Learning Looping Domain-Specific Planners from Example Plans

Elly Winner and Manuela Veloso
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891, USA
(412) 268-4801
{elly,mmv}@cs.cmu.edu

Abstract

Planners are powerful tools for problem solving because they provide a complete sequence of actions to achieve a goal from a particular initial state. Classical planning research has addressed this problem in a domain-independent manner—the same algorithm generates a complete plan for any domain specification. This generality comes at a cost; domain-independent planners have difficulty with large-scale planning problems. To deal with this, researchers have resorted to hand writing domain-specific planners to solve them. An interesting alternative is to use example plans to demonstrate how to solve problems in a particular domain and to use that information to automatically learn domain-specific planners that model the observed behavior. In this paper, we present the ITERANT algorithm for identifying repeated structures in observed plans and show how to convert looping plans into domain-specific template planners, or dsPlanners. Looping dsPlanners are able to apply experience acquired from the solutions to small problems to solve arbitrarily large ones. We show that automatically learned dsPlanners are able to solve large-scale problems much more quickly than are state-of-the-art general-purpose planners and are able to solve problems many orders of magnitude larger than general-purpose planners can solve.

Introduction

Intelligent problem solving requires the ability to select actions autonomously from a specific state to reach objectives. Planning algorithms provide approaches to look ahead and select a complete sequence of actions. Given a domain description consisting of preconditions and effects of the actions the planner can take, an initial state, and a goal, a planning program returns a sequence of actions to transform the initial state into a state in which the goal is satisfied. Classical planning research has addressed this problem in a domain-independent manner—the same algorithm generates a complete plan for any domain specification. However, domain-independent planners have traditionally had difficulty with large-scale planning problems, although many large-scale problems have a repetitive structure, because they do not capture or reason about such repetition. Instead, to solve large-scale problems, programmers have had to rely on the tedious and difficult process of hand writing special-purpose planners that may precisely encode the repetitive structure. However, example plans are often available, and can demonstrate this structure.

In previous work, we showed how to use example plans to learn non-looping domain-specific planners (or dsPlanners), which can solve problems of limited size (Winner & Veloso 2003). Here, we present the novel ITERANT algorithm to learn looping dsPlanners from individual example plans. Loops allow the learned dsPlanner to capture the repetitive structure in many large-scale planning problems and to use this structure to attack arbitrarily large problems.

DsPlanners execute independently of a general-purpose planning program and are very efficient; they return a solution plan in time that is linear in the size of the dsPlanner and of the problem, modulo state-matching effort. We show that looping dsPlanners can solve large-scale planning problems more quickly than can general-purpose planners and that they can solve much larger problems than can general-purpose planners. In our tests, the best general-purpose planners were unable to solve simple problems with more than a few hundred objects within our two-minute time limit, while learned dsPlanners were able to solve similar problems with tens of thousands of objects in seconds. And because dsPlanners are learned directly from example plans, there is no need for tedious hand coding.

In some domains, finding optimal solutions is NP-complete. Therefore, dsPlanners learned automatically from a finite number of example plans cannot be guaranteed to find optimal plans. Our goal is to extend the solvability horizon for planning by reducing planning times and allowing much larger problem instances to be solved. We believe that post-processing plans can help improve plan quality.

Identifying loops in observed plans allows the plans to be reused to quickly solve arbitrarily large similar problem instances. Our research focuses on compressing looping plans into compact domain-specific template planning programs that can solve larger and more complex problems than can current general-purpose planning techniques. However, loop identification could also be used for other purposes, such as improving the performance of case-based or analogical planning methods or identifying promising candidates for macro learning.

We first discuss related work. We then define dsPlanners and explain how we use them to generate the solution plans for new problems. Then we discuss classes of loops, de-
scribe our algorithm for identifying parallel non-nested one-variable loops in observed plans, and illustrate its behavior with examples. Next we present the results of using plans with identified loops as planners and compare this to using state-of-the-art general-purpose planners. Finally, we present our conclusions.

Related Work

Many research efforts have sought to automatically improve general-purpose planning efficiency, most commonly by using learned or hand-written domain knowledge to reduce generative planning search e.g. (Fikes, Hart, & Nilsson 1972; Minton 1988; Kambhampati & Hendler 1992). We focus here on methods that learn and exploit repeated structure within plans.

Case-based and analogical reasoning approaches attempt to avoid generative planning entirely for many problems (Hammond 1996; Kambhampati & Hendler 1992; Leake 1996). Entire plans are stored and indexed as cases for later retrieval. When a new problem is presented, the case-based reasoner searches through its case library for similar problems. If an exact match is found, the previous plan may be returned with no changes. Otherwise, the reasoner must either try to modify a previous case to solve the new problem or to plan from scratch. Many case-based reasoners handle libraries of tens of thousands of cases (Veloso 1994a), but as the libraries get larger, the search times for relevant cases can exceed the time required to plan from scratch for a new case. It is also difficult to find an appropriate similarity metric between problems and to determine when it would be faster simply to plan for the new problem from scratch. However, case-based planners have succeeded in solving larger problems than can be solved by generative planning alone, and, in general, find solutions faster than generative planners (Hammond 1996).

A variant of case-based reasoning that deserves mention is analogical reasoning, which also stores case libraries and attempts to modify previous cases to solve new problems (Veloso 1994a; 1994b). However, in addition to storing the problem and the plan, analogical reasoners also store the problem-solving rationale behind each plan step. This makes it easier to modify previous cases to solve new problems. However, deciding when to abandon modification and plan from scratch is still a problem, as are retrieving cases and attempting checking the condition-action rules against the current state and goal. When a rule applies, the action is attempted, and matching begins again from the beginning of the list.

The decision list algorithm can find a strategy even when the plans it is given cannot be described by a simple strategy (e.g., they are optimal solutions to NP-hard problems). However, the original decision list work by Khardon is able to solve fewer than 50% of 20-block Blocksworld problems, and requires background knowledge about the domain and several thousand state-action pairs to achieve that coverage (Khardon 1999). This work was later extended by Martin and Geffner, who used a different language to describe the rules of the decision list. This allowed them to eliminate the need for background knowledge, to learn policies from several hundred examples, and to achieve coverage of over 70% of 20-block problems. However, their approach only addresses domains with unary actions.

The decision list approach requires so many examples in part because by breaking up example plans into state-action pairs, it disposes of much of the information contained in the plans, such as sequencing and looping behaviors. We believe that by preserving this information we will be able to achieve broad coverage from as few as one example. Enumerating all possible conditions and evaluating them against every observed state-action pair may become prohibitively slow for more complex domains than Blocksworld.

Finally, many researchers have explored hand writing domain-specific planners, e.g., (Bacchus & Ady 2001; Nau et al. 2003). These planners are able to solve larger problems than can general-purpose planners, and are able to solve them more quickly (Long & Fox 2003), but often require months or years to create.
Defining and Using DsPlanners

In this section, we explain in detail the form of the dSPlan-ners our algorithm learns and how they are used for planning.

Defining DsPlanners

A dSPlanner is a domain-specific planning program that, given a planning problem (initial and goal states) as input, either returns a plan that solves the problem or returns fail-

either returns a plan that solves the problem or returns failure, if it cannot do so. DsPlanners are composed of the following programming constructs and planning-specific operators:

- while loops and endwhile statements;
- if, then, else, and endif statements;
- logical structures (and, or, not);
- in_goal_state and in_current_state operators;
- numbered and typed variables;
- the “v” variant indicator for while loops;
- plan predicates; and
- plan operators.

In order for dSPlanners to capture repeated sequences in while loops and to determine that the same sequence of operators in two different plans has the same conditions, they must update a current state as they execute by simulating the effects of the operators they add to the plan. Without this capability, we would be unable to use such statements as: while (condition holds) do (body). Therefore, in order to use a dSPlanner, it must be possible to simulate the execution of the plan. However, since dSPlanner learning requires full models of the planning operators, this is not an additional problem.

Variables are introduced in if-statement and while-loop conditions. Any objects in the problem which match the conditions may be assigned to the variables. Those assignments hold throughout the conditions and body of the if statement or while loop. While-loop variable assignments hold throughout the iterations of the loop unless the variable is labelled “v” for variant, in which case it may be reassigned at each iteration.

DsPlanner 1 shows a dSPlanner that solves all gripper-domain (Long 2000) problems involving moving balls between rooms. The dSPlanner is composed of one while loop: while there is an ball that is not at its goal location, move to the ball (if necessary), pick up the ball, move to goal location of the ball, and drop the ball.

DsPlanner 2 shows a dSPlanner that solves block-stacking problems in the BlocksWorld domain (Long 2000). The task is accomplished with three while loops. The first unstacks all blocks. The second stacks the bottom block of each eventual tower. The third loop stacks the subsequent blocks in each tower.

Planning with DsPlanners

Our algorithm for generating plans from dSPlanners is shown in Algorithm 1. As previously mentioned, while executing the dSPlanner, we must keep track of a current state and of the current solution plan. The current state is initialized to the initial state, and the solution plan is initialized to the empty plan. Executing the dSPlanner consists of applying each of the statements to the current state. Each statement in the dSPlanner is either an plan step, an if statement, or a while loop. If the current statement is a plan step, make sure it is applicable, then append it to the solution plan and apply it to the current state. If the current statement is an if statement, check to see whether it applies to the current state. If it does, apply each of the statements in its body; if not, go on to the next statement. If the current statement is a while loop, check to see whether it applies to the current state. If it does, apply each of the statements in its body until the conditions of the loop no longer apply. Then go on to the next statement.

Sometimes there may be many ways to apply an if statement or a while loop to the current state. For example, if we have a statement like, “if in_current_state (not-eaten(?1:apple)) then eat(?1)”, and there are several uneaten apples in the current state, it is unclear which apple should be eaten. However, one of our primary assumptions is that all objects that match the conditions may be treated
Plan

DsPlanner

Input:

Algorithm 1  DsPlanner-based plan generation

Output: Plan $P$ that solves the given problem.

$P \leftarrow \emptyset$

for all statements $S_n$ in $T$ do

$P \leftarrow P + \text{Apply Statement}(S_n, C, G)$

end for

if $G$ is satisfied by $C$ then

return $P$

else

FAIL

end if

Statement

Input:

ment to the current state

Algorithm 2  Apply Statement: Applying a dsPlanner statement to the current state

Output: Plan fragment $P$ generated by applying $S$ to $C$ and $G$. In the process of applying it, $C$ is changed.

$P \leftarrow \emptyset$

if $S$ is an if statement then

if $\text{Applies Now}(S, C, G)$ then

for all statements $S_i$ in the body of $S$ do

$P \leftarrow P + \text{Apply Statement}(S_i, C, G)$

end for

end if

else if $S$ is a while statement then

while $\text{Applies Now}(S, C, G)$ do

for all statements $S_i$ in the body of $S$ do

$P \leftarrow P + \text{Apply Statement}(S_i, C, G)$

end for

end while

else if $S$ is a plan step then

if not $\text{Applicable}(S, C)$ then

FAIL

end if

$C \leftarrow \text{Apply Step}(S, C)$

$P \leftarrow S$

end if

return $P$

Identifying Loops in Example Plans

The current version of the ITERANT algorithm identifies all non-nested parallel loops over one variable in an observed plan. In the remainder of this section, we discuss some relevant definitions, describe in detail the two main portions of the ITERANT —identifying loop candidates and creating a loop from a candidate)—and illustrate the operation of ITERANT with two examples.

Definitions

Subplans are connected components within in a partially-ordered plan when the initial and goal states are excluded (otherwise every set of steps would be a connected component). They are illustrated in Figure 1.

Matching Subplans are subplans that satisfy the following criteria:

• they are non-overlapping,
• they consist of the same operators,
• the operators in each subplan are causally linked to each other in the same way,
• they have the same conditions and effects in the plan,
• they unify.

We also use the term “matching steps” as a special case of matching subplans (in which the subplans are of length one). The two load operators in Figure 1 are matching steps (as shown in Figure 3, as are the two paint operators.

Parallel Subplans are causally- and threat-independent of each other. Figure 2 shows three parallel subplans within an example plan.

An Unrolled Loop is a set of matching subplans. One of two unrolled loops in the painting and transport example is shown in Figure 3.

A Loop replaces an unrolled loop in the plan. The body of the loop consists of the common subplan in the unrolled loop, but with the differing variable converted into a loop variable. The conditions on its execution are that the goal state contains all goal terms that are supported by steps within the unrolled loop and that the current state when the loop is executed contains all the conditions for the steps within the unrolled loop to execute correctly and support the goals of the plan. The loop represented by the unrolled loop shown in Figure 3 is shown in Figure 4.

A Parallel Loop is a loop in which each iteration of the loop is causally independent from the others—the iterations may be executed in any order. The loop shown in Figures 3 and 4 is a parallel loop. A loop may also have a multi-step body with complex causal structure; it may even in-

the same, so, in this case, it doesn’t matter which apple is eaten.

There are three ways a dsPlanner may fail to generate the correct solution plan. It may have run through the whole dsPlanner and found no solution steps at all, though the initial state is not the same as the goal state. Or, it may have found some plan steps to execute, but, by the end of the dsPlanner, did not reach the goal state. Finally, it may have found some plan steps to execute, but found that they were not applicable to the current state. A failure is detected when we attempt to execute steps that are not applicable in the current state or when the dsPlanner finishes executing and its final state does not match the goal state. The way we currently handle failures is by handing the problem off to a generative planner, and then adding that new solution to the dsPlanner.
Figure 1: An example plan in a painting and transport domain is shown. In the given plan, some objects need to be painted and some need to be loaded into a truck. Painting must be done before loading. Three different subplans are surrounded by dotted lines. There are many other possible subplans, but the steps paint(obj1) and paint(obj3) are not a subplan, since they are not a connected component within the partial ordering.

Figure 2: Three parallel subplans are surrounded by dotted lines.

Figure 3: Two matching subplans of length 1 are surrounded by dotted lines and represent an unrolled loop.

clude other loops. The current version of ITERANT is able to identify all non-nested parallel loops in observed plans.

A Serial Loop is a loop in which each iteration of the loop is causally linked to the others—there is a specific order in which the iterations must be executed. For example, in a package-transport domain, one loop may describe a particular delivery vehicle visiting different locations, loading and unloading packages at each one. Each iteration of the loop consists of loading and unloading packages and then mov-
Figure 4: The painting and transport problem after the load loop is identified. The loop is surrounded by dotted lines. The loop variable is written as ?1, and ranges over all values that meet the conditions of the loop (in this case, obj1 and obj2). Conditions for the loop are shown above the loop.

The ITERANT Algorithm

The ITERANT algorithm can handle domains with conditional effects, but we assume that it has access to a minimal annotated consistent partial ordering of the observed total order plan. Previous work has shown how to find minimal annotated consistent partial orderings of totally-ordered plans given a model of the operators (Winner & Veloso 2002) and has shown that STRIPS-style operator models are learnable through examples and experimentation (Carbonell & Gil 1990).

The ITERANT algorithm, formalized in Algorithm 3, first identifies an unrolled loop (described in the Section “Identifying Unrolled Loops”) and then converts it into a loop (described in the Section “Converting Unrolled Loops into Loops”). The unrolled loop is then removed from the plan and replaced by the loop.

Algorithm 3 ITERANT: Identify non-nested one-variable parallel loops in an observed plan.

Input: Minimal annotated partially ordered plan $\mathcal{P}$.
Output: $\mathcal{P}$ with all non-nested one-variable parallel loops identified.

for all steps $i$ in $\mathcal{P}$ do
  $M_i \leftarrow$ all parallel matching steps with $i$ in $\mathcal{P}$
  if $M_i \neq \emptyset$ then
    $C \leftarrow$ LargestCommonSubplan($M_i + i$, $\mathcal{P}$)
    $L \leftarrow$ MakeLoop($C$)
    $\mathcal{P} \leftarrow \mathcal{P} - C$
    $\mathcal{P} \leftarrow \mathcal{P} + L$
  end if
end for

Identifying Unrolled Loops

The first step in the ITERANT algorithm is to identify a parallel unrolled loop: a set of parallel matching subplans within the observed plan. This process begins with the identification of a set of parallel matching steps, as described in Algorithm 3. Next, ITERANT finds the largest parallel matching subplan common to at least two of those steps. This process takes place in the procedure LargestCommonSubplan, formalized in Algorithm 4. LargestCommonSubplan recursively tries every possible expansion of the existing subplan and returns the one with the most steps per parallel track. First, it identifies the sets of steps that supply conditions to the steps in each parallel track of the existing subplan (StepBack) and the set of steps that rely on effects of the steps in each parallel track of the existing subplan (StepAhead). The initial and goal states are not considered as steps ahead or back. Then, it explores each of these steps as a possible way to expand the subplan. For each step in StepBack and StepAhead for each track, it finds which other tracks also have a matching step in StepBack or StepAhead. If there is at least one other track, the current subplans with the new steps added are recorded as a new unrolled loop. At the end of this process, there is a set of new unrolled loops. LargestCommonSubplan is then recursively applied to each of these to further expand them. The largest resulting candidate is then returned by the algorithm as the final unrolled loop.

Converting Unrolled Loops into Loops

Once an unrolled loop is identified, it must be converted into a loop. As previously defined, an unrolled loop is a set of matching subplans differing in only one variable. The body of the loop is the subplan—with a new loop variable replacing the differing variable. The conditions for the loop’s execution are requirements on the goal state and on the current state while the loop is executing, as described in the Section “Definitions”. The unrolled loop subplans are then removed from the plan and replaced by the new loop.

A Rocket-Domain Example

We will now describe the operation of the ITERANT algorithm on a simple example plan from the rocket domain, illustrated in Figure 5. First, ITERANT searches for sets of parallel matching steps. It finds the steps load(o1, r,
Algorithm 4 LargestCommonSubplan: Identify largest parallel matching subplans of an observed plan common to at least two of the given parallel matching subplans.

**Input:** set $A$ of parallel matching subplans $S_1..S_m$, minimal annotated partially ordered plan $P$.

**Output:** Set of largest parallel matching subplans of plan $P$ common to at least two of $S_1..S_m$.

- for all $S_i$ in $S_1..S_m$ do
  - $StepAhead_{S_i}$ ← steps causally linked from $S_i$
  - $StepBack_{S_i}$ ← steps causally linked to $S_i$
- end for

$UnrolledLoops$ ← $A$

for $i = 1$ to $m$ do
  - for all $s$ in $StepAhead_{S_i}$ do
    - for all $j \neq i$ do
      - if $\exists$ parallel matching step $s'$ in $StepAhead_{S_j}$ then
        - $NewExpLoop$ ← $NewExpLoop + \{S_i + s'\}$
        - $StepBack_{S_j}$ ← $StepBack_{S_j} - s'$
      - end if
    - end for
  - if $|NewExpLoop| > 1$ then
    - $UnrolledLoops$ ← $UnrolledLoops + NewExpLoop$
    - $NewSteps_{S_i}$ ← $NewSteps_{S_i} - s$
  - end if
- end for

end for

for all sets $N \neq A$ in $UnrolledLoops$ do
  - $N$ ← LargestCommonSubplan($N$, $P$)
end for

return set $N$ in $UnrolledLoops$ with the largest subplan.

Algorithm 5 MakeLoop: Create the loop described by the given unrolled loop.

**Input:** Unrolled loop: set of matching subplans $S_1..S_m$, minimal annotated partially ordered plan $P$.

**Output:** The loop described by $S_1..S_m$.

- let $v_i$ be the variable in $S_i$ that $v_j$ is not in $S_j$
- let $v_{loop}$ be the loop variable
- $Loop.body$ ← $S_1$ with $v_{loop}$ replacing $v_1$
- $Loop.conditions$ ← $\emptyset$
- for all steps $s$ in $Loop.body$ do
  - for all conditions $c$ of $s$ not satisfied by steps in $Loop.body$ do
    - $Loop.conditions$ ← $Loop.conditions + GoalStateContains(c)$
  - end for
- end for

A Multi-Step Loop Example

The ITERANT algorithm is also able to detect multi-step loops. We now describe its operation on an example plan from an artificial domain, illustrated in Figure 6. First, ITERANT searches for a set of parallel matching steps. It finds the steps $op1(x)$ and $op1(y)$, which differ only in the values $x$ and $y$. These two one-step parallel matching subplans are then sent to LargestCommonSubplan, which searches for a larger subplan common to both of them.

LargestCommonSubplan begins by finding the $StepAhead$ set for each parallel track. There is one step in $StepAhead$ for each track: the corresponding unload operator. The step $fly(r, s, d)$ is not a possible step ahead since it is not causally linked to the load operators. LargestCommonSubplan also finds the $StepBack$ set for each track. It is empty; since these are the first three steps in the plan and are parallel to each other, they do not depend on any other plan steps. The unload steps cannot be added to the subplans, although they are matching, since they are not threat-independent. LargestCommonSubplan thus returns the original one-step subplan.

A new loop is then created to represent the common one-step subplan. The loop body is created by replacing the differing values ($o1$, $o2$, and $o3$) with the new loop variable, $lv1: load(lv1, r, s)$. The conditions of the loop are that the current state satisfies the conditions of the steps within it ($at(lv1, s)$ and $at(r, s)$) and that the goal state contains the goals supported by the steps in the loop body ($at(lv1, d)$).

This process repeats to uncover the unload loop, and the resulting plan is shown in Figure 5(b).

Figure 6: An example annotated partially ordered plan in an artificial domain that includes a multi-step loop consisting of the steps $op1(x)$ and $op1(y)$, which differ only in the values $x$ and $y$. These two one-step parallel matching subplans are then sent to LargestCommonSubplan, which searches for a larger subplan common to both of them.

$\text{LargestCommonSubplan}$ begins by finding the $\text{StepAhead}$...
set for each parallel track. There is one step in StepAhead for each track: op3(x) and op3(y), respectively. There are no elements in the StepBack set, since neither of these steps depends on any other plan step. Because adding these steps preserves the parallelism and matching of op1(x) and op1(y), they can be added to the subplans. This is the only way to expand the original subplans, and so is the only element in the list of unrolled loops.

LargestCommonSubplan is then executed recursively on this new set of subplans. There are now no elements in StepAhead for any track, but there is one in StepBack for each parallel track: op2(x) and op2(y), on which op3(x) and op3(y) depend. Adding these steps also preserves the parallelism and matching of the existing subplans, so they are added as well. Again, this is the only way to expand the given subplan. LargestCommonSubplan is executed one last time on this new loop expansion and is unable to find any possible “steps ahead” or “steps back,” so this loop expansion is returned.

A new loop is then created to represent the common branching three-step subplan. The loop body is assigned to the common subplan, with a new loop variable, lv, replacing the differing values, x and y. The conditions of the loop are that the current state satisfies the conditions of the steps within it (s(lv)) and that the goal state contains the goals supported by the steps in the loop body (g(lv)). The resulting plan is shown in Figure 7.

Figure 5: An example plan in the rocket domain that involves moving objects o1, o2, and o3 from location s to location d using rocket r. The minimal annotated partial ordering of the plan is shown in (a). The plan after loops are identified is shown in (b). Loops are surrounded by dotted lines. The loop variables are written as lv1 and lv2, and range over all values that meet the conditions of the loops (in these cases, o1, o2, and o3). Conditions for the loops are shown above them.

Figure 7: The example plan shown in Figure 6 after the loop has been identified. The loop is surrounded by dotted lines. The loop variable is written as lv, and ranges over all values that meet the conditions of the loop (in this case, x and y). The conditions of the loop are shown above it.

Using Looping Plans as Domain-Specific Planners

Here, we briefly describe how to convert a looping plan into a looping dSPlanner capable of solving similar problems of arbitrary size. First, the plan is parameterized: values are replaced by variables. The planner is a total ordering of the partially ordered plan. Loops are described as while state-
ments: while the conditions for the loop hold, execute the body of the loop. We describe how to identify loops and their conditions in the section, “Identifying Loops in Example Plans”. Plan steps not contained within loops are part of if statements: if the conditions of the steps hold, execute the steps. The conditions of a set of steps are the current-state terms required for the steps to execute correctly and support the goals of the problem and the goal-state terms that the steps support. We describe how to execute dsPlanners in the section, “Defining and Using DsPlanners”.

Results

We compare general-purpose planning to planning using learned looping dsPlanners.4 To illustrate the effectiveness of identifying loops in plans, our tests focus on performance on large-scale problems of the same form as the example plans. We show that the learned dsPlanners capture the structure of the example plans and are able to apply this knowledge very efficiently to solving much larger problems. In these situations, planning using dsPlanners scales orders of magnitude more effectively than does general-purpose planning.

Rocket-Domain Results

The dsPlanner learned from the rocket domain example shown in Figure 5 is shown in dsPlanner 3. The problems on which we tested the planners vary in the number of objects but consist of the same initial and goal states: the initial state consists of at(rocket, source), and for all objects obj in the problem, the initial state contains at(obj, source) and the goal state contains at(obj, destination). Figure 8 shows the results of executing several different general-purpose planners and the learned dsPlanner on large-scale problems of this form. The learned dsPlanner is orders of magnitude more efficient on large problems than these general-purpose planners, and is able to solve problems with more than 60,000 objects in under a minute.

Multi-Step Loop Domain Results

The dsPlanner learned from the multi-step loop domain example shown in Figures 6 and 7 is shown in DsPlanner 4. As with the rocket domain, the problems on which we tested the planners vary in the number of objects but consist of the same initial and goal states: for all objects obj in the problem, the initial state contains s(obj) and the goal state contains g(obj). Figure 9 shows the results of executing several different general-purpose planners and the learned dsPlanner on large-scale problems of this form. The learned dsPlanner scales much better to large problems than these general-purpose planners, and is able to solve problems with as many as 40,000 objects in under a minute.

4We used the latest versions of several of the best-performing general-purpose planners from the third international planning competition in 2002: VHPOP version 3.0, MIPS version 3, FF version 2.3, and LPG version 1.2.1.

DsPlanner 3 dsPlanner based on the rocket domain problem shown in Figure 5. The variable in each loop is indicated by a “v” preceeding its name.

```
while in_current_state (at(?v1:object, ?2:location))
    and in_current_state (at(?3:rocket, ?2:location))
    and in_goal_state (at(?v1:object, ?4:location))
    do
    end while

if in_current_state (at(?1:rocket ?2:location))
    and in_current_state (in(?3:object ?1:rocket))
    and in_goal_state (at(?3:object ?4:location))
    then
        fly(?1:rocket ?2:location ?4:location)
    end if

while in_current_state (in(?v1:object, ?2:rocket))
    and in_current_state (at(?2:rocket ?3:location))
    and in_goal_state (at(?v1:object, ?3:location))
    do
    end while
```

Figure 8: Timing results of several general-purpose planners and of the learned dsPlanner shown in DsPlanner 3 on large-scale rocket-domain delivery problems. All timing results were obtained on an 800-MHz pentium II with 512 MB of RAM.

DsPlanner 4 DsPlanner based on the multi-step loop domain problem shown in Figures 6 and 7.

```
while in_current_state (s(?v1:type1))
    and in_goal_state (g(?v1:type1)))
    do
        op1(?v1:type1)
        op2(?v1:type1)
        op3(?v1:type1)
    end while
```

Conclusion

In this paper, we contribute the ITERANT algorithm for automatically recognizing template planners from example plans in a specific domain. In particular, we focus on identifying loops in observed plans and on converting looping plans into looping domain-specific template planning programs (dsPlanners). The ITERANT algorithm identifies loops by
finding sets of parallel matching subplans and then converting each set into a loop. Our results show that the looping dsPlanners learned by the ITERANT algorithm are able to take advantage of the repeated structures in some planning problems and solve those problems more quickly than can current state-of-the-art general-purpose planners. In these situations, planning using dsPlanners scales more effectively than general-purpose planning and extends the solvability horizon by solving much larger problems than general-purpose planners can handle.

The applications of this work are much broader than rapid action selection. Agents operate in a world populated by other agents, both human and machine. A fundamental task of these intelligent systems is to reason about the behavior of the agents around them so they can interact appropriately. Our work on extracting algorithmic models of behavior from observed executions could allow computers to be programmed by demonstration, allowing anyone, not just trained professionals, to program computers to perform complex tasks. It could help create general-purpose robots that could be trained to do a new task in minutes, simply by watching it be done. It could facilitate the cooperation of heterogenous agents by allowing them to quickly build models of each others’ behavior, or could allow agents to predict and avoid the troublesome behavior of adversarial or non-cooperative agents. It could allow software to predict accurately and pre-execute commands for users, or even to automatically complete tasks like planning travel and scheduling meetings based on observations of the user’s preferences.

References


