LEARNING CONTROL KNOWLEDGE FOR PLANNING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Yi-Cheng Huang

January 2003

LEARNING CONTROL KNOWLEDGE FOR PLANNING

Yi-Cheng Huang, Ph.D.

Cornell University 2003

Planning is a notoriously hard combinatorial search problem. Recently, new planning paradigms, such as constraint-based and heuristic search based planning techniques have significantly extended the range of practically feasible planning tasks. To obtain a better understanding of their relative strengths and weaknesses, we will present a detailed comparison of the performance of the various recent planning techniques on multi-agent tasks. Our analysis shows that, in a multi-agent setting, heuristic and constraint-based planners often complement each other.

An interesting aspect of these planners is that they do not incorporate domain-specific control knowledge, but instead rely on efficient data representations and search techniques. An alternative approach has been proposed in the TLPLAN system which incorporates declarative control specified in temporal logic formulas. We will show how these control rules can be parsed into the constraint-based planner BLACKBOX so as to greatly improve the system performance.

A motivating question is how to automatically acquire these search controls from the application domain. Despite the long history of research in using machine learning to speed up state-space planning, the techniques that have been developed are not yet in widespread

use for practical planning systems. One limiting factor is that traditional domain-independent planning systems scale so poorly that extensive learned control knowledge is required to raise their performance to an acceptable level. We will present the first positive results on automatic acquisition of such high-level, purely declarative constraints for constraint-based planners using machine learning techniques. In particular, we will show that a new heuristic method for generating training examples, together with a rule induction algorithm, can learn useful control rules in a variety of domains and boost system performance significantly.

ACKNOWLEDGMENTS

I would like to greatly express my appreciation to my advisor, Bart Selman, for his guidance and kindness during the past four years. It was from him I learned to identify and tackle interesting problems, a skill I believe will be a valuable asset in my later career development. He has also been a source of great support for my research, and I therefore am able to complete my dissertation under his advice. Most especially, I am deeply indebted to him for his valuable time given to review and edit my writing. It has been a privilege and an honor to work with him at Cornell. I would also like to thank Professor Claire Cardie and Professor Stephen B. Wicker for their useful suggestions and for serving on my committee.

I also wish to thank Henry Kautz, my mentor when I worked at AT&T research laboratory as a summer intern, for his hard work on the research we have done together and for his many valuable suggestions.

Finally, and most importantly, I would like to thank my parents and my sister for their generosity in relieving me of my responsibility to the family during my academic pursuits.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Imagine a hypothetical housekeeping robot sometime in the future. When the robot receives an order to deliver a cup of coffee to the study room, it will need to find a route to the kitchen from its current location, turn on the coffee maker, get a cup and fill it with coffee, pick up the cup, and finally find a route from the kitchen to the study room. Finding a sequence of actions for the robot to accomplish its “coffee delivery task” is a form of planning.

In general, a planning system is expected to analyze the situation in which an agent finds itself and then find a strategy for achieving the agent’s goals. In addition to planning for physical robots, planning techniques have been used in many practical artificial intelligence systems; for example, software protocol verification, spacecraft assembly at the European Space Agency (Aarup *et al.*, 1994), and observation planning at NASA for the Hubble space telescope (Johnson and Adorf, 1992). In its full generality, planning is a hard combinatorial search problem. Stated more formally, planning is PSPACE-complete (Bylander, 1991; Erol *et al.*, 1992). Therefore, a key challenge in fielding applications is to keep the computational cost of the planning task under control. Efficiency is achieved in current systems by carefully tailoring the planning algorithms for the application under consideration. However, such tailoring requires a significant effort for each application domain. Therefore, what is currently needed are more efficient *general* planning systems and declarative approaches to apply domain-specific knowledge in the systems. Furthermore, planning methods that automatically acquire knowledge or "learn" about their application domain during operation would be very useful.

In fact, we have recently seen exciting advances in general-purpose planning systems. First, there has been a burst of activity in the planning community with the introduction of a new generation of graph-based and constraint-based methods, such as GRAPHPLAN (Blum and Furst, 1995) and BLACKBOX (Kautz and Selman, 1992, 1996, 1999; Weld, 1999). These planners are domain independent and outperform more traditional planners on a range of benchmark problems. Then, at the AIPS 2000 planning competition, another generation of planning systems based on heuristic search techniques surpassed the performance of the constraint-based planners (Bacchus, 2001). Both HSP (Bonet and Geffnet, 1997) and FF (Hoffmann, 2000) are examples of such efficient heuristic search planning systems.

Current heuristic search and constraint-based planners can effectively synthesize plans consisting of several hundreds of actions, which is far beyond the capability of traditional planning systems. The work in this area has greatly benefited from a common set of benchmark problems. Many of these benchmark domains consist of essentially single-agent planning tasks. However, many real-world planning domains involve settings with multiple agents. We will provide a detailed comparison of the performance of the various recent planning techniques on multi-agent tasks. Our analysis shows that depending on the underlying problem domain, in a multi-agent setting, heuristic search and constraint-based planners often nicely complement each other. Our work also provides a new set of challenging benchmark problems, emphasizing multi-agent settings.

An interesting aspect of the recent search- and constraint-based planners is that they do not incorporate domain-specific knowledge but instead rely on efficient graph-based or propositional representations and advanced search techniques. Another approach has been proposed in the TLPLAN system, an example of a powerful planner incorporating declarative control specified in temporal logic formulas. We will demonstrate how these control rules can be parsed into BLACKBOX, leading to an overall speedup of up to one order of magnitude. We also provide a detailed comparison with TLPLAN, and show how the search strategies in TLPLAN lead to efficient plans in terms of the number of actions but with little or no parallelism. The BLACKBOX formalisms, on the other hand, do find highly parallel plans but are less effective in sequential domains. Our results enhance our understanding of the various tradeoffs in planning technology and extend earlier work on control knowledge in the SATPLAN framework by Ernst *et al.*, (1997) and Kautz and Selman (1998). The next natural question to ask is whether this kind of declarative control knowledge can be learned.

Machine learning is a rapidly developing subfield of artificial intelligence. Machine learning methods attempt to uncover general structure in a domain from a set of training cases. A good example of a learning system is the TD-gammon system (Tesauro, 1992) for playing backgammon. TD-gammon used a general learning strategy, called reinforcement learning, to train itself by playing a very large number of games against itself, continuously improving during play. After more than 300,000 training games, combined with some minor handcrafted heuristics, TD-gammon reached world-level play, comparable to the top three human players worldwide. No hand-coded backgammon program has even come close to the level of performance of TD-gammon. The TD-gammon system is therefore a clear example of the potential of machine learning techniques.

There has been earlier work on improving the performance of traditional planning systems using machine learning strategies. For example, the PRODIGY system (Minton, 1988a; Carbonell *et al.*, 1990) learns domain-specific control rules that guide the planner search algorithm by making it branch first on the most promising actions. Unfortunately, despite the long history of research in using machine learning to speed up state-space planning, the techniques that have been developed are not yet in widespread use in practical planning systems. One limiting factor is that traditional domain-independent planning systems scale so poorly that extensive learned control is required to raise their performance to an acceptable level. Therefore, work in this area has focused on learning large numbers of control rules that are specific to the details of the underlying planning algorithms, which can be extremely costly.

In this dissertation, we introduce a general methodology for automatically acquiring high-level, purely declarative constraints using machine learning techniques. In particular, we will present a new heuristic method for extracting training examples from plans generated by the BLACKBOX planner together with a constraint learning system based on an inductive learning programming approach. Our system can learn useful control knowledge in a variety of benchmark domains from small training problems. Only a small number of constraints are needed to reduce solution times by two orders of magnitude or more on larger problems. Training times are short. Moreover, given the declarative form of the constraints, they should be relatively easy to incorporate in other constraint-based planning systems.

1.1 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2, we provide an overview of recent state-of-the-art planning techniques: constraint-based planning and heuristic search planning. Then we give a detailed comparison of both approaches on multi-agent tasks in Chapter 3. In Chapter 4, we show the kinds of declarative control knowledge that can be used effectively in constraint-based planning.

In Chapter 5, we present our planning control knowledge learning system and evaluate its effectiveness on various benchmark domains. Finally, we conclude our work in Chapter 6.

Chapter 2

A Brief Overview of Planning

A planning problem consists of a description of an initial state, a goal state, and a set of actions in some formal language. Most planners we use today accept the STRIPS planning language to describe states and actions (Fikes and Nilsson, 1971). In STRIPS, an action is described using *preconditions* and *effects* that define the transition from states to states. An action can be performed only when the *preconditions* hold in the current state. There are two kinds of effects for an action. An *add* effect is represented by a predicate, and a *delete* effect is represented by the negation of a predicate. A state is defined by the set of facts that holds in the state. The execution of an action is defined by a state change, where each new state is obtained by adding or deleting facts from the previous state description. In other words, instantiated facts of add effects are added into the current state and facts of delete effects are removed from the current state. The planning task is to find a sequence of actions that leads from a given initial state to the goal state.

Initial: `(at pkg1 NYC), (at pkg2 SYR),`
`(at pln NYC),` ...
Goal: `(at pkg1 ITH), (at pkg2 ITH)`

Action:
```
(Load-Airplane
 :parameters (?obj ?airplane ?loc)
 :precond (and (at ?obj ?loc)
               (at (?airplane ?loc))...)
 :effect (and (in ?obj ?airplane)
              (not (at ?obj ?loc)))
...
```

Figure 2.1: A simple logistics problem $\mathcal{L}$.

To be more specific, supposing the prediction for an action $\mathcal{A}$ is denoted as $pre(\mathcal{A})$ and the add and delete effects are specified as $add(\mathcal{A})$ and $delete(\mathcal{A})$, respectively, an action $\mathcal{A}$ can only be performed in the current state $\mathcal{S}$ if $pre(\mathcal{A}) \subset \mathcal{S}$. The state $\mathcal{S}'$ after the execution of an action $\mathcal{A}$ from state $\mathcal{S}$ is defined as follows:

$$\mathcal{S}' = \mathcal{A}(\mathcal{S}) = (\mathcal{S} \setminus delete(\mathcal{A})) \cup add(\mathcal{A}).$$

Then the planning task is to find a sequence of actions $\mathcal{A}_1 \cdots \mathcal{A}_n$, which leads from a given initial state $\mathcal{I}$ to the goal state $\mathcal{G}$ such that

$$\mathcal{G} = \mathcal{A}_n(\cdots(\mathcal{A}_1(\mathcal{I})\cdots).$$

Figure 2.1 gives a simple example problem $\mathcal{L}$ in a logistics domain (Veloso, 1992). In this logistics domain, the task is to deliver a set of packages from different locations to their goal destinations using various vehicles. In the problem $\mathcal{L}$, we see that the initial and goal state are encoded by a set of ground facts. For example, predicate

1 Load-Airplane `(pkg1 pln NYC)`

2 Fly-Airplane `(plane NYC SYR)`

3 Load-Airplane `(pkg2 pln SYR)`

4 Fly-Airplane `(plane SYR ITH)`

5 Unload-Airplane `(pkg1 pln ITH)`

5 Unload-Airplane `(pkg2 pln ITH)`

Figure 2.2: A plan $\mathcal{P}$ for planning problem $\mathcal{L}$.

`(at pkg1 NYC)` in the initial state means that package `pkg1` is initially at New York City. The figure gives one example of an operator or action, called Load-Airplane. It takes an object, an airplane and a location as parameters. The precondition for the action to be applicable requires, among a number of other facts, that the plane be in the actual location. One of the effects of Load-Airplane is, for example, that the object is on board the plane after the action is executed.

Figure 2.2 shows one possible plan to achieve the goal state for problem $\mathcal{L}$. The figure gives a number with each action denoting the time slot in which the action takes place. Note that more than one action can take place at a given time. For example, two Unload-Airplane actions are performed at time step *5*.

The task of a planning system is to find a plan leading from the initial state to the goal state or to determine that no such plan exists. Planning systems generally search for the shortest possible plan, where length is either defined in terms of the total number of time steps or in terms of the total number of actions. As we noted before, planning is a PSPACE-complete problem and therefore a worst-case intractable computational problem (assuming P$\neq$NP; Bylander, 1991; Erol *et al.*, 1992; Backstrom, 1992).

Traditional planning systems generate plans by performing systematic search methods through either the state space or through the space of partial plans. For example, PRODIGY performs a state-space search by a backward-chaining planning algorithm (Carbonell *et al.*, 1990) and UCPOP employs a plan-based search method (Penberthy and Weld, 1992; Barrett *et al.*, 1993).

Recent progress in planning algorithms has shown that heuristic search and constraint-based planners outperform more traditional plan search approaches. Heuristic search planning systems, such as HSP and FF, employ forward-chaining search algorithms and a heuristic function is derived from the specification of the planning instance and used for guiding the search through the state space. In a constraint-based approach, the planning problem is formulated as a large constraint-satisfaction problem and the search is performed using constraint-satisfaction techniques. The resulting search is a hybrid of a state-based and a plan-based search approach. Such a hybrid search strategy can be quite efficient. For example, the constraint-based planner, BLACKBOX, is able to find a plan with 74 actions within 14 time steps for a logistics planning problem in less than eight seconds on a 450 MHz Sparc Ultra machine. The state space explored by BLACKBOX for this planning instance contains approximately 10^{16} distinct states. No traditional planners have been able to solve logistics planning problems of this size, even when given days of computational time.

In the following sections, we will provide a brief overview of recent progress on constraint-based and heuristic search planning systems.

2.1 Constraint-Based Planning

Constraint-based planning systems formulate the planning task as the solving of a large constraint-satisfaction problem (CSP). GRAPHPLAN and its descendents encode a CSP in a data structure called a plan graph, while SATPLAN and its descendent BLACKBOX explicitly convert planning problems into Boolean satisfiability.

2.1.1 Graphplan

The GRAPHPLAN planner (Blum and Furst, 1995) is based on a compact structure called a planning graph. A planning graph encodes the planning problem such that many useful constraints inherent in the problem can be used by various search algorithms to reduce the search time. In addition, planning graphs have polynomial size and can be constructed in polynomial time.

A planning graph is a directed graph with two types of nodes: propositional nodes ("facts") and action nodes, arranged into levels and indexed by time. The first level of a planning graph is a propositional level and consists of propositional nodes from the initial state. The next level is an action level and consists of possible actions at time 1. Then a propositional level containing possibly true propositions at time 2 and an action level containing possible actions at time 2 follows. That is, the levels in a planning graph alternate between propositional levels and action levels.

The planning graph is constructed from the initial state. The initial facts are placed in the first propositional level of the graph. To extend a graph at level i, each possible instantiation for every action is inserted into the action level i, provided its preconditions hold at time i. A NO-OP action for every fact in the previous propositional level is inserted as well. These special NO-OP actions are needed to deal with those facts that do not change from time slot to time slot. In other words, the use of these NO-OP actions is an approach to handle the *frame problem* (McCarthy and Hayes, 1969). For each action node, the facts in the *effects* list are inserted into the next layer of propositional nodes at level $i + 1$. An edge is also established between an action node and each of its effect nodes. Each edge is labeled as either an add edge or a delete edge. An *add* edge indicates the propositional node is an add effect of the action and a *delete* edge indicates the propositional node is a delete effect of the action. Moreover, as shown in Blum and Furst (1995), the time taken to create the planning graph and the size of the planning graph are both polynomial in the length of the problem's description and the number of time steps.

In addition to the planning graph itself, GRAPHPLAN also derives a set of mutual exclusion constraints among nodes. Two propositional nodes are *mutually exclusive* if no valid plan could make both true at the same time. For example, a package cannot be loaded on two different airplanes during the same time slot. Two action nodes are *mutually exclusive* if no valid plan could contain both. For example, an airplane cannot fly and be loaded during the same time slot. These mutual exclusion constraints can be derived by propagating them through the planning graph using a few simple rules and can be of enormous help in reduing the search time.

To search for a plan, GRAPHPLAN alternates between two stages: *planning graph extension* and *plan extraction*. The planning graph is a compact representation of all valid action sequences up to a fixed number of time slots, starting from the initial state. In the extraction phase, this graph is searched for a sequence that leads to the goal state. At stage i, GRAPHPLAN extends the planning graph from stage $i - 1$ and then starts to search for a plan on the extended planning graph when all of propositional facts in the goal state appear in the final layer of propositional level. In other words, each planning graph extension stage extends a new time slot into the planning graph and a plan extraction phase follows to find a plan on the extended planning graph. GRAPHPLAN's systematic search either finds a plan or shows that no valid plan exists in the current planning graph. The planning graph extension and plan extraction continues either until a plan is found or until GRAPHPLAN proves there exists no plan of any length that leads from the initial state to the goal state. An interesting aspect of the GRAPHPLAN system is that the system can be shown to be complete. That is, after a *finite* number of planning graph extensions, the

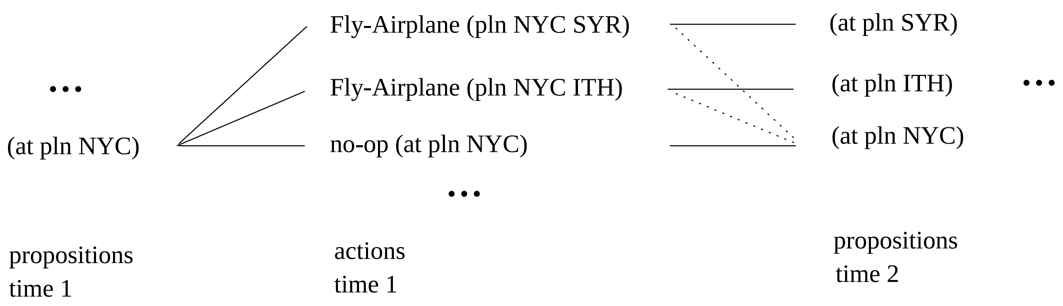

Figure 2.3: Partial planning graph for logistics problem $\mathcal{L}$.

system will either have found a valid plan or it will have shown that the goal is unreachable, no matter how many time steps are allowed.

Although the full details of the planning graph approach are rather involved, Figure 2.3 should give the reader at least some idea of how a planning graph is constructed. We see that `(at pln NYC)` is a precondition for the three actions given at time 1. The effects for each action at time 1 are given in the propositional layer at time 2. Finally, the dotted lines show deleted facts. For example, Fly-Airplane `(pln NYC SYR)` deletes the fact `(at pln NYC)` in the propositional layer at time 2. Not explicitly represented in the figure, but part in the actual planning graph, are the exclusionary relations. We have established that actions Fly-Airplane `(pln NYC SYR)` and Fly-Airplane `(pln NYC ITH)` at time 1 are mutually exclusive. Similarly, propositions `(at pln SYR)` and `(at pln ITH)` at time 2 are also mutually exclusive.

After each planning graph extension phase, GRAPHPLAN enters a plan extraction phase, in which the graph is searched for a plan that reaches all goal conditions. If no such plan is found, the planning graph is extended.

GRAPHPLAN has been shown to outperform traditional planners, such as PRODIGY and UCPOP, on a variety of interesting planning problems and its algorithm has been adopted in several recent planners.

2.1.2 Blackbox

Given a propositional encoding of the planning task, the BLACKBOX planning system uses fast Boolean satisfiability procedures to search for plans (Kautz and Selman, 1992, 1996, 1999). The general planning "as satisfiability framework" is shown in Figure 2.4. At first, the system compiles the planning problem into a propositional formula in conjunctive normal form (CNF), which represents all possible action sequences up to a fixed number of time steps. The formula encodes information in a manner similar to that of the planning graph discussed above. The formula also contains a set of logical clauses that capture the initial state and the goal state. This encoding is such that there is a one-to-one correspondence between valid plans up to the given time bound and satisfying assignments. State-of-the-art Boolean satisfiability procedures ("SAT Solvers"), such as WALKSAT (Selman *et al.*, 1996) or CHAFF (Moskewicz *et al.*, 2001), are used to search for satisfying assignments. If a solution can be found, the assignment can be translated back into a valid plan; otherwise, the time bound will be relaxed and another formula will be generated.

The original implementation of the planning as a satisfiability approach, called SATPLAN, uses a state-based encoding designed by

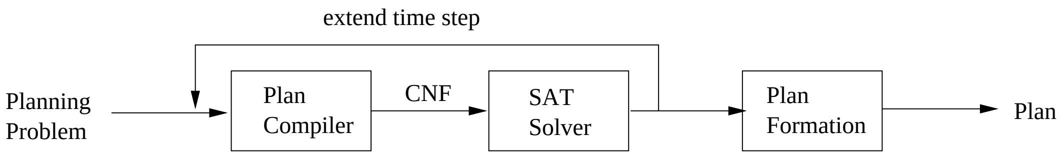

Figure 2.4: Planning as a satisfiability framework.

hand (Kautz and Selman, 1992). The MEDIC (Ernst *et al.*, 1997) planner automatically translates a STRIPS-style planning problem into a range of different encodings according to the choice of action representation and the choice of classical or explanatory frame axioms.

The BLACKBOX planner (Kautz and Selman, 1999) is the latest version of SATPLAN. It uses GRAPHPLAN as a front end to construct a planning graph. The system then compiles the planning graph into a CNF formula. The formula is simplified using a polynomial time Boolean simplification procedure. Finally, the simplified formula is fed to a range of SAT solvers.

The GRAPHPLAN-based compilation procedure in BLACKBOX is straightforward. If a proposition node p has add actions $a_1, \cdots, a_n$, it forms a logical clause

$$(\neg p \bigvee_{i=1}^{n} a_i)$$

(i.e., $p \rightarrow a_1 \vee ... \vee a_n$). This clause represents the fact that when p is true, it has to have been added by at least one of the actions $a_1, \cdots, a_n$. If an action node a has preconditions $p_1, ..., p_n$, the following clauses are added into the CNF formula:

$$\bigwedge_{i=1}^{n} (\neg a \vee p_i)$$

If two nodes p and q are mutually exclusive (note that p and q can be action or fact nodes), the following clause is added:

$$(\neg p \vee \neg q)$$

Finally, facts in the initial state and goal state are added to the CNF formula as unit clauses.

The BLACKBOX planner is competitive with the GRAPHPLAN planner, and both outperform traditional planners. One advantage of the BLACKBOX planner is that because of the logical intermediate representation it is straightforward to add additional domain-specific knowledge to the planner. Basically, one can add additional clauses to the formula before giving it to the SAT solver. Those new clauses lead to additional pruning and can dramatically speed up the SAT solver. We will return to this issue in Chapter 4.

2.1.3 Other Constraint-Based Planning Systems

Since the introduction of GRAPHPLAN and SATPLAN, their algorithms have been applied in many other planning systems, built to extend their capability or improve their performance. For example, IPP (Koehler *et al.*, 1997) is an extended version of GRAPHPLAN that can deal with a subset of more express ADL planning language (Pednault, 1989). In addition, it also introduces a goal orderings technique to derive a total ordering for subsets of goals by performing a static and heuristic analysis of the planning problem at hand. The goal orderings method has been shown to significantly improve the performance of the IPP planning system in most cases (Koehler and Hoffmann, 2000). The same goal orderings technique is exploited in the heuristic search planner FF as well.

The STAN (Fox and Long, 1999) planning system improves the graph construction in two ways. First, the planning graph is represented as a single pair of layers called a *spike*, which is built upon bit vectors and logical operations to accelerate the graph construction. Second, it uses a *wave front* technique to avoid the explicit construction of the graph beyond the fixed point. Both techniques reduce the graph construction time and the memory usage of a planning graph.

GPG (Gerevini and Schubert, 1998) is a planning system that uses combined stochastic local search and tabu list (Glover and Laguna, 1993) techniques to improve GRAPHPLAN's plan search time.

CPlan (Beek and Chen, 1999) is a planning approach that models planning problems as constraint-satisfaction problems (CSPs). CSPs in CPlan are solved by a backtracking algorithm that performs generalized arc consistency propagation and conflict-directed backjumping (Prosser, 1993).

2.2 Heuristic Search Planning

Recent planning systems based on the idea of heuristic search have been shown to be efficient on sequential domains. At the AIPS 2000 planning competition, heuristic planners, FF and HSP, eclipsed the performance of the constraint-based planners (Bacchus and Nau, 2001).

Both HSP and FF are forward-chaining planners. They use either a breadth-first, best-first, or greedy search algorithm. The heuristic functions in HSP and FF rely on a *relaxed* plan graph, in which *delete* effects of actions are not considered. In HSP, the distance from the current state s_c to the goal state s_g is approximated by using the sum of the weights of the propositions that need to hold in s_g. The weight of a proposition p is defined as follows:

$$weight(p) = \begin{cases} 0 & \text{, if } p \in s_g \\ min_{\text{action } a \text{ adds } p}[1 + weight(precond(a))] & \text{, otherwise} \end{cases}$$

In other words, the weight of a proposition p is 0 if it is in s_g. Otherwise, it is one more than the minimal weight of the preconditions of actions that add p. In FF, the distance from the current state to the goal state is approximated by the plan length for achieving the goal in the relaxed plan graph.

2.3 Summary

We have presented a brief overview of current state-of-the-art planning approaches: constraint-based planning and heuristic search base planning. Constraint-based planning systems formulate planning tasks as the solving of constraint-satisfaction problems. Heuristics search planners perform searches based on the heuristic functions computed from relaxed plan graphs. Both planning methods have been shown to outperform most of traditional planners. In order to obtain a better understanding of their relative strengths and weaknesses, in the next chapter we will present a detailed comparison of the methods through both experimental results and theoretical analysis in a multi-agent setting.

Chapter 3

Single-Agent vs. Multi-Agent Planning

We have recently seen exciting advances in practical planning systems. First, it was discovered that constraint-based planning techniques, such as those used in GRAPHPLAN (Blum and Furst, 1995) and BLACKBOX (Kautz and Selman, 1999), can significantly extend the range of practically feasible planning tasks compared to the more traditional planning approaches (Weld, 1999). The effectiveness of constraint-based techniques was apparent at the AIPS 1998 planning competition (Long *et al.*, 2000). Perhaps somewhat surprisingly, at the AIPS 2000 competition, yet another generation of planning systems — this time based on heuristic search techniques — eclipsed the performance of the constraint-based planners (Bacchus and Nau, 2001).

A common set of benchmark domains, as used for example in the AIPS planning competitions, has been an important driving force behind the rapid improvements we have seen in recent years. These benchmarks mostly involve single-agent planning tasks. However, many real-world planning problems involve multiple agents. Much of the research in the area of multi-agent planning considers distributed solution strategies, where each agent plans its actions locally either in collaboration or in competition with the other agents (desJardins *et al.*, 1999). An interesting alternative to explore is whether a centralized domain-independent planner can uncover global plans involving multiple agents working together. If interesting global multi-agent plans could be computed effectively, it would open up a wide range of applications in which groups of cooperative autonomous agents would follow a global plan or strategy to reach a set of common objectives. Such global plans would also provide a useful framework for comparison with plans that are computed in a distributed fashion.

In this section, we will introduce a set of multi-agent planning problems and analyze the performance of the most recent heuristic and constraint-based planners on these problem instances. In order to obtain interesting multi-agent plans, we have to add special additional requirements onto our final plans. Intuitively speaking, we have to require that each agent actively participates in the final plan, or, more precisely, that each agent meets some minimal level of participation. The reason for this is straightforward: without the participation requirement, a single-agent solution often also provides as solution of the multi-agent planning task (*i.e.*, the actions to achieve the overall goals are executed by a single agent, while the other agents remain idle). We will show how it is possible to encode the participa-

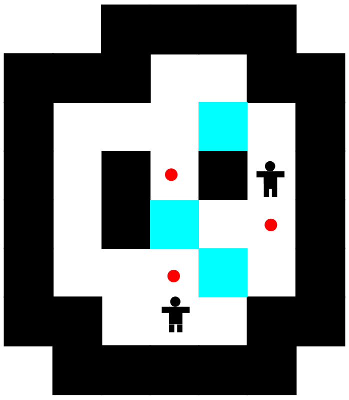

Figure 3.1: A sokoban example game.

tion requirements as part of the planning problem instances. A key advantage of such an approach is that it does not require us to modify the existing planners.

Our experiments show that current planners can find interesting global multi-agent plans. Moreover, in the multi-agent setting, we find that heuristic search and constraint-based planners have highly complementary strengths.

3.1 Performance on Single-Agent Problems

The testbed used in this section includes problems obtained from the miconic domain, the rocket domain, the grid domain, and the sokoban domain. The grid domain was first introduced in the AIPS 1998 planning competition (Long *et al.*, 2000). The miconic domain is based on an elevator control problem, and was part of the AIPS 2000 benchmarks. The rocket domain is from Veloso (1992). Finally, our most challenging domain is based on the Sokoban game invented in Japan in the early 1980s. In this game, a warehouse keeper, or *sokoban* in Japanese, pushes boxes to their correct destinations in a crowded warehouse. The boxes are too heavy to lift. Figure 3.1 shows an example game. There are three boxes in the game; the small circle dots specify the goal locations for the boxes. The boxes can be moved around via “pushes” by either one of the two sokoban agents in the domain. The game looks easy but is in fact remarkably difficult. A key challenge is to avoid getting any of the boxes stuck in a corner.

First, we provide a comparison between two heuristic planners, HSP (Bonet and Geffner, 1997) and FF (Hoffmann, 2000), and one constraint-based planner BLACKBOX(Kautz and Selman, 1996, 1999). We generated problem instances with a single agent each and ran all our experiments on a 400 Mhz SUN Ultra-80 machine. Note that in each domain there is a natural concept of “agent” (or key player). In the miconic domain, an elevator is an agent; in the rocket domain, each rocket is an agent; in the grid domain, a robot is an agent; and in the sokoban domain, a sokoban is an agent. To track the activities of each agent, we identify a so-called *feature action* for each agent in each domain. A feature action is the most characteristic action executed by an agent. For example, for the sokoban agent, it is the “Push” action. The feature actions for other domains are as follows: the “Up” action in the miconic domain, the “Fly” action in the rocket domain, and both “Pickup” and “Pickup-and-Loose” actions in the grid domain. The main function of the feature actions is to provide us with a measure of activity for each agent in terms of solving the overall goal.

Table 3.1: Comparison between BLACKBOX, HSP, and FF on single-agent problems.

problem	BLACKBOX			HSP		
	time	#action	#feature action	time	#action	#feature action
grid-1	12610	23	4	0.83	29	5
grid-2	–	–	–	0.49	28	4
grid-3	–	–	–	2.5	18	2
miconic-1	73.6	23	4	0.36	27	5
miconic-2	15.5	26	6	0.22	33	5
miconic-3	22.1	32	3	0.30	45	5
rocket-1	0.54	19	3	0.30	25	9
rocket-2	0.48	24	4	0.13	35	15
rocket-3	6.88	21	5	1.01	27	11
sokoban-1	2.47	17	7	0.78	21	7
sokoban-2	5.11	21	9	0.96	25	9
sokoban-3	5.25	23	9	1.13	27	9

problem	BLACKBOX			FF		
	time	#action	#feature action	time	#action	#feature action
grid-1	12610	23	4	0.06	25	4
grid-2	–	–	–	0.04	29	4
grid-3	–	–	–	0.14	22	3
miconic-1	73.6	23	4	0.03	23	6
miconic-2	15.5	26	6	0.02	26	4
miconic-3	22.1	32	3	0.04	32	4
rocket-1	0.54	19	3	0.03	19	3
rocket-2	0.48	24	4	0.02	24	4
rocket-3	6.88	21	5	0.05	21	5
sokoban-1	2.47	17	7	0.06	21	7
sokoban-2	5.11	21	9	0.08	25	9
sokoban-3	5.25	23	9	0.10	27	9

This function will become clearer in our multi-agent setting. Note, however, that the choice for feature actions is somewhat subjective and in most cases we could have chosen another action, which would also have provided us with a good measure for the activity level of an agent.

Table 3.1 gives the solution times for our planners on a range of single-agent test problems. It is clear that both HSP and FF are very efficient on all problems. BLACKBOX is slower and cannot find solutions on certain problems in the grid domain.

3.2 Multi-Agent Problems

Many practical planning applications, such as job or airline scheduling, involve multiple agents. In addition to the preference of shorter plans for multi-agent problems, another desirable property for such plans is that each agent actively participates in solving the problem. Plans with such a property usually have a shorter total time span and avoid the cost associated with idling agents.

In order to obtain a better understanding of how heuristic search and constraint-based planners perform on multi-agent environments, we added more agents into the testbed problem instances. Table 3.2 shows the run time on multi-agent problems. It also shows the number of feature actions used by each agent in the plan. More precisely, feature actions (3, 2, 4) for problem sokoban-1 means that the first sokoban performed 3 pushes, the second 2 pushes, and the third 4

Table 3.2: Comparison between BLACKBOX, HSP, and FF on multi-agent problems.

problem	#agent	BLACKBOX		
		time	#action	#feature action
grid-1	3	13.2	25	(1, 1, 2)
grid-2	3	13.9	31	(1, 1, 2)
grid-3	3	149	43	(1, 1, 1)
miconic-1	3	9.24	27	(3, 3, 2)
miconic-2	3	9.45	30	(3, 3, 3)
miconic-3	4	32.6	37	(2, 2, 2, 2)
rocket-1	2	0.39	20	(2, 2)
rocket-2	2	0.72	26	(3, 3)
rocket-3	2	2.53	22	(3, 3)
sokoban-1	3	1.42	19	(3, 2, 4)
sokoban-2	3	1.98	17	(3, 2, 2)
sokoban-3	3	3.2	16	(5, 1, 3)

problem	#agent	HSP		
		time	#action	#feature action
grid-1	3	6.86	25	(0, 3, 1)
grid-2	3	3.41	28	(0, 2, 2)
grid-3	3	7.15	35	(0, 1, 2)
miconic-1	3	2.14	29	(2, 2, 4)
miconic-2	3	0.82	33	(3, 4, 1)
miconic-3	4	1.69	40	(2, 1, 2, 2)
rocket-1	2	0.30	25	(4, 5)
rocket-2	2	0.16	36	(8, 8)
rocket-3	2	1.53	26	(3, 7)
sokoban-1	3	38.4	17	(5, 1, 3)
sokoban-2	3	21.6	16	(3, 1, 5)
sokoban-3	3	37.4	20	(7, 2, 2)

Table 3.2 (Continued).

problem	#agent	BLACKBOX		
		time	#action	#feature action
grid-1	3	13.2	25	(1, 1, 2)
grid-2	3	13.9	31	(1, 1, 2)
grid-3	3	149	43	(1, 1, 1)
miconic-1	3	9.24	27	(3, 3, 2)
miconic-2	3	9.45	30	(3, 3, 3)
miconic-3	4	32.6	37	(2, 2, 2, 2)
rocket-1	2	0.39	20	(2, 2)
rocket-2	2	0.72	26	(3, 3)
rocket-3	2	2.53	22	(3, 3)
sokoban-1	3	1.42	19	(3, 2, 4)
sokoban-2	3	1.98	17	(3, 2, 2)
sokoban-3	3	3.2	16	(5, 1, 3)
problem	#agent	FF		
		time	#action	#feature action
grid-1	3	0.17	27	(4, 0, 0)
grid-2	3	0.11	29	(3, 1, 0)
grid-3	3	1.44	48	(3, 2, 0)
miconic-1	3	0.05	23	(6, 0, 0)
miconic-2	3	0.04	26	(4, 0, 0)
miconic-3	4	0.06	32	(4, 0, 0, 0)
rocket-1	2	0.04	20	(4, 0)
rocket-2	2	0.02	26	(6, 0)
rocket-3	2	0.06	22	(7, 0)
sokoban-1	3	8.76	19	(5, 1, 3)
sokoban-2	3	8.76	19	(5, 1,3)
sokoban-3	3	9.99	18	(7, 1, 1)

pushes. The reason for tracking the number of feature actions performed by each agent is that we can use it to approximately measure the relative activity level of each agent in the plan. In other words, we can use these numbers to identify how actively an agent participates in the solution plan. A higher number of feature actions executed by an agent usually indicates that the agent has a higher activity level in the plan than do agents executing fewer feature actions.

The first observation from Table 3.2 is that BLACKBOX runs more quicly on most of the multi-agent problems than on the single-agent problems. In particular, all the multi-agent grid problems are solved, while only one of the single-agent grid problems can be solved by BLACKBOX. On the other hand, HSP and FF run slightly more slowly on most multi-agent problems in comparison to their run times on single-agent problems. This phenomenon is most obvious on problems in the grid and sokoban domains.

We also observed that BLACKBOX tends to find plans where each agent participates in the solution with a similar activity level. This is not a surprising result since BLACKBOX will try to find plans with shorter time spans and allow many agents to participate at the same time. Therefore, each agent demonstrates similar activity in the plans found by BLACKBOX. HSP and FF found solutions with a greater variance in the activity level – on some problems some of the agents remained idle or with relatively low participation activity throughout the plan. Based on the above observations, the differences shown by the constraint-based planners and heuristic search planners between single-agent and multi-agent problems raise some interesting questions. First, are there ways to *force* planners, especially heuristic search planners, to find plans that meet participation requirements? Second, how will the planners perform in order to find these plans?

3.3 Restriction on the Feature Action

It is not clear how one can modify the planners to find plans for which each agent meets the participation requirements. Therefore, instead of modifying planners themselves, we modify the problem instances so that any plans for the problems ensure minimum participation of each agent.

We modify the feature action in the domain definition so that it will consume one resource unit when it is applied. Then, by restricting the available resource units an agent has in order to execute its feature actions, we can indirectly push the planners to find plans in which each agent meets the participation requirements, or at least participates in the plan at a certain level. This participation can be achieved because by limiting the resource an agent has for its feature actions, problems can only be solved by multiple agents rather than by a single agent. On the other hand, as shown in the previous section, our testbed problems can be solved by a single agent if no restrictions are imposed.

The general outline of a modified feature action is shown in Figure 3.2. Two predicates are added to the original domains. Predicate

```
(:action Feature-Action
 :parameters (?agent ?r ...)
 :precondition (and (resource ?r)
                    (ar ?agent ?r) ...)
 :effect (and (not (ar ?agent ?r)) ...)
)

; problem instance
(:init (resource r1)
       (resource r2)
       ...
       (resource rm)
       (ar agent1 r1) (ar agent1 r2)
       (ar agent2 r1) ... (ar agent2 rm)
       ...
```

Figure 3.2: General outline of a modified feature action and problem instance.

`(resource ?r)` is used to represent resource units, and predicate `(ar ?agent ?r)` specifies which resource `?r` can be used by agent `?agent`. Therefore, to be performed, each feature action will need at least one available resource unit, and it will consume one unit after the execution. We then limit the number of resource units available for each individual agent in the problem. The available resource units for each agent are added into the initial state. For example, if there are two sokobans in a problem and sokoban `s1` and `s2` is allowed two push actions, predicates `(resource r1)`, `(resource r2)`, `(ar s1 r1)`, `(ar s1 r2)`, `(ar s2 r1)`, `(ar s2 r2)` are added into the problem's initial state.

Table 3.3 shows the run time for all planners on the multi-agent problems with restrictions on feature actions. The number of re-

Table 3.3: Comparison between BLACKBOX, HSP, and FF on multi-agent problems with restrictions on feature actions.

problem	#restricted feature action	BLACKBOX time	BLACKBOX #action	BLACKBOX #feature action
grid-1	(2, 2, 2)	33.1	25	(2, 1, 1)
grid-2	(2, 2, 2)	1095	34	(2, 2, 2)
grid-3	(1, 1, 1)	36.4	42	(1, 1, 1)
miconic-1	(3, 3, 3)	31.4	24	(3, 2, 2)
miconic-2	(3, 3, 3)	21.9	30	(3, 2, 3)
miconic-3	(2, 2, 2, 2)	69	39	(2, 2, 1, 2)
rocket-1	(2, 2)	0.44	20	(2, 2)
rocket-2	(3, 3)	1.21	26	(3, 3)
rocket-3	(3, 3)	4.02	22	(3, 3)
sokoban-1	(4, 4, 4)	4.25	19	(3, 2, 4)
sokoban-2	(3, 3, 3)	8.22	20	(3, 3, 3)
sokoban-3	(5, 5, 5)	35.2	16	(5, 1, 3)

problem	#restricted feature action	HSP time	HSP #action	HSP #feature action
grid-1	(2, 2, 2)	19.6	25	(1, 2, 1)
grid-2	(2, 2, 2)	14.4	33	(2, 2, 2)
grid-3	(1, 1, 1)	18.6	39	(1, 1, 1)
miconic-1	(3, 3, 3)	4.6	29	(2, 3, 3)
miconic-2	(3, 3, 3)	1.98	33	(3, 3, 3)
miconic-3	(2, 2, 2, 2)	2.59	40	(2, 2, 1, 2)
rocket-1	(2, 2)	344	26	(2, 2)
rocket-2	(3, 3)	280	28	(3, 3)
rocket-3	(3, 3)	–	–	–
sokoban-1	(4, 4, 4)	–	–	–
sokoban-2	(3, 3, 3)	975	16	(2, 2, 3)
sokoban-3	(5, 5, 5)	–	–	–

Table 3.3 (Continued).

problem	#restricted feature action	BLACKBOX		
		time	#action	#feature action
grid-1	(2, 2, 2)	33.1	25	(2, 1, 1)
grid-2	(2, 2, 2)	1095	34	(2, 2, 2)
grid-3	(1, 1, 1)	36.4	42	(1, 1, 1)
miconic-1	(3, 3, 3)	31.4	24	(3, 2, 2)
miconic-2	(3, 3, 3)	21.9	30	(3, 2, 3)
miconic-3	(2, 2, 2, 2)	69	39	(2, 2, 1, 2)
rocket-1	(2, 2)	0.44	20	(2, 2)
rocket-2	(3, 3)	1.21	26	(3, 3)
rocket-3	(3, 3)	4.02	22	(3, 3)
sokoban-1	(4, 4, 4)	4.25	19	(3, 2, 4)
sokoban-2	(3, 3, 3)	8.22	20	(3, 3, 3)
sokoban-3	(5, 5, 5)	35.2	16	(5, 1, 3)

problem	#restricted feature action	FF		
		time	#action	#feature action
grid-1	(2, 2, 2)	0.56	28	(2, 2, 0)
grid-2	(2, 2, 2)	0.77	36	(3, 1, 0)
grid-3	(1, 1, 1)	22.4	39	(1, 0, 1)
miconic-1	(3, 3, 3)	0.10	25	(3, 3, 0)
miconic-2	(3, 3, 3)	0.07	28	(3, 2, 0)
miconic-3	(2, 2, 2, 2)	0.09	32	(2, 0, 0, 0)
rocket-1	(2, 2)	–	–	–
rocket-2	(3, 3)	–	–	–
rocket-3	(3, 3)	–	–	–
sokoban-1	(4, 4, 4)	–	–	–
sokoban-2	(3, 3, 3)	–	–	–
sokoban-3	(5, 5, 5)	–	–	–

stricted feature actions shows the maximum number of times an agent can execute its feature actions. To give an example, grid-1 has restricted feature actions (2, 2, 2), meaning each agent can perform Pickup actions at most twice. It is apparent that BLACKBOX generates results comparable to those in Table 3.2. The increased run time is probably due to the extra overhead incurred with the action modification. Heuristic search planners HSP and FF also find plans very efficiently on certain problems, particularly problems in the grid and miconic domains. In other words, these results do show that our approach "pushes" heuristic planners to find plans in which the participation requirements are met without the need to modify the planners, while still maintaining their efficiency, at least on certain domains.

3.4 Complexity Analysis

Nevertheless, as Table 3.3 illustrates, heuristic search planners are not as efficient as BLACKBOX on multi-agent problems in rocket and sokoban domains. This reduction may be due to the more intricate combinatorial nature of these domains.

To reach this conclusion, we first experimented with problem rocket-2 by varying the number of available resource units on each rocket. Figure 3.3 shows the result. For the same problem, in spite of the different number of flights available, the run time for BLACKBOX remained the same despite the tightening of resources. In contrast,

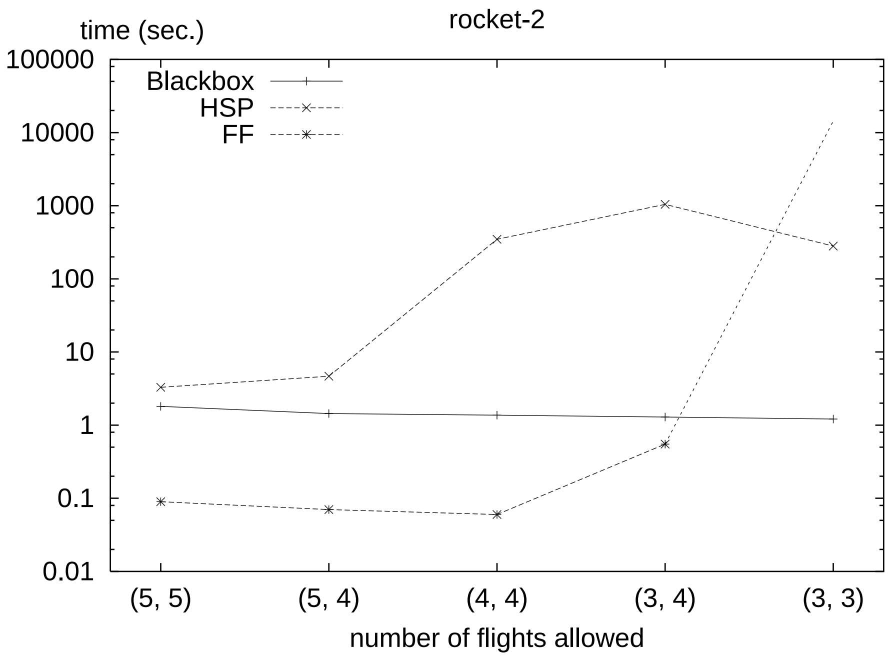

Figure 3.3: Comparisons for problem rocket-2 using different numbers of flights.

we see the run time go up significantly (note the log scale) for heuristic search planners when fewer resources are available. FF, especially, is unable to solve problem rocket-2 within the allowed time frame with 3 flights available for both rockets. In fact, on rocket planning instances with restrictions on feature actions, BLACKBOX can now find plans more quickly than other heuristic planners. We next conducted a more complete set of experiments on the rocket domain, and the results are shown in Table 3.4.

It is apparent that the FF planner either solves the problems in a very short time or cannot solve them at all. On the other hand, HSP seems more robust on these problems. Moreover, we also observe that FF tends to be unbalanced in its flight usage, as shown for rocket-1

Table 3.4: Comparison between BLACKBOX, HSP, and FF on multi-agent rocket problems with restrictions on feature actions.

problem	#r.f.a.	BLACKBOX			HSP		
		time	len.	#f.a.	time	len.	#f.a.
rocket-1	(3, 3)	0.48	20	(2, 2)	2.03	22	(3, 3)
	(3, 2)	0.45	20	(2, 2)	266	21	(3, 2)
	(2, 2)	0.44	20	(2, 2)	344	26	(2, 2)
rocket-2	(4, 3)	1.26	26	(3, 3)	2221	33	(4, 3)
	(3, 4)	1.29	26	(3, 3)	1045	27	(3, 4)
	(3, 3)	1.21	26	(3, 3)	280	28	(3, 3)
rocket-3	(5, 5)	4.61	22	(3, 3)	2.88	26	(5, 5)
	(5, 4)	4.34	22	(3, 3)	11189	27	(5, 4)
	(5, 3)	4.17	22	(3, 3)	–	–	–
	(4, 4)	4.25	22	(3, 3)	–	–	–
	(4, 3)	3.87	22	(3, 3)	–	–	–
	(3, 3)	4.02	22	(3, 3)	–	–	–
rocket-4	(4, 5)	1.84	26	(3, 3)	13.2	29	(4, 5)
	(3, 4)	1.66	26	(3, 3)	47.2	27	(3, 4)
	(3, 3)	1.61	26	(3, 3)	45.1	28	(3, 3)

problem	#r.f.a.	BLACKBOX			FF		
		time	len.	#f.a.	time	len.	#f.a.
rocket-1	(3, 3)	0.48	20	(2, 2)	0.17	24	(3, 1)
	(3, 2)	0.45	20	(2, 2)	0.12	24	(3, 1)
	(2, 2)	0.44	20	(2, 2)	–	–	–
rocket-2	(4, 3)	1.26	26	(3, 3)	0.18	26	(4, 2)
	(3, 4)	1.29	26	(3, 3)	0.55	29	(3, 4)
rocket-2	(3, 3)	1.21	26	(3, 3)	–	–	–
rocket-3	(5, 5)	4.61	22	(3, 3)	0.61	28	(5, 5)
	(5, 4)	4.34	22	(3, 3)	–	–	–
	(5, 3)	4.17	22	(3, 3)	–	–	–
	(4, 4)	4.25	22	(3, 3)	–	–	–
	(4, 3)	3.87	22	(3, 3)	–	–	–
	(3, 3)	4.02	22	(3, 3)	–	–	–
rocket-4	(4, 5)	1.84	26	(3, 3)	0.05	25	(4, 1)
	(3, 4)	1.66	26	(3, 3)	0.04	25	(4, 1)
	(3, 3)	1.61	26	(3, 3)	–	–	–

and rocket-4. Therefore, we suspect that FF may be weaker than the other planners for problems that require balanced participation of all agents. We believe the above results support our conjecture that heuristic search planners are weaker for problems that require more balanced participation of all agents.

From the theoretical point of view, the sokoban domain has been shown to be PSPACE-complete (Culberson, 1997). Therefore, it remains PSPACE-complete when the feature actions are restricted. However, note that with restrictions on the number of flights, the decision of whether there exist plans for problems in the rocket domain becomes NP-complete. The proof is shown in the following theorem and corollary:

Theorem 1. *Flights restricted rocket problem with single rocket (FR-ROCKET$_1$) is NP-complete.*

Proof. Clearly FR-ROCKET$_1$ is in NP. We reduce from the feedback vertex set problem (Gary and Johnson, 1979) to show that problems in FR-ROCKET$_1$ are NP-hard. We show that a digraph $G(V, E)$ has a feedback vertex set V', where $|V'| \leq K$, iff there exists a plan for a planning problem with restricted flight number $\leq |V| + K$ in FR-ROCKET$_1$.

For each digraph $G(V, E)$, we construct a planning problem with $|V| + |E|$ passengers and $|V| + 1$ locations. A rocket is initially located at v_0. For each other locations there is a passenger at v_0 that needs to be delivered. For each $(u, v) \in E$, there is a passenger at u that needs to be delivered to v.

(⇒) For each feedback vertex set $V' \subseteq V$ and $|V'| \leq K$, the planning problem can be solved by loading all $|V|$ passengers at v_0 and flying the rocket from v_0 to any vertex in V' first, and then to the vertices in the subgraph $V \backslash V'$ in an order that is consistent with the arcs in $V \backslash V'$, and finally flying the rocket back to a vertex in V' since V' is a feedback vertex set. The above process can be repeated until all vertices in V have been visited. Passengers initially at v_0 are dropped when it is at the goal and passengers corresponds to (u, v) are picked at u and dropped at v. This requires $|V| + |V'|$ flights and the plan can be solved with bounded flight number of $|V| + K$ since $|V'| \leq K$.

(⇐) If there exists a plan for the planning problem , those vertices that are visited more than once form a feedback vertex set V'. Since every vertex needs to be visited at least once and the number of flights is less than $|V| + K$, $|V'| \leq K$. □

Corollary 1. *Flights restricted rocket problem with multi-rocket (FR-ROCKET) is NP-complete.*

Proof. This can be shown by simply reducing FR-ROCKETS$_1$ to FR-ROCKET since any problem in FR-ROCKETS$_1$ is a problem in FR-ROCKET. □

On the other hand, we do find polynomial time algorithms to solve feature action restricted multi-agent problems in grid and miconic domains. Multi-agent problems in the grid domain include an $N \times N$

grid and several robots that can move between connected grid cells. Each cell in the grid can be either open or locked. Only open adjacent cells can be reached by a robot. Each cell may also contain keys that can be used to unlock a cell if the locked cell and the key have the same shape. Robots can pick up and put down a key. The task for problems in the grid domain is to use robots to deliver keys to their goal cells. For the grid domain, the number of required Pickups can be decided by the number of keys that need to be moved and by the number of keys required to unlock cells. In the worst case, supposing that M keys need to be moved and every cell needs to be unlocked, the maximum number of Pickups is $O(N^2+M)$. If the number of available Pickup actions for robots is greater than the minimum required number, then the problem becomes solvable. Plans can be formed by moving robots in a systematic manner in the grid; for example, from left to right and from top to bottom. Since the grid can be explored in $O(N^2)$ actions by the robots, unlocking cells takes $O(N^4)$ actions and keys can be delivered in $O(MN^2)$ actions. Therefore, feature action restricted problems in the grid domain can be solved in polynomial time.

The miconic domain is based on an elevator control problem. Elevators can move up and down between floors and people can get on or off an elevator. The task is to carry people from their original floors to their destination floors by elevators. The restricted miconic problems can be solved by calculating the required number of "Up" actions (people whose destination is higher than their original floor). If there are enough Up actions allowed in total, all elevators can move down to the lowest floor first. Next, elevators move up to deliver people who need to go up, and then one or more elevators move to the highest floor and move down to serve people who need to go down.

Our analysis and experiments show that constraint-based planners appear to be more robust over a range of multi-agent domains. However, heuristic planners are still highly efficient in certain of our multi-agent benchmark domains.

In addition, the above results demonstrate that our approach to modifying problem instances can be used to evaluate the potential performance for various planners on multi-agent problems. Meanwhile, our encodings allow us to enforce a certain minimal level of participation for each agent. These instances also provide an interesting multi-agent challenge benchmark for evaluating planning systems. The results on the testbed problems show that heuristic search and constraint-based planners complement each other on the test domains.

However, two even more interesting questions remain. First, are there ways to make heuristic search planners more robust on multi-agent problems? Second, are there ways to make constraint-based planners more scalable? Ideally, planning systems should be able to find plans in which participation requirements can be met without modifying problem instances. Therefore, a better solution to the multi-agent problems may be a hybrid system that incorporates both constraint-based and heuristic search planner techniques.

3.5 Summary

We have shown ways of adapting planning benchmark problems to incorporate participation requirements for multi-agent instances. Our proposal for encoding participation requirement into the problem instances has the advantage of maintaining the basic plan search nature of the benchmark problems (solution time provides a good measure of performance). Moreover, no changes need to be made to the planners themselves and no special post-processing of the plans is required. An interesting unanswered question is whether it is possible to similarly encode other quality measures such as “plan robustness” into the planning problem instances.

Using the modified multi-agent benchmark problems, we have given a detailed comparison of constraint-based and heuristic search based planning techniques. Our results show that these planners have complementary strengths. More specifically, heuristic planners are very efficient on certain multi-agent domains, such as grid and miconic domains, but are less efficient on domains with more intricate multi-agent interactions. Constraint-based planners, on the other hand, are more robust over a range of multi-agent problems but are less efficient than heuristic planners on the domains in which the heuristic planners excel. Although ways to make heuristic search planners more robust on multi-agent problems and ways to make constraint-based planners more scalable are still unclear, the complementary nature of these planners suggests that perhaps a hybrid approach would lead to yet a more powerful class of planners. An investigation of the inherent search topology (Hoffmann, 2001) of the multi-agent plan space would also be an interesting future research direction and might provide insights for better solutions.

Chapter 4

Control Knowledge in Planning

In recent years, there has been a burst of activity in the planning community with the introduction of a new generation of constraint and graph-based approaches, such as GRAPHPLAN (Blum and Furst, 1995) and BLACKBOX (Kautz and Selman, 1996, 1999). These planners are domain independent and outperform more traditional planners on a range of benchmark problems. The surprising effectiveness of these planners represents a departure from the long-held belief that the use of domain-specific planning control knowledge is unavoidable during plan search. Nevertheless, control knowledge has the potential to significantly increase the performance of the new planners. In fact, the constraint-based framework behind GRAPHPLAN and BLACKBOX allows, at least in principle, the incorporation of control knowledge in a purely declarative manner by encoding the control as additional constraints.

A recent example of the effectiveness of declarative control knowledge is the TLPLAN system by Bacchus and Kabanza (2000). In the TLPLAN system, control knowledge is represented by formulas in temporal logic. For example, the "next" operator from temporal logic allows specification of what can and cannot happen at the next time step. The control knowledge is used to steer a forward-chaining planner. One of the surprises of this system is that, despite the rather basic search method, with the right control knowledge, the system is highly efficient on a range of benchmark problems, often outperforming GRAPHPLAN and BLACKBOX (Bacchus and Kabanza, 2000). Of course, in this comparison GRAPHPLAN and BLACKBOX ran without any control; in addition, developing the right control formulas for TLPLAN is a non-trivial task.

As Bacchus and Kabanza point out, the forward-chaining search approach is a good match with the declarative control specification. At each node in the search tree, the control formula is evaluated to determine what new nodes are reachable from the current state. With good control knowledge, many nodes are pruned and the search is "pushed" towards the goal state. To give some intuition as to how this is achieved, we point out that the control rules can encode information about the difference between the current state and the goal state by using a predicate that states, for example, "package currently not at goal location."

One interesting research question is whether the same level of control can be effectively incorporated into the GRAPHPLAN or BLACKBOX style planner. This is far from a trivial question because TLPLAN's efficiency stems from the tight coupling between the control rules and the forward-chaining search strategy. In addition, TLPLAN allows for almost arbitrary complex control formulas that can generally be evaluated efficiently at each node (the process is in general intractable but in practice appears efficient for control information (Bacchus and Kabanza, 2000)). In the GRAPHPLAN or BLACKBOX framework the search proceeds very differently. The planning task is captured as a set of constraints mapped out over a fixed number of time steps. In GRAPHPLAN, the constraints are captured in a planning graph, which is subsequently searched for a valid plan. In BLACKBOX, the constraints are translated into a propositional formula, which can be searched with a satisfiability tester of the user's choice. In any case, in neither GRAPHPLAN nor BLACKBOX, dose the search proceed through a set of well-defined world states. In fact, the search may even involve intermediate states that are unreachable from the initial state or even physically impossible to attain. In SATPLAN, especially, the search is difficult to characterize, because the problem is reduced to a generic propositional representation without an explicit link to the original planning problem. The SAT solvers proceed by finding a truth assignment for the encoding (corresponding to a valid plan) without taking into account the fact that the encoding represents a planning problem (Baioletti *et al.*, 1998; Ernst *et al.*, 1997; Kautz and Selman, 1998).

One way to incorporate the control knowledge from TLPLAN into a SATPLAN encoding is to specify additional constraints. We have extended the PDDL planning input language (McDermott, 1998) of the BLACKBOX planner to allow for plan control as specified in temporal logic formulas. The control knowledge is automatically translated into a class of additional propositional clauses, which are added to the planning formula. We will discuss below the kinds of control that can be efficiently translated into a set of constraints, as well as the rules that cannot be captured efficiently.

Using a detailed experimental evaluation, we will show that control knowledge can indeed speed up the plan search significantly in the BLACKBOX framework. We also provide a detailed comparison with TLPLAN. As we will show, our planner becomes competitive with the TLPLAN performance. Initially, this was somewhat of a disappointment to us because we assumed that the BLACKBOX framework with control should be able to outperform TLPLAN with the same control. However, a closer inspection of the results clarified the difference in the approaches.

TLPLAN is good at finding plans with a relatively few actions but it does not do well in domains that allow for parallel actions. In particular, we studied the logistics planning domain (see Appendix A for a complete definition of a logistics domain in PDDL). This domain involves the task of delivering a set of packages to a number of different locations using one or more airplanes and vehicles. When several airplanes are available, one can find parallel plans, in which different packages are moved using different airplanes simultaneously, allowing minimization of the overall time span of the plan. Much of the combinatorics of the domain arises from the problem of finding good ways to interleave the movements of packages and airplanes. The sophisticated control in TLPLAN will dramatically narrow the search. However, combined with the depth-first forward-chaining strategy, the planner generates highly sequential plans. In fact, on the larger problems, the plan will use a single airplane to deliver all packages. (Such plans can actually be found in polynomial time.) BLACKBOX, on the other hand, naturally searches for the shortest parallel plan because it operates by searching plans that fit within a certain number of time steps. So, although both planners with control find plans in comparable amounts of time, BLACKBOX produces plans that have a much higher level of parallelism, thereby tackling the true combinatorial nature of the underlying planning problem. It is not clear how TLPLAN can be made to generate more parallel plans. (We describe several attempts to increase TLPLAN's parallelism below.) Which planner should be preferred will depend on the application and the level of inherent parallelism in the domain.

We believe our analysis provides new insights into the relative performance of different state-of-the-art planning methods. We hope that our findings can be used to further enhance these methods, and in general deepen our understanding of the design space of planning systems (Kambhampati, 1997).

4.1 Temporal Logic for Control

In TLPLAN the control knowledge is encoded in temporal logic formulas (Bacchus and Kabanza, 2000). The best way to introduce this approach is by considering an example control formula:

$$\Box\,(\forall\,[\,p : \texttt{airplane}(p)\,]\;\exists\,[\,l : \texttt{at}(p\;\;l)\,]\;\forall\,[\,o : \texttt{in-wrong-city}(o\;\;l)\,]$$
$$\texttt{in}(o\;\;p) \Rightarrow \bigcirc\;\texttt{in}(o\;\;p))$$

In temporal logic, $\Box\, f$ means f is true in all states from the current state onwards and $\bigcirc\, f$ means f is true in the next state. Therefore, the above formula can be read as "If a package o is in airplane p, p is at location l, and l is not in o's goal city, then package o should stay in airplane p in the *next* time step," and the above sentence should "always" hold. Predicate `in-wrong-city` is defined as follows:

$$\texttt{in-wrong-city}(o\;\;l) \equiv$$
$$\exists\,[\,g : \texttt{goal}(\texttt{at}(o\;\;g))\,]\;\exists\,[\,c : \texttt{in-city}(l\;\;c)\,]\;\neg\texttt{in-city}(g\;\;c)$$

After careful review of the control rules used by Bacchus and Kabanza, we found that the rules can be classified into the following categories:

I Control that involves only static information derivable from the initial state and goal state;

II Control that depends on the current state and can be captured using only static user-defined predicates; and

III Control that depends on the current state and requires dynamic user-defined predicates.

The meaning of these categories may not be immediately obvious. Hopefully, the detailed examples below will clarify the distinctions.

We will focus on two different ways of incorporating control: pruning the planning graph (rules from category I), and adding additional propositional clauses to the planning formula (rules from categories I and II). The rules in category III cannot be captured compactly.

4.1.1 Control by Pruning the Planning Graph

GRAPHPLAN constructs a special graph, called the planning graph, structured from the initial plan specification. GRAPHPLAN uses this graph to search for a plan leading from the initial state to the goal state. BLACKBOX translates the graph into a propositional encoding and then uses various SAT solution methods to search for a satisfying assignment, which corresponds to a plan. The BLACKBOX strategy closely mimics the original SATPLAN approach using linear encodings generated directly from a set of logical axiom schemas. The planning graph contains two types of nodes, propositional nodes ("facts") and action nodes, arranged into levels indexed by time.

Certain control formulas can be used to directly prune nodes in this planning graph. Consider the following control rule (category I) in the logistics domain:

Rule 1 Do not unload an object from an airplane unless the object is at its goal destination.

To illustrate this rule, we consider the problem $\mathcal{L}$ in Figure 2.1 where package `pkg1` is initially located in `NYC` and its destination location is in `ITH`. Once `pkg1` is loaded into an airplane `plane`, it is

Table 4.1: Actions and facts pruned by Rule 1 in Problem $\mathcal{L}$.

Pruned Actions	Prunded Facts
Unload-Airplane (pkg1 plane NYC)	(at pkg1 SYR)
Unload-Airplane (pkg2 plane NYC)	(at pkg2 NYC)
Unload-Airplane (pkg1 plane SYR)	
Unload-Airplane (pkg2 plane SYR)	

not necessary to unload pkg1 at the airports in cities other than ITH. In other words, action Unload-Airplane (pkg1 plane SYR) can be removed from the planning graph at all levels. After pruning these nodes, one can also prune the fact nodes (at pkg1 SYR), since this cannot be achieved by any other action except for the one we just pruned. Table 4.1 shows the actions and facts that are pruned by Rule 1 in the planning graph.

When constructing the planning graph, the planner can be instructed to prune action nodes in the plan graph as implied by the category I rules. In our implementation, Rule 1 is captured by augmenting the original PDDL (McDermott, 1998) planning language used in BLACKBOX as shown in Figure 4.1.

The user-defined predicate "in-wrong-city" is used to determine whether a location is the goal location for an object. It is worth noting that the value for predicate "in-wrong-city" can be decided purely based on the information in the goal state. The instantiated Unload-Airplane actions will not be added to the planning graph when the predicate (in-wrong-city ?obj ?loc), which is the "exclude" condition in the Unload-Airplane rule, is true. The "in-wrong-

```
(:action Unload-Airplane
 :parameters (?obj ?airplane ?loc)
 :precondition (and (obj ?obj)
                    (airplane ?airplane)
                    (location ?loc)
                    (in ?obj ?airplane)
                    (at ?airplane ?loc))
 :effect (and (not (in ?obj ?airplane))
              (at ?obj ?loc)))


(:defpredicate in-wrong-city
 :parameters (?obj ?loc)
    (exists (?goal-loc) (goal (at ?obj ?goal-loc))
       (exists (?city) (in-city ?loc ?city)
          (not (in-city ?goal-loc ?city)))))

(:action Unload-Airplane
 :exclude (in-wrong-city ?obj ?loc))
```

Figure 4.1: Rule 1 (category I) in a logistics domain.

`city`" predicate is purely an auxiliary predicate, and it is not incorporated into the final planning graph.

Before an instantiated action is added into the planning graph, it will be checked against all exclude conditions for that action; if any of them holds, the action will be pruned and the effects introduced by that action at the next propositional node level may be pruned as well.

Table 4.2 shows the planning graph size before and after pruning. We see that, in the logistics domain, when applying the category I control rules from TLPLAN, the number of nodes is reduced by ≈40%.

Table 4.2: The effect of graph pruning (category I rules).

problem	length	#original nodes	#nodes with pruning
logistics-a	11	4246	2825
logistics-b	13	5177	3437
logistics-c	13	6321	3815
logistics-d	14	9842	5135

This approach can be easily applied to the descendants of GRAPHPLAN and BLACKBOX. However, pruning of the planning graph does require that the rules rely solely on information of the goal state and possibly the initial state. Adding new propositional clauses provides a more powerful mechanism for adding control.

4.1.2 Control by Adding Constraints

For some domain-specific knowledge that cannot be captured via pruning of the planning graph, we can often add additional clauses to the propositional plan formulation as used in BLACKBOX to capture the control information. Consider the following control rule (category II):

Rule 2 Do not move an airplane if there is an object in the airplane that needs to be unloaded at that location.

First, note that this rule cannot be captured using graph pruning. For example, we consider removing the action node Fly-Airplane `(plane apt1 apt2)` in layer i (*i.e.*, time i). Whether this can be done would depend on the truth value of the predicates that indicate

```
(:wffctrl r2
 :scope
    (forall (?pln) (airplane ?pln)
       (forall (?loc) (airport ?loc)
          (forall (?obj) (obj ?obj)
             (not (in-wrong-city ?obj ?loc)))))
 :precondition
    (and (at ?pln ?loc)
         (in ?obj ?pln))
 :effect
    (next (at ?pln ?loc)))
```

Figure 4.2: Rule 2 (category II) in a logistics domain.

what is in the `plane` at time i, but, of course, those truth values are not known in advance and are actually different for different plans. Therefore the node cannot be removed from the planning graph without the risk of losing certain plans. What is needed is addition of clauses into the planning formula that in effect capture the rule but also depend on the truth values of the propositions that indicate what is in an airplane.

Logically, Rule 2 can be illustrated as follows:

$$\forall [\, p : \texttt{airplane}(p)\,]\, \forall [\, l : \texttt{airport}(l)\,] \forall [\, o : \texttt{(not}$$
$$\texttt{(in-wrong-city(}o\ \ l\texttt{)))}\,]$$
$$(\texttt{at}(p\ \ l) \wedge\, \texttt{in}(o\ \ p)) \Rightarrow \bigcirc\, \texttt{at}(p\ \ l)$$

Figure 4.2 shows how to translate the above rule in our implementation. The `:scope` field defines the domain where the rule applies, while `:precondition` and `:effect` fields represent the antecedent and consequent for the implication expression that will be added

```
(:wffctrl r1p
 :scope
    (forall (?pln) (airplane ?pln)
       (forall (?obj) (obj ?obj)
          (forall (?loc) (airport ?loc)
             (in-wrong-city ?obj ?loc))))
 :precondition
    (and (at ?pln ?loc)
         (in ?obj ?pln))
 :effect
    (next (in ?obj ?pln)))
```

Figure 4.3: Rule 1 in propositional control form.

to the propositional formula. As mentioned earlier, predicate "`in-wrong-city`" is only used by the system and will not be added into the planning formula. In fact, we can also translate Rule 1 into propositional form as shown in Figure 4.3.

It is worth noting that the predicate "`in-wrong-city`" used in the rules shown in Figures 4.2 and 4.3 is not added to the planning formula. It simply functions as a filter (indicated by the "`:scope`" keyword) for adding the clauses defined by `:precondition` and `:effect` part. This limited function ensures that we do not lose any information because "`in-wrong-city`" is static (independent of current state) and completely defined by the goal state. For example, if the goal destination of package `pkg1` is in city `ITH`, it follows that both `(in-wrong-city pkg1 NYC)` and `(in-wrong-city pkg1 SYR)` are true, independent of the current state. In the next section, we will see that category III rules involve defined predicates that do not have this property.

Table 4.3: Comparison between graph pruning and propositional control of category I control rules.

problem	len.	before simplification		after simplification	
		graph pruning	propositional control	graph pruning	propositioanl control
		#vars	#vars	#vars	#vars
rocket-a	7	1004	1337	826	826
rocket-a	7	1028	1413	868	868
log-a	11	2175	2709	1505	1511
log-b	13	2657	3287	2182	2182
log-c	13	3109	4197	2582	2590
log-d	14	4241	6151	3547	3551
log-e	15	5159	7818	4285	4285
log-1	9	2608	3043	1879	1879
log-2	11	11135	15655	9875	9875
tire-a	12	781	812	269	269
tire-b	30	5875	8504	4985	5114

One reasonable question is how adding category I rules using additional clauses compares to the graph pruning approach discussed earlier. We used the six category I control rules from TLPLAN. In Table 4.3, we present the planning formulas created from the planning graph. The first two columns give the number of variables for the two different strategies. As might be expected, the planning graph pruning strategy gives the smallest number of variables. However, we also include the result of running a polynomial time simplification procedure (Crawford, 1984). As the last two columns illustrate, the remaining numbers of variables are identical for most formulas, with only some small differences for certain instances. When we checked into the remaining differences, we found that they result from some specialized additional pruning done by BLACKBOX specifically for the final layer of the planning graph. Overall, Table 4.3 shows that direct graph pruning or a coding via additional clauses leads to formulas on essentially the same set of variables. In addition, we also verified that the variables actually refer to the same planning propositions in terms of the original planning problem. As a result, the two mechanisms are essentially equivalent for category I rules, although they do capture the planning problem using a different clause set. Below we will compare the computational properties of these two approaches.

4.1.3 Rules With No Compact Encoding

We now consider the category III rules. Although these rules can be translated into additional propositional constraints in principle, they do lead to an impractical number of large clauses to be added to the planning formula. Consider the following rule from TLPLAN:

Rule 3 Do not move a vehicle to a location unless (1) the location is where we want the vehicle to be in the goal, (2) there is a package at that location that needs to be picked up, or (3) there is a package in the vehicle that needs to be unloaded at that location.

The main purpose of Rule 3 is to avoid unnecessary moving of vehicles. First, the rule requires current state information, and therefore cannot be handled by graph pruning. Secondly, in order to translate the rule into propositional constraints, we need to introduce extra predicates to represent the fact that there is a package in the destination location that needs to be loaded or unloaded by the vehicle. For example, in order to avoid unnecessary moving of airplanes (a form of vehicle), one way to encode it is to define the predicate "`need-to-move-by-airplane`" for each airport first:

$$\forall [\, o, l : \texttt{in-wrong-city}\ (o\ \ l) \,]\ \texttt{at}(o\ \ l) \Rightarrow$$
$$\texttt{need-to-move-by-airplane}(l)$$

$$\forall [\, l\ : \texttt{need-to-move-by-airplane}(l) \,] \Rightarrow \exists [\, o : \texttt{in-wrong-city}\ (o\ \ l) \,]$$
$$\texttt{at}(o\ \ l)$$

We can also define the predicate "`need-to-unload-by-airplane`" in a similar way as follows:

$$\forall [\, p : \texttt{airplane}(p) \,]\ \forall [\, o, l : \neg\ \texttt{in-wrong-city}\ (o\ \ l) \,]$$
$$\texttt{in}(o\ \ p) \Rightarrow \texttt{need-to-unload-by-airplane}(p\ \ l)$$

$$\forall [\, p : \texttt{airplane}(p) \,]\ \forall [\, l\ : \texttt{need-to-unload-by-airplane}(p\ \ l) \,] \Rightarrow$$
$$\exists [\, o : \texttt{in}\ (o\ \ p) \,]\ \neg \texttt{in-wrong-city}(o\ \ l)$$

And finally, we will need the following formula to encode Rule 3:

$$\forall [\, p : \texttt{airplane}(p) \,] \forall [\, l1, l2 : (\texttt{not}\ (= l1\ \ l2)) \,]$$
$$(\texttt{at}(p\ \ l1) \wedge \neg\, \texttt{need-to-unload-by-airplane}(p\ \ l2) \wedge$$
$$\neg\, \texttt{need-to-move-by-airplane}(l2))\ \Rightarrow\ \bigcirc\ \texttt{at}(p\ \ l1)$$

Supposing there are n packages, m cities, and k airplanes, the above encoding will introduce $O(mn)$ predicates and $O(mn + km^2)$

propositional clauses in each time step. Furthermore, some of them are lengthy clauses, containing up to $O(mn)$ number of literals. We explored adding this kind of control information to our formulas but, except for the smallest planning problems, the formulas become too large for our SAT solvers. The key difference with category II rules, such as Rule 2, is that in this case we also need to add clauses that capture our defined predicates, e.g., "`need-to-move-by-airplane`". This capture is in contrast with the encoding of, *e.g.*, Rule 2, where, because of the static nature of the defined predicates, they can simply be used as a filter when adding clauses (see discussion on `:scope` above). This filtering cannot be done for our "`need-to-move-by-airplane`" because its truth value depends on where a package is at the current time. As a consequence, the category III rules are examples of rules that cannot effectively be captured into a constraint-based planner such as BLACKBOX. Fortunately, as we will see below, in terms of computational efficiency the category I and II rules appear to do most of the work, at least in the domains we have considered. Therefore, another interesting research question is whether there is a more effective way to encode category III rules.

4.2 Empirical Evaluation

The testbed used in this section is a series of problems from the logistics planning domain and the rocket domain (Veloso, 1992; Blum and Furst, 1995; Selman and Kautz, 1996; McDermott, 1998), and the tireworld domain from the TLPLAN distribution (Bacchus and Kabanza, 1998). In addition, we created two new problem instances: logistics-1 and logistics-2, which can be solved with highly parallel plans (up to 20 actions in parallel). All experiments were run on a 300Mhz Sparc-Ultra machine. Times are given in CPU seconds.

Table 4.4 gives the results of BLACKBOX running on problems with different levels of domain knowledge. We used six category I rules and four category II rules (see Appendix B for the complete set of hand-coded control rules used in experiments). Results of randomized solvers were averaged over ten runs. The column labeled "BLACKBOX" gives the run time without any control knowledge. Subsequent columns give results for the different control strategies. As a general observation, we note that the effect of control knowledge becomes more apparent for the larger problem instances. For example, on our hardest problem, "logistics-e," basic BLACKBOX takes almost one hour, but with category I_a and II control, it only takes 60 seconds. The results on the larger instances are more significant than those for the smaller problems, because the solution times on the smaller instances are often dominated by basic I/O operations such as reading the formula from disk, as opposed to the actual computational time for solving the formulas. Based on the table, we can make the following observations:

Table 4.4: BLACKBOX with control knowledge.

I_a: Category I control knowledge used for pruning planning graph.
I_b: Category I control knowledge added in propositional form.
II: Category II control knowledge added in propositional form.

problem	length	BLACKBOX	BLACKBOX (I_a)	BLACKBOX (I_b)
		time	time	time
rocket-a	7	2.06	4.20	3.41
rocket-b	7	2.87	2.09	2.46
logistics-a	11	3.80	2.78	3.64
logistics-b	13	4.83	3.46	4.56
logistics-c	13	6.75	3.89	5.64
logistics-d	14	15.85	7.29	10.4
logistics-e	15	3522	151	201
logistics-1	9	4.80	3.68	4.97
logistics-2	11	406	–	270
tire-a	12	1.37	1.34	1.35
tire-b	30	114	93	72

problem	length	BLACKBOX (II)	BLACKBOX (I_a&II)	BLACKBOX (I_b&II)
		time	time	time
rocket-a	7	2.10	3.92	3.82
rocket-b	7	3.26	1.85	2.49
logistics-a	11	3.74	2.78	3.63
logistics-b	13	4.75	3.41	4.66
logistics-c	13	6.71	3.94	5.81
logistics-d	14	15.69	6.82	10.23
logistics-e	15	2553	60	148
logistics-1	9	4.83	3.70	4.98
logistics-2	11	360	130	141
tire-a	12	1.36	1.33	1.34
tire-b	30	55	21	23

- Control information does reduce the solution time, especially on the harder problem instances. Consider the data on logistics-e, logistics-2, and tire-b.

- Category I control rules, encoded via graph pruning and as additional clauses (columns I_a and I_b) lead to roughly the same speedup on the larger problems. One notable exception is the logistics-2 problem, where the solution time actually goes up for strategy I_a. This increase is most likely a consequence of the fact that clauses are eliminated by the pruning, leading to less propagation in the SAT solver. We do not see this problem when the knowledge is added via additional clauses (I_b), which therefore appears to be a more robust strategy. The smaller problem instances are already solved within a few seconds by basic BLACKBOX; therefore, not much can be gained from control (again, partly because I/O dominates the overall times).

- Category I rules are most effective on problems from the logistics domain; category II rules are more effective in the tireworld. One interesting research issue is whether one can identify in advance which kinds of rules are most effective for a domain (measurements such as clause-to-variable ratios and degree of unit propagation may be useful here).

- The effect of control is largely cumulative. Our category I rules combined with category II lead to the best overall performance (see columns I_a&II and I_b&II).

Table 4.5: Comparison between BLACKBOX and TLPLAN.

problem	BLACKBOX(I_a&II)			TLPLAN-dfs		
	length	time	#action	length	time	#action
logistics-a	11	2.78	72	13	0.49	51
logistics-b	13	3.41	71	15	0.4	42
logistics-c	13	3.94	83	17	0.64	51
logistics-d	14	6.82	104	26	2.13	70
logistics-e	15	60	107	24	4.27	89
logistics-1	9	3.70	77	15	0.9	44
logistics-2	11	130	147	29	33.16	93
problem	BLACKBOX(I_a&II)			TLPLAN-rand-dfs		
	length	time	#action	length	time	#action
logistics-a	11	2.78	72	15	0.66	57
logistics-b	13	3.41	71	15	0.43	46
logistics-c	13	3.94	83	15	0.72	55
logistics-d	14	6.82	104	18	2.43	75
logistics-e	15	60	107	23	4.57	96
logistics-1	9	3.70	77	15	1.01	49
logistics-2	11	130	147	16	34.52	100

Next, we compare the performance of TLPLAN and BLACKBOX. Table 4.5 gives our results. We note that, in general, TLPLAN is still somewhat faster than BLACKBOX with similar control. Note that both planners now use the same control information except for some category III rules in TLPLAN. However, the differences are much smaller than with the original BLACKBOX without control (see Table 3 in Bacchus and Kabanza, 2000). We believe these results demonstrate that we can meet the challenge, proposed by Bacchus and Kabanza, to effectively incorporate declarative control as used in TLPLAN into a constraint-based planner. Nevertheless, as noted at the beginning of this chapter, we were somewhat disappointed that BLACKBOX with control was not faster than the TLPLAN approach, given the more sophisticated search of the SAT solvers. Therefore, in order to get a better understanding of the issues involved, we consider the plan quality, especially, in terms of the parallel plan length, of the generated plans.

Table 4.5 gives plan length in terms of number of actions and parallel time steps for the logistics domain. Note that the logistics domain allows for a substantial amount of parallelism because several planes can fly simultaneously. We see that TLPLAN often finds plans with fewer actions; however, in terms of parallel lengths the plans are much longer than those found by BLACKBOX. In fact, BLACKBOX because of its plan representation, can often find the minimal length parallel plan. On the larger problems, especially, we observe a substantial difference. For example, on logistics-e, TLPLAN requires 24 time steps versus 15 for BLACKBOX. After a closer look at the plans generated by TLPLAN, we found plans that only use a single airplane to transport the packages. Such plans can be found very quickly in polynomial time but ignore much of the inherent combinatorics of the domain. The reason TLPLAN finds such plans is a consequence of its control rules and the depth-first search strategy.

It is not clear how to improve the (parallel) quality of the plans generated by TLPLAN. Part of the difficulty lies in the fact that the control knowledge is tailored towards more sequential plans. For example, rules that keep an airplane from flying if there are still packages to be picked up at a location prevent a plan where other airplanes can pick up the packages later. In addition, the depth-first style search in TLPLAN also tends to steer the planner towards more sequential plans (*e.g.*, always picking `airplane-1` to move). We experimented with several different search strategies to try to improve the plans generated by TLPLAN. First, we checked whether the parallel quality of plans obtained with breadth-first search was better. This does not appear to be the case because we could only check this on very small instances; the TLPLAN search quickly runs out of memory on larger problems. An approach that does lead to some improvement is to randomize the depth-first search. Table 4.5 shows some improvement in parallel length but still nothing close to the minimal possible. The reason for the improvement is that TLPLAN now can pick different airplanes to move more easily in its branching because the deterministic depth-first search repeatedly selects airplanes in the same order. Interesting questions raised by these observations are how TLPLAN can be made to find more parallel plans and how this would affect its performance.

4.3 Summary

Intuitively speaking, the control in TLPLAN attempts to push the planning problem into a polynomial solvable problem. This attempt is along the lines of the general focus in planning on eliminating or avoiding search as much as possible.[1] TLPLAN achieves its objective in a very elegant manner because the control rules are quite intuitive

[1]Austin Tate is reported to have said “If you need to search you’re dead.”

and purely declarative. In combination with the forward-chaining planner, the rules do lead to polynomial scaling in a number of interesting domains. Yet, our analysis shows that there is a price to be paid for this gain in efficiency; namely, the loss of much of the parallel nature of many planning tasks.[2]

We have shown it possible to obtain the benefits of the control rules in terms of efficiency, without paying the price of reduced parallelism, by incorporating the control rules into the BLACKBOX planner. In a sense, the SAT solvers in BLACKBOX still tackle the combinatorial aspect of the task, but the extra constraints provide substantial additional pruning of the search space.

We implemented the system by enhancing the BLACKBOX planner with a parser for temporal logic control rules, which are translated in additional propositional clauses. We also showed that a subset of the control (category I rules) can also be handled by direct pruning of the planning graph. Our experimental results show a speedup due to the search control of up to one order of magnitude on our problem domains. Given the effectiveness of the control, it would be interesting to add further constraints, such as state invariants (Fox and Long, 1998; Gerevini and Schubert, 1998; Kautz and Selman, 1998).

We believe our work demonstrates that declarative control knowledge can be used effectively in constraint-based planners without loss

[2]Given that these planning problems are NP-complete, it is clear that something will be lost by solving them in polynomial time.

of (parallel) plan quality. Compared to TLPLAN, which is a highly efficient planner in and of itself, the main advantage of our approach is that we maintain parallel plan quality. Overall, incorporating declarative control into constraint-based planning appears to be a promising research direction. With more sophisticated control, another order of magnitude speedup may be achievable.

A natural question to ask is how this search control knowledge can automatically be acquired by the use of rule-based learning techniques. Learning of control knowledge has been explored previously for other, more procedural, planners (see, for example, Etzioni, 1993; Knoblock, 1994; Minton, 1988a; Veloso, 1992).

In the next chapter, we will present the first positive results for acquisition of such high level, declarative control knowledge using machine learning techniques by *training* the planner on a sequence of small problems.

Chapter 5

Learning Control Knowledge for Planning

Deterministic state-space planning is a hard combinatorial problem that arises in tasks such as robot control, software verification, and logistics scheduling. Although in general planning is PSPACE-complete (Bylander, 1991), for particular domains efficient algorithms – *i.e.*, polynomial or at worst exponential with a very low exponent – may exist for finding exact or approximate solutions.

Researchers in explanation-based learning (EBL), inductive learning, case-based reasoning, and genetic programming have long studied the problem of automatically creating efficient planners by learning domain-specific or case-based rules to control a general search engine (Mitchell *et al.*, 1986; Minton, 1988a; Carbonell *et al.*, 1990; Veloso, 1992; Etzioni, 1993; DeJong and Mooney, 1986; Bhatnagar and Mostow, 1994; Kambhampati *et al.*, 1996; Borrajo and Veloso, 1997; Aler *et al.*, 1998; Leckie and Zukerman, 1998). However, the successful practical application of machine learning techniques has been limited by at least two factors. First, traditional domain-independent planning systems (*e.g.*, PRODIGY, SOAR, NONLIN, UCPOP) scale so poorly that extensive learned control knowledge is required to raise their performance to an acceptable level. Second, previous works have focused on learning control rules that are specific to the details of the underlying general planning algorithms. This approach leads to a need for the learning and managing of large numbers of rules, which can be extremely costly in terms of learning time and the required number of training examples. Furthermore, the learned domain-specific rules cannot be reused by other planners, nor can the learning module itself be ported to other systems without extensive modifications.

In recent years, a new generation of planning systems with much improved speed and scalability has become available (Weld, 1999). These systems formulate planning as the solving of a large constraint-satisfaction problem: GRAPHPLAN (Blum and Furst, 1995) and its descendents encode a CSP in a data structure called a plan graph, while SATPLAN and its descendents (Kautz and Selman, 1992, 1996, 1999) explicitly convert planning problems into Boolean satisfiability. The constraint-based formulation opens up the possibility that domain-specific control knowledge can be added to the planner in a purely declarative manner via a set of additional constraints. These new constraints do not make explicit reference to the workings of the underlying constraint-satisfaction algorithm; like the constraints that define the original problem instance, they only refer to the solution (plan) space. In earlier work (Kautz and Selman, 1998; Huang *et al.*, 1999), we showed that the same set of *hand-coded* declarative constraints can provide dramatic reductions in solution time of radically different constraint-satisfaction algorithms (*e.g.*, local search or systematic search), and that a large subset of the constraints can be employed by fundamentally different planning architectures (*e.g.*, the forward-chaining planner TLPLAN [Bacchus and Kabanza, 2000]). The next natural question to ask is whether this kind of declarative control knowledge can be learned.

In this chapter we present our initial positive results on automatic acquisition of such constraints using machine learning techniques. The training set consists of a small number of planning problems (in the experiments here, ten or fewer) together with their optimal solutions. We will describe how positive and negative examples of the target concepts used by the control rules are heuristically extracted from the input data. The rules are generated by an inductive logic programming approach based on the FOIL algorithm (Quinlan, 1990, 1996); unlike much work in inductive logic programming, however, explicit background knowledge in the form of defined predicates is not supplied to the system. Instead, the categorization of the kinds of predicates, the state information that appears in the justified plan, and the inferred type information of objects and predicates are all considered kinds of background knowledge automatically acquired by the system. Experimental evaluation on different benchmark domains from a recent planning competition is quite promising: training time is very short (on the order of a minute), and the system learned small sets of high-quality rules. Adding these rules to our constraint-based planner reduced solution times by two orders of magnitude or more on large problems while maintaining or improving plan quality.

We will illustrate the details of the system using examples from the logistics planning domain (Veloso, 1992), which has appeared as a benchmark in most recent work in planning. In brief, the task in this domain is to move a set of packages from various initial locations to various goal locations. Packages can be moved between locations within a city by truck. Airplanes can transport packages between airports in different cities. The basic actions are loading and unloading packages from vehicles, driving trucks, and flying airplanes. In the formulation used by the constraint-based planners considered here, all actions require one time unit for execution, and any number of non-conflicting actions may occur at the same time step. The number of time steps in a plan is its parallel length, and the number of actions is its sequential length. For example, a plan of parallel length ten with eight actions occurring at each time step would have sequential length 80. A plan can be deemed optimal in terms of its parallel or sequential length, or of some combined measure; in our work, the optimality criteria are to minimize parallel length and then to minimize sequential length.

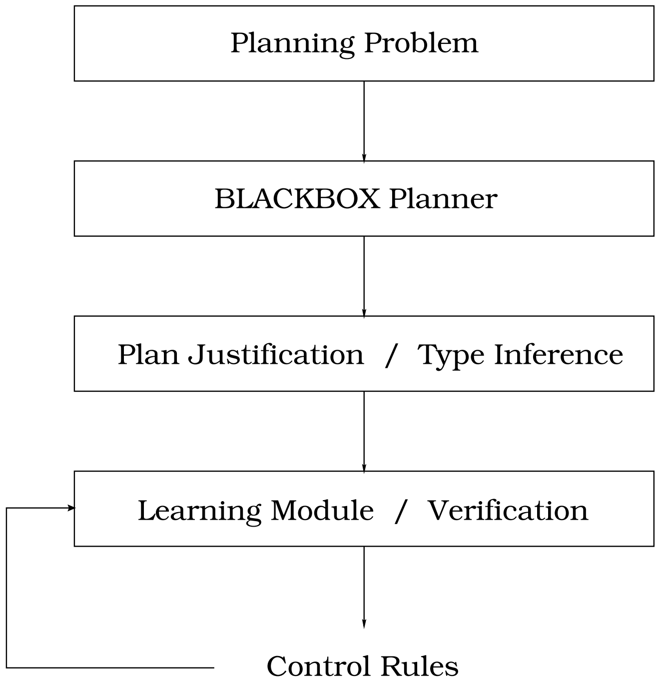

Figure 5.1: The basic learning framework.

5.1 Learning Framework

Our general learning framework is shown in Figure 5.1. The process begins by presenting the planning module with a training problem instance of small to moderate size from the given planning domain. We use the BLACKBOX planner (Kautz and Selman, 1999), which combines both the GRAPHPLAN and SATPLAN systems mentioned above. BLACKBOX generates a plan of optimal parallel length. The plan is then passed through a justification algorithm that minimizes its sequential length by removing sets of unnecessary actions (Fink and Yang, 1992). The justified plan also includes a description of the complete state at each time step, which can be easily extracted from the planning graph with the plan. Meanwhile, a type inference algorithm computes type information for all actions and objects in the problem (Fox and Long, 1998). The justified plan and type information are passed to the learning module.

The learning module uses the input plan in two ways. First, any previously learned rules are verified against the plan, and inconsistent rules are discarded. Second, a set of positive and negative examples of target concepts are heuristically extracted from the plan, as will be described in detail below. This step depends upon the crucial fact that the plan is optimal or near optimal so that noise introduced by incorrectly identified examples is reduced. Those examples not covered by previous rules are used as training data by an inductive rule learning algorithm, which generates one or more new control rules. The type information inferred earlier improves the speed and quality of rule induction. (Note, however, that some induced rules may be incorrect, thus producing a need for the verification step.) The process is then repeated for several problem instances and then the learned rules are verified against all training problems. The final set of learned rules is output in the form of a set of logical axioms, which can be used by either the original planner or a variety of other planning engines.

We next consider aspects of the system in more detail.

5.1.1 Target Concepts for Actions

For each action in the planning domain, the system learns two complementary concepts, *select action* and *reject action*. Each concept is defined by a set of logic programming-type rules in a simple temporal logic. Select rules indicate conditions under which the action must be performed, and reject rules indicate conditions under which it must not be performed.

Both kinds of rules can be divided into two categories according to the information upon which they rely (Huang *et al.*, 1999). A *static* rule is one whose body depends only upon the initial and goal states specified in the planning problem, but not upon the particular time step at which the action should be selected or rejected. Thus, a static rule holds for all time steps in the problem instance. A *dynamic* rule is one whose body also depends upon what is true at the "current" time step. Therefore, it is applied only at the time step when its pre-conditions hold at that state. As described below, static and dynamic rules are learned separately.

The following are examples of different kinds of rules from the logistics domain:

static reject: Do not unload a package from an airplane at an airport if that airport is not in the package's goal city.

dynamic reject: Do not move an airplane if it currently contains a package that needs to be unloaded at that city.

dynamic select: Unload a package from a truck at the package's goal location if the package is currently in the truck.

The logistics domain does not happen to contain any static select rules; these would arise in domains where some particular action

Table 5.1: Types of actions used in training sets for the static and dynamic varieties of the select and reject rules.

	select rule		*reject* rule	
	positive ex.	negative ex.	positive ex.	negative ex.
static	real	virtual	virtual	real
dynamic	real	mutex virtual	mutex virtual	real

must be repeatedly performed at every step of the plan in order to achieve the goal. In fact, only the actions whose preconditions hold in every state can appear in static select rules.

5.1.2 Heuristics for Identifying Training Examples

A traditional EBL approach would find examples of the select and reject concepts by examining a trace of the planner or by re-deriving the solution to a solved instance. In our approach, by contrast, the training examples are heuristically derived from the solved problem instance. The motivation of our heuristics is based on the notion that there is a good chance that the particular actions that appear in an *optimal* solution *must* be selected, and those that do not appear *must* be rejected. In other words, we are using learning to operationalize the optimality criteria of the planner and justification module. This method of generating examples is obviously fairly noisy, and we will describe the techniques we use to minimize the effect of incorrectly labeled examples.

In particular, we assume that the plan $\mathcal{P}$ is found by the planner for a given problem. We will say an instantiated action is

real at time i if it appears in $\mathcal{P}$ at time i,

virtual at time i if all of its preconditions hold at time i but it does not belong to plan $\mathcal{P}$ at time i,

mutex virtual at time i if it is virtual and there exists at least one real action that is mutually exclusive with it at time i.

Two actions are mutually exclusive if they cannot occur at the same time, even if their preconditions both hold; for example, loading an airplane is mutually exclusive with flying that airplane.

Each element of the set of all actions instantiated at all times (up to the length of the solution) is categorized according to this scheme. Then the positive and negative examples for learning static and dynamic select and reject rules for each action are chosen according to the scheme specified in Table 5.1. Note that the training set for learning dynamic rules is a subset of the training set used for learning static rules, because it contains fewer examples based on actions that do not appear in the plan.

Why is the training set for dynamic rules restricted in this manner? In short, to reduce the amount of noise it contains. In general, learning good dynamic rules is more difficult than learning good static rules, because there are more dynamic rules that are consistent with the data. Furthermore, examples based on actions that do not occur are clearly less reliable than those based on actions that do occur. Therefore, it is visalbe to concentrate on examples of non-occurring actions that are relevant to the problem instance. Mutex

Table 5.2: Predicates in logistics domain.

static predicate	*fluent predicate*	*action predicate*
`obj`	`at`	Load-Truck
`truck`	`in`	Unload-Truck
`location`		Drive-Truck
`airplane`		Load-Airplane
`city`		Unload-Airplane
`in-city`		Fly-Airplane

virtual actions tend to be more relevant than others because their preconditions and effects overlap with those of actions that *do* occur.

5.1.3 Rule Induction

Control rules are generated from the training examples by a greedy general-to-specific search in the space of restricted temporal logic programs using an algorithm based on the FOIL procedure (Quinlan, 1990, 1996).

The simple temporal logic programs we consider are constructed as follows. There are three kinds of predicates: *static predicates*, which refer to facts whose truth cannot be changed by any action (*e.g.*, the predicate "`in-city`" used to assert that a particular location is part of a particular city); *fluent predicates*, used for facts whose truth varies over time (*e.g.*, the predicate "`at`" which relates a movable object and a location); and *action predicates*, used for parameterized actions (*e.g.*, "Fly-Airplane"). Table 5.2 summarizes the kinds of predicates in logistics domain. A single modal operator "goal" is used to assert that its argument is one of the specified goals of the planning problem. A *literal* is an expression of the following form or its negation, where P is a predicate, F is a fluent predicate, and each X_i is a variable:

$$X_i = X_j, \quad P(X_1, ..., X_n), \quad \mathsf{goal}(F(X_1, ..., X_n)).$$

A *rule* contains a distinguished literal, its head, and a set of literals that make up its body. An *instantiation* of a rule is created by substituting constants for all of its variables. A rule is *consistent* with a justified plan iff for all of its instantiations, the head is true at each time step at which all literals in the body are true. Note that literals and static predicate literals have the same truth value at all time steps in a justified plan; in particular, "goal" here refers only to *final* goals, not intermediate goals. This language can be enriched in various ways; for example, by including other modal operators such as "next" or "eventually," or by allowing a rule to contain explicit constants; we will explore these extensions in future research.

The head of a select control rule must be a positive action predicate literal, and the head of a reject rule must be a negative action predicate literal. As we have mentioned, static and dynamic rules are learned separately. These two kinds of rules are distinguished by the kinds of literals that may appear in their bodies:

Static rules may contain positive or negative equality literals, static predicate literals, and goal literals. Note that the truth of each of these kinds of literals does not vary across time steps.

$Rules = \emptyset$
$Remaining$ = examples for the target concept R

while $Remaining \neq \emptyset$

  $rule = R \leftarrow$ **null**

  while $rule$ covers negative examples

    Add a literal L to the body of $rule$ that maximizes gain

  Remove from $Remaining$ examples that are covered by $rule$

  Add $rule$ to $Rules$

Figure 5.2: Outline of the rule induction algorithm, based on Quinlan's FOIL.

Dynamic rules may contain all of the above, but must also contain at least one negative or positive (non-goal) fluent literal. Note that when checking the consistency of a dynamic rule, the (non-goal) fluent literals are evaluated at the same time step used to evaluate the rule's head.

It should be noted that all preconditions of the rule head action are implicitly included in the rule body. They are not used to distinguish the type of rules and are not considered as candidate literals in learning.

The outline of our FOIL-based learning algorithm is shown in Figure 5.2. As in FOIL, the literals added to the rule at each step are chosen according the following criteria:

- the literal with the greatest gain if this gain is close to the maximum possible; otherwise
- all determinate literals found;[1] otherwise
- the literal with the highest positive gain; otherwise
- the first literal considered that introduces a new variable.

We experimented with a number of different definitions of the *gain* function. We obtained the best performance using the "Laplace estimate" used in a number of recent learning systems (CN2 [Clark and Niblett, 1989], *etc.*):

$$\text{Gain}(r) = \frac{p+1}{p+n+2} \ ,$$

where r is the candidate rule, p is the number of positive examples covered by r, and n is the number of negative examples covered by r.

Because the Laplace estimate penalizes rules with low coverage, it proved to be more robust against noise in the training set since the sizes of our training sets are relatively small compared to other learning tasks (the number of training examples in our experiments are usually less than a hundred). Noise is also handled, as noted earlier, by pruning learned rules that turn out to be inconsistent with future examples of solved planning problems given to the learner.

[1]A *determinate* literal is one that introduces new variables so that there is exactly one binding for each positive example and at most one binding for each negative example in the partially-constructed rule (Muggleton and Feng, 1992; Quinlan, 1996).

We mentioned above that during plan justification, type inference on all objects and predicates is performed using the algorithm from TIM (Fox and Long, 1998). This inferred type information serves two purposes. First, types are used to reduce the number of rules considered in the learning procedure. For example, when considering candidate equality literals, the arguments of the equality literal must be of the same type. Second, when adding a literal that introduces new variables, inferred types are used to correctly find bindings for every new variable. For instance, when adding a literal (`at` o l) to a rule, where the types of o and l associated to the literal are "package" and "location," respectively, and o is a new variable, only objects with type "package" will be considered as bindings for variable o. In other words, objects with type "truck" or "airplane" that can also be bound to the first argument of the predicate (`at` o l) will not be considered. This use of type information is important for correct acquisition of control rules. In addition, the use of type information also dramatically reduces the number of candidate predicates considered by the learning system.

It is common in inductive logic programming for the user to provide explicit background knowledge to the system in the form of additional relations or axioms (Quinlan, 1990, 1996). For example, the user might write a definition for a predicate that holds "packages that need to be moved." One might argue, however, that manually defining good background knowledge is as difficult as defining good control rules. In our system, by contrast, no additional background knowledge is input by the user. The categorization of the different kinds of predicates, the state information that appears in the justified plan, and the inferred type information of objects and predicates can all be considered kinds of background knowledge automatically acquired by the system.

5.1.4 Backtracking

Since our learning system employs a greedy search approach, incorrect rules that only cover a few examples but are not consistent with the entire plan can be learned. This situation is more likely to happen in the learning of *dynamic* control rules. Therefore, we add a limited chronological backtracking mechanism to the learning system. Each rule is verified when it is learned. If a rule is not consistent with the input plan, the learning system removes the latest added literals from the rule in the previous phase and searches for the next best ones.

5.1.5 Rule Simplification

As shown in the learning procedure outline (Figure 5.2), literals that introduce new variables, such as determinate literals, are added to the control rule when necessary. However, it is common for some new variables to be introduced that are not used to find the control rule in the later learning phases. Therefore, literals that introduce unused new variables may become redundant.

Therefore, before a learned rule is generated, it is simplified by removing unnecessary literals in the rule. This simplification is be

L = literals that does not introduce new variables in *rule*

Mark all literals in L as *necessary*

Mark all variables in lit as *necessary* if $lit \in L$

while $\exists$ variable $var \notin$ *rule* head is *necessary* **and**

$\exists$ lit introduces var is not *necessary*

Mark lit as em necessary

Mark all variables in lit as *necessary*

Remove all literals in *rule* body that are not *necessary*

Figure 5.3: Outline of the rule simplification procedure.

done by first marking all literals that do not introduce new variables as *necessary*, and then by marking the variables in these literals as *necessary* as well. If a variable that is marked as necessary does not appear in the rule head, one must find the literal that introduces the variable and mark the literal and all variables in the literal as necessary. This process is repeated until no more variables and literals can be marked as necessary. Finally, literals that are not marked as necessary in the rule body are discarded. In this way, the output rule will only contain relevant literals in its rule body and reduce the complexity of learned rule. The outline of the above rule simplification procedure is shown in Figure 5.3.

5.1.6 Forming and Using Learned Control Rules

As shown in Chapter 3 and Huang *et al.*(1999), we extended the PDDL planning input language (McDermott, 1998) of the BLACKBOX planner to allow control knowledge to be specified using temporal logic formulas. BLACKBOX uses *static reject* rules to prune the Boolean SAT encodings from the planning graph it creates of planning problems All other kinds of rules are converted into propositional clauses that are added to the SAT encoding.

Therefore, our learning system will generate learned control rules in the extended PDDL control format, and thus the rules created by the learning module can be directly fed back to the planner.

All *static reject* rules are converted into graph pruning control rules in extended PDDL as shown in Section 4.1.1. The conversion is straightforward. All literals in the rule body are placed in the "`exclude`" condition in extended PDDL control and variables are existentially quantified.

When converting *dynamic* control rules, variables are first classified as *static* or *dynamic* variables according to the predicates that introduce them. A variable is *static* if it is introduced by a predicate with a goal modal operator or static predicate, and it is *dynamic* if it is introduced by a dynamic predicate. All predicates that are not dynamic are placed in the "`scope`" condition of extended PDDL control, where a dynamic variable is universally quantified and a static variable is existentially quantified. The "`precondition`" of the extended

PDDL control includes all dynamic predicates in the rule body and dynamic predicates in the precondition of original action definition. The "`effect`" of the extended PDDL control rule includes one dynamic predicate chosen from any dynamic predicates in the effect of the original action definition. The predicate in the `effect` is negated if the learned control rule is a *reject* rule; otherwise, it is not changed.

The general effect of the added clauses is to make the encoded problem easier to solve by increasing the power of the constraint propagation routines used by the system's SAT engines.

5.2 An Example

We will now step through an example of learning a control rule for the logistics domain. We consider a problem instance in which there are three cities (`A`, `B`, and `C`), each containing two locations, an airport and a post office (*e.g.*, `apt-A`, `po-A`, ..., *etc.*). In each city there is a truck (`trk-A`, `trk-B`, and `trk-C`), and there is one airplane (`plane`). There are two packages (`pkg1` and `pkg2`) to be delivered.

The initial and goal states for the problem and the (justified) plan found by the BLACKBOX planner are shown in Table 5.3.

Suppose we are learning *static reject* rules for the action "Unload-Airplane (obj $airplane$ loc)," where the types associated with variable obj, $airplane$, and loc are `object`, `airplane`, and `airport` respectively.

According to the heuristic for generating training examples for *static reject* rules, Unload-Airplane (`pkg1 plane apt-A`) at time 2 is a

Table 5.3: A simple logistics problem and its solution.

Initial:	(at pkg1 apt-A), (at pkg2 apt-B),	
	(at plane apt-A), (at trk-C apt-C),	
	(in-city apt-A A), (in-city po-A A), ...	
Goal:	(at pkg1 po-C), (at pkg2 po-C)	
Plan:	1	Load-Airplane (pkg1 plane apt-A)
	2	Fly-Airplane (plane apt-A apt-B)
	3	Load-Airplane (pkg2 plane apt-B)
	4	Fly-Airplane (pln apt-B apt-C)
	5	Unload-Airplane (pkg1 plane apt-C)
	5	Unload-Airplane (pkg2 plane apt-C)
	6	Load-Truck (trk-C pkg1 apt-C)
	6	Load-Truck (trk-C pkg2 apt-C)
	7	Drive-Truck (trk-C apt-C po-C)
	8	Unload-Truck (trk-C pkg1 po-C)
	8	Unload-Truck (trk-C pkg2 po-C)

Table 5.4: Learning *static reject* rules for the action "Unload-Airplane."

	time	*obj*	*airplane*	*loc*	*c*	*l*
+	2	pkg1	plane	apt-A	A	po-C
+	3	pkg1	plane	apt-B	B	po-C
+	4	pkg1	plane	apt-B	B	po-C
+	4	pkg2	plane	apt-B	B	po-C
–	5	pkg1	plane	apt-C	C	po-C
–	5	pkg2	plane	apt-C	C	po-C

positive example because all of its preconditions hold at time 2, *e.g.*, (in pkg1 plane) and (at plane apt-A), but it does not appear in the plan. On the other hand, Unload-Airplane (pkg1 plane apt-C) at time 5 is a negative example because it appears in the plan at time 5. The complete set of positive and negative examples are shown in the first five columns in Table 5.4.

Following the learning procedure outlined above, since no literal with the maximum possible gain can be found, two determinate literals (in-city loc c) and (goal (at obj l)) are first added to the rule, and they introduce two new variables c and l (whose types are "city" and "location" respectively). The bindings of c and l for each example are shown in the last two columns in Table 5.4.

In the next iteration, a literal $\neg$(in-city l c) is found to give the highest possible gain and is added to the rule. Now the rule covers only positive examples and none of negative examples. Therefore the procedure terminates with the rule:

```
(:wffctrl unload-airplane
 :scope
  (forall (?obj) (obj ?obj)
    (forall (?airplane) (airplane ?airplane)
      (forall (?loc) (location ?loc)
        (exists (?bbva) (in-city ?loc ?bbva)
          (exists (?bbvb) (goal (at ?obj ?bbvb))
            (in-city ?bbvb ?bbva))))))
 :precondition
  (and (at ?airplane ?loc)
       (in ?obj ?airplane))
 :effect
  (next (at ?obj ?loc))
)
```

Figure 5.4: Learned *static reject* rule for the action Unload-Airplane output in extended PDDL.

$$\neg\,\text{Unload-Airplane}\,(obj\ \ airplane\ \ loc) \leftarrow$$
$$(\texttt{in-city}\ loc\ \ c) \wedge (\text{goal}\,(\texttt{at}\ obj\ \ l)) \wedge \neg(\texttt{in-city}\ l\ \ c)$$

The above rule can be read as "Do not unload a package o from an airplane p at an airport a if a is not in the same city as o's goal location."

The learned *static reject* rule for action Unload-Airplane is then generated in the extended PDDL control following the conversion procedure described in Section 5.1.6. Variable obj, $airplane$, and loc are classified as dynamic variables and c and l are static variables. Figure 5.4 shows the learned control output in the extended PDDL control form.

Table 5.5: Learning time (in seconds) and number of rules acquired.

domain	# training problems	learning time	# rules learned
logistics	10	40.97 (22.5)	12 (11)
briefcase	3	1.79 (0.11)	4 (4)
grid	6	19.7 (7.41)	8 (8)
gripper	2	0.54 (0.22)	3 (3)
mystery	6	9.75 (9.01)	2 (2)
tireworld	5	3.34 (1.25)	11 (10)

It is worth noting how the above rule captures the concept of "a package that is not in its goal city," which is the crucial part of forming control rules in the logistics domain and was automatically acquired by our system. In other recent work on learning control knowledge for planning – for example, the learning system in Estlin and Mooney (1996) which blends EBL and inductive learning – similar concepts can only be learned by introducing hand-coded background knowledge that explicitly specifies which locations are in the same city.

5.3 Experimental Results

We have performed an empirical evaluation of our approach on a set of planning domains (logistics, grid, gripper, and mystery) from the 1998 Planning Systems Competition (AIPS 1998), as well as the briefcase and tireworld domain from the PDDL (McDermott, 1998) distribution. All experiments were run on a 450Mhz Sparc 420R.

Table 5.5 summarizes the learning time and number of control rules acquired for each domain. Mystery training problems are from the AIPS 1998 competition. Tireworld training problems are from PDDL distribution. All other training problems are randomly generated small instances. Learning time includes time to both generate and verify rules. Numbers shown in brackets are learning time and number of rules learned without backtracking. In general, our learning times are very short compared to other speed-up learning systems, which typically take from several minutes to several hours to generate a good set of rules. It is also observed that learning with backtracking is indeed able to acquire more control rules for certain domains without increasing the learning time significantly.

In the following, we will present in more details the kinds of rules that can be acquired for each domain and how much system performance improvement can be achieved by learned control rules.

5.3.1 Logistics Domain

As noted earlier, the logistics domain has become a particularly popular benchmark for recent work in planning. Our training set consists of ten randomly generated problems and their optimal plans that could be easily found by the BLACKBOX planner. The following control rules were generated by our learning system after learning on the training problems:

1. *static reject* - Do not load a package onto a truck if the package is at the goal.

2. *static reject* - Do not load a package onto a truck at an airport if the package's goal location is located in a different city.

3. *static reject* - Do not unload a package from a truck at a location if the location is not the package's goal and not an airport.

4. *static reject* - Do not unload a package from a truck at a location if the location is in the same city as the package's goal location.

5. *static reject* - Do not load a package onto an airplane at an airport if the location is in the same city as the package's goal location.

6. *static reject* - Do not unload a package from an airplane at an airport if the airport is not in the same city as the package's goal location.

7. *dynamic select* - Unload a package from a truck at the package's destination.

8. *dynamic select* - Unload a package from a truck if the current location is an airport and it is not in the same city as the package's goal location.

9. *dynamic select* - Unload a package from an airplane if the current airport is in the same city as the package's goal location.

10. *dynamic reject* - Do not move a truck if there is a package in the truck and the current location is an airport and is the package's goal location.

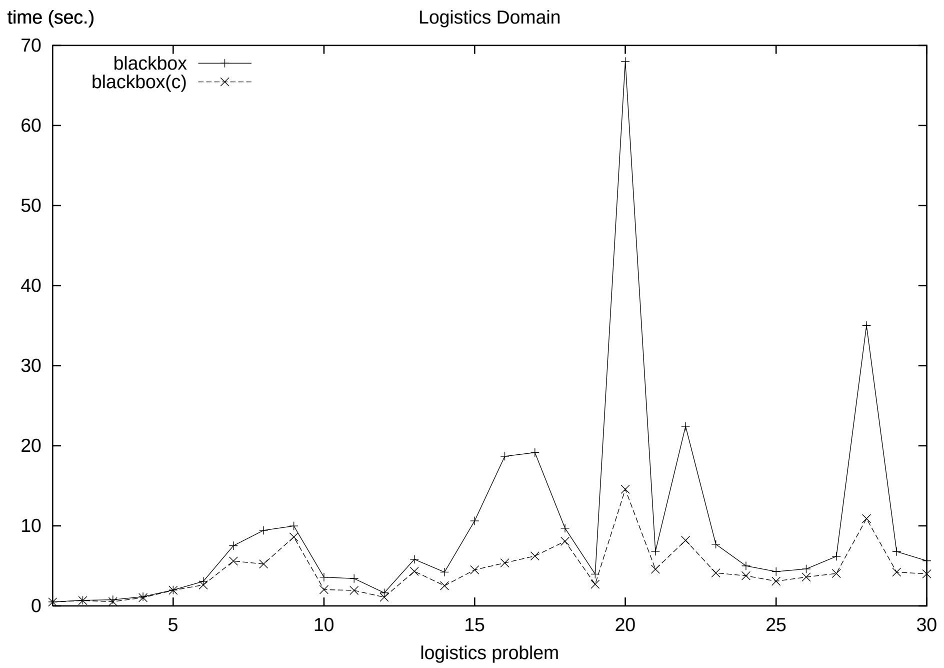

Figure 5.5: Logistics run time for BLACKBOX without and with (c) learned control rules from the BLACKBOX distribution.

11. *dynamic reject* - Do not move a truck if there is a package in the truck and the current location is not an airport and is the package's goal location.

12. *dynamic reject* - Do not fly an airplane if there is a package at the current airport and the package's goal location is in the different city.

Appendix C shows the above learned control rules generated in extended PDDL format. It is apparent from a comparison of the generated rules with the hand-coded control rules in Appendix B that the concept of user-defined predicate "`in-wrong-city`" is correctly captured and imbedded in the rule body.

In order to measure the effectiveness of the learned control rules, we perform an experiment on the 30 logistics problems in the BLACKBOX distribution. BLACKBOX runs with the new chaff SAT solver (Moskewicz *et al.*, 2001), which will be the default SAT solver for BLACKBOX throughout the rest of chapter unless another solver is identified. The run time for BLACKBOX without and with control are shown in Figure 5.5. It is apparent that run time for BLACKBOX is reduced by the use of learned control rules, especially on the more difficult problems. To see this, we generate 30 more difficult problems and perform a similar experiment. In the results shown in Figure 5.6, it is obvious that the speedup is more significant. For example, the run time for problem 14 with control is 99.87 times faster than the run time without control.

It is interesting to compare the rules learned by our system with the hand-coded ones. We consider the hand-coded rules used in Huang *et al.*(1999) and in TLPLAN (Bacchus and Kabanza, 2000).

We found that all the static reject rules used in TLPLAN are correctly acquired by our learning system. In addition, our system learnt several useful dynamic select rules that were not used in TLPLAN. For example, our system learnt the dynamic select rule 7: "Unload a package from a truck at the package's destination." This latter occured because TLPLAN is a purely sequential planner (unlike BLACKBOX, GRAPHPLAN, *etc.*) and so cannot make use of rules that would select for a set of actions of equal priority to be executed simultaneously. (Instead, it would be necessary to write a much more complex

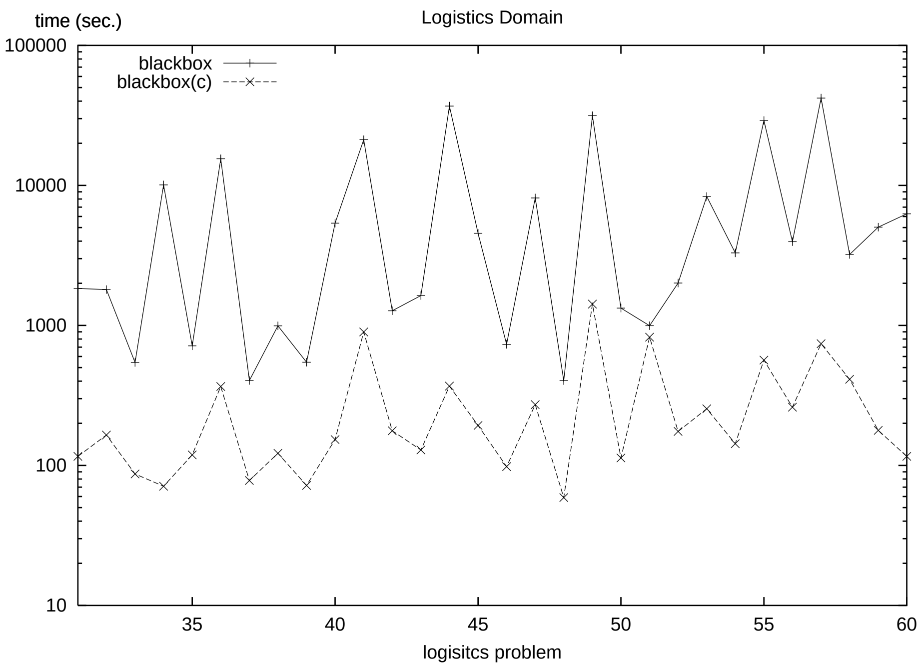

Figure 5.6: BLACKBOX run time on randomly generated hard logistics problems.

select rule that enforced some arbitrary temporal ordering between the unloads).

Our system did fail, however, to learn some of the dynamic reject rules used in TLPLAN. By carefully investigating the learning process, we discovered that two dynamic reject rules that are not output in the final set of control rules are actually learned during the learning phase but are pruned later by the verification module:

13. *dynamic reject* - Do not move an airplane if there is a package at the current airport whose goal is at another city's airport.

14. *dynamic reject* - Do not move an airplane if there is a package at the current airport whose goal is in another city.

Table 5.6: Comparison between BLACKBOX and TLPLAN with different control rule sets.

BLACKBOX(c) runs with rules 1 to 12.
BLACKBOX(c)$_{+13+14}$ runs with rule 1 to 12 and adds rule 13 and 14.
TLPLAN runs with original control rule formula.
TLPLAN$_{-14}$ runs without rule 14.
Times are in seconds. '–' means the problem cannot be solved in 4 hours.

Problem	BLACKBOX(c)	BLACKBOX(c)$_{+13+14}$	TLPLAN$_{-14}$	TLPLAN
log-a	1.03	1.11	–	0.33
log-b	1.97	2.40	0.26	0.27
log-c	2.63	2.52	4550	0.43
log-d	4.04	4.42	3.89	1.40

The above rules are discarded because when there are more than two airplanes but only one package that needs to be moved at the same airport, the above rules will force all airplanes to stay at the airport, which will prevent the planner from finding the optimal (the shortest parallel length) plans in certain circumstances. That is, only one airplane, not all, needs to stay and load the package, while the other airplanes can fly to other airports to deliver other packages. As a result, rules 13 and 14 are discarded, because they are not consistent with two of the training problems that include a similar scenario in all of their optimal plans.

Nevertheless, rule 14 is employed in TLPLAN (note that rule 13 is a subset of rule 14). We conduct an experiment to evaluate the effectiveness of using rule 13 and 14 for both BLACKBOX and TLPLAN on four traditional logistics problems. The result is shown in Table 5.6.

Surprisingly, the performance of BLACKBOX is not raised after adding rule 13 and 14 (remember that using rule 13 and 14 also excludes BLACKBOX from finding optimal plans on certain problem instances as described above). In fact, it runs more slowly on three out of four logistics problems. The reason for the slowdown is probably the cost of the extra added constraints. In other words, the overhead of adding extra clauses introduced by rule 13 and 14 into the CNF is higher than the gained unit propagations for the SAT solver. Another surprising result is that, without rule 14, the performance of the TLPLAN is hindered greatly on certain problems. TLPLAN takes more than 75 minutes to find a plan for logistics-c and cannot find a plan for logistics-a within 4 hours without the help of rule 14. On ther other hand, both problems can be solved by TLPLAN in less than one second with rule 14.

The above experiment shows that the effectiveness of control rules may vary for different planning algorithms and systems. The utility problem (Minton, 1988b) of how state-of-the-art planners can benefit from control knowledge should open up another interesting research direction.

5.3.2 Briefcase Domain

The task in the briefcase domain is to use a briefcase to carry and transport items between different locations. Our learning system obtains the following control rules from three randomly generated training problems:

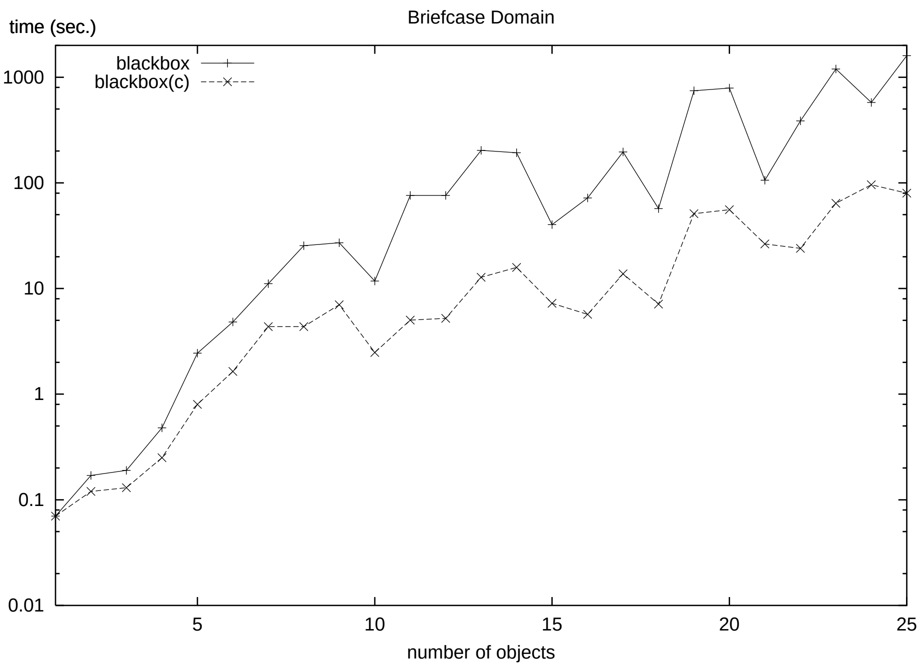

Figure 5.7: Briefcase run time.

1. *static reject* - Do not put an item into the briefcase if the item is at the goal.

2. *static reject* - Do not take an item out of the briefcase at a non-goal location.

3. *dynamic select* - Put an item into the briefcase if the item is not at the goal.

4. *dynamic select* - Take an item out of the briefcase at the item's destination.

Figure 5.7 shows the run time for BLACKBOX with and without learned control on 25 randomly generated problems. The number of objects in each problem ranged from 1 to 25 and the number of locations was fixed at seven for all problems. It is apparent that the run time is greatly reduced when BLACKBOX runs with learned control rules.

5.3.3 Grid Domain

Problems in the grid domain include an $N \times N$ grid and robots that can move between connected cells in the grid. Each cell in the grid can be open or locked. Only open adjacent cells can be reached by a robot. Each cell may also contain keys that can be used to unlock a cell if the locked cell and key have the same shape. The robots can pick up a key, put down a key, or pick a key and let loose a key at the same time. The task in the grid domain is to use the robots to send keys to their goal cell.

Control rules learned by our system over five problems are as follows:

- *static reject* - Do not put down a key at a non-goal cell.
- *static reject* - Do not put down a key at a goal's neighbor cell.
- *static reject* - Do not pick up a key at the key's goal cell.
- *static reject* - Do not pick up a key k_1 and let loose a key k_2 if k_1 is at the goal cell.
- *static reject* - Do not pick up a key k_1 and let loose a key k_2 if k_2 is not at the goal cell.

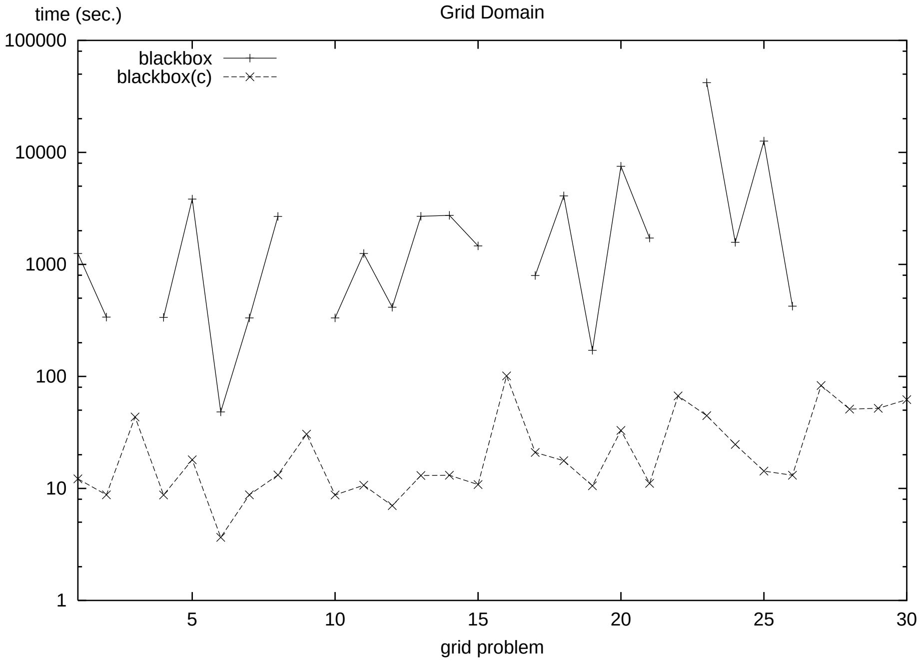

Figure 5.8: Grid domain run time.

We generated 30 more difficult grid problems and ran BLACKBOX with and without the above learned control rules set. The results are shown in Figure 5.8. Problems that cannot be solved by BLACKBOX within the allowed time limit are ssigned no run time in the figure. It is obvious that the run times for BLACKBOX with learned control is significantly improved. In fact, BLACKBOX with learned control rules is able to find plans for all problems in less than 2 minutes while several problems cannot be solved by BLACKBOX without control.

5.3.4 Gripper Domain

The gripper domain consists of a robot that has two arms and can move between rooms. The robot can use its arms to pick up or drop

Table 5.7: Gripper Domain: BLACKBOXwithout and with (c) learned control knowledge.
,

Problem	Time Step	BLACKBOX	BLACKBOX(c)
prob01	7	0.17	0.10
prob02	11	1.75	0.57
prob03	15	97	1.33
prob04	19	–	1.59
prob05	23	–	169
prob06	27	–	6.85

balls. The task is to transport balls between rooms by using the robot.

Our learning system is able to learn the following control rules from two training problems taken from AIPS 1998:

- *static reject* - Do not pick up a ball if the ball is at the goal room.
- *static reject* - Do not drop a ball if the robot is not at the ball's goal room.
- *dynamic select* - Drop a ball if the robot is at the ball's goal room.

Table 5.7 shows the BLACKBOX run times with and without the above rules on some problems from AIPS98. Again, several hard instances that cannot be solved by BLACKBOX without control can now be solved in a very short time by BLACKBOX with control.

5.3.5 Mystery Domain

The task in the mystery domain is to transport packages between networked locations. Only one kind of vehicle, which can move between

Table 5.8: BLACKBOX without and with (c) learned control knowledge on mystery problems.

Times are given in seconds.
'–' in time step: problem has no solution.
'–' in run time: problem cannot be solved in 4 hours.

Problem	Time Step	BLACKBOX	BLACKBOX(c)
p1	5	0.17	0.14
p2	5	6.51	1.66
p3	4	0.68	0.51
p4	–	1.88	0.81
p5	–	17.8	1.79
p6	9	84	13.3
p7	–	1.79	0.48
p8	–	88	6.66
p9	5	1.64	0.64
p10	8	–	29.7
p11	7	0.96	0.41
p12	–	1.40	0.86
p13	8	121	7.88
p14	10	–	62
p15	6	23.8	8.28
p16	–	5.24	1.99
p17	4	3.47	2.13
p18	–	19.9	7.25
p19	6	7.78	2.37
p20	7	25.8	11.8
p21	–	57	2.78
p22	–	302	20.9
p23	–	52	7.38
p24	–	150	8.58
p25	4	0.16	0.15
p26	6	1.88	1.17
p27	4	0.69	0.49
p28	7	0.78	0.36
p29	4	0.59	0.35
p30	6	5.98	2.79

two connected locations, is used to transport packages. Furthermore, each vehicle has limited space for accommodating packages and each location has limited fuel. Vehicles consume fuel when moving between connected locations and can only be refueled with the amount of available fuel at current location.

The following control rules are learned by our system:

- *static reject* - Do not load a package when it is at the goal.
- *static reject* - Do not unload a package when it is not at the goal.

Although not many control rules were acquired from learning, these two rules are actually the only ones that can be written easily by hand.

Table 5.8 shows the run time on problems used in the AIPS 1998 planning competition. Problems with '–' indicated in the time step mean that the problems have no solution and the run times for these problems are the time used to prove that no valid plans exist for them. The two learned control rules can still improve BLACKBOX's performance on all problems. They also help to find a plan for problems p10 and p14, which cannot be solved by BLACKBOX without control.

5.3.6 Tireworld Domain

In tje tireworld domain, the main task is to replace and inflate tires. In order to accomplish the tire replacement job, the necessary tires and tools need to be retrieved from a container.

Table 5.9: BLACKBOX without and with (c) learned control knowledge on tireworld problems.

Problem	# wheels	Time Step	BLACKBOX	BLACKBOX(c)
p1	1	12	0.23	0.22
p2	2	18	1.22	0.97
p3	3	24	6.50	4.67
p4	4	30	124	50.7
p5	5	36	8776	123

Problem p5 is solved by satz with preset plan length and its run time is averaged over 10 runs.

Our learning system generates the following control rules from the problems included in the PDDL distribution:

- *static reject* - Do not remove a wheel if the wheel is at the goal hub.
- *static reject* - Do not put on a wheel at a non-goal hub.
- *static reject* - Do not fetch a wheel if it is not intact.
- *static reject* - Do not put away a wheel into a container if it is intact.
- *static reject* - Do not put away an object into a container if its goal is not in the container.
- *dynamic select* - Fetch an intact wheel from a container immediately.

- *dynamic select* - Remove a wheel immediately after it has been loosened.
- *dynamic select* - Put on a wheel immediately at its goal hub.
- *dynamic select* - Inflate a deflated wheel immediately.
- *dynamic select* - Open a container immediately if is unlocked.
- *dynamic reject* - Do not close a container if there is an item in it whose goal is not in the container.

Table 5.9 shows the run time for BLACKBOX on tireworld problems. It is apparent that the performance on BLACKBOX with control rules is greatly improved. For example, the run time for problem p5 is 73 times faster compared to the run time without control.

5.4 Plan Completeness with Learned Control

In general, domain-specific control knowledge raises planning system performance by reducing the search space the systems explore. This reduction is along the lines of the general focus in planning on eliminating or avoiding search as much as possible. This same kind of approach has been applied in many real-world applications as well. However, in spite of the benefit from control knowledge, it is possible for certain plans to be pruned by the use of incorrect or inappropriate control. Existing optimal plans can also be discarded due to incorrect control. In the worst case, all plans are pruned and no solution can be found. This latter, of course, is not a desirable behavior. Ideally, the system performance should increase without lose of valuable information in the search space. Therefore, whether the plan completeness is preserved or not with the use of learned control knowledge is an important issue.

As mentioned previously, our learning approach is based on the optimality criteria of the BLACKBOX planner and the justification module it uses, which is to minimize parallel plan length and then to minimize sequential length, and the learning system is built to operationalize these optimality criteria. Therefore, in principal, planners can benefit from the control rules learned by our system to avoid non-optimal plans in the search space while preserving the optimal parallel plan completeness. This is a very important feature because for certain domains, *e.g.*, the logistics domain, there exist efficient polynomial algorithms to find non-optimal plans but finding optimal plans is difficult. Therefore, control learned by our system has the potential to greatly improve the time required to find optimal plans. However, this potential also implies that perhaps fewer control rules will be generated, since certain rules that seem useful would be discarded because they do not preserve the parallel plan completeness.

For instance, in Section 5.3.1 we have shown that rule 14 in the logistics domain is crucial for TLPLAN to find solutions on logistics problems (see Table 5.6). However, we also illustrate a simple scenario in which the same rule will prevent BLACKBOX from finding the optimal parallel plans on certain problems. In fact, we have found that rule 14 was learned by our system but pruned during the verification phase, becase it is not consistent with certain training problems. In contrast, other learning approaches or planning systems that use control knowledge tend to minimize the sequential plan length or reduce search space as much as possible. In addition, the plan completeness issue has seldom been mentioned or has been ignored by previous work on other learning systems.

Nevertheless, unlike EBL approaches (Mitchell *et al.*, 1986; Dejong and Mooney, 1986), the rules generated by our learning system are not necessarily logical consequences of the domain. In particular, if the training set is too small, it is possible in principle for the system to learn rules that exclude all solutions to a particular problem instance, although this did not occur in our experiments. On the other hand, our learning approach requires no domain theory as needed in EBL systems. In fact, creating a good and correct domain theory is an non-trivial task. An incomplete or incorrect domain theory often leads to incorrect, useless, or non-existent control rules.

5.5 Related Work

The learning of planning control knowledge by traditional planners has been long studied; for example, work on systems such as the EBL and case-based learning mechanism built upon PRODIGY planning system (Minton, 1988a; Veloso, 1992). The key idea in these approaches is for the planner to learn search heuristics in order to increase the efficiency of the plan search. Below we will briefly describe some of the main approaches.

In the STRIPS planning system (Fikes and Nilsson, 1971), macro-operators were introduced by a technique called "lifting." A macro-operator is a sequence of actions that can be treated as a single operator. Macro-operators can improve the planner's performance because the planner does not need to reason about the intermediate steps encapsulated by the macro-operators. More sophisticated macro-operator techniques have since been proposed (Korf, 1985; Minton, 1990). The automatic abstraction of macro-operators can be viewed as a form of learning. Unfortunately, macro-operators appear to be effective only in certain rather narrow domains.

The PRODIGY/EBL system (Minton, 1988a) learns control rules via an explanation-based learning (EBL) technique. Three kinds of control rules are used in the PRODIGY/EBL system: *select, reject,* and *prefer* rules. A *select* rule guides the search engine as to which action to select next at a particular point in the plan; a *reject* rule tells it what kind of actions to avoid; and a prefer rule says which actions to give preference to when exploring several possible actions. It has been shown that PRODIGY/EBL with control rules outperforms PRODIGY without any control rules. Moreover, the performance improvement of the learned rules is comparable to that of hand-coded rules used in their system. Some advantages of using EBL methods are that rules can be learned from a single example and learned rules can be proved correct.

Another contribution of PRODIGY/EBL is that it pays attention to the *utility* problem of control rules (Minton, 1988b). The utility captures the overall effectiveness of a rule. Rules with low or negative utility are deleted in their system. If a control rule is rarely applicable or expensive to apply, then the cost of having the rule in the system may actually outweigh its savings; the rule therefore has a negative utility. While we have not yet seen this happen with the hand-coded rules used in TLPLAN and the control rule learned from the BLACKBOX planner, we are still interested in the problem of quantizing the utility of learned control rules.

However, a complete and correct domain theory is necessary for performing explanation-based learning. An incorrect or incomplete domain theory often leads to incorrect or zero control acquisition. Furthermore, creating a complete and correct domain theory is not an easy task; in fact, it is a time-consuming process. For example, as pointed out in Minton (1988a), approximately one man-year was spent writing, rewriting, and debugging the schemas used by the PRODIGY/EBL system.

The GRASSHOPPER system (Zeckie and Zukerman, 1998) learns *prefer* rules for PRODIGY using an inductive learning technique. An inductive learning approach does not require a complete domain theory and has the potential to find more effective rules by learning from more than one example at a time. It has been shown that PRODIGY with rules learned by GRASSHOPPER often outperforms PRODIGY with rules learned by PRODIGY/EBL. However, the problem within the inductive learning method is that the correctness of learned rules cannot be formally verified.

The control rules acquired in the systems above are closely tied to the plan search strategy and are generally represented in a procedural planner-dependent format. Therefore, it is practically infeasible to share control information between planners. Another practical problem is that the systems tend to learn too many rules or rules that are too specific or overly general. This problem is of course a general issue in machine learning, where the challenge is often to find the correct level of generalization. In addition, even with control rules, the performance of these "traditional" planners does not approach that of GRAPHPLAN or BLACKBOX, even without any control knowledge.

In addition to the earlier work mentioned above, there is other recent works on speed-up learning that, like ours, combines aspects of supervised learning and rule induction. The systems of Khardon (1999) and of Martin and Geffner (2000) try to learn very large sets of production-style rules that replace, rather than improve, a search engine, and the systems require thousands of training examples and long training times.

The SCOPE system (Estlin and Mooney, 1996, 1997) uses a combination of EBL and inductive logic programming (ILP) techniques to acquire effective control rules for the UCPOP planner. It has been shown that SCOPE achieved a higher success rate and better speedups than the UCPOP+EBL (Kambhampati *et al.*, 1996) approach. The work by Estlin and Mooney (1996, 1997) differs from ours in that it requires the user to explicitly supply background knowledge to the learner in the form of additional predicates and "relational clichés" (Silverstein and Pazzani, 1991). In addition, the control rules learned by SCOPE are specific to the UCPOP planner and can not be exploited by other planning systems without significant modification. Recently Kambhampati (1999) has shown how explanation-based learning techniques can be applied to GRAPHPLAN, but his work does not include any attempt by the system to learn high-level, declarative rules.

Chapter 6

Conclusions and Future Work

Deterministic state-space planning is a hard combinatorial problem that arises in tasks such as robot control, software verification, and logistics scheduling. Research on recent advances in general purpose planning systems has shown such systems to be capable of outperforming traditional planning systems. Recent planning systems such as constraint-based and heuristic search based planners can effectively synthesize plans consisting of several hundreds of actions, a task far beyond the capability of traditional planning systems. The work in this area has greatly benefited from a common set of benchmark problems, which has been an important driving force behind the rapid improvements we have seen in recent years. Many of these benchmark domains consist of essentially single-agent planning tasks. However, many real-world planning domains, such as airplane scheduling and robot control, involve settings with multiple agents.

We have shown how it is possible to adapt planning benchmark problems to incorporate participation requirements for multi-agent instances. Our proposal for encoding participation requirements into the problem instances has the advantage of maintaining the basic plan search nature of the benchmark problems. Furthermore, no changes need to be made to the planners themselves and no special post-processing of the plans is required.

Using the modified multi-agent benchmark problems, we have given a detailed comparison of constraint-based and heuristic search based planning techniques. Our results show that these planners have complementary strengths. More precisely, heuristic planners are very efficient on certain multi-agent domains, such as the elevator task, but are less efficient on domains with more intricate multi-agent interactions. Constraint-based planners, on the other hand, are more robust over the range of multi-agent problems, but are less efficient than heuristic planners on the domains where a forward-chaining search approach with heuristics excels. The complementary nature of these planners suggests that perhaps a hybrid approach would lead to yet a more powerful class of planners.

Nevertheless, both recent heuristic search and constraint-based planning systems are domain independent and rely only on efficient algorithms and heuristics. Domain-dependent control knowledge, on the other hand, has the potential to significantly increase the performance of the new planners. We have shown it is possible to obtain the benefits of the control rules in terms of efficiency, without paying the price of reduced parallelism, by incorporation of control rules into the BLACKBOX planner. In a sense, the solvers in BLACKBOX still tackle the combinatorial aspect of the task but the extra constraints provide substantial additional pruning of the search space.

We implemented the system by enhancing the BLACKBOX planner with extended planning language for temporal logic control rules, which are translated into additional propositional clauses. We also showed that a subset of the control rules can be handled by direct pruning of the planning graph. Our experimental results show a speedup due to the search control of up to two orders of magnitude or more on our problem domains. In sum, our work demonstrates that declarative control knowledge can be used effectively in constraint-based planners without loss of plan quality.

A fascinating research direction is the possible use of machine learning techniques for automatic acquisition of control knowledge by new planners. Although research in machine learning has long studied the problem of creating efficient planners through the learning of domain-specific control knowledge, the techniques that have been developed are not yet in widespread use in practical planning systems. Consequently, we believe that the field of speed-up learning for new planning systems is poised to undergo a resurgence.

In this dissertation, we present the first positive results on acquisition and use of such high-level, purely declarative control knowledge for constraint-based planning using machine learning techniques. Our approach blends aspects of explanation-based learning, supervised learned, and inductive logic programming. We have introduced a new heuristic method for extracting training examples from plans generated by the BLACKBOX planner and have built a rule learning system based on an inductive learning programming approach. Our system can learn useful control rules in a variety of benchmark domains from small training problems. Only a few number of rules are needed to reduce solution times by two orders of magnitude or more on larger problems and training times are short. Unlike previous work on learning control for planning, control rules learned by our system are purely declarative in temporal logic form and thus are not specific to the details of underlying planning algorithms. Therefore, ideally it is feasible to export the learned rules to other planning systems with none or little modifications. In addition, control knowledge learned by our system has the feature to preserve parallel plan completeness. Overall, our learning architecture is simple and modular, and initial empirical evaluation on established benchmarks has shown that control knowledge can be learned that is on a par with that created by hand.

Our ongoing and future work includes a more careful and detailed empirical evaluation of the approach: an investigation of the learning and use of more expressive control rule languages through the inclusion of more modal operators, such as “next” and “previous,” or by allowing a rule to contain explicit constants; and a study of ways to create training problems that will most aid learning. In particular, we are investigating an *active learning* approach, in which the current set of learned control rules is used to influence the creation of the next training problem. We are also working on applying rule pruning techniques to remove redundancy or simplify rules (Quinlan, 1987; Cohen, 1995). Finally, we are investigating how to exploit EBL approaches to improve the correctness of learned control rules, a contribution that would create a more powerful learning system for planning.

Appendix A

Logistics Domain Definition in PDDL

```
(define (domain logistics-strips)
  (:requirements :strips)
  (:predicates  (obj ?obj)
                (truck ?truck)
                (location ?loc)
                (airplane ?airplane)
                (city ?city)
                (airport ?airport)
                (at ?obj ?loc)
                (in ?obj ?obj)
                (in-city ?obj ?city))


  (:action Load-Truck
   :parameters (?obj ?truck ?loc)
   :precondition (and (obj ?obj)
                      (truck ?truck)
                      (location ?loc)
                      (at ?truck ?loc)
                      (at ?obj ?loc))
   :effect (and (not (at ?obj ?loc))
                (in ?obj ?truck)))
```

```
(:action Load-Airplane
 :parameters (?obj ?airplane ?loc)
 :precondition (and (obj ?obj)
                    (airplane ?airplane)
                    (location ?loc)
                    (at ?obj ?loc)
                    (at ?airplane ?loc))
 :effect (and (not (at ?obj ?loc))
              (in ?obj ?airplane)))


(:action Unload-Truck
 :parameters (?obj ?truck ?loc)
 :precondition (and (obj ?obj)
                    (truck ?truck)
                    (location ?loc)
                    (at ?truck ?loc)
                    (in ?obj ?truck))
 :effect (and (not (in ?obj ?truck))
              (at ?obj ?loc)))


(:action Unload-Airplane
 :parameters (?obj ?airplane ?loc)
 :precondition (and (obj ?obj)
                    (airplane ?airplane)
                    (location ?loc)
                    (in ?obj ?airplane)
                    (at ?airplane ?loc))
 :effect (and (not (in ?obj ?airplane))
              (at ?obj ?loc)))
```

```
  (:action Drive-Truck
   :parameters (?truck ?loc-from ?loc-to ?city)
   :precondition (and (truck ?truck)
                      (location ?loc-from)
                      (location ?loc-to) (city ?city)
                      (at ?truck ?loc-from)
                      (in-city ?loc-from ?city)
                      (in-city ?loc-to ?city))
   :effect (and (not (at ?truck ?loc-from))
                (at ?truck ?loc-to)))


  (:action Fly-Airplane
   :parameters (?airplane ?loc-from ?loc-to)
   :precondition (and (airplane ?airplane)
                      (airport ?loc-from)
                      (airport ?loc-to)
                      (at ?airplane ?loc-from))
   :effect (and (not (at ?airplane ?loc-from))
                (at ?airplane ?loc-to)))
)
```

Appendix B

Hand-Coded Logistics Control Rules

```
(define (control logcontrol)
(:domain logistics)

  (:defpredicate in-wrong-city
   :parameters (?obj ?loc)
   :body (exists (?goal-loc) (goal (at ?obj ?goal-loc))
           (exists (?city) (in-city ?loc ?city)
             (not (in-city ?goal-loc ?city)))))


  (:action Load-Truck
   :exclude (goal (at ?obj ?loc)))


  (:action Load-Truck
   :exclude (and (in-wrong-city ?obj ?loc)
                 (airport ?loc)))


  (:action Unload-Truck
   :exclude (and (in-wrong-city ?obj ?loc)
                 (not (airport ?loc))))
```

```
(:action Unload-Truck
 :exclude (and (not (in-wrong-city ?obj ?loc))
               (not (goal (at ?obj ?loc)))))

(:action Load-Airplane
 :exclude (not (in-wrong-city ?obj ?loc)))

(:action Unload-Airplane
 :exclude (in-wrong-city ?obj ?loc))

(:wffctrl w1
 :scope (forall (?trk) (truck ?trk)
          (forall (?obj) (obj ?obj)
            (forall (?loc) (location ?loc)
              (and (in-wrong-city ?obj ?loc)
                   (not (airport ?loc))))))
 :precondition (and (at ?trk ?loc)
                    (at ?obj ?loc))
 :effect (next (at ?trk ?loc)))

(:wffctrl w2
 :scope (forall (?trk) (truck ?trk)
          (forall (?obj) (obj ?obj)
            (forall (?loc) (airport ?loc)
              (in-wrong-city ?obj ?loc))))
 :precondition (and (at ?trk ?loc)
                    (in ?obj ?trk))
 :effect (next (at ?trk ?loc)))
```

```
  (:wffctrl w3
   :scope (forall (?trk) (truck ?trk)
            (forall (?obj) (obj ?obj)
              (forall (?loc) (location ?loc)
                (goal (?obj ?loc)))))
   :precondition (and (at ?trk ?loc)
                      (in ?obj ?trk))
   :effect (next (at ?trk ?loc)))


  (:wffctrl w4
   :scope (forall (?pln) (airplane ?pln)
            (forall (?obj) (obj ?obj)
              (forall (?loc) (airport ?loc)
                (not (in-wrong-city ?obj ?loc)))))
   :precondition (and (at ?pln ?loc)
                      (in ?obj ?pln))
   :effect (next (at ?pln ?loc)))
)
```

Appendix C

Learned Logistics Control Rules

```
(define (control learned-control)
        (:domain logistics-strips)

  (:action load-truck
   :exclude
    (goal (at ?obj ?loc))
  )

  (:action load-truck
   :exclude
    (exists (?bbva) (in-city ?loc ?bbva)
      (exists (?bbvb) (goal (at ?obj ?bbvb))
        (exists (?bbvc) (in-city ?bbvb ?bbvc)
          (and (airport ?loc)
               (not (= ?bbvc ?bbva))))))
  )

  (:action unload-truck
   :exclude
    (and (not (goal (at ?obj ?loc)))
         (not (airport ?loc)))
  )
```

```
(:action unload-truck
 :exclude
  (exists (?bbva) (in-city ?loc ?bbva)
    (exists (?bbvb) (goal (at ?obj ?bbvb))
      (and (not (goal (at ?obj ?loc)))
           (in-city ?bbvb ?bbva))))
)

(:action load-airplane
 :exclude
  (exists (?bbva) (in-city ?loc ?bbva)
    (exists (?bbvb) (goal (at ?obj ?bbvb))
      (exists (?bbvc) (in-city ?bbvb ?bbvc)
        (= ?bbvc ?bbva))))
)

(:action unload-airplane
 :exclude
  (exists (?bbva) (in-city ?loc ?bbva)
    (exists (?bbvb) (goal (at ?obj ?bbvb))
      (exists (?bbvc) (in-city ?bbvb ?bbvc)
        (not (= ?bbvc ?bbva)))))
)

(:wffctrl unload-truck
 :scope
  (forall (?obj) (obj ?obj)
    (forall (?truck) (truck ?truck)
      (forall (?loc) (location ?loc)
        (goal (at ?obj ?loc)))))
 :precondition
  (and (in ?obj ?truck)
       (at ?truck ?loc))
 :effect
  (next (at ?obj ?loc))
)
```

```
(:wffctrl unload-truck
 :scope
  (forall (?obj) (obj ?obj)
    (forall (?truck) (truck ?truck)
      (forall (?loc) (location ?loc)
        (exists (?bbva) (in-city ?loc ?bbva)
          (exists (?bbvb) (goal (at ?obj ?bbvb))
            (and (airport ?loc)
                 (not (in-city ?bbvb ?bbva))))))))
 :precondition
  (and (in ?obj ?truck)
       (at ?truck ?loc))
 :effect
  (next (at ?obj ?loc))
)

(:wffctrl unload-airplane
 :scope
  (forall (?obj) (obj ?obj)
    (forall (?airplane) (airplane ?airplane)
      (forall (?loc) (location ?loc)
        (exists (?bbva) (in-city ?loc ?bbva)
          (exists (?bbvb) (goal (at ?obj ?bbvb))
            (in-city ?bbvb ?bbva))))))
 :precondition
  (and (at ?airplane ?loc)
       (in ?obj ?airplane))
 :effect
  (next (at ?obj ?loc))
)
```

```
(:wffctrl drive-truck
 :scope
  (forall (?truck) (truck ?truck)
    (forall (?loc-from) (location ?loc-from)
      (forall (?loc-to) (location ?loc-to)
        (forall (?bbva) (obj ?bbva)
          (and (airport ?loc-from)
               (goal (at ?bbva ?loc-from)))))))
 :precondition
  (and (in ?bbva ?truck)
       (at ?truck ?loc-from))
 :effect
  (next (at ?truck ?loc-from))
)
```

```
(:wffctrl drive-truck
 :scope
  (forall (?truck) (truck ?truck)
    (forall (?loc-from) (location ?loc-from)
      (forall (?loc-to) (location ?loc-to)
        (forall (?bbva) (obj ?bbva)
          (and (airport ?loc-to)
               (goal (at ?bbva ?loc-from)))))))
 :precondition
  (and (in ?bbva ?truck)
       (at ?truck ?loc))
 :effect
  (next (at ?obj ?loc))
)
```

```
  (:wffctrl drive-truck
   :scope
    (forall (?truck) (truck ?truck)
      (forall (?loc-from) (location ?loc-from)
        (forall (?loc-to) (location ?loc-to)
          (forall (?bbva) (obj ?bbva)
            (and (airport ?loc-to)
                 (goal (at ?bbva ?loc-from)))))))
   :precondition
    (and (in ?bbva ?truck)
         (at ?truck ?loc-from))
   :effect
    (next (at ?truck ?loc-from))
  )


  (:wffctrl fly-airplane
   :scope
    (forall (?airplane) (airplane ?airplane)
      (forall (?loc-from) (airport ?loc-from)
        (forall (?loc-to) (airport ?loc-to)
          (forall (?bbva) (obj ?bbva)
            (goal (at ?bbva ?loc-from))))))
   :precondition
    (and (in ?bbva ?airplane)
         (at ?airplane ?loc-from))
   :effect
   (next (at ?airplane ?loc-from))
  )

)
```

Bibliography

Aarup, M., Arentoft, M. M., Parrod, Y., Stader, J., and Stokes, I. (1994). OPTIMUM-AIV: A knowledge-based planning and scheduling system for spacecraft AIV. In Fox, M. and Zweben, M., editors. *Knowledge Base Scheduling.* Morgan kaufmann. San Mateo, California.

Aler, R., Borrajo, D., and Isasi, P. (1988). Genetic programming and deductive-inductive learning: A multistrategy approach. *Proceedings of the Fifteenth International Conference on Machine Learning* (pp. pages 10–18). Madison, WI: Morgan Kaufmann.

Bacchus, F. and Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, *116*.

Bacchus, F. and Nau, D. (2001). The AIPS'00 planning systems competition. *AI Magazine*, *22(3)*,47–56.

Backstrom, C. (1992). *Computational complexity of reasoning about plans*, Ph.D. Thesis, Linkoping University, Linkoping, Sweden.

Baioletti, M. , Marcugini, S., and Milani, A. (1998). C-SATPlan: a SATPlan-based tool for planning with constraints. AIPS-98 Workshop on Planning as Combinatorial Search, Pittsburgh, PA.

Barrett, A., Golden, K., Penberthy, J.S., and Weld, D.S. (1993). UCPOP user's manual (version 2.0). Technical Report 93-09-06. Department of Computer Science and Engineering, University of Washington.

Beek, P. and Chen, X. (1999). CPlan: a constraint programming approach to planning. *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, Florida.

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

Blum, A. and Furst, M. L. (1995). Fast planning through planning graph analysis. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (pp. 1636–1642). Montreal, Canada.

Bonet, B. and Geffner, H. (1997). A fast and robust action selection mechanism for planning. *Proceedings of the Fourteenth National Conference on Artificial Intelligence* (pp. 714–719). Providence, RI.

Borrajo, D. and Veloso, M. M. (1997) Lazy incremental learning of control knowledge for efficiently obtaining quality plans. In D. Aha (Ed.), *Lazy learning*. Norwell, MA: Kluwer Academic Publishers.

Bylander, T. (1991). Complexity results for planning. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence* (pp. 274–279). Sydney, Australia: Morgan Kaufmann.

Carbonell, J., Knoblock, C., & Minton, S. (1990). PRODIGY: An integrated architecture for planning and learning. In K. VanLehn (Ed.), *Architectures for intelligence*. Hillsdale, NJ: Lawrence Erlbaum.

Clark, P. and Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, *3*, 261–283.

Cohen, W. W. (1995). Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning*. Lakte Tahoe, CA.

Crawford, C. (1984). Compact: A fast simplifier of Boolean formulas. Available via Crawford's web page.

Culberson, J. (1997). Technical Report TR97-02, Department of Computer Science, University of Alberta, Edmonton, Canada.

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

desJardins, M.E., Durfee E. H., Ortiz, C.L. and Wolverton, M.J. (1999). A survey of research in distributed, continual planning. *AI Magazine*, *20*, 13–22.

Ernst, M.D., Millstein, T.D., and Weld, D.S. (1997). Automatic SAT-compilation of planning problems. *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*. Nagoya, Japan.

Erol, K., Nau, D.S., and Subrahmanian, V.S. (1992). On the complexity of domain-independent planning. *Proceedings of the tenth National Conference on Artificial Intelligence* (pp. 381–386).

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

Estlin T.A., and Mooney, R.J. (1997). Learning to improve both efficiency and quality of planning. *IJCAI-97*, Nagoya, Japan.

Etzioni, Oren (1993). Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62(2), 255-302.

Fikes, R. E., and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 5(2): 189-208.

Fink, E. and Yang, Q. (1992). Formalizing plan justifications. *Proceedings of the Nineth Conference of the Canadian Society for Computational Studies of Intelligence* (pp. 9–14).

Fox, M. and Long, D. (1998). The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, *9*, 367–421.

Fox, M. and Long, D. (1999). Efficient implementation of plan graph in STAN. *Journal of Artificial Intelligence Research*, *10*, 87–115.

Gary, M.R. and Johnson, D.S. (1979). *Computers and Intractability: A Guide to the Theory of NP-complete.*

Gerevini, A. and Schubert, L. (1998). Inferring state constraints for domain-independent planning. *Proc AAAI-98*, Madison, WI.

Glover, F. and Laguna, M. (1993). Tabu search. In Reeves, C.R., Ed., *Modern heuristics for combinatorial problems*, Oxford, GB: Blackwell Scientific, 70–150.

Helmert, M. (2001). On the complexity of planning in transportation domains. *Proceedings of the sixth European Conference on Planning.*

Hoffmann, J. (2000). A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. *Proceedings of the twelfth International Symposium on Methodologies for Intelligent Systems*, Charlotte, NC, USA, October 2000.

Hoffmann, J. (2001). Local search topology in planning benchmarks: an empirical analysis. *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, Seattle, WA, USA.

Huang, Y.-C., Selman, B., and Kautz, H. (1999). Control knowledge in planning: benefits and tradeoffs *Proceedings of the Sixteenth National Conference on Artificial Intelligence* (pp. 511–517). Orlando, FL.

Johnson, M. D. and Adrof, H.-M. (1992). Scheduling with neural networks: the case of the Hubble space telescope. *Computers & Operations Research,* 19(3–4):209–240.

Kambhampati, S., Katukam, S., and Qu Y. (1996). Failure driven dynamic search control for partial order planners: An explanation based approach. *Artificial Intelligence*, *88*, 253–315.

Kambhampati, S. (1997). Challenges in bridging plan synthesis paradigms. *Proc. IJCAI-97*, Nagoya, Japan.

Kambhampati, S. (1999). Improving graphplan's search with EBL & DDB techniques. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*. Stockholm, Sweden: Morgan Kaufmann.

Kautz, H. and Selman, B. (1992). Planning as satisfiability. *Proceedings of the Tenth European Conference on Artificial Intelligence* (pp. 359–363). Vienna, Austria: John Wiley & Sons.

Kautz, H. and Selman, B. (1996). Pushing the envelope: planning, propositional logic, and stochastic search. *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (pp. 1194-1201). Portland, OR: AAAI Press.

Kautz, H. and Selman, B. (1998). The role of domain-specific axioms in the planning as satisfiability framework. *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*. Pittsburgh, PA: AAAI Press.

Kautz, H. and Selman, B. (1999). Unifying SAT-based and graph-based planning. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence* (pp. 318–325). Stockholm, Sweden: Morgan Kaufmann.

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

Knoblock, C. (1994). Automatically generating abstractions for planning. *Artificial Intelligence* 68(2).

Koehler, J., Nebel, B., Hoffmann, J., and Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. *Proc. 4th European Conf. on Planning*, S. Steel, ed., vol. 1248 of *LNAI*, Springer.

Koehler, J. and Hoffmann, J. (2000). On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, *12*, 338–386.

Korf, R.E. (1985). Macro-operator: A weak method for learning. *Artificial Intelligence*, *26*, 35–77.

Leckie, C. and Zukerman, I. (1998). Learning search control rules for planning. *Artificial Intelligence*, *101*, 63–98.

Long, D. *et al.* (2000). The AIPS-98 planning competition. *AI Magazine*, *21(2)*, 13-33.

Martin, M. and Geffner, H. (2000). Learning generalized policies in planning using concept languages. *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning.* Breckenridge, CO.

McCarthy, J. and Hayes, P.J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., Michie, D., and Swann, M., editors, *Machine Intelligence 4,* pages 463–502.

McDermott, D., *et al.* (1998). PDDL — the planning domain definition language. Department of Computer Science, Yale University, New Haven.

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

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

Minton, S. (1990). Issues in the design of operator composition systems. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 304–312. San Mateo, CA.

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

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L. and Malik, S. (2001). Chaff: engineering an efficient SAT solver Proc. DAC-01. Las Vegas.

Muggleton, S., and Feng, C. (1992). Efficient induction of logic programs. In S. Muggleton (Ed.), *Inductive logic programming*. London: Academic Press Limited.

Pednault, E. (1989). ADL: Exploring the middle ground between STRIPS and the situation calculus. *KR-89*, 324–332.

Penberthy, J.S. and Weld, D.S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of KR-92*, pages 103–114.

Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, *9*, 268–299.

Quinlan, J. R. (1987). Simplifying decision trees. *International Journal of Man-Machine Studies*, 27:221-234.

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

Quinlan, J. R. (1996). Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, *5*, 139–161.

Selman, B., Kautz, H., and Cohen, B. (1996). Local Search Strategies for Satisfiability Testing, *Dimacs Series in Discr. Math. and Theoretical Computer Science*, Vol. 26, 521–532.

Silverstein, G. and Pazzani, M. J. (1991). Relational clichés: Constraining constructive induction during relational learning. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 432–436). Evanston, IL: Morgan Kaufmann.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning,* 8(3–4):257–277.

Veloso, M. (1992). *Learning by analogical reasoning in general problem solving*. Doctoral dissertation, Department of Computer Science, Carnegie Mellon University, Pittsburgh.

Weld, S. D. (1999). Recent advances in AI planning. *AI Magazine*, *20*, 93–123.