# Feature Discovery for Inductive Concept Learning

Tom E. Fawcett
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003 U.S.A.

COINS Technical Report 90-15
February 27, 1990

## Abstract

This paper describes Zenith, a discovery system that performs constructive induction. The system is able to generate and extend new features for concept learning using agenda-based heuristic search. The search is guided by feature worth (a composite measure of discriminability and cost). Zenith is distinguished from existing constructive induction systems by its interaction with a performance system, and its ability to extend its knowledge base by creating new domain classes. Zenith is able to discover known useful features for the Othello board game.

# 1 Introduction

One of the central concerns of machine learning is that of inductive concept learning from examples, in which a system is given a set of examples and produces a characterization of them. Many induction algorithms have been devised that are able to inductively generalize in different formalisms, using learning rules appropriate for that formalism. For example, decision tree algorithms (Quinlan, 1986) represent concepts implicitly as boolean expressions and split disjunctively on different attribute values at every branch point along a path. Linear threshhold units (Nilsson, 1965) represent the concept predicate as an inequality involving a threshhold and a weighted sum of example features. Many concept-learning systems (Michalski & Chilausky, 1980; Dietterich & Michalski, 1983) represent the concept as a boolean expression, but employ various different procedural biases that constrain the concept form.

Regardless of the formalism, all inductive concept learning methods are sensitive to the set of features used to describe the examples. This vocabulary usually determines not only the form and size of the final concept learned, but the speed of convergence of the learning method. In some cases, as with linear threshhold units, the learning method may never converge if the instance description vocabulary is inappropriate. Before a machine learning system can be used successfully, typically much human effort must go into selecting an appropriate instance description language. The problem of automatically generating or extending such a language has been called *constructive induction* (Michalski, 1983) and *shift of bias* (Utgoff & Mitchell, 1982).

There have been a number of approaches to this problem. Methods such as FRINGE (Pagallo, 1989) and CITRE (Matheus & Rendell, 1989) are concept-based, in that the examples are processed using an existing set of features, then new features are created from the induced concept. For example, FRINGE looks for binary path segments near the leaves of a decision tree, and creates a new feature for each segment. These binary segments represent binary conjunctions of terms. Once these new features are added, the learning algorithm is applied again to the examples and the process continues until no new useful features are created.

Other methods include defining new features by clustering attribute values (Schlimmer & Fisher, 1986; Michalski, 1983) and partitioning numeric attribute ranges (Schlimmer, 1987). Although these are able to change the representation by creating new features, they share the disadvantage that they only work well if the features are already appropriate to the task. These methods are also *passive* in that they do not take advantage of the learning context; they do not interact with a performance system that is using the features to provide classification.

Another approach is to use strong domain knowledge combined with information about the learning goal to guide the creation of new terms. The STABB component (Utgoff & Mitchell, 1982) of LEX used constraint back-propagation to extend its concept description language when it was found to be inadequate. STABB used the preimage of an operator to back-propagate conditions along a solution path; the resulting expression represented exactly a sufficient condition for the operator to lead to a solution.

This paper describes a discovery approach to generating and extending useful descriptive features for concept learning. This approach is motivated by the observation that, during

problem solving, humans discover features that are neither part of the problem specification nor simple combinations of features implied by the specification. A prototype system called Zenith has been implemented, and it has been applied to the Othello board game. It is able to discover features useful for Othello, given just the rules of the game and the ability to play it against an opponent. Zenith uses heuristic search guided by feature *worth* (a composite measure of discriminability and cost) to direct the feature generation process. Because it uses agenda-based heuristic search, it is able to invoke opportunistically a number of methods for feature creation, such as generalization, specialization and constraint back-propagation. Rather than extending the feature set only when it is recognized to be inadequate, Zenith tries continually to improve its set of features.

Zenith is able to attach tests to features in order to examine their performance in detail. The results of these tests provide feedback for proposing new features. Zenith can also augment its knowledge base by defining new classes of objects and creating new functions; although these additions do not directly affect task performance, they may serve as the basis for new features that will.

Section 2 describes the structure and organization of Zenith. Section 3 describes the Othello board game, the task domain. Section 4 presents Zenith's derivation of the "Corner square" feature. Section 5 discusses the derivation and the state of the implementation. Section 6 discusses future research directions.

# 2 System Architecture

## 2.1 Overview

The structure of Zenith is illustrated in Figure 1, and each component is described in detail below. The primary subsystems are the performance system, which plays Othello against an opponent using the current set of features; and the discovery system, which attempts to improve the current set of features using performance feedback. Since the overall goal of Zenith is to discover useful features rather than to play Othello, the performance system is controlled by the discovery system. Games are only played when the discovery system has created new features or has added tests to existing features, and needs feedback in order to continue.

Knowledge in Zenith is represented in a hierarchical network of frames. Each frame has a unique name and one or more slots, each of which has a value. For example, the goal of the performance system is represented by a frame, as are the Othello MOVE operator that defines the state space and the auxiliary predicates and functions used to express the rules of the game. Each feature generated by the system is also represented by a frame.

The features are used by a linear threshhold unit (LTU), one of the simplest learning algorithms. An LTU was chosen for several reasons. An LTU is very fast both to train and to use. Zenith requires a learning method that can accomodate the addition and deletion of features, and an LTU is one of the few methods that can. A hybrid method (such as STAGGER (Schlimmer, 1987)) is undesirable since it may confound limitations in the algorithm and limitations in the feature set. Since the representational limitation of an LTU (linearly discriminable sets) is well known, this confusion is avoided. Finally, the weights of

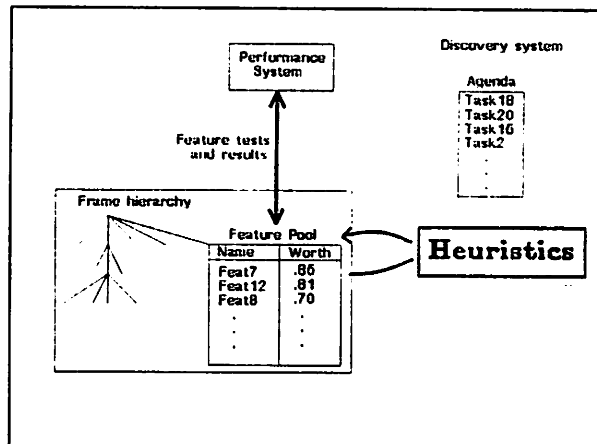


Figure 1: Zenith system structure

the LTU are used as an indication of the discriminability of each feature. In general, it is assumed that the higher the magnitude of a feature's weight, the more discriminating that feature is.[1]

## 2.2 Feature pool

The set of features known to the system are kept in a feature pool. A feature is a function from a state to a real number. Every feature is scaled so that its domain is between $-1$ and $+1$. A *binary* feature is a special kind of feature whose range is the set $\{-1, +1\}$. Binary features are used to express predicates. In Zenith every feature is represented as a frame which contains definitional information as well as performance data, including:

- The range of the feature.
- The weight of the feature as assigned by the LTU. This weight represents the utility of the feature in discriminating linearly between positive and negative examples.
- The average time cost of the feature.
- A composite "worth" score, based on the feature's weight and cost. In the current implementation this is defined simply as $weight/cost$.

Within the pool, each feature is ordered by its worth. Zenith's heuristics are biased to consider features of higher before lower worth. Although features are not discarded, if a feature's worth drops below a certain threshhold, it is declared inactive and is not used in the performance system's evaluation function.

[1]Although other measures could be adopted. The use of the LTU's weights for this purpose is not critical to the technique.

## 2.3 The performance system

### 2.3.1 Preference pairs and the move preference predicate

The performance system uses Zenith's features to play Othello. Its opponent is an Othello-playing program that searches to a fixed depth using a handcoded evaluation function. The performance system chooses its moves using a preference predicate (Utgoff & Saxena, 1987) based on the currently active features. A preference predicate $p$ is a generalization of an evaluation function, such that $p(S_1, S_2)$ is true iff state $S_1$ is better than $S_2$ for the performance system. A preference predicate is used rather than an evaluation function because it can be learned from qualitative information. That is, it needs only the information that the evaluation of one state should be greater than another, rather than the exact evaluations of the two states.

The performance system provides this qualitative information. After each game, the performance system examines the winner's move sequence and extracts *preference pairs* from it. Preference pairs are tuples $\langle S_1, S_2 \rangle$ such that state $S_1$ is better for the player than $S_2$. If the winner made a move resulting in $S_w$, then for every alternate move resulting in a state $S_a$ a preference pair $\langle S_w, S_a \rangle$ is generated. Each game generates approximately 250 such pairs.

### 2.3.2 Learning a move preference predicate

From these pairs, the preference predicate $p$ is learned as follows. If there are active features $f_1, f_2, \ldots, f_n$, then from the preference pair $\langle S_w, S_a \rangle$ the feature vectors

$$\begin{aligned} F_w &= \langle f_1(S_w), f_2(S_w), \ldots, f_n(S_w) \rangle \\ F_a &= \langle f_1(S_a), f_2(S_a), \ldots, f_n(S_a) \rangle \end{aligned}$$

may be generated. These vectors are subtracted giving the *feature delta vector*:

$$F_D = \langle f_1(S_w) - f_1(S_a),\ \ f_2(S_w) - f_2(S_a), \ldots,\ \ f_n(S_w) - f_n(S_a) \rangle$$

This delta vector $F_D$ is then a positive example for $p$, and its negation $-F_D$ is a negative example for $p$. Both examples are used to train the linear threshhold unit (LTU), which is then used in the performance system to choose its move at each turn.

The justification for using the delta vector is as follows[2]. Given that $S_w$ is preferred to $S_a$, a linear evaluation function would satisfy the inequality $\mathbf{W} \cdot F_w > \mathbf{W} \cdot F_a$, where $\mathbf{W}$ is the set of weights on the features. This equation can be rewritten as $\mathbf{W} \cdot (F_w - F_a) > 0$, and

$$\begin{aligned} \mathbf{W} \cdot F_D &> 0 \\ \mathbf{W} \cdot -F_D &< 0 \end{aligned}$$

The LTU is used to find the weight vector $\mathbf{W}$ that satisfies these equations. The LTU is trained after every game on the current set of preference pairs, until a plateau of performance is reached.

[2]A more detailed discussion may be found in (Utgoff, Saxena, Callan & Fawcett, 1989).

### 2.3.3 Selecting moves with the LTU

The LTU is then used by the performance system to choose its moves. The performance system first enumerates all successor states of the current state. The preference predicate learned by the LTU may not be consistent (if the features cannot linearly discriminate the preference pairs); therefore, a simple counting process determines the most preferred successor state. For each pair of states $\langle S_i, S_j \rangle$, the feature delta vector is presented to the LTU, and if the LTU classifies it as a positive example, $S_i$'s count is incremented. After this is done, the successor state with the highest count determines the performance system's move.

## 2.4 The Discovery System

The discovery system is the major part of Zenith, and its structure is based on that of AM (Lenat, 1983). The discovery system is agenda-driven and based on a set of heuristics. The agenda contains a set of tasks, ordered by priority[3]. The basic cycle of the discovery system is to remove the highest rated task from the agenda and execute it. Execution of a task involves checking all the relevant heuristics, and firing those that are applicable. The process of executing the heuristic may suggest new tasks to be considered, and these are entered onto the agenda.

### 2.4.1 Heuristics

A heuristic is simply an IF-THEN rule that is executed in a forward-chaining manner. Both IF and THEN parts can contain arbitrary LISP expressions. However, the IF part typically contains only simple tests to determine applicability, and expressions to bind variables to be used in the THEN part. If the IF part is satisfied, the THEN part executes. The THEN part can have any number of actions. Typically each action either creates a new frame, fills in the slot of an existing frame, or adds a new task to the agenda.

Zenith's heuristics are related to feature generation and testing, and can be separated into several categories:

**Goal transformation heuristics.** These produce an initial set of features from the goal expression, for use in the evaluation function, and so are only used once. These goal transformation heuristics currently used in Zenith are similar to those of (Callan, 1989). For example, if a goal is to achieve the relationship $f(s) < g(s)$, then one such heuristic will produce three separate features measuring $f(s)$, $g(s)$, and $f(s) - g(s)$.

**Feature specialization heuristics.** Specializing a feature can increase its discriminability by excluding negative instances. Test results can be used to guide this process. For example, in the derivation below a feature counting the number of White disks on a

[3]Task priority is similar to AM's task *worth*, although Zenith's process of computing and adjusting priorities is not nearly as elaborate as in AM. When a heuristic suggests a new task, its priority usually defaults to that of the current task. There is no complex formula for scaling down task priorities, and there is no bias toward continuing a line of effort.

board is specialized to a feature counting the number of White disks on corner squares; the latter is much more useful in predicting state value.

Another specialization heuristic is to conjoin two features that have many common positive instances but few negative instances. This is similar to the method used by STAGGER (Schlimmer, 1987) for conjoining features.

**Feature generalization heuristics.** These generalize or cluster a set of existing features, in order to increase generality and efficiency. For example, if one feature counts the number of corner squares owned by White, and another feature counts the number of White-owned squares neighboring a corner, then these two features could be generalized into one that counts the number of White spans emanating from a corner. This is a form of generalization-to-$n$; other forms, such as tree-climbing and condition-dropping generalization are used as well.

**Feature regression heuristics.** When a binary feature proves to be of high worth, it is often useful to look at the conditions under which it will become active. Feature regression heuristics produce new features by back-propagating the conditions of an existing binary feature through the domain operator(s). For example, given a feature that recognizes when a corner square is occupied, a regression heuristic can produce a feature that is activated when a corner square could be taken on the next move. Regressing the conditions of a feature allows the performance system effectively to extend its lookahead for high-worth features.

There is also a body of heuristics that perform bookkeeping tasks. For example, one heuristic activates the performance system when a number of new features have been created, or a number of new tests have been added to existing features.

# 3 The Othello board game

Othello[4] is a two-person game played on an $8 \times 8$ board. One player is White, one is Black. There are 64 discs colored black on one side and white on the other. The starting configuration is shown in Figure 2a. Black always moves first, with players alternating turns.

On a turn, the player can place a disk on any empty square that brackets a span of the opponent's discs ending in a disk of the player's own color. The span can be horizontal, vertical or diagonal. For example, in Figure 2b, Black could place a black disk on e2, f3, f4, f5, f6 or e6. When a player places a disk at the end of a span, all the discs in the span are *flipped* (changed to the player's color), and the player is said to own them. For example, if Black takes square f6 in Figure 2b, the board in Figure 2c would result. A player thus gains disks by either placing them or flipping disks of the other player. It is important to note that 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 most discs wins the game, and

[4]Othello is CBS Inc.'s registered trademark for its strategy disk game and equipment. Game board design ©1974 CBS Inc.

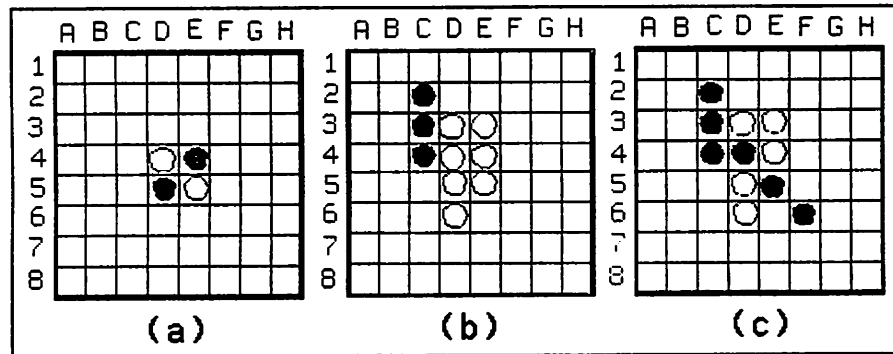


Figure 2: (a) Initial Othello board (b) Board in mid-game (c) After Black plays f6 on (b).

the number of points by which the player has won is simply the difference between the two piece counts.

As a machine learning domain, Othello is noise-free and deterministic (except for the opponent's moves), so experiments can be performed. The number of legal moves at every state is small enough (the average is about 7.5) to make it tractable to examine all successor states from a game state. However, there are about $10^{50}$ legal boards, so there is potentially a very large search space. Since there are about 60 moves in an Othello game, there is a significant credit/blame assignment problem; it is difficult to determine the effect of an individual move on the outcome of a game. Furthermore, the Othello move is geometrically simple but cannot easily be expressed symbolically, and the states are structurally complex. These problems make Othello intractable for existing discovery systems that depend on operators adding and deleting individual conditions (Scott & Markovitch, 1987; Shen & Simon, 1989), or that do not analyze state transitions (Lenat, 1983).

# 4 Derivation of a Corner Square Feature

One of the first things a beginning Othello player notices is the importance of occupying the corner squares. Recall that a disk is *stable* on a board if there is no sequence of subsequent moves that will flip it. Corners are inherently stable, since no span can ever encompass them, and other stable disk configurations can be built out from them. Noticing the importance of corner squares is a critical step in developing a winning strategy for Othello (Rosenbloom, 1982). However, the rules of the game do not mention stability, and they do not distinguish corner squares from other squares.

Initially, Zenith's frame network contains little more than the rules of the game. The Othello board is represented as a set of squares and a Neighbor relation defining them. Zenith is given the goal of winning the game, represented in the following predicate:

```
(DEFUN WIN (BOARD PLAYER)
  (AND (END-OF-GAME BOARD PLAYER)
       (> (COUNT-PIECES BOARD PLAYER)
          (COUNT-PIECES BOARD (OPPONENT PLAYER)))))
```

Zenith's goal transformation heuristics are applied to the WIN definition. Although Zenith can trivially convert any predicate into a binary feature, such features provide little information to direct search. Since the WIN predicate is a conjunction, it is broken apart and heuristics are reapplied to the conjuncts.

After several more steps, Zenith's heuristics have produced an initial set of twelve features, including FEATURE-4, defined as `(COUNT-PIECES BOARD 'WHITE)`. This feature counts the number of white pieces on a board, and was produced from the inequality within WIN. Since all of the features were generated from the goal expression, their values describe the degree to which the goal is satisfied by a state. Therefore, the goal transformation heuristics attach tests to these features that record fluctuations in their values.

Since no other features can be generated at this point, control is passed to the performance system. Initially, the weights on the features are random. After each game, the performance system extracts a set of preference pairs, and the LTU is trained on the entire set of pairs until it reaches a plateau of performance.

The performance system notices that FEATURE-4's values are non-monotonic; that is, in the course of a game its values both increase and decrease. Since FEATURE-4 was constructed directly from the goal, this is an "interesting" event. It indicates that the goal being measured is not protected; it is being "undone" by the opponent. Since FEATURE-4 counts a condition (the number of white discs), a heuristic suggests that the actual values — the squares involved — be recorded. A test is attached to FEATURE-4 to track these, and control is eventually passed back to the performance system.

After several more games no further test results are accrued, and they are analyzed. They are in the form of a set of tuples $\langle s_i; b_{i,1}, b_{i,2}, \ldots, b_{i,n} \rangle$ where each $s_i$ is a square and each $b_{i,j}$ is a board in which the test was satisfied (that is, the piece count decreased).

The data analysis is handled by another set of heuristics, each member of which is responsible for recognizing a different pattern in the data. One heuristic compares the set $S$ of all $s_i$ in the tuples with the domain $D$ of the test (the Square class), and notices that $S$ is a proper subset of $D$. That is, not every square can be involved in a piece count decrease. This corresponds to the observation that not every square on an Othello board can be flipped. The heuristic then splits the domain class (Square) into two subclasses: those that satisfied the test ($S$) and those that did not (the set difference of $D$ and $S$, called $T$). The latter class corresponds to the corner squares (which cannot be flipped once they are taken) and the non-corner squares. Finally, two new features are created as specializations of the original feature (FEATURE-4), one counting the White corner squares (FEATURE-13) and one counting the White non-corner squares (FEATURE-14).

Another heuristic notices that new features have been created, and creates tasks to re-train the LTU weights. Since there are no competing tasks, this is done immediately. After training, FEATURE-13 ends up with a very high weight, representing the high correlation of owning corner squares with winning the game.

# 5 Discussion

Zenith is implemented in Common Lisp, and generates the derivation described above. It is able to derive two well-known Othello features: corner square occupancy and piece mobility

(see (Rosenbloom, 1982) for a discussion of these). Zenith currently contains about fifteen heuristics, several from each of the classes given in Section 2.4.1. Feature regression heuristics have been designed but not yet implemented.

Several additional feature discoveries are possible as extensions of the derivation above, but have not yet been implemented. Zenith can create several new features based on the corner squares. Since it has the concept of stability, by looking at related squares it can discover stable board configurations and generalize them to a concept of edge stability. By specializing FEATURE-13 into binary features and then regressing them through the MOVE operator, it can discover the X squares (b2, g2, b7 and g7). X-squares are dangerous to own because, once taken, it is very difficult to prevent the opponent from taking the corner.

# 6 Future Work

This goal of this research is to provide an architecture and a set of general heuristics that can derive useful features in a range of domains, given only basic knowledge about each domain. This work is still preliminary, but Zenith is able to discover a number of useful Othello features and domain concepts. As with any discovery system, evaluating this approach involves both characterizing the space being searched and analyzing the effectiveness of the heuristics in focusing the search.

Central to Zenith's approach is the use of individual feature performance to guide new feature creation, and this introduces two weaknesses. First, it is a hillclimbing method, and it has the well-documented problems of such techniques (local maxima, plateaus). This could be ameliorated by incorporating a factor similar to AM's *interestingness* into feature worth, such that if a feature is sufficiently interesting it need not be immediately useful in discriminating states. This would provide for the survival of features that capture interesting domain categories but are either not yet cheap enough (and must be simplified or optimized), or not yet accurate enough (and must be specialized).

We are currently experimenting with an interestingness heuristic based on the delta vectors mentioned in Section 2.3.2. A zero delta vector indicates that the current set of features were completely unable to distinguish between two states, one of which was a preferred state. When a zero delta vector is encountered by the performance system, it is recorded along with the states that produced it. Whenever a new feature is produced by the discovery system, the set of zero delta vector states is examined. If the feature produces a non-zero delta, it is marked as inherently interesting, since it is able to distinguish at least two states that no other feature can.

The second implication of Zenith's concentration on individual feature performance is that it may be misled about a feature's worth. For example, two features in conjunction may discriminate much more effectively than either alone could. Ideally a feature should be evaluated in the context of the entire feature pool. Recent advances in evaluating representation quality (Saxena, 1989) have produced algorithms that are able to accomplish this, and we expect to replace the current evaluation method and eliminate this problem.

# 7 Acknowledgements

This material is based on work supported by the Office of Naval Research through a University Research Initiative Program under contract number N00014-86-K-0764. Useful comments were provided by Carla Brodley, Jamie Callan, Margie Connell, David Haines, David Lewis, Sharad Saxena and Paul Utgoff,

# 8 References

Callan, J. P. (1989). Knowledge-based feature generation. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 441-443). Ithaca, NY: Morgan Kaufmann.

Dietterich, T., & Michalski, R. (1983). A comparative review of selected methods for learning from examples. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Lenat, D. (1983). The role of heuristics in learning by discovery: Three case studies. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Matheus, C. J., & Rendell, L. A. (1989). Constuctive induction on decision trees. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 645-650). Detroit, Michigan: Morgan Kaufmann.

Michalski, R. S., & Chilausky, R. L. (1980). Learning by being told and learning from examples: An experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *Policy Analysis and Information Systems, 4*, 125-160.

Michalski, R. S. (1983). A theory and methodology of inductive learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Nilsson, N. J. (1965). *Learning Machines.* New York: McGraw-Hill.

Pagallo, G. (1989). Learning DNF by decision trees. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 639-644). Detroit, Michigan: Morgan Kaufmann.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*, 81-106.

Rosenbloom, P. (1982). A world-championship-level othello program. *Artificial Intelligence, 19*, 279-320.

Saxena, S. (1989). Evaluating alternative instance representations. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 465-468). Ithaca, NY: Morgan Kaufmann.

Schlimmer, J. C., & Fisher, D. (1986). A case study of incremental concept induction. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 496-501). Philadelpha, PA: Morgan Kaufmann.

Schlimmer, J. C. (1987). Incremental adjustment of representations. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 79-90). Irvine, CA: Morgan Kaufmann.

Scott, Paul D., & Markovitch, Shaul (1987). Learning Novel Domains Through Curiosity and Conjecture. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 669-674). Milan, Italy: Morgan Kaufmann.

Shen, Wei-Min, & Simon, Herbert (1989). Rule creation and rule learning through environmental exploration. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 675-680). Detroit, Michigan: Morgan Kaufmann.

Utgoff, P. E., & Mitchell, T. M. (1982). Acquisition of appropriate bias for inductive concept learning. *Proceedings of the Second National Conference on Artificial Intelligence* (pp. 414-417). Pittsburgh, PA: Morgan Kaufmann.

Utgoff, P. E., & Saxena, S. (1987). Learning a preference predicate. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 115-121). Irvine, CA: Morgan Kaufmann.

Utgoff, P. E., Saxena, S., Callan, J. P., & Fawcett, T. E. (1989). *Representation problems in machine learning: a proposal* (COINS Technical Report 89-23). Amherst, MA: University of Massachusetts, Department of Computer and Information Science.