DISTILL: Automated Generation of Domain-Specific Planners

Elly Winner and **Manuela Veloso**
Computer Science Department,Carnegie Mellon University,
5000 Forbes Avenue, Pittsburgh,PA 15213-3891, USA
{elly,mmv}@cs.cmu.edu

Abstract

Domain-independent general-purpose planning have focused on reducing the search involved in an existing general-purpose planning algorithm. 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 planning, but the retrieval and adaptation mechanisms were still domain-independent and efficiency issues were still a concern. We are instead pursuing research in which In this paper, we present introduce an algorithm to create repeating processes, in the form of domain-specific planning programs, from example plans. I will investigate the application of these learned programs to modelling agent preferences and choices. I will also investigate how the programs can be used, extended, and repaired dynamically as an agent encounters new problems and acquires new experience. Finally, I will compare the template-based planning paradigm to existing general-purpose and domain-specific planning programs with a full evaluation on new and existing planning domains. I expect the core contribution of this thesis to be a new planning paradigm in which domain-specific planners are learned by example.

Introduction

Motivation

Intelligent agents must be able to autonomously develop and execute 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 weeks of work and great specific knowledge of the details of the underlying domain-independent planner for humans to formalize useful rules.

In my previous work, I have shown that example plans can reveal more than control information: they can also reveal the process behind their generation. I have introduced SPRAWL, an algorithm for finding the rationale underlying observed example plans. I have also introduced DISTILL, an algorithm for extracting simple non-repeating programs, or templates, from example plans supplemented with their rationales. For my thesis, I will extend these techniques to extract rich algorithmic models of behavior from observed executions by developing methods of identifying and extracting complex repeated structures in example plan bases.

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. My work on extracting algorithmic models of behavior from observed excutions 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 in minutes, simply by watching it be done. It could facilitate the cooperation of heterogenous 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.

Approach

There are three main steps towards achieving this goal. The first is to find the rationale underlying example plans by identifying and explaining the choices made by the agent in constructing the plan. The second step is to construct a domain-specific planner that reflects the choices and preferences observed in the example plans and identified in the rationale. The third step is to use this domain-specific planner to predict behavior or to solve new problems, and to extend or repair it in the presence of unsolved problems or new example plans.

Identifying Underlying Rationales In order to predict and duplicate the behavior patterns of an observed agent, we must understand how the actions chosen by that agent help to achieve its goals. I call this understanding the *rationale* behind the plan. Previous approaches to plan analysis focus either on annotating plans or on finding partially ordered plans. Approaches to annotating orderings have been unable to handle domains with conditional effects or have relied on the underlying planner to provide the annotation, making them unable to analyze observed plans. Some approaches to partial ordering have explored finding a partial ordering of a totally ordered plan, though most deal with finding a partially ordered plan from scratch. While building on this previous work, we focus instead on identifying and explaining the choices made in creating an observed plan.

In Section , I present an algorithm that reveals the rationale underlying observed plans with conditional effects in the form of an annotated partial ordering. The partial ordering reflects the ordering constraints in the plan, helping to identify independent tracks of execution. The ordering constraints are annotated with the reasons they are necessary, allowing subplans to be matched and extracted from complete plans in a principled way (**?**).

Extracting Domain-Specific Planners A principle difficulty in learning from example plans and in agent modelling is using the choices and preferences reflected in observed plans to construct a theory of behavior that can then be used to imitate or predict the actions of an observed planner. Previous approaches fall into two categories: those that extract control knowledge from the example plans that is then used to guide later planning search and those that compile exhaustive databases of observed plans and predict or generate actions by matching those databases.

I propose to develop learning techniques that avoid the cost of planning from scratch and of maintaining exhaustive databases by compiling observed example plans into compact domain-specific planners which are able to duplicate the behavior shown in the example plans and to solve new problems based on that behavior. In Section , I present an algorithm that moves towards this goal. The algorithm extracts simple, non-looping programs, or templates, from example plans supplemented with their rationales. I show that these templates, while storing little information, succeed in capturing observed behavior and in solving many new problems. In fact, templates extracted from only a few example plans are able to solve all problems in limited domains (**?**).

Using and Maintaining Domain-Specific Planners Because my work will focus on extracting domain-specific planners from example plans, it will also have to address how to use and maintain planners which may be incomplete and incorrect. An incomplete planner can result when the example plans given to the learning algorithm cause it to learn a program that cannot solve all problems in a given domain. This will become evident when the planner is given a problem not solvable by its current program. An incorrect planner may result when the learning algorithm optimistically identifies loops in the observed example plans, or identifies loop conditions incorrectly. The planner will be revealed to be incorrect when subsequent execution on a new problem identifies non-executable steps in the loop.

Incomplete planners can be extended to cover new problems by querying an external generative planner for the solutions to the new problems and then incorporating the resulting example plans into the domain-specific planner, either by extracting a new domain specific planner from the updated database of solutions or by online modification. Extracting a new planner from scratch could allow the learning algorithm to develop a better (more compact, more general) program, but online modification would save the learning algorithm the substantial cost of maintaining a database of observed plans and of generating the new planner from scratch each time it is found to be incomplete.

It is not clear how to repair incorrect planners. One possibility is to remove loops found to be incorrect. Another possiblity is to add new conditions to the loops to better characterize their looping and stopping conditions. The problem may even be solvable simply by incorporating the new, troublesome plans into the domain-specific planner. I propose to investigate different methods for repairing incorrect planners and for generating planners unlikely to be incorrect.

Expected Contributions

1. A domain-independent algorithm for extracting the rationale behind an observed plan.
2. An algorithm for extracting domain-specific planners from example plans.
3. An algorithm for repairing or extending domain-specific planners in the presence of new example plans.

Background

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.

Triangle Tables Triangle tables are one of the earliest forms of annotation (?). 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) (?). 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.

Validation Structures Another powerful approach to annotation is the validation structure (?; ?; ?). 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 (?) and for explanation-based generalization of partially ordered and partially instantiated plans (?). The approach has not been applied to plans with conditional effects. Although (?) 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.

Derivational Analogy Derivational analogy (?) 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.

Operator Graphs The final approach to annotation that we will discuss is the operator graph (?; ?). 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 preconditions 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 (?) and to analyze potential recursion (?). 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.

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 (?) 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 (?). 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 (?).

Partial Order Planning There has been a great deal of research into generating partially ordered plans from scratch. UCPOP (?) 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 (?), another well-known planner, is also able to find partially ordered plans in domains with conditional effects (?). 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 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 2. SPRAWL reveals the independence of the two sets of operators.

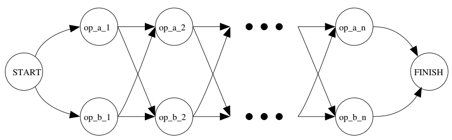

Figure 1: This partial ordering, found by Graphplan, contains many irrelevant ordering constraints.

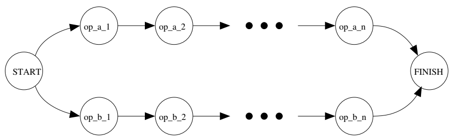

Figure 2: This partial ordering, found by SPRAWL, contains only necessary ordering constraints.

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.

Control Rules Much of the work on learning domain knowledge for planning is focused on learning control rules (**?**; **?**; **?**), 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 (**?**), and computer-learned control rules are often ineffective (**?**). 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 (**?**).

Macro Operators Another common type of learned control knowledge for planning is the macro operator (**?**; **?**), 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 (**?**), but the efficacy of macros has, in general, been limited to hierarchically decomposable domains.

Case-Based Reasoning Another approach to learning planning knowledge, case-based reasoning, attempts to avoid generative planning entirely for many problems (**?**; **?**; **?**). 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 (**?**), 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 (**?**).

Analogical Reasoning A variant of case-based reasoning that deserves mention is analogical reasoning, which also stores case libraries and attempts to modify previous cases to solve new problems (**?**; **?**). 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.

Hierarchical Planning Domain knowledge is also used to hierarchically divide planning domains (?; ?; ?). 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 (?) or may be discovered automatically through an analysis of the planning domain (?; ?; ?).

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 (?). 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 (?).

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) (?), to plan how to resolve goal conflicts or other problems in planning (?), 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) (?).

Automatic Program Generation

A great deal of 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.

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 (?) 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 (?) 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 (?), which learns to generate programs that satisfy specifications by examining and modifying other programs.

Rich and Waters explore many of the difficulties of deductive program synthesis (?). 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.

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 (?; ?). 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 (?; ?), 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 (?), 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 (?; ?; ?; ?). These macros capture repetative 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 (?), 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 (?). Part of the reason it may need so many examples is that by breaking up example plans into state-action pairs, it also disposes of much of the information contained in the plans, such as sequencing and looping behaviors. Also, enumerating all possible conditions and evaluating them against every observed state-action pair may become prohibitively slow for more complex domains than Blocksworld.

Universal Planning

Some researchers have sought to avoid the planning search problem by aquiring and using "universal plans," or precomputed functions that map state and goal combinations to actions. Our work can be seen as a new method of 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 solutions to Markov decision processes, or the Q tables of standard reinforcement learning (?). 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 (?). Q functions may also be represented as decision trees. This format has been found to be very expressive, frequently outperforming neural network representations (?), 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 (?). Relational reinforcement learning provides a method for learning a Q function represented as a logical regression tree over parameterized operators and predicates (?; ?). 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 (?), 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 (?; ?; ?). 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.

Previous Work

Automatically Acquiring Planning Templates from Example Plans

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

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

We introduce the DISTILL algorithm, which automatically extracts these domain-specific planning programs (which we call *templates*) from example plans, and show how to use them to solve planning problems. We call these domain-specific planning programs *templates*. Table 4 shows a simple example template that solves all problems in the gripper domain that involve moving balls from one room to another.

In some domains, finding optimal solutions is NP-complete. Therefore, templates learned automatically from a finite number of example plans cannot be guaranteed to find optimal plans. Our goal is to extend the *solvability horizon* for planning by reducing planning times and allowing much larger problem instances to be solved. We believe that post-processing plans can help improve plan quality.

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

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

Defining Templates A template is a domain-specific planning program that, given a planning problem (initial and goal states) as input, either returns a plan that solves the problem or returns failure, if it cannot do so. Templates are composed of the following programming constructs and planning-specific operators:

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

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

Table 4 shows a template that solves all gripper-domain (**?**) problems involving moving balls between rooms. The template is composed of one while loop: while there is an ball that is not at its goal location, move to the ball (if necessary), pick up the ball, move to goal location of the object, and drop the ball.

Learning Templates: the DISTILL Algorithm The DISTILL algorithm, shown in Table 5, learns templates from sequences of example plans, incrementally adapting the template with each new plan. One benefit of online learning is that it allows a learner with access to a planner to acquire templates on the fly in the course of its regular activity. And because templates are learned from example plans, they reflect the *style* of those plans, thus making them suitable not only for planning, but also for agent modeling.

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

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

Generalizing Situations We make several assumptions about what makes one planning *situation* different than another, and about how the observed planner will solve prob-

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

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

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

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

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

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

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

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

lems. We assume that two objects of the same type will be treated the same by the planner. Thus, two situations are equivalent if they contain the same number and types of objects in the same relationships. We assume that the planner will respond to equivalent situations with the same plan. This allows the DISTILL algorithm to identify common situations that occur in the solutions of several planning problems, and to extract their solutions for independent use in other problems.

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

Next, the parameterized plan is converted into a template, as formalized in the procedure Make_New_If_Statement in Table 5. The conditions on the new if statement are the initial- and goal-state terms that are *relevant* to the plan. Relevant initial-state terms are those which are needed for the plan to run correctly and achieve the goals (**?**). Relevant goal-state terms are those which the plan accomplishes. We use a minimal annotated partial ordering (**?**) of the observed plan to compute which initial- and goal-state terms are relevant. The steps of the example plan compose the body of the new if statement. We store the minimal annotated partial ordering information for use in merging the template into the previously-acquired knowledge base.

Figure 13 shows an example minimal annotated partially ordered plan with conditional effects. Table 6 shows the template DISTILL creates to represent that plan. Note that the conditions on the generated if statement do not include all terms in the initial and goal states of the plan. For example, the template does not require that e(z) be in the initial and goal states of the example plan. This is because the plan steps do not generate e(z), nor do they need it to achieve the goals. Similarly, b(x) and the conditional effects that could generate the term c(x) or prevent its generation are also ignored, since it is not relevant to achieving the goals.

Merging Templates The merging process is formalized in the procedure Add_To_Template in Table 5. The templates learned by the DISTILL algorithm are sequences of non-nested if statements. To merge a new template into its knowledge base, DISTILL searches through each of the if statements already in the template to find one whose body (the solution plan for that problem) matches that of the new problem. We consider two plans to match if:

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

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

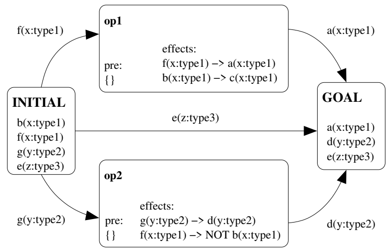

Figure 3: An example plan. The preconditions (pre) are listed, as are the effects, which are represented as conditional effects $a \rightarrow b$, i.e., if a then add b. A non-conditional effect that adds a literal b is then represented as $\{\} \rightarrow b$. Delete effects are represented as negated terms (e.g., $\{a\} \rightarrow NOTb$.

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

We will now describe how to combine two if statement templates, if_1= **if** x **then** abc and if_2= **if** y **then** b, when the body of if_2 is a sub-plan of that of if_1. This process is illustrated in Figure 14. [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 14, merging if_1 and if_2 will result in three new if statements. We will label them if_3, if_4, and if_5. The body of if_3 is set to a and its conditions are $Relevant(x, a)$. The body of if_4 is b and its conditions are $Relevant(x_a, b)$ **or** $Relevant(y, b)$. [3] Finally, the body of if_5 is c and its conditions are $Relevant(x_a b, c)$. Whichever of if_1 or if_2 is already a member of the template is removed and replaced by the three new if statements.

Illustrative Results Table 7 shows a template learned by the DISTILL algorithm that solves all problems in a blocksworld domain with two blocks. There are 555 such problems [4], but the template needs to store only two plan steps, and DISTILL is able to learn it from only 6 example plans.

Table 8 shows a template learned by the DISTILL algorithm to solve all gripper-domain problems with one ball, two rooms, and one robot with one gripper arm. Although there are 1932 such problems, [5] the DISTILL algorithm is able to learn the template from only five example plans. It

[2]Combining two if statements with overlapping bodies is similar. It is illustrated in Figure 15

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

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

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

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

Table 3: The template DISTILL would create to represent the plan shown in Figure 13.

if (x) then
  a
  b
  c
if (y) then
  b

+

if (Relevant(x, a)) then
  a
if (Relevant(x_a, b) or y) then
  b
if (Relevant(x_{ab})) then
  c

Figure 4: Combining two if statements when the body of one is a sub-plan of the body of the other.

if (x) then
  a
  b
if (y) then
  b
  c

+

if (Relevant(x, a)) then
  a
if (Relevant(x_a, b) or
    Relevant(y, b) then
  b
if (Relevant(y_b, c) then
  c

Figure 5: Combining two if statements when their bodies are overlapping.

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

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

successfully generalizes situations within individual plans for use in other plans. Also note that only five plan steps (the length of the longest plan) are stored in the template.

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

Planning with Templates Our algorithm for generating plans from templates is shown in Table 9. As previously mentioned, while executing the template, we must keep track of a current state and of the current solution plan. The current state is initialized to the initial state, and the solution plan is initialized to the empty plan. Executing the template consists of applying each of the statements to the current state. Each statement in the template is either an plan step, an if statement, or a while loop. If the current statement is a plan step, make sure it is applicable, then append it to the solution plan and apply it to the current state. If the current statement is an if statement, check to see whether it applies to the current state. If it does, apply each of the statements in its body; if not, go on to the next statement. If the current statement is a while loop, check to see whether it applies to the current state. If it does, apply each of the statements in its body until the conditions of the loop no longer apply. Then go on to the next statement.

Sometimes there may be many ways to apply an if statement or a while loop to the current state. For example, if we have a statement like, "**if** (**in_current_state** (not-eaten(?a:apple))) **then** eat(?a)", and there are several uneaten apples in the current state, it is unclear which apple should be eaten. However, one of our primary assumptions is that all objects that match the conditions may be treated the same, so, in this case, it doesn't matter which apple is eaten.

Detecting and Handling Failures There are three ways a template may fail to generate the correct solution plan. It may have run through the whole template and found no solution steps at all, though the initial state is not the same as the goal state. Or, it may have found some plan steps to execute, but, by the end of the template, did not reach the goal state. Finally, it may have found some plan steps to execute, but found that they were not applicable to the current state. A failure is detected when we attempt to execute steps that are not applicable in the current state or when the template finishes executing and its final state does not match the goal state. The way we currently handle failures is by handing the problem off to a generative planner, and then to add that new solution to the template.

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

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

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

procedure Apply_Template(T, I, C, G):
    P ← ∅
    for each statement S_n in T do
        P ← P + Apply_Statement(S_n, I, C, G)
    if G is satisfied by C then
        return P
    else
        FAIL

procedure Apply_Statement(S, I, C, G):
    P ← ∅
    if S is an if statement then
        if Applies_Now(S, C, G) then
            for each statement S_i in the body of S do
                P ← P + Apply_Statement(S_i, C, G)
    if S is a while statement then
        while Applies_Now(S, I, C, G) do
            for each statement S_i in the body of S do
                P ← P + Apply_Statement(S_i, C, G)
    if S is a plan step then
        if not Applicable(S, C) then
            FAIL
        C ← Apply_Step(S, C)
        P ← S
    return P
```

Table 6: Template-based plan generation.