TPLP: Page 1–18.

doi:10.1017/S1471068425100173

Automated Hybrid Grounding Using Structural and Data-Driven Heuristics

ALEXANDER BEISER and STEFAN WOLTRAN
TU Wien, Vienna, Austria
(*e-mails:* alexander.beiser@tuwien.ac.at, woltran@dbai.tuwien.ac.at)

MARKUS HECHER
CNRS, Computer Science Research Center of Lens (CRIL), Univ. Artois, Lens, France
(*e-mail:* hecher@cril.fr)

submitted 24 July 2025; revised 24 July 2025; accepted 27 July 2025

Abstract

The grounding bottleneck poses one of the key challenges that hinders the widespread adoption of answer set programming in industry. Hybrid grounding is a step in alleviating the bottleneck by combining the strength of standard bottom-up grounding with recently proposed techniques where rule bodies are decoupled during grounding. However, it has remained unclear when hybrid grounding shall use body-decoupled grounding (BDG) and when to use standard bottom-up grounding. In this paper, we address this issue by developing automated hybrid grounding: we introduce a splitting algorithm based on data-structural heuristics that detects when to use BDG and when standard grounding is beneficial. We base our heuristics on the structure of rules and an estimation procedure that incorporates the data of the instance. The experiments conducted on our prototypical implementation demonstrate promising results, which show an improvement on hard-to-ground scenarios, whereas on hard-to-solve instances, we approach state-of-the-art performance.

KEYWORDS: logic programming, answer set programming, grounding, grounding bottleneck, hybrid grounding, body-decoupled grounding

1 Introduction

The so-called *grounding bottleneck* (Gebser *et al*. 2018; Tsamoura *et al*. 2020) in answer set programming (ASP) is one of the key factors that hinders large-scale adoption of ASP in the industry (Falkner *et al*. 2018). It occurs as part of the grounding step (Kaminski and Schaub 2023), which is an integral part of the state-of-the-art (SOTA) ASP systems, such as `clingo` (Gebser *et al*. 2016) or `dlv` (Leone *et al*. 2006). Grounding replaces the variables of a non-ground ASP program by their domain values, which inherently results in an exponentially larger (Dantsin *et al*. 2001) ground program.

The grounding bottleneck is a long-standing problem, which is the reason why modern grounders like `gringo` (Gebser *et al*. 2015) or `idlv` (Calimeri *et al*. 2017), are highly

optimized systems. They work according to a bottom-up and semi-naive approach (Gebser *et al.* 2015), which instantiates rules along their occurrence on the topological order of the dependency graph of the program. Although these systems are highly optimized and implement advanced rewriting methods, as they incorporate structural information on rules (Bichler *et al.* 2016; Calimeri *et al.* 2018), they are exponential in the number of variables in the worst case.

Body-decoupled grounding (BDG) (Besin *et al.* 2022) is a novel approach that alleviates the grounding bottleneck by decomposing rules into literals and grounding the literals individually. This is achieved by shifting some of the grounding effort from the grounder to the solver, thereby exploiting the power of modern ASP solving technology. Practically, BDG's grounding size is only dependent on the maximum arity a of a program. Experiments on grounding-heavy tasks have shown promising results, by solving previously ungroundable instances. However, BDG on its own is not interoperable with other SOTA techniques, which prohibits BDG from playing to its strengths in practical settings. Hybrid grounding (Beiser et al., 2024) partially alleviates the challenge of interoperability, by enabling the free (manual) partitioning of a program Π into a part $\Pi_{\mathscr{H}}$ grounded by BDG and $\Pi_{\mathscr{G}}$ grounded by bottom-up grounding.

Still, it remains unclear when the usage of BDG is beneficial. Grounding with BDG potentially increases the solving time, as BDG pushes effort spent in grounding to solving. Rewriting techniques, used for example in `idlv`, complicate this matter further. Additionally, BDG's grounding size is solely dependent on the domain, not considering the peculiarities of the instance. We address this challenge by introducing *automated hybrid grounding*, which is an algorithm for detecting those parts of a program that shall be grounded by BDG. Our contributions are three-fold:

- We present the data-structural splitting heuristics, which decides (based on the structure of a rule and the instance's data) whether it is beneficial to ground with BDG.
- We develop the prototype `newground3` that integrates BDG into bottom-up procedures of SOTA grounders and uses BDG according to data-structural heuristics.
- Our experiments show that with `newground3` we approach SOTA performance on solving-heavy scenarios, while beating the SOTA on grounding-heavy scenarios.

The paper is structured as follows. After this introduction (Section 1), we state the necessary preliminaries of ASP and on grounding techniques (Section 2). We continue by showing our data-structural heuristics (Section 3). Next is the high-level description of our prototypical implementation `newground3` (Section 4), which is followed by the conducted experiments (Section 5). The paper ends with a conclusion and discussion (Section 6).

Related work. While SOTA grounders use semi-naive grounding techniques (Gebser *et al.* 2016; Calimeri *et al.* 2017), we focus on the interoperability between SOTA grounders and alternative grounding procedures. Alternative grounding procedures include lazy-grounding (Weinzierl 2017; Weinzierl *et al.* 2020), lazy-grounding with heuristics (Leutgeb and Weinzierl 2018), compilation-based techniques via lazy rule injection (Cuteri *et al.* 2019; Lierler and Robbins 2021), or compilation-based techniques via extensions of the CDNL procedure (Mazzotta *et al.* 2022; Dodaro *et al.* 2023, 2024). Approaches based on ASP Modulo Theory combine ASP with methods from other fields

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

(Liu *et al*. 2012; Banbara *et al*. 2017; Balduccini and Lierler 2017). Structure-based techniques also showed promising results (Bichler *et al*. 2016). We focus on the alternative grounding procedure of BDG (Besin *et al*. 2022). In contrast to the other approaches, BDG is a rewriting approach based on complexity theory. In Beiser et al. (2024) BDG was extended by hybrid grounding and the handling of aggregates. Hybrid grounding enables the free partitioning of a program into a part grounded by semi-naive grounding and a part grounded by BDG. Aggregates are handled by specially crafted rewriting procedures that decouple aggregates. We extend the previous work on BDG by proposing a splitting heuristics that decides when the usage of BDG is useful. Further, we provide an extensive empirical evaluation of the heuristics with our prototype `newground3`. Previously proposed splitting heuristics include heuristics on when to use bottom-up grounding and when to use structural rewritings (Calimeri *et al*. 2018). Related work proposes a machine learning-based heuristics (Mastria *et al*. 2020). In contrast, we focus on a splitting heuristics, when the usage of BDG is beneficial.

2 Preliminaries

Ground ASP. A ground program P consists of ground rules of the form $a_1 \vee \ldots \vee a_l \leftarrow a_{l+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$, where a_i are propositional atoms and l, m, n are non-negative integers with $l \leq m \leq n$. We let $H_r := \{a_1, \ldots, a_l\}$, $B_r^+ := \{a_{l+1}, \ldots, a_m\}$, $B_r^- := \{a_{m+1}, \ldots, a_n\}$, and $B_r := B_r^+ \cup B_r^-$. $r \in P$ is normal iff $|H_r| \leq 1$, a constraint iff $|H_r| = 0$, and disjunctive iff $|H_r| > 1$. The *dependency graph* $\mathscr{D}$ is the directed graph $\mathscr{D} = (V, E)$, where $V = \bigcup_{r \in P} H_r \cup B_r$ and $E = \{(b,h)_+ | r \in P, b \in B_r^+, h \in H_r\} \cup \{(b,h)_- | r \in P, b \in B_r^-, h \in H_r\}$. We refer by $(b,h)_+$ to a positively labeled edge and by $(b,h)_-$ to a negatively labeled edge. A positive cycle consists solely of positive edges. A program P is *tight* iff there is no positive cycle in $\mathscr{D}$, P is not stratified iff there is a cycle in $\mathscr{D}$ that contains at least one negative edge, and P is *head-cycle-free (HCF)* iff there is no positive cycle in $\mathscr{D}$ among any two atoms $\{a, b\} \subseteq H_r$. *IsConstraint(r)* is true iff r is a constraint.

We proceed by defining the semantics of ASP. Let $\mathrm{HB}(P)$ be the Herbrand Base (the set of all atoms). For ground programs this is $\mathrm{HB}(P) = \{p \mid r \in P, p \in H_r \cup B_r\}$. An *interpretation* I is a set of atoms $I \subseteq \mathrm{HB}(P)$. I *satisfies* a rule r iff $(H_r \cup B_r^-) \cap I \neq \emptyset$ or $B_r^+ \setminus I \neq \emptyset$. I is a *model* of P iff it satisfies all rules of P. A rule $r \in P$ is *suitable for justifying* $a \in I$ iff $a \in H_r$, $B_r^+ \subseteq I$, and $I \cap B_r^- = I \cap (H_r \setminus \{a\}) = \emptyset$. A *level mapping* $\psi : I \to \{0, \ldots, |I| - 1\}$ assigns every atom in I a unique value (Lin and Zhao 2003; Janhunen 2006). An atom $a \in I$ is *founded* iff there is a rule $r \in P$ s.t. (i) r is suitable for justifying a and (ii) there are no cyclic-derivations, that is $\forall b \in B_r^+ : \psi(b) < \psi(a)$. I is an *answer set* of a normal (HCF) program P iff I is a model (satisfied) of P, and all atoms in I are founded. The *Gelfond-Lifschitz (GL) reduct* is the classical way to define semantics. The GL reduct of P under I is the program P^I obtained from P by first removing all rules r with $B_r^- \cap I \neq \emptyset$ and then removing all $p \in B_r^-$ from the remaining rules r (Gelfond and Lifschitz 1991). I is an *answer set* of a program P if I is a *minimal model* (w.r.t. $\subseteq$) of P^I.

Non-ground ASP. A non-ground program Π consists of non-ground rules r of the form $p_1(\mathbf{X}_1) \vee \ldots \vee p_\ell(\mathbf{X}_\ell) \leftarrow p_{\ell+1}(\mathbf{X}_{\ell+1}), \ldots, p_m(\mathbf{X}_m), \neg p_{m+1}(\mathbf{X}_{m+1}), \ldots, \neg p_n(\mathbf{X}_n)$, where each $p_i(\mathbf{X}_i)$ is a literal and l, m, n are non-negative integers s.t. $l \leq m \leq n$. A literal $p_i(\mathbf{X}_i)$ consists of a *predicate* p_i and a *term* vector $\mathbf{X}_i = \langle x_1, \ldots, x_z \rangle$. A *term* $x_j \in \mathbf{X}_i$ is a constant or a variable. For a predicate p_i let $|\mathbf{X}_i|$ be its arity $a(p_i) = |p_i| = |\mathbf{X}_i|$, and

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

for a rule $r \in \Pi$, let $a = \max_{p(\mathbf{X}) \in H_r \cup B_r} |\mathbf{X}|$ be the maximum arity. IsVar(x) evaluates to true iff the term x is a variable. We furthermore define var$(r) := \{x \mid x \in \mathbf{X}, p(\mathbf{X}) \in H_r \cup B_r, \text{IsVar}(x)\}$. For non-ground rules we define H_r, B_rô+, B_rô-, and B_r as in the ground case, as we do with the attributes *disjunctive*, *normal*, *constraint*, *stratified*, *tight*, and *HCF*. The size of a rule is $|r| = |H_r \cup B_r|$ and of a program $|\Pi| = \sum_{r \in \Pi} |r|$. Grounding is the instantiation of the variables by their domain. Let $\mathscr{F} = \{p(\mathbf{D}) \mid p(\mathbf{D}) \in \Pi, \forall d \in \mathbf{D} : \neg\text{IsVar}(d)\}$ be the facts and $\text{dom}(\Pi) = \{d \mid p(\mathbf{D}) \in \mathscr{F}, d \in \mathbf{D}\}$ be the domain. Let x be a variable, then $\text{dom}(x) = \text{dom}(\Pi)$. *Naive grounding* $\mathscr{G}_N(\Pi)$ instantiates for each rule all variables by all possible domain values, which results in a grounding size in $\mathscr{O}\left(|\Pi| \cdot |\text{dom}(\Pi)|^{\max_{r \in \Pi} |\text{var}(r)|}\right)$. For non-ground programs the herbrand base HB(Π) is defined as $\text{HB}(\Pi) = \{p(\mathbf{D}) \mid r \in \mathscr{G}_N(\Pi), p(\mathbf{D}) \in H_r \cup B_r\}$. The semantics of a non-ground program Π is defined over its ground version $\mathscr{G}_N(\Pi)$ and carries over from the ground case.

The non-ground dependency graph $\mathscr{D}_\Pi$ of the non-ground program Π carries over from the ground case and is defined over the predicates. $SCC(\Pi)$ refers to the set of *strongly-connected components (vertices)* of $\mathscr{D}_\Pi$. A reduced graph $\mathscr{D}_R(G)$ of a graph $G = (V, E)$ is $\mathscr{D}_R(G) = (V_r, E_r)$, where $V_r = SCC(G)$ and $E_r = \{(s_1, s_2) \mid s_1, s_2 \in SCC(G), s_1 \neq s_2, \exists v_1 \in s_1 \exists v_2 \in s_2 : (v_1, v_2) \in E\}$. Any reduced graph is a directed acyclic graph (DAG). Let p be a predicate and L_Π be a topological order of the reduced dependency graph $\mathscr{D}_R(\mathscr{D}) = (V_r, E_r)$ and let $SCC_\Pi(p)$ be the function $SCC_\Pi(p) : V \to V_r$ that returns the corresponding SCC of p, that is $SCC_\Pi(p) = s$ s.t. $s \in SCC(\Pi)$ and $p \in S$. Let $s = SCC_\Pi(p)$ and $S_{\prec p}(0) = \{s\}$. We iteratively extend $S_{\prec p}$ to a fixed point by $S_{\prec p}(t+1) = \{s | s \in SCC(\Pi), \exists s' \in S_{\prec p}(t) : (s, s') \in E_r\} \cup S_{\prec p}(t)$ for $t > 0$. A fixed point is reached when $S_{\prec p}(t+1) = S_{\prec p}(t)$, which we denote as $S_{\prec p} = S_{\prec p}(t)$. As $\mathscr{D}_R(G)$ is a DAG, such a fixed point always exists (Knaster 1928; Tarski 1955). A predicate p is stratified iff $\forall s \in S_{\prec p}$, there is no cycle with at least one negative edge in s. Further, let *IsStratified(r)* be true iff r contains (only) stratified body predicates $p \in B_r$. Let *IsTight(r)* be true iff $\forall h \in H_r : \forall p \in B_r^+ : SCC_\Pi(h) \neq SCC_\Pi(p)$ - so r occurs in a tight part. The variable graph $\mathscr{D}(r) = (V, E)$ for a rule $r \in \Pi$ is defined as the undirected graph where $V = \text{var}(r)$ and $E = \{(x_i, x_j) \mid x_i, x_j \in \text{var}(r), \exists p(\mathbf{X}) \in H_r \cup B_r : \{x_i, x_j\} \subseteq \mathbf{X}\}$. A tree decomposition (TD) $\mathscr{T} = (T, \chi)$ is defined over an undirected graph $G = (V, E)$ where T is a tree and χ a labeling function $\chi : T \to V$. $\chi(t) \subseteq V$ is called a bag. A TD must fulfill: (i) $\forall v \in V \exists t \in T : v \in \chi(t)$, (ii) $\forall (u, v) \in E \exists t \in T : \{u, v\} \subseteq \chi(t)$, and (iii) every occurrence of $v \in V$ must form a connected subtree in T w.r.t. χ, so $\forall t_1, t_2, t_3 \in T$, s.t. whenever t_2 is on the path between t_1 and t_3, it must hold $\chi(t_1) \cap \chi(t_3) \subseteq \chi(t_2)$. The width of a TD is defined as the largest cardinality of a bag minus one, so $\max_{t \in T} |\chi(t)| - 1$. The treewidth (TW) is the minimal width among all TDs. Further, let φ_r denote the bag size of a minimal TD of the variable graph of r.

Bottom-up/Semi-naive grounding. Grounders `gringo` and `idlv` use (bottom-up) semi-naive database instantiation techniques to ground a program Π (Gebser *et al.* 2016; Calimeri *et al.* 2017). In the following, we sketch the intuition. Let L_Π be a topological order of $G_R(\mathscr{D}_\Pi)$, and let D be the *candidate set*, where $D \subseteq \text{HB}(\Pi)$; initially $D = \mathscr{F}$. Intuitively, the candidate set D keeps track of all possibly derivable literals and is iteratively expanded by moving along the topological order L_Π. For each $v \in L_\Pi$ rules are instantiated according to the candidate set D by a fixed-point algorithm. If a tuple is in D it is possibly true, conversely, if a tuple is not in D, it is surely false. If an SCC contains

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

a cycle, semi-naive techniques are used to prevent unnecessary derivations (Gebser *et al.* 2016; Calimeri *et al.* 2017). The grounding size is exponential in the maximum number of variables $\mathcal{O}\left(\sum_{r\in\Pi} |\mathrm{dom}(\Pi)|^{|\mathrm{var}(r)|}\right)$ in the worst-case. We use the terms *SOTA*, traditional, bottom-up, or semi-naive grounding interchangeably.

Bottom-up grounding solves stratified programs. Bottom-up grounding is typically implemented in a way that enables full evaluation of stratified programs. Technically, this is implemented by partitioning the candidate set D into a surely derived set D_T and a potentially derived set D_{pot}. Conversely, for any $a \in \mathrm{HB}(\Pi)$, but $a \notin D_{pot} \cup D_T$, we know that we can never derive a. This split leads to a series of improvements related to instantiating rules, among them is the full evaluation of stratified programs. However, these improvements have no effect on the grounding size of non-stratified programs in the worst case, thereby remaining exponential in the variable number.

Structure-aware rewritings. Utilizing the rule structure to rewrite non-ground rules is performed by `Lpopt` (Morak and Woltran 2012; Bichler *et al.* 2016). It computes a minimum size TD, which is then used to introduce fresh rules with a preferably smaller grounding size. In more detail, for every rule $r \in \Pi$ `Lpopt` first creates the variable graph $\mathscr{D}(r)$. After computing a minimum-size TD, it introduces fresh predicates and fresh rules for every bag of the TD. The arity of the fresh predicates corresponds to the respective bag size, as does the number of variables per rule. Let $TW(\mathscr{D}(r))$ be the maximum TW of all rules $r \in \Pi$, then $\varphi_r = TW(\mathscr{D}(r)) + 1$ is its bag size. It was shown that `Lpopt` produces a rewriting that is exponential in φ_r, where $\varphi_r \leq \max_{r\in\Pi} |var(r)|$: $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^{\varphi_r})$. Internally, `idlv` uses the concepts of `Lpopt` to reduce the grounding size (Calimeri *et al.* 2018).

Body-decoupled Grounding. BDG (Besin *et al.* 2022) produces grounding sizes that are exponential only in the maximum arity. Conceptually, BDG decouples each rule into its literals which are subsequently grounded. As each literal has at most arity-many variables, its grounding size can be at most exponential in its arity. Semantics is ensured in three ways: (i) For a rule r, all possible values of its head literals are guessed, and (ii) satisfiability, and (iii) foundedness are ensured by explicitly encoding them. Interoperability with other techniques is ensured by hybrid grounding (Beiser et al., 2024).

Let Π be an HCF program and $\Pi_{\mathscr{H}} \cup \Pi_{\mathscr{G}}$ be a partition thereof. Then, let $\mathscr{H}$ be the *Hybrid Grounding* procedure that is executed on $(\Pi_{\mathscr{H}}, \Pi_{\mathscr{G}})$, where $\Pi_{\mathscr{H}}$ is grounded by BDG, and $\Pi_{\mathscr{G}}$ is grounded by bottom-up grounding. Let a be the maximum arity ($a = \max_{r\in\Pi} \max_{p(\mathbf{X})\in H_r\cup B_r} |\mathbf{X}|$) and let c be a constant defined as: where $c = a$ for r being a constraint, $c = 2 \cdot a$ for r occurring in a tight HCF program, and $c = 3 \cdot a$ for r occurring in an HCF program. Then, hybrid grounding for $\mathscr{H}(\Pi, \emptyset)$ has a grounding size[1] of $\approx |\mathrm{dom}(\Pi)|^c$. The coefficients c stem from the nature of the checks we have to perform. For constraints, it is sufficient to check satisfiability, while for normal programs we additionally need to check foundedness, which increases the grounding size to $c = 2 \cdot a$. For HCF programs, cyclic derivations must be prevented. This is handled with level-mappings, where the transitivity check increases the grounding size to $c = 3 \cdot a$.

[1] For brevity we sometimes shorten $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^x)$ with $\approx |\mathrm{dom}(\Pi)|^x$ for an arbitrary $x \in \mathbb{N}$.

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

Algorithm 1 *Heur*$(r,$ `MARKER`$)$ for Computing Data-Structural Heuristics

Data: Rule r, Set `MARKER` of marked rules

1 **if** *IsStratified(r)* **then**
2 $\quad$ `MARKER` $\leftarrow$ `MARKER` $\cup (r, \mathrm{SOTA})$;
3 **else if** $\varphi_r < |\mathrm{var}(r)| \wedge \hat{T}_{\bowtie}(\texttt{Lpopt}(r)) < \hat{T}_{\bowtie}(r)$ **then**
4 $\quad R_l \leftarrow \texttt{Lpopt}(r)$;
5 $\quad$ **for** r_l *in* R_l **do**
6 $\quad\quad$ $\mathit{Heur}_{\mathrm{struct}}(r_l, \texttt{MARKER})$;
7 **else if** $a < \varphi_r \wedge$ *IsConstraint(r)* $\wedge \hat{T}_{\mathscr{H}}(r) < \hat{T}_{\bowtie}(r)$ **then**
8 $\quad$ `MARKER` $\leftarrow$ `MARKER` $\cup (r, \mathrm{BDG})$;
9 **else if** $2 \cdot a < \varphi_r \wedge$ *IsTight(r)* $\wedge \hat{T}_{\mathscr{H}}(r) < \hat{T}_{\bowtie}(r)$ **then**
10 $\quad$ `MARKER` $\leftarrow$ `MARKER` $\cup (r, \mathrm{BDG})$;
11 **else if** $3 \cdot a < \varphi_r \wedge \hat{T}_{\mathscr{H}}(r) < \hat{T}_{\bowtie}(r)$ **then**
12 $\quad$ `MARKER` $\leftarrow$ `MARKER` $\cup (r, \mathrm{BDG})$;
13 **else**
14 $\quad$ `MARKER` $\leftarrow$ `MARKER` $\cup (r, \mathrm{SOTA})$;

3 Automated splitting heuristics

We designed an automated splitting heuristics that decides when it is beneficial to use BDG. This approach is given in Algorithm 1. Intuitively, the decision is based on fixed structural measures, like the number of variables and TW, as well as data-driven grounding-size estimation. Let Π be an HCF program, and $r \in \Pi$, then let $\hat{T}_{\mathscr{H}}(r)$ be the estimated grounding size of BDG, and let $\hat{T}_{\bowtie}(r)$ be the estimated SOTA grounding size. The algorithm takes as input a rule r and the set `MARKER`. Set `MARKER` stores whether a rule r is grounded by BDG or SOTA if $(r, \mathrm{BDG}) \in$ `MARKER` or $(r, \mathrm{SOTA}) \in$ `MARKER` respectively. This is then used to pass $\Pi_{\mathscr{H}} = \{r \mid r \in \Pi, (r, \mathrm{BDG}) \in \texttt{MARKER}\}$ and $\Pi_{\mathscr{G}} = \{r \mid r \in \Pi, (r, \mathrm{SOTA}) \in \texttt{MARKER}\}$ to $\mathscr{H}$.

First, in Lines (1)–(2), the algorithm performs a stratification check, where rules are SOTA-grounded whenever rules occur in stratified parts. Subsequently, the rule structure is checked, and a structural rewriting is performed in Lines (3)–(6), if beneficial. Finally, in Lines (7)–(14). BDG is evaluated and marked whenever it is structurally and data-estimation-wise beneficial.

Example 1.
We show the details and underlying intuitions of the heuristics along the lines of the example shown below. A simple instance graph is given by means of atoms over the edge predicate $e/2$. We guess subgraphs $f/2$, $g/2$, and $h/2$, where we forbid three or more connected segments in subgraph $f/2$, cliques of size ≥ 3 in subgraph $g/2$, and aim at inferring all vertices of a clique of size ≥ 3 in subgraph $h/2$. Let r_1, r_2, r_3 be the rule in Line (2), (3), (4), respectively.

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

```
1 {f(X,Y)} ← e(X,Y). {g(X,Y)} ← e(X,Y). {h(X,Y)} ← e(X,Y).
2 ← f(X1,X2), f(X2,X3), f(X3,X4).
3 ← g(X1,X2), g(X1,X3), g(X2,X3).
4 i(X1) ← h(X1,X2), h(X1,X3), h(X2,X3).
```

Previous results indicate that BDG should be used for *dense rules* on *dense instances* (Besin *et al*. 2022; Beiser *et al*. 2024). However, the terms *dense rule* and *dense instance* were loosely defined and the usage of BDG was guided by intuition. Our algorithm makes these terms precise and *transitions from intuition to computation*.

Variable-based Denseness. Next, we motivate how we consider variable-based denseness.

Example 2.
Observe how r_1 has four and r_2, and r_3 have three variables. Standard bottom-up grounding is exponential in these variables in the worst case. Without considering contributions of data and structural based rewritings for now, bottom-up's grounding size for rule r_1 is $\approx |\mathrm{dom}(\Pi)|^4$, while it is $\approx |\mathrm{dom}(\Pi)|^3$ for r_2, and $\approx |\mathrm{dom}(\Pi)|^3$ for r_3. In contrast, BDG's grounding size is only dependent on the maximum arity and the type of the rule. The maximum arity of all r_1, r_2, and r_3 is 2. As both r_1 and r_2 are constraints, their grounding size is in $\approx |\mathrm{dom}(\Pi)|^2$, while as r_3 is a tight HCF rule its grounding size is $\approx |\mathrm{dom}(\Pi)|^3$. The differences between BDG and SOTA are striking: A reduction from $\approx |\mathrm{dom}(\Pi)|^4$ to $\approx |\mathrm{dom}(\Pi)|^2$ and from $\approx |\mathrm{dom}(\Pi)|^3$ to $\approx |\mathrm{dom}(\Pi)|^2$ for r_1 and r_2, respectively (no difference for r_3).

We cover *variable-based denseness* based on the rule type and a comparison between the number $|\mathrm{var}(r)|$ of the variables and the maximum arity a. Henceforth, whenever the maximum arity adjusted for rule type is strictly smaller than the number of the variables, BDG is used. Let the maximum head arity be $a_h = \max_{p(\mathbf{X}) \in H_r} |\mathbf{X}|$ and the maximum body arity be $a_b = \max_{p(\mathbf{X}) \in B_r} |\mathbf{X}|$. For constraints, using BDG is beneficial whenever $a < |\mathrm{var}(r)|$, for tight HCF rules if $a_h + a_b \leq 2 \cdot a < |\mathrm{var}(r)|$, and for HCF rules if $3 \cdot a < |\mathrm{var}(r)|$.

When the projected grounding sizes match asymptotically, precedence is given to the bottom-up procedure: First, due to the effects of data (discussed below) and second, due to BDG's nature of pushing effort from grounding to solving. Since bottom-up grounding solves stratified programs with a grounding size in $\approx |\mathrm{dom}(\Pi)|^a$, grounding stratified parts with BDG is not beneficial.

Incorporating Rule Structure. To grasp the importance of structure, recall our running example.

Example 3.
We depict the variable graphs of r_1, r_2, and r_3 in Figure 1, which have TWs of 1, 2, and 2 respectively. A minimal TD of the variable graph of r_1 has a bag size of $\varphi_{r_1} = 2$. Take

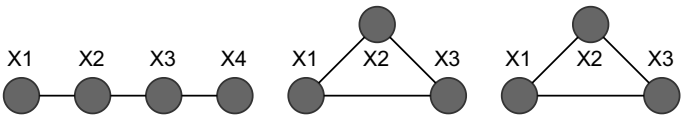

Fig 1. Variable graphs of r_1 (left), r_2 (center), and r_3 (right) for Example 1.

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

for example $\mathscr{T} = (T, \chi)$, *where* $T = (\{t_1, t_2, t_3\}, \{\{t_1, t_2\}, \{t_2, t_3\}\})$ *and* $\chi(t_1) = \{X1, X2\}$, $\chi(t_2) = \{X2, X3\}$, *and* $\chi(t_3) = \{X3, X4\}$. *Based on* $\mathscr{T}$, *we depict in the next listing a possible structural rewriting. Observe the grounding size of* $\approx |\mathrm{dom}(\Pi)|^2$.

```
1 tmp1(X3) ← f(X3,X4). tmp2(X2) ← f(X2,X3), tmp1(X3). ← f(X1,X2), tmp2(X2).
```

In contrast to this, a minimal TD of r_2 *or* r_3 *has a bag size of* $\varphi_r = 3$, *such as* $\mathscr{T} = (T, \chi)$, *where* $T = (\{t_1\}, \emptyset)$ *and* $\chi(t_1) = \{X1, X2, X3\}$. *Using structural rewritings for* r_2 *or* r_3 *has no effect. Therefore, the grounding sizes of BDG and* **`Lpopt`** *match for* r_1 *(both are* $\approx |\mathrm{dom}(\Pi)|^2$*), while BDG achieves a reduction from* $\approx |\mathrm{dom}(\Pi)|^3$ *to* $\approx |\mathrm{dom}(\Pi)|^2$ *for* r_2. *For* r_3, *both have a grounding size of* $\approx |\mathrm{dom}(\Pi)|^3$. *Whenever grounding sizes of BDG and* **`Lpopt`** *match, we give preference to* **`Lpopt`**, *as for BDG there are guesses*[2] *during solving.*

The observations above are incorporated in the heuristics by computing the TW of its variable graph and using `Lpopt` whenever the bag size ϕ_r of a minimal TD is strictly smaller than the number $|\mathrm{var}(r)|$ of variables ($\phi_r < |\mathrm{var}(r)|$). See Lines (3)–(6). Subsequently, a decision between BDG and bottom-up grounding is made based on the bag size of a minimal decomposition compared to the maximum arity of r ($a < \phi_r$), and the rule-type (constraint, tight, non-tight). Thereby, we transition from variable-based denseness to structure-aware denseness, which we incorporate into our algorithm in Lines (7), (9), and (11).

Incorporating Data-Awareness. The incorporation of data into our heuristics is vital. In its absence, BDG may be used when it is unwise to use it. Indeed, BDG is a domain-based grounding procedure, whose grounding size depends entirely on the domain of the program. On the other hand, bottom-up grounding is partially data-aware, as rule bodies perform joins between variables.

Example 4.
To visualize this, consider r_2 *and a graph that is a path with* 100 *vertices. While BDG's grounding size of* r_2 *is* $\approx |100|^2$, *bottom-up's grounding size is* 0.

To incorporate data into heuristics, observe that rule instantiations are similar to joins in a database system, where joins are done in the positive body (Leone *et al*. 2001). Interestingly, join size estimation procedures are common in the literature (Garcia-Molina *et al.* 2008). We estimate the SOTA grounding size according to the join-selectivity criterion (Leone *et al*. 2001)[3].

Let $r \in \Pi$. We compute the join estimation $\hat{T}_{\bowtie}(r)$ in an iterative way, by considering one literal $p_i \in B_r^+$ at a time. We start with the first positive body literal p_{l+1} and end with the last positive body literal p_m, as $B_r^+ = \{p_{l+1}, \ldots, p_m\}$. Further, we denote the computation of all positive predicates up to and including p_i as A_i. Let $\hat{T}(p_{i+1})$ be the estimated number of tuples of p_{i+1}, and $\hat{T}(A_i)$ be the estimated join size up to and including predicate p_i. Let $\mathrm{dom}(X, r)$ be the domain of variable X for the rule r, $\mathrm{dom}(X, p_i)$ be the domain of variable X for literal p_i, and let p_X be $p_X = \{p(\mathbf{X}) \mid p(\mathbf{X}) \in B_r^+, X \in \mathbf{X}\}$, where $X \in \mathrm{var}(r)$ is a variable. We compute a variable's domain size as

[2] Guesses are due to Equations (2), (4), and (9) of Figure 1 in hybrid grounding (Beiser et al. 2024).
[3] A variant of the join-selectivity criterion is used in `idlv` (Calimeri *et al*. 2018).

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

$\mathrm{dom}(X, r) = \bigcup_{p_i(\mathbf{X}) \in p_X} \mathrm{dom}(X, p_i)$. Equations (1)–(3) show our join size estimation for SOTA-grounding for a rule r, where $\hat{T}_{\bowtie}(r)$ refers to the estimation for a rule r.

$$\hat{T}(A_{l+1}) = \hat{T}(p_{l+1}) \tag{1}$$

$$\hat{T}(A_{i+1}) = \hat{T}(A_i \bowtie p_{i+1}) = \frac{\hat{T}(A_i) \cdot \hat{T}(p_{i+1})}{\Pi_{X \in \mathrm{var}(A_i) \cap \mathrm{var}(p_{i+1})} |\mathrm{dom}(X, r)|} \tag{2}$$

$$\hat{T}_{\bowtie}(r) = \hat{T}(A_m) = \hat{T}(A_{m-1} \bowtie p_m) \tag{3}$$

Precise grounding size estimations are possible for hybrid grounding. We show in Equations (4)–(10) the grounding size estimations for non-ground normal (HCF) programs. Each equation estimates the size of the respective hybrid grounding *rules*,[4] as introduced in Beiser et al. (2024). Consider for example Equation (7), which estimates the size of Rules (5)–(7) of the hybrid grounding reduction as introduced in Beiser et al. (2024). It intuitively captures for a rule $r \in \Pi$ whether a literal $p(\mathbf{X}) \in H_r \cup B_r$ for an arbitrary instantiation $p(\mathbf{D}) \in \mathrm{HB}(\Pi)$ contributes to r being satisfied. We estimate this as $\hat{T}^{S3}_{\mathscr{H}}(r)$ in Equation (7). We continue with a brief description of the other equations and their corresponding rules in the hybrid grounding reduction. Equation (4) is the estimation of the head-guess size, for the respective Rule (2). Equations (5)–(7) estimate the size of the satisfiability encoding, where Equations (5) and (6) estimate the impact of variable guessing, saturation, and the constant parts, which relate to the Rules (4) and (8) in hybrid grounding. We already described Equation (7) above. Equations (8)–(10) estimate the size of the foundedness part. Equation (8) estimates the size of the constraint that prevents unfounded answersets, which relates to Rule (12). Equation (9) estimates the size of the variable instantiations, which relates to Rule (9). Finally, Equation (10) is concerned with the estimation when a rule is suitable for justifying an atom, which relates to Rules (10)–(11).

$$\hat{T}^{G}_{\mathscr{H}}(r) = 2 \cdot \left(\Sigma_{h(\mathbf{X}) \in H_r} \Pi_{X \in \mathbf{X}} |\mathrm{dom}(X)|\right) \tag{4}$$

$$\hat{T}^{S1}_{\mathscr{H}}(r) = 2 \cdot \Sigma_{X \in \mathrm{var}(r)} |\mathrm{dom}(X)| \tag{5}$$

$$\hat{T}^{S2}_{\mathscr{H}}(r) = 2 \tag{6}$$

$$\hat{T}^{S3}_{\mathscr{H}}(r) = \Sigma_{p(\mathbf{X}) \in H_r \cup B_r} \Pi_{X \in \mathbf{X}} |\mathrm{dom}(X)| \tag{7}$$

$$\hat{T}^{F1}_{\mathscr{H}}(r) = \Sigma_{h(\mathbf{X}) \in H_r} \Pi_{X \in \mathbf{X}} |\mathrm{dom}(X)| \tag{8}$$

$$\hat{T}^{F2}_{\mathscr{H}}(r) = \Sigma_{h(\mathbf{X}) \in H_r} \left(\Sigma_{Y \in \mathrm{var}(r) \setminus \mathbf{X}} \left(|\mathrm{dom}(Y)| \cdot \Pi_{X \in \mathbf{X}} |\mathrm{dom}(X)|\right)\right) \tag{9}$$

$$\hat{T}^{F3}_{\mathscr{H}}(r) = \Sigma_{h(\mathbf{X}) \in H_r} \left(\Sigma_{p(\mathbf{Y}) \in H_r \cup B_r \setminus \{h(\mathbf{X})\}} \left(\Pi_{Y \in \mathbf{Y}} |\mathrm{dom}(Y)| \cdot \Pi_{X \in \mathbf{X}} |\mathrm{dom}(X)|\right)\right) \tag{10}$$

We are left with Equation (11), which computes $\hat{T}_{\mathscr{H}}(r)$, the hybrid grounding size estimation for a rule r. Equation (11) sums up Equations (4)–(10).

$$\hat{T}_{\mathscr{H}}(r) = \hat{T}^{G}_{\mathscr{H}}(r) + \hat{T}^{S1}_{\mathscr{H}}(r) + \hat{T}^{S2}_{\mathscr{H}}(r) + \hat{T}^{S3}_{\mathscr{H}}(r) + \hat{T}^{F1}_{\mathscr{H}}(r) + \hat{T}^{F2}_{\mathscr{H}}(r) + \hat{T}^{F3}_{\mathscr{H}}(r) \tag{11}$$

[4] To avoid confusion, we distinguish in this paragraph between equation, the grounding size estimation, and *rule*, the equation of the hybrid grounding reduction that is being estimated as introduced in Beiser *et al.* (2024).

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

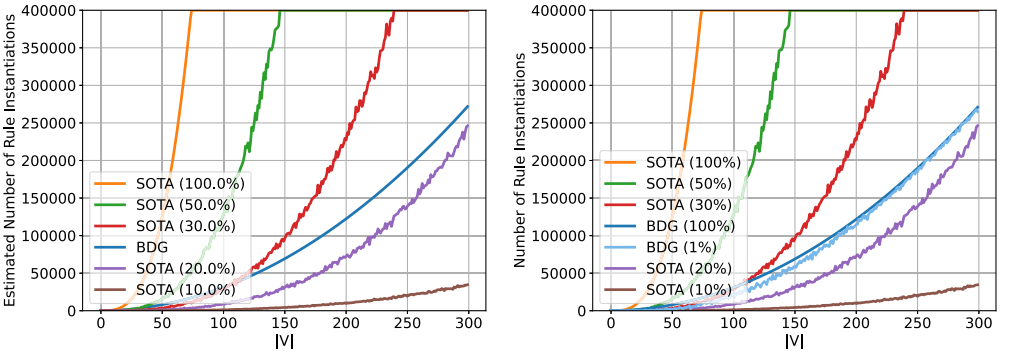

Fig 2. Plot comparing the estimated (left) and actual (right) number of ground rules of r_2 of Example 1. Comparison between SOTA and BDG. x-axis: number of vertices; y-axis: number of rules. Comparing different graph densities, shown as SOTA(x) and BDG(x) for density x.

Example 5.
In Figure 2 we show the estimated and actual number of instantiated rules for bottom-up grounding and BDG, for r_2. The behavior is analyzed on different graph densities (number of edges divided by edges of complete graph in percent) and graph sizes (1 to 300 vertices). The number of tuples $T(p_i)$ can be adequately estimated for our example, so $\hat{T}(p_i) \approx T(p_i)$. While for bottom-up grounding the estimated number of ground rules varies with density, it remains constant for BDG. BDG's number of instantiated rules between a complete (100 %) and a sparse (1 %) graph remains relatively similar. For bottom-up grounding, the number of instantiated rules varies.

Overall we obtain the following result on the grounding size by automated hybrid grounding.

Theorem 1.
Let Π be a non-ground HCF program and k be the maximum TW of any rule in Π. Then, the grounding size of Π, grounded with the markings `MARKER`*, $\Pi_{\mathscr{H}} = \{r \mid r \in \Pi, (r, \text{BDG}) \in \texttt{MARKER}\}$ and $\Pi_{\mathscr{G}} = \{r \mid r \in \Pi, (r, \text{SOTA}) \in \texttt{MARKER}\}$, produced by Algorithm 1 and grounded by $\mathscr{H}(\Pi_{\mathscr{H}}, \Pi_{\mathscr{G}})$, is in $\mathscr{O}\left((|\Pi| \cdot k) \cdot |\text{dom}(\Pi)|^{3 \cdot a}\right)$.*

Proof (idea).
Intuitively, structural parts of the algorithm bound the grounding size to $\mathscr{O}((|\Pi| \cdot k) \cdot |\text{dom}(\Pi)|^{3 \cdot a})$. We are left to prove that this still holds when incorporating data-awareness, which holds on dense instances. The proof is detailed in the appendix. □

4 Prototype implementation `newground3`

Our prototype `newground3`[5] is a full-fledged grounder that combines bottom-up with BDG. It incorporates BDG into the bottom-up procedure, where we decide according to the data-structural heuristics (Algorithm 1) whether to use BDG or not. Furthermore, the algorithm does not pre-impose on the user which SOTA grounder to use, and therefore,

[5] Prototype available under https://github.com/alexl4123/newground.

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

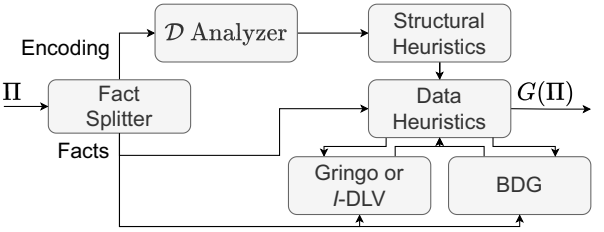

Fig 3. Schematics of the software architecture of the `newground3` prototype.

offers integration with `gringo` and `idlv`. In this section, we discuss implementation choices, highlight implementation challenges, and present the structure of the prototype.

We performed a full-scale redevelopment of the earlier versions of `newground3` (`newground` and `NaGG`), where on a high level, semi-naive grounding is interleaved with BDG. We further extended its input language to the ASP-Core-2 (Calimeri *et al*. 2020) input language standard[6] and improved the grounding performance of `newground`. For the semi-naive grounding parts we use either `gringo`, or `idlv`, whereas, for the BDG part we use a completely redesigned BDG-instantiator. To improve performance even further, we combine Python with Cython and C code.

Architectural Overview. The general architecture of the prototype consists of 4 parts, where we show a schematics in Figure 3. Given a program Π, the *fact splitter and analyzer* (Fact Splitter) written in Cython, separates facts from the encoding. It further computes the number of facts, and fact-domain. This enables an efficient computation of the *positive dependency graph and analysis thereof* ($\mathcal{D}$ Analyzer). Based on these results the *structural heuristics* decides which rules are eligible for grounding with BDG. If no rules are structurally eligible for grounding with BDG then the program is grounded by either `gringo` or `idlv`. Otherwise, the bottom-up procedure is *emulated* and for each strongly connected component in the positive dependency graph, where at least one rule is structurally eligible for grounding with BDG, the data heuristics decides whether to ground the rule with BDG or with a SOTA-approach.

In the development of the prototype we encountered two major challenges: (i) integration and communication with `gringo` and `idlv`, and (ii) suitable domain inference for grounding size estimations of Algorithm 1. To address these, we split the data-structural heuristics into two parts in our implementation: first, the structural heuristics decides, which parts are eligible for grounding with BDG and only then the estimation of the size of the instantiation of the eligible rules is performed. Further, we minimize the number of interactions with `gringo` and `idlv`, as each call to a SOTA-grounder is expensive and should better be avoided. Therefore, we do not infer the domain if the result of the structural heuristics states that BDG should not be used. The emulation is necessary, as neither `gringo` nor `idlv` provides callback functions which let us implement our heuristics directly. In the future a direct implementation of the heuristics in a SOTA grounder would render these calls unnecessary and would improve performance even further.

[6] Currently not all ASP-Core-2 constructs are supported with BDG rewritings. Checks ensure that only supported constructs are considered to be grounded by BDG, while non-groundable ones are grounded by SOTA-techniques.

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

5 Experiments

In the following, we demonstrate the practical usefulness of our automated hybrid grounding approach. We benchmark solving-heavy and grounding-heavy instances, aiming at SOTA-like results on solving-heavy benchmarks, and beating SOTA results on grounding-heavy benchmarks

Benchmark System. We compared `gringo` (Version 5.7.1), `idlv` (1.1.6), `ProASP` (Git branch *master*, short commit hash *2b42af8*), `ALPHA` (Version 0.7.0), and our hybrid grounding system `newground3`. We benchmarked `newground3` with both `gringo`, and `idlv`. Further, we investigated the impact of using our system in combination with `Lpopt` (Version 2.2). We chose `clingo` (Version 5.7.1) with `clasp` (3.3.10) for solving. However, in principle, one could also use `dlv` with `wasp`, or use heuristics to determine the solver of choice (Calimeri *et al*. 2020). For `newground3` we use Python version 3.12.1. Our system has *225 GB* of RAM, and an *AMD Opteron 6272* CPU, with 16 cores, powered by *Debian 10* OS with kernel 4.19.0-16-amd64.

Benchmark Setup. For all experiments and systems, we measure *total time*, which includes grounding and solving time for ground-and-solve systems, or execution time for `ALPHA` and `ProASP`. Further, we measure RAM usage for all systems and experiments. For the ground-and-solve systems we measured grounding performance (grounding time, grounding size, and RAM usage) in a separate run. Every experiment has a timeout of *1800s* and a RAM (and grounding-size) limit of *10 GB*. For integrated grounders and solvers (`ALPHA` and `ProASP`) this RAM limit applies to their execution. For ground-and-solve systems this applies to grounding and solving.

We consider instances as a *TIMEOUT* whenever they take longer than *1800s*, and a *MEMOUT* when their RAM usage exceeds *10 GB*. We set seeds for clingo (*11904657*), and for `Lpopt` (*11904657*). Further, for all generated graph instances for the grounding-heavy experiments we generated random seeds that we saved inside the random instance as a predicate.

5.1 Experiment scenarios and instances

We distinguish between *solving*- and *grounding*-heavy benchmarks. For the solving-heavy benchmarks we compare `idlv`, `gringo`, `newground3` with `gringo` (`NG-G`), `newground3` with `idlv` (`NG-I`), `ALPHA`, and `ProASP` (ground-all). For the grounding-heavy benchmarks we compare grounders `idlv`, `gringo`, `newground3` with `gringo`, `newground3` with `idlv`, `ALPHA`, `ProASP` (ground-all), and `ProASP` with compiling constraints (`ProASP-CS`).

Solving-Heavy Benchmarks. The solving-heavy benchmarks are taken from the 2014 ASP-Competition (Calimeri *et al*. 2016), as they provide a large instance set with readily available efficient encodings. The 2014 ASP-Competition has 25 competition scenarios, where each (with the exception of *Strategic-Companies*) has two encodings, resulting in 49 competition scenarios. Each scenario has a different number of instances. We benchmarked all instances over all scenarios. Further, we preprocessed the encodings s.t. no predicates occur, which have the same predicate name, but differing arity.

We show the encoding of problem *MaximalCliqueProblem* (2014 encoding)[7] as an example:

[7] The whole competition suite can be found at: https://www.mat.unical.it/aspcomp2014/FrontPage.

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

```
1 clique(X) ← node(X), not nonClique(X).
2 nonClique(X) ← node(X), not clique(X).
3 ← clique(X), clique(Y), X < Y, not edge(X,Y), not edge(Y,X).
4 :∼ nonClique(X). [1,X]
```

Intuitively the encoding guesses nodes that are part of the maximal clique (Lines 1,2). If there is a missing edge between a pair of nodes, then it is not a clique (Line 3). We minimize the number of non-clique nodes (Line 4).

Grounding-Heavy Benchmarks. We take grounding-heavy benchmarks from the BDG experiments (Besin *et al*. 2022) and from the hybrid grounding experiments (Beiser et al. 2024). These scenarios take a graph as an input, where we generate random graphs ranging from instance size 100 to 2000 with a step-size of 100 for the BDG scenarios (Besin *et al*. 2022) and random graphs ranging from instance size 20 to 400 with a step-size of 20 for the hybrid grounding scenarios (Beiser et al. 2024). For both, we use graph density levels ranging from 20 % to 100 %.

Further, we adapt the benchmarks from Besin *et al*. (2022) by adding two variations of the 3-*Clique* benchmark. The variations resemble different difficulties for BDG and SOTA grounders. The first listing (3-Clique-not-equal) shows the original formulation from Besin *et al*. (2022), and the second one (3-Clique) depicts the adaptation that makes it easier for SOTA grounders by changing "≠" to "<."

```
1 {f(X,Y)} ← edge(X,Y).
2 ← f(A,B), f(A,C), f(B,C), A ≠B, B ≠ C, A ≠ C.
```

```
1 {f(X,Y)} ← edge(X,Y).
2 ← f(A,B), f(A,C), f(B,C), A < B, B < C, A < C.
```

The adapted[8] scenarios from Besin *et al*. (2022) are called as follows: 3-Clique, 3-Clique-not-equal, directed-Path, directed-Col, 4-Clique, NPRC. The examples S3T4, S4T6, NPRC-AGG, and SM-AGG, are from Beiser *et al*. (2024).

5.2 Experimental hypotheses

H1 The Data-Structural-Heuristics (Algorithm 1) implemented in our prototype `newground3` approaches other SOTA ground-and-solve system's performance on solving-heavy benchmarks.

H2 Data-Structural-Heuristics of `newground3` yields an improvement in performance (solved instances) on grounding-heavy benchmarks, in contrast to other SOTA systems.

5.3 Experimental results and discussion

We show an overview of our results in Table 1 and Figure 4; a detailed solving profile of the grounding-heavy scenario *4-Clique* is given in Figure 5. For details, see supplementary material.

[8] `ProASP`'s syntax currently does not support choice rules, so we adapted the subgraph encoding for `ProASP` with a negative cycle encoding (*f(X,Y) :- edge(X,Y), not nf(X,Y). nf(X,Y) :- edge(X,Y), not f(X,Y).*). This is also used for `ALPHA`.

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

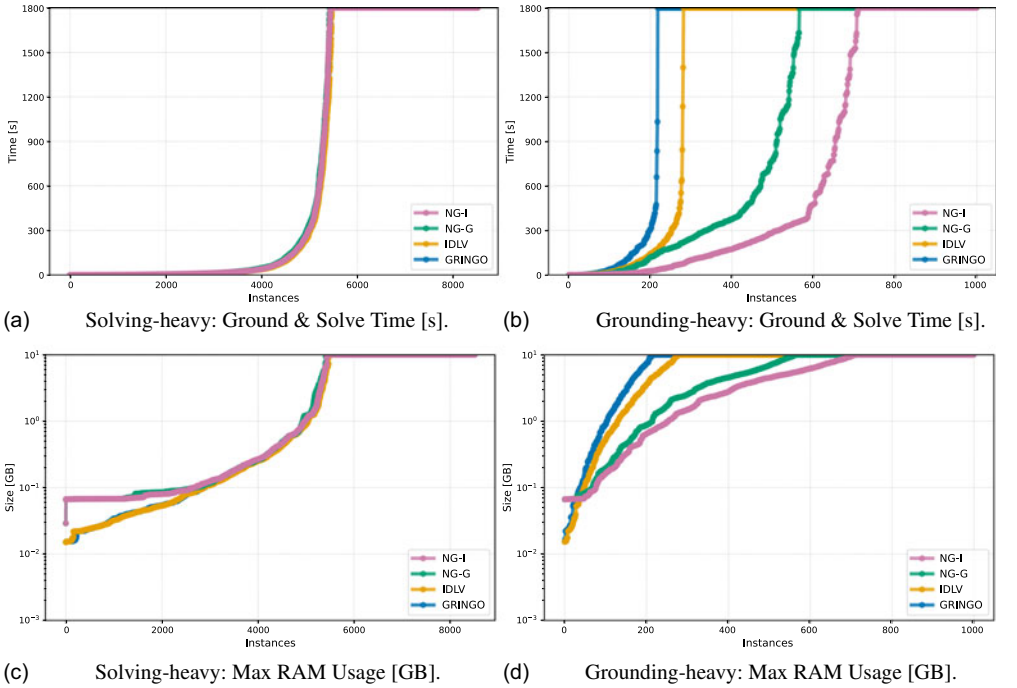

Fig 4. Solving-heavy (Figures 4a and 4c) and grounding-heavy (Figures 4b, and 4d) experiments. x-axis: instances; y-axis: time [s] or size [GB]. Measured `idlv`, `gringo`, `newground3` with `gringo` (`NG-G`), and `newground3` with `idlv` (`NG-I`). Timeout: 1800s; memout: 10 GB.

Discussion of H1. To confirm H1, we focus our attention on the results of the solving-heavy experiments. These are displayed in Figures 4a and 4c and in the lower half of the Table 1. The figures show that that `newground3`'s performance is approximately the same as the other ground-and-solve approaches. The detailed results of the table show that the overall number of solved instances for `gringo` is 5449, for `idlv` 5469, for `NG-G` 5418, and for `NG-I` 5434. The difference between `gringo` and `NG-G` are 31 instances, and for `idlv` and `NG-I` are 35 instances. On in total 8509 solving-heavy instances this resembles an approximate relative difference of 0.36 % for `gringo` *versus* `NG-G` and 0.41 % for `idlv` *versus* `NG-I`. The detailed results show that for `gringo` *versus* `NG-G` there are cases where `gringo` beats `NG-G` and cases where `NG-G` beats `gringo`. The same holds for `idlv` *versus* `NG-I`. As the differences of solved instances between `newground3` and the respective SOTA grounders are minor, we confirm H1.

Discussion of H2. We compare the results for the grounding-heavy scenarios of Figures 4b and 4d, and the upper half of Table 1. While `gringo` solves 218, and `idlv` 281, `newground3` solves 566 in the `NG-G` and 710 in the `NG-I` configuration, from a total of 1000 instances. This is a difference of 34.8 % and 42.9 %, respectively. Also observe the milder increase in RAM usage in Figure 4d and the ability to ground denser instances (Figure 5). As `newground3`'s ability to automatically determine when to use BDG leads to an approximate doubling in the number of solved grounding-heavy instances, we can confirm H2.

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

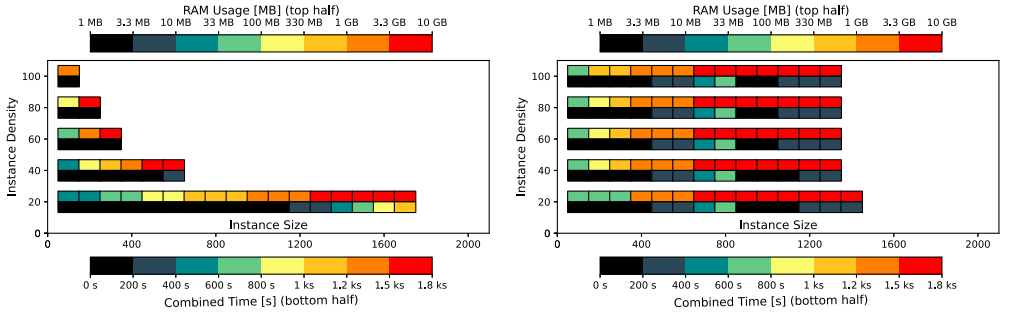

Fig 5. Solving profiles for grounding-heavy scenario *4-Clique* for `gringo` (left) and `newground3` with `gringo` (`NG-G`). One rectangle represents one grounded and solved instance. Timeout: 1800s; memout: 10 GB. Instance size on x-axis, instance density on y-axis.

Table 1. *Experimental results showing all scenarios, those executable by Alpha, and those executable by ProASP, with differing number of instances (#I). We depict solved instances (#S), memouts (M), and timeouts (T) for `gringo`, `idlv`, `NG-G`, `NG-I`, `ALPHA`, and `ProASP`*

Instance Summary	#I	Total #Solved								
		Grounding-Heavy Scenarios								
		`gringo`			`idlv`			`NG-G`		
		#S	M	T	#S	M	T	#S	M	T
All	1000	218	169	613	281	198	521	566	336	98
ProASP	500	149	97	254	158	102	240	280	210	10
		`NG-I`			`ALPHA`			`ProASP-CS`		
		#S	M	T	#S	M	T	#S	M	T
All	1000	**710**	247	43	–	–	–	–	–	–
ProASP	500	288	198	14	147	177	176	**389**	81	30
		Solving-Heavy Scenarios								
		`gringo`			`idlv`			`NG-G`		
		#S	M	T	#S	M	T	#S	M	T
All	8509	5449	650	2410	**5469**	697	2343	5418	524	2567
Alpha	1640	1255	30	355	**1280**	0	360	1251	24	365
ProASP	320	308	0	12	308	0	12	307	0	13
		`NG-I`			`ALPHA`			`ProASP`		
		#S	M	T	#S	M	T	#S	M	T
All	8509	5434	599	2476	–	–	–	–	–	–
Alpha	1640	1272	0	368	183	290	1167	–	–	–
ProASP	320	306	0	14	3	53	264	**311**	0	9

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

Summary of results

For both solving-heavy and grounding-heavy benchmarks `NG-G` and `NG-I` outperformed `ALPHA` significantly. `ProASP` has a comparable performance on solving-heavy benchmarks. On grounding-heavy benchmarks, `ProASP` shows promising results, however only when we use `ProASP` in the compile constraints mode. In the ground-all mode its behavior is similar to `gringo` or `idlv`. This confirms the results of previous studies about the performance of `ProASP` (Dodaro *et al*. 2024). Although the results of `ProASP` are very promising, it is only usable for a small fragment of the scenarios.

6 Conclusion

The advancement of alternative grounding procedures is an important step towards solving the grounding bottleneck. Previous results for the newly introduced BDG method (Besin *et al*. 2022) showed improvements on grounding-heavy tasks. Hybrid grounding (Beiser et al. 2024) enables manual partitioning of a program into a part grounded by standard grounders and a part grounded by BDG. However, due to the challenging predictability of BDG's solving performance, it remained unclear when the usage of BDG is useful.

In this paper, we state a data-structural heuristics, which decides when it is beneficial to use BDG. Our heuristics decision is based on the structure of a rule and the data of the instance. For each rule a minimum TD of the rule's variable graph is computed and compared to the maximum arity of the rule. Whenever the bag size of the minimum TD is smaller, the rule is grounded with bottom-up grounding. Otherwise the grounding size of the rule is estimated for bottom-up grounding by methods from databases, which is compared to an estimate of BDG's grounding size. Whichever is smaller is chosen for grounding. Our prototype `newground3` implements this heuristics by emulating a bottom-up procedure. The results of our experiments show that we approach bottom-up grounders number of solved instances for solving-heavy scenarios, while we approximately double the number of solved instances for grounding-heavy scenarios. We think that this is an important step towards integrating BDG into SOTA grounders. However, there is still future work to be explored for BDG. We argue that near-term research should include improvements of BDG for high-arity programs, as well as for syntactic extensions, highly cyclic rules, large HCF rules, and disjunctive programs.

Supplementary material

Supplementary material and prototype available under: https://github.com/alexl4123/newground.

Acknowledgments

This research was funded in part by the Austrian Science Fund (FWF), grants 10.55776/COE12 and J 4656. This research was supported by Frequentis.

References

Balduccini, M. and Lierler, Y. 2017. Constraint answer set solver EZCSP and why integration schemas matter. *Theory and Practice of Logic Programming* 17, 4, 462–515.

Banbara, M., Kaufmann, B., Ostrowski, M. and Schaub, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming* 17, 4, 408–461.

Beiser, A. G., Hecher, M., Unalan, K. and Woltran, S. 2024. Bypassing the ASP bottleneck: hybrid grounding by splitting and rewriting. In *IJCAI24*, International Joint Conferences on Artificial Intelligence Organization, 3250–3258.

Besin, V., Hecher, M. and Woltran, S. 2022. Body-decoupled grounding via solving: A novel approach on the ASP bottleneck. In *IJCAI22*, International Joint Conferences on Artificial Intelligence Organization, 2546–2552.

Bichler, M., Morak, M. and Woltran, S. 2016. popt: A Rule Optimization tool for answer set programming, *LOPSTR16*, Vol. 10184, LNCS, Springer, 114–130.

Calimeri, F., Dodaro, C., Fuscà, D., Perri, S. and Zangari, J. 2020. Efficiently coupling the I-DLV grounder with ASP solvers. *Theory and Practice of Logic Programming* 20, 2, 205–224.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F. and Schaub, T. 2020. ASP-Core-2 input language format. *Theory and Practice of Logic Programming* 20, 2, 294–309.

Calimeri, F., Fuscà, D., Perri, S. And Zangari, J. 2018. Optimizing answer set computation via heuristic-based decomposition. In PADL18, Vol. 10702, LNCS, IOS Press, 135–151.

Calimeri, F., Fuscà, D., Perri, S., Zangari, J., Maratea, M., Adorni, G., Cagnoni, S. and Gori, M. 2017. I-DLV: The new intelligent grounder of DLV. *Intelligenza Artificiale* 11, 1, 5–20.

Calimeri, F., Gebser, M., Maratea, M. and Ricca, F. 2016. Design and results of the fifth answer set programming competition. *Artificial Intelligence* 231, 151–181.

Cuteri, B., Dodaro, C., Ricca, F. and Schüller, P. 2019. Partial compilation of ASP programs. *Theory and Practice of Logic Programming* 19, 5-6, 857–873.

Dantsin, E., Eiter, T., Gottlob, G. and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys* 33, 3, 374–425.

Dodaro, C., Mazzotta, G. and Ricca, F. 2023. Compilation of tight ASP programs. In *ECAI23*, Vol. 372, FAIA, IOS Press, 557–564.

Dodaro, C., Mazzotta, G. and Ricca, F. 2024. Blending grounding and compilation for efficient ASP solving. In *KR24*, International Joint Conferences on Artificial Intelligence, 317–328.

Falkner, A., Friedrich, G., Schekotihin, K., Taupe, R. and Teppan, E. C. 2018. Industrial applications of answer set programming. *KI - Künstliche Intelligenz* 32, 2-3, 165–176.

Garcia-molina, H., Ullman, J. and Widom, J. 2008. *Database Systems: The Complete Book*, 2nd ed. Pearson/Pearson Prentice Hall, Upper Saddle River, NJ.

Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V. and Schaub, T. 2015. Abstract gringo. *Theory and Practice of Logic Programming* 15, 4-5, 449–463.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T. and Wanko, P. 2016. Theory solving made easy with clingo 5. In ICLP16-TC, Vol. 52, OASIcs, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (Dagstuhl Publishing), 1–15.

Gebser, M., Kaminski, R. and Schaub, T. 2016. Grounding Recursive Aggregates: Preliminary Report. *CoRR*. http://arxiv.org/abs/1603.03884.

Gebser, M., Leone, N., Maratea, M., Perri, S., Ricca, F. and Schaub, T. 2018. Evaluation techniques and systems for answer set programming: a survey. In *IJCAI18*, International Joint Conferences on Artificial Intelligence, 5450–5456.

https://doi.org/10.1017/S1471068425100173 Published online by Cambridge University Press

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9, 3-4, 365–385.

Janhunen, T. 2006. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics* 16, 1-2, 35–86.

Kaminski, R. and Schaub, T. 2023. On the foundations of grounding in answer set programming. *Theory and Practice of Logic Programming* 23, 6, 1138–1197.

Knaster, B. 1928. Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématique* 6, 133–134.

Leone, N., Perri, S. and Scarcello, F. 2001. Improving ASP instantiators by join-ordering methods. In LPNMR01, Vol. 2173, LNCS, Cambridge University Press, 280–294.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* 7, 3, 499–562.

Leutgeb, L. and Weinzierl, A. 2018. Techniques for efficient lazy-grounding ASP solving. In *DECLARE18*, vol. 10997, *LNCS*, Springer, 132–148.

Lierler, Y. and Robbins, J. 2021. DualGrounder: Lazy instantiation via clingo multi-shot framework. In *JELIA21*, Vol. 12678, *LNCS*, Société Polonaise de Mathématique, 435–441.

Lin, F. and Zhao, J. 2003. On tight logic programs and yet another translation from normal logic programs to propositional logic. In *IJCAI03*, Springer, 853–858.

Liu, G., Janhunen, T. and Niemela, I. 2012. Answer set programming via mixed integer programming. In *KR12*, Association for Computing Machinery, 32–42.

Mastria, E., Zangari, J., Perri, S. and Calimeri, F. 2020. A machine learning guided rewriting approach for ASP logic programs. In ICLP20 - TC. EPTCS, Vol. 325, Springer, 261–267.

Mazzotta, G., Ricca, F. and Dodaro, C. 2022. Compilation of aggregates in ASP systems. In *AAAI22*, Vol. 36, Springer, 5834–5841. Issue: 5.

Morak, M. and Woltran, S. 2012. Preprocessing of complex non-ground rules in answer set programming. In *ICLP12*, Vol. 17, LIPIcs, Morgan Kaufmann Publishers Inc., 247–258.

Tarski, A. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics* 5, 2, 285–309.

Tsamoura, E., Gutierrez-basulto, V. and Kimmig, A. 2020. Beyond the grounding bottleneck: Datalog techniques for inference in probabilistic logic programs. In *AAAI20*, Vol. 34, Open Publishing Association (OPA), 10284–10291.

Weinzierl, A. 2017. Blending lazy-grounding and CDNL search for answer-set solving. In *LPNMR17*, Vol. 10377, LNCS, AAAI Press (Association for the Advancement of Artificial Intelligence), 191–204.

Weinzierl, A., Taupe, R. and Friedrich, G. 2020. Advancing lazy-grounding ASP solving techniques – restarts, phase saving, heuristics, and more. *Theory and Practice of Logic Programming* 20, 5, 609–624.