Using Multi-Strategy Learning to Improve Planning Efficiency and Quality

Tara Adrienne Estlin

Report AI98-269 May 1998

estlin@cs.utexas.edu
http://www.cs.utexas.edu/users/estlin/

Artificial Intelligence Laboratory
The University of Texas at Austin
Austin, TX 78712

Using Multi-Strategy Learning to Improve Planning Efficiency and Quality

by

Tara Adrienne Estlin, M.S., B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 1998

Using Multi-Strategy Learning to Improve Planning Efficiency and Quality

Approved by Dissertation Committee:

To my father, for giving me the belief that I could accomplish whatever I set my mind to.

Acknowledgments

There are many people who I would like to thank for their support and for their contributions to this research. First and foremost, I would like to thank my advisor, Ray Mooney, for all of his help and guidance throughout my time as a graduate student. Working with Ray has been a continuous learning experience. He was always available to help point me in the right direction and I am very grateful for all of his advice and teachings. I also I would also like to thank the other members of my committee, which include Bruce Porter, Ben Kuipers, Risto Miikkulainen and Steve Minton for all of their helpful suggestions and comments.

I would like to give special thanks to my husband, David Moriarty. Without David's encouragement I would have never even considered the getting a PhD. David was always there to listen to my concerns and questions and to provide his unfailing support. Many times he brought me back down to reality when I was lost in worry over some inconsequential thing. I am very glad we were able to go through this together.

There are many other people I would like to thank. I have had many wonderful office mates and colleagues over the years who I will miss. I have enjoyed countless office banter with Dan Clancy, John Zelle, Jeff Mahoney and Paul Baffes. Thank you guys for all the times you made me laugh. Begin stuck up in the attic of Taylor didn't seem so bad when I had you around for all the wonderful conversation. Also, a special thank you to John Zelle for helping me start along this research path and for always providing his help and advice.

There are many other friends and colleagues who have offered both support when I needed help and distractions when I needed a break. In particular, I would like to thank Mary Elaine Califf, Sowmya Ramachandran, Cindi Thompson, Andrea Haessley, Charles Calloway and Lance Tokuda.

I also would like to thank other colleagues outside of my department who have made much of this research possible. In particular, I'd like to mention the AI group at the University of Washington and the Prodigy group at Carnegie Mellon for both providing your code and for inspiring me in my research.

This research was supported by the NASA Graduate Student Researchers Program, grant #NGT-51332.

Tara Adrienne Estlin

The University of Texas at Austin
May 1998

Using Multi-Strategy Learning to Improve Planning Efficiency and Quality

Technical Report AI98-269

Tara Adrienne Estlin, Ph.D.
The University of Texas at Austin, 1998

Supervisor: Raymond Mooney

Artificial intelligence planning systems have become an important tool for automating a wide variety of tasks. However, even the most current planning algorithms suffer from two major problems. First, they often require infeasible amounts of computation time to solve problems in most domains. And second, they are not guaranteed to return the best solution to a planning problem, and in fact can sometimes return very low-quality solutions. One way to address these problems is to provide a planning system with domain-specific control knowledge, which helps guide the planner towards more promising search paths. Machine learning techniques enable a planning system to automatically acquire search-control knowledge for different applications. A considerable amount of planning and learning research has been devoted to acquiring rules that improve *planning efficiency*, also known as *speedup learning*. Much less work has been down in learning knowledge to improve the *quality of plans*, even though this is an essential feature for many real-world planning systems. Furthermore, even less research has been done in acquiring control knowledge to improve both these metrics.

The learning system presented in this dissertation, called SCOPE, is a unique approach to learning control knowledge for planning. SCOPE learns domain-specific control rules for a planner that improve *both* planning efficiency and plan quality, and it is one of the few systems that can learn control knowledge for partial-order planning. SCOPE's architecture integrates explanation-based learning (EBL) with techniques from inductive logic programming. Specifically, EBL is used to constrain an inductive search for control heuristics that help a planner choose between competing plan refinements. Since SCOPE uses a very flexible training approach, its learning algorithm can be easily focused to prefer search paths that are better for particular evaluation metrics. SCOPE is extensively tested on several planning domains, including a logistics transportation domain and a production manufacturing domain. In these tests, it is shown to significantly improve both planning efficiency and quality and is shown to be more robust than a competing approach.

Contents

Chapter 1

Introduction

Planning is a ubiquitous and integral part of every day life. We plan when we go to the grocery store, when we take a trip, and even when we take the dog for a walk. Small tasks take very simple plans, such as driving to a restaurant for dinner, while larger tasks can take very extensive and complicated plans. For instance building a new house requires a set of long and detailed plans to be done correctly. Planning also has an important role in the workplace and is used for a variety of tasks such as building pieces of machinery or designing efficiency delivery routes. Much time and effort is often required to construct correct and efficient plans for such problems and many human hours can be spent on this process.

Artificial intelligence (AI) planning systems provide a way to automate many of these planning tasks. Given a set of goals, AI planners generally search through a list of relevant domain actions until a correct list of those actions has been found that can achieve the desired goals. These systems can save countless hours by quickly constructing the necessary steps to perform a particular task. They can also be helpful in constructing *optimal* or *high-quality* plans that will perform the task in the best possible manner. For instance, it may be important to a manufacturer that plans for building machinery minimize certain types of resource consumption. Similarly, it is important for a package delivery service to design delivery routes that allow for the quickest delivery of all packages.

Once a plan has been constructed by a planning system, it can be used in several different ways. The plan is sometimes passed to a human operator who then carries out the specified steps. Another option is to give the plan to a computer execution agent which performs the task with little or no human intervention. Planning systems have become a powerful and popular tool for performing a range of activities, from manufacturing semiconductors (Fargher & Smith, 1994), to cleaning up oil-spills (Agosa & Wilkins, 1996), to scheduling antenna communications with orbiting spacecraft (Chien, Govindjee, Estlin, Wang, & Jr., 1997). As the desire for automation grows more prevalent in today's society, the need for efficient and intelligent planning systems grows as well.

1.1 Acquiring Planning Control Knowledge

Researchers have introduced numerous approaches to planning that have been tested and utilized on a variety of toy and real-world domains. Yet, even the newest domain-independent planning algorithms suffer from two major drawbacks. First, they often require large amounts of computation to solve even relatively simple problems and cannot tractably handle most realistic problems. Second, these algorithms are usually not guaranteed to return the best solution, and often return sub-par solutions to many planning problems. In many real-world settings, returning an optimal or near-optimal solution can be critical and is often more important than returning the solution quickly.

One way to address these problems is to provide a planner with considerable amounts of extra knowledge about a domain. This additional domain information is often termed "control knowledge" and indicates *how* a planner should best achieve its goals. Control knowledge is usually represented in the form of search-control rules or heuristics that can be easily utilized by the planner. Control rules can specify information such as what actions should be added to achieve particular goals, or in what order problem goals should be examined.

For example, consider the simple planning problem shown in Figure 1.1 from a logistics transportation domain created by Veloso (1992). In this domain, packages must be delivered to different locations in different cities. Packages must be transported between cities by airplane and within a city by truck. In Figure 1.1 there are two packages that must be transported from the Austin airport to the Chicago airport. Two possible solutions to this problem are shown in Figure 1.2. Though the problem in Figure 1.1 is relatively simple, a planner will often perform much unnecessary search before arriving at one of these or other possible solutions. For instance, in trying to achieve the first goal, at-obj(pkg1,chicago-airport), the planner may first try to use the truck, by adding the action unload-truck(pkg1,truck1,chicago-airport), and continue along that search path until it finally fails since the truck cannot be used for inter-city delivery. To avoid this unnecessary search, the control rule shown in Figure 1.3 could be utilized. This rule checks if the goal location of the package (the Chicago airport) and the current package location (the Austin airport) are in different cities. If this condition is found to be true, then the control rule directs the planner to select the appropriate action unload-airplane(pkg1,plane1,chicago-airport). This rule could also be expanded to check for other conditions; for instance, it could be beneficial to check if the goal location is an airport. Other rules could also be used that apply in other domain situations.

Control knowledge, such as the rule shown in Figure 1.3, is not necessary for normal or correct planner operation. Without it, the planner is still guaranteed to return a correct plan if given enough time and if a plan does exist. Yet control knowledge can be very useful in addressing the two drawbacks mentioned previously. First, control rules can help a planner to avoid considerable search by directing it towards the most promising search paths early in the planning process. The rule shown in Figure 1.3 is an example of this type of knowledge. Second, rules can also be used to direct a planner towards optimal or *high-quality* solutions. For instance, in Figure 1.2, Plan 2 is a more satisfactory plan since it requires fewer steps. Control rules could be added that direct the planner towards finding Plan 2 as opposed to finding Plan 1. Without the addition of control rules, a planner may be unable to produce solutions in a reasonable amount of time, and the few solutions it can produce may be considered infeasible or of low quality.

Thus, control rules can be a very important component of any planning system. Unfortunately, constructing control rules for a domain is a difficult, laborious task. One way to acquire rules is to consult with a domain expert. However, acquiring the necessary domain information from an expert and coding that information in a form usable by the planner can be a very hard if not impossible task. This problem is one form of the well-known "knowledge-bottleneck problem."

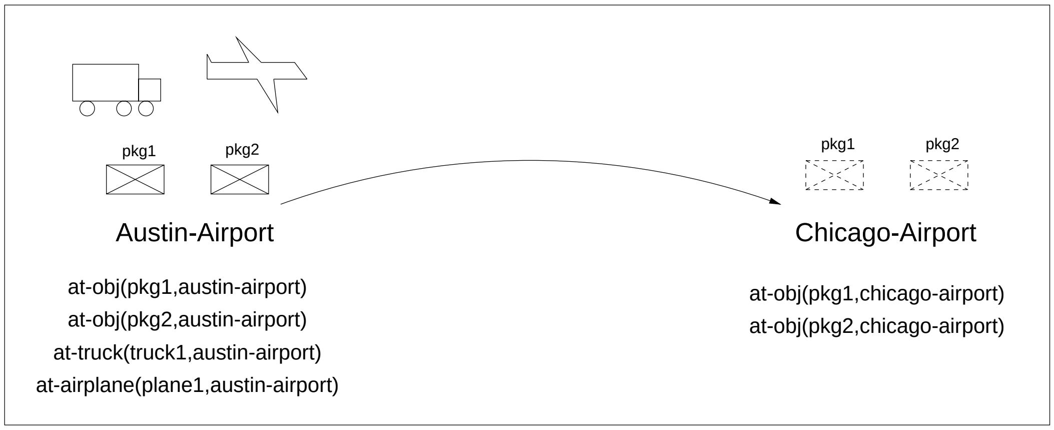

pkg1
pkg2
Austin-Airport
at-obj(pkg1,austin-airport)
at-obj(pkg2,austin-airport)
at-truck(truck1,austin-airport)
at-airplane(plane1,austin-airport)
pkg1
pkg2
Chicago-Airport
at-obj(pkg1,chicago-airport)
at-obj(pkg2,chicago-airport)

Figure 1.1: A problem from a logistics transportation domain. This problem specifies a problem initial state on the left-hand side and a list of goals that must be achieved on the right. The goals for this problem specify that two packages must be moved from the Austin airport to the Chicago airport.

Plan1
load-airplane(pkg1,plane1,austin-airport)
fly-airplane(plane1,austin-airport,chicago-airport)
unload-airplane(plane1,chicago-airport,austin-airport)
fly-airplane(plane1,chicago-airport,austin-airport)
load-airplane(pkg2,plane1,austin-airport)
fly-airplane(plane1,austin-airport,chicago-airport)
unload-airplane(pkg2,plane1,chicago-airport)

Plan2
load-airplane(pkg1,plane1,austin-airport)
load-airplane(pkg2,plane1,austin-airport)
fly-airplane(plane1,austin-airport,chicago-airport)
unload-airplane(plane1,chicago-airport,austin-airport)
unload-airplane(pkg2,plane1,chicago-airport)

Figure 1.2: Two plans for the problem shown in Figure 1.1.

```
IF    current-goal(at-obj(X,Loc)) ∧ true-in-state(at-obj(X,Loc2)) ∧ different-city(Loc,Loc2)
THEN  use operator unload-airplane(X,Plane,Loc)
```

Figure 1.3: A simple control rule for the logistics transportation domain. This rule would direct the planner to choose the action "unload object X from airplane Plane at location Loc" if the current goal was to have object X be at location Loc and if it's true that X's location in the current world state (Loc2) is in a different city than its goal location (Loc).

Many experts cannot easily relay the information that they use in constructing good plans; and even once the information is given, hand-coding that information in the form of planning control rules can be very difficult and time-consuming. Plus, an expert may not always be easily accessible for the domain we are working in. Additionally, every time the domain changes, or we apply the planner to a new domain, we must acquire a new set of control rules.

Research in planning and machine learning attempts to address the problem of acquiring usable control knowledge by developing methods that *automatically* build search-control rules for different domains. By examining past planning scenarios, a learning system can often construct control heuristics that will help a planner perform well on future problems. In the past few decades, a number of learning systems have been built that learn control knowledge for planning. However, these systems are rarely used in planning systems that are prominently used today. The reasons for this are two-fold. First, most past learning systems have been designed to learn control rules for *state-space* planners. State-space planning is a older method of planning that is used infrequently in present-day planners. A much more common and current style of planning is *plan-space* or *partial-order* planning, which has been shown to be more effective than state-space planning in many domains (Barrett & Weld, 1994; Kambhampati & Chen, 1993; Minton, Drummond, Bresina, & Phillips, 1992). Most past learning systems, however, cannot be applied to partial-order planners due to their reliance on the state-space approach.

A second reason many of these learning systems aren't used is that the majority of them have only concentrated on improving planner efficiency, also know as *speedup learning*. Another very important goal is to improve the *quality* of final plans. The quality of a plan usually refers to its optimality, which is measured by a predefined metric. For instance, one measure of plan quality could be the number of steps contained in the plan. In this case, the *optimal* plan is the one containing the least number of steps. Other quality metrics may also be important depending on the domain. Only a few learning systems have addressed the problem of improving plan quality, even though producing high-quality or near-optimal plans is a very important consideration in most real-world planning domains. Additionally, even less work has been done on learning rules to improve *both* efficiency and quality. Little research has been done examining whether these two metrics can be successfully improved simultaneously and what the tradeoffs are in improving one over the other.

This dissertation presents a new approach to learning planning control knowledge that can learn rules for a variety of planning algorithms, including partial-order planning techniques. This approach is shown to not only improve both the efficiency of a current planning system but also the quality of its final solutions. By addressing both of these problems, this research is intended to strengthen the bridge between building AI planning systems and applying these systems successfully to real-world problems.

1.2 Scope: A Control Knowledge Acquisition System

The Scope Learning System is a new approach to acquiring domain-specific search-control knowledge for planning. Scope stands for the *Search Control Optimization of Planning through Experience*. This system automatically constructs control rules that specify when certain plan refinements should be applied, where a plan refinement is any modification to the plan, such as adding a new action or ordering constraint. For instance a control rule might specify when a particular action should be added to a plan in order to achieve a certain goal. These rules help avoid inappropriate plan refinement applications during planning, and thus they help a planner to find solutions quickly and can also lead a planner towards the *best* solution paths.

To acquire search-control rules for planning, Scope uses a unique combination of machine learning techniques. Specifically, it combines a form of *explanation-based learning* (EBL) (Mitchell, Keller, & Kedar-Cabelli, 1986; DeJong & Mooney, 1986) with techniques from *inductive logic programming* (Quinlan, 1990; Muggleton, 1992; Lavrač & Džeroski, 1994). This integration allows Scope to efficiently build very general control knowledge that is useful in applying to unseen planning problems. Control rules are built by performing an inductive search through the space of possible control rules, where the goal is to build general but effective rules that always lead the planners towards good search paths and never towards infeasible ones. In order to perform the inductive search efficiently, the search is biased by using *explanation-based generalization* (EBG) to direct the search towards information that was found useful in solving previous planning examples.

Unlike most past approaches, Scope's algorithm is not tied to a particular type of planning, but instead is intended to apply to many different styles of problem solvers. Thus, Scope can be effectively applied to the prominent partial-order style of planning. Most other learning approaches are ineffective at learning control rules for this type of planner. Another difference between Scope and most past approaches is that Scope has a flexible architecture and can be trained to improve upon different planning metrics. In particular, Scope has been focused to improve both planning efficiency and final plan quality. Most other learning systems concentrate only on improving planning efficiency and almost no systems look at improving both these factors.

The Scope algorithm has been evaluated on a number of different planning problems using several different domains, including the blocksworld domain, a transportation domain and a production manufacturing domain. Scope is shown to significantly improve both planning efficiency and quality on these problems. In particular, Scope is shown to significantly increase planner efficiency, sometimes by an order of magnitude, and is also shown to produce optimal planning solutions for many problems. Additionally, Scope is shown to be more robust than a competing approach at providing speedup on different types of domain definitions.

1.3 Organization of Dissertation

The rest of this dissertation is organized as follows. Chapters 2 and 3 present background knowledge on both planning systems and on learning planning control knowledge. Chapter 4 discusses learning control rules for partial-order planning and presents the partial-order planning algorithm used as a testbed for SCOPE. Chapter 5 presents the SCOPE approach to learning control rules for planning and details the different features of its algorithm. Chapters 6, 7, and 8 discuss how SCOPE was experimentally evaluated on several planning domains and present the results of this evaluation. Chapter 9 discusses related work in the area of learning control rules for planning and for other types of problem solvers. Finally, Chapter 10 presents ideas for future research directions and Chapter 11 reviews the ideas and results presented in this dissertation and discusses relevant conclusions.

1.4 Summary

Control information can play a very important role in most planning systems by allowing the system to solve problems efficiently and also by helping the system to produce high quality solutions. In the past, machine learning techniques have been employed to automatically acquire control knowledge, however, the resulting knowledge has usually been directed at improving only one goal, e.g. improving efficiency *or* improving quality. Little work has been done in trying to learn control rules that improve upon both these metrics.

This dissertation introduces the SCOPE learning system which can learn search-control rules for planning that improve upon both efficiency and quality. SCOPE uses a unique combination of machine learning techniques to acquire control rules. And unlike most past systems, SCOPE can be effectively applied to different styles of planning and it can be used to improve both efficiency and plan quality.

Chapter 2

Background on Planning Methods

Learning search-control information for planning is very dependent on the type of planning system being employed. Thus, the first part of this chapter discusses what is meant by plan generation and reviews the different methods that can be used to form a plan. The second part of the chapter presents the two main metrics that are used to evaluate planning systems: efficiency and plan quality. These two metrics are defined and different methods of measuring them are discussed.

2.1 A Planning Problem

A planning problem is traditionally defined as consisting of three main parts: an *initial-state*, a set of *goals*, and a set of possible *actions*. Most planners operate by beginning with an empty plan and then they add actions one at a time until a solution is found. A planner must be able to perform a number of functions, including choosing a good action to add to the current plan, detecting when a solution has been found, and recognizing dead-end search paths that should be abandoned.

Consider the planning problem introduced in Figure 1.1. The first two parts of this problem would be defined as follows:

- *Initial-state:* at-obj(pkg1,austin-airport),at-obj(pkg2,austin-airport), at-truck(truck1,austin-airport), at-airplane(plane1,austin-airport)
- *Goals:* at-obj(pkg1,chicago-airport), at-obj(pkg2,chicago-airport)

The third set of information is the set of available actions, which depends on the particular task or problem domain. In many standard approaches to planning, domain actions are represented in a STRIPS operator format (Fikes & Nilsson, 1971), which consists of a list of preconditions, an add list and a delete list. Definitions for several actions from the logistics transportation domain (Veloso, 1992) are shown in Figure 2.1. In order for an action to be applied in a plan, its preconditions must be satisfied in the current state of the world. For example, in order to apply the logistics action load-truck(?X,?Y,?Z) it must be true in the current state that object ?X is at location ?Z, and that truck ?Y is also at location ?Z. Once an action is applied, any conditions on its add list are added to the current world state and all delete conditions are removed. In other words, an action's add and delete list describe how the action changes the world.

load-truck(?X,?Y,?Z)
Preconditions: at-obj(?X,?Z), at-truck(?Y,?Z)
Add List: inside-truck(?X,?Y)
Delete List: at-obj(?X,?Z)

unload-truck(?X,?Y,?Z)
Preconditions: inside-truck(?X,?Y), at-truck(?Y,?Z)
Add List: at-obj(?X,?Z)
Delete List: inside-truck(?X)

drive-truck(?X,?Y,?Z)
Preconditions: at-truck(?X,?Y), same-city(?Y,?Z)
Add List: at-truck(?X,?Z)
Delete List: at-truck(?X,?Y)

Figure 2.1: Operator Schemas from the Logistics Transportation Domain

Unfortunately, this action format is insufficiently expressive for most realistic planning domains. In order to perform planning in real-world scenarios, the STRIPS-style action format has been extended by many researchers to include more expressive constructs. One such format is the Action Description Language (ADL) from Pendault (1989). Additional constructs used by ADL and other extended representations include items such as conditional effects, universal quantification of precondition and effect variables, and disjunctive preconditions (Pendault, 1989; McDermott, 1991; Penberthy & Weld, 1992; Chien & DeJong, 1994). These constructs allow the domain writer to create more complicated and expressive operator definitions. For an example of a domain that uses these additional constructs, see the definition of the process planning domain (Gil, 1991) in Appendix A.

2.2 Search Methods

Given a planning problem, many different search methods have been developed for finding a correct plan. Most classical planners, including the original STRIPS planner, employ a *state-based* search. In this approach, the planner always maintains a representation of the current world state to help guides its search for a correct plan. An example of a state-based search in the blocksworld domain (Nilsson, 1980) is shown in Figure 2.2. Here the planner starts with the initial state and searches through new states until a state where all the goals are true is reached. New states are created by applying a domain action. This type of planning can involve either a simple forward search from the initial state, such as shown in Figure 2.2, or a more sophisticated goal-directed search that reasons backwards from the goal state. A well-known goal-directed search technique is *means-end analysis*, which selects operators to add to the plan that reduce the difference between the goal state and the current state. In a state-based planner, all actions in the plan are ordered with respect to all other actions so that the current state of the world can always be determined. This type of planning algorithm is often termed a *total-order* approach since the current plan must be maintained as a

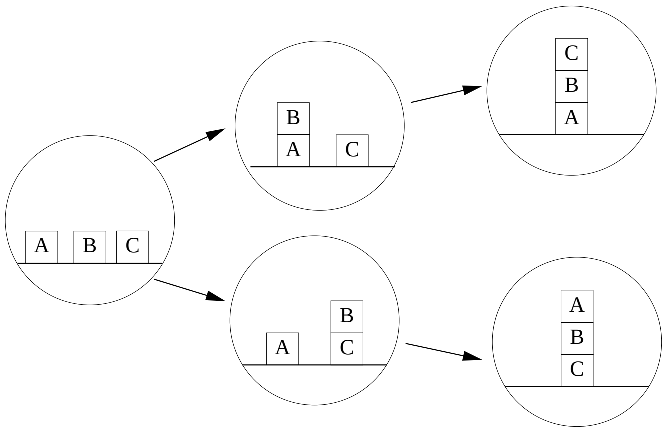


Figure 2.2: A planning *state-based* search. Here the planner searches through a space of world states until a state is reached where the problem goals are true.

completely *ordered* list of actions.

A second style of planning employs a *plan-based* search and does not maintain the current world state as it proceeds. This type of planner begins with a null plan and then searches through the space of possible plans until it finds a solution. Actions are still added to the current plan until a solution is reached, however, instead of saving the plan as a completely ordered list of actions, the current plan is represented as a *partially-ordered* set of actions. During planning, only necessary ordering decisions are saved in the plan; all others are postponed until later in the planning process when more information is available. Though only necessary ordering constraints are included in the current plan, a valid ordering of all operators must always exist. A plan-space planner typically proceeds by repeatedly identifying an unachieved goal or precondition, selecting an operator that will achieve it, and then adding an instantiated version of that operator to the current plan along with any necessary ordering constraints. If a situation arises where one plan action interferes with another plan action, then the planner will attempt to add an additional constraint to resolve the problem. The planning cycle continues until all goals have been achieved. Figure 2.3 shows an example of plan-space planning. This type of planner is often referred to as a *partial-order* planner as opposed to the *total-order* planners discussed above. Many researchers consider partial-order planning a more powerful and efficient planning strategy since premature ordering commitments are delayed until a more informative ordering decision can be made (Barrett & Weld, 1994; Kambhampati & Chen, 1993; Minton et al., 1992). For instance, Barrett and Weld (1994) show that a basic partial-order planning algorithm significantly outperforms two different total-order planning algorithms on a number of different domains, including domains with independent subgoals, serializable subgoals, nonserializable subgoals, heterogeneous sets of subgoals, and complex operator-selection decisions.

One other important distinction that is often made between planning algorithms is whether a planner is *linear* or *nonlinear*. Over the years, these terms have acquired several different connotations and are often confused or mistaken with the terms *state-based* and *plan-based*. The most common meaning is that they refer to the type of goal selection strategy employed by the planner. Linear planners, which are usually equated with state-based planners, typically examine goals in a "linear" order; if two (or more) goals exist, then the first goal and all of its subgoals must be

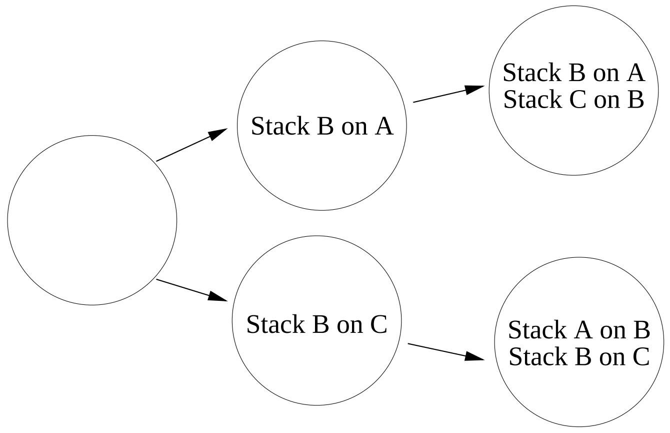


Figure 2.3: A planning *plan-based* search. Here the planner searches through a space of partial plans until a plan where all goals are achieved is found. This type of search is also known as *partial-order planning*.

achieved before the next goal is considered. In contrast, nonlinear planners do not employ the linearity restriction and can examine goals in any order. Partial-order planners are typically nonlinear planners, however they are not required to be. Similarly, state-based (or total-order planners) are usually linear planners, however there are also nonlinear planning algorithms that employ a state-based approach (Warren, 1974; Veloso et al., 1995). Since the terms *linear planner* and *nonlinear planner* are not well-defined, many recent authors choose instead to refer to planners as either *total-order* or *partial-order*.

2.3 Plan Evaluation Metrics

There are two main metrics that are commonly used to evaluate planning algorithms. The first metric, which has perhaps received the most attention in the planning literature, is the *efficiency* of the planning system. Unfortunately, even with the most current algorithms, most planning problems are difficult to solve. Even toy domains, such as the blocksworld, quickly become computationally intractable as problem difficulty increases. This intractability is further aggravated by more expressive domain definitions, such as those that include conditional effects and universal quantification. These constructs can be very important domain representation tools, however, they also greatly contribute to planning complexity making many problems very difficult to solve. Much past planning research has concentrated on improving planning efficiency, with the overall goal of being able to tractably solve larger and harder problems (e.g. Minton, 1989; Etzioni, 1993; Kambhampati et al., 1996).

The efficiency of a planner can be measured in several ways. The most common measurement is the time is takes to find a planning solution. Other ways to measure efficiency include the number of backtracks a planner performs, and the number of partial plans that must be explored before a solution is found.

The second metric used to evaluate planning algorithms is the *quality* of the output plan. A flaw in most current planning methods is that they often return sub-optimal solutions to planning problems. Solutions can be sub-optimal for a number of reasons; a solution may be costly to execute or may use a larger number of resources than necessary. In many real-world planning systems the quality of the final plan may be just as important, if not more important, than the time it takes to generate the plan. For instance, it may be vital in a manufacturing domain for a planner to produce plans with low resource consumption, or with least number of possible steps. There are a variety of notions about what makes a good plan. Some of the more common quality metrics are listed below (Pérez & Carbonell, 1994):

- The length of the plan (or the total number of steps)
- The execution time of the plan
- The resource consumption required
- The robustness of the plan

Depending on the domain being used, different quality metrics will have varying importance. Some research has been done on improving plan quality with the overall goal of consistently producing optimal or near-optimal plans (e.g. Pérez, 1996; Iwamoto, 1994).

In order to make planning on many domains more tractable and also to produce high quality plans, researchers often attempt to utilize domain-specific control knowledge (e.g Fikes & Nilsson, 1971; Minton, 1989; Langley & Allen, 1991). This knowledge can provide a planner with the extra information needed to make wise decisions early in the planning process, thereby avoiding large amounts of search and helping the planner to produce good solutions. The rest of this dissertation discusses a new technique for automatically producing control knowledge for planning systems.

2.4 Summary

Several different methods have been defined for finding a correct plan when given a planning problem. One of the most prominent techniques is *partial-order* planning, where the current plan is saved as a partially-ordered list of actions. Most learning techniques for acquiring control information have been applied to an older style of planning known as the *state-based* planner, which always maintains a representation of the current-world state during planning. Additionally, several different evaluation metrics are commonly used to judge the performance of a planning system. The main two metrics utilized by most researchers are the efficiency of a planning system and the quality of its final solutions. Efficiency is usually measured by the planning time required to find a solution. Final plan quality can be measured in several ways, with the most common being the number of steps contained in the solution. The rest of this dissertation discusses how learning methods can be applied to improve upon these two metrics.

Chapter 3

Learning Planning Control Knowledge

Planning systems can often *automatically* acquire control knowledge by utilizing machine learning techniques. This knowledge can be based directly on user input or derived from past planning experience. For instance, a system which acts as a *learning apprentice* is designed to acquire control information by observing the behavior of a human expert. Other learning systems are built to learn from their own experience without human intervention. These autonomous systems are understandably more complex and difficult to build, however, they are frequently more desirable since they require little or no human overhead. A number of past learning systems have been built that learn control knowledge automatically, with no user intervention. The rest of this section is used to discuss the different methods that have been used to accomplish this task.

3.1 EBL

The most common approach to learning control knowledge is to use *analytical* learning techniques, where the systems learns by problem solving. The most popular analytical method is *explanation-based learning* (EBL) (Mitchell et al., 1986; DeJong & Mooney, 1986), where the system learns by analyzing explanations of problem-solving behaviors. Specifically, an EBL system builds an explanation for why an example is a member of some target concept. When used to learn planning control knowledge, the example usually corresponds to a planning decision and the target concept is explaining why the decision was a good one or a bad one. The explanation built by EBL can be used to construct a control rule, which can then be used to help make future planning decisions that were similar to the original example.

This dissertation uses a form of EBL, called *explanation-based generalization* (EBG) to acquire control knowledge. EBG operates by learning a set of sufficient conditions for being a member of a target concept. These conditions are acquired by generalizing an explanation of why a particular example fits the target concept definition. Explanations are in the form of proof trees composed of inference rules that prove the example is a member of the concept.

Figure 3.1 shows the inputs to an EBG system and gives a sample problem of learning the definition of a cup. Given this set of information shown, the EBG system will determine a generalization of the training example that is an *operational* definition for the goal concept, i.e. a concept definition which satisfies the operationality criterion. EBG works in two main steps, which are shown in Figure 3.2. The first step uses the domain theory to construct an explanation of

1. **Goal concept**: A definition of the concept to be learned in terms of high-level properties. For example, a goal concept for a cup (from Winston, Binford, Katz, and Lowry (1983)) might be:

   OPEN-VESSEL(x) $\wedge$ STABLE(x) $\wedge$ LIFTABLE(x) $\rightarrow$ CUP(x)

2. **Training example**: An example of the goal concept. For example, a training example of a cup might include the following:

   COLOR(OBJ1,RED)
   PART-OF(OBJ1,HANDLE1)
   PART-OF(OBJ1,BOTTOM1)
   ...

3. **Domain Theory**: A set of rules and facts to be used in explaining how the training example is an example of the goal concept. The domain theory for the cup might include a rule such as the following:

   IS(x,LIGHT) $\wedge$ PART-OF(x,y) $\wedge$ ISA(y,HANDLE) $\rightarrow$ LIFTABLE

4. **Operationality criterion:** A specification of how the learned concept definition must be expressed (i.e. the names of available low-level *operational* predicates). The operationality criterion for a cup might be that the definition be in terms of only observable features, such as the weight of the cup or whether it has certain physical parts.

Figure 3.1: Required inputs for an EBG system.

1. **Explain:** Construct an explanation in terms of the *domain theory* that proves how the training example satisfies the goal concept. Each branch of the explanation must terminate in an expression that satisfies the operationality criterion. An example of an explanation generated for the cup example is shown in Figure 3.3.

2. **Generalize:** Determine a set of sufficient conditions under which the explanation structure holds. This can be accomplished by regressing the goal concept through the explanation structure. The conjunction of the resulting regressed expressions constitutes the desired concept definition. Thus, after regressing the goal concept CUP(x) through the explanation shown in Figure 3.3 the following definition is produced:

   (PART-OF(x,xc) $\wedge$ ISA(xc,CONCAVITY) $\wedge$ IS(xc,UPWARD-POINTING) $\wedge$
   PART-OF(x,xb) $\wedge$ ISA(xb,BOTTOM) $\wedge$ IS(xb,FLAT) $\wedge$ PART-OF(x,xh) $\wedge$
   ISA(xh,HANDLE) $\wedge$ IS(x,LIGHT)) $\rightarrow$ CUP(x)

Figure 3.2: Two main steps in the EBG process.

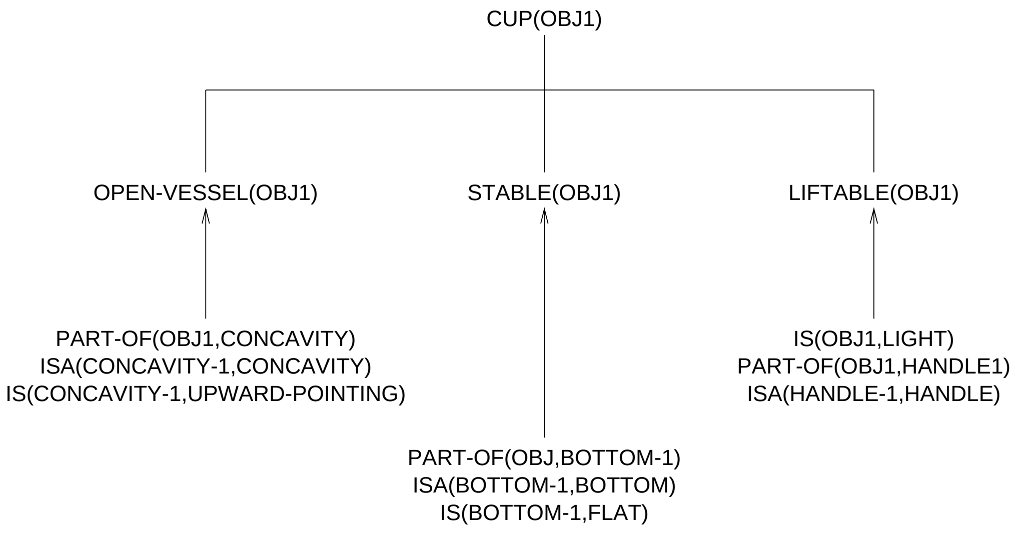


Figure 3.3: The explanation structure generated for the "cup" example.

why the example is a member of the target concept. Figure 3.3 shows an explanation which was constructed for the cup example. The root of the explanation (or proof tree) is the target concept of cup. The bottom leaves of the tree correspond to *operational* predicates that were used in building the explanation. In the second step of EBG, this explanation is generalized by regressing a generalized version of the target concept through the explanation structure. The operational concepts contained in the generalized explanation are then used to construct a concept definition, which can be utilized as a rule for correctly classifying that concept in future problems.

Explanation-based learning and other analytical techniques are considered *knowledge-rich* due to their ability to explain why a particular concept definition was learned. A number of EBL learning systems have been applied to acquire planning control knowledge (Minton, 1989; Etzioni, 1993; Bhatnagar & Mostow, 1994; Kambhampati et al., 1996). Unfortunately, due to its reliance on only a few examples, standard EBL can often produce complex, overly-specific control rules that do not generalize well to new planning situations (Minton, 1988). This situation is commonly known as the *utility problem* (Minton, 1988; Mooney, 1989; Cohen, 1990), where even though the learned rules are correct, the cost of testing their applicability to new planning situations often outweighs their savings. Another problem is that EBL methods are difficult to apply in domains where it is hard to construct a complete and tractable domain theory (Chien, 1989). These drawbacks make it difficult to successfully apply EBL methods to real-world problems.

3.2 Induction

Another method of learning planning control rules is to employ a form of induction, which acquires rules by examining a number of different planning examples. One form of inductive techniques used in this dissertation are *inductive logic programming* techniques, which use a combination of logic programming and machine learning methods, and can provide a good platform for learning control knowledge.

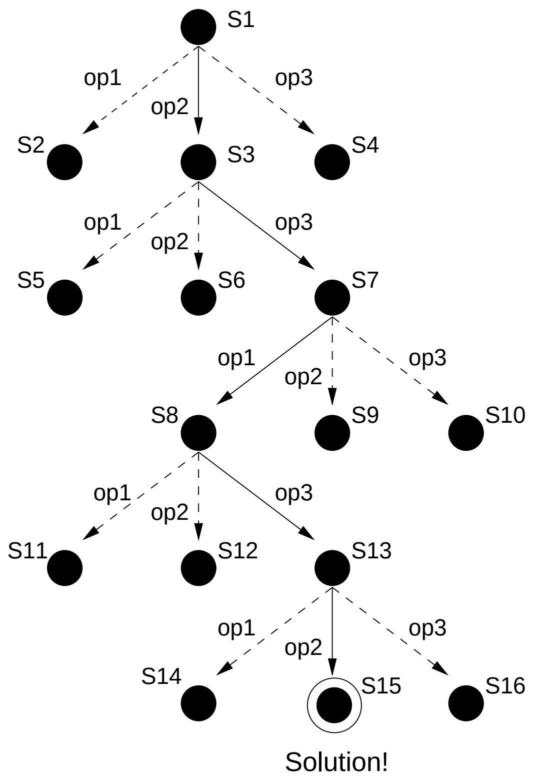


Figure 3.4: A search tree for finding a solution to a planning problem. The solid arrows show the solution path through the tree.

3.2.1 General Inductive Methods

Inductive learning techniques can acquire planning control rules by looking at examples of positive and negative planning decisions. These examples are usually found by solving a set of training problems and extracting examples based on good and bad planning decisions. Positive examples correspond to good planning decisions that occurred on search paths leading to a solution. Negative examples correspond to bad planning decisions that were on other search paths that either failed or were unexamined. Consider the search tree for a planning problem shown in Figure 3.4. Each node corresponds to different search states where a decision was made and each arrow corresponds to different planning operations that could be applied from that state. For instance, in a partial-order planner, the arrows would correspond to different plan refinements that could be made to the current plan. In the figure, the solid arrows show the solution path through the tree, while the dashed arrows show search paths that either failed or were never examined. Positive examples of planning decisions are found along the solution path. Any other examples of planning decisions, which are not along the solution path, could be collected as negative examples.

Empirical methods build control rules by performing an inductive search through the space of possible rules, where the general goal is to build rules that cover some positive planning decisions but don't cover any negative planning decisions. The inductive search is usually guided by a domain-independent bias. For example, one commonly used bias is Occam's razor (Quinlan, 1983) which prefers simple rules over more complex ones.

A main advantage of inductive learning techniques over other learning methods such as EBL is that they tend to learn very general control knowledge since they examine a number of different examples. This quality is important for building very useful control knowledge; general control rules are usually more effective than more specific rules at successfully applying to new planning problems. Several past learning systems have successfully utilized induction for learning planning control knowledge (Mitchell, Utgoff, & Banerji, 1983; Porter & Kibler, 1986; Langley & Allen, 1991; Leckie & Zuckerman, 1993). Unfortunately, inductive techniques have several disadvantages as well. For one, they usually require large numbers of examples to acquire effective control information. Also, these methods can quickly become computationally intractable since they often search through large amounts of information when building control rules. These disadvantages often prevent inductive methods from being successfully used on many domains.

3.2.2 Inductive Logic Programming

Some learning systems, including SCOPE, employ techniques from the field of inductive logic programming (ILP) to acquire knowledge. ILP research addresses the problem of inducing a first-order, definite-clause logic program from a set of examples. This field represents the intersection of standard logic programming and machine learning. Due to the expressiveness of first-order logic, ILP methods can learn relational and recursive concepts that cannot be represented in the attribute/value representations used by most machine-learning approaches. ILP systems have successfully induced small programs for simple tasks such as sorting and list manipulation (Muggleton & Buntine, 1988; Quinlan & Cameron-Jones, 1993); as well as performing well on more complicated tasks such as learning properties of organic molecules (Muggleton et al., 1992) and predicting the past tense of English verbs (Mooney & Califf, 1995).

The ILP algorithm used in this dissertation is a version of the Foil induction algorithm (Quinlan, 1990). Foil learns a function-free, first-order, Horn clause definition of a *target* concept. When learning control for planning systems, the target concept can correspond to concepts such as "when to apply a particular action" or "when to apply a particular ordering constraint". Once a definition is learned for a target concept, it can be utilized as a control rule in solving future problems.

The definition constructed by Foil is in terms of an input set of *background* predicates. The input to Foil consists of an extensional definition for the target concept and extensional definitions for all background predicates. Extensional definitions are in the form of tuples of constants of specified types. For example, suppose Foil is learning a definition of the simple concept "list membership". The input to Foil might be the following:

```
member(Elt,List): (a,[a]), (a,[a,b]), (b,[a,b]), (a,[a,b,c]), ...
components(List,Elt,List): ([a],a,[]), ([a,b],a,[b]), ([a,b,c],a,[b,c]), ...
```

member(Elt,List) is the target predicate for which a definition is being learned. Listed above are a set of positive examples of this concept. Foil also requires negative examples of the target predicate, which can be supplied directly or computed using a closed-world assumption. components(A,B,C) is a background predicate which is true if A is list whose first element is B and whose tail is C. Elt is a type denoting possible elements, which include a, b, and c, and List is a type defined as a list containing items of type Elt.

```
Initialization
    Definition := null
    Remaining := all positive examples
While Remaining is not empty
        Find a clause, C, that covers some examples in Remaining,
            but no negative examples.
        Remove examples covered by C from Remaining.
        Add C to Definition.
```

Figure 3.5: Basic FOIL Covering Algorithm

Given these inputs, FOIL learns a program one clause at a time using a greedy-covering algorithm, which is summarized in Figure 3.5. Clauses are constructed one at a time where each learned clause covers some positives examples of the target concept and no negative examples. For example, a clause that might be learned for member after one iteration of this loop is:

member(A,B) :- components(B,A,C).

This clause covers all positive examples where the element is the first one in the list (e.g. member(a,[a,b])) but does not cover any negatives. A clause that could be learned to cover the remaining examples is:

member(A,B) :- components(B,C,D), member(A,D).

Together these two clauses constitute a correct program for the target predicate member.

The central part of the FOIL algorithm is in the "find a clause" step, which is implemented by a general-to-specific hill-climbing search. When building a clause, FOIL adds antecedents to the developing clause one at a time. At each step FOIL evaluates all possible literals that might be added and selects the one which maximizes an information-based gain heuristic. This heuristic prefers literals that cover more positive examples and fewer negative examples. The algorithm maintains a set of variable binding tuples for all positive and negative examples that satisfy the current clause. Also included in the tuples are bindings for any new variables introduced in the body. The pseudocode in Figure 3.6 summarizes this procedure.

FOIL considers adding literals for all possible variable combinations of each background predicate as long as type restrictions are satisfied and at least one of the predicate arguments is an existing variable bound by the head or a previous literal in the body. Literals are evaluated based on the number of postive and negative tuples covered, where preferred literals are those that cover many positives and few negatives. Let T_+ denote the number of positive tuples in the set T and define:

$$I(T) = -log_2(T_+/|T|).$$

```
Initialize $C$ to the target predicate.
Initialize $T$ to contain the positive tuples in *Remaining* and all the negative tuples.
While $T$ contains negative tuples
    Find the best literal $L$ to add to the clause.
    Form a new training set $T'$ that contains all positive and negative tuple
        extensions that satifiy $L$. A tuple is extended by including a set of
        bindings for the new variables introduced by $L$.
    Replace $T$ by $T'$.
```

Figure 3.6: The "find-a-clause" step in the FOIL algorithm.

The chosen literal is then the one that maximizes the following information-gain heuristic:

$$gain(L) = s \cdot (I(T) - I(T')),$$

where s is the number of tuples in T that have extensions in T' (i.e. number of current positive tuples covered by L).

FOIL also includes many additional features such as methods for testing equality, adding useful literals that do not immediately provide gain (*determinate literals*), and the pre-pruning and post-pruning of clauses to prevent overfitting. More information on FOIL can be found in (Quinlan, 1990; Quinlan & Cameron-Jones, 1993; Cameron-Jones & Quinlan, 1994).

3.2.3 ILP for Control

It has also been argued that ILP techniques can be a useful tool for acquiring control information (Cohen, 1990). Many different problem solving strategies can be easily coded as logic programs and learning mechanisms are also easily implemented in this framework. Logic programming also provides a well-understood representational and computational platform upon which to build. There are a number of current learning systems that employ ILP techniques to induce Horn clause concept definitions (Quinlan, 1990; Muggleton, 1992); the FOIL learning system is one such example. By casting the problem of learning control rules as a concept learning problem, these inductive techniques can often successfully be used to acquire control information.

A logic program is expressed using the definite clause subset of first-order logic, where a definite clause is a disjunction of literals having exactly one unnegated literal. The one unnegated literal represents the clause *head* while the other literals comprise the clause *body*. Computation in this representation is done using a resolution proof strategy on an existentially quantified goal. For example, a simple logic program to sort lists (written in Prolog) is shown in Figure 3.7. The top-level goal of this program is sort(X,Y). An instantiation of this goal is true when Y is a sorted version of the list represented by X. The arguments of a top-level goal are usually partitioned into input and output argument sets. In this example, X is considered the input and Y the output. A program is executed by providing a goal that has its input arguments instantiated. When such a goal is provided, a theorem-prover constructively proves the existence of the goal meeting any constraints provided through the input arguments. In this process, the prover will produce bindings for the output arguments. For example, in our simple sorting program, a top-level goal of the form sort([6,3,1,5,9],Y) would produce the output binding Y = [1,3,5,6,9].

```
sort(X,Y) :- permutation(X,Y), ordered(Y).

permutation([],[]) :- true.
permutation([X|Xs],Ys) :- permutation(Xs,Ys), insert(X,Ys,Ys1).

insert(X,Xs,[X|Xs]) :- true.
insert(X,[Y|Ys],[Y|Ys1]) :- insert(X,Ys,Ys1).

ordered([X]) :- true.
ordered([X,Y|Ys]) :- X ≤ Y, ordered([Y|Ys]).
```

Figure 3.7: Simple Sorting Program

```
insert(X,[X|Xs],Xs) :- insert_control1(X,[X|Xs],Xs),!.
insert(X,[Y|Ys],[Y|Ys1]) :- insert(X,Ys,Ys1).

insert_control1(X,[],[X]).
insert_control1(X,[X|Z],[X,Y|Z]) :- X < Y.
```

Figure 3.8: Improved Insert Predicate

The Prolog programming language provides a practical instantiation of logic programming using a simple control strategy. In Prolog, depth-first search with backtracking is used to search for a proof. If during execution the current search path fails, then the last non-deterministic decision point is backtracked upon and a new path explored. Search control in a Prolog program can be viewed as a *clause-selection* problem (Cohen, 1990), where clause selection is the process of deciding what program clause should be used to reduce a particular subgoal during program execution. Different options in a program are represented using separate clauses which have unifiable heads but different clause bodies. Thus the clause heads can all be matched with the same type of subgoal, however, the bodies contain different conditions which test whether that program option should be selected. If an incorrect clause is selected to solve a program subgoal, then that clause application will eventually be backtracked upon and another matching clause used. Control information is usually incorporated into a Prolog program in the form of clause-selection rules. These rules help avoid inappropriate clause applications, which both greatly reduces backtracking and can help lead the program towards better solution paths.

As an example, consider the simple sorting program shown in Figure 3.7, which sorts a list by generating permutations of the list until it finds one that is ordered. Permutations are generated by permuting the tail of the input list and then inserting the head somewhere in the permuted tail. This program currently performs in $O(N!)$ time. The only nondeterminism comes from the definition of the predicate insert/3 which can either insert an item at the beginning of a list or somewhere in the tail. This nondeterminism can be eliminated by learning a control rule for the first clause that will correctly predict when the item should be placed at the head of the list.

Figure 3.8 shows a modified version of the insert clause definition, which was constructed by the DOLPHIN learning system (Zelle & Mooney, 1993). The first insert clause has been guarded with control information so that attempts to use it inappropriately will immediately fail. This clause will now only be applied when an element is being inserted into an empty list or if the new element is less than the head of the current list. The cut "!" added to the end of the first insert clause makes the clause deterministic. When the first insert clause is selected and the control conditions are found true, then this decision cannot be backtracked upon. The result of adding the control knowledge is an $O(N^2)$ insertion sort program.

Cohen (1990) and Zelle and Mooney (1993) have both introduced systems that acquire control heuristics to improve the performance of Prolog programs. In these systems, a combination of EBL and induction is used to learn control rules that eliminate backtracking in logic programs. Combination learning techniques are discussed more in the next section. This dissertation presents research that successfully extends these methods by applying ILP techniques for control knowledge acquisition in a complex planning system. Also, unlike past systems, the SCOPE learning system uses ILP techniques to learn control rules that not only improve program efficiency, but also improve the quality of produced solutions.

3.3 Multi-Strategy Learning

One other approach to learning control information is to use a combination of learning techniques. Most of these methods attempt to combine EBL with an inductive algorithm where EBL is generally used to construct the control rules and then the rules are further refined using induction. The main goal of these methods is to retain the benefits of a domain theory while also having the flexibility to learn from the data. For example, instead of building a complete proof, *plausible explanation-based learning* (PEBL) (Zweben, Davis, Daun, Drascher, Deale, & Eskey, 1992) first conjectures an example is a member of the target concept, and then confirms the conjecture with empirical data. Other systems have employed *lazy explanation-based learning* (LEBL) which generates incomplete explanations and then incrementally refines any overly-general knowledge using new examples (Tadepalli, 1989; Borrajo & Veloso, 1994).

This dissertation presents a novel multi-strategy learning approach to control-knowledge acquisition for planning systems. The SCOPE learning system also uses a combination of EBL and induction to learn control information. However, instead of generating control rules through EBL and then inductively refining them, SCOPE builds rules using an inductive algorithm. EBL is used to focus the inductive search so that only highly relevant pieces of background information are examined for possible inclusion in a rule. This technique biases the search towards more useful rules, and also keeps the inductive search at a computationally tractable level. This methods also differs from other control-rule learning techniques by employing inductive-logic programming techniques. This type of learning combination has been used in the past to learn control rules for simple logic programs, however, this dissertation presents its first application to a complicated planning system.

3.4 Summary

A number of learning techniques have been utilized to acquire search-control knowledge and many of these have been applied to improve planning systems. The most popular method is explanation-based learning, which constructs control rules by "explaining" why a planning decision was correct or incorrect. Though EBL methods have the advantage of requiring only a few examples, they also have a significant drawback since the rules they construct are very specific and can be very expensive to apply to future problems.

Another learning technique used by some researchers to acquire planning control knowledge is induction. It construct rules by examining a number of examples and then using those examples to perform an inductive search through the space of possible rules. Some inductive systems employ methods from the field of inductive logic programming. ILP techniques combine methods from the fields of induction and logic programming. These techniques have been primarily used in the past to induce logic programs for classifying examples, however, they are also a very effective tool at acquiring control information. Logic programming provides a good platform for representing control knowledge and for easily incorporating control rules into a planning system or other type of problem solver.

Induction is beneficial because it tends to learn very general control rules that can easily apply to new planning situations. Unfortunately, induction also has several drawbacks in that a large number of examples are often required to learn good rules, and also, the inductive search to find a good rule can be computationally large or intractable.

Some learning systems have combined EBL and induction to acquire control information. These combination techniques have the advantages of being able to utilize a domain theory and also being able to learn more general control knowledge. Most combination methods have used EBL to construct control rules and then have inductively refined the rules based on other examples.

The Scope learning system EBL and induction in a different approach from past learning systems; induction is used to learn rules and EBL is utilized to bias the inductive search towards useful information. Scope also employs inductive logic programming techniques to learn rules, which few past learning systems have done.

Chapter 4

Learning Control for Partial-Order Planning

As mentioned in the introduction, previous research in planning and learning systems has been based almost entirely on linear, state-based planning algorithms. Since the introduction of this type of planner, a number of more sophisticated planning approaches have been developed that typically outperform linear, state-based algorithms. Unfortunately, few control-knowledge acquisition systems have been adapted to perform on these newer planning algorithms. One style of planning that has acquired much prominence is partial-order planning. This type of approach is widely used in many current planning systems and thus a partial-order planner was thus identified as a good testbed for the SCOPE control rule learning system.

4.1 The UCPOP Planner

The base planner that was chosen for experimentation is UCPOP (Penberthy & Weld, 1992), a partial-order planner whose step descriptions can include conditional effects and universal quantification. UCPOP has been proven sound and complete, and a significant amount of planning research has been based around its algorithm (e.g. Poet & Smith, 1993; Gerevini & Schubert, 1996; Kambhampati et al., 1996).

4.1.1 Planner Representation

Given a planning problem, which contains an initial state, a set of goals, and a set of domain operators, the goal of UCPOP is to determine a sequence of actions that will transform the initial state into a state where all goals are satisfied. Operators are specified using Pednault's Action Description Language (ADL) (Pendault, 1989), which was discussed in Section 2.1 and is an extension of the well-known STRIPS format (Fikes & Nilsson, 1971). Operators contain precondition, add and delete lists, and they can also contain constructs such as conditional effects, disjunctive preconditions, and universal quantification.

UCPOP searches for a solution in a space of partial plans, where each plan consists of a partial-ordering of actions. A partial plan is best described as a four-tuple $\langle\mathcal{A},\mathcal{B},\mathcal{O},\mathcal{L}\rangle$: where $\mathcal{A}$ is a set of actions, $\mathcal{O}$ is a set of ordering constraints over $\mathcal{A}$, $\mathcal{L}$ is a set of causal links, and $\mathcal{B}$ is a set of binding constraints over variables appearing in $\mathcal{A}$. Specifically, $\mathcal{A}$ contains all actions that have been added as current plan steps. The set of orderings, $\mathcal{O}$, specifies a partial ordering of these actions. Ordering constraints between steps are usually denoted by the "<" relation. For example $A_1 < A_2$ means that step A_1 is constrained to come before A_2. During planning, there must always exist at least one consistent total ordering of all plan steps.

Causal links, contained in $\mathcal{L}$, record dependencies between the effects of one action and the preconditions of another. A link is represented as $A_1 \overset{Q}{\rightarrow} A_2$ where A_1 and A_2 are plan steps and Q is an effect of A_1 and a precondition of A_2. In this case, A_1 is considered the link's producer and A_2 its consumer. These links are used to detect *threats*, which occur when a new action interferes with a past decision. More specifically, if $A_1 \overset{Q}{\rightarrow} A_2$ is a causal link in the current plan and there exists a separate action in the plan A_3, which threatens the link, then the following two conditions are satisfied:

- $\mathcal{O} \cup A_1 < A_3 < A_2$ is consistent, and
- A_3 has $\neg Q$ as an effect (i.e. A has Q as a delete condition).

When a plan contains a threat, it is possible that it will not work as anticipated. To prevent this from happening, the planner must check for and resolve any discovered threats. UCPOP employs three main threat resolution strategies: *promotion*, *demotion*, and *confrontation*. For *promotion*, the planner adds an additional ordering constraint to ensure that A_3, the threatening action, is executed before A_1, the link's producer, i.e. $A_3 < A_1$. Similarly, for *demotion*, an ordering constraint is added that requires A_3 to be executed after A_2, the link's consumer, i.e. $A_2 < A_3$. *Confrontation* can be used to resolve a threat if A_3's threatening effect is conditional. In this case, the planner adds the negation of the conditional effect's antecedent to the agenda. For example, if the threatening effect is conditional with antecedent S and consequent $\neg Q$, then $\neg S$ would be added as a new goal to the agenda.

Since, some actions in a plan can be partially instantiated, sometimes a *threat* is subject to interpretation. For instance, if the link condition Q is clear(a) and A_3 has an effect not(clear(?X)), where ?X is currently uninstantiated, then A_3 could *possibly* incur a threat. UCPOP's strategy in this situation is to wait until a threat is undeniable (e.g. all variables are fully instantiated) before imposing a threat resolution strategy to handle the threat.

The last set B contains a list of binding constraints, which are in the form of codesignation and noncodesignation constraints. Codesignation constraints represent the required unification of two variables ($?X = ?Y$) or a variable and a constant ($?X = A$). Conversely, noncondesignation constraints prohibit the unficiation of two variables ($?X \neq ?Y$). These constraints apply to variables appearing in the pre- and post-conditions of the actions contained in $\mathcal{A}$.

4.1.2 Universal Quantification in UCPOP

Allowing universal quantification in action preconditions and effects allows one to easily describe many real-world situations. For instance, without universal quantification it would be impossible to specify an action such as the UNIX "rm *" which removes all files in a directory. In order to implement this type of expressiveness in domain definitions, UCPOP developers made several simplifying assumptions. First, it is assumed that the world being modeled has a finite, static universe of objects. Thus objects cannot be created or deleted during planning. Second, each object must have a type that is declared in the problem initial state. For example in the blocksworld, for every block X mentioned in the problem, the predicate block(X) must be included in the list of initial state predicates.

In order to establish goals that contain universally quantified formulas, UCPOP maps these formulas into a corresponding ground formula where all universally quantified variables are replaced with constants. This mapping is done by taking the *univesal base* of the quantified formula. The universal base Υ of a first-order, function-free sentence, Δ, is defined a follows:

$$\Upsilon(\Delta) = \Delta \text{ if contains no quantifiers}$$
$$\Upsilon(\forall_{t1}\chi\Delta(\chi)) = \Upsilon(\Delta_1) \wedge ... \wedge \Upsilon(\Delta_n)$$

where the Δ_i corresponds to each possible intrepretation of $\Delta(\chi)$ under the universe of discourse (or all possible objects of type $t1$). For example, suppose that the universe of block is {a,b,c} . If Δ is forall ((block ?x)) (clear ?x)), then the universal base is the following: (and (block a) (block b) (block c)). The definition of Υ can also be extended to handle interleaved universal and existential quantifiers. Please see Weld (1994) for more information.

4.1.3 Algorithm

An overview of the UCPOP algorithm is shown in Figure 4.1. The algorithm takes three inputs: a plan $\langle\mathcal{A},\mathcal{B},\mathcal{O},\mathcal{L}\rangle$, an agenda of outstanding goals, and a set of action schemata Λ. The initial plan for a planning problem has two actions, $A = \{A_0, A_\infty\}$, one ordering constraint, $\mathcal{O} = \{A_0 < A_\infty\}$, no causal links, $\mathcal{L} = \{\}$, and no binding constraints, $\mathcal{B} = \{\}$. The initial and goal states are represented in the initial plan by adding the two actions A_0 and A_∞, where the effects of A_0 correspond to the initial state and the preconditions of A_∞ correspond to the desired goal state. The initial agenda contains all top-level goals; each goal in the agenda is represented as a pair $\langle Q, A_i\rangle$ where Q is a precondition of A_i.

In each planning cycle, Step 1 checks if the agenda is empty and if not, Step 2 selects a goal to work on. If this goal is quantified then the universal base of this expression is posted to the agenda and a new goal is selected. If the goal is a disjunction or conjunction than one goal in the formula is chosen to work on, and if the original goal was a conjunction, then the remaining goals are posted to the agenda to be established later. After a goal has been selected, Step 3 chooses an existing or new action to assert the goal. Once an action is selected, the corresponding ordering constraints, casual links and codesignation constraints are added to $\mathcal{O}$, $\mathcal{L}$, and $\mathcal{B}$, and if a *new* action was selected, it is added to $\mathcal{A}$. Step 4 removes the selected goal from the agenda, and if a *new* action was selected to assert it, that action's preconditions are added to the agenda. Step 5 checks for possible threats and resolves any found by either adding an additional ordering constraint through demotion or promotion, or by adding a new goal to the agenda through confrontation. UCPOP is called recursively until the agenda is empty. On termination, UCPOP uses the constraints found in $\mathcal{O}$ to determine a total ordering of the actions in $\mathcal{A}$, and returns this as the final solution.

```
Algorithm UCPOP($\langle\mathcal{A},\mathcal{B},\mathcal{O},\mathcal{L}\rangle$,agenda,$\Lambda$)

  1. Termination: If agenda is empty, return $\langle\mathcal{A},\mathcal{B},\mathcal{O},\mathcal{L}\rangle$.

  2. Goal Selection: Remove a goal $\langle Q, A_{need}\rangle$ from agenda where $Q$ is a
     precondition of action $A_{need}$.

     (a) If $Q$ is a quantified sentence then post the universal base $\langle\Upsilon(Q), A_c\rangle$
         to agenda. Go to 2.
     (b) If $Q$ is a conjunction of $Q_i$ then post each $\langle Q_i, A_{need}\rangle$ to agenda. Go
         to 2.
     (c) If $Q$ is a disjunction of $Q_i$ then nondeterministically choose one dis-
         junct, $Q_k$, and post $\langle Q_k, A_c\rangle$ to agenda. Go to 2.
     (d) If $Q$ is a literal and a link $A_p \overset{\neg Q}{\rightarrow} A_c$ exists in $\mathcal{L}$, fail (an impossible
         plan).

  3. Operator Selection: Choose either an existing action (from $\mathcal{A}$) or a
     new action $A_{add}$ (instantiated from $\Lambda$) that adds $Q$. Let $\mathcal{O}' = \mathcal{O} \cup$
     $\{A_{add} < A_{need}\}$, $\mathcal{L}' = \mathcal{L} \cup \{A_{add} \overset{Q}{\rightarrow} A_{need}\}$, and let $\mathcal{B}'$ be the updated set
     of bindings. If $A_{add}$ is a new action let $\mathcal{A}' = \mathcal{A} \cup A_{add}$ and $\mathcal{O}' = \mathcal{O}' \cup$
     $A_o < A_{add} < A_\infty$.

  4. Update Goal Set: Let agenda' = agenda - $\{\langle Q, A_{need}\rangle\}$. If $A_{add}$ is
     newly instantiated, then for each condition, $Q_i$, on its precondition list
     add $\langle Q_i, A_{add}\rangle$ to agenda'.

  5. Causal Link Protection: For every action $A_t$ in $\mathcal{A}$ that might threaten
     a causal link $A_p \overset{R}{\rightarrow} A_c$ in $\mathcal{L}$ choose one of the following:

     (a) Demotion: Add $A_t < A_p$ to $\mathcal{O}'$, or
     (b) Promotion: Add $A_c < A_t$ to $\mathcal{O}'$, or
     (c) Confrontation: If $A_t$'s threatening effect is conditional with an-
         tecedent $S$ and consequent $R$, then add $\langle \neg S \setminus MGU(P, \neg R), A_t\rangle$ to
         agenda'.[1]

     If no choice is consistent then fail.

  6. Recursive Invocation: UCPOP($\langle\mathcal{A}',\mathcal{B}',\mathcal{O}',\mathcal{L}'\rangle$,agenda',$\Lambda$)
```

Figure 4.1: The UCPOP Partial-Order Planning Algorithm

4.2 UCPOP in Prolog

In order to learn control rules for UCPOP using the logic programming platform, a version of the UCPOP algorithm was implemented in Prolog. Planning decision points in this program are represented as clause-selection problems (i.e. each decision option is formulated as a separate clause). As explained in Section 3.2.3, this type of representation allows control rules to be easily incorporated into program clauses.

Though the planning algorithm is directly based on UCPOP, there are implementation differences. The most significant difference is that the Prolog planner operates using a depth-first backtracking search with a depth bound[2], while UCPOP normally employs a best-first search strategy. Kambhampati et al. (1996) also run UCPOP in a depth-first search mode in their control-rule learning system. This system is discussed further in related work and to the author's knowledge, is the only other system besides SCOPE that can learn control rules for UCPOP. To evaluate the efficiency of the Prolog planner as compared to the standard Lisp implementation of UCPOP (v2.0) (Barrett et al., 1993), several experiments were run using problem sets from three domains, which are also used for testing the learning algorithm. In these tests, which are discussed in Sections 6 and 7, the Prolog planner performed comparably to UCPOP and in some cases performed better. Though these experiments are not intended to promote a particular search strategy or programming language, they do indicate that the Prolog version of UCPOP is compatible in terms of efficiency to the standard Lisp version of UCPOP.

4.3 Planning Decision Points

There are several important decision points in the UCPOP algorithm where the planner must select from one of several possible plan refinements. Control rules can be very beneficial at these points by helping the planner to make the right decision. UCPOP decision points include goal selection, goal establishment (selecting an existing or new operator), threat selection, and threat resolution. At these points there are often a number of valid plan refinements that the planner must choose from. For instance, there may be several new actions that can be added to achieve a particular goal. In the absence of control information, the Prolog UCPOP will always select the first valid refinement it finds; other refinements may be tried later through backtracking.

An example of a planning decision point from the logistics transportation domain (Veloso, 1992) is shown in Figure 4.2 and Figure 4.3. Figure 4.2 shows a simple problem from this domain where the goal is to deliver one package from the Austin airport to the Chicago airport. Figure 4.3 shows the beginning of the search process to find a solution to this problem. There are two possible plan-refinement candidates in this domain for adding a new action to achieve the goal at-obj(pkg1,Chicago). As shown in the figure, the package can be transported by either using the truck or the plane. In this particular case, only one refinement option (candidate 2) will ever lead to a desirable problem solution; since this is a inter-city delivery, the package must be transported using the plane.

[1]MGU(A,B) is defined as a function that returns the most general unifier of A and B.

[2]The Prolog planner also includes a loop detection advice that prevents depth-first search from getting stuck in infinite loops where the same sequence of action is added repeatedly.

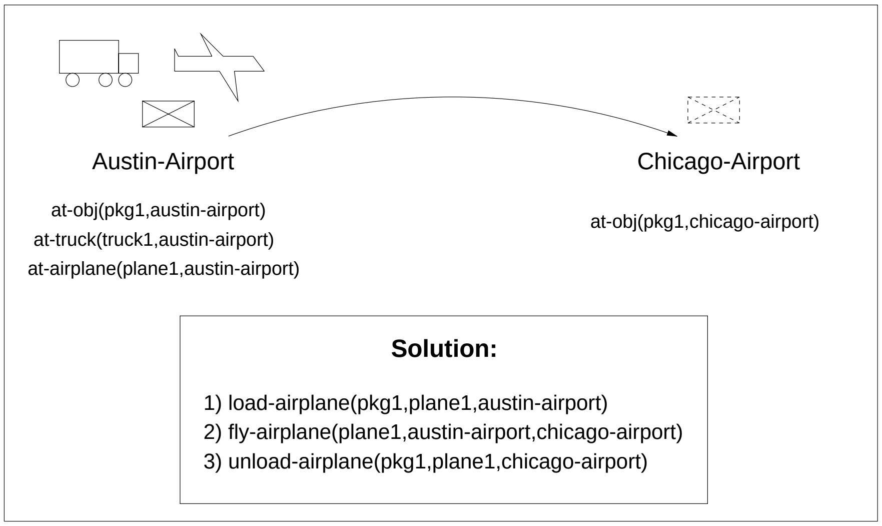

Austin-Airport
at-obj(pkg1,austin-airport)
at-truck(truck1,austin-airport)
at-airplane(plane1,austin-airport)
Chicago-Airport
at-obj(pkg1,chicago-airport)
Solution:
1) load-airplane(pkg1,plane1,austin-airport)
2) fly-airplane(plane1,austin-airport,chicago-airport)
3) unload-airplane(pkg1,plane1,chicago-airport)

Figure 4.2: A simple problem from the logistics transportation domain. The goal of the problem is to deliver a package from Austin to Chicago.

SCOPE has been designed to only learn control rules for decisions that might be backtracked over (e.g. could lead to a failing search path). Though goal selection and threat selection can affect planner performance, these decisions will *never* be backtracked upon, thus, no control rules are learned for these points. Search control rules for the remaining decisions types (goal establishment and threat resolution) are in the form of refinement-selection rules. If effective rules are acquired for these backtracking points, then in most domains, control rules for other decision points are not necessary for improving planner performance.

4.3.1 Control Rule Format

SCOPE learns control rules in the form of refinement-selection rules that define when a particular plan refinement should be applied. A selection rule consists of a conjunction of conditions that must all evaluate to true for the refinement to be used. If at least one condition fails, that refinement will be rejected, and the next refinement candidate evaluated. For example, shown below is a selection rule for the first candidate from Figure 4.3, which contains several control conditions.

```
Select operator unload-truck(?X,?Y,?Z) to establish goal(at-obj(?X,?Z),A₁)
   If  find-existing-action(at-obj(?X,?W),Steps,A₂) ∧
       member(same-city(?Z,?W),InitState) ∧ valid-ordering(A₂,A₁,Orderings).
```

This rule states that the operator unload-truck(?X,?Y,?Z) should be selected to add at-obj(?X,?Z) when there is another action A_2 that asserts object ?X is at a location that is in the same city as the goal location ?Z, and A_2 can be ordered before the action requiring the goal, A_1.

Learned control information is incorporated into the planner so that attempts to select an inappropriate refinement will immediately fail. Each plan refinement may have several control rules that apply to it. Each time the refinement is considered, at least one of its control rules must evaluate to true for the refinement to be applied. On the planning decision shown in Figure 4.3, the

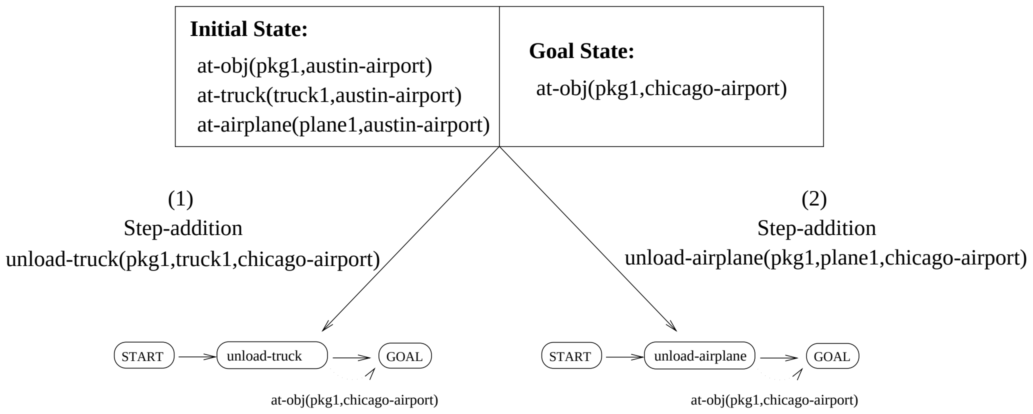

Initial State:
at-obj(pkg1,austin-airport)
at-truck(truck1,austin-airport)
at-airplane(plane1,austin-airport)
Goal State:
at-obj(pkg1,chicago-airport)
(1)
Step-addition
unload-truck(pkg1,truck1,chicago-airport)
(2)
Step-addition
unload-airplane(pkg1,plane1,chicago-airport)
START
unload-truck
GOAL
at-obj(pkg1,chicago-airport)
START
unload-airplane
GOAL
at-obj(pkg1,chicago-airport)

Figure 4.3: The top of the search tree used to find a solution to the problem shown in Figure 4.2. There are two possible plan refinements which can be used to achieve the goal at-obj(X,Chicago).
.

rule shown above would fail for the first refinement candidate. If this was the only rule attached to candidate 1, it would correctly prevent the planner from selecting it. The planner would them move on and immediately select candidate 2. Scope can also make selection rules deterministic or nondeterministic depending on the accuracy of the learned rule. If a ruled is deemed fully accurate, then a Prolog cut "!" is added directly after the call to the rule, which means once the rule has correctly fired, that refinement selection cannot be backtracked upon.

4.4 Summary

This section outlined the UCPOP planning algorithm and how control knowledge is implemented within its framekwork. UCPOP is a partial-order planning algorithm, which can be used on domains containing expressive constructs such as universal quantification and conditional effects. UCPOP operates in a recursive fashion; in each call to the planner, a goal is selected from the agenda and then an operator is selected to achieve the goal. UCPOP records partial orderings of related actions and always verifies that a consistent total-ordering of actions exists. If a situation is detected where an action in the plan could possibly interfere with another action, UCPOP applies several threat resolution strategies to ensure the plan can still be correctly executed. This process is continued until all goals have been solved and all threats have been resolved.

In order to stay in the ILP framework, a version of UCPOP has been implemented in Prolog. In this planner, planning decision points are represented as clause-selection problems so that control information can be easily incorporated. Scope learns control rules for planning decisions that might be backtracked upon. These rules are in the form of refinement-selection rules which evaluate whether a certain plan refinement should be selected to resolve a particular planning subgoal. The process of learning these rules in explained in detail in the next section.

Chapter 5

The Scope Control-Rule Learning System

The SCOPE learning system automatically acquires domain-specific control rules for planning. The input to SCOPE is a planning program and a set of training examples. SCOPE uses the examples to induce a set of control heuristics that can direct the planner towards promising search paths. These heuristics are then incorporated into the original planner and a new planning program is produced. Figure 5.1 shows the three main phases of SCOPE's algorithm. First, in the example analysis phase, SCOPE solves the training examples using the original planner and extracts useful control information about what planning decisions where made. This information is then passed to the control rule induction phase where it is used to generate selection rules for choosing plan refinements. Finally, learned rules are incorporated into the original planner in the program specialization phase, and SCOPE outputs the new planner. The complete algorithm is explained in detail in the next three sections.

5.1 Example Analysis

In the example analysis phase, the training examples are solved using the existing planner. A trace of the planning decision process used to solve each example is stored in a proof tree, where the tree root represents the top-level call to the planner and the tree nodes correspond to different planning procedure calls. The top part of a sample proof tree is shown in Figure 5.2, which was generated for the logistics problem introduced by Figure 4.3. The top-level goal in this proof is a call to the planner that includes the initial agenda (containing the top-level goals) and the initial lists of plan actions, ordering constraints, and causal-links as input arguments.[1] The last argument corresponds to the output plan solution. The remaining proof tree nodes correspond to subsequent planning procedure calls which were used to find a solution. Proof trees are created for each training example provided to SCOPE.

From these proof trees, two main outputs are produced: a set of *refinement-selection* examples and a set of *generalized proof trees*. Refinement-selection examples are used to record successful and unsuccessful applications of a plan refinement. Generalized proof trees provide a background

[1]Binding constraints in our system are maintained through Prolog, therefore, the set of binding constraints, B, is not explicitly represented in planning procedure calls.

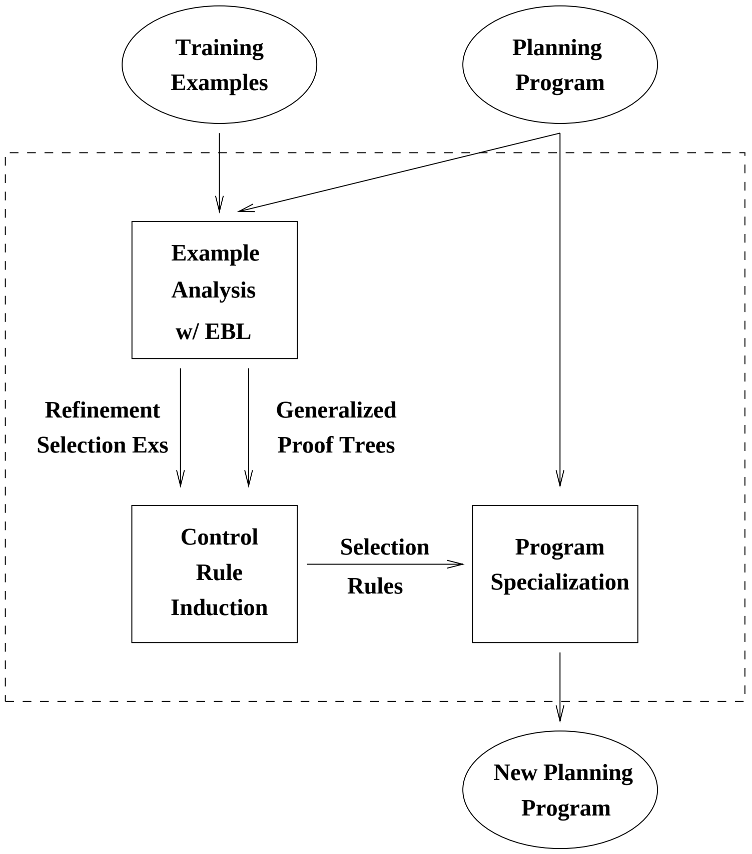


Figure 5.1: Scope's High-Level Architecture

context that explains the success of all correct planning decisions. These two pieces of information are explained in more detail below and are used in the next phase to build control rules.

A refinement-selection example for a particular plan refinement is a planning subgoal to which that refinement was applied. A correct refinement selection is an application of that refinement found on a solution path. An incorrect refinement selection is a refinement application that was tried and later backtracked over.

As an example, consider the planning problem that was introduced in Figure 4.3. The planning subgoal represented by this figure corresponds to the select-new-op procedure call shown in Figure 5.2. The select-new-op procedure selects a new action to add to the plan to achieve a particular goal. This procedure has several inputs including the goal to be achieved and one output, a new action that will achieve the goal. Figure 5.3 shows a stylized version of the subgoal introduced by Figure 5.2. This subgoal would be identified as a positive refinement-selection example for refinement candidate 2 from Figure 4.3 (adding the action unload-plane(pkg1,plane1,chicago-airport)), and would also be classified as a negative selection example for candidate 1 (adding the action unload-truck(pkg1,truck1,chicago-airport)). Positive and negative refinement-selection examples are collected for all sets of competing planning refinements. Refinements are considered "competing" if they can be applied in identical planning decisions, such as the two refinement candidates shown in Figure 4.3. Choosing between competing refinements is where most backtracking occurs in solving a planning problem. Any given training example may produce numerous positive and negative ex-

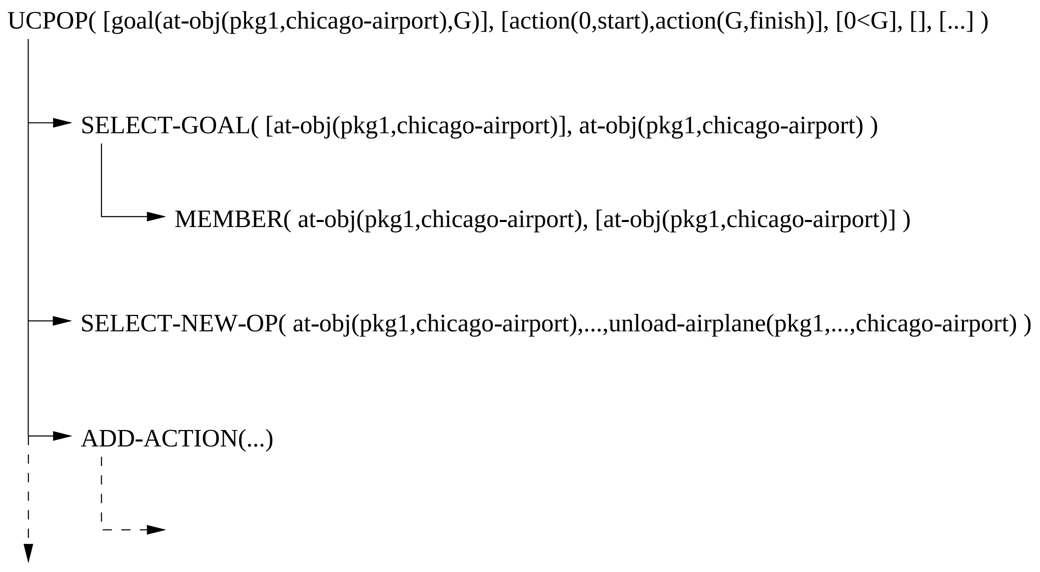

UCPOP([goal(at-obj(pkg1,chicago-airport),G)], [action(0,start),action(G,finish)], [0<G], [], [...])
SELECT-GOAL([at-obj(pkg1,chicago-airport)], at-obj(pkg1,chicago-airport))
MEMBER(at-obj(pkg1,chicago-airport), [at-obj(pkg1,chicago-airport)])
SELECT-NEW-OP(at-obj(pkg1,chicago-airport),...,unload-airplane(pkg1,...,chicago-airport))
ADD-ACTION(...)

Figure 5.2: Solution Proof of Planning Problem from Figure 4.3[2]

For $\mathcal{S}$ = (0:Start,G:Goal),
$\mathcal{O}$ = (0 < G),
$\mathcal{L} = \emptyset$,
agenda = (at-obj(pkg1,chicago-airport),G),
Select a new-operator ?OP **to establish** goal(at-obj(pkg1,chicago-airport),G)

Figure 5.3: The planning subgoal that represents the decision point shown in Figure 4.3.

amples of refinement-selections. These selection examples are used later in induction to represent positive and negative examples of when to apply particular plan refinements.

The second output of the example analysis phase is a set of generalized proof trees. Information from these proof trees is utilized in the induction phase to help build control rules. Generalized proofs are produced by using the following procedure. Standard explanation-based generalization (EBG) techniques (Mitchell et al., 1986; DeJong & Mooney, 1986) are used to generalize each training example proof. This generalization is performed by "retracing" the proof steps that were used to create the original trace using a goal with uninstantiated input arguments. The goal of this generalization is to remove proof elements that are dependent on the specific example facts while maintaining the overall proof structure. After EBG has been applied, some tree nodes can still contain complex lists as arguments. These arguments correspond to items such as the list of plan actions, the list of plan ordering constraints, etc. In order to promote more general control knowledge, any list data structures remaining in a generalized proof tree are generalized directly to variables.

An example of the top part of a generalized proof tree is shown in Figure 5.4. This proof was extracted from the solution trace shown in Figure 5.2 and then generalized using the technique just described. The generalized proof of an example, such as this one, provides a context which

[2]For space purposes, some information is not shown here. For instance, action data structures should also contain precondition and effect lists. Thus, the data structure for action 0 would contain the initial state as a list of effects.

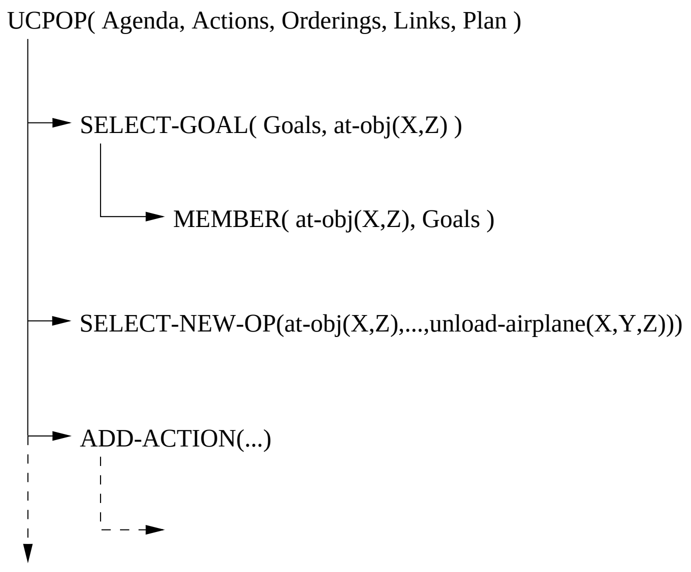

UCPOP(Agenda, Actions, Orderings, Links, Plan)
SELECT-GOAL(Goals, at-obj(X,Z))
MEMBER(at-obj(X,Z), Goals)
SELECT-NEW-OP(at-obj(X,Z),...,unload-airplane(X,Y,Z)))
ADD-ACTION(...)

Figure 5.4: Generalized Proof Tree of Solution Trace from Figure 5.2

"explains" the success of correct decisions.

At the end of the example analysis phase, the set of generalized proof trees and all sets of refinement-selection examples are passed to the control-rule induction phase where they are used to construct control rules.

5.2 Control Rule Induction

The goal of the control-rule induction phase is to produce an operational definition of when it is useful to apply a plan refinement. Given a refinement candidate, C, SCOPE builds a definition of the concept "planning subgoals for which C is useful". For example, in the logistics transportation domain, such a definition is learned for each of the refinements shown in Figure 4.3. In this context, control rule learning can be viewed as relational concept learning. A number of systems have been designed to tackle this type of learning problem (Quinlan, 1990; Muggleton, 1992; Zelle & Mooney, 1994a). SCOPE employs a version of Quinlan's FOIL algorithm to learn control rules through induction.

The choice of a FOIL-like framework is motivated by a number of factors. First, the basic FOIL algorithm is relatively easy to implement and has proven efficient in a number of domains. Second, FOIL has a "most general" bias which tends to produce simple definitions. Such a bias is important for learning rules with a low match cost, which helps avoid the utility problem. Third, it is relatively easy to bias FOIL with prior knowledge (Pazzani & Kibler, 1992). In SCOPE's case, it utilizes the information contained in the generalized proof trees of planning solution traces to bias FOIL towards useful control information.

5.2.1 FOIL in SCOPE

As explained in Section 3.2, FOIL attempts to learn a concept definition in terms of a given set of background predicates. This definition is composed of a set of Horn clauses that cover all of the positive examples of a concept, and none of the negative examples. In acquiring control rules for planning, the concept definition learned by SCOPE is *when* to apply a particular plan refinement. The refinement-selection examples collected in the example analysis phase provide the sets of positive and negative examples for when each refinement should be applied.

SCOPE uses an intensional version of FOIL where background predicates can be defined as Prolog programs (Mooney & Califf, 1995) instead of requiring an extensional representation as in standard FOIL. Instead of matching a literal against a set of tuples to determine whether or not it covers an example, the Prolog interpreter is used to prove whether or not the literal can be satisfied using its intenstional definition. Thus expanded tuples are not maintained and positive and negative examples of the target concept are retested for each alternative specialization of the developing clause.

One major drawback to FOIL (and other similar inductive algorithms) is that the hill-climbing search for a good antecedent to add to a clause can easily explode, especially when there are numerous background predicates with large numbers of arguments. When selecting each new clause antecedent, FOIL tries *all* possible variable combinations for *all* predicates before making its choice. This search grows *exponentially* as the number of predicate arguments increases. SCOPE significantly reduces this search by utilizing the generalized proofs of training examples. By examining the proof trees, SCOPE identifies a small set of potential literals that could be added as antecedents to the current clause definition. Literals are added in a way that utilizes variable connections already established in the proof tree. This approach nicely focuses the FOIL search by only considering literals (and variable combinations) that were found useful in solving the training examples. The process of utilizing generalized proof trees to bias the FOIL search is explained in more detail in the next section.

5.2.2 Building Control Rules from Proof Trees

The generalized proofs of training examples can be seen as giving the context for the appropriate applications of plan refinements within a proof. Some nodes of a generalized proof tree contain calls to "operational" predicates. These are usually low-level predicates that have been classified as "easy to evaluate" within the problem domain, and thus can be used to build efficient concept definitions. The operational nodes of a proof represent all of the primitive conditions that had to be satisfied for the proof to succeed. SCOPE employs induction in an attempt to identify a small set of these simple tests that will provide necessary guidance in determining whether the application of a plan refinement is likely to lead to a solution. Since test conditions that verify a planning decision are sometimes not executed until much later, it is important to consider an entire example proof instead of just the surrounding context of a particular decision. For instance in the blocksworld domain, the planner might not verify that choosing the action putdown(a) to establish the goal clear(a) is correct until much later in the planning process when it checks to see if some other action has asserted holding(a).

SCOPE employs the same general covering algorithm as FOIL, which was presented in Figure 3.5, but modifies the clause construction step. In SCOPE, clauses are successively specialized by considering how their target refinements were used in solving training examples. Several different types of antecedents can be added to a developing clause. First, antecedents can be pulled directly from generalized proof trees. SCOPE can also use other types of antecedents such as negated proof

```
Initialize $C$ to the target predicate.
Initialize $T$ to contain the positive examples in *Remaining* and all the negative
      examples.
While $T$ contains negative tuples or a clause with positive gain cannot be found
      Create a set $S$ of all possible specializations of $C$ by adding the following to
         the clause:
            An operational literal from a generalized proof tree
            A negated literal from a generalized proof tree
            A non-codesignation constraint over variables appearing in $C$
            A determinate literal
            A literal defined in background knowledge
      Select the clause specialization $C'$ from $S$ that scores highest using the
      FOIL information gain metric
      Form a new training set $T'$ that contains all positive and negative examples
            that satifiy $C'$.
      Replace $T$ by $T'$.
```

Figure 5.5: The "find-a-clause" step in the induction algorithm used by SCOPE˙

tree literals, determinate literals, noncodesignation constraints, relational clichés, and other background definitions. Different types of antecedents are explained in detail later in this section. The pseudocode in Figure 5.5 summarizes the procedure SCOPE uses for adding a literal to a clause C.

For example, suppose we are learning a definition for when each of the refinement candidates in Figure 4.3 should be applied. The program predicate representing this type of refinement is select-new-op, which is shown below.

select-new-op(Goal,Steps,Orderings,Links,Agenda,ReturnOp)

This predicate is defined with several arguments including the unachieved goal and an output argument for the selected operator.[3]

For each plan-refinement candidate, SCOPE begins with the most general definition possible. For instance, the most general definition covering candidate 2's refinement-selection examples is Clause 1 in Figure 5.6. This overly general definition covers *all* positive examples and *all* negative examples of when to apply candidate 2, since it will always evaluate to true. **Clause 1** can be specialized by adding antecedents to its body. Most specializations are created by unifying the head of Clause 1 with a (generalized) proof subgoal that was solved by applying candidate 1 and then adding an operational literal from the same proof tree which shares some variables with the subgoal. (Other possible specializations can be created by adding other types of literals, which are described in the next section.) One possible specialization of Clause 1 is shown by Clause 2 in Figure 5.6. Here, a proof tree literal has been added which checks if there is an existing plan step, S2, that establishes the goal airport(Z).

Variables in a newly added antecedent can be connected with the existing rule head in several ways. First, by unifying a rule head with a generalized subgoal, variables in the rule head become unified with variables existing in a proof tree. All operational literals in that proof

[3]Plan state information (i.e. the list of current plan steps, ordering constraints, causal links, and agenda) is also automatically included as arguments to any refinement predicate.

```
select-new-op(Goal,Steps,Ords,Links,Agenda,unload-airplane(X,Y,Z)) :-                (1)
        TRUE

                                   ⇓

select-new-op((at-obj(X,Z),S1),Steps0,Ords,Links,Agenda,unload-airplane(X,Y,Z)) :-  (2)
        establishes(airport(Z),Steps1,S2).

                                   ⇓

select-new-op((at-obj(X,Z),S1),Steps,Ords,Links,Agenda,unload-airplane(X,Y,Z)) :-   (3)
        establishes(airport(Z),Steps,S2).
```

Figure 5.6: The process of constructing a clause. SCOPE begins with a general clause definition (1). The clause is then specialized by adding antecedents (2) and by unifying variables of the same type (3).

that share variables with the generalized subgoal are tested as possible antecedents. This method initially establishes many relevant variable connections between a rule head and its antecedents. For instance, in Clause 2, the variable Z appears in both the clause head and the newly added antecedent.

A second way variable connections may be established is through the standard FOIL technique of unifying variables of the same type. When SCOPE tests a literal for use in a control rule, the literal may contain input parameters that are not bound by the rule head or other existing literals in the rule. Since literals are extracted from different parts of the proof tree, some variable connections may be lost, causing some parameters in a literal to be unbound. If such parameters exist, SCOPE attempts to unify these parameters with terms of the same type that are already present in the rule. For example, Clause 2 from Figure 5.6 has an antecedent with an unbound input, Steps1, which does not match any other variables in the rule. SCOPE can modify the rule, as shown by Clause 3, so that the Steps1 is unified with a term of the same type from the rule head, Steps0. For each unbound input parameter, all possible variable unifications are tested as possible specializations of the current rule and the specialization which maximizes FOIL's information-gain heuristic is selected.

5.2.3 Types of Antecedents

SCOPE considers several different types of control rule antecedents during induction. Besides pulling literals directly from the generalized proof trees, SCOPE can also use negated proof literals, determinate literals, literals representing non-codesignation constraints, relational clichés, and other background knowledge.

Negated Antecedents

A good reason for not selecting one plan-refinement candidate is that another refinement is preferable; therefore, a good antecedent for one refinement's control rule can often be successfully used as a negated rule antecedent for a competing refinement. Potential *negated* antecedents for a particular refinement's control rules are determined by combining the sets of possible rule antecedents for all other competing refinements. Negated copies of these antecedents can be added to a rule in several ways. Standard FOIL only adds antecedents to a rule until all negative examples are removed. If there are any positives left to cover, a new rule is created. Alternatively, SCOPE can consider adding negated antecedents in order to cover more positives. Instead of only appending negated antecedents to the end of a rule, the SCOPE induction algorithm considers conjunctively grouping them with existing negated antecedents. This procedure can sometimes increase the number of positive examples covered by a rule without covering any additional negatives. For example, assume the induction algorithm is considering adding the antecedent not(ant4) to the following rule.

```
rulehead :- ant1, not(ant2), ant3.
```

SCOPE can form either of the two rules shown below, where in standard FOIL, only the first rule would have been considered.

```
rulehead :- ant, not(ant2), ant3, not(ant4).
rulehead :- ant, not(ant2, ant4), ant3.
```

The first rule is more specific than the original rule and could possibly exclude more negative examples. The second rule, on the other hand, is more general than the original rule and could cover more positive examples.

Determinate Literals

Determinate literals can be used to introduce new variables into a clause (Quinlan, 1991; Muggleton, 1992). These are literals which produce only one possible binding for each literal output parameter, and thus their inclusion does not significantly increase the inductive search space. Determinate literals are typically not added through standard induction since they produce little or no gain. However, they can still be useful as rule antecedents by introducing new information into a rule. In SCOPE's induction procedure, all possible determinate literals are automatically added when a new clause is created. If any prove unnecessary they are simply pruned after clause construction has ended. SCOPE currently considers adding two types of determinate literals, find-initial-state(Steps,InitState) and find-final-state(Steps,FinalState); both input a list of plan steps and output the list of propositions representing the initial or final state. These literals are included so control rules can easily access information about the initial and goal states.

Non-codesignation constraints

Another type of potential literal is a non-codesignation constraint of the form $X_i \neq X_j$, where X_i and X_j are variables existing in a clause. This type of antecedent checks if two variables (of the same type) are nonunifiable. For example, it may be beneficial to check if two actions are not the same, as in the following rule from the blocksworld domain.

```
select-new-op((S1,clear(X)),Steps0,Orderings,Links,Agenda,putdown(X)) :-
    establishes(holding(X),Steps0,S2),
    S1 ≠ S2.
```

This rule states that the action putdown(A) should be selected to achieve the goal clear(A) if there is an existing action, S2, that establishes holding(A) and S2 is not the same action which requires clear(A).

Relational Clichés

Another feature that has been added to SCOPE's algorithm is the use of relational clichés. Often during induction, an individual literal may not provide any gain when tested as a possible rule antecedent. However, when grouped with another related literal, the conjunction of the two literals may provide significant gain. In standard hill-climbing, this useful combination of literals may never be discovered if neither literal provides any gain individually. Furthermore, searching through all possible combinations of literals is not a practical consideration. To address this problem, Silverstein and Pazzani (1991) introduced *relational clichés* to suggest potentially useful *combinations* of predicates during relational learning. These clichés provides an efficient means of searching through a restricted subset of the space of predicate combinations.

Relational clichés consist of two parts:

1. A pattern, which is an abstract description of a conjunction of predicates.
2. A set of restrictions, which constrain the predicates and variable bindings that can be used to fill the associated pattern.

SCOPE has a number of different relational clichés that it can utilize. Some clichés are domain-dependent while others are domain-independent and have been used successfully in all domains tested. For example, two domain-independent relational cliché patterns are listed below.

Pattern1:	member(A,C), member(B,C)
Restrictions:	C must be a list of initial or goal state predicates.
Pattern2:	not(member(A,C)),not(member(B,C))
Restrictions:	C must be a list of initial or goal state predicates.

These two patterns allow induction to test for concepts present in the initial and final states that may be defined by two conditions. These particular patterns were introduced based on observations of SCOPE's behavior on the logistics transportation domain. In this domain, it is often useful to check whether an object is in one city in the initial state and a different city in the goal state. For instance, if the current goal is to move object X to a certain location Y, then, the predicates member(at-obj(X,Z),InitState) and member(same-city(Y,Z),InitState) could be used to determine whether object X must be transported between two different cities. To represent this concept, the two general patterns listed above are used. These patterns could be made more specific by adding more specialized restrictions, however, by keeping their description general other useful concepts contained in the initial and goal states may be discovered. Also, these two patterns are not forced to be domain-specific and have been found useful in other domains besides the transportion domain which originally inspired them.

To use relational clichés in our induction algorithm, additional rule specializations are generated by adding all combinations of predicates that fit a defined pattern and associated restrictions. These specializations are then added to the pool of all possible rule specializations from which the rule with the highest gain is chosen. Once an instantiated relational cliché has been found useful, this instantiatation is cached so that it can be easily used in the future with no search required. Caching relational clichés is discussed further in Section 5.6.

Background Knowledge

One last type of antecedent are literals defined in background knolwedge provided by the user. These are literals that are not included in the planning program but could be useful in building control rules.

The ability to add background knowledge allows the user to add literals definitions that might not be discovered through normal hill-climbing. For instance, a literal definition may contain a series of calls to planning program predicates whose combination would not be discovered through hill-climbing and the number of predicates involves may be too large to specify efficiently using a relational cliché. Or, a background literal could be used to specify a concept related to a quality metric, but it is not necessary for correct planning. Background definitions may also contain more expressive features such as recursive concepts or disjunctive conditions. For instance, the following recursive-literal definition was found useful in constructing rules for the blocksworld domain:

```
above(Block1,Block2,InitState) :-
    member(on(Block1,Block2),InitState).

above(Block1,Block2,InitState) :-
    member(on(Block1,X),InitState),
    above(X,Block2,InitState).
```

The above(Block1,Block2,InitState) literal lets a rule check if one block is located above another block in the same stack in the problem initial state.

Literals included as background knowledge are made available to Scope by adding a new Prolog clause definition. Rule specializations are generated using these literals by adding all possible variations of each background literal. Variations of literals are generated by matching each variable in the literal to another variable of the same type already present in the current rule. All variables in a background literal must be declared a particular type so they can be matched to other rule antecedent variables. Rule specializations created in this manner are then added to the pool of specializations from which the next rule modification will be selected.

5.3 Program Specialization Phase

Once rules have been learned for when to select each plan refinement, these rules are passed to the program specialization phase. In this phase, the learned control information is incorporated back

```
select-new-op((at-obj(X,Loc),Aid),Steps,Agenda,unload-airplane(X,P,Loc)) :-
    find-init-state(Steps,Init),
    member(at-obj(X,Loc2), Init),
    not(member(same-city(Loc,Loc2),Init)),
    member(airport(Loc), Init).

select-exist-op((at-truck(T,Loc),Aid),Steps,Agenda,init-state)) :-
    find-init-state(Steps,Init),
    member-pred(at-truck(T,Loc),Init),
    not(member((Aid2,drive-truck(T,Loc,Loc2)),Steps),Aid≠Aid2).
```

Figure 5.7: Learned control rules for the logistics domain.

into the original planner. The basic approach is to guard each plan refinement candidate with the learned selection information. This forces a refinement application to fail quickly on plan subgoals to which the refinement should not be applied.

A plan refinement is "guarded" with control information by incorporating the learned control conditions into the clause that represents the selection of that refinement. This process is similar to the one described in Section 3.2.3. A copy of the head of the learned control clause is appended to the end of the original program clause. The learned rule is then added as a program clause. A refinement may have more than one control clause relating to it. Each of these control rules (which can be considered as one disjunctive rule) is then added to the program as a separate clause. A cut ("!") is appended to the body of each newly added control clause since there is no reason to consider multiple proofs of why a particular plan refinement should be used.

A decision is also made as to whether the control information has made the planner deterministic. If a refinement rule covers no incorrect selection decisions in the induction phase, then it is assumed that the rule is fully accurate and no other refinement candidates will need to be considered. This type of rule is marked as deterministic by adding a Prolog cut (!) after the call to the rule clause, which will prevent any backtracking over the refinement selection if all rule conditions are true. If a refinement rule could not exclude all incorrect decisions in the previous phase, then the planner is still allowed to backtrack over the selection of that refinement (i.e. no cut is added). This type of rule can still substantially improve planning efficiency by preventing many inappropriate applications of that refinement.

Figure 5.7 shows two learned rules for the logistics transportation domain. The first rule selects the new action unload-airplane(X,P,Loc) to achieve the goal at-obj(X,Loc) when X is found to be initially located in a different city than the goal location and when the goal location is an airport. The second rule uses the initial state to achieve the goal at-truck(T,Loc) if there does not exist another action in the plan drive-truck(T,Loc,Loc2) that moves the truck to a new location.

5.4 Why Multi-Strategy Learning?

One question brought out by the last few sections, is what happens if a pure induction algorithm were used to learn control rules without using EBG to bias the search? Or, on the other hand, how would a pure EBL approach perform? This section discusses the benefits of a multi-strategy learning approach to the problem of learning search control for planning.

5.4.1 Using Pure Induction to Learn Control Rules

One possible approach to learning control rules for planning is to employ a pure inductive algorithm. For instance, a similar inductive algorithm to the one presented here could be used without using EBG to bias the search. Background information could be provided by using the same pool of operational planning predicates as used by SCOPE to construct rules.

Unfortunately, a pure inductive search in this setting would most likely prove computationally infeasible. Since no generalized proof trees would be constructed, the induction engine would be responsible for binding *all* variables in each learned clause. The cost of searching through all the different variabilizations of all the background predicates would be extremely high, and likely be intractable to perform.

To better explain the cost of a pure inductive search, a brief theory analysis of the FOIL algorithm is presented below. The main element contributing to the size of FOIL's search space is the number of different literals that must be searched through in order to extend the body of a clause. Pazzani and Kibler (1992) show that this number can be bounded by the following formula:

$$NumLits \leq 2 * AllPred * (Old + MaxA - 1)^{MaxA}$$

where $AllPred$ is the total number of available background predicates, Old is the maximum number of old variables in the clause, and $MaxA$ is the maximum arity of any predicate. One can infer from this formula that additional predicates increase the size of the search space a linear amount, while increasing the arity of the predicates increases the size of the search space exponentially. Also, the search space increases exponentially with the number of distinct variables in the clause.

Most past approaches that have utilized a pure inductive search for acquiring control knowledge have only searched through a handful of simple predicates. For instance, GRASSHOPPPER (Leckie & Zuckerman, 1993), was applied to a state-based planner and does an inductive search on whether certain conditions are present in the current world state. Conversely, the operational planning predicates used by SCOPE can be much more complicated and often have a number of arguments (some up to ten). Thus, an inductive search through all possible variablilizations of these predicates would likely prove infeasible, due to the exponential explosion caused by the number of predicate arguments that must be variabilized.

SCOPE circumvents part of this search problem by using information from generalized proof trees to constrain the search to only certain variable combinations. When looking for a new antecedent to add to a clause, many of the literals examined by SCOPE have all variables already bound to other variables in the current clause. Since both the literals and clause heads are extracted directly from generalized proof trees, many variable connections established in these trees are preserved when constructing clauses. Some literals do have unbound variables, so SCOPE may have to search through some different variabilizations. However, since many literal variables are already bound, the number of variabilizations searched by SCOPE will be much less than those searched through by a pure FOIL approach. SCOPE can also use extra background information, which contains predicates not found in planning proof trees, where all variable connections must be made by the induction engine. However the use of these extra background predicates is very limited, so as not to significantly increase the inductive search. Thus, overall, the search space examined by SCOPE is usually significantly smaller than the search space examined by a pure inductive approach.

5.4.2 Using Pure EBL to Learn Control Rules

It is also possible to learn control rules for planning using a purely explanation-based approach. UCPOP+EBL is a learning system that learns control rules for planning by utilizing EBL to explain planning failures (Kambhampati et al., 1996), and is discussed in more detail in Chapter 9. This learning system has been successfully used to acquire rules that improve the performance of UCPOP. However, as mentioned in Section 3.1, rules learned using EBL alone can often be overly-specific and complex where the cost of applying the rules outweighs their savings. In Section 9.2, an experimental comparison of SCOPE and UCPOP+EBL is presented where SCOPE is shown to outperform UCPOP+EBL in several different experiments. In particular, EBL is shown to be effective at improving planning performance for a very expressive domain, while SCOPE, which uses a combination of EBL and induction, is shown to be much more robust at applying to different domain representations.

5.5 Improving Upon Different Planning Metrics

By learning rules that avoid search paths that lead to backtracking, SCOPE can significantly improve the efficiency of of a planner (Estlin & Mooney, 1996). SCOPE can also be used simultaneously to improve the quality of plans by using a particular method to collect training data.

In order to improve plan quality, as well as efficiency, SCOPE is trained on only high-quality solutions. For instance, when using plan length as a quality evaluation metric, SCOPE should be trained on solution plans containing a low number of steps. Or if a minimal plan execution time is the desired quality metric, SCOPE should be trained on solution plan with small execution times. This approach causes SCOPE to collect positive and negative examples of when to apply a plan refinement based on finding a high-quality plan. Thus, it learns rules that not only avoid dead-end paths, but also that avoid paths that lead to low-quality solutions.

In order to train SCOPE on high-quality solutions, several different techniques can be employed to generate training problem solutions. One is to allow the planner to simply search for optimal or high-quality solutions. For example, when plan length is the quality metric, the search method depth-first iterative deepening (DFID) (Korf, 1985) can often be used to guarantee optimal solutions. For other metrics, such as plan execution time or cost, a branch and bound search can be used that focuses on finding low-cost plans for that metric. Or, it may be possible to use an A^* star search (Hart, Nilsson, & Raphael, 1968) if there exists an admissible heuristic for that quality metric.

Another technique is to employ a human domain expert to solve the training problems. When presented with a problem, a domain expert can often easily provide a high-quality solution. In many real-world settings, this is a viable option, since a domain expert is often available.

5.6 Improving Learning Time

Two additional features have been added to SCOPE to keep learning time at a tractable level. These are caching useful relational clichés and employing a method for incremental training.

5.6.1 Caching Relational Clichés

Relational clichés can be very valuable in finding accurate rules, unfortunately, they can also dramatically increase the inductive search space. For each cliché used, there may be a number of different literal combinations that fit the specified pattern. In order to decrease this extra search, relational clichés are only considered for small numbers of training examples or when an accurate rule cannot be learned through standard hill-climbing. Specifically, SCOPE uses the following procedure. For small numbers of training examples, SCOPE searches through all relational cliché combinations and caches any useful instantiations of the more general patterns. Then once the number of training examples exceed a predefined limit[4], only cached clichés are immediately considered for addition to a clause. SCOPE searches through all relational cliché patterns only if no *single* literal or cached pattern is found to have any gain. Often, SCOPE can find many useful relational combinations early in training and by caching these, SCOPE can avoid the need for much extra search later in the training process.

5.6.2 Incremental Training

SCOPE has also been designed to run in an incremental training mode, which can often save time and help the system to be less memory-intensive. SCOPE was originally designed to run as a batch system where all training examples were input to the system at the same time. However, in order to create a more efficient system, SCOPE has been modified to run in an incremental mode where smaller sets of training examples are input to SCOPE. In this mode, SCOPE reads in a small set of examples, learns control rules for these examples, and then reads in a new set of training examples to examine. Any learned control rules can be saved and then re-used in later trials. This process works in the following manner. For each set of training examples, SCOPE caches any learned rules. In the next trial, before performing induction to build new rules from scratch, any cached rules are first tested. If any of the cached rules are found to be fully accurate on the current set of refinement-selection examples, then the rule with the highest gain is selected (i.e. the rule which covers the largest number of positives). Any positive refinement-selection examples covered by this rule are then removed and the process is repeated until no accurate cached rules can be found or all positive refinement-selection examples have been covered. If any positive examples remain, then a new rule (or set of rules) is built from scratch to cover them.

It is often the case that a very good control rule is learned based on a very small number of train examples. This technique exploits this fact by caching rules built after only a small amount of learning and then re-using them when training on larger numbers of examples without incurring large amounts of learning time. Training in an incremental mode can thus greatly cut down on learning time, especially when many good rules are learned for small numbers of training examples.

[4]This limit was set at 10 or 20 examples in the experiments discussed in this dissertation.

5.7 Summary

This section presented the SCOPE learning system, which automatically acquires search-control knowledge for planning systems. This knowledge is acquired in three main phases. In the first phase, SCOPE solves any input training examples using the original planning program and extracts two pieces of control information. The first is a set of refinement selection examples that record when plan refinement were successfully and unsuccessfully applied in solving the training examples. The second is a set of generalized proof trees which provide a background context that explains the success of correct planning decisions. These are constructed by applying explanation-based generalization techniques to proofs of the training examples. In the second phase of the algorithm, these two pieces of information are used to construct control rules. Here, the FOIL induction algorithm is employed to construct a definition for when to apply each planning refinement. Information in the generalized proof trees is used to bias the FOIL search towards useful control information. Once a set of rules has been constructed for when to apply each refinement, the rules are passed to the third phase of the algorithm which incorporates them into the original planner and outputs a new planning program.

The next few chapters presents results demonstrating that SCOPE can effectively learn control knowledge that improves planner performance in several different domains.

Chapter 6

Experimental Evaluation - Improving Efficiency and Plan Length

This section describes a number of experiments that test SCOPE's ability to improve both the efficiency of a plan and also the plan quality metric of solution length. Experiments are presented in two domains where SCOPE is shown to significantly improve upon both these factors. SCOPE is also shown able to scale gracefully to increasingly hard problems.

6.1 Domain Descriptions

The logistics transportation domain of Veloso (1992) was adopted for one set of experiments. In this domain, packages must be delivered to different locations in a number of cities. Packages are transported between different cities by airplane and between locations in the same city by truck. For the second domain, the standard set of blocksworld operators from Nilsson (1980) were used. The logistics domain contains six operators and blocksworld domain contains four. Operator descriptions for both domains can be found in Appendix A.

In both the logistics and blocksworld domain, plan quality is measured by the length of the plan, i.e. shorter plans are considered more desirable. Thus, the goal in these domains, was to improve planning efficiency and to produce solutions of minimal length.

6.2 Focusing SCOPE on Improving Efficiency and Plan Length

In order to improve plan quality, as well as efficiency, SCOPE is trained on only high-quality solutions. For the blocksworld domain, DFID was used to generate optimal training-problem solutions of minimum length. However, for the logistics domain, DFID could not solve most of the training problems in a reasonable amount of time. For this domain, a bootstrapping technique was used to acquire high-quality solutions for the training problems. First, a training set of simpler problems was generated that could be efficiently solved by DFID. For this case, problems containing only three packages were used. SCOPE was trained on these problems and a set of control rules was generated. These rules were then used to solve problems in the actual training set (containing 5-7 packages). In the majority of the 5-7 package problems, this technique was able to produce optimal (or near-optimal) solutions[1]. About 3% of the problems could not be solved by the control rules. Optimal solutions for these problems were entered by hand. This training approach allows SCOPE to be easily trained on optimal (or near-optimal) solutions for harder problems, with very little help from the user.

6.3 Experimental Design

The goal of these experiments was to improve both planner efficiency and also the quality of generated solutions. Training and test problems were generated for each domain by using a random problem generator which produced random initial and final states. In the logistics domain, states were produced by placing packages in random locations in different cities. Problems in this domain contained between five and seven packages, two trucks and two planes, which were distributed among locations in two cities. In the blocksworld, problems were generated by using the technique described in Minton (1988), where each block is placed either on the table, on top of another block, or is held by the arm. Problems contained between three and six blocks and had between one and four goals.

In both domains, SCOPE was trained on separate examples sets of increasing size. Since plan length was used as the quality metric in these domains, SCOPE was always trained on solutions of minimal length. In blocksworld, these were generated by using DFID to solve the training problems, and in logistics, these were generated by using a bootstrapping technique to solve the training problems, as explained in the preceding section. In each domain, a test set of 100 independently generated problems was used to evaluate performance. Five trials were run for each training set size, after which results were averaged. No time limit was imposed on planning in either domain, but a uniform depth bound on the plan length was used during testing that allowed for all problems to be solved. In the blocksworld the depth bound used was set at 8 and in the logistics domain, the depth bound was set at 100. Tests in the blocksworld were performed on an Ultra Sparc 2 and in the logistics domain on an Ultra Enterprise 5000.

For each trial, SCOPE learned control rules from the given training set and produced a modified planner. Since SCOPE only specializes decisions in the original planner, the new planning program is guaranteed to be sound with respect to the original one. Unfortunately, the new planner is not guaranteed to be complete. Some control rules could be too specialized and thus the new planner may not solve all problems solvable by the original planner. In order to guarantee the completeness of the final planner, a strategy used by Cohen (1990) is adopted. If the final planner fails to find a solution to a test problem, the initial planning program is used to solve the problem. When this situation occurs in testing, both the failure time for the new planner and the solution time for the original planner are included in the total solution time for that problem.

For comparison purposes, one other search method was used to solve the test problems. Best-first search was also evaluated at solving the test problems. These tests were done using the standard Lisp implementation of UCPOP (Barrett & et al., 1995), thus some results could be partially due to implementation difference between Prolog and Lisp.

[1]These solutions are not *guaranteed* to be optimal since it is possible the rules are too specialized and could miss an optimal solution. However, a number of problems were checked by hand, and in all cases optimal solutions had been produced.

6.4 Results

Figure 6.1 and Figure 6.2 show the measured improvements in planning efficiency for both domains. The times shown represent the number of seconds required to solve all of the problems in the test sets after SCOPE was trained on a given number of examples. The best performance occurred when the planner utilized the learned control information. In these tests, SCOPE was able to produce a new planner that was significantly faster than the original depth-first planner. In the logistics domain, the new planner was an average of 8 times faster than the original planner and in the blocksworld domain it was an average of 23 times faster. Also, at all points where control knowledge was added, no backtracking occurred in either new planner.

In both domains, especially the logistics domain, SCOPE was able to learn some very good control rules with only a few training examples, and thus could immediately produce significant speedup. Then, as more and more training examples are seen, SCOPE picks up more specialized control rules that help reduce the planning times either further. Overall, in the logistics domain, an average of 20 rules were learned and in the blocksworld an average of 19 rules were learned.

Best-first search performed significantly worse than both depth-first search and SCOPE on both the logistics and blocksworld domains. In the blocksworld domain, best-first could solve all of the test problems, however the planning time required was greater than that of the depth-first Prolog planner. In general, the best-first search seemed able to solve a large number of problems very quickly, but there were a small number of problems for which it had great difficulty in finding a solution. In the logistics domain, best-first could solve only three of the test problems using a search-limit of 200,000 plans. The final planning time was over fifteen hours and was thus too large to display in the graph.

Plan quality improvements in these domains are shown in Figures 6.3 and 6.4. The lengths shown in the graphs represent average solution lengths returned for the test problems. In the blocksworld, the average length of all optimal solutions is also shown (which was produced by using DFID). Optimal solutions for the logistics problems could not be generated in a reasonable amount of time so this average could not be calculated. In the logistics domain, SCOPE was able to return significantly shorter solutions than those produced by using the original depth-first planner; average solution lengths are reduced from 35.2 steps to 25.1 steps. As already mentioned, best-first search could only solve a few problems in this domain and thus the average plan lengths for this search method are not shown. In the blocksworld domain, SCOPE was able to produce a new planner that returned near-optimal solutions, and again, returned significantly shorter solutions than those returned by depth-first alone. In this domain, SCOPE reduced average solution lengths from 5.07 to 3.91 where the average solution lengths of optimal solutions was 3.70. Best-first search generated average solutions lengths of 3.78 and thus did not achieve optimal solutions, but did produce slightly better solutions than those produced by SCOPE.

As mentioned earlier, the new planner produced by SCOPE is guaranteed to be correct but it is not guaranteed to be complete. It is possible that the control rules produced by SCOPE could be too specialized and might prevent the planner from solving all problems. However, for most experiments presented in this dissertation, this problem occurred very infrequently; after training, the new planner could usually solve between 97%-100% of the test problems that could be solved before learning. And as shown in the next chapter, the new planner can sometimes solve a much higher percentage of test problems than the original planner.

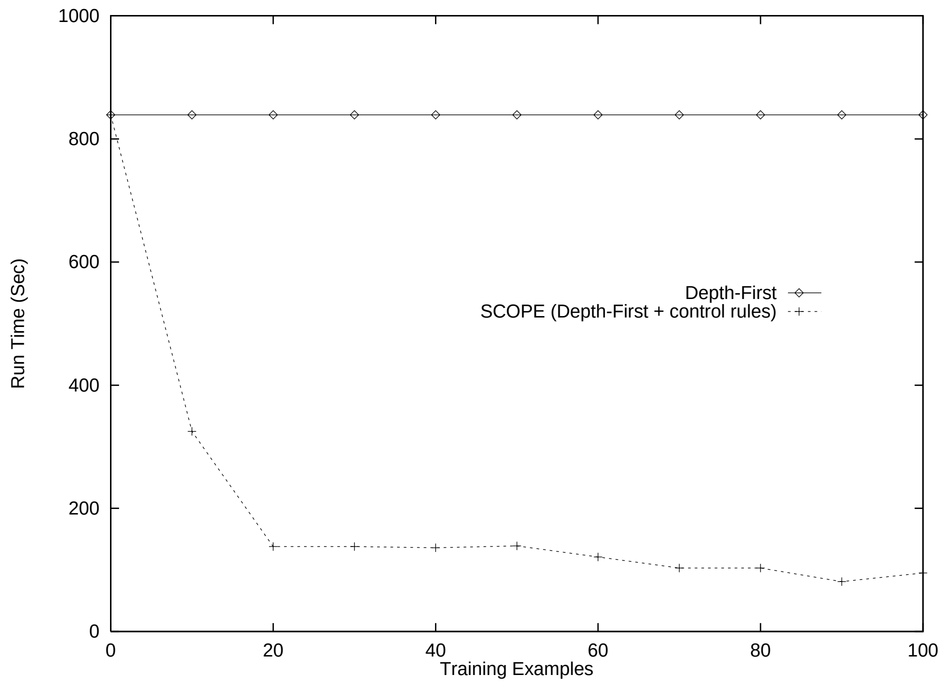


Figure 6.1: Efficiency performance in the logistics transportation domain.

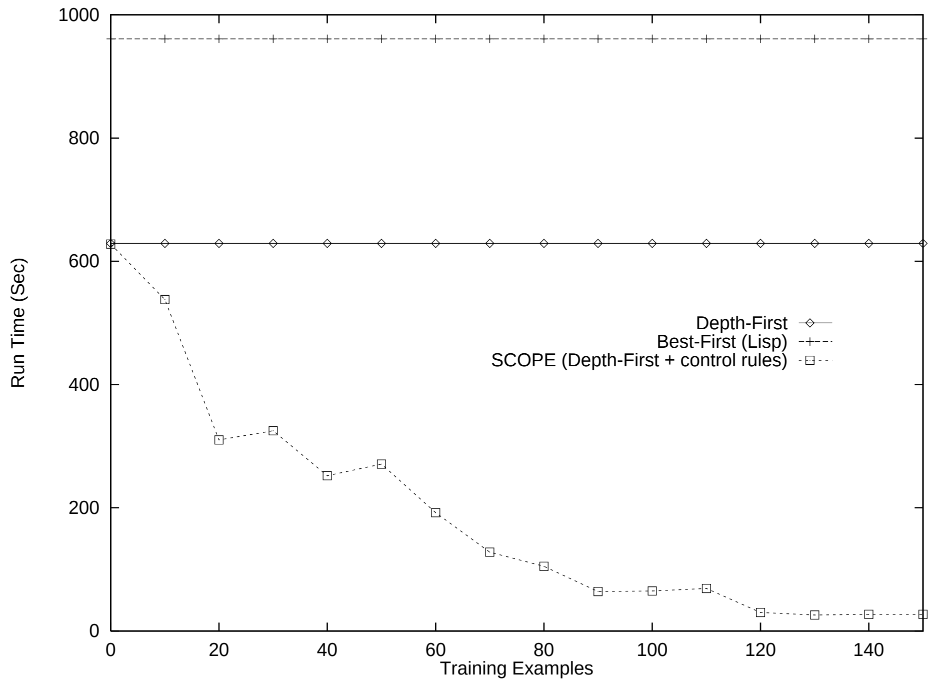


Figure 6.2: Efficiency performance in the blocksworld domain.

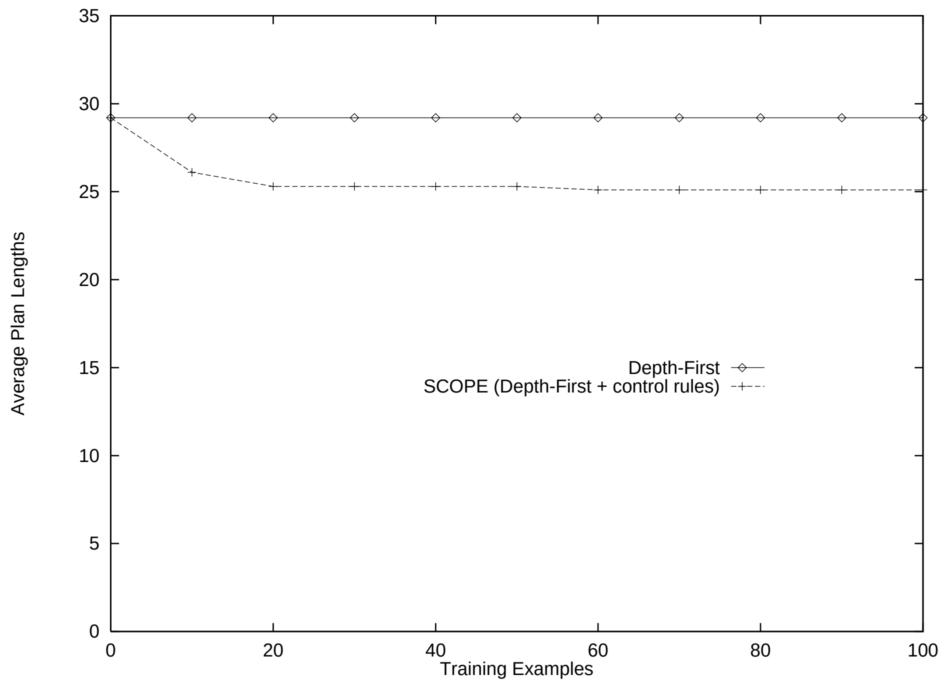


Figure 6.3: Quality performance in the logistics transportation domain.

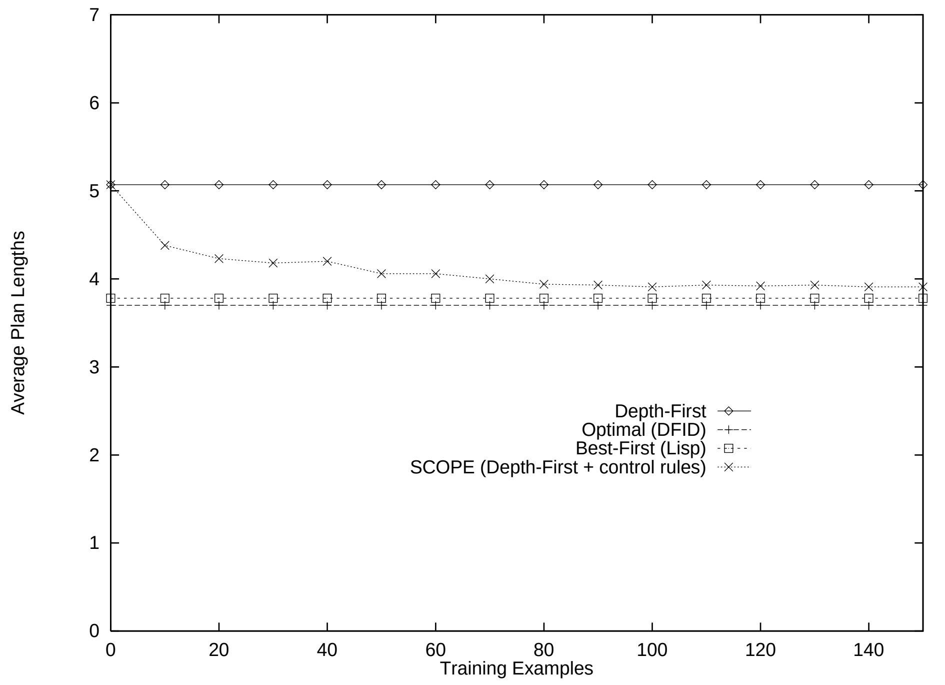


Figure 6.4: Quality performance in the blocksworld domain.

Figures 6.5 and 6.6 show the percentage of test problems that could be solved by the new planner for the experiments just presented. Each graph shows the percentage of test problems solved after training on a certain number of problems. In the logistics domain, the new planner was able to cover around 80% of the test problems after begin trained on just a few examples. Once the learner saw up to 100 training examples, the new planner was covering close to 100% of the test problems. In the blocksworld domain, the new planner was also able to quickly cover a large percentage of the test examples and was again able to cover close to 100% after all training had completed.

6.5 Scalability

Another set of experiments were performed to evaluate how Scope performed on harder problems. These tests were done using increasingly complex problems in the logistics transportation domain. Two different experiments were performed, one where Scope was was trained on simple problems, and one where Scope was retrained on slightly harder problems. In both experiments, results were evaluated using several different test sets of increasing complexity. The goal of these experiments was two-fold; one, to see how well rules learned on simpler problems scaled to harder problems and two, to see what differences arose when Scope was trained on harder problems.

6.6 Training on Simple Problems

In the first experiment, Scope was training on 100 training problems in the logistics domain that contained three packages, two planes, and two trucks, which were distributed among locations in two cities. Solutions were generated using DFID, since as mentioned previously, problems at this level of complexity could be solved by DFID in a reasonable amount of time. Scope's performance after training on these relatively simple problems was then evaluated, to see how well the control rules learned by Scope could scale to problems of different complexity. Six different test sets of increasing complexity were used to perform this evaluation where the number of packages (and goals) ranged from 1 up to 50. During testing, a time limit of 500 seconds per problem was used. No limits on solutions length were imposed on test problems in these experiments, however, a depth bound was used that ensured all test problems could be solved. The depth bound ranged from 12 steps for 1 package problems up to 350 for 50 package problems.

The results from this first experiment are shown in Table 6.1 and Table 6.2. The first set of columns in each table shows the complexity level of each test set. In Table 6.1, the second set of columns shows the percentage of test problems that could be solved within the time limit (using depth-first search) before and after learning. The last set of columns reports the planning time required to solve the test problems. In Table 6.2, the second set of columns shows the average solution lengths for *all problems solved*. These numbers show how long the average solutions were for all test problems that could be solved by the new planner. The last set of columns shows the average solution lengths only for *problems that could be solved by the original planner* (without control information). These last numbers were included to show how Scope improved upon solution quality. Thus these numbers were only gathered for problems that could be solved both by the original planner and by the new planner so that changes in solution length could be observed. Also

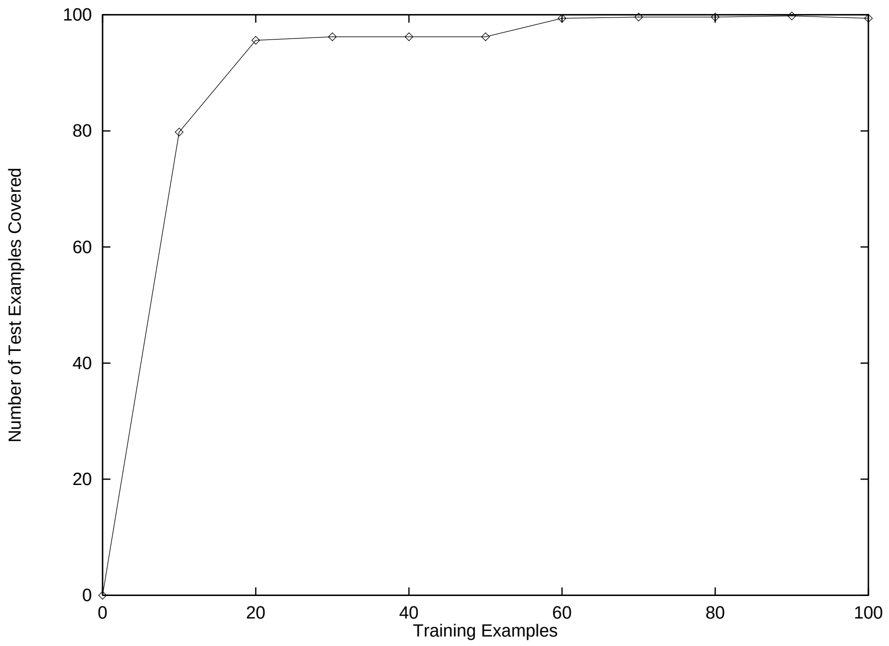


Figure 6.5: Number of test problems covered by the new planner in the logistics transportation domain
.

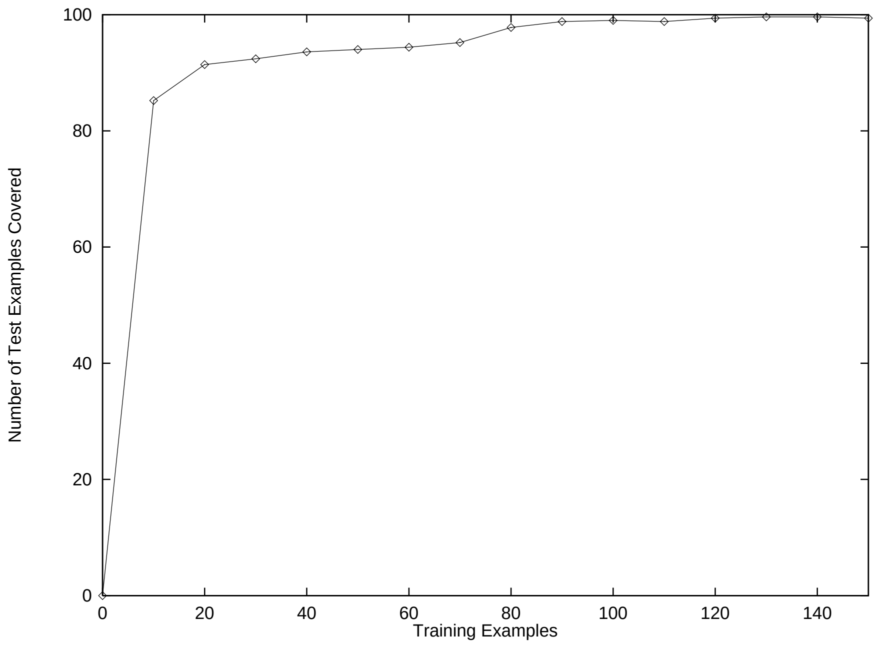


Figure 6.6: Number of test problems covered by the new planner in the blocksworld domain.

Test Sets		**Problems Solved** *(%)*		**Solution Time** *(Secs)*	
Num. of Goals	*Num. of Problems*	*W/o Rules*	*With Rules*	*W/o Rules*	*With Rules*
1	100	100	100	3.9	1.8
2	100	100	100	18.6	4.8
5	100	100	100	374	62.8
10	100	100	100	4112	906
20	100	30	85	45870	10005
50	100	0	24	50000	46201

Table 6.1: Efficiency results on increasingly complex test problems in the logistics domain.

Test Sets		**Plan Lengths** *(All problems solved)*		**Plan Lengths** *(Only problems solvable by original planner)*		
Num. of Goals	*Num. of Problems*	*W/o Rules*	*With Rules*	*W/o Rules*	*With Rules*	*Optimal Lengths*
1	100	6.4	5.7	6.4	5.7	5.7
2	100	11.2	9.8	11.2	9.8	9.8
5	100	25.8	22.5	25.8	22.5	-
10	100	45.0	39.0	45.0	39.0	-
20	100	76.0	69.9	76.0	66.6	-
50	100	-	162.7	-	-	-

Table 6.2: Quality results on increasingly complex test problems in the logistics domain. Results are shown both for all problems that could be solved by the new planner and for problems that could only be solved by the original planner.

included in Table 6.2 are the average optimal solution lengths for some of the simpler test sets. Dashes indicate a number was not available due to the fact that many problems could not be solved by the original planner using depth-first or DFID.

For all six test sets, SCOPE was able to improve planning efficiency, and when possible, increase the percentage of problems solved. For instance, on problems containing 5 packages, SCOPE was able to create a new planner that was 5.9 times faster than the original planner and on the problems containing 20 packages, SCOPE was able to create a new planner that was 4.5 times faster and it also improved the percentage of problems solved from 30% to 85%. In 50 package problems, relatively little speedup was realized, however, the number of test problems solved increased form 0% to 24%.

Unfortunately, it was difficult to gather data on whether plan quality was always improved in all test sets. For many examples in the hardest two test sets, the original planner could not find a solution under the time limit, and thus solutions lengths before and after learning could not be compared. For the first two test sets (containing 1 and 2 package problems), SCOPE was able to significantly improve final solution quality and always generated optimal solutions. For the third, fourth and fifth test sets (containing 5, 10 and 20 package problems), SCOPE was also able to improve final solution quality. For instance, for the 20 package problems that could be solved by the original planner, SCOPE improved the average solution length from 76.0 to 69.9. In the last test set (containing 50 package problems) no problems could be solved by the original planner, so plan quality improvement could not be measured.

Test Sets		**Problems Solved** *(%)*		**Solution Time** *(Secs)*	
Num. of Goals	*Num. of Problems*	*W/o Rules*	*With Rules*	*W/o Rules*	*With Rules*
1	100	100	100	3.9	1.8
2	100	100	100	18.6	4.8
5	100	100	100	374	44.0
10	100	100	100	4112	624
20	100	30	94	45870	6128
50	100	0	22	50000	46917

Table 6.3: Efficiency results on increasingly complex test problems in the logistics domain.

6.7 Training on Harder Problems

In the second experiment, the bootstrapping method presented in Section 6.3 was used to train Scope on harder problems. This experiment was intended to test whether training Scope on harder problems could improve results even further. For this experiment the following procedure was used to generate training examples. First, the rules from the experiment just presented were used to solve harder training problems. These new training problems contained 5-7 packages, as opposed to the problems in the previous experiment which contained only 3 packages. As in Section 6.3, the learned rules were able to solve around 97% of the new training problems. Optimal solutions for any unsolved problems were then entered by hand.

Scope's performance was then tested again on the same six sets of test problems. The results from this second experiment are shown in Tables 6.3 and 6.4. Again, for all six test sets, Scope was able to improve planning efficiency, and when possible, increase the percentage of problems solved. Also, improvements were greater than those seen in the previous experiment. For instance, after training on 3 package problems, Scope increased the coverage of 20 package problems from 30% to 85%. However, after training on the 5-7 package problems, this coverage increased to 94%. Similarly, greater speedup was also realized. Speedup on the 10 package problems increased from 4.5x to 6.6x, and on the 20 packages problems from 4.5x to 7.5x.

On the 50 package problems, solution coverage actually slightly decreased from 24% to 22%. This decrease is most likely caused by the use of a time limit. As discussed in (Etzioni & Etzioni, 1994), using a time limit can decrease speedup, especially when solution times are close to the limit. In these problems, solution times were often very close to the time limit (of 500 seconds). After training on harder problems it's possible that solution times in several of these problems got slightly longer causing the time limit to be reached, while solution times in other (harder) problems might have improved. However, since these harder problems could not be solved under the time limit, this speedup is not reflected in the reported results.

Quality improvements in general remained similar. Scope was still able to produce optimal solutions for the 1 and 2 package problems. After training on 3 package problems, Scope improved the average solution lengths of 5 package problems from 25.8 to 22.5 and after training on 5-7 package problems, they were improved to 22.4. For 10 package problems, lengths were originally improved from 45.0 to 39.0 in the first experiment; and after training on the 5-7 package problems, they were improved to 38.8. For 20 package problems, solution lengths remained the same.

Thus, training on harder problems did help to further improve planning efficiency, however, it had very little effect on quality. One possible reason for is that the solutions produced in the first

Test Sets		**Plan Lengths** *(All problems solved)*		**Plan Lengths** *(Only problems solvable by original planner)*		
Num. of Goals	*Num. of Problems*	*W/o Rules*	*With Rules*	*W/o Rules*	*With Rules*	*Optimal Lengths*
1	100	6.4	5.7	6.4	5.7	5.7
2	100	11.2	9.8	11.2	9.8	9.8
5	100	25.8	22.4	25.8	22.4	-
10	100	45.0	38.8	45.0	38.8	-
20	100	76.0	70.6	76.0	66.6	-
50	100	-	163.3	-	-	-

Table 6.4: Quality results on increasingly complex test problems in the logistics domain. Results are shown both for all problems that could be solved by the new planner and for problems that could only be solved by the original planner.

experiment (where Scope was trained on 3 package problems) might have already been optimal or close to optimal and thus there was little room for further improvement. In the harder test sets, where further quality improvements could be possible, very few problems could be solved by the original planner, and thus quality improvements were difficult to measure.

6.8 Ablation Results

One other set of tests was run to evaluate the different features of Scope's induction algorithm. These experiments test what effect the different antecedent types listed in Section 5.2.3 have on Scope's performance. These tests were done using the problem sets for the logistics transportation domain that were utilized for the first set of experiments from this section. Thus training and test problems contained between 5-7 packages distributed among locations in two cities, and the same high-quality solutions to the training problems, which were generated previously by a bootstrapping method, were also utilized.

Two experiments were run to test the importance of different antecedent types. In the first experiment, Scope's induction algorithm could use literals taken straight from generalized proof tree, but could not use any other types of these literals, such as negated versions of proof tree literals, relational clichés, etc. In the second experiment, the induction algorithm could use a few other types of literals, including negated literals, determinate literals and non-codesignation constraints. However, it could not use relational clichés or extra background predicates, which can incur the most extra search. For both these experiments, both efficiency and quality improvements were measured. Also, Scope's performance using all types of literals is compared against the results for these tests.

Figure 6.7 shows the measured improvements in planning efficiency. Similar to previous results, the times shown represent the number of seconds required to solve all of the problems in the test set after Scope was trained on a given number of examples. The best performance occurred when Scope could utilize all different types of antecedents. When using only regular proof tree literals, Scope is able to get some improvement in efficiency but it is much smaller than when using different types of literals. The speedup incurred when using only regular proof tree literals was 1.5x, while a speedup of 8.8x was produced when using all of the different types of literals. Also, it took longer to converge to a good set of rules in this experiment, and after training

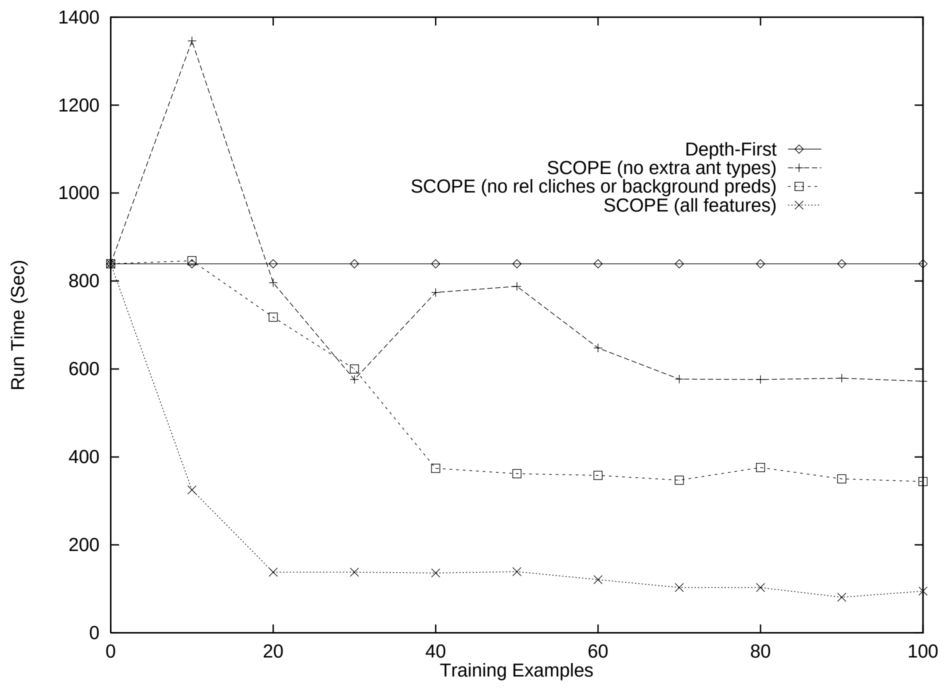


Figure 6.7: Ablation results for efficiency in the logistics transportation domain.

on a small number of examples Scope actually decreased planning efficiency.

In this experiment, Scope was often not able to construct accurate control rules, which don't cover any negative examples, and thus it sometimes could not add a cut after a refinement selection. This causes slower planning times since more backtracking occurs. Also when the new planner was too specialized to solve the test problems, it took much longer to revert back to the old planner to produce a solution, which accounts for the much longer planning times early in the training process.

In the second experiment, where Scope could use negated literals, determinate literals and noncodesignation constraints, Scope was able to further improve planning efficiency, but still performed worse than when it could access relational clichés and other background predicates. In this experiment, Scope was able to achieve a speedup of 2.4x. For this experiment, a better set of control rules were learned than for the first experiment, however there were still some points where Scope had problems learning fully accurate rules.

Figure 6.8 shows the measured improvements in plan quality, where the lengths shown in the graph represent average solution lengths returned for the test problems. Again, in the first two experiments, where Scope did not use all of the different antecedent types, it performed worse than when using standard Scopewhere all literal types were utilized. In the first experiment, where Scope could only use regular proof tree literals, it was able to reduce average solution lengths from 29.2 steps to 28.6 steps. Thus control rules were able to direct Scope to only slightly higher quality solutions. Since some rules covered negative examples, the planner was not always directed towards the best solution paths. In the second experiment, where Scope could use some extra features, it was able to further reduce average solutions lengths from 29.2 steps to 27.7 steps. Finally, when using standard Scope, where all extra features were used, control rules were better able to direct

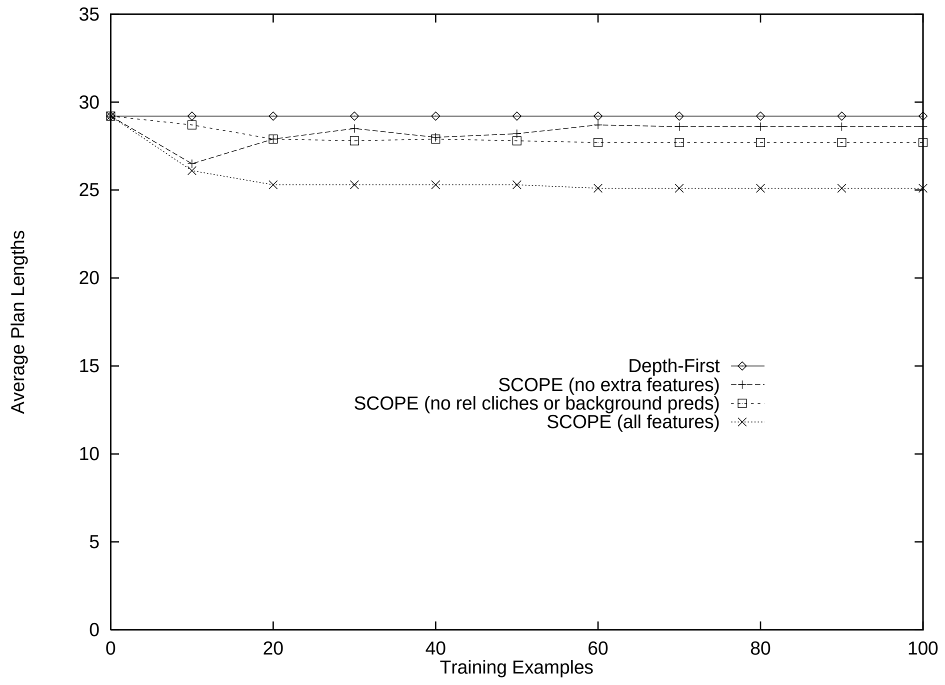


Figure 6.8: Ablation results for quality in the logistics transportation domain.

the planner towards high-quality solutions; in this test, the average solution lengths of the test problems was reduced to 25.1 steps.

Thus, these ablation tests show that for both efficiency and quality improvements, the use of different antecedent types is an important factor in achieving good performance improvements.

6.9 Learning Time

The efficiency results presented in this chapter only considered initial and final planning time and did not include learning time. This section discusses learning time for these experiments and and how it incorporates into previous results.

Learning times from the logistics and blocksworld experiments presented in Section 6.3 are shown in Figure 6.9 and Figure 6.10. In the blocksworld domain, the final learning time after training on 150 examples was around 1500 seconds. In the logistics domain, final learning time was 9576 seconds. (Also, this only includes the time used to learn control rules for the 5-7 package problems. This does not include time required to learn control rules for the 3 package problems, which were used to help solve harder training examples. If this time is included the final learning time for logistics is 122,770 seconds.)

Thus, learning time can be significant for the SCOPE learning system. However, its drawbacks do not outweigh the overall benefits provided by SCOPE. First, learning only has to be performed *once*, and the new planner generated can then be used to solve countless new problems. Thus the extra time spent in the learning process can easily be justified. To further this argument, an amortization analysis was performed that evaluated how quickly a gain would be realized for each domain. In other words, this analysis estimates how many problems the system must solve to

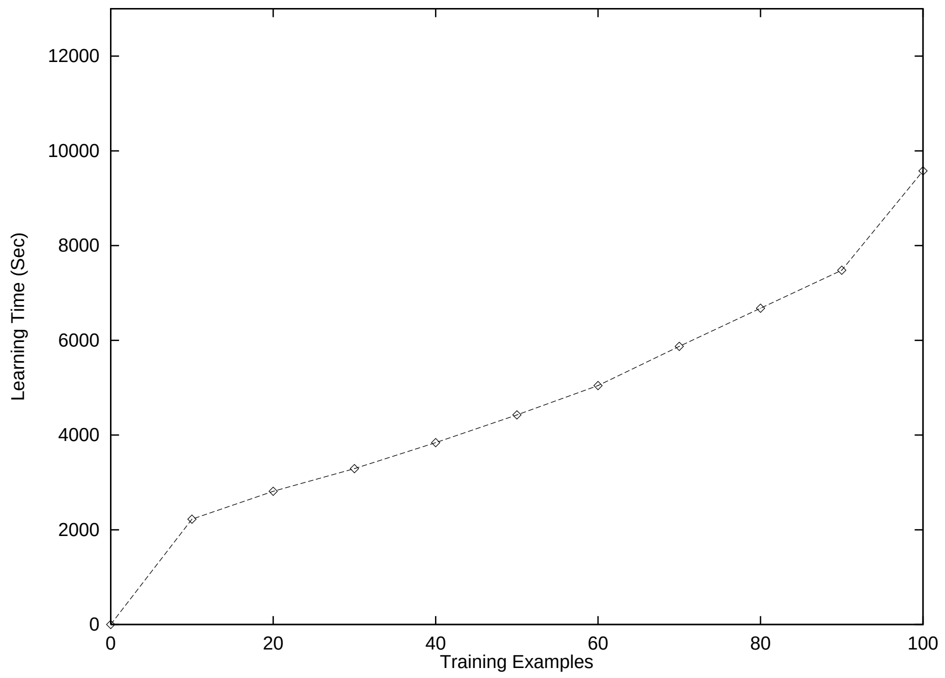


Figure 6.9: Learning time in the logistics domain.

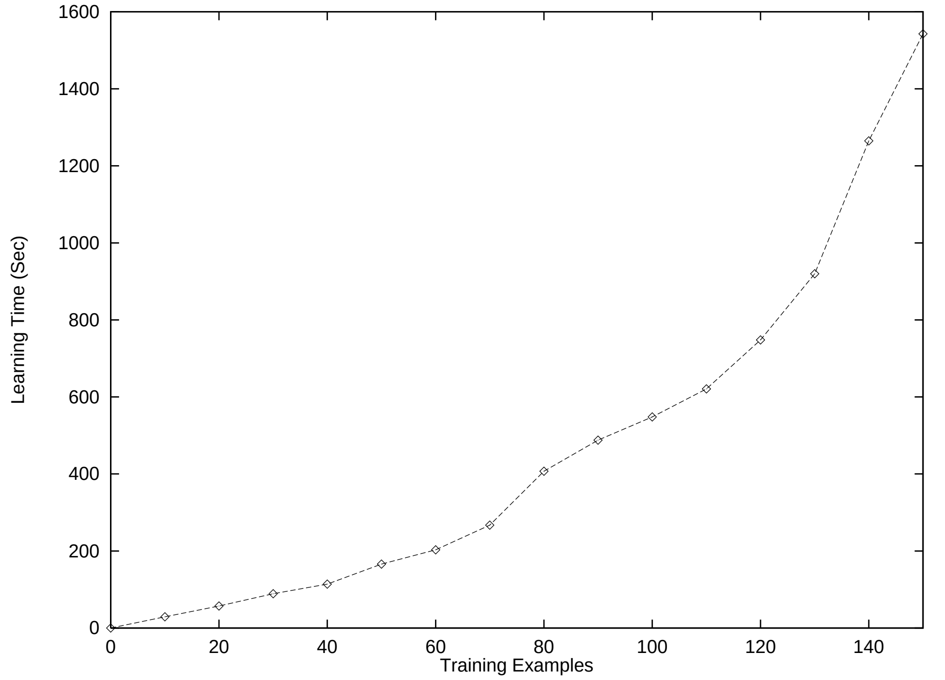


Figure 6.10: Learning time in the blocksworld domain.

make up for the cost of learning time. The following results were produced. For the blocksworld, a gain in planning time is realized after running on at least 250 test examples. In the logistics domain, a gain is realized after running the new planner on at least 1300 examples. (If including time to learn rules for the 3 package problems, a gain is realized after running the new planner on at last 1500 examples.) Thus, in the blocksworld, a relatively small number of problems must be solved before a gain in efficiency is first seen. In logistics, a larger number of problems must be solved to achieve a gain, however, for most real-world applications, this number could be quickly reached. In addition, when tested on increasingly complex problems in this domain, the new planner produced by SCOPE was able to significantly increase the number of test problems that could be solved.

Second, SCOPE is learning to produce *near-optimal* or *optimal* solutions for most novel problems. When using just the base planner (without control information), solutions were often of much lower quality. For instance, in the blocksworld, solutions were 25% longer when using the original planner as opposed to the new planner, which includes control information, and solutions were 14% longer in the logistics domain. Plus, the time required to generate optimal solutions with the base planner is usually much greater than the time to generate just valid solutions. For instance, in the logistics domain, when using a DFID search, the planner using a could still not generate optimal solutions for many 5 package problems even when run for several days.

6.10 Summary

This section presented experimental results for using SCOPE to improve both planning efficiency and the quality metric of solution lengths. Results are presented in two domains, the logistics transportation domain and the blocksworld domain. In these experiments, SCOPE is shown to produce significant improvements. For both domains, the new planner produced by SCOPE was significantly faster than the original planner. Also, SCOPE was able to produce shorter solution plans.

Another set of experiments is also presented that shows SCOPE's ability to scale to increasingly complex problems. These experiments were performed in the logistics domain where SCOPE was shown to produce a new planner that could efficiently solve many problems containing 10, 20 or 50 packages. For all levels of problem complexity tested, SCOPE was able to produce a faster planner, and when possible SCOPE significantly increased the number of test examples that could be solved by the planner under a time limit. SCOPE also produced improvements in plan quality when applied to harder problems. For all test sets, SCOPE was able to improve solution quality for problems that could be solved by the original planner. Additionally, it was shown that at least for simple problems (where optimal solutions were known) that SCOPE was generating optimal solutions.

Chapter 7

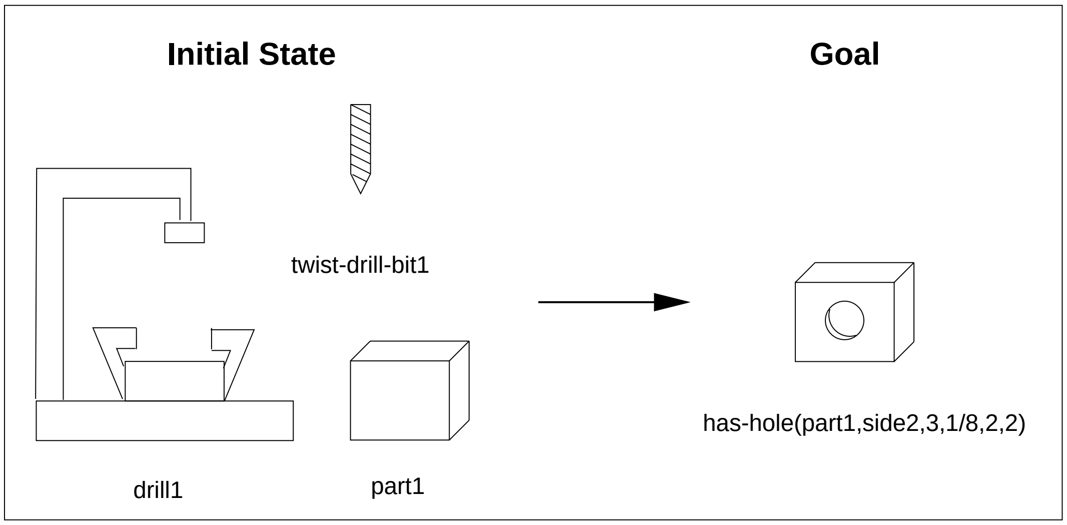
Experimental Evaluation - Improving Efficiency and Plan Cost

This section describes two experiments that test Scope's ability to improve the efficiency of a plan and the plan quality metric of solution cost, and also that test Scope's ability to improve these metrics in a complex, realistic domain. Experiments are done in a large-scale domain where solution cost is not always dependent on solution length; i.e. sometimes a longer solution is a higher quality one. Two different quality metrics for this domain are defined and Scope is shown able to improve on either one, as well as still improving planner efficiency.

7.1 Domain Description

The domain used for these experiments is the process planning domain created by Gil (1991). This is a large-scale complex domain that describes a number of manufacturing processes (e.g. machining, joining, and finishing). *Process planning* can be described as one of the intermediate steps of production manufacturing (Gil, 1991; Pérez, 1995). The first stage of production manufacturing is product design, which involves producing a specific model of a product that satisfies a desired set of specifications. When the design is completed, the next step is to plan the sequences of processes that will be performed on different product components, or *parts*. The parts are then manufactured according to the production plans and finally, the parts are assembled to produce the final product.

The focus of the process planning domain is to automate the *production step* of production manufacturing. Problems in this domain involve finding a sequence of operations that will produce a particular part design. For instance, one possible design goal is to produce a part that has a hole drilled on one side. This hole usually must be of a certain depth and width, and must be drilled in a particular location on the part. An example of such a goal and a sample solution plan that achieves the goal is shown in Figure 7.1. This goal requires a hole to be drilled in side 2 of part 1. (Each side of a part is numbered, thus a rectangular part has 6 sides). Also, this hole must be of depth 3, be of width 1/8, and be in location (2,2) on side 2 of the part. The sample solution plan shows a valid plan for achieving this goal. In this plan, the part is first cleaned and placed in the drill. A spot-drill bit is then used to mark the location of the hole and finally, a twist-drill bit is used to drill the hole. Other possible goal specifications in this domain include producing a part of a particular shape, giving a part a specific metal coating or polish, and smoothing the surface of a



Solution Plan
put-on-machine-table(drill1,part1)
clean(part1)
hold-with-vise(drill1,vise1,part1)
put-in-drill-spindle(drill1,spot-drill1)
drill-with-spot-drill(drill1,spot-drill1,part6,hole1,side1)
remove-tool(drill1,spot-drill1)
put-in-drill-spindle(drill1,twist-drill1)
drill-with-twist-drill(drill1,twist-drill1,part6,hole1,side1,3,1/8,2,2)

Figure 7.1: Example of a problem and sample solution plan in the process planning domain. The goal of this problem is to have a hole of a certain size drilled in a part at a certain location.

part to an adequate level.

For each type of design goal, there are often a variety of alternative processes that can be used. For instance, there are usually a number of different type drill bits that can be used to drill a hole, and a drilling operation can be performed by different machines. In the implementation of this domain used in Gil (1991), machining operations (e.g. drilling, polishing, cutting) are given in the form of planning operators. An example of an operator for drilling is shown in Figure 7.2. This particular operator is for drilling with a *twist-drill* bit in the drill machine. The operator has a number of preconditions that must be satisfied before it is applied, including that the drill machine being used must be holding the part and that the diameter of the drill-bit used to drill the hole matches the width specified in the has-hole goal. Also represented in the process planning domain definition are operators for setting up parts and tools, securing parts in holding devices, cleaning parts, and removing metal burrs from the surface of a part. In total, this implementation of process planning contains 81 operators. Processes represented in this domain are quite complex and can have many interactions, which can cause quite long planning times.

```
drill-with-twist-drill (?Machine,?Drill-bit,?Holding-device,?Part,?Hole,?Side,
                        ?Hole-depth,?Hole-diameter)
Preconditions:
      is-a(?Part,part),
      is-a(?Machine,drill),
      same(?Drill-bit-diameter,?Hole-diameter)
      diameter-of-drill-bit(?Drill-bit,?Drill-bit-diameter)
      is-a(?Drill-bit,twist-drill)
      has-spot(?Part,?Hole,?Side,?Loc-x,?Loc-y)
      holding-tool(?Machine,?Drill-bit),
      holding(?Machine,?Holding-device,?Part,?Side)
Add List:
      has-burrs(?Part),
      has-hole(?Part,?Hole,?Side,?Hole-depth,?Hole-diameter,?Loc-x,?Loc-y)
Delete List:
      is-clean(?Part),
      has-spot(?Part,?Hole,?Side,?Loc-x,?Loc-y)
```

Figure 7.2: Drilling operator from the process planning domain.

7.2 Plan Quality in the Process Planning Domain

Plan quality in the process planning domain can be extremely important. Goals such as minimizing execution time or reducing the number of resources used can be a crucial part of production manufacturing. There are a number of different quality metrics that can be measured for this domain (Pérez, 1995), including:

- The number of plan steps
- The total execution time of the plan
- The dollar cost of the plan
- The number of resources used by the plan
- The feasibility of executing the plan
- The reliability of the plan

Plan length is usually not an accurate measure of plan quality in the process planning domain, since different operators have different execution costs. In fact, it is sometimes the case that a high-quality plan will have more steps than a low-quality one. A more accurate quality measure is the total cost of the plan, which can be measured by execution time or by the dollar cost of each plan step.

One rough measure of the plan execution time can be made by counting the number of set-up steps that exist in the plan (Pérez, 1995). Set-up steps usually prepare a machine or part for an operation. For instance, placing a part in a machine or rotating a part already held in a machine are operations that fit in to this category. Set-up steps are very important in the process planning domain and usually take the longest to execute. Table 7.1 describes a quality metric from Pérez (1995) that reflects this type of quality measure; for the purposes of this dissertation, call this **Quality Metric 1**. This metric assigns each operator a fixed cost where lower values are higher quality. The cost of a plan is determined by adding up the costs of all plan steps. This particular metric assigns a higher cost to set-up steps and reflects the fact that it is often cheaper to switch tools in a machine than it is to manipulate parts.

Other quality measures can also be defined for this domain. For this dissertation, a second metric was created that reflected another possible type of plan cost. Some machines in production manufacturing may be significantly more expensive to operate than others. Thus, a beneficial quality measure could directly reflect machine operation cost. Table 7.2 describes a second quality measure; call this **Quality Metric 2**. This metric states that operators that use a milling machine are more costly than those that use a drill. In the process planning domain, a *milling machine* can be used to both change the dimension of a part and to drill a hole in a part; a *drill* machine can only be used to drill holes. Thus, according to this quality metric, if a plan's only goal is to drill a hole, a plan that used a drill machine to create the hole would be better than a plan that used a milling machine, even though both plans would be correct.

The two quality metrics defined in Tables 7.1 and 7.2 can prefer very different high-quality plans for the same problem. For instance, consider the problem shown in Figure 7.3. This problem has two goals, one to change the size of the part and one to drill a hole in one side. Two solutions plans for this problem are also shown in Figure 7.3. These solution plans score very differently with the two quality metrics defined previously in this chapter. The first plan scores much better (i.e. lower) with Quality Metric 2 since the milling machine is only used for one operation, while in the second plan the milling-machine is used for both operations. Conversely, the second plan scores better with Quality Metric 1, since it requires fewer setup steps than the first plan.

7.3 Focusing Scope on Improving Efficiency and Plan Cost

In the previous chapter, Scope was shown to improve both planning efficiency and the quality metric of plan length. This section focuses on improving efficiency and quality metrics that relate to different types of plan cost.

Again, in order to improve plan quality as well as efficiency, Scope is trained on only high-quality solutions (and when possible, on optimal solutions). When using plan cost as an evaluation metric, Scope should be trained on very low cost plans for all training problems. Unfortunately, the process planning is a very complex domain and the base planner has difficulty producing solutions in a reasonable amount of time. (In the tests shown in the next section, the base planner could only solve 33% of the test problems under a time limit of 500 seconds). Additionally, providing solutions to the training problems that are high quality is even more difficult.

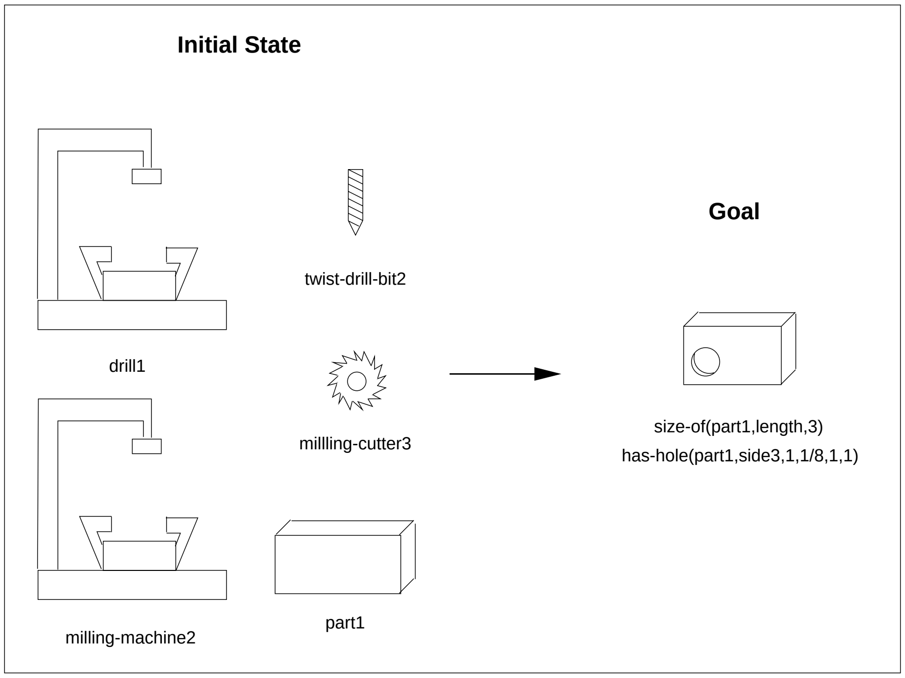
The approach used to generate high-quality solutions for training problems was the following. First, in order to solve all of the training problems, the Prodigy4.0 planner (Carbonell & et al., 1992) was utilized, which contains a large set of hand-coded control rules for the process planning domain. These rules enable the Prodigy planner to solve problems in this domain much quicker than with the base planner used in this dissertation. Second, once solutions for all training problems were generated by Prodigy, two different sets of solutions were created by modifying

Operator Type	**Cost**	**Operators**
Machining operators	1	drill-with-spot-drill, drill-with-twist-drill, drill-with-high-helix-drill, drill-with-spot-drill-in-milling-machine, drill-with-twist-drill-in-milling-machine, face-mill, side-mill
Machine and holding device set-up operators	8	put-holding-device-in-drill, put-holding-device-in-milling-machine, put-on-machine-table, remove-from-machine-table, hold-with-vise, release-from-holding-device
Tool operators	1	put-tool-on-milling-machine, put-in-drill-spindle, remove-tool-from-machine
Cleaning operators	6	clean, remove-burrs

Table 7.1: **Quality Metric 1** - This quality metric reflects execution time in the process planning domain. Higher costs are assigned to steps which involve holding or moving parts.

Operator Type	**Cost**	**Operators**
Milling machine operators	15	drill-with-spot-drill-in-milling-machine, drill-with-twist-drill-in-milling-machine, face-mill, side-mill
Drill operators	1	drill-with-spot-drill, drill-with-twist-drill, drill-with-high-helix-drill
Machine and holding device set-up operators	2	put-holding-device-in-drill, put-holding-device-in-milling-machine, put-on-machine-table, remove-from-machine-table, hold-with-vise, release-from-holding-device
Tool operators	1	put-tool-on-milling-machine, put-in-drill-spindle, remove-tool-from-machine
Cleaning operators	3	clean, remove-burrs

Table 7.2: **Quality Metric 2** - This quality metric reflects a different type of execution cost in the process planning domain. Higher costs are assigned to steps that utilize expensive machines.



(a)

Solution Plan 1	Solution Plan 2
put-on-machine-table mill-mach	put-on-machine-table mill-mach
clean-part	clean-part
hold-with-vise	hold-with-vise
put-tool-on-mill-mach mill-cutter	put-tool-on-mill-mach mill-cutter
face-mill	face-mill
release-from-holding-device	remove-tool milling-cutter
put-on-machine-table drill	put-tool-on-mill-mach twist-drill2
remove-burrs	drill-with-twist-drill
clean-part	
hold-with-vise	
put-in-drill-spindle twist-drill2	
drill-with-twist-drill	

(b)

Plan	Quality Metric 1	Quality Metric 2
Plan 1	62	37
Plan 2	26	40

(c)

Figure 7.3: Example of a problem in the process planning domain. The goal of this problem is to change the part's length to 3 and to have a hole drilled in one side.

the Prodigy solutions, where each set corresponded to a particular quality metric. The set of solutions corresponding to Quality Metric 1 was produced by first examining each solution that was produced by Prodigy, and if that solution could be improved with respect to Quality Metric 1, it was modified by hand so that the new solution was optimal. The same procedure was used to produce a solution set for Quality Metric 2. Most solutions generated by Prodigy had to be modified in some way in order to produce an optimal solution for either quality metric. Two different training sets were then produced, which contained the same problems but different solutions.

7.4 Experimental Design

The goal of these experiments was to improve both planning efficiency and also the quality of generated solutions. Specifically, these experiments were intended to evaluate how well Scope could improve upon different quality metrics. The two quality metrics defined in Section 7.2 were used to measure solution quality for this domain. Scope's goal in these tests is to minimize the solution cost, since lower cost solutions are considered more higher quality.

A random problem generator was built to generate problems for the process planning domain. It operates by creating a random initial state that contains different machines, parts and tools, where parts and tools can be placed in different locations (e.g. the drill machine may be already holding a certain size drill bit). It also creates a random set of goals for any available parts in a problem. These experiments concentrated on goals of cutting parts to a desired size along their three dimensions, and on spotting and drilling holes of different sizes which are located in random locations on a part. Each problem contained from one to three different goals. Using the problem generator, 100 training problems were created and then optimal solutions for Quality Metric 1 and 2 were produced using the technique described in the preceding section.

Two experiments were run, one for each quality metric; call these **Experiment 1** and **Experiment 2**, where Experiment 1 used the training problems optimized for Quality Metric 1 and Experiment 2 used the training problems optimized for Quality Metric 2. For each experiment, Scope was trained on separate example sets of increasing size. Only one trial was run in these experiments due to the time involved in generating training solutions and in learning control rules. A test set of 100 independently generated problems was used to evaluate performance. During testing, a time limit of 500 seconds was imposed per test problem and a depth bound on plan length of 30 steps was used that allowed for all problems to be solved. All tests were performed on a Ultra Enterprise 5000.

Similar to the procedure used in previous experiments, Scope learned control rules from the given training set and produced a modified planner. If the new planning program was ever found to be too specialized (i.e. it failed on one of the training problems) then the initial planning program is used to solve the problem. When this situation occurs in testing, both the failure time for the new planner and the solution time for the original planner are included in the total solution time for that problem.

For comparison purposes, best-first search was also evaluated at solving the test problems. Again, these tests were done using the standard Lisp implementation of UCPOP (Barrett & et al., 1995), thus some results could be partially due to implementation difference between Prolog and Lisp.

7.5 Efficiency Results

Figure 7.4 shows the measured improvements in planning efficiency for both experiments. In both tests Scope was able to greatly improve planner efficiency. In Experiment 1 (where training problems were optimized for Quality Metric 1), Scope was able to speedup the original planner by a factor of 4.8. In Experiment 2 (where training problems were optimized for Quality Metric 2), Scope was able to speedup the original planner by a factor of 5.7. And, for all points where control knowledge was added, no backtracking occurred in the new planner in both experiments.

Scope was able to achieve a slightly higher speedup factor in Experiment 2, where quality relates to machine operation cost. This could be partly due to the fact that fewer control rules were learned for Experiment 2. In Experiment 1, 57 rules were learned and in Experiment 2, 51 rules were learned. In addition, Scope outperformed best-first search in this domain, however best-first search was somewhat faster than the base planner using depth-first search.

In both experiments, Scope was also able to significantly improve the number of test examples that could be solved under the time limit, as shown in Figure 7.5. The original base planner could solve 33% of the test problems under the time limit and best-first search was able to solve 45% of the test problems under the time limit. In both Experiment 1 and Experiment 2, the new planner produced by Scope could solve 86% of the test problems, and thus Scope was able to greatly improve the number of test problems that could be solved in a reasonable amount of time.

Since significant speedup and coverage improvements were realized for both experiments, these tests show that the efficiency improvements generated by Scope are not dependent to a certain type of quality metric.

7.6 Quality Results

Plan quality improvements were collected in two different ways. The goal for both these evaluations was to test whether training Scope on optimal solutions for a particular quality metric helped it to produce a planner that generated solutions that score well with that metric. Thus, ideally, control rules from Experiment 1 should minimize Quality Metric 1 and control rules from Experiment 2 should minimize Quality Metric 2.

First, plan quality was evaluated for all test problems that could be solved by both final planners. In the test set, 79 of the problems could be solved by both new planners. The set of solutions for these problems produced in Experiment 1 were evaluated once using Quality Metric 1 and then again using Quality Metric 2. The same was done for the set of solutions produced in Experiment 2. The results from this evaluation are shown in Table 7.3. The total costs for the problem solutions generated in both experiments are given for both quality metrics.

As shown by the table, the training set optimized for Quality Metric 1 produced control rules which in turn generated solutions that minimized this metric but did not score well with the second quality metric. Similarly, the training set optimized for Quality Metric 2 produced control rules which generated solutions that minimized this metric but did not score well with the first quality metric. The solutions generated by Experiment 1 were 18% lower in cost for Quality Metric 1, than solutions generated be Experiment 2. The solutions generated by Experiment 2 were 35% lower in cost for Quality Metric 2, than solutions generated by Experiment 1. Thus, this evaluation

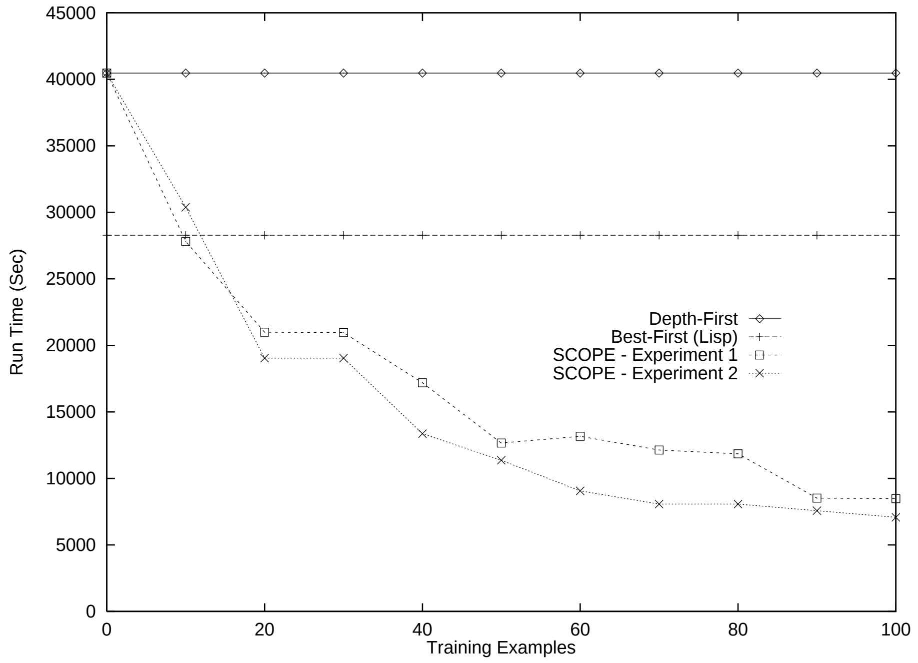


Figure 7.4: Efficiency performance for process planning domain. Two experiments were run optimizing two different quality metrics.

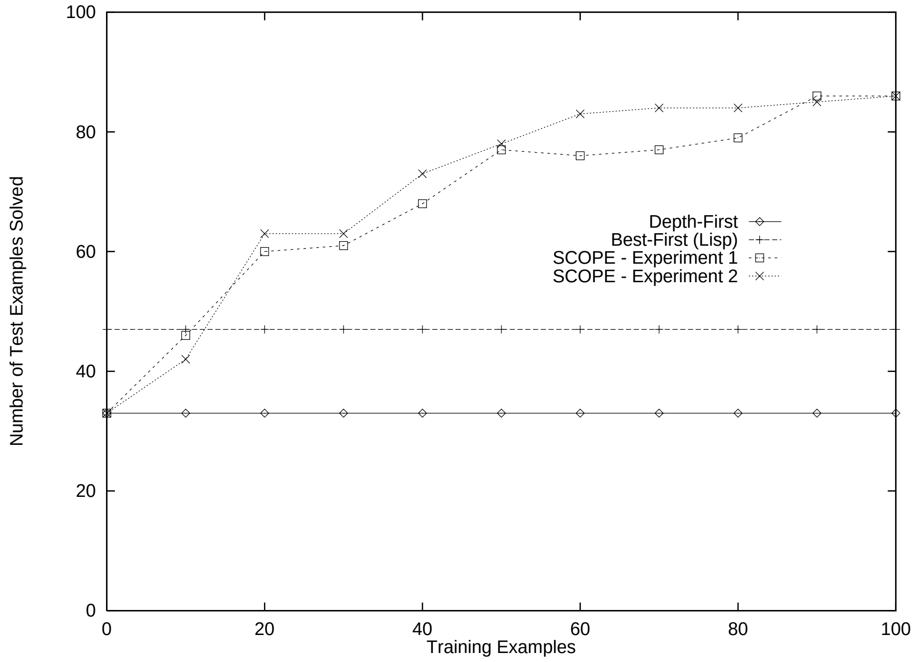


Figure 7.5: Number of test problems solved under a time limit.

Solution Set	**Quality Metric 1**	**Quality Metric 2**
Experiment 1	3881	3606
Experiment 2	4750	2346

Table 7.3: The total cost of solutions produced in two different experiments, as evaluated by two different quality metrics. Only solutions for problems which both new planners could solve are included. In experiment 1, the training problems were optimized for Quality Metric 1. And in experiment 2, the training problems were optimized for Quality Metric 2.

Solution Set	**Quality Metric 1**	**Quality Metric 2**
SCOPE - Exp 1	886	789
SCOPE - Exp 2	1102	527
UCPOP - DF	994	491
UCPOP - BF	1006	788

Table 7.4: The total cost of solutions produced in two different experiments, as evaluated by two different quality metrics. Only solutions for problems that could be solved by both new planners produced by SCOPE, by the base planner (using depth-first) and by standard UCPOP (using best-first) were included.

shows that SCOPE can successfully improve on very different quality metrics.

A second comparison was done that only used solutions for problems that could be solved by both final planners produced by SCOPE, by the base planner (using depth-first), and by the standard Lisp version of UCPOP (using best-first search). In the test set, 31 of the problems could be solved by all four planners. Similar to the last evaluation, the set of solutions for each planner was then evaluated once using Quality Metric 1 and again using Quality Metric 2. The results of this evaluation are shown in Table 7.4.

Again, the training set optimized for Quality Metric 1 produced control rules which in turn generated solutions that minimized this metric but did not score as well with the second quality metric. Similarly, the training set optimized for Quality Metric 2 produced control rules which generated solutions that minimized this metric but did not score well with the first quality metric.

For the first quality metric, the lowest cost solutions were produced by the planner generated in Experiment 1. These solutions were 20% lower in cost than solutions generated by the new planner in Experiment 2, 11% lower in cost than solutions produced by the depth-first version of UCPOP, and 12% lower in cost than solutions generated by best-first UCPOP.

For the second quality metric, the lowest cost solutions were actually produced by the depth-first version of UCPOP. These solutions were 38% lower in cost than solutions produced by the new planner from Experiment 1, 7% lower in cost than solutions produced by the new planner from Experiment 2, and 38% lower in cost than solutions produced by best-first UCPOP. Thus, in Experiment 2 (which was optimized for Quality Metric 2), SCOPE did not produce solutions that were lower in cost than those produced by the base planner. For this quality metric, the base planner output optimal solutions for all test problems solved so there is no room for quality improvements on these test problems. For most of the problems, the new planner from Experiment 2 produced equivalent solutions to those produced by the base planner, however for just a few problems it produced slightly worse solutions, which account for the difference in total plan cost shown in Figure 7.4. The solutions produced in Experiment 2 were significantly lower in cost than solutions produced in either Experiment 1 or by the best-first search version of UCPOP.

Thus, when optimized for a particular quality metric, the new planner produced by Scope was always able to outperform the standard Lisp implementation of UCPOP (using best-first) and was able to outperform the depth-version of UCPOP for one quality metric and only slightly underperformed the depth-version of UCPOP for another quality metric. Plus, the new planner produced by Scope was always able to produce significantly more solutions under the time limit than could be produced by either implementation of UCPOP.

7.7 Learning Time

Again, the efficiency results presented in this chapter only considered initial and final planning time and did not include learning time. This section discusses learning time in the process planning domain.

Learning times from the both experiments are shown in Figure 7.6. In Experiment 1, the final learning time after training on 100 examples was approximately 115 hours. In Experiment 2, the final learning time after 100 training examples was approximately 147 hours. Thus learning times are very high for this domain, as compared to the logistics and blocksworld domains. This is mainly due to the fact that there are a much higher number of decision points in the process planning domain than in the other two domains tested. The process planning domain contains 81 operators, compared to 4 in the blocksworld and 6 in the logistics domains, and there are many more points where control rules could be learned in this domain. Also, Scope must always examine all decision points to see if control knowledge could be useful. So even if a decision point cannot benefit from learned control knowledge (i.e. it's possible that that decision point is rarely used or that backtracking just never occurs at that point), Scope still spends time examining it.

Even with the high learning time, Scope can still be seen as very beneficial in this domain. First of all, it is unclear how much speedup Scope really achieves in this domain since many of the test problems cannot be solved by the base planner under the time limit. Though employing a time limit is often necessary when running experiments such as the one presented in this dissertation, it can often underplay the true speedup achieved by a learning system (Etzioni & Etzioni, 1994). For these experiments, it is unclear how long it would take the base planner (without control knowledge) to solve all of the test problems. The planner was run for several days without a time limit and could only solve a few problems.

Second, Scope is producing high quality solutions. The time required to automatically find high quality solutions is often much more than the time required to find just a valid solution, since the planner might have to perform an extensive search through a number of different solutions. Thus, even though learning time is large in this domain, it can easily be argued that the benefits provided by Scope outweigh any drawbacks introduced by a large learning time.

7.8 Summary

This section presented experimental results for using Scope to improve both planning efficiency and the quality metric of execution cost. These experiments were performed using the process planning

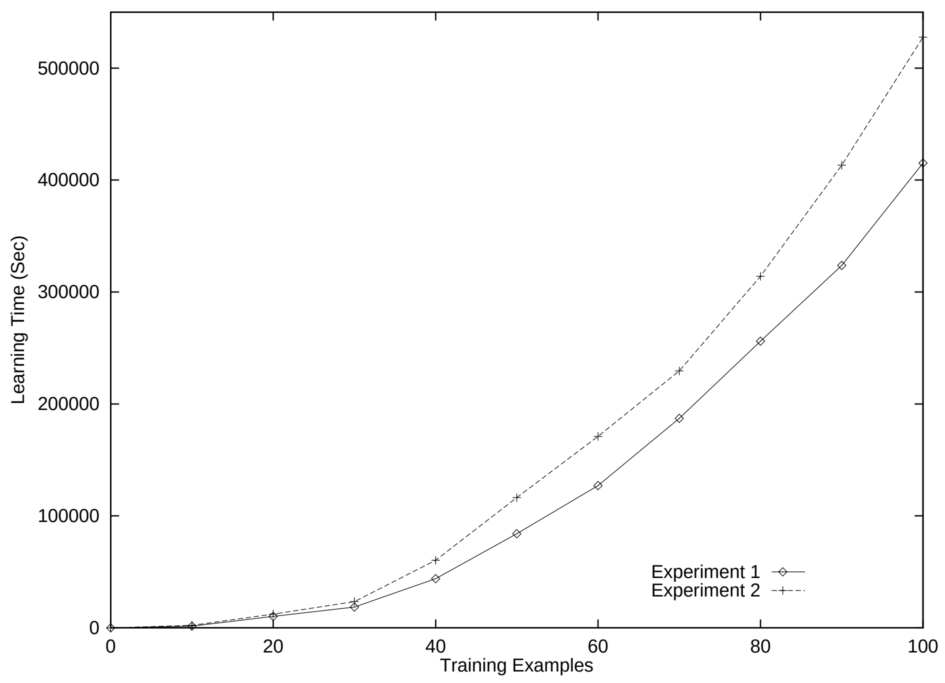


Figure 7.6: Learning time in the process planning domain.

domain, which is a very large and complex domain that describes a number of manufacturing processes.

Two different quality metrics were defined for the process planning domain where each metric emphasized a different type of execution cost. Two different experiments were run, where the goal of each experiment was for SCOPE to produce control rules that emphasized one of the quality metrics and also improve planning efficiency. In both experiments, SCOPE was able to significantly improve planner efficiency and increase the number of test problems that could be solved under a time limit. Also, SCOPE was able to produce solutions that emphasized the chosen quality metric. These results show that SCOPE can successfully be applied to improve both planning efficiency and plan quality, and that these results are not dependent on the particular quality metric being used. They also show that SCOPE can be used effectively to improve planner performance on a realistic, large-scale domain.

Chapter 8

Experimental Evaluation - Improving only Efficiency

One question brought out by the last two chapters, is "what happens if we don't train SCOPE on *high-quality* solutions?" What efficiency gains will SCOPE achieve after looking at solutions that have not been optimized for quality? In order to investigate these questions, SCOPE was run on a similar set of experiments as those presented in Chapter 6. The main change was that SCOPE was trained on the first solution found using depth-first search, instead of being trained on very high-quality solutions. The goal of these experiments was to test whether SCOPE could produce efficiency improvements even when trained on sub-optimal solutions, and also to see what effect these improvements would have on plan quality.

8.1 Experimental Design

These experiments used the logistics transportation and the blocksworld domains to evaluate SCOPE's performance. The same example sets from Chapter 6 were used for training and testing. However, instead of solving the training examples using depth-first iterative deepening, depth-first search was utilized. During training, the first solution found was always used by the learning algorithm. In both domains, depth-first search could solve all problems in a reasonable amount of time, so a bootstrapping training method was not necessary for these tests.

The same two sets of 100 test problems used in Chapter 6 were used to evaluate performance. Five trials were run for each training set size, after which results were averaged. No time limit was imposed on planning but a uniform depth bounds on plan length was used during training and testing that allowed for all problems to bel solved. Depth bounds were set identical to those in Section 6.3; in the blocksworld, the depth bound was set at 8 and in logistics, the depth bound was set at 100. Again, tests in the blocksworld were performed on an Ultra Sparc 2 and in the logistics domain on an Ultra Enterprise 5000.

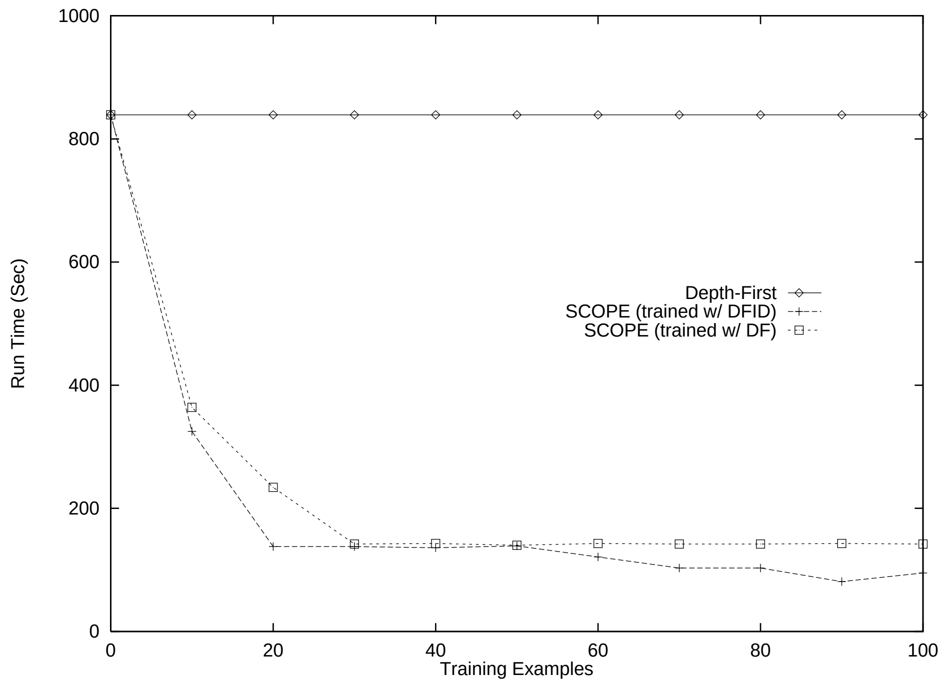


Figure 8.1: Efficiency performance in the logistics transportation domain.

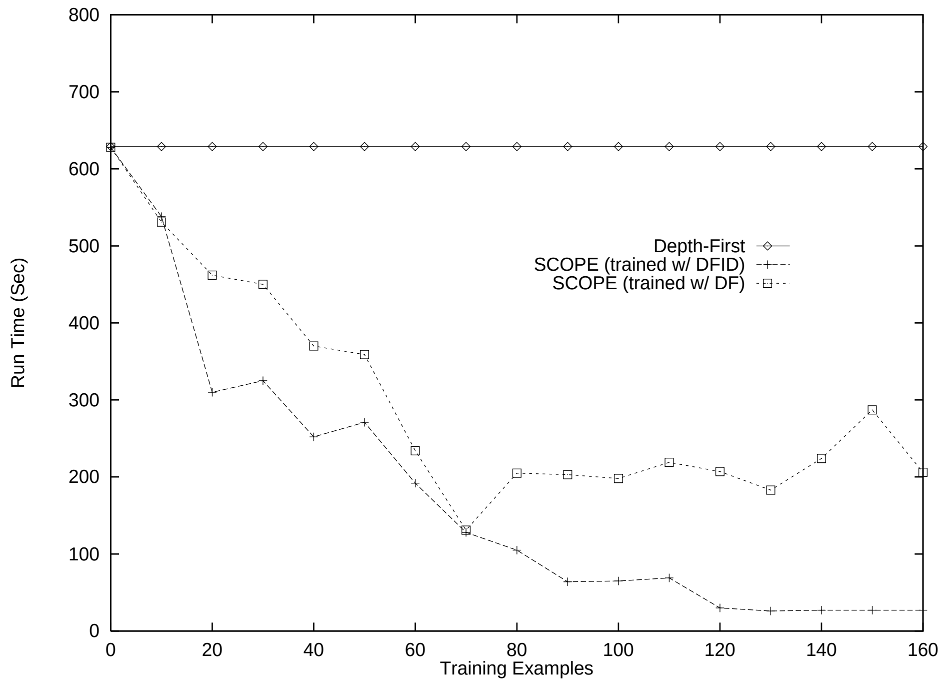


Figure 8.2: Efficiency performance in the blocksworld domain.

8.2 Efficiency Results

Figure 8.2 and Figure 8.1 show the measured improvements in planning efficiency for both domains. The results from the previous set of experiments are also shown for comparison. Again, SCOPE produced a more efficient planner than the original after incorporating learned control information. In these tests, SCOPE was able to produce a new planner that was 6 times faster in the logistics domain, and 3 times faster in the blocksworld domain. In the logistics domain, SCOPE was still able to achieve very significant speedup when trained on non-optimal solutions, although it was not quite as high as the previous experiments, which achieved a speedup factor of 8x. In the blocksworld domain, SCOPE was also able to produce a more efficient planner, however, the speedup was much less than in earlier tests where SCOPE was trained on optimal solutions and a speedup factor of 23x was achieved.

There are several possible reasons why speedup factors are less when SCOPE is trained on solutions that have not been optimized for quality. For one, swaying SCOPE towards shorter solutions means less work for the final planner. Since solutions are shorter, fewer planning iterations must take place. Second, SCOPE was not always able to learn accurate rules in these experiments. In the logistics domain, accurate rules were learned for all decision points. However, in blocksworld, SCOPE is able to learn some good rules, but there are a few decision points where it is unable to learn a fully accurate set of rules. As explained in Section 5.3, when a rule is learned that is not fully accurate (i.e. it covers negative examples), the rule is still used, but no cut is added after the rule. This allows the corresponding refinement application to be still backtracked upon. Thus, the new planner can still solve the test problems, however, the solution times are larger since more backtracking occurs.

The refinement-selection examples that SCOPE cannot learn to accurately cover in these situations usually correspond to similar planning subgoals that were solved in different ways during training. These differences stem from how close the plan is to the depth limit. For instance, consider the two problems shown in Figure 8.3. To solve the first goal for each problem, clear(b), suppose the first action the planner will try to add is unstack(X,b), where X is another block in the problem. This could return a valid plan for each problem, as shown in Figure 8.4. However, if the depth limit given the planner is 7, then a different Plan B will be created that has only 5 steps. In this plan (which is the same as Plan B minus the last three steps) clear(d) is asserted by the step putdown(d), instead of by the step unstack(b,d). Thus, clear(d) will be achieved in different ways for these two problems, even though they are very similar. SCOPE has a hard time learning a rule to cover scenarios such as these, where planner behavior is different due to the depth limit.

This type of problem did not occur in the experiment for the logistics domain. One reason for this could be that logistics problems were solved using a relatively high depth limit (i.e. most problem solutions were well under the depth bound). Where as, in the blocksworld, solutions were often found very close to the depth bound. Unfortunately, it was difficult to raise the depth limit in the blocksworld and still solve the training problems in a reasonable amount of time.

One way to avoid this problem, might be to use a very high depth limit which would not affect most solutions. Unfortunately, as just mentioned, this makes it hard in many domains to solve the training problems in a reasonable amount of time. Another solution, would be to give the learning system a method for reasoning about the depth limit where it can build rules that can evaluate whether the current search path will hit the depth limit before finding a solution. For

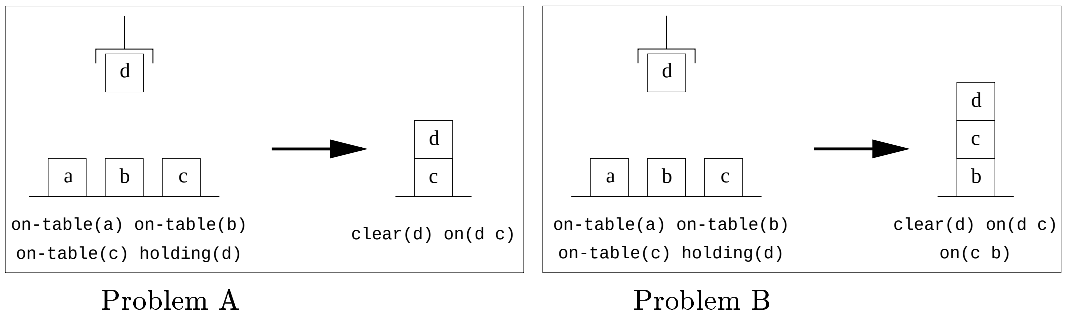


Figure 8.3: Two similar problems from the blocksworld.

Plan A	Plan B
stack(d,c)	putdown(d)
pickup(b)	pickup(c)
stack(b,d)	stack(c,b)
unstack(b,d) *	pickup(d)
	stack(d,c)
	pickup(b)
	stack(b,d)
	unstack(b,d) *

Figure 8.4: Solution plans for the two problems shown above.

instance, the UCPOP+EBL system (Kambhampati et al., 1996), which is discussed in Chapter 9, utilizes extra domain axioms to help it reason about depth-limit failures.

8.3 Quality Results

Changes in plan quality for these experiments are shown in Figure 8.5 and Figure 8.6. Results from these experiments and from the previous set of experiments where SCOPE was trained on high-quality solutions are shown for comparison. When trained on non-optimal solutions, SCOPE produces a new planner that does not significantly change plan quality. In the logistics domain, the new planner has no effect on plan quality and the new planner behaves identical to the original planner. In the blocksworld domain, the new planner does produce a few slightly shorter solutions, however in general solutions remain the same length as with the original planner. Thus, for this training methodology, solution lengths after learning remained very similar to those found before learning. These results demonstrate again how the methodology used in training SCOPE can greatly affect final performance.

8.4 Summary

This section presented experimental results for using SCOPE to improve planning efficiency when trained on solutions not optimized for quality. Results are presented in the two domains used in Chapter 6, the blocksworld and logistics domain. In these experiments, SCOPE is shown able to still improve planning efficiency in both domains. In the logistics domains, efficiency gains were pro-

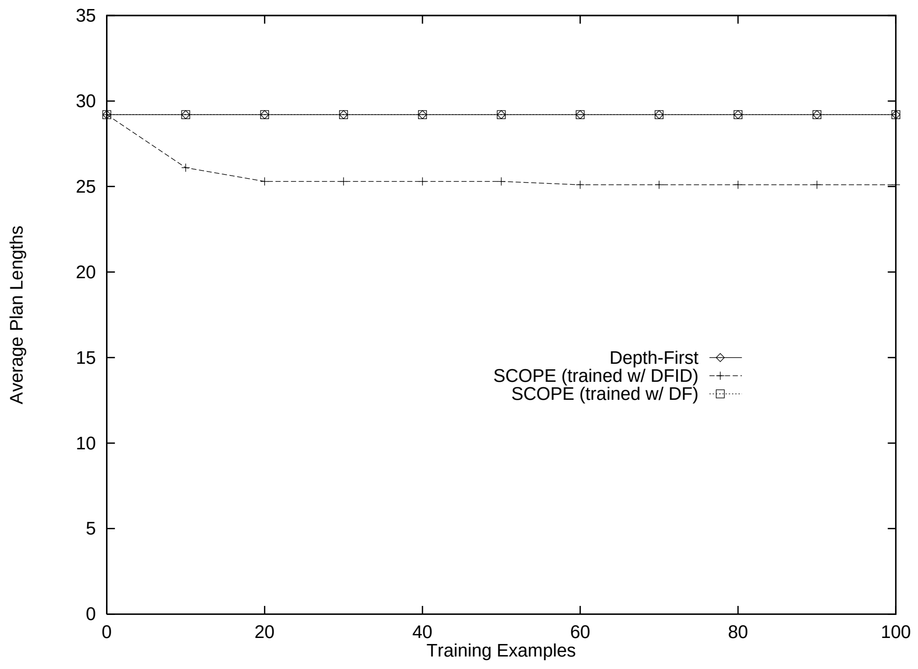


Figure 8.5: Quality performance in the logistics transportation domain. Here, depth-first (without control information) and SCOPE trained with depth-first have the same results.

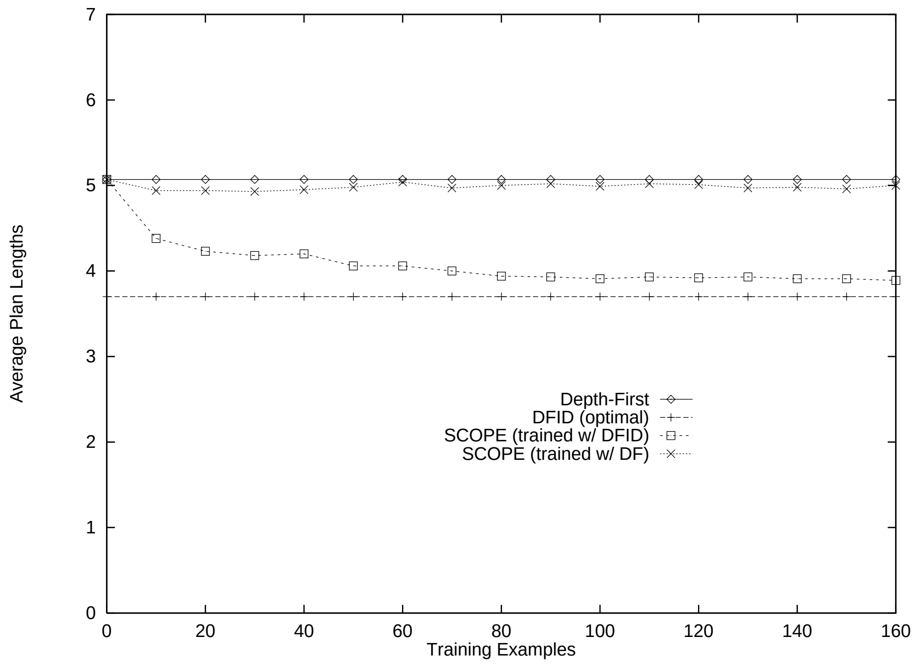


Figure 8.6: Quality performance in the blocksworld domain.

duced that were comparable to those produced in Chapter 6. In the blocksworld domain, efficiency gains were also produced, however, they were much less than those seen in the Chapter 6 experiments. SCOPE had difficulty always building accurate rules in this experiment, due to differences in blocksworld solutions that were caused by the depth limit.

Plan quality changes were also presented for these experiments. In both domains, SCOPE produced a new planner that does not significantly change the quality of solutions.

Chapter 9

Related Work

The work discussed in this dissertation touches on numerous areas of artificial intelligence, including machine learning, planning and inductive logic programming. Some related work in these areas has already been mentioned in other chapters. This chapter presents a broader discussion of work related to SCOPE. Although it would be impossible to mention in detail all related research, a comprehensive summary is attempted of the closest work.

9.1 Learning Control for Problem Solving

Early research in learning control rules has been focused on a variety of problem solving applications, such as symbolic integration, eight-puzzle, and the N-Queens problem. Much early work centered around learning *macro-operators* which record successful operator sequences so they can be reused on future problems. (Fikes & Nilsson, 1971; Korf, 1985; Minton, 1985; Porter & Kibler, 1986). Other systems concentrated on learning heuristics that characterized successful problem solving behavior. To acquire control information, most systems employed a similar method to SCOPE's of analyzing the search space. Positive and negative examples of problem solver behavior are identified (Mitchell et al., 1983; Langley, 1985), and then control heuristics are learned to cover the positive examples and rule out the negatives. Several early approaches also used a combination of induction and EBL to learn control information. The LEX-2 (Mitchell et al., 1983) and MetaLEX (Keller, 1987) systems constructed rules by inducing over complete explanation-based generalizations of problem-solving traces. SCOPE, on the other hand, uses induction to select the most useful pieces of EBG generalizations.

Some work has also been done on learning approximations to EBL rules. The ULS system (Chase et al., 1989) acquired conservative approximations to EBL rules by simply dropping one or two conditions. ULS was very limited in the rules it could generate, unlike SCOPE, which uses an inductive learning mechanism to build rules from scratch. A more closely related system is AxA-EBL (Cohen, 1990), which integrates an induction mechanism to learn approximate EBL rules. AxA-EBL first learns a control rule by applying EBG to the proof of a correct control decision. A pool of candidate control rules is then formed by considering all *k-bound approximations* of this rules, where a k-bounded approximation is formed by dropping k or less rule conditions. AxA-EBL then searches this pool for a small set of rules that maximizes coverage of the positive examples and minimizes coverage of the negative examples. Although quite successful, this system does has several weaknesses. First, the number of k-bounded approximations grows exponentially in k, thus k is limited to very small values. A second problem is that explanations for subgoals only consider the context of that particular subgoal. Often the conditions which cause a particular operator to fail, lie outside the proof of the specific subgoal to which that operator was applied.

The DOLPHIN system (Zelle & Mooney, 1993) improves on AxA-EBL by using a more powerful induction algorithm and analyzing proof trees of *entire* problems instead of only explaining individual subgoal successes. The overall goal of DOLPHIN is Prolog program optimization. Specifically, DOLPHIN learns learns control rules that determine which clauses in a Prolog program to use to solve particular subgoals. These rules are learned through a combination of explanation-based generalization (EBG) and inductive-logic programming. DOLPHIN has been shown successful at improving problem-solver performance in several different domains, such as N-queens and two planning domains which utilized a simple linear, state-based planner. Much of the original inspiration for SCOPE came from ideas introduced in DOLPHIN, however there are a number of differences between the two systems.

SCOPE and DOLPHIN both use a combination of EBG and ILP techniques to learn control rules and both have been used to improve the performance of Prolog programs. However, DOLPHIN has only been applied to simple logic programs whose search spaces were fairly easy to analyze. SCOPE, on the other hand, has been applied to a very complex problem solver, which required a much more efficient EBG algorithm and also a more powerful induction engine to learn control information. DOLPHIN has difficulty efficiently generating explanation structures for a partial-order planner, and for many problems cannot generated this explanation in a reasonable amount of time. This difficulty probably stems from the large search space that must be examined. On the induction side, DOLPHIN's algorithm approach is very limited and cannot construct rules that are expressive enough to analyze complex situations, such as those found in a partial-order planner. DOLPHIN's algorithm can only add rule antecedents straight from generalized proof trees and has no mechanism for modifying an antecedent once it is added to a rule. DOLPHIN also has none of the additional inductive features that have been added to SCOPE (e.g. negated antecedents, relational clichés) and thus can only build very simple control rules. Finally, SCOPE can learn rules that improve upon final plan quality, unlike DOLPHIN which is only focused on improving program performance and can not change the quality of produced solutions.

9.2 Learning Control for Planner Performance

9.2.1 Systems Applied to State-Based planners

A significant amount of research in learning control knowledge has been explicitly directed towards improving the performance of planning systems. Most of this research has concentrated on linear, state-based planners. For instance, the PRODIGY planning and learning system (Minton, 1989) employs a version of EBL, called *explanation-based specialization*, to learn domain-specific control rules for a linear, state-based planner. Rules are learned in response to both planning failures and successes and also from unforeseen goal interactions. Learned control rules can select, reject, or prefer plan-refinement candidates for several different decision types. A number of other systems have also applied EBL to learn search-control for planning. STATIC (Etzioni, 1993) acquires control rules by analyzing the problem domain theory. This system uses EBL to analyze a graph structure that captures the precondition/effect dependencies between the actions in the domain. This analysis is then used to derive goal-ordering rules for the PRODIGY state-based planner. FAILSAFE (Bhatnagar & Mostow, 1994) was designed to learn control rules in domains where the underlying domain theory was recursive. This system uses a forward-searching state-based planner and learns by building incomplete explanations of its execution time failures.

Not all learning approaches have relied on EBL. GRASSHOPPER (Leckie & Zuckerman, 1993) uses an inductive approach to learn planning control knowledge. This system learns control rules for goal, operator, and variable binding decisions. Given a set of training examples, GRASSHOPPER looks for sets of similar decisions that could form the basis for control rules. Rule preconditions are generated by finding a generalized set of current state conditions that hold true at the beginning of each decision. This inductive search for rule conditions is a relatively small since it only searches through world-state conditions. Unfortunately, as mentioned in Chapter 5, applying an inductive search in a partial-order setting would require a much more complicated set of predicates that would most likely render the search infeasible to perform.

All of these learning systems have been shown successful at improving planner performance in several domains, such as the blocksworld and stripsworld. However, unlike SCOPE, each system's architecture is limited to apply only to a linear, state-based planner. Most of the rules generated by these early systems heavily rely on the assumptions of a state-based search and goal linearity, neither which are valid in a UCPOP-style planner. For instance, rules often check for conditions present in the current state and they also often assume that all current actions in the plan are ordered. These systems would also have difficulty analyzing the search space of a partial-order planner. Since partial-order planners operate in a plan-space instead of a state-space, the process of creating and generalizing explanations is more complex. Removing the linearity assumption adds further complications for learning search-control not addressed by these early systems, since many more search paths must be considered in generating an explanation. Unfortunately, very few learning systems have been built to acquire control knowledge for other styles of planning.

HAMLET (Borrajo & Veloso, 1997) is one more recent system, which learns control knowledge for the nonlinear planner PRODIGY4.0 (Carbonell & et al., 1992). Similar to SCOPE, HAMLET uses a combination of EBL with induction to acquire control rules. First, optimal solutions are gathered for each training example by performing an exhaustive search of the problem search space. HAMLET first generates a bounded explanation of each planning decision and then incrementally refines any incomplete or inaccurate explanations. Control rules can either be specialized if they are found to cover negative examples or generalized if they are found to exclude positive examples. Unlike SCOPE, HAMLET uses EBL to build rules, and induction is primarily used to refine learned knowledge. Also, though the Prodigy4.0 planner does remove the linearity restriction for examining goals, it is still a state-based planner and many rule conditions used by HAMLET are directly dependent on current state information. It is thus unclear, how HAMLET would perform on a partial-order planner. Also, HAMLET has only been tested on small domains for the quality metric of plan length. It is thus also unclear, how HAMLET would perform on more complex domains and whether it could improve upon other types of quality metrics.

HAMLET has successfully improved the efficiency performance of the PRODIGY4.0 planner in the blocksworld and logistics planning domains, however, since HAMLET and SCOPE have been tested on completely different planning algorithms, only a very rough comparison can be drawn.

When examining the results reported for HAMLET in Borrajo and Veloso (1997) and the results reported for SCOPE reported in Chapter 6 of this dissertation, the following was noted.

In blocksworld, SCOPE achieved a speedup factor of 23x on problems containing 4-6 goals, while HAMLET achieved a speedup of 2x on problems containing 5 goals and a speedup of 35x on more complicated problems containing 10 goals.[1] In the experiments for HAMLET, the number of blocks in each problem was greatly varied (between 5 to 50), while for SCOPE it only slightly varied (between 3 and 8). SCOPE was able to produce significant quality improvements in this domain, lowering solution lengths an average of 22.9%, however, HAMLET produced very little improvement and only lowered solutions lengths by 2.2%. However, this result could be due to the fact that the PRODIGY planner produces relatively high-quality solutions even before learning.

Both systems were run on increasing complex problems in the logistics transportation domain. In these tests, SCOPE was trained on problems containing 3 packages (and 3 goals) distributed over 2 cities and HAMLET was trained on problems containing 1-2 goals for 5 packages distributed over 1-3 cities. In these tests, SCOPE achieved speedup factors ranging from 2.2 on simple 1 goal problems and up to 6.0 on 5 goal problems and 4.5 on 20 goal problems. Conversely, HAMLET produces relatively small speedup gains, ranging from 1.3 on simple 1 goal problems to 1.5 on 5 goals problems and 1.2 on 20 goals problems. Neither system achieves significant efficiency gains on 50 goals problems, however SCOPE was able to improve test problems coverage from 0% to 24% while, HAMLET only improved coverage from 2% to 4%. SCOPE was also able to more significantly improve test problem coverage at other levels of complexity. For quality results, both systems were able to improve their base planners. The greatest improvement realized by SCOPE was decreasing solution lengths for 10 goals problems by 13.3%. The greatest improvement realized by HAMLET was decreasing solution lengths by 13.2% which was also for 10 goals problems.

In conclusion, both systems were able to significantly improve planning efficiency in the blocksworld domain, however HAMLET was more extensively tested than SCOPE. SCOPE achieved much higher speedup factors in the logistics domain and was able to further improve test problems coverage in this domain. In quality improvements, SCOPE was able to more significantly decrease solution lengths in blocksworld and both systems achieved equivalent improvements in the logistics domain.

9.2.2 Systems Applied to Partial-Order Planners

The only system besides SCOPE to learn control information for partial-order planning is UCPOP+EBL (Kambhampati et al., 1996). This system also learns search control rules to improve the performance of UCPOP, but uses a purely explanation-based approach. Specifically, UCPOP+EBL employs the standard EBL techniques of regression, explanation propagation and rule generation to acquire search-control rules, which are learned in response to past planning failures. UCPOP+EBL learns from both analytical failures (dead-end search paths) and depth limit failures (the search path crosses a depth limit). In order to explain the failures of search paths that cross over a depth limit, this system can utilize extra domain axioms, defined as "readily available physical laws of the domain", that help detect and explain inconsistencies at some depth limit failures. UCPOP+EBL

[1]Note that performance improvements in HAMLET were reported using the number of planning nodes examined as opposed to using cpu time. Though this is a valid method of recording planning performance, it fails to report the time spent in matching rules, and thus could possibly overstate the achieved speedup.

is limited in the rules it can learn, however, since only explainable failures can be utilized for learning. Even with additional domain axioms, it may be difficult or impossible to explain some failures caused by the depth limit. SCOPE, on the other hand, can learn rules to avoid all unpromising paths, as long as a solution path has been identified. SCOPE also utilizes a combination of EBL and induction to learn very general control rules, whereas UCPOP+EBL uses only EBL to build rules.

UCPOP+EBL has been shown to improve planning performance in two domains, the blocksworld domain and the briefcase domain (Penberthy & Weld, 1992). To compare SCOPE to UCPOP+EBL, an experiment was replicated from Kambhampati et al. (1996). This experiment uses three different versions of the blocksworld domain in order to test whether the expressiveness of the domain representation influences the effectiveness of the learner. The first version (called BW-prop) contains only simple propositional preconditions and effects, and contains no conditional effects or universal quantification. The second (called BW-cond) contains conditional effects, and the third (called BW-univ) contains universal quantification. These different domain versions are presented in Appendix A.

The experiment consisted of three phases, each corresponding to one of the domain versions. A set of 100 training problems and a set of 100 test problems was randomly generated and then used for each of the three phases, where each problem contained between three and six blocks and three and four goals. During testing, a time limit of 120 seconds per test problem was used. In addition, experiments in each phase were run with two different goal-selection strategies, one that uses a last-in first-out (LIFO) strategy and one that uses a most-instantiated goals first strategy. (All other experiments up to this point have employed a LIFO goal-selection strategy.)

Both learning systems run UCPOP in a depth-first search mode, however, it should be noted that SCOPE was run used a Prolog version of UCPOP as a base planner, while UCPOP+EBL uses the standard LISP implementation (Barrett & et al., 1995). Also, these experiments were run on different machines. Thus some differences in results could be due to these factors. For instance, the Prolog version of UCPOP was able to solve more of the test problems under the time limit than the Lisp UCPOP, thus speedup factors incurred by the two systems can only be roughly compared.

Table 9.1 shows the results of these experiments. UCPOP+EBL's performance is shown on the left and SCOPE's performance is shown on the right. UCPOP+EBL's performance varies widely across the different domain versions and the different goal-selection strategies. In several of the experiments, very little speedup is realized. However, in one particular experiment, which uses BW-univ and a most-instantiated goals first selection strategy, UCPOP+EBL achieves a very high speedup factor (34x). SCOPE on the other hand is able to achieve significant speedup in all experiments. Surprisingly, the one experiment where the speedup factor achieved by SCOPE is somewhat low (2.7x) is the same experiment in which UCPOP+EBL achieves its greatest performance. This lower speedup is due in part to the fact that the base planner in this experiment was already very efficient (solving 96% of the test problems under the time limit). Plus, SCOPE is able to increase speedup further when trained on more problems. When trained on 200 training problems in the first experiment on BW-univ, the speedup factor reaches 11.3x and test problem coverage increased to 100%. Overall, SCOPE outperforms UCPOP+EBL in these experiments, and seems to be more robust at applying to different domain representations. These results indicate that combining induction with EBL can be more effective than EBL alone at improving the performance of planning

Domain	UCPOP+EBL				SCOPE			
	Scratch		w/ Control		Scratch		w/ Control	
	%Sol	Time	%Sol	Time	%Sol	Time	%Sol	Time
Achieving most instantiated goals first								
BW-prop	51%	7872	68%	5410(1.5x)	92%	1277	97%	402(3.2x)
BW-cond	89%	2821	91%	2567(1.1x)	91%	1268	98%	300(4.2x)
BW-univ	53%	7205	100%	210(34.3x)	96%	982	98%	351(2.7x)
Achieving goals in a LIFO order								
BW-prop	10%	13509	10%	13509(1x)	29%	8936	98%	241(37.1x)
BW-cond	42%	9439	75%	4544(2.1x)	94%	954	99%	124(7.7x)
BW-univ	81%	3126	94%	1699(1.8x)	90%	1359	100%	5(271.8x)

Table 9.1: Comparison between Scope and UCPOP+EBL for improving the efficiency of UCPOP.

systems.

9.2.3 Systems Applied to Decomposition Planners

One other type of learning system that has been developed to increase planning efficiency is X-Learn (Reddy & Tadepalli, 1997), which uses a very different approach to control-knowledge acquisition. Instead of learning separate control rules that are incorporated into an existing planning system, X-Learn builds a new decomposition-based problem solver based on a set of training examples. Similar to Scope, X-Learn also utilizes inductive logic programming techniques, and is constructed using a combination of explanation-based and empirical learning. For each training problem, X-Learn constructs a decomposition rule that covers just that example. Then a "generate-and-test" algorithm is used to combine the rule with any previously learned rules by finding the least general generalization (LGG) of the new rule with the previously formed rules. X-Learn has been shown to produce significant speedup in several planning domains, including a variant of the STRIPS world domain (Fikes, Hart, & Nilsson, 1972) and an air-traffic control domain. Unfortunately, even though X-Learn can produce significant speedup on these domains, it is not guaranteed to produce a complete planner and thus sometimes the final planner cannot produce a solution for all test problems. Also, X-Learn's approach is based around learning a new decomposition based planner and thus, cannot be used to directly improve existing planning techniques, such as partial-order planning; and, unlike Scope, X-Learn only improves problem solver efficiency, and does not attempt to improve solution quality.

9.3 Learning Control for Plan Quality

Most research in learning control rules for planning has been directed at improving planning efficiency. However, some research has concentrated on improving the *quality* of plans produced by a planner. The most prominent of these systems is the Quality learning system (Pérez, 1996) which is built on top of the Prodigy4.0 nonlinear planner. Quality inputs a set of planning domain operators, a domain-dependent metric that evaluates the quality of plans and a set of problems for that domain. It then analyzes the differences between a plan initially produced by the planner and a *better* plan (according to the given quality metric), which can be either input by hand or produced by having the planner search until a better solution is found. Two different learning approaches can be used to construct control information that will lead to higher quality plans. *Control rules* apply to local planning decisions and are generated using explanation-based learning to explain why one plan is better than the other. *Control knowledge trees* helps to make globally optimal decisions by provide estimates of the quality of different planning alternatives. These trees consist of goal, operator and binding nodes for a planning problem and are generated from the problem search trace. QUALITY has been evaluated on the process planning domain and has been shown to significantly improve plan quality for this domain. Unfortunately, QUALITY does not necessarily improve planner efficiency and can sometimes produce much longer planning times. Additionally, QUALITY is built around a state-based planner and thus, it is unclear whether it could be successfully applied to a partial-order planner, especially since many rule conditions refer to current state information.

There are two other learning systems for improving quality which also run on the PRODIGY nonlinear planner. One technique was developed by (Iwamoto, 1994) and uses EBL to acquire control rules for near-optimal solutions in LSI design. This method is similar to QUALITY's control-rule learning algorithm, however it does not make use of the quality evaluation function to build the rule and is more limited in the type of control rules it can build. Iwamoto's system has also concentrated only on building control rules, unlike the QUALITY system which can also construct control knowledge trees to represent plan-quality control information. Iwamoto's system was able to moderately improve upon planner efficiency, however, it has only been tested on problems in a very limited domain.

Another learning system which has addressed improving plan quality is HAMLET. It is also built on the PRODIGY nonlinear planner and was discussed in Section 9.2. Similar to SCOPE HAMLET has been shown to improve upon both plan quality, as well as planner efficiency in the logistics transportation domain. However, the only quality metric examined by this system is the length of the plan, and thus, it is unclear whether HAMLET could improve upon other quality metrics as well. Also, as mentioned before, it is unclear how HAMLET would perform on a partial-order planner.

Most other work in plan quality has not utilized machine learning techniques. Some work has concentrated on examining goal interactions and how they related to solution quality. One technique is to analyze the different types of goal interactions and then develop strategies to deal them (Wilensky, 1983; Pollack, 1992). The LCOS (Least Commitment to Operator Selection) planning strategy (Hayes, 1990) takes a global view of the plan and only makes operator selections that can maximize a given plan quality criteria. Foulser, Li, and Yang (1992) promotes plan merging as a technique for minimizing plan cost, where certain operators in a plan are grouped together. Other work has utilized decision theory to improve plan quality. The PYRRHUS planning system (Williamson & Hanks, 1994) is an extension to UCPOP that finds optimal plans by using utility models to measure the quality of a partial plan. There have also been some domain-dependent approaches to generate high quality plans. The MACHINIST program (Hayes, 1990) generates plans for a machine process domain and has embedded knowledge about feature interaction which helps it to generate minimal length plans. The GARI (Descotte & Latombe, 1985) planner generates plans for metal cutting by using a constraint satisfaction algorithm, where domain control knowledge is encoded with more general domain information in the form of preference rules.

Chapter 10

Future Work

There are a number of issues that would be interesting to pursue in future research. First, there are several enhancements that could be beneficial for Scope's algorithm. These include incorporating a method for constructive induction that could invent new concepts when needed and learning rules for other types of decision points not currently considered by Scope. Second, further experiment evaluation should be performed on different types of domains and quality metrics, and more thoroughly comparing Scope to other learning methods. Third, it would be interesting to apply Scope to other types of planning algorithms to see if similar gains could be produced. This section discusses each of these topics and presents different ideas for implementation.

10.1 Enhancements to Scope's Algorithm

10.1.1 Constructive Induction and Shared Concepts

One possible enhancement to Scope is the incorporation of constructive induction to invent new predicates when needed. Currently, the pool of possible control rule antecedents is drawn mainly from program predicates which were used in the main planning algorithm. However, for complete optimization of a planner or other problem solver, it is often necessary to introduce new concepts for use in the control language. For instance, it may be helpful in the logistics domain, to have a predicate which tests whether two objects are initially in the same city. Currently, it is possible for the user to provide Scope with extra concepts definitions in the form of relational cliches or extra background knowledge. However, a better method would be for the system to learn these concepts automatically. Constructive induction techniques have been implemented in several ILP systems (Kijsirikul et al., 1992; Zelle & Mooney, 1994b) and could help Scope learn more accurate and efficient control rules.

Furthermore, it is often the case that concepts that are useful in making a control decision about a certain plan refinement are also useful for making decisions about other related refinements. However, even with constructive induction, these "shared" concepts must be relearned from scratch for each new control rule. In a Foil-like inductive learner, it is relatively easy to make a newly constructed concept available for reuse to simple adding it to the list of predicates that can be used as control rule antecedents. Unfortunately, the definition of this concept may be incomplete or incorrect, depending on the particular set of training examples that was used to construct it. This problem can be remedied by the following approach. If a previously defined concept is found to be the best choice for the next control-rule antecedent, yet it does not cover all the positive control examples, this concept can be redefined by merging the original set of control examples with the new set and then recursively inducing rules to cover all positive examples. This type of *concept sharing* procedure could help further reduce the inductive search space, and make SCOPE's learning algorithm more efficient.

10.1.2 Learning at Other Decision Points

SCOPE was designed to learn control rule for decision points that could be backtracked upon. These points are portrayed as clause selection problems, which enables control information to be easily incorporated. If learning is successful at all backtracking points, then control information is unnecessary at all other points. For instance, another planning decision point for which a number of past systems have learned control rules is goal selection. However, when using a UCPOP-style planner, goal selection is never backtracked upon (i.e. a correct plan can be found for any ordering of goals). Therefore, if accurate rules are learned for all other decision points (where backtracking does occur) then there is no need for control rules in goal selection. However, it is often the case that accurate control rules are not learned for all other points; SCOPE is sometimes unable to find a rule that can rule out all negative refinement-selection examples. In these cases, incorporating control information for goal selection could be beneficial.

Another decision point for which SCOPE does not learn control information is selecting variable bindings when establishing a goal using the initial state. For instance, if a goal has unbound variables (e.g. on-table(?X)) and there are several possible bindings in the initial state that could be used to establish the goal (e.g. the initial state contains both on-table(a) and on-table(b), then it is possible the decision of what bindings to select will be backtracked upon. In the experiments used to test SCOPE, this has not been an issue, but it could become important when testing on different types of domains or on when testing on more complicated problems. For instance, in the process planning domain, a relatively small number of machines were made available in problem initial states. Increasing the number of available tools and machines of a certain type could cause control information at this point to be necessary.

Both decision points of goal selection and binding selection are difficult to cast as clause-selection problems since it is not known prior to planning what the different options are. Both these decision correspond to selecting an appropriate item from a list. Thus, one possible way to write these decisions in the clause-selection format, is to use two clauses. The first clause would examine the *first* appropriate item off the list of options as a possible choice. The second clause would search the rest of the list for the desired item. Control rules could then be learned that instruct the planner when to select the first clause over the second clause, i.e. rules would evaluate when the first acceptable item should be chosen as opposed to searching farther in the list.

10.1.3 Evaluating Control Rule Utility

Currently, SCOPE automatically includes any learned rule in its final set of control rules, however, some rules are probably much more useful than others. The utility of individual rules can often dramatically vary and too many rules of low utility can even lead to lower performance (Minton, 1988). This occurrence, commonly known as the *utility problem*, can be lessened or prevented by only including the most useful control rules in the final planner. Though rule utility has thus far not been problematic, SCOPE may still be creating a few rules that are overly complex and apply to very few examples. Thus, it might be useful to incorporate a method into SCOPE that directly evaluates control-rule utility. Researchers have introduced a variety of techniques for determining the best rules to save (Greiner & Likuski, 1989; Markovitch & Scott, 1989; Subramanian & Feldman, 1990; Gratch & DeJong, 1992). As yet, no one has applied such techniques to evaluate rules for a partial-order planner, however, such a method should be easy to integrate into SCOPE's learning system.

10.1.4 Induction Bias

Another possible improvement is to replace or modify the standard FOIL information-gain heuristic currently used to bias SCOPE's induction algorithm towards good rules. Though the results with this heuristic have been encouraging, experimenting with different heuristics may be beneficial. One possibility is to replace this heuristic with a metric that more directly measures rule utility. This modification could improve performance by encouraging the system to only select highly-useful control rule antecedents. Another problem with the current heuristic is that a good rule is often discarded because it covers one or two negative control examples. Even if such a rule is retained, it will be considered "nondeterministic" (no cut will be added) and could be backtracked upon, causing a potential loss in speedup. SCOPE could benefit from methods for handling noisy data that have been in employed in FOIL and other related systems (Quinlan, 1990; Muggleton, 1992; Lavrač & Džeroski, 1994). In SCOPE's framework, a small percentage of incorrectly covered examples could be treated as noise, thereby allowing some good rules to be retained and more rules to be marked as deterministic. This procedure could cause even more backtracking to be eliminating, resulting in lower solution times for many examples. Problems that are not correctly covered by rules are already handled by retaining a backup of the original planner. Since most backtracking would be eliminated, the new planner should fail quickly on these examples, thus incurring very little extra time to solve them. Speedup gain on all other examples could make this approach beneficial overall.

10.1.5 Employing Failure Information

One last possible research direction is to utilize information about planning failures to learn more effective control rules. Currently, SCOPE only uses the generalized solution proofs of the training examples to bias the inductive search. Another approach is to also utilize explanations of why particular search paths failed. For instance, the generalized failure explanations created by the UCPOP+EBL learning system could also be used to bias the inductive search. This approach would allow control rules to directly utilize relevant failure information. In particular, rules would have access to the sets of conditions which caused many search paths to fail when solving the the training examples. This information could help SCOPE to build more effective rules which more accurately avoid unpromising search paths.

10.2 Further Experimental Evaluation

It would be interesting to conduct further experiments using SCOPE on other realistic domains and quality metrics. In the process planning domain, more extensive experiments could be run that tested a wider variety of part and problems types. For instance, in addition to the types of problems already tested, problems can require parts to have a particular surface finish or coating. Also, holes in a part can be further refined such as tapping a hole to produce a thread inside or countersinking a hole to cut an angular opening into the top.

A number of other domains could also to test SCOPE. For instance, the UM Translog domain (Andrews et al., 1995) is an extended version of logistics transportation domain, which it is an order of magnitude larger in size (41 actions vs. 6) and which provides more complex features and goal interactions. In this domain, packages must still be delivered between cities, however there are more modes of transportation, vehicles and packages have special loading methods, and some routes of transportation can be temporarily unavailable. The detailed set of operators in this domain provide for long plans with many possible solutions to the same problem. SCOPE could also be evaluated on different quality metrics. For instance in both the UM Translog domain and the Truckworld domain utilized by (Williamson & Hanks, 1994) resource consumption is an important quality measure.

Also, further experiments should be done comparing the multi-strategy learning approach used by SCOPE to a pure explanation-based approach, such as that used by UCPOP+EBL. In the tests comparing these two systems, it was unclear why the EBL approach was so greatly affected by both the type of domain definition used and the type of goal-selection strategy used by the planner. Further tests should also be done on whether the expressiveness of a domain has any effect on SCOPE's ability to learn control rules. It would be useful to compare these two systems on other types of domains to better evaluate their strengths and weaknesses.

10.3 Applying SCOPE to Other Planning Systems

In future work, it would be nice to demonstrate that SCOPE's learning is not limited to improving the performance of only one type of planner. There are several other types of planners besides partial-order that are prominent in the planning community today. One is an *hierarchical-task network* (HTN) planner (Erol et al., 1994a). As opposed to most *operator-based* planners, HTN planners specify plan modifications in terms of task reduction rules. These reduction rules are then used to decompose abstract goals into lower level tasks. A set of similar constraints as found in an operator-based planner (e.g. orderings, causal-links) is maintained in an HTN planner and many of the same methods can be used to resolve possible conflicts. HTN planners have several decision points where control knowledge could be useful, including what task reduction rule to apply, and what constraint resolution method to use. HTN planners are argued by some researchers to be more useful for real-world applications since they offer more flexibility in expressing domain knowledge. There are several well-developed HTN algorithms that would be good testbeds for SCOPE. These include UMCP (Erol et al., 1994b), developed at the University of Maryland, and COLLAGE (Lansky & Getoor, 1995), developed at the NASA Ames Research Center.

Another breed of planners that could be considered are those that use a combination of *plan-space* and *state-space* techniques. Kambhampati and Srivastava (1996) introduced the UCP planning algorithm which casts plan-space and state-space planning methods into a single framework. UCP has the freedom to interleave these two refinement strategies on a singular plan representation. Veloso and Stone (1995) have also developed a new approach to planning that combines these two refinement strategies by using a flexible approach to ordering commitments. FLECS can use both least-commitment and eager-commitment strategies and can vary its use across different problems and domains. This strategy also integrates techniques from both partial-order and total-order planners. Both UCP and FLECS could highly benefit from learned control knowledge since they have been especially designed to take advantage of heuristic knowledge at the choice points just described.

More recently, a new breed of planners has appeared that utilize propositional reasoning and theorem proving techniques to efficiently solve planning problems (Blum & Furst, 1997; Kautz & Selman, 1996). These types of planners operator very differently from more traditional planning styles, where planning is viewed as a state-based or plan-based search. For instance, Graphplan converts a planning problem into a structure called a *planning graph*, and then systematically searches the graph for a solution (Blum & Furst, 1997). Kautz and Selman (1996) describe a method for reducing planning problems into SAT encodings, and then employing either a satisfiability algorithm or a stochastic algorithm to produce solutions. Since, these approaches are quite different in nature to other types of planning systems, it is unclear if a learning system such as SCOPE could be effectively applied to improve performance. Graphplan seems the most likely candidate to benefit from SCOPE since it performs a backtracking search through the planning graph. During this search, it continuously attempts to find a set of actions which maps one set of goals to another until a solution is found. The decision of what action to select to achieve a goal can be backtracked upon, and thus would likely benefit from learned control knowledge. The other approach of reducing planning into SAT would be more difficult to use as a base planner for SCOPE, since it directly employs satisfiability algorithms, and thus search is performed in a much different space. However, using learning to improve performance in such algorithms could be beneficial and is worth further investigation.

Chapter 11

Conclusions

This dissertation has presented a novel approach for learning control knowledge for planning systems. This approach is set apart from other control-rule learners, by several key features. First, it learns control rules that can approve upon both planning efficiency and quality, as opposed to most other approaches which concentrate on one of these metrics. Second, it employs an inductive-logic programming framework and casts the problem of control-rule learning as a clause-selection problem where control information can be easily incorporated. Third, it uses a multi-strategy learning technique, that successfully combines both EBL and induction to acquire control rules.

SCOPE, the control-rule learning system developed in this dissertation, automatically acquires domain-specific control rules for a planner by examining past planning scenarios. Search-control is cast as a clause-selection problem for a Prolog program, where learned control knowledge is incorporated to help the planner immediately select promising plan refinements. Control rules are acquired by using a combination of machine learning techniques. In particular, explanation-based generalization is used to bias an inductive search for control rules towards useful control information. SCOPE's approach is shown successful at learning control rules for a partial-order planner, which is a style of planning to which few other learning systems can be successfully applied.

Experiments in several domains showed that SCOPE could significantly speedup a planner and at the same time, improve the quality of the generated solutions. In the logistics transportation and blocksworld domains, SCOPE was able to produce a much faster planner, and also was able to significantly reduce solution lengths, often to optimal solutions. Additionally, the scalability of SCOPE was demonstrated in the logistics domain, where it was able to generate a new planner that could solve problems at a much higher complexity level than could be solved before learning.

Experiments in the process planning domain demonstrated two main goals. First, SCOPE could successfully apply to a complex, realistic domain, which was over an order of magnitude larger than the other domains tested. Second, SCOPE was able to improve on different types of quality metrics, while still improving efficiency. In particular, SCOPE produced two sets of control rules that generated solutions emphasizing two separate quality metrics and that significantly improved both planning efficiency and the number of test problems that could be solved under a time limit.

SCOPE was also demonstrated to outperform a competing approach that used a purely explanation-based approach to acquire control knowledge. In these tests, an EBL approach was shown to improve planning performance on only one type of domain definition, while SCOPE was shown to produce significant improvements on a variety of domain types and thus appeared to be more robust than a system that used only EBL to learn rules. These experiments indicated that combining induction with EBL can be more effective than using EBL alone to acquire useful control knowledge.

In conclusion, this dissertation presents a novel learning approach for acquiring planning control knowledge, which can learn control information for improving both planning efficiency and quality, and which is one of the few methods proven successful for learning control rules to improve partial-order planners.

Appendix A

Domain Definitions

All domain definitions listed below are given in the following format. Domain operators are specified using the `operator/3` predicate which is defined as:

```
        operator(Op, Preconditions, Postconditions)
```

where `Op` corresponds to the operator name and arguments, `Preconditions` corresponds to a list of preconditions and `Effects` corresponds to a list of effects.

A.1 Blocksworld Domain

The main blocksworld domain definition (Nilsson, 1980) that was used in this dissertation is listed below.

```
operator(putdown(X),
         [holding(X)],
         [on_table(X),clear(X),arm_empty,not(holding(X))]).
operator(pickup(X),
         [on_table(X),clear(X),arm_empty],
         [holding(X),not(on_table(X)),not(clear(X)),not(arm_empty)]).
operator(unstack(X,Y),
         [on(X,Y),clear(X),arm_empty],
         [clear(Y),holding(X),not(on(X,Y)),not(clear(X)),not(arm_empty)]).
operator(stack(X,Y),
         [holding(X),clear(Y)],
         [on(X,Y),clear(X),arm_empty,not(holding(X)),not(clear(Y))]).
```

A.2 Three Versions of the Blocksworld Domain

Three different versions of blocksworld were used in comparing SCOPE to UCPOP+EBL in Chapter 9. There three version are listed below.

A.2.1 BW-Prop

This version contains only simple propositional preconditions and effects.

```
operator(newtower(X,Table,Z),
         [on(X,Z),clear(X),neq(X,Z),bloc(X),bloc(Z),tab(Table)],
         [on(X,Table),clear(Z),not(on(X,Z))]).
operator(puton(X,Y,Z),
         [on(X,Z),clear(X),clear(Y),neq(X,Y),neq(X,Z),neq(Y,Z),bloc(X),
          bloc(Y)],
         [on(X,Y),not(on(X,Z)),clear(Z),not(clear(Y))]).
```

A.2.2 BW-Cond

This version contains conditional effects.

```
operator(puton(X,Y,Z),
         [on(X,Z),clear(X),clear(Y),neq(X,Z),neq(Y,Z),neq(X,Y),bloc(X)],
         [on(X,Y),not(on(X,Z)),when(bloc(Z),clear(Z)),
          when(bloc(Y),not(clear(Y)))]).
```

A.2.3 BW-Univ

This version contains universal quantification in its preconditions.

```
operator(puton(X,Y,Z),
         [on(X,Z),neq(X,Y),neq(X,Z),neq(Y,Z),
          or([tab(Y),and([bloc(Y),forall(bloc(B),not(on(B,Y)))])]),
          forall(bloc(C),not(on(C,X)))],
         [on(X,Y),not(on(X,Z))]).
```

A.3 Logistics Transportation Domain

The logistics transportation definition (Veloso, 1992) that was used in this dissertation is listed below.

```
operator(load_truck(Obj,Truck,Loc),
         [at_obj(Obj,Loc),at_truck(Truck,Loc)],
         [inside_truck(Obj,Truck),not(at_obj(Obj,Loc))]).
operator(load_airplane(Obj,Airplane,Loc),
         [at_obj(Obj,Loc),at_airplane(Airplane,Loc)],
         [inside_airplane(Obj,Airplane),not(at_obj(Obj,Loc))]).
operator(unload_truck(Obj,Truck,Loc),
         [inside_truck(Obj,Truck),at_truck(Truck,Loc)],
         [at_obj(Obj,Loc),not(inside_truck(Obj,Truck))]).
operator(unload_airplane(Obj,Airplane,Loc),
         [inside_airplane(Obj,Airplane),at_airplane(Airplane,Loc)],
         [at_obj(Obj,Loc),not(inside_airplane(Obj,Airplane))]).
operator(drive_truck(Truck,Loc_from,Loc_to),
         [same_city(Loc_from,Loc_to),at_truck(Truck,Loc_from)],
         [at_truck(Truck,Loc_to),not(at_truck(Truck,Loc_from))]).
operator(fly_airplane(Airplane,Loc_from,Loc_to),
         [airport(Loc_to),neq(Loc_from,Loc_to),
          at_airplane(Airplane,Loc_from)],
         [at_airplane(Airplane,Loc_to),
          not(at_airplane(Airplane,Loc_from))]).
```

A.4 Process Planning Domain

The process planning definition (Gil, 1991) that was used in this dissertation is listed below. This domain contains definitions for operators, axioms and functions. Axioms can be used to easily define extra operator effects so that domain operators can be structured in a concise format. Functions are used to define operator preconditions that can be satisifed by calling a Prolog function. For more information on how axioms and functions are used in UCPOP, see Barrett and et al. (1995).

A.4.1 Process Planning Operators

```
operator(drill_with_spot_drill(Machine,Drill_Bit,Holding_Device,Part,Hole,Side),
         [is_a(Part,part),is_a(Machine,drill),is_a(Drill_Bit,spot_drill),
          holding_tool(Machine,Drill_Bit),
          holding(Machine,Holding_Device,Part,Side)],
         [not(is_clean(Part)),has_burrs(Part),has_spot(Part,Hole,Side,Loc_X,Loc_Y)]).

operator(drill_with_twist_drill(Machine,Drill_Bit,Holding_Device,Part,Hole,Side,
         Hole_Depth,Hole_Diameter),
         [is_a(Part,part),is_a(Machine,drill),eq(Drill_Bit_Diameter,Hole_Diameter),
          diameter_of_drill_bit(Drill_Bit,Drill_Bit_Diameter),
          is_a(Drill_Bit,twist_drill),has_spot(Part,Hole,Side,Loc_X,Loc_Y),
          holding_tool(Machine,Drill_Bit),
          holding(Machine,Holding_Device,Part,Side)],
         [not(is_clean(Part)),has_burrs(Part),not(has_spot(Part,Hole,Side,Loc_X,Loc_Y)),
          has_hole(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y)]).

operator(drill_with_high_helix_drill(Machine,Drill_Bit,Holding_Device,Part,Hole,
         Side,Hole_Depth,Hole_Diameter),
         [is_a(Part,part),is_a(Machine,drill),eq(Drill_Bit_Diameter,Hole_Diameter),
          diameter_of_drill_bit(Drill_Bit,Drill_Bit_Diameter),
          is_a(Drill_Bit,high_helix_drill),has_fluid(Machine,Fluid,Part),
          has_spot(Part,Hole,Side,Loc_X,Loc_Y),holding_tool(Machine,Drill_Bit),
          holding(Machine,Holding_Device,Part,Side)],
         [not(is_clean(Part)),has_burrs(Part),
          not(has_spot(Part,Hole,Side,Loc_X,Loc_Y)),
          has_hole(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y)]).

operator(drill_with_straight_fluted_drill(Machine,Drill_Bit,Holding_Device,Part,Hole,
         Side,Hole_Depth,Hole_Diameter),
         [is_a(Part,part),is_a(Machine,drill),eq(Drill_Bit_Diameter,Hole_Diameter),
          diameter_of_drill_bit(Drill_Bit,Drill_Bit_Diameter),
          is_a(Drill_Bit,straight_fluted_drill),smaller(Hole_Depth,2),
          material_of(Part,brass),has_spot(Part,Hole,Side,Loc_X,Loc_Y),
          holding_tool(Machine,Drill_Bit),holding(Machine,Holding_Device,Part,Side)],
         [not(is_clean(Part)),has_burrs(Part),
          not(has_spot(Part,Hole,Side,Loc_X,Loc_Y)),
          has_hole(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y)]).

operator(drill_with_oil_hole_drill(Machine,Drill_Bit,Holding_Device,Part,Hole,Side,
         Hole_Depth,Hole_Diameter),
         [is_a(Part,part),is_a(Machine,drill),eq(Drill_Bit_Diameter,Hole_Diameter),
          diameter_of_drill_bit(Drill_Bit,Drill_Bit_Diameter),
          is_a(Drill_Bit,oil_hole_drill),smaller(Hole_Depth,20),
          has_fluid(Machine,Fluid,Part),has_spot(Part,Hole,Side,Loc_X,Loc_Y),
          holding_tool(Machine,Drill_Bit),holding(Machine,Holding_Device,Part,Side)],
         [not(is_clean(Part)),has_burrs(Part),
          not(has_spot(Part,Hole,Side,Loc_X,Loc_Y)),
          has_hole(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y)]).

operator(drill_with_gun_drill(Machine,Drill_Bit,Holding_Device,Part,Hole,Side,
         Hole_Depth,Hole_Diameter),
         [is_a(Part,part),is_a(Machine,drill),eq(Drill_Bit_Diameter,Hole_Diameter),
          diameter_of_drill_bit(Drill_Bit,Drill_Bit_Diameter),is_a(Drill_Bit,gun_drill),
          has_fluid(Machine,Fluid,Part),has_spot(Part,Hole,Side,Loc_X,Loc_Y),
          holding_tool(Machine,Drill_Bit),holding(Machine,Holding_Device,Part,Side)],
         [not(is_clean(Part)),has_burrs(Part),not(has_spot(Part,Hole,Side,Loc_X,Loc_Y)),
          has_hole(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y)]).

operator(drill_with_center_drill(Machine,Drill_Bit,Holding_Device,Part,Hole,Side,
         Drill_Bit_Diameter,Loc_X,Loc_Y),
         [is_a(Part,part),is_a(Machine,drill),
          diameter_of_drill_bit(Drill_Bit,Drill_Bit_Diameter),
          eq(Drill_Bit_Diameter,Hole_Diameter),is_a(Drill_Bit,center_drill),
          has_spot(Part,Hole,Side,Loc_X,Loc_Y),holding_tool(Machine,Drill_Bit),
          holding(Machine,Holding_Device,Part,Side)],
         [not(is_clean(Part)),has_burrs(Part),not(has_spot(Part,Hole,Side,Loc_X,Loc_Y)),
          has_hole(Part,Hole,Side,1/8,Hole_Diameter,Loc_X,Loc_Y),
          has_center_hole(Part,Hole,Side,Loc_X,Loc_Y)]).

operator(tap(Machine,Drill_Bit,Holding_Device,Part,Hole),
         [is_a(Part,part),is_a(Machine,drill),eq(Drill_Bit_Diameter,Hole_Diameter),
          diameter_of_drill_bit(Drill_Bit,Drill_Bit_Diameter),is_a(Drill_Bit,tap),
          has_hole(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y),
          holding_tool(Machine,Drill_Bit),not(has_burrs(Part)),is_clean(Part),
          holding(Machine,Holding_Device,Part,Side)],
         [not(is_clean(Part)),has_burrs(Part),
          when(is_reamed(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y),
          not(is_reamed(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y))),
          is_tapped(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y)]).

operator(countersink(Machine,Drill_Bit,Holding_Device,Part,Hole),
         [is_a(Part,part),is_a(Machine,drill),angle_of_drill_bit(Drill_Bit,Angle),
          is_a(Drill_Bit,countersink),
          has_hole(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y),
          holding_tool(Machine,Drill_Bit),not(has_burrs(Part)),
          is_clean(Part),holding(Machine,Holding_Device,Part,Side)],
         [not(is_clean(Part)),has_burrs(Part),
          is_countersinked(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y,Angle)]).

operator(counterbore(Machine,Drill_Bit,Holding_Device,Part,Hole),
         [is_a(Part,part),is_a(Machine,drill),
          size_of_drill_bit(Drill_Bit,Counterbore_Size),is_a(Drill_Bit,counterbore),
          has_hole(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y),
          holding_tool(Machine,Drill_Bit),not(has_burrs(Part)),is_clean(Part),
          holding(Machine,Holding_Device,Part,Side)],
         [not(is_clean(Part)),has_burrs(Part),
          is_counterbored(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y,
          Counterbore_Size)]).

operator(ream(Machine,Drill_Bit,Holding_Device,Part,Hole,Side,Hole_Depth,Hole_Diameter),
         [is_a(Part,part),is_a(Machine,drill),eq(Drill_Bit_Diameter,Hole_Diameter),
          diameter_of_drill_bit(Drill_Bit,Drill_Bit_Diameter),is_a(Drill_Bit,reamer),
          smaller(Hole_Depth,2),has_fluid(Machine,Fluid,Part),
          has_hole(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y),
           holding_tool(Machine,Drill_Bit),not(has_burrs(Part)),is_clean(Part),
           holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           when(is_tapped(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y),
                not(is_tapped(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y))),
           is_reamed(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y)]).

operator(side_mill(Machine,Part,Milling_Cutter,Holding_Device,Side,Dim,Value),
          [is_a(Part,part),is_a(Machine,milling_machine),
           is_of_type(Milling_Cutter,milling_cutter),or([eq(Dim,width),eq(Dim,length)]),
           size_of(Part,Dim,Value_Old),smaller(Value,Value_Old),
           smaller_than_2in(Value_Old,Value),side_up_for_machining(Dim,Side),
           holding_tool(Machine,Milling_Cutter),holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),surface_finish_side(Part,Side,rough_mill),
           size_of(Part,Dim,Value),not(size_of(Part,Dim,Value_Old))]).

operator(face_mill(Machine,Part,Milling_Cutter,Holding_Device,Side,Dim,Value),
          [is_a(Part,part),is_a(Machine,milling_machine),
           is_of_type(Milling_Cutter,milling_cutter),eq(Dim,height),
           size_of(Part,Dim,Value_Old),smaller(Value,Value_Old),
           side_up_for_machining(Dim,Side),holding_tool(Machine,Milling_Cutter),
           holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),surface_finish_side(Part,Side,rough_mill),
           size_of(Part,Dim,Value),not(size_of(Part,Dim,Value_Old))]).

operator(drill_with_spot_drill_in_milling_machine(Machine,Drill_Bit,Holding_Device,Part,
          Hole,Side),
          [is_a(Part,part),is_a(Machine,milling_machine),is_a(Drill_Bit,spot_drill),
           holding_tool(Machine,Drill_Bit),holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),has_spot(Part,Hole,Side,Loc_X,Loc_Y)]).

operator(drill_with_twist_drill_in_milling_machine(Machine,Drill_Bit,Holding_Device,Part,
          Hole,Side,Hole_Depth,Hole_Diameter),
          [is_a(Part,part),is_a(Machine,milling_machine),
           eq(Drill_Bit_Diameter,Hole_Diameter),
           diameter_of_drill_bit(Drill_Bit,Drill_Bit_Diameter),is_a(Drill_Bit,twist_drill),
           has_spot(Part,Hole,Side,Loc_X,Loc_Y),holding_tool(Machine,Drill_Bit),
           holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),not(has_spot(Part,Hole,Side,Loc_X,Loc_Y)),
           has_hole(Part,Hole,Side,Hole_Depth,Hole_Diameter,Loc_X,Loc_Y)]).

operator(rough_turn_rectangular_part(Machine,Part,Toolbit,Holding_Device,Diameter_New),
          [is_a(Machine,lathe),is_a(Toolbit,rough_toolbit),shape_of(Part,rectangular),
           size_of(Part,height,H),size_of(Part,width,W),smaller(Diameter_New,H),
           smaller(Diameter_New,W),holding_tool(Machine,Toolbit),
           side_up_for_machining(diameter,Side),holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),not(size_of(Part,height,H)),
           not(size_of(Part,width,W)),size_of(Part,diameter,Diameter_New),
           not(surface_coating_side(Part,side1,Surface_Coating)),
           not(surface_coating_side(Part,side2,Surface_Coating)),
           not(surface_coating_side(Part,side4,Surface_Coating)),
           not(surface_coating_side(Part,side5,Surface_Coating)),
           not(surface_coating_side(Part,side0,Surface_Coating)),
           not(surface_finish_side(Part,side1,Sf1)),not(surface_finish_side(Part,side2,Sf2)),
           not(surface_finish_side(Part,side4,Sf4)),not(surface_finish_side(Part,side5,Sf5)),
           surface_finish_side(Part,side0,rough_turn)]).

operator(rough_turn_cylindrical_part(Machine,Part,Toolbit,Holding_Device,Diameter_New),
         [is_a(Machine,lathe),is_a(Toolbit,rough_toolbit),shape_of(Part,cylindrical),
          size_of(Part,diameter,Diameter),smaller(Diameter_New,Diameter),
          holding_tool(Machine,Toolbit),side_up_for_machining(diameter,Side),
          holding(Machine,Holding_Device,Part,Side)],
         [not(is_clean(Part)),has_burrs(Part),not(size_of(Part,diameter,Diameter)),
          size_of(Part,diameter,Diameter_New),
          not(surface_coating_side(Part,side0,Surface_Coating)),
          not(surface_finish_side(Part,side0,Sf)),
          surface_finish_side(Part,side0,rough_turn)]).

operator(finish_turn(Machine,Part,Toolbit,Holding_Device,Diameter_New),
         [is_a(Machine,lathe),is_a(Toolbit,finish_toolbit),shape_of(Part,cylindrical),
          size_of(Part,diameter,Diameter),finishing_size(Diameter,Diameter_New),
          holding_tool(Machine,Toolbit),not(has_burrs(Part)),is_clean(Part),
          holding(Machine,Holding_Device,Part,side0)],
         [not(is_clean(Part)),has_burrs(Part),not(size_of(Part,diameter,Diameter)),
          size_of(Part,diameter,Diameter_New),
          not(surface_coating_side(Part,side0,Surface_Coating)),
          not(surface_finish_side(Part,side0,Sf)),
          surface_finish_side(Part,side0,finish_turn)]).

operator(make_thread_with_lathe(Machine,Part,Holding_Device,Side),
         [is_a(Part,part),is_a(Machine,lathe),is_a(Toolbit,v_thread),
          shape_of(Part,cylindrical),holding_tool(Machine,Toolbit),not(has_burrs(Part)),
          is_clean(Part),holding(Machine,Holding_Device,Part,side0)],
         [not(is_clean(Part)),has_burrs(Part),
          not(surface_coating_side(Part,side0,Surface_Coating)),
          not(surface_finish_side(Part,side0,Sf)),surface_finish_side(Part,side0,tapped)]).

operator(make_knurl_with_lathe(Machine,Part,Holding_Device,Side),
         [is_a(Part,part),is_a(Machine,lathe),is_a(Toolbit,knurl),
          shape_of(Part,cylindrical),holding_tool(Machine,Toolbit),not(has_burrs(Part)),
          is_clean(Part),holding(Machine,Holding_Device,Part,side0)],
         [not(is_clean(Part)),has_burrs(Part),
          not(surface_coating_side(Part,side0,Surface_Coating)),
          not(surface_finish_side(Part,side0,Sf)),surface_finish_side(Part,side0,knurled)]).

operator(file_with_lathe(Machine,Part,Holding_Device,Lathe_File,Diameter_New),
         [is_a(Part,part),is_a(Machine,lathe),is_a(Lathe_File,lathe_file),
          shape_of(Part,cylindrical),size_of(Part,diameter,Diameter),
          finishing_size(Diameter,Diameter_New),not(has_burrs(Part)),is_clean(Part),
          holding(Machine,Holding_Device,Part,side0)],
         [not(is_clean(Part)),has_burrs(Part),not(size_of(Part,diameter,Diameter)),
          size_of(Part,diameter,Diameter_New),
          not(surface_coating_side(Part,side0,Surface_Coating)),
          not(surface_finish_side(Part,side0,Sf)),
          surface_finish_side(Part,side0,rough_grind)]).

operator(polish_with_lathe(Machine,Part,Holding_Device,Cloth),
         [is_a(Part,part),is_a(Machine,lathe),is_a(Cloth,abrasive_cloth),
          material_of_abrasive_cloth(Cloth,emery),shape_of(Part,cylindrical),
          not(has_burrs(Part)),is_clean(Part),holding(Machine,Holding_Device,Part,side0)],
         [not(is_clean(Part)),has_burrs(Part),
          not(surface_coating_side(Part,side0,Surface_Coating)),
          not(surface_finish_side(Part,side0,S_Q)),
          surface_finish_side(Part,side0,polished)]).

operator(rough_shape(Machine,Part,Cutting_Tool,Holding_Device,Side,Dim,Value),
         [is_a(Part,part),is_a(Machine,shaper),is_a(Cutting_Tool,roughing_cutting_tool),
          size_of(Part,Dim,Value_Old),smaller(Value,Value_Old),not(eq(Dim,diameter)),
          side_up_for_machining(Dim,Side),holding_tool(Machine,Cutting_Tool),
          holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),
           surface_finish_side(Part,Side,rough_shaped),size_of(Part,Dim,Value),
           not(size_of(Part,Dim,Value_Old))]).

operator(finish_shape(Machine,Part,Cutting_Tool,Holding_Device,Side,Dim,Value),
          [is_a(Part,part),is_a(Machine,shaper),is_a(Cutting_Tool,finishing_cutting_tool),
           size_of(Part,Dim,Value_Old),finishing_size(Value_Old,Value),not(eq(Dim,diameter)),
           side_up_for_machining(Dim,Side),holding_tool(Machine,Cutting_Tool),
           not(has_burrs(Part)),
           is_clean(Part),holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),
           surface_finish_side(Part,Side,finish_shaped),size_of(Part,Dim,Value),
           not(size_of(Part,Dim,Value_Old))]).

operator(rough_shape_with_planer(Machine,Part,Cutting_Tool,Holding_Device,Side,Dim,Value),
          [is_a(Part,part),is_a(Machine,planer),is_a(Cutting_Tool,roughing_cutting_tool),
           size_of(Part,Dim,Value_Old),smaller(Value,Value_Old),not(eq(Dim,diameter)),
           side_up_for_machining(Dim,Side),holding_tool(Machine,Cutting_Tool),
           holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),
           surface_finish_side(Part,Side,rough_planed),size_of(Part,Dim,Value),
           not(size_of(Part,Dim,Value_Old))]).

operator(finish_shape_with_planer(Machine,Part,Cutting_Tool,Holding_Device,Side,Dim,Value),
          [is_a(Part,part),is_a(Machine,planer),is_a(Cutting_Tool,finishing_cutting_tool),
           size_of(Part,Dim,Value_Old),finishing_size(Value_Old,Value),not(eq(Dim,diameter)),
           side_up_for_machining(Dim,Side),holding_tool(Machine,Cutting_Tool),
           not(has_burrs(Part)),is_clean(Part),holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),
           surface_finish_side(Part,Side,finish_planed),size_of(Part,Dim,Value),
           not(size_of(Part,Dim,Value_Old))]).

operator(rough_grind_with_hard_wheel(Machine,Part,Wheel,Holding_Device,Side,Dim,Value),
          [is_a(Part,part),is_a(Machine,grinder),is_a(Wheel,grinding_wheel),
           has_fluid(Machine,Fluid,Part),hardness_of_wheel(Wheel,hard),hardness_of(Part,soft),
           not(material_of(Part,bronze)),not(material_of(Part,copper)),
           grit_of_wheel(Wheel,coarse_grit),size_of(Part,Dim,Value_Old),
           smaller(Value,Value_Old),not(eq(Dim,diameter)),side_up_for_machining(Dim,Side),
           holding_tool(Machine,Wheel),holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),
           surface_finish_side(Part,Side,rough_grind),
           size_of(Part,Dim,Value),not(size_of(Part,Dim,Value_Old))]).

operator(rough_grind_with_soft_wheel(Machine,Part,Wheel,Holding_Device,Side,Dim,Value),
          [is_a(Part,part),is_a(Machine,grinder),is_a(Wheel,grinding_wheel),
           has_fluid(Machine,Fluid,Part),hardness_of_wheel(Wheel,soft),
           hardness_of(Part,hard),grit_of_wheel(Wheel,coarse_grit),
           size_of(Part,Dim,Value_Old),smaller(Value,Value_Old),not(eq(Dim,diameter)),
           side_up_for_machining(Dim,Side),holding_tool(Machine,Wheel),
           holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),
           surface_finish_side(Part,Side,rough_grind),size_of(Part,Dim,Value),
           not(size_of(Part,Dim,Value_Old))]).

operator(finish_grind_with_hard_wheel(Machine,Part,Wheel,Holding_Device,Side,Dim,Value),
          [is_a(Part,part),is_a(Machine,grinder),is_a(Wheel,grinding_wheel),
           has_fluid(Machine,Fluid,Part),hardness_of_wheel(Wheel,hard),hardness_of(Part,soft),
           not(material_of(Part,bronze)),not(material_of(Part,copper)),
           grit_of_wheel(Wheel,fine_grit),size_of(Part,Dim,Value_Old),
           finishing_size(Value_Old,Value),not(eq(Dim,diameter)),
           side_up_for_machining(Dim,Side),holding_tool(Machine,Wheel),not(has_burrs(Part)),
           is_clean(Part),holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),
           surface_finish_side(Part,Side,finish_grind),size_of(Part,Dim,Value),
           not(size_of(Part,Dim,Value_Old))]).

operator(finish_grind_with_soft_wheel(Machine,Part,Wheel,Holding_Device,Side,Dim,Value),
          [is_a(Part,part),is_a(Machine,grinder),is_a(Wheel,grinding_wheel),
           has_fluid(Machine,Fluid,Part),hardness_of_wheel(Wheel,soft),
           hardness_of(Part,hard),grit_of_wheel(Wheel,fine_grit),
           size_of(Part,Dim,Value_Old),finishing_size(Value_Old,Value),
           not(eq(Dim,diameter)),side_up_for_machining(Dim,Side),
           holding_tool(Machine,Wheel),not(has_burrs(Part)),is_clean(Part),
           holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),
           surface_finish_side(Part,Side,finish_grind),size_of(Part,Dim,Value),
           not(size_of(Part,Dim,Value_Old))]).

operator(cut_with_circular_cold_saw(Machine,Part,Attachment,Holding_Device,Dim,Value),
          [is_a(Part,part),is_a(Machine,circular_saw),is_a(Attachment,cold_saw),
           size_of(Part,Dim,Value_Old),smaller(Value,Value_Old),not(eq(Dim,diameter)),
           side_up_for_machining(Dim,Side),holding_tool(Machine,Attachment),
           holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),
           surface_finish_side(Part,Side,finish_mill),not(size_of(Part,Dim,Value_Old)),
           size_of(Part,Dim,Value)]).

operator(cut_with_circular_friction_saw(Machine,Part,Attachment,Holding_Device,Dim,Value),
          [is_a(Part,part),is_a(Machine,circular_saw),is_a(Attachment,friction_saw),
           has_fluid(Machine,Fluid,Part),size_of(Part,Dim,Value_Old),
           smaller(Value,Value_Old),not(eq(Dim,diameter)),side_up_for_machining(Dim,Side),
           holding_tool(Machine,Attachment),holding(Machine,Holding_Device,Part,Side)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),
           surface_finish_side(Part,Side,rough_mill),not(size_of(Part,Dim,Value_Old)),
           size_of(Part,Dim,Value)]).

operator(cut_with_band_saw(Machine,Part,Attachment,Dim,Value),
          [is_a(Part,part),is_a(Machine,band_saw),is_a(Attachment,band_file),
           size_of(Part,Dim,Value_Old),smaller(Value,Value_Old),not(eq(Dim,diameter)),
           side_up_for_machining(Dim,Side),holding_tool(Machine,Attachment),
           not(has_burrs(Part)),is_clean(Part),on_table(Machine,Part)],
          [not(is_clean(Part)),has_burrs(Part),
           not(surface_coating_side(Part,Side,Surface_Coating)),
           not(surface_finish_side(Part,Side,S_Q)),surface_finish_side(Part,Side,sawcut),
          not(size_of(Part,Dim,Value_Old)),size_of(Part,Dim,Value)]).

operator(polish_with_band_saw(Machine,Part,Attachment,Side),
         [is_a(Part,part),is_a(Machine,band_saw),is_a(Attachment,saw_band),
          side_up_for_machining(Dim,Side),holding_tool(Machine,Attachment),
          not(has_burrs(Part)),is_clean(Part),on_table(Machine,Part)],
         [not(is_clean(Part)),has_burrs(Part),
          not(surface_coating_side(Part,Side,Surface_Coating)),
          not(surface_finish_side(Part,Side,Old_Sf_Cond)),
          surface_finish_side(Part,Side,polished)]).

operator(weld_cylinders_metal_arc(Machine,Part1,Part2,Part,Electrode,Holding_Device,Length),
         [is_a(Part1,part),is_a(Part2,part),not(eq(Part1,Part2)),
          is_a(Machine,metal_arc_welder),is_a(Electrode,electrode),
          material_of(Part1,Material1),material_of(Part2,Material2),
          shape_of(Part1,cylindrical),shape_of(Part2,cylindrical),
          not(exists(has_hole(Part1,Hole,Side,Depth,Diameter,Loc_X,Loc_Y))),
          not(exists(has_hole(Part2,Hole,Side,Depth,Diameter,Loc_X,Loc_Y))),
          size_of(Part1,diameter,Diameter1),size_of(Part2,diameter,Diameter2),
          eq(Diameter1,Diameter2),size_of(Part1,length,Length1),
          size_of(Part2,length,Length2),new_size(Length1,Length2,Length),
          new_part(Part,Part1,Part2),new_material(Material,Material1,Material2),
          holding_tool(Machine,Electrode),holding(Machine,Holding_Device,Part2,side3)],
         [not(is_a(Part1,part)),not(is_a(Part2,part)),is_a(Part,part),
          material_of(Part,Material),size_of(Part,diameter,Diameter1),
          size_of(Part,length,Length),surface_finish_side(Part,side0,sawcut),
          forall([Sf31],when(surface_finish_side(Part1,side3,Sf31),
                 surface_finish_side(Part,side3,Sf31))),
          forall([Sf62],when(surface_finish_side(Part2,side6,Sf62),
                 surface_finish_side(Part,side6,Sf62))),
          not(holding(Machine,Holding_Device,Part2,side3)),
          holding(Machine,Holding_Device,Part,side3),not(size_of(Part1,diameter,Diameter)),
          not(size_of(Part1,length,Length1)),not(size_of(Part2,diameter,Diameter)),
          not(size_of(Part2,length,Length2)),not(material_of(Part1,Material1)),
          not(material_of(Part2,Material2)),not(is_clean(Part1)),not(is_clean(Part2)),
          forall([Sidea,Surf_Coatinga],not(surface_coating_side(Part1,Sidea,Surf_Coatinga))),
          forall([Sideb,Surf_Coatingb],not(surface_coating_side(Part2,Sideb,Surf_Coatingb))),
          forall([Sidec,Sfc],not(surface_finish_side(Part1,Sidec,Sfc))),
          forall([Sided,Sfd],not(surface_finish_side(Part2,Sided,Sfd)))]).

operator(weld_cylinders_gas(Machine,Part1,Part2,Part,Rod,Holding_Device,Length),
         [is_a(Part1,part),is_a(Part2,part),not(eq(Part1,Part2)),is_a(Machine,gas_welder),
          is_a(Rod,welding_rod),is_a(Torch,torch),material_of(Part1,Material1),
          material_of(Part2,Material2),eq(Material1,Material2),shape_of(Part1,cylindrical),
          shape_of(Part2,cylindrical),
          not(exists(has_hole(Part1,Hole,Side,Depth,Diameter,Loc_X,Loc_Y))),
          not(exists(has_hole(Part2,Hole,Side,Depth,Diameter,Loc_X,Loc_Y))),
          size_of(Part1,diameter,Diameter1),size_of(Part2,diameter,Diameter2),
          eq(Diameter1,Diameter2),size_of(Part1,length,Length1),
          size_of(Part2,length,Length2),new_size(Length1,Length2,Length),
          new_part(Part,Part1,Part2),holding(Machine,Holding_Device,Part2,side3)],
         [not(is_a(Part1,part)),not(is_a(Part2,part)),is_a(Part,part),
          material_of(Part,Material1),size_of(Part,diameter,Diameter1),
          size_of(Part,length,Length),surface_finish_side(Part,side0,sawcut),
          forall([Sf31],when(surface_finish_side(Part1,side3,Sf31),
                 surface_finish_side(Part,side3,Sf31))),
          forall([Sf62],when(surface_finish_side(Part2,side6,Sf62),
                 surface_finish_side(Part,side6,Sf62))),
          not(holding(Machine,Holding_Device,Part2,side3)),
          holding(Machine,Holding_Device,Part,side3),
          not(size_of(Part1,diameter,Diameter)),not(size_of(Part1,length,Length1)),
          not(size_of(Part2,diameter,Diameter)),not(size_of(Part2,length,Length2)),
          not(material_of(Part1,Material1)),not(material_of(Part2,Material2)),
          not(is_clean(Part1)),not(is_clean(Part2)),forall([Sidea,Surf_Coatinga],
          not(surface_coating_side(Part1,Sidea,Surf_Coatinga))),forall([Sideb,Surf_Coatingb],
          not(surface_coating_side(Part2,Sideb,Surf_Coatingb))),forall([Sidec,Sfc],
          not(surface_finish_side(Part1,Sidec,Sfc))),
          forall([Sided,Sfd],not(surface_finish_side(Part2,Sided,Sfd)))]).

operator(metal_spray_coating(Machine,Wire,Part,Side,Another_Machine,Holding_Device),
         [is_a(Part,part),is_a(Machine,electric_arc_spray_gun),
          is_a(Wire,spraying_metal_wire),not(material_of(Wire,tungsten)),
          not(material_of(Wire,molybdenum)),is_clean(Part),not(has_burrs(Part)),
          surface_coating_side(Part,Side,fused_metal),is_of_type(Another_Machine,machine),
          holding(Another_Machine,Holding_Device,Part,Side)],
         [when(material_of(Wire,stainless_steel),
               surface_coating_side(Part,Side,corrosion_resistant)),
          when(material_of(Wire,zirconium_oxide),
               surface_coating_side(Part,Side,heat_resistant)),
          when(material_of(Wire,aluminum_oxide),
               surface_coating_side(Part,Side,wear_resistant)),
          not(surface_coating_side(Part,Side,fused_metal))]).

operator(metal_spray_prepare(Machine,Wire,Part,Side,Another_Machine,Holding_Device),
         [is_a(Part,part),is_a(Machine,electric_arc_spray_gun),
          is_a(Wire,spraying_metal_wire),has_high_melting_point(Wire),is_clean(Part),
          not(has_burrs(Part)),is_of_type(Another_Machine,machine),
          holding(Another_Machine,Holding_Device,Part,Side)],
         [surface_coating_side(Part,Side,fused_metal)]).

operator(clean(Part),
         [is_a(Part,part),is_available_part(Part)],
         [is_clean(Part)]).

operator(remove_burrs(Part,Brush),
         [is_a(Part,part),is_a(Brush,brush),is_available_part(Part)],
         [not(is_clean(Part)),not(has_burrs(Part))]).

operator(put_tool_on_milling_machine(Machine,Attachment),
         [is_a(Machine,milling_machine),
          or([is_of_type(Attachment,milling_cutter),is_of_type(Attachment,drill_bit)]),
          is_available_tool_holder(Machine),is_available_tool(Attachment)],
         [holding_tool(Machine,Attachment)]).

operator(put_in_drill_spindle(Machine,Drill_Bit),
         [is_a(Machine,drill),is_of_type(Drill_Bit,drill_bit),
          is_available_tool_holder(Machine),is_available_tool(Drill_Bit)],
         [holding_tool(Machine,Drill_Bit)]).

operator(put_toolbit_in_lathe(Machine,Toolbit),
         [is_a(Machine,lathe),is_of_type(Toolbit,lathe_toolbit),
          is_available_tool_holder(Machine),is_available_tool(Toolbit)],
         [holding_tool(Machine,Toolbit)]).

operator(put_cutting_tool_in_shaper_or_planer(Machine,Cutting_Tool),
         [or([is_a(Machine,shaper),is_a(Machine,planer)]),
          is_of_type(Cutting_Tool,cutting_tool),is_available_tool_holder(Machine),
          is_available_tool(Cutting_Tool)],
         [holding_tool(Machine,Cutting_Tool)]).

operator(put_wheel_in_grinder(Machine,Wheel),
         [is_a(Machine,grinder),is_a(Wheel,grinding_wheel),
          is_available_tool_holder(Machine),is_available_tool(Wheel)],
         [holding_tool(Machine,Wheel)]).
```

```
operator(put_circular_saw_attachment_in_circular_saw(Machine,Attachment),
         [is_a(Machine,circular_saw),is_of_type(Attachment,circular_saw_attachment),
          is_available_tool_holder(Machine),is_available_tool(Attachment)],
         [holding_tool(Machine,Attachment)]).

operator(put_band_saw_attachment_in_band_saw(Machine,Attachment),
         [is_a(Machine,band_saw),is_of_type(Attachment,band_saw_attachment),
          is_available_tool_holder(Machine),is_available_tool(Attachment)],
         [holding_tool(Machine,Attachment)]).

operator(put_electrode_in_welder(Machine,Electrode),
         [is_a(Machine,metal_arc_welder),is_a(Electrode,electrode),
          is_available_tool_holder(Machine),is_available_tool(Electrode)],
         [holding_tool(Machine,Electrode)]).

operator(remove_tool_from_machine(Machine,Tool),
         [is_of_type(Machine,machine),is_of_type(Tool,machine_tool),
          holding_tool(Machine,Tool)],
         [not(holding_tool(Machine,Tool))]).

operator(put_holding_device_in_milling_machine(Machine,Holding_Device),
         [is_a(Machine,milling_machine),or([is_a(Holding_Device,four_jaw_chuck),
          is_a(Holding_Device,v_block),is_a(Holding_Device,vise),
          is_a(Holding_Device,collet_chuck),is_a(Holding_Device,toe_clamp)]),
          is_available_table(Machine,Holding_Device),
          is_available_holding_device(Holding_Device)],
         [has_device(Machine,Holding_Device)]).

operator(put_holding_device_in_drill(Machine,Holding_Device),
         [is_a(Machine,drill),or([is_a(Holding_Device,four_jaw_chuck),
          is_a(Holding_Device,v_block),is_a(Holding_Device,vise),
          is_a(Holding_Device,toe_clamp)]),is_available_table(Machine,Holding_Device),
          is_available_holding_device(Holding_Device)],
         [has_device(Machine,Holding_Device)]).

operator(put_holding_device_in_lathe(Machine,Holding_Device),
         [is_a(Machine,lathe),or([is_a(Holding_Device,centers),
          is_a(Holding_Device,four_jaw_chuck),is_a(Holding_Device,collet_chuck)]),
          is_available_table(Machine,Holding_Device),
          is_available_holding_device(Holding_Device)],
         [has_device(Machine,Holding_Device)]).

operator(put_holding_device_in_shaper(Machine,Holding_Device),
         [is_a(Machine,shaper),is_a(Holding_Device,vise),
          is_available_table(Machine,Holding_Device),
          is_available_holding_device(Holding_Device)],
         [has_device(Machine,Holding_Device)]).

operator(put_holding_device_in_planer(Machine,Holding_Device),
         [is_a(Machine,planer),is_a(Holding_Device,toe_clamp),
          is_available_table(Machine,Holding_Device),
          is_available_holding_device(Holding_Device)],
         [has_device(Machine,Holding_Device)]).

operator(put_holding_device_in_grinder(Machine,Holding_Device),
         [is_a(Machine,grinder),or([is_a(Holding_Device,magnetic_chuck),
          is_a(Holding_Device,v_block),is_a(Holding_Device,vise)]),
          is_available_table(Machine,Holding_Device),
          is_available_holding_device(Holding_Device)],
         [has_device(Machine,Holding_Device)]).
```

```
operator(put_holding_device_in_circular_saw(Machine,Holding_Device),
         [is_a(Machine,circular_saw),
          or([is_a(Holding_Device,vise),is_a(Holding_Device,v_block)]),
          is_available_table(Machine,Holding_Device),
          is_available_holding_device(Holding_Device)],
         [has_device(Machine,Holding_Device)]).

operator(put_holding_device_in_welder(Machine,Holding_Device),
         [is_of_type(Machine,welder),or([is_a(Holding_Device,vise),
          is_a(Holding_Device,toe_clamp)]),is_available_table(Machine,Holding_Device),
          is_available_holding_device(Holding_Device)],
         [has_device(Machine,Holding_Device)]).

operator(remove_holding_device_from_machine(Machine,Holding_Device),
         [is_of_type(Machine,machine),is_of_type(Holding_Device,holding_device),
          has_device(Machine,Holding_Device),
          is_empty_holding_device(Holding_Device,Machine)],
         [not(has_device(Machine,Holding_Device))]).

operator(add_soluble_oil(Machine,Fluid),
         [is_of_type(Machine,machine),is_a(Part,part),
          or([material_of(Part,steel),material_of(Part,aluminum)]),is_a(Fluid,soluble_oil)],
         [has_fluid(Machine,Fluid,Part)]).

operator(add_mineral_oil(Machine,Fluid),
         [is_of_type(Machine,machine),is_a(Part,part),is_a(Fluid,mineral_oil),
          material_of(Part,iron)],
         [has_fluid(Machine,Fluid,Part)]).

operator(add_any_cutting_fluid(Machine,Fluid),
         [is_of_type(Machine,machine),is_a(Part,part),
          or([material_of(Part,brass),material_of(Part,bronze),material_of(Part,copper)]),
          is_of_type(Fluid,cutting_fluid)],
         [has_fluid(Machine,Fluid,Part)]).

operator(put_on_machine_table(Machine,Part),
         [is_a(Part,part),is_of_type(Machine,machine),not(is_a(Machine,shaper)),
          is_available_part(Part),is_available_machine(Machine)],
         [not(on_table(Another_Machine,Part)),on_table(Machine,Part)]).

operator(put_on_shaper_table(Machine,Part),
         [is_a(Part,part),is_a(Machine,shaper),size_of_machine(Machine,Shaper_Size),
          size_of(Part,length,Part_Size),smaller(Part_Size,Shaper_Size),
          is_available_part(Part),is_available_machine(Machine)],
         [not(on_table(Another_Machine,Part)),on_table(Machine,Part)]).

operator(hold_with_v_block(Machine,Holding_Device,Part,Side),
         [is_of_type(Machine,machine),is_a(Part,part),is_a(Holding_Device,v_block),
          has_device(Machine,Holding_Device),not(has_burrs(Part)),
          is_clean(Part),on_table(Machine,Part),shape_of(Part,cylindrical),eq(Side,side0),
          is_empty_holding_device(Holding_Device,Machine),is_available_part(Part)],
         [not(on_table(Machine,Part)),holding_weakly(Machine,Holding_Device,Part,Side)]).

operator(hold_with_vise(Machine,Holding_Device,Part,Side),
         [is_of_type(Machine,machine),is_a(Part,part),is_a(Holding_Device,vise),
          has_device(Machine,Holding_Device),not(has_burrs(Part)),is_clean(Part),
          on_table(Machine,Part),is_empty_holding_device(Holding_Device,Machine),
          is_available_part(Part)],
         [not(on_table(Machine,Part)),
          when(shape_of(Part,cylindrical),holding_weakly(Machine,Holding_Device,Part,Side)),
          when(shape_of(Part,rectangular),holding(Machine,Holding_Device,Part,Side))]).
```

```
operator(hold_with_toe_clamp(Machine,Holding_Device,Part,Side),
         [is_of_type(Machine,machine),is_a(Part,part),is_a(Holding_Device,toe_clamp),
          has_device(Machine,Holding_Device),not(has_burrs(Part)),is_clean(Part),
          or([shape_of(Part,rectangular),eq(Side,side3),eq(Side,side6)]),
          on_table(Machine,Part),is_empty_holding_device(Holding_Device,Machine),
          is_available_part(Part)],
         [not(on_table(Machine,Part)),holding(Machine,Holding_Device,Part,Side)]).

operator(secure_with_toe_clamp(Machine,Holding_Device,Part,Side),
         [is_of_type(Machine,machine),is_a(Part,part),is_a(Holding_Device,toe_clamp),
          has_device(Machine,Holding_Device),not(has_burrs(Part)),is_clean(Part),
          shape_of(Part,cylindrical),
          holding_weakly(Machine,Another_Holding_Device,Part,Side),
          is_empty_holding_device(Holding_Device,Machine)],
         [not(on_table(Machine,Part)),holding(Machine,Holding_Device,Part,Side)]).

operator(hold_with_centers(Machine,Holding_Device,Part,Side),
         [is_of_type(Machine,machine),is_a(Part,part),is_a(Holding_Device,centers),
          has_device(Machine,Holding_Device),has_center_holes(Part),not(has_burrs(Part)),
          is_clean(Part),on_table(Machine,Part),shape_of(Part,cylindrical),
          is_empty_holding_device(Holding_Device,Machine),is_available_part(Part)],
         [not(on_table(Machine,Part)),holding(Machine,Holding_Device,Part,Side)]).

operator(hold_with_four_jaw_chuck(Machine,Holding_Device,Part,Side),
         [is_of_type(Machine,machine),is_a(Part,part),is_a(Holding_Device,four_jaw_chuck),
          has_device(Machine,Holding_Device),not(has_burrs(Part)),is_clean(Part),
          on_table(Machine,Part),is_empty_holding_device(Holding_Device,Machine),
          is_available_part(Part)],
         [not(on_table(Machine,Part)),holding(Machine,Holding_Device,Part,Side)]).

operator(hold_with_collet_chuck(Machine,Holding_Device,Part,Side),
         [is_of_type(Machine,machine),is_a(Part,part),is_a(Holding_Device,collet_chuck),
          has_device(Machine,Holding_Device),not(has_burrs(Part)),is_clean(Part),
          on_table(Machine,Part),shape_of(Part,cylindrical),
          is_empty_holding_device(Holding_Device,Machine),is_available_part(Part)],
         [not(on_table(Machine,Part)),holding(Machine,Holding_Device,Part,Side)]).

operator(hold_with_magnetic_chuck(Machine,Holding_Device,Part,Side),
         [is_of_type(Machine,machine),is_a(Part,part),is_a(Holding_Device,magnetic_chuck),
          has_device(Machine,Holding_Device),not(has_burrs(Part)),is_clean(Part),
          on_table(Machine,Part),is_empty_holding_device(Holding_Device,Machine),
          is_available_part(Part)],
         [not(on_table(Machine,Part)),holding(Machine,Holding_Device,Part,Side)]).

operator(remove_from_machine_table(Macine,Part),
         [is_of_type(Machine,machine),is_a(Part,part),on_table(Machine,Part),
          is_available_part(Part)],
         [not(on_table(Machine,Part))]).

operator(release_from_holding_device(Machine,Holding_Device,Part,Side),
         [is_of_type(Machine,machine),is_a(Part,part),
          is_of_type(Holding_Device,holding_device),
          holding(Machine,Holding_Device,Part,Side)],
         [not(holding(Machine,Holding_Device,Part,Side)),on_table(Machine,Part)]).

operator(release_from_holding_device_weak(Machine,Holding_Device,Part,Side),
         [is_of_type(Machine,machine),is_a(Part,part),
          is_of_type(Holding_Device,holding_device),
          holding_weakly(Machine,Holding_Device,Part,Side)],
         [not(holding_weakly(Machine,Holding_Device,Part,Side)),on_table(Machine,Part)]).
```

A.4.2 Process Planning Axioms

```
axiom(side_up_for_machining_length,
         [eq(Dim,length),or([eq(Side,side3),eq(Side,side6)])],
          side_up_for_machining(Dim,Side)).

axiom(side_up_for_machining_width,
         [eq(Dim,width),or([eq(Side,side2),eq(Side,side5)])],
          side_up_for_machining(Dim,Side)).

axiom(side_up_for_machining_height,
         [eq(Dim,height),or([eq(Side,side1),eq(Side,side4)])],
          side_up_for_machining(Dim,Side)).

axiom(side_up_for_machining_diameter,
         [eq(Dim,diameter),or([eq(Side,side1),eq(Side,side0)])],
          side_up_for_machining(Dim,Side)).

axiom(machine_available,
         [is_of_type(Machine,machine),not(exists(on_table(Machine,Other_Part)))],
          is_available_machine(Machine)).

axiom(tool_holder_available,
         [is_of_type(Machine,machine),not(exists(holding_tool(Machine,Tool)))],
          is_available_tool_holder(Machine)).

axiom(tool_available,
         [is_of_type(Tool,machine_tool),not(exists(holding_tool(Machine,Tool)))],
          is_available_tool(Tool)).

axiom(table_available,
         [is_of_type(Machine,machine),is_of_type(Holding_Device,holding_device),
          or([not(exists(has_device(Machine,Another_Holding_Device))),
  is_a(Holding_Device,toe_clamp)])],
          is_available_table(Machine,Holding_Device)).

axiom(holding_device_available,
         [is_of_type(Holding_Device,holding_device),
          not(exists(has_device(Machine,Holding_Device)))],
          is_available_holding_device(Holding_Device)).

axiom(part_available,
         [is_a(Part,part),not(exists(holding_weakly(Machine,Holding_Device,Part,Side))),
          not(exists(holding(Machine,Another_Holding_Device,Part,Side)))],
          is_available_part(Part)).

axiom(holding_device_empty,
         [is_of_type(Machine,machine),is_of_type(Holding_Device,holding_device),
          not(exists(holding_weakly(Machine,Holding_Device,Part,Side))),
          not(exists(holding(Machine,Holding_Device,Another_Part,Side)))],
          is_empty_holding_device(Holding_Device,Machine)).

axiom(is_rectangular,
         [is_a(Part,part),exists(size_of(Part,length,L)),exists(size_of(Part,width,W)),
          exists(size_of(Part,height,H))],
          shape_of(Part,rectangular)).

axiom(is_cylindrical,
         [is_a(Part,part),exists(size_of(Part,length,L)),exists(size_of(Part,diameter,D))],
          shape_of(Part,cylindrical)).

axiom(are_sides_of_rectangular_part,
         [is_a(Part,part),shape_of(Part,rectangular)],
          side_of(Part,side1)).

axiom(are_sides_of_rectangular_part,
         [is_a(Part,part),shape_of(Part,rectangular)],
          side_of(Part,side2)).

axiom(are_sides_of_rectangular_part,
         [is_a(Part,part),shape_of(Part,rectangular)],
          side_of(Part,side3)).

axiom(are_sides_of_rectangular_part,
         [is_a(Part,part),shape_of(Part,rectangular)],
          side_of(Part,side4)).

axiom(are_sides_of_rectangular_part,
         [is_a(Part,part),shape_of(Part,rectangular)],
          side_of(Part,side5)).

axiom(are_sides_of_rectangular_part,
         [is_a(Part,part),shape_of(Part,rectangular)],
          side_of(Part,side6)).

axiom(are_sides_of_cylindrical_part,
         [is_a(Part,part),shape_of(Part,cylindrical)],
          side_of(Part,side0)).

axiom(are_sides_of_cylindrical_part,
         [is_a(Part,part),shape_of(Part,cylindrical)],
          side_of(Part,side3)).

axiom(are_sides_of_cylindrical_part,
         [is_a(Part,part),shape_of(Part,cylindrical)],
          side_of(Part,side6)).

axiom(is_machined_surface_quality,
         [is_a(Part,part),or([surface_finish_side(Part,Side,rough_mill),
          surface_finish_side(Part,Side,rough_turn),
          surface_finish_side(Part,Side,rough_shaped),
          surface_finish_side(Part,Side,rough_planed),
          surface_finish_side(Part,Side,finish_planed),
          surface_finish_side(Part,Side,cold_rolled),
          surface_finish_side(Part,Side,finish_mill),
          surface_finish_side(Part,Side,finish_turn),
          surface_finish_quality_side(Part,Side,ground)])],
          surface_finish_quality_side(Part,Side,machined)).

axiom(is_ground_surface_quality,
         [is_a(Part,part),or([surface_finish_side(Part,Side,rough_grind),
          surface_finish_side(Part,Side,finish_grind)])],
          surface_finish_quality_side(Part,Side,ground)).

axiom(has_surface_finish_rectangular_part,
         [is_a(Part,part),shape_of(Part,rectangular),
          surface_finish_side(Part,side1,Surface_Finish),
          surface_finish_side(Part,side2,Surface_Finish),
          surface_finish_side(Part,side3,Surface_Finish),
          surface_finish_side(Part,side4,Surface_Finish),
          surface_finish_side(Part,side5,Surface_Finish),
          surface_finish_side(Part,side6,Surface_Finish)],
          surface_finish(Part,Surface_Finish)).
```

```
axiom(has_surface_finish_cylindrical_part,
         [is_a(Part,part),shape_of(Part,cylindrical),
          surface_finish_side(Part,side0,Surface_Finish),
          surface_finish_side(Part,side3,Surface_Finish),
          surface_finish_side(Part,side6,Surface_Finish)],
          surface_finish(Part,Surface_Finish)).

axiom(has_surface_coating_rectangular_part,
         [is_a(Part,part),shape_of(Part,rectangular),
          surface_coating_side(Part,side1,Surface_Coating),
          surface_coating_side(Part,side2,Surface_Coating),
          surface_coating_side(Part,side3,Surface_Coating),
          surface_coating_side(Part,side4,Surface_Coating),
          surface_coating_side(Part,side5,Surface_Coating),
          surface_coating_side(Part,side6,Surface_Coating)],
          surface_coating(Part,Surface_Coating)).

axiom(has_surface_coating_cylindrical_part,
         [is_a(Part,part),shape_of(Part,cylindrical),
          surface_coating_side(Part,side0,Surface_Coating),
          surface_coating_side(Part,side3,Surface_Coating),
          surface_coating_side(Part,side6,Surface_Coating)],
          surface_coating(Part,Surface_Coating)).

axiom(material_ferrous,
         [is_a(Part,part),or([material_of(Part,steel),material_of(Part,iron)])],
          alloy_of(Part,ferrous)).

axiom(material_non_ferrous,
         [is_a(Part,part),or([material_of(Part,brass),material_of(Part,copper),
          material_of(Part,bronze)])],
          alloy_of(Part,non_ferrous)).

axiom(hardness_of_material_soft,
         [is_a(Part,part),or([material_of(Part,aluminum),alloy_of(Part,non_ferrous)])],
          hardness_of(Part,soft)).

axiom(hardness_of_material_hard,
         [is_a(Part,part),alloy_of(Part,ferrous)],
          hardness_of(Part,hard)).

axiom(high_melting_point,
         [is_a(Wire,spraying_metal_wire),
          or([material_of(Wire,tungsten),material_of(Wire,molybdenum)])],
          has_high_melting_point(Wire)).

axiom(is_machine,
         [or([is_a(Machine,drill),is_a(Machine,lathe),is_a(Machine,shaper),
          is_a(Machine,planer),is_a(Machine,grinder),is_a(Machine,band_saw),
          is_a(Machine,circular_saw),is_a(Machine,milling_machine),
          is_of_type(Machine,welder)])],
          is_of_type(Machine,machine)).

axiom(is_welder,
         [or([is_a(Machine,metal_arc_welder),is_a(Machine,gas_welder)])],
          is_of_type(Machine,welder)).

axiom(is_tool,
         [or([is_of_type(Tool,machine_tool),is_of_type(Tool,operator_tool)])],
          is_of_type(Tool,tool)).

axiom(is_machine_tool,
         [or([is_of_type(Attachment,drill_bit),is_of_type(Attachment,lathe_toolbit),
          is_of_type(Attachment,cutting_tool),is_a(Attachment,grinding_wheel),
          is_of_type(Attachment,band_saw_attachment),
          is_of_type(Attachment,circular_saw_attachment),
          is_of_type(Attachment,milling_cutter),is_a(Attachment,electrode)])],
          is_of_type(Attachment,machine_tool)).

axiom(is_drill_bit,
         [or([is_a(Drill_Bit,spot_drill),is_a(Drill_Bit,center_drill),
          is_a(Drill_Bit,twist_drill),is_a(Drill_Bit,straight_fluted_drill),
          is_a(Drill_Bit,high_helix_drill),is_a(Drill_Bit,oil_hole_drill),
          is_a(Drill_Bit,gun_drill),is_a(Drill_Bit,core_drill),is_a(Drill_Bit,tap),
          is_a(Drill_Bit,countersink),is_a(Drill_Bit,counterbore),
          is_a(Drill_Bit,reamer)])],
          is_of_type(Drill_Bit,drill_bit)).

axiom(is_lathe_toolbit,
         [or([is_a(Toolbit,rough_toolbit),is_a(Toolbit,finish_toolbit),
          is_a(Toolbit,v_thread),is_a(Toolbit,knurl)])],
          is_of_type(Toolbit,lathe_toolbit)).

axiom(is_cutting_tool,
         [or([is_a(Cutting_Tool,roughing_cutting_tool),
          is_a(Cutting_Tool,finishing_cutting_tool)])],
          is_of_type(Cutting_Tool,cutting_tool)).

axiom(is_circular_saw_attachment,
         [or([is_a(Attachment,cold_saw),is_a(Attachment,friction_saw)])],
          is_of_type(Attachment,circular_saw_attachment)).

axiom(is_band_saw_attachment,
         [or([is_a(Attachment,saw_band),is_a(Attachment,band_file)])],
          is_of_type(Attachment,band_saw_attachment)).

axiom(is_milling_cutter,
         [or([is_a(Milling_Cutter,plain_mill),is_a(Milling_Cutter,end_mill)])],
          is_of_type(Milling_Cutter,milling_cutter)).

axiom(is_operator_tool,
         [or([is_a(Tool,lathe_file),is_a(Tool,abrasive_cloth),is_a(Tool,torch),
          is_a(Tool,welding_rod),is_a(Tool,spraying_metal_wire),is_a(Tool,brush)])],
          is_of_type(Tool,operator_tool)).

axiom(is_cutting_fluid,
         [or([is_a(Cutting_Fluid,soluble_oil),is_a(Cutting_Fluid,mineral_oil)])],
          is_of_type(Cutting_Fluid,cutting_fluid)).

axiom(is_holding_device,
         [or([is_a(Holding_Device,v_block),is_a(Holding_Device,vise),
          is_a(Holding_Device,toe_clamp),is_a(Holding_Device,centers),
          is_a(Holding_Device,four_jaw_chuck),is_a(Holding_Device,collet_chuck),
          is_a(Holding_Device,magnetic_chuck)])],
          is_of_type(Holding_Device,holding_device)).
```

A.4.3 Process Planning Functions

```
half_of(X,Y) :-
        (var(X) ->
         X = Y*2
        ; var(Y) ->
          Y = X/2
        ; TempX is Y*2,
          X =:= TempX
          ).

smaller(X,Y) :-
        (var(X) ->
         X = Y-0.5
        ; var(Y) ->
          Y = X+0.5
        ; X<Y
        ).

smaller_than_2in(X,Y) :-
        Temp is X-Y,
        Temp =< 2.

finishing_size(X,Y) :-
        (var(X) ->
         X is Y+0.002
        ; var(Y) ->
          Temp is X-0.002,
          (Temp > 0 ->
           Y is X - 0.002
          ; true
          )
        ; Temp is X-Y,
          Temp =< 0.003
        ).

new_size(D1,D2,D) :-
        (var(D) ->
         D is D1+D2
        ; Temp is D1+D2,
          Temp=D
        ).

new_part(Part,Part1,Part2) :-
        (var(Part) ->
         new_name(Part1,Part2,Part)
        ; true
        ).

new_material(Material,Material1,Material2) :-
        (var(Material) ->
         (Material1=Material2 ->
          Material = Material1
          ; new_name(Material1,Material2,Material)
         )
        ; true
        ).
```

Appendix B

Learned Control Rules

This appendix contains a set of learned control rules for the logistics transportation domain. These rules were learned in the experiments reported in Section 6.3 after SCOPE was trained on 100 training examples.

In the Prolog version of UCPOP used in this dissertation the following predicates are backtracking points and thus can have control rules learned for them. The `find_new_action` predicate selects a new action to achieve a goal. The `find_existing_action` selects an existing action to achieve a goal. The `select_action` predicate decides whether a new or existing action should be used to achieve a goal. The `apply_constraint` predicate decides what threat resolution strategy to use to resolve a threat. In the logistics domain, control rules were learned for all of these points, except for the `apply_constraint` predicate, which was never backtracked upon in these experiments.

```
find_new_action(at_obj(B,C),D,_,_,_,E,no_cond,action(_,unload_truck(B,A,C),
              [inside_truck(B,A),at_truck(A,C)],
              [at_obj(B,C),not(inside_truck(B,A))])) :-
       \+member_goal(goal(at_truck(_,C),D),E).

find_existing_action(at_truck(B,C),D,E,F,_,_,no_cond,[],0,
                   action(0,start,[],A)) :-
       nonvar(C),
       member_action(action(0,start,[],A),E),
       member_pred(at_truck(B,C),A),
       \+nonfixable_threats_to_new_link(0,at_truck(B,C),D,E,F).

find_existing_action(at_airplane(B,C),D,E,F,_,_,no_cond,[],0,
                   action(0,start,[],A)) :-
       member_action(action(0,start,[],A),E),
       member_pred(at_airplane(B,C),A),
       \+threats_to_new_link(0,at_airplane(B,C),D,E,F).

find_existing_action(at_airplane(B,C),D,E,F,_,_,no_cond,[],0,
                   action(0,start,[],A)) :-
       nonvar(C),
       member_action(action(0,start,[],A),E),
       member_pred(at_airplane(B,C),A),
        \+nonfixable_threats_to_new_link(O,at_airplane(B,C),D,E,F).


select_action(goal(at_obj(D,E),F),G,H,I,O,J,K,L,M,N,O,P,old) :-
        find_existing_action(at_obj(D,E),F,G,H,I,O,P,[],C,J),
        can_add_ordering(C,F,H),
        add_clink(C,at_obj(D,E),F,I,K,N),
        add_ordering(C,F,H,M),
        L=G,
        find_init_state(G,B),
        find_goal_state(G,A),
        intercity_delivery(D,B,A).

select_action(goal(at_obj(C,D),E),F,G,H,N,I,J,K,L,M,N,O,old) :-
        find_existing_action(at_obj(C,D),E,F,G,H,N,O,[],B,I),
        can_add_ordering(B,E,G),
        add_clink(B,at_obj(C,D),E,H,J,M),
        add_ordering(B,E,G,L),
        K=F,
        find_init_state(F,A),
        member_goal(goal(at_airplane(_,D),_),N),
        member_pred(airport(D),A).

select_action(goal(at_obj(B,C),D),E,F,G,M,H,I,J,K,L,M,N,old) :-
        find_existing_action(at_obj(B,C),D,E,F,G,M,N,[],A,H),
        can_add_ordering(A,D,F),
        add_clink(A,at_obj(B,C),D,G,I,L),
        add_ordering(A,D,F,K),
        J=E,
        member_goal(goal(at_airplane(_,C),_),M),
        member_goal(goal(at_truck(_,C),D),M).

select_action(goal(at_truck(C,D),E),F,G,H,N,I,J,K,L,M,N,O,old) :-
        find_existing_action(at_truck(C,D),E,F,G,H,N,O,[],B,I),
        can_add_ordering(B,E,G),
        add_clink(B,at_truck(C,D),E,H,J,M),
        add_ordering(B,E,G,L),
        K=F,
        find_init_state(F,A),
        member_pred(airport(D),A),
        \+ (member_pred(at_truck(C,D),A),
            nonfixable_threats_to_new_link(B,at_truck(C,D),E,F,G)).

select_action(goal(at_truck(B,C),D),E,F,G,M,H,I,J,K,L,M,N,old) :-
        find_existing_action(at_truck(B,C),D,E,F,G,M,N,[],A,H),
        can_add_ordering(A,D,F),
        add_clink(A,at_truck(B,C),D,G,I,L),
        add_ordering(A,D,F,K),
        J=E,
        can_add_ordering(D,A,F).

select_action(goal(at_truck(B,C),D),E,F,G,M,H,I,J,K,L,M,N,old) :-
        find_existing_action(at_truck(B,C),D,E,F,G,M,N,[],A,H),
        can_add_ordering(A,D,F),
        add_clink(A,at_truck(B,C),D,G,I,L),
        add_ordering(A,D,F,K),
        J=E,
        member_action(action(D,drive_truck(B,C,_),_,_),E).

select_action(goal(at_truck(B,C),D),E,F,G,M,H,I,J,K,L,M,N,old) :-
        var(B),
        find_existing_action(at_truck(B,C),D,E,F,G,M,N,[],A,H),
        can_add_ordering(A,D,F),
        add_clink(A,at_truck(B,C),D,G,I,L),
        add_ordering(A,D,F,K),
        J=E.

select_action(goal(at_airplane(B,C),D),E,F,G,M,H,I,J,K,L,M,N,old) :-
        find_existing_action(at_airplane(B,C),D,E,F,G,M,N,[],A,H),
        can_add_ordering(A,D,F),
        add_clink(A,at_airplane(B,C),D,G,I,L),
        add_ordering(A,D,F,K),
        J=E,
        can_add_ordering(D,A,F).

select_action(goal(at_airplane(B,C),D),E,F,G,M,H,I,J,K,L,M,N,old) :-
        find_existing_action(at_airplane(B,C),D,E,F,G,M,N,[],A,H),
        can_add_ordering(A,D,F),
        add_clink(A,at_airplane(B,C),D,G,I,L),
        add_ordering(A,D,F,K),
        J=E,
        member_action(action(D,fly_airplane(B,C,_),_,_),E).

select_action(goal(at_airplane(B,C),D),E,F,G,M,H,I,J,K,L,M,N,old) :-
        nonvar(C),
        find_existing_action(at_airplane(B,C),D,E,F,G,M,N,[],A,H),
        can_add_ordering(A,D,F),
        add_clink(A,at_airplane(B,C),D,G,I,L),
        add_ordering(A,D,F,K),
        J=E.
```

Bibliography

Agosa, J. M., & Wilkins, D. E. (1996). Using SIPE-2 to plan emergency response to marine oil spills. *IEEE Expert*, *11*, 6–8.

Andrews, S., Kettler, B., Erol, K., & Hendler, J. (1995). UM Translog: A planning domain for the development and benchmarking of planning systems. Tech. rep. CS-TR-3487, Institute for Advanced Computer Studies, University of Maryland.

Barrett, A., & Weld, D. (1994). Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, *67*, 71–112.

Barrett, A., & et al. (1995). UCPOP: User's manual (version 4.0). Tech. rep. 93-09-06d, Department of Computer Science and Engineering, University of Washington.

Barrett, A., Golden, K., Penberthy, S., & Weld, D. (1993). UCPOP: User's manual (version 2.0). Tech. rep. 93-09-06, Department of Computer Science and Engineering, University of Washington.

Bhatnagar, N., & Mostow, J. (1994). On-line learning from search failure. *Machine Learning*, *15*, 69–117.

Blum, A., & Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, *90*, 281–300.

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

Borrajo, D., & Veloso, M. (1997). Lazy incremental learning of control knowledge for efficiently obtaining quality plans. *Artificial Intelligence Review*, *11*, 371–405.

Cameron-Jones, R. M., & Quinlan, J. R. (1994). Efficient top-down induction of logic programs. *SIGART Bulletin*, *1*(5), 33–42.

Carbonell, J., & et al. (1992). PRODIGY4.0: The manual and tutorial. Tech. rep. CMU-CS-92-150, School of Computer Science, Carnegie Mellon University, Pittsburg, PA.

Chase, M., Zweben, M., Piazza, R., Burger, J., Maglio, P., & Hirsh, H. (1989). Approximating learned search control knowledge. In *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 40–42 Ithaca, NY.

Chien, S. (1989). Using and refining simplifications: Explanation-based learning of plans in intractable domains. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 590–595 Detroit, MI.

Chien, S., & DeJong, G. (1994). Incremental reasoning in explanation-based learning of plans. In *Proceedings of the Second International Conference of AI Planning Systems* Chicago.

Chien, S., Govindjee, A., Estlin, T., Wang, X., & Jr., R. H. (1997). Automated generation of antenna operation procedures: A knowledge-based approach. *Telecommunications and Data Acquisition*, *142*.

Cohen, W. W. (1990). Learning approximate control rules of high utility. In *Proceedings of the Seventh International Conference on Machine Learning*, pp. 268–276 Austin, TX.

DeJong, G. F., & Mooney, R. J. (1986). Explanation-based learning: An alternative view. *Machine Learning*, *1*(2), 145–176. Reprinted in *Readings in Machine Learning*, J. W. Shavlik and T. G. Dietterich (eds.), Morgan Kaufman, San Mateo, CA, 1990.

Descotte, Y., & Latombe, J.-C. (1985). Making compromises among antagonist constraints in a planner. *Artificial Intelligence*, *27*, 183–217.

Erol, K., Nau, D., & Hendler, J. (1994a). HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 1123–1128 Seattle.

Erol, K., Nau, D., & Hendler, J. (1994b). UMCP: A sound and complete planning procedure for hierarchical task-network planning. In *Proceedings of the Second International Conference of AI Planning Systems* Chicago.

Estlin, T. A., & Mooney, R. J. (1996). Multi-strategy learning of search control for partial-order planning. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pp. 843–848 Portland, OR.

Etzioni, O. (1993). Acquiring search control knowledge via static analysis. *Artificial Intelligence*, *60*(2).

Etzioni, O., & Etzioni, R. (1994). Statistical methods for analyzing speedup learning experiments. *Machine Learning*, *14*, 337–347.

Fargher, H., & Smith, R. (1994). Planning in a flexible semiconductor manufacturing environment. In Zweben, M., & Fox, M. (Eds.), *Intelligent Scheduling*, pp. 545–580. Morgan Kaufmann, San Francisco, CA.

Fikes, R., & Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *2*(3/4).

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

Foulser, D., Li, M., & Yang, Q. (1992). Theory and algorithms for plan merging. *Artificial Intelligence*, *52*, 143–181.

Gerevini, A., & Schubert, L. (1996). Accelerating partial-order planners: Some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research*, *5*, 95–137.

Gil, Y. (1991). A specification of manufacturing processes for planning. Tech. rep. CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Gratch, J., & DeJong, G. (1992). COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 235–240 San Jose, CA.

Greiner, R., & Likuski, J. (1989). Incorporating redundant learned rules: A preliminary formal analysis of ebl. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 744–749 Detroit, MI.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost parts. *IEEE transactions on SSC*, *4*, 100–107.

Hayes, C. (1990). *Machining Planning: A Model of an Expert Level Planning Process*. Ph.D. thesis, The Robotics Institue, Carnegie Mellon University.

Iwamoto, M. (1994). A planner with quality goal and its speedup learning for optimization problem. In *Proceedings of the Second International Conference of AI Planning Systems* Chicago.

Kambhampati, S., & Chen, J. (1993). Relative utility of EBG based plan reuse in partial ordering vs. total ordering. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 514–519 Washington, D.C.

Kambhampati, S., Katukam, S., & Qu, Y. (1996). Failure driven search control for partial order planners: An explanation based approach. *Artificial Intelligence*, *88*.

Kambhampati, S., & Srivastava, B. (1996). Universal classical planner: An algorithm for unifying state-space and plan-space planning. In Ghallab, M., & Milani, A. (Eds.), *New Directions in AI Planning*, pp. 61–75 Amsterdam. IOS Press.

Kautz, H., & Selman, B. (1996). Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Ariticial Intelligence*, pp. 1194–1200 Portland, OR.

Keller, R. (1987). *The Role of Explicit Contextual Knowledge in Learning Concepts to Improve Performance*. Ph.D. thesis, Rutgers University, New Brunswick, N. Also appears as tech. report ML-TR-7.

Kijsirikul, B., Numao, M., & Shimura, M. (1992). Discrimination-based constructive induction of logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 44–49 San Jose, CA.

Korf, R. (1985). Depth-first iterative-deepening: An optimal admissable tree search. *Artificial Intelligence*, *27*(1).

Langley, P. (1985). Learning to search: From weak methods to domain specific heuristics. *Cognitive Science*, *9*(2), 217–260.

Langley, P., & Allen, J. (1991). The acquisition of human planning expertise. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 80–84 Evanston,IL.

Lansky, A., & Getoor, L. (1995). Scope and abstraction: Two criteria for localized planning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pp. 1612–1618 Montreal, CA.

Lavrač, N., & Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood.

Leckie, C., & Zuckerman, I. (1993). An inductive approach to learning search control rules for planning. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1100–1105 Chamberry,France.

Markovitch, S., & Scott, P. D. (1989). Utilization filtering: A method for reducing the inherent harmfulness of deductively learning knowledge. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 738–743 Detroit, MI.

McDermott, D. (1991). Regression planning. *International Journal of Intelligent Systems*, *6*, 357–416.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp. 564–569 St. Paul, MN.

Minton, S. (1989). Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, *40*, 63–118.

Minton, S. N. (1988). *Learning Effective Search Control Knowledge: An Explanantion-Based Approach.* Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA.

Minton, S. (1985). Selectively generalizing plans for problem solving. In *Proceedings of the Ninth International Conference on Ariticial Intelligence*, pp. 569–599 Los Angeles, CA.

Minton, S., Drummond, M., Bresina, J. L., & Phillips, A. B. (1992). Total order vs. partial order planning: Factors influencing performance. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pp. 83–92 Cambridge,CA.

Mitchell, T., Utgoff, T., & Banerji, R. (1983). Learning problem solving heuristics by experimentation. In Michalski, R., Mitchell, T., & Carbonell, J. (Eds.), *Machine Learning: An Artificial Intelligence Approach.* Morgan Kaufmann, Palo Alto, CA.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, *1*(1), 47–80.

Mooney, R. J. (1989). The effect of rule use on the utility of explanation-based learning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 725–730 Detroit, MI.

Mooney, R. J., & Califf, M. E. (1995). Induction of first-order decision lists: Results on learning the past tense of English verbs. *Journal of Artificial Intelligence Research*, *3*, 1–24.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pp. 339–352 Ann Arbor, MI.

Muggleton, S., King, R., & Sternberg, M. (1992). Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, *5*(7), 647–657.

Muggleton, S. H. (Ed.). (1992). *Inductive Logic Programming.* Academic Press, New York, NY.

Nilsson, N. (1980). *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA.

Pazzani, M., & Kibler, D. (1992). The utility of background knowledge in inductive learning. *Machine Learning*, *9*, 57–94.

Penberthy, J., & Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pp. 113–114 Cambridge,MA.

Pendault, E. (1989). ADL: Exploring the middle ground between strips and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning.*

Pérez, M. A. (1995). *Learning Search Control Knowledge to Improve Plan Quality.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

Pérez, M. A. (1996). Representing and learning quality-improving search control knowledge. In *Proceedings of the Thirteenth International Conference on Machine Learning* Bari,Italy.

Pérez, M. A., & Carbonell, J. (1994). Control knowledge to improve the plan quality. In *Proceedings of the Second International Conference of AI Planning Systems* Chicago, IL.

Poet, M. A., & Smith, D. E. (1993). Threat-removal strategies for partial-order planning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pp. 492–499 Washington, D.C.

Pollack, M. E. (1992). The uses of plans. *Artificial Intelligence*, *57*, 43–68.

Porter, B. W., & Kibler, D. F. (1986). Experimental goal regression: A method for learning problem-solving. *Machine Learning*, *1*(3), 249–285.

Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess end games. In Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (Eds.), *Machine Learning: An Artificial Intelligence Approach.* Morgan Kaufmann, Los Altos, CA.

Quinlan, J. R. (1991). Determinate literals in inductive logic programming. In *Proceedings of the Eighth International Workshop on Machine Learning* Evanston, IL.

Quinlan, J. R., & Cameron-Jones, R. M. (1993). FOIL: A midterm report. In *Proceedings of the European Conference on Machine Learning*, pp. 3–20 Vienna.

Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, *5*(3), 239–266.

Reddy, C., & Tadepalli, P. (1997). Learning goal-decomposition rules using exercises. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pp. 278–286 Nashville,TN.

Silverstein, G., & Pazzani, M. J. (1991). Relational clichés: Constraining constructive induction during relational learning. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 203–207 Evanston, IL.

Subramanian, D., & Feldman, R. (1990). The utility of EBL in recursive domains. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 942–949 Boston, MA.

Tadepalli, P. (1989). Lazy explanation-based learning: A solution to the intractable theory problem. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* Detroit, MI.

Veloso, M., Carbonell, J., Pérez, A., Borrajo, D., Fink, E., & Blythe, J. (1995). Integrated planning and learning: The PRODIGY architecture. *Journal of Theoretical and Experimental Artificial Intelligence Research*, *7*(1).

Veloso, M., & Stone, P. (1995). FLECS: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research*, *3*, 25–52.

Veloso, M. M. (1992). *Learning by Analogical Reasoning in General Problem Solving.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

Warren, D. (1974). WARPLAN: A system for generating plans. Tech. rep. Memo No. 76, Department of Computational Logic, University of Edinburgh.

Weld, D. (1994). An introduction to least commitment planning. *AI Magazine*, *15*(4), 27–61.

Wilensky, R. W. (1983). *Planning and Understanding: A Computational Approach to Human Reasoning.* Addison-Wesley, Reading, MA.

Williamson, M., & Hanks, S. (1994). Optimal planning with a goal-directed utility model. In *Proceedings of the Second International Conference of AI Planning Systems*, pp. 176–181 Chicago.

Winston, P. H., Binford, T. O., Katz, B., & Lowry, M. (1983). Learning physical descriptions from functional definitions, examples, and precedents. In *Proceedings of the Third National Conference on Artificial Intelligence*, pp. 433–439 Washington, D.C.

Zelle, J. M., & Mooney, R. J. (1993). Combining FOIL and EBG to speed-up logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pp. 1106–1111 Chambery, France.

Zelle, J. M., & Mooney, R. J. (1994a). Combining top-down and bottom-up methods in inductive logic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 343–351 New Brunswick, NJ.

Zelle, J. M., & Mooney, R. J. (1994b). Inducing deterministic Prolog parsers from treebanks: A machine learning approach. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 748–753 Seattle, WA.

Zweben, M., Davis, E., Daun, B., Drascher, E., Deale, M., & Eskey, M. (1992). Learning to improve constraint-based scheduling. *Artificial Intelligence*, *58*, 271–296.