Grounding Planning Tasks Using Tree Decompositions and Iterated Solving

Augusto B. Corrêa,[1] **Markus Hecher,**[2,3] **Malte Helmert,**[1] **Davide Mario Longo,**[2]
Florian Pommerening,[1] **Stefan Woltran**[2]
[1]University of Basel, Switzerland
[2]TU Wien, Institute of Logic and Computation, Austria
[3]Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, USA
{augusto.blaascorrea,malte.helmert,florian.pommerening}@unibas.ch
hecher@mit.edu, davidem.longo@gmail.com, woltran@dbai.tuwien.ac.at

Abstract

Classical planning tasks are commonly described in a first-order language. However, most classical planners translate tasks by grounding them and then rewriting them into a propositional language. In recent years, the grounding step has become a larger bottleneck. In this work, we study how to improve it. We build on top of the most common grounder for planning tasks which uses Datalog to find all reachable atoms and actions. Inspired by recent progress in lifted planning, database theory, and algorithmics, we develop a new method to ground these Datalog programs. Our algorithm can ground more instances than the baseline, and most tasks it cannot ground are out of reach from any ground planner.

Introduction

Classical planning tasks are usually defined using a first-order representation. But most of the state-of-the-art classical planners (e.g., Bonet and Geffner 2001; Hoffmann and Nebel 2001; Helmert 2006; Torralba et al. 2014; Katz and Hoffmann 2014; Francès et al. 2018) do not use it internally: as a first step, they *translate* the first-order representation into a *propositional* one (e.g., Köhler and Hoffmann 2000; Helmert 2009). The new representation facilitates different parts of the planning algorithms, such as generating successor states, representing states, and computing heuristics. Although the translation can increase the size of the task exponentially, it is still worth doing for most domains.

One way to perform this translation – and perhaps the most used one – is the method by Helmert (2009). It uses four steps: normalization, invariant synthesis, grounding, and task generation. While each of these steps is a potential bottleneck, Helmert reports that in a typical domain about 70% of the translation time is required for grounding. In this phase, a relaxed version of the task is encoded as a Datalog program (Ullman 1988, 1989). Grounding the Datalog program overapproximates the actions and atoms that are reachable from the initial state of the task. This works fine in most cases because the Datalog programs are very simple. However, grounding Datalog programs is intractable in general as the number of reachable atoms and actions might be exponential in the size of the program (Vardi 1982; Immerman 1986; Dantsin et al. 2001). As planners improve, the tasks that users want to solve also become larger and harder (e.g., Haslum 2011; Matloob and Soutchanski 2016). For such tasks, grounding is not trivial anymore and sometimes takes more time than solving the task.

In this paper, we compare Helmert's algorithm for Datalog grounding to off-the-shelf grounders (Gebser et al. 2011). To the best of our knowledge, this is the first empirical comparison of this kind in planning. We show that the Answer Set Programming (ASP) grounder `gringo` can ground more Datalog programs than Helmert's algorithm. The crucial problem in the latter is the high memory usage caused by its *program rewriting* technique. It rewrites the Datalog program such that reachable atoms can be computed more efficiently by optimized data-structures. Our empirical results show that this is damaging in many domains.

We propose a new grounder based on *treewidth* (Arnborg, Corneil, and Proskurowski 1987) and *grounding via solving* (Besin, Hecher, and Woltran 2022). Instead of grounding the entire Datalog program in one shot, we first only ground the reachable atoms of the task. This can be efficiently done in all tested instances by rewriting the program based on *tree decompositions* (Morak and Woltran 2012; Bichler, Morak, and Woltran 2016), instead of the method by Helmert. Second, we obtain the set of ground actions from the result of the first step. To do so, we set up an ASP where each stable model corresponds to one ground action. ASP solvers (e.g., Gebser et al. 2019) can enumerate the stable models iteratively without keeping previous iterations in memory, so they are well-suited for the job. Our technique can work with Datalog programs that do not come from planning problems, and could be useful more generally for Datalog grounders.

Our two-phase grounder can ground almost 20% more tasks than the method by Helmert. However, it also adds a considerable overhead. While it seems well suited for tasks that need a lot of memory, it can slow down the grounding phase. A potential concern is that the larger tasks we can now ground are already out of reach for current planners. We show that several of these tasks can still be solved with classical planners. Moreover, we show that it is hopeless to try to ground most of the remaining tasks that still could not be grounded using Datalog-based grounders, as they have more than 10^{30} actions. Thus further improvements in our grounders would not necessarily result in a higher coverage.

Background

Throughout the paper, we assume that planning languages and logic programs are defined using a *function-free logical vocabulary* over an infinite set of *variables* $\mathcal{V}$, a finite set of *constants* $\mathcal{C}$, and a set of *predicate symbols* $\mathcal{P}$. An *atom* $P(\boldsymbol{t})$ is composed of a predicate symbol $P \in \mathcal{P}$ and a k-tuple of terms $\boldsymbol{t}$ (variables or constants), where k is the arity of P. We often omit tuples of terms and refer to the predicate symbol P as an atom. The set of variables in $\boldsymbol{t}$ is denoted as $\mathit{vars}(\boldsymbol{t})$.[1] $P(\boldsymbol{t})$ is a *ground atom* if $\mathit{vars}(\boldsymbol{t}) = \emptyset$.

Logic Programming We consider a fragment of *Answer Set Programming* (ASP) in our paper. For most of the paper, it is enough to consider *Datalog programs* but some parts require more sophisticated programs. A *logic program*, in general, is a pair $\mathcal{L} = \langle \mathcal{F}, \mathcal{R} \rangle$ where $\mathcal{F}$ is a set of ground atoms, called *facts*, and $\mathcal{R}$ is a set of *disjunctive rules*.

A *disjunctive rule* r is a rule of the form

$$H_1 \vee \ldots \vee H_k \leftarrow P_1, \ldots, P_n, \neg N_1, \ldots, \neg N_m.$$

where H_i, P_i and N_i are atoms. The left-hand side of r is called the *head* and is denoted as $\mathit{head}(r)$; the right-hand side of r is called the *body* and is denoted as $\mathit{body}(r)$. We also write $\mathit{body}^+(r)$ and $\mathit{body}^-(r)$ for the subset of positive and negative atoms in $\mathit{body}(r)$. All rules we considered are *safe*, i.e., variables occurring in the head also occur in the body. The set of variables appearing in r is denoted $\mathit{vars}(r)$.

A logic program is *ground* if all atoms in all rules are ground. A non-ground rule can be interpreted as a concise representation of all possible instantiations of the variables in this rule with constants in $\mathcal{C}$. We write $\mathit{Ground}(\mathcal{L})$ to denote the grounding of a logic program $\mathcal{L}$. Note that $\mathit{Ground}(\mathcal{L})$ might be exponentially larger than $\mathcal{L}$. A set of ground atoms $\mathcal{M}$ is a *stable model* of $\mathcal{L}$ if $\mathcal{M}$ satisfies each fact and each rule in $\mathit{Ground}(\mathcal{L})$. A fact $f \in \mathcal{F}$ is satisfied by $\mathcal{M}$ if $f \in \mathcal{M}$. A ground rule r is satisfied by $\mathcal{M}$ if $\mathit{head}(r) \cap \mathcal{M} \neq \emptyset$ is implied by $\mathit{body}^+(r) \subseteq \mathcal{M}$ and $\mathit{body}^-(r) \cap \mathcal{M} = \emptyset$.

A *Datalog program* $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ is a logic program where all rules $r \in \mathcal{R}$ have the form

$$H \leftarrow P_1, \ldots, P_n.$$

i.e., have no negative atoms in the body and exactly one atom in the head. In contrast to general ASPs, each Datalog program has a unique stable model $\mathcal{M}$, called the *canonical model*, or just model. This canonical model is usually computed using a *seminaive evaluation* (Abiteboul, Hull, and Vianu 1995, Ch. 13). This is a bottom-up approach that iteratively derives new atoms from previously derived ones. Specifically, it tracks in which iteration each atom was derived, and must use at least one atom derived in iteration i to unify rules and derive new atoms in iteration $i+1$. All facts in $\mathcal{F}$ are derived in iteration 1. The algorithm iteratively derives more atoms until a fix-point is reached (i.e., no new atom is derived during an iteration). The canonical model $\mathcal{M}$ contains all atoms derived this way.

[1]With some abuse of notation, we use set-theoretical symbols with terms, although they are ordered sequences.

Each rule r of a Datalog program is associated with its *primal graph* $\mathcal{G}_r = \langle V(\mathcal{G}_r), E(\mathcal{G}_r) \rangle$, where $V(\mathcal{G}_r) = \mathit{vars}(r)$ and there is an edge $\{v_1, v_2\} \in E(\mathcal{G}_r)$ iff variables v_1 and v_2 jointly occur in an atom of the rule, i.e., if $v_1, v_2 \in \mathit{vars}(\boldsymbol{t})$ for some $P(\boldsymbol{t}) \in \mathit{head}(r) \cup \mathit{body}(r)$.

A *tree decomposition* of a graph $\mathcal{G} = \langle V(\mathcal{G}), E(\mathcal{G}) \rangle$ is a tuple $\langle T, \chi \rangle$ consisting of a tree $T = \langle V(T), E(T) \rangle$ and a function $\chi : V(T) \to 2^{V(\mathcal{G})}$ mapping tree nodes to *bags* of graph nodes. The function has to satisfy that (1) for each edge $\{v_1, v_2\} \in E(\mathcal{G})$, there exists $n \in V(T)$ such that $\{v_1, v_2\} \subseteq \chi(n)$, and (2) for every vertex $v \in V(\mathcal{G})$, the set $\{n \in V(T) \mid v \in \chi(n)\}$ induces a connected subtree of T. The *width* of a tree decomposition is $w - 1$, where w is the size of the largest bag. The *treewidth* $tw(\mathcal{G})$ of $\mathcal{G}$ is the minimum treewidth among all tree decompositions of $\mathcal{G}$. If $tw(\mathcal{G}_r) = 1$ for a rule r, then $\mathcal{G}$ and r are called *acyclic*.

We are particularly interested in Datalog programs that contain rules with low treewidth, as these programs can be grounded efficiently (Morak and Woltran 2012).

Classical Planning We consider classical PDDL planning tasks in STRIPS with inequalities. Such a task is a tuple $\Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, I, G \rangle$ with the components described below. Sets $\mathcal{P}$ and $\mathcal{C}$ are the *predicate symbols* and *constants* of our first-order language. We assume that the binary predicate symbol $\neq$ is in $\mathcal{P}$ and represents the inequality relation.

An action $A \in \mathcal{A}$ consists of three sets of atoms: a *precondition* $\mathit{pre}(A)$, an *add list* $\mathit{add}(A)$, and a *delete list* $\mathit{del}(A)$. We use $\mathit{vars}(A)$ for the set of variables occurring in any atom in one of the three sets. If $\mathit{vars}(A) = \emptyset$, we call A a *ground action* and if all actions in a task are ground, we call the task a *ground task*. We sometimes use the word "*lifted*" to emphasize that a task or action is not ground.

A state s is a set of ground atoms. A ground action A is *applicable* in s if $\mathit{pre}(A) \subseteq s$. To accurately represent inequalities we assume for this definition that all states implicitly contain $c_1 \neq c_2$ for every pair of distinct constants $c_1, c_2 \in \mathcal{C}$. Applying action A in state s leads to the *successor state* $\mathit{succ}(s, A) = (s \setminus \mathit{del}(A)) \cup \mathit{add}(A)$. A sequence of actions $\pi = \langle A_1, \ldots, A_n \rangle$ is applicable in a state s_0 and has $\mathit{succ}(s_0, \pi) = s_n$ if there are states $s_1, \ldots, s_{n-1}$ where A_i is applicable in s_{i-1} and $\mathit{succ}(s_{i-1}, A_i) = s_i$ for all $i \leq n$.

The *initial state* I of a task is a state and the *goal condition* G is a set of ground atoms. We call states s with $G \subseteq s$ *goal states*. We want to find a *plan*, i.e., a sequence of ground actions π applicable in I such that $\mathit{succ}(I, \pi)$ is a goal state.

Other planning formalisms also define *types* for constants and restrict some variables, so they can only be instantiated with constants of the correct type. It is common (e.g., Helmert 2009) to compile type information away by introducing *type predicates type-T* for each type T. The initial state I is augmented with *type-$T(c)$* for all constants $c \in \mathcal{C}$ of type T (using a default type for constants without a type). Type information on action parameters can then be compiled away by adding type predicates to the action's precondition. We assume our tasks contain such type predicates.

Many planners look for a plan by translating the input into a ground task. The naive way of grounding every action A with every possible combination of constants is intractable, so translation algorithms try to use only relevant substitutions. A common approach is to create only ground actions with *relaxed reachable* preconditions, i.e., with preconditions that can be satisfied when the delete list of all actions is replaced by $\emptyset$. Given a planning task Π, we denote its *delete-relaxation* as Π^+.

Baseline: Fast Downward's Grounder

Perhaps the most commonly used algorithm to obtain a propositional planning task from a lifted representation is the one by Helmert (2009). Helmert *translates* a lifted task into a propositional one in four steps: 1. normalization; 2. invariant synthesis; 3. grounding; 4. generation of the final task. Each one of these steps can be a bottleneck, depending on the domain, but Helmert reports that in common planning domains, around 70% of the translation time is spent on grounding, so we focus on this step.

Given planning task $\Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, I, G \rangle$, the algorithm first encodes the delete-relaxation Π^+ as a Datalog program $\mathcal{D}_{\Pi^+} = \langle \mathcal{F}, \mathcal{R} \rangle$ with facts $\mathcal{F} = I$ and the following rules:

1. For each $A \in \mathcal{A}$ with $\mathit{pre}(A) = \{q_1(\boldsymbol{t}_1), \dots, q_n(\boldsymbol{t}_n)\}$ and $\mathit{vars}(A) = \boldsymbol{t}$, $\mathcal{R}$ contains the *action applicability rule*
$$A\text{-applicable}(\boldsymbol{t}) \leftarrow q_1(\boldsymbol{t}_1), \dots, q_n(\boldsymbol{t}_n).$$
where A-applicable is called an *action predicate*.
2. For each $A \in \mathcal{A}$ with $\mathit{vars}(A) = \boldsymbol{t}$ and for each $P(\boldsymbol{t}') \in \mathit{add}(A)$, $\mathcal{R}$ contains the *action effect rule*
$$P(\boldsymbol{t}') \leftarrow A\text{-applicable}(\boldsymbol{t}).$$
3. For $G = \{g_1(\boldsymbol{t_1}), \dots, g_n(\boldsymbol{t_n})\}$, $\mathcal{R}$ contains the *goal rule*:
$$\mathit{goal} \leftarrow g_1(\boldsymbol{t_1}), \dots, g_n(\boldsymbol{t_n}).$$

The normalization step guarantees that all rules are safe.

Helmert (2009) showed that the canonical model $\mathcal{M}$ of $\mathcal{D}_{\Pi^+}$ contains exactly the atoms that are reachable from I in Π^+. The original algorithm by Helmert uses a modification of the seminaive evaluation described earlier but considers only one atom at each iteration. The key is *how to compute the canonical model $\mathcal{M}$ efficiently*. Helmert's grounder uses *rule decompositions* to split large rules into smaller ones, as smaller rules are generally easier to ground and produce smaller intermediate results. It splits all rules until they have unary or binary bodies. This allows for smarter data structures and an implementation tailored to such rules.

There are two types of rule decompositions in the algorithm. The first selects two atoms $q_1(\boldsymbol{t}_1), q_2(\boldsymbol{t}_2)$ in the body of a rule r and introduces the new rule
$$\mathit{temp}(\boldsymbol{t}) \leftarrow q_1(\boldsymbol{t}_1), q_2(\boldsymbol{t}_2).$$
where $\boldsymbol{t}$ contains all terms in $\boldsymbol{t}_1$ or $\boldsymbol{t}_2$ that occur in other atoms of r and the *temporary predicate temp* is a fresh predicate symbol. It then replaces $q_1(\boldsymbol{t}_1)$ and $q_2(\boldsymbol{t}_2)$ in r with $\mathit{temp}(\boldsymbol{t})$. This is called a *join decomposition* because the new rule enforces the grounder to join q_1 and q_2.

The second type of decomposition chooses an atom $q(\boldsymbol{t})$ from the body of a rule r such that there is a *variable* $v \in \boldsymbol{t}$ that does not occur anywhere else in r. It then adds the rule
$$\mathit{temp}(\boldsymbol{t} \setminus v) \leftarrow q(\boldsymbol{t}).$$
where temp is a fresh predicate symbol, and replaces $q(\boldsymbol{t})$ with $\mathit{temp}(\boldsymbol{t} \setminus v)$ in r. This is called a *projection*.

What is left is how to choose atoms for the decomposition. This choice follows two basic principles: try to project away unnecessary variables as early as possible, and try to join the maximum number of variables (with join decompositions). In the algorithm, this is done greedily based on the total number of variables and the number of joining variables in each relation. Helmert reports that, although it works well in general, there are cases (such as the Rovers domain) where this greedy decomposition is bad. Due to lack of space, we do not go into further details and refer to the paper for more information (Helmert 2009, Section 6).

Note that different choices on how to decompose rules will lead to a different number of temporary predicates. This is one of the main sources of overhead with rule rewriting.

Experiments Throughout the paper we present several methods that complement each other. To motivate new methods better, we present the experimental results for each method as soon as it is introduced. Our first experiment compares the grounder by Helmert to an off-the-shelf ASP grounder, called `gringo` (Gebser et al. 2011). To the best of our knowledge, while others compared to improved versions of Helmert's algorithm (e.g., Fišer 2020), this is the first systematic comparison to an off-the-shelf grounder.

The current implementation of Helmert's algorithm used in Fast Downward[2] is implemented in Python. We use PyPy to speed up the evaluation (compared to the regular Python3 interpreter used in Fast Downward) but to compare it to `gringo` on equal footing, we also reimplemented the algorithm in C++. We compare both implementations against `gringo` with the original Datalog program $\mathcal{D}_{\Pi^+}$ as input.

Our experiments use the hard-to-ground (HTG) data set by Lauer et al. (2021) containing 862 tasks, divided into 8 different domains. All experiments were run on Intel Xeon Silver 4114 processors running at 2.2 GHz. We use a time limit of 30 minutes and a memory limit of 3.8 GiB per task. Our source code is available online (Corrêa et al. 2023).

The first three columns of Table 1 show the number of ground tasks for the three algorithms described above: `FD`, the baseline implementation from Fast Downward run with PyPy; `FD`$^{++}$, our reimplementation in C++; and `gringo`, the off-the-shelf state-of-the-art grounder.

`FD`$^{++}$ and `gringo` can ground a similar number of tasks in most of the domains, but in blocksworld, rovers, and visitall, `gringo` grounds more tasks. The poor performance of `FD`/`FD`$^{++}$ in rovers was already pointed out by Helmert (2009). Looking further into this domain, the challenge is that the set of initial facts is too large and most facts have the same predicate symbol. Given that the preconditions in rovers are also non-trivial (Corrêa et al. 2020), this makes it hard to rewrite the rules in a good way: while we want to decompose the rules to perform clever joins, we also want to minimize the number of joins over large relations. ASP

[2]The strategy to decompose joins in Fast Downward's current implementation no longer matches the paper: it prefers to decompose atoms with fewer variables while the paper prefers atoms with many variables. We use the first option as it produced better results.

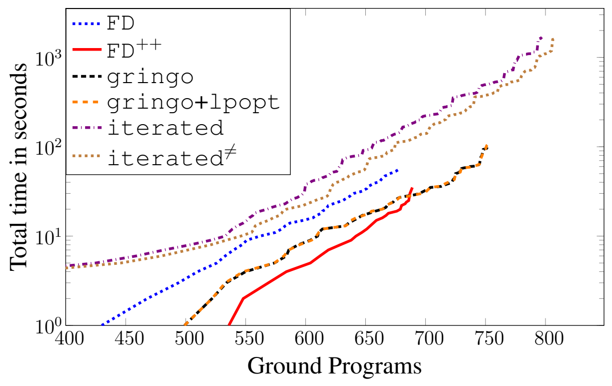

Figure 1: Number of tasks ground per time (in seconds).

grounders, such as `gringo`, apply on-the-fly join ordering techniques based on estimated selectivity and relation size, which seems to be important here.

We also analyzed the run time of all methods. Figure 1 shows the number of ground programs over time. FD^{++} is usually faster than `gringo` in tasks that both can ground. This makes sense as `gringo` is a general ASP solver that can deal with more expressive logic programs and hence has an overhead on data structures, while FD^{++} is tailored to our type of problem. However, as tasks get larger and more complicated to ground, FD^{++} quickly reaches the memory limit while `gringo` manages to ground some of them.

Our hypothesis is that the rule decomposition used in FD^{++} leads to too many temporary atoms and intermediate joins, which consumes too much memory. To test this, we evaluated giving `gringo` the program $\mathcal{D}_{\Pi^+}$ *after* the decompositions used by the FD^{++} algorithm. The results are also shown in Table 1 as "G+FD^{++}". This indeed decreased the number of programs `gringo` could ground to 703, bringing it much closer to the performance of FD^{++}. Most of the loss came from the domain rovers, where the decomposition works poorly. With the decomposition `gringo` could only ground the 5 programs that FD^{++} can ground, while it could solve all of them without the decomposition. However, G+FD^{++} still grounds more programs than FD^{++}. This indicates that the superior performance of `gringo` is not only due to the different input.

Grounding Using Structural Decompositions

Ideally, one would like to exploit the structure of the Datalog program more systematically than FD^{++}'s heuristic approach. Morak and Woltran (2012) show that programs with bounded treewidth can be grounded efficiently. They achieve this result by decomposing the rules of a Datalog program based on a tree decomposition of their primal graph. This decomposition introduces additional auxiliary predicates – similar to the ones used by Helmert (2009). Thereby, it provides indirect guidance to the grounder based on the structure of the rules.

Roughly speaking, for a given rule r the technique first finds a tree decomposition $T = \langle V(T), E(T)\rangle$. For each node $n \in V(T)$ with parent node p_n, it then creates a fresh predicate temp_n and introduces a rule

$$\mathit{temp}_n(\boldsymbol{Y}_n) \leftarrow \{a \in \mathit{body}(r) \mid \mathit{vars}(a) \subseteq \chi(n)\} \cup \{\mathit{temp}_m \mid m \text{ is a child of } n \text{ in } T\}.$$

where $\boldsymbol{Y}_n = \chi(n) \cup \chi(p_n)$. It finally replaces the original rule with this set of n new rules and the rule

$$\mathit{head}(r) \leftarrow \mathit{temp}_{\mathit{root}}(\boldsymbol{Y}_{\mathit{root}}).$$

where $\mathit{temp}_{\mathit{root}}$ is the root node of T. These rules will produce the same instantiations of $\mathit{head}(r)$ as the original one.

Experiments We use `lpopt` (Bichler, Morak, and Woltran 2016), an implementation of the technique described above, to evaluate more systematic decompositions. The tree decomposition computed by `lpopt` has no optimality guarantee because computing an optimal tree decomposition is **NP**-hard (Arnborg, Corneil, and Proskurowski 1987). Table 2 presents statistical data on the treewidth w in our domains. We aggregate the information of domains that contain different domain files (e.g., visitall) into a single row. The rows where column A has a checkmark ✓ correspond to our original Datalog program. The other rows will be discussed in the next section. Some domains (e.g., organic-synthesis and pipesworld) have rules with very high treewidth w, which is caused by the action predicates: as these usually have high arity and include all variables of the rule, they force the tree decomposition to keep all variables until the root, which increases w. In cases where w is large, we do not expect `lpopt`'s rule rewriting to help much.

Table 1 shows the number of ground tasks in our benchmark set using `gringo+lpopt` (in the **Action Predicates** block). This technique grounds exactly the same tasks as `gringo` without `lpopt`. Looking at the number of ground tasks over time in Figure 1 shows that the performance of `gringo` with and without `lpopt` is almost identical. This is expected, because action applicability rules have all variables in the head. Thus, `lpopt` cannot project out any variables during the decomposition, so intermediate predicates end up being very large. Perhaps surprisingly, this happens in all domains, and not only in domains that have large treewidth. These large predicates consume a lot of memory. In fact, all instances that `gringo+lpopt` (and also simply `gringo`) cannot ground are due to running out of memory.

Avoiding to Ground Actions

What happens then if we remove these action predicates from our Datalog programs? In lifted planning, Corrêa et al. (2021) show how to remove action predicates from the Datalog program while preserving all other atoms in the model. Given the following action applicability rule and effect rule

$$A\text{-applicable}(\boldsymbol{t}) \leftarrow q_1(\boldsymbol{t}_1), \dots, q_n(\boldsymbol{t}_n).$$
$$P(\boldsymbol{t}') \leftarrow A\text{-applicable}(\boldsymbol{t}).$$

this method replaces both rules with the *simplified rule*

$$P(\boldsymbol{t}') \leftarrow q_1(\boldsymbol{t}_1), \dots, q_n(\boldsymbol{t}_n).$$

(Note that if A has multiple action effect rules, the same procedure would be done for each of them.)

Domain	Action Predicates					No Action Predicates			Iterated Solving	
	FD	FD^{++}	G	G+FD	G+L	FD^{++}	G	G+L	iterated	iterated$^{\neq}$
blocksworld (40)	32	36	**40**	**40**	**40**	**40**	**40**	**40**	**40**	**40**
childsnack (144)	118	**120**	**120**	**120**	**120**	**144**	**144**	**144**	142	**144**
genome-edit-dist. (312)	**312**	**312**	**312**	**312**	**312**	**312**	**312**	**312**	**312**	**312**
logistics (40)	**40**	**40**	**40**	**40**	**40**	**40**	**40**	**40**	**37**	**37**
organic-synthesis (56)	**21**	**21**	**21**	**21**	**21**	**56**	41	**56**	23	**31**
pipesworld-tankage (50)	32	**35**	**35**	**35**	**35**	40	**50**	**50**	**50**	**50**
rovers (40)	4	5	**40**	5	**40**	**40**	**40**	**40**	**20**	**20**
visitall-multidim. (180)	120	120	**144**	132	**144**	120	**180**	**180**	**174**	**174**
Total (862)	679	689	**752**	703	**752**	802	847	**862**	798	**808**

Table 1: Number of ground tasks by domain for different algorithms. We abbreviate `gringo` with `G`, and `lpopt` with `L`. The results are divided into three different blocks: results for programs with action predicates, without action predicates, and the grounding via iterated solving approach. For each block, the best result for each domain is marked in bold.

Domain	A	w-range	average w
blocksworld	✓	1–2	1.54 ± 0.52
	✗	1–2	1.33 ± 0.50
childsnack	✓	2–10	4.89 ± 2.52
	✗	2–6	2.80 ± 1.30
genome-edit-distance	✓	0–5	2.21 ± 0.60
	✗	1–5	1.90 ± 0.48
logistics	✓	3–4	3.17 ± 0.39
	✗	2–3	2.83 ± 0.41
organic-synthesis	✓	3–22	10.55 ± 3.99
	✗	2–3	2.10 ± 0.29
pipesworld-tankage	✓	9–12	10.62 ± 1.53
	✗	3–5	3.73 ± 0.63
rovers	✓	2–6	4.23 ± 1.27
	✗	2–3	2.35 ± 0.49
visitall-multidimensional	✓	4–6	5.17 ± 0.00
	✗	4–6	5.17 ± 0.00

Table 2: Width w of the tree decomposition computed by `lpopt`. Column A indicates whether action predicates are considered.

Throughout the paper we refer to the Datalog program without action predicates as the *simplified program*. This simplification significantly improves the performance of lifted planners (Corrêa et al. 2021) which do not need the action predicates. However, we cannot know all relaxed-reachable ground actions of our task without these predicates. We solve this problem in the next section, but we first experimentally investigate the simplicity of these programs.

Experiments We compared the methods using the simplified program as input. Table 1 shows the results under the block **No Action Predicates**. The increase in performance is clear: FD^{++} goes from 689 to 802 ground programs; `gringo` goes from 752 to 847; `gringo+lpopt` goes from 752 to all 862 programs. This is due to the simpler structure of the problem. This can be seen in the statistics on treewidth of simplified programs, shown in Table 2. Comparing the statistics for the program with action predicates (column A with ✓) and programs without action predicates (column A with ✗), the decompositions found for the latter have much lower treewidth. In organic-synthesis and pipesworld, the decrease in average width is more than 70%. This means that a treewidth-based approach like `gringo+lpopt` can potentially work much better – and that is visible in the results of Table 1. The only domain where the treewidth is not reduced is visitall-multidimensional. In this domain, the predicate indicating the current position of the agent also has large arity and thus is as harmful as action predicates.

Both FD^{++} and `gringo` fail only in one domain. The former cannot ground all simplified programs of the visitall domain, while the latter fails for some larger organic-synthesis instances. The greedy decomposition used in FD^{++} does not seem to work well with the larger predicates in visitall. For `gringo`, the main issue is that some of the organic-synthesis tasks have large intermediate relations if the rule bodies are joined in a bad order. `gringo+lpopt` can overcome both problems. The tree decomposition helps to amortize the impact of larger predicates and to reduce the size of intermediate joins. This allows `gringo+lpopt` to ground all simplified programs in our benchmark set.

Furthermore, `gringo+lpopt` is faster than FD^{++} and `gringo` even for tasks that all can ground. Figure 2 shows the run time for both `gringo` and `gringo+lpopt`. Clearly, `lpopt` improves the performance of `gringo`. The only exceptions above the diagonal are instances from the domain logistics. The tree decomposition found by `lpopt` seems to do more harm than good here. Logistics has very simple rules so we believe that this is another case where the size of the relations and selectivity might have more impact than decompositions.

Grounding via Iterated Solving

On the one hand, the simplified program from the previous section is much simpler to ground while it still contains all relaxed-reachable atoms of the task. On the other hand, it does not contain any information about the ground actions

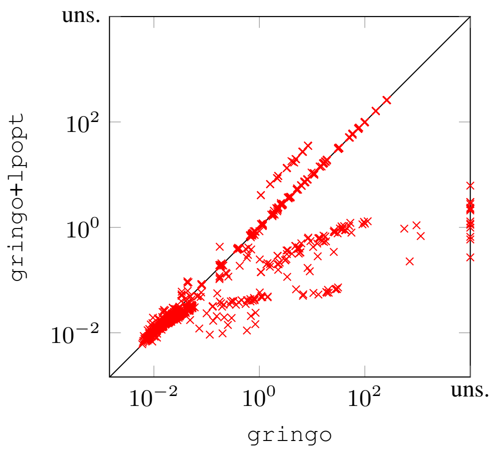

Figure 2: Run time comparison between `gringo` and `gringo+lpopt` on the simplified Datalog programs.

of the task. How do we obtain the set of ground actions from the relaxed-reachable atoms?

One trivial solution is to unify the preconditions of each action with the set of reachable atoms. In other words, we would *join* the preconditions of the action, similarly as done by Corrêa et al. (2020). Although simple in theory, this is not straightforward: the intermediate results of this join can be exponentially large (Ullman 1989; Corrêa et al. 2020).

To mitigate this problem, we introduce a two-phase method, inspired by the techniques of grounding via solving (Besin, Hecher, and Woltran 2022). First, we use the simplified Datalog programs to compute a model $\mathcal{M}$, containing all relaxed-reachable atoms. For each action schema, we then construct a logic program with facts $\mathcal{M}$ where every stable model represents one relaxed-reachable instantiation of the action schema.

The program we construct uses *choice rules* of the form $1\,\{p(X) : r(X)\}\,1$ expressing that exactly one $p(X)$ has to be chosen for which $r(X)$ is in the model. Choice rules can be seen as syntactic sugar in logic programs and can be simulated with two additional predicates *has-p* and p' and the following rules. Rules *has-p* $\leftarrow p(X)$ and $\bot \leftarrow \neg$*has-p* ensure that at least one $p(X)$ is in the model; rule $\bot \leftarrow p(X), p(Y), X \neq Y$ ensures that at most one $p(X)$ is in the model; and rule $p(X) \vee p'(X) \leftarrow r(X)$ allows (but does not force) the model to contain $p(X)$ if it contains $r(X)$.

The program also uses type information compiled into the planning task as type predicates *type-T* as discussed earlier.

For a set of facts $\mathcal{M}$ and an action applicability rule r

$$A\text{-applicable}(\boldsymbol{t}) \leftarrow q_1(\boldsymbol{t}_1), \dots, q_n(\boldsymbol{t}_n).$$

we create the logic program $\mathcal{L} = \langle \mathcal{F}, \mathcal{R} \rangle$ with $\mathcal{F} = \mathcal{M}$, and rules $\mathcal{R}$ as follows. For every variable $V \in \mathit{vars}(r)$ with type T, we introduce a fresh predicate V*-assign* and the following *choice rule*:

$$1\,\{V\textit{-assign}(X) : \textit{type-}T(X)\}\,1.$$

where X is a new variable. This rule forces the stable model to pick exactly one constant of the correct type for each variable and thus form a variable assignment.

Further, for every (non-ground) atom $q_i(\boldsymbol{t}_i) \in \mathit{body}(r)$ with $\mathit{vars}(q_i(\boldsymbol{t}_i)) = \{V_1, \dots, V_k\}$, we introduce the rule

$$\bot \leftarrow V_1\text{-assign}(X_1), \dots, V_k\text{-assign}(X_k), \neg q_i(X_1, \dots, X_k).$$

This rule guarantees that the assignment encoded in the V-assign predicates is consistent with the instantiations of q_i in all stable models of $\mathcal{L}$.

The program $\mathcal{L}$ has multiple stable models. Each one corresponds to one ground action of A. For each variable V, the stable model of has exactly one atom V-assign(c), for the constant c that instantiates V in the ground action.

The overall approach is then to *iteratively solve* and enumerate all stable models of $\mathcal{L}$, thereby constructing all relaxed-reachable actions. This approach relies on the common guess-and-check technique of ASP solvers. The advantage is that we generate one stable model per iteration without keeping track of previous ones. This keeps memory usage in check, as we do not have to consider all ground action at the same time. One could also use conjunctive query solvers, but we do not know any off-the-shelf tool that implements such iterative enumeration.

This new technique can be applied to ground other logic programs that are not related to planning. In particular, it can help in cases where the density of structures is such that they cannot be sufficiently exploited any more. Our approach is specifically designed for positive programs with large predicate arities in the rule heads of programs, which is the case for those rules corresponding to actions in planning.

Experiments We call the two-phase approach described above `iterated`. Our implementation first uses `gringo+lpopt` to get $\mathcal{M}$ from the simplified program, and then uses `clingo` (Gebser et al. 2019) to iteratively generate grounded actions from the logic programs above. Table 1 shows the results in the second-to-last column. Compared to the methods that ground action predicates, `iterated` grounds more tasks in total. It is better in the domains childsnack, organic-synthesis, and pipesworld, but worse in the domains rovers and logistics. As discussed before, both of the latter domains do not seem to work well with methods only exploiting structural properties.

While `iterated` can solve tasks where other methods run out of memory, it is much slower than any other algorithm. Figure 1 compares the run time of `iterated` to all other methods. Even for simple tasks, `iterated` takes more time than any other algorithm – including the Python implementation `FD`. Noticeable overhead comes from the communication of the different components, i.e., the effort of setting up the model and parsing the result.

Still, some tasks were not ground by any method (ignoring methods that ground only simplified programs). More precisely, there are 25 organic-synthesis tasks, and 6 visitall-multidimensional tasks that cannot be grounded by any algorithm. To see how far we are from grounding these tasks, we used a model counter to check (without even generating the model) the number of ground actions in these tasks. We use the model counter and the preprocessor by Lagniez and Marquis (2014, 2017). To do it efficiently, we apply our two-phase approach but, in the second phase, instead of enumerating all ground actions, we translate the program to a propositional formula with the tools by Janhunen (2006), and then simply count the models. For more details on state-of-the-art model counters, we refer to the survey by Fichte, Hecher, and Hamiti (2021).

For visitall-multidimensional, all remaining tasks have $76 \cdot 10^6$ ground actions. This still seems possible to ground, as `iterated` was able to ground larger tasks (e.g., some pipesworld tasks have more than 10^8 ground actions). In contrast, the smallest (in number of ground actions) task from organic-synthesis that we cannot ground has $17.3 \cdot 10^{12}$ ground actions. The largest one has $60 \cdot 10^{35}$ ground actions. Even considering an oracle model that provides the list of relaxed-reachable atoms and actions instantly, these tasks seem out of reach of ground planners. The amount of storage necessary to simply represent these tasks is prohibitive. To deal with these tasks, we could use lifted planners (e.g., Horčík and Fišer 2021; Höller and Behnke 2022) or grounders that compute models more restricted than relaxed-reachability (e.g., Gnad et al. 2019; Fišer 2020).

More Informed Logic Programs

An orthogonal way to improve Datalog-based grounders for planning is to use additional information to refine the model and thus make it smaller. One example of this approach is the work by Fišer (2020) that uses lifted mutexes to improve grounding. Fišer interleaves the `FD` algorithm with a filtering to remove ground atoms violating mutexes.

We propose an approach that does not need a modification of `gringo` or `iterated`. Instead of changing the algorithm, we simply modify the input Datalog program to consider *negated static predicates* in preconditions. Static predicates are those predicates that only occur in preconditions and not in effects. When we create the Datalog program of the task, static predicates never occur in the head of any rule (they are what is called *extensional*). We can exploit this by adding any negatively occurring static predicate to the body of its action applicability rule (also negatively). This changes the canonical model of the Datalog program but it does not influence its uniqueness (Ullman 1988, 1989).

In our benchmark set, the only occurrences of negated static preconditions are inequality constraints. (Other PDDL domains, such as Termes and Tetris, have different negated static predicates in action preconditions.) Below, we refer directly to inequality constraints but the technique would work for any static predicate. Inequalities have an additional advantage, though, as they can be treated as a built-in predicate by `gringo`. Note that this is already the case in PDDL.

The `FD` algorithm already removes actions that violate inequalities in a postprocessing step. Let us call these actions as *impossible actions*. However, `FD` can still produce actions that are only relaxed-reachable via impossible actions. Incorporating inequalities into the logic programs directly solves this problem. To illustrate, consider the example in Figure 3. The canonical model of this program is $\mathcal{M} = \{p(1)\}$, and the task is relaxed unsolvable (because G is not in $\mathcal{M}$). If $X \neq Y$ is removed from the program, then $\mathcal{M} = \{p(1), A\text{-applicable}(1,1), b(1), G\}$. The `FD` algorithm postprocesses the ground actions and identifies that

$$\begin{aligned} p(1)&. \\ A\text{-applicable}(X,Y) &\leftarrow p(X), p(Y), X \neq Y. \\ b(X) &\leftarrow A\text{-applicable}(X,Y). \\ G &\leftarrow b(1). \end{aligned}$$

Figure 3: Datalog program where ignoring inequality would produce a larger ground task, even after postprocessing.

A-applicable$(1,1)$ is an impossible action. Hence, it will be discarded. But it cannot do that for $b(1)$ and G unless it keeps track of all derivations. So this task is still considered solvable by `FD` although it is not.

Experiments We tested our algorithms using inequalities in the logic programs. We report the number of ground tasks for `iterated` with inequalities, called `iterated`$^{\neq}$, in Table 1. For this algorithm, we only use inequalities in the second phase. We do not use them in the first phase because they harm `lpopt`: inequalities make the preconditions very dense and hence the treewidth much higher. For example, the average treewidth found by `lpopt` goes from $2.1(\pm 0.29)$ up to $4.6(\pm 1.45)$ in the simplified programs of organic-synthesis.

Compared to `iterated`, `iterated`$^{\neq}$ grounds 2 additional tasks in childsnack and 8 additional tasks in organic-synthesis. Inequalities also occur in genome-edit-distance, but the instances are not challenging enough to observe any difference. However, even in tasks that both methods can ground, considering inequalities improves the speed of the grounder. Figure 1 also shows the time of `iterated`$^{\neq}$, and is has a clear edge over `iterated` in terms of time. In an extreme example, `iterated` took 840s to ground a task, while `iterated`$^{\neq}$ needed only 7s.

Even with inequalities, there are still tasks in organic-synthesis that could not be grounded. We repeated our counting experiment but now also considering inequalities. In this case, the smallest instance in organic-synthesis has “only” $11 \cdot 10^9$ actions, and the largest has $81 \cdot 10^{33}$. Although this is a reduction of 10^3 actions in each case, these numbers are still to large for current grounders and planners. The numbers for visitall-multidimensional remain unchanged.

Solving Planning Tasks

So far, we have focused on the grounding step. However, this is not the only complicated step of the translation from PDDL to a propositional task. Moreover, tasks that our algorithms can now ground might still be out of reach of planners, producing no additional gain from better grounding.

As a proof of concept, we compared the coverage of LAMA (Richter and Westphal 2010) using different grounders. LAMA originally uses the `FD` algorithm. We tested replacing it with `gringo`, and with `iterated`$^{\neq}$. Table 3 shows the results. In 4 out of the 8 domains, all methods have the same coverage. With its original grounder (`FD`), LAMA achieves a better coverage in 2 domains (pipesworld

Domain	FD	G	iterated$^{\neq}$
blocksworld (40)	**8**	**8**	**8**
childsnack (144)	**103**	**103**	**103**
genome-edit-dist. (312)	**312**	**312**	**312**
logistics (40)	**4**	**4**	**4**
organic-synthesis (56)	18	**27**	23
pipesworld-tankage (50)	**15**	14	14
rovers (40)	4	**40**	20
visitall-multidim. (180)	**84**	79	78
Total (862)	548	**587**	562

Table 3: Coverage of LAMA using different grounders. FD is the original algorithm used in LAMA; G is the gringo algorithm; iterated$^{\neq}$ uses the two-phase grounding approach incorporating inequalities.

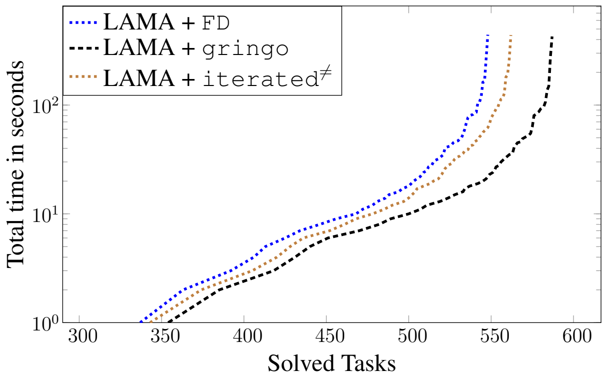

Figure 4: Number of tasks solved over time (in seconds) using LAMA with different grounders.

and visitall-multidim.), while using gringo solves more tasks in rovers and organic synthesis. The number of additional instances solved by LAMA with gringo in these two domains is surprisingly high, particularly in rovers. All tasks in rovers are easy to solve and were beyond LAMA's capacity only due to its grounder. The same can be said about organic-synthesis, a domain known for having short plans.

We also analyze the run time of each method. Figure 4 compares the number of instances solved over time. There is a very clear ordering there: FD is the slowest, followed by iterated$^{\neq}$ and gringo. This agrees with our previous observation (see Figure 1) that iterated$^{\neq}$ is much slower than gringo. However, in the context of *solving* the planning tasks, the slow-down caused by the two-phases of iterated$^{\neq}$ is harmful, as it does not leave enough time for the search component of LAMA.

Overall, our experiments show that using new grounding algorithms helps to increase the number of solved tasks.

Related Work

Our work focuses on Datalog-based grounders for classical planning, but there are other grounding algorithms in the literature. IPP (Köhler and Hoffmann 2000) grounds action schemas one by one, and prunes partially ground actions as soon as they violate static preconditions. One issue with this approach is that the final model is even larger than the relaxed-reachable one. The grounder used in FF (Hoffmann and Nebel 2001) also relies on this pruning technique. It first executes the IPP grounder and then reduces the set of atoms and actions by identifying which ones are relaxed-reachable. Helmert (2009) reports that both of these grounders are usually fast, but FD has better scalability. To add this pruning technique, we would need to change the grounding algorithm, while our approach on how to exploit negated static preconditions only modifies the logic program used as input.

There also are techniques in the planning literature to improve the grounder by Helmert (2009). Fišer (2020) presents an algorithm to exploit mutex groups during grounding. The algorithm uses fact-alternating mutex groups (fam-groups) to prune actions identified as unreachable during grounding. It is not clear how one could incorporate these ideas into our algorithms. Although fam-groups could be encoded as choice rules and aggregates (Gebser et al. 2019) into our logic programs, the extra rules would need to be grounded before finding a stable model, which defeats the purpose of Fišer's algorithm. However, it should be possible to incorporate the pruning based on fam-groups into the seminaive algorithm (as in the original paper).

Another alternative is the incremental grounding by Gnad et al. (2019). Their idea is to ground only part of the relaxed-reachable atoms and actions first, then try to find a plan with this limited set. If no plan is found, the process restarts with more atoms and actions. In their work, Gnad et al. use machine learning to identify the subset of atoms and actions that should be grounded. Their method could be integrated with our iterated algorithm: after computing the first phase (using gringo+lpopt) we could compute only some stable models of the second phase (e.g., a maximum number of instantiations per action schema).

Conclusion

In this work, we studied alternatives to the grounding algorithm by Helmert (2009), which uses logic programming to find the relaxed-reachable actions of a planning task. Our empirical results showed that replacing the original FD algorithm with more modern grounders for logic programs yields superior results in terms of the number of ground tasks. We presented a more sophisticated method, called iterated, that decouples the grounding procedure into two phases: one to obtain all relaxed-reachable atoms, and one to obtain all relaxed-reachable actions. While this method can ground more logic programs, it is also slower than off-the-shelf grounders, such as gringo. Our experiments also showed that most tasks not grounded by any method are impractical to ground, as they have an enormous number of actions. To deal with such tasks, one should look into grounders not based on delete-relaxation or lifted planners.

Both gringo and iterated help classical planners solve more tasks. This means that some tasks that we can now ground were already in reach of classical planners. Outside the planning context, we believe that iterated could be used to ground Datalog programs in general.

Acknowledgments

Augusto B. Corrêa was funded by the Swiss National Science Foundation (SNSF) as part of the project “Lifted and Generalized Representations for Classical Planning” (LGR-Plan). Furthermore, the University of Basel authors were also partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under grant agreement no. 952215. Markus Hecher and Stefan Woltran were funded by the Austrian Science Fund (FWF), grants J 4656 and P 32830, the Society for Research Funding in Lower Austria (GFF, Gesellschaft für Forschungsförderung NÖ) grant ExzF-0004, as well as the Vienna Science and Technology Fund (WWTF) grant ICT19-065. Davide Mario Longo’s work was supported by the FWF projects P30930-N35 and W1255-N23.

References

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley.

Arnborg, S.; Corneil, D. G.; and Proskurowski, A. 1987. Complexity of Finding Embeddings in a k-Tree. *SIAM J. Algebraic Discrete Methods*, 8(2): 277–284.

Besin, V.; Hecher, M.; and Woltran, S. 2022. Body-Decoupled Grounding via Solving: A Novel Approach on the ASP Bottleneck. In *Proc. IJCAI 2022*, 2546–2552.

Bichler, M.; Morak, M.; and Woltran, S. 2016. lpopt: A Rule Optimization Tool for Answer Set Programming. In *Proceedings of the Twenty-Sixth International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*, 114–130. Springer.

Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. *AIJ*, 129(1): 5–33.

Corrêa, A. B.; Francès, G.; Pommerening, F.; and Helmert, M. 2021. Delete-Relaxation Heuristics for Lifted Classical Planning. In *Proc. ICAPS 2021*, 94–102.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation using Query Optimization Techniques. In *Proc. ICAPS 2020*, 80–89.

Corrêa, A. B.; Hecher, M.; Helmert, M.; Longo, D. M.; Pommerening, F.; and Woltran, S. 2023. Code from the ICAPS 2023 paper “Grounding Planning Tasks Using Tree Decompositions and Iterated Solving”. https://doi.org/10.5281/zenodo.7733252.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3): 374–425.

Fichte, J. K.; Hecher, M.; and Hamiti, F. 2021. The Model Counting Competition 2020. *ACM Journal of Experimental Algorithmics*, 26(13): 1–26.

Fišer, D. 2020. Lifted Fact-Alternating Mutex Groups and Pruned Grounding of Classical Planning Problems. In *Proc. AAAI 2020*, 9835–9842.

Francès, G.; Geffner, H.; Lipovetzky, N.; and Ramiréz, M. 2018. Best-First Width Search in the IPC 2018: Complete, Simulated, and Polynomial Variants. In *IPC-9 Planner Abstracts*, 23–27.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19: 27–82.

Gebser, M.; Kaminski, R.; König, A.; and Schaub, T. 2011. Advances in gringo Series 3. In Delgrande, J. P.; and Faber, W., eds., *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, 345–351. Springer Berlin Heidelberg.

Gnad, D.; Torralba, Á.; Domínguez, M. A.; Areces, C.; and Bustos, F. 2019. Learning How to Ground a Plan – Partial Grounding in Classical Planning. In *Proc. AAAI 2019*, 7602–7609.

Haslum, P. 2011. Computing Genome Edit Distances using Domain-Independent Planning. In *ICAPS 2011 Scheduling and Planning Applications woRKshop*, 45–51.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR*, 26: 191–246.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *AIJ*, 173: 503–535.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR*, 14: 253–302.

Höller, D.; and Behnke, G. 2022. Encoding Lifted Classical Planning in Propositional Logic. In *Proc. ICAPS 2022*, 134–144.

Horčík, R.; and Fišer, D. 2021. Endomorphisms of Lifted Planning Problems. In *Proc. ICAPS 2021*, 174–183.

Immerman, N. 1986. Relational Queries Computable in Polynomial Time. *Information and Control*, 68(1-3): 86–104.

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

Katz, M.; and Hoffmann, J. 2014. Mercury Planner: Pushing the Limits of Partial Delete Relaxation. In *IPC-8 Planner Abstracts*, 43–47.

Köhler, J.; and Hoffmann, J. 2000. On the Instantiation of ADL Operators Involving Arbitrary First-Order Formulas. In *ECAI 2000 PuK Workshop*, 74–82.

Lagniez, J.; and Marquis, P. 2014. Preprocessing for Propositional Model Counting. In *Proc. AAAI 2014*, 2688–2694.

Lagniez, J.; and Marquis, P. 2017. An Improved Decision-DNNF Compiler. In *Proc. IJCAI 2017*, 667–673.

Lauer, P.; Torralba, Á.; Fišer, D.; Höller, D.; Wichlacz, J.; and Hoffmann, J. 2021. Polynomial-Time in PDDL Input Size: Making the Delete Relaxation Feasible for Lifted Planning. In *Proc. IJCAI 2021*, 4119–4126.

Matloob, R.; and Soutchanski, M. 2016. Exploring Organic Synthesis with State-of-the-Art Planning Techniques. In *ICAPS 2016 Scheduling and Planning Applications woRKshop*, 52–61.

Morak, M.; and Woltran, S. 2012. Preprocessing of Complex Non-Ground Rules in Answer Set Programming. Technical Report DBAI-TR-2011-72 (Revised Version), Technische Universität Wien.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR*, 39: 127–177.

Torralba, Á.; Alcázar, V.; Borrajo, D.; Kissmann, P.; and Edelkamp, S. 2014. SymBA*: A Symbolic Bidirectional A* Planner. In *IPC-8 Planner Abstracts*, 105–109.

Ullman, J. D. 1988. *Principles of Database and Knowledge-Base Systems. Volume I: Classical Database Systems*. Computer Science Press.

Ullman, J. D. 1989. *Principles of Database and Knowledge-Base Systems. Volume II: The New Technologies*. Computer Science Press.

Vardi, M. Y. 1982. The Complexity of Relational Query Languages (Extended Abstract). In Lewis, H. R.; Simons, B. B.; Burkhard, W. A.; and Landweber, L. H., eds., *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC ’82)*, 137–146. ACM Press.