Learning Domain-Specific Planners From Example Plans

Elly Zoe Winner

CMU-CS-08-112

May 2008

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Manuela Veloso, Chair
Avrim Blum
Reid Simmons
Leslie Kaelbling, M.I.T.

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported under a National Science Foundation Graduate Research Fellowship, by the United States Air Force Grants Nos. F30602-97-2-0250, F30602-98-2-0135, and F30602-00-2-0549, and by BBNT Solutions under subcontract no. 950008572, via prime Air Force contract no. SA-8650-06-C-7606. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

Keywords: Planning, Learning, Domain-specific planning, Program synthesis, Looping plan learning

For Jovan and Manuela, who made me do it.

Abstract

Automated problem solving involves the ability to select actions from a specific state to reach objectives. Classical planning research has addressed this problem in a domain-independent manner—the same algorithm generates a complete plan for any domain specification. While this generality is in principle desirable, it comes at a cost which domain-independent planners incur either in high search efforts or in tedious hand-coded domain knowledge.

Previous approaches to efficient general-purpose planning have focused on reducing the search involved in an existing general-purpose planning algorithm. Others abandoned the general-purpose goal and developed special-purpose planners highly optimized in efficiency for the specific aspects of a particular problem solving domain. An interesting alternative is to use example plans in a particular domain to demonstrate how to solve problems in that domain and to use that information to solve new problems independently of a domain-independent planner. Others have used example plans for case-based and analogical planning, but the retrieval and adaptation mechanisms were still domain-independent and efficiency issues were still a concern. Recently, example plans have been used to induce decision lists, but many examples and hours or even days of computation time were needed to learn the lists.

This thesis presents a novel way of using example plans: by analyzing individual example plans thoroughly, our algorithms reveal the rationale and structure underlying the plan, and use this information to rapidly learn complex, looping domain-specific planners (or dsPlanners) automatically. In this thesis, I introduce the dsPlanner language, a clear, human-readable and -writeable programming language for describing learnable domain-specific planners; the SPRAWL algorithm for analyzing observed plans and uncovering the rationale underlying the plan; the DISTILL algorithm for automatically learning non-looping dsPlanners from sets of example plans; and the LoopDISTILL algorithm for automatically learning looping dsPlanners from examples. I show that the careful analysis of example plans can make learning so efficient that a dsPlanner that covers large classes of arbitrarily large problems can be learned from a single example in under a second for a wide variety of domains. Automatically learned dsPlanners can then be used to solve new planning problems in linear time, modulo state matching effort.

Acknowledgments

I owe an enormous debt of gratitude to my husband Jovan, who joyfully took on making the time and space for the completion of this thesis as his own personal mission; and to my advisor, Manuela Veloso, who went far beyond the call of duty to find a way to help me finish. I could not have continued on with this work as it stretched over months and years without my "real life"—Jovan, Saan, and Rai, and our home overflowing with chaos, sunlight, songs and shrieks, all manner of art works and science experiments (sometimes on each other), and great food and junk food, and love, and laughter and yelling and banging and wailing. And love and thanks to my parents and to my whole extended family, who have never ceased to be my cheerleaders, sounding boards, inveterate eggers-on, and unwavering supporters in whatever I do.

I would like to thank Mary Ellen Verona and Farnam Jahanian for landing me in grad school in the first place, with their infectious excitement about computer science, their support and complete confidence in me, and the self-assigned roles as my friends, mentors, and advisors.

I very much appreciate the support and attention of my thesis committee, and wish to thank Reid Simmons in particular for the exacting care with which he read my thesis and for his many thoughtful comments.

Finally, many thanks to the students, staff, and faculty of Carnegie Mellon who continue to consciously create an environment that I have found to be simultaneously stimulating and safe, and especially to my friends at Carnegie Mellon who made my time there one of the most exciting (intellectually and otherwise) and happy periods of my life: Jovan and Zoran Popovic, Andrej Bauer, Will Uther, Stavros Harizopoulos, Gal Kaminka, Pat Riley, and Belinda Thom immediately leap to mind.

Contents

List of Figures

List of Algorithms

List of DsPlanners

Chapter 1

Introduction

Classical planning is the gold standard for action selection, as the suggested actions are guaranteed to be on a valid and complete path to the goal. The traditional approach to planning, general purpose planning, aims to solve any problem in any planning domain. But this capability comes at a high cost in terms of solution speed. General-purpose planners are not, in practice, able to solve large-scale problems, nor are they easily able to solve problems for real-time applications.

Researchers have explored creating special-purpose planners that can be carefully crafted to suit a particular domain and arrive at solutions quickly, but that must be tediously hand written. Others have used domain-independent learning to learn domain-specific rules from example plans or domain analysis to speed up planning search by extracting heuristics and by learning control rules to guide search, inferring grammars of appropriate solutions, and using example plans as cases on which new solutions can be modeled. All of these approaches rely on an underlying general-purpose planner.

However, for many planning domains, there are simple strategies that can find possibly suboptimal solutions to large classes of problems. Our work is on exploiting this fact to automatically learn free-standing domain-specific planning programs from observed example plans. Other researchers have also worked towards learning domain-specific planners from examples and have created algorithms that are able to develop planners with broad coverage, but they require dozens of examples and hours or even days of computation time to generate the planning program, as they aim at complete coverage of the domain.

Our hypothesis is that a single example, when analyzed, can be used to generate a domain-specific planner (dsPlanner) that covers a subset of problems in the domain. This thesis focuses on domains in which simple non-searching strategies can solve classes of

problems and on using examples which demonstrate repetition to capture the repetative structure of solutions in those domains. Clearly, there are some domains and classes of problems that this approach cannot address gracefully. In the worst case, each example is converted to a dsPlanner that solves only that one problem. We show that in many domains a single well-chosen example can demonstrate the solution for a large class of problems and discuss the limits of our approach and its applicability to various domains.

Our approach assumes a known complete action specification and then carefully analyzes each example plan to reveal the underlying rationale behind each step in the plan. This analysis uncovers the structure of the solution—particularly the sequencing of actions, conditions for those actions, and looping structures—which we capture in learned dsPlanners. We show that our algorithms can learn a dsPlanner with broad coverage from a single example in under a second for many common domains. These learned dsPlanners can then be used on their own—without an underlying general-purpose planner—to solve new arbitrarily large problems in linear time, thus bringing the guarantees of planning to large problems and real-time applications automatically, without having to hand-craft a new planner for each new domain.

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

This thesis presents our work towards the goal of automatically learned scalable and real-time planners. Figure 1.1 shows a pictoral guide to this work. Chapter 2 presents the SPRAWL algorithm for analyzing example plans to reveal the rationale behind action choice and ordering the underlying structure of the example, which is the foundation of our approach. In Chapter 3, we detail our novel dsPlanner programming language, which is well-defined and in addition to being learnable, results in compact, human-readable and -writable planners. Chapter 4 describes the DISTILL algorithm for learning a non-looping dsPlanner from sets of example plans supplemented by their rationales as identified by

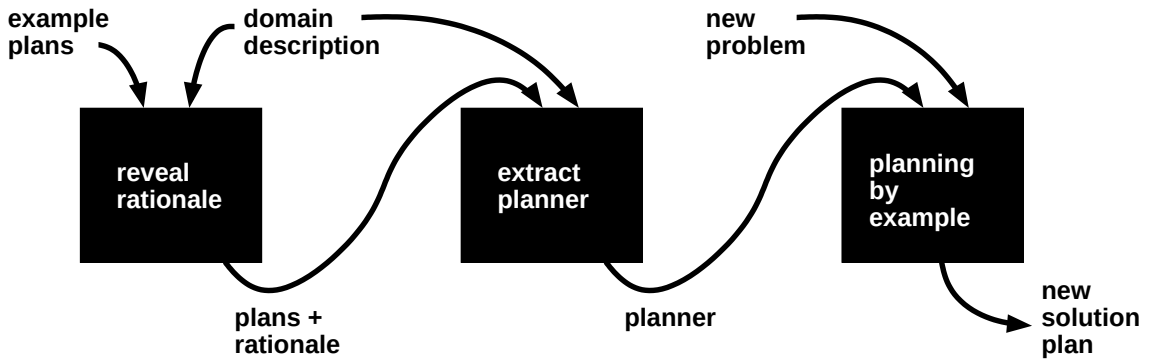


Figure 1.1: A pictoral guide to this work. First, the SPRAWL algorithm reveals the rationale behind observed example plans. The DISTILL and LoopDISTILL algorithms are two approaches towards using the SPRAWL analysis of observed plans to generate a dsPlanner. Finally, dsPlanners are used to plan by example.

SPRAWL. In Chapter 5, we present the LoopDISTILL algorithm for learning a looping dsPlanner from an example plan supplemented with its rationale. Chapter 6 explores the classes of problems that LoopDISTILL can and cannot create dsPlanners to solve by presenting example plans and learned dsPlanners in a variety of planning domains. In Chapter 7, we discuss in detail previous research related to plan analysis, learning from example plans, and program generation. Finally, in Chapter 8, we discuss the limitations and contributions of this work and explore directions for future work. We include Appedices with the domains as used in this thesis.

Chapter 2

The SPRAWL Algorithm: Extracting Rationales from Example Plans

Analyzing example plans and executions is crucial for plan adaptation and reuse, e.g., (Fikes 72), and could be useful for plan recognition and agent modeling, e.g., (Kautz 86). An example plan, represented as an initial state and goal state and a total ordering of instantiated actions, captures an execution sequence. We assume the plan steps are included in this sequence for a reason. We show that the producer/consumer relationships of preconditions and effects capture the rationale underlying the example. In Chapter 4, we show that these rationales can explain how to solve new problems in the form of a non-looping dsPlanner. In Chapter 5, we show that rationales reveal the structure underlying the observed plan, and show how to use this structure to create complex, looping dsPlanners.

A common approach to plan analysis is to create an *annotated ordering*, e.g., (Fikes 72; Regnier 91; Kambhampati 89; Kambhampati 92; Veloso 94b), in which an ordered example plan is supplemented with a rationale for the ordering constraints. Annotated orderings allow systems not only to reuse more flexibly portions of the plans they have observed, but also to reuse the reasoning that created those plans in order to solve new problems.

In recent years, the focus of the planning and agent modeling community has shifted from the simple STRIPS domain-specification language (Fikes 71) toward richer languages like ADL (Pednault 86) and PDDL that capture the conditional effects of real-world actions. Despite the success of the annotated ordering approach for simple domain-specification languages, it has not been applied to plans with conditional effects.

In this chapter, we introduce the SPRAWL algorithm for finding the rationale behind an observed totally ordered plan, represented as a list of instantiated actions, with or without

conditional effects. The rationale behind the plan explains the purpose for which each step is used in the plan and the reason behind each of the ordering constraints. We store this information in a structure we call a *minimal annotated consistent partial ordering*. A consistent partial ordering $\mathcal{P}$ of a totally ordered plan $\mathcal{T}$ is one in which all *relevant* effects (those which affect the fulfillment of the goal) active in $\mathcal{P}$ are also active in $\mathcal{T}$. We call the partial orderings found by SPRAWL *minimal* because they do not include extraneous ordering constraints; each constraint either:

- provides a term[1] upon which a relevant effect depends, or
- prevents a threat to such a term.

Finally, SPRAWL annotates each ordering constraint with the term the constraint provides or protects.

We assume that we are given or that we observe a plan as a sequence of instantiated actions that is valid, i.e., all preconditions of the steps are satisfied, and, when executed, the plan produces the goal state. SPRAWL links the steps of the plan through the literals or terms that they support. Partial orderings are capable of representing these dependencies. [2] In addition, partial orderings can isolate independent sub-plans that can be reused or recognized separately, can identify steps that do not contribute to the goals, and can identify potential parallelism.

We assume that the observed execution sequence is a totally-ordered sequence of instantiated actions. We analyze this sequence using the full domain description to relax the total ordering and understand the rationale behind the observed sequence. The annotations on the ordering constraints should *explain* the rationale behind the plans and allow portions of them easily to be matched, removed, and used independently.

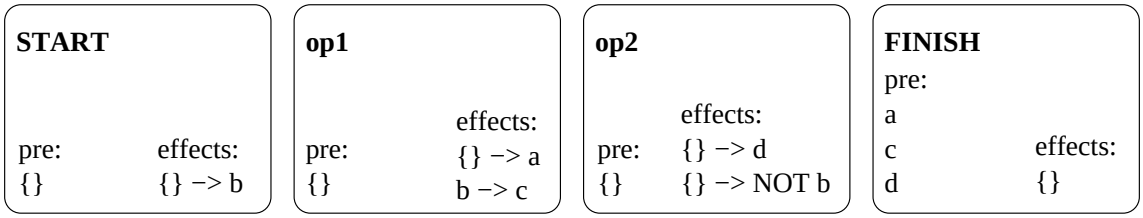
Conditional effects make the task more difficult because they cause the effects of a given step to change depending on what steps come before it, thus making step behavior difficult to predict as the ordering constraints are relaxed. In fact, any ordering must treat each conditional effect in the plan in one of three ways:

- **Use:** make sure the effect occurs;
- **Prevent:** make sure the effect does not occur;

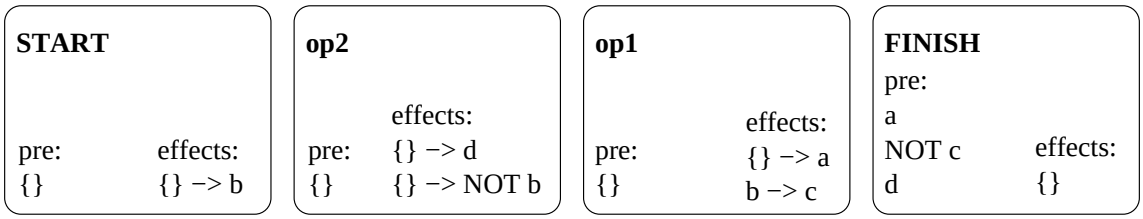
[1] A term is defined as an instantiated predicate or a negated instantiated predicate

[2] A partial order is a precedence relation $\preceq$ with the following three properties 1) reflexivity: $a \preceq a$; 2) asymmetric (no cycles): if $a \preceq b$ then not $b \preceq a$, unless $a = b$; and 3) transitivity: if $a \preceq b$ and $b \preceq c$, then $a \preceq c$. The relation is a "partial" order because there may be incomparable elements: i.e., elements a, b such that neither $a \preceq b$ nor $b \preceq a$. Note that a DAG is a partial order if we define $a \preceq b$ as a path from a to b.

use:



prevent:



ignore:

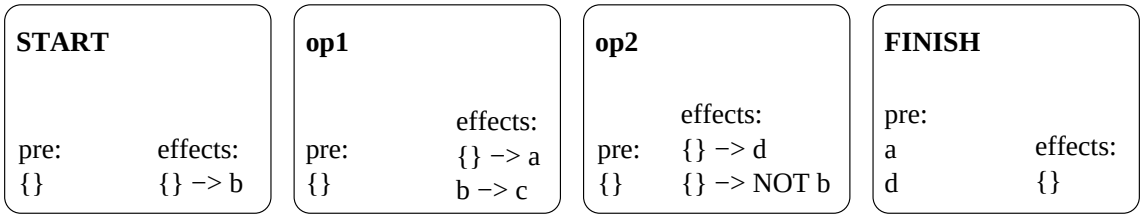


Figure 2.1: Three totally ordered plans that illustrate the three possible ways of treating a conditional effect in an ordering: using it to achieve a goal, preventing it in order to achieve a goal, or ignoring its effect.

- **Ignore:** don't care whether the effect occurs or not.

Figure 2.1 illustrates three totally ordered plans that demonstrate these cases. Note that all three plans have the same initial state and the same operators. We are able to demonstrate all three cases by changing only the goals and, in one case, the ordering of the operators. Each plan begins with a START operator with unconditional effects that produce the initial state conditions. Each plan ends with a FINISH operator with the goal state terms as preconditions. 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 body of each of the three plans consists of the same two actions, op1 and op2, but in each plan they are used differently. The conditional effect in op1 is **used** to create the goal conditions in the first plan; in the second plan the conditional effect is **prevented** from executing so it does not interfere with the goal; and it is **ignored** in the third plan, as whether it executes does not affect the achievement of the goal. In the first plan, the conditional effect of op1 is *used* to generate the goal term c. In the second plan, it is *prevented* from generating the term c, and in the third plan, the effect is irrelevant, so it is *ignored*.

Figure 2.2 shows the annotated partial orderings generated by SPRAWL for each of these cases. The ordering constraints are annotated with a rationale explaining why they are necessary. Although the plans for these three cases are composed of the same steps, SPRAWL reveals that the partial orderings are very different. In the "use" case, SPRAWL identifies that op2 threatens the goal term c (a precondition of the FINISH step), which is created by op1, and enforces the ordering op1 $\rightarrow$ op2 to protect c. In the "prevent" case, SPRAWL surmises that the step op1 must not be able to execute the conditional effect that adds the term c, and so ensures that the condition of this effect, the term b, is not true before the step executes. In this way, SPRAWL discovers the ordering constraint op2 $\stackrel{NOTb}{\rightarrow}$ op1. It also notes that the START step, since it adds b, is a threat to this link, and must therefore come before op2. Finally, SPRAWL identifies that, in the "ignore" case, the conditional effect is irrelevant, so op1 and op2 may run in parallel.

We perform needs analysis on the totally ordered plan to discover which conditional effects are relevant. Needs analysis allows us to ignore incidental conditional effects in the totally ordered plan.

Instead of looking for the optimal (according to some metric) partially ordered plan to solve a problem, we chose to focus on finding partial orderings *consistent* with the given totally ordered plan, or those in which all relevant effects were also active in the total ordering. There are two reasons for this. The first is that the totally ordered plan contains a wealth of valuable information about how to solve the problem, including which operators to use and which conditional effects are relevant. The second is that for many applications, including plan modification and reuse and agent modeling, it is important to be able to analyze an observed or previously generated plan (for example, to find characteristic patterns of behavior or to identify unnecessary steps).

However, since our purpose is to reveal underlying structure, we do have some requirements on the form of the resulting partial ordering; we allow only ordering constraints that affect the fulfillment of the goal terms—those that provide for or protect relevant effects. SPRAWL achieves this via a two-phased approach. It first uses needs analysis to identify

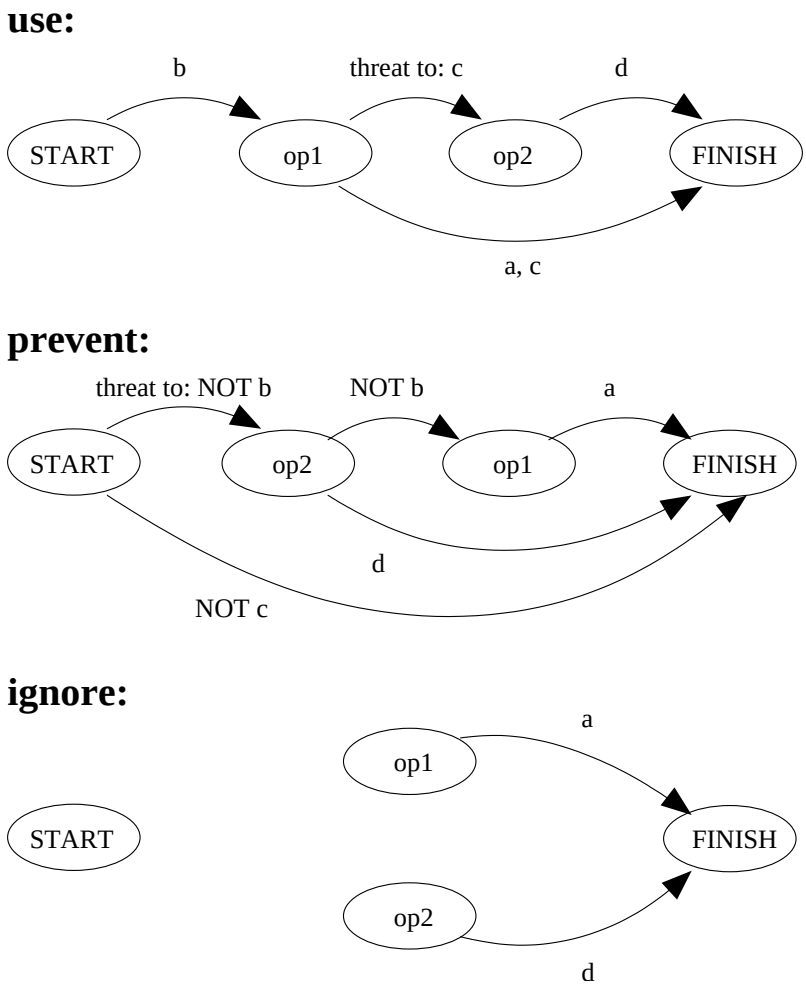


Figure 2.2: The annotated partial orderings generated by SPRAWL for the three totally ordered plans shown in Figure 2.1.

the relevant effects of each operator and the needs of each operator (which terms must be true before the operator executes in order to ensure that the relevant effects occur). SPRAWL is able to use this to find the annotated partial order by treating the operators as if they have no conditional effects—using the needs as preconditions and the relevant effects as non-conditional effects.

The remainder of this chapter is organized as follows. We introduce the needs analysis technique, illustrate its behavior and discuss its complexity. Next, we explain how the

SPRAWL algorithm uses needs analysis to find a partial ordering and discuss the complexity of the entire algorithm. We then discuss the limitations and capabilities of the algorithm and present our conclusions.

2.1 Needs Analysis

Needs analysis, the first step of the SPRAWL algorithm, computes a tree of needs for the totally ordered plan. We first create a goal step called FINISH with the terms of the goal state as preconditions. Needs analysis calculates which terms need to be true before the last step in the plan in order for the preconditions of FINISH to be true afterward. Then it calculates which need to be true before the second-to-last plan step in order for *those* terms to be true. This calculation is executed for each step of the plan, starting from the last step and finishing at the START step, creating a tree of "needs." This needs tree allows us to identify the relevant effects of a given step and most of the dependencies in the plan. However, not all threats are identified in Needs Analysis; SPRAWL uses the needs tree to calculate the remaining threats.

2.1.1 Needs Tree Structure

In this section, we discuss the needs that compose the needs tree as well as the structure of the tree. The needs tree consists of three kinds of needs:

1. **Precondition Needs:** the preconditions of a step are called *precondition needs* of the step—they must be true for the step to be executable. For example, the precondition needs of the FINISH step are the goals of the plan.

2. **Existence Needs:** terms that must be true before a step n in order for n to create a particular term or to maintain a previously existing term are called *existence needs* of the term at the step n. In the "use" example in Figure 2.2, one existence need of the term c at the step op1 is b, since op1 generates c if b is true before it executes.

3. **Protection Needs:** terms that must be true before step n in order for n not to delete a particular term are called *protection needs* of the term at the step n. In the "prevent" example in Figure 2.2, one protection need of the term NOT c at the step op1 is NOT b, since if NOT b is not true before step op1, then op1 adds c (thereby deleting NOT c). Protection needs are only necessary in domains with conditional effects.

For the sake of simplicity, instead of abstract plan steps, we illustrate the three kinds of needs using plan steps from a domain in which we have a sprinkler that, if on, can wet the yard as well as any object that may be in the yard. Figure 2.3 shows the operator sprinkle front-yard. The term on sprinkler is a *precondition need* of the step sprinkle front-yard.

sprinkle front−yard	
	effects:
pre:	{} −> wet front−yard
on sprinkler	at ?obj front−yard −> wet ?obj

Figure 2.3: The step sprinkle front-yard.

To illustrate *existence needs*, let us assume that, after executing the step sprinkle front-yard, wet shoe must be true, where shoe is an instantiation of the variable ?obj. This could be accomplished in two ways:

- by ensuring that at shoe front-yard was true before sprinkle front-yard executed;
- by ensuring that wet shoe was already true before sprinkle front-yard executed, as shown in Figure 2.4. [3]

These two terms are called *existence needs* of wet shoe at the step sprinkle front-yard, since they provide ways for the term wet shoe to be true after the step sprinkle front-yard.

We must also make a distinction between *maintain* existence needs and *create* existence needs. [4] As mentioned above, there are two ways to ensure that wet shoe is true after the execution of the step sprinkle front-yard, both illustrated in Figure 2.4. One way is for wet shoe to have been true previously. We call this a *maintain* existence need since the step does not generate the term, but simply maintains a term that was previously true. However, the step sprinkle front-yard could generate the term wet shoe if at shoe front-yard were true before the step executed. We call this a *create* existence need, since we have introduced a new need in order to satisfy another. Create existence needs are only needed in domains with conditional effects.

[3] In the remainder of the sprinkler examples, we abbreviate the literals sprinkler as sp, front-yard as fy, back-yard as by, and shoe as sh.

[4] Precondition needs and protection needs are always *create* needs.

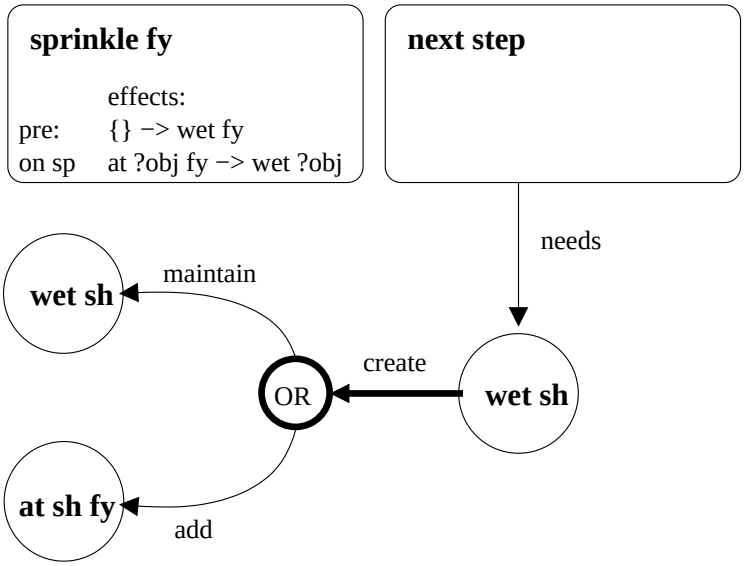


Figure 2.4: Expanding the need wet shoe (shown as wet sh in the figure) in the step sprinkle front-yard (shown as sprinkle fy in the figure). The term wet shoe may be satisfied in either of two ways; this is represented by an OR operator.

Note that, because there may be multiple ways to ensure the existence of a term, the description of needs must include the OR logical operator, as shown in Figure 2.4. It must also include the AND logical operator, since we allow a conditional effect to have multiple conditions, and in order to guarantee that the effect occurs, we must be able to specify that all must be true.

To illustrate *protection needs*, assume that, after executing the step sprinkle front-yard, the term NOT wet shoe must be true. In order to protect the term NOT wet shoe, we must ensure that NOT at shoe front-yard is true before sprinkle front-yard executes. This is called a *protection need* because it protects the term from being deleted (i.e., prevents wet shoe from being added). Protection needs are only necessary in domains with conditional effects.

It is not always necessary to generate new needs to satisfy a need term; it may also be satisfied if a non-conditional effect of the step satisfies it, as illustrated in Figure 2.5. We call such needs *accomplished*, and indicate this in our diagrams with a double circle.

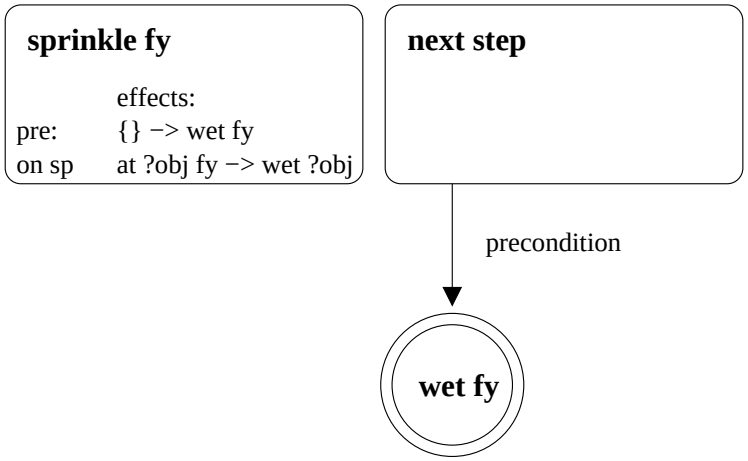


Figure 2.5: A term may be true after a particular step if a non-conditional effect of the previous step accomplishes it. We indicate this with a double circle around the term.

2.1.2 Needs Analysis Algorithm

The needs analysis algorithm is shown in Algorithm 1. We now describe in detail how needs analysis generates the needs of an individual term. Each needed term t must be created and protected from deletion; we represent this as two branches of needs: existence needs and protection needs.[5] As explained previously, t's existence needs at a particular step n are terms that must be true before step n to ensure that t is true after step n. There are two possibilities for existence needs: either t may have been true before step n, or a conditional effect of step n may generate t. [6] The protection needs of t at step n are terms that must be true before step n to ensure that step n does not delete t. Prevention needs are therefore negated conditions of any conditional effects of step n that delete t. [7] Figure 2.6 illustrates the needs tree created to satisfy each needed term.

[5]In domains with no conditional effects, there is just one branch of needs: maintain existence needs, as there are no effects that can either create the need (create existence needs) or delete it (protection needs) that have conditions—or needs—of their own.

[6]Non-conditional effects of step n that add t do not add needs—nothing needs to be true before step n in order for them to occur, which is why create existence needs aren't needed in domains without conditional effects.

[7]If t is deleted by a non-conditional effect of step n, then we call it *unsatisfiable* and end its branch of the needs tree.

Algorithm 1 Needs Analysis.

Input: Totally ordered plan $\mathcal{T} = S_1, S_2, \ldots, S_n$, the START operator S_0 with add effects set to the initial state, and the FINISH operator S_{n+1} with preconditions set to the goal state.
Output: Needs tree N.

```
procedure Needs_Analysis(T, S_0, S_{n+1})
  for all i ← n + 1 down-to 1 do
    for all preconditions p of S_i do
      Expand_Term(i, p)
    end for
  end for
end procedure

procedure Expand_Term(i, c)
  Find_Existence(i, c)
  Find_Protection(i, c)
end procedure

procedure Find_Existence(i, c)
  Add_Maintain_Existence_Need(c, c)
  for all conditional effects e of S_i do
    if e unconditionally adds c then
      c.accomplished ← true
    else if e conditionally adds c then
      Add_Conditions_To_Existence_Needs(e, c)
      for all conditions of e, c_e do
        Expand_Term(i − 1, c_e)
      end for
    end if
  end for
end procedure

procedure Find_Protection(i, c)
  for all conditional effects e of S_i do
    if e unconditionally deletes c then
      c.impossible ← true
      return
    else if e conditionally deletes c then
      Add_Conditions_To_Protection_Needs(e, c)
      for all conditions of e, c_e do
        Expand_Term(i − 1, c_e)
      end for
    end if
  end for
end procedure
```

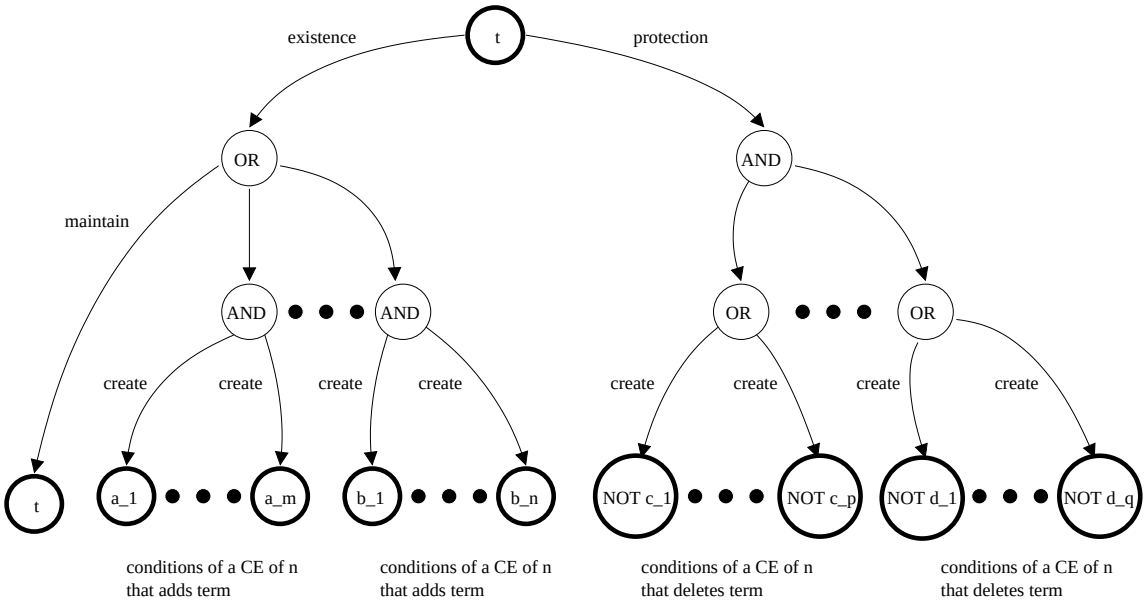


Figure 2.6: The existence needs of a need at a particular step n are calculated by finding all possible ways it can be generated in the previous step and ensuring that at least one of these occurs. The protection needs are calculated by finding all possible ways it can be deleted in the previous step and ensuring that none of these occurs.

2.1.3 Complexity of Needs Analysis Algorithm

Additions of needs are the unit of work in the algorithm. The steps of the algorithm fall into two classes: those with conditional effects and those without conditional effects. Define m to be the number of steps without conditional effects and n to be the number with conditional effects. We can estimate the complexity of performing needs analysis on a plan of $m + n$ steps by assuming the worst case: that the steps with conditional effects all happen after the steps without conditional effects.

This produces two recurrences: R, which calculates the number of need additions for the steps without conditional effects; and F, which calculates the number of need additions for the remaining steps (which have conditional effects). In these recurrences, G is the number of goal terms, P is the bound on the number of preconditions, E is the bound on the number of conditional effects in each step that has conditional effects, and C is the bound on the number of conditions per conditional effect. Recall that m is the number of steps without conditional effects and n is the number with conditional effects.

$$
\begin{aligned}
R(0) &= G \\
R(m) &= P + R(m-1) \\
R(m) &= G + mP \\
F(0,m) = R(m) &= G + mP \\
F(n,m) &= P + F(n-1,m) * EC \\
F(n,m) &= (G + mP)(EC)^n + \frac{P(EC)^n - P}{-1 + EC}
\end{aligned}
$$

The complexity of the algorithm is $O(mP(EC)^n)$. Also note that the complexity of needs analysis on a plan with no conditional effects in linear: $O(mP)$.

2.1.4 An Example with Conditional Effects

We use the totally ordered plan from the sprinkler domain shown in Figure 2.7 to illustrate the behavior of the needs analysis algorithm in domains with conditional effects. First, the algorithm analyzes the last plan step (sprinkle front-yard), which has one precondition need (on sprinkler), to determine how to satisfy the needs of the subsequent step FINISH (wet shoe and wet front-yard). As previously discussed, there are two ways for the step sprinkle front-yard to satisfy wet shoe: either wet shoe could be true before this step executes, or at shoe front-yard must be true before this step executes. So the existence needs of the term wet shoe are *maintain* wet shoe OR *create* at shoe front-yard. As for wet front-yard, the other precondition need of the FINISH step, it is accomplished by the step sprinkle front-yard since it is a non-conditional effect of the step. However, the algorithm continues to look for other ways to accomplish the term. Since there are no conditional effects of sprinkle front-yard that either generate or delete wet front-yard, the algorithm just adds the maintain existence need, *maintain* wet front-yard. Neither wet shoe nor wet front-yard have any protection needs as there are no conditional effects that could delete them.

Next, the algorithm moves back to the previous plan step, move shoe back-yard front-yard, which has the precondition need at shoe back-yard. The needs carried over from previous steps are *maintain* wet shoe OR *create* at shoe front-yard, the existence needs of wet shoe from the FINISH step; *maintain* wet front-yard, the existence need of wet front-yard from the FINISH step; and on sprinkler, the precondition need of the step sprinkle front-yard. The term at shoe front-yard is a non-conditional effect of this step, so it is accomplished, but, as with wet fy in the previous step, the algorithm adds a maintain

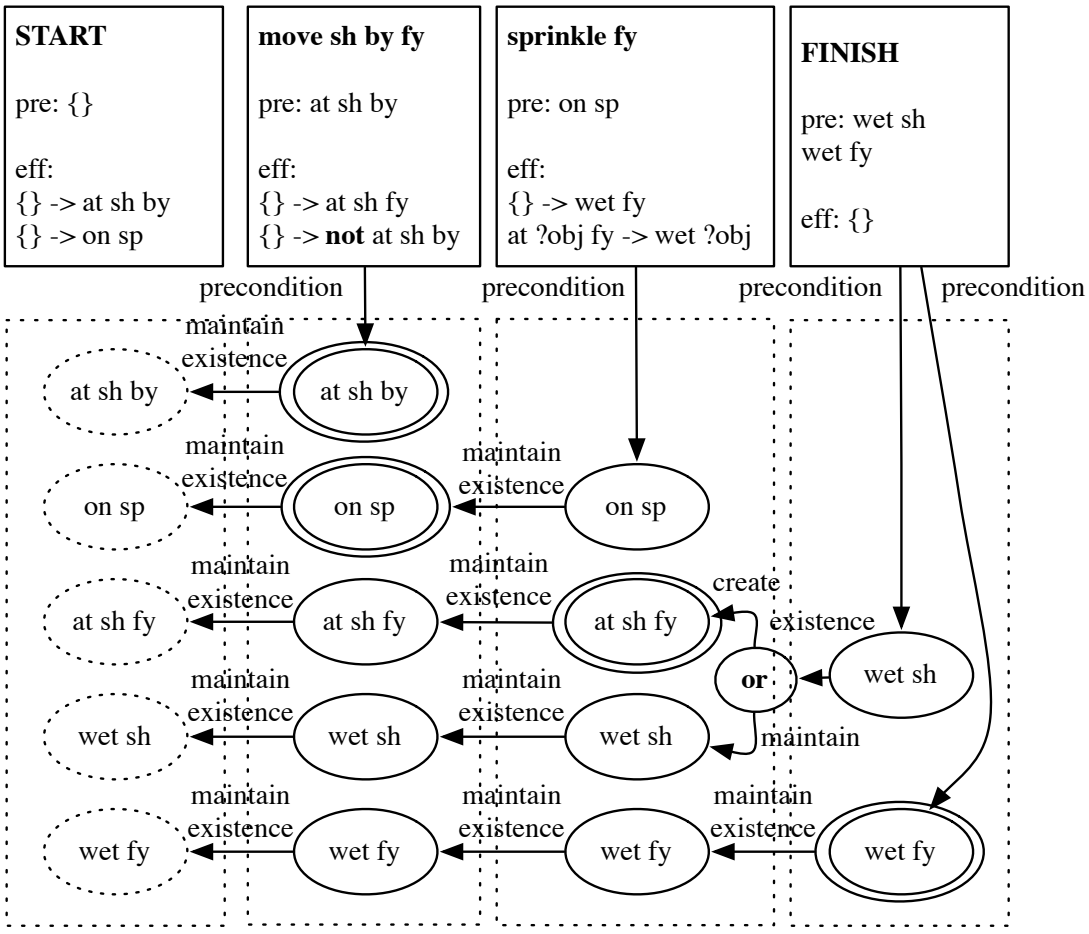


Figure 2.7: A totally ordered plan in the sprinkler domain and its complete needs tree.

existence need (*maintain* at shoe front-yard) in order to find other ways to accomplish the term. The terms *maintain* wet shoe, *maintain* wet front-yard, and on sprinkler cannot be prevented or created by this step, so each is satisfied by a maintain existence need (*maintain* wet shoe, *maintain* wet front-yard, and *maintain* on sprinkler).

Finally, the algorithm reaches the initial state, or START step, and is able to determine which branches of the needs tree can be accomplished and which can not. The remaining branches of the tree are at shoe back-yard, *maintain* at shoe front-yard, *maintain* wet shoe, *maintain* wet front-yard, and *maintain* on sprinkler. Two of the needs, at shoe

back-yard and *maintain* on sprinkler are accomplished by the START step. However, all of the other remaining needs are not accomplished by the START step. We call these needs *unsatisfiable* and indicate this in our diagrams with a dashed circle.

2.1.5 An Example without Conditional Effects

Clearly, the Needs Analysis algorithm is also able to analyze plans without conditional effects, as this is a base case of the general algorithm. We illustrate needs analysis for the totally ordered plan from the Rocket domain shown in Figure 2.8. First, the needs analysis algorithm adds a precondition need for the FINISH step: at item bos. It then analyzes the last plan step (unload item bos) to determine how to satisfy that need. The step unload item bos has two preconditions: atRocket bos and inRocket item, so these are added to the needs tree. The need at item bos of the FINISH step is pushed back as a maintain existence need. Then the needs analysis algorithm moves on to the previous step, fly lax bos. This step has one precondition: atRocket lax, which is added as a precondition need of the step. All needs of the subsequent step are propagated backwards as maintain existence needs. The needs analysis algorithm moves onto the previous step: load item lax. This step has two preconditions: atRocket lax and at item lax, so these are added as precondition needs. All needs of the subsequent step are propagated backwards as maintain existence needs. Finally, the algorithm comes to the START step, which has no preconditions. All needs of the first plan step are propagated forwards as maintain existence needs, and the needs tree is completed.

The needs analysis algorithm is now able to determine which branches of the needs tree are accomplished and which are unsatisfiable. It begins with the needs of the START step. None of these are satisfiable, as there is no previous step to supply them. It then moves forward to the needs of the next step load item lax. The needs atRocket lax and at item lax are satisfied by the previous step (START), so are marked as satisfied. All other needs of this step (at item bos, atRocket bos, inRocket item) are marked unsatisfiable. Next the algorithm proceeds to the next step fly lax bos. The need atRocket lax is satisfied because it is satisfied in the previous step and nothing deletes it. The need inRocket item is satisfied by the previous step. Those two needs are marked satisfied. All other needs of this step (atRocket bos and at item bos) are marked unsatisfiable. Next, the needs analysis algorithm moves to the next plan step, unload item bos. The precondition need atRocket bos is satisfied by the previous step, and is marked satisfied. The precondition need inRocket item is satisfied in the previous step and is not deleted by that step, so is marked satisfied. The need at item bos is marked unsatisfiable. Finally, the needs analysis algorithm reaches the FINISH step. Its precondition, at item bos, is satisfied by

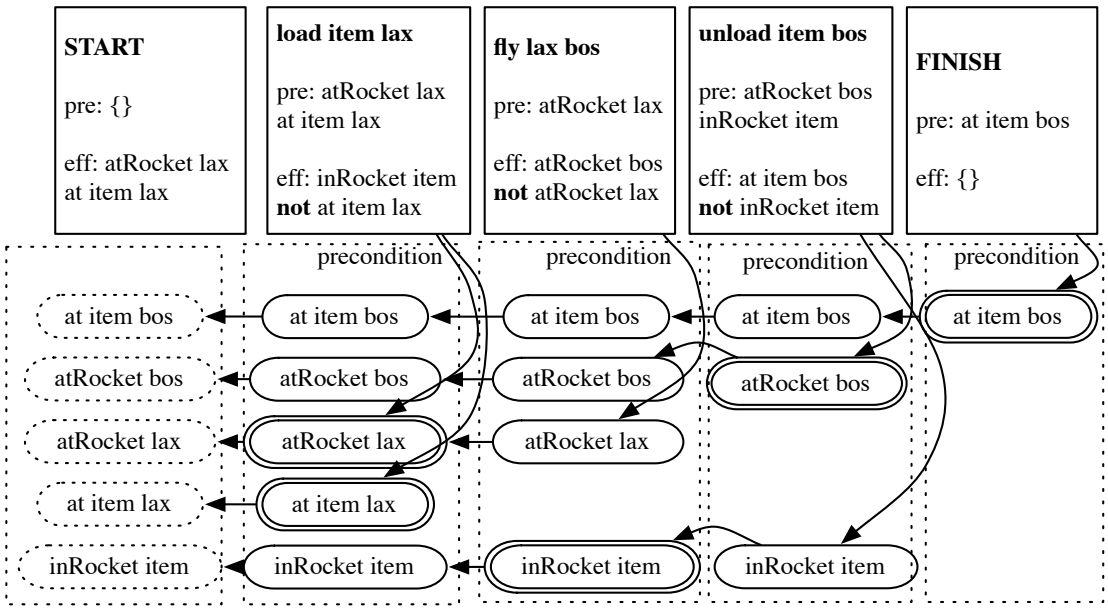


Figure 2.8: A totally ordered plan in the Rocket domain and its complete needs tree. All links in the needs tree between steps and needs are labelled as precondition needs. All links in between needs are create maintain needs; they are not labelled to help preserve the readability of the figure. The needs of the START state are all labelled unsatisfiable (surrounded by a dashed line) because there is no way to achieve these needs before the execution of the START step.

the previous step, and so is marked satisfied.

2.2 The SPRAWL Algorithm

Algorithm 2 shows the SPRAWL partial ordering algorithm. SPRAWL performs needs analysis, then walks backwards along the needs tree and adds causal links in the partial ordering between steps that need terms and the steps that generate them.

Algorithm 2 The SPRAWL algorithm.

Input: Totally ordered plan $\mathcal{T} = S_1, S_2, \ldots, S_n$, the START operator S_0 with add effects set to the initial state, and the FINISH operator S_{n+1} with preconditions set to the goal state.

Output: A minimal annotated consistent partially ordered plan shown as a directed graph $\mathcal{P}$.

```
procedure Find_Partial_Order(T, S_0, S_{n+1})
  tree ← Needs_Analysis(T, S_0, S_{n+1})
  tree ← Trim_Unsatisfiable_Need_Tree_Branches(tree)
  for all i ← n + 1 down-to 1 do
    for all preconditions p of S_i do
      Recurse_Need(i, p, P)
    end for
  end for
  Handle_Threats(tree, P)
end procedure

procedure Recurse_Need(i, p, P)
  Add_Causal_Link(choose one way to provide for p, S_c, P)
  Recurse_Need(i − 1, p.existence, P)
  Recurse_Need(i − 1, p.protection, P)
end procedure

procedure Handle_Threats(tree, P)
  for all causal links S_i → S_j do
    for all c ← 1 up-to i − 1 do
      if Threatens(S_c, S_i → S_j) then
        DEMOTE: Add_Causal_Link(S_c, S_i, P)
      end if
    end for
    for all c ← j + 1 up-to n do
      if Threatens(S_c, S_i → S_j) then
        PROMOTE: Add_Causal_Link(S_j, S_c, P)
      end if
    end for
  end for
end procedure
```

2.2.1 Resolving Threats

We rely heavily on the totally ordered plan to help us resolve threats. There are three ways to resolve threats in a plan with conditional effects (Weld 94):[8]

[8]Only promotion and demotion may be used in plans without conditional effects, as confrontation relies on preventing a conditional effect from occuring or ensuring that it occurs by adding preconditions to the threatening operator.

1. **Promotion** moves the threatened operators before the threatening operator;

2. **Demotion** moves the threatened operator after the threatening operator;

3. **Confrontation** may take place when the threatening effect is conditional. It adds preconditions to the threatening operator to prevent the effect causing the threat from occurring.

To find all possible partial orderings, all these possibilities should be explored. However, since we are provided the totally ordered plan, we do not need to search at all to find a feasible way to resolve the threat; we can simply resolve it in the same way it was resolved in the totally ordered plan. In fact, if threats are resolved in a different way, then the resulting partial ordering would not be consistent with the totally ordered plan.

If, in the totally ordered plan, the threatening operator occurs before the threatened operators, then promotion should be used to resolve the threat in the partial ordering. Similarly, if it occurs after the threatened operators, demotion should be used to resolve the threat in the partial ordering. If the threatening operator occurs between the threatened operators in the totally ordered plan, then we know that confrontation must have been used in the totally ordered plan to prevent the threatening conditional effect from occurring. Needs analysis takes care of confrontation with *protection needs*, shown in Figure 2.6, which ensure that steps that occur between a needed term's creation and use in the totally ordered plan do not delete the term.

2.2.2 Complexity of the SPRAWL Algorithm

The complexity analysis of the Needs Analysis algorithm is very useful in discussing the complexity of partial ordering using needs analysis; the complexity bound of the needs analysis algorithm is also the bound on the size of the needs tree since we used need additions as the unit of work in our analysis.

The complexities of the first two steps in the partial ordering algorithm shown in Algorithm 2 are the same as the complexity of needs analysis. The first step of the SPRAWL algorithm is Needs Analysis and the next two steps traverse the needs tree; since the size of the needs tree is bounded by the complexity of the Needs Analysis algorithm, as discussed above, the complexity of traversing the tree is also the same as the complexity of the needs analysis algorhthm. The maximum number of causal links is $((m+n)+2)^2$, since there could be a causal link between any two steps, including the **START** and **FINISH** steps. Therefore, the complexity of threat resolution is as most $((m+n)+2)^2 * (m+n) * A$,

where A is a bound on the number of adds and deletes per step. The complexity of removing transitive edges fromt eh partial order is at most $((m+n)+2)^3$, since each of the at most $((m+n)+2)^2$ causal links is checked against at most $(m+n)+2$ other links. The complexity of the entire partial ordering algorithm is then:

$$3 * O(mP(EC)^n) + ((m+n)+2)^2 * (m+n) * A + ((m+n)+2)^3 = O(mP(EC)^n + A * (m+n+2)^3)$$

2.2.3 An Example with Conditional Effects

We illustrate how SPRAWL solves problems in domains with conditional effects by stepping through its execution on the sprinkler domain problem shown in Figure 2.7. The first step of the SPRAWL algorithm is to generate the full needs tree, as shown in Figure 2.7. Then SPRAWL trims the needs tree by propagating forward unsatisfiable markings until it reaches a satisfied need, and by propagating forwards satisfied markings as well. This results in the needs tree shown in Figure 2.9.

Now SPRAWL begins to search for the dependencies in the plan. It begins with the final step and moves backwards. The precondition need of the FINISH step wet sh is satisfied by the previous step, sprinkle fy, so a link is added to the partial ordering from sprinkle fy to FINISH labelled with wet sh. The precondition need wet fy is also satisfied by sprinkle fy, so the link from sprinkle fy to FINISH is also labelled with wet fy.

Next SPRAWL moves back to the step sprinkle fy. It has one precondition need, on sp, which is satisfied by the step START, so a link is added to the partial ordering from START to sprinkle fy labelled with on sp. The create existence need at sh fy is satisfied by move sh by fy, so a link is added to the partial ordering from move sh by fy to sprinkle fy labelled with at sh fy.

Then SPRAWL moves on to the step move sh by fy. Its precondition need at sh by is satisfied by START, so a link is added to the partial ordering from START to move sh by fy labelled with at sh by.

Finally, SPRAWL moves to the START step and determines that none of the needs are satisfiable. The preliminary partial ordering has now been found, and is shown in Figure 2.10.

Finally, SPRAWL looks through all links in the plan to find and resolve any threats. There are no threats in this plan, so the minimal annotated consistent partial ordering is found, and is shown in Figure 2.10.

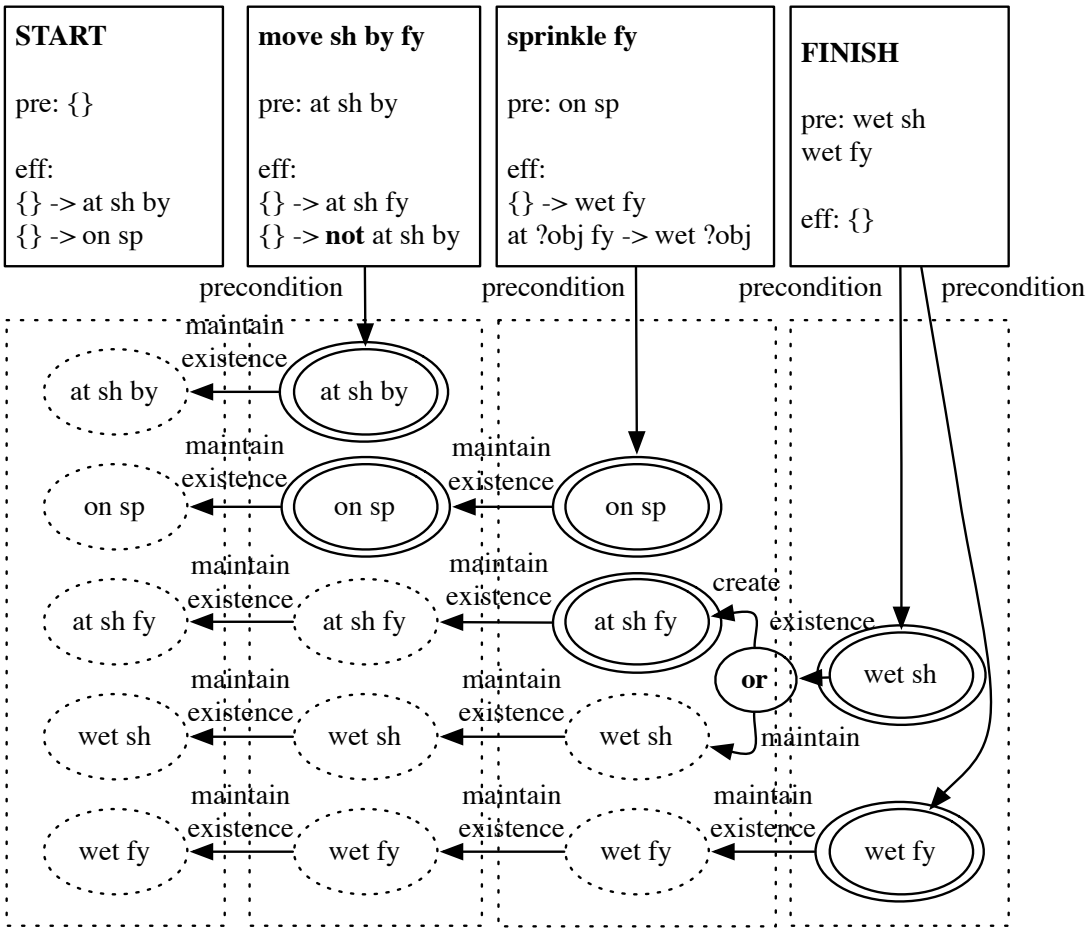


Figure 2.9: A totally ordered plan in the Sprinkler domain and its trimmed needs tree. Unsatisfiable needs are surrounded by dotted circles. Satisfied needs are surrounded by double circles.

2.2.4 An Example without Conditional Effects

We illustrate the execution of SPRAWL in domains with no conditional effects with the Rocket domain example shown in Figure 2.8. The first step in the SPRAWL algorithm is to generate the complete needs tree for the observed plan, as shown in Figure 2.8. The second step is to trim the unsatisfiable branches of the needs tree by propagating unsatisfiable markings backwards until they reach a need that is satisfied, and by propagating satisfied

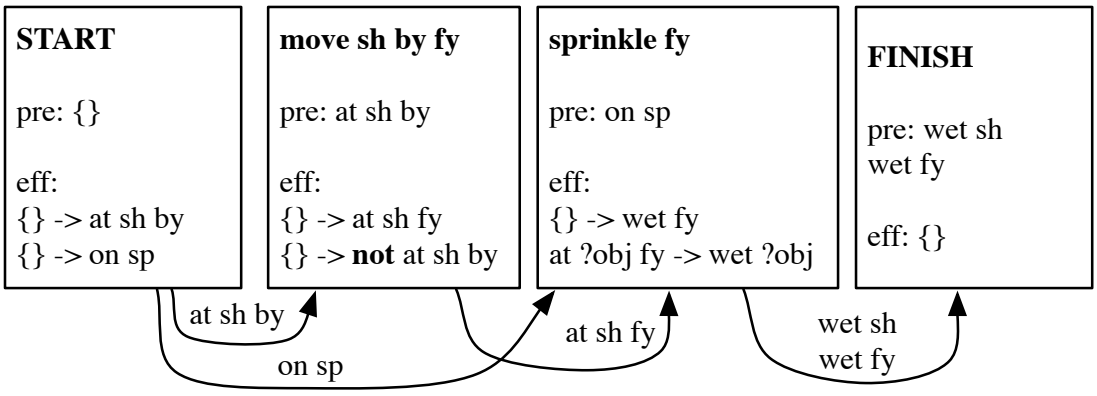


Figure 2.10: The totally ordered plan in the Sprinkler domain shown in Figure 2.7 along with the annotated ordering constraints that make up the full minimal annotated consistent partial ordering found by SPRAWL.

markings backwards as well. The results of this on the Rocket domain example are shown in Figure 2.11.

SPRAWL then begins to build the partial ordering by stepping backwards through the needs tree and determining what steps provide for the needs in the tree. It first examines the FINISH step and determines that the precondition need at item bos is satisfied by the previous step, unload item bos. A link is added to the partial ordering from unload item bos to FINISH, labelled with at item bos.

Next, SPRAWL steps back to the step unload item bos to determine how to provide for its needs. Its precondition need atRocket bos is satisfied by the previous step fly lax bos, so a link is added to the partial ordering from fly lax bos to unload item bos, labelled with atRocket bos. The precondition need inRocket item is satisfied by the step load item lax, so a link is added to the partial ordering between load item lax and unload item bos labelled with inRocket item.

Then SPRAWL moves backwards to the step fly lax bos. Its precondition need atRocket lax is satisfied by the START step, so a link is added to the partial ordering from START to fly lax bos, labelled with atRocket lax. Its maintain existence need inRocket item has already been accounted for, as a link was added from its producer to its consumer.

SPRAWL steps backwards to the step load item lax. Its precondition need atRocket lax is satisfied by the START state, so a link is added to the partial ordering from START

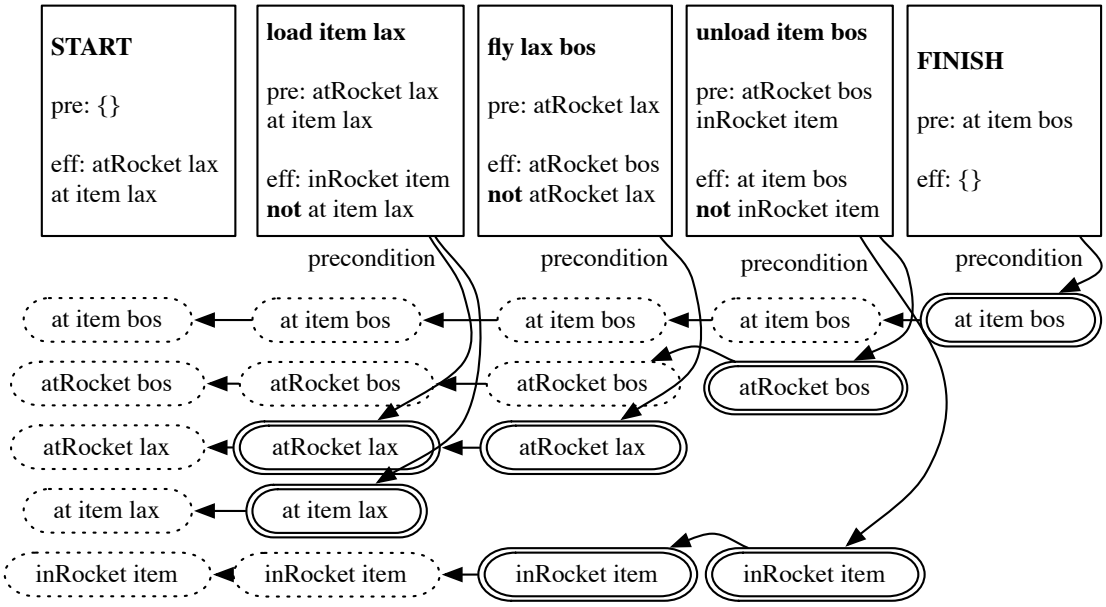


Figure 2.11: A totally ordered plan in the Rocket domain and its trimmed needs tree. Unsatisfiable needs are surrounded by dotted circles. Satisfied needs are surrounded by double circles.

to load item lax, labelled with atRocket lax. Its precondition need at item lax is also satisfied by the START state, so the link between START and load item lax is given another label: at item lax.

Finally, SPRAWL moves back to the initial state and finds that it has no satisfiable needs. The initial partial ordering has been created. It is shown in Figure 2.12. Note that this is a true partial ordering—at this point, the load and fly operators are unordered.

Now SPRAWL searches for threats. It goes one link at a time and determines whether the link is threatened by any plan step. In this example, one link is threatened. The atRocket lax link between START and load item lax is threatened by fly lax bos, since there is no ordering between load item lax and fly lax bos. SPRAWL uses the threat resolution method seen in the total ordering and places a threat link in the partial ordering between load item lax and fly lax bos, as shown in Figure 2.13. The minimal annotated consistent partial ordering (MACPO) of the observed totally ordered plan has now been found.

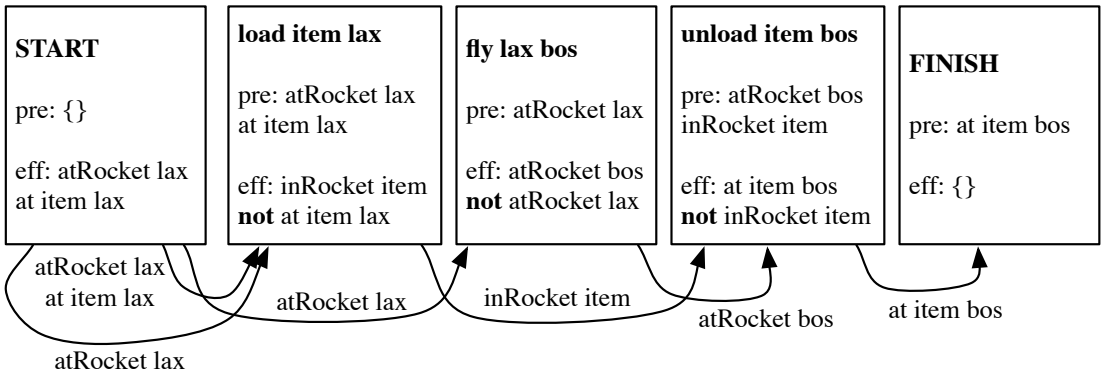


Figure 2.12: The totally ordered plan in the Rocket domain shown in Figure 2.8 along with the preliminary partial ordering found by SPRAWL. This includes all dependencies but does not include threat orderings.

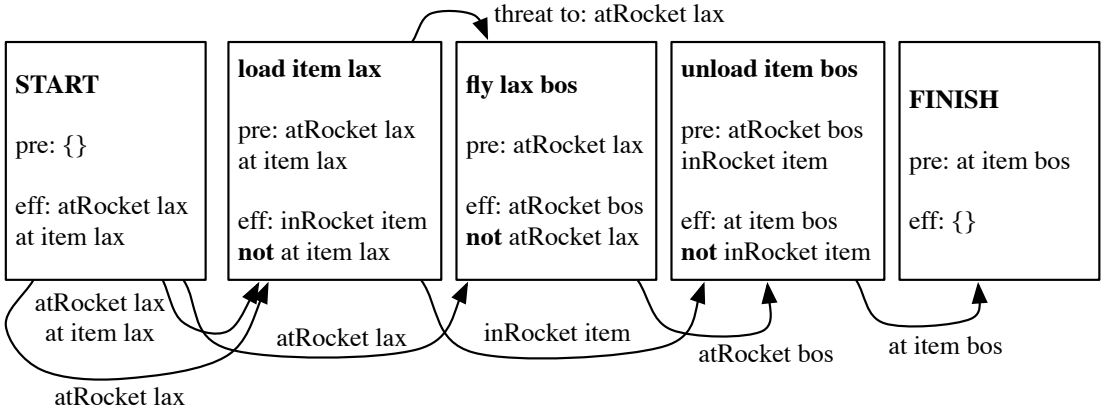


Figure 2.13: The totally ordered plan in the Rocket domain shown in Figure 2.8 along with the annotated ordering constraints that make up the full minimal annotated consistent partial ordering found by SPRAWL.

2.3 Discussion

Because the purpose of the SPRAWL algorithm is to reveal the structure underlying the observed plan, we restrict SPRAWL to *minimal* partial orderings. We define minimal partial

orderings not to have ordering constraints unnecessary to achieving the goals of the problem, as discussed at the beginning of the chapter. As each ordering constraint is guaranteed to be necessary to achieve the goals, we are able to annotate each ordering constraint with the reason it is necessary—the term provided or protected by the ordering.

The SPRAWL algorithm does not create a partially ordered plan from scratch; its purpose is to create an annotated partial ordering of the steps of an observed totally ordered execution to aid in our understanding of the structure of the observed plan. Because of this, we restrict SPRAWL to partial orderings consistent with the totally ordered plan.

However, frequently there are many partial orderings consistent with the totally ordered plan. Here, we discuss the space of possibilities explored by SPRAWL as we have described it, and how that space can be extended to include all possible minimal partial orderings consistent with the totally ordered plan.

2.3.1 Different Total Orderings of the Same Steps May Produce Different Partial Orderings

In some cases, a different total ordering of the same plan steps would produce a different partial ordering, but these are cases in which the relevant effects differ. For example, the use and prevent cases shown in Figure 2.1 consist of the same initial states and the same operators. However, because the total orderings and goal states differ, the relevant effects also differ. SPRAWL would never produce the same partial ordering for both of them; the partial orderings would each preserve the same relevant effects as are active in the respective totally ordered plans. The minimal annotated consistent partial orderings found by SPRAWL are shown in Figure 2.2.

2.3.2 Active Conditional Effects May Differ from Those in Totally Ordered Plan

Though SPRAWL is restricted to partial orderings consistent with the totally ordered plan it is given, this does not mean that all conditional effects active in the totally ordered plan must be active in the partial ordering, or vice versa. There are sometimes irrelevant conditional effects in the totally ordered plan or in the partial ordering, and SPRAWL does not seek to maintain or prevent these irrelevant effects. The ignore case shown as a totally ordered plan in Figure 2.1 demonstrates this. In this problem, one of the active conditional effects in the totally ordered plan is the effect $\mathsf{b} \rightarrow \mathsf{c}$ from step $op1$. However, this effect does not affect the fulfillment of the goal state, and so is not a relevant effect. In fact, as

is shown in Figure 2.2, SPRAWL would enforce no ordering constraints between the two steps in its partial ordering. Though the different orderings produce different final states, the goal terms are true in each of these final states, so it doesn't matter which occurs.

2.3.3 Finding Multiple Partial Orderings

Although, as we discussed, SPRAWL is restricted to partial orderings with no relevant effects not active in the given totally ordered plan, this does not mean that all relevant effects in the totally ordered plan must be relevant effects in the partial ordering. Thus, there could be several possible minimal annotated consistent partial orderings.

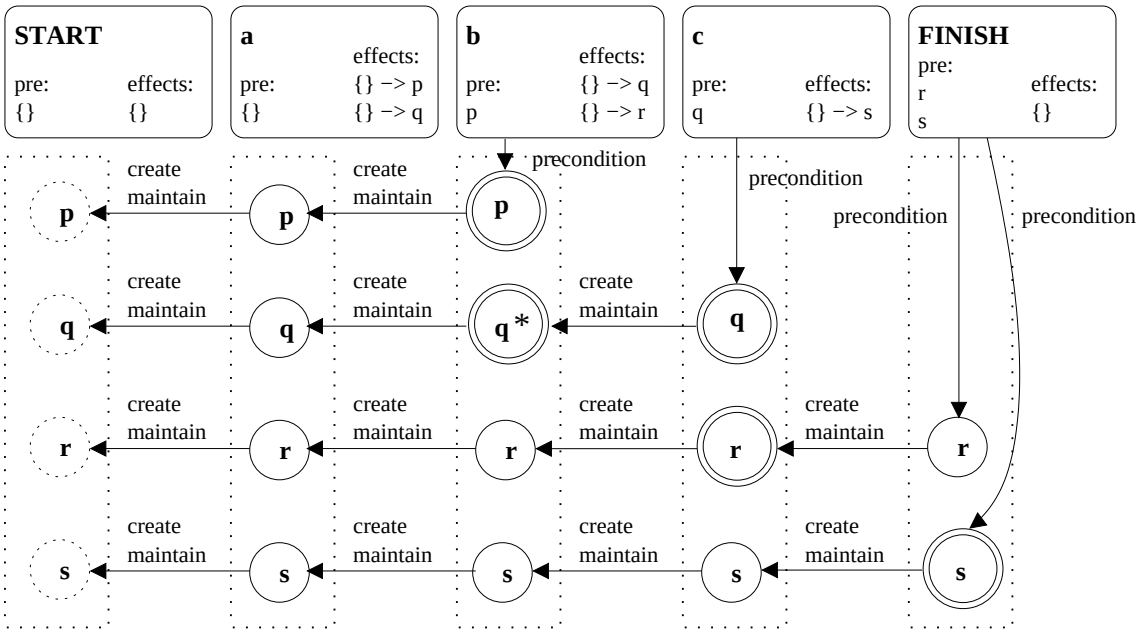


Figure 2.14: An example plan with multiple possible partial orderings (Bäckström 93), and the needs tree created if the algorithm does not terminate branches when they are accomplished. Note that the term q is accomplished by two different steps: a and b. SPRAWL can find both of the two possible partial orderings: one in which step a provides q to step c, and one in which b does. If branches are terminated as they are accomplished, the accomplished need marked q*, which represents step a providing q to step c, would not be found.

Sometimes, there are several relevant effects in the totally ordered plan that achieve the

same aim. Figure 2.14 illustrates an example that neatly illustrates this (Bäckström 93). The totally ordered plan is shown with its needs tree. In this plan, two different relevant effects provide the term q to step c—both step a and step b generate q. Choosing a different relevant effect to generate q creates a different partial order. The two partial orders representing each of the two relevant effect choices are shown in Figures 2.15 and 2.16.

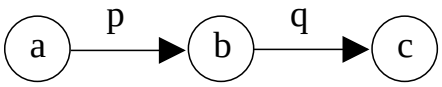


Figure 2.15: The partial ordering of the plan shown in Figure 2.14 if branches are terminated as the needs they fulfill are accomplished.

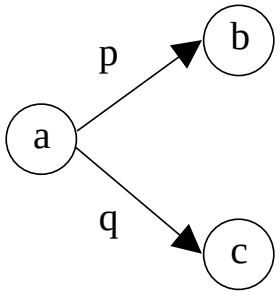


Figure 2.16: Another partial ordering of the example plan shown in Figure 2.14. This ordering can also be discovered by SPRAWL if branches of the needs tree are not terminated as the needs they fulfill are accomplished.

In the interest of efficiency, SPRAWL finds exactly one minimal annotated consistent partial ordering and does not search through different choices to find a "better" one according to any measure. The needs analysis algorithm shown in Table 1 produces a needs tree that encompasses all possible partial orderings consistent with the totally ordered plan, but the version of SPRAWL shown in Table 2 arbitrarily chooses one possible partial ordering from those represented by the needs tree. SPRAWL can be modified to search through more possible partial orderings, however, finding the best partial ordering according to any measure is NP-complete (Bäckström 93). It is our conjecture that the chosen partial ordering affects how an example plan can be reused.

When an OR logical operator is encountered in the needs tree, SPRAWL arbitrarily chooses which of its branches to follow and ignores the others (Algorithm 2). Instead, we could search through the possibilities to find the branch that contributes to the best partial ordering.

Similarly, there is sometimes more than one way to accomplish a need, as with the need q in Figure 2.14. SPRAWL arbitrarily chooses one of these ways to be the need's creator in the partial ordering (Algorithm 2). Again, we could search through all possibilities instead, and choose the one that contributes to the best partial ordering.

Finally, SPRAWL resolves threats in the same way they were resolved in the totally ordered plan. It is possible instead to search over all three ways (promotion, demotion and confrontation) to resolve each. However, the partial ordering is only consistent with the totally ordered plan if threats are resolved in the same way.

2.4 Summary

In this chapter, we have presented the SPRAWL algorithm for uncovering the rationale behind observed totally ordered executions in the form of minimal annotated consistent partial orderings. We first presented our novel Needs Analysis approach to finding the relevant effects and needs of each operator, presented the needs analysis algorithm in detail, discussed its complexity, and illustrated its behavior with examples. We then presented and explained the complete SPRAWL algorithm for finding minimal annotated partial orderings, discussed its complexity, and illustrated its behavior with examples. Finally, we discussed the limitations of the algorithm and some techniques which can extend its capabilities.

Chapter 3

Defining and Using dsPlanners

In this section, we introduce our representation for automatically learned domain-specific planners, or dsPlanners, and explain how they are used for planning.

3.1 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. The dsPlanner is a novel method of representing 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 and **endwhile** statements;
- **if** , **else** , and **endif** statements;
- logical structures (**and** , **or** , **not**);
- **inGoalState** and **inCurState** boolean predicates;
- numbered and typed variables;
- the "v" variant indicator for **while** loops, indicating which parameters may vary across loop iterations;
- instantiated plan predicates; and
- instantiated plan operators.

As it is a principled language devised to enable learning of programmatic structures, the conditions and constructs of the language range over a well-defined set.

Variables are introduced in if-statement and while-loop conditions. Any objects in the problem that 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. Multiple variables within the conditions of an if-statement or while-loop may not map to the same object, as this can lead to incorrect plans.

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

DsPlanners are *programs*; therefore when an if-statement or while-loop is encountered, if the conditions do not match the current state, the program moves on to the next statement. In this sense, a dsPlanner can trivially solve problems that are simplifications of the original example used for learning.

The dsPlanner language is rich enough to allow compact planners for many difficult problems. We demonstrate this by presenting three short hand-written dsPlanners that solve all problems in well-known domains.

DsPlanner 1 shows a simple but suboptimal hand-written dsPlanner that solves all BlocksWorld-domain (Winograd 72) 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, for each block that is stacked on a second block in the goal state, if the second block is already stacked on a third, then go ahead and stack the first block on the second.

DsPlanner 2 solves all gripper-domain problems involving moving balls between rooms. The dsPlanner is composed of one while loop: while there is a 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.

DsPlanner3 shows a hand-written dsPlanner that solves all Rocket-domain (Veloso 94a) 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

DsPlanner 1 A dsPlanner that solves all BlocksWorld-domain problems.

```
while inCurState (on-block(v?1:block v?2:block)) and inCurState (clear(v?1:block)) do
  move-block-to-table(?1 ?2)
end while
while inGoalState (on-block(v?1:block v?2:block)) and not (inGoalState (on-block(v?2:block
v?3:block))) and inCurState (clear(v?1:block)) and inCurState (clear(v?2:block)) do
  move-table-to-block(?1 ?2)
end while
while inGoalState (on-block(v?1:block v?2:block)) and inCurState (on-table(v?1:block)) and in-
CurState (on-block(v?2:block v?3:block)) do
  move-table-to-block(?1 ?2)
end while
```

DsPlanner 2 A 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 inGoalState (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
```

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.

3.2 Planning with DsPlanners

When 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*:

DsPlanner 3 A dsPlanner that solves all rocket-domain problems.

```
while inGoalState (at(v?1:pkg v?2:city)) and not (inCurState (at(v?1:pkg v?2:city))) do
  while inCurState (inside(v?3:pkg v?4:rocket)) and inGoalState (at(v?3:pkg v?5:city)) do
    while inCurState (inside(v?6:pkg ?4:rocket)) and inCurState (at(?4:rocket v?7:city)) and in-
    GoalState (at(v?6:pkg v?7:city)) do
      unload(?6 ?4 ?7)
    end while
    while inCurState (at(?4:rocket v?6:city)) and inCurState (at(v?7:pkg v?6:city)) and inGoal-
    State (at(v?7:pkg v?8:city)) do
      load(?7 ?4 ?6)
    end while
    if inCurState (at(?4:rocket ?6:city)) and inGoalState (at(?1:pkg ?7:city)) and not (inCurState
    (at(?1:pkg ?7:city))) then
      fly(?4 ?6 ?5)
    end if
  end while
  if inCurState (at(?1:pkg ?3:city)) and inCurState (at(?4:rocket ?5:city)) and inGoalState
  (at(?1:pkg ?6:city)) then
    fly(?4 ?5 ?3)
  end if
  if inCurState (at(?1:pkg ?3:city)) and inCurState (at(?4:rocket ?3:city)) and inGoalState
  (at(?1:pkg ?5:city)) then
    load(?1 ?4 ?3)
  end if
end while
while inCurState (at(v?1:rocket v?2:city)) and inGoalState (at(v?1:rocket v?3:city)) do
  fly(?1 ?2 ?3)
end while
```

it consists of applying each of the statements to the current state. Each statement in the dsPlanner is either a plan step (e.g., unload, load, fly, etc.), an if statement, or a while loop. If the current statement is a plan step, 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, 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.

Sometimes there may be many ways to apply an if statement or a while loop to the current state. For example, there may be several possible variable assignments for a given statement. One of our primary assumptions is that any valid variable binding may be

Initial State	Goal State
at pkg1 city1 inside pkg2 rocket inside pkg3 rocket at rocket city2	at pkg3 city2 at pkg1 city2

Table 3.1: An example problem in the Rocket domain

chosen.

3.3 Rocket Domain Example

As an example, consider executing the dsPlanner shown in DsPlanner 3 on the Rocket domain problem shown in Table 3.3. The first while loop could match either pkg3 and city2 or pkg1 and city2—at the beginning of execution, neither package is at its goal destination. Let's assume it matches pkg3 and city2. The second while loop can find only one match: pkg3, rocket, and city2. While pkg2 is also in rocket, the goal state does not specify its destination, and therefore the inGoalState component of the while loop conditions does not match. The third while loop also finds a match: pkg3, rocket, and city2, so the plan step contained within it is added to the result plan: unload(pkg3 rocket city2), and its execution is simulated to update the current state within the dsPlanner.

The third while loop finds no more matches (there are no more packages inside the rocket that have the rocket's current location as their goal destinations), so it completes execution. The fourth while loop cannot find a match—it is looking for a package in the same city as the rocket that has a different destination location, and there is none, so it is skipped. The if statement that follows is checking whether the rocket needs to move to unload the package inside it (matched in the second while loop). In this case, it does not—the package has already been unloaded—so it is skipped as well. The end of the first iteration of the second while loop has now been reached. The dsPlanner reevaluates whether it can match the conditions of the second loop—are there any objects inside the rocket that have a city as their goal destinations? There are not, so the while loop completes execution, and the dsPlanner moves on to the if statement below.

Recall that the dsPlanner is still executing the first while loop, in which it matched pkg3 and city2. The next if statement checks whether pkg3 is in a different city than rocket and needs to move. This is not true, so the if statement is skipped. The next if statement checks whether pkg3 is in the same city as rocket and needs to move. This is

not true, so the if statement is skipped.

The dsPlanner has now completed the first iteration of the first while loop, and is ready to try to match the conditions again. The conditions now have exactly one match: pkg1 and city2. The second while loop no longer has a match, as there are no packages inside the rocket that need to be delivered anywhere, so the second while loop is skipped. The next if statement does have a match: pkg1, city1, rocket, and city2, so the plan step fly(rocket city2 city1) is added to the result plan and its execution is simulated to update the current state within the dsPlanner.

When the next if statement is evaluated, rocket has been moved, so it is now in the same city as pkg1, and the conditions of the if statement match. The plan step in the body of the if statement, load(pkg1 rocket city1), is added to the result plan and its execution is simulated to update the current state within the dsPlanner.

The first while loop has completed its second iteration and is now evaluated again to determine whether its conditions can be matched. They can, again to pkg1 and city2. This time, the second while loop matches—pkg1, rocket, city2. The third while loop does not match, as rocket is not at the goal destination of pkg1, so it is skipped. The fourth while loop also does not match—there are no packages at rocket's current location that need to be moved, so it is skipped as well. The next if statement does match, however, and the plan step in the body of the if statement, fly(rocket city1 city2) is added to the result plan and its execution is simulated to update the current state within the dsPlanner.

The second while loop has finished its iteration and its conditions are now reevaluated to find a match. There is a match: pkg1, rocket, and city2. The third while loop now has a match as well: pkg1, rocket, and city2, so the step in its body, unload(pkg1 rocket city2), is added to the result plan and its execution is simulated to update the current state within the dsPlanner.

The third while loop does not match again, as there are no more packages inside the rocket that need to go anywhere. The fourth while loop also doesn't match, as there are no packages that need to go anywhere (all are properly delivered). The following if statement doesn't match, as all packages are properly delivered. The second while loop does not match again, since there are no packages inside the rocket that need to be delivered. The following two if statements also do not match, as all packages are properly delivered. The first while loop has now finished its third iteration, and its conditions cannot be matched again, as all packages are in their goal destinations. The last while loop also cannot be matched, as there is no goal destination for rocket.

The dsPlanner has completed its execution and has found a correct solution plan to the problem shown in Table 3.3. The solution plan is shown in Table 3.3.

Solution plan
unload(pkg3 rocket city2)
fly(rocket city2 city1)
load(pkg1 rocket city1)
fly(rocket city1 city2)
unload(pkg1 rocket city2)

Table 3.2: The solution plan found by dsPlanner 3 for the Rocket domain problem shown in Table 3.3.

3.4 Summary

In this chapter, we contribute a principled formalism for automatically-generated domain-specific planning programs (dsPlanners), demonstrate how this dsPlanner language can be used to express planning algorithms, and discuss how to use dsPlanners for planning.

Chapter 4

DISTILL: Learning Non-Looping Domain-Specific Planners by Example

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.

We presented the SPRAWL algorithm for finding the rationale underlying observed example plans in Chapter 2 (Winner 02). The SPRAWL algorithm proves that example plans can reveal more than control information: they can also reveal the process behind their generation.

In this chapter, we describe the DISTILL algorithm, which automatically extracts complete non-looping domain-specific planners, or dsPlanners, from sets of example plans supplemented with their rationales. The learning techniques used in the DISTILL algo-

rithm allow problem solving that avoids the cost of generative planning and of maintaining exhaustive databases of observed behavior by compiling observed plans into compact 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 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 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 using the rationales uncovered by SPRAWL to convert new example plans into dsPlanners in if-statement form and merge 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 present the DISTILL algorithm and illustrate its execution. Then we present the results of using DISTILL on size-limited domains.

4.1 The DISTILL Algorithm: Learning Non-Looping dsPlanners

The DISTILL algorithm, shown in Algorithms 3 and 4, learns complete, non-repeating dsPlanners from sequences of example plans supplemented with their rationales, incrementally adapting the dsPlanner with each new plan. We describe the two main portions of the 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. 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.

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

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

```
procedure DISTILL(P, dsP_i)
  A ← Find_Variable_Assignment(P, dsP_i.variables, ∅)
  repeat
    if A = ∅ then
      can't match ← true
    else
      N ← Make_New_If_Statement(Assign(P, A))
      match ← Is_A_Match(N, dsP_i)
    end if
    if not (can't match) and not (match) then
      A ← Find_Variable_Assignment(P, dsP_i.variables, A)
    end if
  until match or can't match
  if can't match then
    A ← Find_Variable_Assignment(P, dsP_i.variables, ∅)
    N ← Make_New_If_Statement(Assign(P, A))
  end if
  dsP_{i+1} ← Add_To_dsPlanner(N, dsP_i)
end procedure
```

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 90); we show in Chapter 2 how to find minimal annotated consistent partial orderings of totally-ordered plans given a model of the operators (Winner 02).

The DISTILL algorithm converts observed plans into dsPlanners, described in Section 4.1.2, and merges them by finding dsPlanners with overlapping solutions and combining them, described in Section 4.1.3. 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.

Algorithm 4 Procedures that support the DISTILL algorithm.

```
procedure Make_New_If_Statement(P_A)
  N ← empty if statement
  for all terms t_m in initial state of P_A do
    if plan body of P_A contains a step that needs t_m or goal state of P_A needs t_m then
      Add_To_Conditions(N, inCurState (t_m))
    end if
  end for
  for all terms t_m in goal state of P_A do
    if plan body of P_A contains a step that t_m relies on then
      Add_To_Conditions(N, inGoalState (t_m))
    end if
  end for
  for all steps s_n in plan body of P_A do
    Add_To_Body(N, s_n)
  end for
  return N
end procedure

procedure Is_A_Match(N, dsP_i)
  for all if-statements I_n in dsP_i do
    if N matches I_n then
      return true
    end if
  end for
end procedure

procedure Add_To_dsPlanner(N, dsP_i)
  for all if-statements I_n in dsP_i do
    if N matches I_n then
      I_n ← Combine(I_n, N)
      return
    end if
  end for
  if N is unmatched then
    Add_To_End(N, dsP_i)
  end if
end procedure
```

4.1.1 Generalizing Situations

We make several assumptions about what makes one planning *situation* different than another, and about how the observed planner solves problems. We assume that two objects

of the same type are 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 also assume that the planner responds 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.

4.1.2 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. Two discrete objects in a plan are not allowed to map onto the same variable, as this can lead to invalid plans (Fikes 72). We also do not allow the same object to be mapped to different variables. Our goal is to model the problem-solving process that is demonstrated.

Next, the parameterized plan is converted into a dsPlanner, as formalized in the procedure Make_New_If_Statement in Algorithm 4. 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 that are needed for the plan to run correctly and achieve the goals (Veloso 94a). Relevant goal-state terms are those that the plan accomplishes. We use a minimal annotated consistent partial ordering (Winner 02) 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.

DsPlanner 4 The dsPlanner DISTILL creates to represent the plan shown in Figure 4.1.

if inCurState (f(?0:type1)) **and inCurState** (g(?1:type2)) **and inGoalState** (a(?:type1)) **and inGoalState** (d(?1:type2)) **then**
  op1
  op2
end if

Figure 4.1 shows an example minimal annotated consistent partially ordered plan with conditional effects. DsPlanner 4 is created by DISTILL 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 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

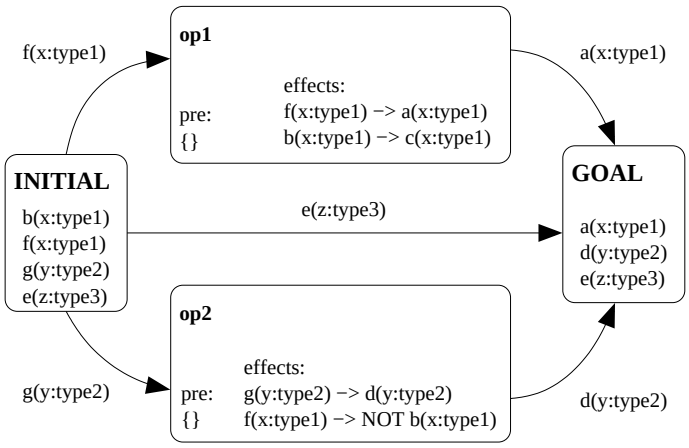


Figure 4.1: An example plan. The preconditions (pre) are listed, as are the effects, which are represented as conditional effects $a \rightarrow b$, i.e., if a then add b. A non-conditional effect that adds a literal b is then represented as $\{\} \rightarrow b$. Delete effects are represented as negated terms (e.g., $\{a\} \rightarrow NOT\ b$).

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.

4.1.3 Merging dsPlanners

The merging process is formalized in the procedure Add_To_dsPlanner in Algorithm 4. 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 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*$_1$. This process is illustrated in Figure 4.2. 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 4.2, merging *if*$_1$ and *if*$_2$ results in three new if statements. We 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** $Relevant(y, b)$. [1] 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, and is illustrated in Figure 4.3. Merging the two if statements *if*$_1$= **if** x **then** ab and *if*$_2$= **if** y **then** bc results 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.

4.2 Illustrative Results

We present results of applying DISTILL to limited domains since the DISTILL algorithm does not 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.

A dsPlanner learned by the DISTILL algorithm that solves all problems in a Blocksworld (Winograd 72) domain with two blocks is shown in DsPlanner 5. There are 2112 such problems, [2] but the dsPlanner stores 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. If more examples were given,

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

[2] Though the initial state must be fully-specified in a problem, the goal state need only be partially specified. There are only three valid fully specified states in the Blocksworld domain with two blocks (block1 on block2, block2 on block1, or both on the table), but there are 704 valid partially specified goal states.

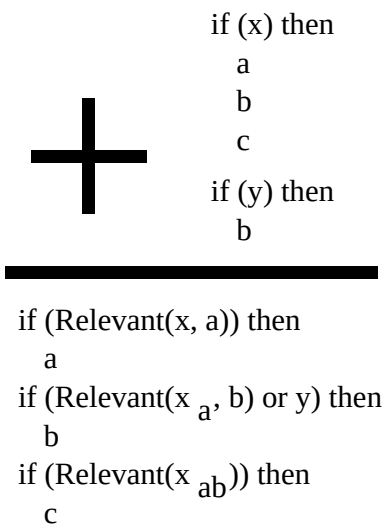


Figure 4.2: Combining two if statements when the body of one is a sub-plan of the body of the other.

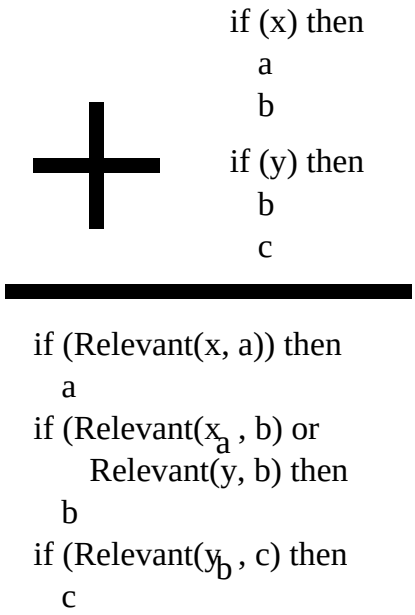


Figure 4.3: Combining two if statements when their bodies are overlapping.

the dsPlanner would *not* continue to grow. As this is a complete dsPlanner, the examples would be found to match existing if-statements and their conditions.

DsPlanner 5 A dsPlanner learned from 6 example plans by the DISTILL algorithm which solves all two-block blocks-world problems.

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

A dsPlanner learned by the DISTILL algorithm that solves all gripper-domain problems with one ball, two rooms, and one robot with one gripper arm is shown in DsPlanner 6.

Although there are 1722 such problems, [3] it is possible for the DISTILL algorithm to learn the dsPlanner from only six example plans. Also note that only five plan steps (the length of the longest plan) are stored in the dsPlanner.

DsPlanner 6 A dsPlanner learned by the DISTILL algorithm from 5 example plans that solves all gripper-domain problems involving one ball, two rooms, and one robot with one gripper arm.

if inCurState (at(?3:ball ?2:room)) **and inCurState** (at-robby(?1:room)) **and** (**inGoalState** (at(?3:ball ?1:room)) **and inGoalState** (¬at(?3:ball ?2:room)) **or inGoalState** (holding(?3:ball)) **or inGoalState** (¬free-arm)) **then**
  move(?1 ?2)
end if
if inCurState (at(?3:ball ?2:room)) **and inCurState** (at-robby(?2:room)) **and** (**inGoalState** (at(?3:ball ?1:room)) **or inGoalState** (¬at(?3:ball ?2:room)) **or inGoalState** (holding(?3:ball)) **or inGoalState** (¬free-arm)) **then**
  pick(?3 ?2)
end if
if inCurState (holding(?3:ball)) **and inCurState** (at-robby(?2:room)) **and** (**inGoalState** (at(?3:ball ?1:room)) **or inGoalState** (¬at(?3:ball ?2:room)) **and** ingoal(¬holding(?3:ball)) **or inGoalState** (free-arm)) **then**
  move(?2 ?1)
end if
if inCurState (holding(?3:ball)) **and inCurState** (at-robby(?1:room)) **and** (**inGoalState** (at(?3:ball ?1:room)) **or inGoalState** (¬holding(?3:ball)) **or inGoalState** (free-arm)) **then**
  drop(?3 ?1)
end if
if inCurState (at-robby(?1:room)) **and** (**inGoalState** (at-robby(?2:room)) **or inGoalState** (¬at-robby(?1:room))) **then**
  move(?1 ?2)
end if

Our results show that dsPlanners achieve a significant reduction in space usage compared to case-based or analogical plan libraries, as they store only the number of plan steps in the longest plan. In addition, dsPlanners are also able to situationally generalize known problems to solve problems that have not been seen; each of the learned planners we have presented are complete, despite being learned from just a few problems, and are able to solve all of the remaining problems in the domain.

[3] As previously mentioned, each problem consists of one fully-specified initial state (in this case, there are 6 valid fully-specified initial states: the ball and robby are both in room1, and robby is not holding the ball, both are in room2 and robby is not holding the ball, both are in room1 and robby is holding the ball, both are in room2 and robby is holding the ball, robby is in room1 and the ball is in room2, and the ball is in room1 and robby is in room2), and one partially-specified goal state (in this case, there are 287).

4.3 Summary

In this chapter, we 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.

Chapter 5

LoopDISTILL: Learning Domain-Specific Planners from Example Plans

General-purpose planners have traditionally had difficulty with large-scale planning problems, although many large-scale problems have a repetitive structure, because they do not capture or reason about such repetition. Instead, to solve large-scale problems, programmers have had to hand write special-purpose planners that may precisely encode the repeated structure. However, example plans are often available, and can demonstrate this structure.

Examples have previously been used to learn domain-specific knowledge that may reduce planning search, and to learn policies not guaranteed to be complete or correct. We observe that example plans can *precisely* reveal the process behind their creation. We have shown in Chapter 2 how to use domain descriptions and problem specifications to find the *rationale* behind example plans, in the form of a minimal annotated consistent partial ordering (Winner 02), introduced the concept of automatically-generated domain-specific planning programs (or dsPlanners) in Chapter 3, and, in Chapter 4 have shown how to use rationales to learn non-looping dsPlanners, which can solve problems of limited size (Winner 03). Here, we present the LoopDISTILL algorithm for *automatically* identifying the repeated structure of example plans to learn looping dsPlanners. DsPlanners execute independently of a general-purpose planning program, perform no search, and return a solution plan in time that is linear in the size of the dsPlanner and of the problem and solution, modulo state-matching effort.

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

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. Instead, the plans they find reflect the style of the example plans used to generate the dsPlanner. 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. We believe that post-processing of plans can help improve plan quality, if needed, and that dsPlanners that model observed solutions can be used not only for planning but also for prediction and agent modelling.

We first present definitions and discuss classes of loops. Then we present the LoopDISTILL algorithm for automatically identifying loops in observed plans, and illustrate its behavior with examples. We then present the results of using learned looping dsPlanners and discuss extensions and limitations of the LoopDISTILL algorithm. Finally, we draw conclusions

5.1 Definitions

Subplans are connected components within a partially-ordered plan when the initial and goal states are excluded (otherwise every set of steps would be a connected component). Three subplans of a painting and transport domain problem are illustrated in Figure 5.1.

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.

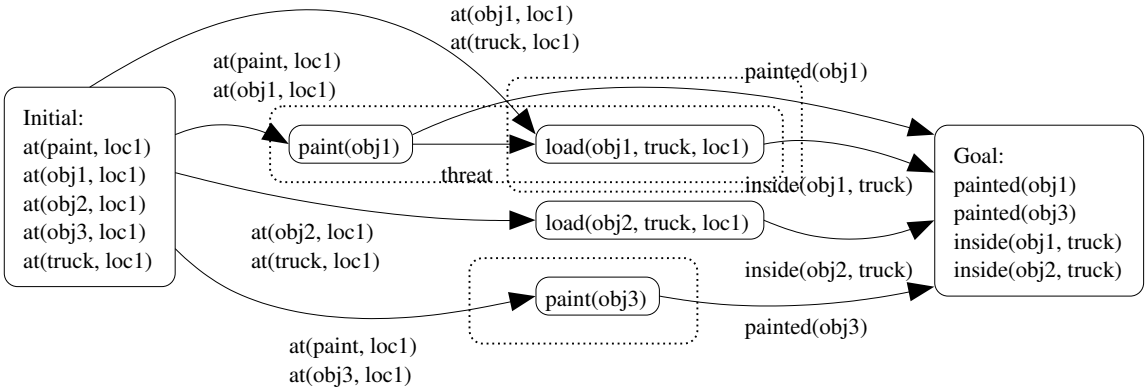


Figure 5.1: An example plan in a painting and transport domain is shown. In the given plan, some objects need to be painted and some need to be loaded into a truck. Painting must be done before loading. Three different subplans are surrounded by dotted lines. There are other possible subplans, but the steps paint(obj1) and paint(obj3) are not a subplan, since they are not a connected component within the partial ordering.

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 5.1 are matching steps, as are the two paint operators.

Parallel Subplans are causally- and threat-independent of each other. Figure 5.2 shows three 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). In Figure 5.1, the paint and load operators for obj1 are serial subplans of length 1.

An Unrolled Loop is a set of matching subplans. There are two unrolled loops in the example shown Figure 5.1; the two load operators make up an unrolled loop, and the paint operators make up the other. One of two unrolled loops is circled in Figure 5.3.

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

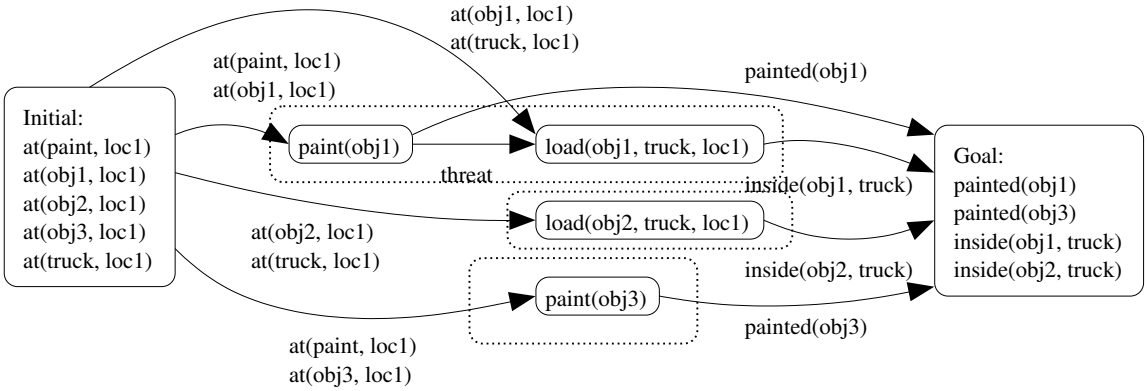


Figure 5.2: Three parallel subplans are surrounded by dotted lines.

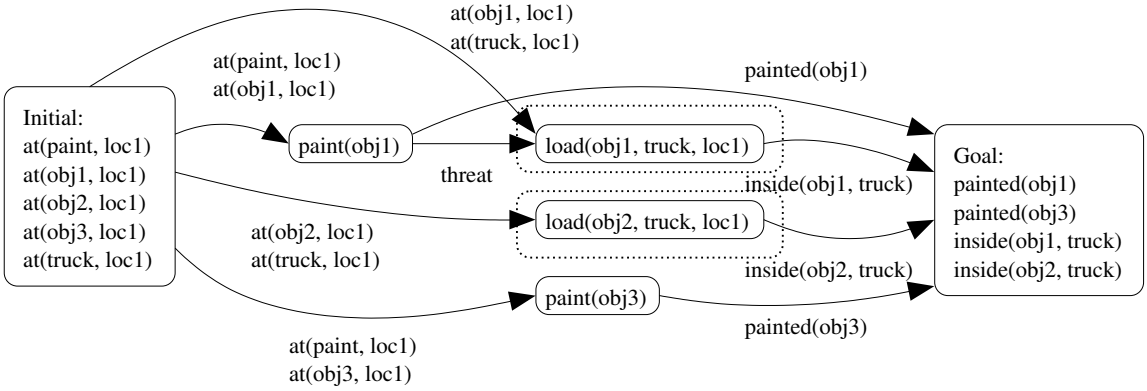


Figure 5.3: Two parallel matching subplans of length 1 are surrounded by dotted lines and represent an unrolled loop.

within the unrolled loop, and that the current state when the loop is executed contains all the conditions for the steps within the unrolled loop to execute correctly and support the goals of the plan. The loop represented by the two load operators in Figure 5.1 is shown in Figure 5.4.

A Parallel Loop is a loop in which each iteration of the loop is causally independent from the others. The loop shown in Figure 5.4 is a parallel loop. A loop may also have a

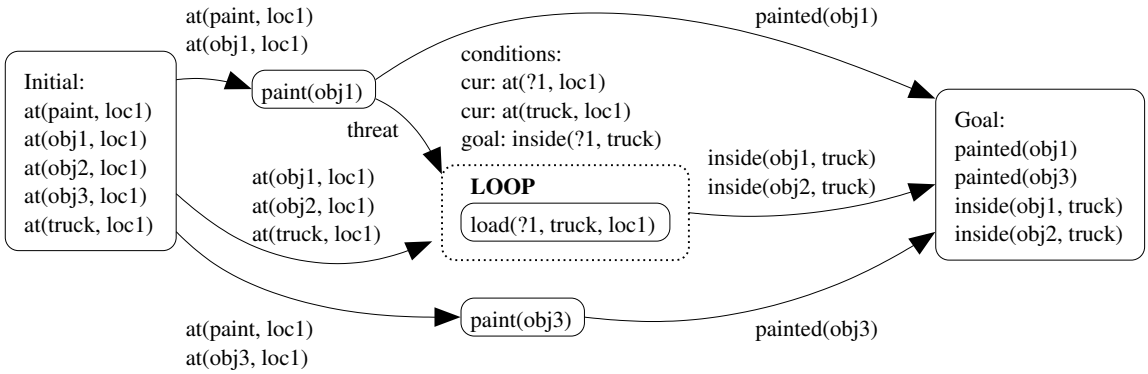


Figure 5.4: The painting and transport problem after the load loop is identified. The loop is surrounded by dotted lines. The loop variable is written as ?1, and ranges over all values that meet the conditions of the loop (in this case, obj1 and obj2). Conditions for the loop are shown above the loop.

multi-step body with complex causal structure; it may even include other loops.[1] LoopDISTILL identifies 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 the Rocket domain (Veloso 94a), one loop may describe the rocket individually picking up and dropping off packages, as shown in Figure 5.5. Each iteration of the loop consists of the rocket flying to the package, loading it, flying to its destination, and unloading it. These iterations must be executed in a specific order since the fly operations are causally linked. LoopDISTILL identifies non-nested serial loops in observed plans.

5.2 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. We discuss in Chapter 2 how to find minimal annotated consistent partial

[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—and that other non-loop causally independent steps could appear throughout the trace of the loop's execution.

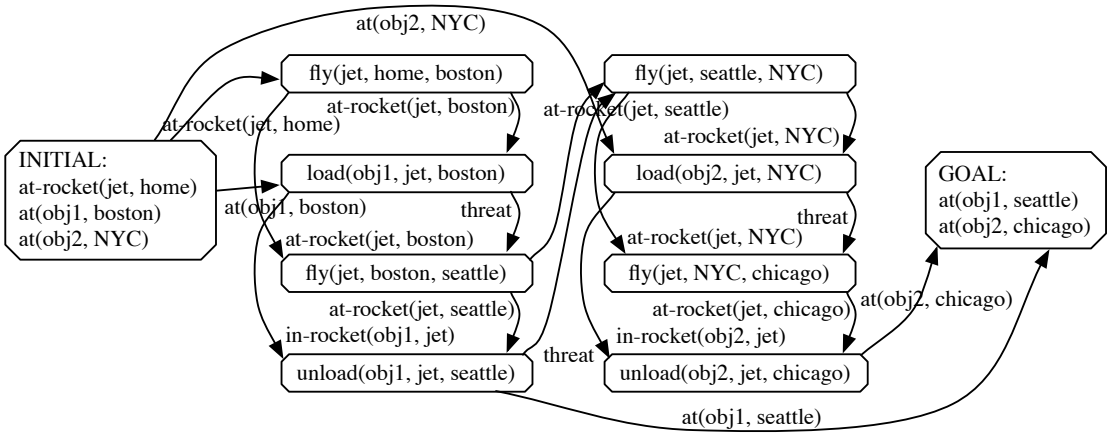


Figure 5.5: An example annotated partially ordered plan in the rocket domain that includes a serial loop.

orderings of totally-ordered plans given a model of the operators (Winner 02) and other work has shown that STRIPS-style operator models are learnable through examples and experimentation (Carbonell 90), so this assumption is not excessively restrictive.

The LoopDISTILL algorithm has two components, formalized in Algorithm 5 and Algorithm 7. The first extracts all non-nested identical parallel loops from the observed plan and the second extracts all non-nested identical serial loops. Both begin by identifying an unrolled loop (described in the Section "Identifying Unrolled 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.

5.2.1 Identifying Parallel Unrolled Loops

A parallel unrolled loop is a set of parallel matching subplans within the observed plan. The process of finding a parallel unrolled loop begins with the identification of a set of parallel matching steps, as described in Algorithm 5. Next, LoopDISTILL expands these matching steps into a set of parallel matching subplans. There may be many ways to do this, and it is impossible to know which will lead to the "best" (according to some measure) representation of the plan without trying to process the rest of the plan fully for each possible loop, so our current algorithm searches for the longest possible loops—the largest parallel matching subplan common to at least two of the steps.

Algorithm 5 The 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
```

The procedure LargestCommonSubplan, formalized in Algorithm 6, recursively tries every possible expansion of the existing subplans 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 thse steps as a possible way to expand the subplan. For each step in *StepBack* or *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.

5.2.2 Identifying Serial Unrolled Loops

The process of finding a serial unrolled loop—a set of serial matching subplans within the observed plan—begins, as shown in Algorithm 7, by stepping through the plan and, for each step, searching for a causally linked matching step. The steps need not be linked directly in the partial ordering, but do need to be ordered with respect to each other (linked directly in the transitive closure of the partial ordering).

Once a pair of causally linked matching steps is found, LoopDISTILL tries to expand the two matching steps into serial matching subplans in the procedure ConnectSerialLoop, shown in Algorithm 8, by matching all the steps that come causally after the first iteration

Algorithm 6 LargestCommonSubplan: Identify largest parallel matching subplans of an observed plan common to at least two of the given parallel matching subplans.

Input: set $\mathcal{A}$ of parallel matching subplans $S_1..S_m$, minimal annotated partially ordered plan $\mathcal{P}$.
Output: Set of largest parallel matching subplans of plan $\mathcal{P}$ common to at least two of $S_1..S_m$.

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

and before the second to steps that come causally after the second iteration. If the subplans cannot be connected, LoopDISTILL continues searching for causally linked steps that match the original step. If they can be connected, then LoopDISTILL has identified two iterations of a serial loop, and proceeds to search for further iterations in the procedure FindOtherSerialIterations, shown in Algorithm 9. This procedure searches through the steps that are directly causally linked from the last iteration of the loop to find a set of steps that match the first step of each iteration. Each of these steps is then explored as a possible first step of the next iteration as LoopDISTILL tries to match the rest of the steps in the iteration. If a new iteration is found, it is added to the loop and the search for new iterations continues.

The iterations of a serial loop *must* be fully causally connected—there may not be any

Algorithm 7 The LoopDISTILL algorithm for identifying 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
      goto next i
    end if
  end for
end for
```

steps not included in the loop that are causally linked from one iteration and to a later iteration. Otherwise, the loop would not be able to execute independently. Similarly, there may not be any steps not included in a particular iteration that are causally linked from and to steps in that iteration.

Matching Serial Loop Iteration Conditions

Matching the conditions for a set of steps for a serial loop is sometimes different than matching them with parallel loops in that some sets of matching iterations have different goal footprints (Veloso 94a). This happens because early iterations of a serial loop support later iterations, and therefore include the goals the later iterations support in their own footprints, so early iterations can have larger goal footprints than later iterations. Consider the example shown in Figure 5.6. The loop is a repetition of the steps resoap and wash for each dish, but the iteration for dish1 contains in its goal footprint that dish2 is clean because it is causally connected to the second iteration.

In some serial loops, early iterations can have identical goal footprints, while subsequent iterations' goal footprints are subsets of this. In the example shown in Figure 5.7, the move-block-table steps constitute a loop. However, the steps move-block-table(4, 3) and move-block-table(3, 2) have identical goal footprints: on(1, 2), on(2, 3), and on(3, 4). The last iteration of the loop—the step move-block-table(2, 1)—has a goal footprint that is a subset of these, though: on(1, 2) and on(2, 3). We handle this by using the smallest matching goal footprint as the basis of the learned loop. The unstacking loop

Algorithm 8 ConnectSerialLoop: Search for two serial matching subplans rooted at given two causally linked matching steps.

Input: i and j, causally linked matching steps; minimal annotated partially ordered plan $\mathcal{P}$.
Output: Set of two serial matching subplans rooted at i and j, or , if there are none.

```
Iteration1 ← i
Iteration2 ← j
BetweenSteps ← steps causally linked from Iteration1 and causally linked to Iteration2
for all s in BetweenSteps do
  if ∃s′ in P causally linked from Iteration2 and matching s then
    Iteration1 ← s
    Iteration2 ← s′
    BetweenSteps ← steps causally linked from Iteration1 and causally linked to Iteration2
  else
    return
  end if
end for
InsideSteps ← steps not in Iteration1 and causally linked from and to steps in Iteration1
for all s in InsideSteps do
  if ∃s′ in P not in Iteration2, causally linked from and to steps in Iteration2, and matching s then
    Iteration1 ← s
    Iteration2 ← s′
    InsideSteps ← steps not in Iteration1 and causally linked from and to steps in Iteration1
  else
    return
  end if
end for
if ∃s′ not in Iteration2 and causally linked from and to steps in Iteration2 then
  return
end if
return {Iteration1, Iteration2}
```

learned from the example in Figure 5.7 is shown in DsPlanner 7. In some domains, and for some problem classes, more iterations of a particular loop may be required in order to learn the most general version of the loop; otherwise the learned loop may be overly specific to the observed example.

Also, sometimes in serial loops, each iteration has the same goal footprint, as each supports the entire eventual goal. For example, in the Rocket domain, if the Rocket picks all packages up and then drops them all off, as shown in Figure 5.8, each iteration of the pickup loop supports *all* packages' arrival at their goal destinations. But if the entire goal footprint were then included in the dsPlanner, as it would be for a parallel loop, the loop would be overly specific to the given example, as shown in DsPlanner 8. To find the more general loop we'd like, when we encounter goal footprints that consist of

Algorithm 9 FindOtherSerialIterations: Search for additional iterations of the serial loop represented by the given serial matching subplans.

Input: $\mathcal{C}$ set of m serial matching subplans $S_1..S_m$, minimal annotated partially ordered plan $\mathcal{P}$.
Output: Largest set of serial matching subplans that includes $\mathcal{C}$

```
while We keep finding loops do
  NextSteps ← steps causally linked from last subplan S_n
  for all steps i in NextSteps do
    if i matches the first steps of S_1..S_n then
      C ← ConnectNewIteration(C, i, 1, P)
      NextSteps ← steps causally linked from new last subplan S_{n+1}
      break out of for statement
    end if
  end for
end while
```

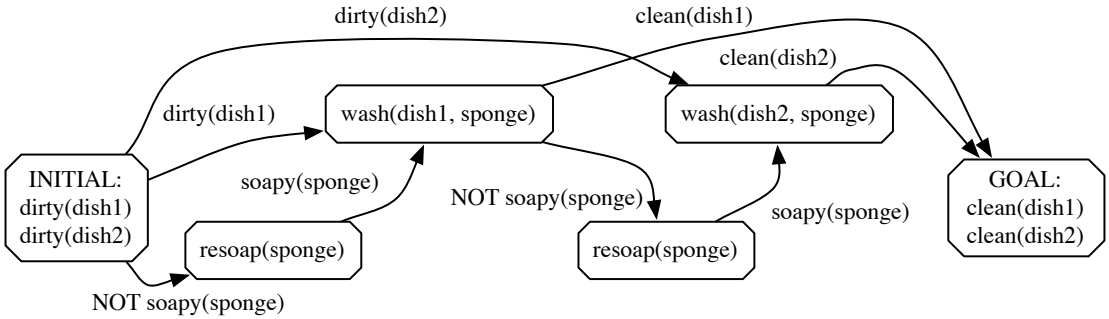


Figure 5.6: An example problem in a dishwashing domain demonstrating a serial loop in which the first iteration has a goal footprint that is a superset of that the second iteration.

DsPlanner 7 The unstacking dsPlanner loop learned by LoopDISTILL from the Blocksworld example shown in Figure 5.7.

```
while inCurState (clear(?v1:block)) and inCurState (on(?v1:block ?v2:block)) and inGoalState
(on(?v2:block ?v1:block)) do
  move-block-table(?v1 ?v2)
end while
```

repeated matching conditions in a serial loop, we trim the goal footprint to be reflected in the conditions of the dsPlanner to only those matching conditions relevant to each iteration, leading to a loop that is more general, as shown in DsPlanner 9.

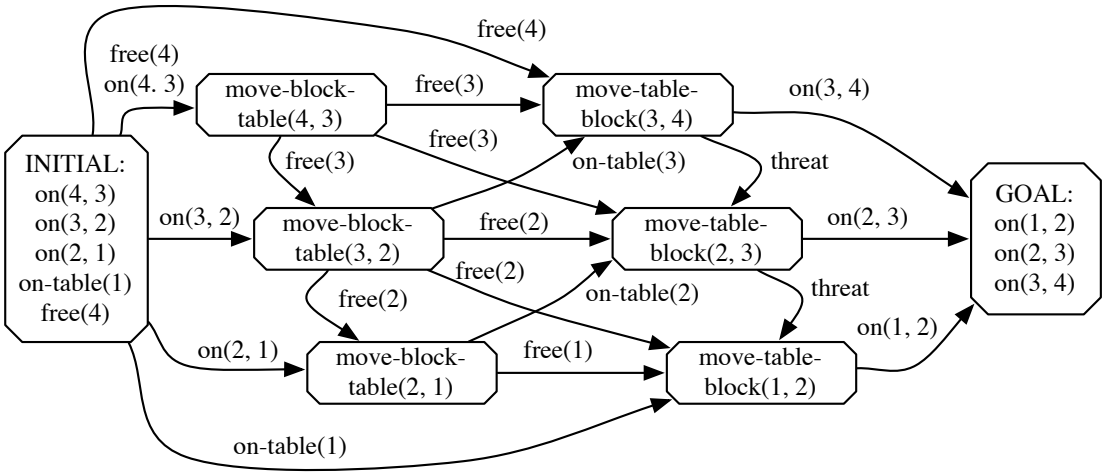


Figure 5.7: An example problem in the Blocksworld domain demonstrating a serial loop in which the first two iterations have identical goal footprints but the third iteration has a goal footprint that is a subset of these.

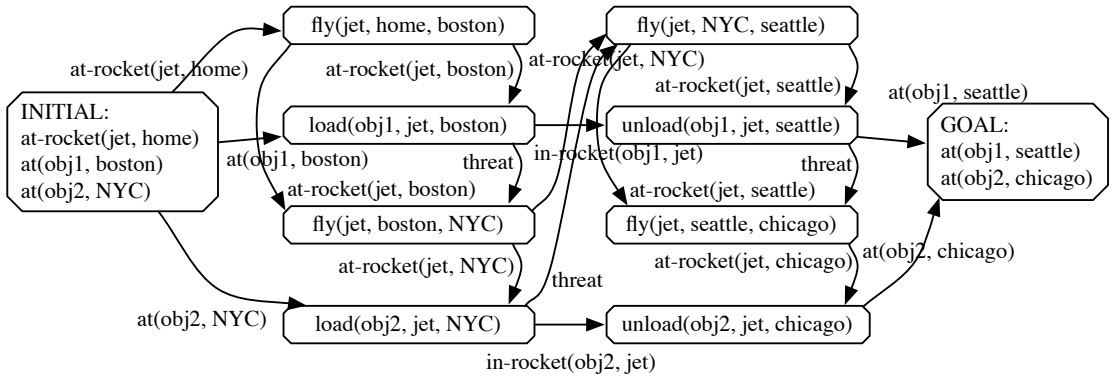


Figure 5.8: An example problem in the Rocket domain demonstrating a serial loop (the "pick up" loop) in which all iterations have identical goal footprints—the entire goal.

5.2.3 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. The body of the loop is the subplan—with a

DsPlanner 8 The overly specific dsPlanner pickup loop that could be learned from the Rocket domain example shown in Figure 5.8 if we don't account for adjusting the size of the goal footprint to those conditions that are relevant to each iteration.

while inCurState (at-rocket(?v1:rocket ?v2:loc)) **and inCurState** (at(?v3:obj ?v4:loc)) **and inGoalState** (at(?v3:obj ?v5:loc)) **and inGoalState** (at(?v6:obj ?v7:loc)) **do**
  fly(?v1 ?v2 ?v4)
  load(?v3 ?v1 ?v4)
end while

DsPlanner 9 The dsPlanner pickup loop learned by LoopDISTILL from the Rocket domain example shown in Figure 5.8. LoopDISTILL adjusts which elements of the goal footprint are represented in the conditions for the loop to those that are relevant to each iteration.

while inCurState (at-rocket(?v1:rocket ?v2:loc)) **and inCurState** (at(?v3:obj ?v4:loc)) **and inGoalState** (at(?v3:obj ?v5:loc)) **do**
  fly(?v1 ?v2 ?v4)
  load(?v3 ?v1 ?v4)
end while

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.

Algorithm 10 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

5.3 Illustrative Examples

We now illustrate the operation of the LoopDISTILL algorithm on simple example plans. First we show the discovery of a parallel loop, and then the discovery of a serial loop.

5.3.1 A Parallel Multi-Step Loop Example

We now illustrate the operation of the LoopDISTILL algorithm on a simple example plan from an artificial domain, illustrated in Figure 5.9. 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.

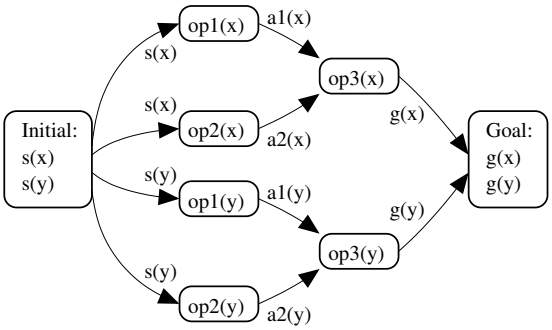


Figure 5.9: An example annotated partially ordered plan in an artificial domain that includes a multi-step loop consisting of the steps op1, op2, and op3. The original totally ordered plan could have been any topological sort of this partial ordering.

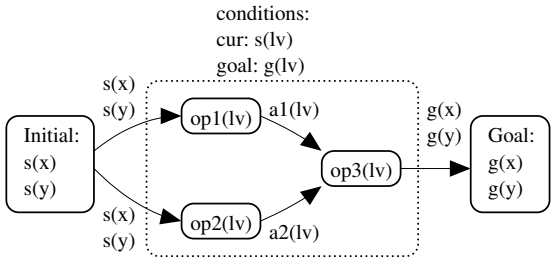


Figure 5.10: The example plan shown in Figure 5.9 after the loop has been identified. The loop is surrounded by dotted lines. The loop variable is written as lv, and ranges over all values that meet the conditions of the loop (in this case, x and y). The conditions of the loop are shown above it.

LargestCommonSubplan begins by finding the steps causally linked directly to and from the set of matching subplans. There is one such step for each track: op3(x) and op3(y), respectively. 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 is one step causally linked directly to or from each of the two parallel subplans: op2(x) and op2(y), on which op3(x) and op3(y) depend. Adding these steps also pre-

serves 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 more steps linked to or from the parallel subplans, 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 5.10.

The learned dsPlanner is shown in DsPlanner 10. We designed this domain to demonstrate that LoopDISTILL can capture loops with complex causal structure. The problems we used to test 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 5.11 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. [2] The *learned* dsPlanner is able to solve problems with as many as 40,000 objects in under a minute.

DsPlanner 10 The looping dsPlanner learned by LoopDISTILL from the multi-step loop domain problem shown in Figures 5.9 and 5.10.

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

5.3.2 A Rocket Domain Example with Parallel Loops

We now describe the operation of the LoopDISTILL algorithm on a simple example plan from the rocket domain, illustrated in Figure 5.12. First, LoopDISTILL searches for sets of parallel matching steps. It finds the steps load(o1, r, s), load(o2, r, s), and load(o3, r, s), which differ only in one variable, which ranges over the values o1, o2, and o3.[3] These three one-step parallel matching subplans are then sent to LargestCommonSubplan, which searches for a larger subplan common to at least two of them.

[2] It takes less than a second to learn the dsPlanner for the multi-step loop domain example we used.

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

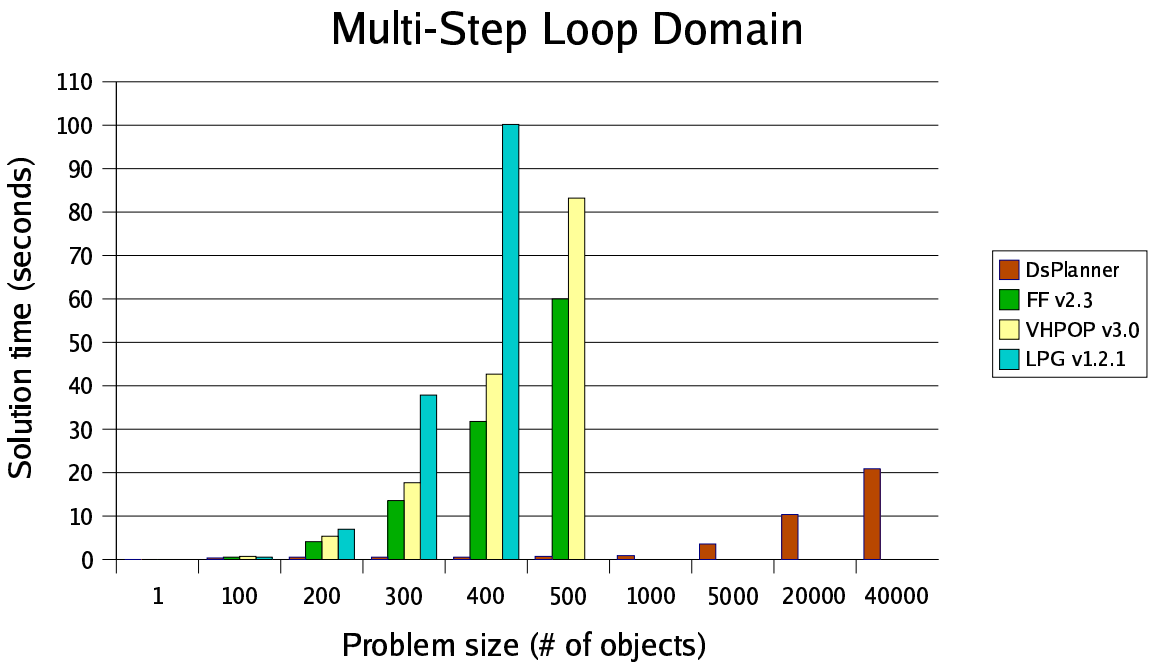


Figure 5.11: Timing results of several general-purpose planners and of the learned ds-Planner shown in DsPlanner 10 on large-scale multi-step loop domain problems. We also tested the MIPS planner, but it wasn't able to solve enough large-scale problems to appear on the graph.

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

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

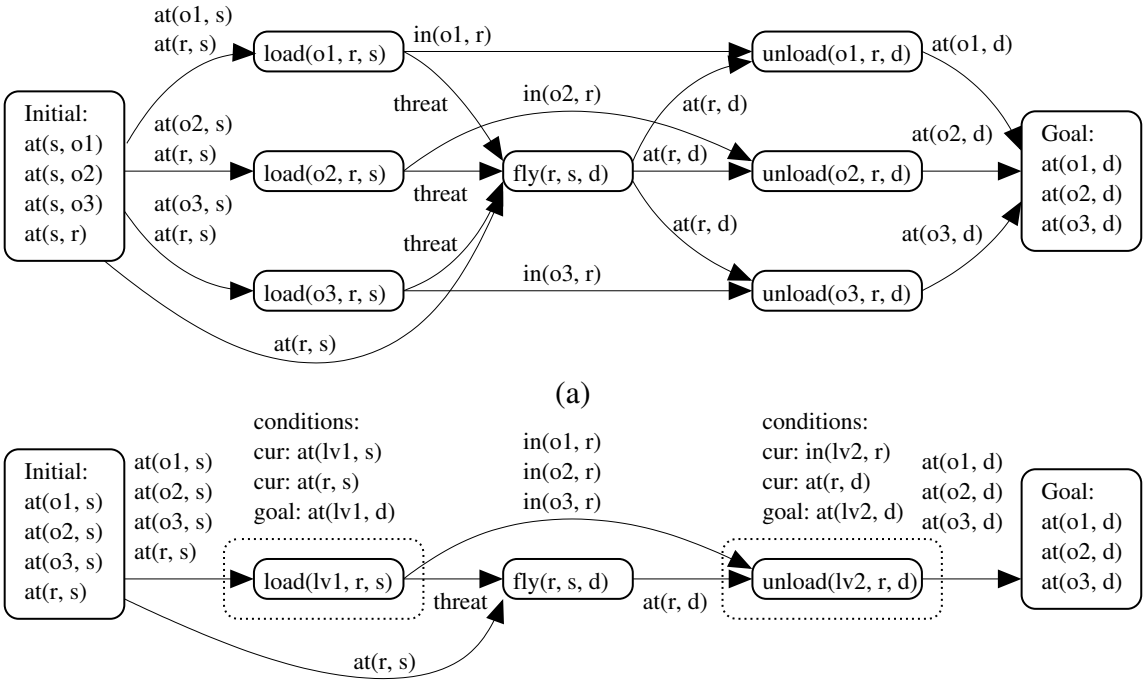


Figure 5.12: An example annotated partially ordered plan in the rocket domain that involves moving objects o1, o2, and o3 from location s to location d using rocket r. The minimal annotated partial ordering of the plan is shown in (a). The plan after loops are identified is shown in (b). Loops are surrounded by dotted lines. The loop variables are written as lv1 and lv2, and range over all values that meet the conditions of the loops (in these cases, o1, o2, and o3). Conditions for the loops are shown above them.

This process repeats to uncover the unload loop, and the resulting plan is shown in Figure 5.12(b).

The dsPlanner learned from the rocket-domain example shown in Figure 5.12 is shown in DsPlanner 11. Figure 5.13 shows the results of executing several different general-purpose planners and the learned dsPlanner on large-scale Rocket-domain problems. 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). Run

times do not include the time required to learn the dsPlanner, though this is negligible at under a second. The learned dsPlanner is able to solve problems with more than 60,000 objects in under a minute.

DsPlanner 11 dsPlanner based on the rocket domain problem shown in Figure 5.12. The variable in each loop is indicated by a "v" preceeding its name.

```
while inCurState (at(?v1:object, ?2:location)) and inCurState (at(?3:rocket,
?2:location)) and inGoalState (at(?v1:object, ?4:location)) do
  load(?v1:object ?3:rocket ?2:location)
end while
if inCurState (at(?1:rocket ?2:location)) and inCurState (in(?3:object ?1:rocket))
and inGoalState (at(?3:object ?4:location)) then
  fly(?1:rocket ?2:location ?4:location)
end if
while inCurState (in(?v1:object, ?2:rocket)) and inCurState (at(?2:rocket ?3:lo-
cation)) and inGoalState (at(?v1:object, ?3:location)) do
  unload(?v1:object ?2:rocket ?3:location)
end while
```

5.3.3 A Rocket Domain Example with Serial Loops

We now look at an example from the Rocket domain, shown in Figure 5.14, which contains a serial loop. First LoopDISTILL searches for a pair of causally linked matching steps in the plan. It finds the steps fly(home, boston) and fly(boston, seattle). The procedure ConnectSerialLoop is then called to try to make a connected serial loop from the two steps. It finds the set of steps that come causally after the first subplan and before the second. There is one such step: load(obj1, boston). It then searches for a step causally linked from the second subplan that matches. No such step exists, so this set of matching subplans is abandoned—it cannot be fully connected into a serial loop.

Next, LoopDISTILL searches for another set of causally linked matching steps and finds the pair fly(home, boston) and fly(seattle, new york), as shown in Figure 5.15. The procedure ConnectSerialLoop finds the set of steps that come causally after the first subplan and before the second: load(obj1, boston), fly(boston, seattle), and unload(obj1, seattle). ConnectSerialLoop then finds a match for each of these steps causally linked from the second subplan: load(obj2, new york), fly(new york, chicago), and unload-(obj2, chicago). The two serial subplans are now connected and represent two iterations of a serial loop.

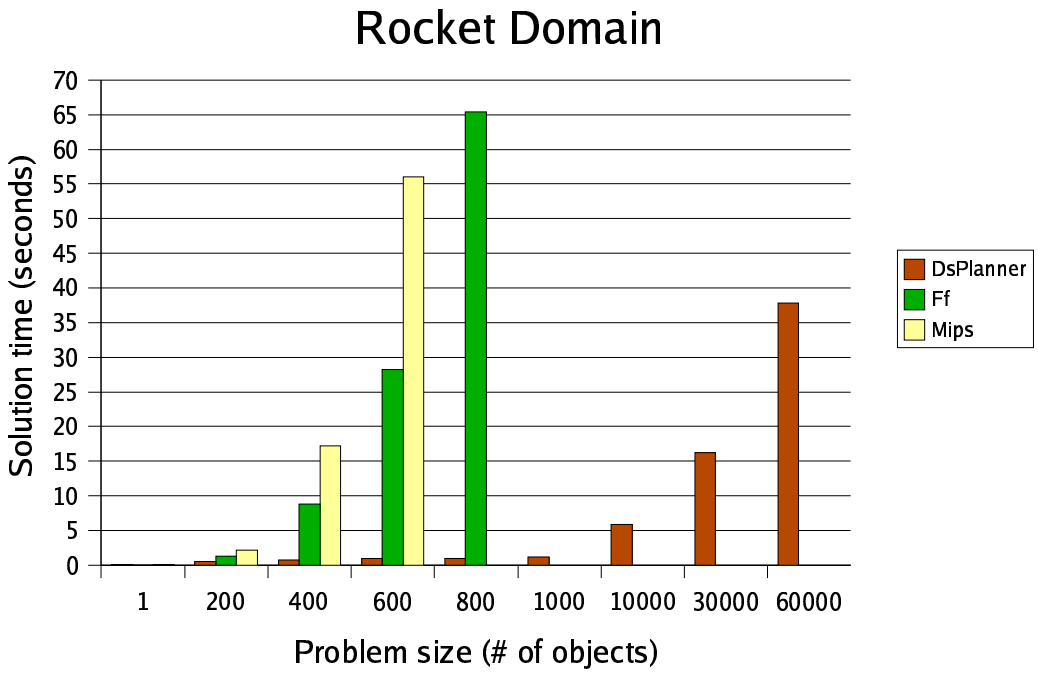


Figure 5.13: Timing results of several general-purpose planners and of the learned dsPlanner shown in DsPlanner 12 on large-scale rocket-domain delivery problems. We also tested VHPOP and LPG but they were not able to solve enough large-scale problems to be appear on this graph.

The procedure FindOtherSerialIterations is called to try to find further iterations of the loop, but, as there are no more steps in the plan, this does not succeed. A new loop is then created to capture the common four-step subplan. The loop body is assigned to the common subplan, and a set of four new loop variables replaces the differing values. The conditions of the loop are that the current state satisfies the conditions of the steps within it, and that the goal state contains the goals supported by the steps in the loop body. The resulting plan is shown in Figure 5.16.

The dsPlanner learned from this Rocket-domain problem is shown in DsPlanner 12. This planner, learned from the single example shown in Figure 5.14, can solve all Rocket-domain problems with goals requiring that objects be at particular destination locations. It does not do so optimally, but models the problem solving process demonstrated in the example from which it was learned, and it enables the solving of a large set of scaled-up problems in the domain.

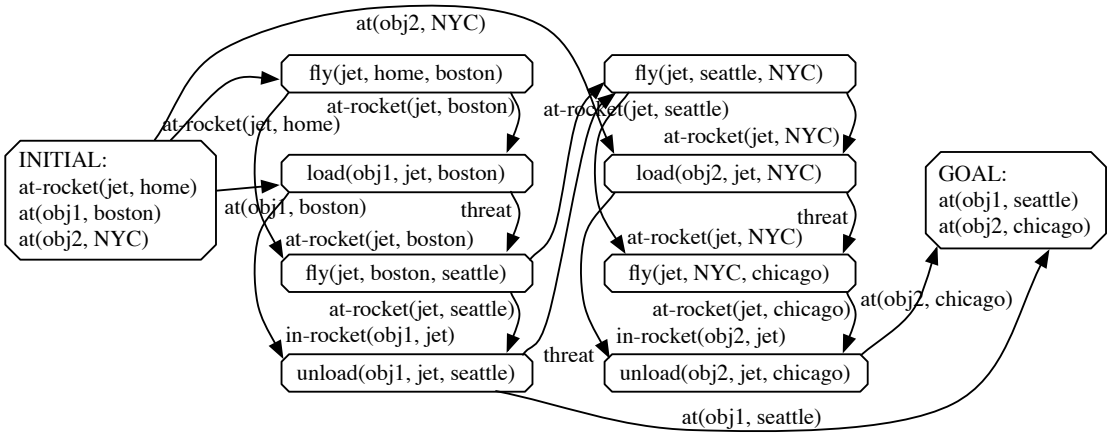


Figure 5.14: An example annotated partially ordered plan in the rocket domain that includes a serial loop.

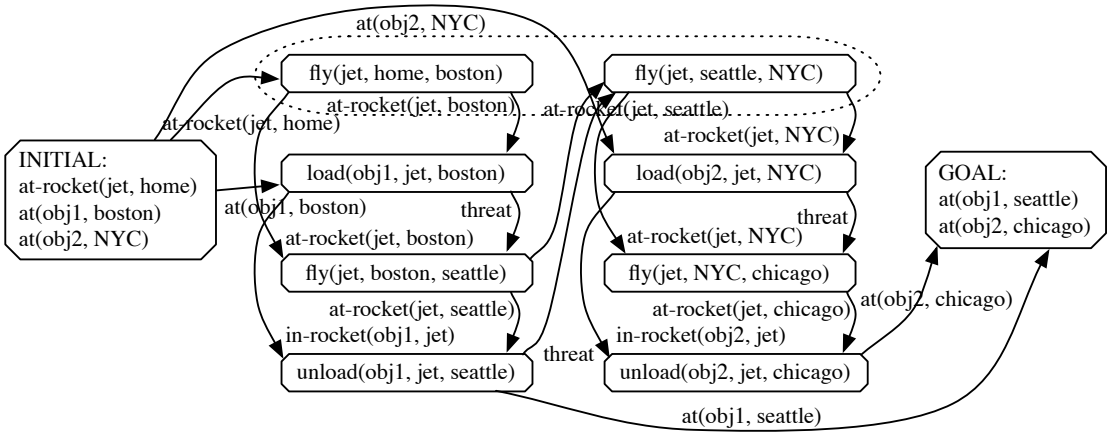


Figure 5.15: The example plan shown in Figure 5.14 with two causally linked matching steps identified.

5.4 Summary

In this chapter, we contribute the LoopDISTILL algorithm for automatically identifying repeated structures in observed plans, determining the body and conditions of the loops

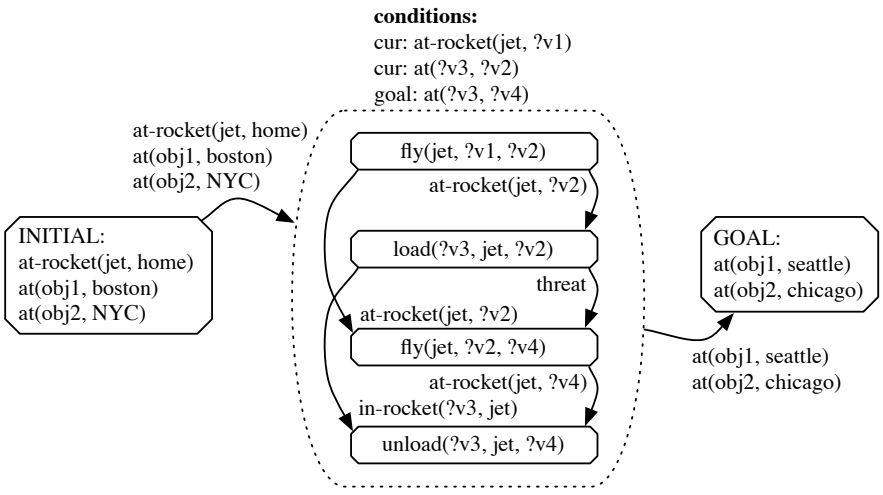


Figure 5.16: The example plan shown in Figure 5.14 after the loop has been identified. The loop variables are shown with question marks in front of them. All parameters of the steps in the loop are variables.

DsPlanner 12 The looping dsPlanner learned by LoopDISTILL from on the rocket-domain example shown in Figures 5.14 and 5.16.

---

while inCurState (at(?v1:object ?v2:loc)) **and inCurState** (at-rocket(?v3:loc)) **and inGoalState** (at(?v1:object ?v4:loc)) **do**
  fly(?v3 ?v2)
  load(?v1 ?v2)
  fly(?v2 ?v4)
  unload(?v1 ?v4)
end while

---

they represent, and converting looping plans into learned looping domain-specific planning programs (dsPlanners). The LoopDISTILL algorithm identifies all identical non-nested loops in an example plan by identifying sets of matching subplans and then converting each set into a loop. Our results show that LoopDISTILL can learn looping dsPlanners with broad coverage very quickly from a single example.

Chapter 6

Learned DsPlanners for Several Domains

The LoopDISTILL algorithm is able to find all identical non-nested loops in an observed plan and to convert that plan into a dsPlanner. We show that looping dsPlanners learned from a single plan compactly capture the structure of the example and can apply this knowledge very efficiently to solving much larger problems. In this work, we do not address the problem of learning looping dsPlanners from multiple examples or of merging looping dsPlanners. A learned dsPlanner is not guaranteed to be complete, since it is learned from an example which may not cover the entire domain.

In this chapter, we discuss the classes of problems that LoopDISTILL can and cannot create dsPlanners to solve by presenting example plans and learned dsPlanners in a wide variety of planning domains. Example plans are observed as total orderings, and are then analyzed by SPRAWL, which reveals the rationale underlying the plan, in the form of a minimal annotated consistent partial ordering. This is the basis for the learning algorithms in LoopDISTILL, and we illustrate the example plans in this chapter in the minimal annotated consistent partial ordering format where space permits. Longer plans we present as total orderings.

6.1 Multi-Step Parallel Loop Domain

The dsPlanner learned from an artificial multi-step parallel loop domain example, shown in Figures 5.9 and 5.10, is shown in DsPlanner 13. We designed this domain to demonstrate that LoopDISTILL can capture parallel loops with complex causal structure. Loop-

DISTILL can learn a dsPlanner that covers the domain from a single example with two iterations of the loop, and the solutions found by the dsPlanner are optimal.

DsPlanner 13 The looping dsPlanner learned by LoopDISTILL from the multi-step parallel loop domain problem shown in Figures 5.9 and 5.10.

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

6.2 Multi-Step Serial Loop Domain

The dsPlanner learned from an artificial multi-step serial loop domain example, shown in Figure 6.1, is shown in DsPlanner 14. We designed this domain to demonstrate that LoopDISTILL can capture serial loops with complex causal structure. LoopDISTILL can learn a dsPlanner that covers this domain from a single example with two iterations of the loop; the solutions found by this dsPlanner are optimal.

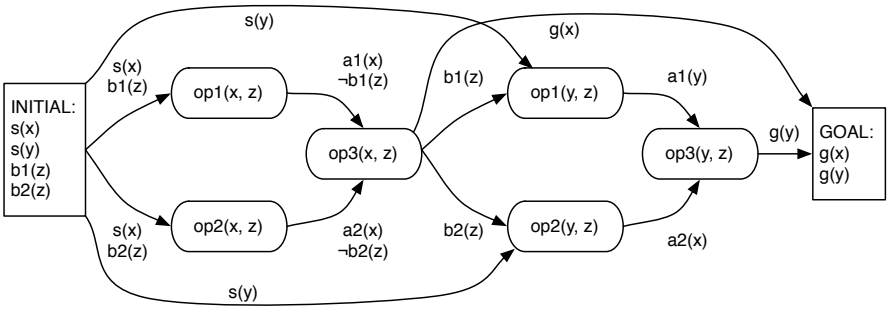


Figure 6.1: An example plan in an artificial domain demonstrating a serial loop with complex causal structure.

DsPlanner 14 The looping dsPlanner learned by LoopDISTILL from the multi-step serial loop domain problem shown in Figures 6.1.

while inCurState (s(?v1) **and inCurState** (b1(z)) **and inCurState** (b2(z)) **and inGoalState** (g(?v1)))
do
  op1(?1, z)
  op2(?1, z)
  op3(?1, z)
end while

6.3 Blocksworld

The Blocksworld domain (Winograd 72) allows for many problem classes that would require solution algorithms to use recursive definitions (such as which blocks are "above" or "below" others in a stack) that are not part of the description of the classical Blocksworld domain. LoopDISTILL cannot generate such recursive definitions on its own at present. However, there are many classes of problems within this domain that LoopDISTILL can ably attack.

One simple problem class is "unstack all" problems. One example problem of this type is shown in Figure 6.2. The dsPlanner learned by LoopDISTILL from this example is shown in DsPlanner 15. The solutions generated by this dsPlanner are optimal.

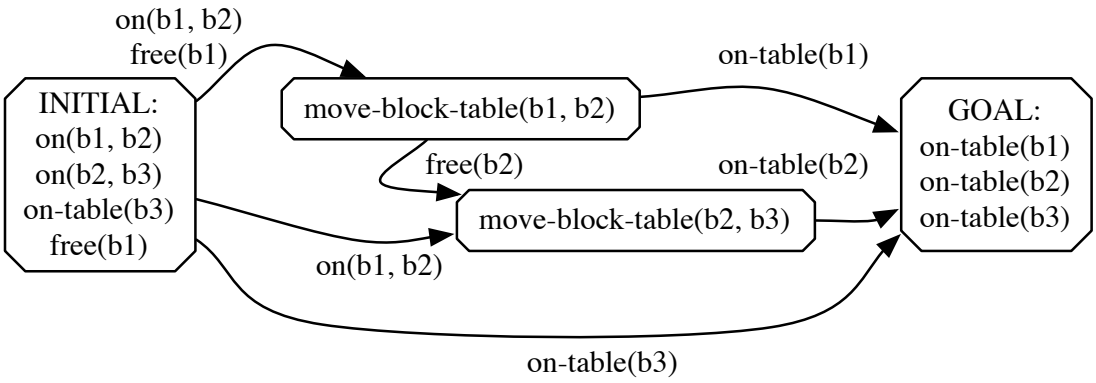


Figure 6.2: An example annotated partially ordered plan in the Blocksworld domain in which the goal is to unstack all blocks.

LoopDISTILL can also learn a dsPlanner that builds a specific tower. It cannot capture

DsPlanner 15 The looping dsPlanner learned by LoopDISTILL from the blocksworld-domain example shown in Figure 6.2. This dsPlanner solves unstacking problems with a serial loop that repeatedly unstacks the top block of the tower.

while inCurState (on(?v1:block ?v2:block)) **and inCurState** (free(?v1:block)) **and inGoalState** (on-table(?v1:block)) **do**
  move-block-table(?v2 ?v1)
end while

the ordering of the serial loop iterations (expressed in runtime as state matching) because the ordering is based on threat links, which are not mined for information in the current LoopDISTILL algorithm. However, it does capture the underlying process. A plan that demonstrates this algorithm is shown in Figure 6.3, and the dsPlanner learned by LoopDISTILL from this plan is shown in DsPlanner 16. Though the dsPlanner does not capture the ordering of the iterations, it is optimal.

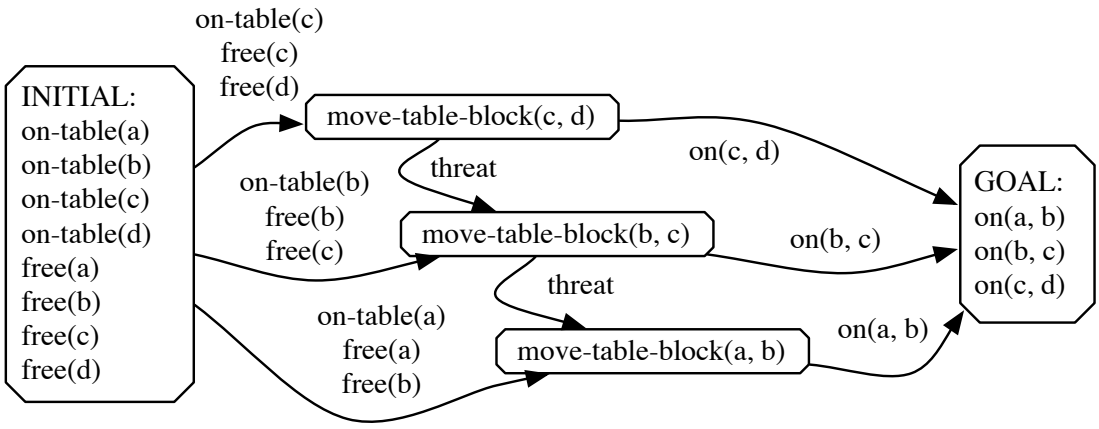


Figure 6.3: An example annotated partially ordered plan in the Blocksworld domain in which the goal is to build a tower.

LoopDISTILL can learn a dsPlanner composed of those two subtasks: unstack the blocks, and then build a specific tower. An example plan of this form is shown in Figure 6.4, and the learned dsPlanner is shown in DsPlanner 17. This dsPlanner is not optimal, as it moves blocks to the table, even if it would be more efficient to move them directly to their destination block. However, it does not unstack blocks that are stacked correctly, so in the worst case it produces a plan with twice the number of steps as the shortest plan—

DsPlanner 16 The looping dsPlanner learned by LoopDISTILL from the blocksworld-domain example shown in Figure 6.3. This dsPlanner solves stacking problems, but does not capture the ordering in which the stacking must be done.

while inCurState (on-table(?v1:block)) **and inCurState** (free(?v2:block)) **and inCurState** (free(?v1:block)) **and inGoalState** (on(?v1:block ?v2:block)) **do**
  move-table-block(?v1 ?v2)
end while

one step for moving the block to the table and one for moving it to its destination block, rather than simply one for moving it to its destination block.

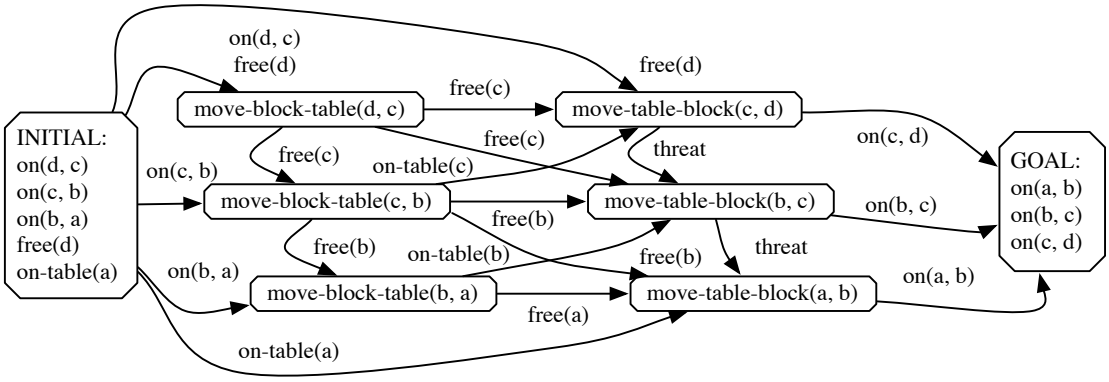


Figure 6.4: An example annotated partially ordered plan in the Blocksworld domain in which the goal is to unstack the blocks and build a new tower.

DsPlanner 17 The looping dsPlanner learned by LoopDISTILL from the blocksworld-domain example shown in Figure 6.4. This dsPlanner unstacks blocks not in their goal orders and solves stacking problems, but does not capture the ordering in which the stacking must be done.

while inCurState (on(?v1:block ?v2:block)) **and inCurState** (free(?v1:block)) **and inGoalState** (on(?v1:block ?v3:block)) **do**
  move-block-table(?v1 ?v2)
end while
while inCurState (on-table(?v1:block)) **and inCurState** (free(?v2:block)) **and inGoalState** (on(?v1:block ?v2:block)) **do**
  move-table-block(?v1 ?v2)
end while

Because dsPlanners cannot capture recursive definitions or the ordering of iterations, LoopDISTILL would be unable to learn a dsPlanner that could achieve, for example, a block unstacking goal (for example "Make block A free") if there were a series of blocks not relevant to the goal on top of the block that was needed; there is no way of capturing in the current dsPlanner that those blocks should be moved, since there is no way of referring to objects that are not in some direct relationship to objects relevant to the goals. Capturing recursive definitions would require the discovery and formalization of new predicates such as "above" in the blocksworld case. I believe that the rationales found by SPRAWL are rich enough to be mined for this sort of information, and discuss it more in Section 8.3.3.

LoopDISTILL cannot learn a dsPlanner that covers the entire Blocksworld domain, but we have shown that it can learn dsPlanners that cover large classes of problems within that domain.

6.4 Schedule

The Schedule domain (Hoffman 03) involves scheduling several machines to do work on parts. The actions schedule a machine to work on a particular part. The doTimeStep operator executes all scheduled operations and frees all machines and parts to be scheduled again. The operations need to be done in a particular order, as some of them undo the work of others (latheing a part removes its paint and makes its surface rough, for example). This domain also permits parallelism by allowing several operations to be scheduled simultaneously (as long as they're on different machines and involve different parts) before executing a time step.

LoopDISTILL has no trouble capturing the appropriate ordering for the operations, as this is demonstrated by example. An example plan in this domain is shown in Figure 6.5, and the learned dsPlanner is shown in DsPlanner 18. The learned dsPlanner covers all problems in this domain, is learned from a single example, and, while not optimal, contains at most twice as many steps as the shortest plan, since doTimeStep is repeated after each operation in the learned dsPlanner and may not be needed that often. Because LoopDISTILL does not match non-identical iterations, the learned dsPlanner does not attempt to schedule as many operations in parallel as possible; the solutions it finds contain at most 8 times as many time steps as the optimal solution, since 8 is the largest number of operations that can occur simultaneously in this domain.

DsPlanner 18 The looping Schedule domain dsPlanner learned by LoopDISTILL from the example shown in Figure 6.5.

while not inCurState (scheduled(?v1:part)) **and not inCurState** (busy(Lathe)) **and not inCurState** (isCyllindrical(?v1:part)) **and inGoalState** (isCyllindrical(?v1:part)) **do**
  doLathe(?v1:part)
  doTimeStep
end while
while not inCurState (scheduled(?v2:part)) **and not inCurState** (busy(drillPress)) **and inCurState** (hasBit(drillPress ?v3:width)) **and inCurState** (canOrient(drillPress ?v4:orient)) **and not inCurState** (hasHole(?v2 ?v3 ?v4)) **and inGoalState** (hasHole(?v2 ?v3 ?v4)) **do**
  doDrillPress(?v2 ?v3 ?v4)
  doTimeStep
end while
while not inCurState (scheduled(?v5:part)) **and not inCurState** (busy(punch)) **and inCurState** (hasBit(punch ?v6:width)) **and inCurState** (canOrient(punch ?v7:orient)) **and inCurState** (temperature(?v5:part cold)) **and not inCurState** (hasHole(?v5 ?v6 ?v7)) **and inGoalState** (hasHole(?v5 ?v6 ?v7)) **do**
  doPunch(?v5 ?v6 ?v7)
  doTimeStep
end while
while not inCurState (scheduled(?v8:part)) **and not inCurState** (busy(grinder)) **and not inCurState** (surfaceCondition(?v8:part smooth)) **and inGoalState** (surfaceCondition(?v8:part smooth)) **do**
  doGrind(?v8)
  doTimeStep
end while
while not inCurState (scheduled(?v9:part)) **and not inCurState** (busy(sprayPainter)) **and inCurState** (hasPaint(sprayPainter ?v10:paint)) **and inCurState** (painted(?v9:part ?v11:paint)) **and inGoalState** (painted(?v9:part ?v10:paint)) **do**
  doSprayPaint(?v9 ?v10)
  doTimeStep
end while
while not inCurState (scheduled(?v12:part)) **and not inCurState** (busy(immersionPainter)) **and inCurState** (hasPaint(immersionPainter ?v13:paint)) **and inCurState** (painted(?v12:part ?v14:paint)) **and inGoalState** (painted(?v12:part ?v13:paint)) **do**
  doImmersionPaint(?v12 ?v13)
  doTimeStep
end while
while not inCurState (scheduled(?v15:part)) **and not inCurState** (busy(polisher)) **and inCurState** (temperature(?v15:part cold)) **and not inCurState** (surfaceCondition(?v15:part polished)) **and inGoalState** (surfaceCondition(?v15:part polished)) **do**
  doPolish(?v15)
  doTimeStep
end while

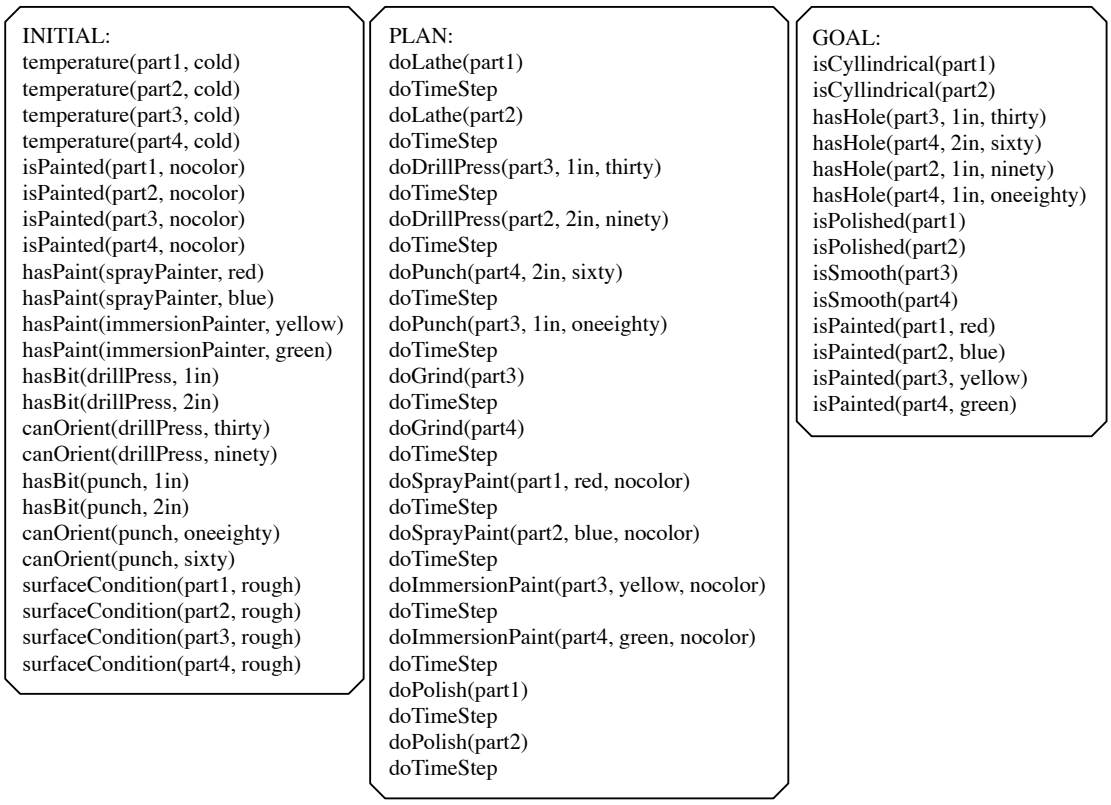


Figure 6.5: An example plan in the Schedule domain.

LoopDISTILL would also be able to capture the possibility of parallelism in this domain if it were able to match non-identical iterations as a loop. This would allow LoopDISTILL to learn a dsPlanner that schedules as many operations at once as possible and then executes a time step from an example demonstrating such parallelism.

6.5 Rocket

The dsPlanner learned from a rocket-domain (Veloso 94a) example is shown in DsPlanner 19. This planner, learned from the single example shown in Figure 6.6, can solve all rocket-domain problems with goals requiring that objects be at particular destination locations. It does not do so optimally, but models the problem solving process

demonstrated in the example from which it was learned. The solution plans are at most $4*nummisplacedobjs$ in length, since, for each object, four steps are executed: the rocket is flown to the object, the object is loaded into the rocket, the rocket is flown to the destination location of the object, and the object is unloaded. The shortest possible solution plans are $2 * nummisplacedobjs$ in length, as in the best case they are all in the same original locaiton and all need to go to the same destination location, and in this case, the rocket must be flown to them, they must all be loaded, the rocket must be flown to the goal destination, and they must all be unloaded. So the solutions found by the learned dsPlanner are at worst twice as long as the shortest possible solutions.

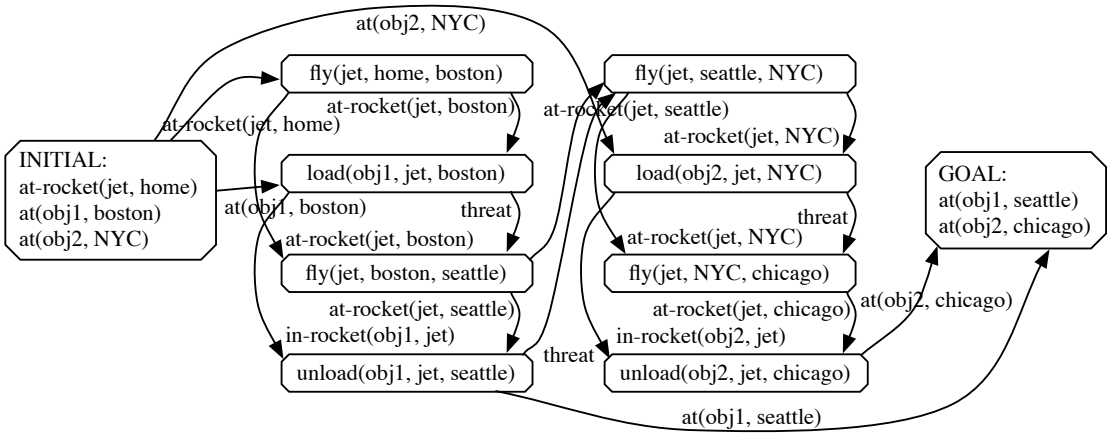


Figure 6.6: An example annotated partially ordered plan in the rocket domain that includes a serial loop.

DsPlanner 19 The looping dsPlanner learned by LoopDISTILL from the rocket-domain example shown in Figures 5.14 and 5.16. This dsPlanner uses one rocket and attends to each package separately, flying to, picking up, and dropping off each package in turn.

---

while inCurState (at(?v1:object ?v2:loc)) **and inCurState** (at-rocket(?3:rocket ?v4:loc)) **and inGoalState** (at(?v1:object ?v5:loc)) **do**
  fly(?3 ?v4 ?v2)
  load(?v1 ?3 ?v2)
  fly(?3 ?v2 ?v5)
  unload(?v1 ?3 ?v5)
end while

Instead of solving rocket-domain problems with only one rocket, we could use differ-

ent rockets for each package, as shown in the example plan in Figure 6.7. LoopDISTILL captures this difference in the dsPlanner it learns from the example, shown in DsPlanner 20. Note that in the dsPlanner learned from the rocket example in which only one rocket was used (DsPlanner 12), the rocket parameter of the loop is *not* a variable. This captures that the same rocket is used in each iteration of the loop. In the dsPlanner learned from this new example, in which a different rocket is used each time, the rocket parameter of the loop is a variable—it may be assigned to a different rocket each time. The solution plan lengths are the same for this algorithm and the previous, but this algorithm can produce plans with steps that can execute in parallel. In the worst case, this algorithm, like the previous one, produces solution plans that require $4 * nummisplacedobjs$ time steps. In the best case (using all rockets to achieve maximal parallelism), this algorithm produces solution plans that require $4 * (nummisplacedobjs/numrockets)$ time steps to execute. In the best case, an optimal plan with multiple rockets would require $2 * (nummisplacedobjs/numrockets)$ time steps to execute.

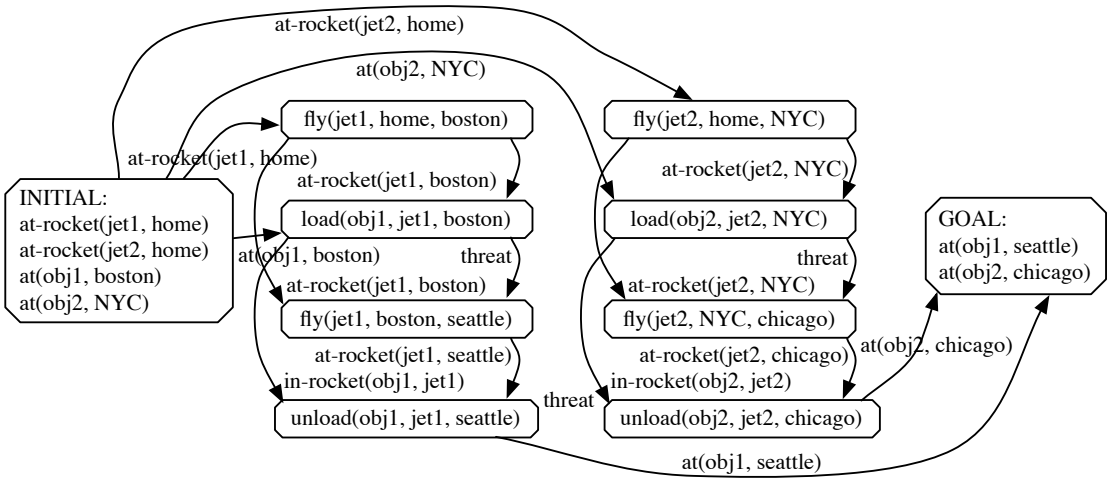


Figure 6.7: An example annotated partially ordered plan in the rocket domain in which multiple rockets are used to deliver the packages.

The same class of problems can also be solved using a different algorithm altogether. Instead of picking up and dropping off each item separately, all the pickups could be done first and then all the drop offs done afterwards. This algorithm is demonstrated in the example plan shown in Figure 6.8. LoopDISTILL captures this algorithm as well. The learned dsPlanner is shown in DsPlanner 21. This planner generates solution plans of length $4 * nummisplacedobjs$, so at worst twice the length of the shortest possible plans.

DsPlanner 20 The looping dsPlanner learned by LoopDISTILL from the rocket-domain example shown in Figures 6.7. This dsPlanner also solves the rocket problem one package at a time—flying to, picking up, and dropping off each in turn—but allows the use of multiple rockets.

while inCurState (at(?v1:object ?v2:loc)) **and inCurState** (at(?v3:rocket ?v4:loc)) **and inGoalState** (at(?v1:object ?v5:loc)) **do**
  fly(?v3 ?v4 ?v2)
  load(?v1 ?v3 ?v2)
  fly(?v3 ?v2 ?v5)
  unload(?v1 ?v3 ?v5)
end while

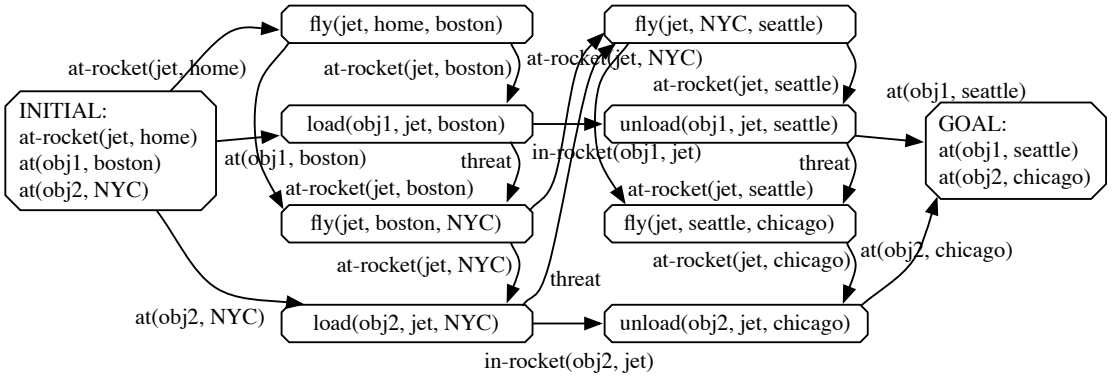


Figure 6.8: An example annotated partially ordered plan in the rocket domain in which packages are first all picked up from their initial locations and then they are all dropped off at their goal locations.

The approach of picking up all the packages and then dropping them off could be modified to use multiple rockets, but then the dsPlanner learned would be identical to that learned when multiple rockets are used to pick up and drop off each package in turn. The minimal annotated consistent partial ordering of those two plans would be identical, since the steps for each package/rocket could be executed in parallel.

If LoopDISTILL were modified to allow nested loops, it would learn the dsPlanner shown in DsPlanner 22 from the rocket-domain example shown in Figure 6.9.

In some sense, if extended to allow nested loops, LoopDISTILL could learn a dsPlanner to solve one-rocket rocket domain problems optimally—the one shown in DsPlanner 22. The catch is that making the solution optimal depends on the order in which the

DsPlanner 21 The looping dsPlanner learned by LoopDISTILL from the rocket-domain example shown in Figures 5.8. This dsPlanner solves rocket domain problems with one rocket by sending the rocket to pick up all the packages that are not at their goal destination, and then by dispatching it to deliver each of those packages to their goal destination.

```
while inCurState (at(?v1:object ?v2:loc)) and inCurState (at-rocket(?3:rocket ?v4:loc)) and in-
GoalState (at(?v1:object ?v5loc)) do
  fly(?3 ?v4 ?v2)
  load(?v1 ?3 ?v2)
end while
while inCurState (in-rocket(?3:rocket ?v1:object)) and inCurState (at-rocket(?3:rocket ?v2:loc))
and inGoalState (at(?v1:object ?v4:loc)) do
  fly(?3 ?v2 ?v4)
  unload(?v1 ?3 ?v4)
end while
```

locations are visited—a choice that cannot be captured in the dsPlanner language, as we assume that all objects that match the conditions for an if statement or while loop are treated the same—any may be chosen and there is no preference given to one over the other.

6.6 Logistics

The Logistics domain (Veloso 94a) is very similar to the Rocket domain in many ways, as it is composed of cities and packages and delivery vehicles, but is more complex, as cities can be composed of several locations, trucks can operate only within a city, and airplanes can fly between cities, but cannot visit places within a city other than the airport. There are many subproblems in the logistics domain that are isomorphic to rocket-domain problems previously discussed in Section 6.5. For example, using one or multiple trucks to deliver packages within a particular city or using one or multiple airplanes to deliver packages (located at the airports) between cities.

LoopDISTILL can also learn a dsPlanner that is composed of solutions to these problem types, and thus attack a more complex set of problems. One example, shown in DsPlanner 6.10 uses one loop to send trucks to pick up all objects not in goal destinations, and another two to deliver them either to the airport (for those packages with goal destinations not within the same city) or to their goal destinations within the same city. In the next loop, airplanes fly between the airports, picking up objects not in their goal cities. The next loop sends the airplanes to deliver the objects they are carrying. Finally, trucks carry

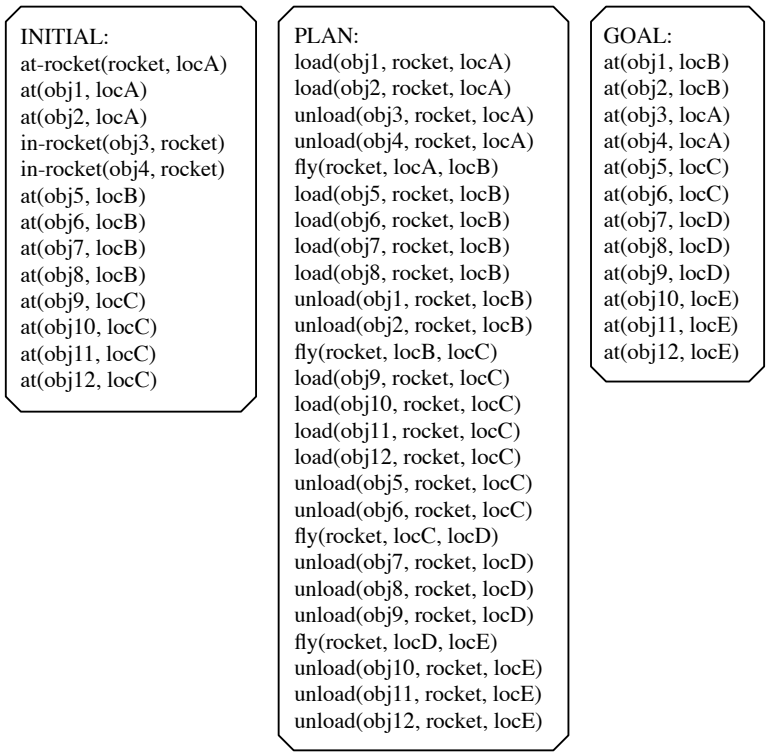


Figure 6.9: An example annotated partially ordered plan in the rocket domain in which multiple rockets are used to deliver the packages.

the objects from the airports to their goal destinations within each city.

LoopDISTILL is able to learn this dsPlanner, which solves all object-moving problems in the logistics domain, from the single example shown in Figure 6.10. Solution plans generated by this dsPlanner are on the order of $12*$ the number of misplaced objects ($nummisplacedobjs$) steps long. Without nested loops, each object may be separately picked up and dropped off by a truck (4 steps), picked up and dropped off by an airplane (4 steps), and then picked up and dropped off by a truck again (4 steps). The best case for an optimal plan would be for all the packages to be in the same initial location and for all to be going to the same destination location. All would still need to be picked up by a truck (1 load each), dropped off at the airport (1 unload each), picked up by an airplane (1 load each), dropped off at their goal city (1 unload each), picked up by a truck (1 load each), and dropped at their goal destination within the city (1 unload each), so the solution

DsPlanner 22 The looping dsPlanner LoopDISTILL could learn from the rocket-domain example shown in Figure 6.9 if it were modified to allow the discovery of nested loops. This dsPlanner solves rocket domain problems with one rocket by sending the rocket to pick up each misplaced item, and then sending it to drop off all the packages inside it to their goal destinations, picking up and dropping off packages as appropriate at each stop along the way.

```
while inCurState (at(?v1:object ?v2:loc)) and inCurState (at-rocket(?3:rocket ?v4:loc)) and in-
GoalState (at(?v1:object ?v5loc)) do
  while inCurState (at(?v6:object ?v4:loc)) and inGoalState (at(?v6:object ?v7:loc)) do
    load(?v6 ?3 ?v4)
  end while
  while inCurState (in-rocket(?v7:object, ?3:rocket)) and inGoalState (at(?v7:object ?v4:loc)) do
    unload(?v7 ?3 ?v4)
  end while
  fly(?3 ?v4 ?v2)
  load(?v1 ?3 ?v2)
end while
while inCurState (at-rocket(?8:rocket ?v9:loc)) and inCurState (at(?v10:object ?v9:loc)) and in-
GoalState (at(?v10:object ?v11:loc)) do
  load(?v10 ?8 ?v9)
end while
while inCurState (at-rocket(?11:rocket ?v12:loc)) and inCurState (in-rocket(?v13:object
?11:rocket)) and inGoalState (at(?v13:object ?v12:loc)) do
  unload(?v13 ?11 ?v12)
end while
while inCurState (in-rocket(?v14:object ?15:rocket)) and inCurState (at-rocket(?15:rocket
?v16:loc)) and inGoalState (at(?v14:object ?v17:loc)) do
  fly(?15 ?v16 ?v17)
  while inCurState (in-rocket(?15:rocket ?v18:object)) and inCurState (at-rocket(?15:rocket
  ?v17:loc)) and inGoalState (at(?v18:object ?v17:loc)) do
    unload(?v18 ?15 ?v17)
  end while
  unload(?v14 ?15 ?v17)
end while
```

plan would be on the order of $6 * nummisplacedobjs$ steps long. The learned dsPlanner finds solutions that are, at worst, twice as long as the shortest possible solution plans.

DsPlanner 23 The looping logistics-domain dsPlanner learned by LoopDISTILL from the example shown in Figure 6.10.

while inCurState (at(?v1:obj ?v2:loc)) **and inCurState** (in-city(?v2:loc ?v3:city)) **and inCurState** (at(?v4:truck ?v5:loc)) **and inCurState** (in-city(?v5:loc ?v3:city)) **and inGoalState** (at(?v1:obj ?v6:loc)) **do**
  drive-truck(?v4 ?v5 ?v2)
  load(?v1 ?v4 ?v2)
end while
while inCurState (in(?v7:obj ?v8:truck)) **and inCurState** (at(?v8:truck ?v9:loc)) **and inCurState** (in-city(?v9:loc ?v10:city)) **and inGoalState** (at(?v7:obj ?v11:loc)) **and inCurState** (in-city(?v11:loc ?v10:city)) **do**
  drive-truck(?v8 ?v9 ?v11)
  unload(?v7 ?v8 ?v11)
end while
while inCurState (in(?v12:obj ?v13:truck)) **and inCurState** (at(?v13:truck ?v14:loc)) **and inCurState** (in-city(?v14:loc ?v15:city)) **and inCurState** (in-city(?v16:airport ?v15:city)) **and inGoalState** (at(?v12:obj ?v17:loc)) **and inCurState** (in-city(?v17:loc ?v18:city)) **do**
  drive-truck(?v13 ?v14 ?v16)
  unload(?v12 ?v13 ?v16)
end while
while inCurState (at(?v19:obj ?v20:airport)) **and inCurState** (at(?v21:airplane ?v22:airport)) **and inCurState** (in-city(?v20:airport ?v23:city)) **and inGoalState** (at(?v19:obj ?v24:loc)) **and inCurState** (in-city(?v24:loc ?v25:city)) **do**
  fly-airplane(?v21 ?v22 ?v20)
  load(?v19 ?v21 ?v20)
end while
while inCurState (in(?v26:obj ?v27:airplane)) **and inCurState** (at(?v27:airplane ?v28:airport)) **and inGoalState** (at(?v26:obj ?v29:loc)) **and inCurState** (in-city(?v29:loc ?v30:city)) **and inCurState** (in-city(?v31:airport ?v30:city)) **do**
  fly-airplane(?v27 ?v28 ?v31)
  unload(?v26 ?v27 ?v31)
end while
while inCurState (at(?v32:obj ?v33:loc)) **and inCurState** (at(?v34:truck ?v35:loc)) **and** (in-city(?v33:loc ?v36:city)) **and inCurState** (in-city(?v35:loc ?v36:city)) **and inGoalState** (at(?v32:obj ?v37:loc)) **and inCurState** (in-city(?v37:loc ?v36:city)) **do**
  drive-truck(?v34 ?v35 ?v33)
  load(?v32 ?v34 ?v33)
end while
while inCurState (in(?v38:obj ?v39:truck)) **and inCurState** (at(?v39:truck ?v40:loc)) **and inCurState** (in-city(?v40:loc ?v41:city)) **and inGoalState** (at(?v38:obj ?v42:loc)) **and inCurState** (in-city(?v42:loc ?v41:city)) **do**
  drive-truck(?v39 ?v40 ?v42)
  unload(?v38 ?v39 ?v42)
end while

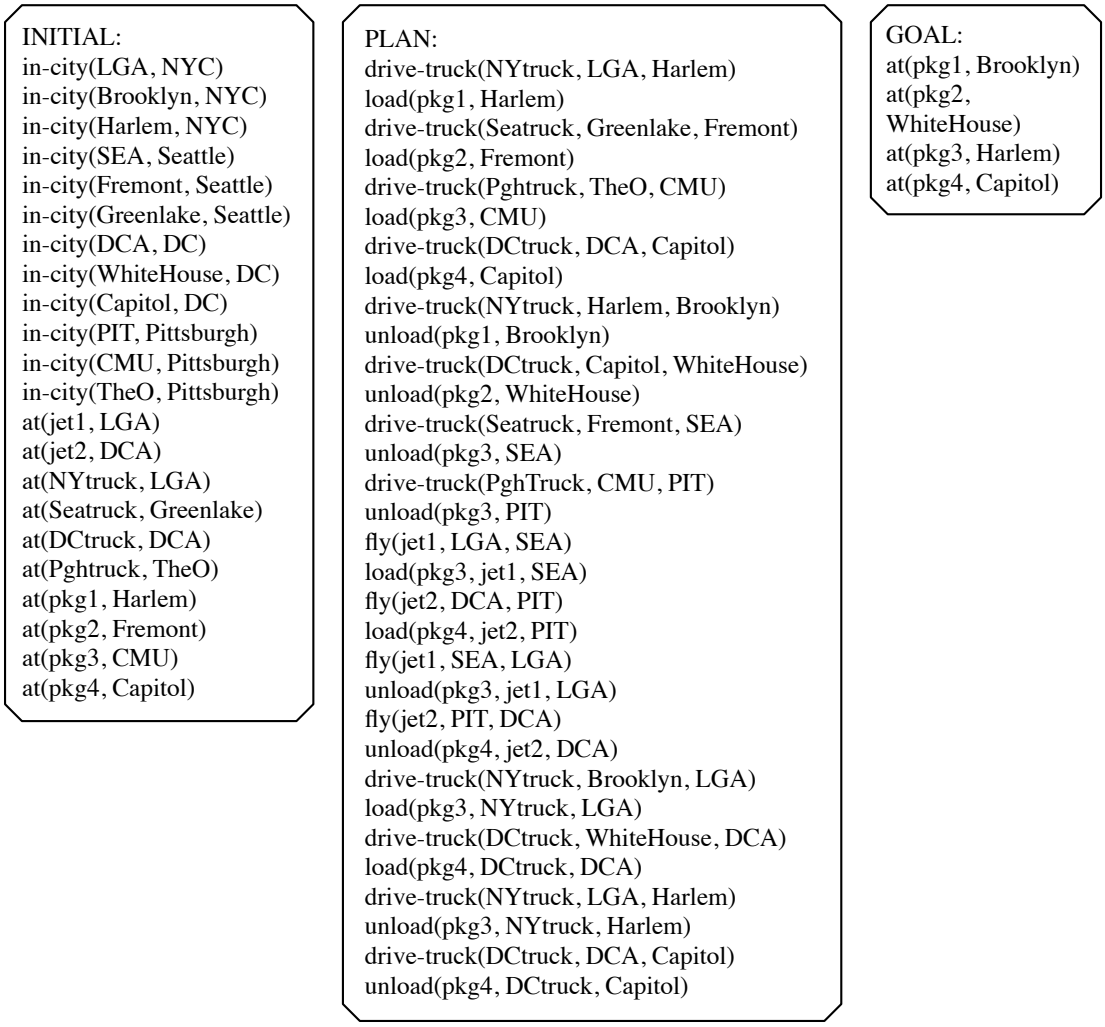


Figure 6.10: An example plan in the Logistics domain that demonstrates a looping solution algorithm for delivery problems.

6.7 Elevator

The Elevator domain (Hoffman 03) models an elevator moving up and down to different floors to pick up and drop off passengers. If we remove the directionality, using the operator move(originFloor, destFloor) to move the elevator rather than up(originFloor,

destFloor) and down(originFloor, destFloor), then the Elevator domain is isomorphic to the rocket domain with only one rocket, and LoopDISTILL can learn a compact dsPlanner that covers large subsets of the domain, as illustrated in Section 6.5.

If LoopDISTILL were extended to match non-identical iterations of a loop, it could learn a compact dsPlanner with directional move operators, as the choice to move up or down could be captured in an if-statement within the loop.

6.8 Gripper

The Gripper domain (McDermott 00) is also very similar to the Rocket domain. There is a robot with some number of gripper arms whose task is to move balls between rooms. All rooms are connected in one step (there is not a map problem embedded). If the robot has an unlimited number of arms, the gripper domain is isomorphic to the Rocket domain with one rocket, and LoopDISTILL can learn a compact dsPlanner that covers large subsets of the domain, as discussed in Section 6.5.

In the general Gripper domain, the robot has some constant number of gripper arms, each of which can carry a ball. Because the Gripper's arms can be full, we cannot simply pick up all the balls and then drop them all off as we could in the Rocket domain. Without nested loops, there isn't a way to capture that when the gripper's arms are full, it must drop off some balls before picking up more. However, we can apply another Rocket domain solution, shown in DsPlanner 19: to have the Robot move the balls one at a time.

6.9 Briefcase

The Briefcase domain (Pednault 88) is also very similar to the Rocket domain. It is isomorphic to the Rocket domain with one rocket (the briefcase) and two locations (home and work). LoopDISTILL is able to generate a compact and complete dsPlanner that provides optimal solutions in terms of number of steps in the solution plan. An example plan is shown in Figure 6.11, and the learned dsPlanner is shown in DsPlanner 24.

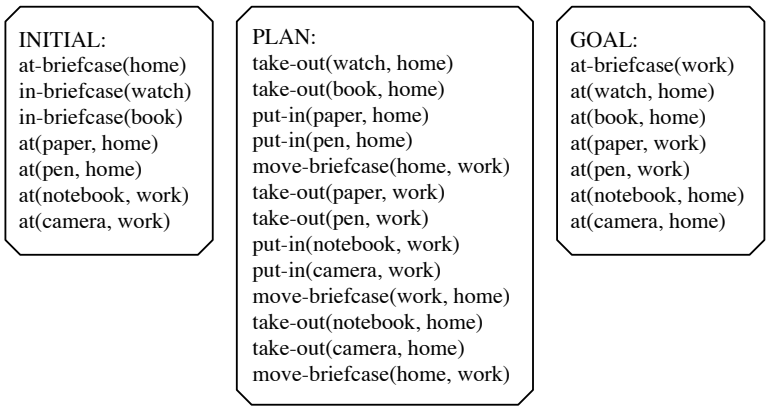


Figure 6.11: An example plan in the Briefcase domain.

6.10 Towers of Hanoi

Towers of Hanoi[1] is a well-known puzzle that lends itself to recursive solution algorithms. If there were no pegs, it would be equivalent to blocksworld—unstacking blocks and restacking them in a particular order. The difficulty with this domain arises with the pegs and capturing concepts such as "above" or "below" and of moving one disk away from its current location to a non-goal location. Perhaps future work will mine the rationales found by SPRAWL for recursive concepts and definitions.

[1]The Towers of Hanoi puzzle was invented by the French mathematician Édouard Lucas in 1883.

DsPlanner 24 The looping dsPlanner learned by LoopDISTILL from the briefcase-domain example shown in Figure 6.11. This dsPlanner is complete and optimal.

while inCurState (in-briefcase(?v1:obj)) **and inCurState** (at-briefcase(?2:loc)) **and inGoalState** (at(?v1:obj ?2:loc)) **do**
  take-out(?v1)
end while
while inCurState (at-briefcase(?3:loc)) **and inCurState** (at(?v4:obj ?3:loc)) **and inGoalState** (at(?v4:obj ?5:loc)) **do**
  put-in(?v4)
end while
if inCurState (at-briefcase(?6:loc)) **and inCurState** (in-briefcase(?7:obj)) **and inGoalState** (at(?7:obj ?8:loc)) **then**
  move-briefcase(?6 ?8)
end if
while inCurState (in-briefcase(?v9:obj)) **and inCurState** (at-briefcase(?10:loc)) **and inGoalState** (at(?v9:obj ?10:loc)) **do**
  take-out(?v9)
end while
while inCurState (at-briefcase(?11:loc)) **and inCurState** (at(?v12:obj ?11:loc)) **and inGoalState** (at(?v12:obj ?13:loc)) **do**
  put-in(?v12)
end while
if inCurState (at-briefcase(?14:loc)) **and inCurState** (in-briefcase(?15:obj)) **and inGoalState** (at(?15:obj ?16:loc)) **then**
  move-briefcase(?14 ?16)
end if
while inCurState (in-briefcase(?v17:obj)) **and inCurState** (at-briefcase(?18:loc)) **and inGoalState** (at(?v17:obj ?18:loc)) **do**
  take-out(?v17)
end while
if inCurState (at-briefcase(?19:loc)) **and inGoalState** (at-briefcase(?20:loc)) **then**
  move-briefcase(?19 ?20)
end if

Chapter 7

Related Work

The work in this thesis touches on three main areas of research: analyzing plans, learning from example plans, and program generation.

7.1 Plan Analysis

Many researchers have addressed the problems of annotating orderings and of finding partially ordered plans. We discuss a selection of the research investigating annotation and partial ordering.

7.1.1 Triangle Tables

Triangle tables are one of the earliest forms of annotation (Fikes 72). In this approach, totally ordered plans are expanded into triangle tables that display which add-effects of each operator remain after the execution of each subsequent operator. From this, it is easy to compute which operators supply preconditions to other operators, and thus to identify the relevant effects of each operator and why they are needed in the plan. Fikes, Hart, and Nilsson used triangle tables for plan reuse and modification. The annotations help to identify which sub-plans are useful for solving the new problem and which operators in these sub-plans are not relevant or applicable in the new situation.

Regnier and Fade alter the calculation of the triangle table by finding which add-effects of each operator are *needed* by subsequent operators (instead of which add-effects remain after the execution of subsequent operators) (Regnier 91). They use the dependencies

computed in this modified triangle table to create a partial ordering of the totally ordered plan.

The triangle table approach has been applied only to plans without conditional effects. When conditional effects are introduced, it is no longer obvious what conditions each operator "needs" in order for the plan to work correctly. Although we do not use the triangle table structure, our needs analysis approach can be seen as an extension of the triangle table approach to handle conditional effects.

7.1.2 Validation Structures

Another powerful approach to annotation is the validation structure (Kambhampati 89; Kambhampati 92; Kambhampati 94). This structure is an annotated partial order created during the planning process. Each partial order link is a 4-tuple called a validation: $< e, t', c, t >$, where the effect e of step t' satisfies the condition C of the step t. The validation structure acts as a proof of correctness of the plan, and allows plan modification to be cast as fixing inconsistencies in the proof. This approach is shown to be effective for plan reuse and modification (Kambhampati 92) and for explanation-based generalization of partially ordered and partially instantiated plans (Kambhampati 94). The approach has not been applied to plans with conditional effects. Although (Kambhampati 89) presents an algorithm for using the validation structures of plans with conditional effects to enable modification and reuse, no method is presented for finding these structures. And since the structures are created during the planning process, no method is presented for finding validation structures of any observed plans, even those without conditional effects.

7.1.3 Derivational Analogy

Derivational analogy (Veloso 94b) is another interesting approach to and use of annotation. In this approach, decisions made during the planning process are explicitly recorded along with the justifications for making them and unexplored alternate decisions. This approach has been shown to be effective for reusing not only previous plans, but also previous lines of reasoning. The approach can handle conditional effects, but, like the validation structure approach, is applicable only to plans that have been created and annotated by the underlying planner.

7.1.4 Operator Graphs

The final approach to annotation that we will discuss is the operator graph (Smith 93; Smith 96). This approach does not analyze plans, but rather interactions between operators relevant to a problem. The operator graph includes one node per operator, and one node per precondition of each operator. A link is made between each node representing a precondition of an operator and the operator node, and between the node of each operator which satisfies a particular precondition and the node representing that precondition. A threat link is also added between the node of each operator which deletes a particular precondition and the node representing that precondition. Smith and Peot use these operator graphs before the planning process to discover when threat resolution may be postponed (Smith 93) and to analyze potential recursion (Smith 96). Operator graphs do not apply to domains with conditional effects, and are less applicable to plan reuse and behavior modeling than other approaches, since they analyze operator interactions, not plans.

7.1.5 Partially Ordering Totally Ordered Plans

There has been some previous work on finding partial orderings of totally ordered plans. As previously mentioned, Regnier and Fade (Regnier 91) used triangle tables to do this for plans without conditional effects. Veloso et al also presented a polynomial-time algorithm for finding a partial ordering of a totally ordered plan without conditional effects (Veloso 90). The algorithm adds links between each operator precondition and the most recent previous operator to add the condition. It then resolves threats and eliminates transitive edges. However, Bäckström shows that this method is not guaranteed to find the most parallel partial ordering, and that, in fact, finding the *optimal* partial ordering according to any metric is NP-complete (Bäckström 93).

7.1.6 Partial Order Planning

There has been a great deal of research into generating partially ordered plans from scratch. UCPOP (Penberthy 92) is one of the most prominent partial-order planners that can handle conditional effects. One of the strengths of UCPOP is its non-determinism; it is able to find all partially ordered plans that solve a particular problem. However, it is difficult to use the same technique to partially order a given totally ordered plan. The total order contains valuable information about dependencies and orderings, but the UCPOP method would discard this information and analyze the orderings from scratch. Not only is this

inefficient, but it may result in a partial ordering of the totally ordered steps that is not consistent with the total order.

Graphplan (Blum 97), another well-known planner, is also able to find partially ordered plans in domains with conditional effects (Anderson 98). However, it produces suboptimal and non-minimal (overconstrained) partial orderings, which does not suit our purpose. Consider the plan in which the steps $op\_a\_1 \ldots op\_a\_n$ may run in parallel with the steps $op\_b\_1 \ldots op\_b\_n$. Graphplan would find the partial ordering shown in Figure 7.1 because it only finds parallelism within an individual time step. In the first time step, op_a_1 and op_b_1 may run in parallel, but there is no other operator that may run in parallel with them, so Graphplan moves to the second time step (in which op_a_2 and op_b_2 may run in parallel). Graphplan constrains the ordering so that no operators from one time step may run in parallel with operators from another. None of the ordering constraints between op_a steps and op_b steps help achieve the goal, so they are not included in the partial ordering created by SPRAWL, shown in Figure 7.2. SPRAWL reveals the independence of the two sets of operators.

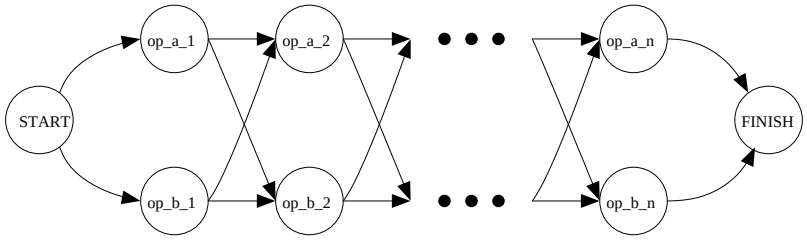


Figure 7.1: This partial ordering, found by Graphplan, contains many irrelevant ordering constraints.

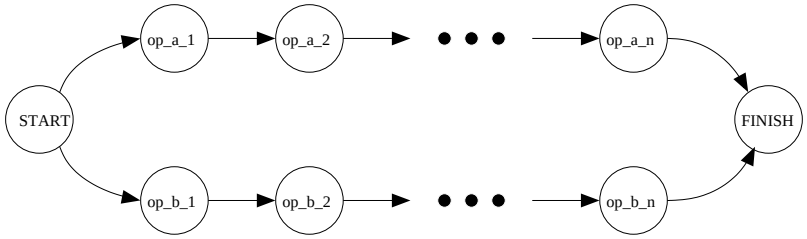


Figure 7.2: This partial ordering, found by SPRAWL, contains only necessary ordering constraints.

7.2 Domain Knowledge to Reduce Planning Search

A large body of work has focussed on acquiring and using domain knowledge to reduce planning search. Our work seeks to avoid this search altogether by generating a domain-specific planning algorithm.

7.2.1 Control Rules

Much of the work on learning domain knowledge for planning is focused on learning control rules (Minton 88a; Katukam 94; Etzioni 93), which 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. However, they have been shown to reduce dramatically the planning time required for certain problems. Learning and managing control rules are the two biggest difficulties in using them. 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 88b), and computer-learned control rules are often ineffective (Minton 88b). 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 88b).

7.2.2 Macro Operators

Another common type of learned control knowledge for planning is the macro operator (Fikes 72; Korf 85), which combines a frequently-occurring sequence of operations into one combined operator. A macro can then be applied by the planner in one step, thus eliminating the search required to find the entire sequence again. Macros have been shown to be very effective for reducing search in hierarchically decomposable domains, such as towers of Hanoi and sliding n-puzzle, because they can capture the common repeated sequences that make up almost every solution in these domains. However, Macros, like control rules, suffer from the utility problem: it is difficult to determine when to add a new macro operator, since adding too many can slow down the planning process. 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 85), but the efficacy of macros has, in general, been limited to hierarchically decomposable domains.

7.2.3 Case-Based Reasoning

Another approach to learning planning knowledge, case-based reasoning, attempts to avoid generative planning for many problems (Hammond 96; Kambhampati 92; Leake 96). 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 94a), 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. Other difficulties with case-based planning include finding an appropriate similarity metric between problems, determining how to modify an existing plan to solve a new problem, and determining when it would be faster simply to plan for the new problem from scratch. However, case-based planners have succeeded in solving larger problems than can be solved by generative planning alone, and, in general, find solutions faster than generative planners (Hammond 96).

7.2.4 Analogical Reasoning

A variant of case-based reasoning that deserves mention is analogical reasoning, which also stores case libraries and modifies previous cases to solve new problems (Veloso 94a; Veloso 94b). 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.

7.2.5 Hierarchical Planning

Domain knowledge is also used to hierarchically divide planning domains (Sacerdoti 74; Knoblock 90; Knoblock 94). This allows the planner to simplify the problem by reasoning about it at a higher level of abstraction. The planner must then use the abstracted solution to find a solution to the full problem. The greatest difficulty with hierarchical planning is identifying a good abstraction hierarchy. Such a hierarchy will abstract away enough details that problem solving is easy at the highest level, but will not abstract away important details; the abstract solution must be easily transformable into a solution at lower levels of abstraction. Information about how to hierarchically divide planning problems may be user-supplied (Sacerdoti 74) or may be discovered automatically through an analysis of the planning domain (Sacerdoti 74; Knoblock 90; Knoblock 94).

7.2.6 Skeletal Planning

Skeletal planning combines hierarchical planning and case-based reasoning by storing previously generated plans and selectively abstracting them to reduce the size of the case library. The theory behind skeletal planning is that plans in a particular domain are divided into a limited number of plan classes and that a single abstract skeletal plan can represent the solutions for all plans in a given class (Friedland 85). Skeletal planning work does not assume that these classes are given to the problem solver. Instead, problem instances are encountered and stored in the case library. When several similar plans are encountered, they are generalized into a skeletal plan that is an abstracted version of all of them. Most skeletal planning work has relied on human users to do this generalization, but some work has been done on applying explanation-based learning to the automatic acquisition of skeletal plans (Bergmann 92).

7.2.7 Meta-Planning

Work in meta-planning seeks to allow planners to reason about the planning process as well as about the problem they are solving. Meta-planning is used both to reason explicitly about which planning steps to take next (such as working on a particular goal, refining an operator, or propagating a constraint) (Stefik 81), to plan how to resolve goal conflicts or other problems in planning (Wilensky 81), or to explicitly formulate constraints about the kind of plan that should be found (one which uses the fewest resources, achieves the most goals, maximizes the value of the goals achieved, or avoids impossible goals) (Wilensky 81).

7.3 Automatic Program Generation

Work in many fields has addressed the problem of automatically generating programs. This work can be divided into two main classes: deductive program synthesis, in which programs are generated from specifications; and inductive program synthesis, in which, as in our work, programs are generated from example executions. Descriptions of work applying inductive program synthesis to planning and action selection follow a brief summary of general deductive and inductive program synthesis.

7.3.1 Deductive Program Synthesis

Deductive program synthesis is the automatic generation of programs from specifications. Researchers have identified many different approaches to attacking this problem. Manna and Waldinger (Manna 92) approach this task as theorem proving: a user describes the input and the desired output of the algorithm and the program synthesis system uses a constructive proof of the algorithm's existence to identify the algorithm. Smith's KIDS system (Smith 91) solves program synthesis in a transformative way: after specifing types and basic functions over those types, the user writes program specifications in a formal language. The KIDS system then applies correctness-preserving transformations to those specifications to create an algorithm that fulfills them. Williams describes the TA system (Williams 88), which learns to generate programs that satisfy specifications by examining and modifying other programs.

There are many difficulties with deductive program synthesis (Rich 92). For example, although many claim that deductive program synthesis will become feasible for end users who are not familiar with programming, program synthesis systems require extensive domain knowledge in the form of types and functions over those types. Generating this domain knowledge requires familiarity with programming methods. Also, these systems depend on program specifications written by their users, so the specifications must be complete and correct in every situation for the resulting program to be correct. But writing complete and correct specifications is a difficult task, and Rich and Waters argue that user-written specifications actually cannot be complete; not only would completeness require that the user specified how the system should act in every possible situation, but it would also require the user to specify fuzzy requirements for efficiency and hard-to-formulate tradeoffs between different aspects of the generated system's performance.

7.3.2 Inductive Program Synthesis

In part to sidestep the excessive demands of completely and correctly describing a desired system, many researchers have investigated inductive program synthesis, or the automatic generation of an algorithm from examples of its desired execution (either example traces of the execution or example input-output pairs). Programs generated inductively cannot be guaranteed to be correct or complete, but input-output pairs (or traces) are often easier to generate than complete and correct specifications, and the generated programs usually cover the "common cases" (the problems like those they are induced from).

Our work on extracting algorithmic models of behavior from observed executions falls within the category of inductive program synthesis (IPS). However, previously developed approaches to IPS are not immediately applicable to our problem. Some IPS algorithms induce programs from input-output pairs (Muggleton 91; Muggleton 94). In planning, this corresponds to inducing an algorithm from example initial and goal states, a formidable task. The problem is made easier in general IPS systems because they are also given a good deal of background information about programming techniques and methods to apply to a problem. This corresponds to background knowledge about how to solve different planning problems in a particular domain. We have chosen not to provide such knowledge.

Some approaches to IPS use program traces as input (Bauer 79; Lau 01), as does our work. However, the 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.

Finally, whereas many approaches to IPS must attempt to induce the purpose of the steps from many examples (Lau 01), 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.

Iterative and Recursive Macro Operators

Inductive program synthesis has been used to generate iterative and recursive macro operators (Schmid 01; Schmid 00; Shell 89; Shavlik 90). These macros capture repetitive behavior and can drastically reduce planning search by encapsulating an arbitrarily long string of operators. However, this technique does not attempt to replace the generative planner, and so does not eliminate planning search.

Decision Lists for Planning

Some work has focussed on analyzing example plans to reveal a strategy for planning in a particular domain in the form of a decision list (Khardon 99), or a list of condition-action pairs. Conditions consist of lists of predicates true in the current state and in the goal state and are of limited length. The decision list is created by transforming the example plans into a set of state-action pairs, enumerating all possible condition-action pairs, evaluating each based on coverage and accuracy on the state-action pairs extracted from the example plans, and adding the best pair (according to some quality criterion) to the decision list, eliminating covered state-action examples, until all examples are covered. Planning with the decision list consists of repeatedly checking the condition-action rules against the current state and goal. When a rule applies, the action is attempted, and matching begins again from the beginning of the decision list.

The decision list algorithm can find a strategy even when the plans it is given cannot be described by a simple strategy (e.g., they are optimal solutions to NP-hard problems). However, it is able to solve fewer than 50% of 20-block Blocksworld problems, and requires over a thousand state-action pairs to achieve that coverage (Khardon 99). Enumerating all possible conditions and evaluating them against every observed state-action pair may become prohibitively slow for more complex domains than Blocksworld.

More recent approaches have eliminated the need for background information and have achieved comparable coverage to domain-independent planners on most domains in the International Planning Competition (Fern 06). However, many examples (either provided or generated) and hours (or even days) of computation time were required to achieve that coverage. The decision list approach requires many examples and so much computation in part because by breaking up example plans into state-action pairs, it disposes of much of the information contained in the plans, such as sequencing and looping behaviors. By preserving this information, we have shown we can rapidly achieve broad coverage from as few as one example.

7.4 Universal Planning

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

The simplest form of a universal plan is a table with an entry for every possible situation (state and goal) that specifies the action to take in that situation, as in general so-

lutions to Markov decision processes, or the Q tables of standard reinforcement learning (Mitchell 97). In complex domains, the size of these tables is prohibitively large, as is the cost of acquiring them. Many methods have been developed to store this information more compactly.

Many researchers have used reinforcement learning to acquire compact Q functions rather than the prohibitively large Q tables. One common form of Q function is a neural network (Anderson 87). Q functions may also be represented as decision trees. This format has been found to be very expressive, frequently outperforming neural network representations (Pyeatt 98), however, the decision tree format is less compressed than neural networks. Some work has been done on generating more compressed decision tree Q functions by avoiding the repetition of common substructures within the tree (Uther 00). Relational reinforcement learning provides a method for learning a Q function represented as a logical regression tree over parameterized operators and predicates (Džeroski 98; Driessens 01). This allows a solution to one problem to be used to solve another similar problem. However, Q functions learned in this way from small problems cannot be applied to larger or more complex problems, and it is unclear whether the size of the learned regression tree is prohibitive 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 87), 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 98a; Cimatti 98b; Jensen 00). 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.

Chapter 8

Conclusions

We first present the contributions of this work. We then explore possible future directions for this work, and finally revisit the contributions of the thesis.

8.1 Contributions

There are five main contributions of this thesis.

The dsPlanner language We introduced a compact, readable, and powerful language for describing domain-specific planners. We have shown that compact planners written in this language either can completely cover or can cover large portions of known domains. Planners written in this language execute in linear time modulo state matching effort. We also introduced the concept of learning complete planning programs—rather than control rules, macros, libraries of past examples, or decision lists—from examples.

The SPRAWL **algorithm** We developed the SPRAWL algorithm for finding the rationale behind an observed plan, in the form of a minimal annotated consistent partial ordering of the observed actions. SPRAWL is able to analyze plans with conditional effects, unlike previous approaches, and is able to execute in polynomial time in domains without conditional effects.

The DISTILL **algorithm** With the DISTILL algorithm, we showed how to learn a non-looping dsPlanner from an observed plan, and how to merge two non-looping dsPlanners to achieve situational generalization as well as significant space savings over a case library.

The LoopDISTILL algorithm We introduced the LoopDISTILL algorithm for learning a looping dsPlanner from an example plan. Loops allow a learned planner to cover

problems of arbitrary size. Our approach is based on a SPRAWL analysis of the observed plan, and we show the startling power of this explicit analysis; we are able to learn dsPlanners with broad coverage from a single example in under a second. Other approaches require dozens, or even hundreds, of examples and hours of computation time.

A thorough empirical evaluation of this approach In this thesis, we not only have contributed algorithms to solve problems, but we have tested our approach on a wide variety of well-known planning domains and carefully explored the kinds of problems this approach can attack well, and the kinds it cannot.

8.2 Discussion

8.2.1 "Best" dsPlanner

While the algorithms presented are guaranteed to lead to a dsPlanner that solves at least the problem it was learned from, they don't necessarily lead to a dsPlanner that's "best" according to any metric—compactness, solution length, solution parallelism, breadth of coverage, etc. Instead, the learned planner is faithful to the example presented. It is well-known that defining what makes one program better than another is a difficult problem in and of itself . Our goal was, in part, for the learning of the dsPlanner to be fast enough for it to be used online. Searching through and evaluating according to some unknown metric all possible dsPlanners that could be learned from a given example would make that impossible.

To begin with, a single totally ordered example plan may represent multiple consistent and minimal partial orderings, and the dsPlanner learned by LoopDISTILL is derived from the single annotated partial ordering that is its input. In order to determine which partial ordering of many would lead to the "best" dsPlanner, all would have to be tried.

The order of loop discovery can also affect the eventual dsPlanner, as discovering and separating some loops can prevent others from being discovered. The current LoopDISTILL algorithm first finds all possible parallel loops (starting its search from the beginning of the plan) and then finds all possible serial loops (starting its search from the beginning of the plan). But in order to find some "best" dsPlanner, we would have to undertake an exhaustive search through dsPlanners created by separating each possible loop in every possible order.

In identifying parallel loops, we use a greedy algorithm that prefers matches with more steps per parallel iteration. One could also prefer matches with more parallel iterations,

or some combined function of the two. Regardless of the metric that is used, though, as mentioned above, the order of loop discovery can affect future loop discovery, so to find some "best" dsPlanner, we would have to undertake an exhaustive search through dsPlanners created by separating every possible subset of iterations into a loop and proceeding with loop discovery in each dsPlanner.

Searching through all possible dsPlanners that could be learned from a given example would be very time-intensive. Instead, we have focussed on learning the dsPlanners quickly, and our results show that useful dsPlanners with broad coverage can be learned very quickly for many domains.

8.2.2 Soundness, Correctness, and Completeness

Sound planners do not output solution plans that don't solve the given problem. Correct planners do not suggest solution steps that are not executable. Complete planners solve all problems in a particular domain. The dsPlanners generated by LoopDISTILL are both sound and correct, but, as for any learned knowledge, are not necessarily complete. DsPlanners are guaranteed to be sound—to return only plans that solve the given problem—by construction, because they simulate the execution of the plan as they generate it; it is clear by the end of the execution of the dsPlanner whether the plan generated can solve the problem or not, so the dsPlanner can simply not return unsuccessful plans. DsPlanners are also guaranteed to be correct—not to suggest non-executable steps—because each plan step in the dsPlanner is contained within either an if statement or a while loop whose conditions guarantee that the preconditions of the step are always met before it is suggested by the dsPlanner.

Even when dsPlanners are not able to find a solution plan, the partial solutions they find could be used as guidance for a generative planner.

8.2.3 Coverage

Conjecture If there exists a dsPlanner representation of a class of problems within a domain and an example that illustrates the structure of that domain, then LoopDISTILL could learn the dsPlanner.

Exceptions

Hidden Iteration Ordering Constraints As we discuss in section 8.3.3, for some loops, LoopDISTILL can capture the structure of the loop, but cannot detect that the iterations must execute in a particular order because the ordering is represented in the partial ordering found by SPRAWL as threat links between the iterations. The current LoopDISTILL algorithm does not mine threat links for information on how to build the dsPlanner, but we believe that these links could provide information that could illuminate these hidden ordering constraints, and that thorough analysis may allow the automatic discovery of recursive definitions like "above" or "below" in Blocksworld.

Nested Loops The LoopDISTILL code does not identify nested loops or allow for if-statements within other if-statements, or within loops, so it could not discover a dsPlanner with these structures.

8.3 Future Work

Our work opens the door to many avenues of future work and exploration. The most direct is to extend the LoopDISTILL algorithm to allow it to extract more from each example plan and model more complex programs, for example by merging looping dsPlanners, capturing nested loops and hidden ordering constraints, and matching non-identical iterations. Another interesting direction for future research would be to explore how to learn dsPlanners from examples that don't come from a benevolent teacher, or that don't reflect an underlying program.

One of the most enticing avenues opened by LoopDISTILL is that of real-time deliberation. With a fast, flexible, learning planning paradigm, real-time agents in complex environments need not be only hand-coded or reactive. True deliberation and explicit reasoning about actions have not been available for time-sensitive applications, as they have simply taken too long. With LoopDISTILL, it is possible for a sound planner with broad coverage to be learned in under a second from a single observed example. With the addition of merging looping dsPlanners, it is easy to imagine a learner deploying with access to a teacher, learning a true planner online in real time which it could then use to solve new problems in linear time.

LoopDISTILL is also naturally suited to agent modelling, as the learned dsPlanner is a model of the observed behavior of the teacher. For our current algorithms to apply, we would need to know the goals of the observed agent in order to analyze their execution.

We would also need to observe a complete sequence of actions (which achieve the agent's goals) and could only model algorithms that can be described by the dsPlanner language. It would be interesting to work towards extracting planners from partial executions (so that an "interrupted" trace could also be mined), and to further explore extensions to the dsPlanner language and learning from examples that do not demonstrate a dsPlanner-describable algorithm.

We now describe in detail some of the next steps that could be taken to extend the capabilities of LoopDISTILL .

8.3.1 Merging Looping dsPlanners and Matching non-Identical Loop Iterations

Merging complex plans, in which there are multiple threads of execution, many of which may affect each other in a variety of ways, and some of which have loops, is very challenging. There are several challenges involved. One is in deciding which subplans should be considered to be matching. Another is determining how to combine dsPlanners while preserving as much flexibility and expressivity as possible in the resulting dsPlanner to robustly and appropriately match subsequent plans as well. Finally, it's important to determine whether any guarantees can be made about the merged dsPlanner solving at least as many problems as the two original dsPlanners combined—that incorporating new plans into a dsPlanner allows us to monotonically increase the number of problems we're able to solve.

It is not clear how aggressively to match non-identical subplans. Trivially, identical subplans can be matched, but subplans supporting different or larger goals, subplans that differ in some of their component steps, subplans differently connected within their "parent" plans, may also be matched. The same issues are involved in matching non-identical iterations of a single loop; it isn't clear how far to take the matching. In this work, we have considered only identical subplans to be matching, but there is much to be gained from merging non-identical subplans. Obviously, two 42-step subplans that differ in a single initial step and a single current state condition, for example, should be considered to match, and the differing step can be contained in an if-statement within the new loop—but it's less clear what to do with two subplans that differ in a larger portion of their steps or conditions.

Merging dsPlanners into one combined dsPlanner involves making choices about how to order unmatched segments of the planner, and to which potentially (or partially) matching portion of one dsPlanner to match a segment of the other. Deciding which portions of

the planners to match would be difficult to revisit or reason about later without undertaking an exhaustive search either through all separately recorded component dsPlanners as the combined dsPlanner is progressively updated, or through all possible combinations of the component dsPlanners. Both approaches would likely be infeasible in a time-sensitive application.

However, it is relatively easy to delay commitment about the ordering of unmatched segments of the dsPlanner. I believe that the best way to store dsPlanners as they are being learned and refined is in annotated partial orderings like those created by SPRAWL. Any merged sections of component dsPlanners would be merged in the partially ordered combined dsPlanner, with links coming into and out of the merged section from and to non-merged sections of the component dsPlanners. Any ordering constraints imposed by the partial ordering would be necessary given which sections were merged, and all other ordering decisions could be deferred until the dsPlanner needed to be executed. But even at that point, the dsPlanner, while it must be executed as a total ordering, could be maintained for future updating or reordering in a partial ordering.

8.3.2 Nested Loops

The current LoopDISTILL code does not identify nested loops, but it is not difficult, in principle, to do so. A loop may be treated like any other plan step. For two loops to match, not only would there need to exist a variable substitution that would allow their steps to match, but they would both need to have the same varying parameters—for example, in a rocket domain problem, a loop that cycles over locations would not match a loop that cycles over objects, even if they were otherwise the same—or a new loop could be created that varies over both parameters. The current LoopDISTILL algorithm can identify nested loops with an appropriate refinement of its step matching process. In DsPlanner 25, we show one possible rocket domain dsPlanner that such a modified LoopDISTILL algorithm could identify.

8.3.3 Uncovering Iteration Ordering Constraints and Expressing Recursive Definitions

For some serial loops, LoopDISTILL can capture the steps of the loops but not the ordering between the iterations. For example, in the Blocksworld domain, the ordering of the steps in a block-stacking loop intended to build a particular tower are explained in the plan rationale found by SPRAWL as threat links in the partial ordering—the bottom blocks need

DsPlanner 25 Hand-written rocket-domain dsPlanner with nested loops.

```
while inCurState (at(?v1:object ?v2:loc)) and inCurState (at-rocket(?v3:loc)) and inGoalState
(at(?v1:object ?v4:loc)) do
  while inCurState (in-rocket(?v5:object)) and inGoalState (at(?v5:object ?v3)) do
    unload(?v5 ?v3)
  end while
  fly(?v3 ?v2)
  while inCurState (at(?v6:object ?v2)) and inGoalState (at(?v6:object ?v7:location)) do
    load(?v6 ?v2)
  end while
  fly(?v2 ?v4)
  while inCurState (in-rocket(?v8:object)) and inGoalState (at(?v8:object ?v4)) do
    unload(?v8 ?v4)
  end while
end while
while inCurState (in-rocket(?v9:object)) and inCurState (at-rocket(?v10:loc)) and inGoalState
(at(?v9:object ?v11:location)) do
  fly(?v10 ?v11)
  while inCurState (in-rocket(?v12:object)) and inCurState (at-rocket(?v?11)) and inGoalState
  (at(?v12:object ?v11)) do
    unload(?v12 ?v11)
  end while
end while
```

to be stacked before the top blocks because otherwise blocks that need to be moved are free to do so. The current LoopDISTILL algorithm does not mine threat links for information on how to build the dsPlanner [1], but it would be interesting to investigate doing so, and how such ordering constraints could be represented in the dsPlanner language.

Similarly, the current dsPlanner language has no way to describe a dsPlanner to clear a particular block when an arbitrary stack of blocks not part of the solution are on top of that block. I believe relationships such as this can be captured from the annotated partial orderings generated by SPRAWL and exploited to expand the expressive capacity of the dsPlanner language and of the dsPlanners learned by LoopDISTILL.

[1] LoopDISTILL does take threat links into account when determining how to order operators, but protected terms on a threat link are not included in the conditions for executing an if-statement or while loop.

8.4 Summary

This dissertation contributes a clear, expressive, human-readable and -writeable programming language for domain-specific planners, or dsPlanners; the SPRAWL algorithm for discovering the rationale underlying observed plans; the DISTILL algorithm for learning non-looping dsPlanners from a set of observed examples supplemented with their rationales; and the LoopDISTILL algorithm for learning a looping dsPlanner from an example plan and its rationale. We show that by thoroughly analyzing observed example plans, our algorithms are able to learn a dsPlanner with broad coverage in many domains from a single example in under a second, and that such dsPlanners can then be used to solve arbitrarily large problems in linear time, modulo state-matching effort. The core contribution of this thesis is a new planning paradigm in which domain-specific planners are learned by example.

Appendix A

Sprinkler Domain

```
(define (domain sprinkler-adl)
(:types location - object
thing - object)
(:predicates (at ?x - thing ?l - location)
     (wet ?x - object))

(:action sprinkle
 :parameters (?l - location)
 :effect (and (wet ?l)
      (forall (?x - thing)
      (when (and (at ?x ?l)
 (not (wet ?x)) )
                              (wet ?x) ) ) ) )

(:action move
 :parameters (?x - thing ?from ?to - location)
 :precondition (and (at ?x ?from) )
 :effect (and (at ?x ?to)
              (not (at ?x ?from)) ) )
)
```

Appendix B

Rocket Domain

```
(define (domain rocket-adl)
(:types location - object
thing - object
item - thing
rocket - thing)
(:predicates (at ?x - thing ?l - location)
     (inside ?i - item ?r - rocket))
(:action load
 :parameters (?i - item ?r - rocket ?l - location)
 :precondition (and (at ?i ?l)
    (at ?r ?l))
 :effect (and (inside ?i ?r)
      (not (at ?i ?l)))
)

(:action unload
 :parameters (?i - item ?l - location ?r - rocket)
 :precondition (and (at ?r ?l)
    (inside ?i ?r))
 :effect (and (at ?i ?l)
      (not (inside ?i ?r)))
)

(:action move
 :parameters (?r - rocket ?from - location ?to - location)
 :precondition (and (at ?r ?from) )
 :effect (and (not (at ?r ?from))
      (at ?r ?to))
)
```

)

Appendix C

Blocksworld Domain

```
(define (domain blocksworld-2-adl)
        (:types block - object)
        (:predicates (on-block ?x - block ?b - block)
     (on-table ?x - block)
                    (clear ?x - block))

        (:action move-b-b
         :parameters (?x ?from ?to - block)
         :precondition (and (on-block ?x ?from)
                            (clear ?to)
                            (clear ?x))
         :effect (and (on-block ?x ?to)
       (not (on-block ?x ?from))
       (clear ?from)
       (not (clear ?to)) )
         )

        (:action move-b-t
         :parameters (?x ?from -block)
         :precondition (and (on-block ?x ?from)
                            (clear ?x))
         :effect (and (on-table ?x)
       (not (on-block ?x ?from))
       (clear ?from))
         )

        (:action move-t-b
         :parameters (?x - block ?to - block)
         :precondition (and (on-table ?x)
```

```
                            (clear ?to)
                            (clear ?x))
          :effect (and (on-block ?x ?to)
        (not (on-table ?x))
        (not (clear ?to)) )
          )
)
```

Appendix D

Gripper Domain

```
(define (domain gripper2)
   (:types room - object
        ball - object)
   (:predicates (at-robby ?r - room)
(at ?b - ball ?r - room)
(free-arm)
(holding ?o - ball)
)
   (:action move
       :parameters  (?from ?to - room)
       :precondition (at-robby ?from)
       :effect (and  (at-robby ?to)
     (not (at-robby ?from)))
     )
   (:action pick
       :parameters (?obj - ball ?room - room)
       :precondition  (and  (at ?obj ?room)
                            (at-robby ?room)
                            (free-arm))
       :effect (and (holding ?obj)
    (not (at ?obj ?room))
    (not (free-arm)))
    )
   (:action drop
       :parameters  (?obj - ball ?room - room)
       :precondition  (and  (holding ?obj)
                            (at-robby ?room))
       :effect (and (free-arm)
                (at ?obj ?room))
```

```
    (not (holding ?obj))
    )
)
```

Appendix E

Dishwashing Domain

```
(define (domain dishes)
(:types dish - object
sponge - object)
(:predicates (dirty ?x - dish)
     (clean ?x - dish)
     (soapy ?x - sponge)
     (havesoap)
     )
(:action wash
 :parameters (?x - dish ?y - sponge)
 :precondition (and (dirty ?x)
              (soapy ?y) )
 :effect (and (clean ?x)
        (not (soapy ?y))
      (not (dirty ?x)) )
)
(:action soap
 :parameters (?x - sponge)
 :precondition (havesoap)
 :effect (and (soapy ?x))
        )
)
```

Appendix F

Multi-Step Parallel Loop Domain

```
(define (domain multistepparallelloop)
        (:types type1 - object)
        (:predicates (s ?x - type1)
                     (g ?x - type1))
        (:action op1
         :parameters (?x - type1)
         :precondition (s ?x)
         :effect (and (a1 ?x))
        )
        (:action op2
         :parameters (?x - type1)
         :precondition (s ?x)
         :effect (and (a2 ?x))
        )
        (:action op3
         :parameters (?x - type1)
         :precondition (and (a1 ?x)
                            (a2 ?x))
         :effect (and (g ?x))
        )
)
```

Appendix G

Multi-Step Serial Loop Domain

```
(define (domain multistepserialloop)
        (:types type1 - object
                type2 - object)
        (:predicates (s ?x - type1)
                     (g ?x - type1)
                     (a1 ?x - type1)
                     (a2 ?x - type1)
                     (b1 ?z - type2)
                     (b2 ?z - type2))
        (:action op1
         :parameters (?x - type1 ?z - type2)
         :precondition (and (s ?x)
                            (b1 ?z))
         :effect (and (a1 ?x)
                      (not (b1 ?z)))
        )
        (:action op2
         :parameters (?x - type1 ?z - type2)
         :precondition (and (s ?x)
                            (b2 ?z))
         :effect (and (a2 ?x)
                      (not (b2 ?z)))
        )
        (:action op3
         :parameters (?x - type1 ?z - type2)
         :precondition (and (a1 ?x)
                            (a2 ?x))
         :effect (and (g ?x)
                      (b1 ?z)
```

```
                    (b2 ?z))
            )
    )
```

Appendix H

Schedule Domain

```
(define (domain schedule)
(:types part - object
color - object
width - object
orientation - object)
(:predicates (isHot ?x - part)
          (isCold ?x - part)
    (isCyllindrical ?x - part)
    (hasHole ?x - part ?y - width ?z - orientation)
    (punchHasBit ?y - width)
    (drillPressHasBit ?y - width)
    (punchCanOrient ?z - orientation)
    (drillPressCanOrient ?z - orientation)
    (sprayerHasPaint ?x - color)
    (immerserHasPaint ?x - color)
    (isPainted ?x - part ?y - color)
    (isUnpainted ?x - part)
    (busyPolisher)
    (idlePolisher)
    (busyRoller)
    (idleRoller)
    (busyLathe)
    (idleLathe)
    (busyGrinder)
    (idleGrinder)
    (busyPuncher)
    (idlePuncher)
    (busyDrillPress)
    (idleDrillPress)
```

```
    (busySprayer)
    (idleSprayer)
    (busyImmerser)
    (idleImmerser)
    (scheduledPolisher ?x - part)
    (scheduledRoller ?x - part)
    (scheduledLathe ?x - part)
    (scheduledGrinder ?x - part)
    (scheduledPuncher ?x - part)
    (scheduledDrillPress ?x - part)
    (scheduledSprayer ?x - part)
    (scheduledImmerser ?x - part)
    (unscheduled ?x - part)
    (isSmooth ?x - part)
    (isPolished ?x - part)
    (isRough ?x - part)
    )
(:action doPolish
   :parameters (?x - part)
 :precondition (and (unscheduled ?x)
                (idlePolisher)
 (isCold ?x))
 :effect (and (busyPolisher)
        (not (idlePolisher))
      (scheduledPolisher ?x)
      (not (unscheduled ?x))
      (isPolished ?x)
      (not (isRough ?x))
      (not (isSmooth ?x)))
)
(:action doRoll
 :parameters (?x - part)
 :precondition (and (unscheduled ?x)
                (idleRoller))
 :effect (and (busyRoller)
        (not (idleRoller))
      (scheduledRoller ?x)
      (not (unscheduled ?x))
      (isUnpainted ?x)
      (isRough ?x)
      (not (isSmooth ?x))
      (not (isPolished ?x))
      (isHot ?x)
      (not (isCold ?x))
      (isCyllindrical ?x))
)
```

```
(:action doLathe
 :parameters (?x - part)
 :precondition (and (unscheduled ?x)
                (idleLathe))
 :effect (and (busyLathe)
        (not (idleLathe))
      (scheduledLathe ?x)
      (not (unscheduled ?x))
      (isRough ?x)
      (not (isSmooth ?x))
      (not (isPolished ?x))
      (isCyllindrical ?x)
      (isUnpainted ?x))
)
(:action doGrind
 :parameters (?x - part)
 :precondition (and (unscheduled ?x)
              (idleGrinder))
 :effect (and (busyGrinder)
(not (idleGrinder))
(scheduledGrinder ?x)
(not (unscheduled ?x))
(isSmooth ?x)
(not (isPolished ?x))
(not (isRough ?x))
(isUnpainted ?x))
)
(:action doPunch
 :parameters (?x - part ?y - width ?z -orientation)
 :precondition (and (unscheduled ?x)
                (idlePuncher)
 (isCold ?x)
 (punchHasBit ?y)
 (punchCanOrient ?z))
 :effect (and (busyPuncher)
        (not (idlePuncher))
      (scheduledPuncher ?x)
      (not (unscheduled ?x))
      (hasHole ?x ?y ?z)
      (isRough ?x)
      (not (isPolished ?x))
      (not (isSmooth ?x)))
)
(:action doDrillPress
 :parameters (?x - part ?y - width ?z - orientation)
 :precondition (and (unscheduled ?x)
```

```
                (idleDrillPress)
 (drillPressHasBit ?y)
 (drillPressCanOrient ?z)
 (isCold ?x))
 :effect (and (busyDrillPress)
        (not (idleDrillPress))
      (scheduledDrillPress ?x)
      (not (unscheduled ?x))
      (hasHole ?x ?y ?z))
)
(:action doSprayPaint
 :parameters (?x - part ?y - color ?z - color)
 :precondition (and (unscheduled ?x)
                (idleSprayer)
 (isPainted ?x ?z)
 (isCold ?x))
 :effect (and (scheduledSprayer ?x)
        (not (unscheduled ?x))
      (busySprayer)
      (not (idleSprayer))
      (not (isSmooth ?x))
      (not (isRough ?x))
      (not (isPolished ?x))
      (not (isPainted ?x ?z))
      (not (isUnpainted ?x))
      (isPainted ?x ?y))
)
(:action doImmersionPaint
 :parameters (?x - part ?y - color ?z - color)
 :precondition (and (unscheduled ?x)
                (idleImmerser)
 (immerserHasPaint ?y)
 (isPainted ?x ?z))
 :effect (and (busyImmerser)
        (not (idleImmerser))
      (scheduledImmerser ?x)
      (not (unscheduled ?x))
      (isPainted ?x ?y)
      (not (isUnpainted ?x))
      (not (isPainted ?x ?z)))
)
(:action doTimeStepPolisher
 :parameters (?x - part)
 :precondition (scheduledPolisher ?x)
 :effect (and (not (scheduledPolisher ?x))
            (unscheduled ?x)
```

```
   (idlePolisher)
   (not (busyPolisher)))
)
(:action doTimeStepRoller
 :parameters (?x - part)
 :precondition (scheduledRoller ?x)
 :effect (and (not (scheduledRoller ?x))
              (unscheduled ?x)
   (idleRoller)
   (not (busyRoller)))
)
(:action doTimeStepLathe
 :parameters (?x - part)
 :precondition (scheduledLathe ?x)
 :effect (and (not (scheduledLathe ?x))
              (unscheduled ?x)
   (idleLathe)
   (not (busyLathe)))
)
(:action doTimeStepGrinder
 :parameters (?x - part)
 :precondition (scheduledGrinder ?x)
 :effect (and (not (scheduledGrinder ?x))
              (unscheduled ?x)
   (idleGrinder)
   (not (busyGrinder)))
)
(:action doTimeStepPuncher
 :parameters (?x - part)
 :precondition (scheduledPuncher ?x)
 :effect (and (not (scheduledPuncher ?x))
              (unscheduled ?x)
   (idlePuncher)
   (not (busyPuncher)))
)
(:action doTimeStepDrillPress
 :parameters (?x - part)
 :precondition (scheduledDrillPress ?x)
 :effect (and (not (scheduledDrillPress ?x))
              (unscheduled ?x)
   (idleDrillPress)
   (not (busyDrillPress)))
)
(:action doTimeStepSprayer
 :parameters (?x - part)
 :precondition (scheduledSprayer ?x)
```

```
 :effect (and (not (scheduledSprayer ?x))
              (unscheduled ?x)
   (idleSprayer)
   (not (busySprayer)))
)
(:action doTimeStepImmerser
 :parameters (?x - part)
 :precondition (scheduledImmerser ?x)
 :effect (and (not (scheduledImmerser ?x))
              (unscheduled ?x)
   (idleImmerser)
   (not (busyImmerser)))
)
)
```

Appendix I

Logistics Domain

```
(define (domain logistics-adl)
  (:types physobj - object
  obj - physobj
  vehicle - physobj
  truck -vehicle
  airplane - vehicle
  location - object
  city - object
  airport - location)
  (:predicates (at ?x - physobj ?l - location)
       (in ?x - obj ?t - vehicle)
       (in-city ?l - location ?c - city))
  (:action load
     :parameters (?obj - obj ?veh - vehicle ?loc - location)
     :precondition (and (at ?obj ?loc)
(at ?veh ?loc))
     :effect (and (in ?obj ?veh)
  (not (at ?obj ?loc)))
)
  (:action unload
     :parameters (?obj - obj ?veh - vehicle ?loc - location)
     :precondition (and (in ?obj ?veh) (at ?veh ?loc))
     :effect (and (not (in ?obj ?veh))
  (at ?obj ?loc))
)
  (:action drive-truck
   :parameters (?truck - truck ?from - location ?to - location
?city - city)
     :precondition (and (at ?truck ?from)
```

```
 (in-city ?from ?city)
 (in-city ?to ?city))
     :effect  (and (at ?truck ?to)
    (not (at ?truck ?from)))
)
  (:action fly-airplane
   :parameters (?plane - airplane ?from - airport ?to - airport)
   :precondition (and (at ?plane ?from) )
   :effect  (and (at ?plane ?to)
      (not (at ?plane ?from)))
)
)
```

Appendix J

Elevator Domain

```
(define (domain elevator)
(:types passenger - object
floor - object)
(:predicates (at ?x - passenger ?y - floor)
      (atLift ?x - floor)
(inLift ?x - passenger)
)
(:action board
 :parameters (?x - passenger ?y - floor)
 :precondition (and (at ?x ?y)
              (atLift ?y) )
 :effect (and (not (at ?x ?y))
        (inLift ?x) )
)
(:action depart
 :parameters (?x - passenger ?y - floor)
 :precondition (and (atLift ?y)
              (inLift ?x) )
 :effect (and (at ?x ?y)
        (not (inLift ?x)) )
)
(:action moveLift
 :parameters (?x - floor ?y - floor)
 :precondition (atLift ?x)
 :effect (and (atLift ?y)
        (not (atLift ?x)) )
)
)
```

Appendix K

Briefcase Domain

```
(define (domain briefcase)
(:types obj - object
    loc - object)
(:predicates (atBriefcase ?x - loc)
         (at ?x - obj ?y -loc)
   (inCase ?x - obj)
 )
(:action putIn
 :parameters (?x - obj ?y - loc)
 :precondition (and (at ?x ?y)
               (atBriefcase ?y) )
 :effect (and (not (at ?x ?y))
        (inCase ?x) )
)
(:action moveBriefcase
 :parameters (?x - loc ?y - loc)
 :precondition (atBriefcase ?x)
 :effect (and (not (atBriefcase ?x))
        (atBriefcase ?y))
)
(:action takeOut
 :parameters (?x - obj ?y - loc)
 :precondition (and (atBriefcase ?y)
                 (inCase ?x))
 :effect (and (at ?x ?y)
        (not (inCase ?x)) )
)
)
```

Bibliography

[Anderson 87] Charles W. Anderson. *Strategy Learning with Multilayer Connectionist Representations*. In Proceedings of the Fourth International Workshop on Machine Learning, pages 103–114, Irvine, California, 1987. Morgan Kaufmann.

[Anderson 98] Corin R. Anderson, David E. Smith & Daniel S. Weld. *Conditional Effects in Graphplan*. In Reid Simmons, Manuela Veloso & Steven Smith, editeurs, Proceedings of the fourth international conference on Artificial Intelligence Planning Systems (AIPS-98), pages 44–53, Pittsburgh, PA, June 1998. AAAI Press.

[Bäckström 93] Christer Bäckström. *Finding Least Constrained Plans and Optimal Parallel Executions is Harder than We thought*. In Christer Bäckström & Erik Sandewall, editeurs, Current Trends in AI Planning: Second European Workshop on Planning (EWSP-93), Frontiers in AI and Applications, pages 46–59, Vadstena, Sweden, Dec 1993. IOS Press.

[Bauer 79] M. Bauer. *Programming by Examples*. Artificial Intelligence, vol. 12, pages 1–21, 1979.

[Bergmann 92] Ralph Bergmann. *Knowledge Acquisition by Generating Skeletal Plans from Real World Cases*. In F. Schmalhofer, G. Strube & T. Wetter, editeurs, Contemporary Knowledge Engineering and Cognition, pages 125–133, 1992.

[Blum 97] Avrim Blum & Merrick Furst. *Fast planning through planning graph analysis*. Artificial Intelligence, vol. 90, pages 281–300, 1997.

[Carbonell 90] Jaime G. Carbonell & Yolanda Gil. *Learning by Experimentation: The Operator Refinement Method*. In R. S. Michalski & Y. Kodratoff,

editeurs, Machine Learning: An Artificial Intelligence Approach, Volume III, pages 191–213. Morgan Kaufmann, Palo Alto, CA, 1990.

[Cimatti 98a] A. Cimatti, M. Roveri & P. Traverso. *Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains*. In Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98), pages 875–881. AAAI Press, 1998.

[Cimatti 98b] A. Cimatti, M. Roveri & P. Traverso. *Strong planning in Non-Deterministic Domains via Model Checking*. In Proceedings of the 4th International Conference on Artificial Intelligence Planning System (AIPS'98), pages 36–43. AAAI Press, 1998.

[Driessens 01] Kurt Driessens. *Relational Reinforcement Learning*. In Michael Luck, Vladimír Marík, Olga Stepánková & Robert Trappl, editeurs, Multi-Agent Systems and Applications, 9th ECCAI Advanced Course ACAI 2001 and Agent Link's 3rd European Agent Systems Summer School (EASSS 2001), volume 2086 of *Lecture Notes in Computer Science*, pages 271–280, Prague, Czech Republic, July 2001. Springer.

[Džeroski 98] Sašo Džeroski, Luc De Raedt & Hendrik Blockeel. *Relational Reinforcement Learning*. In Proceedings of the International Workshop on Inductive Logic Programming, pages 11–22, 1998.

[Etzioni 93] Oren Etzioni. *Acquiring search-control knowledge via static analysis*. Artificial Intelligence, vol. 62, no. 2, pages 255–302, August 1993.

[Fern 06] Alan Fern, Sungwook Yoon & Robert Givan. *Approximate Policy Iteration with a Policy Language Bias: Solving Relational Markov Decision Processes*. Journal of Artificial Intelligence Research, vol. 25, pages 75–118, January 2006.

[Fikes 71] Richard Fikes & Nils J. Nilsson. *STRIPS: a new approach to the application of theorem proving to problem solving*. Artificial Intelligence, vol. 2, no. 3-4, pages 189–208, 1971.

[Fikes 72] Richard E. Fikes, Peter E. Hart & Nils J. Nilsson. *Learning and Executing Generalized Robot Plans*. Artificial Intelligence, vol. 3, no. 4, pages 251–288, 1972.

[Friedland 85] Peter E. Friedland & Yumi Iwasaki. *The Concept and Implementation of Skeletal Plans*. Journal of Automated Reasoning, vol. 1, no. 2, pages 161–208, 1985.

[Hammond 96] Kristian J. Hammond. *CHEF: A Model of Case-Based Planning*. In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), pages 261–271. American Association for Artificial Intelligence, 1996.

[Hoffman 03] Jorg Hoffman. Utilizing problem structure in planning, volume 2854/2003 of *Lecture Notes in Computer Science*, chapitre Chapter 6: The AIPS-2000 Competition, pages 107–112. Springer Berlin/Heidelberg, 2003.

[Jensen 00] R. M. Jensen & M. M. Veloso. *OBDD-based Universal Planning for Synchronized Agents in Non-Deterministic Domains*. Journal of Artificial Intelligence Research, vol. 13, pages 189–226, 2000.

[Kambhampati 89] Subbarao Kambhampati. *Flexible Reuse and Modification in Hierarchical Planning: A Validation Structure Based Approach*. PhD thesis, University of Maryland, College Park, MD, October 1989.

[Kambhampati 92] Subbarao Kambhampati & James A. Hendler. *A validation-structure-based theory of plan modification and reuse*. Artificial Intelligence, vol. 55, no. 2-3, pages 193–258, June 1992.

[Kambhampati 94] Subbarao Kambhampati & Smadar Kedar. *A unified framework for explanation-based generalization of partially ordered and partially instantiated plans*. Artificial Intelligence, vol. 67, no. 1, pages 29–70, 1994.

[Katukam 94] Suresh Katukam & Subbarao Kambhampati. *Learning Explanation-Based Search Control Rules for Partial Order Planning*. In Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-94), volume 1, pages 582–587, 1994.

[Kautz 86] Henry A. Kautz & James F. Allen. *Generalized Plan Recognition*. In Proceedings of the fifth National Conference on Artificial Intelligence (AAAI-86), pages 32–37, Philadelphia, PA, August 1986. AAAI press, Menlo Park, CA.

[Khardon 99] Roni Khardon. *Learning Action Strategies for Planning Domains*. Artificial Intelligence, vol. 113, no. 1-2, pages 125–148, 1999.

[Knoblock 90] Craig A. Knoblock. *Learning Abstraction Hierarchies for Problem Solving*. In Thomas Dietterich & William Swartout, editeurs, Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90), Menlo Park, California, 1990. AAAI Press.

[Knoblock 94] Craig A. Knoblock. *Automatically Generating Abstractions for Planning*. Artificial Intelligence, vol. 68, no. 2, pages 243–302, 1994.

[Korf 85] Richard E. Korf. *Macro-Operators: A Weak Method for Learning*. Artificial Intelligence, vol. 26, no. 1, pages 35–78, April 1985.

[Lau 01] Tessa Lau. *Programming by Demonstration: a Machine Learning Approach*. PhD thesis, University of Washington, Seattle, 2001.

[Leake 96] David B. Leake, editeur. Case-based reasoning: experiences, lessons, and future directions. AAAI Press/The MIT Press, May 1996.

[Manna 92] Zohar Manna & Richard Waldinger. *Fundamentals of Deductive Program Synthesis*. IEEE Transactions on Software Engeineering, vol. 18, no. 8, pages 674–704, August 1992.

[McDermott 00] Drew McDermott. *The 1998 AI Planning Systems Competition*. AI Magazine, vol. 21, no. 2, pages 35–55, Summer 2000.

[Minton 85] Steven Minton. *Selectively Generalizing Plans for Problem-Solving*. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85), pages 596–599, Los Angeles, CA, 1985. Morgan Kaufmann.

[Minton 88a] Steven Minton. Learning effective search control knowledge: An explanation-based approach. Kluwer Academic Publishers, Boston, MA, 1988.

[Minton 88b] Steven Minton. *Learning Effective Search Control Knowledge: An Explanation-Based Approach*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, March 1988.

[Mitchell 97] Tom Mitchell. Machine learning. McGraw Hill, 1997.

[Muggleton 91] Stephen Muggleton. *Inductive Logic Programming*. New Generation Computing, vol. 8, pages 295–318, 1991.

[Muggleton 94] Stephen Muggleton & Luc De Raedt. *Inductive Logic Programming: Theory and Methods*. Journal of Logic Programming, vol. 19/20, pages 629–679, 1994.

[Pednault 86] Edwin Pednault. *Formulating multiagent, dynamic-world problems in the classical planning framework*. In Michael Georgeoff & Amy Lansky, editeurs, Reasoning about actions and plans: Proceedings of the 1986 workshop, pages 47–82, Los Altos, California, 1986. Morgan Kaufmann.

[Pednault 88] Edwin Pednault. *Synthesizing plans that contain actions with context-dependent effects*. Computational Intelligence, vol. 4, no. 4, pages 356–372, 1988.

[Penberthy 92] J. Scott Penberthy & Daniel Weld. *UCPOP: A sound, complete, partial-order planner for ADL.* In Bernhard Nebel, Charles Rich & William Swartout, editeurs, proceedings of the third international conference on knowledge representation and reasoning (KR-92), pages 103–114, Cambridge, MA, October 1992. Morgan Kaufmann.

[Pyeatt 98] Larry D. Pyeatt & Adele E. Howe. *Decision Tree Function Approximation in Reinforcement Learning*. Rapport technique CS-98-112, Colorado State University, Fort Collins, Colorado, 1998.

[Regnier 91] Pierre Regnier & Bernard Fade. *Complete determination of parallel actions and temporal optimization in linear plans of action*. In Joachim Hertzberg, editeur, European Workshop on Planning, volume 522 of *Lecture Notes in Artificial Intelligence*, pages 100–111. Springer-Verlag, Sankt Augustin, Germany, March 1991.

[Rich 92] Charles Rich & Richard C. Waters. *Approaches to Automatic Programming*. Rapport technique 92-04, Mitsubishi Electric Research Laboratories Cambridge Research Center, Cambridge, Massachusetts, July 1992.

[Sacerdoti 74] Earl D. Sacerdoti. *Planning in a Hierarchy of Abstraction Spaces*. Artificial Intelligence, vol. 5, no. 2, pages 115–135, 1974.

[Schmid 00] Ute Schmid & Fritz Wysotzki. *Applying Inductive Program Synthesis to Macro Learning*. In Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000), pages 371–378, Breckenridge, Colorado, April 2000.

[Schmid 01] Ute Schmid. *Inductive Synthesis of Functional Programs*. PhD thesis, Technische Universität Berlin, Berlin, Germany, May 2001.

[Schoppers 87] Marcel J. Schoppers. *Universal Plans for Reactive Robots in Unpredictable Environments*. In Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-1987), pages 1039–1046, Milan, Italy, 1987.

[Shavlik 90] Jude W. Shavlik. *Acquiring recursive and iterative concepts with explanation-based learning*. Machine Learning, vol. 5, pages 39–50, 1990.

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

[Smith 91] D. R. Smith. *KIDS: A knowledge-based software development system*. In M. R. Lowry & R. D. McCartney, editeurs, Automating Software Design. AAAI press, 1991.

[Smith 93] David E. Smith & Mark A. Peot. *Postponing Threats in Partial-Order Planning*. In Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93), pages 500–507, Washington, D.C., 1993. AAAI Press/MIT Press.

[Smith 96] David E. Smith & Mark A. Peot. *Suspending Recursion in Causal-Link Planning*. In B. Drabble, editeur, Proceedings of the third international conference on Artificial Intelligence Planning Systems (AIPS-96, pages 182–190, Edinburgh, Scotland, May 1996.

[Stefik 81] Mark Stefik. *Planning and Metaplanning*. In Nils J. Nilsson & Bonnie Lynn Webber, editeurs, Readings in Artificial Intelligence, pages 272–286. Tioga Publishing, Palo Alto, CA, 1981.

[Uther 00] William T. B. Uther & Manuela Veloso. *The Lumberjack Algorithm for Learning Linked Decision Forests*. In Symposium on Abstraction, Reformulation and Approximation (SARA-2000), Lecture Notes on Artificial Intelligence, pages 219–232. Springer Verlag, 2000.

[Veloso 90] Manuela Veloso, Alicia Pérez & Jaime Carbonell. *Nonlinear planning with parallel resource allocation*. In Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control, pages 207–212, San Diego, CA, November 1990. Morgan Kaufmann.

[Veloso 94a] Manuela M. Veloso. Planning and learning by analogical reasoning. Springer Verlag, December 1994.

[Veloso 94b] Manuela M. Veloso. *PRODIGY/ANALOGY: Analogical Reasoning in General Problem Solving*. In S. Wess, K.-D. Althoff & M. Richter, editeurs, Topics on Case-Based Reasoning, pages 33–50. Springer Verlag, 1994.

[Weld 94] Daniel Weld. *An Introduction to Least Commitment Planning*. AI Magazine, vol. 15, no. 4, pages 27–61, Winter 1994.

[Wilensky 81] Robert Wilensky. *A model for planning in complex situations*. Cognition and Brain Theory, vol. IV, no. 4, Fall 1981.

[Williams 88] Robert S. Williams. *Learning to program by examining and modifying cases*. In John Laird, editeur, Proceedings of the fifth international conference on machine learning (ICML-88). Morgan Kaufmann, 1988.

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

[Winner 03] Elly Winner & Manuela Veloso. *DISTILL: Learning Domain-Specific Planners by Example*. In Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington, D.C., August 2003.

[Winograd 72] Terry Winograd. Understanding natural language. Academic Press, 1972.