Feature Discovery for Problem Solving Systems

Tom E. Fawcett
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003

CMPSCI Technical Report 93-49
May 1993

FEATURE DISCOVERY FOR PROBLEM SOLVING SYSTEMS

A Dissertation Presented

by

TOM ELLIOTT FAWCETT

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 1993

Department of Computer Science

FEATURE DISCOVERY FOR PROBLEM SOLVING SYSTEMS

A Dissertation Presented

by

TOM ELLIOTT FAWCETT

Approved as to style and content by:

Paul E. Utgoff, Chair

Edwina L. Rissland, Member

Andrew G. Barto, Member

Bernard Silver, Member

James Chumbley, Member

W. Richards Adrion, Department Head
Department of Computer Science

ACKNOWLEDGMENTS

When I consider the people who have contributed to this work and to my education, I am surprised by how many there have been, and by how much I owe them.

My greatest thanks go to Paul Utgoff, my advisor, who guided this work from inception to completion. Paul originally suggested that I look at the problem of automatic feature discovery. He gave me the freedom to pursue my own ideas and to make my own mistakes; if he ever regretted that, he did not let it show. I am very grateful for his unflagging enthusiasm and encouragement. Often when I was dismayed by the complexity of my system or the shortcomings of my work, he would remind me of what I had accomplished. Paul has high standards of research, writing and of scientific inquiry, and I hope that I have lived up to them. I am sure I will continue to learn from him.

I would like to thank the members of my committee: Andrew Barto, James Chumbley, Edwina Rissland and Bernard Silver. Each provided valuable comments on my work from their individual perspectives. The dissertation is much better for it.

I am especially grateful to Bernard Silver, a colleague and a worthy opponent. Over the past few years we have had more discussions about this research than I can remember, and they have helped me clarify many issues and have shaped its development. Bernard provided useful comments on nearly every paper and every presentation of this work. His suggestions contributed to making the work more intelligible to people outside of constructive induction. Bernard helped with the application of Zenith to telecommunications network management, and he evaluated the resulting features. Though I am much less critical than I was when I began graduate school, Bernard has kept me from mellowing out too much.

I have had great officemates while at the University of Massachusetts: Sharad Saxena, Jamie Callan, Carla Brodley and Jeff Clouse. Each has provided comments on my papers and presentations, and each has offered insights from their work that helped me understand my own better. I hope I have helped them in the same way. I am especially grateful to Jamie Callan for many discussions on his work and mine, which are related. Some of the problems that I faced with Zenith he had already faced in developing the CINDI system, and the insights he shared were invaluable. He also advised me on how to get over grad school hurdles while keeping my sanity.

I consider myself very fortunate to have been associated with the Self-Improving Systems Department of GTE Laboratories during my graduate career. I was employed

there full-time for four summers, and part-time over the course of several semesters. I wish to thank John Vittal, William Frawley, Oliver Selfridge, Rich Sutton, Judy Franklin, Glenn Iba, John Doleac and Steven Whitehead. They attended many talks on my research at different points in its development, and they provided valuable comments and encouragement. In addition, much of Zenith's development and experimentation was done on the workstations of their department.

I thank Chris Matheus for his comments on papers and talks, as well as for allowing me to borrow a few diagrams. I thank Rich Sutton for his help on temporal difference learning. Mike Pazzani and Armand Prieditis provided comments on previous papers. Discussions with Nick Flann helped me to understand goal regression of logical expressions. Nick also provided information on checkers strategy and his interetation of Arthur Samuel's checkers features. Thanks to Sharon Mallory for helping along the way.

Quintus Computer Systems Incorporated generously provided a copy of Quintus Prolog, in which the Zenith system is written. The C4.5 program, used in Zenith, was provided by Dr. Ross Quinlan. Jeff Clouse and Paul Utgoff wrote the code upon which Zenith's OTHELLO opponents, Wystan and ORFEO, are based. Jamie Callan provided the code for linear threshold unit training, as well as some of the statistics code.

This research was supported in part by a grant from GTE Laboratories Incorporated, by a grant from the Office of Naval Research through a University Research Initiative Program under contract N00014-86-K-0764, and by occasional loans from my parents.

Most importantly, I would like to thank my parents and my sister Judy. They were the strongest sources of emotional support during my graduate years. They provided love and sympathy, they helped me keep my problems in perspective, and they helped me keep my sanity. I would not have finished my degree were it not for them.

ABSTRACT

FEATURE DISCOVERY FOR PROBLEM SOLVING SYSTEMS
MAY 1993
TOM E. FAWCETT
B.S., WORCESTER POLYTECHNIC INSTITUTE
M.S., RUTGERS UNIVERSITY
Ph.D., UNIVERSITY OF MASSACHUSETTS
Directed by: Professor Paul E. Utgoff

Since Samuel's work on checkers over thirty years ago, much effort has been devoted to learning evaluation functions automatically from examples. Many methods have been developed that, given examples of problem states paired with their desired evaluations, can induce an evaluation function from them. However, all such methods are sensitive to the set of features chosen to represent the examples. If the features do not capture those aspects of the examples that are significant for problem solving, the learned evaluation function may be inaccurate or inconsistent. Typically, good feature sets are handcrafted carefully, and a great deal of time and effort goes into refining and tuning them. Very little work has been done on automatically generating sets of features for problem solving domains, or on explaining why known features are useful.

This dissertation presents a method for generating features for problem solving domains. It employs both a declarative problem specificiation and examples of state evaluations, and so combines aspects of both analytical and empirical learning. The feature set is developed iteratively: features are generated, then evaluated, and this information is used to develop new features in turn. Both the contribution of a feature and its computational expense are considered in determining whether and how to develop it further. This method has been applied to two problem solving domains: the Othello board game and the real-world domain of telecommunications network management. Empirical results show that the method is able to generate many known features, several novel features, and to improve concept accuracy in both domains.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This dissertation is about changing representations automatically in order to improve problem solving. It is inspired by the representational shifts that humans seem to experience: as people become more adept at solving problems in a domain, not only does their performance improve but they also seem to change the very way in which they think about the problems [Chase & Simon, 1973]. As people evolve from novices to experts, they perceive a problem situation less in terms of its surface details and more in terms of how it relates to their own goals, plans and strategies for solving problems. In turn, such representation shifts seem to facilitate further problem solving improvement.

This dissertation presents a theory of how representation shifts may be accomplished by intelligent problem-solving programs. It has long been known that programs can improve their own performance through experience, but the improvement depends critically on the quality of the representations given to them by their authors. Programs should be able to change their own representations for the same reason that humans seem to change theirs: because it enables better understanding of problem states and because it facilitates learning.

1.1 Problem Solving as Search

Search is a universal aspect of problem solving and intelligence. Nearly every intelligent task, no matter how constrained, requires some kind of search. Indeed, Newell and Simon (1972) proposed the *problem-space hypothesis* of artificial intelligence, which states that all goal-oriented symbolic reasoning may be expressed as search in a problem space. A stronger form of this hypothesis [Newell, 1980] states that search in a problem space is a fully general model of human intelligence.

Many artificial intelligence (AI) systems implicitly view problem solving as search. Some of these systems have been proposed as general problem solving architectures; for example Newell and Simon's (1963a) General Problem Solver, the SOAR system [Laird, Rosenbloom & Newell, 1986] and the PRODIGY system [Minton & Carbonell, 1987]. Search through a problem space is also the basis for systems that perform

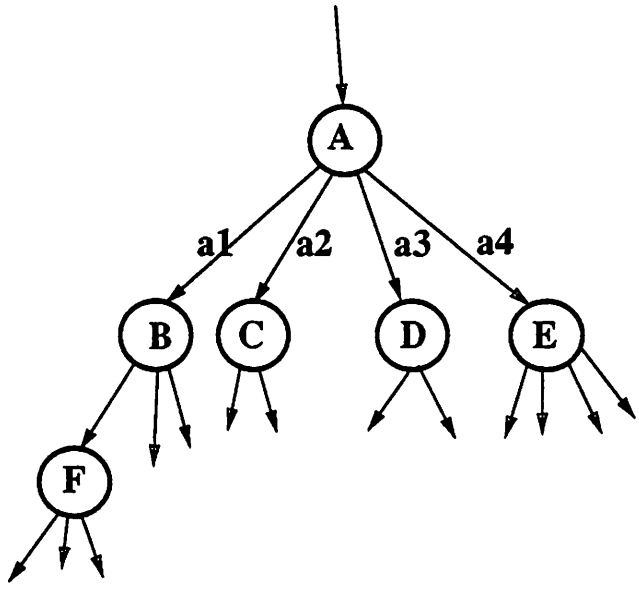


Figure 1.1 An illustration of part of a state-space. The circles are states, and the arrows are actions that can be taken in a state. Legal actions in state A are a1, a2, a3 and a4; the resulting (successor) states are B, C, D and E respectively.

---

specific tasks such as theorem proving [Gelernter, 1959; Newell, Shaw & Simon, 1963b; Kowalski, 1969], planning [Doran, 1969; Fikes, Hart & Nilsson, 1972; Sacerdoti, 1974], game playing [Samuel, 1959; Slagle & Dixon, 1969; De Jong & Schultz, 1988; Schaeffer, Culbertson, Treloar, Knight, Lu & Szafron, 1990], puzzle solving [Schofield, 1967; Michie & Ross, 1969; Iba, 1989] and integral calculus [Mitchell, Utgoff & Banerji, 1983].

Several general methods have been developed for searching a problem space. Problem-reduction methods [Nilsson, 1971] transform a goal specification recursively until it yields conditions that are satisfiable in the initial state. Means-ends analysis [Newell & Simon, 1963a] uses differences between the initial and final state to suggest transformations to the problem, then recursively attempts to solve the resulting transformed problem. State-space search begins at the initial state and searches forward, applying operators to states, until a goal state is reached.

1.2 State Space Search

In *state space* search, each problem step leads to a separate state, and states are implicitly linked together by legal problem steps to form a space, as illustrated in Figure 1.1. Each circle in the figure represents a state. Each arrow represents

an action that can be taken in a state, at the tail of the arrow, producing a new (successor) state, at the head of the arrow.

In state space search, problem solving begins at an initial state. The problem solver selects and applies an action, which determines the next state. The problem solver repeatedly applies actions, moving through the state space, until a goal state is reached. For example, in chess, a state is a chess board configuration, the actions in a state are the legal moves available in that position, and a goal state is any board in which a player has been checkmated or stalemated. The problem solver's objective is to find the best goal state, which may be either the goal state of highest-value, as in OTHELLO; or the closest goal state, one that is reachable using the least expensive sequence of actions.

One way to find the best goal state is to search the entire space exhaustively. However, in many domains the search space is prohibitively large, making exhaustive search infeasible. For example, chess is estimated to have 10^{120} legal board positions (search states). Instead, AI systems perform limited look-ahead in order to determine which of the successor states is best. By searching ahead for a short distance and evaluating the resulting states, a problem solver can get information about the consequences of the available actions without incurring the prohibitive expense of exhaustive search.

Many general search techniques have been developed that use this approach of limited look-ahead. However, all of them depend critically upon the method used for evaluating search states. In their textbook on artificial intelligence, Rich and Knight (1991, Page 63) observe that:

> [General purpose search techniques] can be described independently of any particular task or problem domain. But when applied to particular problems, their efficacy is highly dependent on the way they exploit domain-specific knowledge since in and of themselves they are unable to overcome the combinatorial explosion to which search processes are vulnerable... These techniques continue to provide the framework into which domain-specific knowledge can be placed, either by hand or as a result of automatic learning.

1.3 Guiding Search with an Evaluation Function

Domain-specific knowledge can be used to guide search by incorporating it into a *heuristic evaluation function* [Hart, Nilsson & Raphael, 1968]. An evaluation function orders search states according to their estimated desirability or worth to the problem solver. Evaluation functions are called heuristic because they are usually estimates or approximations of state quality, rather than exact measurements. A problem solver can use these estimates to direct its look-ahead; it can invest greater effort in

pursuing states that are highly rated, and defer (or ignore) search states that appear unpromising. The details of evaluation function-based search methods vary, but all depend critically upon the evaluation function to produce accurate values.

Evaluation functions employ a set of *features* with which to describe states. The set of features determines the representation of the states to the evaluation function. The *primitive* features of a domain are low-level features that are sufficient for describing states unambiguously. For example, in chess the primitive features would specify information such as the contents of every square on a chess board, whose turn it is, the game history for stalemate detection, and so on. However, primitive features often are not suitable for use in an evaluation function because they do not capture aspects of states useful for determining their desirability to the problem solver. The evaluation function usually requires higher-level features that measure aspects relevant to problem solving. For example, some higher-level features for chess are:

1. Whether the White king is in check. This is a simple binary feature whose value is zero or one.

2. The number of moves that White can make. This is a numeric feature.

3. The number of White pieces that are threatened by Black pieces. This is also a numeric feature whose values are between zero and sixteen.

An evaluation function must also specify how the values of these features are combined into a single numerical score. One of the simplest ways of combining feature values is to assign each feature a weight, and add up the product of each feature value and its assigned weight, *i.e.* to construct a linear combination of feature values.

Typically, a human designs an evaluation function for a domain by analyzing the factors that are necessary to produce good decisions, then by designing features that express these factors, and then by deciding how the feature values will be combined into a score. The evaluation function can then be given to a general purpose search routine and used to guide problem solving in the domain.

Unfortunately, neither designing features nor deciding how they should combine is easy for humans. Usually, both tasks require the human to be an expert in the domain. The expert must spend a great deal of time trying different features in different combinations. The process is a time consuming cycle of trial and error:

1. The human expert designs an evaluation function, programs it by hand into an performance system, and waits while the system solves problems.

2. The expert then scrutinizes the problem solving traces and locates mistakes: inferior actions that were taken because evaluations of states were incorrect.

3. The expert then guesses what changes must be made to the evaluation function in order to reduce the mistakes. The expert can decide to create a new feature, modify an existing one, or change the way in which feature values are combined.

Every time the evaluation function is changed it must be tested again. The process continues until the system's performance is deemed acceptable.

1.4 Automating the Generation of Evaluation Functions

Because constructing evaluation functions by hand is a difficult, time-consuming process, much effort in machine learning has been devoted to learning such functions automatically. Although little progress has been made on learning features automatically, many techniques have been developed that can derive evaluation functions automatically, given a good set of features [Samuel, 1967; Berliner, 1979; Lee & Mahajan, 1988; Sutton, 1990; Yee, Saxena, Utgoff & Barto, 1990; Utgoff & Clouse, 1991b].

A common approach to learning an evaluation function automatically is to treat it as a function to be approximated, given examples of its values at different points. In order to do this, examples are created from a problem solving trace and are used to induce an evaluation function. Two classes of methods exist for generating examples from problem solving sequences: temporal difference methods [Sutton, 1988] and state preference methods [Utgoff & Clouse, 1991b]. State preference methods learn a predicate from preferences expressed in the course of problem solving. Temporal difference methods use differences in a function's predictions at successive points to tune the function.

Many functional forms have been explored in learning, such as hyperplanes [Young, 1984], polynomials [Samuel, 1959], trees of linear machines [Utgoff & Brodley, 1991a], and connectionist (neural) networks [Rumelhart & McClelland, 1986]. The feature values may also be converted to symbols and used to learn a symbolic concept [Dietterich & Michalski, 1983; Quinlan, 1987].

Function approximation and inductive concept learning are both being explored actively in machine learning. Techniques have been developed that can learn an evaluation function *given a good set of features*. Because the work on evaluation function learning builds upon existing work in induction and function approximation, much progress has been made already.

1.5 Learning Good Features

In contrast, very little progress has been made on the problem of generating good features. One of the first programs for learning an evaluation function was created by Arthur Samuel (1959), who had to design and code the list of features that his program used. Samuel commented:

> It might be argued that this procedure of having the program select [features] for the evaluation polynomial from a supplied list is much too simple and that the program should generate [features] for itself. Unfortunately, no satisfactory scheme for doing this has yet been devised.[1]

In later work, Samuel (1967) reported considerable progress on methods for combining features, resulting in substantial improvement in his program's performance; yet the problem of automatic feature generation remained "as far in the future as it seemed to be in 1959."[2] Samuel identified the automatic construction of features as a major open problem of great importance.

It has remained an open problem for over thirty years. There are few methods for generating good features for problem solving, and there are no theories that explain how known features were discovered by humans. While many techniques for combining features have been developed in that time, there are few techniques for generating useful features. The lack of progress on feature generation is disappointing. Problem solving systems, some of which approach and exceed human capabilities [Berliner, 1980; Lee & Mahajan, 1988; Schaeffer, Culbertson, Treloar, Knight, Lu & Szafron, 1992], still use hand-coded features.

This lack of progress is also frustrating for a practical reason. A human is still required to devise a suitable feature set for a domain, to evaluate the performance of the program, and to alter the features if necessary. The bulk of development effort is spent generating and testing sets of features, looking for a set that results in effective learning. A good example of the imbalance of effort is Quinlan's (1983) work on decision trees. Quinlan trained a system on examples of a special class of chess end-game configurations. The task was to distinguish between configurations that could lead to a win within 3 moves and those that could not. Once the program was given good features, it was able to learn a decision tree for this problem in 34 seconds. However, it took Quinlan two months of trial and error to design the features.

As another example, Gerald Tesauro (1992a) has developed a system that learns to play backgammon using a temporal difference method. The performance using primitive features is surprisingly good. Nevertheless, adding expert hand-crafted features resulted in substantial performance improvement. As Tesauro states, "It is also possible that a better set of features might give better performance. The features used in Neurogammon were fairly simple, and it is probably the case that the features in Berliner's BKG program or in some of the top commercial programs are more sophisticated and might give better performance."[3] Even systems that are able to generate features internally show sensitivity to the input representation. They still benefit from using sophisticated features that incorporate knowledge of the domain.

---

[1] [Samuel, 1959, Page 87]

[2] [Samuel, 1967, Page 617]

[3] [Tesauro, 1992a, page 455]

Ideally, problem solving systems should learn autonomously, without human intervention. Until feature generation is automated, this goal will remain far from realized. Not only must a human work closely with a learning system, but often the human must be an expert in the very domain that is being learned. Systems should be able to learn in domains in which human expertise has not yet developed.

1.6 Existing Methods for Feature Generation

A few methods for generating features have been developed. The majority of these methods are designed for general classification tasks in which the purpose of classification is not known, so the methods do not exploit knowledge about the domain or information about problem solving. The existing methods may be separated into two classes.

One class of methods may be characterized as *empirical* [Matheus & Rendell, 1989a; Pagallo & Haussler, 1990; Schlimmer & Granger, 1986; Rendell & Seshu, 1990]. Though the details of these methods differ considerably, they all build higher-level features by starting with primitive features and combining them progressively. Such methods incrementally build higher-level features, usually employing feedback from the concept learner to suggest promising combinations. These methods have demonstrated the ability to improve classification performance, and they are domain independent.

However, because the empirical methods begin with the primitive features and combine features one step at a time, these methods are limited in the amount of improvement that they can accomplish. If a desired feature is a complex combination of the primitive features, then such methods must generate and test prohibitively many features before it is derived. These existing methods also suffer because they cannot take advantage of domain knowledge in generating features. They cannot create features using information about the goals and operators of the problem solving system. If the purpose of learning is to improve problem solving performance, such sources of information should not be ignored.

The second class of methods may be termed *analytic*, because they generate features analytically using a domain theory. This approach is exemplified by the STABB [Utgoff, 1986a] system. Analytical methods use information about the domain to deduce appropriate new features, instead of combining primitive features. Because such methods use strong (logically sound) knowledge about the domain, they are able to create complex features in one step, and can guarantee that the features will be useful to the concept learner. However, such systems can only create features that follow deductively from the domain theory. Many domains require useful features that are not deducible from the domain theory, and these analytical systems are incapable of deriving them.

An analytical system also requires that its domain theory be tractable. An intractable domain theory is one that is complete but for which it is computationally prohibitive to construct explanations in terms of the theory [Mitchell, Keller & Kedar-Cabelli, 1986]. Many problem-solving domains have intractable theories: board games such as checkers and chess, VLSI chip design, printed circuit-board layout, and management of telecommunications networks. The analytical systems in constructive induction are unable to construct features for such domains.

1.7 Automatic Feature Generation for Problem Solving Systems

This dissertation presents a theory of feature generation for problem solving systems. Our thesis is that:

> *Useful features for an evaluation function can be created by directed search through a feature space defined by four classes of transformations: goal regression, abstraction, decomposition and specialization.*

By the nature of the transformations, this theory of feature generation combines aspects of both empirical and analytical approaches, but overcomes problems of each. Unlike the empirical approaches, it generates features from a domain theory rather than by combining primitive features. Domain theories may not exist for every domain — for example, the protein folding problem has no known theory — but problem solving domains typically do.

To generate useful features it is necessary to go beyond the strict deduction used by analytical approaches. Features are known to exist in problem solving systems that are strongly predictive of a goal, but do not strictly guarantee it. Strict (logically sufficient) guarantees are expensive, and in most domains they are impractical. The ability to create features that are abstractions and specializations of goals is crucial.

In the following chapters we show that the theory succeeds in two ways. It is *explanatory* in that it describes how many existing features can be generated, and it provides a framework in which the significance of known features can be understood. The theory is also *generative* in that it is capable of generating useful features not previously reported in several domains. Thus the implementation of the theory constitutes an answer to the 30-year old open problem.

1.8 Guide to the Dissertation

The remainder of this dissertation is organized as follows.

In **Chapter 2** we discuss search and search control. We review the work in machine learning on learning evaluation functions automatically, given a good set of features. In **Chapter 3** we then discuss how good features can be created automatically. We review existing work in this subfield of machine learning, called constructive induction. We conclude with a discussion of the limitations of existing approaches, which limitations this thesis seeks to overcome. Our theory is presented in **Chapter 4**, one section devoted to each of its four components. Using examples of known features, Chapter 4 shows how each transformation class can develop and improve features. We conclude the chapter with a presentation of the strategy for controlling the transformations. The Zenith system, an implementation of the theory, is discussed in detail in **Chapter 5**. In **Chapter 6** we discuss the OTHELLO board game, the first domain to which Zenith was applied, and the performance of Zenith in this domain. The second domain, that of telecommunications network management, is discussed in **Chapter 7**. Finally **Chapter 8** presents the lessons learned from Zenith and the conclusions drawn from the empirical investigations. It also discusses the shortcomings of the theory and the future work.

Appendices A and **B** contain the domain theory for OTHELLO and the Zenith's features, discussed in Chapter 6. **Appendices B** and **C** contain the domain theory for Telecommunications Network Management and Zenith's features, discussed in Chapter 7.

CHAPTER 2

CONTROL OF STATE-SPACE SEARCH

This chapter reviews state-space search and evaluation functions, with emphasis on how evaluation functions may be learned. The final section of this chapter briefly reviews alternative means for controlling search that have been investigated in machine learning.

2.1 State-space Search

A state space search problem is a formal characterization of a problem, which contains the following elements:

- A set of states from which search may begin. These are called the *initial states*.
- A set of states that are acceptable as solutions to the problem. These are called the *goal states*.
- A set of *operators* (also called *rules* or *actions*) that describe the legal transitions from one state to another.

Depending on the domain, there may be many possible initial states and many possible goal states. For example, chess has a single initial state (the initial board configuration) but many possible goal states (checkmated or stalemated positions). A goal state may be described implicitly, as in chess, or explicitly, as in the 8-Puzzle.

A *problem instance* is a specific problem, with a specific initial state, to be solved. A *solution* to a problem instance is a path from the designated initial state to a goal state.

The simplest algorithms for searching state spaces are known as brute-force search algorithms. Examples of brute-force algorithms are depth-first search, breadth-first search and backward-chaining search [Gardner, 1981]. Each algorithm visits states in a fixed order until a goal state is reached. These algorithms are undirected because

1. Start with OPEN containing just the initial state.
2. Until a goal is found or there are no nodes left in OPEN do:
   (a) Evaluate the set of nodes in OPEN using the heuristic evaluation function.
   (b) Pick the best (highest value or most preferred).
   (c) For each successor do:
      i. If it has not been generated before, then add it to OPEN, and record its parent.
      ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

Figure 2.1 A best-first search algorithm

they do not reason about the desirability of states or the application of operators in the space. The problem with all brute-force algorithms is that their time complexity increases exponentially with search depth. This tendency for combinatorial explosion severely limits the applicability of the brute-force algorithms.

2.2 Best-first Search

Because of its limitation, brute-force search is rarely used in AI, and some form of best-first search is used instead. Best-first search describes a class of algorithms in which states are evaluated or ranked using a heuristic evaluation function. The values are then used to direct the algorithm to explore more promising nodes before less promising ones. An elementary best-first search algorithm is shown in Figure 2.1. OPEN contains the frontier of the search, consisting of search states that have been generated but not explored. A heuristic evaluation function is used in Step 2a.

A number of best-first search algorithms have been developed, many of which are reviewed by Korf (1988). A* and its incremental version IDA* are applicable to single-agent domains. AO* is a variant of A* that may be used with AND-OR trees, enabling problem reduction in heuristic search. Minimax search [Shannon, 1950] is used in two-player games that models the search process as one in which players on alternating plies are trying to maximize and minimize the evaluations. The evaluations may come from either the heuristic evaluation function or the actual

terminal node values, if the search is sufficiently close to terminal nodes. All best-first search algorithms rely upon heuristic evaluation functions to direct them.

2.3 Directing Best-first Search

In some domains, all goal states are of equal worth and problem solving consists of finding a goal state with minimum effort. That is, the problem solver's goal is to minimize the number of states expanded in the search. In this case, distance-to-goal heuristics are used. A distance-to-goal heuristic is a function from a state onto the non-negative integers that returns an estimate of the distance to the nearest goal state.

2.3.1 Admissible Heuristics

An important class of distance-to-goal heuristics with which AI is concerned is the *admissible* heuristics [Nilsson, 1971]. Admissible heuristics are guaranteed never to overestimate the distance to a goal. This property is desirable because it ensures that the first solution found by A* search will be optimal [Gardner, 1981]. An admissible heuristic must also be effective, of course; it must provide evaluations that are useful for directing search. Effective and admissible heuristics are known for domains such as the Traveling Salesperson problem, tile-sliding puzzles, and Rubik's Cube.

Admissible heuristics have received much attention in artificial intelligence. Research on the nature of admissible heuristics treats them as distance measures on a simplified problem space [Guida & Somalvico, 1979; Gaschnig, 1979]. In this view, a problem space constitutes a graph defined by the initial state, goal state and operators. The problem space can be simplified by, for example, deleting an operator precondition or removing a portion of the goal specification. Simplifying the problem specification is equivalent to adding edges to the original graph, because states that are not neighbors in the original problem space may be neighbors in the simplified problem space [Pearl, 1984]. After a simplified problem space has been constructed, a distance measure can be constructed on it and this measure may then be used as an admissible heuristic in the original problem space. [1]

Figure 2.2, taken from Prieditis's dissertation (1990), illustrates the creation of an edge supergraph. The original problem space produces the original graph at bottom the bottom of the figure. Relaxing the original model results in new edges (dark arrows) being added to the graph, yielding an edge supergraph shown at the top. A

[1] Though much progress has been made with this approach, it has shortcomings. Not every distance measure applied to a simplified problem space yields an admissible heuristic [Prieditis, 1990], and not every admissible heuristic is derivable from a simplified model [Valtorta & Zahid, 1990].

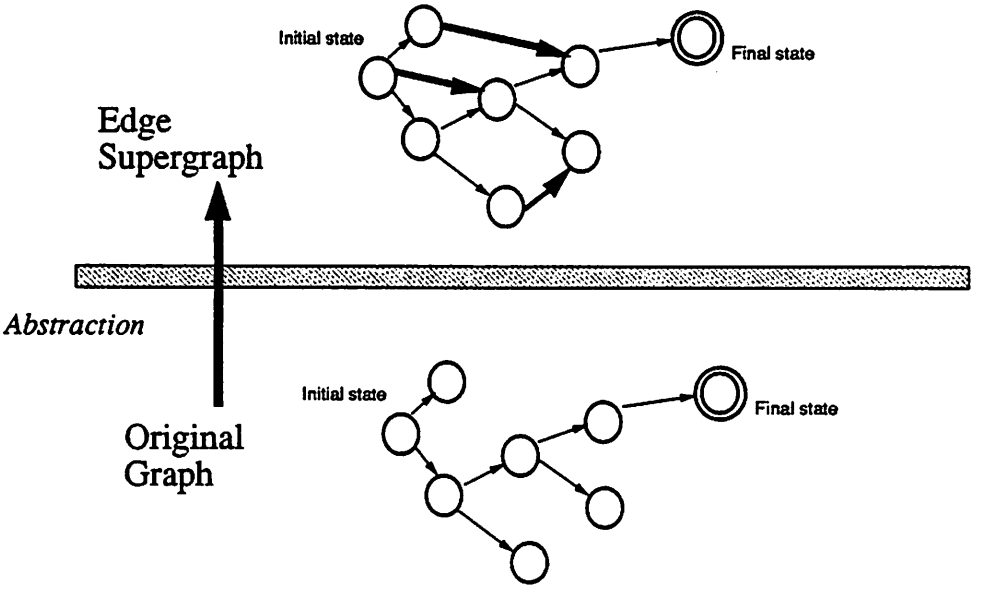


Figure 2.2 An edge supergraph. The dark arrows are the edges added to the original graph by relaxing the model.

---

distance-to-goal function for the supergraph constitutes an admissible distance-to-goal heuristic for the original graph.

Prieditis (1990) has shown that admissible heuristics may be created automatically from a domain theory. His system, ABSOLVER, has two phases. Given a domain specification (a domain model), ABSOLVER first creates a relaxed model for the domain by applying abstracting transformations to the domain specification. The abstracting transformations create relaxed models, for example, by dropping a precondition of an operator or replacing two integers by their sum. After an abstracted problem specification is generated, speedup transformations are applied to it. A speedup transformation can, for example, replace an expression with a closed-form solution, factor a problem into independent subproblems, or replace a calculation with a table lookup. The result of the speedup phase is an exact distance-to-goal metric for the simplified problem. This metric can then be used as a heuristic distance-to-goal estimate in the original problem space. Prieditis proves that all of the transformations used by ABSOLVER preserve admissibility, so any heuristic generated by ABSOLVER is guaranteed to be admissible.

When they can be found, effective admissible heuristics are desirable because they guarantee optimality of solutions found by A*. However, automatic generation of admissible heuristics has been limited to domains in which goal states are of equal value, because only distances are being estimated. In many domains the quality of goals varies, and if the task of problem solving is that of finding a goal of highest quality, then estimating the distance to a goal is no longer adequate for

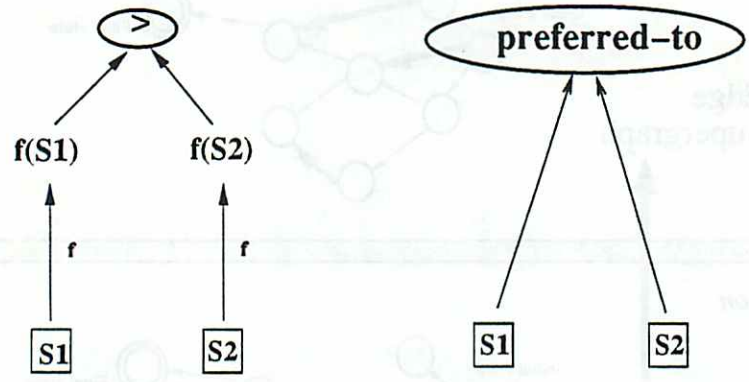


Figure 2.3 Two states ranked by an evaluation function and a preference predicate

problem solving. In addition, automatic generation of admissible heuristics has been applied only to domains that employ STRIPS-style domain models [Sacerdoti, 1974; Waldinger, 1976]. The automatic methods depend upon removing goal conditions and reasoning about the interactions of operator pre- and post-conditions. Generalizing these methods to more general domain models would be a non-trivial task.

2.3.2 Evaluation Functions and Preference Predicates

Because of the limitations of admissible heuristics, more general heuristic evaluation functions are often used instead. A heuristic evaluation function f maps states onto an absolute scale, usually the real numbers. The range values produced by f can be used to rank states. An admissible heuristic is a special kind of heuristic evaluation function that measures the quality of a state as the distance from the state to any goal.

The evaluations of states are only used to determine the states' relative rankings; that is, the values are used only to determine whether one state is to be preferred to another. Instead of using an evaluation function to map states into values, and then using these values to rank the states, a search algorithm can employ a predicate that performs the ranking without assigning values. Such a predicate is called a *preference predicate* [Utgoff & Clouse, 1991b]. A preference predicate $p : S \times S$ is a relation between states such that a pair of states $\langle s_1, s_2 \rangle$ is in p iff s_1 is preferred to s_2 by the problem solver. A search algorithm can use p directly to determine the most preferred of a set of states.

In this dissertation we will use the term "preference predicate" to mean a predicate applied to a pair of states, and the term "evaluation function" to mean a function that maps a state to a numeric value. Figure 2.3 illustrates the difference between using an evaluation function and a preference predicate to rank two states fs_1 and s_2. If an evaluation function f is used, it is applied to each state yielding $f(s_1)$ and $f(s_2)$, and these values are compared using the numerical "greater-than" predicate.

If a preference predicate p is used, the two states are compared directly with the predicate. The function f combined with the "greater-than" predicate comprises a particular kind of preference predicate.

2.4 Learning Evaluation Functions and Preference Predicates

Both evaluation functions and preference predicates may be learned from examples. In machine learning terminology, an *example* is a state or state pair that is to be evaluated by the learned function or concept. A *training instance* is a labeled example, where the label is the value of the example. A training instance for an evaluation function might be $\langle s_{17}, 53\rangle$, indicating that state s_{17} should have the value 53. A training instance for a preference predicate might be $\langle\langle s_{17}, s_{19}\rangle, +\rangle$, indicating that state s_{17} is preferred to state s_{19}; equivalently, $p(s_{17}, s_{19})$ is true. Note that from the training instance $\langle\langle s_{17}, s_{19}\rangle, +\rangle$ its dual $\langle\langle s_{19}, s_{17}\rangle, -\rangle$ may be created, because preferences are antisymmetric.

Creation of examples is not straightforward because the ideal state values or preferences are not readily available. Typically, the information available to a learning system is a problem solving trace consisting of the entire path of states from the initial to the goal state and the final goal value. The trace provides *global* information about problem solving: the value of the goal state given the entire set of choices made. The training information needed to learn an evaluation function is *local*: state evaluations or preferences for nodes throughout the search. To convert the global value into a set of local values, it is necessary to apportion credit or blame for the goal state's value among the choices made in the problem solving trace. This is called the *credit assignment* problem [Minsky, 1963]. All methods that learn from multi-step problem solving must address the credit-assignment problem. Some learning systems employ an explicit component called a *critic* [Widrow, Gupta & Maitra, 1973; Dietterich & Buchanan, 1984; Barto, Sutton & Anderson, 1983] that apportions responsibility to individual problem solving steps. In Temporal Difference (TD) methods, credit assignment is implicit in the weight update equation; the contributions of previous predictions are decayed exponentially, as explained in the next section.

2.4.1 Learning Evaluation Functions

For an evaluation function, training examples are pairs consisting of a state and its desired value. Some methods assign values to states based on some portion of responsibility for the goal state's value [Lee & Mahajan, 1988; Lin, 1991]. Typically, the responsibility (the portion of the goal's value) is inversely proportional to its distance from the goal. This corresponds to the intuition that the further a state is from the final goal state, the less credit or blame it had for the final outcome.

Given these examples, an evaluation function may be learned using any function approximation method for supervised learning [Duda & Hart, 1973].

Some methods update state evaluations incrementally using intermediate evaluations. Incremental updating allows learning to occur throughout performance, rather than just on success and failure. Such methods use differences between temporally successive evaluations to update the evaluation function. These are called *temporal difference* methods [Sutton, 1988] and have been used in a variety of problem solving systems. Samuel's (1959) original checkers player updated the weights of its heuristic evaluation function by comparing a state's evaluation with an evaluation from deeper search. Barto, Sutton and Anderson (1983) used a temporal difference method to drive a pole-balancing system. The system's only terminal state occurred when the system failed, because the pole tipped over or the cart hit one of the end stops of the track. At this point the system received negative reinforcement. Because it only received feedback on failure, the system needed a credit assignment method that could work using only sparse feedback. Holland's (1986) bucket-brigade algorithm assigned credit to classifiers based on their performance in multi-step problem solving. Its system of bids and rewards was a temporal difference method based on temporally adjacent predictions made by the classifiers.

Temporal difference methods are commonly used to adjust the weight vectors in connectionist networks. This is done by computing the difference between the generated evaluation and the temporally successive evaluation, and using it to adjust the weights of the network. For a weight vector w the temporal difference update rule is [Sutton, 1988]:

$$\Delta w_t = \alpha(P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \tag{2.1}$$

where P_i is the prediction (evaluations) at step i, α is a learning rate parameter whose value is between zero and one, λ is a temporal decay parameter between zero and one, and $\nabla_w P_k$ is the vector of partial derivatives of P_t with respect to the each component of the weight vector w. Therefore, an update at step t is influenced by the prediction P_t made in that step, and the newer prediction P_{t+1}, as well as an exponentially decayed sum of past gradients of network outputs with respect to the weights. The parameter λ controls the temporal rate of decay of past gradients: for $\lambda = 0$ the method only uses the current gradient, for $\lambda = 1$ the method sums all past gradients, and for $0 < \lambda < 1$ each gradient contributes exponentially less depending on its temporal distance from the current step.

Equation 2.1 incorporates both credit assignment and learning. Credit is proportional to temporal prediction differences and the influences of previous gradients, mediated by the λ parameter. Learning is accomplished in the connectionist network by updating the weight vector.

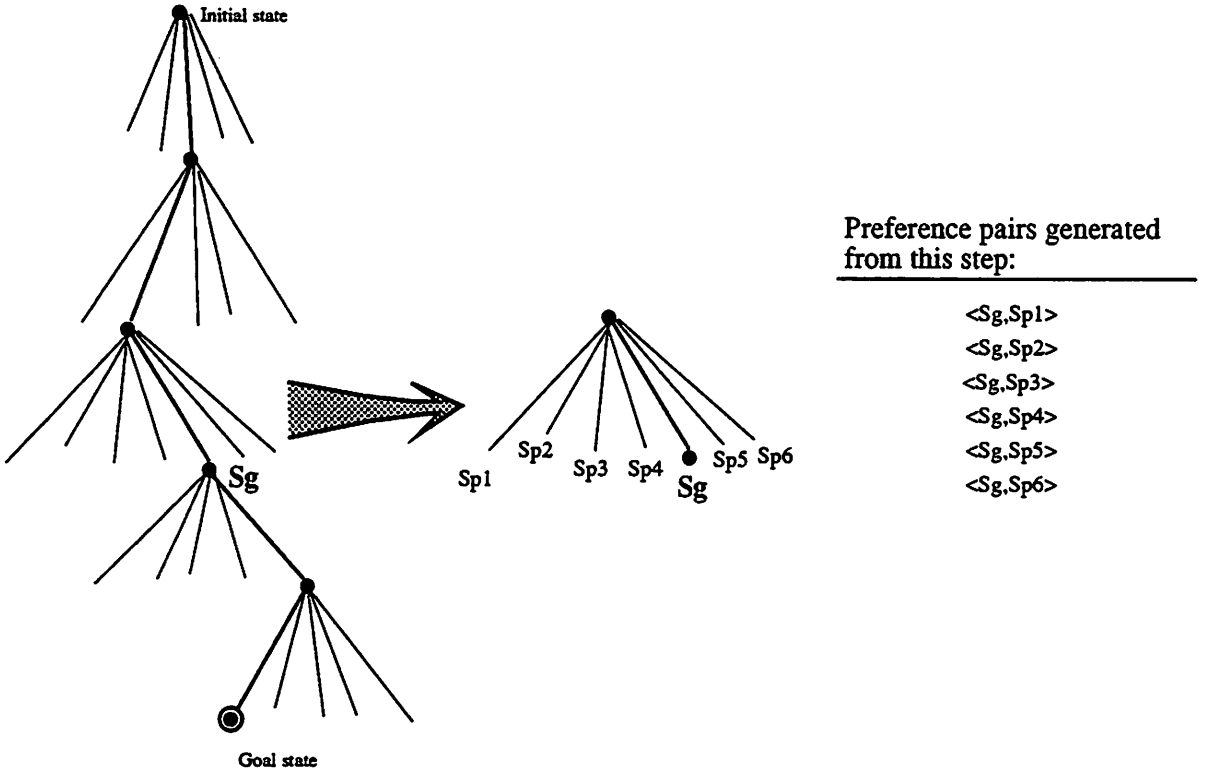


Figure 2.4 Generating preference pairs from a solution path

2.4.2 Learning Preference Predicates

For a preference predicate, a training instance is a pair $\langle\langle s_1, s_2\rangle, +\rangle$ such that state s_1 is preferred to state s_2 by the evaluation function. Such pairs may be created from a problem solving trace as shown in Figure 2.4. For every non-initial state s_g lying on the goal path, and every state s_p that is a sibling of s_g, a preference pair $\langle s_g, s_p\rangle$ is created. Each such pair produces a training instance, $\langle\langle s_g, s_p\rangle, +\rangle$, from which its dual, $\langle\langle s_p, s_g\rangle, -\rangle$, can be generated.

A preference predicate may then be induced from these examples using a concept-learning algorithm. If a state consists of a vector of feature values $\langle f_1, f_2, \cdots f_n\rangle$, the training instance $\langle\langle s_g, s_p\rangle, +\rangle$ may be converted into a feature vector

$$\langle f_{g_1}, f_{g_2}, \cdots f_{g_n}, \quad f_{p_1}, f_{p_2}, \cdots, f_{p_n}\rangle \tag{2.2}$$

which may then be given to a concept learning algorithm that induces concepts from examples. After a concept has been induced from a set of such examples, the performance system can use it by determining whether $\langle s_1, s_2\rangle$ satisfies the concept.

Utgoff and Clouse (1991b) have shown that if the concept learner is a linear threshold unit, feature vector 2.2 can be simplified. For a preference pair $\langle s_1, s_2\rangle$, we

know that $p(s_1, s_2)$ should be true. The relationship $p(s_1, s_2) \leftrightarrow f(s_1) > f(s_2)$ can be used to derive the following inequalities:

$$\begin{array}{rcl} & p(s_1, s_2) & \\ f(s_1) & > & f(s_2) \\ \mathbf{W}^T\mathbf{F}(s_1) & > & \mathbf{W}^T\mathbf{F}(s_2) \\ \mathbf{W}^T(\mathbf{F}(s_1) - \mathbf{F}(s_2)) & > & 0 \end{array} \tag{2.3}$$

The left-hand side of 2.3 can then be used to generate the positive example:

$$\langle (f_{g_1} - f_{p_1}), (f_{g_2} - f_{p_2}), \cdots, (f_{g_n} - f_{p_n}) \rangle \tag{2.4}$$

This delta vector may then be used to train a linear threshold unit (LTU). It has n feature values rather than the $2n$ of vector 2.2.

Preference predicates have been used to learn the game of OTHELLO [Utgoff & Heitman, 1988], the Towers of Hanoi problem [Utgoff & Clouse, 1991b], and backgammon [Tesauro, 1989a]. Preference predicates have also been used to play chess endgames [Huberman, 1968], although the predicates were handcrafted.

One of the issues in preference predicate learning is the quality of the solution paths from which preference pairs are generated. If the goal found was suboptimal, or the path to the goal was longer than it could be, then the resulting preference pairs extracted from the path will not produce good problem solving. One way of ensuring high-quality solutions is to learn from an expert. In a two-player game, this involves playing against an expert and extracting preference pairs from the expert's moves (assuming that the expert wins), or using book moves from experts' games. With single-agent search, the expert must provide the search path directly [Waterman, 1970; Silver, 1986]. Another way of ensuring high-quality solutions is to perform exhaustive or extensive search after a solution is found, to ensure that there is no solution of higher quality that was missed [Mitchell, Utgoff & Banerji, 1983].

2.5 Other Methods for Learning Search Control

This chapter has reviewed methods for learning search state values. The values are used by a general search method such as A* that orders and selects the states to be explored. Methods other than state evaluation functions have been explored in machine learning.

Instead of learning state values, some methods learn the conditions under which operators should be applied to states. The LEX system [Mitchell, Utgoff & Banerji, 1983] used a set of operators for solving integration problems. Each operator had one or more associated concepts, called *heuristics*, that represented the problem forms to

which its operator may be usefully applied. The learning task of LEX was to refine these heuristics as much as possible, in order to reduce search in applying operators to solve integration problems. Keller's (1987) MetaLEX system used an alternate method of controlling operator applications. MetaLEX manipulated a set of explicit expressions defining the usefulness of LEX's operators. Like LEX, Langley's (1983) SAGE system induced descriptions of contexts in which its operators were useful. By improving the descriptions, SAGE improved its search control.

Other systems learn general problem-solving rules for directing search. Some systems learn from failure and create rules that recognize states that result in assured failure [Minton, 1984; Hammond, 1986; Mostow & Bhatnagar, 1987]. Some systems learn from success and create rules, sometimes called macros, that encapsulate useful sequences of operators [Korf, 1982; Iba, 1989]. SOAR [Laird, Rosenbloom & Newell, 1986] learns from success and creates "chunks" from the contents of working memory after a successful problem solving episode. A chunk is a rule leading directly from a problem to its solution; by using chunks, a system can solve problems more quickly because it can avoid the intervening problem solving steps. PRODIGY [Minton, 1988] learns from a variety of problem solving events, such as goal interactions, failed paths, successful paths and sole alternative choices. From these events, PRODIGY can learn selection rules, rejection rules and preference rules to guide its state space search.

Because of their generality, rule-learning systems such as PRODIGY and SOAR have been popular in machine learning. However, one of the drawbacks of existing rule-learning systems is that they rely on deduction and require a *tractable* domain theory. An intractable domain theory is one that is complete but for which it is computationally prohibitive to construct explanations in terms of the theory [Mitchell, Keller & Kedar-Cabelli, 1986]. In addition, such rule-learning systems are not guaranteed to find optimal solutions.

Several methods exist that can produce control rules from intractable domain theories. Tadepalli's method (1989) produces overly general plans that ignore the possibility of complications; a plan is not refined until it fails and has to be specialized to account for the failure. Flann's method (1990) is similar, but produces expressions that are guaranteed to be correct as long as the assumptions under which they were learned hold during performance. Flann's method guarantees correctness by using a set of influence relations that enable it to compute relevant interactions among plan objects.

However, both Tadepalli's and Flann's methods generate rules only for small portions of the search space; neither has been applied the full space of an intractable domain. In addition, neither method is sensitive to the cost of the learned rules, so the methods may acquire new rules indefinitely, regardless of the expense. An evaluation function has the advantage of being applicable over the entire search space, and if the features' expense is bounded, the cost of applying the evaluation function is bounded as well.

There are distinct advantages to using evaluation functions, as opposed to rules or heuristics, for search control. However, evaluation functions are based upon features

that describe problem solving states; an evaluation function is very sensitive to the set of features chosen. The next chapter explains this sensitivity to representation and reviews methods that have been developed to overcome representational weakness.

CHAPTER 3

FEATURE GENERATION

Chapter 2 reviewed search control and showed how inductive learning algorithms can be used to create evaluation functions for best-first search. From a performance trace, examples can be generated that can be used to learn either an evaluation function or a preference predicate. Inductive learning from examples has received much attention in machine learning. Many different concept formalisms have been explored, such as decision trees [Quinlan, 1986], decision lists [Rivest, 1987], linear threshold units [Nilsson, 1965], connectionist networks [Rumelhart & McClelland, 1986], and structured concept descriptions [Lebowitz, 1986]. However, all of these methods are sensitive to the example representation.

3.1 The Sensitivity of Induction to Representation

Inductive learning methods require that the examples be expressed using a set of *features*. Each feature is a function $f : S \rightarrow V$ where S is the set of problem solving states and V is a set of values. V is usually the real numbers or integers. Predicates may be expressed as binary features where $V = \{0, 1\}$. For a discrete-valued attribute A with values $v_1, v_2, \cdots, v_n$, a set of binary features may be created, each of which corresponds to a range value of A. For each value v_i of the attribute A, a binary feature

$$F_i = \begin{cases} 1 & \text{if } A = v_i \\ 0 & \text{otherwise} \end{cases}$$

may be created.

Primitive features of a domain are low-level features that are sufficient for describing states unambiguously. For example, in chess the primitive features would specify the contents of every square on a chess board. It is usually easy to define a set of primitive features for a domain.

Regardless of the formalism, all inductive learning methods are sensitive to the set of features chosen to represent the examples. The set of features strongly influences

the form and size of the final concept, as well as the ability of the concept to generalize to unseen examples (its inherent accuracy). The feature set also influences the speed of convergence of the learning method. In some cases, as with linear threshold units, the learning method may never converge if the feature set is inappropriate.

Primitive features are often used in concept learning because they are easy to define, easy to compute, and because they require very little knowledge about the domain. However, primitive features often are not suitable for use in an evaluation function because they may not capture aspects of a state useful for determining its value. That is, they may not capture aspects that are predictive of the value of the solution eventually found from the state. Therefore, they are not useful for developing evaluation functions for guiding the problem solver. In the terminology of Flann and Dietterich (1986), the primitive features are *structural*, whereas problem solving often requires a *functional* representation that expresses functional differences between examples. For example, to a chess player the knowledge that the king is in check (a functional feature) may be significant, but the specific square that the king is on (a structural feature) may be unimportant. Furthermore, a specific functional feature may take many structural manifestations; for example, there are many chess board configurations in which a king is in check, but they may be structurally very dissimilar [Callan, 1993].

3.2 Manual Feature Generation

Unfortunately, designing good features that express relevant functional aspects of states is not easy. Usually it requires an expert in the domain, who must spend time designing and testing different features:

1. The human designs a set of features, handcodes them, then gives them to a learning system.

2. The learning system generates an evaluation function using the features and a set of examples.

3. A performance system solves problems using the evaluation function. The result is a set of performance traces.

4. The expert then scrutinizes the problem solving traces and locates mistakes in the evaluation function. Based on the mistakes, the expert guesses what new information the system needs in order to reduce the errors. The expert then creates new feature, or changes existing ones, to incorporate this information.

Whenever the feature set is changed it must be tested again. The process continues until the system's performance is deemed acceptable.

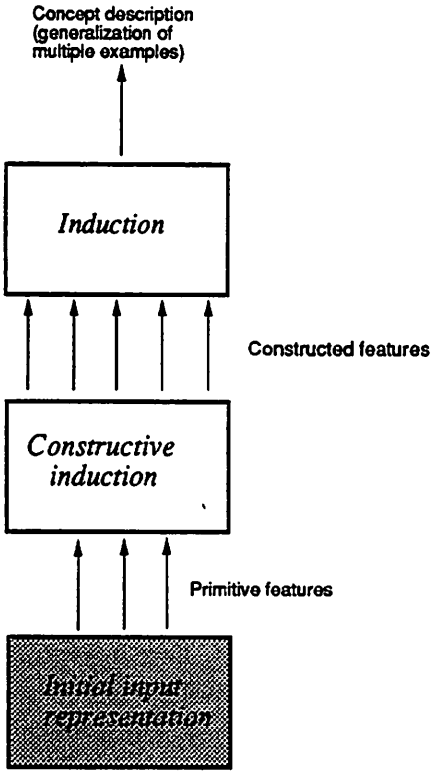


Figure 3.1 The role of feature creation in induction

3.3 Automatic Feature Generation and Induction

Because of the sensitivity of induction to representation, and the fact that structural representations are often inappropriate, effort in machine learning has been devoted to the problem of devising new features for induction. The general problem has been given many names. Because the problem of feature construction is seen as an adjunct to induction, it has been called *constructive induction* [Michalski, 1983], to contrast it with selective induction in which features are selected from a fixed set that is initially given to the concept learner. It is also called the *new term problem* [Dietterich, London, Clarkson & Dromey, 1982]. The set of representable concepts has been called a *bias* of the learning method, so the process of adding new terms has been called *shift of bias* [Utgoff, 1986b]. Because the set of features represents the examples, it is also called *change of representation*. In multi-layer connectionist networks, the internal (hidden-layer) nodes can adapt themselves to recognize recurring input patterns; this phenomenon has been called *learning internal representations* [Rumelhart & McClelland, 1986]. Such representations are also referred to as *distributed representations*.

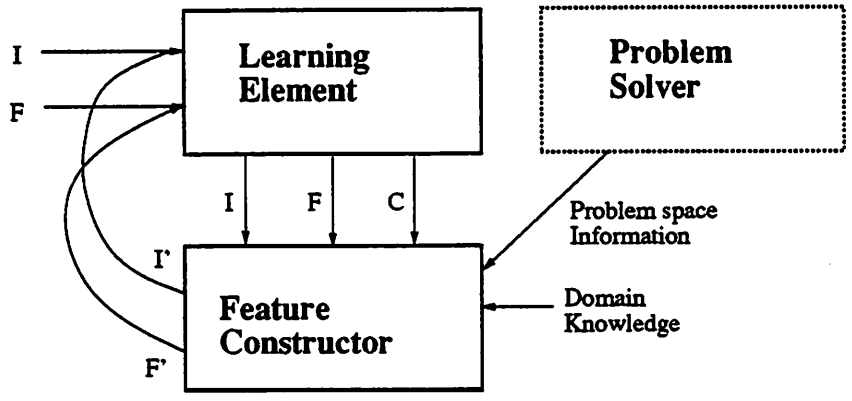


Figure 3.2 A schema for the feature construction process

Regardless of the names and details of the feature generating methods, there are certain aspects common to all of them. One way of viewing the role of feature generation in induction is in the layers illustrated in Figure 3.1.

- Examples are expressed using a vocabulary called the **primitives**, or instance-level features. The primitive feature set is immutable.
- The **constructed feature** layer augments the input representation by adding descriptors to the examples. For example, in chess a constructed feature for a fork might add a description like fork(fsquare,tsquare1,tsquare2), indicating that the piece at square fsquare is involved in a fork threatened by tsquare1 and tsquare2. The descriptions added by features may also be numeric; for example, mobility(white), whose value could be a non-negative integer. Constructive induction operates by adding features at this intermediate level. The constructed features may either replace the primitives or augment them.
- The **induction** layer performs induction over the augmented example descriptions and forms a final concept. That is, it performs a search for an acceptable concept description based on the set of example descriptions provided by the layers below it. The induction process can be as simple as a correction rule for a linear threshold unit [Nilsson, 1965] or as complex as an algorithm like AQ11 [Dietterich & Michalski, 1981] that performs induction over structured, symbolically represented examples.

Feature generation may be seen as a process that changes the features computed in the second layer. Generally, feature generation involves a separate process that is able to examine the results of concept learning and suggest new useful features that may improve it.

Figure 3.2, adapted from a figure used by Matheus (1989b), shows a schematic diagram of a feature construction module and its relationship to a concept learner.

The concept learner is initially provided with a set of instances I and features F, and produces a learned concept C as an evaluation function or preference predicate. This concept, along with the instances and features, is passed to the feature constructor. The feature constructor produces a set of new features F', and a set of new instances I' which are the instances I re-expressed in terms of the new features F'. The feature constructor may also make use of domain knowledge, such as information about definitions of rules or operators in the domain. If a problem solver is present, it may provide additional information that can be used to guide feature generation, such as solution sequences or configurations of nodes encountered in its search. However, it is not necessary for a constructive induction method to exploit all of the information shown in Figure 3.2. Most choose a subset based on their assumptions about the domain and the learning problem.

3.4 Automatic Methods for Feature Construction

This section reviews existing methods in machine learning for feature construction. Because this thesis involves constructive induction in the context of problem solving, and depends greatly on domain knowledge, the reviews will focus on these two aspects of each method. The systems will be ordered roughly by the amount of domain knowledge that they use, progressing from systems that use little or no domain knowledge to those that depend greatly on domain knowledge.

3.4.1 BACKPROP and Connectionist Networks

The simplest connectionist architecture is the one-layer network, called the linear threshold unit (LTU) [Nilsson, 1965]. An LTU maintains a set of weights, and given a set of numeric (possibly binary) feature values, produces a thresholded linear sum of the weights as output. Geometrically, the weights of the LTU represent a hyperplane. In general, n weights constitute an $(n-1)$-dimensional hyperplane that splits the n-dimensional feature space. Given an instance feature vector, if the output of the LTU is less then zero, the instance represented lies on one side of the hyperplane; if the output is greater than zero, it lies on the other side.

The LTU has the well-known restriction that it can only discriminate instances that are linearly separable. This restriction can be eliminated by using multi-layer connectionist networks, in which there is a *hidden layer* that can compute intermediate results that may then be used by the top-level (output) node. The generalized delta rule [Rumelhart & McClelland, 1986, Chapter 8], commonly called the BACKPROP algorithm, can learn weights for a network of LTUs by back-propagating error signals from the output node down into the hidden layer. Through training, the hidden-layer LTUs can learn to compute useful intermediate features of the input nodes for use in

the output LTU, and this is a form of constructive induction. Every distinct set of weight vectors for the hidden layer represents a separate representation of the input layer to the output layer.

There are several drawbacks to connectionist networks as a general method for constructive induction:

- **Domain Knowledge.** As with all inductive methods, connectionist networks can be applied to a set of examples without requiring knowledge of the domain from which they were derived. There are many ways in which domain knowledges can be incorporated into a connectionist network, one of which is to design carefully a set of features.

  The KBANN system [Towell, Shavlik & Noordewier, 1990] builds a connectionist network from a domain theory, but its method is limited to domain theories expressed in propositional logic. KBANN is inappropriate for problem solving systems because neither recursive rules nor state-transition operators are allowed.

  Recently, Tesauro (1992a,1992b) showed that a connectionist network using only structural (primitive) features could learn to play backgammon at computer championship levels, using no background knowledge. Trained with a temporal difference method, Tesauro observed that the network was able to create useful features in its hidden layer, some of which had intuitive meaning to backgammon players. Tesauro suggests that his system, and temporal difference learning in general, is a knowledge-free technique for learning useful features for problem solving.

  However, it is significant that Tesauro's system demonstrated a substantial increase in performance when given expert handcrafted features. Therefore, there is a definite advantage to providing the system with an input representation that incorporates expert knowledge.

- **Training Time.** Empirically, BACKPROP requires significantly more presentations than symbolic algorithms. Two recent empirical studies, [Mooney, Shavlik, Towell & Gove, 1989] and [Fisher & McKusick, 1989], observed that BACKPROP takes substantially longer than ID3 on the same task. The former study concluded that BACKPROP needed one to two orders of magnitude more presentations than ID3 to achieve the same level of accuracy, although BACKPROP was slightly more noise-tolerant.

  Measuring training time by the number of presentations may be misleading. Connectionist networks can typically process examples faster than symbolic algorithms (e.g., ID3). However, they are usually not an order of magnitude faster.

- **Network Design.** The design of connectionist networks has not been completely automated because several aspects of network architecture are not

automatically determinable. The initial settings of a network's weights have a strong influence on both the learning speed and the generalizing ability of the resulting network [Towell, Shavlik & Noordewier, 1990]. Also, the number of intermediate nodes needed for a network to solve a given problem varies a great deal, and is generally not determinable [Fisher & McKusick, 1989]. If too few are used, the network will attain suboptimal accuracy on the examples; if too many are used, the network will "memorize" the inputs and not form useful features, and thus not generalize well to unseen inputs. Current research in connectionist networks includes the development of network architectures that can add intermediate nodes automatically when necessary [Ash, 1989; Fahlman & Lebiere, 1990].

3.4.2 STAGGER

STAGGER [Schlimmer & Granger, 1986] is a concept classification system that uses feedback from its classifications to suggest new features. STAGGER integrates weight learning with boolean function learning in a principled manner. Subsequent work [Schlimmer, 1987] added numeric attribute value partitioning (*i.e.*, learning ranges).

In STAGGER, a concept description is a set of numerically weighted features called *elements*. Each element is a boolean function of attribute values, *e.g.* (SIZE=MEDIUM) AND (COLOR=RED). Each of these elements has two weights associated with it: a measure of its logical necessity (LN) and logical sufficiency (LS). Whenever a new instance is presented to STAGGER, each element is matched against the instance and its weights are used to make a prediction about whether the instance is a positive or negative example of the concept. After the prediction is made, the LS and LN of each element are updated based on the true and predicted classifications.

Feature learning in STAGGER is triggered by concept prediction failure, which occurs when there is no concept description consistent with positive and negative examples. New features are formed by applying the boolean operators AND, OR and NOT to existing features according to a set of heuristics. For example, if STAGGER predicted a negative example to be positive (an error of commission), it might conjoin two features having large LN values to form a new, more specific feature. This new feature would then compete with the others, and if its predictive accuracy fell below that of its component features, the new feature would be removed.

The concept learning of STAGGER allows it to learn linear combinations of features, and the feature learning in turn employs feedback from the weights to direct it. Because STAGGER only generates new features upon prediction failure, it may be seen as having a strong bias toward linearly separable concepts. Its feature generation component serves to shift this bias, using boolean operators, when it is inadequate. STAGGER's method of combining features is a similar to the "subdemon conjugation" method proposed by Selfridge (1959) for the Pandemonium system.

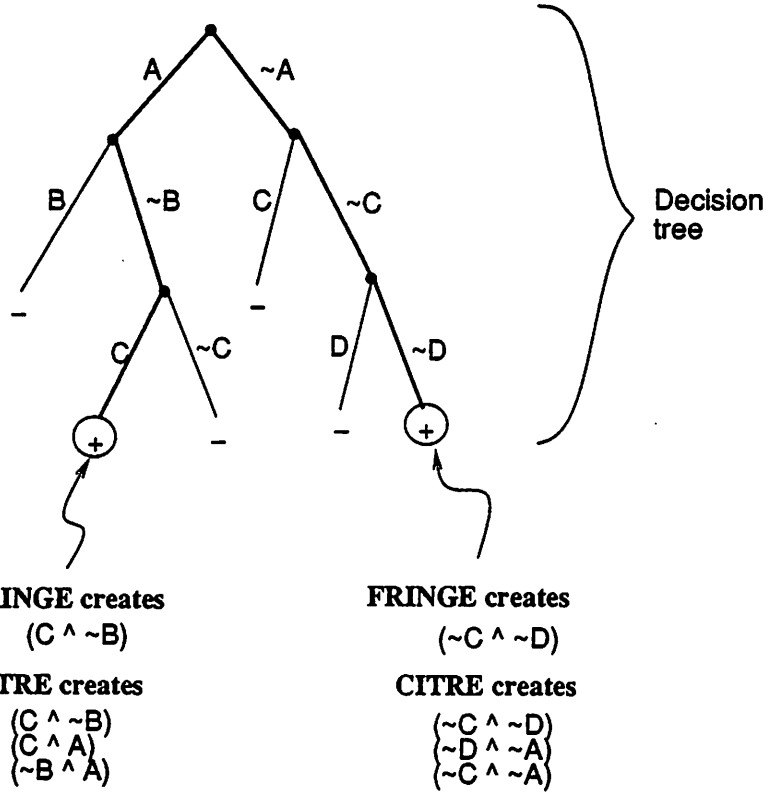


Figure 3.3 Feature creation by CITRE and FRINGE. Circled nodes are positive leaf nodes from which CITRE and FRINGE generate features. Neither filtering nor generalization done by CITRE is shown.

3.4.3 FRINGE and CITRE

FRINGE [Pagallo, 1989; Pagallo & Haussler, 1990] works in conjunction with a decision tree learning system to generate new features. It begins by generating a decision tree (a concept) using its set of features. FRINGE then examines the tree and forms conjunctions of two features that occur at the positive leaf nodes of the decision tree. That is, given a positive leaf node, FRINGE creates a new feature by conjoining the features that are immediately above the leaf on the path. The newly-created features are added back into the set of features available, and the process continues.

FRINGE was applied to five different concept domains, including small random DNF functions, a multiplexor, and 4- and 5-bit parity. FRINGE was able consistently to outperform a decision tree algorithm that used only the primitive features. Both classification accuracy and average tree size were better with FRINGE.

Like FRINGE, CITRE [Matheus & Rendell, 1989a] creates new terms for (and from) decision trees. CITRE's criterion for generating new features is slightly different: it creates features when there is more than one positive-valued leaf in a decision tree (and thus, when the concept is disjunctive). CITRE examines each

path from the root node to a positive-valued leaf node, and creates a feature for every pair of tests in the path. Figure 3.3 illustrates a decision tree and the features that would be created from it by CITRE and FRINGE. Like FRINGE, CITRE adds its constructed features back into its current set before creating a new decision tree.

CITRE is able to use domain knowledge to augment its feature generation process [Matheus, 1990]. CITRE employs two filtering heuristics and one generalizing heuristic, both of which are domain-specific. In the domain of tic-tac-toe, one filter removes features that refer to different piece types, the other removes features involving non-adjacent squares. The generalization heuristic performed spatial translations of patterns. CITRE's feature filters and generalizer are examples of procedural domain knowledge: a user of CITRE must provide a set of procedures that can filter and generalize the features that CITRE will produce.

3.4.4 LIVE

LIVE [Shen & Simon, 1989] is a problem solving system that is able to refine its operators. LIVE is given a set of *actions* (operators) that it can execute, a set of *features* that it can perceive in the environment, a set of *constructors* for creating new features, and a *goal* to achieve in the environment. By performing experiments in the environment, LIVE is able to create rules (STRIPS-style operators with pre- and post-conditions) describing the conditions under which an action will have a certain effect.

LIVE performs difference-based specialization of its operators. It begins with overly general rules that describe the effects of its actions. A violation occurs when an operator applied to a state produces a new state that does not conform to the rules' prediction. After such a violation, new specialized rules are created to account for the violating event. LIVE specializes the rules by discriminating between a state in which its rule predicted correctly and the state that violated the rule.

If a rule precondition is viewed as a concept description, then LIVE's feature generation is triggered by prediction failure of that concept. LIVE learns new features when its existing features are unable to discriminate between these "correct" and "surprising" states. It creates features using its given set of constructors. For example, LIVE used bitwise-OR and bitwise-AND in order to create ANDs and ORs of bit vectors used as features. LIVE utilizes the specific states that could not be discriminated. In this regard, LIVE is similar to STABB (Section 3.4.8), though STABB uses a domain theory to create features and LIVE does not. LIVE does not use its domain theory because by definition its theory is incomplete when discrimination fails.

LIVE instead depends upon a set of constructor functions, which are assumed to be appropriate for the domain. This is a strong assumption — the choice of constructor functions represents a second-order bias that may not be adequate for the features that are needed. In addition, the constructors may produce a very large feature space, depending on their number and the order in which they are searched.

There is also the problem of finding the *right* feature: LIVE stops as soon as it finds some feature that can account for a difference between two states, but there may be many such features, and LIVE may not be able to recover if the wrong choice is made. Other feature generation systems are less sensitive to this problem because they only demand that the features be predictive of the concept, not that they uniquely determine it.

LIVE's use of constructor functions for creating features is a form of procedural domain knowledge, like CITRE's feature filters and generalizers. Unlike CITRE, LIVE depends upon these functions to enable it to create features at all.

3.4.5 MIRO

MIRO [Drastal, Czako & Raatz, 1989] is related to both constructive induction and explanation-based learning, in that it generates new terms for induction from concepts (descriptors) implied by deductive rules from a domain theory. MIRO uses the descriptors to generate higher-order terms that can then be used by induction.

MIRO constructs an *abstract concept description language*, L_A, based on its domain theory. Informally, if a descriptor x (the conclusion on the right-hand side of a rule) can be derived from some example e, but no other descriptors may be derived using x, then x is said to be maximally proven. MIRO assumes that the domain theory's rules deduce abstract descriptors from less abstract descriptors, thus x can be thought of as a maximally abstract descriptor of e. The set of all such maximally proven descriptors, computed from each positive and negative example, constitutes the abstract concept description language L_A.

MIRO uses a greedy algorithm to induce a concept description from this language. Similar to the AQ11 algorithm [Dietterich & Michalski, 1981], MIRO repeatedly selects a seed from the set of positive examples and creates a partial concept description that covers the seed and none of the negative examples. The partial concept description is created using a one-sided variant of the candidate-elimination algorithm [Mitchell, 1978]. The one-sided candidate-elimination method computes a G set, in the abstract language L_A, that is consistent with the seed and the set of all negative examples. MIRO heuristically selects an element of this G set as a "cover" of the seed, which is a partial concept description of the target concept. MIRO then chooses another seed, and continues until all positive examples are covered. Finally the partial concept descriptions are disjoined into a complete concept description, which MIRO returns.

It may happen that MIRO cannot produce a consistent concept description because the abstract language L_A is too strong: it is insufficient to discriminate the positive and negative examples. In this case, MIRO is able to decrease the inductive bias incrementally by adding non-maximally proved descriptors into L_A. MIRO continues to extend L_A as long as L_A is insufficient to discriminate the examples. Eventually, MIRO will add the instance language descriptors into L_A; if these are still insufficient, MIRO will declare failure.

MIRO is one of the few models of constructive induction that incorporates both deduction and induction, and uses a declarative domain theory. However, MIRO is limited in the kind of domain theory it can use. The theory must be expressed as a set of propositional rules that are representable as a finite acyclic AND/OR graph. Therefore, recursive rules are not allowable, and MIRO cannot use state-transition operators.

3.4.6 DRL

DRL [Ragavan & Rendell, 1991] is a system that generates relational features for an inductive concept learning system. A relational feature is a set of objects, each described by an n-tuple of attribute values. Relational features are a form of domain knowledge provided by the user to facilitate learning. Each relation is assumed to be relevant to, but more specific than, the concept to be learned.[1]

DRL uses a set covering algorithm to create a concept description. The set covering algorithm calculates the *relevance* of a relation, which is the number of positive examples covered by the relation divided by the total number of positive examples. In every iteration, DRL ranks the relations by their relevance, then chooses the highest ranking relation to add to the concept description. The set of examples that it covers are removed, and the process continues until no positive examples remain.

DRL creates new relations by unioning and intersecting existing relations. These operations can generalize and specialize, respectively. For example, let

$$\begin{aligned} R_1 &= \{\langle 1,0\rangle, \langle 1,1\rangle\} \\ R_2 &= \{\langle 0,1\rangle, \langle 1,1\rangle\} \end{aligned}$$

From these sets, one can create:

$$R_1 \cap R_2 = \{\langle 1,1\rangle\}$$

which is a specialization, and:

$$R_1 \cup R_2 = \{\langle 1,0\rangle, \langle 1,1\rangle, \langle 0,1\rangle\}$$

which is a generalization.

DRL maintains a list of relations, sorted by relevance, and DRL creates new relations by applying the union and intersection operations to the first and second most relevant relations in its list. After DRL creates new relations, it adds them back into its list of relations, sorted by relevance, for use in covering the example. DRL also applies knowledge of symmetry, if it exists, to the relations it creates.

[1] The authors use the terms "relation", "knowledge relation" and "relational feature" interchangeably. I will use the term "relation" for all of them.

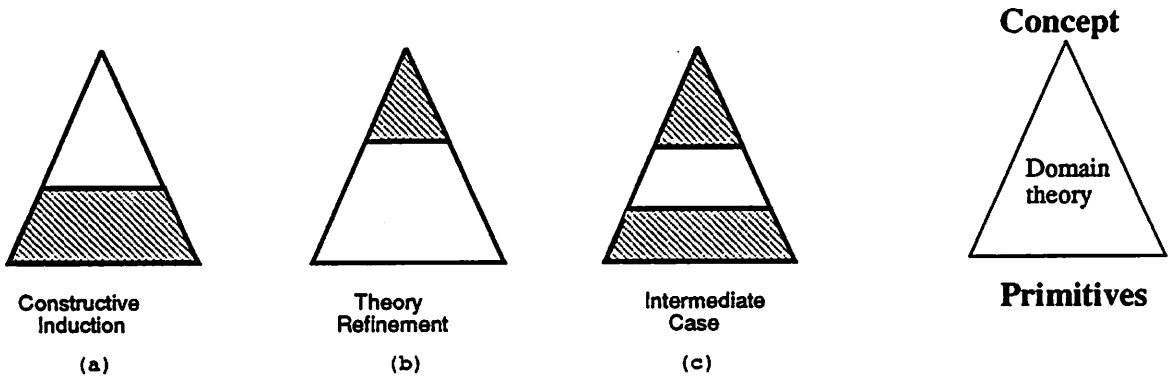


Figure 3.4 The types of gaps in a domain theory. The shaded portion represents the area covered by the domain theory, the unshaded portion must be filled in by induction.

DRL may be compared to PLS0 [Rendell, 1985], in that it uses a measure of feature utility (relation relevance) to select features to be combined, though neither uses this measure to suggest combination operators. Unlike PLS0, DRL uses domain knowledge in the form of explicitly provided relations. Because DRL uses relations, it is not limited to attribute-value pair features.

3.4.7 EITHER

EITHER [Mooney & Ourston, 1991] combines constructive induction and theory refinement. Theory refinement is usually considered a branch of explanation-based learning that deals with the problem of correcting an erroneous domain theory or extending an incomplete one. The goal of theory refinement is somewhat different from that of constructive induction. Constructive induction tries to alter the set of features (the domain theory) solely to improve the final concept induced, while theory refinement tries to improve the domain theory in order to improve its explanatory or predictive power.

Figure 3.4, used by Mooney and Ourston (1991), illustrates the relationship between theory refinement and some work in constructive induction. Most constructive induction methods build features in a bottom-up fashion, starting from the primitives (Figure 3.4a) so that the resulting features enable an accurate concept description to be induced. EITHER is also able to handle the cases shown in Figure 3.4b and c, where the concept definition is specified, and there is a gap occurring at the bottom or in the "middle" of the theory. Note that EITHER, and theory refinement systems in general, does not distinguish between concept learning and feature generation; the domain theory includes rules for intermediate features as well as the final goal concept.

EITHER's domain theory must be a set of rules expressed in predicate logic. Each rule is of the form:

$$\text{Consequent} \leftarrow \text{Antecedent}_1 \wedge \text{Antecedent}_2 \ldots$$

EITHER is given a set of positive and negative examples. EITHER then uses its domain theory to classify the examples. Classification is performed by attempting to prove that the example satisfies the concept definition. If an example is classified correctly, it is ignored; otherwise it is considered either a *failing negative* (a negative example classified as positive) or a *failing positive* (a positive example classified as negative). A failing positive is caused by an overly specific theory, and a failing negative by an overly general theory. The theory can be both overly general and overly specific at the same time.

The theory is generalized by calculating a near-minimum set of antecedent retractions necessary that correct all the failing positives. The theory is specialized by calculating a near-minimum set of leaf-level rule retractions. In both cases, if the retractions cause new failures, EITHER uses induction (currently a version of ID3 [Quinlan, 1986]) to create a new version of the rule that correctly discriminates between positive and negative examples for that rule.

EITHER's induction of these rules utilizes intermediate concepts, generated by forward-chaining from the facts of an example. This use of intermediate concepts is similar to MIRO's generation of descriptors. In this way, if an intermediate concept is highly correlated with the failing rule class, then it is used in the newly created rule.

EITHER can also create new intermediate concepts by using techniques from inverse resolution [Muggleton, 1987]. For example, one technique is called inter-construction, in which rules such as

$$\begin{aligned} x &\leftarrow w \wedge y \wedge z \\ x &\leftarrow u \wedge y \wedge z \end{aligned}$$

are combined to form the new set of rules:

$$\begin{aligned} x &\leftarrow w \wedge v \\ x &\leftarrow u \wedge v \\ v &\leftarrow y \wedge z \end{aligned}$$

This compresses the rules for x by factoring out a common suffix and creating a rule (a feature) for v, the common suffix. By introducing a distinct rule for v, its definition becomes available for the creation of new rules by induction.

3.4.8 STABB

STABB [Utgoff, 1986a; Utgoff, 1986b] is the constructive induction component of LEX, a system that acquires problem-solving heuristics in the domain of symbolic

integration. LEX contains a problem solver, a critic to extract training instances from a problem solution trace, and a generalizer that takes instances from the critic and produces generalized heuristics. LEX has a set of operators that perform algebraic manipulations and integration operations. Each operator has one or more associated concepts, each a *heuristic*, that represent the problem states to which its operator may be applied usefully. The learning goal of LEX is to refine these heuristics as much as possible, in order to reduce search in applying operators to solve integration problems.

LEX represents incompletely learned heuristics using Mitchell's (1977) version space method, by which the most general and most specific boundaries (versions) of a concept are explicitly stored. These boundaries may be adjusted by the generalizer after every training instance involving the heuristic. Version spaces depend on a generalization hierarchy appropriate for the learning task; LEX's hierarchy contains mathematical terms such as *transcendental function*, *integer*, $+$, and *monomial*.

STABB generates new features for LEX's hierarchy, and is invoked when the version space of an operator collapses; that is, when there are no terms left in the region between its general and specific boundaries. STABB is given the sequence of operators that solved the integration problem, and the operator in the sequence whose version space is now empty as a result of generalization. STABB has two methods for creating new terms: constraint back-propagation and least disjunction, of which only the former will be described.

Constraint back-propagation (CBP) generates new terms by propagating the set of all solved states back through an operator sequence that yielded the original solved trace. Given an operator sequence $OP_1, OP_2, \cdots, OP_n$, and a solution state S, CBP computes the operator pre-images $OP_1^{-1}, OP_2^{-1}, \cdots, OP_n^{-1}$ to S in reverse order. The expression S, backed up through an operator pre-image OP_k^{-1}, yields a sufficient condition (call it S_k) for which OP_k will yield a solution. Therefore, if OP_k is the operator whose version space is now empty, CBP generates S_k, adds it as a new term to LEX's hierarchy, and uses it to regenerate the version space for OP_k.

In summary, STABB uses domain knowledge in the form of operator pre-images, and information from the problem solver in the form of solution sequences. Like LIVE, STABB is invoked upon concept failure (version space collapse), and the solution sequence used is the one that caused the failure. STABB creates a new feature that is guaranteed to be useful for the concept that failed.

3.4.9 CINDI

CINDI [Callan & Utgoff, 1991a; Callan & Utgoff, 1991b] was designed to create features for state-space search problem solvers. Given a complete problem specification, CINDI produces a set of features useful for evaluating search states. Unlike most of the systems reviewed in this chapter, CINDI is not iterative: it creates a set of features but does not develop them further. CINDI uses no information from

the instances or the concept; its feature creation is based entirely on problem solving knowledge.

CINDI uses a set of four transformations that break apart a problem specification in various ways:

1. The **LE** transformation applies to logical expressions in which a boolean predicate (*e.g.*$\wedge, \vee, \neg$) has the highest precedence. The LE transformation is applied first to the goal specification of the problem, and then recursively to each of the resulting subexpressions. LE creates a binary feature from each subexpression.

2. The **AE** transformation applies to expressions that use arithmetic operators (*e.g.* $=$ and $<$). From each of the operands, AE creates a feature that calculates the value of the operand in a state.

3. The **UQ** transformation applies to universally quantified expressions. From the expression, UQ creates a numeric function that calculates the percentage of permutations of variable bindings that are satisfied in that state. That is, the feature calculates the *degree of satisfaction* of the expression. Callan (1993) later generalized the UQ transformation to include existentially quantified expressions.

This set is sufficient to create features from any statement in first-order predicate calculus.

The justification for these transformations is that the features generated from the subexpressions usually provide information about how close a state is to satisfying the the original expression. For example, suppose that the goal statement of the problem solving system contains the expression $a < b$. Assuming a mapping of Boolean values into binary digits, this expression produces 1 and 0. As a feature, the expression $a < b$ tells the concept learner only whether the inequality has been satisfied or not. From this expression, the AE transformation produces features measuring a and b separately. These individual values may be more useful because they provide information on how close the expression $a < b$ is to being satisfied, rather than simply whether it is satisfied. If an operator can change a or b predictably, then it will probably be useful to measure the values of a and b in a state.

CINDI's transformations embody several related assumptions about state-space search:

- Complex expressions are comprised of simple expressions, combined in recognizable ways.

- Simple expressions have values that vary non-erratically. That is, operators do not usually change radically the values of simple expressions.

- Simple expressions can provide a more useful gradient than the complex expressions of which they are a part. The resulting gradient is more useful for

an evaluation function because best-first search uses local information to direct search toward the goal.

CINDI's transformations recognize these combinations and decompose them into their simpler constituents. These constituents then become features that are often useful for predicting the quality of a state in state-space search.

3.5 A Brief Summary

The systems reviewed in this chapter all perform some kind of feature generation for the purpose of induction, but there is considerable variation among the assumptions made and knowledge available. Most systems use domain knowledge in some form, but in some cases it is a domain-dependent procedure (for example, CITRE's generalizers or LIVE's feature constructors) provided by the user.

Of the systems, only LIVE, STABB and CINDI are designed for use in problem solving systems, and each exploits knowledge of state-space search in some way. However, each has a shortcoming. LIVE depends upon a set of constructor functions that are appropriate for its domain. Both LIVE and STABB are restricted to tractable search domains. CINDI is applicable to any problem solving domain with a domain theory, but CINDI only generates an initial feature set. It does not develop the feature set further; it assumes that the features are either sufficient as produced, or that they will be developed by an external constructive induction system.

In the next chapter we present a theory of feature generation that is not limited to tractable domains, as LIVE and STABB are. The theory is based upon iterative evaluation and development of features, so unlike CINDI the development is sensitive to feature worth. The feature generation is automatic and requires no human intervention.

CHAPTER 4

A THEORY OF CONSTRUCTIVE INDUCTION

This thesis proposes a transformational theory of feature generation for evaluation functions. The theory is specific to problem solving systems because the justifications for its transformations are based on assumptions about state space search, and because the theory depends upon a problem specification. The thesis is that:

> *Useful features for an evaluation function can be created by directed search through a feature space defined by four classes of transformations: goal regression, abstraction, decomposition and specialization.*

The theory comprises a set of transformation classes and a strategy for controlling the application of transformations to features. The transformations and control strategy are domain independent. Some transformations utilize information from the problem specification in creating new features, but the actions of the transformations are not tailored to an individual domain.

Feature generation begins with a feature created from the performance goal, and progressively develops better features by transforming existing ones. Thus, the performance goal and transformations define a *feature space* through which Zenith searches. The development of this space may be characterized as a modified beam search [Newell, 1978], based on features' usefulness to the concept learner. The transformations generating the space utilize both the problem specification and feedback from the concept learner.

4.1 General assumptions

There are several assumptions upon which the theory is based, some of which were implicit in previous chapters. These general assumptions both constrain the problem and specify the information available to be exploited.

- **A1:** The purpose of concept learning is to improve some problem-solving process.

- **A2:** The purpose of the problem-solving process is to find a solution to the goal; or if the solution quality can be measured, the highest quality solution possible.

Two assumptions are made about how the problem solver searches for solutions, and how the concept learner improves problem solving:

- **A3:** The problem solver pursues a goal by performing a directed search within a state-space. The problem solver generates a set of successor states and evaluates the set, choosing the best.

- **A4:** The problem solver uses the learned concept to determine the best of a set of states. No assumptions are made about the method that it uses for choosing the best. The concept learner may evaluate each state independently, or evaluate subsets of states.

Two assumptions are made about feature discovery:

- **A5:** The purpose of feature discovery is to aid the concept learner in evaluating sets of states; specifically, in distinguishing states that are closer to the goal from those that are further away.

- **A6:** The feature discovery component has access to a specification of the problem, also called a *domain theory*. The problem specification is declarative, and includes:

  - The goal of the performance element.
  - The definitions of operators in the domain, including code for computing both pre- and post-images of conditions with respect to operators.
  - Definitions of ancillary functions and predicates used in the goals and operators. The ancillary functions and predicates are defined down to the operational (instance-level) predicates.
  - Meta-information about the functions and predicates; specifically, which of them are *operational* and which are *state-dependent.*

    An operational predicate is one that matches surface (input representation) features. In a board game, the predicates $owns(Player, Square)$, $neighbor(Square1, Square2)$ and $blank(Square)$ would be operational; whereas $won(Player)$ and $legal\_move(Player, Square)$ would not be operational, because non-trivial computation may be required to verify them.

    A state-dependent predicate is one whose truth value can be directly or indirectly changed by an operator, and thus depends on the state. For example, $owns(Player, Square)$ and $won(Player)$ are state-dependent, but $neighbor(Square1, Square2)$ is not.

Three assumptions are made about features:

- **A7:** A feature is based on a *formula*, which is a conjunction or a disjunction of first-order terms. A feature's value with respect to a state is the number of solutions of the formula in the state.
- **A8:** Every feature has a computational time cost associated with it, which is a measure of the amount of time required to evaluate the feature in a state.
- **A9:** The evaluation function's cost is bounded in some way so that it can use only a subset of the total features generated.

Additional details about features are given in Section 5.3.

4.2 Transformation Classes

The following four sections discuss the transformation classes in general terms, explaining what each accomplishes and giving examples of its use in the derivation of known features. Specific transformations used in Zenith are described for concreteness.

4.2.1 Decomposition

Decomposition transformations are syntactic methods for decomposing a feature functionally. Each decomposition transformation recognizes a specific syntactic form, such as an arithmetic inequality, that can be decomposed into new features. The transformation then creates one or more new features from fragments of the original feature.

The justification for decomposition is that the resulting fragments may provide more useful information to the evaluation function than the original feature. The fragments may be useful because they provide information on local state variations. Best-first search depends upon local information to direct the search toward goal states. Local information about how close an expression is to being satisfied can help the evaluation function because such information allows the function to make distinctions between neighboring states in the state space.

For example, one decomposition transformation looks for arithmetic inequalities such as $A > B$ and produces two new features that calculate A and B separately. These separate values may give the system information about how close the problem solver is to satisfying the original inequality.

Two examples of decomposition are shown in Figure 4.1. In the first example, the expression $A > B$, at top left, is decomposed into the expressions A and B, shown below it. The graphs show sample values of the expressions for states along a path

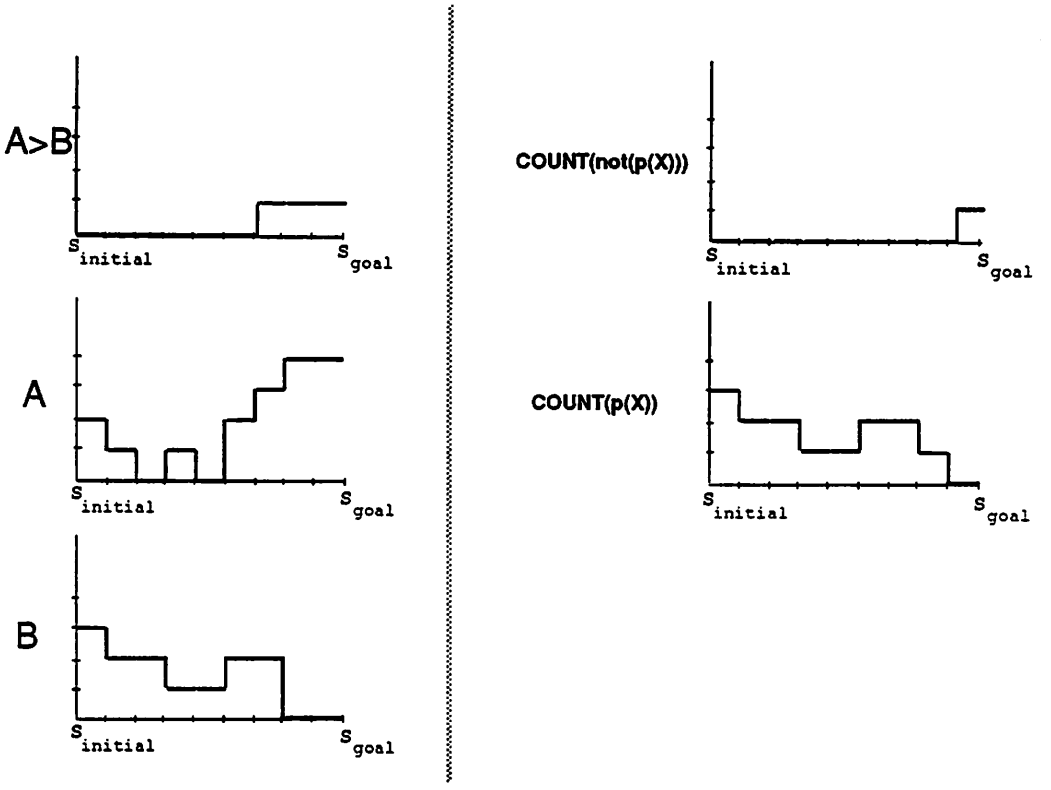


Figure 4.1 Two examples of decompositions. The left three graphs show the expression $A > B$ and its decompositions A and B. The right two graphs show counts of solutions for $not(p(X))$ and its decomposition $p(X)$. In these graphs, TRUE is mapped to one and FALSE is mapped to zero.

---

from $S_{initial}$ to S_{goal}. The expression $A > B$ often may not change from one state to the next, and so its values may not be useful to the evaluation function because they do not allow it to discriminate alternative state choices. The two expressions A and B may vary more often, and providing their values to an evaluation function may allow the function to discriminate states for which the $A > B$ relation is constant. If the evaluation function is a weighted sum of feature values, assigning a positive weight to A and a negative weight to B can direct the problem solver to states with higher A values and/or lower B values.

The example on the right-hand side of Figure 4.1 shows the effects of another decomposition transformation, which removes negations. The figure shows sample values for the expression $\neg p(X)$ and its decomposed expression $p(X)$. Because the domain of X is not known, it is not possible to enumerate the set of X's for which $\neg p(X)$ is true; it is only possible to determine whether $\neg p(X)$ is satisfied. Therefore, the graph of $\neg p(X)$ exhibits a "plateau" similar to that for $A > B$. However, the number of values satisfying $p(X)$ can be counted, and its values vary more within the

state sequence. Therefore the expression $p(X)$ may be more useful to the evaluation function in differentiating alternative states.

As an example of decomposition in OTHELLO, the rules state that a player has won when no moves can be made, and the player has more discs than the opponent. Decomposing the latter condition yields two features: one counting the number of discs of the player, and the other counting the number of discs of the opponent. Measuring these two values independently provides more information to the evaluation function than does the original inequality.

Callan (1993) has investigated a set of transformations for generating an initial set of features for a problem solving system. His transformations produce features by splitting apart the problem specification on the basis of syntactic forms. The justification for Callan's transformations is the same as for transformations of this decomposition class: the transformations produce features that may provide a more useful state-space gradient than would the original feature, and thus may be more effective for guiding the problem solver.

4.2.2 Goal Regression

Goal regression creates the pre-image of a formula with respect to an operator. If a feature is useful for the evaluation function, then it may be beneficial for the evaluation function also to use features that measure how the original feature is affected by the domain operators. For example, in many board games if it is useful to measure the number of pieces owned by a player, it is probably also useful to measure the number of pieces that could be acquired (or lost) by a move. The latter feature is created by regressing the piece ownership term through the move operator.

Goal regression is a common technique in artificial intelligence [Waldinger, 1976; Mitchell, Keller & Kedar-Cabelli, 1986; Utgoff, 1986b]. However, there are two significant differences to the goal regression in this theory:

1. In other systems, usually only the performance goal is regressed. In this theory, the formula of *any feature* can be regressed through an operator. Without this ability, goal regression would be capable only of producing features that were pre-images of the performance goal.

2. Usually goals are regressed along an entire solution path. In this theory, only a *single* goal regression step is done at a time. This difference allows the resulting feature(s) to be tested for usefulness before goal regression is applied again. It is assumed that the nth pre-image is at least as useful as the $n+1$st pre-image.

Goal regression may also be seen as computing enablement and disablement conditions. These conditions are common in the known expert features of board games. In OTHELLO, ownership of corner squares is important and most OTHELLO programs include features that measure corner square ownership. Because of the geometry of the board and the definition of the OTHELLO move, the corner squares

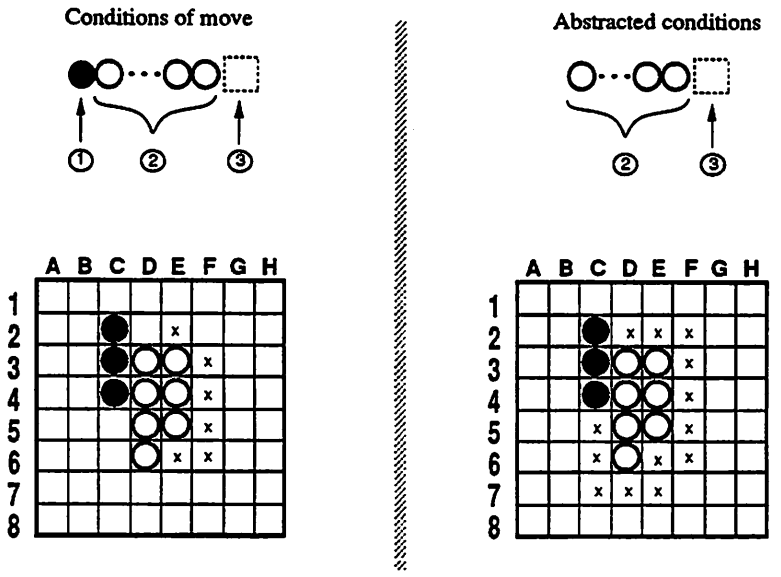


Figure 4.2 An example of abstraction in OTHELLO.

can only be captured from certain configurations involving certain squares, known as the C and X squares (see Figure 6.2). That is, C and X squares *enable* the capture of corner squares. Most OTHELLO programs employ features that check for ownership of corner squares, as well as other features that check for ownership of C and X squares.

In checkers, kings are very desirable to own because of their mobility; they can move both backwards and forwards. A player creates a king by moving a man to the opponent's back row. Both players try to keep their back rows protected to block the opponent's men — that is, to *disable* the creation of opponent kings. Most checkers programs have features that count the number of kings, as well as other features that determine whether the back row is protected.

4.2.3 Abstraction

Features are often created (for example, by goal regression) that could be valuable to the evaluation function but are not worth their cost. Both abstraction and specialization can be used to reduce cost. Abstraction is a form of generalization that removes details from a feature.

One way to determine whether a term is an unnecessary detail is to estimate how difficult it is for the problem solver to achieve it [Sacerdoti, 1974]. If a condition can be achieved easily, it can be considered a detail; if it can only be achieved with much effort (or not at all), it is likely to be an important condition that should not be removed.

An example of abstraction in OTHELLO is shown in Figure 4.2. The definition of a legal move for the black player is a blank square next to a line (in any direction)

of one or more white pieces, terminated by a black piece. A graphical depiction of these three conditions is shown at the top left of the figure. Of the three conditions, the easiest for the black player to influence is the existence of a black piece (condition 1), because the black player can place black pieces directly onto the board, but can neither place white pieces nor create blank squares. Therefore, a more abstract feature can be produced by removing condition (1) of the OTHELLO move. The resulting expression, shown at the top right of Figure 4.2, matches a pattern consisting of one or more white pieces terminated by a blank square. These conditions are the basis for several mobility features used by Rosenbloom (1982). These features are cheaper to evaluate than the original conditions of the OTHELLO move and preserve much of their accuracy. Figure 4.2 also shows, below each pattern, the matches of the pattern on a sample OTHELLO board. On each board a small "x" marks the matches of the blank square (condition 3) on the board.

A second example of abstraction occurs in the specially designated X and C squares of OTHELLO. As mentioned in Section 4.2.2, the X and C squares enable the taking of corner squares. However, owning an X or C square does not alone guarantee that the corner square can be captured; it is also necessary to place pieces on the other side of the X or C square to enable a move into the corner. But these other pieces are usually easy to place, and so they may be considered removable details of the enabling conditions. Thus, features that check X and C squares are *abstractions* of the actual enabling conditions of corner square captures.

4.2.4 Specialization

The fourth class of transformations creates specializations of existing features. Both specialization and abstraction reduce cost, but abstraction increases the domain of (generalizes) a feature, whereas specialization decreases it. Figure 4.3 illustrates this difference.

A common way of specializing a formula is to remove a disjunct from it, if it contains a disjunction. Another way is to replace a call to a recursive predicate with its base case from the domain theory.

A third way of specializing a feature is to restrict it to *invariants* [Puget, 1988]. An invariant may be defined as a domain element, or configuration of domain elements, that always satisfies a condition. The invariants of a feature may be found by searching for domain elements that satisfy a feature in every problem solving state. For example, a feature may be expensive because it tests members of a set to determine which of them satisfy an expensive predicate. If a subset of the domain elements can be found that always satisfies the predicate, a feature can be created that uses only elements from this subset and avoids calling the expensive predicate. The resulting feature will usually be much cheaper than the original and will maintain most of its accuracy.

In OTHELLO, "semi-stability" is an important property of a piece: it determines whether the piece can be captured immediately. Checking all squares for semi-stability

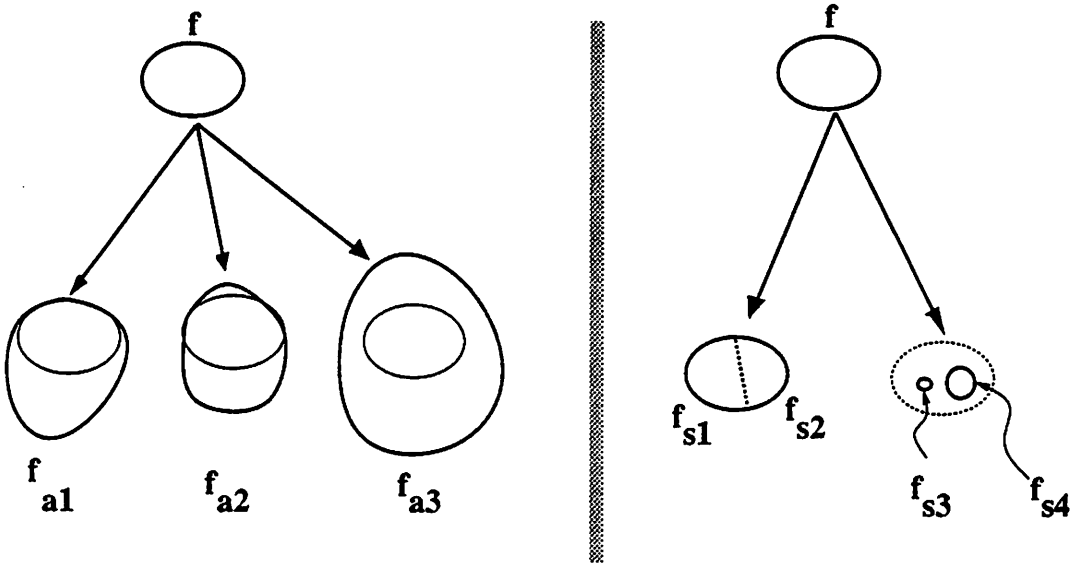
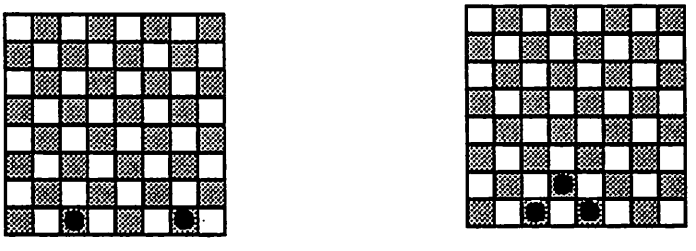


Figure 4.3 Abstraction and specialization applied to a feature f. Abstraction produces f_{a1} through f_{a3} and specialization produces f_{s1} through f_{s4}. Abstraction increases the domain of a feature while specialization restricts the domain.

is expensive, but there are some squares (the corner squares) that invariably satisfy it due to the geometry of the board. It is much easier to check whether individual, known squares are owned than to check each owned square for semi-stability.

In checkers, it is important to prevent the opponent from creating a king by reaching the back row. There are several special configurations of squares that, when occupied, prevent the opponent from doing so. Two such configurations are the Bridge and the Triangle of Oreo, illustrated for the Black player in Figure 4.4. Both configurations may be considered invariants of the "king prevention" property. It is



The Bridge The Triangle of Oreo

Figure 4.4 Two special configurations in checkers that protect the back row of the Black player.

much less expensive to test for these special configurations than to determine whether an arbitrary configuration prevents the opponent from creating a king.

4.3 Controlling the Transformations

In this theory, an initial feature is created from the performance goal. New features are generated subsequently by applying transformations to the existing features.

Unrestricted transformation of features is impractical because most transformations can be applied to most features. Therefore, a control strategy is employed that restricts the application of transformations to features that could plausibly benefit from the transformation. The control strategy is based on the purposes of the transformations and some assumptions about feature generation.

Two data about each feature are used to guide feature generation. Every feature incurs a time cost to compute its value in a state. Because the total time cost of the evaluation function is bounded, usually only a subset of the existing features are used in the evaluation function. A feature's selection for use in the evaluation function is an indication of its quality. The cost of a feature, and its use by the evaluation function give rise to the following two definitions:

1. An **expensive** feature is one whose cost exceeds a specified maximum. This maximum can be relative to the total cost allotted to the evaluation function, or an absolute limit placed on every feature.

2. An **active** feature is one that has been selected for use by the evaluation function. A feature that is not used by the evaluation function is called inactive.

The following rules are used to determine which transformations should be applied to a given feature:

1. Decomposition transformations may be applied to any feature. Because decomposition is based on specific syntactic forms and generally produces features that provide more information than the original, it is beneficial to apply decomposition transformations to both active and inactive features alike. Because the features resulting from decomposition are usually less expensive than the original, both expensive and inexpensive features can be decomposed beneficially.

2. If the feature is *expensive*, abstraction and specialization transformations are applied to it, in order to reduce its cost. In principle, abstraction and specialization could be applied to any feature because reducing feature cost is arguably always worthwhile. However, abstraction and specialization generally involve a trade-off between cost and accuracy. This control rule is based on

the assumption that the accuracy sacrifice is probably not worth making for features that are already inexpensive.

3. Only if a feature is *active* and *inexpensive* will goal regression be applied to it. Goal regression usually produces features that are more expensive than the original, so it is only applied to features that have already proven their worth. This rule is based on the assumption that if a feature is inactive, it is unlikely that a feature based on its pre-image would have a greater contribution to concept accuracy.

These principles are combined into a procedure for feature generation illustrated in Figure 4.5. New features are generated both from active features and from inactive features. Active features may be regressed through operators but inactive features will not be. An inactive feature may have been excluded from the evaluation function because it was not useful, or because it was too expensive for its contribution; in either case its pre-image should not be generated.

A simpler control strategy might keep *no* inactive features, and develop only the active ones. In this case, rejection by the evaluation function would serve as a severe filter that determined which features were worth keeping and developing. However, an inactive feature may not be a bad feature; it may simply be too expensive or overly specific, in which case further development may improve it. At the other extreme, a control strategy might keep *all* inactive features and continue developing them regardless of their worth. This strategy would quickly overwhelm the feature selection mechanism, which must examine the current features and select the best for use by the evaluation function.

Instead, an intermediate strategy is used, in which a fixed number of the inactive features are kept in a reserve set. After feature selection, the inactive features are ordered by their individual contributions to the evaluation function. The features of highest contribution are kept, and the remainder are discarded.

The next chapter discusses the Zenith system, an implementation of this theory.

```
FEATURE-GENERATION:
begin
  generate-from-active-features
  generate-from-inactive-features
end

GENERATE-FROM-ACTIVE-FEATURES:
begin
   For every feature F in active feature set
      if F can be decomposed
          then decompose F
      else if F is expensive
          then apply abstraction and specialization to F
      else apply goal regression to F.
   end for
end

GENERATE-FROM-INACTIVE-FEATURES:
begin
   For every feature F in the inactive feature set
      if F can be decomposed
          then decompose F
      else if F is expensive
          then apply abstraction and specialization to F.
   end for
end
```

Figure 4.5 Control of feature generation

CHAPTER 5

ZENITH: AN IMPLEMENTATION OF THE THEORY

The transformational theory of feature generation presènted in Chapter 4 has been implemented in a system called Zenith, which is discussed in detail in this chapter. The results of applying Zenith to the domains of OTHELLO and Telecommunications Network Management are discussed in Chapters 6 and 7, respectively.

The Zenith system is written in a combination of Quintus Prolog and C, and runs on Sun workstations. Zenith comprises approximately 17,000 lines of Prolog code and 6000 lines of C code. These figures do not include the code for the C4.5 concept learner, either of the OTHELLO opponents, or the 15 Quintus library files required by Zenith.

5.1 Overview

Figure 5.1 shows the basic components of the Zenith system and the data shared among them. Ovals represent processes and boxes represent data. Zenith operates in *cycles*; with each cycle new features are generated and evaluated. The data collected on features in a cycle are used to direct the generation of new features. Each cycle consists of the following three main steps:

1. Problem solving is performed using the current evaluation function. New training instances are generated from the problem solving episode.

2. New features are generated by applying the transformations to existing features. Data on feature performance, provided by the feature selection process, is used to control the generation.

3. Features are selected for the evaluation function, using the current feature set trained on instances from past problem solving. The result is a new evaluation function and data on feature performance.

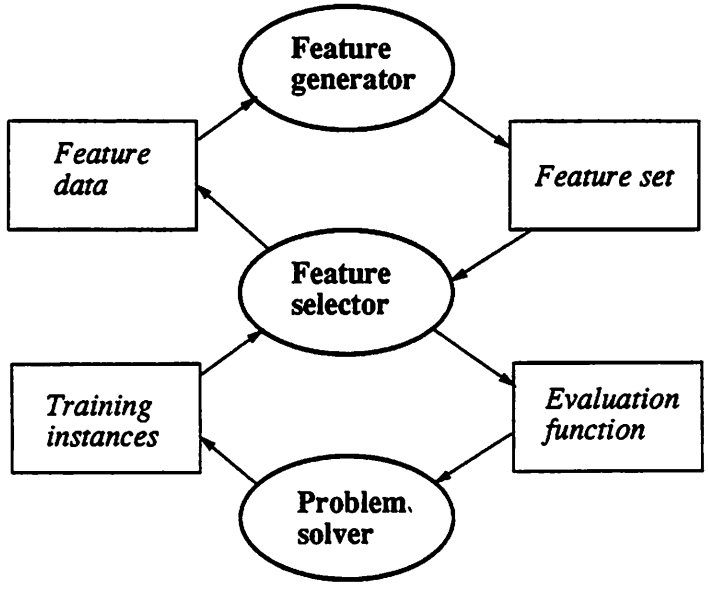


Figure 5.1 The structure of the Zenith system. Ovals represent processes and boxes represent data.

Zenith employs preference predicate training, as described in Section 2.3.2. Preference predicates were used for several reasons. Training instances can be derived either from interaction with an expert or from book games [Utgoff & Clouse, 1991b]. Learning a preference predicate is close to concept learning, for which most constructive induction systems have been designed, and accuracy can be easily measured and reported on the preference pairs. In addition, Zenith's feature selection method, described in Section 5.5, uses differences in concept accuracy to determine the relative value of a feature set. However, there is nothing inherent in either the theory or the Zenith architecture that precludes the use of temporal difference learning.

The remainder of this chapter is organized as follows. Section 5.2 discusses the problem solver and the generation of instances from problem solving episodes. Section 5.3 discusses the formalism used for features in Zenith and the data that are kept on them. Section 5.4 describes the transformations and how new features are generated. Finally, Section 5.5 describes how features are chosen from the existing feature set for inclusion in the evaluation function, and how excess features are pruned.

5.2 Problem Solving and Instance Generation

Zenith's problem solving component performs state-space search guided by the current preference predicate, which is derived from the feature selection step.

5.2.1 Problem Solving

Zenith searches through the state-space, maintaining a frontier of states ordered by preference. The frontier originally contains only the initial state of the domain. In searching for a goal state, the problem solver performs the following steps:

1. Extract the most preferred element of *Frontier*. Because *Frontier* is maintained as an ordered list, the most preferred element is the first.

2. Generate the set OPS of legal operations that can be performed in the current state. Each element of OPS is a fully instantiated operator.

3. For each element OP in OPS, generate the state that results from applying OP to the current state. Call the resulting set *Successors*. Record OPS and *Successors* for use later by the critic.

4. Order the elements of *Successors* using the preference predicate. This is done by applying the current preference predicate to the elements of *Successors* to determine which is the most preferred. If the preference predicate is antisymmetric and transitive, as it is in the Zenith implementation, then this can be done in a single pass over the successor states.

These steps are iterated until some S_i satisfies the goal of the domain.

In a two-player game, once a move is made it cannot be retracted, so there is no point in keeping a frontier of other moves that were possible at previous states in the game. Therefore, for two-player games, at the end of Step 4 the other states in *Frontier* are discarded so that *Frontier* is left containing only the most preferred successor state (in essence, *Frontier* is replaced by a single current state).

In principle, in a two-player game Step 4 could perform extensive search and expand many plies in order to determine which of the immediate successors is best. Zenith examines only the immediate successors, and so performs only one-ply search. This choice was made to simplify the performance component, and to reduce the influence of search on preference predicate quality. Conversely, if features are generated that improve single-ply search, they will also improve multi-ply search. Note, however, that this shallow search does not affect feature generation, because training instances are extracted from the expert's moves rather than from Zenith's own search.

In Step 4, if the preference predicate is antisymmetric and transitive, then the most preferred successor state can be found by using a single sweep across the successor states and retaining the most preferred state. The preference predicate in Zenith is guaranteed to be antisymmetric and transitive because it is trained on every delta vectors and its inverse. However, if either of these properties could not be guaranteed, a more careful counting procedure would have to be used, as explained by Utgoff and Heitman (1988).

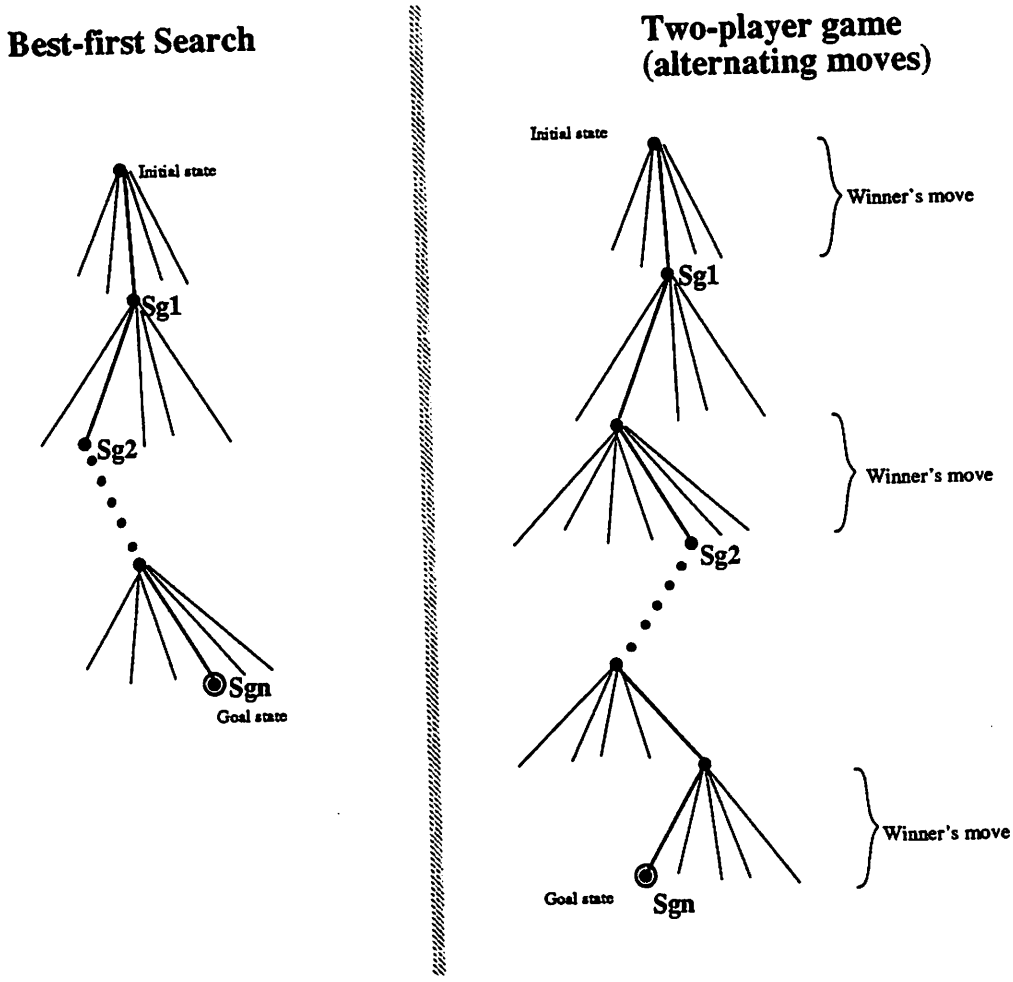


Figure 5.2 Generating preference pairs from a solution path

5.2.2 Preference Pair Generation

When a state S_i is found that satisfies the goal of the domain, the critic is invoked on the performance trace. The critic generates instances from the performance trace and adds them to the instance base. Instances are preference pairs, each of which is of the form $\langle S_a, S_b \rangle$, where state S_a is preferred to state S_b. The instances are generated from the performance trace as described in Section 2.4.2.

Instance generation differs slightly between a two-player game and single-agent (best-first) search. Figure 5.2 illustrates the difference. If problem solving employs single-agent best-first search, the successful performance trace is determined by the goal state found, and the path from the initial state to the goal state is used to generate preference pairs. Every state along the goal path is used to generate preference pairs. A given state S_g along the goal path is preferred to each alternative state S_p. So for each of the alternative choices, a preference pair $\langle S_g, S_p \rangle$ is created. This is similar to

Callan's (1993) method for creating preference pairs from best-first search episodes. However, Zenith's method differs from the method of Utgoff and Saxena (1987) which creates pairs using *every* other node in the frontier that is not on the goal path.

If the domain is a two-player game with players alternating moves, preference pairs are extracted using only the winner's moves; it is assumed that the loser's moves will not provide useful training information. A given move made by the winner determines a preferred state S_g, and each alternative move determines an alternative state S_p. For each of the alternative moves, a preference pair $\langle S_g, S_p \rangle$ is created from the state S_p resulting from the alternative move. Each of these pairs is added to the instance base.

The instances (preference pairs) created by the critic are added to the instance base up to a maximum limit. Instances are accumulated so that training occurs over the entire set of instances seen so far, and no experience is lost. In practice, an upper limit of 3000 instances was used, and it was rarely met.

The instance base stores both the raw instances, expressed in terms of primitives, and the feature vector representation of each instance. The instance base must keep the raw instances because the set of features changes with every cycle, and whenever features are added the feature values must be computed using the raw instances.

5.2.3 Instance Encoding

Experiments with Zenith used two concept forms: a linear threshold unit (LTU) and a univariate decision trees. In both cases, instances for the learning algorithm are encoded as delta vectors created from the preference pairs. An explanation of the use of delta vectors, and a justification for using them with LTUs, appears in Section 2.4.2.

Following the use of Callan (1993), delta vectors were also used to train univariate decision trees. Delta vectors express differences between preferred and non-preferred states. Univariate decision trees have nodes containing univariate tests at each node. Each attribute of an instance had to represent the difference in feature values between the two states of the preference pair.

5.3 Features

We adopt the Prolog notation that a literal beginning with an uppercase letter is a variable, and a literal consisting of all lowercase letters or numbers is a constant. Thus, `X` and `Square` are variables, and `black` and `e4` are constants. Logical AND (conjunction) is indicated either with "$\wedge$" or with commas separating terms. Logical OR (disjunction) is indicated with "$\vee$". Negation is indicated with the symbol "$\neg$".

5.3.1 Feature Formalism

The feature formalism used in Zenith is similar to that of Michalski's (1983) counting arguments rules. Every feature has two components. The first component is equivalent to the body of a Horn clause [Sterling & Shapiro, 1986], and will be called a *formula*. More precisely, in BNF a formula is:

```
Formula ::= Formula ∧ Formula
Formula ::= Formula ∨ Formula
Formula ::= ¬ Formula
Formula ::= Term
```

where `Term` is a first-order term that is not otherwise a `Formula`.

The second component of a feature is a *variable list*, which is a list containing zero or more of the variables occurring in the formula.

A feature is evaluated in a state by counting the distinct values of its variable list that can satisfy the formula. That is, the value of a feature is the cardinality of the set of all unique assignments to variables in VarList that satisfy Formula. Feature values are normalized to $[+1, -1]$ before being presented to the concept learner. This normalization prevents features of differing ranges from biasing the concept learner.

Feature evaluation is illustrated in Figure 5.3, which shows an OTHELLO state, a formula, and four features (f1 through f4). All of the features use the formula shown, but each has a different variable list. The table in Figure 5.3 shows the variable list for each feature, followed by the set of values in the OTHELLO state that satisfy the variable list. The value of each feature, shown in the right-hand column, is the cardinality of this set.

A formalism based on counting is used because it is more expressive than a formula alone. It allows a feature to calculate the *number of ways* in which the formula can be satisfied. The formalism is a superset of a boolean rule. A feature with an empty variable list is equivalent to a rule: its value is 1 if the formula is satisfiable and 0 if the formula is unsatisfiable.

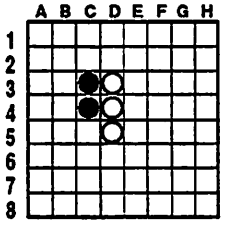
The counting formalism was extended to allow for numeric-valued functions decomposed by the split-arith-comp and split-arith-calc transformations. The extension is a variable annotation that allows the numeric value of a variable to substitute for a number of counted items. If a variable in a feature's variable list is so annotated, then the count for that tuple will be multiplied by n. The motivation for this extension is given in Section 5.4.1. As an illustration of this extension, consider the formula:

$$p(A), q(A, B), r(A, C)$$

and the following set of variable values that satisfy it:

$$\{[1, 1, 1], [2, 4, 8], [3, 9, 27]\}$$

Following are four features, each using this formula and a different variable list. The notation v(X) is the annotation indicating that the value of a variable X should be used in the count.



Formula:

blank(X) ^
neighbor(X,Y) ^
owns(black,Y)

Name	Variable list	Values satisfying variable list	Feature value
f1	[]	{[]}	1
f2	$[X]$	{[b2], [b4], [b5], [c2], [c5], [d2]}	6
f3	$[Y]$	{[c3], [c4]}	2
f4	$[X, Y]$	{[b2, c3], [b4, c3], [b4, c4], [b5, c4], [c2, c3], [c5, c4], [d2, c3]}	7

Figure 5.3 A sample OTHELLO state and four features evaluated in it. All features are based on the formula shown, but each uses a different variable list.

Variable list:	Values:	Count:
[A]	[[1],[2],[3]]	1+1+1 = 3
[v(A)]	[[1],[2],[3]]	1+2+3 = 6
[A,v(B)]	[[1,1],[2,4],[3,9]]	1+4+9 = 14
[A,v(B),v(C)]	[[1,1,1],[2,4,8],[3,9,27]]	1+32+243 = 276

This "v()" annotation was introduced to accommodate domains in which the states maintain counts of domain objects rather than the identities of the objects themselves. For example, in a telecommunications network the identities of individual calls passing through a network are not available; only counts of calls are. It is possible to re-express the problem specification such that call identities are available to be counted. In this case, the standard counting formalism would suffice, and the "v()" annotation would be unnecessary. However, in domains like telecommunications network management this assumption is unrealistic; such information is not kept in an actual telecommunications network, and could not be made available easily. This requirement necessitated extending the formalism.

Other formalisms were considered for features. A rule is a simpler, more common formalism that could have been used. However, a rule only provides information on whether its formula is satisfiable in a state. Because features are used to evaluate individual problem-solving states, they typically measure the degree to which a condition has been achieved, or the opportunity for achieving a condition in the

future. Rules cannot capture such gradations; a large set of them usually would be required in order to approximate the effect of a feature measuring the degree of satisfaction. Similarly, rules with propositional terms were rejected as being too specific and not sufficiently expressive.

When a feature is created it inherits the variable list of the feature from which it was transformed. Variables can be removed from the variable list by the remove-variable transformation, discussed in Section 5.4.3.

5.3.2 Data Kept on Features

Two primary performance measurements are made of each feature. These measurements are used in feature selection and generation.

The *discriminability* of a feature is the ability of the feature, when used in isolation by the concept learner, to discriminate preference pairs. It is the individual ability of the feature to discriminate examples. This measure ignores the effects of other features in the feature set.

The *cost* of a feature is the average amount of time it takes to compute the value of the feature in a state. Cost is determined empirically by measuring the CPU time required to evaluate the feature over the entire set of instances and dividing by the number of instances.

Both discriminability and cost are measured over the entire instance set. However, occasionally a feature is created that is so expensive that much time would be consumed in evaluating the feature on all the instances. For example, if a feature requires an average of 2 seconds per instance, then 100 CPU minutes will be required to compute the feature's value on a set of 3000 instances. Features that are more expensive than the total cost allowed for the evaluation function will never be used in the evaluation function, and the evaluation of such features slows the system. In order to avoid this, when a new feature is created Zenith computes its values on a small random sample of instances in order to measure its time cost. If the feature's cost per instance on this sample exceeds the total evaluation function cost limit, then the feature is deemed *exorbitant* and is not evaluated on the complete set. Instead, its cost and values on the random sample are used to approximate its cost and values on the complete set.

Many other data are recorded for features in addition to cost and discriminability. For example, the following data are recorded:

- The cycles in which the feature was created and destroyed. This information is used for tracking the lifetimes of features.
- The derivation of the feature. Every feature except the initial feature is derived by the application of a transformation to an existing feature. By storing the derivation of every feature it is possible to print a derivation tree containing every feature in the system.

```
1    f9(B) :- count([A], legal_move(A,black), B).
2    Representative #8, active feature #4, weight is 2.86082
3    Delta range is (-7.0,7.0), frequency = 687/890
4    Cost is 98msecs, discriminability is 50.
```

Figure 5.4 An example feature from Zenith and some of the data kept on it.

- The number of instances over which the feature has been evaluated. Whenever instances or features are added, this information is consulted to determine whether the instance vectors must be updated. When the instance vectors are updated, the time cost of the feature is updated as well.

Zenith periodically discards features that have not proven their worth; this is discussed in detail in Section 5.5. Although a feature may have been discarded, its definition is recorded. Whenever a new feature is generated, its definition is compared against all previously generated features, including those that have been discarded. If it is identical to a feature that has previously been generated, the new feature is not created.

5.3.3 An Example Feature from Zenith

Figure 5.4 shows a Zenith feature from the OTHELLO domain, illustrating some of the information kept. Line 1 shows the feature's definition. The feature's name is f9, its variable list is `[A]` and its formula is `legal_move(A,black)`. This feature counts the number of legal moves of the black player.

Line 2 indicates that this is the eighth unique feature (called a representative). The feature is active, and f9 is known as feature #4 to the evaluation function. In this example the evaluation function is a linear threshold unit, and the LTU has assigned a weight of 2.86082 to the feature.

Zenith stores delta values of features. That is, for a state pair $\langle S_1, S_2 \rangle$ it stores the delta value $f(S_1) - f(S_2)$. The delta range reported in line 3 is the minimum and maximum of the delta values seen. The frequency is the number of instances in which the delta value is non-zero; in this case, this feature produces non-zero delta values for 687 of the 890 instances extant.

Finally, line 4 reports the feature's cost as 98 milliseconds as evaluated over the 890 instances. Feature f9 used alone is able to discriminate 50% of the instances; that is, to separate correctly half of the preference pairs. Fifty-percent appears to be no better than random guessing; however, the frequency of the feature is about 77%, so it only differentiates 77% of the pairs; for the remaining 23% of the pairs it has equal values for the states. Among the 77%, it discriminates about 65% of the pairs accurately, resulting in its overall discriminability figure of 50.

Table 5.1 Transformations in Zenith

Class	Name	English description
Decomposition	split-conjunction	Split conjunction into independent parts
	remove-negation	Replace $\neg P$ with P
	split-arith-comp	Split arithmetic comparison into constituents
	split-arith-calc	Split arithmetic calculation into constituents
Abstraction	remove-LC-term	Remove least constraining term of conjunction
	remove-variable	Remove a variable from the features variable list
Goal Regression	regress-formula	Regress the formula of a feature through a domain operator
Specialization	remove-disjunct	Remove a feature's disjunct
	expand-to-base-case	Replace call to recursive predicate with base case
	variable-specialize	Find invariant variable values that satisfy a feature's formula

5.4 Feature Generation

The initial feature used by Zenith is a simple function of the performance goal. Its formula is the performance goal itself, and its variable list is the set of variables in the performance goal.

New features are generated by the application of transformations to existing features. Zenith's transformations are domain-independent and are based on the classes defined in Chapter 4. The transformations are summarized in Table 5.1 and are discussed in the next four sections. Feature generation is *constructive*: a transformation applies to a feature and creates one or more new features from it, without losing the original feature.

When a feature is created it inherits the variable list of the feature from which it was transformed. Transformations can expand predicates when necessary, by replacing a predicate call in a feature with the predicate's definition. Any new variables that are introduced by the predicate expansion are added to the variable list as well.

5.4.1 Decomposition Transformations

Decomposition transformations create features that provide more detailed information about the search path than the features from which they transformed.

Split-conjunction operates on features with conjunctive formulas. It partitions the formula's terms based on the variables it uses, such that none of the partitions share variables. A separate feature is created for each partition. The justification for this decomposition is that the satisfiability of each partition is not dependent on the satisfiability of any other partition. For example, the formula

$$p(X) \wedge q(X,Y) \wedge r(Y) \wedge s(Z) \wedge t(Z)$$

would be partitioned into two features:

$$p(X) \wedge q(X,Y) \wedge r(Y)$$

based on their common use of X and Y, and

$$s(Z) \wedge t(Z)$$

based on their common use of Z.

Remove-negation operates on features whose formulas contain negations. It replaces a negated term $\neg P$ with the expression P, and it adds the variables used in P to the new feature's variable list. This transformation overcomes a limitation of logic programming: the variable values satisfying a negated term cannot be enumerated, but those satisfying a non-negated (positive) term can. The resulting feature counts the values of variables internal to P.

Split-arith-comp transforms a feature whose formula contains an arithmetic comparison ($=, \neq, <, \leq, >, \geq$) between two variables X and Y. The feature is split apart into two features that calculate the values X and Y. This transformation is similar to Callan's (1991b) AE transformation.

Split-arith-calc applies to features that perform an arithmetic calculation. A separate feature is created for each arithmetic operand in the calculation. For example, assume a feature consists of the formula:

$$p(A) \wedge q(A) \wedge r(B) \wedge s(C) \wedge calc(D,(A+B)/C) \wedge t(D)$$

The *calc* predicate calculates a value from its second argument and binds its first argument to the resulting value[1]. The split-arith-calc transformation would create three new features from this formula, one for each of:

1. $p(A) \wedge q(A)$
2. $r(B)$
3. $s(C)$

The transformations split-arith-comp and split-arith-calc are unusual because they generate features directly from the operands of arithmetic expressions. Consider the expression $r(B)$ decomposed from the example above. The new feature should compute the value B generated by $r(B)$. It should not count the number of B values that satisfy $r(B)$ because it is the numeric value of B that is used in the original arithmetic expression. Therefore, these operands are annotated as described in Section 5.3.1. In effect, the annotation states that the variable is the result of counting done implicitly by some predicate in the formula. In the example above, $r(B)$ calculates a value for B, which would be annotated.

[1]Readers familiar with Prolog will recognize this as a renaming of the built-in "is" predicate.

5.4.2 The Goal Regression Transformation

Zenith uses a single goal regression transformation, regress-formula. It produces new features by regressing the formula of an existing feature through a domain operator. The transformation chooses a domain operator, then chooses an agent (a parameter of the operator), and generates the pre-image of the formula with respect to the instantiated operator.

For example, assume regress-formula is applied to a feature with the formula `owns(Player, Square)`. Regress-formula would first choose an operator. OTHELLO has only one operator, expressed as `move(Player,Square2)`, meaning that `Player` moves to `Square2` on the board. Note that in this example the variable `Player` is shared between the formula and the operator. Regress-formula would then select an agent for the operator. There are two agents (players) in OTHELLO: the `black` and `white` players. Regress-formula would thus perform the regression with one value of `Player` at a time. That is, it would regress `owns(black,Square)` through `move(black,Square2)`, then it would regress `owns(white,Square)` through `move(white,Square2)`.

The result of goal regression is a disjunction of cases corresponding to the pre-images of the formula. Regress-formula takes the expression resulting from goal regression and creates a feature from every disjunct in the resulting expression. In the example above, regressing `owns(black,Square)` through `move(black,Square2)` would yield these three disjunctive pre-images:

- `Square` was the square taken by `black` (so `Square=Square2`), *or*
- `Square` was already owned by `black` and was not affected by the move, *or*
- `Square` was owned by `white` and it was flipped by `black`.

So three new features would be created from this regression, each feature corresponding to one of these pre-images.

5.4.3 Abstraction Transformations

Remove-LC-term performs term-dropping abstraction similar to that in ABSTRIPS [Sacerdoti, 1974]. It removes the least constraining term occurring in a formula. Zenith places a term in one of three categories, in order of increasing criticality:

1. A term that can be achieved by an operator.
2. A term that is state-*dependent* (*i.e.*, the term changes between states) but cannot be achieved by an operator.
3. A term that is state-*independent*, *i.e.*, one that never changes.

The criticality of a term is determined automatically by examining the problem specification. ABSTRIPS depended upon the domain having STRIPS-style operators with explicit preconditions, postconditions and deletelists; this made the criticality judgments straightforward. Zenith does not assume that the operators of its domain are representable in STRIPS notation, so it uses somewhat more complex (and less exact) techniques for determining criticality:

- If the term's predicate is declared to be non-state-specific in the problem specification, then the term is of criticality 3.
- The term is of criticality 1 if it can be achieved directly by an operator. This is tested by regressing the term through every operator and looking at the resulting pre-image. If the pre-image does *not* include the term, then Zenith assumes that the agent can achieve the term directly, and the term is assigned a criticality of 1.
- If the term is declared to be state-specific but *does* appear in its own pre-image, then it is assigned a criticality of 2. This means that the term can be affected in some way, but not (apparently) achieved directly by an operator.

Using these principles, remove-LC-term drops the least critical term and creates a new feature from the resulting formula. If several terms are of the same criticality, it creates multiple features, each with one least-critical term removed.

The second abstraction transformation, remove-variable, removes a single variable from a feature's variable list. Remove-variable is classified as an abstraction transformation because the resulting feature provides less information about the formula, and because the resulting feature is also usually less expensive to compute than the original.

When a feature is created it inherits the variable list of the feature from which it was transformed. Variables can then be removed by the remove-variable transformation. An alternative strategy might be to create new features with empty variable lists, and to use an add-variable transformation that adds a variable to a feature's variable list. However, a feature with an empty variable list may have low discriminability because it only distinguishes states in which the formula is satisfied from states in which the formula is not. This coarse distinction may not be useful to the problem solver, and such a feature may be discarded before being developed. By creating features with full variable lists, Zenith begins with a fully informed version of the feature; if the feature proves valuable, Zenith can then discard variables to create less expensive versions of it.

5.4.4 Specialization Transformations

Zenith uses three specialization transformations. Remove-disjunct removes a disjunct occurring in a feature's formula.

Expand-to-base-case replaces a call to a recursive predicate with that predicate's base case. Evaluating a predicate's base case is usually much less expensive than evaluating the recursive predicate.

The variable-specialize transformation looks for variable values that always satisfy a feature's formula, and creates a new feature that uses those specific values. Specifically, given a feature f with a single variable X in its variable list, variable-specialize splits f's formula into a prefix p that binds X and a suffix q that uses X. The following set is then computed:

$$S = \{X \mid p(X) \rightarrow q(X)\}$$

S is computed empirically by sampling Zenith's database of training instances, generating X values and discarding X values that satisfy $p(X) \wedge \neg q(X)$. If any elements remain in S, a new feature f' is created that has formula:

$$p(X) \wedge (X \in S)$$

In f' the conjunction q has been replaced by a set inclusion test. The latter test is usually much cheaper than q.

5.4.5 Controlling Feature Generation

New features are generated by applying transformations to the existing features. Feature generation consists of selecting a feature and selecting a single transformation to apply to it.

Zenith maintains two feature sets, one for **active** features and one for **inactive** features. A feature is active if it has been selected for use in the evaluation function. This selection process is explained in Section 5.5.

Because feature costs vary, the active feature set varies in size, although it is bounded indirectly by the maximum evaluation function cost. The inactive feature set is a reserve set of fixed size for features that have been discarded from the active set. Ideally, Zenith would keep every feature created; however, it is prohibitively expensive to do so because the features quickly overwhelm both the feature selection and generation processes. Because both the active and inactive feature sets are bounded, Zenith's feature generation process constitutes a kind of *beam search* [Newell, 1978] of the feature space, where the frontier of the search is limited. The beam is used as an absolute limit on the number of inactive features kept.

Figure 5.5 shows the feature generation control algorithm. Transformations are first applied to the active features. For a given active feature, the first applicable transformation is applied exhaustively, meaning that every new feature producible by the transformation is generated. Most transformations can produce more than one feature per application. For example, remove-LC-term may find several least-constraining terms to remove.

After each active feature has been processed by generate-from-active-features, the inactive features are processed. They are first sorted in decreasing order by discriminability (the ability of each feature alone to discriminate the instances). This produces a list FL, which is then used to generate new features. Features are removed iteratively from FL and transformations are applied to them. Each inactive feature is allowed to produce at most one new feature. Zenith's redundancy checking, explained in the next section, ensures that if the feature remains inactive in subsequent cycle, further generation will create a new feature rather than re-creating one already generated.

Because features compete to remain in the active and inactive sets, if unlimited feature generation were allowed from the inactive feature set, the competition would be very severe. This restriction serves to limit the number of new features introduced from inactive features in every cycle.

5.4.6 Simplification and Redundancy Checking

When a new feature is generated it is passed to a feature simplifier, which performs certain truth-preserving optimizations and simplifications to a feature's formula. Some transformations create complex expressions that can be greatly simplified. For example, regress-formula can create unsatisfiable sub-expressions whose unsatisfiability can be recognized easily, and removal of such sub-expressions can result in vast reductions in expression size.

In order to effect such reductions, Zenith performs truth-preserving simplifications using domain-independent principles. The feature simplifier performs code conversions such as:

- Simplifying logical expressions, such as

$$(p(X) \wedge \neg p(X)) \Longrightarrow \textbf{false}$$

- Evaluating tests whose operands are constant, such as

$$c1 = c1 \Longrightarrow \textbf{true}$$

  The simplifier can also detect and remove certain expressions that are non-state-specific. For example, if neighbor is known not to be a state-specific relation then `neighbor(a1,e,a2)` can be evaluated by the simplifier and replaced by **true** or **false**.

- Removing redundant tests, *e.g.*

$$p(X) \wedge (q(X) \vee [p(X) \wedge r(X)]) \Longrightarrow p(X) \wedge (q(X) \vee r(X))$$

- Various Prolog-specific code improvements such as moving tests within a conjunction to ensure that variables are bound, and removing needless unifications.

When no further simplifications are possible, the simplifier terminates and the resulting simplified feature is passed to the redundancy checker. The redundancy checker determines whether the feature has been discovered previously. It does this by searching through the list of previously generated features and determining whether the new feature is syntactically equivalent to a previously discovered feature. Two features are considered syntactically equivalent if their formulae are identical up to variables and their variable lists are set-equivalent. If a newly generated feature is syntactically equivalent to a previously generated feature, the new feature is not introduced.

5.5 Feature Selection and Pruning

After new features are generated, an appropriate set must be selected for use in the evaluation function. The resulting evaluation function must be able to evaluate a state within the time limit imposed, and usually there will be too many features for all of them to be used.

5.5.1 Feature Selection

The goal of feature selection is to select a subset of features that produce an evaluation function as accurate as possible but whose combined costs does not exceed the time limit in evaluating a state. It is assumed that the summed costs of the features comprises the cost of the evaluation function, and the cost of combining the feature values into a state evaluation is negligible.

The feature selection step has two results. First, the evaluation function produced will be used in problem solving in the next cycle. Second, the feature selection method partitions the features into active and inactive subsets. This partitioning is used in feature generation in the next cycle.

Determining the optimal subset of features via exhaustive search is prohibitively expensive, so Zenith uses a *sequential backward selection* method [Kittler, 1986], illustrated in Figures 5.6 and 5.7. The method creates an evaluation function from the entire feature set, then it casts out individual features until the total cost of the evaluation function falls below the imposed limit. The feature to be cast out is the one that produces the smallest decrease in concept accuracy when removed from the set. Ties are broken by eliminating the feature with greatest cost. When the process terminates, the remaining features are used in the evaluation function. These are termed *active* features, and the features that have been cast out are termed *inactive*.

At this point, the inactive feature set contains all of the features not selected for inclusion in the evaluation function. Zenith must prune the inactive feature set if it exceeds the beamsize, described in Section 5.4.5. The features in the inactive set

are ordered by their discriminability[2]. The remainder are discarded and will not be generated again.

Samuel's (1959) system also employed a feature selection process. His system used a fixed-size (16) set of active features and a fixed-size (22) reserve queue of inactive features. After every update of the weight set, the system would check each feature's assigned weight. If a given feature in the active set repeatedly had the lowest magnitude weight, it would be "demoted" to the bottom of the reserve queue, and the feature at the top of the reserve queue would be placed into the active set. This replacement scheme was used to ensure that all features had a chance to be used in the evaluation function. Samuel suggested that features that were repeatedly quickly discarded from the active set could be removed permanently by a human.

There are several differences between Samuel's system's feature selection process and the one used in Zenith. Samuel assumed that all features have essentially equal cost, so a feature's contribution (indicated by its weight) was the only factor considered; whereas Zenith's selection method takes into account both cost and contribution. Furthermore, Zenith uses an active feature set that is bounded in cost rather than in size, so its size may vary. Most importantly, Zenith keeps a set of instances against which it can test the accuracy of a set of features. The accuracy of a feature set with and without a given feature is a better indicator of that feature's worth than is coefficient magnitude. Therefore, the sequential backward selection process used in Zenith, though more expensive than Samuel's method, is more accurate.

5.5.2 Measuring Evaluation Function Accuracy

Finally, Zenith measures the accuracy of the evaluation function. The accuracy of the evaluation function serves as an indication of the quality of the current active feature set. To measure accuracy, Zenith repeatedly performs the following steps.

1. Zenith partitions the instance set randomly into training and testing sets, with 2/3 of the instances allocated for training and 1/3 for testing.

2. Zenith trains the concept learner on the training set, producing an evaluation function.

3. The accuracy of the evaluation function is measured by counting the number of testing instances correctly classified. Because the evaluation function is a preference predicate, an instance is a pair $\langle S_a, S_b \rangle$ and the pair is considered correctly classified only if the evaluation function expresses a consistent preference; that is, if it prefers S_a to S_b and does not prefer S_b to S_a. The result of this step is an accuracy value of $\frac{\#\text{Correct instances}}{\#\text{Total instances}}$.

[2] A function of discriminability and cost could be used to order the inactive features. Section 8.2.1 discusses the difficulties encountered in deriving such a function.

This sequence of steps is iterated, with the accuracies from step 3 sampled, until the true average accuracy is known to lie within an interval of ±2% around the averaged accuracy, with 99% probability. When this condition is met, the average accuracy is recorded.

Feature generation and selection is sketched for two cycles in Figure 5.8. In the first cycle shown, Zenith already has active and inactive feature sets (a). From these sets, new features are generated (b). The new features combined with the old compete to be selected by the evaluation function. The selection process results in a new set of active and inactive features (c). In the next cycle, the features are transformed again (d) and again compete for inclusion in the active and inactive feature sets (e).

```
FEATURE-GENERATION:
begin
  generate-from-active-features
  generate-from-inactive-features
end

GENERATE-FROM-ACTIVE-FEATURES:
begin
   For every feature F in active feature set
      if F can be decomposed
          then decompose F exhaustively
      else if F is expensive
          then exhaustively apply abstraction and specialization to F
      else apply goal regression exhaustively to F.
   end for
end

GENERATE-FROM-INACTIVE-FEATURES:
begin
   Sort the inactive feature set F producing an ordered
       feature list FL;
   For every feature F in FL
      if F can be decomposed
          then decompose F
      else if F is expensive
          then apply abstraction and specialization to F.
   end for
end
```

Figure 5.5 Control of feature generation

```
feature-selection()
begin
  FS = initial-feature-set();
  while TotalCost(FS) > MaxAllowedCost do
        WF = find-worst-feature(FS);
        FS = FS - WF;
        end while
  return(FS);
end;

initial-feature-set()
begin
  Initial = null;
  for F in all features
      if ( cost(F) ≤ MaxAllowedCost and
           (discriminability(F) > 0 or range(F) is not empty) )
         then add F to Initial;
      end for;
  return(Initial);
end;

find-worst-feature(FS)
begin
  WorstFeature = null;
  WorstFeatureContribution = -1;
  for F in FS do
      FContrib = differential-accuracy(FS, F);
      if  ( FContrib < WorstFeatureContribution or
            (FContrib = WorstFeatureContribution and cost(F) < cost(WorstFeature) ))
          then          /* F becomes the new "worst feature." */
            WorstFeature = F;
            WorstFeatureContribution = FContrib;
      end for;
  return(WorstFeature);
end;
```

Figure 5.6 The Sequential Backward Selection algorithm used by Zenith to select features.

```
differential-accuracy(FS, F)
 begin
   FSA = accuracy attained by evaluation function on feature set FS;
   FSA2 = accuracy attained by evaluation function on set FS-F;
   return(FSA-FSA2);
 end;
```

Figure 5.7 The Sequential Backward Selection algorithm, continued.

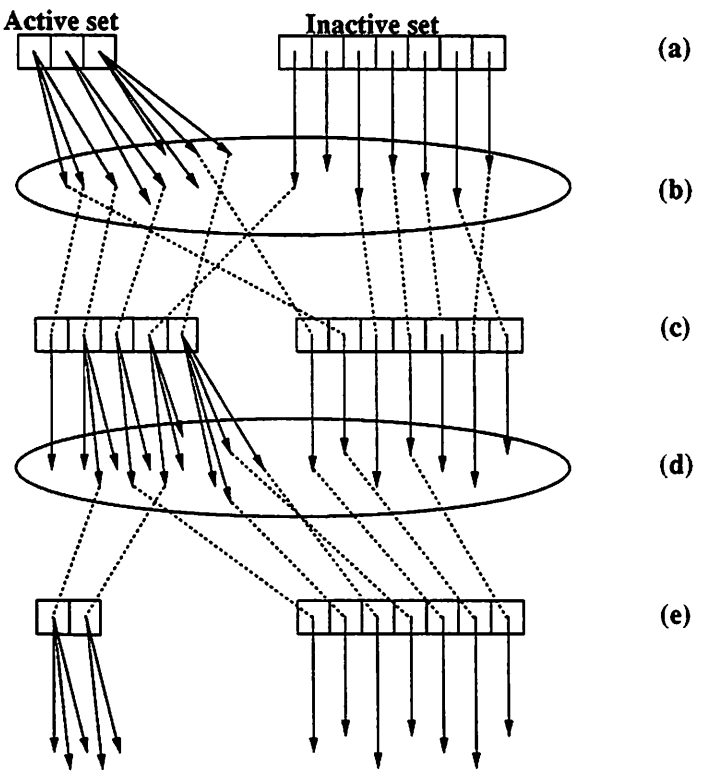


Figure 5.8 An illustration of the feature generation process.

CHAPTER 6

OTHELLO

The first domain to which Zenith was applied is a board game called OTHELLO [1] OTHELLO was chosen as a domain primarily because many features are known for it, so Zenith's discoveries could be compared against the catalog of handcoded features in this domain. Donald Mitchell's Master's thesis (1984) contains a description of many features used in published systems. OTHELLO is also an attractive domain because its problem specification constitutes a complete though intractable domain theory, because it can be played using state-space search, and because it has been used as a domain by other researchers in artificial intelligence [Rosenbloom, 1982; Lee & Mahajan, 1988; De Jong & Schultz, 1988].

Sections 6.1 and 6.2 discuss the game of OTHELLO and the encoding of the OTHELLO problem specification. Section 6.3 presents experiments in this domain showing the effects of Zenith's feature generation. Section 6.4 discusses the features that were created by Zenith, comparing them with known features for OTHELLO.

6.1 The Game of OTHELLO

OTHELLO is a two-player game played on an 8×8 board. One player is designated the White player, the other the Black player. There are 64 discs colored black on one side and white on the other. The starting configuration is shown in Figure 6.1a. Black always moves first, with players alternating turns.

On a turn, the player can place a disc on any empty square that brackets a span of the opponent's discs ending in a disc of the player's own color. The span can be horizontal, vertical or diagonal. For example, in Figure 6.1b, Black could place a black disc on any of the squares marked with an "X". When a player places a disc at the end of a span, all the discs in any formed span are *flipped* (changed to the player's color), and the player is said to own them. For example, if Black takes square f6 in Figure 6.1b, the board in Figure 6.1c would result. A player thus gains discs

[1] OTHELLO is also known as Reversi. OTHELLO is a registered trademark of Tsukada Original, licensed by Anjar Co., ©1973, 1990 Pressman Toy. All rights reserved.

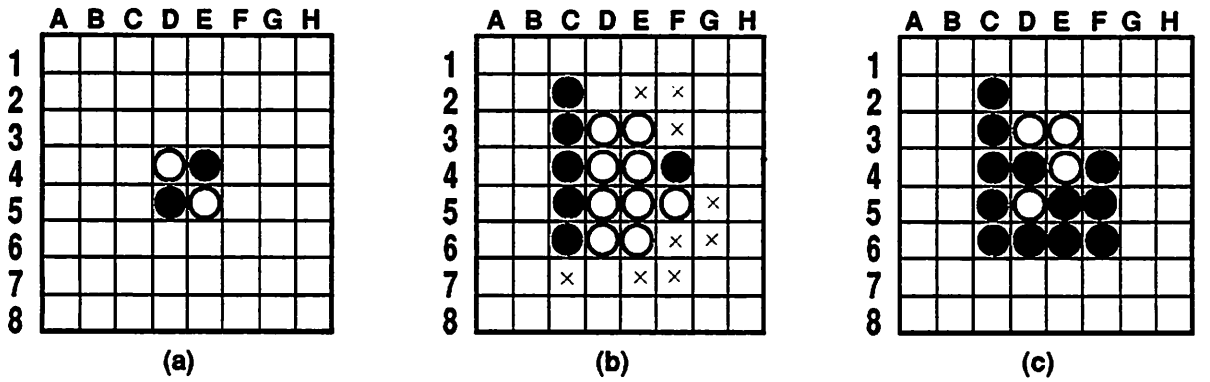


Figure 6.1 OTHELLO boards: (a) Initial OTHELLO board (b) Board in mid-game (c) After Black plays F6 on (b).

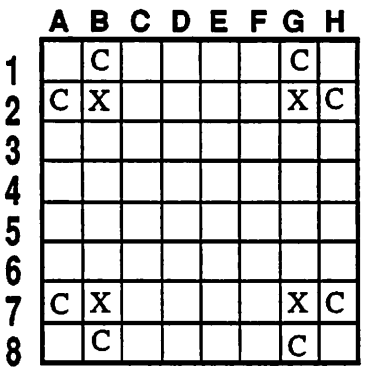


Figure 6.2 Special squares in OTHELLO

by placing them on the board and by flipping discs of the other player. In certain configurations pieces cannot be flipped because no span can be placed through them; these pieces are said to be *stable*.

The game continues until neither player has a legal move, which usually occurs after all 64 squares have been taken. At this point the player with the greater number of discs wins the game, and the number of points by which the player has won is simply the difference between the two piece counts. For analysis, an OTHELLO game is often broken up into three segments: the early game (from 4 to 16 pieces), the middle game (from 17 to 32 pieces), and the late game (from 48 pieces to the end). There are usually 60 moves in an OTHELLO game because the game usually does not end until the board is full. Frey (1986) estimates that OTHELLO, with a search space of 10^{60} nodes, is intermediate in complexity between checkers (with 10^{40} nodes) and chess (with 10^{120} nodes).

There are several distinguished squares on the OTHELLO board, shown in Figure 6.2. A corner square is inherently stable because once it is occupied there is no sequence of moves that will flip a disc on it; in addition, stable squares can form the basis for larger stable regions. Therefore, gaining control of corner squares is a goal in Othello. There are two sets of distinguished squares adjacent to the corners that are significant in corner square control. The squares along the edges immediately adjacent to the corners are called C squares (A2, B1, G1, H2, A7, B8, H7 and G8). X squares (B2, G2, B7 and G7) are diagonally adjacent to the corners. X and C squares are both considered dangerous to own because they can allow the opponent to move into the corner. C squares are somewhat less vulnerable than X squares because it is more difficult to move onto an edge than to flip a piece along a major diagonal.

6.2 The OTHELLO Problem Specification

The problem specification for OTHELLO is given in Appendix A. Following the terminology of explanation-based learning [Mitchell, Keller & Kedar-Cabelli, 1986], the problem specification is also called a *domain theory*. OTHELLO's domain theory is complete but intractable. It is *complete* because all legal moves are provable by the domain theory, and they are provable down to the observables (the instance-level features) of a state, and there are no hidden state variables. Note that completeness does not imply determinism; there is inherent nondeterminism in the domain because the opponent's moves cannot be controlled. OTHELLO is *intractable* because a given move cannot be explained in terms of its effects on the outcome of the game. The depth of the search tree prohibits an explanation of how a given move affects the final score.

The domain theory contains meta-information about predicates:

- `Mode` declarations [Warren, 1977] specify information about Prolog predicates, indicating which arguments of a predicate are used for input, output or both. An input variable to a predicate is a variable that must be bound before a predicate is called; an output variable is one that produces a value. `Mode` declarations are used to determine whether a given formula is legal.

- `Determinate` declarations specify the conditions under which a procedure call is determinate. Determinate calls produce at most one value for a variable. Determinate declarations are specified by the configuration of bound and unbound variables in a predicate call. `Determinate` declarations are used by the simplifier in deciding whether a given variable can be removed from a variable list.

- `Operational` declarations specify predicates that are considered to be facts about a state, or that are so inexpensive that they have negligible cost. For example, the `owns(Player,Square)` predicate in OTHELLO is declared to be operational because it is very inexpensive to test, whereas the `span` predicate is not. An operational predicate is considered to be an atomic operation and its definition will not be inspected by Zenith.

- `State_specific_pred` declarations specify the predicates that vary among states. Any predicate that is not specified as state-specific is assumed to be constant over all states.

The domain theory specifies the operators available in the domain. OTHELLO has one general operator, `move(Player,Square)`, indicating the player making the move and the square to which the player is moving. The domain theory also specifies the pre-images of expressions with respect to the OTHELLO operator. These pre-images are used by the goal regression transformation.

The domain theory specifies Zenith's goal for OTHELLO, which is a win for the Black player[2]. The theory specifies auxiliary predicates necessary for the definition of `win`, such as `span` and `line`. These predicates are in turn defined in terms of the operational predicates, the base-level observable attributes of an OTHELLO state.

It is important to note what the domain theory *does not* include. It contains no information about strategy or tactics, no information about how to win the game, and no knowledge about how to generate or select features. Though it has information on how the squares are connected, it has no special knowledge about corner, edge, C or X squares. It has no information about the board center. It has no concept of stability, mobility and vulnerability of pieces, or the notions of attack and defense. The domain theory contains essentially the basic rules given to a human player.

[2]Zenith always plays Black, although it can play White by using Negmax inversion [Knuth & Moore, 1975].

6.3 Experiments

Zenith's opponent is an expert OTHELLO-playing program called Wystan, provided by Jeff Clouse and Paul Utgoff. Wystan uses a sophisticated set of features and 4-ply search, so its moves are initially much better than Zenith's. Wystan's evaluation function is a weighted sum of feature values, the weights having been derived via TD from a large set of instances. Wystan uses its evaluation function to choose moves for approximately the first forty-six moves of the game, then switches to perfect search for the last fourteen moves.

Zenith's performance component makes a move using the method described in Section 5.2. Several choices were made in order to simplify the performance component:

- The performance component performs 1-ply search: it determines the descendants of the current state and chooses the most preferred successor state using its preference predicate.
- Zenith does not use perfect search in the OTHELLO end-game. The same preference predicate is used throughout the entire game.

The first choice was made primarily to reduce the amount of time spent by the performance element. The second choice was made to keep the performance element domain independent. We did not want to bias the performance component toward any particular two-player game, so we did not assume that the end-game could be searched, or even identified. Hence, Zenith employs a single search method over the entire state space.

After a game, Zenith's critic extracts preference pairs from the moves. The preference pairs, called instances, are added to an instance base. In the experiments below, the capacity of the instance base was limited to 3000 to keep manageable the amount of time taken in calculating feature values; however, this limit was rarely met within 10 cycles because each game added roughly 250 instances. Zenith trains on-line: in every cycle, after instances are added, Zenith trains a new evaluation function on the current instance set, to be used in the following cycle.

At the end of every cycle the *instance accuracy* of the preference predicate is tested and reported. The instance accuracy is the classification accuracy of the preference predicate, measured as follows. The instance set is partitioned randomly into a training set (2/3 of the instances) and a testing set (the remaining 1/3). The preference predicate is trained on the training set, then its accuracy recorded against the testing set. Accuracy is measured by applying the preference predicate to each instance and checking for strict (consistent) preference: for a given preference pair $\langle s_a, s_b \rangle$, the preference predicate has to classify $\langle s_a, s_b \rangle$ as a positive instance (so that $p(s_a, s_b)$ is true) and $\langle s_b, s_a \rangle$ as a negative instance (so that $p(s_b, s_a)$ is false). If an instance satisfies both tests, it is considered to be classified accurately. Accuracy is

determined by measuring the ratio of correctly classified test instances to total test instances.

In order to arrive at a final accuracy, Zenith creates and tests preference predicates repeatedly, as described in Section 5.5, until the average accuracy lies within a 2% confidence interval with 99% probability. At this point the average accuracy is reported.

Performance improvement is also measured by playing a set of ten games against an opponent after every cycle and counting the number of games won. Several choices could have been made for Zenith's OTHELLO opponent:

1. Zenith could play against Wystan. However, even with equivalent feature sets, Wystan is superior: it uses deeper search with its evaluation function, and performs exhaustive search in the end-game. Even with a better feature set than Wystan uses, Zenith could not beat Wystan under these conditions.

2. Zenith could play a version of Wystan that used Zenith's evaluation function. However, there are practical difficulties: Zenith's features are logical clauses in Prolog, and Wystan was hand-coded in C. Translating from Prolog to C is non-trivial.

3. A separate opponent could be created that used Zenith's framework but Wystan's evaluation function. The opponent would use 1-ply search and no exhaustive search in the end-game, but would use Wystan's expert hand-coded features.

Because of the disadvantages of the first two choices, the third choice was made. A separate opponent called ORFEO (ORacle FEature Opponent) is used to test Zenith's performance. ORFEO performs 1-ply search using Wystan's features but performs no end-game search. Thus, ORFEO has Zenith's framework but uses a handcoded expert feature set.

Because neither opponent learns while it plays, Zenith's games against ORFEO are artificially varied slightly. There are several ways that this could be done, such as selecting moves probabilistically. Following the practice of Lee and Mahajan (1988), a portion of the opening game (the first five moves made by each player) is randomized to provide variety in the games. Without this small amount of randomization, the opponents might play the same set of moves repeatedly.

6.3.1 Linear Threshold Unit

The first tests used a linear threshold unit (LTU) as a preference predicate. The LTU was trained using the Recursive Least Squares (RLS) training rule[3] [Young, 1984]. Training and testing of preference pairs is described in Section 2.4.2. For a

[3]The code for LTU calculations was written by James Callan.

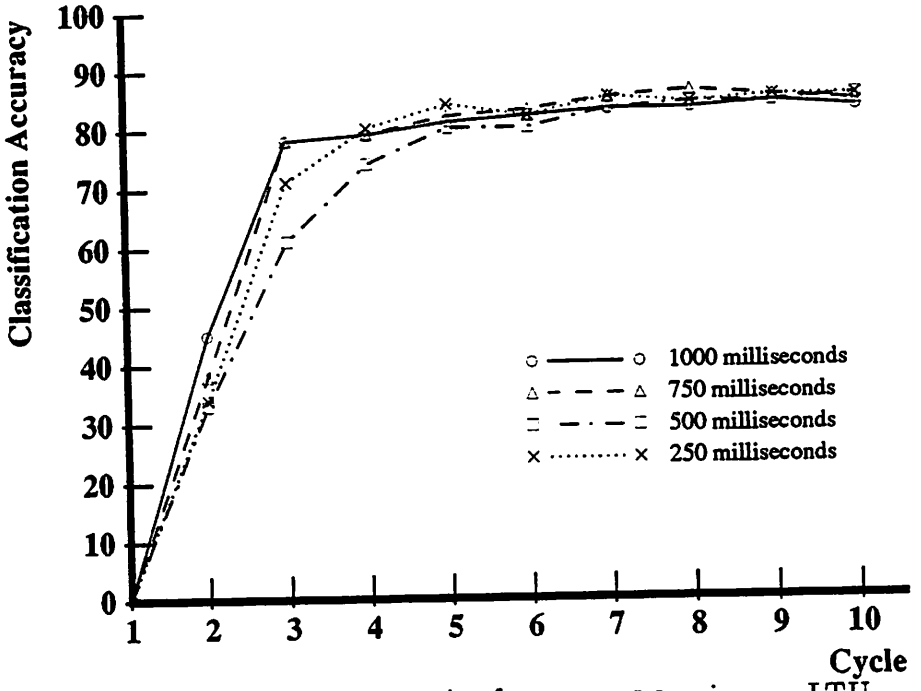


Figure 6.3 Classification accuracies for OTHELLO using an LTU.

given preference pair, the LTU was trained on the difference vectors of the preference pair states. First the difference vector was presented with the desired output value of +1, then the negated delta vector was presented with the desired output value of -1. The LTU was used to classify a new pair of states by calculating the feature delta vector of the states and presenting the it to the LTU. For a positive output, the first state is preferred to the second; for a negative output, the second is preferred to the first.

The presentations of preference pairs are symmetrical with respect to the hyperplane learned by the LTU: if one feature vector should be classified as positive, its negation should be classified as negative. Because of this symmetry, an origin restriction was imposed on the LTU such that the learned hyperplane had to pass through the origin. This restriction forced every non-zero negated vector to lie on the opposite side of the hyperplane from its positive instance vector.

The effect of feature generation can be measured directly as the change in classification accuracy of the LTU. Four experiments were run varying the evaluation function cost limit as an independent parameter. Each of the experiments was allowed to run for ten cycles, at which point the accuracies had stabilized. Figure 6.3 shows classification accuracy as a function of cycle number. The classification accuracies shown in this figure are within ± 2 of the actual accuracies with 99% probability. In these runs, the accuracies rose from zero in the first cycle to a final value between 83% and 86%. As the evaluation function cost limit was raised, the corresponding accuracy attained by Zenith rose only slightly.

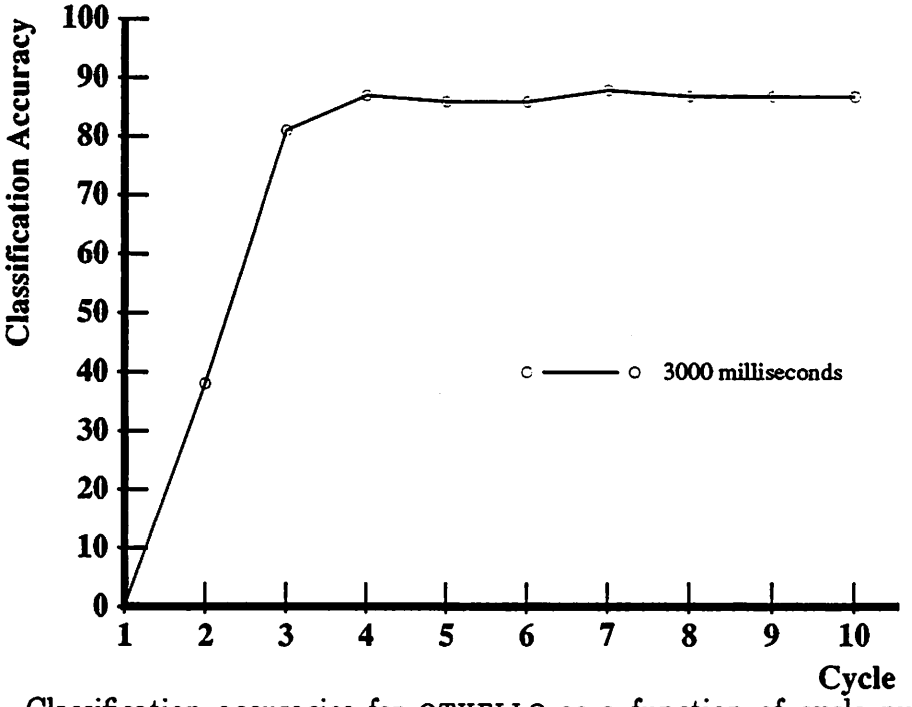


Figure 6.4 Classification accuracies for OTHELLO as a function of cycle number, allowing three second state evaluations.

To test the effect of further increase in the evaluation function cost limit, another experiment was performed in which three seconds (3000 milliseconds) were allowed for each evaluation. The graph of classification accuracy as a function of cycle number is shown in Figure 6.4. The highest accuracy achieved in this run, 87%, is not substantially higher than the accuracies in Figure 6.3, indicating that there may be an accuracy ceiling above which Zenith's feature generation, trained with an LTU, cannot rise.

Tables 6.1 and 6.2 show data on feature generation in these experiments. Table 6.1 shows the total number of features generated by Zenith in the experiments with an LTU. The second column shows the total number of features generated in the run, and the third column shows the number of features in the active set at the end of each run. The last column shows the highest instance accuracy attained in the run. Table 6.1 demonstrates that Zenith does not generate a prohibitive number of features. Indeed, the number of features generated increases sub-linearly in the evaluation function time limit. Because the accuracy figures shown in the table are within ± 2 of the true accuracies, they may be considered virtually identical to each other.

The final size of the active feature set, shown in Table 6.1, does not increase monotonically with the time limit. This may seem counterintuitive because a greater time limit should imply that more features could be used. However, Zenith develops a feature differently depending on whether it is considered expensive, and the definition of expensive is a function of the time limit. Hence, given higher time limits, Zenith

Table 6.1 Number of features generated for OTHELLO (LTU)

Time limit (msecs)	Total generated	Active at end of run	Accuracy attained
250	86	6	85%
500	119	16	85%
750	138	10	86%
1000	98	5	84%
3000	228	14	88%

Table 6.2 Transformation firing data for OTHELLO (LTU)

Transformation	Class	Time Limit				
		250	500	750	1000	3000
split-arith-comp	Decomposition	2	2	2	2	2
split-arith-calc	Decomposition	0	0	0	0	0
split-conjunction	Decomposition	9	4	9	3	11
remove-negation	Decomposition	6	7	6	6	8
split-disjunction	Specialization	6	10	11	19	21
expand-to-base-case	Specialization	18	20	25	7	15
variable-specialize	Specialization	3	5	2	1	3
remove-LC-term	Abstraction	11	6	13	4	7
remove-variable	Abstraction	18	23	15	16	16
regress-formula	Goal regression	12	41	54	39	144

may end up with several expensive features rather than many inexpensive features, if the expensive features attain equal or better accuracy. However, there may be benefits to reducing the cost of expensive features so that more features can be admitted into the active set. This is an area for future study.

Table 6.2 shows the number of times each transformation was used. After each transformation is the total number of times it fired in each run. A "firing" is the generation of a new feature by a transformation. Several observations can be made from this table. Split-arith-calc was not used at all in this domain because it decomposes arithmetic calculations, and the domain theory of OTHELLO performs no arithmetic calculations. Other than that, all transformations were used multiple times in a run.

Performance improvement was also measured at the end of every cycle by having Zenith play against ORFEO. Figure 6.5 shows the results for these games, with the number of games won as a function of cycle number. Because instance classification accuracy improves with the cycle number, Figure 6.5 demonstrates that performance generally improves with classification accuracy; however, the performance is erratic. This seems to be due to the feedback between performance and learning with on-

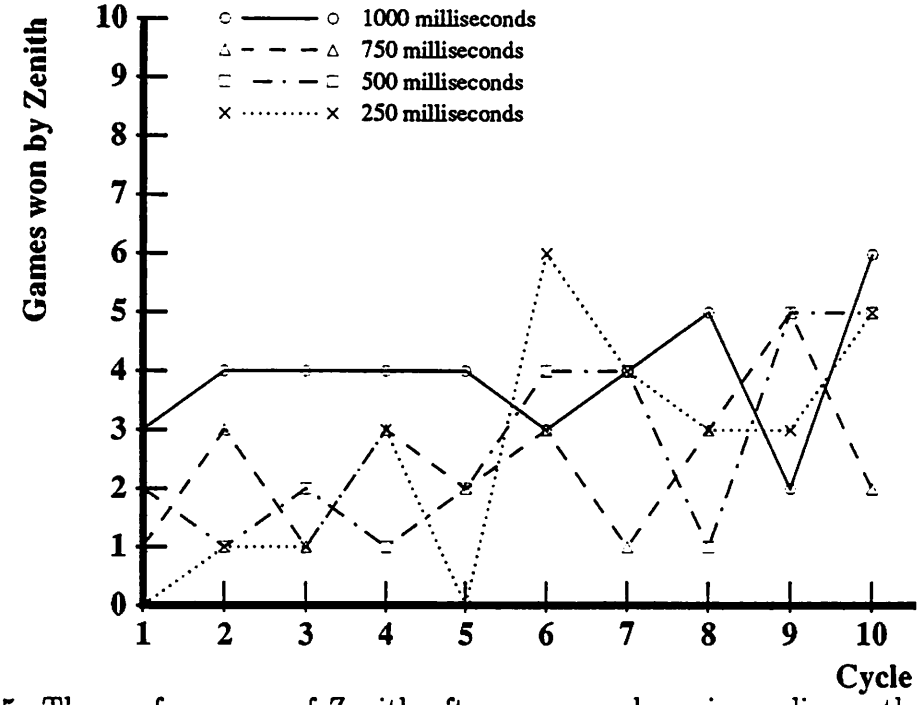


Figure 6.5 The performance of Zenith after every cycle, using a linear threshold unit. Each point represents the number of games out of 10 won by Zenith against the ORFEO opponent.

---

line training. As the evaluation function changes, it causes the problem solver to explore different parts of the instance space, which in turn changes the set of training instances. This phenomenon was observed and has been labeled *temporal crosstalk* by Jacobs (1990). A learning system suffering from temporal crosstalk appears to perform poorly or erratically because it is led continually into new regions of the problem space for which its learned responses may be ineffective or even disastrous.

Donald Mitchell (1984) mentions a similar phenomenon related to novice versus expert learning:

> "At first it may seem that a program using the expert weights should play a better game than one using the novice weights. This assumption is not correct. The expert program should play its best game against another expert because it is using the features and weights which were determined to be more effective for evaluating experts' positions. These weights were less effective than the novices' weights for evaluating novices' positions. The weights which should be more effective for any game are the ones based on the data from subjects whose skill level is comparable to the opponent's." [Mitchell, 1984, page 81]

Within a learning system, this improvement from novice toward expert constitutes a form of concept drift. To investigate this phenomenon, Zenith was run with a static set of expert training instances, and on-line training was disabled. The instances

Table 6.3 Games won against ORFEO by Zenith when trained on expert instances.

Cycle number	1	2	3	4	5	6	7	8	9	10
Games won	1	2	0	4	1	6	5	6	5	2

used were 3000 preference pairs extracted from OTHELLO tournament games played by humans. The results are shown in Table 6.3.

Because of the feedback from performance to the training instance set, the relationship between evaluation function accuracy and performance is complex. Investigation of this relationship is the subject of future work, and is discussed further in Section 8.2.3.

6.3.2 Univariate Decision Trees (C4.5)

In order to investigate the sensitivity of feature generation to the concept form, the second set of OTHELLO experiments used the C4.5 program [Quinlan, 1992] as a concept learner. C4.5 is a learning program that induces a decision tree from a set of examples. A decision tree is a tree with a test at each internal node. Each test compares a feature value to a set of fixed values, if the feature's possible values are discrete, or to a numeric threshold, if the feature's values are continuous. Each test has two or more outcomes, each outcome leading to another node. The leaf nodes of a decision tree are class names. The tree is used to classify an instance by starting at the root of the tree, evaluating the test, and branching to a new node based on the outcome. This is done repeatedly until a leaf node is reached, at which point the class name of the leaf node determines the classification of the instance.

C4.5 is based on ID3 [Quinlan, 1986] but has several extensions, the most important of which is that C4.5 can accommodate continuous-valued features by creating range inclusion tests at nodes. Thus it was possible to use C4.5 with the numeric-valued features that Zenith creates.

C4.5 produced preference predicates in the form of two-class decision trees. For a given preference pair, C4.5 was trained on the delta vector as an example of the class "preferred", and trained on the negated delta vector as an example of the class "not-preferred." A new state pair was tested by classifying both its delta vector and its negated delta vector; only if both the delta vector and its negation were classified correctly was the preference assumed to hold.

Delta vectors were used with C4.5 because they express differences between preferred and non-preferred states. Decision trees created by C4.5 use univariate splits at each node. Because each node in a decision tree splits on only a single instance attribute, each attribute of an instance had to represent the difference in feature values of the two states.

C4.5 was used with the following settings:

- Windowing was disabled. Windowing is a feature whereby C4.5 creates a sample (a window) of instances from which an initial tree is constructed. The initial tree is then used to classify a selection of training instances not in the window, and a set of exceptions is added to the window which is then used to build the next tree. Because Zenith's training cycle does random sampling anyway, this feature was disabled, so that C4.5 created only a single decision tree using all the data it was given.
- The gain ratio criterion was used to select tests at nodes. *(default)*
- The pruning confidence level was set to 10%. *(default)*
- The minimum number of objects on either side of a threshold test was set to 2. *(default)*

A modified sequential backward selection method was used with C4.5. The sequential backward selection method works fairly well, but for n features it requires $O(n^2)$ applications of the concept learner. LTUs are inexpensive to train, so the $O(n^2)$ applications were not prohibitive; however, C4.5 is significantly more expensive. Sequential selection methods are expensive primarily because they apply the concept learner n times in order to determine which of the n features is least useful. However, in some cases the induced concept can be examined directly to determine the least useful feature. This was done with C4.5 by examining the induced trees and evaluating the use of features in the tree. The utility of a feature was measured by summing, for every internal node in the decision tree that tested the feature, the number of instances that reached that node. The resulting count thus measured how many instances used the feature in their classification. The feature with the lowest count was assumed to be the least useful feature, and was discarded. This reduced the entire selection process to $O(n)$ applications of the C4.5 algorithm. To the best of our knowledge, this method of evaluating the worth of a feature has not been used elsewhere.

Four experiments were run, varying the evaluation function cost limit as an independent parameter with the same values as with LTUs. Again, each of the experiments was allowed to run for ten cycles.

Classification accuracy is shown in Figure 6.6. In these runs, the accuracy rose from zero to a final value between 76% and 81%. These accuracies were generally slightly lower than the corresponding instance accuracies for LTUs. This is probably because univariate decision trees are more suitable for discrete symbolic attributes than for Zenith's continuous attributes. For continuous attributes, in addition to determining the best attribute on which to split, a univariate decision tree algorithm must find proper range values for the attribute.

The accuracies of the 1000 millisecond run lagged behind those of the other C4.5 runs, and exhibited a noticeable drop in the tenth cycle. This is due to the modified

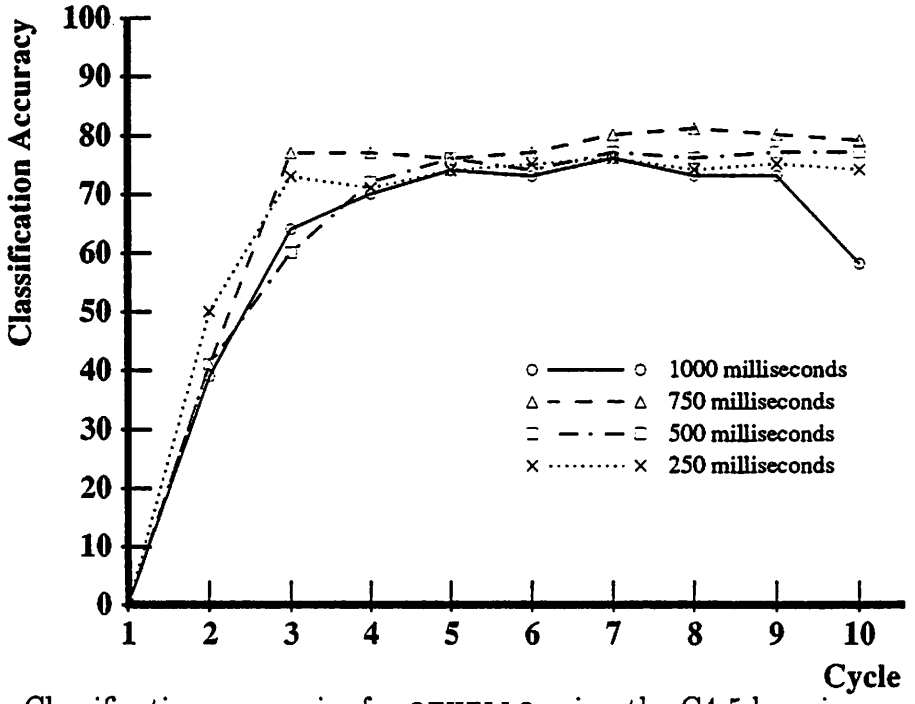


Figure 6.6 Classification accuracies for OTHELLO using the C4.5 learning program. True accuracies lie within a 2% confidence interval with 99% probability.

---

backward selection algorithm. Better features were available in the last cycle, but they were not used because the highest-ranked feature was very expensive. This had the effect of squeezing out other features, and resulted in only a single feature left in the active feature set at the end of the cycle. Section 8.2 discusses Zenith's requirements for feature selection in more detail.

As with LTUs, a final experiment was run in which the evaluation function cost limit was raised to three seconds (3000 milliseconds). The graph of classification accuracy as a function of cycle number is shown in Figure 6.7. The highest accuracy achieved in this run, 78%, is similar to the highest accuracies of Figure 6.3, but the number of features generated is smaller. This may be due to the more erratic feature selection method.

Tables 6.4 and 6.5 show data on feature generation for the C4.5 experiments. Table 6.4 shows, for each value of the evaluation function time limit, the total number of features generated, the number of features in the active set at the end of each run, and the accuracy attained. Again, the 1000 millisecond run had a lower accuracy than expected, and the run concluded with only a single active feature. Table 6.5 shows the number of times each transformation fired in each run. The transformation firing data is similar to the corresponding data for LTUs, except that fewer features were generated in the 750 and 1000 millisecond runs of C4.5. This is because, on average, there were fewer active features in the C4.5 runs than in the corresponding two LTU runs. Because there were fewer active features, there tended to be less development of the active features, which resulted in fewer features being generated altogether.

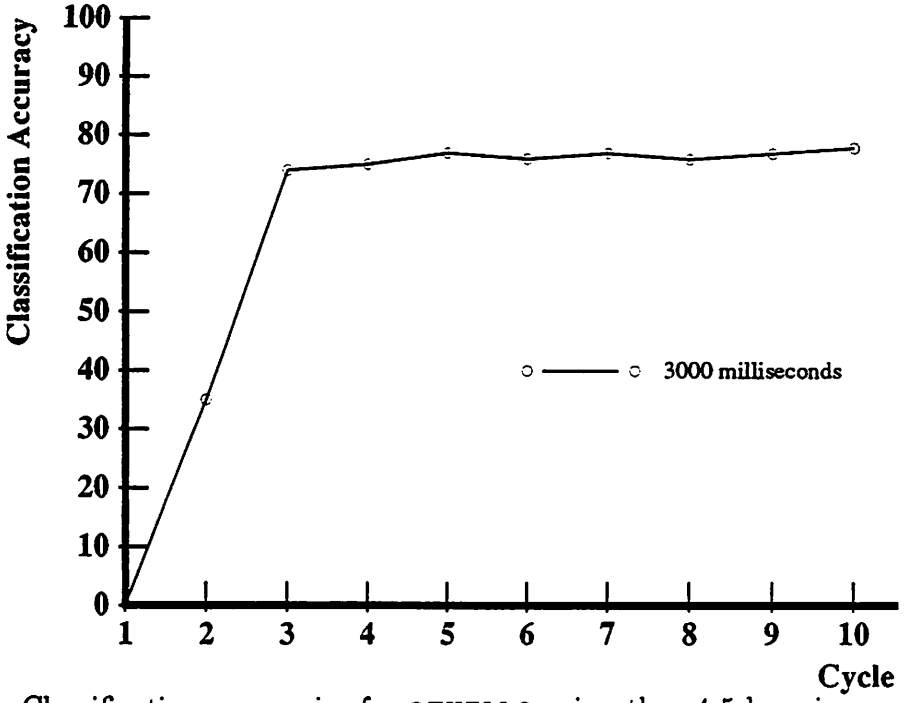


Figure 6.7 Classification accuracies for OTHELLO using the c4.5 learning program, allowing three second state evaluations.

Figure 6.8 shows the results for the 500 and 1000 millisecond evaluation functions.

6.4 Feature Generation in OTHELLO

In this section we discuss the features generated by Zenith and how they compare with known features from OTHELLO. Specifically, we discuss:

- known features that Zenith generated,
- novel features that Zenith generated, and
- known features that Zenith did not generate.

The features generated by Zenith discussed in this section appear in Appendix B.

Many of the known features for OTHELLO have been discussed and analyzed by Mitchell (1984). A more accessible description of the game and some of the important features is included in a journal article by Rosenbloom (1982). Lee and Mahajan (1988) also discuss OTHELLO features. Unfortunately, details of the features used by top-ranked OTHELLO programs are rarely published while the programs are still in competition; Kierulf's (1989) BRAND program is an exception.

Table 6.4 Number of features generated for OTHELLO (C4.5)

Time limit (msecs)	Total generated	Active at end of run	Accuracy attained
250	83	3	76%
500	97	7	77%
750	97	3	81%
1000	98	1	76%
3000	77	3	78%

Table 6.5 Transformation firing data for OTHELLO (C4.5)

Transformation	Class	Time Limit				
		250	500	750	1000	3000
split-arith-comp	Decomposition	2	2	2	2	2
split-arith-calc	Decomposition	0	0	0	0	0
split-conjunction	Decomposition	6	6	7	5	5
remove-negation	Decomposition	6	10	6	6	5
split-disjunction	Specialization	8	12	4	16	20
expand-to-base-case	Specialization	20	14	16	14	9
variable-specialize	Specialization	1	1	1	1	1
remove-LC-term	Abstraction	11	6	19	14	4
remove-variable	Abstraction	22	26	11	9	16
regress-formula	Goal regression	6	24	30	30	14

6.4.1 Generation of Known OTHELLO Features

Figure 6.9 illustrates Zenith's derivations of some of the known OTHELLO features, and two novel ones. The figure shows the features discussed in this section and in Section 6.4.2. Due to space considerations, only a subset of all generated features are shown; Tables 6.1 and 6.4 give information on the total number of features generated for the runs.

Zenith uses generated symbols for its features, such as f17; the feature names in the figure are descriptive names assigned by hand.

The domain theory for OTHELLO specifies the performance goal as `win(black)`, from which an initial feature is created that measures whether the black player has won in a state. This feature is useless for directing search because it only distinguishes end-game states in which Black has won from states in which Black has not, and there are relatively few such states. However, decomposing this initial feature yields features measuring the score for each player (Pieces), and the number of legal moves (Moves).

The Pieces feature counts the number of Black pieces on the board. Regressing the formula of Pieces through the OTHELLO move operator yields a Semi-stable Pieces

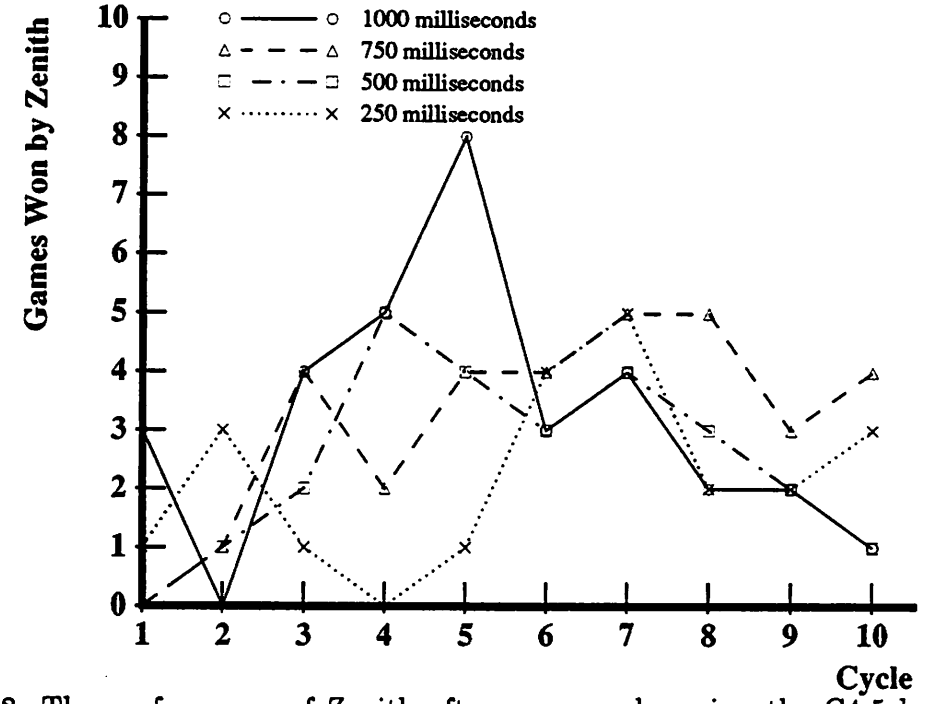


Figure 6.8 The performance of Zenith after every cycle, using the C4.5 learning algorithm. Each point represents the number of games out of 10 won by Zenith against the ORFEO opponent.

---

feature that counts the Black pieces in a state that are semi-stable. Semi-stable pieces are pieces that cannot be acquired by the opponent in a single move. From this feature, remove-negation generates the Axes feature, which measures the total potential for flipping opponent's pieces on the board. Axes measures, for each piece, the number of ways in which it can be flipped. A further abstraction of this, using remove-variable, yielded Semi-unstable Pieces, which counts the pieces that can be taken by the opponent on the next move.

The number of semi-stable pieces is a useful feature of a state to measure, but it is expensive to test every piece for semi-stability. The variable-specialize transformation specialized this expensive feature by looking for squares that are invariantly semi-stable; that is, by looking for squares that invariably satisfy the semi-stability property. Because of the geometry of the OTHELLO board, pieces on the four corner squares are invariantly stable: it is not possible to create a span through a corner square to flip a piece on it. Zenith's variable-specialize transformation found the corner squares, and several others, to be frequently stable. From these squares the variable-specialize transformation created a new feature, called Apparently Always Stable Squares (AASS). Because it includes squares other than the corners, AASS is not equivalent to Corner Squares, although AASS subsumes it.

The AASS feature proved to be both useful and inexpensive. Goal regression was then applied to it, yielding a feature that counted the number of pieces that could be

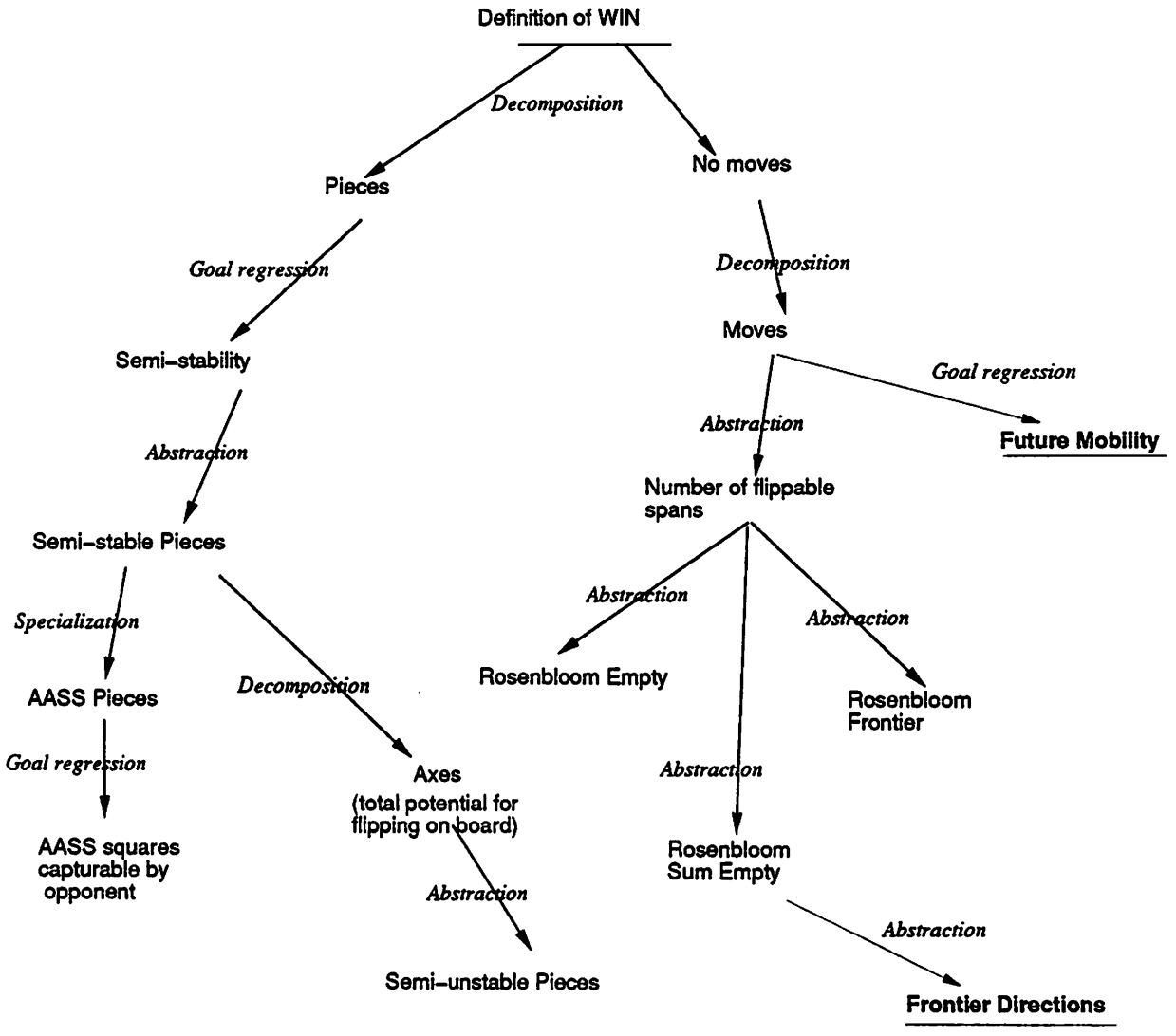


Figure 6.9 OTHELLO features generated by Zenith

taken on the next move. This feature is similar to commonly known features such as X Next to Empty Corner and C Next to Empty Corner.

Zenith created a number of mobility features, shown on the right-hand side of Figure 6.9. From the initial feature, Zenith created a feature that measured the number of moves available (Moves). By expanding the definition of the OTHELLO move, a new feature was created that counted the number of squares involved in each move. By repeatedly abstracting this feature, Zenith created a number of simpler, less costly mobility features including three used by Rosenbloom (1982) (Rosenbloom Empty, Rosenbloom Sum Empty and Rosenbloom Frontier). However, Rosenbloom's features measured the *difference* in mobility between two players, whereas Zenith's features measured the value for each player individually.

6.4.2 Generation of Novel Othello Features

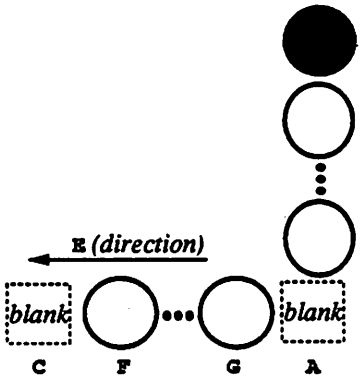
Figure 6.9 also shows several novel features generated by Zenith that, to the best of our knowledge, have not been published or discovered elsewhere. By applying the regress-formula transformation to the Moves feature shown on the right-hand side of the figure, Zenith generated several *future mobility* features that detected moves that could become available in the next state.

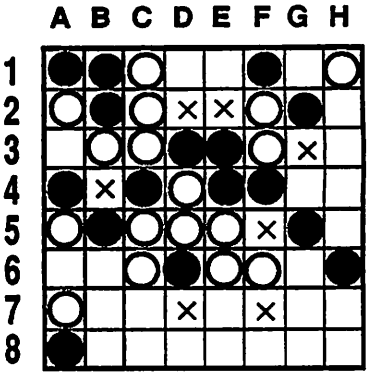
One of these features is illustrated in Figure 6.10. A pattern illustrating the configuration matched by the feature is shown at the left side of Figure 6.10. Intuitively, the feature measures the number of ways in which a move can be created by taking a blank square. At the right side of Figure 6.10 is a sample board to illustrate the feature. Each of the X's on the board is a blank square bound to square A in the pattern (square F5 occurs twice). The variable values generated by the feature in the sample state are:

```
[c3,ne,d2,b4]
[c3,sw,b4,d2]
[e6,ne,f5,d7]
[f2,se,g3,e1]
[f6,s,f7,f5]
[e6,sw,d7,f5]
[f6,n,f5,f7]
[f3,nw,e2,g4]
```

so the value of the feature in this state is 8. Figure 6.10 shows the actual text of the feature.

Another novel feature created by Zenith is called Frontier Directions. This feature measures the *number of directions* in which a frontier exists for a player. Its value ranges from zero to eight. Frontier Directions is illustrated in Figure 6.11, in which the black player has a Frontier Directions value of 6. This feature is a very abstract measure of mobility in that it ignores not only the existence of a bracketing piece but






```
f13(H) :- count([G,E,A,C],(A=B,
                           square(C),
                           blank(C),
                           opponent(black,D),
                           D\==black,
                           direction(E),
                           neighbor(C,E,F),
                           span(F,G,E,D),
                           neighbor(G,E,A),
                           legal_move(A,black),
                           B\==C),
                  H).
```

Figure 6.10 The Future Mobility feature. At top left is the pattern being matched by the feature. At top right is an illustration of the feature applied to an OTHELLO state. At bottom is Zenith's definition of the Future Mobility feature for the Black player.

---

also the number of moves available in the same direction on the board, no matter where they are. It is surprising that such an abstract feature proved useful to Zenith, but it survived in competition with other mobility features. The feature is shown in Figure 6.12.

6.4.3 Known Othello Features Not Generated

Zenith could not generate some of the known features for OTHELLO. There are several reasons that Zenith could not generate all of the known features. Some features are based on a property of OTHELLO that Zenith's transformations are unable to generate. Other features require a form that Zenith is unable to produce. Section 8.2

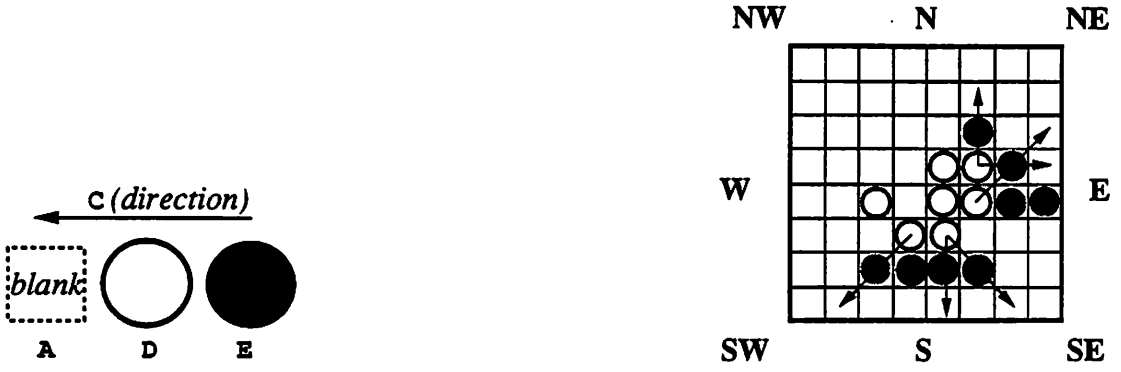


Figure 6.11 The Frontier Directions feature. At left, the pattern being matched by the feature. At right, an illustration of the feature applied to an OTHELLO state. White has neighboring black pieces in the directions {SW,S,SE,E,NE,N}, so the feature value in this state is 6.

```
f73(F) :- count([C], (square(A),
                      blank(A),
                      opponent(black,B),
                      direction(C),
                      neighbor(A,C,D),
                      owns(B,D),
                      neighbor(D,C,E),
                      owns(black,E)),
                      F).
```

Figure 6.12 The Frontier Directions feature for the White player.

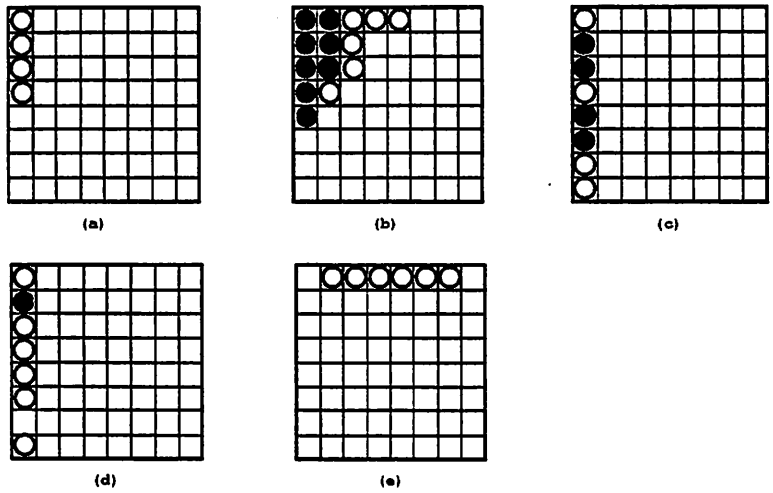


Figure 6.13 Examples of stable and unstable piece configurations: (a) White pieces are stable because they are anchored at the corner (b) All the black pieces are stable. (c) All pieces shown are stable. (d) Only white corner pieces are stable because Black can flip others. (e) None of white's pieces are stable.

discusses improvements that could be made to Zenith to overcome some of these limitations.

The Stability Property. Stability is a property of a piece such that the piece cannot be flipped by any future move. The corner squares are unconditionally stable in that any piece placed in a corner is stable regardless of the occupants of the rest of the board. Other pieces can become stable depending on their neighboring squares. Because corner squares are unconditionally stable, stable regions emanating from the corner often can be created.

Zenith can generate a *semi-stable* feature that recognizes pieces that cannot be retaken with a single move. Zenith can also generate a feature labeled Apparently Always Semi-Stable, which is an empirical approximation of unconditional stability; that is, stability of individual squares without regard to surrounding pieces.

Stability of interior discs is much more expensive to test because it requires testing each square for stability along its axes. Even using efficient handcoded procedures, such stability tests generally take too long to be worthwhile[4]. Instead, most programs use features that check for specific configurations of stable pieces, usually along the edge. Examples of such features are Stable Pieces, Stable Edge Pieces, Nonstable Edge Pieces, Rosenbloom Edge Score and Edge Anchor.

[4]Rosenbloom's (1982) IAGO program used an internal stability feature, but Rosenbloom noted that because of the approximation required, it was most useful in games in which IAGO already had an advantage.

Zenith is unable to generate features for stability for two reasons. First, Zenith does not reason about move *sequences*. Although its regress-formula transformation can regress a formula through an operator application, it cannot derive properties that are true of indefinitely long sequences of operators. Second, its variable-specialize transformation looks only for individual domain elements that satisfy a property. Variable-specialize does not examine *combinations* of domain elements that, taken together, satisfy a property. For example, a piece on a C square next to an empty corner is unstable, but a piece on a C square next to a corner that is occupied by the same color piece is stable. Figure 6.13 shows some examples of stable and unstable configurations, illustrating the complexity of the stability property.

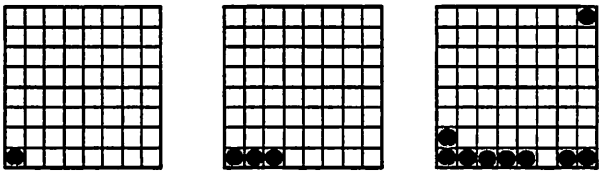
Edge Squares. Pieces on edge squares are important in OTHELLO because they can only be flipped by a move along the edge, so they tend to be more stable than interior squares. They are also important because moves along the edge can be used to take the corner, and because large stable regions can be constructed from a corner and its adjacent edge(s).

Because of the importance of edge squares, many OTHELLO programs employ features involving them. Some features simply count edge pieces in specific configurations (Semi-Stable Edge Pieces, Stable Edge Pieces, NonStable Edge Pieces), and some weight the squares in different ways (Rosenbloom Edge Score, Edge Anchor). However, although edge squares are known to be important, players disagree about whether it is good or bad to own edge pieces [Mitchell, 1984, page 41]; Kierulf (1989) invented a feature that penalized edge configurations that he considered to be dangerous.

Zenith does not create features that employ edge squares, because Zenith's transformations are incapable of recognizing edge squares as a special class. Zenith could create a feature for edge squares by a change to its variable-specialize transformation. Currently this transformation checks a property against instances and accepts only domain elements that are completely invariant with respect to the property (see Section 5.4.4). This test could be weakened so that elements that *usually* satisfy the property are accepted. This change would allow a feature to generate edge squares as regions of high semi-stability or low mobility. This idea is further explored in Chapter 8.

Explicitly Weighted Features. Several features invented for OTHELLO employ explicit weights. Weights are designed into a feature when a non-linear combination of pattern elements is desired. For example, Corner Points is a feature for simple cases of corner and edge stability. A corner alone is worth 3 points, and each neighboring C square of the same color is worth 5 more points; the feature returns the sum of the points.

Zenith does not create features with incorporated weights. That is, it does not create features that assign weights to portions of a formula. However, if a concept learner (such as an LTU) is used that assigns weights to the features, it is possible to simulate the effect of an internally-weighted general feature with a set of more specific features. For example, consider the Corner Points feature shown in Figure 6.14 with three OTHELLO states. Two other features, CP1 and CP2, are also shown. CP1



Corner Points:	3	13	39
CP1:	1	1	3
CP2:	0	2	6
3xCP1 + 5xCP2:	3	13	39

Figure 6.14 An internally weighted feature simulated by two unweighted features

counts the number of corners occupied by Black, and CP2 counts the number of Black edge spans comprising two or more pieces anchored at the corner. When assigned the weights 3 and 5, CP1 and CP2 together produce the same value that Corner Points produces. If the evaluation function assigns weights to features then CP1 and CP2 are sufficient to replace Corner Points.

The disadvantage of employing specialized features is that they duplicate computation effort. In the example above, eight new features must be introduced, one for every span of length n, and much of the work of matching a short span is duplicated in matching a long span. Furthermore, such features must co-exist in the same feature set, and the longer spans may have low utility because they occur less often than the shorter spans that they spatially subsume.

Specialization of Number. Zenith's variable-specialize transformation can specialize the *identities* of domain elements that satisfy a property. Some features are specializations of the *number* of domain elements that satisfy a property. For example, Weighted Sum Empty is a mobility feature like Rosenbloom Sum Empty that weights the number of empty neighbors to determine the feature value: one neighboring disc assigns 6 to that piece's mobility score, two neighboring discs assigns 11 to the score, three discs assigns 15. Four discs assigns 18. Five discs assigns 20, and six and seven discs contribute 21. Thus, Weighted Sum Empty uses specializations of the number of empty neighboring squares that a piece has.

Similarly, the Solitaires feature counts the number of pieces for a player that have only one neighbor of the same color (solitaire discs are thought to be disadvantageous to own because they are "stranded" from the center of play). Thus, Solitaire specializes the number of neighbors that a piece has.

Zenith can create a feature that measures the number of pieces owned by the player, and a feature that measures the number of pieces that have *any number* of neighbors. However, Zenith cannot create a feature that limits the number of neighbors of a piece. Zenith's feature formalism allows it to count the number of

values satisfying a variable, but not to constrain the number of values. There is no straightforward extension to Zenith's formalism that would allow this specialization.

6.4.4 Summary

Through the application of its transformations, Zenith is able to create many known OTHELLO features, as well as several novel features. The features generated are useful for both LTUs and univariate decision trees, although accuracy is higher with LTUs. Because of the expense associated with the univariate decision tree algorithm, a different, less careful feature selection method was used; this choice was responsible for erratic feature development. Section 8.2.1 discusses the requirements of Zenith's feature selection process. There are also groups of OTHELLO features that Zenith is currently unable to generate, although extensions to Zenith's transformations, discussed in Section 8.2, could overcome some of these limitations.

CHAPTER 7

TELECOMMUNICATIONS NETWORK MANAGEMENT

The second domain to which Zenith was applied is that of telecommunications network management (TNM), which concerns the control of traffic in a circuit-switched network. The goal of network management is to maximize the performance of the network, according to some metric. TNM was chosen as a second domain to determine whether the theory of feature generation could be applied to "real world" problems. The differences between OTHELLO and TNM will be discussed with the presentation of the domain in the next section.

7.1 Domain Description

A telecommunications network consists of a set of switches connected by trunk groups. A trunk group is a group of individual lines (trunks), each of which can accommodate a single call at one time. A simple telecommunications network, called ILS-10, is shown in Figure 7.1. The circles in the figure represent switches, the lines between them represent trunk groups, and a number next to a trunk group is the group's capacity, the number of lines in the trunk group.

The purpose of a telecommunications network is to allow calls between switches so that people with phones connected to one switch can talk to people with phones connected to another switch. The network enables calls by establishing paths from origins to destinations. A call is generated outside the network with a given origin and destination. The call starts at its originating switch. The switch allocates a trunk line for the call, based on its destination, and passes the call over the trunk line to the next switch. This next switch repeats the process; it examines the call's destination, allocates a trunk line for the call, then sends the call onward over the line. When the call reaches its destination, it is said to have *completed*[1]; there is now a trunk path from its origin to its destination through which communication can occur.

[1]This terminology may be confusing. A completed call is one whose routing has been completed, but from the point of view of the caller, it has just begun.

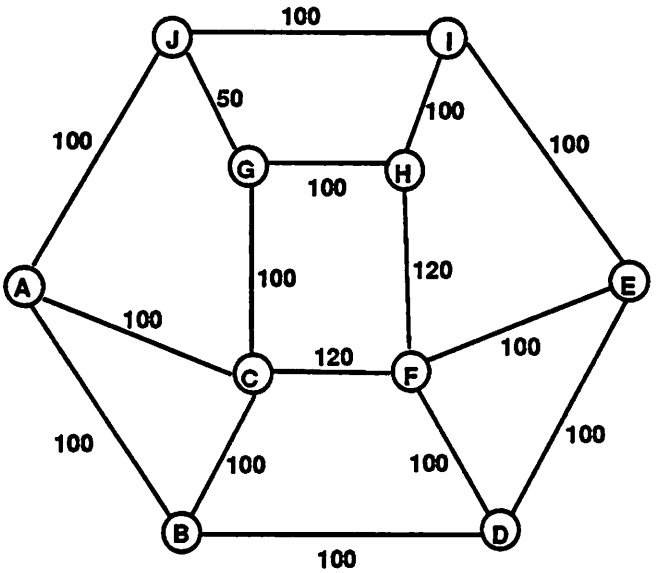


Figure 7.1 The ILS-10 telecommunications network.

For example, in the ILS-10 network a call from switch J to switch D might be routed along the path $[J, I, E, D]$. The trunk lines remain allocated until the call terminates. If at some point in the routing process a switch cannot route the call, then the call is said to have *failed* at the switch, and the call is dropped completely from all trunk lines. No failure recovery is attempted.

7.1.1 Routing Calls

The most important part of the process is how each switch decides to route a call, that is, how each switch decides to which switch the call should next pass. Each switch has a *routing table* which specifies for every destination an ordered list of trunk groups. When a call arrives at the switch, the table is consulted, and the first trunk group in the list is tried. If the trunk group is full or the connected switch is down, the switch tries the next trunk group in the routing table. When the routing table entries are exhausted, the call fails. The topology of the network and the routing tables are designed very carefully to minimize the possibility of cycling in call routing.

Several points should be emphasized about the routing process. Each switch decides only what the next step in a call's path should be; it does not decide what path the call should take. Each switch must make this decision based solely on the call's destination because no other information is passed along. For example, a switch cannot determine which other switches may already have routed the call, so it cannot detect cycles in the routing, nor can it detect that the call is taking an overly circuitous path to its destination. Also, the lifetime of a call cannot be known; once

a call has completed, it stays in place, occupying trunk lines, as long as neither end breaks the connection.

7.1.2 Network Controls

Because trunk groups have finite capacity, traffic patterns change, and because network components sometimes fail, a network may become congested and need to have its routing behavior changed. The routing tables are generally considered fixed, so rather than change them, the routing behavior is modified by the placement of *controls*. A control makes a specific change to a single switch's routing behavior. Examples of controls are:

- The Destination Re-Route (DRR) control, which overrides a routing table and forces a switch to re-route to another switch all calls that are going to a given destination. This is useful, for example, when a switch is known to be overloaded with calls; a DRR can be applied to adjacent switches to route traffic around to other switches.
- The Cancel-To (CANT) control, which cancels some portion of the traffic that would be routed onto a given trunk group. This is useful when, for example, a trunk group is damaged and traffic that would go through it cannot be routed any other way.
- The Code Block (CB) control, which forces a switch to cancel (block) some portion of traffic going to a given destination. This is useful, for example, when a switch is inoperative or overloaded; the network manager can place a Code Blocks on other switches in order to cancel some portion of the traffic destined for the inoperative or overloaded switch.

These are typical of the kinds of controls available. Most controls take an extra argument, the percentage of calls to which the control applies. For example, a CANT of 50% specifies that only 50% of the calls to which it applies should be canceled, and the remainder will be processed exactly as if the control did not exist.

Figure 7.2 shows an example of the DRR control. In the uncontrolled network at left, a call is routed from switch J to switch D using the standard routing tables. The resulting path is $[J, I, E, D]$. In the network on the right, the control DRR(I,D,TIH,100%) has been applied to the network, causing all calls arriving at switch I destined for switch D to be routed onto trunk group TIH (the trunk group that connects switch I to switch H). This causes the same call to take the path $[J, I, H, F, D]$. Such a control might be useful if switch E were inoperational or trunk group TEI were congested.

The goal of telecommunications network management is to place controls to maximize some function of the network's behavior. Typically the function is fairly simple, like the number of completed calls divided by the number of calls attempted.

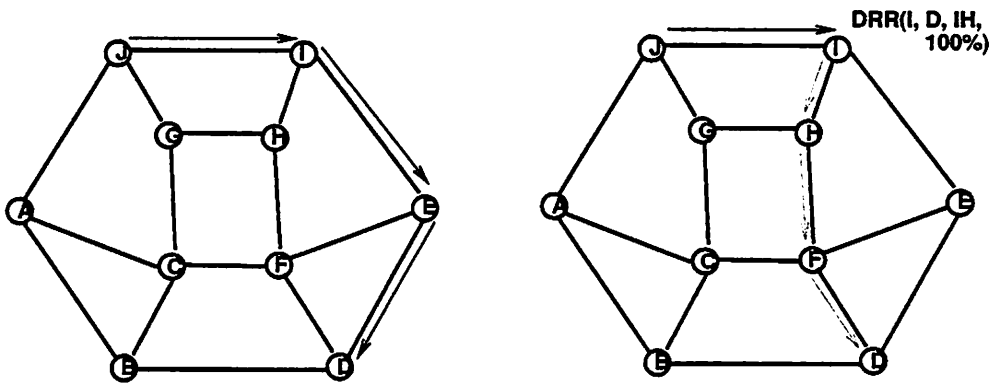


Figure 7.2 An example of the DRR control.

Existing problem-solving systems first try to diagnose the problem (e.g., as being due to equipment failure or unusually high traffic into a single node) before suggesting plans to remedy it.

7.1.3 Simplifying Assumptions

There are several simplifying assumptions made by existing systems for telecommunications network management [Silver, Frawley, Iba, Vittal & Bradford, 1990a; Silver, Vittal, Frawley, Iba & Bradford, 1990b] in order to simplify it as a domain for machine learning research:

- It is not always possible or desirable to maximize the evaluation function, since doing so may involve imposing and removing controls so quickly that true network performance is obscured by the traffic transients created by the controls. More typically, network controls are imposed only when the network evaluation functions drops below a threshold for a certain length of time.

- Any network with wildly varying traffic patterns can defy improvement, simply because of the delay between diagnosing the problem and imposing the controls. Therefore, it is commonly assumed that traffic patterns will remain "reasonably stable" for the amount of time it takes to diagnose a problem and impose controls. This assumption is not always satisfied in real-world networks, but humans usually also fail when it is violated.

- A network that is heavily loaded (operating at 95% capacity or more) can experience unavoidable call failures simply from slight variations in traffic. Therefore, it is assumed that the network is operating within some comfortable margin of its capacity.

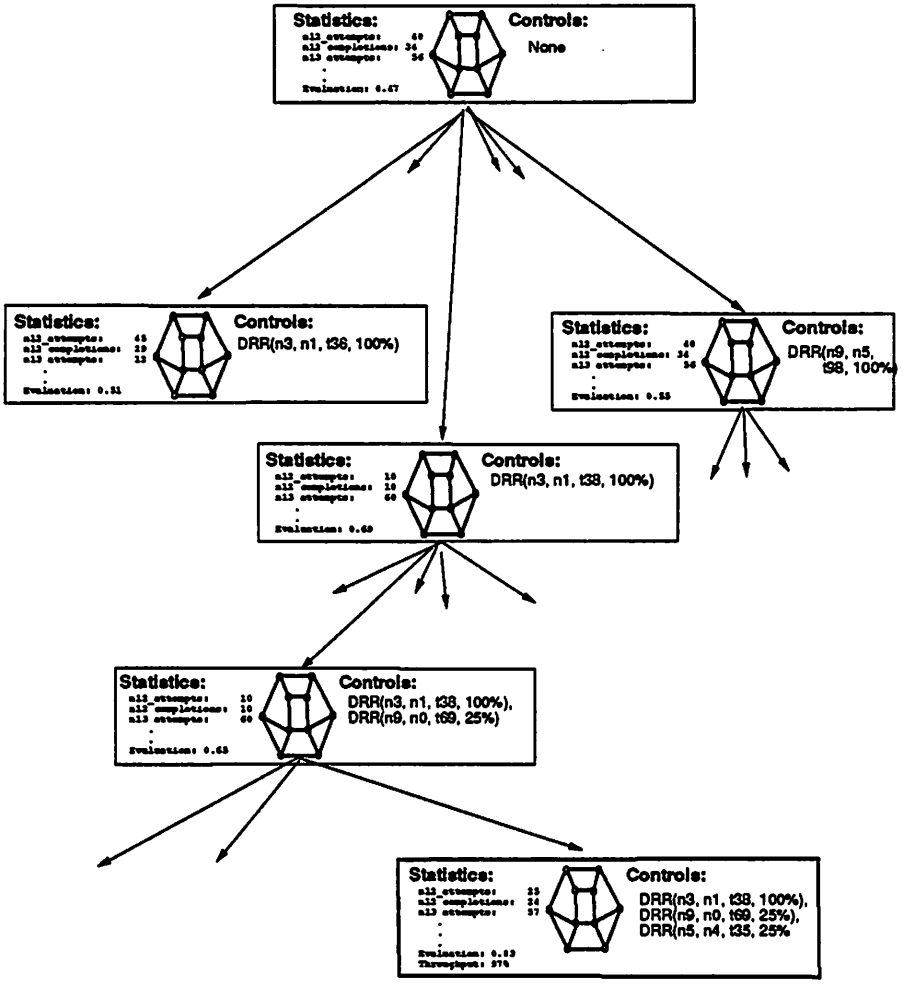


Figure 7.3 State-space search in the domain of TNM.

Using these assumptions, existing problem solving systems coupled with machine learning techniques (Silver, et al. 1990a,Silver, et al. 1990b) are able to diagnose network problems and improve call completion averages. The assumptions used in these systems were adopted by Zenith.

7.1.4 State-space Search in TNM

As with OTHELLO, Zenith performs state-space search in the TNM domain. A state comprises:

- a network description,
- a set of controls that have been placed on the network, and
- a set of statistics.

Figure 7.3 depicts state-space search in TNM. The initial state in TNM is the uncontrolled network. An operator application is the addition of a control to a network. Controls are not removed in the course of the search. The network statistics are local measures of traffic flow, such as the number of calls received by a switch, the number that were routed out of the switch, the number that successfully reached their destination, and the total number of calls carried by a trunk group. These numbers are accumulated over simulated five-minute intervals.

The goal is to raise the throughput of a network above a threshold by individual moves. Viewed as a state-space search, a move is the application of a control to the network, a control being a local policy change of the way in which calls are routed.

Zenith's TNM problem solver adopts a number of heuristics from MacLearn, [Iba, 1988], a macro-learning system that performs state-space search in TNM. These heuristics reduce the branching factor of search by removing from consideration certain controls.

- A control with a given destination will only be placed on a switch if that switch receives traffic for the destination. This rule removes from consideration controls that will not affect traffic.

- A control with a given destination will only be placed on a switch if there is an overflowing trunk group somewhere between the switch and the destination. This rule removes from consideration controls that will not alleviate congested trunk groups.

As with Zenith, MacLearn's problem solver must estimate the effects of a control on a telecommunications network. MacLearn uses a traffic flow model called FLOSIM [Iba, 1988] that incorporates simplifying assumptions about traffic flow in order to determine traffic statistics after a control has been applied. In contrast, Zenith's simulator performs a call-by-call simulation of the traffic flows using a reduced traffic volume.

7.2 The TNM Problem Specification

The problem specification (also called the domain theory) for TNM is given in Appendix C. The complete problem specification comprises several files.

One file contains the domain theory proper. The TNM problem specification includes a description of the DRR (destination re-route) and CB (Code Block) operators, as well as the pre-images of the operators. As with OTHELLO, the operators are expressed in two ways because they are not invertible. The specification file also contains meta-information about the modes, determinacy, operationality, and state-dependence of predicates used in the domain theory.

The forward operator descriptions indicate the effects of an operator applied to a state; that is, given a network state and a new control to be applied to the network, they describe how the network changes as a result. In order to determine this, the forward operator performs a simple call-by-call simulation of the network including the new control, whose result is a complete set of network statistics for the new state. This simulation, and thus TNM's forward operator, is relatively expensive; it requires approximately one-half to two seconds for the simulator to calculate the statistics for a new network state.

The backward operator descriptions specify the conditions that exist in order for an operator to have a certain effect. Specifying the pre-images of operators in TNM is problematic. Though the forward operator definitions can use the simulator to determine the actual effects, the backward operator descriptions are used for goal regression and must specify the symbolic pre-images of the operators in a concise manner. Therefore, the backward operator descriptions are based upon approximations of the effects of the operators. They describe the conditions under which an operator has a certain effect, assuming that no other traffic is present in the network.

Another file describes the network topology: the trunk groups, the switches, the connectivity and the trunk group capacities[2]. These aspects are static and do not change between problems. It also specifies the routing table for the network: for a given switch and an incoming call with a given destination, the table determines the next switch to which the call should be sent.

Unlike OTHELLO, there is no unique initial starting configuration for TNM. There are five network problem files, each of which specifies a different network problem and a solution. A network problem is a traffic configuration such that the throughput of the uncontrolled network under the traffic load is less than 95%. The solution is a set of controls that, when applied to the network, produce a throughput of at least 95%.

7.3 Experiments

As with OTHELLO, the experiments in TNM used both an LTU and a univariate decision tree as a learned concept form. The training strategy was similar to that used for OTHELLO, described in Section 6.3.1. Higher limits were placed on the evaluation function costs because the features in TNM required more execution time than the OTHELLO features. The evaluation function cost limit was assigned different values, from two to five seconds in steps of one second.

In TNM, Zenith does not perform a problem solving episode in every cycle. Instead, Zenith processes a set of four network problems and solutions, from which

[2]The network is "reduced" in that the capacity of each trunk group is a fraction (1/4) of the ILS-10 network. This reduction is done for efficiency in simulation.

it extracts a total of 663 training instances. Each of the four represents a different network problem, and each can be solved by the application of multiple DRR controls on the network. The solutions were provided by hand, so the instances extracted from them were assumed to be of high quality. Because the solutions are provided, Zenith does not train on-line in the TNM domain; unlike with OTHELLO, Zenith is provided with an initial set of expert-derived instances which do not change in the course of feature generation.

As with OTHELLO, instance accuracy and performance were measured at the end of every cycle. Instance accuracy was measured by repeatedly partitioning the instance set into training and testing sets, and measuring the classification accuracy of the trained preference predicate. This method is described in Section 6.3.

At the end of every cycle, performance was tested by using the learned preference predicate to direct problem solving in a set of five problems. Four of the problems were those from which the initial set of instances was extracted; the fifth problem was novel. Beginning with the initial problem state, in which the network's throughput was less than 95%, the performance component used best-first search, directed by the preference predicate, until it found a solution. A solution state was defined to be a network whose throughput is 95% or better.

Performance was evaluated by counting the number of nodes visited in the search. A limit of 2000 was placed on the number of nodes that could be visited during problem solving. Search was terminated when a goal state was found, or when the number of nodes in the frontier exceeded 2000. Therefore, in TNM, Zenith's performance goal was to find a solution state quickly, rather than to find the highest-quality solution state.

7.3.1 Linear Threshold Unit

Four experiments were run varying the evaluation function cost limit as an independent parameter. Each of the experiments was allowed to run for ten cycles, at which point the accuracies had stabilized. Figure 7.4 shows classification accuracy as a function of cycle number. The true accuracies lie within a confidence interval of $\pm 2\%$ with 99% probability. Table 7.1 summarizes the runs and Zenith's feature generation.

In these runs, the instance accuracy rose from 73% in the first cycle to a final value between 81% and 93%. The feature generated in the first cycle of all the runs measured the throughput of the network, and this feature alone accounted for the initial 73% accuracy. Limited to 2 seconds, the evaluation function accuracy could not attain better than 81% classification accuracy, but above 3 seconds the runs were able to attain about 93% accuracy on the instances.

Figure 7.5 shows Zenith's performance in TNM at every cycle. The figure shows, over the five problems, the number of nodes expanded in the search that was performed at the end of every cycle. As with OTHELLO, the performance improvement is erratic. One would expect that as the classification accuracy of the LTU improved,

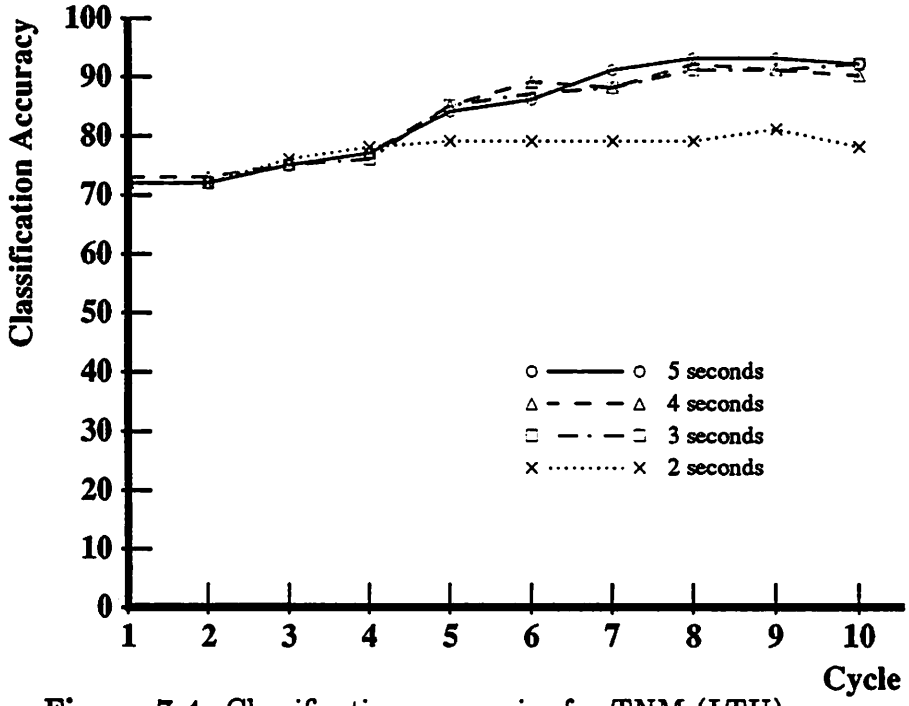


Figure 7.4 Classification accuracies for TNM (LTU).

Table 7.1 Number of features generated for TNM (LTU)

Time limit (seconds)	Total generated	Active at end of run	Accuracy attained
2	85	5	81%
3	87	4	92%
4	88	3	92%
5	119	5	93%

the number of nodes expanded would decrease; that is, the evaluation function would become more accurate at directing the search to a solution node. This does not seem to occur. Inspection of node counts for the individual problems reveals that the evaluation function oscillates in the ability to direct search. The evaluation function may solve Problem 1 quickly in cycle i, and not be able to solve it at all in cycle $i+1$, but be able to solve other problems more quickly instead. There appears to be a trading-off of ability among the problems when the LTU is retrained.

7.3.2 Univariate Decision Trees (C4.5)

Zenith was also run with the C4.5 program as a concept learner. C4.5 learns preference predicates in the form of decision trees. The settings used with C4.5 were the same as those used for OTHELLO, described in Section 6.3.2. Figure 7.6 shows

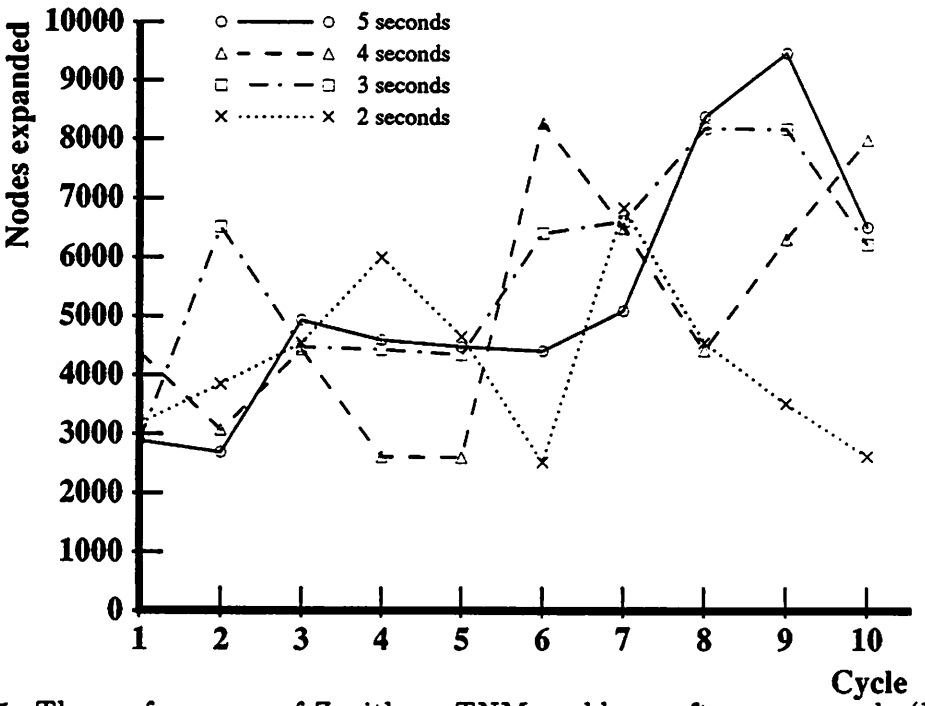


Figure 7.5 The performance of Zenith on TNM problems after every cycle (LTU).

Table 7.2 Number of features generated for TNM (C4.5)

Time limit (seconds)	Total generated	Active at end of run	Accuracy attained
2	79	3	93%
3	80	2	86%
4	84	3	93%
5	96	4	93%

the instance classification accuracies in the TNM domain using the C4.5 program to learn decision trees. Table 7.2 summarizes the runs and Zenith's feature generation.

The accuracies in the first cycle started lower with C4.5 than with the LTU (65% versus 73%), but the final accuracies in the tenth cycle equaled those of the LTU. In the 3 second run, the accuracy never rose above about 88%, due primarily to imperfections in the feature selection method described in Section 6.3.2.

Figure 7.7 shows Zenith's performance in TNM at every cycle. The figure shows, over the five problems, the number of nodes expanded in the search that was performed at the end of every cycle. As with the LTU, performance was erratic; there was no apparent correlation of search effort with classification accuracy. Future work will explore the relationship between classification accuracy and performance, discussed in Section 8.2.3.

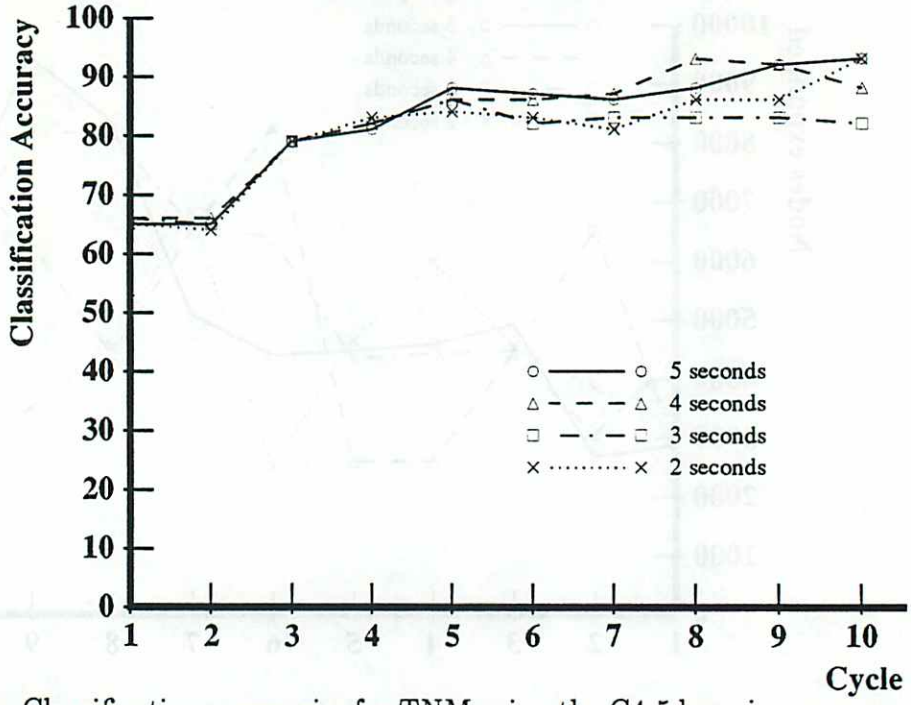


Figure 7.6 Classification accuracies for TNM using the C4.5 learning program. True accuracies lie within a 2% confidence interval with 99% probability.

7.4 Feature Generation in TNM

Figure 7.8 illustrates the derivations of some of the TNM features generated by Zenith. As with OTHELLO, the feature names in the figure were assigned by hand to indicate what the feature computes. The text of the features appears in Appendix D.

The performance goal for TNM is to increase the throughput of the network, defined as the number of calls completed to all switches divided by the number of calls attempted (introduced into the network). Therefore the initial feature, Throughput, measured the overall throughput of a network, and it was sufficient to discriminate preferences among 73% of the instances. This feature was then decomposed into two features measuring the number of calls attempted (Call Attempts) and the number of calls completed (Call Completions), respectively. The number of calls attempted was constant for all states, so it had zero discriminability and was not developed. Call Completions proved a useful feature because the number of calls completing in a state was a strong predictor of that state's desirability.

Zenith regressed the formula of Call Completions through the Destination ReRoute (DRR) operator, yielding Completions Achievable By Reroutes. This feature measures the total number of call completions in a network that would result from all possible single DRR controls applied to a state. Thus, the feature measures the total rerouting potential within a state between all pairs of switches.

The feature Completions Achievable By Reroutes was useful but expensive, so it was specialized in various ways. Other, more specialized features were created by

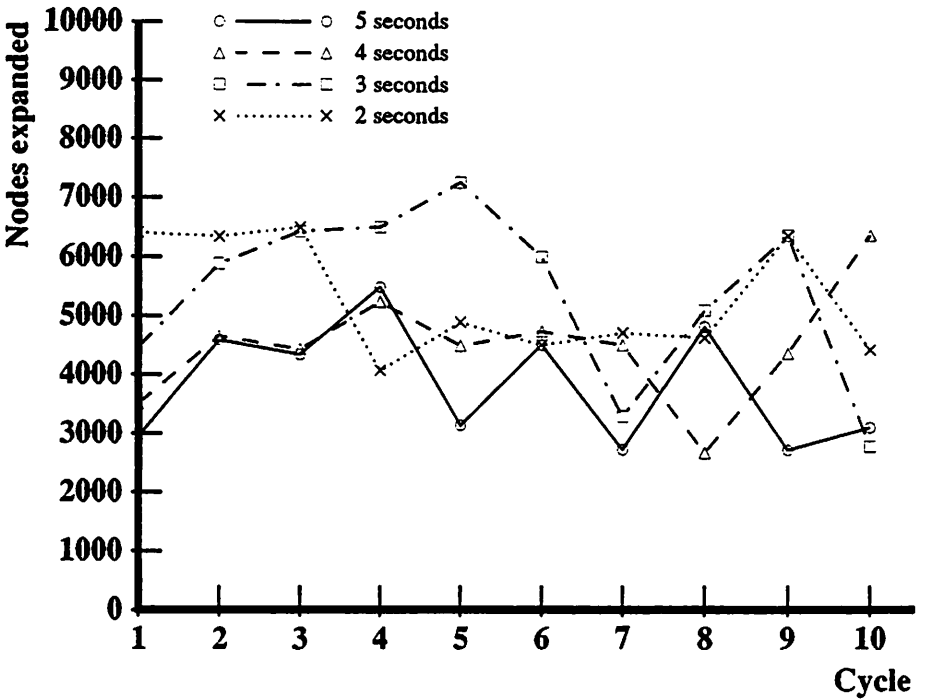


Figure 7.7 The performance of Zenith in TNM, using the C4.5 learning algorithm.

expanding definitions from the domain theory and extracting cases by expand-to-base-case. One such feature was Completions ReRouting Through Non-default TGs, which counts only those completions that would re-route through a non-default trunk group via a non-cyclical route to the destination.

Because the formula of Completions Achievable By Reroutes employed an arithmetic calculation, it was decomposed by split-arith-calc into three features, each of which calculated a subterm of the computation of the total completions. One feature Total Completions, measured the total number of completions to every switch in the network. Another feature, Traffic Between Pairs, measured the total amount of traffic that flowed between every source-destination switch pair in a state. The third feature, Traffic Through Reroutes, calculated the traffic between every source-destination pair using a rerouted (non-default) trunk group. All three of these resulting features were expensive, so abstraction and specialization transformations were applied to them to reduce their cost.

Traffic Between Pairs was specialized into two features, one of which was Cycle-free Traffic Between Pairs. This feature considers only paths that have no cycles in their routing, and thus eliminates from consideration pairs between which no traffic would flow because of the cycle. Another feature, Traffic Between Pairs with No Demand, was a futile specialization. It computes the traffic that will flow from a source switch to a destination switch when there is *no* traffic demand for the destination at the source switch; therefore, it always returns zero in every state and has no discriminability. It was eliminated quickly by Zenith.

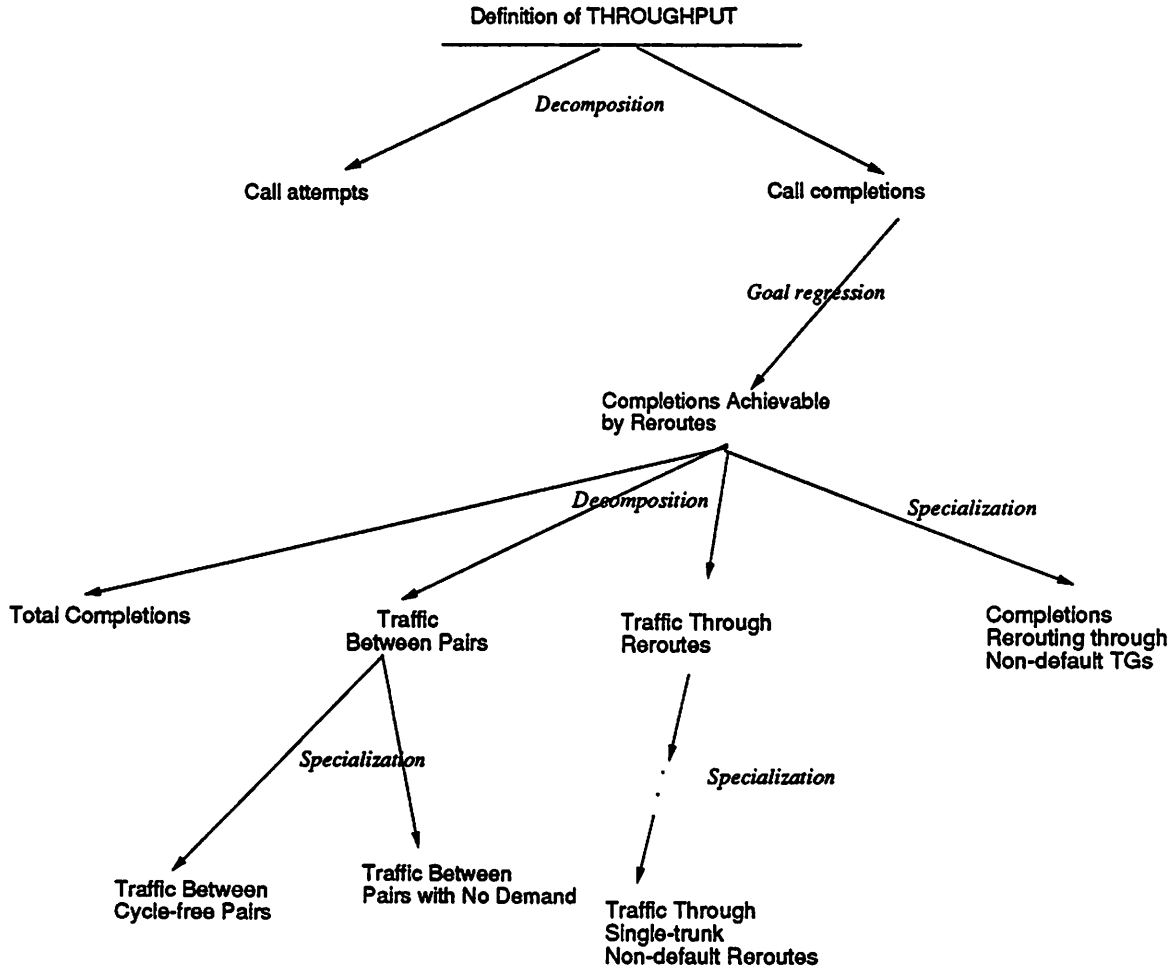


Figure 7.8 TNM features generated by Zenith

The domain theory uses several recursive predicates to specify the paths that call traffic will take. The base case of a traffic path is a pair of switches that are directly connected. Through several applications of a specialization transformation (expand-to-base-case, a feature was created that measured the number of calls rerouted between two directly connected switches. This feature is called Traffic Through Single-trunk Non-default Re-routes in Figure 7.8.

Tables 7.3 and 7.4 show the transformation firing data for Zenith using the LTU and C4.5 concept learners, respectively. Several observations can be made. One of the decomposition transformations, split-arith-comp, was never used in this domain because there are no arithmetic comparisons in the domain theory. Similarly, split-disjunction was not used because there are no disjunctive expressions in the domain theory.

Table 7.3 Transformation firing data for TNM (LTU)

Transformation	Class	Time Limit			
		2	3	4	5
split-arith-comp	Decomposition	0	0	0	0
split-arith-calc	Decomposition	9	9	9	13
split-conjunction	Decomposition	0	0	0	0
remove-negation	Decomposition	1	7	5	9
split-disjunction	Specialization	0	0	0	0
expand-to-base-case	Specialization	29	34	34	46
variable-specialize	Specialization	0	0	0	0
remove-LC-term	Abstraction	21	12	13	6
remove-variable	Abstraction	21	21	23	15
regress-formula	Goal regression	3	3	3	3

Table 7.4 Transformation firing data for TNM (C4.5)

Transformation	Class	Time Limit			
		2	3	4	5
split-arith-comp	Decomposition	0	0	0	0
split-arith-calc	Decomposition	15	8	19	13
split-conjunction	Decomposition	0	0	0	0
remove-negation	Decomposition	4	1	6	7
split-disjunction	Specialization	0	0	0	0
expand-to-base-case	Specialization	32	33	33	38
variable-specialize	Specialization	0	0	0	0
remove-LC-term	Abstraction	15	16	13	11
remove-variable	Abstraction	7	18	7	23
regress-formula	Goal regression	5	3	5	3

Several general observations can be made about Zenith's feature generation in this domain:

1. As with OTHELLO, goal regression in TNM produces new features whose formulae match *all* applications of an operator. For example, Completions Achievable By Reroutes measures the total rerouted traffic between all pairs in the network; that is, it counts the traffic that would travel through all possible reroute paths. As such, the regression of a state through an operator results in the *potential* of an operator in a state. It is possible that some other function, such as the maximum value from any individual operator, might be more useful.

2. Unlike OTHELLO, TNM is not pattern-based. The formulae of features in TNM calculate the flow of traffic, rather than match the patterns of individual call paths. The paths of individual calls are not available in TNM.

   Remove LC term is an abstraction transformation that removes the least constraining term from a conjunction. In TNM, this transformation was used mostly to remove terms from a formula that were no longer necessary for a calculation; removal of such terms rarely made the formula more abstract. Another kind of abstraction might be more appropriate in this domain. Other work in abstraction has investigated transformations appropriate for numerical equations; for example, replacing a function call with a constant [Ellman, 1990], or extracting the dominant term from a sum [Mostow & Fawcett, 1990]. The counterpart to remove-LC-term for numerical formulae may be a transformation that replaces a variable with a zero and eliminates the procedure call(s) that compute the variable's value. The use of such approximating abstractions is a subject for future work.

3. The variable-specialize transformation was not used in TNM. There is no inherent reason why variable-specialize would not be applicable to features in this domain. However, the transformation as implemented only applies to a feature having a single variable in its variable list and in which the formula imposes some constraint on the variable value. The features in TNM tended to have lengthy variable lists, so variable-specialize did not apply. Section 8.2.4 discusses extensions to specialization that would likely increase its applicability in TNM.

Because the abstraction and specialization transformations were not as effective in this domain as they were in OTHELLO, Zenith was less effective in reducing feature cost. As a result, fewer features could be kept in the active feature set (usually three or four) in these runs, and in most of the runs fewer features were developed than in the OTHELLO runs.

Evaluation of Zenith's generated TNM features is not as straightforward as with OTHELLO. The only known problem-solving system for TNM that performs state-space search is MacLearn [Iba, 1988; Silver, Vittal, Frawley, Iba & Bradford, 1990b], which uses network throughput as its evaluation function. MacLearn improves its performance by learning macros to encompass multiple solution steps in problem solving, but it does not change its evaluation function.

Human managers of telecommunications networks do not perform state-space search. They solve network problems using pre-computed plans for foreseeable and previously solved problems [Brandau, Lemmon & Lafond, 1991], so features such as Zenith's may not have much meaning to them. The features may measure recognizable quantities, but the relevance of the quantities to the value of a state may not be apparent. However, the features presented in this section were reviewed by Bernard Silver[3], who found them reasonable and plausible. Silver identified several of the

[3]Personal communication

features as being similar to features used in his hand-crafted NetMan system [Silver, Vittal, Frawley, Iba & Bradford, 1990b].

Zenith has been added as an agent to the integrated learning system (ILS) developed in the Self-Improving Systems Department of GTE Laboratories [Silver, Frawley, Iba, Vittal & Bradford, 1990a; Silver, Vittal, Frawley, Iba & Bradford, 1990b]. Preliminary experiments indicate that Zenith is able to effect performance improvement within the integrated system.

CHAPTER 8

DISCUSSION

This dissertation began with the observation that search is a critical part of problem solving. State-space search is a common way of exploring a problem space, but exhaustive search of the space is impractical in most domains. Many state-space search procedures employ an evaluation function to determine the quality of a search state. Designing evaluation functions may be difficult and time consuming.

Many methods have been created for learning evaluation functions automatically from examples; however, all such methods are sensitive to the set of features used to represent the examples. If the representation is appropriate to the task, learning will result in an accurate evaluation function that is useful for the state space; if the representation is inappropriate, the evaluation function will be ineffective for directing search. Specifying an appropriate representation for a complex problem-solving task may be difficult and may require time-consuming trial-and-error. In order to automate completely the generation of evaluation functions, it is necessary to automate feature generation. The problem of automatic feature generation for problem-solving has remained open since Arthur Samuel's original work on checkers, approximately thirty years ago.

8.1 Evaluation

This dissertation presents a theory of feature generation comprising four classes of transformations and a strategy for controlling them. The four classes are decomposition, goal regression, abstraction and specialization. The thesis pursued in this research is that:

> *Useful features for an evaluation function can be created by directed search through a feature space defined by four classes of transformations: goal regression, abstraction, decomposition and specialization.*

The theory comprises the transformation classes and the basic control strategy, but no commitment is made to the specific transformations and control decisions made in the implementation. As Zenith is applied to additional domains, it is possible

that new transformations could be added, or existing transformations generalized. Section 8.2 discusses extensions that could be made to the transformations, as well as their probable advantages and disadvantages. These extensions are based on the experience of applying Zenith to two domains. It should be noted that no new transformation classes had to be created in order to apply Zenith to a second domain, and the extensions were relatively minor.

Section 6.4 demonstrated that Zenith, when applied to a domain theory for OTHELLO, is able to generate features that are functionally equivalent to many of the known features of that domain. With some extensions to its transformations, primarily the specialization transformations, Zenith could account for most of the known OTHELLO features. Therefore, Zenith succeeds as an *explanatory* theory of feature generation, in that it answers the question *How can known features of a domain be derived?*[1]

When applied to both OTHELLO and telecommunications network management, Zenith also generates "new" features that are useful for expressing preferences among states. These features were reviewed in Sections 6.4.2 and 7.4. These features are both useful and novel. This demonstrates that Zenith succeeds as a *generative* theory of feature discovery, in that it accounts for the generation of new, useful features that were previously unknown in the domains.

In both OTHELLO and TNM, Zenith is able consistently to improve its ability to classify state preference pairs by developing its set of features. Classification accuracy increases smoothly and almost monotonically with the cycle number. Thus, the features generated by Zenith are appropriate for the domain and useful for expressing the concepts necessary for problem solving. Zenith is also able to improve problem-solving performance, although the improvement is erratic. This issue is discussed further in Section 8.2.3.

The theory also can be evaluated as a method for generating features automatically. This research is exploratory and was not intended to be an automatic off-the-shelf method that could be applied easily to new domains. Nevertheless, it is instructive to evaluate Zenith as an automatic feature discovery system.

Zenith requires a domain theory, a problem solver, a means of storing instance states, and (sometimes) changes to the critic. The problem solver is not strictly necessary if instances are to be provided externally, for example from expert problem solving episodes.

The most critical of the components is the domain theory, without which Zenith cannot operate. In general, it is not difficult to supply a domain theory because most of the information required is prerequisite to building a problem solving system for the domain anyway. The most difficult aspect of writing the domain theory is providing the goal regression information, which is discussed further in Section 8.2.5.

[1]The term "explanatory" here refers not to how features were derived, but how they *can be* derived. Zenith is not a cognitive theory.

Zenith is completely automatic: after the components required by the new domain have been provided, no interaction with the user is necessary.

It is difficult to estimate the amount of time required by Zenith to derive a "good" set of features for a new domain. The ten-cycle runs in Chapters 6 and 7 took between 24 hours and six days to complete (running on a Sun-4 workstation). However, a large fraction of that time was spent testing the performance of the preference predicate on every cycle. This time could be reduced by testing performance less often, for example, on every third cycle.

Tables 6.1, 6.4, 7.1 and 7.2 show data on feature generation for OTHELLO and TNM. These data indicate that feature generation increases sublinearly with the amount of time allowed for the evaluation function.

8.2 Problems, Issues and Future Work

Experience with the Zenith implementation has illustrated areas of difficulty and future work for this research. This section discusses each of these areas, alternatives considered, and the advantages and disadvantages of the alternatives.

8.2.1 Feature Selection

One of the most important and time-consuming components of Zenith is feature selection. After Zenith generates new features, it must determine the best subset of features that will fit within its allowed evaluation function cost. Feature selection is critical because it determines not only the quality of the evaluation function but also the features that are considered active, and active features are developed differently from inactive ones.

Zenith imposes several unusual demands on a feature selection method. These demands made it difficult to find a suitable selection strategy.

1. The method must be sensitive to feature cost as well as feature worth; it must choose a subset of features that can be evaluated within the evaluation function cost limit. Some weight-decay methods [Weigend, Rumelhart & Huberman, 1991] are designed to trade off accuracy with the complexity of a network, but they do not incorporate the expense of input features.

2. The method must be able to accommodate groups of features that are very similar, and possibly identical, to each other. This is because Zenith's transformations may produce a new feature that is syntactically different from the original but functionally similar or identical to it. Both old and new features compete with each other, and the selection mechanism must be able to decide that only one should be chosen for the final set.

The simultaneous existence of similar features in a set can confound selection methods that evaluate features independently. A method may *under*estimate the worth of a replicated feature because it essentially divides the worth among the other similar features. One example of this is the phenomenon of *multicolinearity* with LTUs: if several features are linearly dependent, then the weight assigned to them by an LTU learning method may be divided among them.

A method may *over*estimate the worth of such a feature by discarding other features of lower worth rather than discarding one of the similar set. In the latter case, the group of similar features will "clog" the feature set, causing other features to be discarded.

3. The method must be able to evaluate the contribution of a feature in the context of other features in the set. A feature alone may be a poor predictor of preference, but when combined with other features may be stronger.[2] Feature selection methods that evaluate features individually may under-estimate the worth of a feature for this reason.

4. In some cases a feature will be created that produces zeroes on many of the instances; that is, it will not differentiate many of the preference pairs, so many of the resulting delta values will be zero. Zenith's feature selection method must be able to evaluate correctly the contribution of such a feature. Some methods only penalize a feature for poor correlation with preference, but because features are evaluated on *all* states, some penalty must be incurred when the feature indicates no preference.

Determining the optimal acceptable subset of a set of features is an instance of the Knapsack problem [Baase, 1988], known to be NP-complete. Guaranteeing an optimal solution requires evaluating every subset of the features, which requires an exponential number of applications of the learning algorithm, so exhaustive approaches are rarely used in practice. Instead, less expensive heuristic methods are used [Kittler, 1986]. Several heuristic methods were tried in Zenith.

In earlier work with Zenith [Fawcett & Utgoff, 1991], features were selected by reducing their worth to a single number and then sorting them by these values. A feature's worth value combined its cost and discriminability according to the equation:

$$worth(F) = \frac{discriminability(F)}{\sqrt{cost(F)}}$$

The square root in the denominator was used to attenuate the influence of cost, which was always greater than 1. Using this equation, features were sorted by their worth

[2] A classic example is the "exclusive-OR" concept. With the exclusive-OR of two variables, either variable considered alone is useless for predicting the instance classification. Only when both variables are considered is any prediction possible.

score and the n highest-worth features were selected (in this earlier approach the feature set was bounded by size rather than by cost).

This method was unsatisfactory for several reasons. The function $worth(F)$ of a feature is an individual measure that is independent of the contributions of the other features in the set, and Zenith would often select several features that were very similar. Also, the discriminability/cost tradeoff represented by the *worth* function was never perfect because cost would sometimes have a greater influence than desired. The *worth* function was an attempt to express this tradeoff as a single numerical equation, but it was difficult to derive an equation that could express universally the desired relationship between discriminability and cost. Ideally, the feature selection method should choose those features that collectively result in the highest accuracy attainable, within the allowed cost. Though several such equations were tried, none proved satisfactory.

The method ultimately adopted kept feature cost and contribution separate, and used a sequential backward selection method [Kittler, 1986]. Sequential selection methods do satisfy Zenith's four requirements, but they tend to be expensive: for n features they require $O(n^2)$ applications of the concept learner. As mentioned in Section 6.3.2, this expense was acceptable for LTUs but prohibitive for C4.5, so the method was modified for C4.5. The modification enabled the least valuable feature to be estimated after a single run of C4.5, thus reducing the entire selection process to $O(n)$ applications of C4.5.

Even with this modification, feature selection remains one of the most expensive components of Zenith, and probably will remain so. Other selection methods were tested [Kittler, 1986; Kira & Rendell, 1992] but were either too expensive or failed to satisfy Zenith's requirements for feature selection. Because of the criticality of feature selection, the search for an inexpensive but accurate selection method remains an important area of future work.

8.2.2 Feature Generation

New features in Zenith are created by applying transformations to old ones, then letting the combined features compete with each other. The features are forced to compete because the feature set is bounded: the active features are bounded in cost by the evaluation function cost limit, and the inactive features are bounded in number. Any feature that is not active and does not make the inactive feature "cut" is ejected and not developed. Thus, the number of active and inactive features act as a beam width in beam search.

Several variations of this control strategy were tested, such as fixing the size of the active feature set, restricting the number of new features that could be introduced in each cycle [Fawcett & Utgoff, 1991], and changing the size of the inactive feature set. Empirically, the current strategy was found to produce the best results, and was satisfactory for the experiments reported [Fawcett & Utgoff, 1992]. However, it could be improved. Competition in the feature set is still high; in the final cycles of the

runs, many features tend to be thrown away that have non-zero discriminabilities and could possibly be developed further.

One way to improve feature generation would be to make the application of transformations more intentional and focused. For example, Zenith could decide that one of its features was valuable but expensive, and that it should devote one cycle to reducing the feature's cost while ignoring the other features. Zenith would then apply transformations to the feature, perhaps repeatedly, and finally extract the best feature that it had generated and discard the others. This approach would thus concentrate a small "burst of energy" on developing the feature space around a known useful feature.

There are two advantages to decreasing competition in this way. First, the feature versions would only need to compete among themselves because the rest of the feature set would not be changing. This would reduce the amount of effort needed to generate the final features used because Zenith need not invoke its general feature selection method; if Zenith is looking only for a cheaper version of the original feature, it need only take the feature of lowest cost of the set developed from the original feature. The second advantage is one of decreased competition: by developing the feature space around a single feature at a time, Zenith could perhaps do a more thorough job than by developing many features simultaneously. Let f_1 designate the original feature and $f_1, f_2, \cdots, f_n$ be the path to a finally chosen feature f_n. With the current control mechanism, each of the features $f_2, f_3, \cdots, f_{n-1}$ must be good enough to survive the cycle in which it is created so that the next feature can be created from it. If any of $f_2, f_3, \cdots, f_{n-1}$ is ejected from the feature set, f_n may not be created. By "freezing" the rest of the feature set and developing only one feature at a time, the reduced competition enables a portion of the space to be developed more thoroughly.

There are some complexities to this approach. It requires a strategy for determining which feature to work on in a given cycle, and how much effort to devote to trying to improve the feature. Such a strategy must also deal with situations in which a newly-created feature is significantly different – not just cheaper – than the original feature. Because of these complexities, this "focused burst" approach was not implemented in Zenith. However, the approach is promising and incorporating it into Zenith is an important direction for future work.

8.2.3 Preference Classification, Decision Making and Performance

An important issue uncovered by this research is the relationship between decision accuracy and performance. We assumed that as an evaluation function became more accurate, the decisions based on it would improve, and the performance of the problem solver would improve in turn. In Zenith this seems to happen in general, but the performance improvement is not stable. In most runs the classification accuracy of Zenith's preference predicate rises almost monotonically; however, the corresponding effect on problem solving performance is much more erratic than would be expected.

There are several hypotheses that would explain this phenomenon. Temporal crosstalk, discussed in Section 6.3.1, is caused by the interaction of learning and performance. Any system that does on-line training, in which learning is interleaved with performance episodes, is susceptible to temporal crosstalk. This phenomenon was identified by Jacobs (1990), but no solution has been found yet.

Another hypothesis is that certain decisions in a domain are more critical than others, and that once a critical decision is made, the exploration of the search space after that decision is greatly affected. For example, in OTHELLO losing a corner square is critical, and it makes little difference how well other decisions are made after several corners have been lost. However, the instance base may not reflect the importance of this decision; the instances involving this critical decision may be no more prevalent than instances of other, less critical decisions. As a result, two preference predicates may have equal accuracy when measured against an instance base, but one may be much better at making critical decisions, and thus result in much greater performance. In their earlier work on backgammon that used expert-ranked boards, Tesauro and Sejnowski (1989b) emphasized the necessity of intelligently hand-crafting the training set to illustrate particular difficult points.

Unfortunately, there may be no *a priori* way to determine which decisions are critical, and in turn which preference pairs are more important for the predicate to distinguish. It is possible that Temporal Difference (TD) learning [Sutton, 1988] can overcome this problem. However, preliminary experiments by Callan (1993) that used TD learning in a similar framework exhibited similar erratic behavior. The relationship between classification accuracy and problem solving performance is an important topic of future work.

8.2.4 Specialization

Zenith uses three specialization transformations: expand-to-base-case, remove-disjunct and variable-specialize. The first two are standard methods of specializing expressions. The third is unusual in that it specializes variables: it produces sets of values that empirically satisfy the formula or a portion of the formula. In essence, variable-specialize induces from examples the invariants of a portion of a feature. It can also be characterized as a transformation that tries to create a cheaper structural feature from a valuable but expensive functional feature [Callan, 1993]. Section 6.4 referred to a number of possible extensions to the variable-specialize transformation that might account for more of the known OTHELLO features.

Currently, variable-specialize only accepts values that empirically *always* satisfy the conditions of the specialization (the q predicate of Section 5.4.4). One possible extension is to weaken this test so that a value need only satisfy the test most of the time, by imposing a non-zero threshold on the portion of instances in which the element was observed to vary. This would enable variable-specialize to create, for example, a class for squares that were usually but not always stable. The disadvantage of introducing such a threshold is that it would increase the chance of spurious

inclusion in the set(s) generated. The induction performed by the transformation would be more susceptible to coincidences and false matches.

Currently, variable-specialize looks for *individual* domain elements that satisfy the unary predicate q. This could be extended in two ways simultaneously. The transformation could allow q to be an n-ary predicate, and it could examine configurations of variable values that satisfy q, rather than individual values. For example, variable-specialize currently only finds individual OTHELLO squares to be stable. With the extension it could find configurations of squares that are stable together. Other examples, mentioned in Section 4.2.4, are the Bridge and Triangle of Oreo features of checkers, both of which are special configuration of squares that satisfy the "king prevention" property. The disadvantage of such an extension is that many more examples would have to be examined, because the combinations would be rarer among instances than the individual domain elements would be. This extension would be expensive, and like weakening invariance, might lead to spurious configurations being accepted.

Another improvement to variable-specialize might be to prove invariance *analytically* using the domain theory. For example, to most experienced OTHELLO players it is "obvious" that corner squares are stable, and it may seem wasteful for Zenith's variable-specialize transformation to need to search through examples of moves to determine this fact. Variable specialize could be altered so that, given a set of domain elements, it would hypothesize that each is invariant and attempt to prove the hypothesis using the domain theory. Unfortunately, there are disadvantages to using analytical theorem proving to confirm invariance. The greatest disadvantage, and the reason it was not used in Zenith, is that the domain theory may not include all the information necessary to complete the proof. For example, proving the stability of corner squares in OTHELLO requires facts such as:

- The three conditions `blank(X)`, `owns(white,X)` and `owns(black,X)` are mutually exclusive.
- If one square does not satisfy a condition, then no longer span of squares containing that square will satisfy the condition either.

Such facts constitute meta-knowledge about the domain that is not necessary for the completeness of the domain theory, but is necessary for the proof. Though these facts could be added to the domain theory, the problem lies in determining that these facts are necessary for the proof to succeed in the first place. There is no way of knowing whether a proof failed because the hypothesis was not true, or because some critical knowledge was missing. Such knowledge simply cannot be assumed to exist.

The second disadvantage of proving invariance analytically is that strict invariance is not always desirable. As already mentioned, there are benefits to weakening variable-specialize so that it accepts variables that are usually but not always invariant. Proofs of invariance could not easily provide information about the frequency with which conditions are satisfied or violated.

8.2.5 Providing Goal Regression Information

Applying Zenith to a new domain requires a new domain theory, a new problem solver, a new instance manager, and (sometimes) changes to the critic. Of these requirements, supplying the domain theory is the most difficult and time-consuming. Much of the effort of writing a domain theory is involved in specifying the goal regression information; that is, the expressions that specify the pre-images of conditions with respect to the domain's operators.

Many machine learning systems employ the simpler STRIPS operator formalism [Sacerdoti, 1974; Waldinger, 1976], but Zenith cannot use it. The STRIPS formalism requires, for each operator, a set of *preconditions*, a *deletelist* and an *addlist*. The elements of these lists are literals. An operator can only be applied in a state if every condition in its *preconditions* list is satisfied. The application of the operator consists of deleting every condition of *deletelist* from the current state, then adding every condition of *addlist* to the current state. Expressions in the *deletelist* and *addlist* can contain variables bound by the *preconditions*.

The STRIPS operator representation is very constrained and many learning systems use it because it simplifies analysis: operator definitions are easy to inspect, interactions between operators are straightforward to predict, and operator pre-images are easy to determine. Unfortunately, neither OTHELLO nor TNM has operators that can be expressed using the STRIPS representation. OTHELLO's operator requires doubly-nested universal quantifiers, and TNM's controls require functions to determine their effect on network traffic flow. Neither of these can be expressed using just the literals allowed in *addlists* and *deletelists*.

Because of these requirements, Zenith allows operator definitions to be arbitrary Prolog procedures. Unfortunately, this generality prevents pre-images from being computable automatically from the operator definitions. Zenith, like STABB [Utgoff, 1986b], requires that the domain theory author specify the operator pre-images. There are as yet no methods for inverting such operators automatically, so specifying the pre-image expressions for a domain theory remains the most time-consuming part of applying Zenith to a new domain. This remains an open problem for Zenith, as it is in the field of machine learning in general. The complexity of the operators in Zenith's domains is probably more typical of problem solving domains than is the relatively simple STRIPS formalism.

8.2.6 Guiding Goal Regression with Examples

The goal regression performed by Zenith is similar to explanation-based learning (EBL) [Mitchell, Keller & Kedar-Cabelli, 1986], but with several differences, the most significant of which is that Zenith's goal regression is not guided by examples. The use of examples to guide the regression process is critical in EBL, because the purpose of EBL is to create specific rules that will have high applicability in the domain; one

of the assumptions of EBL is that the examples seen are typical and occur frequently in the domain.

Zenith does not use examples to guide goal regression because, strictly speaking, such examples do not exist. Zenith has an instance base containing instances of state preferences, but these are examples of the preference concept. Goal regression is applied to features, so Zenith would need examples of features and their correct values, and such examples are not available. Given only the information that a state S_1 is preferred to another state S_2, it cannot be determined whether the value of a given feature in S_1 should be higher, lower, or equal to the feature's value in S_2. Without this information, there is no way to use state preference information directly.

However, it is possible to use the preference pairs indirectly by making assumptions about feature values. If a feature f correlates positively with state preference, then we can assume that f generally should have a higher value in the preferred state than in the non-preferred state of a pair. We can then extract all such preference pairs in which f has a higher value in the preferred state. These states could serve as examples in explanation-based learning, and they could be used to create specializations of f. The specializations would be created by identifying the most common proofs involved in f's computation. This approach was not explored in Zenith because of the complexity involved, and because the heuristic assumptions involved in selecting instances for f from preference states make use of examples for features less promising than the use of examples in EBL. However, any method of focusing the feature generation process is potentially very valuable, and future work should explore this approach.

8.2.7 Feature Optimization

Two of the transformation classes, abstraction and specialization, are intended to improve features by decreasing their cost without sacrificing much of their accuracy. Abstraction does this by removing conditions of a conjunctive formula, effectively generalizing it. Specialization decreases cost by creating special cases of a feature. Abstraction and specialization are both truth-*compromising* (as opposed to truth-*preserving*) operations; the resulting features usually have different semantics from the original. Zenith also contains a simplifier that simplifies the formulas of features to reduce their cost.

Much more work could be done to optimize Zenith's features. An alternative to removing terms from a formula is to *reorder* them. Term reordering is beneficial because Prolog interprets a conjunction from left to right. Whenever a term backtracks, it essentially forces the re-evaluation of terms to its right. By reordering terms — for example, moving a rarely satisfied term to the beginning of a conjunction — the total number of term evaluations can be decreased greatly. Chase *et al.* (1989) state that removing terms from a conjunction has less of an effect on its cost than reordering the terms would have.

Warren's (1981) work in logic programming showed that a fairly simple approach can reorder terms effectively and produce a significant decrease in processing time. Unfortunately, Warren's method, as well as more sophisticated methods [Smith & Genesereth, 1985], require meta-information about predicates. Specifically, they require that the argument domain sizes[3] for all predicates be known. Zenith uses some meta-knowledge about predicates that is fairly easy for the domain theory writer to specify; however, argument domain sizes can be difficult to estimate. It is possible to learn argument domain sizes, but this would have added an extra learning component to Zenith, increasing its complexity and making the effects of feature generation difficult to isolate. Because of this additional complexity, conjunction reordering techniques were not incorporated into Zenith.

However, future work should include the investigation of such techniques because of their strong effect on feature cost. Other techniques in logic programming, such as partial evaluation [van Harmelen & Bundy, 1988] and abstract interpretation [Cousot & Cousot, 1977], might be used to simplify the domain theory and improve the efficiency of resulting features.

8.3 Conclusions

It has long been recognized that inductive learning is very sensitive to the representation used to express the examples. The ability to determine "good" features automatically has been a goal in machine learning for nearly as long. The theory presented here integrates analytical (theory-driven) and empirical (data-driven) methods for feature discovery. It is motivated by the desire to increase the power and range of constructive induction by going beyond the limitations of the two approaches.

This research makes several contributions to the field of machine learning. It is exploratory research that shows, for the first time, how handcrafted features like those used in expert problem-solving systems can be generated automatically. Though some of the handcrafted features were justified by their authors, little information was provided on how they could have been generated. Zenith is not a cognitive theory, so it does not explain how humans create features. However, it explains very well how similar features can be created by a machine, without the need of an expert.

The theory of feature generation integrates both feature *costs* and feature *benefits*. That is, the work acknowledges that every learned feature adds expense as well as accuracy to the evaluation function. Therefore, Zenith addresses the *utility problem* in machine learning. The utility problem is becoming increasingly important as machine learning methods are applied to complex problem-solving domains. The problem has been acknowledged in the subfield of explanation-based learning [Minton, 1988] and

[3]The domain size of a predicate's argument is the number of domain elements that could match that argument.

to a lesser degree in inductive learning as well [Nunez, 1988; Gennari, 1989; Tan & Schlimmer, 1990].

Zenith is one of the few symbolic systems that integrates feature discovery, concept learning and problem solving. Although previous work in constructive induction has used domains related to problem solving (*e.g.* tic-tac-toe concepts), the resulting features were not used in an evaluation function. Furthermore, Zenith is one of the few constructive induction systems to produce features for "difficult" domains; that is, a domain with a very large search space and an intractable domain theory.

Because Zenith learns in a domain using an intractable theory, its method for constructive induction constitutes a solution to the intractable theory problem in explanation-based learning [Mitchell, Keller & Kedar-Cabelli, 1986; Tadepalli, 1989; Mostow & Fawcett, 1990; Flann, 1990]. Most other methods learn rules for small portions of the search space, and they have not been applied to the full space of an intractable domain. The advantage of Zenith's approach is that it generates an evaluation function that can be used over the entire search space.

This work is exploratory, but it make a significant advance in the state of the art of feature generation. By automating the construction of useful features, it brings us much closer to an ideal of automatic concept learning from examples.

APPENDIX A

THE OTHELLO DOMAIN THEORY

This domain theory is available on-line from the UCI machine learning repository, in the file:

/pub/machine_learning_databases/othello/othello.theory

The repository is accessible via anonymous FTP (username is "anonymous", password is your e-mail address) from the Internet host ICS.UCI.EDU.

Note that this domain theory uses x for the Black player and o for the White player.

```
/*****************************************************************************
      Domain theory for Othello in Prolog.
      (actual dialect is Quintus Prolog, but little is dialect specific).

      Tom Fawcett (fawcett@cs.umass.edu)
      COINS Deptartment, LGRC
      University of Massachusetts
      Amherst, MA  10373
      February, 1992.

References:
       "Automatic Feature Generation for Problem Solving Systems"
       T. Fawcett and P. Utgoff.
       Ninth International Conference on Machine Learning
       Aberdeen, Scotland. 1992.  pp 144-153.


=============================================================================
  Primitives:

       State-independent:
  square(S) - S is a square
  direction(D) - D is a direction
  player(P) - P is a player (x and o)
  neighbor(S1,Dir,S2) - The neighbor of square S1 is S2 in direction Dir.
  opponent(A,B) - the opponent of A is B.

       State-specific:
  owns(P,S) - player P owns square S.
```

```
  blank(S) - square S is blank.

================================================================================

Notes about the representation here:

1) There is no explicit 'board' parameter of these state-specific predicates.
They are all relative to the 'current board', which is state-specific and may
be set via set_current_board.  The implementation of these operational
primitives is irrelevant to the domain theory, but they are defined in
veclowlevel.pl.

2) Similarly, the MOVE operator, when executed, performs a state change.
Rather than passing back a structure denoting the effect of the operator, it
actually *changes* the current board.  Therefore, the caller must copy the
current board before applying the operator, if the previous board is to be
preserved.

This "state change" approach was taken to avoid having to append board
parameters to every primitive in the domain theory.

********************************************************************************/



%  Meta-info about predicates:

%  Standard mode information
:- mode
        win(?),
        score(?, ?),
        opponent(?, ?),
        player(?),
        blank(?),
        owns(?, ?),
        direction(?),
        square(?),
        neighbor(?, ?, ?),
        legal_move(?, ?),
        bs(?, ?, ?),
        span(?, ?, ?, ?),
        in_line(?, ?, ?),
        line(?, ?, ?),
        \+(+).

%  Determinacy of predicates.  Arguments are either bound or var (unbound).
:- determinate
        owns(bound, bound),
        score(bound, var),
        opponent(bound, var), opponent(var, bound),
        neighbor(bound, bound, var), neighbor(var, bound, bound),
```

```
                neighbor(bound, var, bound),
        bs(bound, bound, var),
        span(bound, var, bound, var),   % Dir is redundant in this case.
        line(bound, bound, var).


%  Operationality information
operational(square).
operational(direction).
operational(neighbor).
operational(player).
operational(opponent).
operational(owns).
operational(blank).

%  Declaration of state-specific predicates
state_specific_pred(owns).
state_specific_pred(blank).
state_specific_pred(bs).

%  Calls to these will be used to compute feature cost, when feature cost
%  is measured by predicate calls rather than CPU time.
counted_predicate(square).
counted_predicate(direction).
counted_predicate(neighbor).
counted_predicate(player).
counted_predicate(opponent).
counted_predicate(owns).
counted_predicate(blank).

/*****************************************************************

The following facts should be asserted about every operator:
        OPERATOR(Op)
        PRECONDITIONS(Op, Preconds)
        EFFECTS(Op, Postconds)
        ZENITH_OPERATOR(Op)
        OPPONENT_OPERATOR(Op)
        AGENT_OPERATOR(Op)

where Op is a term specifying the operator and its arguments,
Preconds is an expression specifying the conditions under which Op is legal,
and Postconds specify the postconditions of the operator.

Both Preconds and Postconds should be callable, as they cause a state change
as side-effects.
*****************************************************************/

/************  The MOVE operator  **********/
operator(move(_Player,_Square)).
```

```
zenith_operator(move(x, _Square)).
opponent_operator(move(o, _Square)).

preconditions(move(MP,MS),
              legal_move(MS,MP)).

%  A specification of the effects.  Note that this states that EVERY piece
%  in the span ends up owned by MP, which is redundant because
%  if multiple spans are included, then the move square will be set
%  multiple times.
%
%  In this specification, MP is the player making the move, MS is the square
%  of the move, S2 is the last piece in the span (before the bracket),
%  and FlipSq is a flipped square.
effects(move(MP,MS),
        (forall(bs(MS,S2,MP),                  % For every bracketed span
               forall(in_line(FlipSq,MS,S2),   % For every square in the span
                      set_owns(MP,FlipSq))     %  change ownership to move player
                 ))).

%  Calling this executes the move.
do(move(MP,MS)) :-
        preconditions(move(MP,MS),Preconditions),
        call(Preconditions),           % Make sure preconds are satisfied.
        effects(move(MP,MS), Effects),
        call(Effects).                 % Create the effects.



%   WIN(P) - Player P has won on the current board.
win(P) :-
        end_of_game,                % Used to have a cut here
        player(P),
        opponent(P,Opp),
        score(P,PDiscs),
        score(Opp,OppDiscs),
        PDiscs > OppDiscs.

%  SCORE(Player,Score) - Player has Score on the current board.
score(Player,Score) :-  count([Move], owns(Player,Move), Score).


%   END_OF_GAME - The current board designates the end of a game.
end_of_game :-
        \+legal_move(_,x),
        \+legal_move(_,o).


%   LEGAL_MOVE(?Square, ?Player)
%   Square is a legal move for Player if it is blank and there is a bracketed
%   span emanating from Square.
```

```
legal_move(Square, Player) :-
        square(Square),
        bs(Square, _, Player).

legal_moves(Player,X) :- setof(M,legal_move(M,Player),X), !.
legal_moves(_Player,[]).                        % If no proofs of legal_move


%  BS(?S1, ?S3, ?P) - There is a bracketed span from S1 to S3.
%                A bracketed span of a player is an empty square, followed by
%                a span of discs owned by the opponent, followed by a disc
%                owned by the player.
%                NOTE that S1 is the BLANK square, and S3 is the LAST
%                square owned by the opponent.  The final bracketing piece is
%                not included in the span.
bs(S1, S3, P) :-
        blank(S1),
        opponent(P, Opp),
        direction(Dir),
        neighbor(S1, Dir, S2),
        span(S2, S3, Dir, Opp),
        neighbor(S3, Dir, S4),
        owns(P, S4).


%  SPAN(?S1, ?S2, ?Dir, ?Owner)
%  There is a span from S1 to S2 in direction Dir owned by Owner.  Note that
%  Dir could be inferred from S1 and S2, but is included explicitly for
%  convenience.  Note also that a span has length>=1.
%
%  This returns the longest spans first.
span(S1, S2, Dir, Owner) :-
        owns(Owner, S1),
        neighbor(S1, Dir, S3),
        span(S3, S2, Dir, Owner).
span(S, S, _, Owner) :- owns(Owner,S).


%  IN_LINE(S, Start, End)
%
%  Square S is in the span from Start to End.  Note that this is
%  state independent because it doesn't consider ownership of squares.
%
in_line(S, Start, End) :-
        direction(Dir),
        line(Start, S, Dir),            % There is a line from Start to S,
        line(S, End, Dir).              %  and there is a line from S to End.


%  LINE(?From, ?To, ?Dir)
%
```

```
%  There is a line of squares from From to To in direction Dir.
%  A line can contain only one square, in which case From=To.
%  Note that this is state independent.
%
line(From, From, _).
line(From, To, Dir) :-
        neighbor(From, Dir, Next),
        line(Next, To, Dir).


%  FORALL(p(...),q(...)) -
%  For every satisfaction of p, satisfy q.  Succeeds once.
%
%  This looks to be equivalent to the forall utility in the Quintus library
%  package library(foreach).
%
forall(P, Q) :- call(P), \+call(Q), !, fail.
forall(_P,_Q).


/*************************************************************************
      These define the board topology
*************************************************************************/
%  The directions
direction(n).
direction(ne).
direction(e).
direction(se).
direction(s).
direction(sw).
direction(w).
direction(nw).

%  The squares
square(a1).  square(a2).  square(a3).  square(a4).  square(a5).
square(a6).  square(a7).  square(a8).  square(b1).  square(b2).
square(b3).  square(b4).  square(b5).  square(b6).  square(b7).
square(b8).  square(c1).  square(c2).  square(c3).  square(c4).
square(c5).  square(c6).  square(c7).  square(c8).  square(d1).
square(d2).  square(d3).  square(d4).  square(d5).  square(d6).
square(d7).  square(d8).  square(e1).  square(e2).  square(e3).
square(e4).  square(e5).  square(e6).  square(e7).  square(e8).
square(f1).  square(f2).  square(f3).  square(f4).  square(f5).
square(f6).  square(f7).  square(f8).  square(g1).  square(g2).
square(g3).  square(g4).  square(g5).  square(g6).  square(g7).
square(g8).  square(h1).  square(h2).  square(h3).  square(h4).
square(h5).  square(h6).  square(h7).  square(h8).

%  NEIGHBOR(S1, Dir, S2)
%  Square S2 is the neighbor of S1 in the Dir direction.
%
```

```
neighbor(a1,e,b1). neighbor(a1,se,b2). neighbor(a1,s,a2). neighbor(b1,e,c1).
neighbor(b1,se,c2). neighbor(b1,s,b2). neighbor(b1,sw,a2). neighbor(b1,w,a1).
neighbor(c1,e,d1). neighbor(c1,se,d2). neighbor(c1,s,c2). neighbor(c1,sw,b2).
neighbor(c1,w,b1). neighbor(d1,e,e1). neighbor(d1,se,e2). neighbor(d1,s,d2).
neighbor(d1,sw,c2). neighbor(d1,w,c1). neighbor(e1,e,f1). neighbor(e1,se,f2).
neighbor(e1,s,e2). neighbor(e1,sw,d2). neighbor(e1,w,d1). neighbor(f1,e,g1).
neighbor(f1,se,g2). neighbor(f1,s,f2). neighbor(f1,sw,e2). neighbor(f1,w,e1).
neighbor(g1,e,h1). neighbor(g1,se,h2). neighbor(g1,s,g2). neighbor(g1,sw,f2).
neighbor(g1,w,f1). neighbor(h1,s,h2). neighbor(h1,sw,g2). neighbor(h1,w,g1).
neighbor(a2,n,a1). neighbor(a2,ne,b1). neighbor(a2,e,b2). neighbor(a2,se,b3).
neighbor(a2,s,a3). neighbor(b2,n,b1). neighbor(b2,ne,c1). neighbor(b2,e,c2).
neighbor(b2,se,c3). neighbor(b2,s,b3). neighbor(b2,sw,a3). neighbor(b2,w,a2).
neighbor(b2,nw,a1). neighbor(c2,n,c1). neighbor(c2,ne,d1). neighbor(c2,e,d2).
neighbor(c2,se,d3). neighbor(c2,s,c3). neighbor(c2,sw,b3). neighbor(c2,w,b2).
neighbor(c2,nw,b1). neighbor(d2,n,d1). neighbor(d2,ne,e1). neighbor(d2,e,e2).
neighbor(d2,se,e3). neighbor(d2,s,d3). neighbor(d2,sw,c3). neighbor(d2,w,c2).
neighbor(d2,nw,c1). neighbor(e2,n,e1). neighbor(e2,ne,f1). neighbor(e2,e,f2).
neighbor(e2,se,f3). neighbor(e2,s,e3). neighbor(e2,sw,d3). neighbor(e2,w,d2).
neighbor(e2,nw,d1). neighbor(f2,n,f1). neighbor(f2,ne,g1). neighbor(f2,e,g2).
neighbor(f2,se,g3). neighbor(f2,s,f3). neighbor(f2,sw,e3). neighbor(f2,w,e2).
neighbor(f2,nw,e1). neighbor(g2,n,g1). neighbor(g2,ne,h1). neighbor(g2,e,h2).
neighbor(g2,se,h3). neighbor(g2,s,g3). neighbor(g2,sw,f3). neighbor(g2,w,f2).
neighbor(g2,nw,f1). neighbor(h2,n,h1). neighbor(h2,s,h3). neighbor(h2,sw,g3).
neighbor(h2,w,g2). neighbor(h2,nw,g1). neighbor(a3,n,a2). neighbor(a3,ne,b2).
neighbor(a3,e,b3). neighbor(a3,se,b4). neighbor(a3,s,a4). neighbor(b3,n,b2).
neighbor(b3,ne,c2). neighbor(b3,e,c3). neighbor(b3,se,c4). neighbor(b3,s,b4).
neighbor(b3,sw,a4). neighbor(b3,w,a3). neighbor(b3,nw,a2). neighbor(c3,n,c2).
neighbor(c3,ne,d2). neighbor(c3,e,d3). neighbor(c3,se,d4). neighbor(c3,s,c4).
neighbor(c3,sw,b4). neighbor(c3,w,b3). neighbor(c3,nw,b2). neighbor(d3,n,d2).
neighbor(d3,ne,e2). neighbor(d3,e,e3). neighbor(d3,se,e4). neighbor(d3,s,d4).
neighbor(d3,sw,c4). neighbor(d3,w,c3). neighbor(d3,nw,c2). neighbor(e3,n,e2).
neighbor(e3,ne,f2). neighbor(e3,e,f3). neighbor(e3,se,f4). neighbor(e3,s,e4).
neighbor(e3,sw,d4). neighbor(e3,w,d3). neighbor(e3,nw,d2). neighbor(f3,n,f2).
neighbor(f3,ne,g2). neighbor(f3,e,g3). neighbor(f3,se,g4). neighbor(f3,s,f4).
neighbor(f3,sw,e4). neighbor(f3,w,e3). neighbor(f3,nw,e2). neighbor(g3,n,g2).
neighbor(g3,ne,h2). neighbor(g3,e,h3). neighbor(g3,se,h4). neighbor(g3,s,g4).
neighbor(g3,sw,f4). neighbor(g3,w,f3). neighbor(g3,nw,f2). neighbor(h3,n,h2).
neighbor(h3,s,h4). neighbor(h3,sw,g4). neighbor(h3,w,g3). neighbor(h3,nw,g2).
neighbor(a4,n,a3). neighbor(a4,ne,b3). neighbor(a4,e,b4). neighbor(a4,se,b5).
neighbor(a4,s,a5). neighbor(b4,n,b3). neighbor(b4,ne,c3). neighbor(b4,e,c4).
neighbor(b4,se,c5). neighbor(b4,s,b5). neighbor(b4,sw,a5). neighbor(b4,w,a4).
neighbor(b4,nw,a3). neighbor(c4,n,c3). neighbor(c4,ne,d3). neighbor(c4,e,d4).
neighbor(c4,se,d5). neighbor(c4,s,c5). neighbor(c4,sw,b5). neighbor(c4,w,b4).
neighbor(c4,nw,b3). neighbor(d4,n,d3). neighbor(d4,ne,e3). neighbor(d4,e,e4).
neighbor(d4,se,e5). neighbor(d4,s,d5). neighbor(d4,sw,c5). neighbor(d4,w,c4).
neighbor(d4,nw,c3). neighbor(e4,n,e3). neighbor(e4,ne,f3). neighbor(e4,e,f4).
neighbor(e4,se,f5). neighbor(e4,s,e5). neighbor(e4,sw,d5). neighbor(e4,w,d4).
neighbor(e4,nw,d3). neighbor(f4,n,f3). neighbor(f4,ne,g3). neighbor(f4,e,g4).
neighbor(f4,se,g5). neighbor(f4,s,f5). neighbor(f4,sw,e5). neighbor(f4,w,e4).
neighbor(f4,nw,e3). neighbor(g4,n,g3). neighbor(g4,ne,h3). neighbor(g4,e,h4).
neighbor(g4,se,h5). neighbor(g4,s,g5). neighbor(g4,sw,f5). neighbor(g4,w,f4).
```

```
neighbor(g4,nw,f3). neighbor(h4,n,h3). neighbor(h4,s,h5). neighbor(h4,sw,g5).
neighbor(h4,w,g4). neighbor(h4,nw,g3). neighbor(a5,n,a4). neighbor(a5,ne,b4).
neighbor(a5,e,b5). neighbor(a5,se,b6). neighbor(a5,s,a6). neighbor(b5,n,b4).
neighbor(b5,ne,c4). neighbor(b5,e,c5). neighbor(b5,se,c6). neighbor(b5,s,b6).
neighbor(b5,sw,a6). neighbor(b5,w,a5). neighbor(b5,nw,a4). neighbor(c5,n,c4).
neighbor(c5,ne,d4). neighbor(c5,e,d5). neighbor(c5,se,d6). neighbor(c5,s,c6).
neighbor(c5,sw,b6). neighbor(c5,w,b5). neighbor(c5,nw,b4). neighbor(d5,n,d4).
neighbor(d5,ne,e4). neighbor(d5,e,e5). neighbor(d5,se,e6). neighbor(d5,s,d6).
neighbor(d5,sw,c6). neighbor(d5,w,c5). neighbor(d5,nw,c4). neighbor(e5,n,e4).
neighbor(e5,ne,f4). neighbor(e5,e,f5). neighbor(e5,se,f6). neighbor(e5,s,e6).
neighbor(e5,sw,d6). neighbor(e5,w,d5). neighbor(e5,nw,d4). neighbor(f5,n,f4).
neighbor(f5,ne,g4). neighbor(f5,e,g5). neighbor(f5,se,g6). neighbor(f5,s,f6).
neighbor(f5,sw,e6). neighbor(f5,w,e5). neighbor(f5,nw,e4). neighbor(g5,n,g4).
neighbor(g5,ne,h4). neighbor(g5,e,h5). neighbor(g5,se,h6). neighbor(g5,s,g6).
neighbor(g5,sw,f6). neighbor(g5,w,f5). neighbor(g5,nw,f4). neighbor(h5,n,h4).
neighbor(h5,s,h6). neighbor(h5,sw,g6). neighbor(h5,w,g5). neighbor(h5,nw,g4).
neighbor(a6,n,a5). neighbor(a6,ne,b5). neighbor(a6,e,b6). neighbor(a6,se,b7).
neighbor(a6,s,a7). neighbor(b6,n,b5). neighbor(b6,ne,c5). neighbor(b6,e,c6).
neighbor(b6,se,c7). neighbor(b6,s,b7). neighbor(b6,sw,a7). neighbor(b6,w,a6).
neighbor(b6,nw,a5). neighbor(c6,n,c5). neighbor(c6,ne,d5). neighbor(c6,e,d6).
neighbor(c6,se,d7). neighbor(c6,s,c7). neighbor(c6,sw,b7). neighbor(c6,w,b6).
neighbor(c6,nw,b5). neighbor(d6,n,d5). neighbor(d6,ne,e5). neighbor(d6,e,e6).
neighbor(d6,se,e7). neighbor(d6,s,d7). neighbor(d6,sw,c7). neighbor(d6,w,c6).
neighbor(d6,nw,c5). neighbor(e6,n,e5). neighbor(e6,ne,f5). neighbor(e6,e,f6).
neighbor(e6,se,f7). neighbor(e6,s,e7). neighbor(e6,sw,d7). neighbor(e6,w,d6).
neighbor(e6,nw,d5). neighbor(f6,n,f5). neighbor(f6,ne,g5). neighbor(f6,e,g6).
neighbor(f6,se,g7). neighbor(f6,s,f7). neighbor(f6,sw,e7). neighbor(f6,w,e6).
neighbor(f6,nw,e5). neighbor(g6,n,g5). neighbor(g6,ne,h5). neighbor(g6,e,h6).
neighbor(g6,se,h7). neighbor(g6,s,g7). neighbor(g6,sw,f7). neighbor(g6,w,f6).
neighbor(g6,nw,f5). neighbor(h6,n,h5). neighbor(h6,s,h7). neighbor(h6,sw,g7).
neighbor(h6,w,g6). neighbor(h6,nw,g5). neighbor(a7,n,a6). neighbor(a7,ne,b6).
neighbor(a7,e,b7). neighbor(a7,se,b8). neighbor(a7,s,a8). neighbor(b7,n,b6).
neighbor(b7,ne,c6). neighbor(b7,e,c7). neighbor(b7,se,c8). neighbor(b7,s,b8).
neighbor(b7,sw,a8). neighbor(b7,w,a7). neighbor(b7,nw,a6). neighbor(c7,n,c6).
neighbor(c7,ne,d6). neighbor(c7,e,d7). neighbor(c7,se,d8). neighbor(c7,s,c8).
neighbor(c7,sw,b8). neighbor(c7,w,b7). neighbor(c7,nw,b6). neighbor(d7,n,d6).
neighbor(d7,ne,e6). neighbor(d7,e,e7). neighbor(d7,se,e8). neighbor(d7,s,d8).
neighbor(d7,sw,c8). neighbor(d7,w,c7). neighbor(d7,nw,c6). neighbor(e7,n,e6).
neighbor(e7,ne,f6). neighbor(e7,e,f7). neighbor(e7,se,f8). neighbor(e7,s,e8).
neighbor(e7,sw,d8). neighbor(e7,w,d7). neighbor(e7,nw,d6). neighbor(f7,n,f6).
neighbor(f7,ne,g6). neighbor(f7,e,g7). neighbor(f7,se,g8). neighbor(f7,s,f8).
neighbor(f7,sw,e8). neighbor(f7,w,e7). neighbor(f7,nw,e6). neighbor(g7,n,g6).
neighbor(g7,ne,h6). neighbor(g7,e,h7). neighbor(g7,se,h8). neighbor(g7,s,g8).
neighbor(g7,sw,f8). neighbor(g7,w,f7). neighbor(g7,nw,f6). neighbor(h7,n,h6).
neighbor(h7,s,h8). neighbor(h7,sw,g8). neighbor(h7,w,g7). neighbor(h7,nw,g6).
neighbor(a8,n,a7). neighbor(a8,ne,b7). neighbor(a8,e,b8). neighbor(b8,n,b7).
neighbor(b8,ne,c7). neighbor(b8,e,c8). neighbor(b8,w,a8). neighbor(b8,nw,a7).
neighbor(c8,n,c7). neighbor(c8,ne,d7). neighbor(c8,e,d8). neighbor(c8,w,b8).
neighbor(c8,nw,b7). neighbor(d8,n,d7). neighbor(d8,ne,e7). neighbor(d8,e,e8).
neighbor(d8,w,c8). neighbor(d8,nw,c7). neighbor(e8,n,e7). neighbor(e8,ne,f7).
neighbor(e8,e,f8). neighbor(e8,w,d8). neighbor(e8,nw,d7). neighbor(f8,n,f7).
```

```
neighbor(f8,ne,g7). neighbor(f8,e,g8). neighbor(f8,w,e8).  neighbor(f8,nw,e7).
neighbor(g8,n,g7). neighbor(g8,ne,h7).  neighbor(g8,e,h8). neighbor(g8,w,f8).
neighbor(g8,nw,f7).  neighbor(h8,n,h7). neighbor(h8,w,g8). neighbor(h8,nw,g7).


%  Player facts
player(x).
player(o).
opponent(x, o).
opponent(o, x).

%  Goal information
zenith_goal(win(x)).

%  These are used to instantiate operators for regress_conditions.
%  There should be one fact for each agent.
agent_operator(move(x, _Move)).
agent_operator(move(o, _Move)).



%  Goal regression information.  Format is:
%
%     PREIMAGE(+Condition, +Operation, -Preimage)
%
%  Regress.pl uses these preimage facts.
%
%  The simplifier understands the basic form of
%  Prolog if-then and if-then-else constructs, eg
%     p(X) -> q(X) | r(X)
%  becomes
%     (p(X) ^ q(X)) | (\+p(X) ^ r(X))
%
%  This translation is inaccurate if p is not determinate.
%

%  OWNS
preimage(owns(Player,Square),
         move(MP,MS),                          % Move player, Move square
         (
         %% Case 1: Player is the Move Player
         Player=MP ->
          ( Square=MS -> legal_move(Square,Player)
          | otherwise ->
                ( owns(Player,Square)      % Either Player owned it already
                | (opponent(Player,Opp),   %  or the opponent owned it and
                   owns(Opp,Square),       %   it was flipped.
                   bs(MS,S2,Player),
                   in_line(Square,MS,S2)
                  )
                )
```

```
        )
       %%  Case 2:  Player is NOT the move player
        | opponent(Player, MP) ->
             ( owns(Player,Square),
               \+((bs(MS,S2,MP), in_line(Square,MS,S2)))
             )
        )
        ).   % end of OWNS preimage

%  A blank remains a blank only if it was blank before and
%   the move was not made into it.
preimage(blank(S), move(_,MS), (blank(S), MS\==S) ).

%  Preimages of SPAN(Begin,End,Dir,Owner)
% Case 1: Mover is the Owner
preimage(span(Begin,End,Dir,Owner),
         move(Mover,MS),
         ( Owner=Mover ->
              ( span_with_hole(Begin,End,Dir,Owner,Mover,MS)
              | span_with_subspan(Begin,End,Dir,Owner,Mover,MS)
              | span(Begin,End,Dir,Owner)
              )
         | otherwise ->  span(Begin,End,Dir,Owner)
         )).

%  Almost the preconditons for owns(Owner,Square)
hole(Square, Owner, MoveSquare) :-
      ( Square=MoveSquare,
        legal_move(Square, _Player)
      )
  ; ( opponent(Owner, Opp),
      owns(Opp, Square),
      bs(MS, S2, Owner),
      in_line(Square, MS, S2)
  ).

%  Same as span but can be of length zero.
span_star(Begin, End, Dir, Owner) :-
        ( span(Begin, End, Dir, Owner) | Begin=End).

span_with_hole(Begin, End, Dir, Owner, Mover, MS) :-
        span_star(Begin, Hole, Dir, Owner),
        hole(Hole, Mover, MS),
        neighbor(Hole, Dir, Next),
        span_star(Next, End, Dir, Owner).

span_with_subspan(Begin, End, Dir, Owner, Mover, MS) :-
        span_star(Begin, MS, Dir, Owner),
        bs(MS, BSend, Mover),
        span_star(BSend, End, Dir, Owner).
```

```
/************************************************************
  OWNS(Player, Square) and BLANK(Square).

  These are operational predicates, and their definitions shouldn't be
  in here because they refer to the implementation of boards as
  vectors.  They are in this file so that compile_dt will attach
  counters to them; they are operational and should be counted.
************************************************************/
owns(Player, Square) :-
        player_nc(Player),
        square_nc(Square),
        squarenum(Square, Num),
        current_board(Board),
        vector_element(Num, Board, Ownernum),
        ownernum(Player, Ownernum).

blank(Square) :-
        square_nc(Square),
        current_board(Board),
        squarenum(Square, Num),
        vector_element(Num, Board, 0).
```

APPENDIX B

OTHELLO FEATURES

This is a listing of the OTHELLO features generated by Zenith. A description of the feature formalism appears in Section 5.3.1. The full OTHELLO problem specification appears in Appendix A. For convenience, here is a short description of each relevant predicate:

`\+(p(X))` - `p(X)` is not true (cannot be proved)

`forall(p(X),q(X))` - $\forall X p(X) \rightarrow q(X)$

`square(X)` - `X` is a square

`blank(Sq)` - `Sq` is blank (unoccupied) square.

`bs(Sq1,Sq2,Player)` - There is a bracketed span of discs from `Sq1` to `Sq2`. A bracketed span is a blank at `Sq1` followed by an unbroken span of discs owned by `Player`'s opponent, terminated by a `Player` disc at `Sq2`

`direction(X)` - `X` is a direction; one of {N,NE,E,SE,S,SW,W,NW}

`in_line(Sq, Begin,End)` - Square `Sq` is in a line between `Begin` and `End`

`legal_move(Sq,Player)` - Square `Sq` is a legal move for `Player`

`line(Begin,End,Dir)` - There is a line of squares from `Begin` to `End` in direction `Dir`

`neighbor(Sq1, Dir, Sq2)` - The neighbor of square `Sq1` in direction `Dir` is `Sq2`.

`opponent(X,Y)` - `X` is the opponent of `Y`

`owns(Player,Sq)` - `Player` owns square `Sq`

`player(X)` - `X` is a player (`white` or `black`)

`score(P, S)` - Player `P` owns `S` discs

`span(Sq1,Sq2,Dir,Player)` - There is an unbroken span of `Player`'s discs from `Sq1` to `Sq2` in direction `Dir`

`win(P)` - Player `P` has won

The Othello Features

```
%  PIECES FOR WHITE
f2(C) :- count([B],(opponent(x,A),owns(A,B)),C).


%%  PIECES for BLACK
f3(B) :- count([A],owns(x,A),B).


%%  MOVES FOR BLACK
f9(B) :- count([A],legal_move(A,x),B).
```

```
%%  SEMI-STABLE PIECES FOR WHITE
f6(E) :- count([B],(opponent(x,A),
                    A\==x,
                    opponent(A,x),
                    owns(A,B),
                    \+(bs(C,D,x),in_line(B,C,D))),
               E).



%%  SEMI-STABLE PIECES FOR BLACK
f6(D) :- count([A],(owns(x,A),
                    \+(bs(B,C,o), in_line(A,B,C))),D).



%%  BLACK PIECES OWNED OR FLIPPABLE
f10(E) :- count([D,A,C],((owns(x,A)
                        ; opponent(x,B),
                          owns(B,A),
                          bs(C,D,x),
                          in_line(A,C,D)
                         ),
                         A\==C),
                E).


%%  AXES FOR BLACK
f14(D) :- count([A,B,C],(owns(x,A),
                         bs(B,C,o),
                         in_line(A,B,C)),
                D).



%%  AXES FOR WHITE
f15(E) :- count([D,B,C],(opponent(x,A),
                         owns(A,B),
                         bs(C,D,x),
                         B\==C,
                         in_line(B,C,D)),
                E).
```

```
%%  AASS BLACK PIECES
%% Testing values of [B] such that
%%     owns(x,B)  ===>  \+ (bs(C,D,o),in_line(B,C,D))
%%
%% 64 values seen: 10 invariants, 54 variants, I/Total=0.15625
%% Creating class1/1 for invariants of this feature.
%% Values for class1/1 are:
%% [[a1],[a2],[a8],[b1],[b8],[g1],[g8],[h1],[h2],[h8]]
%%
f27(B) :- count([A],(owns(x,A),class1(A)),B).



%%  AASS SQUARES CAPTURABLE BY OPPONENT
f39(B) :- count([A],(legal_move(A,x),class1(A)),B).



%%  PREDECESSOR OF FRONTIER DIRECTIONS
f49(F) :- count([C],(square(A),
                     blank(A),
                     opponent(x,B),
                     direction(C),
                     neighbor(A,C,D),
                     owns(B,D),
                     neighbor(D,C,E),
                     owns(x,E)),
                F).



%%  PIECES OWNED BY WHITE OR SEMI-UNSTABLE BLACK
f107(E) :- count([A],((owns(o,A)
                     ; opponent(o,B),
                       owns(B,A),
                       bs(C,D,o),
                       in_line(A,C,D)
                      ),
                      A\==C),
                   E).



%%  NO MOVES
f11(A) :- count([],end_of_game,A).
```

```
%%  SIMILAR TO ROSENBLOOM FRONTIER
f89(F) :- count([A],(square(A),
                     blank(A),
                     opponent(x,B),
                     direction(C),
                     neighbor(A,C,D),
                     owns(B,D),
                     neighbor(D,C,E)),
                     F).
```

```
%%   SIMILAR TO ROSENBLOOM EMPTY
f107(G) :- count([E],(square(A),
                      blank(A),
                      opponent(x,B),
                      direction(C),
                      neighbor(A,C,D),
                      span(D,E,C,B),
                      neighbor(E,C,F)),
                 G).
```

```
%%   SIMILAR TO ROSENBLOOM SUM EMPTY
f21(F) :- count([A,C],(square(A),
                       blank(A),
                       opponent(x,B),
                       direction(C),
                       neighbor(A,C,D),
                       owns(B,D),
                       neighbor(D,C,E)),
                  F).
```

APPENDIX C

THE TNM DOMAIN THEORY

```
/*************************************************************************

  TNM_DT.PL - Domain theory for Telecommunications Network Management
              in Prolog.

        Tom Fawcett (fawcett@cs.umass.edu)
        COINS Deptartment, LGRC
        University of Massachusetts
        Amherst, MA  10373
        June, 1991.


A network is described statically using the following facts:
   switch(S)
   trunk(Trunk)
   links(Trunk, Switch1, Switch2)
   capacity(Trunk, N)
   routing_table(Switch, Dest, Trunk)

These are set up by network definition files, eg ils10.pl.
These generally do not change within Zenith runs.

The following dynamic facts are state dependent:
   control_in_effect(State, Control)
   switch_originations(State, Switch, N)
   switch_completions(State, Switch, N)
   switch_demand(State, Switch, Dest, N)
   trunk_usage(State, Trunk, N)
   trunk_attempts(State, Trunk, N)
   trunk_completions(State, Trunk, N)
   total_calls_attempted(State, N)
   total_calls_completed(State, N)
   throughput(State, Throughput)


See the companion file tnm-simulator.pl to see how networks are simulated
and how these values are computed.
```

```
**************************************************************************/
:- mode
        min2(+, +, -),
        \+(+).
operational(min2).
operational(\+).

min2(X, Y, X) :- X =< Y.
min2(X, Y, Y) :- Y < X.


%  Modes of predicates not defined here.
:- mode
        switch(?),
        trunk(?),
        capacity(?, ?),
        routing_table(?, ?, ?),
        control_in_effect(?).


%********** Operationality information **********
operational(switch).
operational(trunk).
operational(capacity).
operational(routing_table).
operational(control_in_effect).

/********** Declaration of state-specific predicates **********/
state_specific_pred(control_in_effect).



%  CONNECTS(?Trunk, ?Switch1, ?Switch2)
%
%  Traffic can flow from Switch1 to Switch2.
%  CONNECTS is not symmetric because of the existence of 1-way trunks.
%
:- mode connects(?, ?, ?).
operational(connects).

connects(T, X, Y) :-
        ( nonvar(T) ->
              ( one_way(T, X, Y) -> true
              | (links(T, X, Y) | links(T, Y, X))
              )
        | one_way(T, X, Y) | links(T, X, Y) | links(T, Y, X)
        ).


/*************************************************************************
  Specifications of switch controls
```

```
*************************************************************************/
:- no_style_check(discontiguous).

/************************** The DRR control  ******************************/
operator(drr(_Switch, _Dest, _Trunk, _Prob)).

preconditions(drr(Switch, Dest, Trunk, Prob),
              ( switch(Switch),
                connects(Trunk, Switch, _),
                switch(Dest),
                Switch \== Dest,
                drr_probability(Prob),
                \+(control_in_effect(drr(Switch, Dest, _, _))),
                \+(control_in_effect(cb(Switch, Dest)))
              ) ).

%  There are a fixed set of probabilities that a DRR can use.
%  These were adopted MacLearn.
drr_probability(0.25).
drr_probability(0.50).
drr_probability(1.00).

%  Note that this does not include would_cause_circular_routing.
%  The performance system is now allowed to create circular routes because
%  another control may make them non-circular.
heuristically_useful(drr(Switch, Dest, _Trunk, _Prob)) :-
        switch_demand(Switch, Dest, Demand),
        Demand > 0,
        before_overflow(Switch, Dest),
        !.

%%  These are independent of controls:
%  SWITCH ORIGINATIONS
%      Originations are created externally - can't be affected by operators.
preimage(switch_originations(A, B), _, switch_originations(A, B)).

%  CONTROL_IN_EFFECT
%      The preimage of this condition is exactly the set of preconditions
%      of the operator.   I don't know if we really need this.
preimage(control_in_effect(Control),
         Control,
         Preconditions) :-
        preconditions(Control, Preconditions).


/********************    Preimages of DRR    ********************/
%  SWITCH COMPLETIONS
preimage( switch_completions(Dest, Completions),
          drr(DRRswitch, Dest, Trunk, _Prob),
          ( DRRswitch = Dest ->
                %%  A DRR has no effect on the switch_completions of the
```

```
            %%  switch where it is placed.
            switch_completions(Dest, Completions)
        | otherwise ->
            %%  Assume that the number of completions increases by the
            %%  traffic through the reroute minus the amount that
            %%  would have completed otherwise (the "actual
            %%  traffic").  This assumes that the other traffic is not
            %%  affected by the reroute.
            switch(DRRswitch),
            switch(Dest),
            connects(Trunk, DRRswitch, _),
            actual_traffic(DRRswitch, Dest, Dest, OldTrafficFromSwitch),
            traffic_through_reroute(DRRswitch, Dest, Dest, Trunk,
                                    NewTrafficFromSwitch),
            switch_completions(Dest, ExistingCompletions),
            Completions is ExistingCompletions - OldTrafficFromSwitch
                                               + NewTrafficFromSwitch
        )
        ).

%  SWITCH DEMAND
%    is like completions except it represents all calls coming through the
%    switch, not just those completing there.
preimage( switch_demand(SDswitch, Demand),
        drr(DRRswitch, Dest, Trunk, _Prob),
        ( DRRswitch = Dest ->
            %%  A DRR has no effect on the switch_completions of the
            %%  switch where it is placed.
            switch_demand(SDswitch, Demand)
        | otherwise ->
            %%  Assume that the switch demand increases by the
            %%  traffic through the reroute minus the amount that
            %%  would have completed otherwise (the "actual
            %%  traffic").  This assumes that the other traffic is not
            %%  affected by the reroute.
            switch(DRRswitch),
            switch(Dest),
            switch(SDswitch),
            connects(Trunk, DRRswitch, _),
            actual_traffic(DRRswitch, SDswitch, Dest, OldTraffic),
            traffic_through_reroute(DRRswitch, SDswitch, Dest, Trunk,
                                    NewTraffic),
            switch_demand(SDswitch, ExistingDemand),
            Demand is ExistingDemand - OldTraffic + NewTraffic
        )
        ).

/*
%  TRUNK ATTEMPTS
%     represents all calls trying to use the trunk
preimage( trunk_attempts(Trunk, Attempts),
```

```
          drr(DRRswitch, Dest, DRRTrunk, _Prob),
          ( DRRswitch = Dest ->
               %% check this
               switch_demand(SDswitch, Demand)
          | otherwise ->
               switch(DRRswitch),
               switch(Dest),
               connects(DRRTrunk, DRRswitch, _),
               actual_traffic(DRRswitch, S2, Dest, OldTraffic),
               routed_trunk(S2, Dest, Trunk),
               trunk_attempts(Trunk, OldAttempts),
               traffic_through_reroute(DRRswitch, S2, Dest, DRRTrunk,
                                       NewTraffic),
               Attempts is OldAttempts - OldTraffic + NewTraffic
          )
          ).

%   trunk_completions(State, Trunk, N)
*/

/*************************** The CB control ******************************/

/*operator( cb(_Switch, _Dest)). */

preconditions(cb(Switch, Dest),
              ( switch(Switch),
                switch(Dest),
                Switch \== Dest,
                \+(control_in_effect(cb(Switch, Dest))),
                \+(control_in_effect(drr(Switch, Dest, _Trunk, _Prob)))
              ) ).

/******    Preimages of CB   *******/
%  SWITCH COMPLETIONS
preimage( switch_completions(SCswitch, Completed),
          cb(Switch, Dest),
          (
          Switch = SCswitch ->
               %%  If this control is being placed on the destination
               %%  affected, the control will have no effect.
               switch_completions(SCswitch, Completed)
          | SCswitch \== Dest ->
               %%  Assume the control will have no affect on any but the
               %% destination switch.  This is a first-order approximation.
               switch_completions(SCswitch, Completed)
          | otherwise ->
               %%  Estimate the number of completed calls that would have
               %%  reached the destination before the code block.
               %%  Naively, this is simply the number of calls offered to
               %%  the destination (ie, that were affected by the code block).
               actual_traffic(Switch, Dest, Dest, OldTrafficFromSwitch),
```

```
                switch_completions(Dest, ExistingCompleted),
                Completed is ExistingCompleted - OldTrafficFromSwitch
          )
          ).

%  SWITCH DEMAND
%    is like completions except it represents all calls coming through the
%    switch, not just those completing there.
preimage( switch_demand(SDswitch, Demand),
          cb(Switch, Dest),
          (
          Switch = SDswitch ->
               %%  If this control is being placed on the destination
               %%  affected, the control will have no effect.
               switch_demand(SDswitch, Demand)
          | otherwise ->
               %%  Estimate the number of completed calls that would have
               %%  flowed through SDswitch before the code block.
               actual_traffic(Switch, SDswitch, Dest, OldTraffic),
               switch_demand(Dest, ExistingDemand),
               Demand is ExistingDemand - OldTraffic
          )
          ).

heuristically_useful(cb(Switch, Dest)) :-
        switch_demand(Switch, Dest, Demand),
        Demand > 0,
        before_overflow(Switch, Dest),
        !.


/*  Glenn's BEFORE OVERFLOW heuristic.

        "The switch where the control is placed must be such that there occurs
        at least one overflowing trunk on the path from that switch to the
        destination.  Otherwise the control will not affect overflowing
        traffic."
*/
before_overflow(Switch, Dest) :-
        trunk_on_path(Switch, Dest, OverflowingTrunk),
        overflowing(OverflowingTrunk),
        !.                                          % force deterministic

impose_controls(Control, Net) :-
        \+list(Control),
        !,
        impose_control(Control, Net).
impose_controls([], _).
impose_controls([C|Rest], Net) :-
        impose_control(C, Net),
        impose_controls(Rest, Net).
```

```
% This was added for the ILS: none = [] = nothing.
impose_control(none, Net) :- impose_control(nothing, Net).
% The control 'nothing' is known specially.
impose_control(Control, Net) :-
        ( operator(Control) -> preconditions(Control, Preconditions)
        | Control == nothing -> Preconditions = true
        ),
        current_net(CN),
        ( CN \== Net -> set_current_net(Net)
        ; true),
        ( call(Preconditions) ->  set_current_net(CN)
        ; set_current_net(CN),
          format('~N***** Error in impose_control: ~w is illegal~n',
                 [Control]),
          fail ),
        assert(control_in_effect(Net, Control)),
        ( CN == Net -> assert(control_in_effect(Control))
        ; true).

/*************************************************************************
  ROUTED_TRUNK(+Switch, +Dest, +Trunk)
     For a call at Switch going to Dest, Trunk is the next trunk that
     should be taken.
     This does not catch Switch=Dest.
*************************************************************************/
:- mode  routed_trunk(+, +, ?).
operational(routed_trunk).

routed_trunk(Switch, Dest, Trunk) :-
        ( control_in_effect(drr(Switch, Dest, _, _)) ->
             control_in_effect(drr(Switch, Dest, Trunk, _))
        | control_in_effect(cb(Switch, Dest)) ->
             fail
        | routing_table(Switch, Dest, Trunk)
        ).

% For convenience, this gives you the next trunk and its connected switch.
:- mode  routed_trunk_and_switch(+, +, -, -).
operational(routed_trunk_and_switch).

routed_trunk_and_switch(Switch, Dest, Trunk, NextSwitch) :-
        routed_trunk(Switch, Dest, Trunk),
        connects(Trunk, Switch, NextSwitch).

trunk_on_path(From, To, Trunk) :-
        trunk_on_path(From, To, Trunk, []).

trunk_on_path(Switch, Switch, _, _) :- !, fail.          % No such trunk
trunk_on_path(Switch, Dest, Trunk, TrunksSoFar) :-
        routed_trunk(Switch, Dest, Trunk),
```

```
        \+memberchk(Trunk, TrunksSoFar).
trunk_on_path(Switch, Dest, Trunk, TrunksSoFar) :-
        routed_trunk(Switch, Dest, NextTrunk),
        \+memberchk(NextTrunk, TrunksSoFar),      % Detects cycles
        connects(NextTrunk, Switch, NextSwitch),
        trunk_on_path(NextSwitch, Dest, Trunk, [NextTrunk|TrunksSoFar]).

overflowing(Trunk) :-
        trunk(Trunk),
        trunk_attempts(Trunk, Attempts),
        trunk_completions(Trunk, Completions),
        Attempts > Completions.


%  This is separate so that it can be atomic.
:- mode cycle_free_route(?, ?, ?).
operational(cycle_free_route).

cycle_free_route(From, To, Dest) :-
        \+ check_routing_circularity(From, To, Dest, nothing).


:- mode cycle_in_route(?, ?, ?).
operational(cycle_in_route).

cycle_in_route(From, To, Dest) :-
        check_routing_circularity(From, To, Dest, nothing).

check_routing_circularities(Control) :-
        switch(Source),
        switch(Dest),
        Source \== Dest,
        check_routing_circularity(Source, Dest, Control, []).

check_routing_circularity(Dest, Dest, _, _) :- !, fail.
check_routing_circularity(Switch, _, _, SwitchesVisited) :-
        memberchk(Switch, SwitchesVisited),
        !.
check_routing_circularity(Switch, Dest, Control, Visited) :-
        routed_trunk_crc(Switch, Dest, Control, NextTrunk),
        connects(NextTrunk, Switch, NextSwitch),
        check_routing_circularity(NextSwitch, Dest, Control,
                                  [Switch|Visited]).

%  Special version for use with check_routing_circularity.
%  This returns the routed trunk assuming that Control is in effect.
routed_trunk_crc(Switch, Dest, drr(Switch, Dest, Trunk, _), Trunk) :-
        !.
routed_trunk_crc(Switch, Dest, _, Trunk) :-
        routed_trunk(Switch, Dest, Trunk).
```

```
%  Zenith's goal for TNM
%  Maximize is a meta-predicate
zenith_goal(maximize(throughput(_X)) ).

zenith_operator(drr(_Switch, _Dest, _Trunk, _Prob)).
zenith_operator( cb(_Switch, _Dest)).

%  All operators are executable by the agent (Zenith)
agent_operator(X) :- operator(X).

:- mode throughput(-).
throughput(X) :-
        total_attempted(A),
        total_completed(C),
        X is C/A.

:- mode total_completed(-).
total_completed(X) :-
        count([S,v(NCalls)],
              (switch(S), switch_completions(S, NCalls)),
              X).

:- mode total_attempted(-).
total_attempted(X) :-
        count([S,v(NCalls)],
              (switch(S), switch_originations(S, NCalls)),
              X).

:- mode switch_completions(?, ?).
operational(switch_completions).
state_specific_pred(switch_completions).
switch_completions(S, C) :-
        current_net(Net),
        switch_completions(Net, S, C).

:- mode  switch_originations(?, ?).
operational(switch_originations).
state_specific_pred(switch_originations).
switch_originations(S, O) :-
        current_net(Net),
        switch_originations(Net, S, O).

:- mode switch_demand(?, ?, ?).
operational(switch_demand).
state_specific_pred(switch_demand).
switch_demand(S, Dest, Demand) :-
        current_net(Net),
        switch_demand(Net, S, Dest, Demand).
```

```
:- mode trunk_usage(?, ?).
operational(trunk_usage).
state_specific_pred(trunk_usage).
trunk_usage(T, U) :-
        current_net(Net),
        trunk_usage(Net, T, U).

:- mode trunk_attempts(?, ?).
operational(trunk_attempts).
state_specific_pred(trunk_attempts).
trunk_attempts(T, A) :-
        current_net(Net),
        trunk_attempts(Net, T, A).

:- mode trunk_completions(?, ?).
operational(trunk_completions).
state_specific_pred(trunk_completions).
trunk_completions(T, C) :-
        current_net(Net),
        trunk_completions(Net, T, C).

/*****************************************************************************
  ACTUAL_TRAFFIC(+From, +To, +Dest, -Traffic)

  The number of calls that make it (unidirectionally)
  between From and To while going to Dest.  If To=Dest,
  this is is the number of calls completed from From.
  This depends on the actual statistics of the current state.
*****************************************************************************/
:- mode actual_traffic(+, +, +, -),
        actual_traffic(+, +, +, +, -).

actual_traffic(From, To, Dest, 0) :- cycle_in_route(From, To, Dest).
actual_traffic(From, To, Dest, N) :-
        cycle_free_route(From, To, Dest),
        switch_demand(From, Dest, Demand),
        actual_traffic(From, To, Dest, Demand, N).

actual_traffic(To, To, _, Demand, Demand).  % Traffic has arrived.
actual_traffic(From, To, Dest, Demand, N) :-
        From \== To,
        routed_trunk_and_switch(From, Dest, NextTrunk, NextSwitch),
        trunk_completion_ratio(NextTrunk, Ratio),
        SurvivingDemand is integer(Demand * Ratio),
        actual_traffic(NextSwitch, To, Dest, SurvivingDemand, N).


%  This is the amount of traffic from Source to Dest that passes
%  through Trunk.
:- mode actual_traffic_through_trunk(+, +, +, -).
actual_traffic_through_trunk(From, Dest, _, 0) :-
```

```
        cycle_in_route(From, Dest, Dest).
actual_traffic_through_trunk(Source, Dest, Trunk, TrafficThroughTrunk) :-
        cycle_free_route(Source, Dest, Dest),
        switch_demand(Source, Dest, Traffic),
        actual_traffic_through_trunk(Source, Dest, Trunk, Traffic,
                                     TrafficThroughTrunk).

:- mode actual_traffic_through_trunk(+, +, +, +, -).
%----  If we've reached the destination without seeing Trunk, return zero.
actual_traffic_through_trunk(Dest, Dest, _, _, 0) :- !.
%----  If this is the trunk we're interested in.
actual_traffic_through_trunk(Source, Dest, Trunk, Demand,
                             TrafficThroughTrunk) :-
        Source \== Dest,
        routed_trunk(Source, Dest, Trunk),
        trunk_completion_ratio(Trunk, Ratio),
        TrafficThroughTrunk is integer(Demand * Ratio).
%----  Otherwise keep going toward destination.
actual_traffic_through_trunk(Source, Dest, Trunk, Demand,
                             TrafficThroughTrunk) :-
        Source \== Dest,
        routed_trunk_and_switch(Source, Dest, NextTrunk, NextSwitch),
        NextTrunk \== Trunk,
        trunk_completion_ratio(NextTrunk, Ratio),
        SurvivingDemand is integer(Demand * Ratio),
        actual_traffic_through_trunk(NextSwitch, Dest, Trunk,
                                     SurvivingDemand,
                                     TrafficThroughTrunk).


/*************************************************************************
  TRUNK_COMPLETION_RATIO(?Trunk, ?Ratio)
  Ratio is the completion ratio (completed/attempted) of Trunk.
*************************************************************************/
:- mode trunk_completion_ratio(?, ?).
operational(trunk_completion_ratio).

trunk_completion_ratio(Trunk, Ratio) :-
        trunk_attempts(Trunk, Attempts),
        trunk_completions(Trunk, Completions),
        ( Attempts =:= 0 ->
              Ratio=1.0
        | Ratio is Completions / Attempts
        ).


/*************************************************************************
  TRAFFIC_THROUGH_REROUTE(+Source, +To, +Dest, +RerouteTrunk, -Traffic)

  Binds Traffic to the amount of traffic that would make it from
  Source to To trying to get to Dest if it took RerouteTrunk.
```

```
  It is assumed that RerouteTrunk is *not* the trunk that actually
  would be taken, else this probably returns something wrong.
  It is also assumed that RerouteTrunk does not introduce
  circular routing; if it does, then traffic_through_reroute will return
  a strange number.

  To and Dest are separate arguments so that this predicate can be calculate
  traffic through an intermediate node that is not the destination.
**************************************************************************/
:- mode traffic_through_reroute(+, +, +, ?, -).

%%%%% Traffic will be zero if there is no traffic for Dest through this node.
traffic_through_reroute(Source, _, Dest, _, 0) :-
        switch_demand(Source, Dest, 0),
        !.
traffic_through_reroute(Source, To, Dest, Trunk, 0) :-
        connects(Trunk, Source, NextSwitch),
        cycle_in_route(NextSwitch, To, Dest),
        !.
%%%%% Else compute the first link of the reroute and call
%%%%% traffic_through_reroute_2 to do the rest.
traffic_through_reroute(Source, To, Dest, Trunk, Traffic) :-
        connects(Trunk, Source, NextSwitch),
        \+routed_trunk(Source, Dest, Trunk),
        cycle_free_route(NextSwitch, To, Dest),
        switch_demand(Source, Dest, Demand),
        trunk_completion_ratio(Trunk, Ratio),
        SurvivingDemand is integer(Demand * Ratio),
        traffic_through_reroute2(Source, NextSwitch, To, Dest,
                                 SurvivingDemand, Traffic).

/**************************************************************************
  TRAFFIC_THROUGH_REROUTE2(+Source, +Switch, +To, +Dest, +Demand, -Traffic)

  Determines the amount of traffic flowing from Source to To on its way to
  Dest.  Switch is where we currently are, and Demand is the demand
  at Switch.  Traffic is the final output.

  Note that this is inefficient because it calls actual_traffic_through_trunk
  once per recursive call, which has to calculate the traffic along the
  path up to the trunk.
**************************************************************************/
:- mode traffic_through_reroute2(+, +, +, +, +, -).
%====  If this is the To node,
traffic_through_reroute2(_, To, To, _, Traffic, Traffic).
%====  Recursive case, if Switch \== To.
traffic_through_reroute2(Source, Switch, To, Dest, Demand, Traffic) :-
        Switch \== To,
        routed_trunk_and_switch(Switch, Dest, NextTrunk, NextSwitch),
        capacity(NextTrunk, Capacity),
        trunk_attempts(NextTrunk, Attempts),
```

```
trunk_completions(NextTrunk, Completions),
actual_traffic_through_trunk(Source, Dest, NextTrunk, OldTraffic),
NewAttempts is Attempts - OldTraffic + Demand,
( NewAttempts =:= 0 ->
      SurvivingDemand = Demand
;     SurvivingDemand is
          integer(Demand * (Completions / NewAttempts))),
%  I don't know if we need to do this:
min2(SurvivingDemand, Capacity, SurvivingDemand1),
traffic_through_reroute2(Source, NextSwitch, To, Dest,
                         SurvivingDemand1, Traffic).
```

APPENDIX D

TNM FEATURES

This is a listing of the TNM features generated by Zenith. The feature formalism is described in Section 5.3.1. The problem specification appears in Appendix C. For convenience, here is a short description of each relevant predicate:

- **State-independent operational predicates:**

  switch(S) - S is a switch.

  trunk(Trunk) - T is a trunk.

  connects(Trunk, Switch1, Switch2) - Switch1 and Switch2 are directly connected via trunk group Trunk.

  capacity(Trunk, N) - Trunk's capacity is N simultaneous calls.

  routing_table(Switch, Dest, Trunk) - Routing table entry: A call to Dest arriving at Switch should be routed via Trunk.

- **State-dependent operational predicates.** These are all with respect to the current state:

  routed_trunk(Switch, Dest, Trunk) - A call with destination Dest arriving at Switch will be routed onto Trunk.

  switch_originations(Switch, N) - Switch has originated N calls.

  switch_completions(Switch, N) - N calls have completed to Switch.

  switch_demand(Switch, Dest, N) - N calls have arrived at Switch with destination Dest.

  trunk_usage(Trunk, N) - Trunk has successfully handled N calls.

  trunk_attempts(Trunk, N) - N calls have tried to use Trunk.

  trunk_completions(Trunk, N) - N calls have been allocated successfully on Trunk.

  total_calls_attempted(N) - N total calls were placed.

  total_calls_completed(N) - N total calls were completed.

  throughput(Throughput) - The throughput of the network in this state is Throughput.

  trunk_completion_ratio(Trunk, Ratio) - Ratio is the number of completions divided by the number of attempts. If attempts=0, Ratio=1.

  cycle_free_route(From, To, Dest) - Calls from From bound ultimately for Dest will pass through To without encountering a cycle.

  cycle_in_route(From, To, Dest) - The opposite of cycle_free_route.

- **Non-operational predicates:**

\+p(X) - p(X) is not true (cannot be proved)

forall(p(X),q(X)) - $\forall X p(X) \rightarrow q(X)$

trunk_on_path(From, To, Trunk) - Trunk is on the path from switch From to switch To.

overflowing(Trunk) - Trunk is overflowing (attempts greater than completions).

actual_traffic(From, To, Dest, Traffic) - Traffic is the number of calls that make it (unidirectionally) between From and To while going to Dest. This depends on the actual statistics of the current state. If To=Dest, this is is the number of calls completed from From.

traffic_through_reroute(Source, To, Dest, RerouteTrunk, Traffic) - Traffic is the amount of traffic that would make it from Source to To trying to get to Dest if it took RerouteTrunk. It is assumed that RerouteTrunk does not introduce circular routing.

traffic_through_reroute2(Source, Switch, To, Dest, Demand, Traffic) - Traffic is the amount of traffic flowing from Source to To on its way to Dest. Demand is the existing demand at Switch.

```
%  THROUGHPUT
f1(D) :- count([v(C)],(total_attempted(A),
                       total_completed(B),
                       C is B/A),
               D).



%  CALL ATTEMPTS
f2(C) :- count([A,v(B)],(switch(A),
                         switch_originations(A,B)),
               C).



%  CALL COMPLETIONS
f3(C) :- count([A,v(B)],(switch(A),switch_completions(A,B)),C).
```

```
% COMPLETIONS ACHIEVABLE BY REROUTES
f5(I) :- count([v(H),D,B,C,A,H,G,E,F],
               (switch(A),
                switch(B),
                B\==A,
                connects(C,B,D),
                actual_traffic(B,A,A,E),
                traffic_through_reroute(B,A,A,C,F),
                switch_completions(A,G),
                H is G-E+F
                ),
              I).


% TRAFFIC THROUGH REROUTES
f6(H) :- count([v(F)],(switch(A),
                       switch(B),
                       B\==A,
                       connects(C,B,D),
                       actual_traffic(B,A,A,E),
                       traffic_through_reroute(B,A,A,C,F),
                       switch_completions(A,G)),
                H).


%  ALL PAIRS, ALL SWITCH COMPLETIONS
f8(H) :- count([v(G)],(switch(A),
                       switch(B),
                       B\==A,
                       connects(C,B,D),
                       actual_traffic(B,A,A,E),
                       traffic_through_reroute(B,A,A,C,F),
                       switch_completions(A,G)),
                H).


% TRAFFIC BETWEEN PAIRS
f7(H) :- count([v(E)],(switch(A),
                       switch(B),
                       B\==A,
                       connects(C,B,D),
                       actual_traffic(B,A,A,E),
                       traffic_through_reroute(B,A,A,C,F),
                       switch_completions(A,G)),
                H).
```

```
% TOTAL COMPLETIONS
f8(H) :- count([v(G)], (switch(A),
                        switch(B),
                        B\==A,
                        connects(C,B,D),
                        actual_traffic(B,A,A,E),
                        traffic_through_reroute(B,A,A,C,F),
                        switch_completions(A,G)),
                H).



% COMPLETIONS REROUTING THROUGH NON-DEFAULT TGs
f22(M) :- count([D,L,K,E,C,G,H,B,F,A,I,J,v(L)],
                (switch(A),
                 switch(B),
                 B\==A,
                 connects(C,B,D),
                 actual_traffic(B,A,A,E),
                 connects(C,B,F),
                 \+routed_trunk(B,A,C),
                 cycle_free_route(F,A,A),
                 switch_demand(B,A,G),
                 trunk_completion_ratio(C,H),
                 I is integer(G*H),
                 traffic_through_reroute2(B,F,A,A,I,J),
                 switch_completions(A,K),
                 L is K-E+J),
                M).



% TRAFFIC BETWEEN CYCLE-FREE PAIRS
f16(I) :- count([B,A,E,F,v(G)],(switch(A),
                                switch(B),
                                B\==A,
                                connects(C,B,D),
                                cycle_free_route(B,A,A),
                                switch_demand(B,A,E),
                                actual_traffic(B,A,A,E,F),
                                traffic_through_reroute(B,A,A,C,G),
                                switch_completions(A,H)),
                               I).
```

```
% TRAFFIC THROUGH SINGLE-TRUNK NON-DEFAULT RE-ROUTES
f61(K) :- count([E,F,C,G,H,B,A,I,v(I)],(switch(A),
                                        switch(B),
                                        B\==A,
                                        connects(C,B,D),
                                        cycle_free_route(B,A,A),
                                        switch_demand(B,A,E),
                                        actual_traffic(B,A,A,E,F),
                                        connects(C,B,A),
                                        \+routed_trunk(B,A,C),
                                        cycle_free_route(A,A,A),
                                        switch_demand(B,A,G),
                                        trunk_completion_ratio(C,H),
                                        I is integer(G*H),
                                        switch_completions(A,J)),
                        K).


% TRAFFIC BETWEEN PAIRS WITH NO DEMAND
f49(G) :- count([B,A,v(E)],(switch(A),
                            switch(B),
                            B\==A,
                            connects(C,B,D),
                            actual_traffic(B,A,A,E),
                            switch_demand(B,A,0),
                            switch_completions(A,F)),
                G).


% COMPLETIONS VIA SINGLE-STEP NON-DEFAULT RE-ROUTES
f61(K) :- count([E,F,C,G,H,B,A,I,v(I)],
                (switch(A),
                 switch(B),
                 B\==A,
                 connects(C,B,D),c
                 ycle_free_route(B,A,A),
                 switch_demand(B,A,E),
                 actual_traffic(B,A,A,E,F),
                 connects(C,B,A),
                 \+routed_trunk(B,A,C),
                 cycle_free_route(A,A,A),
                 switch_demand(B,A,G),
                 trunk_completion_ratio(C,H),
                 I is integer(G*H),
                 switch_completions(A,J)),
                K).
```

BIBLIOGRAPHY

Ash, T. (1989). *Dynamic node creation in backpropagation networks*, (ICS Report 8901), San Diego, CA: University of California, Institute for Cognitive Science.

Baase, S. (1988). *Computer algorithms.* Reading, MA: Addison-Wesley.

Barto, A. G., Sutton, R. S., & Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics, 13*, 835-846.

Berliner, H. (1979). On the construction of evaluation functions for large domains. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence.* Tokyo, Japan: Morgan Kaufmann.

Berliner, H. J. (1980). Backgammon computer program beats world champion. *Artificial Intelligence, 14*, 205-220.

Brandau, R., Lemmon, A., & Lafond, C. (1991). Experience with extended episodes: Cases with complex temporal structure. *Proceedings of the DARPA Workshop on Case-Based Reasoning* (pp. 1-12). Washington, D.C.: Morgan Kaufmann.

Callan, J. P., & Utgoff, P. E. (1991a). Constructive induction on domain knowledge. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 614-619). Anaheim, CA: MIT Press.

Callan, J. P., & Utgoff, P. E. (1991b). A transformational approach to constructive induction. *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 122-126). Evanston, IL: Morgan Kaufmann.

Callan, J. P. (1993). *Knowledge-based feature generation for inductive learning.* Doctoral dissertation, Department of Computer Science, University of Massachusetts, Amherst, MA.

Chase, W., & Simon, H. (1973). Perception in chess. *Cognitive Psychology, 4*, 55-81.

Chase, M., Zweben, M., Piazza, R., Burger, J., Maglio, P., & Hirsh, H. (1989). Approximating learned search control knowledge. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 218-220). Ithaca, NY: Morgan Kaufmann.

Cousot, P., & Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages.*

De Jong, K. A., & Schultz, A. C. (1988). Using experience-based learning in game playing. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 284-290). Ann Arbor, MI: Morgan Kaufman.

Dietterich, T., & Michalski, R. (1981). Inductive learning of structural descriptions. *Artificial Intelligence, 16*, 257-294.

Dietterich, T. G., London, B., Clarkson, K., & Dromey, G. (1982). Learning and inductive inference. In Cohen & Feigenbaum (Eds.), *The Handbook of Artificial Intelligence: Volume III.* San Mateo, CA: Morgan Kaufmann.

Dietterich, T., & Michalski, R. (1983). A comparative review of selected methods for learning from examples. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Dietterich, T., & Buchanan, B. (1984). The role of the critic in learning systems. In Selfridge, Rissland & Arbib (Eds.), *Adaptive Control of Ill-Defined Systems.* New York: Plenum Press.

Doran, J. (1969). Planning and robots. In Meltzer & Michie (Eds.), *Machine Intelligence 5.* Edinburgh University Press.

Drastal, G., Czako, G., & Raatz, S. (1989). Induction in an abstraction space: A form of constructive induction. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 708-712). Detroit, Michigan: Morgan Kaufmann.

Duda, R. O., & Hart, P. E. (1973). *Pattern classification and scene analysis.* New York: Wiley & Sons.

Ellman, T. (1990). Mechanical generation of heuristics through approximation of intractable theories. *Proceedings of the AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions.* New Brunswick, NJ.

Fahlman, S. E., & Lebiere, C. (1990). The cascade correlation architecture. *Advances in Neural Information Processing Systems, 2*, 524-532.

Fawcett, T. E., & Utgoff, P. E. (1991). A hybrid method for feature generation. *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 137-141). Evanston, IL: Morgan Kaufmann.

Fawcett, T. E., & Utgoff, P. E. (1992). Automatic feature generation for problem solving systems. *Machine Learning: Proceedings of the Ninth International Conference* (pp. 144-153). San Mateo, CA: Morgan Kaufmann.

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

Fisher, D. H., & McKusick, K.B. (1989). An empirical comparison of ID3 and back-propagation. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 788-793). Detroit, Michigan: Morgan Kaufmann.

Flann, N. S., & Dietterich, T. G. (1986). Selecting appropriate representations for learning from examples. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 460-466). Philadelphia, PA: Morgan Kaufmann.

Flann, N. (1990). Applying abstraction and simplification to learn in intractable domains. *Proceedings of the Seventh International Conference on Machine Learning* (pp. 277-285). Austin, TX: Morgan Kaufmann.

Frey, P. W. (1986). Algorithmic strategies for improving the performance of game-playing programs. *Evolution, Games and Learning* (pp. 355-365). New York, NY: North Holland.

Gardner, A. (1981). Search. In Barr & Feigenbaum (Eds.), *The Handbook of Artificial Intelligence: Volume I.* Los Altos, CA: William Kaufmann.

Gaschnig, J. (1979). A problem similarity approach to devising heuristics: First results. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp. 301-307). Tokyo, Japan.

Gelernter, H. (1959). Realization of a geometry theorem-proving machine. In Feigenbaum & Feldman (Eds.), *Computers and thought.* New York: McGraw-Hill.

Gennari, J. H. (1989). Focused concept formation. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 379-382). Ithaca, NY: Morgan Kaufmann.

Guida, G., & Somalvico, M. (1979). A method of computing heuristics in problem solving. *Information Sciences, 19*, 251-259.

Hammond, Kristian (1986). Learning to anticipate and avoid planning problems through the explanation of failures. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 556-560). Philadelpha, PA: Morgan Kaufmann.

Hart, P., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on SSC, 4*, 100-107.

Holland, J. H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Michalski, Carbonell &

Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Huberman, B. J. (1968). *A program to play chess end games.* Doctoral dissertation, Department of Computer Sciences, Stanford University.

Iba, G. A. (1988). *The application of heuristic search and discovery of macro-operators to network traffic control.*, (Technical Memorandom TM-0062-10-88-506), GTE Laboratories, Inc.

Iba, G.A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning, 3*, 285-317.

Jacobs, R. A. (1990). *Task decomposition through competition in a modular connectionist architecture.* Doctoral dissertation, Department of Computer and Information Science, University of Massachusetts, Amherst, MA.

Keller, R. M. (1987). Concept learning in context. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 91-102). Irvine, CA: Morgan Kaufmann.

Kierulf, A. (1989). New concepts in computer Othello: Corner value, edge avoidance, access, and parity. In Levy (Ed.), *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad.* Halsted Press.

Kira, K., & Rendell, L. (1992). A practical approach to feature selection. *Machine Learning: Proceedings of the Ninth International Conference* (pp. 249-256). San Mateo, CA: Morgan Kaufmann.

Kittler, J. (1986). Feature selection and extraction. In Young & Fu (Eds.), *Handbook of pattern recognition and image processing.* New York: Academic Press.

Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Machine Learning, 6*, 293-326.

Korf, R. E. (1982). A program that learns to solve Rubik's cube. *Proceedings of the Second National Conference on Artificial Intelligence* (pp. 164-167). Pittsburgh, PA: William Kaufmann.

Korf, R. E. (1988). Search: A survey of recent results. In Howard Shrobe & American Association for Artificial Intelligence (Eds.), *Exploring Artificial Intelligence.* San Mateo: Morgan Kaufman.

Kowalski, R. (1969). Search strategies for theorem-proving. In Meltzer & Michie (Eds.), *Machine Intelligence 5.* Edinburgh University Press.

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

Langley, P. (1983). Learning effective search heuristics. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence.* Karlsruhe, West Germany: William Kaufmann.

Lebowitz, M. (1986). Concept learning in a rich input domain: Generalization-Based Memory. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Lee, K. F., & Mahajan, S. (1988). A pattern classification approach to evaluation function learning. *Artificial Intelligence, 36,* 1-25.

Lin, Long-Ji (1991). Programming robots using reinforcement learning and teaching. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 781-786). Anaheim, CA: MIT Press.

Matheus, C. J., & Rendell, L. A. (1989a). Constructive induction on decision trees. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 645-650). Detroit, Michigan: Morgan Kaufmann.

Matheus, C. J. (1989b). A constructive induction framework. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 474-475). Ithaca, NY: Morgan Kaufmann.

Matheus, C. J. (1990). *Feature construction: An analytic framework and an application to decision trees.* Doctoral dissertation, Department of Computer Science, University of Illinois, Urbana-Champaign, IL.

Michalski, R. S. (1983). A theory and methodology of inductive learning. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Michie, D., & Ross, R. (1969). Experiments with an adaptive graph traverser. In Meltzer & Michie (Eds.), *Machine Intelligence 5.* Edinburgh University Press.

Minsky, M. (1963). Steps towards artificial intelligence. In Feigenbaum & Feldman (Eds.), *Computers and thought.* New York: McGraw-Hill.

Minton, S. (1984). Constraint-based generalization: Learning game-playing plans from single examples. *Proceedings of the Fourth National Conference on Artificial Intelligence* (pp. 251-254). Austin, TX: Morgan Kaufmann.

Minton, S., & Carbonell, J. G. (1987). Strategies for learning search control rules: An explanation-based approach. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 228-235). Milan, Italy: Morgan Kaufmann.

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

Mitchell, D. (1984). *Using features to evaluate positions in experts' and novices' othello games*, (Masters thesis), Evanston, IL: Department of Psychology, Northwestern University.

Mitchell, T. M. (1977). Version spaces: A candidate elimination approach to rule learning. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (pp. 305-310). Morgan Kaufmann.

Mitchell, T. M. (1978). *Version spaces: An approach to concept learning.* Doctoral dissertation, Department of Electrical Engineering, Stanford University, Palo Alto, CA.

Mitchell, T. M., Utgoff, P. E., & Banerji, R. B. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

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

Mooney, R., Shavlik, J., Towell, G., & Gove, A. (1989). An experimental comparison of symbolic and connectionist learning algorithms. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 775-780). Detroit, Michigan: Morgan Kaufmann.

Mooney, R., & Ourston, D. (1991). Constructive induction in theory refinement. *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 178-182). Evanston, IL: Morgan Kaufmann.

Mostow, J., & Bhatnagar, N. (1987). Failsafe – a floor planner that uses EBG to learn from its failures. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 249-255). Milan, Italy: Morgan Kaufmann.

Mostow, J., & Fawcett, T. (1990). Generating useful approximations: A transformational model. *Proceedings of the AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions.* New Brunswick, NJ.

Muggleton, S. (1987). Duce, an oracle based approach to constructive induction. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 287-292). Milan, Italy: Morgan Kaufmann.

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, NJ: Prentice-Hall.

Newell, A., & Simon, H. A. (1963a). GPS, a program that simulates human thought. In Feigenbaum & Feldman (Eds.), *Computers and Thought.* New York: McGraw-Hill.

Newell, A., Shaw, J.C., & Simon, H. A. (1963b). Empirical explorations with the logic theory machine: A case history in heuristics. In Feigenbaum & Feldman (Eds.), *Computers and Thought.* New York: McGraw-Hill.

Newell, A. (1978). *HARPY: Production systems and human cognition,* (Technical Report CMU-CS-78-140), Carnegie-Mellong University, Computer Science.

Newell, A. (1980). Reasoning, problem solving and decision processes: The problem space as a fundamental category. In Nickerson (Ed.), *Attention and Performance VIII.* Hillsdale: Erbaum.

Nilsson, N. J. (1965). *Learning machines.* New York: McGraw-Hill.

Nilsson, N. (1971). *Problem solving methods in artificial intelligence.* New York: McGraw-Hill.

Nunez, M. (1988). Economic induction: A case study. *Proceedings of the Third European Working Session on Learning* (pp. 139-145). Glasgow, Scotland: Pitman.

Pagallo, G. (1989). Learning DNF by decision trees. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 639-644). Detroit, Michigan: Morgan Kaufmann.

Pagallo, G., & Haussler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning,* 71-99.

Pearl, J. (1984). *Heuristics: Intelligent search strategies for computer problem solving.* Reading, Ma: Addison-Wesley.

Prieditis, A. (1990). *Discovering effective admissible heuristics by abstraction and speedup: A transformational approach.* Doctoral dissertation, Rutgers University.

Puget, J. F. (1988). Learning invariants from explanations. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 200-204). Ann Arbor, MI: Morgan Kaufman.

Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess end games. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1,* 81-106.

Quinlan, J. R. (1987). Decision trees as probabilistic classifiers. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 31-37). Irvine, CA: Morgan Kaufmann.

Quinlan, J. R. (1992). *C4.5: Programs for machine learning.* Morgan Kaufmann.

Ragavan, H., & Rendell, L. (1991). Relations, knowledge and empirical learning. *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 188-192). Evanston, IL: Morgan Kaufmann.

Rendell, L. (1985). Substantial constructive induction using layered information compression: Tractable feature formation in search. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 650-658). Los Angeles, CA: Morgan Kaufmann.

Rendell, L., & Seshu, R. (1990). Learning hard concepts through constructive induction: Framework and rationale. *Computational Intelligence, 6*, 247-270.

Rich, E., & Knight, K. (1991). *Artificial intelligence.* McGraw-Hill.

Rivest, R. (1987). Learning decision lists. *Machine Learning, 2.*

Rosenbloom, P. (1982). A world-championship-level othello program. *Artificial Intelligence, 19*, 279-320.

Rumelhart, D. E., & McClelland, J. L. (1986). *Parallel distributed processing.* Cambridge, MA: MIT Press.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence, 5*, 115-135.

Samuel, A. (1959). Some studies in machine learning using the game of Checkers. *IBM Journal of Research and Development, 3*, 211-229.

Samuel, A. (1967). Some studies in machine learning using the game of Checkers II: Recent progress. *IBM Journal of Research and Development, 11*, 601-617.

Selfridge, O. G. (1959). Pandemonium: A paradigm for learning. *Proceedings of the Symposium on the Mechanization of Thought Processes* (pp. 513-526). Teddington, England: National Physical Laboratory, H.M. Stationary Office, London.

Schaeffer, J., Culbertson, J., Treloar, N., Knight, B., Lu, P., & Szafron, D. (1990). *Reviving the game of checkers*, University of Alberta, Department of Computing Science.

Schaeffer, J., Culbertson, J., Treloar, N., Knight, B., Lu, P., & Szafron, D. (1992). A world championship caliber checkers program. *Artificial Intelligence, 53*, 273-289.

Schlimmer, J. C., & Granger, R. H., Jr. (1986). Incremental learning from noisy data. *Machine Learning, 1*, 317-354.

Schlimmer, J. C. (1987). Incremental adjustment of representations. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 79-90). Irvine, CA: Morgan Kaufmann.

Schofield, P. (1967). Complete solution of the eight puzzle. In Michie & Collins (Eds.), *Machine Intelligence I.* Edinburgh and London: Oliver & Boyd.

Shannon, C. (1950). Programming a computer for playing chess. *Philosophical Magazine, 40*, 356-375.

Shen, Wei-Min, & Simon, Herbert (1989). Rule creation and rule learning through environmental exploration. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 675-680). Detroit, Michigan: Morgan Kaufmann.

Silver, B. (1986). Precondition analysis: Learning control information. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Silver, B., Frawley, W., Iba, G., Vittal, J., & Bradford, K. (1990a). ILS: A framework for multi-paradigmatic learning. *Proceedings of the Seventh International Conference on Machine Learning* (pp. 348-356). Austin, TX: Morgan Kaufmann.

Silver, B., Vittal, J., Frawley, W., Iba, G., & Bradford, K. (1990b). ILS: A framework for integrating multiple heterogeneous learning agents. *Proceedings of Second Generation Expert Systems, 10th International Workshop on Expert Systems and Their Applications* (pp. 301-313).

Slagle, J.R., & Dixon, J.K. (1969). Experiments with some programs that search game trees. *J. ACM, 16*, 189-207.

Smith, D.E., & Genesereth (1985). Ordering conjunctive queries. *Artificial Intelligence, 26*, 171-215.

Sterling, L., & Shapiro, E. (1986). *The Art of Prolog.* MIT Press.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning, 3*, 9-44.

Sutton, R. S. (1990). Integrated architectures for learning, planning and reacting based on approximating dynamic programming. *Proceedings of the Seventh International Conference on Machine Learning* (pp. 216-224). Austin, TX: Morgan Kaufmann.

Tadepalli, P. (1989). Lazy explanation-based learning: A solution to the intractable theory problem. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 694-700). Detroit, Michigan: Morgan Kaufmann.

Tan, M., & Schlimmer, J. C. (1990). Two case studies in cost-sensitive concept acquisition. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 854-860). Boston, MA: Morgan Kaufmann.

Tesauro, G. (1989a). Connectionist learning of expert preferences by comparison training. In Touretzky (Ed.), *Advances in Neural Information Processing Systems.* Morgan Kaufmann.

Tesauro, G., & Sejnowski, T. J. (1989b). A parallel network that learns to play backgammon. *Artificial Intelligence, 39*, 357-390.

Tesauro, G. (1992a). Temporal difference learning of backgammon strategy. *Machine Learning: Proceedings of the Ninth International Conference* (pp. 451-457). San Mateo, CA: Morgan Kaufmann.

Tesauro, G. (1992b). Practical issues in temporal difference learning. *Machine Learning, 8*, 257-277.

Towell, G., Shavlik, J., & Noordewier, M. (1990). Refinement of approximate domain theories by knowledge-based neural networks. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 861-866). Boston, MA: Morgan Kaufmann.

Utgoff, P. E. (1986a). Shift of bias for inductive concept learning. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Utgoff, P. E. (1986b). *Machine learning of inductive bias.* Hingham, MA: Kluwer.

Utgoff, P. E., & Saxena, S. (1987). Learning a preference predicate. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 115-121). Irvine, CA: Morgan Kaufmann.

Utgoff, P. E., & Heitman, P. S. (1988). Learning and generalizing move selection preferences. *Proceedings of the AAAI Symposium on Computer Game Playing* (pp. 36-40). Palo Alto, CA.

Utgoff, P. E., & Brodley, C. E. (1991a). *Linear machine decision trees*, (COINS Technical Report 91-10), Amherst, MA: University of Massachusetts, Department of Computer and Information Science.

Utgoff, P. E., & Clouse, J. A. (1991b). Two kinds of training information for evaluation function learning. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 596-600). Anaheim, CA: MIT Press.

Valtorta, M., & Zahid, M. (1990). Some heuristics cannot be derived from simplified models. *Proceedings of the AAAI-90 Workshop on Automatic Generation of Approximations and Abstractions* (pp. 87-97). New Brunswick, NJ.

van Harmelen, F., & Bundy, A. (1988). Explanation-based generalization = partial evaluation. *Artificial Intelligence, 34*, 401-412.

Waldinger, R. (1976). Achieving several goals simultaneously. In Elcock & Michie (Eds.), *Machine Intelligence.* New York: Wiley & Sons.

Warren, D.H.D. (1977). *Implementing prolog - compiling predicate logic programs,* (39 & 40), University of Edinburgh, Artificial Intelligence.

Warren, D. (1981). Efficient processing of interactive relational database queries expressed in logic. *Seventh International Conference on Very Large Data Bases* (pp. 272-281). Cannes, France.

Waterman, D. A. (1970). Generalization learning techniques for automating the learning of heuristics. *Artificial Intelligence, 1,* 121-170.

Weigend, A., Rumelhart, D., & Huberman, B. (1991). Generalization by weight-elimination with application to forecasting. *Advances in Neural Information Processing.* San Mateo, CA: Morgan Kaufmann.

Widrow, B., Gupta, N., & Maitra, S. (1973). Punish/reward: Learning with a critic in adaptive threshold systems. *IEEE Transactions on Systems, Man and Cybernetics, SMC-3,* 455-465.

Yee, R. C., Saxena, S., Utgoff, P. E., & Barto, A. G. (1990). Explaining temporal-differences to create useful concepts for evaluating states. *Proceedings of the Eighth National Conference on Artificial Intelligence.* Boston, MA: Morgan Kaufmann.

Young, P. (1984). *Recursive estimation and time-series analysis.* New York: Springer-Verlag.