Learning to Solve Complex Planning Problems: Finding Useful Auxiliary Problems

Peter Stone and **Manuela Veloso**
School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213-3890
pstone,veloso@cs.cmu.edu

Abstract

Learning from past experience allows a problem solver to increase its solvability horizon from simple to complex problems. For planners, learning involves a training phase during which knowledge is extracted from simple problems. But how are these *simple* problems constructed? All current learning and problem solving systems require the user to provide the training set. However it is rarely easy to identify problems that are both simple and useful for learning, especially in complex applications. In this paper, we present our initial research towards the automated or semi-automated identification of these simple problems. From a difficult problem and a corresponding partially completed search episode, we extract auxiliary problems with which to train the learner. We motivate this overlooked issue, describe our approach, and illustrate it with examples.

Introduction and Problem Formulation

Researchers in Machine Learning and in Planning have developed several systems that use simple planning problems to learn how to solve more difficult problems (among several others, (Laird, Rosenbloom, & Newell 1986; DeJong & Mooney 1986; Minton 1988; Veloso & Carbonell 1993; Borrajo & Veloso 1994)). However, all current systems require that the simpler problems be provided by the user. This requirement is a gap in automated learning that we propose to fill. Using a previously untried approach, we are developing a system that will find *auxiliary problems* that are likely to be useful in learning how to solve a *difficult* problem.

A planning problem can be considered *difficult* for a particular planner if it cannot solve the problem with a reasonable amount of effort. Learning how to solve difficult problems is a three-step process:

- First, the learner must find some simpler, i.e. more directly solvable, problems that are somehow related to the original problem. These problems are called *auxiliary problems* (Polya 1945). This first step is the one which we address here.
- Second, the learner must take these auxiliary problems and learn by solving them. Since they are *simpler* than the original problem, the planner should be able to solve the auxiliary problems with relatively little effort.
- Third, the learner must use the knowledge gained by solving the auxiliary problems to find a solution to the original problem.

This process is successful if the time to carry out all three steps is significantly less than the time it would have taken to solve the original problem without the benefit of learning.

Current learning systems work towards executing the second and third steps of the above process while finessing the first. Training problems from which to learn are usually generated randomly, entered by the user, or defined by a set of rules according to some preset measure of simplicity, such as number of goals, which is not necessarily accurate (Minton 1988; Etzioni 1993; Knoblock 1994; Bhansali 1991; Veloso 1992; Katukam & Kambhampati 1994).

The goal of our research is to provide a general method for finding auxiliary problems that are most likely to help us learn how to approach a difficult problem. To be useful, these auxiliary problems must be solvable with relatively little effort on the part of the planner; otherwise they are no more approachable than the original problem. Yet the auxiliary problems must also retain some complexities, if only on a smaller, more learnable scale. If we simplify the problem too much we will not learn anything relevant to the original problem. Finding auxiliary problems that are neither too complex nor too simple is a difficult task.

We believe that we will be able to accomplish this task because we are considering some available information that previous attempts at this task have ignored. Some of the few attempts at decomposing a problem into *simpler* problems are based on capturing different abstraction levels or problem spaces in the domain definition (Knoblock 1994; Rosenbloom, Newell, & Laird 1990; Sacerdoti 1974). These approaches have one thing in common: a static analysis of the domain and problem, either automated or done by the user. But simply looking at the problem may not provide any information as to what is causing the planner to have difficulty. We propose to examine not only the original problem and domain definition, but also the planner's unsuccessful solution attempt. The partially completed search space from the original problem will help us discover why the problem is so hard *for a particular planner*. In this way, we will be able to determine what aspects of the original problem we most need to learn.

Motivation

> If you cannot solve the proposed problem try to solve first some related problem. Could you imagine a more accessible related problem? . . . A more special problem? . . . Could you solve a part of the problem? (Polya 1945, p.xvii)

This quote is from Polya's 1945 book *How To Solve It*. In this work on heuristic problem solving, Polya lays out several methods in which humans can solve difficult problems. Although computers do not necessarily need to learn in the same way that humans do, the human learning process is a good source of motivation.

For example, Polya considers the following problem: "*Inscribe a square in a given triangle. Two vertices of the square should be on the base of the triangle, the two other vertices of the square on the two other sides of the triangle, one on each.*"(Polya 1945, p.23). Polya considers simplifying this problem by eliminating one of the goals. It is easy to draw a square with three (instead of four) vertices on the triangle: two on the base and one on another edge. Simply draw a line from a point on an edge straight down to the base and finish the square with edges of the same length. Once we have accomplished this simpler task, it is much easier to see how to go about inscribing a square in the triangle.

This method of learning can be seen as *lazy-evaluation learning*: when you find a problem that you cannot solve, practice solving auxiliary problems until you see how to solve the original problem. Rather than doing all the practice problems ahead of time, you can learn only what you need to know. People who use lazy-evaluation learning may not learn their subject matter so thoroughly, but they will also not waste time learning techniques that they will never have to use.

Similarly in planning, and especially in complex real-world applications, lazy-evaluation learning is efficient because you cannot predict ahead of time which problems will be difficult for your planner to solve. When you find a set of tasks that you would like to solve with a planner, you first have to create a domain. In so doing, you are often faced with several decisions about how to represent objects and operators in your domain. These decisions do not affect your conception of the domain, but they may greatly affect the performance of the planning algorithm on different problems. Thus when you set your planner to work on an apparently straightforward problem, you may be surprised to find that the planner does not solve your problem in a straightforward way. The planner may choose the wrong operators to achieve particular goals or it may select unsuccessful instantiations of operators. Whatever the case, you will have created a new situation in which learning is needed. You will need a learning method to find the heuristics that can steer you clear of the exponentially-sized dead-ends in the search space.[1]

If planners could be provided with ways of finding auxiliary problems, then they could learn heuristics that would potentially help them solve a difficult problem. But finding simpler planning problems for planners is not trivial. For example, identifying abstraction levels and potentially useful decompositions by applying algorithms that syntactically analyze domain definitions has been shown to be very much representation dependent. All past considerations of this problem — including Polya's — have tried to find simpler problems by statically analyzing the original problem. In this paper, we propose to use unsuccessful attempts at solving a planning problem to help us gain additional information as to how to find appropriate simplifications.

Approach

We are conducting our research within PRODIGY, an integrated architecture for research in planning and learning (Carbonell, Knoblock, & Minton 1990). Like all planners, it uses heuristics to guide its search through its search space. Although PRODIGY is capable of using a wide variety of different heuristics, there is no collection of heuristics that works efficiently in all domains (Stone, Veloso, & Blythe 1994). The focus of the PRODIGY project has been on understanding how an AI planning system can acquire expertise by using different machine learning strategies. Since several learning modules already exist in PRODIGY, it is ideally suited to our research. Once we create appropriate auxiliary problems, we can hand them to one of these existing modules in order to complete the learning process. Thus we can concern ourselves exclusively with finding auxiliary problems.

Since our task is to find auxiliary problems when PRODIGY comes across a *difficult* problem, we must first define what we consider a difficult problem. Here we have several options. A difficult problem could be:

- one that PRODIGY does not solve in a fixed amount of time or in a fixed number of search steps;
- one for which PRODIGY finds a suboptimal solution;
- one that causes PRODIGY to backtrack a disproportionate number of times;
- or one for which a user is unsatisfied with PRODIGY's behavior for any other reason.

For us, all of these cases are acceptable definitions of difficult problems and we allow room in our system for all of them. In particular, we will allow both PRODIGY and the user to label a problem as *difficult*.

[1] Changing the domain representation is another alternative that has not been explored in automated learning systems. It is widely done by planner developers whose systems do not incorporate learning capabilities; however, learning usually occurs by adding more knowledge to the initial domain specification in order to direct the planner through the search space at planning time. Finding methods for changing the domain representation is a current challenging line of research for learning.

Analyzing the partial search

Once we have a difficult problem, we can begin learning how to solve it. As mentioned earlier, we are trying to find auxiliary problems based not only on the problem description, but also based on the partial search space that results from PRODIGY's attempt at solving the original problem. As such, the size of the partial search space will affect our results. Again, we will try to accommodate different possible approaches. The user could fix the size of the partial search space by bounding the number of search steps or amount of time that PRODIGY may work on the original problem. Another possibility is that this bound could be randomized to create differently sized search spaces. Finally, the user could manually interrupt PRODIGY's search at any time and pass both the problem and the partial search space to our system. All of these possibilities are acceptable. We only require that our system have access to a partial search by PRODIGY of the problem's search space.

Since we use the partial search to extract information, its size can affect its usefulness. For example, a very early interrupt may not give the planner the opportunity to explore a rich enough search space. On the other hand, an excessively long partial search episode, aside from taking a long time, may mislead our learning algorithm in its attempt to extract the relevant information. It may even be the case that different partial search episodes will provide different useful information. We will address this issue by using a generate-and-test technique to exploit different partial search episodes.

PRODIGY's partial search space can provide many clues as to what makes the original problem difficult for our planner. For example, we can determine which subgoals PRODIGY solved and which it did not. Among those that it solved, we can determine which subgoals it solved quickly, and which required a large amount of search. We can also find out if, due to goal or operator interactions, the problem forced PRODIGY to repeatedly solve the same subgoals. Furthermore, we can determine at which points PRODIGY backtracked the most. All of these clues, among others, can provide information as to what auxiliary problems are most likely to be useful for learning how to solve the original problem.

Simplifying the problem

After examining the partial search space, there are several ways in which we can try to simplify the problem.

- The most straightforward way is to reduce the number of goals in the problem: if a difficult problem has five goals to solve, a problem with just three of them is likely to be simpler.
- Another way to simplify a problem is to use subsets of troublesome subgoals as the goals of the auxiliary problem. For example, consider the situation where we determine that PRODIGY has a hard time solving a goal G_1. If G_1 can be achieved by an operator with preconditions $S_1, S_2, \ldots, S_k$, then candidate auxiliary problems will be ones with different combinations of the preconditions S_1 through S_k satisfied in the initial state. The remaining preconditions will be new goals in the auxiliary problems.
- Some other possible ways of simplifying a problem are changing the order of its goals, changing the set of relevant domain operators, changing the objects available as possible bindings, and changing the initial state in a variety of ways. Note that changing a problem could mean either reducing its size or adding information to it.

The aim of our research is to find a general method for using the clues from the search space to choose one or more of these options to create useful auxiliary problems. Currently our system is in the early implementation stages. Table 1 shows a high-level view of the algorithm.

1. Try to solve the difficult problem.
2. While PRODIGY does not solve the problem,
   - (a) Determine from the search space which goals and subgoals PRODIGY solved easily, which it solved only after some backtracking, and which it did not solve at all.
     - If there were several unsolved goals, then suggest trying auxiliary problems with fewer goals.
     - If PRODIGY solved each goal at some point, then output auxiliary problems with different goal orderings.
   - (b) Determine from the search space at which type of choice point PRODIGY backtracked the most. If it backtracked most at . . .
     - subgoal choice points, suggest auxiliary problems with fewer goals.
     - operator choice points, suggest auxiliary problems with different goals or different initial states.
     - binding choice points, suggest auxiliary problems with fewer objects.
   - (c) Choose a limited number of the suggested auxiliary problems for training.
   - (d) Train PRODIGY on the chosen auxiliary problems.
   - (e) Use the newly-learned heuristics to try to solve the difficult problem.

Table 1: Generating and learning from simple problems.

As our research proceeds, our system will be able to use clues from the search space in more sophisticated ways as well.

Example

To understand our task in a real-world context, consider an extended version of the rocket domain introduced in (Veloso & Carbonell 1993). This domain is a simple instance of a real world class of transportation domains where there is a limited number of resources that are consumed and can or cannot be reachieved while planning. In this particular version of the domain, fuel is an unrenewable resource which prohibits rockets from moving more than once. The domain includes objects which can be loaded in and out of rockets, and locations to which rockets can fly. One possible representation of the operators is as follows:

- (Load-Rocket <object> <rocket>) changes the location of <object> to be (inside <rocket>). <object> and <rocket> must be in the same location.

```
(Operator
 Load-Rocket
  (params <object> <rocket>)
  (preconds
   ((<object> OBJECT)
    (<o-place> ORIGIN)
    (<rocket> ROCKET))
   (and (at <rocket> <o-place>)
        (at <object> <o-place>)))
  (effects
   ()
   ((del (at <object> <o-place>))
    (add (inside <object> <rocket>)))))
```

- (Unload-Rocket <object> <rocket>) changes the location of <object> to be (at <location>). <rocket> must be at <location> and <object> must be in <rocket>).

```
(Operator
 Unload-Rocket
  (params <object> <d-place> <rocket>)
  (preconds
   ((<object> OBJECT)
    (<d-place> DESTINATION)
    (<rocket> ROCKET))
   (and (inside <object> <rocket>)
        (at <rocket> <d-place>)))
  (effects
   ()
   ((del (inside <object> <rocket>))
    (add (at <object> <d-place>)))))
```

- (Move-Rocket <rocket> <location- 1> <location- 2>) changes the location of <rocket> from <location- 1> to <location- 2>. <rocket> must have fuel to be moved and this operator consumes the rocket's fuel. Note that a rocket cannot be refueled as there is no operator that adds fuel.

```
(Operator
 Move-Rocket
  (params <rocket> <loc-from> <loc-to>)
  (preconds
   ((<rocket> ROCKET)
    (<loc-from> LOCATION)
    (<loc-to>
     (and LOCATION
          (diff <loc-to> <loc-from>))))
   (and (has-fuel <rocket>)
        (at <rocket> <loc-from>)))
  (effects
   ()
   ((del (at <rocket> <loc-from>))
    (add (at <rocket> <loc-to>))
    (del (has-fuel <rocket>)))))
```

Problems in this domain consist of trying to move objects from their initial locations to specific destinations.

When given this domain representation, PRODIGY has a difficult time with some apparently simple problems. For example, consider the following problem with five objects and two rockets:

Initial State	Goal State
(at objA Pittsburgh)	(at objA London)
(at objB Pittsburgh)	(at objB London)
(at objC Pittsburgh)	(at objC London)
(at objD Pittsburgh)	(at objD Tokyo)
(at objE Pittsburgh)	(at objE Tokyo)
(at rocket1 Pittsburgh)	
(at rocket2 Pittsburgh)	

An Optimal Solution

<Load-Rocket objA rocket1>
<Load-Rocket objB rocket1>
<Load-Rocket objC rocket1>
<Move-Rocket rocket1 Pittsburgh London>
<Unload-Rocket objA rocket1>
<Unload-Rocket objB rocket1>
<Unload-Rocket objC rocket1>
<Load-Rocket objD rocket2>
<Load-Rocket objE rocket2>
<Move-Rocket rocket2 Pittsburgh Tokyo>
<Unload-Rocket objD rocket2>
<Unload-Rocket objE rocket2>

PRODIGY does not directly find this solution when searching with its default heuristics. There is no explicit information in the domain telling it to send one rocket to London and one to Tokyo. PRODIGY also does not realize to load all the objects that are going to the same destination before flying the rocket. We could use heuristics that guide PRODIGY straight to a solution, but we explore the use of learning algorithms rather than relying on the user to specify the way in which the planner should search for a solution.

What are some good auxiliary problems for this problem? Recall that one possible way of creating auxiliary problems is to reduce the number of goals. However, it is not always obvious which goals to eliminate. In this case, considering a problem with goal state (and (at objA London) (at objD Tokyo) (at objE Tokyo)) is indeed likely to provide useful information for solving our problem (see Table 2). On the other hand, an auxiliary problem with just one goal is not likely to help. Since a problem with one goal is directly solved by PRODIGY, it is too simple to lead to useful learning. At the other extreme, a problem with four goals in the goal state is almost as difficult as the original problem. If able to find a solution, PRODIGY could certainly use it to learn how to solve the original problem; however, if PRODIGY cannot find a solution in a relatively short amount of time, then the auxiliary problem is not useful for learning.

This problem also has useful auxiliary problems that do not involve reducing the number of goals. One example is the auxiliary problem whose initial state includes having objA already loaded into rocket1 with objD and objE loaded into rocket2. As before, there is the danger of simplifying the problem too much or too little. Starting with

Goal State	Solution Found?	Time (sec)
(at objA London)	yes	.45
(at objA London) (at objD Tokyo) (at objE Tokyo)	yes	9.7
(at objA London) (at objB London) (at objD Tokyo) (at objE Tokyo)	no	500

Table 2: PRODIGY's performance on problems with fewer goals than the original problem. The problem with just one goal is solved almost immediately: there is no backtracking from which to learn. At the other extreme, the problem with four goals is not solved in a reasonable amount of time. It is the problem with three goals that is most likely to be useful for learning. This problem is suggested as a good auxiliary problem by an examination of a partial search (one hundred seconds in duration) of the original problem: (at objA London), (at objD Tokyo), and (at objE Tokyo) are achieved at some point during the partial search while the other two goals are not.

all five objects in the correct rockets renders the problem trivial, whereas starting with only one object loaded does not simplify the problem enough.

The partial searches of this rocket problem leave many other options open for possible auxiliary problems. But rather than listing more, we would like to emphasize the phenomenon illustrated in the previous two paragraphs: *finding useful auxiliary problems is not trivial*. Of the three auxiliary problems which were created by removing goals from the original problem, one is too simple to be useful, one is too difficult to be useful, and only one may possibly be useful for learning since it is easily but not trivially solvable. Similarly, problems with slightly altered initial states can be too simple, too difficult, or possibly useful. Our system will identify the auxiliary problems that are most likely to be useful and then use them to reduce the total time necessary to solve complex problems.

Conclusion

Learning is absolutely necessary for solving complex planning problems, especially in real-world domains in which domain representation issues can make planning difficult. Since seemingly simple problems can turn out to be quite difficult for a planner to solve, we need a method of learning heuristics to solve a particular problem. Current learning systems overlook the difficult task of finding auxiliary problems on which to train. Our approach is to generate auxiliary problems by analyzing not only the problem and domain specifications, but also the planner's unsuccessful attempt at solving the problem. By generating the auxiliary problems and learning from them, we will efficiently solve complex planning problems.

References

Bhansali, S. 1991. *Domain-based program synthesis using planning and derivational analogy*. Ph.D. Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign.

Borrajo, D., and Veloso, M. 1994. Incremental learning of control knowledge for nonlinear problem solving. In *Proceedings of the European Conference on Machine Learning, ECML-94*, 64–82. Springer Verlag.

Carbonell, J. G.; Knoblock, C. A.; and Minton, S. 1990. Prodigy: An integrated architecture for planning and learning. In VanLehn, K., ed., *Architectures for Intelligence*. Hillsdale, NJ: Erlbaum. Also Technical Report CMU-CS-89-189.

DeJong, G. F., and Mooney, R. 1986. Explanation-based learning: An alternative view. *Machine Learning* 1(2):145–176.

Etzioni, O. 1993. Acquiring search-control knowledge via static analysis. *Artificial Intelligence* 62(2):255–301.

Katukam, S., and Kambhampati, S. 1994. Learning explanation-based search control rules for partial order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 582–587.

Knoblock, C. A. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* 68.

Laird, J. E.; Rosenbloom, P. S.; and Newell, A. 1986. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning* 1:11–46.

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

Polya, G. 1945. *How to Solve It*. Princeton, NJ: Princeton University Press.

Rosenbloom, P. S.; Newell, A.; and Laird, J. E. 1990. Towards the knowledge level in SOAR: The role of the architecture in the use of knowledge. In VanLehn, K., ed., *Architectures for Intelligence*. Hillsdale, NJ: Erlbaum.

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

Stone, P.; Veloso, M.; and Blythe, J. 1994. The need for different domain-independent heuristics. In *Proceedings of the Second International Conference on AI Planning Systems*, 164–169.

Veloso, M. M., and Carbonell, J. G. 1993. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning* 10:249–278.

Veloso, M. M. 1992. *Learning by Analogical Reasoning in General Problem Solving*. Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Available as technical report CMU-CS-92-174. A revised version of this manuscript will be published by Springer Verlag, 1994.