Lifted Successor Generation using Query Optimization Techniques

Augusto B. Corrêa and **Florian Pommerening** and **Malte Helmert**
University of Basel, Switzerland
{augusto.blaascorrea,florian.pommerening,malte.helmert}@unibas.ch

Guillem Francès
Universitat Pompeu Fabra, Spain
guillem.frances@upf.edu

Abstract

The standard PDDL language for classical planning uses several first-order features, such as schematic actions. Yet, most classical planners ground this first-order representation into a propositional one as a preprocessing step. While this simplifies the design of other parts of the planner, in several benchmarks the grounding process causes an exponential blowup that puts otherwise solvable tasks out of reach of the planners. In this work, we take a step towards planning with lifted representations. We tackle the successor generation task, a key operation in forward-search planning, directly on the lifted representation using well-known techniques from database theory. We show how computing the variable substitutions that make an action schema applicable in a given state is essentially a query evaluation problem. Interestingly, a large number of the action schemas in the standard benchmarks result in acyclic conjunctive queries, for which query evaluation is tractable. Our empirical results show that our approach is competitive with the standard (grounded) successor generation techniques in a few domains and outperforms them on benchmarks where grounding is challenging or infeasible.

Introduction

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

While there is a remarkable diversity of planning techniques (e.g., Kautz and Selman 1992; Bonet and Geffner 2001; Torralba et al. 2017), most PDDL planners ground the first-order representation of the problem into a propositional one as a preprocessing step. This makes sense since the description and implementation of solution techniques becomes easier at the propositional level. The ground representation can be exponentially larger in the number of parameters of the action schemas, but there are efficient grounding techniques (e.g., Helmert 2009) that overall seem to make this preprocessing an effective strategy. The shortcoming of this strategy, however, is that the entire set of ground actions needs to be stored in memory.

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

Our work is a step towards planning directly on the first-order representations. We use well-known techniques from database theory to tackle the task of successor generation, one of the key operations when planning using heuristic search. The enumeration of all applicable ground actions derived from an action schema in a given state s can be seen as a database query where s is a database and the action precondition is a query. We introduce successor generation techniques based on standard query evaluation algorithms (Ullman 1989). We analyze a large number of benchmarks from the literature and find out that the preconditions of the majority of action schemas correspond to acyclic conjunctive queries, which can be evaluated in time polynomial in the size of the state and number of applicable actions (Yannakakis 1981).

We report empirical results for a prototype implementation of these techniques on a wide set of benchmarks, focusing on successor generation. The use of lifted heuristics (cf. Ridder 2013) in our implementation is left for future work. Over the IPC benchmarks, our prototype is competitive with a grounded baseline planner in a few domains. In a set of focus benchmarks known to be challenging for grounding, however, our prototype outperforms the grounded planners, which often are not able to compute the ground representation of the problem. In these hard-to-ground domains, our prototype solves almost three times more instances than the previous state of the art for lifted planners (Ridder 2013). Interestingly, a simple greedy best-first search with the goal-count heuristic obtains, to the best of our knowledge, the best performance reported so far on the challenging Organic Synthesis domain (Matloob and Soutchanski 2016).

First-Order Classical Planning Problems

A STRIPS planning task is a tuple $\Pi = \langle \mathcal{P}, O, \mathcal{A}, s_0, \gamma \rangle$, where $\mathcal{P}$ is a set of *predicate symbols*, O is a set of *objects*, $\mathcal{A}$ is a set of *actions schemas*, s_0 is the *initial state*, and γ is a *goal condition*.

Every predicate $P \in \mathcal{P}$ has an associated arity. If P is an n-ary predicate and $\vec{t} = \langle t_1, \dots, t_n \rangle$ is a tuple of variables or objects from O, then $P(\vec{t})$ is an *atom*. When we *ground* (i.e., substitute) the variables of an atom $P(\vec{t})$ with objects from O, we obtain a *ground atom*. A *state*, such as s_0, is a set of ground atoms, interpreted as the set of all true atoms in this state. Similarly, the goal condition γ is a set of ground atoms. Any state s such that $\gamma \subseteq s$ is a *goal state*. An action schema $a[\Delta] \in \mathcal{A}$ is a tuple $\langle \mathit{pre}(a[\Delta]), \mathit{add}(a[\Delta]), \mathit{del}(a[\Delta]) \rangle$, corresponding to the *precondition*, the *add list* and the *delete list* of $a[\Delta]$. These three elements of $a[\Delta]$ are finite sets of atoms defined over $\mathcal{P}$, such that Δ is the set of free variables occurring in any atom of some element of $a[\Delta]$.

We can *ground* an action schema $a[\Delta]$ by substituting the free variables Δ by objects in O, which results in a *ground action* a without free variables (sometimes referred only as *action* when the context is clear).

An action a is *applicable* in a state s if $\mathit{pre}(a) \subseteq s$. When applying a in s, the *successor state* s' is defined as $s' = (s \setminus \mathit{del}(a)) \cup \mathit{add}(a)$. A sequence of ground actions $a_1, \dots, a_n$ is applicable in state s if each a_i is applicable in the state generated by applying $a_1, \dots, a_{i-1}$ from s. The solution of a planning problem is a sequence of ground actions applicable to s_0 that leads to a goal state. This sequence of actions is called a *plan*.

Database Theory

We briefly review some relevant background from database theory (Ullman 1989; Abiteboul, Hull, and Vianu 1995).

A *database* $\mathcal{D} = \langle D, \mathcal{R} \rangle$ is composed of a *domain* D and a set $\mathcal{R}$ of finite relations over D. Each relation $R \in \mathcal{R}$ is a set $R \subseteq D^{ar(R)}$, where $ar(R) \in \mathbb{N}$ is the *arity* of the relation. We adhere to the convention of identifying databases with a logical theory, where a tuple $\vec{c} \in R$ is seen as a ground atom $R(\vec{c})$ of a first-order language.

A *query* Q is a mapping between a database $\mathcal{D}$ and a relation $Q(\mathcal{D})$. We focus on *conjunctive queries* where $Q(\mathcal{D})$ is defined as

$$\{\vec{c} \mid \exists \vec{z}\, \varphi(\vec{c}, \vec{z})\} \text{ for some } \varphi(\vec{x}, \vec{z}) = \bigwedge\nolimits_{i=1}^{n} R_i(\vec{t_i})$$

where each $\vec{t_i}$ is a tuple of length $ar(R_i)$ with elements from D and variables from $\vec{x}$ and $\vec{z}$. We write $\mathit{vars}(\vec{t})$ for the set of variables used in such a tuple. Conjunctive queries are often written in *rule form* with *distinguished variables* $\vec{x}$:

$$Q(\vec{x}) \text{ :- } R_1(\vec{t_1}), \dots, R_n(\vec{t_n}),$$

which leaves implicit the existential quantification of variables $\vec{z}$ that appear on the right-hand side (the *body* of the rule) but not on the left-hand side (its *head*).

To illustrate, consider the database $\langle D, \{R_1, R_2\} \rangle$ with $D = \{1, 2, 3\}$, $R_1 = \{\langle a, b, c \rangle \in D^3 \mid a \leq b, c \text{ is odd}\}$, and $R_2 = \{\langle a, b \rangle \in D^2 \mid a \neq b\}$. Then the conjunctive query $Q_1(x) \text{ :- } R_1(2, x, x)$ evaluates to $R_{Q_1} = \{\langle 3 \rangle\}$ and $Q_2(y, x) \text{ :- } R_1(2, x, x), R_2(x, y)$ evaluates to $R_{Q_2} = \{\langle 1, 3 \rangle, \langle 2, 3 \rangle\}$.

Every conjunctive query $Q(\vec{x}) \text{ :- } R_1(\vec{t_1}), \dots, R_n(\vec{t_n})$ can be associated with a hypergraph $\mathcal{H}(Q) = \langle V, E \rangle$ with one vertex $v \in V$ for each variable occurring in the body of Q and one hyperedge $e_i = \mathit{vars}(\vec{t_i}) \in E$ for each atom $R_i(\vec{t_i})$. The *GYO reduction* of such a hypergraph is another hypergraph obtained through a simple iterative procedure that removes one hyperedge $e \in E$ at each step until E has a single hyperedge or no edge removal can be performed. At each step, we can remove $e \in E$ iff another hyperedge $f \in E$ exists such that the variables in $e \setminus f$ only appear in e. We say that such a step removes e in favor of f. $\mathcal{H}(Q)$ is *acyclic* iff its GYO reduction is a hypergraph with a single hyperedge (Ullman 1989).[1] A conjunctive query Q is *acyclic* iff its hypergraph $\mathcal{H}(Q)$ is.

Complexity of Query Evaluation If the input size of a query evaluation problem is $I = \|Q\| + \|\mathcal{D}\|$ and its output size is $U = \|Q(\mathcal{D})\|$, one would prefer query evaluation algorithms that are *output-polynomial*, in the sense that their time and space complexity is polynomial in $I + U$. No complete algorithm can have a runtime better than $O(I + U)$, since it needs to read the input completely and output $Q(\mathcal{D})$. Conjunctive query evaluation is NP-hard in general (Chandra and Merlin 1977). For example, computing $|Q(\mathcal{D})|$ is #P-complete (Creignou, Khanna, and Sudan 2001). Hence, no efficient general algorithm can exist unless P = NP. However, output-polynomial algorithms exist for acyclic conjunctive queries (Yannakakis 1981). Papadimitriou and Yannakakis (1999) show that conjunctive query evaluation considering query size or number of variables as the parameter is $W[1]$-complete, and that although the extension of acyclic queries with atoms of the form $x \neq y$ is NP-hard, it is fixed parameter tractable with regard to the same parameters as above (Papadimitriou and Yannakakis 1999).

Relational Algebra Later, we describe conjunctive query evaluation algorithms in terms of the select, project, join and rename (SPJR) operations of *relational algebra*, a formalism which is equivalent in expressive power to conjunctive queries (Codd 1970), but has a more operational flavor. We give an intuitive description of this algebra and refer to Abiteboul, Hull, and Vianu (1995) for a formal definition.

The SPJR algebra is based on named relations, often called *tables*. A table is a relation where each position (called *column*) has an *attribute name*. Given two tables R and S, the basic operations are (*i*) to *rename* a column; (*ii*) to *project* R into a set of attributes Y, obtaining a relation $\pi_Y(R)$ where some columns of R have been removed or rearranged; (*iii*) to *select* some tuples from R that either coincide on two different attributes x_i and x_j ($\sigma_{x_i = x_j}(R)$), or for which an attribute x_i has a particular constant value c ($\sigma_{x_i = c}(R)$); and (*iv*) to *join* R and S. The (natural) join $R \bowtie S$ selects all tuples from the Cartesian product of R and

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

S that match on shared attribute names, and then projects out all copies but one of the duplicate attributes. In addition to these four basic operations, one can define the *semi-join* $R \ltimes S$ as the projection of $R \bowtie S$ to the attributes of R.

Evaluating Conjunctive Queries

The evaluation of conjunctive queries can be described in terms of *query programs*, sequences of assignments of SPJR algebra expressions to relation variables, such as $\langle R := R \ltimes S, S := S \ltimes R, Q := \pi_X(R \bowtie S)\rangle$. These are interpreted like imperative programming languages, i.e., the first step in the example above replaces the relation R with $R \ltimes S$, and the next step uses this updated relation. These assignments are local to the program, i.e. the actual database is not changed. Query programs that only use (semi-)joins are called (semi-)join programs.

A conjunctive query $Q(\vec{x}) \coloneq R_1(\vec{t}_1), \ldots, R_n(\vec{t}_n)$ can in general be evaluated in two steps: take the natural join of all atoms in the body and then project the result into the distinguished attributes. To write this as a query program, we first map each logical atom $R_i(\vec{t}_i)$ to a relational-algebraic relation with attributes $vars(\vec{t}_i)$, selecting tuples that agree with $\vec{t}$ on constants and repeated variable names. The resulting relation is called R'_i. For example, if $R = \{\langle a, b, c\rangle \in \{1, 2, 3\}^3 \mid a \leq b, c \text{ is odd}\}$ and $t = \langle 2, x, x\rangle$, then $R' = \{\langle x \mapsto 3\rangle\}$. This mapping can be evaluated in time linear in the size of R_i by a query program consisting of selection and projection operations. The query program for Q then is $Q := \pi_X(R'_1 \bowtie \cdots \bowtie R'_n)$ where the join is left-associative. The evaluation of this query program can take time exponential in both I and U due to a combinatorial explosion caused by the joins. In general, the order of joins, as well as interleaving joins and projections, can make an exponential difference in time and space complexity.

Evaluating Acyclic Conjunctive Queries

When the conjunctive query is acyclic, output-polynomial evaluation algorithms exist. We next sketch two standard algorithms: an algorithm based on a full reducer of a query (Bernstein and Goodman 1981) and the algorithm by Yannakakis (1981).

Full Reducer We first consider conjunctive queries $Q(\vec{x}) \coloneq R_1(\vec{t}_1), \ldots, R_n(\vec{t}_n)$ where $vars(\vec{x}) = \bigcup_i vars(\vec{t}_i)$. In such queries all variables are distinguished and no projection is needed. A *full reducer* of such a query is a semi-join program that filters out tuples from the relations R'_i described above in a way that all remaining tuples are part of a tuple in $Q(\mathcal{D})$. Full reducers exist only for acyclic conjunctive queries (Bernstein and Goodman 1981). Say the GYO algorithm removes the hyperedges of $\mathcal{H}(Q)$ in the order $(e_1, f_1), \ldots, (e_m, f_m)$, where (e_i, f_i) indicates that iteration i removed hyperedge e_i in favor of hyperedge f_i. Since Q is acyclic, all hyperedges but the last one can be removed, so the set of all e_i and f_i covers all hyperedges. If E_i and F_i are the relations that induced the hyperedges e_i and f_i, then the semi-join program $\langle E_1 := E_1 \ltimes F_1, \ldots, E_m := E_m \ltimes F_m, F_m := F_m \ltimes E_m, \ldots, F_1 := F_1 \ltimes E_1\rangle$ is a full reducer for Q. After computing the full reducer, we can compute $Q := F_m \bowtie E_m \bowtie \ldots \bowtie E_1$, guaranteeing that *no intermediate relation is larger than* $\|Q(\mathcal{D})\|$. The evaluation of the full reducer plus the subsequent sequence of joins takes time $O(n(I \log I + U \log U))$ (Ullman 1989), i.e., it is output-polynomial. We call the algorithm (computing and evaluating a full reducer and computing a full join program) the *fully-reduced join algorithm.*

Yannakakis' Algorithm If not all variables are distinguished, the fully-reduced join algorithm is no longer guaranteed to be output-polynomial. In such cases, *Yannakakis' algorithm* (1981) has better asymptotic guarantees, as it interleaves the joins with projections. It starts by computing and executing a full reducer as explained above, and then constructs a *parse tree* $\mathcal{P}(Q)$ of the query where every atom $R_i(\vec{t}_i)$ is a node and node $R_i(\vec{t}_i)$ is a child of $R_j(\vec{t}_j)$ iff the hyperedge corresponding to $R_i(\vec{t}_i)$ was removed in favor of the hyperedge corresponding to $R_j(\vec{t}_j)$ in the GYO algorithm. The algorithm then generates a query program by traversing $\mathcal{P}(Q)$ bottom-up and appending $R'_j := \pi_{\vec{t}_j \cup (\vec{t}_i \cap \vec{x})}(R'_i \bowtie R'_j)$ to the program each time it visits a relation $R_i(\vec{t}_i)$ with parent $R_j(\vec{t}_j)$. Once the tree traversal reaches the root node $R_i(\vec{t}_i)$ of $\mathcal{P}(Q)$, it appends the final assignment $Q := \pi_{\vec{x}}(R'_i)$, which projects the root node to the distinguished attributes. While traversing the parse tree, all intermediate relations have size bounded by $O(IU)$ (Yannakakis 1981; Ullman 1989). Furthermore, the algorithm has runtime $O((I + U)^2)$ and is thus output-polynomial.

A Database Perspective of Classical Planning

Given a planning task $\Pi = \langle \mathcal{P}, O, \mathcal{A}, s_0, \gamma\rangle$, we consider a state s to be a database $\mathcal{D}(s) = \langle O, \{R_{P,s} \mid P \in \mathcal{P}\}\rangle$ where the objects of Π form the domain and there is one relation for every predicate. The relation $R_{P,s}$ contains all tuples for which the corresponding ground atom of P is in s, i.e.,

$$R_{P,s} = \{\vec{o} \mid P(\vec{o}) \in s\}.$$

Each state can be seen as such a database and thus finding the set of applicable actions for a state can be expressed as a query. Let $a[\Delta] \in \mathcal{A}$ be an action schema with precondition $\{P_1(\vec{t}_1), \ldots, P_n(\vec{t}_n)\}$. The set of applicable ground actions for $a[\Delta]$ are the ones that are grounded with object tuples the following conjunctive query over $\mathcal{D}(s)$ which we call the precondition query of $a[\Delta]$:

$$Q(\Delta) \coloneq R_{P_1,s}(\vec{t}_1), \ldots, R_{P_n,s}(\vec{t}_n)$$

To illustrate, consider the standard formulation of the Gripper domain, where a robot with two grippers must move some balls from one room to another. Let s be the state where the robot and two balls are in room r_A and a third ball is in gripper g_1. The database $\mathcal{D}(s)$ contains the relations $R_{\text{at},s} = \{\langle \mathrm{b}_1, \mathrm{r}_\mathrm{A}\rangle, \langle \mathrm{b}_2, \mathrm{r}_\mathrm{A}\rangle\}$, $R_{\text{carry}} = \{\langle \mathrm{b}_3, \mathrm{g}_1\rangle\}$, $R_{\text{at-robby},s} = \{\langle \mathrm{r}_\mathrm{A}\rangle\}$, and $R_{\text{free},s} = \{\langle \mathrm{g}_2\rangle\}$. Action schema *pick*$[b, r, g]$ models picking up ball b in room r with robot gripper g, and has preconditions $\{\text{at}(b, r), \text{at-robby}(r), \text{free}(g)\}$. Its applicable instantiations are described by the conjunctive query $Q(b, r, g) \;\text{:-}\; R_{\text{at},s}(b, r), R_{\text{at-robby},s}(r), R_{\text{free},s}(g)$. The query evaluates to $\{\langle \mathrm{b}_1, \mathrm{r}_\mathrm{A}, \mathrm{g}_2\rangle, \langle \mathrm{b}_2, \mathrm{r}_\mathrm{A}, \mathrm{g}_2\rangle\}$ showing that in s the two applicable instantiations are to pick up b_1 or b_2 with gripper g_2.

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

Acyclic Action Schemas

Precondition queries of acyclic action schemas can be evaluated efficiently with the fully-reduced join algorithm, which in this case is polynomial in the size of the state and the number of applicable instantiations.

Consider now an action schema $a[\Delta]$ with $|\Delta| = n$ free variables where only one $v \in \Delta$ appears in the add or delete list while all others only occur in the precondition. If there are m objects the number of applicable grounded actions for $a[\Delta]$ can be m^n. However, there can be at most m different effects a grounded action can have as the choices for all variables except v have no effect on the successor. We say that the $n - 1$ variables different from v are *existentially quantified*.[2] We do not care about which objects are grounding these variables, we only care for which value of v objects exist that satisfy the preconditions.

Instead of mentioning all parameters of the action in the query's head as distinguished variables, we only mention those that occur in the effect. Existential quantification of action parameters directly corresponds to the existential quantification in conjunctive queries. As an example, consider the action *introduce*$[a, b, c]$ with the precondition $\{\textit{knows}(a, b), \textit{knows}(b, c)\}$ and the add effect $\{\textit{knows}(a, c)\}$. If two people have more than one common friend that can introduce them, there will be several ways to introduce them to each other – all with the same effect. The query $Q(a, b, c) \;\text{:-}\; \textit{knows}(a, b), \textit{knows}(b, c)$ contains them all while the query $Q(a, c) \;\text{:-}\; \textit{knows}(a, b), \textit{knows}(b, c)$ contains at most one tuple for each choice of a and c.

Precondition queries for schemas with existentially quantified variables can be evaluated with Yannakakis' algorithm. As we discussed earlier, this algorithm is quadratic in the size of the input I and the output U. Compared to the fully-reduced algorithm this does not sound like an improvement. However, the output in this case is slightly different. The output here has the different tuples only considering the distinguished attributes, instead of all attributes. This can be significant smaller in schemas where there are many existentially quantified free variables.

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

[2]PDDL fragments more general than STRIPS allow explicit existential quantification. While our algorithm would also cover this, here we only consider implicit quantification in STRIPS where parameters are only used in the precondition.

Cyclic Action Schemas

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

We consider only a static strategy based on a partial execution of a full reducer. We first obtain a semi-join program computed from the GYO reduction of the precondition query hypergraph. Since the query in this case is cyclic, the semi-join program does not correspond to a full reducer. It can, however, be seen as a "partial reducer" that still filters out unnecessary tuples of some relations. Whenever we want to instantiate a cyclic action schema, we first evaluate this "partial reducer" and then compute a complete join program of the precondition atoms ordered by increasing arity with ties broken according to the order of predicates in the input.

Experimental Results

We implemented a lifted planner using the successor generator methods previously described. The source code is publicly available (Corrêa et al. 2020). Our planner supports the PDDL fragment representing STRIPS with equalities, inequalities, and types. It first compiles equalities and types into static predicates. Schemas with parameters of an empty type are removed. We handle inequality constraints in all our query programs by removing tuples that violate a constraint after every join. There is a more sophisticated algorithm for acyclic conjunctive queries with inequalities that is fixed-parameter tractable in the size of the query and the number of variables (Papadimitriou and Yannakakis 1999) and could improve the results. We consider this future work.

All experiments were run on an Intel Xeon Silver 4114 processor running at 2.2 GHz with maximum runtime of 30 minutes and a maximum memory of 16 GiB.

We use 59 domains divided into two sets. The first has 53 domains from the nine International Planning Competitions (1998 to 2018) that our planner supports. If a domain was used in an optimal and a satisficing track, we report the results from the optimal track. In total, the set has 1560 instances. We also tested the domains from the satisficing track, with very similar results to the optimal track (Corrêa 2019, Section 5.5). Our focus is on hard-to-ground problems, so we also use six hard-to-ground domains:

- *Organic Synthesis*: The entire set contains 56 instances and is split into three domains "Original" (20), "MIT" (18), and "Alkene" (18). [3] This domain was also used in the IPC, but only a subset of the easiest instances was used there. The IPC set also contains a variant using an

[3]http://www.cs.ryerson.ca/~mes/publications/. This set of instances was designed by Russell Viirre and converted into PDDL by Hadi Qovaizi.

action schema split method (Areces et al. 2014), which is considerably easier to ground.

- *Pipesworld-Tankage*: This is a variant of the domain with the same name introduced in IPC 2004 (Hoffmann et al. 2006). The IPC version used manually split action schemas to simplify the grounding. Here we use the considerably harder to ground non-split version. There are 50 instances in this domain.
- *Genome Edit Distance*: This domain was introduced by Haslum (2011) as a challenging practical application of planning. The domain was introduced in several different formulations. We only use the two relational multi-step formulations (split and non-split) because the others use PDDL features not supported by our planner. Each formulation has 156 instances.

We call the first set of benchmarks the *IPC set*, the second one the *HTG set* and the union of the two the *full set*. Whenever we mention the Organic Synthesis domain of the IPC set, we refer to its non-split variant.

We first analyze different successor generator techniques using a breadth-first search algorithm considering all action schemas as unit cost.

Naive Joins

To establish a baseline, we first use the naive join program to evaluate precondition queries. We use three variants that differ in the order in which they join the relations. Variant J uses the predicate order given by the action schema; variant J^R uses a random order; and variant $J^<$ orders the relations by increasing arity breaking ties according to the order of J. We use all of these variants in a breadth-first search and evaluate on the full benchmark set. Results for J^R are averaged over three runs.

We first report the results for the IPC set. Using the predicate order from the input (J) surprisingly performed the best, solving 454 tasks. Ordering the relations by arity ($J^<$) performed similarly with a coverage of 443, while a random order (J^R) led to worse results of 350–357 tasks solved (average 352.3). This difference in coverage shows the large impact of join order on performance and that both orders have a positive effect. The most interesting result in the IPC set is the Organic Synthesis domain. Here, J^R was only able to solve 2 instances while $J^<$ solves 10 and J solves 11. Using the settings of the IPC 2018, limiting memory to 8 GiB, J still solves 10 instances. This is a significant result since no planner in the IPC 2018 solved more than 8 instances of this domain optimally. This shows that database techniques can enhance the performance in this domain, although it is still necessary to consider some information about the structure of the precondition when instantiating it.

In the HTG set, J also achieves the highest coverage, with 83 instances, while $J^<$ solves 78 and J^R solves 56–58 (average 57.3). In the original version of the Organic Synthesis domain, no method solves a single instance. All methods run out of memory when trying to instantiate action schemas. As expected, the naive joins produce very large intermediate results that exhaust memory. Although the database techniques enhance the performance in the IPC version of Organic Synthesis, there is still room for improvement.

	$\|A\|$	**Acyc.**	**Acyc.** $\neq$	$\exists$**-quant.**
Barman (2011, 2014)	12	91.7%	91.7%	83.3%
Elevators (2008, 2011)	6	16.7%	16.7%	0.0%
Freecell	10	70.0%	70.0%	30.0%
GED	21	71.4%	100.0%	0.0%
Hiking	7	57.1%	85.7%	42.9%
NoMystery	3	66.7%	66.7%	33.3%
Org. Synt.	760	8.5%	91.4%	91.4%
Org. Synt. split	17564	90.2%	100.0%	55.9%
Pipesw. NoTank.	6	33.3%	33.3%	100.0%
Pipesw. Tank. split	10	10.0%	10.0%	100.0%
Rovers	9	88.9%	88.9%	66.7%
TPP	4	75.0%	75.0%	50.0%
Others (39 domains)		100.0%	100.0%	22.22%
IPC Domains		87.8%	90.7%	30.4%
Org. Synt. Original	52	9.6%	90.3%	90.3%
Org. Synt. MIT	52	9.6%	90.3%	90.3%
Org. Synt. Alkene	12	0.0%	100.0%	100.0%
GED Multi-step	14	35.7%	100.0%	0.0%
GED Multi-step, split	21	71.4%	100.0%	0.0%
Pipesworld Tankage	4	0.0%	0.0%	100.0%
HTG Domains		21.0%	80.1%	63.5%

Table 1: Proportions of action schemas in A that are acyclic (Acyc.), acyclic when ignoring inequalities (Acyc. $\neq$), or that have existentially quantified parameters ($\exists$-quant.). Domains where all actions are acyclic are grouped together. Lines for multiple domains are averaged over domains.

Acyclic Schemas

The naive join methods fail in cases where the intermediate relations become too large. For acyclic action schemas this issue can be avoided by computing a full reducer. But how many of the action schemas are acyclic? The second column in Table 1 shows the proportion of acyclic schemas in each domain. In the IPC set the average proportion of acyclic schemas is 87.8%. This is good news for our method, since it means that grounding these schemas can be done efficiently in all states. Unfortunately the situation looks worse in the hard-to-ground instances, where the average proportion of acyclic schemas is only 21%. In particular, the challenging Organic Synthesis domain shows up in both benchmark sets with less than 10% acyclic schemas. Looking closer at the hypergraph of the precondition queries of these schemas, it turns out that most of the cycles are caused by inequality constraints. If we ignore inequalities, the incidence of acyclic schemas increases to over 90% in all Organic Synthesis variants and to 80.1% in all hard-to-ground domains (shown in the third column of Table 1). The notable exception is the domain Pipesworld-Tankage where all action schemas are cyclic even when ignoring inequality relations.

To analyze the effect of evaluating acyclic precondition queries efficiently, we ran our breadth-first search with the configuration $FR^{SJ,<}$, which uses the GYO algorithm in a preprocessing step to test which queries are cyclic. As a side effect, this step finds a full reducer for acyclic schemas and a partial reducer for cyclic ones. During the search, all queries first execute this reducer. For acyclic schemas the algorithm then executes the join program in the order established by the hyperedge removals. For cyclic schemas, the final join program is $J^<$. Even though J performed better than $J^<$ we use $J^<$ here because it is more principled.

Compared to $J^<$, using $FR^{SJ,<}$ to generate successors improves the coverage from 443 to 464 in the IPC set and from 78 to 99 in the HTG set. The largest improvement is in the non-split IPC version of Organic Synthesis, where $FR^{SJ,<}$ solves 19 of the 20 instances. All tasks solved by $FR^{SJ,<}$ in this domain used 1.2 GiB or less while the IPC 2018 planners could solve only 8 tasks with a limit of 8 GiB.

The main advantage of using the full reducer is that it avoids large intermediate relations. Trying to estimate how often this occurs, we compared the largest intermediate relation size during the expansion of the initial state for $J^<$ and $FR^{SJ,<}$. In instances with only acyclic actions, the largest intermediate relation computed by $J^<$ is sometimes two order of magnitudes larger than the one computed by $FR^{SJ,<}$. In instances with cyclic schemas, the initial partial reducer semi-join program used by $FR^{SJ,<}$ also seems to help. In larger tasks of the Organic Synthesis domain variants, $J^<$ ran out-of-memory, while $FR^{SJ,<}$ could still instantiate all schemas. There were also 27 tasks where $J^<$ still can instantiate all schemas, but $FR^{SJ,<}$ produces intermediate relations 2–5 orders of magnitude smaller. There is no theoretical guarantee that $FR^{SJ,<}$ performs better than $J^<$ in cyclic schemas, but this happens more often than not in our benchmarks. However, even with $FR^{SJ,<}$ the cyclic schemas require more tuples than the acyclic ones in most cases, so future work on better join orders might pay off.

When comparing total search time, $FR^{SJ,<}$ has similar runtime to J and $J^<$. Although one might expect that the $FR^{SJ,<}$ method would be slower due to the additional overhead of evaluating a semi-join program prior to the full join program, the method is competitive on all instances. In the Organic Synthesis domain, the breadth-first search is faster with $FR^{SJ,<}$ than with the other methods.

Existentially Quantified Preconditions

We also discussed a successor generator based on Yannakakis' algorithm that has a potential gain for instances with existentially quantified parameters. For example, the Organic Synthesis domain has action schemas with 17 parameters where only four appear in the effect. Some domains such as Sokoban and Pipesworld have at least one existentially quantified variable in all their schemas (see Table 1). On average, the proportion of action schemas with one or more existentially quantified variables per domains is 29.4% and such actions occur in 35 of our 59 domains.

As there are many instances with a potential gain, we implemented and evaluated a successor generator Y that uses Yannakakis' algorithm to create only one grounded action for each choice of distinguished variables. For cyclic action schemas Y uses the same fall back strategy as $FR^{SJ,<}$. Surprisingly, using Y in a breadth-first search is slightly worse than using $FR^{SJ,<}$. It solves three tasks less on the IPC set and one less on the HTG set.

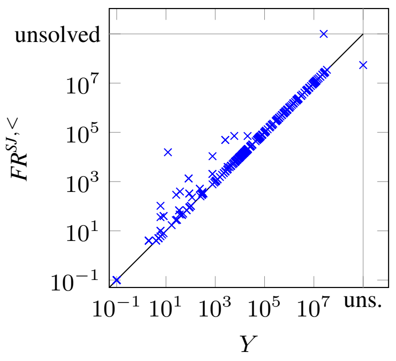

Figure 1: Comparison of the number of generated states before the last layer with breadth-first search. Showing only instances with at least one action schema with existential quantified variables.

Although several instances have potential savings in the number of generated states using Y, we did not see an improvement in our experiments. Figure 1 compares the number of generated states before the last layer using Y and $FR^{SJ,<}$ in a breadth-first search. We only show instances where at least one action schema has a variable that can be existentially quantified. Most of the instances where Y reduces the number of generations are in the different Organic Synthesis versions (IPC split and non-split and the versions in the hard-to-ground set).

We also do not see much gain in time and memory. The only domain where there is a reduction in peak memory is Organic Synthesis, where Y significantly reduces the number of generated successors. However, in general, Y has an additional overhead compared to $FR^{SJ,<}$ and thus the time saved by avoiding duplicated states usually does not pay off.

Comparison to Grounded Planning

We also compare the performance of our two best methods, $FR^{SJ,<}$ and Y, to a state-of-the-art grounded planner. We first compare our methods to the breadth-first search implemented in Fast Downward 19.06 (Helmert 2006).

The IPC set consists mostly of tasks that are easy to ground, so we would expect a grounded planner to perform better. Indeed, Fast Downward solves 638 instances, while $FR^{SJ,<}$ solves 464. Still, the overhead of grounding action schemas in each state is manageable and the coverage of $FR^{SJ,<}$ is within 5 tasks of the coverage of Fast Downward in 42 of the 53 domains. The notable exception in the IPC set is of course the hard-to-ground domain Organic Synthesis. As we saw in our previous experiment, $FR^{SJ,<}$ and Y solve 19 of the 20 IPC tasks. In fact, Y needs a total of 10.5 seconds to solve all these 19 instances. Fast Downward solves only 8 and fails to ground the remaining instances. This indicates that faster search techniques or better heuristics would not make a difference in these cases.

On hard-to-ground instances, we would expect our methods to perform better. The first block of Table 2 shows coverage results for Fast Downward's and our blind search methods on the HTG set. We see that the gap in Organic Syn-

		BFS			GBFS				
	# of Inst.	FD	$FR^{SJ,<}$	Y	FD	$FR^{SJ,<}$	Y	LAMA	L-RPG
Organic Synthesis	56	20	**44**	**44**	20	47	**50**	20	14
Original	20	1	**8**	**8**	1	11	**14**	1	0
MIT	18	2	**18**	**18**	2	**18**	**18**	2	0
Alkene	18	17	**18**	**18**	17	**18**	**18**	17	14
Genome Edit Distance	312	**46**	44	44	**312**	**312**	**312**	**312**	113
Multi-step	156	**24**	22	22	**156**	**156**	**156**	**156**	48
Multi-step, split	156	**22**	**22**	**22**	**156**	**156**	**156**	**156**	65
Pipesworld-Tankage	50	**14**	11	10	20	**22**	**22**	18	10
Total	418	80	**99**	98	352	381	**384**	350	137

Table 2: Coverage for the HTG set. 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). We also compare them to the L-RPG and LAMA planners. For each search configuration, the best method in each domain is highlighted.

thesis widens with larger instances that $FR^{SJ,<}$ and Y can still solve but Fast Downward fails to ground. For the domains Genome Edit Distance and Pipesworld-Tankage, Fast Downward still solves more tasks, although the difference to our lifted methods is smaller. In these large instances we expect Fast Downward's main bottleneck to be the grounding while the main bottleneck of our methods is the search. We thus expect that using an informed search with a heuristic function will change the results.

To test this hypothesis, we 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 unsatisfied in the state being evaluated. The second block of Table 2 shows how a greedy best-first search with this heuristic performs in Fast Downward, $FR^{SJ,<}$ and Y.

We can see that the heuristic indeed had a larger benefit for lifted methods which now dominate the results of Fast Downward. With the heuristic, the search expands fewer states which means the overhead of computing the successor states is less relevant. A grounded planner on the other hand requires the same amount of effort to ground the task. This can be seen in the Organic Synthesis domain, where all tasks that Fast Downward fails to solve run out of memory. The same is true for the Pipesworld domain, where our methods solve 4 tasks that Fast Downward is not able to ground.

On the HTG set, our methods have faster runtimes even for the instances which Fast Downward can ground. Figure 2 shows the total time spent by Y and by Fast Downward's GBFS with the goal-count heuristic. The lifted method is faster than Fast Downward in most of the tasks. In particular in the domain Genome Edit Distance (no-split), the grounding process takes too long and dominates the total time. This plot also shows that the split versions end up reducing some of the benefits of our methods. When comparing peak memory, the same behavior is observed. Our planner has a lower peak memory than Fast Downward in almost all tasks. The tasks where Fast Downward used more memory are the ones where it also needed more time. In fact, in these instances,

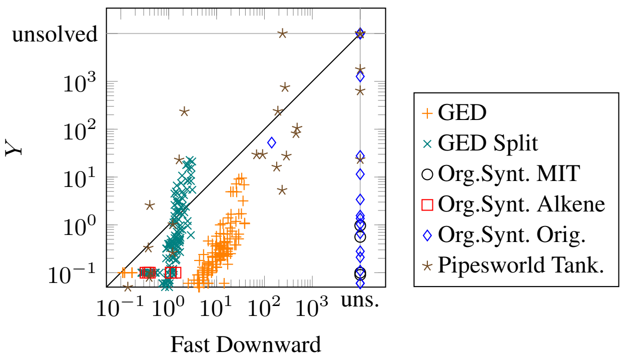

Figure 2: Total time (in seconds) for Y and Fast Downward with GBFS and the goal-count heuristic for the HTG set. The time computed for Fast Downward is the sum of the grounding and the search procedures.

both the runtime 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 grounding is the main bottleneck. If these tasks were scaled up, Fast Downward would probably not be able to finish their grounding, while our lifted planner would still have a chance to solve them.

Using the heuristic also helps on the IPC instances where the gap between Fast Downward shrinks from 174 tasks to 163. Fast Downward's GBFS solves 1213 tasks while both $FR^{SJ,<}$ and Y solves 1050 tasks when equipped with the goal-count heuristic. While the effect is smaller on these benchmarks, we still see that the heuristic had more benefit in the lifted than in the grounded planner. Better heuristics could make lifted planners competitive on these domains.

Solving Hard-to-Ground Domains

To get a better understanding of our methods in the hard-to-ground domains, we also compared them to complete planning systems. We compared our methods to LAMA (Richter and Westphal 2010) and to L-RPG, the state-of-the-art plan-

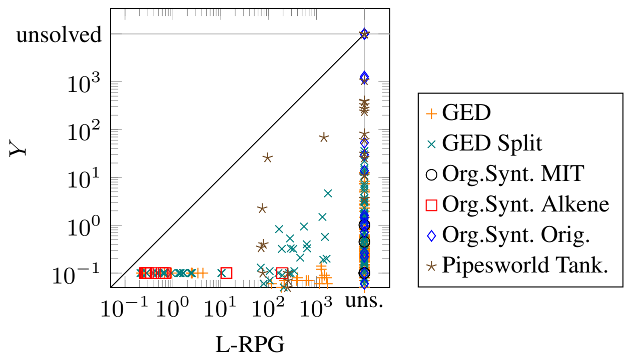

Figure 3: Total time (in seconds) for Y and L-RPG with GBFS and the goal-count heuristic for the HTG set.

ner using lifted representations (Ridder 2013). The last two columns of Table 2 show the coverage results of LAMA and L-RPG. As we can see, LAMA has a very similar coverage as a GBFS with goal-count in Fast Downward. This reinforces the result that the only bottleneck of these domains is the grounding and not the search itself. Adding a powerful search method to a grounded planner does not improve the coverage in these domains, simply because the search itself is not the most challenging part. The time and memory usage of LAMA is similar to GBFS with goal-count: both time and memory consumption are dominated by the translator component because of the grounding.

The comparison to the L-RPG planner is also very favorable to our methods. L-RPG is a lifted planner that also uses lifted heuristics. The best configuration of L-RPG uses a lifted version of the FF heuristic (Hoffmann and Nebel 2001; Ridder 2013). Additionally, L-RPG computes equivalence relations between objects to find symmetries. The total coverage of L-RPG is 137, while Y solves 384 tasks. L-RPG does not use sophisticated techniques to generate successors and thus this is the main bottleneck of the planner. These results show that for these hard-to-ground domains the successor generation is one of the main bottlenecks for lifted planners. Our methods are also faster than L-RPG in all instances. Figure 3 compares the total time of L-RPG and Y. In the smallest instances, the preprocessing to compute equivalence relations takes more time than our translations but does not pay off. In larger instances, our methods are faster than L-RPG in spite of a less informed heuristic. This could be due to L-RPG spending too much time instantiating action schemas or to the lifted FF computation of L-RPG being too expensive in these domains.

Related Work

Planning techniques that do not depend on the ground representation of the problem have long existed, although recent research has focused on grounded planning. For instance, some planning-as-satisfiability approaches use encodings that avoid the need for grounding all actions at preprocessing by using propositions representing the actual grounding of the action executed at each time step (Kautz and Selman 1992; Kautz, McAllester, and Selman 1996; Robinson et al. 2008). Similar encodings have been proposed in recent compilations of numeric and temporal planning to SMT or CSP (Bofill, Espasa, and Villaret 2016; Espasa et al. 2019; Bit-Monnot 2018). Other approaches to planning, such as partial order planning, have also explored the use of lifted instead of ground actions (Penberthy and Weld 1992; Younes and Simmons 2002; 2003).

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

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

Areces et al. (2014) develop an automatic action schema splitting technique that reduces the number of parameters of action schemas, at the cost of modifying the state space topology. Since theirs is a model reformulation approach, it has the advantage that it can be coupled with any planner. Gnad et al. (2019) present a machine-learning method that incrementally grounds larger and larger parts of the full set of ground actions until a plan can be found. Lifted approaches have also been considered for other planning-related tasks such as the computation of problem invariants and symmetries (Rintanen 2017; Röger, Sievers, and Katz 2018; Sievers et al. 2019; Fišer 2020).

Moving beyond the planning literature, there exists a wealth of related work in the database field that goes beyond the techniques used in this paper. For example, Gottlob, Leone, and Scarcello (2002) discuss generalizations of query acyclicity based on hypertree-width that could be leveraged in future work. It is also possible to approach the successor generation problem from other perspectives that are strongly related to database theory, such as constraint satisfaction (Gottlob, Leone, and Scarcello 2000; Vardi 2000; Dechter 2003). The Rete pattern matching algorithm (and associated data structures) by Forgy (1982) also addresses closely related problems.

Conclusions

We have shown how to efficiently perform lifted successor generation in classical planning using well-known database techniques. The problem of generating all ground actions that derive from an action schema and are applicable in a given state is equivalent to evaluating a query given by the schema precondition in a database given by the state. In many cases, the action preconditions of standard planning domains fall into the tractable case of acyclic conjunctive queries. We experimentally evaluate different query optimization techniques for both the acyclic and cyclic cases. Our results show that this approach has an acceptable overhead in most standard benchmarks, and is a preferable alternative to state-of-the-art grounded planners in domains that have traditionally been excluded from the International Planning Competitions because they are too hard to ground.

There are many possible future directions for this work. Firstly, it would be interesting to cover more expressive modeling languages supporting features such as conditional effects and axioms. Secondly, more efficiently dealing with actions with many parameters makes it possible to revisit some of the efficiency trade-offs in algorithms that make use of macro actions (Botea et al. 2005). Finally, the techniques developed in this paper could be combined with lifted heuristics (Ridder 2013) or other recent planning techniques that do not require a set of ground actions (Francès et al. 2017).

Acknowledgments

This work was supported by the Swiss National Science Foundation (SNSF) as part of the project “Certified Correctness and Guaranteed Performance for Domain-Independent Planning” (CCGP-Plan). Guillem Francès is funded by grant TIN-2015-67959-P, MEC, Spain. We thank Bram Ridder for his assistance on the usage of the L-RPG planner and Mikhail Soutchanski for his assistance with the Organic Synthesis domain.

References

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

Areces, C.; Bustos, F.; Dominguez, M. A.; and Hoffmann, J. 2014. Optimizing planning domains by automatic action schema splitting. In *Proc. ICAPS 2014*, 11–19.

Bernstein, P. A., and Goodman, N. 1981. Power of natural semijoins. *SICOMP* 10(4):751–771.

Bit-Monnot, A. 2018. A constraint-based encoding for domain-independent temporal planning. In *Proc. CP 2018*, 30–46.

Bofill, M.; Espasa, J.; and Villaret, M. 2016. The RANTANPLAN planner: system description. *The Knowledge Engineering Review* 31(5):452–464.

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

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. *JAIR* 24:581–621.

Chandra, A. K., and Merlin, P. M. 1977. Optimal implementation of conjunctive queries in relational databases. In *Proc. STOC 1977*, 77–90.

Codd, E. F. 1970. A relational model of data for large shared data banks. *Communications of the ACM* 13(6):377–387.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Code from the paper “Lifted Successor Generation using Query Optimization Techniques”. https://doi.org/10.5281/zenodo.3687008.

Corrêa, A. B. 2019. Planning using lifted task representations. Master’s thesis, University of Basel.

Creignou, N.; Khanna, S.; and Sudan, M. 2001. *Complexity Classifications of Boolean Constraint Satisfaction Problems*, volume 7 of *SIAM Monographs on Discrete Mathematics and Applications*. SIAM.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Espasa, J.; Coll, J.; Miguel, I.; and Villaret, M. 2019. Towards lifted encodings for numeric planning in Essence Prime. In *CP 2019 Workshop on Constraint Modelling and Reformulation*.

Fagin, R. 1983. Acyclic database schemes (of various degrees): A painless introduction. In *Colloquium on Trees in Algebra and Programming*, 65–89.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *AIJ* 2:189–208.

Fišer, D. 2020. Lifted fact-alternating mutex groups and pruned grounding of classical planning problems. In *Proc. AAAI 2020*.

Forgy, C. 1982. Rete: A fast algorithm for the many patterns/many objects match problem. *AIJ* 19(1):17–37.

Francès, G.; Ramírez, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely declarative action representations are overrated: Classical planning with simulators. In *Proc. IJCAI 2017*, 4294–4301.

Gnad, D.; Torralba, Á.; Domínguez, M. A.; Areces, C.; and Bustos, F. 2019. Learning how to ground a plan – Partial grounding in classical planning. In *Proc. AAAI 2019*, 7602–7609.

Gottlob, G.; Leone, N.; and Scarcello, F. 2000. A comparison of structural CSP decomposition methods. *AIJ* 124(2):243–282.

Gottlob, G.; Leone, N.; and Scarcello, F. 2002. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences* 64(3):579–627.

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

Haslum, P. 2011. Computing genome edit distances using domain-independent planning. In *ICAPS 2011 Scheduling and Planning Applications woRKshop*, 45–51.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *AIJ* 173:503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Hoffmann, J.; Edelkamp, S.; Thiébaux, S.; Englert, R.; dos Santos Liporace, F.; and Trüg, S. 2006. Engineering benchmarks for planning: the domains used in the deterministic part of IPC-4. *JAIR* 26:453–541.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In *Proc. ECAI 1992*, 359–363.

Kautz, H.; McAllester, D.; and Selman, B. 1996. Encoding plans in propositional logic. In *Proc. KR 1996*, 374–384.

Koller, A., and Petrick, R. 2011. Experiences with planning for natural language generation. *Computational Intelligence* 27(1):23–40.

Matloob, R., and Soutchanski, M. 2016. Exploring organic synthesis with state-of-the-art planning techniques. In *ICAPS 2016 Scheduling and Planning Applications woRKshop*, 52–61.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language – Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University.

McDermott, D. 1996. A heuristic estimator for means-ends analysis in planning. In *Proc. AIPS 1996*, 142–149.

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

Pednault, E. P. D. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. KR 1989*, 324–332.

Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. KR 1992*, 103–114.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR* 39:127–177.

Ridder, B. 2013. *Lifted Heuristics: Towards More Scalable Planning Systems*. Ph.D. Dissertation, King's College London.

Rintanen, J. 2017. Schematic invariants by reduction to ground invariants. In *Proc. AAAI 2017*, 3644–3650.

Robinson, N.; Gretton, C.; Pham, D. N.; and Sattar, A. 2008. A compact and efficient SAT encoding for planning. In *Proc. ICAPS 2008*, 296–303.

Röger, G.; Sievers, S.; and Katz, M. 2018. Symmetry-based task reduction for relaxed reachability analysis. In *Proc. ICAPS 2018*, 208–217.

Sievers, S.; Röger, G.; Wehrle, M.; and Katz, M. 2019. Theoretical foundations for structural symmetries of lifted PDDL tasks. In *Proc. ICAPS 2019*, 446–454.

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient symbolic search for cost-optimal planning. *AIJ* 242:52–79.

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

Vardi, M. Y. 2000. Constraint satisfaction and database theory: a tutorial. In *Proc. PODS 2000*, 76–85.

Yannakakis, M. 1981. Algorithms for acyclic database schemes. In *Proc. VLDB 1981*, 82–94.

Younes, H. L. S., and Simmons, R. G. 2002. On the role of ground actions in refinement planning. In *Proc. AIPS 2002*, 54–62.

Younes, H. L. S., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *JAIR* 20:405–430.