Lifted Planning: Recent Advances in Planning Using First-Order Representations

Augusto B. Corrêa¹, Giuseppe De Giacomo²

¹University of Basel, Switzerland
²University of Oxford, England, UK
augusto.blaascorrea@unibas.ch, giuseppe.degiacomo@cs.ox.ac.uk

Abstract

Lifted planning is usually defined as planning directly over a first-order representation. From the mid-1990s until the late 2010s, lifted planning was sidelined, as most of the state-of-the-art planners first ground the task and then solve it using a propositional representation. Moreover, it was unclear whether lifted planners could scale. But as planning problems become harder, they also become infeasible to ground. Recently, lifted planners came back into play, aiming at problems where grounding is a bottleneck. In this work, we survey recent advances in lifted planning. The main techniques rely either on state-space search or logic satisfiability. For lifted search-based planners, we show the direct connections to other areas of computer science, such as constraint satisfaction problems and databases. For lifted planners based on satisfiability, the advances in modeling are crucial to their scalability. We briefly describe the main planners available in the literature and their techniques.

1 Introduction

A planning problem consists of the following: an initial state, a set of actions, and a goal, all formally specified using some logic vocabulary (predicates, objects, and perhaps functions). The objective of a planner is to synthesize a strategy/program – called a plan – to fulfill the goal starting from the initial state. A particular flavor of planning is classical planning, where actions are deterministic and states are fully observable. In this case, a plan is a sequence of actions. In this survey, we focus on classical planning.

There are different ways to represent planning problems, depending on the logic vocabulary [McDermott et al., 1998; Pednault, 1989; Bäckström and Nebel, 1995]. In particular, first-order (FO) languages have been advocated as a convenient way to represent planning problems [Newell and Simon, 1963; Green, 1969b; Fikes and Nilsson, 1971; Lifschitz, 1987; Levesque, 1996; Reiter, 2001; Levesque, 2005]. When finitely many objects and no functions are used, this FO representation becomes a compact way to represent an exponentially larger propositional representation, obtained by grounding, i.e., instantiating all predicates with the available objects.

In the 1990s, Kautz and Selman [1992] showed that using propositional satisfiability (SAT) to solve planning problems was an effective technique. SAT planning translates the planning task into a SAT formula and uses a SAT solver to find a model. This encoding bounds the maximum length of a plan, so if no plan is found with the assumed length, the planner iteratively increases this bound and produces a new (and longer) formula, until a model is found. When this happens, the model is converted into a plan. SAT planning disregards any FO structure of the problem, and instead works directly with the grounded propositional representation.

Although the supremacy of SAT planners did not last forever, most of the following works still used propositional representations. The dominant paradigm in planning in the last decades was state-space search [Bonet and Geffner, 2001]: the planner starts a search from the initial state, expanding promising successor states in some order, and stops when the goal is reached or all reachable states have been explored. The search is guided according to a heuristic to avoid expanding the entire state space [Hoffmann and Nebel, 2001; Helmert, 2006; Torralba et al., 2014; Francès et al., 2018; Seipp, 2023].

Despite this dominance of propositional representations, recently a new class of so-called lifted planners has emerged. These planners use FO representations, as in most of the early work in planning [Newell and Simon, 1963; McCarthy, 1963; Green, 1969a; Fikes and Nilsson, 1971; Bibel, 1986; Levesque, 1996; Reiter, 2001]. However, they are not based on forms of logical implication but on grounding on-demand while performing a state-space search. This new generation of planners is competitive with the state-of-the-art propositional ones, and can also deal with a number of objects for which grounding is prohibitive.

In this chapter, we survey recent lifted planning techniques for classical planning. These improvements have direct connections to logic programming, constraint satisfaction problems, and database theory. We show the two main techniques used by modern lifted planners, search and satisfiability.

Due to the space limits, we do not go into great detail for every technique, but we focus on those that have closer relation to other communities.
2 Lifted Planning

Throughout the paper we assume familiarity with basic concepts of first-order logic and classical planning (e.g., heuristic search, optimal vs satisficing planning). We assume that planning languages use a function-free logical vocabulary over an infinite set of variables \( \mathcal{V} \), a finite set of constants \( \mathcal{C} \), and a set of predicate symbols \( \mathcal{P} \). An atom \( p(T) \) is composed of a predicate symbol \( p \in \mathcal{P} \) and a \( k \)-tuple of terms \( T \) (variables or constants), where \( k \) is the arity of \( p \). The set of variables in \( T \) is denoted by \( \text{vars}(T) \). We say that \( p(T) \) is a ground atom if \( \text{vars}(T) = \emptyset \). In the rest of the paper, variables are denoted by uppercase letters (\( X, Y, C_1 \)); constants are denoted by lowercase italic letters (\( a, b, t \)); predicate symbols are denoted by lowercase upright names (\( p(a, t) \)).

For simplicity, we focus on STRIPS planning tasks [Fikes and Nilsson, 1971]. This is also the minimal fragment that all planners listed later support. Most of the techniques can be extended to more expressive formalisms.

A lifted planning task is a tuple \( \Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, I, G \rangle \) where the sets \( \mathcal{P} \) and \( \mathcal{C} \) are the predicate symbols and constants of our first-order language, \( \mathcal{A} \) is the set of action schemas, \( I \) is the initial state, and \( G \) is the goal.

An action schema \( A \in \mathcal{A} \) consists of three sets of atoms:\(^{1}\) a precondition \( \text{pre}(A) \), an add list \( \text{add}(A) \), and a delete list \( \text{del}(A) \). We use \( \text{vars}(A) \) for the set of variables occurring in any atom in one of the three sets. If \( \text{vars}(A) = \emptyset \), we call \( A \) a ground action. It is possible to obtain a ground action from an action schema \( A \) by substituting its variables with constants from \( \mathcal{C} \). A substitution function \( \sigma : \mathcal{V} \rightarrow \mathcal{C} \) applied to \( A \) results in the ground action \( \sigma(A) \), where each variable \( v \in \text{vars}(A) \) is replaced with \( \sigma(v) \) — this is done to all elements in the precondition, add list, and delete list. We sometimes say that \( \sigma(A) \) instantiates the action \( A \).

A state \( s \) is a set of ground atoms (seen as a propositional interpretation). We assume that all states implicitly contain \( c_1 \neq c_2 \) for every pair of distinct constants \( c_1, c_2 \in \mathcal{C} \). A ground action \( \sigma(A) \) is applicable in \( s \) if \( \text{pre}(\sigma(A)) \subseteq s \). Applying action \( A \) in state \( s \) leads to the successor state \( \text{succ}(s, \sigma(A)) = (s \setminus \text{del}(\sigma(A))) \cup \text{add}(\sigma(A)) \). A sequence of actions \( \pi = \langle \sigma_1(A_1), \ldots, \sigma_n(A_n) \rangle \) is applicable in a state \( s_0 \) and has \( \text{succ}(s_0, \pi) = s_n \) if there are states \( s_1, \ldots, s_{n-1} \) where \( \sigma_i(A_i) \) is applicable in \( s_{i-1} \) and \( \text{succ}(s_{i-1}, \sigma_i(A_i)) = s_i \) for all \( i \leq n \).

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

We sometimes also refer to propositional planning: planning over a propositional representation. Both lifted and propositional planners receive a lifted planning task as input, but propositional planners ground the task in advance, producing a large set of ground actions. Lifted planners can always skip this grounding step. When we mention lifted or propositional planning, we refer to the representation only and not a specific technique used to plan.

Throughout our paper, we use the running example introduced next.

Example 1. Consider the simple logistics problem depicted in Figure 1. There are three locations \( a, b \) and \( c \), a truck \( t \) initially at \( a \), and a package \( p \) at \( a \). The package \( p \) must be delivered to location \( c \). The planning task \( \Pi = \langle \mathcal{P}, \mathcal{C}, \mathcal{A}, I, G \rangle \) has the elements

\[
\mathcal{P} = \{\text{at}/2, \text{package-at}/2, \text{conn}/2, \text{loaded}/2\},
\mathcal{C} = \{a, b, c, t, p\},
\mathcal{A} = \{\text{move}(C_1, C_2, T), \text{pick}(C, P, T), \text{drop}(C, P, T)\},
\mathcal{I} = \{\text{conn}(a, b), \text{conn}(b, a), \text{conn}(b, c), \text{conn}(c, b), \text{at}(t, b), \text{package-at}(p, a)\},
\mathcal{G} = \{\text{package-at}(p, c)\}.
\]

The actions have the following preconditions and add/delete lists:

\[
\begin{align*}
\text{pre}(\text{move}(C_1, C_2, T)) &= \{\text{conn}(C_1, C_2), \text{at}(T, C_1)\} \\
\text{add}(\text{move}(C_1, C_2, T)) &= \{\text{at}(T, C_2)\} \\
\text{del}(\text{move}(C_1, C_2, T)) &= \{\text{at}(T, C_1)\} \\
\text{pre}(\text{pick}(C, P, T)) &= \{\text{at}(T, C), \text{package-at}(P, C)\} \\
\text{add}(\text{pick}(C, P, T)) &= \{\text{loaded}(P, T)\} \\
\text{del}(\text{pick}(C, P, T)) &= \{\text{package-at}(P, C)\} \\
\text{pre}(\text{drop}(C, P, T)) &= \{\text{at}(T, C), \text{loaded}(P, T)\} \\
\text{add}(\text{drop}(C, P, T)) &= \{\text{package-at}(P, C)\} \\
\text{del}(\text{drop}(C, P, T)) &= \{\text{loaded}(P)\}.
\end{align*}
\]

The substitution function \( \sigma = \{C_1 \rightarrow b, C_2 \rightarrow a, T \rightarrow t\} \) produces the ground action \( \sigma(\text{move}(C_1, C_2, T)) = \text{move}(b, a, t) \) where

\[
\begin{align*}
\text{pre}(\text{move}(b, a, t)) &= \{\text{conn}(b, a), \text{at}(t, b)\} \\
\text{add}(\text{move}(b, a, t)) &= \{\text{at}(t, a)\} \\
\text{del}(\text{move}(b, a, t)) &= \{\text{at}(t, b)\}.
\end{align*}
\]

As \( \text{pre}(\text{move}(b, a, t)) \subseteq I \), this ground action is applicable in the initial state \( I \). Its application generates the successor state \( \text{succ}(s, \text{move}(b, a, t)) = \{\text{conn}(b, a), \text{conn}(b, a), \text{conn}(b, c), \text{conn}(c, b), \text{at}(t, a), \text{package-at}(p, a)\} \).

A plan for this task is

\[
\pi = \langle \text{move}(b, a, t), \text{pick}(a, p, t), \text{move}(a, b, t), \text{move}(b, c, t), \text{drop}(c, p, t)\rangle.
\]

\(^{1}\)To simplify the formalism, we do not consider action costs.
3 Classical Work on FO Representations

Early work on planning and action theory often focused on first-order representations. In fact, most of the work dealt with logic vocabularies much more powerful than the one we consider here (e.g., with infinitely many objects). Planning on FO representations is not something new and it was the obvious choice for decades. Our discussion below is not exhaustive due to space limitation. We list only the main earlier "paradigms" of planning that used FO representation.

Newell and Simon [1963] presented the General Problem Solver (GPS), which can be seen as a prototypical planning system. GPS could solve problems expressed in FO formulas. It used means-ends analysis to perform a state-space search: given the current state and a goal, the planner performs the action that reduced the difference between the two. For tasks with too many objects, the means-ends analysis of GPS does not scale due to the state explosion.

Situation calculus [McCarthy, 1958; McCarthy, 1963] is also defined on FO representations. It has been used to study reasoning about action, including the famous frame problem [McCarthy and Hayes, 1969]. Perhaps the predominant version of situation calculus nowadays is the formalism by Reiter [2001], which has been applied to planning as well [Levesque, 1996; Reiter, 2001; Levesque, 2005; De Giacomo et al., 2016]. Moreover, previous work also studied the recasting of situation calculus as logic programming [Kowalski, 1979; Bibel, 1986]. Besides situation calculus, most of the work on reasoning about actions considers planning on FO representations, e.g., [Sandewall, 1995; Shanahan, 1997; Thielscher, 2005].

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

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

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

McDermott [1996] introduced Unpop, a state-space search planner based on means-ends analysis. Unpop’s overall idea is similar to delete-relaxation heuristics later used in the HSP planner [Bonet and Geffner, 2001]. Moreover, McDermott’s algorithm is also similar to those used to compute lifted delete-relaxed heuristics [Corrêa et al., 2021] nowadays.

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

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

4 Lifted Heuristic Search

Recently, lifted planning made its way back into the picture with heuristic search planners. This was first made possible by using techniques from constraint satisfaction and from databases.

Modern lifted heuristic search planners perform a ground search: while the representation of actions is lifted, the explored state space is still ground. This is a crucial difference from previous approaches (e.g., SNLP, UCPOP, VHPOP) that used partially ground actions and atoms in the plan-space search — e.g., grounding only the variables of an action that were relevant to the validate a search node.

Different design factors are important when developing a planner. Literature on lifted planning has focused on two: successor generation and heuristic computation. The first one is challenging because lifted successor generation is the same as solving first-order queries (i.e., which instantiations of an action schema are applicable in a given state?). The second one is also important, as good estimates are crucial to the performance of heuristic search algorithms.

As these two aspects are orthogonal, we discuss them separately. A priori, we can choose an arbitrary successor generator and arbitrary heuristic functions to create our own lifted search.
4.1 Successor Generation

The successor generation problem is the following: given an action schema \( a \in \mathcal{A} \) together with a state \( s \), enumerate all instantiations of \( \text{vars}(a) \) yielding ground actions that are applicable in \( s \).

**Example 2.** In our running example Example 1, given the action schema \( \text{move}(C_1, C_2, T) \) and the initial state \( I \), a successor generator should return the following instantiations of the action: \( \text{move}(b, a, t), \text{move}(b, c, t) \).

There are different ways to solve this problem. All approaches listed below reduce successor generation to some well-known combinatorial problem, and then use specialized algorithms or tools to compute the answers.

Constraint Satisfaction

Francès [2017] was the first to emphasize lifted successor generation. Francès reduces the successor generation to a constraint satisfaction problem (CSP). For our purposes, it is sufficient to consider a CSP as a pair \((X, C)\) of variables \(X\) (each \( V \in X \) with pre-specified domain \( D_V \)) and a set \( C\) of constraints. A constraint \( R_{V_1} \times \cdots \times V_n \subseteq D_{V_1} \times \cdots \times D_{V_n}\) defines what are the valid value assignments of variables \( V_1, \ldots, V_n \) in any solution.

Given an action schema \( A \) and a state \( s \), we construct a CSP in which all solutions correspond to instantiations \( \sigma \) such that \( \sigma(\text{pre}(A)) \subseteq s \), i.e., \( \sigma(\text{pre}(A)) \) is applicable in \( s \).

**Example 3.** Consider the initial state \( I \) of our running example. Given the action schema \( A = \text{move}(C_1, C_2, T) \) from Example 1, we define the CSP \((X, C)\) for \( A \) as

\[
X = \text{vars}(A) = \{C_1, C_2, T\},
C = \{R_{C_1, C_2}, R_{T, C_1}\}
\]

where

\[
R_{C_1, C_2} = \{(c_1, c_2) \mid \text{conn}(c_1, c_2) \in I\}
R_{T, C_1} = \{(t_1, c_2) \mid \text{at}(t_1, c_2) \in I\}.
\]

Each variable \( V \in X \) has domain \( D_V = C \). The solution to this CSP are the assignments:

\[
\{\langle C_1 \mapsto b, C_2 \mapsto a, T \mapsto t \rangle, \langle C_1 \mapsto b, C_2 \mapsto c, T \mapsto t \rangle\}.
\]

Each of the evaluations corresponds to an applicable instantiation of the action in the initial state \( I \).

Once an applicable instantiation is found, producing the successor state is straightforward: we simply replace the variables with their respective constants, delete the atoms in the (ground) delete list, and add those in the (ground) add list.

To solve these CSPs, the planner calls an off-the-shelf solver. Although our example uses the simple STRIPS formalism, the planner by Francès [2017] supports a much richer logic. It supports functional STRIPS [Geffner, 2000], an extension where function symbols are allowed, and existential STRIPS [Francès and Geffner, 2016], a fragment that supports existentially quantified preconditions.

Conjunctive Queries

The Powerlifted planner, introduced by Corrêa et al. [2020], considers successor generation as a conjunctive query problem. A conjunctive query is an FO query that can be written using only logical conjunction and existential quantifiers. Answering a conjunctive query is NP-hard in general [Abiteboul et al., 1995], however there are cases for which this problem is tractable [Yannakakis, 1981; Gottlob et al., 2002].

The preconditions of an action schema is a conjunctive query, and the current state is a database. Each answer of the query corresponds to an applicable instantiation of the action schema. Powerlifted deals only with STRIPS extended with object typing and inequalities.

**Example 4.** In our running example, the preconditions of the action \( \text{move}(C_1, C_2, T) \) can be represented as the following conjunctive query:

\[
\exists C_1, C_2, T. \text{conn}(C_1, C_2) \land \text{at}(T, C_1).
\]

This is similar to the approach by Francès [2017], as conjunctive queries are a specific type of CSPs [Chandra and Merlin, 1977; Kolaitis and Vardi, 2000]. For instance, Example 3 is a conjunctive query. However, Powerlifted has two key differences: first, by focusing exclusively on conjunctive queries, Powerlifted exploits tractable fragments to solve these queries faster; second, instead of using a dedicated solver, it reimplements all the necessary algorithms within the planner, which reduces overhead.

While the second point is simply engineering, the first deserves more attention. To be efficient, Powerlifted decomposes [Yannakakis, 1981; Gottlob et al., 2002] the conjunctive queries. A query decomposition defines the order in which predicates must be unified. By doing so, we guarantee that the computation is efficient. In fact, queries that are easy to decompose (e.g., acyclic queries) are solved in polynomial time [Abiteboul et al., 1995]. Corrêa et al. [2020] showed that most of the domains used in the planning literature have actions that yield acyclic conjunctive queries, and hence the lifted search only has a polynomial overhead compared to a ground state-space search. Furthermore, algorithms for solving acyclic conjunctive queries also take existentially quantified variables [Yannakakis, 1981] into account. If a query is not acyclic, Powerlifted uses a heuristic technique to solve it.

Another planner applying database techniques to the lifted successor generation problem is CPDDL [Horčík and Fišer, 2021; Horčík et al., 2022; Horčík and Fišer, 2023]. In contrast to the built-in query solver of Powerlifted, CPDDL uses SQLite to solve the queries. SQLite uses other optimizations (not based on query decomposition) that can speed up the query answering. Moreover, the optimizations from SQLite work on cyclic queries too, while Powerlifted has to rely on heuristic methods. But as CPDDL uses SQLite off the shelf, it is unclear which specific optimizations the solver uses.

CPDDL supports a much more expressive fragment than the two previous planners. In fact, CPDDL supports the entire PDDL fragment used in the IPCs (quantified effects; conditional effects; disjunctive preconditions). Although there is no publication systematically comparing the successor generators of CPDDL and Powerlifted, both planners seem to be on par in performance [Horčík and Fišer, 2021].
Maximum Clique Enumeration

Stählberg [2023] introduced another method to implement successor generation based on maximum clique enumeration. The algorithm works in yet a different extension of STRIPS that allows for negative preconditions. It first builds a graph based on the action schema structure and current state, called the substitution consistency graph. Given an action schema $A$ with $\text{vars}(A) = \{v_1, \ldots, v_n\}$ and a set of constants $\mathcal{C} = \{c_1, \ldots, c_m\}$, the substitution consistency graph $G = (V, E)$ is defined as follows: for each $v \in \text{vars}(A)$ and each $c \in \mathcal{C}$, there exists a vertex $[v/c] \in V$ (i.e., this vertex represents instantiation $v$ with $c$); the set of edges is defined as $E = V^2 \setminus \mathcal{I}$, where $\mathcal{I}$ are inconsistent edges. An edge is inconsistent if it either assigns two different constants to the same variable, violates a positive precondition, or violates a negative precondition. If all predicates in the precondition have arity of at most 2, any maximum clique in $G$ corresponds to an applicable instantiation. Enumerating all the maximum cliques gives all the applicable instantiations. However, if the precondition contains predicates with arity higher than 2, the algorithm only guarantees an overapproximation of the applicable instantiations. In this case, we must post-process the answers to check for the satisfiability of the precondition.

Their implementation is on top of Powerlifted. An interesting observation is that the substitution consistency graph for an action schema with $k$ variables is $k$-partite. Hence, we can use specific algorithms for $k$-partite graphs, which work better than general algorithms. While it is not clear if their method dominates the one by Corrêa et al., Stählberg shows that it is easy to predict which one is best for a given domain based on a few sampled states.

4.2 Heuristic Computation

Most of the recent heuristics in lifted planning are inspired by a counterpart in propositional planning. We list some lifted heuristic functions below.

K-ary Relaxation

The first non-trivial heuristic estimate was the k-ary relaxation by Lauer et al. [2021]. In the k-ary relaxation of an atom $p(v_1, \ldots, v_n)$, the atom is projected on $\binom{n}{k}$ new atoms, containing all combinations of $V_1, \ldots, V_n$ with $k$ variables. A particular case is the unary relaxation, where an n-ary atom is projected on $n$ new unary atoms.

Example 5. Let $p(X,Y,Z)$ be an atom. Its unary relaxation $p|_1 = \{p_1(X), p_2(Y), p_3(Z)\}$.

The same relaxation is done for action schemas\(^2\) and states. For a planning task $\Pi$, we denote its unary relaxation by $\Pi|_1$. For a given state $s$, the planner computes a plan from $s|_1$ (the unary relaxation of $s$). The length of this relaxed plan can be used as a heuristic estimate for the original state.

Computing a plan in the unary relaxed task can still take exponential time. However, it becomes tractable when we consider delete relaxation. In short, the delete relaxation of a planning task $\Pi$ is a relaxation where all delete lists are redefined to be empty. This implies that once a fact is true in a state $s$, it remains true in all states reachable from $s$. If a task has only empty delete lists, we say it is delete-free. For a planning task $\Pi$, its delete-free version is denoted as $\Pi^+$. Lauer et al. [2021] proved that, for a delete-free unary relaxed task $\Pi^+|_1$, we can compute a plan for $\Pi^+|_1$ in polynomial time in its size. On the flip side, the heuristic is not much more informative than a heuristic that simply counts the number of unachieved atoms in the goal. But while it does not help much as a single heuristic function, this unary-relaxation heuristic improves the search when used to break ties between states that have the same $f$-value in a greedy best-first search.

Lauer et al. [2021] also show that while computing the k-ary relaxation for $k > 1$ is challenging, we can use the k-ary relaxation for a handful of atoms, and use unary relaxation for the remaining ones. The observation is that some atoms help the heuristic much more than others. For these “useful” atoms, we want to keep their information intact.

Classical Delete-Relaxation Heuristics

Delete-relaxation heuristics are a well-established family of heuristics in classical planning [Bonet and Geffner, 2001; Hoffmann and Nebel, 2001; Helmert and Domshlak, 2009]. Its typical cycle is: the planner computes the delete-relaxation of the task, finds a relaxed plan, and uses the length of this relaxed plan as a heuristic for the evaluated state.\(^3\)

Corrêa et al. [2021] showed that we can compute delete-relaxation heuristics over the lifted representation using Datalog. This extends the idea of using Datalog for grounding of planning tasks [Helmert, 2009].

A Datalog program is a pair $D = (F, R)$, where $F$ is a set of ground atoms, called the facts, and $R$ is the set of rules with the format

$$h(T) \leftarrow b_1(T_1), \ldots, b_n(T_N),$$

where the left-hand side is the head, the right-hand side the body, and $T \subseteq \bigcup_{i=1}^n T_i$. A rule is used to infer new atoms. By unifying the body of a rule $r$ with a set of facts, we can infer its head (with the same variable substitution as the body). It is easier to think of Datalog from a fixpoint perspective: starting from a set $\mathcal{M}$ of atoms, infer all possible new atoms by unifying the rules with $\mathcal{M}$; add these new atoms to $\mathcal{M}$, and repeat the process until a fixpoint is reached. By starting with $\mathcal{M} = F$, we compute the canonical model of $D$.

Example 6. The delete-relaxation of Example 1 can be encoded as the Datalog program $D = (F, R)$ where $F$ is the state being evaluated, and $R$ is as follows:

- move($C_1, C_2, T$) $\leftarrow$ conn($C_1, C_2$), at($T, C_1$).
- at($T, C_2$) $\leftarrow$ move($C_1, C_2, T$).
- pick($C, P, T$) $\leftarrow$ at($T, C$), package-at($P, C$).
- loaded($P, T$) $\leftarrow$ pick($C, P, T$).
- drop($C, P, T$) $\leftarrow$ at($T, C$), loaded($P, T$).
- package-at($P, C$) $\leftarrow$ drop($C, P, T$).
- goal $\leftarrow$ at($p, c$).

\(^2\)The relaxation is applied to the variables of the action. Atoms in precondition, add list, or delete list that contain one of the variables being projected out are removed.

\(^3\)This is true for non-admissible heuristics. Admissible delete-relaxed heuristics do not compute a relaxed plan.
Rules with action predicates (i.e., predicates with action schema names) in their heads are called action rules. The last rule is the goal rule, and goal is a special predicate called the goal predicate. The other rules (with action predicates in the body) are called effect rules.

The canonical model \( M \) of this Datalog program contains all atoms that are relaxed reachable from \( I \).

To compute a heuristic \( h \) for a state \( s \), the algorithm by Corrêa et al. [2021] assigns to each effect rule the weight of 1, and a weight of 0 to all others. It also assigns a value of 0 to every atom in \( s \). During the computation of \( M \), whenever inferring a new atom \( p \) through a unified rule \( r \), the value of \( p \) is the sum of the values of all body atoms in \( r \) plus the weight of \( r \). Corrêa et al. [2021] prove that the value of the goal atom is the same as the additive heuristic \( h^{\text{add}}(s) \) [Bonet and Geffner, 2001].

There is a second and more powerful version of this computation, introduced by Corrêa et al. [2022]. This newer version can also compute more informed heuristics, like \( h^{\text{FF}} \) [Hoffmann and Nebel, 2001]. Instead of using a simple Datalog program, Corrêa et al. [2022] use annotated Datalog programs. However, explaining this more approach takes more space, so we did not include it here. There are also more sophisticated search algorithms that combine the Datalog-based heuristics with other techniques [Corrêa and Seipp, 2022], such as width-search [Lipovetzky and Geffner, 2012; Lipovetzky and Geffner, 2017; Francès et al., 2017], preferred operators [Richter and Helmert, 2009], and queue-alternation [Rögger and Helmert, 2010].

All these Datalog-based heuristics were implemented on top of Powerlifted. The \( h^{\text{add}} \) implementation is also available in CPDDL.

**Homomorphisms**

Datalog-based heuristics compute a lifted heuristic that is identical to its ground counterpart. A different approach is to use homomorphisms [Horčík and Fišer, 2021; Horčík et al., 2022]. In this context, we are interested in homomorphisms between constants, so a homomorphism is a self-map \( m : C \rightarrow C \).

The planner first computes a homomorphism between constants of the task. This homomorphism is used to reduce the number of objects. Then, the algorithm grounds the smaller task, and uses it to extract a heuristic estimate – computing the heuristic over the ground representation. Grounding the task becomes much easier. At the same time, (optimal) plans are preserved [Horčík et al., 2022], which implies that admissible heuristics in the smaller ground task are also admissible in the original one.

In our running example, a homomorphism \( m \) could map \( c_3 \mapsto c_2 \), and all other constants to themselves. This reduces the size of the task while preserving all plans, although with potentially redundant actions.

The question is how to find good homomorphisms. The best known method was introduced by Horčík and Fišer [2023], and is based on Gaifman graphs. The Gaifman graph of a state \( s \) is constructed by taking each element of a structure as a vertex. Two vertices are connected by an edge if and only if the corresponding elements occur together in some atom of the structure. An important metric in Gaifman graphs is their diameter, as it indicates how closely related two elements are.

We can create Gaifman graphs for states and action schemas. For states, the elements are the constants \( C \) and the state itself is the structure. So two constants \( c_1 \) and \( c_2 \) (corresponding to vertices in the Gaifman graph) are connected in the graph iff there is an atom in the state containing \( c_1 \) and \( c_2 \). For an action schema \( A \), the elements are the variables in \( \text{vars}(A) \) and the structure is \( \text{pre}(A) \). Two variables have an edge iff they appear in a same atom in \( \text{pre}(A) \).

Horčík and Fišer [2023] prove that the difference in diameter between a state \( s \) and its successor \( \text{succ}(s, \sigma(A)) \) is bounded by the diameter of \( A \). They combine this information to compute which constants should be mapped to each other. Intuitively, one does not want to collapse constants that are distant to each, as they are unrelated.

Finding homomorphisms using Gaifman graphs is limited to action schemas with bounded diameters and it is sensitive to the problem formulation (e.g., unary predicates). The homomorphisms by Horčík et al. [2022] are randomly selected, and do not have such limitations. However, they perform only moderately worse than the method using Gaifman graphs. Overall, heuristics based on homomorphisms perform similarly to the Datalog-based ones.

Ridder and Fox [2014] had a similar idea to approximate \( h^{\text{FF}} \) in a lifted representation. However, they used equivalence classes. Their empirical results with the L-RPG planner show that the equivalence classes creates a lot of “shortcuts” in the plans, and so the search becomes uninformed. In fact, Corrêa et al. [2020] showed that L-RPG is not competitive with modern lifted planners.\(^4\)

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

**Landmarks**

Landmarks are atoms or actions that must occur in every plan. Landmarks are a long-standing feature in planning [Porteous et al., 2001; Hoffmann et al., 2004; Richter et al., 2008; Helmert and Domshlak, 2009]. They have also been translated to lifted planning. In our running example, \( \text{at}(t, a) \) and \( \text{at}(t, c) \) are fact landmarks; \( \text{pick}(a, p, t) \) and \( \text{drop}(c, p, t) \) are action landmarks.

Wichlacz et al. [2022] introduced two methods to extract fact landmarks, i.e., disjunctive sets of atoms that must occur in every plan. Their definition of lifted landmarks accounts for partially grounded atoms. A single partially grounded landmark can correspond to different ground landmarks.

The first method is based on necessary subgoals: it uses a backchaining process, starting from the goal atoms, to identify landmarks. Initially, it obtains all action schemas

---

\(^4\)In the previous section, we did not discuss L-RPG’s successor generator because it is a brute-force instantiation.
that could add a goal atom. In other words, for a predicate \( p(c_1, \ldots, c_m) \in G \) it collects all action schemas adding atoms with predicate symbol \( p \). These are action schemas that could be instantiated to achieve \( p(c_1, \ldots, c_m) \). It then partially grounds the action schemas (consistently with the constants used in \( p(c_1, \ldots, c_m) \)) and intersects the preconditions of the collected action schemas. Atoms in this intersection are necessary subgoals, which are treated as landmarks. The process continues iteratively with the newly found landmarks, until a fixpoint is reached.

The second method is slightly more elaborate, and uses cuts on lifted fact-alternating mutexes (FAM-groups) [Fiser, 2020]. Due to space reasons, we omit its details.

Wichlacz et al. [2022] used these methods to compute lifted landmark count heuristics [Richter et al., 2008; Richter and Westphal, 2008]. The idea is to evaluate states based on the number of satisfied landmarks, preferring states that satisfy more.

Recently, Wichlacz et al. [2023] introduced a way to compute the landmark cut (LMC) heuristic [Helmert and Domshak, 2009] in the lifted setting. The LMC heuristic generates disjunctive action landmarks also backchaining from the goal. During the backchaining procedure, it relies on sequential computation of the \( h^{\text{max}} \) heuristic — a delete-relaxed heuristic that is admissible. However, this is expensive in the lifted setting, so Wichlacz et al. [2023] showed different ways to circumvent this problem.

All the landmark-based methods described here were implemented on top of Powerlifted only.

5 Lifted Planning as Satisfiability

Not all recent advances on lifted planning use search. Some of the most successful methods were based on satisfiability. Instead of performing a state-space search, we encode the lifted planning task into a logical formula and check if the formula is satisfiable. A model of the formula encodes a plan.

This is similar to the planning as satisfiability approach by Kautz and Selman [1992]. However, the encodings we refer to below are lifted encodings. They do not refer to specific ground states or ground actions but use variables to encode action instantiations.

5.1 SAT

SAT encodings for lifted planning were first proposed by Ernst et al. [1997]. Later, Robinson et al. [2009] proposed an approach that does not require full grounding. They split up the action schemas and only partially ground actions. Their approach also exploited parallel execution of actions to make the encoding more compact.

More recently, Höller and Behnke [2022] presented a stateless encoding of lifted planning into propositional logic called LiSAT. The only state explicitly encoded is the initial state. The inspiration for their encoding comes from partial order planning [Penberthy and Weld, 1992], discussed in Section 3.

Roughly speaking, given a bound \( L \) to the plan length, the propositional formula encodes the \( L \) possible steps of the plan. A step is simply the selection of an action schema and its instantiations. The encoding does not keep track of the state at each step but it keeps track of what has been deleted and added at each step. This is a key difference to the approach by Robinson et al. [2009], which had to represent states explicitly and so required grounding.

The intuition is that if an atom is needed (e.g., in the goal), then it must be added by some previous step, and must not be deleted until the step where it is required. This can be done using a propositional encoding that is quadratic on the size of the lifted task. (See Höller and Behnke [2022] for an example.)

The encoding by Höller and Behnke together with state-of-art SAT solvers yields a performance on-par with the heuristic search planners. To the best of our knowledge, LiSAT is the state-of-the-art lifted algorithm for optimal planning. Höller and Behnke note that, although they use propositional logic, any other encoding — e.g., CSP — would suffice.

LiSAT uses Powerlifted as a wrapper for translation and internal representation. It supports STRIPS with negative preconditions and types. While we do not consider costs in this survey, all previous methods did indeed support action costs. LiSAT, however, does not.

5.2 QBF

Shaik and van de Pol [2022] propose an encoding of lifted planning problems to quantified Boolean formula (QBF). Their encoding is linear in the number of action schemas, predicates, and plan length, and logarithmic in the number of constants.

The QBF lifted planner, Q-planner, achieves remarkable performance in certain domains [Matloob and Soutchanski, 2016]. Q-planner is able to solve several tasks that other planners cannot due to high memory consumption. Their encoding also deals with STRIPS with negative preconditions.

6 Future Directions

Our paper provides an overview of the recent advances in lifted planning. But some areas still need further research. We list some ideas of future work next.

So far, a lot of effort was put into the question of how to translate classical planning heuristics from the propositional to the lifted case. Most of the work, however, focused on delete-relaxation and landmark heuristics. It is still an open question how to translate other families of heuristics [Helmert and Domshak, 2009] to the lifted setting.

In particular, abstraction [Sievers and Helmert, 2021] and operator-counting heuristics [Pommerening et al., 2014] are prominent in optimal planning, but there is no work on how to use them in lifted planning.

In more expressive fragments of PDDL, such as FOND planning [Muise et al., 2012] or numeric planning [Helmert, 2002] grounding can also become an obstacle. Lifted planning has potential to also help in these cases.

Another interesting future direction is handling infinitely many objects. Most of the current work uses PDDL, where all objects are defined in advance. However, earlier works [Green, 1969b; Reiter, 2001] considered infinitely many objects already at the initial state. In this direction, there has been some recent work in bounded situation calculus.
[De Giacomo et al., 2016; Calvanese et al., 2018]. In this special case, we have infinitely many objects but at any given state only a bounded number of them occur simultaneously in any reachable state. This is related to the problem of planning with object creation [Hoffmann et al., 2009; Fuentetaja and de la Rosa, 2016; Edelkamp et al., 2019; Corrêa et al., 2024]. It would be interesting to revisit some approaches — such as partial order planning and situation calculus — discussed in Section 3 considering the progresses of modern lifted planners. Perhaps some of these previous approaches are competitive with the newest planners. For example, Younes and Simmons [2002] showed that lifted partial-order planners outperform ground planners in a few domains. With newer lifted planners, it would be interesting to study how effective these least-commitment techniques are in the FO setting and the current used benchmarks.

Acknowledgments

We thank Remo Christen and Sasha Rubin for their comments on earlier versions of this paper.

Augusto B. Corrêa was funded by the Swiss National Science Foundation (SNSF) as part of the project “Lifted and Generalized Representations for Classical Planning” (LGR-Plan). Giuseppe De Giacomo was partially supported by the ERC Advanced Grant WhiteMech (No. 834228). Furthermore, this work was also partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under grant agreement no. 952215.

References


