Delete-Relaxation Heuristics for Lifted Classical Planning

Augusto B. Corrêa,[1] **Guillem Francès,**[2] **Florian Pommerening,**[1] **Malte Helmert**[1]
[1] University of Basel, Switzerland
[2] Universitat Pompeu Fabra, Spain
{augusto.blaascorrea,florian.pommerening,malte.helmert}@unibas.ch
guillem.frances@upf.edu

Abstract

Recent research in classical planning has shown the importance of search techniques that operate directly on the lifted representation of the problem, particularly in domains where the ground representation is prohibitively large. In this paper, we show how to compute the additive and maximum heuristics from the lifted representation of a problem. We do this by adapting well-known reachability analysis techniques based on a Datalog formulation of the delete relaxation of the problem. Our adaptation allows us to obtain not only the desired heuristic value, but also other useful heuristic information such as helpful actions. Our empirical evaluation shows that our lifted version of the additive heuristic is competitive with its ground counterpart on most of the standard international competition benchmarks, and significantly outperforms other state-of-the-art lifted heuristic methods in the literature.

Introduction

Heuristic search has been one of the dominant approaches to classical planning over the last decades. Although planning problems are often specified in some first-order logic language, most heuristics to date have been defined and implemented assuming a ground, propositional representation of the problem (Haslum and Geffner 2000; Bonet and Geffner 2001; Hoffmann and Nebel 2001; Edelkamp 2001; Helmert 2006; Richter and Westphal 2010). Thus, most planners take the first-order representation of the problem and ground it at preprocessing time, usually through efficient techniques that combine grounding with relaxed reachability analysis and work well for most standard benchmarks (Helmert 2009).

In recent years, however, a number of *hard-to-ground* planning problems have emerged (Haslum 2011; Koller and Petrick 2011; Matloob and Soutchanski 2016). These problems are challenging for standard planners because the ground representation is too large to compute without running out of memory. One way to address this problem is to restrict grounding to those objects and actions in the problem that are deemed *relevant* (Lang and Toussaint 2009; Gnad et al. 2019), which requires good relevance estimators. Another option is to rewrite the problem encoding automatically to reduce the number of ground actions, but this could obscure the problem structure (Areces et al. 2014).

In this paper we take a different approach, and aim at computing well-known classical planning heuristics *directly from the lifted representation* of the problem. More specifically, we extend the Datalog formulation of relaxed reachability by Helmert (2009) to perform the computation of the additive heuristic h^{add}, the maximum heuristic $h^{\max}$, and other relevant information of the delete relaxation such as helpful actions (Bonet and Geffner 2001; Hoffmann and Nebel 2001). Our work takes inspiration from recently-published techniques that generate the successors of a state within a forward search context using query optimization techniques (Corrêa et al. 2020). Combined with these techniques, our work results in a forward-search heuristic planner that can entirely skip the standard grounding step, which seems to be a necessary condition to solve some of the above-mentioned *hard-to-ground* problems. We discuss several optimizations and techniques to alleviate the cost of computing the heuristics in a lifted manner, and benchmark our novel algorithms on a suite with both standard and *hard-to-ground* planning domains from the literature. Overall, our lifted heuristics outperform previous lifted heuristics and are surprisingly competitive with their ground counterparts.

The rest of the paper is organized as follows. We start with the necessary background on classical planning and Datalog. We present the standard Datalog-based formulation of reachability analysis, and extend it to compute h^{add} and $h^{\max}$ values. We describe an algorithm to perform such computation, discuss different implementation choices and optimizations, and evaluate them experimentally. We close by discussing related work and conclusions.

Background

We introduce some essential background and notation. Throughout the paper, we use boldface $\mathbf{t}$ to denote tuples of symbols of some kind. We use $\mathbb{R}^+$ for the non-negative reals, and also use $\mathbb{R}^\infty = \mathbb{R}^+ \cup \{\infty\}$. Both planning languages and Datalog assume a *logical vocabulary* made up of an infinite set of variables $\mathcal{V}$, a finite number of *predicate symbols* $\mathcal{P}$, each with a fixed non-negative integer arity, and a finite set of *constants*. All vocabularies we consider are function-free. If P is an n-ary predicate and $\mathbf{t} = \langle t_1, \ldots, t_n \rangle$ is a tuple of constants and variables, then $P(\mathbf{t})$ is an *atom*. When $\mathbf{t}$ has no variables, the atom is called *ground*. A *variable substitution* σ maps variables from $\mathcal{V}$ to constants.

Planning

We consider STRIPS planning tasks with inequality constraints in their preconditions, following Corrêa et al. (2020). A (lifted) STRIPS planning task is a tuple $\Pi = \langle \mathcal{P}, O, \mathcal{A}, s_0, \gamma \rangle$. $\mathcal{P}$ is a finite set of *predicate symbols*, and O is a finite set of constants, also called *objects*. A *state* is a set of ground atoms and describes the situation in the world where exactly those atoms are true. The *initial state* s_0 describes the world at the start of the planning process. The set of *action schemas* $\mathcal{A}$ describes how the world can be changed. Each action schema $a[\Delta] \in \mathcal{A}$ consists of a *precondition* $pre(a[\Delta])$, an *add list* $add(a[\Delta])$, a *delete list* $del(a[\Delta])$, a set of *inequality constraints* $ineq(a[\Delta])$, and a cost $cost(a[\Delta]) \in \mathbb{R}^+$. The first three components are sets of atoms, whose variables are from the set of variables Δ. Each inequality constraint is a pair $\langle X, Y \rangle$ that forbids that variables X and Y of Δ be substituted with the same object.

An action schema $a[\Delta]$ with $\Delta = \emptyset$ is called a *ground action* (sometimes just *action*). Note that ground actions have no variables or inequality constraints, and their precondition, add, and delete lists are sets of ground atoms. A variable substitution σ that is defined on all variables Δ of an action schema $a[\Delta]$ and respects all of its inequality constraints is called a *grounding* of the schema. The application of a grounding to a schema results in the componentwise application of the grounding to all atoms in $pre(a[\Delta])$, $add(a[\Delta])$, and $del(a[\Delta])$. A planning task Π where all action schemas are ground is called a *ground* planning task.

We say that ground action a is *applicable* in a state s, if s satisfies all preconditions of a, i.e., if $pre(a) \subseteq s$. Applying an applicable action a in a state s results in the *successor state* $s[\![a]\!] = (s \setminus del(a)) \cup add(a)$. A sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$ is applicable in a state s if a_i is applicable in state $s[\![a_1]\!] \cdots [\![a_{i-1}]\!]$ for all i and we write the successor state as $s[\![\pi]\!] = s[\![a_1]\!] \cdots [\![a_n]\!]$. The *goal* γ of our planning task defines the conditions we want to reach by applying actions. It is a set of ground atoms and any state s such that $\gamma \subseteq s$ is called a *goal state*. A solution to the planning task or *plan* is a sequence of ground actions $\pi = \langle a_1, \ldots, a_n \rangle$ that is applicable in the initial state s_0 and leads to a goal state, i.e., $\gamma \subseteq s_0[\![\pi]\!]$. The cost of such a plan is $\sum_i cost(a_i)$, and a plan with minimal cost is an *optimal plan*.

Delete Relaxation

The *delete relaxation* (Bonet and Geffner 2001) of a (ground or lifted) planning task Π is the task Π^+ obtained from Π by setting $del(a[\Delta]) = \emptyset$ for all actions schemas $a[\Delta]$. By ignoring the delete lists, applying an action a in a state s can only add ground atoms to the state, so all actions that are applicable in s remain applicable in $s[\![a]\!]$. Thus Π^+ is an overapproximation of Π in the sense that it preserves all plans. A plan for Π^+ is often called a *relaxed plan* for Π.

The delete relaxation of a task Π can be used for grounding purposes (Helmert 2009): an atom not reachable in Π^+ is not reachable in Π either, so any ground action having such an atom in its precondition can safely be ignored.

Computing optimal plans for a delete-free task is NP-complete (Bylander 1994), but several approximations compute (possibly suboptimal) plans in polynomial time (Bonet and Geffner 2001; Hoffmann and Nebel 2001). These polynomial relaxed plans are a common source of heuristic guidance on the non-relaxed task Π.

One such delete-relaxed heuristic is the additive heuristic h^{add} defined originally for ground STRIPS tasks without action costs (Bonet and Geffner 2001), and extended to action costs in (Keyder and Geffner 2008). Its definition relies on an estimate $h(p, s)$ of the cost of achieving ground atom p from state s in the delete-free Π^+:

$$h(p,s) = \begin{cases} 0, & \text{if } p \in s \\ \min\limits_{a \in A_p} \left\{ cost(a) + \sum\limits_{q \in pre(a)} h(q,s) \right\}, & \text{otherwise,} \end{cases} \quad (1)$$

where A_p is the set of ground actions with p in the add list. The value of h^{add} is then defined as $h^{\text{add}}(s) = \sum_{p \in \gamma} h(p, s)$. The reason why h^{add} does not result in optimal estimates for Π^+ is that when computing the cost of reaching a set of atoms, say $\{p, q\}$, it ignores that reaching p might help in reaching q too. Replacing the $\sum$ operator by the $\max$ operator in the equations above yields the admissible $h^{\max}$ heuristic (Bonet and Geffner 2001).

Datalog

A *Datalog rule* r has the form $\phi_0 \leftarrow \phi_1, \ldots, \phi_m$, for $m \geq 0$, where each ϕ_i is an atom, ϕ_0 is called the rule *head*, and $\phi_1, \ldots, \phi_m$ is called the rule *body*. All atoms in Datalog are function-free. We use $head(r)$ to denote the atom ϕ_0 and $body(r)$ to denote the set $\{\phi_1, \ldots, \phi_m\}$. A *Datalog program* is a pair $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$, where $\mathcal{F}$ is a set of ground atoms called the set of *facts* and $\mathcal{R}$ is a set of Datalog rules. We use $Const(\mathcal{D})$ to denote the set of all constants appearing in $\mathcal{D}$, and $Atoms(\mathcal{D})$ to denote the (finite) set of all ground atoms made up from predicates and constants appearing in $\mathcal{D}$. We denote by $Ground(r)$ the set of rules obtained by applying all possible variable substitutions of variables in $r \in \mathcal{R}$ by constants in $Const(\mathcal{D})$. We also use $Ground(\mathcal{R}) = \cup_{r \in \mathcal{R}} Ground(r)$.

Given a Datalog rule $r \equiv \phi_0 \leftarrow \phi_1, \ldots, \phi_m$ with variables $v_1, \ldots, v_k$, we define $r_\forall \equiv \forall v_1 \ldots v_k \,.\, \phi_1 \wedge \ldots \wedge \phi_m \rightarrow \phi_0$. The *canonical model* of a Datalog program $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$ is the set $\mathcal{M} \subseteq Atoms(\mathcal{D})$ of ground atoms ϕ such that $\mathcal{F} \cup \{r_\forall \mid r \in \mathcal{R}\} \models \phi$. The canonical model is unique, and can be computed in time polynomial in the number $|\mathcal{M}|$ of atoms in the model. However, $|\mathcal{M}|$ can be exponential in the size of $\mathcal{D}$. In fact, if the set of rules $\mathcal{R}$ is considered as fixed, computing $\mathcal{M}$ is P-complete, but if $\mathcal{R}$ is considered as part of the input, then it is EXPTIME-complete (Abiteboul, Hull, and Vianu 1995; Dantsin et al. 2001).

Datalog Formulation of Delete Heuristics

We now introduce the Datalog-based approach by Helmert (2009) to ground a planning task through a relaxed reachability analysis, and make explicit a connection with delete heuristics that we will exploit in subsequent sections.

Datalog-Based Grounding of Planning Tasks

Let $\Pi = \langle \mathcal{P}, O, \mathcal{A}, s_0, \gamma \rangle$ be a planning task and s one of its states. We define $\mathcal{D}(\Pi, s) = \langle \mathcal{F}, \mathcal{R} \rangle$ as the Datalog program where $\mathcal{F}$ contains all ground atoms in s, and $\mathcal{R}$ is defined as follows: If $\gamma = \{\gamma_1, \ldots, \gamma_n\}$, $\mathcal{R}$ contains a rule

$$\texttt{goal} \leftarrow \gamma_1, \ldots, \gamma_n. \tag{2}$$

For each action schema $a[\Delta] \in \mathcal{A}$ with $pre(a[\Delta]) = \{\phi_1, \ldots, \phi_n\}$, $\mathcal{R}$ contains one *applicability rule*

$$a\texttt{-applicable}(\mathbf{x}) \leftarrow \phi_1, \ldots, \phi_n, \tag{3}$$

where $\mathbf{x}$ contains all variables in Δ in some arbitrary order. We call $a\texttt{-applicable}$ an *action predicate*. Finally, for each atom $\psi(\mathbf{x}') \in add(a[\Delta])$, $\mathcal{R}$ contains one *effect rule*

$$\psi(\mathbf{x}') \leftarrow a\texttt{-applicable}(\mathbf{x}). \tag{4}$$

Helmert (2009) shows that the canonical model $\mathcal{M}$ of $\mathcal{D}(\Pi, s_0)$ contains exactly (i) all ground atoms reachable in the delete relaxation Π^+, (ii) ground atom $a\texttt{-applicable}(\mathbf{o})$ iff ground action $a[\mathbf{o}]$ is applicable in some reachable state of Π^+, and (iii) the atom $\texttt{goal}$ iff Π^+ is solvable, i.e., all goal atoms are reachable from s_0 in Π^+.

Correspondence with h^{add} and $h^{\max}$

We can establish a straightforward correspondence between a simple generalization of the Datalog program $\mathcal{D}(\Pi, s)$ and the h-estimate of the cost of reaching any atom from state s in the planning task Π given in Equation (1). For that, we need to extend Datalog programs with weights that will be used to capture action costs.

Definition 1 (Weighted Datalog program)**.** *A* weighted Datalog program *is a tuple $\langle \mathcal{F}, \mathcal{R} \rangle$, where $\mathcal{F}$ is a set of facts and $\mathcal{R}$ a set of weighted Datalog rules. A* weighted Datalog rule *r is made up of a head $head(r)$ and a body $body(r)$, as for standard Datalog rules, plus a weight $w(r) \in \mathbb{R}^+$. The set $Ground(r)$ and the canonical model of a program are defined analogously to the standard case; in particular, variable substitutions leave the weight of rules unchanged, and the definition of the canonical model ignores rule weights.*

Definition 2. *Let Π be a planning task. $\mathcal{D}_w(\Pi, s) = \langle \mathcal{F}, \mathcal{R} \rangle$ is the weighted Datalog program defined as $\mathcal{D}(\Pi, s)$, but where each rule $r \in \mathcal{R}$ has weight $w(r) = cost(a)$, if $head(r) = a\texttt{-applicable}(\cdot)$, and $w(r) = 0$ otherwise.*

To establish our correspondence, we define a value $v_{\mathcal{D}}(p)$ for any ground atom in a weighted Datalog program:

Definition 3 ($v_{\mathcal{D}}$)**.** *Let $\mathcal{D}$ be a weighted Datalog program. The* value $v_{\mathcal{D}}(p)$ *of atom $p \in Atoms(\mathcal{D})$ is*

$$v_{\mathcal{D}}(p) = \begin{cases} 0, & \text{if } p \in \mathcal{F} \\ \min\limits_{r \in A(p)} \left\{ w(r) + \sum\limits_{q \in body(r)} v(q) \right\}, & \text{otherwise,} \end{cases} \tag{5}$$

where $A(p) = \{r \mid r \in Ground(R) \text{ and } head(r) = p\}$, and the $\min$ of an empty set is assumed to be ∞. When clear from the context, we denote $v_{\mathcal{D}}$ with the simpler v.

The following proposition can be proven by rewriting the sets of equations defining v and h until equivalence. Modifying $v_{\mathcal{D}}$ to obtain $h^{\max}$ values instead is straightforward.

Algorithm 1 Computing v for a weighted Datalog program $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$

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

17: function NewRules(p, M, R)
18:     return {(head ←^w body) ∈ Ground(R) |
        p ∈ body and body ⊆ M}
```

Proposition 1. *Let Π be a planning task and s one of its states. Let h be the function defined in Eqs.* (1)*. Then,*

1. *If p is a ground atom of Π, then $v_{\mathcal{D}_w(\Pi,s)}(p) = h(p, s)$.*
2. *If p is a ground atom $a\texttt{-applicable}(\mathbf{o})$, then $v_{\mathcal{D}_w(\Pi,s)}(p) = cost(a) + \sum_{q \in pre(a)} h(q, s)$.*
3. *$v_{\mathcal{D}_w(\Pi,s)}(\texttt{goal}) = h^{\text{add}}(s)$.*

Lifted Computation of h^{add}

We next present an algorithm that takes as input a weighted Datalog program $\mathcal{D}$ and computes $v(p)$ for each ground atom $p \in Atoms(\mathcal{D})$. Our approach builds on the algorithm given by Helmert (2009) to incrementally compute the canonical model of any standard Datalog program. The main difference with Helmert's algorithm is that ours computes the v values as it builds the canonical model of the program.

The pseudocode for our algorithm is shown in Algorithm 1. The algorithm (lines 1–3) uses (i) a hash table to store the values of v, mapping (ground) atoms to real values, and providing a default value of ∞ for uninitialized entries; (ii) a priority queue of atoms sorted by a given real priority, breaking ties arbitrarily; (iii) a set $\mathcal{M}$ that grows monotonically along the execution of the algorithm and at the end contains all ground atoms in the canonical model.

The main loop (lines 7–15) works as a generalized Dijkstra algorithm, with the lowest-cost ground atom p removed from the queue at each iteration, added to the model, then checked to see if it triggers any new ground rule. Indeed, the NewRules function computes all ground rules whose body (a) is included in the model $\mathcal{M}$ computed so far, and (b) contains the atom p being considered. This is done through a first-order unification procedure described in more detail in the next subsection. For each newly-triggered ground rule,

if the v-value of its head atom improves the previous stored value, we update it and enqueue the atom.

Once the priority queue pops an atom p with $V[p] = C$, it will never pop an atom p' with $V[p'] < C$. This happens because all rule weights are non-negative by definition and, as the algorithm is a generalized version of Dijkstra's, as soon as an atom p is added to $\mathcal{M}$, the value of $V[p]$ is minimal and equals its desired v-value.

Efficient Implementation

Algorithm 1 allows to compute the h^{add} value of any state s from the lifted representation of the planning task Π alone: first generate the weighted Datalog program $\mathcal{D}_w(\Pi, s)$, then compute the v-value of the `goal` atom. As we are only interested in obtaining the h^{add} value, the algorithm can stop as soon as the `goal` atom is extracted from the queue, since at that point $V[\texttt{goal}] = v_{\mathcal{D}_w(\Pi,s)}(\texttt{goal}) = h^{\text{add}}(s)$. We use this *early stopping* rule in all the experiments below, and call the entire procedure L-h^{add}.

As in typical algorithms for computing h^{add} from the ground representation of Π (Liu, Koenig, and Furcy 2002), the runtime and memory consumption of L-h^{add} is worst-case polynomial in the size of the ground representation of Π, which can in turn be exponentially larger than Π. However, while algorithms working on the ground representation always have to pay the price of this one-off exponential grounding step, L-h^{add} grounds the task only lazily, explicating those atoms and actions with h^{add} value no larger that the h^{add} value of the atoms of the goal. This could result in a time advantage, but this is typically not the case, since this grounding step is amortized over many calls to the heuristic. However, in terms of memory consumption there is no such amortization effect. The memory consumption of grounded implementations equals the total size of all ground atoms and actions, which in some tasks can easily exhaust the available memory, whereas the memory consumption of L-h^{add} can be sublinear in that size of the grounded problem.

An interesting implementation choice is posed by the NEWRULES function, which is in charge of computing all ground rules that are (newly) triggered by the atom p that was just added to the model $\mathcal{M}$ being built. Although its definition in Algorithm 1 is given in terms of the set of ground rules of the input Datalog program $\mathcal{D} = \langle \mathcal{F}, \mathcal{R} \rangle$, the challenge is to implement it without explicitly computing the set $\mathit{Ground}(\mathcal{R})$. To do that, the function needs to perform a particular first-order unification query which consists on finding, for each (lifted) rule $r \in \mathcal{R}$, all substitutions σ of the variables in r by $\mathit{Const}(\mathcal{D})$ such that the set that results from applying σ to all atoms in $\mathit{body}(r)$ is contained in $\mathcal{M}$, and at the same time contains a given ground atom p.

To perform this unification efficiently, we use the rule-rewriting approach described by Helmert (2009). Its main idea is to split the rules in $\mathcal{R}$ into smaller rules so that the unification query in NEWRULES can be implemented efficiently with specialized data structures. This split is done only once at preprocessing time. We refer the reader to the original paper for details on these data structures, and discuss here only the rule rewriting procedure and how it affects the v-values defined in Eq. (5).

The rewriting first removes duplicate occurrences of the same variable in the same atom of some rule, then splits each rule into a number of rules with at most two atoms in the body. This rule splitting iteratively picks one rule $r \equiv \phi_0 \leftarrow \phi_1, \ldots, \phi_i, \ldots, \phi_j, \ldots, \phi_m \in \mathcal{R}$ and replaces it with a pair of new rules r_1 and r_2, which are either of the form

$$\begin{aligned} r_1 &\equiv \phi_0 \leftarrow \theta, \phi_1, \ldots, \phi_{i-1}, \phi_{i+1}, \ldots, \phi_m \\ r_2 &\equiv \theta \leftarrow \phi_i \end{aligned}$$

or of the form

$$\begin{aligned} r_1 &\equiv \phi_0 \leftarrow \theta, \phi_1, \ldots, \phi_{i-1}, \phi_{i+1}, \ldots, \phi_{j-1}, \phi_{j+1}, \ldots, \phi_m \\ r_2 &\equiv \theta \leftarrow \phi_i, \phi_j \end{aligned}$$

Both rewriting steps require the introduction of a new auxiliary atom θ. The choices in how to perform these splits embody well-known query optimization heuristics that aim at pushing projections through joins and minimizing the join computation effort. Each of these rewriting steps can be extended to weighted rules by setting $w(r_1) = w(r)$ and $w(r_2) = 0$, and it can be shown that doing so, the value of $v(p)$ for all ground atoms p in the original program is preserved after each rewriting step.

Optimizations

We consider two novel optimizations of the weighted Datalog program $\mathcal{D}_w(\Pi, s)$ aiming at speeding up Algorithm 1.

Action Predicate Removal Röger et al. (2020) have recently shown that in many of the standard planning tasks, the number of ground actions is vastly larger than that of ground atoms. When this is the case, it might be preferable to remove the $a\texttt{-applicable}(\cdot)$ atoms from rules (3–4) and use one single rule to link together the precondition $\{\phi_1, \ldots, \phi_n\}$ of each action schema with each atom $\psi(\mathbf{x})$ of its add list:

$$r \equiv \psi(\mathbf{x}) \leftarrow \phi_1, \ldots, \phi_n, \tag{6}$$

where $w(r) = \mathit{cost}(a)$. This results in smaller canonical models, and has the additional benefit of immediately projecting away all those variables that are not relevant for the add effect at hand. This optimization is always performed before the rule splitting, as it can impact how the algorithm splits the rules. Given the particular structure of our program $\mathcal{D}_w(\Pi, s)$, it can be shown that the value $v(p)$ of all atoms in the program that are not the removed $a\texttt{-applicable}(\cdot)$ atoms remains unaffected by this transformation.

Duplicate Rule Removal The rule splitting technique generates a large number of intermediate auxiliary predicates, some of which are often defined by sets of rules that are syntactically equivalent, up to variable renaming. Since by construction these auxiliary predicates occur only in the head of one rule, it is easy to detect equivalent classes of auxiliary predicates and replace all occurrences of each predicate in a class by an arbitrary representative of the class. As the rules defining the extension of auxiliary predicates have all weight 0 and this rewriting step only replaces syntactically equivalent atoms, the rewriting does not affect the v values of non-auxiliary atoms.

Reducing Evaluation Time

As the evaluation of h^{add} for a given state might be one of the bottlenecks of the search, it is important to keep the number of evaluations low. One way to reduce evaluations is by applying *deferred evaluation* (Helmert 2006). Deferred evaluation algorithms do not evaluate a state when generated, but only when expanded. States are added to the open list with the h-value of their parent. We call *lazy* an algorithm that uses deferred evaluation, and *eager* one that does not. Richter and Helmert (2009) show that deferred evaluation paired with heuristics such as h^{add} can reduce the number of evaluations necessary to solve a problem, but also that on its own, deferred evaluation can make the performance of the planner worse, as it makes the search more uninformed. Indeed, deferred evaluation works best when combined with additional heuristic mechanisms, such as *preferred operators* (Hoffmann and Nebel 2001; Helmert 2006). Preferred operators, also referred to as *helpful actions*, are actions that are considered particularly promising on a given state, typically because they are part of a relaxed plan or because they can make actions of this plan applicable. The planner can use them to prune states not generated by preferred operators (at the price of making the search incomplete), or to prioritize such states when selecting the next state for expansion. Both strategies can help the lazy search find promising successors without evaluating them (Richter and Helmert 2009).

We can extract preferred operators while computing the lifted h^{add} heuristic by keeping track of the *best achievers* for each atom in the model of our weighted Datalog program. The best achiever of an atom p is the body of the rule minimizing the right-hand side of (5). This is an adaptation of the idea proposed in (Keyder and Geffner 2008) to extract a relaxed plan without computing the planning graph. In Algorithm 1, we can add a hash table B mapping atoms to its best achievers. When we update the value of $V[\mathit{head}]$ (line 13), we also set $B[\mathit{head}] = \mathit{body}$. Once the `goal` atom is achieved, we backtrack from it collecting its achievers $B[\texttt{goal}] = \gamma_1, \ldots, \gamma_n$, then the achievers $B[\gamma_1], \ldots, B[\gamma_n]$, and so on, until we reach a fixed point. The union of all the best achievers corresponds to the preconditions of all actions in the relaxed plan. Any action producing one of these atoms is considered helpful.

Experimental Results

We have implemented the described algorithms and optimizations in the open-source planning system introduced by Corrêa et al. (2020). The source code used is available online (Corrêa et al. 2021). Our experiments were run on an Intel Xeon Silver 4114 processor running at 2.2 GHz with a runtime limit of 30 minutes and a maximum 16 GiB of memory per task.

We benchmark our algorithms on 35 domains, divided into two disjoint sets. The *IPC* set contains 1001 tasks from all 29 domains encoded in STRIPS with types from the satisficing track of International Planning Competitions held to date. The *HTG* set contains 418 tasks over the 6 *hard-to-ground* domains used by Corrêa et al. (2020). All of these domains are encoded in STRIPS with types and inequalities.

We have only benchmarked our lifted implementation L-h^{add} of h^{add}, not $h^{\max}$. We use it to guide two variations of a standard greedy best-first search (GBFS), one eager and one lazy, as explained in the previous section. In all cases, states are expanded using the "full reducer" lifted successor generation technique described in (Corrêa et al. 2020).

Impact of Optimizations

We first discuss the impact of the two optimizations to the $\mathcal{D}_w(\Pi, s)$ program we use to compute L-h^{add}: action predicate removal and duplicate rule removal. Figure 1 shows coverage on the entire set including IPC and HTG instances. The baseline version using none of the optimizations solves 774 tasks. Duplicate rule removal has a very mild effect, taking the total number of solved tasks after 30 minutes to 782. Action predicate removal has a more significant impact, increasing coverage to 821. The combination of both optimizations has the strongest effect and takes the total number of solved instances to 862.

When factoring into the analysis the benchmark set, in the IPC set, each individual optimization increases coverage by 7. In contrast, in the HTG set action predicate removal increases coverage by 40 tasks, while duplicate rule removal only by 1. This can be due to the fact that HTG domains are usually challenging for the grounding precisely because of the large number of ground actions in their instances.

Finally, we benchmarked an additional rewriting technique to take into account negated preconditions when computing our model. In the HTG set, all Organic Synthesis variants and both variants of the Genome Edit Distance (GED) domain have inequality constraints on the action schemas, which in our Datalog formulation are ignored, with the subsequent potential loss of heuristic information. We ran an experiment where instead of ignoring them, we replace each inequality constraint $X \neq Y$ in *ineq*$(a[\Delta])$ with a fresh *not-equal(X,Y)* atom, then add initial facts *not-equal*(o, o') for every pair of different objects o, o' in the task. In our results, this caused more harm than good. Only in 2 of the 368 instances in the HTG set using inequality constraints the compilation resulted in a different initial h^{add} value, but the overhead of dealing with the additional atoms in the state evaluation decreases the total number of solved tasks from 183 to 160 in the GED variants and from 46 to 17 in the Organic Synthesis variants. We therefore ignore this technique in what follows.

Deferred Evaluation and Preferred Operators

Next we discuss the two strategies previously introduced to mitigate the cost of computing the lifted h^{add} heuristic. The first four result columns in Table 1 show the results for L-h^{add} with four variations of greedy best-first search (GBFS): eager GBFS ("Eager h^{add}" column), lazy GBFS ("Lazy h^{add}" column), lazy GBFS with preferred-operator based pruning ("Lazy h^{add}+PO/Pr" column), and lazy GBFS with preferred-operator based boosted dual queue ("Lazy h^{add} + PO/DQ" column). All these methods have already been discussed above.

The three lazy variations improve coverage compared to the eager GBFS with L-h^{add} in both benchmark sets. While

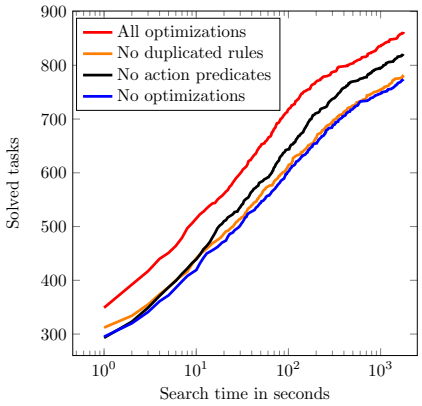

Figure 1: Solved tasks over time with different optimizations on the Datalog program for computing L-h^{add}.

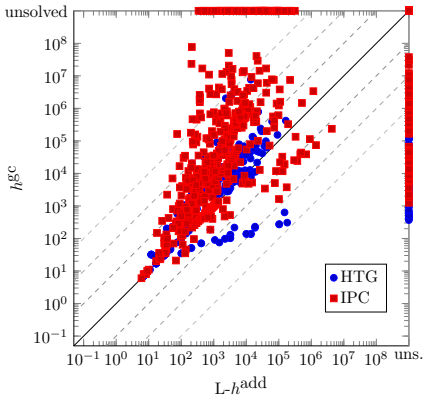

Figure 2: Number of total state evaluations: h^{gc} vs. L-h^{add}.

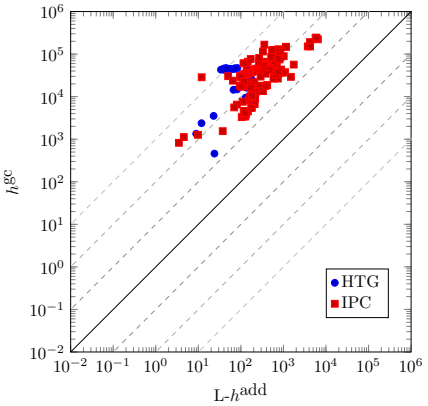

Figure 3: Number of state evaluations per second: h^{gc} vs. L-h^{add}. Only tasks solved by both approaches shown.

the improvement is less expressive when using lazy GBFS alone, the combination with any of the two possible usages of preferred operators makes much more difference. On the IPC set, Lazy h^{add} + PO/DQ solves the highest number of tasks: 754 out of 1001, compared e.g. to 616 of the eager approach. The incomplete Lazy h^{add} + PO/Pr increases coverage significantly too, solving 699 tasks. Looking at individual domains, Lazy h^{add} + PO/DQ (respectively, Lazy h^{add} + PO/Pr) performs better than Eager h^{add} in 18 (17) domains, and worse only in 3 (4).

On the HTG set, the Lazy h^{add} + PO/DQ search never performs worse than eager h^{add} search, and overall is again the best choice among the four variations, solving 362 out of 418 tasks. Most of the increase in coverage with respect to the 251 instances solved by the eager approach comes from the split variant of the Genome Edit Distance GED domain. This large increase is simply not there if preferred operators are not used. All in all, the results show that the extraction of preferred operators during the lifted computation of h^{add} and its usage within a lazy search approach are key for good performance.

Comparison to Other Lifted Methods

We now compare our L-h^{add} implementation to two previous lifted heuristic methods in the literature. The "h^{gc}" column in Table 1 shows the performance of the same greedy best-first search with a simple goal-count heuristic that counts the number of unachieved goal atoms in the given state. The "L-RPG" column shows the performance of the L-RPG lifted planner by Ridder and Fox (2014), which uses a lifted version of the relaxed planning graph (Hoffmann and Nebel 2001) to compute a heuristic similar to h^{FF}. We discuss these two baselines in turn.

Comparison to Goal-count Heuristic We first analyzed the cost-effectiveness of L-h^{add} by comparing it to the much simpler goal-count heuristic h^{gc}. Note that h^{gc} can be easily computed from the lifted representation, and it is cheap enough so that lazy search makes little sense for it.[1]

Figures 2 and 3 respectively show how informed and expensive each of the two heuristics is in our entire benchmark set when compared to each other. Fig. 2 plots the total number of necessary state evaluations until a plan is found using h^{gc} and Eager GBFS + L-h^{add}, while Fig. 3 plots the number of state evaluations per second. As expected, L-h^{add} is clearly more informed in a vast majority of cases, requiring fewer state evaluations to solve the task, but is at the same time much more expensive than h^{gc}.

Looking at coverage on the IPC benchmark, h^{add} + PO/DQ solves 157 instances more than h^{gc}. There are 15 domains where it solves more instances than h^{gc}, and 5 where the opposite happens.

In contrast, on the HTG benchmark h^{gc} solves 20 instances more than h^{add} + PO/DQ. The advantage of h^{gc} comes entirely from the split variation of the Genome Edit Distance domain, where the number of action schemas and the plan length is known to be substantially higher than in the standard version (Haslum 2011; Areces et al. 2014). Because of its higher cost, this is likely to affect L-h^{add} more adversely than h^{gc}. Additionally, h^{add} seems particularly uninformed in this domain. A standard eager GBFS with our L-h^{add} solves 28 of the 156 instances, only 6 more than a blind breadth-first search. Only in 4 of these 28 instances does L-h^{add} find a plan with fewer state evaluations than h^{gc}.

Over the entire benchmark set, h^{add} + PO/DQ performs better than h^{gc} in 16 domains, whereas the opposite happens in 7 domains. While it seems clear that on IPC domains the higher cost of h^{add} + PO/DQ pays off and the better heuristic guidance results in more solved tasks, on HTG domains the picture is less clear, and more domains would be needed for better understanding of the cost-effectiveness of L-h^{add}.

Comparison to L-RPG Since L-RPG does not support constant symbols in preconditions and effects, we report the results for only 867 instances for the IPC benchmark

[1] We indeed ran a lazy GBFS with h^{gc} (not shown in results table), but it was consistently worse than the eager version.

	Lifted (New contributions)				**Lifted** (Other)		**Ground** (FD)	
	Eager	**Lazy**			**Eager**		**Eager**	**Lazy**
Coverage	h^{add}	h^{add}	h^{add}+ PO/Pr	h^{add}+ PO/DQ	h^{gc}	L-RPG	h^{add}	h^{add}+ PO/DQ
airport (50)	23	24	24	24	26	–	36	**40**
barman-sat14-strips (20)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
blocks (35)	**35**	**35**	**35**	**35**	**35**	32	**35**	**35**
childsnack-sat14-strips (20)	0	0	**7**	**7**	0	–	2	2
depot (22)	9	10	**18**	**18**	13	8	14	17
driverlog (20)	16	16	8	14	19	14	19	**20**
freecell (80)	73	77	75	75	30	0	**80**	**80**
grid (5)	2	2	**5**	**5**	3	2	4	**5**
gripper (20)	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
logistics00 (28)	**28**	**28**	**28**	**28**	**28**	18	**28**	**28**
logistics98 (35)	15	18	29	29	5	8	27	**31**
miconic (150)	**150**	**150**	**150**	**150**	**150**	93	**150**	**150**
movie (30)	**30**	**30**	**30**	**30**	**30**	0	**30**	**30**
mystery (30)	18	17	16	17	15	12	18	**19**
nomystery-sat11-strips (20)	1	2	5	**6**	**6**	–	**6**	**6**
openstacks-strips (30)	18	9	9	9	8	–	24	**28**
parking-sat11-strips (20)	10	19	**20**	**20**	0	–	19	**20**
parking-sat14-strips (20)	2	5	16	16	0	–	6	**18**
pipesworld-notankage (50)	22	25	39	36	37	27	27	**42**
pipesworld-tankage (50)	12	15	21	21	21	12	21	**36**
psr-small (50)	42	43	0	**50**	48	–	**50**	**50**
rovers (40)	14	16	38	38	18	24	30	**40**
satellite (36)	26	23	**36**	**36**	11	21	31	**36**
thoughtful-sat14-strips (20)	6	9	10	10	5	2	15	**16**
tpp (30)	15	16	28	28	12	25	24	**30**
trucks-strips (30)	8	9	10	10	5	–	15	**16**
visitall-sat11-strips (20)	1	2	2	2	**20**	0	4	4
visitall-sat14-strips (20)	0	0	0	0	**12**	0	0	0
zenotravel (20)	**20**	**20**	**20**	**20**	**20**	13	**20**	**20**
IPC Sum (1001)	616	640	699	754	597	331	755	**839**
genome-edit-distance (156)	155	153	155	155	**156**	50	155	**156**
genome-edit-distance-split (156)	28	33	130	130	**156**	62	71	101
organic-synthesis-alkene (18)	**18**	**18**	**18**	**18**	**18**	14	**18**	**18**
organic-synthesis-MIT (18)	**18**	**18**	15	**18**	**18**	0	2	2
organic-synthesis-original (20)	10	**12**	**12**	**12**	**12**	0	1	1
pipesworld-tankage-nosplit (50)	22	21	**30**	29	22	11	17	20
HTG Sum (418)	251	255	360	362	**382**	137	264	298
Total Sum (1419)	867	895	1059	1116	979	468	1019	**1137**

Table 1: Domain-wise coverage for all tested methods. Number of instances in each domain shown in parenthesis next to domain name. Best results over all algorithms shown in bold typeface; best results over lifted planners shown with shaded background. Domains not supported by L-RPG have their coverage marked as "–".

set. This does not affect the results in the HTG benchmark, where L-RPG can process all domains. In the (reduced) IPC set, L-h^{add} solves a total of 483 instances, for 339 of L-RPG. In the HTG set, L-h^{add} solves 251 instances, for 137 solved by L-RPG. L-h^{add} solves more instances than L-RPG in 23 domains, whereas the contrary never happens.

Comparison to Grounded h^{add}

Finally, we compare the lifted L-h^{add} to its ground, standard counterpart, as implemented in Fast Downward (Helmert 2006), which we denote FD-h^{add}. We run FD-h^{add} both in an eager GBFS (to get a clearer picture) and in a lazy GBFS with preferred-operator based boosted dual queue (which is the best-performing configuration for L-h^{add}).[2]

On the IPC set, FD-h^{add} has superior coverage in both search configurations. Whereas in an eager search FD-h^{add} solves 22.6% more instances than L-h^{add}, in the lazy configuration its advantage decreases to 11.3%. The lazy strategy and preferred operators usage thus offer a superior per-

[2]We note however that the operators considered as *preferred* by L-h^{add} are those achieving some precondition of any operator in the relaxed plan, whereas Fast Downward considers an operator preferred only if it is included in the relaxed plan found by h^{add}.

formance boost for the more expensive L-h^{add} heuristic. We note that the domains where L-h^{add} performs much worse than FD-h^{add} are often (the STRIPS version of) domains such as Airport, Trucks, or Openstacks, which indeed have been partially pre-grounded to convert them into STRIPS from previous encodings in more expressive languages such as ADL. An effect of this pre-grounding is that these STRIPS encodings have a large number of action schemas with few, if any, action parameters, which makes our lifted approach clearly the wrong tool to tackle them.

On the HTG set, L-h^{add} with a boosted dual-queue solves more instances than its ground counterpart in 4 of the 6 domains, whereas the opposite happens in just 1 domain. Overall, L-h^{add} solves 21.5% more instances. FD-h^{add} also performs poorly in the split variant of GED, which is consistent with our previous analysis on how little informed h^{add} is in this domain. Indeed, an eager GBFS with the standard (ground) h^{FF} heuristic (Hoffmann and Nebel 2001) solved all 156 tasks of the domain.

All in all, our results show that while solving 11.3% fewer IPC instances than its ground counterpart, the lifted computation of both h^{add} and related heuristic information such as preferred operators is a significant step towards closing the gap between lifted and ground heuristic techniques, offering a very significant boost with respect to preexisting lifted heuristic methods.

Related Work

Recent years have seen a surge of interest in lifted planning techniques (Lang and Toussaint 2009; Areces et al. 2014; Gnad et al. 2019), likely motivated by the emergence of new planning problems that are not easy to ground using standard techniques (Haslum 2011; Koller and Petrick 2011; Matloob and Soutchanski 2016), as discussed in the introduction. However, planning approaches that do not rely on grounding the entire task as a preprocessing step are far from new, and not at all restricted to the planning-as-heuristic-search paradigm (Penberthy and Weld 1992; Younes and Simmons 2002; Robinson et al. 2008).

On the heuristic side, McDermott (1996) is closely related to our work. The author presents a heuristic based on a *regression-match graph*, which estimates the number of operators necessary to achieve each subgoal of the task by applying a backward-chaining-like inference method on a representation of the planning task that uses ground states but lifted action schemas. More recently, Ridder and Fox (2014) introduce a heuristic algorithm (L-RPG) based on a lifted version of the delete-free relaxed planning graph on which the h^{FF} heuristic is based (Hoffmann and Nebel 2001). The heuristic that L-RPG computes however is not equivalent to h^{FF}, as it takes into account almost-equivalence relationships between objects of the task for performance reasons. While indeed this often results in fast heuristic computations, it comes at a price in decreased heuristic quality (Ridder and Fox 2014). We have compared L-RPG to our approach in the experimental results previously reported.

More recently, Lauer (2020) has introduced lifted heuristics based on the *unary relaxation*, which is a further relaxation of the delete relaxation that we use in this work. For a predicate P with arity n, his method creates n predicates $P_1, \ldots, P_n$, where P_i is the relational-algebraic projection of P over its i-th argument. Applying this relaxation to all elements of a delete-relaxed task (predicates, action schemas, initial state, and goal) results in the unary-relaxed version of the task, where all predicates have arity at most one. This allows the planner to compute a relaxed plan and associated heuristic value very quickly. Since the source code of Lauer's approach is not available and the work is still in a preliminary stage, we did not empirically compare his approach to ours. However, in his experiments Lauer reports that the heuristic value is often not informative, and that the overall performance of his planner is not better than a GBFS with the goal-count heuristic.

Outside the planning literature, we can see a similarity between our lifted computation of h^{add} and the Rete pattern matching algorithm (Forgy 1982). A Rete-based system uses a tree where each node corresponds to some element (in our context, a predicate symbol) in the body of a rule. Also, each node has a table to memorize the facts produced matching this element. The path from a root node (which does not correspond to any element of a rule) to a leaf node passes through all elements in the body of some rule. When a new fact is explored, it is added to its corresponding node, and the information is passed through the network. Once it reaches a leaf node, it produces a new fact and this fact is added to the knowledge-base. This process is similar to our exploration of facts, and the network and data structures used are similar to the ones used by Helmert (2009).

Conclusions

We have introduced a novel technique to compute both the h^{add} and h^{max} heuristics, along with other relevant heuristic information, *directly from the lifted representation of a classical planning task*. In contrast to previous work, our heuristic is not a lifted approximation of a well-established heuristic, but actually computes the exact same heuristic, allowing a cleaner experimental comparison of the advantages and disadvantages of the lifted computation. We have combined our heuristics with the lifted successor generation techniques introduced by Corrêa et al. (2020) in order to obtain a forward-search heuristic planner that can entirely skip the standard grounding preprocessing step. The empirical performance of our planner is significantly better than that of previous lifted heuristic methods, and makes an important step in closing the gap with ground heuristic methods.

Our Datalog-based heuristic framework offers interesting possibilities for integrating other standard delete-relaxed heuristics such as h^{FF}. Furthermore, the optimizations to the Datalog program that we have presented and evaluated can be easily ported to the grounding algorithm of Fast Downward (Helmert 2006, 2009). In the future, we look forward to implementing other Datalog evaluation techniques that appear well-suited to our problem, such as *magic sets* (Abiteboul, Hull, and Vianu 1995), and to exploit exciting recent advances in the detection of symmetries from lifted tasks that could potentially speed up the computation of our Datalog evaluation (Röger, Sievers, and Katz 2018).

Acknowledgments

This work was funded by the Swiss National Science Foundation (SNSF) as part of the project "Certified Correctness and Guaranteed Performance for Domain-Independent Planning" (CCGP-Plan). Furthermore, this research was also partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under grant agreement no. 952215. G. Francès is supported by grant IJC2019-039276-I, MICINN, Spain.

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.

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

Bylander, T. 1994. The Computational Complexity of Propositional STRIPS Planning. *AIJ* 69(1–2): 165–204.

Corrêa, A. B.; Francès, G.; Pommerening, F.; and Helmert, M. 2021. Code from the paper "Delete-Relaxation Heuristics for Lifted Classical Planning". https://doi.org/10.5281/zenodo.4594669.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation using Query Optimization Techniques. In *Proc. ICAPS 2020*, 80–89.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys* 33(3): 374–425.

Edelkamp, S. 2001. Planning with Pattern Databases. In *Proc. ECP 2001*, 84–90.

Forgy, C. 1982. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *AIJ* 19(1): 17–37.

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.

Haslum, P. 2011. Computing Genome Edit Distances using Domain-Independent Planning. In *ICAPS 2011 Scheduling and Planning Applications woRKshop*, 45–51.

Haslum, P.; and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In *Proc. AIPS 2000*, 140–149.

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.

Keyder, E.; and Geffner, H. 2008. Heuristics for Planning with Action Costs Revisited. In *Proc. ECAI 2008*, 588–592.

Koller, A.; and Petrick, R. 2011. Experiences with Planning for Natural Language Generation. *Computational Intelligence* 27(1): 23–40.

Lang, T.; and Toussaint, M. 2009. Relevance grounding for planning in relational domains. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 736–751. Springer.

Lauer, P. 2020. *Unary Relaxation*. Bachelor's thesis, Saarland University.

Liu, Y.; Koenig, S.; and Furcy, D. 2002. Speeding Up the Calculation of Heuristics for Heuristic Search-Based Planning. In *Proc. AAAI 2002*, 484–491.

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. 1996. A Heuristic Estimator for Means-Ends Analysis in Planning. In *Proc. AIPS 1996*, 142–149.

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 Helmert, M. 2009. Preferred Operators and Deferred Evaluation in Satisficing Planning. In *Proc. ICAPS 2009*, 273–280.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR* 39: 127–177.

Ridder, B.; and Fox, M. 2014. Heuristic Evaluation Based on Lifted Relaxed Planning Graphs. In *Proc. ICAPS 2014*, 244–252.

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.; Helmert, M.; Seipp, J.; and Sievers, S. 2020. An Atom-Centric Perspective on Stubborn Sets. In *Proc. SoCS 2020*, 57–65.

Röger, G.; Sievers, S.; and Katz, M. 2018. Symmetry-based Task Reduction for Relaxed Reachability Analysis. In *Proc. ICAPS 2018*, 208–217.

Younes, H. L. S.; and Simmons, R. G. 2002. On the Role of Ground Actions in Refinement Planning. In *Proc. AIPS 2002*, 54–62.