Learning Template 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

June 26, 2004

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-specific 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 identifing 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 state-of-the-art general-purpose planners and are able to solve problems many orders of magnitude larger than general-purpose planners can solve.

1 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 statisfied. 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 repetative 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 repeated structure. However, example plans are often available, and can demonstrate this structure.

In previous work, we introduced the concept of automatically-generated domain-specific template planning programs (or dsPlanners) and showed how to use example plans to learn non-looping dsPlanners, which can solve problems of limited size [11]. Here, we present the novel ITERANT algorithm for automatically identifing the repeated structure of example plans to learn looping dsPlanners that model the behavior demonstrated in the observed plans. 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. And because dsPlanners are learned directly from example plans, there is no need for tedious hand coding.

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. Then we discuss classes of loops, describe 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.

2 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. [3]. We focus here on methods that learn and exploit repeated structure within plans.

Case-based and analogical reasoning, e.g., [9], apply planning experience from solving previous problems to solving a new one. Similarly, the internal analogy technique [4] reuses the planning experience gleaned from solving one part of a particular problem to solving other parts of the same problem.

Iterative and recursive macro operators and control rules, e.g., [7], capture repetitive behavior and can drastically reduce planning search by encapsulating an arbitrarily long string of operators. However, unlike our approach, this technique does not attempt to replace the generative planner, and so does not eliminate planning search.

Some work, like our own, has focussed on analyzing example plans to reveal a strategy for planning in a particular domain. One example of this approach is BAGGER2, which learns recurrences that capture some kinds of repetition [8]. BAGGER2 was able to learn recurrences from very few example plans, but relied on background knowledge and wasn't able to capture parallel repetition.

Another example of the strategy-learning approach is the decision list [5]: a list of condition-action pairs derived from example state-action pairs. This technique also relies on background knowledge, is able to solve fewer than 50% of 20-block Blocksworld problems, and requires over a thousand state-action pairs to achieve that coverage [5].

Finally, many researchers have explored hand writing domain-specific planners, e.g., [1]. These planners are able to solve more problems than general-purpose planners, and are able to solve them more quickly [6], but often require months or years to create.

3 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.

3.1 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.

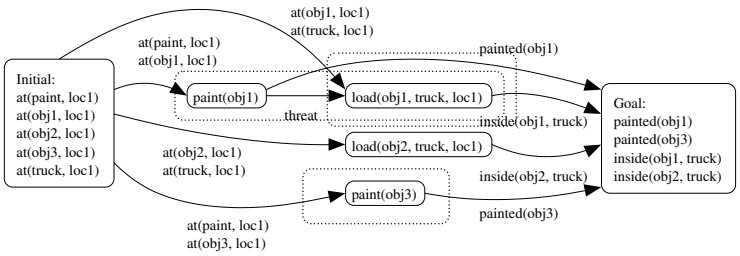

Figure 1: An example plan in a painting and transport domain is shown. In the given plan, some objects need to be painted and some need to be loaded into a truck. Painting must be done before loading. Three different subplans are surrounded by dotted lines. There are many other possible subplans, but the steps paint(obj1) and paint(obj3) are not a subplan, since they are not a connected component within the partial ordering.

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.

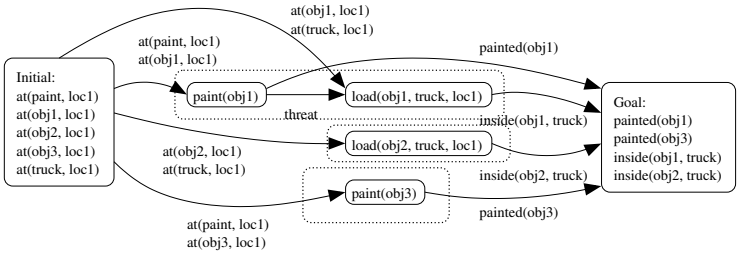

Figure 2: Three parallel subplans are surrounded by dotted lines.

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.

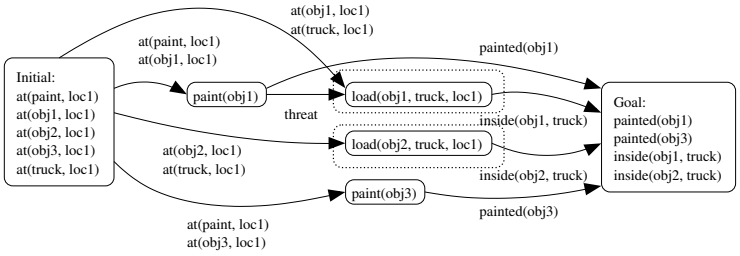

Figure 3: Two matching subplans of length 1 are surrounded by dotted lines and represent an unrolled loop.

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.

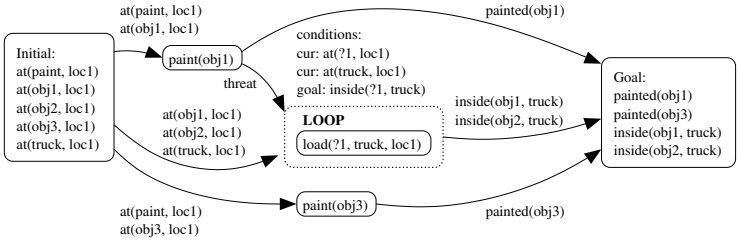

Figure 4: The painting and transport problem after the load loop is identified. The loop is surrounded by dotted lines. The loop variable is written as ?1, and ranges over all values that meet the conditions of the loop (in this case, obj1 and obj2). Conditions for the loop are shown above the loop.

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 include other loops.[1] 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 moving from the current location to a new one. These iterations must be executed in a specific order since the move operations are causally linked.

3.2 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 [10] and has

[1] Note that an observed total-order execution of a multi-step parallel loop need not present the steps of the loop in a specific order—it could be any topological sort of the loop.

shown that STRIPS-style operator models are learnable through examples and experimentation [2].

The ITERANT algorithm, formalized in Algorithm 1, 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 1 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 P do
  M_i ← all parallel matching steps with i in P
  if M_i ≠ ∅ then
    C ← LargestCommonSubplan(M_i + i, P)
    L ← MakeLoop(C)
    P ← P − C
    P ← P + L
  end if
end for
```

3.3 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 1. 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 2. 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.

Algorithm 2 LargestCommonSubplan: Identify largest parallel matching subplans of an observed plan common to at least two of the given parallel matching subplans.

Input: set $\mathcal{A}$ of parallel matching subplans $S_1..S_m$, minimal annotated partially ordered plan $\mathcal{P}$.

Output: Set of largest parallel matching subplans of plan $\mathcal{P}$ common to at least two of $S_1..S_m$.

for all S_i in $S_1..S_m$ **do**
  $StepAhead_{S_i} \leftarrow$ steps causally linked from S_i
  $StepBack_{S_i} \leftarrow$ steps causally linked to S_i
end for
$UnrolledLoops \leftarrow \mathcal{A}$
for $i = 1$ to m **do**
  for $NewSteps \leftarrow$ first $StepAhead_{S_i}$, then $StepBack_{S_i}$ **do**
    for all s in $NewSteps_{S_i}$ **do**
      $NewExpLoop \leftarrow \{S_i + s\}$
      for all $j \neq i$ **do**
        if $\exists$ parallel matching step s' in $NewSteps_{S_j}$ **then**
          $NewExpLoop \leftarrow NewExpLoop + \{S_j + s'\}$
          $NewSteps_{S_j} \leftarrow NewSteps_{S_j} - s'$
        end if
      end for
      if $|NewExpLoop| > 1$ **then**
        $UnrolledLoops \leftarrow UnrolledLoops + NewExpLoop$
        $NewSteps_{S_i} \leftarrow NewSteps_{S_i} - s$
      end if
    end for
  end for
end for
for all sets $\mathcal{N} \neq \mathcal{A}$ in $UnrolledLoops$ **do**
  $\mathcal{N} \leftarrow$ LargestCommonSubplan($\mathcal{N}, \mathcal{P}$)
end for
return set $\mathcal{N}$ in $UnrolledLoops$ with the largest subplan

3.4 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.

Algorithm 3 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 $\mathcal{P}$.
Output: The loop described by $S_1..S_m$.
  let v_i be the variable in S_i that $\forall j$ is not in S_j
  let v_{loop} be the loop variable
  $Loop.body \leftarrow S_1$ with v_{loop} replacing v_1
  $Loop.conditions \leftarrow \emptyset$
  for all steps s in $Loop.body$ **do**
    for all conditions c of s not satisfied by steps in $Loop.body$ **do**
      $Loop.conditions \leftarrow Loop.conditions + CurrentStateContains(c)$
    end for
    for all goal terms g dependent on s **do**
      $Loop.conditions \leftarrow Loop.conditions + GoalStateContains(c)$
    end for
  end for

3.5 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, s), load(o2, r, s), and load(o3, r, s), which differ only in one variable, which ranges over the values o1, o2, and o3.[2] These three one-step parallel matching subplans are then sent to LargestCommonSubplan, which searches for a larger subplan common to at least two 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.

[2] It could also have identified the unload loop first.

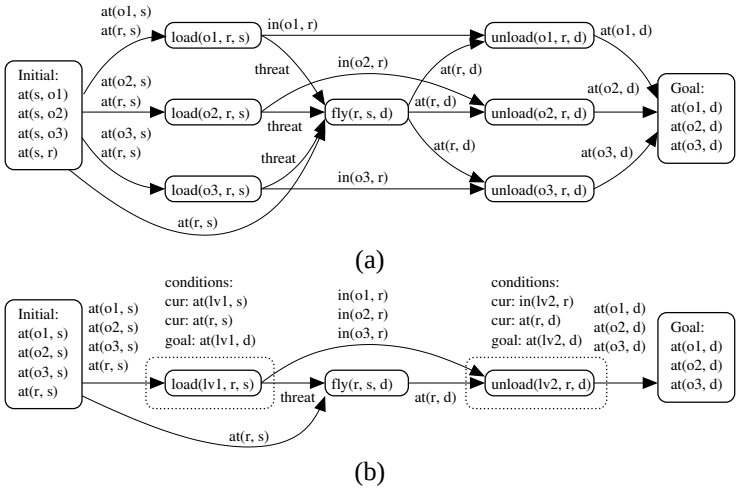

Figure 5: An example plan in the rocket domain that involves moving objects o1, o2, and o3 from location s to location d using rocket r. The minimal annotated partial ordering of the plan is shown in (a). The plan after loops are identified is shown in (b). Loops are surrounded by dotted lines. The loop 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.

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).

3.6 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: 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

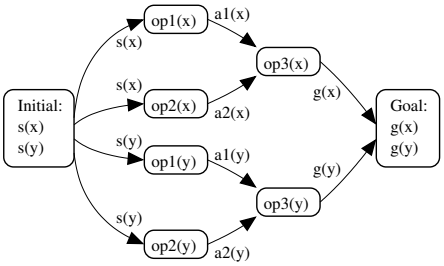

Figure 6: An example annotated partially ordered plan in an artificial domain that includes a multi-step loop consisting of the steps op1, op2, and op3. The original totally ordered plan could have been any topological sort of this partial ordering.

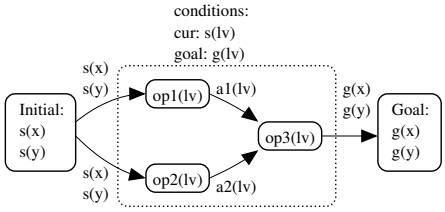

Figure 7: The example plan shown in Figure 6 after the loop has been identified. The loop is surrounded by dotted lines. The loop variable is written as lv, and ranges over all values that meet the conditions of the loop (in this case, x and y). The conditions of the loop are shown above it.

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.

4 Using Looping Plans as Domain-Specific Template 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.[3] The planner is a total ordering of the partially ordered plan. Loops are described as while statements: while the conditions for the loop hold, execute the body of the loop. We describe how to identify loops and their conditions above 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, ex-

[3]Two discrete objects in a plan are never allowed to map onto the same variable. As discussed in [3], this can lead to invalid plans.

ecute 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 then have our own executor of these dsPlanners.

5 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.

5.1 Rocket-Domain Results

The dsPlanner learned from the rocket domain example shown in Figure 5 is shown in dsPlanner 1. 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 5.1 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.

DsPlanner 1 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
   load(?v1:object ?3:rocket ?2:location)
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
   unload(?v1:object ?2:rocket ?3:location)
end while
```

[4] We 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.

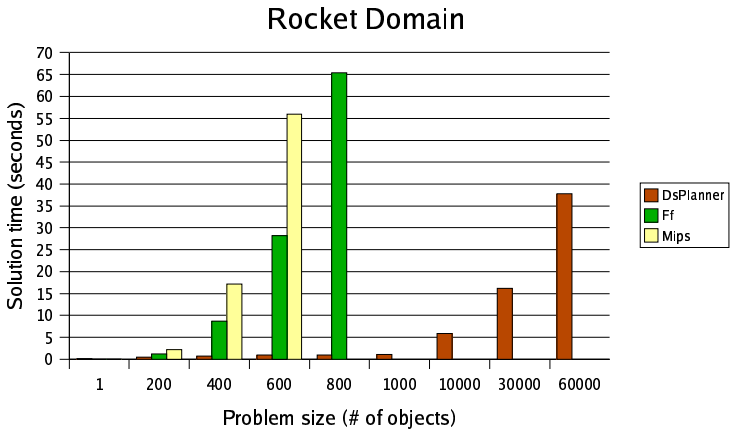

Figure 8: Timing results of several general-purpose planners and of the learned dsPlanner shown in DsPlanner 1 on large-scale rocket-domain delivery problems. All timing results were obtained on an 800-MHz pentium II with 512 MB of RAM.

5.2 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 2. 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 5.2 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.

DsPlanner 2 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
```

6 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 iden-

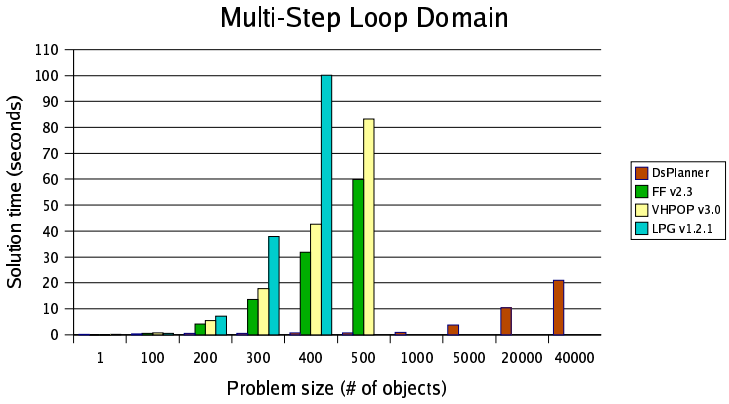

Figure 9: Timing results of several general-purpose planners and of the learned dsPlanner shown in dsPlanner 2 on large-scale multi-step loop domain problems.

tifying 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.

References

[1] Fahiem Bacchus and Michael Ady. Planning with resources and concurrency: A forward chaining approach. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-2001)*, pages 417–424, Seattle, August 2001.

[2] Jaime G. Carbonell and Yolanda Gil. Learning by experimentation: The operator refinement method. In R. S. Michalski and Y. Kodratoff, editors, *Machine Learning: An Artificial Intelligence Approach, Volume III*, pages 191–213. Morgan Kaufmann, Palo Alto, CA, 1990.

[3] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[4] Angela Hickman and Marsha Lovett. Partial match and search control via internal analogy. In *Proceedings of the 13th Annual Conference of the Cognitive Science Society*, pages 744–749, Chicago, 1991.

[5] Roni Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1-2):125–148, 1999.

[6] Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, December 2003.

[7] Ute Schmid. *Inductive Synthesis of Functional Programs*. Number 2654 in Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, 2003.

[8] Jude W. Shavlik. Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, 5:39–50, 1990.

[9] Manuela M. Veloso. *Planning and Learning by Analogical Reasoning*. Springer Verlag, December 1994.

[10] Elly Winner and Manuela Veloso. Analyzing plans with conditional effects. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)*, pages 271 – 280, Toulouse, France, April 2002.

[11] Elly Winner and Manuela Veloso. DISTILL: Learning domain-specific planners by example. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*, Washington, D.C., August 2003.