Automatically Acquiring Planning Templates from Example Plans

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

Abstract

General-purpose planning can solve problems in a variety of domains but can be quite inefficient. Domain-specific planners are more efficient but are difficult to create. In this paper, we introduce template-based planning, a novel paradigm for automatically generating domain-specific programs, or *templates*. We present the DISTILL algorithm for learning templates automatically from example plans and explain how templates are used to solve planning problems. DISTILL converts a plan into a template and then merges it with previously learned templates. Our results show that the templates automatically learned by DISTILL compactly represent its domain-specific planning experience. Furthermore, the templates situationally generalize the given example plans, thus allowing them to efficiently solve problems that have not previously been encountered.

Introduction

Planning is a powerful tool for action selection, since it offers a guarantee that a proposed plan achieves an agent's goals. If efficient, agents could re-plan to deal with unexpected situations. However, general-purpose planning is too slow to use in most real-time systems and does not scale to large problems. In order for planning to be feasible in these situations, some knowledge about the domain being solved must be used in the planning process, either by using a domain-specific planner or by using domain-specific knowledge to narrow the search.

Many researchers have focused on learning domain-specific control knowledge for planning automatically, usually in the forms of control rules, macro operators, and plan case libraries. There have also been several efforts focusing on writing domain-specific planners to quickly solve planning problems in particular domains without resorting to generative planning. These programs are currently hand-written, but this process is tedious and often quite difficult.

In this paper, we introduce the DISTILL algorithm, which automatically extracts these domain-specific planning programs (which we call *templates*) from example plans, and show how to use them to solve planning problems. We call these domain-specific planning programs *templates*. Table 1 shows a simple example template that solves all problems in the gripper domain that involve moving balls from one room to another.

```
while (in_goal_state (at(?1:ball ?2:room)) and
       in_current_state (at(?1:ball ?3:room)) and
       not same (?2:room ?3:room) and
       in_current_state (at-robby(?5:room))) do
   if (not same (?3:room ?5:room)) then
      move(?4 ?5 ?3)
   pick(?1 ?4 ?3)
   move(?4 ?3 ?2)
   drop(?1 ?4 ?2)
```

Table 1: A simple template that solves all gripper-domain problems involving moving balls from one room to another.

In some domains, finding optimal solutions is NP-complete. Therefore, templates 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.

Our work on the DISTILL algorithm for learning templates focuses on converting new example plans into templates in if-statement form and merging them, where possible. Our results show that merging templates produces a dramatic reduction in space usage compared to case-based or analogical plan libraries. We also show that by constructing and combining the if statements appropriately, we can achieve automatic *situational generalization*, which allows templates to solve problems that have not been encountered before without resorting to generative planning or requiring adaptation.

We first formalize the concept of templates. Next, we present our novel DISTILL algorithm for learning templates from example plans and present our results. We then discuss how to use templates to solve planning problems. Finally, we explore future work and present our conclusions.

Related Work

Control rules (Minton 1988a; Katukam & Kambhampati 1994; Etzioni 1993) act as a search heuristic during the planning process by "recommending" at certain points which branch of the planning tree the planner should explore first. They do not reduce the complexity of the planning task,

since they cannot *eliminate* branches of the search tree. They also capture only very local information (preference choices at specific branches of the planning search tree), ignoring common sequences of actions or repeated structures in example plans. It is difficult for people to write good control rules, in part because one must know the problem-solving architecture of the planner in order to provide useful advice about how it should make choices (Minton 1988b), and computer-learned control rules are often ineffective (Minton 1988b). Also, using control rules introduces a new problem for planners: when to create and save a new rule. Unrestricted learning creates a *utility* problem, in which learning more information can actually be counterproductive: it can take longer to search through a library of rules to find the ones that would help to solve a planning problem than to find the solution to the problem by planning from scratch (Minton 1988b).

Macro operators (Fikes, Hart, & Nilsson 1972; Korf 1985) combine frequently-occurring sequences of operations into combined operators. A macro can then be applied by the planner in one step, thus eliminating the search required to find the entire sequence again. Each new macro operator adds a new branch to the planning tree at every search node. Although they can decrease the search depth, the added breadth can make planning searches slower, so, as with control rules, it is difficult to determine when to add a new macro operator. Some research has studied the problem of how to learn only the most useful macros (Minton 1985), but the efficacy of macros has, in general, been limited to hierarchically decomposable domains.

Another approach to learning planning knowledge, *case-based reasoning*, attempts 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. Utility is also a problem for case-based planners; many handle libraries of tens of thousands of cases (Veloso 1994a), but, as with control rules, as the libraries get larger, the search times for relevant cases can exceed the time required to plan from scratch for a new case.

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 from the library and determining whether to save new cases.

Some work has addressed learning programs for planning, but this has been limited to learning *iterative* (Shell & Carbonell 1989) and *recursive* (Schmid 2001) macros: macro operators that contain repeated sequences of steps.

Defining Templates

A template 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 failure, if it cannot do so. Templates are composed of the following programming constructs and planning-specific operators:

- **while** loops;
- **if** , **then** , **else** statements;
- logical structures (**and** , **or** , **not**);
- **in_goal_state** , **in_current_state** , **in_initial_state** operators;
- **same** operator;
- plan predicates; and
- plan operators.

In order for templates 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 template, it must be possible to simulate the execution of the plan. However, since template learning requires full models of the planning operators, this is not an additional problem.

Table 1 shows a template that solves all gripper-domain (Long 2000) problems involving moving balls between rooms. The template 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 object, and drop the ball.

Learning Templates: the DISTILL Algorithm

The DISTILL algorithm, shown in Table 2, learns templates from sequences of example plans, incrementally adapting the template with each new plan. One benefit of online learning is that it allows a learner with access to a planner to acquire templates on the fly in the course of its regular activity. And because templates are learned from example plans, they reflect the *style* of those plans, thus making them suitable not only for planning, but also for agent modeling.

DISTILL can handle domains with conditional effects, but we assume that it has access to a complete model of the operators and to a minimal annotated partial ordering of the observed total order plan. Previous work has shown that operator models are learnable through examples and experimentation (Carbonell & Gil 1990; Wang 1994) and has shown how to find minimal annotated partial orderings of totally-ordered plans given a model of the operators (Winner & Veloso 2002).

The DISTILL algorithm converts observed plans into templates (see "Converting Plans into Templates") and merges them by finding templates with overlapping solutions and combining them (see "Merging Templates"). In essence, this builds a highly compressed case library. However, another key benefit comes from merging templates with overlapping

Input: Minimal annotated consistent partial order $\mathcal{P}$,
current template T_i.
Output: New template T_{i+1}, updated with $\mathcal{P}$

procedure DISTILL ($\mathcal{P}$, T_i):
  $\mathcal{A} \leftarrow$ Find_Variable_Assignment($\mathcal{P}$, $T_i.variables$, $\emptyset$)
  until match **or** can't match **do**
    if $\mathcal{A} = \emptyset$ **then**
      can't match
    else
      $\mathcal{N} \leftarrow$ Make_New_If_Statement(Assign($\mathcal{P}$, $\mathcal{A}$))
      match $\leftarrow$ Is_A_Match($\mathcal{N}$, T_i)
    if not can't match **and not** match **then**
      $\mathcal{A} \leftarrow$ Find_Variable_Assignment($\mathcal{P}$, $T_i.variables$, $\mathcal{A}$)
  if can't match **then**
    $\mathcal{A} \leftarrow$ Find_Variable_Assignment($\mathcal{P}$, $T_i.variables$, $\emptyset$)
    $\mathcal{N} \leftarrow$ Make_New_If_Statement(Assign($\mathcal{P}$, $\mathcal{A}$))
  $T_{i+1} \leftarrow$ Add_To_Template($\mathcal{N}$, T_i)

procedure Make_New_If_Statement($\mathcal{P}_\mathcal{A}$):
  $N \leftarrow$ empty if statement
  for all terms t_m in initial state of $\mathcal{P}_\mathcal{A}$ **do**
    if exists a step s_n in plan body of $\mathcal{P}_\mathcal{A}$ **such that**
        s_n needs t_m **or** goal state of $\mathcal{P}_\mathcal{A}$ needs t_m **then**
      Add_To_Conditions(N, **in_current_state** (t_m))
  for all terms t_m in goal state of $\mathcal{P}_\mathcal{A}$ **do**
    if exists a step s_n in plan body of $\mathcal{P}_\mathcal{A}$ **such that**
        t_m relies on s_n **then**
      Add_To_Conditions(N, **in_goal_state** (t_m))
  for all steps s_n in plan body of $\mathcal{P}_\mathcal{A}$ **do**
    Add_To_Body(N, s_n)
  return N

procedure Is_A_Match($\mathcal{N}$, T_i):
  for all if-statements I_n in T_i **do**
    if $\mathcal{N}$ matches of I_n **then**
      return true

procedure Add_To_Template($\mathcal{N}$, T_i):
  for all if-statements I_n in T_i **do**
    if $\mathcal{N}$ matches I_n **then**
      $I_n \leftarrow$ Combine(I_n, $\mathcal{N}$)
      return
  if $\mathcal{N}$ is unmatched **then**
    Add_To_End($\mathcal{N}$, T_i)

Table 2: The DISTILL algorithm: updating a template with a new observed plan.

solutions: this allows the template to find *situational generalizations* (Harris 1995) for individual sections of the plan, thus allowing it to reuse those sections when the same situation is encountered again, even in a completely different planning problem.

Generalizing Situations

We make several assumptions about what makes one planning *situation* different than another, and about how the observed planner will solve problems. We assume that two objects of the same type will be treated the same by the planner. Thus, two situations are equivalent if they contain the same number and types of objects in the same relationships. We assume that the planner will respond to equivalent situations with the same plan. This allows the DISTILL algorithm to identify common situations that occur in the solutions of several planning problems, and to extract their solutions for independent use in other problems.

Converting Plans into Templates

The first step of incorporating an example plan into the template is converting it into a parameterized if statement. First, the entire plan is parameterized. DISTILL chooses the first parameterization that allows part of the solution plan to match that of a previously-saved template. If no such parameterization exists, it randomly assigns variable names to the objects in the problem. [1]

Next, the parameterized plan is converted into a template, as formalized in the procedure Make_New_If_Statement in Table 2. The conditions on the new if statement are the initial- and goal-state terms that are *relevant* to the plan. Relevant initial-state terms are those which are needed for the plan to run correctly and achieve the goals (Veloso 1994a). Relevant goal-state terms are those which the plan accomplishes. We use a minimal annotated partial ordering (Winner & Veloso 2002) of the observed plan to compute which initial- and goal-state terms are relevant. The steps of the example plan compose the body of the new if statement. We store the minimal annotated partial ordering information for use in merging the template into the previously-acquired knowledge base.

Figure 1 shows an example minimal annotated partially ordered plan with conditional effects. Table 3 shows the template DISTILL creates to represent that plan. Note that the conditions on the generated if statement do not include all terms in the initial and goal states of the plan. For example, the template does not require that e(z) be in the initial and goal states of the example plan. This is because the plan steps do not generate e(z), nor do they need it to achieve the goals. Similarly, b(x) and the conditional effects that could generate the term c(x) or prevent its generation are also ignored, since it is not relevant to achieving the goals.

if (**in_current_state** (f(?0:type1)) **and**
  in_current_state (g(?1:type2)) **and**
  in_goal_state (a(?0:type1)) **and**
  in_goal_state (d(?1:type2))) **then**
  op1
  op2

Table 3: The template DISTILL would create to represent the plan shown in Figure 1.

Merging Templates

The merging process is formalized in the procedure Add_To_Template in Table 2. The templates learned by

[1] Two discrete objects in a plan are never allowed to map onto the same variable. As discussed in (Fikes, Hart, & Nilsson 1972), this can lead to invalid plans.

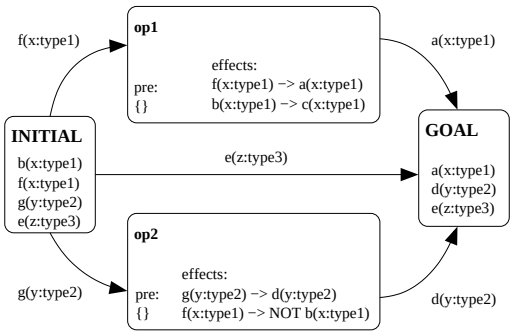

Figure 1: An example plan. The preconditions (pre) are listed, as are the effects, which are represented as conditional effects $a \rightarrow b$, i.e., if a then add b. A non-conditional effect that adds a literal b is then represented as $\{\} \rightarrow b$. Delete effects are represented as negated terms (e.g., $\{a\} \rightarrow NOTb$.

the DISTILL algorithm are sequences of non-nested if statements. To merge a new template into its knowledge base, DISTILL searches through each of the if statements already in the template to find one whose body (the solution plan for that problem) matches that of the new problem. We consider two plans to match if:

- one is a sub-plan of the other, or
- they overlap: the steps that end one begin the other.

If such a match is found, the two if statements are combined. If no match is found, the new if statement is simply added to the end of the template.

We will now describe how to combine two if statement templates, *if*$_1$= **if** x **then** abc and *if*$_2$= **if** y **then** b, when the body of *if*$_2$ is a sub-plan of that of *if*$_1$. This process is illustrated in Figure 2. [2]. For any set of conditions C and any step s applicable in the situation C, we define C_s to be the set of conditions that hold after step s is executed in the situation C. We also define a new function, $Relevant(C, s)$, which, for any set of conditions C and any plan step s, returns the conditions in C that are relevant to the step s.

As shown in Figure 2, merging *if*$_1$ and *if*$_2$ will result in three new if statements. We will label them *if*$_3$, *if*$_4$, and *if*$_5$. The body of *if*$_3$ is set to a and its conditions are $Relevant(x, a)$. The body of *if*$_4$ is b and its conditions are $Relevant(x_a, b)$ **or** $Relevant(y, b)$. [3] Finally, the body of *if*$_5$ is c and its conditions are $Relevant(x_a b, c)$. Whichever of *if*$_1$ or *if*$_2$ is already a member of the template is removed and replaced by the three new if statements.

Illustrative Results

Table 4 shows a template learned by the DISTILL algorithm that solves all problems in a blocks-world domain with two blocks. There are 555 such problems [4], but the template needs to store only two plan steps, and DISTILL is able to learn it from only 6 example plans.

Table 5 shows a template learned by the DISTILL algorithm to solve all gripper-domain problems with one ball, two rooms, and one robot with one gripper arm. Although there are 1932 such problems, [5] the DISTILL algorithm is able to learn the template from only five example plans. It successfully generalizes situations within individual plans for use in other plans. Also note that only five plan steps (the length of the longest plan) are stored in the template.

Our results show that templates achieve a significant reduction in space usage compared to case-based or analogical plan libraries. In addition, templates are also able to situationally generalize known problems to solve problems that have not been seen, but are composed of previously-seen situations.

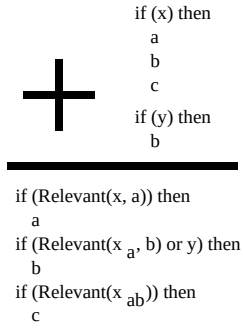

Figure 2: Combining two if statements when the body of one is a sub-plan of the body of the other.

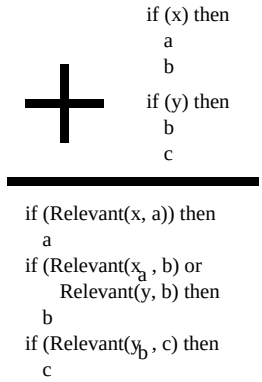

Figure 3: Combining two if statements when their bodies are overlapping.

[2] Combining two if statements with overlapping bodies is similar. It is illustrated in Figure 3

[3] Note that, though $Relevant(x, a) \subseteq x$, $Relevant(y, b) = y$.

[4] Though the initial state must be fully-specified in a problem, the goal state need only be partially specified. There are only three possible fully specified states in the blocksworld domain with two blocks, but there are 185 valid partially specified states.

[5] As previously mentioned, each problem consists of one fully-specified initial state (in this case, there are 6 valid fully-specified initial states), and one partially-specified goal state (in this case, there are 322).

if (**in_current_state** (clear(?1:block)) **and**
in_current_state (on(?1:block ?2:block)) **and**
(**in_goal_state** (on(?2:block ?1:block)) **or**
in_goal_state (on-table(?1:block)) **or**
in_goal_state (clear(?2:block)) **or**
in_goal_state (¬on(?1:block ?2:block)) **or**
in_goal_state (¬clear(?1:block)) **or**
in_goal_state (¬on-table(?2:block))
)) **then**
move-from-block-to-table(?1 ?2)
if (**in_current_state** (clear(?1:block)) **and**
in_current_state (clear(?2:block)) **and**
in_current_state (on-table(?2:block)) **and**
(**in_goal_state** (on(?2:block ?1:block)) **or**
in_goal_state (¬clear(?1:block)) **or**
in_goal_state (¬on-table(?2:block))
)) **then**
move-from-table-to-block(?2 ?1)

Table 4: A template learned by the DISTILL algorithm that solves all two-block blocks-world problems.

if (**in_current_state** (at(?3:ball ?2:room)) **and**
in_current_state (at-robby(?1:room)) **and**
(**in_goal_state** (at(?3:ball ?1:room)) **or**
in_goal_state (¬at(?3:ball ?2:room)) **or**
in_goal_state (holding(?3:ball))
)) **then**
Move(?1 ?2)
if (**in_current_state** (at(?3:ball ?2:room)) **and**
in_current_state (at-robby(?2:room)) **and**
(**in_goal_state** (at(?3:ball ?1:room)) **or**
in_goal_state (¬at(?3:ball ?2:room)) **or**
in_goal_state (holding(?3:ball))
)) **then**
Pick(?3 ?2)
if (**in_current_state** (holding(?3:ball)) **and**
in_current_state (at-robby(?2:room)) **and**
(**in_goal_state** (at(?3:ball ?1:room)) **or**
(**in_goal_state** (¬at(?3:ball ?2:room)) **and**
in_goal_state (¬holding(?3:ball)))
)) **then**
Move(?2 ?1)
if (**in_current_state** (holding(?3:ball)) **and**
in_current_state (at-robby(?1:room)) **and**
(**in_goal_state** (at(?3:ball ?1:room)) **or**
in_goal_state (¬holding(?3:ball))
)) **then**
Drop(?3 ?1)
if (**in_current_state** (at-robby(?1:room)) **and**
in_goal_state (at-robby(?2:room))
) **then**
Move(?1 ?2)

Table 5: A template learned by the DISTILL algorithm that solves all gripper-domain problems involving one ball two rooms, and one robot with one gripper.

Planning with Templates

Our algorithm for generating plans from templates is shown in Table 6. As previously mentioned, while executing the template, 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 template consists of applying each of the statements to the current state. Each statement in the template 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(?a:apple))) **then** eat(?a)", 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 the same, so, in this case, it doesn't matter which apple is eaten.

Detecting and Handling Failures

There are three ways a template may fail to generate the correct solution plan. It may have run through the whole template 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 template, 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 template 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 to add that new solution to the template.

Future Work

We are actively pursuing several research directions in template-based planning: enriching the process of converting an example plan into a template, refining the process of merging templates, and better handling failures by allowing the partial solutions generated by the template to be used to help guide the general-purpose planner's search.

The DISTILL algorithm's process of converting an example plan into a template can be extended to identify and extract loops in the example plan. To do this, it must be able to define the running conditions and stopping conditions for the loop and to determine when two loops can be merged. Merging two loops is more difficult than merging if statements; for example, two loops cannot be merged unless they

Input: Template T, initial state $\mathcal{I}$, current state $\mathcal{C}$ (initialized to $\mathcal{I}$), and goal state $\mathcal{G}$.
Output: Plan P that solves the given problem.

procedure Apply_Template(T, $\mathcal{I}$, $\mathcal{C}$, $\mathcal{G}$):
  $P \leftarrow \emptyset$
  for each statement S_n in T **do**
    $P \leftarrow P +$ Apply_Statement(S_n, $\mathcal{I}$, $\mathcal{C}$, $\mathcal{G}$)
  if $\mathcal{G}$ is satisfied by $\mathcal{C}$ **then**
    return P
  else
    FAIL

procedure Apply_Statement(S, $\mathcal{I}$, $\mathcal{C}$, $\mathcal{G}$):
  $P \leftarrow \emptyset$
  if S is an if statement **then**
    if Applies_Now(S, $\mathcal{C}$, $\mathcal{G}$) **then**
      for each statement S_i in the body of S **do**
        $P \leftarrow P +$ Apply_Statement(S_i, $\mathcal{C}$, $\mathcal{G}$)
  if S is a while statement **then**
    while Applies_Now(S, $\mathcal{I}$, $\mathcal{C}$, $\mathcal{G}$) **do**
      for each statement S_i in the body of S **do**
        $P \leftarrow P +$ Apply_Statement(S_i, $\mathcal{C}$, $\mathcal{G}$)
  if S is a plan step **then**
    if not Applicable(S, $\mathcal{C}$) **then**
      FAIL
    $\mathcal{C} \leftarrow$ Apply_Step(S, $\mathcal{C}$)
    $P \leftarrow S$
  return P

Table 6: Template-based plan generation.

have the same stopping conditions, even if they contain the same steps.

The process of merging templates can be refined by extending the matching capabilities of the DISTILL algorithm. Currently, DISTILL finds only the first match between a new template and previously constructed templates. We could also search for the longest match available. This would involve searching over different variable bindings for the new plan, since different bindings could result in different matches. Also, we currently allow a new template to match only one previously constructed template. We may find that merging a new template with as many other templates as possible results in better situational generalization.

Conclusions

In this paper, we contribute a formalism for automatically-generated domain-specific planning programs (templates) and the novel DISTILL algorithm, which automatically learns templates from example plans. The DISTILL algorithm first converts an observed plan into a template and then combines it with previously-generated templates. Our results show that templates learned by the DISTILL algorithm require much less space than do case libraries. Templates learned by DISTILL also support situational generalization, extracting commonly-solved situations and their solutions from stored templates so they can be reused in different problems.

Acknowledgements

This research is sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number No. F30602-00-2-0549. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of DARPA or AFRL.

References

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.

Etzioni, O. 1993. Acquiring search-control knowledge via static analysis. *Artificial Intelligence* 62(2):255–302.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence* 3(4):251–288.

Hammond, K. J. 1996. Chef: A model of case-based planning. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, 261–271. American Association for Artificial Intelligence.

Harris, J. R. 1995. Where is the child's environment? a group socialization theory of development. *Psychological Review* 102(3):458–489.

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

Katukam, S., and Kambhampati, S. 1994. Learning explanation-based search control rules for partial order planning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-94)*, volume 1, 582–587.

Korf, R. E. 1985. Macro-operators: A weak method for learning. *Artificial Intelligence* 26(1):35–78.

Leake, D. B., ed. 1996. *Case-Based Reasoning: experiences, lessons, and future directions*. AAAI Press/The MIT Press.

Long, D. 2000. The AIPS-98 planning competition. *AI Magazine* 21(2):13–34.

Minton, S. 1985. Selectively generalizing plans for problem-solving. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, 596–599. Los Angeles, CA: Morgan Kaufmann.

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

Minton, S. 1988b. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. Ph.D. Dissertation, Carnegie-Mellon University, Pittsburgh, PA.

Schmid, U. 2001. *Inductive Synthesis of Functional Programs*. Ph.D. Dissertation, Technische Universität Berlin, Berlin, Germany.

Shell, P., and Carbonell, J. 1989. Towards a general framework for composing disjunctive and iterative macro-operators. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*.

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

Veloso, M. M. 1994b. Prodigy/analogy: Analogical reasoning in general problem solving. In Wess, S.; Althoff, K.-D.; and Richter, M., eds., *Topics on Case-Based Reasoning*. Springer Verlag. 33–50.

Wang, X. 1994. Learning planning operators by observation and practice. In *Proceedings of the Second International Conference on AI Planning Systems, AIPS-94*, 335–340.

Winner, E., and Veloso, M. 2002. Analyzing plans with conditional effects. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)*. Forthcoming.