PLANNING WITH DIFFERENT REPRESENTATIONS

Inauguraldissertation
zur
Erlangung der Würde eines Doktors der Philosophie
vorgelegt der
Philosophisch-Naturwissenschaftlichen Fakultät
der Universität Basel

von

AUGUSTO BLAAS CORREA

Basel, 2024

Genehmigt von der Philosophisch-Naturwissenschaftlichen Fakultät
auf Antrag von

Erstbetreuer: Prof. Dr. Malte Helmert, Universität Basel

Zweitbetreuer: Prof. Dr. Ivan Dokmanić, Universität Basel

Externer Experte: Prof. Dr. Torsten Schaub, Universität Potsdam

Basel, den 17.09.2024

Prof. Dr. Marcel Mayor
Universität Basel, Dekan

To my father.

So in the future, the sister of the past,
I may see myself as I sit here now
but by reflection from that which then
I shall be.

— James Joyce, *Ulysses*

ABSTRACT

Classical planning tasks are represented using a logical language. It is common to use a first-order representation, as this makes the description of the tasks more compact. Planners have long relied on translating these first-order representations into propositional ones in a process called grounding. Earlier work showed that grounding can improve the performance of classical planners. But this can also lead to exponentially larger encodings, and tasks with short plans become intractable simply because of the size of their representation.

In this thesis, we focus on lifted planning, which operates directly on the first-order representation. This bypasses the need for grounding, avoiding its computational overhead. We show how to build a lifted planner, called Powerlifted, that uses different techniques from database theory and logic programming to achieve state-of-the-art performance. First, we show how to perform a state-space search over the lifted representation. We address the lifted successor generation problem and show that this is equivalent to solving conjunctive queries. By exploiting the acyclicity of conjunctive queries, Powerlifted can generate successors efficiently in many domains. Second, we show how to use Datalog to compute delete-relaxation heuristics, such as the additive and the FF heuristics, directly over the lifted representation. Finally, we show how to translate other state-of-the-art techniques, such as preferred operators and width search, from the ground to the lifted setting.

We then show that some ideas from our work on lifted planning can be carried over to the ground setting. Using techniques implemented in Powerlifted, we show how to improve the grounding algorithms commonly used by classical planners. This is significant, as grounding is a crucial phase of most classical planners developed over the last two decades. Our new algorithm uses the grounding via solving paradigm from answer set programming, in which the grounding of a problem is decoupled into several logic programs, which must be solved individually to produce the propositional representation. In our experiments, our method outperforms other commonly used grounders from the literature.

Finally, we propose an extension to the classical planning formalism allowing for object creation within action effects. In practice, we show that Powerlifted can support this fragment almost effortlessly and that lifted planners, in general, seem to be a good fit for planning with object creation. Our experimental results show that object creation does not add overhead to Powerlifted while allowing for more expressive planning formalisms.

ACKNOWLEDGMENTS

It is a mystery to me how people manage to write acknowledgments in fewer than 100 pages. My impulse is to acknowledge every single living being that my eyes have met since the start of my Ph.D. By the eighth paragraph, I could picture myself citing my neighbor's parrot. But it turns out that society becomes judgmental when you thank your former plumber or the bus driver who helped bring you home from Ikea on that Thursday evening. So we will keep it short.

I contacted Malte for the first time in 2016. I had just received a scholarship to study in Munich, and I asked Malte if I could do some sort of research visit while I was in Germany. Malte replied that I was welcome to visit the group for a month. One year later, the visit took place, and, suddenly, the only thing that mattered to me was to find a way to come back to Basel, to do my Master's and Ph.D. with the AI group. It was not easy (the number of cyclic dependencies in any immigration process tends to infinity) but Malte – together with Heike Freiberger and Yvonne Walser – solved all bureaucratic matters and allowed me to come here. I've never seen an advisor put so much effort into helping a student join his group, particularly an undergrad student. But Malte did that. After I joined the group, he continued to support me, but now also scientifically. Malte never spared any effort to answer my questions or to fuel my overambitious ideas. And for all of this, I will always be grateful. Malte: I am not sure if you are aware, but you changed my life. Thank you!

I also want to thank Torsten Schaub for agreeing to join my thesis committee as the external examiner, despite my complicated time constraints and deadlines. Moreover, in 2022, Torsten (together with Roland Kaminski, Javier Romero, and Klaus Strauch) suggested us to contact Stefan Woltran on how to use tree decompositions to ground planning tasks. The rest of the story is told in Chapter 6.

There are three people who worked with me in Basel who deserve more credit for this thesis than meets the eye: Florian Pommerening, Guillem Francès, and Jendrik Seipp. When I first visited the AI group, randomness had me working with Florian. This is the closest I've ever been to winning the lottery. Florian taught me to write papers, to read papers, to run experiments, to rerun experiments, and all the rest. Guillem only joined the group later, when I was also starting as a Master's student. I don't know why, but once Guillem asked if I wanted to read a paper on description logic and discuss it with him. At that time, I didn't know how to say no, so I accepted the offer. Fast-forward almost 7 years, and that first discussion sparked my interest in knowledge representation for planning, which turned

into this thesis. Guillem is, to this day, one of my best friends. With Jendrik, on the other hand, it took a while until I got to know him. Jendrik always seemed more sophisticated and cooler than the rest, so I was scared he would find me underwhelming. But Jendrik is actually one of the finest lads out there, always smiling and telling good jokes. As we started to work together, Jendrik quickly became one of my favorite co-authors. Every time I talk to him, I feel as if I learned twenty new things about programming, grammar, music, or whatever. To all three of you: thank you very much!

Everyone in the AI group was always welcoming to me. When I started as a Hiwi, I thought that everyone in the group was smarter than me. Now, years later, I continue thinking that. Many thanks to my colleagues Clemens Büchner, Remo Christen, Simon Dold, Salomé Eriksson, Claudia Grundke, Florian Pommerening, Gabriele Röger, Tanja Schindler, and David Speck, and to my former colleagues Liat Cohen, Patrick Ferber, Guillem Francès, Cedric Geissmann, Manuel Heusner, Thomas Keller, Jendrik Seipp, and Silvan Sievers.

I was also fortunate to visit two great groups during my Ph.D. and to make friends along the way. My first stop was in Vienna, where I worked with Davide Mario Longo and Markus Hecher. Our work turned into a paper, which then turned into a planner that won the IPC (also with Guillem and Jendrik). Davide and Markus were also crucial in this thesis, answering my silly questions about logic programming and databases. Markus and Davide (and Sandra), I have no words to express my gratitude for your help.

My second stop was in Oxford, to work with Giuseppe De Giacomo. I was lucky to be welcomed by Antonio Di Stasio and Shufang Zhu, who helped me love and hate Oxford. There, I also had the chance to meet Sasha Rubin, who became a dear friend of mine. My time in Oxford contains some of the best months of my Ph.D., and I remember it fondly. Giuseppe, Antonio, Shufang, and Sasha, thank you for everything.

While I was writing this thesis, I was often impressed by the willingness and commitment of busy people to spend their time proofreading my writing. I want to thank all those who wasted brain cells deciphering earlier portions of this work: Clemens Büchner, Remo Christen, Guillem Francès, Markus Hecher, Davide Mario Longo, André G. Pereira, Jendrik Seipp, David Speck, and Sasha Rubin. In particular, a special thanks to Florian Pommerening, who proofread many chapters of this thesis, and to Przemysław Wałęga who took his time to suggest me some seminal papers in database theory.

Basel is not the most exciting city in the world, but I was blessed to find many friends here who made it lively. A huge thanks to Selaudin Agolli, Nadine Engeler, Mason Minot, Jonas H. Müller Korndörfer, Marjorie Pacheco Moraes, Marcelo Pereira, Agni Ramadani, Alex Rovner, Upnishad Sharma, and many others. And, around Eu-

rope, I was equally successful with my endeavor of finding good company. Many thanks to Vinícius Allegrini, Thiago Bell, João Bittar, Thomas Brunner, Michael Bernreiter, Philipp Foth, Camila Gehling, Elma Kablarevic, Sanja Lukumbuzya, Anna Rapberger, and Carmel Saig.

Across the Atlantic, there are also dozens of people who influenced my journey to get here. Above all, I want to express my lasting gratitude to Marcus Ritt and André G. Pereira: you are by far the two most influential people in my life, and I see your fingerprints everywhere I look. I also want to extend this acknowledgment to other three other Brazilians who embraced me during these years: Alex Gliesch, Felipe Meneguzzi, and Tadeu Zubaran.

Last, but not least, I would like to thank my father, César, and my stepmother, Kelly, for their endless support, and for making me a fan of Grêmio.

To all of you: thank you!

CONTENTS

1

INTRODUCTION

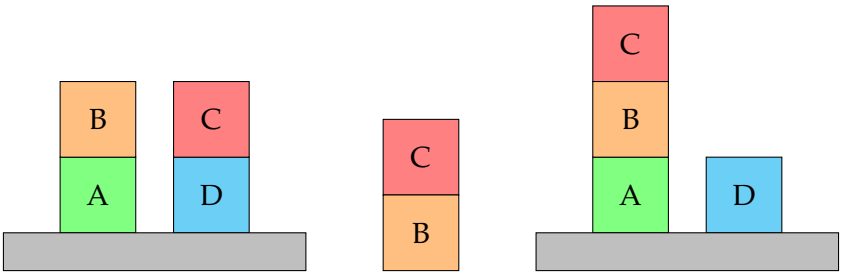
Imagine you want to travel somewhere for your summer holidays. There is a lot to prepare before you can enjoy your time off. First, where should you go? Somewhere close and cheap but not so exciting, or somewhere more expensive that would be much more memorable? Should you travel by plane or train? The plane is faster but airports are time-consuming as well. And what should you pack? Depending on the forecast, you can leave the raincoat at home and save some space in your hand-luggage for a few more shirts, shorts, underwear, and shoes. You definitely want to bring some ties, as ties are needed at all times (Wodehouse, 1930). You also want to bring some books, toiletries, and other smaller things. But do you have space for all that? Should you bring a larger suitcase?

The number of choices in this simple scenario is enough to give us a headache. Luckily, we can automate these decisions using *automated planning*. In short, automated planning is the problem of finding a sequence of actions — what to pack or how to travel — that achieves a desired goal — reach your holiday destination on time, within budget, and without forgetting anything essential.

An automated planning *task* is usually defined as follows: given an initial state of your world, a goal, and actions that modify the world, find a sequence of actions that achieves the goal. This definition is generic enough to represent different families of problems, e.g., Rubik's cube, transportation problems, or simulation of chemical reactions.

An advantage of automated planning is that using a specific *representation* for all these problems allows us to implement *domain-independent planners*. This means that a planner (i.e., an algorithm designed to solve planning tasks) is not focused on solving a single problem. It can solve any problem that can be encoded in a *formalism* that is accepted as input by the planner. These formalisms are usually defined using *logic*.

For example, consider the *Blocks World* domain, where blocks are arbitrarily stacked up on a table, forming different towers. The goal is to rearrange the blocks in some specific configuration by moving one at a time. Figure 1.1 displays the initial state (1.1a), the goal condition (1.1b), and a state satisfying the goal — a *goal state* (1.1c). The only



(a) Initial state. (b) Goal of the task. (c) Goal state.

Figure 1.1: Examples of initial state, goal, and a goal state for a Blocksworld task with four blocks.

action in this domain is to pick a block on top of a tower and place it on top of another tower or on the table.

The missing piece is how to represent the actions that modify our state. The straightforward approach is to enumerate every single possible action:

move A from the table to B
move A from the table to C
move A from the table to D
move A from B to C
move A from B to D
move A from B to the table
move A from C to B
move A from C to D
move A from C to the table
move A from D to B
move A from D to C
move A from D to the table
move B from the table to A
⋮
move D from C to B
move D from C to the table.

There are 48 actions in total. Clearly, most of these actions are not *applicable* to our initial state. For example, we cannot move A from the table to the top of D: we need to remove B from the top of A first. On the flip side, we can achieve our goal with a *plan* containing a single action: move C from the top of D to the top of B. Only one of the 48 actions is necessary to solve this task.

For a task with n blocks, we have $O(n^3)$ actions. If $n = 1\,000$, then we have approximately $1\,000\,000\,000$ actions. It is clear that the total number of actions becomes huge as n increases. Moreover, there is always a plan using only $2n$ actions: move every block to the table (this takes at most n actions) and re-stack them according to the goal (at most n actions). In our case with $1\,000$ blocks, this means that the shortest solution would always use at most 0.000001% of the given actions. Just storing the actions could be more costly than solving the problem itself.

If we look more closely at our problem, however, we can see that all actions above follow the same *schema*: pick a block X — which can be any of the four blocks — from the top of another block Y or the table and place it either on the table or on top of a third block Z. This allows us to represent action at a higher level, simply based on their schemas:

move X from Y to Z.

The "blocks" X, Y, and Z are called *variables* and work as placeholders for the concrete objects A, B, C, D and the table.

With this new action representation, it is no longer enough to simply find a sequence of actions. The planner now must also infer *which blocks to use at each step of the plan*. For example, our plan above (moving C from D to the top of B) can be found using the following mapping:

X → C, Y → D, Z → B

The second representation — called a *lifted representation* — is much more compact than the first one — called a *propositional* or *ground* representation. But there is no free lunch: while this compactness makes the problem description easier, it makes the job of the planner harder. The additional effort of inferring which objects to use at each step can become a new bottleneck for the planner. Pragmatically speaking, different representations always lead to different advantages and disadvantages.

In planning, problems are usually defined using a lifted representation. Most planners then convert it into a propositional representation by *grounding* the task. In larger problems, though, grounding becomes very expensive. This limits the reach of planners, and consequently the applicability of planning to larger problems.

In this thesis, we study how to plan with different representations. In particular, all the representations we use can be characterized *logically*. Our first contribution is to show how to implement an efficient planner that works on the lifted representation directly. We focus particularly on how to implement lifted *heuristic search* algorithms. By using techniques from database theory, knowledge representation, and other areas, we build an efficient and competitive *lifted planner*. Our new lifted planner, called *Powerlifted*, is the first lifted planner

to be competitive with *ground planners* — those grounding the tasks beforehand. Powerlifted includes different state-of-the-art *heuristics* (e.g., Bonet and Geffner, 2001) and specialized methods for *successor generation* based on structural decompositions (Yannakakis, 1981).

Playing devil's advocate, we also show that many of the techniques used in Powerlifted can be used to ground planning tasks. Based on one of the most popular grounders for planning (Helmert, 2009), we implement a new grounder that uses logic programming. We show how to optimize this grounder by exploiting different techniques, such as rule decomposition (Bichler et al., 2016; Bliem et al., 2020) and grounding via solving (Besin et al., 2022).

Finally, we study an important extension to the traditional planning formalism: object creation. In classical planning problems, objects are defined at the beginning, and the set of objects is immutable. In this extension, actions can *create new objects*. This brings us to a much more powerful fragment of planning. We show that Powerlifted can be easily adapted to work on this new flavor of planning. Planning with object creation presents several challenges but it also offers new insights. This new area is promising, as it expands the scope of problems that planning can address.

1.1 CONTRIBUTIONS & STRUCTURE

This structure of this thesis is as follows:

- Chapter 2 provides the background knowledge necessary for the thesis. It formalizes the notion of planning tasks and heuristic search, and introduces necessary concepts from logic programming. At the end, we provide a brief summary of earlier planning techniques.

- Part i studies how to implement an efficient lifted planner. Our first topic is how to *generate successor states* during search. We show that there is a direct connection between this problem and the problem of conjunctive query answering (Chandra and Merlin, 1977; Codd, 1970) in database theory. With this connection, we implement a lifted successor generator exploiting structural decompositions of conjunctive queries (Gottlob et al., 2002; Yannakakis, 1981).

  We also show how to compute delete-relaxation heuristics (Bonet and Geffner, 2001) over lifted representations. This time, we exploit the connection between delete-relaxation heuristics and logic programming. Using Datalog, we implement efficient lifted versions of h^{add}, $h^{\max}$, and h^{FF} (Bonet and Geffner, 2001; Hoffmann and Nebel, 2001). Last, we study how to extend other techniques from ground planning — such as width-search (Lipovetzky and Geffner, 2012) — to the lifted setting.

- In Part ii, we are interested in bringing the conclusions from Part i to the context of *grounding*. More specifically, we study the grounding algorithms for planning that rely on logic programming (Helmert, 2009). We combine the structural decomposition techniques from the lifted successor generation together with the techniques used to compute lifted delete-relaxation heuristics, and use them to ground the planning tasks (Bichler et al., 2016). Our new grounder uses the grounding via solving (Besin et al., 2022) approach from answer set programming: in the first step, we overapproximate a set of reachable atoms (but not the actions); then, we use this set to finally extract the set of ground actions. This two-step approach slows down the grounder but has the merit of consuming less memory. Overall, our new grounder can ground more tasks than previous implementations from the literature.

- Part iii discusses *object creation*. We extend our basic classical planning formalism to handle problems where new objects are introduced as an effect of actions. Our extension builds on top of the lifted planning formalism, and fits well with the algorithms developed in Part i. We explain how to extend the basic version of Powerlifted to deal with object creation efficiently.

  From a theoretical perspective, we prove that this fragment is semi-decidable. If a plan exists for such a task, Powerlifted is guaranteed to find it. From the practical side, we study some design choices to improve Powerlifted in the context of object creation. In more detail, we explain possible adaptations of width search for this extension, and present potential improvements and research directions.

- Part iv summarizes the main contributions of this thesis, poses some unanswered questions, and proposes a few ideas of future work.

1.2 EXPERIMENTAL SETUP

We use the same experimental setup throughout this thesis. All experiments were run on an Intel Xeon Silver 4114 processor running at 2.2 GHz with maximum runtime of 30 minutes and a maximum memory of 16 GiB. Furthermore, we always use the Downward Lab toolkit (Seipp et al., 2017) in our tests. The source code and scripts used are publicly available online (Corrêa, 2024).

Planners always receive a PDDL (*Planning Domain Definition Language*) as input (Haslum et al., 2019; McDermott et al., 1998).[1] We benchmark our techniques on two sets of problems. The first set, called

1 However, at later parts, the PDDL language will be extended.

the *IPC set*, contains 1001 STRIPS tasks (properly defined in Chapter 2) from 29 different domains used in the first nine editions of the IPC. The second set, called the *HTG set*, contains 862 hard-to-ground (HTG) tasks over 8 different domains. This set contains problems that cannot be easily converted into propositional tasks (i.e., ground), and so give more insight into the capabilities of lifted planners and grounding algorithms.

The HTG set is a merge of two different sets used in the literature (Corrêa et al., 2020; Lauer et al., 2021), and has been used as the *de facto* standard to evaluate lifted planners (Horčík and Fišer, 2021; Ståhlberg, 2023). When reporting experimental results, we usually focus on this set.

Our general metrics of performance are time, memory, and coverage (total number of solved tasks). These are our main criteria when comparing different implementations (e.g., two different planners). In more specific scenarios – such as when we compare two search algorithms implemented in the same planner – we rely on more fine-grained metrics. For heuristic search algorithms in particular, we often compare the number of expanded states and plan length. The first gives an idea of how informed the search is; the second estimates the quality of the solution.

1.3 PUBLICATIONS

Most of the content in this thesis has been previously published in proceedings of academic conferences. The core ideas can be found in the following seven papers:

(i) Augusto B. Corrêa, Florian Pommerening, Malte Helmert, and Guillem Francès (2020). "Lifted Successor Generation using Query Optimization Techniques." In Proc. ICAPS 2020. AAAI Press, pp. 80–89.

(ii) Augusto B. Corrêa, Guillem Francès, Florian Pommerening, and Malte Helmert (2021). "Delete-Relaxation Heuristics for Lifted Classical Planning." In Proc. ICAPS 2021. AAAI Press, pp. 94–102.

(iii) Augusto B. Corrêa, Florian Pommerening, Malte Helmert, and Guillem Francès (2022). "The FF Heuristic for Lifted Classical Planning." In Proc. AAAI 2022. AAAI Press, pp. 9716–9723.

(iv) Augusto B. Corrêa and Jendrik Seipp (2022). "Best-First Width Search for Lifted Classical Planning." In Proc. ICAPS 2022. AAAI Press, pp. 11–15.

(v) Augusto B. Corrêa, Markus Hecher, Malte Helmert, Davide Mario Longo, Florian Pommerening, and Stefan Woltran (2023).

"Grounding Planning Tasks Using Tree Decompositions and Iterated Solving." In Proc. ICAPS 2023. AAAI Press, pp. 100–108. **Runner Up, Best Student Paper Award at ICAPS 2023**.

(vi) Augusto B. Corrêa, Giuseppe De Giacomo, Malte Helmert, and Sasha Rubin (2024). "Planning with Object Creation." In Proc. ICAPS 2024. AAAI Press, pp. 104–113.

(vii) Augusto B. Corrêa, and Giuseppe De Giacomo (2024). "Lifted Planning: Recent Advances in Planning Using First-Order Representations." In Proc. IJCAI 2024, to appear.

Paper (i) introduces Powerlifted, the first competitive lifted planner, and shows how to generate successor states in a lifted state-space search (Chapter 3). Papers (ii) and (iii) show how to compute delete-relaxation heuristics over the lifted representation (Chapter 4). Both of them use Datalog programs to do so. Paper (iv) studies how to efficiently implement width search (Lipovetzky and Geffner, 2012; Lipovetzky and Geffner, 2017) and other algorithms (Röger and Helmert, 2010) in the lifted setting (Chapter 5). Paper (v) goes in the opposite direction: it shows how to use the insights in lifted planning to optimize the grounding of planning tasks (Chapter 6). Paper (vi) introduces an extension of PDDL where new objects can be constructed as effect of an action (Chapter 7). Finally, paper (vii) surveys the main lifted planning techniques introduced in the last ten years (content spread through several chapters).

The techniques presented in these papers were combined into three different planners that participated in the IPC 2023:

- Augusto B. Corrêa, Guillem Francès, Markus Hecher, Davide Mario Longo and Jendrik Seipp (2023). "The Powerlifted Planning System in the IPC 2023." In Tenth International Planning Competition (IPC 2023), Deterministic Part.

- Augusto B. Corrêa, Guillem Francès, Markus Hecher, Davide Mario Longo and Jendrik Seipp (2023). "Scorpion Maidu: Width Search in the Scorpion Planning System." In Tenth International Planning Competition (IPC 2023), Deterministic Part. **Winner, Deterministic Sequential Satisficing Track**.

- Augusto B. Corrêa, Guillem Francès, Markus Hecher, Davide Mario Longo and Jendrik Seipp (2023). "Levitron: Combining Ground and Lifted Planning." In Tenth International Planning Competition (IPC 2023), Deterministic Part. **Winner, Deterministic Sequential Satisficing Track**.

Scorpion Maidu and Levitron were the two co-joint winners of the satisficing track of the IPC 2023. Scorpion Maidu is a ground planner built on top of Scorpion (Seipp, 2018) but adds large collection of width

search algorithms, based on the ideas of paper (iv) above. Levitron is a hybrid planner that uses Powerlifted on tasks that it cannot ground within the available resources, and uses Scorpion Maidu otherwise.

Apart from the aforementioned papers and planners, the author contributed to the following works during his doctoral studies:

- Rik de Graaff, Augusto B. Corrêa and Florian Pommerening (2021). "Concept Languages as Expert Input for Generalized Planning: Preliminary Results." In ICAPS 2021 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS 2021).
- Malte Helmert, Silvan Sievers, Alexander Rovner and Augusto B. Corrêa (2022). "On the Complexity of Heuristic Synthesis for Satisficing Classical Planning: Potential Heuristics and Beyond". In Proc. ICAPS 2022. AAAI Press, pp. 124–133.
- Mohammad Abdulaziz, Florian Pommerening and Augusto B. Corrêa (2022). "Mechanically Proving Guarantees of Generalized Heuristics: First Results and Ongoing Work." In Proc. GenPlan 2022.
- Lucas Galery Käser, Clemens Büchner, Augusto B. Corrêa, Florian Pommerening and Gabriele Röger (2022). "Machetli: Simplifying Input Files for Debugging." In System Demonstrations at ICAPS 2022.
  Best System Demonstration Award at ICAPS 2022.
- Augusto B. Corrêa, Clemens Büchner and Remo Christen (2023). "Zero-Knowledge Proofs for Classical Planning Problems." In Proc. AAAI 2023. AAAI press, pp. 11955–11962.
- Clemens Büchner, Remo Christen, Augusto B. Corrêa, Salomé Eriksson, Patrick Ferber, Jendrik Seipp and Silvan Sievers (2023). "Fast Downward Stone Soup 2023." In Tenth International Planning Competition (IPC 2023), Deterministic Part.
  Runner-up, Deterministic Sequential Satisficing and Agile Tracks.
- Daniel Doebber, André Grahl Pereira and Augusto B. Corrêa (2023). "OpCount4Sat: Operator Counting Heuristics for Satisficing Planning." In Tenth International Planning Competition (IPC 2023), Deterministic Part.
- Augusto B. Corrêa and Jendrik Seipp (2024). "Consolidating LAMA with Best-First Width Search." In ICAPS 2024 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP 2024).

These works do not fit the main arc of the thesis, and will not be discussed in detail here.

2

BACKGROUND

In this chapter, we formalize the basic concepts used throughout the thesis. Most of the content is related to logic programming and classical planning.

We assume basic knowledge about logic and computational complexity. In particular, we assume familiarity the basic concepts of first-order logic, although we briefly discuss some of them below to introduce notation and refresh some parts. For a thorough discussion on logic, we refer to the textbook by Ebbinghaus et al. (1994).

Appendix A includes a brief overview of the computational complexity concepts needed for this thesis. If the terms **P**, **NP**, **PSPACE**, **EXPTIME**, and **EXPSPACE** are familiar to the reader, Appendix A can be safely skipped.

2.1 CONVENTIONS

Throughout this thesis, we assume that logical languages (i.e., conjunctive queries, logic programs, and planning tasks) are defined using a *function-free logical vocabulary* L over an infinite set $\mathcal{V}$ of *variables*, a finite set of $\mathcal{C}$ *constants*, and a set $\mathcal{P}$ of *predicate symbols*.

variables
constants
predicate symbols

To denote variables, we use capital letters $(X, Y, Z, \dots)$ possibly indexed $(X_1, Y_3, Z_{42}, \dots)$. Constants are denoted by lowercase upright strings $(\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{truck}, \dots)$. Predicate symbols are strings written also in lowercase, but are italicized $(p, q, \mathit{move}, \dots)$. A *term* is either a variable or a constant. We use boldface symbols $(\boldsymbol{T}, \boldsymbol{X}_3, \dots)$ to denote tuples of terms.

term

An *atom* $p(\boldsymbol{T})$ is composed of a predicate symbol $p \in \mathcal{P}$ and a k-tuple of terms $\boldsymbol{T} = \langle T_1, \dots, T_{ar(p)} \rangle$, where $ar(p)$ is the arity of p; for clarity, we sometimes write p/k to define a predicate symbol p with $ar(p) = k$. The set of variables in $\boldsymbol{T}$ is denoted as $\mathit{vars}(\boldsymbol{T})$. With some abuse of notation, we use set-theoretical symbols with terms, although they are ordered sequences.

atom

An atom $p(\boldsymbol{T})$ is called a *ground atom* iff $\mathit{vars}(\boldsymbol{T}) = \emptyset$.

ground atom

Given a (non-ground) atom $p(\boldsymbol{T})$, we can *ground* this atom by replacing all its variables $\mathit{vars}(\boldsymbol{T})$ with constants in $\mathcal{C}$. Formally, a *substitution function* σ (partial or total) maps variables in $\mathcal{V}$ to constants in $\mathcal{C}$. With some abuse of notation, we extend σ to atoms: we write $\sigma(p(X, Y))$

substitution function

to indicate $p(\sigma(X), \sigma(Y))$. We further extend it to sets of atoms, e.g., $\sigma(\{p(X), p(Y)\}) := \{p(\sigma(X)), p(\sigma(Y))\}$. The grounding of an atom (or set of atoms) is also called an *instantiation*.

instantiation

2.2 LOGIC PROGRAMMING

logic program

A *logic program* is a set of *rules* of the form

rules

$$h_1 \vee \ldots \vee h_k \leftarrow p_1, \ldots, p_j, \neg n_1, \ldots, \neg n_m.$$

head
body

where h_i, p_i and n_i are all atoms. Given a rule r, its *head* is the set denoted by $head(r) = \{h_1, \ldots, h_k\}$; the *body* of r is the set denoted by $body(r) = \{p_1, \ldots, p_j, n_1, \ldots, n_m\}$. Furthermore, the set of positive body atoms is denoted by $body^+(r) = \{p_1, \ldots, p_j\}$, while the set of negative body atoms is denoted by $body^-(r) = \{n_1, \ldots, n_m\}$. The set of variables appearing in r is denoted $vars(r)$. If $vars(r) = \emptyset$, then r is a *ground rule*.

fact

If $body(r) = \emptyset$ and $head(r)$ is a ground atom, then r is called a *fact*. It is common to explicitly distinguish facts from other rules when defining logic programs: for convenience, we denote logic programs as a pair $\mathcal{L} = \langle \mathcal{F}, \mathcal{R} \rangle$ where $\mathcal{F}$ is a finite set of facts, and $\mathcal{R}$ is a finite set of (non-fact) rules. When writing down logic programs, we typically write them as a list. The set of facts $\mathcal{F}$ is always given first, and we omit the symbol $\leftarrow$ when enumerating them.

Example 2.1 *Consider the following logic program* $\mathcal{L}$

$$\begin{aligned} &pet(\text{bob}).\\ &pet(\text{jack}).\\ &fish(\text{jack}).\\ &dog(X) \vee cat(X) \leftarrow pet(X), \neg fish(X). \end{aligned}$$

with three facts and a single rule, which we denote by r. The head of r is $\{dog(X), cat(X)\}$*, the body of r is* $\{pet(X), \neg fish(X)\}$*,* $body^+(r) = \{dog(X)\}$*, and* $body^-(r) = \{fish(X)\}$*.*

This program declaratively represents that, for any X, if X is a pet but not a fish, then X is either a dog or a cat.

extensional predicate

A logic program is *ground* if all its rules are ground. A predicate symbol (from the underlying logical vocabulary) is *extensional* if it occurs in rule bodies but never in a rule head.[1] In the rest of the thesis, we only consider *safe rules*, where variables occurring in the head also occur in the body.

safe rule

Datalog

In most of this thesis, however, it is sufficient to consider a simpler fragment of logic programs: *Datalog* programs. A Datalog program

1 In the planning literature, extensional predicates are sometimes called "static".

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

$$h \leftarrow p_1, \ldots, p_j.$$

i.e., have no negative atoms in the body and exactly one atom in the head. In Part ii, we use more powerful logic programs, but we stick with Datalog here to avoid overcomplicating the definitions now.

Example 2.2 *Consider the following Datalog program $\mathcal{D}$:*

$$\begin{aligned}
&\mathit{parent}(\text{alice}, \text{charlie}).\\
&\mathit{parent}(\text{charlie}, \text{dave}).\\
&\mathit{parent}(\text{charlie}, \text{eve}).\\
&\mathit{ancestor}(X, Y) \leftarrow \mathit{parent}(X, Y).\\
&\mathit{ancestor}(X, Z) \leftarrow \mathit{parent}(X, Y), \mathit{ancestor}(Y, Z).
\end{aligned}$$

It encodes the ancestry relation based on parenthood. Alice is a parent of Charlie, and Charlie is a parent of both Dave and Eve. Hence, both Alice and Charlie are ancestors of Dave and Eve.

Herbrand base

interpretation

The *Herbrand base* $\mathcal{H}(\mathcal{D})$ of a Datalog program $\mathcal{D}$ is the set of all ground atoms obtained by instantiating predicates from $\mathcal{P}$ with constants in $\mathcal{C}$. An *interpretation* $\mathcal{I} \subseteq \mathcal{H}(\mathcal{D})$ is a subset of atoms. We say that a ground atom a is true under $\mathcal{I}$ if $a \in \mathcal{I}$, and it is false otherwise.

Let $\mathit{Ground}(r)$ be the set of all groundings of a rule $r \in \mathcal{R}$ (i.e., all possible substitutions of variables in r with constants in $\mathcal{C}$), and let $\mathit{Ground}(\mathcal{D})$ be the grounding of all rules of $\mathcal{D}$. An interpretation $\mathcal{I}$ *satisfies* the ground rule r in $\mathit{Ground}(\mathcal{D})$ if $\mathit{body}(r)$ is true under $\mathcal{I}$. An interpretation is a (Herbrand) *model* iff, for every $r \in \mathit{Ground}(\mathcal{D})$ that is satisfied by $\mathcal{I}$, the head $\mathit{head}(r)$ is also true under $\mathcal{I}$.

model

canonical model

seminaive evaluation

Each Datalog program has a unique *minimal* model $\mathcal{M}$ (in number of atoms), sometimes referred to as the *canonical model*. This canonical model is usually computed using a *seminaive evaluation* (Abiteboul et al., 1995, Ch. 13). This is a bottom-up approach that iteratively derives new atoms from previously derived ones. First, it tracks in which iteration each atom was derived. In each iteration, the algorithm unifies rules with atoms derived from previous iterations to derive new ones. To avoid duplicates and guarantee termination, the seminaive evaluation algorithm enforces that at least one atom derived at iteration i must be used when unifying rules at 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).

Example 2.3 *The canonical model $\mathcal{M}$ of the Datalog program $\mathcal{D}$ from the previous example is*

$$\begin{aligned}\mathcal{M} \coloneqq \{&\mathit{parent}(\text{alice}, \text{charlie}),\ \mathit{parent}(\text{charlie}, \text{dave}),\\ &\mathit{parent}(\text{charlie}, \text{eve}),\ \mathit{ancestor}(\text{alice}, \text{charlie}),\\ &\mathit{ancestor}(\text{charlie}, \text{dave}),\ \mathit{ancestor}(\text{charlie}, \text{eve}),\\ &\mathit{ancestor}(\text{alice}, \text{dave}),\ \mathit{ancestor}(\text{alice}, \text{eve})\}\end{aligned}$$

For a given Datalog program $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$, deciding if an atom a is in the model $\mathcal{M}$ of $\mathcal{D}$ is **EXPTIME**-complete (Immerman, 1986; Vardi, 1982).[2]

2.3 CLASSICAL PLANNING

A *planning task* consists of the following: an initial state, a set of actions, and a goal, all formally specified using some logic vocabulary. The objective of a *planner* is to synthesize a strategy/program — called a *plan* — to fulfill the goal starting from the initial state. There are different models of planning: single- or multi-agent; fully-observable or partially observable states; deterministic, stochastic, or non-deterministic actions; etc. Each combination of these properties leads to a different variant, bringing us different difficulties and different algorithms.

In this work, we focus on the simplest flavor of planning, the *classical planning* model: actions are deterministic and discrete, states are fully-observable, and tasks only have one agent. To be less verbose, we refer to this flavor simply as *planning*.

STRIPS

We consider planning tasks in *STRIPS* (Fikes and Nilsson, 1971) with inequalities. A planning task is a 5-tuple $\Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, I, G \rangle$ where

- $\mathcal{P}$ is the set of *predicate symbols* from our logical vocabulary;
- $\mathcal{C}$ is the set of *constants* from our logical vocabulary;
- $\mathcal{A}$ is a set of *action schemas*;
- I is the *initial state*;
- G is the *goal*.

We assume that the binary predicate symbol $\neq$ is always in $\mathcal{P}$ and represents the inequality relation.

precondition
add list
delete list
action cost
effects

An action schema $a \in \mathcal{A}$ is a tuple $a = \langle \mathit{pre}(a), \mathit{add}(a), \mathit{del}(a), \mathit{cost}(a) \rangle$ where the *precondition* $\mathit{pre}(a)$, the *add list* $\mathit{add}(a)$, and the *delete list* $\mathit{del}(a)$ are sets of atoms, and $\mathit{cost}(a) \in \mathbb{N}$ is the *action cost*. We use $\mathit{vars}(a)$ for the set of variables occurring in any atom of the precondition, add list, and delete list. We sometimes refer to action *effects*, indicating the

2 We consider the case of *combined complexity*: facts and rules are part of the input (i.e., not fixed in advance). If we assume that the rules are fixed, then deciding if $a \in \mathcal{M}$ can be done in polynomial time (Immerman, 1986; Vardi, 1982).

add list and the delete list of an action together. If $vars(a) = \emptyset$, we call a a *ground action* and if all actions in a task are ground, we call the task a *ground task*. If the task is not ground, then it is a *lifted task*. In most chapter, action costs do not play a major role in our definitions and contributions. When this is case, we define an action a simply as a triple $a = \langle pre(a), add(a), del(a) \rangle$ to streamline notation. Whenever we do not explicitly describe the cost of an action a, the reader may assume that $cost(a) = 1$.

ground action
ground task
lifted task

Given an action schema $a \in \mathcal{A}$, we can ground a similarly as we ground atoms. Let σ be a substitution function mapping $vars(a)$ to $\mathcal{C}$. To produce a ground action, denoted as $\sigma(a)$, we can simply apply σ to the precondition, add list, and delete list of a. More formally, $\sigma(a) = \langle \sigma(pre(a)), \sigma(add(a)), \sigma(del(a)) \rangle$.

A *state* s is a set of ground atoms. A ground action a is *applicable* in s if $pre(a) \subseteq s$. 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* $succ(s, a) = (s \setminus del(a)) \cup add(a)$. A sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$ is applicable in a state s_0 and has $succ(s_0, \pi) = s_n$ if there are states $s_1, \ldots, s_{n-1}$ where a_i is applicable in s_{i-1} and $succ(s_{i-1}, a_i) = s_i$ for all $i \leq n$.

state
applicable action
successor state

The *initial state* I of a task is a state and the *goal* 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 $succ(I, \pi)$ is a goal state. Given a plan $\pi = \langle a_1, \ldots, a_n \rangle$, we define its *cost* as

initial state
goal
plan
cost

$$cost(\pi) = \sum_{i=1}^{n} cost(a_i).$$

Example 2.4 *Consider a simplified logistics task where a truck* t *starting at some city* a *must reach city* c*. City* a *is adjacent (i.e., connected) to city* b*, which in turn is adjacent to* c*. This problem can be encoded as a planning task* $\Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, I, G \rangle$ *as follows:*

$$\begin{aligned}
\mathcal{P} &= \{at/2, adj/2\} \\
\mathcal{C} &= \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{t}\} \\
\mathcal{A} &= \{drive(T, C_1, C_2)\} \\
I &= \{adj(\mathsf{a}, \mathsf{b}), adj(\mathsf{b}, \mathsf{a}), \\
&\quad\ adj(\mathsf{b}, \mathsf{c}), adj(\mathsf{c}, \mathsf{b}), \\
&\quad\ at(\mathsf{t}, \mathsf{a})\} \\
G &= \{at(\mathsf{t}, \mathsf{c})\}.
\end{aligned}$$

The action schema $drive(T, C_1, C_2)$ *has the following precondition, add list, and delete lists:*

$$\begin{aligned}
pre(drive(T, C_1, C_2)) &= \{at(T, C_1), adj(C_1, C_2)\} \\
add(drive(T, C_1, C_2)) &= \{at(T, C_2)\} \\
del(drive(T, C_1, C_2)) &= \{at(T, C_1)\}.
\end{aligned}$$

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 to compile type information away by introducing unary type predicates *type-T* for each type T (e.g., Helmert, 2009). 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.

Planning tasks are usually encoded using PDDL (Haslum et al., 2019; McDermott et al., 1998). It is the *de facto* standard in classical planning, and it is used in the *International Planning Competition* (IPC). PDDL problems use a first-order representation and our planning formalism above can easily be represented in PDDL. However, it is important to also note that PDDL supports much more expressive features as well. Throughout the thesis, we introduce planning problems formally as above, and we only mention PDDL when necessary; it is not required to know PDDL syntax to understand the rest of this text.

delete-free

An important fragment of classical planning is *delete-free classical planning*. A planning task Π is called *delete-free* if $del(a) = \emptyset$ for all action schemas a of Π. It is possible also to relax any planning task into a delete-free one, just by redefining its delete-lists. Given a planning task Π, we denote its *delete-relaxation* as Π^+, which is equivalent to Π but with all action schemas having empty delete lists. By ignoring the delete lists, applying an action in a state s only adds more ground atoms to the state. Thus, all actions that are applicable in s remain applicable in any successor state of s. The relaxation Π^+ is an overapproximation of Π: any atom *reachable* (i.e., achieved by any sequence of actions) in Π is also (relaxed) reachable in Π^+. The converse is not always true.

delete-relaxation

reachable

Complexity of Classical Planning

There are two decision problems commonly associated with classical planning. The first one, called PlanEx, asks if the task is *solvable*:

PlanEx
Input: A planning task Π.
Question: Is there a plan π for Π?

The second one, PlanLen, asks if a plan with a certain bounded length exists:

PlanLen
Input: A planning task Π, a number $k \in \mathbb{N}$.
Question: Is there a plan π for Π where $|\pi| \leq k$?

	Del.-Free	PLANEX	PLANLEN
Ground	✓	**P**	**NP**-comp.
	✗	**PSPACE**-comp	**PSPACE**-comp.
Lifted	✓	**EXPTIME**-comp.	**NEXPTIME**-comp.
	✗	**EXPSPACE**-comp.	**NEXPTIME**-comp.

Table 2.1: Complexity of different fragments of classical planning. Results for ground planning are due to Bylander (1994); results for lifted planning are due to Erol et al. (1995). Column "Del.-Free" indicates whether we assume tasks to be delete-free or not.

It is useful to analyze these problems based on how we represent the input task, as the complexity differs depending on whether Π is lifted or ground. If Π is a lifted task, the decision problems are usually exponentially harder to solve. Table 2.1 summarizes the results. We also consider the special case of delete-free tasks, as they will also be studied throughout this thesis.

Planning as Search

Since the early 2000s, *state space search* (Bonet and Geffner, 2001) is the ruling paradigm to solve planning tasks. A *state space* is a directed graph where the vertices are all possible states of a planning task Π, and there exists an edge $s_1 \xrightarrow{a} s_2$ between two states s_1 and s_2 iff there exists an action schema a and variable substitution σ for the variables in $vars(a)$ such that $\sigma(a)$ is applicable in s_1 and $s_2 \in succ(s_1, \sigma(a))$. A path from the initial state to any goal state is equivalent to a plan.

state space search

state space

To handle large state spaces, planners apply a *heuristic search* (Pearl, 1984) approach. A *heuristic function* h assigns to each state s the estimated distance from s to the closest goal state, denoted $h(s)$. When no goal state can be reached from s, the *heuristic value* $h(s)$ is infinite. A heuristic-search planner then uses this estimate, prioritizing states which are considered more promising.

heuristic function

A heuristic is *admissible* if it never overestimates the distance from s to the closest goal state. Admissible heuristics are important because they can guarantee optimal solutions when used with specific search algorithms, such as A^* (Hart et al., 1968). When combined with an admissible heuristic, A* (with state re-opening) guarantees an optimal solution, when a solution exists.

When any solution suffices, we can use *greedy best-first search* (GBFS), invented by Doran and Michie (1966). GBFS is usually faster than A* but does not guarantee optimality.

A* and GBFS work similarly: starting with a list, called an *open list*, containing only the initial state I, we iteratively select the most

open list

promising state s on the open list and expand s. Here, "to expand a state" means that we generate all successors of s and place them on the open list. State s is removed from the open list as well. We repeat this process until we select a goal state for expansion, indicating that we found a path in the state space from I to some goal state.

The only difference between A* and GBFS is in how we select the most promising state from the open list. GBFS orders the open list based on the heuristic value of each state, while A* additionally takes into account the length of the shortest known path from the initial state to the state, denoted as g. We do not go into further details about the behavior of A* and GBFS. We refer the reader to the classic book by Russell and Norvig (2020) for an introduction to search algorithms.

An important question in classical planning is how to come up with good heuristic functions. One common way is to use the cost of a *relaxed plan* (a plan for Π^+) as heuristic value (e.g., Hoffmann and Nebel 2001, Keyder and Geffner 2008). We denote the cost of an *optimal relaxed plan* as h^+. Note that h^+ is never higher than the cost of an optimal plan for the original (non-relaxed) task, so it can be used as an admissible heuristic. In general, delete-relaxation heuristics that approximate h^+ tend to compare favorably with other heuristics (Betz and Helmert, 2009; Hoffmann, 2005).

relaxed plan

h^+

CHAPTER NOTES AND HISTORY

Early work on planning and action theory often focused on first-order representations. In fact, most of the work dealt with logic vocabularies much more powerful than the one we consider here (e.g., with infinitely many objects). Planning on first-order representations is not something new and it was the immediate choice for decades. Below, we list some earlier "paradigms" of planning that used first-order representations.

Newell and Simon (1963) presented the General Problem Solver (GPS), which can be seen as a prototypical planning system. GPS could solve problems expressed in first-order formulas. It used means-ends analysis to perform a state-space search: given the current state and a goal, the planner performs an action that reduces the difference between the two.

Situation calculus (McCarthy, 1958, 1963) has also been used to study reasoning about actions, including the famous frame problem (McCarthy and Hayes, 1969). Perhaps the predominant version of situation calculus nowadays is the formalism by Reiter (2001), which has been applied to planning as well (De Giacomo et al., 2016; Levesque, 2005; Reiter, 2001). Moreover, previous work also studied the recasting of situation calculus as logic programming (Bibel, 1986; Kowalski, 1979).

Another early paradigm was planning via theorem proving (Green, 1969a,b). In this scenario, a planning problem is encoded in predicate logic and the goal is a first-order query. The answer to this query, obtained via resolution, corresponds to a plan. The QA3 system by Green (1969a) is probably the most well-known (historical) planner using theorem proving.

Fikes and Nilsson (1971) combined the insights from GPS and QA3. They introduced the STRIPS planner. STRIPS was not only a logical formalism but actually a full-fledged planning system. It allowed the user to describe an action theory in first-order using a specific syntax. At its core, the STRIPS system used techniques from GPS to control the search, and QA3 to unify action preconditions. However, the original STRIPS formalism was not bulletproof either — cf. Lifschitz (1987) gives a forceful critique of the semantics of STRIPS. In the decades following its original publication, the definition of the "STRIPS formalism" has changed (and it is rather ambiguous nowadays). Although still first-order, STRIPS is less expressive than situation calculus. In contrast to situation calculus, STRIPS requires a pre-defined finite set of objects.

Pednault (1989) tried to bridge the gap between STRIPS and situation calculus with the Action Description Language (ADL). Besides being more expressive than STRIPS (allowing quantified preconditions and effects, for instance), ADL also presented a solution to the frame problem. ADL was mainly focused on problems with finitely many objects.

McDermott (1996) introduced Unpop, a state-space search planner based on means-ends analysis. Unpop's overall idea is similar to delete-relaxation heuristics later used in the HSP planner (Bonet and Geffner, 2001). Moreover, McDermott's algorithm is similar to those used to compute lifted delete-relaxed heuristics, and that we present in Chapter 4.

Another important paradigm in the 1990s was refinement planning. A refinement planner performs a plan-space search: it gradually adds actions to a plan, trying to satisfy a series of goals, and backtracking to refine the plan when some constraint is violated (Weld, 1994). A large portion of the work in refinement planning focused on partial order planners. A partial order planner can place actions into a plan without specifying which comes first. In this setting, a plan is not a sequence of actions, but a partial-order. Successful implementations included SNLP (McAllester and Rosenblitt, 1991), UCPOP (Penberthy and Weld, 1992), and VHPOP (Younes and Simmons, 2003). Younes and Simmons (2002) investigated the impact of ground action in partial order planners, and showed that ground actions help in general, but that some benefits of the ground representation (e.g., enforcing constraints on the domains of variables) can also be exploited by the lifted representation. Their

work is an example of how successes from the ground representation can be translated to the first-order setting.

The Planning Domain Definition Language (PDDL) was introduced as the standard language during the first editions of the IPC (Haslum et al., 2019; McDermott, 2000; McDermott et al., 1998). PDDL was defined as a common encoding for the competing planners, and it remains so until today. It also uses a first-order representation with a finite set of objects. In the initial IPCs, most planners only supported a fragment of PDDL similar to STRIPS. Nowadays, most planners support fragments closer to (or more expressive than) ADL.

In the 1990s, Kautz and Selman (1992) showed that using propositional satisfiability (SAT) to solve planning problems was an effective technique. SAT planning translates the planning task into a SAT formula and uses a SAT solver to find a model. This encoding bounds the maximum length of a plan, so if no plan is found with the assumed length, the planner iteratively increases this bound and produces a new (and longer) formula, until a model is found. When this happens, the model is converted into a plan. SAT planning disregards any first-order structure of the problem, and instead works directly with the propositional representation. Although the supremacy of SAT planners did not last, most of the following works still used propositional representations. The dominant paradigm in planning in the last decades was state-space search (Bonet and Geffner, 2001; Francès et al., 2018; Helmert, 2006; Hoffmann and Nebel, 2001; Seipp, 2023; Torralba et al., 2014).

Part I

LIFTED PLANNING

3

LIFTED SUCCESSOR GENERATION

Classical planning relies on a model of the world encoded in some suitable representation language. One such language is PDDL (Haslum et al., 2019; McDermott et al., 1998), a first-order logic-based language developed to support the IPC and to standardize previous research efforts (Fikes and Nilsson, 1971; Pednault, 1989).

While there is a remarkable diversity of planning techniques, most planners nowadays ground the first-order representation of the problem into a propositional one as a preprocessing step. Although the ground representation can be exponentially larger in the number of variables of the action schemas, there are efficient grounding techniques (e.g., Helmert 2009) that make this preprocessing an effective strategy. The shortcoming, however, is that the entire set of ground actions needs to be stored in memory.

Most IPC benchmarks are not challenging to ground (Areces et al., 2014). Nevertheless, several interesting planning problems are difficult not because of their combinatorial structure, but because of the intractable size of their ground representations. These *hard-to-ground problems* arise in different contexts, such as natural language processing, genomics, and organic synthesis (Haslum, 2011; Koller and Petrick, 2011; Matloob and Soutchanski, 2016).

In this chapter, we give a first step towards planning directly on the first-order representations. We use well-known techniques from *database theory* to tackle the task of *successor generation*, one of the key operations when planning using heuristic search. The successor generation problem is the following: given an action schema a together with a state s, enumerate all instantiations of $vars(a)$ yielding ground actions that are applicable in s.

successor generation

For example, given the action $drive(T, C_1, C_2)$ from Example 2.4 where

$$\begin{aligned} pre(drive(T, C_1, C_2)) &= \{at(T, C_1), adj(C_1, C_2)\} \\ add(drive(T, C_1, C_2)) &= \{at(T, C_2)\} \\ del(drive(T, C_1, C_2)) &= \{at(T, C_1)\}, \end{aligned}$$

and a state $s = \{adj(\mathrm{a}, \mathrm{b}), adj(\mathrm{b}, \mathrm{a}), adj(\mathrm{b}, \mathrm{c}), adj(\mathrm{c}, \mathrm{b}), at(\mathrm{t}, \mathrm{b})\}$, a successor generator should produce all the instantiations of variables T, C_1,

and C_2 such that $drive(T, C_1, C_2)$ is applicable in s. In this case, there are only two: $drive(\mathsf{b}, \mathsf{a}, \mathsf{t})$ and $drive(\mathsf{b}, \mathsf{c}, \mathsf{t})$.

The enumeration of all applicable ground actions derived from an action schema in a given state s can be seen as a *conjunctive query* where s is a database and the action precondition is a query. In view of this, we introduce successor generation techniques based on standard query evaluation algorithms (Ullman, 1989). We analyze our IPC and HTG benchmark sets and find out that the preconditions of the majority of action schemas correspond to acyclic conjunctive queries, which can be evaluated in time polynomial in the size of the state and number of applicable actions (Yannakakis, 1981).

We implement a lifted planner that uses these techniques for successor generation. Our planner is competitive with ground baseline planners in domains that are hard to ground. In these domains, our planner solves three times more instances than previous lifted planners (Ridder, 2013).

3.1 CONJUNCTIVE QUERIES

We briefly review some relevant background from database theory (Abiteboul et al., 1995; Ullman, 1989).

database

A *relational signature* σ is a finite set of relation symbols, where each relation $r \in \sigma$ has an associated arity $ar(r)$. A *database* D over σ consists of a *domain* Dom and a set $\mathscr{R}$ of finite relations over σ. We stick to the convention of identifying databases with a logical theory, where a tuple $\langle \mathsf{c}_1, \ldots, \mathsf{c}_m \rangle \in r$ is seen as a ground atom $r(\mathsf{c}_1, \ldots, \mathsf{c}_m)$ of a first-order language.

conjunctive query

A *conjunctive query* Q over a database D is a first-order logic formula with the form

$$\exists Y_1, \ldots, Y_m \,.\, p_1(\boldsymbol{X}_1, \boldsymbol{Y}_1) \wedge \cdots \wedge p_k(\boldsymbol{X}_k, \boldsymbol{Y}_k).$$

where $vars(\boldsymbol{X}_i) \subseteq \{X_1, \ldots, X_n\}$, $vars(\boldsymbol{Y}_i) \subseteq \{Y_1, \ldots, Y_m\}$ for all $1 \leq i \leq k$, and $\cup_{i=1}^{k} vars(\boldsymbol{X}_i) = \{X_1, \ldots, X_n\}$. The free variables $X_1, \ldots, X_n$ are the *distinguished variables* of the query.

distinguished variables

Conjunctive queries are often written in *rule form*:

$$q(X_1, \ldots, X_n) \leftarrow p_1(\boldsymbol{X}_1, \boldsymbol{Y}_1), \ldots, p_k(\boldsymbol{X}_k, \boldsymbol{Y}_k).$$

This rule form leaves implicit the existential quantification of the non-distinguished variables $Y_1, \ldots, Y_m$. The relation $q(X_1, \ldots, X_n)$ denotes the *answer* to the conjunctive query.

A conjunctive query can be interpreted as a logic program with a single non-recursive rule. We write $body(Q)$ and $head(Q)$ to denote the right-hand and left-hand sides of a conjunctive query Q.

Example 3.1 *Consider the conjunctive query Q:*

$$q(Y) \leftarrow r(X, Y), s(Y).$$

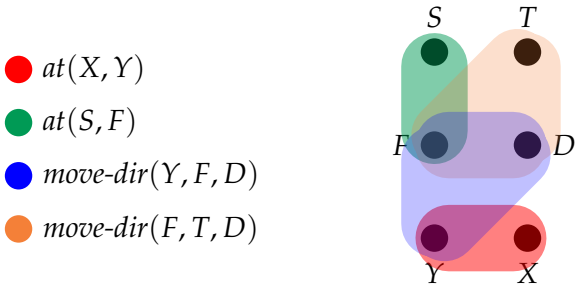


Figure 3.1: Hypergraph $\mathcal{H}(Q_{Ex})$ associated with the query Q_{Ex} from Example 3.2.

The only distinguished variable is Y. The head of the query is $head(Q) = q(Y)$, and $body(Q) = \{r(X,Y), s(Y)\}$.

Let $D = \{r(\mathrm{a},\mathrm{b}), r(\mathrm{c},\mathrm{d}), s(\mathrm{d})\}$ be a database. The answer to Q on D is $\{q(\mathrm{d})\}$.

Conjunctive query answering is **NP**-complete (Chandra and Merlin, 1977). Given a database D and a query

$$q(\boldsymbol{T}) \leftarrow p_1(\boldsymbol{T}_1), \ldots, p_n(\boldsymbol{T}_n).$$

where $vars(\boldsymbol{T}) \subseteq \bigcup_{1 \leq i \leq n} vars(\boldsymbol{T}_i)$, the decision problem asks if the query answer $q(\boldsymbol{T})$ is non-empty. It is easy to come up with a non-deterministic algorithm to solve this problem: guess an instantiation for all variables in the query and check if it unifies the body. It is also easy to show **NP**-hardness. For example, we can reduce the *Hamiltonian path problem* to a conjunctive query. Given a directed graph $G = \langle V, E \rangle$ where $|V| = n$, we construct a database D_{HP} with two binary relations *conn* and *ineq* where

$$conn = \{\langle u, v \rangle \mid (u, v) \in E\},$$
$$ineq = \{\langle u, v \rangle \mid u, v \in V, u \neq v\}.$$

Solving the query below over D_{HP} decides if there is a Hamiltonian path in G:

$$q(V_1, \ldots, V_n) \leftarrow conn(V_1, V_2), \ldots, conn(V_{n-1}, V_n),$$
$$ineq(V_1, V_2), ineq(V_1, V_3), \ldots, ineq(V_{n-1}, V_n).$$

Every conjunctive query Q can be associated with a *hypergraph* $\mathcal{H}(Q) = \langle V(Q), E(Q) \rangle$ with one vertex $v \in V(Q)$ for each variable occurring in $body(Q)$ and one hyperedge $e_i = vars(\boldsymbol{T}_i) \in E(Q)$ for each atom $p_i(\boldsymbol{T}_i)$ in $body(Q)$.

hypergraph

Example 3.2 *Consider the following query, which we refer to as Q_{Ex}:*

$$q(X, Y, S, F, T, D) \leftarrow at(X, Y), at(S, F),$$
$$move\text{-}dir(Y, F, D), move\text{-}dir(F, T, D).$$

Its associated hypergraph $\mathcal{H}(Q_{Ex})$ is shown in Figure 3.1.

GYO reduction

The *GYO reduction* (Graham, 1979; Yu and Ozsoyoglu, 1979) of such a hypergraph is another hypergraph obtained through a simple iterative procedure that removes one hyperedge $e \in E(Q)$ at each step until $E(Q)$ has a single hyperedge or no edge removal can be performed. At each step, we can remove $e \in E(Q)$ iff another hyperedge $f \in E(Q)$ exists such that the variables in $e \setminus f$ only appear in e. We say that such a step *removes e in favor of f*.

The hypergraph $\mathcal{H}(Q)$ is *acyclic* iff its GYO reduction is a hypergraph with a single hyperedge (Ullman, 1989).[1] A conjunctive query Q is *acyclic* iff its hypergraph $\mathcal{H}(Q)$ is acyclic.

acyclic query

Example 3.3 *One possible GYO reduction of the hypergraph $\mathcal{H}(Q_{Ex})$ — see Example 3.2 — is the following:*

1. *remove $at(X, Y)$ in favor of $move\text{-}dir(Y, F, D)$,*
2. *remove $move\text{-}dir(Y, F, D)$ in favor of $move\text{-}dir(F, T, D)$,*
3. *remove $at(S, F)$ in favor of $move\text{-}dir(F, T, D)$.*

Only the hyperedge $move\text{-}dir(F, T, D)$ is still in $\mathcal{H}(Q_{Ex})$. Hence, $\mathcal{H}(Q_{Ex})$ is acyclic, and so is Q_{Ex}.

If the input size of a query evaluation problem is $I = \|Q\|$ and U is the representation size of its answer, we prefer query evaluation algorithms that are output-polynomial (i.e., time and space are polynomial in $I + U$). No complete algorithm can have a runtime better than $O(I + U)$, since it needs to read the input completely and output the query answer. Conjunctive query evaluation is **NP**-hard in general (Chandra and Merlin, 1977), so no efficient general algorithm can exist unless $\mathbf{P} = \mathbf{NP}$. However, output-polynomial algorithms exist for acyclic conjunctive queries (Yannakakis, 1981). In contrast, cyclic conjunctive queries do not have such guarantees of efficiency (Bernstein and Goodman, 1981). When computing a cyclic conjunctive query, we might generate intermediate relations that are exponentially larger than the output.

3.2 RELATIONAL ALGEBRA REDUX

relational algebra

Conjunctive queries are equivalent in expressive power to the select, project, join and rename (SPJR) fragment of *relational algebra* (Codd, 1970). We give an intuitive description of this algebra and refer to Abiteboul et al. (1995) for a formal definition.

named relations

The SPJR algebra is based on *named relations*, often called *tables*. A table is a relation where each position (called *column*) has an *attribute name*. We write $r(X_1, \ldots, X_n)$ to denote an n-ary named relation r with

1 We use the standard database theory characterization of hypergraph α-acyclicity (Fagin, 1983).

attribute names $X_1, \ldots, X_n$. As for sequences of terms, we also use set-theoretical notation on attribute names.

Given two tables $r(\boldsymbol{X})$ and $s(\boldsymbol{Y})$, the basic operations are

(i) to *rename* a column;

(ii) to *project* $r(\boldsymbol{X})$ into a set of attributes $\boldsymbol{Y} \subseteq \boldsymbol{X}$, obtaining a relation $\pi_Y(r(\boldsymbol{X}))$ where some columns of $r(\boldsymbol{X})$ have been removed or rearranged; *projection*

(iii) to *select* some tuples from $r(\boldsymbol{X})$ that either coincide on two different attributes $X_i, X_j \in \boldsymbol{X}$ ($\sigma_{X_i=X_j}(r(\boldsymbol{X}))$), or for which an attribute $X_i \in \boldsymbol{X}$ has a particular constant value c ($\sigma_{X_i=\mathrm{c}}(r(\boldsymbol{X}))$); *selection*

(iv) to *join* $r(\boldsymbol{X})$ and $s(\boldsymbol{Y})$. The *(natural) join* $r(\boldsymbol{X}) \bowtie s(\boldsymbol{Y})$ selects all tuples from the Cartesian product of $r(\boldsymbol{X})$ and $s(\boldsymbol{Y})$ that match on shared attribute names (i.e., $\boldsymbol{X} \cap \boldsymbol{Y}$), and then projects out all copies but one of the duplicate attributes. Here, we assume the join operation ($\bowtie$) is left-associative. *(natural) join*

In addition to these four basic operations, one can define the *semi-join* $r(\boldsymbol{X}) \ltimes s(\boldsymbol{Y})$ as the projection of $r(\boldsymbol{X}) \bowtie s(\boldsymbol{Y})$ to $\boldsymbol{X}$ (i.e., $\pi_{\boldsymbol{X}}(r(\boldsymbol{X}) \bowtie s(\boldsymbol{S}))$. *semi-join*

While relational algebra deals with named relations, the relations in our databases are unnamed. To bridge this gap, we can simply associate each relation in our database with a sequence of attribute names. In our conjunctive queries, our attribute names will be the same as the variables used. We can then solve any conjunctive query

$$q(\boldsymbol{T}) \leftarrow p_1(\boldsymbol{T_1}), \ldots, p_n(\boldsymbol{T_n}).$$

that does not mention constants (i.e., $\boldsymbol{T}_i$ only contains variables for all $1 \leq i \leq n$) by considering each $p_i(\boldsymbol{T_i})$ as a named relation and then computing the following relational algebra operation:

$$q(\boldsymbol{T}) \coloneqq \pi_{\boldsymbol{T}}(p_1(\boldsymbol{T_1}) \bowtie \cdots \bowtie p_n(\boldsymbol{T}_n)). \tag{3.1}$$

If some atom $p(\boldsymbol{T_i})$ mentions a constant c, we introduce a fresh attribute name *Attr*-c to refer to this element in $\boldsymbol{T_i}$. We then select only columns where *Attr*-c equals c (i.e., $\sigma_{Attr\text{-}\mathrm{c}=c}(p(\boldsymbol{T_i}))$). For example, if our query has atom $p(X, \mathrm{a})$ in the body, then we consider it as a named relation $p(X, Attr\text{-}\mathrm{a})$, and we start by selecting $\sigma_{Attr\text{-}\mathrm{a}=a}(p(X, Attr\text{-}\mathrm{a}))$.

3.3 EVALUATING CONJUNCTIVE QUERIES IN PRACTICE

A conjunctive query Q over a database D can be evaluated in two steps: unify all atoms in the body with the database D, and then project the result into the distinguished attributes — see (3.1). The unification step is done by simply performing the natural join of all atoms in the

body of the query. But this is not so simple: bad join orders can lead to exponentially large intermediate results. To avoid this, it is common to *decompose* this unification into smaller steps, ordering how the body of the query should be evaluated. This sequence is called a *query program*. A query program is a sequence of assignments of relational algebra expressions. For example, given a rule

query program

$$q(X, Y, Z) \leftarrow p(X, Y), p(Y, Z), s(Z)$$

we could have the following query program:

$$\begin{aligned} p(Y, Z) &:= p(Y, Z) \ltimes s(Z) \\ s(Z) &:= s(Z) \ltimes p(Y, Z) \\ p(Y, Z) &:= p(Y, Z) \ltimes p(X, Y) \\ p(X, Y) &:= p(X, Y) \ltimes p(Y, Z) \\ q(X, Y, Z) &:= s(Z) \bowtie p(Y, Z) \bowtie p(X, Y). \end{aligned}$$

These operations are interpreted like imperative programming languages — e.g., the first step in the example above replaces the relation $p(Y, Z)$ with $p(Y, Z) \ltimes s(Z)$, and the next step uses this updated relation. When evaluating a query program, each atom in $body(Q)$ is interpreted as a different named relation. So, in the example above, $p(X, Y)$ and $p(Y, Z)$ correspond to different tables, as their attribute names are different.

The assignments of a query program are *local*: the actual database is not changed, but we operate on a local *copies* (also called *materializations*). To illustrate, assume that in the query program above, the atom $p(Y, Z)$ can initially be unified as $p(\mathrm{a}, \mathrm{b})$ and $p(\mathrm{a}, \mathrm{c})$ and atom $s(Z)$ as $s(\mathrm{b})$. The local copy of $p(Y, Z)$ then contains $p(\mathrm{a}, \mathrm{b})$ and $p(\mathrm{a}, \mathrm{c})$, while the copy of $s(Z)$ contains only $s(\mathrm{b})$. After step $p(Y, Z) := p(Y, Z) \ltimes s(Z)$, our local $p(Y, Z)$ contains only $p(\mathrm{a}, \mathrm{b})$ — as $p(\mathrm{a}, \mathrm{c})$ was removed because it does not semi-join with our local copy of $s(Z)$.

Query programs that only use (semi-)joins are called *(semi-)join programs*.

(semi-)join programs

Acyclic Conjunctive Queries

When the conjunctive query is acyclic, output-polynomial evaluation algorithms exist. We next introduce two standard algorithms for this particular case (Bernstein and Goodman, 1981; Yannakakis, 1981).

We first consider conjunctive queries Q

$$q(\boldsymbol{T}) \leftarrow p_1(\boldsymbol{T}_1) \ldots, p_n(\boldsymbol{T}_n).$$

where $vars(\boldsymbol{T}) = \bigcup_i vars(\boldsymbol{T}_i)$. This means that all variables are distinguished and no projection is needed. A *full reducer* of such a query is

full reducer

a semi-join program that filters out tuples from the relations p_i that do not unify with the rest of the body.

Full reducers exist only for acyclic conjunctive queries (Bernstein and Goodman, 1981), and they can be computed during the GYO reduction. Assume that the GYO reduction removes the hyperedges of $\mathcal{H}(Q)$ in the order $(e_1, f_1), \ldots, (e_m, f_m)$, where (e_i, f_i) indicates that iteration i removed hyperedge e_i in favor of hyperedge f_i. (Note that each hyperedge e_i or f_i corresponds to some atom $p_j(\boldsymbol{T}_j)$ in the body of the query.) Since Q is acyclic, all hyperedges but the last one can be removed, so the set of all e_i and f_i covers all hyperedges. Then the semi-join program

$$
\begin{aligned}
f_1 &:= f_1 \ltimes e_1 \\
&\vdots \\
f_m &:= f_m \ltimes e_m \\
e_m &:= e_m \ltimes f_m \\
&\vdots \\
e_1 &:= e_1 \ltimes f_1
\end{aligned}
$$

is a full reducer for $\mathcal{Q}$.

After evaluating the full reducer, we can compute the query answer simply by joining all atoms in the body in any order. It is guaranteed that *no intermediate relation is larger than the answer* (Ullman, 1989). The evaluation of the full reducer with n relations plus the final sequence of joins takes time $O(n(I \log I + U \log U))$, so it is output-polynomial.

Example 3.4 *The GYO reduction from Example 3.3 yields the following semi-join program:*

$$
\begin{aligned}
\mathit{move\text{-}dir}(Y, F, D) &:= \mathit{move\text{-}dir}(Y, F, D) \ltimes \mathit{at}(X, Y) \\
\mathit{move\text{-}dir}(F, T, D) &:= \mathit{move\text{-}dir}(F, T, D) \ltimes \mathit{move\text{-}dir}(Y, F, D) \\
\mathit{move\text{-}dir}(F, T, D) &:= \mathit{move\text{-}dir}(F, T, D) \ltimes \mathit{at}(S, F) \\
\mathit{at}(S, F) &:= \mathit{at}(S, F) \ltimes \mathit{move\text{-}dir}(F, T, D) \\
\mathit{move\text{-}dir}(Y, F, D) &:= \mathit{move\text{-}dir}(Y, F, D) \ltimes \mathit{move\text{-}dir}(F, T, D) \\
\mathit{at}(X, Y) &:= \mathit{at}(X, Y) \ltimes \mathit{move\text{-}dir}(Y, F, D).
\end{aligned}
$$

After executing the full reducer, the query can be computed with the a join of all its relations:

$$
\begin{aligned}
q(X, Y, S, F, T, D) := \mathit{at}(X, Y) &\bowtie \mathit{at}(S, F) \\
&\bowtie \mathit{move\text{-}dir}(Y, F, D) \bowtie \mathit{move\text{-}dir}(F, T, D).
\end{aligned}
$$

It is guaranteed that no intermediate join has more tuples than the answer of the query.

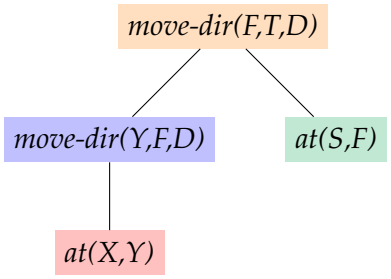


Figure 3.2: Join tree obtained from the GYO reduction of $\mathcal{H}(Q_{Ex})$ (see Example 3.3 and Figure 3.1).

Now, assume that $vars(\boldsymbol{T}) \subset \bigcup_i vars(\boldsymbol{T}_i)$. In words, not all variables are distinguished. In this scenario, the full reducer is no longer guaranteed to be output-polynomial.

Yannakakis' algorithm

In such cases, *Yannakakis' algorithm* (1981) has better asymptotic guarantees, as it interleaves the joins with projections. It starts by checking if the query is acyclic using the GYO reduction, as explained above. As it performs this check, the algorithm also constructs a *join tree* $\mathcal{T}(Q)$ of the query where

1. every atom $p_i(\boldsymbol{T}_i)$ is a node, and
2. node $p_i(\boldsymbol{T}_i)$ is a child of $p_j(\boldsymbol{T}_j)$ iff the hyperedge corresponding to $p_i(\boldsymbol{T}_i)$ was removed in favor of the hyperedge corresponding to $p_j(\boldsymbol{T}_j)$.

Example 3.5 *Figure 3.2 shows the join tree constructed from the GYO reduction of $\mathcal{H}(Q_{Ex})$, in Example 3.3.*

The algorithm then evaluates the query by traversing $\mathcal{T}(Q)$ three times. First, it traverses the tree $\mathcal{T}(Q)$ bottom-up, semi-joining each parent $p_j(\boldsymbol{T}_j)$ with its child $p_i(\boldsymbol{T}_i)$ (i.e., $p_j(\boldsymbol{T}_j) \ltimes p_i(\boldsymbol{T}_i)$) . Second, it traverses $\mathcal{T}(Q)$ top-down, now semi-joining each child $p_i(\boldsymbol{T}_i)$ with its parent $p_j(\boldsymbol{T}_j)$ (i.e., $p_i(\boldsymbol{T}_i) \ltimes p_j(\boldsymbol{T}_j)$). These two traversals are equivalent to evaluating the full reducer of the query, as in the GYO algorithm,[2] and all spurious tuples that do not join any tuple in the query answer are removed. Again, each node corresponds to a local copy of the original relation, so the updates are also local.

The third and last traversal is also bottom-up. When visiting a node $p_i(\boldsymbol{T}_i)$ with parent $p_j(\boldsymbol{T}_j)$, the algorithm updates the parent relation with the following *project-join*:

$$p_j(\boldsymbol{T}') \coloneqq \pi_{\boldsymbol{T}_j \cup (\boldsymbol{T}_i \cap \boldsymbol{T})}(p_i(\boldsymbol{T}_i) \bowtie p_j(\boldsymbol{T}_j))$$

2 The full reducer can also be explained directly in terms of join trees. We decided to introduce it in terms of semi-join programs first to make the order of the operations more explicit.

where $\boldsymbol{T}$ are the distinguished variables of the query and $\boldsymbol{T}' = \boldsymbol{T}_j \cap \boldsymbol{T}$. Once the tree traversal reaches the root node $p_k(\boldsymbol{T}_k)$ of $\mathcal{T}(Q)$, it evaluates the assignment

$$q(\boldsymbol{T}) \leftarrow \pi_{\boldsymbol{T}}(p_k(\boldsymbol{T}_k)),$$

which projects the root node to the distinguished attributes.

While traversing the join tree, all intermediate relations have size bounded by $O(IU)$ (Ullman, 1989; Yannakakis, 1981). Furthermore, the algorithm has runtime $O((I+U)^2)$, so it is output-polynomial.

3.4 A DATABASE PERSPECTIVE OF CLASSICAL PLANNING

Given a planning task $\Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, I, G \rangle$, we consider a state s to be a database $D(s) = \langle \mathcal{C}, \{r_p \mid p \in \mathcal{P}\} \rangle$ where the objects of Π form the domain and there is one relation for every predicate. The relation r_p contains all tuples for which the corresponding ground atom of p is in s, i.e.,

$$r_p = \{\langle c_1, \ldots, c_n \rangle \mid p(c_1, \ldots, c_n) \in s\}.$$

Finding the set of applicable actions for a state can then be expressed as a query. Let $a \in \mathcal{A}$ be an action schema, and $pre(a) = \{p_1(\boldsymbol{T}_1), \ldots, p_n(\boldsymbol{T}_n)\}$ its precondition. We also define $\boldsymbol{T} = vars(a)$. The set of applicable ground actions for a are the ones that are grounded with object tuples the following conjunctive query over $D(s)$ which we call the precondition query of a:

$$q(\boldsymbol{T}) \leftarrow r_{p_1}(\boldsymbol{T}_1), \ldots, r_{p_n}(\boldsymbol{T}_n)$$

Example 3.6 *Consider the standard Gripper domain, where a robot with two grippers must move some balls from one room to another. Each gripper can carry at most one ball. The robot has three actions: move between the two rooms; pick up a ball with one of the grippers; drop one ball.*

Let s be the state where the robot and two balls are in room room_A *and a third ball* ball_3 *is being carried by the robot with its gripper* gripper_1*. The second gripper,* gripper_2*, is free to pick up another ball. The database $D(s)$ contains the relations*

$$\begin{aligned} r_{at} &= \{\langle \text{ball}_1, \text{room}_\text{A} \rangle, \langle \text{ball}_2, \text{room}_\text{A} \rangle\}, \\ r_{carry} &= \{\langle \text{ball}_3, \text{gripper}_1 \rangle\}, \\ r_{at\text{-}robby} &= \{\langle \text{room}_A \rangle\}, \\ r_{free} &= \{\langle \text{gripper}_2 \rangle\}. \end{aligned}$$

Action schema $pick(B, R, G)$ models picking up ball B in room R with robot gripper G, and has preconditions

$$pre(pick(B, R, G)) = \{at(B, R), at\text{-}robby(R), free(G)\}.$$

Its applicable instantiations are described by the conjunctive query

$$q_{pick}(B, R, G) \leftarrow r_{at}(B, R), r_{at\text{-}robby}(R), r_{free}(G).$$

The query evaluates to $\{\langle \text{ball}_1, \text{room}_\text{A}, \text{gripper}_2\rangle, \langle \text{ball}_2, \text{room}_\text{A}, \text{gripper}_2\rangle\}$ *showing that in* s *the two applicable instantiations are to pick up* ball_1 *or* ball_2 *with gripper* gripper_2 *in* room_A.

Whether a precondition query can be efficiently evaluated depends on whether its hypergraph is acyclic. In the rest of the chapter, we refer to action schemas in which the precondition query has an acyclic hypergraph as *acyclic action schemas* and the remaining action schemas as *cyclic action schemas*.

acyclic action schemas

Acyclic Action Schemas

Precondition queries of acyclic action schemas can be evaluated efficiently with the full reducer algorithm. In this case, the computation is polynomial in the size of the state and the number of applicable ground actions: we simply execute the GYO reduction on the precondition query and evaluate the full reducer on the given state.

Sometimes, however, we can do even better than that. In some instances, a state s might have n different applicable actions that lead to the *same successor state*. It turns out that we can improve our successor generator by identifying some of the applicable actions leading to a same state while evaluating the query.

Example 3.7 *Consider the action introduce*(X, Y, Z) *below. It encodes that if a person* X *knows* Y*, and* Y *knows* Z*, then* X *and* Z *can be introduced to each other:*

$$\begin{aligned} \mathit{vars}(\mathit{introduce}(X, Y, Z)) &= \{X, Y, Z\} \\ \mathit{pre}(\mathit{introduce}(X, Y, Z)) &= \{\mathit{knows}(X, Y), \mathit{knows}(Y, Z)\} \\ \mathit{add}(\mathit{introduce}(X, Y, Z)) &= \{\mathit{knows}(X, Z)\} \\ \mathit{del}(\mathit{introduce}(X, Y, Z)) &= \emptyset. \end{aligned}$$

Finding the applicable actions in a state corresponds to solving the following query:

$$q_{introduce}(X, Y, Z) \leftarrow r_{knows}(X, Y), r_{knows}(Y, Z).$$

If two people have more than one common friend that can introduce them, there are several ways to introduce them to each other — all with the same effect.

In the example above, variable Y can be seen as *existentially quantified*. This implies that Y is not distinguished, so it can be left out of the query answer. Instead of mentioning all variables of the action schema

in the query's head as distinguished variables, we only mention those that occur in the effect. Existential quantification of action parameters directly corresponds to the existential quantification in conjunctive queries.

Example 3.8 *Quantifying the variable Y existentially, the precondition query of Example 3.7 becomes*

$$q_{introduce}(X, Z) \leftarrow r_{knows}(X, Y), r_{knows}(Y, Z).$$

This new query contains at most one tuple for each choice of X and Z.

Precondition queries for schemas with existentially quantified variables can be evaluated with Yannakakis' algorithm. As we discussed earlier, this algorithm is quadratic in the size of the input I and the output U. Compared to the full reducer this does not sound like an improvement. However, the output is slightly different: it contains tuples that are different with respect only to the distinguished variables. This can be a much smaller set in action schemas where there are many existentially quantified free variables. For example, if we have state $s = \{knows(\mathrm{a}, \mathrm{b}_1), \ldots, knows(\mathrm{a}, \mathrm{b}_{100}), knows(\mathrm{b}_1, \mathrm{c}), \ldots, knows(\mathrm{b}_{100}, \mathrm{c})\}$, the query in Example 3.7 has 100 instantiations in its answer, while the query in Example 3.8 has only one.

To extract a valid plan for the original PDDL task, we still need to know parameters for the existentially quantified parameters. We compute them by slightly modifying the algorithm. During the tree traversal, we keep one instantiation of the existentially quantified variables for every feasible tuple instantiating the distinguished variables. In this way, we can create fully grounded actions without generating redundant copies.

Cyclic Action Schemas

When the precondition of the action schema is cyclic, a join program can have intermediate relations exponential in the size of the state and number of applicable actions. One way to mitigate this are *evaluation plans* (Abiteboul et al., 1995). These are strategies to decrease the chance of exponentially large intermediate relations.

We consider only a static strategy based on a partial execution of a full reducer. We first obtain run GYO algorithm until no hyperedge can be removed. Since the query is cyclic, more than one hyperedge will be left when the algorithm stop, and the semi-join program computed (until it stops) does not correspond to a full reducer. It can, however, be seen as a "partial reducer" that still filters out unnecessary tuples of some relations. Whenever we want to instantiate a cyclic action schema, we first evaluate this "partial reducer" and then compute a complete join program of the precondition atoms ordered by increasing arity with ties broken according to the order of predicates in the input.

3.5 EXPERIMENTAL RESULTS

Powerlifted

We implemented a lifted planner using the successor generator methods previously described. Our planner is called *Powerlifted*. It supports the PDDL fragment representing STRIPS with inequalities, and types, which are compiled into static predicates.

Powerlifted represents states as set of tables, only keeping track of the atoms of each state. Ground planners (where all possible atoms are known in advance), on the other hand, usually keep track which atoms are true and also which ones are false. In most domains, the ground representation is more efficient (Corrêa, 2019), but the lifted one has the merit of avoiding any sort of grounding.

We start analyzing different successor generator techniques using breadth-first search, and considering all action schemas as unit cost.

Naive Joins

To establish a baseline, we first use a naive join program to evaluate precondition queries. We use three variants that differ in the order in which they join the relations:

- J uses the predicate order as given by the action schema;
- $J^<$ orders the relations by increasing arity, breaking ties according to the order of J;
- J^R joins the relations in a random order. Results are averaged over three runs.

We first report the results for the IPC set. Table 3.1, section "Baselines", shows the results. In 16 of the 29 domains, all methods have identical performance. Using the predicate order from the input (J) performed the best, solving 260 tasks. Ordering the relations by arity ($J^<$) performed slightly worse with a coverage of 237, while a random order (J^R) led to worse results with an average coverage of 223. This difference in coverage shows the large impact of join order on performance and that both orders have a positive effect. The most drastic difference in the IPC set is the freecell domain. Here, J^R and $J^<$ solves 0 tasks, while J solves 14.

In the HTG set, J also achieves the highest coverage. Results are shown in Table 3.2. While J solves 124 tasks, $J^<$ solves 117 and J^R solves 89 on average. In pipesworld-tankage-nosplit domain, bad join orders lead to huge intermediate relations, which consume too much memory. In other domains (e.g., visitall-multidimensional), while the bad join order does not exhaust memory, it still slows down the search and decreases coverage.

To estimate how much larger these intermediate relations are, we checked the size of the largest intermediate relation when generation

	Baselines				
Coverage on IPC set	J	$J^<$	J^R	$FR^{SJ,<}$	Y
airport (50)	**18**	**18**	**18**	**18**	**18**
barman-sat14-strips (20)	**0**	**0**	**0**	**0**	**0**
blocks (35)	**18**	**18**	**18**	**18**	**18**
childsnack-sat14-strips (20)	**0**	**0**	**0**	**0**	**0**
depot (22)	3	3	2	**4**	3
driverlog (20)	**7**	**7**	**5**	**7**	**7**
freecell (80)	**14**	0	0	13	13
grid (5)	**1**	**1**	**1**	**1**	**1**
gripper (20)	**7**	**7**	6	**7**	**7**
logistics00 (28)	**10**	**10**	**10**	**10**	**10**
logistics98 (35)	1	1	1	1	1
miconic (150)	45	45	40	45	45
movie (30)	**30**	**30**	**30**	**30**	**30**
mystery (30)	13	12	8	**15**	**15**
nomystery-sat11-strips (20)	**1**	0	0	**1**	0
openstacks-strips (30)	7	7	7	7	7
parking-sat11-strips (20)	**0**	**0**	**0**	**0**	**0**
parking-sat14-strips (20)	**0**	**0**	**0**	**0**	**0**
pipesworld-notankage (50)	**13**	12	12	**13**	**13**
pipesworld-tankage (50)	7	2	3	**8**	**8**
psr-small (50)	**43**	**43**	**43**	**43**	**43**
rovers (40)	**4**	**4**	**4**	**4**	**4**
satellite (36)	**3**	**3**	**3**	**3**	**3**
thoughtful-sat14-strips (20)	**1**	0	0	0	0
tpp (30)	**5**	**5**	**5**	**5**	**5**
trucks-strips (30)	**2**	**2**	**2**	**2**	**2**
visitall-sat11-strips (20)	**0**	**0**	**0**	**0**	**0**
visitall-sat14-strips (20)	**0**	**0**	**0**	**0**	**0**
zenotravel (20)	7	7	5	7	7
Sum (1001)	260	237	223	**262**	260

Table 3.1: Coverage of different lifted successor generation methods in the IPC set using breadth-first search.

	Baselines				
Coverage on HTG set	J	$J^<$	J^R	$FR^{SJ,<}$	Y
blocksworld-large (40)	**0**	**0**	**0**	**0**	**0**
childsnacks-large (144)	3	3	3	**4**	3
genome-edit-distance (312)	**44**	**44**	**44**	**44**	**44**
logistics-large (40)	**5**	**5**	0	**5**	**5**
organic-synthesis (56)	23	25	13	**44**	**44**
pipesworld-tankage (50)	11	2	3	**12**	11
rovers-large (40)	**0**	**0**	**0**	**0**	**0**
visitall-multidimensional (180)	**38**	**38**	26	**38**	**38**
Sum (862)	124	117	89	**147**	145

Table 3.2: Coverage of different lifted successor generation methods in the HTG set using breadth-first search.

the successors of the initial state. In the pipesworld-tankage-nosplit domain (HTG set), there are instances where J had $\approx 30\,000$ tuples in its largest intermediate relation, while $J^<$ had ≈ 10 million, and J^R ran out of memory when joining tables with more than 50 million tuples. In the IPC set, the visitall-sat14-strips domain also had similarly drastic results: in the initial state, J has at most 4 tuples in any intermediate relation, while the random join J^R has more than 17 million in the largest tasks. Although the search did not run out of memory, these large relations damaged running time. There are also domains where $J^<$ is consistently superior to J, such as the miconic and zenotravel domains. However, the differences are not significant enough to impact the total coverage in these domains.

Acyclic Schemas

The naive join methods fail when the intermediate relations become too large. For acyclic action schemas this can be avoided with the full reducer. But how many of the action schemas are acyclic? The third column ("Acyc.") in Table 3.3 shows the proportion of acyclic schemas in each domain. In the IPC set the average proportion of acyclic schemas is 87.1%, and 19 domains have only acyclic schemas. This is good news for our method, since grounding these schemas can be done efficiently in all states. The situation looks worse in the hard-to-ground domains, where the average proportion of acyclic schemas is 57.8%. In particular, the challenging organic-synthesis domain has only 5% acyclic schemas. Looking closer at the hypergraph of the

precondition queries of these schemas, it turns out that most of the cycles are caused by inequalities. If we ignore them, the incidence of acyclic schemas increases to over 90% in organic-synthesis, and to 81.9% in the HTG set (fourth column of Table 3.3). The notable exception is in pipesworld-tankage where all action schemas are cyclic even when ignoring inequality relations. None of the domains in our IPC set has inequalities in the precondition, so ignoring them does not affect acyclicity.

Therefore, we do not consider inequalities as part of the query when computing the full reducer, and we handle them by removing tuples that violate such constraining after every join. There is a more sophisticated algorithm for acyclic conjunctive queries with inequalities that is fixed-parameter tractable in the size of the query and the number of variables (Papadimitriou and Yannakakis, 1999) and could improve the results. We consider this future work.

To test the benefit of exploiting acyclicity, we ran our breadth-first search with the configuration $FR^{SJ,<}$, which uses the GYO reduction in a preprocessing step to test which queries are cyclic. As a side effect, this step finds a full reducer for acyclic schemas and a partial reducer for cyclic ones. During the search, all queries first execute the full/partial reducer before the final join. For acyclic schemas the algorithm then executes the join program in the order established by the hyperedge removals. For cyclic schemas, the final join program is J.

Compared to J, using $FR^{SJ,<}$ to generate successors improves the coverage from 260 to 262 in the IPC set and from 124 to 147 in the HTG set. The largest improvement is in the organic-synthesis domain, where J and $J^{<}$ solved 23 and 25 tasks respectively, while $FR^{SJ,<}$ solves 44 instances.

The main advantage of using the full reducer is that it avoids large intermediate relations. Trying to estimate how often this occurs, we compared the largest intermediate relation size when expanding the initial state using J and $FR^{SJ,<}$. Figure 3.3 shows the results for the HTG set. In instances with a majority of acyclic actions, the largest intermediate relation computed by J is sometimes five order of magnitude larger than the one computed by $FR^{SJ,<}$. In domains with cyclic schemas, the initial partial reducer semi-join program used by $FR^{SJ,<}$ also seems to help in general. However, in the rovers-large domain the opposite is true: J outperforms $FR^{SJ,<}$, sometimes by more than two orders of magnitude. In this domain, some tasks have very large initial states (more than 70 000 atoms), but $\approx$ 80% of these atoms are in the same relation (*visible*). It is then important that join orders minimize this relation as early as possible. This happens by chance with J, but not with $FR^{SJ,<}$. One way to solve this issue is to consider secondary heuristic methods (such as relation size) when computing a full/partial reducer.

	$\lvert\mathcal{A}\rvert$	**Acyc.**	**Acyc.** $\neq$	$\exists$**-quant.**
barman-sat14-strips	12	91.7%	91.7%	83.3%
freecell	10	70.0%	70.0%	30.0%
nomystery-sat11-strips	3	66.7%	66.7%	33.3%
pipesworld-notankage	6	33.3%	33.3%	100.0%
pipesworld-tankage	10	10.0%	10.0%	100.0%
rovers	9	88.9%	88.9%	66.7%
satellite	5	80.0%	80.0%	40.0%
thoughtful-sat14-strips	21	61.9%	61.9%	38.0%
tpp	4	75.0%	75.0%	50.0%
Other 19 domains	68	100.0%	100.0%	25.0%
IPC Domains		87.1%	87.1%	34.9%
blocksworld-large	3	100.0%	100.0%	0.0%
childsnack-contents	48	62.5%	75.0%	25.0%
genome-edit-distance	14	35.7%	100.0%	0.0%
genome-edit-distance-split	21	71.4%	100.0%	0.0%
organic-synthesis	116	4.3%	91.4%	91.4%
pipesworld-tankage	4	0.0%	0.0%	100.0%
rovers-large	9	88.9%	88.9%	66.7%
visitall-multidim.	72	100.0%	100.0%	0.0%
HTG Domains		57.8%	81.9%	35.9%

Table 3.3: Proportions of action schemas that are acyclic (Acyc.), acyclic when ignoring inequalities (Acyc. $\neq$), and that have existentially quantified parameters ($\exists$-quant.). Rows **IPC Domains** and **HTG Domains** show the average proportions over each set.

When comparing total search time, $FR^{SJ,<}$ is usually faster than the baseline methods. Figure 3.4 compares total search time for $FR^{SJ,<}$ and J in the HTG set. In most domains, both methods have similar run time. However, in organic-synthesis and pipesworld-tankage, $FR^{SJ,<}$ is sometimes much faster. Although one could expect that the $FR^{SJ,<}$ method would be slower due to the additional overhead of evaluating a semi-join program prior to the full join program, this preprocessing steps pays off and $FR^{SJ,<}$ is competitive on all instances. It is interesting to see that $FR^{SJ,<}$ is also superior in the pipesworld-tankage domain, albeit all its actions being cyclic. This shows that our method for cyclic actions — using the semi-join program computed by the GYO reduction until it stops — is better than a naive join. In the IPC set, we

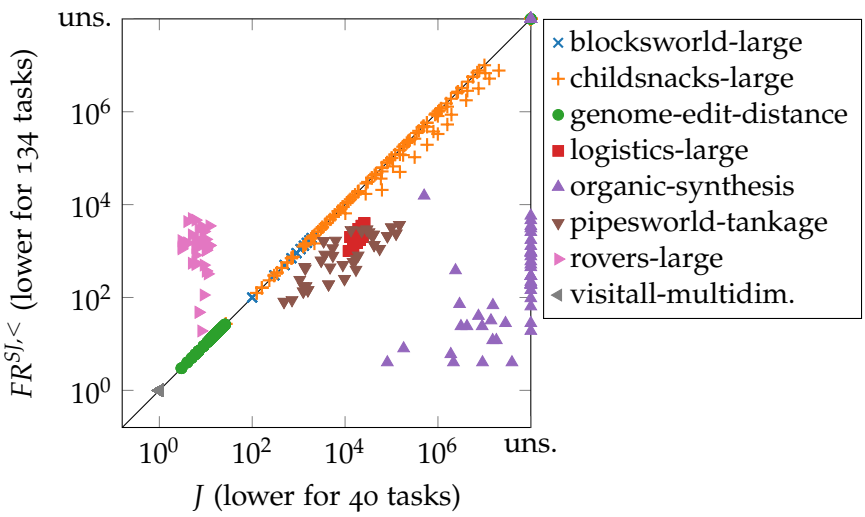


Figure 3.3: Size of the largest intermediate relation when computing successor states for the initial state of each task in the HTG set.

observed a similar trend, although the running times of $FR^{SJ,<}$ and J are more similar.

Existentially Quantified Preconditions

We also discussed a successor generator based on Yannakakis' algorithm that has a potential gain for instances with existentially quantified variables. For example, the organic-synthesis domain has action schemas with 17 variables where only four appear in the effect. On average, the proportion of action schemas with one or more existentially quantified variables per domains is 34.9% in the IPC set, and 35.9% in the HTG set (see Table 3.3).

As there are many tasks with a potential gain, we implemented and evaluated a successor generator Y that uses Yannakakis' algorithm to create only one grounded action for each choice of distinguished variables. Using Y, the successor generator outputs one instantiation for each different successor state, instead of each applicable instantiation. This means that if several instantiations lead to the same state, Y is able to identify them and save some effort. For cyclic action schemas Y uses the same fall back strategy as $FR^{SJ,<}$. In our implementation, variables occurring in inequalities are considered distinguished to preserve completeness.[3] We also do not consider inequalities when

3 To illustrate why this is necessary, consider the following query: $q(X,Y,Z) \leftarrow p(X,Y), q(Y,A), r(Z), A \neq Z$. A possible join tree (that ignores the inequality) for this query has $p(X,Y)$ as the root, and the other two positive atoms as its children. As we keep only one instantiation of the non-distinguished variable A for every instantiation of Y, it could be that the chosen instantiation violates $A \neq Z$, while other instantiations for A satisfy the inequality. However, we can only find out which values of A violate the inequality after we know the instantiations of Z. To simplify

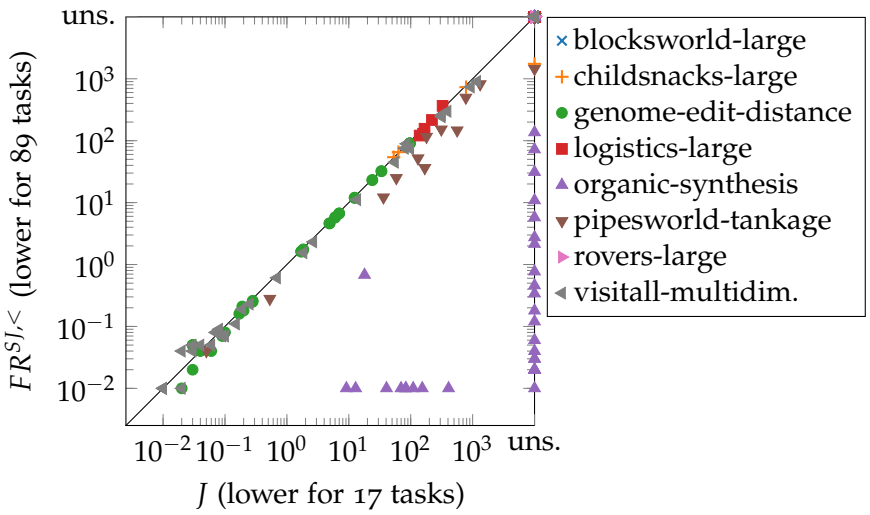


Figure 3.4: Time comparison between J and $FR^{SJ,<}$ on the HTG set.

computing the join trees, and we enforce them after the joins as done in the implementation of $FR^{SJ,<}$.

Surprisingly, using Y in a breadth-first search is slightly worse than using $FR^{SJ,<}$ (see Tables 3.1 and 3.2). It solves two tasks less on each set.

Although several instances have potential savings in the number of generated states using Y, we did not see any improvement. All instances where Y reduces the number of duplicated successors states are either in the movie domain (IPC set) or in organic-synthesis (HTG set). Still, it did not increase coverage in either of these domains.

We also do not see much gain in time and memory usage. Once more, the only domain where there is an observable reduction in memory is organic-synthesis, where Y reduces the number of generated successors. However, in general, Y has an additional overhead compared to $FR^{SJ,<}$ and thus the time saved by avoiding duplicated states usually does not pay off.

Comparison to Ground Planners

We also compare the performance of our best method, $FR^{SJ,<}$, to a ground planner. For this experiment, we first use the breadth-first search implemented in Fast Downward (Helmert, 2006), version 23.06.

The IPC set consists mostly of tasks that are easy to ground, so we expect a Fast Downward to perform better than any of the methods using Powerlifted. Indeed, Fast Downward solves 345 instances, while $FR^{SJ,<}$ solves 262. Table 3.4 shows the results under the header "BFS".

our algorithm, we consider A also as distinguished, so we keep all its instantiations. There are more sophisticated solutions that we did not explore.

	BFS		GBFS	
	FD	$FR^{SJ,<}$	FD	$FR^{SJ,<}$
Coverage IPC set (1001)	345	262	**683**	603
blocksworld-large (40)	1	0	2	**4**
childsnacks-large (144)	9	4	**32**	26
genome-edit-distance (312)	48	44	**312**	**312**
logistics-large (40)	9	5	**21**	20
organic-synthesis (56)	21	44	21	**49**
pipesworld-tankage (50)	16	12	20	**23**
rovers-large (40)	3	0	**15**	4
visitall-multidimensional (180)	**72**	38	**72**	65
Coverage HTG set (862)	179	147	495	**503**

Table 3.4: Coverage comparison between Fast Downward (FD) and Powerlifted with $FR^{SJ,<}$. Using two different configurations: one with breadth-first search (BFS); and one with greedy best-first search (GBFS) using the goal-count heuristic.

Fast Downward was equal or superior to Powerlifted with the $FR^{SJ,<}$ successor generator in all domains of this set.

On hard-to-ground domains, we expect Powerlifted to be more competitive with Fast Downward. Table 3.4 also shows the results for this set. The only domain where $FR^{SJ,<}$ improves the total coverage is the organic-synthesis domain. In all other domains, Fast Downward solves up to 5 instances more than Powerlifted, with one exception: the visitall-multidimensional domain. In this domain, an agent must visit a single cell in an n-dimensional grid, for $3 \leq n \leq 5$ (Lauer et al., 2021). Although this domain is challenging to ground because of the many variables of actions and predicates, many tasks have short plans, and so the search is relatively simple. In fact, Fast Downward found plans for all tasks that it could ground in visitall-multidimensional. As this domain is over-represented in the benchmark set, this ended up skewing the total results as well.

Generally speaking for the HTG set, we expect Fast Downward's main bottleneck to be the grounding while the main bottleneck of Powerlifted is the search. So our hypothesis is that guiding the search with a more informed heuristic changes the results.

To test this hypothesis, we implemented the *goal-count heuristic* (Fikes and Nilsson, 1971) in our planner. The goal-count heuristic simply counts the number of ground atoms in the goal G that are unsatisfied in the state being evaluated. We then use the goal-count

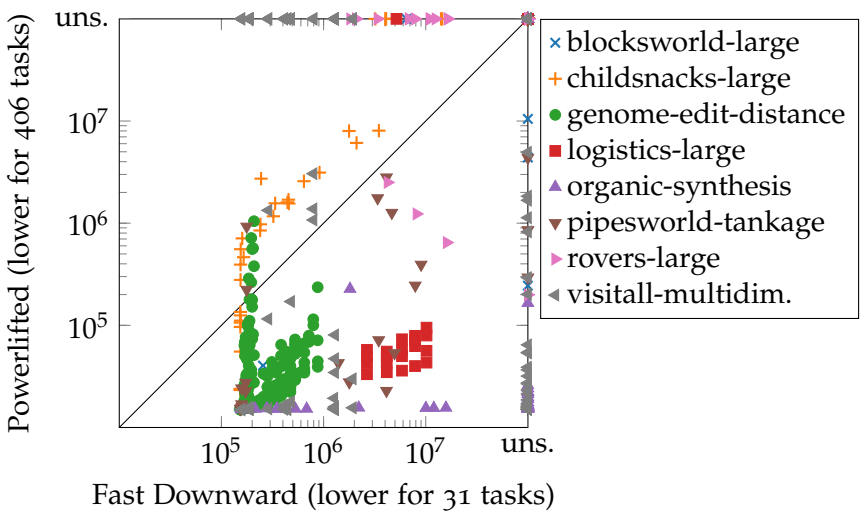


Figure 3.5: Peak memory (in kB) for Powerlifted with $FR^{SJ,<}$ and Fast Downward on the HTG set. Both use the configuration with GBFS and the goal-count heuristic.

heuristic to guide a greedy best-first search (GBFS). Goal-count is not one of the most informed heuristics in the literature, but it has the benefit of being action-independent, and therefore apt for both of ground and lifted representations directly.

The second block of Table 3.4 (with header "GBFS") shows how a GBFS with goal-count performs in Fast Downward and Powerlifted with $FR^{SJ,<}$. We can see that the heuristic indeed had a larger benefit for Powerlifted which now outperforms Fast Downward in total coverage on the HTG set. With the heuristic, the search expands fewer states which means the overhead of computing the successor states is less relevant. A ground planner on the other hand requires the same amount of effort to ground the task. This can be seen in the organic-synthesis domain, where all tasks that Fast Downward fails to solve, run out of memory. The same is true for the pipesworld-tankage domain, where $FR^{SJ,<}$ solves 3 tasks that Fast Downward is not able to ground. In Chapter 4, we will see that with more informed heuristics, the gap between Powerlifted and ground planners in the HTG set becomes even larger.

When comparing memory consumption, Powerlifted has a lower memory consumption than Fast Downward in most tasks. Figure 3.5 plots the results of both planners when using GBFS with the goal-count heuristic. Powerlifted with $FR^{SJ,<}$ consumes less memory in 406 tasks, while Fast Downward is more memory efficient in 31 (only counting those solved by both planners). The only domain where Powerlifted with $FR^{SJ,<}$ consistently used more memory was the childsnacks-large domain. In some domains (logistics-large, organic-synthesis, and most of the genome-edit-distance tasks), the lifted planner used 2 or 3 orders of magnitude less memory than Fast Downward.

	LAMA	L-RPG	$FR^{SJ,<}$
Coverage IPC set (1001)	**917**	324	603
blocksworld-large (40)	**12**	0	**4**
childsnacks-large (144)	**115**	–	26
genome-edit-distance (312)	**312**	110	**312**
logistics-large (40)	**36**	0	20
organic-synthesis (56)	21	14	**49**
pipesworld-tankage (50)	18	11	**23**
rovers-large (40)	**17**	0	4
visitall-multidimensional (180)	**72**	19	65
Coverage HTG set (862)	**603**	154	503

Table 3.5: Coverage comparison between LAMA, L-RPG, and Powerlifted with $FR^{SJ,<}$. For Powerlifted, we run a greedy best-first search using the goal-count heuristic. L-RPG fails to run in childsnacks-large, but we could not identify the source of the problem.

Solving Hard-to-Ground Domains

To get a better understanding of our methods, we also compared them to complete off-the-shelf planning systems. We compared our methods to LAMA (Richter and Westphal, 2010) and to L-RPG (Ridder, 2013; Ridder and Fox, 2014). Table 3.5 show the coverage results of LAMA and L-RPG in comparison to Powerlifted with $FR^{SJ,<}$. LAMA is a ground planner that has been considered state-of-the-art for more than a decade, while L-RPG is the only PDDL planner using lifted representations directly. LAMA is built on top of Fast Downward, so it faces exactly the same grounding challenges as Fast Downward. L-RPG implements an approximation of the FF heuristic (Hoffmann and Nebel, 2001), computed directly over the lifted representation. (The FF heuristic is discussed in detail in the next chapter.) Additionally, L-RPG computes equivalence relations between objects to find symmetries. However, L-RPG does not have a dedicated algorithm to generate successor states: it simply enumerates all possible action instantiations and checks for their applicability.

LAMA and GBFS with the goal-count heuristic in Fast Downward perform similarly in many domains of the HTG set (organic-synthesis, pipesworld-tankage, rovers-large, visitall-multidimensional). This reinforces the result that the main bottleneck of these domains is the grounding and not the search itself. Adding a powerful search method to a ground planner does not improve the coverage in these domains, simply because the search itself is not the hardest part. In these do-

mains, the time and memory usage of LAMA is similar to GBFS with goal-count: both run time and peak memory are dominated by the grounding step.

On the other domains of the HTG set and on the IPC set, however, LAMA is superior to the other methods. Grounding still consumes a lot of time in these domains, but the search component of LAMA is so powerful that it compensates for the grounding time. For these domains, having a more informed lifted heuristic is beneficial too (see Chapter 4).

The comparison to the L-RPG planner is very favorable to Powerlifted, even though L-RPG uses a more informed heuristic to guide the search. L-RPG aborted within seconds on all instances of the childsnacks-large domain. Unfortunately, we were not able to identify why this happened, so we decided do not report values for this domain. The total coverage of L-RPG in the HTG set is 154, while Powerlifted with $FR^{SJ,<}$ solves 477 tasks (when not considering the 26 tasks solved on the childsnacks-large domain). These results show that for hard-to-ground domains the successor generation is one of the main bottlenecks for lifted planners. Our methods are also faster than L-RPG in all tasks. In the smallest instances, just the computation of the equivalence relations in L-RPG already takes more time than Powerlifted takes to find a plan. In larger instances, our methods are faster than L-RPG in spite of a less informed heuristic. This could be due to L-RPG spending too much time instantiating action schemas or to the lifted FF computation of L-RPG being too expensive in these domains.

3.6 SUMMARY

In this chapter, we gave a first step towards a heuristic search planner using a lifted representation. We studied how to efficiently perform lifted successor generation in classical planning using well-known database techniques. The problem of generating all ground actions that derive from an action schema and are applicable in a given state is equivalent to evaluating a query given by the schema precondition in a database given by the state. Often, the action preconditions of standard planning domains fall into the tractable case of acyclic conjunctive queries.

Our planner, called Powerlifted, implements several successor generators based on query optimization techniques. We empirically evaluated these different methods for both the acyclic and cyclic action schemas. Our results show that this approach has an acceptable overhead in most standard benchmarks, and is a preferable alternative to state-of-the-art ground planners in many domains that are hard to ground.

But there are still many ideas that could be tried out to improve performance. In particular, it would be interesting to test other sorts of structural decompositions (Gottlob et al., 2002), particularly for cyclic actions. As we will see in Part ii in the context of grounding, *all action schemas* in our benchmark sets have low (hyper)treewidth, which means that certain structural decompositions could help without being very expensive. Another potential topic of future research is to study different decompositions (i.e., query programs, join trees) for acyclic queries. We mentioned earlier that the full reducer and Yannakakis' algorithm can be phrased in terms of join trees. It is well-known in database theory that different join trees can lead to very different performances (Scarcello et al., 2007), although the asymptotic complexity remains unchanged. In classical planning, this could also have an observable impact.

In the next chapter, we move forward to study *how to compute informed heuristics over the lifted representation*. Together with successor generation, informed heuristic functions are key aspects of efficient heuristic search planners. Our next goal is to compute delete-relaxation heuristics only using information about the current state and the action schemas. More informed heuristics lead to fewer expansions, which reduces the overhead caused by lifted successor generation. As we saw in the previous experiments, LAMA improved on GBFS with the goal-count heuristic simply by using a more informed search. Our objective is to obtain the same improvement on Powerlifted. However, we will be faced with another problem: how to compute heuristics *efficiently* in our setting?

CHAPTER NOTES AND HISTORY

We first developed Powerlifted in 2020 (Corrêa et al., 2020). Since then, other lifted successor generation methods were built on top of Powerlifted (e.g., Ståhlberg, 2023). This work also served as inspiration for more recent lifted planners (e.g., Horčík and Fišer, 2021), which also rely on database techniques to generate successors.

Our current implementation differs from the original one in a few aspects, which actually impact performance. There were two significant issues with the original implementation of Powerlifted:

1. variables occurring in inequalities were not considered distinguished in Yannakakis' algorithm, and
2. the full reducer and Yannakakis' successor generators performed the two traversals of the join-tree in the wrong order.[4]

These two issues were later fixed (Corrêa et al., 2021).

Planning techniques that do not depend on grounding long existed (see chapter notes of Chapter 2), although recent research has focused

4 This problem was identified by Davide Mario Longo.

on ground planning. For instance, some planning-as-satisfiability approaches use encodings that avoid the need for grounding all actions at preprocessing step by using propositions representing the actual grounding of the action executed at each time step (Kautz et al., 1996; Kautz and Selman, 1992; Robinson et al., 2008). Similar encodings have been proposed in recent compilations of numeric and temporal planning to SMT or CSP (Bit-Monnot, 2018; Bofill et al., 2016; Espasa et al., 2019). Other approaches to planning, such as partial order planning, have also explored the use of lifted instead of ground actions (Penberthy and Weld, 1992; Younes and Simmons, 2002, 2003). However, Powerlifted (and other modern lifted planners) have a crucial difference: they perform a ground search, where while the representation of actions is lifted, the explored state space is still ground. This is different from previous approaches in partial order planning (e.g., SNLP, UCPOP, VHPOP) that used partially ground actions and atoms in the plan-space search — e.g., grounding only the variables of an action that were relevant to validate a search node.

In the context of planning as heuristic search, the Unpop planner by McDermott (1996) plans with lifted action schemas using simple unification and regression techniques, but it was outperformed by other contemporary heuristic search planners that used propositional representations. Many of the heuristic search planning techniques use some form of preprocessing that combines grounding with a relaxation-based *reachability analysis* that avoids grounding some actions that can be proven not to be applicable in any state reachable from the initial state (Helmert, 2009). The ground actions that result from this procedure are often clustered in a decision-tree-like data structure that speeds up the successor generation task by up to two orders of magnitude on some IPC benchmarks (Helmert, 2006). The query optimization techniques we present in this work are inspired by the ones that Helmert (2009) applies to the grounding step, but we apply them online, and explicitly exploit the acyclicity of precondition queries. One can also try to use Helmert's approach in the successor generation.

More recent work in the context of lifted planning in the heuristic search context includes the work by Ridder (2013). Ridder's L-RPG planner focuses more on lifted versions of standard heuristics than on the successor generation task. We compared the performance of L-RPG with our approach in the experimental results section, which offers empirical evidence on the importance of an efficient lifted successor generator.

Areces et al. (2014) develop an automatic action schema splitting technique that reduces the number of parameters of action schemas, at the cost of modifying the state space topology. Since theirs is a model reformulation approach, it has the advantage that it can be coupled with any planner. Gnad et al. (2019) present a machine-

learning method that incrementally grounds larger and larger parts of the full set of ground actions until a plan can be found. Lifted approaches have also been considered for other planning-related tasks such as the computation of problem invariants and symmetries (Fišer, 2020; Rintanen, 2017; Röger et al., 2018; Sievers et al., 2019).

Francès and Geffner (2016) implement a planner that exploits existentially quantified variables in preconditions and goal directly. They show a connection between planning with existentially quantified variables and constraint satisfaction problems. However, their planner represents tasks propositionally. Francès (2017) implement a lifted successor generator for FSTRIPS — STRIPS extended with function symbols. His planner simply reduces the preconditions to a CSP and then uses an off-the-shelf solver to compute the successors. However, the implementation still requires that all reachable ground atoms (but not actions) are known in advance. As we will see in the next chapters, this is also not straightforward in all domains.

4

LIFTED DELETE-RELAXATION HEURISTICS

Heuristic search (Bonet and Geffner, 2001) is a successful approach to ground planning. The main idea is to search for a solution in the space of all applicable action sequences. Using a heuristic, the planner focuses the search on promising sequences and thus speeds up the process.

In the previous chapter, we showed how to implement an efficient successor generation at a lifted level, but our planner still lacks a good heuristic function to guide the search. To benefit from the advances in ground planning, in particular in the area of heuristic search, we need strong *lifted heuristics* as well. Moreover, our earlier results show that stronger heuristics (which results in fewer expansions) will also reduce the overhead of our lifted successor generator.

In this chapter, we study how to compute delete-relaxation heuristics *directly from the lifted representation*. More specifically, we extend the Datalog formulation of relaxed reachability by Helmert (2009) to compute the additive heuristic h^{add}, the maximum heuristic $h^{\max}$ (Bonet and Geffner, 2001), and the FF heuristic h^{FF} (Hoffmann and Nebel, 2001) over the lifted representation. We also show how to extract other relevant information (e.g., useful atoms, preferred operators) from the Datalog evaluation.

A first implementation of these heuristics gives good guidance to the search, but they are too expensive to compete even with the goal-count heuristic from the previous chapter. This straightforward implementation grounds part of the (relaxed) state space to extract a relaxed plan. In larger tasks, such as the ones in our HTG set, this can be as expensive as grounding the entire task.

To tackle this problem, we introduce the notion of an *annotated Datalog program*, where atoms and rules are associated with instructions. After evaluating a Datalog query, we execute the instructions of all atoms and rules used to derive this query. This allows us to collect a relaxed plan and compute the desired heuristics. Additionally, this allows us to simplify annotated Datalog programs to make them cheaper to evaluate, while still preserving the same heuristic values. In our experimental results, Powerlifted with the lifted h^{FF} heuristic achieves state-of-the-art performance among lifted planners and is

competitive with Fast Downward (Helmert, 2006) and LAMA (Richter and Westphal, 2010) using a ground implementation of the heuristic.

4.1 DELETE-RELAXATION HEURISTICS OVER GROUND TASKS

As mentioned earlier (Chapter 2), computing optimal plans for a *ground* delete-relaxed task Π^+ is **NP**-complete (Bylander, 1994). But there are approximations that compute (possibly) suboptimal plans for Π^+ in polynomial time (Bonet and Geffner, 2001; Hoffmann and Nebel, 2001) or that give a lower bound to the length (or cost) of the optimal plan for Π^+ (Bonet and Geffner, 2001; Helmert and Domshlak, 2009).

We are interested in three delete-relaxation heuristics:

- the *additive heuristic* h^{add} (Bonet and Geffner, 2001; Keyder and Geffner, 2008);
- the *max heuristic* $h^{\max}$ (Bonet and Geffner, 2001; Keyder and Geffner, 2008); and
- the *FF heuristic* h^{FF} (Hoffmann and Nebel, 2001).

The additive and the FF heuristic estimate the cost of a *relaxed plan*: plans for the delete-relaxation Π^+. This is a common source of heuristic guidance on the non-relaxed task Π, where the length of the relaxed plan can be used as a distance estimate. Both h^{add} and h^{FF} are inadmissible, so they cannot guarantee an optimal solution when used with A*, for example. The max heuristic, in contrast, underestimates the cost of the optimal relaxed plan h^+ (i.e., cost of the optimal plan in Π^+; see Chapter 2) and is then admissible.

Additive and Max Heuristic

We introduce h^{add} and $h^{\max}$ together, as they are analogous. Given a ground planning task $\Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, I, G \rangle$,[1] we define $h(p, s)$ as the cost of achieving ground atom p from a state s:

$$h(p,s) = \begin{cases} 0, & \text{if } p \in s \\ \min\limits_{a \in A_p} \left(cost(a) + \sum\limits_{q \in pre(a)} h(q,s) \right), & \text{otherwise,} \end{cases} \tag{4.1}$$

where A_p is the set of ground actions with p in the add list. This system of equations has a *unique maximal solution*. If all actions have costs larger than zero, the maximal solution is unique. We assume in the rest of this chapter that costs are larger than zero — tasks originally

1 Recall that a ground task is simply a task where $vars(a) = \emptyset$ for all $a \in \mathcal{A}$ (Chapter 2).

Algorithm 1 Extract relaxed plan from computation of h^{add}.

Input: Evaluation of best achievers for every atom in the task, after computing h^{add}.
Output: Relaxed plan π.

```
1: π := ⟨⟩
2: for g ∈ G do
3:     BackChain(g)
4: return Reverse(π)

5: function BackChain(p)
6:     if p ∉ I then
7:         π.Add(a_p)
8:         for q ∈ pre(a_p) do
9:             BackChain(q)
       return
```

with zero cost actions can be transformed so they have a small cost $\varepsilon > 0$ cost. Moreover, we let $\min \varnothing = \infty$ so $h(p, s)$ maps to $\mathbb{R} \cup \{\infty\}$.

The value of the *additive heuristic* h^{add} is then defined as *additive heuristic*

$$h^{\text{add}}(s) = \sum_{p \in G} h(p, s). \tag{4.2}$$

We can extract a plan from the computation of h^{add}. First, we define the *best achiever* a_p of an atom p as *best achiever*

$$a_p = \arg \min_{\substack{a \in \mathcal{A} \\ p \in add(a)}} cost(a) + \sum_{q \in pre(a)} h(q, s). \tag{4.3}$$

if $p \notin s$, and $a_p = \varnothing$ otherwise. We can back-chain from the goal atoms through their best achievers to extract the plan. Algorithm 1 shows the pseudocode. Assume that π is a global variable. For each atom g in the goal, the algorithm adds its best achiever a_g to the plan, and then continues back-chaining through the atoms in $pre(a_g)$. This chaining continues until we reach an atom that is in the initial state ($p \in I$). Note that we must reverse the order of π at the end.

When computing the cost to reach a precondition $pre(a) = \{p, q\}$, Equation (4.1) ignores that reaching p might help in reaching q too. So h^{add} might overestimate the real cost to achieve the goal, and is non-admissible.

Replacing the $\sum$ operator in (4.1) and (4.2) by the max operator gives us the *max heuristic* $h^{\max}$ (Bonet and Geffner, 2001): *max heuristic*

$$h'(p, s) = \begin{cases} 0, & \text{if } p \in s \\ \min\limits_{a \in A_p} cost(a) + \max\limits_{q \in pre(a)} h'(q, s), & \text{otherwise,} \end{cases} \tag{4.4}$$

$$h^{\max}(s) = \max_{p \in G} h'(p, s). \tag{4.5}$$

We can view $h^{\max}$ as an optimistic version of h^{add}: $h^{\max}$ assumes that the cost to achieve a set of atoms is the same is achieving its most expensive component individually; h^{add} assumes that the cost to achieve the set is the sum of all parts. In contrast to h^{add}, $h^{\max}$ is admissible, as the cost of achieving a set of atoms cannot be lower than the cost of achieving each of the atoms in the set individually.

FF Heuristic

The h^{add} heuristic assumes that there is no positive synergy when achieving action preconditions. To solve this issue, Hoffmann and Nebel (2001) introduce the *FF heuristic*, h^{FF}.

FF heuristic

Let a_p be the best achiever of p as in (4.3).[2] Let also

$$\pi(p) = \begin{cases} \{\}, & \text{if } p \in s \\ \{a_p\} \cup \bigcup_{q \in pre(a_p)} \pi(q), & \text{otherwise.} \end{cases} \tag{4.6}$$

If we fix a_p for each reachable atom p and if G is reachable, we can compute a unique solution to these equations by recursively evaluating it starting from G. Let $\pi(G) = \bigcup_{g \in G} \pi(g)$. In this case, $\pi(G)$ can be sequenced into a relaxed plan π_{FF} and the FF heuristic is defined as $h^{\text{FF}}(s) = cost(\pi_{\text{FF}})$. If G is not reachable in the delete relaxation, then $h^{\text{FF}}(s) = \infty$.

The FF heuristic h^{FF} (Hoffmann and Nebel, 2001) is part of state-of-the-art ground planners since its creation (e.g., Richter et al., 2011).

4.2 LIFTED RELAXED REACHABILITY

Instead of directly discussing how to compute h^{add}, $h^{\max}$, and h^{FF} with a lifted representation, we start with a simpler problem: how to compute all atoms that are *reachable* in a lifted delete-relaxed task?

For a given planning task $\Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, I, G \rangle$ and a state s, Helmert (2009) encodes Π^+ as a Datalog program $\mathcal{D}_s = \langle \mathcal{F}, \mathcal{R} \rangle$. The set of facts $\mathcal{F}$ contains all ground atoms in s, and $\mathcal{R}$ is defined as follows: for each action schema $a(\boldsymbol{T}) \in \mathcal{A}$ with variables $\mathbf{T}$ and $pre(a(\boldsymbol{T})) = \{p_1(\boldsymbol{T}_1), \ldots, p_n(\boldsymbol{T}_n)\}$, $\mathcal{R}$ contains the *action applicability rule*

action applicability rule

$$a\text{-applicable}(\boldsymbol{T}) \leftarrow p_1, (\boldsymbol{T}_1) \ldots, p_n(\boldsymbol{T}_n).$$

where a-applicable$(\boldsymbol{T})$ is an *action applicability atom* using the fresh predicate symbol a-applicable.

action applicability atom

For each $q_i(\boldsymbol{T}_i) \in add(a(\boldsymbol{T}))$, $\mathcal{R}$ also contains the *action effect rule*

action effect rule

$$q_i(\boldsymbol{T}_i) \leftarrow a\text{-applicable}(\boldsymbol{T}).$$

Last, $\mathcal{R}$ also contains a *goal rule*

goal rule

2 There are different ways to select the best achiever of an atom. While Hoffmann and Nebel choose the best achievers based on $h^{\max}$, Keyder and Geffner (2008) do so with h^{add}. We follow Keyder and Geffner — see Equation (4.3).

$$goal \leftarrow g_1, \ldots, g_m.,$$

where *goal* is a fresh predicate symbol, and the body of the goal rule contains all (ground) atoms in the goal $G = \{g_1, \ldots, g_m\}$ of the task.

Note that each action schema a then produces $1 + |add(a)|$ rules. We assume all actions have at least one precondition and add a dummy precondition if this is not the case.

Helmert (2009) shows that the canonical model $\mathcal{M}$ of $\mathcal{D}_s$ contains exactly

(i) all ground atoms reachable in Π^+,

(ii) a ground atom a-applicable$(c_1, \ldots, c_m)$ if and only if the ground action $a(c_1, \ldots, c_m)$ is applicable in some reachable state of Π^+, and

(iii) the atom *goal* iff Π^+ is solvable, i.e., atoms in the goal G are reachable from the initial state I in Π^+.

If an atom is present in $\mathcal{M}$, then it is a *relaxed reachable* atom. If the atom *goal* in the head of the goal rule is relaxed reachable, then the task is *relaxed solvable*. Given a delete-relaxed task Π^+ with initial state I, we denote its associated Datalog program as $\mathcal{D}_I$.

relaxed reachable

relaxed solvable

Example 4.1 *Consider Example 2.4, from Chapter 2. The* delete-relaxation $\Pi^+ = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}^+, I, G \rangle$ *of this planning task is defined as*

$$\begin{aligned}
\mathcal{P} &= \{at/2, adj/2\} \\
\mathcal{C} &= \{a, b, c, t\} \\
\mathcal{A} &= \{drive^+(T, C_1, C_2)\} \\
I &= \{adj(a, b), adj(b, a), \\
&\quad\ adj(b, c), adj(c, b), \\
&\quad\ at(t, a)\} \\
G &= \{at(t, c)\}.
\end{aligned}$$

The action schema $drive^+(T, C_1, C_2)$ *has the following precondition and add list:*

$$\begin{aligned}
pre(drive^+(T, C_1, C_2)) &= \{at(T, C_1), adj(C_1, C_2)\} \\
add(drive^+(T, C_1, C_2)) &= \{at(T, C_2)\}
\end{aligned}$$

(It does not have a delete list as it is a delete-relaxed action.)

Algorithm 2 Computing $\mathcal{M}$ for a Datalog program $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$

```
1: queue := Queue(Atom)
2: M := ∅
3: for fact ∈ F do
4:     queue.Push(fact)
5: while not queue.Empty() do
6:     p := queue.Pop()
7:     if p ∉ M then
8:         M := M ∪ {p}
9:         for (head ← body) ∈ UnifyRules(p, M, R) do
10:            queue.Push(head)
11: return M

12: function UnifyRules(p, M, R)
13:     return {(head ← body) ∈ Ground(R) | p ∈ body, body ⊆ M}
```

Its relaxed reachability Datalog program $\mathcal{D}_I$ is the following program:

$$
\begin{aligned}
&adj(\mathrm{a},\mathrm{b}).\\
&adj(\mathrm{b},\mathrm{a}).\\
&adj(\mathrm{b},\mathrm{c}).\\
&adj(\mathrm{c},\mathrm{b}).\\
&at(\mathrm{t},\mathrm{a}).\\
&drive^{+}\text{-}applicable(T, C_1, C_2) \leftarrow at(T, C_1), adj(C_1, C_2).\\
&at(T, C_2) \leftarrow drive^{+}\text{-}applicable(T, C_1, C_2).\\
&goal \leftarrow at(\mathrm{t},\mathrm{c}).
\end{aligned}
$$

Helmert (2009) uses an algorithm similar to seminaive evaluation (see Chapter 2) on relaxed reachability Datalog programs to ground planning tasks. Algorithm 2 shows the pseudocode of Helmert's algorithm. As in the seminaive evaluation, Algorithm 2 builds the model $\mathcal{M}$ iteratively. It first adds all facts $\mathcal{F}$ to a queue of atoms named *queue*. Then, it iteratively removes atoms from the queue. If the removed atom a is not yet in $\mathcal{M}$, we add a to it. We then unify the bodies of all possible rules in $\mathcal{R}$ using a together with other atoms already in $\mathcal{M}$. For each unified rule, we add its head atom to the queue. The algorithm terminates once *queue* is empty. Instead of processing atoms in batches (atoms produced during the first iteration, during the second iteration, etc.) as in the seminaive evaluation, Algorithm 2 explores one atom at a time. At the end of its execution, $\mathcal{M}$ is the canonical model.

4.3 DATALOG-BASED HEURISTICS

We can adapt the lifted reachability analysis by Helmert (2009) to compute the h^{add}, $h^{\max}$, and h^{FF} heuristics. We start with h^{add} and $h^{\max}$.

Given a planning task Π, a state s, a relaxed reachability Datalog program $\mathcal{D}_s = \langle \mathcal{F}, \mathcal{R} \rangle$, we assign a *weight* $w(r)$ to each rule $r \in \mathcal{R}$. If r is a action applicability rule corresponding to action a, then $w(r) = \mathit{cost}(a)$. All other rules have weight 0. We call this extension a *weighted Datalog* program. For a given rule r, we write $w(r)$ on the $\leftarrow$ symbol to indicate its weight. For example, the following rule r_a

weighted Datalog

$$a\text{-applicable} \overset{10}{\leftarrow} q.$$

indicates that $w(r_a) = \mathit{cost}(a) = 10$. When a rule has weight of 0, we do not write it down on the $\leftarrow$ symbol.

We then consider a function v that is a maximal solution for the following equations:

$$v(p) = \begin{cases} 0, & \text{if } p \in \mathcal{F} \\ v(f^{\text{add}}(p)), & \text{otherwise,} \end{cases} \tag{4.7}$$

$$v(r) = w(r) + \sum_{q \in \mathit{body}(r)} v(q) \tag{4.8}$$

$$f^{\text{add}}(p) \in \operatorname*{arg\,min}_{\substack{r \in \mathit{Ground}(\mathcal{R}) \\ \mathit{head}(r) = p}} v(r). \tag{4.9}$$

Equations (4.7) and (4.8) are defined over $\mathcal{M}$ and equation (4.9) is defined over $\mathit{Ground}(\mathcal{R})$. This system of equations also has a unique maximal solution if all actions have costs larger than zero. Once more, in tasks with zero cost actions, these are considered to have a small cost of $\varepsilon > 0$.

The system of equations above is equivalent to Equations (4.1), but using Datalog program instead of planning tasks. All atoms p reachable in Π^+ have $v(p) = h(p, s)$: every atom $f \in \mathcal{F}$ has $v(f) = 0 = h(f, s)$; an atom p reachable in Π^+ but not in $\mathcal{F}$ is reachable via an effect rule, so

$$v(p) = v(f^{\text{add}}(p))$$

where $f^{\text{add}}(p)$ is an effect rule r_1 of the form[3]

$$r_1 = p \leftarrow a\text{-applicable}.$$

This implies that

$$\begin{aligned} v(p) &= v(f^{\text{add}}(p)) \\ &= w(r_1) + v(a\text{-applicable}). \end{aligned}$$

3 We omit the parameters of all atoms here, as they play no role in our argument, to simplify the presentation.

But $w(r_1) = 0$ (by definition) and $v(a\text{-applicable}) = v(f^{\text{add}}(a\text{-applicable}))$. As the only rules with predicate symbol a-applicable in the head are action applicability rules, $f^{\text{add}}(a\text{-applicable})$ must be an action applicability rule r_2 of the form

$$r_2 = a\text{-applicable} \overset{cost(a)}{\leftarrow} q_1, \ldots, q_n.,$$

therefore

$$\begin{aligned} v(p) &= v(f^{\text{add}}(p)) \\ &= w(r_1) + v(a\text{-applicable}) \\ &= 0 + v(f^{\text{add}}(a\text{-applicable})) \\ &= w(r_2) + v(q_1) + \cdots + v(q_n) \\ &= cost(a) + v(q_1) + \cdots + v(q_n) \qquad \text{[def. of rule weight]} \end{aligned}$$

which is the same as $h(p, s)$ in (4.1), when $q_1, \ldots, q_n$ are the preconditions of a.[4]

Then h^{add} over a lifted relaxed reachability program is equivalent to the following:

$$h^{\text{add}}(s) = v(goal). \tag{4.10}$$

The max heuristic $h^{\max}$ can be computed over the lifted relaxed reachability program by simply replacing the summation in Equation (4.8) with a max operator, as done in (4.4) and (4.5).

derivation

A *derivation* is a sequence of atoms from $\mathcal{F}$ and ground rules from $Ground(\mathcal{D}_s)$ where all atoms in the body of a rule either

- occur earlier in the derivation, or
- are heads of rules that appear earlier in the derivation.

An atom p in a derivation is a proof that $p \in \mathcal{M}$, while for a rule r, it proves that $head(r) \in \mathcal{M}$. We assume that each atom is derived at most once. If the last atom derived is p, we call the derivation a *derivation of* p.

An *achiever choice function* $f : \mathcal{M} \setminus \mathcal{F} \to Ground(\mathcal{R})$ maps atoms p to ground rules r with $head(r) = p$. Given an achiever choice function, we can construct a derivation of an atom by back-chaining through the atoms in achiever bodies. This back-chaining is similar to the extraction of the relaxed plan from h^{add} and h^{FF} in the ground case (Algorithm 1). For each atom in $\mathcal{M} \setminus \mathcal{F}$ that was not seen earlier, we recursively add its derivation to the start of the sequence. Equation (4.9) defines the f^{add} achiever choice function, which uses h^{add} values to choose the achiever of each atom. This is analogous to (4.3) for computing the

4 Note that $\mathcal{M}$ has more atoms than the set of reachable atoms in Π^+ — namely, atoms *goal* and of the form a-applicable. They are not necessary to argue that $v(p) = h(p, s)$ for relaxed reachable atoms p.

Algorithm 3 Computing v for a weighted Datalog program $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$

```
1:  V := DefaultHashTable(Atom, ℝ^∞, ∞)
2:  queue := PriorityQueue(Atom, ℝ^+)
3:  M := ∅
4:  for fact ∈ F do
5:      V[fact] := 0
6:      queue.Push(fact, 0)
7:  while not queue.Empty() do
8:      p := queue.PopMin()
9:      if p ∉ M then
10:         M := M ∪ {p}
11:         for (head ←^w body) ∈ UnifyRules(p, M, R) do
12:             cost := w + Σ_{q∈body} V[q]
13:             if cost < V[head] then
14:                 V[head] := cost
15:                 queue.Push(head, cost)
16: return V
```

best achiever on propositional tasks. Extending the terminology, we call the achiever $f^{\text{add}}(p)$ as the best achiever of p.

After computing h^{add}, we can extract a relaxed plan similarly to the ground planning case. Given the derivation of *goal*, we can create a plan composed by all the action applicability atoms in the derivation.

The question now is how to evaluate v. We can do so v while computing $\mathcal{M}$ using a modification of Algorithm 2. Algorithm 3 shows this modification. We store the v-value of each reached atom in a hash table V (line 1). The queue is ordered according to v (line 2). Whenever we generate a new atom p (line 11), its v-value is computed according to (4.7) (line 13). This order implicitly defines the achiever choice function f^{add} satisfying (4.9).

Algorithm 3 works as a generalized Dijkstra's algorithm, so we can use an *early stopping* approach: when we remove *goal* from the queue, the algorithm can stop, as $v(\mathit{goal})$ is already defined, and we can extract the heuristic value.

THE FF HEURISTIC

Algorithm 3 implicitly constructs a derivation with an achiever choice function that maps $\mathit{head}(r)$ to r. Note that the achiever choice function depends on how the queue breaks ties.

We can compute h^{FF} from $\mathcal{M}$ considering the derivation of *goal* for the achiever choice function f^{add} in (4.9). When back-chaining from *goal*, we collect all atoms P in its derivation. Let the set $\pi'_{\text{FF}} \subseteq P$ be the set of all action applicability atoms, i.e., atoms of the form

a-applicable($\boldsymbol{X}$). For the computed derivation, this set π'_{FF} matches the set of all actions a with parameters $\boldsymbol{X}$ for which a-applicable($\boldsymbol{X}$) is necessary to derive the goal. This is analogous to (4.6) but in terms of atoms, not ground actions. We can then compute h^{FF} for a state s as

$$h^{\text{FF}}(s) = \sum_{a\text{-applicable} \in \pi'_{\text{FF}}} cost(a). \tag{4.11}$$

4.4 PROBLEMS WITH OUR APPROACH

Unfortunately, we do not expect our approach to scale to larger problems. In some domains, even if we stop Algorithm 3 as soon as it derives the *goal* atom, the effort necessary to find this derivation is too large. Often, this approach is as expensive as grounding the entire task. Moreover, because we use it to evaluate heuristics and guide the search, we must repeat this procedure for every visited state.

A culprit of this problem are the action applicability atoms — i.e., a-applicable($\boldsymbol{T}$). They contain all the variables in the action, so their arities are much larger than the other predicates. Instantiating these atoms is equivalent to grounding actions.

We can simplify our relaxed reachability Datalog programs by removing the atoms a-applicable($\boldsymbol{T}$). To do so, we combine the action applicability rule

$$a\text{-applicable}(\boldsymbol{T}) \overset{cost(a)}{\leftarrow} q_1, \ldots, q_n$$

with the action effect rule

$$p \leftarrow a\text{-applicable}(\boldsymbol{T})$$

into a new *action rule*

$$p \overset{cost(a)}{\leftarrow} q_1, \ldots, q_n$$

for each add effect p of action schema a.

However, we now do not have enough information to compute Equation (4.11). Therefore, we need to find new ways of computing h^{FF} over the lifted representation. As a first approximation, we consider a heuristic that just adds the weights of all rules in a derivation of G. We call it the *rule-based FF heuristic* and denote it as $h^{\text{R-FF}}$.

rule-based FF heuristic

Where h^{FF} computes a relaxed plan, $h^{\text{R-FF}}$ computes a set of ground rules that are sufficient to derive the goal atom. The main difference between $h^{\text{R-FF}}$ and h^{FF} is that $h^{\text{R-FF}}$ might count the cost of the same action multiple times. This occurs when the action adds several atoms that are necessary to reach the goal.

Example 4.2 *Consider the following relaxed reachability Datalog program, in a state* $s = \{p(\mathrm{c})\}$*, and consider that* $cost(a) = 1$*:*

$$p(\mathrm{c}).$$
$$\textit{a-applicable}(X) \overset{1}{\leftarrow} p(X).$$
$$q(X) \leftarrow \textit{a-applicable}(X).$$
$$t(X) \leftarrow \textit{a-applicable}(X).$$
$$\textit{goal} \leftarrow q(\mathrm{c}), t(\mathrm{c}).$$

The value of the FF heuristic is $h^{\mathrm{FF}}(s) = 1$*.*

Once we simplify the program by removing the predicate a-applicable, we obtain the program

$$p(\mathrm{c}).$$
$$q(X) \overset{1}{\leftarrow} p(X).$$
$$t(X) \overset{1}{\leftarrow} p(X).$$
$$\textit{goal} \leftarrow q(\mathrm{c}), t(\mathrm{c}).$$

where $h^{\text{R-FF}}(s) = 2$*, because both rules — the one deriving* $q(\mathrm{c})$ *and the one deriving* $t(\mathrm{c})$ *— are counted separately.*

This is similar to h^{add} but h^{add} counts the cost of an action every time one of its effects is used whereas $h^{\text{R-FF}}$ counts the cost of an action at most once for each of its effects. Our hypothesis is that actions usually add very few atoms that are necessary to reach the goal and thus the values of $h^{\text{R-FF}}$ and h^{FF} are close.

An alternative way to think of $h^{\text{R-FF}}$ is using a task transformation that replaces every action schema a where $add(a) = \{p_1, \ldots, p_n\}$ with n new action schemas $a_1, \ldots, a_n$ where $add(a_i) = \{p_i\}$. For any state s in this transformation, $h^{\text{R-FF}}(s) = h^{\mathrm{FF}}(s)$.

The intention of $h^{\text{R-FF}}$ is to approximate h^+, so over-counting actions by treating their effects separately means that the heuristic loses accuracy. In the next section we introduce a general framework for associating a computation with a Datalog program. We will then show how h^{FF} can be computed in this framework without loss of accuracy, even without the action applicability atoms.

4.5 ANNOTATED DATALOG

An *annotated Datalog program* is a Datalog program where every fact $p \in \mathcal{F}$ and every rule $r \in \mathcal{R}$ is annotated with a sequence of *instructions* denoted $ann(p)$ and $ann(r)$. An instruction can refer to the variables used in the rule. It represents a (sequence of) commands to be executed. We assume that instructions allow for commands of traditional programming languages (e.g., C++), but we do not fully

instructions

Algorithm 4 Executing an annotated Datalog program.

```
1:  function BackChain(p)
2:      if visited[p] then
3:          return
4:      visited[p] := True
5:      if p ∈ F then
6:          Execute(ann(p))
7:      else
8:          for q ∈ Body(f^add(p)) do
9:              BackChain(q)
10:         Execute(ann(f^add(p)))
```

formalize it. When a rule with instruction $\mathfrak{I}$ is grounded with substitution $\sigma : \mathcal{V} \mapsto \mathcal{C}$, we consider the ground rule to have instruction $\sigma(\mathfrak{I})$.

The semantics of an annotated Datalog program $\mathcal{D}_s$ are relative to a derivation of an atom p. Recall that a derivation is a sequence over $\mathcal{F} \cup \mathit{Ground}(\mathcal{D})$. To *execute* a Datalog program for the derivation of p, we map each element of this sequence to its associated instruction and execute the instructions in this order.

Algorithm 4 shows how to find a derivation of an atom and execute the Datalog program for it. The idea is similar to Algorithm 1, used to extract relaxed plans using the best achievers. The back-chaining procedure in Algorithm 4 handles each atom at most once (lines 2–4). Facts in $\mathcal{F}$ need no further derivation, so their annotation is executed directly (lines 5–6). For other atoms, the algorithm ensures that their derivation is included and the corresponding instructions are executed before executing the instructions of the achiever (lines 8–10).

We demonstrate that annotated Datalog programs are useful by expressing different concepts in them. We base all examples on the Datalog program $\mathcal{D}_s$ and the achiever choice function f^{add}, as defined in (4.9). Other achiever choice functions, such as the one based on $h^{\max}$, are also possible but they can produce different results. In fact, the execution of the annotations could depend on the order that atoms are explored in Algorithm 4. The annotations we present next, however, are well-defined for a fixed achiever choice function, and do not depend on the exact order we execute them.

Useful Atoms

useful atom

Given a relaxed plan π^+, an atom is *useful* (Hoffmann and Nebel, 2001) if

1. it appears in the goal G, or
2. it is in the precondition of some action a in π^+, where a has another useful atom in its add list.

In other words, a useful atom is either a goal atom or an atom in the precondition of an action used to reach another useful atom.

preferred operator

A *preferred operator* (Helmert, 2006), also referred to as *helpful actions* in the literature (Hoffmann and Nebel, 2001), is a ground action having a useful atom in its add list.[5] These are considered particularly promising on a given state, typically because they are part of a relaxed plan or because they can make actions of this plan applicable. This information can be explored in different contexts. For example, a search algorithm might give priority to states generated via preferred operators (Richter and Helmert, 2009; Richter and Westphal, 2010).

In $\mathcal{D}_s$, useful atoms are those occurring in a derivation of the goal except for $\mathcal{F}$ and atoms of the form a-applicable$(\boldsymbol{X})$. To compute them with annotated Datalog programs, we use the annotation

$$ann(r) = [\texttt{Mark } head(r) \texttt{ as useful}]$$

for all action effect rules $r \in \mathcal{R}$, and an empty annotation for all action applicability rules and all $f \in \mathcal{F}$.

Given f^{add}, this annotation is well-defined. All useful atoms in the derivation are marked as such, independently of the order they are visited.

In our search algorithms later, we extract useful atoms and use it to select preferred actions in the successor generation step (Chapter 3). Whenever we generate the successors of a state, we check, for each applicable actions, if it has a useful atom in its add list and, if so, we mark it as preferred. There are other possible mechanisms to exploit useful atoms (Hoffmann and Nebel, 2001).

Rule-Based FF

The heuristic $h^{\text{R-FF}}$ can also be expressed with an annotated Datalog program. For each action a, we annotate its corresponding action effect rules r with

$$ann(r) = [\texttt{Add } cost\texttt{(a) to h}]$$

and use an empty annotation in all other cases. The variable `h` behaves as a global variable for all annotations, and it contains the value of $h^{\text{R-FF}}(s)$ after the execution. If the atom *goal* was derived, `h` is initialized as 0; otherwise as ∞.

For a fixed choice of the best achievers, as with f^{add}, this annotation is also well-defined: each action effect rule r in the derivation will add its corresponding cost to the global h-value. As the final value is not influenced by the order these additions occur, the annotation is well-defined and independent on the exact order we execute them.

5 In Fast Downward (Helmert, 2006), which we use later in our experiments, preferred operators are exactly those ground actions occurring in the relaxed plan found by h^{add}. In the FF planner (Hoffmann and Nebel, 2001), helpful actions are those that are selected in the first layer of the derivation. Thus, our definitions differ.

FF

To compute the FF heuristic using annotations, we construct the relaxed plan π_{FF} — see (4.6) and the definition of π_{FF} after it — as we back-chain from the derivation of *goal*. Here, we consider that π_{FF} is a global variable to all annotations.

For all actions a we annotate the corresponding action applicability rule $r = a\text{-applicable}(\boldsymbol{X}) \leftarrow q_1, \ldots, q_n$ with

$$ann(r) = [\texttt{Include } a(\boldsymbol{X}) \texttt{ in } \pi_{\text{FF}}]$$

and use empty annotations in all other cases. When executing an instruction for an atom p, if $p \notin s$, the instruction adds the achiever $a(\boldsymbol{X}) = a_p$ to π_{FF}. After the execution of p, the variable π_{FF} contains all actions of $\pi(p)$. The base case of $p \in s$ is trivial ($\pi(p) = \emptyset$). In the inductive step, the achiever choice of p is $A = f^{\text{add}}(p)$ which is added to π_{FF} by the annotation of the rule achieving p. The actions required to achieve preconditions of this action are already included in π_{FF} according to the induction hypothesis. As the derivation of p only relies on actions in $\pi(p)$, no additional actions are included in the set, so π_{FF} has the value of $\pi(goal)$ after the execution for goal.

This annotation is also independent of the order Algorithm 4 computes the derivation of G, assuming a fixed achiever choice function. Every action applicability rule r in the derivation is added to the set π_{FF} at some point, and adding elements to a set in different orders does not affect the final result.

4.6 TRANSFORMATIONS OF ANNOTATED DATALOG

We introduced rule-based FF specifically because removing action predicates is important for performance. Yet, all examples above use action predicates. To make use of this optimization, we now discuss how such optimizations can be done on any annotated Datalog program without changing its semantics. Moreover, we are not only limited to the removal of action predicates.

We introduce four generic transformations of annotated Datalog programs:

(i) rule merging,

(ii) rule decomposition,

(iii) predicate collapsing, and

(iv) variable renaming.

These transformations change the set of rules and predicate symbols used in the program which affects the derivations and thus the semantics of the program. However, we can still show that the semantics

before and after any transformation are equivalent in the following sense: for any derivation under an achiever choice function f in the transformed program, there is a derivation under an achiever choice function f' in the original program such that the execution of both programs under these derivations produces the same results. Moreover, in our use case, all atoms of the planning task that have a certain v-value under f have the same v-value under f'. This implies that any execution based on f^{add} achievers in the transformed program corresponds to an execution in the original program for one of the possible choices of f^{add} achievers.

Rule Merging

rule merging

Rules can be *merged* to eliminate intermediate atoms. This was the optimization we used when defining $h^{\text{R-FF}}$, where we removed atoms of the form a-applicable($\boldsymbol{X}$) from $\mathcal{D}_s$. For an atom p let $\mathcal{R}_p^+ \subseteq \mathcal{R}$ be the rules with head p and let $\mathcal{R}_p^- \subseteq \mathcal{R}$ be the rules where the body contains p. Rules can be merged if

- $\mathcal{R}_p^+$ and $\mathcal{R}_p^-$ do not overlap,
- there are no other rules or facts with the same predicate symbol as p,
- no rule has the atom *goal* in the head.

With some abuse of notation, the set of *merged rules* $\mathcal{R}'$ contains the rule

$$r^{\pm} = \mathit{head}(r^-) \leftarrow (\mathit{body}(r^-) \setminus \{p\}) \cup \mathit{body}(r^+)$$

for every combination of rules $r^+ \in \mathcal{R}_p^+$ and $r^- \in \mathcal{R}_p^-$.[6] The weight of the merged rule is $w(r^{\pm}) = w(r^+) + w(r^-)$ and its annotation is $\mathit{ann}(r^{\pm}) = [\mathit{ann}(r^+); \mathit{ann}(r^-)]$.

We can then replace $\mathcal{R}$ by $(\mathcal{R} \setminus (\mathcal{R}_p^+ \cup \mathcal{R}_p^-)) \cup \mathcal{R}'$.

Example 4.3 *Consider the following rules:*

$$r_1 = \mathit{a\text{-}applicable}(X, Y, Z) \overset{w}{\leftarrow} q_1(X), q_2(X, Y), q_3(Y, Z).$$
$$r_2 = p_1(Y, Z) \leftarrow \mathit{a\text{-}applicable}(X, Y, Z).$$
$$r_3 = p_2(Y, Z) \leftarrow \mathit{a\text{-}applicable}(X, Y, Z).$$

They can be replaced by the rules

$$r_{1,2} = p_1(Y, Z) \overset{w}{\leftarrow} q_1(X), q_2(X, Y), q_3(Y, Z).$$
$$r_{1,3} = p_2(Y, Z) \overset{w}{\leftarrow} q_1(X), q_2(X, Y), q_3(Y, Z).$$

6 There might be a collision of variable names when merging rules. This can be solved by first mapping $\mathit{vars}(p)$ to its corresponding names in r^-, and mapping all other variables in $\mathit{vars}(r^+) \setminus \mathit{vars}(p)$ to fresh names.

Assume we want to compute h^{FF} and useful atoms, then $ann(r_1)$ would be $[\texttt{Include } a(\boldsymbol{X}) \texttt{ in } \pi_{\text{FF}}]$ *and* $ann(r_i) = [\texttt{Mark } head(r_i) \texttt{ as useful}]$ *for* $i \in \{2, 3\}$*. In that case, we have*

$$ann(r_{1,i}) = [\texttt{Include } a(\boldsymbol{X}) \texttt{ in } \pi_{\text{FF}}\texttt{;}\\ \texttt{Mark } head(r_i) \texttt{ as useful}].$$

A derivation that uses a ground rule $r^- \in Ground(\mathcal{D}_s)$ must contain a ground rule $f(p) \in Ground(\mathcal{D}_s)$ to derive p, where $f(p)$ is the achiever of p. Those two rules can be replaced by their corresponding ground merged rule $r^{\pm}$ in the transformed program. Likewise, a merged rule in a derivation for the transformed program can be replaced by its components in the original program. The derived value, its v-value, and the sequence of executed instructions is the same.

In words, rule merging "shortcuts" some rules in the derivation: for every rule r with $p \in head(r)$ and every rule r' with $p \in body(r')$, we replace p in $body(r')$ with $body(r)$.

Rule Decomposition

rule decomposition

The *rule decomposition* transformation divides rules with large bodies into multiple smaller rules. The goal is to create an implicit join tree for the rules of the Datalog program (Helmert, 2009): we split the rule into smaller rules, while storing the intermediate results in a new predicate. These smaller rules are simpler to unify, and by decomposing them, we give guidance to our algorithm on which atoms to join. Let $r = p \leftarrow q_1, \ldots, q_n$ be a rule in an annotated Datalog program and let $1 \leq k \leq n$. Let $\boldsymbol{X}$ be the set of variables defined as

$$\boldsymbol{X} = \left(\bigcup_{k+1 \leq i \leq n} vars(q_i) \right) \cap \left(\bigcup_{1 \leq i \leq k} vars(q_i) \cup vars(p) \right).$$

By introducing a new predicate symbol aux, rule r can be split into two rules

$$r_1 = aux(\boldsymbol{X}) \leftarrow q_{k+1}, \ldots, q_n.$$
$$r_2 = p \leftarrow q_1, \ldots, q_k, aux(\boldsymbol{X}).$$

with $w(r_1) = 0$ and $w(r_2) = w(r)$. The annotation $ann(r_1)$ stores the values of the variables $\bigcup_{k+1 \leq i \leq n} vars(q_i)$ that r_1 depends on. The annotation $ann(r_2)$ is the same as $ann(r)$ but replacing those variables by the values stored by $ann(r_1)$.

Example 4.4 *Rule* $r_{1,2} = p_1(Y, Z) \leftarrow q_1(X), q_2(X, Y), q_3(Y, Z)$ *from Example 4.3 can be split into*

$$r_1 = aux(Y) \leftarrow q_1(X), q_2(X, Y),$$
$$r_2 = p_1(Y, Z) \leftarrow aux(Y), q_3(Y, Z)$$

If $r_{1,2}$ had the annotation $[\texttt{Add}\ a(X,Y,Z)\ \texttt{to}\ \pi_{\text{FF}}]$ the new rules would have the annotations

$$\begin{aligned} ann(r_1) &= [\texttt{Instantiation}[aux(Y)] = (X,Y)] \\ ann(r_2) &= [(\texttt{X},\texttt{Y}) = \texttt{Instantiation}[aux(Y)]; \\ &\qquad \texttt{Add}\ a(\texttt{X},\texttt{Y},Z)\ \texttt{to}\ \pi_{\text{FF}}]. \end{aligned}$$

Executing $ann(r)$ has the same effect as executing $ann(r_2)$ after $ann(r_1)$. As aux never occurs outside r_1 and r_2, any achiever choice function for $aux(\boldsymbol{X})$ must map to r_1 ground with $\boldsymbol{X}$. In a derivation, this rule will therefore occur before r_2 leading to the execution of $ann(r_1)$ before $ann(r_2)$. So, rule decomposition does not change the semantics of the annotated Datalog program.

Rule decomposition is a program rewriting technique (Ullman, 1988, 1989). There are different possibilities on how to choose the exact split (Bichler et al., 2016; Helmert, 2009; Morak and Woltran, 2012). This may reduce the computational effort to construct the canonical model $\mathcal{M}$ when using our algorithms described above. Assume $q_1(a)$ was just removed from the queue and we have to find a rule $r \in Ground(r_{1,2})$ such that $body(r) \subseteq \mathcal{M}$. If we first join $q_1(X)$ with $q_3(Y,Z)$ we could get a larger intermediate result than if we first join with $q_2(X,Y)$. In the split rules, only the efficient join is possible.

Predicate Collapsing

When the atoms of two predicate symbols p_1 and p_2 are reachable in the same ways throughout the Datalog program, we know that if $p_1(\boldsymbol{X})$ has a certain derivation, then $p_2(\boldsymbol{X})$ has the same derivation. In other words, the relations p_1 and p_2 are symmetric. In that case, we can replace $p_1(\boldsymbol{X})$ with $p_2(\boldsymbol{X})$ without changing the semantics of the Datalog program. Before we express this formally, we look into an example.

Example 4.5 *Consider a Datalog program with following rules:*

$$\begin{aligned} r_1 &= p_1(X) \leftarrow q(X,Y), t(Y). \\ r_2 &= p_2(X) \leftarrow q(X,Y), t(Y). \\ r_3 &= t(Z) \leftarrow q(Z,Y), p_1(Y). \\ r_4 &= t(Z) \leftarrow s(Z,Y), p_2(Y). \end{aligned}$$

where the weights and annotations of r_1 and $_2$ are identical. In this example, we can replace p_1 by p_2 without affecting the semantics:

$$\begin{aligned} r_1' &= p_2(X) \leftarrow q(X,Y), t(Y). \\ r_2 &= p_2(X) \leftarrow q(X,Y), t(Y). \\ r_3' &= t(Z) \leftarrow q(Z,Y), p_2(Y). \\ r_4 &= t(Z) \leftarrow s(Z,Y), p_2(Y). \end{aligned}$$

Note that since the rules of a Datalog program are a set, r'_1 and r_2 become identical, so the resulting Datalog program only contains three rules. Reducing the number of predicate symbols also reduces the size of $\mathcal{M}$, which might speed up the computation of the heuristics.

The optimized program has the following rules:

$$r_2 = p_2(X) \leftarrow q(X, Y), t(Y).$$
$$r'_3 = t(Z) \leftarrow q(Z, Y), p_2(Y).$$
$$r_4 = t(Z) \leftarrow s(Z, Y), p_2(Y).$$

predicate collapsing

Formally, a predicate p_1 and p_2 can be *collapsed* if for each rule

$$r_1 = p_1(\boldsymbol{X}) \leftarrow q_1, \ldots, q_n.$$

there is a rule

$$r_2 = p_2(\boldsymbol{X}) \leftarrow q_1, \ldots, q_n.$$

with $ann(r_1) = ann(r_2)$ and $w(r_1) = w(r_2)$, and vice-versa. Remember that every $p \in \mathcal{F}$ is equivalent to a rule with an empty body, i.e. $p \leftarrow \top$. If we replace p_1 by p_2, any derivation of G in the resulting Datalog program corresponds to a derivation in the original program and the v-values of all derived facts and rules remain the same.

Predicate collapsing is useful together with the rule decomposition techniques. If we have multiple rules that have similar decompositions (i.e., are decomposed into the same smaller rules), we can collapse all the new auxiliary predicates that were introduced during the decomposition.

Variable Renaming

variable renaming

Variable names used in a rule r have no impact on the canonical model or the back-chaining through a derivation. They can be *renamed* without changing the semantics of a Datalog program as long as the variables occurring in $ann(r)$ are renamed accordingly.

Similarly to predicate collapsing, variable renaming is useful when combined with other techniques. Variable renaming might make two rules identical,[7] so we can keep only one of them. When decomposing rules, there are cases where we can only collapse the auxiliary predicates if we rename the variables to something canonical.

4.7 EXPERIMENTAL RESULTS

We implemented the Datalog-based heuristics in Powerlifted. All our configurations use the successor generator $FR^{SJ,<}$, based on the full-reducer (Chapter 3).

We tested the following configurations:

7 To be identical, we also consider that their annotations must be the same.

(i) the lifted h^{add}, $h^{\text{R-FF}}$, and h^{FF} heuristics;

(ii) the lifted goal-count heuristic h^{gc} from Chapter 3;

(iii) the lifted h^{gc} heuristic using the *unary relaxation with disambiguation heuristic* $h^{\text{gc, ur-d}}$as a tie-breaker (Lauer et al., 2021). This is a relaxation that transforms any n-ary predicate $p(X_1, \ldots, X_n)$ into n different unary predicates $p^1(X_1), p^2(X_2), \ldots, p^n(X_n)$, unless they are static (i.e., do not appear in the effect of any action). This is the best configuration in the experimental results by Lauer et al.; and

(iv) the ground versions of $h^{\max}$ and h^{FF} from Fast Downward (Helmert, 2006).

In some of our experiments, we tested different search algorithms (both lifted and ground versions). First, for experiments with the admissible heuristic $h^{\max}$, we used the A* search algorithm. Second, we also use greedy best-first search (GBFS), as done in previous chapters. Third, we use a GBFS but with *deferred evaluation* (Helmert, 2006). Deferred evaluation algorithms do not evaluate a state when generated, but only when expanded. States are added to the open list with the h-value of their parent. We call *lazy GBFS* when it uses deferred evaluation, and *eager GBFS* when it does not. Deferred evaluation does not influence the completeness of GBFS, but it might save time by avoiding computing the h-value of some states. As our lifted heuristics might be expensive to compute, we expect it to pay-off.

deferred evaluation

lazy GBFS

eager GBFS

Richter and Helmert (2009) show that deferred evaluation paired with delete-relaxation heuristics can reduce the number of evaluations, but also that on its own, deferred evaluation can make the performance of the planner worse, as it makes the search less informed. Indeed, deferred evaluation works best when combined with additional mechanisms, such as preferred operators (Helmert, 2006; Hoffmann and Nebel, 2001). For example, the planner can use them to prune states not generated by preferred operators (at the price of making the search incomplete), or to prioritize such states when selecting the next state for expansion.

In our implementation of lazy GBFS, we use two open lists and alternate between them when selecting the next state for expansion. Both open lists are ordered by the same heuristic, but one of them — the *preferred list* — contains only states generated via preferred operators. Instead of alternating at each expansion (once the preferred list; once the non-preferred), we use a *boosted dual-queue* approach (Richter and Helmert, 2009): we keep a counter for each open list, initialized to 0. Whenever a state is removed from an open list, the priority of that list decreases by 1. At each iteration, the next state is removed from the list that has higher priority. When the search makes progress (i.e., whenever it finds a state with a lower h-value than any

Transformations		Action applicability rules	
		not merged	merged
Auxiliary Predicates	not collapsed	1072	1144
	not collapsed + VR	1069	1139
	collapsed	1088	1176
	collapsed + VR	**1100**	**1244**

Table 4.1: Coverage of h^{FF} with eager GBFS on both benchmark sets (1863 tasks) under different Datalog transformations. All runs include the rule-splitting transformation.

state before), the priority of the preferred list is "boosted" by adding 1000 to its counter.

We refer to the eager GBFS simply as "Eager", and to the lazy GBFS with the boosted dual-queue approach as "Lazy + PO".

Transformations

Our first experiment studies which transformation (Section 4.6) are beneficial to the planner's performance. To test this, we ran all combinations of rule merging, predicate collapsing and variable renaming (VR).

There are different ways to decompose rules. Here, we use the rule decomposition by Helmert (2009). His grounder requires all rules of the Datalog program to be in a specific form ensured by rule decomposition. Every rule of the Datalog program must either be a rule of the form

$$h(\boldsymbol{T}) \leftarrow b_1(\boldsymbol{T}_1), b_2(\boldsymbol{T}_2).$$

where $\mathit{vars}(\boldsymbol{T}) \subseteq (\mathit{vars}(\boldsymbol{T}_1) \cup \mathit{vars}(\boldsymbol{T}_2))$, or of the form

$$h(\boldsymbol{T}) \leftarrow b_1(\boldsymbol{T}_1).$$

where $\mathit{vars}(\boldsymbol{T}) \subset \mathit{vars}(\boldsymbol{T}_1)$. The first rule is called a join rule, and the second a projection rule. The decomposition procedure picks an original rule with more than two atoms in the body, and iteratively decomposes it by replacing one or two atoms (depending on whether the new decomposed rule is a join or a projection rule) with a new one — as in Example 4.4. We will revisit this rule decomposition procedure in greater detail in Chapter 6. In our implementation, we tried to be as similar as possible to Helmert's algorithm to better compare it with ground planners later. Thus, we cannot disable rule decomposition in our experiments, and we use the same method to decompose rules.

Table 4.1 shows the coverage of Eager GBFS with h^{FF} after different transformations. The conclusions are the same if we use other

heuristics or the Lazy + PO GBFS, so we do not detail these other results. All transformations are beneficial and in particular removing action predicates by merging rules is critical for performance. Using all transformations achieves the best performance, increasing the baseline coverage from 1072 to 1244.

Removing action predicates always increases the coverage by at least 70 tasks. However, merging rules can affect how ties in the achiever choice function are broken which affects the performance. The benefit of merged rules is thus not a clear dominance, and we saw a few IPC domains where coverage decreased (e.g., pipesworld-split, trucks). The variable renaming transformation is mainly useful in conjunction with other transformations as they create more rules that become identical with canonical variable names.

In the rest of this chapter, our experiments use the following transformations: we start by merging action applicability and action effect rules. We then use the rule decomposition by Helmert. In all resulting rules, we rename the variables to canonical names to maximize the number of predicate symbols the algorithm can collapse. We finally collapse auxiliary predicates introduced by the rule decomposition transformation where possible.

Admissible Datalog-Based Heuristics

To evaluate the performance of our only admissible heuristic, $h^{\max}$, we compare a lifted implementation of A* search with $h^{\max}$ with the breadth-first search (BFS) implemented in Chapter 3. To guarantee that the BFS returns an optimal solution, we set the cost of all action schemas to 1.

Table 4.2 shows the number of solved tasks for these two methods (under the header "Lifted"). We can see that A* with $h^{\max}$ performs worse than BFS, solving 12 tasks less in total. Despite the better coverage in the HTG set (+25 tasks solved), the improvements are local to some domains. The only two domains where A* with $h^{\max}$ has strictly better coverage are rovers-large and visitall-multidimensional. In the organic-synthesis domain, both methods have the same coverage, but they differ in which tasks are solved. The bad performance of A* happens because the computation of $h^{\max}$ is expensive, but the heuristic does not add much more information to the search. Therefore, A* with $h^{\max}$ has the computational overhead to calculate the heuristic, but does not save enough state expansions in return. This is illustrated in Figure 4.1, which shows the number of expansions per second for each method. (We only consider tasks that took more than 1 second to be solved.) In all instances solved by both methods, BFS expands at least 10 times more states per second than A* with $h^{\max}$. This indicates that indeed, using $h^{\max}$ is expensive but uninformed, and the planner is better off using a simple blind search.

	Lifted		Ground	
Coverage	BFS	$h^{\max}$	BFS	$h^{\max}$
IPC Sum (1001)	262	225	**340**	337
blocksworld-large (40)	0	0	**1**	**1**
childsnacks-large (144)	4	1	**9**	6
genome-edit-distance (312)	44	44	**48**	**48**
logistics-large (40)	5	5	**9**	5
organic-synthesis (56)	**44**	**44**	21	20
pipesworld-tankage (50)	12	8	**16**	10
rovers-large (40)	0	1	3	**5**
visitall-multidim. (180)	38	69	**72**	**72**
HTG Sum (862)	147	172	**179**	167
Total Sum (1863)	409	397	**519**	504

Table 4.2: Coverage of breadth-first search (BFS) and A* with $h^{\max}$ using and the lifted (using Powerlifted) and ground (using Fast Downward) representations.

A similar behavior occurs when comparing the ground BFS and the ground A* with $h^{\max}$ from Fast Downward (Table 4.2). Here, BFS is not only superior to A* in the IPC set (+8 tasks solved), but also in the HTG set (+12). This shows that even with a much faster computation of $h^{\max}$, the heuristic does not help the search. Comparing the lifted A* with $h^{\max}$ and its ground counterpart, the lifted version is superior only in the organic-synthesis domain. As explained in the previous chapter, grounding this domain is challenging and the optimal plans are rather short (less than 20 actions). So it pays off to use the lifted heuristic, as the search needed to find the plan is not so extensive. We also see that the gap between Powerlifted and Fast Downward in the visitall-multidimensional domain is much smaller than in the previous chapter. This happens because Fast Downward's grounder can only ground 72 tasks, so both BFS and A* with $h^{\max}$ have this upper limit in Fast Downward.

Non-Admissible Datalog-Based Heuristics

Our next experiment compares the performance of different non-admissible Datalog-based heuristics: h^{add}, $h^{\text{R-FF}}$, and h^{FF}. Table 4.3 show the coverage for these heuristics. We focus on the HTG set. With both Eager and Lazy + PO GBFS, coverage improves when switching from h^{add} to h^{FF}, but we see a larger improvement switching from Eager (which does not use preferred operators) to Lazy + PO. This confirms similar results in ground planning (Richter and Helmert,

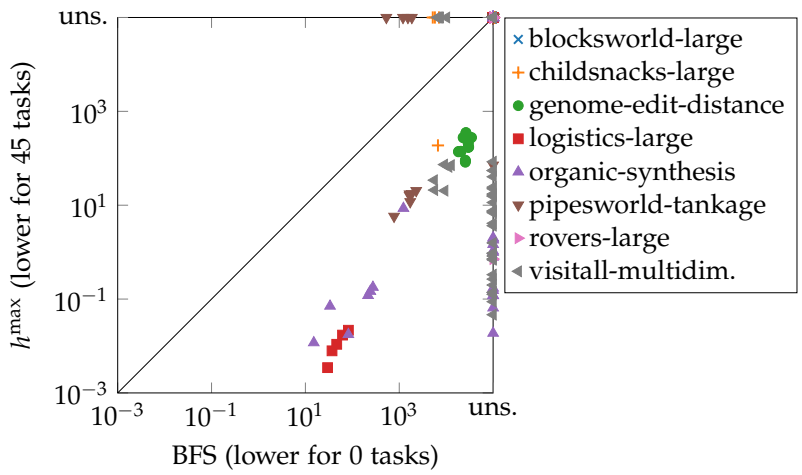


Figure 4.1: Expansions per second on the HTG set for lifted BFS and lifted A* with $h^{\max}$.

	Eager			Lazy + PO		
Coverage	h^{add}	h^{FF}	$h^{\text{R-FF}}$	h^{add}	h^{FF}	$h^{\text{R-FF}}$
IPC Sum (1001)	629	702	677	759	**820**	816
blocksworld-large (40)	1	4	0	6	**9**	4
childsnacks-large (144)	34	27	30	**82**	73	69
genome-edit-distance (312)	185	294	225	289	**311**	310
logistics-large (40)	8	9	9	**40**	**40**	**40**
organic-synthesis (56)	47	48	48	**49**	48	**49**
pipesworld-tankage (50)	22	23	25	28	**32**	**32**
rovers-large (40)	11	36	36	**40**	**40**	**40**
visitall-multidim. (180)	118	101	104	**143**	**143**	**143**
HTG Sum (862)	426	542	477	677	**696**	687
Total Sum (1863)	1055	1244	1154	1436	**1516**	1503

Table 4.3: Coverage of all configurations using non-admissible Datalog-based heuristics.

2009) showing that while h^{FF} is generally an improvement over h^{add}, using preferred operators impacts the search performance more. As with ground planning, the results for different domains vary and h^{add} sometimes gives better guidance than h^{FF} (e.g., h^{add} solves more tasks in childsnacks-large with both Eager and Lazy + PO configurations). As h^{add} is more greedy than the other methods, this is expected in tasks where greedy behavior leads to a plan more quickly.

With Eager GBFS, the lifted h^{FF} has an edge over the simpler rule-based FF heuristic $h^{\text{R-FF}}$, although $h^{\text{R-FF}}$ has superior coverage in a few domains (e.g., pipesworld-tankage, childsnacks-large). However, using

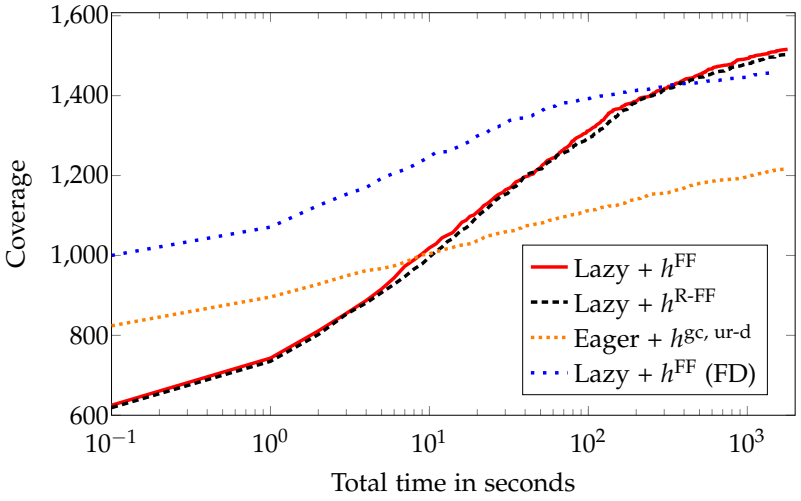


Figure 4.2: Solved tasks over time for different methods.

the Lazy + PO GBFS, h^{FF} and $h^{\text{R-FF}}$ have a similar performance: h^{FF} solves only 13 instances more than $h^{\text{R-FF}}$ in both sets, and only 9 more in the HTG set. In the HTG set, $h^{\text{R-FF}}$ solves more tasks than h^{FF} only in the organic-synthesis domain. In 4 domains of the IPC set, $h^{\text{R-FF}}$ has higher coverage than h^{FF}. But h^{FF}solves more tasks in 7 domains of this set and with larger differences in coverage (e.g., +11 tasks in openstacks).

Compared to h^{add} the overestimation done by $h^{\text{R-FF}}$ is limited by the maximal number of effects in an action. We hypothesized that usually not all effects of an action are required and thus the overestimation would be low. To test this, we measured the number of useful atoms per action in the relaxed plan found by $h^{\text{R-FF}}$ in the initial state. If this proportion p is 1 then $h^{\text{R-FF}}$ is equal to h^{FF} for this state. With higher values, the overestimation is larger. Among the 1863 tasks, 87.3% (1627) have $p \leq 5$, while only 10.5% (196) have $p \leq 2$. This shows that while the overestimation of $h^{\text{R-FF}}$ is still low, it is not as low as we initially expected. This overestimation does not necessarily influence the heuristic quality as scaling all heuristic values with the same factor has no effect in our search algorithms.

Other Lifted Planners

We also investigated how our methods compare to the other lifted heuristics in the literature that do not use Datalog. We ran experiments using h^{gc} (Chapter 3), and with the $h^{\text{gc, ur-d}}$ heuristic (Lauer et al., 2021) that break ties in h^{gc} based on the unary relaxation of the delete-relaxed task. Because of this further relaxation, it is currently not possible to extract a relaxed plan or preferred operators from these estimates. We thus use these heuristics with Eager search. Table 4.4 shows the results.

	Lifted Methods			Fast Downward	
	Lazy	Eager		Eager	Lazy
Coverage	h^{FF}	h^{gc}	$h^{\text{gc, ur-d}}$	h^{FF}	h^{FF}
IPC Sum (1001)	820	597	575	775	**862**
blocksworld-large (40)	9	4	7	4	**12**
childsnacks-large (144)	73	26	98	51	**115**
genome-edit-distance (312)	311	**312**	**312**	**312**	**312**
logistics-large (40)	**40**	20	0	30	32
organic-synthesis (56)	**48**	**48**	47	20	20
pipesworld-tankage (50)	**32**	22	10	15	19
rovers-large (40)	**40**	1	16	11	13
visitall-multidim. (180)	143	65	**151**	72	72
HTG Sum (862)	**696**	498	641	515	595
Total Sum (1863)	**1516**	1095	1216	1290	1457

Table 4.4: Coverage of our best method (Lazy + PO with h^{FF}), other lifted methods, and Fast Downward configurations. We write only "Lazy" for Lazy + PO configurations.

When comparing to our configurations using Eager search, h^{FF} is competitive with the methods based on h^{gc}. In total h^{FF} solves 28 tasks more than $h^{\text{gc, ur-d}}$. The advantage is mainly in the IPC domains where h^{FF} solves 127 tasks more than $h^{\text{gc, ur-d}}$. On the HTG domains, the picture is reversed and h^{FF} solves 99 tasks less than $h^{\text{gc, ur-d}}$(cf. Table 4.3).

However, all our methods using Lazy + PO are superior in coverage. All the Datalog-based heuristics are still better on the IPC set but also solve more tasks on the HTG set. As mentioned above, the preferred operators have more impact than a better heuristic and it is not known how to find preferred operators over the unary relaxation task efficiently.

We analyzed coverage over time for h^{FF}, $h^{\text{R-FF}}$, and $h^{\text{gc, ur-d}}$ in Figure 4.2. The search using $h^{\text{gc, ur-d}}$ has much higher coverage in approximately the first 10 seconds because it is fast to compute. Hence, tasks that do not require a deep search are solved quickly. In fact, the lifted h^{FF} and $h^{\text{R-FF}}$ computation are worst-case exponential in the PDDL-size, while $h^{\text{gc, ur-d}}$ is polynomial (Lauer et al., 2021). Indeed, for small tasks the overhead of computing h^{FF} or $h^{\text{R-FF}}$ does not pay off. For larger tasks, the stronger heuristic guidance is worth spending more time to compute h^{FF} and $h^{\text{R-FF}}$.

When analyzing memory, our methods are superior to $h^{\text{gc, ur-d}}$ even with small limits. Figure 4.3 compares coverage for different memory

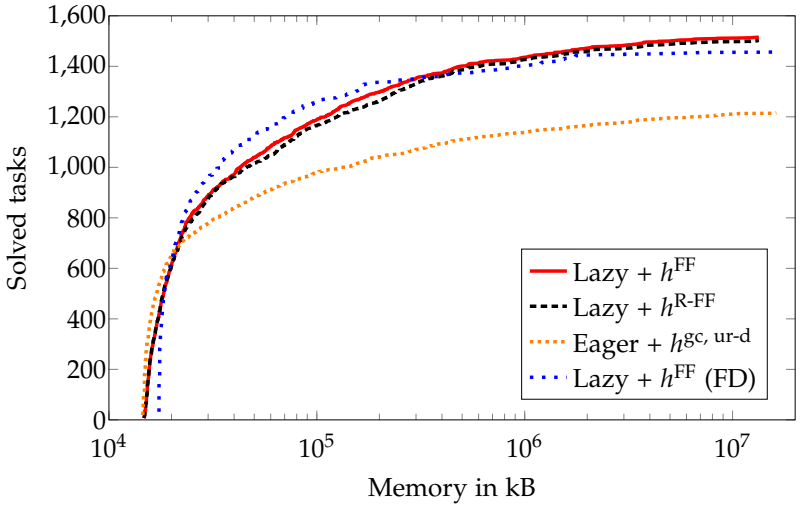


Figure 4.3: Solved tasks per second for different memory limits.

limits. Using a limit as low as 100 MiB, the difference in coverage between our methods and $h^{\text{gc, ur-d}}$ is already larger than 150 tasks.

Ground Planners

Our last experiment compares Powerlifted with the lifted h^{FF} heuristic to Fast Downward (Helmert, 2006) with the ground h^{FF} heuristic. Both use a generalized Dijkstra algorithm for their heuristic computation but the lifted implementation requires a more expensive unification step with this algorithm. Additionally, the two planners break ties in the heuristic computation differently, so they are not guaranteed to expand the same set of states. We thus treat this experiment as comparing two planners rather than comparing two implementations of h^{FF}. Table 4.4 shows the results for coverage.

Using Eager, Fast Downward outperforms Powerlifted in general. In the IPC set, it solves 73 more tasks in total. In 9 domains, Fast Downward solved 5 or more tasks more than Powerlifted. The largest difference was in barman, where Fast Downward solved 11 tasks more. The only domains where Powerlifted solves more tasks are visitall, parking-sat14, and thoughtful. This difference in performance is expected since the IPC tasks are easy to ground and the main challenge is the search itself. In such cases the advantage of not having to ground the task does not offset the more expensive heuristic computation in Powerlifted. When comparing the results in the HTG set, the planners are on par. Nonetheless, the coverage in some domains differs a lot, such as in logistics-large and organic-synthesis.

With Lazy + PO, the trend is similar: Fast Downward is superior on the IPC set, while Powerlifted is superior on the HTG set. However, on the IPC set, the advantage of Fast Downward reduces from 73 to 42 tasks. On the HTG set, the advantage of Powerlifted increases from

27 to 101 tasks. In total, Powerlifted has the highest coverage in this setting.

As Figure 4.2 shows, Fast Downward solves more tasks than the lifted methods in the first seconds. As most of the tasks in the combined benchmark are easy to ground and the ground heuristic computation is much faster, Fast Downward solves even more tasks than $h^{\text{gc, ur-d}}$ straight away. In fact, Powerlifted with h^{FF} only passes Fast Downward in number of solved tasks after around 400 seconds.

Comparing memory, Fast Downward has a similar performance to our methods. We can see in Figure 4.3 that with limits smaller than 1 GiB, Fast Downward has the highest coverage. For larger limits, the lifted planner is slightly superior.

4.8 SUMMARY

In this chapter, we showed how to compute different delete-relaxation directly from a lifted representation of a planning task. We showed how to compute three well-known heuristics: $h^{\max}$, h^{add}, and h^{FF}. Moreover, we introduced the new rule-based FF heuristic, $h^{\text{R-FF}}$, which treats the effects of an action independently and can thus overestimate h^{FF} and h^{add}.

Therefore, we introduced annotated Datalog programs, which associate every rule and fact with a sequence of instructions, called an annotation. After evaluating a Datalog query, the actions belonging to a relaxed plan can be extracted by executing the annotations along its derivation. We showed how such annotated Datalog programs can be simplified without changing their semantics. This allows us to compute the delete-relaxation heuristic from smaller annotated Datalog programs which are consequently faster to evaluate. We believe annotated Datalog programs can be used beyond the definition of heuristics, such as for generating landmarks (Zhu and Givan, 2003), as they allow us to express other computations over the relaxed task.

Empirically, in the non-admissible setting, our lifted planner Powerlifted using a GBFS with the lifted h^{FF} heuristic is competitive with its ground counterparts. The performance of Powerlifted is also improved when we include other search techniques, such as deferred evaluation and preferred operators (Helmert, 2006; Richter and Helmert, 2009). In domains that are hard to ground, this configuration solves more tasks than any other lifted method. In IPC domains, which are not particularly hard to ground, it reduces the gap to Fast Downward. Unfortunately, the same improvement does not occur in the admissible setting, where we are only interested in optimal solutions. There, A* search with the lifted $h^{\max}$ heuristic adds a significant overhead to the planner, while not adding much information to the search. This leads to a decrease in the number of solved tasks.

In the next chapter, we study other techniques that will help Powerlifted to solve even more tasks. These techniques will combine delete-relaxation heuristics with more recent search algorithms, such as best-first width search (Lipovetzky and Geffner, 2017).

CHAPTER NOTES AND HISTORY

Our work is not the first one to compute delete-relaxed heuristics in a lifted level. Related work on this problem can be split into three categories: planners computing an explicit relaxed planning graph (Blum and Furst, 1997), logic programming (McDermott, 1999), and homomorphisms (Horčík and Fišer, 2021).

In the first category, Ridder and Fox (2014) introduced the concept of lifted relaxed planning graphs (RPGs). To avoid the computational blow up when computing lifted RPGs, Ridder and Fox use almost-equivalence relationships between objects. Two objects are said almost-equivalent if they can instantiate the same parameters in the same predicates of the task. Although this speeds up the heuristic computation, it also decreases the heuristic quality. In fact, the planner implemented by Ridder and Fox performs consistently worse than more modern lifted planners, as demonstrated in the previous chapter (see Chapter 3).

Lauer et al. (2021) introduced the k-ary relaxation. In the k-ary relaxation of an atom $\mathrm{p}(V_1, \ldots, V_n)$, the atom is projected on $\binom{n}{k}$ new atoms, containing all combinations of $V_1, \ldots, V_n$ with k variables. A particular case is the unary relaxation (discussed above), where an n-ary atom is projected on n new unary atoms. The same relaxation is done for action schemas and states. The relaxation is also applied to the variables of the action. For a given state s, the planner computes a plan from the unary relaxation of s. The length of this relaxed plan is used as a heuristic estimate for the original state.

Computing a plan in the unary relaxed task can still take exponential time. However, it becomes tractable when we consider its delete relaxation. Lauer et al. (2021) proved that, for a delete-free unary relaxed task, we can compute a plan in polynomial time in its size. On the flip side, the heuristic is not much more informative than the goal-count heuristic. But while it does not help much as a single heuristic function, this unary-relaxation heuristic improves the search when used as tie-breaker.

Lauer et al. (2021) also show that while computing the k-ary relaxation for $k > 1$ is challenging, we can use a k-ary relaxation for a handful of atoms, and use unary relaxation for the remaining ones. The observation is that some atoms help the heuristic much more than others. For these "relevant" atoms, we want to keep their information intact. In our experiments, we compared our implementations to

$h^{\text{gc, ur-d}}$, which does not relaxed static atoms — those not affected by any action schema effect.

In the second category, McDermott (1996) introduced a planner using backward-chaining to count how many actions are needed to reach each goal atom of the task individually. This backward-chaining is done in a Prolog-like inference method. However, this method is not complete (McDermott, 1996).

The ideas from this chapter were introduced in two different papers (Corrêa et al., 2021, 2022). In the earlier paper, we showed how to compute the h^{add} heuristic and how to extract preferred operators using the back-chaining method. For this purpose, no annotations are needed, and we can simply assign a weight to each Datalog rule. In the second paper, we introduced the idea of annotated Datalog and also the rule-based FF heuristic, h^{FF}. Both works mention how to compute $h^{\max}$, but none of them presents experimental results for A* with $h^{\max}$. We implemented A* and $h^{\max}$ later to help the empirical comparison done by Horčík and Fišer (2021).

Datalog-based heuristics compute a lifted heuristic that is identical to its ground counterpart. A different approach is to use *homomorphisms* (Horčík and Fišer, 2021; Horčík et al., 2022). In this context, we are interested in homomorphisms between constants, so a homomorphism is a self-map $m\colon \mathcal{C} \mapsto \mathcal{C}$. The planner first computes a homomorphism between constants of the task. This homomorphism is used to reduce the number of objects. Then, the algorithm grounds the smaller task, and uses it to extract a heuristic estimate – computing the heuristic over the ground representation. For delete-free tasks, (optimal) plans are preserved (Horčík et al., 2022), which implies that admissible heuristics in the smaller ground task are also admissible in the original one. The almost-equivalence relation by Ridder and Fox (2014) can also be seen as a form of homomorphism.

Homomorphisms in lifted planning are similar to *domain-abstractions* in non-ground answer set programs (Saribatur et al., 2021). Both aim at reducing the set of constants by using self-maps, while over-approximating the set of solutions to their problems. It is still open how to translate the methods from answer set programming (e.g., domain-abstractions via CEGAR) to lifted planning.

5

LIFTED WIDTH SEARCH

In the previous chapters we studied how to construct an efficient lifted heuristic search planner. So far, we have implemented textbook search algorithms, such as GBFS and A*, and standard classical planning heuristics, such as h^{FF} and $h^{\max}$.

In this chapter, we show how to implement a different search algorithm: *best-first width search* (BFWS) (Lipovetzky and Geffner, 2017). Width search evaluates states based on their *novelty* (Lipovetzky and Geffner, 2012): the size of the smallest set of atoms that has not occurred in any previously evaluated state. The search then prioritizes states with smaller novelty values. In practice, planners bound the size of the set they check by some constant k, and if the evaluated state does not have a novel set of size k or less, the state has novelty $k+1$. Almost all planners in the literature limit k to 2 (e.g., Lipovetzky and Geffner, 2012).

Usually, it is very fast to evaluate the novelty of state. As a result, width search planners have been successful in recent IPCs satisficing and agile tracks (Corrêa et al., 2023c; Francès et al., 2018). Furthermore, these methods also excel in *simulation settings*, where the planner does not have access to an action model of the task (Francès et al., 2017). This is similar to the setting of lifted planning, where obtaining the ground actions can be expensive. This correspondence is a motivator for our study, since it suggests that there might be a synergy between lifted planning and width-based search.

The work in this chapter is mainly an engineering effort. While the concepts of novelty and width-search are easy to translate to the lifted setting, their commonly used data structures do not scale. In the ground setting — where all reachable atoms are known in advance —, the planner needs simple data structures (e.g., bitsets) to keep track of novel atoms. In contrast, implementing best-first width search algorithms in a lifted planner requires keeping track of the reachable atoms as they are discovered. By making this tracking efficient, we obtain a lifted planner that achieves state-of-the-art performance for our HTG set. Perhaps surprisingly, our lifted BFWS is superior to its ground counterpart for the IPC set.

Moreover, we also introduce a new method for combining BFWS with other heuristics using greedy best-first search with multiple open

list queues (Röger and Helmert, 2010). This adds an exploitative aspect to the exploration-focused BFWS, leading to higher coverage in some domains.

5.1 BEST-FIRST WIDTH SEARCH

best-first width search
novelty

Best-first width search (Lipovetzky and Geffner, 2017) is a search algorithm that uses *novelty* measures (Lipovetzky and Geffner, 2012) to select which states to expand next. The novelty $w(s)$ of a state s is the size of the smallest non-empty set of ground atoms Q such that s is the first state visited by the search where $Q \subseteq s$.

Example 5.1 *Assume that we have evaluated the following three states:*

$$s_1 = \{q, r\},$$
$$s_2 = \{r, t\},$$
$$s_3 = \{t, v\}.$$

The atoms seen already are $\{q, r, t, v\}$*. Then, the state*

$$s_4 = \{q, p\}$$

has novelty $w(s_4) = 1$*, because the smallest (non-empty) set of atoms in* s_4 *containing only atoms not seen yet is* $\{p\}$*, which has size* 1*.*

If we compute the novelty of the state

$$s_5 = \{r, t, v\}$$

then $w(s_5) = 2$*, because the atoms* r *and* v *have not occurred* together *in any previously visited state, and all individual atoms have already been seen.*

partition functions

The simplest BFWS variant prioritizes in the open list those states with minimal w-value. However, the strongest BFWS planners apply novelty measures based on *partition functions* of the search space (Francès et al., 2018, 2017; Lipovetzky and Geffner, 2017). The novelty $w_{\langle f_1,\ldots,f_n\rangle}(s)$ of a state s given functions $\langle f_1, \ldots, f_n\rangle$ is the size of the smallest set of atoms Q such that s is the first state visited where $Q \subseteq s$ among all states S where $f_i(s) = f_i(s')$ for $1 \leq i \leq n$ and for all $s' \in S$.

Example 5.2 *Assume that* h *is some arbitrary heuristic function, and let* $w_{\langle h\rangle}$ *be the novelty computed over the* h*-partition. Say also that we have two states:*

$$s_1 = \{q, r\},$$
$$s_2 = \{r, t\},$$

and $h(s_1) = 10$*,* $h(s_2) = 15$*. The state*

$$s_3 = \{q, t\}$$

with $h(s_3) = 10$ *has* $w_{\langle h \rangle}(s_3) = 1$, *because the set* $\{t\}$ *never occurred in a previously visited state that has the same h-value as* s_3. *This subset did occur in* s_2, *but* $h(s_2) \neq h(s_3)$.

In practice, planners only evaluate novelty up to a bound k, where usually $k = 2$. If a state s has no novel set of size k or less, then $w(s) = k + 1$.

An advantage of width search algorithms is that, *a priori*, the evaluation of w in a given state only depends on the state itself and the set of previously visited states. In other words, the evaluation of a state is *black-box* with respect to the structure of the problem (Francès et al., 2017). This makes width search an attractive option for hard-to-ground tasks, where it is expensive to obtain ground actions.

The main black-box BFWS algorithms use $w_{\langle \#r, \#g \rangle}(s)$ where the partition functions $\langle \#r, \#g \rangle$ define $\#r(s)$ as the number of *relevant atoms* that are true in s, and $\#g(s)$ as the number of goal atoms in G that are true in s. Choosing the set of relevant atoms is a parameter of the search. There are different approaches for this choice point. Next, we focus on the following two methods from the literature (Francès et al., 2017):

(i) BFWS(R_0), where the set of relevant atoms is the empty set; and

(ii) BFWS(R_X), where the set of relevant atoms is the set of useful atoms (Chapter 4) computed from a relaxed plan from s_0.[1]

We study BFWS(R_0) because it is the baseline version of width-based search with simulators and it does not require any knowledge about the action structures. The choice of BFWS(R_X) comes from Chapter 4, where we show how to extract a relaxed plan efficiently from the lifted representation.[2]

5.2 BALANCING EXPLORATION AND EXPLOITATION

exploration
exploitation

One important design choice of a planner is how it balances *exploration* and *exploitation*. Exploration focuses on parts of the state space that are distinct from previously visited ones; exploitation favors more promising parts of the state space. For example, choosing the next expanded state at random is a form of exploration, while choosing it based on a heuristic is exploitation. Modern planners aim at finding a good balance between both. In Chapter 4, we introduced two techniques for that: preferred operators and alternation of open lists.

Width search was originally introduced as a method to find a good exploration-exploitation balance (Lipovetzky and Geffner, 2017). But

1 Recall that an atom is useful for a state s if it appears in the precondition of an action of a relaxed plan from s.

2 We do not consider the other methods from Francès et al. (2017) in the lifted setting because they violate our assumption that we do not know all reachable atoms in advance (which is prohibitive due to the expensive grounding)

in some tasks, using only the novelty value of states can be misleading, since it only focuses on exploring unseen parts of the state space, without exploiting any information about the structure of the problem. We propose some ways to mitigate this issue next.

Dual-Queue BFWS

In Chapter 4, we showed that using a second open list (i.e., queue) in greedy best-first search helps to increase coverage. This second open list contains only atoms that were reached via preferred operators, while the first queue contains all atoms.

We extend this approach to BFWS. By definition, however, preferred operators are state-dependent. Thus, we need to extract a relaxed plan and the useful atoms for each state, which is equivalent to evaluating h^{add} (Bonet and Geffner, 2001) in every state. To avoid this overhead, we compute a relaxed plan only for the initial state I and consider the useful atoms of I as useful for every state in the search. This is a relaxation on the definition of useful atoms, but one of our hypotheses is it that this can still speed up the search.

From this relaxation of preferred operators, we build a dual-queue BFWS (DQ-BFWS). Assume that the set of relevant atoms of our search is R, the dual-queue BFWS, called *DQ-BFWS(R)* for short, works as follows. In the first queue Q_1, we insert all generated states. In the second queue Q_2, we insert only states reached via preferred operators. Both queues are ordered by $w_{\langle \#r, \#g \rangle}$, using R as the set of relevant atoms. We also use a boosting mechanism, as in Chapter 4. However, our boosting is slightly different: we associate a *priority value* $p_i \in \mathbb{Z}$ with each queue Q_i and let the search pick the queue with largest associated priority. The value of p_1 and p_2 are set to 0 initially. We reduce p_i by 1 every time we select a state from q_i, and we *boost* it by $C \in \mathbb{N}$ ($p := p + C$) whenever we expand a state s from it that contains more goal atoms than all states seen before s. The key difference to the boosting in Chapter 4 is that now we boost the queue based on the number of achieved goal atoms, instead of a heuristic value h.

DQ-BFWS(R)

Adding More Heuristics to BFWS

Another approach for adding goal-direction to a BFWS search is to alternate between open lists sorted by novelty measures and open lists sorted by heuristics. Katz et al. (2017) show that alternating between open lists (Röger and Helmert, 2010) using only heuristics based on novelty evaluators can be beneficial. In our work, we run BFWS and alternate between an open list ordered by $w_{\langle \#r, \#g \rangle}$ and an open list ordered by h^{add} or h^{FF}.[3] This goes in the opposite direction compared

3 We introduce the algorithm in terms of h^{add} but it is analogous for h^{FF}.

to DQ-BFWS: we have the overhead of computing h^{add} for every state, but by alternating with a novelty-based queue, we might achieve a better balance between exploration and exploitation and reduce the number of evaluated states. Our hypothesis for this approach is that the extra effort used to compute h^{add} will pay off by avoiding some expansions during search.

Since we always evaluate h^{add}, we can also extract useful atoms and preferred operators for every state without further overhead. We denote this version as *BFWS*$([R, h])$, where R is the set of relevant atoms and h is a heuristic function. For both evaluators R and h^{add}, BFWS$([R, h^{\text{add}}])$ also uses an open list variant that only contains states reached via preferred operators. Thus, BFWS$([R, h^{\text{add}}])$ has four open lists in total. It alternates between the open lists of R and h^{add} each time it selects a new state to be expanded. For a given evaluator, the search then decides between the regular open list or the open list with only states reached via preferred operators based on the counter C, as explained above.

BFWS$([R, h])$

5.3 IMPLEMENTATION

Any implementation of a width search needs to keep track of the set of reached atoms. For $k = 1$, the straightforward implementation is to keep a bitmap where each position corresponds to a ground atom P, and the corresponding bit is set to 1 if P has been reached. To compute the novelty of a state s, we simply check if all atoms in the state have their corresponding bit set to 1. If not, then $w(s) = 1$ and we update the bitmap accordingly.

To generalize the computation for larger k, we can create a bitmap of size $\binom{n}{k}$, where n is the total number of ground atoms, and each entry corresponds to a set of ground atoms of size k. The entry for set Q is set to 1 iff Q has been achieved. The evaluation and the update of the bitmap is analogous to the $k = 1$ case. For $w_{\langle \#r, \#g \rangle}$, the same idea still applies. The difference is that we need to create one bitmap for each $(\#r, \#g)$ pair.

Unfortunately, for hard-to-ground planning tasks, such an approach is usually infeasible because it requires computing all reachable atoms in advance, which is often too expensive. Even when this computation is feasible, the number of atoms is often too large and so creating bitmaps for all sets of size k adds too much overhead.

To avoid grounding, we propose an alternative implementation based on the state representation of Chapter 3. Remember that a state in the lifted representation is a set of relations. For each relation, we associate each tuple with an index, similarly as for the bitmap. This indexing is done on demand and an atom is indexed only once it is reached. We store a hash table that maps each reached atom to an index. To check if an atom p has been seen and indexed before, it

suffices to check if there is an entry in the hash table with key p. If not, we add the entry for p mapping it to a fresh index value.[4]

For $k = 1$, we evaluate $w(s)$ by checking if each atom in s has an entry in the hash table just described. For larger values of k, we keep track of the reached sets Q by storing a tuple with the indices of the atoms in Q in a set. In detail, to check if a tuple Q is reached for the first time, we first obtain the tuple Q' of all indices of atoms in Q. Then, we check if Q' is in the set and if not, we know that the state is novel and add Q' to the set.

Both the bitmap used in the ground version and the data structures used in our implementation (i.e., the hash table used for indexation, and the set of tuples of indices) have an amortized access time of $O(1)$. Since our data structures have far more overhead compared to the bitmap approach, it is important to use an efficient implementation. In our case, we tested different well-known C++ implementations of hash sets/tables: the built-in implementation from the standard library; the implementation from Abseil;[5] and the Parallel Hashmap library.[6] Abseil and Parallel Hashmap reimplement traditional data structures in C++, but they focus on performance. Parallel Hashmap is built on top of Abseil, but it removes some non-determinism present in the original implementation.

We also use a common optimization for width-based planners (Francès et al., 2017): if applying action A in state s yields state s', where $\#r(s) = \#r(s')$ and $\#g(s) = \#g(s')$, then we only consider sets that contain an atom in $add(A)$ to evaluate $w_{\langle \#r, \#g \rangle}(s')$.

5.4 EXPERIMENTS

We implemented the following width search algorithms in Powerlifted:

- BFWS(R_0);
- BFWS(R_X);
- DQ-BFWS(R_X);
- BFWS($[R_X, h]$), where h is a heuristic function given as parameter. We study specifically $[R_X, h^{\text{add}}]$ and $[R_X, h^{\text{FF}}]$.

We use $C = 1000$ for all our two configurations using boosting, DQ-BFWS(R_X) and BFWS($[R_X, h]$). We also tested all our configuration using $k = 1$ and $k = 2$.

4 This implementation is well-suited also for simulations where we do not know all reachable atoms in advance, or for problems where the set of constants (and hence atoms) can change during the plan (Chapter 7).

5 Available at: https://abseil.io/ (Accessed on June 27th, 2024)

6 Available at: https://github.com/greg7mdp/parallel-hashmap (Accessed on June 27th, 2024)

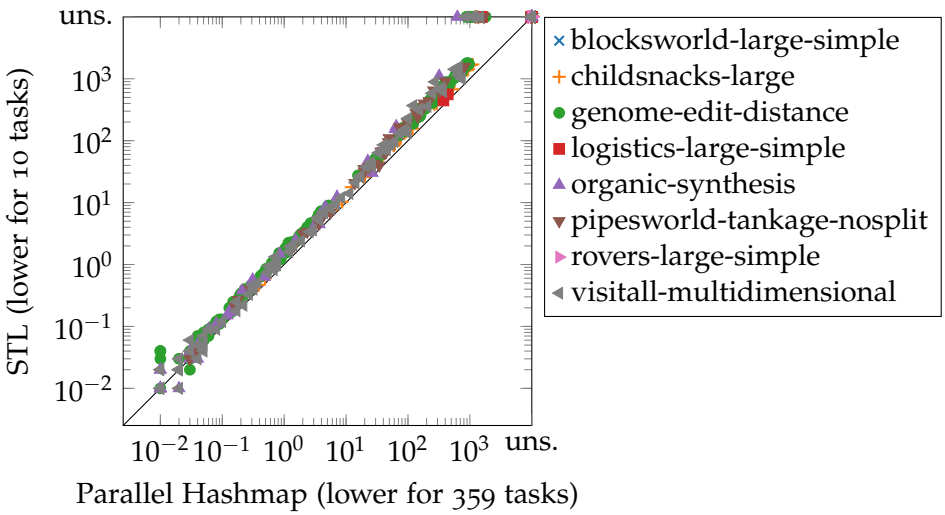


Figure 5.1: Search time for BFWS(R_0) using different implementations for the data structures storing previously seen tuples.

Efficient Hash Tables

For our first experiment, we tested different versions of BFWS(R_0), using different implementations of hash tables. Figure 5.1 shows the search time for the versions using the standard template library (STL) and Parallel Hashmap. Parallel Hashmap is consistently faster, and it even solves 57 instances that the STL version does not. Except for some simple tasks, Parallel Hashmap always produced better results. The Parallel Hashmap and the Abseil implementation performed similarly, but Parallel Hashmap has the benefit of being deterministic. With respect to memory, there is no visible difference between the implementations, as the memory usage is usually dominated by the large number of atoms that the hash tables contain. Therefore, we use the Parallel Hashmap implementation in our experiments.

Comparing Different Lifted Width-Search Algorithms

We also compare all lifted width search algorithms implemented in Powerlifted. Table 5.1 shows the results.[7] For each configuration, we tested one version using $k = 1$ and one using $k = 2$. The two best methods are BFWS($[R_X, h^{\text{add}}]$) and BFWS($[R_X, h^{\text{FF}}]$), both with $k = 1$. This indicates that the open list alternation provides a better exploration-exploitation balance than the other approaches. Indeed, we see that all three planners using multiple open lists yield a strong performance. DQ-BFWS(R_X) solves roughly as many tasks

7 We split the visitall-multidimensional domain into two because the FS-blind planner, that we later compare to, does not support some tasks in this domain.

	R_0		R_X		DQ(R_X)		$[R_X, h^{\text{add}}]$		$[R_X, h^{\text{FF}}]$	
value of k	1	2	1	2	1	2	1	2	1	2
IPC Sum (1001)	623	725	680	741	676	736	838	821	**857**	852
blocksworld-large (40)	7	6	7	5	8	3	**21**	12	19	11
childsnacks-large (144)	26	60	39	67	40	65	100	100	101	**102**
genome-edit-dist. (312)	306	307	**312**	**312**	**312**	**312**	309	310	309	311
logistics-large (40)	10	10	32	31	31	31	**40**	**40**	**40**	**40**
organic-synthesis (56)	49	48	39	49	49	49	**50**	49	**50**	49
pipesworld-tankage (50)	31	43	44	47	44	47	**48**	44	47	47
rovers-large (40)	0	0	3	1	2	1	**40**	23	**40**	28
visitall-3-and-4-dim. (120)	114	108	114	111	**116**	**116**	101	103	101	101
visitall-5-dim (60)	50	48	51	48	**51**	**51**	42	42	41	42
HTG Sum (862)	593	630	651	671	653	675	**751**	723	748	731
Total Sum (1863)	1216	1355	1331	1412	1329	1411	1589	1544	**1605**	1583

Table 5.1: Number of solved tasks by different lifted BFWS configurations and different values of k on the IPC (summarized) and HTG sets.

as BFWS(R_X), its single-queue counterpart, while BFWS($[R_X, h^{\text{add}}]$) and BFWS($[R_X, h^{\text{FF}}]$) solve roughly the same number of tasks across all domains. The two planners obtain a much higher total coverage than all other planners (751 and 748 tasks with $k = 1$), showing that it is indeed beneficial to make DQ-BFWS(R_X) more goal directed by combining it with h^{add} or h^{FF}. We also see that DQ-BFWS(R_X) is complementary to BFWS($[R_X, h^{\text{add}}]$): they obtain the highest coverage among all evaluated planners in two and six domains, respectively. In the two domains where DQ-BFWS(R_X) performs best (genome-edit-distance and visitall-multidimensional), h^{add} and h^{FF} seem to increase the computational effort without giving enough guidance in return — we observed a similar behavior in these two domains in Chapter 4.

To estimate the impact of the open list alternation in BFWS($[R_X, h^{\text{add}}]$) and BFWS($[R_X, h^{\text{FF}}]$), we compared the number of expansions of these methods and BFWS(R_X). As R_X is one of the open lists in BFWS($[R_X, h]$), this gives us an intuition of how much the other open lists affect the search. Figure 5.2 shows the number of expanded states for BFWS(R_x) and $[R_X, h^{\text{add}}]$ in the HTG set. In tasks solved by both, BFWS($[R_X, h^{\text{add}}]$) usually expands fewer states than BFWS(R_X). This shows us that while computing h^{add} in every state can be potentially expensive, the information extracted from it (heuristic value, preferred operators) does add a lot of information to the search, leading to both a reduction in the number of expanded states and an increase in coverage.

Table 5.1 also shows that increasing the value of k has less impact than switching the search algorithm. Most configurations benefit from a higher value of k, but this is not the case for BFWS($[R_X, h^{\text{add}}]$) and BFWS($[R_X, h^{\text{FF}}]$), which evaluate a heuristic function at every state. In these two configurations, using $k = 2$ consumes too much time,

	Ground	Lifted				
	R_0	R_0	R_X	DQ(R_X)	$[R_X, h^{\text{add}}]$	$[R_X, h^{\text{FF}}]$
IPC Sum (1001)	714	725	741	736	838	**857**
blocksworld-large (40)	0	6	5	3	**21**	19
childsnacks-large (144)	73	60	67	65	100	**101**
genome-edit-dist. (312)	**312**	307	**312**	**312**	309	309
logistics-large (40)	0	10	31	31	**40**	**40**
organic-synthesis (56)	0	48	49	49	**50**	**50**
pipesworld-tankage (50)	18	43	47	47	**48**	47
rovers-large (40)	**2**	0	1	1	**40**	**40**
visitall-3-and-4-dim. (120)	37	108	111	**116**	101	101
visitall-5-dim (60)	–	48	48	**51**	42	41
HTG Sum (862)	442	630	671	675	**751**	748
Total Sum (1863)	1156	1355	1412	1411	1589	**1605**

Table 5.2: Number of solved tasks by different planners using BFWS on the IPC (summarized) and HTG benchmark sets. For the ground version of R_0, we use the implementation from the FS-blind planner. As FS-blind does not support predicates with arity higher than 4, we split the visitall-multidimensional domain into two rows. Lifted implementations use their best corresponding value for k (see text).

but since the planner already spends time computing h^{add} or h^{FF}, this extra effort reduces the coverage. Using $k = 1$, on the other hand, seems to add enough exploration to the search while not being too expensive.

In the following experiments, we use the respective best value for k for each configuration. For BFWS(R_0), BFWS(R_X), and DQ(R_X) we use $k = 2$, while BFWS($[R_X, h^{\text{add}}]$) and BFWS($[R_X, h^{\text{FF}}]$) use $k = 1$.

Comparison to the Ground Implementation

Next we compare our lifted BFWS(R_0) implementation to the corresponding ground implementation. For the ground version, we use the FS-blind planner, which participated in the IPC 2018 (Francès et al., 2018, 2017). In both cases, we use $k = 2$. For the IPC set, the lifted implementation is on par with the ground version: the lifted version solves 725 tasks, while the ground version solves 714 tasks. We find these results remarkable, since our implementation is tailored for large tasks, and we expected the bitmap representation to be superior for the smaller problems. For the HTG set, the lifted version is preferable, solving more tasks than the ground version in 5 of the 8 commonly

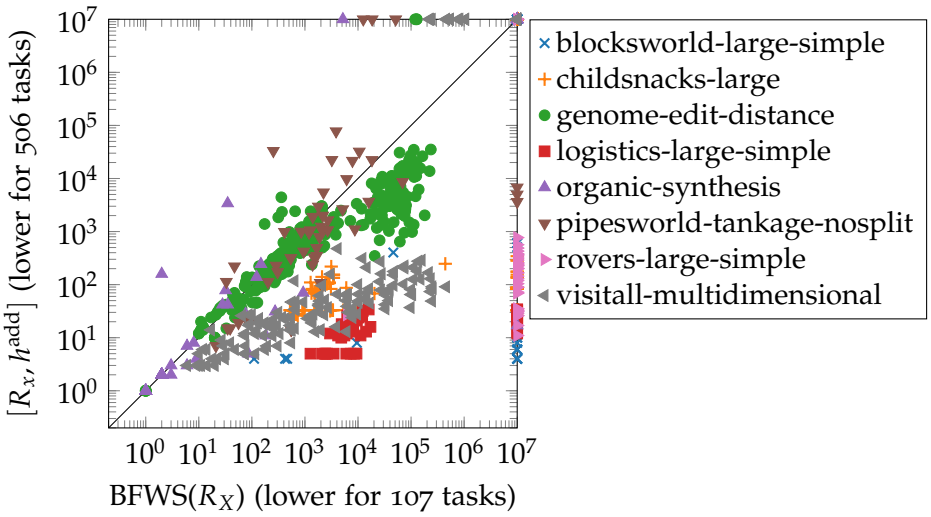


Figure 5.2: Expanded states for BFWS(R_X) and $[R_X, h^{\text{add}}]$ on the HTG set.

supported domains (see Table 5.2, which also includes our other lifted configurations).[8]

Comparison to Other Methods

We now compare the lifted implementations of the different algorithms described above to state-of-the-art ground and lifted planners, focusing on the HTG set.

As baselines for ground planners, we use

- LAMA (Richter and Westphal, 2010), which uses the FF heuristic and landmarks; [9]
- Dual-BFWS (Francès et al., 2018; Lipovetzky and Geffner, 2017), a state-of-the-art width search planner.

For lifted planners, we compare to three approaches:

- lazy GBFS with preferred operators using h^{FF} (Lazy + PO version from Chapter 4), denoted by L-h^{FF}; and
- the goal-count heuristic using the unary relaxation heuristic as tiebreaker — which is the best method from Lauer et al. (2021) and also used in Chapter 4 — denoted by L-$h^{\text{gc, ur-d}}$.

Table 5.3 shows overall coverage for these methods. We also include our two best lifted BFWS methods to help us with the comparison.

8 Since the FS planner does not support predicates with arity higher than 4, it cannot handle the visitall-5-dim instances.

9 LAMA is an anytime-planner, but we report results only for its first iteration, as the following searches cannot increase coverage but simply find better plans

	Baselines				Lifted BFWS	
	LAMA	Dual-BFWS	L-$h^{\text{gc, ur-d}}$	L-h^{FF}	$[R_X, h^{\text{add}}]$	$[R_X, h^{\text{FF}}]$
IPC Sum (1001)	917	**953**	575	821	838	857
blocksworld-large (40)	12	4	7	9	**21**	19
childsnacks-large (144)	**116**	109	98	72	100	101
genome-edit-dist. (312)	**312**	**312**	**312**	311	309	309
logistics-large (40)	36	4	0	**40**	**40**	**40**
organic-synthesis (56)	21	20	47	48	**50**	**50**
pipesworld-tankage (50)	18	18	10	27	**48**	47
rovers-large (40)	16	13	16	**40**	**40**	**40**
visitall-3-and-4-dim. (120)	60	36	100	98	**101**	**101**
visitall-5-dim (60)	12	6	**51**	42	42	41
HTG Sum (862)	603	522	641	687	**751**	748
Total Sum (1863)	1520	1475	1216	1508	1589	**1605**

Table 5.3: Number of solved tasks by different planners on the IPC (summarized) and HTG benchmark sets.

We first compare our best lifted BFWS, BFWS($[R_X, h^{\text{FF}}]$), with L-h^{FF}. This is similar to our previous comparison between BFWS(R_X) and BFWS($[R_X, h^{\text{add}}]$): BFWS($[R_X, h^{\text{add}}]$) has extra open lists ordered by the h^{add} heuristic when compared to BFWS(R_X); now when comparing L-$h^{\text{FF}}$and BFWS($[R_X, h^{\text{FF}}]$), the second has an extra open lists ordered by novelty value. Figure 5.3 compare their run times. In most cases, BFWS($[R_X, h^{\text{FF}}]$) is faster than L-h^{FF}. This shows that despite the extra work to keep more open lists and evaluate the novelty of every state, this does translate into better performance. This improvement further translates into coverage, where our two best lifted BFWS methods are superior to L-h^{FF} and L-$h^{\text{gc, ur-d}}$ in both IPC and HTG sets. Looking at individual domains, it is interesting to see that in six domains, BFWS($[R_X, h^{\text{FF}}]$) solves at least as many tasks as the stronger of the two ingredient planners (L-h^{FF} and BFWS(R_X)) for that domain.

However, adding exploration might cause the search to behave too greedily, which can lead to plans that are much longer than necessary. To evaluate if this happens for our lifted BFWS algorithms, we compared the plans found by L-h^{FF} and BFWS($[R_X, h^{\text{FF}}]$) on the HTG set. Figure 5.4 shows the results. In general, BFWS($[R_X, h^{\text{FF}}]$) found shorter plans than L-h^{FF}. This is unexpected, because the additional open lists used by BFWS($[R_X, h^{\text{FF}}]$) do not account for action costs or distance estimates.

Compared to state-of-the-art ground planners, BFWS($[R_X, h^{\text{add}}]$) and BFWS($[R_X, h^{\text{FF}}]$) are superior on the HTG set, but LAMA and Dual-BFWS are still better on the IPC set. Nonetheless, the gap between our best lifted configuration and LAMA decreased by 36 instances in the IPC benchmark, when compared to the gap between L-h^{FF} and LAMA.

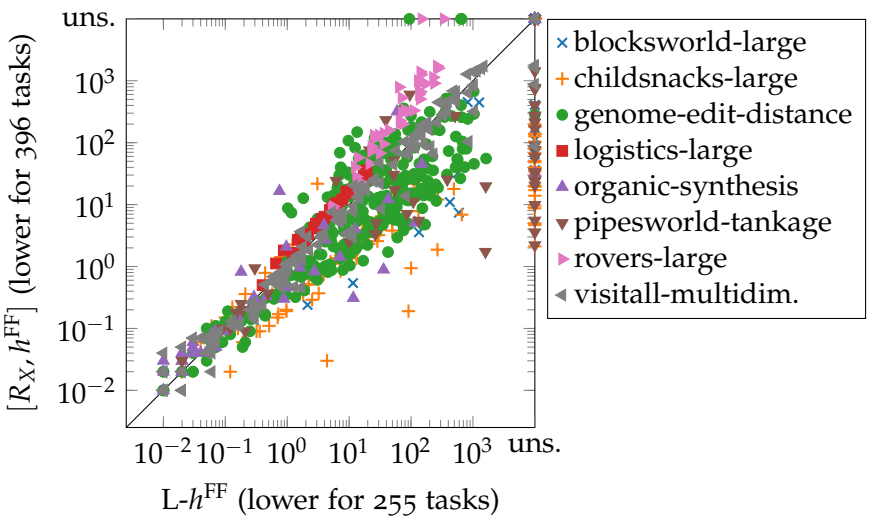


Figure 5.3: Run time for L-h^{FF} and BFWS($[R_X, h^{add}]$) on the HTG set.

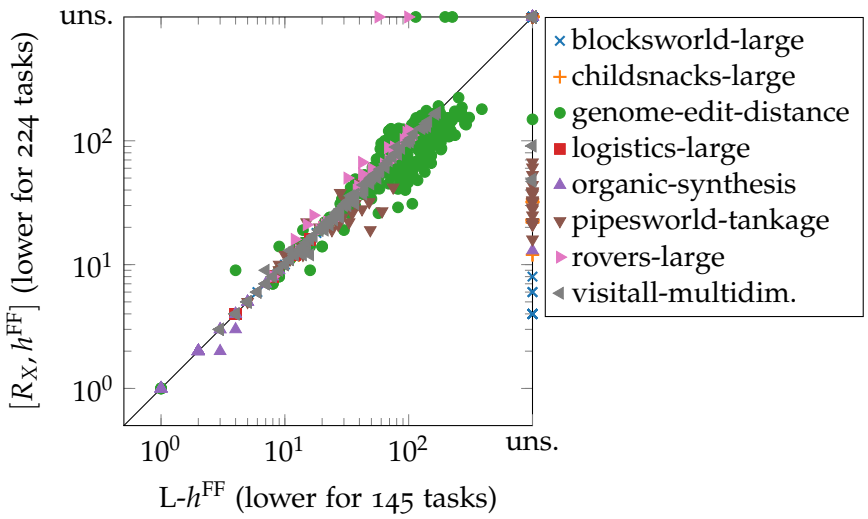


Figure 5.4: Plan length for L-h^{FF} and BFWS($[R_X, h^{add}]$) on the HTG set.

5.5 SUMMARY

In this chapter, we investigated how to implement BFWS in a lifted setting. Common data structures used by ground planners do not scale, but a re-implementation of BFWS, taking into account the lifted representation, reaches state-of-the-art performance. We also presented ways to make the search more informed by using different evaluators and preferred operators together with the novelty criteria. In this manner, we enhanced the exploratory behavior of BFWS with the exploitative behavior of the h^{add} and h^{FF}. In our experiments, we showed that the novelty measures have high synergy with both heuristics, increasing the number of solved tasks.

Our work is an initial study of BFWS with lifted implementations. There are other BFWS-based algorithms that can be implemented using our new representation (e.g., Katz et al., 2017). For more sophisticated BFWS variants, such as BFWS(f_6) (Lipovetzky and Geffner, 2017), one needs to adapt landmarks to the lifted setting (Wichlacz et al., 2021).

CHAPTER NOTES AND HISTORY

Balancing exploration and exploitation is a longstanding challenge in classical planning (Asai and Fukunaga, 2017; Fickert, 2018; Hoffmann and Nebel, 2001; Katz et al., 2017; Nakhost and Müller, 2009; Richter and Westphal, 2008; Vidal, 2011). In recent years, BFWS has emerged as the most successful approach for this problem (Francès et al., 2017; Lipovetzky and Geffner, 2012; Lipovetzky and Geffner, 2017).

Follow-up work combined BFWS with other search techniques. For example, Katz et al. (2017) combine the concept of novelty with heuristic estimates, extending the definition of novelty by Shleyfman et al. (2016) to take into account the heuristic value of the states. This allows them to quantify how novel a state is, so the search can be guided directly by this value. However, as this metric is not goal-aware, Katz et al. need to break ties using traditional goal-aware functions.

Another successful approach is due to Fickert (2020), who uses an orthogonal approach to the one by Katz et al.: instead of using novelty as the main guidance for the search, Fickert uses traditional heuristics to guide a greedy best-first search, and uses a lookahead strategy to find states with lower heuristic values quickly. This lookahead strategy is designed to reach states satisfying relaxed subgoals (Lipovetzky and Geffner, 2014). To make the procedure efficient, he uses novelty pruning (Fickert, 2018; Lipovetzky and Geffner, 2012) to reduce the number of evaluated states. Fickert shows that there is a synergy between the novelty-based lookahead and the h^{CFF} heuristic (Fickert and Hoffmann, 2017), as the result from the lookahead can be used to trigger the refinement procedure of h^{CFF}. This idea was also used in the OLCFF planner (Fickert and Hoffmann, 2018) from the IPC 2018.

The earlier versions of BFWS (e.g., Lipovetzky and Geffner, 2017), however, still outperform most of the recent novelty-based techniques (cf. Corrêa and Seipp, 2024).

All the algorithms and almost all the results presented in this chapter appeared in a previous publication (Corrêa and Seipp, 2022). However, we provided here a more detailed discussion of some experiments (e.g., different data structures; different values of k) that did not fit in the original work. For the IPC 2023, we extended BFWS($[R_X, h]$) to the ground setting, adding even more open lists and different estimates to it. Our Scorpion Maidu planner (Corrêa et al., 2023c), the winner of the satisficing track of IPC 2023, uses alternation of open lists also in combination with novelty estimates. Moreover, it uses

more sophisticated open lists, such as type-based queues (Xie et al., 2014), to add even more exploration to the planner. Scorpion Maidu also uses `gringo` (Gebser et al., 2011) as a grounder, instead of the traditional grounder from Fast Downward (Helmert, 2009). Grounding algorithms, in general, are discussed in Chapter 6.

Part II

PROPOSITIONAL PLANNING

6

GROUNDING PLANNING TASKS

Up to this point, we focused on lifted planning, where the planner works directly on top of a first-order representation. However, most of the classical planners (e.g., Bonet and Geffner, 2001; Francès et al., 2018; Helmert, 2006; Hoffmann and Nebel, 2001; Katz and Hoffmann, 2014; Torralba et al., 2014) do not use a lifted representation: as a first step, they *translate* the lifted task into a *propositional* one (e.g., Helmert, 2009; Köhler and Hoffmann, 2000). The new representation helps different parts of the planning algorithms, such as generating successor states (see Chapter 3), representing states (see Chapter 3), and computing heuristics (see Chapter 4). Although the translation can increase the size of the task exponentially, it is still worth doing for some domains.

One way to perform this translation — and perhaps the most commonly 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, as introduced in Chapter 4. Grounding this Datalog program overapproximates the ground actions and atoms that are reachable from the initial state of the task. This works fine in most cases because the Datalog programs are 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 (Dantsin et al., 2001; Immerman, 1986; Vardi, 1982). As planners improve, the tasks that users want to solve also become larger and harder (e.g., Haslum, 2011; Matloob and Soutchanski, 2016). For example, as we illustrated in the Part i, grounding tasks from our HTG set is far from trivial and can take more time than solving the task.

In this chapter, we act like Luís Figo switching from Barcelona to Real Madrid in 2000, and we also switch teams. Instead of continuing to translate successful ground techniques to the lifted setting, we do the opposite: we show how all the insights obtained when developing Powerlifted — decomposing Datalog rules, acyclicity, etc. — can be used to ground planning tasks more efficiently.

First, 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 rule decomposition 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 *tree decompositions* (Arnborg et al., 1987) and *grounding via solving* (Besin et al., 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 when combined with rule rewriting techniques using tree decompositions (Bichler et al., 2016; Morak and Woltran, 2012). Second, we obtain the set of ground actions from the result of the first step. To do so, we set up an ASP program 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 two-phase grounder can ground 10% more tasks than the method by Helmert in the HTG set. However, it also adds a considerable amount of 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. But we show that several of these tasks can still be solved with classical planners.

In this chapter, we introduce 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.

6.1 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.

In this chapter, we focus on the third step: grounding. Helmert introduced the lifted relaxed reachability (Chapter 4) to perform this

grounding. We revisit this idea here, and include more details that were not relevant in Chapter 4.

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 $\mathcal{D}_I = \langle \mathcal{F}, \mathcal{R} \rangle$ with facts $\mathcal{F} = I$ and the following rules:

1. For each $a \in \mathcal{A}$ with $pre(a) = \{q_1(\boldsymbol{T}_1), \ldots, q_n(\boldsymbol{T}_n)\}$ and $vars(a) = \boldsymbol{T}$, $\mathcal{R}$ contains the *action applicability rule*

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

   where *a-applicable* is called an *action predicate*.

2. For each $a \in \mathcal{A}$ with $vars(a) = \boldsymbol{T}$ and for each $p(\boldsymbol{T}') \in add(a)$, $\mathcal{R}$ contains the *action effect rule*

   $$p(\boldsymbol{T}') \leftarrow a\text{-}applicable(\boldsymbol{T}).$$

3. For $G = \{g_1(\boldsymbol{T_1}), \ldots, g_n(\boldsymbol{T_n})\}$, $\mathcal{R}$ contains the *goal rule*:

   $$goal \leftarrow g_1(\boldsymbol{T_1}), \ldots, g_n(\boldsymbol{T_n}).$$

The additional type predicates (Chapter 2) introduced in each action schema guarantee that all rules are safe. (This is also done by Helmert's algorithm.)

Recall from Chapter 4 that the canonical model $\mathcal{M}$ of $\mathcal{D}_I$ contains exactly the atoms that are reachable from I in Π^+. To compute $\mathcal{M}$, Helmert's algorithm modifies the seminaive evaluation (Chapter 2) by considering only one atom at each iteration.

The key is *how to compute the canonical model $\mathcal{M}$ efficiently*. Helmert decomposes large rules into smaller ones, as these are generally easier to ground and produce smaller intermediate results. The algorithm splits all rules until they have one or two atoms in their bodies. This allows for better 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

$$aux(\boldsymbol{T}) \leftarrow q_1(\boldsymbol{T}_1), q_2(\boldsymbol{T}_2).$$

where $\boldsymbol{T}$ contains all terms in $\boldsymbol{T}_1$ or $\boldsymbol{T}_2$ that occur in other atoms of r and the *auxiliary predicate aux* is a fresh predicate symbol. It then replaces $q_1(\boldsymbol{T}_1)$ and $q_2(\boldsymbol{T}_2)$ in r with $aux(\boldsymbol{T})$. This is called a *join rule* 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

$$aux(\boldsymbol{T} \setminus V) \leftarrow q(\boldsymbol{T}).$$

where *aux* is a fresh predicate symbol, and replaces $q(\boldsymbol{T})$ with $aux(\boldsymbol{t} \setminus V)$ in r. This is called a *projection rule*. Although projection rules do not decrease the size of the original rule, they project out irrelevant variables as early as possible.

Example 6.1 *Assume we want to decompose the following rule:*

$$a\text{-}applicable(X, Y, Z, A) \leftarrow q_1(X, Y, Z), q_2(Y, Z), q_3(X, A).$$

We can decompose it by creating a new rule with $q_1(X, Y, Z)$ and $q_2(Y, Z)$ in the body to guarantee these two atoms will be joined first:

$$a\text{-}applicable(X, Y, Z, A) \leftarrow aux(X, Y, Z), q_3(X, A).$$
$$aux(X, Y, Z) \leftarrow q_1(X, Y, Z), q_2(Y, Z).$$

What is left is how to choose atoms for the decomposition. Note that different choices on how to decompose rules lead to different performances. This is one of the main sources of overhead with rule decomposition.

To decompose rules, Helmert's algorithm follows two basic principles: project out unnecessary variables as early as possible, and join the maximum number of variables (with join decompositions).[1] This is done greedily based on the total number of variables and the number of joining variables in each atom. While the rule has more than two atoms in the body, we pick two atoms p and q according to the following sequence of rules (without loss of generality, assume that p has at least the same number of variables as q). Say that in our decomposed join rule, the joining variables in p and q are denoted by $\boldsymbol{X}$. Then, we use the following rules to chose p and q:

1. Prefer joins where $|vars(p)| - |\boldsymbol{X}|$ is largest.
2. If ties need to be broken, prefer joins where $|vars(q)| - |\boldsymbol{X}|$ is largest.
3. If ties still need to be broken, prefer joins where the $|\boldsymbol{X}|$ is lowest.

In practice, these rules enforce the algorithm to join atoms that share several variables, but where we can also project out many variables while doing the join. These rules are nondeterministic as several choices for p and q might lead to the same values. We assume an additional fixed deterministic tiebreaker (e.g., lexicographic order over the variable names) to remove this nondeterminism.

Helmert reports that, although his rule decomposition method works well in general, there are cases (such as the rovers domain) where this greedy decomposition is bad.

1 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 current implementation as it produced better results.

Experiments

Our first experiment compares the grounder by Helmert to `gringo` (Gebser et al., 2011, version 5.2.2), an off-the-shelf grounder. 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.

In the IPC set, all instances can be grounded by all tested methods. While there are some differences with respect to run time and memory consumption, they are minor in this set. The few domains that have some observable differences (e.g., rovers, logistics, visitall) also have larger counterparts in the HTG set, but the differences are clearer in the larger tasks. Therefore, to avoid an overload of information, we restrict most of our observations to the HTG set.

The current implementation of Helmert's algorithm used in Fast Downward 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 $\mathcal{D}_I$ as input.

The first three columns of Table 6.1 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 closer at 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 complicated (Chapter 3), 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. We reported a similar issue in rovers when studying lifted successor generators in Chapter 3.

We also analyzed the run time of all methods. Figure 6.1 shows the number of programs grounded over time. `FD`$^{++}$ is usually faster than `gringo` in tasks that both can ground. This is expected, as `gringo` is a general grounder 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 ground atoms and intermediate joins, which consumes too much memory. To test this, we evaluated giving `gringo` the program $\mathcal{D}_I$ *after* the decompositions used by the `FD` algorithm. The

Domain	FD	FD^{++}	G	G+FD	G+L
IPC Sum (1001)	**1001**	**1001**	**1001**	**1001**	**1001**
blocksworld-large (40)	**40**	**40**	**40**	**40**	**40**
childsnack-large (144)	**130**	**130**	**130**	**130**	**130**
genome-edit-dist. (312)	**312**	**312**	**312**	**312**	**312**
logistics-large (40)	**40**	**40**	**40**	**40**	**40**
organic-synthesis (56)	**21**	**21**	**21**	**21**	**21**
pipesworld-tankage (50)	**42**	**42**	**42**	**42**	**42**
rovers-large (40)	17	21	**40**	22	**40**
visitall-multidim. (180)	150	150	**174**	168	**174**
HTG Sum (862)	752	756	**799**	775	**799**

Table 6.1: Number of grounded tasks by domain for different algorithms. We abbreviate `gringo` with `G`, and `lpopt` with `L`.

results are also shown in Table 6.1 as "G+FD". This indeed decreased the number of programs `gringo` could ground to 775, bringing it 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 one task more than FD^{++}, while it grounded 43 more tasks without the decomposition. However, G+FD still outperforms FD^{++}, being superior in rovers and visitall-multidimensional. This indicates that the superior performance of `gringo` is not only due to the different input, and the state-of-the-art techniques in `gringo` play a larger role than the rule decomposition of FD^{++}.

6.2 A FIRST DETOUR: TREE DECOMPOSITIONS

tree decomposition

A *tree decomposition* (Arnborg et al., 1987) of a graph $G = \langle V, E \rangle$ is a tuple $\langle T, \chi \rangle$ consisting of a tree $T = \langle N, F \rangle$ and a function $\chi : N \to 2^V$ mapping tree nodes to sets of graph vertices called *bags*. The function has to satisfy that

bags

1. for each edge $\{v_1, v_2\} \in E$ in the graph G, there exists a node $n \in N$ in the tree T such that $\{v_1, v_2\} \subseteq \chi(n)$,
2. for every vertex $v \in V$ in the graph G, there exists a node $t \in N$ in the tree T with $v \in \chi(T)$, and
3. for every vertex $v \in V$ in the graph G, the set $\{n \in N \mid v \in \chi(n)\}$ of nodes in T with v in the bag induces a connected subtree of T.

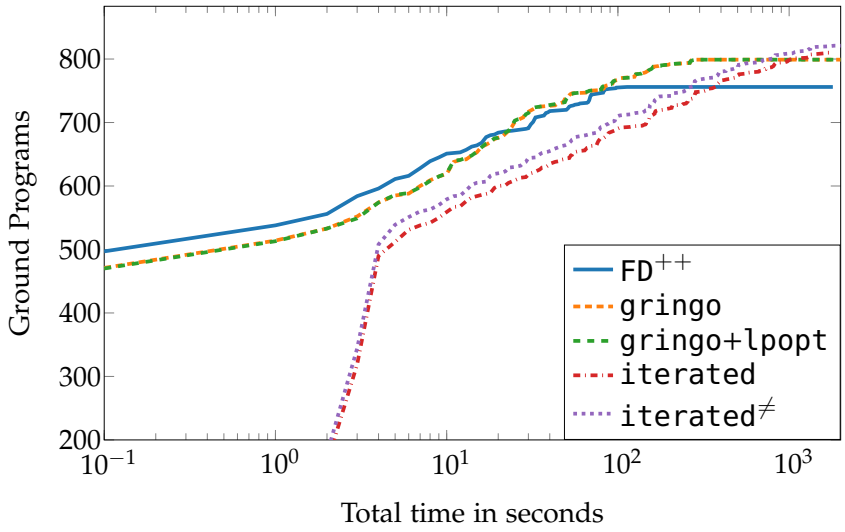


Figure 6.1: Number of tasks ground per time (in seconds).

The *width* of a tree decomposition is $w - 1$, where w is the size of the largest bag. The *treewidth* $tw(G)$ of G is the minimum treewidth among all tree decompositions of G. Computing the treewidth of a graph is **NP**-hard (Arnborg et al., 1987). *treewidth*

Given a Datalog program $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$, we can associate each rule $r \in \mathcal{R}$ with a graph $G_r = \langle V_r, E_r \rangle$, where $V_r = vars(r)$ and there is an edge $(v_1, v_2) \in E_r$ iff variables v_1 and v_2 occur together in an atom of the rule, i.e., $v_1, v_2 \in vars(\boldsymbol{T})$ for some $p(\boldsymbol{T}) \in head(r) \cup body(r)$. The treewidth of a rule r is the treewidth $tw(G_r)$.

Tree decompositions are similar to join trees (Chapter 3). But while in our join trees for conjunctive queries the nodes correspond to atoms, in a tree decomposition each node corresponds to a set of variables (the bags) that might cover variables in more than one atom.

6.3 GROUNDING USING STRUCTURAL DECOMPOSITIONS

Ideally, we want 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 by decomposing the rules of a Datalog program based on their tree decompositions. This 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.

For a given rule r, the algorithm by Morak and Woltran first computes a tree decomposition $T = \langle N, F \rangle$. For each node $n \in N$ with

parent node p_n, it then creates a fresh predicate $temp_n$ and introduces a rule

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

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

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

where *root* is the root node of T. These rules produce the same instantiations of $head(r)$ as the original one.

The objective here is no different from the rule decomposition technique discussed in Chapter 4 or the decomposition performed by Helmert (2009). Comparatively, the only difference is *how exactly* the rules are decomposed. The correctness of this rule decomposition technique follows from the same reasons given in Chapter 4.

Experiments

We use `lpopt` (Bichler et al., 2016), an implementation of the technique described above, to evaluate more systematic decompositions. The tree decomposition computed by `lpopt` has no optimality guarantee.

Table 6.2 presents statistical data on the treewidth *tw* 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 are discussed in the next section. Some domains (e.g., organic-synthesis and pipesworld) have rules with high treewidth *tw*, 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 *tw*.

Table 6.1 shows the number of ground tasks in our benchmark set using `gringo+lpopt` (`G+L`). This technique grounds exactly the same tasks as `gringo` without `lpopt`, and looking at the number of ground tasks over time in Figure 6.1 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, due to the action predicates. Therefore, `lpopt` cannot project out any variables during the decomposition, so intermediate predicates end up being too large. Note that this happens in all domains, and not only in domains that have large treewidth. Moreover, grounding these large action predicates requires a lot of memory. In fact, in all instances where `gringo+lpopt` (and also simply `gringo`) fails, it runs out of memory.

Domain	A	tw-range	average tw
blocksworld-large	✓	1–2	1.54 ± 0.52
	✗	1–2	1.33 ± 0.50
childsnack-large	✓	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-large	✓	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-large	✓	2–6	4.23 ± 1.27
	✗	2–3	2.35 ± 0.49
visitall-multidim.	✓	4–6	5.17 ± 0.83
	✗	4–6	5.17 ± 0.83

Table 6.2: Treewidth tw of the tree decomposition computed by `lpopt`. Column A indicates whether action predicates are considered. In the last column, entry $X \pm Y$ indicates an average of X and standard deviation of Y. The visitall-multidimensional domain contains different domain files with different action schemas. For this domain, we compute the average (and std. deviation) across all domain files.

6.4 AVOIDING TO GROUND ACTIONS

What happens then if we remove these action predicates from our Datalog programs? In Chapter 4 we saw that removing action predicates when computing delete-relaxation heuristics had a significant impact in the performance of Powerlifted. Our aim is to do the same for grounding.

Recall that given an action applicability rule and an effect rule such as

$$a\text{-applicable}(\boldsymbol{T}) \leftarrow q_1(\boldsymbol{T}_1), \ldots, q_n(\boldsymbol{T}_n).$$
$$p(\boldsymbol{T}') \leftarrow a\text{-applicable}(\boldsymbol{T}).$$

Domain	FD^{++}	G	G+FD	G+L
IPC Sum (1001)	**1001**	**1001**	**1001**	**1001**
blocksworld (40)	**40**	**40**	**40**	**40**
childsnack (144)	**144**	**144**	**144**	**144**
genome-edit-dist. (312)	**312**	**312**	**312**	**312**
logistics (40)	**40**	**40**	**40**	**40**
organic-synthesis (56)	**56**	41	**56**	**56**
pipesworld-tankage (50)	**50**	**50**	**50**	**50**
rovers (40)	**40**	**40**	**40**	**40**
visitall-multidim. (180)	150	**180**	168	**180**
HTG Sum (862)	832	847	850	**862**

Table 6.3: Number of ground simplified program for different algorithms, without action predicates. We abbreviate `gringo` with `G`, and `lpopt` with `L`.

we can replace both rules with the merged rule[2]

$$p(\boldsymbol{T}') \leftarrow q_1(\boldsymbol{T}_1), \ldots, q_n(\boldsymbol{T}_n).$$

simplified (Datalog) program

In the rest of the chapter, we refer to the Datalog program without action predicates as the *simplified (Datalog) program*. This simplification improves the performance of lifted planners (Chapter 4) which do not need the action predicates. However, we cannot know all relaxed-reachable ground actions of our task without these predicates. At first sight, this is an important drawback, because ground planners *must know all the actions* in advance. Otherwise, we are just back to the lifted planning setting, where the ground actions must be decided at every state. We give a solution to this problem later, but we first experimentally verify the simplicity of these programs.

Experiments

We compared all previous methods again but now using the simplified program as input. Table 6.3 shows the number of ground tasks for each method. The increase in performance is clear: FD^{++} goes from 756 to 832 ground programs; `gringo` goes from 799 to 847; `gringo`+FD^{++} goes from 775 to 850; `gringo+lpopt` goes from 799 to all 862 tasks. This is due to the simpler structure of the programs. Table 6.2 shows that the average treewidth of the decompositions found by `lpopt` for the programs with (column A with ✓) and without (column A with ✗) action predicates. The decompositions found for the latter have much

2 Note that if a has multiple action effect rules, we repeat the procedure for each one of them.

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 6.3. 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-multidimensional domain. The greedy decomposition used in `FD`$^{++}$ does not seem to work well with the larger predicates in visitall-multidimensional. `gringo` fails for some larger organic-synthesis problems, because some of these tasks have large intermediate relations if the rule bodies are joined in a bad order. On the other hand, `gringo+lpopt` can overcome both problems. The tree decomposition helps `gringo` 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+FD`$^{++}$ also solves the problem with the organic-synthesis domain, as it seems that without the action predicates the decomposition of `FD`$^{++}$ works well.

Moreover, `gringo+lpopt` is faster than all other methods even for tasks that all can ground. Figure 6.2 shows the run time for both `gringo` and `gringo+lpopt`. Clearly, `lpopt` improves the performance of `gringo`. Concentrating on tasks that took longer than 1 second to be ground, the only tasks above the diagonal are from the domain logistics. The tree decomposition found by `lpopt` seems to do more harm than good here. Logistics has simple rules, so we believe that this is another case where the size of the relations and selectivity might have more impact than decompositions. The domains where both methods have an almost identical performance are blocksworld and visitall-multidimensional, where the simplified programs do not have a large reduction (or no reduction at all) in the treewidth of the rules.

6.5 A SECOND DETOUR: ANSWER SET PROGRAMMING

Until now, whenever we discussed logic programs, we focused exclusively on Datalog programs. We now extend our discussion to *answer set programming*. First, recall that we first defined rules using the following format (Chapter 2):

answer set programming

$$h_1 \vee \ldots \vee h_k \leftarrow p_1, \ldots, p_j, \neg n_1, \ldots, \neg n_m.$$

Once we allow for disjunction in the head and negated atoms (i.e., atoms appearing under negation) in the body, our notion of canonical model is lost.

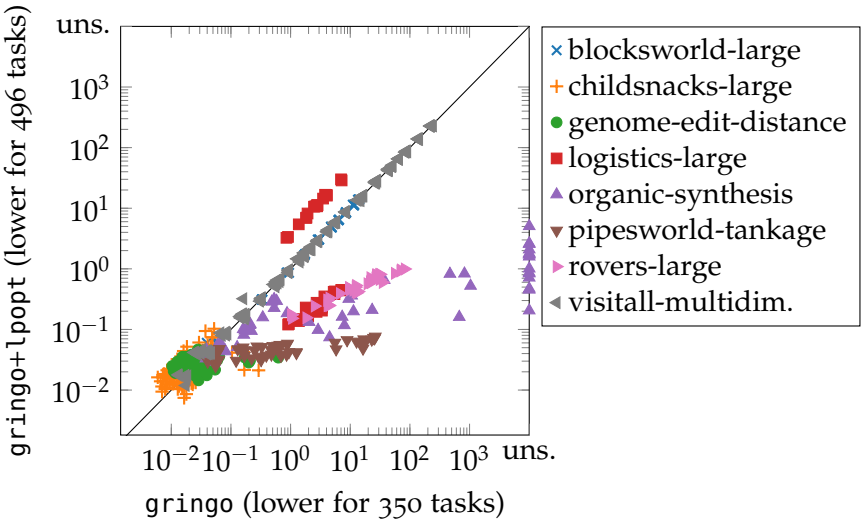


Figure 6.2: Run time comparison between `gringo` and `gringo+lpopt` on the simplified Datalog programs.

Example 6.2 *We use the same logic program $\mathcal{L}$ as Example 2.1:*

$$pet(\text{bob}).$$
$$pet(\text{jack}).$$
$$fish(\text{jack}).$$
$$dog(X) \vee cat(X) \leftarrow pet(X), \neg fish(X).$$

The Herbrand base $\mathcal{H}(\mathcal{L})$ of $\mathcal{L}$ is

$$\mathcal{H}(\mathcal{L}) = \{pet(\text{bob}), pet(\text{jack}), fish(\text{bob}), fish(\text{jack}), dog(\text{bob}), dog(\text{jack}), cat(\text{bob}), cat(\text{jack})\},$$

and the set of ground rules Ground$(\mathcal{L})$ contains the following rules

$$pet(\text{bob}).$$
$$pet(\text{jack}).$$
$$fish(\text{jack}).$$
$$dog(\text{bob}) \vee cat(\text{bob}) \leftarrow pet(\text{bob}), \neg fish(\text{bob}).$$
$$dog(\text{jack}) \vee cat(\text{jack}) \leftarrow pet(\text{jack}), \neg fish(\text{jack}).$$

A possible interpretation is

$$\mathcal{I}_1 = \{pet(\text{bob}), pet(\text{jack}), fish(\text{jack}), dog(\text{bob})\}.$$

Another possible interpretation is

$$\mathcal{I}_2 = \{pet(\text{bob}), pet(\text{jack}), fish(\text{jack}), dog(\text{bob}), cat(\text{bob})\}.$$

There are multiple interpretations for $\mathcal{L}$ in Example 6.2, as the example shows. But not all interpretations are equally satisfactory: it is not the intention of our program that bob can be both a cat and a dog, which is violated by $\mathcal{I}_2$. In answer set programming, we consider the *stable model* semantics (Gelfond and Lifschitz, 1988, 1991) for interpretations. To define stable models, we first define the notion of a *reduct*. The reduct $\mathcal{L}^{\mathcal{I}}$ of a logic program $\mathcal{L} = \langle \mathcal{R}, \mathcal{F} \rangle$ with respect to an interpretation $\mathcal{I}$ is defined as $\mathcal{L}^{\mathcal{I}} = \langle \mathcal{R}^{\mathcal{I}}, \mathcal{F}^{\mathcal{I}} \rangle$, where

stable model

reduct

$$\mathcal{F}^{\mathcal{I}} = \mathcal{F},$$
$$\mathcal{R}^{\mathcal{I}} = \{head(r) \leftarrow body^{+}(r) \mid r \in Ground(\mathcal{R}), body^{-}(r) \cap \mathcal{I} = \emptyset\},$$

$body^{+}(r)$ are the positive atoms of $body(r)$, and $body^{-}(r)$ are the negated atoms of $body(r)$. The reduct $\mathcal{L}^{\mathcal{I}}$ can be obtained from a set of ground rules, by deleting those rules that contain some (ground) atom in $\mathcal{I}$ occurring negatively in the body of r, and by deleting all negated atoms in the body of the remaining ground rules.

The interpretation $\mathcal{I}$ is a stable model of $\mathcal{L}$ if $\mathcal{I}$ is a model of $\mathcal{L}$ and no proper subset of $\mathcal{I}$ is a model of the reduct $\mathcal{L}^{\mathcal{I}}$ (Bliem et al., 2020).

Example 6.3 *The reducts $\mathcal{L}^{\mathcal{I}_1}$ and $\mathcal{L}^{\mathcal{I}_2}$ are the same:*

$$pet(\text{bob}).$$
$$pet(\text{jack}).$$
$$fish(\text{jack}).$$
$$dog(\text{bob}) \vee cat(\text{bob}) \leftarrow pet(\text{bob}).$$

Therefore, $\mathcal{I}_2$ is not *a stable model as $\mathcal{I}_1 \subset \mathcal{I}_2$ and $\mathcal{I}_1$ is also a model of the reduct $\mathcal{L}^{\mathcal{I}_2}$. On the other hand, $\mathcal{I}_1$ is indeed a stable model of $\mathcal{L}$. But it is not the only one either:*

$$\mathcal{I}_3 = \{pet(\text{bob}), pet(\text{jack}), fish(\text{jack}), cat(\text{bob})\}$$

is also a stable model of $\mathcal{L}$.

Deciding whether there *exists* a stable model of a given logic program is **NP**-hard (Dantsin et al., 2001). [3] The problem remains hard even if the rules (but not the facts) are fixed.

In our ASP programs, we also use *choice rules*. These are rules of the form

choice rules

$$n\,\{p(X) : r(X)\}\,m.$$

expressing that at least n and at most m atoms $p(X)$ have to be chosen, and for each chosen atom $r(X)$ must also be in the model. We only need the specific case where $n = m = 1$:

$$1\,\{p(X) : r(X)\}\,1.$$

3 More precisely, the problem of deciding whether a stable model exists is complete for the second level of the polynomial hierarchy – i.e., $\Sigma_2^{\mathbf{P}}$-complete. This is believed to fall in between **NP** and **PSPACE**.

stating that exactly one $p(X)$ must be chosen for which $r(X)$ is in the model. This rule is just syntactic sugar and it can be simulated with two additional predicates *has-p*/0 and $p'/1$ and the following rules.[4] The rules

$$\begin{aligned} \textit{has-p} &\leftarrow p(X). \\ \bot &\leftarrow \neg \textit{has-p}. \end{aligned}$$

ensure that at least one $p(X)$ is in the model; the rule

$$\bot \leftarrow p(X), p(Y), X \neq Y.$$

ensures that at most one $p(X)$ is in the model; and the rule

$$p(X) \vee p'(X) \leftarrow r(X).$$

allows the model to contain $p(X)$ if it contains $r(X)$, but does not enforce it — it could contain $p'(X)$ instead. (A stable model cannot contain both $p(X)$ and $p'(X)$ due to the minimality condition.)

When a choice rule is ground, we simply represent it as a set of ground atoms. Given ground atoms $g_1, \ldots, g_n$, the choice rule

$$1 \{g_1, \ldots, g_n\} 1.$$

indicates that exactly one g_i for $1 \leq i \leq n$ can be part of any stable model.

6.6 GROUNDING VIA ITERATED SOLVING

On the one hand, the simplified program, without action applicability rules, is much simpler to ground while its model still contains all relaxed-reachable atoms of the task. On the other hand, it does not contain any information about the ground actions 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 *join* the preconditions of the action, similarly to what we did in Chapter 3 for the lifted successor generation. Although simple in theory, this is not straightforward: the intermediate results of this join can be exponentially large. While we can try to solve this issue with more sophisticated methods (e.g., Gottlob et al., 2002), keeping track of all intermediate results would still yield a large memory footprint. An alternative is to produce one ground action at a time sacrificing run time for memory. Unfortunately, we are not aware of any specific solver with this particular feature (i.e., generate answers to the join one by one, without keeping track of all intermediate results).

4 This assumes that predicate symbol p occurs only in one choice rule. If $p(X)$ appears in more choice rules or in the head of other rules, we must use extra predicate symbols.

But there are other techniques in the literature that we can use off-the-shelf and have the exact advantages that we are looking for: generate ground actions one by one without keeping track of exponentially large intermediate results. We introduce a two-phase method, inspired by the techniques of *grounding via solving* (Besin et al., 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.

grounding via solving

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

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

we create the logic program $\mathcal{L}^r = \langle \mathcal{F}, \mathcal{R} \rangle$ with $\mathcal{F} = \mathcal{M}$, and rules $\mathcal{R}$ as follows. For a given rule r with $body(r) = \{q_1(\boldsymbol{T}_1), \ldots, q_n(\boldsymbol{T}_n)\}$, and a variable $V \in vars(r)$, let $\{q_1^V(\boldsymbol{T}_1^V), \ldots, q_m^V(\boldsymbol{T}_m^V)\} \subseteq body(r)$ be the set of all atoms in the body of r such that $V \in \boldsymbol{T}_i^V$ for $1 \leq i \leq m \leq n$. For every variable $V \in vars(r)$, we introduce a fresh predicate $V\text{-}assign$ and the following choice rule:

$$1\,\{V\text{-}assign(V) : q_1^V(\boldsymbol{T}_1^V), \ldots, q_m^V(\boldsymbol{T}_m^V)\}\,1.$$

where $V \in \boldsymbol{T}_i$ for $1 \leq i \leq m$. This rule forces the stable model to pick for each variable exactly one constant that unifies with the set $\mathcal{F}$ of facts and thus form a variable assignment.

Further, for every (non-ground) atom $q_i(X_1, \ldots, X_k) \in body(r)$, 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 $q_i(X_1, \ldots, X_k)$ in all stable models of $\mathcal{L}^r$.

The program $\mathcal{L}^r$ 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 $V\text{-assign}(c)$, 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 $\mathcal{L}^r$, 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 in check, as we do not have to generate all ground actions at once.

One could try to use our grounding via solving with simpler ASP programs in the second phase. The most straightforward idea is to create an ASP program for each action schema a containing the single rule

$$a\text{-applicable}(\boldsymbol{T}) \leftarrow p_1(\boldsymbol{T}_1), \ldots, p_n(\boldsymbol{T}_n).$$

where $\{p_1(\boldsymbol{T}_1), \ldots, p_n(\boldsymbol{T}_n)\} = pre(a)$ and the set of facts is also the set of reachable atoms from the first phase. But note also that our logic program in the second phase must be ground before the stable models can be computed. Unfortunately, grounding this single rule can be very expensive. In our logic program $\mathcal{L}^r$ above, however, grounding is trivial, as we see in the next example.

Example 6.4 *In a logistics problem (Example 2.4), assume we have a drive action that allows a truck to drive from a city C_1 to an adjacent city C_2. In our Datalog program, we have the following two rules:*

$$\begin{aligned} \mathit{drive}\text{-applicable}(T, C_1, C_2) &\leftarrow \mathit{at}(T, C_1), \mathit{adj}(C_1, C_2). \\ \mathit{at}(T, C_2) &\leftarrow \mathit{drive}\text{-applicable}(T, C_1, C_2). \end{aligned}$$

When we remove action applicability predicates, we obtain the following rule r in our simplified program:

$$\mathit{at}(T, C_2) \leftarrow \mathit{at}(T, C_1), \mathit{adj}(C_1, C_2).$$

Let's also say that we have the following model $\mathcal{M}$ from our first stage:

$$\begin{aligned} &\mathit{adj}(\mathrm{a}, \mathrm{b}). \\ &\mathit{adj}(\mathrm{b}, \mathrm{a}). \\ &\mathit{adj}(\mathrm{b}, \mathrm{c}). \\ &\mathit{adj}(\mathrm{c}, \mathrm{b}). \\ &\mathit{at}(\mathrm{t}, \mathrm{a}). \\ &\mathit{at}(\mathrm{t}, \mathrm{b}). \\ &\mathit{at}(\mathrm{t}, \mathrm{c}). \end{aligned}$$

This encodes that cities a *and* b *are adjacent, and so are cities* b *and* c*, and that the truck* t *can be in any of the three cities.*

In our second stage, we generate the following program $\mathcal{L}^r$:

$$\begin{aligned} &1\,\{T\text{-}\mathit{assign}(T) : \mathit{at}(T, C_1)\}\,1. \\ &1\,\{C_1\text{-}\mathit{assign}(C_1) : \mathit{at}(T, C_1), \mathit{adj}(C_1, C_2)\}\,1. \\ &1\,\{C_2\text{-}\mathit{assign}(C_2) : \mathit{adj}(C_1, C_2)\}\,1. \\ &\bot \leftarrow T\text{-}\mathit{assign}(T), C_1\text{-}\mathit{assign}(C_1), \neg \mathit{at}(T, C_1). \\ &\bot \leftarrow C_1\text{-}\mathit{assign}(C_1), C_2\text{-}\mathit{assign}(C_2), \neg \mathit{adj}(C_1, C_2). \end{aligned}$$

Grounding this program with $\mathcal{M}$ we obtain (omitting the facts from $\mathcal{M}$):

$1\,\{T\text{-}assign(\mathrm{t})\}\,1.$
$1\,\{C_1\text{-}assign(\mathrm{a}), C_1\text{-}assign(\mathrm{b}), C_1\text{-}assign(\mathrm{c})\}\,1.$
$1\,\{C_2\text{-}assign(\mathrm{a}), C_2\text{-}assign(\mathrm{b}), C_2\text{-}assign(\mathrm{c})\}\,1.$
$\bot \leftarrow T\text{-}assign(\mathrm{t}), C_1\text{-}assign(\mathrm{a}), \neg at(\mathrm{t},\mathrm{a}).$
$\bot \leftarrow T\text{-}assign(\mathrm{t}), C_1\text{-}assign(\mathrm{b}), \neg at(\mathrm{t},\mathrm{b}).$
$\bot \leftarrow T\text{-}assign(\mathrm{t}), C_1\text{-}assign(\mathrm{c}), \neg at(\mathrm{t},\mathrm{c}).$
$\bot \leftarrow C_1\text{-}assign(\mathrm{a}), C_2\text{-}assign(\mathrm{b}), \neg adj(\mathrm{a},\mathrm{b}).$
$\bot \leftarrow C_1\text{-}assign(\mathrm{b}), C_2\text{-}assign(\mathrm{a}), \neg adj(\mathrm{b},\mathrm{a}).$
$\bot \leftarrow C_1\text{-}assign(\mathrm{b}), C_2\text{-}assign(\mathrm{c}), \neg adj(\mathrm{b},\mathrm{c}).$
$\bot \leftarrow C_1\text{-}assign(\mathrm{c}), C_2\text{-}assign(\mathrm{b}), \neg adj(\mathrm{c},\mathrm{b}).$

This program has 4 *stable models:*

1. $\mathcal{M} \cup \{T\text{-}assign(\mathrm{t}), C_1\text{-}assign(\mathrm{a}), C_2\text{-}assign(\mathrm{b})\}$
2. $\mathcal{M} \cup \{T\text{-}assign(\mathrm{t}), C_1\text{-}assign(\mathrm{b}), C_2\text{-}assign(\mathrm{a})\}$
3. $\mathcal{M} \cup \{T\text{-}assign(\mathrm{t}), C_1\text{-}assign(\mathrm{b}), C_2\text{-}assign(\mathrm{c})\}$
4. $\mathcal{M} \cup \{T\text{-}assign(\mathrm{t}), C_1\text{-}assign(\mathrm{c}), C_2\text{-}assign(\mathrm{b})\}$

They correspond to the following relaxed-reachable instantiations of the action drive:

1. $drive(\mathrm{t},\mathrm{a},\mathrm{b})$
2. $drive(\mathrm{t},\mathrm{b},\mathrm{a})$
3. $drive(\mathrm{t},\mathrm{b},\mathrm{c})$
4. $drive(\mathrm{t},\mathrm{c},\mathrm{a})$

Note also that not all combinations of assignment of the choice rules yield a stable model: for example, choosing C_1-assign(a) and C_2-assign(a) never leads to a stable model.

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 ground actions from the logic program above. Table 6.4 shows the results. Compared to the methods that ground action predicates (Table 6.1), `iterated` grounds more tasks in total. It is better in the domains childsnack, organic-synthesis, and pipesworld, but worse in domains rovers. As discussed earlier, the rovers domain is challenging for the methods only exploiting structural properties.

Domain	`iterated`	`iterated`$^{\neq}$
IPC Sum (1001)	**1001**	**1001**
blocksworld (40)	**40**	**40**
childsnack (144)	142	**144**
genome-edit-dist. (312)	**312**	**312**
logistics (40)	**40**	**40**
organic-synthesis (56)	23	**32**
pipesworld-tankage (50)	**50**	**50**
rovers (40)	**23**	**23**
visitall-multidim. (180)	**180**	**180**
HTG Sum (862)	810	**821**

Table 6.4: Number of grounded tasks for `iterated` and `iterated`$^{\neq}$. We abbreviate `gringo` with `G`, and `lpopt` with `L`.

While `iterated` can solve tasks where other methods run out of memory, it is much slower than any other algorithm. Figure 6.1 shows the run time of `iterated`. Even for simple tasks, `iterated` takes more time than any other algorithm. Noticeable overhead comes from the communication of the different components, i.e., the effort of setting up the model, handling the result, calling `clingo` multiple times for each action schema.

Still, some tasks were not grounded by any method (ignoring methods that ground only simplified programs). More precisely, there are 33 organic-synthesis 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 et al. (2021).

The smallest (in number of ground actions) task from organic-synthesis that we cannot ground has $30 \cdot 10^8$ ground actions. The largest one has $60 \cdot 10^{35}$ ground actions. For comparison, the largest instance that our algorithms could ground had $20 \cdot 10^7$ ground actions in total. 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.

6.7 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 filter to remove ground atoms violating mutexes.

We propose an approach that does not modify `gringo` or `iterated`. This is desirable because as these specialized tools evolve, we can still use them off-the-shelf without having to adapt any implementation. So instead of changing the algorithm, we simply modify the Datalog program to consider *negated static predicates* in preconditions. Static predicates are those predicates that only occur in preconditions but never in effects. When we create the Datalog program of the task, static predicates never occur in the head of any rule (they are extensional predicates). We can exploit this by adding any negatively occurring static predicate to the body of its action applicability rule (also negatively). This transforms our Datalog program into a *stratified Datalog* program, but it still preserves the uniqueness of the canonical model (Ullman, 1988, 1989).

stratified Datalog

In our benchmark set, the only occurrences of negated static preconditions are inequality constraints. (Other known 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 works for any static predicate. Inequalities have the additional advantage that they can be treated as a built-in predicate by `gringo`. (Note that this is also the case in PDDL.)

But Fast Downward, for example, already removes actions that violate inequalities in a postprocessing step (Helmert, 2009). Let us call these actions *impossible actions*. However, `FD` can still produce actions that are only relaxed-reachable via impossible actions. Incorporating inequalities into the logic programs solves this problem.

Example 6.5 *Consider the following Datalog program:*

$$\begin{aligned} p(\mathrm{c}).& \\ \textit{a-applicable}(X,Y) &\leftarrow p(X), p(Y), X \neq Y. \\ b(X) &\leftarrow \textit{a-applicable}(X,Y). \\ G &\leftarrow b(\mathrm{c}). \end{aligned}$$

The canonical model of this program is

$$\mathcal{M} = \{p(\mathrm{c})\},$$

and the task is relaxed unsolvable, since the goal atom G is not in $\mathcal{M}$.

If $X \neq Y$ is removed from action applicability rule, then the canonical model is

$$\mathcal{M} = \{p(\mathrm{c}), \textit{a-applicable}(\mathrm{c},\mathrm{c}), b(\mathrm{c}), G\}.$$

The FD *algorithm postprocesses the ground actions and identifies that a-applicable*(c, c) *is an impossible action, because it violates a static precondition. Hence, this action is discarded. But* FD *cannot do that for* $b(c)$ *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 6.4. 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 when adding inequalities.

Compared to `iterated`, `iterated`$^{\neq}$ grounds 2 additional tasks in childsnack and 9 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 6.1 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 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" $23 \cdot 10^8$ actions, and the largest has $81 \cdot 10^{33}$. Although this is a significant reduction in the number of actions in the worst case, these numbers are still too large for current grounders and planners.

6.8 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 6.5 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 logistics, while using `gringo` solves more tasks in rovers and organic synthesis. LAMA with `gringo` solves many additional instances in these two domains, particularly in

Domain	FD	G	iterated$^{\neq}$
blocksworld (40)	**12**	**12**	**12**
childsnack (144)	**116**	**116**	**116**
genome-edit-dist. (312)	**312**	**312**	**312**
logistics (40)	**36**	32	30
organic-synthesis (56)	21	**30**	27
pipesworld-tankage (50)	**18**	**18**	**18**
rovers (40)	17	**40**	23
visitall-multidim. (180)	**72**	**72**	**72**
Total (862)	604	**632**	610

Table 6.5: 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.

rovers. All tasks in rovers are easy to solve and were beyond LAMA's capacity only due to its grounder. We can say the same about organic-synthesis, a domain known for having short plans. Using iterated$^{\neq}$ is never the best option in any of our domains.

We also analyze the run time of each method. Figure 6.3 compares the number of instances solved over time. There is a clear ordering there: FD is the slowest, followed by iterated$^{\neq}$ and gringo. This agrees with our previous observation (see Figure 6.1) 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.

6.9 SUMMARY

In this chapter, 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 also presented a more sophisticated method, called iterated$^{\neq}$, that decouples the grounding procedure into two phases: one to obtain all relaxed-reachable atoms, and one to obtain all relaxed-reachable actions. This approach uses the insights obtained from Part i. While this method can ground more logic programs, it is also slower than off-

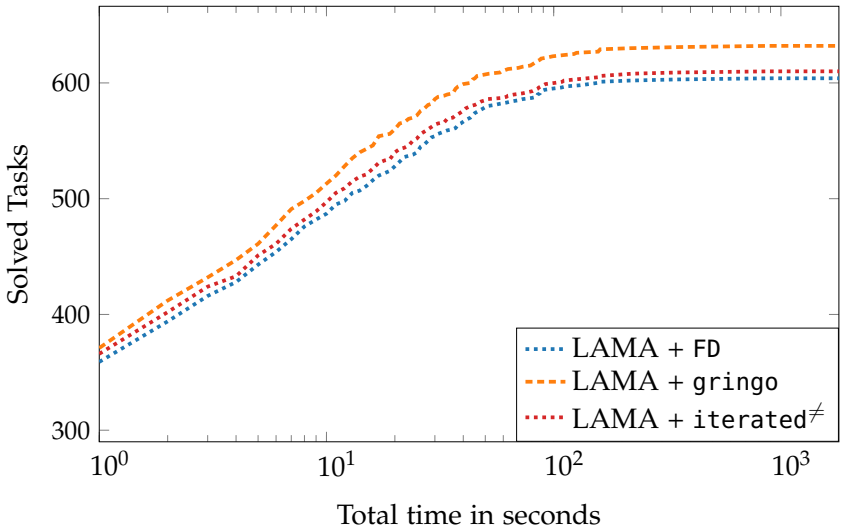


Figure 6.3: Number of tasks solved over time (in seconds) using LAMA with different grounders.

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. Both `gringo` and `iterated`$^{\neq}$ help classical planners solve more tasks. This means that some tasks that we can now ground were already in reach of classical planners.

To deal with the tasks that we could not ground, one should look into grounders not based on delete-relaxation or lifted planners. We also believe that `iterated` could be used to ground Datalog programs in general, outside the planning context.

CHAPTER NOTES & HISTORY

We focused 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 the FF planner (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 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 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 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).

An alternative solution 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).

An orthogonal approach is to optimize how the domains are encoded, instead of improving the grounding algorithms. Haslum (2007) show that many planning domains are harder than necessary due to "accidental complexity": encodings hiding structures of the task that could be exploited by planners and grounders. Haslum shows that, in some cases, it is possible to automatically remove this accidental complexity using some simpler transformations. A different way to optimize the model is via *action schema splitting*. By splitting action schemas into smaller ones, grounding becomes easier. This split can be done by hand (e.g., Haslum, 2011) or automatically (Areces et al., 2014; Elahi and Rintanen, 2024). To help the grounding process, these algorithms transform an action schema with multiple variables into several schemas with few variables. This is the same idea as Helmert's decomposition techniques in Datalog that we also applied here. Elahi and Rintanen (2024) show that good action schema splits are preferable over lifted planning in many domains.

One of our contributions in this chapter was the use of structural decompositions to speed up the grounder. Longo (2023) continues this study with other decomposition techniques. In his work, Longo uses hypertree decompositions instead of tree decompositions. He shows that we can efficiently compute optimal hypertree decompositions for all rules in our benchmark. In the HTG set, the *hypertreewidth* (analogous to treewidth) is at most 2, which is lower than the treewidth computed by `lpopt`. However, decomposing rules based on their hypertreewidth yields worse performance than using `lpopt`. At first sight, the hypertree decompositions significantly improve the performance of the grounder in the rovers domain. But Longo shows that this is by chance: the decomposition found by his method seems to be "lucky",

as they quickly filter the large initial relations of this domain. By switching some tiebreakers and parameters in his algorithm, the new decompositions found no longer perform well in the rovers domain. This supports our earlier hypothesis that in some domains we might want to use extra information when decomposing rules (e.g., size of relation in the initial state).

Our original paper (Corrêa et al., 2023e) already presented all the techniques introduced in this chapter. There are some minor differences, however. Most importantly, our original definition of `iterated` used only type predicates in the choice rule defining the values for $\mathit{V\text{-}assign}(X)$. The current presentation includes in the choice rule all atoms that mention variable X. Although not strictly necessary for correctness, this change decreased run time in almost 30% in some instances. Additionally, since the original publication, we extended our grounder to support other off-the-shelf grounders, such as I-DLV (Calimeri et al., 2017; Faber et al., 2012). The results are very similar if we switch `gringo` with I-DLV. We only presented the results using `gringo` to keep consistency with the published work, and because it presents a better performance.

Part III

PLANNING WITH OBJECT CREATION

7

PLANNING WITH OBJECT CREATION

Bob, a former classical planning researcher, opens a new logistics company. Real life, however, is not so simple. He first needs to decide how many trucks he should buy. Buying many trucks is not an issue — Bob became very rich working on classical planning — but he still wants to minimize his expenses. He decides to tackle this problem using classical planning. Bob encodes the delivery locations and the roads connecting them in PDDL (Haslum et al., 2019; McDermott, 2000). He then declares a bunch of truck *objects* in advance, and cleverly encodes his actions to balance the costs between buying a new truck and doing more deliveries with the same truck. But how does he know how many trucks to declare in advance? Can he estimate an upper bound? Computing it seems as if he is solving the problem *himself*, so he goes with a rough estimate of 10 trucks. But does the optimal solution only require 10 trucks? What if it requires much more? Bob gets worried and increases the number of trucks to 1,000.

He finally formalizes his problem and runs it on some classical planners. All planners take months to solve his task. At the end, the optimal plan uses 11 trucks. Even if 98.9% of the truck objects were irrelevant, they still impacted the performance of the planners (Fuentetaja and de la Rosa, 2016; Silver et al., 2021). Bob ends up frustrated with the whole procedure. If only there was a native way to let planners introduce more objects as they plan.

In this chapter, we introduce a novel way of dealing with problems where the objects are not all known upfront. Instead of preemptively declaring all objects in the definition of the task, objects can be *created* via action effects. There are three immediate benefits to this extension:

(i) it makes the encoding simpler and more natural for several domains (e.g., Long and Fox, 2003),

(ii) it reduces the amount of expert knowledge needed in the domain encoding (e.g., Petrov and Muise, 2023), and

(iii) it might improve performance of planners by reducing state size and number of unnecessary objects (e.g., Fuentetaja and de la Rosa, 2016).

On the theoretical side, we prove that classical planning with object creation is *semi-decidable*. In other words, if a plan exists we are guar-

anteed to find it. However, no algorithm can recognize, in general, if a task is unsolvable. This is still the case even if we restrict our input to delete-free tasks. If we talk about *bounded plan existence* though, the problem becomes decidable.

On the practical side, we introduce a PDDL (Haslum et al., 2019; McDermott et al., 1998) extension that allows for object creation in the effect of actions. Moreover, we show that Powerlifted (and potentially any lifted planner with a similar representation) is well-suited to solve this problem. From Chapter 3, we know that lifted successor generation can be cast as a conjunctive query problem. A conjunctive query can be solved based only on the current state being expanded. This implies that when generating successors for two different states, the algorithm implemented in Powerlifted does not care if the set of objects differs. With minor changes in state representation and parsing, Powerlifted is already fully-equipped to execute a breadth-first search in tasks with object creation. Beyond breadth-first search, we extend the width search algorithms from Chapter 5 to take into account the number of objects in the evaluated state. In our experimental results, Powerlifted has better performance when using the PDDL extension in comparison to the original PDDL encodings where all objects are declared beforehand and object creation is simulated with auxiliary predicates.

7.1 DETAILS OF FIRST-ORDER LOGIC

Hitherto, all previous chapters assumed a background on logic. In this chapter, we use a more specific notation of *first-order logic*.

We consider *first-order languages* $\mathcal{L} = \langle \mathcal{V}, \mathcal{C}, \mathcal{P} \rangle$. As before, $\mathcal{V}$ is a finite set of *variables*, $\mathcal{C}$ is a finite set of *constants*, and $\mathcal{P}$ is a finite set of *predicate symbols*. We continue to restrict our definitions to languages without function symbols.

interpretation

An *interpretation* over a first-order language $\mathcal{L}$ is a tuple $\mathcal{I} = \langle \mathcal{U}^{\mathcal{I}}, \{c^{\mathcal{I}}\}_{c \in \mathcal{C}}, \{p^{\mathcal{I}}\}_{p \in \mathcal{P}} \rangle$ consisting of

objects
universe

- a finite set $\mathcal{U}^{\mathcal{I}}$ of *objects* called the *universe*;
- for each constant $c \in \mathcal{C}$, its interpretation $c^{\mathcal{I}} \in \mathcal{U}^{\mathcal{I}}$.
- for each predicate symbol $p \in \mathcal{P}$, its interpretation $p^{\mathcal{I}} \subseteq (\mathcal{U}^{\mathcal{I}})^{ar(p)}$. We write $p(o_1, \ldots, o_{ar(p)})$ to indicate that $\langle o_1, \ldots, o_{ar(p)} \rangle \in (\mathcal{U}^{\mathcal{I}})^{ar(p)}$

For the interpretation of predicates, we also define $\mathcal{P}^{\mathcal{I}} = \{p^{\mathcal{I}}\}_{p \in \mathcal{P}}$ to shorten notation. In our context, interpretations are always finite.

7.2 PLANNING FORMALISM

To add object creation to our formalism, we must change our definition of a planning task (Chapter 2). States now correspond to first-order

interpretations, where the set of existing objects is given by the universe of the state, and not by a fixed set of constants. The initial set $\mathcal{C}$ of constants represents only those constants that occur in the goal or in some action schema. When applying an action a to transition from some state s to a state s', if an object is created, a new object $o \notin \mathcal{U}^s$ is added to $\mathcal{U}^{s'}$.

planning task with object creation

Formally, a *planning task with object creation* is a tuple $\Pi = \langle \mathcal{L}, \mathcal{A}, I, G \rangle$, where $\mathcal{L} = \langle \mathcal{P}, \mathcal{C}, \mathcal{V} \rangle$ is a first-order language; I is the *initial state*; G is the *goal*; $\mathcal{A}$ is a finite set of *action schemas*, defined below.

States are interpretations over $\mathcal{L}$. We assume a *fixed* interpretation of constants, where a constant $c \in \mathcal{C}$ is always mapped to an object $o_c \in \mathcal{U}^s$ in any state s (i.e., $c^s = o_c$ for all s). This means that every constant c is always mapped to the same object o_c, and any two states have exactly the same interpretation of constants. (Recall that, as just mentioned, constants in $\mathcal{C}$ are those appearing in the goal and in action schemas.) Therefore, we drop the interpretation of constants from our notation, and write states simply as $s = \langle \mathcal{U}^s, \{p^s\}_{p \in \mathcal{P}} \rangle$. Later, created objects will correspond to objects in $\mathcal{U}^s$ but never to constants.

The goal G is a set of ground atoms.

In contrast to other chapters, we drop the assumption that our states have an implicit inequality relation ($\neq$).

Example 7.1 *We use the logistics scenario from the introduction as a running example. Our task has a package* p_1 *initially located at city* c_1*. There are two cities,* c_1 *and* c_2*, which are connected. The city* c_1 *is the headquarters of the company. The goal is to move* p_1 *from* c_1 *to* c_2*.*

The predicate symbols are $at/2, in/2, adj/2$ *and* $headquarters/1$*. The initial state* I *has:*

$$\begin{aligned} \mathcal{U}^I &:= \{o_{p_1}, o_{c_1}, o_{c_2}\} \\ \mathcal{P}^I &:= \{adj(o_{c_1}, o_{c_2}), adj(o_{c_2}, o_{c_1}), \\ &\qquad at(o_{p_1}, o_{c_1}), headquarters(o_{c_1})\}. \end{aligned}$$

We define the goal $G = \{at(p_1, c_2)\}$.

Action schemas are defined in Example 7.2 below.

An action schema $a \in \mathcal{A}$ is a triple $a = \langle pre(a), add(a), del(a) \rangle$ where each element is a set of atoms as before. However, in contrast to our original definition (Chapter 2), the set of variables of action a is now a pair $vars(a) = \langle params(a), fresh(a) \rangle$, where

- $params(a) \subseteq \mathcal{V}$ is the set of *action parameters*; — *action parameters*
- $fresh(a) \subseteq \mathcal{V}$ is the set of *fresh variables*; — *fresh variables*
- $params(a) \cap fresh(a) = \emptyset$.

We define $vars(pre(a)), vars(add(a))$ and $vars(del(a))$ as sets of variables, where

- $vars(pre(a)) \subseteq params(a)$;
- $vars(add(a)) \subseteq params(a) \cup fresh(a)$;
- $vars(del(a)) \subseteq params(a)$.

Intuitively, action parameters must be instantiated with objects in the universe of the current state, and fresh variables must be instantiated with new objects. The object selected to instantiate a fresh variable is a called a *fresh object*.

fresh object

Example 7.2 *We detail two action schemas, buy and move, from our running example, where*

$$\begin{aligned} pre(buy) &= \{headquarters(L)\} \\ add(buy) &= \{at(T, L)\} \\ del(buy) &= \emptyset \\ pre(move) &= \{at(T, L), adj(L, M)\} \\ add(move) &= \{at(T, M)\} \\ del(move) &= \{at(T, L)\}. \end{aligned}$$

Action schema buy says "if L is the headquarters, then add a new truck to L"; and move says "move a truck T from location L to M".

These two action schemas have the following action parameters and fresh variables:

$$\begin{aligned} params(buy) &= \{L\} \\ fresh(buy) &= \{T\} \\ params(move) &= \{T, L, M\} \\ fresh(move) &= \emptyset. \end{aligned}$$

When applying the buy action in a given state, the variable T must be instantiated with a fresh object, while variable L must be instantiated with an object that exists in the current state. For action move, all variables must be instantiated with existing objects.

Given a state s and an action schema $a \in \mathcal{A}$, a *variable assignment function* $\sigma_{s,a}$ maps variables in $params(a) \cup fresh(a)$ to objects.[1] We enforce the following two properties on $\sigma_{s,a}$:

variable assignment function

- $\sigma_{s,a}(v) \in \mathcal{U}^s$ for all $v \in params(a)$;

1 This is similar to the definition of substitution function, but here we map variables to a universe, and not to the set of constants (Chapter 2). We can assume that the image of the function is an underlying infinite universe $\mathcal{U}$, containing all possible objects. However, including such a universe with infinitely many objects to the object does not fit conceptually with the notion of a classical planning task. This approach makes things more complicated once we allow for fragments that are more expressive than STRIPS (see chapter notes).

- $\sigma_{s,a}(v) \notin \mathcal{U}^s$ for all $v \in \mathit{fresh}(a)$.

Any variable assignment function maps action parameters to objects in the state, and fresh variables to objects not in the state.

A *ground action* $\sigma_{s,a}(a)$ is *applicable* in s if $s \models \sigma_{s,a}(\mathit{pre}(a))$.[2] The *successor state* $\mathit{succ}(s, \sigma_{s,a}(a))$ is defined as follows. Let $\mathit{new}(\sigma_{s,a})$ be the set of new objects introduced by $\sigma_{s,a}$ defined as

$$\mathit{new}(\sigma_{s,a}) = \{\sigma_{s,a}(v) \mid v \in \mathit{fresh}(a)\}.$$

Then $\mathit{succ}(s, \sigma_{s,a}) = \langle \mathcal{U}', (p')_{p \in \mathcal{P}} \rangle$ is defined as

$$\begin{aligned} \mathcal{U}' &= \mathcal{U}^s \cup \mathit{new}(\sigma_{s,a}) \\ p' &= (p^s \setminus \sigma_{s,a}(\mathit{del}(a)) \cup \sigma_{s,a}(\mathit{add}(a))). \end{aligned}$$

Example 7.3 *Let $\sigma_{I,buy}$ be defined as follows:*

$$\begin{aligned} \sigma_{I,buy}(L) &= \mathrm{c}_1 \\ \sigma_{I,buy}(T) &= \mathrm{t}. \end{aligned}$$

As $\mathit{fresh}(buy) = \{T\}$, the set of new objects introduced by $\sigma_{I,buy}$ is

$$\mathit{new}(\sigma_{I,buy}) = \{\mathrm{t}\},$$

and the successor state $s_1 = \mathit{succ}(I, \sigma_{I,buy})$ is $s_1 = \langle \mathcal{U}^{s_1}, \mathcal{P}^{s_1} \rangle$, where

$$\begin{aligned} \mathcal{U}^{s_1} &:= \{\mathrm{p}_1, \mathrm{c}_1, \mathrm{c}_2, \mathrm{t}\} \\ \mathcal{P}^{s_1} &:= \{\mathit{adj}(\mathrm{c}_1, \mathrm{c}_2), \mathit{adj}(\mathrm{c}_2, \mathrm{c}_1), \\ &\qquad \mathit{at}(\mathrm{p}_1, \mathrm{c}_1), \mathit{at}(\mathrm{t}, \mathrm{c}_1), \mathit{headquarters}(\mathrm{c}_1)\}. \end{aligned}$$

The reachable state space of our task can thus be interpreted as a graph over first-order interpretations. It is important to observe that although we have variables mapped outside the universe $\mathcal{U}^s$ in $\sigma_{s,a}$, this does not influence any semantics of first-order logic. The only time the semantics of interpretations is necessary is when checking action applicability ($s \models \sigma_{s,a}(\mathit{pre}(a))$), but here it is enough to consider a restriction $\sigma_{s,a}|_{\mathit{params}(a)}$ of $\sigma_{s,a}$, which is well-defined over the interpretation s. The new objects (not in $\mathcal{U}^s$) are only necessary to identify how any two connected interpretations change in the state space.

A *plan* for state I is a sequence $\sigma_{s_0,a_1}(a_1), \ldots, \sigma_{s_{n-1},a_n}(a_n)$ of ground actions such that $s_0, \ldots, s_n$ are states where $s_0 = I$ and $\sigma_{s_{i-1},a_i}(a_i)$ is applicable in s_{i-1} and $s_i = \mathit{succ}(s_{i-1}, \sigma_{s_{i-1},a_i})$ for $1 \leq i \leq n$, and $s_n \models G$.

We introduce two natural decision problems:

ObjCreation-PlanEx	
Input:	A planning task with object creation Π.
Question:	Is there a plan π for Π?

2 Note that since variables in precondition must be action parameters, which are mapped to objects in the $\mathcal{U}^s$, the operator $\models$ is well-defined.

ObjCreation-PlanLen	
Input:	A planning task with object creation Π, a number $k \in \mathbb{N}$.
Question:	Is there a plan π for Π where $\|\pi\| \leq k$?

ObjCreation-PlanEx is analogous to PlanEx in the case without object creation; ObjCreation-PlanLen is analogous to PlanLen. Remember from Chapter 2 that PlanEx is **EXPSPACE**-complete and PlanLen is **NEXPTIME**-complete when using lifted representation for Π, and both are **PSPACE**-complete with a propositional representation. As we show in the next section, there is no algorithm deciding ObjCreation-PlanEx in general. However, when the answer to ObjCreation-PlanEx is "yes", a breadth-first search is guaranteed to eventually find a plan.

PDDL Extension

We extend the PDDL syntax with the keywords `:new`, which allows for the creation of objects The syntax is as follows:

```
(:new (?v1 ... ?vN) eff)
```

where `v1`, `...`, `vN` are fresh variables and `eff` is an effect. In contrast to the logic formalism above, the PDDL syntax declares all fresh variables as part of the effects. We chose this design because it is closer to the PDDL concept, and it fits more naturally with more expressive PDDL fragments (see chapter notes).[3]

Example 7.4 *In our running example, the action buy is written in PDDL as*

```
(:action buy
 :parameters (?L)
 :precondition (headquarters ?L)
 :effect (:new (?T) (at ?T ?L)))
```

This extension simplifies many PDDL models. With standard PDDL, domain experts need to puzzle out how to simulate object creation. This usually involves adding extra predicates and modifying conditions to take these predicates into account. For example, in the original Settlers domain (Long and Fox, 2003) vehicles can be created during the search. To encode this in PDDL, the authors introduced a new predicate `potential` indicating that an object is a potential vehicle.

3 In PDDL, parameters can be typed. We also allow for this in our PDDL extension, but do not include types in our mathematical formalization for simplicity, as standard predicate logic is untyped. Likewise, PDDL does not have an explicit split between add and delete lists in the effects, but a single section called `:effect`. It is simple to convert this into our add and delete lists distinction: positive atoms in the effect are part of the add list, while negated atoms are part of the delete list.

This leads to a domain model that appears less natural than a version with native PDDL object creation.

Example 7.5 *Under standard PDDL syntax, the action schema from Example 7.4 is written as*

```
(:action buy
 :parameters (?L ?T)
 :precondition (and (headquarters ?L)
                    (not (bought ?T)))
 :effect (and (bought ?T)
              (at ?T ?L)))
```

where `bought` *is a new predicate necessary to track which trucks have already been bought. The action has a new parameter* `?T`*, and all (potentially buyable) trucks need to be declared in advance. If several trucks can instantiate* `?T`*, this leads to a combinatorial explosion, although all successor states would be symmetric.*

7.3 DECIDABILITY RESULTS

Planning with object creation is undecidable. We can use our formalism to decide if a Turing Machine accepts a given input. The proof relies on the usual technique of expanding the tape of the Turing Machine on demand (cf. Reiter 2001, Hoffmann et al. 2009).

We first introduce the necessary notation for TMs.

Definition 7.1 (Turing Machine) *A* Turing Machine *(TM) is given by a tuple* $M = \langle Q, \Sigma, \delta, q_0, q_{accept} \rangle$*, where*

- Q *is a finite set of* states*;*
- Σ *is a finite set of* symbols*, called the* input alphabet*;*
- $\delta : (Q \setminus \{q_{accept}\}) \times (\Sigma \cup \{\square\}) \to Q \times (\Sigma \cup \{\square\}) \times \{\mathtt{L}, \mathtt{R}\}$ *is the* transition function*;*
- $q_0 \in Q$ *is the* start state*;*
- $q_{accept} \in Q$ *is the* accept state*.*

The machine has a head *that can move left (*`L`*) and right (*`R`*), and a working* tape*. We assume the tape is infinite to the right, but not to the left. Given an input* $x \in \Sigma^*$*, the tape starts with* x *written on its* $|x|$ *left-most cells. The special symbol* $\square$ *(which is not contained in* Σ*) is on all the other (infinitely many) cells, denoting that they are empty. The head of the machine starts at the left-most cell (i.e., the first symbol of* x*).*

A configuration *of the TM is described by the current position of its head, its current state, and the current content of the tape.*

Theorem 1 ObjCreation-PlanEx *is undecidable.*

Proof. We reduce the problem of deciding whether a given TM M accepts a given input x to the problem of deciding if there is a plan for Π.

We use the following predicates for Π:

- $\mathit{state}(q)$ encodes that the TM is currently in state q;
- $\mathit{transition}(q_1, s_1, q_2, s_2, d)$ encodes that $\delta(q_1, s_1) = (q_2, s_2, d)$, i.e., from state q_1 when reading s_1 there is a transition that changes state to q_2, writes s_2, and moves the head in direction d;
- $\mathit{head}(c)$ indicates that the head is at cell c;
- $\mathit{next}(c_1, c_2)$ indicates that c_2 is immediately to the right of c_1 in the tape;
- $\mathit{right\text{-}limit}(c)$ indicates that cell c is the current right-most cell; and
- $\mathit{symbol}(c, s)$ encodes that cell c has the symbol s written in it.

We also use the following constant symbols:

- $\square$ encodes the blank symbol;
- $\mathtt{s}_1, \ldots, \mathtt{s}_n$ encode the symbols of the input alphabet;
- $\mathtt{c}_1, \ldots, \mathtt{c}_n$ encode the first n cells of our TM, for an input word x with $|x| = n$;
- `L` and `R` encode the left and the right directions; and
- q_{accept} encodes the accepting state of same name.

Our task has the following action schemas:

(i) read a symbol at cell c and move the head to the left;

(ii) read a symbol at cell c and move the head to the right, to a cell that has been reached before or is in the input; and

(iii) read a symbol at cell c and move the head to the right, to a fresh cell, while expanding the tape.

Action schema (i) is denoted by a_{left}. It has the action parameters $\mathit{params}(a_{\text{left}}) = \{Q_1, Q_2, S_1, S_2, C_1, C_2\}$ and the fresh variables $\mathit{fresh}(a_{\text{left}}) = \emptyset$. We define the precondition $\mathit{pre}(a_{\text{left}})$ as

$$\begin{aligned}\mathit{pre}(a_{\text{left}}) :=\{&\mathit{state}(Q_1), \mathit{transition}(Q_1, S_1, Q_2, S_2, \mathtt{L}),\\ &\mathit{head}(C_1), \mathit{symbol}(C_1, S_1), \mathit{next}(C_2, C_1)\}.\end{aligned}$$

The add list $\mathit{add}(a_{\text{left}})$ and the delete list $\mathit{del}(a_{\text{left}})$ are defined as

$$\begin{aligned}\mathit{add}(a_{\text{left}}) :=&\{\mathit{state}(Q_2), \mathit{symbol}(C_1, S_2), \mathit{head}(C_2)\}\\ \mathit{del}(a_{\text{left}}) :=&\{\mathit{state}(Q_1), \mathit{symbol}(C_1, S_1), \mathit{head}(C_1)\}.\end{aligned}$$

Action schema (ii) is denoted by a_{right}. Its action parameters are $params(a_{\text{right}}) = \{Q_1, Q_2, S_1, S_2, C_1, C_2\}$, and its fresh variables are defined as $fresh(a_{\text{right}}) = \emptyset$. The precondition $pre(a_{\text{right}})$ is defined as follows:

$$\begin{aligned} pre(a_{\text{right}}) \coloneqq \{&state(Q_1), transition(Q_1, S_1, Q_2, S_2, \mathsf{R}), \\ &head(C_1), symbol(C_1, S_1), next(C_1, C_2)\}. \end{aligned}$$

The add list $add(a_{\text{right}})$ and the delete list $del(a_{\text{right}})$ are defined as

$$\begin{aligned} add(a_{\text{right}}) &\coloneqq \{state(Q_2), symbol(C_1, S_2), head(C_2)\} \\ del(a_{\text{right}}) &\coloneqq \{state(Q_1), symbol(C_1, S_1), head(C_1)\}. \end{aligned}$$

Finally, we detail action schema (iii), denote by $a_{\text{right-create}}$. It has the action parameters $params(a_{\text{right-create}}) = \{Q_1, Q_2, S_1, S_2, C_1\}$ and the fresh variables $fresh(a_{\text{right-create}}) = \{C_2\}$. We define the precondition $pre(a_{\text{right-create}})$ as

$$\begin{aligned} pre(a_{\text{right-create}}) \coloneqq \{&state(Q_1), transition(Q_1, S_1, Q_2, S_2, \mathsf{R}), \\ &head(C_1), symbol(C_1, S_1), right\text{-}limit(C_1)\} \end{aligned}$$

and the add list $add(a_{\text{right-create}})$ and the delete list $del(a_{\text{right-create}})$ are defined as follows:

$$\begin{aligned} add(a_{\text{right-create}}) \coloneqq \{&state(Q_2), symbol(C_1, S_2), head(C_2), \\ &right\text{-}limit(C_2), next(C_1, C_2), symbol(C_2, \square)\} \\ del(a_{\text{right-create}}) \coloneqq \{&state(Q_1), symbol(C_1, S_1), head(C_1), \\ &right\text{-}limit(C_1)\} \end{aligned}$$

Action schemas a_{left} and a_{right} are symmetric, but they move the head to different directions. Action schema $a_{\text{right-create}}$, on the other hand, is similar to a_{right} but it extends the tape to the right with the movement of the head. This is necessary to simulate the infinite tape to the right. This simulation created a new object representing the fresh cell, and it initializes it with the blank symbol $\square$.

Let $x = s_1, \ldots, s_n \in \Sigma^*$ be the input word written in the first $n \in \mathbb{N}$ cells of the TM.[4] In the initial state $I = \langle \mathcal{U}^I, \{\mathrm{c}^{\mathcal{I}}\}_{\mathrm{c} \in \mathcal{C}}, \mathcal{P}^I \rangle$, the interpretation of constants $\{\mathrm{c}^{\mathcal{I}}\}_{\mathrm{c} \in \mathcal{C}}$ maps each constant c to an object in $\mathrm{o_c} \in \mathcal{U}^I$, the interpretation $\mathcal{P}^I$ contains

- $transition(q_1, s_1, q_2, s_2, d)$ iff $\delta(q_1, s_1) = (q_2, s_2, d)$;
- $state(q_0)$;
- $symbol(c_i, s_i)$ for all $1 \leq i \leq n$;
- $next(c_i, c_{i+1})$ for all $1 \leq i < n$;

4 For simplicity and without loss of generality, we ignore the special case where $n = 0$.

Procedure 5 Compute plan for tasks with object creation

```
1: S ← ∅
2: openList ← {s_0}
3: while openList ≠ ∅ do
4:     s ← openList.Extract()
5:     if s ⊨ G then return plan
6:     for all a ∈ A do
7:         for all σ_{s,a}(a) such that s ⊨ σ_{s,a}(pre(a)) do
8:             openList ← openList ∪ {succ(s, σ_{s,a}(a))}
9: return unsolvable
```

- $\mathit{right\text{-}limit}(c_n)$; and
- $\mathit{head}(c_1)$.

The universe $\mathcal{U}^I$ contains all objects mentioned in $\mathcal{P}^I$ and all objects mapped to by constants. The goal is defined as $\{\mathit{state}(q_{accept})\}$.

The initial state of our task exactly encodes the initial configuration of M in input x: the head starts at the first cell, the state is the initial state q_0, and the input is encoded in the first cells of the tape. Observing the action schemas more closely, we can see that any reachable state s has exactly one q such that $\mathit{state}(q) \in \mathcal{P}^s$ and one c such that $\mathit{head}(c) \in \mathcal{P}^s$, because the effect of actions never add such atoms without deleting the previous one. Moreover, every cell c has exactly one symbol s such that $\mathit{symbol}(c, s) \in \mathcal{P}^s$. This means that every reachable state of the planning task encodes exactly one position for the head, one state of the TM as the current state, and assigns exactly one symbol to each one of the (finitely many) cells in the state. This is equivalent to a configuration of the TM. Thus, we have a correspondence between states of the planning task and configurations. As our initial state is the initial configuration of the TM, and the actions simulate the possible transitions between configurations of the TM, each reachable state represents a configuration that is reachable from the initial configuration. Therefore, task Π can simulate M precisely. If there exists a plan for Π, it can be converted into an accepting sequence of transitions of M. Conversely, if there is an accepting sequence of transitions, it corresponds to a plan of Π. □

Theorem 1 shows that planning with object creation is undecidable in general. However, when plans exist we can still compute them. In other words, ObjCreation-PlanEx is semi-decidable.

First, consider Procedure 5. It shows a general state-space search (without duplicate elimination). It works just the same for tasks with object creation. If openList behaves as a FIFO, then Procedure 5 is a breadth-first search. Note that although the state space is infinite, we are searching for a finitely long path in a finitely branching state space. For such scenarios, breadth-first search is semi-complete because it

considers all (finitely many) paths of length k before considering any longer path. So if a solution exists, it is found after a finite computation. This implies the following result:

Theorem 2 *Procedure 5 finds a plan in finite time for any solvable planning task with object creation.*

Together with Theorem 1, this leads to the following theorem:

Theorem 3 ObjCreation-PlanEx *is semi-decidable.*

Procedure 5 can also be extended to accommodate heuristic estimates or other optimizations.

If we are interested in plans of bounded length, the problem is decidable. To see this, we can again run breadth-first search, keeping track of the length of generated paths and rejecting the input as soon as we exceed the given bound k.

Theorem 4 ObjCreation-PlanLen *is decidable.*

Delete-Free Planning with Object Creation

We now present a special case of planning with object creation: delete-free planning. A planning task with object creation is called delete-free if all its action schemas have empty delete lists. This is analogous to our definition for planning tasks *without* object creation. In classical planning, delete-free tasks are important because they are easier to solve (Chapter 2), so they can be used to compute heuristic values (Chapter 4) and other properties (e.g., Porteous et al., 2001).

Unfortunately, delete-free planning with object creation is also undecidable. We show this using a reduction from Datalog$^{\pm}$ (Calì et al., 2010), defined next.

A *Datalog*$^{\pm}$ program $\mathcal{D}^{\pm} = \langle \mathcal{F}, \mathcal{R} \rangle$ is a Datalog program where $\mathcal{F}$ is the set of facts and $\mathcal{R}$ is the set of rules. In Datalog$^{\pm}$, rules are allowed to have existentially quantified variables in their head. More formally, a rule $r \in \mathcal{R}$ in a Datalog$^{\pm}$ program is either a Datalog rule (as in Chapter 2) or of the form

Datalog$^{\pm}$

$$\exists \boldsymbol{Y}\, q(\boldsymbol{X}, \boldsymbol{Y}) \leftarrow p_1(\boldsymbol{X}_1), \ldots, p_n(\boldsymbol{X}_n).$$

where $\boldsymbol{X} \subseteq \bigcup_{i=1}^{n} vars(\boldsymbol{X}_i)$ and $\boldsymbol{Y} \cap (\bigcup_{i=1}^{n} vars(\boldsymbol{X}_i)) = \emptyset$. The existential quantification in the head ranges over an infinite *universal domain* `Dom`.

universal domain

We extend our notation from Datalog to Datalog$^{\pm}$ programs as follows: the set $body(r)$ of atoms is the body of the rule; $head(r)$ is the head of the rule; given $r \in \mathcal{R}$, $free(r)$ denotes the set of free variables occurring in $body(r)$. Note that $head(r)$ is not an atom (as for the case of Datalog). If r has an existential quantification, the quantifier is also in $head(r)$.

A rule $r \in \mathcal{R}$ can be grounded by replacing the free variables in $\mathit{free}(r)$ with elements from $\mathtt{Dom}$. Let $\mathit{Ground}(r)$ be the (possibly infinite) set of all groundings of a rule $r \in \mathcal{R}$, and let

$$\mathit{Ground}(\mathcal{D}^{\pm}) = \bigcup_{r \in \mathcal{R}} \mathit{Ground}(r).$$

An interpretation $\mathcal{I}$ *satisfies* the ground rule r in $\mathit{Ground}(\mathcal{D}^{\pm})$ if $\mathit{body}(r)$ is true under $\mathcal{I}$. An interpretation is a *model* if $\mathcal{I} \models \mathit{head}(r)$, for every $r \in \mathit{Ground}(\mathcal{D}^{\pm})$ that is satisfied by $\mathcal{I}$. Note that this is analogous to the Datalog case, except that the grounding of the rules maps the free variables (but not the existentially quantified ones) to an infinite universal domain $\mathtt{Dom}$. This universal domain contains the set $\mathcal{C}$ of constants occurring in atoms of $\mathcal{F}$, i.e., $\mathcal{C} \subseteq \mathtt{Dom}$.

Datalog$^{\pm}$ captures the notion of value invention in logic programming (Cabibbo, 1998). The key idea is that the existentially quantified variables $\boldsymbol{Y}$ might be instantiated with new constants (similar to Skolem constants), not occurring in $\mathcal{F}$.

In contrast to Datalog, Datalog$^{\pm}$ programs do not have a unique minimal model. Note also that models can be infinite.

Example 7.6 *Consider the following Datalog$^{\pm}$ program $\mathcal{D}^{\pm} = \langle \mathcal{F}, \mathcal{R} \rangle$:*

$$\mathit{next}(0,1).$$
$$\exists Z\, \mathit{next}(Y,Z) \leftarrow \mathit{next}(X,Y).$$

The following infinite interpretation is a model of $\mathcal{D}^{\pm}$:

$$\mathcal{I}_1 = \{\mathit{next}(0,1), \mathit{next}(1,2), \mathit{next}(2,3), \mathit{next}(3,4) \dots\}.$$

But so is the following:

$$\mathcal{I}_2 = \{\mathit{next}(0,1), \mathit{next}(1,2), \mathit{next}(2,4), \mathit{next}(4,8), \dots\}.$$

Datalog$^{\pm}$ chase

The precise semantics of *Datalog$^{\pm}$* is defined according to a *chase* (Aho et al., 1979; Maier et al., 1979). Here, we consider the *oblivious* chase procedure (Calì et al., 2013). We refer to both the procedure and its output as "chase". Let D be a set of ground atoms, called a *database*, and let $\mathtt{Dom}^D \subseteq \mathtt{Dom}$ be the set of constants occurring in some atom in D. A rule r is *obliviously applicable* to D iff there exists a substitution function $\delta : \mathit{free}(r) \to \mathtt{Dom}^D$ such that $\delta(\mathit{body}(r)) \subseteq D$. Assume r is a rule that is obliviously applicable to D using a substitution function δ. The *extension* δ' of δ maps each existentially quantified variable $y \in \boldsymbol{Y}$ in $\mathit{head}(r)$ to a fresh element $\delta(y) \in \mathtt{Dom} \setminus \mathtt{Dom}^D$.

obliviously applicable

extension

The chase procedure is similar to the seminaive evaluation algorithm (Chapter 2). It iteratively checks which new atoms can be reached until a fixpoint is achieved — if such a fixpoint exists. Let $\mathcal{M}^0 := \mathcal{F}$, and define $\mathcal{M}^i$ as the set of facts reached at the i-th iteration. At iteration i, we use $D = \bigcup_{j=0}^{i-1} \mathcal{M}^j$ as our database, and the procedure

checks which rules are obliviously applicable to D. To avoid duplicates, for each obliviously applicable rule r with substitution function δ, the chase procedure enforces that $\delta(body(r)) \cap \mathcal{M}^{i-1} \neq \emptyset$. In words, at iteration i, the chase "fires" all substitutions that are obliviously applicable to the current database D and that use at least one atom from the previous iteration $\mathcal{M}^{i-1}$. Moreover, $\delta'(head(r))$ is added to $\mathcal{M}^i$ for every obliviously applicable rule r with substitution function δ and extension δ' used at iteration i. (Note that there are infinitely many different extensions δ' for one fixed δ, but we assume that the algorithm picks one arbitrarily, as they are all symmetric with respect to the names chosen for the new constants.)

It is important to observe an important feature of the oblivious chase: the extension δ' of a substitution function δ always maps existentially quantified variables to elements not occurring in the current database D, i.e., not in $\texttt{Dom}^D$. However, D could already have an atom satisfying the head. For example, consider the following example:

$$p(0).$$
$$\exists Y\, p(Y) \leftarrow p(X).$$

Initially, the chase sets $D = \mathcal{M}^0 = \mathcal{F} = \{p(0)\}$. On the first iteration, the single rule can be unified with the substitution function $\delta(X) = 0$, and the extension $\delta'(Y) = 1$. But there is already an element that satisfies the head of the rule: $Y \mapsto 0$ satisfies the head as $p(0)$ is in the database D, so we would not need another atom. However, the chase is *oblivious* to this fact (and hence its name).[5]

The chase is not guaranteed to terminate.[6] For example, the chase runs forever in the program of Example 7.6. But we can still consider its results in the limit. We then define:

$$\texttt{chase}(\mathcal{F}, \mathcal{R}) = \bigcup_{i=0}^{\infty} \mathcal{M}^i$$

Deciding if a ground atom $q(c_1, \ldots, c_n) \in \texttt{chase}(\mathcal{F}, \mathcal{R})$ is undecidable (Calì et al., 2013). Without loss of generality, we assume that $c_1, \ldots, c_n \in \texttt{Dom}^{\mathcal{F}}$.

We reduce the problem of checking if $q(c_1, \ldots, c_n) \in \texttt{chase}(\mathcal{F}, \mathcal{R})$ to the plan existence problem for delete-free planning tasks with object creation.

Theorem 5 DeleteFree-ObjCreation-PlanEx *is undecidable.*

Before diving into the proof of Theorem 5, we introduce a few useful lemmas.

5 There are other variants of the chase algorithm, such as the restricted chase (Calì et al., 2013), that do not have this property. For our purposes, the oblivious chase is enough while still being simpler than the restricted one.

6 This is true also for the restricted chase.

Assume that π^+ is a plan for a delete-free planning task with object creation Π^+. Assume also that π^+ contains two ground actions $\sigma_{s_1,a}(a)$ and $\sigma_{s_2,a}(a)$ such that

$$\sigma_{s_1,a}(v) = \sigma_{s_2,a}(v), \text{ for all } v \in \mathit{params}(a). \tag{7.1}$$

redundant actions

Ground actions $\sigma_{s_1,a}(a)$ and $\sigma_{s_2,a}(a)$ are called *redundant actions*. If we have redundant actions in π^+, we can find a new plan $\hat{\pi}^+$ where only the first of these redundant actions occur, and $|\hat{\pi}^+| < |\pi^+|$.

Lemma 6 *Let π^+ be a plan for a delete-free planning task with object creation Π^+ containing two redundant actions $\sigma_{s_1,a}(a)$ and $\sigma_{s_2,a}(a)$, where $\sigma_{s_1,a}(a)$ occurs first. Then, there exists a plan $\hat{\pi}^+$ where $\sigma_{s_2,a}(a)$ does not occur and $|\hat{\pi}^+| < |\pi^+|$.*

Proof. First, assume the simple case when $\mathit{fresh}(a) = \emptyset$. This implies that

$$\sigma_{s_1,a}(\mathit{add}(a)) = \sigma_{s_2,a}(\mathit{add}(a)).$$

As we are dealing with delete-free tasks, once we add an atom p to a state s, all atoms reached from s will contain p. Therefore, once $\sigma_{s_1,a}(a)$ is applied in π^+, applying $\sigma_{s_2,a}(a)$ does not add any new atom — they were all added by $\sigma_{s_1,a}(a)$. So $\sigma_{s_2,a}(a)$ has no impact in π^+, and simply removing $\sigma_{s_2,a}(a)$ from π^+ yields a plan $\hat{\pi}^+$.

Now, consider the case where $\mathit{fresh}(a) \neq \emptyset$. This means that the add lists are different, because the fresh variables of a must always be instantiated with different objects.

Let $\{\mathrm{o}_1^1, \ldots, \mathrm{o}_n^1\}$ and $\{\mathrm{o}_1^2, \ldots, \mathrm{o}_n^2\}$, for $n \geq 1$, be the new objects introduced by $\sigma_{s_1,a}(a)$ and $\sigma_{s_2,a}(a)$ respectively. We claim that whenever we use an object o_i^2 in π^+, we can use o_i^1 instead, for $1 \leq i \leq n$.

When $\sigma_{s_2,a}(a)$ is applied, for any atom $p(\mathrm{o}_1, \ldots, \mathrm{o}_i^2, \ldots, \mathrm{o}_m)$ added by $\sigma_{s_2,a}(a)$ there is already an atom $p(\mathrm{o}_1, \ldots, \mathrm{o}_i^1, \ldots, \mathrm{o}_m)$ in the state s_2, which was added by $\sigma_{s_1,a}(a)$ (which was applied before by definition). So in any subsequent action $\sigma_{s',a'}(a')$, for which $p(\mathrm{o}_1, \ldots, \mathrm{o}_i^2, \ldots, \mathrm{o}_m) \in \sigma_{s',a'}(\mathit{pre}(a'))$, the action is still applicable in s' if we replace o_i^2 with o_i^1, since we do not have negated atoms in the precondition.

We can then obtain a plan $\hat{\pi}^+$ by removing $\sigma_{s_2,a}(a)$ from π^+, and replacing every occurrence of the objects $\mathrm{o}_1^2, \ldots, \mathrm{o}_n^2$ created by $\sigma_{s_2,a}(a)$ with their respective objects $\mathrm{o}_1^1, \ldots, \mathrm{o}_n^1$ created by $\sigma_{s_1,a}(a)$. As just argued, the preconditions of all actions using o_i^2 are still applicable when replacing o_i^2 by o_i^1. Moreover, as the goal only mentions constants appearing in the initial state, its reachability is not affected by the removal of o_i^2. Last, as $\sigma_{s_2,a}(a)$ occurs after $\sigma_{s_1,a}(a)$ in π^+ (by assumption), the new plan is still applicable, since the first action is either $\sigma_{s_1,a}(a)$ or another action which were not modified nor removed.

As the new plan $\hat{\pi}^+$ has one action fewer than π^+, it follows directly that $|\hat{\pi}^+| < |\pi^+|$.

□

simple plan

A plan without redundant actions is called a *simple plan*.

Lemma 7 *If a delete-free planning task with object creation Π^+ is solvable, then it has a simple plan.*

Proof. Given a plan π^+ for Π^+, we can remove redundant actions one by one as described in Lemma 6, until none is left.

Note that as we remove actions, other actions might become redundant. However, as our first plan is finite, we only remove a finite number of actions from it. □

We are now ready to prove Theorem 5.

Proof of Theorem 5. Given a Datalog$^\pm$ program $\mathcal{D}^\pm = \langle \mathcal{F}, \mathcal{R} \rangle$ and a ground atom $q(c_1, \ldots, c_n)$ — where $c_1, \ldots, c_n \in \mathtt{Dom}^{\mathcal{F}}$ —, we show how to reduce the problem of checking if $q(c_1, \ldots, c_n) \in \mathtt{chase}(\mathcal{F}, \mathcal{R})$ to a delete-free planning task with object creation Π^+.

Π^+ has the same predicate symbols as $\mathcal{D}$. The set $\mathcal{C}$ of constants in Π^+ contains all constants occurring in $\mathcal{F}$.

Our task has one action schema for each $r \in \mathcal{R}$. Given a rule r in the form

$$\exists \boldsymbol{Y}\, q(\boldsymbol{X}, \boldsymbol{Y}) \leftarrow p_1(\boldsymbol{X}_1), \ldots, p_n(\boldsymbol{X}_n).$$

we introduce an action schema a_r to Π^+ where

$$\begin{aligned} \mathit{params}(a_r) &= \{X_i \mid X_i \in \boldsymbol{X}\} \\ \mathit{fresh}(a_r) &= \{Y_i \mid Y_i \in \boldsymbol{Y}\} \\ \mathit{pre}(a_r) &= \{p_i(\boldsymbol{X}_i) \mid p_i(\boldsymbol{X}_i) \in \mathit{body}(r)\} \\ \mathit{add}(a_r) &= \{q(\boldsymbol{X}, \boldsymbol{Y})\}. \end{aligned}$$

In the initial state $I = \langle \mathcal{U}^I, \{c^{\mathcal{I}}\}_{c \in \mathcal{C}}, \{\mathcal{P}^I\} \rangle$, the interpretation of constants $\{c^{\mathcal{I}}\}_{c \in \mathcal{C}}$ maps each constant c to an object $o_c \in \mathcal{U}^I$, and the interpretation $\mathcal{P}^I$ contains $p(o_{c_1}, \ldots, o_{c_n})$ iff $p(c_1, \ldots, c_n) \in \mathcal{F}$. The universe $\mathcal{U}^I$ contains all objects mentioned in $\mathcal{P}^I$.

The goal G is the singleton set $\{q(o_{c_1}, \ldots, o_{c_n})\}$.

There is an one-to-one correspondence between actions and rules, and the initial state corresponds to the initial set I of facts $\mathcal{F}$. Assume action schema a corresponds to a rule r. Then there exists a ground action $\sigma_{s,a}(s)$ applicable in a given state s iff there also exists a substitution function $\delta(\mathit{free}(r))$ such that r is obliviously applicable to the database $D = s$. It is easy to construct δ from $\sigma_{s,a}$: let $\delta(X) = \sigma_{s,a}(X)$ for all $X \in \mathit{free}(r)$. Since $\mathit{free}(r) = \mathit{params}(a)$, this is well-defined. Moreover, we can compute the extension δ' as follows: $\delta'(Y) = \sigma_{s,a}(Y)$ for all $Y \in \mathit{fresh}(a)$. Once more, this works because $\mathit{fresh}(a)$ corresponds to the existentially quantified variables in r. Thus, applying an action to a state is equivalent to an oblivious application of a rule in the chase, where the database corresponds to the state. The other way around (from obliviously applicable rules to ground actions) is analogous.

If there exists a sequence of ground rules that reaches $q(c_1, \ldots, c_n)$ in the chase, then this sequence can be transformed into a sequence of

ground actions, which corresponds to a plan, as the goal of our task is to reach $q(c_1, \ldots, c_n)$.

And if there exists a plan for Π^+, then it can be transformed into a series of rules corresponding to a derivation (see Chapter 4) of $q(o_{c_1}, \ldots, o_{c_n})$. From Lemma 7 we know that if Π^+ is solvable, it has a simple plan, and we can transform any (non-simple) plan into a simple one. So it is sufficient to consider only simple plans, which are equivalent to derivations. As our initial state exactly encodes the initial set of facts of $\mathcal{D}^{\pm}$, the goal atom $q(o_{c_1}, \ldots, o_{c_n})$ is only reachable in Π^+ (i.e., the task is solvable) iff $q(c_1, \ldots, c_n) \in \texttt{chase}(\mathcal{F}, \mathcal{R})$. □

This undecidability result might be surprising at first, as the similar problem of planning with infinitely many constants is **EXPTIME**-complete when restricted to the delete-free case and finite initial states (Erol et al., 1995). This is another extension of our original formalism. In contrast to object creation, planning with infinitely many constants already assumes the existence of an infinite number of objects at the initial state. If the initial state also has infinitely many atoms, then the plan existence problem is semi-decidable. However, Erol et al. (1991) proved that when initial states are finite, the problem is decidable. This implies that the undecidability in our formalism comes from the creation of *fresh* names on demand, and not only from the infinite state space.

Theorem 5 brings an indirect problem: computing heuristic based on relaxed plans (e.g., Chapter 4) is undecidable for tasks with object creation. Therefore, to compute good heuristics, we cannot rely on delete-relaxation alone. One alternative is to apply for additional relaxations (e.g., Lauer et al., 2021). Additionally, there are several decidable fragments based on the structure of the rules of Datalog$^{\pm}$ programs (Calì et al., 2013; Calì et al., 2010).

7.4 OVERALL PROCEDURE IN PRACTICE

There are still some details missing for a practical implementation of Procedure 5. For example, we want to quickly generate successor states, and also to define how to come up with fresh objects during object creation.

To generate successor states, we need to find all variable assignments for the action parameters leading to applicable actions. Let s be a state and a an action schema. We can find all ground actions $\sigma^1_{s,a}(a), \ldots, \sigma^m_{s,a}(a)$ by computing all $\sigma^i_{s,a}$ such that $s \models \sigma^i_{s,a}(pre(a))$. As states are finite, solving this problem is decidable.

Now assume that $p(X) \in add(a)$, where X is a fresh variable. At a given state s, we need to instantiate X to a fresh object that is not in $\mathcal{U}^s$. There are infinitely many ways to do so. The new object could be assigned to a natural number, or it could be an arbitrarily long sequence of characters, like `aaaa`, or anything else that is not in $\mathcal{U}^s$.

But all these choices are just names assigned to the new object, and they do not influence the semantics of the successor state. In other words, they are just syntactic. Any such choice of name is *isomorphic* to the other ones. Choosing one well-defined method to come up with names is sufficient. We call a function that chooses the next fresh object in a given state a *choice function*.

choice function

One of the simplest ways is to map every object $o \in \mathcal{U}^I$ to an index $\mathrm{id}(o) = i$ for $i \in \mathbb{N}$. Whenever we need a fresh object in a state s, we compute the minimum $j \in \mathbb{N}$ not assigned to any object in $\mathcal{U}^s$. We then introduce a new object named j and set $\mathrm{id}(j) = j$. In other words, new objects are identified by the minimum unused index in the current state.[7] A successor state keeps the same mapping as its parent state, besides the newly created objects. For example, if our state s has three objects, we can map them to indices 1, 2, and 3. If the action $\sigma_{s,a}$ adds a new object, this object can be assigned to index 4. The successor state $succ(s, \sigma_{s,a})$ still maps the three original objects to 1, 2, and 3, but it also maps the fourth object to 4.

This brings us to yet another efficiency concern. Assume that we have a state s and two actions a and b. Let us also assume that a and b have the trivially true precondition $pre(a) = pre(b) = \emptyset$. Moreover, $add(a) := \{p(V)\}$, while $add(b) := \{q(V)\}$, and V is a fresh variable in both actions. The sequences $\langle \sigma_{s,a}(a), \sigma_{succ(s,\sigma,a),b}(b) \rangle$ and $\langle \sigma_{s,b}(b), \sigma_{succ(s,\sigma,b,a)}(a) \rangle$ lead to two different states, but both are semantically equivalent: they only differ by the names used to identify the created objects. The two resulting states are *isomorphic*, and keeping only one of them is sufficient.

State-space search algorithms usually rely on duplicate state detection, but this is not enough here because we want to detect all *isomorphic states*. Unfortunately, no polynomial-time algorithms are known for this (Grohe and Schweitzer, 2020).

isomorphic states

This problem is similar to the one faced by orbit space search algorithms (Alkhazraji et al., 2014; Domshlak et al., 2015). In orbit space, search nodes correspond to equivalence classes of states instead of individual states. Two states are considered equivalent if they are detected to be symmetric. This symmetry detection is usually done based on *canonical states*.

canonical states

Ideally, a canonical state would be a unique representative of an equivalence class. During search, it is sufficient to store the canonical state for each encountered equivalence class and then use standard duplicate elimination techniques. The efficiency of canonical state computation is a crucial part of the performance of orbit space search planners. In practice, computing true canonical representatives is considered to be too expensive, and therefore canonical states are approx-

7 This requires keeping track of the id-value for created objects as we process the effect of an action. For a choice function where this is not necessary, see Corrêa et al. (2024b).

imated by a greedy procedure. This leads to some lost opportunities for detecting equivalence, but does not affect correctness.

In planning with object creation, one might expect symmetrical states to often occur, as the different names given to new objects are another source of symmetry. We can tackle this problem as in orbit search, by using (exact or approximate) canonical states for each equivalence class.

Example 7.7 *Let us say we have two states s_1 and s_2, and* $\mathrm{a}, \mathrm{b}, \mathrm{c}$ *and* d *are objects created during search:*

$$\mathcal{U}^{s_1} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}, \qquad \mathcal{P}^{s_1} = \{p(\mathrm{a}), p(\mathrm{c}), q(\mathrm{a}), q(\mathrm{b})\},$$
$$\mathcal{U}^{s_2} = \{\mathrm{b}, \mathrm{c}, \mathrm{d}\}, \qquad \mathcal{P}^{s_2} = \{p(\mathrm{b}), p(\mathrm{d}), q(\mathrm{b}), q(\mathrm{c})\}.$$

These states are equivalent via the object mapping $\{\mathrm{a} \mapsto \mathrm{b}, \mathrm{b} \mapsto \mathrm{c}, \mathrm{c} \mapsto \mathrm{d}\}$. *In this case, this can be already detected by a simple algorithm approximating canonical representatives by mapping each object to its index in a lexicographical order: for s_1 we map* $\{\mathrm{a} \mapsto 1, \mathrm{b} \mapsto 2, \mathrm{c} \mapsto 3\}$, *and in s_2 we map* $\{\mathrm{b} \mapsto 1, \mathrm{c} \mapsto 2, \mathrm{d} \mapsto 3\}$. *In both cases, we end up with the same state s', showing equivalence:*

$$\mathcal{U}^{s'} = \{1, 2, 3\}, \quad \mathcal{P}^{s'} = \{p(1), p(3), q(1), q(2)\}.$$

7.5 IMPLEMENTATION

We extended Powerlifted to support object creation. There a few open design choices that we discuss next.

The successor generators introduced in Chapter 3 are sufficient to compute the applicable ground actions. Recall that the precondition of an action a can be interpreted as a conjunctive query. The query is always answered with the tuples in the current state, which corresponds to the interpretation of predicates in our state. As the interpretation of predicates always refers to objects in the universe (by definition), the conjunctive query can be answered as usual. If we answer this query over a state s, every tuple in the answer corresponds to a function σ mapping $\mathit{params}(a)$ to $\mathcal{U}^s$ such that $\sigma_{s,a}(a)$ is applicable. We can still exploit the same structural properties (e.g., acyclicity) in this case.

We use the choice function mapping objects to natural numbers as described above (i.e., a new object o is mapped to the smallest natural number j such that there is no o' where $\mathrm{id}(o') = j$). We did not implement any isomorphism check between states, and we rely on syntactic duplicate detection. We leave more sophisticated techniques based on (approximate) canonical representatives as future work.

To improve the search, we modified the lifted width search from Chapter 5. Remember that a best-first width search (BFWS) (Lipovetzky and Geffner, 2017) uses a novelty measure to choose which states to expand. The novelty $w(s)$ of a state s is the size of the

smallest non-empty set of ground atoms Q such that s is the first state visited where $s \models Q$. A more informed version of novelty is $w_{\langle f_1(s),\ldots,f_n(s)\rangle}$, which is computed only considering tuples in states s' where $f_1(s) = f_1(s'), \ldots, f_n(s) = f_n(s')$. We implemented BFWS with $w_{\langle \#g\rangle}(s)$, where $\#g$ is the number of atoms in the goal satisfied in s. We only compute $w_{\langle \#g\rangle}$ up to pairs (i.e., $k = 2$). If there is no new pair in s, then $w_{\langle \#g\rangle}(s) = 3$.

Novelty measures do not seem to fit with object creation: introducing a fresh object makes the state have a novelty of 1, so BFWS always prioritizes states that create objects. In domains where the number of created of objects is unbounded, this could lead to an infinite sequence of actions. To solve this, our implementation only consider tuples of atoms that do not mention new objects. In other words, we compute the novelty of a state over those tuples that only mention objects in $\mathcal{U}^I$. However, this has the opposite effect: new objects do not account for the novelty of a state, so they do not add any information to the BFWS. What can happen in this case is that BFWS finds plans creating the minimum number of objects necessary. As we see in our experimental results next, this does indeed often happen in our benchmarks. Yet, this modified BFWS still improves our planner.

It would be interesting to extend our delete-relaxation heuristics (Chapter 4) to object creation, but this is not a simple task and requires developing its own theory. As argued before, since delete-free planning with object creation is undecidable, we need to either exploit decidable fragments or to find further relaxations that yield decidable problems.

7.6 EXPERIMENTAL RESULTS

For our experiments, we introduced new domains that contain object creation. So in this chapter, we do not rely on the IPC and HTG sets used so far. Instead, we use four new PDDL domains with object creation as our benchmark. Two of them are based on previously existing domains that encode object creation by listing all possible objects at the initial state and using auxiliary predicates to simulate object creation. This made it necessary in the original PDDL to introduce a bound on the number of objects that can be created and then experiment with different bounds to deal with tasks that are wrongly considered unsolvable because the number of objects is too low.

For each domain, we have two versions:

- one using our new PDDL extension, called the *extended version*; *extended version*
- one where all potential objects are declared in the initial state, called the *standard version*. *standard version*

This comparison is imperfect because only the extended version captures the underlying problem faithfully, but it allows us to compare our planner using object creation with existing planning systems.

Cluster Management

CLUSTER MANAGEMENT In this new domain, we must produce a set of files. These files are produced by executing scripts on certain inputs. For example, executing script S with input I_1 might output O_1, and executing S with I_2 might output O_2. Our benchmark contains instances with up to 100 files and 20 different scripts. We also have a cluster with multiple CPUs, where we can load and execute these scripts with the corresponding files. The actions are to load a file or script in a CPU, to execute a script, to save a file into memory, and to add a new CPU to the cluster. So if a script must be used multiple times (with different inputs to produce different outputs), it might be preferable to load it only once in one of the CPUs and leave it there. The problem in this domain is to find the optimal amount of CPUs to obtain a certain set of goal files as quick as possible. We can add new CPUs to our cluster using an action that creates a new CPU object. In the standard version, we pre-declare 5 CPUs.

Commutative Rings

COMMUTATIVE RINGS This domain was introduced by Petrov and Muise (2023). A task in this domain is a statement in elementary algebra. A plan is a proof for this statement. The domain focuses on tasks related to commutative rings. One task, for example, is to prove that for all commutative rings R, $a \times 0 = 0$ for every $a \in R$. Action schemas represent the axioms of commutative rings, equality operations, definitions of products, sums, and inverses. Object creation can be used to model existential axioms and build complex expressions. For instance, given a commutative ring R, for any $a, b \in R$ there exists an element $a + b \in R$. So we can construct a new object c and define $c = a + b$ to be used later. In the original domain, Petrov and Muise (2023) introduce a fixed number of undeclared variables in the initial state. However, this makes the task harder to ground, while also bounding the number of proofs the planner can explore. We used the original tasks of this domain, but compiled away conditional effects, which are not supported by Powerlifted.

Logistics Company

LOGISTICS COMPANY This is a new domain, and it is similar to the running example in our paper. We have a set of connected locations and a set of packages that must be delivered to specific locations. Our company has headquarters in a few locations, and we can buy trucks that appear at one of these headquarters. Actions are to move a truck, (un)load packages, and buy a new truck. The challenge is to find a good balance of how many trucks to buy to deliver all packages efficiently. While all tasks are solvable with one single truck, it might be that using multiple trucks decreases the plan length. In our instances, the number of locations varies between 3 and 1 000, the number of packages from 1 to 100, and the number of headquarters from 1 to 20. In the standard version, the number of declared trucks at the initial state is twice the number of headquarters in the task.

	PWL^{++}		PWL		FD	
	B	W	B	W	B	W
Cluster Management (20)	3	9	2	**14**	5	12
Commutative Ring (15)	2	11	9	**14**	10	12
Logistics Company (20)	3	**19**	5	8	5	6
Settlers (20)	3	**8**	3	6	3	5
Total (75)	11	**47**	19	42	23	35

Table 7.1: Coverage of PWL^{++}, PWL, and FD on our benchmark. For each planner, we tested a configuration with breadth-first search (B) and best-first width search (W).

Settlers

SETTLERS This domain is based on the Settlers domain used in IPC 2002 (Long and Fox, 2003). The domain focuses on resource management. Products and factories must be built from raw materials and used in the manufacturing or transportation of further materials. The objective is to construct a variety of building types at various specified locations. The original domain is numeric: the quantity of resources at each location is defined by numeric fluents. But these fluents are discrete and their maximum values are always bounded, so one can emulate them using predicates to encode a successor relation over the natural numbers. Long and Fox (2003) mention that this domain highlights the necessity of object creation during plan execution. In the standard version, all objects had to be declared in advance, which made grounding harder and made the modeling more convoluted. We removed "maritime" objects – wharves, docks, ships – in our version, because the original instances were too challenging for all planners.

Results

We ran Powerlifted on both versions of our benchmarks (standard and extended versions). Powerlifted using the standard versions is denoted as PWL, and using the extended versions as PWL^{++}. For each, we tested two configurations: a breadth-first search (BFS), and the BFWS as explained in the previous section.

Table 7.1 shows the coverage of all methods in our benchmark. PWL^{++} outperforms PWL when using BFWS, but it has lower coverage with BFS. Using BFS, PWL^{++} might create too many objects. Two of the domains (Cluster Management and Logistics Company) have actions that create new objects and are applicable to every state. Therefore, the branching factor increases at each layer of the search, which hurts

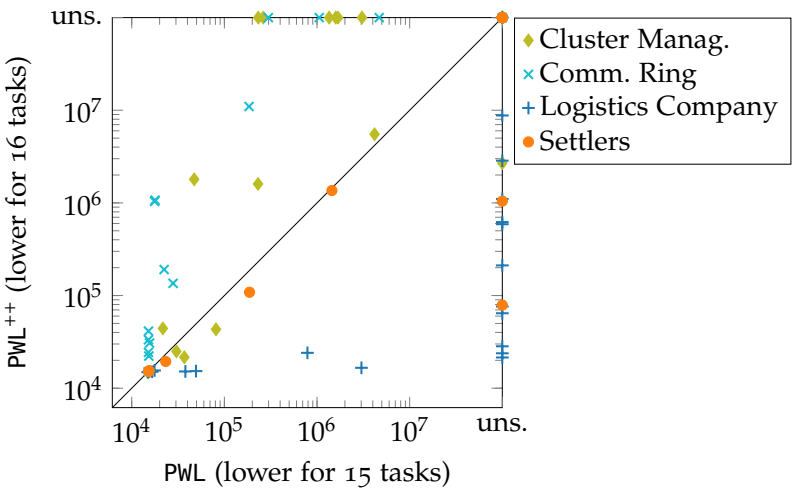


Figure 7.1: Memory consumption using PWL and PWL^{++}.

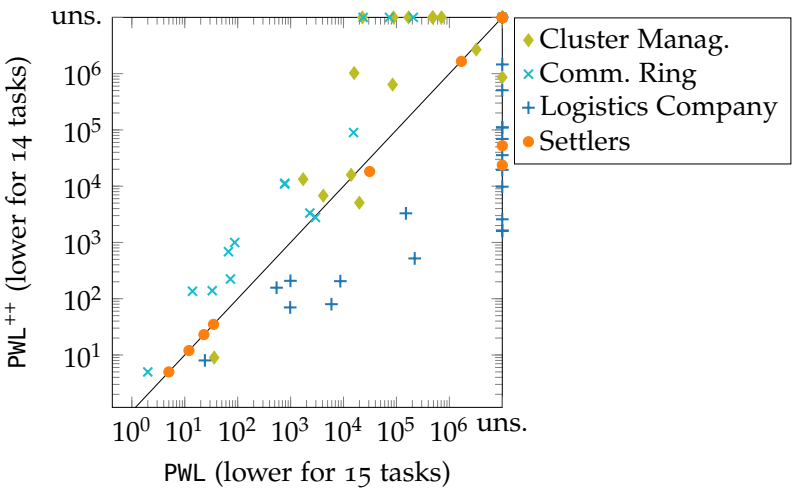


Figure 7.2: Expanded states using PWL and PWL^{++}.

performance. In contrast, the BFWS implementation does not consider tuples considering newly created objects. This can be an issue with the extended versions, as only some objects are considered when evaluating the novelty of a state. But this is not always problematic. In the Logistics Company domain, for example, using one truck is already enough to solve the task so BFWS does not favor the creation of new trucks. This increases coverage in the extended version but also impacts plan quality. In some cases, the optimal plan had length 10, but PWL^{++} with BFWS only found plans with more than 100 steps.

With respect to run time and memory, there are large differences depending on the domain. Figure 7.1 compares the memory usage for PWL^{++} and PWL, both using the BFWS configuration. In a few instances of the Commutative Rings domain, PWL^{++} with BFWS used almost 100 times more memory than the PWL counterpart. In this domain, some instances in the standard version do not use any undeclared variables. In the extended version, the planner is not aware that they are not needed, so they are introduced multiple times. This blows up the size of the state space, increasing the number of expansions and, consequently, time and memory. To solve this problem it is crucial to have heuristic estimates that can better decide when more objects are needed. On the other hand, we observe the opposite behavior in the Logistics company domain, where sometimes PWL with BFWS consumes almost 100 times more memory. As mentioned before, we can solve any problem with a single truck, so PWL^{++} with BFWS always uses one truck, while PWL already has more trucks at the initial state, so the branching factor is larger already at the initial state.

Figure 7.2 compares the number of expanded states for the same two methods. As expected, in the domains where PWL^{++} consumes more memory, it also expands more states. However, the differences between PWL^{++} and PWL are not as large as we expected (as in some instances the memory differed by a factor of almost 100).

Powerlifted has some advantages when solving tasks that are hard to ground, but its search capabilities are not on par with ground planners (see Part i). To compare PWL^{++} with a state-of-the-art heuristic search planner, we also ran Fast Downward (Helmert, 2006) on the standard versions. We denote Fast Downward as FD in the rest. Results are also shown in Table 7.1.

When using BFS, FD is significantly better than PWL^{++} and somewhat better than PWL. For larger problems, however, grounding becomes an issue for FD. This can be seen when comparing FD with BFWS and PWL^{++} with BFWS. Although the coverage of FD increases by almost 50% when switching from BFS to BFWS, it is still worse than PWL^{++}. Only in the Cluster Management domain, FD outperforms PWL^{++}. This is also the only domain where all tasks can be grounded within seconds, as the number of ground actions is low (a few thousands) even declaring all objects in advance. In the Logistics Company domain, where several

of the declared objects are not necessary (although helpful) grounding becomes a major bottleneck. In this domain, FD has worse coverage than PWL^{++}. The same happens in the Commutative Rings domain, even though in this domain we have at most one undeclared variable per task — but in the grounding several actions use this object. As noted by Petrov and Muise (2023), adding one single undeclared variable to the initial state is already enough to make the grounding much harder.

We also analyzed the number of expansions for each method. The results vary with the domains. For example, PWL^{++} expands fewer states than PWL and FD in the Logistics Company and Cluster Management domains, because the search in PWL^{++} is guided directly to a goal state using the minimum amount of objects. However, the plan found by PWL^{++} is usually longer than the ones found by PWL and FD. In the Settlers domain, all methods have a similar number of expansions. As object creation is restricted in this domain (depends on the resources available at the state), it is not so impactful in the performance.

7.7 SUMMARY

We formalized an extension of classical planning that allows for object creation and removal during plan execution. In our formalism, this creation happens as part of action effects. In general, planning with object creation is semi-decidable, even if we consider common restrictions such as delete-free planning.

Our main insight in this chapter is that lifted planners are well-suited for object creation. We implemented support for object creation on top of Powerlifted, and showed that, at least for our STRIPS-like fragment, there are only a few things that needed adaptation. In our experimental results, support for this extension caused no harm to the planner performance. It was also on par with Fast Downward.

It would be useful to study how to efficiently identify isomorphic states to reduce the search space. To further scale performance, one could use more informed heuristic estimates for tasks with object creation and have a more sophisticated integration of width-based search with object creation.

CHAPTER NOTES & HISTORY

Object creation has been considered several times as an important feature for large-scale planning systems (Hoffmann et al., 2009; Long and Fox, 2003; Petrov and Muise, 2023). So far, all methods trying to solve this problem used compilations. Fuentetaja and de la Rosa (2016) present an automatic compilation of "irrelevant objects" (i.e., objects whose name do not specifically matter) into counters. While this is sufficient in certain domains, it has inherent limitations. Two irrelevant

objects can only be compiled into the same counter if they are fungible and can be easily interchanged in plans. In our logistics company example this is not the case, as trucks can have different properties in a plan (e.g., each truck can be at a different location and carry a different set of packages). Moreover, counters must be first compiled into relations, which forces them to be bounded. Similar compilations have been proposed to automatically encode indistinguishable objects into counters represented by numeric variables (Riddle et al., 2016).

Edelkamp et al. (2019) encode planning with object creation as a model checking problem. Their method seems more flexible than the one by Fuentetaja and de la Rosa, as the creation is not restricted to only a few objects. Edelkamp et al. also propose the idea of object removal. However, the semantics of both object creation and removal are not fully specified in their work.

In situation calculus (McCarthy, 1963), infinitely many objects have already been considered (cf. Reiter, 2001), although mostly in theory. In comparison to our work, the most relevant result in situation calculus is the work by De Giacomo et al. (2016). They show that bounded situation calculus – when the number of objects in tuples is bounded – is decidable. This is essentially the same as planning with object creation for tasks with a bounded number of objects allowed in the interpretation of predicates at any given state. Moreover, De Giacomo et al. also show that verification of an expressive class of first-order μ-calculus temporal properties in bounded action theories is **EXPTIME**-complete.

Our original work (Corrêa et al., 2024a) used a richer fragment than our STRIPS-like formalism from this chapter. There, we considered all PDDL features of classical planning — arbitrary preconditions, universally quantified effects, conditional effects, etc. We also showed how to implement *object removal*. We did not include this more powerful fragment in the chapter, as our implementation only considered STRIPS anyway. To the best of our knowledge, our original paper was the first one to fully specify the complete semantics of object creation — although Hoffmann et al. (2009) study a fragment that subsumes STRIPS but has orthogonal features such as indirect effects. Our proof of undecidability in the paper is equivalent to the one presented here, and it follows the classic structure of using new objects to extend the tape of a Turing Machine (Hoffmann et al., 2009; Reiter, 2001). The result that delete-free planning with object creation is undecidable is a novel contribution of this thesis. The original paper did not have any results about delete-free tasks; however, it did have a longer discussion about the decidability of state-bounded tasks.

object removal

Part IV

CONCLUSION

8

CONCLUSION

In Part i, we implemented Powerlifted, a heuristic search planner that works directly on a lifted representation of planning tasks. We started by studying how to generate successor states during a state-space search and showed how to cast this problem into a conjunctive query answering problem. Then, using insights from database theory (Chandra and Merlin, 1977; Codd, 1970; Yannakakis, 1981), we showed how to exploit the structure of action schemas — i.e., acyclicity — to implement efficient successor generators.

Later, we also studied how to extract heuristics from the lifted representation. Our study focused on delete-relaxation heuristics, and we showed how to compute lifted versions of h^{FF} (Hoffmann and Nebel, 2001), h^{add}, and $h^{\max}$ (Bonet and Geffner, 2001). By reducing our delete-relaxed task into a Datalog program, we could couple several optimizations (e.g., annotations, rule decomposition, predicate removal) to efficiently compute these heuristics. Additionally, we implemented more recent techniques, such as BFWS (Francès et al., 2017; Lipovetzky and Geffner, 2017), that helped us boost the performance of Powerlifted.

Our final configuration of Powerlifted achieves state-of-the-art performance among lifted planners. In tasks that are hard to ground, Powerlifted can solve more problems than any other planner, while still being competitive with state-of-the-art ground planners in tasks where grounding is not so challenging.

In Part ii we translated the knowledge obtained from building Powerlifted to the ground planning setting. We showed how to use several techniques from Powerlifted to speed up the grounding of planning tasks.

Using external tools such as `gringo` (Gebser et al., 2011) and `lpopt` (Bichler et al., 2016), we created a new grounder that works in two steps. First, we compute all relaxed reachable atoms of the task, and then we use this set of atoms to compute the actions. While being slower, this new technique has a wider reach than the traditional grounding approaches in planning. However, due to its overhead, our two-step grounder did not help much when simply considering coverage.

In our last endeavor, we extended the classical planning formalism to allow object creation (Part iii). In this fragment, actions can create fresh objects as part of their effect. This extension led us to a semi-decidable flavor of classical planning.

However, we showed that Powerlifted — and, potentially, any lifted planner using state space search — can easily deal with this extension. We implemented support for object creation in Powerlifted and showed that this usually does not add any overhead. Furthermore, we illustrated how width-based search can also be modified to support object creation, which also helped the performance of Powerlifted in these new domains.

OPEN QUESTIONS & FUTURE WORK

We now highlight some ideas for future work that were left unanswered by this thesis.

Treewidth Decompositions in Lifted Planning

In Chapter 6 we showed that decomposing Datalog rules using tree decompositions (computed by `lpopt`) helped `gringo` to ground more tasks. This occurred in the so-called simplified Datalog programs, where action predicates have been removed. In Powerlifted, we used the same Datalog programs to compute delete-relaxation heuristics (Chapter 4). However, there we used the method by Helmert (2009) to decompose the rules (equivalent to the $\texttt{FD}^{++}$ decomposition in Chapter 6). A possibility is to combine the insights from Chapter 6 and use `lpopt`'s tree decomposition within Powerlifted to compute h^{add}, h^{FF}, and $h^{\max}$. We expect this to produce a similar improvement as in the grounder.

We can also use tree decomposition in our successor generators (Chapter 3). Recall that, for cyclic action schemas, we used a heuristic method to avoid large intermediate results. We can replace this heuristic method with a decomposition method based on tree decomposition, as answering conjunctive queries with treewidth tw is exponential only in tw. Moreover, Longo (2023) showed that all action schemas in the IPC and HTG sets have very low *hypertreewidth* (Gottlob et al., 2002) — never higher than 2. While hypertree decompositions were tested as a replacement for `lpopt` in our grounding algorithms without much success (Longo, 2023), they could work better for the successor generation case.

More Refined Grounders

Our experiments on Chapter 6 also showed that, unfortunately, our grounding via iterated solving is very close to its practical limits,

as most of the unsolved tasks have a prohibitive number of action schemas and, therefore, just storing them is impractical. A possible way out is to refine our grounder even further, potentially adding more steps to prune some of these actions.

A direct option is to include a backward reachability analysis phase (Helmert, 2009). Backward reachability analysis prunes actions and atoms that do not help find a goal state. The classic example is a Logistics task with many packages, but where some packages are not mentioned in the goal. In this case, there is no reason to keep information of these packages, so atoms and actions mentioning them can be removed. This is what backward reachability analysis does. We can try to exploit this idea using additional rules or using a new step in our grounder.

This seems related to the concept of *magic sets* (Bancilhon et al., 1986). The idea of magic sets is to rewrite the Datalog rules so that a bottom-up evaluation (e.g., seminaive evaluation from Chapter 2) does not consider irrelevant facts that are generated. In practice, magic set rewriting prunes atoms that would not be considered by a top-down evaluation (e.g., Prolog back-chaining). This forces the evaluation to be more focused on atoms that influence the goal, which might lead to improvement in performance.

Delete-Relaxation Heuristics for Object Creation

We showed in Chapter 7 that delete-free planning with object creation is semi-decidable. This is bad news for most delete-relaxation heuristics, which need to compute relaxed plans in the evaluated states. There are simple workarounds to this problem — for example, check for relaxed plans of bounded lengths — but we might be able to do better than this.

We can use some insights from our theoretical results to help us come up with decidable heuristic functions. For example, we can further relax our problem by assuming that all created objects are homomorphic, or refine this relaxation all objects created by a same action schema are homomorphic. In practice, this allows us to use only one new object (in total or per action schema), and whenever we have an object creation effect, we reuse the same object. This brings us back into a decidable case — the total number of objects is now bounded — while still giving us a distance estimate. This proposal is not far from the work by Horčík et al. (2022) discussed in the notes of Chapter 4, but here we exploit homomorphisms just for the fresh objects.

Another possibility is to look for decidable fragments of delete-free planning with object creation. One possible way is to first look for cases where atom containment in the chase is decidable, as we demonstrated that delete-free planning with object creation and the chase procedure are closely related. There is an extensive body of work

introducing fragments of Datalog$^{\pm}$ for which the chase is guaranteed to terminate (Calì et al., 2013; Calì et al., 2010). These fragments are usually based on the structure of the rules (e.g., guarded or linear rules). It would be an interesting first step to study if, when we encode our planning tasks as Datalog$^{\pm}$ programs, these programs fall into the decidable cases. If so, we can try to adapt the algorithms to compute models for Datalog$^{\pm}$ programs to extract heuristics, similarly to what we did in Chapter 4.

APPENDIX

A

COMPUTATIONAL COMPLEXITY REDUX

We present the necessary concepts in computational complexity next. We assume familiarity with general notions of theoretical computer science, such as Turing Machines (TMs), nondeterminism, and asymptotic functions. For a more detailed overview of complexity theory and related concepts, we recommend the textbook by Wigderson (2019).

To study how hard a problem is, we usually rely on its *decision problem*: a yes-or-no version of it. For example, *"Given a Boolean formula ϕ, is ϕ satisfiable?"* and *"Given a graph G, is G connected?"* are decision problems. A particular formula ϕ or a particular graph G are called *instances*. We say that an algorithm correctly decides or *solves* a decision problem if it correctly outputs yes or no for any given instances.

decision problem

Example A.1 *The decision problem* SAT *is defined as*

INPUT*: A Boolean formula ϕ.*
QUESTION*: Is ϕ satisfiable?*

The concrete input formula $\phi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_3 \vee x_4) \wedge (\overline{x_3} \vee \overline{x_4})$ is an instance of SAT*, and the following (nondeterministic) algorithm solves the problem:*

1. *Guess an assignment of true or false to each variable in ϕ;*
2. *Check if ϕ is satisfied by the assignment or not. If it is, return yes; otherwise, return no.*

Decision problems are split in different *classes*, based on the amount of resources an algorithm needs to solve an instance of this problem. We are particularly interested in two specific resources: time and memory. An algorithm for a given decision problem runs in *polynomial time* if it solves any instance of the corresponding decision problem in $O(n^c)$ steps, where c is some real-valued constant and n is the *input size* of the instance. This input size is simply the number of bits necessary to represent the instance in the input tape of a TM. We assume reasonable compact encodings of instances, and we denote the input size of an instance P as $\|P\|$.

polynomial time

Analogously, a problem runs in *polynomial space* if, for any given

polynomial space

instance, it solves the problem using at most $O(n^c)$ memory cells.[1] More generally, a problem is said solvable in $f(n)$ time (resp. space), if there is an algorithm that solves it in $O(f(n))$ steps (resp. cells).

time complexity

Decision problems can be classified in different classes by their *time complexity*. The class **P** contains all problems for which a polynomial-time *deterministic* algorithm exists, while the class **NP** contains all problems for which a polynomial-time *nondeterministic* algorithm exists. It is clear that $\mathbf{P} \subseteq \mathbf{NP}$, but we do not know if the inclusion is proper. Although most researchers believe that the inclusion is indeed proper (i.e., $\mathbf{P} \subsetneq \mathbf{NP}$) the question has been open for decades (Cook, 1971).

Example A.2 SAT *is in* **NP**, *as there is a polynomial-time nondeterministic algorithm for it (Example A.1). We do not know if* SAT *is in* **P**.

exponential time

Similarly, the class **EXPTIME** contains all problems for which an *exponential-time* deterministic algorithm exists; the class **NEXPTIME** corresponds to its nondeterministic counterpart. As for **P** and **NP**, we know that $\mathbf{EXPTIME} \subseteq \mathbf{NEXPTIME}$, but we do not know if the inclusion is proper.

Example A.3 *Let* Chess *be the problem of deciding whether the White pieces have a winning strategy in a given (generalized) chess position on an $n \times n$ board.* Chess *is in* **EXPTIME** *(Fraenkel and Lichtenstein, 1981).*

space complexity

Savitch's theorem

Problems can also be categorized according to their *space complexity*: **PSPACE** contains all problems solvable by a deterministic algorithm running in polynomial space; **EXPSPACE** contains all problems solvable by a deterministic algorithms running in exponential space. But what about their nondeterministic counterparts, **NPSPACE** and **NEXPSPACE**? Perhaps surprisingly, Savitch (1970) showed that nondeterminism does not add power to **PSPACE** and **EXPSPACE** — i.e., $\mathbf{PSPACE} = \mathbf{NPSPACE}$ and $\mathbf{EXPSPACE} = \mathbf{NEXPSPACE}$.[2] However, we do not know much about the relations between space complexity and time complexity classes.

Our current understanding about all these classes is the following:

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXPTIME} \subseteq \mathbf{NEXPTIME} \subseteq \mathbf{EXPSPACE}.$$

We do know, however, that $\mathbf{P} \subset \mathbf{EXPTIME}$ (Hartmanis and Stearns, 1965) and $\mathbf{NP} \subset \mathbf{NEXPTIME}$ (Cook, 1973). This means that there are problems solvable in (non)deterministic exponential time that cannot be solved in (non)deterministic polynomial time. The open-questions related to **P**, **NP**, **EXPTIME**, and **NEXPTIME** then ask if

1 Ignoring the memory used to represent the input. This is not important for the specific case of polynomial-space algorithms, but it becomes relevant once we restrict the algorithms to use logarithmic space, for example.

2 Savitch's theorem is actually stronger than this statement but these two results are enough for this thesis.

nondeterminism adds any computational power with respect to run time. Moreover, Stearns et al. (1965) show that **PSPACE** $\subset$ **EXPSPACE**.

hardness

For any class C, a problem is said *C-hard* if it is at least as hard as any other problem in C. Proving hardness of a problem P is usually done via *reduction*: starting from a known C-hard problem P', we show how to reduce in polynomial time any instance of P' to an instance of P, such that all yes/no instances are mapped accordingly. Intuitively, if we can solve P efficiently, we can also solve P' efficiently by reducing its instances to instances of P and then solving them. A problem is said *complete* for C if it is both in C and C-hard.

completeness

Example A.4 SAT *is* **NP***-complete (Cook, 1971).*

If there exists an efficient algorithm to solve SAT, then every problem in **NP** can be solved efficiently by first reducing it to SAT and then calling the previous efficient algorithm.

BIBLIOGRAPHY

Mohammad Abdulaziz, Florian Pommerening, and Augusto B. Corrêa (2022). "Mechanically Proving Guarantees of Generalized Heuristics: First Results and Ongoing Work." In: *IJCAI 2022 Workshop on Generalization in Planning*.

Serge Abiteboul, Richard Hull, and Victor Vianu (1995). *Foundations of Databases*. Addison-Wesley (cit. on pp. 11, 22, 24, 31).

Alfred V. Aho, Catriel Beeri, and Jeffrey D. Ullman (1979). "The Theory of Joins in Relational Databases." In: *ACM Transactions on Database Systems* 4.3, pp. 297–314 (cit. on p. 130).

Yusra Alkhazraji, Michael Katz, Robert Mattmüller, Florian Pommerening, Alexander Shleyfman, and Martin Wehrle (2014). "Metis: Arming Fast Downward with Pruning and Incremental Computation." In: *Eighth International Planning Competition (IPC-8): Planner Abstracts*, pp. 88–92 (cit. on p. 135).

Carlos Areces, Facundo Bustos, Martín Ariel Dominguez, and Jörg Hoffmann (2014). "Optimizing Planning Domains by Automatic Action Schema Splitting." In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*. Ed. by Steve Chien, Alan Fern, Wheeler Ruml, and Minh Do. AAAI Press, pp. 11–19 (cit. on pp. 21, 44, 115).

Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski (1987). "Complexity of Finding Embeddings in a k-Tree." In: *SIAM J. Algebraic Discrete Methods* 8.2, pp. 277–284 (cit. on pp. 94, 98, 99).

Masataro Asai and Alex Fukunaga (2017). "Exploration Among and Within Plateaus in Greedy Best-First Search." In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*. Ed. by Laura Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith. AAAI Press, pp. 11–19 (cit. on p. 89).

François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman (1986). "Magic Sets and Other Strange Ways to Implement Logic Programs." In: *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS 1986)*. ACM, pp. 1–15 (cit. on p. 149).

Philip A. Bernstein and Nathan Goodman (1981). "Power of Natural Semijoins." In: *SIAM Journal on Computing* 10.4, pp. 751–771 (cit. on pp. 24, 26, 27).

Viktor Besin, Markus Hecher, and Stefan Woltran (2022). "Body-Decoupled Grounding via Solving: A Novel Approach on the ASP Bottleneck." In: *Proceedings of the 31st International Joint Conference on Artificial Intelligence (IJCAI 2022)*. Ed. by Luc De Raedt. IJCAI, pp. 2546–2552 (cit. on pp. 4, 5, 94, 107).

Christoph Betz and Malte Helmert (2009). "Planning with h^+ in Theory and Practice." In: *ICAPS 2009 Workshop on Heuristics for Domain-Independent Planning (HDIP)*, pp. 64–69 (cit. on p. 16).

Wolfgang Bibel (1986). "A Deductive Solution for Plan Generation." In: *New Generation Computing* 4.2, pp. 115–32 (cit. on p. 16).

Manuel Bichler, Michael Morak, and Stefan Woltran (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)*. Springer, pp. 114–130 (cit. on pp. 4, 5, 63, 94, 100, 147).

Arthur Bit-Monnot (2018). "A constraint-based encoding for domain-independent temporal planning." In: *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming*. Ed. by John Hooker. Springer-Verlag, pp. 30–46 (cit. on p. 44).

Bernhard Bliem, Michael Morak, Marius Moldovan, and Stefan Woltran (2020). "The Impact of Treewidth on Grounding and Solving of Answer Set Programs." In: *Journal of Artificial Intelligence Research* 67, pp. 35–80 (cit. on pp. 4, 105).

Avrim Blum and Merrick L. Furst (1997). "Fast Planning Through Planning Graph Analysis." In: *Artificial Intelligence* 90.1–2, pp. 281–300 (cit. on p. 74).

Miquel Bofill, Joan Espasa, and Mateu Villaret (2016). "The RANTANPLAN planner: system description." In: *The Knowledge Engineering Review* 31.5, pp. 452–464 (cit. on p. 44).

Blai Bonet and Héctor Geffner (2001). "Planning as Heuristic Search." In: *Artificial Intelligence* 129.1, pp. 5–33 (cit. on pp. 4, 15, 17, 18, 47–49, 80, 93, 147).

Clemens Büchner, Remo Christen, Augusto B. Corrêa, Salomé Eriksson, Patrick Ferber, Jendrik Seipp, and Silvan Sievers (2023). "Fast Downward Stone Soup 2023." In: *Tenth International Planning Competition (IPC-10): Planner Abstracts*.

Tom Bylander (1994). "The Computational Complexity of Propositional STRIPS Planning." In: *Artificial Intelligence* 69.1–2, pp. 165–204 (cit. on pp. 15, 48).

Luca Cabibbo (1998). "The Expressive Power of Stratified Logic Programs with Value Invention." In: *Information and Computation* 147.1, pp. 22–56 (cit. on p. 130).

Andrea Calì, Georg Gottlob, and Michael Kifer (2013). "Taming the Infinite Chase: Query Answering under Expressive Relational Constraints." In: *Journal of Artificial Intelligence Research* 48, pp. 115–174 (cit. on pp. 130, 131, 134, 150).

Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, and Andreas Pieris (2010). "Datalog+/-: A Family of Languages for Ontology Querying." In: *Datalog Reloaded - First International Workshop (Datalog 2010)*. Ed. by Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers. Springer, pp. 351–368 (cit. on pp. 129, 134, 150).

Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari (2017). "I-DLV: The new intelligent grounder of DLV." In: *Intelligenza Artificiale* 11.1, pp. 5–20 (cit. on p. 116).

Ashok K. Chandra and Philip M. Merlin (1977). "Optimal implementation of conjunctive queries in relational databases." In: *Proceedings of the 9th Annual ACM Symposium on the Theory of Computing (STOC 1977)*. Ed. by John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison. ACM, pp. 77–90 (cit. on pp. 4, 23, 24, 147).

Edgar F. Codd (1970). "A relational model of data for large shared data banks." In: *Communications of the ACM* 13.6, pp. 377–387 (cit. on pp. 4, 24, 147).

Stephen A. Cook (1971). "The complexity of theorem-proving procedures." In: *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC 1971)*. Ed. by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman. ACM, pp. 151–158 (cit. on pp. 154, 155).

Stephen A. Cook (1973). "A Hierarchy for Nondeterministic Time Complexity." In: *Journal of Computer and System Sciences* 7.4, pp. 343–353 (cit. on p. 154).

Augusto B. Corrêa (2019). "Planning using Lifted Task Representations." MA thesis. University of Basel (cit. on p. 32).

Augusto B. Corrêa (2024). *Code from the Ph.D. thesis "Planning with Different Representations"*. https://doi.org/10.5281/zenodo.12706513 (cit. on p. 5).

Augusto B. Corrêa, Clemens Büchner, and Remo Christen (2023a). "Zero-Knowledge Proofs for Classical Planning Problems." In: *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence (AAAI 2023)*. Ed. by Yiling Chen and Jennifer Neville. AAAI Press, pp. 11955–11962.

Augusto B. Corrêa and Giuseppe De Giacomo (2024). "Lifted Planning: Recent Advances in Planning Using First-Order Representations." In: *Proceedings of the 33rd International Joint Conference on Artificial Intelligence (IJCAI 2024)*. Ed. by Kate Larson. IJCAI, pp. 8010–8019.

Augusto B. Corrêa, Giuseppe De Giacomo, Malte Helmert, and Sasha Rubin (2024b). *Planning with Object Creation – Technical Report*. Tech. rep. CS-2024-002. University of Basel, Department of Mathematics and Computer Science (cit. on p. 135).

Augusto B. Corrêa, Giuseppe De Giacomo, Malte Helmert, and Sasha Rubin (2024a). "Planning with Object Creation." In: *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024)*. Ed. by Sara Bernardini and Christian Muise. AAAI Press, pp. 104–113 (cit. on p. 143).

Augusto B. Corrêa, Guillem Francès, Markus Hecher, Davide Mario Longo, and Jendrik Seipp (2023b). "Levitron: Combining Ground and Lifted Planning." In: *Tenth International Planning Competition (IPC-10): Planner Abstracts*.

Augusto B. Corrêa, Guillem Francès, Markus Hecher, Davide Mario Longo, and Jendrik Seipp (2023c). "Scorpion Maidu: Width Search in the Scorpion Planning System." In: *Tenth International Planning Competition (IPC-10): Planner Abstracts* (cit. on pp. 77, 89).

Augusto B. Corrêa, Guillem Francès, Markus Hecher, Davide Mario Longo, and Jendrik Seipp (2023d). "The Powerlifted Planning System in the IPC 2023." In: *Tenth International Planning Competition (IPC-10): Planner Abstracts*.

Augusto B. Corrêa, Guillem Francès, Florian Pommerening, and Malte Helmert (2021). "Delete-Relaxation Heuristics for Lifted Classical Planning." In: *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*. Ed. by Robert P. Goldman, Susanne Biundo, and Michael Katz. AAAI Press, pp. 94–102 (cit. on pp. 43, 75).

Augusto B. Corrêa, Markus Hecher, Malte Helmert, Davide Mario Longo, Florian Pommerening, and Stefan Woltran (2023e). "Grounding Planning Tasks Using Tree Decompositions and Iterated Solving." In: *Proceedings of the Thirty-Third International Conference on Automated Planning and Scheduling (ICAPS 2023)*. Ed. by Sven Koenig, Roni Stern, and Mauro Vallati. AAAI Press, pp. 100–108 (cit. on p. 116).

Augusto B. Corrêa, Florian Pommerening, Malte Helmert, and Guillem Francès (2020). "Lifted Successor Generation using Query Optimization Techniques." In: *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*. Ed. by J. Christopher Beck, Erez Karpas, and Shirin Sohrabi. AAAI Press, pp. 80–89 (cit. on pp. 6, 43).

Augusto B. Corrêa, Florian Pommerening, Malte Helmert, and Guillem Francès (2022). "The FF Heuristic for Lifted Classical Planning." In: *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022)*. Ed. by Vasant Honavar and Matthijs Spaan. AAAI Press, pp. 9716–9723 (cit. on p. 75).

Augusto B. Corrêa and Jendrik Seipp (2022). "Best-First Width Search for Lifted Classical Planning." In: *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*. Ed. by Sylvie Thiébaux and William Yeoh. AAAI Press, pp. 11–15 (cit. on p. 89).

Augusto B. Corrêa and Jendrik Seipp (2024). "Consolidating LAMA with Best-First Width Search." In: *ICAPS 2024 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)* (cit. on p. 89).

Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov (2001). "Complexity and Expressive Power of Logic Programming." In: *ACM Computing Surveys* 33.3, pp. 374–425 (cit. on pp. 93, 105).

Giuseppe De Giacomo, Yves Lespérance, and Fabio Patrizi (2016). "Bounded situation calculus action theories." In: *Artificial Intelligence* 237, pp. 172–203 (cit. on pp. 16, 143).

Rik De Graaff, Augusto B. Corrêa, and Florian Pommerening (2021). "Concept Languages as Expert Input for Generalized Planning: Preliminary Results." In: *ICAPS 2021 Workshop on Knowledge Engineering for Planning and Scheduling*.

Daniel Doebber, André Grahl Pereira, and Augusto B. Corrêa (2023). "OpCount4Sat: Operator Counting Heuristics for Satis- ficing Planning." In: *Tenth International Planning Competition (IPC-10): Planner Abstracts*.

Carmel Domshlak, Michael Katz, and Alexander Shleyfman (2015). *Symmetry Breaking in Deterministic Planning as Forward Search: Orbit Space Search Algorithm*. Tech. rep. IS/IE-2015-03. Technion (cit. on p. 135).

James E. Doran and Donald Michie (1966). "Experiments with the Graph Traverser program." In: *Proceedings of the Royal Society A* 294, pp. 235–259 (cit. on p. 15).

Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas (1994). *Mathematical Logic*. 2nd. Springer-Verlag (cit. on p. 9).

Stefan Edelkamp, Alberto Lluch-Lafuente, and Ionut Moraru (2019). *Introducing Dynamic Object Creation to PDDL Planning*. https://openreview.net/forum?id=rkxRj58y5N (cit. on p. 143).

Mojtaba Elahi and Jussi Rintanen (2024). "Optimizing the Optimization of Planning Domains by Automatic Action Schema Splitting." In: *Proceedings of the Thirty-Eigth AAAI Conference on Artificial Intelligence (AAAI 2024)*. Ed. by Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan. AAAI Press, pp. 20096–20103 (cit. on p. 115).

Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian (1991). *Complexity, Decidability and Undecidability Results for Domain-Independent Planning: A Detailed Analysis*. Tech. rep. CS-TR-2797, UMIACS-TR-91-154, SRC-TR-91-96. University of Maryland (cit. on p. 134).

Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian (1995). "Complexity, Decidability and Undecidability Results for Domain-Independent Planning." In: *Artificial Intelligence* 76.1–2, pp. 75–88 (cit. on pp. 15, 134).

Joan Espasa, Jordi Coll, Ian Miguel, and Mateu Villaret (2019). "Towards Lifted Encodings for Numeric Planning in Essence Prime." In: *CP 2019 Workshop on Constraint Modelling and Reformulation* (cit. on p. 44).

Wolfgang Faber, Nicola Leone, and Simona Perri (2012). "The Intelligent Grounder of DLV." In: *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*. Ed. by Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 247–264 (cit. on p. 116).

Ronald Fagin (1983). "Acyclic Database Schemes (of Various Degrees): A Painless Introduction." In: *Colloquium on Trees in Algebra and Programming*, pp. 65–89 (cit. on p. 24).

Johannes K. Fichte, Markus Hecher, and Florim Hamiti (2021). "The Model Counting Competition 2020." In: *ACM Journal of Experimental Algorithmics* 26.13, pp. 1–26 (cit. on p. 110).

Maximilian Fickert (2018). "Making Hill-Climbing Great Again through Online Relaxation Refinement and Novelty Pruning." In: *Proceedings of the 11th Annual Symposium on Combinatorial Search (SoCS 2018)*. Ed. by Vadim Bulitko and Sabine Storandt. AAAI Press, pp. 158–162 (cit. on p. 89).

Maximilian Fickert (2020). "A Novel Lookahead Strategy for Delete Relaxation Heuristics in Greedy Best-First Search." In: *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)*. Ed. by J. Christopher Beck, Erez Karpas, and Shirin Sohrabi. AAAI Press, pp. 119–123 (cit. on p. 89).

Maximilian Fickert and Jörg Hoffmann (2017). "Complete Local Search: Boosting Hill-Climbing through Online Relaxation Refinement." In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*. Ed. by Laura Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith. AAAI Press, pp. 107–115 (cit. on p. 89).

Maximilian Fickert and Jörg Hoffmann (2018). "OLCFF: Online-Learning h^{CFF}." In: *Ninth International Planning Competition (IPC-9): Planner Abstracts*, pp. 17–19 (cit. on p. 89).

Richard E. Fikes and Nils J. Nilsson (1971). "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." In: *Artificial Intelligence* 2, pp. 189–208 (cit. on pp. 12, 17, 21, 39).

Daniel Fišer (2020). "Lifted Fact-Alternating Mutex Groups and Pruned Grounding of Classical Planning Problems." In: *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2020)*. Ed. by Vincent Conitzer and Fei Sha. AAAI Press, pp. 9835–9842 (cit. on pp. 45, 97, 111, 114, 115).

Avierzi S. Fraenkel and David Lichtenstein (1981). "Computing a perfect strategy for $n \times n$ Chess requires time exponential in n." In: *Journal of Combinatorial Theory (Series A)* 31.2, pp. 199–214 (cit. on p. 154).

Guillem Francès (2017). "Effective Planning with Expressive Languages." PhD thesis. Universitat Pompeu Fabra (cit. on p. 45).

Guillem Francès and Héctor Geffner (2016). "$\exists$-STRIPS: Existential Quantification in Planning and Constraint Satisfaction." In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*. Ed. by Subbarao Kambhampati. AAAI Press, pp. 3082–3088 (cit. on p. 45).

Guillem Francès, Hector Geffner, Nir Lipovetzky, and Miquel Ramiréz (2018). "Best-First Width Search in the IPC 2018: Complete, Simulated, and Polynomial Variants." In: *Ninth International Planning Competition (IPC-9): Planner Abstracts*, pp. 23–27 (cit. on pp. 18, 77, 78, 85, 86, 93).

Guillem Francès, Miquel Ramírez, Nir Lipovetzky, and Héctor Geffner (2017). "Purely Declarative Action Representations are Overrated: Classical Planning with Simulators." In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*. Ed. by Carles Sierra. IJCAI, pp. 4294–4301 (cit. on pp. 77–79, 82, 85, 89, 147).

Raquel Fuentetaja and Tomás de la Rosa (2016). "Compiling irrelevant objects to counters. Special case of creation planning." In: *AI Communications* 29.3, pp. 435–467 (cit. on pp. 119, 142, 143).

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub (2019). "Multi-shot ASP solving with clingo." In: *Theory and Practice of Logic Programming* 19.1, pp. 27–82 (cit. on pp. 94, 109, 115).

Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub (2011). "Advances in gringo Series 3." In: *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*. Ed. by James P. Delgrande and Wolfgang Faber. Springer Berlin Heidelberg, pp. 345–351 (cit. on pp. 90, 94, 97, 147).

Michael Gelfond and Vladimir Lifschitz (1988). "The Stable Model Semantics for Logic Programming." In: *Proceedings of the Fifth International Conference and Symposium on Logic Programming*. Ed. by Robert A. Kowalski and Kenneth A. Bowen. MIT Press, pp. 1070–1080 (cit. on p. 105).

Michael Gelfond and Vladimir Lifschitz (1991). "Classical Negation in Logic Programs and Disjunctive Databases." In: *New Generation Computing* 9.3/4, pp. 365–386 (cit. on p. 105).

Daniel Gnad, Álvaro Torralba, Martín Ariel Domínguez, Carlos Areces, and Facundo Bustos (2019). "Learning How to Ground a Plan – Partial Grounding in Classical Planning." In: *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*. AAAI Press, pp. 7602–7609 (cit. on pp. 44, 115).

Georg Gottlob, Nicola Leone, and Francesco Scarcello (2002). "Hypertree decompositions and tractable queries." In: *Journal of Computer and System Sciences* 64.3, pp. 579–627 (cit. on pp. 4, 43, 106, 148).

Marc H. Graham (1979). *On the Universal Relation*. Tech. rep. University of Toronto (cit. on p. 24).

Cordell Green (1969a). "Application of theorem proving to problem solving." In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI 1969)*. Ed. by Donald E. Walker and Lewis M. Norton. William Kaufmann, pp. 219–239 (cit. on p. 17).

Cordell Green (1969b). "Theorem-proving by resolution as a basis for question-answering systems." In: *Machine Intelligence 4*. Ed. by Bernard Meltzer and Donald Michie. Edinburgh University Press, pp. 183–205 (cit. on p. 17).

Martin Grohe and Pascal Schweitzer (2020). "The graph isomorphism problem." In: *Communications of the ACM* 63.11, pp. 128–134 (cit. on p. 135).

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107 (cit. on p. 15).

Juris Hartmanis and Richard Edwin Stearns (1965). "On the Computational Complexity of Algorithms." In: *Transactions of the American Mathematical Society* 117, pp. 285–306 (cit. on p. 154).

Patrik Haslum (2007). "Reducing Accidental Complexity in Planning Problems." In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*. Ed. by Manuela M. Veloso, pp. 1898–1903 (cit. on p. 115).

Patrik Haslum (2011). "Computing Genome Edit Distances using Domain-Independent Planning." In: *ICAPS 2011 Scheduling and Planning Applications woRKshop*, pp. 45–51 (cit. on pp. 21, 93, 115).

Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise (2019). *An Introduction to the Planning Domain Definition Language*. Vol. 13. Synthesis Lectures on Artificial Intelligence and Machine Learning 2. Morgan & Claypool (cit. on pp. 5, 14, 18, 21, 119, 120).

Malte Helmert (2006). "The Fast Downward Planning System." In: *Journal of Artificial Intelligence Research* 26, pp. 191–246 (cit. on pp. 18, 38, 44, 48, 59, 65, 72, 73, 93, 141).

Malte Helmert (2009). "Concise Finite-Domain Representations for PDDL Planning Tasks." In: *Artificial Intelligence* 173, pp. 503–535 (cit. on pp. 4, 5, 14, 21, 44, 47, 50–53, 62, 63, 66, 67, 90, 93–97, 99, 100, 111, 113–115, 148, 149).

Malte Helmert and Carmel Domshlak (2009). "Landmarks, Critical Paths and Abstractions: What's the Difference Anyway?" In: *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*. Ed. by Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis. AAAI Press, pp. 162–169 (cit. on p. 48).

Malte Helmert, Silvan Sievers, Alexander Rovner, and Augusto B. Corrêa (2022). "On the Complexity of Heuristic Synthesis for Satisficing Classical Planning: Potential Heuristics and Beyond." In: *Proceedings of the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022)*. Ed. by Sylvie Thiébaux and William Yeoh. AAAI Press, pp. 124–133.

Jörg Hoffmann (2005). "Where 'Ignoring Delete Lists' Works: Local Search Topology in Planning Benchmarks." In: *Journal of Artificial Intelligence Research* 24, pp. 685–758 (cit. on p. 16).

Jörg Hoffmann, Piergiorgio Bertoli, Malte Helmert, and Marco Pistore (2009). "Message-Based Web Service Composition, Integrity Constraints, and Planning under Uncertainty: A New Connection." In: *Journal of Artificial Intelligence Research* 35, pp. 49–117 (cit. on pp. 125, 142, 143).

Jörg Hoffmann and Bernhard Nebel (2001). "The FF Planning System: Fast Plan Generation Through Heuristic Search." In: *Journal of Artificial Intelligence Research* 14, pp. 253–302 (cit. on pp. 4, 16, 18, 41, 47, 48, 50, 58, 59, 65, 89, 93, 114, 147).

Rostislav Horčík and Daniel Fišer (2021). "Endomorphisms of Lifted Planning Problems." In: *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)*. Ed. by Robert P. Goldman, Susanne Biundo, and Michael Katz. AAAI Press, pp. 174–183 (cit. on pp. 6, 43, 74, 75).

Rostislav Horčík, Daniel Fišer, and Álvaro Torralba (2022). "Homomorphisms of Lifted Planning Tasks: The Case for Delete-free Relaxation Heuristics." In: *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI 2022)*. Ed. by Vasant Honavar and Matthijs Spaan. AAAI Press, pp. 9767–9775 (cit. on pp. 75, 149).

Neil Immerman (1986). "Relational Queries Computable in Polynomial Time." In: *Information and Control* 68.1-3, pp. 86–104 (cit. on pp. 12, 93).

Tomi Janhunen (2006). "Some (in)translatability results for normal logic programs and propositional theories." In: *Journal of Applied Non-Classical Logics* 16.1–2, pp. 35–86 (cit. on p. 110).

Lucas Galery Käser, Clemens Büchner, Augusto B. Corrêa, Florian Pommerening, and Gabriele Röger (2022). "Machetli: Simplifying Input Files for Debugging." In: *ICAPS 2022 System Demonstrations and Exhibits*.

Michael Katz and Jörg Hoffmann (2014). "Mercury Planner: Pushing the Limits of Partial Delete Relaxation." In: *Eighth International Planning Competition (IPC-8): Planner Abstracts*, pp. 43–47 (cit. on p. 93).

Michael Katz, Nir Lipovetzky, Dany Moshkovich, and Alexander Tuisov (2017). "Adapting Novelty to Classical Planning as Heuristic Search." In: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*. Ed. by Laura Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith. AAAI Press, pp. 172–180 (cit. on pp. 80, 89).

Henry Kautz, David McAllester, and Bart Selman (1996). "Encoding Plans in Propositional Logic." In: *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR 1996)*. Ed. by Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro. Morgan Kaufmann, pp. 374–384 (cit. on p. 44).

Henry Kautz and Bart Selman (1992). "Planning as Satisfiability." In: *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 1992)*. Ed. by Bernd Neumann. John Wiley and Sons, pp. 359–363 (cit. on pp. 18, 44).

Emil Keyder and Héctor Geffner (2008). "Heuristics for Planning with Action Costs Revisited." In: *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*. Ed. by Malik Ghallab,

Constantine D. Spyropoulos, Nikos Fakotakis, and Nikos Avouris. IOS Press, pp. 588–592 (cit. on pp. 16, 48, 50).

Jana Köhler and Jörg Hoffmann (2000). "On the Instantiation of ADL Operators Involving Arbitrary First-Order Formulas." In: *Proceedings of the ECAI 2000 Workshop on New Results in Planning, Scheduling and Design (PuK2000)*, pp. 74–82 (cit. on pp. 93, 114).

Alexander Koller and Ronald Petrick (2011). "Experiences with Planning for Natural Language Generation." In: *Computational Intelligence* 27.1, pp. 23–40 (cit. on p. 21).

Robert A. Kowalski (1979). *Logic for Problem Solving*. Vol. 7. The Computer Science Library: Artificial Intelligence Series. North-Holland (cit. on p. 16).

Jean-Marie Lagniez and Pierre Marquis (2014). "Preprocessing for Propositional Model Counting." In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*. Ed. by Carla E. Brodley and Peter Stone. AAAI Press, pp. 2688–2694 (cit. on p. 110).

Jean-Marie Lagniez and Pierre Marquis (2017). "An Improved Decision-DNNF Compiler." In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*. Ed. by Carles Sierra. IJCAI, pp. 667–673 (cit. on p. 110).

Pascal Lauer, Álvaro Torralba, Daniel Fišer, Daniel Höller, Julia Wichlacz, and Jörg Hoffmann (2021). "Polynomial-Time in PDDL Input Size: Making the Delete Relaxation Feasible for Lifted Planning." In: *Proceedings of the 30th International Joint Conference on Artificial Intelligence (IJCAI 2021)*. Ed. by Zhi-Hua Zhou. IJCAI, pp. 4119–4126 (cit. on pp. 6, 39, 65, 70, 71, 74, 86, 134).

Hector J. Levesque (2005). "Planning with Loops." In: *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*. Ed. by Leslie Pack Kaelbling and Alessandro Saffiotti. Professional Book Center, pp. 509–515 (cit. on p. 16).

Vladimir Lifschitz (1987). "On the Semantics of STRIPS." In: *Reasoning about Actions and Plans*. Ed. by M. Georgeff and A. Lansky. Morgan Kaufmann, pp. 1–9 (cit. on p. 17).

Nir Lipovetzky and Héctor Geffner (2012). "Width and Serialization of Classical Planning Problems." In: *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*. Ed. by Luc De Raedt, Christian Bessiere, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter Lucas. IOS Press, pp. 540–545 (cit. on pp. 4, 7, 77, 78, 89).

Nir Lipovetzky and Héctor Geffner (2014). "Width-based Algorithms for Classical Planning: New Results." In: *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*. Ed. by Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan. IOS Press, pp. 1059–1060 (cit. on p. 89).

Nir Lipovetzky and Hector Geffner (2017). "Best-First Width Search: Exploration and Exploitation in Classical Planning." In: *Proceedings*

of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI 2017). Ed. by Satinder Singh and Shaul Markovitch. AAAI Press, pp. 3590–3596 (cit. on pp. 7, 74, 77–79, 86, 89, 136, 147).

Derek Long and Maria Fox (2003). "The 3rd International Planning Competition: Results and Analysis." In: *Journal of Artificial Intelligence Research* 20, pp. 1–59 (cit. on pp. 119, 124, 139, 142).

Davide Mario Longo (2023). "On the Potential of Structural Decomposition of Database and AI Problems." PhD thesis. Technische Universität Wien (cit. on pp. 115, 148).

David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv (1979). "Testing Implications of Data Dependencies." In: *ACM Transactions on Database Systems* 4.4, pp. 455–469 (cit. on p. 130).

Rami Matloob and Mikhail Soutchanski (2016). "Exploring Organic Synthesis with State-of-the-Art Planning Techniques." In: *ICAPS 2016 Scheduling and Planning Applications woRKshop*, pp. 52–61 (cit. on pp. 21, 93).

David A. McAllester and David Rosenblitt (1991). "Systematic Nonlinear Planning." In: *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI 1991)*. Ed. by Thomas L. Dean and Kathleen R. McKeown. AAAI Press/MIT Press, pp. 634–639 (cit. on p. 17).

John McCarthy (1958). "Programs with Common Sense." In: *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*. Her Majesty's Stationary Office, London, pp. 75–91 (cit. on p. 16).

John McCarthy (1963). *Situations, actions, and causal laws*. Memo 2, Stanford University Artificial Intelligence Project, Stanford, California (cit. on pp. 16, 143).

John McCarthy and Patrick J. Hayes (1969). "Some Philosophical Problems from the Standpoint of Artificial Intelligence." In: *Machine Intelligence 4*. Ed. by Bernard Meltzer and Donald Michie. Edinburgh University Press, pp. 463–502 (cit. on p. 16).

Drew McDermott (1996). "A Heuristic Estimator for Means-Ends Analysis in Planning." In: *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS 1996)*. Ed. by Brian Drabble. AAAI Press, pp. 142–149 (cit. on pp. 17, 44, 75).

Drew McDermott (1999). "Using Regression-Match Graphs to Control Search in Planning." In: *Artificial Intelligence* 109.1–2, pp. 111–159 (cit. on p. 74).

Drew McDermott (2000). "The 1998 AI Planning Systems Competition." In: *AI Magazine* 21.2, pp. 35–55 (cit. on pp. 18, 119).

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins (1998). *PDDL – The Planning Domain Definition Language – Version 1.2*. Tech. rep. CVC TR-98-003/DCS TR-1165. Yale University: Yale Center for Computational Vision and Control (cit. on pp. 5, 14, 18, 21, 120).

Michael Morak and Stefan Woltran (2012). *Preprocessing of Complex Non-Ground Rules in Answer Set Programming*. Tech. rep. DBAI-TR-2011-72 (Revised Version). Technische Universität Wien (cit. on pp. 63, 94, 99).

Hootan Nakhost and Martin Müller (2009). "Monte-Carlo Exploration for Deterministic Planning." In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*. Ed. by Craig Boutilier. AAAI Press, pp. 1766–1771 (cit. on p. 89).

Allen Newell and Herbert A. Simon (1963). "GPS: A Program that Simulates Human Thought." In: *Computers and Thought*. Ed. by E. A. Feigenbaum and J. Feldman. Oldenbourg, pp. 279–293 (cit. on p. 16).

Christos H. Papadimitriou and Mihalis Yannakakis (1999). "On the Complexity of Database Queries." In: *Journal of Computer and System Sciences* 58.3, pp. 407–427 (cit. on p. 35).

Judea Pearl (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley (cit. on p. 15).

Edwin P. D. Pednault (1989). "ADL: Exploring the Middle Ground between STRIPS and the Situation Calculus." In: *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR 1989)*. Ed. by Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter. Morgan Kaufmann, pp. 324–332 (cit. on pp. 17, 21).

J. Scott Penberthy and Daniel S. Weld (1992). "UCPOP: A Sound, Complete, Partial Order Planner for ADL." In: *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR 1992)*. Ed. by Bernhard Nebel, Charles Rich, and William Swartout. Morgan Kaufmann, pp. 103–114 (cit. on pp. 17, 44).

Alice Petrov and Christian Muise (2023). "Automated Planning Techniques for Elementary Proofs in Abstract Algebra." In: *ICAPS 2023 Scheduling and Planning Applications woRKshop* (cit. on pp. 119, 138, 142).

Julie Porteous, Laura Sebastia, and Jörg Hoffmann (2001). "On the Extraction, Ordering, and Usage of Landmarks in Planning." In: *Proceedings of the Sixth European Conference on Planning (ECP 2001)*. Ed. by Amedeo Cesta and Daniel Borrajo. AAAI Press, pp. 174–182 (cit. on p. 129).

Raymond Reiter (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press (cit. on pp. 16, 125, 143).

Silvia Richter and Malte Helmert (2009). "Preferred Operators and Deferred Evaluation in Satisficing Planning." In: *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*. Ed. by Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis. AAAI Press, pp. 273–280 (cit. on pp. 59, 65, 68, 73).

Silvia Richter and Matthias Westphal (2008). *The LAMA Planner — Using Landmark Counting in Heuristic Search*. IPC 2008 short papers, http://ipc.informatik.uni-freiburg.de/Planners (cit. on p. 89).

Silvia Richter and Matthias Westphal (2010). "The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks." In: *Journal of Artificial Intelligence Research* 39, pp. 127–177 (cit. on pp. 41, 48, 59, 86, 112).

Silvia Richter, Matthias Westphal, and Malte Helmert (2011). "LAMA 2008 and 2011 (planner abstract)." In: *IPC 2011 Planner Abstracts*, pp. 50–54 (cit. on p. 50).

Bram Ridder (2013). "Lifted Heuristics: Towards More Scalable Planning Systems." PhD thesis. King's College London (cit. on pp. 22, 41, 44).

Bram Ridder and Maria Fox (2014). "Heuristic Evaluation Based on Lifted Relaxed Planning Graphs." In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*. Ed. by Steve Chien, Alan Fern, Wheeler Ruml, and Minh Do. AAAI Press, pp. 244–252 (cit. on pp. 41, 74, 75).

Pat Riddle, Jordan Douglas, Mike Barley, and Santiago Franco (2016). "Improving Performance by Reformulating PDDL into a Bagged Representation." In: *ICAPS 2016 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*, pp. 28–36 (cit. on p. 143).

Jussi Rintanen (2017). "Schematic Invariants by Reduction to Ground Invariants." In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI 2017)*. Ed. by Satinder Singh and Shaul Markovitch. AAAI Press, pp. 3644–3650 (cit. on p. 45).

Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar (2008). "A Compact and Efficient SAT Encoding for Planning." In: *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*. Ed. by Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen. AAAI Press, pp. 296–303 (cit. on p. 44).

Gabriele Röger and Malte Helmert (2010). "The More, the Merrier: Combining Heuristic Estimators for Satisficing Planning." In: *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*. Ed. by Ronen Brafman, Héctor Geffner, Jörg Hoffmann, and Henry Kautz. AAAI Press, pp. 246–249 (cit. on pp. 7, 78, 80).

Gabriele Röger, Silvan Sievers, and Michael Katz (2018). "Symmetry-based Task Reduction for Relaxed Reachability Analysis." In: *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*. Ed. by Mathijs de Weerdt, Sven Koenig, Gabriele Röger, and Matthijs Spaan. AAAI Press, pp. 208–217 (cit. on p. 45).

Stuart Russell and Peter Norvig (2020). *Artificial Intelligence: A Modern Approach*. Pearson (cit. on p. 16).

Zeynep G. Saribatur, Thomas Eiter, and Peter Schüller (2021). "Abstraction for non-ground answer set programs." In: *Artificial Intelligence* 300, p. 103563 (cit. on p. 75).

Walter J. Savitch (1970). "Relationships Between Nondeterministic and Deterministic Tape Complexities." In: *Journal of Computer and System Sciences* 4.2, pp. 177–192 (cit. on p. 154).

Francesco Scarcello, Gianluigi Greco, and Nicola Leone (2007). "Weighted hypertree decompositions and optimal query plans." In: *Journal of Computer and System Sciences* 73.3, pp. 475–506 (cit. on p. 43).

Jendrik Seipp (2018). "Fast Downward Scorpion." In: *Ninth International Planning Competition (IPC-9): Planner Abstracts*, pp. 77–79 (cit. on p. 7).

Jendrik Seipp (2023). "Scorpion 2023." In: *Tenth International Planning Competition (IPC-10): Planner Abstracts* (cit. on p. 18).

Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert (2017). *Downward Lab*. https://doi.org/10.5281/zenodo.790461 (cit. on p. 5).

Alexander Shleyfman, Alexander Tuisov, and Carmel Domshlak (2016). "Blind Search for Atari-Like Online Planning Revisited." In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*. Ed. by Subbarao Kambhampati. AAAI Press, pp. 3251–3257 (cit. on p. 89).

Silvan Sievers, Gabriele Röger, Martin Wehrle, and Michael Katz (2019). "Theoretical Foundations for Structural Symmetries of Lifted PDDL Tasks." In: *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*. Ed. by Nir Lipovetzky, Eva Onaindia, and David E. Smith. AAAI Press, pp. 446–454 (cit. on p. 45).

Tom Silver, Rohan Chitnis, Aidan Curtis, Joshua B. Tenenbaum, Tomás Lozano-Pérez, and Leslie Pack Kaelbling (2021). "Planning with Learned Object Importance in Large Problem Instances using Graph Neural Networks." In: *Proceedings of the Thirty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2021)*. Ed. by Kevin Leyton-Brown and Mausam. AAAI Press, pp. 11962–11971 (cit. on p. 119).

Simon Ståhlberg (2023). "Lifted Successor Generation by Maximum Clique Enumeration." In: *Proceedings of the 26th European Conference on Artificial Intelligence (ECAI 2023)*. Ed. by Kobi Gal, Ann Nowé, Grzegorz J. Nalepa, Roy Fairstein, and Roxana Rădulescu. IOS Press, pp. 2194–2201 (cit. on pp. 6, 43).

Richard Edwin Stearns, Juris Hartmanis, and Philip M. Lewis II (1965). "Hierarchies of memory limited computations." In: *Proceedings of the Sixth Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1965)*. IEEE Computer Society, pp. 179–190 (cit. on p. 155).

Álvaro Torralba, Vidal Alcázar, Daniel Borrajo, Peter Kissmann, and Stefan Edelkamp (2014). "SymBA*: A Symbolic Bidirectional A*

Planner." In: *Eighth International Planning Competition (IPC-8): Planner Abstracts*, pp. 105–109 (cit. on pp. 18, 93).

Jeffrey D. Ullman (1988). *Principles of Database and Knowledge-Base Systems. Volume I: Classical Database Systems*. Computer Science Press (cit. on pp. 63, 111).

Jeffrey D. Ullman (1989). *Principles of Database and Knowledge-Base Systems. Volume II: The New Technologies*. Computer Science Press (cit. on pp. 22, 24, 27, 29, 63, 111).

Moshe Y. Vardi (1982). "The Complexity of Relational Query Languages (Extended Abstract)." In: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (STOC '82)*. Ed. by Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber. ACM Press, pp. 137–146 (cit. on pp. 12, 93).

Vincent Vidal (2011). "YAHSP2: Keep It Simple, Stupid." In: *IPC 2011 Planner Abstracts*, pp. 83–90 (cit. on p. 89).

Daniel S. Weld (1994). "An Introduction to Least Commitment Planning." In: *AI Magazine* 15.4, pp. 27–61 (cit. on p. 17).

Julia Wichlacz, Daniel Höller, and Jörg Hoffmann (2021). "Landmark Heuristics for Lifted Planning – Extended Abstract." In: *Proceedings of the 14th Annual Symposium on Combinatorial Search (SoCS 2021)*. Ed. by Hang Ma and Ivan Serina. AAAI Press, pp. 242–244 (cit. on p. 89).

Avi Wigderson (2019). *Mathematics and Computation: A Theory Revolutionizing Technology and Science*. Princeton University Press (cit. on p. 153).

P. G. Wodehouse (1930). *Very Good, Jeeves*. Herbert Jenkins (cit. on p. 1).

Fan Xie, Martin Müller, Robert C. Holte, and Tatsuya Imai (2014). "Type-based Exploration with Multiple Search Queues for Satisficing Planning." In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*. Ed. by Carla E. Brodley and Peter Stone. AAAI Press, pp. 2395–2401 (cit. on p. 90).

Mihalis Yannakakis (1981). "Algorithms for Acyclic Database Schemes." In: *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB 1981)*. IEEE Press, pp. 82–94 (cit. on pp. 4, 22, 24, 26, 28, 29, 147).

Håkan L. S. Younes and Reid G. Simmons (2002). "On the Role of Ground Actions in Refinement Planning." In: *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002)*. Ed. by Malik Ghallab, Joachim Hertzberg, and Paolo Traverso. AAAI Press, pp. 54–62 (cit. on pp. 17, 44).

Håkan L. S. Younes and Reid G. Simmons (2003). "VHPOP: Versatile Heuristic Partial Order Planner." In: *Journal of Artificial Intelligence Research* 20, pp. 405–430 (cit. on pp. 17, 44).

C. T. Yu and M. Z. Ozsoyoglu (1979). "An algorithm for tree-query membership of a distributed query." In: *Third International Computer*

Software and Applications Conference (COMPSAC 1979), pp. 306–312 (cit. on p. 24).

Lin Zhu and Robert Givan (2003). "Landmark Extraction via Planning Graph Propagation." In: *ICAPS 2003 Doctoral Consortium*, pp. 156–160 (cit. on p. 73).

COLOPHON

This document was typeset using the `classicthesis` package (v4.6) developed by André Miede and Ivo Pletikosić (`https://bitbucket.org/amiede/classicthesis/`). The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"*.