Polynomial-Time in PDDL Input Size: Making the Delete Relaxation Feasible for Lifted Planning

Submission #14

Abstract

Polynomial-time heuristic functions for planning are commonplace since 20 years. But polynomial-time in which input? Almost all existing approaches are based on a grounded task representation, not on the actual PDDL input which is exponentially smaller. This limits practical applicability to cases where the grounded representation is “small enough”. Previous attempts to tackle this problem for the delete relaxation leveraged symmetries to reduce the blow-up. Here we take a more radical approach, applying an additional relaxation to obtain a heuristic function that runs in time polynomial in the size of the PDDL input. Our relaxation splits the predicates into smaller predicates of fixed arity $K$. We show that computing a relaxed plan is still NP-hard (in PDDL input size) for $K \geq 2$, but is polynomial-time for $K = 1$. We implement a heuristic function for $K = 1$ and show that it can improve the state of the art on benchmarks whose grounded representation is large.

Introduction

Heuristic search is a dominant paradigm for effective planning (e.g. (Hoffmann and Nebel 2001; Helmer and Domshlak 2009; Richter and Westphal 2010; Seipp 2019)). Polynomial-time computable heuristic functions are an essential ingredient to this success, and have been extensively investigated since 20 years. A particularly important technique is the delete relaxation (Bonet and Geffner 2001), which ignores negative effects (in a propositional encoding), essentially pretending that state variables accumulate their values rather than switching between them. Most state-of-the-art satissficing planning systems (which do not prove optimality of the solution returned) still use the delete relaxation or extensions thereof today (e.g. (Helmert et al. 2011; Domshlak, Hoffmann, and Katz 2015; Cenamor, de la Rosa, and Fernández 2016)).

Virtually all of these approaches however suffer from the fact that “polynomial-time” is relative to the size of a grounded task representation. This is in contrast to the actual PDDL input of the planning system, which is lifted, specifying predicates and action schemas parameterized with variables ranging over a finite universe of objects. The grounded representation size is exponential in the size of that input, specifically in the arity of the predicates and action schemas. This is not a practical problem when the grounded representation is small enough to be feasible. Yet in a variety of application scenarios that is not so (e.g. (Hoffmann et al. 2006; Koller and Hoffmann 2010; Koller and Petrick 2011; Haslum 2011; Matloob and Soutchanski 2016)).

Lifted planning has always been considered (e.g. (Penberthy and Weld 1992; Younes and Simmons 2003)), and indeed was dominant in the early 90s (Russell and Norvig 1995). There has been little progress however on transferring the wealth of known heuristic functions to the lifted setting. The only previous attempt considered the delete relaxation and leveraged symmetries to reduce the grounding blow-up in relaxed planning (Ridder and Fox 2014). Later works devised lifted domain analyses to reduce task size (Röger, Sievers, and Katz 2018; Sievers et al. 2019; Fišer 2020).

Here we take a more radical approach, applying an additional relaxation to obtain a heuristic that runs in time polynomial in the size of the PDDL input. Our relaxation splits the predicates $P(x_1, \ldots, x_n)$ in the PDDL input task $\Pi$ into smaller predicates $P_i(x_{i1}, \ldots, x_{ik})$ of arity $K$, where $\{i_1, \ldots, i_K\} \subseteq \{1, \ldots, n\}$. Specifically, every occurrence of $P$ is replaced by the conjunction of $P_i$ for all size-$K$ subsets of $P$’s parameters. The size of the resulting lifted planning task $\Pi^K$ is exponential only in $K$, hence polynomial for fixed $K$. This is a relaxation in conjunction with the delete relaxation, in the sense that every plan for $\Pi$ is a delete-relaxed plan for $\Pi^K$. We show that computing a delete-relaxed plan for $\Pi^K$ is still NP-hard (in PDDL input size) for $K \geq 2$, but is polynomial-time for $K = 1$. We implement a heuristic function for $K = 1$, and we devise an optimization that leverages some $K = 2$ information from static predicates.

Our implementation is on top of the code base of the Power Lifted Planner recently introduced by Corrêa et al. (2020), which grounds predicates and actions lazily during the forward search process. Standard International Planning Competition (IPC) benchmarks are not suited for evaluation as they are designed to challenge search rather than the grounding process. The only benchmarks currently available to challenge grounding are the ones by Areces et al. (2014), which contain action schemas of large arity (their work was about splitting large action schemas into several smaller ones). Correia et al. used these benchmarks. Here we go beyond this by exploring different reasons for being hard-to-ground: (a) large action-schema arity;
(b) large predicate arity, which entails large action-schema arity but may have other consequences; (c) large object universe, which can be problematic even for small action/predicate arity. For (a) we use Arecos et al.’s benchmarks; for (b) we generalize two IPC domains (visi and childsnack) that have a naturally scalable dimensionality parameter; for (c) we generate larger instances of some IPC benchmark domains in a spirit similar to one experiment reported about by Ridder and Fox (2014). For both (b) and (c), we take care to generate huge instances that are however within (and just beyond) reach of current lifted planners, in a manner similar to typical benchmark design in the IPC (Long and Fox 2003; Hoffmann et al. 2006; Gerevini et al. 2009; Coles et al. 2012; Vallati, Chripa, and McCluskey 2018). The design of this benchmark suite tailored to the evaluation of lifted planning is another contribution of our work. Our experiments show that our new polynomial-time lifted heuristic functions can improve the state of the art on these benchmarks, in particular through combination with goal counting.

**Background**

A lifted planning task is a tuple $\Pi = (P, O, A, I, G)$ where $P$ is a set of (first-order) predicates, $A$ is a set of action schemas, $O$ is a set of objects, $I$ is the initial state, and $G$ is the goal. Predicates $P \subseteq \{x\} \cup \{y\} \cup \{z\}$ whenever we want to explicitly declare them. The arity of $P$ is $|X_P|$. We denote individual parameters with $x, y, z \in X_P$. We can instantiate a predicate, i.e., replace the set of parameters by objects from $O$ or other variables by applying a substitution.

An action schema $a = (X_a, \text{pre}(a), \text{add}(a), \text{del}(a))$ is a tuple with a set of parameter variables $X_a$, as well as preconditions, add list, and delete list, all of which are sets of predicates in $P$ instantiated by substituting each of their variables by some element in $X_a \cup O$. As with predicates, the arity of $a$ is $|X_a|$, and we can instantiate action schemas by replacing each $x \in X_a$ by some $o \in O$ to obtain ground actions. The set of ground actions (or actions for short) is $A^O$. Note that, as the arity of predicates and action schemas is not bounded, $P^O$ and $A^O$ are of size exponential in the size of $\Pi$.

A grounded action $a$ is applicable in a state $s$ if $\text{pre}(a) \subseteq s$. The resulting state of applying $a$ on $s$ is $(s \setminus \text{del}(a)) \cup \text{add}(a)$. A sequence of actions $a_1, \ldots, a_n$ is applicable in a state $s$ if there exists a sequence of states $s_0, \ldots, s_n$ s.t. $s_0 = s$, and for all $i \in [1, k]$ $s_i$ is the result of applying $a_i$ in $s_{i-1}$. We deal with the problem of finding a plan for an arbitrary planning task $\Pi$, that is, a sequence of ground actions applicable in $I$ and resulting in some $s_n$ such that $G \subseteq s_n$.

The delete-relaxation consists of ignoring the delete list $\text{del}(a)$ of all action schemas. The FF heuristic (Hoffmann and Nebel 2001) estimates the distance from any state $s$ as the length of a relaxed plan, which can be computed in polynomial time in the size of the ground task.

Previous work by Corrêa et al. (2020) has shown that evaluating whether there exists an instantiation of an action schema that is applicable on a state is closely connected to the problem of resolving conjunctive queries in database theory (Ullman 1989). A database $DB = (D, R)$ has a domain $D$ and a set of relations $R$ over $D$, such that $R_i \subseteq D^{\times n_i}$. Following planning nomenclature, $R$ is a state $s_{DB}$ over a set of predicates $P_{DB}$ and objects $O_{DB} = D$. A conjunctive query $Q$ over $DB$ is a set of variables $X_Q$ and set of relations $R_i \in R$ instantiated with objects in $D$ and/or variables in $X_Q$, $Q$ corresponds to the problem of finding a substitution of variables in $X_Q$ by objects in $D$ such that the relations in the query belong to the database. The preconditions of an action schema can be seen as a conjunctive query that corresponds to find which instantiations of the action schema are applicable in $s_{DB}$. Evaluating conjunctive queries (hence, lifted successor generation) is NP-hard in general (Chandra and Merlin 1977), but it is tractable for acyclic conjunctive queries (Yannakakis 1981).

We say that an action schema has acyclic preconditions if the corresponding conjunctive query is acyclic. For a detailed introduction, we refer the reader to the work by Corrêa et al. (2020).

As running example we will use an extension of the visiting IPC domain, where an agent must visit all tiles in a 2D grid. We generalize this to $d$-dimensional hypercube grids with side length $l$, and we permit goals requiring to visit a subset of the locations. Figure 1 sketches the encoding of our running example for $d = 3$.

The positions in the hypercube are tuples of indices in $\mathbb{N}_l = \{i \in \mathbb{N} \mid 1 \leq i \leq l \}$. The set of all positions is $\mathbb{N}_l^d$. Similar to the original domain, the player is at some position in the beginning and can move to adjacent positions. Note that we specify a separate move- action schema for each dimension, so that we need to encode adjacency only over the numbers $\{1, \ldots, l\}$ (next predicate), not over positions (number tuples) as in the standard benchmark. Furthermore, instead of requiring the player to visit all positions, the requirement is to visit a subset of positions $G \subseteq \mathbb{N}_l^d$. This example’s ground representation is exponential in $d$ (which equals maximal predicate arity) as it needs to enumerate all possible positions so $|P^O| \geq l^d$. The same blowup occurs in the lifted task in case all positions need to be visited, i.e. if $G = \mathbb{N}_l^d$. Yet if the number of goal positions is polynomial in $d$, then the ground task is exponentially larger than the lifted task.

**Complexity of Lifted Relaxed Planning**

It is well known that a relaxed plan can be computed in polynomial time in the size of the ground task (Bonet and Geffner 2001; Hoffmann and Nebel 2001). In lifted planning, however, there are (at least) two sources of hardness:

1. The number of ground actions $|A^O|$ is exponential in lifted task size. This might incur exponential effort in determining applicable actions, a key step underlying all known relaxed planning algorithms.
2. The number of ground atoms $|\mathcal{P}|$ also is exponential in lifted task size. Hence both trivial upper bounds on relaxed plan length – number of ground atoms, number of ground actions – are not polynomial in this setting.

Indeed, delete-relaxed planning on lifted planning tasks was shown to be EXPTIME-complete (Erol, Nau, and Subrahmanian 1995). To better understand the sources of complexity at play here, we consider two further restrictions, and show that the problem is still hard (1) even if the predicate arity is restricted to be constant and (2) even if checking action applicability can be performed efficiently.

The first result follows directly from results of recent work on the problem of lifted successor generation (Corrêa et al. 2020), which showed an equivalence to answering conjunctive queries, viewing action-schema preconditions as queries over the state. Answering such a query is hard if it is cyclic in a certain sense. We can use this insight for a simple reduction from query answering to planning, in which a relaxed plan exists iff an applicable action exists in the initial state iff the answer to a query is true.

**Theorem 1.** It is NP-hard to decide relaxed plan existence in lifted planning, even if predicate arity is constant.

**Proof.** We use a reduction from conjunctive queries. Let $Q$ be a conjunctive query over a database $DB$. Consider a task $(\mathcal{P}, \mathcal{O}, \mathcal{A}, I, G)$ where $\mathcal{P} = \mathcal{P}_{DB} \cup \{goal\}$ (goal is a 0-arity predicate), $I = DB$, $G = \{goal\}$, and $A = \{a\}$ with $X_a = X_Q$, $pre(a) = Q$, $add(a) = goal$. Then the task is (relaxed) solvable if and only if some instantiation of $a$ is applicable on $I$: i.e., if the conjunctive query $Q$ is not empty.

For our second result, we encode a counter with $n$ binary variables, where the plan is to count from 0 to $2^n - 1$. Notably, this can be done with extremely simple action schemas, in particular ones with acyclic precondition queries, so that this source of complexity is independent from the previous one:

**Theorem 2.** There exist families of lifted planning tasks $\{\Pi_1, \Pi_2, \ldots\}$ with acyclic action-schema preconditions where delete-relaxed plans have exponential length.

---

**Proof.** We define $\Pi_n$ as $(\mathcal{P}_n, \mathcal{O}, \mathcal{A}_n, I_n, G_n)$ where $\mathcal{O} = \{o_{0, 0}, o_{1, 0}\}$, $\mathcal{P}_n = \{P(x_1, \ldots, x_n)\}$, $I_n = \{P(o_0, \ldots, o_0)\}$, $G_n = \{P(o_1, \ldots, o_1)\}$, and $\mathcal{A}_n = \{a_1, \ldots, a_n\}$.

The action schemas are $a_i(x_1, \ldots, x_{i-1})$ for $1 \leq i \leq n$, with $pre(a_i) = \{P(x_1, \ldots, x_{i-1}, o_{0, 0}, o_{i, 1})\}$ and $add(a_i) = \{P(x_1, \ldots, x_{i-1}, o_{1, 0}, o_{i, 0})\}$. Every relaxed plan has to achieve $2^n - 1$ ground atoms, applying $2^n - 1$ actions.

Open questions remain of course regarding the precise complexity of lifted delete-relaxed planning, in particular regarding upper bounds. We leave this to future work, and focus on more practical matters here. We introduce an additional relaxation that removes the complexity just identified.

**K-ary Predicate Splitting**

To simplify the computation of relaxed plans at a lifted level, we apply a relaxation based on splitting each $n$-ary predicate into several $K$-ary predicates where $K < n$ is a parameter for our approach. For a given $K$, the splitting operation $(\mathcal{P}_K)$ replaces the predicate by $(\binom{n}{K})$ sub-predicates that correspond to all possible combinations of $K$ parameters.

For example, consider the predicate $at(x, y, z)$ from our example in Figure 1. Then, $at|_1 = \{at_1(x), at_2(y), at_3(z)\}$ and $at|_2 = \{at_4(x, y), at_5(y, z), at_6(x, z)\}$. The same operation can be applied to grounded atoms in the initial state or goal as well as to action schemas by applying it to $pre$, $add$, and $del$ (e.g. see Figure 2). We also define this operation over sets of predicates, action schemas, etc., as the union of applying $(\mathcal{P}_K)$ to each individual in the set, e.g., $P|_K = \bigcup_{P \in P} P|_K$.

Based on this splitting operation, we define the $K$-ary relaxation of a lifted planning task.

**Definition 1 (K-ary Relaxation).** Let $\Pi = (\mathcal{P}, \mathcal{O}, \mathcal{A}, I, G)$ be a lifted planning task and $K$ be a constant. We define the $K$-ary relaxed task $\Pi|_K$ as a task $(\mathcal{P}_K, \mathcal{O}, \mathcal{A}|_K, I|_K, G|_K)$.

Obviously, plans for $\Pi|_K$ are not necessarily plans for $\Pi$, so this is an approximation. Observe that, together with the delete relaxation, it is an over-approximation and thus indeed constitutes a relaxation.
Theorem 3. Let \( \Pi = (\mathcal{P}, \mathcal{O}, A, T, \mathcal{G}) \) be a lifted planning task, \( K \) be a constant, and \( \Pi_K = (\mathcal{P}|_K, \mathcal{O}, A|_K, T|_K, \mathcal{G}|_K) \) be the \( K \)-ary relaxed task. Then every plan for \( \Pi \) is a delete-relaxed plan for \( \Pi_K \).

Proof. Every plan for \( \Pi \) is a delete-relaxed plan for \( \Pi \), so it suffices to show that delete-relaxed plans are preserved in \( \Pi_K \). Let \( (a_1, \ldots, a_n) \) be a delete-relaxed plan for \( \Pi \), let \( \mathcal{I} = s_0, s_1, \ldots, s_n \) be the (relaxed) states traversed by that plan in \( \Pi \), and let \( \mathcal{I}|_K = s_0, s_1', \ldots, s_n' \) be the states traversed by that plan in \( \Pi_K \). We show, by induction over \( i \), that \( a_i|_K \) is applicable in \( s_i|_K \) and \( a_i|_K \) is applicable in \( s_i'|_K \). For the base case \( i = 0 \), \( \Pi \) holds by construction. For the inductive case, say the claim holds for \( i - 1 \). Then \( s_i'|_K = s_i|_K \), so by construction of \( \pi(a_i|_K) \), \( a_i|_K \) is applicable in \( s_i|_K \). Regarding (2),

\[
\begin{align*}
s_i|_K &= [s_i|_K \cup \{a_i|_K\}]_K & \text{Def. of action application} \\
&= [s_i|_K|_K \cup \{a_1|_K\}]_K & \text{Prop. of set projection} \\
&= s_i'|_K \cup \{a_1|_K\} & \text{Induction Hypothesis} \\
&= s_i'|_K \cup \{a_1|_K\} & \text{Def. of } a|_K \\
&= s_i' & \text{Def. of action application}
\end{align*}
\]

Importantly, the same is not true without the delete relaxation: we do not have a guarantee that every plan for \( \Pi \) is a (non-delete-relaxed) plan for \( \Pi_K \). This is because, when deleting \( P|_K \) in \( \Pi_K \), we may delete split atoms associated also with other instantiations of the same predicate. For example, in a state that contains both \( P(a, b) \) and \( P(a, c) \), an action that deletes \( P(a, b) \) in \( \Pi \) deletes \( P(a, c) \) in \( \Pi_K \), so that the outcome state in \( \Pi_K \) does not contain \( P(a, c) \).

Regarding the complexity of delete-relaxed planning in \( \Pi_K \), all predicates in \( \Pi_K \) have a bounded arity of at most \( K \). So the length of a relaxed plan for \( \Pi_K \) is polynomial in the size of \( \Pi \) and the complexity source identified by Theorem 2 disappears. The complexity source identified by Theorem 1 remains valid though for \( K \geq 2 \), as answering conjunctive queries is \textbf{NP}-hard even in this case. Indeed, the action schemas resulting from 2-ary predicate splitting have cyclic preconditions. So deciding either a relaxed plan for \( \Pi_K \) exists remains hard in general. Here we exploit the case \( K = 1 \), \textit{unary} predicate splitting, where as we shall see next relaxed plans can be computed in polynomial time.

### Unary Relaxed Planning

Even though the number of ground actions in the unary-split task \( \Pi_1 \) is still exponential in the size of \( \Pi \), delete-relaxed plans for \( \Pi_1 \) can be computed in polynomial time.

The unary relaxetion heuristic (\( h'_{ur} \), Alg. 1) accomplishes this, in a manner analogous to the computation of relaxed plans in ground tasks (Hoffmann and Nebel 2001). It constructs a best-supporter function that maps each ground atom in \( \Pi_1 \) to a ground action. Starting at the initial state, the algorithm iteratively computes a larger set of reachable atoms

\[ F, \] enabling in each iteration the preconditions of best supporter actions for new atoms. The key to polynomial-time behavior is that, in contrast to the algorithms commonly used on ground tasks, we do not enumerate applicable ground actions in each step. Instead, we merely keep track of a best supporter for each ground atom. There are polynomially many ground atoms in \( \Pi_1 \), and it turns out we can identify best supporters efficiently.

\begin{algorithm}
\begin{algorithmic}[1]
\Function{GetBestSupporter}{\( \Pi \), \( P(o), F \) :}
\ForEach{\( a(x_1, \ldots, x_n) \in A \), \( i \in \{1, \ldots, n\} \), s.t. \( P(x_i) \in add(a) \)}
\label{alg:unary}
\If{\( \exists o_1, \ldots, o_n \in \mathcal{O} \) s.t. \( o_i = o \land \mathcal{P}(o_1, \ldots, o_n) \subseteq F \)}
\caption*{\textbf{Step 1:}}
\Else
\If{\( Q(x_j) \in \mathcal{P}(a) \)}
\caption*{\textbf{Step 2:}}
\EndIf
\EndIf
\EndFor
\Return any \( a(o_1, \ldots, o_n) \) s.t. \( o_j \in \mathcal{O} \), \( j \neq i \)
\EndFunction

\begin{algorithmic}
\Function{ExtractRelaxedPlan}{\( \Pi, \mathcal{G} \)}:
\State \( queue \leftarrow G \setminus T_1 \)
\State \( plan \leftarrow \langle \rangle \)
\While{\( queue \neq \langle \rangle \)}
\State \( f \leftarrow queue.pop() \)
\If{\( bs[f] \notin plan \)}
\State \( plan.append(bs[f]) \)
\EndIf
\State \( queue \leftarrow queue \cup \mathcal{P}(bs[f]) \setminus T_1 \)
\EndWhile
\State \Return \text{reverse(plan)}
\EndFunction
\end{algorithmic}
\end{algorithm}

Namely, the function \textbf{GetBestSupporter} iterates over action schemas instead of ground actions. We consider action schemas \( a(x_1, \ldots, x_n) \in A \) where \( P(x_i) \in add(a) \). Then, we check for each parameter \( j \neq i \) separately with which objects \( x_j \) can be instantiated such that the action preconditions are contained in \( F \). The check in line 18 evaluates to true if and only if \( \exists o_1, \ldots, o_{j-1}, o_{j+1}, \ldots, o_n \in \mathcal{O} \) such that \( \mathcal{P}(a(o_1, \ldots, o_{j-1}, o, o_{j+1}, \ldots, o_n)) \subseteq F \), i.e., iff there exist-
exists an instantiation of \( a \) that achieves the atom \( P(o) \) and whose precondition is contained in \( F \). This holds because, all preconditions being unary, the objects able to instantiate each parameter can be checked independently.

Once the best supporters have been chosen, relaxed plan extraction (ExtractRelaxedPlan) can easily be done in polynomial time. We process atoms one by one, starting with the goal atoms, inserting best-supporter actions into the relaxed plan and adding their preconditions to the atoms queue.

**Theorem 4.** Algorithm 1 runs in time polynomial in the size of \( \Pi \), and returns a delete-relaxed plan for \( \Pi |_1 \) iff such a plan exists.

**Proof.** Say Algorithm 1 returns a relaxed plan. This plan achieves all goals, as it contains \( \text{bs}[f] \) for all \( f \in G|_1 \), and \( f \in \text{add} (\text{bs}[f]) \). It is applicable in \( \mathcal{T} \) as, for every action included in the plan, a supporting action will be inserted for every precondition not already true in \( \mathcal{T} \). If the algorithm terminates without finding a relaxed plan (i.e., the main loop ends due to \( F = F' \)) then \( F \) contains all reachable ground atoms, so some goal atom is unreachable from \( \mathcal{T} \).

Those “for each” loops in Algorithm 1 which iterate over elements of the lifted task \( \Pi \) obviously perform a polynomial number of iterations. The same is true for all other loops because the number of ground atoms \( |P|^{|G|} \) is polynomial in \( |\Pi| \). The loops in lines 4 and 24 have at most one iteration per grounded atom, as in each iteration at least one new ground atom is added to \( F \). Regarding the loop in line 8, note that there is at most a best supporter for each goal atom \( |P|^{|G|} \), so at most polynomially many atoms are inserted in the queue. \( \square \)

Our implementation of Algorithm 1 extends \( F \) in a breadth-first manner, and always chooses (line 19) the object in \( O \), whose preconditions \( Q(o) \) for all \( Q(X_j) \in \text{pre}(a) \) were inserted first. In other words, among all possible supporters, we choose the ones that have minimal \( h^{\text{max}} \) value in \( \Pi |_1 \) (Bonet and Geffner 2001).

**Disambiguation with Static Predicates**

We next devise an optimization leveraging static predicates to obtain a better heuristic function. To motivate this, consider again the running example in Figures 1 and 2. Under unary relaxation, the heuristic value is at most \( 3 * |G| \), regardless of which positions need to be visited in the goal, because we can move from any coordinate to any other coordinate (e.g., move-

\( x \) \( (1, 1, 1, 3) \) is applicable in the initial state, going from \( x \)-coordinate 1 to 3 in a single step). This happens because we split not only the \( at \) predicate, but also the \( next \) predicate used to determine which numbers are adjacent to each other.

Ideally, we would like to at least obtain something resembling Manhattan distance, still separating the dimensions (by splitting the \( at \) and \( visited \) predicates), but capturing movements within each dimension correctly. To achieve the latter, we must preserve the adjacency information in \( next \). It turns out that this is indeed possible while still keeping the computational cost at bay, i.e., while preserving independence across the parameters of each action schema.

We modify the GetBestSupporter function in Algorithm 1, through a refined version of object collection at each position \( j \) in the second foreach loop. Say we need to support the atom \( P(o) \), with action schema \( a(x_1, \ldots, x_n) \) and \( i \in \{1, \ldots, n\} \) such that \( P(x_i) \in \text{add}(a) \). Our modification replaces the full set of objects \( O \) assigned to \( O_j \) in line 15 by a more restricted set \( O(a(x_i=o,j)). \) That set contains only those objects which, when \( x_i \) is instantiated with \( o \), can instantiate \( x_j \) while satisfying the static predicates. Precisely, let \( P_{st} \) be the set of static predicates, i.e., \( P_{st} \in \mathcal{P} \) such that \( P_{st} \not\in \text{add}(a) \) for any \( a \in \mathcal{A} \). Let \( \text{pre}_{st}(a, i, j) \) with \( i, j \in [1, n] \) be the set of static preconditions of \( a(x_1, \ldots, x_n) \) and pairs of sub-indices that correspond to \( x_i \) and \( x_j \), \( \text{pre}_{st}(a, x_i, x_j) = \{(P_{st}, y, z) \mid P_{st} \in P_{st}, P_{st}(x_1', \ldots, x_i') \in \text{pre}(a), x_i' = x_i, x_j' = x_j \}. \) In our example, \( \text{pre}_{st}(\text{move}-x, -x, x') = \{(\text{next}, 1, 2)\} \) as there is a precondition with the static predicate next having \( x \) has first and \( x' \) as second argument.

Define: \( O_{a(x_i=o,j)} := \bigcap \{P_{st}, y, z\in\text{pre}_{st}(a,x_i,x_j)\} \{o' \in O \mid \exists o_1, \ldots, o_t \text{ s.t. } P_{st}(o_1, \ldots, o_t) \in \mathcal{T}, a_{o_1} = o_2, a_{o_2} = o'\}. \)

We denote the resulting heuristic function with \( h^{ur-d} \).

For example, say we need to achieve \( a_{\text{at}}(3) \), and consider \( \text{move}-x(x, y, z, x') \) with \( x' = 3 \). In the previous version of Algorithm 1, the set of objects associated with the first argument \( j = 1 \) will be simply \( O \), allowing to move to 3 from anywhere. In our refined algorithm, that object set is \( \{2, 4\} \) due to the static precondition \( \text{next}(x, x') \). The relaxed plan for our running example then is \( \text{move}-x(1, 1, 1, 2), \text{move}\)-\( x(2, 1, 1, 3), \text{move}-y(1, 1, 1, 2), \text{move}-z(1, 1, 1, 2), \text{move}-z(1, 1, 2, 3), \text{move}-z(1, 1, 3, 4), \) resulting in heuristic value \( h^{ur-d}(\mathcal{T}) = 6 \).

Note that this is only a (tractable) approximation of the set of instantiations valid according to the static predicates when using predicate splitting with \( K = 2 \) for static predicates and \( K = 1 \) for the rest. We are instantiating each parameter independently, and therefore the set of objects associated with each parameter can be computed in polynomial time, at expenses of admitting instantiations that would not satisfy the static predicates in the original problem or even the \( K = 2 \) relaxation. Note further that one could apply this disambiguation to non-static predicates as well. But that would require to re-compute the set of objects, not only for every state during search, but also at each iteration of the algorithm anytime the set \( F \) changes. Restricting the disambiguation to static predicates, in contrast, allows us to pre-compute the sets of objects for each action schema, object, and parameter position once before the search starts, with respect to \( \mathcal{T} \) instead of \( F \).

**Experiments**

We implemented \( h^{ur} \) and the static disambiguation variant \( h^{ur-d} \) on top of the Power Lifted (PWL) planner (Corrêa et al. 2020), which uses Breadth First Search (BFS) and Greedy Best-First Search (GBFS) with goal counting (\( h^{gs} \)) (Fikes and Nilsson 1971). Apart from GBFS with \( h^{ur} / h^{ur-d} \), we also consider a combination with goal count-
We design new benchmarks for categories (b) and (c). We extend standard IPC domains, aiming for large instances that are hard to ground, but with simple enough goals such that some instances can be solved by current lifted planners. We scale the instances by parameters controlling task size and goal complexity, allowing us to observe how the performance of different planners is affected.

For (b), we create new variants of Visitall and Childsnack, which have a naturally scalable dimensionality parameter other consequences; and (c) large object universe, which can be problematic even for small arity domains. The benchmark set will be made publicly available.

Our benchmarks of category (a) are simply the ones previously used to evaluate hard-to-ground planning (Areces et al. 2014; Correia et al. 2020). These consist of three domains: Genome Edit Distance (GED) (Haslum 2011), Organic Synthesis (Masoumi, Antoniuzzi, and Soutchanski 2015) and Pipesworld-Tankage (Hoffmann et al. 2006). We include GED here for historical reasons only: it actually is not that widely used to evaluate hard-to-ground planning (Areces et al. 2014; Correia et al. 2020). These consist of three domains: Genome Edit Distance (GED) (Haslum 2011), Organic Synthesis (Masoumi, Antoniuzzi, and Soutchanski 2015) and Pipesworld-Tankage (Hoffmann et al. 2006). We include GED here for historical reasons only: it actually is not that hard to ground, and FD’s pre-process succeeds on all its instances.
that controls predicate arity. The Visitall extension is our running example. We create instances with $d \in \{3, 4, 5\}$ dimensions. For each of these cases we control the difficulty of instances by changing the number of goal locations from 1 to 3 and their relative position with respect to the starting location, close or far. For each of these categories we create 10 instances by scaling the size of the hypercube, starting at $l = 6$ and increasing $l$ in each instance by 2 (for $d = 5$), 4 (for $d = 4$), or 6 (for $d = 3$) to reach hard instances in all categories.

In Childsnack one has to prepare sandwiches, where some children may eat only certain kind of ingredients (e.g. gluten-free) (Fuentetaja and de la Rosa 2016). The dimensionality parameter $n$ is the number of contents on each sandwich (modeled as a predicate $P(s, c_1, \ldots, c_n)$), which is normally fixed but which we scale here. Each child has preferences, e.g., allowing only tomatoes and salad. We create different variants scaling the number of children (3, 5, and 7), which is also the number of goals. In each category, we scale task size by increasing the amount of contents available, as well as more generous preferences for the children.

Finally, for (c) we include huge instances of IPC Blocksworld, Logistics, and Rovers, keeping the goal simple enough so that some tasks are within reach for current lifted planners (Ridder and Fox (2014) ran a similar experiment, but the benchmarks are not publicly available).

For Blocksworld, we scale the number of blocks from 100 to 1900, increasing by 200 blocks per instance. In the initial state, all blocks are placed on the table (we experimented with arbitrary initial states but were unable to find instances too hard to ground yet within reach of lifted planners). For Logistics, all tasks contain one city, one airplane, one truck, and ten packages. We scale the number of locations starting with 1000 and increasing by 250 in each instance. For Rovers, we generated tasks with a single rover, one objective and one camera. We scale the number of waypoints starting from 1000 and increasing by 500 in each instance.

Results

Table 1 shows coverage results. L-RPG is not competitive, which must be interpreted with care given the implementation differences. We remark that, as intended in our design, our heuristic functions are very fast. Indeed, the node generation rate (number of generated states per second) is almost the information loss in unary splitting. Our new heuristics all do better, but particularly with disambiguation. Interestingly, the tie-breaking combination with $h^{sc}$ works best by far, hinting that our heuristics are unstable and profit from the clear progress identified by reaching more goal atoms. The picture in Childsnack is similar, except here there is no “goal location distance” parameter that we could scale, and $h^{sc}$ is hopeless throughout. Finally, results in the large IPC domains are mixed. In Logistics, our new heuristics are uninformative and fall far behind $h^{sc}$.

In Blocksworld and Rovers we obtain substantially better search information however. In Rovers, $h^{ur}$ achieves best results, the only case where disambiguation is systematically detrimental.

Conclusion

Delete-relaxation heuristics are paramount in classical planning, yet take exponential time in the size of the lifted planning task input. We have introduced a possible remedy, using additional relaxations to achieve polynomial-time behavior. Our results with a first simple technique, unary relaxation, are highly promising and already show that the state of the art can be improved. Exciting avenues opened by this research are, e.g., larger tractable fragments of predicate splitting, flexible splitting onto arbitrary sets of parameter tuples, clever methods for choosing such sets, etc.

References


