MODEL-BASED REFINEMENT OF SEARCH HEURISTICS

BY MICHAEL W. BARLEY

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

L. Steinberg

and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

May, 1996

ABSTRACT OF THE DISSERTATION

Model-Based Refinement of Search Heuristics

by Michael W. Barley

Dissertation Director: L. Steinberg

Search is one of the central problem-solving paradigms in Artificial Intelligence. Unfortunately, search can be quite expensive. Problem-solvers frequently use rejection heuristics to lower the cost of their searches. Unfortunately, these rejection heuristics can also prevent a problem-solver from solving a problem.

I describe a new learning algorithm, SHAPeS, that enables the learner to modify both rejection heuristics implemented explicitly as search control rules and those implemented implicitly by either being embedded within the program code or by being incorporated into the search space generators. The SHAPeS learning algorithm only requires a solution to be supplied rather than needing that solution's problem-solving trace. This frees the system from having to find the solution itself. The SHAPeS algorithm is generally applicable to search-based problem-solvers. I discuss both the conditions for the algorithm's correctness and its computational complexity.

I implemented a restricted version of SHAPeS, called **Bacall**, and in this thesis report on the results of experiments conducted to assess the utility of the **Bacall** learned modifications. The experiments show that not only can **Bacall** learned modifications extend a problem-solver coverage but can also enable it to run faster. The experiments also compare the utility of the **Bacall** learned modifications with the utility of current approaches to modifying rejection heuristics. The experiments show that

Bacall learned modifications can improve the problem-solver's performance more than comparable modifications learned by the alternative approaches.

Lastly, I identify three types of search control knowledge interactions that complicate the analysis of rejection heuristics' effects upon a problem-solver's coverage. These interactions cause modifications to a problem-solver's search control knowledge to have counter-intuitive effects upon the problem-solver's coverage. For example, removing rejection rules or specializing their preconditions can cause the problem-solver to become unable solve some problems. I also identify conditions that are sufficient to eliminate these types of interactions and that simplify the analysis of the effect of modifying a problem-solver's search control logic upon its coverage.

Acknowledgements

They say it takes a whole village to raise a child, I would say it takes at least as much to raise a PhD, especially when they've taken thirteen years to complete. So, unfortunately, I cannot acknowledge everyone who has helped so the following is necessarily only a partial list.

The fact that I did complete I attribute to three people: Lou Steinberg, Patricia Riddle, and Haym Hirsh. Lou has been my advisor for almost all of my graduate school career. He has been patient, supportive, and critical. He endured many a long silence, especially after I moved from New Jersey to Seattle, and must have wondered what I was doing during those periods. Even so, he continued to tell me that my research was important and to champion my continued existence as a graduate student at the department's annual culling of the graduate herd. This did not stop him from asking critical questions about my work. I don't remember ever leaving a meeting where his questions hadn't sharpened my understanding of my work.

Patricia has been a constant source of practical advice and suggestions on the practical side of doing a PhD. She has helped rein in many of my impulses to widen and deepen my research by reminding me that this is only a PhD dissertation and need not answer every possible question. She often called to my attention the fact that one can continue to pursue these questions after the dissertation is done. She read and reread my earliest drafts of my thesis; her comments often making it clear where I expected my readers to be clairvoyant. Patricia also helped focus the dissertation by her questions of "Do you have to include that?"

Haym has been a friend and a committee member for a number of years. He has been consistently supportive and even enthusiastic about my work. For someone who has been working on a topic for over ten years, it's easy to feel that everything that one has

done is intuitively obvious to the even the casual obverser. Even though enthusiasm is a prominent component of Haym's personality, he always made me feel that my research was important and worth continuing. Haym's comments on my work were often from angles I had not considered and which enriched my own perspective on the subject.

Marv Paull was another of my committee members whose questions often caused me to look at my work anew. While most of my committee specialized in AI, Marv was my outsider. He would ask how my work related to the broader field of computer science. He was interested in understanding whether what I was doing was simply repeating work done under a different name in a different part of computer science, or whether it was new and if so, how it contributed to the broader context. From Marv, I learned not to restrict my view to my specialization but to remember to look around and see what the other areas of computer science could contribute to my understanding of my problems.

Saul Amarel and Dana Nau are the remaining members of my committee. It is often easy to lose oneself in the details of one's work, Saul helped me to, at least occasionally, pull back and ask what was important about my work and why. Dana was my outside member, and while we only met together a few times, I found those times to be both instructive and enjoyable.

Tom Mitchell has influenced my graduate career in a number of ways. Happily, he reviewed my application to Rutgers and thought I could make a contribution. He was also instrumental in my obtaining a summer internship at Schlumberger's Research Lab, where I began my transition from a software developer to a researcher. The insight, that gradually evolved into my thesis, came from a conversation with Tom in 1987 at the AI conference that summer in Seattle.

Jack Mostow started my work on replay, which led from Bogart (my replay system) to Bogart (my search-heuristic refinement system). He asks interesting questions and is quick to grasp the heart of technical work. Back in 1990, it was Jack who first made me aware of the overlap between Neeraj Bhatnagar's work and my own. At the time that overlap was not obvious, my work was focussed on circumventing loss of domain coverage due to strong linearity's search bias, and my superficial reading of Neeraj's

work lead me to view his work as focussing on how to learn a form of goal protection. Jack's comment forced me to try to understand the similarities and differences between Neeraj's and my researches. That attempt enabled me to understand much better why my learning technique worked and how to generalize it to work for any search heuristic.

Without PRODIGY, I would have had to build yet another problem-solver that differed from everyone elses and which made comparisons difficult. For that, I thank Steve Minton. Steve has always been helpful in answering my questions about how PRODIGY's planner and learner worked.

My understanding of my work would never have proceeded so far without the work of Neeraj Bhatnagar and Soowon Lee. They have been generous in their discussions of their own systems and how they compare with mine.

During my time at Rutgers (and since then) I have benefitted from innumerable discussions about AI with John Bresina, William Cohen, Chun Liew, Stacy Marsella, Sridhar Mahadevan, and Prasad Tadepalli. Discussions with Mike Berman and Shiv Chaudhuri have broaden my awareness of and perspective on Computer Science.

During my first few years in Seattle, I worked as a consultant at the Boeing Advanced Technical Center, where I was free to pursue my thesis research and to talk to the research community there. Kish Sharma championed my contract there and made sure that I had whatever computing resources I needed. Art Murphy, Dave Purdon, and Keith Williamson formed my intellectual community those early years at Boeing and provided much needed feedback on my research.

As Boeing entered one of its recurring downswings, Kish was no longer able to renew my contract and I had to find another computing resource. Happily, Oren Etzioni stepped forward and designated me as a visiting researcher to the University of Washington. This allowed to use UW CS Department computing facilities to finish up my work. Since this took four years, this was quite a contribution on Oren's part. Especially, as Oren allowed me to store my data on his project's disk when the general disk storage allowance (10 Mb) proved to be about a tenth of what I needed. Aside from the computing resources, I also benefitted from my talks at the University of Washington with Oren, Steve Hanks, Dan Weld, and Tony Barrett.

Over the years I have had the good fortune to talk with a number of people about my work and have benefitted from their insights. Some of these people have been Rao Kambhampati, Craig Knoblock, Paul Rosenbloom, and Manuela Veloso.

Finally, my research could not have happened without the support, encouragement, and understanding of my family. My parents never suggested that it shouldn't take longer to get one's PhD than it did to complete all of one's pre-university education. They were always supportive. My wife and son made all this work worthwhile by being there when I would finally come home in the evenings and by forcing me to realize there was a lot more to life than school. Oh course, all the trips to Europe, South America, and Africa helped to do that too. Cindy Barley also encouraged my academic wanderings and introduced me to the cinema, history, and helped improve my writing skills.

Dedication

This is dedicated to Patricia J. Riddle, who, while she helped my graduate days last many more years than I had originally anticipated, made them infinitely richer than I could have imagined.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Problem

Search is one of the central problem-solving paradigms in Artificial Intelligence. Unfortunately, search can be quite expensive. In domains where this is so (for example, in domains containing NP-hard problems), problem-solvers frequently trade completeness of domain coverage for speed in solving those problems still covered by using heuristics to speed up their searches. There are two types of heuristics: preference heuristics and rejection heuristics. *Preference* heuristics determine the order the search space is explored by describing when one search branch should be tried before another. *Rejection* heuristics reduce the size of the search space by describing when a search branch should be rejected altogether. Because rejection heuristics actually reduce the size of the space to be searched, they can cause much more speed-up than their equivalent preference heuristics. However, this is at the expense of being able to cause the problem-solver to fail to solve some solvable problems. Because they offer both greater rewards and greater punishments, this thesis focuses on rejection heuristics.

Rejection heuristics can be viewed as a method for trading problem *coverage* (i.e., the range of problems the problem-solver can solve) for problem-solving speed (i.e., the inverse of the average amount of time spent attempting to solve a problem). Since any one user is unlikely to want to solve every solvable problem in a given domain, *completeness* (i.e., being able to solve all solvable problems) is not necessary for a problem-solver to be useful. Therefore making a problem-solver faster as long as it doesn't cause the user's problems to become unsolvable is an acceptable tradeoff. Unfortunately, it is not always possible to predict the effects of a new rejection heuristic upon the problem-solver's coverage. Therefore, when the problem-solver is given a

problem which its current rejection heuristics prevent it from solving, one would like the user and the problem-solver to interact so that the system can modify the appropriate rejection heuristics to be able to solve similar problems in the future. This is the main focus of this thesis.

The system should modify the rejection heuristic so that it no longer rejects a particular branch at a search node. One can think of the rejection heuristic as an IF-THEN rule, where the IF part (i.e., the rule's preconditions) identifies when to reject a branch and the THEN part as doing the rejection (i.e., the postcondition). The modification should specialize the precondition so that the rule will no longer reject that branch.

A minor focus of this thesis is to simplify the task of reasoning about the effects of adding and/or modifying rejection heuristics upon the coverage of a problem-solver. In particular, it would be useful to know when it is guaranteed that modifying a rejection heuristic will extend the coverage (i.e., that all problems that the problem-solver could solve before, it can still solve). To this end, three types of rejection heuristic interactions are identified that complicate the task of reasoning about the effects of change to the current set of rejection heuristics. This thesis also identifies sufficient conditions for eliminating these three types of interactions.

1.2 Current Approaches and Their Limitations

1.2.1 FS2 and SOAR

There are two systems that modify their rejection heuristics: Bhatnagar's FS2[3] and Lee's version of SOAR[11]. I will briefly discuss how they modify their rejection heuristics and discuss the systems in more detail in Chapter 2.

FS2 treats its explicit rejection rules (which Bhatnagar calls "heuristic censors") as a type of preference rule. When the system decides it is not making progress, it checks whether there are any nodes which have edges pruned away by a heuristic censor. If there are some then it looks for the one that seems closest to a solution and traverses one of the pruned away edges. When a goal is achieved, the FS2 checks the solution

path (from the point where that goal was chosen to the point where it was achieved) for any edges which had been pruned and then traversed. For each such edge, the goal is regressed from its point of achievement back through that edge. The regressed goal is an expression that describes the preconditions needed for that path to be traversed (i.e., for that goal to be achieved in that manner). The negation of the expression is added to the preconditions of the heuristic censor(s) which pruned away that edge. I call this an *success-based edge-level* modification.

Lee's approach[11] to handling rejection heuristics in SOAR is completely different. Lee's approach is to use a sequence of planning methods to attempt to solve problems. Each planning method has a different combination of rejection heuristics (which Lee calls "planning biases"), the methods are ordered so that the initial method has the least coverage and each subsequent method's coverage is a superset of the preceding method's. When a problem is presented, SOAR checks whether it has any rules specifying which methods should not be used on this problem and starts with the lowest method not ruled out. If that method fails to solve the problem then a new rule is learned that will prevent that method from trying to solve similar[1] problems. I call that rule a *failure-based problem-level* modification to the rejection heuristic.

1.2.2 Limitations

Both systems treat their rejection heuristics as a type of preference heuristic, i.e., they do not permanently prune away any part of the search space, they simply defer the searching of that space. This means that for unsolvable problems, the systems will attempt to exhaustively search the complete space. This may not be acceptable because of the potentially high costs of such search attempts. However, it might be acceptable for doing off-line modifications of the rejection heuristics. Therefore I will look at how both systems modify their rejection heuristics in terms of an off-line process. I assume the online problem-solver obeys the rejection heuristics. The off-line process for FS2

[1] In all these systems, generalization techniques are used to go beyond the specific problem to wider class of similar problems. The generalization technique used will define how the problems are similar to the original problem.

begins by deciding where it believes progress is being impeded and for SOAR at the point where the first method to attempt the problem fails to solve it.

I now discuss two specific limitations of these two systems. The first limitation concerns how the off-line versions of these two systems search for which rejection heuristics to modify. Both systems search for which rejection heuristics to modify by searching for a solution while using the rejection heuristics as exotic types of preference heuristics. After finding a solution they then examine the discovered solution path to determine which edges would have been rejected by which rejection heuristics. These rejection heuristics then become the ones to modify. It is the search for a solution that constitutes most of their cost of identifying which rejection heuristics to modify. For some problems this cost of finding a solution (from which to determine which rejection heuristics to modify) will be quite high. The source of this expense is the relinquishing of constraints upon the search, i.e., the relaxing of the rejection heuristics. What is needed is another source of constraints that can be applied to keeping this new search space small.

The second limitation concerns which rejection heuristics the systems can modify. One would like the problem-solvers to be both fast and flexible. Unfortunately, the goals of speed and flexibility are often in conflict. For example, one common technique for making a problem-solver flexible is to have it use explicit search control rules[2] (SCRs). These SCRs enable the problem-solver to modify its search tactics dynamically, e.g., learning new SCRs opportunistically from each new problem encountered. However, SCRs are more expensive to use (i.e., cause the problem-solver to be slower) than embedding the search control knowledge in the problem-solver's code. Thus, the goal of making the problem-solver fast usually encourages embedding most of the standard search control knowledge in the problem-solver's code. One can view this as a spectrum along which speed can be gained at the expense of losing flexibility (and vice versa). At one end there are faster but less flexible problem-solvers where all the search control knowledge is embedded as code, at the other end are the more flexible but slower problem-solvers where all the search control knowledge is represented as SCRs that

[2]Rejection heuristics would be one type of SCR.

```
Original Goal Generator:
   If P is an unsatisfied precondition of a step in the current plan
   then generate goal P.

Original Rejection Heuristic:
   If S is the most recent step added to the plan that has
        unsatisfied preconditions
   and G is one of the current generated goals
   and G is not one of S's preconditions
   then reject goal G.

Goal Generator After Test Movement:
   If P is a unsatisfied precondition of a step in the current plan
    and S is the most recent step added to the plan that has
        unsatisfied preconditions
    and not(P is not one of S's preconditions)
    then generate goal P.

Goal Generator After Test Incorporation:
   If S is the most recent step added to the plan that has
       unsatisfied preconditions
   and P is one of S's preconditions
   then generate goal P.
```

Table 1.1: Test Movement and Incorporation Example

need to be interpreted.

Another example of where these goals come into conflict concerns whether all the problem-solver's steps are explicitly and distinctly represented. To enhance the flexibility of the problem-solving system, the steps of generating and of rejecting search alternatives should be kept separate and explicit. However, often the problem-solver can be speeded up by moving the rejection tests into the generator, this is known as *test movement*[5]. For example, Table 1.1 shows a generator and a rejection heuristic for a planning domain. In this example, initially **P** would be bound to every unsatisfied precondition of the steps in the plan, then generated as a goal candidate, and finally the rejection heuristic would reject all those that weren't unsatisfied preconditions of the most recent step added to the plan that still had unsatisfied preconditions. After test movement (some simplification has been done in the example), **P** would still be bound to every unsatisfied precondition but only those that didn't satisfy the rejection test

(i.e., **P** must be a precondition of the most recently added step that still had unsatisfied preconditions) would actually get generated as goal candidates. This speeds up the problem-solver as it means that some candidates will no longer be generated that are "doomed" to be rejected anyway. However, after the test movement, the information that the unsatisfied preconditions of any step are legal alternatives has been hidden within the generator.

With test movement, one can still see in the generator the logic that generated all of the legal alternatives. However, another speed-up technique, *test incorporation*[2], loses even this. In test incorporation, not only is the rejection test moved into the generator, it actually replaces part of the generator. In this example, after test incorporation, the generator no longer creates bindings for **P** from every unsatisfied precondition from every step, instead it only creates bindings for every unsatisfied precondition of the most recently added step that still has unsatisfied preconditions. Now, the information that unsatisfied preconditions of any step are legal alternatives, has been totally removed from the generator. This can be viewed as another spectrum along which speed can be gained at the expense of flexibility (and vice versa). At one end are the faster but less flexible problem-solvers where all the rejection knowledge has been incorporated into the generator. At the other end are the more flexible but slower problem-solvers where all the rejection knowledge remains distinct and separate from the generation knowledge.

FS2 is limited to which rejection heuristics it can modify by how the heuristics are represented. Specifically, FS2 can only modify those rejection heuristics that are represented as explicit SCRs. Lee's SOAR is limited to which rejection heuristics it can modify by its set of planning methods. Specifically, Lee's SOAR can only modify (or "suspend") a rejection heuristic which appears in earlier methods but not in later methods. One would like to be able to modify any rejection heuristic regardless of how it is represented and regardless of whether it appears in some of a predetermined set of methods.

1.3 Overview

The goals are to be able to modify any rejection heuristic regardless of how it is implemented in the problem-solver and to introduce two new sources of knowledge that the problem-solver can use to constrain its search for which rejection heuristics to modify. One of these sources of knowledge is a solution that is acceptable to the user. If the user is a domain expert using the problem-solver as a labor-saving tool, then the system can act as a Learning-Apprentice[15]. Ideally, when the system is unable to solve a problem, it provides whatever assistance it can to the user (e.g., provide bookkeeping functions, etc.) to enable him to solve the problem and then attempt to learn enough from that solution so that it can solve that type of problem in the future. The other source of knowledge is an annotated theory of the problem-solver.

I assume that the search control embedded in the system's code cannot be directly modified but that explicit search control rules can be used to modify the results of any rejection heuristics within that code. I also assume that there can be explicit generation search control rules as well as explicit rejection search control rules. The explicit generation search control rules compute additional branches for a search node. I further assume the following model of the search control. For any given search node, all of the search control (that generates and rejects branches) that is embedded in the system's code is executed before any of the explicit search control rules are interpreted. Additionally, I assume that explicit generation search control rules can be used to re-introduce branches that were pruned away within the system's code. Lastly, I assume that the explicit rejection search control rules are interpreted after the explicit generation search control rules have been interpreted.

Given this search control model, the system can use a theory of the problem-solver to enable it to modify any of its rejection heuristics. The theory distinguishes between those generators and rejection heuristics that are embedded in code and those that are implemented as explicit search control rules. The distinction is made by annotating the generators and rejection heuristics with how they are implemented.

If the system is to be able to modify every rejection heuristic then the annotated

theory of the problem-solver must describe the generation of the complete search space for a domain and every rejection heuristic that is being used to prune away portions of it. Usually the complete search spaces are very much larger than those searched by the problem-solver.

Because of the size of this space, I will introduce constraints that prune it back to a more manageable size. If the user is knowledgeable about the task the problem-solver is being asked to perform then he may be able to provide sufficient constraints to enable the system to efficiently identify which rejection heuristics to modify. If one assumes that the system will be in a Learning-Apprentice situation, where the user is an expert and is using the system as a labor-saving tool, then when the system is unable to solve a problem it should provide whatever assistance it can to the user (e.g., provide bookkeeping functions, etc.) to enable him to solve the problem. After the user has solved the problem, the system can analyze the solution to determine whether that solution was in its complete search space and if so, then to determine which rejection heuristics prevented the problem-solver from finding it. While this thesis has focussed on planning tasks, I will argue in Section 5.3, that my approach works for any type of search-based task. Since the thesis has focussed on planning, the solutions have been described as solution paths (i.e., plans). However, different types of tasks have different types of solutions. For example, design tasks generate artifact solutions, in which case, the system would require a description of the artifact to be supplied as the solution. Note that the system is only interested in the solution, not in how the user came up with the solution.

Thus, in my approach the system uses a supplied solution to generate constraints on its search for which rejection heuristics to modify to appropriately extend its coverage. After the system has identified these rejection heuristics, it can use standard Explanation-Based Learning (EBL)[14] techniques to explain why the current set of rejection heuristics led to the problem-solver's inability to solve this problem and use that explanation to compute the appropriate modifications. After the modifications have been computed, the rejection heuristic's annotations determine how the modifications will be installed. For example, if the rejection heuristic was implemented as

```
Input:
    - a problem specification
    - a solution to the problem
    - the problem-solver's search control theory described
      explicitly as search control rules that generate and
      reject alternatives
    - annotations on the search control rules that indicate
      how they are implemented in the problem-solver

Output:
    - if no solution is currently in the problem's search space
      but the supplied solution is in the problem-solver's
      complete search space for the problem
      then appropriately modify the problem-solver's explicit
           search control theory
      else leave the explicit search control theory unchanged
```

Table 1.2: I/O Description of SHAPeS

system code then the modification will be installed in a new explicit generation rule that re-instates that branch, while if it was implemented as an explicit SCR then the modification will be installed as a direct modification to that SCR.

Table 1.2 describes the input-output specification of the algorithm. Figure 1.1 shows a flow diagram for the SHAPeS[3] algorithm. The algorithm has three parts. The first identifies which rejection heuristics should be modified. The second computes the appropriate modification for each identified heuristic. The final part installs the modification into the problem-solver's set of explicit search control rules. A restricted version of this algorithm has been implemented as an extension to the PRODIGY[13] problem-solver. This extension is called Bacall. The main restriction is that Bacall can only modify the linearity[19] and the strong linearity [6] rejection heuristics. *Linearity* directs the problem-solver to work on goals in a strict depth-first fashion, i.e., pick a top-level goal and an operator to achieve it, then pick one of that operator's preconditions and an operator to achieve that, etc. *Strong linearity* prohibits the problem-solver from interleaving the solutions to different subproblems. I will discuss both Bacall and

[3]SHAPeS stands for **S**olution-based rejection-**H**euristic Modifier that uses an **A**nnotated theory of the **P**roblem-**S**olver.

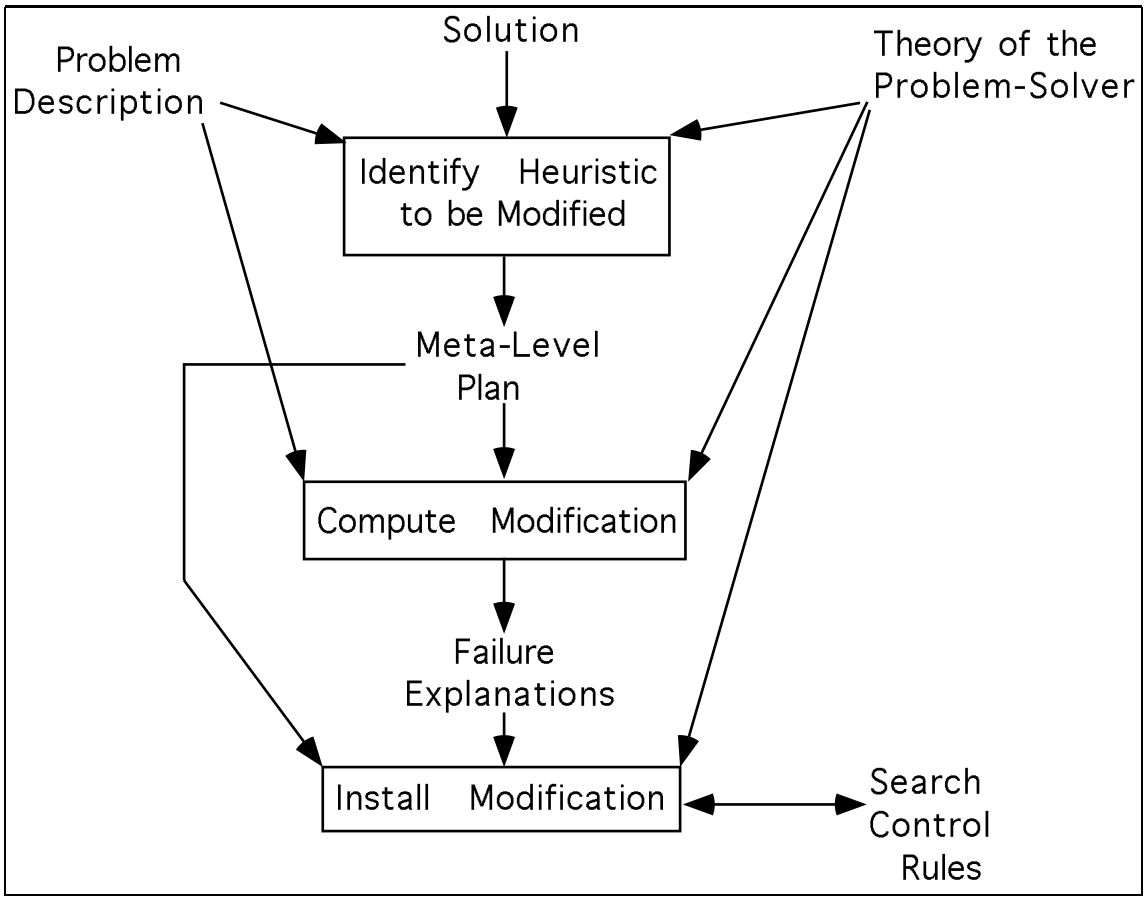

Figure 1.1: SHAPeS Flow Diagram

these rejection heuristics in more detail in Chapter 6.

1.4 Thesis Contributions

This thesis focuses on systems that can extend their domain coverage by modifying their rejection heuristics and has made contributions at a number of different levels. The ideas presented in this thesis allow a learner to:

- Reason about and modify both rejection heuristics that are implemented explicitly and those that are implemented implicitly.
- Use the solution to discover which rejection heuristics to modify rather than using the search path that leads to that solution.

The former allows a system to enhance its performance (both by compiling out search through incorporating rejection heuristics directly into the generators and by directly

embedding the search control logic into the program code) without sacrificing the flexibility usually associated with implementing the search logic explicitly as search control rules. The latter frees the system from having to find the solution itself, which could be prohibitively expensive for the system to do.

The ideas presented in this thesis include:

- Expanding the traditional representation of the theory of problem-solvers to explicitly separate the implicit search logic from the explicit search logic.
- Using a supplied solution to constrain the learner's search for the rejection heuristics to modify.
- Identifying three types of search control knowledge interactions that complicate the analysis of rejection heuristics' effects upon a problem-solver's coverage and describing sufficient conditions for eliminating them.

This thesis has evaluated these ideas by:

- Describing a new algorithm, SHAPeS, that uses a supplied solution and an expanded theory of the problem-solver to modify its rejection heuristics.
- Analyzing the correctness, generality, and computational complexity of the SHAPeS algorithm, in particular, describing when it is guaranteed to identify, in linear time, the rejection heuristics to be modified.
- Implementing a restricted version of SHAPeS, called Bacall.
- Experimentally comparing Bacall's modification method with the currently implemented alternatives.

1.5 Reader's Guide to Thesis

The next chapter describes the related research. In Chapter 3 I use a simple example to describe the framework used in this thesis. Chapter 4 describes the SHAPeS algorithm using the example introduced in the framework chapter. The correctness, generality,

and computational cost of the SHAPeS algorithm is discussed in Chapter 5. Empirical analyses of the utility of the modifications produced by Bacall (a limited implementation of the SHAPeS algorithm) is describes in Chapter 6. Chapter 7 concludes by summarizing the results and suggesting avenues for future research.

Chapter 2

Related Research

This research is concerned with automatically modifying a problem-solver's search heuristics in order to improve its performance. A problem-solver's performance can be measured along three dimensions: the range of problems it can solve (i.e., its coverage); its speed; and the quality of its solutions. Research is currently being carried on in all three areas. We will first discuss the research that is the most germane to this thesis, namely, investigations into how to improve a problem-solvers coverage.

2.1 Increasing Coverage

2.1.1 FS2

FS2[3] dynamically learns, relaxes, and modifies its rejection heuristics (which Bhatnagar calls "heuristic censors") during the course of its attempt to solve a problem. During its attempt to solve a problem, FS2 may learn overgeneral rejection heuristics that impede its progress towards finding a solution. When FS2 believes this to be the case, it selects an edge that was rejected by one of these learned heuristics and locally "relaxes" those rejection heuristics by traversing that edge. If FS2 believes it has made progress (which it defines as achieving one of the problem's goals) then it checks whether any of the edges from the root to its current node had relaxed rejections. If there were any such edges then the weakest preconditions for making that progress (i.e., achieving that goal) is computed for that edge and and its negation is and-ed to the offending rejection heuristic's preconditions.

2.1.2 SOAR

Lee's approach[11] to handling rejection heuristics in SOAR is completely different. Lee's approach is to use a totally-ordered sequence of planning methods to attempt to solve problems. Each planning method has a different combination of rejection heuristics (which Lee calls "planning biases"), the methods are ordered so that the initial method has the least coverage and each subsequent method's coverage is a superset of the preceding method's. The final planning method is expected to have complete coverage. When a problem is presented, SOAR checks whether it has any rules specifying which methods should not be used on this problem and starts with the lowest method not ruled out. If that method fails to solve the problem then a new rule is learned that will prevent that method from trying to solve similar problems. If there is a higher method left, then the next method in the sequence tries to solve the problem. Lee calls this approach *coarse-grained* multi-method planning.

Lee also investigated a finer-grained approach. In this approach, anytime a new set of subgoals arises the system uses rules to decide which planning method to use. If that method fails to achieve those subgoals then the system tries the next method in the sequence until either a solution is found or the problem is determined to be unsolvable. In either case the system will learn new rules to avoid using methods that have already proven to be inadequate. Lee's empirical investigations showed that the finer-grained approach is superior because the rejection heuristics can be used for the majority of the problem and only suspended where necessary. Lee's finer-grained approach is at the search node-level and is coarser[1] than FS2's edge-level approach. Intuitively, we would expect that, everything else being equal, the edge-level approach will suspend the rejection heuristic even less and therefore search fewer nodes.

2.2 Increasing Speed

One of the initial impetuses to using learning in planning domains was to find ways of automatically speeding up general purpose planners. For example, systems would be

[1] It is coarser in the sense that it rejects more alternatives at a time and consequently is less selective.

faster if they learned to avoid paths that were doomed to fail. This has been a very large area of research and we will only look at three systems. These three systems illustrate the evolution of the appreciation for the importance of search heuristics. PRODIGY-EBL represents the initial attempts to learn sound rejection rules by using a complete theory of the problem-solver. These rules proved expensive to learn which led to attempts, as exemplified by FS2, to decrease the cost of learning rejection rules by using minimal theories of the problem-solver. However, these rules could be very overgeneral causing the problem-solver to have a very limited domain coverage. This led to a search for a better compromise between the quality of the rejection rules learned and the cost of learning them. HAMLET is an example of a system that attempts such a compromise. While these systems learned more than just rejection search control rules, I will only discuss that one aspect.

2.2.1 PRODIGY-EBL

PRODIGY-EBL[13] learned rejection search control rules from failed search subtrees. The learner used a theory of the problem-solver to identify the leaf failure reasons which were then regressed up the subtree. While the theory was relatively complete, it did not capture all of the relevant aspects of every failure. In particular, the theory did not mention that the absence of top-level goals could affect whether a problem was solvable. For example, while PRODIGY could not solve the standard Sussman's Anomaly, it could solve that problem if the goal **clear(A)** was added to the problem description. However, when the learner computed the explanation for why PRODIGY failed to solve Sussman's Anomaly, no mention is made of the fact that there are no other top-level goals present. The use of an incomplete theory of the problem-solver led to overgeneral rejection rules, i.e., a rejection heuristic. PRODIGY-EBL was not attempting to learn heuristic rejection rules, it was trying to learn "logically correct" rejection rules. The problem was that to do so, it either had to make them overspecific (e.g., to add the condition "and no other top-level goals are present" to the preconditions of the rejection rule learned from Sussman's Anomaly) or to unacceptably increase the learning cost by extending PRODIGY's search to cover the problem's complete search

space.

2.2.2 FS2

While PRODIGY-EBL tried to learn rejection rules that would not eliminate any solutions, FS2 was much more aggressive. FS2 did not mind eliminating solutions as long as it didn't eliminate all of them. Therefore FS2 was able to use a much simpler and much cheaper theory (which Bhatnagar called an *impossibility theory*) to explain its search failure nodes. This was feasible because, unlike PRODIGY, if FS2 thought that its rejection rules were overconstraining its search, it could relax them. FS2's impossibility theory explained search dead ends by describing pairs of conditions which could not be simultaneously true. If one of the pair's conditions was the current goal and the other was true in the current state then that pair was considered to be the explanation for that failure. The pair was then regressed back to the step that achieved the latter condition. The regressed condition became the precondition for when not to select that operator to work towards that goal[2]. Unfortunately, the impossibility theory is domain specific and each new domain requires a new one.

2.2.3 HAMLET

HAMLET[4] learns PRODIGY select rules for the different types of choice points found in NOLIMIT[20], PRODIGY's non-linear problem-solver. HAMLET uses two mechanisms to produce overgeneral preconditions for these search control rules. One is to use an explanation template (that is specific for the type of choice point for which the select rule is being learned) that limits what will appear in the explanation of why selecting that choice is good. This produces an approximation to the explanation that is formed by simply doing goal regression. If it is overgeneral then it can be considered to be a heuristic since it is not guaranteed to choose correctly. We call the rules learned via bounded explanation *bounded rules*.

The second mechanism is to do inductive generalization over pairs of bounded rules.

[2]FS2 was a forward-chaining planner which did not explicitly do operator subgoaling, but which did select which top-level goal it was currently working towards.

HAMLET uses a number of inductive operators to merge the preconditions of the two rules. For example, one operator is to take the intersection of the two rules' preconditions. HAMLET also has refinement operators that it uses to selectively specialize its induced operators when it finds that a choice made by such a rule failed. One such refinement operator is to add back in some of the preconditions in the set difference of the preconditions of the two bounded rules.

2.3 Increasing Solution Quality

This was the last of problem-solver performance dimensions to be attacked. While many systems may implicitly learn search control rules to improve solution quality, QUALITY[17] does so explicitly.

2.3.1 QUALITY

Perez[17] attacked the problem of learning search control rules that lead to better solutions. Learning was triggered when the user could find a better solution (according to a given objective function[3]) than the one found by the planner's current set of search control rules. Her learning algorithm was given the domain theory (i.e., the domain operators and inference rules), the objective function, the search trace for the solution found by the current search control rules, and the search trace the planner would have needed to have created to find the user's solution. The learner compared the two traces to identify why the user's plan was better and then to locate the decision points where the two traces diverged. The learner computed explanations of why the user's plan is better and operationalized those explanations with respect to the information available at those decision points. These explanations provided the precondition for preference search control rules.

[3]The objective function must compute the quality of the plan as a sum of the costs of the individual operators.

Chapter 3

Framework

In this chapter I define and illustrate terms used throughout the rest of this thesis. I start by describing an example domain, then describe a problem-solver whose search control logic has been specialized to solve problems in this domain, then continue by defining and illustrating what I mean by a theory of the problem-solver, and finish by using the theory of the problem-solver to explain why the problem-solver is unable to solve a particular solvable problem in that domain.

3.1 Tiles: An Example Domain

Table 3.1 describes the Tile domain. I will be using this domain in this and the next chapters to illustrate the learning algorithm. The domain consists of problems concerned with moving a single tile around a four-by-four board. Aside from the square containing the tile, squares are either empty or contain an immovable hole. Problems have an initial board configuration and a goal description (which describes where the tile should end up). There are only four operators, **UP**, **DOWN**, **RIGHT**, and **LEFT**, which move the tile in the corresponding direction. These operators are shown in Table 3.1. The board has holes at various locations. From the operator descriptions one sees that the tile cannot be moved off the board nor can it moved onto a square that has a hole. The holes can appear in different locations in different problems. Figure 3.1 shows a problem in this domain.

3.2 FCPS: An Example Problem-Solver

There are a number of systems[10, 13] that make their problem-solving components flexible by making some of the search control knowledge explicitly modifiable, which

RIGHT(?Col, ?Row)
Preconditions:
TILE-AT(?Col, ?Row)
?Col < 4
not(HOLE-AT(?Col + 1, ?Row))
Deletes:
TILE-AT(?Col, ?Row)
Adds:
TILE-AT(?Col + 1, ?Row)

LEFT(?Col, ?Row)
Preconditions:
TILE-AT(?Col, ?Row)
?Col > 1
not(HOLE-AT(?Col - 1, ?Row))
Deletes:
TILE-AT(?Col, ?Row)
Adds:
TILE-AT(?Col - 1, ?Row)

UP(?Col, ?Row)
Preconditions:
TILE-AT(?Col, ?Row)
?Row < 4
not(HOLE-AT(?Col, ?Row + 1))
Deletes:
TILE-AT(?Col, ?Row)
Adds:
TILE-AT(?Col, ?Row + 1)

DOWN(?Col, ?Row)
Preconditions:
TILE-AT(?Col, ?Row)
?Row > 1
not(HOLE-AT(?Col, ?Row - 1))
Deletes:
TILE-AT(?Col, ?Row)
Adds:
TILE-AT(?Col, ?Row - 1)

Table 3.1: Tile Operators

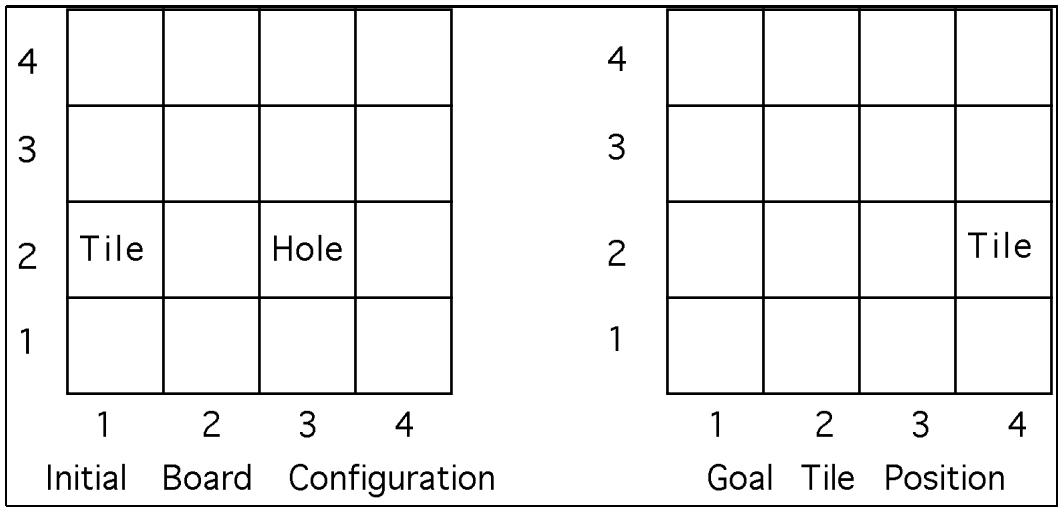

Figure 3.1: An Example Problem in the Tile Domain

enables their learning components to automatically tailor this search control knowledge to new domains and/or new tasks. However, even though much of the problem-solver's search control knowledge may be explicitly modifiable, there usually remains a portion that is immutable. The immutable portion typically decides which search choice selection needs to be made next and whether the choice selection is backtrackable. The explicitly modifiable portions usually make the choice selections, e.g., deciding which operator to use to achieve a given goal, etc. I will follow Minton's[13] terminology and call the former the *problem-solving procedure* and call the latter the *decision procedures*. While the decision procedures are modifiable, not all parts of the decision procedure need be directly modifiable. For example, the decision procedure may be implemented as a mixture of embedded code and explicit search control rules. While the system's learning component cannot directly modify the embedded code, it might be able to indirectly modify it by adding search control rules.

These decision procedures can be classified according to how they affect the solution construction, e.g., for planning problems there are operator selection decisions, parameter binding selection decisions, plan step location decisions, etc. Each type of decision is made by a corresponding decision procedure. The decision procedure creates an ordered list of candidates to be tried in the solution. I will be primarily concerned with whether problems are solvable, not with how efficiently they are solved. Therefore

I will ignore candidate ordering and view decision procedures as producing unordered sets of candidates.

The decision procedures are modified by adding new search control rules or by modifying the old ones. Not all of the decision procedures need be implemented as search control rules. Some parts of a decision procedure can be implemented as code within the problem-solver. Those parts so implemented are called *embedded*. Search control routines implemented as embedded code run much faster than when implemented as rules (which need to be interpreted). However, the problem-solver's ability to use search control rules enables the user to easily modify the behavior of the problem-solver as his experience with the problem-solver and the domain increases. I assume there are the following phases in a decision procedure:

- Embedded generation of initial candidates.
- Rule-based generation of candidates.
- Rule-based rejection of candidates.

The example problem-solver, **FCPS**, is a **F**orward **C**haining **P**roblem-**S**olver. **FCPS** searches through a space of partial solutions candidates. Table 3.2 shows the pseudo-code for **FCPS**'s problem-solving procedure. **FCPS** does depth-first search and calls one decision procedure, INSERT-STEP. Table 3.3 shows the pseudo-code for this decision procedure. The code had been made domain specific to simplify the example. **FCPS** is doing forward chaining by adding operators to the end of the plan. If an operator's preconditions are not satisfied when it is added to the plan then adding that operator is illegal. The LEGAL-MOVE(?Move, ?Board-Position) meta-level predicate tests whether all of ?Move's preconditions are satisfied by ?Board-Position. The problem-solver uses two rejection heuristics to cut down the amount of search the problem-solver will do. One rejection heuristic is to only move the tile towards its goal destination, e.g., only if the the goal destination for the tile is above the current tile position can the tile be moved up. The other rejection heuristic is that the tile will first be moved to its goal column and then to its goal row. These rejection heuristics have

```
SET OPEN STACK TO CONTAIN THE ROOT NODE;
LOOP FOREVER
        WHEN THE OPEN STACK IS EMPTY THEN EXIT WITH FAILURE;
        POP NODE OFF OF OPEN STACK;
        WHEN THE NODE IS A SOLUTION THEN EXIT WITH THE
              SOLUTION;
        USE THE INSERT-STEP DECISION PROCEDURE TO PRODUCE
              CHILDREN FOR THIS NODE;
        PUSH ALL NEW CHILDREN ONTO THE OPEN STACK;
END LOOP.
```

Table 3.2: Pseudo-Code for FCPS Problem-Solving Procedure

been moved into the embedded candidate generation code. **FCPS** does not currently have any search control rules, i.e., its only decision procedure, INSERT-STEP, is totally implemented in the code.

The space generated by the embedded search control code is called the *underlying search space*. There could be both generation and rejection search control rules in **FCPS**'s decision procedure. The generation search control rules add edges (and possibly nodes) to the underlying search space generated by **FCPS**'s embedded search control code. This new space is called the *explicit generation space*. The rejection search control rules remove edges from the explicit generation space producing a space called the *explicit search space*. When there are no generation search control rules, the explicit generation space is the same as the underlying search space. When there are no rejection search control rules, the explicit search space is the same as the explicit generation space.

3.3 A Theory of the FCPS Problem-Solver

I view problem-solvers as incrementally constructing solutions to problems. Given a problem, the problem-solver begins with an initial partial solution (which may be vacuous) and searches for a complete solution by exploring different sequences of extensions applied to the initial partial solution. These extensions are the problem-solver's operations upon partial solutions, i.e., they are meta-level operators. The problem-solver's

Generation Rules Implemented in Code

If CURRENT-BOARD(?B)
& LEGAL-MOVE(RIGHT(?ROW-1, ?COL-1), ?B)
& GOAL(TILE-AT(?ROW-2, ?COL-2))
& ?ROW-1 < ?ROW-2
Then GENERATE MOVE RIGHT(?ROW-1, ?COL-1)

If CURRENT-BOARD(?B)
& LEGAL-MOVE(LEFT(?ROW-1, ?COL-1), ?B)
& GOAL(TILE-AT(?ROW-2, ?COL-2))
& ?ROW-1 > ?ROW-2
Then GENERATE MOVE LEFT(?ROW-1, ?COL-1)

If CURRENT-BOARD(?B)
& LEGAL-MOVE(UP(?ROW-1, ?COL-1), ?B)
& GOAL(TILE-AT(?ROW-2, ?COL-2))
& ?ROW-1 = ?ROW-2
& ?COL-1 < ?COL-2
Then GENERATE MOVE UP(?ROW-1, ?COL-1)

If CURRENT-BOARD(?B)
& LEGAL-MOVE(DOWN(?ROW-1, ?COL-1), ?B))
& GOAL(TILE-AT(?ROW-2, ?COL-2))
& ?ROW-1 = ?ROW-2
& ?COL-1 > ?COL-2
Then GENERATE MOVE DOWN(?ROW-1, ?COL-1)

Table 3.3: Pseudo-code for FCPS's INSERT-STEP Decision Procedure

search space consists of nodes, which represent the partial solutions, and edges, which represent the meta-level operators that transforms the parent node's partial solution into the child's partial solution. Given a node, the problem-solver generates its children and prunes away those children rejected by its heuristics. Unfortunately, these rejection heuristics sometimes eliminate all paths to a problem's solutions.

When such a problem is encountered, the learner uses a theory of the problem-solver to determine which rejection heuristics are responsible and to suggest how the problem-solver should be modified. The theory describes the problem-solver in terms of generating and rejecting solution construction choices. One approach to analyzing the failure proceeds by identifying a meta-level path (i.e., a sequence of generated solution construction choices) that leads to a solution and then identifying which heuristics eliminated any of the meta-level steps on that path. I assume that the theory is represented by sets of generation and rejection rules.

One reason to do analysis upon a theory of the problem-solver instead of upon the problem-solver's itself (i.e., its source code) is that the theory can be couched in a much higher level language that focuses on the generation and pruning of the search space. However, a much more important reason is that two important tools in optimizing search code are the movement of and the incorporation of test (i.e., rejection) code into generation code. As I mentioned in Section 1.2.2, test movement and test incorporation cause some parts of the search space to never be generated. While this speeds up the problem-solver it can also make the learner's task much more difficult because the meta-level paths to solutions may not appear in the generation space. While one wants to be able to use test movement and test incorporation to speed up the problem-solver, one may need to separate the moved and/or incorporated tests from the original solution construction choice generation to allow identification of solution meta-level paths and of the heuristics eliminating them. By doing this separation in the theory of the problem-solver, the problem-solver code can still be speeded up via test movement and test incorporation while giving the learner access to a generation space containing solutions and thus enabling the learner to identify, regardless of how they're implemented in the problem-solver, those rejection heuristics that are eliminating the solutions.

For the learner to successfully use the theory to find a solution not currently in the problem-solver's search space and to identify which rejection heuristics eliminated it, the theory must describe a generation space that includes that solution and must describe what pruned it. To do this the theory breaks the problem-solver's embedded generation phase into two phases: the *implicit generation* of candidates; and the *implicit rejection* phase. Thus, while the problem-solver's decision procedures can be described as having three phases (as I did in Section 3.2), the theory will describe them as having the following four phases:

- Implicit generation of legal candidates.
- Implicit rejection of legal candidates.
- Rule-based generation of candidates.
- Rule-based rejection of candidates.

Since one doesn't know which solution one will want to reincorporate back into the problem-solver's search space, the implicit generation phase should generate a space, the *implicit generation space*, where all legal paths are represented in the implicit generation space. The implicit generation space is what I was calling the "complete search space" in Section 1.3. The implicit rejection phase then prunes back the implicit generation space back to the problem-solver's underlying search space. To ensure that all legal paths are in the implicit generation space, only those tests that check legality of the path should be in the generation rules. Figure 3.2 shows the relationship between the four phases of the decision procedure and the search spaces they create.

The learner uses the theory to decide how to modify the problem-solver's search control logic to extend the problem-solver's domain coverage. If the rejection rule in the theory that eliminated the solution is implemented as embedded code then the learner cannot directly modify the rejection code (which might be incorporated directly into the generation code and consequently only be there in the sense of certain parts of the generation code being absent). Instead the learner will add an explicit generation rule to the problem-solver's search control rules. However, if the rejection rule in the theory

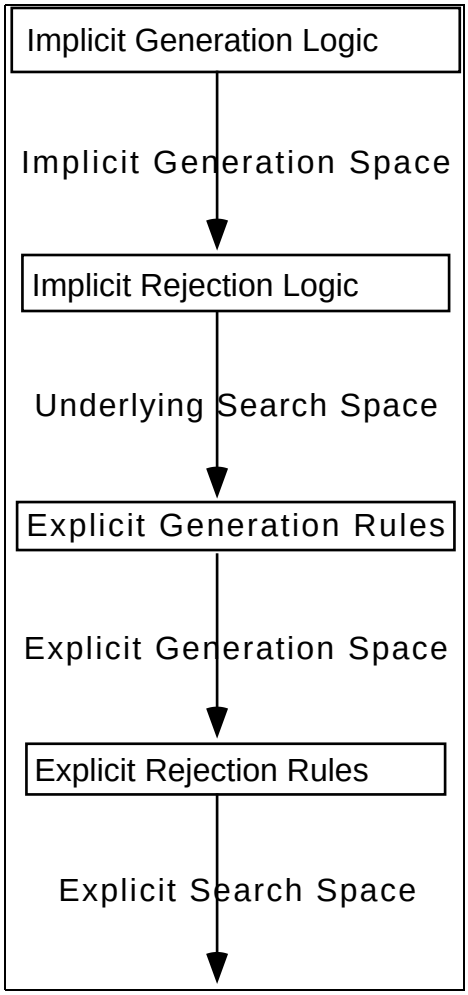

Figure 3.2: The relationship between the decision procedure's phases and their search spaces

Implicit Generation Rules:

Gen-RIGHT(?ROW-1, ?COL-1)
If CURRENT-BOARD(?B) & LEGAL-MOVE(RIGHT(?ROW-1, ?COL-1),?B)
Then GENERATE MOVE RIGHT(?ROW-1, ?COL-1)

Gen-LEFT(?ROW-1, ?COL-1)
If CURRENT-BOARD(?B) & LEGAL-MOVE(LEFT(?ROW-1, ?COL-1),?B)
Then GENERATE MOVE LEFT(?ROW-1, ?COL-1)

Gen-UP(?ROW-1, ?COL-1)
If CURRENT-BOARD(?B) & LEGAL-MOVE(UP(?ROW-1, ?COL-1),?B)
Then GENERATE MOVE UP(?ROW-1, ?COL-1)

Gen-DOWN(?ROW-1, ?COL-1)
If CURRENT-BOARD(?B) & LEGAL-MOVE(DOWN(?ROW-1, ?COL-1),?B)
Then GENERATE MOVE DOWN(?ROW-1, ?COL-1)

Table 3.4: A Theory of FCPS: Implicit Generation Rules

that eliminated the solution is implemented in the problem-solver as a rejection search control rule then a direct approach is needed, i.e., to directly modify that rejection search control rule. Tables 3.4 and 3.5 show the theory of FCPS.

3.4 Example of Using Theory of FCPS to Extend Coverage

Figure 3.3 shows a problem that **FCPS** currently is unable to solve. There is a hole between where the tile is initially and where the goal specifies it should end up. **FCPS** cannot solve this problem because the problem-solver's embedded generation rules say that when the goal position is to the right of the current position of the tile that the only possible branch that could be generated is the one for the RIGHT tile operator. However, when the tile is moved next to the hole then even the RIGHT operator will not be generated because a RIGHT operator here is not legal. Figure 3.4 shows the problem-solver's underlying search space. From this search space one cannot find solutions nor understand how to modify FCPS's search control knowledge to enable it to find solutions for this problem. Figure 3.5 shows part of the implicit generation space generated by the theory of FCPS. The figure does not show any of the edges that

Implicit Rejection Rules:

Rej-RIGHT(?ROW-1, ?COL-1)
If CANDIDATE(RIGHT(?ROW-1, ?COL-1))
& GOAL(TILE-AT(?ROW-2, ?COL-2))& ?ROW-1 $\geq$?ROW-2
Then REJECT MOVE RIGHT(?ROW-1, ?COL-1)

Rej-LEFT(?ROW-1, ?COL-1)
If CANDIDATE(LEFT(?ROW-1, ?COL-1)
& GOAL(TILE-AT(?ROW-2, ?COL-2)) & ?ROW-1 $\leq$?ROW-2
Then REJECT MOVE LEFT(?ROW-1, ?COL-1)

Rej-UP-1(?ROW-1, ?COL-1)
If CANDIDATE(UP(?ROW-1, ?COL-1)
& GOAL(TILE-AT(?ROW-2, ?COL-2)) & ?COL-1 $\geq$?COL-2
Then REJECT MOVE UP(?ROW-1, ?COL-1)

Rej-UP-2(?ROW-1, ?COL-1)
If CANDIDATE(UP(?ROW-1, ?COL-1)
& GOAL(TILE-AT(?ROW-2, ?COL-2)) & ?ROW-1 $\neq$?ROW-2
Then REJECT MOVE UP(?ROW-1, ?COL-1)

Rej-DOWN-1(?ROW-1, ?COL-1)
If CANDIDATE(DOWN(?ROW-1, ?COL-1)
& GOAL(TILE-AT(?ROW-2, ?COL-2)) & ?COL-1 $\leq$?COL-2
Then REJECT MOVE DOWN(?ROW-1, ?COL-1)

Rej-DOWN-2(?ROW-1, ?COL-1)
If CANDIDATE(DOWN(?ROW-1, ?COL-1)
& GOAL(TILE-AT(?ROW-2, ?COL-2)) & ?ROW-1 $\neq$?ROW-2
Then REJECT MOVE DOWN(?ROW-1, ?COL-1)

Table 3.5: A Theory of FCPS: Implicit Rejection Rules

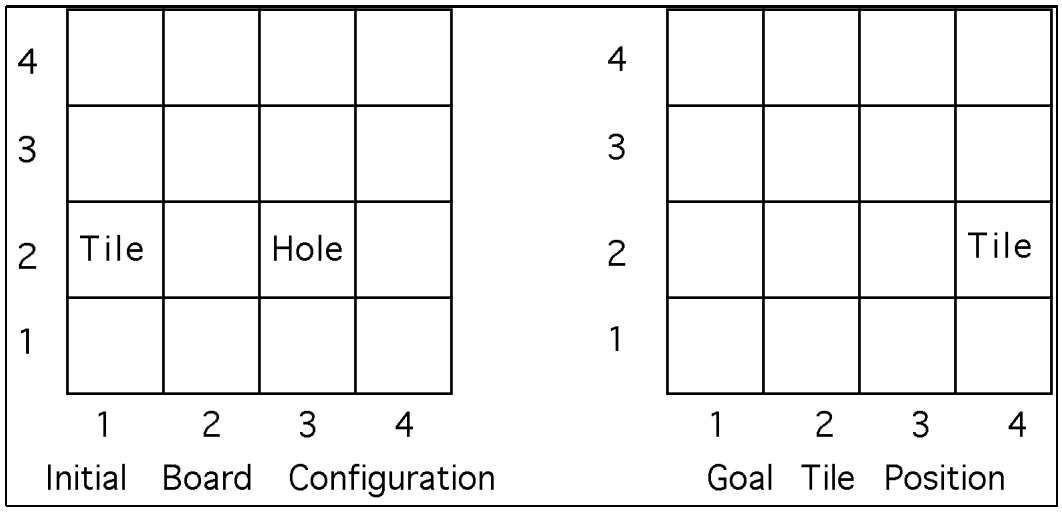

Figure 3.3: Problem that FCPS is Currently Unable to Solve

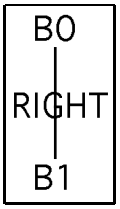

Figure 3.4: FCPS's Underlying Search Space for Problem

simply reverse the last edge and the figure only shows a depth of five. The supplied solution path is shown as the darkened path leading to the boxed node **B16**. Figure 3.6 shows both the implicit search space generated by the theory of FCPS and which edges have been pruned away by implicit rejection rule set. By checking the implicit rejection rules' preconditions, one can identify which ones are blocking a particular solution path. For example, one can see that the heavily marked path only has one edge blocked, specifically, it is blocked from adding the **UP** operator just below node **B1**. Checking the implicit rejection rules' preconditions one finds that both implicit rejection rules **Rej-UP-1** and **Rej-UP-2** are blocking that edge. To enable FCPS to solve this problem, one only needs to add a generation rule (that adds that edge back into the explicit generation space) to FCPS's search control rules. In the next chapter I will describe one approach to computing an appropriate generation rule.

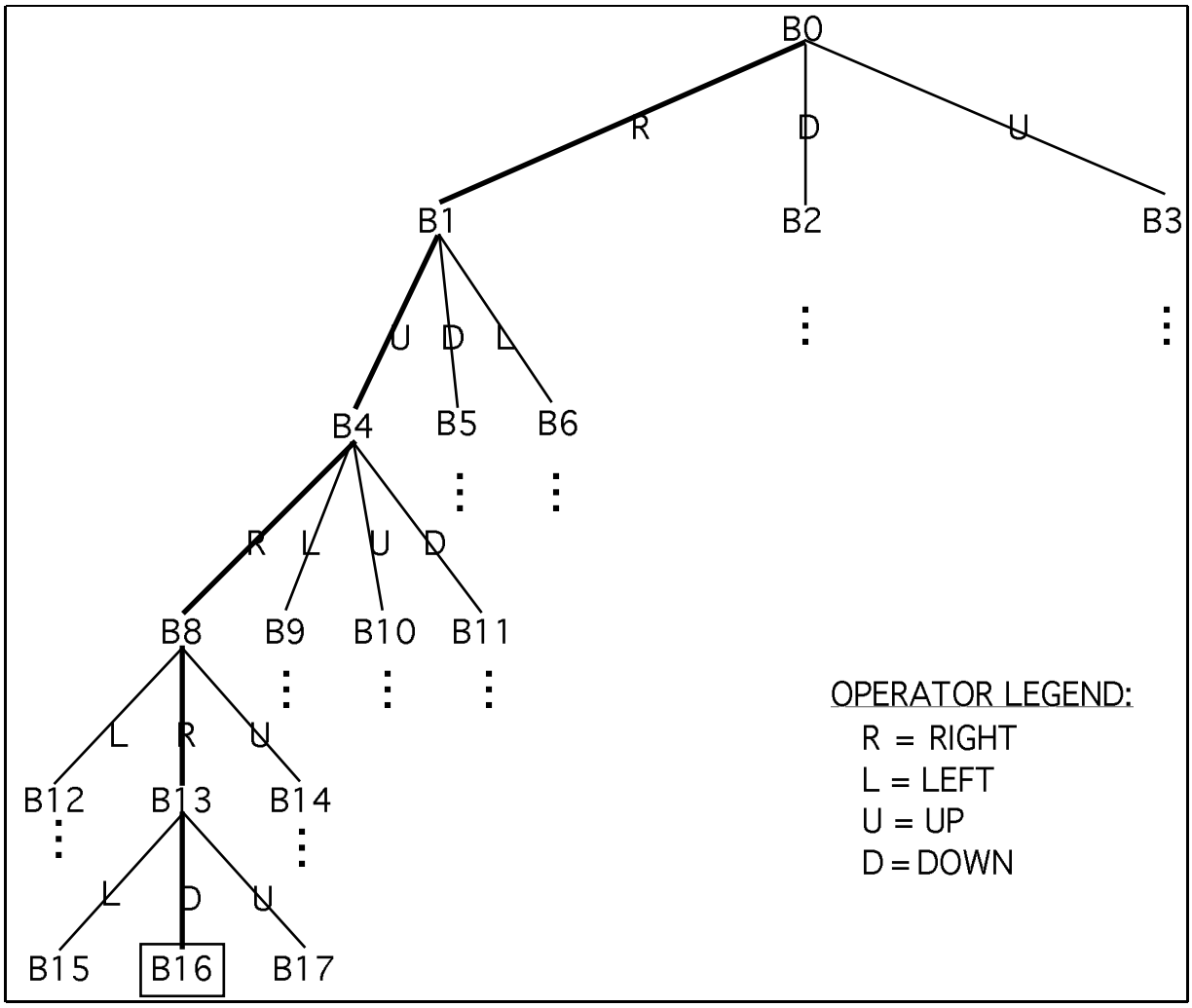

Figure 3.5: FCPS's Implicit Generation Space for Problem

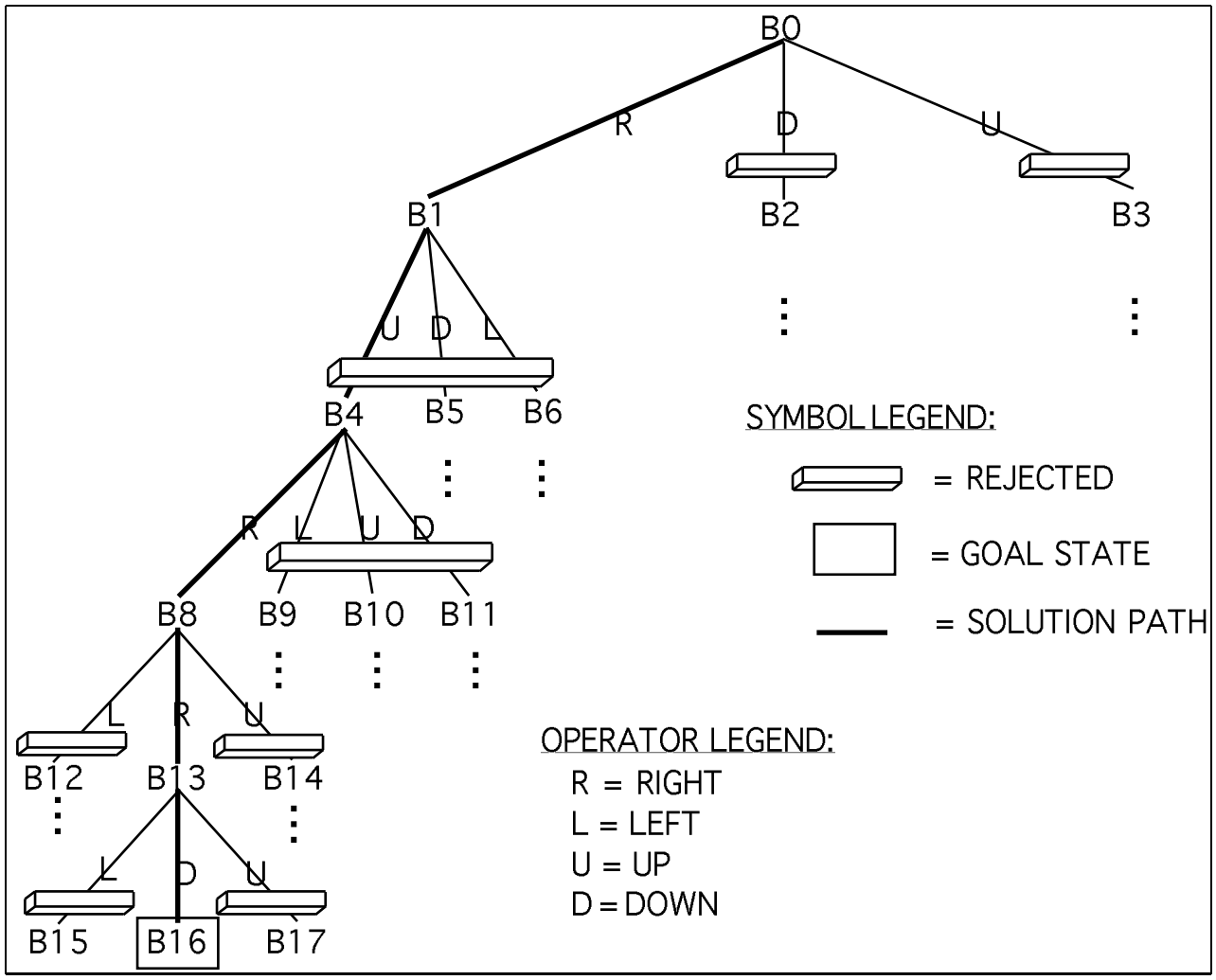

Figure 3.6: FCPS's Implicit Search Space for Problem

Chapter 4

The SHAPeS Algorithm

This chapter describes the SHAPeS[1] algorithm for modifying a problem-solver's rejection heuristics to extend its domain coverage.

The next section gives an overview of the algorithm in terms of its main procedures and describes an example in terms of its input and output to these procedures. Section 4.2 then describes these procedures in more detail. Section 4.3 describes some of the design alternatives.

4.1 Overview of Algorithm

In this section I present an overview of the SHAPeS algorithm. I will use the example introduced in Section 3.4.

Figure 4.1 shows a flow diagram for the SHAPeS algorithm. The algorithm has three components. The first one identifies which rejection heuristics should be modified. The second computes the appropriate modification for each identified heuristic. The last component installs the modification into the problem-solver's set of explicit search control rules.

I now look at the example's input to the SHAPeS algorithm. Figure 4.2 shows the problem description. Table 4.1 shows a solution to the problem. Tables 4.2 and 4.3 show a theory of the problem-solver.

Table 4.4 shows the meta-level plan (discovered by the first component) which would enable FCPS to find the solution shown in Table 4.1. The meta-level steps in this plan

[1]SHAPeS stands for **S**olution-based Rejection-**H**euristic Modifier that uses an **A**nnotated theory of the **P**roble**m**-**S**olver.

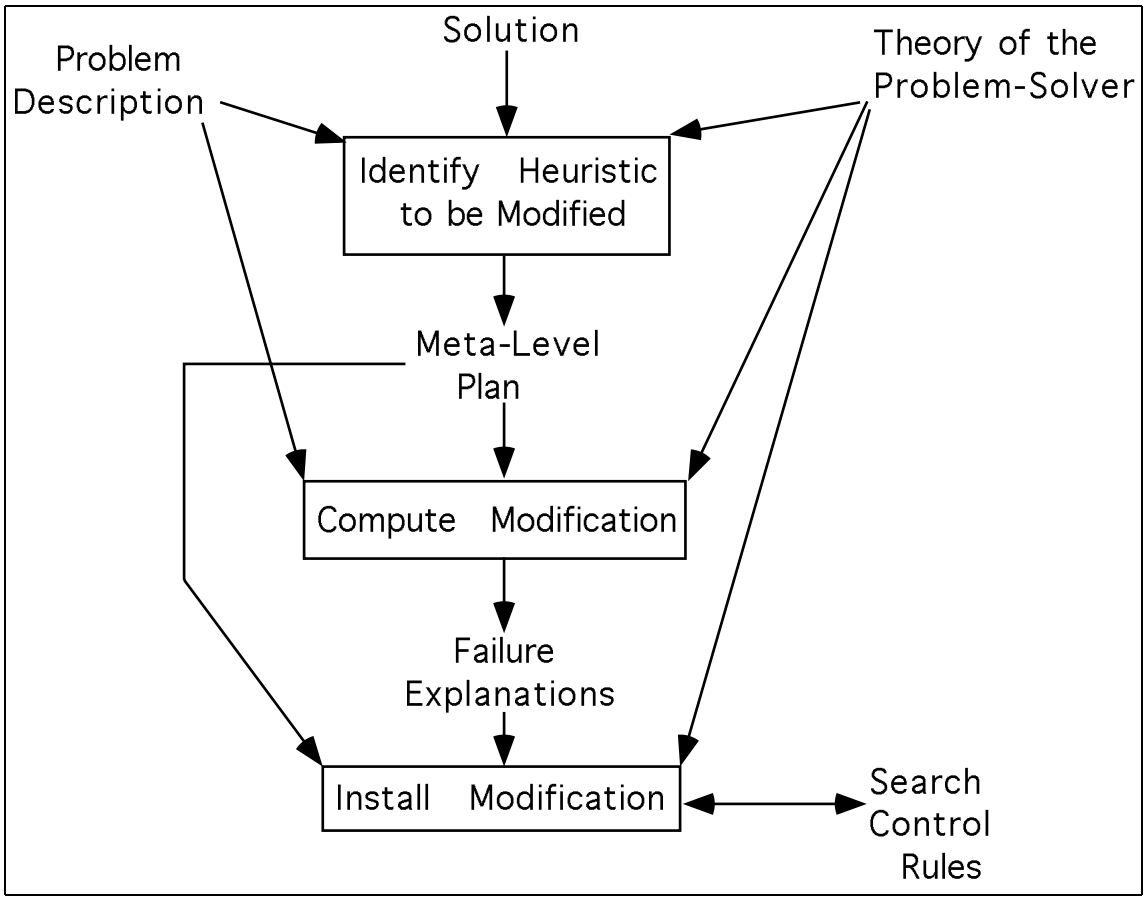

Figure 4.1: SHAPeS Flow Diagram

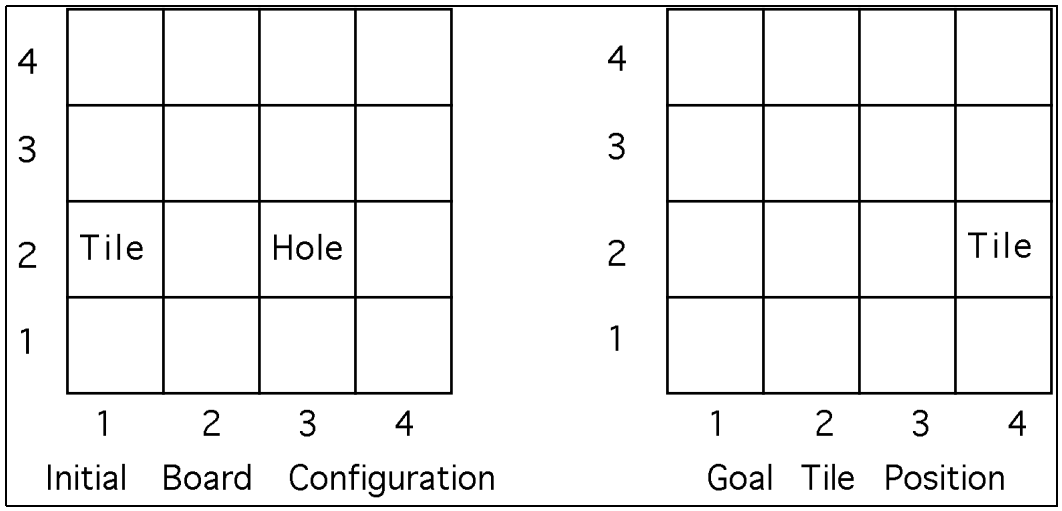

Figure 4.2: Problem that FCPS is Currently Unable to Solve

RIGHT(1,2)
UP(2,2)
RIGHT(2,3)
RIGHT(3,3)
DOWN(4,3)

Table 4.1: Solution to Problem

Implicit Generation Rules:

Gen-RIGHT(?ROW-1, ?COL-1)
If CURRENT-BOARD(?B) & LEGAL-MOVE(RIGHT(?ROW-1, ?COL-1),?B)
Then GENERATE MOVE RIGHT(?ROW-1, ?COL-1)

Gen-LEFT(?ROW-1, ?COL-1)
If CURRENT-BOARD(?B) & LEGAL-MOVE(LEFT(?ROW-1, ?COL-1),?B)
Then GENERATE MOVE LEFT(?ROW-1, ?COL-1)

Gen-UP(?ROW-1, ?COL-1)
If CURRENT-BOARD(?B) & LEGAL-MOVE(UP(?ROW-1, ?COL-1),?B)
Then GENERATE MOVE UP(?ROW-1, ?COL-1)

Gen-DOWN(?ROW-1, ?COL-1)
If CURRENT-BOARD(?B) & LEGAL-MOVE(DOWN(?ROW-1, ?COL-1),?B)
Then GENERATE MOVE DOWN(?ROW-1, ?COL-1)

Table 4.2: A Theory of FCPS: Implicit Generation Rules

Implicit Rejection Rules:

Rej-RIGHT(?ROW-1, ?COL-1)
If CANDIDATE(RIGHT(?ROW-1, ?COL-1))
& GOAL(TILE-AT(?ROW-2, ?COL-2))& ?ROW-1 $\geq$?ROW-2
Then REJECT MOVE RIGHT(?ROW-1, ?COL-1)

Rej-LEFT(?ROW-1, ?COL-1)
If CANDIDATE(LEFT(?ROW-1, ?COL-1)
& GOAL(TILE-AT(?ROW-2, ?COL-2)) & ?ROW-1 $\leq$?ROW-2
Then REJECT MOVE LEFT(?ROW-1, ?COL-1)

Rej-UP-1(?ROW-1, ?COL-1)
If CANDIDATE(UP(?ROW-1, ?COL-1)
& GOAL(TILE-AT(?ROW-2, ?COL-2)) & ?COL-1 $\geq$?COL-2
Then REJECT MOVE UP(?ROW-1, ?COL-1)

Rej-UP-2(?ROW-1, ?COL-1)
If CANDIDATE(UP(?ROW-1, ?COL-1)
& GOAL(TILE-AT(?ROW-2, ?COL-2)) & ?ROW-1 $\neq$?ROW-2
Then REJECT MOVE UP(?ROW-1, ?COL-1)

Rej-DOWN-1(?ROW-1, ?COL-1)
If CANDIDATE(DOWN(?ROW-1, ?COL-1)
& GOAL(TILE-AT(?ROW-2, ?COL-2)) & ?COL-1 $\leq$?COL-2
Then REJECT MOVE DOWN(?ROW-1, ?COL-1)

Rej-DOWN-2(?ROW-1, ?COL-1)
If CANDIDATE(DOWN(?ROW-1, ?COL-1)
& GOAL(TILE-AT(?ROW-2, ?COL-2)) & ?ROW-1 $\neq$?ROW-2
Then REJECT MOVE DOWN(?ROW-1, ?COL-1)

Table 4.3: A Theory of FCPS: Implicit Rejection Rules

(Gen-RIGHT(1,2))
(Gen-UP(2,2), Rej-UP-1(2,2), Rej-UP-2(2,2))
(Gen-RIGHT(2,3))
(Gen-RIGHT(3,3))
(Gen-DOWN(4,3))

Table 4.4: SHAPeS's Meta-Level Plan to Produce Solution

CURRENT-BOARD(?B)
& LEGAL-MOVE(UP(?X, ?Y), ?B)
& NOT(LEGAL-MOVE(RIGHT(?X, ?Y), ?B))
& GOAL(TILE-AT(?V, ?Y))
& ?X $\leq$?V
& ?X $\neq$?V

Table 4.5: Failure Explanation

are annotated with which rejection heuristics (if any) would reject this step. The meta-level plan is represented by a sequence of ordered-tuples. The first component of the tuple is the generation meta-step that creates the branch and the remaining components identify the rejection heuristics that prune this branch. It is these rejection heuristics that are to be modified. In this case, the Rej-UP-1 and Rej-UP-2 rejection heuristics will be modified.

Back in Figure 4.1, one saw that this annotated meta-level plan, the problem description, and the theory of the problem-solver are the inputs used by the second component to compute the modifications. Standard EBL is used to compute the failure explanation that will be used in modifying the identified rejection heuristics. Table 4.5 shows the computed failure explanation[2]. The explanation is that the RIGHT operator is not a legal move (because of the hole to the right of the tile), the goal position for the tile is on the same row as it is currently, therefore both UP and DOWN operators are rejected because they don't move the tile towards its goal position, and the LEFT operator is also rejected because it doesn't move the tile towards its goal position. The annotated meta-level plan is modified so that this failure explanation is associated with the rejection heuristics it will modify.

[2]The exact explanation depends upon the system's operationality criteria.

If CURRENT-BOARD(?B)
& LEGAL-MOVE(UP(?X, ?Y), ?B)
& NOT(LEGAL-MOVE(RIGHT(?X, ?Y), ?B))
& GOAL(TILE-AT(?V, ?Y))
& ?X < ?V
& LEGAL-MOVE(UP(?X, ?Y), ?B)
Then GENERATE MOVE UP(?X ?Y)

Table 4.6: New Generation Rule

Also back in Figure 4.1, one saw that the annotated meta-level plan with its attached failure explanations together with the theory of the problem-solver are the inputs used by the last component to install the modification to the rejection heuristics. Since both heuristics are implemented as embedded code, only a single explicit generation rule is needed. Table 4.6 shows the generation rule to be added to FCPS's set of explicit search control rules.

4.2 Detailed Algorithm Description

4.2.1 Identify Rejection Heuristics to be Modified (IRHM)

In order to identify the rejection heuristics that are to be modified, SHAPeS uses a theory of the problem-solver to search for a meta-level plan that describes how the problem-solver could solve the given problem if it were allowed to ignore some of its rejection heuristics (i.e., those rejection heuristics that will be modified). However, if such meta-plans were cheap to find then rejection heuristics would not be needed in the first place. Unfortunately, the search space is usually too large to solve all problems cheaply. Since the rejection heuristics are being ignored, the search space is back to its original size. Thus a way is needed to reduce the search space to a size that is relatively cheap to search. This is done by having the user supply a solution and then transforming the supplied solution into a set of (solution) constraints that reduce the cost of finding the meta-level plan by rejecting all meta-steps that are inconsistent with these constraints.

The following section describes how the plan and its blocking steps are found for the example. The section after that describes in more detail the procedure for identifying

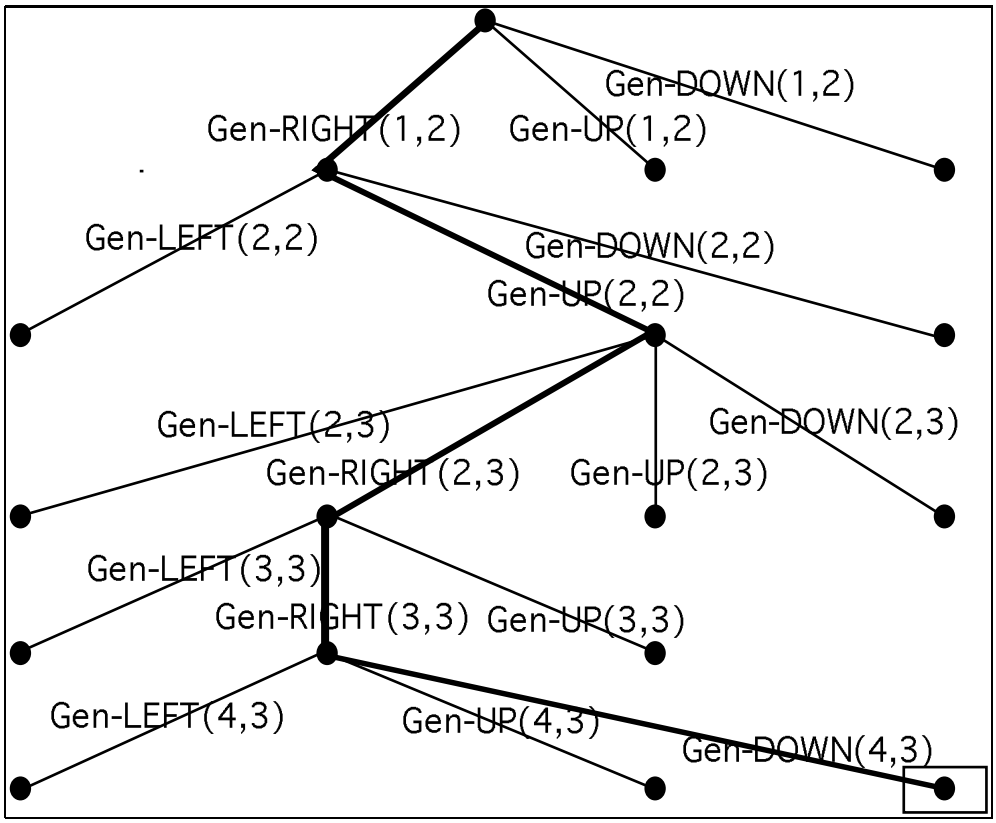

Figure 4.3: Search Tree for Meta-Level Plan to Find the Supplied Solution

the rejection heuristics to be modified.

Example Continued

In this section I show how SHAPeS identifies Rej-UP-1 and Rej-UP-2 as the rejection rules to be modified. The first step is to turn the supplied solution into constraints on the search for a meta-level plan that constructs the supplied solution. Since FCPS creates a plan by successively adding steps to the end of the current plan until the tile is finally in the goal position, the supplied solution directly constrains what step can added at each point in the search.

SHAPeS, using the supplied solution to constrain the search for the meta-level plan that constructs that solution, builds the search tree shown in figure 4.3. From the root, which has the empty plan, there are three applicable generation rules. However, only one of them produces a partial plan that is consistent with the supplied solution.

```
CREATE THE ROOT NODE OF THE SEARCH TREE FROM THE PROBLEM DESCRIPTION;
UNTIL ONE OF THE UNEXPANDED NODES MATCHES THE SOLUTION
 LOOP
     PICK ONE OF THE UNEXPANDED NODES AND MARK IT EXPANDED;
     USE THE IMPLICIT GENERATION RULES TO CREATE THE CANDIDATES;
     ELIMINATE ANY CANDIDATE THAT IS INCONSISTENT WITH THE
         SUPPLIED SOLUTION;
     MAKE ALL SURVIVING CANDIDATES CHILDREN OF THE PICKED NODE;
 END LOOP;
```

Table 4.7: Description of IRHM Algorithm

Therefore only it is expanded. SHAPeS continues to only expand nodes that are consistent with the supplied solution and eventually arrives at the node whose partial plan matches the supplied solution (this is the leaf node which is boxed). SHAPeS then checks which edge(s) would be pruned by rejection rules. In this case, the Gen-UP(2, 2) edge would be pruned by Rej-UP-1 and Rej-UP-2.

Description

Table 4.7 describes the procedure that creates the meta-plan for finding the supplied solution. I assume that there may be techniques for determining that the problem-solver cannot transform a given candidate partial solution into a given total solution. For example, if the problem-solver only extends partial solutions along a path, then if the candidate contains components that are not in the given total solution, the problem-solver will not be able to extend that partial solution into the given total solution. Where such techniques exist, I assume they will be used to prune the search space. For example, the implementation, **Bacall** (described in Chapter 6), uses the fact that PRODIGY monotonically extends partial solutions into total solutions to do such pruning.

The procedure begins by initializing the search tree from the description of the problem. The rest of the procedure simply uses its model of the problem-solver's search control theory and any constraints derivable from the solution to search for a meta-plan that creates the supplied solution. Any meta-step that is inconsistent with the set of

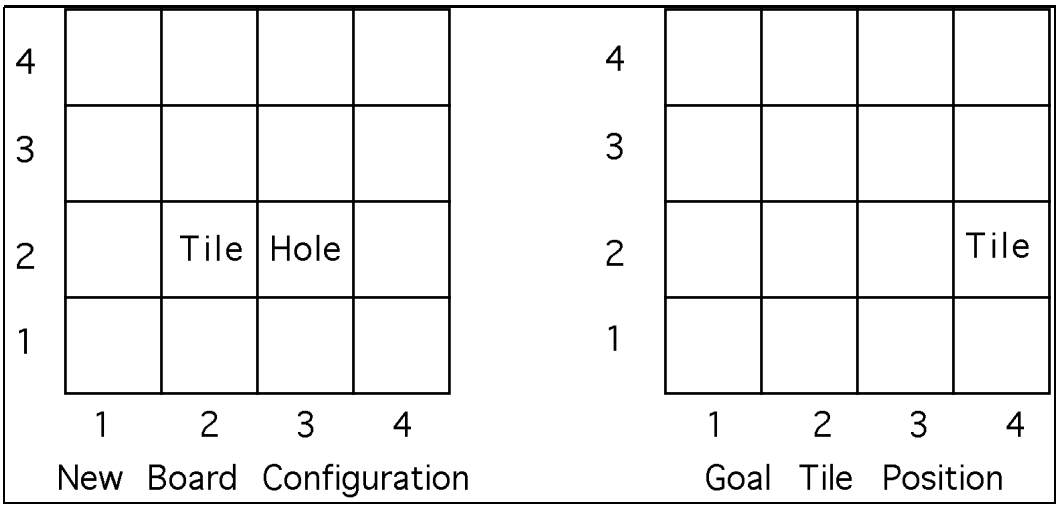

Figure 4.4: Failure Problem

derived constraints is rejected.

4.2.2 Compute Modifications

After the meta-level plan has been found and the rejection heuristics to be modified have been identified then the modifications are computed. For each edge in the meta-level plan that would have been pruned away, a modification is computed. Given such an edge, the modification is computed by using EBL to explain why using the current rejection rules causes the problem-solver to fail to find a solution from the node generating that edge.

Example Continued

Figure 4.4 shows the problem that FCPS is trying to solve at the point (i.e., the search node) where it needs to move the tile up. Figure 4.5 shows the search tree that EBL uses to explain why the FCPS cannot find a solution from this node when using its current set of rejection heuristics. In this example, the search tree is very small, all edges from this node are rejected. The leaves of the search tree are labeled with the names and instantiations of the rejection rules that rejected the edges coming into these nodes. The explanation for failure at that node (i.e., the root node of Figure 4.5) is simply the conjunction of the leaf failure explanations. This explanation is then generalized

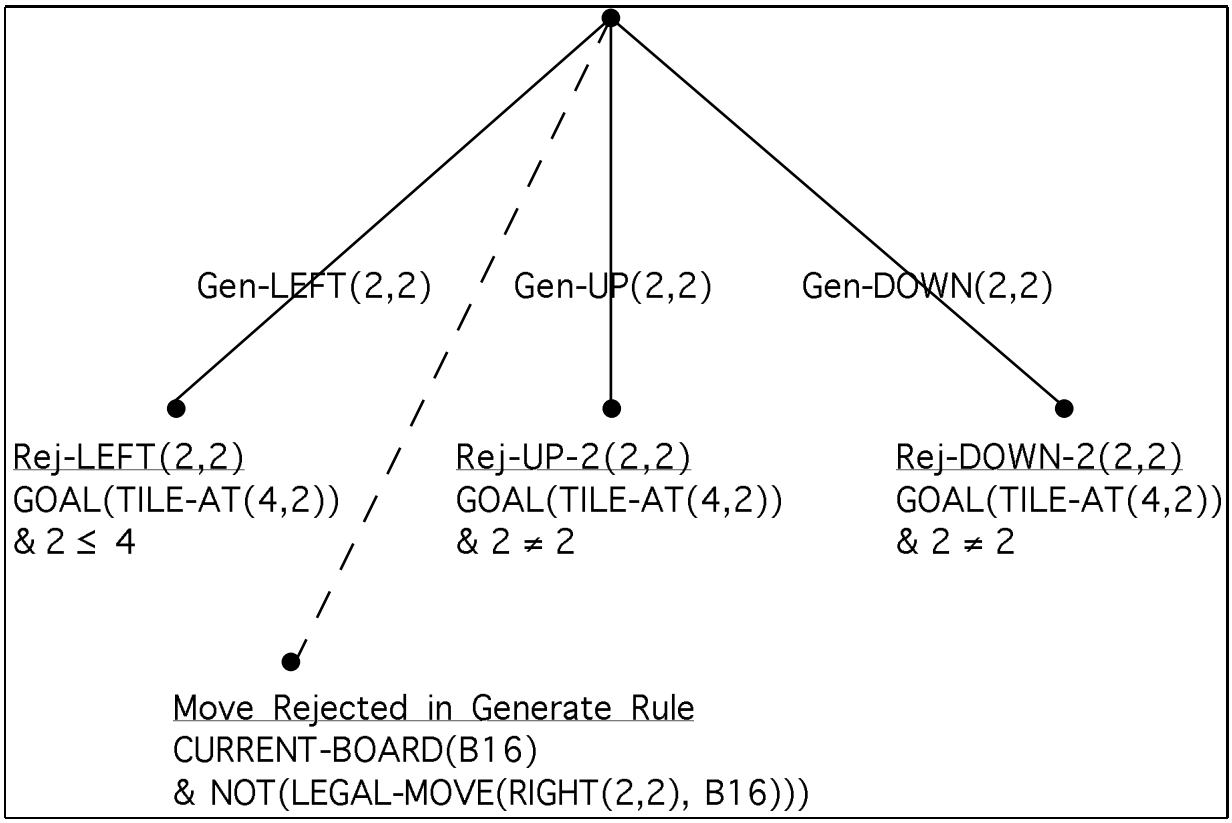

Figure 4.5: Failure Explanation's Search Tree

```
SET CURRENT-STEP TO FIRST STEP IN META-PLAN;
SET THE CURRENT SEARCH CONTROL THEORY TO BE THE PROBLEM-SOLVER'S
  SEARCH CONTROL THEORY;
LOOP UNTIL THERE IS NO STEP IN THE META-PLAN AFTER CURRENT-STEP:
     SET CURRENT-STEP TO STEP AFTER CURRENT-STEP;
     WHEN THE EDGE LEADING INTO CURRENT-STEP HAS BEEN REJECTED
       AND THE EDGE IS STILL REJECTED BY THE CURRENT SEARCH
       CONTROL THEORY
     THEN USE THE CURRENT SEARCH CONTROL THEORY TO CREATE A NEW
            SEARCH TREE, WHOSE ROOT REPRESENTS THE STATE OF THE
            PROBLEM-SOLVER AT THE TIME THE PARENT OF
            CURRENT-STEP WAS TO BE EXPANDED;
          IF THE SEARCH FOUND A SOLUTION
          THEN EXIT
          ELSE FROM THE LEAVES OF THAT SEARCH TREE, BACKPROPAGATE
                 THE FAILURE REASONS TO THE ROOT;
               IN THE META-PLAN, PUT THE GENERALIZED VERSION OF
                 THE FAILURE EXPLANATION IN THE PARENT OF
                 CURRENT-STEP;
               USE THESE FAILURE EXPLANATIONS TO UPDATE
                 THE CURRENT SEARCH CONTROL THEORY;
END LOOP;
```

Table 4.8: Computing Modifications

producing the explanation shown back in Table 4.5. The modification is that when this explanation holds then the UP tile operator should not be pruned away. The final form of this modification depends upon how the rejection heuristic is implemented in the problem-solver. I will look at this in Section 4.2.3.

Description

At this point in the general description of the SHAPeS algorithm, the algorithm has found a meta-plan that describes how the problem-solver could construct the supplied solution, assuming it could ignore some of its rejection heuristics. I will now show how SHAPeS uses that meta-plan to compute modifications for these rejection heuristics. Table 4.8 describes the algorithm for computing these modifications to the rejection heuristics that rejected edges in the meta-plan. The algorithm uses the theory of the problem-solver to generate explanations of why the problem-solver fails to solve the

given problem. The algorithm does this working forward from the beginning of the meta-plan, finding the step just before any step that would have been rejected, and constructs a new search tree using that step as the root. If the search fails to find a solution then the failure explanations at the leaves are regressed back to the root to form the modification for the rejection heuristics. If a solution is found then no more modifications are computed.

4.2.3 Install Modifications

After the modifications have been computed, they must be installed in the problem-solver. This installation depends upon how the theory is implemented in the problem-solver. I will now finish this example and then look at the procedure that does the installation.

Example Continued

SHAPeS has computed the modification for the rejected **Gen-UP(2,2)** edge. How the modification is installed depends upon how the two rejection heuristics (**Rej-UP-1** and **Rej-UP-2**) rejecting that edge, are implemented in FCPS. Since they are both implemented as system code, the modification is installed by creating one new explicit generation rule. This new generation rule adds the UP operator to the list of alternatives at a node when UP would have been a candidate and rejected by **Rej-UP-1** and **Rej-UP-2**. Table 4.9 shows the generation rule to be added to FCPS's search control rules. The conjuncts in the preconditions come from simplifying the union of the failure explanation and the preconditions for generating UP(?ROW,?COL). Specifically, conjuncts 1-5 come from the failure explanation, and conjuncts 1-2 also comes from the precondition for generating UP(?ROW,?COL).

If either had been implemented as an explicit rejection rule then one would also need to modify that rejection rule. For example, if **Rej-UP-1** was implemented as an explicit search control rule then the modification would be installed by adding the negation of the modification to **Rej-UP-1**'s preconditions. Table 4.10 shows what the new preconditions would have been for **Rej-UP-1**.

If (1) CURRENT-BOARD(?B)
& (2) LEGAL-MOVE(UP(?CURRENT-ROW, ?COL), ?B)
& (3) NOT(LEGAL-MOVE(RIGHT(?CURRENT-ROW, ?COL), ?B))
& (4) GOAL(TILE-AT(?GOAL-ROW, ?COL))
& (5) ?CURRENT-ROW $\leq$?GOAL-ROW
Then GENERATE MOVE UP(?CURRENT-ROW, ?COL)

Table 4.9: New Generation Rule to be Added

CANDIDATE(UP(?CURRENT-ROW, ?CURRENT-COL)
$\wedge$ GOAL(TILE-AT(?GOAL-ROW, ?GOAL-COL))
$\wedge$?CURRENT-COL $\geq$?GOAL-COL
$\wedge$ $\neg$(CURRENT-BOARD(?B)
$\wedge$ TRUE(TILE-AT(?CURRENT-ROW, ?CURRENT-COL), ?B)
$\wedge$ GOAL(TILE-AT(?GOAL-ROW, ?CURRENT-COL))
$\wedge$ NOT(LEGAL-MOVE(RIGHT(?CURRENT-ROW, ?CURRENT-COL), ?B))
$\wedge$?CURRENT-ROW $<$?GOAL-ROW)

Table 4.10: Installed Precondition for Rejection SCR

Description

SHAPeS modifies the edge-level behavior of rejection heuristics. Before describing the installation procedure I would like to clarify the difference between edge-level and node-level modifications. The implicit rejection rules **Rej-RIGHT**, **Rej-LEFT**, **Rej-UP-1**, and **Rej-DOWN-1** all reject their associated moves because they increase the distance between where the tile is now and where it is supposed to end up. Assume that instead of these four operator specific rules, one had written one general rule that rejected any move that increased the distance between the tile's current position and its goal position. Such a search control rule is shown in Table 4.11. Also assume that it had been implemented as an explicit search control rule. With these changes to the problem-solver and its theory, the example would have generated a slightly different failure explanation but would be essentially the same as before. If one wanted to modify this rejection heuristic at the node level then one would simply add the negation of the failure explanation to the reject rule's precondition. Table 4.12 shows the form of the new rule. Let N be the name of the node in the search space where the general rejection heuristic blocked the meta-level path to the given solution. Given this modification then

If CANDIDATE(?OP)
& CURRENT-BOARD(?B)
& ?CUR-POS = TILE-POSITION(?B)
& ?GOAL-POS = TILE-POSITION(ADD-OP(?OP,?B))
& GOAL(TILE-AT(?ROW, ?COL))
& DISTANCE((?ROW, ?COL), ?CUR-POS)
< DISTANCE((?ROW, ?COL), ?GOAL-POS)
THEN REJECT MOVE ?OP

Table 4.11: General Difference Reduction Heuristic

If CANDIDATE(?OP)
& NOT(*FAILURE-EXPLANATION*)
& CURRENT-BOARD(?B)
& ?CUR-POS = TILE-POSITION(?B)
& ?GOAL-POS = TILE-POSITION(ADD-OP(?OP,?B))
& GOAL(TILE-AT(?ROW, ?COL))
& DISTANCE((?ROW, ?COL), ?CUR-POS)
< DISTANCE((?ROW, ?COL), ?GOAL-POS)
THEN REJECT MOVE ?OP

Table 4.12: Node-Level Modification of Heuristic

this general rule would not have rejected any move at N (before its modification the rule had rejected the LEFT, UP, and DOWN operators). However, if one only wants to modify the rejection heuristic for the edge that was on the meta-level path then one would also add a test that ?OP was *not* the UP operator. Table 4.13 shows the form of this modified rule. Given this extended modification then this general rule would still have rejected the LEFT and DOWN operators at N. If one had assumed that the general rejection rule had been implemented as embedded code then installing the edge-level modification would have been more straightforward but installing the node-level modification would have been more complicated.

If CANDIDATE(?OP)
& NOT(*FAILURE-EXPLANATION*)
& NOT(?OP = UP(?X, ?Y))
& CURRENT-BOARD(?B)
& ?C = TILE-POSITION(?B)
& ?F = TILE-POSITION(ADD-OP(?OP,?B))
& GOAL(TILE-AT(?X, ?Y))
& DISTANCE((?X, ?Y), ?C) ≤ DISTANCE((?X, ?Y), ?F)
THEN REJECT MOVE ?OP

Table 4.13: Edge-Level Modification of Heuristic

FOR EACH COMPUTED MODIFICATION
DO
    WHENTHECOMPUTED MODIFICATION AFFECTS ANY REJECTION
        HEURISTICS IMPLEMENTED IN PROBLEM-SOLVER CODE

THEN CREATE A NEW GENERATION SEARCH CONTROL RULE
            WHOSE POSTCONDITION IS THE APPROPRIATELY VARIABLIZED
            ALTERNATIVE AND WHOSE PRECONDITIONS IS COMPUTED AS
            DESCRIBED IN TABLE 4.16;
    WHEN THE COMPUTED MODIFICATION AFFECTS ANY REJECTION
        HEURISTICS IMPLEMENTED AS A SEARCH CONTROL RULE
    THEN FOR EACH AFFECTED SEARCH CONTROL RULE
        MODIFY THE PRECONDITION OF THAT SEARCH CONTROL RULE
        AS SHOWN IN TABLE 4.15;
END DO;

Table 4.14: Installing the Modification Set

NP = EP $\wedge \neg$ (CM $\wedge$ TA)
WHERE NP is the new precondition,
        EP is the existing precondition,
        CM is the computed modification.
        TA isa test that alternative specified in the
            postcondition is the alternative to be added
            back into the search space.

Table 4.15: Computing the Preconditions for Modified Rejection Search Control Rules

NP = GP $\wedge$ CM $\wedge \neg$ (RP1 $\vee \ldots \vee \ldots$ RPn)
WHERE NP is the new preconditions,
        GP is the preconditions of the original generation rule
            for the alternative to be generated,
        CM is the computed modification,
        RPi is the precondition of the i^{th} rejection rule
            implemented in code that might reject the
            alternative to be generated but didn't in the
            meta-level path.

Table 4.16: Computing the Preconditions for Generation-Type Modifications

Table 4.14 shows the pseudo-code for installing the computed modification. The computed modifications describe when their associated rejection heuristics should not prune the alternative used in the supplied solution, for example, I only want the modification to apply to the appropriate edge and not to the entire node. Installing the modification to a rejection heuristic implemented in the problem-solver as a search control rule is relatively straightforward. Table 4.15 shows the pseudo-code for computing the new preconditions for the rejection search control rule. I want the rejection rule not to apply to the alternative when the computed modification conditions are satisfied, otherwise it should behave just as it did before. Therefore I modify the rule's precondition by and-ing in the negation of the computed modification with the negation of a test that the rule is checking the appropriate alternative. I need to do this to each rejection rule (implemented as a search control rule) associated with the computed modification.

Installing a modification to a rejection heuristic implemented in code is slightly more difficult. It requires a new generation search control rule to be added to the problem-solver's set of search control rules. Table 4.16 shows the pseudo-code for computing the preconditions for the new generation rule. To only add in the appropriate alternative, the postcondition of the new generation rule is set to the proper variablization of that alternative. In addition to only adding in the appropriate edge, I also do not want the modification to interfere with any of the other rejection heuristics implemented in code. This is accomplished by and-ing in the computed modification with the negation of the disjunction of the preconditions of all the other rejection heuristics (implemented in code) that can (but did not) reject this alternative. Only one generation rule is added regardless of the number of rejection heuristics (implemented as code) associated with the modification.

4.3 Design Alternatives for Specializing Rejection Rules

I assume that the rejection rule is represented as an IF-THEN rule of the following form:

```
IF p(?NODE, ?ALTERNATIVE)
```

```
THEN REJECT ?ALTERNATIVE,
```

where **p** is the rejection test, **?NODE** is the search node being expanded, and **?ALTERNATIVE** is the candidate under consideration. I assume that the relaxation of the rejection heuristic is of the form:

```
 IF p(?NODE, ?ALTERNATIVE) AND NOT(r(?NODE, ?ALTERNATIVE))
THEN REJECT ?ALTERNATIVE,
```

where **r** is the specialization. When a specialization causes a rejection rule to no longer reject an edge, I say that the rejection heuristic has been *suspended* or *relaxed* at that edge.

There are a number of different design choices for specializing the rejection rule. Four of them are: justification, granularity, implementation, and the trigger for the specialization. I will discuss these below.

4.3.1 Justification

One design choice is the type of rationale used to justify specializing the rejection rule. Specifically, what does one want **r** to test for? Essentially, if one ignores the cost-effectiveness of the specialization, then the ideal justification for specializing a rejection rule is that the current rejection rule prevents the problem-solver from finding the desired solution and the specialization should enable the problem-solver to find it without unduly expanding the search space. Thus, the ideal justification is composed of both the failure explanation and the success explanation. If only the failure component is present then the rejection heuristic might be suspended when there is no reason to believe that the suspension will lead to success (except for the fact that it did so for at least one problem). If only the success component is present then the rejection heuristic might be suspended when a solution was already going to be found, and perhaps increase the cost of reaching it.

Unfortunately, the ideal of using both components may not be cost effective, i.e., the cost of creating and/or using it may exceed its benefits. For example, a failure

explanation[3] is usually expensive to compute and often expensive to test. While a success explanation is cheap to compute and often cheap to test[4], it is usually much more specific. The problem with a much more specific explanation is that if the underlying reason for the specialization is very general then one may need to generate a large number of special case specializations to attain the desired coverage. Thus the cost of computing and testing all the specialized success explanations could exceed the costs of computing and testing the equivalent coverage of a failure explanation. Since the ideal specialization combines both the failure and success explanations, it would have the worst cost features of both. The alternative is to approximate the failure and/or the success explanations. If one views the specialization rationale as having two axes, one being the fidelity of the success approximation to the full success explanation and the other being the fidelity of the failure approximation to the full failure explanation, then this ideal is one of the extreme positions. There are three other extreme positions:

- Use neither the success nor the failure explanations, i.e., **r** is always true.
- Use only the success explanation.
- Use only the failure explanation.

Using neither explanation is the same as removing the rejection rule. As I showed in Chapter 2, the FS2 approach just uses success explanations, while the SOAR approach just uses a characterization of the failure conditions. SHAPeS primarily uses the failure explanations.

I expect that there is no one combination of approximations of success and failure explanations that create the most cost-effective specialization of rejection rules for all domains. However, this is an area that requires more research.

---

[3]While I have been assuming that the specialization test has been generated via an explanation-based process (i.e., Explanation Based Learning[14]), induction techniques might also be used.

[4]Success explanations are usually cheap to test because the variables are existentially quantified while the tests in failure explanations usually contain some variables that are universally quantified.

4.3.2 Granularity

Another design choice is: Once the system has decided that a rejection heuristic should be suspended, for what part of the search should it be suspended? The obvious choices are: for the entire domain, for the entire problem, for a subtree of the search tree, for that node in the search tree, for that edge in the search tree. The system could test the domain definition for indications that using the rejection rule in that domain would prevent the problem-solver from solving most problems in that domain. SOAR effectively adopts the second approach, by selecting from a sequence of problem-solvers, one that does not use that rejection rule.

In general, it is not clear how to implement a subtree grain-size suspension. However, implementing a node grain-size suspension is straightforward. **?ALTERNATIVE** would be left a free variable, then the rejection heuristic would be suspended for all alternatives at nodes satisfying the specialized preconditions. Implementing edge grain-size suspension is also straightforward, **?ALTERNATIVE** is simply instantiated to the variablized meta-level step used in computing the specialized preconditions. FS2 adopts this last approach by appropriately instantiating **?ALTERNATIVE** and suspending the rejection rule only for edges that satisfied the specialized preconditions. SHAPeS also uses this last approach.

If one assumes that the rejection rule is approximately correct (i.e., that it does not usually cause a solvable problem to become unsolvable) then one wants the specialization to suspend the rejection heuristic only as much as is necessary to regain lost coverage. However, if one assumes that the rejection rule is only causing solvable problems to become unsolvable then one wants to totally specialize (i.e., remove) the rejection rule's preconditions.

4.3.3 Implementation

Another design choice is the method of effecting the suspension. As I mentioned earlier in this chapter, some of the rejection rules may be embedded within the problem-solver's code while other rejection rules may be implemented as explicit IF-THEN

search control rules. The system may choose to only suspend the embedded rejection rules, only the explicit search control rules, or both. SOAR, in a sense, avoids this decision by implementing the suspension by selecting between problem-solvers based upon the rejection rules they use. FS2 only suspends its explicitly learned rejection rules by adding the new test conditions to the rejection rule's preconditions. SHAPeS can specialize both embedded rejection rules (by adding explicit generation rules) and explicit rejection rules (by modifying the explicit rejection rules' preconditions).

4.3.4 Triggering Rejection Rule Specialization

The last design choice I will look at is that of when a specialization is computed. Essentially, the only time to specialize a rejection rule is when the system has detected that the rejection rule is preventing the problem-solver from solving a problem. SOAR detects this by attempting to solve a problem with the most constrained (i.e., has the most rejection rules) problem-solver and if it fails to solve the problem falling back to successively less constrained problem-solvers until it either fails with a complete problem-solver (i.e., the problem is unsolvable) or it finds a problem-solver that solves the problem. SOAR then uses the fact that each of the preceding problem-solvers were unable to solve the problem as the basis for learning to avoid using those problem-solvers for similar problems in the future.

FS2 uses a three-step process to detect when a rejection rule is preventing the problem-solver from solving a solvable problem. The first step is deciding that a rejection rule may be preventing the problem solver from finding a solution. This occurs when either the problem-solver has exhausted its search space or a predefined number of meta-steps has occurred since the last time a goal was achieved or a rejection rule was overridden. The second step is deciding which node , pruned by a rejection rule, is most likely to be on a path to a solution. FS2 chooses that node which has the most top-level goals true and the shortest partial plan. At that node, FS2 overrides those rejection rules[5] that pruned this node and adds that node back into the search space

[5]These rejection rules are still in force elsewhere in the search tree.

and continues the search. The third step occurs when FS2 determines that progress has been made (i.e., that one of the top-level goals has been achieved). From this node in the problem-solver's meta-level solution path, FS2 determines which rejection rules were overridden. FS2 then computes specializations for those rejection rules at the nodes where they were overridden.

Unlike both SOAR and FS2 which successively suspend their rejection rules until they either find a solution or that the problem is unsolvable, SHAPeS relies on an external agency to tell it that the problem its problem-solver just failed to solve is actually solvable. Furthermore, SHAPeS expects the external agency to supply it with a solution. SHAPeS then uses that supplied solution to constrain its search for a meta-level path that constructs that solution. When SHAPeS finds such a path, it, like FS2, determines which rejection rules must be overridden to find this solution and then computes the specializations for those rules at the nodes where they were ignored.

If the problems that the problem-solver is attempting to solve are complex then there is little reason to believe that the problem-solver can find a solution within this extended search space in an acceptable amount of time. If one assumes that the user is a domain expert (or has access to a domain expert) then the user may prefer to interactively guide the problem-solver to an acceptable solution rather than wait for the problem-solver to successively expand its search space looking for a solution. Thus SHAPeS has adopted this learning apprentice approach[15].

Chapter 5

Analysis

In this chapter I discuss my correctness, generality, and computational complexity claims about the SHAPeS algorithm. In these discussions of correctness, I will analyze the effect of modifying the problem-solver's search control logic upon its coverage. This will be easier to talk about if I base my discussion upon the modifications to the theory of the problem-solver. However, interactions among the theory's search control rules complicates even this analysis. Therefore before analyzing the correctness of the SHAPeS algorithm, I first discuss some of the search control rule interactions that complicate such analyses. I also propose some conditions that are sufficient for eliminating these interactions. I will assume that these conditions hold for this discussion, and thus simplify the analysis tasks.

5.1 Analysis of Search Control Rule Interactions

There are many ways that search control rules can interact and these interactions often make it quite difficult to predict what happens to a problem-solver's coverage as new rules are added or as old rules are modified. While one cannot eliminate all interactions, there are some ways to eliminate certain types of obviously undesirable interactions. This section looks at three specific types of undesirable rule interactions and at some ways they can be avoided.

5.1.1 In Situ Candidate-Set-Updating Interactions

There are at least two approaches to updating candidate sets. One approach is for the system to make a copy of the candidate set at the beginning of each phase[1] of a decision procedure. During the phase the preconditions of the search control rules query the candidate set while the postconditions update the copy. At the end of the phase, the system makes the updated copy the new candidate set that is passed on to the next phase. We'll call this the *candidate-set-copy updating* approach. The other approach is not to make a separate copy of the candidate set but for the system to update the candidate set *in situ*, i.e., in place. When candidate sets are updated in situ then the contents of a candidate set can change each time a search control rule is fired. If the search control rules query the contents of their candidate set then the result of firing that set of search control rules can depend upon the order in which the search control rules fired.

If the system learns new search control rules, and the order the rules fire is determined by the order in which they were learned, then the system can behave differently depending upon the order of the training problems. This obviously complicates any analysis of the effect of the learning upon the problem-solver. Candidate-set-copy updating does not cause this problem.

5.1.2 Candidate-Set-Testing Interactions

Another type of interaction, *candidate-set-testing* interactions, occurs when search control rules query the contents of candidate sets. For example, assuming one has a problem-solver that has decision procedures that choose which goal to work on next and that choose which operator to use to achieve the selected goal, then this interaction could occur if an operator rejection rule queries the contents of the goal candidate set. In which case, adding, modifying, or removing a goal search control rule could affect the application of some operator rejection rules. This can lead to the situation

[1]Remember a decision procedure is all the logic associated with making a particular decision, e.g., deciding which operator to use to achieve a particular goal. In this framework, a decision procedure has four phases: an embedded generation phase, an embedded rejection phase, a rule-based generation phase, and a rule-based rejection phase.

```
CREATE A ROOT NODE CONTAINING AN INITIAL PLAN WHICH CONSISTS
        SOLELY OF A PSEUDO-STEP WITH THE PROBLEM'S GOALS AS ITS
        PRECONDITIONS;
SET OPEN STACK TO CONTAIN THE ROOT NODE FOR THIS SEARCH SPACE;
LOOP FOREVER
        WHEN THE OPEN STACK IS EMPTY THEN EXIT WITH FAILURE;
        POP NODE OFF OF OPEN STACK TO BECOME THE CURRENT NODE;
        WHEN THE CURRENT NODE IS A SOLUTION THEN EXIT WITH THE
                SOLUTION;
        USE THE CHOOSE-GOALS DECISION PROCEDURE TO PRODUCE
                THE CANDIDATE GOAL SET
        FOR EACH GOAL IN THE CANDIDATE GOAL SET
                USE THE CHOOSE-OPERATORS DECISION PROCEDURE
                        TO PRODUCE CHILDREN FOR THIS NODE;
        PUSH ALL NEW CHILDREN ONTO THE OPEN STACK;
END LOOP.
```

Table 5.1: Pseudo-Code for BCPS Problem-Solving Procedure

```
OP1:
Preconditions: P2
Adds: G1

OP2:
Adds: G2
```

Table 5.2: Abstract Domain Definition 1

where, for example, adding a rejection rule actually causes the search space to expand or conversely removing a rejection rule can cause the search space to shrink.

To illustrate this I will need a more sophisticated problem-solver than FCPS. Table 5.1 shows the problem-solving procedure for a backward-chaining problem-solver, BCPS. BCPS uses two decision procedures: CHOOSE-GOALS and CHOOSE-OPERATORS. CHOOSE-GOALS creates a set of candidate goals. BCPS then uses CHOOSE-OPERATORS to create a set of candidate operators for each goal in the candidate goal set. Each of these operators constitute a way of continuing the current node's plan by adding that operator to the beginning of the current node's plan. Each continuation is stored as a child node of the current node. A plan is a solution when all of the plan's unsatisfied preconditions are true in the problem's initial state.

Goal Generation Search Control Rules
If CURRENT-PLAN(?P)
    & UNSATISFIED-PRECONDITION-IN(?G, ?P)
Then GENERATE GOAL ?G

Goal Rejection Search Control Rules

Table 5.3: CHOOSE-GOALS Decision Procedure

Operator Generation Control Rules

If CURRENT-PLAN(?P)
    & CURRENT-GOAL(?G)
    & OPERATOR(?OP)
    & HAS-ADD(?OP, ?G)
    & NOT(UNSATISFIED-PRECONDITION-IN(?X, ?P)
        & HAS-DELETE(?OP, ?X))
Then GENERATE OPERATOR ?OP

Operator Rejection Control Rules
If CANDIDATE-OP(?OP)
    & CURRENT-GOAL(G1)
    & CANDIDATE-GOAL(G2)
Then REJECT OPERATOR ?OP

If CANDIDATE-OP(?OP)
    & CURRENT-GOAL(G2)
    & CANDIDATE-GOAL(G1)
Then REJECT OPERATOR ?OP

Table 5.4: CHOOSE-OPERATORS Decision Procedure

Goal Description: G1 & G2
Initial State: P2

Table 5.5: Example Problem

If CURRENT-PLAN(?P)
    & CANDIDATE-GOAL(G1)
    & CANDIDATE-GOAL(G2)
Then REJECT GOAL G1

Table 5.6: Goal Rejection Rule that Enables BCPS to Solve Example Problem

Table 5.2 defines an abstract domain to illustrate candidate-set-testing interactions. There are two operators: OP1 and OP2. Tables 5.3 and 5.4 show the CHOOSE-GOALS and CHOOSE-OPERATORS decision procedures specialized to this abstract domain. CHOOSE-GOALS generates a goal candidate set containing all the plan's unsatisfied preconditions. CHOOSE-OPERATORS has been crafted to ensure that BCPS cannot solve the problem shown in Table 5.5, which would normally be easily solvable. The rejection rules in CHOOSE-GOALS reject all operators if the current goal is either **G1** or **G2** and the other goal is in the goal candidate set. Thus regardless of which goal is chosen first, all operators will be rejected and no solution can be found. If one adds the goal rejection rule shown in Table 5.6 to CHOOSE-GOALS then BCPS will be able to solve the problem. However, if all the rejection rules had queried the solution structure[2] instead of the candidate sets, then adding this goal rejection rule would not have enabled BCPS to solve the problem. In this case, where the rules query the candidate sets, I show that adding a rejection rule causes the search space to expand.

As I will show in Section 5.2.1, this type of interaction can be avoided if the preconditions of the search control rules only query the state of the partial solution rather than being allowed to query the contents of other decision procedure's candidate sets. This restriction also avoids candidate-set-testing interactions.

[2]For example, the rules could have queried whether **G1** or **G2** were unsatisfied in the current partial solution.

OP3:
Adds: G3, G4

Table 5.7: Abstract Domain Definition 2

(Goal Rejection Control Rules)
G3 Goal Reject Rule
If CANDIDATE-GOAL(G3)
    & CANDIDATE-GOAL(G5)
Then REJECT GOAL G3

G4 Goal Reject Rule
If CANDIDATE-GOAL(G4)
    & CANDIDATE-GOAL(G5)
Then REJECT GOAL G4

Table 5.8: Goal Reject Rules for BCPS's CHOOSE-GOALS Decision Procedure

Problem:
Goal Description: G3 & G4 & G5
Initial State: G5

Table 5.9: Redundant Path Example Problem

5.1.3 Redundant Path Interactions

Even if both candidate-set-testing and in situ candidate-set-updating interactions are eliminated, there are still other interactions that can occur between search control rules. In particular, rejection rules that individually *sanction* (i.e., do not reject) the same solutions need not still sanction those solutions when put together. This particular type of interaction is called a *redundant path* interaction. Redundant path interactions occur because rejection rules are not rejecting object-level paths (e.g., plans) per se, but are actually rejecting meta-level paths. Thus, if there are two meta-level paths to a solution, where one rejection rule rejects one meta-level path to that solution while the other rule rejects the other meta-level path, then, while individually they both sanction that object-level path, they eliminate it when put together.

I now show an example of two reject rules where neither individually rejects the

problem's solution but when they are both present they do reject the solution. Table 5.7 shows the abstract domain definition used to illustrate this type of interaction. Table 5.8 shows the two rejection rules and Table 5.9 shows the problem. The solution is simply the plan containing operator **OP3**. When both rules are present the problem is unsolvable for BCPS because there are no operators that achieve **G5**, therefore **G5** will remain a goal candidate, and consequently the goal reject rules reject both **G3** and **G4**. Therefore BCPS will fail to solve this problem when these two rules are present. However, if one removes either rule then BCPS can solve the problem. For example, if one removes the *G3 Goal Reject Rule* then the CHOOSE-GOALS decision procedure will produce a candidate set containing **G3** and **G5**. The CHOOSE-OPERATORS decision procedure will then select OP3 to achieve **G3**, and the problem is solved.

Note, however, if the search procedure is systematic[16, 12] (i.e., there is never more than one meta-level path to any partial solution) then if the rejection rules separately sanction a solution then they will sanction that solution jointly.

5.1.4 Summary

I have identified three types of undesirable search control rule interactions that can occur in BCPS, namely: (1) in situ candidate-set-updating; (2) candidate-set-testing; and (3) redundant path interactions. Candidate-set-testing interactions occur when the rejection and/or selection rules' preconditions query the contents of the candidate set. Writing the search control rules' preconditions so that they only query the state of the partial solution eliminates this type of interaction. When candidate-set-testing interactions occur, the effect of adding a search control rule is very difficult to predict. For example, adding generation rules can cause the problem-solver to become unable to solve certain problems, while adding reject rules can cause the opposite effect. If, in addition to querying the contents of the candidate set, the candidate set is updated in place then in situ candidate-set-updating interactions occur. This can cause the problem-solver to behave differently depending upon the order that phase's search control rules are applied. Both (1) querying the state of the partial solution and (2) updating the candidate set separately from the copy of the candidate set that

is being queried, will eliminate this type of interaction. Redundant path interactions occur when the problem-solver does not systematically explore the search space, i.e., there are multiple meta-level paths to the same partial solution. Redundant path interaction make it difficult to use the behavior of individual rejection rules to predict their behavior when combined, specifically predicting when adding a reject rule to a rule set will cause the problem-solver to become unable to solve a problem. Redundant path interactions can be eliminated by having the problem-solver systematically explore the search space.

When these three types of interactions are eliminated the effects of adding search control rules become more predictable. In particular, the following become true:

- Adding new generate search control rules never decreases the problem-solver's coverage and adding reject search control rules never increases it.
- Specializing the preconditions of a generate search control rule never increases the problem-solver's coverage and specializing the preconditions of a reject search control rule never decreases the coverage.
- The effect of the search control rules becomes independent upon the order they are applied within their individual phases.
- For a given problem, if a new reject rule and the current set of reject rules have solutions in common that they sanction then adding that new reject rule will not cause the problem-solver to be unable to solve that problem. Otherwise, adding the new reject rule will cause the problem-solver to fail to solve that problem.

5.2 Correctness

I will discuss two types of correctness. The first is the correctness of the preserved coverage, i.e., when is it guaranteed that SHAPeS will not prune away any solutions that were in any problem's search space created by the original search control theory? The second type of correctness is the correctness of the extended coverage, i.e., when is it guaranteed that SHAPeS will enlarge the original problem's search space

so that it contains a solution to that problem? Notice that neither of these types of correctness talk about whether the problem-solver will now return the given solution nor about whether the problem-solver will now be able to solve the problem. Whether the problem-solver can solve the problem, after SHAPeS has modified its search control theory, depends upon both the topology of the search space and how the problem-solver traverses the search space. For example, the search tree might have infinite paths and the problem-solver might do depth-first search.

5.2.1 Coverage Preservation Correctness

Claim 5.2.1 (Correctness of Search Space Preservation) *Given a problem-solver, a theory of its search control logic, a problem, an EBL learner, and a solution to that problem, such that:*

1. *The theory of the problem-solver's search control logic (used by the learner) only references the structure of the node's partial solution.*
2. *The learner's EBL's operationality criterion is grounded in the structure of the partial solution.*

then any node that was in the problem-solver's search space for any problem before the problem-solver's search control theory is modified is still in the problem-solver's search space after the theory has been modified.

If the premises in Claim 5.2.1 hold, then modifications that explicitly add candidates to the set of alternatives at search space nodes cannot cause any nodes within the search space to be pruned away. SHAPeS modifications only explicitly add candidates to the set of alternatives because the modifications only add generation search control rules and/or specialize existing rejection search control rules. By "explicitly" I mean the direct action of the modification, however, the modification may indirectly cause alternatives to be pruned away. I will now argue that if the premises in Claim 5.2.1 hold then this cannot be so.

The first premise in the Claim is that the preconditions of all rules in the search control theory only reference the structure of the node's partial solution. When this

holds, it means that given the same partial solution, then neither adding new generation rules nor specializing existing rejection rules will prune away any of the old alternatives for extending this partial solution. Since the initial partial solution is the same both before and after the search control theory has been modified then all nodes that were in the search space before the modification will still be there after the modification. Therefore modifying the search control theory will not eliminate any solution paths that were in the problem-solver's search space before the modification.

The second premise in the Claim is that the learner's operationality is grounded in the partial solution structure, i.e., that all the predicates that are in the preconditions of the new generation rules or of the newly specialized rejection rules either test the structure of the node's partial solution or instantiate the alternative mentioned in the rule's postcondition. When the second premise holds then if the first premise holds before the modification then it will also hold after the modification, i.e., these modifications will not cause the updated theory of the problem-solver to violate the first premise.

5.2.2 Coverage Extension Correctness

Claim 5.2.2 (Correctness of Coverage Extension) *Given a problem-solver, a theory of its search control logic, a problem, an EBL learner, and a solution to that problem such that:*

1. *The given solution is in the problem-solver's implicit generation space.*

2. *The terminal node failure reasons are regressable over the failed search tree.*

3. *All of the conditions for coverage preservation hold.*

4. *The problem-solver allows search control rules to be added/modified for its decision procedures and executes the rules after the search control implemented in system code has been executed.*

5. *The failure explanations are operationalizable.*

```
CREATE THE ROOT NODE OF THE SEARCH TREE FROM THE PROBLEM DESCRIPTION;
UNTIL ONE OF THE UNEXPANDED NODES MATCHES THE SOLUTION
 LOOP
     PICK ONE OF THE UNEXPANDED NODES AND MARK IT EXPANDED;
     USE THE IMPLICIT GENERATION RULES TO CREATE THE CANDIDATES;
     ELIMINATE ANY CANDIDATE THAT IS INCONSISTENT WITH THE
         SUPPLIED SOLUTION;
     MAKE ALL SURVIVING CANDIDATES CHILDREN OF THE PICKED NODE;
 END LOOP;
```

Table 5.10: Description of IRHM Algorithm

then SHAPeS will modify the problem-solver's search control theory such that the problem now has a solution in the problem-solver's explicit search space.

Claim 5.2.2 describes the conditions I claim guarantee coverage extension correctness. I will now go through each of SHAPeS's three main procedures and show how this is guaranteed. First I show that *Identify Rejection Heuristics to be Modified* is guaranteed both to find a meta-level path that constructs the given solution and to identify the rejection heuristics that prevented the problem-solver from traversing that path. Then I show that given the path and identified rejection heuristics that *Compute Modifications* further refines the identification to those rejection heuristics (and their meta-level steps) which it is sufficient to modify to extend the explicit search space to include a solution. I then show that *Compute Modifications* computes conditions for each of these rejection heuristics so that they will not prevent the problem-solver from finding a solution. Last, I show that *Install Modifications* correctly uses the computed modifications to modify the problem-solver so that a solution is now in its explicit search space.

Extension Correctness of IRHM

Table 5.10 shows the *Identify Rejection Heuristics to be Modified*(IRHM) logic. Assuming the given solution is in the problem-solver's implicit generation space then the theory's implicit generation rules can generate a path to that solution. Since there exists a path to that solution, then one can use a complete (i.e., one that search the

```
SET CURRENT-STEP TO FIRST STEP IN META-PLAN;
SET THE CURRENT SEARCH CONTROL THEORY TO BE THE PROBLEM-SOLVER'S
  SEARCH CONTROL THEORY;
LOOP UNTIL THERE IS NO STEP IN THE META-PLAN AFTER CURRENT-STEP:
     SET CURRENT-STEP TO STEP AFTER CURRENT-STEP;
     WHEN THE EDGE LEADING INTO CURRENT-STEP HAS BEEN REJECTED
       AND THE EDGE IS STILL REJECTED BY THE CURRENT SEARCH
       CONTROL THEORY
     THEN USE THE CURRENT SEARCH CONTROL THEORY TO CREATE A NEW
            SEARCH TREE, WHOSE ROOT REPRESENTS THE STATE OF THE
            PROBLEM-SOLVER AT THE TIME THE PARENT OF
            CURRENT-STEP WAS TO BE EXPANDED;
          IF THE SEARCH FOUND A SOLUTION
          THEN EXIT
          ELSE FROM THE LEAVES OF THAT SEARCH TREE, BACKPROPAGATE
                 THE FAILURE REASONS TO THE ROOT;
               IN THE META-PLAN, PUT THE GENERALIZED VERSION OF
                 THE FAILURE EXPLANATION IN THE PARENT OF
                 CURRENT-STEP;
               USE THESE FAILURE EXPLANATIONS TO UPDATE
                 THE CURRENT SEARCH CONTROL THEORY;
END LOOP;
```

Table 5.11: Computing Modifications

entire space) search method (e.g., iterative-deepening) to find that path. If there is any way to determine that a given node cannot lead to the supplied solution then that node should not be expanded. Once a path to the solution has been found, it is easy for SHAPeS to check at each meta-level step to identify which rejection heuristics would apply to that step. Therefore given these assumptions *Identify Rejection Heuristics to be Modified* will both find a meta-level path that constructs the given solution and identify the rejection heuristics that prevented the problem-solver from traversing that path.

Extension Correctness of Compute Modifications

Table 5.11 shows the *Compute Modifications* procedure. It starts with the first meta-level step in the path found by IRHM that has an associated rejection heuristic. Using the current theory of the problem-solver's search control logic (including the explicit

FOR EACH COMPUTED MODIFICATION
DO
WHEN THE COMPUTED MODIFICATION AFFECTS ANY REJECTION
HEURISTICS IMPLEMENTED IN PROBLEM-SOLVER CODE

THEN CREATE A NEW GENERATION SEARCH CONTROL RULE
WHOSE POSTCONDITION IS THE APPROPRIATELY VARIABLIZED
ALTERNATIVE AND WHOSE PRECONDITIONS IS COMPUTED AS
DESCRIBED IN TABLE 4.16;
WHEN THE COMPUTED MODIFICATION AFFECTS ANY REJECTION
HEURISTICS IMPLEMENTED AS A SEARCH CONTROL RULE
THEN FOR EACH AFFECTED SEARCH CONTROL RULE
MODIFY THE PRECONDITION OF THAT SEARCH CONTROL RULE
AS SHOWN IN TABLE 4.15;
END DO;

Table 5.12: Installing the Modification Set

portions of the search control logic), the procedure starts a search for a solution from the parent of this step. If it finds one then there does exist a solution in the problem-solver's search space and no modifications need be computed. However, if there is no solution found then if the terminal node failure reasons are regressable over the failed search tree and the failure explanations are operationalizable then EBL will be used to compute a failure explanation that generalizes the current situation. This generalization describes when the rejection heuristics that rejected this step cause this step's parent to fail. This is repeated for each step in the meta-level path that is associated with a rejection heuristic. Assuming that all the conditions for coverage preservation hold then as EBL computes further failure explanations, they will not cause any new steps in the meta-level path to now be rejected. Therefore, given these assumptions, for each identified rejection heuristic, *Compute Modifications* further refines the identification to those rejection heuristics (and their meta-level steps) which it is sufficient to modify to extend the search space to include a solution. It then computes conditions for each of these rejection heuristics that identifies when these heuristics should no longer reject an alternative to enable the problem-solver to find a solution to the given problem.

Extension Correctness of Install Modifications

If CANDIDATE(?OP)
& CURRENT-BOARD(?B)
& ?C = TILE-POSITION(?B)
& ?F = TILE-POSITION(ADD-OP(?OP,?B))
& GOAL(TILE-AT(?X, ?Y))
& DISTANCE((?X, ?Y), ?C) ≤ DISTANCE((?X, ?Y), ?F)
THEN REJECT MOVE ?OP

Figure 5.1: General Difference Reduction Heuristic

If CANDIDATE(?OP)
& NOT(*FAILURE-EXPLANATION*)
& **NOT(?OP = UP(?X, ?Y))**
& CURRENT-BOARD(?B)
& ?C = TILE-POSITION(?B)
& ?F = TILE-POSITION(ADD-OP(?OP,?B))
& GOAL(TILE-AT(?X, ?Y))
& DISTANCE((?X, ?Y), ?C) ≤ DISTANCE((?X, ?Y), ?F)
THEN REJECT MOVE ?OP

Figure 5.2: Edge-Level Modification of Heuristic

Table 5.12 shows the *Install Modifications* procedure. I will assume that the problem-solver allows search control rules to be added/modified for its decision procedures and executes these rules after the embedded search control logic has been executed. I now need to show that *Install Modifications* correctly uses the computed modifications to modify the problem-solver (via its search control rules) so that a solution to the given problem is now in the problem-solver's search space. We know that and-ing the negation of the computed modifications to the preconditions of their respective rejection heuristics would extend the problem-solver's search space sufficiently to include a solution. However, the algorithm goes beyond this for two reasons. The first reason is that the algorithm cannot directly modify the problem-solver's embedded search control logic and therefore must do this indirectly. This means that I add generation search control rules to undo the effects of embedded rejection heuristics. The second reason is that SHAPeS is doing edge-level modifications. As I showed in the general difference reduction heuristic in Section 4.2.3, rejection rules need not be search alternative specific. To ensure that the modification is only occurring at the edge and not at the node level, the algorithm adds the negation of a test for the specific edge being unblocked. Figure 5.2 shows the general difference reduction heuristic and its edge-level modification.

It's clear that the installation of the computed modifications into the preconditions of their respective rejection search control rules will not prevent the problem-solver from being able to solve the given problem. Now I need to show that this is true for the installation for the computed modifications associated with embedded rejection heuristics. This type of installation not only puts the computed modification into the new generation search control rule's preconditions, it also puts in the preconditions from the generation code that originally generated the edge and the negation of the disjunction of the preconditions of all the embedded rejection heuristics that could reject this edge but did not do so in the meta-level path.

At an edge in the meta-level path, if that edge was rejected by a embedded rejection heuristic then the installation adds a generation search control rule that will add that edge back into the search space. No embedded rejection heuristic can prune it away because they execute before the search control rules. No rejection search control rule can prune it away because any that could, would have been identified as pruning it and would have a computed modification associated with it to ensure that it doesn't prune it. Since originally, this edge was pruned away by a embedded rejection heuristic then it must have been generated by implicit generation code. Therefore adding the preconditions of that generation code to the preconditions of this generation search control rule will evaluate to true for this edge. Also the negation of the disjunction of the preconditions of all the embedded reject heuristics that could reject this edge but that did not do so on this meta-level path, will evaluate to true for this edge. Therefore the computed preconditions for this generation rule will evaluate to true to generate this edge. Therefore this installation will enable the problem-solver to find a solution for the given problem.

5.3 Generality

I claim that the SHAPeS algorithm works for any search-based problem-solver that fits the architecture (described in Section 3.3) and for any rejection heuristic, domain, and task as long as the coverage extension conditions specified in Claim 5.2.2 are satisfied. This claim follows from my argument in Section 5.2 concerning when the algorithm is

guaranteed to be correct. Since none of these arguments depended upon the specific rejection heuristics, domains, or tasks that were involved, the guarantees are independent of them.

5.4 Computational Complexity

Unfortunately, the more general something is, the less that can be usually said about it. The SHAPeS algorithm I have presented is very general and its computational complexity will depend upon the task, domain, and architecture being used. While I cannot say much about upper bounds on SHAPeS's computational complexity, there are some things I can say about its lower bounds. In this section, I discuss the various aspects of a problem-solving situation that affect the computational complexity of the SHAPeS algorithm and relate them to the specific problem-solving example.

5.4.1 Computational Complexity of IRHM

Claim 5.4.1 (Sufficient Conditions for Linear Complexity of IRHM) *Given a problem-solver, a theory of its search control logic, a problem, and a solution to that problem such that:*

1. *The cost of testing a node (in the implicit generation space) for consistency with the supplied solution is linear with respect to the size of the supplied solution.*

2. *There is a maximum number of children that any node (in the implicit generation space) can generate for this problem.*

3. *There is no need to backtrack in the IRHM algorithm to find a meta-path to the supplied solution.*

Then the cost of finding a meta-level path to the supplied solution is linear with respect to the size of the supplied solution.

Identifying the rejection heuristics that are to be modified (to enable the problem-solver to solve the problem) can be quite expensive. The identification is done by

carrying out a search in implicit generation space, i.e., in a space where all the rejection heuristics have been suspended. The search is for a meta-level path to a solution to the problem. Thus the size of the space being searched can be quite large, potentially the size of the implicit generation space. There are two ideas behind requiring a solution to be supplied to the learner. One is to avoid searching this space when there are no solutions. The other is to tightly constrain the search when there is a solution, thus making the identification cheaper. I make two claims in this section. The first claim describes sufficient conditions for when the cost (of identifying the rejection heuristics to be modified) will be linear with respect to the size of the supplied solution. The second concerns sufficient conditions for when IRHM (Identify Rejection Heuristics to be Modified) will not need to backtrack. First, the notion of the "size" of a solution needs to be made more precise. The system is searching for a solution within a particular space. I am defining the *size* of the solution to be the number of edges traversed from the root to the node containing the solution. This is analogous to the size of a plan being the number of steps within the plan. This definition assumes that a solution can be broken down into a number of components each one of which was selected from a set of alternatives and that each edge along the path corresponds to selecting a component. Note that this definition does not work when an edge simply adds a constraint to the partial solution. However, if I assume that each constraint actually removes at least one candidate from the candidate set then since I have assumed that there is a maximum number of candidates (that any choice point can generate for this problem) then that number is also the maximum number of constraint adding edges that there can be for each component in the given solution. Since I am defining size to show when a process will be linear, if my assumption about a maximum number of constraint adding edges per component holds then the definition of size will still be appropriate. Also note that any meta-level path to this solution will have the same number of components to select, the only difference might be the number of constraint adding edges. Thus, as far as the computational complexity argument is concerned all the meta-level paths to this

```
CREATE THE ROOT NODE OF THE SEARCH TREE FROM THE PROBLEM DESCRIPTION;
UNTIL ONE OF THE UNEXPANDED NODES MATCHES THE SOLUTION
 LOOP
     PICK ONE OF THE UNEXPANDED NODES AND MARK IT EXPANDED;
     USE THE IMPLICIT GENERATION RULES TO CREATE THE CANDIDATES;
     ELIMINATE ANY CANDIDATE THAT IS INCONSISTENT WITH THE
         SUPPLIED SOLUTION;
     MAKE ALL SURVIVING CANDIDATES CHILDREN OF THE PICKED NODE;
 END LOOP;
```

Figure 5.3: Description of IRHM Algorithm

solution will be the same size[3].

Figure 5.3 shows the IRHM (*Identify the Rejection Heuristics to be Modified*) algorithm. If one doesn't have to backtrack in the IRHM algorithm to find the meta-level path to the supplied solution then the supplied solution must be in the implicit generation space described by the theory of the problem-solver's search control logic. Assuming that one can derive sufficient constraints from the supplied solution so that there is no need to backtrack then one simply does a depth-first traversal (without backtracking) of the implicit generation space to find a path to the supplied solution. As I discussed above, any meta-level path this is found will have the "same number" of edges which will correspond to the "size" of the supplied solution. Thus, if the assumptions of Claim 5.4.1 hold then the cost of finding this meta-level path will be linear with respect to the size of the supplied solution.

Claim 5.4.2 (Sufficient Conditions for IRHM not Needing to Backtrack) *Given a problem-solver, a theory of its search control logic, a problem, and a solution to that problem such that:*

1. *The supplied solution exists in the problem-solver's implicit generation space as described by the theory of its search control logic.*

2. *The implicit generation logic generates nodes' children which lead to mutually exclusive sets of partial/total solutions.*

[3]This size will be the product of the maximum number of candidates per choice point and the number of components in the solution.

3. *It is possible to determine which of the children of a node are consistent with the supplied solution.*

Then the IRHM algorithm will not need to backtrack to find a meta-level path to the supplied solution.

Claim 5.4.2 follows directly from its assumptions. Given a solution, one starts with the root of the implicit generation space and generates the alternatives for that node. By these assumptions at most one of those alternatives can be consistent with the supplied solution and since the solution is in the implicit generation space, at least one of the alternatives must be consistent with the solution. Also by these assumptions, one can determine which alternative it is that is consistent with the solution and just explore that alternative. One continues doing this until one finally reaches a node that contains the solution. One was always able to select the "right" alternative, therefore backtracking is unnecessary.

The second assumption not only helps guarantee that one can find a meta-level path to the solution without backtracking, it also guarantees that the search is systematic[16]. Thus, if the other two assumptions hold then for any problem-solver (e.g., SNLP[1]) which is systematic, one can identify which of its search heuristics to modify in time linear with respect to the size of the supplied solution.

5.4.2 Computational Complexity of Compute Modifications

There are two computationally expensive activities in the *Compute Modifications* component. One is the growing of the search tree from the parent of the blocked node. The other is the backpropagation of the failure nodes' reasons back to that parent. I will look at these separately.

I will first look at the cost of building the search tree for the case where there is only one blocked step in the meta-plan and then I will look at the case where there is more than one blocked step. Given one blocked step on the meta-path then to compute the modification one needs to only grow the search tree from the parent of the blocked step. This will be a subtree of the original search tree grown when the problem-solver

initially failed to solve the problem. The main difference between the cost of growing the original tree and growing this subtree is the difference in costs of creating a search node in the original tree and in this subtree. The differences in the costs of creating the search nodes arises from the fact that the problem-solver created the original search tree and its search theory may be embedded while the learner is creating this subtree by interpreting the generation and rejection heuristic search control rules. If one assumes that:

- The difference between the cost of executing code versus interpreting the corresponding rules is a constant factor.
- There is a maximum number of alternatives that any rejection heuristic will prune at a choice point.
- There is a maximum number of rejection heuristics that can be incorporated into any generator.

then the difference in cost between creating the node's children with the problem-solver versus with the learner will be a constant factor. Thus the difference in cost between constructing the original tree with the problem-solver and constructing the subtree with the learner will be a constant factor.

If there is more than one blocked step then how does the computational complexity of learning their modifications during one training example compare to that of learning them separately? The cost of learning them separately depends upon the order in which they are learned. The major factor for each of the blocked steps will be how the search spaces are affected by the earlier learned modifications. Therefore if one learns them in the same order (and for equivalent subproblems) then the costs of deriving the failure explanations for the parents of the blocked step sets will be roughly the same regardless of whether one learned them in the same training instance or in different training instances. If one learns them in a different order then the difference in costs can go either way.

The cost of backpropagating the failure conditions depends upon the nature of the expressions being backpropagated and of the actions through which they're being

backpropagated and can be arbitrarily expensive. If no simplification is being done then the size of the expression being propagated tends to grow exponentially with the depth of the search tree. Thus the cost of backpropagating the failure conditions will tend to be at least exponential with respect to the depth of the search tree. The rejection heuristics affect the number of nodes through which the failure explanations need be backpropagated, i.e., the more nodes rejected by the heuristics, the fewer to be backpropagated through. However, the larger the number of rejection heuristics, the more potential for overlap[4] between them. Since all the failure reasons at the nodes are being backpropagated then the larger the overlap[5], the larger the terminal node failure expressions to be backpropagated.

One could simplify the expressions being backpropagated and this would reduce the cost of backpropagation. However, simplification can be arbitrarily expensive, and the cost of the simplifications might overshadow any savings gained by backpropagating the simpler expressions. This might be avoided by not attempting to "totally" simplify the expressions but to "cheaply" simplify them.

While the cost of computing the failure explanation may be expensive, the failure explanations computed can be used in another way to modify the problem-solver's search control theory. The computed failure explanations can be used in augmenting the system's collection of rejection heuristics. In order to compute the failure explanation of the parent of the blocked node, the learner must first compute the failure explanations of all the descendants of that parent. What one is proving is that all these descendants are doomed to fail. Therefore one could make use of these additional failure explanations to create new rejection heuristics. These can be used[6] to prune away those "doomed" portions of the search space in the future and will have cached the explanations of why they were doomed. Thus the cost of computing the failure explanation can be

---

[4]Two rejection heuristics are said to overlap when they both reject the same alternative.

[5]The overlap is larger at a node when there are more rejection heuristics involved in rejecting all the edges from that node.

[6]Assuming that they pass some utility test that assures the system that it is more useful to have that particular rejection heuristic in its rule set (with the consequent costs of having to test for it) than to do the search to discover that it fails.

amortized over all the rejection heuristics that are also learned.

5.4.3 Computational Complexity of Install Modifications

The cost of modifying the search control theory is proportional to the number of rejection heuristics needing to be modified. In most circumstances this will be dominated by the cost of computing the modifications.

5.5 Summary

In this chapter I have identified three types of search control rule interactions and ways to eliminate them. The main effect of eliminating them is that it becomes easier to reason about the effect of modifying search control rules upon the topology of the search space. For example, without these types of interactions specializing the preconditions of a rejection search control rule cannot cause edges to be removed from the resulting search space.

Assuming these interactions have been eliminated, I then analyzed the correctness of the SHAPeS algorithm with respect to the problem-solver's coverage. I showed that my method of modifying the rejection heuristics could not cause the problem-solver to become unable to solve any problem that it was previously able to solve and that a solution would be guaranteed to to lie in the problem-solver's explicit search space. These arguments assumed that the problem-solver's architecture followed that described in Section 3.3, otherwise no assumptions were made about the rejection heuristics, domains, or problem-solving task. The absence of these types of assumptions argues for the generality of the SHAPeS algorithm.

In the last section, I discussed the computational complexity of the various procedures in the SHAPeS algorithm. I discussed sufficient conditions for when the cost of identifying the rejection heuristics to be modified would be linear with respect to the size of the supplied solution. I then discussed when the cost of computing the modifications would be no more than a constant factor of attempting to solve the relevant subproblems sequentially. I concluded with an observation that the cost of installing

the modifications will be dominated by the cost of computing the modifications.

Chapter 6

Utility Analysis

6.1 Introduction

In the last two chapters I described and analyzed SHAPeS, the Edge-Level Failure-Based (ELF) search-heuristic modification algorithm. In this chapter, I will discuss the usefulness of this algorithm. In particular, I will explore whether the modifications it produces are useful. Specifically, I want to explore two issues: (1) can these modifications enable a problem-solver to perform better; and (2) can these modifications improve performance more than current alternatives. The first issue, I call "intrinsic usefulness" and the second, "comparative usefulness".

A problem-solver's behavior can be measured along the following dimensions: (1) domain coverage; (2) problem-solving speed; and (3) solution quality. *Domain coverage* refers to how many of the domain's solvable problems the problem-solver is able to solve. *Problem-solving speed* refers to how long it takes to either solve or fail to solve the problem. Since I will only be looking at plans, I will consider *solution quality* to refer simply to the length of the plan, i.e., the longer the plan, the lower its quality. In all the experiments, the solution quality (i.e., length) remained the same or only changed slightly (i.e., was either a step longer or shorter). Therefore I will ignore solution quality for the rest of this chapter.

I will distinguish between two types of behavior changes for problem-solvers: tradeoffs and monotonic changes. *Tradeoffs* occur where there are both improvements and degradation in performance along some of these dimensions. One example of a tradeoff is where the problem-solver is now able to solve some previously unsolvable problems but is no longer able to solve others. Another example is where the problem-solver's domain coverage increases but its problem-solving speed decreases. A performance change

is *monotonic* when either there is improvement along one dimension and no degradation along any dimension or there is degradation along one dimension but no improvement along any dimension. An example would be where the problem-solver's coverage and problem-solving speed both improve while the solution quality remains unchanged.

A problem-solver's overall behavior is considered improved if either (1) it improves monotonically; or (2) a tradeoff occurs that is acceptable to the user. While the first type of improvement is independent of the user, the second type is obviously not.

6.2 Experimental Set-up

In this section, I will describe the experimental set-up. In particular, I will describe the implementation of the ELF algorithm, the domains, the training and testing problem sets, and the ELF generated modification sets.

6.2.1 Implementation: Bacall

Unfortunately, the implementation, called **Bacall**, preceded the theory and is restricted to refining one specific rejection heuristic, Linearity[19]. The *Linearity* heuristic rejects working on goals in a non-depth-first fashion, i.e., it picks one of the problem's goal conditions to achieve and then an operator to achieve it and if any of its preconditions need to be achieved it picks one of those preconditions and continue to subgoal until it picks an operator all of whose preconditions are already achieved, then it pops back up a level, etc. I implemented **Bacall** as an extension to the PRODIGY planner[13]. PRODIGY extends traditional totally-ordered-plan means-ends planning by possessing a facility for allowing the user to easily encode search control rules for a particular domain. Like many other such planners, PRODIGY also uses the Strong Linearity[6] rejection heuristic to reduce the number of totally-ordered plans it explores. Informally, *Strong Linearity* forbids subplans for different goals to be interleaved. However, Strong Linearity also causes PRODIGY to fail to solve some solvable problems. PRODIGY (like most totally-ordered means-ends planners) fuses together the Linearity and Strong Linearity rejection heuristics, causing the modification of one to also modify the other.

Bacall learns search control modifications that directly modify PRODIGY's Linearity rejection heuristic and consequently indirectly modify Strong Linearity as well. These modifications enable PRODIGY to solve problems that these heuristics had caused to be unsolvable. The basic difference between the **Bacall** implementation and the SHAPeS algorithm (as discussed in Chapter 4) is that **Bacall** only addresses these rejection heuristics while SHAPeS is much more general.

6.2.2 Domains

These explorations take place in two domains: BlocksWorld and a modified version of the STRIPS robot world, which I call One-Way STRIPS[1]. One-Way STRIPS is different from the standard STRIPS domain in the following ways: (1) the doors are one-way; (2) the robot can carry an indefinite number of blocks; (3) the rooms are arranged in a line ; (4) the rooms at the two ends are special, one room (call it FIRST) has only a single door which leads out of it, the other (call it LAST) has only a single door which leads into it; and (5) all the rooms in between have two doors, a door on FIRST's side of the room which leads into the room, and a door on LAST's side of the room which leads out of the room. Thus the robot can start in any room and only go towards any room between its current room and LAST. These domains will be described in more detail in Appendix B.

6.2.3 Problem Sets

For each domain, there were two types of problem sets generated. The first type, called the *in situ* problem sets, explores how PRODIGY's performance changes on the problem set where the modifications were learned. The second type, called the *scalability* problem sets, explore how PRODIGY's performance changes on a set of problems that are larger than the problems where **Bacall** learned the modifications. Additionally the *in situ* problem set was designed so that it contained all problems of a certain "size" and reasonable complexity such that they could be attempted by

[1]The reason for modifying STRIPS is to enable PRODIGY's Strong Linearity heuristic to prune away all solutions for some problems.

PRODIGY (with a reasonable search control theory) without running out of resources in a reasonable amount of time (e.g., 3 cpu hours). For BlocksWorld this was the set of all non-trivial[2] four-block BlocksWorld problems, where all isomorphic[3] problems are eliminated. The *scalability* problem set was designed so that about 500 problems could be attempted without encountering too many resource boundaries within a tolerable period of time (e.g., 30 cpu hours). For BlocksWorld, this was a set of 500 non-isomorphic, non-trivial eight-block problems. For One-Way STRIPS, the first type of problem set turned out to be the set of non-isomorphic three-block problems (these had anywhere from 2 to 7 rooms), and the second type was the set of non-isomorphic four-block problems (these had anywhere from 2 to 9 rooms).

The experiments were designed to minimize the variability of cpu time reported for a problem set. The experiments were carried out on a Sun SPARC workstation. Each problem set was run in a new core image and only run when no other users were present on the system. I re-ran the same problem set (which took over an hour of cpu time) thirty times under my normal experiment operating conditions. The average cpu time per run was 761.1 cpu seconds. The standard deviation was 1.2 cpu seconds, i.e., .16% of the average run cpu time. Therefore, for these runs, a deviation of more than .5% will be statistically significant. The *in situ* problems always ran to completion (regardless of whether a solution was found or not) and thus were not affected by resource horizon[18] effects. However, this was not true for the scalability problems, and consequently those results could be so affected.

6.2.4 Modification Sets

From both *in situ* problem sets, **Bacall** learned a set of **Bacall** refinements which would enable PRODIGY to solve more problems in that problem set[4]. A refinement is

[2]A trivial BlocksWorld problem is one where either all the blocks in the initial state or goal state are on the table.

[3]Problems that only differ in the labeling of the blocks.

[4]Though the PRODIGY system is fairly robust it is still experimental software and has occasional lapses. There were a few cases where PRODIGY/EBL was unable to compute an explanation for why PRODIGY was unable to solve a particular problem. When this happens, **Bacall** is unable to create refinements.

expressed as a standard PRODIGY search control select rule. Although PRODIGY's select rules were originally used to select one of the alternatives as being better than all the non-selected alternatives, they could also "*select*" an alternative that hadn't been computed as an alternative. It is in this latter capacity that **Bacall** uses the select rules[5]. I call a set of **Bacall** refinements a *modification set*. For these explorations of how **Bacall**'s refinements affected coverage, problem-solving speed, and solution quality, I created all non-redundant[6] modification sets from the set of all the refinements that **Bacall** learned from the training set. These modification sets provide a range of tradeoffs and monotonic improvements.

6.3 Exploring ELF's Intrinsic Utility

In this section I explore the type of improvements that ELF's modifications (as implemented in **Bacall**) can make to PRODIGY's performance. ELF modifications cannot cause performance to monotonically degrade since, at least one problem that was previously unsolvable (for the problem-solver) is now solvable. Therefore ELF modifications can only cause either tradeoffs or monotonic improvements. I discuss these two types of improvement separately. In Section 6.3.1 I discuss a case where ELF modifications make monotonic improvements to the problem-solver's performance. In Section 6.3.2 I discuss a case where the ELF modifications cause a tradeoff to occur in the problem-solver's performance.

6.3.1 Monotonic Improvement

The purpose of ELF modifications is to extend the problem-solver's domain coverage. For monotonic improvement to occur, all previously solvable[7] problems must remain

[5]The former use of select rules was canceled for the **Bacall** learned modifications by adding a rule that "selects" all available alternatives.

[6]A modification set is *redundant* if a modification can be removed from it without affecting which problems it will solve in its *in situ* problem set.

[7]I mean solvable by the problem-solver with its current search control theory, not whether it is theoretically solvable.

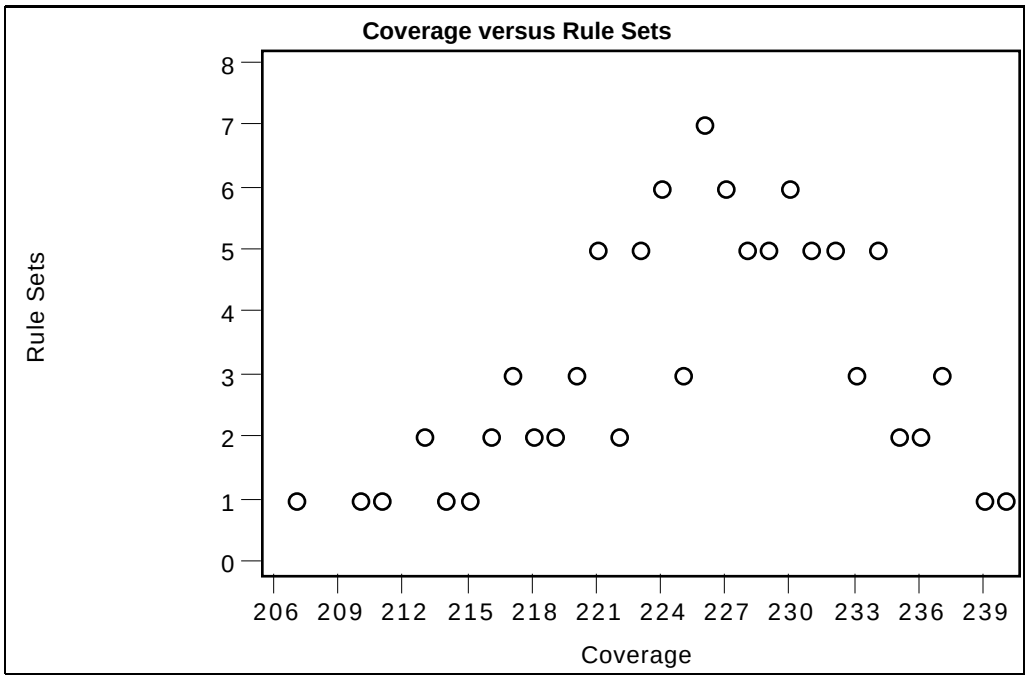

Figure 6.1: Coverage vs Number of Modification Sets

solvable, and neither the problem-solver's speed nor the solution quality may degrade[8]. The ELF algorithm does not directly address the issues of problem-solving speed nor of solution quality. How these are affected by ELF modifications depends upon the quality of the search control theory used by the problem-solver.

I will now look at an instance where ELF modifications made monotonic improvements to a problem-solver behavior over the problems where the modifications were learned (i.e., *in situ*). Afterwards I will show that these monotonic improvements remain when the problems are scaled up.

In Situ Improvement

The purpose of the *in situ* experiment is to examine how PRODIGY's performance can change on a set of problems as **Bacall** incrementally learns all the modifications it can from that set. Specifically, **Bacall** learns all the modifications that it can for this set of problems and then I run PRODIGY with all meaningful combinations of **Bacall** modifications. To do this exhaustively, I chose 4-block problems. As mentioned in

[8]I am referring to the average speed and average solution quality not to the individual ones.

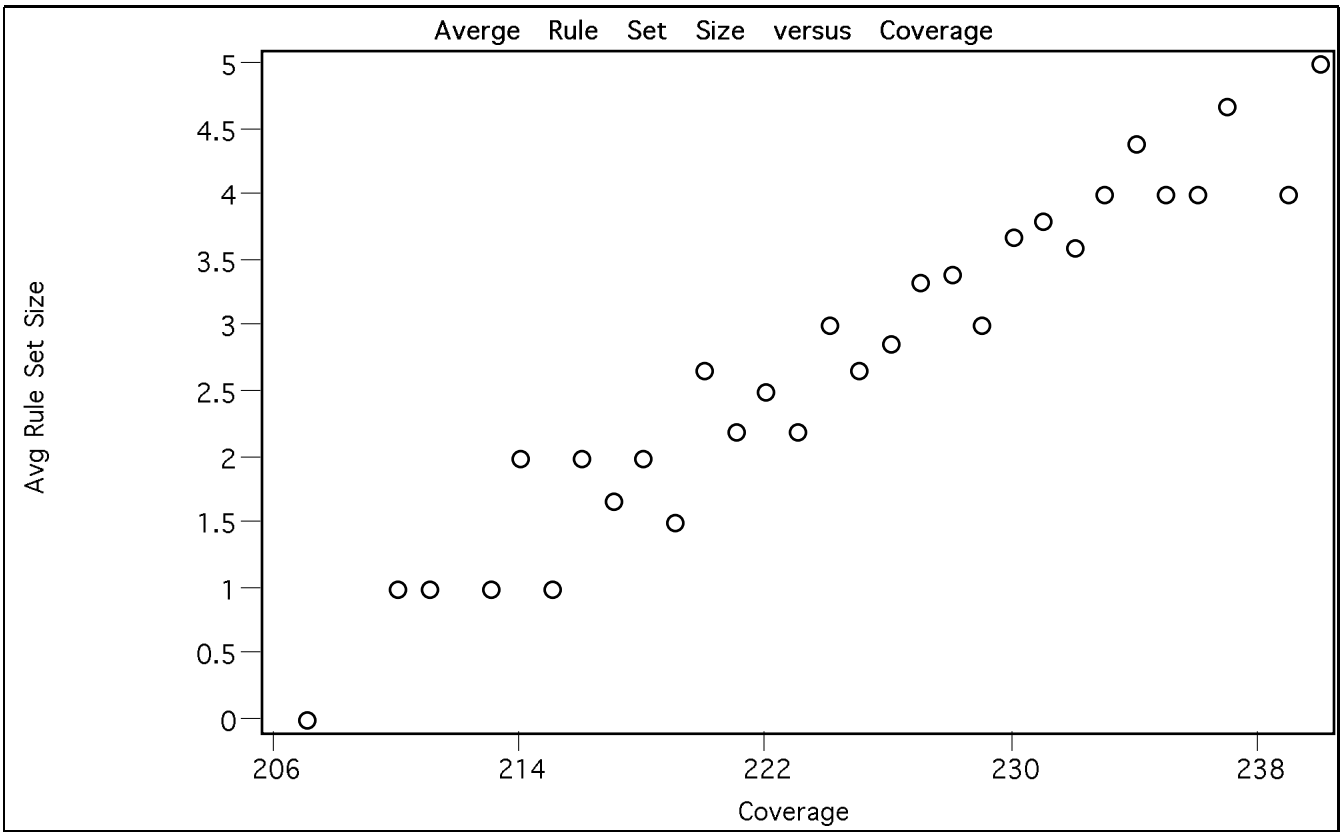

Figure 6.2: Coverage vs Size of Modification Set

Section 6.2.3, the *in situ* problem set for BlocksWorld contained all the non-trivial, non-isomorphic four-block problems. This meant a total of 269 problems in the problem set. From this problem set, **Bacall** was able to learn eight modifications, which generated 96 non-redundant modification sets (this includes the empty modification set). The empty modification set (which I will refer to as the *base modification set*) solved 206 problems and the *best* (i.e., the one with the widest coverage) of the modification sets solved 240 of the 269 problems. The coverage of the other modification sets ranged in between. Figure 6.1 shows the number of modification sets per coverage. For example, one sees that there was only one modification set that solved exactly 215 problems, while there were seven modification sets that solved exactly 226 problems.

Figure 6.2 shows the relationship between the size (i.e., the number of rules) of the modification set and the coverage achieved using that modification set. One sees that, on average, as coverage increases so does the size of the modification set and that its growth seems "reasonable", i.e., not exponential.

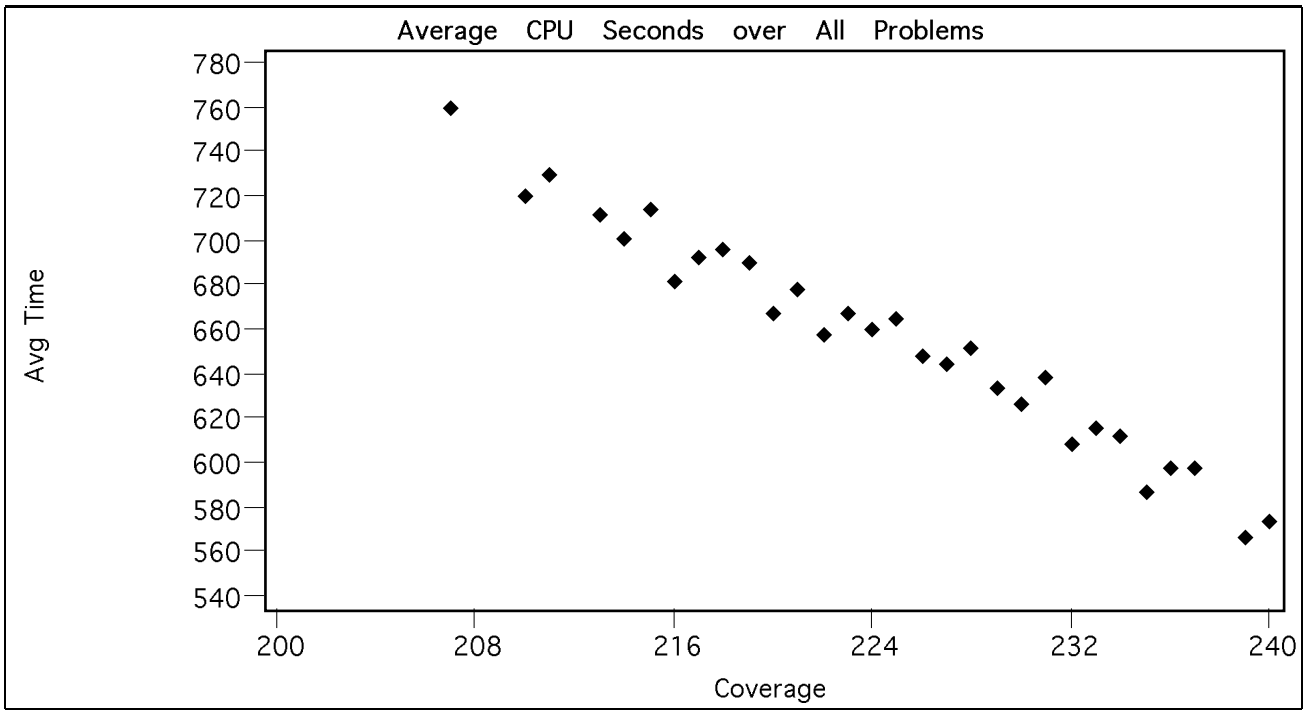

Figure 6.3: Average CPU Seconds over All Problems

One question is how does PRODIGY's problem-solving speed change as its coverage increases. Figure 6.3 shows how, on average, the number of CPU seconds (taken to try all 269 problems) changes as the **Bacall** modification sets increase PRODIGY's coverage. It shows that PRODIGY (without any **Bacall** modifications) takes 760 CPU seconds to attempt all 269 problems, but, with the best **Bacall** modification set it only takes 575 CPU seconds, a 30% increase in speed. Figure 6.3 shows that, on average, as coverage increases so does PRODIGY's problem-solving speed. It also shows that the best (coverage) modification set increased the coverage by 15%.

One would like to understand what is causing this speedup. To this end, the problem set has been split into three subsets. One subset, the *base problem subset*, contains only problems that the base (i.e., empty) modification set could solve. Another subset, the *unsolvable problem subset*, contains those problems that no modification set could solve. The final subset, the *enabled problem subset*, contains the remaining problems, i.e., those problems not solvable by the base modification set but which were solvable using one of the other **Bacall** modification sets. The best modification set (i.e., the one with the widest coverage) solved all problems in this set. There were 206 problems in the base

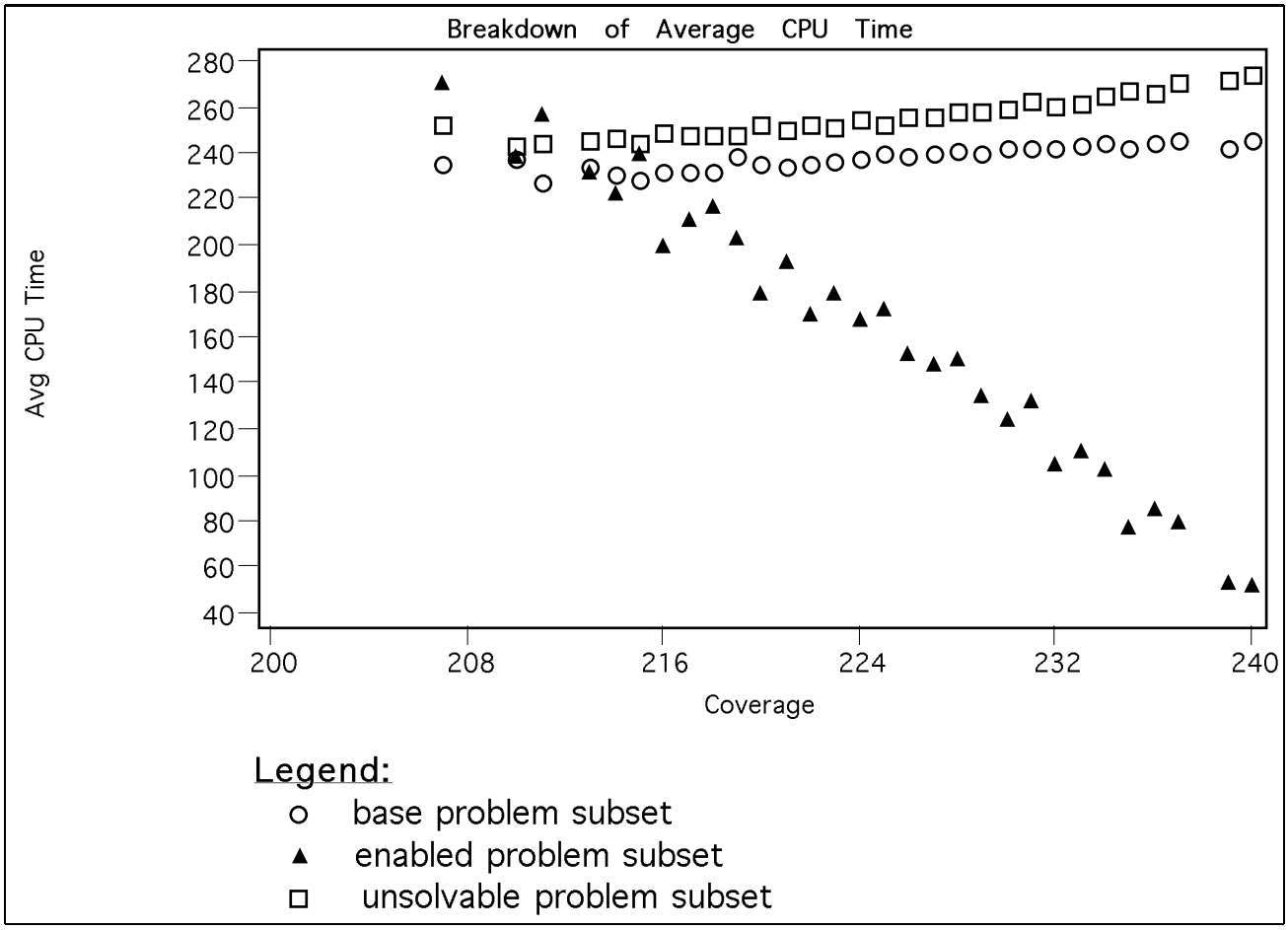

Figure 6.4: Breakdown of Average CPU Time by Problem Subset

problem subset, 29 problems in the unsolvable problem subset, and 34 problems in the enabled problem subset. All the problems in the base problem subset were solved by every modification subset. It is only the problems in the enabled problem subset that shifted from being "unsolvable" to being "solvable" and no problem shifted from being "solvable" to being "unsolvable" (this need not be true).

Figure 6.4 shows how average CPU time of the **Bacall** modifications sets breaks down by problem subset. Even though the base problem subset had many more problems than either of the other two subsets (i.e., 206 versus 29 and 34) the base modification set took about the same CPU time to attempt all the problems in each subset. The figure shows that as the coverage increases the average CPU time for the base problem subset remains about the same or rises slightly (from 236 CPU seconds for the base modification set to 246 CPU seconds for the best modification set, a 4% increase) and that for the unsolvable problem subset the time increases slightly more steeply (from 252 CPU seconds for the base modification set to 274 CPU seconds for the best modification set, a 9% increase). However, the average CPU time taken to attempt the enabled problem subset decreased dramatically (from 271 CPU seconds for the base modification set to 53 CPU seconds for the best modification set, a 80% decrease) as problems shifted from being "unsolvable" to "solvable". As described in Section 6.2.3, deviations of more than .5% in the run's cpu time are statistically significant.

Why is the average CPU time decreasing as coverage increases for the enabled problem subset? It could be that as the coverage is increasing, the average amount of CPU time spent per node is decreasing or that the number of nodes being created is decreasing (or for both reasons). Figure 6.5 shows how the average CPU time per enabled problem node changes as the coverage increases. It shows that at first the average CPU time per node increases as the coverage increases but that towards the end it starts decreasing. However, this behavior is very different from the behavior one sees for the average CPU time for attempting enabled problems in Figure 6.4 and consequently is not the major cause for the increase in problem-solving speed that one sees there. Therefore one needs to see what happens (as coverage increases) to the number of nodes created while attempting to solve the problems in the enabled problem

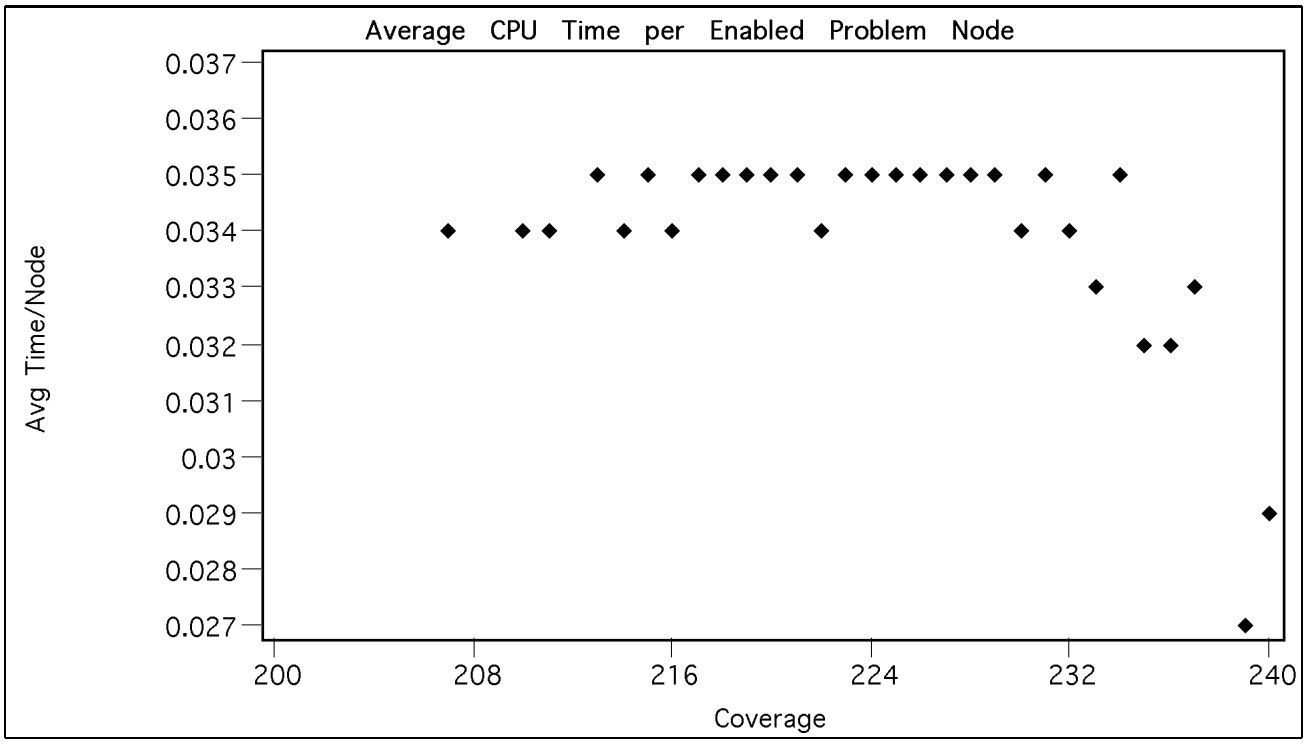

Figure 6.5: Average CPU Time per Enabled Problem Node

subset.

Figure 6.6 shows the average number of nodes created (while attempting to solve the problems in the enabled problem subset) changes as more of them change from being "unsolvable" to being "solvable". The number of nodes created decreases substantially (from 7913 nodes for the base modification set to 1826 nodes for the best modification set, a 75% reduction). Since, as I showed in Figure 6.4, the enabled problem subset took about a third of the CPU time for all the problems, then reducing that by 75% accounts for most of the 30% overall speedup in the problem-solver that one saw in Figure 6.3.

Scalability of Improvement

In the *in situ* experiment, I examined how the **Bacall** modifications could affect PRODIGY's performance on the set of problems where the modifications were learned. The scalability experiment examines how **Bacall** modifications can affect PRODIGY's performance on problems that are much larger than the ones where the modifications

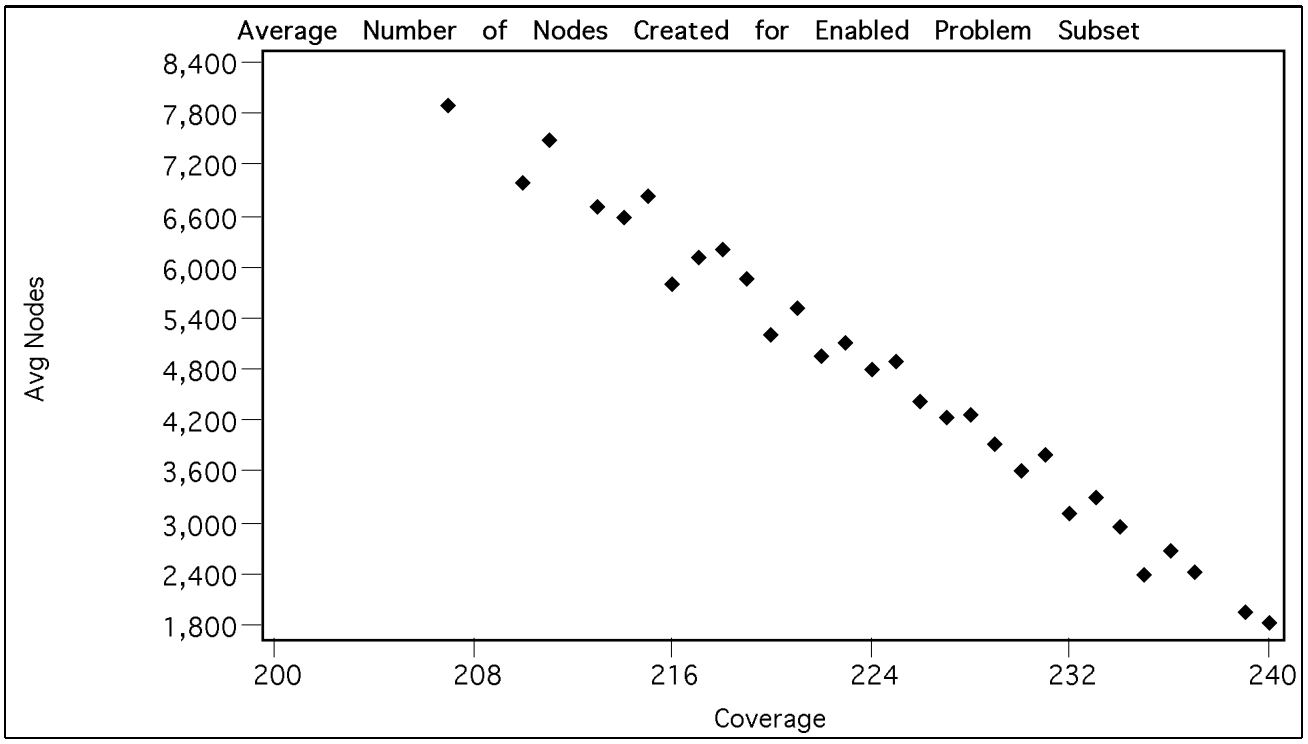

Figure 6.6: Average Number of Nodes Created for Enabled Problem Subset

were learned. The modifications are the ones learned from the 4-block problem set. I will examine how they affect PRODIGY's performance on 500 randomly generated non-trivial non-isomorphic 8-block BlocksWorld problems.

Unfortunately, using the original search control theory caused PRODIGY to frequently exceed the defined resource limits [9]. To get around this, I added a number of search control rules to the base search control theory. These rules enabled the base modification set to create only 6,308 nodes instead of its original 23,167 and enabled the best modification set to create only 5,914 nodes instead of its original 17,407. Thus for the *in situ* problems, the base modification set created, on average, about 24 nodes per problem to solve (or fail to solve) a problem. These new search control rules rejected working on goals that would need to be undone to achieve remaining goals and rejected operators and bindings that would not increase PRODIGY'S chances of solving the problem[10]. I call this new search control theory an "expert" search control theory to

[9]The resource limits were 3,000 CPU seconds and 25,000 nodes.

[10]For example, if the goal was to clear box **B**, and currently box **A** was on top of box **B**, then binding **?X** to box **C** for operator **unstack(?X, ?Y)** was unlikely to be useful and was therefore rejected.

differentiate it from the original (and more naive) search control theory. I used the expert search control theory for the scalability experiments.

Even with this expert search control theory, PRODIGY was not able to finish its attempts to solve each problem. In particular, there were 33 problems where either the base or the best modification set hit the specified resource limits. Of those 33 problems: both modification sets hit their resource limits on 17; the base modification set alone hit a resource limit on 15; and the best modification set alone hit a resource limit on one problem. The 17 problems where both modification sets hit their resource limits are called the *doubly censored* problems, the 15 problems where only the base modification set hit its resource limit are called the *base censored* problems, and the problem where only the best modification set hit its resource limit is called the *best censored* problem. The best modification set solved all 15 base censored problems. The base modification set failed to solve the best censored problem and just missed hitting its resource limit.

Until recently, machine learning researchers compared the speed of two problem-solvers by comparing their accumulated cpu times for solving a random set of problems, the one with the smallest accumulated cpu time was the "faster". When the "faster" problem-solver always finished its attempts at solving a problem, this approach was valid. However, as Segre, Elkan, and Russell[18] point out, if the "faster" system has problems censored by resource limits then experiments that compare these two systems' problem-solving speeds can exhibit an horizon effect, i.e., changing the resource bounds can radically change which system appears faster. Etzioni and Etzioni[8] describes two conservative statistical tests for analyzing speedup experiments. Both of these tests ignore the magnitude of the differences between the systems' problem-solving times. Instead they either just look at the sign of the differences or they look at both the sign and the ranking of the differences. This helps them to be less sensitive to the effect of censoring and to the effects of outliers. Unfortunately, in this system it is precisely the outliers which make it possible for the modified system to be faster. In these experiments, the outliers are usually the problems that were unsolvable for the base modification set but which become solvable for the best modification set. These problems are outliers because there is usually a substantial difference between how long

it takes the system to find a solution and how long it takes to fail. With the expert search control theory the system often goes directly to the solution. Because they ignore the actual magnitudes of the differences in problem-solving times, the tests proposed by Etzioni and Etzioni[8] are not appropriate here.

One standard statistical approach to handling censored data (i.e., data where the problem-solver's resource consumption is terminated early because it has exceeded its resource limits) is to throw out the doubly censored data (i.e., data where both problem-solvers fail to finish their attempts to solve a problem because they both exceed their resource limits) and assume that the trends observed in the uncensored and singly censured data extrapolate to the doubly censored data. As Etzioni and Etzioni[8] point out, this assumption may not be true in speedup learning experiments. However, the intent is not to "prove" that using the best modification set causes PRODIGY to be faster when attempting to solve the scalability problem set than when it uses the base modification set. Instead, the intent is simply to show that it is plausible that this can be so. The data in Table 6.1 shows their relative performance over all non-doubly censured data.

Table 6.1 shows that the best modification set extended PRODIGY's coverage by 51 problems (i.e., by more than 10%), decreased the CPU time used by a little less than 75% and the number of nodes created by a little more than 75%, while increasing the average CPU time spent per node by 10%. In the *in situ* experiment, the improved performance came primarily from the decrease in the nodes created for the problems which the base modification set had been unable to solve but which the best modification set could solve. One sees from Table 6.1 that the same is true here. However, one also sees that the best modification set decreased the number of nodes created by 30% for the base-solvable problems over the base modification set. I believe this is due to the amount of overlap between the preconditions of the rejection rules in the search control theory being used, where a node may be pruned by multiple rejection heuristics. Were a less redundant search control theory used, the number of nodes being created by the best modification set could be greater than the number being created by the base

	PROBLEMS	BASE NODES	BEST NODES	BASE CPU Sec	BEST CPU Sec	BASE CPU Sec/Node	BEST CPU Sec/Node
BASE	407	43,035	28,981	12,793	10,799	0.297	0.373
ENABLED	51	232,771	7,348	64,604	2,462	0.278	0.335
UNSOLVABLE	25	35,934	36,385	8,146	8,780	0.227	0.241
ALL	483	311,740	72,714	85,543	22,040	0.274	0.303

Table 6.1: Data on Non-Doubly Censured Problems for Scalability Experiment

modification set. Also, the "expert" nature[11] of the search control theory being used causes the best modification set's increase (of the number of nodes being created for the non-solvable problems) to be quite small (approximately a 1% increase).

Can one say anything about how the use of resource limits might have biased the analysis? Certainly, the 15 problems that PRODIGY was able to solve with the best modification set but exceeded its CPU time limit with the base modification set underestimated the savings achieved by now being able to solve those problems. However, the 17 problems which were doubly censured underestimated the cost of the best modification set's extended search space. It is quite possible that were those 17 problems allowed to run to completion that the relative advantage in performance of the best modification set over the base modification might have been reversed. In which case, instead of being an instance of monotonic improvement this would be one of trading problem-solving speed for extended coverage.

6.3.2 Tradeoffs

In the previous section I showed instances where the ELF modifications made monotonic improvements to the problem-solver's performance. In this section I will look at instances where the ELF modifications do not monotonically improve PRODIGY but rather represent a tradeoff in the problem-solver's performance. Whether this tradeoff

[11] Especially due to the fact that the theory emphasizes rejecting bad paths over preferring good paths.

is an improvement depends upon the user's situation and his alternatives. As in the look at monotonic improvements, I will first look at the tradeoffs that occur in an *in situ* problem set and then look at how they change when one scales up. The domain will be the One-Way STRIPS domain. After I examine the tradeoffs offered by the **Bacall** modifications, I will examine in Sections 6.4.1 and 6.4.2 how they compare with the tradeoffs presented by current modification alternatives.

In Situ Tradeoffs

I only have eight modification sets for this experiment, one being the base modification set. The problem set has 302 3-block problems (the number of rooms varies from two to seven). Table 6.2 shows the data for this experiment. The base modification set can solve only 18 of the 302 problems. All eight modification sets have the same number of solution steps for those 18 problems, so solution quality remains static. However, as the coverage increases, the cpu time also increases. Coverage increases much more dramatically in this case than in the BlocksWorld case. One of the reasons for the large increase in problem-solving time is that for many problems in the problem set, more than one modification is needed for the problem to become solvable. This was not the case for BlocksWorld, where usually an unsolvable problem only needed one new **Bacall** modification to make it solvable. This need for multiple modifications means that the **Bacall** modifications often expand the search space without bringing in solutions. Table 6.2 shows the percentage that modification sets increase coverage and cpu time from the base modification set. It also shows the ratio of increase in coverage over the increase in cpu time. While the actual numbers are not significant, the figures suggest that, in general, the coverage is increasing faster than the cpu time.

Scalability of Tradeoffs

The scalability problem set contained 664 non-trivial non-isomorphic 4-block One-Way STRIPS problems (with the number of rooms ranging from two to nine). I only ran the base modification set and the best **Bacall** modification set (the modification set that solved 253 of the 302 *in situ* problems). The base modification set was only able

DOMAIN COVERAGE	RULESET SIZE	AVERAGE NODES	AVERAGE CPU SECS	% INCREASE COVERAGE	% INCREASE CPU TIME	RATIO OF INCREASES
18	1	31,820.00	735.52			
51	2	77,483.00	2,341.14	183%	218%	.84
67	1	116,605.00	3,686.82	272%	401%	.68
99	1	131,842.00	4,430.53	450%	502%	.90
181	2	165,853.00	6,668.72	906%	807%	1.12
253	1	180,341.00	7,441.60	1306%	912%	1.43

Table 6.2: *in situ* Coverage-Speed Tradeoffs

DOMAIN COVERAGE	RULESET SIZE	AVERAGE NODES	AVERAGE CPU SECS	% INCREASE COVERAGE	% INCREASE CPU TIME	RATIO OF INCREASES
2	1	110,159	2,481.2			
460	1	2,658,410	137,737.45	22,900%	5,451%	4.20

Table 6.3: Scalability Coverage-Speed Tradeoffs

to solve 2 of the 664 problems. The best *in situ* modification set was able to solve 460 of the problems, however none of these 460 problems were the 2 problems solved by the base modification set. Thus, the **Bacall** modifications traded coverage as well as speed. Since the two coverages do not intersect one cannot talk about the change in solution quality. In Table 6.3 I show that the number of solvable problems increased 230-fold while the problem-solving time increased only 56-fold. If one looks at the ratio between how much the domain coverage expanded and how much the problem-solving time increased, one sees that for the scalability problem set this ratio is much higher than for the *in situ* problem set. This suggests that as the problems get bigger, the cost (in problem-solving speed) of increasing the domain coverage will decrease even further.

6.4 Exploring ELF's Comparative Utility

In the last section I showed that ELF modifications could change a problem-solver's performance either through monotonic improvements or through tradeoffs. This is one perquisite for the algorithm to be useful. However, since there already exist other algorithms that modify rejection heuristics, another perquisite is that for each existing alternative there must be situations where the ELF algorithm is more effective than that alternative algorithm. I say one modification is more *effective* than another when it produces more performance improvement.

There are two existing algorithms for modifying rejection heuristics: SOAR [11] has a problem-level failure-based (*PLF*) algorithm and FS2 [3] has an edge-level success-based (*ELS*) algorithm. The SOAR PLF algorithm learns modifications that check whether to suspend a rejection heuristic for all nodes within the problem's search space based on a characterization of when SOAR could not solve the training problem using that rejection heuristic. The FS2 ELS algorithm learns modifications that check whether to suspend a rejection heuristic for a specific edge, based on an explanation of why that edge's specific alternative led to achieving a top-level goal of the training problem. In Section 6.4.1 I show a situation where ELF-produced modifications are more effective than PLF-produced modifications. In Section 6.4.2 I show a situation where ELF-produced modifications are more effective than ELS-produced modifications.

However, the ELF algorithm will not always produce more effective modifications than the other algorithms. For example, the PLF algorithm might produce more effective modifications in situations where the rejection heuristic (to be modified) only prunes solutions (and prunes away all solutions to many problems).

While I cite the SOAR PLF and the FS2 ELS approaches as existing alternatives to the ELF algorithm, these implementations of the PLF and the ELS algorithms diverge in places from their original implementations. I will discuss these divergences in the Sections 6.4.1 and 6.4.2.

The modifications a learner can make to a system's rejection heuristics can differ in three ways: generality, effectiveness, and overhead. *Generality* refers to the range of

circumstances where the modification will be applied. *Effectiveness* refers to whether applying the modification will lead to the problem being solved. *Overhead* refers to how much it costs to test the modification's preconditions and/or how much additional space much be searched. In these discussions, I will use these three ways to compare the ELF approach to the others.

6.4.1 Problem-Level vs Edge-Level Modifications

The motivation behind problem-level modifications is to avoid having the modification test at each edge whether the heuristic should be suspended. The problem-level approach does it once per problem instead of once per edge. There are two problems with this general approach. The first is that while placing the modification tests at the problem-level decreases the average cost per node, it will normally significantly increase the branching factor when the modification tests are satisfied[12] because the rejection heuristic will be suspended for the entire search tree rather than just at a few nodes. If placing the tests at the edge-level results in a constant increase in the per node costs, then for sufficiently large problems (i.e., problems that with long solution paths in the search space) the cost of extra tests at the nodes will be less than the cost of the extra nodes (which should increase exponentially with respect to the depth of the search)[9]. The second problem with this approach is that if the search control theory has meta-level interference then the more globally the modification applies, the more likely that some problems that were solvable will become unsolvable. In general, one would expect edge-level modifications to be a more effective way of modifying a problem-solver's behavior.

[12]For problems where the problem-level modification tests are not satisfied, the edge-level modification may still apply. For these problems, the problem-level approach will not affect the search space size while the edge-level approach may cause the search space size to increase. However, assuming that edge-level modifications usually only apply at a small percentage of nodes in the search space then the increases in search space size that occur when problem-level modifications apply usually dominate any increases in search space size that occur when only edge-level modifications apply.

PLF Implementation

The SOAR PLF's implementation creates a modification whose precondition is a characterization of when that rejection heuristic caused the system to fail to solve the training problem. This characterization was learned inductively.

This characterization was overspecialized because it is operationalized in terms of the description of the training problem. Therefore if the problem description is modified so that the training problem conditions are changed but the new problem is a subproblem along the path to the solution of the training problem, then presumably one would still like the heuristic to be suspended. However, if one is limited to problem-level decisions it is hard to see how one could detect this situation. Since the easiest way to test two rejection-heuristic modification algorithms is when they produce the same change in coverage, I suspend the rejection heuristic at the problem-level whenever the ELF modifications caused a problem to become solvable.

It should be noted that **Bacall** only learns to suspend the Linearity rejection heuristic[13] and therefore for problems that the ELF modifications caused to become solvable, the PLF "modification" suspends the Linearity heuristic[14] for the entire problem.

The implementation of the PLF modifications involves modifications to PRODIGY's search control theory and to the domain definition (which are described more fully in Appendix C). Suspending a heuristic for an entire problem should decrease the per node costs. However, since the suspension of Linearity at the problem level is achieved by using edge-level search control rules to force PRODIGY to do strict backward-chaining, then the per node costs actually increased substantially in this implementation. Consequently, I will only report on the number of nodes created and not on the cpu time.

---

[13]PRODIGY also normally uses another goal rejection heuristic (select-first-goals), which is a strengthening of Linearity that restricts the order the subgoals can selected. Since select-first-goals is a strengthening of Linearity, when Linearity is suspended, select-first-goals must also be suspended.

[14]Select-first-goals is also suspended for the entire problem. This further increases the branching factor.

MODIFICATION SET	PROBLEMS SOLVED	PROBLEMS ATTEMPTED	NODES CREATED
ELF	253	302	180,341
PLF	0	130	325,543

Table 6.4: One-Way STRIPS: ELF vs PLF

One-Way STRIPS Experiments

Because PLF modification preconditions will only be learned for and applied at the problem level, they will, of necessity, be less general than ELF preconditions. That is, whenever a PLF modification applies to a problem, the corresponding ELF modification will always apply somewhere in that problem, but not vice versa. However, the effectiveness of PLF modifications can be greater because the per-node cost should decrease because the rejection heuristic's preconditions are no longer being tested at each node. Assuming the absence of meta-level interference then when a PLF modification applies to a problem, there will be a higher probability that the problem will be solved. Unfortunately, it also means that the overhead may also be much greater because a much larger space may be searched. The question is whether the increased coverage is worth more to the user than any increased costs due to increases in the search spaces.

Table 6.4 compares the performance of the best ELF modification set with a partial performance of a comparable PLF modification set. I tried the PLF set on the enabled problem set. Unfortunately, on every problem attempted in this particular experiment the PLF set always hit the CPU time limit without having solved the problem. This resulted in the experiment taking too long to actually complete. I ended up only running the PLF modification set on 130 of the problems. The ELF nodes created figure is for its attempt on all 302 problems. The PLF nodes created figure is for its attempt on 130 of those problems. While the PLF figure for the 130 problems is much larger than the ELF figure for all 302 problems, I don't know how many more nodes would need to be created for the PLF set to finish its attempts on those 130 problems and on the remaining 172 problems. Because there might have been rejection heuristic interaction,

I do not know how many of the 302 problems the PLF set would have ended up solving. However, these figures suggest that the PLF set had a much larger space to search than the ELF modification set. While there may be situations where the PLF approach would be superior to the ELF approach, this does not appear to be one of them.

6.4.2 Success-Based vs Failure-Based Explanations

Just as modification algorithms can be compared according to the level at which the modifications are applied, they can also be compared according to how the modification preconditions are computed. This section compares the use of modifications whose preconditions have been computed from an explanation of why the system failed to solve the problem with the use of modifications whose preconditions have been computed from an explanation of how the system could have solved the problem.

Since rejection heuristics are used to reduce the branching factor, care is needed to avoid unnecessarily increasing the branching factor. Thus, rejection heuristic should only be modified when there is some impasse that prevents the system from solving a solvable problem which one wants the system to be able to solve in the future. In these instances, one still wants the modification to avoid unnecessarily expanding the search space for other problems. There are two ways that the expansion would be unnecessary: (1) the problem-solver could already solve the problem within the original search space; or (2) the expanded portion of the search space does not contain any solutions to the problem. Failure-based modifications focus on avoiding the former problem, while success-based modifications focus on avoiding the latter. It should be noted that neither approach is entirely successful. While the failure-based modification might correctly predict that the current partial solution cannot be extended into a solution, there might be an earlier choice point which can still lead to a solution. Analogously, while the solution-based modification might enable the problem-solver to achieve a subgoal that it would not otherwise be able to achieve, that may not be sufficient to ensure that the problem-solver will be able to extend the current partial solution into a complete solution for the given problem.

One modifies rejection heuristics because there is some impasse that prevents the

system from solving a given problem. The modification's precondition needs to key the instantiation of the alternative to be relevant to the impasse being resolved. For example, for the success-based approach to be useful for modifying Linearity, it should identify the goals that are interfering so that the modification isn't "unnecessarily" triggered. Consider Sussman's anomaly[15]. Assume one uses the success conditions of the solution to trigger the modification that adds the `(clear a)` goal to the set of top-level goal candidates. However, the reason the modification is necessary is because of the goal interference and therefore to avoid unnecessarily expanding the search space for other problems, that goal interference must be detected by the modification's preconditions. How much of the goal interference description is included in the modification's preconditions can vary. The total goal interference description includes not only which goals interfere but also what must be initially true for the goals to interfere. For example, for Sussman's anomaly one could include not only that goals `(on a b)` and `(on b c)` can interact but also that block `c` must be on top of block `a` for the interference to occur. The less of the description that is included the more likely the search space will be expanded without affecting whether the problem-solver solves the problem. One wants to at least include the goals that are interfering. So, while the success-based approach tries to keep the failure description down to a minimum, it cannot avoid including some part of the failure description.

In addition to wanting the modifications to avoid unnecessarily expanding the search space, one would like them to apply as generally as possible, i.e., if the modification's postcondition would turn an otherwise unsolvable problem into a solvable problem then one would like the modification's preconditions to apply. This is another motivation behind the failure-based approach. While the failure-based approach focuses on identifying when the current search space probably does not contain a solution, it relies on the fact that the alternative it is putting back into the search space did lead to a solution for a similar problem (i.e., failed for the same reason) in the past. The failure-based approach assumes that the entire solution that worked for the past problem may

[15]Sussman's anomaly is a BlocksWorld problem where there are three blocks A, B, and C. In the initial state, C is on A and both A and B are on the table. The goal is to have A on top of B, which is to be on top of C.

not work for this one, but that adding this edge is likely to connect to some solution. For example, in Sussman's anomaly, the specific solution required that `block a` was on the `table` (because the system `pick`ed `a` off of the `table` rather than `unstack`ing it off of some other block). So, if it were presented with a variation of Sussman's anomaly where `block a` was on top of some other block, the success-based modification would not be applicable whereas the failure-based modification would be. There are many such variations of Sussman's anomaly (e.g., `block b` is on top of some other block, some block is on top of `block b`, some block is on top of `block c`, etc.). However, one has no guarantee that the failure-based modification will only be triggered when it can resolve the precondition's impasse.

While intuitively it seems that the failure-based approach will produce modifications that are more generally applicable than the success-based approach and which will less often expand the search space when the problem-solver can already solve the problem, this does not mean that it is generally better than the success-based approach. While this is true in the experiment described later in this section, it need not always be so. A success-based modification can be more generally applicable than its corresponding failure-based modification. Even when they are not more general, this is not necessarily a liability. For example, if most of the problems that are being attempted are truly unsolvable (or need modifications that have not been made yet) then the more general the rule the larger the search space that needs to be exhaustively searched to find out the problem-solver cannot solve the problem. Also, only expanding the search space when the problem-solver would not otherwise be able to solve the problem is not always the best thing to do. If the modification expands the search space so that a solution is found much quicker than in the old search space then (while this problem has not gone from unsolvable to solvable) the problem-solver has improved (at least for this problem, on average it may turn out not to speed up the problem-solver). Thus, it is clear that one approach is not always better than the other. However, one would like to be able to characterize when one approach is better than the other and I will say more about this in the concluding section of this chapter.

ELS Implementation

FS2 is a forward-chaining problem-solver that selects a top-level goal to work on and uses rejection heuristics (called heuristic censors) to reduce the number of alternatives to consider. FS2 learns the rejection heuristics on the fly as it attempts to solve the given problem. When FS2 has exhausted its pruned search space, it looks for heuristically pruned nodes (i.e., nodes that are suspended because of rejection heuristics) to unprune. If it now successfully achieves one of its top-level goals then it identifies the nodes along the solution path that had been unpruned. FS2 now modifies the rejection heuristics that had pruned these nodes by regressing the achieved goal back to the previously pruned nodes. These regressed conditions are then respectively added to the preconditions of the rejection heuristics that had pruned that node. For a more detailed description see Chapter 2.

I tried to use FS2's procedure for computing the modification, but there were two problems with this. One is that the system never finds out what it was about the problem that forced the system to modify its rejection heuristic. This shows up more clearly when the rejection heuristic being modified is Linearity, where the reason a goal may be currently unachievable is because it interacts with another goal. For example, the latter goal may have already been achieved and deleted a condition needed to achieve the former goal and there is no way to re-achieve that deleted condition. However, FS2's approach to modifying the heuristic will never notice this goal interaction because it only looks at the current goal under consideration and thus the modification will never test for this goal interaction. This would overgeneralize the modification and cause it to expand the search space unnecessarily. The other problem arises from trying to use FS2's modification procedure in the context of a means-ends problem-solver instead of in the context of a forward-chaining problem-solver. One way of looking at means-ends planning is to say that it first selects steps to be added to the plan and then actually adds them to the end of the plan when all of their preconditions have been recursively satisfied, whereas, forward-chaining selects the step it wants to add and then immediately adds it to the end of the plan. With forward-chaining, the achieved

Rule Number	1	2	3	4	5	6	7	8
Success-based	9	2	3	2	19	6	3	3
Failure-based	6	8	4	3	12	10	19	6

Figure 6.7: Comparison of Coverage Extension by Rule

goal only needs to be regressed back to the place where the step (which would have been pruned by the rejection heuristic) was added to the path. When one does this for means-ends problem-solvers, if the pruned step had preconditions that needed to be subgoaled upon and the modification only regresses back to the addition of that step into the plan, then the modification would not be triggered by the training problem.

Therefore I modified FS2's ELS. Since the only rejection heuristic that was going to be modified was Linearity and this always involved at least two interacting goals, all interacting goals were regressed through their portion of the solution. Since these modifications were going to be used in a means-ends analytical problem-solver, the interacting goals were regressed back to where the pruned step was selected, this means that they are regressed back through all the solution steps that were created by subgoaling to achieve the preconditions, etc., of the step that was pruned. This latter change allowed the computed modification to at least solve the training problem.

BlocksWorld Experiments

Bacall learned the failure-based (ELF) modifications from 8 problems in the BlocksWorld *in situ* problem set. **Bacall** learned success-based (ELS) modifications from the same 8 problems. I then made all[16] possible non-redundant modification sets from these 8 and ran them on the *in situ* problem set to get speed-coverage tradeoff information. Figure 6.7 shows how much each of the learned rules extended PRODIGY's coverage. For example, the ELS modification learned from the first training problem extended PRODIGY's coverage by 9 problems while the ELF modification learned from that training problem only extended PRODIGY's coverage by 6 problems. However,

[16]There were 120 non-redundant modification sets constructible from the 8 ELS-based modifications.

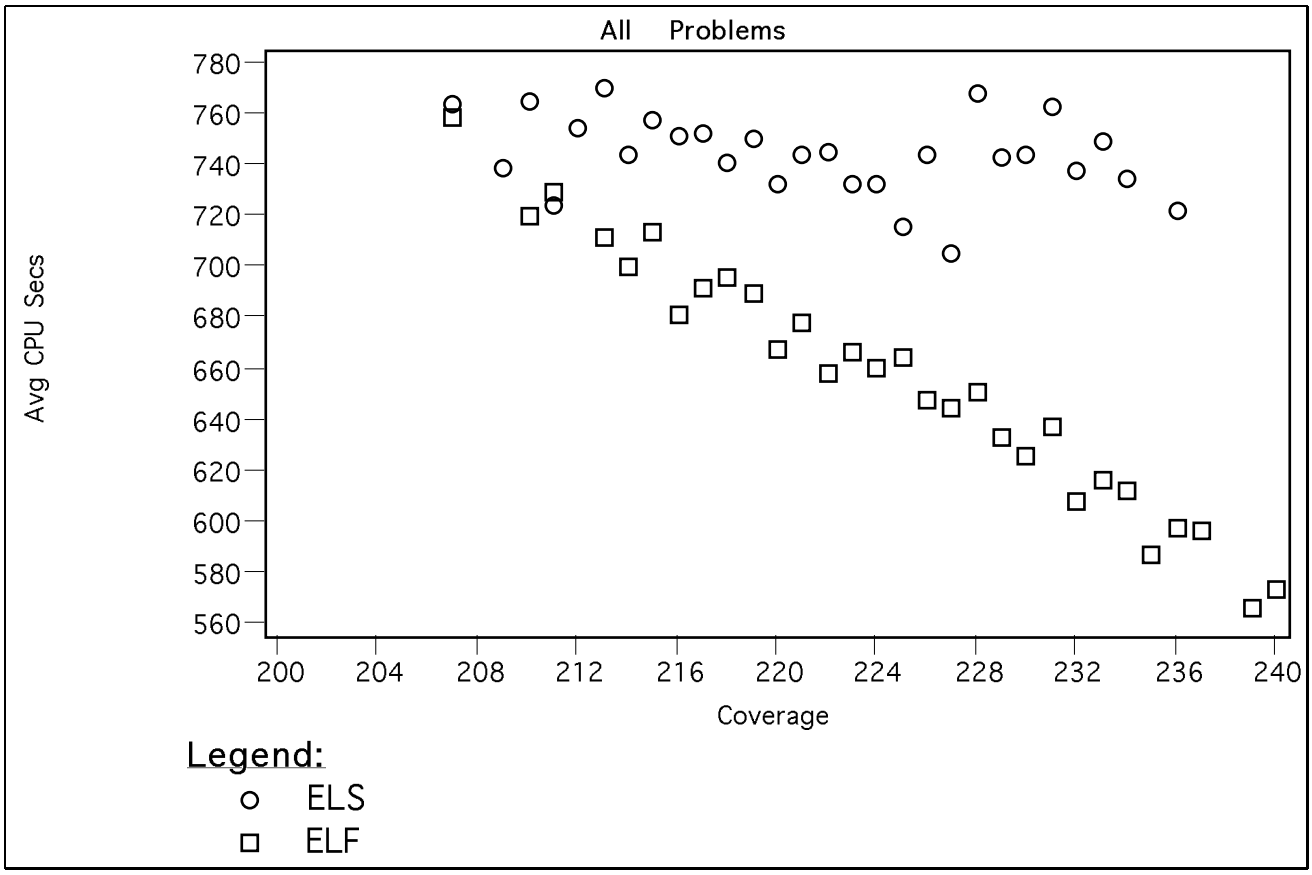

Figure 6.8: ELS-ELF BlocksWorld Data on All Problems

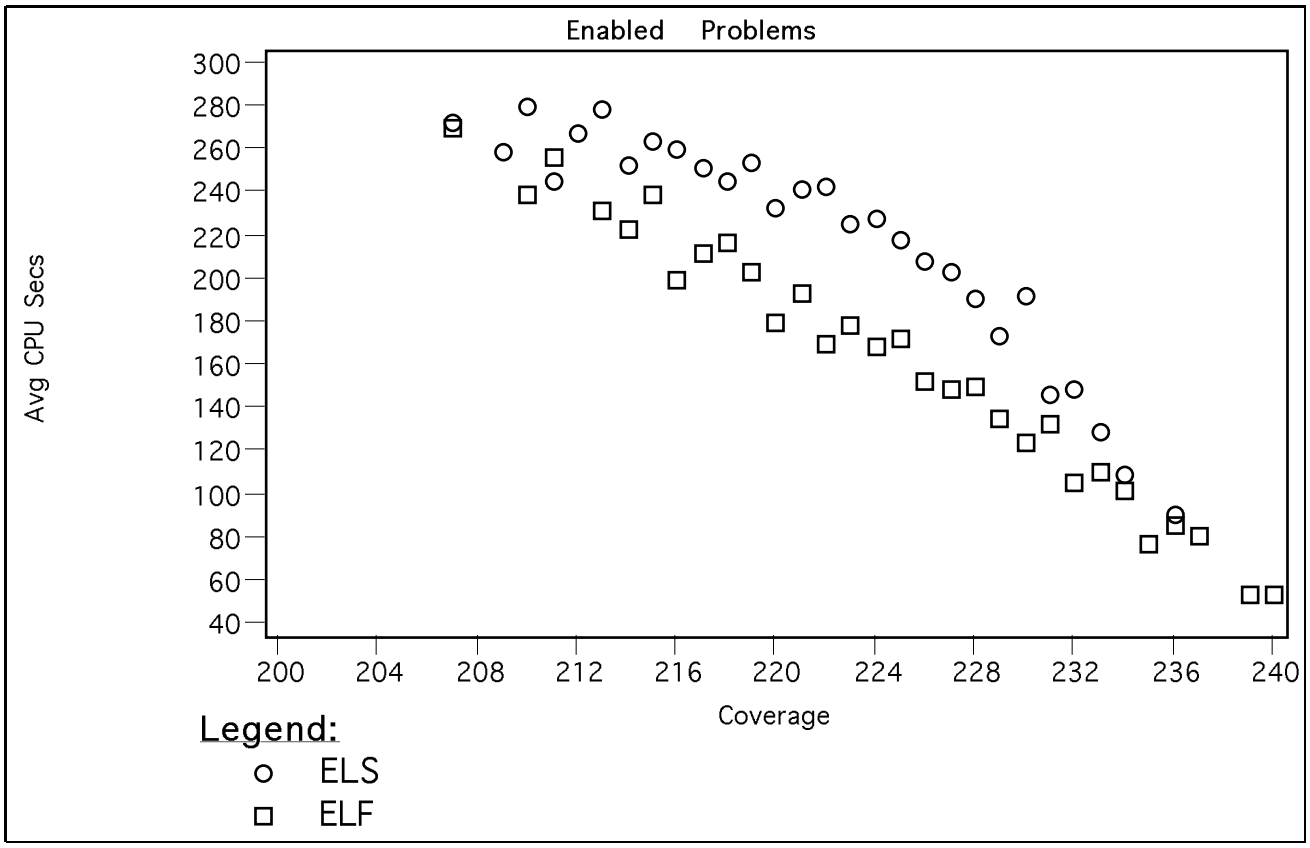

Figure 6.9: ELS-ELF BlocksWorld Data on Enabled Problems

the ELS modifications only extended PRODIGY's coverage by an average of 5.9 problems, while the ELF modifications extended PRODIGY's coverage by an average of 8.5 problems. This supports my intuition that failure-based preconditions should be (on average) more general than success-based ones.

Figure 6.8 shows the relative performance of the ELS and ELF modification sets over all problems. As the coverage increases, the ELF modification sets are progressively faster than the corresponding ELS modification sets. Figure 6.9 shows their relative performance on the enabled problems. While ELF's performance usually seems to be superior to ELS's, it is certainly not enough to explain the difference one sees in Figure 6.8. Also, it appears that as coverage increases, the difference in their performance may actually be decreasing for this set of problems.

Figure 6.10 shows the relative performance of the ELS and ELF modifications sets on the base problem set. It shows that for most of the initial change in coverage, both

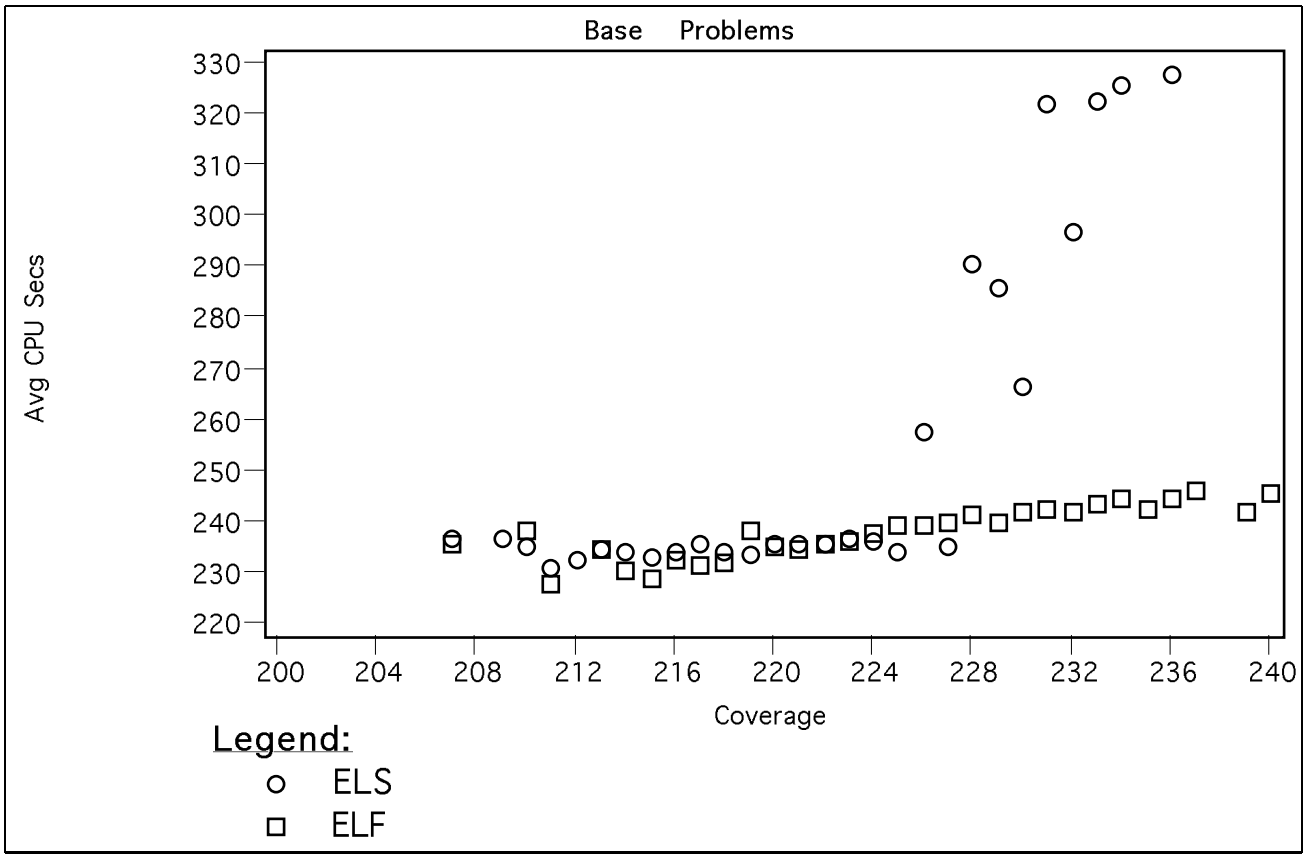

Figure 6.10: ELS-ELF BlocksWorld Data on Base Problems

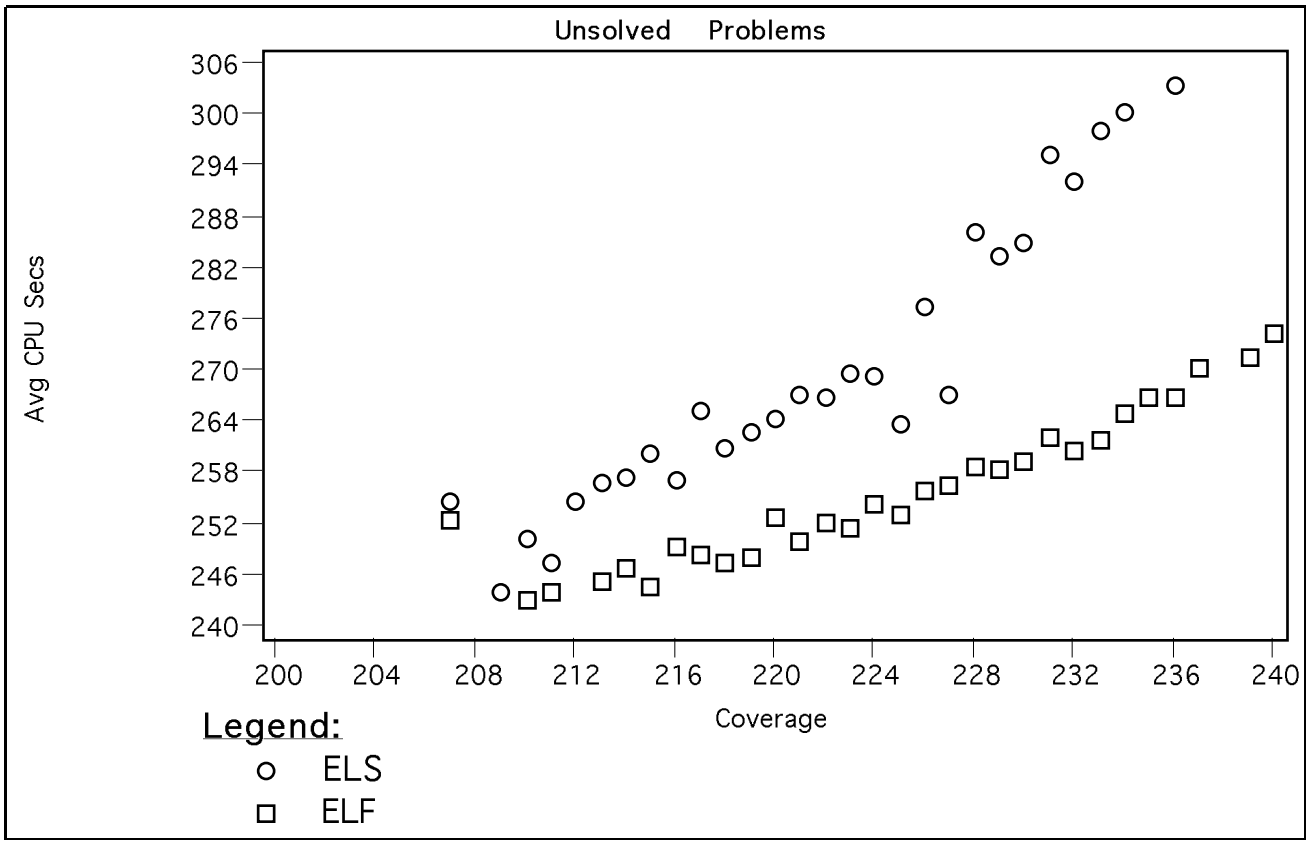

Figure 6.11: ELS-ELF BlocksWorld CPU Time Data on Unsolved Problems

the ELS and ELF modification sets perform equally well and that only in the latter part of the graph do the ELS modification sets perform significantly worse than the ELF modification sets. In particular, when the coverage is 226 problems, ELS starts performing worse than ELF. Why is this happening? On Figure 6.7, there was one rule, rule number 5, that had much greater coverage extension than any other ELS rule. Rule number 5 extended the base coverage (of 207 problems) by 19 problems giving a coverage of 226 problems. Unfortunately, while rule number 5 is good at extending the search space to include solutions, it is also good at extending the search space where there are no solutions. However, this only explains part of the overall performance difference we observed.

Figure 6.11 shows the relative performance of the ELS and ELF modification sets on the unsolved problem set. It shows the same divergence in performance that one sees in the entire problem set. Figures 6.12 and 6.13 show that as the coverage increases the ELS modifications sets both create more nodes and have a higher (CPU sec/node)

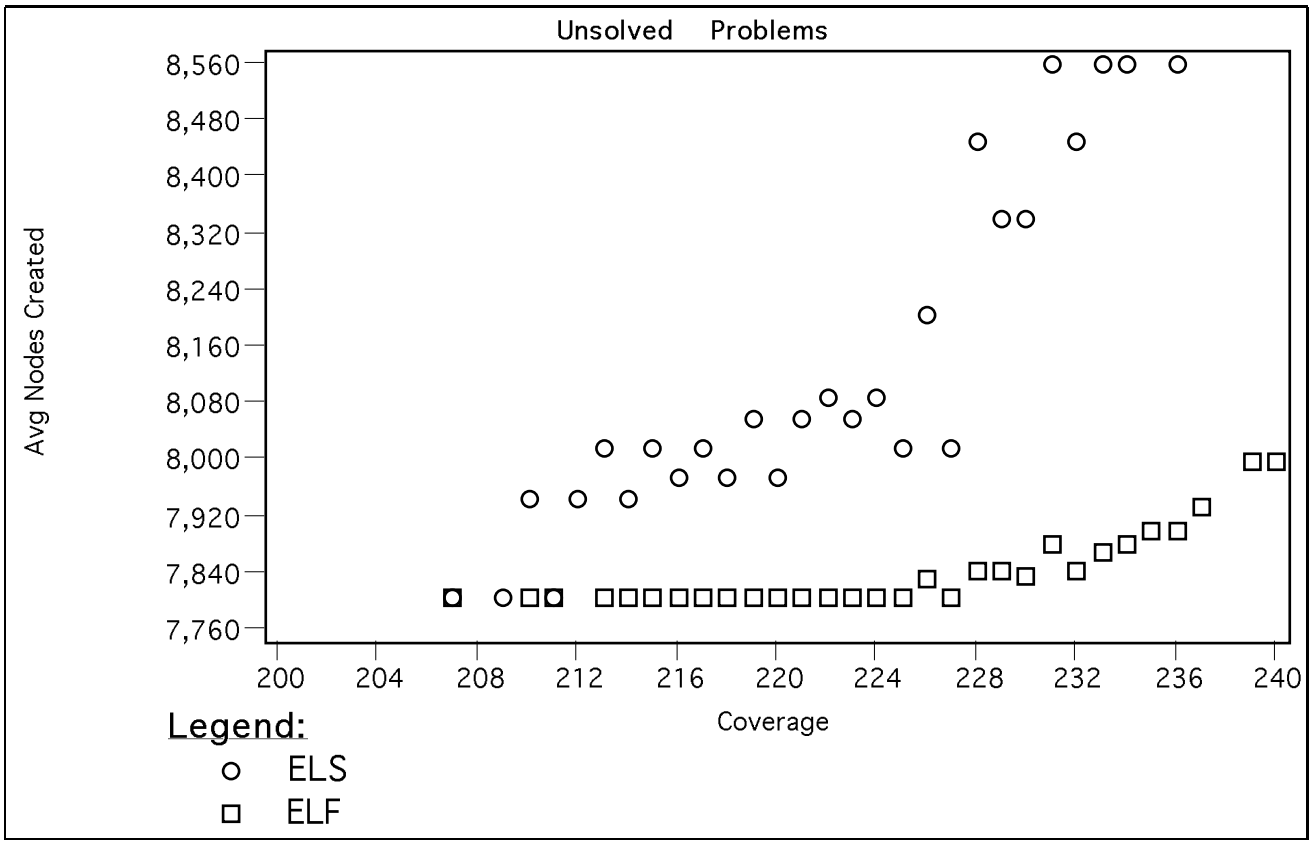

Figure 6.12: ELS-ELF BlocksWorld Node Data on Unsolved Problems

overhead than the ELF modifications sets for the problems in the unsolved problem set. In Figure 6.12, one sees that ELS rule number 5 is having the same effect on the unsolved problems that it did with the base problems.

6.5 Conclusions

6.5.1 Intrinsic Utility

The experiments described above have shown that ELF modifications can indeed improve the performance of problem-solvers using rejection heuristics. In fact, since these modifications are guaranteed to enable the problem-solver to now solve a problem that it couldn't previously solve, the problem-solver's performance cannot monotonically degrade. At worst, the modifications can represent a tradeoff that is not acceptable to the user. At best, the modifications can make monotonic improvements to the problem-solver's average performance. In the BlocksWorld *in situ* experiment I showed that the

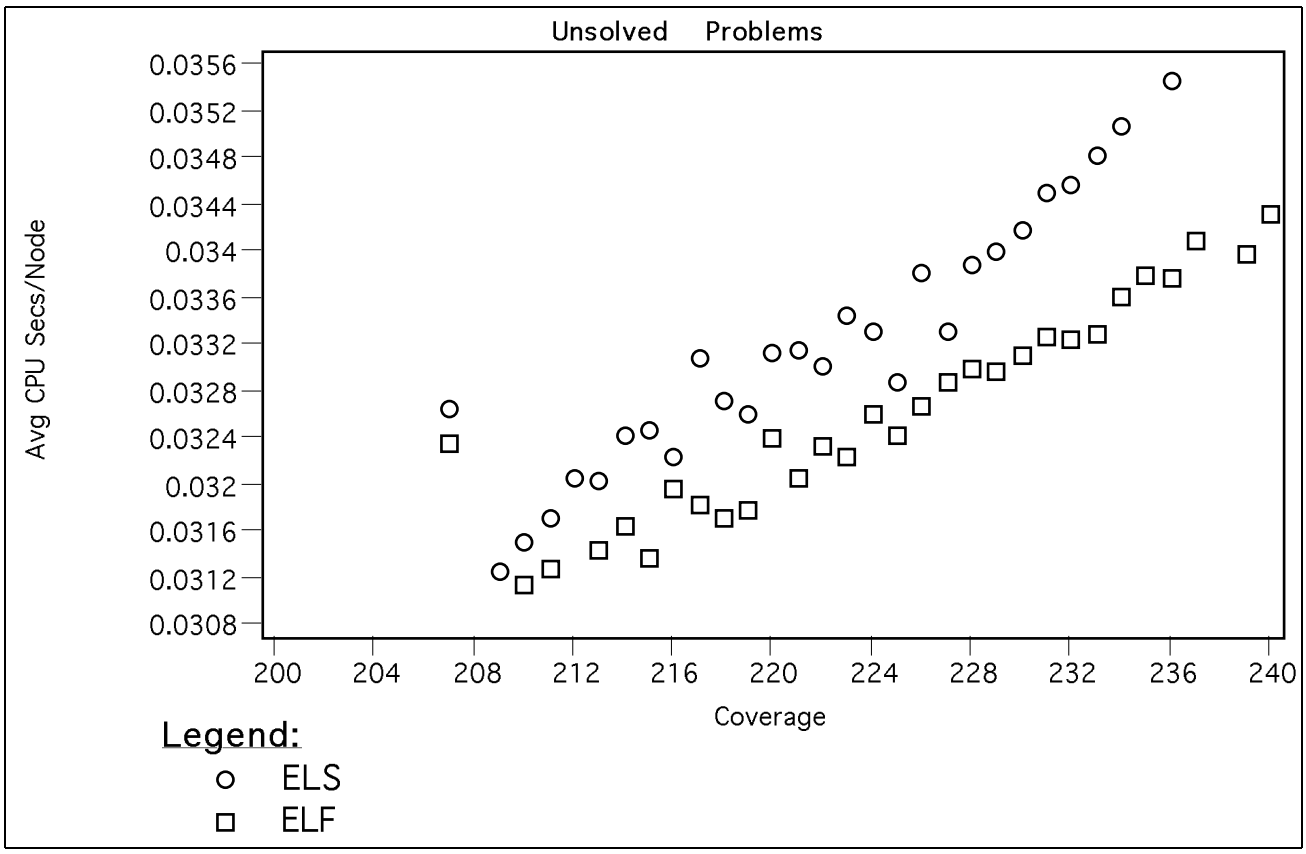

Figure 6.13: ELS-ELF BlocksWorld Time/Node Data on Unsolved Problems

"best" ELF modifications to Linearity enabled PRODIGY to increase its coverage by 15% and its speed by 30%. In the BlocksWorld scalability experiment I showed that increasing the problem size could enhance this problem-solving speed improvement. The same modification set increased PRODIGY's coverage of the BlocksWorld scalability problem set only by 10% but decreased its average time per problem by 75%. The scalability experiment used a more expert search control theory than the one used in the *in situ* experiment. This expert search control theory did not change PRODIGY's coverage but it did enable PRODIGY to generate far fewer nodes while attempting to solve a problem. Thus one expects that had the original search control theory been used, then the speed improvement would have been even greater.

In the One-Way STRIPS experiments, I showed that the ELF modifications need not make monotonic improvements to the problem-solver's performance. Instead, the modifications can cause tradeoffs in the problem-solver's performance. For example, in the One-Way STRIPS *in situ* experiment, while the best modification set increased the problem-solver's its coverage by 1300% it also increased the cpu time by 900%. However, the data suggests that, at least in this domain, the percentage increase in cpu time decreases both as the coverage increases and as the problems become larger.

The more eager the rejection heuristics are to prune away alternatives the more likely that ELF modifications will increase coverage at the expense of problem-solving speed, the extreme case being where the rejection heuristics reject all alternatives. This will be a very fast problem-solver with a very small (i.e., non-existent) coverage. In this case failing to solve problems is very cheap and extending coverage will obviously slow the problem-solver down. In general, where failing to solve problems is expensive, extending coverage should speed up the problem-solver.

6.5.2 Comparative Utility

The experiments above have also shown that for each of the existing methods of modifying rejection heuristics, there are situations where ELF is better. However, ELF modifications will not always be better than either approach. It is not always possible to determine when a rejection heuristic will be unsuited to a domain, i.e., where

the rejection heuristic is almost always wrong. When this is the case, problem-level failure-based (PLF) modifications may be better. It may be cheaper to suspend it at the problem level than to test for its suspension at the edge level. However, in domains where the rejection heuristics are reasonable, ELF modifications should be better than PLF modifications. In general, ELF modifications will: (1) expand the search space less; (2) have less potential for meta-level interference; and (3) be triggered on more problems. In general, PLF modifications will decrease the per node cost when triggered. However, since the lowered per node cost is essentially a constant factor while the search space expansion is a exponential factor, one would expect that in most cases the lowered per node cost will be wiped out by the increase in the search space size. However, this is a simplified analysis and more work is needed to determine how well it holds in practice.

While it seems clear that, in general, ELF modifications will produce more improvement than corresponding PLF modifications, it is not clear that either ELF or ELS will, in general, produce more improvement than the other. This is because, in general, one does not want to expand the search space unnecessarily. ELS focuses on expanding the search space when the expansion will resolve the "impasse" described in its precondition even though it may not actually be an impasse in this instance. ELF focuses on expanding the search space when the current search space has an impasse even though its expansion may not contain a resolution of that impasse. Thus both are incomplete descriptions of when the search space expansion is necessary. ELF modifications seem to be more generally applicable than ELS modifications. This seems to cause the ELS modifications to have less overlap between which problems they cause to become solvable. This means that for a particular coverage more ELS modifications may need to be learned, but also that for that particular coverage, less of the search space may need to be expanded. Unfortunately, the choice between the ELS modifications or the ELF modifications often turn into a choice between different tradeoffs and this can only be decided by the user's needs. However, it is clear that both success-based and failure-based descriptions can be useful in modifying rejection heuristics. I believe that much more useful modifications can result from a better mixing of the success and the failure

descriptions. This is something that needs to explored in the future.

Chapter 7

Conclusion

In this chapter I summarize the results discussed in this thesis, and briefly describe some of the current limitations and areas for future research.

7.1 Summary of Results

7.1.1 SHAPeS Algorithm

This thesis described the SHAPeS algorithm for extending a problem-solver's coverage by specializing its rejection heuristics. SHAPeS uses a supplied solution and an annotated theory of the problem-solver to compute edge-level failure-based refinements. These refinements specialize the preconditions of rejection heuristics that prevented the problem-solver from finding the supplied solution. Aside from refining rejection heuristics that are expressed explicitly as search control rules, SHAPeS's use of a annotated theory of the problem-solver enables it to refine rejection heuristics that are embedded within the system's code.

I identified sufficient conditions for when SHAPeS can identify, in linear time, the rejection heuristics to be refined. Previous systems used search in the unpruned search space to identify these rejection candidates. Such searches can be very costly.

If the generators, the rejection heuristics, and the operational predicates for the EBL learner query only the status of the partial solution, then SHAPeS's refinements will monotonically extend the problem-solver's coverage. However, the solutions found by the problem-solver may change.

7.1.2 Bacall Experiments

Bacall is a restricted implementation of the SHAPeS algorithm. Experiments with Bacall show that there are situations where edge-level failure-based (ELF) refinements not only extend the problem-solver's coverage but also make it faster on average. Experiments were also run to compare ELF refinements with both edge-level success-based (ELS) refinements and problem-level failure-based (PLF) refinements.

The experiments comparing ELF and ELS show that neither is inherently superior to the other. ELF refinements tend to overspecialize the rejection heuristic which usually leads to a larger increase in the problem-solver's coverage than do ELS refinements. However, this also means that the space searched increases for the problems which don't become solvable because of the refinement. In these experiments I have shown a domain, BlocksWorld, where ELF is clearly superior because it not only provides greater coverage but also increases the average speed of the problem-solver. I also showed a domain, One-Way STRIPS, where even though ELF provided much greater coverage, it also more dramatically reduced the problem-solver's average speed. It seems unlikely that there is any a priori way to always determine which would be best to use.

Experiments with PLF suggest that when a rejection heuristic prunes away large portions of the search space that problem-level refinements can be too coarse to be cost-effective. For example, in these experiments the PLF refinements caused the problem-solver to often exceed the cpu time limits.

7.1.3 Undesirable Search Control Rule Interactions

This thesis identified three types of undesirable search control rule interaction (namely, candidate-set-testing, in situ candidate-set-updating, and redundant path interactions) and specified sufficient conditions for eliminating them. These interactions make it difficult to analyze the effects of adding or modifying search control rules. In situ candidate-set-updating interactions cause the results of applying a set of search control rules to be order-dependent. Both candidate-set-testing and in situ candidate-set-updating test interactions can cause changes to a set of search control rules to have counter-intuitive

effects, e.g., adding rejection rules causing the search space to become larger, etc. Redundant path interactions make it impossible to predict when adding a new rejection rule will make a problem unsolvable simply by knowing that the new rejection rule does not, by itself, prune away all the solutions in the problem-solver's current search space for that problem.

Candidate-set-testing and in situ candidate-set-updating interactions can be avoided if all the search control rules' preconditions only test the status of the current partial solution candidate instead of testing candidate sets (e.g., whether a condition is a current goal candidate) that are modified during the decision cycle (e.g., rejecting certain goal conditions as current candidates). Redundant path interactions can be avoided by using a systematic search procedure.

7.1.4 Generality

This thesis has argued that the problem of refining rejection heuristics occurs in all search-based systems that use rejection heuristics to trade problem coverage for problem-solving speed. As I argued in Section 5.3, the SHAPeS algorithm not only refines rejection heuristics for the planning task but works for search-based tasks in general.

7.2 Limitations and Future Research

7.2.1 Identifying Rejection Heuristics Needing Refinement

SHAPeS represents one extreme while FS2 and SOAR represent the other extreme in the amount of search needed to identify the rejection heuristics that need to be refined. SHAPeS uses a supplied solution to provide necessary and sufficient constraints for directly identifying the heuristics needing refinement. SOAR and FS2 use no information from the user to constrain its search for the heuristics.

Providing the entire solution may place an undesirable burden on the user, but searching unaided for the heuristics (that need refinements) may take too long. One would like the user and problem-solver to work together to identify these rejection heuristics. One approach is to extend PRODIGY's facility for letting the user manually

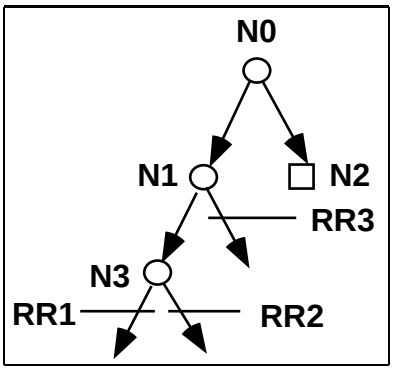

Figure 7.1: Search Tree for Rejection Rule RR4

guide the problem-solver's choice point selections. One extension might allow the user to ask the problem-solver to look at the annotated theory to see all the legal alternatives and allow the user to select one for the problem-solver to try. Another extension might be to allow the user to specify conditions when the problem-solver should stop to ask the user for guidance. For example, the user could specify that when the problem-solver begins to attempt to achieve a certain goal after another specific goal has been achieved that the problem-solver should stop and ask for assistance.

7.2.2 Maintaining Explanation Dependencies

When one uses EBL to either learn or refine a rejection rule, it computes an explanation that depends upon the rejection heuristics that terminated the various search tree paths. If these rejection heuristics are later modified then the explanation may need to be recomputed. For example, let Figure 7.1 be the search tree for problem P from which rejection rule **RR4** was learned. The box at node **N2** denotes a solution there. However, the left subtree under node **N0** leads to failure. A new rejection rule, **RR4**, can be learned that will reject the edge from node **N0** to **N1**. Rejection rules **RR1**, **RR2**, and **RR3** are responsible for all the pruned edges in this part of the search tree. The preconditions for **RR4** are computed by regressing the preconditions of **RR1**, **RR2**, and **RR3** from the edges they pruned back up through node **N1** and become the preconditions for **RR4**. If later one refines **RR1**, **RR2**, or **RR3** then the explanation (for why P would fail to be solved) needs to be recomputed and **RR4**'s precondition updated. If **RR4** has been used to explain other failures, then its update needs to be

propagated to those explanations as well.

However, these modifications are already learned in an ad hoc manner. This is because the problem-solver already learns both rejection rules and their refinements. Even if the dependencies are not maintained, the learner will automatically make these updates in an incremental fashion as they are found to be necessary and/or useful. For example, if propagating the refinement would have made a rejection rule over-general then eventually the problem-solver will be unable to solve a problem, then the user will provide a solution for the system to use to refine that rejection rule. Alternatively, if it would have made a rejection rule over-specific then eventually when the problem-solver explored subtrees that failed to contain solutions, new rejection rules would be learned. It is not clear when propagating the changes will be more cost-effective than using the learner to incrementally make the changes. More research needs to be done to understand the respective merits of the two approaches.

7.2.3 Expanded Refinement Justification

Both failure-based and success-based explanations have problems when used as the sole justification of edge-level refinements to rejection heuristics. Failure-based justifications do not guarantee that the re-introduced edge *will* lead to a solution. Currently the success-based justifications are computed from a solution. Unfortunately, these solution-based success-based justifications tend to be too specific.

One approach is to compute more general success-based explanations is Etzioni's use of problem space graphs (PSG). While Etzioni's STATIC[7] computes the requisite conditions for the guaranteed success of nodes in a PSG, the conditions one wants computed should guarantee success if the given edge is used. I should explore variations of STATIC that can compute these conditions. If, on average, they prove to be too general then one can "and" them with the failure explanation.

Appendix A

Search Control Rule Syntax

The rest of the appendices in this thesis will show examples of search control rules. This appendix will briefly describe the syntax used to represent the search control rules. This syntax comes from the PRODIGY search control rule language.

There are different types of search control rules, namely: select, reject, and preference search control rules. There are different parts of the search which a search control rule can affect, namely: which node to expand; which goal to pursue; which operator with which to pursue it; and how to instantiate that operator's parameters. At the beginning of each group of search control rules is a label or the form: *SCR-part-type-RULES*, where "part" is either node, goal, op, or bindings, and "type" is either select, reject, or preference. Each search control rule is of the form: (NAME (lhs (...)) (rhs (ACT TYPE ITEM))), where NAME is the name of the rule, "lhs" stands for "left-hand-side" of the rule, "..." are the preconditions, "rhs" stands for "right-hand-side" of the rule, ACT is the type of search control rule (e.g., "select"), TYPE is the part being acted upon (e.g., "goal"), and ITEM is a variable which has been appropriately bound by the rule's preconditions. Variables are surrounded by "<" and ">", e.g., <NODE> is a variable named NODE. The preconditions are in normal LISP prefix notation.

Appendix B

Domain Definitions

In this appendix, we describe the domains used in the experiments discussed in Chapter 6. Part of these domain definitions include certain domain-independent search control rules. These will be described first in the next section. The domain definitions for BlocksWorld and for One-Way STRIPS will then be described in subsequent sections. These descriptions include both the specific domain theories and any domain-dependent search control rules that were used in the experiments.

B.1 Domain-Independent Search Control Rules

The following are the domain-independent search control rules used in the experiments:

```
*SCR-NODE-REJECT-RULES*:
 (DEPTH-BOUND-SEARCH
 ;; THIS RULE IMPLEMENTS A DEPTH LIMIT ON THE SEARCH
 ;; THE DEPTH LIMIT WAS DEFINED PER PROBLEM SET
 ;; AS BEING 2 + THE LONGEST OPTIMAL SOLUTION FOR
 ;; ALL PROBLEMS IN THAT PROBLEM SET
  (lhs (and (primary-candidate-node <node>)
            (below-dfid-limit <node>)))
  (rhs (reject node <node>)))
*SCR-GOAL-SELECT-RULES*:
 (SELECT-FIRST-GOAL
 ;; THIS IS A REJECTION HEURISTIC WHICH ASSUMES THAT THE
 ;; ORDER THE PRECONDITIONS APPEAR IN THE OPERATOR'S PRECONDITION
 ;; LIST CORRESPONDS TO THE ORDER IN WHICH THEY SHOULD BE ACHIEVED
```

```
  (lhs (and (current-node <node>)
            (not-top-level-node <node>)
            (primary-candidate-goal <node> <goal>)))
  (rhs (select goal <goal>)))
 (SELECT-ALL-TOP-LEVEL-GOALS
 ;; THIS IS NOT A REJECTION HEURISTIC, THIS RULE SELECTS ALL
 ;; TOP-LEVEL GOALS IN ORDER TO PREVENT BACALL
 ;; LEARNED GOAL SELECTION RULES FROM REMOVING THEM
 ;; FROM CONSIDERATION
  (lhs (and (current-node <node>)
            (is-top-level-node <node>)
            (candidate-goal <node> <goal>)))
  (rhs (select goal <goal>)))
*SCR-BINDINGS-REJECT-RULES*:
 (ADD-GOAL-PROTECTION
 ;; THIS REJECTS ANY OPERATOR'S BINDINGS THAT WOULD ADD
 ;; ANY OF THE PLAN'S CURRENT PROTECTED GOALS
  (lhs (and (current-node <node>)
            (mb-protected-goal <node> <pg>)
            (current-op <node> <op>)
            (mb-candidate-bindings <node> <BINDINGS>)
            (projected-adds <node> <op> <BINDINGS>  <adds>)
            (is-member <pg> <adds>)))
  (rhs (reject bindings <BINDINGS>))))
 (DELETION-GOAL-PROTECTION
 ;; THIS REJECTS ANY OPERATOR'S BINDINGS THAT WOULD DELETE ANY
 ;; OF THE PLAN'S CURRENT PROTECTED GOALS
  (lhs (and (current-node <node>)
       (mb-protected-goal <node> <pg>)
       (current-op <node> <op>)
```

```
      (mb-candidate-bindings <node> <BINDINGS>)
      (projected-deletions <node> <op> <BINDINGS> <deletions>)
      (is-member <pg> <deletions>)))
 (rhs (reject bindings <BINDINGS>))))
(PRECONDITION-GOAL-PROTECTION
;; THIS REJECTS ANY OPERATOR'S BINDINGS THAT WOULD INVALIDATE THE
;; PLAN'S CURRENT SUBGOAL HIERARCHY BY HAVING A PRECONDITION
;; THAT IS INCONSISTENT WITH A CURRENTLY PROTECTED GOAL
 (lhs (and (current-node <node>)
           (mb-protected-goal <node> <pg>)
           (current-op <node> <op>)
           (mb-candidate-bindings <node> <B>)
           (projected-preconds <node> <op> <B> <preconds>)
           (clobbers <pg> <preconds>)))
 (rhs (reject bindings <B>)))
```

B.2 BlocksWorld

In addition to the domain-independent search control rules described in Section B.1, the BlocksWorld domain used additional search control rules. The following describes those search control rules and the domain definitions used in the experiments for the BlocksWorld domain. There are two sets of search control rules that were used for this domain. The first set (called the *naive* search control rules) includes two search control rules. The second set of search control rules (called the *expert* search control rules) includes an additional eight search control rules that dramatically reduces the search space.

B.2.1 Domain Theory

The following is the PRODIGY definition of the **BlocksWorld** domain operators:

```
(PICK-UP
```

```
 (params (<ob1>))
 (preconds (and (object <ob1>)
                (clear <ob1>)
                (on-table <ob1>)
                (arm-empty)))
 (effects ((del (on-table <ob1>))
           (del (clear <ob1>))
           (del (arm-empty))
           (add (holding <ob1>)))))

(PUT-DOWN
 (params (<ob>))
 (preconds (and (object <ob>)
                (holding <ob>)))
 (effects ((del (holding <ob>))
           (add (clear <ob>))
           (add (arm-empty))
           (add (on-table <ob>)))))

(STACK
 (params (<sob> <sunderob>))
 (preconds (and (object <sunderob>)
                (object <sob>)
                (clear <sunderob>)
                (holding <sob>)))
 (effects ((del (holding <sob>))
           (del (clear <sunderob>))
           (add (arm-empty))
           (add (clear <sob>))
           (add (on <sob> <sunderob>)))))
```

```
(UNSTACK
 (params (<us-ob> <underob>))
 (preconds
          (and (object <us-ob>)
               (object <underob>)
               (on <us-ob> <underob>)
               (clear <us-ob>)
               (arm-empty)))
 (effects ((del (on <us-ob> <underob>))
           (del (clear <us-ob>))
           (del (arm-empty))
           (add (holding <us-ob>))
           (add (clear <underob>)))))
```

B.2.2 Search Control Rules

Naive Search Control Rules

The domain-specfic *naive* search control rules (which were used in the in-situ experiments described in Section 6.3.1) were:

```
*SCR-GOAL-PREFERENCE-RULES*:
 (PREFER-WORKING-ON-LOWER-BLOCKS
 ;; THIS HEURISTIC PREFERS WORKING ON GOALS DEALING WITH
 ;; BLOCKS THAT ARE LOWER IN THE DESTINATION STACK THAN
 ;; WITH ONES THAT ARE HIGHER
  (lhs (and (current-node <node>)
            (candidate-goal <node> (on <block1> <block2>))
            (candidate-goal <node> (on <block2> <block3>))))
  (rhs (prefer goal (on <block2> <block3>)
                    (on <block1> <block2>)))
```

```
*SCR-BINDINGS-REJECT-RULES*:
 (EARLY-GOAL-ACHIEVEMENT-ELIMINATION
 ;; THIS HEURISTIC REJECTS BINDINGS THAT WOULD
 ;; CAUSE THE SELECTED OPERATOR TO ACHIEVE A
 ;; CONDITION THAT IS THE SAME AS ONE OF THE
 ;; CURRENT SUPER-GOALS
  (lhs (and (current-node <node>)
            (on-goal-stack <node> <goal>)
            (current-op <node> <op>)
            (mb-candidate-bindings <node> <BINDINGS>)
            (projected-adds <node> <op> <BINDINGS>  <adds>)
            (is-member <goal> <adds>)))
  (rhs (reject bindings <BINDINGS>)))
```

Expert Search Control Rules

The search control rules that begin with SFG, e.g., SFG-PROMOTE-HOLDING-OVER-CLEAR-SEL, are those that were designed to override the SELECT-FIRST-GOAL reject rule (implemented as a goal selection search control rule). The search control rules that were used in the scalability experiments (described in Section 6.3.1) were:

```
*SCR-GOAL-SELECT-RULES*:
 (SFG-PROMOTE-HOLDING-OVER-CLEAR-SEL
  (lhs (and (current-node <node>)
        (candidate-goal <node> (holding <block-1>))
        (candidate-goal <node> (clear <block-2>))
        (is-beneath <node> <block-1> <block-2>)))
  (rhs (select goal (holding <block-1>)))
*SCR-GOAL-REJECT-RULES*:
 (DONT-SEL-GOALS-THAT-MUST-BE-DISTURBED
```

```
  (lhs (and (current-node <node>)
        (candidate-goal <node> (on <x> <y>))
        (known <node> (on <z> <y>))
        (is-beneath <node> <u> <y>)
        (not-equal <u> <x>)
        (or (candidate-goal <node> (on <u> <v>))
            (candidate-goal <node> (on <v> <u>))
            (candidate-goal <node> (on-table <u>)))))
  (rhs (reject goal (on <x> <y>))))
 (SFG-PROMOTE-HOLDING-OVER-CLEAR-REJ
  (lhs (and (current-node <node>)
        (get-sel-goal-scr-bindings
             <node>
             SFG-PROMOTE-HOLDING-OVER-CLEAR-SEL)
        ;; IF SFG-PROMOTE-HOLDING-OVER-CLEAR-SEL FIRED FOR THIS NODE
        ;; THEN THE CLAUSE ABOVE ESTABLISHES THE BINDINGS LIST FROM
        ;;      FROM THAT FIRING (I.E., BINDS <block-2> TO
        ;;      WHATEVER IT WAS BOUND TO THEN) AND RETURNS TRUE
        ;; ELSE IT RETURNS FALSE
        (candidate-goal <node> (clear <block-2>))))
  (rhs (reject goal (clear <block-2>))))
 (REJECT-WORKING-ON-UPPER-BLOCKS
  (lhs (and (current-node <node>)
        (candidate-goal <node> (on <block1> <block2>))
        (candidate-goal <node> (on <block2> <block3>))))
  (rhs (reject goal (on <block1> <block2>))))
*SCR-OP-REJECT-RULES*:
 (CLEAR-BLOCK-BY-UNSTACKING
  (lhs (and (current-node <node>)
        (current-goal <node> (clear <x>))
```

```
        (known <node> (on <y> <x>))
        (candidate-op <node> <op>)
        (not-equal <op> unstack)))
  (rhs (reject operator <op>)))
 (HOLD-BLOCK-BY-PICK-UP
  (lhs (and (current-node <node>)
        (current-goal <node> (holding <x>))
        (known <node> (on-table <x>))
        (candidate-op <node> <op>)
        (not-equal <op> pick-up)))
  (rhs (reject operator <op>)))
*SCR-BINDINGS-REJECT-RULES*:
 (CLEAR-BY-UNSTACKING-BLOCK-ON-TOP-OF-IT
  (lhs (and (current-node <node>)
            (current-goal <node> (clear <x>))
            (current-op <node> unstack)
            (known <node> (on <y> <x>))
            (candidate-bindings <node> (<z> <x>))
            (not-equal <z> <y>)))
  (rhs (reject bindings (<z> <x>))))
 (HOLD-BY-UNSTACKING-IT-OFF-CURRENT-BLOCK
  (lhs (and (current-node <node>)
            (current-goal <node> (holding <x>))
            (current-op <node> unstack)
            (known <node> (on <x> <y>))
            (candidate-bindings <node> (<x> <z>))
            (not-equal <z> <y>)))
  (rhs (reject bindings (<x> <z>))))
```

B.3 One-Way STRIPS

This section describes the domain defintion for One-Way STRIPS that were used in the experiments described in Section 6.4.1.

B.3.1 Domain Theory

The following describes the domain operators. The most notable differences between this definition and the standard definition for the STRIPS domain is that the pickup operator's preconditions do not check whether the robot's hand is empty, allowing the robot to carry an indefinite number of blocks at one time. The other main difference is in the problem definitions, which all use one-way doors (i.e., doors which only connect one room to another but not back again).

```
(PICKUP-OBJ
 (params (<o1> <rm1>))
 (preconds
  (and (inroom <o1> <rm1>)
       (robot-inroom <rm1>)
       (next-to-robot <o1>)))
 (effects
  ((del (inroom <o1> <rm1>))
   (del (next-to-robot <o1>))
   (add (holding <o1>)))))

(PUTDOWN
 (params (<o2> <o2-rm>))
 (preconds
  (and (holding <o2>)
       (robot-inroom  <o2-rm>)))
 (effects
  ((del (holding <o2>))
```

```
   (add (next-to-robot <o2>))
   (add (inroom <o2> <o2-rm>)))))

(GO-THRU-DR
 (params (<ddx> <rry> <rrx>))
 (preconds
  (and (connects <ddx> <rry> <rrx>)
       (robot-inroom <rry>)
       (next-to-door <ddx>)))
 (effects
  ((del (next-to-robot <*19>)) ; robot must be next to door only
   (del (robot-inroom <*20>))
   (add (robot-inroom <rrx>)))))


(GOTO-DR
 (params (<dx> <rx>))
 (preconds
  (and (connects <dx> <rx> <ry>)
       (robot-inroom <rx>)))
 (effects
  ((del (next-to-robot <*18>))
   (del (next-to-door <*dxx>))
   (add (next-to-door <dx>)))))


(GOTO-OBJ
 (params (<b>))
 (preconds
  (and (inroom <b> <rm>)
```

```
      (robot-inroom <rm>)))
(effects ((del (next-to-robot <*109>))
      (del (next-to-door <*drr>))
      (add (next-to-robot <b>)))))
```

B.3.2 Domain Dependent Search Control Rules

One-Way STRIPS had no domain-dependent search control rules. It only used the domain-independent search control rules described in Section B.1.

Appendix C

Transforming PRODIGY into a Backward Chaining Planner

In our experiments in Section 6.4.1, we compare suspending the **Linearity**[1] rejection rule for individual edges in a problem's search space with suspending it for the entire problem. Unfortunately, when the **Linearity** rejection rule is removed, hybrid[2] chaining (which is typical of means-ends problem-solvers) can generate a very redundant search space[3]. This redundancy means that the same partial plan can be visited many times. However, by using strict backward chaining (instead of hybrid chaining) we eliminate this particular type of redundancy. Thus, PRODIGY does strict backward chaining when **Linearity** is not being enforced for any part of the problem and otherwise does the standard means-ends hybrid chaining.

When an operator is selected to be added to the plan that has no unsatisfied preconditions, PRODIGY automatically chains forward. To prevent this, each operator in the plan must have at least one unsatisfied precondition until the backward chaining is completed. To do this we add an artifical precondition, **can-apply-1**, to all domain operators and add a goal select search control rule that adds all the unsatisfied preconditions to the set of goals under consideration. We also add a new domain operator that just adds **can-apply-1** to the world model.

However, when the selected operator has just one unsatisfied precondition, that precondition automatically becomes **PRODIGY**'s selected goal and no goal select

---

[1]Since the **Select First Subgoal** rule is a specialization of **Linearity**, it must also be suspended whenever **Linearity** is suspended.

[2]Hybrid chaining involves doing both backward and forward chaining.

[3]The type of redundancy referred to here occurs when the exact same partial plan (including goal-subgoal structure) is present at different nodes in the search space, but was arrived at by different problem-solving paths.

rules are executed. To enable the goal select rules to be executed, we need to add another new precondition, **can-apply-2**, into all the domain operators. We also need to add a new domain operator that just adds **can-apply-2** to the world model. Both **can-apply-1** and **can-apply-2** preconditions are needed to ensure that PRODIGY runs the goal select rules.

In addition to adding these two new preconditions to all domain operators and these two new operators, we also need the following search control rules to cause PRODIGY to do strict backward chaining:

```
*SCR-GOAL-SELECT-RULES*:
 (SELECT-REGRESSED-GOAL
  ;; THIS MAKES ALL NON-ACHIEVED GOALS AVAILABLE
  ;; TO BE SELECTED
  (lhs (and (current-node <node>)
        (~ (known <node> (can-apply-1)))
        (regressed-goal <node> <goal>)))
  (rhs (select goal <goal>)))
*SCR-GOAL-REJECT-RULES*:
 (REJECT-CAN-APPLY-1
  ;; DEFER (CAN-APPLY-1) UNTIL ALL
  ;; OUTSTANDING GOALS HAVE BEEN ACHIEVED
  (lhs (and (current-node <node>)
            (candidate-goal <node> (can-apply-1))
            (candidate-goal <node> <goal>)
            (not-equal (can-apply-1) <goal>)
            (not-equal (can-apply-2) <goal>)
            (~ (is-known <node> <goal>))))
  (rhs (reject goal (can-apply-1))))
 (REJECT-CAN-APPLY-2
  ;; DEFER (CAN-APPLY-2) UNTIL ALL
  ;; OUTSTANDING GOALS HAVE BEEN ACHIEVED
```

```
  (lhs (and (current-node <node>)
            (candidate-goal <node> (can-apply-2))
            (candidate-goal <node> <goal>)
            (not-equal (can-apply-1) <goal>)
            (not-equal (can-apply-2) <goal>)
            (~ (is-known <node> <goal>))))
  (rhs (reject goal (can-apply-2))))
*SCR-GOAL-PREFERENCE-RULES*:
 (PREFER-CAN-APPLY-1
  ;; IF (CAN-APPLY-1) DID NOT GET REJECTED THEN PREFER IT OVER
  ;; ALL OTHERS (BECAUSE THEY MUST ALREADY BE TRUE IN THE
  ;; INITIAL STATE).
  (lhs (and (current-node <node>)
            (candidate-goal <node> (can-apply-1))
            (candidate-goal <node> <G>)
            (not-equal <G> (can-apply-1))
            (not-equal <G> (can-apply-2))))
  (rhs (prefer goal (can-apply-1) <G>)))
 (PREFER-CAN-APPLY-2
  ;; IF (CAN-APPLY-2) DID NOT GET REJECTED THEN PREFER IT OVER
  ;; ALL OTHERS (BECAUSE THEY MUST ALREADY BE TRUE IN THE
  ;; INITIAL STATE).
  (lhs (and (current-node <node>)
            (candidate-goal <node> (can-apply-2))
            (candidate-goal <node> <G>)
            (not-equal <G> (can-apply-1))
            (not-equal <G> (can-apply-2))))
  (rhs (prefer goal (can-apply-2) <G>)))))
```

In addition to getting PRODIGY to do backward chaining, we would like it to do the

same type of pruning when doing strict backward chaining as it does when it does hybrid chaining. There were a number of explicit search control rules that were used in our One-Way STRIPS experiments, namely: **DEPTH-BOUND-SEARCH**, **SELECT-FIRST-GOAL**, **SELECT-ALL-TOP-LEVEL-GOALS**, **ADD-GOAL-PROTECTION**, **DELETION-GOAL-PROTECTION**, and **PRECONDITION-GOAL-PROTECTION**. The first one (**DEPTH-BOUND-SEARCH**) and the goal protection search control rules are all still there. However, the second one is suspended when we do strict backward chaining because it is a specialization of the **LINEARITY** rule and the third one is superceeded by the **SELECT-REGRESSED-GOAL** rule.

There are two rejection heuristics that are directly embedded in PRODIGY's code. One heuristic rejects goal loops, i.e., rejects any subgoal that is the same as a super-goal that is being worked on. The other heuristic rejects state loops, i.e., rejects applying an operator that achieves the same state as one that already occurs earlier in the plan. These heuristics are simulated by the following explicit search control rules that were used in the One-Way STRIPS problem-level modification experiments (described in Section 6.4.1):

```
*SCR-NODE-REJECT-RULES*:
    (REJECT-GOALSET-LOOP
     ;;  THIS IMPLEMENTS GOAL LOOP REJECTION
     (lhs (and (primary-candidate-node <node>)
               (goalset-loop <node>)))
     (rhs (reject node <node>)))
    (REJECT-PROJECTED-STATE-LOOP
     ;; THIS IMPLEMENTS DUPLICATE STATE REJECTION
     (lhs (and (primary-candidate-node <node>)
               (~ (known <node> (can-apply-1)))
               ;;I.E., HAVEN'T FINISHED SUBGOALING YET
               (dup-world-projectedp <node>)))
     (rhs (reject node <node>)))))
```

Appendix D

Examples of Learned Search Control Rules

D.1 Bacall Learned Search Control Rules for BlocksWorld

The following are the eight rules that **Bacall** learned for the BlocksWorld training set.

```
(bacall-select-rule1
   (lhs (and (current-node <@!r74>)
             (candidate-goal <@!r74> (on <@!r73> <@!r72>))
             (known <@!r74> (on-table <@!r72>))
             (candidate-goal <@!r74> (on <@!r72> <@!r71>))
             (candidate-goal <@!r74> (on <@!r70> <@!r69>))
             (candidate-goal <@!r74> (on <@!r68> <@!r67>))
             (known <@!r74> (on-table <@!r68>))
             (not-equal <@!r68> <@!r70>)
             (known <@!r74> (~ (clear <@!r68>)))
             (forall (<@!r66>)
                     (known <@!r74> (object <@!r66>))
                     (and (or (is-equal <@!r66> <@!r73>)
                              (known <@!r74>
                               (~ (on <@!r66> <@!r72>))))
                          (or
                           (and
                            (known <@!r74>
                             (~ (on <@!r66> <@!r68>)))
                            (or (not-equal <@!r66> <@!r70>)
```

```
                              (not-equal <@!r68> <@!r69>)))
                             (and (is-equal <@!r66> <@!r69>)
                              (known <@!r74>
                               (~ (holding <@!r69>)))
                              (forall (<@!r65>)
                               (known <@!r74> (object <@!r65>))
                               (or (is-equal <@!r65> <@!r68>)
                                (is-equal <@!r65> <@!r70>)
                                (known <@!r74>
                                 (~ (on <@!r65> <@!r69>)))))))))))
   (rhs (select goal (clear <@!r68>))))


(bacall-select-rule2
    (lhs (and (current-node <@!r100>)
              (candidate-goal <@!r100> (on <@!r99> <@!r98>))
              (known <@!r100> (on-table <@!r98>))
              (candidate-goal <@!r100> (on <@!r98> <@!r97>))
              (candidate-goal <@!r100> (on <@!r96> <@!r95>))
              (candidate-goal <@!r100> (on <@!r94> <@!r93>))
              (not-equal <@!r94> <@!r96>)
              (known <@!r100> (~ (holding <@!r94>)))
              (known <@!r100> (~ (clear <@!r94>)))
              (forall (<@!r92>)
                      (known <@!r100> (object <@!r92>))
                      (and (or (is-equal <@!r92> <@!r99>)
                               (known <@!r100>
                                (~ (on <@!r92> <@!r98>))))
                           (or
                            (and
                             (known <@!r100>
```

```
                                (~ (on <@!r92> <@!r94>)))
                               (or (not-equal <@!r92> <@!r96>)
                                (not-equal <@!r94> <@!r95>)))
                              (and (is-equal <@!r92> <@!r95>)
                               (known <@!r100>
                                (~ (holding <@!r95>)))
                               (forall (<@!r91>)
                                (known <@!r100> (object <@!r91>))
                                (or (is-equal <@!r91> <@!r94>)
                                 (is-equal <@!r91> <@!r96>)
                                 (known <@!r100>
                                  (~ (on <@!r91> <@!r95>)))))))
                             (or
                              (and
                               (known <@!r100>
                                (~ (on <@!r92> <@!r94>)))
                               (or (not-equal <@!r92> <@!r96>)
                                (not-equal <@!r94> <@!r95>)))
                              (and (is-equal <@!r92> <@!r95>)
                               (known <@!r100>
                                (~ (holding <@!r95>)))
                               (forall (<@!r90>)
                                (known <@!r100> (object <@!r90>))
                                (or (is-equal <@!r90> <@!r94>)
                                 (is-equal <@!r90> <@!r96>)
                                 (known <@!r100>
                                  (~ (on <@!r90> <@!r95>)))))))))))
   (rhs (select goal (clear <@!r94>))))

(bacall-select-rule3
```

```
(lhs (and (current-node <@!r95>)
          (candidate-goal <@!r95> (on <@!r94> <@!r93>))
          (known <@!r95> (on-table <@!r93>))
          (candidate-goal <@!r95> (on <@!r93> <@!r92>))
          (candidate-goal <@!r95> (on <@!r91> <@!r90>))
          (candidate-goal <@!r95> (on <@!r89> <@!r88>))
          (known <@!r95> (on-table <@!r89>))
          (not-equal <@!r89> <@!r91>)
          (known <@!r95> (~ (clear <@!r89>)))
          (forall (<@!r87>)
                  (known <@!r95> (object <@!r87>))
                  (and (or (is-equal <@!r87> <@!r94>)
                           (known <@!r95>
                            (~ (on <@!r87> <@!r93>))))
                       (or
                        (and
                         (known <@!r95>
                          (~ (on <@!r87> <@!r89>)))
                         (or (not-equal <@!r87> <@!r91>)
                          (not-equal <@!r89> <@!r90>)))
                        (and (not-equal <@!r87> <@!r91>)
                         (known <@!r95>
                          (~ (clear <@!r87>)))
                         (known <@!r95>
                          (~ (holding <@!r87>)))
                         (forall (<@!r86>)
                          (known <@!r95> (object <@!r86>))
                          (or (is-equal <@!r86> <@!r89>)
                           (and
                            (known <@!r95>
```

```
                                    (~ (on <@!r86> <@!r87>)))
                                   (or (not-equal <@!r86> <@!r91>)
                                    (not-equal <@!r87> <@!r90>)))
                                  (and (is-equal <@!r86> <@!r90>)
                                   (known <@!r95>
                                    (~ (holding <@!r90>)))
                                   (forall (<@!r85>)
                                    (known <@!r95>
                                     (object <@!r85>))
                                    (or (is-equal <@!r85> <@!r89>)
                                     (is-equal <@!r85> <@!r91>)
                                     (is-equal <@!r85> <@!r90>)
                                     (is-equal <@!r85> <@!r87>))))))))))))
    (rhs (select goal (clear <@!r89>))))

(bacall-select-rule4
    (lhs (and (current-node <@!r113>)
              (candidate-goal <@!r113> (on <@!r112> <@!r111>))
              (known <@!r113> (on-table <@!r111>))
              (candidate-goal <@!r113> (on <@!r111> <@!r110>))
              (candidate-goal <@!r113> (on <@!r109> <@!r108>))
              (known <@!r113> (~ (clear <@!r108>)))
              (known <@!r113> (~ (holding <@!r108>)))
              (forall (<@!r107>)
                      (known <@!r113> (object <@!r107>))
                      (and (or (is-equal <@!r107> <@!r112>)
                               (known <@!r113>
                                (~ (on <@!r107> <@!r111>))))
                           (or
                            (and
```

```
                        (candidate-goal <@!r113>
                         (on <@!r106> <@!r105>))
                        (known <@!r113>
                         (on-table <@!r106>))
                        (not-equal <@!r106> <@!r109>)
                        (not-equal <@!r106> <@!r108>)
                        (known <@!r113>
                         (~ (clear <@!r106>)))
                        (forall (<@!r104>)
                         (known <@!r113>
                          (object <@!r104>))
                         (or
                          (known <@!r113>
                           (~ (on <@!r104> <@!r106>)))
                          (and
                           (is-equal <@!r104> <@!r108>)
                           (not-equal <@!r108> <@!r107>)
                           (forall (<@!r103>)
                            (known <@!r113>
                             (object <@!r103>))
                            (or
                             (is-equal <@!r103> <@!r106>)
                             (is-equal <@!r103> <@!r109>)
                             (known <@!r113>
                              (~ (on <@!r103> <@!r108>)))
                             (is-equal <@!r103> <@!r107>)))))))
                       (known <@!r113>
                        (~ (on <@!r107> <@!r108>)))))))))
(rhs (select goal (holding <@!r108>))))
```

```
(bacall-select-rule5
    (lhs (and (current-node <@!r76>)
              (candidate-goal <@!r76> (on <@!r75> <@!r74>))
              (candidate-goal <@!r76> (on <@!r74> <@!r73>))
              (candidate-goal <@!r76> (on <@!r72> <@!r71>))
              (candidate-goal <@!r76> (on <@!r70> <@!r69>))
              (known <@!r76> (on-table <@!r70>))
              (not-equal <@!r70> <@!r72>)
              (known <@!r76> (~ (holding <@!r74>)))
              (known <@!r76> (~ (clear <@!r70>)))
              (forall (<@!r68>)
                      (known <@!r76> (object <@!r68>))
                      (and (or (is-equal <@!r68> <@!r75>)
                               (known <@!r76>
                                (~ (on <@!r68> <@!r74>))))
                           (or
                            (and
                             (known <@!r76>
                              (~ (on <@!r68> <@!r70>)))
                             (or (not-equal <@!r68> <@!r72>)
                              (not-equal <@!r70> <@!r71>)))
                            (and (is-equal <@!r68> <@!r71>)
                             (known <@!r76>
                              (~ (holding <@!r71>)))
                             (forall (<@!r67>)
                              (known <@!r76> (object <@!r67>))
                              (or (is-equal <@!r67> <@!r70>)
                               (is-equal <@!r67> <@!r72>)
                               (known <@!r76>
                                (~ (on <@!r67> <@!r71>)))))))))))
```

```
    (rhs (select goal (clear <@!r70>))))


(bacall-select-rule6
    (lhs (and (current-node <@!r97>)
              (candidate-goal <@!r97> (on <@!r96> <@!r95>))
              (candidate-goal <@!r97> (on <@!r94> <@!r93>))
              (candidate-goal <@!r97> (on <@!r92> <@!r91>))
              (known <@!r97> (on-table <@!r92>))
              (not-equal <@!r92> <@!r94>)
              (known <@!r97> (~ (clear <@!r95>)))
              (known <@!r97> (~ (holding <@!r95>)))
              (known <@!r97> (~ (clear <@!r92>)))
              (forall (<@!r90>)
                      (known <@!r97> (object <@!r90>))
                      (and (or
                            (and
                             (known <@!r97> (on-table <@!r95>))
                             (candidate-goal <@!r97>
                              (on <@!r95> <@!r89>))
                             (forall (<@!r88>)
                              (known <@!r97> (object <@!r88>))
                              (or (is-equal <@!r88> <@!r96>)
                               (known <@!r97>
                                (~ (on <@!r88> <@!r95>)))
                               (is-equal <@!r88> <@!r90>))))
                            (known <@!r97>
                             (~ (on <@!r90> <@!r95>))))
                           (or
                            (and
                             (known <@!r97>
```

```
                                (~ (on <@!r90> <@!r92>)))
                               (or (not-equal <@!r90> <@!r94>)
                                (not-equal <@!r92> <@!r93>)))
                              (and (is-equal <@!r90> <@!r93>)
                               (known <@!r97>
                                (~ (holding <@!r93>)))
                               (forall (<@!r87>)
                                (known <@!r97> (object <@!r87>))
                                (or (is-equal <@!r87> <@!r92>)
                                 (is-equal <@!r87> <@!r94>)
                                 (known <@!r97>
                                  (~ (on <@!r87> <@!r93>)))))))))))
    (rhs (select goal (clear <@!r92>))))

(bacall-select-rule7
    (lhs (and (current-node <@!r46>)
              (candidate-goal <@!r46> (on <@!r45> <@!r44>))
              (candidate-goal <@!r46> (on <@!r43> <@!r42>))
              (known <@!r46> (on-table <@!r43>))
              (not-equal <@!r43> <@!r45>)
              (known <@!r46> (~ (clear <@!r43>)))
              (forall (<@!r41>)
                      (known <@!r46> (object <@!r41>))
                      (or (and (known <@!r46>
                                (~ (on <@!r41> <@!r43>)))
                               (or (not-equal <@!r41> <@!r45>)
                                (not-equal <@!r43> <@!r44>)))
                          (and (is-equal <@!r41> <@!r44>)
                               (known <@!r46>
                                (~ (holding <@!r44>)))
```

```
                                (forall (<@!r40>)
                                 (known <@!r46>
                                  (object <@!r40>))
                                 (or (is-equal <@!r40> <@!r43>)
                                  (is-equal <@!r40> <@!r45>)
                                  (known <@!r46>
                                   (~ (on <@!r40> <@!r44>)))))))))))
    (rhs (select goal (clear <@!r43>))))
```

The next two rules were learned from one problem in the training set and have been treated as two parts of one rule. The bindings are captured by the first rule and passed along to the second rule via the predicate "get-sel-goal-scr-bindings".

```
(bacall-select-rule8a
    (lhs (and (current-node <@!r159>)
              (candidate-goal <@!r159> (on <@!r158> <@!r157>))
              (known <@!r159> (on-table <@!r158>))
              (candidate-goal <@!r159> (on <@!r156> <@!r155>))
              (known <@!r159> (~ (clear <@!r158>)))
              (known <@!r159> (~ (clear <@!r155>)))
              (known <@!r159> (~ (holding <@!r155>)))
              (forall (<@!r154>)
                      (known <@!r159> (object <@!r154>))
                      (or (known <@!r159>
                                 (~ (on <@!r154> <@!r155>)))
                          (and (candidate-goal <@!r159>
                                (on <@!r153> <@!r152>))
                               (known <@!r159>
                                (on-table <@!r153>))
                               (not-equal <@!r153> <@!r156>)
                               (not-equal <@!r153> <@!r155>)
```

```
                                    (known <@!r159>
                                     (~ (clear <@!r154>)))
                                    (known <@!r159>
                                     (~ (clear <@!r153>)))
                                    (forall (<@!r151>)
                                     (known <@!r159>
                                      (object <@!r151>))
                                     (or
                                      (known <@!r159>
                                       (~ (on <@!r151> <@!r153>)))
                                      (and
                                       (is-equal <@!r151> <@!r155>)
                                       (not-equal <@!r155> <@!r154>)
                                       (forall (<@!r150>)
                                        (known <@!r159>
                                         (object <@!r150>))
                                        (or
                                         (is-equal <@!r150>
                                          <@!r153>)
                                         (is-equal <@!r150>
                                          <@!r156>)
                                         (known <@!r159>
                                          (~ (on <@!r150> <@!r155>)))
                                         (is-equal <@!r150>
                                          <@!r154>)))))))))))
    (rhs (select goal (clear <@!r158>))))

(bacall-select-rule8b
    (lhs (and (current-node <@!r159>)
              (get-sel-goal-scr-bindings <@!r159>
```

```
                bacall-select-rule8a)
            (candidate-goal <@!r159> (holding <@!r155>))))
    (rhs (select goal (holding <@!r155>))))
```

D.2 ELS Learned Search Control Rules for BlocksWorld

The following are the eight rules which correspond to the eight ones learned by **Bacall**. Their preconditions were computed by backpropagating the goal achieved through the plan that achieved that goal. The postcondition of the rules are the goal selections that will allow the PRODIGY problem-solver to now solve that problem.

```
(bacall-fs2-select-rule1
    (lhs (and (current-node <node>)
              (candidate-goal <node> (on <b2> <b1>))
              (candidate-goal <node> (on <b1> <b3>))
              (known <node> (on <b3 <b2>))
              (known <node> (clear <b3>))
              (known <node> (arm-empty))
              (known <node> (clear <b1>))
              (known <node> (on-table <b1>))
              (known <node> (object <b3>))
              (known <node> (on-table <b2>))
              (known <node> (object <b1>))
              (known <node> (object <b2>))))
    (rhs (select goal (clear <b2>)))

(bacall-fs2-select-rule2
    (lhs (and (current-node <node>)
              (candidate-goal <node> (on <b2> <b0>))
              (candidate-goal <node> (on <b0> <b3>))
              (known <node> (on <b3> <b2>))
```

```
                (known <node> (clear <b3>))
                (known <node> (arm-empty))
                (known <node> (clear <b0>))
                (known <node> (on-table <b0>))
                (known <node> (object <b3>))
                (known <node> (object <b1>))
                (known <node> (on <b2> <b1>))
                (known <node> (object <b0>))
                (known <node> (object <b2>))))
    (rhs (select goal (clear <b2>)))

(bacall-fs2-select-rule3
    (lhs (and (current-node <node>)
                (candidate-goal <node> (on <b1> <b0>))
                (candidate-goal <node> (on <b0> <b3>))
                (known <node> (on <b3> <b2>))
                (known <node> (clear <b3>))
                (known <node> (arm-empty))
                (known <node> (on <b2> <b1>))
                (known <node> (object <b2>))
                (known <node> (clear <b0>))
                (known <node> (on-table <b0>))
                (known <node> (object <b3>))
                (known <node> (on-table <b1>))
                (known <node> (object <b0>))
                (known <node> (object <b1>))))
    (rhs (select goal (clear <b1>)))

(bacall-fs2-select-rule4
    (lhs (and (current-node <node>)
```

```
            (candidate-goal <node> (on <b1> <b0>))
            (candidate-goal <node> (on <b0> <b2>))
            (known <node> (on <b3> <b2>))
            (known <node> (clear <b3>))
            (known <node> (arm-empty))
            (known <node> (object <b3>))
            (known <node> (on <b2> <b1>))
            (known <node> (clear <b0>))
            (known <node> (on-table <b0>))
            (known <node> (object <b2>))
            (known <node> (on-table <b1>))
            (known <node> (object <b0>))
            (known <node> (object <b1>))))
    (rhs (select goal (holding <b2>)))

(bacall-fs2-select-rule5
    (lhs (and (current-node <node>)
            (candidate-goal <node> (on <b2 <b1>))
            (candidate-goal <node> (on <b1 <b3>))
            (known <node> (on <b3> <b2>))
            (known <node> (clear <b3>))
            (known <node> (arm-empty))
            (known <node> (object <b0>))
            (known <node> (on <b1> <b0>))
            (known <node> (clear <b1>))
            (known <node> (object <b3>))
            (known <node> (on-table <b2>))
            (known <node> (object <b1>))
            (known <node> (object <b2>))))
    (rhs (select goal (clear <b2>)))
```

```
(bacall-fs2-select-rule6
    (lhs (and (current-node <node>)
              (candidate-goal <node> (on <b2> <b0>))
              (candidate-goal <node> (on <b0> <b3>))
              (known <node> (on <b3 <b2>))
              (known <node> (clear <b3>))
              (known <node> (arm-empty))
              (known <node> (on <b1> <b0>))
              (known <node> (clear <b1>))
              (known <node> (object <b1>))
              (known <node> (on-table <b0>))
              (known <node> (object <b3>))
              (known <node> (on-table <b2>))
              (known <node> (object <b0>))
              (known <node> (object <b2>))))
    (rhs (select goal (clear <b2>)))

(bacall-fs2-select-rule7
    (lhs (and (current-node <node>)
              (candidate-goal <node> (on <b0> <b3>))
              (candidate-goal <node> (on <b2> <b1>))
              (known <node> (on <b1> <b0>))
              (known <node> (clear <b1>))
              (known <node> (arm-empty))
              (known <node> (on <b3> <b2>))
              (known <node> (clear <b3>))
              (known <node> (on-table <b0>))
              (known <node> (object <b3>))
              (known <node> (object <b0>))
```

```
            (known <node> (on-table <b2>))
            (known <node> (object <b1>))
            (known <node> (object <b2>))))
    (rhs (select goal (clear <b0>)))

(bacall-fs2-select-rule8a
    (lhs (and (current-node <node>)
            (candidate-goal <node> (on <b0> <b3>))
            (candidate-goal <node> (on <b3> <b1>))
            (known <node> (on <b3> <b2>))
            (known <node> (clear <b3>))
            (known <node> (arm-empty))
            (known <node> (on <b2> <b1>))
            (known <node> (object <b2>))
            (known <node> (on <b1> <b0>))
            (known <node> (object <b1>))
            (known <node> (on-table <b0>))
            (known <node> (object <b3>))
            (known <node> (object <b0>))))
    (rhs (select goal (clear <b0>)))

(bacall-fs2-select-rule8a
    (lhs (and (current-node <node>)
            (get-sel-goal-scr-bindings <node>
                bacall-select-rule251)
            (candidate-goal <node> (holding <b1>))))
    (rhs (select goal (holding <b1>)))
```

References

[1] A. Barrett and D. Weld. Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1), 1994.

[2] J. Bennett and T. Dietterich. The test incorporation hypothesis and the weak methods. Technical Report TR 86-30-4, Dept. of Computer Science, Oregon State University, Corvallis, Oregon, 1986.

[3] N. Bhatnagar. *On-Line Learning From Search Failures.* PhD thesis, Rutgers University, New Brunswick, NJ, May 1992.

[4] D. Borrajo and M. Veloso. Incremental learning of control knowledge for nonlinear problem solving. In *Proceedings of the European Conference on Machine Learning*, April 1994.

[5] W. Braudaway. *Knowledge compilation for incorporating constraints.* PhD thesis, Rutgers University, New Brunswick, NJ, December 1991.

[6] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3), 1987.

[7] O. Etzioni. *A Structural Theory of Explanation-Based Learning.* PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1990.

[8] O. Etzioni and R. Etzioni. Statistical methods for analyzing speedup learning experiments. *Machine Learning*, 14, March 1994.

[9] O. Etzioni and S. Minton. Why EBL produces overly-specific knowledge: A critique of the PRODIGY approaches. In *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen, Scotland, 1992.

[10] J. Laird, A. Newell, and P. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1), 1987.

[11] S. Lee. *Multi-Method Planning.* PhD thesis, University of Southern California, Los Angeles. CA, April 1994.

[12] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference of Artificial Intelligence*, Anaheim, CA, 1991.

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

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

[15] T. Mitchell, S. Mahadevan, and L. Steinberg. Leap: A Learning Apprentice Approach for VLSI Design. In *Proceedings of the Ninth International Conference on Artificial Intelligence*, Los Angeles, CA, 1985.

[16] J. Pearl. *Heuristics: intelligent search strategies for computer problem solving.* Addison-Wesley, 1984.

[17] M. Perez and J. Carbonell. Control knowledge to improve plan quality. In *Proceedings of the Second International Conference on AI Planning Systems*, June 1994.

[18] A. Segre, C. Elkan, and A. Russell. A critical look at experimental evaluation of ebl. *Machine Learning*, 6(2), 1991.

[19] G. Sussman. *A Computer Model of Skill Acquisition.* American Elsevier Publishing Company, 1975.

[20] M. Veloso. *Learning by Analogical Resoning in General Problem Solving.* PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, August 1992.

Vita

Michael W. Barley

1966 Graduated from Central High School, London, England.

1982 B.A., University of California at San Diego, La Jolla, California, USA.

1983 M.Sc., Brunel University, Uxbridge, England.

1983-85 Teaching Assistant, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, USA.

1984 Summer Intern, Schlumberger Research Lab, Ridgefield, Connecticut, USA.

1985-90 Research Assistant, Department of Computer Science, Rutgers University, New Brunswick, New Jersey, USA.

1990-92 Consultant, Advanced Technology Center, Boeing Computer Services, Bellevue, Washington, USA.

1994-95 Instructor, Cogwell College North, Department of Computer Science, Kirkland, Washington, USA.

1995 Instructor, University of Washington Extension, Seattle, Washington, USA.

1995- Advanced Computing Technologist, Research and Technology, Boeing Information Services and Support, Bellevue, Washington, USA.

1996 Ph.D. in Computer Science, Rutgers University, New Brunswick, New Jersey, USA.