Computational Intelligence, Volume 11, Number 4, 1995

KNOWLEDGE-BASED FEATURE DISCOVERY FOR EVALUATION FUNCTIONS

Tom E. Fawcett

NYNEX Science and Technology, 400 Westchester Ave., White Plains, NY 10604 USA
Phone: (914) 644-2193
E-mail: fawcett@nynexst.com

ABSTRACT

Since Samuel's work on checkers over thirty years ago, much effort has been devoted to learning evaluation functions. However, all such methods are sensitive to the feature set chosen to represent the examples. If the features do not capture aspects of the examples significant for problem solving, the learned evaluation function may be inaccurate or inconsistent. Typically, good feature sets are carefully handcrafted and a great deal of time and effort goes into refining and tuning them. This paper presents an automatic knowledge-based method for generating features for evaluation functions. The feature set is developed iteratively: features are generated, then evaluated, and this information is used to develop new features in turn. Both the contribution of a feature and its computational expense are considered in determining whether and how to develop it further.

This method has been applied to two problem solving domains: the Othello board game and the domain of telecommunications network management. Empirical results show that the method is able to generate many known features and several novel features, and to improve concept accuracy in both domains.

Key words: constructive induction, feature discovery, evaluation functions, learning

INTRODUCTION

In his early work in artificial intelligence, Arthur Samuel (1959) developed a program that was able to play checkers. Samuel's program learned an evaluation function for board positions, based on a set of features used to characterize a position. By adjusting the coefficients of these features it was able to achieve a modest level of proficiency at the game. However, much of the program's power came from Samuel's careful design of the features. Samuel (1959) commented:

> It might be argued that this procedure of having the program select [features] for the evaluation polynomial from a supplied list is much too simple and that the program should generate [features] for itself. Unfortunately, no satisfactory scheme for doing this has yet been devised. (p. 87)

In later work, Samuel (1967) greatly improved the method for combining features, resulting in substantial improvement in the program's performance; yet the problem of automatic feature generation remained "as far in the future as it seemed to be in 1959" (p. 617). Samuel identified the automatic construction of features as a major open problem of great importance.

For over thirty years it has remained an open problem. Many game-playing programs have been written that achieve high levels of performance using evaluation functions (Berliner 1980; Rosenbloom 1982; Lee & Mahajan 1988). These programs match and sometimes exceed the abilities of humans. Like Samuel's program, some of these systems are able to learn their evaluation functions automatically; however,

all of them depend on hand-coded features to describe problem states. It is still necessary for a human to devise a suitable feature set for a domain, to evaluate the performance of the program, and to alter the features if necessary.

Even systems that are able to generate features internally show sensitivity to input representation. For example, Tesauro (1992) has developed a system that learns to play backgammon using a temporal difference method. The performance using primitive features is very good; nevertheless, adding expert hand-crafted features resulted in substantial performance improvement. As Tesauro (1992) states, "The features used in Neurogammon were fairly simple, and it is probably the case that the features in Berliner's BKG program or in some of the top commercial programs are more sophisticated and might give better performance" (p. 455). Even systems that are able to generate features internally may benefit from using sophisticated features that incorporate knowledge of the domain.

In machine learning, the field of constructive induction has developed methods for extending representations for inductive concept learning (Rendell 1985; Schlimmer 1987; Matheus 1990; Pagallo & Haussler 1990). However, the methods generally treat concept learning as an isolated task. They do not exploit knowledge about the goals and operators of the domain, nor do they use feedback from problem solving to guide feature generation. They typically use weak, general operators for combining features, rather than operators that exploit domain knowledge. Some constructive induction work exploits domain knowledge, but either remains isolated from problem solving (Drastal, Czako, & Raatz 1989) or imposes restrictions on the problem domain (Utgoff 1986).

This paper presents a knowledge-based approach to feature generation for evaluation functions. The ultimate goal of this work is to be able to generate, for any domain, features that are useful for an evaluation function. The method was designed to require little more than what a human is given. It starts with basic features and progressively refines and develops them, using both domain knowledge and feedback from learning. The transformations that accomplish the refinement are divided into four classes, each of which serves a different purpose.

This method has been implemented in a system called Zenith. Samuel's original work on checkers motivated this research; however, because many more features are known for the Othello board game, Othello was used as the primary domain of exploration. Zenith has also been tested in a second domain, that of Telecommunications Network Management. In both domains, the method is able to generate useful features for evaluation functions.

The remainder of this paper is organized as follows. Section 1 discusses the domain of Othello. Section 2 reviews some general assumptions about learning and problem solving. Section 3 discusses the basic theory behind the method, apart from the implementation details, which are covered in Section 4. Section 5 discusses the empirical results of Zenith applied to Othello, and Section 6 describes the features that Zenith generated in Othello, as well as the features it could not generate. Section 7 discusses some of the problems and issues that were brought to light by this research, and possible extensions to Zenith that would address them. Section 8 concludes the paper.

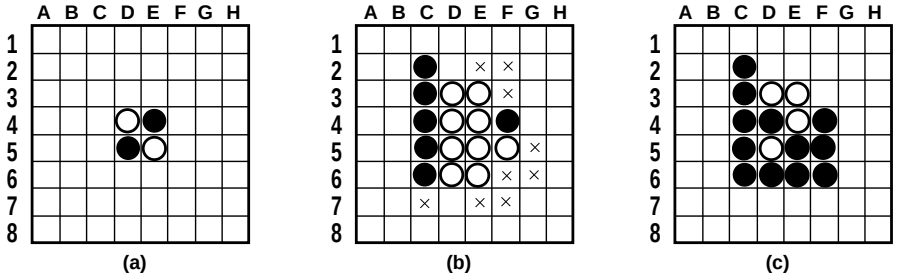


FIGURE 1. Othello boards: (a) Initial Othello board (b) Board in mid-game (c) After Black plays F6 on (b).

1. THE GAME OF OTHELLO

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

Othello is a two-player game played on an 8×8 board. One player is designated the White player, the other the Black player. There are 64 discs colored black on one side and white on the other. The starting configuration is shown in Figure 1a. Black always moves first, with players alternating turns.

On a turn, the player can place a disc on any empty square that brackets a span of the opponent's discs ending in a disc of the player's own color. The span can be horizontal, vertical or diagonal. For example, in Figure 1b, Black could place a black disc on any of the squares marked with an "X". When a player places a disc at the end of a span, all the discs in any formed span are *flipped* (changed to the player's color), and the player is said to own them. For example, if Black takes F6 in Figure 1b, the board in Figure 1c would result. A player thus gains discs by placing them on the board and by flipping discs of the other player. In certain configurations pieces cannot be flipped because no span can be placed through them; these pieces are said to be *stable*.

The game continues until neither player has a legal move, which usually occurs after all 64 squares have been taken. At this point the player with the greater number of discs wins the game, and the number of points by which the player has won is simply the difference between the two piece counts. For analysis, an Othello game is often broken up into three segments: the early game (from 4 to 16 pieces), the

[1]Othello is also known as Reversi. Othello is a registered trademark of Tsukada Original, licensed by Anjar Co.,

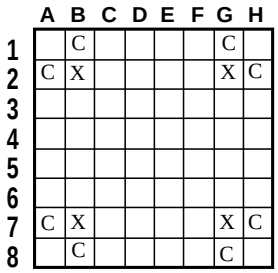


FIGURE 2. Special squares in Othello

middle game (from 17 to 32 pieces), and the late game (from 48 pieces to the end). There are usually 60 moves in an Othello game because the game usually does not end until the board is full. Frey (1986) estimates that Othello, with a search space of 10^{60} nodes, is intermediate in complexity between checkers (with 10^{40} nodes) and chess (with 10^{120} nodes).

There are several distinguished squares on the Othello board, shown in Figure 2. A corner square is inherently stable because once it is occupied there is no sequence of moves that will flip a disc on it; in addition, stable squares can form the basis for larger stable regions. Therefore, gaining control of corner squares is a goal in Othello. There are two sets of distinguished squares adjacent to the corners that are significant in corner square control. The squares along the edges immediately adjacent to the corners are called C squares (A2, B1, G1, H2, A7, B8, H7 and G8). X squares (B2, G2, B7 and G7) are diagonally adjacent to the corners. X and C squares are both considered dangerous to own because they can allow the opponent to move into the corner. C squares are somewhat less vulnerable than X squares because it is more difficult to move onto an edge than to flip a piece along a major diagonal.

2. PROBLEM SOLVING, EVALUATION FUNCTIONS AND FEATURES

The purpose of learning an evaluation function is to aid a problem solver that performs state-space search. The problem solver generates a set of successors of a given state, then evaluates the set using its evaluation function, and chooses one determined to be the best. This process iterates until a goal state is reached. The overall goal of learning is to improve the performance of the problem solver, either by increasing the quality of the goal state found, or by decreasing the amount of search done by the problem solver in finding a goal. Because the evaluation function directs the problem solver, increasing the accuracy of the evaluation function should improve problem solving performance.

Evaluation functions may be expressed in different forms. In this work, an evaluation function is learned from pairs of states in which one of the states is preferred to the other (Utgoff & Clouse 1991). Both linear threshold units and decision trees

have been used as function forms.

The purpose of feature generation is to aid the evaluation function in evaluating problem states. Informally, a feature is useful to the evaluation function if the feature identifies some aspect of a state that is predictive of its quality. A useful feature serves to distinguish states that lie on a goal path from states that lead away from the goal, or to a goal state of lesser quality.

3. KNOWLEDGE-BASED FEATURE DISCOVERY

The method of feature discovery we propose is transformational, in which new features are developed and refined from existing ones. The method comprises a set of transformation classes and a strategy for controlling them. The transformations and control strategy are domain independent; some transformations utilize information from the problem specification in creating new features, but the actions of the transformations are not tailored to an individual domain.

The feature generation component has access to a specification of the problem, also called a *domain theory*. The problem specification is declarative, and includes information such as the performance goal, executable definitions of operators in the domain, and expressions for calculating the pre-images of expressions with respect to operators. However, the domain theory contains no information about strategy or tactics, or how to search the space effectively. It also has no knowledge about how to generate or select features.

Feature generation begins with a feature created from the performance goal. Progressively better features are developed by transforming existing ones. For example, in Othello the initial feature measures whether a state is a final state in which the Black player has won. By decomposing and refining this initial feature with its transformations, Zenith is able to generate more useful features. Zenith's feature generation for Othello is discussed in Section 6. The performance goal and transformations define a feature space through which Zenith searches. The development of this space may be characterized as a modified beam search (Newell 1978) based on features' usefulness to the concept learner.

Within our framework, a feature is based on a logical formula comprised of first-order terms. A feature's value with respect to a state is the number of solutions of the formula in the state. Every feature has a computational time cost associated with it, which is a measure of the amount of time required to evaluate the feature in a state. The evaluation function's cost is bounded in some way so that it can use only a subset of the total features generated. Additional details about features are given in Section 4.1.

The following four sections discuss the transformation classes in general terms, explaining what each accomplishes and giving examples of its use in the derivation of known features.

3.1. Decomposition

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

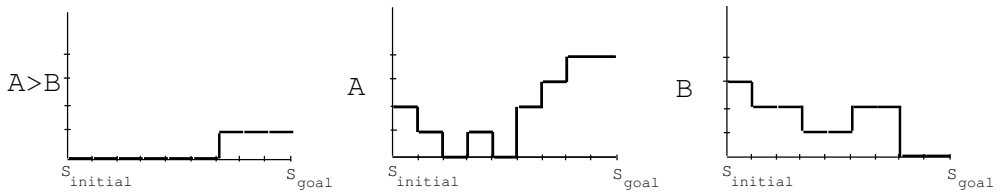


FIGURE 3. A graph of values of the expression $A > B$, at left, and its decompositions A and B. In these graphs, TRUE is mapped to one and FALSE is mapped to zero. Each point on the X axis is a state on the path from the initial state to the goal state.

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

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

An example of decomposition is shown in Figure 3. The expression $A > B$, at left, is decomposed into the expressions A and B, shown next to it. The graphs show sample values of the expressions for states along a path from $S_{initial}$ to S_{goal}. The expression $A > B$ may not change from one state to the next, so its values may not be useful to the evaluation function because they do not allow it to discriminate alternative state choices. The two expressions A and B may vary more often, and providing their values to an evaluation function may allow the function to discriminate states for which the $A > B$ relation is constant. If the evaluation function is a weighted sum of feature values, assigning a positive weight to A and a negative weight to B can direct the problem solver to states with higher A values and/or lower B values.

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

3.2. Goal Regression

Goal regression creates the pre-image of a formula with respect to an operator. If a feature is useful for the evaluation function, then it may be beneficial for the evaluation function also to use features that measure how the original feature is affected by the domain operators. For example, in many board games if it is useful

to measure the number of pieces owned by a player, it is probably also useful to measure the number of pieces that could be acquired (or lost) by a move. The latter feature is created by regressing the piece ownership term through the move operator.

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

1. In other systems, usually only the performance goal is regressed. In this theory, the formula of *any feature* can be regressed through an operator. Without this ability, goal regression would be capable only of producing features that were pre-images of the performance goal.
2. Usually goals are regressed along an entire solution path. In this theory, only a *single* goal regression step is done at a time. This difference allows the resulting feature(s) to be tested for usefulness before goal regression is applied again. It is assumed that the nth pre-image is at least as useful as the $n+1$st pre-image.

Goal regression may also be viewed as computing enablement and disablement conditions. These conditions are common in the known features of board games. In Othello, ownership of corner squares is important and most Othello programs include features that measure corner square ownership. In Othello, the corner squares can only be captured from certain configurations involving certain squares, known as the C and X squares (see Figure 2); C and X squares enable the capture of corner squares. Most Othello programs employ features that check for ownership of corner squares, as well as other features that check for ownership of C and X squares.

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

3.3. Abstraction

Features are often created (for example, by goal regression) that could be valuable to the evaluation function but are not worth their cost. Both abstraction and specialization can be used to reduce cost. Abstraction is a form of generalization that removes details from a feature. If a condition can be achieved easily, it can be considered a detail; if it can only be achieved with much effort (or not at all), it is likely to be an important condition that should not be removed (Sacerdoti 1974).

An example of abstraction in Othello is shown in Figure 4. The definition of a legal move for the black player is a blank square next to a line (in any direction) of one or more white pieces, terminated by a black piece. A graphical depiction of these three conditions is shown at the top left of the figure. Of the three conditions, the easiest for Black to influence is the existence of a black piece (condition 1), because Black can place black pieces directly onto the board, but can neither place white pieces nor create blank squares. Therefore, a more abstract feature can be produced by removing condition (1) of the Othello move. The resulting expression, shown at the top right of Figure 4, matches a pattern consisting of one or more white pieces terminated by a blank square. These conditions are the basis for several mobility features used by Rosenbloom (1982). These features are cheaper to evaluate than the original conditions of the Othello move and preserve much of their accuracy.

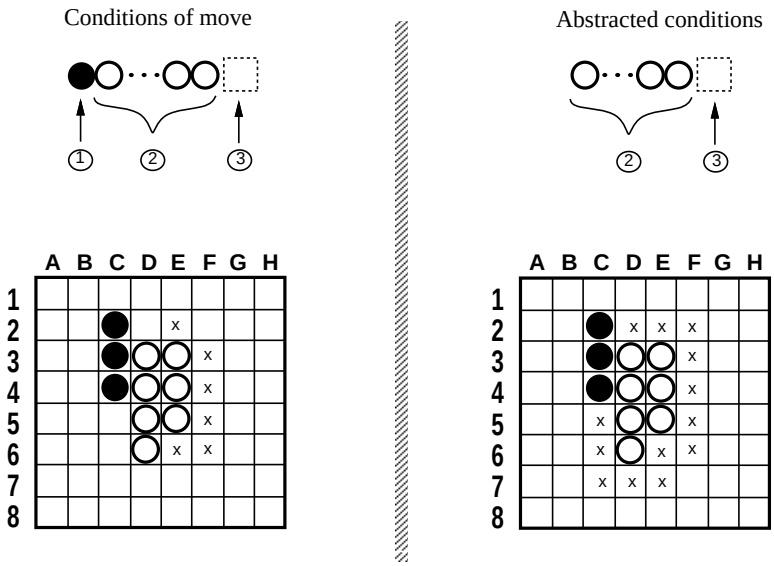


FIGURE 4. An example of abstraction in Othello.

Figure 4 also shows, below each pattern, the matches of the pattern on a sample Othello board. On each board a small "x" marks the matches of the blank square (condition 3) on the board.

A second example of abstraction occurs in the specially designated X and C squares of Othello. As mentioned in Section 3.2, the X and C squares enable the taking of corner squares. However, owning an X or C square does not alone guarantee that the corner square can be captured; it is also necessary to place pieces on the other side of the X or C square to enable a move into the corner. But these other pieces are usually easy to place, and so they may be considered removable details of the enabling conditions. Thus, features that check X and C squares are abstractions of the actual enabling conditions of corner square captures.

3.4. Specialization

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

A common way of specializing a disjunctive formula is to remove one of its disjuncts. Zenith's Remove-disjunct transformation accomplishes this, by choosing a disjunct occurring in a feature's formula.

Another way of specializing a feature is to to replace a call to a recursive predicate with its base case from the domain theory. Zenith's transformation expand-to-base-case replaces a call to a recursive predicate with that predicate's base case. Evaluating a predicate's base case is usually much less expensive than evaluating the recursive predicate.

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

In Othello, "semi-stability" is an important property of a piece: it determines whether the piece can be captured immediately. Checking all squares for semi-stability is expensive, but there are some squares (the corner squares) that invariably satisfy it due to the geometry of the board. It is much easier to check whether individual, known squares are owned than to check each owned square for semi-stability.

In checkers, it is important to prevent the opponent from creating a king by reaching the back row. There are special formations of pieces along the back row that allow a player to block the opponent. Two such configurations are the Bridge and the Triangle of Oreo (Samuel 1959). Both configurations may be considered invariants of the "king prevention" property. It is much less expensive to test for these special configurations than to determine whether an arbitrary configuration prevents the opponent from creating a king.

3.5. Summary

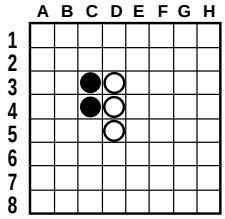
Decomposition and goal regression transformations can create new "subgoal" features. Decomposition transformations may be seen as a syntactic subgoal creation method, and goal regression as a semantic subgoal creation method. The abstraction and specialization components refine features by removing details and finding special cases, both of which can reduce cost.

4. THE ZENITH SYSTEM

The theory discussed in the previous section has been implemented in a system called Zenith. This section describes the architecture of the implemented system. Additional details, including discussion of alternatives considered, are available in Fawcett's thesis (1993).

Zenith operates in cycles. With each cycle new features are generated and evaluated. The data collected on features in a cycle are used to direct the generation of new features. Each cycle consists of the following three main steps:

1. Problem solving is performed using the current evaluation function. New training instances are generated from the problem solving episode.
2. New features are generated by applying the transformations to existing features. Data on feature performance, provided by the feature selection process, is used to control the generation.
3. Features are selected for the evaluation function, using the current feature set trained on instances from past problem solving. The result is a new evaluation function and data on feature performance.



Formula:

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

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

FIGURE 5. A sample Othello state and four features evaluated in it. All features are based on the formula shown, but each uses a different variable list.

The next two sections present the formalism used for features and the set of transformations used in Zenith. The final two sections discuss how features are generated and selected.

4.1. Feature Formalism

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

Formula ::= Formula $\wedge$ Formula
Formula ::= Formula $\vee$ Formula
Formula ::= $\neg$ Formula
Formula ::= Term

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

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

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

Feature evaluation is illustrated in Figure 5, which shows an Othello state, a formula, and four features (f1 through f4). All of the features use the formula shown, but each has a different variable list. The table in Figure 5 shows the variable list for

TABLE 1. Transformations in Zenith

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

each feature, followed by the set of values in the Othello state that satisfy the variable list. The value of each feature, shown in the right-hand column, is the cardinality of this set.

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

When a feature is created it inherits the variable list of the feature from which it was transformed. Variables can be removed from the variable list by the remove-variable transformation, discussed in Section 4.2.

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

1. The *discriminability* of a feature is the ability of the individual feature to discriminate the instances when used by the concept learner. Because this is an individual measure, it disregards the effects of other features in the feature set.
2. The *cost* of a feature is the average amount of time it takes to compute the value of the feature in a state. Cost is determined empirically by measuring CPU time.

4.2. Transformations

Features are generated using the domain-independent transformations shown in Table 1. A transformation applies to a feature and creates one or more new features from it, without losing the original feature. The next four sections explain each of the transformations in Table 1.

Decomposition Transformations. Zenith employs three decomposition transformations. Split-conjunction operates on features with conjunctive formulas. It partitions the formula's terms based on the variables it uses, such that none of the partitions share variables. A separate feature is created for each partition. The justification for this decomposition is that the satisfiability of each partition is not

dependent on the satisfiability of any other partition. For example, the formula:

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

would be partitioned into two features:

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

based on their common use of X and Y, and

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

based on their common use of Z.

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

Split-arith-comp transforms a feature whose formula contains an arithmetic comparison ($=, \neq, <, \leq, >, \geq$) between two variables X and Y. The feature is split apart into two features that calculate the values X and Y. This transformation is similar to the AE transformation of CINDI (Callan & Utgoff 1991a; 1991b). Split-arith-calc applies to features that perform an arithmetic calculation. A separate feature is created for each arithmetic operand in the calculation.

Goal Regression Transformations. Zenith uses a single goal regression transformation, regress-formula, which produces new features by regressing the formula of an existing feature through a domain operator. The transformation chooses a domain operator, then chooses a player, and generates the pre-image of the formula with respect to the instantiated operator.

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

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

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

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

Abstraction Transformations. Zenith has two abstraction transformations. The transformation Remove-LC-term performs term-dropping abstraction similar to that in ABSTRIPS (Sacerdoti 1974). It removes the least constraining term occurring in a formula. Zenith places a term in one of three categories, in order of increasing criticality:

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

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

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

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

The second abstraction transformation, remove-variable, removes a single variable from a feature's variable list. Remove-variable is classified as an abstraction transformation because the resulting feature provides less information about the formula, and because the resulting feature is also usually less expensive to compute than the original. By creating features with full variable lists, Zenith begins with a fully informed version of the feature; if the feature proves valuable, Zenith can then discard variables to create less expensive versions of it.

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

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

S is computed empirically by sampling Zenith's database of training instances, generating X values and discarding X values that satisfy $p(X) \wedge \neg q(X)$. If any elements

remain in S, a new feature f' is created that has formula:

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

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

4.3. Controlling the Transformations

Zenith's initial feature is created directly from the performance goal. New features are generated subsequently by applying transformations to the existing features.

Unrestricted transformation of features is impractical because most transformations can be applied to most features. Therefore, a control strategy is employed that restricts the application of transformations to features that could plausibly benefit from the transformation. Two data about each feature are used to guide feature generation. Every feature incurs a time cost to compute its value in a state. Because the total time cost of the evaluation function is bounded, usually only a subset of the existing features are used in the evaluation function. A feature's selection for use in the evaluation function is an indication of its quality. The cost of a feature and its use by the evaluation function give rise to the following two definitions:

- An **expensive** feature is one whose cost exceeds a maximum threshold. In the implemented system, this threshold is 10% of the maximum allowed cost of the entire evaluation function.
- An **active** feature is one that has been selected for use by the evaluation function. A feature that is not used by the evaluation function is called inactive.

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

1. Decomposition transformations may be applied to any feature. Because decomposition is based on specific syntactic forms and generally produces features that provide more information than the original, it is beneficial to apply decomposition transformations to both active and inactive features alike. Features resulting from decomposition are usually less expensive than the original, so both expensive and inexpensive features can be decomposed beneficially.
2. If the feature is *expensive*, abstraction and specialization transformations are applied to it, in order to reduce its cost. In principle, abstraction and specialization could be applied to any feature because reducing feature cost is arguably always worthwhile. However, abstraction and specialization generally involve a trade-off between cost and accuracy. This control rule is based on the assumption that the accuracy sacrifice is probably not worth making for features that are already inexpensive.
3. Only if a feature is *active* and *inexpensive* will goal regression be applied to it. Goal regression usually produces features that are more expensive than the original, so it is only applied to features that have already proven their worth. This rule is based on the assumption that if a feature is inactive, it is unlikely that a feature based on its pre-image would have a greater contribution to concept accuracy.

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

4.4. Feature Selection

After generating new features, Zenith must select a subset of them to use for the evaluation function. The feature selection step has two results. First, the evaluation function produced will be used in problem solving in the next cycle. Second, the feature selection method partitions the features into active and inactive subsets. This partitioning is used in feature generation in the next cycle.

Determining an optimal subset of features is prohibitively expensive. Zenith uses a *sequential backward selection* method (Kittler 1986). The method creates an evaluation function from the entire feature set, then it casts out individual features until the total cost of the evaluation function falls below the imposed limit. The feature to be cast out is the one that produces the smallest decrease in concept accuracy when removed from the set. When the process terminates, the remaining features are used in the evaluation function. These are termed the *active* features, and the features that have been cast out are inactive.

Feature selection and generation are closely linked. A simpler strategy might keep *no* inactive features and develop only the active ones. However, an inactive feature may not be bad; it may simply be too expensive or too specific, in which case further development may improve it. At the other extreme, a control strategy might keep *all* inactive features and continue developing them regardless of their worth. This strategy would quickly overwhelm the feature selection mechanism, which must examine the current features and select the best for use by the evaluation function.

Instead, an intermediate strategy is used in which a fixed number of the inactive features are kept in a reserve set. After feature selection, the inactive features are ordered by their individual contributions to the evaluation function. The features of highest contribution are kept, and the remainder are discarded. In the experiments below, the 20 inactive features of highest discriminability were kept, and the remainder were discarded. Thus, Zenith performs a beam search through feature space with a beam size of 20.

5. OTHELLO EXPERIMENTS AND RESULTS

Zenith's primary domain of application was the board game of Othello. This section describes Zenith's performance in that domain. Zenith has also been applied to Telecommunications Network Management (Fawcett & Utgoff 1992; Fawcett 1993).

The Othello problem specification corresponds to the rules given to a human player[2]. It contains a definition of the Othello board, a specification of the Othello move, and statement of the goal. Zenith's goal in Othello is a win for the Black player. The theory specifies auxiliary predicates necessary for the definition of `win`, such as

[2]The Othello problem specification is available via the Internet from the machine learning database repository at the University of California at Irvine. The URL is `ftp://ics.uci.edu/pub/machine-learning-databases/othello/new-othello.theory`.

`span` and `line`. These predicates are in turn defined in terms of the operational predicates, the primitive observable attributes of an Othello state.

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

The domain theory also contains meta-information about predicates. It specifies mode declarations (Warren 1977) about Prolog predicates, indicating which arguments of a predicate are used for input, output or both. Mode declarations are used to determine whether a formula is legal. The domain theory specifies the conditions under which a predicate call is determinate, and which predicate calls are operational. An operational predicate is considered to be an atomic operation and its definition will not be inspected by Zenith.

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

5.1. Experiments

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

Zenith's performance component makes a move using the preference predicate (Utgoff & Saxena 1987; Utgoff & Clouse 1991). Several choices were made in order to simplify the performance component:

- The performance component performs 1-ply search. It determines the descendants of the current state and chooses the most preferred successor state using its preference predicate. This choice was made primarily to reduce the amount of time spent by the performance element.
- Zenith does not use perfect search in the Othello end-game. The same preference predicate is used throughout the entire game. This was done to keep the performance element domain independent. We did not want to bias the performance component toward any particular two-player game, so we did not assume that the end-game could be searched, or even identified. Hence, Zenith employs a single search method over the entire state space.

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

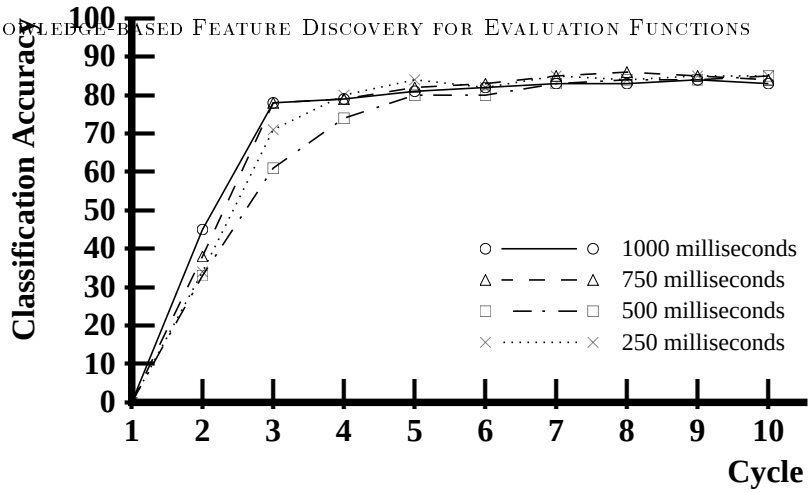


FIGURE 6. Classification accuracies for Othello using a linear threshold unit.

At the end of every cycle the *instance accuracy* of the preference predicate is tested and reported. The instance accuracy is the classification accuracy of the preference predicate, measured as follows. The instance set is partitioned randomly into a training set (2/3 of the instances) and a testing set (the remaining 1/3). The preference predicate is trained on the training set, then its accuracy recorded against the testing set. Accuracy is measured by applying the preference predicate to each instance and checking for strict (consistent) preference. The accuracy is the ratio of correctly classified test instances to total test instances.

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

Performance improvement is also measured by playing a set of ten games against an opponent after every cycle and counting the number of games won. A separate opponent was created that used Zenith's framework but Wystan's evaluation function. The opponent uses 1-ply search and no exhaustive search in the end-game, but uses Wystan's expert hand-coded features. A separate opponent called ORFEO (ORacle FEature Opponent) is used to test Zenith's performance. ORFEO performs 1-ply search using Wystan's features but performs no end-game search. Thus, ORFEO has Zenith's framework but uses a handcoded expert feature set.

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

Linear Threshold Unit. The first tests used a linear threshold unit (LTU) as a preference predicate. The LTU was trained using the Recursive Least Squares (RLS) training rule (Young 1984). For a given preference pair, the LTU was trained on the difference vectors of the preference pair states. First the difference vector was presented with the desired output value of +1, then the negated delta vector was

TABLE 2. Number of features generated for Othello (LTU)

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

presented with the desired output value of -1. The LTU was used to classify a new pair of states by calculating the feature delta vector of the states and presenting the it to the LTU. For a positive output, the first state is preferred to the second; for a negative output, the second is preferred to the first.

The effect of feature generation can be measured directly as the change in classification accuracy of the LTU. Four experiments were run varying the evaluation function cost limit as an independent parameter. Each of the experiments was allowed to run for ten cycles, at which point the accuracies had stabilized. Figure 6 shows classification accuracy as a function of cycle number. In these runs, the accuracies rose from zero in the first cycle to a final value between 83% and 86%. As the evaluation function cost limit was raised, the corresponding accuracy attained by Zenith rose only slightly.

To test the effect of further increase in the evaluation function cost limit, another experiment was performed in which three seconds (3000 milliseconds) were allowed for each evaluation. The highest accuracy achieved in this run, 87%, is not substantially higher than the accuracies in Figure 6, indicating that there may be an accuracy ceiling above which Zenith's feature generation, trained with an LTU, cannot rise.

Tables 2 and 3 show data on feature generation in these experiments. Table 2 shows the total number of features generated by Zenith in the experiments with an LTU. The second column shows the total number of features generated in the run, and the third column shows the number of features in the active set at the end of each run. The last column shows the highest instance accuracy attained in the run. Table 2 demonstrates that Zenith does not generate a prohibitive number of features as the time limit is raised.

The final size of the active feature set, shown in Table 2, does not increase monotonically with the time limit. This may seem counterintuitive because a greater time limit should imply that more features could be used. However, Zenith develops a feature differently depending on whether it is considered expensive, and the definition of expensive is a function of the time limit. Hence, given higher time limits, Zenith may end up with several expensive features rather than many inexpensive features, if the expensive features attain equal or better accuracy. However, there may be benefits to reducing the cost of expensive features so that more features can be admitted into the active set. This is an area for future study.

Table 3 shows the number of times each transformation was used. After each transformation is the total number of times it fired in each run. A "firing" is the generation of a new feature by a transformation. Split-arith-calc was not used at all in this domain because it decomposes arithmetic calculations, and the domain theory of Othello performs no arithmetic calculations. Other than that, all transformations

TABLE 3. Transformation firing data for Othello (LTU)

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

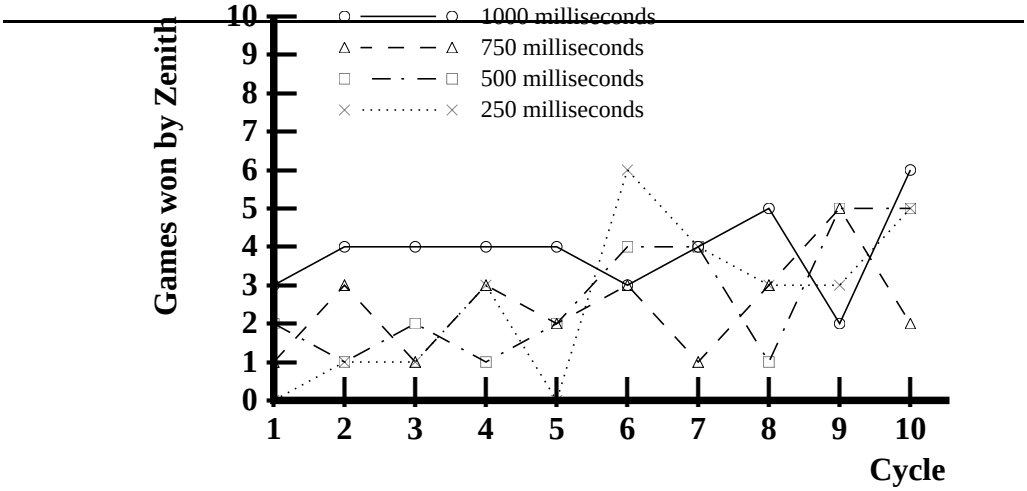


FIGURE 7. The performance of Zenith after every cycle, using a linear threshold unit. Each point represents the number of games out of 10 won by Zenith against the ORFEO opponent.

were used multiple times in a run.

Performance improvement was also measured at the end of every cycle by having Zenith play against ORFEO. Figure 7 shows the results for these games, with the number of games won as a function of cycle number. Because instance classification accuracy improves with the cycle number, Figure 7 demonstrates that performance generally improves with classification accuracy; however, the performance is erratic. This may be due to the feedback between performance and learning with on-line training. As the evaluation function changes, it causes the problem solver to explore different parts of the instance space, which in turn changes the set of training instances. This phenomenon was observed and has been labeled *temporal crosstalk* by Jacobs (1990). A learning system suffering from temporal crosstalk appears to perform poorly or erratically because it is led continually into new regions of the problem space for which its learned responses may be ineffective or even disastrous.

Donald Mitchell (1984) mentions a similar phenomenon related to novice versus expert learning. He argues that a program plays most effectively when its evaluation

function is tuned against opponents whose skill level is comparable to its own. Within a learning system, this improvement from novice toward expert constitutes a form of concept drift. Because of the feedback from performance to the training instance set, the relationship between evaluation function accuracy and performance is complex. Investigation of this relationship is the subject of future work, and is discussed further in Section 7.3.

5.2. Univariate Decision Trees (C4.5)

In order to investigate the sensitivity of feature generation to the concept form, the second set of Othello experiments used the C4.5 program (Quinlan 1993) as a concept learner. C4.5 is a learning program that induces a decision tree from a set of examples. C4.5 is based on ID3 (Quinlan 1986) but has several extensions, the most important of which is the ability to accommodate continuous-valued numeric features. C4.5 was used with windowing disabled, but with all other options defaulted.

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

A modified sequential backward selection method was used with C4.5. The utility of a feature was measured by summing, for every internal node in the decision tree that tested the feature, the number of instances that reached that node. The resulting count thus measured how many instances used the feature in their classification. The feature with the lowest count was assumed to be the least useful feature, and was discarded.

Four experiments were run, varying the evaluation function cost limit as an independent parameter with the same values as with LTUs. Again, each of the experiments was allowed to run for ten cycles. Classification accuracy is shown in Figure 8. In these runs, the accuracy rose from zero to a final value between 76% and 81%. These accuracies were generally slightly lower than the corresponding instance accuracies for LTUs. This is probably because univariate decision trees are more suitable for discrete symbolic attributes than for Zenith's continuous attributes. For continuous attributes, in addition to determining the best attribute on which to split, a univariate decision tree algorithm must find proper range values for the attribute.

The accuracies of the 1000 millisecond run lagged behind those of the other C4.5 runs, and exhibited a noticeable drop in the tenth cycle. This is due to the modified backward selection algorithm. Better features were available in the last cycle, but they were not used because the highest-ranked feature was very expensive. This had the effect of squeezing out other features, and resulted in only a single feature left in the active feature set at the end of the cycle. Section 7 discusses Zenith's requirements for feature selection in more detail.

As with LTUs, a final experiment was run in which the evaluation function cost limit was raised to three seconds (3000 milliseconds). The highest accuracy achieved in this run, 78%, is similar to the highest accuracies of Figure 6, but the number of features generated is smaller. This may be due to the more erratic feature selection method.

Table 4 shows data on feature generation for the C4.5 experiments. The table shows, for each value of the evaluation function time limit, the total number of fea-

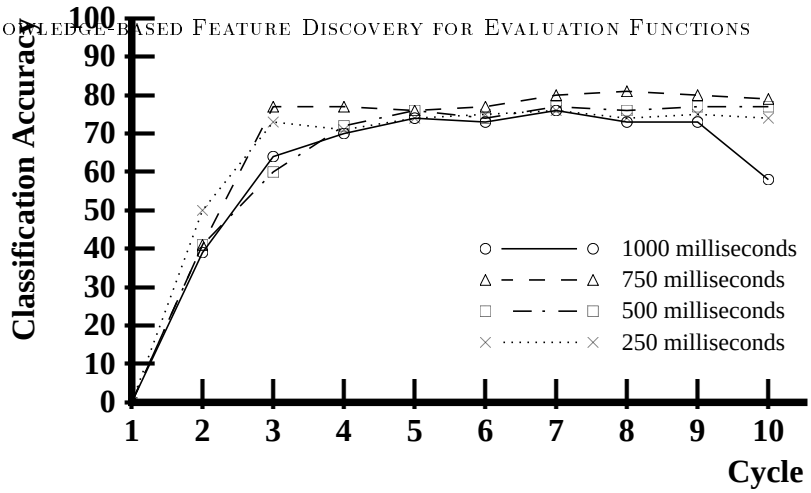


FIGURE 8. Classification accuracies for Othello using the C4.5 learning program.

TABLE 4. Number of features generated for Othello (C4.5)

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

tures generated, the number of features in the active set at the end of each run, and the accuracy attained. Again, the 1000 millisecond run had a lower accuracy than expected, and the run concluded with only a single active feature. The transformation firing data are similar to the corresponding data for LTUs, except that fewer features were generated in the 750 and 1000 millisecond runs of C4.5. This is because on average there were fewer active features in these C4.5 runs. Game-playing performance with C4.5 exhibited behavior similar to that reported with LTUs.

6. FEATURE GENERATION IN OTHELLO

This section discusses features generated by Zenith and how they compare with known features from Othello. Specifically, we discuss known features that Zenith generated, novel features that Zenith generated, and known features that Zenith did not generate.

Many of the known features for Othello have been discussed and analyzed by Mitchell (1984). A more accessible description of the game and some of the important features is included in a journal article by Rosenbloom (1982). Lee and Mahajan (1988) also discuss Othello features. Unfortunately, details of the features used by top-ranked Othello programs are rarely published while the programs are

still in competition; Kierulf's BRAND (1989) and Peer Gynt (1990) programs are exceptions.

6.1. Generation of Known Othello Features

Figure 9 illustrates Zenith's derivations of some of the known Othello features, and two novel ones. The figure shows the features discussed in this section and in Section 6.2. Due to space considerations, only a subset of all generated features are shown. Tables 2 and 4 give information on the total number of features generated for the runs.

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

Zenith's initial feature is created from the performance goal, `win(black)`, which is specified by the domain theory. This initial feature is binary, and returns one when applied to states in which Black has won and zero otherwise. This feature is useless for directing search because it only distinguishes end-game states in which Black has won from states in which Black has not. However, decomposing this initial feature yields features measuring the score for each player (Pieces), and the number of legal moves (Moves).

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

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

The AASS feature proved to be both useful and inexpensive. Goal regression was then applied to it, yielding a feature that counted the number of pieces that could be taken on the next move. This feature is similar to commonly known features such as X Next to Empty Corner and C Next to Empty Corner.

Zenith created a number of mobility features, shown on the right-hand side of Figure 9. From the initial feature, Zenith created a feature that measured the number of moves available (Moves). By expanding the definition of the Othello move, a new feature was created that counted the number of squares involved in each move. By repeatedly abstracting this feature, Zenith created a number of simpler, less costly mobility features including three used by Rosenbloom (1982) (Rosenbloom Empty, Rosenbloom Sum Empty and Rosenbloom Frontier). However, Rosenbloom's features

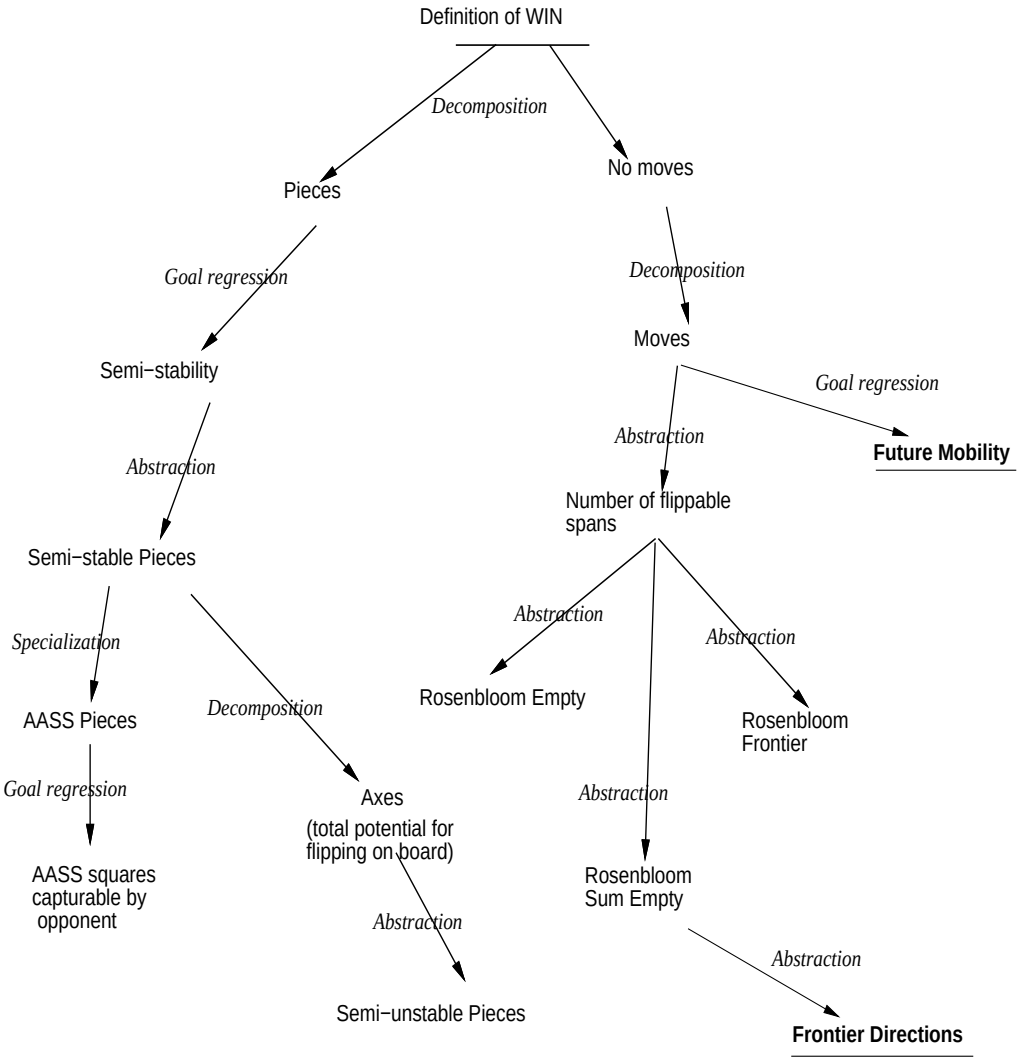
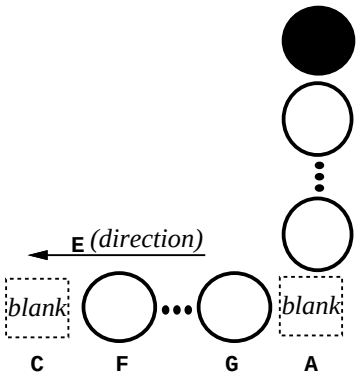


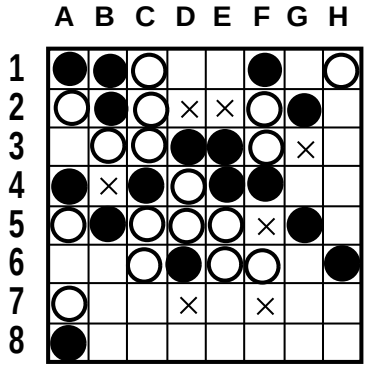
FIGURE 9. Othello features generated by Zenith

measured the difference in mobility between two players, whereas Zenith's features measured the value for each player individually.

6.2. Generation of Novel Othello Features

Figure 9 also shows several novel features generated by Zenith that, to the best of our knowledge, have not been published or discovered elsewhere. By applying the regress-formula transformation to the Moves feature shown on the right-hand side of the figure, Zenith generated several *future mobility* features that detected moves that could become available in the next state.






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

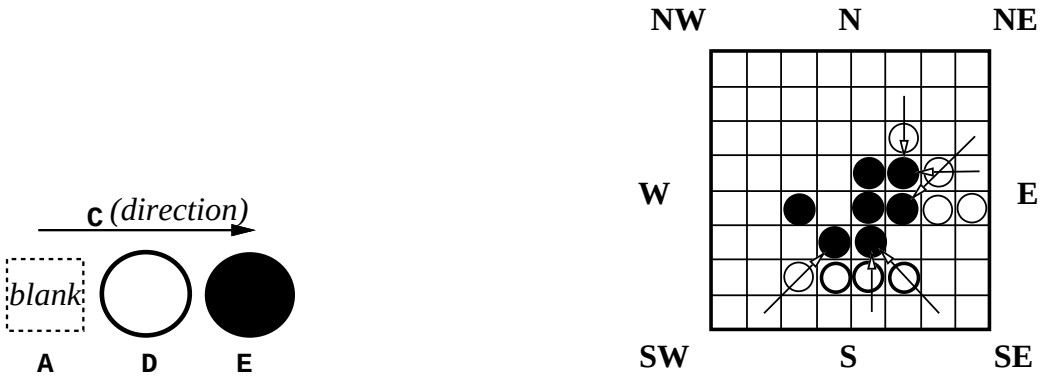
FIGURE 10. The Future Mobility feature. At top left is the pattern being matched by the feature. At top right is an illustration of the feature applied to an Othello state. At bottom is Zenith's definition of Future Mobility for the Black player.

One of these features is illustrated in Figure 10. A pattern illustrating the configuration matched by the feature is shown at the left side of Figure 10. Intuitively, the feature measures the number of ways in which a move can be created by taking a blank square. At the right side of Figure 10 is a sample board to illustrate the feature. Each of the X's on the board is a blank square bound to square `A` in the pattern (square F5 occurs twice). The variable values generated by the feature in the sample state are:

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

so the value of the feature in this state is 8. Figure 10 shows the actual text of the feature.

Another novel feature created by Zenith is called Frontier Directions. This feature measures the number of directions in which a frontier exists for a player. Its value ranges from zero to eight. Frontier Directions is illustrated in Figure 11, in which




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

FIGURE 11. The Frontier Directions feature. At top left is the pattern matched by the feature. At top right is an illustration of the feature applied to an Othello state. White has neighboring black pieces in the directions {SW,S,SE,E,NE,N}, so the feature value in this state is 6. At bottom is Zenith's definition of Frontier Directions for the Black player.

the `black` player has a Frontier Directions value of 6. This feature is a very abstract measure of mobility in that it ignores not only the existence of a bracketing piece but also the number of moves available in the same direction on the board, no matter where they are. It is surprising that such an abstract feature proved useful to Zenith, but it survived in competition with other mobility features.

6.3. Known Othello Features Not Generated

Zenith was unable to generate all known features for Othello. Some features are based on a property of the Othello domain that Zenith's transformations are unable to generate. Other features require a form that Zenith is unable to produce. Section 7 discusses improvements that could be made to Zenith to overcome some of these limitations.

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

corner often can be created.

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

Stability of interior discs is much more expensive to test because it requires testing each square for stability along its axes. Even using efficient handcoded procedures, such stability tests generally take too long to be worthwhile[3]. Instead, most programs use features that check for specific configurations of stable pieces, usually along the edge. Examples of such features are Stable Pieces, Stable Edge Pieces, Nonstable Edge Pieces, Rosenbloom Edge Score and Edge Anchor.

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

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

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

Zenith does not create features that employ edge squares because Zenith's transformations are incapable of recognizing edge squares as a special class. Zenith could create a feature for edge squares by a change to its variable-specialize transformation. Currently this transformation checks a property against instances and accepts only domain elements that are completely invariant with respect to the property (see Section 4.2). This test could be weakened so that elements that *usually* satisfy the property are accepted. This change would allow a feature to generate edge squares as regions of high semi-stability or low mobility. This idea is further explored in Section 7.

Specialization of Number. Zenith's variable-specialize transformation can specialize the *identities* of domain elements that satisfy a property. Some features are specializations of the *number* of domain elements that satisfy a property. For example, Weighted Sum Empty is a mobility feature like Rosenbloom Sum Empty that weights

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

the number of empty neighbors to determine the feature value: one neighboring disc assigns 6 to that piece's mobility score, two neighboring discs assigns 11 to the score, three discs assigns 15, etc. Thus, Weighted Sum Empty uses specializations of the number of empty neighboring squares that a piece has.

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

Zenith can create a feature that measures the number of pieces owned by the player, and a feature that measures the number of pieces that have *any number* of neighbors. However, it cannot create a feature that limits the number of neighbors of a piece. Zenith's feature formalism allows it to count the number of values satisfying a variable, but not to constrain the number of values. There is no straightforward extension to Zenith's formalism that would allow this specialization.

7. PROBLEMS, ISSUES AND FUTURE WORK

This research is exploratory, designed to answer the question *How could expert features be automatically generated for evaluation functions?* The previous section demonstrated that Zenith, when applied to a domain theory for Othello, is able to generate many of the known features of that domain. It is also able to generate some useful novel features. However, as with most exploratory work, experience with the Zenith system has illustrated some problems with the theory and has suggested directions for future work.

7.1. Evaluating and Selecting Features

Feature selection is one of the most important and time-consuming components of Zenith. It is critical because it determines not only the quality of the evaluation function but also the features that are considered active, and active features are developed differently from inactive ones.

Zenith imposes several unusual demands on a feature selection method.

1. The method must be sensitive to feature cost as well as feature worth.
2. The method must be able to accommodate groups of features that are very similar, and possibly identical, to each other. The simultaneous existence of similar features in a set can confound selection methods that evaluate features independently.
3. The method must be able to evaluate the contribution of a feature in the context of other features in the set. A feature alone may be a poor predictor of preference, but when combined with other features may be stronger. Feature selection methods that evaluate features individually may under-estimate the worth of a feature for this reason.
4. In some cases a feature will be created that produces zeroes on many of the instances; that is, it will not differentiate many of the preference pairs, so many of the resulting delta values will be zero. Some methods only penalize a feature for poor correlation with preference, but because features are evaluated on *all* states, some penalty must be incurred when the feature indicates no preference.

Determining the optimal acceptable subset of a set of features is NP-complete, so

exhaustive approaches are rarely used in practice. Instead, less expensive heuristic methods are used (Kittler 1986; Kira & Rendell 1992). Several heuristic methods were tried in Zenith.

In earlier work (Fawcett & Utgoff 1991), features were selected by reducing their worth to a single number that combined cost and discriminability. This method was unsatisfactory for several reasons. The worth function was an individual measure that was independent of the contributions of the other features in the set, and Zenith would often select several features that were very similar. Also, the discriminability/cost trade-off represented by the worth function was never perfect because cost would sometimes have a greater influence than desired. It was difficult to derive an equation that could express universally the desired relationship. Though several such equations were tried, none proved satisfactory.

The method ultimately adopted kept feature cost and contribution separate, and used a sequential backward selection method (Kittler 1986). Sequential selection methods do satisfy Zenith's four requirements, but they tend to be expensive: for n features they require $O(n^2)$ applications of the concept learner. As mentioned in Section 5.2, this expense was acceptable for LTUs but prohibitive for C4.5, so the method was modified for C4.5. The modification enabled the least valuable feature to be estimated after a single run of C4.5, thus reducing the entire selection process to $O(n)$ applications of C4.5. Even with this modification, feature selection remains one of the most expensive components of Zenith, and probably will remain so.

7.2. Control of Feature Generation

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

Several variations of this control strategy were tested, such as fixing the size of the active feature set, restricting the number of new features that could be introduced in each cycle (Fawcett & Utgoff 1991), and changing the size of the inactive feature set. The current strategy was found to produce the best results and was satisfactory for the experiments reported; however, it has shortcomings. Competition in the feature set is still high; in the final cycles of the runs, many features tend to be thrown away that have non-zero discriminabilities and could possibly be developed further.

One way to improve feature generation would be to make the application of transformations more intentional and focused. For example, Zenith could devote one cycle to reducing the cost of a valuable but expensive feature. Zenith would then apply transformations to the feature, perhaps repeatedly, and finally extract the best feature that it had generated and discard the others. One advantage of this approach is decreased competition: by developing the feature space around a single feature at a time, Zenith could do a more thorough job than by developing many features simultaneously.

7.3. Decision Making and Performance

An important issue in this research is the relationship between decision accuracy and performance. We assumed that as an evaluation function became more accurate, the decisions based on it would improve, and the performance of the problem solver would improve in turn. In Zenith this seems to happen in general, but the perfor-

mance improvement is not stable. Although the classification accuracy usually rises monotonically, the corresponding effect on problem solving performance is erratic.

Several hypotheses might explain this phenomenon. Temporal crosstalk, discussed in Section 5.1, is caused by the interaction of learning and performance. Any system that does on-line training, in which learning is interleaved with performance episodes, is susceptible to temporal crosstalk. This phenomenon was identified by Jacobs (1990), but no solution has been found yet.

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

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

7.4. Specialization

Zenith uses three specialization transformations: expand-to-base-case, remove-disjunct and variable-specialize. The first two are standard methods of specializing expressions. The third is unusual in that it specializes variables: it produces sets of values that empirically satisfy the formula or a portion of the formula. In essence, variable-specialize induces from examples the invariants of a portion of a feature. It can also be characterized as a transformation that tries to create a cheaper structural feature from a valuable but expensive functional feature (Callan 1993). Section 6 referred to a number of possible extensions to the variable-specialize transformation that might account for more of the known Othello features.

Currently, variable-specialize only accepts values that empirically *always* satisfy the conditions of the specialization (the q predicate of Section 4.2). One possible extension is to weaken this test so that a value need only satisfy the test most of the time, by imposing a non-zero threshold on the portion of instances in which the element was observed to vary. This would enable variable-specialize to create, for example, a class for squares that were usually but not always stable. The disadvantage of introducing such a threshold is that it would increase the chance of spurious inclusion in the set(s) generated. The induction performed by the transformation would be more susceptible to coincidences and false matches.

Currently, variable-specialize looks for individual domain elements that satisfy the unary predicate q. This could be extended in two ways simultaneously. The transformation could allow q to be an n-ary predicate, and it could examine configu-

rations of variable values that satisfy q, rather than individual values. For example, variable-specialize currently only finds individual Othello squares to be stable. With the extension it could find configurations of squares that are stable together. Other examples, mentioned in Section 3.4, are the Bridge and Triangle of Oreo features of checkers, both of which are special configuration of squares in checkers. Unfortunately, such an extension would have to examine many more examples, so it would be more expensive. Like weakening invariance, it might lead to spurious configurations being accepted.

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

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

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

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

7.5. Providing Goal Regression Information

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

Many machine learning systems employ the simpler, more constrained STRIPS operator formalism (Sacerdoti 1974; Waldinger 1976). This formalism lends itself to analysis: operator definitions are easy to inspect, interactions between operators are straightforward to predict, and operator pre-images are easy to determine. Unfortunately, neither Othello nor Telecommunications Network Management (TNM) has

operators that can be expressed using the STRIPS representation. Othello's operator requires doubly-nested universal quantifiers, and TNM's network controls require functions to determine their effect on network traffic flow.

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

7.6. Feature Optimization

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

Much more work could be done to optimize Zenith's features. An alternative to removing terms from a formula is to reorder them. Term reordering is beneficial because Prolog interprets a conjunction from left to right, so whenever a term backtracks it forces re-evaluation of terms to its right. By reordering terms (for example, by moving a rarely satisfied term to the beginning of a conjunction) the total number of term evaluations can be decreased substantially. Future work should investigate such techniques because of their strong effect on feature cost (Warren 1981; Smith & Genesereth 1985).

8. CONCLUSIONS

It has long been recognized that inductive learning is very sensitive to representation. Zenith is an attempt to answer the question raised by Arthur Samuel over thirty years ago, *How can a system generate useful features automatically?* Specifically, this work addresses the problem of feature generation for evaluation functions, an area that has received little attention in constructive induction. The ultimate goal of this work is to produce, for any domain, features that are as good as those created by humans.

Zenith, when applied to a domain theory for Othello, is able to generate features that are functionally equivalent to known features of that domain. With some extensions to its transformations, primarily the specialization transformations, Zenith could account for many of the known Othello features. Therefore, Zenith succeeds as an explanatory theory of feature generation[4]. Zenith is also able to generate some novel features that are useful for evaluating states. This demonstrates that Zenith is

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

promising as a generative theory of feature discovery.

Zenith is able to consistently improve its ability to classify state preference pairs by developing its set of features. Classification accuracy increases smoothly with the cycle number. Thus, the features generated by Zenith are appropriate for the domain and useful for expressing the concepts necessary for problem solving. Zenith is also able to improve problem-solving performance, although the improvement is erratic.

Tables 2 and 4 present data on the number of features generated. These data indicate that feature generation increases sublinearly with the amount of time allowed for the evaluation function. Similar results were demonstrated in Telecommunications Network Management (Fawcett 1993). This provides evidence that the method does not become prohibitively expensive as evaluation time is increased.

The theory of feature generation integrates both feature costs and feature benefits. The work acknowledges that every feature used in an evaluation function adds expense as well as accuracy. Therefore, Zenith addresses the utility problem in machine learning. This problem is most prominent in explanation-based learning (Minton 1988), but has been addressed in inductive learning as well (Nunez 1988; Gennari 1987; Tan & Schlimmer 1991).

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

ACKNOWLEDGEMENTS

Paul Utgoff advised this work and for provided valuable comments on previous papers. Comments by Jamie Callan, Bernard Silver, Chris Matheus, Michael Pazzani and Jeff Clouse on previous papers improved the presentation of this work. Jeff Clouse provided Wystan, Zenith's opponent.

This research was supported in part by a grant from GTE Laboratories Inc., and by the Office of Naval Research through a University Research Initiative Program under contract N00014-86-K-0764. Quintus Computer Systems Inc. provided a copy of Quintus Prolog, in which Zenith is written.

REFERENCES

BERLINER, H. J. 1980. Backgammon computer program beats world champion. Artificial Intelligence, **14**:205–220.

CALLAN, J. P., and P.E. UTGOFF. 1991a. Constructive induction on domain knowledge. In *Proceedings of the Ninth National Conference on Artificial Intelligence*. pp. 614–619. Anaheim, CA: MIT Press.

CALLAN, J. P., and P.E. UTGOFF. 1991b. A transformational approach to constructive induction. In *Proceedings of the Eighth International Workshop on Machine Learning*. Evanston, IL: Morgan Kaufmann. pp.122–126.

CALLAN, J. P. 1993. Knowledge-based feature generation for inductive learning. Ph.D. Dissertation, University of Massachusetts, Amherst, MA.

DE JONG, K. A., and A.C. SCHULTZ. 1988. Using experience-based learning in game playing. In *Proceedings of the Fifth International Conference on Machine Learning*. Ann Arbor, MI: Morgan Kaufmann. pp.284–290.

DRASTAL, G., CZAKO, G., and S. RAATZ. 1989. Induction in an abstraction space: A form of constructive induction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*. Detroit, Michigan: Morgan Kaufmann. pp.708–712.

FAWCETT, T. E., and P.E. UTGOFF. 1991. A hybrid method for feature generation. In *Proceedings of the Eighth International Workshop on Machine Learning*. Evanston, IL: Morgan Kaufmann. pp.137–141.

FAWCETT, T. E., and P.E. UTGOFF. 1992. Automatic feature generation for problem solving systems. In *Proceedings of the Ninth International Conference on Machine Learning*. San Mateo, CA: Morgan Kaufmann. pp.144–153.

FAWCETT, T. E. 1993. Feature Discovery for Problem Solving Systems. Ph.D. Dissertation, University of Massachusetts at Amherst, Amherst, MA. Also available as Technical Report 93-49 from the University of Massachusetts. Available electronically via `ftp://ftp.cs.umass.edu/pub/techrept/techreport/1993/UM-CS-1993-049.ps`.

FREY, P. W. 1986. Algorithmic strategies for improving the performance of game-playing programs. In *Evolution, Games and Learning*. New York, NY: North Holland. pp.355–365.

GENNARI, J. H. 1987. Focused concept formation. In *Proceedings of the Sixth International Workshop on Machine Learning*. Ithaca, NY: Morgan Kaufmann. pp.379–382.

JACOBS, R. A. 1990. Task decomposition through competition in a modular connectionist architecture. Ph.D. Dissertation, University of Massachusetts, Amherst, MA. also COINS Technical Report 90-44.

KIERULF, A. 1989. New concepts in computer Othello: Corner value, edge avoidance, access, and parity. In Levy, D. N. L., and Beal, D. F., editorss., *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad*. Halsted Press.

KIERULF, A. 1990. Smart Game Board: A Workbench for Game-Playing Programs, with Go and Othello as Case Studies. Ph.D. Dissertation, Swiss Federal Institute of Technology (ETH), Zürich.

KIRA, K., and L. RENDELL. 1992. A practical approach to feature selection. In *Proceedings of the Ninth International Conference on Machine Learning*. San Mateo, CA: Morgan Kaufmann. pp.249–256.

KITTLER, J. 1986. Feature selection and extraction. In *Handbook of pattern recognition and image processing*. New York: Academic Press. pp.59–83.

LEE, K. F., and S. MAHAJAN. 1988. A pattern classification approach to evaluation function learning. Artificial Intelligence **36**(1):1–25.

MATHEUS, C. J. 1990. Feature construction: An analytic framework and an application to decision trees. Ph.D. Dissertation, University of Illinois, Urbana-Champaign, IL. Also report no. UIUCDCS-R-89-1559 and UILU-ENG-89-1740.

MICHALSKI, R. S. 1983. A theory and methodology of inductive learning. In *Machine learning: An artificial intelligence approach*. San Mateo, CA: Morgan Kaufmann. pp.83–134.

MINTON, S. 1988. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*. Saint Paul, MN: Morgan Kaufmann. pp.564–569.

MITCHELL, T., KELLER, R., and S. KEDAR-CABELLI. 1986. Explanation-based generalization: A unifying view. Machine Learning **1**(1):47–80.

MITCHELL, D. 1984. Using features to evaluate positions in experts' and novices' othello games. Master's thesis, Department of Psychology, Northwestern University, Evanston, IL.

NEWELL, A. 1978. Harpy: Production systems and human cognition. Technical Report Technical Report CMU-CS-78-140, Carnegie-Mellon University.

NUNEZ, M. 1988. Economic induction: A case study. In *Proceedings of the Third European Working Session on Learning*. Glasgow, Scotland: Pitman. pp.139–145.

PAGALLO, G., and D. HAUSSLER. 1990. Boolean feature discovery in empirical learning. Machine Learning **5**(1):71–99.

PUGET, J. F. 1988. Learning invariants from explanations. In *Proceedings of the Fifth International Conference on Machine Learning*. Ann Arbor, MI: Morgan Kaufmann. pp.200–204.

QUINLAN, J. R. 1986. Induction of decision trees. Machine Learning **1**(1):81–106.

QUINLAN, J. R. 1993. C4.5: Programs for machine learning. Morgan Kaufmann.

RENDELL, L. 1985. Substantial constructive induction using layered information compression: Tractable feature formation in search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, volume 1. Los Angeles, CA: Morgan Kaufmann. pp.650–658.

ROSENBLOOM, P. 1982. A world-championship-level othello program. Artificial Intelligence **19**:279–320.

SACERDOTI, E. D. 1974. Planning in a hierarchy of abstraction spaces. Artificial Intelligence **5**(2):115–135.

SAMUEL, A. 1959. Some studies in machine learning using the game of Checkers. IBM Journal of Research and Development **3**(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**(6):601–617.

SCHLIMMER, J. C. 1987. Incremental adjustment of representations. In *Proceedings of the Fourth International Workshop on Machine Learning*. Irvine, CA: Morgan Kaufmann. pp.79–90.

SMITH, D., and M. GENESERETH. 1985. Ordering conjunctive queries. Artificial Intelligence **26**:171–215.

SUTTON, R. S. 1988. Learning to predict by the method of temporal differences. Machine Learning **3**(1):9–44.

TAN, M., and J.C. SCHLIMMER. 1991. Two case studies in cost-sensitive concept acquisition. In *Proceedings of the Eighth National Conference on Artificial Intelligence*. Boston, MA: Morgan Kaufmann. pp.854–860.

TESAURO, G., and T.J. SEJNOWSKI. 1989. A parallel network that learns to play backgammon. Artificial Intelligence **39**:357–390.

TESAURO, G. 1992. Temporal difference learning of backgammon strategy. In *Proceedings of the Ninth International Conference on Machine Learning*. San Mateo, CA: Morgan Kaufmann. pp.451–457.

UTGOFF, P. E., and J.A. CLOUSE. 1991. Two kinds of training information for evaluation function learning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*. Anaheim, CA: MIT Press. pp.596–600.

UTGOFF, P. E., and S. SAXENA. 1987. Learning a preference predicate. In *Proceedings of the Fourth International Workshop on Machine Learning*. Irvine, CA: Morgan Kaufmann. pp.115–121.

UTGOFF, P. E. 1986. Machine learning of inductive bias. Kluwer, Hingham, MA. Reviewed in IEEE Expert, Fall 1986.

WALDINGER, R. 1976. Achieving several goals simultaneously. Machine Intelligence. Wiley & Sons, New York. pp.94–136.

WARREN, D. 1977. Implementing prolog – compiling predicate logic programs. Technical Report 39 & 40, University of Edinburgh.

WARREN, D. 1981. Efficient processing of interactive relational database queries expressed in logic. In *Seventh International Conference on Very Large Data Bases*, pp.272–281.

YOUNG, P. 1984. Recursive estimation and time-series analysis. Springer-Verlag, New York.

LIST OF FIGURES