TU WIEN Informatics

Novel Techniques for Circumventing the ASP Bottleneck

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Alexander G. Beiser, BSc
Matrikelnummer 11904657

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Mitwirkung: Dipl.-Ing. Dr.techn. Markus Hecher

Wien, 23. Jänner 2025

Alexander G. Beiser	Stefan Woltran

TU WIEN Informatics

Novel Techniques for Circumventing the ASP Bottleneck

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Alexander G. Beiser, BSc
Registration Number 11904657

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Assistance: Dipl.-Ing. Dr.techn. Markus Hecher

Vienna, January 23, 2025

Alexander G. Beiser	Stefan Woltran

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

Erklärung zur Verfassung der Arbeit

Alexander G. Beiser, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 23. Jänner 2025

Alexander G. Beiser

Acknowledgements

"As we express our gratitude, we must never forget that the highest appreciation is not to utter words, but to live by them." - John F. Kennedy

I am deeply grateful to my supervisors, Stefan Woltran and Markus Hecher, for their guidance, fruitful discussions, and availability at *any* time. They introduced me to the topic of the grounding bottleneck and body-decoupled grounding shortly after their BDG paper was published in 2022. Their continued guidance, support, and advice, from my Bachelor's thesis in 2023 to our publication at IJCAI24, laid the foundation for this work. Further, I especially want to thank Stefan for supporting and enabling my attendance at the IJCAI24 conference on Jeju, South Korea (2024), and the TAASP workshops (2023 in Potsdam, 2024 in Klagenfurt).

I am grateful to my defense committee—Peter Knees, Martin Nöllenburg, and Stefan Woltran—for their time and availability during a busy holiday period. I also want to thank Juliane Auerböck for handling administrative matters and Markus Bartel and Matthias Nitzschke for keeping the benchmark servers up and running.

I dearly want to thank my family, and especially my parents Brigitte and Andreas, for their personal and emotional support, but most for their time and being there for me - I would never have gotten as far without them. They have been following me along my journey wherever my path led me. They were always proud and excited of my achievements and cheered me up whenever I needed it. Last but not least, I want to thank my friends and colleagues for their support, where I especially want to thank those (see names below) who reviewed my Master's Thesis draft and provided me with valuable feedback: Dominik Apel, Ivana Bocevska, Jonas Bodingbauer, Giovanni Buraglio, Thomas Hader, Alina Schärmer, Davide Soldà, and Kaan Unalan.

I conclude by recognizing that no acknowledgment section can fully capture the countless individuals who contribute to any scholarly work, as we all are *standing on the shoulders of giants*.

Kurzfassung

Symbolische künstliche Intelligenz ist exakt und zuverlässig. Allerdings verhindern sogenannte *combinatorial Explosions* das Lösen vieler Industrieprobleme. In dem Logikprogrammierparadigma Answer Set Programming (ASP), kommt dieses Phänomen der combinatorial Explosions häufig in der *Grounding Phase* vor. *Grounding*, also das Ersetzen von Variablen durch ihre Domäne, führt zu einem exponentiell größeren Programm. Dieses Problem wird auch *Grounding Bottleneck* genannt.

Die hier vorliegende Diplomarbeit zeichnet sich durch zwei primäre Beiträge zur Lösung des Grounding Bottlenecks aus: (i) Die effektive Kombination zwischen Body-decoupled Grounding (BDG) und (traditionellem) semi-naivem Grounding, und (ii) durch die Weiterentwicklung des BDG-Ansatzes für normale und zyklische Regeln.

Im Detail sind die Beiträge wie folgt: (1) Wir führen Hybrid Grounding ein, welches die freie (manuelle) Aufteilung eines Programmes in einen durch traditionelle Techniken und einen durch BDG gegroundeten Teil ermöglicht. (2) Weiters präsentieren wir *automated Hybrid Grounding*, welches die manuelle Aufteilung unter Verwendung von Heuristiken automatisiert. (3) Auch stellen wir eine verbesserte BDG-Reduktion für normale ASP-Programme vor, welche die Grounding-Zeit von $\mathcal{O}(|\Pi| \cdot |\text{dom}(\Pi)|^{2 \cdot a})$ auf $\mathcal{O}(|\Pi| \cdot |\text{dom}(\Pi)|^{a+1})$ reduziert. (4) Schließlich demonstrieren wir durch Lazy-BDG das effektive Grounden von zyklischen Programmen. Hierbei wird der Aufwand, der in der BDG-Grounding Phase anfällt, auf die Lösungsphase über Propagatoren verschoben.

Unsere Ergebnisse für (1) und (2) zeigen, dass BDG effizient implementiert und auf eine Weise verwendet werden kann, die orthogonal zu traditionellem Grounding ist. Dies führt zu einer Groundingmethode, die bei schwer lösbaren Problemen weder Effizienzeinbußen noch -steigerungen bewirkt, bei schwer zu grundierenden Problemen jedoch Zugewinne verbuchen kann. Weiterhin zeigen unsere Experimente zu (3) und (4), dass diese Ansätze sehr vielversprechend sind, da wir sowohl traditionelle Grounder, als auch die bisherige BDG-Methode übertreffen.

Abstract

Symbolic Artificial Intelligence is known for its reliability and exactness. Unfortunately, many symbolic approaches suffer from combinatorial explosions that render industry problems unsolvable. This is especially prevalent in the *grounding phase* of the logic-programming paradigm Answer Set Programming (ASP). Grounding, replacing variables with their domain, yields an exponentially larger program, the so-called *grounding bottleneck*. *Body-decoupled grounding* (BDG), a state-of-the-art method for easing the grounding bottleneck, achieves good results on grounding-heavy scenarios. However, it lacks interoperability with other state-of-the-art methods like semi-naive grounding and performs poorly on normal and non-tight ASP programs.

We tackle the challenges of BDG by (i) enabling the interoperability of BDG with semi-naive grounding, by introducing *hybrid grounding* and *automated hybrid grounding*, and (ii) advancing the BDG approach for normal and non-tight rules with *FastFound* and *Lazy-BDG*.

In more detail, we introduce *hybrid grounding*, which enables the free (manual) partitioning of a program into a part grounded by traditional techniques and a part grounded by BDG. Next, with *automated hybrid grounding*, we developed heuristics for automatically deciding when a usage of BDG is appropriate. We implemented this approach in our prototype `newground3`, which we compared experimentally to `gringo` and `idlv`. While on solving-heavy benchmarks, the difference in solved instances is less than 1%, `newground3` manages to increase the number of solved instances by about 35% on grounding-heavy benchmarks. This leads us to conclude that BDG can be used orthogonally to traditional grounders.

Furthermore, we improve the BDG reduction with *FastFound*, a method that enables a reduction in grounding size from $\mathcal{O}(|\Pi| \cdot |\text{dom}(\Pi)|^{2 \cdot a})$ to $\mathcal{O}(|\Pi| \cdot |\text{dom}(\Pi)|^{a+1})$. *Lazy-BDG* enables the effective grounding of non-tight programs by shifting effort spent in the grounding phase to the solving phase using propagators. Further, we demonstrate the usefulness of *FastFound* and *Lazy-BDG* by implementing them in our prototype `newground3` and beating both the original BDG formulation and the state-of-the-art grounders `gringo` and `idlv` on synthetic benchmarks.

Contents

CHAPTER 1

Introduction

Answer Set Programming (ASP) [71] is a logic-programming paradigm widely used in symbolic Artificial Intelligence (symbolic AI). ASP and its related symbolic AI approaches are employed in problem solving scenarios where certainty is a crucial factor, such as in train schedule design [1], or optimizing aircraft schedules [109]. However, one crucial standing deficiency of ASP is its *grounding bottleneck* [64]. In practice, the grounding bottleneck prohibits the usage of ASP for large industrial or scientific problems [49]. This thesis advances the state-of-the-art by proposing alternative solutions for solving the grounding bottleneck based on the novel *body-decoupled grounding* (BDG) [12] technique.

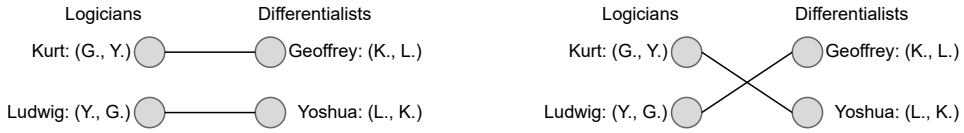
We begin by emphasizing the importance of using ASP for the prominent stable-matching (SM) [52] problem, followed by an introduction to the grounding bottleneck and the motivation for using BDG to alleviate this issue (Section 1.1). Next, we briefly outline the major challenges of BDG and explain how this thesis contributes to solving the grounding bottleneck (Section 1.2). We then provide an overview of the thesis structure in Section 1.3 and conclude the chapter by discussing related work (Section 1.4).

1.1 From the Stable Matching Problem to BDG

The *stable matching* (SM)[1] problem [52] is about matching individuals of two groups. Its introduction and subsequent discussion were the essential contributions for the award of the *2012 Nobel Memorial Prize in Economic Science* [110]. SM was originally posed as a problem of finding stable marriages between men and women. However, we pose[2] it in the context of the current ongoing debate in Artificial Intelligence between symbolists (henceforth called *logicians*)

[1] Historically called the stable marriage problem. SM and its derivatives are in widespread usage, such as for assigning graduated medicine students to universities [79]. An overview is given in [50].

[2] We describe the detailed setting in the Appendix (Page 156).



(a) Stable matching (b) Unstable matching

Figure 1.1: Matching logicians and differentialists (e.g., Kurt and Geoffrey), based on their preference lists. An edge between two vertices indicates a current match, Figure 1.1a depicts a stable match, whereas Figure 1.1b shows an unstable one.

and subsymbolists (henceforth called *differentialists*), who need to find each other: The goal is to match logicians and differentialists exactly one-to-one.

Each individual has a preference list that must be taken into account. Let us make matters concrete by saying *Kurt* and *Ludwig* are logicians, while *Geoffrey* and *Yoshua* are differentialists. Further, *Kurt* prefers to work with *Geoffrey* rather than with *Yoshua*, which we write as the following preference list: *Kurt:*$(G., Y.)$. Everyone has such a preference list: *Ludwig*: $(Y., G.)$; *Geoffrey*: $(K., L.)$; *Yoshua*: $(L., K.)$. Given the preference list, matching Kurt with Geoffrey and Ludwig with Yoshua results in a stable matching. However, the matching is unstable if Kurt matches with Yoshua and Ludwig matches with Geoffrey. We illustrate this example in Figure 1.1.

To make matters precise, we define the *strong stability condition* via its negation: A matching is unstable whenever (i) an individual X of a group is matched to a partner Z of the other group, although X prefers Y to Z, and additionally, (ii) Y prefers X to their current match. A matching is *stable* whenever it is not unstable.

The goal is to find a stable matching, where we assume we have n logicians and n differentialists. Specialized algorithms can solve SM in $\mathcal{O}(n^2)$ [50]. Although a quadratic increase is reasonable for many practical problems, specialized algorithms cannot easily be tweaked to changes. In the following, we consider two small changes in our SM problem definition to showcase this: (1) Assume that matchings must not exactly be one-to-one, so we allow logicians and differentialists to have multiple collaborators. How do we have to change the algorithm for that? Is the algorithm still in $\mathcal{O}(n^2)$? Alternatively, consider the following slight change in the problem definition: (2) We want to allow for a few violations of the strong stability condition. How would we have to adapt the algorithm this time?

Answering these questions requires special effort and likely adaptation of the algorithm. These investigations and adaptations will take up precious time for scientists or developers. To save time, ideally, we (only) want to specify the problem description without detailing *how* the problem is solved. This is what symbolic AI wants to achieve.

Answer Set Programming: Symbolic AI approaches like *Answer Set Programming (ASP)* [71] operate on a declarative paradigm: One encodes the problem statement directly in ASP and ASP systems compute an answer.

Take our SM[3] example from before, with *Kurt*, *Ludwig*, *Geoffrey*, and *Yoshua*. As the preferences of our logicians and differentialists are known upfront, we call them *facts* and encode them by two predicates: *logAssScore* (logician assigns score to differentialist) and *diffAssScore* (differentialist assigns score to logician). As the predicates have to relate three things (logicians, differentialists, scores), we say their *arity* is 3. Encoding the preference lists produces the following listing:

```
1 logAssScore(kurt,geoffrey,2). logAssScore(kurt,yoshua,1).
2 logAssScore(ludwig,geoffrey,1). logAssScore(ludwig,yoshua,2).
3 diffAssScore(geoffrey,kurt,2). diffAssScore(geoffrey,ludwig,1).
4 diffAssScore(yoshua,kurt,1). diffAssScore(yoshua,ludwig,2).
```

Besides the facts, we must also include our SM problem definition in an *encoding*. An encoding consists of rules and typically follows the so-called *guess-and-check* paradigm. According to this paradigm, we first guess possible matches and later restrict those matches to actual stable matches by constraints.

We encode the guess in two rules. The intuition behind the guess is that we either want to derive that they are surely matched or certainly not matched, which we express in the predicates *match* and *nonMatch* respectively. Both predicates have arity 2, as they relate logicians to differentialists. As we want this rule to hold for arbitrary logicians and differentialists, we use variables (*L*,*D*) instead of concrete names. The encoding shown in the listing below displays a common pattern seen in ASP: Guessing answers with rules that negatively (*not*) depend on each other.

```
1 match(L,D) :- logAssScore(L,_,_), diffAssScore(D,_,_), not nonMatch(L,D).
2 nonMatch(L,D) :- logAssScore(L,_,_), diffAssScore(D,_,_), not match(L,D).
```

After guessing all possible matches we restrict them to stable matches by constraints. For this, we first must discuss the necessary ASP syntax: An ASP rule is comprised of a head (H_r) and a body (B_r): $H_r \leftarrow B_r$. A *constraint* is a rule where the head is empty ($H_r = \emptyset$), a rule is *disjunctive* whenever $|H_r| > 1$, and a rule is *normal* whenever $|H_r| \leq 1$.

Our first check ensures an exact one-to-one match, so each logician matches precisely to one differentialist, and one differentialist matches to precisely one logician. We show this in two encodings, where the first one below shows how we restrict each individual to have at most one match:

```
1 :- match(L1,D), match(L,D), L != L1. :- match(L,D), match(L,D1), D != D1.
```

Our next encoding shows how we ensure that each logician and each differentialist has at least one match. For this, we introduce the *assigned* predicate which expresses that a logician, or a differentialist has an assigned match.

```
1 assigned(L) :- match(L,_). assigned(D) :- match(_,D).
2 :- logAssScore(L,_,_), not assigned(L). :- diffAssScore(D,_,_), not assigned(D).
```

[3] Problem encoding adapted from the 2014 ASP-Competition [25].

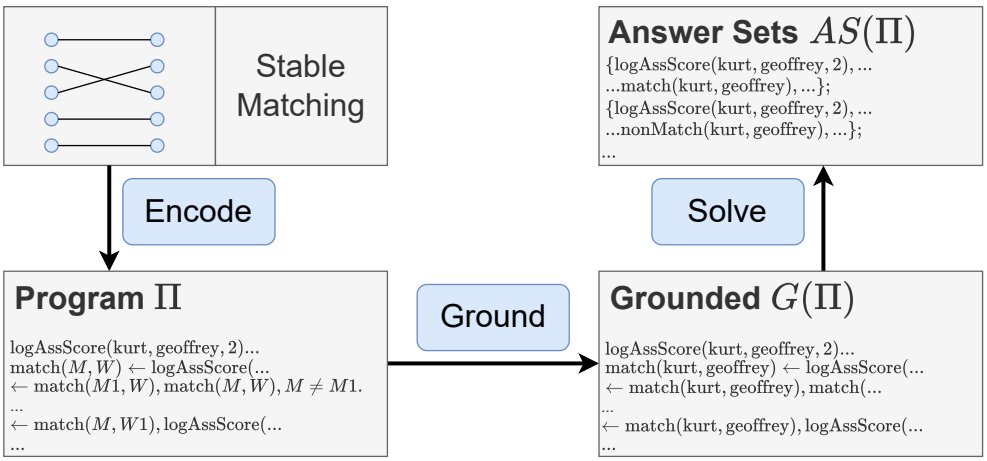


Figure 1.2: The traditional state-of-the-art ground-and-solve approach for ASP systems. A problem (Stable Matching) is *encoded* into a non-ground program Π. *Grounding* instantiates its variables with the respective domain values. *Solving* generates answer set(s), the solution(s) of the problem.

Combining both encodings above, so that each individual has at most one match and that each individual has at least one match, we infer that each individual has exactly one match.

The only check left is to ensure the strong stability condition. As we require each answer set to resemble a stable matching, we encode it via a constraint[4]. This expresses that whenever there is a logician L, that is currently in a match $D1$, however there is another differentialist D s.t. the score of D is higher than the one of $D1$ ($Sld > Sld1$), and D prefers L to its current match $L1$ ($Sdl \geq Sdl1$), then the matching is unstable.

```
:- match(L,D1), logAssScore(L,D,Sld), D1 != D, logAssScore(L,D1,Sld1),   Sld >  Sld1,
   match(L1,D), diffAssScore(D,L,Sdl),        diffAssScore(D,L1,Sdl1), Sdl >= Sdl1.
```

The full ASP encoding is shown in the listing below:

```
% Facts: logAssScore/3. diffAssScore/3.
% Guess matching
match(L,D) :- logAssScore(L,_,_), diffAssScore(D,_,_), not nonMatch(L,D).
nonMatch(L,D) :- logAssScore(L,_,_), diffAssScore(D,_,_), not match(L,D).
% Exactly a one-to-one match:
:- match(L1,D), match(L,D), L != L1. :- match(L,D), match(L,D1), D != D1.
% All must be assigned:
assigned(L) :- match(L,_). assigned(D) :- match(_,D).
:- logAssScore(L,_,_), not assigned(L). :- diffAssScore(D,_,_), not assigned(D).
% Strong stability condition
:- match(L,D1), logAssScore(L,D,Sld), D1 != D, logAssScore(L,D1,Sld1),   Sld >  Sld1,
   match(L1,D), diffAssScore(D,L,Sdl),        diffAssScore(D,L1,Sdl1), Sdl >= Sdl1.
```

Adapting the encoding above to the SM variants described in (1) and (2) is now easy. For (1), we only need to remove Line (6) from the encoding. To incorporate (2), we introduce the

[4]We are aware that more efficient encodings exist, however we opted for a direct encoding. Both encodings, the shown and the more efficient one can be found in the 2014 ASP-Competition [25] benchmark suite.

sscf predicate, which expresses a matching error, and the *relaxation* predicate, which holds the number of matching errors we may perform. Further, we have to change the encoding by transforming the strong stability condition from a constraint to a rule, which enables the counting of the number of matching errors. Counting, or other arithmetic operations over sets, can be encoded by aggregates. The listing below shows a selection of the adapted encoding. We use the *count-aggregate* (*#count*) in Line (5) to count (L, D) tuples.

```
1 relaxation(6).
2 % Strong stability condition
3 sscf(L,D) :- match(L,D1), logAssScore(L,D,Smw), D1 != D, logAssScore(L,D1,Smw1),   Smw >  Smw1,
4    match(L1,D), diffAssScore(D,L,Swm),        diffAssScore(D,L1,Swm1), Swm >= Swm1.
5 :- X <= #count{L,D:sscf(L,D)}, relaxation(X).
```

Given such an encoding, there is only one thing left: We have to compute an answer to our encoding. In ASP we can use tools such as `clingo` [58] or DLV [93] to derive answer sets. These tools operate on the ground-and-solve paradigm [64], where first *grounding* and then *solving* are performed. Grounding refers to instantiating variables with their respective domain. In our stable matching example, the domain consists of the logicians *kurt* and *ludwig*, the differentialists *geoffrey* and *yoshua*, and the scores *1* and *2*. The solving phase then computes the answer sets by a *conflict-driven-nogood-learning* (CDNL) algorithm. In Figure 1.2, we show a schematics of the ground-and-solve approach.

Grounding Bottleneck: The grounding bottleneck [64], [126] arises in the grounding phase in the context of the ground-and-solve paradigm in state-of-the-art systems. Essentially, it is an incarnation of a combinatorial explosion problem that prohibits the solving of large practical instances. In our SM encoding from above, the bottleneck arises in the one-to-one constraints:

```
1 :- match(L1,D), match(L,D), L != L1. :- match(L,D), match(L,D1), D != D1.
```

This line consists of two rules, where each rule has three variables. As each variable is replaced with all domain values, the grounding size is cubic[5] in the domain size. More formally, given our Π and its domain $\mathrm{dom}(\Pi)$, then the grounding size is in $\mathcal{O}\left(|\Pi| \cdot |\,\mathrm{dom}(\Pi)|^3\right)$.

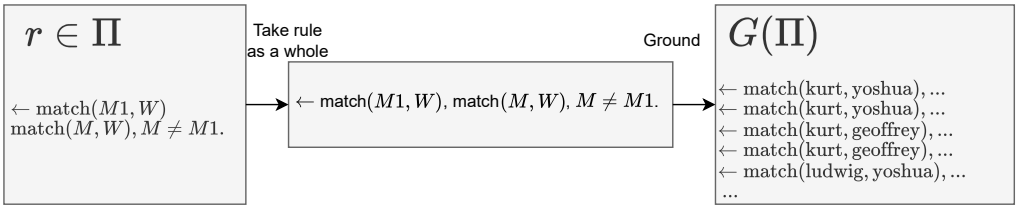
In practice, modern grounders do not naively instantiate variables by their domain values but instead, use the bottom-up/semi-naive grounding techniques [61]. State-of-the-art systems include `clingo`'s `gringo` [56], or DLV's `idlv` [23]. These specialized grounding techniques enable simplifications, like on-the-fly evaluation of stratified programs or fact propagation. Still, in the worst case[6] their grounding size is exponential in the number of variables.

We follow for the general case: Let Π be a program, $\mathrm{dom}(\Pi)$ be the domain (non-ground terms) of Π and $\mathcal{V}$ the maximum number of variables of any rule in Π. Then, the grounding size of a program Π is in the worst-case displayed in Equation 1.1:

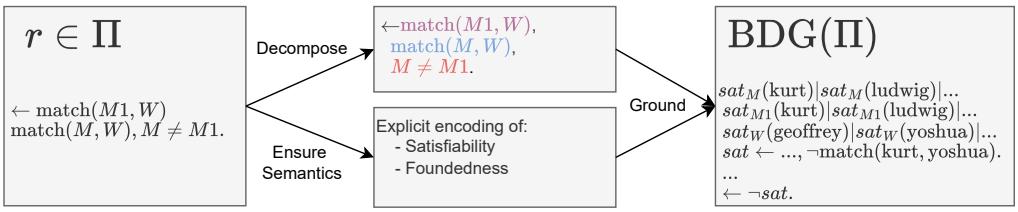
$$\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^{\mathcal{V}}) \tag{1.1}$$

[5]Note that the strong stability condition (Lines (11)-(12)) of the original encoding has more (8) variables. However, instance definitions, state-of-the-art grounders, and alternative (non-intuitive) encodings effectively reduce its grounding size from being to the power of 8, to (effectively) only being quadratic in the domain size.

[6]See Chapter 5 for an in-depth discussion.



(a) Traditional grounding treats the rules as a whole.



(b) BDG decomposes the rule and encodes semantics explicitly.

Figure 1.3: Schematic comparison how traditional grounding (Figure 1.3a) works in comparison to BDG (Figure 1.3b). Given a rule $r \in \Pi$, traditional grounding takes the rule as it is and instantiates the variables var $= \{M1, W, M\}$ ($\mathcal{V} = |\text{var}(r)| = 3$) with all possible domain values. Traditional grounding has an exponential grounding size in the number of variables: $\approx |\text{dom}(\Pi)|^{\mathcal{V}}$. BDG decomposes the rule into its literals (indicated by different colors and lines). The predicate arities are: $a(\text{match}) = 2$ and $a(M \neq M1) = 2$. Therefore, the maximum arity is $a = 2$. BDG has an exponential grounding size in the maximum arity: $\approx |\text{dom}(\Pi)|^a$. Thus, while traditional grounding has a cubic ($\approx |\text{dom}(\Pi)|^3$) grounding size, BDG has only a quadratic one ($\approx |\text{dom}(\Pi)|^2$).

The *grounding bottleneck* is a direct consequence of Equation 1.1. This problem prevents the solving of larger industrial applications [49]. Fortunately, many approaches try to solve or at least ease the grounding bottleneck. Examples of these are lazy-grounding [130], compilation based approaches [38], or body-decoupled grounding [12] (BDG). While lazy-grounding and compilation based techniques describe a system-level approach that drops grounding altogether, BDG is essentially a rewriting technique. We argue that the further development of BDG is promising for alleviating the grounding bottleneck, as it can be effectively combined with state-of-the-art solvers and is compatible with the use of lazy-grounding and compilation based approaches[7] .

Body-decoupled Grounding (BDG): Historically, BDG was inspired by complexity theory as it is a complexity theoretic reduction from non-ground normal ASP to ground disjunctive ASP. Its practical significance stems from the fact that it *decouples* non-ground rules in the sense that each predicate is grounded on its own. Semantics is ensured by explicitly encoding satisfiability and foundedness. This has the benefit that, in comparison to semi-naive grounding, its grounding size is *only* exponential in the maximum arity of the program.

[7]The main requirement is that lazy-grounding and compilation based techniques support cyclic disjunctive rules.

Take the one-to-one constraints from our SM example above, which we recall in the listing below:

```
1 :- match(L1,D), match(L,D), L != L1. :- match(L,D), match(L,D1), D != D1.
```

Grounding these constraints via traditional grounding yields a grounding size that is cubic in the domain size, as there are 3 variables. In contrast to this, BDG's grounding size is only quadratic in the domain size, as the maximum arity is 2.

More generally, let Π be a program, $\mathrm{dom}(\Pi)$ be the domain of the program and a be the maximum arity of the program. Then we depict the grounding size of Π grounded with BDG in Equation (1.2):

$$\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^{a}) \tag{1.2}$$

Comparing Equation (1.2) to Equation (1.1), observe that while traditional grounding has a grounding size exponential in the number of the variables, BDG has a grounding size exponential in the maximum arity. We show in Figure 1.3 a schematic comparison of traditional state-of-the-art techniques (Figure 1.3a), to BDG (Figure 1.3b).

1.2 Challenges and Contributions

Body-decoupled grounding shows significant improvements upon the state-of-the-art on the grounding bottleneck. However, we identify two main challenges that hinder its widespread adoption. The first one is its *limited interoperability* with other state-of-the-art techniques. This problem is intensified by BDG being best used for fragments, or even single rules, of a program. The intuitive reason for that is that BDG pushes effort from grounding to solving and, therefore, should only be used on grounding-heavy problems. An additional reinforcing factor is that it is currently unclear *for which parts* or for which rules BDG should best be used.

The second major deficiency of BDG is its poor performance for (non-ground) *normal* and *non-tight* rules. Normal rules are rules with a non-empty head. Non-tight rules have a cycle in the respective positive dependency graph. Let Π be a program, $\mathrm{dom}(\Pi)$ be its domain, and a be its maximum arity, then the grounding size of the program is in $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^{a})$ for constraints, in $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^{2 \cdot a})$ for normal rules, and $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^{3 \cdot a})$ for non-tight (normal) parts. For practical problems, every increase beyond *domain size* to the power of a is infeasible.

We show a detailed discussion of the current major challenges of BDG in Section 3.3.

1.2.1 Aim of this Thesis

In its entirety, this thesis aims at easing the grounding bottleneck by (1) investigating the interoperability of the body-decoupled grounding approach with other state-of-the-art grounding techniques and (2) advancing BDG to be competitive with state-of-the-art grounders and solvers, for normal and non-tight rules.

1.2.2 Contributions

We consider the following four points as our main contributions, which are in line with our two main identified challenges of BDG and the general aim of the thesis:

1. *Hybrid grounding* (Chapter 4): Enabling the interoperability between body-decoupled grounding and state-of-the-art semi-naive grounders. We published hybrid grounding as a part of our publication at IJCAI24 [10].
2. *Automated hybrid grounding* (Chapter 5): The development and testing of heuristics for deciding when a BDG usage is beneficial.
3. *FastFound* (Chapter 6): Improving the BDG reduction for normal ASP programs, from being exponential in $2 \cdot a$, to only being exponential in $a + 1$.
4. *Lazy-BDG* (Chapter 7): Making cyclic programs feasible for BDG by deferring cyclical evaluations to the solving phase.

In addition to the four major contributions in the bullet points above, we count our identification of BDG's current issues (Section 3.3) and the novel partial applicability result for BDG in non-ground disjunctive programs (Section 4.3), as our minor results.

While we already published hybrid grounding [10], we are working on publications for the rest of the contributions.

1.3 Structure of this Thesis

This introductory chapter (Chapter 1) provides a gentle introduction into the field of Answer Set Programming and the grounding bottleneck. Further, it states the main contributions of this thesis (Section 1.2) and an extensive related work section (Section 1.4).

In Chapter 2 we provide the reader with general background on the field of Answer Set Programming and related concepts, while in Chapter 3 we give a detailed background on body-decoupled grounding. We provide the reader with information on our understanding of symbolic AI (Section 2.1), an introduction to Answer Set Programming (Section 2.2), necessary concepts from complexity theory (Section 2.3), background on current state-of-the-art grounders (Section 2.4), and an introduction how current state-of-the-art solvers work, with a focus on how they handle non-tight rules (Section 2.5). In the BDG, chapter we first introduce the BDG reduction in Section 3.1, followed by the current state-of-the-art of BDG (Section 3.2), and close with our discussion on the current major challenges that BDG faces (Section 3.3).

In Chapter 4 we introduce hybrid grounding [10]. We provide an intuition of Hybrid Groundin in Section 4.1, show the details of the hybrid grounding reduction in Section 4.2, and obtain novel results on the applicability of BDG in Section 4.3.

Chapter 5 introduces *automated hybrid grounding*, where we first recall the main results of the state-of-the-art techniques (Section 5.1), followed by the introduction of our heuristics

(Section 5.2), and discuss its implementation in `newground3` (Section 5.3). We close the chapter by showcasing the experimental results of `newground3` in Section 5.4.

Chapter 6 introduces *FastFound*, the improved BDG reduction for normal rules. We state the reduction in Section 6.1, extend it to HCF programs in Section 6.2, show examples thereof in Section 6.3, and close with our conducted experiments in Section 6.4.

Our final main chapter is Chapter 7 which is all about how cyclic rules can be handled with BDG and hybrid grounding. We first demonstrate how BDG dismantles cycles (Section 7.1), go on to state the level mappings technique for hybrid grounding (Section 7.2), discuss a first idea how cycles can be repaired (Section 7.3), and finally state the lazy cycle repair algorithm (Lazy-BDG) in Section 7.4. The chapter is closed by the experiments section (Section 7.5).

Finally, Chapter 8 closes this thesis with a summary and conclusions.

1.4 Related Work

Although the roots of Answer Set Programming (ASP) [71] can be traced back as far as Prolog, most important work was achieved by defining the *stable model semantics* [73] and the introduction of answer sets [72]. Another major contribution was the definition of the *Logic of Here-and-There* [115], which has led to years of fruitful research on the theoretical foundations of ASP [20], [112]. Overviews of these developments are provided by [19], [44], [82], [119].

A significant part of ASP's practical success can be attributed to the availability of grounders and solvers from early on. Historically, the SMODELS system [122] was one of the earliest solvers, followed by the DLV system [93]. These were followed by SAT-based solvers like *Cmodels* [96] and *ASSAT* [99]. Nevertheless, research is still conducted in this direction [80]. However, arguably the most important development in solvers was the development of *conflict-driven answer set solving* systems [63], such as *CLASP* [59] and *WASP* [3]. Internally, these systems use a *conflict-driven-nogood-learning* (CDNL) algorithm [63], which is similar to *conflict-driven-clause-learning* (CDCL) [101] in SAT-solving.

Grounding [85] is the instantiation of variables with domain values. Most modern solvers have a *grounder* attached to them. Historically, to enable finite groundings, programs were forced to comply with $\omega-$, or $\lambda-$restricted programs [69], [125]. Parallel to $\lambda-$restricted programs, the first version of *Gringo* the grounder was introduced [69]. State-of-the-art (SOTA) grounders, like the latest version of *Gringo* [56], the grounder of DLV [47], and $\mathcal{I}$-DLV [23], use more involved instantiation techniques, such as semi-naive database evaluation techniques. Further, `idlv` uses advanced treewidth-guided rule-rewriting techniques and heuristics to decide when to use them [24]. Further, machine learning based rewriting heuristics have been evaluated [102].

In SOTA systems, grounders and solvers are combined into one coherent system, like *Clingo* [58]. The so-called *ground- and solve-paradigm* stems from the inherent grounding and solving phases of these systems. Multi-shot-solving reduces the amount of grounding necessary for repeated grounding calls [60]. For communication between grounders and solvers intermediate languages are used. `idlv` uses the *SMODELS Lparse* [124] format, while `gringo` uses the ASPIf (ASP

Intermediate Format) [86]. Other intermediate languages have been proposed [81], such as *ASPils* (ASP Intermediate Language Standard) [57]. An overview of ground-and-solve systems is given in [87]. ASP competitions enable the comparison of systems and encodings. Typically, ASP competitions feature system tracks that compare ASP systems on a fixed encoding. There have been seven ASP competitions [2], [25], [26], [35], [65]–[67], where an overview is given in [68]. We took benchmark instances and encodings from the fifth competition [25].

As the grounding step is highly exponential (in the worst case) and the evaluation of non-ground ASP programs is NEXPTIME complete in general [31], large groundings may prohibit the evaluation of programs. This problem is called the *grounding bottleneck* [64], [126].

Multiple techniques were developed to cope with the grounding bottleneck. The first discussed is lazy-grounding, which combines grounding and solving into a conceptually single step. The goal is to prevent unnecessary instantiations of rules, which is achieved by instantiating rules if their body is satisfied. Although the technique is promising, SOTA lazy-systems are not competitive with SOTA-ground-and-solve systems [64]. Prominent traditional lazy-grounding approaches are GASP [30], ASPeRiX [91], and Omiga [32], which rely on sequences of *firing* rules, on which also the respective search space relies (which is different from the predicate (atom) focused CDNL approaches). The lazy-grounding system of ALPHA [130] adapts the CDNL-algorithm to combine the benefits of lazy-grounding, with the benefits of CDNL. Other research has extended the input language of lazy-solvers to include aggregates [17]. Also, some techniques try to use lazy-grounding with specialized search-techniques [33]. Further, others investigate and introduce prominent solver techniques in lazy-grounding, like domain-independent heurstics for speeding up the procedure [94], [131].

An approach related to lazy-grounding is the compilation based approach. Herein, parts of a program are typically grounded in a traditional way and serve as a domain, whereas the other parts are *compiled* into efficient subprocedures. These subprocedures take over both the grounding and solving step of these parts, and in order to ensure semantics they inject additional constraints, or nogoods.

In one of the earlier compilation based approaches, a partial answer set is iteratively extended by a compiled process that injects additional constraints [29]. This compiled process uses compiler techniques to be competitive, such as loop unrolling. A very similar approach is DualGrounder [97], which instead of compiling rules to optimized *C++* procedures, exploits clingo's multi-shot capabilities.

The other prominent branch of compilation techniques uses extensions of the CDNL-procedure (propagators), which was demonstrated to be a feasible approach [27]. The first pure propagator based approach included the compilation of constraints in ASP [28]. In contrast to the constraint injecting approach, this version directly extends the set of nogoods of the CDNL-algorithm. Later versions extend the syntax of ASP to aggregates [103] and to tight programs [39]. Further, traditional grounding techniques have been combined with compilation based techniques [38].

ASP Modulo Theory is yet another approach that can be used to circumvent the grounding bottleneck. It does this by integrating the best methods and tools of other fields, like constraint programming [63]. ASP-solvers enhanced with CSP are CLINGCON [7], [113], EZCSP [6], and

EZSMT [123]. For a comparative analysis of these see [95]. Also, ASP-solvers enhanced with mixed-integer programing exist, such as MINGO [100]. Additional developments include the language extension of HEX-programs, which eases the integration of external problem solving tools [41], or the bound-founded ASP extension, with the system named CHUFFED [5].

These prominent examples share that they extend the ASP input language with other language constructs. Further, these language constructs are nowadays implemented mainly by extending the CDNL-algorithm loop of modern solvers [58], [84].

In stark contrast to the developments above, recently a complexity theoretic grounding approach [10], [12] was introduced. Its foundations lie in the complexity theoretic discussions of ASP [31], where it is a well known result that ground disjunctive ASP is Σ_2^P complete [40]. Later results showed that under the assumption of bounded predicate arities, non-ground normal ASP is also Σ_2^P complete. For this reason [12] introduced the body-decoupled grounding (BDG) reduction from non-ground normal to ground disjunctive ASP. This work also showed excellent practical results upon instances having the grounding bottleneck. Later work added aggregates to the input language and extended the reduction to a hybrid grounding procedure, where programs are split (manually) into parts grounded by the BDG reduction and parts that are grounded by other state-of-the-art methods [10]. Other related complexity theoretic approaches extend the input language to disjunctive ASP, with a reduction to epistemic logic programming (ELP) [13], or propose a reduction from ELPs to disjunctive ASP [15].

Completely different approaches than the ones discussed thus far are those that make use of heuristics to improve grounding [24] and an approach that tries to estimate the grounding sizes of logic programs [77]. Further, some approaches facilitate treewidth to split up big programs into smaller programs [107], which was significantly extended in the LPOPT tool [14].

A well known alternative technique is *metaprogramming* [84]. The core concept behind metaprogramming is to give a program another program as an input, where computations are performed on the input program. Metaprogramming has many applications in ASP. The compilation based approaches discussed above, utilize the metaprogramming paradigm, although the metaprograms are not written in ASP. Using ASP-only metaprograms, one can alter the semantics of ASP [84]. Also, metaprogramming was used in various different approaches, such as for *manifold programs*, which enables the computation of consequences (such as brave/cautious) [48]. This concept was used in the ASPRIN system as well, which introduces advanced preference handling for ASP [18]. Further, metaprogramming was used in the planning domain for PLASP [37]. Lastly, it was used in [45] for the automatic integration of guess and check programs.

CHAPTER 2

Background

The background chapter sets the stage for the following discussion. It will provide the necessary background material and will put things into perspective to understand the subsequent thesis's main chapters. In Section 2.1 we will briefly discuss AI and symbolic AI. We continue by introducing ASP in Section 2.2, which is followed by relevant definitions from complexity theory in Section 2.3. The remaining sections discuss state-of-the-art techniques for the ground-and-solve paradigm in detail, namely semi-naive grounding in Section 2.4 and Conflict-Driven-Nogood-Learning in Section 2.5.

2.1 (Symbolic) Artificial Intelligence

Artificial Intelligence (AI) does not have one generally accepted definition. In [116], they classify definitions according to two axes: the *thinking/acting* axis and the *human-like/rational* axis. *Acting* definitions put a focus on the (externally) perceived human likeness or rationality of an agent. *Thinking* definitions focus on the internals of the agent. Be it in the sense that an agent's internal processes *think* like a human or like a rational agent (in the logical sense).

Our working definition puts a focus on the *acting* side, with no distinction between *human-like* and *rational* concepts: *Artificial Intelligence consists of the concepts, introduced by entities, that (should) reproduce (fragments of) intelligence*[1]. With this definition, we emphasize that AI was developed by some *entities* and is therefore artificial. Further, we do not limit the definition to any particular technique or paradigm, like neural-networks or logic programming

[1]To the best of our knowledge, this (exact) definition has not been stated before. Our definition was shaped by definitions from lectures, subsequent discussions thereof with fellow students, professors, or colleagues. More concretely, this definition was influenced by the lectures of Thomas Eiter (Intro. to AI) and Wulfram Gerstner (Comp. Neurosc.: Neur. Dyn.), by the lectures and discussions of the first, second, and third edition of the ACM Digital Humanism Summer School (2022, 2023 and 2024 respectively; chaired by Stefan Woltran et. al.), by the first AI Summer School 2023 at TU Wien, and by various books and publications, such as [75], [83], [89], [104], [106], [116], [118], [120], [128], [132].

(*concepts*). Additionally, we do not dare to define what intelligence is ("*[...] (should) reproduce [...] intelligence.*"). Moreover, we consider work that focuses on a particular *fragment*[2] of intelligence as part of AI.

The field of AI is generally split into the sub-fields of *symbolic*, *subsymbolic*, and *neurosymbolic* AI. The distinction between symbolic and subsymbolic approaches can be made on their level of computation, while neurosymbolic techniques are the combinations thereof. Symbolic approaches operate on the level of manipulating high-level *symbols* [55], while subsymbolic approaches take computation to levels below, such as vector manipulations. We illustrate their difference by stating how the two approaches differ in solving the following example: *If there is a cat and the cat is petted, then the cat purrs.* This can be encoded in the following logical rule:

$$\text{purr} \leftarrow \text{cat}, \text{pet}$$

A symbolic approach then knows the logical rule and infers *purr* from knowing whether both *cat* and *pet* are present in its knowledge base. On the contrary, a subsymbolic approach has vector embeddings of the atoms *cat*, *pet*, and *purr*, and of the implication ($\leftarrow$). The inference is then based on manipulating the vectors of *cat* and *pet* in the context of the implication ($\leftarrow$), from which it (approximately) derives the vector *purr*.

Typically, *symbolic* approaches are considered to be exact and discrete, while *subsymbolic* approaches are probabilistic and continuous[3]. Prominent examples of symbolic AI include logic programming (such as Prolog [88] or ASP [71]), automated theorem proving (such as Z3 [108]), planning (such as STRIPS [51]), or expert systems (such as R1 [105]), while subsymbolic AI includes deep-learning [90], reinforcement-learning [121], or large language models [89] and its underlying transformer architecture [129].

This thesis belongs to the field of symbolic AI, with a specific focus on the ASP paradigm. Rooted in symbolic AI, ASP has a precisely defined syntax and semantics.

We conclude by putting two pointers on what AI is: In the seminal work *the Philosphy of AI and the AI of Philosphy* by McCarthy [104], the field of AI is discussed from a philosophical standpoint. They bring presuppositions of AI to light, provide advice on how philosophers should deal with AI, and mainly discuss symbolic AI. Two of the *Three problems in computer science* (by Valiant [128]), can be attributed to AI. The *characterization of a semantics for cognitive computation*, can be seen as a problem how to combine symbolic and subsymbolic AI. Further, the *characterization of cortical computation* is concerned with the study of "how knowledge is represented in the brain and what the algorithms are for computing the most basic behavioral tasks" [128], which can be seen as the problem of how to obtain general AI.

2.2 Answer Set Programming

Answer Set Programming (ASP) [71] is a popular problem-modelling- and solve-paradigm, rooted in symbolic AI and logic programming. A program Π is defined as a set of rules $r \in \Pi$.

[2] Like logical thinking, pattern recognition, or planning.

[3] This is in striking similarity with the distinction between thinking *fast* and *slow* by Kahneman [83].

From a practical perspective, a program Π typically consists of an encoding (the scenario, e.g., finding a clique in a graph) and an instance (the data, e.g., a graph of 18 vertices). In more detail, an encoding often follows the guess (or sometimes called generate) and check paradigm, where one first generates possible answers, which are later restricted (checked) by constraints [44]. An important distinction in ASP programs is the ground/non-ground distinction. In contrast to ground programs, non-ground programs may have variables that are an abstract placeholder for a concrete instantiation of a domain value.

We show a small example that illustrates the ground/non-ground distinction. The following program consists of three rules, where two are facts ($b(1)$ and $b(2)$) and one is non-ground ($a(X,Y) \leftarrow b(X), b(Y)$).

```
1 b(1). b(2).
2 a(X,Y) :- b(X), b(Y).
```

Grounding[4] refers to the instantiation of the variables by their domain. In the example, grounding translates to instantiating the variables X and Y with the domain values 1 and 2, which produces the following ground program:

```
1 b(1). b(2).
2 a(1,1) :- b(1), b(1). a(1,2) :- b(1), b(2).
3 a(2,1) :- b(2), b(1). a(2,2) :- b(2), b(2).
```

Notice that the one non-ground rule gave rise to four ground rules.

For describing rules, we use the $\leftarrow$ notation for theoretical concepts and algorithms, while we use the *:-* notation for listings, as they are consistent with the ASP-Core-2 standard [22]. Further, we use in theoretical discussions $\neg$ and $\vee$, while we use *not* and | for the listings respectively. In the following, we introduce ASP along the ground (Section 2.2.1), non-ground (Section 2.2.2) distinction, and close with a brief discussion on applications (Section 2.2.4). The following definitions of ASP were mainly taken from [10], [12], [44].

2.2.1 Ground ASP

Ground ASP programs consist of ground rules. Ground rules contain atoms a_i. As the name suggests, atoms are indivisible. Indivisible in the sense that they are the smallest building blocks semantics can be assigned to, which is assigned in a propositional way, so either true or false (potentially unknown when including strong negation).

Syntax

We show in Equation (2.1) a (disjunctive) ground rule r, of a program Π ($r \in \Pi$). a_i are atoms and i is a non-negative integer. The head H_r is defined as $H_r \coloneqq \{a_1, \ldots, a_l\}$, the (positive) body B_r^+ as $B_r^+ \coloneqq \{a_{l+1}, \ldots, a_m\}$, the (negative) body B_r^- as $B_r^- \coloneqq \{a_{m+1}, \ldots, a_n\}$, and the

[4] The concept of grounding in ASP is not to be confused with grounding in other areas of symbolic AI, which typically stated as the *grounding problem*, is the problem of assigning meaning to abstract symbols [76].

body as $B_r := B_r^+ \cup B_r^-$. Further, the size of a rule is $|r| = |H_r \cup B_r^+ \cup B_r^-|$ and of a program $|\Pi| = \sum_{r \in \Pi} |r|$.

$$\underbrace{a_1 \vee \ldots \vee a_l}_{H_r} \leftarrow \underbrace{\overbrace{a_{l+1}, \ldots, a_m}^{B_r^+}, \overbrace{\neg a_{m+1}, \ldots, \neg a_n}^{B_r^-}}_{\text{body}:B_r := B_r^+ \cup B_r^-} \tag{2.1}$$

A rule r is *normal* iff $|H_r| \leq 1$. In the special case of $|H_r| = 0$, r is considered to be a *constraint*. $|H_r| > 1$ is called *disjunctive*. If a program Π contains a disjunctive rule r, it is called a *disjunctive program*. r is a *fact* whenever $|H_r| = 1$ and $|B_r^+ \cup B_r^-| = 0$. We denote the set of facts as $\mathcal{F}$. A normal program consists only of normal rules. The set of *atoms* $\mathrm{at}(r)$ occurring in r as $\mathrm{at}(r) := H_r \cup B_r^+ \cup B_r^-$ and the set of atoms of the entire program $\mathrm{at}(\Pi)$ as $\mathrm{at}(\Pi) := \bigcup_{r \in \Pi} \mathrm{at}(r)$ (we sometimes write at instead of $\mathrm{at}(\Pi)$). We define $body(\Pi) := \{B_r \mid r \in \Pi\}$ and $body_\Pi(p) := \{B_r \mid r \in \Pi, p \in H_r\}$. heads$(\Pi)$ is defined as heads$(\Pi) := \{p \mid p \in H_r, r \in \Pi\}$.

Graph Theory for Ground ASP

We assume the reader to be familiar with the basic concepts of graph theory. Therefore, we only introduce notions of graph theory necessary for the following discussion. More information can be found in any graph theory textbook, such as [36]. A graph G is a mathematical tuple $G = (V, E)$, where V is the set of vertices and E is the set of edges. In directed graphs, an edge $(u, v) \in E$ has a direction from u to v, whereas in an undirected graph $(u, v) \in E$, there is no direction.

v is reachable from u (denoted as vRu), iff there exists a corresponding sequence of edges $((u, v_1), \ldots, (v_n, v))$. A cycle exists if v is reachable from v. A graph is cyclic if it contains a cycle (vRv). For a directed graph, a strongly connected component (SCC) is a set of vertices, iff $\forall v, u \in \text{SCC} : v \neq u \rightarrow vRu$. A directed acyclic graph (DAG) is a directed graph that has no cycles.

Definition 2.1 (Reduction of a directed graph)**.** *The reduction $\mathcal{G}^R$ of a directed graph $\mathcal{G}$, is defined as $\mathcal{G}^R = (V^R, E^R)$, s.t. V^R are the SCCs of $\mathcal{G}$, whereas E^R is given by the edges between the SCCs.*

Observe that any $\mathcal{G}^R$ is a directed-acyclic graph (DAG). Next we introduce the dependency graphs of a program Π and define stratified, tight, and head cycle free programs.

Definition 2.2 (Dependency Graph (Ground ASP))**.** *We denote the dependency graph of a program Π as $\mathcal{D}$. The vertices are defined as: $V := \bigcup_{r \in \Pi} H_r \cup B_r^+ \cup B_r^-$. The edges are defined as: For every $r \in \Pi$, there is an edge $(b, h)_+$ for every two atoms $b \in B_r^+$, $h \in H_r$ and for every $r \in \Pi$, there is an edge $(b, h)_-$ for every two atoms $b \in B_r^-$, $h \in H_r$. We consider an edge $(b, h)_+$ as positive and an edge $(b, h)_-$ as negative.*

Definition 2.3 (Stratified Program)**.** *A program Π is not stratified, iff there is a cycle in its dependency graph $\mathcal{D}$ that contains at least one negative edge. Conversely, a program Π is stratified iff no such cycle exists.*

Definition 2.4 (Positive Dependency Graph (Ground ASP))**.** *We denote the positive dependency graph of a program* Π *as* $\mathcal{D}_P$*. The vertices are defined as:* $V \coloneqq \bigcup_{r \in \Pi} H_r \cup B_r^+$*. The edges are defined as: For every* $r \in \Pi$*, there is an edge* (b, h) *for every two atoms* $b \in B_r^+$*,* $h \in H_r$*.*

Given the program Π, we denote with $SCC(\Pi)$ the set of strongly connected components of the positive dependency graph. Further, given $\mathcal{D}_P = (V, E)$ and $a \in V$, then $SCC(\Pi, a)$ returns the strongly connected component of a w.r.t. $\mathcal{D}_P$. With $|SCC(\Pi, a)|$ we measure the size of the SCC, where we abuse notation[5] by stating that whenever $|SCC(\Pi, a)| = 1$, a is tight in Π. Further, let $S \in SCC(\Pi)$, then $rules(S) = \{r \mid r \in \Pi, H_r \cap S \neq \emptyset, B_r^+ \cap S \neq \emptyset\}$, and $p \in \text{hpred}(\Pi)$, then $rules(p) = \{r \mid r \in \Pi, p \in H_r\}$.

Definition 2.5 ((Tight) Normal Program)**.** *A (normal) program* Π *is not tight, iff there is a cycle in its positive dependency graph* $\mathcal{D}_P$*. Conversely,* Π *is tight iff no such cycle exists.*

Definition 2.6 (Head Cycle Free (HCF) Program)**.** *A (disjunctive) program* Π *is not HCF, iff there is a cycle in its positive dependency graph* $\mathcal{D}_P$*. Conversely,* Π *is HCF iff no such cycle exists.*

The notion of a stratified program is important for state-of-the-art grounding techniques. Further, observe our clear distinction between (tight) normal and HCF (disjunctive) programs. We use this non-standard terminology in order to compactly write *normal (HCF) programs* to express that a program may either be normal or HCF.

Example 2.1. *We show examples for non-tight and not-stratified programs. The program* Π_{G1} *shown in the next listing is non-tight. This comes as in its (positive) dependency graph* $\mathcal{D}_P = (V, E)$*:* $\{(b, a), (a, b)\} \subseteq E$*.*

```
1 a :- b
2 b :- a.
```

The next program Π_{G2} *is not-stratified. This comes as there are two negative edges in its dependency graph* $\mathcal{D} = (V, E)$*:* $\{(b, a)_-, (a, b)_-\} \subseteq E$*.*

```
1 a :- not b.
2 b :- not a.
```

The last program Π_{G3} *shows a HCF program.*

```
1 a | b | c :- d.
```

Semantics

In the following, we define semantics of ground ASP. Intuitively, for a rule $r \in \Pi$, whenever the body holds, the head must hold as well. If this is the case for all rules of the program, then we have an answer set.

[5] Note that by our definition, self-loops have a size of $|SCC(\Pi, a)| \neq 1$.

There are multiple definitions of how to characterize the semantics of ASP [98]. For our discussion, however, the well-founded semantics [117] (characterized by satisfiability and by foundedness), and the Gelfond-Lifschitz reduct [73], are the most relevant.

An interpretation $\mathcal{I}$ is a set of atoms. An interpretation satisfies a rule $r \in \Pi$ iff $(H_r \cup B_r^-) \cap \mathcal{I} \neq \emptyset$, or $B_r^+ \setminus \mathcal{I} \neq \emptyset$. If all rules of the program Π are satisfied, then the program is satisfied and $\mathcal{I}$ is called a *model*.

Taking Π_{G2} from above, we consider two interpretations: $\mathcal{I}_1 = \{a, b\}$ and $\mathcal{I}_2 = \{a\}$. Observe that both interpretations are models.

Definition 2.7 (Gelfond-Lifschitz reduct (GL))**.** *To obtain the GL reduct $\Pi^{\mathcal{I}}$ of a program Π w.r.t. $\mathcal{I}$, one (i) removes all rules $r \in \Pi$ with $B_r^- \cap \mathcal{I} \neq \emptyset$, and (ii) for all remaining rules $r \in \Pi$, removing all atoms $\neg a_i$, where $a_i \in B_r^-$.*

The GL reduct for $\mathcal{I}_1$ is $\Pi_{G2}^{\mathcal{I}_1} = \emptyset$. For $\mathcal{I}_2$ it is $\Pi_{G2}^{\mathcal{I}_2} = \{a\}$.

Definition 2.8 (Answer Set (GL))**.** *A model $\mathcal{I}$ is an answer set of a program Π, iff it is the minimal model (w.r.t. $\subseteq$) of $\Pi^{\mathcal{I}}$.*

We use $\mathcal{AS}(\Pi)$ to express the set of all answer sets of Π.

As $\mathcal{I}_1$ is not a model of $\Pi_{G2}^{\mathcal{I}_1}$, $\mathcal{I}_1$ is not an answer set. On the other hand, $\mathcal{I}_2$ is a minimal model of $\Pi_{G2}^{\mathcal{I}_2}$ and therefore an answer set.

Definition 2.9 (Foundedness)**.** *Let $\mathcal{I}$ be an interpretation, Π be a normal (HCF) program, s.t. $\mathcal{I}$ is a model of Π. Then the rule $r \in \Pi$ is suitable for justifying an atom $a \in \mathcal{I}$ iff (1) $a \in H_r$, (2) $B_r^+ \subseteq \mathcal{I}$, (3) $\mathcal{I} \cap B_r^- = \emptyset$, and (4) $\mathcal{I} \cap (H_r \setminus \{a\}) = \emptyset$ hold.*

*Given an interpretation I and a normal (HCF) program Π, s.t. I is a model of Π, then an atom $h \in I$ is **founded** iff (1) there exists a rule $r \in \Pi$, s.t. r is suitable for justifying $h \in \Pi$, and (2) there exists a function ϕ, s.t. $\phi : I \to \{0, ..., |I| - 1\}$ and for all $p \in B_r^+$ it must be the case that $\phi(p) < \phi(h)$.*

Note that there is no rule suitable for justifying in program Π_{G2} for any atom of $\mathcal{I}_1$. Therefore, $\mathcal{I}_1$ is not founded. For $\mathcal{I}_2$ there exists a rule suitable for justifying the (only) atom in $\mathcal{I}_2$. Further, as Π_{G2} is tight $\mathcal{I}_2$ is founded.

Switching to program Π_{G1}, observe that both $\mathcal{I}_1$ and $\mathcal{I}_2$ are not founded. For $\mathcal{I}_2$ this arises, as there is no rule suitable for justifying $\{a\}$. In contrast to this, for $\mathcal{I}_1$ there exist rules suitable for justifying $\mathcal{I}_1$. However, there does not exist a function ϕ.

Definition 2.10 (Answer Set (Well-Founded))**.** *A model $\mathcal{I}$ is an answer set of a normal (HCF) program Π, iff it is founded.*

For Π_{G2} the interpretation $\mathcal{I}_1$ is not an answer set, whereas $\mathcal{I}_2$ is an answer set. For Π_{G1} both interpretations are not answer sets.

Observation 2.1 (Equivalence of Answer Set Formulations)**.** *Observe that the definitions based on the Gelfond-Lifschitz reduct (GL) and the Well-Founded semantics coincide, for normal (HCF) programs.*

2.2.2 Non-ground ASP

Non-ground ASP has literals consisting of terms. A literal is similar to an atom, although it allows for additional language constructs. Grounding refers to the instantiation of variables in terms with their domain values. Semantics is defined by the grounded version of the program.

Syntax

Rules consist of literals, which consist of terms. We start by defining terms.

Definition 2.11 (Term (Adapted from [22]))**.** *A term can be a constant, a variable, an arithmetic term, or a functional term. We say a term is non-ground iff it contains a variable.*

We continue with defining literals, where our definitions are based on the ASP-Core-2 input language format [22]. For details, we refer to the standard. Note that we assume that each literal has a (multiple of) literal name(s). For predicate literals, the literal name coincides with its predicate name, whereas for comparison and aggregate literals we take all predicate names occurring inside the literal as a set.

Definition 2.12 (Literal (Adapted from [22]))**.** *We consider literals to fall into three different categories: A predicate, a comparison, and an aggregate literal.*

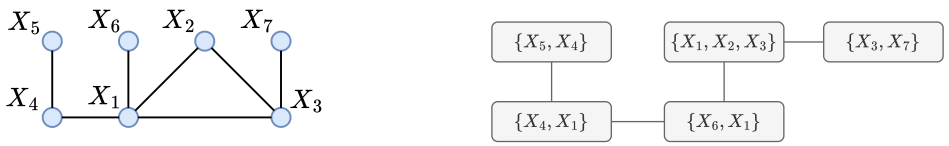
A predicate literal has the form $p(t_1, \ldots, t_n)$, where p is the predicate name and t_1 to t_n are terms. $n = |t_1, \ldots, t_n| = |\boldsymbol{T}|$ is the arity of the predicate literal.

A comparison literal has the form $t \prec u$. t and u are terms, and the comparison operator $\prec$ has the operators $\prec \in \{\text{“}<\text{”}, \text{“}\leq\text{”}, \text{“}>\text{”}, \text{“}\geq\text{”}, \text{“}=\text{”}, \text{“}\neq\text{”}\}$. We require comparison literal occurrence to be restricted to the body of the rule. Anticipating the semantic discussion, semantically the operators are defined as usual.

An aggregate literal is defined as $aggr(E) \prec u$, where $aggr \in \{count, sum, min, max\}$, E is an element multiset, $\prec \in \{\text{“}<\text{”}, \text{“}\leq\text{”}, \text{“}>\text{”}, \text{“}\geq\text{”}, \text{“}=\text{”}, \text{“}\neq\text{”}\}$, and u is a term. We require aggregate literal occurrence to be restricted to the body of the rule.

Syntactically, we define a choice literal as aggregate literals occurring in the head. It is typically written as $l \prec aggr(E) \prec u$, where l defines a lower bound and u an upper bound. The literal derives its name, as semantically it can be used to guess between u- and l-many predicates occurring in its element multiset set E.

Non-ground rules have the form displayed in Equation (2.2). We define H_r, B_r^+, B_r^-, and most other related concepts as in the ground case. However, $p(\boldsymbol{X}) \in H_r \cup B_r^+ \cup B_r^-$ is a literal. Further,



(a) Variable graph $\mathcal{D}(r_{\text{graph}})$. (b) Tree decomposition of $\mathcal{D}(r_{\text{graph}})$.

Figure 2.1: Graph structures for Example 2.3.

let $|\Pi| = \sum_{r\in\Pi} ||r||$, where $|r| = ||H_r|| + ||B_r^+|| + ||B_r^-||$, and $||\mathcal{W}|| = 1 + \sum_{p(\boldsymbol{X})\in\mathcal{W}} |\boldsymbol{X}|$, where $\mathcal{W} \in \{H_r; B_r^+; B_r^-\}$. Lastly, a set of rules is a program Π.

$$\underbrace{p_1(\boldsymbol{X}_1) \vee \ldots \vee p_l(\boldsymbol{X}_l)}_{H_r} \leftarrow \underbrace{\overbrace{p_{l+1}(\boldsymbol{X}_{l+1}), \ldots, p_m(\boldsymbol{X}_m)}^{B_r^+}, \overbrace{\neg p_{m+1}(\boldsymbol{X}_{m+1}), \ldots, \neg p_n(\boldsymbol{X}_n)}^{B_r^-}}_{\text{body:}B_r := B_r^+ \cup B_r^-} \tag{2.2}$$

Example 2.2. *The following listing depicts a rule with all types of literals. It consists of a predicate $d(X, Y)$, a comparison $X < Y$, an aggregate $\#count\{Z : e(Z, X)\} < 3$, and a choice construct $0 \leq \{a(X); b(X); c(X)\} \leq 3$.*

```
1 0 <= {a(X);b(X);c(X)} <= 3 :- d(X,Y), X < Y, #count{Z: e(Z,X)} < 3.
```

We use bold-faced vector notation for variable vectors $\boldsymbol{X} = \langle x_1, \ldots, x_m\rangle$. Checking membership of x_i is done by $x_i \in \boldsymbol{X}$. We write a non-ground literal $p(x_1, \ldots, x_m)$ as $p(\boldsymbol{X})$. Further, let $\text{hpred}(\Pi) := \{p(\boldsymbol{X}) \mid p(\boldsymbol{X}) \in H_r, r \in \Pi\}$, $\text{pred}(\Pi) := \{p(\boldsymbol{X}) \mid p(\boldsymbol{X}) \in H_r \cup B_r^+ \cup B_r^-\}$. For a rule r, let $\text{var}(r) = \bigcup_{p_i(\boldsymbol{X}_i)\in H_r\cup B_r^+\cup B_r^-} \boldsymbol{X}_i$ be the variables occurring in the rule. Further, we define $\text{hpred}(\Pi) := \{p(\boldsymbol{X}) \mid p(\boldsymbol{X}) \in H_r, r \in \Pi\}$, $\text{pred}(\Pi) := \{p(\boldsymbol{X}) \mid p(\boldsymbol{X}) \in H_r \cup B_r^+ \cup B_r^-, r \in \Pi\}$. For non-ground rules, we let $\text{at} := \{p \mid p(\boldsymbol{X}) \in H_r \cup B_r^+ \cup B_r^-, r \in \Pi\}$ be the literal names.

We assume for any rule $r \in \Pi$, that for any $p(\boldsymbol{X}) \in H_r \cup B_r^+ \cup B_r^-$ it holds that $|\boldsymbol{X}| \leq a$, where a is the arity of p. This means that we do not allow literals such as $p(f(X, Y), Z)$ to occur. However, we note that our theory still holds when we adapt the arity formulation to a *number of (global) variables per literal* formulation.

Graph Theory for Non-Ground ASP

We introduce the necessary concepts from graph theory for non-ground ASP. For additional details, we refer the reader to [36]. The notions of dependency graph (non-ground) $\mathcal{D}$ and positive dependency graph (non-ground) $\mathcal{D}_P$ carry over by switching the wording of atoms to literal names.

Definition 2.13 (Variable Graph of a rule $r \in \Pi$)**.** *Let $\mathcal{D}(r)$ be the variable graph of the rule $r \in \Pi$. We then define the undirected graph $\mathcal{D}(r) = (var(r), E)$, where there is an edge $(x_i, x_j) \in E$ iff they occur together in a literal ($\exists p_i(\boldsymbol{X}_i) \in H_r \cup B_r^+ \cup B_r^- : \{x_i, x_j\} \subseteq \boldsymbol{X}_i$).*

Example 2.3. *We show the variable graph of the following rule r_{graph} in Figure 2.1a.*

```
1 :- f(X1,X2), f(X1,X3), f(X2,X3), f(X1,X4), f(X4,X5), f(X1,X6), f(X3,X7).
```

Definition 2.14 (Tree Decomposition (Adapted from [36]))**.** *Given an undirected graph $G = (V, E)$, a tree-decomposition $\mathcal{T}$ is a tuple $\mathcal{T} = (T, \chi)$, where T is a tree, and χ is a function $\chi : T \to V$, s.t. for a $t \in T$, then $\chi(t) \subseteq V$ is called bag.*

Then $\mathcal{T} = (T, \chi)$ must fulfill the following conditions: (i) For every $v \in V$ it must hold $\exists t \in T : v \in \chi(t)$, (ii) for every $(u, v) \in E$ it must hold $\exists t \in T : \{u, v\} \subseteq \chi(t)$, (iii) every occurrence of $v \in V$ must form a connected subtree in T w.r.t. χ, i.e., $\forall t_1, t_2, t_3 \in T$, 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 decomposition is defined as the largest cardinality of a bag minus one, so $\max_{t \in T} |\chi(t)| - 1$. The treewidth is the minimal width among all decompositions (TW).

We show a (minimal) tree-decomposition of rule r_{graph} in Figure 2.1b.

Semantics

In order to obtain the answer sets of a non-ground program the program is first grounded and then the ground program is evaluated according to the semantics as defined in Section 2.2.1. Grounding refers to the process of the instantiation of the variables with their domain.

We introduce the *naive grounding* procedure by restricting ASP to non-ground ASP without arithmetics, comparison operators, or aggregates. Given a set of facts $\mathcal{F}$, let $p(\boldsymbol{D})$ be a literal, $\boldsymbol{D}$ be the (grounded) term vector, and $d \in \boldsymbol{D}$ be a term. Then, we can define the domain of the program as $\text{dom}(\Pi) := \{d \mid d \in \boldsymbol{D}, p(\boldsymbol{D}) \in \mathcal{F}\}$. For brevity, we use dom instead of $\text{dom}(\Pi)$ in some cases.

Let $\boldsymbol{X} = \text{var}(r)$, $\boldsymbol{D} \in \text{dom}(\boldsymbol{X})$, and $\boldsymbol{X}_1 \subseteq \boldsymbol{X}$. Then a subscript $\boldsymbol{D}_{\langle \boldsymbol{X}_1 \rangle}$ refers to the term vector $\boldsymbol{D}$ restricted to $\boldsymbol{X}_1$. Observe $|\boldsymbol{X}_1| = |\boldsymbol{D}_{\langle \boldsymbol{X}_1 \rangle}| \leq |\boldsymbol{X}| = |\boldsymbol{D}|$. Take for example $\boldsymbol{X} = \langle x_1, x_2, x_3 \rangle$, $\boldsymbol{D} = \langle 1, 2, 3 \rangle$, and $\boldsymbol{X}_1 = \langle x_1, x_3 \rangle$, then $\boldsymbol{D}_{\langle \boldsymbol{X}_1 \rangle} = \langle 1, 3 \rangle$.

Definition 2.15 (Naive Grounding)**.** *Given a non-ground program Π, naive grounding refers to the process where for every rule r, we instantiate every variable of r with all domain values. So, for every $r \in \Pi$, $\boldsymbol{D} \in dom(var(r))$:*

$$p_1(\boldsymbol{D}_{\langle \boldsymbol{X}_1 \rangle}) \vee \ldots \vee p_l(\boldsymbol{D}_{\langle \boldsymbol{X}_l \rangle}) \leftarrow p_{l+1}(\boldsymbol{D}_{\langle \boldsymbol{X}_{l+1} \rangle}), \ldots, p_m(\boldsymbol{D}_{\langle \boldsymbol{X}_m \rangle}), \\ \neg p_{m+1}(\boldsymbol{D}_{\langle \boldsymbol{X}_{m+1} \rangle}), \ldots, \neg p_n(\boldsymbol{D}_{\langle \boldsymbol{X}_n \rangle})$$

We show an example for naive grounding. Take the listing below:

```
1 a(1,2).
2 b(X1,X2) :- a(X1,X2).
```

Then naive grounding produces the following grounded program:

```
1 a(1,2).
2 b(1,1) :- a(1,1). b(1,2) :- a(1,2).
3 b(2,1) :- a(2,1). b(2,2) :- a(2,2).
```

Note that more intelligent grounders would produce the following program:

```
1 a(1,2). b(1,2).
```

Observation 2.2 (Grounding Size of Naive Grounding)**.** *Observe that the grounding size of naive grounding is exponential in the number of the variables. Let* $|\Pi|$ *be the size of the program,* $|dom(\Pi)|$ *be the size of the domain (inferred from the facts* $\mathcal{F}$*), and let* $\mathcal{V} = \max_{r \in \Pi} |var(r)|$ *be the maximum number of variables. Then the grounding size is in* $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{\mathcal{V}}\right)$.

Note that we sometimes write $\approx |\text{dom}(\Pi)|^{\mathcal{V}}$, or $\approx |\text{dom}|^{\mathcal{V}}$ instead of $\mathcal{O}\left(|\Pi| \cdot |\text{dom}(\Pi)|^{\mathcal{V}}\right)$ for brevity.

2.2.3 The Saturation Technique

The saturation technique [43] is an encoding paradigm that can be used to perform a for-all check, to assert a property holds for all assignments. We explain the technique along the lines of ensuring satisfiability for a rule. Consider the following program:

```
1 {b(1);b(2)}.
2 a(X) :- b(X).
```

We use saturation to explicitly encode satisfiability of the rule in the second line. The first part of the saturation technique consists of a disjunctive guess. For satisfiability this guess is the guess of all variable assignments. As the domain $\text{dom}(\Pi)$ is $\{1, 2\}$, it suffices to check variable assignments $\{X \leftarrow 1, X \leftarrow 2\}$. We show the guess in the next listing, where we encode a variable guess with a *sat_x* predicate.

```
1 sat_x(1) | sat_x(2).
```

The second part of the saturation technique is the encoding of the property. For satisfiability, this boils down to checking whether either a $b(X)$ does not hold, or an $a(X)$ holds. When any of the two conditions is fulfilled, the rule is satisfied. We encode this in the following listing.

```
1 sat :- sat_x(1), not b(1). sat :- sat_x(2), not b(2).
2 sat :- sat_x(1), a(1).     sat :- sat_x(2), a(2).
```

The final part consists in closing the loop. This is done by *saturating* the variable guesses, where we set all variables to true whenever we can derive sat. This is shown in the next listing.

```
1 sat_x(1) :- sat. sat_x(2) :- sat.
```

This example demonstrates the saturation technique.

We can extend the above example by removing the original $a(X) \leftarrow b(X)$ rule and instead guess the a-atoms by $\{a(1); a(2)\}$. Additionally, we can restrict the answer sets to the models of the original program by $\neg$sat. For answer sets we need to encode foundedness, which can be done for our toy example by specifying two constraints. We show the resulting program below.

```
1 {b(1);b(2)}. {a(1);a(2)}.
2 % Saturation technique:
3 sat_x(1) | sat_x(2).
4 sat :- sat_x(1), not b(1). sat :- sat_x(2), not b(2).
5 sat :- sat_x(1), a(1). sat :- sat_x(2), a(2).
6 sat_x(1) :- sat. sat_x(2) :- sat.
7 :- not sat.
8 % Foundedness (toy-problem):
9 :- not b(1), a(1). :- not b(2), a(2).
```

The saturation technique is a substantial part of the body-decoupled grounding approach. Note that the above example demonstrates the application of body-decoupled grounding to a toy problem. Additional information and examples can be found in [44].

2.2.4 Applications

Prominent industrial examples include a decision-support system for space shuttles [111], train scheduling for the Swiss Federal Railways [1], optimizing aircraft schedules [109], or robot-planning in factories [46]. Furthermore, symbolic approaches are an integral part of neuro-symbolic AI [53]. Examples include NeurASP [133], or a framework for explainable-AI [42].

On a didactical and practical side, *ASPChef* [4] introduces *recipes* and provides visualizations of results. On the other hand, *Clinguin* [9] introduces graphical user interfaces purely defined in ASP.

2.3 Computational Complexity Theory

Computational complexity theory "[...] is the area of computer science that contemplates the reasons why some problems are so hard to solve by computers." [114, p. v] We expect that the reader is familiar with basic notions of complexity theory, such as Turing machines and the P vs. NP debate. We mainly consider *decision problems* $\mathcal{P}$, which consist of a question and an instance. The question is a yes-no question. An introduction into these concepts is given in [114].

P are the (decision) problems that are solvable by a deterministic Turing machine in polynomial time. NP are the (decision) problems solvable by a non-deterministic Turing machine in polynomial time. co-NP are the reversed (decision) problems solvable by a non-deterministic Turing machine in polynomial time. For our purposes, we define a complexity class $\mathcal{C}$ to be the set of problems, that are decidable for a given model of computation (Turing machine), in a certain mode (deterministic vs. non-deterministic), with a given resource limit (polynomial time). Let A be a problem s.t. $\mathsf{A} \in \mathcal{C}$. Subsequently, we say A is a member of $\mathcal{C}$, or is in $\mathcal{C}$. P, NP, and co-NP

are *complexity classes*. Example: The *Horn-satisfiability* problem is in P, NP, and co-NP, the *boolean-satisfiability* problem is in NP, and the *boolean-tautology* problem is in co-NP.

One distinctive feature of the NP class is that given a "yes" instance, the solution (≈ certificate) can be verified in polynomial time by a deterministic turing machine. Conversely, a certificate for a "no" instance for the co-NP class can be verified in polynomial time by a deterministic turing machine.

Reductions: A reduction is intuitively the mapping of one problem A, to another problem B. This can be of use when efficient solvers for B exist, but not for A, or for proving complexity results.

Definition 2.16 (Many-One Reduction)**.** *Let A and B be problems, and A and B their respective sets of instances. Then, a reduction $\mathcal{R}$ is a mapping $\mathcal{R} : A \to B$, s.t. instance $i \in A$ is a positive instance of A, iff $\mathcal{R}(i) \in B$ is a positive instance of B.*

We require $\mathcal{R}$ to be efficiently computable. Therefore, $\mathcal{R}$ needs to be computable by a deterministic Turing machine in polynomial time. Further, we require that the algorithm for problem B is only called (at most) once.

Reductions define a *weaker/stronger* order between different complexity classes. For the problems A and B, and a many-one reduction $\mathcal{R}$ from A to B, we write $\mathsf{A} \leq_R \mathsf{B}$.

This gives rise to two important concepts: *Hardness* and *completeness*.

Definition 2.17 (Hardness and Completeness)**.** *Let $\mathcal{C}$ be a complexity class, and D be a problem. Then D is $\mathcal{C}$-hard if any problem $\mathsf{E} \in \mathcal{C}$ can be many-one reduced to D, so $\mathsf{E} \leq_R \mathsf{D}$.*

If additionally, D is a member of $\mathcal{C}$ ($\mathsf{D} \in \mathcal{C}$), then D is said to be $\mathcal{C}$-complete.

Prominent problems related to logic that are complete in the respective classes include the *Horn-satisfiability* problem that is P-complete, the *boolean-satisfiability* problem that is NP-complete, or the *boolean-tautology* problem that is co-NP-complete. It is widely believed that $\mathsf{P} \neq \mathsf{NP} \neq \text{co-NP}$.

Polynomial Hierarchy: Reductions define an order. The order that extends the P, NP, and co-NP classes, is called the polynomial hierarchy. Given complexity classes $\mathcal{C}$ and $\mathcal{D}$, putting $\mathcal{D}$ as a superscript of $\mathcal{C}$, defines that $\mathcal{D}$ is used as an oracle $\mathcal{C}^{\mathcal{D}}$. An oracle $\mathcal{D}$ means that all problems in $\mathcal{D}$ can be solved in constant time. Take for example P^{NP}, which is the class that solves problems in polynomial time, provided it can solve problems in NP in constant time.

The hierarchy is typically written as the symbols Δ_i^{P}, Σ_i^{P}, and Π_i^{P}, where the superscript P says that we are in the polynomial hierarchy and the subscript i defines the position in the order.

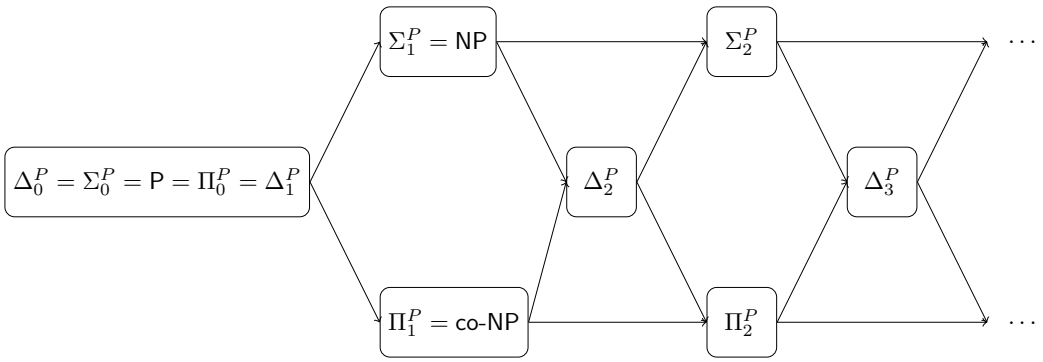


Figure 2.2: Diagram of the polynomial hierarchy. An edge (A, B) symbolizes that complexity class A is included in class B. The left side $\Delta_0^\mathsf{P} = \Sigma_0^\mathsf{P} = \mathsf{P} = \Pi_0^\mathsf{P}$ depicts the base case (Equation (2.3)). The other layers are inductively defined in Equations (2.5)–(2.7). Lastly, Δ_1^P is shown in Equation (2.4).

The hierarchy is inductively defined. Equation (2.3) shows the base case. For $i \geq 0$ the induction is shown in Equations (2.5)–(2.7), and Equation (2.4) shows the specific case for Δ_1^P.

$$\Delta_0^\mathsf{P} = \Sigma_0^\mathsf{P} = \mathsf{P} = \Pi_0^\mathsf{P} \tag{2.3}$$

$$\Delta_1^\mathsf{P} = \mathsf{P}^{\Sigma_0^\mathsf{P}} = \mathsf{P}^\mathsf{P} = \mathsf{P} \tag{2.4}$$

$$\Delta_{i+1}^\mathsf{P} := \mathsf{P}^{\Sigma_i^\mathsf{P}} \tag{2.5}$$

$$\Sigma_{i+1}^\mathsf{P} := \mathsf{NP}^{\Sigma_i^\mathsf{P}} \tag{2.6}$$

$$\Pi_{i+1}^\mathsf{P} := \text{co-NP}^{\Sigma_i^\mathsf{P}} \tag{2.7}$$

ASP and Complexity Theory: Lastly, we focus on decision problems for ASP and state their complexities. In [31], they state the computational complexity results for ASP (ground and non-ground), while in [40], they explicitly focus on bounded predicate arities. Typical ASP decision problems are:

1. *Answer Set Check:* Let Π be a program and $\mathcal{I}$ be an interpretation. Is $\mathcal{I}$ an answer set?
2. *Answer Set Existence:* Let Π be a program. Does Π have an answer set?
3. *Brave Reasoning:* Let Π be a program and p be a ground literal. Does there exist an answer set $\mathcal{I}$ of Π, s.t. $p \in \mathcal{I}$?
4. *Cautious Reasoning:* Let Π be a program and p be a ground literal. Does every answer set $\mathcal{I}$ of Π have the property $p \in \mathcal{I}$?

In the following, we will focus on the *answer set existence* problem, as this is also essential for BDG. For the complexity results, we need to distinguish between different classes of programs. In general, we can identify three axes with each having three categorical values: (i) ground/non-ground(bounded-arities)/non-ground, (ii) no negation/stratified negation/not-stratified, (iii) non-disjunctive/HCF/disjunctive. Depending on the exact combination, we obtain different results. We show the overview table, which we adapted from [40], in Table 2.1.

	$\{\}$	$\{\text{not}_s\}$	$\{\text{not}\}$
	Ground ASP		
$\{\}$	P	P	NP
$\{\vee_h\}$	NP	NP	NP
$\{\vee\}$	NP	Σ_2^P	Σ_2^P
	Non-Ground (bounded-arity) ASP		
$\{\}$	co-NP	Δ_2^P	Σ_2^P
$\{\vee_h\}$	Σ_2^P	Σ_2^P	Σ_2^P
$\{\vee\}$	Σ_2^P	Σ_3^P	Σ_3^P

Table 2.1: Complexity classes for the answer set existence problem for ground, non-ground (bounded predicate arities), and non-ground ASP. Each entry signifies membership and hardeness (completeness). Columns: $\{\text{not}\}$ non-stratified default negation, $\{\text{not}_s\}$ stratified default negation, $\{\}$ no negation. Rows: $\{\vee\}$ (cyclic/non-tight) disjunction, $\{\vee_h\}$ HCF programs (tight disjunction), $\{\}$ no disjunction (normal programs). Note the increase in complexity from ground to non-ground (bounded predicate arities) and observe classes with matching computational complexity (which give rise to possible reductions). Adapted from [40].

Generally speaking, ground normal ASP with no negation is the weakest problem (P-complete), whereas non-ground disjunctive ASP is the hardest one ($\mathsf{NEXP}^\mathsf{NP}$-complete). However, for practical purposes non-ground (general) ASP can largely be ignored, as the (maximal) arities of a program are known beforehand. Therefore, the table entries for ground and non-ground (bounded-arity) ASP are of special practical value. Observe that cells with matching complexity classes give rise to many-one reductions.

Prominent reductions for ASP are: The alternative grounding procedure BDG [12], which is from non-ground (bounded-arity) normal ASP to ground disjunctive ASP. A reduction [13] from non-ground (bounded-arity) disjunctive ASP to epistemic logic programming (ELP). Note that (ground) ELPs are also Σ_3^P-complete. The *selp* reduction [15] reduces ELPs to disjunctive ASP.

2.4 Bottom-up/Semi-naive Grounding

Bottom-up/Semi-naive grounding [23], [61] refers to the currently used state-of-the-art grounding technique used by grounders such as `gringo` or `idlv`. As we aim at integrating body-decoupled grounding (BDG) with SOTA grounding in Chapter 5, we introduce in this section the basic workings behind bottom-up/semi-naive grounding. The name bottom-up grounding stems from the way one can think of how this grounding technique works. Given a program Π, bottom-up grounding first analyzes the (positive) dependency graph $\mathcal{D}_P$ (Definition 2.2) of the program. It continues by analyzing the dependency graph for its SCCs, and creating the reduced dependency graph $\mathcal{D}_P^R$ (Definition 2.1). Next they define the topological order L_Π over $\mathcal{D}_P^R$.

Now a crucial observation is that one can ground all rules in a single $v \in V(\mathcal{D}_P^R)$, independently of its successors in L_Π. Observe that $v \in V(\mathcal{D}_P^R)$ translates to a SCC in $\mathcal{D}_P$. Therefore, bottom-

up grounding boils down to traversing L_Π and iteratively ground the vertices $v \in V(\mathcal{D}_P^R)$ (the SCCs of $\mathcal{D}_P$).

Going from intuition to a more precise formulation, let HB(Π) be the *Herbrand Base*, the set of all ground literals, which can be inferred by naive grounding. Then, let the candidate set D ($D \subseteq$ HB) be the set that is obtained via bottom-up/semi-naive grounding. D is iteratively computed by expanding the candidate set. In the base case $D = \mathcal{F}$. For each iteration, the candidate set is expanded by those head atoms that can be inferred from D. We define $ground_D(\Pi)$ as all ground rules that can be inferred from D, $\mathcal{G}(\Pi)$ is the set of ground rules (naive grounding). In more detail, if the positive body B_r^+ is in the candidate set D, then it is also in $ground_D(\Pi)$ (Equation (2.8)).

$$ground_D(\Pi) = \{r \mid r \in \mathcal{G}(\Pi), B_r^+ \subseteq D\} \tag{2.8}$$

Example 2.4. *Let* Π_{E1} *be:*

```
1 e(1,2). f(X,Y) :- e(X,Y).
2 a(X,Y) :- f(X,Y). f(Y,X) :- a(X,Y).
3 c(X,Y) :- a(X,Y). :- c(X,Y), c(Y,X).
```

Let $D = \mathcal{F} = \{e(1,2)\}$*, then* $ground_D(\Pi) = \{e(1,2); f(1,2) \leftarrow e(1,2)\}$.

The simplest form of the bottom-up algorithm is depicted in Algorithm 2.1. It does not use the topological order, but recursively calls itself to obtain a fixed-point on the newly inferred heads from Equation (2.8). In more detail, given a non-ground program Π and a candidate set D, in Line (1) Equation (2.8) is called. In Line (2) the fixed-point check is performed, in Line (3) the algorithm calls itself recursively, and in Line (5) the algorithm returns the grounded program.

Algorithm 2.1: Bottom-up-grounding; Algorithm adapted from [61].

Data: Program Π, candidate set D
Result: Grounded program G
1 $G \leftarrow ground_D(\Pi)$;
2 **if** $hpred(G) \not\subseteq D$ **then**
3 | **return** bottom-up-grounding($\Pi, D \cup$ hpred(G)) ;
4 **end**
5 **return** G

Example 2.5. *We show how Algorithm 2.1 works on program* Π_{E1}*. Initially, let* $D = \mathcal{F} = \{e(1,2)\}$*, which gets extended in* 5 *recursive calls (we write iteration henceforth). At iteration* 5 *we have* $hpred(G) \subseteq D$*. Observe how* G *is re-derived at every iteration (#It.).*

Although the above algorithm works, in principle, it has one serious drawback: it re-derives already grounded rules. Therefore, the more advanced form of the bottom-up algorithm, depicted in Algorithm 2.2, performs grounding along the topological order L_Π defined over the SCCs.

Given a program Π, and a candidate set $D = \mathcal{F}$, Algorithm 2.2 performs bottom-up grounding along the SCCs. Line (1) initializes the grounded program, Line (2) iterates over the topological order. Then, Line (3) takes the current SCC C and calls Algorithm 2.1 to infer the

#It.	D	G	hpred(G)
0	$\{e(1,2)\}$	$\{e(1,2); f(1,2) \leftarrow e(1,2)\}$	$\{e(1,2); f(1,2)\}$
1	$\{e(1,2); f(1,2)\}$	$\{e(1,2); f(1,2) \leftarrow e(1,2); a(1,2) \leftarrow f(1,2)\}$	$\{e(1,2); f(1,2); a(1,2)\}$
2	$\{e(1,2); f(1,2); a(1,2)\}$	$\{e(1,2); f(1,2) \leftarrow e(1,2); a(1,2) \leftarrow f(1,2); f(2,1) \leftarrow a(1,2); c(1,2) \leftarrow a(1,2)\}$	$\{e(1,2); f(1,2); a(1,2) f(2,1); c(1,2)\}$
3	$\{e(1,2); f(1,2); a(1,2) f(2,1); c(1,2)\}$	$\{e(1,2); f(1,2) \leftarrow e(1,2); a(1,2) \leftarrow f(1,2); f(2,1) \leftarrow a(1,2); c(1,2) \leftarrow a(1,2); a(2,1) \leftarrow f(2,1)\}$	$\{e(1,2); f(1,2); a(1,2) f(2,1); c(1,2); a(2,1)\}$
4	$\{e(1,2); f(1,2); a(1,2) f(2,1); c(1,2); a(2,1)\}$	$\{e(1,2); f(1,2) \leftarrow e(1,2); a(1,2) \leftarrow f(1,2); f(2,1) \leftarrow a(1,2); c(1,2) \leftarrow a(1,2); a(2,1) \leftarrow f(2,1); c(2,1) \leftarrow a(2,1); f(1,2) \leftarrow a(2,1)\}$	$\{e(1,2); f(1,2); a(1,2) f(2,1); c(1,2); a(2,1); c(2,1)\}$
5	$\{e(1,2); f(1,2); a(1,2) f(2,1); c(1,2); a(2,1); c(2,1)\}$	$\{e(1,2); f(1,2) \leftarrow e(1,2); a(1,2) \leftarrow f(1,2); f(2,1) \leftarrow a(1,2); c(1,2) \leftarrow a(1,2); a(2,1) \leftarrow f(2,1); c(2,1) \leftarrow a(2,1); f(1,2) \leftarrow a(2,1); \leftarrow c(1,2), c(2,1); \leftarrow c(2,1), c(1,2)\}$	$\{e(1,2); f(1,2); a(1,2) f(2,1); c(1,2); a(2,1); c(2,1)\}$

grounded program of the current SCC. In Line (4) the returned program G' is added to G, and the candidate set D is expanded. Finally, in Line (6) the grounded program is returned.

Algorithm 2.2: Ordered bottom-up-grounding; Algorithm adapted from [61].

Data: Program Π, candidate set D
Result: Grounded program G

```
1 G ← ∅ ;
2 for C in L_Π do
3 |   G' ← bottom-up-grounding(C, D) ;
4 |   (G, D) ← (G ∪ G', D ∪ hpred(G')) ;
5 end
6 return G
```

Example 2.6. *We show how Algorithm 2.2 works on program Π_{E1}. Initially, let $D = \mathcal{F} = \{e(1,2)\}$. We define the topological order L_Π to be $L_\Pi = (\{e\}; \{f; a\}; \{c\})$. Constraints are handled as soon as possible. Observe how bottom-up grounding does not re-derive already derived rules or candidate set atoms.*

#It.	D	G	G'	hpred(G)
0	$\{e(1,2)\}$	$\emptyset$	$\{e(1,2)\}$	$\{e(1,2)\}$
1	$\{e(1,2)\}$	$\{e(1,2)\}$	$\{f(1,2) \leftarrow e(1,2);$ $a(1,2) \leftarrow f(1,2);$ $f(2,1) \leftarrow a(1,2);$ $a(2,1) \leftarrow f(2,1);$ $f(1,2) \leftarrow a(2,1)\}$	$\{f(1,2); f(2,1);$ $a(1,2); a(2,1)\}$
2	$\{e(1,2)$ $f(1,2); f(2,1);$ $a(1,2); a(2,1)\}$	$\{e(1,2); f(1,2) \leftarrow e(1,2);$ $a(1,2) \leftarrow f(1,2);$ $f(2,1) \leftarrow a(1,2);$ $a(2,1) \leftarrow f(2,1);$ $f(1,2) \leftarrow a(2,1)\}$	$\{c(1,2) \leftarrow a(1,2);$ $c(2,1) \leftarrow a(2,1);$ $\leftarrow c(1,2), c(2,1);$ $\leftarrow c(2,1), c(1,2)\}$	$\{c(1,2), c(2,1)\}$

Algorithm 2.2 achieves not re-deriving grounded rules, as long as the SCCs are not cyclic. If they are cyclic, these are re-derived in Line (3) of the algorithm. For cyclic SCCs, the semi-naive grounding approach is proposed. Intuitively, semi-naive grounding keeps track of all new atoms that were instantiated in the last call to the ground-rule-procedure, and prohibits their re-instantiation. The details can be found in [61], however they are not necessary to understand the remaining chapters of the thesis.

2.4.1 Simplifications

SOTA grounders use multiple simplifications that are enabled by the bottom-up grounding procedure. Important for these simplifications is the split of the candidate set D, into D_T, the surely true literals, and D_{pot}, the potentially true atoms. Conversely, if a literal $p \notin D_T \cup D_{\text{pot}}$, p is surely false. This simplification can be titled *propagation of true atoms*. It encompasses: (1) removal of facts from the positive body, (2) discarding of rules with negative literals over a fact (3) discarding of rules whenever the head is a fact, and (4) gathering of new facts whenever a rule body is empty.

Further, they use propagation of false atoms. With these simplifications, they can evaluate certain constraints and obtain new facts. Further, they can evaluate the whole class of *stratified programs* (Definition 2.3).

Example 2.7. *Simplifications for program* Π_{E1}. *Observe that* Π_{E1} *is stratified. In Example 2.6 we saw that the constraints* $\{\leftarrow c(1,2), c(2,1); \leftarrow c(2,1), c(1,2)\}$ *are part of the grounding. As we can derive* $\{c(1,2); c(2,1)\} \subseteq D_T$, *we know that* Π_{E1} *has no answer sets.*

2.4.2 Rule Instantiation

In Algorithm 2.3, we depict the simplified rule instantiation mechanism of state-of-the-art grounders (adapted from [61]). Let $r \in \Pi$, D be the candidate set, D_T ($D_T \subseteq D$) be the surely

Algorithm 2.3: ground-backtrack$_{r,R,D}$; Algorithm adapted from [61].

Data: Substitution σ, true candidate set D_T, safe body ordering $(b_1, \ldots, b_n)$
Result: grounded rule(s) G, true candidate set D_T

```
1  if n = 0 then
2      H = {hσ | h ∈ H_r(r)};
3      B = {pσ | p ∈ B_r^+, pσ ∉ D_T} ∪ ;
4            {not aσ | a ∈ B_r^- \ R, aσ ∈ D}∪ ;
5            {not aσ | a ∈ B_r^- ∩ R} ;
6      B^- = {pσ | p ∈ B_r^-} ;
7      if B = ∅ then
8          D_T ← D_T ∪ H;
9      end
10     return ({H ← B | B^- ∩ D_T = ∅, H ∩ D_T = ∅}, D_T) ;
11 else
12     G ← ∅ ;
13     foreach σ' ∈ match_{D_T,D}(σ, b_1) do
14         (G, F) ← (G, F) ∪ ground-backtrack_{r,R,D}(σ', D_T, (b_2, ..., b_n)) ;
15     end
16     return (G, D_T) ;
17 end
```

true atoms in the candidate set, R all literals occurring recursively in $\mathcal{D}$ (overall, not positive dependency graph), σ a (ground) substitution ($\sigma : \text{var}(r) \to \text{dom}(r)$), and $(b_1, \ldots, b_n)$ a *safe body ordering*. A body ordering is safe whenever a variable occurrence in a negative body literal is preceded by a positive occurrence. Further, let $match_{D_T,D}(\sigma, b_1)$ be a set of possible substitutions, inferred by σ and D, w.r.t. D_T and D.

Then the algorithm intuitively traverses the safe body order (Lines (12)–(16)), intantiates the rule r by recursively calling itself (Line (14)), and if it finds an assignment (Lines (1)–(10)), it returns the rule (Line (10)). By backtracking (Line (14)) it generates all rules and returns them (Line (16)). In more detail, the final assignment instantiates the head (Line (2)) and the body (Lines (3)–(5)). The positive body literals are integrated if they are not surely true (Line (3)), whereas for the negative body, if they are not recursive and in the candidate set, then they are added (Line (4)), and if they are recursive, they are always added (Line (5)). The negative body is instantiated in Line (6). If the body is empty, then the head is surely true (a fact) (Lines (7)–(8)). In any case, the rule is instantiated in Line (10), as long as no literal of the negative body, or the head, is in D_T.

Although Algorithm 2.3 incorporates several simplifications, it still is in the worst case a naive instantiation procedure. Therefore, it is still exponential in the number of variables in the worst-case.

Example 2.8. *We demonstrate how Algorithm 2.3 works by an example. We want to instantiate the following rule r:*

```
1 a(X) :- b(X,Y), c(Y,Z), not d(Z).
```

We assume $D = \{b(2,1); b(1,2); c(2,3); c(2,4); c(2,5); d(4); d(5)\}$, $D_T = \{d(4); c(2,5)\}$, $R = \emptyset$, and $G = \emptyset$. We show the instantiation in Table 2.2, where we focus on σ and $(b_1, \ldots, b_n)$. Let $(b_1, \ldots, b_n) = (b(X,Y); c(Y,Z); not\ d(Z))$ be our safe body order. Note that a body ordering like $(b(X,Y); not\ d(Z); c(Y,Z))$ is not safe. We handle negative body literals in Lines (2)–(10) of the algorithm and let them pass through Lines (13)–(15). Note that in Table 2.2 we show the current iteration at Line (15) of the algorithm. The procedure yields two instantiated rules: $\{a(1) \leftarrow b(1,2), c(2,3); a(1) \leftarrow b(1,2), not\ d(5)\}$.

#It.	σ	$(b_1, \ldots, b_n)$
0	$\emptyset$	$(b(X,Y); c(Y,Z); not\ d(Z))$
1	$\{X \leftarrow 2; Y \leftarrow 1\}$	$(c(Y,Z); not\ d(Z))$
return Line (16): $(\emptyset, D_T)$, as $match_{D_T,D}(\sigma, c(Y,Z)) = \emptyset$		
2	$\{X \leftarrow 1; Y \leftarrow 2\}$	$(c(Y,Z); not\ d(Z))$
2.1	$\{X \leftarrow 1; Y \leftarrow 2, Z \leftarrow 3\}$	$(not\ d(Z))$
2.1.1	$\{X \leftarrow 1; Y \leftarrow 2, Z \leftarrow 3\}$	$()$
return Line (10): $(\{a(1) \leftarrow b(1,2), c(2,3)\}, D_T)$, as $not\ d(3) \notin D$		
2.2	$\{X \leftarrow 1; Y \leftarrow 2, Z \leftarrow 4\}$	$(not\ d(Z))$
2.2.1	$\{X \leftarrow 1; Y \leftarrow 2, Z \leftarrow 4\}$	$()$
return Line (10): $(\emptyset, D_T)$, as $B^- \cap D_T = \{d(4)\} \neq \emptyset$		
2.3	$\{X \leftarrow 1; Y \leftarrow 2, Z \leftarrow 5\}$	$(not\ d(Z))$
2.3.1	$\{X \leftarrow 1; Y \leftarrow 2, Z \leftarrow 5\}$	$()$
return Line (10): $(\{a(1) \leftarrow b(1,2), not\ d(5)\}, D_T)$, as $c(2,5) \in D_T$		

Table 2.2: Instantiation of rule $a(X) \leftarrow b(X,Y), c(Y,Z), not\ d(Z)$ according to Algorithm 2.3 and Example 2.8.

2.5 Conflict-Driven-Nogood-Learning (CDNL)

We present the main ideas and concepts behind Conflict-Driven-Nogood-Learning (CDNL) and thereby present how ASP solvers work, with a focus on preventing cyclic derivations. See [63] for a detailed discussion about the topic.

The solving step takes a ground program Π, translates the program into nogoods and then performs the *Conflict-Driven-Nogood-Learning* (CDNL) [62], [63] to obtain an answer set. Cycles are checked with the unfound-set approach and a lazy-solving algorithm. CDNL is conceptually

similar to *Conflict-Driven-Clause-Learning* (CDCL) [101], which is the SOTA algorithm for boolean-satisfiability (SAT) solving.

A *nogood* is a conjunction of (signed) atoms $\{\boldsymbol{T}\alpha_1, \ldots, \boldsymbol{F}\alpha_z\}$, which represents a forbidden assignment. Meaning, that whenever all $\boldsymbol{T}\alpha_1, \ldots, \boldsymbol{F}\alpha_z$ hold, then we know there cannot be an answer set. Such a signed atom is typically written as $\boldsymbol{T}\alpha_1$ resembling that α_1 occurs positively in the nogood, whereas $\boldsymbol{F}$ indicates a negative occurrence. Very similar to this concept are clauses (as in SAT), which are a disjunction of atoms. The relation between nogoods and clauses is that $\{\boldsymbol{T}\alpha_1, \ldots, \boldsymbol{F}\alpha_z\}$ holds iff $\neg\alpha_1 \vee \ldots \vee \alpha_z$ does not hold.

Solvers encode ASP programs as a set of nogoods. We focus in this section on the representation of the program as nogoods, with a particular emphasis on nogoods that are part of the cyclic derivation prevention. However, we provide an intuition how the CDNL algorithm works (details in [63]): Each atom is assigned a truth value ($\boldsymbol{T}$ true, or $\boldsymbol{F}$ for false). A full assignment is denoted as $\mathcal{A}$, the restriction to the true assignment as $\mathcal{A}^{\boldsymbol{T}}$, and conversely for the false assignment as $\mathcal{A}^{\boldsymbol{F}}$. CDNL iteratively assigns truth values to atoms. If a nogood is unit (all except one atom evaluates to true under $\mathcal{A}$), it uses *unit propagation* to infer that the last atom must evaluate to false. However, if in the process a nogood is true, we are in a *conflict*. Therefore, we must backtrack, use *conflict analysis and resolution* to learn new (better) nogoods, and assign other truth values. If everything is assigned and there is no conflict we have an answer set, but if we have a conflict and cannot backtrack we know that there is no answer set.

2.5.1 Tight Nogoods

Nogoods can in principle be put into two categories, those for tight and those for non-tight programs. As the nogoods for the non-tight part build upon the concept of the tight part, we start by presenting nogoods for the tight part. The nogoods of the tight part heavily depend on the concept of *Clark completion*. Therefore, we will first define the Clark completion, followed by the encoding of the nogoods. Afterwards, we will discuss the unfound-set method.

Definition 2.18 (Clark Completion, adapted from [63])**.** *The clark completion of a ground tight normal program* Π *is defined as the following propositional logic formula* $\mathcal{C}$*:*

$$\mathcal{C} = \{p_{B_r} \leftrightarrow p_2 \wedge \ldots \wedge p_m, \neg p_{m+1}, \ldots, \neg p_n \mid B_r \in \mathit{body}(\Pi), B_r = \{p_2, \ldots, p_m, \neg p_{m+1}, \ldots, \neg p_n\}\} \cup \tag{2.9}$$

$$\{p \leftrightarrow p_{B_r^1} \vee \ldots \vee p_{B_r^k} \mid p \in \mathrm{at}(\Pi), \mathit{body}_\Pi(p) = \{B_r^1, \ldots, B_r^k\}\} \tag{2.10}$$

Given a ground tight program Π *and a set of atoms* $\mathcal{I}$*, and let* $\mathcal{B}' \subseteq \mathcal{B} = \{p_{B_r} \in \mathit{body}(\Pi)\}$*.* $\mathcal{I}$ *is an answer set whenever* $\mathcal{I} \cup \mathcal{B}'$ *satisfies* $\mathcal{C}$*.*

Intuitively, Clark completion captures the notion of answer sets in a similar way to the satisfiability and foundedness notions (for tight programs). While Equation (2.9) captures the notion that a body holds whenever all of its atoms are true. Equation (2.10) encodes that if an atom p holds, there must exist a suitable body, and if there is a body that holds, then its respective head must holds as well.

This notion is directly encoded into nogoods, by defining:

$$\delta(B_r) = \{\boldsymbol{F}B_r, \boldsymbol{T}p_2, \ldots, \boldsymbol{T}p_m, \boldsymbol{F}p_{m+1}, \ldots, \boldsymbol{F}p_n\} \tag{2.11}$$
$$\Delta(B_r) = \{\{\boldsymbol{T}B_r, \boldsymbol{F}p_2\}, \ldots, \{\boldsymbol{T}B_r, \boldsymbol{F}p_m\}, \{\boldsymbol{T}B_r, \boldsymbol{T}p_{m+1}\}, \ldots, \{\boldsymbol{T}B_r, \boldsymbol{T}p_n\}\} \tag{2.12}$$
$$\Delta(p) = \{\{\boldsymbol{F}p, \boldsymbol{T}B_r^1\}, \ldots, \{\boldsymbol{F}p, \boldsymbol{T}B_r^k\}\} \tag{2.13}$$
$$\delta(p) = \{\boldsymbol{T}p, \boldsymbol{F}B_r^1, \ldots, \boldsymbol{T}B_r^k\} \tag{2.14}$$

Notice how nogood Equations (2.11)–(2.12) capture Clark's Equation (2.9), and how nogood Equations (2.13)–(2.14) capture Clark's Equation (2.10).

Using Equations (2.11)–(2.14) we obtain the following nogoods for the (whole) program Π:

$$\Delta_{\text{body}(\Pi)} = \{\delta(B_r) \mid B_r \in \text{body}(\Pi)\} \cup \{\delta \mid \delta \in \Delta(B_r), B_r \in \text{body}(\Pi)\} \tag{2.15}$$
$$\Delta_{\text{atom}(\Pi)} = \{\delta(p) \mid p \in \text{atom}(\Pi)\} \cup \{\delta \mid \delta \in \Delta(p), p \in \text{atom}(\Pi)\} \tag{2.16}$$
$$\Delta_{\Pi} = \Delta_{\text{body}(\Pi)} \cup \Delta_{\text{atom}(\Pi)} \tag{2.17}$$

Equation (2.15) captures the body nogoods, Equation (2.16) captures the nogoods for the atoms, and Equation (2.17) combines the body and the atoms nogoods.

In [63] they establish the result that an answer set of a ground tight normal program Π corresponds to the unique solution of Equation (2.17):

Theorem 2.1 (Adapted from Theorem 3.3 in [63])**.** *Let Π be a ground tight normal program and $\mathcal{I} \subseteq \text{at}(\Pi)$ be a set of atoms, and let the assignment $\mathcal{A}$ be defined as follows:*

$$\begin{aligned}\mathcal{A} =& \{\boldsymbol{T}p \mid p \in \mathcal{I}\} \cup \{\boldsymbol{F}p \mid p \in \text{at}(\Pi) \setminus \mathcal{I}\} \cup \\ & \{\boldsymbol{T}B_r \mid B_r \in \mathit{body}(\Pi), B_r^+ \subseteq \mathcal{I}, B_r^- \cap \mathcal{I} = \emptyset\} \cup \\ & \{\boldsymbol{F}B_r \mid B_r \in \mathit{body}(\Pi), \left(B_r^+ \cap (\text{at}(\Pi) \setminus \mathcal{I})\right) \cup (B_r^- \cap \mathcal{I}) \neq \emptyset\}\end{aligned}$$

Then, $\mathcal{I}$ is an answer set of Π iff $\mathcal{A}$ is the unique solution for Δ_Π s.t. $\mathcal{A}^{\boldsymbol{T}} \cap \text{at}(\Pi) = \mathcal{I}$.

Example 2.9. *To show the added nogoods in an example, let Π be:*

```
1    e(1). q(1). f(1) :- e(1), q(1). f(1) :- q(1).
```

We get the following nogoods:

$$\begin{aligned}\Delta_{\mathit{body}(\Pi)} =& \{\{\boldsymbol{F}\emptyset\}, \{\boldsymbol{F}\{e(1), q(1)\}, \boldsymbol{T}e(1), \boldsymbol{T}q(1)\}, \{\boldsymbol{F}\{q(1)\}, \boldsymbol{T}q(1)\}, \\ & \{\boldsymbol{T}\{e(1), q(1)\}, \boldsymbol{F}e(1)\}, \{\boldsymbol{T}\{e(1), q(1)\}, \boldsymbol{F}q(1)\}, \{\boldsymbol{T}\{q(1)\}, \boldsymbol{F}q(1)\}\}\end{aligned} \tag{2.18}$$
$$\begin{aligned}\Delta_{\mathit{atom}(\Pi)} =& \{\{\boldsymbol{T}e(1), \boldsymbol{F}\emptyset\}, \{\boldsymbol{T}q(1), \boldsymbol{F}\emptyset\}, \{\boldsymbol{T}f(1), \boldsymbol{F}\{e(1), q(1)\}, \boldsymbol{F}\{q(1)\}\} \\ & \{\boldsymbol{F}e(1), \boldsymbol{T}\emptyset\}, \{\boldsymbol{F}q(1), \boldsymbol{T}\emptyset\}, \{\boldsymbol{F}f(1), \boldsymbol{T}\{e(1), q(1)\}\}, \{\boldsymbol{F}f(1), \boldsymbol{T}\{q(1)\}\}\}\end{aligned} \tag{2.19}$$
$$\Delta_\Pi = \Delta_{\mathit{body}(\Pi)} \cup \Delta_{\mathit{atom}(\Pi)} \tag{2.20}$$

A short note on the facts $e(1)$ *and* $q(1)$*: Observe nogood* $\{\boldsymbol{F}\emptyset\}$ *in Equation (2.18), which de facto assigns true to* $\emptyset$*. Together with the two nogoods* $\{\boldsymbol{T}e(1), \boldsymbol{F}\emptyset\}, \{\boldsymbol{T}q(1), \boldsymbol{F}\emptyset\}$*, this leads the unit propagation to always fulfill* $e(1)$ *and* $q(1)$*.*

Let $\mathcal{I} = \{e(1), q(1), f(1)\}$ *and* $\mathcal{A}$ *be defined as* $\mathcal{A} = \{\boldsymbol{T}e(1), \boldsymbol{T}q(1), \boldsymbol{T}f(1), \boldsymbol{T}\{e(1), q(1)\}, \boldsymbol{T}\{q(1)\}\}$*. Then* $\mathcal{I}$ *is an answer set, as* $\mathcal{A}$ *is the unique solution of* Δ_Π*.*

2.5.2 Unfound-set and Loop Nogoods

In the case that a ground normal program Π is not tight, the nogoods from Δ_Π do not suffice to capture the answer set semantics. In fact, they *only* capture supported models. Therefore, one has to add additional nogoods. The method presented in [63] and also discussed here is the *unfound-set* method. The basic idea is the following: Knowing that there is a cyclic component in the ground dependency graph, one tries to find *external support* from outside of the cyclic component. External support manifests itself in the dependency graph as an incoming edge into the SCC. Alternatively, if the external support is a fact, then there is no incoming edge.

In more detail, given a ground normal program Π, let $U \subseteq \mathrm{at}(\Pi)$, we define the external bodies $\mathrm{EB}_\Pi(U)$ of U for Π as:

$$\mathrm{EB}_\Pi(U) = \{B_r^r \mid r \in \Pi, h \in H_r, h \in U, B_r^+ \cap U = \emptyset\}$$

This directly leads us to the notion of the unfounded set. Intuitively, a $B_r^r \in \mathrm{EB}_\Pi(U)$ can provide non-circular support for U. Therefore, if B_r^r holds for an assignment $\mathcal{A}$, it provides non-circular support. Conversely, if no external body can be found for U that holds for $\mathcal{A}$, we find an unfounded set.

Definition 2.19 (Adapted from Definition 3.1 in [63])**.** *Let* Π *be a ground normal program,* $\mathcal{A}$ *an assignment and* $U \subseteq \mathrm{at}(\Pi)$*. Then* U *is an unfounded set of* Π *w.r.t.* $\mathcal{A}$*, iff* $EB_\Pi(U) \subseteq \mathcal{A}^{\boldsymbol{F}}$*.*

With the notion of unfound set defined, we continue to define the loop formula for $U \subseteq \mathrm{at}(\Pi)$ in Equation (2.21). Intuitively this loop formula encodes that if at least one atom of the set U holds, there must exist at least one external support body. This is encoded in the loop nogood in Equation (2.22). Lastly, in Equation (2.23) we add loop nogoods for all conceivable sets U.

$$\left(\bigvee_{p \in U} p\right) \rightarrow \left(\bigvee_{B_r \in \mathrm{EB}_\Pi(U)} p_{B_r}\right) \tag{2.21}$$

$$\lambda(p, U) = \{\boldsymbol{T}p, \boldsymbol{F}B_r^1, \ldots, \boldsymbol{F}B_r^k\} \tag{2.22}$$

$$\Lambda_\Pi = \bigcup_{\emptyset \subset U \subseteq \mathrm{at}(\Pi)} \{\lambda(p, U) \mid p \in U\} \tag{2.23}$$

With the combination of Λ_Π and Δ_Π we capture the answer set semantics for ground normal programs:

Theorem 2.2 (Adapted from Theorem 3.7 in [63])**.** *Given a ground normal program* Π *and a set of atoms* $\mathcal{I} \subseteq \mathrm{at}(\Pi)$*, we define the assignment* $\mathcal{A}$ *as:*

$$\begin{aligned}\mathcal{A} =&\{\boldsymbol{T}p \mid p \in \mathcal{I}\} \cup \{\boldsymbol{F}p \mid p \in \mathrm{at}(\Pi) \setminus \mathcal{I}\} \cup \\ &\{\boldsymbol{T}\beta \mid \beta \in \mathit{body}(\Pi), B_r^+ \subseteq \mathcal{I}, B_r^- \cap \mathcal{I} = \emptyset\} \cup \\ &\{\boldsymbol{F}\beta \mid \beta \in \mathit{body}(\Pi), \left(B_r^+ \cap (\mathrm{at}(\Pi) \setminus \mathcal{I})\right) \cup (B_r^- \cap \mathcal{I}) \neq \emptyset\}\end{aligned}$$

Then $\mathcal{I}$ *is an answer set of* Π *iff* $\mathcal{A}$ *is the unique solution for* $\Delta_\Pi \cup \Lambda_\Pi$ *s.t.* $\mathcal{A}^{\boldsymbol{T}} \cap \mathrm{at}(\Pi) = \mathcal{I}$*.*

Example 2.10. *We show the loop nogoods in an example. Let* Π *be:*

```
1    a(1) :- b(1). b(1) :- a(1).
```

This program has only one answer set that is empty. To check this, we generate the nogoods:

$$\Delta_{\mathit{body}(\Pi)} = \{\{\boldsymbol{F}\{b(1)\}, \boldsymbol{T}b(1)\}, \{\boldsymbol{F}\{a(1)\}, \boldsymbol{T}a(1)\}, \{\boldsymbol{T}\{b(1)\}, \boldsymbol{F}b(1)\}, \{\boldsymbol{T}\{a(1)\}, \boldsymbol{F}a(1)\}\} \tag{2.24}$$

$$\Delta_{\mathit{atom}(\Pi)} = \{\{\boldsymbol{F}a(1), \boldsymbol{T}\{b(1)\}\}, \{\boldsymbol{F}b(1), \boldsymbol{T}\{a(1)\}\}, \{\boldsymbol{T}a(1), \boldsymbol{F}\{b(1)\}\}, \{\boldsymbol{T}b(1), \boldsymbol{F}\{a(1)\}\}\} \tag{2.25}$$

$$\Lambda_\Pi = \{\{\boldsymbol{T}a(1), \boldsymbol{F}\{b(1)\}\}, \{\boldsymbol{T}b(1), \boldsymbol{F}\{a(1)\}\}, \{\boldsymbol{T}a(1)\}, \{\boldsymbol{T}b(1)\}\} \tag{2.26}$$

$$\Delta_\Pi \cup \Lambda_\Pi = \Delta_{\mathit{body}(\Pi)} \cup \Delta_{\mathit{atom}(\Pi)} \cup \Lambda_\Pi \tag{2.27}$$

Observe the two loop nogoods $\{\boldsymbol{T}a(1)\}, \{\boldsymbol{T}b(1)\}$ *in Equation (2.26). Therefore, let* $\mathcal{I} = \emptyset$ *and* $\mathcal{A} = \{\boldsymbol{F}a(1), \boldsymbol{F}b(1), \boldsymbol{F}\{b(1)\}, \boldsymbol{F}\{a(1)\}\}$*. Then* $\mathcal{I}$ *is an answer set, as* $\mathcal{A}$ *is the unique solution to* $\Delta_\Pi \cup \Lambda_\Pi$*.*

2.5.3 Unfound Set Algorithm

Although we capture with this the answer set semantics, there still remains a significant practical problem. Λ_Π is exponential in the size of Π (in general). Therefore, most solvers *lazily* add $\lambda(p, U) \in \Lambda_\Pi$, meaning they are added only if required. Algorithm 2.4 shows the adapted unfound-set algorithm from [63]. Line (1) gathers all atoms that occur in any SCC in the set S. These are the atoms we have to check for their unfoundedness. Line (3) gets a single atom from an arbitrary SCC. This amounts to selecting the *current SCC under investigation*. Therefore, Lines (3)–(17) operate on a single SCC. Line (5) checks whether U is an unfounded set. If so it is returned. Lines (8)–(16) iterate over all positive bodies. If an external support body B_r was found outside of the current SCC (Line (9)), all directly supported atoms are removed from the unfounded set U and from S (Line (11)). However, if B_r is from the current SCC its positive body atoms are added to the unfounded set (Line (14)). We simplified the algorithm from [63] by not taking into account *source* pointers. Their inclusion would reduce the number of investigated atoms in S on successive calls.

Algorithm 2.4: Adapted UnfoundSet from Algorithm 3 in [63]

Data: Logic program Π, assignment $\mathcal{A}$, and information about SCCs.
Result: An unfounded set of Π w.r.t. $\mathcal{A}$.

```
1  S = {p | p ∈ at(Π), |SCC(Π,p)| > 1, p ∈ A^T} ; /* All cyclic atoms.  */
2  while S ≠ ∅ do
3      U ← {p} s.t. p ∈ S ;          /* Take a single arbitrary p ∈ S */
4      do
5          if EB_Π(U) ⊆ A^F then
6              return U;
7          end
8          for B_r ∈ EB_Π(U) \ A^F do
9              if B_r^+ ∩ (SCC(Π,p) ∩ S) = ∅ then
10                 for q ∈ U s.t. B_r ∈ body_Π(q) do
11                     U ← U \ {q}; S ← S \ {q};
12                 end
13             else
14                 U ← U ∪ (B_r^+ ∩ (SCC(Π,p) ∩ S)) ;
15             end
16         end
17     while U ≠ ∅;
18 end
```

Example 2.11 (Ctd. Example 2.10)**.** *The following shows the workings of the unfound set algorithm for the program in Example 2.10 and an assignment* $\mathcal{A}^{\boldsymbol{T}} \cap \mathrm{at}(\Pi) = \{a(1); b(1)\}$. *We show in Table 2.3 how the UnfoundSet algorithm derives* $U = \{b(1); a(1)\}$. *This results in the following two nogoods being added, which prevents the cyclic derivation:*

$$\Lambda_\Pi(U) = \{\{\boldsymbol{T}a(1)\}, \{\boldsymbol{T}b(1)\}\}$$

#It.	Line	Event	#It.	Line	Event
0	1	$S = \{a(1); b(1)\}$	2	5	$\mathrm{EB}_\Pi(U) = \emptyset$
	3	$U = \{b(1)\}; p = b(1)$ (arbitrarily chosen)		9	**return** $U = \{b(1); a(1)\}$
1	5	$\mathrm{EB}_\Pi(U) = \{a(1)\}$			
	8	$B_r = \{a(1)\}$			
	14	$U = \{b(1); a(1)\}$			

Table 2.3: Execution of Algorithm 2.10, for Example 2.11. The algorithm derives that additional nogoods are needed.

CHAPTER 3

Body-decoupled Grounding

Body-decoupled grounding (BDG) [12] is a novel grounding procedure. Originally, it was introduced as a complexity theoretic reduction from non-ground (bounded-arity) normal, to ground disjunctive ASP. This comes as it is a well known result in ASP that non-ground (bounded-arity) normal ASP is Σ_2^P complete, as is ground disjunctive ASP [40]. Its practical usefulness stems from its performance on grounding-heavy instances. It is therefore, considered as a promising approach towards solving the grounding bottleneck.

This chapter serves as a background on body-decoupled grounding. We start by introducing the BDG reduction in Section 3.1 and show in Section 3.2 improvements and extensions of BDG. In Section 3.3 we state the deficiencies of the BDG approach. We consider the detailed discussion of the BDG shortcomings, as our first contribution, as to the best of our knowledge, this is the first time this has been done in such detail and length.

3.1 BDG Reduction for tight ASP

Body-decoupled grounding intuitively works by taking an arbitrary rule r, *decoupling* it, while still preserving semantics. By decoupling we mean grounding each literal on its own. Semantics is preserved by encoding the well-founded model semantics of ASP, thereby explicitly encoding satisfiability and foundedness. Satisfiability is encoded with saturation, which implicitly checks every variable instantiation. For foundedness, a direct approach is used, where for each guessed head-literal, an existence check is made if a suitable rule can be found, whose body is fulfilled.

Regarding the limits of BDG, recall that BDG is a reduction from non-ground (bounded-arity) normal ASP, to ground disjunctive ASP. This definition inherently *limits* the applicability of BDG to non-ground normal (HCF) programs. Conversely, disjunctive programs can not be handled in their entirety. Note that by abusing notation, we drop the explicit mentioning of "(bounded-arity)".

Guess Answer Set Candidates

$h(\boldsymbol{D}) \vee \overline{h}(\boldsymbol{D}) \leftarrow$ for every $h(\boldsymbol{X}) \in \text{heads}(\Pi), \boldsymbol{D} \in \text{dom}(\boldsymbol{X})$ (3.1)

Ensure Satisfiability

$\bigvee_{d \in \text{dom}(\mathbf{x})} \text{sat}_x(d) \leftarrow$ for every $r \in \Pi, x \in \text{var}(r)$, (3.2)

$\text{sat}_r \leftarrow \text{sat}_{x_1}(\boldsymbol{D}_{\langle x_1 \rangle}), \ldots, \text{sat}_{x_t}(\boldsymbol{D}_{\langle x_t \rangle}), \neg p(\mathbf{D})$ for every $r \in \Pi, p(\boldsymbol{X}) \in B_r^+$, $\boldsymbol{D} \in \text{dom}(\boldsymbol{X}), \boldsymbol{X} = \langle x_1, \ldots, x_t \rangle$ (3.3)

$\text{sat}_r \leftarrow \text{sat}_{x_1}(\boldsymbol{D}_{\langle x_1 \rangle}), \ldots, \text{sat}_{x_t}(\mathbf{D}_{\langle x_t \rangle}), p(\mathbf{D})$ for every $r \in \Pi, p(\mathbf{X}) \in B_r^- \cup H_r$, $\mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle$ (3.4)

$\text{sat} \leftarrow \text{sat}_{r_1}, \ldots, \text{sat}_{r_n}$ $\Pi = \{r_1, \ldots, r_n\}$ (3.5)

$\text{sat}_x(d) \leftarrow \text{sat}$ for every $r \in \Pi, x \in \text{var}(r)$, $d \in \text{dom}(\mathbf{x})$ (3.6)

$\leftarrow \neg\text{sat}$ (3.7)

Prevent Unfoundedness

$\bigvee_{d \in \text{dom}(y)} \text{uf}_y(\langle \mathbf{D}, d \rangle) \leftarrow h(\mathbf{D})$ for every $r \in \Pi, h(\mathbf{X}) \in H_r$, $\mathbf{D} \in \text{dom}(\mathbf{X}), y \in \text{var}(r), y \notin \mathbf{X}$ (3.8)

$\text{uf}_r(\mathbf{D}_\mathbf{X}) \leftarrow \text{uf}_{y_1}(\mathbf{D}_{\langle \mathbf{X}, y_1 \rangle}), \ldots, \text{uf}_{y_t}(\mathbf{D}_{\langle \mathbf{X}, y_t \rangle}), \neg p(\mathbf{D}_\mathbf{Y})$ for every $r \in \Pi_1, h(\mathbf{X}) \in H_r$, $p(\mathbf{Y}) \in B_r^+, \mathbf{D} \in \text{dom}(\langle \mathbf{X}, \mathbf{Y} \rangle)$, $\mathbf{Y} = \langle y_1, \ldots, y_t \rangle$ (3.9)

$\text{uf}_r(\mathbf{D}_\mathbf{X}) \leftarrow \text{uf}_{y_1}(\mathbf{D}_{\langle \mathbf{X}, y_1 \rangle}), \ldots, \text{uf}_{y_t}(\mathbf{D}_{\langle \mathbf{X}, y_t \rangle}), p(\mathbf{D}_\mathbf{Y})$ for every $r \in \Pi_1, h(\mathbf{X}) \in H_r$, $p(\mathbf{Y}) \in B_r^- \cup (H_r \setminus \{h(\mathbf{X})\})$, $\mathbf{D} \in \text{dom}(\langle \mathbf{X}, \mathbf{Y} \rangle), \mathbf{Y} = \langle y_1, \ldots, y_t \rangle$ (3.10)

$\leftarrow \text{uf}_{r1}(\mathbf{D}), \ldots, \text{uf}_{rm}(\mathbf{D})$ for every $h(\boldsymbol{X}) \in \text{heads}(\Pi), \boldsymbol{D} \in \text{dom}(\boldsymbol{X})$, $\{r_1, \ldots, r_m\} = \{r | r \in \Pi, h(\boldsymbol{X}) \in H_r\}$ (3.11)

Figure 3.1: The body-decoupled grounding (BDG) procedure, defined for a non-ground tight program Π (BDG(Π)). Let $|\Pi|$ be the size of a program, $|\,\text{dom}(\Pi)|$ be the size of the domain, and a be the maximum arity. Then BDG decouples rules, which results in a grounding size that is exponential in the arity: $\mathcal{O}(|\Pi| \cdot |\text{dom}(\Pi)|^{2 \cdot a})$. Adapted from [12].

We now proceed with the details of BDG, which we show in Figure 3.1 and we refer to it as the reduction BDG. BDG assumes a non-ground (tight) normal program as an input. In essence, BDG works by (i) guessing head literals (Equation (3.1)), (ii) ensuring satisfiability of the rules (Equations (3.2)–(3.7)), and (iii) ensuring foundedness of the literals in the answer set (Equations (3.8)–(3.11)). Keep in mind that although it is a reduction, it can also be seen as a rewriting procedure that takes a program Π and rewrites it into another the program BDG(Π). Recap that an interpretation $\mathcal{I}$ satisfies a rule iff $(H_r \cup B_r^-) \cap \mathcal{I} \neq \emptyset$ or $B_r^+ \setminus \mathcal{I} \neq \emptyset$. This is precisely encoded in Equations (3.2)–(3.7) with the help of the saturation technique.

The second part of the reduction encodes foundedness. For foundedness, remind yourself that $\mathcal{I}$ is an answer set iff all its atoms are founded. Foundedness for tight programs reduces to checking if every atom in $\mathcal{I}$ has a rule r that is suitable for justifying it. A rule $r \in \Pi$ is *suitable for justifying* an atom $a \in \mathcal{I}$ iff (1) $a \in H_r$, (2) $B_r^+ \subseteq \mathcal{I}$, (3) $\mathcal{I} \cap B_r^- = \emptyset$, and (4) $\mathcal{I} \cap (H_r \setminus \{a\}) = \emptyset$ hold. This is precisely encoded in the Equations (3.8)–(3.11).

In even more detail, Equation (3.1) guesses head atoms, Equation (3.2) guesses the variable assignments for a rule, Equations (3.3)–(3.4) encode satisfiability for a given variable assignment, Equation (3.5) encodes that all rules must be satisfied, Equation (3.6) is part of the saturation guess,

to check every variable assignment and Equation (3.7) encodes that if a rule is not satisfied, then it is not an answer set. Further, Equation (3.8) guesses variable assignments for the foundedness check of those variables that are not part of the head variables, Equations (3.9)–(3.10) encode foundedness for a given variable assignment and Equation (3.11) encodes that there must exist at least one rule that justifies the head.

Example 3.1. *We demonstrate the BDG rewriting on the program* Π *in the listing below.*

```
1 {f(1,1);f(1,2); f(2,1)}.
2 #program rules.
3 c(X1) :- f(X1,X2), f(X1,X3), f(X2,X3).
```

We apply BDG to the rule in Line (3). The rule has three positive body literals and one head literal. Program Π *has a domain of* $\mathrm{dom}(\Pi) = \{1; 2\}$*, also* $\mathrm{dom}(X_1) = \mathrm{dom}(X_2) = \mathrm{dom}(X_3) = \{1; 2\}$*, however* $\mathrm{dom}(\langle X_1, X_2\rangle) = \mathrm{dom}(\langle X_1, X_3\rangle) = \mathrm{dom}(\langle X_2, X_3\rangle) = \{(1, 1); (1, 2); (2, 1); (2, 2)\}$*, and* $\mathrm{dom}(\langle X_1, X_2, X_3\rangle) = \{(1, 1, 1); (1, 1, 2); (1, 2, 1); (1, 2, 2); (2, 1, 1); (2, 1, 2); (2, 2, 1); (2, 2, 2)\}$*. The next listing shows the head-guess (Equation (3.1)). Note that instead of using the disjunction we used a more compact formulation with a choice construct.*

```
1 {c(1);c(2)}.
```

The next we must encode satisfiability of the rule. We do this by applying Equations (3.2)–(3.7).

```
1  % Guess Variables:
2  sat_X1(1)|sat_X1(2).  sat_X2(1)|sat_X2(2).  sat_X3(1)|sat_X3(2).
3  % f(X1,X2):
4  sat_r:-sat_X1(1),sat_X2(1),not f(1,1).sat_r:-sat_X1(1),sat_X2(2), not f(1,2).
5  sat_r:-sat_X1(2),sat_X2(1),not f(2,1).sat_r:-sat_X1(2),sat_X2(2), not f(2,2).
6  % f(X1,X3):
7  sat_r:-sat_X1(1),sat_X3(1),not f(1,1).sat_r:-sat_X1(1),sat_X3(2), not f(1,2).
8  sat_r:-sat_X1(2),sat_X3(1),not f(2,1).sat_r:-sat_X1(2),sat_X3(2), not f(2,2).
9  % f(X2,X3):
10 sat_r:-sat_X2(1),sat_X3(1),not f(1,1).sat_r:-sat_X2(1),sat_X3(2), not f(1,2).
11 sat_r:-sat_X2(2),sat_X3(1),not f(2,1).sat_r:-sat_X2(2),sat_X3(2), not f(2,2).
12 % c(X1):
13 sat_r:-sat_X1(1), c(1). sat_r:-sat_X1(2), c(2).
14 % All rules satisfied:
15 sat:-sat_r.
16 % Saturation:
17 sat_X1(1):-sat. sat_X1(2):-sat. sat_X2(1):-sat. sat_X2(2):-sat.
18 sat_X3(1):-sat. sat_X3(2):-sat.
19 % Ensure satisfiability:
20 :- not sat.
```

Finally, we encode foundedness of our rule, by applying Equations (3.8)–(3.11).

```
1 % Guess Variables:
2 1{uf_X2(1,1);uf_X2(1,2)}1 :- c(1). 1{uf_X2(2,1);uf_X2(2,2)}1 :- c(2).
3 1{uf_X3(1,1);uf_X3(1,2)}1 :- c(1). 1{uf_X3(2,1);uf_X3(2,2)}1 :- c(2).
4 % f(X1,X2):
5 uf_r(1):-uf_X2(1,1), not f(1,1). uf_r(1):-uf_X2(1,2), not f(1,2).
6 uf_r(2):-uf_X2(2,1), not f(2,1). uf_r(2):-uf_X2(2,2), not f(2,2).
```

```
% f(X1,X3):
uf_r(1):-uf_X3(1,1), not f(1,1). uf_r(1):-uf_X3(1,2), not f(1,2).
uf_r(2):-uf_X3(2,1), not f(2,1). uf_r(2):-uf_X3(2,2), not f(2,2).
% f(X2,X3):
uf_r(1):-uf_X2(1,1), uf_X3(1,1), not f(1,1).
uf_r(1):-uf_X2(1,1), uf_X3(1,2), not f(1,2).
uf_r(1):-uf_X2(1,2), uf_X3(1,1), not f(2,1).
uf_r(1):-uf_X2(1,2), uf_X3(1,2), not f(2,2).
uf_r(2):-uf_X2(2,1), uf_X3(2,1), not f(1,1).
uf_r(2):-uf_X2(2,1), uf_X3(2,2), not f(1,2).
uf_r(2):-uf_X2(2,2), uf_X3(2,1), not f(2,1).
uf_r(2):-uf_X2(2,2), uf_X3(2,2), not f(2,2).
% Prevent Unfoundedness:
:-uf_r(1). :-uf_r(2).
```

Solving and restricting BDG(Π) *to* $\mathrm{at}(\Pi)$*, yields the* 8 *desired answer sets:* $\mathcal{AS}(BDG(\Pi))\cap\mathrm{at}(\Pi) = \mathcal{AS}(\Pi) = \{\emptyset; \{f(2,1)\}; \{f(1,2)\}; \{f(1,2); f(2,1)\}; \{f(1,1); c(1)\}; \{f(1,1); f(1,2); c(1)\}; \{f(1,1); f(2,1); c(1); c(2)\}; \{f(1,1); f(1,2); f(2,1); c(1); c(2)\}\}$. *Note that the foundedness part of literal* $f(X_2, X_3)$ *has a cubic size.*

3.2 Extensions and Improvements of BDG

The original BDG reduction BDG, as shown in Figure 3.1, assumes a non-ground (tight) normal program as an input. Besides introducing the original formulation in [12], they further extended BDG to normal programs and introduced the variable justifying independence technique. Specialized aggregate rewriting techniques were introduced as a part of the publication in [10].

In the following section, we will briefly introduce BDG for normal ASP, the variable justifying independence technique, and aggregate rewritings for BDG. Note that we consider the extension to Epistemic Logic Programming [13] as out of scope of this thesis.

3.2.1 BDG for Cyclic Programs

To extend BDG to normal programs, in [12] they propose to use level mappings. We show this extension in Figure 3.2. The level mappings technique is an additional rewriting step that explicitly defines an order upon the atom derivation. Equation (3.12) guesses a derivation order for every pair of head-literals, Equation (3.13) prevents non-transitive derivations, and Equation (3.14) incorporates the derivation order into the foundedness check. Worst case grounding size increases to be in $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^{3 \cdot a})$.

Example 3.2. *We adapt* Π *from Example 3.1 to be a cyclic program* $\Pi_{BDG'}$ *and use level mappings to prevent cyclic derivations.*

```
{f(1,1);f(1,2);f(2,1)}.
#program rules.
c(X1) :- f(X1,X2), f(X1,X3), f(X2,X3).
f(X1,X2) :- c(X1), c(X2).
```

Add. Rules for Found. of Normal Programs:

$$[p(\boldsymbol{D}) \prec p'(\boldsymbol{D}')] \vee [p'(\boldsymbol{D}') \prec p(\boldsymbol{D})] \leftarrow \quad \text{for every } p(\boldsymbol{X}), p(\boldsymbol{X}') \in \text{heads}(\Pi), \boldsymbol{D} \in \text{dom}(\boldsymbol{X}), \boldsymbol{D}' \in \text{dom}(\boldsymbol{X}'), p(\boldsymbol{D}) \neq p'(\boldsymbol{D}') \tag{3.12}$$

$$\leftarrow [p_1(\mathbf{D}_1) \prec p_2(\mathbf{D}_2)], [p_2(\mathbf{D}_2) \prec p_3(\mathbf{D}_3)], [p_3(\mathbf{D}_3) \prec p_1(\mathbf{D}_1)] \quad \text{for every } p_1(\boldsymbol{X}_1), p_2(\boldsymbol{X}_2), p_3(\boldsymbol{X}_3) \in \text{heads}(\Pi), \boldsymbol{D}_1 \in \text{dom}(\boldsymbol{X}_1), \boldsymbol{D}_2 \in \text{dom}(\boldsymbol{X}_2), \boldsymbol{D}_3 \in \text{dom}(\boldsymbol{X}_3), p_1(\boldsymbol{D}_1) \neq p_2(\boldsymbol{D}_2), p_1(\boldsymbol{D}_1) \neq p_3(\boldsymbol{D}_3), p_2(\boldsymbol{D}_2) \neq p_3(\boldsymbol{D}_3) \tag{3.13}$$

$$\text{uf}_r(\boldsymbol{D}_{\boldsymbol{X}}) \leftarrow \text{uf}_{y1}(\boldsymbol{D}_{\langle \boldsymbol{X}, y_1 \rangle}), \ldots, \text{uf}_{y_l}(\boldsymbol{D}_{\langle \boldsymbol{X}, y_l \rangle}), \neg[p(\boldsymbol{D}_{\boldsymbol{Y}}) \prec h(\boldsymbol{D}_{\boldsymbol{X}})] \quad \text{for every } r \in \Pi, h(\boldsymbol{X}) \in H_r, p(\boldsymbol{Y}) \in B_r^+, \boldsymbol{D} \in \text{dom}(\langle \boldsymbol{X}, \boldsymbol{Y} \rangle), \boldsymbol{Y} = \langle y_1, \ldots, y_l \rangle, p(\boldsymbol{D}_{\boldsymbol{Y}}) \notin \mathcal{F} \tag{3.14}$$

Figure 3.2: Using level mappings for non-ground normal ASP. Worst case grounding size increases to be in $\mathcal{O}(|\Pi| \cdot |\text{dom}(\Pi)|^{3 \cdot a})$. Adapted from [12].

We have already shown how BDG can be used for Line (3) in Example 3.1. In the following, we show the additional needed rules defined by Equations (3.12)–(3.14). Note that using BDG for Line (4) carries over and is therefore not shown.

There are 5 *heads that need to be taken into account: heads*$(\Pi) = \{c(1); c(2); f(1,1); f(1,2); f(2,1)\}$. *The first part of the encoding guesses the order, which we show in the listing below.*

```
1 prec(c(1),c(2))|prec(c(2),c(1)).  prec(c(1),f(1,1))|prec(f(1,1),c(1)).
2 prec(c(1),f(1,2))|prec(f(1,2),c(1)). prec(c(1),f(2,1))|prec(f(2,1),c(1)).
3 prec(c(2),f(1,1))|prec(f(1,1),c(2)). prec(c(2),f(1,2))|prec(f(1,2),c(2)).
4 prec(c(2),f(2,1))|prec(f(2,1),c(2)). prec(f(1,1),f(1,2))|prec(f(1,2),f(1,1)).
5 prec(f(1,1),f(2,1))|prec(f(2,1),f(1,1)).
6 prec(f(1,2),f(2,1))|prec(f(2,1),f(1,2)).
```

The next step is to prevent non-transitive guesses (Equation (3.13)). We show in the following listing only a snippet, as there are already 30 *ground rules, which are needed for this encoding.*

```
1 :-prec(c(1),c(2)), prec(c(2),f(1,1)), prec(f(1,1),c(1)).
2 :-prec(c(1),c(2)), prec(c(2),f(1,2)), prec(f(1,2),c(1)).
3 :-prec(c(1),c(2)), prec(c(2),f(2,1)), prec(f(2,1),c(1)).
4 :-prec(c(1),f(1,1)), prec(f(1,1),f(1,2)), prec(f(1,2),c(1)).
5 % ... 26 more rules ...
```

The last check is the incorporation into the foundedness property (Equation (3.14)). We show in the listing below how the encoding of Example 3.1 has to be extended, to account for the prevention of cyclic derivations for the body literals $f(X_1, X_2)$.

```
1 % f(1,X2) < c(1):
2 uf_r(1) :- uf_X2(1,1), not prec(f(1,1),c(1)).
3 uf_r(1) :- uf_X2(1,2), not prec(f(1,2),c(1)).
4 % f(2,X2) < c(2):
5 uf_r(2) :- uf_X2(2,1), not prec(f(2,1),c(2)).
6 uf_r(2) :- uf_X2(2,2), not prec(f(2,2),c(2)).
```

Note the substantial increase in grounding size in grounding size for even a small example. This comes, as in general the prevention of non-transitive guesses (Equation (3.13)) is responsible for the increase in grounding size to $\mathcal{O}(|\Pi| \cdot |\text{dom}(\Pi)|^{3 \cdot a})$. *However, Equation (3.12) and Equation (3.14) still have a grounding size that is quadratic in the domain.*

Improved Foundedness, replacing Rules (3.9)–(3.11) of Figure 3.1

$$\mathrm{uf}_{\mathrm{rch}(\boldsymbol{Y},\boldsymbol{X})}(\boldsymbol{D}_{\langle \mathrm{rch}(\boldsymbol{Y},\boldsymbol{X})\rangle}) \leftarrow \mathrm{uf}_{y_1}(\mathbf{D}_{\langle \mathbf{X},y_1\rangle}),\ldots,\mathrm{uf}_{y_t}(\mathbf{D}_{\langle \mathbf{X},y_t\rangle}),\neg p(\mathbf{D}_{\mathbf{Y}}) \quad \text{for every } r\in\Pi, h(\mathbf{X})\in H_r, p(\mathbf{Y})\in B_r^+, \mathbf{D}\in \mathrm{dom}(\langle \mathbf{X},\mathbf{Y}\rangle), \mathbf{Y}=\langle y_1,\ldots,y_t\rangle \tag{3.15}$$

$$\mathrm{uf}_{\mathrm{rch}(\boldsymbol{Y},\boldsymbol{X})}(\boldsymbol{D}_{\langle \mathrm{rch}(\boldsymbol{Y},\boldsymbol{X})\rangle}) \leftarrow \mathrm{uf}_{y_1}(\mathbf{D}_{\langle \mathbf{X},y_1\rangle}),\ldots,\mathrm{uf}_{y_t}(\mathbf{D}_{\langle \mathbf{X},y_t\rangle}), p(\mathbf{D}_{\mathbf{Y}}) \quad \text{for every } r\in\Pi, h(\mathbf{X})\in H_r, p(\mathbf{Y})\in B_r^-\cup(H_r\setminus\{h(\mathbf{X})\}), \mathbf{D}\in \mathrm{dom}(\langle \mathbf{X},\mathbf{Y}\rangle), \mathbf{Y}=\langle y_1,\ldots,y_t\rangle \tag{3.16}$$

$$\leftarrow h(\boldsymbol{D}), \bigwedge_{r\in\{r'\mid r'\in\Pi, h(\boldsymbol{X})\in H'_r\}} \left[\bigvee_{p(\boldsymbol{Y})\in B_r^+\cup B_r^-} \mathrm{uf}_{\mathrm{rch}(\boldsymbol{Y},\boldsymbol{X})}(\boldsymbol{D}_{\langle \mathrm{rch}(\boldsymbol{Y},\boldsymbol{X})\rangle})\right] \quad \text{for every } h(\boldsymbol{X})\in \mathrm{heads}(\Pi), \boldsymbol{D}\in \mathrm{dom}(\boldsymbol{X}) \tag{3.17}$$

Figure 3.3: Using variable independence for an improved body-decoupled grounding (BDG) procedure BDG′. BDG′ is defined for a non-ground tight program Π (BDG′(Π)). However, worst case grounding size is still in $\mathcal{O}(|\Pi|\cdot|\mathrm{dom}(\Pi)|^{2\cdot a})$. Adapted from [12].

3.2.2 Variable justifying Independence

Variable justifying Independence is a technique for improving the grounding size of the foundedness part of BDG [12]. The essential idea is to minimize the number of variables of the head that need to be instantiated, by exploiting the structure of the rule. In the best case this can lead to a reduction in grounding size.

In more detail, given a rule $r\in\Pi$ they construct a variable graph[1] $\mathcal{D}'(r)$ that takes (only) into account body variable connections (and no head variable connections). The procedure works by checking whether body variables can reach head variables. If no, they can exploit this knowledge, by disregarding the unreachable head variables.

Let $\mathcal{W}=var(\Pi)$ be the set of variables, $r\in\Pi$ a rule, $\boldsymbol{X}=var(H_r)$, and $\boldsymbol{Y}=var(B_r)$. Then $\mathrm{rch}(\boldsymbol{Y},\boldsymbol{X})$ is the function $\mathrm{rch}(\boldsymbol{Y},\boldsymbol{X}):\mathcal{W}^2\to\mathcal{W}$, that returns those variables $\boldsymbol{X}$, reachable by $\boldsymbol{Y}$. More formally, let $E(\mathcal{D}'(r))$ denote the set of edges of $\mathcal{D}'(r)$, then:

$$\mathrm{rch}(\boldsymbol{Y},\boldsymbol{X}) \coloneqq \{x \mid x\in\boldsymbol{X}; \exists y\in\boldsymbol{Y}: xRy=((y,v_1),\ldots,(v_n,x)); \{(y,v_1);\ldots;(v_n,x)\}\subseteq E(\mathcal{D}'(r))\}$$

Note that $\mathrm{rch}(\boldsymbol{Y},\boldsymbol{X})\subseteq\boldsymbol{X}$. With this defined we are ready to present the updated reduction in Figure 3.3. Equations (3.15)–(3.16) encode the semantics of foundedness w.r.t. the set of reachable head variables. Lastly, Equation (3.17) is adapted to account for all occurrences of $\mathrm{uf}_{\mathrm{rch}(\boldsymbol{Y},\boldsymbol{X})}(\boldsymbol{D}_{\langle \mathrm{rch}(\boldsymbol{Y},\boldsymbol{X})\rangle})$.

Example 3.3 (Variable Independence)**.** *Take the following rule:*

```
1 h(X1,Y1) :- a(X1,X2), a(X1,X3), a(X2,X3),
2     b(Y1,Y2), b(Y1,Y3), b(Y2,Y3),
3     c(Z1,Z2), c(Z1,Z3), c(Z2,Z3).
```

Intuitively, using variable justifying independence, we can ground the literals $\{a(X1,X2),a(X1,X3),a(X2,X3)\}$ *without taking* $Y1$ *into account. The converse holds for*

[1] Observe the difference to Definition 2.13.

the $\{b(Y1,Y2), b(Y1,Y3), b(Y2,Y3)\}$ *literals. Further, the* $\{c(Z1,Z2), c(Z1,Z3), c(Z2,Z3)\}$ *literals can safely ignore both* $X1$ *and* $Y1$*. Observe that rch*$(\{X1,X2,X3\},\{X1,Y1\}) = \{X1\}$*, rch*$(\{Y1,Y2,Y3\},\{X1,Y1\}) = \{Y1\}$*, and rch*$(\{Z1,Z2,Z3\},\{X1,Y1\}) = \emptyset$*. Therefore, the foundedness part of BDG when using BDG′, when grounding the literals* a *and* b *yields a grounding size of* $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^3)$*, while the grounding size of literals* c *is* $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^2)$.

3.2.3 Aggregate Rewritings for BDG

Aggregate rewriting techniques were introduced as one part[2] of the publication [10]. These rewritings split the aggregates into their parts and keep the resulting maximum rule-arity low. Thereby, they enable efficient grounding of aggregates with BDG. In the following, we provide the intuition behind the two most promising rewriting techniques: $\mathcal{RM}^{\text{count}}$ and $\mathcal{RA}$. Further, we show two examples which shall highlight the basic ideas.

$\mathcal{RM}^{\text{count}}$ dismantles the aggregate into its parts. Take for example an aggregate *count*$\{X : f(X,Y)\} \geq 3$. The idea is to explicitly encode that at least 3 different X occur. This can be done by enumerating the variables accordingly and introducing appropriate constraints. We show two listings, where the first listing depicts the input rule, whereas the second one shows the rewritten rule.

```
1 a(Z) :- b(Z), #count{X:f(X,Y)} >= 3.
```

We rewrite the rule from the listing above with the $\mathcal{RM}^{\text{count}}$ technique and obtain the following rewritten rule.

```
1 a(Z) :- b(Z), f(X1,Y1), f(X2,Y2), f(X3,Y3), X1 != X2, X1 != X3, X2 != X3.
```

$\mathcal{RM}^{\text{count}}$ is applicable for monotonic aggregates occurring in the positive body of a rule that contain a fixed integer bound. These restrictions are dropped by extending the $\mathcal{RM}^{\text{count}}$ technique to the $\mathcal{RS}$ technique, as detailed in [10].

The $\mathcal{RA}$ technique decomposes the aggregate into rules. These rules can then be grounded with BDG. We directly show an example in the next listing as our input:

```
1 a(Z) :- b(Z), #count{X1: f(X1,X2), f(X1,X3), f(X2,X3)} >= 3.
```

In more detail $\mathcal{RA}$ introduces for each aggregate element[3] $(X1 : f(X1,X2), f(X1,X3), f(X2,X3))$ one rule, which is subsequently grounded by BDG. The rewritten program is shown in the listing below, where the part grounded by BDG is indicated by the #program rules. line.

```
1 a(Z) :- b(X), #count{X1 : r(X1)} >= 3.
2 #program rules.
3 r(X1) :- f(X1,X2), f(X1,X3), f(X2,X3).
```

[2] Our IJCAI24 publication [10] is split into two parts: One part about hybrid grounding (which is part of this thesis) and one part about aggregate rewriting techniques (which was part of a Bachelor's Thesis [8]).

[3] An aggregate element has the form $\boldsymbol{T} : B_r^+, B_r^-$.

3.2.4 Observations

In [12] they prove correctness of the reduction, further show its grounding size, and go on to discuss partial applicability. Their partial applicability theorem states that a program Π can be split into two parts, Π_1 and Π_2, where one part is grounded by BDG and the other part by standard-techniques. However, their split has two major shortcomings. One is that shared heads are not allowed and the other one that shared cycles are forbidden. We discuss this in Section 3.3.1 and present *hybrid grounding* in Chapter 4, which overcomes this deficiency.

Further, let $|\Pi|$ be the size of a program, $|\,\mathrm{dom}(\Pi)|$ be the size of the domain, and a be the maximum arity. Then the grounding size of BDG is in general exponential in the arity: $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^a)$. In more detail it is a for constraints, $2 \cdot a$ for tight rules, and $3 \cdot a$ for non-tight (cyclic) normal rules. The $2 \cdot a$ and $3 \cdot a$ prevent BDG from being used for normal and cyclic programs in practice. We discuss this deficiency in Section 3.3.4.

3.3 Shortcomings of Body-decoupled Grounding

In the following, we will detail current shortcomings of BDG. In Section 3.3.1 we talk in detail about limited interoperability, in Section 3.3.2 we discuss current syntax limitations, in Section 3.3.3 we discuss what it takes to automate hybrid grounding (splitting), in Section 3.3.4 performance problems with normal and cyclic BDG are discussed, in Section 3.3.5 arity splitting problems are discussed, in Section 3.3.6 we talk about domain grounding, and in Section 3.3.7 we discuss commonly brought up objections.

3.3.1 Limited Interoperability

Arguably, the most pressing shortcoming of BDG is its limited interoperability, resulting from Theorem 3 in [12].:

Theorem 3.1. *Partial applicability (Adapted[4] from Theorem 3 in [12]): Given a non-ground normal program Π, a partition Π_1, Π_2 with hpred$(\Pi_1) \cap$ hpred$(\Pi_2) = \emptyset$ s.t. $\forall S \in SCC(\Pi)$: either $S \cap$ hpred$(\Pi_1) \neq \emptyset$, or $S \cap$ hpred$(\Pi_2) \neq \emptyset$ (no shared cycles). The answer sets of BDG$(\Pi_1) \cup \mathcal{G}(\Pi_2)$ restricted to $\mathrm{at}(\mathcal{G}(\Pi))$ match those of $\mathcal{G}(\Pi)$.*

Note the two important restrictions in Theorem 3: (i) hpred$(\Pi_1) \cap$ hpred$(\Pi_2) = \emptyset$ and (ii) no shared cycles. Requirement (i) forces us to have distinct heads. We illustrate this with an example:

```
1 c(X1) :- g(X1).
2 c(X1) :- f(X1,X2), f(X1,X3), f(X1,X4), f(X2,X3), f(X2,X4), f(X3,X4).
```

Ideally, we would want to ground Line (1) with SOTA grounders and Line (2) with BDG. This comes as Line (2) has four (densely interacting) variables. However, we cannot do this as requirement (i) prevents shared heads between BDG and SOTA techniques.

Requirement (ii) prevents us from having shared cycles. Take the example in the listing below:

[4] Changed notation for shared cycles and reduction to notation used in the thesis.

```
1 f(X1,X2) :- g(X1,X2).
2 g(X1,X2) :- f(X1,X2), f(X1,X3), f(X1,X4), f(X2,X3), f(X2,X4), f(X3,X4).
```

Again, we would want to ground Line (1) with SOTA techniques and Line (2) with BDG. Notice however the cycle of $\{f, g\}$ in the positive dependency graph $\mathcal{D}_P$. Therefore, requirement (ii) prohibits this.

To address this problem, we introduced *hybrid grounding* [10], see also Section 4.1. Hybrid grounding enables the free splitting of a program Π, into a part Π_1 and a part Π_2, where Π_1 is grounded by BDG, and Π_2 is grounded by SOTA techniques.

3.3.2 Limited accepted Syntax

Theoretically, the original BDG formulation accepted the standard ASP syntax, so a normal (HCF) program Π where each rule is composed of:

$$p_1(\boldsymbol{X}_1) \vee \ldots \vee p_l(\boldsymbol{X}_l) \leftarrow p_{l+1}(\boldsymbol{X}_{l+1}), \ldots, p_m(\boldsymbol{X}_m), \neg p_{m+1}(\boldsymbol{X}_{m+1}), \ldots, \neg p_n(\boldsymbol{X}_n)$$

Arguably, this included the whole ASP-Core-2.0 standard, although it was left open how special literals like *aggregates*, or *comparison* operators are treated.

On the practical side, the tool `newground`, the original implementation of BDG, supported predicates and comparisons. Comparisons were handled at grounding time with a pre-defined semantics. This comes, as at grounding time the truth value of $1 \neq 1$, or $1 < 3$ is already known. However, there were several limitations, such as not supporting arithmetics $(Y + 1)$, or variable assignments $(X = Y + 1)$.

In [10] special techniques for handling aggregates were suggested, which were based on [8]. Further, the next iteration of the prototype `NaGG` featured support for aggregates, partial support for arithmetics and variable assignments.

Still, many questions were left open. First and foremost, the effective integration of arithmetics is crucial for practical purposes. Further, variable assignments in comparisons or aggregates are currently only partially supported and not efficient. These two problems are closely linked to the domain problem (Section 3.3.6). Further work must also be put into an efficient integration of choice rules (head-aggregates), weak constraints, and recursive aggregates.

3.3.3 When should BDG be used?

The introduction of hybrid grounding in [10] (Section 4.1) raises the question which part of a program to ground using BDG and which part to ground with traditional means. This question was already raised in [12] and in [10].

In Chapter 5 we develop a heuristics (Algorithm 5.3) that marks program parts to be grounded by BDG, and parts to be grounded by traditional means. This algorithm analyzes the rule structure, the instance data, and builds upon the results from hybrid grounding. We thereby also introduce the notion of a dense rule, which is intuitively a rule with many variables, which are closely linked together. We illustrate the intuition of a dense rule in the listing below.

```
1 :- f(X1,X2), f(X2,X3), f(X3,X4).
2 :- f(X1,X2), f(X1,X3), f(X1,X4), f(X2,X3), f(X2,X4), f(X3,X4).
```

While the rule in Line (1) has more variables (4) than the maximum arity (2), the variables are not closely linked together (*densely interacting*, see Chapter 5). In contrast, Line (2) has both more variables than the maximum arity and has densely interacting variables. Therefore, it is marked for grounding with BDG (if the data (instance) permits this, see Section 3.3.6).

We implemented Algorithm 5.3 in our prototype `newground3`, where we conducted benchmarks on both solving and grounding-heavy scenarios. Thereby, we demonstrate its practical usefulness.

3.3.4 Bad Performance for Normal and for Cyclic Programs

BDG and hybrid grounding state that the grounding time and size is (only) exponentially dependent on the maximum arity a: $\mathcal{O}\left(|\Pi| \cdot |\text{dom}(\Pi)|^a\right)$. However, this only holds for constraints. For (tight) normal fragments this increases to $2 \cdot a$ and for normal ones to $3 \cdot a$.

We show the significance of this increase in the following example. Firstly, consider a constraint with four densely interacting variables. Grounding it with BDG yields a grounding size of $\approx |\text{dom}(\Pi)|^2$, as the arity $a = 2$. Observe that SOTA techniques have a grounding size of $\approx |\text{dom}(\Pi)|^4$.

```
1 :- f(X1,X2), f(X1,X3), f(X1,X4), f(X2,X3), f(X2,X4), f(X3,X4).
```

In the listing below we modify this rule to be (tight) normal. Grounding this rules with BDG increases the grounding size to $\approx |\text{dom}(\Pi)|^4$, due to the additional effort used for checking *foundedness*. Observe that the grounding size of BDG equals the grounding size of SOTA grounding. Therefore, we argue that using BDG is not beneficial in this case, as BDG pushes effort to the solver, which likely decreases the solving performance.

```
1 g(X1,X2) :- f(X1,X2), f(X1,X3), f(X1,X4), f(X2,X3), f(X2,X4), f(X3,X4).
```

Moving to non-tight normal programs, grounding line 2 with BDG yields a grounding size of $\approx |\text{dom}(\Pi)|^6$, while state-of-the-art grounding remains at $\approx |\text{dom}(\Pi)|^4$. Therefore, using BDG is clearly not beneficial.

```
1 f(X1,X2) :- g(X1,X2).
2 g(X1,X2) :- f(X1,X2), f(X1,X3), f(X1,X4), f(X2,X3), f(X2,X4), f(X3,X4).
```

A considerable part of this thesis (two chapters) is entirely devoted to solving these problems. In Chapter 6 we introduce a novel foundedness encoding (*FastFound*) that reduces normal grounding size from $2 \cdot a$ to $1 + a$. Further, in Chapter 7 we introduce *Lazy-BDG*. Lazy-BDG skips the grounding phase for checking cyclic derivation entirely and ensures non-cyclic derivation (foundedness) via an adapted unfound-set approach. This is also in line with possible applications of BDG in lazy-grounding, as discussed in [10].

3.3.5 The Arity Problem

When looking at a single rule, BDG reduces the grounding size from $\approx |\text{dom}|^{\mathcal{V}}$ ($\mathcal{V}$ number of variables), to $\approx |\text{dom}|^{a}$ (a arity). This eases the grounding bottleneck for a considerable number of instances. However, there are still some cases where the grounding bottleneck prevails. One of the causes thereof is the arity problem discussed here.

To illustrate the arity problem note the rule in the listing below (*Grounding Explosion Problem* taken from [94]).

```
1 b(X1,X2,X3,X4,X5,X6) :- a(X1),a(X2),a(X3),a(X4),a(X5),a(X6).
```

Observe that predicate b has arity 6. If we use BDG for this rule r, BDG has the same asymptotic grounding size, as if we would ground r with SOTA techniques. This comes, as each body variable is contained in the head and therefore we do not need to guess it. In these cases, the grounding size of non-groung normal ASP grounded by BDG reduces from $\approx |\text{dom}|^{2 \cdot a}$ to $\approx |\text{dom}|^{a}$.

Splitting b into smaller parts results in a significantly reduced grounding size. However, it is unknown how this can be done. Therefore, we state it as an open problem to identify predicates, or rules, where we are able to split a predicate into one (or several) with smaller arity[5].

3.3.6 Domain Dependent Grounding

The nature of the bottom-up/semi-naive grounding algorithm (see Section 2.4) in SOTA techniques, enables the incorporation of some domain knowledge. However, this is only partially possible in BDG. We illustrate this in the following example:

```
1 e(1,2). e(2,3). e(3,4). e(4,5). e(5,6), ..., e(n-1,n).
2 {f(X,Y)} :- e(X,Y).
3 :- f(X1,X2), f(X1,X3), f(X1,X4), f(X2,X3), f(X2,X4), f(X3,X4).
```

Our example is a graph problem with edge facts e (Line (1)), where we guess a subgraph defined by the predicate f (Line (2)). Intuitively, the encoding states that no clique of size four may occur in the subgraph (Line (3)). Further, we assume that our instance defined by the edge facts e is a line, starting at vertex 1, going to vertex n.

Let $\mathcal{F}$ be the facts (the input graph) and $|\mathcal{F}| = n - 1$. Then, grounding above program by SOTA techniques, will result in a grounded program of size $\approx 2 \cdot (n-1) \in \mathcal{O}(n)$, i.e., linear in the input. One side effect of using the bottom-up/semi-naive techniques is that they are able to infer that the rule in Line (3) can never be satisfied. This comes as the *join-operation* over the four variables $\{\text{X1}, \text{X2}, \text{X3}, \text{X4}\}$ will never result in a success, as (i) no clique in the input graph exists, and therefore, (ii) no clique in the subgraph can exist.

On the other hand, BDG's grounding is domain based. Therefore, if we would apply BDG to the rule in Line (3) in the above example, we first get $|\text{dom}(\mathcal{F})| = n$, and therefore a grounding size

[5] SOTA techniques like `idlv` apply partial arity reduction by removing isolated variables [23], however, we aim for a more involved reduction technique.

of $\approx n^2 \in \mathcal{O}(n^2)$. Therefore, the impact of the instance is striking: while for SOTA techniques we get a grounding size that is linear, BDG has a grounding size that is quadratic.

Now let us change $\mathcal{F}$ to be a complete graph, with $|\mathcal{F}| \approx n^2$, and $\text{dom}(\mathcal{F}) = n$. The change in grounding size is significant: While BDG (still) has a quadratic grounding size $\approx n^2 \in \mathcal{O}(n^2)$, SOTA techniques have a grounding size of $\approx n^4 \in \mathcal{O}(n^4)$.

We draw the following conclusions for improving BDG: (i) Any efficient integration of BDG into modern grounders needs to take into account the instance, and (ii) it would be desirable to encode the instance properties directly into BDG. *In this thesis we propose a heuristics that solves (i) (Chapter 5.3), while we leave (ii) as an open problem.*

3.3.7 Shifting effort from the Grounder to the Solver

A frequent[6] objection regarding the BDG approach is: "what we gain in grounding, we lose in solving," i.e., it is unclear that there is an overall benefit by shifting effort from grounding to solving. However, empirical results [10], [12] have shown that BDG is indeed capable of solving more instances. In Sections 5.4.5, 6.4, and 7.5, we will further highlight this observation.

Still, using BDG blindly for entire programs is not beneficial, as programs may contain parts where applying BDG results in worse performance. This is especially prevalent for the ASP competition encodings, whose grounding size (and grounding time) is comparatively small to their solving time (see also Section 5.4.5). Therefore, it is crucial to analyze the program in detail before applying BDG and apply BDG only to those parts that are critical in grounding.

The loss in solving performance stems from shifting effort from the grounder to the solver. This comes, as BDG can in principle be seen as a trade-off between structure and size: applying BDG breaks the rule structure in order to save space at grounding. The loss in structure makes the job for the solver harder and raises the question how this can be prevented. We take a first approach in easing this problem in Chapter 7 by introducing Lazy-BDG, which reinfers structure for non-tight ASP. Still, how structure can be preserved for other parts of the BDG reduction is an open problem.

Apart from the discussion above, which focuses on the *answer set existence problem*, we want to raise possible future vectors of research by asking what effect BDG has on problems that require multiple answer sets, such as brave- or cautious-reasoning, counting problems, or optimization. This stems from the fact that the nogoods inferred by the solver differ between a program grounded by BDG and one grounded by standard techniques.

[6] Raised after our presentations and the following discussions during IJCAI24 (33rd International Joint Conference on Artificial Intelligence on Jeju, South Korea) and TAASP24 (Workshop on Trends and Applications of Answer Set Programming, in Klagenfurt, Austria).

CHAPTER 4

Hybrid Grounding

The advent of alternative grounding procedures, such as body-decoupled grounding [11] (BDG, Section 3), was an essential step towards solving the grounding bottleneck. Take for example the rule r in the following listing, and imagine it being embedded in a program Π:

```
1 :- f(X1,X2), f(X1,X3), f(X2,X3).
```

This rule has three variables $\{\text{X1}, \text{X2}, \text{X3}\}$. In traditional ASP we obtain an answer by grounding (variable instantiations) and solving (obtaining an answer). However, the size of the grounded program is exponential in the number of variables. Therefore, the grounding size of the rule is cubic in the domain size ($\approx |\text{dom}|^3$). For many practical instances a cubic size is not feasible, which leads to non-solvable instances. Using BDG instead of traditional techniques yields a grounding size that is only dependent on the (maximum) arity, and quadratic in the above example ($\approx |\text{dom}|^2$).

Still, BDG has major disadvantages on its own: Its limited interoperability with other state-of-the-art approaches, the missing knowledge where and when to actually use BDG, or its relatively bad performance on normal and cyclic rules.

In this chapter we tackle the problem of the limited interoperability, by introducing *hybrid grounding*. Hybrid grounding enables the free (manual) splitting of a program Π into a part Π_1 grounded by BDG, and a part Π_2 grounded by traditional means.

In more detail, this chapter starts with the intuition behind hybrid grounding (Section 4.1) and continues by stating the details of the hybrid grounding reduction (Section 4.2). Further, we introduce in this chapter novel results about applicability of body-decoupled grounding in non-tight disjunctive programs (Section 4.3). Most parts of this chapter are based on the hybrid grounding part of our publication *Bypassing the ASP Bottleneck: hybrid grounding by Splitting and Rewriting* [10], with the exception of the applicability of BDG for non-tight disjunctive programs. Note that we show hybrid grounding's non-tight version in Chapter 7, Section 7.2.

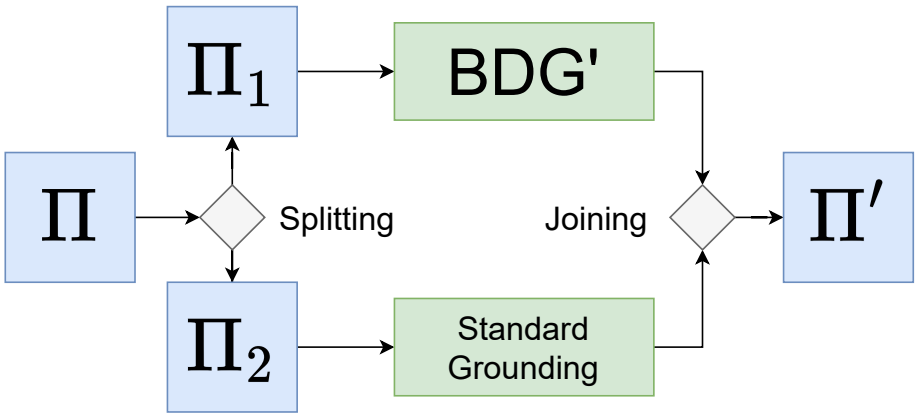


Figure 4.1: Schematics of the workings of hybrid grounding $\mathcal{H}(\Pi_1, \Pi_2)$. A (tight) normal (HCF) program Π is split into Π_1 and Π_2. Π_2 is grounded by standard state-of-the-art (SOTA) grounders, while Π_1 is grounded by an adapted body-decoupled grounding (BDG') procedure. Joining the resulting grounded programs yields Π', the grounded program $\Pi' = \mathcal{H}(\Pi_1, \Pi_2)$.

4.1 Hybrid Grounding for tight ASP

Conceptually, hybrid grounding enables the free (manual) splitting of a program Π into a part grounded by state-of-the-art grounders and a part grounded by an adapted BDG procedure. The need for hybrid grounding originates in BDG's limited interoperability, as discussed in Section 3.3.1. In Figure 4.1 we display the schematics of hybrid grounding.

BDG on its own allows limited partial applicability. Given a program Π, that is split into Π_1 and Π_2, it must be the case that $\text{hpred}(\Pi_1) \cap \text{hpred}(\Pi_2) = \emptyset$. Put otherwise, BDG does not allow for shared heads. This stems from the foundedness check in BDG.

Example 4.1. *The foundedness check in BDG is the root cause for limited interoperability: We illustrate this in the following program snippet:*

```
1 c(X) :- a(X).
2 % BDG GROUNDED:
3 #program rules.
4 c(X1) :- f(X1,X2), f(X1,X3), f(X1,X4), f(X2,X3), f(X2,X4), f(X3,X4).
```

We assume that the rule in Line (1) is grounded by traditional means, whereas the rule in Line (4) is grounded by BDG. Intuitively, the foundedness check of BDG tries to prove foundedness for a particular variable configuration for a given interpretation $\mathcal{I}$. If the foundedness check fails $\mathcal{I}$ cannot be an answer set (unfound).

Let us assume for the sake of explanation that Line (1) can justify $c(2)$, while Line (4) can justify $c(1)$ (but not $c(2)$). Then the foundedness in BDG fails, as although $c(2)$ was derived, it cannot be justified by Line (4).

In more detail, the foundedness check in BDG works by only considering the part grounded by BDG (Line (4)). BDG encodes that a particular rule ($r = 4$, Line (4)) is unable to justify a head atom $c(2)$ with the atom $uf_4(2)$. We assumed that Line (4) cannot derive $c(2)$, therefore $uf_4(2)$ is derived.

The actual check is encoded by a constraint that says, if for all rules r in the BDG part $uf_r(2)$ holds, then the program is unfound. We showcase in the following listing this constraint in our case:

```
1 :- uf_4(2).
```

This comes, as the only rule in the BDG part that can possibly justify $c(2)$ is the rule in Line (4). However, as we fail in its justification, subsequently the constraint fires, leading to an erroneous behavior.

Grounding all rules by BDG would resolve partial applicability. However, this is not advised as using BDG for all parts leads to a loss in performance. Therefore, our approach relies on *splitting*. Abstractly, we partition Π into Π_1 (grounded by BDG) from Π_2, by introducing auxiliary predicates in Π_1. Intuitively, these auxiliary predicates enable inference of the exact source responsible for justifying an atom a in an interpretation $\mathcal{I}$.

4.2 Hybrid Grounding Reduction for tight ASP

We show in Figure 4.2 the details of the tight hybrid grounding reduction. Let Π be a non-ground normal (HCF) program and Π_1 and Π_2 be a partition thereof. Then, $\mathcal{H}(\Pi_1, \Pi_2)$ defines the hybrid grounding procedure, whereas BDG grounds Π_1 and Π_2 is grounded by SOTA techniques. We require Π_1 to be tight, and that $\forall a \in \text{heads}(\Pi_1) : |\text{SCC}(\Pi, a)| = 1$ (no shared cycles in the positive dependency graph), and that a has no self-loop. Conversely, there can be cycles in Π_2.

In the following, we discuss in detail the workings of the hybrid grounding procedure. Rules (4.1)–(4.3) introduce the auxiliary predicates (Rules (4.1)–(4.2)), and add the SOTA-grounded rules (Rule (4.3)). The satisfiability check (Rules (4.4)–(4.10)) works precisely as in BDG (Section 3).

The foundedness check (Rules (4.11)–(4.14)) is adapted to account for the auxiliary predicates. In particular, Rule (4.11) has been adapted to guess non-head variables, only if the respective auxiliary head holds. Further, Rule (4.14) was adapted for the auxiliary predicate h' for the particular rule r. The other rules (Rules (4.12)–(4.13)) remain untouched.

Lemma 1. *Correctness, adapted from [10]: Given a partition of any non-ground HCF program Π into a tight program Π_1 and a program $\Pi_2{=}\Pi \setminus \Pi_1$, s.t. $\forall a \in \mathit{hpred}(\Pi_1) : |\mathit{SCC}(\Pi, a)| = 1$, and a has no self-loop. Then, the answer sets of $\mathcal{H}(\Pi_1, \Pi_2)$ restricted to $at(\mathcal{G}(\Pi))$ match bijectively those of $\mathcal{G}(\Pi)$.*

Glue Π_2 to Π_1 and Ground Π_2

$h'(\mathbf{D}) \vee \overline{h'(\mathbf{D})} \leftarrow$ — for every $h(\mathbf{X}) \in \text{hpred}(\Pi_1)$, $\mathbf{D} \in \text{dom}(\mathbf{X})$ (4.1)

$h(\mathbf{D}) \leftarrow h'(\mathbf{D})$ — for every $h(\mathbf{X}) \in \text{hpred}(\Pi_1)$, $\mathbf{D} \in \text{dom}(\mathbf{X})$ (4.2)

r — for every $r \in \mathcal{G}(\Pi_2)$ (4.3)

Satisfiability of Π_1

$\bigvee_{d \in \text{dom}(\mathbf{x})} \text{sat}_x(d) \leftarrow$ — for every $r \in \Pi_1, x \in \text{var}(r)$, where $\Pi_1 = \{r_1, \ldots, r_n\}$ (4.4)

$\text{sat} \leftarrow \text{sat}_{r_1}, \ldots, \text{sat}_{r_n}$ — for every $r \in \Pi_1, x \in \text{var}(r)$, where $\Pi_1 = \{r_1, \ldots, r_n\}$ (4.5)

$\text{sat}_r \leftarrow \text{sat}_{x_1}(\mathbf{D}_{\langle x_1 \rangle}), \ldots, \text{sat}_{x_t}(\mathbf{D}_{\langle x_t \rangle}), \neg p(\mathbf{D})$ — for every $r \in \Pi_1, p(\mathbf{X}) \in B_r^+$, $\mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle$ (4.6)

$\text{sat}_r \leftarrow \text{sat}_{x_1}(\mathbf{D}_{\langle x_1 \rangle}), \ldots, \text{sat}_{x_t}(\mathbf{D}_{\langle x_t \rangle}), p(\mathbf{D})$ — for every $r \in \Pi_1, p(\mathbf{X}) \in B_r^-$, $\mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle$ (4.7)

$\text{sat}_r \leftarrow \text{sat}_{x_1}(\mathbf{D}_{\langle x_1 \rangle}), \ldots, \text{sat}_{x_t}(\mathbf{D}_{\langle x_t \rangle}), h(\mathbf{D})$ — for every $r \in \Pi_1, h(\mathbf{X}) \in H_r$, $\mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle$ (4.8)

$\text{sat}_x(d) \leftarrow \text{sat}$ — for every $r \in \Pi_1, x \in \text{var}(r)$, $d \in \text{dom}(\mathbf{x})$ (4.9)

$\leftarrow \neg\text{sat}$ (4.10)

Prevent Unfoundedness of Atoms in Π_1

$\bigvee_{d \in \text{dom}(y)} \text{uf}_y(\langle \mathbf{D}, d \rangle) \leftarrow h'(\mathbf{D})$ — for every $r \in \Pi_1, h(\mathbf{X}) \in H_r$, $\mathbf{D} \in \text{dom}(\mathbf{X}), y \in \text{var}(r), y \notin \mathbf{X}$ (4.11)

$\text{uf}_r(\mathbf{D_X}) \leftarrow \text{uf}_{y_1}(\mathbf{D}_{\langle \mathbf{X}, y_1 \rangle}), \ldots, \text{uf}_{y_t}(\mathbf{D}_{\langle \mathbf{X}, y_t \rangle}), \neg p(\mathbf{D_Y})$ — for every $r \in \Pi_1, h(\mathbf{X}) \in H_r$, $p(\mathbf{Y}) \in B_r^+, \mathbf{D} \in \text{dom}(\langle \mathbf{X}, \mathbf{Y} \rangle)$, $\mathbf{Y} = \langle y_1, \ldots, y_t \rangle$ (4.12)

$\text{uf}_r(\mathbf{D_X}) \leftarrow \text{uf}_{y_1}(\mathbf{D}_{\langle \mathbf{X}, y_1 \rangle}), \ldots, \text{uf}_{y_t}(\mathbf{D}_{\langle \mathbf{X}, y_t \rangle}), p(\mathbf{D_Y})$ — for every $r \in \Pi_1, h(\mathbf{X}) \in H_r$, $p(\mathbf{Y}) \in B_r^- \cup (H_r \setminus \{h(\mathbf{X})\})$, $\mathbf{D} \in \text{dom}(\langle \mathbf{X}, \mathbf{Y} \rangle), \mathbf{Y} = \langle y_1, \ldots, y_t \rangle$ $\mathbf{D} \in \text{dom}(\langle \mathbf{X}, \mathbf{Y} \rangle), \mathbf{Y} = \langle y_1, \ldots, y_t \rangle$ (4.13)

$\leftarrow \text{uf}_r(\mathbf{D}), h'(\mathbf{D})$ — for every $r \in \Pi_1, h(\mathbf{X}) \in H_r$, $\mathbf{D} \in \text{dom}(\mathbf{X})$ (4.14)

Figure 4.2: The hybrid grounding procedure $\mathcal{H}(\Pi_1, \Pi_2)$. Given a non-ground normal (HCF) program Π and a partition thereof into Π_1 and Π_2, $\mathcal{H}(\Pi_1, \Pi_2)$ creates a *ground* disjunctive program. We require $\forall a \in \Pi_1 : |\text{SCC}(\Pi, a)| = 1$ (Π_1 is tight, and no shared cycles in positive dependency graph), and that a has no self-loop. We thereby interleave classical grounding on Π_2 with body-decoupled grounding on Π_1.

Observation 4.1. *Runtime, adapted from [10]: Let Π be any non-ground tight program, where predicate arities are bounded by a. Then, $\mathcal{H}(\Pi, \emptyset)$ runs in:*

- $\mathcal{O}((|\Pi|) \cdot |\text{dom}(\Pi)|^a)$*, if the only non-ground rules in Π are constraints.*
- $\mathcal{O}((|\Pi|) \cdot |\text{dom}(\Pi)|^{2 \cdot a})$*, otherwise.*

Example 4.2. *We show the workings of hybrid grounding with a small example, consider for this the program Π_{HG} in the listing below.*

```
1 {f(1);f(2)}. {g(1);g(2)}.
2 c(X) :- f(X).
3 #program rules.
4 c(Y) :- g(Y).
```

We use the hybrid grounding reduction by using standard grounding techniques for Lines (1)–(2), and use BDG for Line (4). So we apply $\mathcal{H}(\Pi_1, \Pi_2)$ where $\Pi_1 = \{c(Y) \leftarrow g(Y)\}$, and $\Pi_2 = \Pi \setminus \Pi_1$. The next listing depicts the code that is grounded by standard techniques (Equation 4.3).

```
1 {f(1);f(2)}. {g(1);g(2)}.
2 c(X):-f(X).
```

The split is performed by the Equations (4.1)–(4.2). They guess the head atoms and the auxiliary head atom c'.

```
1 {c'(1);c'(2)}. c(1):-c'(1). c(2):-c'(2).
```

The satisfiability part (Equations (4.4)–(4.10)) is encoded in the same way as it is done for BDG. We show it in the next listing.

```
1 sat_Y(1)|sat_Y(2).
2 sat_r:-sat_Y(1),c(1). sat_r:-sat_Y(2),c(2).
3 sat_r:-sat_Y(1),not g(1). sat_r:-sat_Y(2),not g(2).
4 sat_Y(1):-sat. sat_Y(2):-sat.
5 sat:-sat_r. :- not sat.
```

Foundedness differs from normal BDG, as the foundedness check is only invoked, when the auxiliary head c' holds. This is in contrast to BDG, where it is always checked as long as c holds. Recall that in Example 4.1 we discussed that this difference leads BDG to not being interoperable with standard grounding techniques. We show the foundedness check of hybrid grounding in the listing below.

```
1 uf_r(1):-not g(1). uf_r(2):-not g(2).
2 :-uf_r(1),c'(1). :-uf_r(2),c'(2).
```

4.3 Partial Application of BDG for Non-Ground Disjunctive Programs

BDG is in general not applicable for non-ground cyclic programs, as the reduction of BDG is defined from non-ground normal (HFC) programs (Σ_2^P complete), to ground disjunctive programs (Σ_2^P complete). However, the complexity class rises from Σ_2^P completeness to Σ_3^P completeness for non-ground disjunctive programs (See Table 4.1).

Despite that, we show in Theorem 4.1 that it is still possible to apply BDG to fragments of a non-ground disjunctive program. We define in the following theorem the conditions for the fragment that are required for applicability. The conditions intuitively say that we can either apply BDG to the fragment if the fragment is tight, or the fragment is strictly normal.

	$\{\}$	$\{\text{not}_s\}$	$\{\text{not}\}$
$\{\}$	co-NP	Δ_2^P	Σ_2^P
$\{\vee_h\}$	Σ_2^P	Σ_2^P	Σ_2^P
$\{\vee\}$	Σ_2^P	Σ_3^P	Σ_3^P

Table 4.1: Complexity classes for the answer set existence problem of non-ground ASP under bounded predicate arities. BDG (Section 3) is applicable for classes in blue (normal and HCF programs), but not for those in red (cyclic disjunctive programs). We show that BDG can be applied for a subprogram of a cyclic disjunctive program, where its SCC of the positive dependency graph is HCF or normal. Table adapted from [40].

Theorem 4.1. *(Partial application of BDG in non-ground disjunctive programs) Let* $SCC(\Pi)$, $rules(S)$, *and* $\text{at}(r)$ *be defined as usual. Further, let* Π_1, Π_2 *be a partition of a non-ground disjunctive program* Π*, s.t. for every* $p \in hpred(\Pi_1)$*, one of the following holds:*

1. $|SCC(\Pi, p)| = 1$ *and* p *has no self-loop; So* p *occurs in a tight fragment.*
2. $|SCC(\Pi, p)| > 1$ *and* $S = SCC(\Pi, p), \forall q \in S : \forall r \in rules(q)\ |H_r| = 1$.

Then the answer sets of $\mathcal{H}(\Pi_1, \Pi_2)$ *restricted to* $\text{at}(\mathcal{G}(\Pi))$ *bijectively match those of* $\mathcal{G}(\Pi)$.

Proof. (Idea) We need to prove two things: (i) given an answer set $\mathcal{I} \in \mathcal{AS}(\mathcal{G}(\Pi))$, then $\exists \mathcal{I}_\mathcal{H} \in \mathcal{AS}(\mathcal{H}(\Pi_1, \Pi_2))$ s.t. $\mathcal{I}_\mathcal{H} \cap \text{at}(\mathcal{G}(\Pi)) = \mathcal{I}$, and (ii) given an answer set $\mathcal{I}_\mathcal{H} \in \mathcal{AS}(\mathcal{H}(\Pi_1, \Pi_2))$, then $\exists \mathcal{I} \in \mathcal{AS}(\mathcal{G}(\Pi))$ s.t. $\mathcal{I}_\mathcal{H} \cap \text{at}(\mathcal{G}(\Pi)) = \mathcal{I}$. The proofs for both sides amount to first checking whether the answer set candidates are models and then verify that they are answer sets.

To check (i), we must first extend $\mathcal{I}$ to $\mathcal{I}_\mathcal{H}$. Then we need to check whether $\mathcal{I}_\mathcal{H}$ satisfies all rules and whether $\mathcal{I}_\mathcal{H}$ is indeed the minimal model (w.r.t. $\subseteq$) of $\mathcal{H}(\Pi_1, \Pi_2)^{\mathcal{I}_\mathcal{H}}$. For checking satisfiability we can restrict ourselves to the newly introduced rules by $\mathcal{H}(\Pi_1, \Pi_2)$.

For the other direction we first set $\mathcal{I} = \text{at}(\mathcal{G}) \cap \mathcal{I}_\mathcal{H}$ (remind yourself that our assumption is that $\mathcal{I}_\mathcal{H}$ is an answer set of $\mathcal{H}(\Pi_1, \Pi_2)$). Then we need to check whether $\mathcal{I}$ satisfies $\mathcal{G}(\Pi)$ and whether $\mathcal{I}$ is the minimal model of $\mathcal{G}(\Pi)^\mathcal{I}$. □

Example 4.3. *We illustrate the application of Theorem 4.1 on the following example. The example illustrates an application where there is a saturation check (Lines (1)–(3)) and is later followed by a BDG call (Line (7)). The rule in Line (7) could also be a disjunctive (HCF) rule. Observe however that it is not possible to use BDG for a rule in Lines (2)–(3), as there is a cycle* $(\{a, disj\})$ *and* a *occurs disjunctively in Line (1).*

```
1 a(1)|a(2). {z(1);z(2)}.
2 disj :- a(X), not z(X).
3 a(1) :- disj. a(2) :- disj.
4 e(1,1). e(1,2). e(1,3).
5 {f(X,Y)} :- e(X,Y), disj.
6 #program rules.
7 d(X1) :- f(X1,X2), f(X1,X3), f(X2,X3).
```

CHAPTER 5

Automated Hybrid Grounding

An automated decision procedure for deciding whether to use body-decoupled grounding or bottom-up grounding is highly desirable. This comes, as the results of Hybrid Grounding (Chapter 4) enable the free (manual) partition of a normal (HCF) program Π into a part Π_1 that is grounded via body-decoupled grounding (BDG) and a part Π_2 that is grounded by means of state-of-the-art techniques.

This chapter derives the *Data Heuristics*, which takes both structure and data (instance) into account. The structure is intuitively the *denseness* of a rule in terms of its treewidth in the variable graph. We approximate the number of instantiated rules, to incorporate the instance into our data heuristics. Our experiments show that using the *Data Heuristics* our prototype `newground3` is able to obtain state-of-the-art performance on solving-heavy tasks, while surpassing them on grounding-heavy tasks.

We start by reviewing the state-of-the-art grounders and their strengths and weaknesses (Section 5.1), which is followed by the introduction of our heuristics (Section 5.2). This is continued by showing our novel prototype `newground3`, which is an effective integration of the data heuristics into a bottom-up grounding procedure (Section 5.3). We close by verifying our claims with our experiments (Section 5.4).

5.1 State-of-the-art Grounding

We shortly summarize the main results of state-of-the-art grounders, necessary for deriving our heuristics.

5.1.1 SOTA-Grounding (Bottom-up/Semi-naive techniques)

`gringo` and `idlv` use (bottom-up) semi-naive database instantiation techniques to ground a program Π [23], [61]. This is performed by analyzing the positive-dependency graph $\mathcal{D}_P$ of

Π. As $\mathcal{D}_P$ may form cycles, they first analyze $\mathcal{D}_P$ for SCCs and create the respective reduced dependency graph $\mathcal{D}_P^R$ (See also Section 2.1). Grounding is performed along a topological order L_p of the $\mathcal{D}_P^R$. The *bottom-up* grounding algorithm is the algorithm defined that grounds along L_p, while additionally keeping track of a candidate set D. This candidate set D intuitively keeps track of the tuples that are *possibly derivable*. Conversely, if a tuple is not in D it is surely false. Rules are instantiated according to the candidate set D. If an SCC contains a cycle, then semi-naive techniques are used to prevent unnecessary derivations [23], [61].

Observe that by following this technique, grounding is in the worst-case exponential in the maximum number of variables of a rule $r \in \Pi$, where $\mathcal{V} = \max_{r \in \Pi} |\operatorname{var}(r)|$: $\mathcal{O}\left(|\Pi| \cdot |\operatorname{dom}(\Pi)|^{\mathcal{V}}\right)$

5.1.2 Bottom-up grounding evaluates 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 $a \notin D_{pot} \cup D_T$ we know that we can never derive a. This idea leads to the following improvements when instantiating a rule $r \in \Pi$: (i) If $a \in D_T$ and $a \in B_r^+$, then remove a from B_r^+. (ii) When $a \notin D_{pot} \cup D_T$, $a \in B_r^+$, then do not instantiate r. (iii) If $a \in D_T$ and $a \in H_r$, then do not instantiate r. (iv) If $a \in D_T$ and $a \in B_r^-$, then do not instantiate r. (v) If $a \notin D_{pot} \cup D_T$, then remove a from B_r^-. (vi) Gather facts, whenever a rule body is empty.

Together, these improvements enable the evaluation of stratified programs. Remind yourself that stratified programs do not have negative cycles, i.e., cycles in their dependency graph which contain at least one negative edge. However, these improvements have in the worst case no effect on the grounding size of non-stratified programs, and grounding still remains exponential in the maximum number of variables.

5.1.3 Treewidth aware rewritings

Utilizing the rule structure to rewrite non-ground rules is performed by `Lpopt`. It computes a minimum size tree decomposition, which is then used to introduce fresh rules with a preferably smaller grounding size.

In more detail, for every rule $r \in \Pi$ it first creates the variable graph $\mathcal{D}(r)$ (See also Section 2.13). After computing a minimum size tree decomposition (Section 2.14), it introduces fresh predicates and fresh rules for every bag of the tree decomposition. The arity of the fresh predicates correspond to the respective bag size, as does the number of variables per rule.

Let $TW(\mathcal{D}(r))$ be the treewidth, then $\mathcal{B} = TW(\mathcal{D}(r)) + 1$ is its bag size. It was shown that `Lpopt` produces a rewriting that is exponential in $\mathcal{B}$, where $\mathcal{B} \leq \mathcal{V}$ ($\mathcal{V} = \max_{r \in \Pi} |var(r)|$): $\mathcal{O}(|\Pi| \cdot |\operatorname{dom}(\Pi)|^{\mathcal{B}})$.

More details can be found in [14], [107]. Internally `idlv` uses the concepts of `Lpopt` to reduce the grounding size [24].

Algorithm 5.1: $Heur_{\text{struct}}(r, marker)$; Variable Heuristics

Data: Rule r, ground instructions *marker*

if *IsStratified(r)* **then**
  marker $\leftarrow$ marker $\cup$ (r, SOTA) ;
else if $|\operatorname{var}(r)| > a \wedge$ *IsConstraint(r)* $\wedge \mathcal{AP}(r)$ **then**
  marker $\leftarrow$ marker $\cup$ (r, BDG) ;
else if $|\operatorname{var}(r)| > 2 \cdot a \wedge$ *IsTight(r)* $\wedge \mathcal{AP}(r)$ **then**
  marker $\leftarrow$ marker $\cup$ (r, BDG) ;
else if $|\operatorname{var}(r)| > 3 \cdot a \wedge \mathcal{AP}(r)$ **then**
  marker $\leftarrow$ marker $\cup$ (r, BDG) ;
else
  marker $\leftarrow$ marker $\cup$ (r, SOTA) ;
end

5.1.4 Body-decoupled Grounding

Body-decoupled grounding produces grounding sizes that are exponential in the maximum arity. We discussed the details in Chapter 3 and restrict ourselves here to state the main results w.r.t. the grounding size. Let a be the maximum arity ($a = \max_{r \in \Pi} \max_{p(\boldsymbol{X}) \in H_r \cup B_r} |\boldsymbol{X}|$), then BDG has a grounding size in $\mathcal{O}(|\Pi| \cdot |\operatorname{dom}(\Pi)|^c)$, where $c = a$ for constraints, $c = 2 \cdot a$ for (tight) normal (HCF) programs, and $c = 3 \cdot a$ for (non-tight) normal programs. Note that the other improvements we discussed, such as Variable Justifying Independence (Section 3.2.2), do not lead to an improvement in the worst-case grounding size of BDG.

5.2 Automated Splitting Heuristics

Automated Splitting is enabled by the development of heuristics that take into account both the structure of a rule and the data of the instance. We start by introducing the variable heuristics (Algorithm 5.1), followed by the structural heuristics (Algorithm 5.2), which is then extended to the data-structural heuristics (Algorithm 5.3).

5.2.1 Variable Heuristics

We incorporate the results from bottom-up grounding and BDG discussed so far in the variable heuristics in Algorithm 5.1. Therefore, stratified parts are grounded with bottom-up procedures, whereas BDG is used for non-stratified parts whenever the (adjusted) arity is strictly smaller than the number of variables. Let a be the maximum arity of a rule, IsConstraint(r) be true iff r is a constraint, and IsTight(r) be true iff r is tight. With $\mathcal{AP}(r)$ we denote that BDG can handle the rule r in principle, so that it is in a normal (HCF) component and its literals are syntactically supported. The *marker* set denotes the result and saves which rule should be grounded with which technique. It is initially an empty set that is then filled with tuples of (r, type), where r is a rule and type is type $\in \{\text{BDG}, \text{SOTA}\}$.

In more detail, stratified rules are always grounded with SOTA-grounders (Lines (1)–(2)). If r is a constraint and its maximum arity is strictly smaller than the number of variables in the rule, then it is grounded with BDG (Lines (3)–(4)). The same holds when r is (tight) normal and the maximum arity times 2 is strictly smaller than the number of variables (Lines (5)–(6)), or when it is cyclic and the maximum arity times 3 is strictly smaller than the number of variables (Lines (7)–(8)). If none of these cases hold, it is grounded by SOTA-techniques (Lines (9)–(10)). Note that the strictly smaller condition is necessary, as BDG performs worse than standard grounders when their asymptotic grounding sizes match.

Theorem 5.1. *Let Π be a non-ground normal (HCF) program grounded with the markings produced by Algorithm 5.1, where $\mathcal{AP}(r)$ holds for every $r \in \Pi$. Then its grounding size is in $\mathcal{O}\left(|\Pi| \cdot |\operatorname{dom}|^{3 \cdot a}\right)$.*

Proof. For any non-ground normal (HCF) program Π we apply Algorithm 5.1 rule by rule. If a rule is in a stratified part, then it is grounded by bottom-up grounding and has a grounding size that is in $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{a}\right)$.

The remaining cases of the algorithm (Lines (3)–(11)) decide whether to use BDG. Note that by assumption $\mathcal{AP}(r)$ holds for all rules $r \in \Pi$. We now perform a case distinction. Whenever r is a constraint and $|\operatorname{var}(r)| > a$ then we use BDG for constraints and obtain a grounding size of $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{a}\right)$. Conversely, if r is a constraint and $|\operatorname{var}(r)| \leq a$, then it is also exponential in the arity. For r being normal, tight, and $|\operatorname{var}(r)| > 2 \cdot a$, we ground with BDG and obtain a grounding size of $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{2 \cdot a}\right)$. Conversely when $|\operatorname{var}(r)| \leq 2 \cdot a$, the grounding size is still bounded by $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{2 \cdot a}\right)$ when grounded by bottom-up grounding. We are left with (non-tight) normal rules, which are grounded with BDG whenever $|\operatorname{var}(r)| > 3 \cdot a$, where we obtain a grounding size of $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{3 \cdot a}\right)$ with BDG. The converse gives us a grounding size that is bounded by the same asymptotic time.

As $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{a}\right) \in \mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{2 \cdot a}\right) \in \mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{3 \cdot a}\right)$, we obtain an overall bound per rule of $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{3 \cdot a}\right)$. For the program Π we follow $\mathcal{O}\left(|\Pi| \cdot |\operatorname{dom}|^{3 \cdot a}\right)$. □

Example 5.1. *We show several rules and their classification according to Algorithm 5.1. Let Π be a program s.t. $\Pi = \Pi_1 \cup \Pi_2$ is a partition. Π_1 is shown in the next listing.*

```
1 e(1,2). e(1,3). e(1,4). e(1,5). % ... additional input
2 c(X,Y) :- e(X,Y).
3 {f(X,Y)} :- e(X,Y).
4 {g(X,Y)} :- e(X,Y).
```

The next rules shown are part of Π_2, i.e., $r \in \Pi_2$. The rule shown in the next listing depicts a rule that is stratified and therefore grounded by bottom-up grounders.

```
1 d(X1) :- c(X1,X2), c(X1,X3), c(X2,X3).
```

The next rule is in the non-stratified part, has 4 variables, and a maximum arity of 2. It is therefore grounded by BDG.

```
1 :- f(X1,X2), f(X2,X3), f(X3,X4).
```

Our rule shown below is similar to the rule above. It has 3 *variables and a maximum arity of* 2*, and thereby grounded by BDG.*

```
1 :- f(X1,X2), f(X1,X3), f(X2,X3).
```

The next rule is a normal rule with 5 *variables. It has a maximum arity of* 2*. So we ground it with BDG.*

```
1 c(X1) :- f(X1,X2), f(X1,X3), f(X1,X4), f(X1,X5), f(X2,X3), f(X2,X4), f(X2,X5)
      , f(X3,X4), f(X3,X5), f(X4,X5).
```

We finally show a cyclic rule with 7 *variables. It has a maximum arity of* 2 *that is grounded with BDG.*

```
1 d(X1) :- g(X1,X2), g(X1,X3), g(X1,X4), g(X1,X5), g(X1,X6), g(X1,X7), g(X2,X3)
      , g(X2,X4), g(X2,X5), g(X2,X6), g(X2,X7), g(X3,X4), g(X3,X5), g(X3,X6), g
      (X3,X7), g(X4,X5), g(X4,X6), g(X4,X7), g(X5,X6), g(X5,X7), g(X6,X7).
2 g(X1,X2) :- d(X1), d(X2).
```

5.2.2 Structural Heuristics

We introduce the *Structural Heuristics* (Algorithm 5.2) by extending the variable heuristics (Algorithm 5.1) with the support to rewrite rules with `Lpopt`. The heuristics is structural in the sense that it takes into account the variable graph structure of the rule. This is done by performing a tree decomposition and analyzing the treewidth of each rule. Let $TW(\mathcal{D}(r))$ be the treewidth of the variable graph of r and $\mathcal{B}(r) = TW(\mathcal{D}(r)) + 1$ be its bagsize. Note that the bagsize corresponds to the number of variables in the rewritten rule. $\text{flag}_{\texttt{Lpopt}}$ denotes a flag which is true or false, depending on the user's choice whether to use `Lpopt`. The other notions are defined as for the variable heuristics.

In detail the structural heuristics grounds stratified rules with bottom-up grounders (Lines (1)–(2)). `Lpopt` is invoked whenever the bag size of the tree decomposition is smaller than the number of variables (Lines (3)–(7)). First the `Lpopt` procedure is invoked in Line (4) to produce a rewritten rule. Next it recursively calls the structural heuristics, to determine whether the rewritten rules are grounded with BDG or with bottom-up grounding (Lines (5)–(7)). The algorithm continues by taking a case distinction, similar to the variable heuristics, whether to use BDG or bottom-up grounding (Lines (8)–(16)).

Next we derive a result about the number of generated non-ground rules of `Lpopt`. Up to now only the number of generated ground rules was discussed [14], [107].

Lemma 2. *We rewrite a rule* $r \in \Pi$ *with* `Lpopt`*, where* Π *is a program. Let* k *be the treewidth of the variable graph of* r*. Then the resulting set of (non-ground) rules* $\gamma = \texttt{Lpopt}(r)$ *is in* $\gamma \in \mathcal{O}(||r|| \cdot k)$.

Proof. `Lpopt`(r) produces a set γ of rules. This is done by performing a tree decomposition on the variable graph and then creating a new rule for each bag of the tree decomposition. `Lpopt`

Algorithm 5.2: $\mathit{Heur}_{\text{struct}}(r, marker)$; Structural Heuristics

Data: Rule r, ground instructions *marker*

if *IsStratified(r)* **then**
  marker $\leftarrow$ marker $\cup$ (r, SOTA) ;
else if $\mathcal{B}(r) < |\text{var}(r)| \wedge \mathit{flag}_{\texttt{Lpopt}}$ **then**
  $R_l \leftarrow \text{lpopt}(r)$;
  for r_l *in* R_l **do**
    $\mathit{Heur}_{\text{struct}}(r_l, \text{marker})$;
  end
else if $\mathcal{B}(r) > a \wedge$ *IsConstraint(r)* $\wedge \mathcal{AP}(r)$ **then**
  marker $\leftarrow$ marker $\cup$ (r, BDG) ;
else if $\mathcal{B}(r) > 2 \cdot a \wedge$ *IsTight(r)* $\wedge \mathcal{AP}(r)$ **then**
  marker $\leftarrow$ marker $\cup$ (r, BDG) ;
else if $\mathcal{B}(r) > 3 \cdot a \wedge \mathcal{AP}(r)$ **then**
  marker $\leftarrow$ marker $\cup$ (r, BDG) ;
else
  marker $\leftarrow$ marker $\cup$ (r, SOTA) ;
end

introduces fresh literals for each bag (with the exception of the root bag). Additionally, they introduce fresh domain literals for the variables of a rule.

We assume an `Lpopt` implementation where each literal $p(\boldsymbol{X}) \in r$ is introduced exactly once. For negative body literals `Lpopt` needs to ensure variable safety by introducing variable domain literals for the variables of negative body literals, which we denote as $\text{dom}(X_i)$. Note that $||p(\boldsymbol{X})|| = |\boldsymbol{X}| + 1$, and $||\text{dom}(X_i)|| = 2$. So for each literal $||p(\boldsymbol{X})|| + \sum_{X_i \in \boldsymbol{X}} ||\text{dom}(X_i)|| = 3|\boldsymbol{X}| + 1$. Note the inequality $3|\boldsymbol{X}| + 1 \leq 3|\boldsymbol{X}| + 3 = 3 \cdot ||p(\boldsymbol{X})||$. Therefore, by summing over all literals we get $\sum_{p(\boldsymbol{X}) \in r} 3 \cdot ||p(\boldsymbol{X})|| = 3 \cdot ||r|| \in \mathcal{O}(||r||)$.

We continue with the size of the fresh literals. For join nodes of the nice tree decomposition, we generate rules that have 3 freshly introduced literals (which is the maximum). Let $\chi(n)$ and $\chi(n-1)$ be bags s.t. $\chi(n)$ is the parent of $\chi(n-1)$. Then the variables of the freshly introduced literal $p_{\chi(n-1)}(\boldsymbol{Y})$ are defined by $\boldsymbol{Y} = \chi(n) \cap \chi(n-1)$. Observe how $|\boldsymbol{Y}| \leq \mathcal{B}(r) = k + 1$. Further, it is a well known result that nice tree decompositions exist that are linear $(4 \cdot n)$ in the number of vertices (n) of the input graph [16]. We have $|var(r)|$ vertices, so we follow that we have at most $4 \cdot |var(r)|$ many bags and $4 \cdot |var(r)| - 1$ many edges (as the bags form a tree). We follow that the overall size of the freshly introduced literals is $4 \cdot |var(r)| \cdot (3 \cdot (k+1)) \in \mathcal{O}(|var(r)| \cdot k)$.

Lastly, we need to generate $|var(r)|$ many domain literals $\text{dom}(X_i)$. For each such generated rule $\text{dom}(X_i) \leftarrow b_r(\boldsymbol{X})$ we need to find a suitable body literal $b_r(\boldsymbol{X})$ (s.t. $X_i \in \boldsymbol{X}$). It needs to hold $b_r \in B_r^+$ and ideally b_r should be as small as possible. However, in the worst case $|\boldsymbol{X}| = k + 1$. Therefore, the size of every such rule is bounded by $||\text{dom}(X_i)|| + ||b_r(\boldsymbol{X})|| \leq 2 + k + 2 \in \mathcal{O}(k)$.

We need to introduce $|var(r)|$ many, so the size of the domain rules is in $\mathcal{O}(|var(r)| \cdot k)$.

Combining the three parts we obtain $\mathcal{O}(|var(r)| \cdot k) \subseteq \mathcal{O}(||r|| \cdot k)$, as $|var(r)| \leq ||r||$ by definition. □

We continue with our theorem about the grounding size of Algorithm 5.2.

Theorem 5.2. *Let Π be a non-ground normal (HCF) program grounded with the markings produced by Algorithm 5.2, where $\mathcal{AP}(r)$ holds for every $r \in \Pi$, let c be the number of rules of Π, let k be the maximum treewidth of any rule of Π, and let flag$_{\texttt{Lpopt}}$ be true.*

Then its grounding size is in $\mathcal{O}\left((|\Pi| \cdot k) \cdot |\operatorname{dom}|^{3 \cdot a}\right)$.

Proof. For any non-ground normal (HCF) program Π we apply Algorithm 5.2 rule by rule. If a rule is in a stratified part, then it is grounded by bottom-up grounding and has a grounding size that is in $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{a}\right)$. As flag$_{\texttt{Lpopt}}$ is true, we apply `Lpopt`. If $\mathcal{B}(r) < |\operatorname{var}(r)|$ then `Lpopt` is applied. The naive usage of `Lpopt` applied to the rule r yields a set of rules γ, that have max $\mathcal{B}(r)$ variables. Further, `Lpopt` generates $|\gamma| \in \mathcal{O}\left(||r|| \cdot k\right)$ many (non-ground) rules (Lemma 2). Therefore its naive grounding size is in $\mathcal{O}\left((||r|| \cdot k) \cdot |\operatorname{dom}|^{\mathcal{B}(r)}\right)$. We perform a recursive call in Line (6) which bounds it to the maximum arity (as shown next).

Observe that for Lines (8)–(16) the max bag size and the number of variables match ($\mathcal{B}(r) = |\operatorname{var}(r)|$), due to the `Lpopt` procedure. Note that by assumption $\mathcal{AP}(r)$ holds for all rules $r \in \Pi$. These lines decide whether to use BDG, where we now proceed with a case distinction. Whenever r is a constraint and $\mathcal{B}(r) > a$ then we use BDG for constraints and obtain a grounding size of $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{a}\right)$. Conversely, if r is a constraint and $\mathcal{B}(r) \leq a$, then it is exponential in the arity. For r being normal, tight, and $\mathcal{B}(r) > 2 \cdot a$, we ground with BDG and obtain a grounding size of $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{2 \cdot a}\right)$. Conversely when $\mathcal{B}(r) \leq 2 \cdot a$, the grounding size is still bounded by $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{2 \cdot a}\right)$ when grounded by bottom-up grounding. We are left with (non-tight) normal rules, which are grounded with BDG whenever $\mathcal{B}(r) > 3 \cdot a$, where we obtain a grounding size of $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{3 \cdot a}\right)$ with BDG. The converse gives us a grounding size that is bounded by the same asymptotic time.

As $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{a}\right) \in \mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{2 \cdot a}\right) \in \mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{3 \cdot a}\right)$, every rule which was not rewritten by `Lpopt` has a grounding size that is in $\mathcal{O}\left(||r|| \cdot |\operatorname{dom}|^{3 \cdot a}\right)$. For the `Lpopt` rewritten rules we obtain $\mathcal{O}\left(||r|| \cdot k \cdot |\operatorname{dom}|^{3 \cdot a}\right)$. As $\sum_{r \in \Pi} ||r|| \cdot k = k \cdot \sum_{r \in \Pi} ||r|| = k \cdot |\Pi|$, we follow $\mathcal{O}\left(|\Pi| \cdot k \cdot |\operatorname{dom}|^{3 \cdot a}\right)$. □

The difference between the theoretical grounding size of the variable heuristics (Theorem 5.1) and the structural heuristics (Theorem 5.2) is only minor, and results from the treewidth of the variable graph. Keep in mind that the variable heuristics grounds rules with BDG that are rewritten with `Lpopt` in the structural heuristics. Therefore, it is expected that the variable heuristics performs worse on those problems, as BDG pushes effort from the grounder to the solver.

Example 5.2. *We display the changes in used grounders compared to Example 5.1. The only difference occurs for the rule shown in the next listing, which is a rule with a variable graph of treewidth* 1*, bagsize of* 2*, but has* 4 *variables. Therefore,* `Lpopt` *is used to rewrite the rule.*

```
1 :- f(X1,X2), f(X2,X3), f(X3,X4).
```

The `Lpopt` *rewritten rules are depicted in the next listing.*

```
1 % Lpopt Rewritten Rule:
2 temp_decomp_0_0(X3) :- f(X3,X4).
3 temp_decomp_0_1(X2) :- f(X2,X3), temp_decomp_0_0(X3).
4 :- f(X1,X2), temp_decomp_0_1(X2).
```

As for each rewritten rule $|var(r)| < 2 \cdot a$*, or respectively* $|var(r)| < a$ *holds, bottom-up grounding is used to ground these rules.*

5.2.3 Data-Structural Heuristics

The data heuristics (Algorithm 5.3) incorporates instance knowledge in the selection process. This stems from the fact that although $Heur_{\text{struct}}$ has nice theoretical properties, it can fail due to its lack of instance considerations. One of the major contributing factors to these failures is caused by SOTA-techniques using instance knowledge, which BDG cannot use. While BDG uses domain-grounding, SOTA-techniques use candidate set grounding, which propagates facts, or surely false atoms. Further, applying `Lpopt` can lead to an increase in grounding size, as it (partially) destroys the structure of the rule, thereby making variable join operations more difficult[1]. Both shortcomings are particularly prevalent on sparse instances.

To alleviate these issues, first observe that rule instantiations with SOTA-techniques, are similar to joins in a database system. It resembles joins of the predicates of the postive body B_r^+ [92]. Interestingly for us, one can estimate the resulting size of the join [54], which only depend on the tuple size $T(p)$ of a predicate p, and the domain size $|\operatorname{dom} X|$ of a variable X in a rule. This aligns with the data that is needed to compute the size of instantiated BDG rules. Therefore, the idea of the data heuristics is to take the method that has a smaller estimated number of instantiated rules, where we compare the estimate of the join size of SOTA-grounders to the estimate of the number of instantiated rules of BDG.

We estimate the SOTA-grounding size according to the join-selectivity criterion of [92], which is similar to the method proposed in [54]. Further, note that a variant of this criterion is used in `idlv` [24]. We compute the join size in an iterative fashion, for a rule r with a positive body $p_{l+1}, \ldots, p_m$, Let p_{i+1} be the current predicate under investigation, A_i be the positive predicatess up to and including p_i. Therefore, let $T(p_{i+1})$ be the number of tuples of p_{i+1}, and $T(A_i)$ be the estimated join size up to and including predicate p_i. Let $\operatorname{dom}(X, p_{i+1})$ be the domain of variable X for predicate p_{i+1}, and $\operatorname{dom}(X, A_i)$ be the domain of X up to and including p_{i+1}. We update

[1]Therefore, applying `Lpopt` blindly on the whole program is not reasonable. A more reasonable approach is to estimate the grounding size and only use it, if the projected grounding size is smaller. `idlv` does this in a similar way [24], however, `idlv` is closed source and does not feature suitable callbacks to integrate additional heuristics or rewritings. Thus, an integration into `idlv` is not possible.

$\mathrm{dom}(X, A_{i+1})$ as $\mathrm{dom}(X, A_{i+1}) = \min\{\mathrm{dom}(X, A_i), \mathrm{dom}(X, p_{i+1})\}$. Equation (5.1) shows our join size estimation for SOTA-grounding, and with $T_{\bowtie}(r)$ we denote the estimated join size of the rule r.

$$T(A_i \bowtie p_{i+1}) = \frac{T(A_i) \cdot T(p_{i+1})}{\Pi_{X \in \mathrm{var}(A_i) \cap \mathrm{var}(p_{i+1})} \max\{\mathrm{dom}(X, A_i), \mathrm{dom}(X, p_{i+1})\}} \tag{5.1}$$

For hybrid grounding the size estimation is straight-forward according to the definition provided in Rules (4.1)–(4.14). We assume to have pre-computed the variable domains $\mathrm{dom}(X)$ for a rule r, s.t. $X \in \mathrm{var}(r)$. Then Equation (5.2) details the estimation of the head-guess size, for the respective Equations (4.1) and (4.2). Note that there is no need to estimate the impact of Equation (4.3), as we consider a single rule r. Equations (5.3)–(5.5) estimate the size of the satisfiability encoding, where Equation (5.3) estimates the impact of variable guessing and saturation (Respective Equations (4.4) and (4.9)). Equation (5.4) accounts for the constant parts (Respective Equations (4.5) and (4.9)), and Equation (5.5) computes the estimate for the literals (Respective Equations (4.6)–(4.8)). Equations (5.6)–(5.8) estimate the size of the unfound-part. Equation (5.6) estimates the size of the constraint (Respective Equation (4.14)), Equation (5.7) estimates the size of the variable instantiations (Respective Equation (4.11)), and Equation (5.8) estimates the size of the literals (Respective Equations (4.12) and (4.13)). Finally, Equation (5.9) sums up all contributions.

$$T^G_{\mathcal{H}}(r) = 2 \cdot \left(\Sigma_{h(\boldsymbol{X}) \in H_r} \Pi_{X \in \boldsymbol{X}, h(\boldsymbol{X}) \in H_r} |\,\mathrm{dom}(X)|\right) \tag{5.2}$$

$$T^{S1}_{\mathcal{H}}(r) = 2 \cdot \Sigma_{X \in \mathrm{var}(r)} |\,\mathrm{dom}(X)| \tag{5.3}$$

$$T^{S2}_{\mathcal{H}}(r) = 2 \tag{5.4}$$

$$T^{S3}_{\mathcal{H}}(r) = \Sigma_{p(\boldsymbol{X}) \in r} \Pi_{X \in \boldsymbol{X}} |\,\mathrm{dom}(X)| \tag{5.5}$$

$$T^{F1}_{\mathcal{H}}(r) = \Sigma_{h(\boldsymbol{X}) \in H_r} \Pi_{X \in \boldsymbol{X}} |\,\mathrm{dom}(X)| \tag{5.6}$$

$$T^{F2}_{\mathcal{H}}(r) = \Sigma_{h(\boldsymbol{X}) \in H_r} \left(\Sigma_{Y \in \mathrm{var}(r) \setminus \boldsymbol{X}} \left(|\,\mathrm{dom}(Y)| \cdot \Pi_{X \in \boldsymbol{X}} |\,\mathrm{dom}(X)|\right)\right) \tag{5.7}$$

$$T^{F3}_{\mathcal{H}}(r) = \Sigma_{h(\boldsymbol{X}) \in H_r} \left(\Sigma_{p(\boldsymbol{Y}) \in r \setminus h(\boldsymbol{X})} \left(\Pi_{Y \in \boldsymbol{Y}} |\,\mathrm{dom}(Y)| \cdot \Pi_{X \in \boldsymbol{X}} |\,\mathrm{dom}(X)|\right)\right) \tag{5.8}$$

$$T_{\mathcal{H}}(r) = T^G_{\mathcal{H}}(r) + T^{S1}_{\mathcal{H}}(r) + T^{S2}_{\mathcal{H}}(r) + T^{S3}_{\mathcal{H}}(r) + T^{F1}_{\mathcal{H}}(r) + T^{F2}_{\mathcal{H}}(r) + T^{F3}_{\mathcal{H}}(r) \tag{5.9}$$

We update Algorithm 5.2 to account for the estimates of the grounding sizes, which is depicted in Algorithm 5.3. Note that for `Lpopt` (Lines (1) and (4)–(7)) we compute the grounding sizes of the rewritten program as a whole. Therefore, $T(r)$ is the *best* (smaller) estimate of r, $R_l = \texttt{Lpopt}(r)$, and $T_{\bowtie}(R_l)$ is $\Sigma_{r_l \in R_l} T(r_l)$. Furthermore, we incorporate in Lines (7)–(12) that we only ground via BDG if the estimated rule size is actually smaller than the estimated rule size of the join.

Example 5.3. *We show an example of the estimated number of instantiations according to the rule in Line (3) of the following listing. Note that this rule has* 3 *densely interacting variables.*

Algorithm 5.3: *Heur*$(r, marker)$; Data-Structural Heuristics

Data: Rule r, ground instructions *marker*

$R_l \leftarrow \text{lpopt}(r)$;
if *IsStratified(r)* **then**
  marker $\leftarrow$ marker $\cup (r, \text{SOTA})$;
else if $\mathcal{B}(r) < |\text{var}(r)| \wedge$ *flag*$_{\texttt{Lpopt}} \wedge T_{\bowtie}(r) > T_{\bowtie}(R_l)$ **then**
  for r_l *in* R_l **do**
    Heur$_{\text{struct}}(r_l, \text{marker})$;
  end
else if $\mathcal{B}(r) > \max_{a \in r} a \wedge$ *IsConstraint(r)* $\wedge \mathcal{AP}(r) \wedge T_{\mathcal{H}}(r) < T_{\bowtie}(r)$ **then**
  marker $\leftarrow$ marker $\cup (r, \text{BDG})$;
else if $\mathcal{B}(r) > 2 \cdot \max_{a \in r} a \wedge$ *IsTight(r)* $\wedge \mathcal{AP}(r) \wedge T_{\mathcal{H}}(r) < T_{\bowtie}(r)$ **then**
  marker $\leftarrow$ marker $\cup (r, \text{BDG})$;
else if $\mathcal{B}(r) > 3 \cdot \max_{a \in r} a \wedge \mathcal{AP}(r) \wedge T_{\mathcal{H}}(r) < T_{\bowtie}(r)$ **then**
  marker $\leftarrow$ marker $\cup (r, \text{BDG})$;
else
  marker $\leftarrow$ marker $\cup (r, \text{SOTA})$;
end

Therefore its grounding size is cubic in the domain for bottom-up grounding procedures. As its maximum arity is 2*, BDG's grounding size is quadratic.*

```
1 e(1,2). e(1,3). % ... facts, assumed as input to the program.
2 {f(X,Y)} :- e(X,Y).
3 :- f(X1,X2), f(X1,X3), f(X2,X3).
```

The facts resemble a graph defined by the edge facts $e(X, Y)$*. We analyze the behavior on different graph densities (number of edges divided by edges of complete graph) and graph sizes (1 to 100 vertices). The estimation is performed along Equation (5.1) for bottom-up grounding and with Equation (5.9) for BDG. The results are shown in Figure 5.1. Density has an effect on estimation for bottom-up (SOTA) grounders, but not on BDG. High density graphs (like the complete graph with density* 1.0*) show a rapid increase in the number of estimated rules. Sparse graphs (e.g.,* 0.01*) show a very mild increase. BDG has a comparatively large constant, therefore the usage of SOTA-techniques on small graphs is suggested.*

5.3 `newground3`: Prototype Implementation

Our prototype `newground3` is a full-fledged grounder that combines SOTA and BDG grounders. Algorithm 5.3 is used to decide between SOTA and BDG grounding. Furthermore, the algorithm does not pre-impose on the user which SOTA grounder to use, and therefore, offers integration with `gringo` and `idlv`. In this section, we discuss implementation choices, highlight implementation challenges, and present the structure of the prototype.

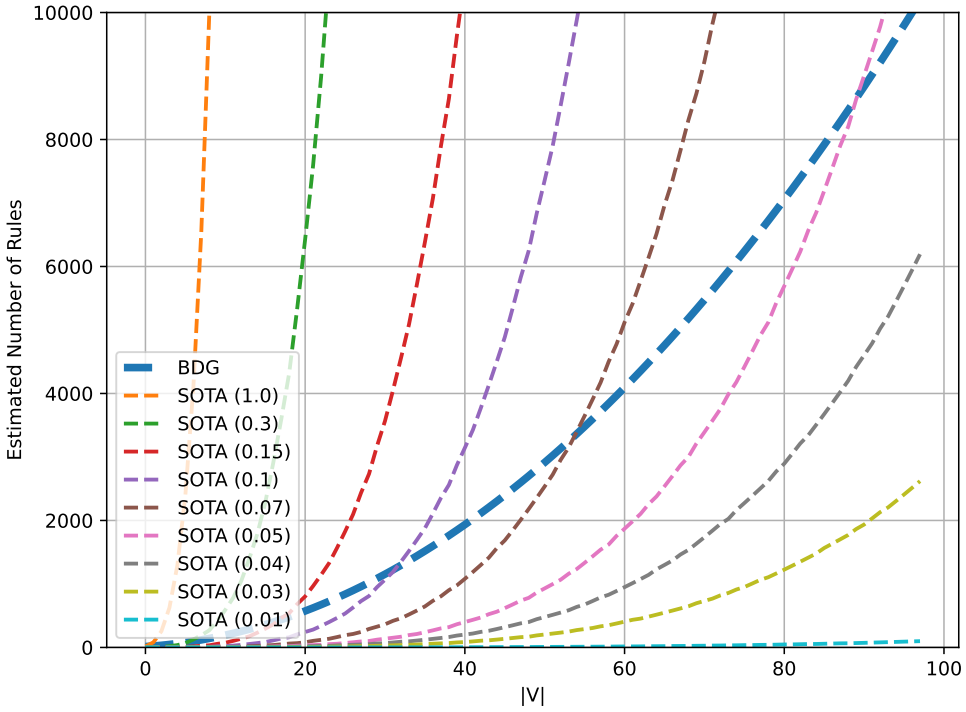


Figure 5.1: Plot comparing the estimated number of grounded rules for the encoding of Example 5.3, for various graph densities (number of edges divided by edges of complete graph), between bottom-up (SOTA), and body-decoupled grounding (BDG). Denser graphs yield a sharper increase in estimation. BDG's estimation is independent of the graph density. Further, BDG has a relatively large constant, which prevents its usage for sparse graphs. The x-axis shows number of vertices, whereas y-axis shows estimated number of rules.

Two major problems emerged in the implementation of `newground3`: (I) Integration and communication with `gringo` and `idlv`, and (II) suitable domain inference for rule estimations of Algorithm 5.3. Note that we performed a full-scale redevelopment of the earlier versions of `newground3` (`newground` and `NaGG`). We further extended its input language to the ASP-Core-2 [22] input language standard[2] and improved the grounding performance of `newground`. On a high level, we did this by interleaving semi-naive grounding with body-decoupled grounding. For the semi-naive grounding parts we use either `gringo`, or `idlv`, whereas, for the body-decoupled grounding part we use a completely redesigned BDG-instantiator. Further, we combine *Python* with *Cython*[3] and *C* code to additionally increase its performance.

[2]As not all ASP-Core-2 constructs can be handled with rewritings based on BDG, we implemented checks s.t. only constructs are considered to be grounded by BDG that can be grounded, while the non-groundable ones are grounded by SOTA-techniques. Non-BDG groundable concepts include non-tight disjunctive parts, and at the time of writing parts where arithmetics is used to infer variables (e.g., $X = Y + Z$).

[3]`https://cython.org/`

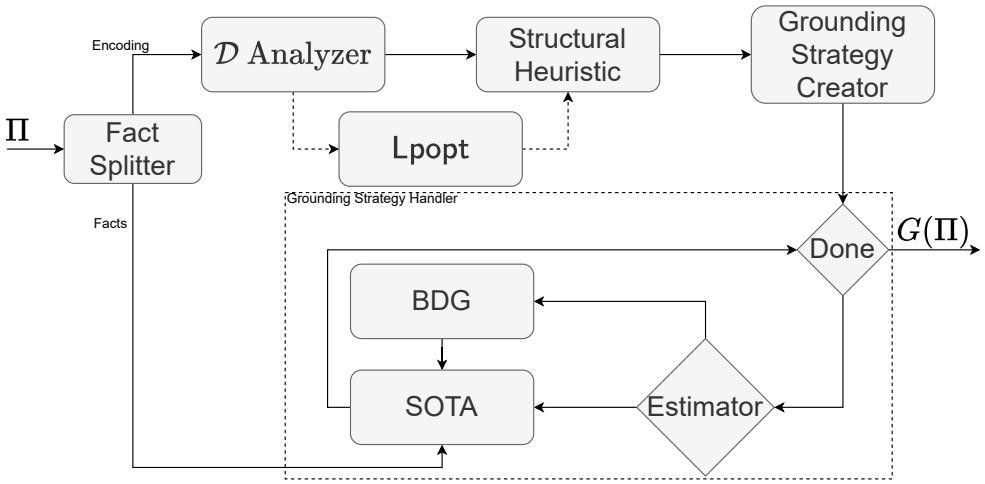


Figure 5.2: Schematics of the `newground3` prototype. Given a program Π, `newground3` outputs a grounded program $G(\Pi)$. The *fact splitter and analyzer* (Fact Splitter) separates facts from encoding. The dependency graph creator and analyzer ($\mathcal{D}$ Analyzer) analyzes SCCs. Using `Lpopt` is optional. The structural heuristics decides whether BDG is useful. This is taken into account by the grounding strategy creator, which produces a bottom-up grounding and BDG compliant grounding strategy. The grounding strategy handler grounds according to the grounding strategy. In that, it incorporates the data heuristics (Estimator).

5.3.1 Structure - Overview

The general architecture of the prototype consists of 6 parts, where we show a schematics in Figure 5.2. (i) The *fact splitter and analyzer*, separates facts from the encoding. It further computes the number of facts, and fact-domain. (ii) The *dependency graph creator and analyzer* computes a (positive and negative) dependency graph from the encoding. Further, it infers which parts of the program are *stratified* and analyzes the *strongly connected components (SCCs)* of the dependency graph. (iii) Our *Lpopt* (optional) heuristics, estimates from the results of the dependency graph analyzer and from the fact-domain, if a treewidth-aware rewriting is useful (Lpopt part of Algorithm 5.3). (iv) Next, we call the structural Algorithm 5.2 to decide up-front which parts are eligible for grounding with BDG and which are (definitely) not. (v) Based on the SCCs of the dependency graph and the results of the Algorithm 5.2, we compute the *grounding strategy*. (vi) The *grounding strategy handler* uses the previous results to iteratively invoke either `gringo` or `idlv`, or our completely overhauled *BDG instantiator*. Importantly, in this step we obtain and utilize the domain and tuple information, as needed for Algorithm 5.3.

We stress that we decided to split Algorithm 5.3 into three parts in our implementation. First, we approximate whether the usage of *Lpopt* is beneficial. Next, we decide with the structural heuristics, which parts are eligible for grounding with BDG. And finally, we decide on the estimated number of instantiated rules if BDG is actually used. These implementation choices improve performance. The *grounding strategy handler* makes successive calls to `gringo` and

idlv, and infers the domain and tuples values for the estimation process. However, these calls are expensive and should better be avoided. Therefore, we do not infer the domain if the result of the structural heuristics tells us that BDG should not be used. Further, our grounding strategy creator minimizes the number of needed calls to SOTA grounders. *We stress that in a (direct) future implementation of BDG in a SOTA grounder, these calls are not necessary, which would improve grounding performance even further.*

5.3.2 Fact Splitter and Analyzer

Written in *Cython*, the *fact splitter and analyzer* resembles a parser-like structure, which takes a (syntactically correct) program and proceeds by splitting it into a fact and an encoding part. We base our choice of implementation on the observation that, while in pure *Python* it is relatively easy to process complex algorithmical concepts (like a saturation check), *Python* is relatively slow when presented with too many input rules. The splitting enables *Python* to be called for a relatively small (non-ground) encoding, as typically, the (non-ground) encoding is much smaller than the fact-base.

5.3.3 Dependency Graph Creator and Analyzer

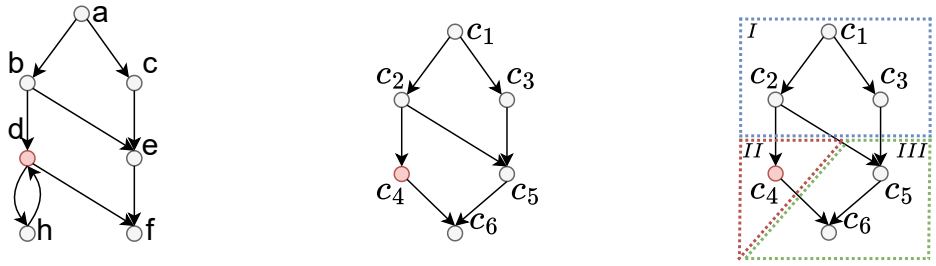
From our non-ground encoding, we proceed with the creation of the dependency graph. We utilize from clingo's AST-library the *transformer*[4], and the *NetworkX* library for this task. While we create the dependency graph, we further parse the non-ground rules into our rule-representation. In our rule-representation, among other parameters, we save whether the rule occurrence is *tight*, its corresponding SCC in the dependency graph, and its variable-graph treewidth[5].

The *transformer* of clingo's AST-library (abbrevation for Abstract Syntax Tree) parses a logic program, and enables developers callback points in various stages in the parsing. For example, by implementing the *visit_Rule* callback, one gets access to the parsed AST-object of a rule. We showcase in Example 8.1 in the Appendix how to use the clingo transformer.

5.3.4 Lpopt Heuristics

Regarding the usage of Lpopt we approximated the number of instantiations according to the Data-Heuristics (Algorithm 5.3). Therefore, we only deem the usage of Lpopt useful, if it is structurally possible to achieve an improvement, and the estimated number of generated ground rules of Lpopt is smaller than the number of estimated rule-instantiations of the original rule. As at the decision time the exact domains, and tuple-sizes are not known, we need to approximate them. This is done by observing the (max) arity of the fact base and the total fact-base size. Let T be the number of tuples in the fact-base, and a be the maximum arity of the fact base. Then T_α is the approximated *tuple* size per arity: $T_\alpha = T^{\frac{1}{a}}$ We use T_α in the Equations (5.1)–(5.9).

[4] https://potassco.org/clingo/python-api/current/clingo/ast.html
[5] Which is actually computed in our structural algorithm.



(a) (b) (c)

Figure 5.3: Dependency graph (a), reduced dependency graph (b), and grounding strategy (c) of Example 5.4. (a): Red node indicates rule grounded with BDG. (b) Reduced dependency graph resembles a DAG. $c_1, \ldots, c_6$ shows a topological ordering. (c) Grounding strategy (boxes I to III). Blue box (I) is grounded first. Then the red box (II) with the rule marked for possible BDG grounding. And lastly, the green box (III).

5.3.5 Structural Heuristics

In the next step we use Algorithm 5.2 for marking rules for being grounded with SOTA, or with BDG methods. In this step, we further compute additional metrics for the rules, like the variable graph treewidth. To combine, both the heuristics and the additional computation, we extend a `clingo` transformer.

5.3.6 Grounding Strategy Creator

Provided the markings of rules from the structural heuristics, the grounding strategy creator, generates a semi-naive grounding compliant order. In essence, this means that the program is analyzed according to the dependency graph $\mathcal{D}$, and whether rules are marked with BDG.

The first step in the creation of the strategy is the computation of a *topological* order on the reduced dependency graph $\mathcal{D}_P^R$. From this topological order *bins*[6] are created. Bins are a collection of rules, with the constraint that all rules in one bin can be grounded either, (a) by one SOTA-ground call, or (b) by one BDG call. If no rule is eligible for grounding with BDG, then all rules are put in a single bin. Note that we keep the relative ordering of the *bins* w.r.t. the dependency graph.

[6]We use *bin* and *grounding-level* interchangebly.

Example 5.4. *We illustrate this in an example. Let us consider the following program:*

```
1 a(1). a(2). a(3). a(4).
2 {b(X)} :- a(X).  {c(X,Y)} :- a(X), a(Y).
3 d :- c(X1,X2), c(X1,X3), c(X2,X3), X1 < X2, X1 < X3, X2 < X3.
4 d :- h.
5 h :- d.
6 e(Z) :- c(X,Y), Z = X + Y.
7 e(X) :- b(X), b(Y), X < Y.
8 f(X) :- e(X), d.
```

We depict in Figure 5.3a the dependency graph of the program above. The rule in Line (3) is marked for grounding with BDG (red node in Figure 5.3a). In Figure 5.3b we show the reduced dependency graph and one topological ordering. The topological ordering is indicated by the c_1 to c_6. Figure 5.3c shows the produced grounding strategy, from the reduced graph. The boxes mark the bins with the ordering (I) to (III).Observe that c_1 to c_3 are put into one bin. This comes, as there is no rule marked for grounding with BDG in these nodes. Additionally, there is no node on a path between any of the nodes of the bin, where there is a rule marked for grounding with BDG. However, as c_4 contains a node marked for grounding with BDG, it must be put in a bin different from c_1 to c_3. Further, c_6 cannot be put into bin (I), as the path $c_2 - c_4 - c_6$ contains a node (c_4), where at least rule is marked for grounding with BDG.

However, c_5 could be put into bin (I). Therefore, observe that there are multiple possible bin configurations, which depend on the topological ordering. To showcase this, another possible bin configuration would be: I: c_1, c_2; II: c_4; III: c_3, c_5, c_6.

5.3.7 Data Heuristics (Grounding Strategy Handler)

Making use of the generated bins, the *grounding strategy handler* implements the final parts of the *data heuristics*. This corresponds to the implementation of the *estimations* of the instantiated rule sizes. We show the pseudo-code in Algorithm 5.4.

Let $\mathcal{F}$ be the facts, $\mathcal{B}$ be the bins, and $(b_S, b_B) \in \mathcal{B}$ is a tuple that denotes the rules that are marked for SOTA- (b_S) and for BDG-grounding (b_B), respectively. $\mathcal{H}$ is the hybrid-, and $\mathcal{G}$ is the SOTA-grounding procedure. *domain_inference* infers the domain that is needed for the rule size estimation. Finally, $G(\Pi)$ is the grounded program.

Intuitively, the algorithm grounds each bin $b_S, b_B \in \mathcal{B}$ iteratively. Thereby, we ensure accurate domain inference for the estimation procedure and for the instantiation of BDG. Line (1) initializes the grounding $G(\Pi)$ with the facts. Lines (2)–(14) iterate over each bin $b_S, b_B \in \mathcal{B}$. Then, Lines (3)–(11) iterate over every rule r_B marked with BDG by the structural heuristics. For each r_B it is decided (Line (4)) whether to be grounded with BDG (Lines (5)–(7)), or to be grounded by SOTA means (Line (9)). Rules that are to be grounded with SOTA grounders are grounded in Lines (12)–(14).

Due to the communication challenges between `newground3` and SOTA grounders, we discuss Lines (5)–(7) and (12)–(14) in more detail. `idlv` and `gringo` take a standard ASP-Core-2 [22] as its input. However, (I) their standard output formats differ, and (II) they were not designed for

successive grounding calls. On standard settings `idlv` uses the *SMODELS* output format (or *Lparse* format) [124], while `gringo` uses the ASPIf (ASP Intermediate Format) [86]. Note that there was an ongoing discussion on intermediate formats [81], and other proposals such as the *ASPils* (ASP Intermediate Language Standard) [57].

Algorithm 5.4: *Handler*$(\mathcal{B}, \mathcal{F})$; Grounding Strategy Handler

Data: Bins $\mathcal{B}$, Facts $\mathcal{F}$
Result: Grounded program $G(\Pi)$

$G(\Pi) \leftarrow \mathcal{F}$;
for $(b_S, b_B) \in \mathcal{B}$ **do**
  for $r_B \in b_B$ **do**
    if $T_{\mathcal{H}}(r_B) < T_{\bowtie}(r_B)$ **then**
      $\Pi_t \leftarrow \mathcal{H}(r_B, \emptyset)$;
      domain_inference(Π_t) ;
      $G(\Pi) \leftarrow G(\Pi) \cup \Pi_t$;
    else
      $b_S \leftarrow b_S \cup r_B$;
    end
  end
  $\Pi_t \leftarrow \mathcal{G}(b_S, G(\Pi))$;
  domain_inference(Π_t) ;
  $G(\Pi) \leftarrow G(\Pi) \cup \Pi_t$;
end
return $G(\Pi)$;

Algorithm 5.4 requires iterative grounding calls. However, neither `gringo`, nor `idlv`, are able to take one of their standard output formats as an input. Furthermore, their *string* output format was not designed to be used for successive grounding calls. This comes, as `idlv` deviates from the ASP-Core-2 standard in some cases, when they use heuristics for producing smaller groundings. They deviate in predicate naming, as they use upper-case-letters as the first letter. Take for example *Valves Location Problem* from the 2014 ASP-Competition [25]. There we obtain for `idlv`[7] the following rule in the grounded program:

```
1 Aux_10(5,1,pipe(1,2)):-deliver(pipe(1,2),pipe(1,5)).
```

The predicate *Aux_10* has an upper-case first-letter, which is not supported by the ASP-Core-2 standard. Therefore, a successive solver call to `clingo` (as `clingo` accepts a valid (extended) ASP-Core-2 program) results in a syntax error. In contrast, with `gringo`[8], we obtain (among other rules) the following rule:

```
1 #delayed(1).
```

[7] Tested with the *new* encoding of *Valves Location Problem*, and *instance 0001-ValvesLocationProblem-166-0.asp*, `idlv` version 1.1.6.

[8] Tested with the same example as `idlv` was tested, and `gringo` version 5.6.2.

According to the ASP-Core-2 standard there are no rules that start with a #. In contrast to this, `clingo` takes extensive use of such constructs for defining (for example) $\#show$ statements [70]. However, the actual problem is that successive grounding operations produce *lexer* errors in `gringo`[9].

Fortunately, both `gringo` and `idlv` support the *SMODELS* format. As a result, we decided to use *SMODELS* as our intermediate format for communication between SOTA grounders and `newground3`. But, as SOTA grounders do not provide an interface for *integrating callbacks* into their semi-naive grounding procedure, and as we observed that SOTA grounders spend a considerable amount of time on large ground programs, we decided to use SOTA grounders solely for domain inference. Therefore, we re-ground in Line (12) in Algorithm 5.4 on the whole program, and we *stress that a direct integration of BDG (with the above described heuristics) into a SOTA grounder would yield an even better performance*.

Novel BDG Instantiator

Our completely re-implemented BDG instantiator uses a combination of Cython, Python and C. While the previous versions relied on a `clingo` transformer, the current version uses internal data-structures to produce the BDG-rewritten program.

Domain Inference

As discussed above, we use the *SMODELS* format as our intermediate format for communication between SOTA grounders and `newground3`. Using *SMODELS* with its block-style format has the benefit for efficient inference of the domain. This comes, as all atoms occur in the second block of the format.

5.4 Experiments

In the following, we demonstrate the practical usefulness of our automated hybrid grounding approach. We benchmark our system on solving- and grounding-heavy benchmarks, with the aim of achieving SOTA results on solving-heavy benchmarks, and beating SOTA results on grounding-heavy benchmarks.

5.4.1 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 heuristics hybrid grounding system `newground3`. We benchmarked our system together with both `gringo`, and `idlv`. Further, we investigated the impact of using it in conjunction with `Lpopt` (Version 2.2).

We chose `clingo` (Version 5.7.1) with `clasp` (3.3.10) as our solver of use. However, we want to note that in principle one could also use `dlv` with `wasp`, or use heuristics to decide on the solver to use, as e.g., suggested in [21].

[9] Re-grounding the problem instance from above we get the error *error: lexer error, unexpected #delayed.*

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. The operating system is a *Debian 10* with kernel 4.19.0-16-amd64.

5.4.2 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, -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, the grounded program and solving.

For the ground- and-solve approaches, we used `clingo` as our solver. 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.

5.4.3 Experiment Scenarios and Instances

We distinguish between *solving*- and *grounding*-heavy benchmarks. For the solving-heavy benchmarks we compare `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `gringo` and `Lpopt` (NG-G-TW), `newground3` with `idlv` (NG-I), ALPHA (Alpha), and ProASP (ProASP) (ground-all). We executed `newground3` in the *relevance mode*. If activated we use BDG only on those scenarios, where the number of variables for a rule grounded with BDG is strictly greater than the number of variables for any rule grounded by SOTA grounders.

For the grounding-heavy we compare `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `idlv` (NG-I), ALPHA (Alpha), ProASP (ProASP) (ground-all), and ProASP (ProASP-CS) (compile constraints). For all grounding-heavy benchmarks `Lpopt` (NG-G-TW) is never activated (and therefore, not explicitly measured). We executed `newground3` without the *relevance mode*.

Solving-Heavy Benchmarks

The solving-heavy benchmarks are taken from the 2014 ASP-Competition [25], 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 an old and a new encoding, resulting in 49 competition scenarios. Each scenario has a different number of instances (see Tables 1, and 2). 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 problem *23O-MaximalCliqueProblem-New*[10] as an example:

```
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 no edge between any two nodes in the clique can be found, then it is no clique (Line (3)). Finally, we want to maximize the number of cliques, or equivalently minimize the number of non-clique nodes (Line (4)). An instance defines *nodes* and *edges*:

```
1 node(1). node(2). ...
2 edge(1,2). edge(1,3). ...
```

Grounding-Heavy Benchmarks

We take grounding-heavy benchmarks from [12] and from [10]. These scenarios take as an input a graph, where we generate random graphs ranging from instance size 100 to 2000 with a step-size of 100, on graph density levels ranging from 0.2 to 1.0 (number of edges divided by the number of edges of a complete graph).

Further, we adapt the benchmarks from [12] by adding two variations of the clique example. The variations resemble different difficulties for BDG and SOTA grounders. The first listing shows the original formulation (31-Cliqu.(!=)) from [12], and the second one, the adaptation which makes it easier for SOTA grounders (30-Cliqu.) 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[11] scenarios from [12] are called as follows: 00-directed-clique (30-Cliqu.), 01-directed-clique-not-equal (31-Cliqu.(!=)), 02-directed-paths (32-Path.), 03-directed-coloring (33-Col.), 04-directed-4-clique (34-4Cliqu.), 05-nprc (35-NPRC.). The examples 36-S3T4, 37-S4T4, 38-NPRC, and 39-SM-AGG, are from [10], and examples 40-SM-NEW and 40-SM-OLD are stable matching encodings with denser instances.

[10] The whole competition suite can be found in: `https://www.mat.unical.it/aspcomp2014/FrontPage`

[11] As ProASP's syntax does currently not support choice rules, 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).*). We also used this adaptation for ALPHA.

5.4.4 Experimental Hypotheses

H1 The Data-Structural-Heuristics (Algorithm 5.3) implemented in our prototype `newground3` leads to approximately equal results on solving-heavy benchmarks, in comparison to other SOTA ground-and-solve systems.

H2 The Data-Structural-Heuristics (Algorithm 5.3) implemented in our prototype `newground3` yields an improvement in performance (solved instances) on some grounding-heavy benchmarks, in comparison to other SOTA ground-and-solve systems.

5.4.5 Experimental Results

We show our detailed results in 4 tables (each for grounding- and solving-heavy instances): Table 1 shows the number of solved instances, Table 4 shows the overall total-time and RAM usage, Table 2 shows the number of grounded instances, and Table 3 shows the grounding-time and -size. Figure 5.4 shows the solving performance on solving- and grounding-heavy scenarios for ground- and solve-systems, Figure 5.5 extends it to ALPHA, Figure 5.6 to ProASP, and Figure 5.7 shows the grounding performance of ground- and solve-systems on grounding- and solving-heavy scenarios. In the following, we argue first that H1 can be confirmed and then that H2 holds.

H1

We focus our attention on the results of the solving-heavy experiments. These are displayed in Figure 5.4, in the solving-heavy part of Table 1 (solving table), and in the solving-heavy part of Table 2 (grounding table). Figure 5.4 shows that `newground3`'s performance is approximately the same compared to the other ground-and-solver approaches. Table 1 shows 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` vs. NG-G and 0.41% for `idlv` vs. NG-I. Also observe that for `gringo` vs. NG-G there are cases where `gringo` beats NG-G and cases where NG-G beats `gringo`. Take for example *16-Incr.*, where `gringo` has 77 solved instances and NG-G has 75, out of 500 instances, and example *02-Valv.-New*, where `gringo` has 74 solved instances and NG-G has 76, out of 318 instances. The same holds for `idlv` vs. NG-I. As these differences are minor, we argue that H1 is confirmed by our experiments.

H2

We compare the results for the grounding-heavy benchmark in Tables 1, and 2, and Figures 5.4, and 5.8. Note in Figure 5.4 the difference between `gringo` and `idlv`, to NG-G and NG-I. This manifests itself also in Table 1, where `gringo` solves 221, and `idlv` 293, compared to 584 of NG-G and 642 of NG-I, from a total of 1000 instances.

Further, observe Figure 5.8, which shows the detailed solving profile for scenario *31-Cliqu.(!=)*. Whereas `gringo` (Figure 5.8a) and `idlv` (Figure 5.8c) were able to solve most instances when

the instance density is low, they failed to solve instances with higher instance densities (80 and 100). In contrast to this NG-G (Figure 5.8b), and NG-I (Figure 5.8d) were able to solve about 4 times as many instances on high instance densities. Observe the solving time jump at instance size $\approx$ 1200, which we argue is due to a change in used heuristics in `clingo` (solver-side). Finally, the other solving profiles shown in the Appendix are in line with the solving profile in Figure 5.8.

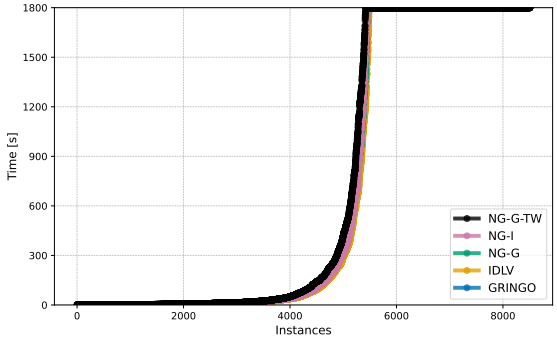
As `newground3`'s ability to automatically determine when to use BDG led to an approximate doubling in the number of solved grounding-heavy instances, we can confirm H2.

Discussion of Hybrid Approaches

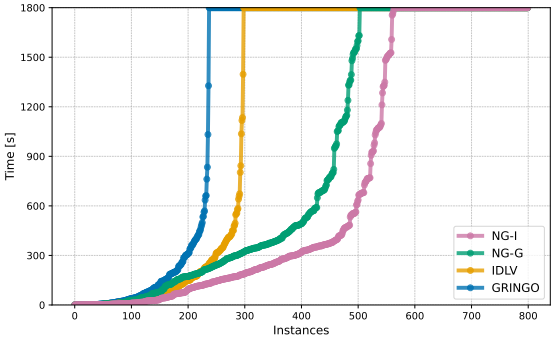
For both solving- and grounding-heavy benchmarks NG-G and NG-I outperformed ALPHA significantly. ProASP has a comparable performance on solving-heavy benchmarks. On grounding-heavy benchmarks, ProASP shows promising results, however only when we use ProASP in the compile constraints mode. However, ALPHA and ProASP are only usable for a small fragment of scenarios.

`newground3` with `Lpopt`

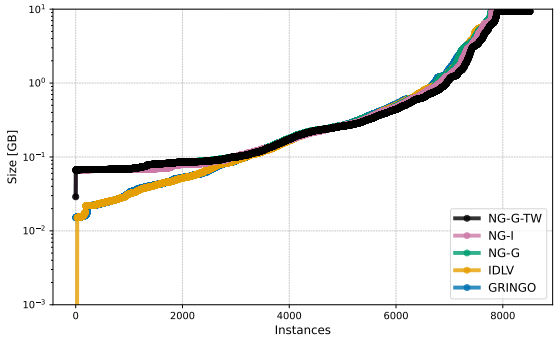
Our experimental setup of NG-G-TW combines BDG with `Lpopt`. NG-G-TW grounds more solving-heavy instances than NG-G (8019 vs. 7925), however,less than `gringo` (8019 vs. 8037). In solving NG-G-TW solves less than NG-G (5409 vs. 5418), and less than `gringo` (5409 vs. 5449). Although these numbers do neither show significant improvements, nor losses, the truly interesting number occur when viewing individual scenarios. Take for example (01-Perm.-New), where NG-G-TW improves upon `gringo` (195 vs. 171). However, it also has losses in other instances, like (15-Stabl.), where NG-G-TW solves less than `gringo` (32 vs. 48). Therefore, we conclude that reducing the grounding size with `Lpopt` does not necessarily lead to better solving results.



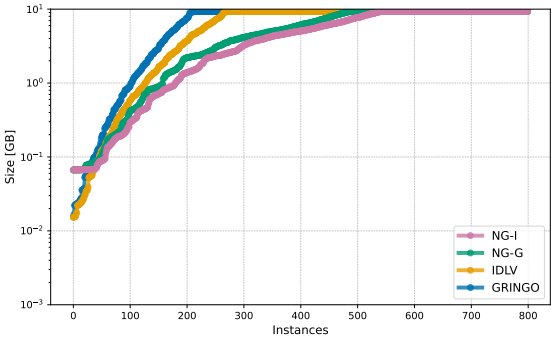
(a) Solving-heavy. Grounding and Solving Time [s] cactus plot.



(b) Grounding-heavy. Grounding and Solving Time [s] cactus plot.

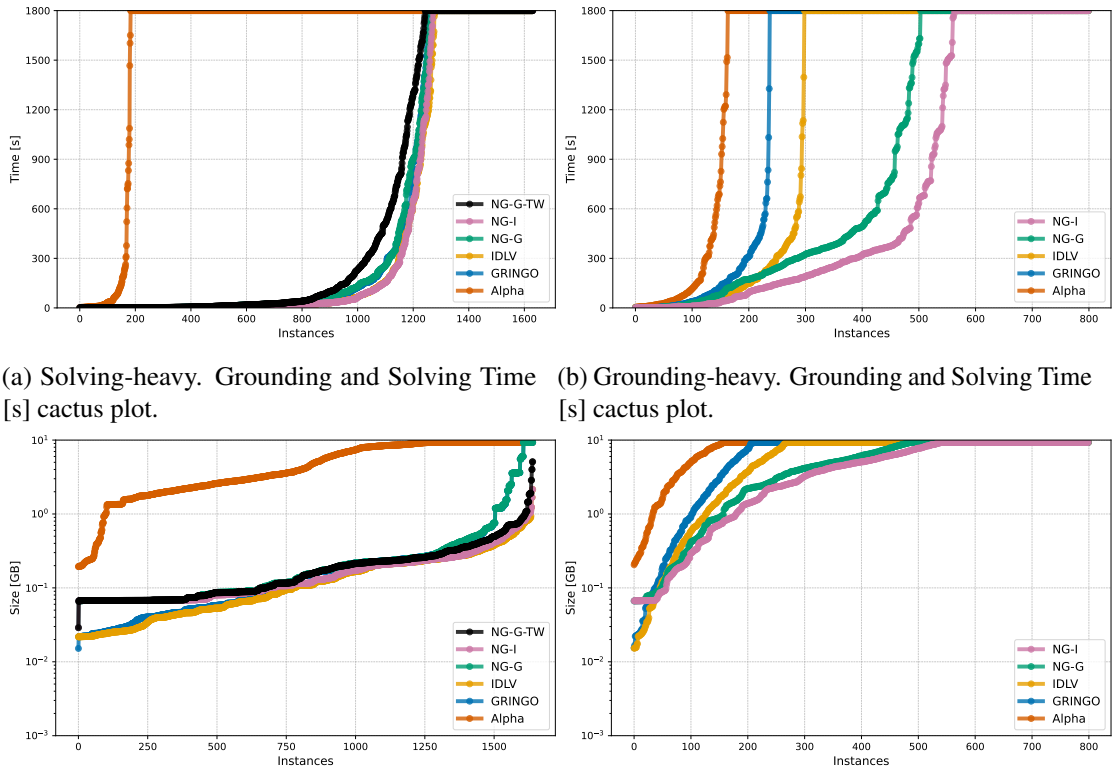


(c) Solving-heavy. Max RAM Usage [GB].



(d) Grounding-heavy. Max RAM Usage [GB].

Figure 5.4: `newground3` manages to obtain the same results on solving-heavy benchmarks as do `gringo` and `idlv` obtain (Figure 5.4a, and Figure 5.4c). `newground3` solves more benchmarks on grounding-heavy benchmarks than `gringo` and `idlv` (Figure 5.4b, and Figure 5.4d). Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `gringo` and `Lpopt` (NG-G-TW), and `newground3` with `idlv` (NG-I). Timeout: 1800s; Memout: 10GB.



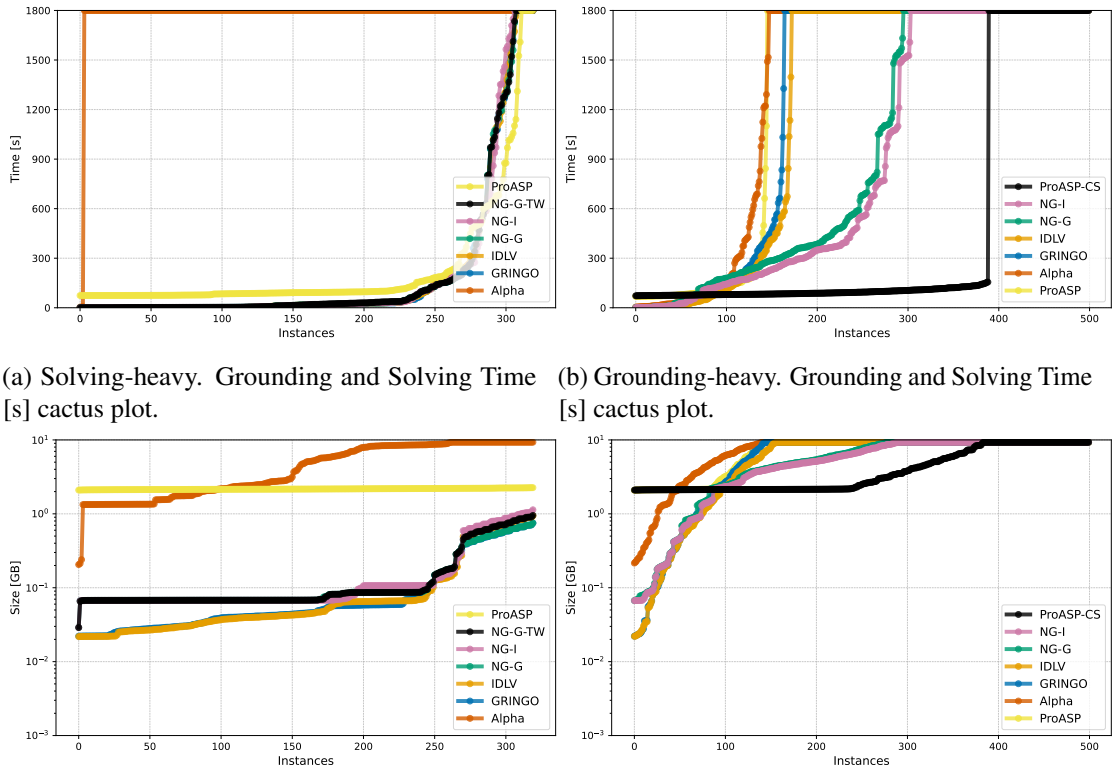
(a) Solving-heavy. Grounding and Solving Time [s] cactus plot.

(b) Grounding-heavy. Grounding and Solving Time [s] cactus plot.

(c) Solving-heavy. Max RAM Usage [GB].

(d) Grounding-heavy. Max RAM Usage [GB].

Figure 5.5: ALPHA performs worse than ground-and-solve systems on both solving-, and grounding-heavy benchmarks. `newground3` manages to obtain the same results on solving-heavy benchmarks as do `gringo` and `idlv` obtain (Figure 5.5a, and Figure 5.5c). `newground3` solves more benchmarks on grounding-heavy benchmarks than `gringo` and `idlv` (Figure 5.5b, and Figure 5.5d). Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `gringo` and `Lpopt` (NG-G-TW), `newground3` with `idlv` (NG-I), and ALPHA (Alpha). Timeout: 1800s; Memout: 10GB.



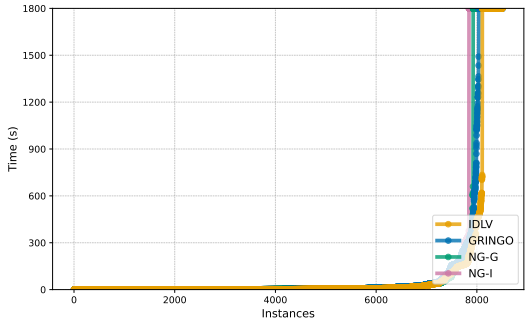
(a) Solving-heavy. Grounding and Solving Time [s] cactus plot.

(b) Grounding-heavy. Grounding and Solving Time [s] cactus plot.

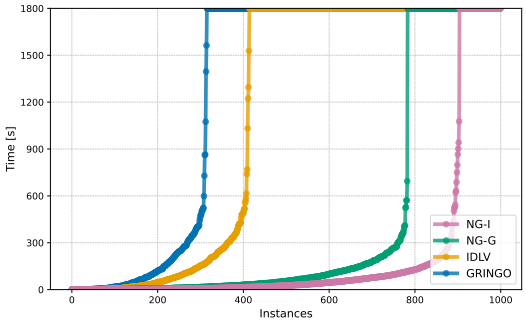
(c) Solving-heavy. Max RAM Usage [GB].

(d) Grounding-heavy. Max RAM Usage [GB].

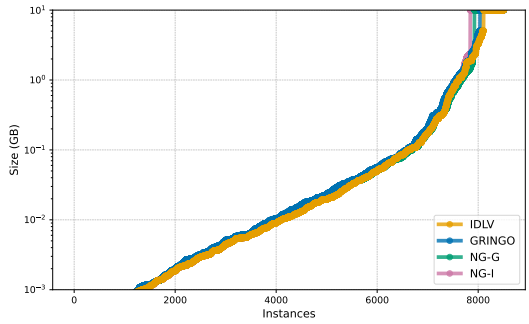
Figure 5.6: ProASP (ProASP), and ProASP with CS (ProASP-CS) show potential on both solving- and grounding-heavy benchmarks, however it is limited in input syntax. ALPHA performs worse than ground-and-solve systems on both solving, and grounding-heavy benchmarks. `newground3` manages to obtain the same results on solving-heavy benchmarks as do `gringo` and `idlv` obtain (Figure 5.6a, and Figure 5.6c). `newground3` solves more benchmarks on grounding-heavy benchmarks than `gringo` and `idlv` (Figure 5.6b, and Figure 5.6d). Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `gringo` and `Lpopt` (NG-G-TW), `newground3` with `idlv` (NG-I), and ALPHA (Alpha). Timeout: 1800s; Memout: 10GB.



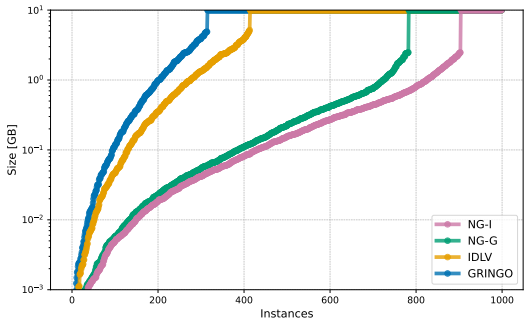
(a) Solving-Heavy Instances. Grounding Time [s] cactus plot.



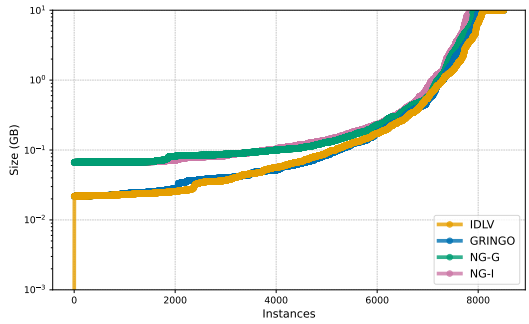
(b) Grounding-Heavy Instances. Grounding Time [s] cactus plot.



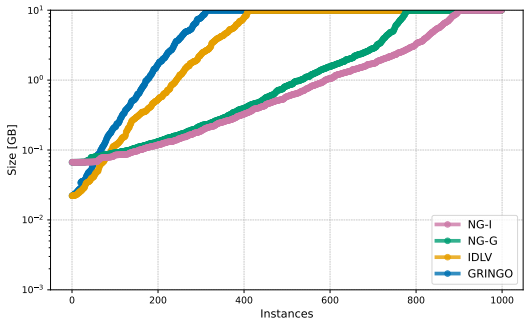
(c) Solving-Heavy Instances. Grounding Size [GB] cactus plot.



(d) Grounding-Heavy Instances. Grounding Size [GB] cactus plot.

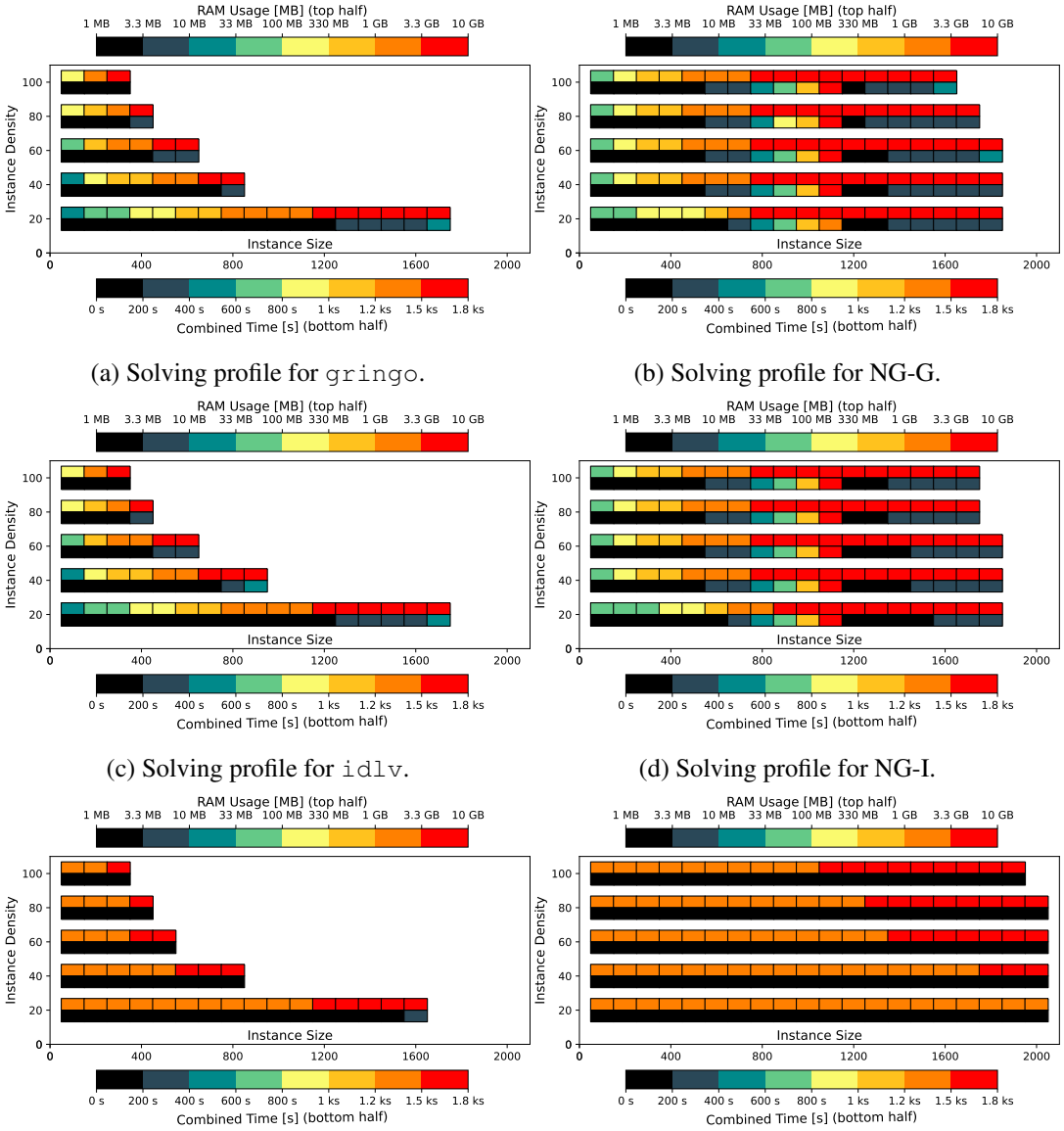


(e) Solving-Heavy Instances. RAM Usage [GB] cactus plot.



(f) Grounding-Heavy Instances. RAM Usage [GB] cactus plot.

Figure 5.7: Solving and Grounding-heavy instances, grounding specific benchmarks for ground-and-solve approaches. Cactus plots measuring grounding times of solving-heavy (Figure 5.7a), and grounding-heavy (Figure 5.7b), grounding sizes of solving-heavy (Figure 5.7c), and grounding-heavy (Figure 5.7d), and (total) RAM usages of solving-heavy (Figure 5.7e), and grounding-heavy (Figure 5.7f). Timeout: 1800s; Memout: 10GB.



(a) Solving profile for `gringo`.

(b) Solving profile for NG-G.

(c) Solving profile for `idlv`.

(d) Solving profile for NG-I.

(e) Solving profile for ProASP.

(f) Solving profile for ProASP$_{CS}$.

Figure 5.8: Solving profiles for all benchmarked systems for the *31-Cliqu.(!=)* example. Contrasting the traditional Ground-and-Solve techniques, to the hybrid systems it is evident that the hybrid systems perform better. Compare Figure 5.8a to 5.8b, 5.8c to 5.8d, and 5.8e to 5.8f. Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `gringo` and `Lpopt` (NG-G-TW), `newground3` with `idlv` (NG-I), ProASP (ProASP) and, ProASP where constraints are lazily grounded (ProASP$_{CS}$). Timeout: 1800s; Memout: 10GB.

CHAPTER 6

FastFound: Using Saturation for Foundedness

FastFound improves the hybrid grounding [10] reduction, by using saturation [43]. Recall that the grounding size of body-decoupled grounding [11] is dependent on the maximum arity a of a program Π. It is in $\mathcal{O}\left(|\Pi| \cdot |\operatorname{dom}(\Pi)|^{a}\right)$ for constraints and in $\mathcal{O}\left(|\Pi| \cdot |\operatorname{dom}(\Pi)|^{2 \cdot a}\right)$ for normal rules. The increase from a to $2 \cdot a$ prevents the practical usage of BDG for normal rules.

With FastFound we reduce the grounding size for normal rules to $\mathcal{O}\left(|\Pi| \cdot |\operatorname{dom}(\Pi)|^{a+1}\right)$. While in the original formulation, the foundedness check encoded each guess explicitly, we use saturation to implicitly perform the for-all check.

The FastFound reduction is introduced in Section 6.1, which is followed by an extension to HCF programs (Section 6.2), a demonstration on Examples (Section 6.3), and experiments showcasing its practical usefulness (Section 6.4).

6.1 The FastFound Reduction

Our FastFound reduction shown in Figure 6.1 consists of three parts: Guessing of the heads (Rules (6.1)–(6.3)), ensuring satisfiability (Rules (6.4)–(6.9)), and ensuring foundedness (Rules (6.10)–(6.19)). The head guess and the satisfiability parts are the same as in hybrid grounding, whereas the foundedness part differs. Our FastFound reduction enables the free (manual) partition of a normal program Π into a part Π_1 grounded by the reduction and a part Π_2 grounded by traditional techniques.

The intuition behind the new-foundedness part is to use saturation to guess every possible variable assignment for a non-ground head literal. This corresponds to the grounding and as in standard-grounding, only a subset of the thereby grounded head literals holds for an arbitrary interpretation $\mathcal{I}$. This has one important implication: We only need to verify foundedness of a head literal, if it

Glue Π_2 to Π_1 and Ground Π_2

$h'(\mathbf{D}) \vee \overline{h'(\mathbf{D})} \leftarrow$ for every $r \in \Pi_1$, $h(\mathbf{X}) \in \text{hpred}(\Pi_1), \mathbf{D} \in \text{dom}(\mathbf{X})$ (6.1)

$h(\mathbf{D}) \leftarrow h'(\mathbf{D})$ for every $r \in \Pi_1$, $h(\mathbf{X}) \in \text{hpred}(\Pi_1), \mathbf{D} \in \text{dom}(\mathbf{X})$ (6.2)

r for every $r \in \mathcal{G}(\Pi_2)$ (6.3)

Satisfiability of Π_1

$\bigvee_{d \in \text{dom}(\mathbf{x})} \text{sat}_x(d) \leftarrow$ for every $r \in \Pi_1, x \in \text{var}(r)$, where $\Pi_1 = \{r_1, \ldots, r_n\}$ (6.4)

$\text{sat} \leftarrow \text{sat}_{r_1}, \ldots, \text{sat}_{r_n}$ for every $r \in \Pi_1, x \in \text{var}(r)$, where $\Pi_1 = \{r_1, \ldots, r_n\}$ (6.5)

$\text{sat}_r \leftarrow \text{sat}_{x_1}(\mathbf{D}_{\langle x_1 \rangle}), \ldots, \text{sat}_{x_t}(\mathbf{D}_{\langle x_t \rangle}), \neg p(\mathbf{D})$ for every $r \in \Pi_1, p(\mathbf{X}) \in B_r^+$, $\mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle$ (6.6)

$\text{sat}_r \leftarrow \text{sat}_{x_1}(\mathbf{D}_{\langle x_1 \rangle}), \ldots, \text{sat}_{x_t}(\mathbf{D}_{\langle x_t \rangle}), p(\mathbf{D})$ for every $r \in \Pi_1, p(\mathbf{X}) \in B_r^- \cup H_r$, $\mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle$ (6.7)

$\text{sat}_x(d) \leftarrow \text{sat}$ for every $r \in \Pi_1, x \in \text{var}(r)$, $d \in \text{dom}(\mathbf{x})$ (6.8)

$\leftarrow \neg\text{sat}$ (6.9)

Foundedness of Π_1

$\bigvee_{d \in \text{dom}(x)} \text{just}_x(d) \leftarrow$ for every $r \in \Pi_1, \boldsymbol{X} = \text{var}(H_r), x \in \boldsymbol{X}$ (6.10)

$1\{\text{justh}_y(d_1, \boldsymbol{D}), \ldots, \text{justh}_y(d_k, \boldsymbol{D})\}1 \leftarrow h'(\boldsymbol{D})$ for every $r \in \Pi_1, \boldsymbol{X} = \text{var}(H_r), \boldsymbol{D} \in \text{dom}(\boldsymbol{X})$, $y \in (\text{var}(r) \setminus \boldsymbol{X}), \{d_1, ..., d_k\} = \text{dom}(y)$ (6.11)

$\text{just}_y(d) \leftarrow \text{justh}_y(d, \boldsymbol{D}), \text{just}_{x_1}(d_1), \ldots, \text{just}_{x_t}(d_t)$ for every $r \in \Pi_1, \langle x_1, \ldots, x_t \rangle = \boldsymbol{X} = \text{var}(H_r)$, $\langle d_1, \ldots, d_t \rangle = \boldsymbol{D} \in \text{dom}(\boldsymbol{X})$, $y \in (\text{var}(r) \setminus \text{var}(H_r)), d \in \text{dom}(y)$ (6.12)

$\text{just} \leftarrow \text{just}_{r_1}, \ldots, \text{just}_{r_n}$ let $\Pi_1 = \{r_1, \ldots, r_n\}$ (6.13)

$\text{just}_r \leftarrow \text{lit}_{p_2}, \ldots, \text{lit}_{p_m}, \text{lit}_{p_{m+1}}, \ldots, \text{lit}_{p_n}$ for every $r \in \Pi_1$, where $B_r^+ = \{p_2, \ldots, p_m\}$ where $B_r^- = \{p_{m+1}, \ldots, p_n\}$ (6.14)

$\text{lit}_p \leftarrow \text{just}_{x_1}(\mathbf{D}_{\langle x_1 \rangle}), \ldots, \text{just}_{x_t}(\mathbf{D}_{\langle x_t \rangle}), p(\mathbf{D})$ for every $r \in \Pi_1, p(\mathbf{X}) \in B_r^+$, $\mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle$ (6.15)

$\text{lit}_p \leftarrow \text{just}_{x_1}(\mathbf{D}_{\langle x_1 \rangle}), \ldots, \text{just}_{x_t}(\mathbf{D}_{\langle x_t \rangle}), \neg p(\mathbf{D})$ for every $r \in \Pi_1, p(\mathbf{X}) \in B_r^-$, $\mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle$ (6.16)

$\text{just}_r \leftarrow \text{just}_{x_1}(\mathbf{D}_{\langle x_1 \rangle}), \ldots, \text{just}_{x_t}(\mathbf{D}_{\langle x_t \rangle}), \neg h'(\mathbf{D})$ for every $r \in \Pi_1, h(\mathbf{X}) \in H_r$, $\mathbf{D} \in \text{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle$ (6.17)

$\text{just}_x(d) \leftarrow \text{just}$ for every $r \in \Pi_1, x \in \text{var}(H_r)$, $d \in \text{dom}(\mathbf{x})$ (6.18)

$\leftarrow \neg\text{just}$ (6.19)

Figure 6.1: Alternative foundedness check ($\mathcal{HG}_\mathcal{F}$) for hybrid grounding that uses saturation for checking foundedness for normal (tight) ASP. Grounding size is in $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{a+1}\right)$.

is part of the interpretation $p \in \mathcal{I}$. Conversely, if $p \notin \mathcal{I}$ we can skip this literal (Rule (6.17)). If we are required to verify it, we need to find at least one body variable assignment s.t. the body holds (Rules (6.11)–(6.12)). This is the case when all positive body predicates $p \in B_r^+$ hold (Rules (6.14) and (6.15)), and not a single negative body predicate $p \in B_r^-$ holds (Rules (6.14), and (6.16)). The Rules (6.10) and (6.18) are technically necessary for encoding saturation and

ensure that all head instantiations are checked for foundedness. Finally, Rule (6.13) encodes that all rule heads must be justified, and Rule (6.19) prevents unfounded answer sets.

The grounding size of $\mathcal{HG}_\mathcal{F}$ is in $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{a+1}\right)$. Rules (6.11)–(6.12) have in the worst case $a+1$ variables, whereas all other rules have in the worst case only a variables. In more detail, let r be a rule, $a_h(r)$ be rule's maximum head arity and $a_B(r)$ be the rule's maximum body arity. Then let $a_{max} = \max_{r\in\Pi} \max\{a_h(r)+1, a_B(r)\}$, which means that the grounding size of $\mathcal{HG}_\mathcal{F}$[1]. is in $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{a_{max}}\right)$.

6.1.1 Theorems and Proofs

Theorem 6.1. *Given a partition of any non-ground tight normal program Π into a program Π_1 and a program Π_2, then answer sets of $\mathcal{HG}_\mathcal{F}(\Pi_1, \Pi_2)$ restricted to $\mathrm{at}(\mathcal{G}(\Pi))$ bijectively match the answer sets of $\mathcal{G}(\Pi)$.*

Proof (detailed-idea). We proceed by proving two directions, (i) whenever given a program Π, every answer set of $\mathcal{G}(\Pi)$ is an answer set of $\mathcal{HG}_\mathcal{F}(\Pi_1, \Pi_2)$, and the contrary (ii) (w.r.t. to the restriction of $\mathrm{at}(\mathcal{G}(\Pi))$). First (i) and then (ii) is shown.

(i): $\mathcal{G}(\Pi) \rightarrow \mathcal{HG}_\mathcal{F}(\Pi_1, \Pi_2)$
Let $\mathcal{I} \in \mathcal{AS}(\mathcal{G}(\Pi))$ be an answer set of Π. We need to check, whether there exists an $\mathcal{I}_{\mathcal{HG}_\mathcal{F}} \in \mathcal{AS}(\mathcal{HG}_\mathcal{F}(\Pi_1, \Pi_2))$, where $\mathcal{I}_{\mathcal{HG}_\mathcal{F}} \cap \mathrm{at}(\mathcal{G}(\Pi)) = \mathcal{I}$. We construct $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$ in a way s.t. $\mathcal{I} \subseteq \mathcal{I}_{\mathcal{HG}_\mathcal{F}}$, and $\mathcal{I}_{\mathcal{HG}_\mathcal{F}} = \mathcal{I} \cup \mathcal{I}_{\text{aux}} \cup \mathcal{I}_{\text{sat}} \cup \mathcal{I}_{\text{found}}$, where $\mathcal{I}_{\text{aux}}$ are those atoms from Rule (6.1), $\mathcal{I}_{\text{sat}}$ from Rules (6.4)–(6.9), and $\mathcal{I}_{\text{found}}$ from Rules (6.10)–(6.19).

For all rules $r \in \mathcal{G}(\Pi_2)$, we know that r is satisfied ($B_r^+ \setminus \mathcal{I} \neq \emptyset$, or $(H_r \cup B_r^-) \cap \mathcal{I} \neq \emptyset$) by construction. Further, $\mathcal{G}(\Pi_2)$ justify a subset of $\mathcal{I}_{\Pi_2} \subseteq \mathcal{I}$. $\mathcal{I}_{\Pi_2}$ are all those $h(\mathbf{D})$ s.t. $h(\mathbf{D}) \in \text{heads}(\mathcal{G}(\Pi_2))$, $h(\mathbf{D}) \in \mathcal{I}$, and $\exists r \in \mathcal{G}(\Pi_2)$, s.t. $B_r^+ \subseteq \mathcal{I}$ and $B_r^- \cap \mathcal{I} = \emptyset$ (founded). Let $\mathcal{I}_{\Pi_1}$ be those atoms that need to be justified with $\mathcal{HG}_\mathcal{F}$ ($\mathcal{I}_{\Pi_1} = \mathcal{I} \setminus \mathcal{I}_{\Pi_2}$).

We now proceed by showing that we can satisfy all Rules (6.1)–(6.19), and justify all $\mathcal{I}_{\Pi_1}$, by constructing a $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$. Observe that by the results of [10], [12] it is known that Rules (6.1)–(6.9) are satisfied by $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$, and also justify all respective satisfiability ($\mathcal{I}_{\text{sat}}$), and auxiliary-head ($\mathcal{I}_{\text{aux}}$) atoms in $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$. We construct $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$ such that for each $h_r(\mathbf{D}) \in H(\mathcal{G}(\Pi_1))$, there is at least one compatible $h'_r(\mathbf{D}) \in \mathcal{I}_{\mathcal{HG}_\mathcal{F}}$. Although this justifies all atoms $\mathcal{I}_{\Pi_1}$, we are left to show that Rules (6.10)–(6.19) indeed are satisfied and atoms $\mathcal{I}_{\text{found}}$ are justified.

In order that $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$ is indeed an answer set, it must hold that just $\in \mathcal{I}_{\text{found}}$. First (a) we check that "just" being part of $\mathcal{I}_{\text{found}}$, leads in our construction to $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$ being an answer set. Later (b), we show by contradiction that provided $\mathcal{I}$ is an answer set, "just" must be part of $\mathcal{I}_{\text{found}}$.

(a): Due to just $\in \mathcal{I}_{\text{found}}$, also for every rule $r \in \Pi_1$, $x_h \in \mathrm{var}(H_r)$, and $d \in \mathrm{dom}(x)$, it must hold that $\text{just}_{x_h}(d) \in \mathcal{I}_{\text{found}}$, due to Rule (6.18). Furthermore, for all $r \in \Pi_1$, $\text{just}_r \in \mathcal{I}_{\text{found}}$, due to Rule (6.13). Now let's take an arbitrary rule $r \in \Pi_1$. As shown we know $\text{just}_r \in \mathcal{I}_{\text{found}}$. We

[1] Note that $\mathcal{HG}$ has a worst-case grounding size that is in $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{\max_{r\in\Pi} a_h(r)+a_B(r)}\right)$.

take a case distinction, on the two possibilities, which is either either (1) by Rule (6.14), or (2) by Rule (6.17).

If (1) just_r is justified by Rule (6.14), then it is necessary that $\{\text{lit}_{p_1}, \ldots, \text{lit}_{p_n}, \text{lit}_{p_{n+1}}, \ldots, \text{lit}_{p_m}\} \subset \mathcal{I}_{\text{found}}$ hold and therefore Rules (6.15)–(6.16) need to justify them. As shown above, Rule (6.18) justifies all head-variables $\text{just}_{x_h}(d)$. However, in order that Rules (6.15)–(6.16) can justify their heads, their respective body-variable atoms ($\text{just}_x(d)$ s.t. $x \in B_r^+$) need to hold as well, in addition to the body literals (which are by construction in $\mathcal{I}$). The body-variable atoms are derived by Rule (6.12), which necessitates that Rule (6.11) holds, which requires $h'(\mathbf{D})$ to hold. Observe how Rule (6.17) is satisfied by $h'(\boldsymbol{D}) \in \mathcal{I}_{\text{aux}}$. However, that $h'(\mathbf{D})$ holds cannot be required in general, where we go to the other case. (2) By Rule (6.17) it holds that whenever $h'(\mathbf{D}) \notin \mathcal{I}_{\text{aux}}$, and by all head variable atoms being true, then we derive just_r by Rule (6.17). Rules (6.15)–(6.16) are satisfied, as by construction either $\text{lit}_{p_i} \notin \mathcal{I}_{\text{found}}$, one body variable $\text{just}_x(d) \notin \mathcal{I}_{\text{found}}$, or one body literal does not hold. This also ensures that lit_{p_i} are justified. As $h'(\mathbf{D}) \notin \mathcal{I}_{\text{aux}}$, Rule (6.11) is satisfied, and no body-variable $\text{justh}_y(d_i, \boldsymbol{D})$ needs to be justified. Therefore, Rule (6.12) is satisfied and no body variable literal $\text{just}_x(d)$ needs to be justified.

The two possibilities comprise all cases, therefore we follow that "just" being part of $\mathcal{I}_{\text{found}}$, leads in our construction to $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$ being an answer set.

(b): Towards a contradiction, assume that just $\notin \mathcal{I}_{\text{found}}$. By Rule (6.13) we know that there exists a rule r, s.t. $\text{just}_r \notin \mathcal{I}_{\text{found}}$, and therefore by Rules (6.12), and (6.17), their respective bodies cannot hold. Furthermore, due to Rules (6.10)–(6.12), and (6.18), we obtain in this case a single variable assignment in the variable literals just_x.

It needs to hold for this case that h'$(\mathbf{D}) \in \mathcal{I}_{\text{found}}$ (Rule (6.17)), and there is one body literal that falsifies the body (Rules (6.15)–(6.16)). Now there are two cases: (1) h'$(\mathbf{D})$ was derived by Rule (6.1), however, another rule justifies h$(\mathbf{D})$. But then we can remove h'$(\mathbf{D})$ from $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$. (2) h'$(\mathbf{D})$ was derived by Rule (6.1), and no other rule justifies h$(\mathbf{D})$. By construction of $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$ (for each h$(\mathbf{D})$ at least one compatible h'$(\mathbf{D})$), and by the Rules (6.15)–(6.17), this can only be the case when the respective ground rule $r \in \mathcal{G}(\Pi_1)$ (and no other rule) cannot justify h$(\mathbf{D})$. Therefore $\mathcal{I}$ is not an answer set, which contradicts our assumption.

We showed that all Rules (6.1)–(6.19) are satisfied, and we constructed a set of literals $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$ that is justified by $\mathcal{HG}_\mathcal{F}(\Pi_1, \Pi_2)$. Therefore, $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$ is an answer set.

(ii): $\mathcal{HG}_\mathcal{F}(\Pi_1, \Pi_2) \rightarrow \mathcal{G}(\Pi)$
We need to show that every answer set $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$ of $\mathcal{HG}_\mathcal{F}(\Pi_1, \Pi_2)$, is an answer set of $\mathcal{G}(\Pi)$, provided the atoms are restricted to $\text{at}(\mathcal{G}(\Pi))$, so $\mathcal{I} = \mathcal{I}_{\mathcal{HG}_\mathcal{F}} \cap \text{at}(\mathcal{G}(\Pi))$.

Observe that by $\mathcal{HG}_\mathcal{F}$, all rules of Π_2 are grounded via SOTA-grounders, i.e., $\mathcal{G}(\Pi_2)$. Therefore, by the assumption that $\mathcal{I}_{\mathcal{HG}_\mathcal{F}}$ is an answer set, all rules $r \in \mathcal{G}(\Pi_2)$ are satisfied. Further, $r \in \mathcal{G}(\Pi_2)$ justify $\mathcal{J}_{\Pi_2} \subseteq \mathcal{I}$ many atoms.

We still need to show that all rules of $\mathcal{G}(\Pi_1)$ are satisfied, and we still need to justify the atoms of $\mathcal{I} \setminus \mathcal{J}_{\Pi_2}$. However, by [10], [12] we know that the rules in $\mathcal{G}(\Pi_1)$ are satisfied. So we are left with showing justification of the remaining atoms. By construction all atoms $\mathcal{I} \setminus \mathcal{J}_{\Pi_2}$ are justified by $\mathcal{HG}_\mathcal{F}(\Pi_1, \Pi_2)$, but this does not automatically imply justification by $\mathcal{G}(\Pi_1)$.

Towards a contradiction we assume that a h(**D**) is justified by $\mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$ (and therefore by a h'(**D**)), but not by $\mathcal{G}(\Pi_1)$. Knowing h'(**D**) we know the respective non-ground rule r_{ng}. Therfore, we know that r_{ng} cannot justify h(**D**). To show that this is a contradiction, we show that we cannot derive "just".

Observe that Rule (6.17) cannot fire, as by assumption h'(**D**) holds. Therefore, the only possible way to derive just$_r$ is by Rule (6.14), and consequently by Rules (6.15)–(6.16). As an intermediate step, we take an arbitrary variable assignment of the body variables of r_{ng}, i.e., $\mathrm{var}(r) \setminus \mathrm{var}(H_r)$. This correponds to an assignment of the variables in Rules (6.11)–(6.12). By definition this means that either $B_r^+ \not\subseteq \mathcal{I}$, or $B_r^- \cap \mathcal{I} \neq \emptyset$. Observe that any of these variable-assignments lead to a failure of the Rules (6.15)–(6.16). Therefore, by the subset-minimality of disjunctive ASP, w.r.t. to the head-variable assignments (Rule (6.10)), obtain a single head variable assignment, and we fail in obtaining "just". As "just" does not hold, $\mathcal{I}_{\mathcal{HG}_{\mathcal{F}}}$ is not an answer set, which is a contradiction to our initial assumption. Therefore, every atom h(**D**) that is justified by $\mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$ (and therefore by a h'(**D**)), is also justified by $\mathcal{G}(\Pi)$.

As we showed that all rules $r \in \mathcal{G}(\Pi)$ are satisfied, and all atoms $\mathcal{I} = \mathcal{I}_{\mathcal{HG}_{\mathcal{F}}} \cap \mathrm{at}(\mathcal{G}(\Pi))$ are justified, we obtain that $\mathcal{I}$ is indeed an answer set. □

Lemma 3. *Assuming grounding with $\mathcal{HG}_{\mathcal{F}}(\Pi, \emptyset)$, grounding size is in $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{a+1}\right)$, where $a = \max_{r \in \Pi}\{\max_{p(\mathbf{X}) \in r}\{|\mathbf{X}|\}\}$.*

Proof. First observe that we assume that Rules (6.1)–(6.19) are worst-case instantiated, in the sense that for any variable vector $\boldsymbol{X}$, they are instantiated with their respective domains $\boldsymbol{D} \in \mathrm{dom}(\boldsymbol{X})$. This leads to an exponential grounding size in the number of variables of the rewritten procedure ($\mathcal{O}\left(|\Pi| \cdot |\,\mathrm{dom}(\boldsymbol{X})|^{|\boldsymbol{X}|}\right)$). However, with our rewriting procedure, we are able to bound $|\boldsymbol{X}|$ to the maximum arity of the respective rule (and therefore also of the entire program).

Rules (6.1)–(6.9) have already been proven in [10], to have a grounding size of $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{a}\right)$. We are left with showing the result for the Rules (6.10)–(6.19). Observe that for Rules (6.13)–(6.14 6.19) $|\boldsymbol{X}| = 0$, for Rules (6.10, 6.18) $|\boldsymbol{X}| = 1$, for Rules (6.14)–(6.17) $|\boldsymbol{X}| = a$, and for Rules (6.11)–(6.12) $|\boldsymbol{X}| = a + 1$. The dominating factor is therefore $|\boldsymbol{X}| = a + 1$. For an arbitrary rule the grounding size is therefore $\mathcal{O}(|\,\mathrm{dom}(\mathrm{var}(r))|^{|a+1|})$, and for the entire program it is in $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{a+1}\right)$. □

6.2 Extending FastFound to HCF Programs

We extend the FastFound reduction to HCF programs by proposing two encodings. The first encoding directly incorporates disjunctive rules into its guessing, which might be intuitive, but comes with a penalty in grounding size. The second approach uses shifting [74], which was proven to coincide with ASP semantics for HCF programs [34].

6.2.1 Disjunctive Heads

From a complexity perspective, the *answer set existence* problem for HCF programs is Σ_2^P complete, which is the same as for normal programs, Therefore, it was stated in [10] that BDG works in principle for HCF programs. However, previous results did not put an emphasis on this distinction. Therefore, in Figure 6.2 we present the details for using HCF programs. Notice that the auxiliary predicate h' intuitively decouples the disjunctive head from the body and results in the same BDG reduction as for normal (tight) programs.

Glue Π_2 to Π_1 and Ground $\mathcal{G}(\Pi_2)$
Then replace Rules (6.1)–(6.2) with the following:

$$h'(\mathbf{D}) \vee \overline{h'(\mathbf{D})} \leftarrow \qquad \text{for every } r \in \Pi_1, \mathbf{X} = \text{var}(H_r), \mathbf{D} \in \text{dom}(\mathbf{X}) \quad (6.20)$$

$$\bigvee_{h_i \in H_r} h_i(\boldsymbol{D}_{\langle h_i \rangle}) \leftarrow h'(\mathbf{D}) \qquad \text{for every } r \in \Pi_1, \mathbf{X} = \text{var}(H_r), \mathbf{D} \in \text{dom}(\mathbf{X}) \quad (6.21)$$

Satisfiability of Π_1: Same as in Figure 6.1
Foundedness of Π_1: Same as in Figure 6.1

(6.22)

Figure 6.2: Detailed rewriting for HCF programs, for $\mathcal{HG}_{\mathcal{F}}$. The auxiliary head h'decouples the body from the disjunctive head. Therefore, the normal BDG-check suffices, while still enabling the subset-minimality of disjunctive (HCF) heads. Grounding size is in $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{\beta}\right)$, where $\beta = \max_{r \in \Pi}\{|\text{var}(H_r)|, 1 + \max_{p(\mathbf{X}) \in r}\{|\mathbf{X}|\}\}$

Lemma 4. *Given a partition of any non-ground HCF program Π into a program Π_1 and a program Π_2, then answer sets of $\mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$ (disjunctive heads) bijectively match the answer sets of $\mathcal{G}(\Pi)$, restricted to $\text{at}(\mathcal{G}(\Pi))$.*

Proof (high-level-idea). Proving correctness boils down to proving that every answer set $\mathcal{I} \in \mathcal{AS}(\mathcal{G}(\Pi))$ is an answer set of $\mathcal{AS}(\mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2))$, and the other way around. As this was already proven for normal programs in Theorem 6.1, we are left with proving the result for HCF programs. There, it is only necessary to focus on disjunctive rules in Π_1, as the results of Theorem 6.1 carry over to normal rules, and rules in Π_2 are grounded by SOTA techniques.

$\mathcal{G}(\Pi) \rightarrow \mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$
Proving that every answer set $\mathcal{I} \in \mathcal{AS}(\mathcal{G}(\Pi))$ is an answer set of $\mathcal{AS}(\mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2))$, effectively means constructing a set of atoms $\mathcal{I}_{\mathcal{HG}_{\mathcal{F}}}$, and then proving that $\mathcal{I}_{\mathcal{HG}_{\mathcal{F}}}$ is indeed an answer set of $\mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$. The construction of $\mathcal{I}_{\mathcal{HG}_{\mathcal{F}}}$ works similar as for normal rules. For an arbitrary disjunctive rule $r \in \Pi_1$, one has to argue that Rule (6.21) indeed is subset minimal. This corresponds to the additional disjunctive foundedness condition $\mathcal{I} \cap (H_r \setminus \{\text{h}\}) = \emptyset$, so a rule with a disjunctive head can only justify a single head atom. The intuitive idea is to prove that h'is true iff the body of the rule r is true, as then the disjunctive part has exacly the same semantics as the SOTA grounded program.

$\mathcal{G}(\Pi) \leftarrow \mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$
The other direction, i.e., $\mathcal{I}_{\mathcal{HG}_{\mathcal{F}}} \in \mathcal{AS}(\mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2))$, is an answer set of $\mathcal{AS}(\mathcal{G}(\Pi))$ (restricted

to $\mathrm{at}(\mathcal{G}(\Pi))$), boils down to checking that whenever there is a disjunctive rule that justifies a head atom $\mathrm{h}(\mathbf{D})$ in $\mathcal{HG}_{\mathcal{F}}$, the corresponding rule in $\mathcal{G}(\Pi)$ must justify it as well. □

Lemma 5. *Grounding with the novel BDG (HCF disjunctive heads) approach $\mathcal{HG}_{\mathcal{F}}(\Pi, \emptyset)$, results in a grounding size of $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{\beta}\right)$, where $\beta = \max_{r\in\Pi}\{|\operatorname{var}(H_r)|, 1+\max_{p(\mathbf{X})\in r}\{|\mathbf{X}|\}\}$*

Proof (idea). For normal rules we obtain a grounding size of $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{a+1}\right)$, where a is the maximum arity, i.e., $a = \max_{r\in\Pi}\{\max_{p(\mathbf{X})\in r}\{|\mathbf{X}|\}\}$, as shown in Lemma 3.

Observe that for an arbitrary HCF rule $r \in \Pi_1$, Rules (6.20)–(6.21) have $|\boldsymbol{X}| = |\operatorname{var}(H_r)|$. This is the number of variables occurring in the (disjunctive) head of r. As this is potentially larger than the maximum arity, we derive $\beta = \max_{r\in\Pi}\{|\operatorname{var}(H_r)|, 1 + \max_{p(\mathbf{X})\in r}\{|\mathbf{X}|\}\}$, and a grounding size of $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{\beta}\right)$. □

6.2.2 Shifting

The idea of the *shifting* encoding is to transform the disjunctive (HCF) rules into normal rules [74]. In more detail, given a disjunctive (HCF) rule

$$h_1(\boldsymbol{X}_1) \vee \ldots \vee h_l(\boldsymbol{X}_l) \leftarrow p_{l+1}(\boldsymbol{X}_{l+1}), \ldots, p_n(\boldsymbol{X}_n), \neg p_{n+1}(\boldsymbol{X}_{n+1}), \ldots, \neg p_m(\boldsymbol{X}_m)$$

the rewriting generates l-many rules of the form:

$$\begin{aligned} h_i(\boldsymbol{X}_i) \leftarrow & p_{l+1}(\boldsymbol{X}_{l+1}), \ldots, p_n(\boldsymbol{X}_n), \neg p_{n+1}(\boldsymbol{X}_{n+1}), \ldots, \neg p_m(\boldsymbol{X}_m), \\ & \neg h_1(\boldsymbol{X}_1), \ldots, \neg h_{i-1}(\boldsymbol{X}_{i-1}), \neg h_{i+1}(\boldsymbol{X}_{i+1}), \ldots, \neg h_l(\boldsymbol{X}_l) \end{aligned}$$

For l-many head predicates, one generates l-many normal rules. Therefore, one possibility to use BDG for HCF programs is to use the shifting technique in a pre-processing step and call $\mathcal{HG}_{\mathcal{F}}$ (Figure 6.1) in a second-step. However, this has the drawback of duplicated effort in terms of grounding. Therefore, a direct integration into the reduction is preferable. This shifting-integrated reduction is shown in Figure 6.3.

The crucial changes in the reduction are performed in the Rules (6.27), and (6.29). Notably, Rule (6.27) includes head literals (lit_{h_1,h_i}), whereas Rule (6.29) includes $H_r \setminus h_i$. Furthermore, all foundedness-predicates of all rules are (explicitly) indexed by h_i.

Lemma 6. *Given a partition of any non-ground HCF program Π into a program Π_1 and a program Π_2, then answer sets of $\mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$ (shifting) bijectively match the answer sets of $\mathcal{G}(\Pi)$, restricted to $\mathrm{at}(\mathcal{G}(\Pi))$.*

Proof (high-level-idea). It is necessary to prove that each answer set of $\mathcal{G}(\Pi)$ is an answer set of $\mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$, and vice versa.

$\mathcal{G}(\Pi) \rightarrow \mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$
We take an arbitrary answer set $\mathcal{I}$ of $\mathcal{G}(\Pi)$ and extended to $\mathcal{I}^{\mathrm{RED}}_{\mathcal{HG}_{\mathcal{F}}}$, in order to construct an answer

Head guess rules as in Rules (6.1)–(6.3)
Satisfiability of Π_1: Same as Rules (6.4)–(6.9) in Figure 6.1
Foundedness of Π_1

$$\bigvee_{d \in \mathrm{dom}(x)} \mathrm{just}_{h,x}(d) \leftarrow \quad \text{for every } r \in \Pi_1, h(\boldsymbol{X}) \in H_r, x \in \boldsymbol{X} \tag{6.23}$$

$$1\{\mathrm{justh}_{h,y}(d_1, \boldsymbol{D}), \ldots, \mathrm{justh}_{h,y}(d_k, \boldsymbol{D})\}1 \leftarrow h'(\boldsymbol{D}) \quad \text{for every } r \in \Pi_1, h(\boldsymbol{X}) \in H_r, \boldsymbol{D} \in \mathrm{dom}(\boldsymbol{X}), y \in (\mathrm{var}(r) \setminus \boldsymbol{X}), \{d_1, ..., d_k\} = \mathrm{dom}(y) \tag{6.24}$$

$$\mathrm{just}_{h,y}(d) \leftarrow \mathrm{justh}_{h,y}(d, \boldsymbol{D}), \mathrm{just}_{h,x_1}(d_1), \ldots, \mathrm{just}_{h,x_t}(d_t) \quad \text{for every } r \in \Pi_1, h(\boldsymbol{X}) \in H_r, \langle x_1, \ldots, x_t \rangle = \boldsymbol{X}, \langle d_1, \ldots, d_t \rangle = \boldsymbol{D} \in \mathrm{dom}(\boldsymbol{X}), y \in (\mathrm{var}(r) \setminus \mathrm{var}(H_r)), d \in \mathrm{dom}(y) \tag{6.25}$$

$$\mathrm{just} \leftarrow \mathrm{just}_{h_1,r_1}, \ldots, \mathrm{just}_{h_l,r_1}, \mathrm{just}_{h_1,r_n}, \ldots, \mathrm{just}_{h_l,r_n} \quad \text{let } \Pi_1 = \{r_1, \ldots, r_n\}, \{h_1(\boldsymbol{X}_1), \ldots, h_l(\boldsymbol{X}_l)\} = H_r \tag{6.26}$$

$$\mathrm{just}_{h_i,r} \leftarrow \mathrm{lit}_{h_i,p_{l+1}}, \ldots, \mathrm{lit}_{h_i,p_m}, \mathrm{lit}_{h_i,p_{m+1}}, \ldots, \mathrm{lit}_{h_i,p_m} \; \mathrm{lit}_{h_i,h_1}, \ldots, \mathrm{lit}_{h_i,h_{i-1}}, \mathrm{lit}_{h_i,h_{i+1}}, \mathrm{lit}_{h_i,h_l} \quad \text{for every } r \in \Pi_1, h_i \in H_r, H_r = \{h_1, \ldots, h_l\}, B_r^+ = \{p_{l+1}, \ldots, p_m\}, B_r^- = \{p_{m+1}, \ldots, p_n\}, \tag{6.27}$$

$$\mathrm{lit}_{h,p} \leftarrow \mathrm{just}_{h,x_1}(\mathbf{D}_{\langle x_1 \rangle}), \ldots, \mathrm{just}_{h,x_t}(\mathbf{D}_{\langle x_t \rangle}), p(\mathbf{D}) \quad \text{for every } r \in \Pi_1, h \in H_r, p(\mathbf{X}) \in B_r^+, \mathbf{D} \in \mathrm{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle \tag{6.28}$$

$$\mathrm{lit}_{h,p} \leftarrow \mathrm{just}_{h,x_1}(\mathbf{D}_{\langle x_1 \rangle}), \ldots, \mathrm{just}_{h,x_t}(\mathbf{D}_{\langle x_t \rangle}), \neg p(\mathbf{D}) \quad \text{for every } r \in \Pi_1, h \in H_r, p(\mathbf{X}) \in (B_r^- \cup H_r \setminus h), \mathbf{D} \in \mathrm{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle \tag{6.29}$$

$$\mathrm{just}_{h,r} \leftarrow \mathrm{just}_{h,x_1}(\mathbf{D}_{\langle x_1 \rangle}), \ldots, \mathrm{just}_{h,x_t}(\mathbf{D}_{\langle x_t \rangle}), \neg h'(\boldsymbol{D}) \quad \text{for every } r \in \Pi_1, h(\mathbf{X}) \in H_r, \mathbf{D} \in \mathrm{dom}(\mathbf{X}), \mathbf{X} = \langle x_1, \ldots, x_t \rangle \tag{6.30}$$

$$\mathrm{just}_{h,x}(d) \leftarrow \mathrm{just} \quad \text{for every } r \in \Pi_1, h(\boldsymbol{X}) \in H_r, x \in \boldsymbol{X}, d \in \mathrm{dom}(\mathbf{x}) \tag{6.31}$$

$$\leftarrow \neg \mathrm{just} \tag{6.32}$$

Figure 6.3: Alternative foundedness check ($\mathcal{HG}_{\mathcal{F}}$) for HCF programs using the shifting technique. Grounding size is in $\mathcal{O}\left(|\Pi| \cdot |dom(\Pi)|^{a+1}\right)$.

set $\mathcal{I}_{\mathcal{HG}_{\mathcal{F}}}$ of $\mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$ ($\mathcal{I}_{\mathcal{HG}_{\mathcal{F}}} = \mathcal{I} \cup \mathcal{I}^{\mathrm{RED}}_{\mathcal{HG}_{\mathcal{F}}}$). The crucial part is to prove that $\mathcal{I}_{\mathcal{HG}_{\mathcal{F}}}$ is indeed an answer set. However, this endeavor is simplified due to the results of Theorem 6.1. In more detail, this means one can safely ignore normal rules, only needs to check satisfiability of the Rules (6.23)–(6.32), and verify that those atoms in $\mathcal{I}_{\mathcal{HG}_{\mathcal{F}}}$, which are required for Rules (6.23)–(6.32), are actually supported.

(i): $\mathcal{G}(\Pi) \leftarrow \mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$
We take an arbitrary answer set $\mathcal{I}_{\mathcal{HG}_{\mathcal{F}}}$ of $\mathcal{HG}_{\mathcal{F}}(\Pi_1, \Pi_2)$, and check whether $\mathcal{I} = \mathcal{I}_{\mathcal{HG}_{\mathcal{F}}} \cap \mathrm{at}(\mathcal{G}(\Pi))$ is an answer set of $\mathcal{G}(\Pi)$. The (only) critical part in the proof is the check of disjunctive (HCF) rules. On a high level this holds, as by Rules (6.27), and (6.29) only at most one atom is supported per grounded rule (which equals HCF semantics).

After showing both directions one can conclude that the answer sets bijectively match. □

Lemma 7. *Grounding with the novel BDG (HCF shifting) approach $\mathcal{HG}_{\mathcal{F}}(\Pi, \emptyset)$, results in a grounding size of $\mathcal{O}\left(h \cdot |\Pi| \cdot |dom(\Pi)|^{a+1}\right)$, where $h = \max_{r \in \Pi}\{|H_r|\}$ and $a = \max_{r \in \Pi}\{\max_{p(\mathbf{X}) \in r}\{|\mathbf{X}|\}\}$.*

Proof (idea). We obtain the same arities as for $\mathcal{HG}_{\mathcal{F}}$. This is $a = \max_{r \in \Pi}\{\max_{p(\mathbf{X}) \in r}\{|\mathbf{X}|\}\}$, as shown in Lemma 3. However, Rules (6.23)–(6.31) (can) occur $h = \max_{r \in \Pi}\{|H_r|\}$ many times. As h is not a fixed constant (input-dependent) we have to add this constant. □

6.3 Examples

Example 6.1 (Saturation Foundedness Example)**.** *In the following, we use the $\mathcal{HG}_\mathcal{F}$ and $\mathcal{HG}$ rewritings in order to show the differences between those two methods. The example program has 65536 answer sets. Note that for this example SOTA techniques, and $\mathcal{HG}$ are cubic, whereas $\mathcal{HG}_\mathcal{F}$ is quadratic in grounding size. The following is our input program* Π*:*

```
1 e(1..4,1..4).
2 {f(X,Y)} :- e(X,Y).
3 g(X) :- f(X,Y), f(X,Z), f(Y,Z).
```

We rewrite the rule in Line (3) for both $\mathcal{HG}_\mathcal{F}$ and $\mathcal{HG}$. The following listing shows the non-rewritten rules and the satisfiability encoding. Note that the satisfiability encoding is the same between $\mathcal{HG}_\mathcal{F}$ and $\mathcal{HG}$, and therefore shown once. Further note, that due to space limitations we show the non-ground *rewriting.*

With respect to Figure 6.1, Line (2) shows Rule (6.3), Lines (4)–(5) show Rule (6.4), Lines (6)–(8) show Rule (6.8), Lines (9)–(10) show Rules (6.6)–(6.7), and Line (11) shows Rules (6.5), and (6.9).

```
1 %%%%%%%%%%%%%% ORIGINAL-PROGRAM %%%%%%%%%%%%%%
2 e(1..4,1..4). {f(X,Y)}:-e(X,Y).
3 %%%%%%%%%%%%%% SAT-ENCODING %%%%%%%%%%%%%%%%%%%%%%%
4 sat_X(4)|sat_X(2)|sat_X(3)|sat_X(1). sat_Y(4)|sat_Y(2)|sat_Y(3)|sat_Y(1).
5 sat_Z(4)|sat_Z(2)|sat_Z(3)|sat_Z(1).
6 sat_X(1):-sat.sat_X(2):-sat.sat_X(3):-sat.sat_X(4):-sat.
7 sat_Y(4):-sat.sat_Y(2):-sat.sat_Y(3):-sat.sat_Y(1):-sat.
8 sat_Z(4):-sat.sat_Z(2):-sat.sat_Z(3):-sat.sat_Z(1):-sat.
9 sat_r:-sat_X(X1),g(X1). sat_r:-sat_X(X1),sat_Y(X2),not f(X1,X2).
10 sat_r:-sat_X(X1),sat_Z(X2),not f(X1,X2). sat_r:-sat_Y(X1),sat_Z(X2),not f(X1,X2).
11 :- not sat. sat:-sat_r.
```

The following listing depicts the rewriting with $\mathcal{HG}$. Observe Line (8)*, where the corresponding rule has* 3 variables*. Therefore, the grounding size is cubic of $\mathcal{HG}$. Line (2) shows the Rules (4.1)–(4.2), Lines (4)–(5) show the Rule (4.11), Lines (6)–(8) show the Rules (4.12)–(4.13), and Line (9) shows the Rule (4.14).*

```
1 %%%%%%%%%%%%%% HEAD-GUESS %%%%%%%%%%%%%%%%%%%%%%%
2 {g'(4);g'(2);g'(3);g'(1)}. g(X1):-g'(X1).
3 %%%%%%%%%%%%%% UFOUND %%%%%%%%%%%%%%%%%%%%%%%%%%%%
4 1{ujust_Y(4,X);ujust_Y(2,X);ujust_Y(3,X);ujust_Y(1,X)}1:-g'(X).
5 1{ujust_Z(4,X);ujust_Z(2,X);ujust_Z(3,X);ujust_Z(1,X)}1:-g'(X).
6 ujust_X(X1):-ujust_Y(X2,X1),not f(X1,X2).
7 ujust_X(X1):-ujust_Z(X2,X1),not f(X1,X2).
8 ujust_X(X1):-ujust_Y(X2,X1),ujust_Z(X3,X1),not f(X2,X3).
9 :-0 < #count{1:ujust_X(X1)},g'(X1).
```

The $\mathcal{HG}_\mathcal{F}$ rewriting is displayed in the listing below. Note that all rules have at maximum two variables that need to be grounded. Line (2) shows Rules (6.1)–(6.2), Line (4) shows the Rule (6.10), Line (5) shows the Rule (6.18), Lines (8)–(9) show the Rule (6.11), Line (10) shows the Rule (6.12), Lines (12)–(13) show the Rules (6.15)–(6.17), and Line (14) shows the Rules (6.13), (6.14), and (6.19).

```
1  %%%%%%%%%%%%% HEAD-GUESS %%%%%%%%%%%%%%%%%%%%%%
2  {g'(2);g'(1);g'(4);g'(3)}. g(X1):-g'(X1).
3  %%%%%%%%%%%%% FOUND-START %%%%%%%%%%%%%%%%%%%%%
4  %%%%%%%%%%%%% HEAD-VARIABLES %%%%%%%%%%%%%%%%%%
5  just_X(2)|just_X(1)|just_X(4)|just_X(3).
6  just_X(2):-just.just_X(1):-just.just_X(4):-just.just_X(3):-just.
7  %%%%%%%%%%%%% NON-HEAD-VARIABLES %%%%%%%%%%%%%%
8  1{just_h_Y(2,X1);just_h_Y(1,X1);just_h_Y(4,X1);just_h_Y(3,X1)}1 :- g'(X1).
9  1{just_h_Z(2,X1);just_h_Z(1,X1);just_h_Z(4,X1);just_h_Z(3,X1)}1 :- g'(X1).
10 just_Y(Y):-just_h_Y(Y,X1),just_X(X1). just_Z(Y):-just_h_Z(Y,X1),just_X(X1).
11 %%%%%%%%%%%%% JUST-LITERALS %%%%%%%%%%%%%%%%%%%
12 just_r:-just_X(X1),not g'(X1). just_1:-just_X(X1),just_Y(X2),f(X1,X2).
13 just_2:-just_X(X1),just_Z(X2),f(X1,X2). just_3:-just_Y(X1),just_Z(X2),f(X1,X2).
14 just:-just_r. just_r:-just_1,just_2,just_3. :- not just.
```

Example 6.2 (HCF Example)**.** *The following listing shows our example input program* Π*. We demonstrate the rewritings for* $\mathcal{HG}$ *and* $\mathcal{HG}_{\mathcal{F}}$ *(both disjunctive-head and shift encoding). Line (2) contain the rules that shall be grounded by SOTA techniques, while in Line (4) are the rules that shall be rewritten. The encoding is for the sole purpose to show, how to encode HCF programs. SOTA techniques are linear in their grounding size for* Π*. The program has 16 answer sets. Also note the two disjunctive rules, with the shared head predicate* $g(X)$*.*

```
1  %%%%%%%%% ORIGINAL-ENCODING %%%%%%%%%%%%%
2  e(1). e(2). {h(X)} :- e(X). g(X) | k(X) :- e(X).
3  %%%% REWRITE THE FOLLOWING RULE: %%%%
4  f(X) | g(X) :- h(X).
```

The next listing shows the shared parts of all reductions. Line (2) stems from Rule (4.3), which is the SOTA grounded part. The remaining lines show the satisfiability encoded part. In more detail, Line (4) resembles Rules (4.4), and (4.9), Line (5) shows Rules (4.6)–(4.8), and Line (6) shows Rules (4.5), and (4.10).

```
1  %%%%%%%%%%% SHARED-ENCODING %%%%%%%%%%%
2  %%%%%%%%% SOTA-GROUNDED-PART %%%%%%%%%%
3  e(1).e(2). {h(X)}:-e(X). g(X) | k(X) :- e(X).
4  %%%%%%%%%%%%%% SAT-CHECK %%%%%%%%%%%%%%%
5  sat_X(1):-sat. sat_X(2):-sat. sat_X(1)|sat_X(2).
6  sat_r:-sat_X(X1),f(X1). sat_r:-sat_X(X1),g(X1).  sat_r:-sat_X(X1),not h(X1).
7  sat:-sat_r. :- not sat.
```

Our first rewriting is the guess rewriting (Figure 4.2). Line (2) shows Rules (4.1), and (4.2), Lines (4) and (6) show Rules (4.12)–(4.13), whereas Lines (5) and (7) show Rules (4.14). Note that Rule (4.11) is not needed, as all variables in the body occur also in the head of the original rule. Further, observe that Lines (4)–(5) encode a guessed f'*, whereas Lines (6)–(7) encode a guessed* g'*.*

```
1  %%%%%%%%%%% GUESS-ENCODING %%%%%%%%%%%%%%
2  %%%%%%%%%%%%%% Head Guess %%%%%%%%%%%%%%%
3  {f'(1);f'(2)}. {g'(1);g'(2)}. f(X1):-f'(X1). g(X1):-g'(X1).
4  %%%%%%%%%%%%%% UFOUND CHECK %%%%%%%%%%%%%%
5  ujust_0_0_X(X1):-f'(X1),not h(X1). ujust_0_0_X(X1):-f'(X1),g(X1).
6  :-0 < #count{1:ujust_0_0_X(X1)},f'(X1).
7  ujust_0_1_X(X1):-g'(X1),not h(X1). ujust_0_1_X(X1):-g'(X1),f(X1).
8  :-0 < #count{1:ujust_0_1_X(X1)},g'(X1).
```

The next discussed rewriting is the saturation disjunctive head encoding (Figure 6.2). Line (3) resembles Rules (6.20) and (6.21), which contain the eponymous disjunctive head. Line (5) are Rules (6.10) and (6.18). Lines (6)–(7) are Rules (6.15)–(6.17). Finally Line (9) was generated by Rules (6.13), (6.14), and (6.19).

```
1 %%%%%%%%% DISJ.-HEAD-ENCODING %%%%%%%%%%%
2 %%%%%%%%%%%%%% Head Guess %%%%%%%%%%%%%%%
3 {head'(1);head'(2)}. f(X) | g(X) :- head'(X).
4 %%%%%%%%%%%%%% FOUND CHECK %%%%%%%%%%%%%%%
5 just_X(1)|just_X(2). just_X(1):-just. just_X(2):-just.
6 just_0:-just_X(X1),not head'(X1).
7 just_0_1:-just_X(X1),h(X1).
8 %%%%%%%%%%%%%% OVERALL %%%%%%%%%%%%%%%%%%%
9 just:-just_0. :- not just. just_0:-just_0_1.
```

The last discussed rewriting is the shift encoding (Figure 6.3). Line (3) was generated by Rules (6.1)–(6.2), Lines (6) and (10) were generated by Rule (6.23), Lines (7)–(8), and (11)–(12) were generated by Rules (6.28)–(6.30), and Lines (14)–(15) were generated by Rules (6.26), (6.27), and (6.32).

```
1 %%%%%%%%%%% SHIFT-ENCODING %%%%%%%%%%%%
2 %%%%%%%%%%%%%% HEAD-GUESS %%%%%%%%%%%%%%
3 {f'(1);f'(2)}.{g'(1);g'(2)}. f(X1):-f'(X1). g(X1):-g'(X1).
4 %%%%%%%%%%%%%% FOUND-CHECK %%%%%%%%%%%%%
5 %%%%%% For f(x):
6 just_0_X(1)|just_0_X(2). just_0_X(1):-just. just_0_X(2):-just.
7 just_0:-just_0_X(X1),not f'(X1).
8 just_0_1:-just_0_X(X1),h(X1). just_0_2:-just_0_X(X1),not g(X1).
9 %%%%%% For g(X):
10 just_1_X(1)|just_1_X(2). just_1_X(1):-just. just_1_X(2):-just.
11 just_1:-just_1_X(X1),not g'(X1).
12 just_1_1:-just_1_X(X1),h(X1). just_1_2:-just_1_X(X1),not f(X1).
13 %%%%%%%%%%%%%% OVERALL %%%%%%%%%%%%%%%%%
14 just_0:-just_0_1, just_0_2. just_1:-just_1_1, just_1_2.
15 just:-just_0, just_1. :- not just.
```

6.4 Experiments

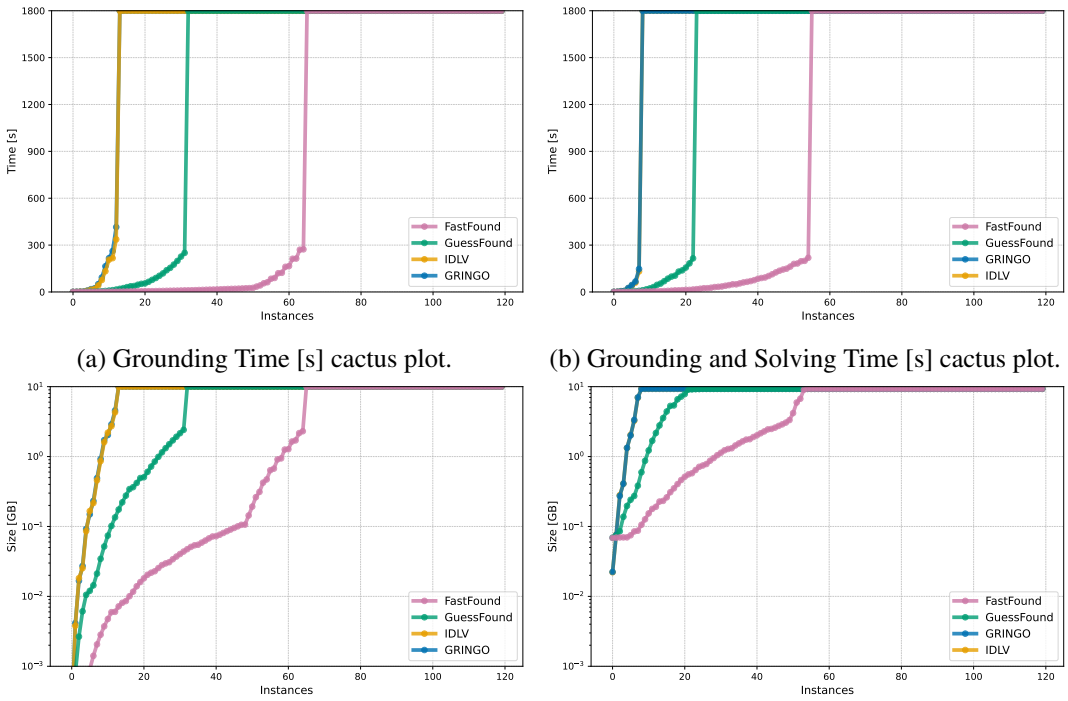
To demonstrate the usefulness of $\mathcal{HG}_{\mathcal{F}}$ we conduct a series of experiments. Note that the benchmark system and overall setup is the same as in Section 5.4. We integrated $\mathcal{HG}_{\mathcal{F}}$ into our `newground3` prototype. Regarding the experimental setup, we compared SOTA ground-and-solve systems to $\mathcal{HG}_{\mathcal{F}}$ and $\mathcal{HG}$. For the other details, we refer to Section 5.4.

6.4.1 Benchmark Scenarios

We restrict our experiment setup to demonstrating the viability of $\mathcal{HG}_{\mathcal{F}}$, in comparison to $\mathcal{HG}$ and other SOTA techniques. Further, we want to emphasize that the heuristic splitting algorithms from Chapter 5 can be adapted to suit $\mathcal{HG}_{\mathcal{F}}$, which leads to an efficient integration of $\mathcal{HG}_{\mathcal{F}}$ into SOTA techniques.

The setting of our benchmarks is in graph space, to be more specific in sub-graph space. We ask whether there exists a sub-graph of a graph, with one of three different properties (therefore, three scenarios). We generated complete graphs of sizes ranging from 10 to 400 (40 instances per scenario, step size of 10).

Our first benchmark scenario *Four-Clique* consists of limiting the number of vertices that occur in a four-clique to at-most N, where we set $N = 20$ for our benchmarks. We show in the following listing our example encoding for scenario *Four-Clique*. Observe that while this encoding is $\approx |\text{dom}|^4$ for SOTA methods, it is $\approx |\text{dom}|^3$ for $\mathcal{HG}$, and $\approx |\text{dom}|^2$ for $\mathcal{HG}_{\mathcal{F}}$:



(a) Grounding Time [s] cactus plot.

(b) Grounding and Solving Time [s] cactus plot.

(c) Grounding Size [GB] cactus plot.

(d) Max RAM usage [GB] cactus plot.

Figure 6.4: Cactus plots demonstrating viability of FastFound. Compared `gringo`, `idlv`, $\mathcal{HG}$ (GuessFound), and $\mathcal{HG}_{\mathcal{F}}$ (FastFound). $\mathcal{HG}_{\mathcal{F}}$ grounds and solves many more instances, than `gringo`, `idlv` or $\mathcal{HG}$. Left side (Figures 6.4a and 6.4c) was measured during grounding, right side (Figures 6.4b and 6.4d) was measured during grounding and solving. Timeout: 1800s; Memout: 10GB.

```
{f(X,Y)} :- edge(X,Y).
:- #count{X:c(X)}>20.
% The following rule is grounded with the novel foundedness approach:
c(X):-f(X,X1),f(X,X2),f(X,X3),f(X1,X2),f(X1,X3),f(X2,X3).
```

The second and third scenarios consist of finding a k clique in a hypergraph, where k is six for the second (*Hyper-Six-Clique*) and k is seven for the third (*Hyper-Seven-Clique*) scenario. Additionally, we infer every two first vertices occurring in the first hyper-edge. Observe that this encoding is $\approx |\text{dom}|^6$ for SOTA methods, $\approx |\text{dom}|^5$ for $\mathcal{HG}$, and $\approx |\text{dom}|^3$ for $\mathcal{HG}_{\mathcal{F}}$:

```
{f(X,Y,Z)} :- edge(X,Y), edge(Y,Z).
d(X,Y) :- f(X,Y,Z).
% The following rule is grounded with the novel foundedness approach:
c(X1,X2) :- f(X1,X2,X3), f(X4,X5,X6), f(X1,X4,X5), f(X2,X4,X5), f(X3,X6,X5),
    f(X6,X1,X2), f(X3,X4,X5), d(X1,X2).
```

The third scenario (*Hyper-Seven-Clique*) is shown in the listing below. Observe that this encoding is $\approx |\text{dom}|^7$ for SOTA methods, $\approx |\text{dom}|^5$ for $\mathcal{HG}$, and $\approx |\text{dom}|^3$ for $\mathcal{HG}_{\mathcal{F}}$:

```
{f(X,Y,Z)} :- edge(X,Y), edge(Y,Z).
d(X,Y) :- f(X,Y,Z).
% The following rule is grounded with the novel foundedness approach:
c(X1,X2) :- f(X1,X2,X3), f(X4,X5,X6), f(X1,X4,X5), f(X2,X4,X5), f(X3,X6,X5),
    f(X6,X1,X2), f(X3,X4,X5), f(X7,X1,X6), f(X7,X2,X3), f(X7,X4,X5), d(X1,X2)
        .
```

6.4.2 Hypotheses

We study the following hypotheses:

(H1) $\mathcal{HG}_{\mathcal{F}}$ can reduce the overall grounding and solving time, compared to SOTA methods, and $\mathcal{HG}$.

(H2) $\mathcal{HG}_{\mathcal{F}}$ can reduce the maximum memory usage of both the grounder and the solver, compared to SOTA methods, and $\mathcal{HG}$.

6.4.3 Experimental Results

We show cactus plots of our results in Figure 6.4. Observe the reduced grounding time (Figure 6.4a) and size (Figure 6.4c) of the *Saturation* ($\mathcal{HG}_{\mathcal{F}}$) technique. Further, observe the reduced overall grounding and solving time (Figure 6.4b), and the reduced maximum memory usage of both the grounder and solver (Figure 6.4d). Therefore, hypothesis H2 can be confirmed. Finally, in Table 6.1 we show the number of grounded and solved instances, where the $\mathcal{HG}_{\mathcal{F}}$ (Saturation) manages to solve most instances. We therefore can confirm H1 as well.

These results lead us to state that the theoretically reduced grounding size by $\mathcal{HG}_{\mathcal{F}}$, can have an effect on the number of solved instances.

Scen.	#Insts.	Total #Solved											
		gringo			idlv			$\mathcal{HG}$			$\mathcal{HG}_{\mathcal{F}}$		
		#S	M	T	#S	M	T	#S	M	T	#S	M	T
Total-SUM	120	8	25	87	8	8	104	22	81	17	**55**	50	15
Four-Clique	40	6	5	29	6	5	29	18	15	7	**40**	0	0
Hyper-Six-Clique	40	1	11	28	1	2	37	2	35	3	**8**	26	6
Hyper-Seven-Clique	40	1	9	30	1	1	38	2	31	7	**7**	24	9

Table 6.1: FastFound ($\mathcal{HG}_{\mathcal{F}}$) can yield a performance improvement for grounding-heavy benchmarks for both grounding and solving. We report number of solved instances on three synthetic datasets. Benchmarked systems include `gringo`, `idlv`, BDG with guess-foundedness ($\mathcal{HG}$), and BDG with saturation-foundedness ($\mathcal{HG}_{\mathcal{F}}$). Measured number of solved instances (#S), memouts (M), timeouts (T). Timeout of 1800s, Memout of 10GB.

Scen.	#Insts.	Total #Grounded											
		gringo			idlv			$\mathcal{HG}$			$\mathcal{HG}_{\mathcal{F}}$		
		#G	M	T	#G	M	T	#G	M	T	#G	M	T
Total-SUM	120	13	25	82	13	7	100	32	77	11	**65**	39	16
Four-Clique	40	10	5	25	10	5	25	28	9	3	**40**	0	0
Hyper-Six-Clique	40	2	10	28	2	1	37	2	35	3	**13**	18	9
Hyper-Seven-Clique	40	1	10	29	1	1	38	2	33	5	**12**	21	7

Table 6.2: FastFound ($\mathcal{HG}_{\mathcal{F}}$) can yield a performance improvement for grounding-heavy benchmarks for grounding. We report number of solved instances on three synthetic datasets. Benchmarked systems include `gringo`, `idlv`, BDG with guess-foundedness ($\mathcal{HG}$), and BDG with saturation-foundedness ($\mathcal{HG}_{\mathcal{F}}$). Measured number of grounded instances (#G), memouts (M), timeouts (T). Timeout of 1800s, Memout of 10GB.

CHAPTER 7

Lazy-BDG: Using Propagators for Cyclic Programs

Lazy-BDG demonstrates the efficient integration of body-decoupled grounding with lazy-grounding for cyclic rules. So far the current SOTA technique for handling cycles in BDG is the *level mappings* [12] technique, which was later extended to hybrid grounding [10]. Intuitively, level mappings create an order on the derivation sequence of atoms. However, as level mappings are written as non-ground rules and they have to cope with transitive relations, their grounding size is cubic in the maximum arity of any predicate of the program. This prevents its usage for practical problems. It is well known that cycles are a challenge for BDG, as discussed in [127], where they introduced the *ordered derivations method*. But still, in their experiments both level mappings and ordered derivations performed worse than `gringo`.

Therefore, the question arises what else can be done to tackle the cycle challenge for BDG. This chapter offers a solution with *Lazy-BDG*, by combining body-decoupled grounding, lazy-grounding, and the state-of-the-art solver technique *unfound-set method* [63] (see Section 2.5.2). However, this *unfound-set method* cannot be used directly, as it requires the original (grounded) structure of the program. This information is lost when using BDG. Therefore, we propose a 2-step approach, where we use special rewritings together with an *adapted unfound-set* algorithm. The adapted unfound-set algorithm is implemented as a *propagator* [58] in our prototypical implementation and injects appropriate *nogoods* in the CDNL algorithm (see also Section 2.5). Our experiments demonstrate that this approach is promising, as we were able to beat the level-mappings technique, `gringo`, and `idlv` on grounding-heavy synthetic scenarios.

We start by discussing why cycles are a problem for BDG (Section 7.1), continue to present the state-of-the-art level mappings method with hybrid grounding (Section 7.2), and by reviving cycles for hybrid grounding (Section 7.3). Then we present our lazy-approach to the dependency graph repair in Section 7.4. We close the chapter by showing our experimental results in Section 7.5.

Note: For the remainder of this chapter we will, except if explicitly stated otherwise, not distinguish between $\mathcal{HG}_\mathcal{F}$ (Figure 6.1) and $\mathcal{HG}$ (Figure 4.2), as cyclic information is lost in both methods in the same way. Examples will mostly be given by the $\mathcal{HG}$ approach.

7.1 How BDG dismantles Cycles

We demonstrate by an example, how BDG *dismantles* cycles.

Example 7.1. *In the following we will observe how different the dependency graphs for programs grounded by SOTA methods and programs grounded by BDG techniques look like. Let us consider the cyclic program Π in the listing below. We show in Figure 7.1a its dependency graph. Grounding and solving it yields two answer sets: $\{\{e(1,2)\}; \{e(1,2); f(1,2); f(2,1); q(1,2); q(2,1)\}\}$.*

```
1 e(1,2). {f(X,Y)} :- e(X,Y).
2 q(X,Y) :- f(X,Y). f(X,Y) :- q(Y,X).
```

The grounding of the program is shown in the next listing and its dependency graph in Figure 7.1c. Observe the difference between the non-ground and the ground dependency graph. While the length of the cycle of the non-ground dependency graph was 2, the length of the ground dependency graph is 4. Further, the fact $e(1,2)$ was removed from the dependency graph, which is expected as the bottom-up SOTA grounder can evaluate stratified programs (as detailed in Chapter 5.1.1).

```
1 e(1,2). {f(1,2)}.
2 q(1,2):-f(1,2). f(2,1):-q(1,2).
3 q(2,1):-f(2,1). f(1,2):-q(2,1).
```

Now let us use the BDG reduction, as shown in Figure 4.2. Let $\Pi_1 = \{q(X,Y) \leftarrow f(X,Y); f(X,Y) \leftarrow q(Y,X)\}$ and $\Pi_2 = \{e(1,2); \{f(X,y)\} \leftarrow e(X,Y)\}$, with $\mathcal{HG}(\Pi_1, \Pi_2)$.

In the next listing we show $\mathcal{HG}(\Pi_1, \Pi_2)$ without grounding where we restrict it to $\mathrm{at}(\mathcal{G}(\Pi)) \cup \{h' | h \in H(\Pi_1)\}$ (i.e., without auxiliary atoms, which is shown in the Appendix). We show the dependency graph in Figure 7.1b. Observe that there is no connection between f and q.

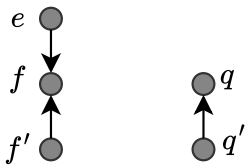
```
1 e(1,2). {f(X,Y)}:-e(X,Y).
2 {f'(1,1);f'(2,1);f'(1,2);f'(2,2)}.
3 {q'(1,1);q'(2,1);q'(1,2);q'(2,2)}.
4 q(X1,X2):-q'(X1,X2). f(X1,X2):-f'(X1,X2).
```

In the next listing we show the grounded version of $\mathcal{HG}(\Pi_1, \Pi_2)$, and in Figure 7.1d its dependency graph. Observe that as in the grounding $\mathcal{G}(\Pi)$, there is no $e(1,2)$ present, as this was handled in the SOTA grounding of Π_2. Further, observe that there are no connections between any f or q atoms, and that the grounding encompasses the full grounding.

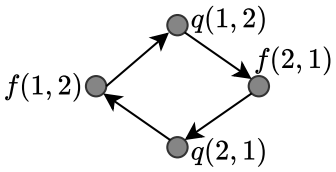
Solving this program (incorrectly) yields 8 answer sets (projected to $\mathrm{at}(\mathcal{G}(\Pi))$). The reason for this is that the dependency graph of $\mathcal{HG}$ is non-cyclic, although the dependency graph of $\mathcal{G}(\Pi)$ is cyclic. Therefore, SOTA solvers cannot infer cyclic information, which leads to the failure in solving.

```
1 e(1,2). {f(1,2)}.
2 {q'(1,1);q'(2,1);q'(1,2);q'(2,2)}.
3 q(1,1):-q'(1,1). q(2,1):-q'(2,1).
```

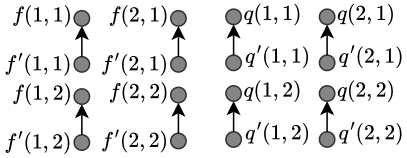
(a) Dependency graph of Π.



(b) Dependency graph of (non-ground) $\mathcal{HG}(\Pi_1, \Pi_2)$



(c) Dependency graph of $\mathcal{G}(\Pi)$.



(d) Dependency graph of (ground) $\mathcal{HG}(\Pi_1, \Pi_2)$

Figure 7.1: Dependency graphs of Example 7.1. Grounding a non-ground program Π yields different cycle lengths (Figures 7.1a, and 7.1c). Using $\mathcal{HG}$ dismantles the cyclicity (Figures 7.1a, and 7.1b). This also holds for the ground case (Figure 7.1d). Therefore, using $\mathcal{HG}$ for grounding, and solving without taking additional steps (like using level mappings, or lazy-bdg) leads to incorrect answer sets.

```
4 q(1,2):-q'(1,2). q(2,2):-q'(2,2).
5 {f'(1,1);f'(2,1);f'(1,2);f'(2,2)}.
6 f(1,1):-f'(1,1). f(2,1):-f'(2,1).
7 f(1,2):-f'(1,2). f(2,2):-f'(2,2).
```

Summing up the above groundings and dependency graphs, we observe that BDG dismantles the cycles and thereby loses the cyclic information. Therefore, SOTA solvers cannot infer cyclic parts, which leads to a failure in solving. The major conclusion we derive is that cycles have to be treated differently in BDG. Possible ideas range from level mappings, over non-ground re-creation of the dependency graph, to lazily inferring them in the solving phase.

7.2 Preventing Cyclic Derivations via Level Mappings

Level mappings treat the problem of the dismantled cycles by going back to the definition of foundedness. It is the current state-of-the-art BDG method how to tackle the cycle challenge. Intuitively, level mappings do not revive the cycles, but define a derivation order, which stems from the *foundedness* definition (see Definition 2.9 for details): In addition to a rule that is

Replacing Rules (4.3) for (Shared) SCCs by:

$\mathcal{G}(\{r'\})$ — for every $r' \in \Pi_2$, with $\forall h \in H_{r'}$: $S = SCC(\Pi, h), rules(S) \cap \Pi_1 = \emptyset$ (7.1)

$a \leftarrow a_{l+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n, (a_{l+1} \prec a), \ldots, (a_m \prec a)$ — for every $r' \in \Pi_2$, with $\exists h \in H_{r'} : S = SCC(\Pi, h)$ $rules(S) \cap \Pi_1 \neq \emptyset, r \in \mathcal{G}(\{r'\}), H_r = \{a\}$, $B_r^+ = \{a_{l+1}, \ldots, a_m\}, B_r^- = \{a_{m+1}, \ldots, a_n\}$ (7.2)

$\leftarrow a_{l+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n, \neg a$ — for every $r' \in \Pi_2$, with $\exists h \in H_{r'} : S = SCC(\Pi, h)$ $rules(S) \cap \Pi_1 \neq \emptyset, r \in \mathcal{G}(\{r'\}), H_r = \{a\}$, $B_r^+ = \{a_{l+1}, \ldots, a_m\}, B_r^- = \{a_{m+1}, \ldots, a_n\}$ (7.3)

Add. Rules for Found. of SCCs:

$[p_1(\mathbf{D}_1) \prec p_2(\mathbf{D}_2)] \vee [p_2(\mathbf{D}_2) \prec p_1(\mathbf{D}_1)] \leftarrow$ — for every SCC $S \in \mathrm{SCC}(\Pi)$ with $rules(S) \cap \Pi_1 \neq \emptyset$, $p_1(\mathbf{X}_1), p_2(\mathbf{X}_2) \in S, \mathbf{D}_1 \in \mathrm{dom}(\mathbf{X}_1)$, $\mathbf{D}_2 \in \mathrm{dom}(\mathbf{X}_2), p_1(\mathbf{D}_1) \neq p_2(\mathbf{D}_2)$ (7.4)

$\leftarrow [p_1(\mathbf{D}_1) \prec p_2(\mathbf{D}_2)], [p_2(\mathbf{D}_2) \prec p_3(\mathbf{D}_3)], [p_3(\mathbf{D}_3) \prec p_1(\mathbf{D}_1)]$ — for every SCC $S \in \mathrm{SCC}(\Pi)$ with $rules(S) \cap \Pi_1 \neq \emptyset$, $p_1(\mathbf{X}_1), p_2(\mathbf{X}_2), p_3(\mathbf{X}_3) \in S, \mathbf{D}_1 \in \mathrm{dom}(\mathbf{X}_1)$, $\mathbf{D}_2 \in \mathrm{dom}(\mathbf{X}_2), \mathbf{D}_3 \in \mathrm{dom}(\mathbf{X}_3), p_1(\mathbf{D}_1) \neq p_2(\mathbf{D}_2)$, $p_2(\mathbf{D}_2) \neq p_3(\mathbf{D}_3), p_1(\mathbf{D}_1) \neq p_3(\mathbf{D}_3)$ (7.5)

$\mathrm{uf}_r(\mathbf{D_X}) \leftarrow \mathrm{uf}_{y_1}(\mathbf{D}_{\langle \mathbf{X}, y_1 \rangle}), \ldots, \mathrm{uf}_{y_\ell}(\mathbf{D}_{\langle \mathbf{X}, y_\ell \rangle}), \neg[p(\mathbf{D}_Y) \prec h_r(D_\mathbf{X})]$ — for every SCC $S \in \mathrm{SCC}(\Pi)$ with $r \in rules(S) \cap \Pi_1$, $\{h(\mathbf{X})\} = H_r, p(\mathbf{Y}) \in B_r^+, \mathbf{D} \in \mathrm{dom}(\langle \mathbf{X}, \mathbf{Y} \rangle)$, $\mathbf{Y} = \langle y_1, \ldots, y_\ell \rangle, p(\mathbf{D_Y}) \notin \mathcal{F}$ (7.6)

$\mathrm{uf}_{r_r}(\mathbf{D_X}) \leftarrow \neg[h_r(\mathbf{D}_Y) \prec h(D_\mathbf{X})]$ — for every SCC $S \in \mathrm{SCC}(\Pi)$ with $r \in rules(S) \cap \Pi_1$, $\{h(\mathbf{X})\} = H_r, p(\mathbf{Y}) \in B_r^+, \mathbf{D} \in \mathrm{dom}(\langle \mathbf{X}, \mathbf{Y} \rangle)$, $\mathbf{Y} = \langle y_1, \ldots, y_\ell \rangle, p(\mathbf{D_Y}) \notin \mathcal{F}$ (7.7)

Replace Rules (4.14) with:

$\leftarrow \mathrm{uf}_r(\mathbf{D}), h_r(\mathbf{D})$ — for every $h(\mathbf{X}) \in \mathrm{hpred}(\Pi_1), \mathbf{D} \in \mathrm{dom}(\mathbf{X})$, $\{r_1, \ldots, r_m\} = \{r \mid r \in \Pi_1, \{h(\mathbf{X})\} = H_r\}$ (7.8)

$\leftarrow \mathrm{uf}_{r_r}(\mathbf{D}), h_r(\mathbf{D})$ — for every $h(\mathbf{X}) \in \mathrm{hpred}(\Pi_1), \mathbf{D} \in \mathrm{dom}(\mathbf{X})$, $\{r_1, \ldots, r_m\} = \{r \mid r \in \Pi_1, \{h(\mathbf{X})\} = H_r\}$ (7.9)

Figure 7.2: Erroneous cyclic derivations are prevented by an explicit guess of a derivation order. Let Π be a non-ground non-tight normal program, and Π_1, Π_2 be a partition thereof. $\mathcal{H}_{lv}(\Pi_1, \Pi_2)$ is the Hybrid Grounding procedure with level mappings. This is an extension of the Hybrid Grounding procedure for tight (HCF) programs (Figure 4.2). Let $S \in SCC(\Pi)$, then $rules(S) = \{r \mid r \in \Pi, H_r \cap S \neq \emptyset, B_r^+ \cap S \neq \emptyset\}$.

suitable for justifying an atom, there must exist a function ϕ, s.t. $\phi : I \to \{0, ..., |I| - 1\}$, and for all $p \in B_r^+$ it must be the case that $\phi(p) < \phi(h)$. Intuitively this means that the (positive) body of r is derived before the head. This section was adapted from the supplementary material of [10].

Level mappings create such an order, by comparing two elements at a time. One can view level mappings as a strict total order $\prec$: $\mathcal{I}^2 \to \{0, 1\}$, s.t. for all $x, y, z \in \mathcal{I}$ it holds (1) $0 = x \prec x$, (2) if $1 = x \prec y$ then $0 = y \prec x$, (3) if $1 = x \prec y$ and $1 = y \prec z$, then also $1 = x \prec z$, and (4) if $x \neq y$ then $1 = x \prec y$ or $1 = y \prec x$.

So intuitively, while ϕ explicitly assigns numbers to identify a suitable derivation order, level mappings define a precedence "$\prec$" operator over atoms.

For practical purposes in ASP, the main drawbacks of this approach are twofold: First, we have to guess an ordering for every pair of atoms (thereby slowing down the solver). And second, we have to check the transitivity relation. As transitivity relates 3 atoms, this leads to a massive

grounding problem.

We show the hybrid grounding reduction $\mathcal{H}_{lv}$ for non-ground non-tight normal programs in Figure 7.2. Rules that are grounded by traditional procedures and are not part of a shared cycle are grounded without any additional rewritings (Equation (7.1)). If there is a shared cycle, we prevent cyclic derivations (Equation (7.2)) and ensure satisfiability (Equation (7.3)). We guess an explicit derivation order via level mappings (Equation (7.4)) and prevent non-transitive derivation orders in Equation (7.5). Rules part of Π_1 are adapted to prevent cyclic derivations (Equation (7.6)) and prevent cyclic derivations of the auxiliary rules (Equation (7.7)). The constraints in Equations (7.8)–(7.9) ensure that no rule, or auxiliary rule is unfounded.

Notice Rule (7.5) which has a grounding size of $\mathcal{O}\left(|\Pi| \cdot |\operatorname{dom}(\Pi)|^{3 \cdot a}\right)$, where a is the maximum arity of the program. Due to this, the usage of level mappings is not practical for most applications.

7.3 Reviving Cycles: Dependency Graph Repair

The reviving cycles approach is intuitively simple: Whenever BDG grounds parts of a non-trivial SCC, we prevent erroneous cyclic derivations by reviving the cycle, s.t. the solver-inherent cycle handling mechanisms can detect and prevent cyclic derivations. We do this by adding those body predicates to the head guess that are in the same SCC as the head. When comparing it to level mappings, its main difference can be seen in the dependency graphs. While in Figure 7.1b there is no cycle, in Figure 7.3 there is one. We depict the reduction in Figure 7.4, which takes a non-ground non-tight normal program as in input.

In more detail, Rule (7.10) is used for trivial SCCs. Rule (7.11) is used for the non-trivial SCCs. For each such rule, we gather the set of positive body predicates that are in the same SCC as the head predicate $H_r = \{h\}$: $\boldsymbol{P}{:=}\{p(\boldsymbol{Y}) | p(\boldsymbol{Y}) \in B_r^+, \text{SCC}(\Pi, h) = \text{SCC}(\Pi, p)\}$. These are then added to the body of the head guess.

Example 7.2 (Ctd. of Example 7.1)**.** *Using the adapted hybrid grounding procedure of Figure 7.4 yields the following head guess:*

```
1 e(1,2). {f(X,Y)}:-e(X,Y).
2 {q'(X,Y)} :- f(X,Y). {f'(X,Y)} :- q(Y,X).
3 q(X1,X2):-q'(X1,X2). f(X1,X2):-f'(X1,X2).
```

Observe that grounding the non-ground head guess from above yields exactly those dependencies in the ground dependency graph that are necessary for the solver to handle cycles. In Figure 7.3 we show the dependency graph of the non-ground head guess. Observe the cycle including the predicate heads $\{f, f', q, q'\}$.

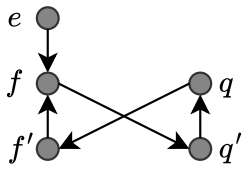


Figure 7.3: Dependency graph of Π rewritten by the $\mathcal{HG}$ procedure of Figure 7.4.

Although the method presented in Figure 7.4 is conceptually simple, its worst case grounding size is the same as for bottom-up grounding. So let $\mathcal{V} = \max_{r \in \Pi} |\operatorname{var}(r)|$, then its grounding

Replacing Rules (4.1) by:

$$h'(\mathbf{D}) \vee \overline{h'(\mathbf{D})} \leftarrow \quad \text{for every } r \in \Pi_1, \text{ with } \{h(\mathbf{X})\} = H_r, \; |\operatorname{SCC}(\Pi, h)| = 1, \mathbf{D} \in \operatorname{dom}(\mathbf{X}) \tag{7.10}$$

$$h'(\boldsymbol{D}_{\langle \boldsymbol{X} \rangle}) \vee \overline{h'(\boldsymbol{D}_{\langle \boldsymbol{X} \rangle})} \leftarrow p_i(\boldsymbol{D}_{\langle \boldsymbol{Y}_i \rangle}), \ldots, p_j(\boldsymbol{D}_{\langle \boldsymbol{Y}_j \rangle}) \quad \text{for every } r \in \Pi_1, \text{ with } \{h(\mathbf{X})\} = H_r, \; \boldsymbol{P} := \{p(\boldsymbol{Y}) | p(\boldsymbol{Y}) \in B_r^+, \operatorname{SCC}(\Pi, h) = \operatorname{SCC}(\Pi, p)\} = \{p_i(\boldsymbol{Y}_i), \ldots, p_j(\boldsymbol{Y}_j)\}, |\operatorname{SCC}(\Pi, h)| > 1, \; \boldsymbol{D} \in \operatorname{dom}(\langle \boldsymbol{X}, \boldsymbol{Y}_i, \ldots, \boldsymbol{Y}_j \rangle) \tag{7.11}$$

Figure 7.4: Hybrid Grounding cycle revive procedure $\mathcal{HG}(\Pi_1, \Pi_2)$ for handling non-tight normal programs. This enables the solver to handle non-trivial SCCs without any additional techniques.

size is in $\mathcal{O}\left(|\Pi| \cdot |\operatorname{dom}(\Pi)|^{\mathcal{V}}\right)$. However, in the special case when the variables of the cyclic body predicates coincide with the variables of the head, we obtain the desired grounding size of body-decoupled grounding (a being the maximum arity of the program): $\mathcal{O}\left(|\Pi| \cdot |\operatorname{dom}(\Pi)|^{a}\right)$. Therefore, for practical problems where the variables of the head and the cyclic B_r^+ atoms overlap, the method of Figure 7.4 is expected to yield good results.

7.4 Lazy-BDG: Novel Lazy Dependency Graph repair

Lazy-BDG revives cycles lazily. While in Section 7.3 we revived cycles before the grounding step, Lazy-BDG does it during solving. The conceptual idea is to change the standard unfound-set approach (Algorithm 2.4) for ground ASP, to an adapted form which can handle non-ground ASP. To enable this, we propose a two-step solution, where the first step rewrites the non-ground program and the second step is the adapted (non-ground) unfound set algorithm.

The main challenge the adapted unfound-set algorithm has to tackle is that the grounded SCCs of $\mathcal{G}(\Pi)$ and $\mathcal{HG}(\Pi_1, \Pi_2)$ do not coincide (see also Section 7.1). Therefore, in addition to the ground information we have to include non-ground dependency graph information in the solving step. However, this is not sufficient, as the bodies of Π were split-up, and therefore the external body literals EB of $\mathcal{G}(\Pi)$ and $\mathcal{HG}(\Pi_1, \Pi_2)$ do not match. To circumvent this problem, we introduce a rewriting (Algorithm 7.1) that lets us distinguish between two types of head literals: a and a_r, where the later one is unique on a per non-ground rule basis. This lets us use a_r as external body literals EB_r for a, and in addition use them as the external body nogoods.

The adapted unfound-set algorithm (Algorithm 7.2) takes this into account. Intuitively, the algorithm tries to find external support for a set of true atoms U. It does this by following the path of the non-ground dependency graph, while keeping track of external body literals a_r. If it finds external support, so whenever for an atom $a \in \mathcal{A}^{\boldsymbol{T}}$ it finds an a_r s.t. $\operatorname{SCC}(\Pi, a) \neq \operatorname{SCC}(\Pi, a_r)$, then it continues with with another atom that is true in the current assignment. If it finds an unfound set U, it lazily injects relevant nogoods to the CDNL procedure. We first detail the rewriting step, which is followed by the adapted unfound-set algorithm.

Algorithm 7.1: RewritingAlgorithm

Data: Set Π_1 (BDG), set Π_2 (SOTA), set of *SCCs*
Result: Set Π'_1, set Π'_2

```
1  Π'_1 ← ∅; Π'_2 ← ∅ ;
2  for SCC ∈ SCCs do
3      if |SCC| > 1 and ∃r ∈ rules(SCC) s.t. r ∈ Π_1 then
           ;                    /* Handle non-trivial SCC with BDG rule */
4          for r ∈ rules(SCC) do
5              r' ← {h_r ← B_r} ;
6              r'' ← {h ← h_r} ;
7              if r ∈ Π_2 then
8                  Π'_2 ← Π'_2 ∪ {r'} ;
9              else
10                 Π'_1 ← Π'_1 ∪ {r'} ;
11             end
12             Π'_2 ← Π'_2 ∪ {r''};
13         end
14     else
           ;                                          /* All other rules */
15         for r ∈ rules(SCC) do
16             if r ∈ Π_2 then
17                 Π'_2 ← Π'_2 ∪ {r} ;
18             else
19                 Π'_1 ← Π'_1 ∪ {r} ;
20             end
21         end
22     end
23 end
24 return Π'_1, Π'_2 ;
```

7.4.1 Rewriting

To determine cyclic support we rewrite the program $\Pi = \Pi_1 \cup \Pi_2$ to distinguish between two types of head literals: a and a_r. The rewriting algorithm is depicted in Algorithm 7.1. We assume as an input a program Π_1, which is a set of rules that shall be grounded with BDG, and a set Π_2 which shall be grounded by traditional means[1]. The output are the sets Π'_1, and Π'_2, which contain the rewritten rules that shall be grounded with BDG, and by traditional means, respectively.

We consider SCCs based on the positive dependency graph $\mathcal{D}_P(\Pi)$ of the non-ground program. Therefore, $SCC \in SCC(\Pi)$ is a set of predicate names. Let $rules(SCC)$ be $rules(SCC) = \{r \mid r \in \Pi, H_r \cap SCC \neq \emptyset, B_r^+ \cap SCC \neq \emptyset\}$.

[1] Note that one can use Algorithm 5.3 to determine Π_1 and Π_2

(a) Pre-Rewriting (pos) dependency graph. (b) Post-Rewriting (pos) dependency graph.

Figure 7.5: Dependency graphs before and after the rewriting by Algorithm 7.1 for Example 7.3.

In more detail, we loop over every SCC in the non-ground program (Line (2)). Line (3) ensures that the rewriting is only used, if at least one rule in the current SCC shall be grounded by BDG means, and the SCC is non-trivial. Then Lines (5)–(6) rewrite the rule into two rules: Rule r' and r'', where r' captures the original rule with a unique head atom name and r'', which infers the original head atom from the newly introduced one. Lines (7)–(11) mark r' to be grounded by BDG, or by SOTA techniques, as r' contains the original body. In Line (12) rule r'' is marked to be handled with SOTA techniques. Lines (15)–(21) then handle those SCCs that shall not be rewritten.

Example 7.3. *We consider* Π*, shown in the following listing:*

```
1 c(1). c(2). d(X) :- c(X), not nd(X). nd(X) :- c(X), not d(X).
2 a(X) :- d(X). a(X) :- b(X). b(X) :- a(X).
```

We obtain the following SCCs: $SCC(\Pi) = \{\{c\}, \{d\}, \{nd\}, \{a, b\}\}$. *Let* Π_1 *be* $\Pi_1 = \{a(X) \leftarrow b(X)\}$, $\Pi_2 = \Pi \setminus \Pi_1$. *Note that* $\{a, b\}$ *is the only non-trivial SCC, where also a rule shall be grounded with BDG.*

Therefore, we obtain for $rules(\{a, b\})$ *the set* $rules(\{a, b\}) = \{a(X) \leftarrow b(X); a(X) \leftarrow d(X); b(X) \leftarrow a(X)\}$. *This leads to the following rewritten program* Π'*:*

```
1 c(1). c(2). d(X) :- c(X), not nd(X). nd(X) :- c(X), not d(X).
2 a_1(X) :- d(X). a_2(X) :- b(X). b_1(X) :- a(X).
3 a(X) :- a_1(X). a(X) :- a_2(X). b(X) :- b_1(X).
```

Algorithm 7.2: Non-Ground UnfoundSet

Data: A (rewritten) ground program Π, assignment $\mathcal{A}$, and rewriting information.
Result: An unfounded set U, and its external bodies $\mathrm{EB_r}$, of Π w.r.t. $\mathcal{A}$.

$S = \{p \mid p \in \mathrm{at}(\Pi), |\mathrm{SCC}(\Pi, p)| > 1, \exists q \in \mathrm{SCC}(\Pi, p) : q.\mathrm{name} \in \mathrm{names}(BDG), p \in \mathcal{A}^{\boldsymbol{T}}\}$;
$\mathrm{safe}_{U_r} \leftarrow \emptyset$; $U_\mathrm{r} \leftarrow \emptyset$;
while $S \neq \emptyset$ **do**
  $U \leftarrow \{p\}$ s.t. $p \in S$;
  while $U \neq \emptyset$ **do**
    $\mathrm{EB_r} \leftarrow \mathrm{getPredecessors}(U, U_\mathrm{r})$;
    foundFlag $\leftarrow$ false;
    if $|EB_r \cap \mathcal{A}^{\boldsymbol{T}}| > 0$ **then**
      for $h_r \in \left(EB_r \cap \mathcal{A}^{\boldsymbol{T}}\right)$ **do**
        if $(SCC(\Pi, h_r) \neq SCC(\Pi, p)) \vee \left(h_r \in safe_{U_r}\right)$ **then**
          $U \leftarrow U \setminus \{h\}$; $S \leftarrow S \setminus \{h\}$;
          $\mathrm{safe}_{U_r} \leftarrow \mathrm{safe}_{U_r} \cup \{h_r\}$;
          $\mathrm{safe}_\mathrm{tmp} \leftarrow \mathrm{inferSafeSuccessors}(h)$;
          $\mathrm{safe}_{U_r} \leftarrow \mathrm{safe}_{U_r} \cup \mathrm{safe}_\mathrm{tmp}$; $U_r \leftarrow U_r \setminus \mathrm{safe}_\mathrm{tmp}$;
          foundFlag $\leftarrow$ true;
          break;
        else
          $\boldsymbol{B} \leftarrow \mathrm{findPositiveBodies}(h_r)$;
          for $\beta \in \boldsymbol{B}$ **do**
            $U \leftarrow U \cup (\beta^+ \cap (\mathrm{SCC}(\Pi, p) \cap S))$;
            foundFlag $\leftarrow$ true;
          end
          $U_\mathrm{r} \leftarrow U_\mathrm{r} \cup \{h_r\}$;
          break;
        end
      end
    end
    if *foundFlag* $=$ *false* **then**
      return $U, \mathrm{EB_r}$;
    end
  end
end
return $\emptyset, \emptyset$;

With the sets $\Pi'_1 = \{a_2(X) \leftarrow b(X)\}$, $\Pi'_2 = \Pi' \setminus \Pi'_1$. *We show in Figure 7.5a the positive dependency graph before the rewriting, and in Figure 7.5b the positive dependency graph after the rewriting.*

7.4.2 Non-ground Unfound-Set

In detail the algorithm works as follows: The algorithm first gathers all to be checked atoms in the set S (Line (1)). S contains all those true atoms, that are part of a non-trivial SCC, where at least one atom is grounded by BDG. Then the algorithm checks for each such atom, whether there exists support for it (Line (4)). This is done by successively extending a set of unfounded atoms U, until either support is found (success, Line (33)), or U cannot be extended anymore (failure, Line (29)). In the failure case, nogoods representing the unfounded set and its respective external body atoms are added to the current state.

As long as there are still atoms in U left to check (Line (5)), it calls the getPredecessors(U, U_r) procedure. getPredecessors(U, U_r) gets all *external bodies* EB_r of U that are not part of U_r. Note that EB_r are in our case the heads h_r of the rewritten rules of Algorithm 7.1. More formally:

$$\text{getPredecessors}(U, U_r) = \{h_r \mid h \in U, r \in \mathcal{G}(\Pi), r = \{h \leftarrow h_r\}, r \notin U_r\}$$

If there are still true external bodies left (Line (8)), then we continue trying to find support (Lines (9)–(26)). Otherwise, we immediately fail (Line (29)).

We try to find support by going through each external body h_r one by one (Line (9)). If h_r is either from a different SCC, or is already marked as safe, as h_r was previously marked as externally supported (Line (10)), then we update its dependencies (Lines (11)–(14)) and restart (Lines (15)–(16)) the loop (Line (5)). In detail, we change the dependencies by removing h (the non-rewritten literal) from U and S (Line (11)). Next we add h_r to the safe_{U_r} set, which only has an impact in the $SCC(\Pi, h_r) \neq SCC(\Pi, h)$ case (Line (12)).

Then we call the inferSafeSuccessors(h) function (Line (13)). This function infers those external bodies safe_tmp, which can be inferred by knowing that h is safe. In more detail:

$$\text{inferSafeSuccessors}(h) \coloneqq \{a_r \mid r \in \mathcal{G}(\Pi), h \in B_r^+, B_r^+ \cap S = \emptyset, a_r \in H_r\}$$

We still are required to add the supported external bodies to safe_{U_r} and remove them from U_r, to enable each a_r to be considered as a possible support for a in a subsequent step (Line (14)).

The other case is when h_r cannot support h (Line (18)). Then we call the findPositiveBodies(h_r) function in Line (18). findPositiveBodies(h_r) tries to infer positive bodies for h_r. Due to our rewriting, there is one unique non-ground body for h_r, which must be instantiated for this check. findPositiveBodies(h_r) implements a join operation on the literals of the body, w.r.t. the grounded head h_r, non-ground rule $r \in \Pi$, and the assignment $\mathcal{A}$. So more formally:

$$\text{findPositiveBodies}(h_r) = \{B_{r'} \mid r' \in \mathcal{G}(r), B_{r'}^+ \subseteq \mathcal{A}^{\boldsymbol{T}}, B_{r'}^- \subseteq \mathcal{A}^{\boldsymbol{F}}, h_r \in H_{r'}\}$$

If we find bodies (Lines (20)-(21)), we extend the unfound-set U with those body atoms that are in the same SCC as the head. Further, we also flag the external body h_r in U_r, to not be

considered as an external body in a subsequent step (Line (23)). Lastly, Line (24) restarts the loop in Line (5). We note that Lines (18)–(23) can be implemented lazily, by getting one instantiated body at a time, to prevent (unnecessary) combinatorial explosions.

Example 7.4 (Ctd. Example 7.3)**.** *We show how Algorithm 7.2 works for the rewritten program in Example 7.3. The example program has* 4 *answer sets:* $\mathcal{AS} = \{\{c(1); c(2); nd(1); nd(2)\}; \{c(1); c(2); d(1); nd(2); a(1); b(1)\}; \{c(1); c(2); nd(1); d(2); a(2); b(2)\}; \{c(1); c(2); d(1); d(2); a(1); a(2); b(1); b(2)\}\}$.

We showcase the workings for $\{d(1); d(2)\} \subset \mathcal{I}$ *(*$\mathcal{I}$ *being the answer set). As Algorithm 7.2 is called in the check-phase of the CDNL-procedure, the current assignment* $\mathcal{A}$ *already fulfills satisfiability of every rule, and every* $a \in \mathcal{I}$ *has a rule that justifies it. Our assignment is therefore* $\mathrm{at}(\Pi') \cap \mathcal{A}^{\boldsymbol{T}} = \{c(1); c(2); d(1); d(2); a(1); b(1); a(2); b(2)\ a_1(1); a_2(1), a_1(2); a_2(2); b_1(1); b_1(2)\}$. *It remains to show foundedness by Algorithm 7.2. We display the example execution in Table 7.1 that derives that no additional nogoods are needed.*

Example 7.5 (Ctd. Example 7.3)**.** *We show how Algorithm 7.2 can derive nogoods for the rewritten program in Example 7.3. We change Example 7.4 by considering the case* $\{nd(1); nd(2)\} \subset \mathcal{I}$*, with the assignment* $\mathrm{at}(\Pi') \cap \mathcal{A}^{\boldsymbol{T}} = \{c(1); c(2); nd(1); nd(2); a(1); b(1); a(2); b(2)\ a_2(1), a_2(2); b_1(1); b_1(2)\}$.

This assignment is a forbidden cyclic derivation, where we continue to show how Algorithm 7.2 prevents it. We display the example execution in Table 7.2 that derives $U = \{b(1); a(1)\}$ *and* $EB_r = \{a_1(1)\}$*. From this we follow the nogoods:*

$$\lambda(b(1), U) = \{\boldsymbol{T}b(1), \boldsymbol{F}a_1(1)\}$$
$$\lambda(a(1), U) = \{\boldsymbol{T}a(1), \boldsymbol{F}a_1(1)\}$$
$$\Lambda_\Pi(U) = \{\{\boldsymbol{T}b(1), \boldsymbol{F}a_1(1)\}, \{\boldsymbol{T}a(1), \boldsymbol{F}a_1(1)\}\}$$

By adding these nogoods to the solver we prevent the cyclic derivation.

7.5 Experiments

To demonstrate the feasibility of Lazy-BDG we implemented Algorithm 7.2 in `newground3` as a `clingo`-propagator. Lazy-BDG not only achieves much better results than the previous state-of-the-art BDG approach (level mappings), but also beats `gringo` and `idlv` on our synthetic scenarios. Note that our prototypical implementation is in python. Therefore, we expect to obtain even better results for future non-prototypical implementations in a combined grounder and solver. Note that the benchmark system and overall setup is the same as in Section 5.4.

7.5.1 Propagator

A propagator is part of the CDNL procedure that can track progress and add nogoods. The clingo propagator interface [84] in python consists of four methods: *init(init)*: that is called once before solving, to initialize data structures. *propagate(control,changes)*: during solving

#It.	Line	Event
0	1	$S = \{a(1); b(1); a(2); b(2)\}$
	2	$\text{safe}_{U_r} = \emptyset; U_r = \emptyset$
	4	$U = \{b(1)\}; p = b(1)$ (arbitrarily chosen)
1	6	$\text{EB}_r = \{b_1(1)\}$
	9	$h_r = b_1(1)$
	10	$SCC(\Pi, h_r) = SCC(\Pi, p) \wedge$ $h_r \notin \text{safe}_{U_r}$
	19	$\beta = \{a(1)\}$
	20	$U = \{b(1); a(1)\}$
	23	$U_r = \{b_1(1)\}$
2	6	$\text{EB}_r = \{a_1(1); a_2(1)\}$
	9	$h_r = a_2(1)$
	10	$SCC(\Pi, h_r) = SCC(\Pi, p) \wedge$ $h_r \notin \text{safe}_{U_r}$
	19	$\beta = \{b(1)\}$
	20	$U = \{b(1); a(1)\}$
	23	$U_r = \{b_1(1); a_2(1)\}$
3	6	$\text{EB}_r = \{a_1(1)\}$
	9	$h_r = a_1(1)$
	10	$SCC(\Pi, h_r) \neq SCC(\Pi, p)$
	11	$U = \{b(1)\};$ $S = \{b(1); a(2); b(2)\}$
	12	$\text{safe}_{U_r} = \{a_1(1)\}$
	13	$\text{safe}_{\text{tmp}} = \{b_1(1)\}$
	14	$\text{safe}_{U_r} = \{a_1(1); b_1(1)\};$ $U_r = \{a_2(1)\}$
4	6	$\text{EB}_r = \{b_1(1)\}$
	9	$h_r = b_1(1)$
	10	$h_r \in \text{safe}_{U_r}$
	11	$U = \emptyset; S = \{a(2); b(2)\}$
	12	$\text{safe}_{U_r} = \{a_1(1); b_1(1)\}$
	13	$\text{safe}_{\text{tmp}} = \{a_1(1); a_2(1)\}$
	14	$\text{safe}_{U_r} = \{a_1(1); b_1(1); a_2(1)\};$ $U_r = \emptyset$
5	4	$U = \{b(2)\}; p = b(2)$ (equivalent to #It. 1–4; not shown)

Table 7.1: Execution of Algorithm 7.2 for Example 7.4. Iterations ≥ 5 are not shown.

it is called with a *PropagateControl* (control) and a list of tracked changes. *PropagateControl*: can be used to add nogoods, add literals, trigger propagation, and get the current assignment. *undo(thread_id, assignment, changes)*: is the opposite of *propagate*. *check(control)*; is called on total assignments, otherwise similar to propagate. We implemented our Algorithm 7.2 in the *check* function. Therefore, we operate on total assignments in our algorithm. Note that one could implement the algorithm also partly in *propagate*.

In Figure 7.6 we show the schematics of the implementation and workings of Lazy-BDG. Given a program Π Algorithm 7.1 rewrites the program to prevent an information-loss. Subsequently

#It.	Line	Event	#It.	Line	Event
0	1	$S = \{a(1); b(1); a(2); b(2)\}$	2	6	$\text{EB}_r = \{a_2(1); a_1(1)\}$
	2	$\text{safe}_{U_r} = \emptyset; U_r = \emptyset$		9	$h_r = a_2(1)$
	4	$U = \{b(1)\}; p = b(1)$ (arbitrarily chosen)		10	$SCC(\Pi, h_r) = SCC(\Pi, p) \wedge$ $h_r \notin \text{safe}_{U_r}$
1	6	$\text{EB}_r = \{b_1(1)\}$		19	$\beta = \{b(1)\}$
	9	$h_r = b_1(1)$		20	$U = \{b(1); a(1)\}$
	10	$SCC(\Pi, h_r) = SCC(\Pi, p) \wedge$ $h_r \notin \text{safe}_{U_r}$		23	$U_r = \{b_1(1); a_2(1)\}$
	19	$\beta = \{a(1)\}$	3	6	$\text{EB}_r = \{a_1(1)\}$
	20	$U = \{b(1); a(1)\}$		8	$\text{EB}_r \cap \mathcal{A}^T = \emptyset$
	23	$U_r = \{b_1(1)\}$		28	*foundFlag = false*
				29	**return** $U = \{b(1); a(1)\}$, $\text{EB}_r = \{a_1(1)\}$

Table 7.2: Execution of Algorithm 7.2 for Example 7.5. The algorithm derives that additional nogoods are needed.

hybrid grounding grounds the program. Whenever `clingo` derives a full assignment $\mathcal{A}$ we need to check it for cyclic derivations. This is performed by our adapted unfound set algorithm (Algorithm 7.2), which was implemented as a `clingo`-propagator.

7.5.2 Benchmark Scenarios

We adapted our scenarios from Chapter 6 to cyclic programs. Further, we reduced our instance sizes to a suitable fragment. Therefore, for scenario *C-Four-Clique* we generated 80 complete graphs from size 1 to 80. For the other scenarios we used the 20 first instances (size 1 to 20).

The scenarios are shown in the listings below. Our first scenario is the adapted four-clique (*C-Four-Clique*), where we combine neighboring cliques (Line (3)). Note that the level mappings technique has an approximate grounding size of $\approx |\text{dom}|^6$, whereas SOTA techniques have a grounding size of $\approx |\text{dom}|^4$. The grounding size of Lazy-BDG is $\approx |\text{dom}|^2$ (we used $\mathcal{HG}_{\mathcal{F}}$ for the benchmarks). However, this is internally processed and therefore, we consider Lazy-BDG as a combined grounder and solver.

```
1 {ff(X,Y)} :- edge(X,Y).
2 f(X,Y) :- ff(X,Y).
3 f(X,Y) :- c(X), c(Y), not f(X,Y).
4 % The following rule is grounded with the novel foundedness approach:
```

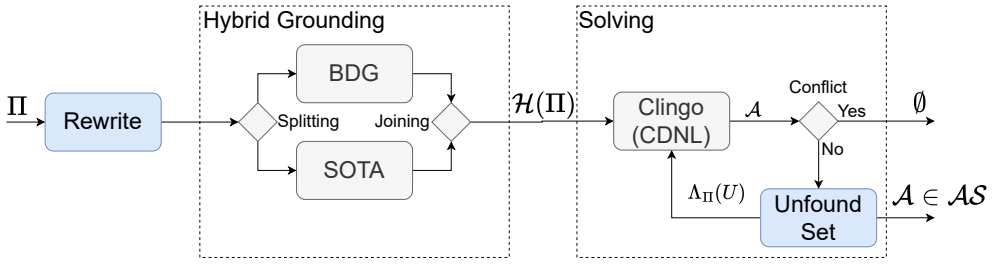


Figure 7.6: Schematics of the prototypical implementation and workings of Lazy-BDG. Given a program Π, the rewriting algorithm (Algorithm 7.1) prevents information-loss. Grounding $\mathcal{H}(\Pi)$ is performed by hybrid grounding, which results (when using FastFound) in an (intermediate) grounding size of $\mathcal{O}(|\Pi| \cdot |\text{dom}(\Pi)|^{a+1})$. The CDNL algorithm produces full assignments $\mathcal{A}$, where we know that all rules are satisfied, and whenever $a \in \mathcal{A}^T$, then there is a rule r that is suitable for justifying a. It remains to prevent cyclic derivations, which is done with the adapted unfound set algorithm (Algorithm 7.2). Either additional nogoods ($\Lambda_\Pi(U)$) are derived, which prevent cyclic derivations, or an answer set ($\mathcal{A} \in \mathcal{AS}$) is found. Objects colored in blue mark the contributions of Lazy-BDG.

```
5 c(X):-f(X,X1),f(X,X2),f(X,X3),f(X1,X2),f(X1,X3),f(X2,X3).
```

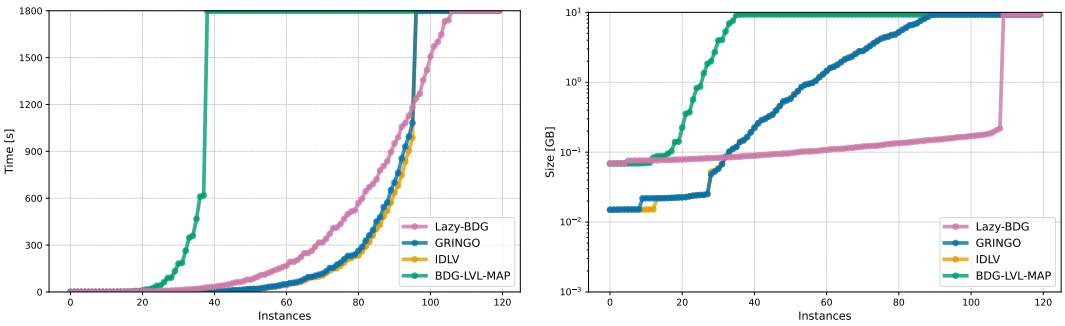
The second scenario *C-Hyper-Six-Clique* features a cyclic $\{d, c\}$ component. level mappings[2] have a grounding size of $\approx |\text{dom}|^9$, and SOTA techniques have one of $\approx |\text{dom}|^6$, whereas Lazy-BDG has an internal grounding size of $\approx |\text{dom}|^3$.

```
1 {f(X,Y,Z)} :- edge(X,Y), edge(Y,Z).
2 d(X,Y) :- f(X,Y,Z).
3 % The following rule is grounded with the novel foundedness approach:
4 c(X1,X2) :- f(X1,X2,X3), f(X4,X5,X6), f(X1,X4,X5), f(X2,X4,X5), f(X3,X6,X5),
5     f(X6,X1,X2), f(X3,X4,X5), d(X1,X2).
6 d(X1,X2) :- c(X2,X1).
```

The third scenario *C-Hyper-Seven-Clique* extends scenario *C-Hyper-Six-Clique* to a seven clique. level mappings have a grounding size of $\approx |\text{dom}|^9$, and SOTA techniques have one of $\approx |\text{dom}|^7$, whereas Lazy-BDG has an internal grounding size of $\approx |\text{dom}|^3$.

```
1 {f(X,Y,Z)} :- edge(X,Y), edge(Y,Z).
2 d(X,Y) :- f(X,Y,Z).
3 % The following rule is grounded with the novel foundedness approach:
4 c(X1,X2):-f(X1,X2,X3), f(X4,X5,X6), f(X1,X4,X5), f(X2,X4,X5), f(X3,X6,X5),
5     f(X6,X1,X2), f(X3,X4,X5), f(X7,X1,X6), f(X7,X2,X3), f(X7,X4,X5), d(X1,X2)
          .
6 d(X1,X2) :- c(X2,X1).
```

[2] An improved level mappings approach, which focuses on atoms occurring in the same SCC, results in an approximate grounding size of $\approx |\text{dom}|^6$.



(a) Grounding and Solving Time [s] cactus plot.

(b) Max RAM usage [GB] cactus plot.

Figure 7.7: Cactus plots of benchmarks for demonstrating viability of Lazy-BDG. Measured `gringo`, `idlv`, $\mathcal{HG}$ (BDG-LVL-MAP), and $\mathcal{HG}_{\mathcal{F}}$ (Lazy-BDG). We were able to solve more instances with our *unfound-set* implementation than `idlv`, or `gringo`. Further, UNFOUND-SET uses just a fraction of the RAM that the other methods use. Timeout: 1800s; Memout: 10GB.

7.5.3 Hypotheses

We consider the following hypotheses:

(H1) Lazy-BDG (UNFOUND-SET) has a smaller RAM usage than SOTA techniques or level mappings, on grounding-heavy benchmarks.

(H2) Lazy-BDG (UNFOUND-SET) is able to yield an improvement in number of solved instances on grounding-heavy scenarios.

7.5.4 Experimental Results

We show our results in Figure 7.7 and Table 7.3. Observe the substantial differences in RAM usage (Figure 7.7b), where Lazy-BDG's RAM usage increase is relatively mild, in comparison to `gringo`'s, `idlv`'s, and level mapping's. From this observation we confirm hypothesis H1. Further, Lazy-BDG was able to solve more instances than LVL.-MAP and slightly more instances than `gringo` or `idlv` (Table 7.3). Our implementation is prototypical, still we were able to ground-and-solve more instances than `gringo` and `idlv` on our synthetic datasets, which enables us to confirm H2.

Scen.	#Insts.	Total #Solved											
		`gringo`			`idlv`			LVL.-MAP.			Lazy-BDG		
		#S	M	T	#S	M	T	#S	M	T	#S	M	T
Total-SUM	120	96	0	24	96	0	24	38	0	82	**106**	0	14
C-Hyper-Six-Clique	20	**17**	0	3	**17**	0	3	10	0	10	15	0	5
C-Hyper-Seven-Clique	20	12	0	8	12	0	8	9	0	11	**14**	0	6
C-Four-Clique	80	67	0	13	67	0	13	19	0	61	**77**	0	3

Table 7.3: Lazy-BDG can yield a performance improvement for grounding-heavy benchmarks. We report number of solved instances on three synthetic (cyclic) scenarios. Benchmarked systems include `gringo`, `idlv`, BDG with level-mappings (LVL.-MAP.), and our Lazy-BDG technique. Measured number of solved instances (#S), memouts (M), timeouts (T). Timeout of 1800s, Memout of 10GB.

CHAPTER 8

Summary and Conclusion

We conclude the thesis with a brief discussion of the main results, and future work. This thesis's main contributions can be pinpointed to the advancement of the Body-decoupled Grounding (BDG) method. Although BDG is historically a complexity theoretic method, it shines for practical purposes in decreasing the impact of the *grounding bottleneck* in Answer Set Programming (ASP). This stems from its ability to decrease the grounding size from being exponential in the number of the variables per rule (state-of-the-art ground-and-solve approaches), to being (only) exponential in the maximum arity of a predicate.

Essentially, this thesis enables the interoperability of BDG with state-of-the-art systems and pushes the BDG method to be competitive for normal and non-tight (cyclic) rules. The details of these findings correspond to the main chapters of the thesis: (i) *hybrid grounding* [10] in Chapter 4, (ii) *automated hybrid grounding* in Chapter 5, (iii) *FastFound* in Chapter 6, and (iv) *Lazy-BDG* in Chapter 7.

We think that especially (i) and (ii) will be of practical importance and should greatly ease the integration of BDG into state-of-the-art grounders. This comes, as these contributions provide the theoretical underpinnings for interoperability (i) and show how it can be practically used (ii).

On the other hand, (iii) and (iv) are primarily of theoretical significance, demonstrating the potential for future development of the BDG approach. One of the key achievements of FastFound was the reduction of the grounding size[1] for non-ground normal rules from $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^{2 \cdot a})$ to $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^{a+1})$. With Lazy-BDG, the (intermediate) grounding size was reduced from $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^{3 \cdot a})$ to $\mathcal{O}(|\Pi| \cdot |\mathrm{dom}(\Pi)|^{a+1})$. Further, with both (iii) and (iv), we were able to beat the state-of-the-art on synthetic grounding-heavy benchmarks, which, in combination with the interoperability gained by (i) and (ii), makes them promising future directions of research.

[1] Let Π be a non-ground normal (HCF) program, $|\Pi|$ be the program size, $|\mathrm{dom}(\Pi)|$ be the program's domain size, and a be the maximum arity of any predicate in Π.

Future Work: Although we think that BDG and this thesis are of great importance towards solving the grounding bottleneck, there are still many questions left open, and problems to be solved. With respect to BDG, we identified its shortcomings and open questions in Section 3.3, where we identified seven different areas of future work, of which this thesis addresses three in detail and three other shortcomings partially[2].

Conversely, one problem remains fully open: *The Arity Problem* (Section 3.3.5). In essence, it asks whether it is possible to split literals in ASP into literals with smaller arity, while preserving semantics.

However, for major performance gains besides the arity problem, we think that future work should address two problems that we partly covered in this thesis: *Domain Dependent Grounding* (Section 3.3.6) and *Shifting effort from the Grounder to the Solver* (Section 3.3.7). Essentially, domain dependent grounding asks whether we can incorporate more instance, or domain information into the BDG approach. We partly covered this by taking a step back to a meta level and introduced heuristics that analyzes the instance. However, it is preferable to include this instance knowledge directly into the BDG reduction by adapting it. The objection behind *shifting effort from the grounder to the solver* can be summarized as *what we gain in grounding, we lose in solving*, where we intentionally stated it as broad as possible. We include multiple different future vectors of research in this line. Among them is an extended Lazy-BDG approach that tackles foundedness or satisfiability, or the study of the impact of BDG on problems that need multiple answer sets, like brave- or cautious-reasoning, counting, or optimization.

Further, we deem the investigation of interoperability with other non-standard grounding methods, like compilation based techniques, as promising. This stems from the recent successes of combining bottom-up methods with compilation based techniques in the ProASP solver [38]. This raises the question of whether it is feasible to combine BDG with compilation based techniques. We see BDG and compilation based techniques as orthogonal to each other in the sense that BDG can be used as a rewriting technique for ProASP. However, at the current state of development, both the theory and the practical implementation of compilation based techniques support the normal (tight) fragment of ASP programs. As BDG produces ground disjunctive (cyclic) ASP programs, BDG is, at the time of writing, not applicable to compilation based techniques, which might change in the future.

Finally, we strive towards an integration of the BDG approach into state-of-the-art grounders like `gringo` or `idlv`. This thesis establishes a solid foundation for exploring this integration, leading to more efficient problem-solving across various applications.

[2] In more detail, this thesis addressed primarily *Limited Interoperability* (Section 3.3.1), *When should BDG be used?* (Section 3.3.3), and *Bad Performance for Normal and Cyclic Programs* (Section 3.3.4). Further, this thesis partly addressed *Limited accepted Syntax* (Section 3.3.2), *Domain Dependent Grounding* (Section 3.3.6), and *Shifting effort from the Grounder to the Solver* (Section 3.3.7).

Overview of Generative AI Tools Used

Generative AI-tools were used to generate Figure 14 in the Appendix (all figures shown in Figure 1). Besides that no AI-tools, besides spell-checkers (Grammarly, and ChatGPT 4o), were used. For spell-checking with ChatGPT 4o we used prompts similar to the one shown in the following listing (this particular prompt was used with ChatGPT 4o on 30^{th} of December 2024):

```
1 Dear ChatGPT, please take a look at the pdf I attached. It is part of my
      current MSC thesis, and I want you to look over it regarding spelling
      mistakes.
2 I am a non-native speaker, and therefore, this is important for me.
3 If you find a spelling mistake, please (i) provide me with the sentence where
       it occurs (mark the error in bold), (ii) the corrected sentence, (iii)
      the page number in the pdf, where it occurs.
```

Regarding the generation of Figure 14, we used ChatGPT 4o (via the web-interface) on the 23^{rd} of December 2024. The original prompt is shown in the listing below and generated Figure 1a.

```
1 Hi ChatGPT, draw me a "fortune teller-ball observatory high up in the alps".
2 This observatory is part of an institute doing latest research on artificial
      intelligence.
3 Draw the master piece in a comic style.
```

In the next listing we show the prompt for generating Figure 14a (Figure 1b), where we passed Figure 1a as an additional input.

(a) The original generated figure for the fortune-teller-ball observatory.

(b) Ivory Tower

(c) Fortune teller-ball-observatory

(d) Tower of Wisdom

Figure 1: Figures produced with ChatGPT 4o on 23^{rd} of December 2024.

```
1 The attached master piece shows a "fortune teller-ball observatory high up in
     the alps" that is part of a larger research institute doing cutting-edge
     research on artificial intelligence.
2 I want you to draw me the "ivory tower" of this institute.
```

The next listing depicts our prompt for Figure 14b (Figure 1c), where we passed Figure 14a (Figure 1b) as an additional input:

```
1 The attached master piece shows an "ivory tower high up in the alps" that is
    part of a larger research institute doing cutting-edge research on
    artificial intelligence.
2 I want you to draw me the "fortune-teller-ball observatory" of this institute
    .
```

Finally, the next listing depicts how we generate Figure 14c (Figures 1d) by additionally including Figure 14a (Figure 1b), and Figure 14b (Figure 1c) as inputs.

```
1 The attached master pieces show a "fortune-teller-ball observatory" and an "
    ivory tower," both "high up in the snowy alps."
2 They are part of a larger research institute doing cutting-edge research on
    artificial intelligence.
3 I want you to draw me "tower of wisdom" of this institute.
```

Bibliography

[1] D. Abels, J. Jordi, M. Ostrowski, T. Schaub, A. Toletti, and P. Wanko, "Train scheduling with hybrid answer set programming", *TPLP*, vol. 21, no. 3, pp. 317–347, 2021. DOI: 10.1017/S1471068420000046.

[2] M. Alviano, F. Calimeri, G. Charwat, M. Dao-Tran, C. Dodaro, G. Ianni, T. Krennwallner, M. Kronegger, J. Oetsch, A. Pfandler, J. Pührer, C. Redl, F. Ricca, P. Schneider, M. Schwengerer, L. K. Spendier, J. P. Wallner, and G. Xiao, "The fourth answer set programming competition: Preliminary report", in *LPNMR13*, P. Cabalar and T. C. Son, Eds., ser. Lecture Notes in Computer Science, vol. 8148, 2013, pp. 42–53. DOI: 10.1007/978-3-642-40564-8_5.

[3] M. Alviano, C. Dodaro, N. Leone, and F. Ricca, "Advances in WASP", in *LPNMR15*, F. Calimeri, G. Ianni, and M. Truszczynski, Eds., ser. Lecture Notes in Computer Science, vol. 9345, 2015, pp. 40–54. DOI: 10.1007/978-3-319-23264-5.

[4] M. Alviano and L. A. Rodriguez Reiners, "ASP chef: Draw and expand", in *KR24*, 2024, pp. 720–730, ISBN: 9781956792058. DOI: 10.24963/kr.2024/68.

[5] R. A. Aziz, G. Chu, and P. J. Stuckey, "Stable model semantics for founded bounds", *TPLP*, vol. 13, no. 4, pp. 517–532, 2013. DOI: 10.1017/S147106841300032X.

[6] M. Balduccini and Y. Lierler, "Constraint answer set solver EZCSP and why integration schemas matter", *TPLP*, vol. 17, no. 4, pp. 462–515, 2017. DOI: 10.1017/S1471068417000102.

[7] M. Banbara, B. Kaufmann, M. Ostrowski, and T. Schaub, "*Clingcon* : The next generation", *TPLP*, vol. 17, no. 4, pp. 408–461, 2017. DOI: 10.1017/S1471068417000138.

[8] A. G. Beiser, "Body-decoupled Grounding for Answer Set Programming extended with Aggregates", Bachelor's Thesis, TUWien, 2023.

[9] A. G. Beiser, S. Hahn, and T. Schaub, "ASP-driven user-interaction with clinguin", *ICLP24*, 2024.

[10] A. G. Beiser, M. Hecher, K. Unalan, and S. Woltran, "Bypassing the ASP bottleneck: Hybrid grounding by splitting and rewriting", in *IJCAI24*, 2024, pp. 3250–3258. DOI: 10.24963/ijcai.2024/360.

[11] V. Besin, "A novel method for grounding in answer-set programming", Master's Thesis, TUWien, 2023.

[12] V. Besin, M. Hecher, and S. Woltran, "Body-decoupled grounding via solving: A novel approach on the ASP bottleneck", in *IJCAI22*, 2022, pp. 2546–2552. DOI: 10.24963/ijcai.2022/353.

[13] V. Besin, M. Hecher, and S. Woltran, "On the structural complexity of grounding – tackling the ASP grounding bottleneck via epistemic programs and treewidth", in *ECAI23*, K. Gal, A. Nowé, G. J. Nalepa, R. Fairstein, and R. Rădulescu, Eds., ser. Frontiers in Artificial Intelligence and Applications, vol. 372, 2023, pp. 247–254. DOI: 10.3233/FAIA230277.

[14] M. Bichler, M. Morak, and S. Woltran, "Lpopt: A rule optimization tool for answer set programming", in *LOPSTR16*, ser. Lecture Notes in Computer Science, vol. 10184, 2016, pp. 114–130. DOI: 10.1007/978-3-319-63139-4_7.

[15] M. Bichler, M. Morak, and S. Woltran, "Selp: A single-shot epistemic logic program solver", *TPLP*, vol. 20, no. 4, pp. 435–455, 2020, DOI: 10.1017/S1471068420000022.

[16] H. L. Bodlaender and T. Kloks, "Efficient and constructive algorithms for the pathwidth and treewidth of graphs", *JAlgo*, vol. 21, no. 2, pp. 358–402, 1996, ISSN: 01966774. DOI: 10.1006/jagm.1996.0049.

[17] J. Bomanson, T. Janhunen, and A. Weinzierl, "Enhancing lazy grounding with lazy normalization in answer-set programming", in *AAAI19*, vol. 33, 2019, pp. 2694–2702. DOI: 10.1609/aaai.v33i01.33012694.

[18] G. Brewka, J. Delgrande, J. Romero, and T. Schaub, "Asprin: Customizing answer set preferences without a headache", in *AAAI15*, vol. 29, 2015, pp. 1467–1474. DOI: 10.1609/aaai.v29i1.9398.

[19] G. Brewka, T. Eiter, and M. Truszczyński, "Answer set programming at a glance", *Commun. ACM*, vol. 54, no. 12, pp. 92–103, 2011, ISSN: 15577317. DOI: 10.1145/2043174.2043195.

[20] P. Cabalar, M. Dieguez, F. Olivier, and T. Schaub, "Temporal here-and-there with constraints", *TAASP23*, 2023.

[21] F. Calimeri, C. Dodaro, D. Fuscà, S. Perri, and J. Zangari, "Efficiently coupling the i-DLV grounder with ASP solvers", *TPLP*, vol. 20, no. 2, pp. 205–224, 2020. DOI: 10.1017/S1471068418000546.

[22] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, and T. Schaub, "ASP-core-2 input language format", *TPLP*, vol. 20, no. 2, pp. 294–309, 2020. DOI: 10.1017/S1471068419000450.

[23] F. Calimeri, D. Fuscà, S. Perri, and J. Zangari, "I-DLV: The new intelligent grounder of DLV", *Intell. Artif.*, vol. 11, no. 1, M. Maratea, G. Adorni, S. Cagnoni, and M. Gori, Eds., pp. 5–20, 2017, ISSN: 22110097. DOI: 10.3233/IA-170104.

[24] F. Calimeri, D. Fuscà, S. Perri, and J. Zangari, "Optimizing answer set computation via heuristic-based decomposition", *PADL18*, Lecture Notes in Computer Science, vol. 10702, F. Calimeri, K. Hamlen, and N. Leone, Eds., pp. 135–151, 2018. DOI: 10.1007/978-3-319-73305-0_9.

[25] F. Calimeri, M. Gebser, M. Maratea, and F. Ricca, "Design and results of the fifth answer set programming competition", *Artif. Intell.*, vol. 231, pp. 151–181, 2016, ISSN: 00043702. DOI: 10.1016/j.artint.2015.09.008.

[26] F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. C. Santoro, M. Sirianni, G. Terracina, and P. Veltri, "The third answer set programming competition: Preliminary report of the system competition track", in *LPNMR11*, J. P. Delgrande and W. Faber, Eds., ser. Lecture Notes in Computer Science, vol. 6645, 2011, pp. 388–403. DOI: 10.1007/978-3-642-20895-9_46.

[27] B. Cuteri, C. Dodaro, F. Ricca, and P. Schüller, "Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis", *TPLP*, vol. 17, no. 5-6, pp. 780–799, 2017, ISSN: 14710684. DOI: 10.1017/S1471068417000254.

[28] B. Cuteri, C. Dodaro, F. Ricca, and P. Schüller, "Overcoming the grounding bottleneck due to constraints in ASP solving: Constraints become propagators", in *IJCAI20*, C. Bessiere, Ed., 2020, pp. 1688–1694. DOI: 10.24963/ijcai.2020/234.

[29] B. Cuteri, C. Dodaro, F. Ricca, and P. Schüller, "Partial compilation of ASP programs", *TPLP*, vol. 19, no. 5, pp. 857–873, 2019, ISSN: 14710684. DOI: 10.1017/S1471068419000231.

[30] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi, "GASP: Answer set programming with lazy grounding", *Fundam. Inform.*, vol. 96, no. 3, pp. 297–322, 2009, ISSN: 01692968. DOI: 10.3233/FI-2009-180.

[31] E. Dantsin, T. Eiter, and G. Gottlob, "Complexity and expressive power of logic programming", *ACM Comput. Surv.*, vol. 33, no. 3, pp. 374–425, 2001. DOI: 10.1145/502807.502810.

[32] M. Dao-Tran, T. Eiter, M. Fink, G. Weidinger, and A. Weinzierl, "OMiGA : An open minded grounding on-the-fly answer set solver", *JELIA12*, Lecture Notes in Computer Science, vol. 7519, L. F. Del Cerro, A. Herzig, and J. Mengin, Eds., pp. 480–483, 2012. DOI: 10.1007/978-3-642-33353-8_38.

[33] B. De Cat, M. Denecker, M. Bruynooghe, and P. Stuckey, "Lazy model expansion: Interleaving grounding with search", *JAIR*, vol. 52, pp. 235–286, 2015, ISSN: 10769757. DOI: 10.1613/jair.4591.

[34] R. Dechter, "Propositional semantics for disjunctive logic programs", *Ann. Math. Artif. Intell.*, vol. 12, no. 1-2, pp. 53–87, 1994. DOI: 10.1007/BF01530761.

[35] M. Denecker, J. Vennekens, S. Bond, M. Gebser, and M. Truszczyński, "The second answer set programming competition", in *LPNMR09*, E. Erdem, F. Lin, and T. Schaub, Eds., ser. Lecture Notes in Computer Science, vol. 5753, 2009, pp. 637–654. DOI: 10.1007/978-3-642-04238-6_75.

[36] R. Diestel, *Graph theory* (Graduate texts in mathematics 173), 2. ed. New York, NY: Springer, 2000, 312 pp., ISBN: 9780387989761.

[37] Y. Dimopoulos, M. Gebser, P. Lühne, J. Romero, and T. Schaub, "Plasp 3: Towards effective ASP planning", *TPLP*, vol. 19, no. 3, pp. 477–504, 2019. DOI: 10.1017/S1471068418000583.

[38] C. Dodaro, G. Mazzotta, and F. Ricca, "Blending grounding and compilation for efficient ASP solving", in *KR24*, P. Marquis, M. Ortiz, and M. Pagnucco, Eds., 2024, pp. 317–328. DOI: 10.24963/kr.2024/30.

[39] C. Dodaro, G. Mazzotta, and F. Ricca, "Compilation of tight ASP programs", in *ECAI23*, K. Gal, A. Nowé, G. J. Nalepa, R. Fairstein, and R. Rădulescu, Eds., ser. Frontiers in Artificial Intelligence and Applications, vol. 372, 2023, pp. 557–564. DOI: 10.3233/FAIA230316.

[40] T. Eiter, W. Faber, M. Fink, and S. Woltran, "Complexity results for answer set programming with bounded predicate arities and implications", *Ann. Math. Artif. Intell.*, vol. 51, no. 2, pp. 123–165, 2007, ISSN: 15737470. DOI: 10.1007/s10472-008-9086-5.

[41] T. Eiter, M. Fink, G. Ianni, T. Krennwallner, C. Redl, and P. Schüller, "A model building framework for answer set programming with external computations", *TPLP*, vol. 16, no. 4, pp. 418–464, 2016. DOI: 10.1017/S1471068415000113.

[42] T. Eiter, T. Geibinger, N. Higuera, and J. Oetsch, "A logic-based approach to contrastive explainability for neurosymbolic visual question answering", in *IJCAI23*, 2023, pp. 3668–3676. DOI: 10.24963/ijcai.2023/408.

[43] T. Eiter and G. Gottlob, "On the computational cost of disjunctive logic programming: Propositional case", *Ann. Math. Artif. Intell.*, vol. 15, no. 3, pp. 289–323, 1995, ISSN: 15737470. DOI: 10.1007/BF01536399.

[44] T. Eiter, G. Ianni, and T. Krennwallner, "Answer set programming: A primer", in *Reasoning Web. Semantic Technologies for Information Systems*, S. Tessaris, E. Franconi, T. Eiter, C. Gutierrez, S. Handschuh, M.-C. Rousset, and R. A. Schmidt, Eds., ser. Lecture Notes in Computer Science, vol. 5689, 2009, pp. 40–110. DOI: 10.1007/978-3-642-03754-2_2.

[45] T. Eiter and A. Polleres, "Towards automated integration of guess and check programs in answer set programming: A meta-interpreter and applications", in *LPNMR04*, ser. Lecture Notes in Computer Science, vol. 2923, 2004, pp. 100–113. DOI: 10.1007/978-3-540-24609-1_11.

[46] E. Erdem, V. Patoglu, and Z. G. Saribatur, "Integrating hybrid diagnostic reasoning in plan execution monitoring for cognitive factories with multiple robots", in *ICRA15*, Seattle, WA, USA: IEEE, 2015, pp. 2007–2013. DOI: 10.1109/ICRA.2015.7139461.

[47] W. Faber, N. Leone, and S. Perri, "The intelligent grounder of DLV", *Correct Reasoning*, Lecture Notes in Computer Science, vol. 7265, E. Erdem, J. Lee, Y. Lierler, and D. Pearce, Eds., pp. 247–264, 2012. DOI: 10.1007/978-3-642-30743-0_17.

[48] W. Faber and S. Woltran, "Manifold answer-set programs for meta-reasoning", in *LPNMR09*, E. Erdem, F. Lin, and T. Schaub, Eds., ser. Lecture Notes in Computer Science, vol. 5753, 2009, pp. 115–128. DOI: 10.1007/978-3-642-04238-6_12.

[49] A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, and E. C. Teppan, "Industrial applications of answer set programming", *Künstl. Intell.*, vol. 32, no. 2, pp. 165–176, 2018. DOI: 10.1007/s13218-018-0548-6.

[50] E. M. Fenoaltea, I. B. Baybusinov, J. Zhao, L. Zhou, and Y.-C. Zhang, "The stable marriage problem: An interdisciplinary review from the physicist's perspective", *Physics Reports*, vol. 917, pp. 1–79, 2021, ISSN: 03701573. DOI: 10.1016/j.physrep.2021.03.001.

[51] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving", *Artif. Intell.*, vol. 2, no. 3, pp. 189–208, 1971, ISSN: 00043702. DOI: 10.1016/0004-3702(71)90010-5.

[52] D. Gale and L. S. Shapley, "College admissions and the stability of marriage", *Am. Math. Mon.*, vol. 69, no. 1, pp. 9–15, 1962, ISSN: 19300972. DOI: 10.1080/00029890.1962.11989827.

[53] A. d. Garcez and L. C. Lamb, "Neurosymbolic AI: The 3rd wave", *Artif Intell Rev*, vol. 56, no. 11, pp. 12 387–12 406, 2023, ISSN: 15737462. DOI: 10.1007/s10462-023-10448-w.

[54] H. Garcia-Molina, J. Ullman, and J. Widom, *Database systems: the complete book*. Pearson; 2nd edition, 2008, 1203 pp., pp. 797–798, ISBN: 0131873253.

[55] M. Garnelo and M. Shanahan, "Reconciling deep learning with symbolic artificial intelligence: Representing objects and relations", *Curr. Opin. Behav. Sci.*, vol. 29, pp. 17–23, 2019, ISSN: 23521546. DOI: 10.1016/j.cobeha.2018.12.010.

[56] M. Gebser, A. Harrison, R. Kaminski, V. Lifschitz, and T. Schaub, "Abstract gringo", *TPLP*, vol. 15, no. 4, pp. 449–463, 2015. DOI: 10.1017/S1471068415000150.

[57] M. Gebser, T. Janhunen, M. Ostrowski, T. Schaub, and S. Thiele, "A versatile intermediate language for answer set programming", *NMR08*, pp. 150–159, 2008.

[58] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko, "Theory solving made easy with clingo 5", *ICLP16*, OASIcs, vol. 52, pp. 1–15, 2016, ISSN: 21906807. DOI: 10.4230/OASICS.ICLP.2016.2.

[59] M. Gebser, R. Kaminski, B. Kaufmann, J. Romero, and T. Schaub, "Progress in clasp series 3", in *LPNMR15*, F. Calimeri, G. Ianni, and M. Truszczynski, Eds., ser. Lecture Notes in Computer Science, vol. 9345, 2015, pp. 368–383. DOI: 10.1007/978-3-319-23264-5_31.

[60] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, "Multi-shot ASP solving with clingo", *TPLP*, vol. 19, no. 1, pp. 27–82, 2019. DOI: 10.1017/S1471068418000054.

[61] M. Gebser, R. Kaminski, and T. Schaub, "Grounding recursive aggregates: Preliminary report", *GTTV15*, M. Denecker and T. Janhunen, Eds., 2015. DOI: 10.48550/arXiv.1603.03884.

[62] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub, "Conflict-driven answer set solving", in *IJCAI07*, M. M. Veloso, Ed., 2007, pp. 386–392.

[63] M. Gebser, B. Kaufmann, and T. Schaub, "Conflict-driven answer set solving: From theory to practice", *Artif. Intell.*, vol. 187, pp. 52–89, 2012, ISSN: 00043702. DOI: 10.1016/j.artint.2012.04.001.

[64] M. Gebser, N. Leone, M. Maratea, S. Perri, F. Ricca, and T. Schaub, "Evaluation techniques and systems for answer set programming: A survey", in *IJCAI18*, 2018, pp. 5450–5456. DOI: 10.24963/ijcai.2018/769.

[65] M. Gebser, L. Liu, G. Namasivayam, A. Neumann, T. Schaub, and M. Truszczyński, "The first answer set programming system competition", in *LPNMR07*, C. Baral, G. Brewka, and J. Schlipf, Eds., ser. Lecture Notes in Computer Science, vol. 4483, 2007, pp. 3–17. DOI: 10.1007/978-3-540-72200-7_3.

[66] M. Gebser, M. Maratea, and F. Ricca, "The seventh answer set programming competition: Design and results", *TPLP*, vol. 20, no. 2, pp. 176–204, 2020. DOI: 10.1017/S1471068419000061.

[67] M. Gebser, M. Maratea, and F. Ricca, "The sixth answer set programming competition", *JAIR*, vol. 60, pp. 41–95, 2017, ISSN: 10769757. DOI: 10.1613/jair.5373.

[68] M. Gebser, M. Maratea, and F. Ricca, "What's hot in the answer set programming competition", in *AAAI*, vol. 30, 2016. DOI: 10.1609/aaai.v30i1.9872.

[69] M. Gebser, T. Schaub, and S. Thiele, "GrinGo: A new grounder for answer set programming", in *LPNMR07*, ser. Lecture Notes in Computer Science, vol. 4483, 2007, pp. 266–271. DOI: 10.1007/978-3-540-72200-7_24.

[70] M. Gebser, P. Wanko, R. Kaminski, B. kauf, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, and S. Thiele. "Potassco guide version 2.2.0". 120 pp., URL: https://github.com/potassco/guide/releases (accessed 21.01.2025). (2019).

[71] M. Gelfond and N. Leone, "Logic programming and knowledge representation—the a-prolog perspective", *Artif. Intell.*, vol. 138, no. 1, pp. 3–38, 2002, ISSN: 00043702. DOI: 10.1016/S0004-3702(02)00207-2.

[72] M. Gelfond and V. Lifschitz, "Classical negation in logic programs and disjunctive databases", *New. Gener. Comput.*, vol. 9, no. 3, pp. 365–385, 1991, ISSN: 18827055. DOI: 10.1007/BF03037169.

[73] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming", *JICSLP88*, R. Kowalski and K. Bowen, Eds., pp. 1070–1080, 1988. DOI: 10.2307/2275201.

[74] M. Gelfond, V. Lifschitz, H. Przymusinska, and M. Truszczynski, "Disjunctive defaults", in *KR91*, 1991, pp. 230–237.

[75] Y. N. Harari, *Sapiens: A Brief History of Humankind*. Harvill Secker, 2014, ISBN: 9781846558238.

[76] S. Harnad, "The symbol grounding problem", *Physica D: Nonlinear Phenomena*, vol. 42, no. 1, pp. 335–346, 1990, ISSN: 01672789. DOI: 10.1016/0167-2789(90)90087-6.

[77] N. Hippen and Y. Lierler, "Estimating grounding sizes of logic programs under answer set semantics", *JELIA21*, Lecture Notes in Computer Science, vol. 12678, W. Faber, G. Friedrich, M. Gebser, and M. Morak, Eds., pp. 346–361, 2021. DOI: 10.1007/978-3-030-75775-5_23.

[78] S. Hochreiter, "Toward a broad AI", *Commun. ACM*, vol. 65, no. 4, pp. 56–57, 2022, ISSN: 15577317. DOI: 10.1145/3512715.

[79] R. Irving, D. Manlove, and S. Scott, "The hospitals/residents problem with ties", in *SWAT00*, ser. Lecture Notes in Computer Science, vol. 1851, Jul. 2000, pp. 259–271. DOI: 10.1007/3-540-44985-X_24.

[80] T. Janhunen, "Implementing stable-unstable semantics with ASPTOOLS and clingo", *PADL22*, Lecture Notes in Computer Science, vol. 13165, J. Cheney and S. Perri, Eds., pp. 135–153, 2022. DOI: 10.1007/978-3-030-94479-7_9.

[81] T. Janhunen, "Intermediate languages of ASP systems and tools", *CEUR07*, pp. 12–25, 2007.

[82] T. Janhunen and I. Niemela, "The answer set programming paradigm", *AI Mag.*, vol. 37, no. 3, pp. 13–24, 2016. DOI: 10.1609/aimag.v37i3.2671.

[83] D. Kahneman, *Thinking, fast and slow.* (Thinking, fast and slow.). New York, NY, US: Farrar, Straus and Giroux, 2011, 499 pp., 499 pp., ISBN: 9780374275631.

[84] R. Kaminski, J. Romero, T. Schaub, and P. Wanko, "How to build your own ASP-based system?!", *TPLP*, vol. 23, no. 1, pp. 299–361, 2023, DOI: 10.1017/S1471068421000508.

[85] R. Kaminski and T. Schaub, "On the foundations of grounding in answer set programming", *TPLP*, vol. 23, no. 6, pp. 1138–1197, 2023, DOI: 10.1017/S1471068422000308.

[86] R. Kaminski, T. Schaub, and P. Wanko, "A tutorial on hybrid answer set solving with clingo", in *Reasoning Web. Semantic Interoperability on the Web*, G. Ianni, D. Lembo, L. Bertossi, W. Faber, B. Glimm, G. Gottlob, and S. Staab, Eds., ser. Lecture Notes in Computer Science, vol. 10370, Cham, 2017, pp. 167–203. DOI: 10.1007/978-3-319-61033-7_6.

[87] B. Kaufmann, N. Leone, S. Perri, and T. Schaub, "Grounding and solving in answer set programming", *AI Mag.*, vol. 37, no. 3, pp. 25–32, 2016, ISSN: 23719621. DOI: 10.1609/aimag.v37i3.2672.

[88] P. Körner, M. Leuschel, J. Barbosa, V. S. Costa, V. Dahl, M. V. Hermenegildo, J. F. Morales, J. Wielemaker, D. Diaz, S. Abreu, and G. Ciatto, "Fifty years of prolog and beyond", *TPLP*, vol. 22, no. 6, pp. 776–858, 2022. DOI: 10.1017/S1471068422000102.

[89] A. K. Lampinen, I. Dasgupta, S. C. Y. Chan, H. R. Sheahan, A. Creswell, D. Kumaran, J. L. McClelland, and F. Hill, "Language models, like humans, show content effects on reasoning tasks", *PNAS Nexus*, vol. 3, no. 7, D. Abbott, Ed., pgae233, 2024, ISSN: 27526542. DOI: 10.1093/pnasnexus/pgae233.

[90] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning", *Nature*, vol. 521, no. 7553, pp. 436–444, 2015, ISSN: 14764687. DOI: 10.1038/nature14539.

[91] C. Lefèvre and P. Nicolas, "The first version of a new ASP solver : ASPeRiX", in *LPNMR09*, E. Erdem, F. Lin, and T. Schaub, Eds., ser. Lecture Notes in Computer Science, vol. 5753, 2009, pp. 522–527. DOI: 10.1007/978-3-642-04238-6_52.

[92] N. Leone, S. Perri, and F. Scarcello, "Improving ASP instantiators by join-ordering methods", in *LPNMR01*, T. Eiter, W. Faber, and M. L. Truszczyński, Eds., ser. Lecture Notes in Computer Science, vol. 2173, 2001, pp. 280–294. DOI: 10.1007/3-540-45402-0_21.

[93] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The DLV system for knowledge representation and reasoning", *TOCL*, vol. 7, no. 3, pp. 499–562, 2006, ISSN: 15293785. DOI: 10.1145/1149114.1149117.

[94] L. Leutgeb and A. Weinzierl, "Techniques for efficient lazy-grounding ASP solving", *DECLARE18*, Lecture Notes in Computer Science, vol. 10997, D. Seipel, M. Hanus, and S. Abreu, Eds., pp. 132–148, 2018. DOI: 10.1007/978-3-030-00801-7_9.

[95] Y. Lierler, "Relating constraint answer set programming languages and algorithms", *Artif. Intell.*, vol. 207, pp. 1–22, 2014, ISSN: 00043702. DOI: 10.1016/j.artint.2013.10.004.

[96] Y. Lierler and M. Maratea, "Cmodels-2: SAT-based answer set solver enhanced to non-tight programs", in *LPNMR03*, V. Lifschitz and I. Niemelä, Eds., ser. Lecture Notes in Computer Science, vol. 2923, 2003, pp. 346–350. DOI: 10.1007/978-3-540-24609-1_32.

[97] Y. Lierler and J. Robbins, "DualGrounder: Lazy instantiation via clingo multi-shot framework", *JELIA21*, Lecture Notes in Computer Science, vol. 12678, W. Faber, G. Friedrich, M. Gebser, and M. Morak, Eds., pp. 435–441, 2021. DOI: 10.1007/978-3-030-75775-5_29.

[98] V. Lifschitz, "Thirteen definitions of a stable model", in *Fields of Logic and Computation*, A. Blass, N. Dershowitz, and W. Reisig, Eds., ser. Lecture Notes in Computer Science, vol. 6300, 2010, pp. 488–503. DOI: 10.1007/978-3-642-15025-8_24.

[99] F. Lin and Y. Zhao, "ASSAT: Computing answer sets of a logic program by SAT solvers", *Artif. Intell.*, vol. 157, no. 1, pp. 115–137, 2004, ISSN: 00043702. DOI: 10.1016/j.artint.2004.04.004.

[100] G. Liu, T. Janhunen, and I. Niemela, "Answer set programming via mixed integer programming", in *KR12*, G. Brewka and T. Eiter, Eds., 2012, pp. 32–42.

[101] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability", *IEEE Trans. Comput.*, vol. 48, no. 5, pp. 506–521, 1999, ISSN: 00189340. DOI: 10.1109/12.769433.

[102] E. Mastria, J. Zangari, S. Perri, and F. Calimeri, "A machine learning guided rewriting approach for ASP logic programs", *EPTCS*, vol. 325, F. Ricca, A. Russo, S. Greco, N. Leone, A. Artikis, G. Friedrich, P. Fodor, A. Kimmig, F. A. Lisi, M. Maratea, A. Mileo, and F. Riguzzi, Eds., pp. 261–267, 2020, ISSN: 20752180. DOI: 10.4204/EPTCS.325.31.

[103] G. Mazzotta, F. Ricca, and C. Dodaro, "Compilation of aggregates in ASP systems", in *AAAI22*, vol. 36, 2022, pp. 5834–5841. DOI: 10.1609/aaai.v36i5.20527.

[104] J. McCarthy, "The philosophy of AI and the AI of philosophy", in *Philosophy of Information*, ser. Handbook of the Philosophy of Science, P. Adriaans and J. van Benthem, Eds., 2008, pp. 711–740, ISBN: 9780444517265. DOI: 10.1016/B978-0-444-51726-5.50022-4.

[105] J. P. McDermott, "RI: An expert in the computer systems domain", in *AAAI80*, R. Balzer, Ed., 1980, pp. 269–271.

[106] T. Miller, "Explanation in artificial intelligence: Insights from the social sciences", *Artif. Intel.*, vol. 267, pp. 1–38, 2019, ISSN: 00043702. DOI: 10.1016/j.artint.2018.07.007.

[107] M. Morak and S. Woltran, "Preprocessing of complex non-ground rules in answer set programming", *ICLP12*, LIPIcs, vol. 17, pp. 247–258, 2012, ISSN: 18688969. DOI: 10.4230/LIPICS.ICLP.2012.247.

[108] L. de Moura and N. Bjorner, "Z3: An efficient SMT solver", *TACAS08*, Lecture Notes in Computer Science, vol. 4963, C. R. Ramakrishnan and J. Rehof, Eds., pp. 337–340, 2008. DOI: 10.1007/978-3-540-78800-3_24.

[109] A.-D. Nguyen, L. Pham, N. Lindsay, L. Sun, and S. C. Tran, "Optimized eVTOL aircraft scheduling - an answer set programming based approach", in *AIAA24*, 2024. DOI: 10.2514/6.2024-3939.

[110] Nobel Prize Outreach AB 2024, *The prize in economic sciences*, 2012.

[111] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry, "An a-prolog decision support system for the space shuttle", in *PADL01*, I. V. Ramakrishnan, Ed., ser. Lecture Notes in Computer Science, vol. 1990, 2001, pp. 169–183. DOI: 10.1007/3-540-45241-9_12.

[112] F. Olivier and C. Schultz, "Here-and-there with constraints for spatial reasoning", *TAASP23*, 2023.

[113] M. Ostrowski and T. Schaub, "ASP modulo CSP: The clingcon system", *TPLP*, vol. 12, no. 4, pp. 485–503, 2012. DOI: 10.1017/S1471068412000142.

[114] C. H. Papadimitriou, *Computational complexity*. Addison-Wesley, 1994, 523 pp., ISBN: 9780201530827.

[115] D. Pearce, "A new logical characterisation of stable models and answer sets", *NMELP96*, Lecture Notes in Computer Science, vol. 1216, J. Dix, L. M. Pereira, and T. C. Przymusinski, Eds., pp. 57–70, 1996. DOI: 10.1007/BFb0023801.

[116] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010, p. 2.

[117] D. Sacca and C. Zaniolo, "Stable models and non-determinism in logic programs with negation", in *PODS90*, D. J. Rosenkrantz and Y. Sagiv, Eds., Nashville Tennessee USA, 1990, pp. 205–217. DOI: 10.1145/298514.298572.

[118] R. M. Sapolsky, *Behave: The Biology of Humans at Our Best and Worst*. Penguin Publishing Group, 2017, ISBN: 9780735222786.

[119] T. Schaub and S. Woltran, "Special issue on answer set programming", *Künstliche Intell.*, vol. 32, no. 2, pp. 101–103, 2018, ISSN: 16101987. DOI: 10.1007/s13218-018-0554-8.

[120] M. Shanahan, "Talking about large language models", *Commun. ACM*, vol. 67, no. 2, pp. 68–79, 2024, ISSN: 15577317. DOI: 10.1145/3624724.

[121] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play", *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018. DOI: 10.1126/science.aar6404.

[122] P. Simons, I. Niemelä, and T. Soininen, "Extending and implementing the stable model semantics", *Artif. Intell.*, vol. 138, no. 1, pp. 181–234, 2002, ISSN: 00043702. DOI: 10.1016/S0004-3702(02)00187-X.

[123] B. Susman and Y. Lierler, "System description: SMT-based constraint answer set solver EZSMT", *ICLP16*, OASIcs, vol. 52, M. Carro, A. King, N. Saeedloei, and M. De Vos, Eds., pp. 1–15, 2016, ISSN: 21906807. DOI: 10.4230/OASIcs.ICLP.2016.1.

[124] T. Syrjänen. "LParse 1.0 user's manual". 99 pp., URL: https://www.tcs.hut.fi/Software/smodels/lparse.ps (accessed 15.01.2025). (2000).

[125] T. Syrjänen, "Omega-restricted logic programs", in *LPNMR01*, T. Eiter, W. Faber, and M. L. Truszczyński, Eds., ser. Lecture Notes in Computer Science, vol. 2173, 2001, pp. 267–280. DOI: 10.1007/3-540-45402-0_20.

[126] E. Tsamoura, V. Gutierrez-Basulto, and A. Kimmig, "Beyond the grounding bottleneck: Datalog techniques for inference in probabilistic logic programs", in *AAAI20*, vol. 34, 2020, pp. 10 284–10 291. DOI: 10.1609/aaai.v34i06.6591.

[127] K. Unalan, "Body-decoupled grounding in normal answer set programs", Bachelor's Thesis, TUWien, 2022.

[128] L. G. Valiant, "Three problems in computer science", *J. ACM*, vol. 50, no. 1, pp. 96–99, 2003, ISSN: 1557735X. DOI: 10.1145/602382.602410.

[129] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need", *NeurIPS17*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., pp. 5998–6008, 2017.

[130] A. Weinzierl, "Blending lazy-grounding and CDNL search for answer-set solving", in *LPNMR17*, M. Balduccini and T. Janhunen, Eds., ser. Lecture Notes in Computer Science, vol. 10377, Cham: Springer International Publishing, 2017, pp. 191–204. DOI: 10.1007/978-3-319-61660-5_17.

[131] A. Weinzierl, R. Taupe, and G. Friedrich, "Advancing lazy-grounding ASP solving techniques – restarts, phase saving, heuristics, and more", *TPLP*, vol. 20, no. 5, pp. 609–624, 2020, ISSN: 14710684. DOI: 10.1017/S1471068420000332.

[132] H. Werthner, C. Ghezzi, J. Kramer, J. Nida-Rümelin, B. Nuseibeh, E. Prem, and A. Stanger, Eds., *Introduction to Digital Humanism: A Textbook*, Cham: Springer Nature Switzerland, 2024, ISBN: 9783031453038. DOI: 10.1007/978-3-031-45304-5.

[133] Z. Yang, A. Ishay, and J. Lee, "NeurASP: Embracing neural networks into answer set programming", in *IJCAI20*, 2020, pp. 1755–1762. DOI: 10.24963/ijcai.2020/243.

Appendix

Additional Implementation Details

Our prototype is available under: `https://github.com/alexl4123/newground`.

Example 8.1. *The following example demonstrates the usage of the* `clingo` *transformer. It showcases a minimum example, by printing the used rules, and saving all variables in the program. The transformer is called with the program:*

```
a(1).a(2).b(X) :- a(X),a(Y).
```

Note that the following code is PYTHON *code. It produces the following output:*

```
> python transformer_example.py
a(1).
a(2).
b(X) :- a(X); a(Y).
All variables in program: {'X': True, 'Y': True}
```

```
from clingo.ast import Transformer

class MyTransformer(Transformer):

    def __init__(self):
        self.my_variables = {}

    def visit_Rule(self, node):
        self.visit_children(node)
        print(str(node))
        return node

    def visit_Variable(self, node):
        self.visit_children(node)
        if str(node) not in self.my_variables:
            self.my_variables[str(node)] = True
        return node
```

Listing 8.1: Transformer Class Example I. Showcasing the usage of a transformer - implementing the transformer.

```python
from clingo.ast import parse_string

if __name__ == "__main__":
    my_program = "a(1).a(2).b(X) :- a(X),a(Y)."
    my_transformer = MyTransformer()
    parse_string(my_program, lambda stm: my_transformer(stm))
    print(f"All variables in program: {my_transformer.my_variables}")
```

Listing 8.2: Transformer Class Example Part II. Showcasing the usage of a transformer - calling the transformer.

How BDG dismantles Cycles: Full Grounding

In Section 7.1 we showed for a program Π its non-ground, and grounded $\mathcal{HG}(\Pi_1, \Pi_2)$ versions, restricted to $\mathrm{at}(\mathcal{G}(\Pi)) \cup \{h' | h \in H(\Pi_1)\}$. The listing below shows the necessary non-ground auxiliary atoms.

```
%%%%% q(X,Y) :- f(X,Y). %%%%%%
sat_0:-sat_0_X(X1),sat_0_Y(X2),f(X1,X2).
sat_0:-sat_0_Y(X1),sat_0_X(X2),not q(X1,X2).
sat_0_Y(1):-sat. sat_0_Y(2):-sat. sat_0_Y(1)|sat_0_Y(2).
sat_0_X(1):-sat. sat_0_X(2):-sat. sat_0_X(1)|sat_0_X(2).
ujust_0_X_Y(X1,X2):-f'(X1,X2),not q(X2,X1).
:-0 < #count{1:ujust_0_X_Y(X1,X2)},f'(X1,X2).
%%%%%% f(X,Y) :- q(Y,X). %%%%%%%
%% HEAD GUESSES %%
%% SAT PART %%
sat_1:-sat_1_X(X1),sat_1_Y(X2),q(X1,X2).
sat_1:-sat_1_X(X1),sat_1_Y(X2),not f(X1,X2).
sat_1_X(1):-sat. sat_1_X(2):-sat. sat_1_X(1)|sat_1_X(2).
sat_1_Y(1):-sat. sat_1_Y(2):-sat. sat_1_Y(1)|sat_1_Y(2).
ujust_1_X_Y(X1,X2):-q'(X1,X2),not f(X1,X2).
:-0 < #count{1:ujust_1_X_Y(X1,X2)},q'(X1,X2).
:- not sat.
sat:-sat_0,sat_1.
```

And this listing shows the necessary ground auxiliary atoms.

```
sat_1_Y(1);sat_1_Y(2).
sat_1_X(1);sat_1_X(2).
sat_0_Y(1);sat_0_Y(2).
sat_0_X(1);sat_0_X(2).
sat_0:-sat_0_X(2),sat_0_Y(1),not q(1,2).
sat_0:-sat_0_X(2),sat_0_Y(2),not q(2,2).
sat_0:-sat_0_X(1),sat_0_Y(1),not q(1,1).
sat_0:-sat_0_X(1),sat_0_Y(2),not q(2,1).
sat_0:-f(1,2),sat_0_Y(2),sat_0_X(1).
sat_0:-f(2,2),sat_0_Y(2),sat_0_X(2).
sat_0:-f(1,1),sat_0_Y(1),sat_0_X(1).
sat_0:-f(2,1),sat_0_Y(1),sat_0_X(2).
sat_1:-sat_1_Y(2),sat_1_X(1),not f(1,2).
sat_1:-sat_1_Y(2),sat_1_X(2),not f(2,2).
sat_1:-sat_1_Y(1),sat_1_X(1),not f(1,1).
sat_1:-sat_1_Y(1),sat_1_X(2),not f(2,1).
sat_1:-q(1,2),sat_1_Y(2),sat_1_X(1).
sat_1:-q(2,2),sat_1_Y(2),sat_1_X(2).
sat_1:-q(1,1),sat_1_Y(1),sat_1_X(1).
sat_1:-q(2,1),sat_1_Y(1),sat_1_X(2).
sat:-sat_1,sat_0.
sat_0_X(2):-sat.
sat_0_X(1):-sat.
sat_0_Y(2):-sat.
sat_0_Y(1):-sat.
sat_1_X(2):-sat.
```

```
27 sat_1_X(1):-sat.
28 sat_1_Y(2):-sat.
29 sat_1_Y(1):-sat.
30 :-not sat.
31 ujust_1_X_Y(1,1):-q'(1,1),not f(1,1).
32 ujust_1_X_Y(2,1):-q'(2,1),not f(2,1).
33 ujust_1_X_Y(1,2):-q'(1,2),not f(1,2).
34 ujust_1_X_Y(2,2):-q'(2,2),not f(2,2).
35 :-q'(1,1),0<#count{1:ujust_1_X_Y(1,1)}.
36 :-q'(2,1),0<#count{1:ujust_1_X_Y(2,1)}.
37 :-q'(1,2),0<#count{1:ujust_1_X_Y(1,2)}.
38 :-q'(2,2),0<#count{1:ujust_1_X_Y(2,2)}.
39 ujust_0_X_Y(1,1):-f'(1,1),not q(1,1).
40 ujust_0_X_Y(2,1):-f'(2,1),not q(1,2).
41 ujust_0_X_Y(1,2):-f'(1,2),not q(2,1).
42 ujust_0_X_Y(2,2):-f'(2,2),not q(2,2).
43 :-f'(1,1),0<#count{1:ujust_0_X_Y(1,1)}.
44 :-f'(2,1),0<#count{1:ujust_0_X_Y(2,1)}.
45 :-f'(1,2),0<#count{1:ujust_0_X_Y(1,2)}.
46 :-f'(2,2),0<#count{1:ujust_0_X_Y(2,2)}.
```

Additional Experimental Details

We show additional experimental details for the automated hybrid grounding experiments from Chapter 5. In Figures 2–7 we show additional solving profiles. In Figures 8–13 we show additional grounding profiles. In Table 3 we show overall time and memory usage for solving, while in Table 4 we show overall time ane memory usage for grounding. Tables 5 and 6 show detailed results of FastFound (Section 6.4). Table 7 shows the detailed results for Lazy-BDG (Section 7.5). Note that we think that the solving behavior of `gringo` for (12-Strat.) is an anomaly, as it was not able to solve any instance, whereas NG-G and NG-G-TW solve 12 - however neither `Lpopt` nor BDG was used by the heuristic.

Statistics for H1 of Section 5.4.4

We performed a *Mann-Whitney U test* (for `gringo` and NG-G) to put additional weight to our claim that H1 holds (or cannot be rejected), with an $H1_0$ of both distributions being equal. We use it to test whether there is a difference between the two populations (`gringo` and NG-G; $\alpha = 0.01$). As each scenario has a different number of instances, we account for this by adjusting each scenarios number of instances to 20 instances. 20 was arbitrarily chosen and different values do not have an effect. We obtain the same results for e.g., 100. Performing the two-sided *Mann-Whitney U test* we obtain a *p-value* of 0.89, and $0.89 > 0.01$. As we cannot reject $H1_0$, we also cannot reject H1. Therefore, our hypothesis that `newground3` with our Algorithm 5.3 does neither perform worse, nor better than ground-and-solve systems on solving-heavy benchmarks can not be rejected, which is exactly what we wanted to show.

Solving-Heavy scenarios: Usage of BDG and `Lpopt`

`Lpopt` was used for: (01-Perm), (01-Perm.-New), (04-Conn.), (06-Bottl.), (07-Nomi.), (07-Nomi.-New), (09-Rico.), (09-Rico.-New), (10-Cross.), (10-Cross.-New), (14-Weigh.), (14-Weigh.-New), (15-Stabl.), (15-Stabl.-New), (16-Incr.), (16-Incr.-New), (19-Abst.), (19-Abst.-New), (24-Lab.), (24-Lab.-New), (25-Min.), (25-Min.-New), (26-Hanoi.), and (28-Part.).

BDG was marked (structurally) for: (22-Knight.), (28-Part.).

BDG was used for: (28-Part.).

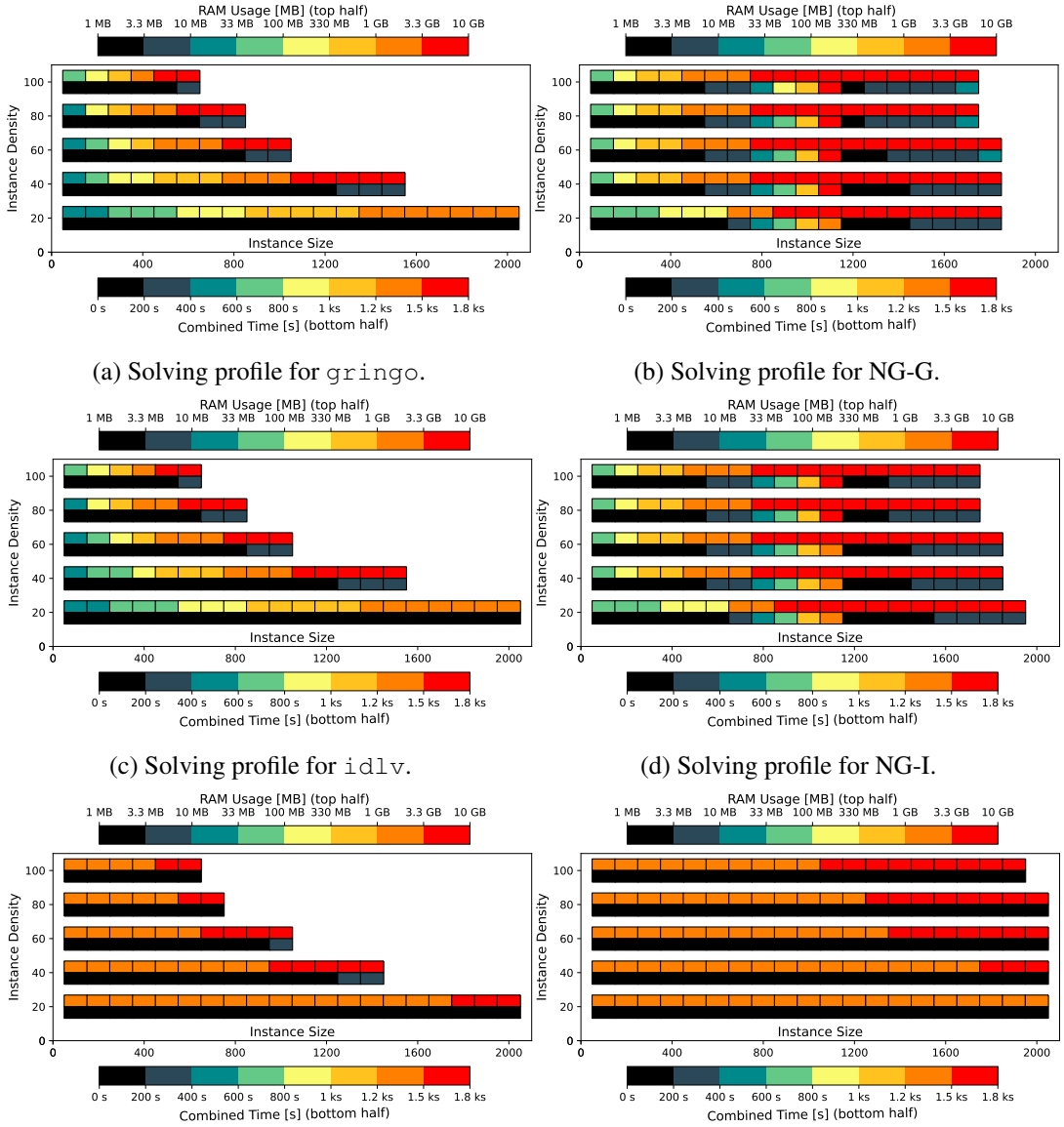
Grounding-Heavy scenarios: Usage of BDG and `Lpopt`

For the grounding-heavy scenarios we did not benchmark NG-G-TW separately, as for most scenarios the variable graph of the rule responsible for the grounding bottleneck is a clique. BDG was used for scenarios: (30-Cliq.), (31-Cliqu.(!=)), (32-Path.), (33-Col.), (34-4Cliqu.), (36-S3T4), (37-S4T4), (38-NPRC), (39-SM-Agg).

BDG was not used for: (35-NPRC).

Additional Plots and Tables

The following list shows full names for a problem from the 2014 ASP competition with its used abbrevation:
01N-PermutationPatternMatching-N01-New (01-Perm.-New), 02N-ValvesLocationProblem-N02 (02-Valv.), 04N-ConnectedMaximumDensityStillLife-N04 (04-Conn.), 04N-ConnectedMaximumDensityStillLife-N04-New (04-Conn.-New), 05N-GracefulGraphs-N05 (05-Grac.), 05N-GracefulGraphs-N05-New (05-Grac.-New), 06N-BottleFillingProblem-N06 (06-Bottl.), 06N-BottleFillingProblem-N06-New (06-Bottl.-New), 07N-Nomistery-N07 (07-Nomi.), 07N-Nomistery-N07-New (07-Nomi.-New), 08N-Sokoban-N08 (08-Sok.), 08N-Sokoban-N08-New (08-Sok.-New), 09N-RicochetRobots-N09 (09-Rico.), 09N-RicochetRobots-N09-New (09-Rico.-New), 10O-CrossingMinimization-O10 (10-Cross.), 10O-CrossingMinimization-O10-New (10-Cross.-New), 12N-StrategicCompanies-N12 (12-Strat.), 13O-Solitaire-O13 (13-Sol.), 13O-Solitaire-O13-New (13-Sol.-New), 14O-WeightedSequenceProblem-O14 (14-Weigh.), 14O-WeightedSequenceProblem-O14-New (14-Weigh.-New), 15O-StableMarriage-O15 (15-Stabl.), 16O-IncrementalScheduling-O16 (16-Incr.), 17N-QualitativeSpatialReasoning-N17-New (17-Qual.-New), 19N-AbstractDialecticalFrameworksWell-N19 (19-Abst.), 19N-AbstractDialecticalFrameworksWell-N19-New (19-Abst.-New), 20N-VisitAll-N20 (20-Visit.), 20N-VisitAll-N20-New (20-Visit.-New), 21N-ComplexOptimizationOfAnswerSets-N21 (21-Compl.), 22N-KnightTourWithHoles-N22-New (22-Knight.-New), 23O-MaximalCliqueProblem-O23-New (23-Max.-New), 24O-Labyrinth-O24 (24-Lab.), 24O-Labyrinth-O24-New (24-Lab.-New), 25O-MinimalDiagnosis-O25 (25-Min.), 25O-MinimalDiagnosis-O25-New (25-Min.-New), 26O-HanoiTower-O26 (26-Hanoi.), 26O-HanoiTower-O26-New (26-Hanoi.-New), 27O-GraphColoring-O27 (27-Graph.), 27O-GraphColoring-O27-New (27-Graph.-New), 28-PartnerUnits-28 (28-Part.), 28-PartnerUnits-28-New (28-Part.-New),



(a) Solving profile for `gringo`.

(b) Solving profile for NG-G.

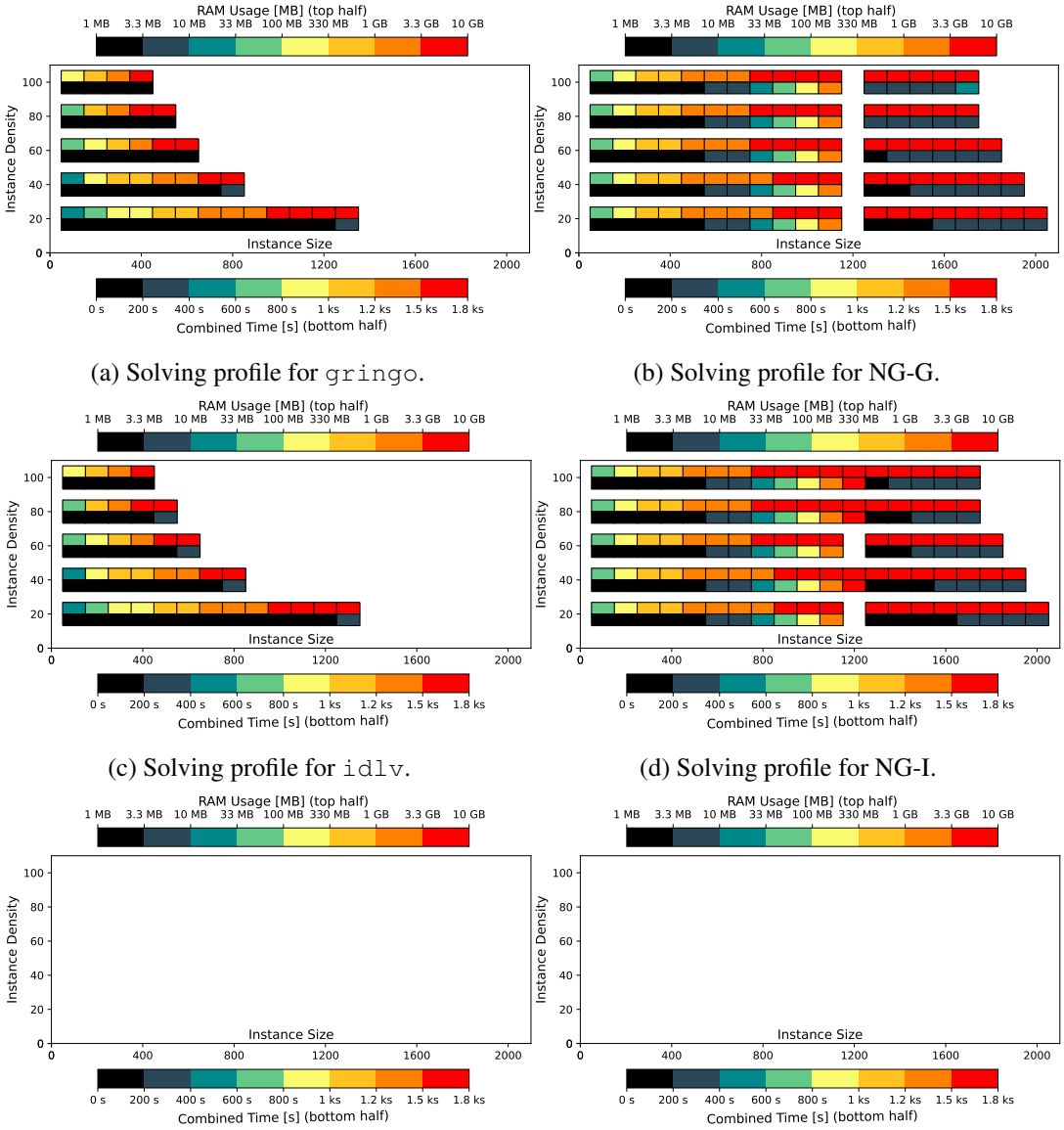
(c) Solving profile for `idlv`.

(d) Solving profile for NG-I.

(e) Solving profile for ProASP

(f) Solving profile for ProASP-CS.

Figure 2: Solving profiles for all systems for the *30-Cliqu.* example. Compare the traditional ground-and-solve systems (Figure 2a, and Figure 2c), to the `newground3` approaches (Figure 2b, and Figure 2d), to the ProASP approaches ((Figure 2e, and Figure 2f), Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `idlv` (NG-I), ProASP (ProASP), and ProASP with CS (ProASP-CS). Timeout: 1800s; Memout: 10GB.



(a) Solving profile for `gringo`.

(b) Solving profile for NG-G.

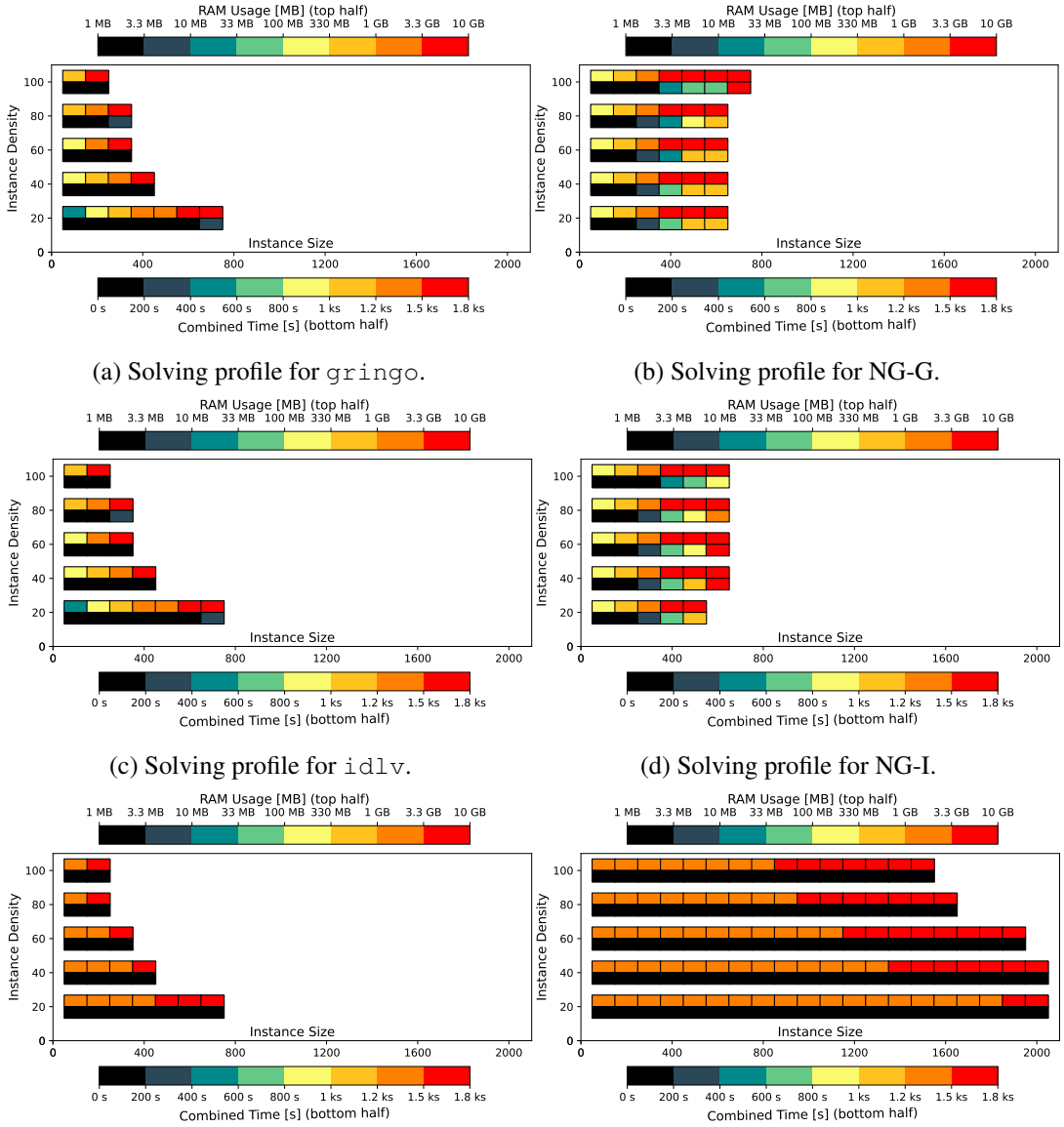
(c) Solving profile for `idlv`.

(d) Solving profile for NG-I.

(e) Solving profile for ProASP

(f) Solving profile for ProASP-CS.

Figure 3: Solving profiles for all systems for the *32-Path.* example. Compare the traditional ground-and-solve systems (Figure 3a, and Figure 3c), to the `newground3` approaches (Figure 3b, and Figure 3d), to the ProASP approaches ((Figure 3e, and Figure 3f), Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `idlv` (NG-I), ProASP (ProASP), and ProASP with CS (ProASP-CS). Timeout: 1800s; Memout: 10GB.



(a) Solving profile for `gringo`.

(b) Solving profile for NG-G.

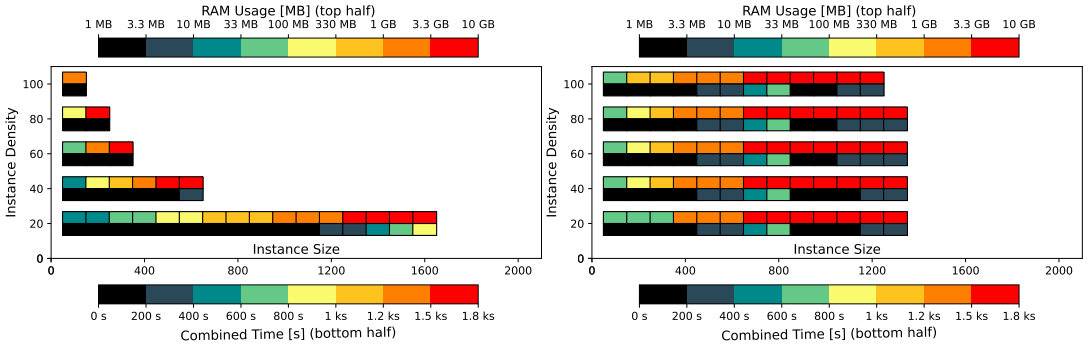
(c) Solving profile for `idlv`.

(d) Solving profile for NG-I.

(e) Solving profile for ProASP

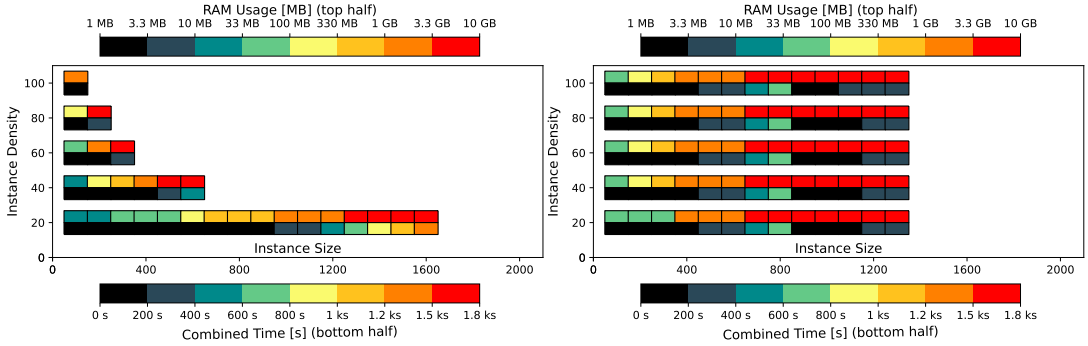
(f) Solving profile for ProASP-CS.

Figure 4: Solving profiles for all systems for the *33-Col.* example. Compare the traditional ground-and-solve systems (Figure 4a, and Figure 4c), to the `newground3` approaches (Figure 4b, and Figure 4d), to the ProASP approaches ((Figure 4e, and Figure 4f), Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `idlv` (NG-I), ProASP (ProASP), and ProASP with CS (ProASP-CS). Timeout: 1800s; Memout: 10GB.



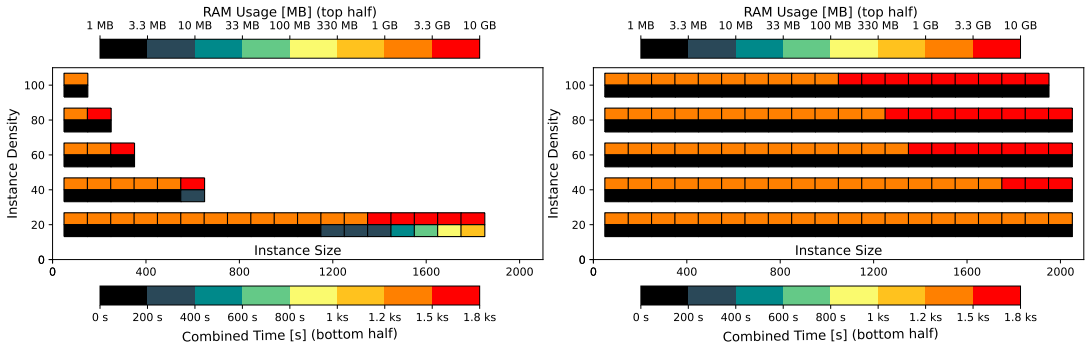
(a) Solving profile for `gringo`.

(b) Solving profile for NG-G.



(c) Solving profile for `idlv`.

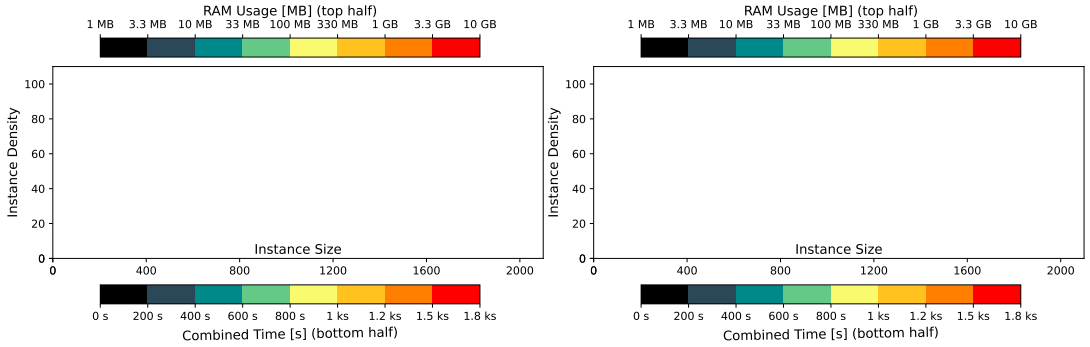
(d) Solving profile for NG-I.



(e) Solving profile for ProASP

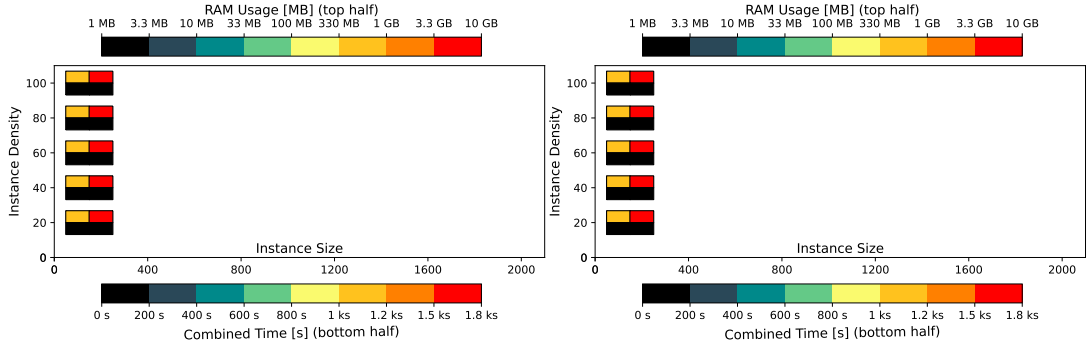
(f) Solving profile for ProASP-CS.

Figure 5: Solving profiles for all systems for the *34-Cliqu4.* example. Compare the traditional ground-and-solve systems (Figure 5a, and Figure 5c), to the `newground3` approaches (Figure 5b, and Figure 5d), to the ProASP approaches ((Figure 5e, and Figure 5f), Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `idlv` (NG-I), ProASP (ProASP), and ProASP with CS (ProASP-CS). Timeout: 1800s; Memout: 10GB.



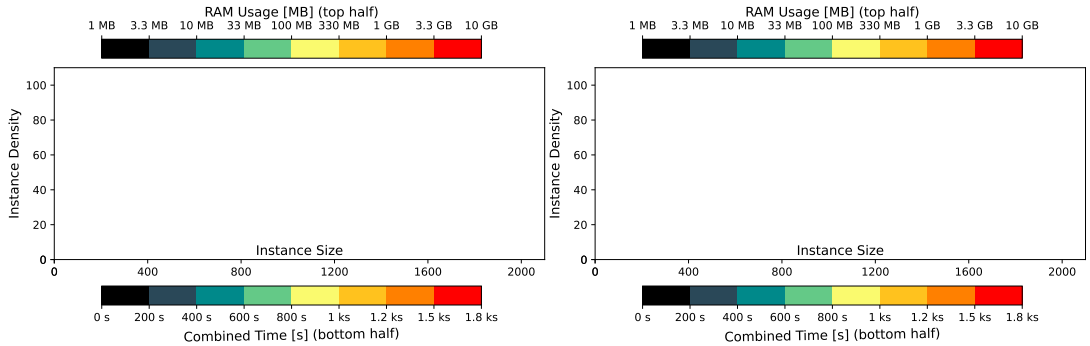
(a) Solving profile for `gringo`.

(b) Solving profile for NG-G.



(c) Solving profile for `idlv`.

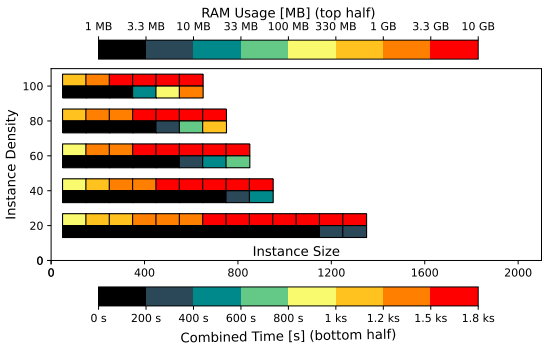
(d) Solving profile for NG-I.



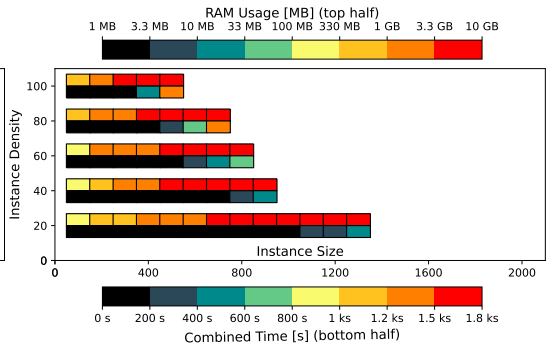
(e) Solving profile for ProASP

(f) Solving profile for ProASP-CS.

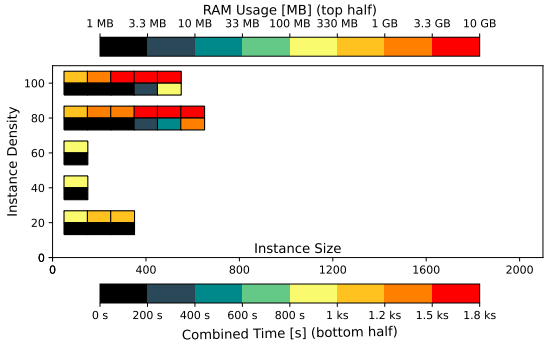
Figure 6: Solving profiles for all systems for the *35-NPRC.* example. Compare the traditional ground-and-solve systems (Figure 6a, and Figure 6c), to the `newground3` approaches (Figure 6b, and Figure 6d), to the ProASP approaches ((Figure 6e, and Figure 6f), Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `idlv` (NG-I), ProASP (ProASP), and ProASP with CS (ProASP-CS). Timeout: 1800s; Memout: 10GB.



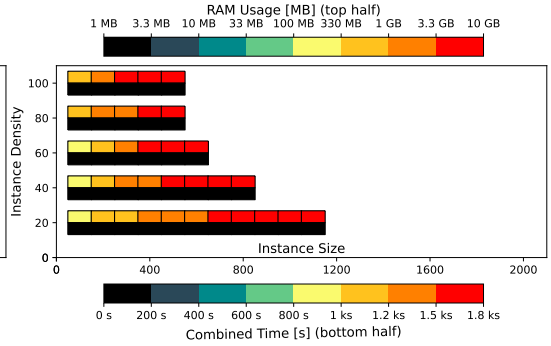
(a) *30-Cliqu.* solving profile for ALPHA.



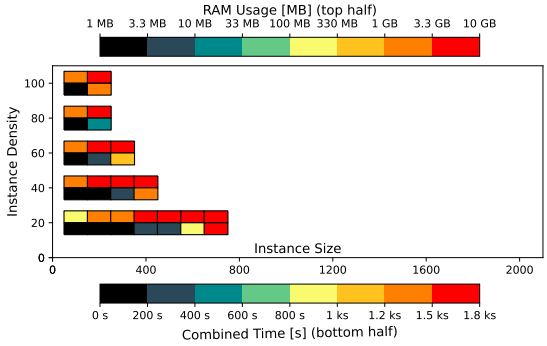
(b) *31-Cliqu.(!=)* solving profile for ALPHA.



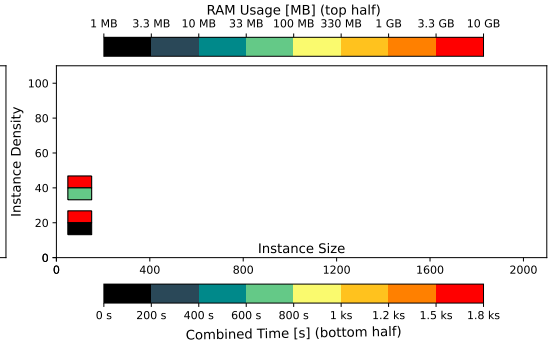
(c) *32-Path.* solving profile for ALPHA.



(d) *33-Col.* solving profile for ALPHA.

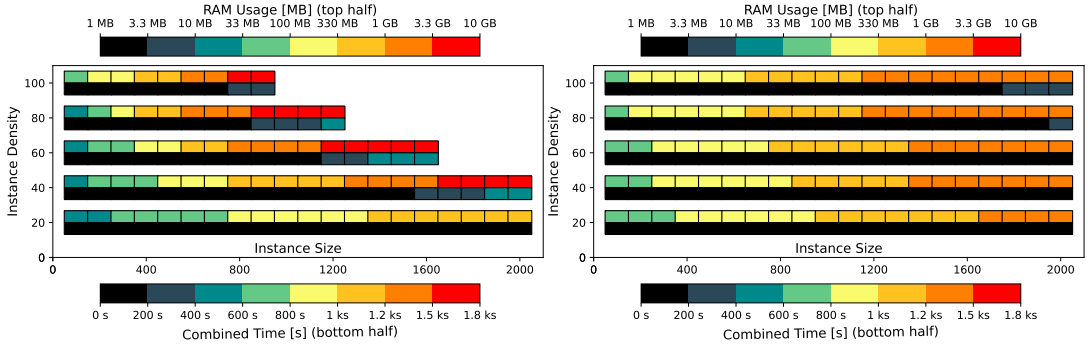


(e) *34-Cliqu4.* solving profile for ALPHA.



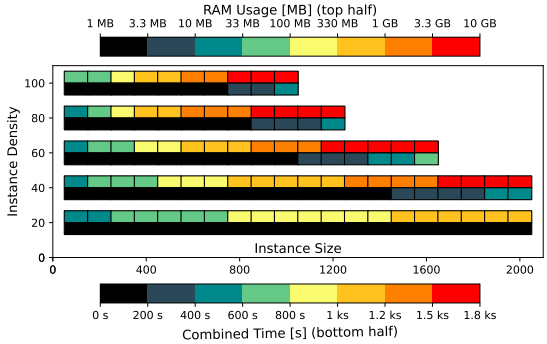
(f) *35-NPRC.* solving profile for ALPHA.

Figure 7: Solving profiles for the ALPHA system. Scenario *30-Cliqu.* in Figure 7a, scenario *31-Cliqu(!=).* in Figure 7b, scenario *32-Path.* in Figure 7c, scenario *33-Col.* in Figure 7d, scenario *34-Cliqu4* in Figure 7e, and scenario *35-NPRC* in Figure 7f. Timeout: 1800s; Memout: 10GB.

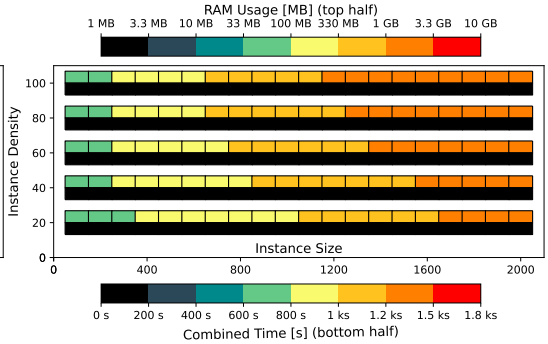


(a) Grounding profile for `gringo`.

(b) Grounding profile for NG-G.

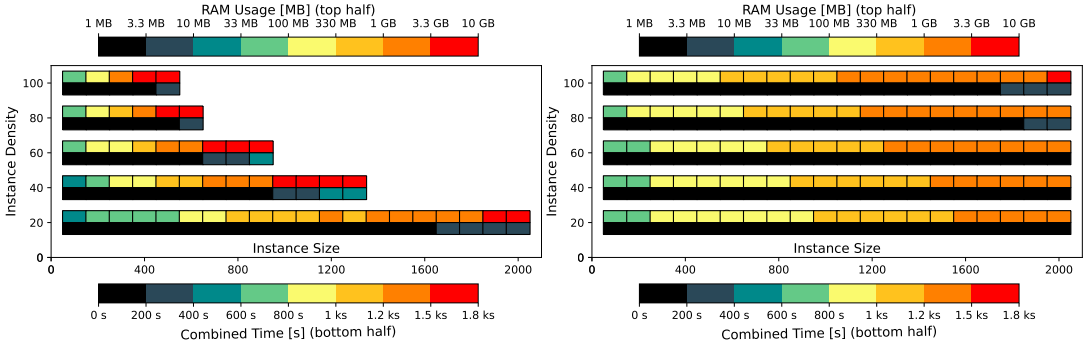


(c) Grounding profile for `idlv`.



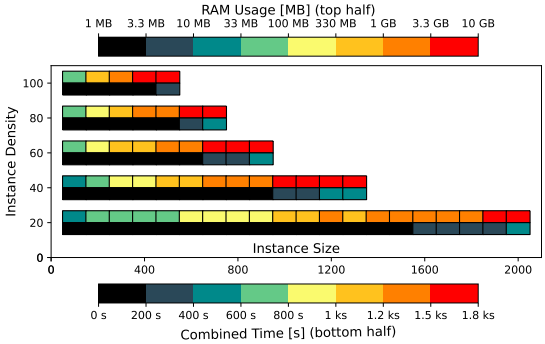
(d) Grounding profile for NG-I.

Figure 8: Grounding profiles for all ground-and-solve systems for the *30-Cliqu.* example. Compare the traditional ground-and-solve systems (Figure 8a, and Figure 8c), to the `newground3` approaches (Figure 8b, and Figure 8d). Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), and `newground3` with `idlv` (NG-I). Timeout: 1800s; Memout: 10GB.

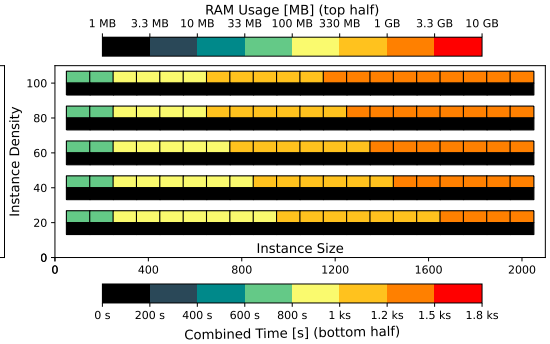


(a) Grounding profile for `gringo`.

(b) Grounding profile for NG-G.

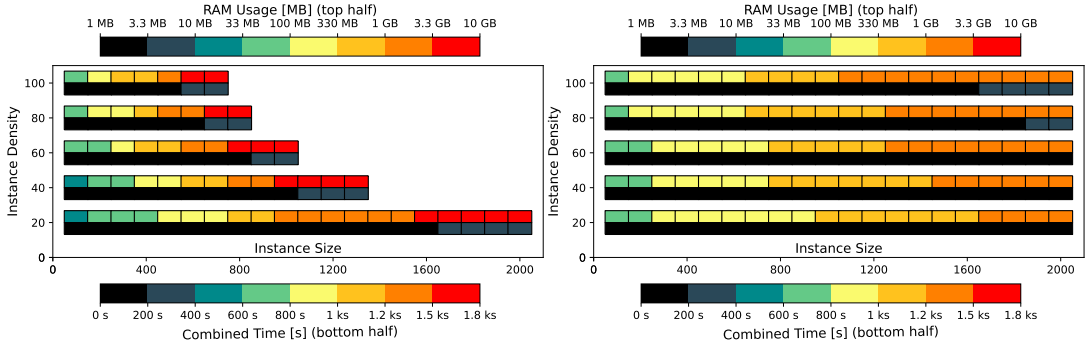


(c) Grounding profile for `idlv`.



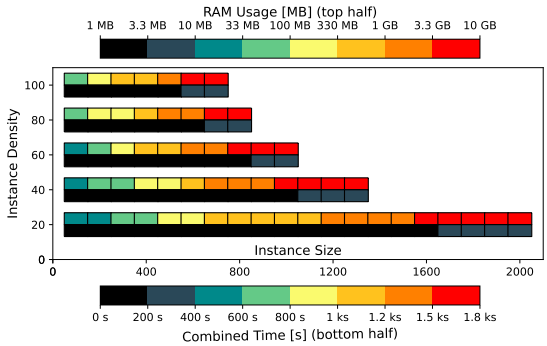
(d) Grounding profile for NG-I.

Figure 9: Grounding profiles for all ground-and-solve systems for the *31-Cliqu.(!=)* example. Compare the traditional ground-and-solve systems (Figure 9a, and Figure 9c), to the `newground3` approaches (Figure 9b, and Figure 9d). Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), and `newground3` with `idlv` (NG-I). Timeout: 1800s; Memout: 10GB.

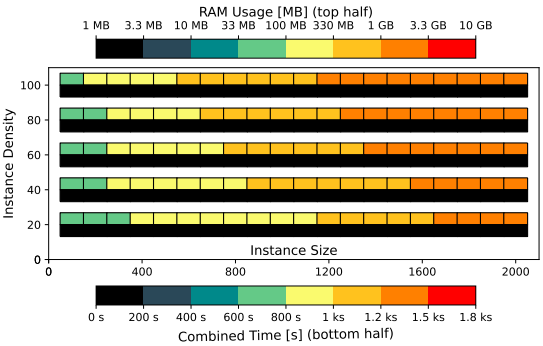


(a) Grounding profile for `gringo`.

(b) Grounding profile for NG-G.

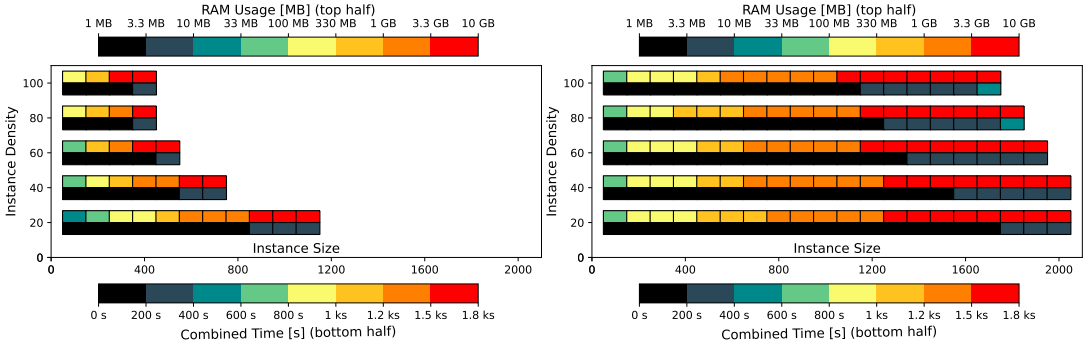


(c) Grounding profile for `idlv`.



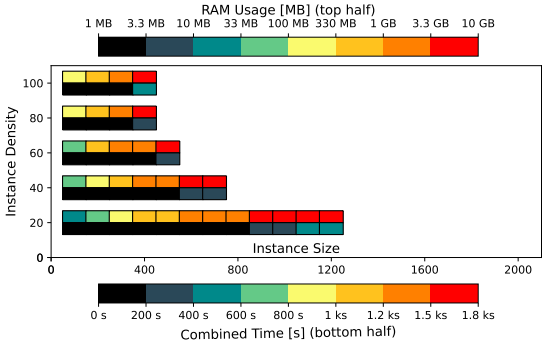
(d) Grounding profile for NG-I.

Figure 10: Grounding profiles for all ground-and-solve systems for the *32-Path.* example. Compare the traditional ground-and-solve systems (Figure 10a, and Figure 10c), to the `newground3` approaches (Figure 10b, and Figure 10d). Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), and `newground3` with `idlv` (NG-I). Timeout: 1800s; Memout: 10GB.

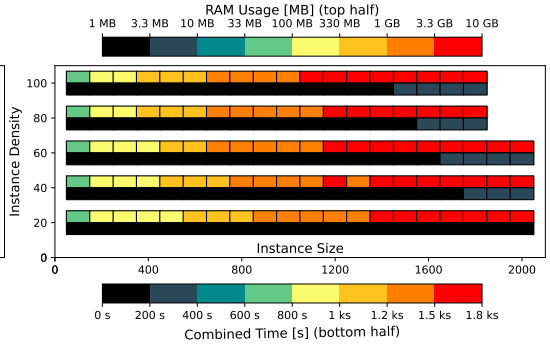


(a) Grounding profile for `gringo`.

(b) Grounding profile for NG-G.

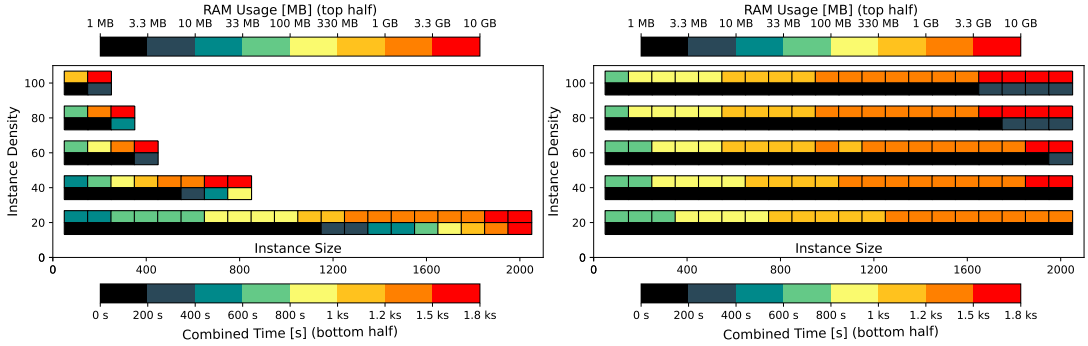


(c) Grounding profile for `idlv`.



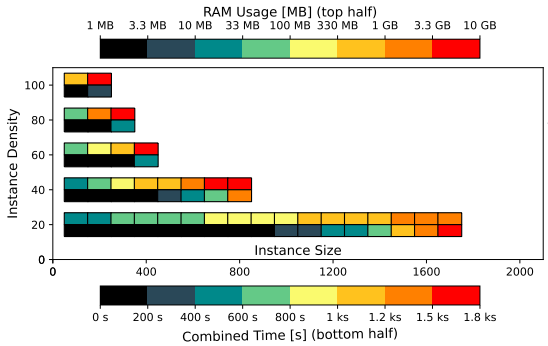
(d) Grounding profile for NG-I.

Figure 11: Grounding profiles for all ground-and-solve systems for the *33-Col.* example. Compare the traditional ground-and-solve systems (Figure 11a, and Figure 11c), to the `newground3` approaches (Figure 11b, and Figure 11d). Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), and `newground3` with `idlv` (NG-I). Timeout: 1800s; Memout: 10GB.

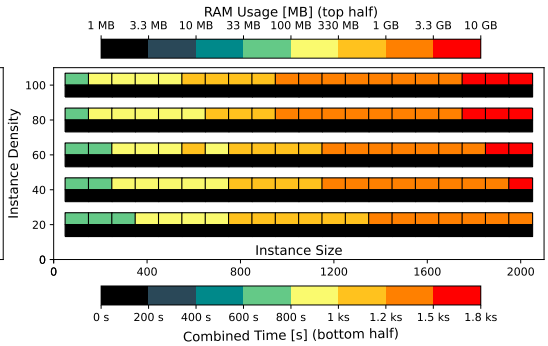


(a) Grounding profile for `gringo`.

(b) Grounding profile for NG-G.

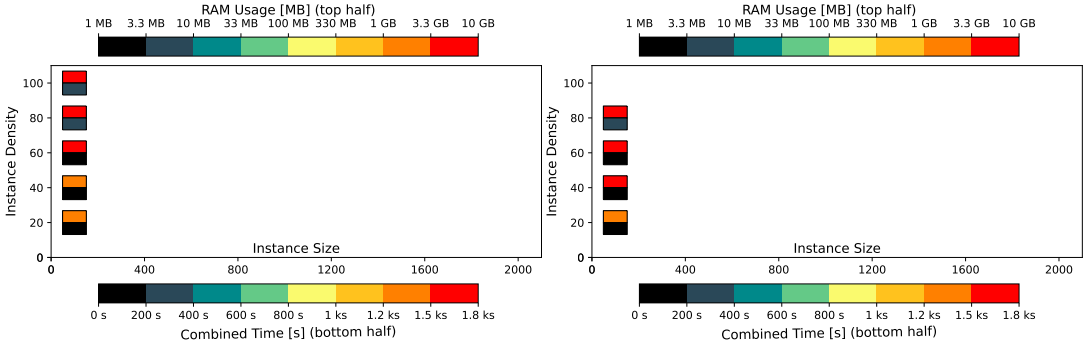


(c) Grounding profile for `idlv`.



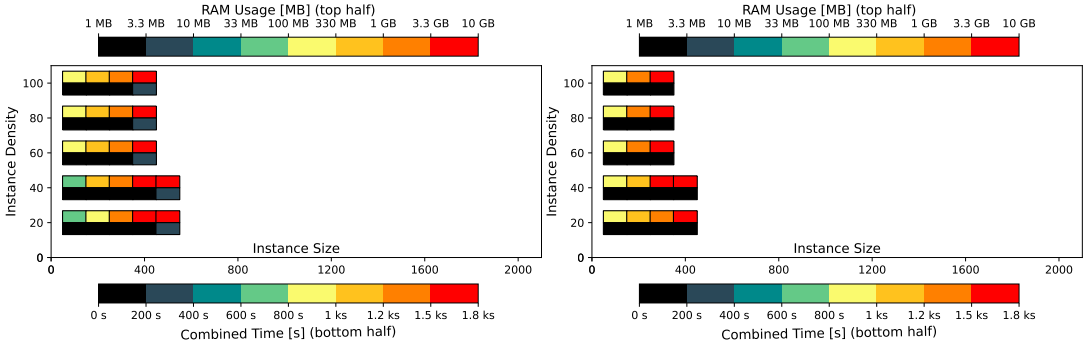
(d) Grounding profile for NG-I.

Figure 12: Grounding profiles for all ground-and-solve systems for the *34-4Cliqu.* example. Compare the traditional ground-and-solve systems (Figure 12a, and Figure 12c), to the `newground3` approaches (Figure 12b, and Figure 12d). Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), and `newground3` with `idlv` (NG-I). Timeout: 1800s; Memout: 10GB.



(a) Grounding profile for `gringo`.

(b) Grounding profile for NG-G.



(c) Grounding profile for `idlv`.

(d) Grounding profile for NG-I.

Figure 13: Grounding profiles for all ground-and-solve systems for the *35-NPRC.* example. Compare the traditional ground-and-solve systems (Figure 13a, and Figure 13c), to the `newground3` approaches, although, *the heuristic decided against the usage of BDG* (Figure 13b, and Figure 13d). Measured `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), and `newground3` with `idlv` (NG-I). Timeout: 1800s; Memout: 10GB.

Instance Name	#I	Total #Solved																				
		gringo			idlv			NG-G			NG-G-TW			NG-I			ALPHA			ProASP		
		#S	M	T	#S	M	T	#S	M	T	#S	M	T	#S	M	T	#S	M	T	#S	M	T
		Grounding-Heavy Scenarios																				
Total-SUM	1000	221	171	608	293	203	504	584	329	87	-	-	-	642	245	113	147	177	176	-	-	-
ProASP-SUM	500	149	97	254	158	102	240	280	210	10	-	-	-	288	198	14	147	177	176	389	81	30
30-Cliqu.	100	61	19	20	61	19	20	92	8	0	-	-	-	92	8	0	45	46	9	99	1	0
31-Cliqu.(!=)	100	39	18	43	39	18	43	91	9	0	-	-	-	91	9	0	44	33	23	99	1	0
32-Path.	100	38	24	38	39	23	38	89	6	5	-	-	-	92	6	2	-	-	-	-	-	-
33-Col.	100	20	16	64	20	16	64	31	59	10	-	-	-	29	60	11	38	57	5	92	5	3
34-4Cliqu.	100	29	31	40	28	32	40	66	34	0	-	-	-	66	34	0	18	0	82	99	1	0
35-NPRC.	100	0	13	87	10	17	73	0	100	0	-	-	-	10	87	3	2	41	57	0	73	27
36-S3T4	100	12	17	71	18	17	65	98	0	2	-	-	-	100	0	0	-	-	-	-	-	-
37-S4T4	100	5	8	87	18	16	66	100	0	0	-	-	-	100	0	0	-	-	-	-	-	-
38-NPRC	100	17	19	64	60	40	0	17	79	4	-	-	-	62	37	1	-	-	-	-	-	-
39-SM-Agg	100	0	6	94	0	5	95	0	34	66	-	-	-	0	4	96	-	-	-	-	-	-
		Solving-Heavy Scenarios																				
Total-SUM	8509	5449	650	2410	5469	697	2343	5418	524	2567	5409	423	2677	5434	599	2476	-	-	-	-	-	-
Alpha-SUM	1640	1255	30	355	1280	0	360	1251	24	365	1243	0	397	1272	0	368	183	290	1167	-	-	-
ProASP-SUM	320	308	0	12	308	0	12	307	0	13	307	0	13	306	0	14	3	53	264	311	0	9
01-Perm.	201	155	46	0	167	34	0	155	34	12	163	0	38	167	28	6	-	-	-	-	-	-
01-Perm.-New	201	171	30	0	201	0	0	171	24	6	195	0	6	201	0	0	180	1	20	-	-	-
02-Valv.	318	57	0	261	56	0	262	57	0	261	57	0	261	57	0	261	-	-	-	-	-	-
02-Valv.-New	318	74	0	244	77	0	241	76	0	242	76	0	242	77	0	241	-	-	-	-	-	-
04-Conn.	26	4	0	22	4	0	22	4	0	22	4	0	22	4	0	22	-	-	-	-	-	-
04-Conn.-New	26	4	0	22	4	0	22	4	0	22	4	0	22	4	0	22	-	-	-	-	-	-
05-Grac.	60	35	0	25	32	0	28	29	0	31	29	0	31	29	0	31	-	-	-	-	-	-

Instance Name	#I	Total #Solved																				
		gringo			idlv			NG-G			NG-G-TW			NG-I			ALPHA			ProASP		
		#S	M	T	#S	M	T	#S	M	T	#S	M	T	#S	M	T	#S	M	T	#S	M	T
05-Grac.-New	60	26	0	34	23	0	37	26	0	34	26	0	34	29	0	31	-	-	-	-	-	-
06-Bottl.	100	100	0	0	100	0	0	100	0	0	100	0	0	100	0	0	0	51	49	100	0	0
06-Bottl.-New	100	100	0	0	100	0	0	100	0	0	100	0	0	100	0	0	0	77	23	-	-	-
07-Nomi.	59	21	5	33	19	5	35	18	4	37	19	2	38	20	5	34	-	-	-	-	-	-
07-Nomi.-New	59	21	5	33	19	5	35	18	5	36	19	2	38	20	5	34	-	-	-	-	-	-
08-Sok.	1080	1072	8	0	1072	8	0	1072	7	1	1072	7	1	1072	8	0	-	-	-	-	-	-
08-Sok.-New	1080	401	9	670	397	8	675	404	7	669	404	8	668	397	7	676	-	-	-	-	-	-
09-Rico.	50	50	0	0	50	0	0	50	0	0	50	0	0	50	0	0	-	-	-	-	-	-
09-Rico.-New	50	50	0	0	50	0	0	50	0	0	50	0	0	50	0	0	-	-	-	-	-	-
10-Cross.	85	37	0	48	38	0	47	37	0	48	40	0	45	38	0	47	-	-	-	-	-	-
10-Cross.-New	85	49	0	36	48	0	37	49	0	36	49	0	36	49	0	36	-	-	-	-	-	-
12-Strat.	37	0	37	0	12	0	25	12	0	25	12	0	25	12	0	25	-	-	-	-	-	-
13-Sol.	27	22	0	5	22	0	5	22	0	5	22	0	5	21	0	6	-	-	-	-	-	-
13-Sol.-New	27	25	0	2	0	0	27	25	0	2	25	0	2	0	0	27	-	-	-	-	-	-
14-Weigh.	65	59	0	6	59	0	6	59	0	6	55	0	10	59	0	6	-	-	-	-	-	-
14-Weigh.-New	65	65	0	0	65	0	0	65	0	0	65	0	0	65	0	0	0	63	2	-	-	-
15-Stabl.	50	48	0	2	48	0	2	48	0	2	32	0	18	48	0	2	0	50	0	-	-	-
15-Stabl.-New	50	50	0	0	50	0	0	50	0	0	50	0	0	50	0	0	-	-	-	-	-	-
16-Incr.	500	77	218	205	74	300	126	75	225	200	74	215	211	71	238	191	-	-	-	-	-	-
16-Incr.-New	500	331	47	122	331	92	77	335	53	112	332	31	137	337	78	85	-	-	-	-	-	-
17-Qual.	159	159	0	0	159	0	0	159	0	0	159	0	0	159	0	0	-	-	-	-	-	-
17-Qual.-New	159	159	0	0	159	0	0	159	0	0	159	0	0	159	0	0	-	-	-	-	-	-
19-Abst.	200	200	0	0	200	0	0	200	0	0	200	0	0	200	0	0	-	-	-	-	-	-
19-Abst.-New	200	200	0	0	200	0	0	200	0	0	200	0	0	200	0	0	-	-	-	-	-	-
20-Visit.	100	62	0	38	61	0	39	61	0	39	61	0	39	61	0	39	-	-	-	-	-	-
20-Visit.-New	100	95	0	5	95	0	5	94	0	6	94	0	6	93	0	7	3	2	95	100	0	0

Instance Name	#I	Total #Solved																			
		`gringo`			`idlv`			NG-G			NG-G-TW			NG-I			ALPHA			ProASP		
		#S	M	T	#S	M	T	#S	M	T	#S	M	T	#S	M	T	#S	M	T	#S	M	T
21-Compl.	78	76	0	2	77	0	1	74	0	4	74	0	4	78	0	0	-	-	-	-	-	-
21-Compl.-New	78	78	0	0	77	0	1	75	0	3	75	0	3	74	0	4	-	-	-	-	-	-
22-Knight.	300	10	245	45	11	245	44	10	165	125	2	158	140	11	230	59	-	-	-	-	-	-
22-Knight.-New	300	41	0	259	42	0	258	41	0	259	41	0	259	41	0	259	0	46	254	-	-	-
23-Max.	50	6	0	44	6	0	44	6	0	44	6	0	44	6	0	44	-	-	-	-	-	-
23-Max.-New	50	5	0	45	17	0	33	5	0	45	5	0	45	17	0	33	-	-	-	-	-	-
24-Lab.	246	222	0	24	220	0	26	222	0	24	213	0	33	218	0	28	0	0	246	-	-	-
24-Lab.-New	246	222	0	24	219	0	27	222	0	24	215	0	31	219	0	27	0	0	246	-	-	-
25-Min.	250	250	0	0	250	0	0	250	0	0	250	0	0	250	0	0	-	-	-	-	-	-
25-Min.-New	250	250	0	0	250	0	0	250	0	0	250	0	0	250	0	0	-	-	-	-	-	-
26-Hanoi.	60	58	0	2	58	0	2	56	0	4	58	0	2	56	0	4	-	-	-	-	-	-
26-Hanoi.-New	60	60	0	0	60	0	0	60	0	0	60	0	0	60	0	0	0	0	60	60	0	0
27-Graph.	60	28	0	32	28	0	32	28	0	32	28	0	32	28	0	32	-	-	-	-	-	-
27-Graph.-New	60	53	0	7	53	0	7	53	0	7	53	0	7	53	0	7	0	0	60	51	0	9
28-Part.	112	33	0	79	32	0	80	7	0	105	7	0	105	0	0	112	-	-	-	-	-	-
28-Part.-New	112	78	0	34	77	0	35	75	0	37	75	0	37	74	0	38	0	0	112	-	-	-

Table 1: Detailed solving results for the automatic splitter on solving- and grounding-heavy benchmarks. Number of solved instances (Total #Solved). Compared `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `gringo` and `Lpopt` (NG-G-TW), `newground3` with `idlv` (NG-I), ALPHA (Alpha), ProASP (ProASP) and, ProASP where constraints are lazily grounded (ProASP$_{CS}$). *Alpha-SUM*, and *ProASP-SUM* are the solving heavy benchmarks restricted to the fragment where the program syntax is supported by ALPHA, and ProASP respectively. Measured "#S" are the number of solved instances, "M" are the number of *MEMOUTS*, and "T" are the number of *TIMEOUTS*. Timeout: 1800s; Memout: 10GB.

Instance Name	#Insts.	Total #Grounded														
		gringo			idlv			NG-G			NG-G-TW			NG-I		
		#G	M	T	#G	M	T	#G	M	T	#G	M	T	#G	M	T
Grounding-Heavy Scenarios																
Total-SUM	1000	315	189	496	414	170	416	783	204	13	-	-	-	904	91	5
30-Cliqu.	100	77	19	4	78	18	4	100	0	0	-	-	-	100	0	0
31-Cliqu.(!=)	100	53	27	20	54	23	23	100	0	0	-	-	-	100	0	0
32-Path.	100	58	25	17	58	22	20	100	0	0	-	-	-	100	0	0
33-Col.	100	31	25	44	32	23	45	94	6	0	-	-	-	96	4	0
34-4Cliqu.	100	37	29	34	34	28	38	100	0	0	-	-	-	100	0	0
35-NPRC.	100	5	6	89	22	17	61	4	96	0	-	-	-	17	79	4
36-S3T4	100	15	22	63	18	18	64	100	0	0	-	-	-	100	0	0
37-S4T4	100	8	5	87	18	16	66	100	0	0	-	-	-	100	0	0
38-NPRC	100	31	25	44	100	0	0	26	68	6	-	-	-	96	4	0
39-SM-Agg	100	0	6	94	0	5	95	59	34	7	-	-	-	95	4	1
Solving-Heavy Scenarios																
Total-SUM	8509	8037	395	77	8107	375	27	7925	584	0	8019	458	32	7842	528	139
01-Perm.	201	181	20	0	189	12	0	169	32	0	201	0	0	181	20	0
01-Perm.-New	201	191	10	0	201	0	0	181	20	0	201	0	0	201	0	0
02-Valv.	318	318	0	0	318	0	0	318	0	0	318	0	0	318	0	0
02-Valv.-New	318	318	0	0	318	0	0	318	0	0	318	0	0	318	0	0
04-Conn.	26	26	0	0	26	0	0	26	0	0	26	0	0	26	0	0
04-Conn.-New	26	26	0	0	26	0	0	26	0	0	26	0	0	26	0	0
05-Grac.	60	60	0	0	60	0	0	60	0	0	60	0	0	60	0	0
05-Grac.-New	60	60	0	0	60	0	0	60	0	0	60	0	0	60	0	0
06-Bottl.	100	100	0	0	100	0	0	100	0	0	100	0	0	100	0	0
06-Bottl.-New	100	100	0	0	100	0	0	100	0	0	100	0	0	100	0	0

Instance Name	#Insts.	Total #Grounded														
		gringo			idlv			NG-G			NG-G-TW			NG-I		
		#G	M	T	#G	M	T	#G	M	T	#G	M	T	#G	M	T
07-Nomi.	59	58	1	0	59	0	0	56	3	0	59	0	0	58	1	0
07-Nomi.-New	59	58	1	0	59	0	0	56	3	0	59	0	0	58	1	0
08-Sok.	1080	1076	4	0	1075	5	0	1073	7	0	1073	7	0	1073	7	0
08-Sok.-New	1080	1076	4	0	1075	5	0	1073	7	0	1073	7	0	1073	7	0
09-Rico.	50	50	0	0	50	0	0	50	0	0	50	0	0	50	0	0
09-Rico.-New	50	50	0	0	50	0	0	50	0	0	50	0	0	50	0	0
10-Cross.	85	85	0	0	85	0	0	85	0	0	85	0	0	85	0	0
10-Cross.-New	85	85	0	0	85	0	0	85	0	0	85	0	0	85	0	0
12-Strat.	37	0	37	0	37	0	0	37	0	0	37	0	0	37	0	0
13-Sol.	27	27	0	0	27	0	0	27	0	0	27	0	0	27	0	0
13-Sol.-New	27	27	0	0	0	0	27	27	0	0	27	0	0	0	0	27
14-Weigh.	65	65	0	0	65	0	0	65	0	0	65	0	0	65	0	0
14-Weigh.-New	65	65	0	0	65	0	0	65	0	0	65	0	0	65	0	0
15-Stabl.	50	50	0	0	50	0	0	50	0	0	50	0	0	50	0	0
15-Stabl.-New	50	50	0	0	50	0	0	50	0	0	50	0	0	50	0	0
16-Incr.	500	293	130	77	320	180	0	206	294	0	250	250	0	250	250	0
16-Incr.-New	500	465	35	0	467	33	0	465	35	0	463	5	32	438	62	0
17-Qual.	159	159	0	0	159	0	0	159	0	0	159	0	0	159	0	0
17-Qual.-New	159	159	0	0	159	0	0	159	0	0	159	0	0	159	0	0
19-Abst.	200	200	0	0	200	0	0	200	0	0	200	0	0	200	0	0
19-Abst.-New	200	200	0	0	200	0	0	200	0	0	200	0	0	200	0	0
20-Visit.	100	100	0	0	100	0	0	100	0	0	100	0	0	100	0	0
20-Visit.-New	100	100	0	0	100	0	0	100	0	0	100	0	0	100	0	0
21-Compl.	78	78	0	0	78	0	0	78	0	0	78	0	0	78	0	0
21-Compl.-New	78	78	0	0	78	0	0	78	0	0	78	0	0	78	0	0
22-Knight.	300	147	153	0	160	140	0	117	183	0	111	189	0	120	180	0

Instance Name	#Insts.	Total #Grounded														
		gringo			idlv			NG-G			NG-G-TW			NG-I		
		#G	M	T	#G	M	T	#G	M	T	#G	M	T	#G	M	T
22-Knight.-New	300	300	0	0	300	0	0	300	0	0	300	0	0	300	0	0
23-Max.	50	50	0	0	50	0	0	50	0	0	50	0	0	50	0	0
23-Max.-New	50	50	0	0	50	0	0	50	0	0	50	0	0	50	0	0
24-Lab.	246	246	0	0	246	0	0	246	0	0	246	0	0	246	0	0
24-Lab.-New	246	246	0	0	246	0	0	246	0	0	246	0	0	246	0	0
25-Min.	250	250	0	0	250	0	0	250	0	0	250	0	0	250	0	0
25-Min.-New	250	250	0	0	250	0	0	250	0	0	250	0	0	250	0	0
26-Hanoi.	60	60	0	0	60	0	0	60	0	0	60	0	0	60	0	0
26-Hanoi.-New	60	60	0	0	60	0	0	60	0	0	60	0	0	60	0	0
27-Graph.	60	60	0	0	60	0	0	60	0	0	60	0	0	60	0	0
27-Graph.-New	60	60	0	0	60	0	0	60	0	0	60	0	0	60	0	0
28-Part.	112	112	0	0	112	0	0	112	0	0	112	0	0	0	0	112
28-Part.-New	112	112	0	0	112	0	0	112	0	0	112	0	0	112	0	0

Table 2: Detailed grounding results for the automatic splitter on solving- and grounding-heavy benchmarks. Number of Grounded instances (Total #Grounded). Compared `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `gringo` and `Lpopt` (NG-G-TW), `newground3` with `idlv` (NG-I), Measured "#G" are the number of grounded instances, "M" are the number of *MEMOUTS*, and "T" are the number of *TIMEOUTS*. Timeout: 1800s; Memout: 10GB.

Instance Name	#I	Total Time (T[h]), Total Mem-Usage (M[TB])													
		gringo		idlv		NG-G		NG-G-TW		NG-I		ALPHA		ProASP	
		T	M	T	M[	T	M	T	M	T	M	T	M	T	M
		Grounding-Heavy Scenarios													
Total-SUM	1000	387.98	8.12	353.87	7.60	246.76	5.23	-	-	212.38	4.57	-	-	-	-
ProASP-SUM	500	175.32	3.78	171.73	3.70	132.41	3.19	-	-	126.41	3.11	185.26	4.05	65.58	2.28
30-Cliqu.	100	19.33	0.51	19.34	0.51	11.45	0.44	-	-	10.70	0.43	30.44	0.73	2.99	0.30
31-Cliqu.(!=)	100	30.06	0.70	30.04	0.70	12.28	0.45	-	-	11.56	0.45	30.77	0.73	2.97	0.30
32-Path.	100	29.81	0.70	29.89	0.70	13.03	0.45	-	-	12.20	0.44	-	-	-	-
33-Col.	100	39.35	0.84	39.45	0.84	38.96	0.75	-	-	39.49	0.75	31.59	0.76	6.55	0.39
34-4Cliqu.	100	36.57	0.76	37.70	0.75	19.72	0.58	-	-	19.45	0.58	43.25	0.86	3.07	0.30
35-NPRC.	100	50.00	0.98	45.20	0.91	50.00	0.98	-	-	45.21	0.91	49.21	0.97	50.00	0.98
36-S3T4	100	43.79	0.88	41.10	0.84	7.83	0.12	-	-	2.45	0.10	-	-	-	-
37-S4T4	100	47.20	0.94	41.09	0.84	1.61	0.13	-	-	1.71	0.14	-	-	-	-
38-NPRC	100	41.87	0.85	20.06	0.55	41.88	0.85	-	-	19.61	0.54	-	-	-	-
39-SM-Agg	100	50.00	0.98	50.00	0.98	50.00	0.50	-	-	50.00	0.25	-	-	-	-
		Solving-Heavy Scenarios													
Total-SUM	8509	1665.48	11.50	1650.63	11.71	1675.12	11.76	1706.34	10.43	1669.58	11.39	-	-	-	-
Alpha-SUM	1640	233.60	0.84	219.54	0.27	235.91	0.86	257.88	0.37	220.92	0.30	734.69	8.65	-	-
ProASP-SUM	320	17.17	0.04	17.09	0.05	17.13	0.05	17.21	0.06	17.32	0.06	158.50	1.63	20.34	0.68
01-Perm.	201	21.84	0.66	18.61	0.52	21.91	0.66	27.33	0.12	18.66	0.52	-	-	-	-
01-Perm.-New	201	16.51	0.51	0.51	0.04	16.61	0.51	11.20	0.09	0.54	0.04	16.68	0.53	-	-
02-Valv.	318	133.48	0.10	133.30	0.11	133.46	0.10	133.47	0.10	133.09	0.11	-	-	-	-
02-Valv.-New	318	125.57	0.39	125.86	0.39	125.20	0.37	125.27	0.37	125.34	0.37	-	-	-	-
04-Conn.	26	11.04	0.00	11.05	0.00	11.04	0.00	11.04	0.00	11.05	0.00	-	-	-	-
04-Conn.-New	26	11.04	0.00	11.03	0.00	11.04	0.00	11.04	0.00	11.04	0.00	-	-	-	-
05-Grac.	60	16.92	0.00	17.00	0.00	17.90	0.00	17.90	0.00	18.24	0.00	-	-	-	-

Instance Name	#I	Total Time (T[h]), Total Mem-Usage (M[TB])													
		gringo		idlv		NG-G		NG-G-TW		NG-I		ALPHA		ProASP	
		T	M	T	M[	T	M	T	M	T	M	T	M	T	M
05-Grac.-New	60	19.86	0.00	19.98	0.00	19.84	0.00	19.86	0.00	18.17	0.00	-	-	-	-
06-Bottl.	100	0.34	0.03	0.36	0.04	0.38	0.03	0.43	0.04	0.40	0.05	50.00	0.91	2.25	0.21
06-Bottl.-New	100	0.27	0.02	0.20	0.02	0.31	0.02	0.31	0.02	0.24	0.03	50.00	0.95	-	-
07-Nomi.	59	21.83	0.15	21.26	0.14	21.74	0.15	21.31	0.10	21.40	0.14	-	-	-	-
07-Nomi.-New	59	21.75	0.15	21.25	0.14	21.72	0.15	21.27	0.10	21.36	0.14	-	-	-	-
08-Sok.	1080	8.38	0.50	9.00	0.49	8.49	0.53	8.62	0.53	9.72	0.51	-	-	-	-
08-Sok.-New	1080	360.35	0.62	360.95	0.61	357.11	0.63	357.16	0.63	362.40	0.61	-	-	-	-
09-Rico.	50	2.20	0.00	1.43	0.00	2.22	0.00	1.41	0.00	1.42	0.00	-	-	-	-
09-Rico.-New	50	2.17	0.00	1.76	0.00	2.04	0.00	1.60	0.00	1.78	0.00	-	-	-	-
10-Cross.	85	26.32	0.00	26.26	0.00	26.33	0.01	25.78	0.01	26.27	0.01	-	-	-	-
10-Cross.-New	85	21.98	0.00	21.37	0.00	21.98	0.01	22.00	0.01	21.37	0.01	-	-	-	-
12-Strat.	37	18.50	0.00	12.81	0.07	12.99	0.10	12.99	0.10	12.95	0.10	-	-	-	-
13-Sol.	27	2.53	0.00	2.59	0.00	2.98	0.00	2.99	0.00	3.02	0.00	-	-	-	-
13-Sol.-New	27	1.16	0.00	13.50	0.00	1.17	0.00	1.17	0.00	13.50	0.00	-	-	-	-
14-Weigh.	65	5.90	0.02	6.16	0.02	5.92	0.02	9.33	0.02	6.18	0.02	-	-	-	-
14-Weigh.-New	65	0.39	0.00	0.31	0.00	0.31	0.00	0.44	0.00	0.32	0.00	32.50	0.63	-	-
15-Stabl.	50	2.35	0.11	1.27	0.01	2.38	0.11	15.32	0.03	1.28	0.01	25.00	0.49	-	-
15-Stabl.-New	50	0.09	0.01	0.09	0.01	0.10	0.01	0.10	0.01	0.10	0.01	-	-	-	-
16-Incr.	500	216.63	3.43	215.52	4.02	217.17	3.96	213.91	3.94	216.29	3.99	-	-	-	-
16-Incr.-New	500	88.66	1.04	89.72	1.37	88.29	1.05	90.96	0.88	88.17	1.37	-	-	-	-
17-Qual.	159	3.04	0.08	3.35	0.08	3.11	0.09	3.11	0.09	3.04	0.08	-	-	-	-
17-Qual.-New	159	3.34	0.09	3.45	0.09	3.33	0.09	3.33	0.09	3.38	0.09	-	-	-	-
19-Abst.	200	0.81	0.01	0.65	0.01	0.85	0.02	0.85	0.02	0.68	0.01	-	-	-	-
19-Abst.-New	200	0.70	0.01	0.51	0.01	0.74	0.02	0.73	0.02	0.55	0.01	-	-	-	-
20-Visit.	100	22.17	0.01	22.60	0.01	22.18	0.01	22.18	0.01	22.60	0.02	-	-	-	-
20-Visit.-New	100	9.01	0.01	9.27	0.01	8.87	0.01	8.88	0.01	9.44	0.01	48.50	0.50	5.81	0.21

Instance Name	#I	Total Time (T[h]), Total Mem-Usage (M[TB])													
		gringo		idlv		NG-G		NG-G-TW		NG-I		ALPHA		ProASP	
		T	M	T	M[	T	M	T	M	T	M	T	M	T	M
21-Compl.	78	3.88	0.02	4.87	0.03	4.69	0.02	4.70	0.02	3.65	0.03	-	-	-	-
21-Compl.-New	78	3.46	0.02	3.43	0.03	3.95	0.02	3.98	0.02	4.69	0.03	-	-	-	-
22-Knight.	300	145.30	2.70	144.49	2.69	145.31	2.68	149.09	2.68	144.79	2.69	-	-	-	-
22-Knight.-New	300	130.32	0.06	129.88	0.06	130.29	0.06	130.26	0.06	130.68	0.06	150.00	2.39	-	-
23-Max.	50	23.65	0.01	23.65	0.01	23.67	0.01	23.67	0.01	23.66	0.01	-	-	-	-
23-Max.-New	50	23.74	0.01	19.99	0.02	23.74	0.01	23.72	0.01	20.03	0.01	-	-	-	-
24-Lab.	246	23.64	0.04	25.33	0.04	24.60	0.04	31.76	0.05	25.43	0.04	123.00	0.63	-	-
24-Lab.-New	246	23.64	0.04	25.28	0.04	24.57	0.04	31.65	0.05	25.30	0.04	123.00	0.62	-	-
25-Min.	250	0.71	0.08	0.54	0.06	0.65	0.07	0.73	0.07	0.82	0.08	-	-	-	-
25-Min.-New	250	0.46	0.05	0.52	0.05	0.51	0.05	0.54	0.05	0.57	0.06	-	-	-	-
26-Hanoi.	60	3.74	0.00	3.67	0.00	4.36	0.00	3.73	0.00	4.21	0.00	-	-	-	-
26-Hanoi.-New	60	0.28	0.00	0.27	0.00	0.35	0.00	0.35	0.00	0.29	0.00	30.00	0.14	1.64	0.13
27-Graph.	60	18.31	0.00	18.49	0.00	18.30	0.00	18.30	0.00	18.49	0.00	-	-	-	-
27-Graph.-New	60	7.54	0.00	7.20	0.00	7.53	0.00	7.54	0.00	7.19	0.00	30.00	0.08	10.65	0.13
28-Part.	112	40.38	0.46	40.37	0.45	53.16	0.03	53.27	0.03	56.00	0.02	-	-	-	-
28-Part.-New	112	19.32	0.01	19.65	0.01	19.70	0.01	19.72	0.01	19.81	0.01	56.00	0.78	-	-

Table 3: Detailed solving results for the automatic splitter on solving- and grounding-heavy benchmarks. Compared `idlv` (idlv), `gringo` (gringo), `newground3` with `gringo` (NG-G), `newground3` with `gringo` and `Lpopt` (NG-G-TW), `newground3` with `idlv` (NG-I), ALPHA (Alpha), ProASP (ProASP) and, ProASP where constraints are lazily grounded (ProASP$_{CS}$). *ALPHA-SUM* and *ProASP-SUM* are the solving heavy benchmarks restricted to the fragment where the program syntax is supported by ALPHA, and ProASP respectively. Measured grounding time in hours (T[h]), and grounding size in terabytes (M[TB]). Timeout: 1800s; Memout: 10GB.

Instance Name	#I	Grounding Time (T[h]), Grounding Size (M[TB])									
		gringo		idlv		NG-G		NG-G-TW		NG-I	
		T	M	T	M	T	M	T	M	T	M
		Grounding-Heavy Scenarios									
Total-SUM	1000	338.63	7.29	291.70	6.43	118.87	2.99	-	-	57.96	2.07
30-Cliqu.	100	11.11	0.35	11.25	0.35	1.68	0.08	-	-	1.01	0.07
31-Cliqu.(!=)	100	22.76	0.56	22.86	0.56	1.79	0.08	-	-	1.06	0.08
32-Path.	100	18.97	0.54	19.02	0.53	1.90	0.08	-	-	1.14	0.08
33-Col.	100	32.31	0.74	32.54	0.74	4.50	0.35	-	-	3.00	0.32
34-4Cliqu.	100	34.14	0.67	35.83	0.68	2.07	0.13	-	-	1.33	0.12
35-NPRC.	100	47.74	0.94	37.51	0.82	47.76	0.96	-	-	40.90	0.85
36-S3T4	100	41.72	0.84	40.50	0.83	0.27	0.01	-	-	0.10	0.01
37-S4T4	100	46.23	0.92	40.42	0.83	0.41	0.02	-	-	0.28	0.03
38-NPRC	100	33.65	0.74	1.77	0.13	36.63	0.78	-	-	1.76	0.27
39-SM-Agg	100	50.00	0.98	50.00	0.98	21.86	0.50	-	-	7.38	0.25
		Solving-Heavy Scenarios									
Total-SUM	8509	282.08	5.67	222.92	5.08	311.91	6.05	288.30	5.46	348.89	6.95
01-Perm.	201	10.56	0.26	5.57	0.16	16.88	0.35	0.36	0.01	9.92	0.22
01-Perm.-New	201	5.14	0.15	0.15	0.00	8.07	0.19	0.22	0.01	0.18	0.00
02-Valv.	318	0.67	0.01	0.45	0.01	0.73	0.01	0.75	0.01	0.52	0.01
02-Valv.-New	318	1.24	0.04	1.23	0.03	1.32	0.04	1.33	0.04	1.31	0.03
04-Conn.	26	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00
04-Conn.-New	26	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00
05-Grac.	60	0.01	0.00	0.01	0.00	0.01	0.00	0.02	0.00	0.02	0.00
05-Grac.-New	60	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.01	0.00
06-Bottl.	100	0.20	0.00	0.19	0.00	0.24	0.00	0.27	0.00	0.23	0.00
06-Bottl.-New	100	0.19	0.00	0.11	0.00	0.22	0.00	0.22	0.00	0.15	0.00

Instance Name	#I	Grounding Time (T[h]), Grounding Size (M[TB])									
		gringo		idlv		NG-G		NG-G-TW		NG-I	
		T	M	T	M	T	M	T	M	T	M
07-Nomi.	59	0.78	0.03	0.66	0.01	1.14	0.03	0.27	0.01	0.67	0.01
07-Nomi.-New	59	0.80	0.03	0.68	0.01	1.15	0.03	0.28	0.01	0.69	0.01
08-Sok.	1080	4.42	0.11	4.80	0.11	5.37	0.12	5.49	0.12	5.43	0.11
08-Sok.-New	1080	4.52	0.11	4.94	0.11	5.47	0.12	5.60	0.12	5.58	0.12
09-Rico.	50	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.01	0.00
09-Rico.-New	50	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.01	0.00
10-Cross.	85	0.00	0.00	0.00	0.00	0.01	0.00	0.02	0.00	0.01	0.00
10-Cross.-New	85	0.00	0.00	0.00	0.00	0.01	0.00	0.02	0.00	0.01	0.00
12-Strat.	37	18.50	0.36	0.22	0.01	0.41	0.01	0.41	0.01	0.30	0.01
13-Sol.	27	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.01	0.00
13-Sol.-New	27	0.00	0.00	13.50	0.26	0.00	0.00	0.00	0.00	13.50	0.26
14-Weigh.	65	0.03	0.00	0.07	0.00	0.04	0.00	0.08	0.00	0.08	0.00
14-Weigh.-New	65	0.00	0.00	0.00	0.00	0.01	0.00	0.02	0.00	0.01	0.00
15-Stabl.	50	0.67	0.02	0.05	0.00	0.71	0.02	0.08	0.00	0.06	0.00
15-Stabl.-New	50	0.04	0.00	0.04	0.00	0.05	0.00	0.05	0.00	0.05	0.00
16-Incr.	500	138.10	2.41	99.27	2.20	148.67	2.79	136.68	2.74	137.13	2.73
16-Incr.-New	500	22.12	0.36	18.37	0.45	24.40	0.42	38.15	0.44	27.56	0.54
17-Qual.	159	0.22	0.01	0.19	0.01	0.29	0.01	0.29	0.01	0.26	0.01
17-Qual.-New	159	0.22	0.01	0.21	0.01	0.29	0.01	0.29	0.01	0.28	0.01
19-Abst.	200	0.20	0.00	0.01	0.00	0.24	0.00	0.23	0.00	0.05	0.00
19-Abst.-New	200	0.20	0.00	0.01	0.00	0.24	0.00	0.22	0.00	0.05	0.00
20-Visit.	100	0.00	0.00	0.00	0.00	0.02	0.00	0.02	0.00	0.02	0.00
20-Visit.-New	100	0.02	0.00	0.01	0.00	0.04	0.00	0.04	0.00	0.02	0.00
21-Compl.	78	0.16	0.00	1.06	0.00	0.18	0.00	0.19	0.00	1.05	0.00
21-Compl.-New	78	0.16	0.00	0.79	0.00	0.18	0.00	0.19	0.00	0.81	0.00
22-Knight.	300	67.18	1.59	65.62	1.52	92.96	1.87	93.83	1.89	85.36	1.74

Instance Name	#I	Grounding Time (T[h]), Grounding Size (M[TB])									
		`gringo`		`idlv`		NG-G		NG-G-TW		NG-I	
		T	M	T	M	T	M	T	M	T	M
22-Knight.-New	300	1.35	0.01	0.19	0.00	1.40	0.01	1.40	0.01	0.24	0.00
23-Max.	50	0.02	0.00	0.01	0.00	0.02	0.00	0.02	0.00	0.02	0.00
23-Max.-New	50	0.01	0.00	0.00	0.00	0.02	0.00	0.02	0.00	0.01	0.00
24-Lab.	246	0.05	0.00	0.16	0.00	0.10	0.00	0.14	0.00	0.20	0.00
24-Lab.-New	246	0.05	0.00	0.15	0.00	0.10	0.00	0.14	0.00	0.20	0.00
25-Min.	250	0.32	0.01	0.27	0.00	0.32	0.01	0.38	0.01	0.41	0.01
25-Min.-New	250	0.23	0.00	0.29	0.00	0.29	0.01	0.31	0.01	0.35	0.00
26-Hanoi.	60	0.01	0.00	0.00	0.00	0.01	0.00	0.02	0.00	0.01	0.00
26-Hanoi.-New	60	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.01	0.00
27-Graph.	60	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.01	0.00
27-Graph.-New	60	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00	0.01	0.00
28-Part.	112	3.67	0.15	3.59	0.15	0.14	0.00	0.12	0.00	56.00	1.09
28-Part.-New	112	0.02	0.00	0.02	0.00	0.03	0.00	0.04	0.00	0.03	0.00

Table 4: Detailed grounding results for the automatic splitter on solving- and grounding-heavy benchmarks. Compared `idlv` (`idlv`), `gringo` (`gringo`), `newground3` with `gringo` (NG-G), `newground3` with `gringo` and `Lpopt` (NG-G-TW), and `newground3` with `idlv` (NG-I). Measured grounding time in hours (T[h]), and grounding size in terabytes (M[TB]). Timeout: 1800s; Memout: 10GB.

Scen.	#Insts.	Total Time (T[h]), Total Mem-Usage (M[TB])							
		gringo		idlv		$\mathcal{HG}$		$\mathcal{HG}_{\mathcal{F}}$	
		T	M	T	M	T	M	T	M
Total-SUM	120	56.08	1.11	56.07	1.11	48.89	1.01	33.34	0.72
Four-Clique	40	17.07	0.34	17.06	0.34	10.82	0.26	0.55	0.04
Hyper-Six-Clique	40	19.50	0.38	19.50	0.38	19.03	0.37	16.15	0.34
Hyper-Seven-Clique	40	19.51	0.38	19.51	0.38	19.04	0.38	16.64	0.34

Table 5: Comparing total (grounding + solving) time and total memory usage (while grounding and solving) for FastFound ($\mathcal{HG}_{\mathcal{F}}$), GuessFound ($\mathcal{HG}$), `gringo` and `idlv`. This table displays the detailed (solving) results for the experiments of Section 6.4. T total time in hours [h], M total memory usage in terabytes [TB]. Timeout of 1800s, Memout of 10GB.

Scen.	#Insts.	Grounding Time (T[h]), Grounding Size (M[TB])							
		gringo		idlv		$\mathcal{HG}$		$\mathcal{HG}_{\mathcal{F}}$	
		T	M	T	M	T	M	T	M
Total-SUM	120	53.26	1.05	53.13	1.05	42.88	0.85	27.35	0.54
Four-Clique	40	14.70	0.30	14.56	0.30	5.28	0.11	0.10	0.00
Hyper-Six-Clique	40	19.06	0.37	19.06	0.37	18.59	0.36	13.38	0.26
Hyper-Seven-Clique	40	19.50	0.38	19.50	0.38	19.02	0.37	13.87	0.27

Table 6: Comparing grounding time and size for FastFound ($\mathcal{HG}_{\mathcal{F}}$), GuessFound ($\mathcal{HG}$), `gringo` and `idlv`. This table displays the detailed (grounding) results for the experiments of Section 6.4. T total time in hours [h], M total memory usage in terabytes [TB]. Timeout of 1800s, Memout of 10GB.

Scen.	#Insts.	Total Time (T[h]), Total Mem-Usage (M[TB])							
		gringo		idlv		LVL.-MAP.		Lazy-BDG	
		T	M	T	M	T	M	T	M
Total-SUM	120	15.53	0.42	15.18	0.42	42.00	0.81	17.37	0.12
C-Hyper-Six-Clique	20	1.67	0.05	1.67	0.05	5.00	0.09	3.90	0.04
C-Hyper-Seven-Clique	20	4.12	0.09	4.11	0.09	5.86	0.12	4.19	0.05
C-Four-Clique	80	9.75	0.27	9.41	0.27	31.13	0.60	9.28	0.03

Table 7: Comparing total (grounding + solving) time and total memory usage (while grounding and solving) for Hybrid Grounding with level-mappings (LVL.-MAP.), Lazy-BDG, `gringo` and `idlv`. This table displays the detailed (solving) results for the experiments of Section 7.5. T total time in hours [h], M total memory usage in terabytes [TB]. Timeout of 1800s, Memout of 10GB.

(a) Ivory Tower (b) Fortune teller-ball-observatory (c) Tower of Wisdom

Figure 14: Illustrations of the stable-matching example problem. Figure 14a depicts the ivory tower of the logicians, Figure 14b shows the fortune-teller-ball observatory of the differentialists, and Figure 14c depicts the future tower of wisdom, which shall house both logicians and differentialists. All were created with ChatGPT-4o's web-interface on 23.12.2024.

Fictitious Stable Matching Problem (Details)

We pose the stable matching problem in the following *fictitious* setting:
A large research institute high up in the Alps wants to gain a head-start in the next AI hype-cycle by developing a broad AI [78] agent. They know that this is only possible by combining the reasoning and abstraction capabilities of the symbolic approaches developed by the logicians, with the pattern matching and learning power of the machine learning techniques, developed by the machine learning scientists (henceforth called *differentialists*). Management of the institute comes up with a plan, which brings the two research communities together. The institute is quite large and comprises of many faculty buildings, such as an *ivory tower* for the logicians, and a *fortune teller observatory* for differentialists. However, they lack space for bringing together logicians with differentialists. Therefore, they plan to build the *tower of wisdom*, which shall house both.

Unfortunately for management, logicians and differentialists are picky people regarding with whom they want to work with. Therefore, management proposes that each logician should create a list, which ranks the differentialists they want to work with, and vice versa for the differentialists. Based on this list, the management creates a *matching*. This matching should adhere to the *strong stability condition*, and the resulting problem should be a stable matching problem, as introduced in Chapter 1. We assume that there are as many logicians as there are differentialists.

The illustrations in Figure 14 depict the buildings for different disciplines, from the introductory example on the stable matching problem: the *ivory tower of the logicians*, the *fortune-teller-ball observatory* of the differentialists