Knowledge-Based Feature Generation for Inductive Learning

James P. Callan
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003

CMPSCI Technical Report 93-07
January 15, 1993

KNOWLEDGE-BASED FEATURE GENERATION FOR INDUCTIVE LEARNING

A Dissertation Presented

by

JAMES P. CALLAN

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

DOCTOR OF PHILOSOPHY
February 1993
Department of Computer Science

Knowledge-Based Feature Generation For Inductive Learning

A Dissertation Presented

by

James P. Callan

Approved as to style and content by:

To my teachers,
for guiding me
and inspiring me.

Thank you.

ACKNOWLEGMENTS

I have read many acknowledgements over the years, in conference papers, in books, and in dissertations. I was sometimes surprised at the depth of feeling expressed, and the number of people acknowledged. Now I understand. These pages cannot convey the depth of my own feelings for the many people who have helped me. All I can say is that I will always remember and be grateful.

First, of course, is Paul Utgoff. Paul invested great time and energy teaching me how to conduct research and trying to improve my writing style. He also gave me complete freedom to pursue my ideas, which must have been frustrating at times. In addition to what he did for me personally, Paul has always demonstrated integrity, scholarship, and a habit of leading by example. I consider myself fortunate to have worked with him.

I have also been fortunate in having an excellent Ph.D. committee. Edwina Rissland, Andy Barto, Chris Matheus and Patrick Kelly have each taken the time to give me different insights into work that I thought I knew well. I appreciate their efforts and thank them all.

I thank Bruce Croft for providing me with computers at a time when I needed them badly. Most of the experiments reported in this dissertation were conducted on his machines. I also thank Ross Quinlan for giving me a copy of his C4.5 program.

The graduate students in the Machine Learning group have provided me with friendship, advice and critical discussion for several years. There is no adequate expression of my debt to them. I am particularly grateful to Tom Fawcett, because his work and mine have often overlapped. Our discussions gave me a perspective on my work that I could not have developed alone. I also thank him for sharing his Othello domain theory with me. I am grateful to Sharad Saxena for bringing to our group rigorous thinking and a broad knowledge of science and mathematics. I have missed him often since he graduated. I thank Carla Brodley for helping me to keep graduate school in perspective, for spending a lot of time helping me with LMDT, and for her detailed comments on my dissertation. Jeff Clouse generously shared his WYSTAN program for playing Othello, and helped me in understanding both Othello and temporal difference learning.

Margie Connell, David Haines, David Lewis, and Rick Yee have each provided me with interesting comments and advice at various stages of my research. Their help is much appreciated. I also thank Adele Howe, Dan Suthers, Philip Johnson, and Kishore Swaminathan for their advice and support.

I thank John Buonaccorsi and Chengda Yang, of the University of Massachusetts Statistical Consulting Center, for assistance in the design of experiments.

This research was supported by the Digital Equipment Corporation, under contracts VW-758557, VW-776934, and TV-798043, and by the Office of Naval Research through a University Research Initiative Program, under contract N00014-86-K-0764.

Last, and most important, I thank Adéle, Ethan and Chelsea for their support, encouragement, and sacrifices over several years. Although I have (usually) enjoyed graduate school, the best part of these years has been going home.

ABSTRACT

KNOWLEDGE-BASED FEATURE GENERATION
FOR INDUCTIVE LEARNING
FEBRUARY 1993
JAMES P. CALLAN, B.A., UNIVERSITY OF CONNECTICUT
M.S., UNIVERSITY OF MASSACHUSETTS
PH.D., UNIVERSITY OF MASSACHUSETTS
Directed by: Professor Paul E. Utgoff

Inductive learning is an approach to machine learning in which concepts are learned from examples and counterexamples. One requirement for inductive learning is an explicit representation of the characteristics, or *features*, that determine whether an object is an example or counterexample. Obvious or easily available representations do not reliably satisfy this requirement, so *constructive induction* algorithms have been developed to satisfy it automatically. However, there are some features, known to be useful, that have been beyond the capabilities of most constructive induction algorithms.

This dissertation develops *knowledge-based feature generation*, a stronger, but more restricted, method of constructive induction than was available previously. Knowledge-based feature generation is a heuristic method of using one general and easily available form of domain knowledge to create *functional* features for one class of learning problems. The method consists of heuristics for creating features, for pruning useless new features, and for estimating feature cost. It has been tested empirically on problems ranging from simple to complex, and with inductive learning algorithms of varying power. The results show knowledge-based feature generation to be a general method of creating useful new features for one class of learning problems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

Inductive learning is an approach to machine learning in which concepts are learned from examples. The success or failure of an inductive algorithm at learning a particular concept depends on how the examples are represented. A 'good' representation enables the algorithm to learn a concept that describes both the examples seen and examples not yet seen. A 'bad' representation may make it impossible to discover an accurate concept.

Inductive learning algorithms are sensitive to the representation of examples because inductive algorithms define a concept by the characteristics that examples of the concept share and counterexamples do not. For example, if in a set of objects those with handles are identified as *cups*, an inductive algorithm might conclude that having a handle is necessary and sufficient to be a cup. Clearly there is more to being a cup than having a handle. However, inductive learning algorithms can only reason about those characteristics of objects that the representation makes explicit. Sometimes the obvious or easily available representations are insufficient for learning a particular concept. The *representation problem* is the problem of determining which characteristics must be made explicit in order to learn a particular concept [Utgoff & Mitchell, 1982].

Research papers about inductive learning are littered with comments that illustrate the importance and difficulty of the representation problem. For example, Quinlan (1983) reported spending two person-months to design a representation that would enable ID3 to learn a class of chess endgames. Dietterich and Flann (1986) described Wyl, a Checkers program that needed one representation of Checkers boards for learning and another for playing the game. Lenat and Brown (1984) reported that the success of the AM program at discovering mathematical concepts was due in part to a fortuitous choice of representation.

Constructive induction algorithms are heuristic algorithms that try to improve a representation by adding new features [Michalski, 1983]. Most constructive induction algorithms create new features, and determine the effectiveness of existing features, from the same stream of examples seen by the inductive learning algorithm. The lack of additional information about the examples or the domain from which they are drawn limits the types of new features that can be discovered. In principle, a constructive induction algorithm can create any new feature that is a function of the original features. In practice, complexity considerations restrict attention to new features that are simple logical or arithmetic functions of existing features (e.g., [Schlimmer & Granger, 1986; Pagallo, 1989; Matheus, 1990b; Murphy & Pazzani, 1991; Watanabe & Rendell, 1991]).

Some useful features cannot be created without having access to more than just a set of examples. For example, a *fork* in chess is a position in which one piece threatens two or more of the opponent's pieces. If the chess board is represented as a vector of features that describe the contents of each square, there is no simple logical or arithmetic function of features that reliably indicates the presence of a fork. One can only create the fork feature if one knows what it means for one piece to threaten another.

The difference between *fork* and the features created by most constructive induction algorithms is the difference between functional and structural features. A *structural* feature is one whose value can be determined with small computational effort [Flann & Dietterich, 1986]. A *functional*[1] feature is one that measures something related to problem-solving goals in the domain [Flann & Dietterich, 1986]. The functional and structural characteristics are usually related inversely. It is possible for a relatively simple combination of structural features to define an important domain-dependent relationship (a functional feature). However, functional features more often use quantified variables to describe complex relationships among unidentified structural features. As a result, functional features are usually beyond the abilities of most constructive induction algorithms.

How are functional features created? Previous research on constructive induction has shown that functional features can be created by first creating complex structural features and then using domain knowledge to transform constants into variables [Matheus & Rendell, 1989; Matheus, 1990b]. However, the method requires a very specific form of domain knowledge, and is limited in practice to simple functional features.

This dissertation explores whether it is possible to create complex functional features within the inductive learning paradigm. Such a method is desirable for several reasons. First, it should reduce the need for manual intervention in the creation of representations for inductive learning. Second, such a method might be expected to improve the accuracy of existing inductive learning algorithms, because it would make explicit a set of characteristics that would otherwise be implicit in examples. Finally, it would increase our understanding of the representation problem, one of the important unsolved problems in Machine Learning.

[1]The name *functional* for a class of features can be confusing, because of the word's other meanings. The name derives from the argument that a representation should describe characteristics relating to an object's purpose, or *function*, rather than its appearance. For example, a representation for cups should include upward-concavity, flat-bottom, liftable, and insulating, but not color, shape, or texture [Winston, Binford, Katz & Lowry, 1983].

Can complex functional features be created within the inductive learning paradigm? The thesis below provides a possible answer for one broad class of learning problems.

Primary Thesis: Functional features that are useful for learning search control knowledge can be created automatically using heuristics and a search problem specification.

The restriction to a single class of problems provides two benefits. First, a source of domain knowledge (a search problem specification) is available for all problems in the class. Second, the desired concepts all share a characteristic, described later, that suggests a new approach to feature creation. The restriction to heuristic methods insures that features can be created within the inductive learning paradigm.

The research described in this dissertation had three specific objectives. The first two were intended to support the thesis, if possible, while the third was intended to determine its limitations.

Objective 1: Develop a heuristic method that uses easily available domain knowledge to create functional features.

Objective 2: Determine whether the features reliably improve the performance of inductive learning algorithms on a variety of problems.

Objective 3: Determine the limitations of the method.

These objectives shaped the thesis research, and are reflected in the organization of the chapters in this dissertation. Chapter 10 returns to these objectives to determine whether they were met.

1.1 Search Control Knowledge

Search is a general and well-known approach to problem-solving in which the problem-solver repeatedly selects one state to explore next. In theory, all that is needed to solve a search problem is a specification of the problem and an exhaustive search procedure. In practice, exhaustive search is too slow, because the number of possible search states is exponential in the distance to a solution. A search space with branching factor b and depth d contains $O(b^d)$ states. A small increase in the size of the problem (either b, or especially d) causes a much larger increase in the number of search states. Search in large problem spaces is feasible only if the problem-solver has a good method of deciding which search states to explore first.

Many problem-solving methods explore next the state that appears most promising, as indicated by an *evaluation function.* An evaluation function can take many forms.

- It might return a numeric value, in which case the most valued state is the state with the highest (or lowest) numeric value.
- It might indicate a preference between a pair of states, in which case the most valued state is the state that is preferred most often.
- It might indicate that a state does or does not lead to a solution, in which case any state that leads to a solution is valued more highly than every state that does not lead to a solution.

It is often difficult to create an evaluation function that makes the desired distinctions between search states. One approach is to create and test evaluation functions manually until some stopping criteria is reached. Another approach is to use an inductive learning algorithm to create the evaluation function [Samuel, 1959; Samuel, 1967; Lee & Mahajan, 1988]. However, as described above, inductive learning can only succeed if search states are described by an appropriate set of features.

1.2 An Approach to Feature Creation

All state-space search problems are specified by describing an initial state, a goal state, and a set of operators for transforming one state into another [Newell & Simon, 1972]. This dissertation refers to such a description as a *search problem specification*, or as a *problem specification* for short. Search problem specifications are a form of domain knowledge that is correct and complete, but rarely tractable. That is, they specify problems completely, but in ways that make infeasible an exhaustive enumeration of paths from initial state to goal state.

A search problem specification is presumably available for every search problem of interest. This presumption is based on the observation that search control knowledge is intended for use by a problem-solving system. The problem-solving system must, in turn, have some knowledge of the state in which it started (the *initial state*), the actions of which it is capable (the *operators*), and a goal state for which it is searching. This knowledge defines a state-space search problem [Newell & Simon, 1972].

How can a problem specification be used to create complex functional features? This question is addressed by considering the difference between the description of the goal state, and the concept that the inductive algorithm should learn. The goal state implicitly defines a Boolean function on search states that is *TRUE* if and only if a state is a goal. In contrast, the desired concept is an evaluation function that enables the problem solver to hillclimb to a goal state. The desired concept implicitly defines a gradient whose value increases (or decreases) monotonically as a goal state is approached.

The description of the goal state, as a whole, provides no information about whether one state is closer than another to a goal. However, the description is not always monolithic. It is often composed of subexpressions that each describe a subgoal or constraint. Collectively they specify the goal state, but individually they may indicate progress in reaching the goal.

This dissertation develops a set of heuristics for breaking apart goal specifications into their component pieces and for transforming those pieces into complex functional features. The methods of feature creation, collectively called *knowledge-based feature generation* [Callan, 1989; Callan & Utgoff, 1991a; Callan & Utgoff, 1991b], are heuristic because they are developed for search problems that are generally intractable. The heuristics are augmented with a set of *syntactic pruning rules* that recognize and discard useless features after they are created, and a set of optimizing techniques that make the method practical.

1.3 Overview of the Results

Experiments were conducted with four search control problems of varying complexity and four learning algorithms of varying power, in an effort to determine the generality of the method and its applicability to a wide class of inductive learning algorithms. The problems addressed were to learn search control information for a blocks world task, tic-tac-toe, the game Othello, and printed wiring board component placement. The learning algorithms were the Perceptron, RLS, C4.5 and LMDT algorithms. Two types of experiments were conducted. Eighteen experiments examined the accuracy of the learning algorithms when trained on a set of examples and then tested on another disjoint set of examples drawn from the same population. Another six experiments examined the effect of a learned concept on problem-solving performance.

The first set of experiments demonstrated the utility of the functional features created automatically by knowledge-based feature generation. The improvement in the ability of the concept to classify unseen examples varied from small to large, depending upon the problem, but in all cases the differences were consistently repeatable, and in all but one they were statistically significant.

The second set of experiments confirmed the utility of knowledge-based feature generation. Four of the the five experiments showed that an average improvement in classification accuracy led to an average improvement in problem-solving ability. However, while the first set of experiments was quite stable with training data chosen randomly, the second set of experiments was not. Some learned concepts were good at problem-solving while others, trained under the same conditions with slightly different data, were not. Analysis of the results suggests that the erratic behavior may have been due to training and testing data that were noisy and not representative of the states actually encountered during problem-solving.

The conclusion drawn from the experimental results is that knowledge-based feature generation is an effective method of creating complex functional features.

There are several reasons for this conclusion. First, the results from the first set of experiments are very clear. In every case tested, an improvement was found. Second, improvement was found across a range of inductive learning algorithms. No other method of constructive induction has been tested on so many different learning algorithms. Finally, the improvements in the accuracy of the evaluation functions were shown to lead to improved problem solving in at least some cases.

1.4 Significance of the Research

The research described in this dissertation is significant for three reasons. First, it demonstrates a *new approach* to constructive induction. This new approach is characterized by heuristic use of knowledge that is general and easily available for a specified class of learning problems. Second, the research contributes a *specific method* of constructive induction (knowledge-based feature generation). The principle advantage of this method is its ability to create complex functional features for intractable domains, and to do so without search. Third, the *CINDI program*, which implements the method, addresses a variety of issues normally overlooked in constructive induction research. In particular, CINDI demonstrates simple, effective, rules for recognizing useless features, methods for estimating feature complexity, and the importance of feature optimization in complex domains.

1.5 Guide to the Thesis

The next chapter begins by analyzing the information contained in the specification of a search problem. That analysis motivates the development of *knowledge-based feature generation*, a method of transforming the specification into a set of terms that represent search states. The chapter also discusses the efficiency of features produced automatically, and describes how to estimate the computational cost of each term.

Chapter 3 describes the problem of identifying in advance those terms that will be most useful to the inductive algorithm. Knowledge-based feature generation includes a partial solution to this problem, in the form of heuristic pruning rules that eliminate unnecessary features. Knowledge-based feature generation also provides an estimate of the cost of each feature, which could be a useful component of a more comprehensive solution to feature selection. Chapter 3 describes an example feature selection algorithm in which feature costs are taken into account.

Chapters 5, 6, 8, and 7 describe experiments that empirically confirm the utility of knowledge-based feature generation. Results from four domains are described: the blocks-world, tic-tac-toe, the game of Othello, and printed wiring board (PWB) component placement.

Both the method and the empirical results are analyzed in Chapter 9, in an attempt to understand when the method is applicable and why the resulting terms are useful for inductive learning. That chapter also describes related research on overcoming the obstacles that make both inductive learning and explanation-based learning difficult to apply to the problem of learning search control knowledge. In particular, the chapter describes related work on constructive induction and the intractable domain theory problem.

Chapter 10 summarizes the earlier chapters, and discusses some of the remaining open problems that this dissertation does not address.

Finally, the appendices contain the problem specifications and other information necessary to reproduce the results obtained in Chapters 5 through 8.

CHAPTER 2

KNOWLEDGE-BASED FEATURE GENERATION

A state-space search problem is defined by an initial state, a set of goal states, and a set of problem-solving operators [Newell & Simon, 1972]. In principal, it does not matter whether the problem is defined in a particular representation or language. In practice, search problems are usually defined in whatever representation is most convenient for the problem-solver.

The task of creating features from *any* representation of a search problem is broader than necessary to explore the thesis. This dissertation supports the thesis with a method that creates features for search problems represented in a single representation: A many-sorted first order predicate calculus, often referred to in this dissertation simply as first order predicate calculus. A first order predicate calculus representation was chosen because it is general. Many of the representations used commonly for search problems can be expressed in a many-sorted first order predicate calculus. Propositional calculus and Horn clause logic are two well-known examples.

The next section briefly reviews the many-sorted first order predicate calculus. It is intended for those who are familiar with formal languages, but rusty on the details of first order predicate calculus. It is followed by Section 2.2, which analyzes the information in the definition of a search problem, and then develops a method of transforming that information into features that describe search states. Section 2.4 discusses the problem of optimizing features that are created automatically from the specification of a search problem. Section 2.5 describes how to determine, in advance of their use, the costs of features. Finally, Section 2.6 describes a computer program that implements the feature creation, optimization, and cost estimation techniques presented in this chapter.

2.1 Many-Sorted First Order Predicate Calculus

First order predicate calculus is a formal language in which one can make precise statements about both individuals and sets of individuals. A statement can be composed of constants, variables, predicates, functions, connectives and quantifiers.

A constant refers to, or names, a specific individual. A predicate asserts that an ordered tuple of individuals satisfies some property. A function maps an ordered tuple of individuals to a single individual. A connective enables statements to be combined. A quantifier enables a statement to refer to a set of individuals.

The following are some examples of English statements, and some of their possible representations in first order predicate calculus.

English	First Order Predicate Calculus
It is Wednesday	Wednesday()
Pat is a student	Student (Pat)
All students graduate	$\forall s, (\neg \text{Student(s)} \vee \text{Graduate(s)})$
Pat knows all students	$\forall s, (\neg \text{Student}(s) \vee \text{Knows(Pat,}s))$
Pat knows a student	$\exists s, (\text{Student}(s) \wedge \text{Knows(Pat,}s))$

A language is *first-order* if there is no way to refer by name to relations that exist among objects [Enderton, 1972]. For example, one cannot say "All relationships among students are transitive" in a first order language.

A first-order language is a *predicate calculus* if it does not contain proper axioms [Mendelson, 1979]. Transitivity and irreflexivity are two examples of proper axioms.

The example "All students graduate" and "Pat knows all students," presented above, illustrate a sometimes inconvenient characteristic of logical languages: all variables initially range over the universe of individuals. If one wishes to refer to a subset of the universe, one must create a variable that ranges over the universe and then explicitly restrict it to a subset. A *many-sorted* logical language is one in which variables can range over different universes [Enderton, 1972]. In a many-sorted logical language, one might express the English sentence "Pat knows a student" as

$$\exists s \epsilon Students, \text{Knows(Pat,}s)$$

The distinction between the many-sorted representation of "Pat knows a student" and the single-sorted representation is one of convenience. The power of the representation is unchanged [Enderton, 1972].

2.2 A Transformational Approach to Feature Construction

It is not difficult to use a state-space search problem specification to create a numeric evaluation function. The goal condition defines a single function, f_G, whose value is 1 (True) for goal states and 0 (False) for other states. However, f_G is both an inadequate evaluation function and an insufficient vocabulary for learning an evaluation function, because it varies rarely. Its value remains constant from the initial state until immediately before a goal is reached (Figure 2.1).

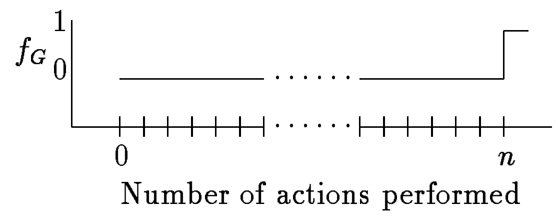

Figure 2.1 The behavior during problem-solving of a function that is 1 for goals and 0 for other states.

A larger and more varied vocabulary can be obtained by decomposing the single constraint represented by the goal condition into a set of constraints, or subgoals, each of which defines a new feature. Additional variation can sometimes be introduced by transforming Boolean constraints into numeric features having a range of discrete values. The result is a set of features, each of which makes explicit the problem-solver's progress in solving part of the original problem. If the decomposition can be applied recursively, it yields features that represent progressively smaller parts of the original problem. Ideally the problem-solver's progress at each state is reflected in a change to the value of at least one feature.

It is also useful to treat the preconditions of problem-solving operators as additional goal conditions, and apply decomposition to them as well. It makes sense because, in practice, it is common to move goal state requirements into the preconditions of problem-solving operators so that fewer search states are created [Mostow & Bhatnagar, 1987]. One could forbid this practice, requiring instead that all goal state requirements reside in the specification of the goal, but the resulting method would be less widely applicable.

Figure 2.2 illustrates how decomposition and transformation of a search problem specification can result in an improved vocabulary. Boolean features f_1 and f_2 are the result of decomposing the goal condition (f_G in Figure 2.1). The multi-valued numeric feature f_3 is the result of transforming the Boolean constraint represented by feature f_2. The evaluation function h that is constructed from these features is superior to f_G, because it is a better approximation of a function that varies monotonically as a goal is approached.

Statements in first order predicate calculus consist of constants, predicates, connectives, functions, quantifiers and variables. Three of these result in Boolean values, and are therefore candidates for being transformed into numeric functions: connectives, quantifiers and predicates. The following subsections describe how such a transformation can be accomplished.

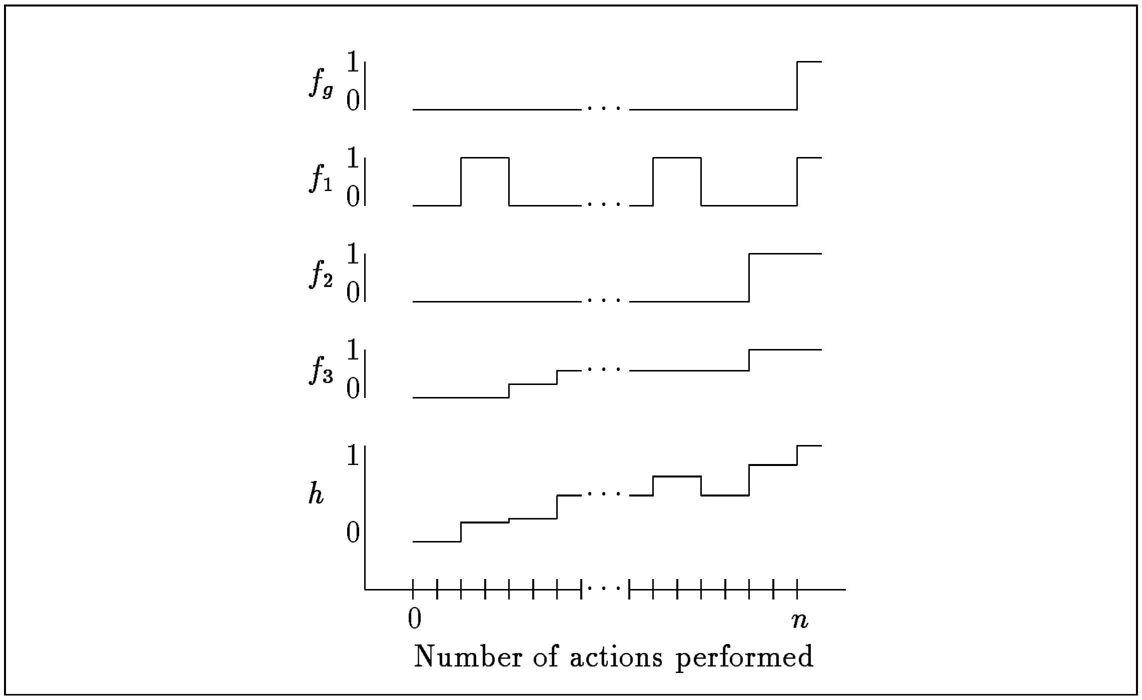

Figure 2.2 The behaviors during problem-solving of three features (f_1, f_2, and f_3) and an evaluation function (h) constructed from them.

2.2.1 A Transformation for Connectives

Connectives compose sentences to form more complicated sentences. The sentences of interest in a search problem specification describe the goal condition and the operator preconditions. Those sentences, and the simpler sentences from which they are composed, are constraints that are relevant to controlling search. The *LE* transformation [Callan & Utgoff, 1991a; Callan & Utgoff, 1991b], developed as part of the thesis research, makes those constraints explicit. It does so by decomposing sentences with connectives into sets of simpler sentences. The resulting sentences, called *LE terms*, map search states to Boolean values.

The *LE* transformation decomposes a sentence by removing negation ($\neg$) and the connectives AND ($\wedge$) and OR ($\vee$). Its definition is:

$$Q^m_{i=1}(\vee^n_{j=1} B_j) \underset{LE}{\Rightarrow} \{(Q^m_{i=1} B_1), \ldots, (Q^m_{i=1} B_n)\}$$

$$Q^m_{i=1}(\wedge^n_{j=1} B_j) \underset{LE}{\Rightarrow} \{(Q^m_{i=1} B_1), \ldots, (Q^m_{i=1} B_n)\}$$

$$Q^m_{i=1}(\neg B) \underset{LE}{\Rightarrow} Q^m_{i=1} B$$

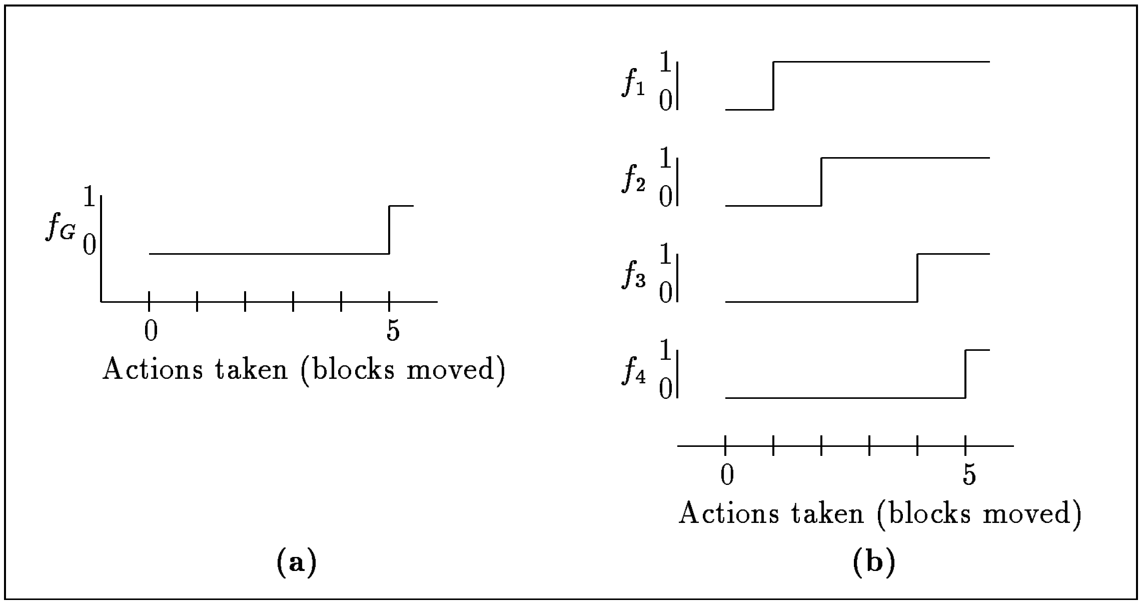

Figure 2.3 The effect of the *LE* transformation. (a) The value of the goal statement for a blocks-world task during problem-solving. (b) Behavior during the same period of the four *LE* terms that are produced from the goal statement.

In the definition above, $Q_{i=1}^{m}$ represents a list of m quantified variables, where $m \geq 0$. Each Q_i matches either $\forall v_i \epsilon S_i$ or $\exists v_i \epsilon S_i$. B_j represents one of n quantified statements, and B represents a single quantified statement. After the new *LE* terms are created, they are optimized by removing unnessary quantifiers.

The effect of the *LE* transformation can be illustrated with an example from an experiment discussed in Chapter 5. In that experiment, the problem-solver's goal is to stack four blocks in a specified order (on (d,Table) $\wedge$ on (c,d) $\wedge$ on (b,c) $\wedge$ on (a,b)). The *LE* transformation decomposes the goal into four Boolean features (on (d,Table), on (c,d), on (b,c), and on (a,b)). Figure 2.3 illustrates the behavior of these features along an optimal solution path for the problem that begins with A on B, B on the table, D on C, and C on the table.

The *LE* transformation is normally applied first to the specification of a goal or the precondition of an operator, and then recursively to the resulting *LE* terms. If it is applied wherever possible, the result is a hierarchy containing between c and $2c$ *LE* terms, with the goal or operator precondition at the root; c is the number of connectives and negation symbols in the root.

When the *LE* transformation decomposes a statement, it does not preserve the dependencies that may exist among the subexpressions. As a result, the *LE* transformation decomposes the two different statements on the left below into the same set of new terms.

$$\exists a \epsilon A, (p_1(a) \land p_2(a)) \quad \underset{LE}{\Rightarrow} \quad \{(\exists a \epsilon A, p_1(a)), (\exists a \epsilon A, p_2(a))\}$$

$$(\exists a \epsilon A, p_1(a)) \land (\exists a \epsilon A, p_2(a)) \quad \underset{LE}{\Rightarrow} \quad \{(\exists a \epsilon A, p_1(a)), (\exists a \epsilon A, p_2(a))\}$$

This characteristic of the *LE* transformation is not undesirable. One approach to problem-solving is to initially ignore some dependencies among subproblems, in order to reduce complexity. For example, the HACKER program is based on a heuristic, called the *linear assumption*, that treated all subgoals as independent. Violations of this assumption were handled, when they arose, by other heuristics [Sussman, 1975]. The ability to ignore temporarily some constraints makes it possible to find subgoals for problems that have highly interrelated constraints. A set of *LE* terms enables the inductive algorithm to decide for itself which dependencies are most important, and in what order they should be satisfied. This behavior is possible because dependency information about any two *LE* terms remains available in their closest common ancestor in the *LE* term hierarchy.

Early work on knowledge-based feature generation referred to *LE* terms as *subgoals* or *subproblems* [Callan, 1989], because many of them represent constraints that the problem-solver must satisfy. However, it is a mistake to imply that every sentence produced by the *LE* transformation corresponds to a meaningful constraint, subgoal or subproblem. Some sentences produced by the *LE* transformation are always true or always false. The problem of detecting such sentences in advance of their use as terms for inductive learning is discussed in Section 3.1.

2.2.2 A Transformation for Quantifiers

The *UEQ* transformation [Callan & Utgoff, 1991a; Callan & Utgoff, 1991b], developed as part of the thesis research, provides more information about the satisfaction of a condition. It does so by transforming quantified sentences (that is, assertions about the number of individuals in a set that satisfy a given condition) into numeric functions. The transformation is applied to any quantified statement. In particular, it may be applied to the goal condition, operator preconditions, and *LE* terms. Numeric functions created by the *UEQ* transformation are called *UEQ* terms.

The *UEQ* transformation transforms a sentence by calculating the percentage of permutations of variable bindings that satisfy the specified condition. Its definition is:

$$Q_{i=1}^{m} B \quad \underset{UEQ}{\Rightarrow} \quad \frac{\Sigma_{v_1 \epsilon S_1} \ldots \Sigma_{v_m \epsilon S_m} \begin{cases} \text{if } B & 1 \\ \text{otherwise} & 0 \end{cases}}{\Pi_{i=1}^{m} |S_i|}$$

In the definition above, $Q_{i=1}^{m}$ represents m quantified variables, where $m > 0$. Each Q_i matches either $\forall v_i \epsilon S_i$ or $\exists v_i \epsilon S_i$. B represents the statement quantified by $Q_{i=1}^{m}$.

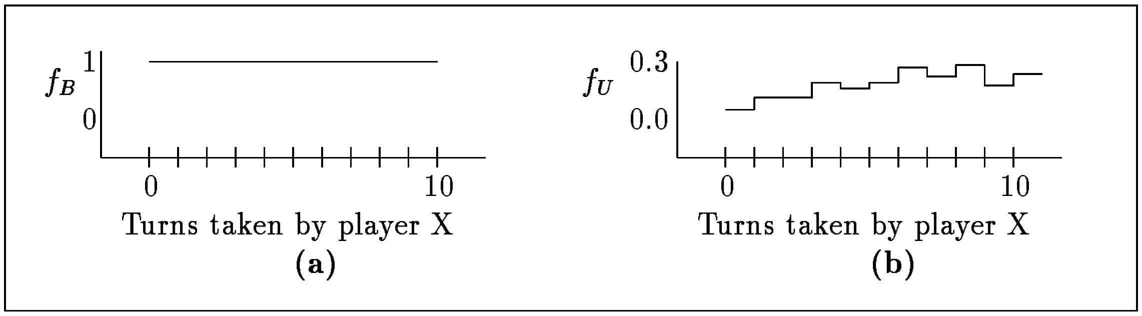

Figure 2.4 The effect of the *UEQ* transformation. (a) Behavior of an *LE* term for Othello during the first ten turns taken by player X. (b) Behavior during the same period of the *UEQ* term that is produced from the *LE* term.

The effect of the *UEQ* transformation can be seen in features created for the game Othello. An Othello player may only make a move if it is his turn and there is an empty square into which he is permitted to place a disc. The *LE* transformation decomposes this operator precondition into two Boolean features. One of them represents whether there is an empty square into which the problem-solver (player 'X') can place a disc ($\exists$ s1 ϵ Squares, legal_move (s1, X)). The *UEQ* transformation creates from this Boolean feature a numeric feature that calculates the percentage of squares into which the problem-solver can place a disc.

$$\frac{\Sigma_{\text{s1 } \epsilon \text{ Squares}} \begin{cases} \text{if legal\_move (s1, X)} & 1 \\ \text{otherwise} & 0 \end{cases}}{|\text{Squares}|}$$

Figure 2.4 illustrates the behavior of both the Boolean *LE* feature and the numeric *UEQ* feature when one human expert plays another human expert. (See Chapter 8 and Appendix D for more information about experiments conducted with Othello.)

The reason that a *UEQ* term is useful depends upon whether it was created from a universally ($\forall$) or existentially ($\exists$) quantified expression. If the function was created from a universal quantifier, it indicates 'how much' of the constraint is satisfied in a given state. If the function was created from an existential quantifier, it indicates 'how close' the problem-solver is to violating the constraint. The distinction between these points can be illustrated with an example.

If a constraint for a blocks-world problem requires that all blocks be on a table ($\forall b \epsilon B, \text{on}(b, \text{Table})$), then the corresponding *UEQ* term indicates the percentage of blocks on the table. As the problem-solver approaches a goal state, the value of this *UEQ* function would be expected to approach 1.

If a constraint for a blocks-world problem requires that at least one block be on a table ($\exists b \epsilon B, \text{on}(b, \text{Table})$), the corresponding *UEQ* term again indicates the percentage of blocks on the table. If the value of this function in a state is near 0, then the problem-solver must exercise care to keep it above 0. However, if the function is near 1, then the problem-solver has more freedom in selecting its future actions. The *value ordering* heuristic recommends exploring first those states that maintain as many options as possible for the future [Dechter & Pearl, 1987].

The computation carried out by a *UEQ* term does not depend on whether the term was created from a universally or existentially quantified expression. However, in one case the value must converge to 1 as a goal is approached, while in the other case any value above 0 is sufficient.

The *UEQ* transformation is a restricted form of Michalski's (1983) *counting arguments* rules. The counting arguments rules are applied to concept descriptions that contain quantified variables. Each rule counts the permutations of objects that satisfy a condition. However, the source of the condition is not specified by the counting arguments rules. In contrast, the *UEQ* transformation is applied to expressions generated by the *LE* transformation.

Counting the permutations that satisfy a condition can be complex computationally, depending upon the number of permutations and the cost of determining whether the condition is satisified. Counting is only practical if the sets are finite and not "too large". The decision about what is "too large" depends upon how much time one is willing to spend evaluating features. Use of large sets can be made more practical by optimizing feature computations, as discussed in Section 2.4. Cost estimates can be made when features are created (Section 2.5), in order to prevent use of features that are prohibitively expensive.

The cost of overly expensive features could also be reduced by sampling the space of permutations instead of examining it exhaustively. This approach is appealing, because it would yield features with different levels of precision (*resolution*). However, it is unclear what are the research issues, or how effective it would be. The most expensive features created during the thesis experiments invariably produced constant values. A sampling approach would have enabled recognition of this trait with lower computational effort, which is desirable, but it would not have affected learning performance. Feature selection identified and discarded the constant features prior to use of the learning algorithm.

2.2.3 Transformations for Predicates

A predicate asserts that a specific relationship holds between its arguments. When the relationship does not hold, a predicate provides no guidance to the problem-solver about how to behave. What the problem-solver needs is a function that suggests a direction in which to proceed, or a function that suggests how much of a change to make, in order to satisfy the predicate. Either type of function would provide more guidance to the problem-solver than a Boolean predicate.

It is unclear how to design a single transformation that creates the desired functions for all predicates. The transformations developed for connectives and quantifiers depend upon an understanding of the functions that connectives and quantifiers compute. No such understanding is possible for predicates, because the set of relations is infinite and not confined to any single domain. Therefore it is unlikely that a single transformation will suffice for predicates.

It may be impossible to develop a single transformation for all predicates, but it is not impossible to develop different transformations for different classes of predicates. For example, arithmetic equality ($=$) and inequality ($\neg, <, \leq, >, \geq$) predicates describe an ordering of individuals along an unspecified, but domain-dependent, dimension. An understanding of the role played by arithmetic equality and inequality predicates enables the creation of domain-independent transformations for those predicates, as described below. The same is true of other classes of predicates.

It is likely that there are some predicates for which no meaningful transformation can be developed. Some predicates, for example the predicate 'odd-integer', seem to be inherently Boolean. An integer either is or is not a odd. It is unclear how such a predicate could be transformed into a function that provides more information than the predicate itself.

The following section presents the *AE* transformations for arithmetic equality and inequality predicates. The *AE* transformations are used throughout the remainder of this dissertation. Section 10.3.3 outlines a transformation for predicates on sets, to suggest that transformations are possible for other classes of predicates.

2.2.3.1 The *AE-M* and *AE-E* Transformation

Arithmetic predicates can be divided into those that impose an ordering ($\{<, \leq, >, \geq\}$) and those that require or forbid a difference ($\{=, \neq\}$). Features created from sentences that impose an ordering should reveal the extent to which the ordering is satisfied in a given state. Features created from sentences that forbid or require a difference should reveal the extent to which a difference exists in a given state. The *AE* transformation [Callan & Utgoff, 1991a] satisfied these requirements by creating features that calculated the *average difference* between operands of an arithmetic predicate, over all permutations of variable bindings.

The *AE* transformation had two limitations. First, it conflated information about permutations that did and did not satisfy the quantified sentence. Second, it applied to sentences that consisted of only a single arithmetic predicate, thereby ignoring sentences that expressed relationships among groups of arithmetic predicates. Both of these limitations have been overcome by this thesis research.

There are two new *AE* transformations, developed as part of the thesis research, that can be applied to a Boolean sentence containing one or more arithmetic predicates. The *AE-M* transformation creates a *margin* term. The *AE-E* transformation creates an *error* term. The two types of terms are complementary. Both measure the average differences between operands of arithmetic predicates, and both propagate the results through arithmetic analogs of the sentence. The difference between margin

```
AE-M ($\Sigma_{v_1 \epsilon S_1}, \ldots, \Sigma_{v_m \epsilon S_m}, B$):
For each unique function type $t$ in expression $B$, do:

  1. Make a copy $M$ of expression $B$.

  2. Prune from $M$ all $=$ operators and all operands of AND and OR operators that
     do not contain an arithmetic function of type $t$.

  3. Make the following operator replacements in $M$.
     (a) Replace AND with MIN. Replace OR with MAX.
     (b) Replace $a \geq b$ and $a > b$ with $a - b$. Replace $a \leq b$ and $a < b$ with $b - a$.
     (c) Replace $a \neq b$ with ABS $(a - b)$.

  4. Create the feature:

     $$\frac{\Sigma_{v_1 \epsilon S_1} \ldots \Sigma_{v_m \epsilon S_m} \begin{cases} \text{if } B & M \\ \text{otherwise} & 0 \end{cases}}{\Pi_{i=1}^{m} |S_i|}$$
```

Figure 2.5 The algorithm for the AE *margin* (*AE-M*) transformation.

and error terms lies in when measurements are made, and how measurements are propagated.

Margin terms make measurements for permutations of variable bindings that satisfy the quantified expression. The arithmetic analog of the quantified expression is a copy in which MIN replaces AND, and MAX replaces OR. An algorithmic definition of the *AE-M* transformation is given in Figure 2.5.

Error terms make measurements for permutations of variable bindings that do not satisfy the quantified expression. The arithmetic analog of the quantified expression is a copy in which MIN replaces OR, and MAX replaces AND. An algorithmic definition of the *AE-E* transformation is given in Figure 2.6.

It does not make sense to apply MIN or MAX to numeric values from different units of measurement, so a single sentence may result in a set of *AM-M* and *AM-E* terms, one for each unit of measurement referenced in the sentence. The search problem specification is assumed not to specify the unit of measurement, or *type*, of each function. Instead, type equivalence of functions is determined automatically. Functions have equivalent types if their values are compared, added, or subtracted anywhere in the search problem specification.

Margin terms can be described informally as a measure of the robustness of those permutations that satisfy the constraint. Large margins are preferable to small margins. Error terms can be described informally as a measure of the work remaining to be done. Small errors are preferable to large errors.

```
$AE\text{-}E\ (\Sigma_{v_1 \epsilon S_1}, \ldots, \Sigma_{v_m \epsilon S_m}, B)$:
For each unique function type $t$ in expression $B$, do:
    1. Make a copy $E$ of expression $B$.
    2. Prune from $E$ all $\neq$ operators and all operands of AND and OR operators that
       do not contain an arithmetic function of type $t$.
    3. Make the following operator replacements in $E$.
       (a) Replace OR with MIN. Replace AND with MAX.
       (b) Replace $a \leq b$ and $a < b$ with $a - b$. Replace $a \geq b$ and $a > b$ with $b - a$.
       (c) Replace $a = b$ with ABS $(a - b)$.
    4. Create the feature:
       $$\frac{\Sigma_{v_1 \epsilon S_1} \ldots \Sigma_{v_m \epsilon S_m} \begin{cases} \text{if } B & 0 \\ \text{otherwise} & E \end{cases}}{\Pi_{i=1}^{m} |S_i|}$$
```

Figure 2.6 The algorithm for the AE *error* (*AE-E*) transformation.

The effect of the *AE-M* and *AE-E* transformations can be seen in features created for a circuit layout problem. One objective of the problem-solver is to find a placement of computer chips on the circuit board in which no chip overlaps (lies on top of) another chip. This constraint can be expressed for a given chip c as shown below.

$$\begin{aligned} &\forall c_j \epsilon \text{Comps}, \\ &\quad ((c == c_j) \vee \\ &\quad (\text{XMin}(c) > \text{XMax}(c_j)) \vee \\ &\quad (\text{XMin}(c_j) > \text{XMax}(c)) \vee \\ &\quad (\text{YMin}(c) > \text{YMax}(c_j)) \vee \\ &\quad (\text{YMin}(c_j) > \text{YMin}(c))) \end{aligned}$$

The constraint used in the Chapter 7 experiments is slightly more complex, because it allows the placement of chips on both sides of the circuit board. The additional complexity makes exposition difficult, hence the use of the simpler version above.

The *AE-M* transformation creates from this constraint two numeric *margin* features. One feature calculates the average distance in the horizontal dimension ('X' axis) of c to other components. The other feature performs the same calculation for the vertical dimension ('Y' axis). The feature for horizontal distance is shown below.

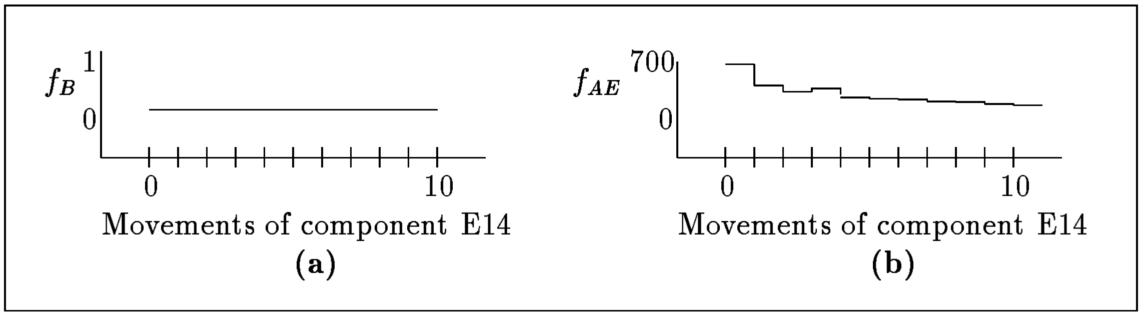

Figure 2.7 (a) Behavior of an *LE* term for PWB component placement during the first ten movements of chip E14. (b) Behavior during the same period of the *AE-E* term that is produced from the *LE* term.

$$\frac{\Sigma_{c_j \epsilon \text{Comps}} \begin{cases} \text{if } ((c == c_j) \vee & \text{Max}(\text{XMin}(c) - \text{XMax}(c_j), \\ \quad (\text{XMin}(c) > \text{XMax}(c_j)) \vee & \quad \text{XMin}(c_j) - \text{XMax}(c)) \\ \quad (\text{XMin}(c_j) > \text{XMax}(c)) \vee & \\ \quad (\text{YMin}(c) > \text{YMax}(c_j)) \vee & \\ \quad (\text{YMin}(c_j) > \text{YMax}(c))) & \\ & \\ \text{otherwise} & 0 \end{cases}}{|\text{Comps}|}$$

The *AE-E* transformation creates a corresponding pair of numeric *error* features that calculate the average amount of overlap in the horizontal and vertical directions. The feature for the horizontal overlap is shown below.

$$\frac{\Sigma_{c_j \epsilon \text{Comps}} \begin{cases} \text{if } ((c == c_j) \vee & 0 \\ \quad (\text{XMin}(c) > \text{XMax}(c_j)) \vee & \\ \quad (\text{XMin}(c_j) > \text{XMax}(c)) \vee & \\ \quad (\text{YMin}(c) > \text{YMax}(c_j)) \vee & \\ \quad (\text{YMin}(c_j) > \text{YMax}(c))) & \\ & \\ \text{otherwise} & \text{Min}(\text{XMax}(c_j) - \text{XMin}(c), \\ & \quad \text{XMax}(c) - \text{XMin}(c_j)) \end{cases}}{|\text{Comps}|}$$

Figure 2.7 illustrates the behavior of both the Boolean *LE* feature and one numeric *AE-M* feature when a problem-solver begins placing a set of chips that were initially overlapping one another. (See Chapter 7 and Appendix C for more information about experiments conducted with circuit board layout.)

The *AE* transformations could conceivably calculate other relationships between the operands, for example total difference, minimum difference or maximum difference. The *average* difference is more informative than the minimum or maximum because it summarizes the relationships among a population of objects, rather than one relationship between a single pair of objects (as would the minimum or maximum difference).

If one operand of an expression is a constant, the resulting margin and error terms measure the average distance to a threshold along some domain-dependent dimension. If neither operand is a constant, the terms measure the average distance between pairs of objects whose locations vary along the dimension. In both cases, the problem-solver's task is to drive the error terms to zero. States in which margin terms are near zero may leave the problem-solver with fewer options for satisfying constraints than states in which margin terms are higher.

The utility of arithmetic margin and error terms can be seen in an example from a circuit board layout problem. A constraint for this problem might require that an object be completely contained within the circuit board outlines. An error term would measure the amount by which an object violated the constraint. A margin term would measure the distance of the object to the edge of the board. An advantage of separate error and margin terms is that they allow a learning algorithm to attach different importance to the two conditions. A learning algorithm might place great emphasis on keeping the error term at zero, while placing little or no emphasis on the margin term.

2.3 Assumptions

The transformational approach to feature construction described above relies on several assumptions being satisfied. Some of the assumptions were stated explicitly, but others were implicit in the discussion. This section describes and discusses the important assumptions underlying knowledge-based feature generation.

Problem specification: A search problem specification is assumed to be available for knowledge-based feature generation. The problem specification must describe the goal state(s) and the problem-solving actions or operators. A description of the initial state is not necessary.

Domain dependence: The features created are domain-dependent. Knowledge-based feature generation does not create domain-independent features.

First Order Predicate Calculus: Knowledge-based feature generation is described in terms of First Order Predicate Calculus because that language is general and widely understood. However, the methods developed apply to other languages as well. Knowledge-based feature generation can be applied to any language that has connectives, quantifiers, and arithmetic equality and inequality.

Numeric features: Knowledge-based feature generation produces numeric features. The features are easist to use with inductive learning algorithms that directly handle numeric features. One could, if desired, convert the numeric features into Boolean or propositional features for use in a symbolic learning algorithm. STAGGER [Schlimmer, 1987], C4.5 [Quinlan, 1992], and ARGOT [Shaefer, 1988] are examples of learning algorithms that perform such a conversion automatically.

Finite sets: The *UEQ*, *AE-E* and *AE-M* transformations depend upon variables being quantified over finite sets. Statements that quantify variables over infinite sets produce features with infinite loops.

Small sets: The *UEQ*, *AE-E* and *AE-M* transformations are best applied when variables are quantified over sets whose sizes are not "too large." The definition of "too large" varies, depending upon the time one is willing to spend evaluating features. The experiments in Chapter 7 were conducted with features that quantify variables over a set containing 601 elements.

Hillclimbing: A hillclimbing approach to problem-solving [Newell & Simon, 1972] is considered desirable when it is possible. Plateaus, false peaks, and other problems associated with hillclimbining are assumed to be due to inadequacies of the representation of the state-space. The role of knowledge-based feature generation is to eliminate as many of these inadequacies as possible. The role of inductive learning is to learn an evaluation function that enables hillclimbing in the state-space.

It is also important to state clearly what are *not* assumptions of the thesis research.

Search problem: There are no assumptions about the search problem being solved. The *LE* transformation is more effective if the problem is disjunctive or serially-decomposable, but these characteristics are not requirements.

State-space: There are no assumptions about characteristics of the state-space. It need not support hillclimbing in the initial representation.

Evaluation function: There are no assumptions about the form of an evaluation function that would support hillclimbing. The target evaluation function need not be linear.

2.4 Optimization

One often overlooked aspect of feature generation is the cost of evaluating the resulting features. One can define a feature's *cost* to be the time necessary to assign it a value. Its *worth* is the time necessary to search for equivalent information. If a feature's cost outweighs its worth, then the feature *slows* the overall problem-solving effort [Berliner, 1984; Valtorta, 1983]. In that case, blind search would be faster than use of the feature.

It is straightforward to calculate a lower bound on the cost of each feature, as discussed below in Section 2.5. Unfortunately, there is no equally straightforward method of calculating a feature's worth, so one cannot dynamically determine whether a feature is worth its cost. However, it is possible to make each feature as efficient as possible, so that its chances of being worth its cost are improved. If a feature is already worth its cost, making it more efficient simply reduces the overall cost of problem-solving.

Analysis of the features produced by knowledge-based feature generation reveals one major source of inefficiency in both *LE* and *UEQ* features. This inefficiency can be eliminated with two optimizations, as discussed below.

2.4.1 Code Motion

Quantified sentences often occur in the definitions of *LE* and *UEQ* features. The usual method of evaluating a quantified sentence is to generate systematically a permutation of variable bindings and then test whether the permutation satisfies the specified condition. The generation and testing of permutations continues until either all permutations have been examined, or until a stopping criterion is reached. The stopping criteria differ for universal ($\forall$) and existential ($\exists$) quantifiers.

The time needed to evaluate a quantified statement can often be reduced if, as each element is added to the permutation, the permutation is tested against one or more of the conditions in the quantified sentence. Additional elements are added to the permutation only if the preceding elements satisfy the condition(s). This approach is an example of shifting loop-invarient conditions out of a loop. Techniques for accomplishing it are well-known [Aho, Sethi & Ullman, 1986]. Two such optimizations have been investigated.

The first of these is called *code-motion-around-quantifiers*. It applies to both *LE* and *UEQ* terms. It moves conjuncts and disjuncts outside of the scope of quantifiers whenever possible. For example, it would translate Equation 2.1 into Equation 2.2.

$$\forall u, v \epsilon S, (p_1(u) \wedge p_2(u, v)) \tag{2.1}$$

$$\forall u \epsilon S, (p_1(u) \wedge \forall v \epsilon S, p_2(u, v)) \tag{2.2}$$

Equation 2.2 is an improvement over Equation 2.1 because in cases where $p_1(u)$ is not satisfied, no bindings of variable v need be considered.

The second of these optimizations is called *code-motion-around-summations*. It applies to *UEQ* terms. It moves conjuncts and disjuncts outside of the scope of summation statements whenever possible. This optimization is slightly more subtle than moving code around quantifiers, because summation statements count the number of times a condition is satisfied. Care must be taken to increment counters appropriately so that this optimization does not change the computation performed by the feature. For example, Equation 2.3 is translated into Equation 2.4.

$$\sum_{u \epsilon S} \sum_{v \epsilon S} \begin{cases} \text{if } p_1(u) \wedge p_2(u,v) & 1 \\ \text{otherwise} & 0 \end{cases} \tag{2.3}$$

$$\sum_{u \epsilon S} \begin{cases} \text{if } \neg p_1(u) & 0 \\ \text{otherwise} & \sum_{v \epsilon S} \begin{cases} \text{if } p_2(u,v) & 1 \\ \text{otherwise} & 0 \end{cases} \end{cases} \tag{2.4}$$

Equation 2.4 is more efficient than Equation 2.3 because in cases where $p_1(u)$ is not satisfied, no bindings of variable v need be considered.

By themselves, the code motion optimizations produce significant improvements in the costs of features. However, an even greater improvement is realized when code motion optimizations are performed after loop reordering.

2.4.2 Loop Reordering

Quantified statements and summation statements are evaluated using program loops. *UEQ* features frequently have several summations nested without intervening computations, while *LE* features frequently have several quantifiers nested without intervening statements. Nested summations and nested quantifiers are evaluated with nested loops.

Although the order of a series of nested loops is irrelevant to the computation, it determines the effectiveness of the code motion optimizations. The ordering of a series of loops determines how far a conjunct or disjunct can move. For example, if in Equations 2.1 and 2.3 variable u occurred *after* variable v, then no code motion would be possible.

An improved ordering of nested loops can be obtained by a heuristic that places first those loops whose variables are referenced along the greatest number of subsequent execution paths. The intent of the heuristic is to make it possible for the code motion optimizations to move as many conjuncts and disjuncts as possible out of the inner loops.

The combination of loop reordering and code motion heuristics can make a dramatic improvement in the speed of a feature. The speeds of several features described in Appendix D were increased by a factor of 15.

2.5 Estimates of Feature Costs

It is straightforward to calculate a lower bound on the cost of each term, where *cost* is defined as the time needed to assign the term a value. A lower bound can be found by considering the number of quantifiers in the term's definition, and the sizes of the sets to which they apply. This bound does not also serve as an upper bound, because it assigns $O(1)$ cost to recursive functions and functions whose definitions are unknown.

Lower bound estimates of a term's cost enable one to supply terms to the learning algorithm incrementally, beginning with the cheapest, or to reject a term whose cost exceeds some threshold.

2.6 The CINDI Program

The transformations, optimization and cost estimation described above, and the syntactic pruning rules described in Section 3.1, are implemented in a computer program called CINDI. Its input is a problem specification expressed in first order predicate calculus. Its output is a set of new terms and estimates of their costs, in either C or a Lisp-like language.

The CINDI program can be viewed as a translation program: search problem specifications are translated into features. The character string representation is inconvenient for complex translation tasks because it does not represent the hierarchical structure of logical and arithmetic expressions. The CINDI program uses a directed acyclic graph (DAG) as its internal representation, because a DAG is a hierarchical representation. The internal nodes of the DAGs created by CINDI represent arithmetic and logical operations. A node's children are the arguments of the operation. Leaf nodes represent atomic values that cannot be decomposed further. The atomic values used by CINDI are numbers, names, *TRUE* and *FALSE*.

Although knowledge-based feature generation is presented above as an unordered set of transformations, pruning heuristics and optimizations, they work best when applied in a particular order (Figure 2.8). Heuristic feature selection should be performed as early as possible, in order to prevent the proliferation of duplicate or useless terms. Code motion optimizations should be delayed until after *all* transformations are applied, in order that optimization not alter what the features compute. If code motion optimizations are applied to directed acylic graphs *before* the *AE-E*, *AE-M* and *UEQ* transformations are applied, a different set of features is produced.

The CINDI feature create program implements the algorithm described in Figure 2.8. Code generation from the directed acyclic graphs is straightforward. See [Aho, Sethi & Ullman, 1986] for a discussion of the issues involved. CINDI annotates the code for each feature with information about how the feature was created. The information produced is sufficient to reconstruct the 'family tree' of each feature created by CINDI. Examples of features produced by CINDI are shown in Figures 2.9 and 2.10, and Appendices A through D.

1. Parse the input, producing directed acyclic graphs for the goal and the precondition of each operator.
2. Apply the *LE* transformation recursively to each graph. The result is a set of graphs, each representing one *LE* term.
3. Apply the loop reordering optimization to each graph.
4. Use syntactic pruning heuristics to delete graphs that would result in duplicate or useless terms.
5. Apply the *AE-M*, *AE-E* and *UEQ* transformations to each graph.
6. Apply code motion optimizations to each graph.
7. Estimate feature costs.
8. Generate code, in either C or a Lisp-like language.

Figure 2.8 The CINDI feature creation algorithm.

CINDI is a relatively fast program. Each set of features described in this dissertation (Chapters 5-8, Appendices A-D) was generated in just a few seconds on a DECStation 5000.

```
;  id=5, Author=LE, status=0, parent-id=user
;  Optimizations:  3 6
f_6:
(EXISTS u in B
  (FORALL w IN B
    (! (on w u))))
```

Figure 2.9 The Lisp-like code written by CINDI for an *LE* term. The feature is descended from an operator condition that some block have an empty top.

```
/*  id=5, Author=LE, status=0, parent-id=user  */
/*  Optimizations:  3 6   */
float f_6 ()
  {
  float CINDI_stack[100];
    { /* Exists */
    B_element u;
    for (u=first_element_in_B();
         isa_element_in_B(u);
         u=next_element_in_B(u))
      {
        { /* ForAll */
        B_element w;
        for (w=first_element_in_B();
             isa_element_in_B(w);
             w=next_element_in_B(w))
          {
          CINDI_stack[1] = on(w, u);
          CINDI_stack[0] = (1 - CINDI_stack[1]);
          if (CINDI_stack[0] == 0) break;
          }
        }
      if (CINDI_stack[0] != 0) break;
      }
    }
  return (CINDI_stack[0]);
  }
```

Figure 2.10 The C code written by CINDI for an *LE* term. The feature is descended from an operator condition that some block have an empty top.

CHAPTER 3
FEATURE SELECTION

Like other heuristic feature generation algorithms, knowledge-based feature generation sometimes create features that are either *constant*, *redundant* or *irrelevant*. A constant feature is one whose value never changes. A redundant feature is one whose value can be recognized by the learning algorithm to be a function of the value(s) of other feature(s). An irrelevant feature is one whose value does not correlate with the desired concept.

All three types of feature are undesirable because they define unnecessary dimensions in the space searched by the inductive learning algorithm. When unnecessary dimensions are added, the size of the learning algorithm's search space increases while the density of concepts that describe the examples remains constant or decreases. These effects make it more difficult for the learning algorithm to find a concept that describes the examples.

Most inductive learning algorithms attempt to solve the problem of unnecessary features with a *feature selection* component that identifies and discards unnecessary features. Some constructive induction algorithms also perform feature selection (e.g. [Matheus & Rendell, 1989]). One advantage of performing feature selection during or immediately after constructive induction occurs when feature selection for the constructive induction algorithm requires less effort than feature selection for the inductive learning algorithm. A second advantage occurs when the constructive induction algorithm produces a fixed number of features at a time. In this case, reducing the number of unnecessary features increases the number of useful features that are generated.

Knowledge-based feature generation can produce constant, duplicate, and redundant features, so it is worth questioning whether any of these might be recognized and eliminated during or immediately after feature generation. Three different techniques can help identify unnecessary features: empirical, analytic, heuristic.

Most constructive induction algorithms that address the problem of unnecessary features do so with empirical, or *data-directed*, solutions. The two most common approaches are either to monitor how well each feature's value correlates to the desired class, or to monitor the utility of each feature to the learning algorithm. STAGGER [Schlimmer & Granger, 1986] and IB3-CI [Aha, 1991] adopt the former approach, while CITRE [Matheus, 1990a; Matheus, 1990b] adopts the latter. In both approaches, features with low correlation or low utility are periodically discarded.

Data-directed feature selection algorithms work well when data is available, but knowledge-based feature generation does not use data for feature creation. One could impose the requirement that data be available for feature selection, but doing so is not necessary. Inductive learning algorithms already decide, based upon data, which features to include in the creation of a concept. The inductive learning algorithm's feature selection would be augmented best by heuristic or analytic feature selection *before* the data are represented in the new features.

No constructive induction algorithm performs analytic, or *theory-directed*, feature selection. This might appear surprising, because several constructive induction algorithms have domain theories that they could consult. MIRO [Drastal, Czako & Raatz, 1989] and ZENITH [Fawcett & Utgoff, 1992; Fawcett & Utgoff, 1991] both use domain theories to guide the construction of new terms, but use data-directed methods to perform feature selection. STABB [Utgoff, 1986a] does not need a feature selection strategy, because its constructive induction algorithm does not generate unnecessary features.

It might be argued that explanation-based learning methods [Mitchell, Keller & Kedar-Cabelli, 1986] perform theory-directed feature selection, because those features that are not required by the domain theory are not used in the resulting concept description. Explanation-based learning methods also require access to data, but their data requirements are small. A theory-directed feature selection algorithm based upon explanation-based learning might be possible when domain theories are tractable. However, knowledge-based feature generation is designed to generate features for search problems with intractable domain theories. Performing explanation-based learning with intractable domain theories is the subject of much of the current research in explanation-based learning (e.g., [Flann, 1990; Tadepalli, 1989]).

Finally, several constructive induction algorithms use heuristic methods to perform feature selection. CITRE [Matheus, 1990a; Matheus, 1990b] and OXGATE [Gunsch, 1991] both use domain-specific heuristics to eliminate features thought to be unnecessary. BACON [Langley, Bradshaw & Simon, 1983] uses domain-independent heuristics to determine which terms are most useful in developing a concept. Domain-specific heuristics are not appropriate for knowledge-based feature generation, because it is designed to be domain-independent. However, domain-independent heuristics are appropriate. The possibility of developing domain-independent heuristics that eliminate some of the unnecessary features produced by knowledge-based feature generation is pursued in the next section.

3.1 Heuristic Feature Selection

The *LE* transformation creates terms by extracting embedded expressions from a statement in first order predicate calculus. Sometimes taking an expression out of context causes it to become constant, or to duplicate other such expressions.

Constant and redundant terms are undesirable because they needlessly increase the dimensionality of the space searched by the inductive algorithm.

A set of three *syntactic pruning* rules recognize and delete constant or redundant *LE* terms. They are:

- **Duplicate pruning:** Deletes terms that are syntactically equivalent under canonical ordering and variable naming.
- **Quantified variable relation pruning:** Deletes sentences that state only whether or not two quantified variables are identical (e.g., $\forall v_1, v_2 \epsilon S,\ v_1 = v_2$).
- **Static relation pruning:** The adjacency relationship between squares of a chessboard is an example of a constant relation. Constant relations are identified manually, for reasons described below. Once constant relations are identified, terms that involve *only* constant relations are deleted automatically.

The remaining terms must be evaluated using other methods, for example based upon feedback from a learning algorithm.

3.1.1 Constant Relations

The CINDI program requires that constant relations be identified manually by an external, presumably human, source. This requirement is the result of an assumption that the problem specification may not specify every effect of every operator. If every effect of every operator were known to CINDI, then it would be straightforward to reason about which relations could be changed. All that would be required would be a simple check of the search problem specification for the existence of statements that add tuples to, or delete tuples from, the relation in question.

The assumption that the search problem specification may not specify every effect of every action is deliberate. Even if every effect of every action is known to the problem-solving system, some of that knowledge may be represented procedurally instead of declaratively. It would be a mistake to require a completely declarative specification, for two reasons. First, complete and correct declarative specifications for complex problems can be difficult to develop. Second, knowledge-based feature generation is based on the thesis that features can be created from *easily available* knowledge.

Imposing additional requirements would make knowledge-based feature generation and the CINDI program more powerful, but would also make them harder to use. In general, it is easier and faster for a person to identify those relations that are constant than to develop more detailed problem specifications.

3.2 Data-Directed Feature Selection

Knowledge-based feature generation lacks the data that is necessary to conduct data-directed feature selection. The absence of data enables knowledge-based feature generation to create an initial vocabulary for inductive learning, but also prevents it from empirically evaluating that vocabulary. The inability to evaluate features empirically prevents knowledge-based feature generation from detecting and deleting those useless features that are not identified by heuristic feature selection.

The inability of knowledge-based feature generation to evaluate features empirically is not a fatal flaw, because most inductive learning algorithms perform data-directed feature selection. Those features that escape knowledge-based feature generation's feature selection heuristics can be identified and discarded by the inductive learning algorithm. The principle criticism of this approach is not that it will not work, but that it is inefficient computationally.

Although different inductive algorithms perform feature selection differently, most algorithms share two characteristics. First, inductive algorithms have traditionally assumed that values exist for each feature. This assumption requires that the computational cost of evaluating every feature be incurred, even if some features are subsequently ignored by the inductive algorithm (although see [Tan & Schlimmer, 1990] for an exception). Second, the ancestory and computational expense of a feature have rarely been considered in data-directed feature selection algorithms (although, see [Tan & Schlimmer, 1990]). Such consideration is desirable, because some features may require much more computation than other features.

It is not difficult to resolve these inefficiencies, but to do so for each inductive algorithm is beyond the scope of this thesis research. The solution adopted for this dissertation is to create a single data-directed feature selection algorithm that is independent of any single inductive learning algorithm. There has not previously been a single data-directed feature selection that could be used in conjunction with any inductive learning algorithm, but recent work on comparing representations for inductive learning makes the development of such an algorithm straightforward.

GFS is a data-directed feature selection algorithm, developed as part of this thesis research, that can be used with any inductive learning algorithm. GFS is not part of knowledge-based feature generation. Instead, it is a 'batch' algorithm that can be run on a set of examples to identify which of the features are useful to the inductive learning algorithm. The GFS algorithm requires a set of labelled examples, estimates of the cost or complexity of each feature, and the ability to run the inductive learning algorithm on selected subsets of the examples. Upon completion, the algorithm identifies which features are constant, which are duplicate, and which are redundant. A high-level description of the GFS algorithm is shown in Figure 3.1.

The GFS algorithm begins by eliminating constant features from the dataset. A feature is considered constant if its value does not change in any of the examples in the dataset.

```
1. Input feature costs, examples with classifications, and desired inductive learning algorithm.
2. Eliminate constant features.
3. Eliminate duplicate features.
4. While there remains a feature not known to be useful, do:
   (a) Call the most expensive feature not known to be useful $f_?$.
   (b) If according to ACR the set of features *with* $f_?$ is better than the set *without* $f_?$,
       • Then $f_?$ is known to be useful,
       • Else discard $f_?$.
```

Figure 3.1 The GFS feature selection algorithm.

In its second step, the GFS algorithm eliminates duplicate features from the dataset. Two features are considered duplicates if their values are identical in every example in the dataset. The syntactic pruning rules (Section 3.1) ensure that no two features have exactly the same definition, but different computations may nevertheless produce identical values. When two features are found to be duplicates, the GFS algorithm deletes the one whose cost or complexity is higher.

The final step of the GFS algorithm is to reduce the set of features to a minimal subset. A *minimal subset* of features is defined by the GFS algorithm to be the lowest cost subset of features whose discriminatory power is approximately equal to that of the original set of features. The intent in this final step is to reduce both the dimensionality of the learning algorithm's search space and the cost of evaluating features.

The set of features is reduced to a minimal subset by repeatedly finding the most expensive unseen feature in the dataset, and comparing the dataset *with* that feature to the dataset *without* that feature. Saxena's (1991) ACR algorithm (described below) is used to make the comparison, because ACR considers both the complexity and classification accuracy of concepts created with two *sets* of features. Decisions must be based on the relative value of *sets* of features for the interaction of one feature with other features to be considered. A feature is discarded only if the dataset with the feature is no better than the dataset without the feature. This *sequential backward selection* [Kittler, 1986] continues until each feature has been considered once.

The ACR algorithm was designed to compare the effectiveness of two different representations for a given learning task and a given learning algorithm [Saxena, 1991]. ACR makes its decision by conducting a series of *trials* in which it identifies the superior representation using just a subset of the examples. A *trial* consists of the following steps, performed repeatedly on randomly drawn samples of increasing size.

1. Obtain a random sample of examples expressed in each representation.
2. Train the learning algorithm on the examples in each sample. The result is two concept descriptions.
3. Calculate, for each concept, the number of bits necessary both to store the concept and identify the examples that it misclassifies. Call the number of bits the *estimated codelength* for the corresponding representation.
4. Use historical information to determine whether the smaller estimated codelength has been seen for each representation.
5. If it is determined that the smallest estimated codelength has been seen,
    (a) Then end the trial and report that the representation with the smaller estimated codelength is superior,
    (b) Else if it is possible to obtain larger samples,
        i. Then return to step 1 and do so,
        ii. Else report that there is no difference between the two representations.

ACR continues performing trials until it determines that the difference between the two representations is either statistically significant, or not statistically significant. If the difference between the two representations is large, only a small number of trials may be required before the difference between the two representations is recognized as statistically significant. When the difference is small, a large number of trials may be required.

Sequential backward selection with the ACR algorithm has several advantages. First, it is almost guaranteed to discard features with low discriminatory power while retaining features with high discriminatory power. The guarantee is not absolute because ACR is a probablistic algorithm. Second, given two features of equal discriminatory power, it is likely to discard the more expensive of the two. Third, ACR performs a comparison with less expense and higher confidence than running the inductive learning algorithm on large random samples of data expressed in each representation. Finally, one can adjust the 'greediness' of the algorithm by adjusting the threshold upon which ACR bases its decisions.

The 'greediness' of sequential backward selection can be adjusted by altering the certainty threshold used by ACR. If ACR must be 99% certain before it reports that one representation is better than another, then it will often report that there is no difference when just one feature is deleted from a set. When there is no difference, the feature is discarded because it is assumed not to add any new information. If the threshold is lowered to 95%, ACR is more likely to report a difference when just one feature is deleted from a set. A lower threshold makes sequential backward selection 'less greedy' by making it less likely that features with low discriminatory power will be discarded.

It is natural to ask whether features of low discriminatory power are worth saving. The answer is often 'yes'. If the presence of a particular feature is worth about 1% of classification accuracy in the learned concept, one might conclude that the expense of evaluating the feature is not worth the added accuracy that it obtains. However, deleting ten such features could reduce the accuracy of the learned concept by 10%, which is a substantial decrease.

In general, it is difficult to know *a priori* at what level to set the certainty threshold. One simple solution is to run the GFS algorithm repeatedly, each time with a lower threshold, until there is little or no improvement in concept accuracy.

A final issue is whether the first two steps of the GFS algorithm are really necessary, given the power of the sequential backward selection strategy, The answer is 'yes'. Although the sequential backward selection strategy is capable of identifying and eliminating constant features and duplicate features, the cost of doing so is high. Sequential backward selection with ACR involves repeatedly taking random samples of data, running the learning algorithm on them, and performing statistical tests on the results. In contrast, the special case tests for constant and duplicate features simply check the dataset for columns (feature values) that are constant or dulicates. The special case tests for constant and duplicate features operate so much more quickly than ACR that it is more practical to handle them separately.

CHAPTER 4

EXPERIMENTAL DESIGN AND METHODOLOGY

Knowledge-based feature generation is based on the thesis that functional features improve the ability of inductive algorithms to learn search control information. An empirical investigation was designed to test the thesis on inductive algorithms with differing representational abilities and search problems of varying complexity.

One can evaluate a constructive induction algorithm by measuring the effect of its new features on the accuracy of concepts learned by an inductive learning algorithm [Schlimmer & Granger, 1986; Pagallo, 1989; Matheus & Rendell, 1989; Aha, 1991]. Measurements are obtained by repeatedly partioning data into two disjoint sets, training the inductive algorithm on one set, and testing the accuracy of the learned classifier or concept on the other set [Langley, 1988]. This is called the *dataset accuracy* approach to evaluation in this dissertation, to emphasize that the results apply to an unchanging set of examples. One can either use a fixed number of measurements, as in n-way cross validation, or one can vary the number of measurements until a mean is determined with a specified confidence level. The latter approach is better suited for comparing results obtained with different amounts of data, because the sampling error can be controlled. Every result reported in this dissertation is accurate $\pm 1.0\%$, with 99% confidence. Errors due to small sample sizes were avoided by requiring that all measurements be based on 30 or more measurements. [1]

The dataset accuracy experiments provide results for each combination of representation and learning algorithm. For each combination there was just one independent variable, the size of the training set, which varied from 1% to 50% of the dataset. The test set was disjoint from the training set. The size of the test set was held constant at 50% of the dataset. The dependent variable was the accuracy of the learned evaluation function on the test set.

A larger test set could have been used, but doing so would have required a smaller training set, thereby providing less information about the independent variable. A smaller test set could have been used, thereby allowing a larger training set, but

[1]Thirty measurements is considered a conservative minimum number of measurements by statisticians [Devore, 1991].

the test data would then have been less representative of the domain than was the training data. The division of 50% (or less) for training and 50% for testing was adopted because it avoids both problems.

Experiments with datasets of instances are based upon two assumptions. First, if the dataset is itself a sample from some larger distribution, humans often assume that the dataset is representative of the larger population. Second, it is assumed that either each instance in the dataset is equally important, or the importance of a class of instances is proportional to its size in the population of instances[2]. These assumptions can be difficult to satisfy in complex domains.

Therefore, a second set of experiments was conducted to investigate the effectiveness of some of the learned concepts or classifiers at actually controlling a search algorithm. This second approach to testing is called the *performance accuracy* approach in this dissertation, to emphasize that results apply to problem-solving performance. Each of the evaluation functions tested in a performance accuracy experiment was created during one of the dataset accuracy experiments.

4.1 Learning Algorithms

The experiments tested the effectiveness of knowledge-based feature generation on inductive algorithms ranging from simple to complex. The expectation was that knowledge-based features would be most useful to those algorithms with limited representational abilities of their own, but that all algorithms would be affected by the presence of the new features. Knowledge-based feature generation produces numeric features, so the selection of algorithms was necessarily restricted to those that perform well with numbers. The algorithms chosen for these experiments were the Perceptron, Recursive Least Squares, C4.5 and Linear Machine Decision Trees learning algorithms.

The Perceptron and Recursive Least-Squares algorithms have weak concept description languages. Both algorithms represent concepts as a single linear threshold unit (LTU) [Nilsson, 1965; Young, 1984]. The Perceptron is guaranteed to converge eventually to a perfect classifier if examples and counterexamples are linearly separable. If examples and counterexamples are not linearly separable, the Perceptron cycles [Minsky & Papert, 1972]. The Recursive Least Squares algorithm is not guaranteed to converge to a perfect classifier, but its behavior is more stable when examples and counterexamples are not linearly separable.

The weakness of the LTU concept description language was considered desirable because it offers fewer opportunities for the learning algorithm to overcome any inadequacies in the instance description language. The success or failure of the

[2]In a database of preference pairs, some pairs may be more important than others. For example, one action might result in a slightly more efficient solution than another, while a third action would prevent a solution. The preference for an efficient solution is less important than the preference not to prevent a solution.

learned concept is more directly attributable to the instance description language when the concept description language is weak. Therefore, one might expect LTUs to be particularly appropriate for an empirical investigation of feature creation.

The C4.5 algorithm is an extension of the ID3 algorithm for creating decision trees [Quinlan, 1992]. C4.5 can represent perfectly concepts that are separable by a set of hyperplanes oriented perpendicular to an axis. Numeric attributes are incorporated into the symbolic decision-tree model by automatically mapping them into Boolean attributes that represent whether the number is above or below a threshold. C4.5 can learn accurately more complex concepts than can the Perceptron or RLS, because of its more powerful concept description language. The power of C4.5 was considered desirable because C4.5 is representative of inductive learning algorithms used commonly.

The Linear Machine Decision Tree (LMDT) algorithm is an extension of the Perceptron Tree algorithm for creating decision trees [Utgoff, 1989; Utgoff & Brodley, 1991]. LMDT can represent perfectly concepts that are separable by a set of hyperplanes at any orientation. Numeric attributes are an inherent part of LMDT's concept description language. LMDT can learn accurately more complex concepts than can either RLS or C4.5, because of its more powerful concept description language. In particular, LMDT is able to create linear machines that act as new features. LMDT is more capable than either C4.5 or RLS at compensating for deficiencies in the instance representation language. As a result, one might expect it to be the least likely to need the features created by knowledge-based feature generation.

The four algorithms described above represent a spectrum of inductive learning algorithms that work well with numeric attributes. Two algorithms (Perceptron and RLS) learn linear concepts and two (C4.5 and LMDT) learn non-linear concepts. The four algorithms also differ in the power of their concept description languages. The Perceptron and RLS have the weakest languages, and LMDT has the strongest. Consequently, the Perceptron and RLS should be the most sensitive to the instance representation, while LMDT should be the least sensitive. The Perceptron, RLS and LMDT are "numeric" algorithms, while C4.5 is a "symbolic" algorithm that can handle numbers. Collectively, this set of four algorithms provided a means of testing the effectiveness of knowledge-based feature generation under a wide variety of conditions.

4.2 Experimental Domains

Experiments were conducted on four problems with search spaces ranging from small to intractable. The smaller problems (stacking blocks, tic-tac-toe) allowed the experiments to be conducted with perfectly accurate training data, and produced features with derivations that were short and easy to understand. The larger problems (Othello, printed wiring board (PWB) component placement) allowed experiments to be conducted with more complex features, but required less comprehensive training data. The larger problems also tested the ability of knowledge-based feature generation to "scale up" to problems of significant size.

4.3 Experimental Procedures

Each experiment consisted of five procedures: Problem specification, feature generation, training data generation, feature selection, and learning. The procedures are discussed individually in the subsections below. Each procedure was held constant, to the extent possible, throughout all of the experiments. Deviations from the described procedures were occasionally necessary to accomodate the characteristics of a particular problem. Any such deviations are noted in the descriptions of the individual experiments.

4.3.1 Problem Specification

Problem specification consists of 1) describing the search problem in First Order Predicate Calculus, and 2) determining the problem-solver's representation of search states. The search problem specification is necessary because it is the input to knowledge-based feature generation. The problem-solver's representation of search states is necessary as a baseline against which to judge features created by knowledge-based feature generation.

Whenever possible, outside sources were used to guide the development of the search problem specification and the baseline problem-solver's representation. When outside sources were unavailable, attempts were made to use "obvious" or "common sense" representations and specifications. All of the problem specifications, with descriptions of their adaptation or development, are provided in Appendices A though D.

4.3.2 Feature Generation

Knowledge-based features were created for a problem by running the CINDI program on a search problem specification and identifying constant relations manually in response to the program's questions. The program's output was always a set of features expressed in the C programming language. Each set of features created by CINDI was assigned a unique name beginning with the prefix "C-". For example, the set of features created by CINDI for stacking blocks was assigned the name C-BLK.

4.3.3 Training Data Encoding

One approach to learning an evaluation function is to train an inductive learning algorithm on a set of examples with known values. This approach works well in domains where a single value can be assigned accurately to each example. However, in some domains, particularly in complex domains, it may be unclear what value to assign a given search state.

An alternate approach is to train an evaluation function on state preferences pairs, in which each member of the pair is a state, and one state is designated as preferred to the other. Preference pairs are often easier to gather, because they can be obtained by observing a skilled problem-solver [Utgoff & Clouse, 1991]. For example, one might assume that a move made by a chess expert is preferred to each of the alternate moves not selected.

None of the learning algorithms selected (Perceptron, RLS, C4.5, LMDT) are capable of learning directly from pairs of examples. Instead, each algorithm was trained to recognize a preference relation on pairs of feature vectors. In all experiments, a preference relation $p(\vec{s_i}, \vec{s_j})$ was encoded as a *delta vector* $\vec{s_i} - \vec{s_j}$ and assigned the value $+1.0$. A negative training instance (i.e., a counterexample of the relation) was encoded as $\vec{s_j} - \vec{s_i}$ and assigned the value -1.0. The following subsections explain why the delta vector encoding of preference pairs is justified for each learning algorithm.

4.3.3.1 Linear Threshold Units (LTUs)

The Perceptron is trained on examples whose values are known only to be greater than or less than 0. The concept learned is a linear evaluation function that maps a feature vector to a numeric value. The key to getting the Perceptron to learn from state preference pairs is to encode the pair of examples as a single example. The *delta vector* encoding represents the preference pair $p(\vec{s_i}, \vec{s_j})$ as the feature vector $\vec{s_i} - \vec{s_j}$. Previous research showed that the delta vector encoding preserves the information necessary for a Perceptron to learn an evaluation function h over instances[Utgoff & Clouse, 1991]. The following description summarizes that research.

The goal is to train an evaluation function h that state i (represented by feature vector $\vec{s_i}$) should be rated more highly than state j (represented by feature vector $\vec{s_j}$).

$$h(\vec{s_i}) \;>\; h(\vec{s_j}) \tag{4.1}$$

A linear threshold unit h multiplies the feature vector $\vec{s}$ by a weight vector $\vec{W}$ to obtain $h(\vec{s})$. Therefore $h(\vec{(s)})$ can be rewritten as $\vec{W^T} \cdot \vec{s}$.

$$\vec{W}^T \cdot \vec{s_i} \;>\; \vec{W}^T \cdot \vec{s_j} \tag{4.2}$$

Equation 4.2 can be rewritten so that the two references to $\vec{W}$ are replaced by one. Doing so introduces the delta vector $\vec{s_i} - \vec{s_j}$.

$$\left(\vec{W}^T \cdot \vec{s_i}\right) - \left(\vec{W}^T \cdot \vec{s_j}\right) \;>\; 0 \tag{4.3}$$

$$\vec{W}^T \cdot (\vec{s_i} - \vec{s_j}) > 0 \tag{4.4}$$
$$h(\vec{s_i} - \vec{s_j}) > 0 \tag{4.5}$$

Therefore, Equations 4.1 and 4.5 are equivalent.

The Perceptron algorithm adjusts the weight vector of a linear threshold unit based only on the knowledge that $h(\vec{s})$ should be greater than or less than 0 [Nilsson, 1965; Minsky & Papert, 1972]. The equivalence of Equation 4.1 to Equation 4.5 shows that training a Perceptron on delta vectors is the same as training it to assign a higher value to the preferred of a pair of instances.

The RLS algorithm finds coefficients (weights) for a linear function h [Young, 1984]. RLS is trained that the desired or 'true' value of a given instance $\vec{s}$ is a given real number y. The RLS algorithm can be trained with delta vectors because Equations 4.1 through 4.5 hold for all linear functions. However, all that is known about the desired or true value of a delta vector is that it should be greater than (or less than) 0. Its exact value is not known.

One solution is to train RLS that $h(\vec{s_i} - \vec{s_j}) = c$ (and $h(\vec{s_j} - \vec{s_i}) = -c$) for some constant positive real number c. The advantage of this solution is that it enables an evaluation function to be learned from preference data. The disadvantage is that it represents all preferences as equally important or strong.

The value chosen for c is unimportant because RLS uses the following rules to adjust the weight vector $\vec{W}$, given an instance $\vec{s}$ and its desired value y.

$$\begin{aligned} P &= P - P\vec{s}[1 + \vec{s}^T P \vec{s}]^{-1} \vec{s}^T P \\ \vec{K} &= P\vec{s} \\ \vec{W} &= \vec{W} - \vec{K}[\vec{s}^T \vec{W} - y] \end{aligned}$$

In these equations, $y = c$ and P is a $n \times n$ matrix of time-varying values that represent the correlation of each pair of features. It is clear by inspection that the values of matrix P and vector $\vec{K}$ do not depend upon the value chosen for c. The value c influences only the scalar value $\vec{s}^T \vec{W} - y$ that is applied to the adjustment vector $\vec{K}$. Different values of c cause proportionally different adjustments to $\vec{W}$. The *ordering* that h imposes on search states is not affected by proportionally different changes to $\vec{W}$. Therefore, for the purposes of learning an evaluation function from preference data, the exact value of c is unimportant. The value 1.0 was chosen arbitrarily for use in the thesis experiments.

4.3.3.2 C4.5

The C4.5 algorithm builds univariate decision trees that classify objects into two or more classes [Quinlan, 1992]. C4.5 cannot learn a numeric evaluation function directly, because it has no mechanism for imposing an ordering on classes nor of generalizing results from one class to another. However, it was shown above that a

Perceptron can learn a numeric evaluation function indirectly by learning a preference relation $p(\vec{s_i}, \vec{s_j})$. The question addressed here is whether a similar approach can be used for C4.5.

A univariate decision tree for a two class problem is created by selecting for each node a Boolean test on one feature. Boolean tests are created for numeric features by testing whether the value is above or below a threshold determined automatically. The problem of constructing a decision tree is to determine at a given node which feature f_i to test.

A univariate test on the value of one feature in one instance is of limited use in constructing a preference predicate that must compare two instances. What is needed is a mechanism for comparing two or more feature values. The delta vector encoding of two feature vectors into one, described in previous sections, can address this problem.

A delta vector is created by subtracting one feature vector from another. At each node in the tree, C4.5 can test a numeric feature value by comparing it to a threshold determined automatically. The combination of delta vector encoding and comparison to a threshold enables C4.5 to test whether the k'th feature of instances s_i and s_j satisfies the constraint $s_{i_k} \geq s_{j_k} + t_k$, where t_k is a threshold determined automatically by C4.5. This test can be considered a simple form of multivariate test, although it occurs in a univariate decision tree.

The delta vector representation does not enable C4.5 to represent arbitrary mathematical constraints. However, empirical tests show that on traditional classification problems, C4.5 compares favorably with a multivariate learning algorithm [Brodley & Utgoff, 1992], suggesting that many problems of interest can be addressed with multiple constraints of moderate complexity. The delta vector representation also eliminates C4.5's ability to perform traditional univariate tests. However, univariate tests are of little use in a preference predicate, where the problem is to compare two instances.

If C4.5 determines that a node in the decision tree does not classify its instances correctly, the tree is extended with additional tests. This process continues until the data are classified correctly. The tree is then pruned, to avoid overfitting the data.

4.3.3.3 LMDT

The LMDT algorithm builds multivariate decision trees that classify objects into two or more classes [Utgoff & Brodley, 1991; Brodley & Utgoff, 1992]. LMDT cannot learn a numeric evaluation function directly, because it has no mechanism for imposing an ordering on classes nor of generalizing results from one class to another. However, it was shown above that a Perceptron could learn a numeric evaluation function indirectly by learning a preference relation $p(\vec{s_i}, \vec{s_j})$. The question addressed here is whether a similar approach can be used for LMDT.

LMDT is not capable of learning directly from pairs of examples. That problem can be postponed by saying that two instances $\vec{s_i}$ and $\vec{s_j}$ are to be encoded by a function e, specified below, that returns a single instance vector. Given an encoding function e, the desired outcome is the following.

$$p(\vec{s_i}, \vec{s_j}) \longleftrightarrow (\mathrm{LMDT}(e(\vec{s_i}, \vec{s_j})) = \mathit{TRUE}) \tag{4.6}$$

$$\neg p(\vec{s_i}, \vec{s_j}) \longleftrightarrow (\mathrm{LMDT}(e(\vec{s_i}, \vec{s_j})) = \mathit{FALSE}) \tag{4.7}$$

The actual class names are irrelevant. *TRUE* and *FALSE* are convenient, but other values could be used.

A linear machine decision tree is created by first trying to create a single linear machine that correctly classifies all instances. A linear machine is a set of n linear threshold units (LTUs), where n is the number of classes to be learned. In the case of a preference predicate, $n = 2$.

The problem of learning a linear machine for two classes is to find two weight vectors such that the following is true.

$$p(\vec{s_i}, \vec{s_j}) \longleftrightarrow \left(\vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j}) > \vec{W_2}^T \cdot e(\vec{s_i}, \vec{s_j})\right) \tag{4.8}$$

$$\neg p(\vec{s_i}, \vec{s_j}) \longleftrightarrow \left(\vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j}) < \vec{W_2}^T \cdot e(\vec{s_i}, \vec{s_j})\right) \tag{4.9}$$

LMDT initializes all weight vectors to 0, so $\vec{W_1}$ and $\vec{W_2}$ are initially equal. Each time the linear machine misclassifies an instance, both weight vectors are adjusted by the same amount, but in opposite directions. The magnitude of the adjustment is determined by a variable β that is gradually annealed from 1.0 to 0.0 [Utgoff & Brodley, 1991; Brodley & Utgoff, 1992]. For example, if an instance is classified as *TRUE* when it should be *FALSE*, the following adjustments are made.

$$\vec{W_1} = \vec{W_1} - \beta\left(\vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j})\right) \tag{4.10}$$

$$\vec{W_2} = \vec{W_2} + \beta\left(\vec{W_2}^T \cdot e(\vec{s_i}, \vec{s_j})\right) \tag{4.11}$$

If an instance is misclassified as *FALSE*, the signs of the adjustments above are reversed. Therefore, at all times, $\vec{W_1} = (-\vec{W_2})$, which allows a simplified restatement of the problem.

$$p(\vec{s_i}, \vec{s_j}) \longleftrightarrow \left(\vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j}) > -\vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j})\right) \tag{4.12}$$

$$\neg p(\vec{s_i}, \vec{s_j}) \longleftrightarrow \left(\vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j}) < -\vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j})\right) \tag{4.13}$$

For clarity, the remaining discussion focuses on $p(\vec{s_i}, \vec{s_j})$. An analagous argument holds for $\neg p(\vec{s_i}, \vec{s_j})$.

Simplification of Equation 4.12 yields the following.

$$\begin{aligned} \vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j}) + \vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j}) &> 0 \qquad (4.14) \\ 2\vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j}) &> 0 \qquad (4.15) \\ \vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j}) &> 0 \qquad (4.16) \end{aligned}$$

Equation 4.16 is exactly the problem of training a single LTU from preference data. Section 4.3.3.1 showed that a delta vector encoding suffices for this problem. The argument is reproduced below.

$$\begin{aligned} \vec{W_1}^T \cdot e(\vec{s_i}, \vec{s_j}) &> 0 \qquad (4.17) \\ \vec{W_1}^T \cdot (\vec{s_i} - \vec{s_j}) &> 0 \qquad (4.18) \\ \vec{W_1}^T \cdot \vec{s_i} &> \vec{W_1}^T \cdot \vec{s_j} \qquad (4.19) \\ h(\vec{s_i}) &> h(\vec{s_j}) \qquad (4.20) \end{aligned}$$

Therefore a single linear machine, trained on preference pairs encoded as delta vectors, implicitly learns a numeric evaluation function h.

LMDT builds decision trees composed of linear machines. If, after training, a linear machine still cannot correctly classify its training instances, LMDT extends the tree below that linear machine. The training procedure for each descendent linear machine is identical to the training procedure for its parent, except that fewer training instances are involved.

4.3.4 Training Data Generation

State preference training methods require a dataset of *preference pairs* in which each member of the pair is a state, and one state is designated as preferred to the other. The hypotheses being tested concerned the quality of the representations, not the characteristics of the learning algorithms, so every attempt was made to provide the learning algorithm with accurate, noise-free, training data.

For problems in which the search space was small, exhaustive search was used to generate preference pairs. Every state along a path to a goal was considered preferred to every state not along a path to a goal. Among states that were on paths to goals, states on shorter paths were preferred to states on longer paths.

For problems in which the search space was too large to search exhaustively, guidance from an expert was used when available. When expert guidance was not available, preference pairs were generated by perturbing states with known values. The descriptions of individual experiments provide more information about these cases.

4.3.5 Feature Selection

The syntactic pruning rules, discussed in Section 3.1, were applied automatically by CINDI. After training data were generated, the first two steps of the GFS algorithm were performed on the data to identify any remaining constant or duplicate features. GFS automatically removes features that are constant in the data. It also deletes from a set of duplicate features all but the least computationally complex, as estimated by CINDI. The resulting "pruned" subset of features had a "-P" suffix appended to its name. For example, the pruned set of features for stacking blocks was C-BLK-P.

An additional feature selection step could have been performed, but was not. The third step of the GFS algorithm is to perform sequential backward selection with the ACR algorithm, to further prune a set of features. This step was not performed because decisions made by ACR apply only to a specified learning algorithm. To be completely accurate, the last step of the GFS algorithm would have had to have been performed separately for each of the four learning algorithms tested. The result could have been four slightly different representations, all produced by CINDI, for each dataset accuracy experiment, which would have made the results more difficult to evaluate.

CHAPTER 5

EXPERIMENTS IN THE BLOCKS WORLD

The blocks-world has been one of the artificial domains most widely studied in Artificial Intelligence. In a blocks-world domain, an agent must manipulate one or more well-defined objects in a well-defined environment to achieve a well-defined goal. Blocks-world environments have been popular because they enable the researcher to control all experimental characteristics. In particular, both the size of the search space and the degree of uncertainty (if any) can be controlled easily.

In the *stacking blocks* problem [Callan & Utgoff, 1991a], the environment contains a table and a set of four blocks labelled A through D. The problem-solver's task is to stack the blocks as shown in Figure 5.1b. The order of blocks within a stack is important. A has to be on B, B has to be on C, C has to be on D, and D has to be on the table. Starting states are generated randomly.

There is no well-known specification for the stacking blocks task. However, several introductory Artificial Intelligence textbooks use a common set of operators and a similar representation for blocks-world tasks [Winston, 1977; Nilsson, 1980; Rich & Knight, 1991]. The problem specification designed for this experiment adhered to the set of operators and representational conventions shared by the introductory textbooks. The complete specification, annotated with comments, is contained in Appendix A.

One appealing characteristic of the blocks-stacking tasks is its simplicity. The problem specification is not complex, so it is not difficult to trace the creation of a feature from beginning to end. Another advantage of this problem is the small size of its search space. The distance of every state to the nearest goal state is easy to determine, which makes it possible to train the inductive learning algorithm with completely accurate information.

5.1 Training Data

Training data was obtained for the stacking blocks task by exhaustive search. Training data consisted of preference pairs, as described in Chapter 4. One state was

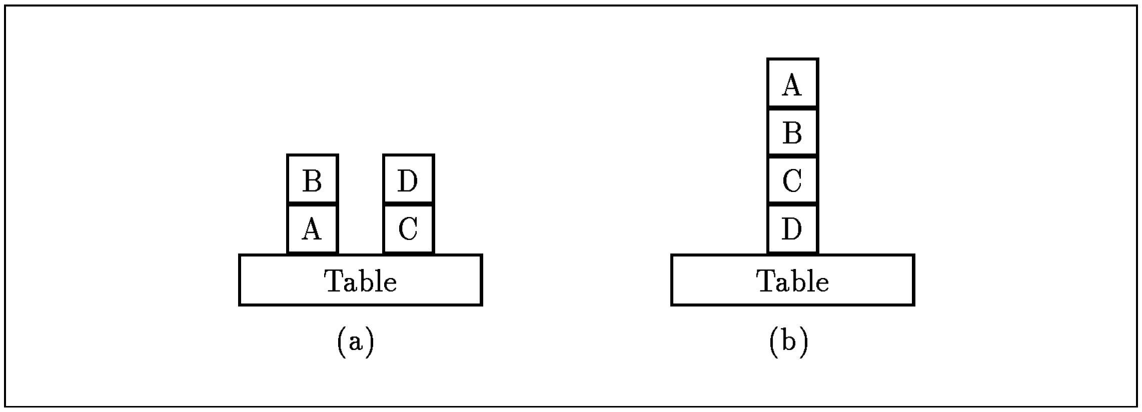

Figure 5.1 An initial state (a) and the goal state (b) for the stacking blocks problem.

Table 5.1 Distribution of the features CINDI created for stacking blocks.

	LE	UEQ	AE	Total
Generated by CINDI	9	4	0	13
Pruned by GFS steps 1 & 2	2	0	0	2
Total	7	4	0	11

preferred to another if and only if it was closer to a goal state than the other. Every state in the blocks-stacking task eventually leads to a goal, so no other preference criteria were necessary. The distance from a state to the nearest goal state was determined by breadth-first search.

5.2 Feature Creation

The CINDI program was applied to the specification contained in Appendix A. It generated 13 new features, distributed as shown in Table 5.2. The first two steps of the GFS algorithm identified two features as constant. The remaining set of 11 features is representation *C-BLK-P* in the experiments below. (The prefix *C-* indicates that CINDI created the features. The suffix *-P* indicates that the feature set was pruned.)

The problem specification for the blocks problem is not complex, so it is easy to trace the creation of a feature from beginning to end. For example, a precondition of the ***stack-block*** operator required that there be a block with an empty top ($\exists b_1, \neg\exists b_2, \text{on}(b_2, b_1)$). Two terms were created from that precondition. One was a Boolean term, created by the *LE* transformation, that indicated whether some block had an empty top. The other was a numeric term, created by the *UEQ* transformation, that indicated the percentage of blocks with empty tops.

Two hand-coded representations, labelled B1-B16 and L1-L4, were created for comparison with the terms created by CINDI. The hand-coded representations were intended to be the kind of representations that a problem-solver might use. Terms B1-B16 were Boolean terms that indicated the postition of the blocks. B1 indicated whether A was on B, B2 indicated whether A was on C, and so on. Terms L1-L4 were numeric. Each term indicated the position of one block. A term's value was 0 if its block was on top of A, 1 if its block was on B, and so on; its value was 5 if its block was on the table.

5.3 Accuracy on a Dataset

The dataset of 1,966 instances representing preferences in the stacking blocks task was randomly sampled to generate independent training and testing sets. The sizes of training and testing sets ranged from 20 examples (1% of the dataset) to 983 examples (50% of the dataset). The size of the testing set was fixed at 983 examples so that classification accuracies would be comparable among training sets of different sizes. Training and testing sets were disjoint.

The object of the experiment was to determine the average accuracy of the evaluation functions that would be created with different representations and different amounts of training data. The average accuracy for a particular representation and amount of training data was determined by repeatedly creating, training and then testing evaluation functions, until either 30 evaluation functions had been tested or average accuracy was determined $\pm 1.0\%$, with 99% confidence, whichever occurred later.

5.3.1 Perceptron

The Perceptron algorithm consistently created its most accurate evaluation functions with a combination of structural features and features created automatically by CINDI (Figure 5.2). The combination of knowledge-based and Boolean structural features was best for creating evaluation functions. The combination of knowledge-based and numeric structural features was second best, yielding evaluation functions about 4.0% less accurate.

The improvement in classification accuracy obtained by adding knowledge-based features to structural features was dramatic. Adding the *C-BLK-P* features created by CINDI to the *B1-B16* structural features yielded a 9.3% improvement. Adding the *C-BLK-P* features to the *L1-L4* structural features yielded a 20% improvement. The improvements in classification accuracy obtained by adding knowledge-based features to structural features are statistically significant ($P < 0.01$).

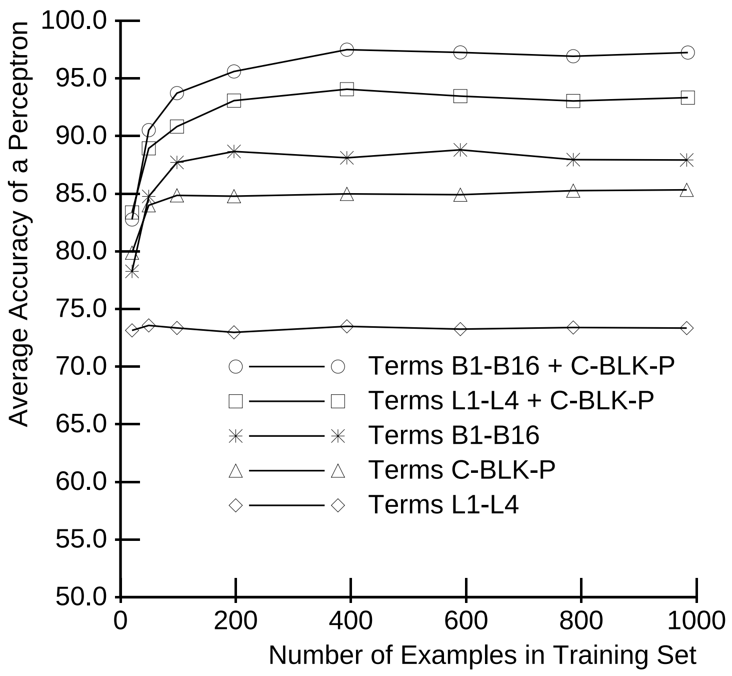

Figure 5.2 Average accuracy of five Perceptron evaluation functions for the stacking blocks problem.

Evaluation functions created with the *B1-B16* structural representation were about 2.5% more accurate than evaluation functions created with CINDI's *C-BLK-P* representation. The *C-BLK-P* evaluation functions were about 12% more accurate than evaluation functions created with the *L1-L4* structural representation. The differences in classification accuracy obtained with structural and knowledge-based features was statistically significant ($P < 0.01$).

The Perceptron training rule produces its most accurate classifiers when it is exposed repeatedly to each training instance and the classes are linearly separable. The minimum number of repetitions necessary is $2n$, where n is the number of features in a training instance [Nilsson, 1965]. One disadvantage to the Perceptron training rule is that there is no upper bound on the number of repetitions that might be necessary.

The experiments with the Perceptron rule were conducted with $4n$ repetitions, where n was the number of features in the representation. A few additional experiments were conducted with n ranging from 10 to 100, to ensure that four repetitions were reasonable for this data. The results obtained with $10 \leq n \leq 100$ were less than 1.0% different from results obtained with $n = 4$. Values of n greater than 100 were not tried due to the amount of computation that would have been required.

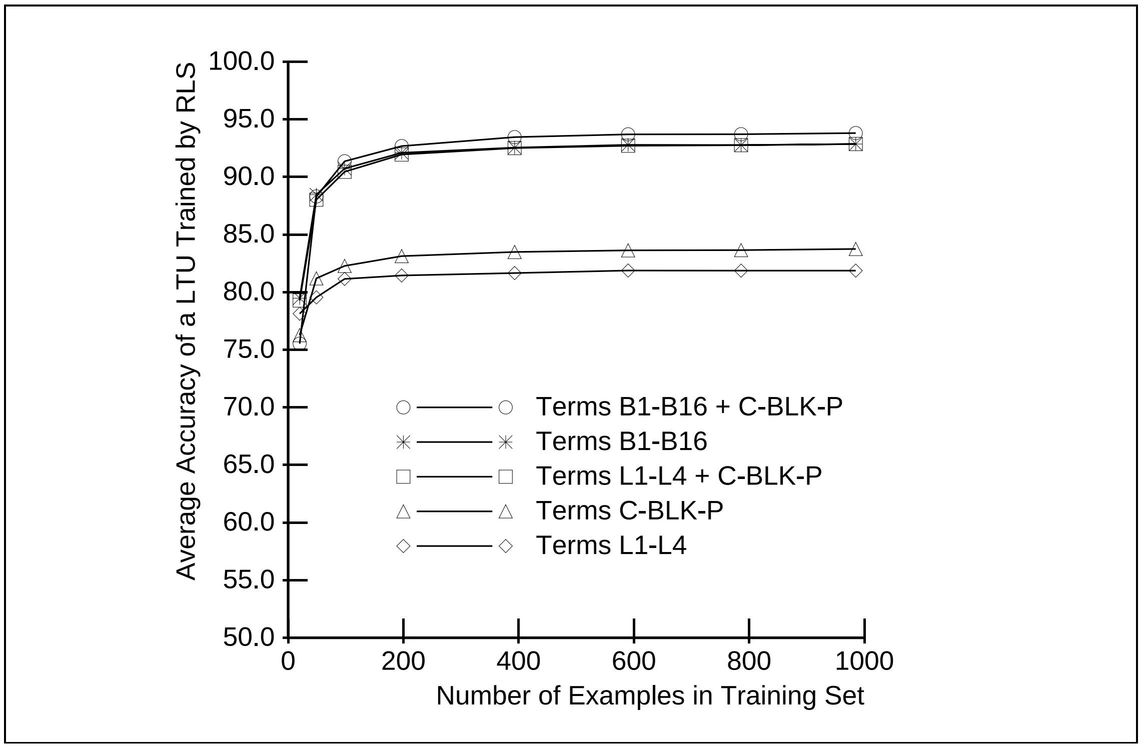

Figure 5.3 Average accuracy of five RLS evaluation functions for the stacking blocks problem.

5.3.2 RLS

The average performance of the B1-B16 representation was improved only slightly by the addition of the *C-BLK-P* terms (Figure 5.3). In contrast, the average performance of the L1-L4 representation improved by over 10%.

5.3.3 C4.5

The C4.5 algorithm consistently created its most accurate evaluation functions with a combination of structural features and features created automatically by CINDI (Figure 5.4). The combination of knowledge-based and Boolean structural features was best for creating evaluation functions. The combination of knowledge-based and numeric structural features was second best, yielding evaluation functions about 2.6% less accurate. The improvement in classification accuracy obtained by adding knowledge-based features to structural features was statistically significant ($P < 0.01$) for both the Boolean (*B1-B16*) and numeric (*L1-L4*) structural representations.

The *C-BLK-P* representation created by CINDI enabled more accurate evaluation functions than either of the structural representations. The difference between the

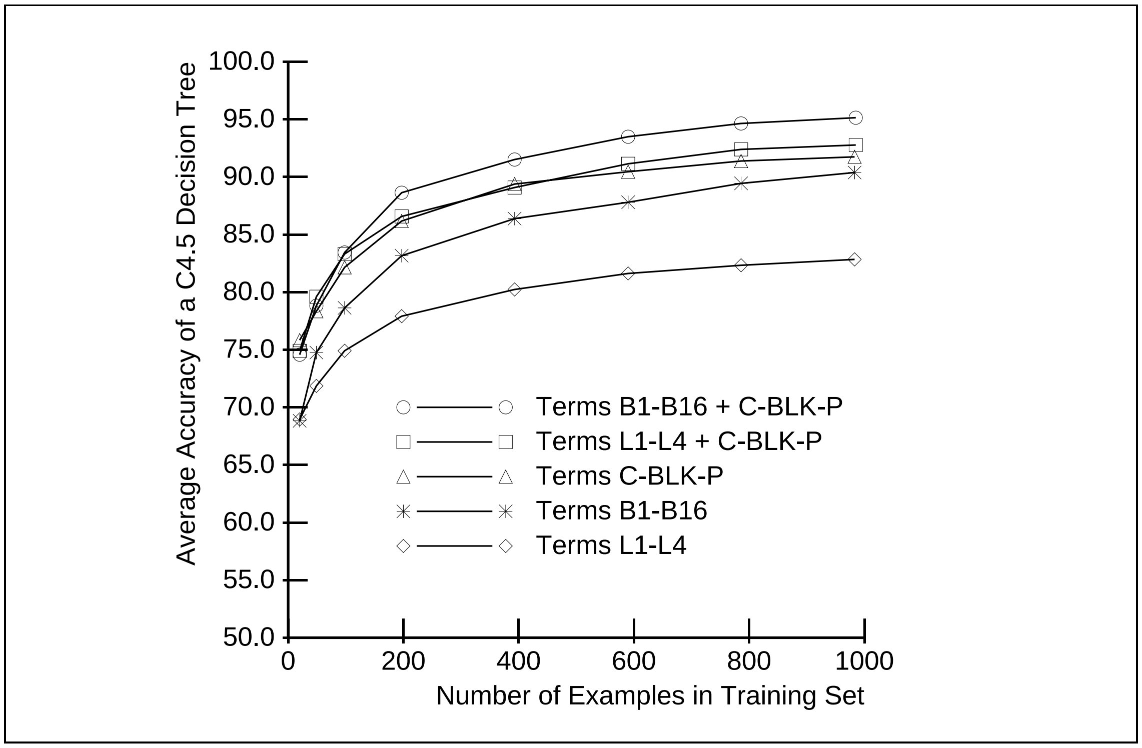

Figure 5.4 Average accuracy of five C4.5 evaluation functions for the stacking blocks problem.

C-BLK-P and *B1-B16* representations was small, but statistically significant ($P < 0.01$). Evaluation functions in the *B1-B16* representation were just 1.25% less accurate than evaluation functions in the *C-BLK-P* representation. The combination of the two was 3.75%-5.0% more effective than either alone.

Evaluation functions in the *L1-L4* structural representation were the least accurate. Adding *C-BLK-P* features to this representation improved the accuracy of evaluation functions by 10%, producing evaluation functions that were second best in these experiments.

5.3.4 LMDT

The LMDT algorithm consistently created its most accurate evaluation functions with a combination of structural features and features created automatically by CINDI (Figure 5.5). The combination of knowledge-based and Boolean structural features was best for creating evaluation functions. The combination of knowledge-based and numeric structural features was second best, yielding evaluation functions about 4.25% less accurate. The improvement in classification accuracy obtained by adding

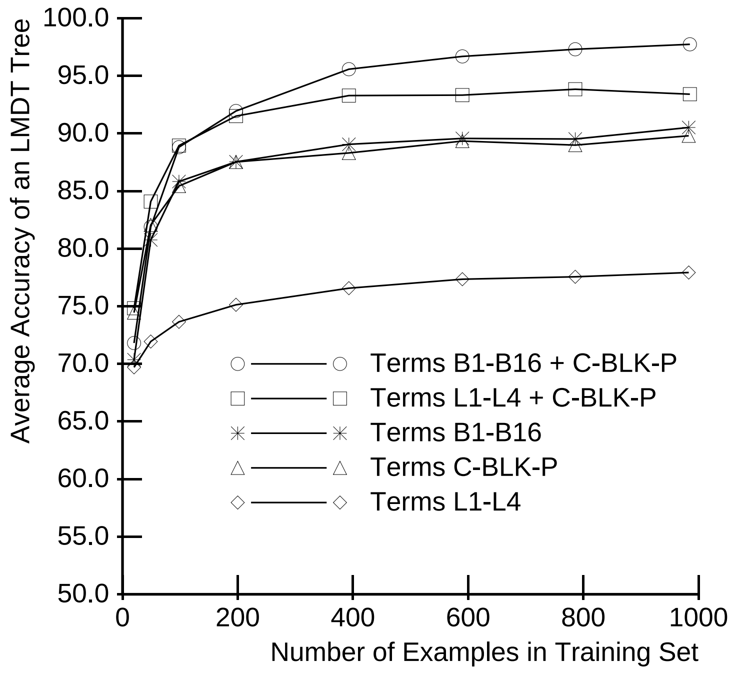

Figure 5.5 Average accuracy of five LMDT evaluation functions for the stacking blocks problem.

knowledge-based features to structural features was statistically significant ($P < 0.01$) for both the Boolean (*B1-B16*) and numeric (*L1-L4*) structural representations.

The *B1-B16* structural representation and the *C-BLK-P* representation created by CINDI were about equally useful for creating evaluation functions. The combination of the two was about yielded evaluation functions about 7% more accurate than did either alone.

Evaluation functions in the *L1-L4* structural representation were the least accurate. Adding *C-BLK-P* features to this representation improved the accuracy of evaluation functions over 15%, producing evaluation functions that were second best in these experiments.

5.4 Summary

One can rank representations according to how accurate are the evaluation functions that they enable with different learning algorithms. According to this metric, the *B1-B16+C-BLK-P* representation is the most useful, the *L1-L4+C-BLK-P* representation is second most useful, and the *L1-L4* representation is least useful.

The experiments disagreed about how useful the *B1-B16* and *C-BLK-P* representations are. C4.5 found the *C-BLK-P* representation more useful, LMDT found the two about equally useful, and the Perceptron and RLS both found *B1-B16* more useful. It is unclear why the four algorithms differed so much in their reactions to these two representations individually, because all four algorithms found the combination of the two representations to be the most useful.

CHAPTER 6
EXPERIMENTS WITH TIC-TAC-TOE

Previous research on feature creation has considered the problem of creating features for the game tic-tac-toe [Matheus, 1990a; Matheus, 1990b; Aha, 1991]. The problems considered previously were to identify those tic-tac-toe states that were instances of the goal state. The problem of identifying goal states would be a poor test of knowledge-based feature generation, because the input to KBFG is a search problem specification that specifies the goal. The *LE* transformation creates from the goal specification Boolean features for *three in a row*, *three in a column*, and *three in a diagonal*, which collectively make 100% accurate classification straightforward.

A better test of knowledge-based feature generation is to learn which of two available moves leads more quickly to winning the game. This experiment conducts such a test.

6.1 Training Data

A set of 2,816 tic-tac-toe preference pairs was generated by exhaustive search. Each preference pair consisted of two states. States from which X could win were preferred to states from which X could not win. Among states from which X could win, states closer to a winning state were preferred. No goal states were included in the set.

6.2 Feature Creation

The CINDI program created 16 features, distributed as shown in Table 6.2, from a specification of the tic-tac-toe goal and operator. The specification is provided in Appendix B. The first two steps of the GFS algorithm were applied to the full set of 2,816 instances. The algorithm identified and eliminated 13 constant features. The remaining 3 features were labelled the *C-TIC-P* representation.

Representation *C-TIC-P* was compared with two other representations of tic-tac-toe states:

Table 6.1 Distribution of the features CINDI created for tic-tac-toe.

	LE	UEQ	AE	Total
Generated	8	8	0	16
Pruned by GFS steps 1 & 2	7	6	0	13
Total	1	2	0	3

- *SQ9:* a representation with one numeric feature for each square of the tic-tac-toe board. A feature was -1 if O owned the square, 0 if the square was blank, and $+1$ if X owned the square.
- *SQ9+C-TIC-P:* a representation created automatically by combining the *C-TIC-P* and *SQ2* representations.

6.3 Accuracy on a Dataset

The dataset of 2,816 tic-tac-toe instances was sampled randomly to generate independent training and testing sets. The sizes of training sets ranged from 28 examples (1% of the dataset) to 1,408 examples (50% of the dataset). The size of the testing set was fixed at 1,408 examples, so that classification accuracy would be comparable among training sets of different sizes. Training and testing sets were disjoint.

The object of the experiment was to determine the average accuracy of the evaluation functions that would be created with different representations and different amounts of training data. The average accuracy for a particular representation and amount of training data was determined by repeatedly creating, training and then testing evaluation functions, until either 30 evaluation functions had been tested or average accuracy was determined $\pm 1.0\%$, with 99% confidence, whichever occurred later.

6.3.1 Perceptron

The Perceptron algorithm consistently created its most accurate evaluation functions with the *SQ9+C-TIC-P* representation. The *SQ9* and *C-TIC-P* representations consistently yielded evaluation functions 11% less accurate, on average, then when combined (Figure 6.1). The differences between the *SQ9+C-TIC-P* learning curve and the other two learning curves are statistically significant ($P < 0.01$).

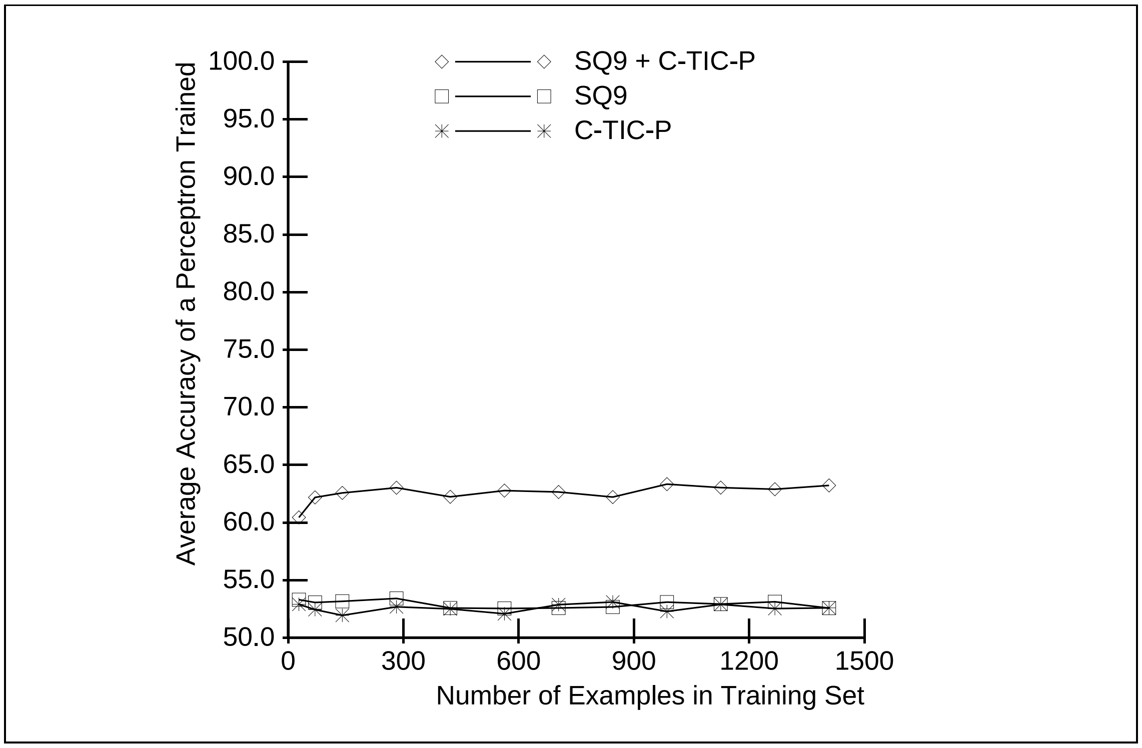

Figure 6.1 Average accuracy of three Perceptrons at identifying the preferred of two tic-tac-toe states.

6.3.2 RLS

The RLS algorithm consistently created its most accurate evaluation functions with the *SQ9+C-TIC-P* representation. The *SQ9* and *C-TIC-P* representations consistently yielded evaluation functions 10-12% less accurate, on average, then when combined (Figure 6.2). The differences between the *SQ9+C-TIC-P* learning curve and the other two learning curves are statistically significant ($P < 0.01$).

6.3.3 C4.5

The C4.5 algorithm consistently created its most accurate evaluation functions with the *SQ9+C-TIC-P* and *C-TIC-P* representations. The *SQ9* representation consistently yielded evaluation functions 10-12% less accurate, on average, then when combined with *C-TIC-P* (Figure 6.3). The differences between the *SQ9* learning curve and the other two learning curves are statistically significant ($P < 0.01$). The differences between the *SQ9+C-TIC-P* and *C-TIC-P* learning curves are statistically significant for training sets containing at least 563 examples (20% of the data).

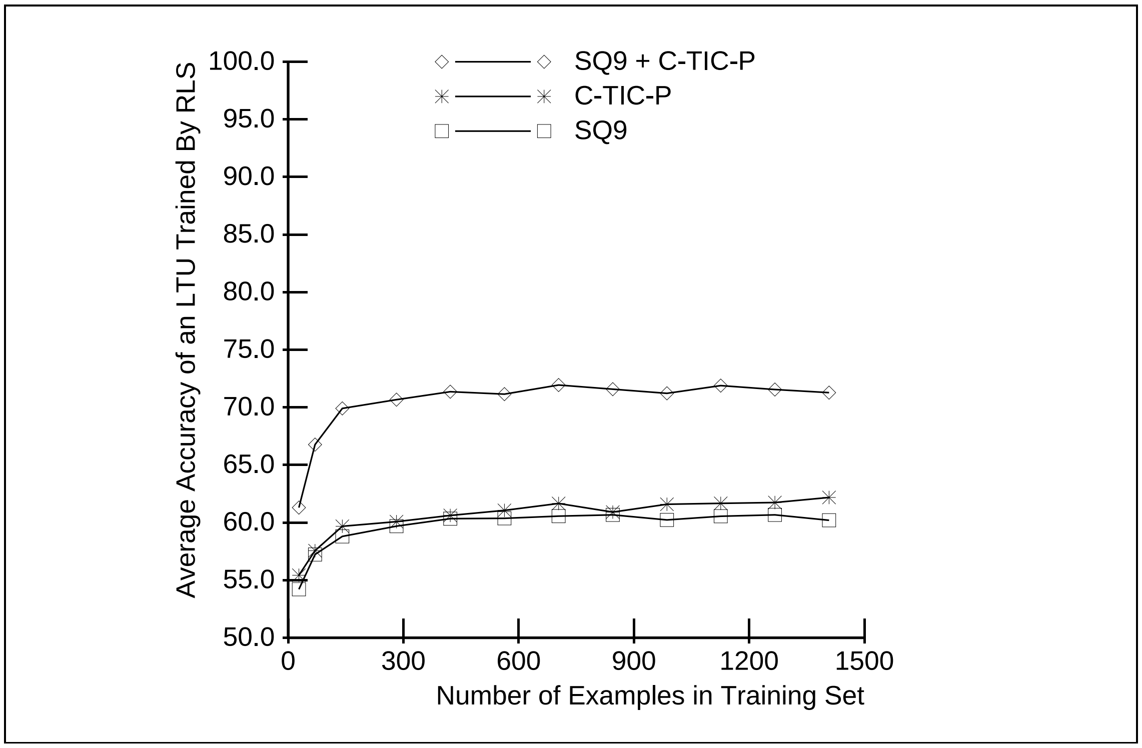

Figure 6.2 Average accuracy of three LTUs trained by RLS at identifying the preferred of two tic-tac-toe states.

6.3.4 LMDT

The LMDT algorithm avoids overfitting the training data by performing a 'pruning' phrase after the decision tree is built. The pruning phase requires a set of instances that is disjoint from the set of training data. In the experiments with LMDT, the pruning data was always 20% of the training data available to LMDT.[1]

The LMDT algorithm consistently created its most accurate evaluation functions with the *SQ9+C-TIC-P* representation (Figure 6.4). The *C-TIC-P* yielded evaluation functions that were less accurate, but still more accurate than those obtained with the *SQ9* representation. The differences between the learning curves are statistically significant ($P < 0.01$) for training sets of at least 70 examples (2.5% of the dataset).

[1]The figure 20% was suggested by Carla Brodley, coauthor of LMDT (personal communication).

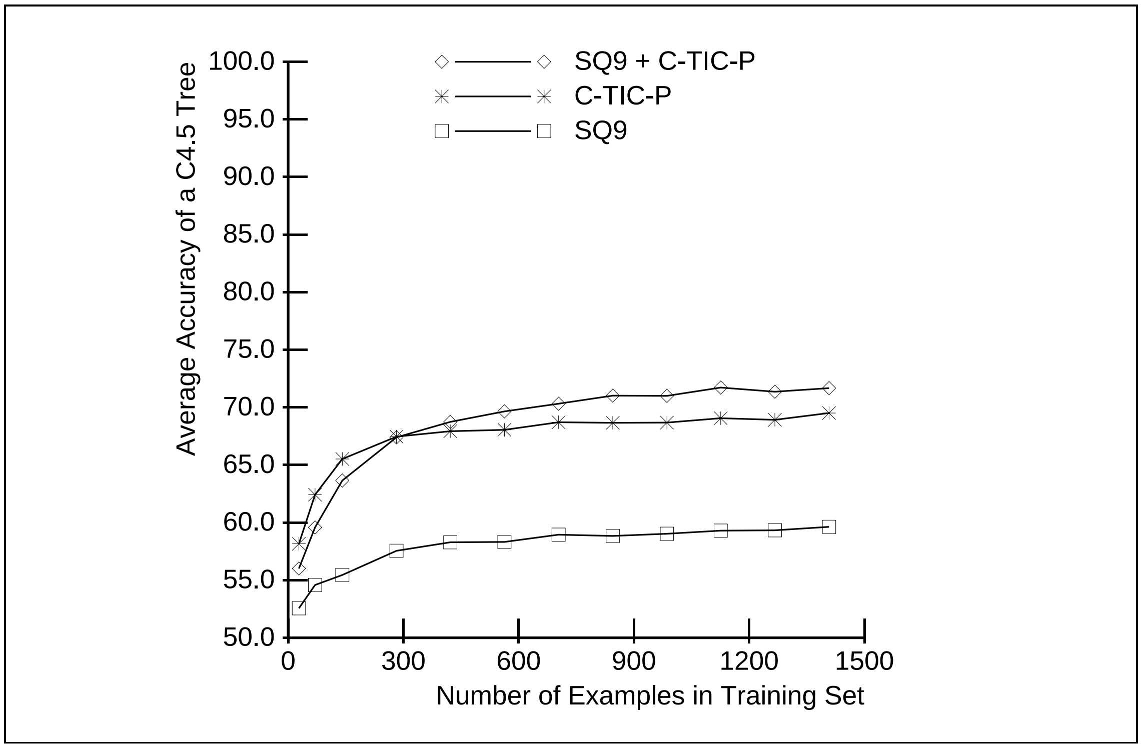

Figure 6.3 Average accuracy of three C4.5 decision trees at identifying the preferred of two tic-tac-toe states.

6.4 Summary

One can rank representations according to the accuracy of the evaluation functions that they enable with different learning algorithms. According to this metric, the *SQ9+C-TIC-P* representation is the most useful, the *C-TIC-P* representation is second most useful, and the *SQ9* representation is least useful. Three of the four learning algorithms yielded the same ranking of representations. The fourth, the Perceptron, found the *SQ9* and *C-TIC-P* representations about equally useful.

Adding the *C-TIC-P* features created by CINDI to the *SQ9* structural features improved by about 11% the accuracy of evaluation functions produced by all four algorithms. The algorithms differed in how useful they found the *C-TIC-P* features alone. C4.5 found them almost as useful as the combined representation, while the Perceptron and RLS found them about as bad as the structural representation *SQ9*.

If evaluation functions are ranked by the accuracy of the evaluation functions they created with the combined representation, then RLS was ranked first, C4.5 was ranked a close second, LMDT was ranked third, and the Perceptron was ranked last. This ordering of learning algorithms is surprising, because with the exception of the Perceptron, the ordering is inversely related to the representational power of

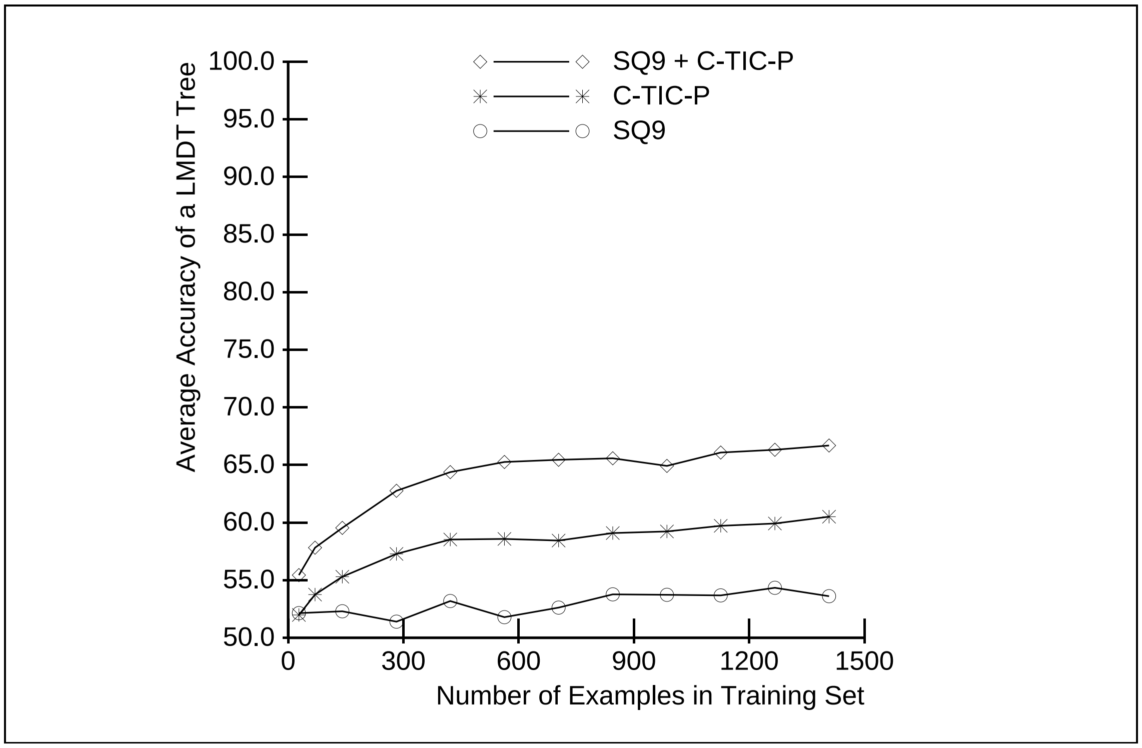

Figure 6.4 Average accuracy of three LMDT trees at identifying the preferred of two tic-tac-toe states.

the concept description languages. One normally expects the more powerful concept description language to perform as well as, or better than, more restricted languages.

There are two reasons for LMDT's performance relative to C4.5. First, this version of LMDT used the accuracy of a node to decide when to prune a feature from that node; if pruning the feature appeared to cause no significant loss of accuracy, the feature was pruned. Later versions of LMDT used an information gain metric, which produced more accurate concepts. A brief test with that version of LMDT showed that about half of the difference between LMDT and C4.5 was due to the pruning criteria. Second, LMDT was trained on 20% less data than C4.5, because part of the training data were set aside for LMDT's pruning phase. C4.5 trains on all of its data, and then estimates the error that is caused by overfitting the data.

The data support the hypothesis that the strong performance of RLS relative to C4.5 and LMDT is due to the nature of the desired concept. LMDT uses the Thermal Perceptron rule to train its linear machines. The author's experience with RLS and the Thermal Perceptron has been that RLS usually produces more accurate concepts than the Thermal Perceptron. LMDT only has an advantage over RLS if the desired concept is highly non-linear. The desired concept for tic-tac-toe is unknown, but examination of the decision trees produced by LMDT revealed that the most accurate trees consisted of just one node. If the desired concept is nearly linear, as the LMDT decision trees suggest, then it is not surprising that RLS would produce more accurate concepts than LMDT. It is also not surprising that RLS would produce more accurate concepts than C4.5 in such a situation, because C4.5 cannot create hyperplanes at arbitrary orientations.

CHAPTER 7

EXPERIMENTS WITH PWB COMPONENT PLACEMENT

Printed wiring board (PWB) component placement is the problem of finding locations on a circuit board for each of a set of electronic components, subject to a set of constraints called design rules [Breuer, Friedman & Iosupovicz, 1981; Preas & Karger, 1986]. PWB component placement has been studied for years because of its industrial applications, but it is also an example of the more general bin packing under constraints problem. The complexity of the problem depends upon the number of constraints, objects and possible object locations. In general, PWB placement problems are NP-hard [Sahni & Bhatt, 1980].

Utgoff's POOL algorithm is a parallel, object-oriented, approach to the PWB component placement problem[1]. The algorithm (Figure 7.1) consists of an initialization phase, and a loop executed repeatedly until the goal is reached or deemed unreachable. The algorithm treats components as active agents responsible on each cycle for moving a little closer to a final configuration that satisfies all constraints. Experiments showed that it was possible to control the movements of the components with heuristic evaluation functions created manually. It remained an open question whether such evaluation functions could be created automatically.

Knowledge-based feature generation can transform a set of design rules into a set of features that describe a component's location. If the features are appropriate for the task, and if a suitable source of training data is available, it should be possible for an inductive algorithm to learn an evaluation function for PWB component placement. One advantage of an automatic approach would be to lower the cost of changing design rules, because the manual labor involved would be reduced.

The final set of experiments investigated whether knowledge-based feature generation produced features that were useful for PWB component placement. It was recognized at the outset that PWB component placement was a risky domain in which to test knowledge-based feature generation, because there was no obvious source of training data for the POOL algorithm. However, the lack of good training data is a characteristic shared by many complex problems. PWB component placement

[1]Unpublished algorithm. Personal communication.

1. Group all movable components in the center of the board.
2. Use an iterative force-directed placement algorithm to locate components near other components with which they are connected electrically. Overlapping component locations are allowed.
3. Until all design rules are satisified or deemed unsatisfiable, do:
   (a) Instruct components to determine where they would like to move next. Each component uses a heuristic evaluation function to select from among its current location and movement by a random small amount along the eight major compass directions (N, NE, E, ..., SW). Components do not communicate.
   (b) Instruct each component to move to the location it selected above.

Figure 7.1 The POOL algorithm.

was viewed as an opportunity to investigate how knowledge-based feature generation might perform in highly complex domains with noisy data.

7.1 Training Data

The size of the search space for PWB component placement depends on the number of components and component locations. The PWB component placement problem used in these experiments was a double-sided circuit board from a large minicomputer designed and manufactured during the late 1980s. The circuit board contained 348 movable components on side one, and 216 movable components on side 2, for a total of 564 movable components.[2] The POOL algorithm allows nine locations to be considered for each component on each cycle, yielding a branching factor of 5,076.

The length of a solution to the PWB component placement problem depends upon the skill of the evaluation function controlling the search. Solutions were obtained with manually-designed evaluation functions in as few as 225 cycles, although 300-350 was more common.

The search space for the PWB component placement is too complex to enumerate. It is unclear how to sample the space randomly and uniformly. Unlike Othello, there

[2]The locations of 37 additional components were determined by the design engineer to be critical. The engineer selected those locations, and marked them *Fixed* (i.e., not movable) before providing the design to the (human) layout designer. The POOL algorithm, like other PWB component placement programs, considers *Fixed* components to be immovable obstacles.

Table 7.1 Distribution of the features CINDI created for PWB component placement.

	LE	UEQ	AE	Total
Generated	16	8	36	60
Pruned by GFS steps 1 & 2	8	8	16	32
Total	8	0	20	28

was no published material available showing the decisions of an expert layout designer at each step of a PWB component placement solution. Nor was it clear that human choices made sequentially would be useful guidance for controlling a parallel, object-oriented, algorithm in which all components move simultaneously.

A set of state preferences was created by allowing an untrained, but learning, evaluation function to perform the PWB component placement task. Preferences were recorded every time the movement of one component changed the number of design rules that were satisfied. Locations with more design rules satisfied were deemed preferable to locations with fewer preferences. Likewise, locations nearer to satisfying design rules were preferable to locations farther from satisfying design rules. 20,000 such preferences were selected during a run of the POOL algorithm in which 1.5 million preferences were generated. The selection criteria was random, subject to the restriction that they be evenly distributed with respect to the design rules.

The algorithm that generated PWB preferences had several flaws for which no solutions were found. For examples, there were situations in which it would have been better to violate design rules temporarily, to free up space in a particular region of the board. There were also situations in which some design rule violations were more important than others. It was unclear how to eliminate these flaws without creating manually an evaluation function much like the one that was to be learned automatically. Such an evaluation function can be created, but doing so provides little general guidance about how to learn evaluation functions for complex tasks.

7.2 Feature Creation

The CINDI program was applied to the PWB component placement specification contained in Appendix C. The program generated 60 new features, distributed as shown in Table 7.2. The first two steps of the GFS algorithm identified 30 of the features as constant, and two as duplicates. The remaining set of 28 features is representation *C-PWB-P* in the experiments below.

The features created for the PWB component placement task were all easy to interpret, due to the simplicity of the problem specification. For example, the goal statement was initially divided by the *LE* transformation into three new features, each of which corresponded to a design rule to be satisfied. The design rules were further transformed by *LE*, *UEQ* and *AE* transformations, until a set of 60 features were generated. In general, the features were easy to understand. For example, one design rule required that no component overlap (i.e., physically occupy the same space as) another component. The *AE-E* transformation created a corresponding feature that determined how much a component overlapped other components.

Representation *C-PWB-P* was compared to a representation, *DR*, that consisted of Boolean features indicating whether design rules were satisfied. The *DR* representation contained three features, one for each design rule.

It did not make sense to construct a *DR+C-PWB-P* representation for the PWB component placement experiments, because the *C-PWB-P* representation was already a superset of the *DR* representation. CINDI began by breaking the description of the goal state into its component subgoals, which happened to correspond to the design rules. The design rules were then further decomposed and transformed into additional features.

7.3 Accuracy on a Dataset

Experiments were conducted with the Perceptron, RLS, C4.5 and LMDT inductive learning algorithms. In each experiment, evaluation functions were trained on instances selected randomly from the dataset of 20,000 PWB placement preferences, and then tested on a disjoint set of instances selected randomly from the same dataset of 20,000 instances. Training and testing continued until the average accuracy of evaluation functions produced by that learning algorithm, representation, and amount of training data, was known within $\pm 1.0\%$ with 99% confidence.

7.3.1 Perceptron

The Perceptron training rule produces its most accurate classifiers when it is exposed repeatedly to each training instance and the classes are linearly separable. The minimum number of repetitions necessary is $2n$, where n is the number of features in a training instance [Nilsson, 1965]. One disadvantage to the Perceptron training rule is that there is no upper bound on the number of repetitions that might be necessary.

The experiments with the Perceptron rule were conducted with $4n$ repetitions, where n was the number of features in the representation. A few additional experiments were conducted with n ranging from 10 to 100, to ensure that four repetitions were reasonable for this data. The results obtained with $10 \leq n \leq 100$ were less than 1.0% different from results obtained with $n = 4$. Values of n greater than 100 were not tried due to the amount of computation that would have been required.

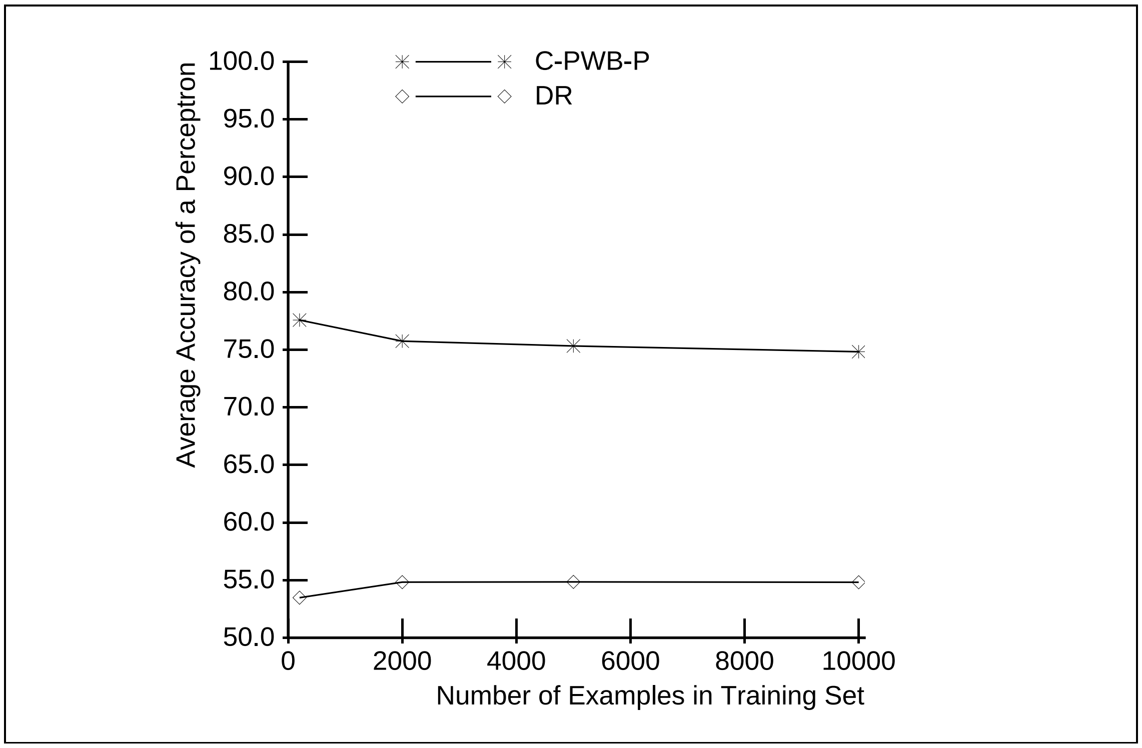

Figure 7.2 Average accuracy of two Perceptron evaluation functions at identifying PWB component placement preferences.

The Perceptron rule consistently created its most accurate evaluation functions with the *C-PWB-P* representation (Figure 7.2). Evaluation functions created with the *DR* representation were more than 20% less accurate, on average. The difference between the two learning curves is statistically significant ($P < 0.01$).

7.3.2 RLS

The RLS algorithm consistently created its most accurate evaluation functions with the *C-PWB-P* representation (Figure 7.3). Evaluation functions created with the *DR* representation were about 15% less accurate, on average. The difference between the two learning curves is statistically significant ($P < 0.01$).

7.3.3 C4.5

The C4.5 algorithm consistently created its most accurate evaluation functions with the *C-PWB-P* representation (Figure 7.4). Evaluation functions created with the *DR* representation were more than 15% less accurate, on average. The difference between the two learning curves is statistically significant ($P < 0.01$).

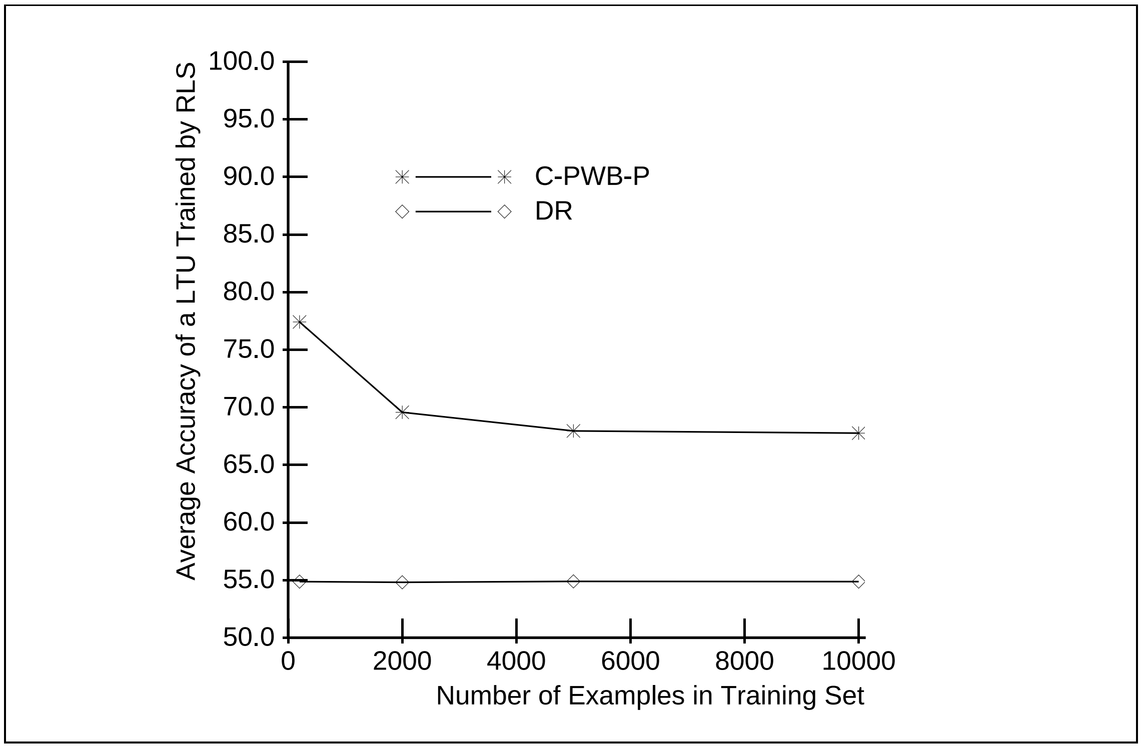

Figure 7.3 Average accuracy of two RLS evaluation functions at identifying PWB component placement preferences.

7.3.4 LMDT

The LMDT algorithm consistently created its most accurate evaluation functions with the *C-PWB-P* representation (Figure 7.5). Evaluation functions created with the *DR* representation were more than 15% less accurate, on average. The difference between the two learning curves is statistically significant ($P < 0.01$).

7.4 Performance at a Task

The effectiveness of the evaluation functions at actually performing PWB component placement was tested on a PWB design from a late 1980s minicomputer. The locations chosen by a highly experienced human layout designer are shown in Figure 7.6. The goal of the experiment was not to reproduce such an orderly placement. Instead, the goal was to find *any* placement of components that satisfied the design rules.

The first ten evaluation functions produced in the experiments above by the Perceptron, RLS, C4.5 and LMDT learning algorithms were tested, for a total of 80

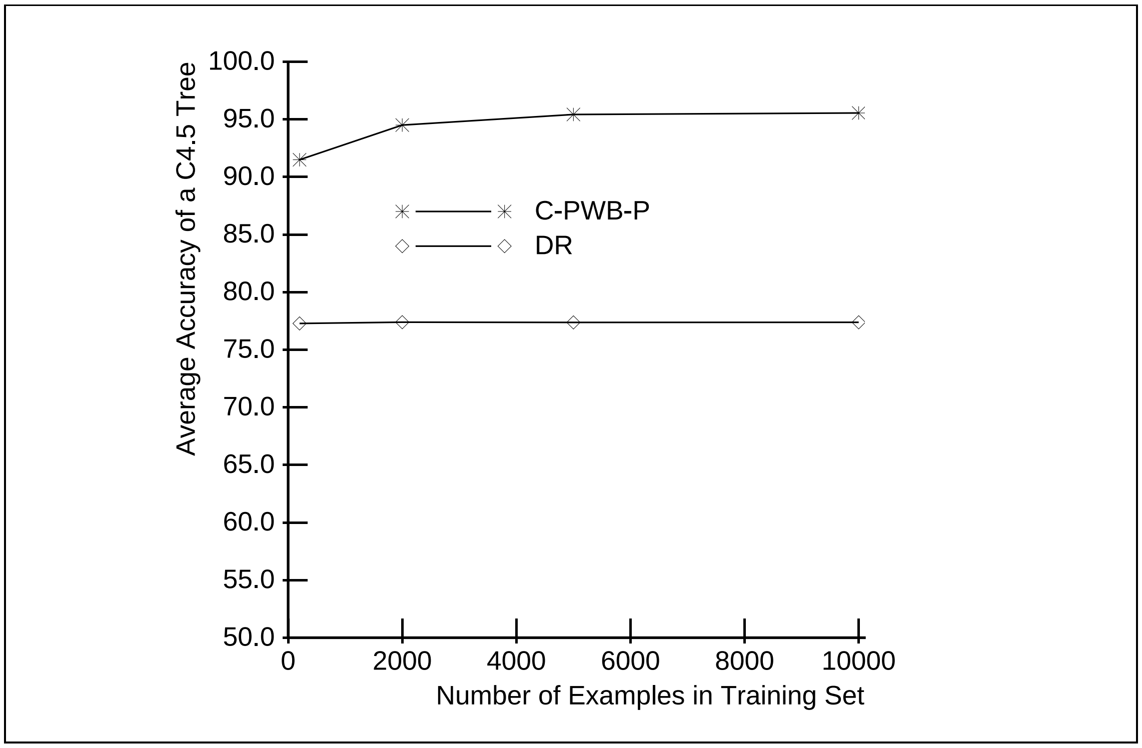

Figure 7.4 Average accuracy of two C4.5 decision trees at identifying PWB component placement preferences.

evaluation functions tested (10 × 4 algorithms × 2 representations). The evaluation functions were inserted into the POOL algorithm, described above, to select from among the locations available to each component on each cycle. Evaluation functions began with an initial placement of components determined by an iterative force-directed placement algorithm (e.g., [Preas & Karger, 1986]). This initial placement left all of the components located near the center of the placement area, overlapping one another and violating the minimum spacing design rule. Each evaluation function was permitted to operate until one of the following conditions occured:

- All design rules were satisfied,
- 20 cycles elapsed during which the average number of design rule violations increased steadily,
- 20 cycles elapsed during which no component changed location, or
- 1,000 cycles elapsed.

A few experiments were conducted during which the number of design rule violations was permitted to increase for more than 20 cycles. Additional cycles did not help.

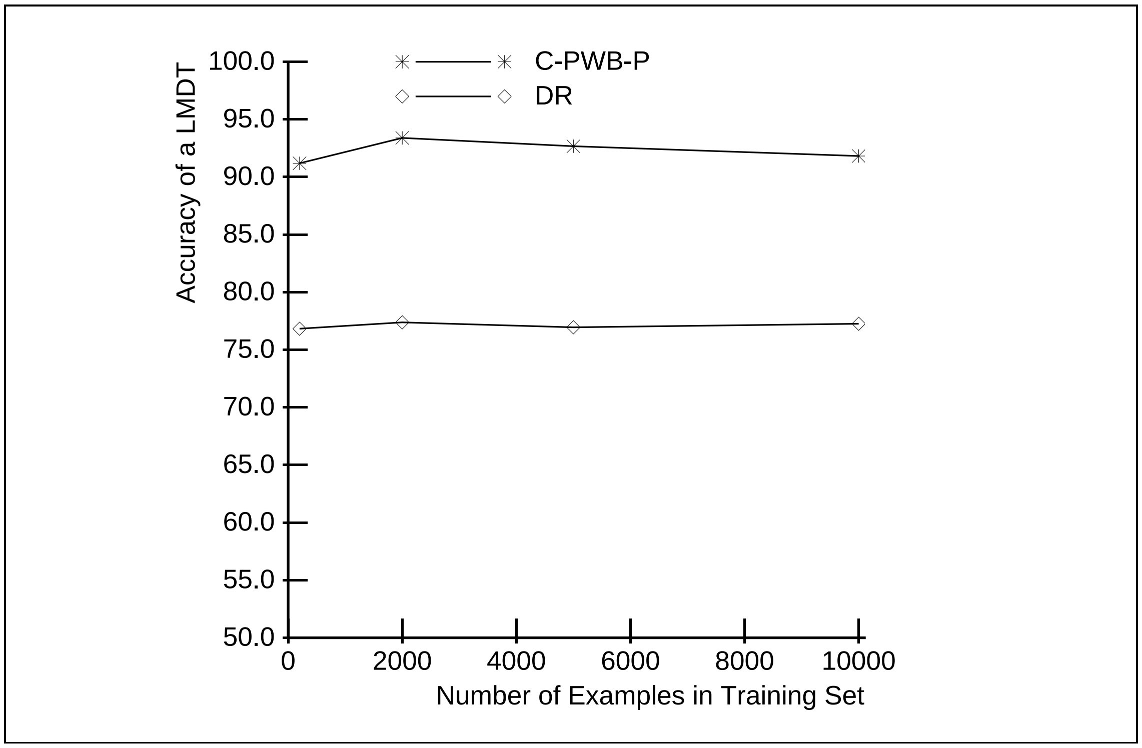

Figure 7.5 Average accuracy of two LMDT evaluation functions at identifying PWB component placement preferences.

A few experiments were permitted to continue beyond 1,000 cycles, to ensure that the number 1,000 was not too low. In every experiment, the evaluation function had made whatever progress it was going to make by 500 or 600 cycles. Only very minor improvement was ever observed after the 600'th cycle.

Only one of the 80 evaluation functions was able to find a placement of components that satisfied the design rules (Figure 7.7). The successful evaluation function was created by the Perceptron learning rule. Several other evaluation functions created by the Perceptron and C4.5 learning algorithms came quite close to succeeding, but did not. Possible reasons for this result are discussed below.

How does one measure the effectiveness of evaluation functions that were unable to complete the task? One approach is to measure the percentage of components that comply with design rules when the placement algorithm stops. The advantage of this metric is that it measures the work accomplished by the evaluation function. The disadvantage is that it ignores the cost of the work remaining. A design with a small number of badly misplaced components could conceivably be harder to complete than one with a large number of slightly misplaced components. Visual inspection of dozens of designs suggested that, while conceivable, this problem did not plague the designs

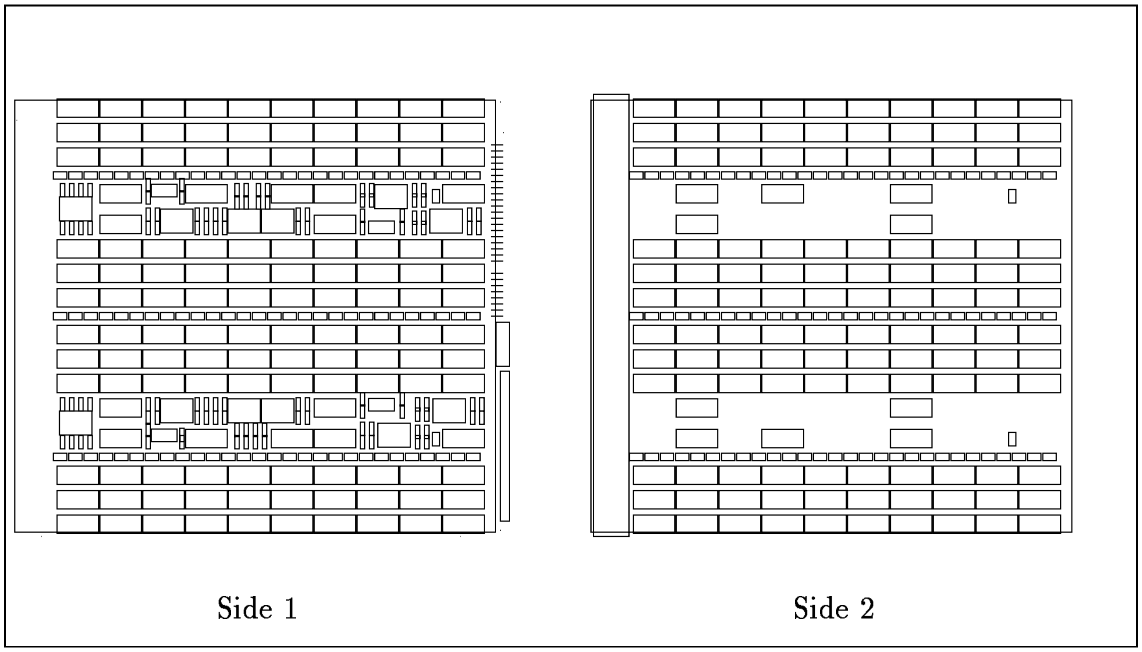

Figure 7.6 The component locations chosen by an experienced human layout designer for the PWB component placement task. Components are placed on both sides of the printed wiring board.

created by the evaluation functions. Incomplete designs with high compliance usually had small areas of congestion that prevented some components from being placed properly. Incomplete designs with low compliance usually had many overlapping components or many components not placed within the board outlines.

A more appropriate metric would have required domain knowledge beyond what was available during the experiment.

The C4.5 algorithm produced the most effective evaluation functions for controlling the POOL algorithm. C4.5, LMDT and the Perceptron all produced better results with the C-PWB-P representation than with the DR representation. As expected, the LMDT algorithm appeared the least sensitive to the representation.

The RLS algorithm differed from the other algorithms in that it produced its most effective evaluation functions with the DR representation. However, this result should be viewed with caution. None of the evaluation functions produced by the RLS algorithm were very effective. The advantage of the DR representation over the C-PWB-P representation was due to the differing characteristics they evoked in the evaluation functions. Evaluation functions trained with the C-PWB-P representation often did not move the components at all. Evaluation functions trained with the DR representation often moved components wildly. The wild movement of components yielded an initial improvement as component overlaps were eliminated, but was

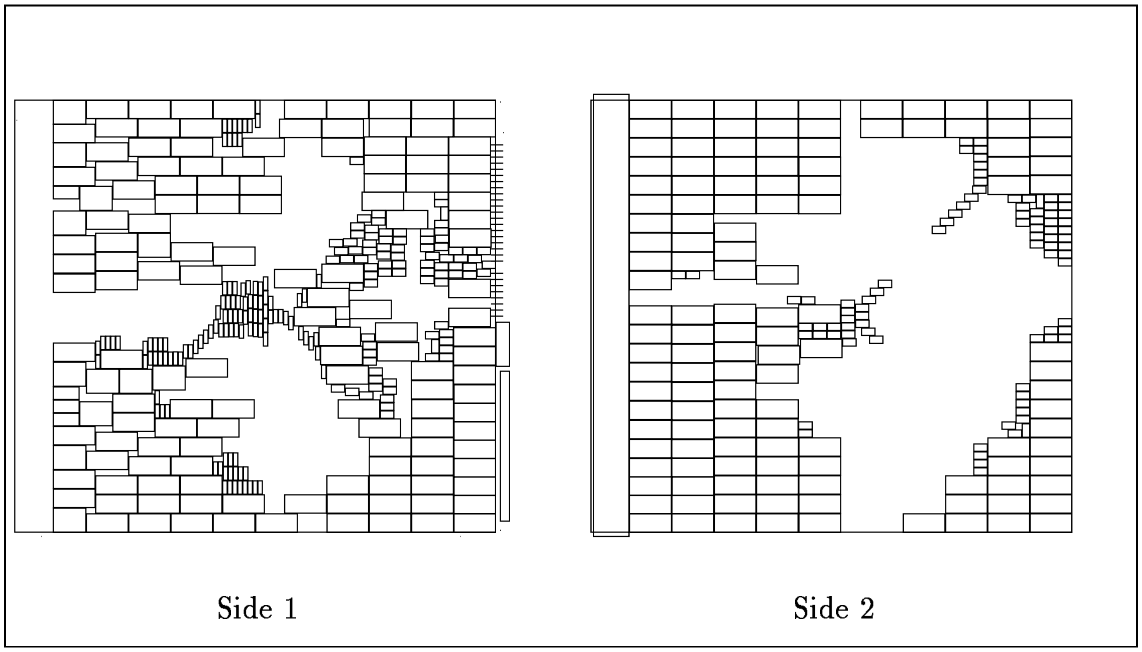

Figure 7.7 The component locations chosen by an evaluation function learned for the PWB component placement task. Components are placed on both sides of the printed wiring board.

followed by deterioration as components sailed off of the board or back over other components. This characteristic reveals itself in Table 7.4 by average iterations below 100.

7.5 Summary

One can rank representations according to the accuracy of the evaluation functions that they enable with different learning algorithms. According to this metric, the *C-PWB-P* representation created automatically by CINDI is the most useful. The structural *DR* representation was least useful. All four of the dataset accuracy experiments, and three of the four performance accuracy experiments, yielded the same ranking of representations.

Evaluation functions in the *C-PWB-P* representation were between 13% and 20% more accurate on dataset accuracy experiments than evaluation functions in the *DR* representation. The performance accuracy experiments varied more widely, as shown in Table 7.4.

Table 7.2 Average effectiveness at PWB component layout of evaluation functions created by four different learning algorithms.

	Average Compliance			Average
	DR 1	DR 2	DR 3	Iterations
C4.5, using C-PWB-P	100.00%	63.35%	58.30%	1000.0
C4.5, using DR	98.44%	3.00%	2.84%	79.7
Perceptron, using C-PWB-P	99.90%	48.11%	42.03%	643.89
Perceptron, using DR	99.67%	10.77%	10.69%	66.56
LMDT, using C-PWB-P	99.52%	44.59%	41.63%	480.40
LMDT, using DR	99.61%	30.22%	29.69%	722.50
RLS, using C-PWB-P	100.00%	1.55%	1.47%	220.33
RLS, using DR	99.59%	9.52%	9.41%	58.00

One question raised by the Perceptron, RLS, and LMDT dataset accuracy experiments is why accuracy *decreased* as a function of the number of examples in the training set. The causes for the LTU algorithms (Perceptron and RLS) and LMDT are probably different.

For a population of 20,000 examples, 3,444 examples are required to obtain results that are representative within $\pm 1.0\%$, with 99% confidence. Smaller training sets might be considered undesirable because they are not sufficiently representative. However, it may be that by masking complexity, small training sets allowed the algorithms to find hyperplanes that were reasonably good approximations. As the full complexity of the domain became apparent, the LTU algorithms may have been overwhelmed. This possibility is supported by statistics gathered during the training of Perceptrons. As the training set size was increased, the Perceptron weight vectors were more likely to cycle, indicating that the Perceptron could not find a hyperplane that separated the classes. No such evidence was available for RLS, so it is not known whether the RLS algorithm could have been affected similarly.

The inverse relationship of number of training instances to concept accuracy is less surprising for LMDT, because LMDT uses an annealing rate that ignores the number of instances in the training set. Larger training sets give LMDT fewer opportunities to consider each training instance.

A second question, raised by the performance accuracy experiments, is why the Perceptron and LMDT algorithms showed one behavior in the dataset accuracy experiments and another behavior in the performance accuracy experiments. The accuracy of LMDT in the dataset accuracy experiments was not apparent in the results of the performance accuracy experiments. The Perceptron behaved oppositely, doing better in the performance accuracy experiments than one might have expected from the datast accuracy experiments.

The PWB component placement preference predicates were trained on preferences generated automatically by a program with no special knowledge of PWB component placement. Preferences were generated whenever a random movement caused a change in the number of design rules violated. There were situations in which a preference would be generated among locations that appeared identical under all representations, or in which the preferred location was actually less desirable in spite of having more design rules satisfied. Both of these situations made the data inconsistent, or 'noisy'. The distribution of training and testing instances also intentionally did not match the distribution of instances encountered during problem-solving, in order to prevent the program from satisfying one class of design rules at the expense of another class. The 'noisy' nature of the training and testing data may have affected some of the algorithms more than the others, which would explain why some behaved as expected and others did not.

CHAPTER 8
EXPERIMENTS WITH OTHELLO

Othello[1] is a board game that is played between two players on an 8 x 8 board with discs that are black on one side and white on the other. Black makes the first move on a board configured as shown in Figure 8.1a. A disc can only be placed in a square if the square is unoccupied and adjacent to a string of the opponents discs terminated by one of the player's own discs. For example, in Figure 8.1a, black can play into squares d3, c4, f5, or e6. When a disc is played, every adjacent string of opponent's discs is turned over, so that it becomes owned by the moving player. Figure 8.1b shows the board after black has moved into square e6. Players alternate placing discs on the board. If a player has no move, that turn is forfeited. Play continues until neither player can move. The player with more discs at the end of the game wins.

Although the rules of Othello are simple, the game is not easy to master. The search space for Othello has been estimated at 10^{60} boards [Frey, 1986], which places it between Checkers (10^{40} boards [Samuel, 1959]) and Chess (10^{120} boards [Shannon, 1950]) in complexity. Exhausitive search is only possible during the end of the game. Othello computer programs currently begin exhaustive search at about move 45 (15 moves left in the game) [Rosenbloom, 1982; Frey, 1986; Levy & Beal, 1989; Kierulf, 1989].

Othello offers three advantages as a domain in which to test knowledge-based feature generation. First, it has a large and complex search space. Second, there is a large body of published material that analyzes Othello in great detail. This published material provides both a source of training data and a reference point against which to judge features. Finally, Othello is an example of a domain in which humans routinely handcraft both features and heuristic evaluation functions for use in computer programs.

[1]Othello® is a registered trademark of Tsukuda Original, licensed by Anjar Co., © 1973, 1990, Pressman Toy. All rights reserved.

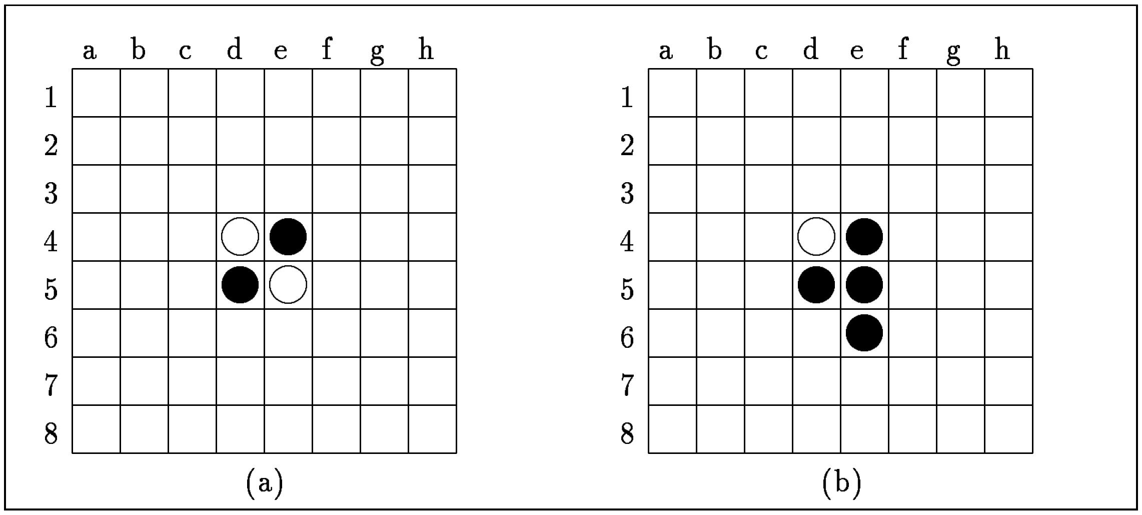

Figure 8.1 (a) The initial configuration of Othello pieces, and (b) the configuration that results after black places a piece in square 36.

8.1 Training Data

Twenty eight games between expert players were selected from published sources. Each game satisfied one of the following criteria:

- It was played at an OTHELLO World Championship tournament,
- It was played between regional, national or international champions,
- It involved players rated 'Advanced' or better by the US Othello Association, or
- It involved computer programs thought to be highly competent.

These criteria were intended to ensure relatively accurate training data. The data are not completely accurate because computer analysis of some games revealed small mistakes during the endgame play of humans [Johnson, 1992; Kling, 1992].

Nineteen games were played between humans, five were played between computers, and four were played between a human and a computer. The number of games used was determined by the availability of published games and computational resources. The intent was to use as many examples as possible, in order to ensure accurate learning. The games and their sources are presented in Appendix D.

The twenty eight Othello games, hereafter referred to as 'book' games, provided information about the choices made by experts. It is reasonable to surmise that the moves chosen by experts are preferred to the alternative moves. These preferences were represented as delta vectors (Section 4.3.3). Preferences that occurred during

Table 8.1 Distribution of the features CINDI created for Othello.

	LE	UEQ	AE	Total
Generated	26	20	4	50
Pruned by GFS steps 1 & 2	20	2	1	23
Total	6	18	3	27

the last 14 moves of the game were discarded, because it is common for computer programs to use perfect search during endgame play. The resulting dataset contained just over 10,000 preference pairs, but was truncated to 10,000 preference pairs for convenience.

8.2 Feature Creation

A Prolog specification of the Othello goal state and move operator is available from the Machine Learning database at the Irvine campus of the University of California. This specification was transformed manually from Prolog into a First Order Predicate Calculus specification suitable as input to the CINDI feature generation program. The resulting specification is provided in Appendix D.

The CINDI program created 50 features from the specification of the Othello goal and operator, distributed as shown in Table 8.2. The first two steps of the GFS algorithm identified 20 of the features as constant and three as duplicates. The remaining set of 27 features is representation *C-OTH-P* in the experiments below.

Most of the features created by CINDI are variations of well-known concepts in Othello. For example, the CINDI program generates features that count the number of available moves (*mobility*), the difference in the number of squares occupied by the two players (*disc differential*), and the inability of a player to make a move (*a wipeout*).

Two representations from locally available Othello programs[2] were selected for comparison with the *C-OTH-P* representation. The *SQ2* representation consisted of two Boolean features per square on an Othello board. One feature indicated whether the square was occupied by black, the other indicated whether the square was occupied by white. The *WYST* representation consisted of six features measuring the mobility, stability and desirability of a position. The features were created and

[2]The locally available Othello programs were developed by Paul Utgoff, Jeff Clouse, and a student in a graduate course on game-playing programs (personal communications).

refined manually over a period of one and a half years, and were estimated by their author to have required about four person-months of effort.[3]

The *SQ2* and *WYST* representations were chosen for this experiment because they occupy different ends of the representational spectrum. The *SQ2* representation records state information compactly for efficient manipulation by a search algorithm. It is the type of representation upon which knowledge-based feature generation tries to improve. In contrast, the *WYST* representation attempts to make explicit the information necessary to make high quality decisions. It is the type of representation that knowledge-based feature generation tries to create automatically.

A fifth representation, called *SQ2+C-OTH-P*, was created automatically by combining the features from the *SQ2* and *C-OTH-P* representations.

8.3 Accuracy on a Dataset

The dataset of 10,000 delta vectors representing the Othello preferences of experts was randomly sampled to generate independent training and testing sets. The sizes of training sets ranged from 100 examples (1% of the dataset) to 5,000 examples (50% of the dataset). The size of the testing set was fixed at 5,000 examples, so that classification accuracy would be comparable among training sets of different sizes. Training and testing sets were disjoint.

The object of the experiment was to determine the average accuracy of the evaluation functions that would be created with different representations and different amounts of training data. The average accuracy for a particular representation and amount of training data was determined by repeatedly creating, training and then testing evaluation functions, until either 30 evaluation functions had been tested or average accuracy was determined $\pm 1.0\%$, with 99% confidence, whichever occurred later.

The size of the *SQ2* and *SQ2+C-OTH-P* representations (128 and 155 features, respectively) posed a problem for experimentation. The computational complexities of the Perceptron, RLS, C4.5 and LMDT algorithms all depend in part upon the number of features in the representation. The computational complexity of experimenting with all of the algorithms exceeded the time and computational resources available. Therefore, experiments were conducted with just two algorithms: RLS and LMDT.

The RLS algorithm was chosen to represent linear concept learners because its computational complexity, $O(n^2)$ for each instance, was lower than that of the Perceptron's $\omega(n^2)$ for each instance ($O(n)$ per exposure, but 2n exposures are a minimum). The LMDT algorithm was chosen to represent non-linear concept learners because it

[3]Jeff Clouse, personal communication.

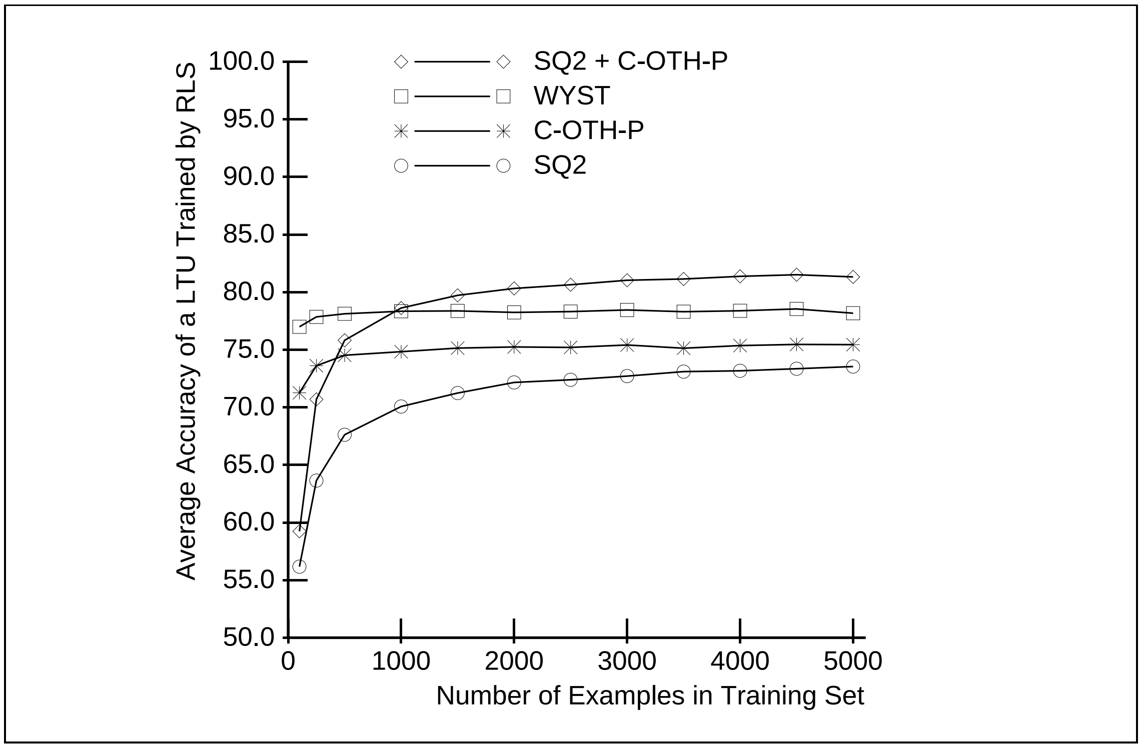

Figure 8.2 Average accuracy of four RLS evaluation functions at identifying the preference's of Othello experts.

was easier than C4.5 to incorporate into the performance accuracy experiments.[4] It was also assumed initially that LMDT would produce more accurate concepts than C4.5, due to the differing power of their concept description languages. Experiments with the stacking blocks, tic-tac-toe, and PWB component layout problems, which occurred later, suggested that this assumption might be incorrect.

The reduction in the number of algorithms tested was considered acceptable because the results from the stacking blocks, tic-tac-toe and PWB component placement experiments were all relatively stable. All of the learning algorithms generally agreed about which representations were best and worst. The degree of improvement obtained by adding knowledge-based features to structural features was usually very similar among different learning algorithms.

8.3.1 RLS

[4]The version of C4.5 used in the experiments did not support simultaneous use of two or more decision trees for classification. This limitation would have made it difficult to conduct Othello tournaments between pairs of evaluation functions.

The RLS algorithm consistently created its most accurate evaluation functions with *SQ2+C-OTH-P*, a representation that required no manual effort to create (Figure 8.2). *SQ2+C-OTH-P* is a combination of the *SQ2* structural representation used in some Othello programs, and the *C-OTH-P* representation created by CINDI. The *WYST* representation, which required four person-months of manual effort to create, yielded evaluation functions that were about 3% less accurate. The *C-OTH-P* representation, which was created automatically, yielded evaluation functions about 2% less accurate than *WYST* evaluation functions and 5% less accurate than *SQ2+C-OTH-P* evaluation functions. The structural *SQ2* representation was worst in this experiment, yielding evaluation functions 8% less accurate than the *SQ2+C-OTH-P* evaluation functions. The differences between the learning curves are statistically significant ($P < 0.01$) for training sets larger than 1500 examples.

The number of examples necessary to reach asymptotic accuracy increased with the number of features in the representation. It is not surprising that more examples are required when there are more degrees of freedom, but it is cause for concern. In this experiment, the most useful representation contained 155 features. The literature on Othello suggests that a subset of the 155 features would produce substantially similar performance.

8.3.2 LMDT

Experiments with the LMDT algorithm were similar to the experiments with the RLS algorithm, with one exception. The LMDT algorithm avoids overfitting the training data by performing a 'pruning' phrase after the decision tree is built. The pruning phase requires a set of instances that is disjoint from the set of training data. In the experiments with LMDT, the pruning data was always 20% of the training data available to LMDT.[5]

The stability of the results with RLS, coupled with the computational complexity of LMDT, caused the minimum number of evaluation functions created to be reduced from 30 to ten. However, all results reported below are accurate ± 1.0, with 99% accuracy.

The LMDT algorithm consistently created its most accurate evaluation functions with *WYST*, a representation that required four person-months to create. The *SW2+C-OTH-P* representation, which required no manual effort to create, yielded evaluation functions that were about 1.5% less accurate. The *C-OTH-P* representation, which was created automatically, yielded evaluation functions about 2.5% less accurate than *SQ2+C-OTH-P* evaluation functions and 4% less accurate than *WYST* evaluation functions. The structural *SQ2* representation was worst in this experiment, yielding evaluation functions 8% less accurate than the *WYST* evaluation functions. The differences between the learning curves are statistically significant ($P < 0.01$) for training sets larger than 1500 examples.

[5]The figure 20% was suggested by Carla Brodley, coauthor of LMDT (personal communication).

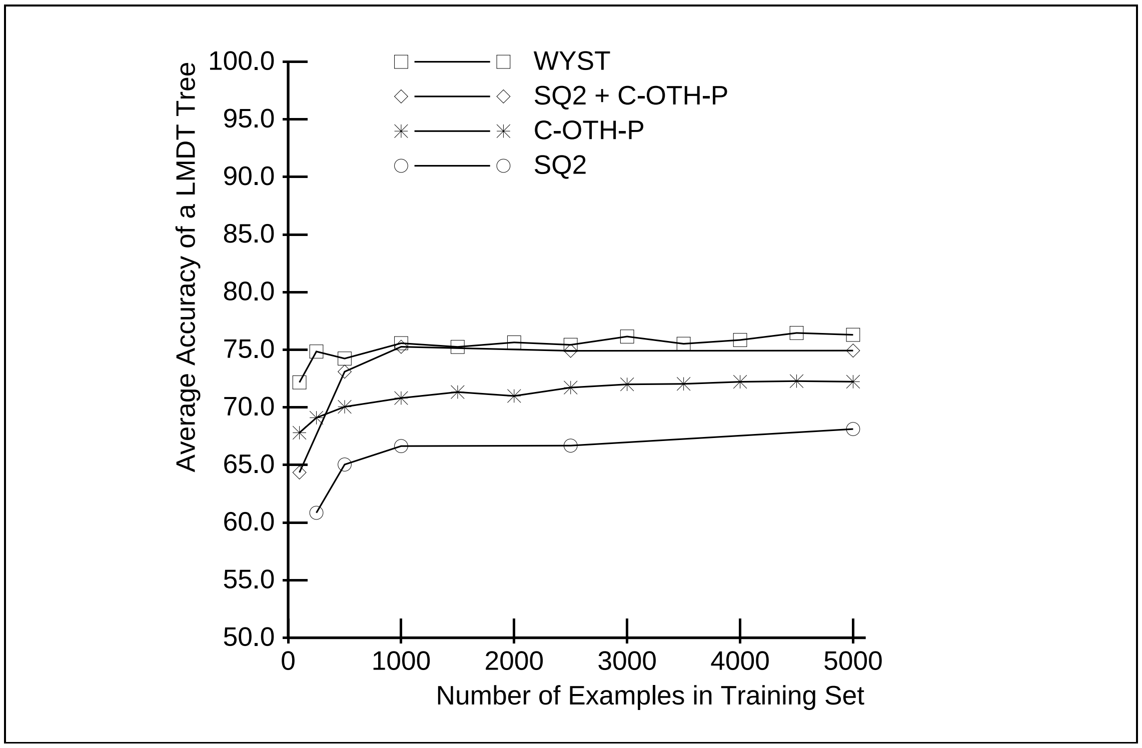

Figure 8.3 Average accuracy of four LMDT evaluation functions at identifying the preference's of Othello experts.

As was the case with the RLS algorithm, the number of examples necessary to reach asymptotic accuracy increased with the number of features in the representation.

8.4 Performance at a Task

The experiment was conducted by selecting from each RLS training cycle an "average" linear threshold unit. Selection was done automatically by a procedure that saved the LTU whose classification accuracy was closest to the average accuracy for the particular training set size. The "contestants" were 48 LTUs (4 representations $\times$ 12 training set sizes). Each LTU competed with other LTUs whose training sets were of the same size.

Each pair of contestants played 10 games. Contestants alternated who went first. The first ten moves were made randomly, as suggested by [Lee & Mahajan, 1988], so that the contestants could be evaluated on more than just two games. The last fourteen moves of each game were made by perfect search, as is common with Othello

Table 8.2 Results of Othello tournaments between linear threshold units trained with four different representations.

Opponents	Winner Name	Winner Games	Loser Name	Loser Games
SQ2+C-OTH-P vs *C-OTH-P*	*SQ2+C-OTH-P*	83	*C-OTH-P*	34
SQ2+C-OTH-P vs *WYST*	*SQ2+C-OTH-P*	74	*WYST*	44
SQ2+C-OTH-P vs *SQ2*	*SQ2+C-OTH-P*	65	*SQ2*	53
SQ2 vs *C-OTH-P*	*SQ2*	67	*C-OTH-P*	50
SQ2 vs *WYST*	*SQ2*	62	*WYST*	57
WYST vs *C-OTH-P*	*WYST*	61	*C-OTH-P*	58

programs. During the middle of the game, contestants took turns placing pieces on the board. An LTU "selected" a move by making choices among pairs of possible moves. Each pair of possible moves (m_i, m_j) was encoded as a delta vector and presented to the LTU for classification. If the LTU returned a positive value, m_i was considered preferred to m_j. If the LTU returned a negative value, m_j was considered preferred to m_i. The move preferred over all other moves was deemed selected by the LTU. When an LTU exhibited no preference among moves (a rare situation), a decision was made randomly.

Table 8.4 summarizes the results of the Othello tournaments between linear threshold units trained with four different representations. Each row is based upon 120 games (12 pairs of LTUs competing in 10 games each). Tie games are not shown

The utility of a representation can be measured by the number of games won by evaluation functions using the representation. According to this metric, the *SQ2+C-OTH-P* representation, which required no manual effort to create, was the most effective. The *SQ2* structural representation was second most effective. The *WYST* representation, which required four person-months of effort to create, was third. The *C-OTH-P* representation, which was created automatically, was least effective in these experiments.

The relative effectiveness of the *SQ2+C-OTH-P*, *WYST*, and *C-OTH-P* representations match the results from the RLS performance accuracy experiments (Section 8.3.1) that created the evaluation functions. However, the difference between the *SQ2+C-OTH-P* and *WYST* representations was small in the performance accuracy experiments, while it is large here. A second difference is the effectiveness of the *SQ2* representation in these performance accuracy experiments. In the dataset accuracy experiments, the *SQ2* representation was the least effective.

Table 8.3 How often evaluation functions in each representation selected the Othello move preferred by an expert Othello player.

Representation	Average Accuracy
WYST	35.8%
SQ2+C-OTH-P	35.5%
C-OTH-P	33.8%
SQ2	27.6%
(random selection)	16.0%

8.5 Performance in Book Games

The Othello dataset accuracy and performance accuracy experiments agreed that the *SQ2+C-OTH-P* representation was most useful for learning evaluation functions. However, the dataset accuracy and performance accuracy experiments agreed on little else. Therefore, a third experiment was conducted, to measure each representation by how often its evaluation functions were able to identify each move preferred in the set of book games described above. The evaluation functions tested were those used in the previous experiments. Moves were selected by each evaluation function as described above. Statistics were also gathered for a random selection strategy, to give further perspective on the results.

Table 8.5 reports the results. The value reported for random selection is the average of results from 28 book games. The remaining values are each the average of results from 336 games (12 LTUs per representation, 28 book games).

Random move selection yielded the correct move 16% of the time, which is consistent with results reported by Rosenbloom (1982). Evaluation functions based on the *SQ2* features selected the correct move 27.6% of the time. Addition of the *C-OTH-P* features to the *SQ2* features improved the accuracy of move selection to 35.5%. The WYST representation, which was created manually, fared only slightly better.

Accuracy was also measured in ten move increments, to provide a glimpse of how useful each representation is at different stages of an Othello game. Evaluation functions produced for the *WYST* and *SQ2+C-OTH-P* representations were both the most accurate and the most consistent. The *WYST* evaluation functions were about 2% more accurate than the *SQ2+C-OTH-P* evaluation functions during the opening moves of the game. The situation was reversed during the late-middle and early endgame portions of the game, when the *SQ2+C-OTH-P* evaluation functions were 1%-2% more accurate. Othello experts view the middle and endgame portions of the game as more important than the opening moves.

Table 8.4 How often evaluation functions in each representation selected the Othello move preferred by an expert Othello player, by stage of game.

	Average Accuracy					
	Moves	Moves	Moves	Moves	Moves	Moves
Representation	1-10	11-20	21-30	31-40	41-50	51-60
WYST	39.2%	37.5%	34.9%	30.2%	32.9%	42.5%
SQ2 + C-OTH-P	37.7%	35.3%	34.8%	32.3%	33.7%	40.5%
C-OTH-P	39.3%	35.2%	33.1%	25.9%	30.8%	40.0%
SQ2	31.8%	21.6%	24.3%	25.4%	29.4%	35.9%
(random selection)	20.0%	10.0%	11.0%	11.0%	15.0%	32.0%

The *C-OTH-P* representation produced evaluation functions that were initially the most accurate. However, as the game progressed, the accuracy of the *C-OTH-P* evaluation functions deteriorated. The improvement in accuracy near the end of the game would be irrelevant to most Othello programs, because perfect search is commonly employed at about move 45.

The *SQ2* representation yielded evaluation functions that were the least accurate at every stage of the game. These results are consistent with the results from the dataset accuracy experiment. Unfortunately, these results shed little light on why the *SQ2* evaluation functions performed so well in the performance accuracy experiments.

8.6 Summary

The results of the three experiments are not as easy to intepret because there were significant differences in the results of the various experiments. This summary focuses first on what the experiments had in common, then discusses those issues on which they disagreed.

The *SQ2+C-OTH-P* representation, which required no manual effort to create, performed well in all of the experiments. The evaluation functions created with the *SQ2+C-OTH-P* representation were the most effective in the RLS dataset accuracy experiment and the performance accuracy experiment. The *WYST* representation, which required four months of manual effort to create, was slightly more effective in the other dataset accuracy experiment and in comparison to the book games.

The experiments differed greatly in how useful they found the *SQ2* structural representation. The dataset accuracy experiments and the comparison to book games all suggest that the *SQ2* representation was the least effective of the representations tested. However, the *SQ2* reprensentation yielded evaluation functions that did well in the performance accuracy experiments. Why did evaluation functions based on the structural representation play Othello so well?

Examination of the LTU weights revealed the strategies supported by the various representations. The structural features led to a strategy of seizing the edges of the board early in the game, while the functional features led to a strategy of maintaining mobility.

The strategies supported by the different representations explain the results in the dataset accuracy experment and the comparison to book games. Human Othello players believe that structural representations of Othello are effective for naive, or beginner, strategies, but insufficient for expert-level performance [Rosenbloom, 1982; Mitchell, 1984]. Games between highly skilled players are often decided by mobility or the lack thereof. Therefore it is not surprising that representations that support reasoning about mobility are better than a structural representation at predicting the choices made by expert players.

It remains unclear why the structural representation yielded such good performance in the performance accuracy experiments. The simple strategy of seizing edge squares and avoiding X and C squares outperformed the more complex strategies enabled by the *WYST* and *C-OTH-P* representations. It may be that the training data, based on the preferences of Othello experts, were not representative of the game positions that occur between players of varying skill. Mitchell (1984) has observed that it makes sense to employ different evaluation functions for players of different levels of expertise so that this problem can be avoided.

It has been argued that representative data are not a problem if the learning algorithm trains during problem-solving, as with Temporal Differences (TD) learning [Sutton, 1988]. A Temporal Differences experiment was conducted as part of the thesis research. The evaluation functions alternated winning streaks, but none could gain a clear superiority. There was little variation in the moves selected from one game to the next; the same games were often played repeatedly. The reason for this behavior is unknown. It is possible that the evaluation functions were becoming expert in narrow regions of the Othello search space. Whenever dislodged from such a region, perhaps by an opponent's unexpected move, the evaluation function appeared to begin forgetting about the old region while learning about the new region. Jacobs (1990) calls this type of phenomenon *temporal crosstalk*.

A possible solution to temporal crosstalk is to augment the TD learner with a small memory that stores a sample of preferences encountered. The stored preferences could be used periodically to remind the TD learner about portions of the search space that it has not visited recently. This architecture is similar to the CABOT system [Callan, Fawcett & Rissland, 1991], because it combines both a memory of past experiences and the ability to learn. However, CABOT learned to use its memory more effectively, whereas the architecture proposed here would use its memory to learn more effectively.

CHAPTER 9
DISCUSSION OF RESULTS

Two types of experiments were conducted. The *dataset accuracy* experiments tested the effectiveness of each representation by training a classifier with a set of examples and then testing the classifier on a disjoint set of examples drawn from the same population. The *performance accuracy* experiments tested the effectiveness of each representation by comparing the ability of classifiers at actually performing the given task. Experiments were conducted with four inductive learning algorithms (Perceptron, RLS, C4.5, LMDT) and four tasks (stacking blocks, tic-tac-toe, Othello, PWB component layout).

Eighteen sets of dataset accuracy experiments were conducted (Table 9). In each case representations that included knowledge-based features produced more accurate classifiers than representations that did not. Differences ranged from 1% to 20%. The difference in accuracy produced by knowledge-based features was statistically significant in all but one case (the stacking blocks task, the B1-B16 representation, and the RLS algorithm).

Five sets of performance accuracy experiments were conducted (Table 9). In four of the five experiments, representations that included knowledge-based features produced better performance than representations that did not include knowledge-based features.

The dataset accuracy experiments demonstrated that knowledge-based feature generation produces features that reliably improve the accuracy of the learned concept. In 16 out of the 18 dataset accuracy experiments, the addition of knowledge-based features to a structural representation improved average concept accuracy at least 7%. In some experiments the improvement in average concept accuracy was nearly 20%. In all but one experiment, the improvement was statistically significant.

The performance accuracy experiments were intended to verify that an improvement in the ability to select among search states causes an improvement in the ability to solve problems. The performance accuracy experiments demonstrated that knowledge-based feature generation produces features that usually improve the ability of the learned concept to solve problems. In four out the five performance accuracy experiments the addition of knowledge-based features to a structural representation produced an average improvement in problem-solving abilities.

Table 9.1 Summary of results of eighteen dataset accuracy experiments.

Domain	Algorithm	Average Change
Stacking Blocks	Perceptron (B1-B16)	+ 9%
Stacking Blocks	Perceptron (L1-L4)	+ 20%
Stacking Blocks	RLS (B1-B16)	+ 1%
Stacking Blocks	RLS (L1-L4)	+ 11%
Stacking Blocks	C4.5 (B1-B16)	+ 4%
Stacking Blocks	C4.5 (L1-L4)	+ 10%
Stacking Blocks	LMDT (B1-B16)	+ 7%
Stacking Blocks	LMDT (L1-L4)	+ 15%
Tic-Tac-Toe	Perceptron	+ 11%
Tic-Tac-Toe	RLS	+ 11%
Tic-Tac-Toe	C4.5	+ 12%
Tic-Tac-Toe	LMDT	+ 13%
PWB Placement	Perceptron	+ 20%
PWB Placement	RLS	+ 13%
PWB Placement	C4.5	+ 18%
PWB Placement	LMDT	+ 15%
Othello	RLS	+ 8%
Othello	LMDT	+ 7%

The experimental results demonstrate that knowledge-based feature generation can create functional features useful for inductive learning. The results also confirm that the features can be created in linear time, using only a search problem specification and a set of heuristics.

In one of the eighteen dataset accuracy experiments and one of the five performance accuracy experiments the addition of knowledge-based features to a structual representation made little improvement. This result is not surprising, because there are demonstrably good features that knowledge-based feature generation does not create. The results suggest that knowledge-based feature generation should be one of several feature creation methods available for improving a representation, instead of being the only method.

Table 9.2 Summary of results of five performance accuracy experiments.

Domain	Algorithm	Effect of New Features
Othello	RLS	More games won
PWB Placement	Perceptron	Higher compliance with design rules
PWB Placement	RLS	Lower compliance with design rules
PWB Placement	C4.5	Higher compliance with design rules
PWB Placement	LMDT	Higher compliance with design rules

9.1 When is Knowledge-Based Feature Generation Effective?

Knowledge-based feature generation works best on problems that are serially decomposable, and that are described by quantification and arithmetic inequalities. If the problem is serially decomposable, the *LE* transformation is likely to produce features that indicate whether the individual subgoals are satisfied in a particular state. If the problem contains quantification and arithmetic inequalities, the *UEQ* and *AE* transformations may be able to create features that indicate progress in problem-solving even when no subgoal becomes satisfied or violated.

It is possible for knowledge-based feature generation to create features for problems that are neither serially decomposable nor described by quantification or arithmetic inequalities, but the resulting Boolean features are less able to describe progress in problem-solving at a given state. For example, the Eight Puzzle can be described as a conjunction of eight subgoals, each indicating the desired position of one tile. Eight knowledge-based features would be produced from such a specification, one for each subgoal. The resulting representation would indicate progress each time a tile was moved into or out of its desired location. However, other movements would cause no change in any of the eight features.

Knowledge-based feature generation also seems most effective when the scope of the *UEQ* transformation is limited. The *UEQ* transformation normally applies to quantified statements. It transforms a statement by replacing the outermost adjacent quantifiers with summation and division operators. The result is a feature that calculates the percentage of variable bindings that satisfy the Boolean expression. Experience suggests that it is perhaps better to replace only the outer quantifier with summation and division operators. The difference can be seen in the features created for the statement "Every man loves a woman" ($\forall m \exists w, \text{loves}(m, w)$). The *UEQ* transformation produces a feature that determines the probability that a randomly selected man loves a randomly selected woman. If only the outer quantifier were replaced with summation and division operators, the feature would calculate the percentage of men that love a woman.

In its current form, the degree to which quantifiers are expanded can be controlled by the way in which the problem is specified. Portions of the problem statement may be most naturally expressed as predicates, which are themselves defined in terms of other predicates. The problem specification for Othello, presented in Appendix D, is similar to a computer program that is defined by nested calls to subroutines. The *UEQ* transformation does not expand automatically the predicates it encounters in a statement, so a predicate definition effectively limits the scope of the transformation. If the statement "Every man loves a woman" were represented as $\forall m$, loves-a-woman(m), the resulting feature would indicate the percentage of men that love a woman.

There are advantages and disadvantages to the way that the *UEQ* transformation handles quantifiers. One advantage is that it does not matter in what order quantifiers are specified. Another advantage is that predicates, which are assumed to be motivated by domain considerations, limit the scope of *UEQ* quantifier transformations. The disadvantage is that there will be cases in which the *UEQ* transformation does not create the most desirable feature.

It is straightforward to generate both types of features, and leave the choice between them to the learning algorithm. The cost of creating the additional features is low. However, this strategy must be viewed with caution, because the complexity of some learning algorithms (e.g., RLS) depends upon the number of features in the representation.

9.2 Feature Characteristics

Post-mortem analysis of the knowledge-based features revealed two recurring types of inefficient behavior. Both types of behavior were caused by particular ways of expressing domain knowledge, and neither type could be identified easily with simple syntactic rules. The first type of inefficient behavior involved using generate-and-test to identify elements of a relation. The second type of inefficient behavior involved creating features out of statements that, by definition, are constant. The problems, and their possible solutions, are outlined below.

9.2.1 Generate-and-Test Features

The statement $\exists v_1, v_2 \epsilon S, p(v_1, v_2) \wedge (\ldots)$ requires that a pair of elements satisfy the relation p and some other condition not specified here. The *UEQ* transformation creates a feature by replacing the existential quantifier $\exists$ with summation ($\sum$) and division statemements. The resulting feature generates pairs of variable bindings and then tests them, essentially conducting an exhaustive search, to find pairs of values that satisfy the relation p.

An improvement in search speed can be obtained if a relation is stored in a way that allows its retrieval given just a subset of its elements. Such a relation can be used to seed, or initialize, a permutation of variables. The remaining variables can still be determined by generate-and-test. This approach offers a large improvement in search speed, because it eliminates part of the search for permutations that satisfy the conditions in the quantified statement. It is particularly useful for static relations, such as the adjacency of squares on a chessboard.

It is not difficult to store, or *cache*, relations. Some dialects of the Prolog programming language do so automatically [Quintus Computer Systems, 1990]. However, most programming languages do not, so that ability must be explicitly provided by the code generators for those languages.

One difficulty with storing, or *caching*, relations is that it is difficult to know automatically when to do so or how such a relation should be indexed. The decision about whether or not to store a relation depends upon its size, the frequency with which it is used, and perhaps where in the computation of a feature value it is used. The decision about how best to index a relation depends upon the order in which elements of the relation are known; this order is feature-dependent, and could be different for each feature. Therefore, such decisions would be most reliable if they were made after analysis of the set of features, rather than after analysis of any single feature.

9.2.2 Constant Features

The syntactic pruning rules discussed in Section 3.1 were designed to identify and eliminate features that produce constant values. However, in the experiments, some contant features were created that the current set of rules could not identify. Those constant features not identified by the syntactic pruning rules were identified and discarded later by the GFS feature selection algorithm, but only after it had expended substantial effort.

The arithmetic inequalities $<$ and $>$ impose an ordering on their arguments. One can use these inequalities to create sentences that are syntactically correct but impossible to satisfy. For example, the sentence

$$\forall v_1, v_2 \epsilon S, f(v_1) < f(v_2)$$

cannot be true of a non-empty finite set S and a well-defined function f. Sentences that are impossible to satisfy (or not satisfy) may occur in the original search problem specification, or may be created when the *LE* transformation removes an expression from its surrounding context.

Sentences that are impossible to satisfy define Boolean features that are *FALSE* in every search states. Sentences that are impossible not to satisfy define Boolean features that are *TRUE* in every search states. Features with constant values are sometimes interesting because they represent universal truths about the domain.

However, features with constant values do not distinguish among search states, so it is best to discard constant features when learning search control knowledge.

Specific cases of sentences that are impossible to satisfy, or impossible not to satisfy, can be handled by syntactic pruning rules designed for those cases. For example, one could design rules to identify and discard certain types of sentences containing inequalities. However, the more general case of identifying sentences that are never true (or always true) requires deductive capabilities not present in knowledge-based feature generation.

9.3 Related Research on Knowledge-Based Feature Generation

A widely-adoped approach to constructive induction is to repeatedly create new features as simple Boolean or arithmetic functions of existing features. This approach to constructive induction is heuristic search, so success depends upon both the branching factor and the distance of the initial state to a goal state [Callan, 1990; Matheus, 1990b]. In this case, the initial state is the initial set of features and a goal state is any set of features that provides a desired learning behavior. An extremely poor initial vocabularly may make intractable the search for a desired representation.

Knowledge-based feature generation was originally proposed as a method of creating an initial representation for concept learning [Callan, 1989]. An unanswered question was whether the initial representation should consist of knowledge-based features alone, or a combination of knowledge-based features and the problem-solver's (presumably structural) features.

The experimental results support the conclusion that it is usually better to create an initial representation for concept learning by augmenting the problem-solver's representation with additional features. In fourteen out of fifteen dataset experiments, the best performance was obtained with a combination of knowledge-based and structural features. Only in one experiment did knowledge-based features alone outperform the combination.

9.4 Related Research on Constructive Induction

Research on the representation problem can be divided into statistical and deterministic approaches. Statistical approaches perform *feature extraction* for statistical pattern recognition. Deterministic approaches perform *constructive induction* for inductive learning. Both approaches address the representation problem, but with differing methods and purposes.

Feature extraction algorithms assume that classifications are random variables from a distribution whose probability density functions are known or estimated. The intent of feature extraction is to reduce the dimensionality of the representation while preserving information [Tou & Gonzalez, 1974; Kittler, 1986]. Dimensionality reduction is desirable because it reduces the costs of transmitting data and building (copying) pattern classifiers. The costs associated with *finding* a pattern classifier are ignored in pattern recognition.

Constructive induction algorithms assume that classifications are made by a *concept*, which is a decision function that must be determined by an inductive learning algorithm. The intent of constructive induction is to improve the performance of the inductive learning algorithm [Michalski, 1983]. Performance may be measured by the speed with which a concept is learned, by the accuracy of the concept learned, or by some other metric. Costs associated with transmitting data and building (copying) concept descriptions are ignored.

Among constructive induction algorithms, one can classify algorithms by how the search for new features is biased. The principle sources of bias are the structure of the learned concept, the accuracy with which the learned concept classifies examples, and domain knowledge. FRINGE [Pagallo, 1989], CITRE [Matheus, 1990a; Matheus, 1990b] and IB3-CI [Aha, 1991] are biased by the structure of the concept. BACON [Langley, Bradshaw & Simon, 1983] and STAGGER [Schlimmer & Granger, 1986] are biased by the accuracy with which the concept classifies examples. CITRE, STABB [Utgoff, 1986b] and knowledge-based feature generation are biased by domain knowledge. The presence of CITRE in two different classes illustrates that a search for new features can have multiple sources of bias.

Knowledge-based feature generation is most similar to constructive induction algorithms whose search is biased by domain knowledge. The rest of this chapter surveys systems, algorithms and approaches with important similarities to knowledge-based feature generation. A system, algorithm or approach is considered related if it involves change of representation, induction, and domain knowledge. The discussion of related work is loosely organized by the amount of effort that must be expended to use the relevant domain knowledge. The discussion begins with systems, algorithms and approaches that are able to use domain knowledge directly and continues with systems that expend progressively more effort to transform the domain knowledge from its original form into a form more appropriate for a specific task.

9.4.1 CITRE and IB3-CI

CITRE [Matheus, 1990a; Matheus, 1990b] and IB3-CI [Aha, 1991] illustrate the use of domain knowledge supplied for a specific task. Both systems create new features by applying Boolean operators to existing features. CITRE allows the structure of the developing concept to guide its search for new features, while IB3-CI is guided by the accuracy of the developing concept. Once features are created, both systems use domain knowledge to *prune* the set of constructed features. For example, CITRE's

Tic-Tac-Toe domain knowledge consists of two constraints that restrict the feature construction operators to adjacent squares occupied by the same player [Matheus, 1990b].

There are three advantages to this use of domain knowledge. First, it does not require a correct or complete domain theory. Second, it provides a means of pruning features before their use, which can improve both the speed and accuracy of the learning algorithm. Finally, it can be used with any inductive algorithm. The most important disadvantage is that it is not yet clear how such domain knowledge can be supplied automatically.

9.4.2 OXGate

OXGate [Gunsch, 1991] extends the use of specialized domain knowledge in constructive induction by outlining a framework in which domain knowledge can influence all aspects of feature generation. Like CITRE and IB3-CI, OXGate can use domain knowledge to prune newly constructed features that violate domain-dependent conditions. OXGate can also use domain knowledge to influence the selection of feature construction operators and the features to which they are applied. The widespread use of domain knowledge in OXGate offers an extremely focused approach to solving the representation problem. Its success or failure rests upon acquiring the appropriate domain knowledge, and then controlling its application. The difficulty of these tasks is not yet known.

9.4.3 Counting Arguments Rules

Michalski's (1983) *counting argument* rules illustrate one way that a more general type of domain knowledge can be transformed into features. The counting arguments rules are applied to statements that contain quantified variables. Each rule counts the permutations of objects that satisfy a condition. However, [Michalski, 1983] does not specify the source of the condition to which the rules are applied.

Knowledge-based feature generation's *UEQ* transformation is similar to the *counting arguments* rules, but the *UEQ* transformation is applied to expressions generated by the *LE* transformation.

9.4.4 Abstraction

Knowledge-based feature generation has much in common with work that tries to apply a *hierarchical planning* approach to search problems in which no hierarchy is readily apparent. The intent of that work is to generate automatically a set of less detailed, or more *abstract*, problems whose solutions would guide search. Gaschnig (1979), Guida and Somalvico (1979) and Pearl (1984) all showed that abstract problems could be defined by deleting parts of the original problem definition. However, they were faced with two problems for which they had no solution. First, the number of abstractions of a problem is combinatorially explosive, so attention must somehow be restricted to a small subset. Second, the combined costs of blind search in the abstract space and guided search in the original problem space was *greater* than the cost of breadth-first search in the original problem space [Valtorta, 1983].

The *LE* transformation creates abstractions, but it is not subject to either of the above problems. Combinatorial explosion is not a problem because of the restricted way in which the *LE* transformation is applied. It can only create between c and $2c$ abstractions, where c is the number of connectives and negations in the problem definition. The cost of searching is not a problem, because *LE* terms define statements that are either true or not true of the current state; they do *not* define problems that must be solved.

9.4.5 ABSOLVER

There are other overlaps between knowledge-based feature generation and ABSOLVER, a system that represents the most recent work on using abstractions. ABSOLVER addressed the earlier problems with abstractions by extending the set of abstracting transformations, adding a set of optimizing transformations, and carefully controlling the application of transformations [Mostow & Prieditis, 1989; Prieditis, 1991]. The *LE* transformation is a restricted form of ABSOLVER's `drop_pre(p,o)` and `drop_goal(p)` transformations, and it implicitly includes ABSOLVER's `Remove Irrelevant` optimizing transformation. The *UEQ* transformation is equivalent to ABSOLVER's `count(p)` transformation.

One important difference is that knowledge-based feature generation uses a first order predicate calculus representation, while ABSOLVER uses a STRIPS-style representation. First order predicate calculus is the more general of the two. STRIPS-style representations do not admit universal quantifiers, nor do they admit all forms of negatation. These limitations make it unclear how ABSOLVER could be applied to a problem like OTHELLO, where an operator's effects depend upon complex relationships between a reference element and a set of elements.

9.4.6 ZENITH

ZENITH is a hybrid system that uses both abstraction and deduction to create new features for inductive learning [Fawcett & Utgoff, 1992; Fawcett & Utgoff, 1991]. The combination of approaches offers several advantages over either approach alone. Abstraction enables ZENITH to create features from intractable domain theories, and to create features that are not deductively implied by domain theories. Deduction enables ZENITH to create features that measure enabling conditions of other features.

As with other systems that incorporate abstraction, there are several similarities between ZENITH and knowledge-based feature generation. Both represent their domain knowledge in first order predicate calculus. In addition to the problem definition, ZENITH's domain knowledge includes operator preimage expressions and meta-knowledge. ZENITH's `Split-conjunction` and `Remove-negation` transformations collectively make up two thirds of the *LE* transformation; ZENITH does not also split disjuncts. *All* of ZENITH's transformations count the number of ways that a condition can be satisfied, which makes them similar to the *UEQ* transformation. Finally, the CINDI program implicitly incorporates a capability equivalent to ZENITH's `expand-definition` transformation.

ZENITH's set of abstracting transformations is very similar to those of knowledge-based feature generation, but there are important differences between the two approaches. One difference is ZENITH's deductive capabilities which, when combined with abstraction, enable it to create features that represent the enabling conditions of goals and constraints. A second difference is that ZENITH's features are created in response to problem-solving performance, whereas knowledge-based feature generation uses no such feedback. The ability to use feedback enables ZENITH to recognize useless new features, and to focus its search for new features around existing features of high utility. Finally, Zenith has nothing that corresponds to the *AE-M* and *AE-E* transformations, although both could be incorporated into Zenith if desired.

The advantage of knowledge-based feature generation over Zenith is the ease with which knowledge-based feature generation can be applied to new domains. The CINDI program, written in C, is self-contained. Zenith requires a Prolog environment, an integrated problem-solver, and an integrated learning algorithm. If one had a new domain and access to both programs, one could use CINDI to generate an initial set of features and begin learning immediately. Once a domain-specific problem-solver was integrated with Zenith, one could replace features created by CINDI with features created by Zenith.

9.4.7 MIRO

The problem of using domain knowledge to create new terms for inductive learning has also been addressed in MIRO [Drastal, Czako & Raatz, 1989]. MIRO's new terms are the intermediate nodes that arise when it uses a domain theory to construct all possible proof trees for a set of labelled training instances. The differences between MIRO and knowledge-based feature generation can be seen in how they exploit domain knowledge. MIRO uses it to create new terms that are deductive consequents of instance features, but it can do so only if the domain theory is tractable and expressed in a restricted language. Knowledge-based feature generation can use intractable domain theories expressed in a more general language, but it can do so only with heuristic transformations.

9.4.8 STABB

STABB's constraint back-propagation procedure is a more goal-oriented approach to constructive induction. It is performed by taking a goal concept and backpropagating its description through a sequence of operators [Utgoff, 1986a]. The result is a set of ordered pairs, (o_i, S_i); each o_i is an operator which, if applied to a state $s \epsilon S_i$, leads towards the goal. The descriptions of each set S_i form the basis of constructive induction. Each part of a description is checked against the current vocabulary. Expressions that match the definitions of existing attributes are replaced by those attributes. Expressions that do not match the definition of any existing attributes define new attributes.

The advantage of constraint back-propagation in constructive induction is that it produces exactly the set of new attributes needed to distinguish among the states along and not along the solution path. Its power comes from its access to a solution sequence, its ability to back-propagate the description of a set of states through the operators, and its access to an attribute hierarchy. However, this advantage is offset by a disadvantage. If operators are complex, it may not be obvious how to propagate descriptions of sets of states backwards through operators; not all operators have clearly defined counterparts that operate in the backwards direction. This characteristic limits the domains in which STABB can be applied.

CHAPTER 10
CONCLUSIONS

The previous chapters describe knowledge-based feature generation, how it works, the experiments that support it, and its strengths and weaknesses. Their detail is necessary for a rigorous treatment of the subject, but it can also obscure the larger issues. This chapter is intended to provide more perspective. It reviews what has been done, and identifies issues for future research.

10.1 Summary

This dissertation began with the observation that it can be a long leap from the directly observable characteristics of a problem to certain, well-known, useful features for those problems. It is not clear how feature construction algorithms that are based on heuristic search (e.g., [Schlimmer & Granger, 1986; Pagallo, 1989; Matheus, 1990b; Murphy & Pazzani, 1991; Watanabe & Rendell, 1991]) or deductive proof (e.g., [Tadepalli, 1989; Drastal, Czako & Raatz, 1989; Flann, 1990]) can make this leap. Heuristic search succeeds when the initial representation does not require major changes, and deductive methods succeed when the domain theory is not complex. However, it is unlikely that either approach is capable of discovering a feature like *mobility* for Othello if given only the rules of the game as domain knowledge and the contents of each square as an initial representation.

Heuristics enable search-based methods to create features for intractable domains, while domain knowledge enables deductive methods to create features relevant to a particular goal. The thesis is that heuristics and domain knowledge can be combined into a new method of feature construction that offers some of the strengths of search-based and deductive methods of feature creation.

Primary Thesis: Functional features that are useful for learning search control knowledge can be created automatically using heuristics and a search problem specification.

The thesis claims were restricted to a single broad class of learning problems because doing so provided a well-defined source of domain knowledge and motivated the development of a set of feature creation heuristics.

The thesis research had three objectives, which were stated in Chapter 1 and are repeated here.

Objective 1: Develop a heuristic method that uses easily available domain knowledge to create functional features.

Objective 2: Determine whether the features reliably improve the performance of inductive learning algorithms on a variety of problems.

Objective 3: Determine the limitations of the method.

The question to be answered in this last chapter of the dissertation is "Were these objectives met?"

10.1.1 Knowledge-Based Feature Generation

The first of the thesis objectives was development of a heuristic method that uses easily available domain knowledge to create functional features. The easily available domain knowledge for search problems is a specification that describes an initial state, a set of search operators, and the goal state(s). The desired set of features would enable recognition of the state in a set of states that is closest to a goal state. The problem faced in Chapter 2 was how to use the former to create the latter.

Goal states are described in search problem specifications by a set of conditions, or *constraints*, that a goal state must satisfy. The set of constraints defines a Boolean feature, whose value is *TRUE* if and only if a state is a goal. Search control knowledge requires an ability to distinguish among states that are different distances from a goal state, something this Boolean feature does not do. Section 2.2 argues that this distinction among states can be made by measuring the degree to which various search states satisfy the goal constraints. A heuristic was then developed that systematically breaks the description of a goal into its component parts. Each of the component parts defines a Boolean feature. Finer-grained distinctions among search states are made by additional heuristics that transform some types of Boolean features into numeric features.

The features created by knowledge-based feature generation may be simple to state, but determining their truth (for Boolean features) or their value (for numeric features) may involve complex computations. Section 2.4 addressed the issue of making complex features as efficient to evaluate as possible, so that their representational benefits outweigh their computational costs.

Chapter 3 discussed the problem of recognizing, as quickly as possible, which features are useful and which are useless. It presented two approaches to the problem. Heuristic feature selection is a set of rules that identify useless features by their syntactic structure. Data-directed feature selection is a general method of identifying the most useful features once a set of examples and a learning algorithm are available. The data-directed feature selection is based almost entirely upon Saxena's (1991) ACR algorithm.

Section 2.6 described a program called CINDI that implements the feature creation transformations, feature optimization, and heuristic feature selection.

10.1.2 Experimental Results

The second of the thesis research objectives was to determine whether the developed method of feature creation reliably improves the performance of inductive learning algorithms on a variety of problems. A series of experiments was designed in which four inductive learning algorithms (the Perceptron, RLS, C4.5, LMDT) were trained on search control tasks from four different learning problems (stacking blocks, tic-tac-toe, Othello, PWB component placement). Two types of experiments were conducted. In the *dataset accuracy* experiments, a population of instances was randomly partitioned into training and test data, the algorithm was trained on the training data, and then tested on the test data. In the *performance accuracy* experiments, a learned concept was required to control a search algorithm, and then its performance was evaluated using domain-specific criteria.

The experimental results demonstrated that knowledge-based feature generation reliably improves the performance of these inductive learning algorithms on a variety of tasks. Seventeen of eighteen dataset accuracy experiments showed a clear, statistically significant, improvement. Four of the five performance accuracy experiments showed an improvement. In one case where no improvement was found, it was interesting to note that another inductive learning algorithm working from the same data *did* exhibit an improvement. In other words, sometimes features were found that one inductive algorithm found useful but another did not.

10.1.3 Limitations of the Method

The final of the thesis research objectives was to determine the limitations of the method. Those limitations were discussed in Chapter 9. The limitations are generally caused by the inability of the heuristics to create numeric features for problem specifications that contain neither quantifiers nor arithmetic operators. The lack of numeric features means that there may be states among which the features should, but do not, distinguish. Knowledge-based feature generation is also hindered by features that are overly complex in some cases.

Among problems that have quantifiers and/or arithmetic operators, knowledge-based feature generation was found to be generally effective. The transformations apply to statements in first order predicate calculus, a language that is general and expressive. The method is not limited to use with any particular inductive learning algorithm, as the experiments demonstrate. It can also be used in combination with other feature creation algorithms.

10.2 Contributions to Machine Learning

This dissertation describes a new approach to constructive induction. The approach is based upon heuristic transformation of a general and easily available form of domain knowledge. This dissertation demonstrates the approach with *knowledge-based feature generation*, a specific method of transforming knowledge into features. It may be possible to develop other feature creation methods that are also consistent with the general approach.

Knowledge-based feature generation is unique among constructive induction algorithms because it does not require feedback from a learning algorithm or a learned concept, it does not conduct heuristic search, and it can create complex functional features for intractable domains. The method also has the advantage that it has been demonstrated to work well with several inductive learning algorithms.

The CINDI program, which implements knowledge-based feature generation, addresses issues thought to be important but never confronted in constructive induction research. In particular, it demonstrates simple, effective, rules for recognizing useless features, methods for estimating feature complexity, and the importance of feature optimization in complex domains. These methods address problems that have been ignored in previous constructive induction research.

Prior research showed that functional features could be useful for inductive learning (e.g., [Flann & Dietterich, 1986]), but the methods for creating them were either complex computationally (e.g., [Rendell, 1985]) or required very specific domain knowledge (e.g., [Matheus & Rendell, 1989]). Knowledge-based feature generation is a solution to this problem.

In experiments with knowledge-based feature generation, a combination of structural and functional features usually enabled more accurate learning than either structural or functional features alone. This result supports and extends previous research in which a combination of structural and functional features was found to be superior to structural features alone [Matheus & Rendell, 1989; Matheus, 1990b]. These results suggest that future work should include both efforts to better understand functional features and efforts to understand the relationship between functional and structural features.

This dissertation also provides evidence that some features are generally useful, i.e., useful for a variety of learning algorithms. This result is not surprising, because humans can develop features for problem-solving without knowing the environment in which the features will be used (e.g., [Mitchell, 1984]). However, the constructive induction research community has behaved as if features could only be created for particular learning algorithms exhibiting particular types of behavior. This dissertation makes clear that knowledge of the learning algorithm, while perhaps useful, *is not always necessary* for constructive induction. It is possible to create useful new features and to discard useless features without knowing anything about the learning algorithm.

This research also has the potential to change the way people use inductive learning algorithms. Past research on constructive induction demonstrated the utility of *approaches* to feature creation, but few outside the research community actually implemented those approaches. Anecdotal evidence suggests that most people still create representations manually. The CINDI program is unusual because it is a 'standalone' program. The effort required to use the program is low, and the experimental evidence shows that CINDI is capable of creating useful features for a range of problems and a range of learning algorithms. An 'off the shelf' program like CINDI, which requires no programming effort, and which can be run in a matter of seconds, might begin to change the status quo.

Finally, the thesis research is significant because it shows that in constructive induction there is a middle ground between ignoring general forms of domain knowledge and embracing it with deduction. One of the methods in this middle ground, knowledge-based feature generation, is shown to be a tractable source of features that can not be created by knowledge-free constructive induction algorithms. There are probably other algorithms for other classes of problems. This dissertation is important both for the research that it reports, and the research that it suggests.

10.3 Directions For Future Research

The research described in this dissertation raises a variety of questions that are worthy of future research. This section describes six such questions. The first three questions concern ways in which knowledge-based feature generation can be made more efficient or more broadly applicable. Questions four and five concern the behavior of functional features in constructive induction and inductive learning algorithms. The final question asks whether knowledge-based feature generation, or some similar feature creation method, can create features for other classes of learning problems. Each of these questions is described below in more detail.

10.3.1 Separate *UQ* and *EQ* Transformations

This dissertation offers just one transformation to handle both universal ($\forall$) and existential ($\exists$) quantifiers. The justification for handling both quantifiers with one transformation is given in Chapter 2. The results of doing so have been acceptable.

UEQ terms count the number of permutations that satisfy a condition. This type of term is appropriate for the universal quantifier, which requires that *all* permutations satisfy a condition. However, a feature that considers *every* permutation may be unnecessarily expensive for the existential quantifier, which requires that just *one* permutation satisfy the condition. One question for future research is whether a less expensive term might be created from existentially quantifier statements.

Note that the issue is not whether the *UEQ* transformation produces meaningful features from existentially quantified statements. When the *UEQ* transformation is applied to an existential quantifier, the resulting term indicates the amount of freedom the problem-solver has in satisfying the condition. Such a measurement can be useful in selecting search states [Dechter & Pearl, 1987]. An issue for future research is whether less expensive terms of equivalent utility can be created.

10.3.2 Improved Feature Efficiency

It is desirable that features be as efficient to evaluate as possible, because every feature can be evaluated once per search state. An improvement in the speed of a feature enables the search procedure to search more states with the same effort, or to search the same number more rapidly. More importantly, a difference in efficiency may determine whether a feature's cost outweighs its benefit; in this case, the difference in speed determines whether the feature should be used or discarded.

The current set of transformations, pruning rules, optimizing transformations and code generators are designed to be understandable and to produce meaningful features. They do not always produce the most efficient features. Section 9.2 discusses some approaches to improved feature efficiency, but there are doubtless other approaches.

10.3.3 Transformations on Other Predicates

Chapter 2 demonstrates that it is possible to transform sentences with arithmetic equality and inequality predicates into numeric features. It is an open question whether there are similarly useful transformations for other predicates.

It is likely that useful transformations can be developed for predicates that assert relationships between sets. For example, the predicates that assert subset and superset relationships $(\subset, \subseteq, \supset, \supseteq)$ order sets in a manner analagous to arithmetic inequality predicates $(<, \leq, >, \geq)$. Therefore one might transform these predicates by creating numeric features that express the *difference* between two sets. One might represent the difference by the number of elements on which the two sets differ, or by the number of elements that the two sets have in common.

If it is possible to develop transformations that produce meaningful features from predicates on sets, there are probably other class of predicates for which transformations might be developed. Finding them all would be a time-consuming task. For now, it is probably a better use of resources to apply knowledge-based feature generation to problems of interest, developing new transformations whenever a need arises.

10.3.4 Constructive Induction on Functional Features

Knowledge-based feature generation does not preclude use of other constructive induction algorithms. One could use CINDI to create functional features and then use another algorithm, like FRINGE [Pagallo, 1989] or CITRE [Matheus, 1990a; Matheus, 1990b], to create new structural features. The ability to mix and match constructive induction algorithms could yield better representations for inductive learning than any single constructive induction algorithm.

One could also apply FRINGE or CITRE to the functional features created by CINDI. Would any useful new features be created? The answer is not known. It is possible that some feature construction methods developed for structural features are also applicable to functional features. Perhaps useful new functional features can be created as Boolean or arithmetic functions of existing functional features. If so, a combination of methods might be more effective than either alone.

Knowledge-based feature generation creates features ranging from complex to simple. If another constructive induction algorithm can combine simple functional features into more complex functional features, one could begin learning with the simplest functional features created by knowledge-based feature generation. More complex functional features would be constructed, as needed, by combining simpler functional features. A complex functional feature created in this manner would sometimes only approximate a functional feature created by knowledge-based feature generation, due to the loss of constraints among its components. However, information about the ancestory of each feature, for example as produced by the CINDI program, would enable the constructive induction algorithm to identify this correspondance. The algorithm could then decide whether to replace its approximation with the feature created by knowledge-based feature generation.

The hybrid approach, like most existing constructive induction algorithms, would start with simple features and add more complex features as needed. If it worked, it ought to be faster than simply running CINDI, using all of the features initially, and then discarding those that prove ineffective.

10.3.5 Interaction Between Structural and Functional Features

The experiments described in Chapters 5, 6, 8 and 7 raise the question of how structural and functional features interact. Some interaction appears to occur, because the combination of structural and functional features consistently yielded more accurate evaluation functions than either structural or functional alone. It is unlikely that the superior performance of the combined representations is due to having a greater number of features. Although the combined representations always had the largest number of features, there were also many examples of smaller representations being more useful than larger representations.

The nature of the interaction between structural and functional features is unknown. However, it is possible that the two different types of features play different roles in representing search control knowledge. Perhaps structural features identify particular contexts in which functional features can select particular policies. Perhaps it is the functional features that identify the contexts. A better understanding of this interaction should lead to constructive induction algorithms that deliberately supply, and inductive algorithms that deliberately exploit, different types of features.

10.3.6 Other Classes of Learning Problems

Some of the learning problems studied commonly are classification problems, as opposed to search-control problems. In a classification problem, each instance must be assigned to one of a small number of classes. Fisher's (1936) Iris data set is one well-known example. It consists of 150 descriptions of Iris plants, each belonging to one of three species. The field of medicine provides classification problems in which one wants to determine whether a patient has a particular disease. Examples include heart disease [Detrano, Janosi, Steinbrunn, Pfisterer, Schmid, Sandhu, Guppy, Lee & Froelicher, 1989], breast cancer, liver disorders and hepatitis[1]. Much is known about the domains (botany, biology, human physiology) in which these classification problems occur. However, this general 'first principles' knowledge has not played a role in constructive induction.

Knowledge-based feature generation was developed for a single class of inductive learning problems. However, the *LE*, *UEQ*, and *AE* transformations are general transformations that apply to sentences in a logical language. Their use is not necessarily restricted to search-control problems. It is an open question whether knowledge-based feature generation, or some similar method, can transform other types of domain knowledge into features that are useful for inductive learning.

Many knowledge-based systems have been developed, but few of them learn from experience. Knowledge-based feature generation could be an important part of adding learning to these systems, because it could use the system's knowledge to create a representation for inductive learning. It is not yet clear whether knowledge-based feature generation can be extended beyond search control problems, or whether similar methods of creating representations can be developed for other problem classes. If either could be achieved, the effort required to add learning abilities to existing systems could be reduced by taking advantage of knowledge already in the system for other purposes.

[1]Databases for breast cancer, liver disorders and hepatitis are available from the University of California at Irvine (UCI) repository of machine learning databases.

APPENDIX A
BLOCKS-WORLD SPECIFICATION

This appendix presents the search problem specification that was used in the experiments described in Section 5. The search problem specification presented here was written by the author.

```
INITIAL:
    B := {a,b,c,d}

GOAL:
    on (d,Table) and
    on (c,d) and
    on (b,c) and
    on (a,b)

OPERATORS:
    stack:
        condition:
            exists u,v in B,
                forall w in B,
                    ((not on (w,u)) and (not on (w,v)))
        action:
            set_on (u,v)
    unstack:
        condition:
            exists u,v in B,
                (on (u,v) and
                 forall w in B, (not on (w,u)))
        action:
            (not on (u,v)) and on (u,Table)
```

A.1 Features Generated

The following is the complete output generated by the CINDI program when it created features for stacking blocks. All input to the program is shown below, except for the problem specification, which is shown above.

```
                    CINDI version 1.24

Is on a static (constant) relation [y|n]?  n

13 features were created (LE=9, AE=0, UEQ=4, Pruned=9).

Please select an output format for the features:
  c:  the C programming language.
  f:  a generic functional notation.
f

;  id=0, Author=LE, status=0, parent-id=user
;  Optimizations:
f_1:
(on d Table)

;  id=1, Author=LE, status=0, parent-id=user
;  Optimizations:
f_2:
(on c d)

;  id=2, Author=LE, status=0, parent-id=user
;  Optimizations:
f_3:
(on b c)

;  id=3, Author=LE, status=0, parent-id=user
;  Optimizations:
f_4:
(on a b)

;  id=4, Author=LE, status=0, parent-id=user
;  Optimizations:
f_5:
(EXISTS u,v in B
  (FORALL w IN B
    (AND
      (! (on w u))
      (! (on w v)))))

;  id=5, Author=LE, status=0, parent-id=user
;  Optimizations:  3 6
f_6:
(EXISTS u in B
  (FORALL w IN B
    (! (on w u))))

;  id=7, Author=LE, status=0, parent-id=5
;  Optimizations:  3 6
f_7:
(EXISTS u in B
  (FORALL w IN B
    (on w u)))

;  id=9, Author=UEQ, status=0, parent-id=user
;  Optimizations:
f_8:
(/
  (SUM u IN B
    (SUM v IN B
      (SUM w IN B
        (IF
          (AND
            (! (on w u))
            (! (on w v)))
          1
          0))))
  (*
    |B|
    (*
      |B|
      |B|)))

;  id=10, Author=UEQ, status=0, parent-id=5
;  Optimizations:  3
f_9:
(/
  (SUM u IN B
    (SUM w IN B
      (IF
        (! (on w u))
        1
        0)))
  (*
    |B|
    |B|))

;  id=12, Author=UEQ, status=0, parent-id=7
;  Optimizations:  3
f_10:
```

```
(/
  (SUM u IN B
    (SUM w IN B
      (IF
        (on w u)
        1
        0)))
  (*
    |B|
    |B|))

;  id=14, Author=LE, status=0, parent-id=user
;  Optimizations:
f_11:
(EXISTS u,v in B
  (AND
    (on u v)
    (FORALL w IN B
      (! (on w u)))))

;  id=15, Author=LE, status=0, parent-id=user
;  Optimizations:
f_12:
(EXISTS u in B
  (EXISTS v in B
    (on u v)))

;  id=18, Author=UEQ, status=0, parent-id=user
;  Optimizations:  0 4
f_13:
(/
  (SUM u IN B
    (COND
      ((! (FORALL w IN B
            (! (on w u))))
        0.000000)
      (TRUE
        (SUM v IN B
          (IF
            (on u v)
            1
            0)))))
  (*
    |B|
    |B|))
```

```
Estimates of feature costs are:
cost_f_1:
1.000000

cost_f_2:
1.000000

cost_f_3:
1.000000

cost_f_4:
1.000000

cost_f_5:
(*
  |B|
  |B|)

cost_f_6:
(*
  |B|
  |B|)

cost_f_7:
(*
  |B|
  |B|)

cost_f_8:
(*
  |B|
  (*
    |B|
    |B|))

cost_f_9:
(*
  |B|
  |B|)

cost_f_10:
(*
  |B|
  |B|)
```

```
cost_f_11:
(*
  |B|
  |B|)

cost_f_12:
(*
  |B|
  |B|)

cost_f_13:
(*
  |B|
  (+
    |B|
    |B|))
```

APPENDIX B

TIC-TAC-TOE SPECIFICATION

The following is the extended First Order Predicate Calculus specification of tic-tac-toe used in the experiments described in Section 6. The search problem specification presented here was written by the author.

```
INITIAL:
    Players := {X, O}
    Squares := {S1,  S2,  S3,  S4,  S5,  S6,  S7,  S8, S9}
    Directions := {n, ne, e, se, s, sw, w, nw}

GOAL:
    win (X)

OPERATORS:
    move:
        condition:
            (exists p in Players,
                turn_to_move (p)) and
            (exists s in Squares,
                ((not owns (X, s)) and
                 (not owns (O, s))))

        action:
            set_owns (p, s)

PREDICATES:
    win (P) :
        exists s_i in Squares,
            (owns (P, s_i) and
             (exists s_j in Squares,
                (exists d in Directions,
                    (neighbor (s_i, d, s_j) and
                     owns (P, s_j) and
                     (exists s_k in Squares,
                         (neighbor (s_j, d, s_k) and
                          owns (P, s_k)))))))
```

```
RELATIONS:
    neighbor:
        { (S1, E, S2), (S2, E, S3), (S2, W, S1), (S3, W, S2),
          (S4, E, S5), (S5, E, S6), (S5, W, S4), (S6, W, S5),
          (S7, E, S8), (S8, E, S9), (S8, W, S7), (S9, W, S8),
          (S1, S, S4), (S4, S, S7), (S4, N, S1), (S7, N, S4),
          (S2, S, S5), (S5, S, S8), (S2, N, S5), (S8, N, S5),
          (S3, S, S6), (S6, S, S9), (S3, N, S6), (S9, N, S6),
          (S1, SE, S5), (S5, SE, S9), (S5, NW, S1), (S9, NW, S5),
          (S2, SE, S6), (S4, SE, S8), (S2, NW, S6), (S8, NW, S4),
          (S2, SW, S4), (S3, SW, S5), (S4, NE, S2), (S5, NE, S3),
          (S5, SW, S7), (S6, SW, S8), (S7, NE, S5), (S8, NE, S6) }
```

B.1 Features Generated

The following is the complete output generated by the CINDI program when it created features for tic-tac-toe. All input to the program is shown below, except for the problem specification, which is shown above.

```
                  CINDI version 1.24

Is owns a static (constant) relation [y|n]?  n
Is neighbor a static (constant) relation [y|n]?  y
Is turn_to_move a static (constant) relation [y|n]?  y

16 features were created (LE=8, AE=0, UQ=8, Pruned=9).

Please select an output format for the features:
  c:  the C programming language.
  f:  a generic functional notation.
f

;  id=0, Author=LE, status=0, parent-id=user
;  Optimizations:  3
f_1:
(EXISTS s_i in Squares
  (owns X s_i))

;  id=1, Author=LE, status=0, parent-id=user
;  Optimizations:  1 4 6
f_2:
(EXISTS s_j in Squares
  (AND
    (owns X s_j)
    (EXISTS d in Directions
      (AND
        (EXISTS s_k in Squares
          (AND
            (neighbor s_j d s_k)
            (owns X s_k)))
        (EXISTS s_i in Squares
          (neighbor s_i d s_j))))))

;  id=4, Author=LE, status=0, parent-id=1
;  Optimizations:  1 4 6
f_3:
(EXISTS s_k in Squares
  (AND
    (owns X s_k)
    (EXISTS d in Directions
      (EXISTS s_j in Squares
        (neighbor s_j d s_k)))))

;  id=7, Author=UEQ, status=0, parent-id=user
;  Optimizations:  1
f_4:
(/
  (SUM s_i IN Squares
    (IF
      (AND
        (owns X s_i)
        (EXISTS s_j in Squares
          (AND
            (owns X s_j)
            (EXISTS d in Directions
              (AND
                (neighbor s_i d s_j)
                (EXISTS s_k in Squares
                  (AND
                    (neighbor s_j d s_k)
                    (owns X s_k))))))))
      1
      0))
  |Squares|)

;  id=8, Author=UEQ, status=0, parent-id=0
;  Optimizations:  3
```

```
f_5:
(/
  (SUM s_i IN Squares
    (IF
      (owns X s_i)
      1
      0))
  |Squares|)

;  id=9, Author=UEQ, status=0, parent-id=1
;  Optimizations:  0 4
f_6:
(/
  (SUM s_j IN Squares
    (COND
      ((! (owns X s_j))
        0.000000)
      (TRUE
        (SUM d IN Directions
          (COND
            ((! (EXISTS s_k in Squares
                  (AND
                    (neighbor s_j d s_k)
                    (owns X s_k))))
              0.000000)
            (TRUE
              (SUM s_i IN Squares
                (IF
                  (neighbor s_i d s_j)
                  1
                  0))))))))
  (*
    |Squares|
    (*
      |Directions|
      |Squares|)))

;  id=11, Author=UEQ, status=0, parent-id=4
;  Optimizations:  0 4
f_7:
(/
  (SUM s_k IN Squares
    (COND
      ((! (owns X s_k))
        0.000000)
      (TRUE
        (SUM d IN Directions
          (SUM s_j IN Squares
            (IF
              (neighbor s_j d s_k)
              1
              0))))))
  (*
    |Squares|
    (*
      |Directions|
      |Squares|)))

;  id=13, Author=LE, status=0, parent-id=user
;  Optimizations:
f_8:
(AND
  (EXISTS p in Players
    (turn_to_move p))
  (EXISTS s in Squares
    (AND
      (! (owns X s))
      (! (owns O s)))))

;  id=15, Author=LE, status=0, parent-id=user
;  Optimizations:
f_9:
(EXISTS s in Squares
  (AND
    (! (owns X s))
    (! (owns O s))))

;  id=16, Author=LE, status=0, parent-id=15
;  Optimizations:
f_10:
(EXISTS s in Squares
  (! (owns X s)))

;  id=17, Author=LE, status=0, parent-id=15
;  Optimizations:
f_11:
(EXISTS s in Squares
  (! (owns O s)))

;  id=19, Author=LE, status=0, parent-id=17
;  Optimizations:
f_12:
(EXISTS s in Squares
  (owns O s))

;  id=20, Author=UEQ, status=0, parent-id=15
;  Optimizations:
f_13:
(/
  (SUM s IN Squares
    (IF
      (AND
        (! (owns X s))
        (! (owns O s)))
      1
      0))
  |Squares|)

;  id=21, Author=UEQ, status=0, parent-id=16
;  Optimizations:
f_14:
(/
  (SUM s IN Squares
    (IF
      (! (owns X s))
      1
      0))
  |Squares|)

;  id=22, Author=UEQ, status=0, parent-id=17
;  Optimizations:
f_15:
(/
  (SUM s IN Squares
    (IF
      (! (owns O s))
      1
      0))
  |Squares|)

;  id=24, Author=UEQ, status=0, parent-id=19
;  Optimizations:
f_16:
(/
  (SUM s IN Squares
    (IF
      (owns 0 s)
      1
      0))
  |Squares|)

Estimates of feature costs are:
cost_f_1:
|Squares|

cost_f_2:
(*
  |Squares|
  (*
    |Directions|
    (+
      |Squares|
      |Squares|)))

cost_f_3:
(*
  |Squares|
  (*
    |Directions|
    |Squares|))

cost_f_4:
|Squares|

cost_f_5:
|Squares|

cost_f_6:
(*
  |Squares|
  (*
    |Directions|
    (+
      |Squares|
      |Squares|)))

cost_f_7:
(*
  |Squares|
  (*
    |Directions|
    |Squares|))

cost_f_8:
(+
  |Players|
  |Squares|)

cost_f_9:
|Squares|

cost_f_10:
|Squares|

cost_f_11:
|Squares|

cost_f_12:
|Squares|

cost_f_13:
|Squares|

cost_f_14:
|Squares|

cost_f_15:
|Squares|

cost_f_16:
|Squares|
```

APPENDIX C

PWB COMPONENT PLACEMENT SPECIFICATION

The POOL algorithm[1] requires that each component search independently for a location that satisfies the design rules. This requirement effectively transforms the placement of the PWB into multiple small search problems, each carried out by a single component. The problem specification below states the PWB component placement problem for a single component.

Each component movement was limited by the size of the component. A component could not move more than half its width or height in a single movement. This restriction is represented in the sizes of the HorizMoves and VertMoves sets.

Each component was permitted to know the location of every other component. However, the components could not communicate, so they did not know where another component would move next (if it moved at all).

```
INITIAL:
    HorizMoves := {-width/2, ..., -1, 0, 1, ..., width/2}
    VertMoves := {-height/2, ..., -1, 0, 1, ..., height/2}

GOAL:
    (Fixed (c) OR
      (on_board (c) AND
       no_overlap (c) AND
       spacing_ok (c)))

OPERATORS:
    move:
        condition:
            TRUE
        action:
            Exists dx in HorizMoves,
                Exists dy in VertMoves,
```

[1]Unpublished algorithm by Paul Utgoff (personal communication).

```
                    MoveComponent (c, dx, dy)

PREDICATES:
    no_overlap (c):
        ForAll c_j in Comps,
            ((c == c_j) OR
             ((Side (c) != Side (c_j)) AND
              SurfaceMount (c) AND
              SurfaceMount (c_j)) OR
             (XMin (c)   >  XMax (c_j)) OR
             (XMin (c_j) >  XMax (c)) OR
             (YMin (c)   >  YMax (c_j)) OR
             (YMin (c_j) >  YMax (c)))

    on_board (c):
        ((XMin(c) >= XMin(BRD)) AND
         (YMin(c) >= YMin(BRD)) AND
         (XMax(c) <  XMax(BRD)) AND
         (YMax(c) <  YMax(BRD)))

    spacing_ok (c):
        (ForAll c_j in Comps,
            ((c == c_j) OR
             ((Side (c) != Side (c_j)) AND
              SurfaceMount (c) AND
              SurfaceMount (c_j)) OR
             ((XMin (c) - 100) >  XMax (c_j)) OR
             ((XMax (c) + 100) <  XMin (c_j)) OR
             ((YMin (c) - 100) >  YMax (c_j)) OR
             ((YMax (c) + 100) <  YMin (c_j))))
```

C.1 Features Generated

The following is the output generated by the CINDI program when it created features for PWB component layout. The cost estimates have been eliminated because they are easy to recreate. See the previous appendices for examples of cost estimates produced by CINDI. All input to the program is shown below, except for the problem specification, which is shown above.

The variable c in the features below is not explicitly bound in any feature. This characteristic is caused by the goal statement, which refers to a variable c that is not explicitly bound. In both cases, c is assumed to refer to the component for which the feature is being evaluated. The goal for a component c is to satisfy the constraints specified. The features describe for a component **c** the extent to which the constraints are satisfied in a particular state.

```
                   CINDI version 1.24

Is Fixed a static (constant) relation [y|n]?  y
Is XMin a static (constant) relation [y|n]?  n
Is Side a static (constant) relation [y|n]?  y
Is SurfaceMount a static (constant) relation [y|n]?  y
Is YMin a static (constant) relation [y|n]?  n
Is XMax a static (constant) relation [y|n]?  n
Is YMax a static (constant) relation [y|n]?  n

60 features were created (LE=16, AE=36, UQ=8, Pruned=14).

Please select an output format for the features:
  c:  the C programming language.
  f:  a generic functional notation.
f

;  id=1, Author=LE, status=0, parent-id=user
;  Optimizations:
f_1:
(AND
  (on_board c)
  (no_overlap c)
  (spacing_ok c))

;  id=2, Author=LE, status=0, parent-id=1
;  Optimizations:
f_2:
(on_board c)

;  id=3, Author=LE, status=0, parent-id=1
;  Optimizations:
f_3:
(no_overlap c)

;  id=4, Author=LE, status=0, parent-id=1
;  Optimizations:
f_4:
(spacing_ok c)

;  id=5, Author=LE, status=0, parent-id=2
;  Optimizations:
```

```
f_5:
(>=
  (XMin c)
  (XMin BRD))

;  id=6, Author=LE, status=0, parent-id=2
;  Optimizations:
f_6:
(>=
  (YMin c)
  (YMin BRD))

;  id=7, Author=LE, status=0, parent-id=2
;  Optimizations:
f_7:
(<
  (XMax c)
  (XMax BRD))

;  id=8, Author=LE, status=0, parent-id=2
;  Optimizations:
f_8:
(<
  (YMax c)
  (YMax BRD))

;  id=11, Author=LE, status=0, parent-id=3
;  Optimizations:
f_9:
(FORALL c_j IN Comps
  (>
    (XMin c)
    (XMax c_j)))

;  id=12, Author=LE, status=0, parent-id=3
;  Optimizations:
f_10:
(FORALL c_j IN Comps
  (>
    (XMin c_j)
    (XMax c)))

;  id=13, Author=LE, status=0, parent-id=3
;  Optimizations:
f_11:
```

```
(FORALL c_j IN Comps
  (>
    (YMin c)
    (YMax c_j)))

;  id=14, Author=LE, status=0, parent-id=3
;  Optimizations:
f_12:
(FORALL c_j IN Comps
  (>
    (YMin c_j)
    (YMax c)))

;  id=17, Author=LE, status=0, parent-id=4
;  Optimizations:
f_13:
(FORALL c_j IN Comps
  (>
    (-
      (XMin c)
      100.000000)
    (XMax c_j)))

;  id=18, Author=LE, status=0, parent-id=4
;  Optimizations:
f_14:
(FORALL c_j IN Comps
  (<
    (+
      (XMax c)
      100.000000)
    (XMin c_j)))

;  id=19, Author=LE, status=0, parent-id=4
;  Optimizations:
f_15:
(FORALL c_j IN Comps
  (>
    (-
      (YMin c)
      100.000000)
    (YMax c_j)))

;  id=20, Author=LE, status=0, parent-id=4
;  Optimizations:
```

```
f_16:
(FORALL c_j IN Comps
  (<
    (+
      (YMax c)
      100.000000)
    (YMin c_j)))

;  id=27, Author=AE-E, status=0, parent-id=2
;  Optimizations:
f_17:
(COND
  ((AND
      (>=
        (XMin c)
        (XMin BRD))
      (>=
        (YMin c)
        (YMin BRD))
      (<
        (XMax c)
        (XMax BRD))
      (<
        (YMax c)
        (YMax BRD)))
    0.000000)
  (TRUE
    (MAX
      (-
        (XMin BRD)
        (XMin c))
      (-
        (XMax c)
        (XMax BRD)))))

;  id=28, Author=AE-E, status=0, parent-id=2
;  Optimizations:
f_18:
(COND
  ((AND
      (>=
        (XMin c)
        (XMin BRD))
      (>=
        (YMin c)
        (YMin BRD))
      (<
        (XMax c)
        (XMax BRD))
      (<
        (YMax c)
        (YMax BRD)))
    0.000000)
  (TRUE
    (MAX
      (-
        (YMax c)
        (YMax BRD))
      (-
        (YMin BRD)
        (YMin c)))))

;  id=29, Author=AE-M, status=0, parent-id=2
;  Optimizations:
f_19:
(COND
  ((AND
      (>=
        (XMin c)
        (XMin BRD))
      (>=
        (YMin c)
        (YMin BRD))
      (<
        (XMax c)
        (XMax BRD))
      (<
        (YMax c)
        (YMax BRD)))
    (MIN
      (-
        (XMin c)
        (XMin BRD))
      (-
        (XMax BRD)
        (XMax c))))
  (TRUE
    0.000000))

;  id=30, Author=AE-M, status=0, parent-id=2
;  Optimizations:
f_20:
(COND
  ((AND
      (>=
        (XMin c)
        (XMin BRD))
      (>=
        (YMin c)
        (YMin BRD))
      (<
        (XMax c)
        (XMax BRD))
      (<
        (YMax c)
        (YMax BRD)))
    (MIN
      (-
        (YMax BRD)
        (YMax c))
      (-
        (YMin c)
        (YMin BRD))))
  (TRUE
    0.000000))

;  id=31, Author=AE-E, status=0, parent-id=3
;  Optimizations:
f_21:
(/
  (SUM c_j IN Comps
    (COND
      ((OR
          (==
            c
            c_j)
          (AND
            (!=
              (Side c)
              (Side c_j))
            (SurfaceMount c)
            (SurfaceMount c_j))
          (>
            (XMin c)
            (XMax c_j))
          (>
            (XMin c_j)
            (XMax c))
          (>
            (YMin c)
            (YMax c_j))
          (>
            (YMin c_j)
            (YMax c)))
        0.000000)
      (TRUE
        (MIN
          (-
            (XMax c_j)
            (XMin c))
          (-
            (XMax c)
            (XMin c_j))))))
  |Comps|)

;  id=32, Author=AE-E, status=0, parent-id=3
;  Optimizations:
f_22:
(/
  (SUM c_j IN Comps
    (COND
      ((OR
          (==
            c
            c_j)
          (AND
            (!=
              (Side c)
              (Side c_j))
            (SurfaceMount c)
            (SurfaceMount c_j))
          (>
            (XMin c)
            (XMax c_j))
          (>
            (XMin c_j)
            (XMax c))
          (>
            (YMin c)
            (YMax c_j))
          (>
            (YMin c_j)
            (YMax c)))
        0.000000)
      (TRUE
        (MIN
          (-
            (YMax c_j)
            (YMin c))
          (-
            (YMax c)
            (YMin c_j))))))
  |Comps|)

;  id=34, Author=AE-M, status=0, parent-id=3
;  Optimizations:
f_23:
(/
  (SUM c_j IN Comps
    (COND
      ((OR
          (==
            c
            c_j)
          (AND
            (!=
              (Side c)
              (Side c_j))
            (SurfaceMount c)
            (SurfaceMount c_j))
          (>
            (XMin c)
            (XMax c_j))
          (>
            (XMin c_j)
            (XMax c))
          (>
            (YMin c)
            (YMax c_j))
          (>
            (YMin c_j)
            (YMax c)))
        (MAX
          (-
            (XMin c)
          (XMax c_j))
        (-
          (XMin c_j)
          (XMax c))))
    (TRUE
      0.000000)))
  |Comps|)

;  id=35, Author=AE-M, status=0, parent-id=3
;  Optimizations:
f_24:
(/
  (SUM c_j IN Comps
    (COND
      ((OR
          (==
            c
            c_j)
          (AND
            (!=
              (Side c)
              (Side c_j))
            (SurfaceMount c)
            (SurfaceMount c_j))
          (>
            (XMin c)
            (XMax c_j))
          (>
            (XMin c_j)
            (XMax c))
          (>
            (YMin c)
            (YMax c_j))
          (>
            (YMin c_j)
            (YMax c)))
        (MAX
          (-
            (YMin c)
            (YMax c_j))
          (-
            (YMin c_j)
            (YMax c))))
      (TRUE
        0.000000)))
  |Comps|)

;  id=36, Author=AE-E, status=0, parent-id=4
;  Optimizations:
f_25:
(/
  (SUM c_j IN Comps
    (COND
      ((OR
          (==
            c
            c_j)
          (AND
            (!=
              (Side c)
              (Side c_j))
            (SurfaceMount c)
            (SurfaceMount c_j))
          (>
            (-
              (XMin c)
              100.000000)
            (XMax c_j))
          (<
            (+
              (XMax c)
              100.000000)
            (XMin c_j))
          (>
            (-
              (YMin c)
              100.000000)
            (YMax c_j))
          (<
            (+
              (YMax c)
              100.000000)
            (YMin c_j)))
        0.000000)
      (TRUE
        (MIN
          (-
            (XMax c_j)
            (-
              (XMin c)
              100.000000))
            (-
              (+
                (XMax c)
                100.000000)
              (XMin c_j))))))
  |Comps|)

;  id=37, Author=AE-E, status=0, parent-id=4
;  Optimizations:
f_26:
(/
  (SUM c_j IN Comps
    (COND
      ((OR
          (==
            c
            c_j)
          (AND
            (!=
              (Side c)
              (Side c_j))
            (SurfaceMount c)
            (SurfaceMount c_j))
          (>
            (-
              (XMin c)
              100.000000)
            (XMax c_j))
          (<
            (+
              (XMax c)
              100.000000)
            (XMin c_j))
          (>
            (-
              (YMin c)
              100.000000)
            (YMax c_j))
          (<
            (+
              (YMax c)
              100.000000)
            (YMin c_j)))
        0.000000)
      (TRUE
        (MIN
          (-
            (YMax c_j)
            (-
              (YMin c)
              100.000000))
          (-
            (+
              (YMax c)
              100.000000)
            (YMin c_j))))))
  |Comps|)

;  id=39, Author=AE-M, status=0, parent-id=4
;  Optimizations:
f_27:
(/
  (SUM c_j IN Comps
    (COND
      ((OR
          (==
            c
            c_j)
          (AND
            (!=
              (Side c)
              (Side c_j))
            (SurfaceMount c)
            (SurfaceMount c_j))
          (>
            (-
              (XMin c)
              100.000000)
            (XMax c_j))
          (<
            (+
              (XMax c)
              100.000000)
            (XMin c_j))
          (>
            (-
              (YMin c)
              100.000000)
            (YMax c_j))
          (<
            (+
              (YMax c)
              100.000000)
            (YMin c_j)))
        (MAX
          (-
            (-
              (XMin c)
              100.000000)
            (XMax c_j))
          (-
            (XMin c_j)
            (+
              (XMax c)
              100.000000))))
      (TRUE
        0.000000)))
  |Comps|)

;  id=40, Author=AE-M, status=0, parent-id=4
;  Optimizations:
f_28:
(/
  (SUM c_j IN Comps
    (COND
      ((OR
          (==
            c
            c_j)
          (AND
            (!=
              (Side c)
              (Side c_j))
            (SurfaceMount c)
            (SurfaceMount c_j))
          (>
            (-
              (XMin c)
              100.000000)
            (XMax c_j))
          (<
            (+
              (XMax c)
              100.000000)
            (XMin c_j))
          (>
            (-
              (YMin c)
              100.000000)
            (YMax c_j))
          (<
            (+
              (YMax c)
              100.000000)
            (YMin c_j)))
        (MAX
          (-
            (-
              (YMin c)
              100.000000)
            (YMax c_j))
          (-
            (YMin c_j)
            (+
              (YMax c)
              100.000000))))
      (TRUE
        0.000000)))
  |Comps|)

;  id=41, Author=AE-E, status=0, parent-id=5
;  Optimizations:
f_29:
(COND
  ((>=
      (XMin c)
      (XMin BRD))
    0.000000)
  (TRUE
    (-
      (XMin BRD)
      (XMin c))))

;  id=42, Author=AE-M, status=0, parent-id=5
;  Optimizations:
f_30:
(COND
  ((>=
      (XMin c)
      (XMin BRD))
    (-
      (XMin c)
      (XMin BRD)))
  (TRUE
    0.000000))

;  id=43, Author=AE-E, status=0, parent-id=6
;  Optimizations:
f_31:
(COND
  ((>=
      (YMin c)
      (YMin BRD))
    0.000000)
  (TRUE
    (-
      (YMin BRD)
      (YMin c))))

;  id=44, Author=AE-M, status=0, parent-id=6
;  Optimizations:
f_32:
(COND
  ((>=
      (YMin c)
      (YMin BRD))
    (-
      (YMin c)
      (YMin BRD)))
  (TRUE
    0.000000))

;  id=45, Author=AE-E, status=0, parent-id=7
;  Optimizations:
f_33:
(COND
  ((<
      (XMax c)
      (XMax BRD))
    0.000000)
  (TRUE
    (-
      (XMax c)
      (XMax BRD))))
```

```
;  id=46, Author=AE-M, status=0, parent-id=7
;  Optimizations:
f_34:
(COND
  ((<
      (XMax c)
      (XMax BRD))
    (-
      (XMax BRD)
      (XMax c)))
  (TRUE
    0.000000))

;  id=47, Author=AE-E, status=0, parent-id=8
;  Optimizations:
f_35:
(COND
  ((<
      (YMax c)
      (YMax BRD))
    0.000000)
  (TRUE
    (-
      (YMax c)
      (YMax BRD))))

;  id=48, Author=AE-M, status=0, parent-id=8
;  Optimizations:
f_36:
(COND
  ((<
      (YMax c)
      (YMax BRD))
    (-
      (YMax BRD)
      (YMax c)))
  (TRUE
    0.000000))

;  id=49, Author=AE-E, status=0, parent-id=11
;  Optimizations:
f_37:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (XMin c)
          (XMax c_j))
        0.000000)
      (TRUE
        (-
          (XMax c_j)
          (XMin c)))))
  |Comps|)

;  id=50, Author=AE-M, status=0, parent-id=11
;  Optimizations:
f_38:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (XMin c)
          (XMax c_j))
        (-
          (XMin c)
          (XMax c_j)))
      (TRUE
        0.000000)))
  |Comps|)

;  id=51, Author=UEQ, status=0, parent-id=11
;  Optimizations:
f_39:
(/
  (SUM c_j IN Comps
    (IF
      (>
        (XMin c)
        (XMax c_j))
      1
      0))
  |Comps|)

;  id=52, Author=AE-E, status=0, parent-id=12
;  Optimizations:
f_40:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (XMin c_j)
          (XMax c))
        0.000000)
      (TRUE
        (-
          (XMax c)
          (XMin c_j)))))
  |Comps|)

;  id=53, Author=AE-M, status=0, parent-id=12
;  Optimizations:
f_41:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (XMin c_j)
          (XMax c))
        (-
          (XMin c_j)
          (XMax c)))
      (TRUE
        0.000000)))
  |Comps|)

;  id=54, Author=UEQ, status=0, parent-id=12
;  Optimizations:
f_42:
(/
  (SUM c_j IN Comps
    (IF
      (>
        (XMin c_j)
        (XMax c))
      1
      0))
  |Comps|)

;  id=55, Author=AE-E, status=0, parent-id=13
;  Optimizations:
f_43:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (YMin c)
          (YMax c_j))
        0.000000)
      (TRUE
        (-
          (YMax c_j)
          (YMin c)))))
  |Comps|)

;  id=56, Author=AE-M, status=0, parent-id=13
;  Optimizations:
f_44:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (YMin c)
          (YMax c_j))
        (-
          (YMin c)
          (YMax c_j)))
      (TRUE
        0.000000)))
  |Comps|)

;  id=57, Author=UEQ, status=0, parent-id=13
;  Optimizations:
f_45:
(/
  (SUM c_j IN Comps
    (IF
      (>
        (YMin c)
        (YMax c_j))
      1
      0))
  |Comps|)

;  id=58, Author=AE-E, status=0, parent-id=14
;  Optimizations:
f_46:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (YMin c_j)
          (YMax c))
        0.000000)
      (TRUE
        (-
          (YMax c)
          (YMin c_j)))))
  |Comps|)

;  id=59, Author=AE-M, status=0, parent-id=14
;  Optimizations:
f_47:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (YMin c_j)
          (YMax c))
        (-
          (YMin c_j)
          (YMax c)))
      (TRUE
        0.000000)))
  |Comps|)

;  id=60, Author=UEQ, status=0, parent-id=14
;  Optimizations:
f_48:
(/
  (SUM c_j IN Comps
    (IF
      (>
        (YMin c_j)
        (YMax c))
      1
      0))
  |Comps|)

;  id=61, Author=AE-E, status=0, parent-id=17
;  Optimizations:
f_49:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (-
            (XMin c)
            100.000000)
          (XMax c_j))
        0.000000)
      (TRUE
        (-
          (XMax c_j)
          (-
            (XMin c)
            100.000000)))))
  |Comps|)

;  id=62, Author=AE-M, status=0, parent-id=17
;  Optimizations:
f_50:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (-
            (XMin c)
            100.000000)
          (XMax c_j))
        (-
          (-
            (XMin c)
            100.000000)
          (XMax c_j)))
      (TRUE
        0.000000)))
  |Comps|)

;  id=63, Author=UEQ, status=0, parent-id=17
;  Optimizations:
f_51:
(/
  (SUM c_j IN Comps
    (IF
      (>
        (-
          (XMin c)
          100.000000)
        (XMax c_j))
      1
      0))
  |Comps|)

;  id=64, Author=AE-E, status=0, parent-id=18
;  Optimizations:
f_52:
(/
  (SUM c_j IN Comps
    (COND
      ((<
          (+
            (XMax c)
            100.000000)
          (XMin c_j))
        0.000000)
      (TRUE
        (-
          (+
            (XMax c)
            100.000000)
          (XMin c_j)))))
  |Comps|)

;  id=65, Author=AE-M, status=0, parent-id=18
;  Optimizations:
f_53:
(/
  (SUM c_j IN Comps
    (COND
      ((<
          (+
            (XMax c)
            100.000000)
          (XMin c_j))
        (-
          (XMin c_j)
          (+
            (XMax c)
            100.000000)))
      (TRUE
        0.000000)))
  |Comps|)
```

```
;  id=66, Author=UEQ, status=0, parent-id=18
;  Optimizations:
f_54:
(/
  (SUM c_j IN Comps
    (IF
      (<
        (+
          (XMax c)
          100.000000)
        (XMin c_j))
      1
      0))
  |Comps|)

;  id=67, Author=AE-E, status=0, parent-id=19
;  Optimizations:
f_55:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (-
            (YMin c)
            100.000000)
          (YMax c_j))
        0.000000)
      (TRUE
        (-
          (YMax c_j)
          (-
            (YMin c)
            100.000000)))))
  |Comps|)

;  id=68, Author=AE-M, status=0, parent-id=19
;  Optimizations:
f_56:
(/
  (SUM c_j IN Comps
    (COND
      ((>
          (-
            (YMin c)
            100.000000)
          (YMax c_j))
        (-
          (-
            (YMin c)
            100.000000)
          (YMax c_j)))
      (TRUE
        0.000000)))
  |Comps|)

;  id=69, Author=UEQ, status=0, parent-id=19
;  Optimizations:
f_57:
(/
  (SUM c_j IN Comps
    (IF
      (>
        (-
          (YMin c)
          100.000000)
        (YMax c_j))
      1
      0))
  |Comps|)

;  id=70, Author=AE-E, status=0, parent-id=20
;  Optimizations:
f_58:
(/
  (SUM c_j IN Comps
    (COND
      ((<
          (+
            (YMax c)
            100.000000)
          (YMin c_j))
        0.000000)
      (TRUE
        (-
          (+
            (YMax c)
            100.000000)
          (YMin c_j)))))
  |Comps|)
```

```
;  id=71, Author=AE-M, status=0, parent-id=20
;  Optimizations:
f_59:
(/
  (SUM c_j IN Comps
    (COND
      ((<
          (+
            (YMax c)
            100.000000)
          (YMin c_j))
        (-
          (YMin c_j)
          (+
            (YMax c)
            100.000000)))
      (TRUE
        0.000000)))
  |Comps|)

;  id=72, Author=UEQ, status=0, parent-id=20
;  Optimizations:
f_60:
(/
  (SUM c_j IN Comps
    (IF
      (<
        (+
          (YMax c)
          100.000000)
        (YMin c_j))
      1
      0))
  |Comps|)
```

APPENDIX D
OTHELLO SPECIFICATION

This appendix presents the book games and search problem specification that were used in the experiments described in Section 8.

D.1 Book Games

The following are the games used to create the dataset of 10,000 expert Othello preferences.

1. Source: [Rosenbloom, 1982], p284.
   Players: Black: Iago (Rosenbloom), White: The Moor (Levy & O'Connell).
   Outcome: Black, by 51 to 13.
   Moves: d3 c3 c4 c5 d6 e3 f3 e7 b4 a3 e6 f6 d2 c1 e2 f4 f5 f1 c2 f2 a4 a5 b5 a6 b3 c6 e1 d1 b2 a1 b1 d7 d8 b7 c7 a2 a8 g2 g4 b6 h1 a7 g3 h4 g5 g6 f7 h5 h7 h3 h6 h8 h2 e8 f8 g1 g7 g8 c8 b8

2. Source: [Rosenbloom, 1982], p286.
   Players: Black: Iago (Rosenbloom, White: Hiroshi Inoue.
   Outcome: White, by 43 to 21.
   Moves: d3 c3 c4 c5 d6 f4 b3 b4 c6 b5 a3 e2 e3 d2 a4 b6 f1 e1 d1 c1 b1 e7 f5 e6 e8 f6 c2 a5 a6 f3 g4 h3 h5 g2 g5 g3 f7 f8 g8 h6 f2 h4 d8 d7 c7 b8 h1 b7 g6 h2 h7 g1 a8 c8 h8 g7 a7 a1 a2 b2

3. Source: [Rosenbloom, 1982], p291.
   Players: Black: Jonathan Cerf, White: Takuya Mimura.
   Outcome: Black, by 44 to 20
   Moves: f5 d6 c5 f4 e3 d3 e6 g5 c6 f3 g4 f6 c4 c3 d2 c2 f2 e2 g3 e7 h6 f1 b3 h3 h4 d7 d1 e1 c1 b1 c7 b4 a4 a5 a6 b6 b5 d8 h2 a2 a3 a7 g6 h5 g2 b2 f7 f8 e8 h1 g1 g7 a1 h7 a8 b7 c8 b8 g8 h8

4. Source: [Rosenbloom, 1982], p313, Fig. 5.1.
   Players: Black: Takuya Mimura, White: Jonathan Cerf.

Outcome: White, by 43 to 21.
Moves: f5 f6 e6 d6 c5 e3 d3 c4 c6 b5 b4 g5 d7 e7 g6 c7 f8 d2 e2 b6 c3 f4 f3 e8 g4 f1 c1 d1 c2 g3 d8 c8 b3 h4 b8 f7 h5 h3 a4 a6 a5 a3 g8 h6 b2 a1 a2 b1 e1 b7 g7 f2 a8 a7 g2 h2 g1 h1 h8 h7

5. Source: [Levy & Beal, 1989], pp 61-64.
Players: Unknown (Alex Selby's Polygon program is probably one of them).
Outcome: White, by 48 to 16.
Moves: e6 f4 c3 c4 d3 d6 f6 c6 f5 g5 g6 e3 f2 f7 c5 b5 e7 f3 h4 f8 d7 e8 c8 h6 h5 h3 d8 g4 g8 b4 a4 a5 b3 b6 c7 b7 a3 f1 g3 g2 h7 h8 h1 h2 a8 b8 g1 e1 e2 g7 d1 d2 c2 b2 c1 a1 a7 a6 a2 b1

6. Source: [Johnson, 1992]
Players: Black: Kierulf, White: Shaman.
Outcome: White, by 44 to 20
Moves: f5 d6 c3 d3 c4 f4 c5 b4 b5 c6 f3 e6 e3 g6 a5 d2 b6 c2 e2 b3 a3 f1 g5 a6 e1 d1 f7 f6 h5 a4 a7 b2 c7 e7 f8 e8 a1 a2 d8 a8 d7 b7 b8 c8 b1 c1 g1 f2 g3 h3 g2 g8 h2 g4 h4 h1 g7 h8 h7 h6

7. Source: [Johnson, 1992]
Players: Black: Shaman, White: Juhem.
Outcome: Black, by 50 to 14
Moves: f5 d6 c5 f4 e3 c6 d3 f6 e6 d7 g4 c4 g5 c3 f7 d2 e7 f2 e2 f3 c8 g3 f1 h4 h5 e1 c2 e8 b6 b5 a5 b4 h3 g6 d1 h6 g1 h2 a3 b1 c7 a6 d8 a4 a7 b8 g7 h8 h7 g8 f8 b7 a8 g2 b3 a2 h1 b2 a1 c1

8. Source: [Johnson, 1992]
Players: Black: Englund, White: Shaman.
Outcome: White, by 47-17. Moves: f5 f6 e6 f4 e3 c5 c4 e7 b5 e2 f2 d2 f3 g4 g5 d6 f7 g6 h4 h3 h6 c6 g3 d3 c1 f1 e8 b6 c2 c3 e1 d8 c8 d7 b3 b4 a5 f8 g8 a6 a4 b7 c7 d1 g1 h5 h2 a7 g2 a3 h7 h1 a2 a1 a8 h8 g7 b8 b2 b1

9. Source: [Johnson, 1992]
Players: Black: Shaman, White: Ralle.
Outcome: Black, by 35 to 29.
Moves: f5 f6 e6 f4 c3 d6 f3 c5 g4 g3 f7 g6 e7 g5 c4 d7 h6 h4 b6 e3 c6 f8 h5 c7 d3 e8 g7 b3 b4 h8 h3 a6 c2 d2 a4 h7 e1 d1 f2 f1 e2 b5 a5 a3 a2 a1 b2 h2 g2 b1 c1 h1 g1 d8 c8 a7 a8 g8 b7 b8

10. Source: [Johnson, 1992]
Players: Black: Shaman, White: Kaneda.
Outcome: Black, by 35 to 29.
Moves: f5 d6 c4 d3 c5 f4 e3 f3 c2 c6 f6 e6 c7 c3 d7 e7 d2 b4 b3 g4 g6 d8 c8 d1 b5 f7 g5 h5 h3 h6 f2 a6 a5 h4 g3 e2 f1 e1 c1 h2 e8 f8 g8 a4 a3 b6 g7 a2 a7 a8 b2 b8 b7 h8 h7 a1 b1 g1 h1 g2

11. Source: [Johnson, 1992]
    Players: Black: Rose, White: Shaman.
    Outcome: White, by 30 to 34.
    Moves: f5 d6 c3 d3 c4 f4 c5 b4 c6 e6 e3 b5 b6 b3 a6 a5 g5 d7 a3 c7 a4 g6 a7 d2 e2 c2 f7 f6 f3 h4 c8 g4 g3 f8 d8 e8 e7 b8 h5 h6 h7 h8 f2 f1 b1 c1 d1 h3 g8 g7 a8 b7 b2 a1 a2 e1 g2 h1 h2 g1

12. Source: [Johnson, 1992]
    Players: Black: Shaman, White: Melnikov.
    Outcome: Black, by 52 to 12.
    Moves: f5 f6 e6 f4 g6 c5 g4 g5 f3 e3 h5 g3 f2 h4 h3 e7 d8 f7 e8 f1 d6 h7 b5 c6 c7 f8 g2 b6 d7 c4 h6 h1 h8 c8 g8 g7 h2 a4 c3 d3 b7 a8 b8 b3 a6 a7 a5 d2 c1 b4 c2 b2 a3 a1 a2 g1 e2 d1 e1 b1

13. Source: [Johnson, 1992]
    Players: Black: Shaman, White: Feldborg.
    Outcome: Black, by 33 to 30.
    Moves: f5 f6 e6 f4 e3 c5 g5 d6 g6 h6 f7 d3 e7 e8 g4 f8 c7 c6 d8 c8 d7 e2 c4 f3 b5 h5 h3 a5 a6 a7 b6 c3 d2 a4 b3 b4 f1 c1 d1 e1 a3 a2 b1 c2 g3 h2 h1 h4 h7 g7 g2 b2 b7 g1 f2 a1 h8 g8 b8

14. Source: [Johnson, 1992]
    Players: Black: Marconi, White: Shaman.
    Outcome: White, by 20 to 44.
    Moves: f5 f6 e6 f4 e3 c5 g5 g3 g4 f3 g6 d3 f2 h4 h5 h6 c4 c3 d6 e2 c2 c7 d7 e1 h2 d2 d1 h3 h7 c6 b6 b5 d8 e7 b3 b4 a5 c1 b2 a4 f8 a1 g2 e8 g1 f1 f7 c8 b8 h1 g7 h8 g8 a7 a2 a3 a6 a8 b1 b7

15. Source: [Johnson, 1992]
    Players: Black: Shaman, White: Brightwell.
    Outcome: Black, by 36 to 28.
    Moves: f5 d6 c4 d3 c5 f4 e3 f3 c2 c6 e6 d2 g4 b6 b5 c3 b4 c1 f2 a4 a5 a6 d7 b3 b2 g3 h4 f1 a2 g5 f6 h6 h5 c7 c8 e8 d8 e7 f8 f7 h7 h3 h2 a1 a3 g6 g8 g2 h1 g1 e1 e2 g7 h8 b7 a8 b8 a7 d1 b1

16. Source: [Johnson, 1992]
    Players: Black: Feinstein, White: Shaman.
    Outcome: White, by 46 to 18.
    Moves: f5 f6 e6 f4 e3 c5 c4 e7 g4 d3 d6 c6 g5 g6 f7 h5 d8 c3 h4 d7 b5 b4 f3 f8 c8 h3 c7 a5 b6 e8 b7 f2 g3 e2 h6 h7 g2 a6 f1 a8 d1 h2 c2 b3 b2 a1 g7 h1 g1 b8 d2 g8 a4 b1 c1 h8 a2 a3 a7 e1

17. Source: [Johnson, 1992]
    Players: Black: Shaman, White: Stepanov.
    Outcome: Black, by 38 to 26.

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

18. Source: [Lisnovsky, 1992]
Players: Black: Lisnovsky, White: Rockwell.
Outcome: White, by 41 to 23.
Moves: f5 d6 c5 f4 e3 f6 f3 c4 d3 c3 e6 b4 b6 b5 c6 d2 a5 f7 e2 c2 e7 d8 e8 f8 d7 c8 g4 f1 a3 c7 g5 g6 b3 f2 e1 c1 d1 h6 h5 h3 h7 a6 h4 h8 g7 g3 g8 a4 a7 b2 a2 a1 b1 g2 h1 h2 g1 b7 b8 a8

19. Source: [Kling, 1992]
Players: Black: Piau, White: Shaman.
Outcome: White, by 36-28.
Moves: f5 f6 e6 f4 e3 c5 g5 g3 g4 f3 c4 d3 d6 b5 b3 c3 f2 h6 h4 c7 c6 f7 d7 g6 d2 e2 h5 d8 e7 e8 d1 a3 b6 c1 a5 b4 a4 e1 h7 h2 f8 g8 b8 f1 g2 g7 b2 a2 a1 h1 c8 a8 h8 a6 c2 h3 g1 b1 a7 b7

20. Source: [Canuck, 1992]
Players: Black: Harold (Scott), White: Colin Springer.
Outcome: Black, by 34 to 30.
Moves: d3 c5 f6 f5 e6 e3 c3 f3 c4 b4 c6 c2 d2 b3 e2 d1 f4 e1 f2 b5 a5 a6 a7 d6 b6 c7 a4 f7 a3 g6 f1 g5 g4 h3 h4 b2 h5 h6 g3 h2 c8 g1 f8 e7 d7 d8 e8 b7 c1 b1 a1 a2 a8 b8 h1 g2 h7 g7 h8 g8

21. Source: [Canuck, 1992]
Players: Black: Colin Springer, White: Desdemona's Revenge (Hsieh & Springer).
Outcome: Black, by 35 to 29.
Moves: e6 f4 e3 d6 c5 f3 c4 c6 f5 g6 d7 c7 e7 f6 f7 f8 e8 d8 b6 a6 g4 b5 g3 b3 b4 a3 b7 d3 b8 e2 g5 h6 c8 h3 h5 h4 f2 f1 c3 a8 d2 d1 a4 c2 a7 a5 c1 b1 b2 a1 a2 g7 h8 e1 g8 g2 h2 h1 g1 h7

22. Source: [Canuck, 1992]
Players: Black: Harold (Scott), White: Desdemona's Revenge (Hsieh & Springer).
Moves: e6 f6 f5 d6 e7 g5 c5 f4 c4 d7 f7 c6 c8 e8 f8 d8 c7 g6 d3 c3 e3 e2 c2 d2 f1 f2 d1 c1 b1 b3 g1 f3 b4 a3 g3 b5 e1 g4 b6 a5 h4 b7 a4 b2 h7 h6 h5 h2 a7 a6 a8 b8 g7 g2 h3 h8 g8 a1 a2

23. Source: [Canuck, 1992]
Players: Black: Zeus (Barker), White: Harold (Scott)
Outcome: White, by 43 to 21. (The score published in the Othello Quarterly is 53 to 11, which does not match the game history).
Moves: d3 c5 f6 f5 e6 e3 c3 d2 f4 f3 c2 g4 c4 d6 d1 e7 h3 b4 e2 b3 a3 f1 g5 f2

a4 b1 e1 c1 f7 b5 c7 a5 a6 b6 c6 g3 d7 h5 h4 h2 g2 g1 b2 e8 f8 a1 a2 h1 d8 h6 g6 b8 b7 a7 h7 g7 c8 a8 g8 h8

24. Source: [Canuck, 1992]
Players: Black: Rambo (Mathews), White: Jonathan (Rose).
Outcome: White, by 34 to 30.
Moves: f5 f6 e6 f4 e3 c5 c6 d6 e7 d7 e8 c7 b8 f3 c3 e2 g6 d3 g5 b3 b6 c8 d8 a6 g3 g4 f2 f8 g8 f7 h3 h6 h4 f1 d1 h5 h7 d2 c1 c2 b1 c4 a3 g2 b4 b5 a4 a5 a7 h2 h1 b2 a1 a2 e1 a8 g1 b7 h8 g7

25. Source: [Canuck, 1992]
Players: Black: Fugazi (Brockington), White: Desdemona's Revenge (Hsieh & Springer).
Outcome: Black, by 50 to 14.
Moves: f5 f6 e6 f4 g5 e7 f7 h5 g4 g6 e3 f8 g3 c4 h6 h7 d7 d3 c3 d6 e8 h2 c6 b6 b4 c7 c8 d8 f3 f2 e2 e1 d2 c5 d1 c1 b5 a3 g8 b3 f1 g1 a6 a5 b8 a7 a4 h4 a8 b7 h3 g2 h1 g7 h8 a2 b2 b1 a1 c2

26. Source: [Hornstein, 1992]
Players: Black: C. Hewlett, White: J. Merickel.
Outcome: Black, by 42 to 22.
Moves: e6 f6 f5 d6 c3 f4 c6 d3 e3 f2 f3 e2 d1 e1 d7 c1 g3 g5 e7 e8 f8 c8 h4 h3 d2 c2 c4 g6 d8 g8 g4 b3 f7 h5 g1 b4 a3 c7 a5 b5 c5 a6 b6 a4 a7 b2 g2 h1 f1 b7 a1 a2 b1 g7 h6 a8 b8 h7 h2 h8

27. Source: [Hornstein, 1992]
Players: Black: P. Stanton, White: B. Rose.
Outcome: White, by 34 to 30.
Moves: f5 d6 c4 d3 c5 f4 d2 f6 d7 c7 e7 d8 f3 e3 g4 g5 f8 h3 e2 f2 b8 e1 f1 c1 d1 g1 e6 c3 g3 h4 g6 h7 b2 b3 a2 b4 a4 g7 f7 c6 h8 c8 e8 g8 h6 h5 c2 a8 h2 a3 a1 g2 h1 b1 b6 b7 a7 a6 b5 a5

28. Source: [Hornstein, 1992]
Players: Black: G. Johnson, White: P. Stanton.
Outcome: Black, by 49 to 14.
Moves: f5 f6 e6 f4 e3 c5 g5 g3 g4 d6 f7 f3 d3 f2 e2 h4 h5 h6 e7 c3 f1 d1 c2 d2 c7 g6 e1 g1 c1 b1 b5 f8 c4 a6 b3 e8 g7 d7 c6 b6 a5 a3 a7 h8 g8 c8 h3 a4 b4 a8 b8 b7 d8 h2 h7 h1 a1 g2 b2

D.2 Search Problem Specification

The search problem specification presented here is a modified version of an OTHELLO problem specification, written by Tom Fawcett, that is available through the Machine Learning Database at the University of California, Irvine (Internet node ics.uci.edu).

Tom Fawcett's OTHELLO problem specification was written in the Prolog programming language. The CINDI program for performing knowledge-based feature generation processes problem specifications written in an extension[1] of First Order Predicate Calculus. The conversion from Prolog to extended First Order Predicate Calculus consisted of the following steps.

- Convert implicit existential quantification into explicit existential quantification.
- Convert all rules with a common left-hand-side into a single predicate defined as the conjunction of the right-hand-sides of those rules.
- One-for-one syntactic conversions (e.g., replace ',' with '$\wedge$').

This conversion was done manually. Care was taken not to change the semantics of the original OTHELLO problem specification.

Two changes were made that were not strictly syntactic conversions. The first was the introduction of the predicate **blank**, which was added to make two parts of the problem specification easier to read. The effect of **blank** on feature generation was minimal. The second change was the splitting of the `move` operator into `move_X` and `move_O` operators, which was done to make the operators easier to understand. If `move` had not been split into `move_X` and `move_O`, CINDI would have generated a few more useless features for Othello. The set of useful features would have remained unchanged.

The extended First Order Predicate Calculus specification of OTHELLO used in the experiments described in Section 8 follows.

```
INITIAL:
    Players := {X, O}
    Mover := {X}
    Directions := {n, ne, e, se, s, sw, w, nw}
    Squares := {a1,  a2,  a3,  a4,  a5,  a6,  a7,  a8,
                b1,  b2,  b3,  b4,  b5,  b6,  b7,  b8,
                c1,  c2,  c3,  c4,  c5,  c6,  c7,  c8,
                d1,  d2,  d3,  d4,  d5,  d6,  d7,  d8,
                e1,  e2,  e3,  e4,  e5,  e6,  e7,  e8,
                f1,  f2,  f3,  f4,  f5,  f6,  f7,  f8,
                g1,  g2,  g3,  g4,  g5,  g6,  g7,  g8,
                h1,  h2,  h3,  h4,  h5,  h6,  h7,  h8}
    Ownership := { (X, d4), (O, e4), (O, d5), (X, e5) }
```

[1]The extension to First Order Predicate Calculus was a set of conventions for identifying the different parts of a search problem specification (e.g., specification of the initial state, etc).

```
GOAL:
    win (X)

OPERATORS:
    move_X:
        condition:
            (turn_to_move (X) and
             exists s1 in Squares,
                legal_move (s1, X))

        action:
            forall s2 in Squares,
                (not bs (s1, s2, X) or
                 forall flipsq in Squares,
                    (not in_line (flipsq, s1, s2) or
                     set_owns (X, flipsq)))
    move_O:
        condition:
            (turn_to_move (O) and
             exists s1 in Squares,
                legal_move (s1, O))

        action:
            forall s2 in Squares,
                (not bs (s1, s2, O) or
                 forall flipsq in Squares,
                    (not in_line (flipsq, s1, s2) or
                     set_owns (O, flipsq)))

PREDICATES:
    win (P) :
        end_of_game() and
        exists opp in Players,
            (opponent (P, opp) and
             score (P) > score (opp))

    end_of_game () :
        (not exists s1 in Squares, legal_move (s1, X)) and
        (not exists s1 in Squares, legal_move (s1, O))

    legal_move (square, player) :
        blank (square) and
        exists s3 in Squares, bs (square, s3, player)

    bs (s1, s3, p) :
        blank (s1) and
        exists opp in Players,
            (opponent (p, opp) and
             exists d in Directions,
                exists s2 in Squares,
                    (neighbor (s1, d, s2) and
                     span (s2, s3, d, opp) and
                     exists s4 in Squares,
                        (neighbor (s3, d, s4) and
                         owns (p, s4))))

    span (s1, s2, d, owner) :
        (owns (owner, s1) and
         (s1 == s2)) or
        (owns (owner, s1) and
         exists s3 in Squares,
            (neighbor (s1, d, s3) and
             span (s3, s2, d, owner)))

    blank (s) :
        not exists p in Players,
            owns (p, s)

    in_line (s, start, end) :
        exists d in Directions,
            (line (start, s, d) and
             line (s, end, d))

    line (from, to, dir) :
        (from == to) or
        exists next in Squares,
            (neighbor (from, dir, next) and
             line (next, to, dir))

RELATIONS:
    opponent:
        { (X,O), (O,X) }

    neighbor:
        { (a1,e,b1), (a1,se,b2), (a1,s,a2), (b1,e,c1),
          (b1,se,c2), (b1,s,b2),  (b1,sw,a2), (b1,w,a1),
          (c1,e,d1),  (c1,se,d2), (c1,s,c2), (c1,sw,b2),
          (c1,w,b1), (d1,e,e1), (d1,se,e2),  (d1,s,d2),
          (d1,sw,c2), (d1,w,c1),  (e1,e,f1), (e1,se,f2),
          (e1,s,e2),  (e1,sw,d2), (e1,w,d1), (f1,e,g1),
          (f1,se,g2), (f1,s,f2), (f1,sw,e2),  (f1,w,e1),
          (g1,e,h1), (g1,se,h2),  (g1,s,g2), (g1,sw,f2),
(g1,w,f1),  (h1,s,h2), (h1,sw,g2), (h1,w,g1),
(a2,n,a1), (a2,ne,b1), (a2,e,b2),  (a2,se,b3),
(a2,s,a3), (b2,n,b1),  (b2,ne,c1), (b2,e,c2),
(b2,se,c3),  (b2,s,b3), (b2,sw,a3), (b2,w,a2),
(b2,nw,a1), (c2,n,c1), (c2,ne,d1),  (c2,e,d2),
(c2,se,d3), (c2,s,c3),  (c2,sw,b3), (c2,w,b2),
(c2,nw,b1),  (d2,n,d1), (d2,ne,e1), (d2,e,e2),
(d2,se,e3), (d2,s,d3), (d2,sw,c3),  (d2,w,c2),
(d2,nw,c1), (e2,n,e1),  (e2,ne,f1), (e2,e,f2),
(e2,se,f3),  (e2,s,e3), (e2,sw,d3), (e2,w,d2),
(e2,nw,d1), (f2,n,f1), (f2,ne,g1),  (f2,e,g2),
(f2,se,g3), (f2,s,f3),  (f2,sw,e3), (f2,w,e2),
(f2,nw,e1),  (g2,n,g1), (g2,ne,h1), (g2,e,h2),
(g2,se,h3), (g2,s,g3), (g2,sw,f3),  (g2,w,f2),
(g2,nw,f1), (h2,n,h1),  (h2,s,h3), (h2,sw,g3),
(h2,w,g2),  (h2,nw,g1), (a3,n,a2), (a3,ne,b2),
(a3,e,b3), (a3,se,b4), (a3,s,a4),  (b3,n,b2),
(b3,ne,c2), (b3,e,c3),  (b3,se,c4), (b3,s,b4),
(b3,sw,a4),  (b3,w,a3), (b3,nw,a2), (c3,n,c2),
(c3,ne,d2), (c3,e,d3), (c3,se,d4),  (c3,s,c4),
(c3,sw,b4), (c3,w,b3),  (c3,nw,b2), (d3,n,d2),
(d3,ne,e2),  (d3,e,e3), (d3,se,e4), (d3,s,d4),
(d3,sw,c4), (d3,w,c3), (d3,nw,c2),  (e3,n,e2),
(e3,ne,f2), (e3,e,f3),  (e3,se,f4), (e3,s,e4),
(e3,sw,d4),  (e3,w,d3), (e3,nw,d2), (f3,n,f2),
(f3,ne,g2), (f3,e,g3), (f3,se,g4),  (f3,s,f4),
(f3,sw,e4), (f3,w,e3),  (f3,nw,e2), (g3,n,g2),
(g3,ne,h2),  (g3,e,h3), (g3,se,h4), (g3,s,g4),
(g3,sw,f4), (g3,w,f3), (g3,nw,f2),  (h3,n,h2),
(h3,s,h4), (h3,sw,g4),  (h3,w,g3), (h3,nw,g2),
(a4,n,a3),  (a4,ne,b3), (a4,e,b4), (a4,se,b5),
(a4,s,a5), (b4,n,b3), (b4,ne,c3),  (b4,e,c4),
(b4,se,c5), (b4,s,b5),  (b4,sw,a5), (b4,w,a4),
(b4,nw,a3),  (c4,n,c3), (c4,ne,d3), (c4,e,d4),
(c4,se,d5), (c4,s,c5), (c4,sw,b5),  (c4,w,b4),
(c4,nw,b3), (d4,n,d3),  (d4,ne,e3), (d4,e,e4),
(d4,se,e5),  (d4,s,d5), (d4,sw,c5), (d4,w,c4),
(d4,nw,c3), (e4,n,e3), (e4,ne,f3),  (e4,e,f4),
(e4,se,f5), (e4,s,e5),  (e4,sw,d5), (e4,w,d4),
(e4,nw,d3),  (f4,n,f3), (f4,ne,g3), (f4,e,g4),
(f4,se,g5), (f4,s,f5), (f4,sw,e5),  (f4,w,e4),
(f4,nw,e3), (g4,n,g3),  (g4,ne,h3), (g4,e,h4),
(g4,se,h5),  (g4,s,g5), (g4,sw,f5), (g4,w,f4),
(g4,nw,f3), (h4,n,h3), (h4,s,h5),  (h4,sw,g5),
(h4,w,g4), (h4,nw,g3),  (a5,n,a4), (a5,ne,b4),
(a5,e,b5), (a5,se,b6), (a5,s,a6), (b5,n,b4),
(b5,ne,c4), (b5,e,c5), (b5,se,c6), (b5,s,b6),
(b5,sw,a6), (b5,w,a5), (b5,nw,a4), (c5,n,c4),
(c5,ne,d4), (c5,e,d5), (c5,se,d6), (c5,s,c6),
(c5,sw,b6), (c5,w,b5), (c5,nw,b4), (d5,n,d4),
(d5,ne,e4), (d5,e,e5), (d5,se,e6), (d5,s,d6),
(d5,sw,c6), (d5,w,c5), (d5,nw,c4), (e5,n,e4),
(e5,ne,f4), (e5,e,f5), (e5,se,f6), (e5,s,e6),
(e5,sw,d6), (e5,w,d5), (e5,nw,d4), (f5,n,f4),
(f5,ne,g4), (f5,e,g5), (f5,se,g6), (f5,s,f6),
(f5,sw,e6), (f5,w,e5), (f5,nw,e4), (g5,n,g4),
(g5,ne,h4), (g5,e,h5), (g5,se,h6), (g5,s,g6),
(g5,sw,f6), (g5,w,f5), (g5,nw,f4), (h5,n,h4),
(h5,s,h6), (h5,sw,g6), (h5,w,g5), (h5,nw,g4),
(a6,n,a5), (a6,ne,b5), (a6,e,b6), (a6,se,b7),
(a6,s,a7), (b6,n,b5), (b6,ne,c5), (b6,e,c6),
(b6,se,c7), (b6,s,b7), (b6,sw,a7), (b6,w,a6),
(b6,nw,a5), (c6,n,c5), (c6,ne,d5), (c6,e,d6),
(c6,se,d7), (c6,s,c7), (c6,sw,b7), (c6,w,b6),
(c6,nw,b5), (d6,n,d5), (d6,ne,e5), (d6,e,e6),
(d6,se,e7), (d6,s,d7), (d6,sw,c7), (d6,w,c6),
(d6,nw,c5), (e6,n,e5), (e6,ne,f5), (e6,e,f6),
(e6,se,f7), (e6,s,e7), (e6,sw,d7), (e6,w,d6),
(e6,nw,d5), (f6,n,f5), (f6,ne,g5), (f6,e,g6),
(f6,se,g7), (f6,s,f7), (f6,sw,e7), (f6,w,e6),
(f6,nw,e5), (g6,n,g5), (g6,ne,h5), (g6,e,h6),
(g6,se,h7), (g6,s,g7), (g6,sw,f7), (g6,w,f6),
(g6,nw,f5), (h6,n,h5), (h6,s,h7), (h6,sw,g7),
(h6,w,g6), (h6,nw,g5), (a7,n,a6), (a7,ne,b6),
(a7,e,b7), (a7,se,b8), (a7,s,a8), (b7,n,b6),
(b7,ne,c6), (b7,e,c7), (b7,se,c8), (b7,s,b8),
(b7,sw,a8), (b7,w,a7), (b7,nw,a6), (c7,n,c6),
(c7,ne,d6), (c7,e,d7), (c7,se,d8), (c7,s,c8),
(c7,sw,b8), (c7,w,b7), (c7,nw,b6), (d7,n,d6),
(d7,ne,e6), (d7,e,e7), (d7,se,e8), (d7,s,d8),
(d7,sw,c8), (d7,w,c7), (d7,nw,c6), (e7,n,e6),
(e7,ne,f6), (e7,e,f7), (e7,se,f8), (e7,s,e8),
(e7,sw,d8), (e7,w,d7), (e7,nw,d6), (f7,n,f6),
(f7,ne,g6), (f7,e,g7), (f7,se,g8), (f7,s,f8),
(f7,sw,e8), (f7,w,e7), (f7,nw,e6), (g7,n,g6),
(g7,ne,h6), (g7,e,h7), (g7,se,h8), (g7,s,g8),
(g7,sw,f8), (g7,w,f7), (g7,nw,f6), (h7,n,h6),
(h7,s,h8), (h7,sw,g8), (h7,w,g7), (h7,nw,g6),
(a8,n,a7), (a8,ne,b7), (a8,e,b8), (b8,n,b7),
(b8,ne,c7), (b8,e,c8), (b8,w,a8), (b8,nw,a7),
(c8,n,c7),  (c8,ne,d7), (c8,e,d8), (c8,w,b8),
(c8,nw,b7), (d8,n,d7), (d8,ne,e7),  (d8,e,e8),
(d8,w,c8), (d8,nw,c7),  (e8,n,e7), (e8,ne,f7),
(e8,e,f8),  (e8,w,d8), (e8,nw,d7), (f8,n,f7),
(f8,ne,g7), (f8,e,g8), (f8,w,e8),  (f8,nw,e7),
(g8,n,g7), (g8,ne,h7),  (g8,e,h8), (g8,w,f8),
(g8,nw,f7),  (h8,n,h7), (h8,w,g8), (h8,nw,g7) }
```

D.3 Features Generated

The following is the output generated by the CINDI program when it created features for Othello. The cost estimates have been eliminated because they are easy to recreate. See the previous appendices for examples of cost estimates produced by CINDI. All input to the program is shown below, except for the problem specification, which is shown above.

```
                 CINDI version 1.24

Is owns a static (constant) relation [y|n]?  n
Is opponent a static (constant) relation [y|n]?  y
Is score a static (constant) relation [y|n]?  n
Is neighbor a static (constant) relation [y|n]?  y
Is turn_to_move a static (constant) relation [y|n]?  y

50 features were created (LE=26, AE=4, UQ=20, Pruned=106).

Please select an output format for the features:
  c:  the C programming language.
  f:  a generic functional notation.
f

;  id=0, Author=LE, status=0, parent-id=user
;  Optimizations:
f_1:
(end_of_game)

;  id=1, Author=LE, status=0, parent-id=user
;  Optimizations:
f_2:
(>
  (score X)
  (score O))
```

```
;  id=2, Author=LE, status=0, parent-id=0
;  Optimizations:
f_3:
(! (EXISTS s1 in Squares
    (legal_move s1 X)))

;  id=3, Author=LE, status=0, parent-id=0
;  Optimizations:
f_4:
(! (EXISTS s1 in Squares
    (legal_move s1 O)))

;  id=4, Author=LE, status=0, parent-id=2
;  Optimizations:  3
f_5:
(EXISTS s1 in Squares
  (legal_move s1 X))

;  id=5, Author=LE, status=0, parent-id=3
;  Optimizations:  3
f_6:
(EXISTS s1 in Squares
  (legal_move s1 O))

;  id=6, Author=LE, status=0, parent-id=4
;  Optimizations:  3
f_7:
(EXISTS s1 in Squares
  (blank s1))

;  id=7, Author=LE, status=0, parent-id=4
;  Optimizations:  3
f_8:
(EXISTS s1 in Squares
  (EXISTS s3 in Squares
    (bs s1 s3 X)))

;  id=9, Author=LE, status=0, parent-id=5
;  Optimizations:  3
f_9:
(EXISTS s1 in Squares
  (EXISTS s3 in Squares
    (bs s1 s3 O)))

;  id=10, Author=LE, status=0, parent-id=6
;  Optimizations:  3
f_10:
(EXISTS s1 in Squares
  (EXISTS p in Players
    (owns p s1)))

;  id=12, Author=LE, status=0, parent-id=7
;  Optimizations:  1 3 4 6
f_11:
(EXISTS d in Directions
  (EXISTS s3 in Squares
    (AND
      (EXISTS s4 in Squares
        (AND
          (neighbor s3 d s4)
          (owns X s4)))
      (EXISTS s2 in Squares
        (AND
          (span s2 s3 d O)
          (EXISTS s1 in Squares
            (neighbor s1 d s2)))))))

;  id=15, Author=LE, status=0, parent-id=9
;  Optimizations:  1 3 4 6
f_12:
(EXISTS d in Directions
  (EXISTS s3 in Squares
    (AND
      (EXISTS s4 in Squares
        (AND
          (neighbor s3 d s4)
          (owns O s4)))
      (EXISTS s2 in Squares
        (AND
          (span s2 s3 d X)
          (EXISTS s1 in Squares
            (neighbor s1 d s2)))))))

;  id=18, Author=LE, status=0, parent-id=12
;  Optimizations:  3 6
f_13:
(EXISTS s3 in Squares
  (EXISTS d in Directions
    (EXISTS s2 in Squares
      (span s2 s3 d O))))
```

```
;  id=19, Author=LE, status=0, parent-id=12
;  Optimizations:  1 3 4 6
f_14:
(EXISTS s4 in Squares
  (AND
    (owns X s4)
    (EXISTS d in Directions
      (EXISTS s3 in Squares
        (neighbor s3 d s4)))))

;  id=22, Author=LE, status=0, parent-id=15
;  Optimizations:  3 6
f_15:
(EXISTS s3 in Squares
  (EXISTS d in Directions
    (EXISTS s2 in Squares
      (span s2 s3 d X))))

;  id=23, Author=LE, status=0, parent-id=15
;  Optimizations:  1 3 4 6
f_16:
(EXISTS s4 in Squares
  (AND
    (owns O s4)
    (EXISTS d in Directions
      (EXISTS s3 in Squares
        (neighbor s3 d s4)))))

;  id=24, Author=LE, status=0, parent-id=18
;  Optimizations:  1 3 4 6
f_17:
(EXISTS s2 in Squares
  (AND
    (owns O s2)
    (EXISTS s3 in Squares
      (==
        s2
        s3))))

;  id=25, Author=LE, status=0, parent-id=18
;  Optimizations:  1 3 4 6
f_18:
(EXISTS s2 in Squares
  (AND
    (owns O s2)
    (EXISTS s3 in Squares
      (EXISTS d in Directions
        (EXISTS s3__1 in Squares
          (AND
            (neighbor s2 d s3__1)
            (span s3__1 s3 d O)))))))

;  id=27, Author=LE, status=0, parent-id=19
;  Optimizations:  3 6
f_19:
(EXISTS s4 in Squares
  (owns X s4))

;  id=28, Author=LE, status=0, parent-id=22
;  Optimizations:  1 3 4 6
f_20:
(EXISTS s2 in Squares
  (AND
    (owns X s2)
    (EXISTS s3 in Squares
      (==
        s2
        s3))))

;  id=29, Author=LE, status=0, parent-id=22
;  Optimizations:  1 3 4 6
f_21:
(EXISTS s2 in Squares
  (AND
    (owns X s2)
    (EXISTS s3 in Squares
      (EXISTS d in Directions
        (EXISTS s3__1 in Squares
          (AND
            (neighbor s2 d s3__1)
            (span s3__1 s3 d X)))))))

;  id=31, Author=LE, status=0, parent-id=23
;  Optimizations:  3 6
f_22:
(EXISTS s4 in Squares
  (owns O s4))

;  id=35, Author=LE, status=0, parent-id=25
;  Optimizations:  1 3 4 6
f_23:
(EXISTS d in Directions
  (EXISTS s3__1 in Squares
    (AND
      (EXISTS s2 in Squares
        (neighbor s2 d s3__1))
      (EXISTS s3 in Squares
        (span s3__1 s3 d O)))))

;  id=39, Author=LE, status=0, parent-id=29
;  Optimizations:  1 3 4 6
f_24:
(EXISTS d in Directions
  (EXISTS s3__1 in Squares
    (AND
      (EXISTS s2 in Squares
        (neighbor s2 d s3__1))
      (EXISTS s3 in Squares
        (span s3__1 s3 d X)))))

;  id=44, Author=AE-E, status=0, parent-id=user
;  Optimizations:  7
f_25:
(COND
  ((AND
      (end_of_game)
      (>
        (score X)
        (score O)))
    0.000000)
  (TRUE
    (-
      (score O)
      (score X))))

;  id=45, Author=AE-M, status=0, parent-id=user
;  Optimizations:  7
f_26:
(COND
  ((AND
      (end_of_game)
      (>
        (score X)
        (score O)))
    (-
      (score X)
      (score O)))
  (TRUE
    0.000000))

;  id=46, Author=AE-E, status=0, parent-id=1
;  Optimizations:
f_27:
(COND
  ((>
      (score X)
      (score O))
    0.000000)
  (TRUE
    (-
      (score O)
      (score X))))

;  id=47, Author=AE-M, status=0, parent-id=1
;  Optimizations:
f_28:
(COND
  ((>
      (score X)
      (score O))
    (-
      (score X)
      (score O)))
  (TRUE
    0.000000))

;  id=48, Author=UEQ, status=0, parent-id=4
;  Optimizations:  3
f_29:
(/
  (SUM s1 IN Squares
    (IF
      (legal_move s1 X)
      1
      0))
  |Squares|)

;  id=49, Author=UEQ, status=0, parent-id=5
;  Optimizations:  3
```

```
f_30:
(/
  (SUM s1 IN Squares
    (IF
      (legal_move s1 O)
      1
      0))
  |Squares|)

;  id=50, Author=UEQ, status=0, parent-id=6
;  Optimizations:  3
f_31:
(/
  (SUM s1 IN Squares
    (IF
      (blank s1)
      1
      0))
  |Squares|)

;  id=51, Author=UEQ, status=0, parent-id=7
;  Optimizations:  3
f_32:
(/
  (SUM s1 IN Squares
    (SUM s3 IN Squares
      (IF
        (bs s1 s3 X)
        1
        0)))
  (*
    |Squares|
    |Squares|))

;  id=53, Author=UEQ, status=0, parent-id=9
;  Optimizations:  3
f_33:
(/
  (SUM s1 IN Squares
    (SUM s3 IN Squares
      (IF
        (bs s1 s3 O)
        1
        0)))
  (*
    |Squares|
    |Squares|))

;  id=54, Author=UEQ, status=0, parent-id=10
;  Optimizations:  3
f_34:
(/
  (SUM s1 IN Squares
    (SUM p IN Players
      (IF
        (owns p s1)
        1
        0)))
  (*
    |Squares|
    |Players|))

;  id=56, Author=UEQ, status=0, parent-id=12
;  Optimizations:  0 3 4
f_35:
(/
  (SUM d IN Directions
    (SUM s3 IN Squares
      (COND
        ((! (EXISTS s4 in Squares
              (AND
                (neighbor s3 d s4)
                (owns X s4))))
          0.000000)
        (TRUE
          (SUM s2 IN Squares
            (COND
              ((! (span s2 s3 d 0))
                0.000000)
              (TRUE
                (SUM s1 IN Squares
                  (IF
                    (neighbor s1 d s2)
                    1
                    0)))))))))
  (*
    |Directions|
    (*
      |Squares|
      (*
        |Squares|
        |Squares|))))

;  id=59, Author=UEQ, status=0, parent-id=15
;  Optimizations:  0 3 4
f_36:
(/
  (SUM d IN Directions
    (SUM s3 IN Squares
      (COND
        ((! (EXISTS s4 in Squares
              (AND
                (neighbor s3 d s4)
                (owns O s4))))
          0.000000)
        (TRUE
          (SUM s2 IN Squares
            (COND
              ((! (span s2 s3 d X))
                0.000000)
              (TRUE
                (SUM s1 IN Squares
                  (IF
                    (neighbor s1 d s2)
                    1
                    0)))))))))
  (*
    |Directions|
    (*
      |Squares|
      (*
        |Squares|
        |Squares|))))

;  id=61, Author=UEQ, status=0, parent-id=18
;  Optimizations:  3
f_37:
(/
  (SUM s3 IN Squares
    (SUM d IN Directions
      (SUM s2 IN Squares
        (IF
          (span s2 s3 d O)
          1
          0))))
  (*
    |Squares|
    (*
      |Directions|
      |Squares|)))

;  id=62, Author=UEQ, status=0, parent-id=19
;  Optimizations:  0 3 4
f_38:
(/
  (SUM s4 IN Squares
    (COND
      ((! (owns X s4))
        0.000000)
      (TRUE
        (SUM d IN Directions
          (SUM s3 IN Squares
            (IF
              (neighbor s3 d s4)
              1
              0))))))
  (*
    |Squares|
    (*
      |Directions|
      |Squares|)))

;  id=64, Author=UEQ, status=0, parent-id=22
;  Optimizations:  3
f_39:
(/
  (SUM s3 IN Squares
    (SUM d IN Directions
      (SUM s2 IN Squares
        (IF
          (span s2 s3 d X)
          1
          0))))
  (*
    |Squares|
    (*
      |Directions|
      |Squares|)))

;  id=65, Author=UEQ, status=0, parent-id=23
;  Optimizations:  0 3 4
f_40:
(/
  (SUM s4 IN Squares
    (COND
      ((! (owns O s4))
        0.000000)
      (TRUE
        (SUM d IN Directions
          (SUM s3 IN Squares
            (IF
              (neighbor s3 d s4)
              1
              0))))))
  (*
    |Squares|
    (*
      |Directions|
      |Squares|)))

;  id=66, Author=UEQ, status=0, parent-id=24
;  Optimizations:  0 3 4
f_41:
(/
  (SUM s2 IN Squares
    (COND
      ((! (owns O s2))
        0.000000)
      (TRUE
        (SUM s3 IN Squares
          (IF
            (==
              s2
              s3)
            1
            0)))))
  (*
    |Squares|
    |Squares|))

;  id=67, Author=UEQ, status=0, parent-id=25
;  Optimizations:  0 3 4
f_42:
(/
  (SUM s2 IN Squares
    (COND
      ((! (owns O s2))
        0.000000)
      (TRUE
        (SUM s3 IN Squares
          (SUM d IN Directions
            (IF
              (EXISTS s3__1 in Squares
                (AND
                  (neighbor s2 d s3__1)
                  (span s3__1 s3 d O)))
              1
              0))))))
  (*
    |Squares|
    (*
      |Squares|
      |Directions|)))

;  id=68, Author=UEQ, status=0, parent-id=27
;  Optimizations:  3
f_43:
(/
  (SUM s4 IN Squares
    (IF
      (owns X s4)
      1
      0))
  |Squares|)

;  id=69, Author=UEQ, status=0, parent-id=28
;  Optimizations:  0 3 4
f_44:
(/
  (SUM s2 IN Squares
    (COND
      ((! (owns X s2))
        0.000000)
      (TRUE
        (SUM s3 IN Squares
          (IF
            (==
              s2
              s3)
            1
            0)))))
  (*
    |Squares|
    |Squares|))

;  id=70, Author=UEQ, status=0, parent-id=29
;  Optimizations:  0 3 4
f_45:
(/
  (SUM s2 IN Squares
    (COND
      ((! (owns X s2))
        0.000000)
      (TRUE
        (SUM s3 IN Squares
          (SUM d IN Directions
            (IF
              (EXISTS s3__1 in Squares
                (AND
                  (neighbor s2 d s3__1)
                  (span s3__1 s3 d X)))
              1
              0))))))
  (*
    |Squares|
    (*
      |Squares|
      |Directions|)))

;  id=71, Author=UEQ, status=0, parent-id=31
;  Optimizations:  3
f_46:
(/
  (SUM s4 IN Squares
    (IF
      (owns O s4)
      1
      0))
  |Squares|)

;  id=74, Author=UEQ, status=0, parent-id=35
;  Optimizations:  0 3 4
f_47:
(/
  (SUM d IN Directions
    (SUM s3__1 IN Squares
      (SUM s3 IN Squares
        (COND
          ((! (span s3__1 s3 d O))
            0.000000)
          (TRUE
            (SUM s2 IN Squares
              (IF
                (neighbor s2 d s3__1)
                1
                0)))))))
  (*
    |Directions|
    (*
      |Squares|
      (*
        |Squares|
        |Squares|))))

;  id=77, Author=UEQ, status=0, parent-id=39
;  Optimizations:  0 3 4
f_48:
(/
  (SUM d IN Directions
    (SUM s3__1 IN Squares
      (SUM s3 IN Squares
        (COND
          ((! (span s3__1 s3 d X))
            0.000000)
          (TRUE
            (SUM s2 IN Squares
              (IF
                (neighbor s2 d s3__1)
                1
                0)))))))
  (*
    |Directions|
    (*
      |Squares|
      (*
        |Squares|
        |Squares|))))

;  id=80, Author=LE, status=0, parent-id=user
;  Optimizations:
```

```
f_49:
(AND
  (turn_to_move X)
  (EXISTS s1 in Squares
    (legal_move s1 X)))

;  id=118, Author=LE, status=0, parent-id=user
;  Optimizations:
f_50:
(AND
  (turn_to_move O)
  (EXISTS s1 in Squares
    (legal_move s1 O)))
```

Bibliography

Aha, D. W. (1991). Incremental constructive induction: An instance-based approach. *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 117-121). Evanston, IL: Morgan Kaufmann.

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers, principles, techniques, and tools.* Reading, MA: Addison-Wesley.

Berliner, H. J. (1984). Search vs knowledge: An analysis from the domain of games. In Elithorn & Banerji (Eds.), *Artificial and Human Intelligence.* New York: Elsevier Science Publishers.

Breuer, M. A., Friedman, A. D., & Iosupovicz, A. (1981). A survey of the state of the art of design automation. *IEEE Transactions on Computers*, 58-75.

Brodley, C. E., & Utgoff, P. E. (1992). *Multivariate versus univariate decision trees*, (Coins Technical Report 92-8), Amherst, MA: University of Massachusetts, Department of Computer and Information Science.

Callan, J. P. (1989). Knowledge-based feature generation. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 441-443). Ithaca, NY: Morgan Kaufmann.

Callan, J. P. (1990). *Use of domain knowledge in constructive induction*, (COINS Technical Report 90-95), Amherst, MA: University of Massachusetts, Department of Computer and Information Science.

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., Fawcett, T. E., & Rissland, E. L. (1991). CABOT: An adaptive approach to case-based search. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence* (pp. 803-808). Sidney, Australia: Morgan Kaufmann.

Canuck, I. (1992). 1991 Waterloo computer open. *Othello Quarterly, 14*, 10-11.

Dechter, R., & Pearl, J. (1987). Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence, 34*, 1-38.

Detrano,R., Janosi,A., Steinbrunn,W., Pfisterer, M., Schmid, J., Sandhu, S., Guppy, K., Lee, S., & Froelicher, V. (1989). International application of a new probability algorithm for the diagnosis of coronary artery disese. *American Hournal of Cardiology, 64*, 304-310.

Devore, J. L. (1991). *Probability and statistics for engineering and the sciences.* Belmont, CA: Wadsworth.

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.

Enderton, H.B. (1972). *A mathematical introduction to logic.* New York: Academic Press.

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.

Fisher, R. A. (1936). Multiple measures in taxonomic problems. *Annals of Eugenics, 7*, 179-188.

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.

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.

Guida, G., & Somalvico, M. (1979). A method of computing heuristics in problem solving. *Information Sciences, 19*, 251-259.

Gunsch, G. (1991). Opportunistic constructive induction: Using fragments of domain knowledge to guide constructive induction. *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 147-152). Evanston, IL: Morgan Kaufmann.

Hornstein, J. M. (1992). The 1992 D.C. open. *Othello Quarterly, 14*, 16.

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.

Johnson, G. (1992). Perfect play. *Othello Quarterly, 14*, 3-5.

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.

Kittler, J. (1986). Feature selection and extraction. In Young & Fu (Eds.), *Handbook of pattern recognition and image processing.* New York: Academic Press.

Kling, A. (1992). Piau vs. Shaman. *Othello Quarterly, 14*, 8-11.

Langley, P., Bradshaw, G., & Simon, H. (1983). Rediscovering Chemistry with the BACON system. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Langley, P. (1988). Machine learning as an experimental science. *Machine Learning, 3*, 5-8.

Lee, K. F., & Mahajan, S. (1988). A pattern classification approach to evaluation function learning. *Artificial Intelligence, 36*, 1-25.

Lenat, D. B., & Brown, J. S. (1984). Why AM and EURISKO appear to work. *Artificial Intelligence, 23*, 269-294.

Levy, D. N. L., & Beal, D. F. (1989). *Heuristic programming in Artificial Intelligence: The first computer olympiad.* Halsted Press.

Lisnovsky, M. (1992). Othello from Argentina. *Othello Quarterly, 14*, 6-7.

Matheus, C. J., & Rendell, L. A. (1989). 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. (1990a). Adding domain knowledge to SBL through feature construction. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 803-808). Boston, MA: Morgan Kaufmann.

Matheus, C. J. (1990b). *Feature construction: An analytic framework and an application to decision trees.* Doctoral dissertation, Department of Computer Science, University of Illinois, Urbana-Champaign, IL.

Mendelson, E. (1979). *Introduction to mathematical logic.* New York: D. Van Nostrand.

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.

Minsky, M., & Papert, S. (1972). *Perceptrons: An introduction to computational geometry (expanded edition).* Cambridge, MA: MIT Press.

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

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., & Prieditis, A. E. (1989). Discovering admissible heuristics by abstracting and optimizing: A transformational approach. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 701-707). Detroit, Michigan: Morgan Kaufmann.

Murphy, P., & Pazzani, M. (1991). ID2-of-3: Constructive induction of *m-of-n* concepts for discriminators in decision trees. *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 183-187). Evanston, IL: Morgan Kaufmann.

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, NJ: Prentice-Hall.

Nilsson, N. J. (1965). *Learning machines.* New York: McGraw-Hill.

Nilsson, N. J. (1980). *Principles of artificial intelligence.* Palo Alto, CA: Tioga.

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.

Pearl, J. (1984). *Heuristics: Intelligent search strategies for computer problem solving.* Reading, Ma: Addison-Wesley.

Preas, B. T., & Karger, P. G. (1986). Automatic placement: A review of current techniques. *Proceedings of the 23rd Design Automation Conference* (pp. 622-629). Las Vegas: IEEE Computer Society Press.

Prieditis, A.E. (1991). Machine discovery of effective admissible heuristics by means-ends analysis. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence.* Sidney, Australia: Morgan Kaufmann.

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. (1992). *C4.5: Programs for machine learning.* Morgan Kaufmann.

Quintus Computer Systems, Inc (1990). *Quintus Prolog reference manual.* Mountain View California.

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.

Rich, E., & Knight, K. (1991). *Artificial intelligence.* McGraw-Hill.

Rosenbloom, P. (1982). A world-championship-level othello program. *Artificial Intelligence, 19*, 279-320.

Sahni, S., & Bhatt, A. (1980). The complexity of design automation problems. *Prceedings of the 17th Design Automation Conference* (pp. 402-411). IEEE Computer Society Press.

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.

Saxena, S. (1991). *Predicting the effect of instance representations on inductive learning.* Doctoral dissertation, Department of Computer Science, University of Massachusetts, Amherst, MA.

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.

Shaefer, C. (1988). The ARGOT strategy. *Proceedings of the First International Workshop on Change of Representation and Inductive Bias.* New York: Philips Laboratories.

Shannon, C. (1950). Programming a computer for playing chess. *Philosophical Magazine, 40*, 356-375.

Sussman, G. J. (1975). *A computer model of skill acquisition.* New York: American Elsevier.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning, 3*, 9-44.

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.

Tou, J. T., & Gonzalez, R. C. (1974). *Pattern recognition principles.* Reading, MA: Addison-Wesley.

Utgoff, P. E., & Mitchell, T. M. (1982). Acquisition of appropriate bias for inductive concept learning. *Proceedings of the Second National Conference on Artificial Intelligence* (pp. 414-417). Pittsburgh, PA: 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. (1989). Perceptron trees: A case study in hybrid concept representations. *Connection Science, 1*, 377-391.

Utgoff, P. E., & Brodley, C. E. (1991). *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. (1991). 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. (1983). A result on the computational complexity of heuristic estimates for the A* algorithm. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (pp. 777-779). Karlsruhe, West Germany: William Kaufmann.

Watanabe, L., & Rendell, L. (1991). Feature construction in structural decision trees. *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 218-222). Evanston, IL: Morgan Kaufmann.

Winston, P. H. (1977). *Artificial intelligence.* Reading, MA: Addison-Wesley.

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

Young, P. (1984). *Recursive estimation and time-series analysis.* New York: Springer-Verlag.