

An Inductive Approach to Learning Search Control Rules for Planning

Christopher Leckie

Artificial Intelligence Systems Section
Telecom Australia Research Laboratories
770 Blackburn Rd, Clayton
Victoria 3168, AUSTRALIA
Email: c.leckie@trl.oz.au

Ingrid Zukerman

Department of Computer Science
Monash University
Clayton, Victoria 3168
AUSTRALIA
Email: ingrid@bruce.cs.monash.edu.au

Abstract

One method for reducing the time required for plan generation is to learn search control rules from experience. Most of the recent work in learning search control knowledge has concentrated on explanation-based learning techniques. An alternative approach is to use inductive learning. An inductive approach does not require a complete and tractable domain theory and has the potential to create more effective rules by learning from more than one example at a time. In this paper, we describe Grasshopper, an inductive system that learns search control rules for a typical plan generation system. We also provide an empirical evaluation of Grasshopper by comparing it with an existing explanation-based learning system.

1 Introduction

The process of generating a plan to satisfy a set of goals can be modelled as a sequence of decisions at different levels of detail. These decisions include choosing which partial solution to complete from the available alternatives, which unsatisfied goal to solve next, which planning operator to use in order to attain this goal, and which objects in the domain to use to instantiate this operator. A domain independent planning system uses weak search methods to make these decisions by trial and error. However, by making use of problem solving experience it should be possible to learn domain specific search control rules which reduce inefficiencies in the default search strategy. In this paper, we describe an inductive learning technique for improving the speed of a classical planning system. This technique was implemented in a system called Grasshopper¹, which learns search control rules to guide the behaviour of a STRIPS-type planner [Fikes and Nilsson, 1971] within a given domain.

In recent years, explanation-based learning (EBL) techniques have become the standard approach for this kind of application. For example, systems such as PRODIGY/EBL [Minton, 1988], SOAR [Laird *et al.*, 1985] and STATIC [Etzioni, 1991] all use some variation on EBL in order to increase the efficiency of a planner. However, there can be potential drawbacks to using an explanation-based approach. First, it is a knowledge-intensive method. A domain theory

is used to prove that a training example is an instance of a particular type of decision. This can pose problems in domains where it is difficult to construct a complete and computationally tractable domain theory [Chien, 1989]. Second, explanation-based approaches learn from a single example at a time. This makes it harder to determine the relative importance of each part of a newly learned rule. As a result, additional domain rules or heuristics may need to be encoded to ensure that the new search control rules are stored in an operational form. Otherwise, the cost of testing the applicability of a new rule on future problems may outweigh its savings. This is known as the *utility problem* [Minton, 1988].

Grasshopper overcomes these problems by using an inductive approach to learning search control rules for goal, operator and variable binding decisions. Given a number of training instances, the system looks for sets of similar decisions that could form the basis of new search control rules. It then generates preconditions for these rules by characterising the problem solving contexts in which each of these decisions occurred. This approach obviates the need to refer to a domain theory, and also improves the utility of new search control rules by taking into consideration more than one training problem when learning a new rule.

In the next section, we describe in more detail the decisions that arise during planning. In Section 3, we describe our inductive approach. In Section 4, we present the results of testing Grasshopper on several domains using the PRODIGY planner [Minton, 1988], and compare these results with the performance of the PRODIGY/EBL system. These results demonstrate that our inductive approach can produce significant reductions in the time required to generate a plan compared to EBL. In Section 5, we discuss these results and outline issues warranting further research. In Section 6, we outline related work on learning search control knowledge. Throughout this paper, we follow the STRIPS convention for representing plans and use the "blocks world" domain to illustrate our approach.

2. Search Decisions in Planning

A planning problem is defined by a goal state to be achieved, a starting state from which the goal must be achieved, and a set of operators that can be used to achieve the goal. There are four operators in the standard representation of the blocks world domain [Nilsson, 1980], namely, PUT-DOWN,

¹ Grasshopper was the name given to the student in the television series "Kung Fu".

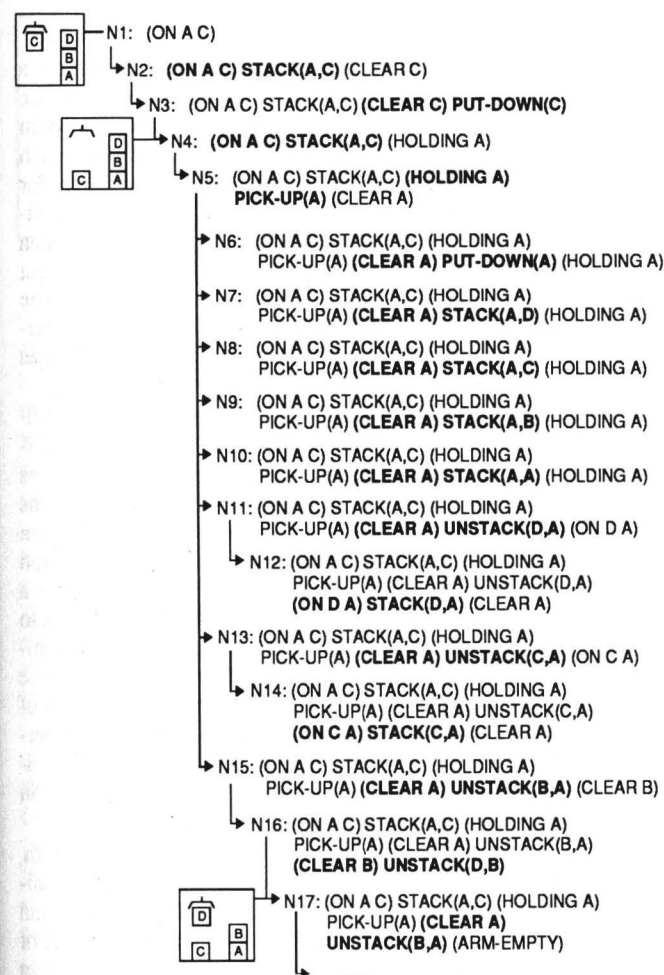


Figure 1: Example of a Plan Search Tree

PICK-UP, STACK and UNSTACK. Figure 1 shows an example of a plan search tree for the top-level goal (ON A C). Each node in the search tree contains a description of the problem state at that node, a goal stack, and a list of the children of that node. As mentioned in the Introduction, at each successive stage in generating a plan the following decisions need to be made.

(1) **Node Decisions:** choice of a node containing goals that have yet to be satisfied. In this example, we use the depth-first strategy.

(2) **Goal Decisions:** choice of a goal to be satisfied. In this example, it is always the latest subgoal. However, in problems where the given goal is a conjunction of simple goals, the choice is important if we are to avoid harmful interactions between the plans for each top-level goal.

(3) **Operator Decisions:** choice of an operator for the current goal. For example, at node N5 in Figure 1 there are three different operators which may achieve the goal (CLEAR A): PUT-DOWN, STACK and UNSTACK.

(4) **Binding Decisions:** choice of instantiations for the current operator. These are generated by matching the pre-conditions of each operator against the current problem state. In situations where there is more than one match, there will be several alternative instantiations of the same operator. For example, UNSTACK is instantiated three times at node N5 in Figure 1.

Once all of these decisions have been made, the planner will attempt to generate a new node in the plan search tree. The opportunity for learning in plan generation lies in improving this decision-making process in order to avoid unsuccessful choices.

3. Grasshopper: An Inductive Approach

Grasshopper is designed to analyse a trace of a planning system's behaviour during search, and suggest modifications to the planner's search strategy that will improve its performance in the future. Each time the planner solves a problem, the resulting plan search tree is passed to Grasshopper for analysis. Grasshopper then extracts from the plan search tree any interesting decisions that appeared during the search. These decisions are stored and then used as the basis for learning new search control rules. These search control rules are merged with the default rules of the domain to produce a new search strategy that can be used to guide the planner on subsequent problems.

Like PRODIGY/EBL [Minton, 1988], Grasshopper has been designed to learn about goal, operator and binding decisions. Grasshopper does not learn about node decisions because any wasted nodes can be explained in terms of the other three kinds of decisions. There are five main stages in Grasshopper:

(1) First, the *Decision Extraction* stage extracts interesting planning decisions from a new plan search tree.

(2) Next, the *Decision Clustering* stage groups these decisions according to the type of search control advice they provide.

(3) The *Choice Characterisation* stage then learns a unique description for each cluster of decisions. This description is used to recognise when the corresponding advice is applicable.

(4) The *Rule Generation* stage compiles the characterisation and advice from a cluster of decisions into a search control rule. This may result in a new rule, or an updated version of an existing rule.

(5) Finally, the *Utility Optimisation* stage estimates the utility of new rules, and decides whether they should be passed on to the planner.

In the following subsections, we describe the operation of each of these five stages. Since the same basic approach is used for the three kinds of planning decisions, we focus on learning about operator decisions, and describe any major deviations from this method where appropriate. For a more detailed description of Grasshopper, see [Leckie, 1993].

3.1 Decision Extraction

When Grasshopper is given a new plan search tree, its first task is to extract interesting decisions for learning. This is done by applying two basic guidelines: (1) do not attempt to learn from a decision if all of its choices have been doomed to failure by an incorrect choice at a higher level decision, and (2) do not attempt to learn from a decision if there was only one possible choice. In other words, Grasshopper is interested in decisions where there is more than one choice available, and one of those choices eventually led to a successful plan.

The first guideline forces Grasshopper to filter out nodes

that are not ancestors of the successful leaf node of the plan search tree. Given a successful leaf node for a plan, it is a simple matter to trace its ancestors back up to the root of the tree. We refer to the successful leaf node and its ancestors as the *successful nodes* of the plan. For example, in the problem in Figure 1, the successful nodes are {N1, N2, N3, N4, N5, N15, N16, N17, ...}. In contrast, nodes N6 to N14 contain no decisions with successful choices because they are all derived from incorrect operator or binding choices at node N5. Consequently, they are excluded by Grasshopper.

Grasshopper then applies the second guideline in order to find interesting examples of each kind of planning decision from each of the successful nodes in the plan. For example, there is only one possible operator choice for the goal (ON A C) at node N1 in Figure 1. Consequently, node N1 is excluded by the second guideline. In contrast, there are several possible operator choices for the goal CLEAR. Consequently, nodes N2, N5 and N15 are all extracted as examples of interesting operator decisions. The same procedure is used to extract examples of goal and binding decisions. Clearly, there are no interesting examples of goal interactions in Figure 1 because the problem involves only one top-level goal.

3.2 Decision Clustering

Once Grasshopper has extracted the interesting examples of each kind of decision, it needs to cluster these decisions according to the type of advice they provide. Interesting decisions are organised into a data structure known as a *choice table*. Separate choice tables of each kind of planning decision are maintained in Grasshopper as a long-term memory. A choice table clusters decisions using two levels of indexes. At the first level, decisions are indexed according to their *decision type*. For example, operator decisions have the same decision type if they all involve the same kind of goal. Consequently, nodes N2, N5 and N15 from Figure 1 would all be indexed under the decision type CLEAR. At the second level, decisions of the same decision type are sub-indexed according to their *choice type*. For example, operator decisions have the same choice type if they all have the same operator as their successful choice. Consequently, the decision type for the goal CLEAR would be sub-indexed so that node N2 is stored under the choice type PUT-DOWN, while N5 and N15 are stored under the choice type UNSTACK.

The same procedure is followed for goal and binding decisions. In the case of goal decisions, the decision type is defined by the predicates of the interacting pair of goals, while the choice type is the goal that avoids harmful interactions when satisfied first. In the case of binding decisions, the decision type is defined by the goal predicate and operator type for the decision, while the choice type denotes whether or not the chosen list of bindings was successful. Grasshopper has to store unsuccessful binding decisions so that it has a supply of counter-examples for learning. This is not necessary for goal and operator decisions, where the counter-examples of a choice type are implicit in the other choice types belonging to the same decision type.

The choice types indexed under the same decision type represent competing choices for the same type of decision. Clustering decisions in this way structures the learning problem for Grasshopper, since each choice type represents an opportunity for a search control rule.

3.3 Choice Characterisation

Each successful choice type in a choice table represents a target concept for learning. Before Grasshopper can encode a rule for such a choice type, it needs to characterise when that advice is applicable. Sometimes the choice is clear, such as when one type of goal is always preferred to another. For other decision types the correct choice may not be so obvious, such as when two successful choice types have been observed for the same decision type. In this case, the correct choice depends on the context of the current decision. The role of the Choice Characterisation stage is to learn a generalised description of the decision context that discriminates the correct choice for a decision from any distractors.

For example, we saw in the last subsection that the decision type CLEAR has at least two choice types, i.e., PUT-DOWN and UNSTACK. In other words, in some situations PUT-DOWN is the correct choice, while in other situations the planner should use UNSTACK. The difference between these two situations is the problem state where each decision was made. Consequently, if Grasshopper is to generate a search control rule for each of these choice types, it needs to encode into the rule the problem state features that discriminate one situation from the other. Specifically, each choice type must be discriminated from every other choice type of the same decision type. This involves searching for a generalised problem state description that matches all the decisions in the target choice type, and none of the decisions in any of the competing choice types.

We have adapted the FOIL learning algorithm [Quinlan, 1990] to perform this task. FOIL learns an intensional definition for a target relation that is given as a set of positive and negative examples. This definition is represented as a set of Horn clauses that define the target relation in terms of a set of background relations. These Horn clauses are constructed so that between them they match all the positive examples of the target relation without matching any of the negative examples. FOIL uses an information-based heuristic to guide a greedy search for the relations to be used in the body of each clause. Interested readers are referred to [Quinlan, 1990] for a detailed description of the operation of FOIL.

In our application, the target relation is the choice type to be characterised and the background relations encode the context of the decision. Consider the problem of characterising the choice of the operator UNSTACK for the goal CLEAR. The target relation contains two positive examples, corresponding to the operator decisions at nodes N5 and N15, and one negative example, corresponding to the node N2. The background relations encode the goal predicate and the current problem state for each of these examples. Using the FOIL algorithm, Grasshopper finds that the feature (ON ?Y ?X) is unique to decisions where UNSTACK is successful for the goal (CLEAR ?X). Similarly, Grasshopper finds that the feature (HOLDING ?X) is unique to decisions where PUT-DOWN is successful for this goal.

Each clause that Grasshopper learns in this way forms the basis for a new search control rule. The main advantage of using a learning algorithm such as FOIL is that it provides an efficient method for characterising each choice type. In addition, it is able to learn about choice types that require a disjunctive description, i.e., choice types that cannot be characterised by a single clause.

If the current node is ?N
 and the current goal is (CLEAR ?X)
 and UNSTACK is a candidate operator choice at ?N
 and ?O is a candidate operator choice at ?N
 and ?O is not equal to UNSTACK
 and (ON ?Y ?X) holds at ?N
 then prefer operator UNSTACK to ?O

Figure 2: Stylised Example of a Search Control Rule

3.4 Rule Generation

Once interesting decisions have been clustered and characterised, the Rule Generation stage can begin. Rules are required only for those choice types that have been confused with alternative choices of the same decision type. The resulting rules may then be used by the planner to avoid making the wrong choice in subsequent problem solving.

For each choice type that requires a rule, Grasshopper needs to encode the antecedent and the consequent of the rule. The antecedent encodes the context of the decision. This is used by the planner to recognise whether the rule is applicable at a given node in a plan search tree. It specifies the decision type, the competing candidate choices, and the characteristic features of the successful choice type. The consequent of a rule encodes the advice contained in the choice type. One approach is to encode this advice so that the successful choice is the only available option for the planner. However, this may lead to the correct choice being pruned from the search space if the characterisation used in the rule is too general. Consequently, Grasshopper encodes the advice of a rule as a preference for the successful choice type over the other candidate choices. Figure 2 illustrates an example of such a search control rule.

3.5 Utility Optimisation

Any choice type in the choice table that contained erroneous choices before the successful choice was found may warrant a rule. Such a rule saves the cost of repeating these mistakes in the future. However, there is an overhead to adding new rules. Every time a node is expanded, we incur the cost of testing this rule. If the savings of the rule are small, or if the rule is seldom applicable, then the utility of the rule to the planner is low. Hence, we need to estimate the utility of new rules before they are added to the planner.

We can calculate a measure of this utility using the information collected from the training problems as they appear in the choice table. We begin by calculating the cost of solving each training problem. This is just the cost of creating the nodes in the search tree. A search control rule would reduce this cost by the number of wasted nodes that it would save from the training examples, e.g., nodes N6-N14 in Figure 1. In return, there would be an additional overhead of matching the rule at every node in the tree, excluding the nodes that have been saved. When this process is repeated for every choice type in all three choice tables, the expression for the total choice cost is as follows:

$$Cost = \sum_{i=1}^t [C_c n_i + (1-x_i) C_c w_i + x_i C_m \sum_{j=1}^t n_j + x_i C_m \sum_{j=1}^t (1-x_j) w_j]$$

where:

t = total number of choice types,

n_i = number of decisions represented by choice type i ,
 w_i = number of wasted nodes in decisions of choice type i ,
 $x_i \in \{0, 1\}$ 1 if the rules for choice type i are included,
 0 otherwise,
 C_c = average cost of creating a node, and
 C_m = average cost of matching a rule to a node.

Note that in order to simplify this calculation, we have used average values for the costs of matching rules and creating nodes.

We use this approximate expression to help quantify the effect that new candidate rules would have on the training problems. The problem of determining which rules to remember can thus be resolved by finding the combination of rules that minimise this cost function. Unfortunately, however, in order to calculate the additional overhead of any particular rule, we need to know which other rules have been included. As a result, the optimisation of this cost function is a non-linear 0-1 integer programming problem [Taha, 1975]. There are 2^t possible alternatives in the search space for such a problem. However, in all of the domains we have encountered, t is sufficiently small to allow us to perform the search using explicit enumeration.

4. Experimental Results

In this section, we examine the performance of Grasshopper in practice, using the PRODIGY planner as our test-bed. The performance of Grasshopper was compared with two alternative strategies. The first alternative was the default search strategy used by PRODIGY. The other alternative was the search strategy learned by PRODIGY/EBL, which has become a standard example of the state of the art in EBL. Our aim is to compare the utility of our inductive learning approach to that of EBL.

Our inductive algorithm has been tested in four domains, three of which we present in this section. The first two domains are the versions of the blocks world and STRIPS world used in [Minton, 1988]. In each case, a training phase of 50 problems was followed by a testing phase of 100 problems. The problems were randomly generated and varied in complexity from one to five goals. The training problems were filtered to exclude examples that were too hard or too easy. The third domain implements a parser for strings of the ambiguous grammar shown in Figure 3. This simple domain is included to illustrate the advantages of inductive learning when a problematic domain theory is involved. In this case, a training phase of 10 problems was sufficient to learn appropriate search control rules. It was followed by a testing phase of 50 problems.

In the first and third experiments, the length of the search on each problem was limited to a maximum of either 300 CPU seconds or 2500 nodes, whichever came first. In the STRIPS world domain, the CPU bound was raised to 750 seconds so that at least one strategy was able to solve all of the test problems. The tests were run on a Sun Sparcstation

S → aX	S → aY	S → aZ
X → bX ₁	X ₁ → cX ₁	X ₁ → x
Y → bY ₁	Y ₁ → cY ₁	Y ₁ → y
Z → bZ ₁	Z ₁ → cZ ₁	Z ₁ → z

Figure 3: Grammar Rules for the Parser Domain

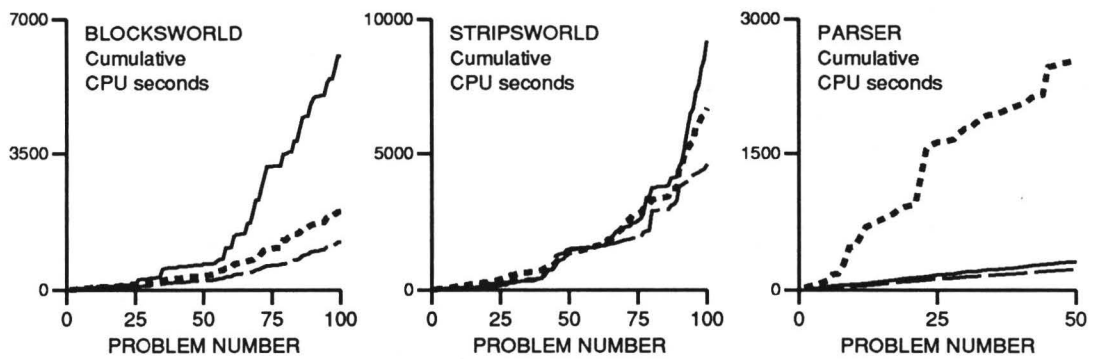


Figure 4: Test Results (— no rules, - - - PRODIGY/EBL, — Grasshopper)

IPC using Sun Common Lisp 3.0. It should be noted that PRODIGY/EBL was able to create search control rules for node decisions in addition to goal, operator and binding decisions. Further, in the blocks world and STRIPS world domains, PRODIGY/EBL had access to explicitly encoded domain axioms that could be used to help simplify the explanations that were generated.

The results of these tests are plotted in Figure 4. These graphs show the cumulative cost of the search in terms of the CPU time spent on each problem. According to these results, the rules learned by Grasshopper were most effective at reducing the total search time required in all of the domains.

In terms of the total number of nodes expanded, there was little difference between Grasshopper and PRODIGY/EBL in the blocks world and parser domains. In the STRIPS world domain, PRODIGY/EBL produced greater savings in the number of nodes expanded, as shown in Table 1. However, PRODIGY/EBL required more rules than Grasshopper to achieve these savings. This means that these savings were offset by the additional computational cost of having to test a larger set of rules at the nodes of each search tree. Thus, the utility of the rules learned using the inductive approach is significantly higher than that of the rules learned using the explanation-based approach.

5. Discussion and Further Work

Our results have demonstrated the advantages of inductive learning in addressing the utility problem. By learning from more than one example at a time, it is possible to avoid learning rules that are overly specialised to individual examples. This raises several issues regarding the role of a domain theory in learning search control knowledge. An advantage of using a domain theory in explanation-based learning is that it provides a strong bias in the search for a set of features that characterise a target concept. This reduces the number of training examples required for learning. Consequently, it is often a criticism of inductive learning techniques that they may require a large number of examples before they can

detect any significant patterns. However, Grasshopper was able to achieve its level of performance using the same number of training examples used by PRODIGY/EBL. The only noticeable difference was that PRODIGY/EBL usually required less time for learning than Grasshopper. The largest difference observed was a factor of approximately two in the STRIPS world domain.

Another advantage of domain theories is that it is possible to verify the correctness of the rules that are learned. For example, some characterisations learned by Grasshopper were initially too specific or too general. However, most of these errors were eliminated by the end of the training phase, since Grasshopper relearns a rule if it sees a new example. Those errors that persisted still led to rules that were useful part of the time, even though they gave the wrong advice on some occasions. In contrast, correctness does not guarantee utility. For example, although the rules learned by PRODIGY/EBL in the parser domain were correct, they were far too complex to be useful because of the recursive nature of the domain theory. Grasshopper was able to learn much simpler rules because it tested for the feature that characterised each string, namely, whether it contained the letter "x", "y" or "z". Although this is an artificial domain, it illustrates the ability of inductive techniques to learn from heuristic features that are hard to explain or not mentioned in the domain theory. An interesting issue for further research is to study these kinds of domains in more detail.

Finally, our cost function was able to provide an efficient test of the utility of the rules learned by Grasshopper. In these three domains, all of the rules that were learned by Grasshopper were deemed to be useful and passed on to the planner. The performance of these rules in practice supports this decision. An issue for further testing is to repeat these tests using subsets of the final set of rules in order to determine whether there were any errors of commission.

6. Related Work

The current trend in learning search control knowledge to reduce the cost of planning has been towards EBL. So far, we have concentrated on PRODIGY/EBL. Other explanation-based systems that address the utility problem are SOAR [Tambe and Rosenbloom, 1989] and STATIC [Etzioni, 1991]. SOAR uses a technique known as "chunking" to learn search control knowledge from experience. It avoids the problem of expensive chunks by restricting the ways in which domains can be encoded. STATIC takes a different approach by learning search control rules from the domain

Strategy	Active Rules	Nodes Expanded	Unsolved Problems
Default	-	37174	7
Grasshopper	14	10145	0
PRODIGY/EBL	29	2708	1

Table 1: Results for the STRIPS World Domain

definition only, without referring to training examples. It avoids expensive rules by excluding rules that require a recursive proof. Both of these approaches to intractable domain theories have the effect of restricting the range of concepts that can be learned. In contrast, Grasshopper can learn heuristic features that are not amenable to explanation. In addition, neither of these systems attempts to evaluate the utility of the knowledge that is learned, unlike Grasshopper and PRODIGY/EBL.

There are a number of systems that have used inductive techniques to learn search control knowledge. Early systems, such as LEX [Mitchell *et al.*, 1981], SAGE [Langley, 1985] and UPL [Ohlsson, 1987] learned search control rules to guide problem-solvers that use production rules. Unlike Grasshopper, these systems were not designed to work with a planning system. Consequently, they did not attempt to learn about goal interactions. In addition, these systems were developed before the utility problem became a major issue. As a result, they did not investigate the utility of the knowledge learned using inductive methods. More recently, unsupervised inductive learning has been used in a system called DAEDALUS [Langley and Allen, 1991] to learn concept hierarchies that can guide a planner. These concept hierarchies are used to provide advice on planning decisions by recalling the choices that were made in similar situations in the past. In terms of utility, the process of matching a new decision against a concept hierarchy is a significant computational overhead. However, their main concern is with cognitive issues, rather than with the utility problem.

Finally, COMPOSER [Gratch and DeJong, 1991] uses a similar approach to our own for estimating the utility of rules in combination. Their approach is more elaborate, in that they model how the savings of a rule change when other rules are added. In addition, they use statistical tests to determine upper and lower bounds on the utility of a set of rules, and use a greedy strategy for selecting rules. The trade-off of using a more elaborate model is that COMPOSER may need more training examples than Grasshopper in order to satisfy the statistical tests. In addition, COMPOSER requires extra time for plan generation during the training phase so that it can measure the match cost of rules before they are added. An issue for further study is whether a more elaborate model such as this leads to significant improvements in utility, or whether there is a diminishing return due to increased training time in practice.

7. Conclusions

In this paper, we have addressed the problem of how to improve the speed of a planning system by learning from experience. Specifically, we have developed and tested an inductive learning algorithm which generates search control rules for goal, operator and binding decisions in planning. Unlike EBL methods, this approach can generate rules of high utility without the need for additional domain knowledge or heuristics to indicate which domain features are relevant to a rule. In addition, we have presented a new method for estimating the utility of learned rules by studying the effect of new rules in combination on the performance of the planner. We have demonstrated in three test domains that our approach can significantly reduce plan search time in comparison to an explanation-based system.

8. Acknowledgements

The permission of the Director, Telecom Research Laboratories, to publish this paper is hereby acknowledged. We are extremely grateful to the PRODIGY team at Carnegie Mellon University for providing us with a copy of their system.

9. References

- [Chien, 1989] S.A. Chien. Using and Refining Simplifications: Explanation-based Learning of Plans in Intractable Domains. In *IJCAI-89 Proceedings*, pp. 590-595.
- [Etzioni, 1991] O. Etzioni. STATIC: A Problem-Space Compiler for PRODIGY. In *AAAI-91 Proceedings*, pp. 533-540.
- [Fikes and Nilsson, 1971] R.E. Fikes and N.J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Artificial Intelligence 2*, pp. 189-208.
- [Gratch and DeJong, 1991] J. Gratch and G. DeJong. A Hybrid Approach to Guaranteed Effective Control Strategies. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 509-513.
- [Laird *et al.*, 1985] J.E. Laird, P.S. Rosenbloom and A. Newell. Chunking in SOAR: The Anatomy of a General Learning Mechanism. In *Machine Learning 1*(1), pp. 11-46.
- [Langley, 1985] P. Langley. Learning to Search: From Weak Methods to Domain-Specific Heuristics. In *Cognitive Science 9*, pp. 217-260.
- [Langley and Allen, 1991] P. Langley and J.A. Allen. The Acquisition of Human Planning Expertise. In *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 80-84.
- [Leckie, 1993] C.A. Leckie. Inductive Learning of Search Control Rules for Planning. Ph.D. Thesis, Dept. of Computer Science, Monash University, Australia.
- [Minton, 1988] S. Minton. Quantitative Results Concerning the Utility of Explanation-Based Learning. In *AAAI-88 Proceedings*, pp. 564-569.
- [Mitchell *et al.*, 1981] T.M. Mitchell, P.E. Utgoff, B. Nudel and R. Banerji. Learning Problem-Solving Heuristics Through Practice. In *IJCAI-81 Proceedings*, pp. 127-134.
- [Nilsson, 1980] N.J. Nilsson. *Principles of Artificial Intelligence*, Tioga Publishing Corporation.
- [Ohlsson, 1987] S. Ohlsson. Transfer of Training in Procedural Learning: A Matter of Conjectures and Refutations? In *Computational Models of Learning*, L. Bolc (ed.), Springer-Verlag, Berlin, pp. 55-88.
- [Quinlan, 1990] J.R. Quinlan. Learning Logical Definitions from Relations. In *Machine Learning 5*(3), pp. 239-266.
- [Taha, 1975] H.A. Taha. *Integer Programming. Theory, Applications, and Computations*, Academic Press.
- [Tambe and Rosenbloom, 1989] M. Tambe and P. Rosenbloom. Eliminating Expensive Chunks by Restricting Expressiveness. In *IJCAI-89 Proceedings*, pp. 731-737.