LoopDISTILL: 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

Many large-scale planning problems exhibit looping structure that translates into costly repeated problem solving effort, leading to the failure of many general-purpose planners to scale up. Instead, domain-specific planners can be more effective by reasoning about specific domain characteristics, such as repeated structure. While hand-writing such domain specific planners can be challenging, giving examples of plans for concrete problems with repeated structure is quite simple. In this work, we present the LoopDISTILL algorithm for *automatically* acquiring *looping domain-specific planners* from example plans. LoopDISTILL identifies repeated structures in the example plans and then converts the looping plans into domain-specific planners, or dsPlanners. Looping dsPlanners are able to apply experience acquired from the solutions to small example problems to solve arbitrarily large ones. We show that automatically learned dsPlanners are able to solve large-scale problems more effectively and to solve problems many orders of magnitude larger than general-purpose planners can solve.

Introduction

Many large-scale planning problems have a repetitive structure. Example plans are available for many domains, and can demonstrate this structure. Previous work introduced the concept of automatically-generated domain-specific planning programs (or dsPlanners) and showed how to use example plans to learn non-looping dsPlanners, which can solve problems of limited size (Winner & Veloso 2003). Here, we present the novel LoopDISTILL algorithm for *automatically* identifying the repeated structure of example plans to learn looping dsPlanners. DsPlanners execute independently of a general-purpose planning program and 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 effectively than can general-purpose planners and 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.

Finding optimal solutions to general planning problems 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, even if not necessarily optimally, to be solved. We believe that post-processing of plans can help improve plan quality, if needed.

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, describe our algorithm for automatically identifying loops in observed plans, and illustrate its behavior with examples. Next we present the results of using learned looping dsPlanners and compare this to using state-of-the-art general-purpose planners. Finally, we draw conclusions.

Related Work

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. (Minton 1988; Kambhampati & Hendler 1992). We focus here on methods that learn and exploit repeated structure within plans.

Case-based and analogical reasoning, e.g., (Veloso 1994), apply planning experience from previous problems to solving a new one. Similarly, the internal analogy technique (Hickman & Lovett 1991) 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., (Schmid 2003), capture repetitive behavior and can drastically reduce planning search by encapsulating an arbitrarily long sequence of operators. However, unlike our approach, this technique does not attempt to replace the generative planner, and so does not eliminate planning search.

Context free grammars have also been used as a parsing technique to extract instantiated patterns in examples that exhibit structural dependencies (Oates, Desai, & Bhat 2002), but do not necessarily capture looping terminating conditions and show how to solve new problems.

Some early work, which inspired our own work, also focused 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 (Shavlik 1990). BAGGER2 was

able to learn recurrences from few examples, but relied on background knowledge and did not capture parallel repetition.

Another example of the strategy-learning approach is the decision list (Khardon 1999): 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.

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 more problems than 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 the form of the dsPlanners 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), 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**);
- **inGoalState** and **inCurState** operators;
- numbered and typed variables;
- the "v" variant indicator for **while** loops;
- plan predicates; and
- plan operators.

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. Assignments hold throughout the conditions and body of the if statement or while-loop. While-loop variable assignments hold for all iterations of the loop unless the variable is labelled "v" for variant, in which case it may be reassigned at each iteration.

DsPlanner 1 solves all gripper-domain 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.

Planning with DsPlanners

To use the dsPlanner to solve a planning problem, first initialize the current state to the initial state and the solution plan to the empty plan. Then apply each of the statements to the current state. Each statement in the dsPlanner is either a 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.

A failure is detected when suggested plan steps are not applicable in the current state or when the dsPlanner finishes executing and its final state does not match the goal state. We handle failures by handing the problem off to a generative planner and then adding that new solution to the dsPlanner.

DsPlanner 1 A simple dsPlanner that solves all gripper-domain problems involving moving balls from one room to another.

```
while inCurState (at(v?1:ball v?2:room)) and inGoal-
State (at(v?1:ball v?3:room)) do
  if inCurState (at-robby(?5:room)) then
    move(?5 ?2)
  end if
  if inCurState (at-robby(?3:room)) then
    move(?3 ?2)
  end if
  pick(?1 ?2)
  move(?2 ?3)
  drop(?1 ?3)
end while
```

Identifying Loops in Example Plans

The current version of the LoopDISTILL algorithm identifies all non-nested loops with identical iterations in an observed plan. In the remainder of this section, we discuss some relevant definitions, describe in detail the two main portions of the LoopDISTILL algorithm—identifying loop candidates and creating a loop from a candidate)—and illustrate the operation of LoopDISTILL 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). Two subplans of a painting and transport domain problem are illustrated in Figure 1. 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 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 are the two paint operators.

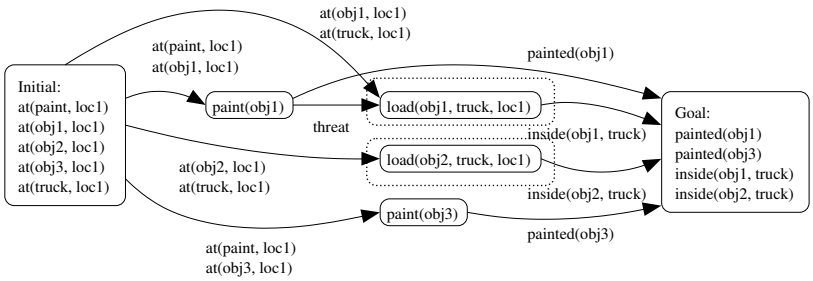

Figure 1: Two parallel matching subplans of length 1 are surrounded by dotted lines and represent an unrolled loop.

Parallel Subplans are causally- and threat-independent of each other. Figure 1 shows two parallel subplans.

Serial Subplans are causally linked to each other and are connected in the partial ordering (there are no plan steps which rely on one subplan and precede the next).

An Unrolled Loop is a set of matching subplans. One of two unrolled loops is circled in Figure 1.

A Loop replaces an unrolled loop in the plan. The body of the loop consists of the common subplan, but with the differing variables converted into loop variables. 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.

A Parallel Loop is a loop in which each iteration of the loop is causally independent from the others. The unrolled loop shown in Figure 1 is a parallel loop. A loop may also have a multi-step body with complex causal structure[1] The current version of LoopDISTILL is able to identify 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. The current version of LoopDISTILL identifies non-nested serial loops in observed plans.

The LoopDISTILL Algorithm

The LoopDISTILL 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), so this assumption is not restrictive.

The LoopDISTILL algorithm has two components, formalized in Algorithm 1 and Algorithm 2. The first extracts parallel loops from the observed plan and the second extracts serial loops. Both begin by identifying an unrolled loop (described in the Section "Identifying Unrollec Loops") and then converting 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 LoopDISTILL algorithm for identifying non-nested parallel loops in an observed plan.

Input: Minimal annotated partially ordered plan $\mathcal{P}$.
Output: $\mathcal{P}$ with all non-nested 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
```

Algorithm 2 LoopDISTILL algorithm for identify non-nested serial loops in an observed plan.

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

```
for all steps i in P do
  for all steps j in P causally linked from i do
    C ← ConnectSerialLoop(i, j, P)
    if C ≠ then
      C ← FindOtherSerialIterations(C, P)
      L ← MakeLoop(C)
      P ← P − C
      P ← P + L
      break / goto next i...?
    end if
  end for
end for
```

[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.

Identifying Unrolled Loops

Identifying Parallel Unrolled Loops The process of identifying a parallel unrolled loop—or a set of parallel matching subplans within the observed plan—begins with the identification of a set of parallel matching steps, as described in Algorithm 1. Next, LoopDISTILL finds the largest parallel matching subplan common to at least two of those steps. This process takes place in the procedure LargestCommonSubplan. 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.

Identifying Serial Unrolled Loops The process of finding a serial unrolled loop begins by stepping through the plan and, for each step, searching for a causally linked matching step. If such a step is found, LoopDISTILL tries to expand the two matching steps into serial matching subplans in the procedure ConnectSerialLoop. If the steps can be connected, then LoopDISTILL searches for additional iterations of the loop they represent in the procedure FindOtherSerialIterations. If they cannot be connected, LoopDISTILL continues searching for causally linked steps that match the original step.

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. The unrolled loop subplans are then removed from the plan and replaced by the new loop.

A Multi-Step Loop Example

We will now illustrate the operation of the LoopDISTILL algorithm on a simple example plan from an artificial domain, illustrated in Figure 2. First, LoopDISTILL 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.

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 $v_{i,j}$ be the jth variable in S_i that $\forall k$ is not in S_k
  let $v_{loop,j}$ be the jth loop variable
  $Loop.body \leftarrow S_1$ with $v_{loop,j}$ replacing $v_{1,j}$ $\forall j$
  $Loop.conditions \leftarrow \emptyset$
  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

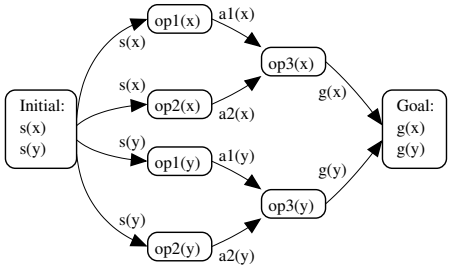

Figure 2: An example annotated partially ordered plan in an artificial domain that includes a multi-step loop consisting of the steps op1, op2, and op3. The original totally ordered plan could have been any topological sort of this partial ordering.

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 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 3.

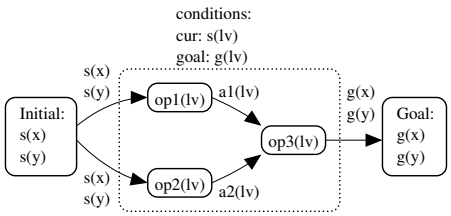

Figure 3: The example plan shown in Figure 2 after the loop has been identified. The loop is surrounded by dotted lines. The loop variable is written as lv, and ranges over all values that meet the conditions of the loop (in this case, x and y). The conditions of the loop are shown above it.

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 parametrized: values are replaced by variables.[2] 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. Plan steps not contained within loops are expressed as 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 goal-state terms that are dependent on those steps.

Results

We compare general-purpose planning, using several well-known general-purpose planners, to planning using learned looping dsPlanners. 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 a one-way rocket-domain example (**?**) is shown in dsPlanner 2. The problems on which we tested the planners vary in the number of objects to transport, but have a single rocket and two locations and 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 4 shows the results of executing several different general-purpose planners and the *learned* dsPlanner on large-scale problems of this form. Run times for the dsPlanner do not include the time required to learn the dsPlanner, though this is negligible [3]. The learned dsPlanner is orders of magnitude more efficient on large problems than the general-purpose planners, and is able to solve problems with more than 60,000 objects in under a minute.

DsPlanner 2 dsPlanner based on a one-way rocket domain problem. The variable in each loop is indicated by a "v" preceding its name.

```
while inCurState (at(v?1:obj, ?2:loc)) and inCurState
(at(?3:rocket, ?2:loc)) and inGoalState (at(v?1:obj,
?4:loc)) do
  load(?1 ?3 ?2)
end while
if inCurState (at(?1:rocket ?2:loc)) and inCurState
(in(?3:obj ?1:rocket)) and inGoalState (at(?3:obj
?4:loc)) then
  fly(?1 ?2 ?4)
end if
while inCurState (in(v?1:obj, ?2:rocket)) and in-
CurState (at(?2:rocket ?3:loc)) and inGoalState
(at(v?1:obj, ?3:loc)) do
  unload(?1 ?2 ?3)
end while
```

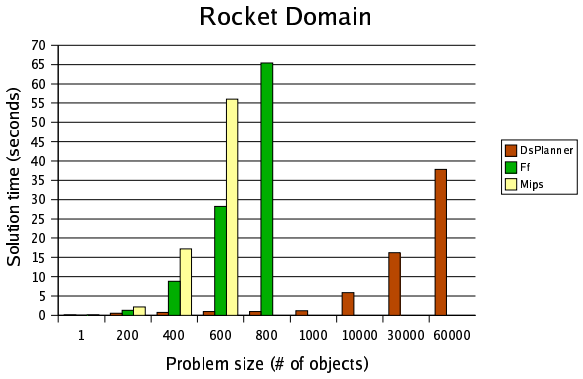

Figure 4: Timing results of several general-purpose planners and of the learned dsPlanner shown in DsPlanner 2 on large-scale rocket-domain delivery problems. All timing results were obtained on an 800-MHz Pentium II with 512 MB of RAM.

[2] Two discrete objects in a plan are never allowed to map onto the same variable as this can lead to invalid plans.

[3] It takes less than a second to learn the dsPlanner for the rocket-domain example we used

Multi-Step Loop Domain Results

The dsPlanner learned from the multi-Step loop domain example shown in Figures 2 and 3 is shown in DsPlanner 3. 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 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 3 DsPlanner based on the multi-step loop domain problem shown in Figures 2 and 3.

```
while inCurState (s(?v1:type1) and inGoalState
(g(?v1:type1))) do
   op1(?1)
   op2(?1)
   op3(?1)
end while
```

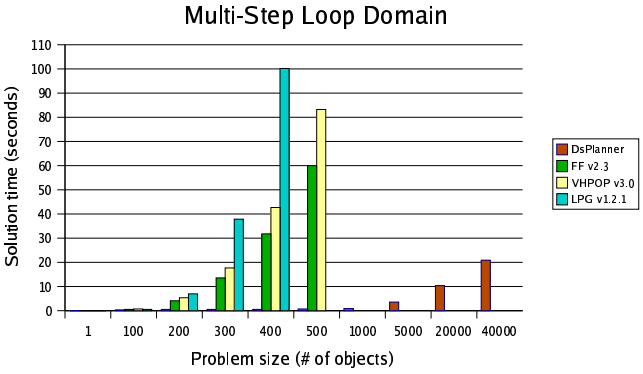

Figure 5: Timing results of several general-purpose planners and of the learned dsPlanner shown in dsPlanner 3 on large-scale multi-step loop domain problems.

Conclusion

In this paper, we contribute the LoopDISTILL algorithm for automatically identifying repeated structures in observed plans, determining the body and conditions of the loop they represent, and converting looping plans into looping domain-specific planning programs (dsPlanners). The LoopDISTILL algorithm identifies parallel 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 LoopDISTILL algorithm are able to take advantage of the repeated structures in planning problems. In these situations, planning using dsPlanners scales more effectively than general-purpose planning and extends the solvability horizon by solving problems orders of magnitude larger than general-purpose planners can handle. While these advantages could be expected of domain-specific planners, our core contribution is demonstrating that they can be achieved by automatically learned domain-specific planners.

References

Bacchus, F., and Ady, M. 2001. Planning with resources and concurrency: A forward chaining approach. In *Proceedings of IJCAI-2001*, 417–424.

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

Hammond, K. J. 1996. Chef: A model of case-based planning. In *Proceedings of AAAI-96*, 261–271.

Hickman, A., and Lovett, M. 1991. Partial match and search control via internal analogy. In *Proceedings of the CogSci1991*, 744–749.

Kambhampati, S., and Hendler, J. A. 1992. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence* 55(2-3):193–258.

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

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *JAIR* 20:1–59.

Minton, S. 1988. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Boston, MA: Kluwer Academic Publishers.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *JAIR* 20:379–404.

Oates, T.; Desai, D.; and Bhat, V. 2002. Learning k-reversible context-free grammars from positive structural examples. In *Proceedings of ICML-2002*.

Schmid, U. 2003. *Inductive Synthesis of Functional Programs*. Number 2654 in LNAI. Springer-Verlag.

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

Shell, P., and Carbonell, J. 1989. Towards a general framework for composing disjunctive and iterative macro-operators. In *Proceedings of IJCAI-89*.

Veloso, M. M. 1994. *Planning and Learning by Analogical Reasoning*. Springer Verlag.

Winner, E., and Veloso, M. 2002. Analyzing plans with conditional effects. In *Proceedings of AIPS-02*, 271 – 280.

Winner, E., and Veloso, M. 2003. DISTILL: Learning domain-specific planners by example. In *Proceedings of ICML-03*.