Generalized Planning via Abstraction: Arbitrary Numbers of Objects

León Illanes, Sheila A. McIlraith
Department of Computer Science
University of Toronto, Toronto, Canada
{lillanes, sheila}@cs.toronto.edu

Abstract

We consider a class of generalized planning problems based on the idea of quantifying over sets of similar objects. We show how we can adapt fully observable nondeterministic planning techniques to produce generalized solutions that are easy to instantiate over particular problem instances. We also describe how we can reformulate a classical planning problem into a quantified one. The reformulation allows us to solve the original planning task without grounding every action with respect to all objects in the problem, and a single solution can be applied to a possibly infinite set of related classical planning tasks. We report experimental results that show our approach is a practical and promising technique for solving an interesting class of problems.

1 Introduction

The potential for broad applicability of sequential decision making, and in particular for AI automated planning, is burgeoning with increased automation in key sectors including transportation, fulfillment, and e-commerce. A common property of many of these decision-making tasks is that they are addressing the same basic problem from day to day. For example, each day, UPS™ must decide how to ship packages of varying numbers, sizes, and types, to varying locations, via different modes of transportation. Fulfillment centres similarly restock shelves daily. What changes from day to day is the number and types of objects, how they need to be moved, and where they are shelved in the fulfillment centre. These businesses typically operate at a massive scale. In 2015 UPS delivered roughly 4.7 billion packages to 10 million customers, via 110,000 delivery vehicles and 1,955 daily flight segments (UPS 2016).

If we look at these decision-making problems as a collective, we see that UPS's package delivery problems, or Amazon's restocking problems, each correspond to a *family* of problems that, when viewed abstractly, share a common solution form. Solving such problems using automated planning has two limitations: the first is that automated planning treats each problem instance as a new planning problem without leveraging patterns that are common to all solutions. More critically, while AI planning algorithms continue to advance in speed and efficiency, they cannot solve problems of the massive scale demanded by these industries.

In this paper we advance a vision for a new approach to sequential decision making that leverages the power and efficiency of state-of-the-art automated planning techniques, while exploiting abstraction and generalization to automatically generate what can be seen as domain-specific, but instance-independent planners that can be instantiated – offline or at execution time – to perform sequential decision making at a massive scale for families of planning problems. These domain-specific and instance independent planners are compact, simple programs with looping structures and conditionals that follow different control flow depending on the problem instance.

Our work realizes this vision by leveraging ideas from a number of different areas of research including generalized planning (e.g., (Levesque 2005; Hu and Levesque 2009; Srivastava, Immerman, and Zilberstein 2008; Hu and Levesque 2010; Hu and De Giacomo 2011)), numeric planning (e.g., (Srivastava et al. 2011; Illanes and McIlraith 2017)), and techniques for recognizing indistinguishable objects in planning domains (Riddle et al. 2015; 2016; Fuentetaja and de la Rosa 2016).

We introduce LOOM, a planning algorithm that solves families of problems where separate instances differ in the number of objects present. Our approach ignores the distinctions between certain objects of the same type and computes a cyclic policy for the resulting abstract problem. The resulting policy is a form of domain-specific but instance-independent planner that can be applied over concrete problem instances. We evaluate LOOM over a suite of generalized planning problems. The results show our approach is capable of quickly finding general solutions, and that these can be instantiated into plans for specific problems orders of magnitude faster than planning from scratch, without significant changes in plan quality.

2 Preliminaries

The families of planning problems we consider can be viewed as a form of generalized planning problem (e.g., (Srivastava et al. 2011; Hu and De Giacomo 2011)). We in turn define these generalized planning problems in terms of *basic planning problems*. Here, a basic planning problem comprises a domain description and specifics of the problem instance. A basic planning problem *domain*, $\mathcal{D}$ is a tuple $\langle T, P, A\rangle$, where:

- T is a set of (possibly hierarchical) types,
- P is a set of (typed) predicates, and
- A is a set of action schemata.

Each action schema $a \in A$, is defined by a tuple $a = \langle \texttt{params}(a), \texttt{pre}(a), \texttt{eff}(a)\rangle$, where:

- $\texttt{params}(a)$ is a sequence of typed variables called the parameters of a, and
- $\texttt{pre}(a)$ and $\texttt{eff}(a)$, respectively known as the preconditions and effects of a, are sets of first-order literals built using the predicates in P and the variables in $\texttt{params}(a)$.

For convenience, we will refer to the set of atoms that appear as positive literals in the effects of an action as $\texttt{eff}^+(a)$. The set of atoms that appear in negative literals will be referred to as $\texttt{eff}^-(a)$.

Finally, a *basic planning problem* is a tuple $\mathcal{P} = \langle \mathcal{D}, O, I, G\rangle$, where:

- $\mathcal{D}$ is a planning domain $\langle T, P, A\rangle$;
- O is a set of typed object constants, using the types in T;
- I, the initial state of $\mathcal{P}$, is a set of ground atoms in first-order logic, built using the predicates in P and the object constants of O;
- G, the goal condition of $\mathcal{P}$, is a set of ground literals built in the same way as the atoms in I.

A *state* is a truth assignment to each predicate in P ground with the subset of object constants in O that respect the predicate types. In keeping with the convention established in classical planning, a state is parsimoniously represented by the set of positive literals. Those literals not mentioned are interpreted as false, following a closed-world assumption. I is defined in this manner.

A ground formula that is restricted to a conjunction of literals can also be represented parsimoniously as a set of literals and corresponds to the set of states for which this conjunctive formula holds. We refer to such a representation as a *partial state*. The goal condition, G, is defined in this manner. We use the notation $S \models S'$ to denote that the condition captured by partial state S' holds in state S.

Given an action schema $a \in A$ and the set of object constants O, we can build a *ground action* by substituting object constants of the appropriate type for every parameter of a. Using the sequence of object constants $\mathbf{o} = (o_0, o_1, \ldots)$, we will refer to this ground action as $a(\mathbf{o}) = a(o_0, o_1, \ldots)$.

Now, given a ground action $a(\mathbf{o})$ and some state S, we say that $a(\mathbf{o})$ is applicable over S if and only if $S \models \texttt{pre}(a(\mathbf{o}))$. The state that results of applying $a(\mathbf{o})$ over state S is given by the following transition function:

$$\delta(S, a(\mathbf{o})) = (S \smallsetminus \texttt{eff}^-(a(\mathbf{o}))) \cup \texttt{eff}^+(a(\mathbf{o}))$$

Note that the transition function can easily be adapted to work over partial states.

Running example (Package delivery domain)**.** To avoid confusing the reader with unnecessary detail, we consider a very simple domain in which a number of packages must be delivered from a source location to either of two other locations, A or B. The packages are transported by a truck, and can be loaded and unloaded from it at any of the three locations. In a more detailed variant of this problem, we would have types for package, truck, and location. Here we limit ourselves to typing packages using specially named predicates for the different locations and the truck.

We specify the domain as $\mathcal{D} = \langle T, P, A\rangle$, where:

$$\begin{aligned}
T = &\ \{\texttt{pkg}\}\\
P = &\ \{\texttt{at-src(pkg)}, \texttt{at-A(pkg)}, \texttt{at-B(pkg)},\\
&\ \texttt{in-truck(pkg)}, \texttt{truck-at-src},\\
&\ \texttt{truck-at-A}, \texttt{truck-at-B}\}\\
A = &\ \{\texttt{load-at-src(pkg)}, \texttt{unload-at-src(pkg)},\\
&\ \texttt{load-at-A(pkg)}, \cdots, \texttt{drive-A-B}\}
\end{aligned}$$

In the interest of space, we provide a single example of an action, $a = \texttt{load-at-A(pkg)}$:

$$\begin{aligned}
\texttt{params}(a) &= \{p\colon \texttt{pkg}\}\\
\texttt{pre}(a) &= \{\texttt{at-A}(p), \texttt{truck-at-A}\}\\
\texttt{eff}(a) &= \{\neg\texttt{at-A}(p), \texttt{in-truck}(p)\}
\end{aligned}$$

The example illustrates what we believe is a common issue in the application of planning techniques to real-world problems. Oftentimes, a user will need to solve a single problem multiple times with slight variations on the initial state (total number of packages, and how many are destined for each location), but the same high-level goal description (every package must be at its destination). There is, then, a need for a system that can either take advantage of the work done in previous iterations, or one that effectively produces a general solution that can easily be applied to every instance.

In this work, we envision the latter case, and imagine a user that will describe the problem in a general way, establishing the high-level goal and the common properties of all possible initial states. The system will then produce a general solution that is guaranteed to work on any problem instance that satisfies the conditions established by the generalized problem. Verifying whether a given instance does match the general problem can be easily done, and instantiating the general solution into a specific solution for the instance is much faster than planning for the instance from scratch. In the next sections we will describe an adequate formalism for specifying these tasks and an algorithm for producing the solutions. In addition, in Section 6, we briefly outline a method for reducing the modeling burden by inferring a generalized problem out of a single classical instance.

3 Generalized Planning Problems

A generalized planning problem $\mathcal{P}$ is informally defined as a (possibly infinite) set of basic planning problems, so that $\mathcal{P} = \{\mathcal{P}_0, \mathcal{P}_1, \ldots\}$. In this section we describe how such sets can be characterized succinctly by defining a class of generalized planning problems that we call *quantified planning problems*. Such quantified planning problems will serve as the input specification to our solver.

We define a quantified planning problem as a tuple $\mathcal{P}_q = \langle \mathcal{D}_q, O_q, I_q, G_q\rangle$, where:

- $\mathcal{D}_q$ is a *quantified planning domain*,
- O_q is a set of (non-quantifiable) typed objects,
- I_q is the initial *quantified state*, and
- G_q is the *quantified goal condition*.

In turn, a quantified planning domain is defined as a tuple $\mathcal{D}_q = \langle T, T_q, P, A\rangle$, where:

- T is a set of types,
- T_q is a set of *quantifiable types* that refine the types of T,
- P is a set of typed predicates, and
- A is a set of action schemata.

A quantified state is defined by a set of first-order logic atoms built using the predicates in P and the constants from O_q, and a set of existentially quantified atoms built using the same predicates and constants in addition to distinct variables used for quantification.

The quantified goal condition similarly consists of a set of first-order logic literals constructed from P and O_q, and a distinct set of universally quantified atoms drawn from P.

Note that in all cases the predicates are typed, so the quantification must be type-bounded. To represent this, we define the following notation:

$$\forall[x\colon p(x)]\ \varphi \overset{def}{=} \forall x.\, p(x) \rightarrow \varphi$$

$$\exists[x\colon p(x)]\ \varphi \overset{def}{=} \exists x.\, p(x) \wedge \varphi$$

The sets of types, predicates, and action schemata are defined exactly in the same way as for the basic planning problems described in Section 2. That said, some care must be taken in the case of ground actions. Although they are defined in the same way as before, their effects differ slightly.

Consider for example a quantified state in which the fact $\exists[x\colon \texttt{type}(x)]\ \texttt{P}(x)$ holds, and some action that has the effect $\neg\texttt{P}(c)$. If c was the only witness for the existentially quantified fact held, then the result of applying this action over this state is a state in which the fact does not hold. Otherwise, if c was just one among many objects for which the fact held, then the result is a state in which the fact still holds. As such, the application of an action can have multiple possible outcomes, and this observation is key to the realization of our solver. Formally, we deal with this form of uncertainty by modifying the transition function so that it produces the set of all the quantified states that may be reached by applying a given ground action over a given quantified state.

Running example (Generalized package delivery)**.** Consider the simple planning domain example $\mathcal{D} = \{T, P, A\}$ described in Section 2 and imagine a system that is meant to perform this same task every day, but with different numbers of packages meant to go to each destination. The corresponding quantified domain can be given by $\mathcal{D}_q = \langle T, T_q, P, A\rangle$, where the types in T_q refine T by having one type of package for each destination location:

$$T_q = \{\texttt{pkg-for-A}, \texttt{pkg-for-B}\}$$

The generalized planning problem is given by $\mathcal{P}_q = \langle \mathcal{D}_q, O_q, I_q, G_q\rangle$ where:

$$\begin{aligned} O_q = &\quad \varnothing, \\ I_q = &\quad \{\texttt{truck-at-src}, \\ &\qquad \exists[a\colon \texttt{pkg-for-A}(a)]\ \texttt{at-src}(a), \\ &\qquad \exists[b\colon \texttt{pkg-for-B}(b)]\ \texttt{at-src}(b)\}, \text{ and} \\ G_q = &\quad \{\forall[a\colon \texttt{pkg-for-A}(a)]\ \texttt{at-A}(a), \\ &\qquad \forall[b\colon \texttt{pkg-for-B}(b)]\ \texttt{at-B}(b)\}. \end{aligned}$$

Here, there are no objects other than the packages, which will be quantified. Furthermore, the packages are partitioned into two sets by means of the type refinement: those that are to be taken to location `A` and those that must be taken to `B`. The initial state description establishes that the truck is at the source location, and that there is at least one package meant for `A` and at least one package meant for `B` at the source location. The goal states that every package meant for location `A` (or `B`) must indeed be at `A` (or `B`).

3.1 Corresponding Basic Planning Problems

As mentioned above, a generalized planning problem is meant to represent a set of basic planning problems. Here we give a precise description of the basic planning problems that correspond to a particular quantified planning problem.

As the notation suggests, all the basic problems that correspond to the quantified problem $\mathcal{P}_q = \langle\langle T, T_q, P, A\rangle, O_q, I_q, G_q\rangle$ share the same domain $\mathcal{D} = \langle T, P, A\rangle$. The set of basic planning problems is formed by all problems of the form $\mathcal{P}_i = \langle \mathcal{D}, O_i, I_i, G_i\rangle$, such that:

- $O_i \supseteq O_q$,
- every $o \in O_i \setminus O_q$ is of a type $t \in T$ that is refined by some type $t_q \in T_q$,
- $I_i \models I_q$, and
- $G_i \models G_q$ and every $g \in G_i$ is either already present in G_q or is a witness for some existentially quantified $g' \in G_q$.

3.2 Generalized Plans, Policies and Solutions

As we have highlighted before, the goal of generalized planning is to find a single plan that can be applied to every problem in a given set. Since it is infeasible to expect that a singular sequence of actions would be an adequate solution for multiple planning problems, a generalized plan must include either a procedure to generate a specific plan for a given instance or a system that allows the online execution of the generalized plan over the instance. Note that in the absence of uncertainty over the outcomes of actions, the two approaches are effectively equivalent since the execution of the generalized plan may be simulated to obtain a plan.

In our work, the solutions to our quantified planning problems – our generalized plans – take the form of partial policies. In general, a policy is a function that maps from states to actions. In this case, we are interested in policies that map from partial quantified states to partially grounded actions. The instantiation of such a policy into a plan for a specific instance requires simulating its execution. At every step, we must perform a very restricted search over the objects in the instance in order to find some which allow full grounding of

$$\Pi(S) = \begin{cases} \texttt{load-at-src(pkg-for-A)} & \text{if } \texttt{truck-at-src} \in S \wedge \exists[a\colon \texttt{pkg-for-A}(a)]\ \texttt{at-src}(a) \in S \\ \texttt{load-at-src(pkg-for-B)} & \text{if } \texttt{truck-at-src} \in S \wedge \exists[a\colon \texttt{pkg-for-A}(a)]\ \texttt{at-src}(a) \notin S \\ & \wedge \exists[b\colon \texttt{pkg-for-B}(b)]\ \texttt{at-src}(b) \in S \\ \texttt{drive-src-A()} & \text{if } \texttt{truck-at-src} \in S \wedge \exists[a\colon \texttt{pkg-for-A}(a)]\ \texttt{at-src}(a) \notin S \\ & \wedge \exists[b\colon \texttt{pkg-for-B}(b)]\ \texttt{at-src}(b) \notin S \\ \texttt{unload-at-A(pkg-for-A)} & \text{if } \texttt{truck-at-A} \in S \wedge \exists[a\colon \texttt{pkg-for-A}(a)]\ \texttt{in-truck}(a) \in S \\ \texttt{drive-A-B()} & \text{if } \texttt{truck-at-A} \in S \wedge \exists[a\colon \texttt{pkg-for-A}(a)]\ \texttt{in-truck}(a) \notin S \\ \texttt{unload-at-B(pkg-for-B)} & \text{if } \texttt{truck-at-B} \in S \wedge \exists[b\colon \texttt{pkg-for-B}(b)]\ \texttt{in-truck}(b) \in S \end{cases}$$

Figure 1: A policy that solves the generalized package delivery problem. The goal is to take every package of type `pkg -for-A` to the `A` location, and every package of type `pkg -for-B` to the `B` location. In the initial state, every package is at the source location. Note that the actions selected by the policy are lifted, and refer to types instead of specific objects. The actual execution of the policy requires inspecting the current state to select appropriate objects to ground the actions.

the action. Given a quantified planning problem and an associated policy, the policy is a solution if and only if it can produce a plan for every instance in the set of corresponding basic planning problems.

An interesting property that results from using policies as solutions is that a given policy can be used to solve problems other than those that exactly correspond to the generalized problem. Consider a given policy that solves a specific generalized planning problem, and consider a separate problem which shares the same goal. If the initial state of this new problem is contained within the envelope of the policy, then the policy is also a solution for it.

Running example (Policy for generalized package delivery problem)**.** The policy Π defined in Figure 1 is a solution for the running example. The execution of the policy over any instance results in a plan that first loads every package that must go to `A` onto the truck, then loads every package that goes to `B`, then drives the truck to `A`, unloads every package that must be at `A`, drives to `B`, and finally unloads every package that must be at `B`. The specific order in which the packages of the same type are loaded or unloaded from the truck is not described by the policy.

4 Finding Generalized Solutions

To synthesize a policy that solves a given generalized planning problem, our algorithm repeatedly repairs an existing weak partial policy that is guaranteed to be terminating. The algorithm is finished when the repairs result in the policy being goal-closed. At this point the policy has been proven to be a strong cyclic solution for the problem. To give precise details about the main algorithm and its key subprocedures, we first need to define how a policy is implemented and how its termination can be verified by representing it as a graph.

Following Fully Observable Nondeterministic (FOND) planner, PRP (Muise, McIlraith, and Beck 2012), we define the implementation of a partial policy in terms of a *rule set* $\mathcal{R}$ and a selection function Φ. $\mathcal{R}$ is a set of pairs of partial states and actions, and $\mathcal{R}(S)$ denotes the subset of state-action pairs $\langle S', A\rangle \in \mathcal{R}$ such that $S \models S'$. Φ is a partial function that selects a single action from a set of state-action pairs. Then, a partial policy can be defined as $\Pi(S) = \Phi(\mathcal{R}(S))$, and we will say that any state S for which $\Pi(S)$ is defined is a state that is *handled* by Π. With this formalism, a plan can be generalized into a partial policy by repeatedly regressing from the reached state back towards just before the first action in the plan. At each step, this produces a partial state and an action, so the result is a rule set. Finally, we use the selection function that always chooses the action that is paired with the partial state that is closest to the goal. The formalism also allows us to easily combine multiple such policies together by simply using the union of their rule sets.

The policy Π defined by rule set $\mathcal{R}$ and selection function Φ can be represented as a directed graph with vertices for each partial state S such that there is some pair $\langle S, A\rangle \in \mathcal{R}$ and two additional vertices, $\top$ and $\bot$, that respectively represent all goal states and all unhandled states. There is an edge between the vertices corresponding to partial states S and S' if and only if there is some state in $\delta(S, \Pi(S))$ that satisfies S'. Similarly, there is an edge from S to $\top$ if there is a goal satisfying state in $\delta(S, \Pi(S))$, and there is an edge from S to $\bot$ whenever there is an unhandled state in $\delta(S, \Pi(S))$. We say this edge is labeled by the action $\Pi(S)$.

The main procedures used by our algorithm are described in Algorithm 1 and Algorithm 2. At a high level, the algorithm works by exploring the states that are reachable by the current policy (loop in lines 6–14). Whenever it reaches a state that cannot be handled, it attempts to update the policy (line 11). The update procedure works by repeatedly generating a weak plan that can reach from the unhandled state to the goal or to some other already handled state (line 20). The plan is subsequently regressed into a weak policy (line 25), and an attempt is made to verify that integrating this policy onto the main one will result in a terminating policy (line 27). If this verification is successful, the main policy is updated (line 28). Otherwise, the algorithm continues to find alternative weak plans to deal with the unhandled state. If no plan is found, the state is marked as a deadend (line 22) and the repair fails. If the initial state is recognized as a deadend, then we know that our system cannot find a generalized solution for the problem at hand. When this occurs in our implementation, we return the best policy found so far.

Algorithm 1: The LOOM planning algorithm

```
1  Algorithm LOOM
     Input: A generalized planning problem
            $\mathcal{P} = \langle\langle T, P, A\rangle, I, G\rangle$
     Output: A solution policy $\Pi$ for $\mathcal{P}$
2    Initialize $\Pi$ to the empty policy.
3    repeat
4      Open $\leftarrow \{I\}$
5      Seen $\leftarrow \varnothing$
6      repeat
7        $S \leftarrow$ Open.pop()
8        if $S \not\models G$ and $S \notin$ Seen then
9          Seen.add($S$)
10         if $\Pi(S) = \bot$ then
11           UPDATE($\Pi, S, A, G$)
12         if $\Pi(S) \neq \bot$ then
13           Open $\leftarrow$ Open $\cup\ \delta(S, \Pi(S))$
14     until Open is empty
15     PROCESSDEADENDS()
16   until $\Pi$ does not change
17   return $\Pi$
18 Procedure UPDATE
     Input: A policy $\Pi$, a state $S$, a set of actions $A$, a
            goal $G$
19   repeat
20     $\pi \leftarrow$ NEXTWEAKPLAN($S, A, G, \Pi$)
21     if $\pi = \bot$ then
22       Mark $S$ as a deadend.
23       return
24     else
25       $\Pi' \leftarrow$ REGRESS($\pi$)
26     $\Gamma \leftarrow$ GRAPH($\Pi + \Pi'$)
27   until TERMINATES($\Gamma$)
28   $\Pi \leftarrow \Pi + \Pi'$
```

As with PRP, the outer loop of the main algorithm (lines 3–16) ensures that the process only finishes when no unhandled states remain, or when there is no solution. Deadends can be handled by forbidding state-action pairs that result in deadends, and restarting the process from scratch without using them. If the initial state is detected to be a deadend, then the algorithm returns no solution.

Finally, the test for termination works by decomposing the graph corresponding to the policy into its strongly connected components, and recursively verifying that it is impossible to infinitely loop within one of these. For a single strongly connected component, the process consists of removing all edges that are labeled with actions that delete some quantified atom that is not added by some other edge in the same component. If the resulting graph is acyclic, then the policy is known to terminate. Otherwise, removing some edges can cause it to no longer be strongly connected and the process can be applied recursively. Effectively, the procedure guarantees that the graph does not have a cycle in which atoms

Algorithm 2: The termination verification procedure used by LOOM

```
1  Procedure TERMINATES
     Input: A graph $\Gamma$
     Output: TRUE only if $\Gamma$ corresponds to a
             terminating policy
2    repeat
3      Remove $e$ from $\Gamma$.
4    until there is no edge $e$ in $\Gamma$ that deletes some
       quantified atom that is not added elsewhere in $\Gamma$
5    if $\Gamma$ is acyclic then
6      return TRUE
7    if no edge was removed then
8      return FALSE
9    foreach strongly connected component $\Gamma'$ of $\Gamma$ do
10     if $\neg$TERMINATES($\Gamma'$) then
11       return FALSE
12   return TRUE
```

are deleted and added again, so that during execution the same concrete states are visited over and over, in an infinite loop. Note that the process is correct but not necessarily complete, and some terminating policies may be rejected.

Lemma 1 (Soundness of TERMINATES.)**.** *The execution of* TERMINATES(Γ) *itself always terminates, and returns* TRUE *only if* Γ *corresponds to a terminating policy.*

Proof sketch. The proof is based on the proof of Theorem 4 in (Srivastava et al. 2011), which addresses policies for qualitative numeric planning problems, where actions nondeterministically increment and decrement a set of numeric state variables and where the goal is to set all of them to zero. The proof operates by induction over the recursive step. As a base case consider an input graph Γ_0 which results in returning immediately (line 6) after removing some edges (lines 3–4). We know the only cycles in Γ_0 are ones in which some quantified atom is deleted without being added again. In any concrete problem instance, repeatedly deleting these quantified atoms will eventually result in all of them being deleted. Then, the quantified statement must become false and the cycle must terminate. Therefore, every cycle in Γ_0 eventually terminates, and so Γ_0 represents a terminating policy.

Now consider an arbitrary graph Γ that does not become acyclic after removing edges. TERMINATES(Γ) will return TRUE only if every strongly connected component of Γ is proven to terminate (lines 9–11). By induction, the procedure is correct for every such component. If every component must terminate, then there are no possibly infinite loops in Γ, and the policy represented by it is terminating. ☐

Theorem 1 (Soundness of LOOM)**.** *A policy* Π *returned by* LOOM *for problem* $\mathcal{P}$ *is a solution for* $\mathcal{P}$.

Proof sketch. From 1, any policy returned by LOOM terminates. LOOM returns when *Open* is empty and there are no deadends to process. At this point, every state reachable by the policy has been inserted into *Open* at some point in the last iteration of the outer loop. Every state that was removed from *Open* is either a goal state or is handled by Π. Since every terminal state reachable by Π is a goal state and Π is known to terminate, then Π must be a solution for $\mathcal{P}$. $\square$

5 Executing Generalized Policies

So far, we have described a formalism in which a user may specify a generalized planning problem, and an algorithm that can correctly produce general policies for problems so specified. Now, given a particular basic planning problem, the user will be interested in two things. First, the user must establish whether the basic planning problem belongs to the generalized problem for which the policy was computed. Subsequently, the user will want to instantiate the generalized policy over the basic problem, in order to obtain an executable plan.

Verifying whether the given instance matches the general problem can be easily done by ensuring that the instance satisfies the definition of a corresponding basic planning problem given in Section 3.1. The actual instantiation of the policy can be done online, by directly executing it over the basic planning problem. At each step, evaluating the policy over the current state will produce a partially grounded action, in which some of the action parameters refer to ground objects, and some refer to existentially quantified ones. By construction, finding a fully grounded action that is applicable to the current state is always possible, and can be done simply by searching over the space of objects that match the types of the parameters until finding a mapping that results in an executable ground action.

6 Generalizing a Basic Planning Problem

Modeling a real world problem as a generalized planning problem requires identifying which types of objects to quantify over. This can be challenging for some. Consider extending our simple running example to more accurately represent a distribution network by allowing for a large number of different locations that are not all directly connected to each other. It is easy to see that no memoryless policy can generalize to arbitrarily connected locations, as any solution would have to implement a graph search algorithm. Nonetheless, the request may seem natural to an uninformed end user, who would model the problem with quantified locations and would finally obtain no solution.

To ease this modeling issue, we propose and describe a method that builds a quantified planning problems out of an single basic planning problem example. As expected, the method produces quantified planning problems that only quantify types that can effectively be quantified. Furthermore, the description of the task clearly specifies which basic planning problems correspond to it.

The approach is based on existing reformulation techniques that have been designed to address symmetry issues in classical planning. In particular, our work uses the techniques described by Fuentetaja and de la Rosa (2016) and by Riddle et al. (2016). At a very high level, both of these works strive to reduce symmetry in a given planning problem by identifying sets of indistinguishable objects. They then reformulate the problem to model those sets by counting its elements instead of representing every object explicitly.

For our purposes, we can use the exact same techniques to identify sets of indistinguishable objects. Each of these sets is then to be represented by a new quantifiable type that refines the actual type of the objects. This results in a quantified planning problem that represents problems in which the number of elements in those sets is different.

7 Experimental Evaluation

We implemented our main LOOM algorithm by modifying the existing codebase for FOND planner PRP[1], which in turn is based on the implementation of the Fast Downward Planning System (Helmert 2006). The modifications include adapting the parser to deal with quantified planning problems, implementing the termination verification procedure, and many other lesser modifications. The use of the existing systems allows us to take advantage of advances in FOND and classical planning algorithms and heuristics.

In addition, we implemented a system for instantiating a policy into a plan for a particular basic planning problem instance. This process works by simulating the execution of the policy over the problem. At every step, the policy suggests a partially grounded action, and the system searches among the objects of the problem for some that satisfy the preconditions of the action in the current state. The resulting ground action is applied over the state, and the process is repeated. The trace obtained from this execution corresponds to a plan for the problem.

Finally, we also implemented the generalization process described in Section 6. This implementation is based on the work done for Baggy (Riddle et al. 2016).

In general, the performance of a generalized planning system has multiple dimensions, and there are important tradeoffs expected between them (Srivastava, Immerman, and Zilberstein 2011). In particular, we are interested in measuring (1) how long it takes for our system to produce a generalized solution for a given generalized problem, (2) how quickly we can test whether or not that solution is applicable to a given basic planning problem, (3) how long it takes to instantiate the solution into a plan, and (4) the quality (length) of the resulting instantiated plan. More generally, we are interested in empirically verifying that our approach is feasible, and that in many cases it is actually beneficial to compute generalized solutions instead of solving the basic planning problems separately.

To this end, we benchmarked our algorithms over a set of generalized planning problems. The set includes generalized planning problems from existing literature (Recycling), well-known classical planning problems that can be modeled with quantification (Logistics, Hamburger), and some new domains specifically crafted to showcase the advantages and possible pitfalls of our approach (Construction, Roundabout). In all cases, the modeling assumes full observability and deterministic actions.

[1]https://bitbucket.org/haz/planner-for-relevant-policies

	Loom				Lama-First	
Problem (Size Range)	Solving time (s)	Policy size	Instantiation time (s) (normalized) [speedup]	Plan length (normalized)	Planning time (s) (normalized)	Plan length (normalized)
Recycling (10–200)	0.03	10	5.39 [55%]	6	11.99	6
Logistics (5–1000)	0.53	28	0.04 [-33%]	6.93	0.03	6.93
Hamburger (10–100)	0.03	14	0.05 [81%]	13	0.26	15.03
Construction (5–100)	0.17	19	0.10 [93%]	18	1.47	19.4
Roundabout (8–48)	297.89	11	0.004 [33%]	3.25	0.006	3.25

Table 1: Summary of the results obtained by LOOM and LAMA-FIRST over a suite of benchmark generalized planning problems. The reported instantiation times, planning times, and plan lengths correspond to the averages over multiple basic planning problems, normalized by the number of quantifiable objects in them. The range of problem sizes appears in parentheses on the first column. In most domains, the quality of the instantiated solutions obtained with LOOM is comparable to that obtained with LAMA-FIRST, but the instantiation time is significantly smaller. The relative speedup of instantiating a policy instead of planning from scratch is shown in square brackets.

As a baseline to compare with, we use a trivial generalized planner that essentially ignores the generalized problem description and, when presented with a basic planning problem, uses a classical planner to find a plan. Since we are interested in finding solutions quickly, rather than finding optimal solutions, the baseline uses a greedy best first search algorithm using the FF (Hoffmann and Nebel 2001) and LAMA (Richter and Westphal 2010) heuristics (i.e., the LAMA-FIRST configuration of Fast Downward).

A summary of our results is displayed in Table 1. In most of our benchmarks, finding a generalized solution is surprisingly easy. An explanation for this is that the quantified problems essentially amount to non-symmetrical reformulations of the basic planning problems, and finding a solution policy is only slightly harder than finding a solution for the smallest corresponding basic planning problem.

An interesting exception is the Roundabout domain (in which vehicles are routed one at a time through a roundabout), where our system is actually unable to prove that the solution it builds is guaranteed to terminate. Although the resulting solution can be proven to terminate by hand, the fast and incomplete termination procedure we use to do this automatically is incapable of doing so.

Finally, instantiating an existing policy over a given basic planning problem is, in most cases, significantly faster than planning from scratch. Furthermore, for massive instances, a classical plan would be prohibitive in size. The abstract policy presents a compact encoding of the solution that can be instantiated on demand at execution time. The resulting plans are of comparable quality to those obtained using a fast satisficing planner. This suggests a confirmation of the general merits of our approach, as it is clear that there are domains for which finding and repeatedly applying general solutions results in better performance than the traditional alternative.

A specific case of results is shown in Figure 2. There, we show how in the Construction domain a policy obtained by LOOM is significantly more efficient than planning from scratch for every instance. Results for other domains exhibit similar basic profiles.

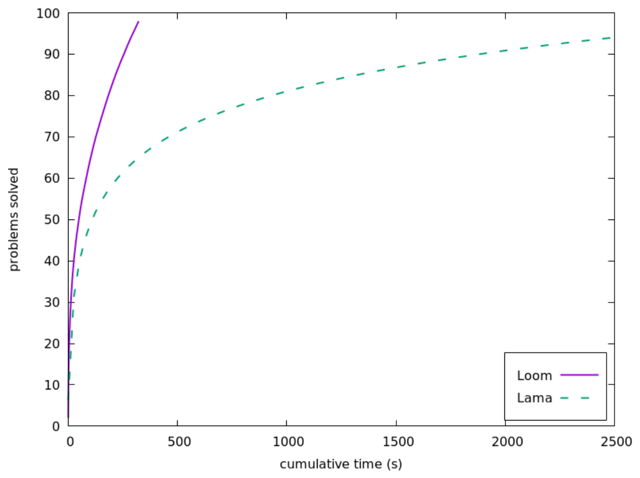

Figure 2: Cumulative number of problems solved over time by both approaches in the Construction domain. The policy computed by LOOM solves all problems in under 6 minutes. Planning from scratch every time takes over 40 minutes. Experiments in other domains result in similar performances.

8 Related Work

Our work is situated within the context of generalized planning, but borrows many ideas and techniques from a number of other areas in planning and artificial intelligence. The broad idea of doing abstraction by grouping objects together has been explored for symmetry breaking in planning tasks (Fox and Long 1999), and has more recently been used for problem decomposition (Abdulaziz, Norrish, and Gretton 2015) and in the problem reformulation approaches that we mentioned in Section 6 (Fuentetaja and de la Rosa 2016; Riddle et al. 2015; 2016).

An interesting perspective regarding symmetry breaking is that most work on that topic focuses on imposing a canonical order over the possible symmetric solutions. In contrast, our work provides a system for formulating problems in such a way that a solution represents a family of symmetric ground solutions. The process of actually selecting a ground solution from the generalized one is easy, and the generalized solution can also be used in an online manner.

An idea that relates to the use of abstraction by object aggregation is used for producing generalized solutions out of example plans by the planner Aranda (Srivastava, Immerman, and Zilberstein 2008). There, objects are grouped together whenever they satisfy the same set of unary predicates, and the imprecisions that arise in the problem's transition system are dealt with by doing 3-valued logic analysis (Sagiv, Reps, and Wilhelm 2002).

Recent work has focused on modeling the imprecisions and uncertainty produced by abstraction as unfair fully-observable nondeterministic effects. One such example is a line of work concerned with the synthesis of finite-state controllers for conformant and generalized planning (Bonet and Geffner 2015; Bonet et al. 2017), where the unfairness is handled by enforcing relevant trajectory constraints. A different example is the use of abstract policies to guide search in domains with numeric state variables (Illanes and McIlraith 2017), where the issues with unfairness are dealt by falling back to traditional search methods whenever the guidance is ineffective. In our work, we directly exploit the fact that we do have a model for how the unfair nondeterminism resolves and implement a termination verification process into the search for a solution.

The specific termination algorithm we use is based on the one used for the verification of policies for qualitative numeric planning problems, where actions arbitrarily and nondeterministically increment or decrement a set of numeric state variables (Srivastava et al. 2011). A key distinction stems from the fact that in our work actions are explicitly deterministic. We only use nondeterminism to model the results of applying the same action over different states that happen to be indistinguishable in our representation formalism, whereas the actions in their setting do actual nondeterministic effects. This has a significant consequence in that it means finding policies for their problems is theoretically harder, and therefore proving that a given policy may effectively loop forever is actually easier. As such, their termination verification algorithm is complete, whereas ours is not. In a further interesting contrast to our work, the policy search paradigm presented by Srivastava et al. is based on exploring the space of goal-closed policies until finding one that is guaranteed to terminate. Conversely, LOOM can be said to search the space of terminating policies until reaching one that is goal-closed.

Finally, a different proposal for reformulating generalized planning problems into fully-observable nondeterministic planning problems and then using an off-the-shelf planner for the latter class of problems was recently given in (Bonet and Geffner 2018). The work presented in that paper is framed within the broader context of dealing with generalized planning problems in which the sets of ground actions vary across the different basic planning problems. The specific problem we deal with, in which different basic problems involve different sets of objects, falls within this context. Of course, the sort of FOND planners needed to solve problems like these are those that produce strong cyclic solutions, and which are based on the assumption of fair nondeterminism. As such, the translation approach is directly applicable only to a restricted class of problems in which this assumption does not significantly mislead the planners. Put simply, this restriction limits the approach to work only on problems in which actions affect the generalized objects only in one direction, so that work done by some action is not undone by another. To use the approach in more general cases, the authors point out the need for special translations such as the one described in (Bonet et al. 2017), although they note that that particular translation is not sound. In our work, we address the impact of unfair nondeterministic actions by using the sound but incomplete termination verification procedure described in Algorithm 2. This is in no way limited to the aforementioned restricted class of problems, and is in fact complete for it.

9 Conclusions and Future Work

As discussed in Section 1, our work advances a vision for sequential decision making that recognizes the importance of domain-specific solvers and strives to design systems that can automatically generate them. We believe LOOM presents a major step towards realizing this vision. The abstract policies we generate form a sort of domain-specific planner, providing compact solutions to families of planning problems of massive (often unbounded) size, and our experiments demonstrate that they can be used to find plans significantly faster than a standard satisficing classical planner.

The endeavor and our results raise a number of important questions. First, our approach uses an efficient but incomplete method for verifying policy termination, and exploring alternatives is an interesting open avenue for research.

Furthermore, the quality of the instantiated plans is comparable to that of a satisficing planner, but in order to use something like our approach in a real-world industrial setting we would like to further optimize for quality. We believe our approach can be adapted to find richer policy structures, which can then be used with execution monitoring techniques for optimization (e.g., Fritz and McIlraith 2007).

Finally, we have argued that generalized plans can be understood to be domain-specific planners, and as such that generalized planning tasks are effectively program synthesis tasks. We believe our work – and other work in generalized planning – can be connected to many important research topics in the areas of program and software synthesis, broadly construed (e.g., (Manna and Waldinger 1980; Solar-Lezama 2008; Srivastava, Gulwani, and Foster 2010)) and that this work further advances the connection between planning and program synthesis.

Acknowledgements

We gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC), and from Microsoft Research. In addition, we wish to extend our gratitude to Xi Yan for her invaluable help in setting up and running our experiments, and to Rodrigo Toro Icarte for comments on an earlier draft.

References

Abdulaziz, M.; Norrish, M.; and Gretton, C. 2015. Exploiting symmetries by planning for a descriptive quotient. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 1479–1486.

Bonet, B., and Geffner, H. 2015. Policies that generalize: Solving many planning problems with the same policy. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2798–2804.

Bonet, B., and Geffner, H. 2018. Features, projections, and representation change for generalized planning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, 4667–4673.

Bonet, B.; Giacomo, G. D.; Geffner, H.; and Rubin, S. 2017. Generalized planning: Non-deterministic abstractions and trajectory constraints. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, 873–879.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, 956–961.

Fritz, C., and McIlraith, S. A. 2007. Monitoring plan optimality during execution. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, 144–151.

Fuentetaja, R., and de la Rosa, T. 2016. Compiling irrelevant objects to counters. special case of creation planning. *AI Communications* 29(3):435–467.

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

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

Hu, Y., and De Giacomo, G. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, 918–923.

Hu, Y., and Levesque, H. 2009. Planning with loops: Some new results. In *Proceedings of the 1st Workshop on Generalized Planning (GenPlan@ICAPS)*, 35–42.

Hu, Y., and Levesque, H. 2010. A correctness result for reasoning about one-dimensional planning problems. In *Proceedings of the 12th International Conference on Knowledge Representation and Reasoning (KR)*, 362–371.

Illanes, L., and McIlraith, S. A. 2017. Numeric planning via abstraction and policy guided search. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, 4338–4345.

Levesque, H. J. 2005. Planning with loops. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, 509–515.

Manna, Z., and Waldinger, R. 1980. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems* 2(1):90–121.

Muise, C. J.; McIlraith, S. A.; and Beck, J. C. 2012. Improved non-deterministic planning by exploiting state relevance. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 172–180.

Richter, S., and Westphal, M. 2010. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Riddle, P. J.; Barley, M. W.; Franco, S.; and Douglas, J. 2015. Automated transformation of PDDL representations. In *Proceedings of the 8th Symposium on Combinatorial Search (SoCS)*, 214–215.

Riddle, P.; Douglas, J.; Barley, M.; and Franco, S. 2016. Improving performance by reformulating PDDL into a bagged representation. In *Proceedings of the 8th Workshop on Heuristic Search for Domain-independent Planning (HS-DIP@ICAPS)*, 28–36.

Sagiv, M.; Reps, T.; and Wilhelm, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems* 24(3):217–298.

Solar-Lezama, A. 2008. *Program synthesis by sketching*. Ph.D. Dissertation, University of California, Berkeley.

Srivastava, S.; Zilberstein, S.; Immerman, N.; and Geffner, H. 2011. Qualitative numeric planning. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI)*, 1010–1016.

Srivastava, S.; Gulwani, S.; and Foster, J. S. 2010. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 313–326.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI)*, 991–997.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011. A new representation and associated algorithms for generalized planning. *Artificial Intelligence* 175(2):615–647.

UPS. 2016. UPS fact sheet. https://www.pressroom.ups.com/pressroom/ContentDetailsViewer.page?ConceptType=FactSheets&id=1426321563187-193.