Grounding Planning Tasks Using Tree Decompositions and Iterated Solving (Extended Abstract)[†]

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

Classical planning tasks are usually described using first-order languages (Haslum et al. 2019). However, most classical planners use *propositional representations*. Planners then need to *ground* the task before they can solve it. In the planning literature, the most popular grounder is the one used by Fast Downward (Helmert 2006; Helmert 2009), denoted here as `FD`. The key idea of `FD` is to encode a *delete-relaxed* version of the task (Bonet and Geffner 2001) as a Datalog program. Grounding this *Datalog program* gives us all delete-relaxed *reachable atoms* of the task.

In this work, we study how to ground planning tasks more efficiently. Following `FD`, we use Datalog programs to ground planning tasks. Inspired by recent progress in lifted planning, database theory, and algorithmics, we develop a new method to ground these Datalog programs. Our new algorithm grounds more tasks than any other tested grounder.

Grounding Planning Tasks using Datalog

We illustrate the main idea of the `FD` grounder using an example. For a complete formalization and detailed explanation, we refer to the conference version (Corrêa et al. 2023). A planning task Π is associated to a set $\mathcal{P}$ of predicates, a set $\mathcal{C}$ of constants, a set $\mathcal{A}$ of action, an initial state I, and a goal G. Each action has preconditions and effects, which are sets of atoms over $\mathcal{P}$. The initial state I describes which ground atoms are initially true, and the goal G is the set of atoms that we want to make true.

Consider a task Π with a single action A with two parameters x and y, preconditions $\{p(x), p(y), \neg q(x, x)\}$, and effect $\{q(x, y)\}$. Furthermore, let us define I as $\{p(0), p(1)\}$. (The goal G does not influence our simplified example.) `FD` grounds this task by first obtaining its *delete-relaxation* (Bonet and Geffner 2001): it ignores all negated preconditions and negated effects. The delete-relaxation of Π can be encoded as the following Datalog program:

$$p(0).\ p(1).$$
$$A(x, y) \leftarrow p(x), p(y).$$
$$q(x, y) \leftarrow A(x, y).$$

The canonical model of this program has all atoms that are *relaxed-reachable*. This is an overapproximation of all atoms needs to solve the original task, and it might contain atoms that are not reachable in the original (non-relaxed) task. However, this information can still be useful (e.g., to compute heuristics), and guarantees that the ground representation can be computed efficiently (Dantsin et al. 2001) while preserving all solutions of the original task.

To find this canonical model, `FD` first *decomposes the rules* into smaller ones that might be easier to ground. The decomposition strategy enforces that all new rules have a body with at most two atoms. The idea is to simulate a join tree, similarly to what is done by Bichler, Morak, and Woltran (2016). However, the specifics of this decomposition are mostly heuristic. `FD` tries to simply maximize the number of joining variables when decomposing rules. This greedy decision leads to bad decompositions in some cases, which produces too many intermediate atoms.

Our Work

We first compared `FD` to off-the-shelf logic programming grounders, such as `gringo` (Gebser et al. 2011), to estimate how far `FD` is from state-of-the-art grounders. We use a benchmark set containing 862 tasks that are hard to ground (Lauer et al. 2021). Each run had a 4 GiB memory limit and 30 minutes time limit. `FD` grounds 689 of these tasks, while `gringo` can ground 752. As `gringo` is a state-of-the-art grounder, it is unsurprising that it has better performance.

Corrêa et al. (2020) show that most planning domains produce *acyclic* Datalog rules. Logic programs with acyclic rules or, more generally, with rules of low (hyper) treewidth can be ground efficiently (Morak and Woltran 2012). To exploit this, we used `lpopt` (Bichler, Morak, and Woltran 2016) to preprocess the Datalog programs given to `gringo`. By doing so, we decompose the rules to speed up the grounder. This is similar to the decomposition approach used in `FD`, but `lpopt` uses structural information while `FD` does not. We denote this new version as `gringo+lpopt`. Overall, `gringo+lpopt` cannot ground any additional task compared to `gringo`. `lpopt` does not help `gringo` because of the *action predicates*. These are predicates representing possible instantiation of actions – such as predicate

[†]This is an abridged version of a paper published at ICAPS 2023 (Corrêa et al. 2023).

A in the previous example – and they have the maximum arity, containing all parameters in the body. Hence, `lpopt` cannot find useful decompositions of the rules.

One wonders what happens if we simply *remove the action predicates*. In our example above, we then only have a single rule $q(x, y) \leftarrow p(x), p(y)$. This makes the programs much easier to ground. In fact, when action predicates are removed, `gringo+lpopt` can ground all 862 tasks in the benchmark set. Unfortunately, we do not know which ground actions are relaxed-reachable anymore, we only know the relaxed-reachable atoms. This is not enough information for most planners in the literature.

To reconstruct the ground actions, we first compute the relaxed-reachable atoms using `gringo+lpopt` as in the previous paragraph. Then we use this information to compute the actions in a second step. This second step transforms each action A into a logic program, and iteratively finds stable models for this program, such that each stable model corresponds to a relaxed-reachable instantiation of A. This method is called *grounding via iterated solving*.

Given the rule r

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

we create a logic program $\mathcal{L}$ as follows. For every variable V in r, we introduce a fresh predicate V-assign and the following *choice rule*:

$$1\,\{V\text{-assign}(X) : q_k(X)\}\,1.$$

where X is a new variable and q_k is a unary predicate such that $q_k(V)$ is in the body of r.[1] This rule forces the stable model to pick exactly one constant for each variable and thus form a variable assignment.

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

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

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 $\mathcal{L}$.

We implemented this grounding via iterated solving and it outperforms `gringo`. While `gringo` grounds 752 tasks, our iterated solving method `iterated` grounds 798. If we add more information to `iterated`, such as inequalities, it can ground up to 808 tasks. This is more than any other method that grounds actions. However, `iterated` consumes much more time than `gringo` even for simple tasks. Considering that planners need to still find a plan after grounding it, this can be harmful.

One could assume that giving more time or memory to `iterated` would lead us to ground all tasks in our set, since they can be ground when actions predicates are removed. Not so. We show, via model-counting, that for 6 tasks we still have chances of grounding, but for the remaining tasks this is out of reach. In these tasks, the number of ground actions is at least 10^{13}. The amount of storage to represent these tasks is prohibitive, so improving our grounder would not solve the problem. Hence, we believe that `iterated` is close to the practical limit for this set.

[1] This predicate is guaranteed to exist due to the normalization of planning tasks. See the original paper for more details.

Conclusion

Overall, we studied how to improve grounding algorithms for planning. We compared standard algorithms in the planning literature to off-the-shelf grounders. These off-the-shelf grounders can already improve the performance of planners. We also introduced a new method, called grounding via iterated solving that grounds more tasks but is slower. This new method can be applied to ground other logic programs that are not related to planning. Our approach is designed for positive programs with large predicate arities in the rule heads of programs. In particular, it can help in cases where the density of structures is such that they cannot be sufficiently exploited any more.

References

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.; 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. Grounding planning tasks using tree decompositions and iterated solving. In *Proc. ICAPS 2023*.

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

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.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*, volume 13 of *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool.

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.

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.

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.