	1,733	25
	11	30	24,045	45,915	50,267	42	119,702	42

only consider cases where $p \leq h$, i.e. where there exists at least one answer set[2]. Instances are denoted by $p|h$.

Wheel is a three-colouring problem on a graph with n vertices organized as a bicycle wheel, as given by Lefèvre et al. (2017). Again we only consider instances where there exists at least one answer set, i.e. where n is odd.

The performance of all heuristics presented so far depends on two external factors: a seed used to initialize a pseudorandom number generator, and the order of rules in the input program. The latter is the case because nogoods get known to the heuristic in the order they are produced by the grounder, which processes the input sequentially. To compensate for such effects, each test case was repeated nine times, using three different seeds and three different orderings of the input program[3]. Median numbers of choices over these nine runs per problem instance and heuristic are reported in Table 1. In cells containing ∞, the time-out of 60 seconds was reached in the majority of runs for the respective problem instance and heuristic. Due to space restrictions, results for variants of *DD* and *GDD* (except for the pyromaniacal variants) are not shown. It could be observed that they behave very similarly to the main variants of these heuristics in most cases.

Ongoing and future work will extend our experimental analysis. In addition to numbers of choices reported here, additional data like numbers of conflicts, time and space requirements, and statistical data like standard deviations will be recorded by means of a controlled benchmarking environment on dedicated hardware. It would also be interesting to compare Alpha to other lazy-grounding systems as well as to traditional systems. In addition, we plan to incorporate a larger set of benchmark problems in our experiments. Preliminary data is available on our website[4], while comprehensive results are to be published in the near future.

[2] All our heuristics take ca. the same number of choices to prove that a program is unsatisfiable.
[3] The input program was used in its default order, in reverse order, and in a shuffled order produced using a pseudorandom number generator.
[4] `http://www.kr.tuwien.ac.at/research/systems/alpha/`

6 Discussion

This paper describes a work in progress. Although the heuristics presented are still under development and only a brief experimental study has been conducted, promising results can be seen. The novel family of dependency-driven heuristics (*DD*, *GDD*, etc.) was repeatedly able to outperform Alpha's naive heuristic (N) as well as the two BerkMin-inspired heuristics (*BMI* and *BBQ*). *DD* performed significantly better than the more general *GDD*. Apparently it pays off to exploit the structure of rules, even though a purely nogood-based solver would usually ignore this. The pyromaniacal variants of *DD* seem exceptionally suited to support Alpha's current decision procedure: they performed considerably above average and beat all other heuristics more often than not.

The strong results of the pyromaniacal heuristics on the *Configuration* problem can surely be attributed to the underconstrained nature of this problem. Surprisingly, these heuristics are not able to beat even the naive one on one of the pigeon cases and all three wheel instances. While evidence suggests that all non-naive heuristics are more or less competitive also on more constrained problems like *Pigeon* and *Wheel*, there is obviously more work to be done to improve the performance of these heuristics and the solver in general.

The surprising stability of the naive heuristic clearly originates in its similarity to the pyromaniacal heuristics. Their shared characteristic of invariably guessing true first clearly favours Alpha's algorithm, which currently cannot recognize lack of support as efficiently as pre-grounding systems can. Therefore, future improvements in this area will probably alleviate the pyromaniacs' lead.

It is peculiar that the "hardness" of problems sometimes does not monotonically increase with the problem size, e.g. the *Configuration* instance 4 seems to be especially hard for some heuristics. A thorough analysis of the solver's behaviour will be necessary to determine the cause of this.

7 Conclusion

To improve the performance of lazy-grounding ASP solvers, we have developed novel domain-independent and domain-specific branching heuristics. To the best of our knowledge, this investigation is the first of its kind. Early experimental results on the CDNL-based lazy-grounding system Alpha demonstrate a basic feasibility. Novel dependency-driven heuristics were commonly able to outperform Alpha's default heuristic as well as two heuristics inspired by BerkMin. Variants that prefer the firing of a rule over its non-firing are especially advantageous. Our techniques are in principle applicable to other lazy-grounding solvers as well, although implications from the dependency on CDNL remain to be identified.

While our results are encouraging, many opportunities to vary the heuristics still exist. The focus so far was on heuristics steering the solver, heuristics driving the grounding are future work. They could determine how much to ground and when to forget parts of the grounding to save space. This work will be followed by a more detailed study on a larger set of benchmark problems and measurements.

References

Alviano, M., Dodaro, C., Faber, W., Leone, N., and Ricca, F. 2013. WASP: A native ASP solver based on constraint learning. In *LPNMR*. LNCS, vol. 8148. Springer, 54–66.

Calimeri, F., Fuscà, D., Perri, S., and Zangari, J. 2016. $\mathcal{I}$-dlv: The new intelligent grounder of dlv. In *AI*IA*. LNCS, vol. 10037. Springer, 192–207.

Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. 2009. GASP: Answer set programming with lazy grounding. *Fundamenta Informaticae 96,* 3, 297–322.

Dao-Tran, M., Eiter, T., Fink, M., Weidinger, G., and Weinzierl, A. 2012. OMiGA: An open minded grounding on-the-fly answer set solver. In *JELIA*. LNAI, vol. 7519. Springer, 480–483.

de Cat, B., Denecker, M., Bruynooghe, M., and Stuckey, P. J. 2015. Lazy model expansion: Interleaving grounding with search. *J. Artif. Intell. Res. (JAIR) 52*, 235–286.

Dodaro, C., Gasteiger, P., Leone, N., Musitsch, B., Ricca, F., and Schekotihin, K. 2016. Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). *TPLP 16,* 5-6, 653–669.

Dovier, A., Formisano, A., Pontelli, E., and Vella, F. 2016. A GPU implementation of the ASP computation. In *PADL*. LNCS, vol. 9585. Springer, 30–47.

Faber, W., Leone, N., and Perri, S. 2012. The intelligent grounder of dlv. In *Correct Reasoning*. LNCS, vol. 7265. Springer, 247–264.

Gebser, M., Kaminski, R., König, A., and Schaub, T. 2011. Advances in gringo series 3. In *LPNMR*. LNCS, vol. 6645. Springer, 345–351.

Gebser, M., Kaufmann, B., Otero, R., Romero, J., Schaub, T., and Wanko, P. 2013. Domain-specific heuristics in answer set programming. In *AAAI Conference on Artificial Intelligence*. AAAI Press, 350–356.

Gebser, M., Kaufmann, B., and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187-188*, 52–89.

Gebser, M., Ryabokon, A., and Schenner, G. 2015. Combining heuristics for configuration problems using answer set programming. In *LPNMR*. LNAI, vol. 9345. Springer International Publishing, 384–397.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *JICSLP*. MIT Press, 1070–1080.

Goldberg, E. and Novikov, Y. 2002. Berkmin: A fast and robust SAT-solver. In *Design, Automation and Test in Europe Conference and Exhibition, 2002*. IEEE, 142–149.

Lefèvre, C., Béatrix, C., Stéphan, I., and Garcia, L. 2017. ASPeRiX, a first-order forward chaining approach for answer set computing. *TPLP 17,* 3, 266–310.

Lefèvre, C. and Nicolas, P. 2009. A first order forward chaining approach for answer set computing. In *LPNMR*. LNAI, vol. 5753. Springer, 196–208.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. 2001. Chaff: engineering an efficient SAT solver. In *Design Automation Conference*. IEEE, 530–535.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell. 25,* 3/4, 241–273.

van Beek, P. 2006. Backtracking search algorithms. In *Handbook of constraint programming*, F. Rossi, P. van Beek, and T. Walsh, Eds. Elsevier, 85–134.

Weinzierl, A. 2013. Learning non-ground rules for answer-set solving. In *2nd Workshop on Grounding and Transformations for Theories With Variables.*

Weinzierl, A. 2017. Blending lazy-grounding and CDNL search for answer-set solving. In *LPNMR*. To appear, preprint available at `http://www.kr.tuwien.ac.at/research/systems/alpha/blending_lazy_grounding.pdf`.