DISTILL: Learning Domain-Specific Planners by Example

Elly Winner ELLY@CS.CMU.EDU
Manuela Veloso MMV@CS.CMU.EDU
Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213

Abstract

An interesting alternative to domain-independent planning is to provide example plans to demonstrate how to solve problems in a particular domain and to use that information to learn domain-specific planners. Others have used example plans for case-based planning, but the retrieval and adaptation mechanisms for the inevitably large case libraries raise efficiency issues of concern. In this paper, we introduce dsPlanners, or automatically generated domain-specific planners. We present the DISTILL algorithm for learning dsPlanners automatically from example plans. DISTILL converts a plan into a dsPlanner and then merges it with previously learned dsPlanners. Our results show that the dsPlanners automatically learned by DISTILL compactly represent its domain-specific planning experience. Furthermore, the dsPlanners situationally generalize the given example plans, thus allowing them to efficiently solve problems that have not previously been encountered. Finally, we present the DISTILL procedure to automatically acquire one-step loops from example plans, which permits experience acquired from small problems to be applied to solving arbitrarily large ones.

1. Introduction

Intelligent agents must develop and execute autonomously a strategy for achieving their goals in a complex environment, and must adapt that strategy quickly to deal with unexpected changes. Solving complex problems with classical domain-independent planning techniques has required prohibitively high search efforts or tedious hand-coded domain knowledge, while universal planning and action-selection techniques have proven difficult to extend to complex environments.

Researchers have focused on making general-purpose planning more efficient by using either learned or hand-coded control knowledge to reduce search and thereby speed up the planning process. Machine learning approaches have relied on automatically extracting control information from domain and example plan analysis, with relative success in simple domains. Hand-coded control knowledge (or hand-written domain-specific planners) has proved more useful for more complex domains. However, it frequently requires great specific knowledge of the details of the underlying domain-independent planner for humans to formalize useful rules.

In this paper, we introduce the concept of dsPlanners, or automatically generated domain-specific planning programs. We then describe the DISTILL algorithm, which automatically extracts complete non-looping dsPlanners from example plans, and our method for identifying one-step loops in example plans.

The learning techniques used in the DISTILL algorithm allow problem solving that avoids the cost of generative planning and of maintaining exhaustive databases of observed behavior by compiling observed plans into compact domain-specific planners, or dsPlanners. These dsPlanners are able to duplicate the behavior shown in the example plans and to solve problems based on that behavior. Other planning methods have exponential time complexity, but dsPlanners return a solution plan or failure with complexity that is linear in the size of the planners, and the size of the solution, modulo state matching effort. The current DISTILL algorithm learns non-looping dsPlanners from example plans supplemented with their rationales. We show that these dsPlanners succeed in compactly capturing observed behavior and in solving many new problems. In fact, dsPlanners extracted from only a few example plans are able to solve all problems in limited domains.

Due to the complexity of finding optimal solutions in planning, 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

Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003.

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 dsPlanners focuses on converting new example plans into dsPlanners in if-statement form and merging them, where possible. Our results show that merging dsPlanners 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 dsPlanners to solve problems that have not been encountered before without resorting to generative planning or requiring adaptation.

We first discuss related work. Then, we formalize the concept of dsPlanners. Next, we present the novel DISTILL algorithm for learning complete non-looping dsPlanners from example plans and present our results. We then discuss our algorithm for automatically identifying one-step loops in example plans.

2. Related Work

We discuss related work in three main areas: in classical planning and learning, since our approach is another method of learning for planning; in automatic program generation, since our approach is to learn a planning program; and in universal planning, since the planning program our techniques learn is, effectively, a universal plan.

2.1. Domain Knowledge to Reduce Planning Search

A large body of work has focussed on acquiring and using domain knowledge to reduce planning search. Some of the most commonly used forms of domain knowledge are control rules, e.g., (Minton, 1988); macro operators, e.g., (Fikes et al., 1972); case-based and analogical reasoning, e.g., (Kambhampati & Hendler, 1992; Veloso, 1994); and hierarchical and skeletal planning, e.g., (Sacerdoti, 1974; Friedland & Iwasaki, 1985) [1]. Many of these methods suffer from the *utility* problem, in which learning more information can actually be counterproductive because of difficulty with storage and management of the information and with determining which information to use to solve a particular problem. In many cases, domain knowledge written by humans is also much more useful than that learned by computers. However, writing domain knowledge by hand is often very time-consuming and difficult, in part because writing effective domain knowledge often requires a deep understanding of the problem-solving architecture of the underlying planner. Though each of these techniques has been shown to reduce dramatically the planning time required to solve certain problems, they do not reduce the complexity of the planning problem and cannot, in general, solve problems without falling back on generative planning with a general-purpose planner. Our approach avoids generative planning search and the reliance on a general-purpose planner by acquiring a domain-specific planning program from observed example plans.

One technique used to identify control knowledge is explanation-based learning (EBL). This technique consists of explaining a success or failure in a search conducted by a specific planner. Given a state and a set of goals, EBL generates the explanation by filtering away the facts of the state or other goals that are not relevant to the success or failure of the search. Hence, EBL, applied to planning, produces control knowledge that guides the choices encountered in the search process of an existing planner. DsPlanners also rely on explaining example plans by looking at the weakest preconditions between state and goals. However, dsPlanners are not control knowledge for an existing planner; they are themselves planners: complete programs that create an output plan.

2.2. Automatic Program Generation

Work on automatic program generation can be divided into two main classes: *deductive* and *inductive program synthesis*. In deductive program synthesis, programs are generated from specifications, e.g., (Smith, 1991). We do not know of a general and concise way to describe the desired behavior of a domain-specific planner except through examples. Generating a program from examples is called inductive program synthesis. Our work falls into this category. Some work in inductive program synthesis induces programs from input/output pairs, e.g., (Muggleton, 1991). In planning, this corresponds to inducing a planning program by looking only at pairs of initial and goal states. Other work induces programs from example execution traces, e.g., (Bauer, 1979; Lau, 2001). In planning, this corresponds to inducing a planner from example problems and solution plans that solve the problems. This is the approach we have taken.

However, work in inductive program synthesis is not immediately applicable to our problem. In much inductive program synthesis work, example execution traces are annotated to mark the beginnings and ends of loops, to specify loop invariants and stopping conditions, to mark conditionals, etc. This kind of labelling cannot be obtained automatically from observed executions, so we do not allow it in our work. Another difference is that, whereas many approaches to IPS must attempt to induce the purpose of

[1] Because the research we discuss in the Related Work section is so broad, we list only representative examples of the work and limit ourselves to only a brief mention or description of different approaches.

the steps from many examples, in our planning-based approach, the purpose of each step is automatically deduced via plan analysis. This information is critical to rapidly and correctly identifying the conditions for executing a sequence of steps or for terminating a loop. Despite these differences, several researchers have explored the application of inductive program synthesis to planning.

Inductive program synthesis has been used to generate iterative and recursive macro operators, e.g., (Schmid, 2001). These macros 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 has also focussed on analyzing example plans to reveal a strategy for planning in a particular domain in the form of a decision list (Khardon, 1999), or a list of condition-action pairs. These condition-action pairs are derived from example state-action pairs. This technique is able to solve fewer than 50% of 20-block Blocksworld problems and requires over a thousand state-action pairs to achieve coverage (Khardon, 1999). In our work, we preserve and analyze the structure of the observed plans in order to extract as much information as possible from limited evidence. Our hope is that this will result in a more successful algorithm that requires orders of magnitude fewer examples.

2.3. Universal Planning

Some researchers have sought to avoid the planning search problem by acquiring and using "universal plans," or precomputed functions that map state and goal combinations to actions. Our work can be seen as a new method of acquiring and storing universal plans.

One previous approach to automatically acquiring a universal plan is reinforcement learning. There has, however, been limited success in applying the solution to one problem to another, particularly to a larger or more complex problem. There have also been many approaches to finding more compact ways of representing the learned universal plan, e.g., (Uther & Veloso, 2000), but such plans can still be prohibitively large for interesting problems in complex domains.

Decision trees have also been used in a purely planning context. Schoppers suggests decision trees splitting on state and goal predicates (Schoppers, 1987), but finds these trees by conducting a breadth-first search for solutions—a method which is too time-consuming for most domains.

Other researchers have used Ordered Binary Decision Diagrams (OBDDs) to represent universal plans (Cimatti et al., 1998). OBDDs provide an effective way to compress a universal plan without losing precision, however are currently generated via blind backwards search from goal states, a method that is impractical in complex domains.

3. Defining and Using dsPlanners

We now introduce the concept of a dsPlanner, 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. The dsPlanner is a novel method of storing planning knowledge. It is expressive and compact and does not rely on an underlying general-purpose planner.

DsPlanners 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** and **in_current_state** operators;
- **v** variant indicators;
- plan predicates; and
- plan operators.

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

The dsPlanner language is rich enough to allow compact planners for many difficult problems. We demonstrate this by presenting two short hand-written dsPlanners that solve all problems in well-known domains. Our current algorithm for learning dsPlanners from examples, DISTILL, does not find looping dsPlanners. However, we continue to work towards this goal, and in Section 5, we describe the DISTILL procedure for identifying simple loops in example plans.

Table 1 shows a simple but suboptimal hand-written dsPlanner that solves all BlocksWorld-domain (Veloso, 1994) problems that involve building towers of blocks. The dsPlanner is composed of three while loops: first, all blocks should be unstacked; then, the second-to-bottom block of every tower should be stacked onto the bottom block; then,

```
plan ← {}
while (in_current_state (on-block(?1:block ?2:block)) and
        in_current_state (clear(?1:block))) do
    operator ← move-block-to-table(?1:block ?2:block)
    execute operator
    plan ← plan + operator
while (in_goal_state (on-block(v?1:block v?2:block)) and
        not (in_goal_state (on-block(v?2:block v?3:block))) and
        in_current_state (clear(v?1:block)) and
        in_current_state (clear(v?2:block))) do
    operator ← move-table-to-block(?1:block ?2:block)
    execute operator
    plan ← plan + operator
while (in_goal_state (on-block(v?1:block v?2:block)) and
        in_current_state (on-table(v?1:block)) and
        in_current_state (on-block(v?2:block v?3:block))) do
    operator ← move-table-to-block(?1:block ?2:block)
    execute operator
    plan ← plan + operator
return plan
```

Table 1. A dsPlanner that solves all BlocksWorld-domain problems. In future example dsPlanners, for the sake of compactness, we replace the operator executing and plan updating and returning notation with the name of the operator to execute and add to the plan.

```
while (in_goal_state (at(v?1:pkg v?2:city)) and
        not (in_current_state (at(v?1:pkg v?2:city)))) do
    while (in_current_state (inside(v?3:pkg v?4:rocket)) and
            in_goal_state (at(v?3:pkg v?5:city))) do
        while (in_current_state (inside(v?6:pkg ?4:rocket)) and
                in_current_state (at(?4:rocket v?7:city)) and
                in_goal_state (at(v?6:pkg v?7:city))) do
            unload(?6:pkg ?4:rocket ?7:city)
        while (in_current_state (at(?4:rocket v?6:city)) and
                in_current_state (at(v?7:pkg v?6:city)) and
                in_goal_state (at(v?7:pkg v?8:city))) do
            load(?7:pkg ?4:rocket ?6:city)
        if (in_current_state (at(?4:rocket ?6:city))) then
            fly(?4:rocket ?6:city ?5:city)
    if (in_current_state (at(?1:pkg ?3:city)) and
            in_current_state (at(?4:rocket ?5:city))) then
        fly(?4:rocket ?5:city ?3:city)
    if (in_current_state (at(?1:pkg ?3:city)) and
            in_current_state (at(?4:rocket ?3:city))) then
        load(?1:pkg ?4:rocket ?3:city)
while (in_current_state (at(v?1:rocket v?2:city)) and
        in_goal_state (at(v?1:rocket v?3:city))) do
    fly(?1:rocket ?2:city ?3:city)
```

Table 2. A dsPlanner that solves all Rocket-domain problems.

for each block that is stacked on a second block in the goal state, if the second block is already stacked on a third, go ahead and stack the first block on the second.

Table 2 shows a hand-written dsPlanner that solves all Rocket-domain (Veloso, 1994) problems. The dsPlanner is composed of two while loops: while there is some package that is not at its goal location, execute the following loop: while there is some package in the rocket that should arrive at a goal destination, unload all packages in the rocket that should end up in the rocket's current city, load all packages in the rocket's current city that should go elsewhere, then fly the rocket to the goal destination of the package inside it that should be delivered to a goal destination. Once the rocket contains no more packages that should be delivered to goal destinations, fly the rocket to the location of the original misplaced package, load it into the rocket, and begin the rocket-emptying loop again. Once all the packages are correctly placed, fly each rocket to its goal location.

We now describe how to generate plans from dsPlanners. As previously mentioned, while executing the dsPlanner, we must keep track of a current state and of the current solution plan. The current state is initialized to the initial state, and the solution plan is initialized to the empty plan. Executing the dsPlanner is the same as executing a *program*: it consists of applying 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. Once execution of the dsPlanner is finished and all suggested plan steps have been determined to be applicable, the final state must be checked to ensure that it satisfies the goals. If it does, the generated plan is returned. Otherwise, the dsPlanner must return failure. As previously mentioned, dsPlanner execution is of linear time complexity in the size of the dsPlanner, the size of the problem, and the size of the solution.

4. Learning Non-Looping dsPlanners

The current version of the DISTILL algorithm, shown in Table 3, learns complete, non-repeating dsPlanners from sequences of example plans, incrementally adapting the dsPlanner with each new plan. In Section 5, we present the DISTILL procedure for identifying simple loops in example plans. We describe the two main portions of the current DISTILL algorithm (converting example plans into dsPlanners and merging dsPlanners) in detail in the rest of this section. We use online learning in DISTILL because it allows a learner with access to a planner to acquire dsPlanners on the fly as it encounters gaps in its knowledge in the

course of its regular activity. And because dsPlanners 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.

Input: Minimal annotated consistent partial order $\mathcal{P}$,
current dsPlanner dsP_i.
Output: New dsPlanner dsP_{i+1}, updated with $\mathcal{P}$

procedure DISTILL ($\mathcal{P}$, dsP_i):
  $\mathcal{A} \leftarrow$ Find_Variable_Assignment($\mathcal{P}$, $dsP_i.variables$, $\emptyset$)
  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($\mathcal{N}$, dsP_i)
    if not can't match **and not** match **then**
      $\mathcal{A} \leftarrow$ Find_Variable_Assignment($\mathcal{P}$, $dsP_i.variables$, $\mathcal{A}$)
  if can't match **then**
    $\mathcal{A} \leftarrow$ Find_Variable_Assignment($\mathcal{P}$, $dsP_i.variables$, $\emptyset$)
    $\mathcal{N} \leftarrow$ Make_New_If_Statement(Assign($\mathcal{P}$, $\mathcal{A}$))
  $dsP_{i+1} \leftarrow$ Add_To_dsPlanner($\mathcal{N}$, dsP_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($\mathcal{N}$, dsP_i):
  for all if-statements I_n in dsP_i **do**
    if $\mathcal{N}$ matches I_n **then**
      return true

procedure Add_To_dsPlanner($\mathcal{N}$, dsP_i):
  for all if-statements I_n in dsP_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($\mathcal{N}$, dsP_i)

Table 3. The DISTILL algorithm: learning a dsPlanner from example plans.

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

The DISTILL algorithm converts observed plans into dsPlanners, described in Section 4.1, and merges them by finding dsPlanners with overlapping solutions and combining them, described in Section 4.2. In essence, this builds a highly compressed case library. However, another key benefit comes from merging dsPlanners with overlapping solutions: this allows the dsPlanner to find *situational generalizations* 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.

4.1. Converting Plans into dsPlanners

The first step of incorporating an example plan into the dsPlanner 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 dsPlanner. If no such parameterization exists, it randomly assigns variable names to the objects in the problem. [2]

Next, the parameterized plan is converted into a dsPlanner, as formalized in the procedure Make_New_If_Statement in Table 3. The conditions of 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, 1994). Relevant goal-state terms are those which the plan accomplishes. We use a minimal annotated consistent 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 consistent partial ordering information for use in merging the dsPlanner into the previously-acquired knowledge base.

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 4. The dsPlanner DISTILL creates to represent the plan shown in Figure 1.

Figure 1 shows an example minimal annotated consistent partially ordered plan with conditional effects. Table 4 shows the dsPlanner DISTILL creates to represent that plan. Note that the conditions on the generated if statement do

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

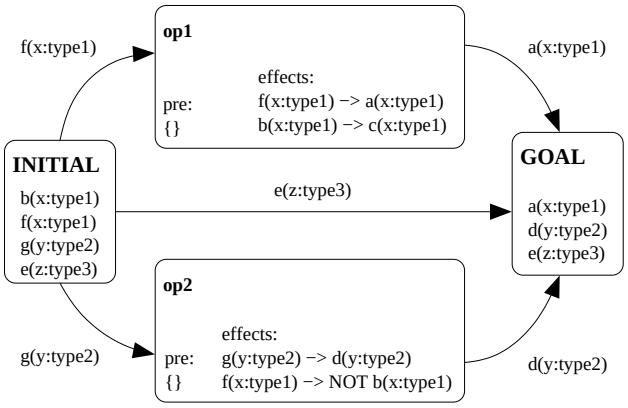

Figure 1. An example plan. The preconditions (pre) are listed, as are the effects, which are represented as conditional effects $a \rightarrow b$, i.e., if a then add b. A non-conditional effect that adds a literal b is then represented as $\{\} \rightarrow b$. Delete effects are represented as negated terms (e.g., $\{a\} \rightarrow NOT\ b$).

not include all terms in the initial and goal states of the plan. For example, the dsPlanner 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.

4.2. Merging dsPlanners

The merging process is formalized in the procedure Add_To_dsPlanner in Table 3. The dsPlanners learned by the DISTILL algorithm are sequences of non-nested if statements. To merge a new dsPlanner into its knowledge base, DISTILL searches through each of the if statements already in the dsPlanner 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 dsPlanner.

We will now describe how to combine two if statement dsPlanners, *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*$_1$. 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.

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 $Relevant(x_{ab}, c)$. Whichever of *if*$_1$ or *if*$_2$ is already a member of the dsPlanner is removed and replaced by the three new if statements.

Combining two if statements with overlapping bodies is similar. Merging the two if statements *if*$_1$= **if** x **then** ab and *if*$_2$= **if** y **then** bc will result in three new if statements, labelled *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** $Relevant(y, b)$. Finally, the body of *if*$_5$ is c and its conditions are $Relevant(y_b, c)$. Again, whichever of *if*$_1$ or *if*$_2$ is already a member of the dsPlanner is removed and replaced by the three new if statements.

4.3. Illustrative Results

We present results of applying DISTILL to limited domains since we have not yet added to DISTILL the ability to learn looping dsPlanners from observed plans. Our results show that, even without the ability to represent loops, the dsPlanners learned by DISTILL are able to capture complete domains from few examples and to store these complete solutions very compactly.

Table 5 shows the dsPlanner learned by the DISTILL algorithm that solves all problems in a blocks-world domain with two blocks. There are 704 such problems, [4] but the dsPlanner needs to store only two plan steps, and it is possible for DISTILL to learn the dsPlanner from only 6 example plans. These six example plans were chosen to cover the domain; more examples could be required for the complete dsPlanner to be learned if the examples were randomly selected.

Table 6 shows the dsPlanner learned by the DISTILL algorithm that solves all gripper-domain problems with one ball, two rooms, and one robot with one gripper arm. Although there are 1722 such problems, [5] it is possible for the DISTILL algorithm to learn the dsPlanner from only six ex-

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

[4] Though the initial state must be fully-specified in a problem, the goal state need only be partially specified. There are only four valid fully specified states in the blocksworld domain with two blocks, but there are 176 valid partially specified goal 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 287).

```
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 5. The dsPlanner learned by the DISTILL algorithm that solves all two-block blocks-world problems.

ample plans. Also note that only five plan steps (the length of the longest plan) are stored in the dsPlanner.

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

5. Identifying Loops

We now present the DISTILL procedure to identify one-step loops in which repeated behaviors are not causally linked. A looping plan in the rocket domain involves loading several packages into the rocket, flying the rocket to the goal destination, and then unloading all the packages. In this example, the sequence of package loads is a loop, since they are repeated identical behaviors. No package load in this loop is causally linked to the other loads. The unloading sequence is a similar loop. The algorithm we use to identify such loops is shown in Table 7, and Table 8 shows the dsPlanner code our algorithm extracts from the above example.

The DISTILL procedure for identifying loops in observed plans identifies steps that are not linked in the transitive closure of the partial ordering of the plan (and thus run in parallel). If the parallel plan steps are the same operator, differ in only one variable, and have the same needs and effects, they are considered part of a loop. The conditions for the loop's execution are the needs and effects of the steps it encompasses. The repeated steps are removed from the plan and replaced by the newly created loop. Many solutions to planning problems involve the repetition of steps, which, when translated into appropriate loops, will greatly

```
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)) or
            in_goal_state (¬free-arm)
        )) 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)) or
            in_goal_state (¬free-arm)
        )) 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)) or
                    in_goal_state (free-arm)))
        )) 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)) or
            in_goal_state (free-arm)
        )) then
    drop(?3 ?1)
if (in_current_state (at-robby(?1:room)) and
        (in_goal_state (at-robby(?2:room)) or
            in_goal_state (¬at-robby(?1:room)))
        ) then
    move(?1 ?2)
```

Table 6. The dsPlanner learned by the DISTILL algorithm that solves all gripper-domain problems involving one ball, two rooms, and one robot with one gripper arm.

increase the complexity of the problems the planner can solve.

6. Conclusions

In this paper, we contribute a formalism for automatically-generated domain-specific planning programs (dsPlanners) and present the DISTILL algorithm, which automatically learns non-looping dsPlanners from example plans. The DISTILL algorithm first converts an observed plan into a dsPlanner and then combines it with previously-generated dsPlanners. Our results show that dsPlanners learned by the DISTILL algorithm require much less space than do case libraries. dsPlanners learned by DISTILL also support situational generalization, extracting commonly-solved situations and their solutions from stored dsPlanners. This allows dsPlanners to reuse previous planning experience to solve different problems. We also discuss our work to-

Input: Transitive closure of the minimal annotated consistent partial order, $\mathcal{P}$
Output: New minimal annotated consistent partial order $\mathcal{L}$ that represents $\mathcal{P}$, but has identified loops.

procedure Find_Loops($\mathcal{P}$):
  $\mathcal{L} \leftarrow$ Find_A_Loop($\mathcal{P}$)
  while ($\mathcal{L}$ reflects changes in $\mathcal{P}$) **do**
    $\mathcal{P} \leftarrow \mathcal{L}$
    $\mathcal{L} \leftarrow$ Find_A_Loop($\mathcal{P}$)
  return $\mathcal{L}$

procedure Find_A_Loop($\mathcal{P}$):
  for each step s_i in $\mathcal{P}$ **do**
    $NewLoop.body \leftarrow s_i$
    $NewLoop.needs \leftarrow$ needs of s_i
    $NewLoop.effs \leftarrow$ effects of s_i
    while exists step s_j in $\mathcal{P}$ **such that**
        s_i and s_j are not linked in $\mathcal{P}$ **and**
        s_i and s_j are the same operator **and**
        s_i and s_j differ in a single variable **and**
        s_i and s_j have the same needs **and**
        s_i and s_j have the same effects **do**
      $NewLoop.body \leftarrow NewLoop.body$ + s_j
      $NewLoop.needs \leftarrow NewLoop.needs$ + needs of s_j
      $NewLoop.effs \leftarrow NewLoop.effs$ + effects of s_j
    if $NewLoop.body$ contains more than s_i **then**
      $\mathcal{P} \leftarrow \mathcal{P}$ - steps in $NewLoop.body$
      $\mathcal{P} \leftarrow \mathcal{P}$ + $NewLoop$

Table 7. The DISTILL procedure for identifying one-step loops.

while (**in_current_state** (at(?0:rocket ?1:city)) **and**
      in_current_state (at(v?2:pkg ?1:city)) **and**
      in_goal_state (at(v?2:pkg ?3:city))) **do**
  load(v?2:pkg ?0:rocket ?1:city)
if (**in_current_state** (at(?0:rocket ?1:city)) **and**
      in_current_state (inside(?2:pkg ?0:rocket)) **and**
      in_goal_state (at(?2:pkg ?3:city))) **then**
  fly(?0:rocket ?1:city ?3:city)
while (**in_current_state** (at(?0:rocket ?1:city)) **and**
      in_current_state (inside(v?2:pkg ?0:rocket)) **and**
      in_goal_state (at(v?2:pkg ?1:city))) **do**
  unload(v?2:pkg ?0:rocket ?1:city)

Table 8. The learned dsPlanner that represents a looping rocket-domain plan.

wards automatically acquiring looping dsPlanners. Looping dsPlanners will allow observed solutions to small problems to be used to solve arbitrarily large ones.

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

Bauer, M. (1979). Programming by examples. *Artificial Intelligence*, *12*, 1–21.

Carbonell, J. G., & Gil, Y. (1990). Learning by experimentation: The operator refinement method. In R. S. Michalski and Y. Kodratoff (Eds.), *Machine learning: An artificial intelligence approach, volume III*, 191–213. Palo Alto, CA: Morgan Kaufmann.

Cimatti, A., Roveri, M., & Traverso, P. (1998). Automatic OBDD-based generation of universal plans in non-deterministic domains. *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)* (pp. 875–881). AAAI Press.

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

Friedland, P. E., & Iwasaki, Y. (1985). The concept and implementation of skeletal plans. *Journal of Automated Reasoning*, *1*, 161–208.

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

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

Lau, T. (2001). *Programming by demonstration: a machine learning approach*. Doctoral dissertation, University of Washington, Seattle.

Minton, S. (1988). *Learning effective search control knowledge: An explanation-based approach*. Boston, MA: Kluwer Academic Publishers.

Muggleton, S. (1991). Inductive logic programming. *New Generation Computing*, *8*, 295–318.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, *5*, 115–135.

Schmid, U. (2001). *Inductive synthesis of functional programs*. Doctoral dissertation, Technische Universität Berlin, Berlin, Germany.

Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable environments. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-1987)* (pp. 1039–1046). Milan, Italy.

Smith, D. R. (1991). KIDS: A knowledge-based software development system. In M. R. Lowry and R. D. McCartney (Eds.), *Automating software design*. AAAI press.

Uther, W. T. B., & Veloso, M. (2000). The lumberjack algorithm for learning linked decision forests. *Symposium on Abstraction, Reformulation and Approximation (SARA-2000)* (pp. 219–232). Springer Verlag.

Veloso, M. M. (1994). *Planning and learning by analogical reasoning*. Springer Verlag.

Winner, E., & Veloso, M. (2002). Analyzing plans with conditional effects. *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)* (pp. 271 – 280). Toulouse, France.