Planning using Lifted Task Representations

Augusto B. Corrêa

augusto.blaascorrea@unibas.ch
17-066-507

November 26th, 2019

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Florian Pommerening

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence

Abstract

Most automated planners use heuristic search to solve the tasks. Usually, the planners get as input a *lifted representation* of the task in PDDL, a compact formalism describing the task using a fragment of first-order logic. The planners then transform this task description into a *grounded representation* where the task is described in propositional logic. This new grounded format can be exponentially larger than the lifted one, but many planners use this grounded representation because it is easier to implement and reason about.

However, sometimes this transformation between lifted and grounded representations is not tractable. When this is the case, there is not much that planners based on heuristic search can do. Since this transformation is a required preprocess, when this fails, the whole planner fails.

To solve the grounding problem, we introduce new methods to deal with tasks that cannot be grounded. Our work aims to find good ways to perform heuristic search while using a lifted representation of planning problems. We use the point-of-view of *planning as a database progression problem* and borrow solutions from the areas of relational algebra and database theory.

Our theoretical and empirical results are motivating: several instances that were never solved by any planner in the literature are now solved by our new lifted planner. For example, our planner can solve the challenging Organic Synthesis domain using a breadth-first search, while state-of-the-art planners cannot solve more than 60% of the instances. Furthermore, our results offer a new perspective and a deep theoretical study of lifted representations for planning tasks.

Acknowledgements

First, I would like to thank Florian Pommerening, Guillem Francès, and Malte Helmert. Florian, for being the most helpful advisor I could ever have, not only through this thesis but also throughout my whole degree. Guillem, for the contributions and collaboration to this work and other parts of my life. And Malte, for giving me the opportunity to come to Basel and supporting me during my entire degree. I will always be grateful for all the help and support that I have received.

I also want to thank Salomé Eriksson, Patrick Ferber, Cedric Geissmann, Thomas Keller, Gabriele Röger, Jendrik Seipp, and Silvan Sievers, who made me feel part of the AI group during these last two years.

Throughout this thesis, I also adopted several suggestions and corrections provided by friends of mine. I owe a huge debt to my great friends and proofreaders Alex Gliesch, Alexander Rovner, André G. Pereira, Giorgi Grigalashvili, John Gamboa, Linnea Ingmar, and Thiago Bell. I shall also thank many other friends with whom I had the chance to share my life. Special to Amanda, Bruno, Camila, João, Jonas, Kazia, Maurício, Nicole, Normand, Philipp, Thomas, Vinícius, Tadeu, and Zora.

Last, but not least, I would like to thank my father César, my almost-mother Kelly, my sister Clara, and my grandmother Ema for all the sacrifice, support, love, and encouragement.

To all of you: thank you very much.

Contents

Chapter 1

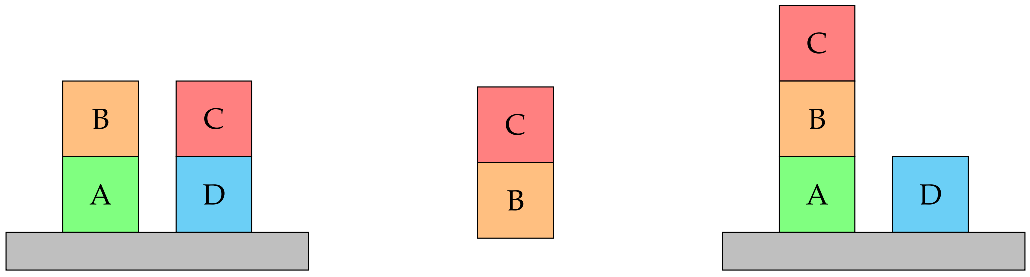
Introduction

Planning is an important area of model-based Artificial Intelligence. Broadly speaking, in a planning problem we want to find an *action sequence* leading from an *initial state* to a certain *goal*. A more specific area of planning is *classical planning*. In this setting, there is a finite set of deterministic actions that can be used and the information about the state of the problem is completely observable. Several popular families of problems, called *domains* can be formulated as classical planning tasks: e.g., Rubik's cube, transportation problems, simulation of chemical reactions, etc. Ideally, one wants to find a single planning algorithm, a *planner*, that can efficiently solve any possible domain of interest. These planners are called *domain-independent* planners.

A common way to represent planning tasks is using a *factored representation*. A *fact* is a statement about the problem that can be true or false in a given situation. We can describe each *state* of the task as a set of facts indicating the information that is true in this state. For example, consider the Blocksworld domain [Bacchus, 2001], illustrated in Figure 1.1a. This problem involves a set of blocks, which are stacked on top of each other, forming many stacks placed directly on a table. Starting from an arbitrary initial configuration of blocks and stacks, the goal of the task is to find a sequence of movements of the blocks leading to a given goal configuration. The only type of movement allowed, in our definition of the problem, is to pick up a block without other blocks on it and stack it on top of another block. We can represent the state shown in Figure 1.1a with the following facts:

A is a block.
B is a block.
C is a block.
D is a block.
A is on the table.
D is on the table.
B is on top of A.
C is on top of D.
The top of B is clear.
The top of C is clear.

The representation of a state contains only the facts that are true in this state, called the *positive facts*. These facts are obtained from a "universal set" of facts that could be true in some state of the task. Facts contained in this "universal set" but not contained in a state – e.g., A is on top of B – are assumed to be false. Similarly, the goal of the planning task is also a set of facts. We say that we solve a planning task once we find a



(a) A state of task. (b) Single goal fact of the task. (c) Goal state.

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

sequence of actions leading from the initial state to a state where all goal facts are true. In our example, let us assume that the state of Figure 1.1a is also the initial state, and that goal is the following single fact:

C is on top of B.

Figure 1.1b illustrates this goal condition. To reach the goal, we need to find a sequence of actions leading from the initial configuration to a configuration where the block C is stacked on top of B. Once we find such a state, we say that we found a *goal state*. An example of a goal state satisfying the single goal condition is depicted in Figure 1.1c.

To reach the goal state from the initial state, we can move block C from the top of D and place it on top of B. It is possible to represent this movement in a single action. A possible way to represent the actions of the task is to use a construction similar to an if-then statement. In our running example, the action specified above would be defined as follows

If the following facts are true in the current state....

- The top of B is clear.
- The top of C is clear.
- C is on top of D.

Then, the following facts are true in the successor state...

- C is on top of B.
- The top of B is *not* clear.
- C is *not* on top of D.

If the *preconditions* of the action are satisfied in a given state, we can decide to apply it to this state and produce its respective *effects*. In contrast to states, actions can have negated facts in their preconditions and effects. Intuitively, an action can only be applied if all its preconditions are satisfied. If the state from Figure 1.1a had the position of blocks C and D exchanged, then the preconditions *"The top of C is clear"* and *"C is on top of D"* would be false in this state and the action would not be applicable. If the action is applied, the positive facts in the action effects are added to the current state and the negative ones are removed. The new state (after additions and removals of facts) is a *successor state*.

We now have a sequence of actions leading from our initial state, Figure 1.1a, to a goal state, Figure 1.1c. (In this naive example, this sequence only consists of a single action.) We say that this sequence is a *plan* of our task.

Planning tasks are rarely represented using natural language, as we did so far. Instead, they are usually represented in logical formalisms, such as the *Planning Domain Definition Language* (PDDL) [McDermott, 2000]. In a PDDL planning task, the world is modeled in terms of *objects*. The relations between these objects or their own characteristics are described in the form of Boolean *predicates*. Such relations are changed by *actions*, which modify these predicates directly. The initial state and the goal of a task are also described in terms of such objects and predicates. The PDDL formalism is of special interest because it is used in the *International Planning Competition* (IPC)[1].

For example, in our Blocksworld domain, the objects are the blocks A, B, C, and D. The predicates are *on(?X,?Y)*, which represents that block $?X$ is directly placed on top of block[2] $?Y$; *ontable(?X)*, which indicates that block $?X$ is placed directly on the table; and the predicate *clear(?X)* representing that the top of block $?X$ is clear. Since there is always a single table in any Blocksworld instance, we can represent this domain without having an object for the table itself. Rewriting our initial state from Figure 1.1a using these predicates, we achieve:

ontable(A)
ontable(D)
on(B, A)
on(C, D)
clear(B)
clear(C).

The goal condition of the task is represented analogously as *on(C, B)*.

In planning, actions are also named. We can call the single action of our plan as *move(C, D, B)*. In natural language, one could interpret this name as "move C from the top of D to the top of B". We can then describe the action *move(C,D,B)* as follows, separating it into preconditions and effects instead of if-then statements:

Preconditions:

- *clear(B)*
- *clear(C)*
- *on(C, D)*

Effects:

- *on(C, B)*
- ¬*clear(B)*
- ¬*on(C, D)*

[1]More information about the IPC and its previous editions can be found at `http://www.icaps-conference.org/index.php/Main/Competitions`

[2]$?X$ and $?Y$ are placeholders for objects of the task.

where $\neg F$ is the negation of fact F.

However, this is not how actions are generally defined in PDDL. To see why, let us first assume we need to define a move action for every triple of blocks in our task, like the action defined above. For a task with n blocks, we have $O(n^3)$ move actions. Although this is still polynomial, this cubic growth makes the number of actions in instances with more blocks become very large.

To overcome the large number of ground actions in a domain, we can use *lifted actions*. In this more compact representation, actions are described in first-order logic *action schemas*, which do not necessarily specify objects of the task, but use *free variables* that must be instantiated in order to obtain a ground action. Our action *move(C,D,B)* is an instantiation of *move(?X,?Y,?Z)*, where *?X*, *?Y*, and *?Z* are free variables that are substituted with C, D, and B, respectively. This substitution of free variables for objects is called *grounding*. An action specifying the exact objects it affects is called a *ground action*.

The following representation of the action *move(?X, ?Y, ?Z)* is an illustrative example of an action schema:

Preconditions:

- *clear(?Z)*
- *clear(?X)*
- *on(?X, ?Y)*
- *?X* $\neq$ *?Y* $\neq$ *?Z*.

Effects:

- *on(?X, ?Z)*
- $\neg$*clear(?Z)*
- $\neg$*on(?X, ?Y)*.

(We add an extra precondition, *?X* $\neq$ *?Y* $\neq$ *?Z*, indicating that the objects substituting these three free variables must all be different.)

To be more compact, PDDL domains are defined by this first-order logic formalism. However, most state-of-the-art planners use a fully grounded representation of the tasks. To transform the lifted representation into a grounded representation, most planners need to perform a preprocessing grounding step. When grounding a PDDL task, the planner enumerates a (sufficiently large) set of possible ground actions based on the objects represented in the task instead of using the PDDL first-order representation. The reason for so many planners to use grounded representations might be historical. Bonet and Geffner [2001] introduced a planner based on *heuristic search* using grounded representation. At the time, this planner was far better than any other planner. The main idea of their planner is to solve a planning task by performing a search over all states of the task, trying to find a plan. Because the number of states might be too large, the search explores only the most promising states according to a heuristic function used to estimate the distance from a state to the goal. This new approach motivated further research in the same direction, which resulted in even better planners using this representation [Hoffmann and Nebel, 2001; Helmert, 2006]. This positive-feedback loop continues to motivate more researchers to consider this representation. Additionally, we can also argue that, when considering heuristic search, it is easier to create better heuristic functions when considering a completely grounded representation of the problem.

Planning using grounded actions demonstrated to be a good choice until the IPC 2018. In this edition of the IPC, new domains that are particularly hard to ground were introduced. In these domains, it is not necessarily difficult to find a plan, but it is very hard to produce a grounded version of the planning task. These domains showed that most planners using grounded actions cannot deal with the combinatorial explosion caused by the instantiation of action schemas.

As mentioned before, our Blocksworld example would need $O(n^3)$ ground actions for its move action schema. In fact, such an extension would still be relatively easy to ground. However, in some other domains, it is possible that the grounded task is exponentially larger than its PDDL representation [Erol *et al.*, 1995].

Still, there are many alternatives and improvements to the grounding procedure in the literature. The most used grounding method is probably the work by Helmert [2009]. However, even such efficient techniques are not powerful enough to handle all domains. An example of this is the Organic Synthesis domain [Masoumi *et al.*, 2015; Matloob and Soutchanski, 2016] used in the IPC 2018. In this domain, atoms are bonded forming different molecules, and the objective is to find a plan from an initial set of bonds to a goal molecule. Due to the symmetries and many parameters in every action schema, even instances with very short plans generate millions of grounded actions using the algorithm proposed by Helmert (and quadrillions of grounded actions when using a naive grounding method). Various techniques have been studied in the planning community in order to avoid the potentially huge overhead caused by grounding, e.g., splitting of predicates and action schemas [Robinson *et al.*, 2009; Areces *et al.*, 2014], partially grounded state spaces [Gnad *et al.*, 2019], elimination of symmetries prior to grounding [Röger *et al.*, 2018]. Despite the performance improvements, these techniques do not eliminate the scalability concern in grounding. As reported by Haslum [2007], many planning domains are not particularly hard, but their difficulty arises from an "accidental complexity", coming from the selected representation of the problem. Under this perspective, it is expected that many simple domains will be excessively complex and hard to ground just as a result of their representations.

In this thesis, we focus on a more drastic way to try to avoid the potential harm caused by grounding: *lifted planning*. In lifted planning, we skip the preprocessing grounding step and plan using the PDDL action schemas directly. We focus here on the case of lifted planning using heuristic search. In practice, instead of preprocessing all possible ground actions at first, we perform a grounding procedure in each state, only generating ground actions that are applicable (i.e., with satisfied preconditions) in the current state. There is an underlying trade-off in the decision between grounded and lifted planning. While the grounding preprocessing might be expensive, its costs might be amortized over all states. In contrast, using a lifted representation and generating ground actions for a specific state might be very cheap, but doing that for a large number of states might be worse than preprocessing all actions.

We want to answer the question of *how to efficiently perform lifted planning?* The answer we provide here relies on techniques from database theory. First, we must notice that it is possible to reformulate our whole planning problem from the point-of-view of databases [Lin and Reiter, 1997]. Going back to our Blocksworld example, we can represent a state of our task using a database perspective as follows. Instead of using a set of *ground atoms* (i.e., facts), we represent a state as a collection of *tables*, each corresponding to a specific predicate. The table for a specific predicate has an entry for each *tuple* of objects that instantiate it on the corresponding state. For example, if *on(B,A)* is a true fact in a state, then the tuple *(B, A)* instantiates the predicate *on* in this state and hence this tuple is an entry in the table corresponding to *on*. Following this, the state in Figure 1.1a would then be represented using the following tables:

ontable
A
D

on	
B	*A*
C	*D*

clear
B
C

Since a state is a collection of tables, we can see a successor state as an updated collection of these tables. When applying an action to a state, we simply modify the entries of each table accordingly to the effects of the action.

Now, we need to come up with a good way to find ground actions that are applicable to a state. Consider the action schema *move(?X, ?Y, ?Z)*. We can interpret the grounding process as "find objects to substitute *?X, ?Y*, and *?Z* such that *?X* and *?Z* are in table *clear* and the tuple *(?X,?Y)* is in the table *on*." This is equivalent to performing a *join* program[3] over these tables. A join of two tables produces a new table with all combinations of entries that have equal values for the variables with the same name in both tables[4]. For example, the join of *clear(?X)* with *on(?X, ?Y)* names the single free variable of *clear* with the same name as the first free variable (i.e., first column in the table) of *on*. This join would consist of all tuples in the *on* table such that the first element is also an entry for the *clear* table. In the example above, this would be the tuples *(B, A)* and *(C,D)*. To conclude the grounding of the action, we need to join our intermediate table containing $\{$*(B, A), (C, D)*$\}$ to *clear(?Z)*. Since the tables do not share a variable with the same name, the join is equivalent to the Cartesian product of the tables. This produces a final table with tuples *(B, A, B)*, *(B, A, C)*, *(C, D, C)*, and *(C, D, B)*. The first and the third ones violate the inequality $?X \neq ?Z$ and are removed in a post-processing stage; the second and the fourth tuples are applicable to our state. Grounding *move(?X, ?Y, ?Z)* with these two applicable tuples, we obtain *move(B, A, C)* and *move(C, D, B)*. If we apply *move(B, A, C)* we will end up in a state where block *B* is stacked on top of block *C*. If we apply action *move(C, D, B)*, we obtain the state showed in Figure 1.1c.

In database terms, we could write the join program corresponding to the precondition of this action as:

$$\textit{clear(?Z)} \bowtie \textit{clear(?X)} \bowtie \textit{on(?X, ?Y)}$$

This sequence of joins produces a new table with three columns, one for each object being grounded. The entries in the resulting table are exactly the tuples instantiating *move(?X,?Y,?Z)* into an applicable ground action, including the ground action belonging to our plan, *move(C,D,B)*.

This change in perspective shows also further good news: now we can see that it is possible to generate all the ground applicable actions for a state efficiently, as long as the join of the action preconditions is easy to compute. Luckily, the database community has extensive results on this topic [Ullman, 1989; Gottlob *et al.*, 2001].

In this thesis, we apply some of the most successful methods from the database theory community to the setting of lifted planning. We first formalize the relation between database theory and planning (Chapter 2) and discuss previous approaches from the literature to deal with the problem of grounding (Chapter 3). We show that, for most

[3]Assuming that inequality constraints are post-processed. See Chapter 4.

[4]If both tables do not have a free variable with the same name, then the output of the join is a Cartesian product of the tables.

of the planning domains, lifted planning using database techniques to ground actions at each state can be done efficiently (Chapter 4). Our work focuses mainly on how to generate successors in tasks using lifted representation. Thus, our problem can be stated as how to efficiently find the applicable ground actions in any arbitrary state of the task given only the state and the action schemas. Our theoretical results show that, in many domains, we might expect only a logarithmic overhead in the number of applicable actions (per state) in the worst case. In the cases where such joins cannot be efficiently computed, we introduce simple strategies that still produce the ground actions quickly. Our strategies are implemented in a new lifted planner, which generates successor states based on the join program approach (Chapter 5). In the Organic Synthesis domain, which is considered to be the hardest domain to ground in all IPCs seen so far, our planner is able to solve all its instances using a simple goal-counting heuristic [Fikes and Nilsson, 1971]. In contrast, no planner using grounded representation could solve more than half of its instances in the IPC 2018. In domains where grounding is not necessarily a problem, our planner performs worse than grounded planners, but the difference in performance is not as large as initially expected.

Chapter 2

Background

In this chapter, we formalize the definitions of planning and its relation to database theory, introduced in Chapter 1. First, we begin by defining a *planning task*. Our definition is formulated in order to be similar to *STRIPS* [Fikes and Nilsson, 1971] using PDDL, but the results throughout the thesis can be generalized to other formalisms and definition languages as well. Then, we introduce the details for *grounded* and for *lifted planning*, together with some discussion about the usage of each one. This discussion is followed by the introduction of *heuristic search*, which is probably the most common way to solve planning problems and is also the focus of this thesis. Last, we show how to interpret planning problems as *database progressions*, as demonstrated in the previous chapter.

2.1 Classical Planning

In *classical planning*, a world is modeled as discrete, deterministic, fully-observable, and sequential. The objective of a *planner* is to find a sequence of *actions* transforming the *initial state* of the modeled world into a state satisfying a specific *goal condition*. This sequence of actions from the initial state to a goal state is what we call a *plan*.

A planning task is formally represented as a 5-tuple $\Pi = (\mathcal{P}, \mathcal{A}, O, s_0, \gamma)$. It has a finite set of *objects* O, which represents the constant elements of the task. The interactions among these objects are described using *predicates symbols* $\mathcal{P}$. If $P \in \mathcal{P}$ is an n-ary predicate and $t_1, \ldots, t_n$ are free variables, then $P(t_1, \ldots, t_n)$ is an *atom*. When we instantiate (i.e., substitute) the variables of an atom $P(t_1, \ldots, t_n)$ with some objects in O, we obtain a *ground atom*. A ground atom is equivalent to a (possible) fact about our task, as posed in the previous chapter. Ground atoms indicate the properties of specific objects, while the predicates are simply high-level specifications of these properties without referring to any specific object. The former is in propositional logic, while the latter is in first-order logic.

Example 2.1. In the example from Chapter 1, we have $O = \{A, B, C, D\}$ and the set P contains the predicate symbols *clear, on* and *ontable*. An example of an atom in this case is *on(?X,?Y)*, where *?X* and *?Y* are free variables. Likewise, an example of a ground atom is *on(A,B)*, since A and B are objects from the set of objects O.[1] $\triangle$

A set of ground atoms is called a *state*. Put simply, a state is the set of properties that are currently true in our model. The planning task has a given *initial state*, denoted as s_0. This initial state is interpreted as what the world looks like at the beginning of the task.

[1]We usually write free variables using a question mark (*?*) at the beginning, following the PDDL syntax.

We also have a *goal* condition γ, which is also a set of ground atoms. Intuitively, γ is the set of ground atoms that we want to make true. Throughout the thesis, we say that a set of ground atoms A is *satisfied* in a state s iff $A \subseteq s$. In this way, any state satisfying this goal condition is called as a *goal state*.

The way we transform the initial state into a goal state is by applying *actions*. The actions are represented by a finite set of *action schemas* $\mathcal{A}$. An action schema $a[\Delta] \in \mathcal{A}$ has a set of free variables Δ, a precondition, an add list, and a delete list. Due to their central role in this work, we define action schemas explicitly.

Definition 2.1 (Action Schema). *Given a planning task* $\Pi = (\mathcal{P}, \mathcal{A}, O, s_0, \gamma)$, *an* action schema $a[\Delta] \in \mathcal{A}$ *is a tuple* $(\mathit{pre}(a[\Delta]), \mathit{add}(a[\Delta]), \mathit{del}(a[\Delta]))$ *where* $\mathit{pre}(a[\Delta]), \mathit{add}(a[\Delta])$ *and* $\mathit{del}(a[\Delta])$ *are the* precondition, *the* add list *and the* delete list *of* $a[\Delta]$, *respectively.*

The precondition, the add and delete lists are finite sets of atoms defined over $\mathcal{P}$, *such that* Δ *is the finite set of* free variables *such that* $\Delta = \mathit{free}(\mathit{pre}(a[\Delta])) \cup \mathit{free}(\mathit{add}(a[\Delta])) \cup \mathit{free}(\mathit{del}(a[\Delta]))$, *where* $\mathit{free}(\chi)$ *is the set of free variables in a set* χ.

We can instantiate an action schema $a[\Delta]$ by substituting the free variables Δ in its precondition, add and delete lists by objects in O. By performing this instantiation, we say that we are *grounding* the action schema. The result of this grounding procedure is a *grounded action* a without free variables (sometimes referred only as *action* when the context is clear). Note that, by grounding an action schema $a[\Delta]$, its precondition, add and delete lists become finite sets of ground atoms. The precondition of a grounded action a is written $\mathit{pre}(a)$, and similarly for the add and delete lists.

Example 2.2. In the Blocksworld example of Chapter 1, we grounded the action schema *move(?X, ?Y,? Z)* into the grounded action *move(C, D, B)* by instantiating *?X* to *C*, *?Y* to *D*, and *?Z* to *B*. $\triangle$

A grounded action a is *applicable* to a state s if every ground atom in $\mathit{pre}(a)$ is also contained in s. When we apply a to s, we obtain a *successor state* s', defined as $s' = \{s \cup \mathit{del}(a)\} \setminus \mathit{add}(a)$. The intuition is that if we apply an action to a state, we add to this the properties (i.e., ground atoms) contained in the add list of the action and remove those contained in its delete list. The set of all successors of state s is written $\mathit{succ}(s)$. Analogously, a sequence of grounded actions $a_1, \ldots, a_n$ is applicable to a state s if each a_i is applicable to the state generated by applying $a_1, \ldots, a_{i-1}$ from s. The goal of a planner is to find a sequence of grounded actions $a_1, \ldots, a_n$ applicable to s_0 and leading to some goal state s_γ such that $\gamma \subseteq s_\gamma$. This sequence of actions is called a *plan*.

Our definition is focused on STRIPS domains [Fikes and Nilsson, 1971], but it can be generalized to other formalisms. The important aspect of such restriction is that we do not consider conditional effects, axioms, and negated preconditions. Many planning domains also use typed objects. We can easily remove the types by adding one new predicate for each type, and setting the initial state and action preconditions consistent with the original types for each object, similarly as performed by Helmert [2009].

However, there is still something missing in our planning task. To solve the task, we need to find a plan for it, which consists of a sequence of grounded actions. But all we have from the definition of Π is a set of action schemas. We say that our action schemas are *lifted* (as opposed to grounded) and we want to obtain the grounded actions from these schemas.

The most common method to obtain a set of grounded actions from a lifted representation is to pre-compute a subset of all possible grounded actions and use this new representation with grounded actions. Ideally, one wants to compute the minimal subset of actions that can ever be part of any plan from s_0 to some goal state. Unfortunately,

```
(define (domain visit-all)
  (:predicates (connected ?x ?y)
               (at-robot ?x)
               (visited ?x)
               (cell ?x)
  )
  (:action move
   :parameters (?curpos ?nextpos)
   :precondition (and (at-robot ?curpos)
                      (connected ?curpos ?nextpos)
                      (cell ?curpos)
                      (cell ?nextpos))
   :effect (and (at-robot ?nextpos)
                (not (at-robot ?curpos))
                (visited ?nextpos)))
)
```

Figure 2.1: Description of the Visit-All domain in PDDL. The section `:predicates` defines $\mathcal{P}$, and `:action` defines a single action of $\mathcal{A}$. (There can be multiple `:action` sections.)

this procedure is not easy and overapproximations are usually used. Thus, although the description was given in a lifted representation, a planner compiles it into a grounded representation of the task, which might increase the number of actions exponentially in the number of action schemas [Erol *et al.*, 1995]. This is what we call *grounded planning*: we preprocess all action schemas to generate a set of grounded actions and use this new set in the planning algorithm.

An alternative way to grounded planning is the so-called *lifted planning*. In this setting, the state is still represented by a set of ground atoms, while the only information regarding actions used by the planner is the set of lifted action schemas provided by the domain description. The instantiation of the free variables of the action schemas is performed while planning, not in a preprocessing stage.

There is always a trade-off between these two representations. In grounded planning, by pre-computing a set of grounded actions before actually executing a planning algorithm, we might save the computational effort of instantiating all schemas in many different iterations of the solver. On the other hand, by planning using lifted representation, we might save resources (e.g., memory) by not pre-grounding all action schemas, since many of the instantiations will never be part of any plan.

Planning models are usually described using some description language. One of the most used is PDDL [McDermott, 2000]. In PDDL, two different files are used to describe the task: a domain and an instance file. The former includes all the high-level characterizations of the problem, such as the action schemas and predicates. The latter, as the name indicates, is an instantiation of the problem, where we define the set of objects O, the initial state s_0 and the goal condition γ.

Example 2.3 (Visit-All domain and instance)**.** We illustrate the definitions above with the Visit-All domain defined in Figure 2.1. In this domain, an agent must visit every cell of a grid. The domain description is given in PDDL. We do not explain the PDDL syntax in detail, since the elements needed for the formal definition of a domain are easy

```
(define (problem grid)
 (:domain visit-all)
 (:objects
        loc-x0-y0 loc-x0-y1 loc-x1-y0 loc-x1-y1
 )
 (:init (at-robot loc-x1-y1)
        (visited loc-x1-y1)
        (connected loc-x0-y0 loc-x1-y0)
        (connected loc-x0-y0 loc-x0-y1)
        (connected loc-x0-y1 loc-x1-y1)
        (connected loc-x0-y1 loc-x0-y0)
        (connected loc-x1-y0 loc-x0-y0)
        (connected loc-x1-y0 loc-x1-y1)
        (connected loc-x1-y1 loc-x0-y1)
        (connected loc-x1-y1 loc-x1-y0)
        (cell loc-x0-y0)
        (cell loc-x0-y1)
        (cell loc-x1-y0)
        (cell loc-x1-y1))
 (:goal (and (visited loc-x0-y0)
             (visited loc-x0-y1)
             (visited loc-x1-y0)
             (visited loc-x1-y1)))
)
```

Figure 2.2: PDDL definition of an instance for the Visit-All domain. Section `:object` defines O, section `:init` defines the initial state s_0, and `:goal` defines the goal condition γ.

to observe. Note that we rewrite the predicates from PDDL syntax format *(P ?x)* to the more classical notation of *P(?x)*.

The domain defines the following elements

$$\begin{aligned}\mathcal{P} =& \{\textit{connected}, \textit{at-robot}, \textit{visited}, \textit{cell}\},\\ \mathcal{A} =& \{\textit{move}[\textit{?curpos, ?nextpos}]\},\end{aligned}$$

where predicate *connected* is a binary predicate and the other three are unary predicates. The action schema *move*[*?curpos, ?nextpos*] is the triple

$$\begin{aligned}\textit{pre}(\textit{move}[\textit{?curpos, ?nextpos}]) =& \textit{at-robot(?curpos)} \wedge \textit{connected(?curpos, ?nextpos)}\\ & \wedge \textit{cell(?curpos)} \wedge \textit{cell(?nextpos)}\\ \textit{add}(\textit{move}[\textit{?curpos, ?nextpos}]) =& \{\textit{at-robot(?nextpos)}, \textit{visited(?nextpos)}\}\\ \textit{del}(\textit{move}[\textit{?curpos, ?nextpos}]) =& \{\textit{at-robot(?curpos)}\}.\end{aligned}$$

In words, the PDDL description has four predicate symbols indicating relations between objects: *connected* has two parameters and indicates whether two cells are connected in the grid; *at-robot* indicates in which cell the robot currently is; *visited* indicates whether a cell $?x$ has already been visited or not; *cell* identifies whether an object is a cell or not. To modify our world, we have only one action, *move*, which takes two parameters (intuitively, the current position of the agent and the position to where it wants to move) and checks where (i) the agent is really at *?curpos* and (ii) *?nextpos* is connected to *?curpos*. In PDDL, instead of add and delete lists, we have a conjunction of ground atoms and negated ground atoms representing the *effects* of the action schema. Without loss of generality, this formula can also be seen as a set of ground atoms, where the negated atoms correspond to the delete list and the positive ground atoms to the add list. When this action is applied to our world, we remove the ground atoms indicating that the robot is at *?curpos*, add the ground atoms indicating the new position of the agent and also add the information that the new cell has already been visited. Note that the action is lifted: it is in first-order logic and it has free variables. In order to apply this action into a state, we need to select which objects will instantiate the free variables and then replace the free variables with these objects. In a single state, there might be several different instantiations that lead to applicable grounded actions.

Figure 2.2 shows a possible Visit-All instance. The PDDL section *objects* defines all the objects O. (Note that the agent is not represented by any object, but this is just a modeling decision.) The *init* section defines the ground atoms that are true in the initial state s_0. This is a subset of the ground atoms based on $\mathcal{P}$ from the domain definition grounded by the objects O. In contrast to our definition, the *goal* section in PDDL defines the goal condition γ as a conjunction of ground atoms. As done with the action schemas effects, we can also interpret this conjunction as a set. Thus, we consider γ as a set of ground atoms in this work. $\triangle$

In the next section, we go from *how to represent planning tasks* to *how to solve planning tasks*. We introduce heuristic search, the main technique applied by state-of-the-art planners.

2.2 Planning as Heuristic Search

The most popular planning technique is *state space search*. This claim can be confirmed when we look into the participants of the IPC 2018: only one of the 22 competitors did not rely on heuristic search. A *state space* is a transition system containing all possible states of a planning task Π, where two states s_1 and s_2 have a transition $s_1 \rightarrow s_2$ iff $s_2 \in$ *succ*(s_1) and this transition is labeled with the grounded action that generated s_2 from s_1. It is straightforward to recognize that the task of finding a plan for Π is equivalent to finding a path in the state space from the initial state to a goal state. Similarly, the objective of *optimal planning* is to find the shortest of these paths.

Unfortunately, a state-space can have the size exponential in the number of atoms of the task and thus a brute-force search becomes intractable. For example, in a planning task of the Visit-All domain with n locations, the entire state space has $2^{n^2} \cdot 2^n \cdot 2^n \cdot 2^n$ states. If we consider only the part of the state space that is *reachable* from the initial state, the number of states decreases by at least by a factor of 2^{n^2}. However, such a solution is still not tractable, since deciding which states are reachable is as hard as solving the planning task itself.

To handle large state spaces, planners apply a *heuristic search* approach. The planner uses some estimator to evaluate how promising it is that a given state leads the search to a goal state. With this evaluator, the planner prioritizes the most promising states, expecting that they lead the search to a goal state more quickly. A *heuristic* is formally defined as a function mapping a state to a non-negative number, estimating the distance from the state to its closest goal state or to infinity, when no such goal state can be reached from the state. The quest to find good heuristics for planning is an important

A	B
0	1
1	1
2	2
3	1
3	3

(a) Relation $R_1(A, B)$.

B	C
0	1
1	2
1	3
1	4
2	4

(b) Relation $R_2(B, C)$.

A
0
1
2
3

(c) $\pi_A(R_1(A, B))$.

A	B
1	1
2	2
3	3

(d) $\sigma_{A=B}(R_1(A, B))$.

Figure 2.3: Example of database relations and operations.

area of research.

Definition 2.2 (Heuristic). *Given a state space $\mathcal{S}$ with set of states S, a* heuristic *is a function $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$.*

For a given state-space $\mathcal{S}$ with a set of states S and a state $s \in S$, we denote the length of the shortest path from s to a goal state using $h^*(s)$. A heuristic is said to be *admissible* if it never overestimates the distance from s to the closest goal state. In other words, h is admissible iff $h(s) \leq h^*(s)$ for every state s of the state space S. Admissible heuristics are important because they can guarantee optimal solutions when used with specific search algorithms, such as the *A* algorithm* [Hart *et al.*, 1968]. When combined with an admissible heuristic, A* guarantees an optimal solution, when a solution exists. When not interested in an optimal solution, one could use a *greedy best-first search* (GBFS), introduced by Doran and Michie [1966], which usually is faster than A* but does not guarantee optimality. Both algorithms are based on *expansions*: starting with a list containing only the initial state s_0, we iteratively select the most promising state s (according to some arbitrary function) of this list and expand s. "Expand a state" means that we generate all successors of s and place them on the list. We repeat this process until we select a goal state for expansion, indicating that we found a path in the state space from s_0 to some goal state. We do not go into further details or the differences of A* and GBFS and we refer the reader to the classic book by Russell and Norvig [2010] for an introduction to search algorithms.

Heuristic search involves the question of whether we should use grounded or lifted planning. The grounding effort in grounded planning is independent of the heuristic since we compute a set of grounded actions without making use of any heuristic information (at least, that is the case in the given methods in the literature, to the best of our knowledge). In contrast, the grounding effort necessary in a lifted task representation (with concrete states and lifted action schemas) is directly linked to the quality of the heuristic and the number of expansions needed to achieve a goal state. In lifted planning, we instantiate the action schemas only when expanding a state. Since, in general, more informative heuristics lead the search to a goal state more quickly and thus require fewer expansions, they might help to minimize the overhead added by planning using lifted representations.

2.3 A Database Theory Perspective

Next, we change our perspective with respect to planning. We reformulate the representation of states and state spaces from the point-of-view of database theory, more specifically from relational algebra. The definitions and terminology we use are based on the book by Ullman [1988; 1989].

2.3.1 Basics of Database Theory

We start by introducing the *relational model*. Let D be a *domain* and $\boldsymbol{X} = (X_1, \dots, X_n)$ be a sequence of names, also called as *attributes*. We define *unique*$(\boldsymbol{X}) = \{X_i \mid 1 \leq i \leq n\}$ as the set of unique names in $\boldsymbol{X}$. A *named tuple* is a function t : *unique*$(\boldsymbol{X}) \to D$. For a specific tuple $r = (a_1, \dots, a_n)$ we define

$$t_r(\boldsymbol{X}) = \{X_i \mapsto a_i \mid 1 \leq i \leq n\}.$$

In our context, we define a *database* as a set of *unnamed relations*. An n-ary unnamed relation $R \subseteq D^n$ is a set of tuples over domain D for some $n \in \mathbb{N}^0$. Given an unnamed relation R and attributes $\boldsymbol{X}$, $R(\boldsymbol{X})$ is an n-ary *relation* and is defined as

$$R(\boldsymbol{X}) = \{t_r(\boldsymbol{X}) \mid r \in R \text{ and } t_r(\boldsymbol{X}) \text{ is well defined}\}.$$

When we want to specify the value of a single element X_i of $\boldsymbol{X}$ in a tuple $r = (a_1, \dots, a_n)$, we write $t_r(X_i) = a_i$. We only use this notation when the sequence of names $\boldsymbol{X}$ in which X_i belongs is clear from context.

Whenever we want to denote that a named relation has many named attributes, but we do not care specifically about any of them, we denote these attribute names using bold uppercase variables, such as in $R(\boldsymbol{X})$. With some abuse of notation, we use set-theory symbols to compare sequences of names. For example, if we want to indicate that all attributes in $\boldsymbol{Y}$ are also in $\boldsymbol{X}$ (but not caring about order), we write $\boldsymbol{Y} \subseteq \boldsymbol{X}$. We extend the same reason to the union and set difference operators.

It is useful to think of relations as *tables* with named columns (the attributes), where each row is a tuple mapping a name to a value of a given domain. For example, assume we have $D = \mathbb{Z}$ and $R \subseteq D^3$ is defined as $R = \{(1,1,1), (0,0,1), (1,2,3)\}$. If the attributes $\boldsymbol{X}$ are defined as $\boldsymbol{X} = (A, A, B)$, then the ternary relation $R(\boldsymbol{X})$ has the following named tuples:

$$R(\boldsymbol{X}) = \{t_{(1,1,1)}(\boldsymbol{X}), t_{(0,0,1)}(\boldsymbol{X})\},$$

where

$$t_{(1,1,1)}(\boldsymbol{X}) = \{A \mapsto 1, B \mapsto 1\},$$
$$t_{(0,0,1)}(\boldsymbol{X}) = \{A \mapsto 0, B \mapsto 1\}.$$

The third tuple of R, $(1,2,3)$, is not part of $R(\boldsymbol{X})$ because $t_{(1,2,3)}(\boldsymbol{X})$ is not well-defined.

In order to simplify the notation, we usually denote relations by simply using the tuples corresponding to the named tuple functions. For example, we can write $R(\boldsymbol{X})$ simply as

$$R(\boldsymbol{X}) = \{(0,0,0), (1,1,1)\}.$$

Example 2.4. Assume that a database has two relations $R_1(A, B)$ and $R_2(B, C)$, as showed in Figure 2.3a and Figure 2.3b, respectively. The top-row of each column indicates the attribute name. The former has two attributes, A and B, and has tuples $(0,1), (1,1), (2,2), (3,1)$ and $(3,3)$, where the first element of each tuple is the value for attribute A and the second is the value for B. The latter also has two attributes, B and C, and five tuples: $(0,1), (1,2), (1,3), (1,4)$, and $(2,4)$, where the first value corresponds to attribute B and the second to C. $\triangle$

A	B	C
0	1	2
0	1	3
0	1	4
1	1	2
1	1	3
1	1	4
2	2	4
3	1	2
3	1	3
3	1	4

(a) $R_1(A,B) \bowtie R_2(B,C)$.

A	B
0	1
1	1
2	2
3	1

(b) $R_1(A,B) \ltimes R_2(B,C)$.

B	A
0	1
1	1
2	2

(c) $R_1(B,A) \ltimes R_2(B,C)$.

Figure 2.4: Example of database join and semi-join operations.

We can use the relations of a database to infer further knowledge about it. In the previous example, for instance, one might be interested to know which values of B in $R_1(A,B)$ are also associated with some C in $R_2(B,C)$. (Spoiler: the answer is $\{1,2\}$.) We can formalize these inference procedures using *database operations*. Such operations produce new relations that correspond to the ones we want to infer.

Cartesian Product. Let R and S be two relations of arity n and m and $\boldsymbol{X} = (X_1, \ldots, X_n)$, $\boldsymbol{Y} = (Y_1, \ldots, Y_m)$ be two sequences of names. Then the *Cartesian product* $R(\boldsymbol{X}) \times S(\boldsymbol{Y})$ is a relation over the names $\boldsymbol{Z} = (R.X_1, \ldots, R.X_n, S.Y_1, \ldots, S.Y_m)$ with

$$R(\boldsymbol{X}) \times S(\boldsymbol{Y}) = \{t_{t_1 \times t_2} \mid t_1 \in R(\boldsymbol{X}), t_2 \in S(\boldsymbol{Y})\}$$

where $t_{t_1 \times t_2}(R.X_i) = t_1(X_i)$, for all $1 \leq i \leq n$, and $t_{t1 \times t2}(S.Y_i) = t_2(Y_i)$, for all $1 \leq i \leq m$.

Projection. A *projection* $\pi_{\boldsymbol{Y}}(R(\boldsymbol{X}))$, where $\boldsymbol{Y}$ is a sequence of names such that all elements of $\boldsymbol{Y}$ are also contained in the sequence of names $\boldsymbol{X}$, is defined as $\pi_{\boldsymbol{Y}}(R(\boldsymbol{X})) = \{t|_{\boldsymbol{Y}} \mid t \in R(\boldsymbol{X})\}$, where $t|_{\boldsymbol{Y}}$ represents the function t restricted to the names of $\boldsymbol{Y}$.

Selection. Given an equality comparison $X_i = X_j$ for $X_i, X_j \in \boldsymbol{X}$, we say that the *selection* $\sigma_{X_i=X_j}(R(\boldsymbol{X}))$ is defined as

$$\sigma_{X_i=X_j}(R(\boldsymbol{X})) = \{t \mid t(X_i) = t(X_j)\}.$$

If we have n equality comparisons $E_1, \ldots, E_n$, we can write $\sigma_{E_1,\ldots,E_n}(R(\boldsymbol{X}))$ instead of $\sigma_{E_1}(\sigma_{E_2}(\ldots(\sigma_{E_n}(R(\boldsymbol{X})))))$.

Join. Let R and S be two relations of arity n and m and $\boldsymbol{X} = (X_1, \ldots, X_n)$, $\boldsymbol{Y} = (Y_1, \ldots, Y_m)$ be two sequences of names. Then the (natural) *join* $R(\boldsymbol{X}) \bowtie S(\boldsymbol{Y})$ is a relation over the names $\boldsymbol{Z} = \boldsymbol{X} \cup (\boldsymbol{Y} \setminus \boldsymbol{X})$ with

$$\begin{aligned} R(\boldsymbol{X}) \bowtie S(\boldsymbol{Y}) = &\{t' \mid t \in R(\boldsymbol{X}) \times S(\boldsymbol{Y}), \\ &\text{s.t. } t(R.X_i) = t(S.X_i) \text{ for all } X_i \in \{\boldsymbol{X} \cap \boldsymbol{Y}\}\} \end{aligned}$$

where $t'(X_i) = t(R.X_i)$, for all $1 \leq i \leq n$, and $t'(Y_i) = t(S.Y_i)$, for all $Y_i \in \{\boldsymbol{Y} \setminus \boldsymbol{X}\}$. As in the projection, duplicated tuples are removed.

Semi-join. A *semi-join* of $R(\boldsymbol{X})$ by $S(\boldsymbol{Y})$ is the projection of the join $R(\boldsymbol{X}) \bowtie S(\boldsymbol{Y})$ onto the attributes $\boldsymbol{X}$. Using our notation, it is equivalent to $\pi_{\boldsymbol{X}}(R(\boldsymbol{X}) \bowtie S(\boldsymbol{Y}))$. The semi-join operation of $R(\boldsymbol{X})$ by $S(\boldsymbol{Y})$ is denoted as $R(\boldsymbol{X}) \ltimes S(\boldsymbol{Y})$.

We illustrate these operations with a few examples.

Example 2.5. Figures 2.3c and 2.3d illustrate examples of the projection and selection operations, respectively, over relation $R_1(A, B)$. The join and the semi-join operations are shown in Figures 2.4a and 2.4b, based on $R_1(A, B)$ and $R_2(B, C)$ from Figure 2.3. The Cartesian product operation is not illustrated.

Figure 2.4c shows the same semi-join operation as Figure 2.4b but switching the attribute names of the first relation from (A, B) to (B, A). Note that although the set of tuples in $R_1(A, B)$ and $R_1(B, A)$ still comes from the same unnamed relation R_1, the different sequence of names results in a different relation. $\triangle$

Imagine that we have two relations in our database: a relation $P(I, V)$, recording the purchases by clients in some online store during the last week, with tuples (i, v) where i is the ID number of the client and v is the value of the purchase; and a relation $R(I, D)$, recording the date where the clients first registered into the online store service, with tuples (i, d) where i is the ID number of the client and d is the registration date. Given these two relations, we might be interested to infer the registration dates from clients who purchased any item in the last week. This type of question-based on the knowledge in our database is called a *query*. Roughly speaking, a query is a formal way to express a question about our database. The answer to a query $Q(\boldsymbol{X})$ is always a relation with attributes $\boldsymbol{X}$. In this thesis, we work with a restricted type of queries, called *conjunctive queries* [Ullman, 1989]. We introduce the logical view of queries first. Formally, a conjunctive query is a first-order formula in the form

$$(\exists Z_1) \dots (\exists Z_m) \psi(X_1, \dots, X_n, Z_1, \dots, Z_n),$$

where $\psi(X_1, \dots, X_n, Z_1, \dots, Z_n)$ is a *conjunction of relations*. The relation with the tuples instantiating the free variables $X_1, \dots, X_n$, such that the formula above is satisfied for every tuple in the relation, corresponds to the answer of the query. In our online store example, we can represent the registration dates from clients who purchased some item in the last week by the first-order formula

$$(\exists I)(\exists V)(P(I, V) \wedge R(I, D)).$$

Instead of writing first-order logic sentences to represent queries, we use Prolog notation [Kowalski, 1979]. For example, the query above can be rewritten using Prolog notation as

$$Q(D) :- P(I, V), R(I, D).$$

We say that $Q(D)$ is the *head* of the query, while $P(I, V), R(I, D)$ are the *body* relations of the query. The head of the query contains only the free variables of the formula. The meaning of this query is to retrieve the possible values for the free variable D that (i) are in the second element of some tuple in $R(I, D)$ and (ii) are associated to some value of I in $R(I, V)$ such that this value also occurs as the first element of some tuple in $P(I, V)$. We also do not care about the values assigned to the variables I and V, since are not in the head.

```
(:action pick
    :parameters (?ball ?room ?gripper)
    :precondition (and (ball ?ball) (room ?room)
                       (gripper ?gripper) (at ?ball ?room)
                       (at-robby ?room) (free ?gripper))
    :effect (and (carry ?ball ?gripper)
                 (not (at ?ball ?room))
                 (not (free ?gripper))))
```

Figure 2.5: PDDL action schema corresponding to the action of picking up a ball from a room using a specific gripper.

We can interpret conjunctive queries using the database operations described above. The logical conjunction between two relations is equivalent to the join of these relations. The existential quantification over variables $Z_1, \ldots, Z_m$ is equivalent to projecting the query only over variables $X_1, \ldots, X_n$ in the head of the query. In our running example, $Q(D)$ is equivalent to the following relation:

$$\pi_D(P(I, V) \bowtie R(I, D)).$$

The query answer $Q(D)$ is the relation containing only tuples satisfying the first-order formula corresponding to the query.

An interesting practical aspect of conjunctive queries is that they can be computed using only joins (or semi-joins), projections, and selections. However, this does not mean that the computation is always tractable. It might happen that, while computing the query answer, intermediate results have an exponential number of tuples in the size of the final relation. We discuss the tractability issues of conjunctive queries in Chapter 4 in the context of classical planning.

Given this short introduction to database theory, we can start to show how these definitions might help our understanding of planning.

2.3.2 Planning as Database Progression

The perspective used in this thesis is to consider states of the planning task as databases. From this point-of-view, given a planning task $\Pi = (\mathcal{P}, \mathcal{A}, O, s_0, \gamma)$, we consider that each predicate symbol $P \in \mathcal{P}$ corresponds to an unnamed relation in a given state s, where a tuple $(o_1, \ldots, o_n)$ with $o_1, \ldots, o_n \in O$ is in P at state s iff $P(o_1, \ldots, o_n)$ is true in s. Intuitively, every state can be represented as a set of unnamed relations, where each predicate corresponds to a single set of tuples. Thus, each state can be seen as a database and the application of an action in a state can be seen as an update of the database.

Now, planning can be reformulated as finding a sequence of updates from an initial to a goal database. This reformulation is not novel in the literature. Lin and Reiter [1997] define a state as a set of unnamed relations, each corresponding to a predicate of the task. They interpret each STRIPS operator (i.e., action) as an update to such a database and denote the task of planning as *database progression* from an initial state to a goal. In their context, Lin and Reiter use situation calculus and even second-order logic to express more powerful variants of STRIPS, while we use relational calculus and simply first-order logic fragments and focus on "canonical" STRIPS tasks (i.e., tasks where preconditions are conjunctions, negative literals are forbidden in the preconditions, effects are sets of atoms).

The reformulation of states as databases leads to an elegant view of action schemas in lifted planning, as briefly demonstrated in Chapter 1. Applying one action to a state is equivalent to updating the values of its unnamed relations. It turns out that verifying which instantiations of an action schema are applicable to a state is equivalent to solving a conjunctive query. We do that by representing the precondition of each action schema as a conjunctive query, where each atom appearing in the precondition is represented by a relation with attributes defined by the free variables of the action schema. Since we consider that a precondition of a ground action is *"satisfied"* if all its atoms are true in a state, we can also see it as a conjunctive formula over its ground atoms. Thus, finding the tuples leading to instantiations where the preconditions are satisfied is equivalent to solve a conjunctive first-order logic formula, which in turn can be represented by some conjunctive query. Hence, we can consider the task of finding tuples satisfying a precondition the same as answering the correspondent conjunctive query.

We illustrate our perspective with an example.

Example 2.6 (Gripper action schema)**.** We show how to ground an action schema of the Gripper domain. In this domain, a robot with some grippers must pick up a set of balls from different rooms and transport them to another room. The action schema used is shown in Figure 2.5. It corresponds to the action of picking up a ball with a specific gripper in a given room. In a state s, we want to find which substitutions for the free variables of the action schema satisfy the precondition. Since the predicate symbol of each atom corresponds to an unnamed relation in the database, we perform the conjunctive query

$$\begin{aligned} \mathit{Q(?ball, ?room, ?gripper)} :- \; & \mathit{ball(?ball)}, \mathit{room(?room)}, \mathit{gripper(?gripper)}, \\ & \mathit{at(?ball, ?room)}, \mathit{at\text{-}robby(?room)}, \mathit{free(?gripper)}. \end{aligned}$$

The resulting relation is the set of tuples, where each tuple has one element corresponding to each of the free variables of the query. For example, we could have the tuple (b_1, r_1, g_1) in the resulting relation for a state s, where the first parameter indicates a substitution for *?ball*, the second for *?room* and the third for *?gripper*. Since b_1 is a possible substitution for *?ball* resulting in a sequence of joins, it means that there is a tuple containing b_1 in the relation *ball(?ball)* and *at(?ball, ?room)* in a state s. Progressing this reasoning to all elements of the tuple, we can see that every tuple found by this conjunctive query in s represents an instantiation of the schema such that all atoms in the precondition are in s when instantiated. In other words, every tuple present in the resulting relation *Q(?ball, ?room, ?gripper)* is an instantiation of the free variables corresponding to an applicable grounded action in s. By performing a conjunctive query for each action schema in a state s, we obtain the exact grounded actions that are applicable to s. $\triangle$

More formally, given an action schema $a[\Delta]$ with precondition $\mathit{pre}(a[\Delta])$, we create a query $Q(\boldsymbol{X})$ where the sequence $\boldsymbol{X}$ has one attribute corresponding each free variable in Δ. The body of the query has one relation for each atom in $\mathit{pre}(a[\Delta])$ with attributes accordingly to $\boldsymbol{X}$.

• • •

In Chapter 3, we present work related to this thesis. We discuss previous lifted planning approaches and some techniques related to the question of grounded versus lifted planning. Last, we also introduce some related work on database theory.

In the following chapters, we investigate how to efficiently obtain the set of applicable grounded action using queries based on the preconditions of the action schemas. We also show that in most IPC domains, such queries can be efficiently computed.

Chapter 3

Related Work

Planning using lifted task representations has been an interesting topic of research in the planning community in the last decades. Perhaps the closest work to our thesis is the work by McDermott [1996]. McDermott proposes a heuristic estimator based on a so-called regression-match graph, which estimates the number of operators necessary to achieve each subgoal of the task by applying an inference method similar to backward-chaining [Hewitt, 1969]. His representation of a lifted task uses concrete states and lifted action schemas. This work was also the base for the introduction of planning as heuristic search by Bonet and Geffner [2001] and the heuristics introduced by them, such as h^{add}, are similar to the estimators used by McDermott.

Another important study of lifted planning was done by Ridder [2014]. Ridder proposes a lifted representation very similar to the one used by McDermott, using lifted actions and concrete state representation. Ridder also uses a so-called *almost equivalent* relations between PDDL objects of a same type. Once such relations are encountered, it is possible to abstract several equivalent objects in a lifted representation. He also demonstrates how to use such representation to compute a lifted version of the FF heuristic [Hoffmann and Nebel, 2001] and several other heuristics based on abstractions.

Previous work also explores lifted representations using other planning techniques. Among these, the VHPOP planner [Younes and Simmons, 2003] is perhaps the closest to our work. VHPOP was influenced by UCPOP [Penberthy and Weld, 1992], another partial-order planner from the early 1990s. In partial-order planning, the planner searches for a plan in a *plan space* instead of performing a state-space search. VHPOP combines partial-order planning with the heuristic estimate h^{add} introduced by Bonet and Geffner [2001], adapted to the context of partial-order planning. Although the heuristics implemented in VHPOP can deal with lifted and grounded actions, Younes and Simmons use only grounded actions in their IPC participation. In their previous work, Younes and Simmons [2002] also show that partial-order planning with lifted actions reduces the branching factor while searching in plan space and that it can be beneficial in some circumstances. The work by Ridder [2014] is also based on ideas from partial-order planning.

An important part of our work is the relationship between planning and database theory. This relation is not novel though. For example, Helmert [2009] reformulates a planning task into a Datalog program (see Ullman [1988] or Abiteboul *et al.* [1995]) in order to compute a relaxed reachability analysis and uses more sophisticated database techniques to perform this computation more quickly. As mentioned in Chapter 2, Lin and Reiter [1997] study the possible interpretation of action schemas as mappings from databases to databases. Using database theory terminology, Lin and Reiter define a state as a database, where each relation corresponds to a STRIPS predicate and the sets of tuples in a relation are the tuples of objects instantiating such predicate in this state. They interpret each STRIPS operator as an update to the database representing a state, what they call a *database progression*. This is essentially the same interpretation used in this thesis. They use situation calculus to introduce the idea of planning as database progression and show that in many formalisms of planning (more precisely, in simple extensions of STRIPS) we need to use second-order logic to be able to use such database progression idea. Although Lin and Reiter, as in this thesis, use the interpretation that a planning problem can be seen as a database progression problem, all their results are proved using situation calculus. In our approach, however, we instead apply results from database theory and rely on relational algebra and heuristic search to solve the tasks.

Another important part of our thesis is to compare heuristic search planners using both lifted and grounded representation. The state-of-the-art heuristic search planners use grounded representations. Thus, they also need to use different techniques to try to mitigate the overhead caused by grounding. Many of the heuristic search planners are extensions of the Fast Downward planning system [Helmert, 2006]. Fast Downward uses a finite domain representation (FDR) of the tasks and it is necessary to translate the tasks from PDDL to SAS$^+$ [Bäckström and Nebel, 1995]. The translation procedure used in Fast Downward is introduced by Helmert [2009]. The complete procedure consists of four steps: normalization, invariant synthesis, grounding, and FDR task generation. The bottleneck (and probably the greatest improvement introduced by Helmert [2009]) is the grounding procedure. The key insight of this procedure is to only ground relaxed-reachable actions, those actions that possibly can be applied in a task when ignoring (i) the delete lists in the effects of actions, and (ii) negative preconditions. It is easy to check that all actions necessary by some plan from the initial state must be relaxed-reachable. As mentioned earlier, Helmert introduces a Datalog program to represent a planning task without delete lists and negative preconditions. By computing the canonical model of this Datalog program, we can identify which actions are relaxed-reachable and should be grounded.

In fact, Helmert [2009] demonstrates empirically that his grounding procedure is significantly better than the naive grounding approach (i.e., instantiating all possible combinations of grounded actions) in terms of the number of grounded actions and time. It is possible to see that this still holds for some IPC 2018 domains. For example, the first (and easiest) instance from the Organic Synthesis domain generates $4.3 \cdot 10^6$ actions using the naive grounding by enumeration, while the method by Helmert generates only 360 grounded actions. There are still cases, however, where scalability is an issue even when using the methods by Helmert. In the same Organic Synthesis domain, there are instances where the naive grounding approach instantiates approximately $71 \cdot 10^{12}$ actions, while the method by Helmert generates $884,400$ grounded actions. Although the latter is a significant improvement compared to the former, this is still above the capabilities of most planners. Another interesting aspect is to compare the number of grounded actions with the plan length of the same instances. In the eleventh instance of the Organic Synthesis domain, for example, where the grounding method by Helmert produces 884.400 grounded actions, we find an optimal plan with just three actions. By comparing the number of grounded actions and the plan length, we can see that most of the grounded actions are not used and thus any planner would be better off by not grounding them.

Yet, there is still room for improvement with respect to grounding algorithms, such as symmetry-based task reduction [Röger *et al.*, 2018] or grounding only parts of the relaxed-reachable state space [Gnad *et al.*, 2019]. The work by Röger *et al.* [2018] introduces a preprocessing method able to capture object symmetries in the PDDL task and then performs the grounding for the entire symmetric group, instead of individually grounding each object. Gnad *et al.* [2019] propose a partial ground method based on learning. Although their method seems to improve results in many domains, it is limited to domains where representative but small instances are easily generated. This limitation is a significant constraint for domains that are inherently hard to ground even for the smallest instances, such as the Organic Synthesis domain.

Another popular method for reducing the grounding overhead is predicate and action splitting. In the first paper proposing planning as a satisfiability problem, Kautz and Selman [1992] propose to split predicates with three or more arguments in order to facilitate the instantiation of these predicates for the SAT solver. They demonstrate in practice that such splitting is useful and improves the performance of the solver. Still in the context of planning as satisfiability, Robinson *et al.* [2009] propose a split representation of actions and showed that, similarly as demonstrated by Kautz and Selman, such splitting increases the performance, despite the size increase of the logic formulations. Cashmore *et al.* [2013] also show a split representation of planning tasks using quantified Boolean formulas (QBF) and partially lifted representation of actions and states. In the context of planning as heuristic search, Areces *et al.* [2014] propose an automatic way to perform action schema splitting (using the PDDL representation) in order to reduce the number of parameters of each action schema. In practice, their method seems very useful even for the IPC 2018 domains that are hard to ground. For instance, the Organic Synthesis domain is significantly easier to ground using such action split schema. This was the strategy used in the IPC 2018 and most of the planners presented a better performance in the split variant of the domain.

Other approaches to planning also make use of more sophisticated tools to deal with grounding. An interesting case is the planner *plasp* [Gebser *et al.*, 2011], which compiles a PDDL planning task into an Answer Set Programming (ASP) problem [Simons *et al.*, 2002] and then uses a specialized ASP solver. Although not properly a lifted planner, *plasp* makes use of the state-of-the-art ASP solvers and their efficient grounders to handle larger tasks. Similarly, Zhou *et al.* [2015] introduce a new planner modeled using Picat, a logic-based multiparadigm language, which uses tabled logic programming to solve planning tasks. Due to this strategy, Picat avoids prior grounding and thus has a lower memory consumption than most of the search-based planners.

• • •

The next chapter introduces our main contributions to lifted planning. Our focus is on lifted successor generators and how to efficiently compute them in the context of planning. We analyze several aspects of the complexity of conjunctive queries and show that several planning domains are in tractable cases. Furthermore, we also introduce extensions that are useful in the context of classical planning in order to improve the performance of these successor generators.

Chapter 4

Lifted Successor Generators

In this chapter, we introduce the main ideas to generate successor states using only lifted action schemas. This is one of the two main challenges for planning using heuristic search with lifted task representations. (The other one is to come up with good heuristics using only the actions schemas.) The essential problem here is not to generate a successor *per se*, but to identify which instantiations of an action schema are applicable to a given state. To do so, we must find the exact substitutions of free variables of an action schema such that the preconditions of this action are satisfied in the current state. In this sense, we are actually discussing methods to check the applicability of actions using only their schemas. However, we call these methods as successor generator methods because some of them, in fact, change the set of transitions to successors of some states. We introduce, for example, a method that identifies transitions leading to the same successors while computing the applicable instantiations of a schema and thus might reduce the branching factor of the state (i.e., number of transitions from this state).

In grounded planning, we can simply iterate over all grounded actions and apply those with preconditions satisfied. Thus, we always obtain all successor states of any state in $O(|G|)$ time, where G is the set of grounded actions. It is not uncommon that planners also use specialized data structures to generate successors more efficiently. But in grounded planning it is assumed that the grounding step was performed efficiently. The overall quality of a planner relies not only on its planning component but also on its grounding. For example, as mentioned in Chapter 3, a naive grounding enumerating all possible grounded actions might generate more than 10^{12} actions in the Organic Synthesis domain. This large number of actions might cause a planner to run out of resources even though most of the actions are not useful in any plan.

This is a possible advantage of lifted planning. It might be useful to avoid the grounding procedure in several circumstances and simply instantiate the lifted action schemas to produce all applicable instantiations only when expanding a state. However, a bad implementation might be even worse than the naive grounding by enumeration.

In this chapter, we study different methods with which we can implement lifted successor generators. We start by analyzing the trivial methods to obtain successor states in a lifted representation and end up presenting more refined methods. As previously mentioned in Chapter 1, our methods make use of database theory results.

4.1 Naive Successor Generators

The most straightforward method to implement a lifted successor generator is by brute-forcing all action schema instantiations. Given an action schema $a[\Delta]$ and a set of objects O, we enumerate all instantiations of variables in Δ by objects in O. This method is not expected to perform well, except in very small instances. In fact, in the context of lifted planning, this enumeration must be performed for every expanded state. This brute-force successor generator is not considered in the rest of this work.

But this method can be easily improved. For example, we can use the information of static predicates (i.e., predicates that never occur in the effect of any action) to reduce the set of candidate enumerations. These types of improvements are similar to the ones described by Helmert [2009] in the context of grounding algorithms. In his context, Helmert is able to reduce the number of grounded actions simply by checking types and static preconditions and discarding partial instantiations of action schemas that violate such conditions while computing the grounding. In the context of lifted planning, these enhancements can also improve the performance of the brute-force generator.

We now come back to the perspective of planning as database progression. We can bring some sophistication from relational algebra to our successor generators. A trivial way to do so is to consider the precondition of an action schema as a conjunctive query. The answer to this query contains only the tuples representing applicable instantiations of the schema in this state. This idea was formally introduced in Chapter 2. The notion behind this successor generator is that, for some action schema $a[\Delta]$ in some state s, each atom A in $\mathit{pre}(a[\Delta])$ has an unnamed relation in s associated to its predicate symbol. Each atom A also has a corresponding relation in the precondition with attribute names defined by the free variables. To find the instantiations of all free variables in $\mathit{free}(\mathit{pre}(a[\Delta]))$ satisfying all atoms in the precondition, we must perform the join program over the relations representing these atoms. (Since we argued before that all free variables appear in the preconditions because we preprocessed types, we can ignore effects for now.)

This is what we call the *join program successor generator*. A join program can be seen simply as a clever way to enumerate all applicable instantiations of free variables for an action schema, but with the clear advantage that it automatically filters out inapplicable instantiations as soon as it can. Hence, the join program successor generator is considered as a baseline in this thesis, since its efficiency (in theory) should not be worse than the complete enumeration.

The join program for a given precondition can be defined in many different ways, though. The key point here is that, in practice, the order in which the relations are joined might lead to very different performances. It is common that bad orderings produce intermediate results with size exponential in the input relations [Ullman, 1989]. In our experiments (Chapter 5), we study different ordering methods for these join programs.

4.2 Full Reducer Successor Generator

The join program successor generator introduced in the previous section is extremely dependent on the order of the join program. Luckily, there are some cases where we can compute an optimal order for this program in a feasible time. This provides an "island of tractability" for query answering problems [Gottlob *et al.*, 2016]. This "island" is defined based on the *hypergraph topology* of the conjunctive query. We next introduce

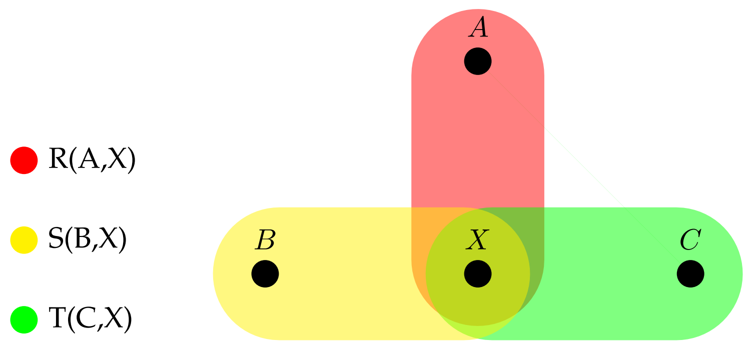


Figure 4.1: Example of hypergraph for the query $Q(A, B, C, X) \coloneq R(A, X), S(B, X), T(C, X)$.

this tractable case in the context of general conjunctive queries and then briefly explain its adaptation to planning.

4.2.1 Full Reducer

The efficiency for computing a conjunctive query depends on its *hypergraph*. A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ has a set of *hypervertices* $\mathcal{V}$ and a set of *hyperedges* $\mathcal{E} \subseteq 2^{\mathcal{V}}$. A hypergraph is a structure similar to a graph, but where hyperedges can connect more than two hypervertices. Every conjunctive query $Q(\boldsymbol{X})$ has a hypergraph associated to it. For every named attribute in $\boldsymbol{X}$ we have one associated hypervertex in $\mathcal{V}$. Similarly, every named relation $E(\boldsymbol{X})$ in the body induces a hyperedge including the hypervertices associated to the attributes $\boldsymbol{X}$. In the next paragraphs, we omit the attribute names, but they are still needed for the algorithm.

Hypergraphs can be classified as *cyclic* or *acyclic*. However, acyclicity of hypergraphs is not uniquely defined [Goodman and Shmueli, 1982; Beeri *et al.*, 1983; Fagin, 1983]. We consider here the definition of α-acyclicity [Fagin, 1983], which can be tested in polynomial time [Goodman and Shmueli, 1982; Tarjan and Yannakakis, 1984; Ullman, 1989]. We can compute the acyclicity of a hypergraph by iteratively selecting two hyperedges E and F and testing whether the nodes in $E \setminus F$ are unique to E. If they are, then we say that E is an *ear* and remove the hyperedge E from the hypergraph. We call this an *ear removal* and say that E *was removed in favor of* F. We repeat this procedure until we fall into one of these two cases:

1. There is only one hyperedge remaining. In this case, the hypergraph and *its corresponding conjunctive query are acyclic*.

2. There is no possible ear removal in the hypergraph. In this case, the hypergraph and *its corresponding conjunctive query are cyclic*.

There might exist several sequences of ear removal for a same hypergraph. However, any sequence of ear removals for an acyclic hypergraph leads to case (1). Analogously, if the hypergraph is cyclic, it always ends in case (2) [Ullman, 1989].

Example 4.1. Let us say that we are interested to know if the hypergraph of the following query acyclic:

$$Q(A, B, C, D, X) \coloneq R(A, X), S(B, X), T(C, X).$$

Figure 4.1 shows the hypergraph of $Q(A, B, C, X)$. The algorithm presented above works as follows. We select two hyperedges of the hypergraph arbitrarily, let us say $R(A, X)$ and $S(B, X)$, and test whether the hypervertices contained in $R(A, X) \backslash S(B, X)$ are unique to $R(A, X)$. The only hypervertex in this set difference is A and A is unique to $R(A, X)$. Thus, $R(A, X)$ is an ear and it is removed in favor of $S(B, X)$.

Next, we select the two remaining hyperedges $S(B, X)$ and $T(C, X)$ and test whether the hypervertices in $S(B, X) \setminus T(C, X)$ are unique to $S(B, X)$. Once again, the set difference contains only one hypervertex, in this case B. Since B is unique to $S(B, X)$, we remove $S(B, X)$ in favor of $T(C, X)$.

Now, we have only one hyperedge left in the hypergraph, indicating that the hypergraph is acyclic. (It represents the case (1.) above.) The order of ear removals was: (i) $R(A, X)$ was removed in favor of $S(B, X)$; and (ii) $S(B, X)$ was removed in favor of $T(C, X)$. △

If the hypergraph is acyclic, we can obtain from this sequence of ear removals a specific sequence of *semi-joins* ensuring that no intermediate relation will be larger than the input size plus the size of the final output relation. This order of semi-joins is computed as follows: start with an empty sequence σ and start from the last ear removal, for each ear removal of E in favor of F add $F \ltimes E$ to the *front of* σ and $E \ltimes F$ to the *back of* σ. In each semi-join $E \ltimes F$, we update the relation E to the resulting relation of this semi-join, and we denote it as $E := E \ltimes F$. In practice, we do not overwrite the relation E, but we store the intermediate results of E after performing the semi-joins in auxiliary variables.

The sequence σ as described above is called the *full reducer* semi-join program of the original query. A semi-join program fully reduces a relation E in the body of a given query $Q(\boldsymbol{X})$ if it eliminates from E all tuples not needed to answer $Q(\boldsymbol{X})$ [Bernstein and Goodman, 1981]. When we apply the full reducer semi-join program, all relations in the body of the query become fully reduced.

After computing the full reducer, we compute a sequence of joins from the reversed order of ears removals. If we removed ears in order $R_1, \ldots, R_n$ then we compute the final sequence of joins as $R_n \bowtie \ldots \bowtie R_1$ (from left to right). When we join $R_n \bowtie \ldots \bowtie R_{i+1}$ to R_i, the size of the intermediate relation cannot decrease, since every tuple of R_i joins with one or more tuples of $R_n \bowtie \ldots \bowtie R_{i+1}$. This occurs because the last semi-join that updated R_i filtered all tuples that do not join with any tuple in $R_n \bowtie \ldots \bowtie R_{i+1}$. This is true because R_i was removed in favor of some $R_j \in \{R_{i+1}, \ldots, R_k\}$ and, by definition of ear removal, R_j and R_i share at least one common attribute and the semi-joins $R_i \ltimes R_j$ and $R_j \ltimes R_i$ worked as filters for the final relations. As a consequence of this monotonic increase[1], no intermediate relation can have more tuples than the final relation (otherwise the monotonicity would be violated). We can see that such order of semi-joins and joins never generates intermediate relations larger than the input and output sizes. A more detailed explanation and intuition of the algorithm is given by Ullman [1989].

Example 4.2. Given the hypergraph of Figure 4.1 and the order of ear removals from

[1]Technically, it can happen that the join of the intermediate relation with R_i produces an empty relation (no tuple had matching values for the shared attributes). This is the special case the query has no tuple satisfying the body rule.

Example 4.1, the full reducer for this query is:

$$\begin{aligned}R(A,X) &\coloneqq R(A,X) \ltimes S(B,X)\\ S(B,X) &\coloneqq S(B,X) \ltimes T(C,X)\\ T(C,X) &\coloneqq T(C,X) \ltimes S(B,X)\\ S(B,X) &\coloneqq S(B,X) \ltimes R(A,X).\end{aligned}$$

The final sequence of joins, computed after the full reducer, is:

$$T(C,X) \bowtie S(B,X) \bowtie R(A,X).$$

The result of the query $Q(A,B,C,X)$ will be the relation computed from this join after computing the full reducer. △

In a broader view, *acyclic conjunctive queries* are very efficient to compute and the amount of memory needed is polynomial in the input plus the output sizes. Finding the full reducer semi-join program of a hypergraph can be done in linear time in the number of hyperedges [Yu and Ozsoyoglu, 1979]. The complexity of evaluating the full reducer semi-join program is $O(kI \log I)$, where I is the total input size and k is the number of relations in the body of the query (i.e., number of hyperedges). Similarly, the complexity of evaluating the final sequence of join operations is $O(k(I \log I + U \log U))$, where U is the output size of the query answer [Ullman, 1989].

In contrast, *cyclic conjunctive queries* do not have such guarantees of efficiency, since there is no full reducer semi-join program for them [Bernstein and Goodman, 1981]. When computing a cyclic conjunctive query, we might have intermediate states of the computation that are exponentially larger than the final output. These very large intermediate states cannot be always avoided and some cyclic queries have no ordering of join operations that eliminates large intermediate states.

4.2.2 Full Reducers in Planning

The definition of acyclic and cyclic hypergraphs can easily be extended to the scope of action schema instantiations as conjunctive queries. In other words, we can create a hypergraph for the precondition of each action schema $a[\Delta]$, where the nodes are the free variables Δ and the hyperedges correspond to the atoms in the preconditions *pre*$(a[\Delta])$. A node corresponding to a free variable x of Δ belongs to the hyperedge corresponding to atom $P(\boldsymbol{X})$ iff x is a free variable occurring in $\boldsymbol{X}$. Then, we can use the same algorithm described above to verify whether this action schema has an *acyclic precondition* or a *cyclic precondition*.

As argued in the previous subsection, it holds that the intermediate relation size only increases and no intermediate relation can have more tuples than the last relation U, the output of the query. In our case, U contains only the instantiations of the action schema that are applicable in the current state. Similarly, the total input size I is polynomial in the size of the largest unnamed relation of the state (in the number of tuples) and k is the number of atoms in the precondition. Consequently, the complexity of computing the full reducer program, $O(kI \log I)$, is polynomial in the number of atoms in the precondition times the size of the state. Furthermore, the time complexity for computing the final sequence of joins is $O(k(I \log I + U \log U))$, which is still polynomial in the size of the state and the number of applicable grounded actions. The baseline join

```
:precondition (and (at ?player ?position)
                   (at ?stone ?from)
                   (clear ?to)
                   (move-dir ?position ?from ?dir)
                   (move-dir ?from ?to ?dir)
                   (is-goal ?to))
```

Figure 4.2: PDDL action schema corresponding to the action of pushing a stone to a goal position in the Sokoban domain of IPC 2011.

program successor generator described before has time complexity $O(I^k)$, as the size of the intermediate relation can become exponentially long in k. In contrast, the successor generator using complete enumeration has worst-case complexity of $O(n^{|O|})$, where n is the arity of the action schema being instantiated and $|O|$ is the number of objects in the domain. Thus, for acyclic preconditions, the full reducer successor generator can be (in theory) computed more efficiently than the previous methods.

In the rest of this work, we call this method *full reducer successor generator*. Although right now it is a simple method to identify instantiations of action schemas which are applicable to a given state, we extend it further in the next chapters.

We illustrate the definition of acyclic and cyclic preconditions using an action schema example from the Sokoban domain. In this domain, a player has to push a set of stones to goal positions.

Example 4.3 ("Push to goal" action in Sokoban)**.** Figure 4.2 shows the conjunctive precondition of the action schema where a *?player* at *?position* facing a *?stone* from direction *?dir* pushes it *?from* some position *?to* some goal position. Figure 4.3 shows the respective hypergraph representing this action schema. We do not represent relations with one parameter because they can be filtered out by joining them with other relations that share the same parameter without increasing the size of the non-unary relation (e.g., join *is-goal(?to)* and *clear(?to)* to *move-dir(?from, ?to, ?dir)* works as a filter for the latter).

In the following, we use aliases for the relations. These also work as copies of the original relations into auxiliary variables, so we can modify them in the full reducer program without overwriting the initial data. The aliases used are:

$$
\begin{aligned}
R_1 &:= \textit{at(?player, ?position)}\\
R_2 &:= \textit{at(?stone, ?from)}\\
R_3 &:= \textit{move-dir(?position, ?from, ?dir)}\\
R_4 &:= \textit{move-dir(?from, ?to, ?dir)}.
\end{aligned}
$$

It is possible to see that the hypergraph from Figure 4.3 is acyclic. One possible sequence of ear removals is the following: remove R_1 in favor of R_3, remove R_2 in favor of R_4, remove R_3 in favor of R_4. At the end of this sequence, only the hyperedge R_4 remains and thus the hypergraph is acyclic. (This falls into the case (1) described earlier.)

To compute the full reducer σ we start from the last ear removal and add the update of the ear to the back of σ and the update to the relation which the ear was removed to its front. This makes the updated relation of the last ear removals to be in the middle of the sequence, while the updated relation of the first ear removals are at the start and at

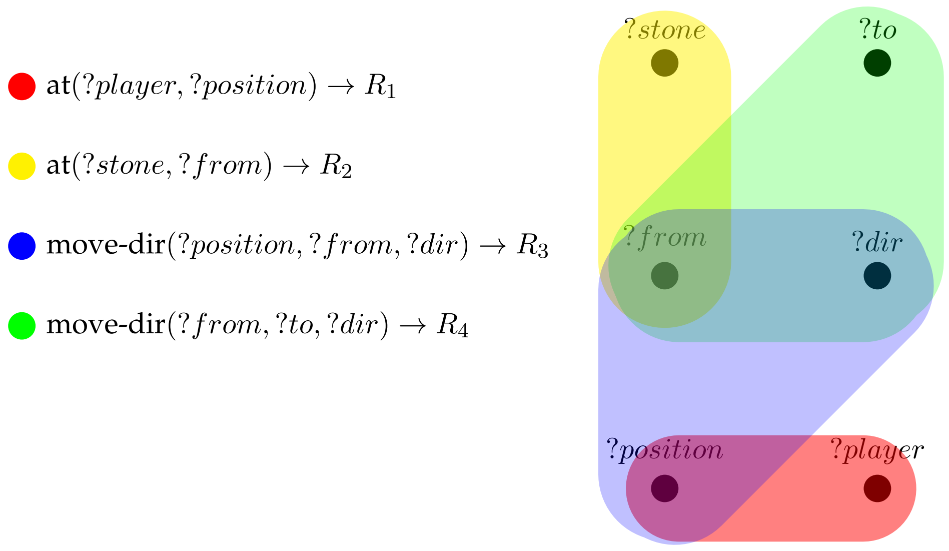

🟥 at(?player, ?position) → R1
🟨 at(?stone, ?from) → R2
🟦 move-dir(?position, ?from, ?dir) → R3
🟩 move-dir(?from, ?to, ?dir) → R4
?stone
?to
?from
?dir
?position
?player

Figure 4.3: Hypergraph representing the conjunctive query checking whether the action schema from Figure 4.2 is applicable.

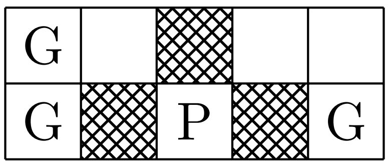


Figure 4.4: Positions of stones (crosshatched tiles), player (P tile), and goal position (G tiles). Each tile is labeled x–y, where x is its column index (starting at 0) and y is its row index (starting at 0). Top-leftmost tile is labeled 0–0, while the bottom-rightmost tile is called 4–1.

the end of σ. The full reducer σ would then be:

$$\begin{aligned}
R_3 &:= R_3 \ltimes R_1\\
R_4 &:= R_4 \ltimes R_2\\
R_4 &:= R_4 \ltimes R_3\\
R_3 &:= R_3 \ltimes R_4\\
R_2 &:= R_2 \ltimes R_4\\
R_1 &:= R_1 \ltimes R_3.
\end{aligned}$$

To finish the computation of the query, we must take the full-join in the reverse order in which we performed the ear removals. Since the ear removal order was R_1, R_2, R_3 and we ended with R_4 still in the graph, the last sequence of joins is

$$R_4 \bowtie R_3 \bowtie R_2 \bowtie R_1.$$

When we join, lets say, R_2 to $R_4 \bowtie R_3$, we cannot decrease the size of the relation being computed, because every tuple in the relation R_2 joins with at least one relation of $R_4 \bowtie R_3$, since R_2, *at(?stone, ?from)*, was an ear removed in favor of R_4, *move-dir(?from, ?to, ?dir)*.

Now, we show how this method works for a concrete state. We use the state represented by Figure 4.4. We call this state as s in the next paragraphs. The representation of

at	
p	2–1
s1	2–0
s2	1–1
s3	3–1

(a) Relation *at*.

clear
0–0
1–0
3–0
4–0
0–1
4–1

(b) Relation *clear*.

is-goal
0–0
0–1
4–1

(c) Relation *is-goal*.

Figure 4.5: Unnamed relations of state s from Figure 4.4. Showing only those occurring in the preconditions of the action schema of Figure 4.2. The object *p* indicates the player, and *s1, s2, s3* indicate the stones. We omit the unnamed relation *move-dir* because of its excessive length, but since it is a static predicate, its initial values should be clear.

s as a database is shown in Figure 4.5. The direction in which a move can be made are denoted as *N, S, E, W* corresponding to north, south, east, and west, respectively. Note the detail about different attribute names. Relations *at(?player, ?position)* and *at(?stone, ?from)* initially are identical to relation corresponding to the predicate symbol *at* in s. However, as the sequence of operations proceeds, they start to differ due to its different naming functions. The same can be said about relations with respect to the predicate symbol *move-dir*.

We start by processing the unary relations. If two unary relations have a same free variable X, we join them into a new relation. Then, we join this new relation to all relations with higher arity also containing the free variable X. This removes all tuples in which X does not satisfy the unary predicates. This step serves as an early filter to the relation with higher arity. In our example, we join *is-goal(?to)* to *clear(?to)*. The resulting relation is

is-goal(?to) ⋈ *clear(?to)*
?to
0–0
0–1
4–1

Then we join this relation to all high-arity relation containing the free variable *?to*. In this case, only *move-dir(?from, ?to, ?dir)*. The filtered relation is

$$\textit{move-dir(?from, ?to, ?dir)} := \textit{move-dir(?from, ?to, ?dir)} \bowtie (\textit{is-goal(?to)} \bowtie \textit{clear(?to)})$$

From now on, we restart using the aliases previously defined. Hence, *move-dir(?from, ?to, ?dir)* is called R_4 once again. The tuples of R_4 after the operation above are

R_4		
?from	*?to*	*?dir*
0–1	0–0	N
1–0	0–0	W
0–0	0–1	S
1–1	0–1	W
3–1	4–1	E
4–0	4–1	S

Once the unary relations are preprocessed, we apply the full reducer semi-join program. The first semi-join in σ is $R_3 := R_3 \ltimes R_1$:

$R_3 := R_3 \ltimes R_1$		
?position	*?from*	*?dir*
2–1	1–0	N
2–1	1–1	W
2–1	3–1	E

And the second semi-join $R_4 := R_4 \ltimes R_2$ in σ:

$R_4 := R_4 \ltimes R_2$		
?from	*?to*	*?dir*
1–1	0–1	W
3–1	4–1	E

The third is $R_4 := R_4 \ltimes R_3$, which does not filter any tuple out of R_4 and hence is not showed. The fourth semi-join in σ is $R_3 := R_3 \ltimes R_4$:

$R_3 := R_3 \ltimes R_4$		
?position	*?from*	*?dir*
2–1	1–1	W
2–1	3–1	E

The two last joins of the full reducer are $R_2 := R_2 \ltimes R_4$ and $R_1 := R_1 \ltimes R_3$, respectively.

$R_2 := R_2 \ltimes R_4$	
?stone	*?from*
s2	1–1
s3	3–1

$R_1 := R_1 \ltimes R_3$	
?player	*?position*
p	2–1

Finally, we can execute the final sequence of joins $R_4 \bowtie R_3 \bowtie R_2 \bowtie R_1$. The final relation is the following:

Benchmark	Action Schemas	Acyclic Precond	Avg. Proportion
IPC 1998-2018	59520	56668 (95.8%)	83.4%
IPC 2018	18437	16005 (86.9%)	72.8%
IPC 2018 w/o Org. Synthesis	113	99 (87.7%)	78.1%
Org. Synthesis – Original	760	65 (8.6%)	8.6%
Org. Synthesis – Split	17564	15841 (90.2%)	90.2%

Table 4.1: Total number of action schemas, total number of acyclic preconditions using different benchmarks, and average of the proportion of acyclic schemas per domain. Number inside paranthesis indicates the ratio of acylic preconditions to the total number of action schemas. We restrict to benchmarks for the optimal track (in editions where there was a division of tracks). We also do not consider action schemas which are already grounded in the domain file (e.g., action schemas for the Trucks domain).

$R_4 \bowtie R_3 \bowtie R_2 \bowtie R_1$					
?player	*?stone*	*?position*	*?from*	*?to*	*?dir*
p	s2	2–1	1–1	0–1	W
p	s3	2–1	3–1	4–1	E

which contains the only two tuples representing valid moves pushing stones to a goal position in the current state. The first tuple is the instantiation where the player pushes the stone at the left and the second tuple instantiates the action where the player pushes the stone at his right side.

$\triangle$

The full reducer successor generator deals with the case of action schemas where the hypergraph of its precondition is acyclic. However, it gives no clue on how to deal with the case where preconditions are cyclic. Unfortunately, this case might lead to intermediate relations that are exponential in the size of the output and no efficient algorithm is known.

In Section 4.3, we introduce methods to deal with cyclic preconditions. These methods try to apply strategies that minimize the size of intermediate relations when computing the join program.

4.2.3 Case Study: Acyclicity of IPC domains

The full reducer successor generator is extremely useful when we have acyclic action schemas. But are the preconditions of planning domains usually acyclic? If most of the domains have action schemas with cyclic preconditions, the full reducer successor generator might be useless.

We ran an experiment using the IPC domains to evaluate the proportion of cyclic and acyclic preconditions over all action schemas. Table 4.1 shows the number of acyclic preconditions in different benchmarks from the IPCs from 1998 to 2018. We considered the action schemas of all STRIPS domains used for the optimal tracks, excluding those which are already grounded in the domain description. Only 4.2% of the action schemas have cyclic preconditions when we consider all action schemas of all domains. In total, we tested 65 different domains and 22 of them have at least one action schema

Benchmark	Action Schemas	Acyclic Precond	Avg. Proportion
IPC 1998-2018	59520	59367 (99.7%)	86.7%
Org. Synthesis – Original	760	695 (91.5%)	91.5%
Org. Synthesis – Split	17654	17654 (100.0%)	100.0%

Table 4.2: Total number of action schemas, total number of acyclic preconditions using different benchmarks, and average of the proportion of acyclic schemas per domain. Number inside paranthesis indicates the ratio of acylic preconditions to the total number of action schemas.

with cyclic preconditions. One domain is responsible for a large chunk of such cyclic schemas. As one might guess, this domain is the hard-to-ground Organic Synthesis. Once we exclude both variants of this domain (original and split) from the benchmark, the number of actions with cyclic preconditions over all IPC domains is very small, only 113 of the 41.196 schemas.

We also verified the proportion of acyclic schemas per domain. The domains where the proportion of acyclic action schemas is less than 50% are: Caldera, Data Network, Elevators, Organic Synthesis, Pipesworld (Tankage and Notankage), Termes, TidyBot, and Tetris. Of these domains, only Organic Synthesis and TidyBot have more than 10 action schemas. The most extreme case occurs in the Pipesworld Tankage (with the no-split domain), where all action schemas are cyclic. The average proportion of acyclic action schemas over all domains is 83.4%. This shows that although there is a significant number of cyclic schemas, most of the instantiations can still be computed in tractable time.

When we focus on the recent domains introduced in the IPC 2018, the proportion of cyclic preconditions increases. Caldera (both versions), Data Network, Settlers, Spiders, and Termes have action schemas with cyclic preconditions. The only two domains tested from IPC 2018 where all action schemas have acyclic preconditions are Nurikabe and Spider. It is important to remember that domains containing only grounded action were not considered, which is the case of the Petri Nets Alignment domain introduced in the IPC 2018.

Still, the Organic Synthesis domain raises a new interesting question. Focusing only on the two variants of this domain, the original one and the one using action schema splitting [Areces *et al.*, 2014], we can see that the proportion of cyclic preconditions drastically changes when we use the splitting technique. In the original version, 695 of the 760 action schemas have cyclic preconditions, a total of 91.4%. In the split version, 1.723 of the 17.564 preconditions are cyclic, only 9.8%. A similar behavior can be observed in Caldera: the only cyclic precondition in the original Caldera formulation becomes acyclic once the action schema splitting is performed. However, in this split version of the Caldera domain, the splitting also introduces new cycles to preconditions that were initially acyclic. The goal of the action schema splitting by Areces *et al.* is to reduce the interface (i.e., number of parameters) of the action schemas. It might be the case that this method also performs an underlying decomposition of the query in some special cases. In database theory terms, a query decomposition is a reformulation of a query into a new but equivalent query such that the latter presents a hypergraph topology which is easier to evaluate, perhaps even becoming (fixed-parameter) tractable [Gottlob *et al.*, 2000, 2002].

Most of the cyclic preconditions in IPC domains occur due to inequality constraints.

(Here we only consider $\neq$ and not $<$.) In the results reported above, we treat the inequality predicate symbol like any other regular predicate symbol. However, inequality constraints are native predicates in PDDL with a built-in semantics and indicate that two parameters must be instantiated by different objects. These semantics are identical to the built-in inequality predicate in relational algebra, for example. To illustrate, assume that we have the atom *!=(?a, ?b)*. This atom means that the objects substituting *?a* and *?b* must be different. We can consider the notion of *acyclic queries with inequalities* [Papadimitriou and Yannakakis, 1999], where the hypergraph does not consider built-in inequality constraints. Unfortunately, even though our hypergraphs get simpler when ignoring inequalities, the problem becomes NP-complete. In the literature, it is still possible to find fixed-parameter tractable algorithms with respect to the query size for acyclic queries with inequality constraints (e.g., Papadimitriou and Yannakakis [1999]).

In the planning scenario, we can also make use of this natural semantics of inequality predicates. However, since the relations usually do not have more than hundreds of tuples, we can solve the inequalities issue with a simple post-processing stage. In this post-processing, we interleave the join program with inequality checks. When computing a join program, we can identify when relations containing free variables in a single inequality constraint are joined. In this case, we loop over resulting tuples and discard those violating the inequality constraints. Although this method is not tractable in any fixed parameter, our experiments demonstrate that it is still efficient enough for the planning problems tested (Chapter 5).

Using this extra stage, we can construct our hypergraph and analyze the acyclicity of a query based only on the other relations which are not inequality constraints. By adding this post-processing step, we also lose the guarantee that no intermediate relation is larger than the final output. However, our experiments also show that this sacrifice is a good trade-off. Table 4.2 shows the number of action schemas that are acyclic when not considering inequality constraints on the hypergraph. The proportion of cyclic action schemas over all action schemas decreases significantly. In particular, the Organic Synthesis domain presents a reduction of more than 90% in the number of cyclic action schemas. The number of domains with cyclic actions also falls from 22 to 20. Other domains also had a significant reduction in the number of cyclic schemas using this method: GED, Hiking, and TidyBot. The average proportion of acyclic schemas overall domains increases from 83.4% to 86.7% when ignoring inequality constraints.

Furthermore, our experiments shows that such post-processing does not harm the full reducer successor generator in any case, since most of the relations are relatively small and this stage is computed very quickly.

4.3 Extensions

In this section, we present two extensions to the full reducer successor generator. The main objective of these two extensions is to enhance the performance of the generator in a few special (but common) cases.

The first extension deals with the case where we have free variables in the preconditions of an action schema that are not affected by any effect. In formal terms, this is the case where

$$\Delta \neq \mathit{free}(\mathit{add}(a[\Delta])) \cup \mathit{free}(\mathit{del}(a[\Delta])).$$

In this case, variables that are not in the add and delete lists could be existentially quantified in the precondition. However, many planners do not allow existentially quantified

```
(:action dummy
 :parameters (?a ?b ?c)
 :precondition (and (S ?a)
                    (S ?b)
                    (S ?c))
 :effect (and (not (S ?a)))
)
```

Figure 4.7: Action schema with three parameters, *?a, ?b, ?c*, and only one occurs in the effect, *?a*

variables in the preconditions. We show how the full reducer successor generator always detects such cases automatically and reduces in the number of applicable actions in a state, without losing correctness or completeness.

The second extension tries to minimize the potential harm of the full reducer successor generator in cases where the action schema is acyclic. Our approaches to such cases have no guarantee of any optimality and most of them are similar to the methods that also enhance the naive successor generator.

4.3.1 Extension 1: Free Variables only in Preconditions

We motivate the use of this extension with an example. Although our example is hand-tailored, we show later that several IPC domains have action schemas with free variables only in the preconditions.

Example 4.4. In the example of Figure 4.7, we have an action schema with three parameters (*?a, ?b, ?c*) but only one occurs in the free variables of the effects.

Let s be a state of the planning task and assume that there are n ground atoms of the type *(S x)* in s. Thus, the number of applicable instantiations of this schema in s is n^3. However, there are only n different successor states of s: the n instantiations of parameter *?a*, since it is the only one which is changed by some effect. All possible n^2 combinations of instantiations of *?b* and *?c* for a fixed *?a* lead to a same state, since *?b* and *?c* are not affected. It means that s would have a branching factor (i.e., number of outgoing transitions in the state space) of n^3 but leading to only n different successors.

Our example is relatively small to maintain its simplicity. Note that as the number of parameters in the action schema increases, the difference between the branching factor and the number of different successors would also increase. △

This situation actually occurs in several IPC domains. For example, the Organic Synthesis domain has action schemas with 17 parameters where only four appear in some atom in the effect lists. But not only the Organic Synthesis domain presents such complicated situations. Domains such as Agricola, TidyBot, Woodworking, Data Networks, and Satellite have several action schemas where at least four free variables in the parameters are not affected by the add and delete lists. We tested 76 STRIPS domains used in all IPCs and in exactly half of them, 38, there was at least one action schema with free variables in its preconditions which were not in the effects. This test also shows that if a successor generator is able to avoid such unnecessary instantiations, it could have a great improvement in performance in general. (In Chapter 5 we give more details about this experiment.)

Variables that do not show up in the effect can be *existentially quantified*. We do not care about the values of existentially quantified variables, we simply care that a valid instantiation for them exists. In fact, Francès and Geffner [2016] demonstrate that existentially quantified variables arise naturally when modeling different planning and domains and treating them differently can improve the performance of the planners.

We can adapt our full reducer successor generator to deal with this case. This extension can detect variables that should be existentially quantified in the precondition and project them out from the query answer. Once again, we explain it with the example shown in Figure 4.7. The query for this action schema is

$$Q(A, B, C) \coloneq S(A), S(B), S(C).$$

However, since the effect simply affects the objects instantiated by the attribute A, we simply do not care about the instantiations of B and C. What we really care about is whether there is at least one tuple instantiating B and C for each possible value of A. In other words, B and C can be existentially quantified in the precondition. Thus, we can simply rewrite the query above as

$$Q(A) \coloneq S(A), S(B), S(C).$$

This scenario is nothing more than a projection in the resulting relation of the query. This is a so called *project-join* program [Ullman, 1989]. A project-join program is a conjunctive query where only some free variables are of interest. For example, in the query

$$Q(\boldsymbol{X}) \coloneq R_1(\boldsymbol{Y}_1), R_2(\boldsymbol{Y}_2), \ldots, R_n(\boldsymbol{Y}_n)$$

where $\boldsymbol{X} \subset \boldsymbol{Y}_1 \cup \ldots \cup \boldsymbol{Y}_n$, we only need to keep record of the free variables $\boldsymbol{X}$, because these are the values which we want to retrieve in the query answer. These are the so called *distinguished variables* of the query, the ones occurring in the heads. In relational algebra, this means that we are computing the conjunctive query using a join program but projecting the tuples only into the distinguished variables.

A subtle modification in the full reducer successor generator is able to automatically deal with project-join programs. Instead of applying a sequence of join operations after executing the semi-join operations of the full reducer, we interleave the join operations with the necessary projections. This algorithm is called Yannakakis' Algorithm for project-join expressions [Yannakakis, 1981].

The algorithm for computing the project-join of a query $Q(\boldsymbol{X})$ works as follows:

1. Compute and execute the full reducer of $Q(\boldsymbol{X})$.

2. Compute the parse tree P of the original query, where every relation in the body is a node and a relation $R(\boldsymbol{Y})$ is a child of $S(\boldsymbol{Z})$ iff $R(\boldsymbol{Y})$ was an ear eliminated in favor of $S(\boldsymbol{Z})$.

3. Traverse P in a bottom-up order. Whenever we visit a relation $R(\boldsymbol{Y})$ such that $S(\boldsymbol{Z})$ is its parent, we replace the latter by the relation

$$S'(\boldsymbol{W}) \coloneqq \pi_{\boldsymbol{W}}(R(\boldsymbol{Y}) \bowtie S(\boldsymbol{Z})),$$

   where $\boldsymbol{W} = \boldsymbol{Z} \cup (\boldsymbol{Y} \cap \boldsymbol{X})$.

4. Once the tree traversal reaches the root node of P, apply a projection on the distinguished attributes $\boldsymbol{X}$ in the relation of the root node.

The crucial step in Yannakakis' algorithm is the third step. Once a relation $S(\boldsymbol{Z})$ is joined with all its children, its result is equivalent to the complete join of $S(\boldsymbol{Z})$ and all its children, projected only on the attributes $\boldsymbol{Z}$ and whatever distinguished attributes appear in its descendants of the parse tree.

This project-join algorithm is quadratic in the sum of the input size I and the output size U [Yannakakis, 1981; Ullman, 1989]. However, the output size in this case is slightly different. The output size of this algorithm is the number of different tuples only considering the distinguished attributes, instead of all attributes. Hence, in queries such as the one for Example 4.4, the output size is significantly smaller when considering only the different tuples over the distinguished attributes. Note that when applying Yannakakis' algorithm to generate successors, we are not only checking the applicable instantiations for a given action schema but also projecting out the instantiations which generate duplicated successors. This is the reason why we consider our methods as successor generators, instead of simply calling them "generators of applicable actions". Also, when using Yannakakis' algorithm, we lose the guarantee that no intermediate relation has size bounded by the output size. Still, we can guarantee that no intermediate relation will be larger than $2IU$ [Ullman, 1989].

Example 4.5 (Ullman [1989]). We start with five relations $R_1(C, D, E)$, $R_2(B, C, D)$, $R_3(D, E, G)$, $R_4(A, B, C)$, and $R_5(B, F)$. The query considered is

$$Q(A, G) = R_1(C, D, E) \bowtie R_2(B, C, D) \bowtie R_3(D, E, G) \bowtie R_4(A, B, C) \bowtie R_5(B, F).$$

Assume that the full reducer program was already computed and evaluated and thus all the relations are already fully reduced. The parse tree of the full reducer of $Q(A, G)$ is

$$\begin{tikzcd}
 & & R_1(C,D,E) \arrow[dl, dash] \arrow[dr, dash] & \\
 & R_2(B,C,D) \arrow[dl, dash] \arrow[dr, dash] & & R_3(D,E,G) \\
R_4(A,B,C) & & R_5(B,F) &
\end{tikzcd}$$

In our example, let the fully reduced relations be the following

$R_1(C,D,E)$		
C	D	E
c_1	d_1	e_1
c_1	d_2	e_1

$R_2(B,C,D)$		
B	C	D
b_1	c_1	d_1
b_1	c_1	d_2

$R_3(D,E,G)$		
D	E	G
d_1	e_1	g_1
d_1	e_1	g_2
d_2	e_1	g_1

$R_4(A,B,C)$		
A	B	C
a_1	b_1	c_1
a_2	b_1	c_1

$R_5(B,F)$	
B	F
b_1	f_1
b_2	f_1

The algorithm starts visiting the nodes of the parse tree bottom-up. Let us start by the node representing relation $R_4(A, B, C)$. We join it with its parent $R_2(B, C, D)$. The resulting relation has attributes B, C, D, the attributes of the parent node, plus attribute A, since it is the only distinguished attribute in the child. We denote this new relation as $S_{2,4}(A, B, C, D)$ and replace $R_2(B, C, D)$ with it. Next, we join the recently replaced relation $S_{2,4}(A, B, C, D)$ with the other child $R_5(B, F)$. Since F is not distinguished and

B is already in the parent node, the resulting relation still contains just A, B, C, and D. This new relation is denoted $S_{2,4,5}(A, B, C, D)$ and replaces $S_{2,4}(A, B, C, D)$ in the tree. The tuples in $S_{2,4,5}(A, B, C, D)$ are

$S_{2,4,5}(A, B, C, D)$			
A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_1	c_1	d_2
a_2	b_1	c_1	d_1
a_2	b_1	c_1	d_2

Now, the third layer of the parse tree is already visited and we move forward to the second layer. Let $S_{2,4,5}(A, B, C, D)$ be the first node visited in this layer. This node is joined with its parent node $R_1(C, D, E)$. We project into the new relation only the attributes of the parent, C, D, E, plus the distinguished attribute of $S_{2,4,5}(A, B, C, D)$, which in this case is only A. The new relation is $S_{1,2,4,5}(A, C, D, E)$ replaces the root node. Now, the only relation still not joined to the root is $R_3(D, E, G)$. We join it with $S_{1,2,4,5}(A, C, D, E)$, resulting in the relation $S_{1,2,3,4,5}(A, C, D, E, G)$. Attribute G is kept in the new relation because it is distinguished. Now all the relations in the second layer are already joined to their parents. We proceed to the root node next. Once we visit the root node, we simply project its relation $S_{1,2,3,4,5}(A, C, D, E, G)$ to the distinguished attributes. This leads to the relation that is the output of the algorithm, $Q(A, G)$. We show $S_{1,2,4,5}(A, C, D, E)$, $S_{1,2,3,4,5}(A, C, D, E, G)$, and the output relation $Q(A, G)$ below. Note that $Q(A, G)$ is just a projection of $S_{1,2,3,4,5}(A, C, D, E, G)$ over the distinguished attributes, but it contains only 4 tuples while $S_{1,2,3,4,5}(A, C, D, E, G)$ has 6.

$S_{1,2,4,5}(A, C, D, E)$			
A	C	D	E
a_1	c_1	d_1	e_1
a_1	c_1	d_2	e_1
a_2	c_1	d_1	e_1
a_2	c_1	d_2	e_1

$S_{1,2,3,4,5}(A, C, D, E, G)$				
A	C	D	E	G
a_1	c_1	d_1	e_1	g_1
a_1	c_1	d_1	e_1	g_2
a_1	c_1	d_2	e_1	g_1
a_2	c_1	d_1	e_1	g_1
a_2	c_1	d_1	e_1	g_2
a_2	c_1	d_2	e_1	g_1

$Q(A, G)$	
A	G
a_1	g_1
a_1	g_2
a_2	g_1
a_2	g_2

$\triangle$

Modifications

There are three modifications that we need to perform to use the Yannakakis' algorithm in the case of planning.

(i) A planner must describe the plan as a sequence of ground actions, where the objects match exactly the respective parameters of the action schemas. Thus, in order to have a ground action, we need to keep track of at least one object for each non-distinguished variable in each tuple of the query. For example, if we project a relation R into the distinguished variables $\boldsymbol{X}$ and this projects out the attribute A, we must still keep track of at least one matching value of attribute A for each remaining tuple of $\pi_{\boldsymbol{X}}(R)$. In practice, for each combination of objects of the projected attributes, we keep a complete tuple containing all attributes, including the ones being projected out.

(ii) The inequalities are also not part of the original algorithm. Although there are algorithms for such purpose in the literature [Papadimitriou and Yannakakis, 1999], we stick to the simple post-processing of inequalities. It is important, though, that the projection is performed after the computation of the inequalities.

When we compute the projection, we still keep at least one value for the attributes projected out. Suppose that, we keep only a tuple violating an inequality constraint, while there were other tuples that do not violate this inequality. If this is the case, the post-processing of inequalities will remove this tuple later. But this removal can make an action inapplicable and hence the task unsolvable.

When the inequalities are processed prior to the projection, we ensure that the remaining tuples do not violate any constraint and will not be removed unsafely.

(iii) Because of our inequality post-processing, it might occur that two distinguished variables are in a single inequality constraint, so we need to process them in every join of the tree. This post-processing forbids us to assume the original upper-bound of $2UI$ in the intermediate relations.

4.3.2 Extension 2: Efficiently Instantiating Cyclic Preconditions

In this section, we discuss strategies for the case of action schemas with cyclic preconditions. The main goal here is to find a strategy that minimizes the chance of producing intermediate relations with size exponential in the size of the join program. In practice, these strategies define an order to perform the join program over all relations in the precondition.

In this work, we are interested in what we call *static strategies*. Such strategies order the join program based on the structure of the query and not based on the tuples of the relations. In contrast to static strategies, there are dynamic strategies, which also consider the characteristics of the relations when ordering the join program. For example, we could order our join program based on the number of tuples in each relation. Although dynamic strategies can be very good for cyclic queries, they present the drawback that they must be recomputed at every single state of our search. Trying to avoid this possible overhead, we limit ourselves to static strategies in this work. However, we emphasize that dynamic strategies might have a performance at least as good as our static strategies.

Naive Join

The first and most straightforward strategy is to simply perform a complete join program of the predicates in the precondition in any arbitrary order. For example, the order in which the predicates are given or randomized order.

Ordered Naive Join

A simple extension of the Naive Join is to order the join program by the arity of its predicates. For example, one could start by joining the relations with the lowest arity first. Intuitively, by joining relations of lower arity first, we might keep our intermediate relations more compact with respect to its arity and cardinality. It can also be useful to also consider free variables in each predicate when ordering or breaking ties since this would avoid the case where the relations do not share any free variable and hence a Cartesian product is performed.

Semi-Join Program + Secondary Strategy

Another option is to use a semi-join program first and then perform a join program. For example, we could use the fact that we always precompute a semi-join program when testing whether a precondition is acyclic or not and reuse this semi-join program to filter out our relations before the full join. In this scenario, we try to compute a full reducer but we fail (i.e., at some point there is no possible ear removal in the hypergraph but there are still hyperedges left). However, previous ear removals are meaningful because they semi-join relations that have a common free variable and thus filter out tuples from these relations. Hence, we can keep this semi-join program and apply it before computing the join of all relations. To order this last complete join, we can rely on one of the other strategies mentioned in this section. This semi-join and join programs can be preprocessed for each action schema and the strategies reused in each state.

When discussing the experimental results of our methods, in Chapter 5, we show that the instantiation of cyclic preconditions is not the bottleneck of our approach and thus the methods introduced here are sufficient for a good performance.

• • •

In the next chapter, we present the experimental results comparing our different methods introduced in this chapter. We also introduce some results comparing different ways to represent states. We will also compare our lifted planning methods to well-established grounded planners.

Chapter 5

Experiments

In this section, we describe the experimental results of our lifted successor generators. We first show the results comparing different ways to represent states in a lifted planner. Then, we compare the different successor generators using the lifted action representation introduced before. We conclude the chapter presenting a comparison of our best lifted successor generator to state-of-the-art grounded planners.

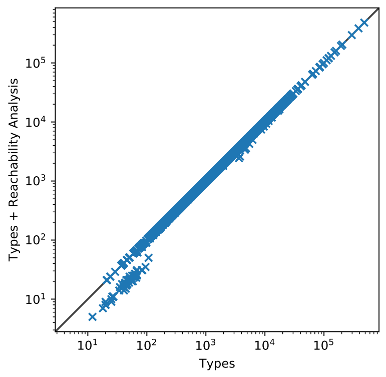
We implemented a planner prototype using the lifted successor generators and the extensions previously described. The source code of the planner is available online[1]. The planner has two main components: (i) a translator component, based on the translator from Fast Downward [Helmert, 2006, 2009]; and (ii) a search component implemented from scratch. Our planner first translates a task from PDDL to an intermediate representation, compiling types into unary predicates and filtering out trivially inapplicable action schemas (e.g., if an action schema needs an object of type T but there is no object with such type in the task.) The search component performs a heuristic search over this translated task. We implemented a breadth-first search and a GBFS in the search component. The search component is written in C++ 17 and it is integrated with Lab [Seipp *et al.*, 2017] for experiments.

Our planner supports an extension of STRIPS allowing equalities and inequalities in preconditions. The main benchmark used consists of all domains using such formalism from all optimal tracks of the IPCs. As previously mentioned, we could also support negative preconditions simply by transforming the domains into positive normal form. In total, this benchmark has 1560 instances over 53 different domains. In some specific experiments, we also used larger benchmarks, or benchmarks from the satisficing track of the IPCs or even domains which were never part of the IPC. These larger benchmarks are described in detail in their specific experiments. All experiments were run on an Intel Xeon Silver 4114 processor running at 2.2 GHz. Each task was allowed total time of 30 minutes and a total memory usage of 16 GB.

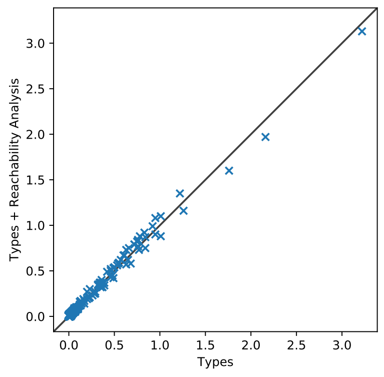
5.1 State Representation

Our first question in the empirical part is how to represent a state. We consider two types of representations: *complete* and *sparse*. In the complete representation, a state is simply a sequence of bits, each bit associated with a ground atom. However, since we do not perform any grounding, we must find an efficient way to obtain an overapproximation of the reachable ground atoms of the planning task. A possible way is to enumerate a

[1]`https://zenodo.org/record/3539283#.XcrXsNF7kUE`



(a) Number of ground atoms



(b) Time in seconds

Figure 5.1: Comparison of state size in the number of ground atoms and processing time for the method enumerating all possible ground atoms only based on type restrictions and the one based on types and also on the soft reachability analysis.

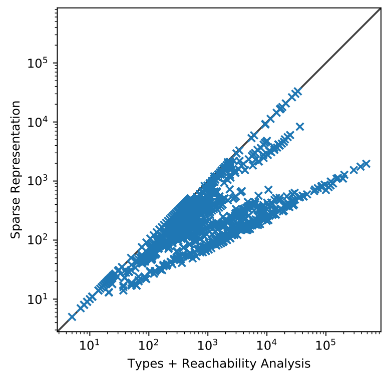
sufficiently large list of ground atoms when translating the task. Unfortunately, it is not obvious how to perform an efficient enumeration in terms of the computation time and the number of ground atoms produced.

Our first experiment attempts to find the number of ground atoms generated in each of the tasks of our benchmark using a complete state representation. A simple way to use complete states without grounding is to enumerate all possible ground atoms. In PDDL tasks where type information is also given, we can narrow down this enumeration to only consider ground atoms that satisfy the type requirements. Unfortunately, this leads to large states in many domains. For example, in the Logistics98 domain, many tasks need more than 20 kilobytes per state. The task with the most number of objects in this domain needs approximately 40 kilobytes per state.

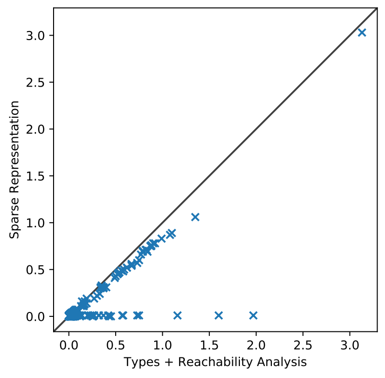
To avoid the excessive creation of ground atoms, we implemented a *soft reachability analysis*. Our analysis is an overapproximation of the one performed by Helmert [2009]. In a brief description, our analysis is similar to a breadth-first search over a reachability graph. This reachability graph $G = (V, E)$ is created as follows: for every n-ary predicate $P(t_1, \dots, t_n)$ of the task, we create n nodes $\langle P, i\rangle$ for $1 \leq 1 \leq n$. Let $a[\Delta]$ be an action schema where predicate $P(t_1, \dots, t_n)$ appears in the precondition and a predicate $Q(s_1, \dots, s_m)$ appears in the effect. If $t_i = s_j$ for $1 \leq i \leq n$, $1 \leq j \leq m$, then we add an edge from $\langle P, i\rangle$ to $\langle Q, j\rangle$ in E. Note that it might happen that the predicate symbols P and Q are the same and that t_i might be a constant or a free variable of Δ. Similarly, if s_j occurs only in the effect, then we add an edge from all other nodes to $\langle Q, j\rangle$. We define E as the set that contains all edges added following this description.

Once G is created, we verify which nodes are reachable from every other node. We use a strong notion of reachability, where a node $u \in G$ is reachable from node $v \in G$ if there is a path from v to u. If node $\langle Q, j\rangle \in G$ can be reached from $\langle P, i\rangle \in G$ then we assume that every object instantiating the i-th argument of a ground atom with predicate symbol P can instantiate a predicate with symbol Q as its j-th argument. We perform this analysis based on the ground atoms given in the initial state s_0.

This can be seen as an overapproximation of a relaxed reachability analysis performed by Helmert [2009] where we split each Datalog rule with a body of length n into n new rules with atomic bodies and remove those in which the free variables of the rule



(a) Ground atoms



(b) Time in seconds

Figure 5.2: Comparison of initial state size in the number of ground atoms and processing time for the method enumerating all possible ground atoms only based on types and the soft reachability analysis and the method generating a sparse representation of the initial state.

head are not present in the body.

Figure 5.1a compares the size of the state when enumerating the ground atoms based only on the restrictions on types and when also performing the reachability analysis. We can see a reduction in size in exactly 200 instances. The time spent in the translation component when using each method is shown in Figure 5.1b. We can observe that the reachability analysis is a good trade-off: the computation of the reachability graph and its analysis is not expensive and in most cases, it is still cheaper than to enumerate all possible ground atoms.

However, Figure 5.1a also demonstrates that even with the soft reachability analysis, some states are still too large to be represented. The problem with our soft reachability analysis is that when a free variable or constant appears only in the effect, then the only safe assumption we can perform is that any object might reach it. This is too weak for efficient reachability analysis. Hence, even though our soft reachability analysis improves the state size representation, it is still not practical.

In contrast to the complete representation, a sparse state representation simply represents a state as a set of relations, where each relation is a set of tuples, as in the database theory interpretation. In this case, the size of a state using this sparse representation is not constant during the whole search and, hence, some implementation optimizations are not possible. Yet, the sparse state representation has the advantage that the number of ground atoms in any state is bounded by the size of reachable atoms. This guarantees that the number of ground atoms considered in the sparse case is never larger than in the complete case. However, since we do not know the ground atoms in advance, we cannot use a single bit per ground atom in the sparse case.

We compared the method using type constraints and the soft reachability analysis with the scenario of using a sparse state representation. In this experiment, we compare the state sizes in the number of atoms and consider only the initial state of each task (since the sparse representation has variable size). Figures 5.2a and 5.2b show the state size in the number of ground atoms and the processing time of the translator component for each method over the 1560 tasks of our benchmark. As expected, the sparse representation is smaller in general. Also, since we do not perform any translation procedure exclusively in the state representation in the sparse case, we end up needing less time to translate the task.

Although the sparse representation is smaller in the number of ground atoms used, it might also generate overhead due to more complicated data structures. For example, we could use one bit per ground atom in the complete state representation. On the other hand, the sparse representation needs to represent every ground atom as a pair (i, t) where i is an integer identifying the predicate and t is a tuple of integers representing the objects instantiating it. Assume that in a state s all n ground atoms of the task are true. The complete state representation needs $\Theta(n)$ bits to represent this state, while the sparse representation would need $O(n^a b)$ bits, where a is the maximum arity of any predicate and b is the number of bits used to represent an integer. (Here, we are also not considering extra bits used to represent data structure more compactly, such as extra pointers.) In order to amortize the space consumption, we can use a *packed representation* of the ground atoms. For example, we can keep the list of already expanded states using packed representations of the ground atoms. This packed representation hashes every tuple of each relation into an integer of b bits using a perfect hash function. Then, it also hashes the set of hashed tuples (now represented as integer numbers) of every relation into another integer. Thus, the packed version of a sparse state representation uses $O(br)$ bits, where r is the number of relations in the state[2]. Besides that, the time overhead of the sparse representation is also significant. Let us say that we want to check whether a ground atom is true in a given state s. In the complete case, this ground atom is represented by an index in a bit-mask and we can check it in constant time. In the sparse case, if we use a hash set data structure as a container for the ground atoms, we have the worst-case complexity of $O(n)$ (although the amortized complexity is $O(1)$).

Still, we decided to stick with the sparse representation in our implementation. The main reason for that is because this representation fits our successor generators more intuitively, which is the focus of this work.

5.2 Comparison of Lifted Successor Generators

Our next experiment compares different methods to generate successors. We first tested the following methods:

(i) **Naive Join Program** (J): Join the relations using the predicate order as given by the PDDL definition of the action schema.

(ii) **Randomly Ordered Join Program** (J^R): Join the relations using a randomly ordered join program. Results averaged over three runs.

(iii) **Join Program ordered by arity** ($J^<$): Join the relations ordered by the ascending arity of their predicates.
For this method, we also tested ordering the join program by descending arities, but this led to worse results.

(iv) **Full Reducer** ($FR^{SJ,<}$): For acyclic preconditions, perform the full reducer and then the join program in the order established by the ear removals.
For cyclic preconditions, perform the semi-join program computed until no ears were left in the hypergraph, then computes a join program over all relations ordered by arity.
We also tested some different strategies for cyclic preconditions in the method

[2]In our experiments, we had to use a long representation of integers using $b = 64$.

Domain	# of tasks	J	$J^<$	$FR^{SJ,<}$
depot	22	2	**3**	**3**
freecell	80	**13**	7	**13**
gripper	20	6	**7**	**7**
nomystery-opt11-strips	20	**7**	6	6
organic-synthesis-opt18-strips	20	11	10	**19**
parcprinter-08-strips	30	**9**	8	8
pegsol-opt11-strips	20	**16**	**16**	15
pipesworld-notankage	50	**13**	11	12
pipesworld-tankage	50	**8**	6	6
scanalyzer-08-strips	30	**10**	8	9
scanalyzer-opt11-strips	30	**7**	4	6
sokoban-opt08-strips	30	10	**13**	**13**
sokoban-opt11-strips	20	8	**10**	**10**
woodworking-opt08-strips	30	**7**	6	**7**
woodworking-opt11-strips	20	**2**	0	**2**
zenotravel	20	5	**7**	**7**
Other domains	1274	355	355	355
Total	1560	454	443	**464**

Table 5.1: Number of solved instances per domain using a breadth-first search with the three different successor generator methods based on database techniques. Showing only domains where there was difference in coverage between different methods.

$FR^{SJ,<}$, such as ignoring the semi-join program, joining first the ears removed in the final join program, and also randomly ordering the join program. However, these also led to worse results.

In all methods, inequality constraints are post-processed.

Our first results compare these four different methods using a breadth-first search over all instances in our benchmark.

Coverage: As one might expect, a breadth-first search does not perform well in most of the IPC domains and thus all methods have very low coverage. The randomly ordered naive join, J^R, has the worst coverage, 350.0 on average. The join program ordered by arity, $J^<$, has the second-worst coverage, with 443 instances solved. The join program ordered by the PDDL definition solves 464 tasks. The best method concerning coverage is the full reducer successor generator, $FR^{SJ,<}$. This method achieved a total coverage of 464.

Among the best three methods, very few domains had a difference in coverage. Table 5.1 compares the coverage between J, $J^<$, and $FR^{SJ,<}$ in domains where at least one method obtained different total coverage. Most of them present a difference of at most three instances. The only exception in this case is the most interesting result: the Organic Synthesis domain has a jump from 10 to 19 instances solved when using the successor generator based on the full reducer. Most of the Organic Synthesis instances have considerably short plans and grounding the actions is the critical point to solve the domain. If we sum up the time needed for $FR^{SJ,<}$ to solve these 19 instances, it is below one minute. The instance not solved by $FR^{SJ,<}$ ran out of memory. For the other two methods, a few instances also ran out of memory while trying to instantiate action schemas during the search. However, it is remarkable that our worst method presented in Table 5.1 for this domain, $J^<$, already solves 2 more instances than any planner in the IPC 2018. This demonstrates that even our worst lifted method is already powerful enough to solve the domain and indicates that the Organic Synthesis domain is hard only due to the grounding.

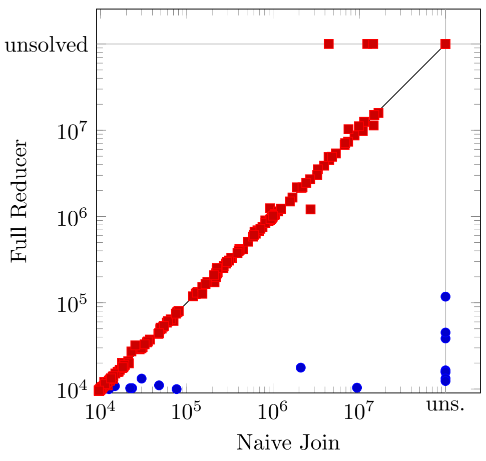
On the other hand, the randomly ordered join program, J^R, was only able to solve two instances of the Organic Synthesis domain. This shows that although our methods based on database techniques can enhance the performance in this domain, it is still necessary to consider some information about the structure of the precondition (i.e., query) when instantiating it.

Comparing J^R and J: The comparison between J^R and J raises an interesting question. Do the IPC domain descriptions contain some underlying model information that helps the successor generators? We claim that yes, this happens in practice. Our hypothesis is that when users are modeling a specific domain, they tend to think about the domain with respect to its objects instead of the predicates. Hence, the user ends up (unconsciously) "clustering" predicates related containing the same free variable when defining the preconditions and effects. In this manner, J uses these "clusters" to avoid joins between relations without a common free variable (which induce Cartesian products over the relations). In contrast, the randomization provided by J^R breaks such clusters and creates more situations where the Cartesian product is necessary, leading to larger intermediate relations that significantly slow down the planner.

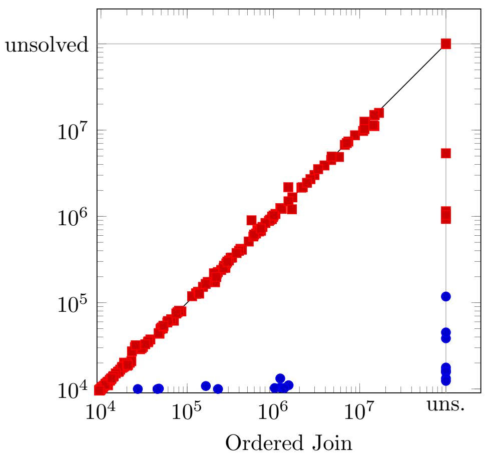
To prove this hypothesis, we measured the hit rate of free variables doing the join program when using J and J^R. When performing a join program $R_1 \bowtie R_2 \bowtie \dots R_n$, we say that we have a hit when a free variable in R_i was already part of some other relation $R_1, R_2, \dots, R_{i-1}$. Similarly, we count as a miss when the free variable was not part of any relation in $R_1, R_2, \dots, R_{i-1}$. The hit rate of an action schema is the proportion of hits compared to the sum of hits and misses. Using J (and ignoring grounded actions), we have an average hit rate of 0.95 in our benchmark. The randomized method J^R has an average hit rate of 0.87. This corroborates with our hypothesis, showing that there might be bias during modeling planning domains that "clusters" objects when describing action schemas. It might be also useful to consider this kind of bias when ordering (or breaking ties) in our join programs.

Memory and Time Consumption: We also compared the peak memory usage and time for all methods. The results for peak memory consumption are presented in the plots of Figure 5.3. We show only the results comparing $FR^{SJ,<}$ to the other two best methods, J and $J^<$. In most of the instances, all methods present similar memory usages. However, $FR^{SJ,<}$ is significantly more efficient in the Organic Synthesis domain. For the instances that J and $J^<$ also solve, $FR^{SJ,<}$ never reaches 20 MB, while the other methods have a peak close to 10 GB, caused exclusively by larger intermediate relations while generating successors. This is not surprising: $FR^{SJ,<}$ avoids extremely large intermediate relations while instantiating actions and ends up needing less memory in general.

When comparing total search time, we observe a similar behavior. Most of the instances present a similar total search time, but the Organic Synthesis instances need less time when using $FR^{SJ,<}$. (In fact, most of the Organic Synthesis instances are solved very quickly and are placed in the left-bottom corner of the plot.) Although one might expect that the $FR^{SJ,<}$ method would be slower due to the additional overhead of com-

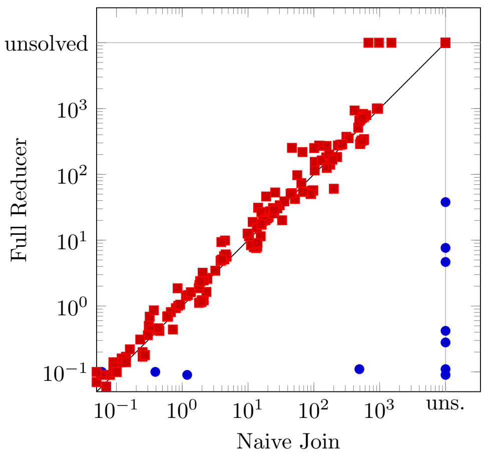


(a) $FR^{SJ,<}$ compared to J

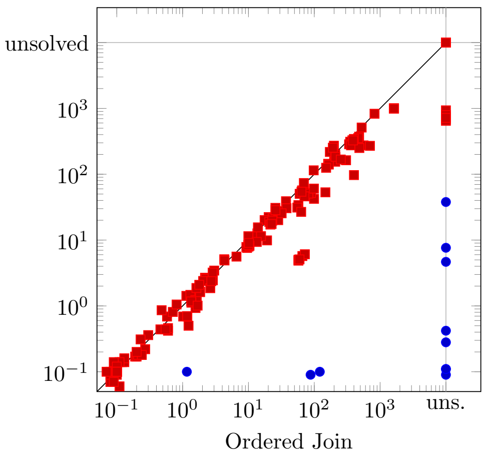


(b) $FR^{SJ,<}$ compared to $J^<$

Figure 5.3: Peak memory usage in KB for different successor generator methods. Showing only instances solved by at least one method. Organic Synthesis instances are marked in blue.



(a) $FR^{SJ,<}$ compared to J

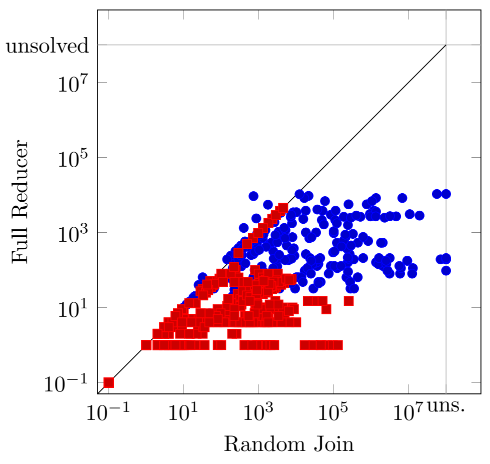


(b) $FR^{SJ,<}$ compared to $J^<$

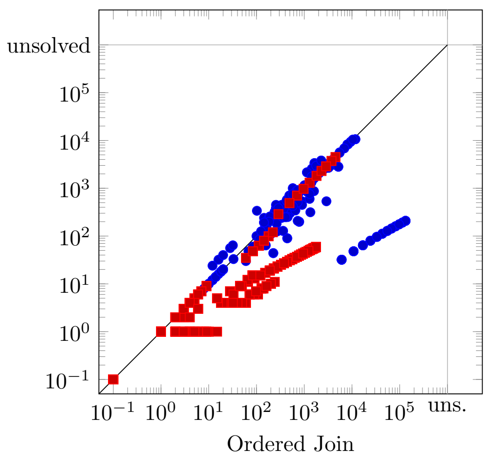
Figure 5.4: Total search time in seconds for different successor generator methods. Showing only instances solved by at least one method. Organic Synthesis instances are marked in blue.

puting a semi-join program prior to the full join program, the method is competitive in all instances. Interestingly, this overhead seems to be a good trade-off, since it leads to smaller intermediate relations and hence quicker join programs.

Largest Intermediate Relation: To better evaluate the differences between the methods, we ran an experiment computing the largest intermediate relation obtained when generating successors for the initial state of every task. In this way, we can compare our methods not only with respect to solved instances. Broadly speaking, a method with lower intermediate relations consumes less memory and generally presents better performance. Although the initial state might not be representative of most of the states in the search, we expect the results to generalize up to some point throughout the search. Figure 5.5 shows the plots comparing $FR^{SJ,<}$ to J^R and $J^<$. The results for J are very similar to the ones for $J^<$, so only the second one is shown. The instances marked in blue circles contain action schemas with cyclic preconditions. As expected, $FR^{SJ,<}$ is



(a) $FR^{SJ,<}$ compared to J^R



(b) $FR^{SJ,<}$ compared to $J^<$

Figure 5.5: Largest intermediate relations when expanding s_0 for different successor generator methods. Showing only instances solved by at least one method. Instances with action schemas with cyclic preconditions are marked in blue.

superior to the other two methods when considering instances with only acyclic preconditions. It is also interesting to note that $FR^{SJ,<}$ performs better than J^R and $J^<$ in most instances with cyclic preconditions as well. This indicates that our extension for cyclic preconditions is informed enough. In a few domains, $J^<$ has better performance than $FR^{SJ,<}$. This happens for cyclic action schemas which have no applicable instantiation to the initial state and that, simply by chance, $J^<$ is able to detect it earlier than $FR^{SJ,<}$ (e.g., there is an empty intermediate relation earlier in the join program of $J^<$ than in $FR^{SJ,<}$). In Figure 5.5 we can also observe instances without cyclic schemas but with very large intermediate relations. All these instances are from the Childsnack domain. The largest instances of this domain have close to 5000 applicable operators in the initial state and the largest intermediate relations is actually the final one for all cases. In fact, all methods have the same largest intermediate relation in these tasks, which are the final applicable tuples.

Cyclic action schemas: Focusing only on the strategy for cyclic action schemas, we investigated how much of the total search time is spent instantiating cyclic action schemas for tasks that were solved given our resource limits. The good news is that in most cases our strategies have a very good performance and are not the bottleneck of our planner, as one could expect. Considering the 86 tasks with cyclic actions that took more than 1 second to be solved, only 12 of them spent more than 10% of the time instantiating cyclic action schemas. The domains with the highest ratio were NoMystery, Hiking, and Freecell. Only NoMystery had instances where the instantiation of cyclic schemas used more than 20% of the time, reaching a maximum of 51% in an instance solved in 67 seconds. We expect that by scaling up the instance sizes this proportion would also increase. There are several other strategies that could be implemented to mitigate the issue with cyclic action schemas if needed. One of the main strategies (although not tested in our work) would be to implement hypertree decomposition [Gottlob *et al.*, 2002]. Such decomposition strategies are widely used not only in the database community but also in the scope of constraint programming problems [Vardi, 2000; Gottlob *et al.*, 2016]. Hypertree decompositions are fixed-parameter tractable and it might be inter-

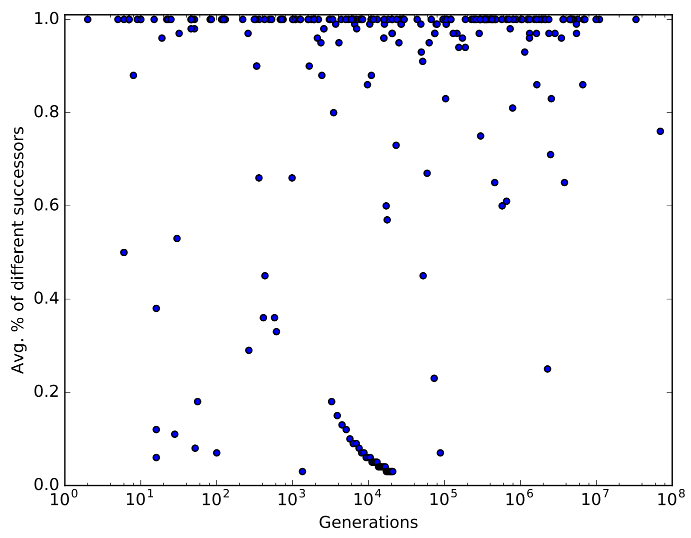


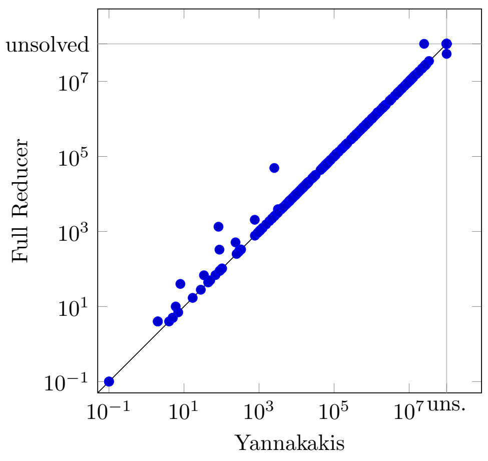
Figure 5.6: Plot comparing the number of states generated and the average proportion of different successors over all states of each instance.

esting to study it in the context of classical planning. For our current scenario, however, due to the size of most of the IPC problems, our heuristic strategies based on static analysis of the preconditions already have a good performance and are not critical to our planner.

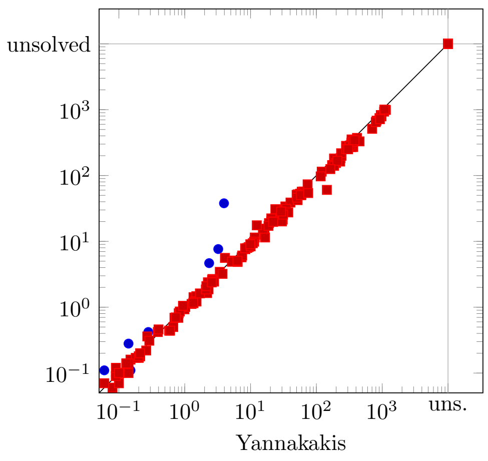
5.3 Comparison to the Yannakakis' Project-Join Method

Yannakakis' algorithm introduced in Chapter 4 has a potential gain for instances where many free variables appear only in the precondition of the action schemas. Although many instances present a potential gain, it is not clear whether this translates into practice for the IPC instances. Trying to quantify this amount, we ran an experiment where, for each state expanded in the search, we count the number of ground actions that lead to a different successor. In the end, we compute the proportion of instantiations that led to duplicates over all states in the search space. In other words, we are comparing the proportion of the generations that could be saved by using Yannakakis' algorithm, which would avoid multiple instantiations of actions leading to the same state (except for self-loops). The results are shown in the plot of Figure 5.6. Most of the instances present zero or close to zero gain when using Yannakakis' algorithm instead of $FR^{SJ,<}$. Yet, several instances would be considerably benefited by Yannakakis' algorithm. In particular, the Organic Synthesis domain has instances where only 6% of its instantiations lead to unique successors. Additionally, a few large instances also have potential gains close to 40%. In tasks with millions of states expanded, saving 20% of the state generations might lead to a significant performance boost. Also, we expect that this might contribute to an increase in the coverage by solving larger instances.

Unfortunately, the successor generator using the Yannakakis' algorithm, denoted from here on as Y, did not translate the potential gains into practice. In our experiment, Y solves 461 tasks, while $FR^{SJ,<}$ solves 464. In fact, Y barely reduced the number of states generated on average overall domains. Figure 5.7a shows a plot comparing the number of states generated before the last layer using each method. We can see that for



(a) Number of generated states until the last layer.



(b) Total search time. Organic Synthesis instances are marked in blue.

Figure 5.7: Generated states and total search time (in seconds) for $FR^{SJ,<}$ and Y.

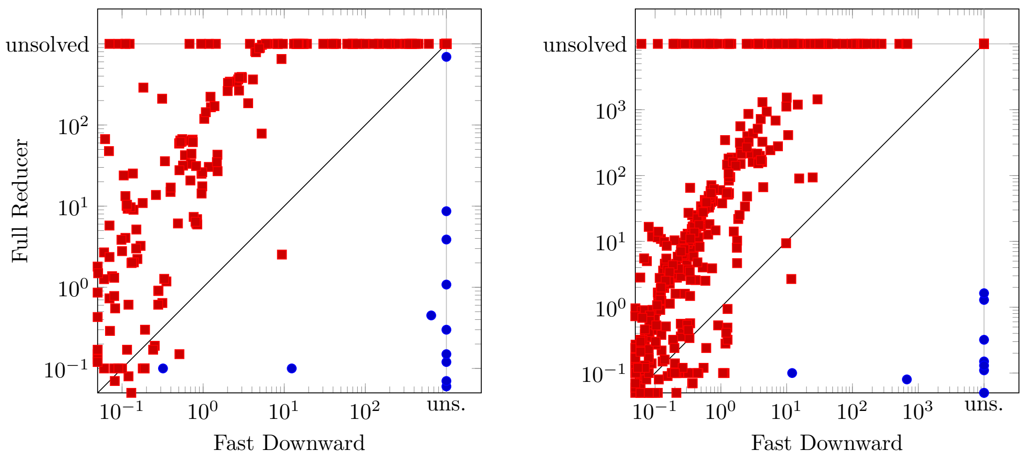
most of the instances, Y and $FR^{SJ,<}$ generate a very similar number of states. The main reduction, in proportion, is in Organic Synthesis, as expected. In the other domains, the gain was not significant. Besides that, both methods need almost the same search time in all solved tasks, as shown in Figure 5.7b. Although Y is superior to J and $J^<$, we do not have any evidence that it pays off compared to $FR^{SJ,<}$. The main reason for that is the overhead added by Y in comparison to $FR^{SJ,<}$. The Yannakakis' algorithm has an asymptotic complexity on the product of the input and output relations, while the $FR^{SJ,<}$ method has time complexity polynomial on the sum of the input and output relations.

Later on, however, we show that Y can improve coverage as soon as the size of the domains scale up.

5.4 Comparison to State-of-the-Art Grounded Planners

We also compared the performance of our two best methods, $FR^{SJ,<}$ and Y, to state-of-the-art grounded planners. We compare our methods to Fast Downward [Helmert, 2006], which is the base of most planners using heuristic search. The version of Fast Downward used was 19.06. In order to have a better comparison, we also implemented the *goal-count heuristic* Fikes and Nilsson [1971] in our planner. The goal-count heuristic is an action-independent heuristic that simply counts the number of ground atoms in the goal condition γ that are not satisfied in the state s being evaluated. This heuristic can be seen as an edit distance metric between a state and the goal condition. Intuitively, this heuristic assumes a state with more goal ground atoms is closer to a goal state. Thus, we compare our two best methods and Fast Downward using a breadth-first search algorithm and a GBFS guided by the goal-count heuristic. We provide the experimental results for the former first.

Our best method $FR^{SJ,<}$ is competitive with Fast Downward using a breadth-first search in some domains. In 42 of the 53 domains, the difference in coverage between $FR^{SJ,<}$ and Fast Downward was 5 instances or fewer. (We are not counting domains where one of the methods had coverage of 0.) Fast Downward solves 638 instances and $FR^{SJ,<}$ solves 464. Most of the domains are very easy to ground and thus it pays



(a) Using breadth-first search. (b) Using GBFS with goal-count as heuristic.

Figure 5.8: Total time in seconds for $FR^{SJ,<}$ and Fast Downward for different search methods. Showing only instances solved by at least one method. Organic Synthesis instances are marked in blue. Time for Fast Downward considers translator and search components time.

off to use the Fast Downward grounding procedure. However, as already known, Fast Downward has issues when grounding larger domains, such as Organic Synthesis. In this domain, particularly, Fast Downward can only ground eight of the 20 instances, while $FR^{SJ,<}$ and Y can solve 19 of them. This shows that, although grounded planners might be expected to perform better on average, a lifted planner using our techniques is still capable of solving problems on which a grounded planner has no chance.

The behavior is similar when using GBFS with the goal-count heuristic. While both $FR^{SJ,<}$ and Y solve 1050 instances, Fast Downward can solve 1213. Once again, the only where our methods achieve higher coverage is Organic Synthesis. In this scenario using a GBFS, the difference between our lifted planner and a grounded planner such as Fast Downward is even larger: Fast Downward can solve the same eight instances with breadth-first search and with GBFS, since its bottleneck is the grounding and not the search, while our methods equipped with GBFS can solve all 20 instances now. It is also interesting to notice the impressive performance of Y with the goal-count heuristic: it solves the entire Organic Synthesis domain in less than five seconds.

Another important question is whether our planner is able to compete with Fast Downward in terms of time and memory consumption. Figure 5.8 shows the plots comparing the total time of $FR^{SJ,<}$ and Fast Downward for different search methods. (For Fast Downward, total time indicates the sum of translation and search times.) In fact, Fast Downward is almost always faster than our planner. The only exception (besides tasks with very short running time) is the Organic Synthesis domain. This confirms that, for small and medium-size instances, the grounding costs are amortized during the search. When compared to Y, we observe the exact same trend. The same behavior occurs with respect to memory usage: except for small tasks and the Organic Synthesis domain, Fast Downward always needs less memory than $FR^{SJ,<}$ and Y.

5.5 Hard-to-Ground but Easy-to-Plan Domains

Although our methods are competitive with Fast Downward, our previous experiments make us wonder whether the Organic Synthesis domain is the only domain where our

		BFS			GBFS			
	# of Inst.	$FR^{SJ,<}$	Y	FD	$FR^{SJ,<}$	Y	FD	L-RPG
Organic Synthesis	56	**44**	**44**	20	47	**50**	20	14
Original	20	**8**	**8**	1	11	**14**	1	0
Alkene	18	**18**	**18**	17	**18**	**18**	17	14
MIT	18	**18**	**18**	2	**18**	**18**	2	0
Genome Edit Distance	312	44	44	**46**	**312**	**312**	**312**	113
Multi-step	156	22	22	**24**	**156**	**156**	**156**	48
Multi-step, split	156	**22**	**22**	**22**	**156**	**156**	**156**	65
Pipesworld Tankage	50	11	10	**14**	**22**	**22**	20	10
Total	418	**99**	98	80	381	**384**	352	137

Table 5.2: Coverage for the three different hard-to-ground domains. Comparing our two best methods, $FR^{SJ,<}$ and Y, to Fast Downward (FD) with two different search configuration: breadth-first search (BFS), and greedy best-first search using goal-count as heuristic (GBFS). Showing also the results for the L-RPG planner. For each search configuration, the best method in each domain is highlighted.

approaches are successful. To investigate this hypothesis, we compared our methods to Fast Downward in benchmarks with larger instances.

First, we investigated whether our methods are still competitive with Fast Downward in the instances used for the satisficing tracks of the IPCs, which are usually larger than the tasks from the optimal tracks. We used 730 instances over 33 different benchmarks used in the IPCs that use the extended STRIPS formalism supported by our planner. We compared only $FR^{SJ,<}$ and Y to Fast Downward, using once again two different search methods, a breadth-first search and a GBFS with the goal-count heuristic. When using the breadth-first search, $FR^{SJ,<}$ solves 67 tasks while Y solves 66. (The only instance solved by $FR^{SJ,<}$ and not by Y was a Scanalyzer instance which ran out of time with Y.) Fast Downward solves 94 instances with the same search configuration. When we use GBFS with goal-count, $FR^{SJ,<}$ solves 239 instances while Y solves 234. Once again, Fast Downward is superior, solving 338 instances.

When using a breadth-first search, 17 domains have at least one instance solved by some method. When using a GBFS, this number increases to 25. In both configurations, our methods have the same coverage as Fast Downward in only 5 of these domains. (Although the set of 5 domains is different using breadth-first search and GBFS.) In one domain, though, we achieve better coverage. This domain is, once again, the Organic Synthesis domain. The Organic Synthesis instances used for the satisficing track of the IPC 2018 are significantly harder than the ones used for the optimal track. This time, our best method is Y and it can only solve 12 of the 20 instances when using breadth-first search. However, this is still far superior to Fast Downward, since it only solves 3 of the 20 instances using this search method. Furthermore, Y with GBFS and goal-count solves a total of 18 instances, while the same setting does not improve coverage for Fast Downward.

The domains from the satisficing tracks of the IPCs are harder than the ones used for the optimal track, but they are not necessarily hard-to-ground and easy-to-plan. Most of these domains were developed to challenge planners without focusing on grounding and preprocessing. Thus, our last experiment focuses only on domains that are

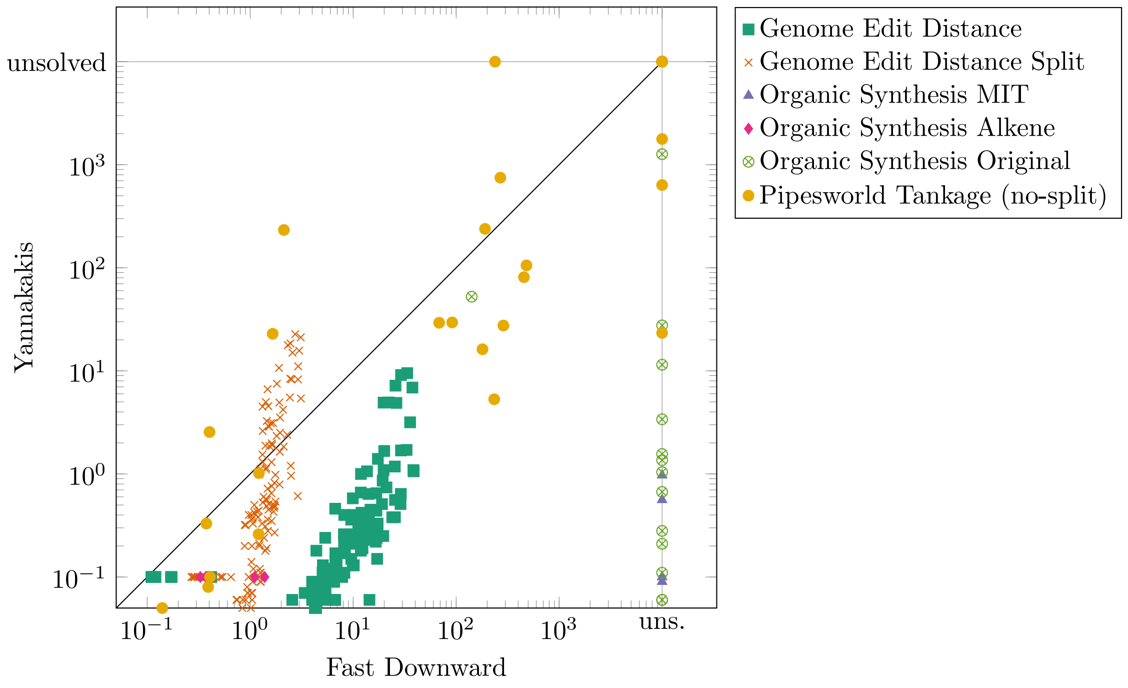


Figure 5.9: Total time (in seconds) for Y and Fast Downward with GBFS and the goal-count heuristic in the hard-to-ground domains. The time computed for Fast Downward is the sum of the grounding and the search procedures.

extremely challenging to ground. The domains supported by our planner are the following:

- **Organic Synthesis**: We consider here not only the IPC instances, but the entire set of 56 instances[3]. The set of instances is further divided into three subsets: "Organic Synthesis Original", "Organic Synthesis MIT", and "Organic Synthesis Alkene". Note that some of these 56 instances were also used for the IPC 2018.

- **Pipesworld Tankage (no-split)**: The Pipesworld Tankage domain, used in the IPC 2004 [Hoffmann and Edelkamp, 2005], with the original non-split action schemas. This domain was provided in two versions, one with regular action schemas and one where the action schemas were splitted manually to reduce the number of free variables of some schemas. The non-split version is considerably harder to ground.

- **Genome Edit Distance**: This domain was introduced by Haslum [2011] as a challenging practical application of planning. The domain was introduced in different formulations. The hardest formulation is the single-step version, but this version contains axioms, conditional effects, negated preconditions, and functions. Unfortunately, our planner cannot deal with all these extensions of STRIPS. Hence, we focus on the two versions of the so-called relational multi-step formulation (split and non-split). Although grounded planners can ground all instances using these formulations, the grounding still consumes a significant amount of time and resources.

[3]The instances can be found at `http://www.cs.ryerson.ca/~mes/publications/`. This set of instances was designed by Dr. Russell Viirre and converted into PDDL by Hadi Qovaizi. The authors also thank Prof. Mikhail Soutchanski for his assistance.

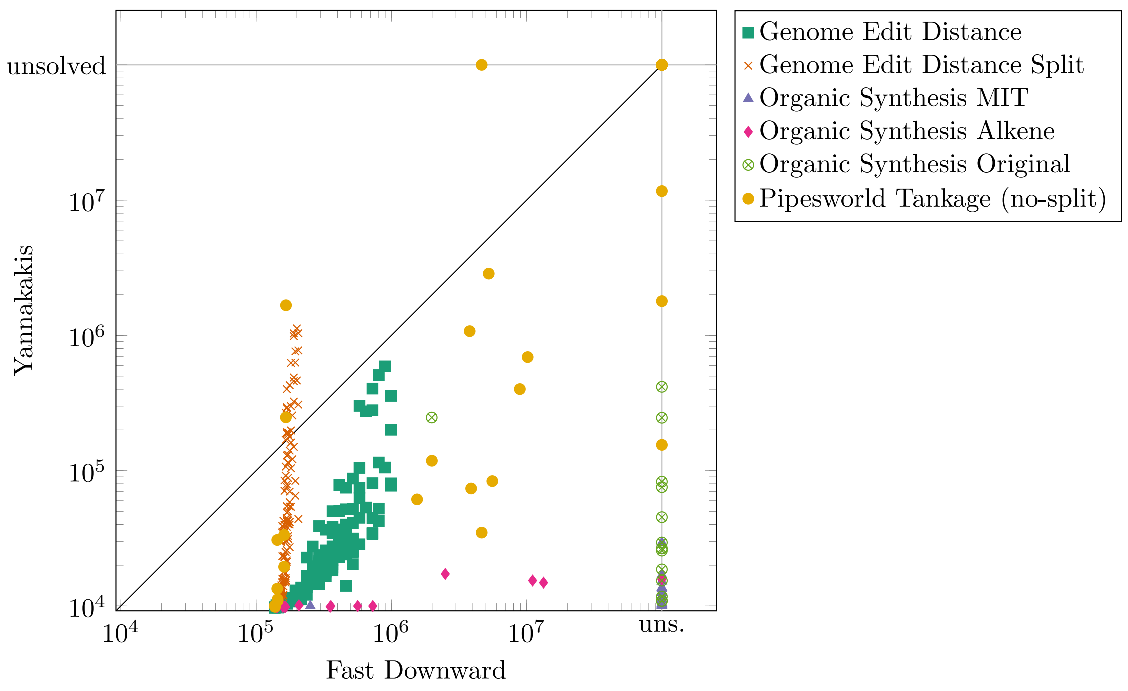


Figure 5.10: Peak memory (in kilobytes) for Y and Fast Downward with GBFS and the goal-count heuristic in the hard-to-ground domains. The value reported for Fast Downward is the peak memory considering all components (translator and search).

The two first blocks of Table 5.2 shows the coverage for all combinations of methods and search methods tested over these domains. Focusing on the runs using breadth-first search, we can see that our methods are competitive with Fast Downward in all domains and achieve a better coverage overall. In the Genome Edit Distance and the Pipesworld domain, our methods have a slightly lower coverage than Fast Downward though. This changes completely when we add the goal-count heuristic information to the search. Then, the search itself is not the bottleneck of the planner anymore, but now the bottleneck is the grounding. In this way, our methods can search over the state space very quickly, while Fast Downward first needs to ground the task, which consumes a lot of time and memory. For example, all tasks in the Organic Synthesis domain that Fast Downward failed to solve (considering both configurations), ran out of memory. The same is true for the Pipesworld domain. In this domain, there were four tasks (over 50) where our methods found a plan but Fast Downward was not able to ground the task.

In these hard-to-ground domains, our methods have faster run times even in the instances which Fast Downward can ground. Figure 5.9 shows the total time spent by Y and by Fast Downward in the configuration using GBFS with goal-count. The lifted method is faster than Fast Downward in most of the tasks. Particularly in the Genome Edit Distance (no-split) domain, the grounding process takes too long and dominates the total time. This plot also shows that the splitted versions end up reducing the benefits of our methods.

When comparing peak memory, our methods are also superior in these domains. Figure 5.10 compares the peak memory for Y and for Fast Downward with GBFS and the goal-count heuristic. All the instances of the Genome Edit Distance (split) domain use the same amount of memory in Fast Downward because they only need the amount of memory that the planner pre-allocates. Comparing the plots of Figure 5.9 and Figure 5.10, we can see some similarities. In most of the time, the tasks where Fast Down-

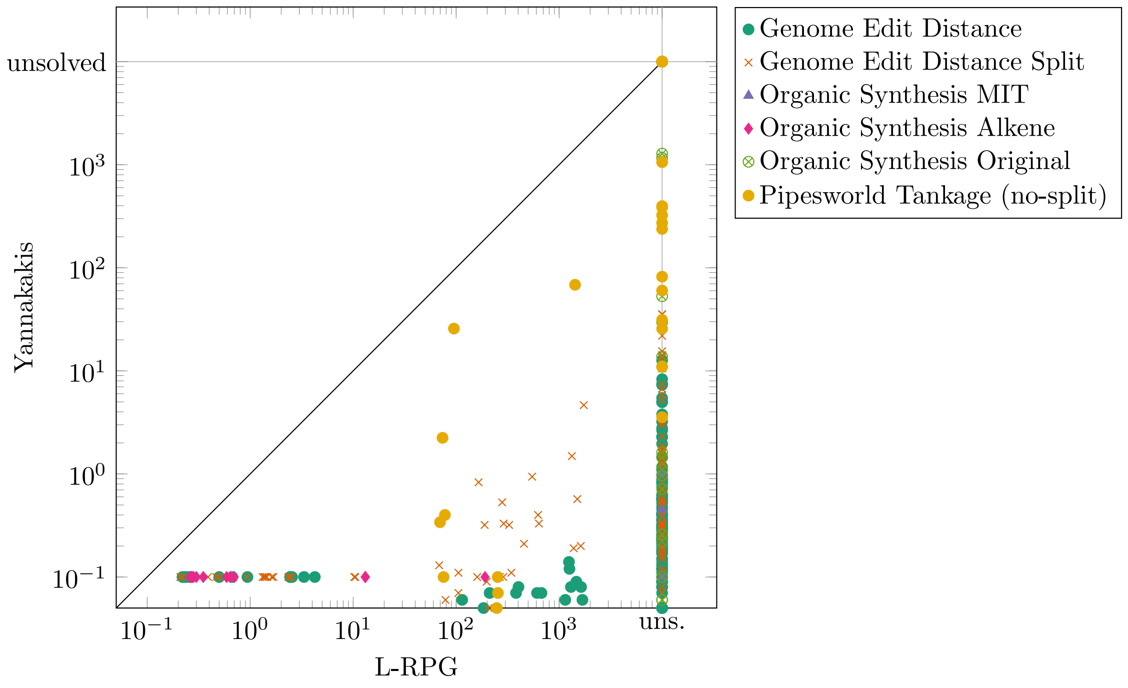


Figure 5.11: Total time (in seconds) for Y, with GBFS and the goal-count heuristic, and for L-RPG in the hard-to-ground domains.

ward used more memory are also the ones where it also needed more time. In fact, in these instances, both the run time and memory usage were dominated by the translator component. This supports the main claim of our work: the lifted methods introduced here are more adequate tools for domains where the grounding the main bottleneck. If these tasks were scaled up, Fast Downward would probably not be able to finish the grounding, while our lifted planner still has chance to solve them.

5.6 Comparison to Lifted Planners

We also compared our methods to the L-RPG planner [Ridder, 2014]. The L-RPG planner might be considered the state-of-the-art heuristic search planner using lifted representations. As explained before, it uses a lifted representation of action schemas together with a grounded representation of states. Furthermore, it also computes equivalence relations between the objects of the task in order to find symmetrical objects and speed-up the search. It is worth to note that, in contrast to our methods, the L-RPG planner was implemented to test heuristics using lifted representations and it does not focus on successor generation, as we do here.

The coverage of L-RPG is showed in the last column of Table 5.2. We tested the best configuration of L-RPG, which uses a lifted version of the FF heuristic [Hoffmann and Nebel, 2001]. We can see that both of our methods when used together with GBFS, solve more instances than L-RPG. In fact, our methods also solve more instances than L-RPG even when using a breadth-first search, except for the Genome Edit Distance domains. These results show that, in these hard-to-ground domains, the successor generation is one of the biggest challenges. They also demonstrate that even a better-informed heuristic is not useful in such domains if the successor generation is not efficient.

We also compared the time usage of our methods and L-RPG. Figure 5.11 shows the total time for Y, using GBFS with goal-count, and L-RPG. Our method is far more efficient than L-RPG in all cases. Once again, these results show that in these domains a more efficient successor generation method is more important than a more informed heuristic. Since these tasks have plans which are easy to find, the goal-count heuristic is informed enough to guide the search. In the case of L-RPG with FF, the heuristic is also informed, but the successor generation method by L-RPG becomes the bottleneck.

5.7 Limitations

Our method is competitive with grounded planners in "easy-to-ground and easy-to-plan" problems and it outperforms Fast Downward in "hard-to-ground and easy-to-plan" domains. However, are our methods already good enough for "hard-to-ground and hard-to-plan" tasks?

Unfortunately not. We tested our methods in the STRIPS domains introduced by Gnad *et al.* [2019]. These domains are based on IPC domains but scaled up significantly. For example, their "Blocksworld large" domains have instances with more than 100 blocks. We tested 100 instances of these domains (Blocksworld, Depots, Satellite, and TPP) and we were not able to solve any, independently of our search algorithm. The main cause for this bad performance is simply that the state space is too large and, given our available heuristics, our planner cannot handle them. This shows a relation between heuristic quality and the performance of a lifted planner: a bad heuristic will lead to more expansions and, since every new state expanded needs to have its action schemas instantiated by the lifted planner, this cause a lot of harm to the planner. It is safe to say that a bad heuristic is more harmful to a lifted planner, where the grounding overhead is per expaded state, then to a grounded planner, where the grounding overhead is constant to the task.

Chapter 6

Conclusion

Discussion

In this thesis, we introduced a new approach to planning using lifted representations. We focused on the problem of successor generation in planning, which is an important operation in planners based on heuristic search. In our study, using a perspective of planning as successive database progressions, we used well-known and very successful techniques from database theory and relational algebra to create a lifted planner. We showed how, given a state s, the enumeration of all applicable instantiations of an action schema can be understood as a database query, where the state is the database and the action preconditions form the query. We demonstrated how to apply query answering techniques to perform this instantiation during the search. Furthermore, we showed that a very large percentage of standard problems result in queries that are acyclic, which is a positive result, since this represents a special case where the worst-case complexity for computing the query is reduced from exponential in the size of the state to polynomial in the size of the state and the number of applicable actions. We implemented successor generators based on database techniques specific for acyclic queries (e.g., full-reducer program and Yannakakis' algorithm). We conducted an extensive empirical analysis of the performance of our prototype implementation over several benchmarks, including domains from the IPCs and domains that were too hard to be included in these competitions. Perhaps surprisingly, our planner is competitive with the most popular grounded planner, Fast Downward. It also outperforms the L-RPG planner, which can be considered the state-of-the-art planner using heuristic search with a lifted representation. To the best of our knowledge, our planner is the first planner able to scale up to domains such as Organic Synthesis. Table 6.1 shows an overview (with respect to coverage) of the results presented in this thesis.

Our main contribution is not the lifted planner itself, but a conceptual contribution for planning using lifted representations. Our theoretical and empirical analysis sheds new light on a topic that has not been much explored by the planning community. We showed that indeed and perhaps contradicting the folklore, lifted planning is not inherently worse than grounded planning and it can achieve very good performance in several benchmarks. In our opinion, the purpose of a lifted planner is to increase the number of problems that can be solved using automated planning, and not necessarily to compete with grounded planners. The characteristics of planners using lifted representations simply make them more suitable to a different "spectrum" of problems than grounded planners. While grounded planners are better suited to problems where the search is the bottleneck, our method is better suited to problems where grounding is the

		BFS			GBFS		
	# of Inst.	$FR^{SJ,<}$	Y	FD	$FR^{SJ,<}$	Y	FD
IPC – Optimal Track	1560	464	461	586	1050	1050	1168
IPC – Satisficing Track	730	67	66	94	239	234	338
Hard-to-ground Domains	418	99	98	80	381	384	352

Table 6.1: Summary of the coverage results presented in this thesis. The coverage is grouped for each distinct benchmark set used.

critical part.

We believe that our method can be extended to deal with problems that are hard-to-ground *and hard-to-plan*. There is a lot of room for improvement in our method, in particular concerning better heuristics for lifted representations. There was no advancement in this area in recent years and we believe this is mainly due to historical reasons. In particular, because the winners of the early editions of the IPC were grounded planners. We expect that, given our motivating results on the potential of lifted planners, this might stimulate research on the direction of heuristics for lifted representations. Additionally, lifted planning might sound more challenging than grounded planning also simply because, theoretically, the planner might need to solve a NP-complete problem at every state (i.e., finding the applicable actions). As shown in our experiments, additional complexity does not translate into a huge overhead in practice and these NP-complete problems are fairly simple in almost all domains tested. This phenomenon is not necessarily surprising since several other research areas (e.g., SAT and Operations Research) have developed very efficient techniques to solve problems that are very hard in theory.[1]

In summary, this work introduces a new way to think about lifted planning and demonstrates that there is still many possibilities for improvements in this area.

Future Work

We conclude this work by presenting some ideas for future work that can help both lifted and grounded planners.

Heuristics for Lifted Representations

There are two main issues when computing heuristics in a lifted representation: (i) we do not have the grounded actions and hence many structures (e.g., causal graph and domain-transition graphs [Helmert, 2004]) are also not easy to compute; and (ii) the computation of a heuristic might be as expensive as grounding the entire state space. For example, the heuristic introduced by McDermott [1996] performs some kind of

[1]The author cannot resist to quote Vardi [2010] – *"When I was a graduate student, SAT was a "scary" problem, not to be touched with a 10-foot pole. Garey and Johnson's classical textbook showed a long sad line of programmers who have failed to solve NP-complete problems. Guess what? These programmers have been busy! [...] Today's SAT solvers, which enjoy wide industrial usage, routinely solve SAT instances with over one million variables. How can a scary NP-complete problem be so easy? What is going on? [...] Indeed, SAT does seem hard in the worst case. There are SAT instances with a few hundred variables that cannot be solved by any existant SAT solver. "So what?" shrugs the practitioner, "these are artificial problems"* – and hope the same occurs to lifted planning.

backward-chaining from the goal atoms until it reaches the ground atoms which are true in the state being evaluated. This backward-chaining works by regressing each fact in the goal and, for this regression, it is necessary to ground the action schemas. For larger problems, the heuristic might be too expensive to compute and its cost per state might be similar to grounding the entire task.

In this work, we used goal-count because it is an estimate that does not rely on actions. A possible choice for future implementation in this direction is to use estimates based on the width of the tasks [Lipovetzky and Geffner, 2012, 2014]. These techniques have shown good performance in several planning domains and do not need any information about transitions in the state space of the structure of the problem. In fact, Francès *et al.* [2017] show that using width-based search methods as black-box evaluators, having access only to the structure of the states and the atoms in the goal, can overperform state-of-the-art planners in suboptimal planning.

Last, we can consider heuristics using partially grounded actions, where only part of the free variables are instantiated. This approach is similar to abstract some of the transitions in the state space. It is, however, not clear how to decide which free variables to abstract without having much knowledge of the state space structure.

Representational Features

The formulations supported by our planner right now are very restrictive. There are, however, some features extending the STRIPS formalisms that can be easily integrated into our approach. Conditional effects and universally quantified effects can be seen as conjunctive queries. Negated preconditions can be preprocessed or, alternatively, we can create "complementary" relations for relations that appear negated in some precondition.

Axioms are not so trivial to adapt to our planner, but we can use other database techniques to deal with them. We can represent the axioms as a set of Datalog rules [Ullman, 1988]. In a given state, the extensional database (EDB) of the Datalog program is represented by the state relations itself, while the intentional database (IDB), which are the inference rules, are the axioms. We can then evaluate axioms (possibly with negation and recursion) using traditional Datalog methods.

Structural Decompositions of Action Schemas

As shown in Chapter 5, some instances had more than 30% of this run-time spent on the instantiation of cyclic action schemas. In our benchmarks, we treated it as acceptable because the size of these instances is considerably small. However, as we aim at scaling up the size of instances solvable by our planner, it might be interesting to look into better successor generator methods for cyclic action schemas. The main idea in this part is to try to analyze and apply structural decompositions of action schemas [Gottlob *et al.*, 2000, 2001, 2002]. This can lead to further insights about the hardness of instantiating cyclic action schemas. For example, it might be more efficient to use fixed-parameter tractable algorithms based on the hypertree width of the action schemas or even some other heuristic approaches that do not rely on decomposition Gottlob *et al.* [2002, 2016].

The idea of partially grounding actions might also be useful to turn cyclic action schemas into acyclic ones. For example, if we are able to identify free variables that, if removed from the hypergraph of the query, would make the hypergraph acyclic, we could ground only these free variables by some enumeration strategy (also creating several distinct copies of the action schema, one for each possible combination of values instantiating the free variables) and then use our techniques for acyclic queries to evaluate them more quickly. Although this might be slightly more expensive, in particular, because we would have several copies of each schema, this also has the potential to reduce the memory usage of cyclic action schemas, once they turn acyclic.

Magic-Sets and Grounding

The most popular grounding method in planning is already based on database techniques [Helmert, 2009]. As mentioned in Chapter 3, Helmert formulates the planning task as a Datalog program and uses the minimal model of this Datalog program as a representation of the relaxed-reachable state space. However, finding this minimal model is very expensive and time consuming for many domains.

Using more advanced database techniques, we can also try to improve this grounding procedure. One common optimization in Datalog programs are the so-called magic-sets [Bancilhon *et al.*, 1986]. Magic-sets introduce new predicates to the Datalog program to speed-up the computation of its minimal model. This rewriting of the program allows a faster computation of the minimal model using forward chaining (i.e., bottom-up computation) and simultaneously cuts down the irrelevant intermediate facts produced during the computation. This optimization technique is not only useful in relational databases but also in Answer Set Programming [Alviano *et al.*, 2012], for example. When applied to the Datalog program used to ground the relaxed-reachable state, this optimization could lead to savings in time and memory.

Bibliography

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

Mario Alviano, Wolfgang Faber, Gianluigi Greco, and Nicola Leone. Magic sets for disjunctive datalog programs. *Artificial Intelligence*, 187:156–192, 2012.

Carlos Areces, Facundo Bustos, Martín Ariel Dominguez, and Jörg Hoffmann. Optimizing planning domains by automatic action schema splitting. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS 2014)*, pages 11–19, 2014.

Fahiem Bacchus. The AIPS '00 planning competition. *AI Magazine*, 22(3):47–56, 2001.

Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11:625–656, 1995.

François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS 1986)*, pages 1–15, 1986.

Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, 1983.

Philip A. Bernstein and Nathan Goodman. Power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.

Blai Bonet and Hector Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.

Michael Cashmore, Maria Fox, and Enrico Giunchiglia. Partially grounded planning as quantified boolean formula. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013)*, pages 29–36, 2013.

James E. Doran and Donald Michie. Experiments with the graph traverser program. *Proceedings of the Royal Society A*, 294(1437):235–259, 1966.

Kutluhan Erol, Dana S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1–2):75–88, 1995.

Ronald Fagin. Acyclic database schemes (of various degrees): A painless introduction. In *Proocedings of the 8th Colloquium Trees in Algebra and Programming (CAAP 1983)*, pages 65–89, 1983.

Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

Guillem Francès and Hector Geffner. ∃-strips: Existential quantification in planning and constraint satisfaction. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pages 3082–3088, 2016.

Guillem Francès, Miquel Ramírez, Nir Lipovetzky, and Hector Geffner. Purely declarative action descriptions are overrated: Classical planning with simulators. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI 2017)*, pages 4294–4301, 2017.

Martin Gebser, Roland Kaminski, Murat Knecht, and Torsten Schaub. plasp: A prototype for PDDL-based planning in ASP. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, pages 358–363, 2011.

Daniel Gnad, Alvaro Torralba, Martın Domınguez, Carlos Areces, and Facundo Bustos. Learning how to ground a plan – partial grounding in classical planning. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI-19)*, 2019.

Nathan Goodman and Oded Shmueli. The tree property is fundamental for query processing. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS 82)*, pages 40–48, 1982.

Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM*, 48(3):431–498, 2001.

Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *Journal of Computing and System Sciences*, 64(3):579–627, 2002.

Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree decompositions: Questions and answers. In *Proceedings of the 35th ACM Symposium on Principles of Database Systems (PODS 2016)*, pages 57–74, 2016.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Patrik Haslum. Reducing accidental complexity in planning problems. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 1898–1903, 2007.

Patrik Haslum. Computing genome edit distances using domain-independent planning. In *ICAPS 2011 Workshop on Planning and Learning (PAL 2011)*, pages 45–51, 2011.

Malte Helmert. A planning heuristic based on causal graph analysis. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 161–170, 2004.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5–6):503–535, 2009.

Carl Hewitt. PLANNER: A language for proving theorems in robots. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI 1969)*, pages 295–302, 1969.

Jörg Hoffmann and Stefan Edelkamp. The deterministic part of IPC-4: An Overview. *Journal of Artificial Intelligence Research*, 24:519–579, 2005.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Henry A. Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 1992)*, pages 359–363, 1992.

Robert A. Kowalski. *Logic for problem solving*, volume 7 of *The computer science library: Artificial intelligence series*. North-Holland, 1979.

Fangzhen Lin and Raymond Reiter. How to progress a database. *Artificial Intelligence*, 92(1–2):131–167, 1997.

Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 540–545, 2012.

Nir Lipovetzky and Hector Geffner. Width-based algorithms for classical planning: New results. In *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, pages 1059–1060, 2014.

Arman Masoumi, Megan Antoniazzi, and Mikhail Soutchanski. Modeling organic chemistry and planning organic synthesis. In *Proceedings of the 1st Global Conference on Artificial Intelligence (GCAI 2015)*, pages 176–195, 2015.

Rami Matloob and Mikhail Soutchanski. Exploring organic synthesis with state-of-the-art planning techniques. In *ICAPS 2016 Scheduling and Planning Applications Workshop (SPARK 2016)*, 2016.

Drew V. McDermott. A heuristic estimator for means-ends analysis in planning. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS 96)*, pages 142–149, 1996.

Drew V. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2):35–55, 2000.

Christos H. Papadimitriou and Mihalis Yannakakis. On the complexity of database queries. *Journal of Computer and System Sciences*, 58(3):407–427, 1999.

J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR 92)*, pages 103–114, 1992.

Bernardus Ridder. *Lifted heuristics: towards more scalable planning systems*. PhD thesis, King's College London, 2014.

Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. SAT-based parallel planning using a split representation of actions. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 281–288, 2009.

Gabriele Röger, Silvan Sievers, and Michael Katz. Symmetry-based task reduction for relaxed reachability analysis. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS 2018)*, pages 208–217, 2018.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3rd Edition)*. Pearson Education, 2010.

Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. `https://doi.org/10.5281/zenodo.790461`, 2017.

Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.

Robert Endre Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I of *Principles of computer science series*. Computer Science Press, 1988.

Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II of *Principles of computer science series*. Computer Science Press, 1989.

Moshe Y. Vardi. Constraint satisfaction and database theory: A Tutorial. In *Proceedings of the 19th ACM Symposium on Principles of Database Systems (PODS 2000)*, pages 76–85, 2000.

Moshe Y. Vardi. On P, NP, and computational complexity. *Communications of the ACM*, 53(11):5, 2010.

Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB 1981)*, pages 82–94, 1981.

Håkan L. S. Younes and Reid G. Simmons. On the role of ground actions in refinement planning. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS 2002)*, pages 54–62, 2002.

Håkan L. S. Younes and Reid G. Simmons. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, 20:405–430, 2003.

C. T. Yu and M. Z. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *Proceedings of the IEEE Computer Society's Third International Computer Software and Applications Conference (COMPSAC 1979)*, pages 306–312, 1979.

Neng-Fa Zhou, Roman Barták, and Agostino Dovier. Planning as tabled logic programming. *Theory and Practice of Logic Programming*, 15(4–5):543–558, 2015.