Automatic Feature Generation for Problem Solving Systems

Tom E. Fawcett
Paul E. Utgoff
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

COINS Technical Report 92-9
January 1992

Abstract

Existing methods for constructive induction usually isolate feature generation from problem solving, and do not exploit information about the purpose for which features are created. This paper describes a theory of feature generation that creates features using both a domain theory and feedback from a concept learner. An evaluation function can then be learned using these features that is able to direct a problem-solver. The theory has been implemented in a system called Zenith, which has been applied to two domains. Zenith is able to generate useful features for each domain, given only a domain theory and the ability to solve problems in the domain.

Automatic Feature Generation for Problem Solving Systems

Tom E. Fawcett
Paul E. Utgoff
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

COINS Technical Report 92-9
January 1992

Abstract

Existing methods for constructive induction usually isolate feature generation from problem solving, and do not exploit information about the purpose for which features are created. This paper describes a theory of feature generation that creates features using both a domain theory and feedback from a concept learner. An evaluation function can then be learned using these features that is able to direct a problem-solver. The theory has been implemented in a system called Zenith, which has been applied to two domains. Zenith is able to generate useful features for each domain, given only a domain theory and the ability to solve problems in the domain.

1 Introduction

In his pioneering work in artificial intelligence, Arthur Samuel (1959) developed a program that was able to play the board game checkers. Samuel's program used a set of features to characterize board positions, and 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. Although later work [Samuel, 1963] improved on the method for combining features, they still had to be designed and hand-coded by a human expert. Samuel identified the automatic construction of such features as a major open problem of great importance.

The problem has remained open for over 30 years. In that time, many game-playing programs have been written that achieve expert levels of performance, matching and sometimes exceeding the abilities of humans [Berliner, 1980; Rosenbloom, 1982; Lee & Mahajan, 1988]. Like Samuel's checker playing program, some of these systems are able to tune their evaluation functions automatically; however, all of them depend upon hand-coded features to describe problem states. Until now, no research has addressed the automatic generation of such features, and no theory yet exists of how handcoded features are discovered.

Some methods have been developed that are able to generate new features for inductive concept learning [Schlimmer & Granger, 1986; Matheus & Rendell, 1989; Pagallo & Haussler, 1990]. However, these methods are generally not applicable to problem solving systems because they isolate concept learning from problem solving. They do not exploit information about the goals, constraints and operators of the domain, nor do they use feedback from a problem solver to guide feature generation. If the purpose of learning is to improve problem solving performance, such sources of information should not be ignored.

Existing methods of constructive induction attempt to create useful features by combining the primitive features in various ways. However, inspection of features designed by humans [Rosenbloom, 1982; Mitchell, 1984] reveals that most are not combinations of the primitive features of the domain, but rather they reflect the goals and operators of the domain for which they were designed. They measure the degree of achievement of important goals and subgoals, both strategic and tactical, known to be significant. In the terminology of Flann and Dietterich (1986), the features are *functional* rather than *structural* in nature.

STABB [Utgoff, 1986] uses problem solving knowledge for the generation of new features. However, it is limited to so-called *tractable* domains [Mitchell, Keller & Kedar-Cabelli, 1986], in which complete explanations of solution paths can be generated, because it is from these explanations that the features are created. Features for intractable domains must be heuristic rather than

logically sufficient.

This paper presents a theory of feature generation for concept learning in problem solving systems. Section 2 describes our theory in general terms. The theory has been implemented in a system called Zenith, described in Section 3. Zenith has been applied to two domains: the board game OTHELLO, and the domain of telecommunications network management. Section 4 describes these domains and Zenith's feature derivations for them, demonstrating that Zenith is able to create useful features and to improve performance in both domains. Section 5 discusses the results and concludes.

2 Components of a Theory of Feature Generation

Zenith is based on a transformational theory of feature generation [Fawcett & Utgoff, 1991], in which features are generated by the controlled application of transformations to other features. We have identified four classes of transformations: decomposition, abstraction, regression and specialization. Each class comprises one or more specific transformations in Zenith. Section 3 describes both these specific transformations and a strategy for controlling their application.

A feature is represented as a conjunction or disjunction of first-order terms. These terms are called the *conditions* of the feature. A feature is evaluated with respect to a problem solving state by determining whether the conditions are satisfiable in the state.

The system starts with a single feature created automatically from the performance goal of the system, and then progressively develops a set of features by successively refining the existing features through application of the transformations. Each of the four transformation classes performs a different function and interacts with the others.

2.1 Decomposition

Decomposition transformations are syntactic methods for splitting apart features. Each decomposition transformation recognizes a specific form, such as an arithmetic inequality, that can be split apart and made into new features. Callan and Utgoff (1991) show how decomposition results in features that may produce a more useful gradient than would the original feature, and thus may be more effective at guiding the problem solver.

For example, one decomposition transformation looks for arithmetic inequalities like $f(X) > g(Y)$ and produces two new features that calculate $f(X)$ and $g(Y)$ separately. Being able to measure these values separately may give the system information about how close the problem solver is to satisfying the original inequality. As an example, the rules of OTHELLO 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 measuring the number of discs of the player, and the other measuring the

number of discs of the opponent. Measuring these two values independently is more useful to the concept learner, and thus to the performance system, than is the original inequality.

2.2 Goal regression

Goal regression creates the pre-image of a condition with respect to an operator. If a feature is useful for the concept learner, then it may be beneficial to create other features that measure how the original feature is affected by the domain operators. For example, 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 condition through the move operator.

Goal regression has been used in many other machine learning systems [Silver, 1986; Utgoff, 1986; Minton, 1988], but there are two significant difference to the goal regression in this theory. First, whereas goal regression is usually performed along an entire operator path, in this theory it is only applied over a single operator step at a time. This allows it to be used in intractable domains. Second, whereas goal regression is usually applied to the goal conditions, in this theory it can be applied to the conditions of any feature.

2.3 Abstraction

Features are often created (for example, by goal regression) that could contribute much to the learned concept but are not worth their computational cost. Abstraction removes details from a feature, which has two benefits. It may make a feature less expensive, and it may also enable further simplification and specialization.

The abstraction component must determine which expressions in a feature are details that can be removed without sacrificing too much accuracy. To do this, it must have a policy for determining how critical each expression is. Zenith uses a half-order theory of criticality, adopted from ABSTRIPS [Sacerdoti, 1974]. This theory is domain independent and comprises a small set of rules that assigns criticality to each condition of a feature. These rules are based on based on ABSTRIPS's principles of how easily the problem solver can achieve the condition.

As an example from OTHELLO, the definition of a move for the white player is a line of squares consisting of a white piece adjacent to one or more black pieces, terminated by a blank square. Of these three conditions, the easiest for the white player to influence is the existence of a white piece, because the white player can directly place white pieces on the board, but can neither place black pieces nor create blank squares. Therefore, a more abstract feature could be produced by removing the first condition of the OTHELLO move. The resulting expression matches a pattern consisting of one or more black pieces terminated by a blank square, and is the basis for several mobility features

used by Rosenbloom (1982). Empirically, these features are cheaper to evaluate than the original conditions of the OTHELLO move and preserve most of their accuracy.

2.4 Specialization

The fourth class of transformations creates specializations of existing features. A specialized feature can often be much less expensive than the original feature.

One way of specializing a feature is to search for *invariants* of the feature; that is, individual domain elements or configurations of domain elements that always satisfy the conditions of the feature. For example, a feature may be expensive because it tests a large set of domain elements to determine which of them satisfy an expensive condition. If special cases can be found that always satisfy the condition, a feature can be created to recognize just these special cases. The resulting feature will usually be much cheaper and will maintain most of the accuracy of the original.

A second specialization technique is that of extracting a specific explanation used by a feature. Because features may use non-operational terms that are defined by rules in the domain theory, this type of specialization involves extracting a specific highly-used rule path and creating a new feature that uses only that path. By checking only this one path, the feature may be substantially less expensive than the original feature. This technique is similar to explanation-based learning [Mitchell, Keller & Kedar-Cabelli, 1986].

2.5 Summary

The decomposition and goal regression components 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.

3 The Zenith System

The Zenith system is based on the theory in the previous section. Zenith's architecture is similar to that of Samuel's (1963) first checkers player, with the addition of a feature construction module. Note that the system is provided only with the initial problem specification, after which the system learns autonomously.

Zenith is cyclic: it solves a problem using its current evaluation function, extracts instances from the problem solving episode, creates new features, then learns a new evaluation function to be used in the next cycle.

The problem solver pursues a goal by performing a state-space search; it generates a set of successor states and uses the learned concept to determine the best next state. The purpose of feature generation is to aid the concept

learner in distinguishing states that lead to a more desirable goal state.

3.1 The Concept Learner

When the performance component completes a problem-solving episode, the solution path is passed to a critic, which creates training instances from it. These instances are then used by the concept learner to induce a general concept that can be used as an evaluation function to direct search. The concept takes the form of a preference predicate [Utgoff & Clouse, 1991] that determines when one state is preferred to another.

In the implementation, a linear threshold unit (LTU) is used as a concept learner, because it provides a fast, simple method for combining the influences of the features. It is possible that a nonlinear combination of features would produce in some cases a more accurate evaluation function. Further research will include experiments to determine the sensitivity of system performance to the concept form.

Given the set of all features constructed so far, the system must decide which should be included in the evaluation function, and which should not. A time limit on the evaluation function is provided externally as part of the problem specification. The system must create an evaluation function that is as accurate as possible but that does not exceed the time limit in evaluating a state.

Determining the optimal subset of features via exhaustive search is prohibitively expensive, so Zenith uses a *sequential backward selection* method [Kittler, 1986]. The method begins by creating a feature set consisting of *all* features that have been generated so far. It then removes any feature whose cost exceeds the time limit. The remaining set is used to train an evaluation function. Individual features are then cast out repeatedly until the total cost of the evaluation function falls below the imposed limit. The feature to be cast out is the one that contributes least to concept accuracy, that is, the feature that produces the smallest decrease in concept accuracy when removed from the set. Ties are broken by selecting the feature with greater cost. The process halts when the total cost of the set of features is below the imposed time limit. The remaining features are used in the evaluation function, and are termed *active*. The features that have been cast out are termed *inactive*.

Note that every time new features are generated the concept learner re-evaluates the entire set to determine which should be included. Thus, due to changes in the feature set, a feature that is inactive may become active at a later time.

3.2 Features and Transformations

Features are expressed in a form similar to that used by Michalski's (1983) counting arguments rules. A feature comprises a set of conditions (a conjunction or disjunction) and a set of variables. The feature computes the number

Class	Name	English description
Decomposition	split conjunction	Split conjunction into independent parts
	remove negation	Replace $not(X)$ with X
	split arith comp	Split arithmetic comparison into constituents
	split arith calc	Split arithmetic calculation
Abstraction	remove LC condition	Remove least constraining condition of conjunction
Goal Regression	regress condition	Regress condition of a feature through a domain operator.
Specialization	variable specialize	Find invariant variable values that satisfy a feature's conditions
	remove disjunct	Remove a feature's disjunct
	expand base case	Replace call to recursive predicate with base case.

Table 1. Implemented transformations in Zenith

of distinct values of the set of variables that can satisfy the conditions. A feature may be seen as a generalized form of a boolean expression, in that it calculates not just whether the conditions are satisfiable but the number of ways in which the conditions can be satisfied. This representation is more expressive than a set of conditions alone because it allows a feature to calculate not just whether the conditions are satisfied, but the number of ways in which the conditions can be satisfied.

The catalog of transformations currently used in Zenith is shown in Table 1. A transformation applies to a feature and creates one or more new features from it, without losing the original feature.

A single goal regression transformation is used that produces new features by regressing the conditions of an existing feature through a domain operator. A separate feature is created for every resulting pre-image. A general abstraction transformation is used, which removes the least critical condition of a feature using the theory of criticality mentioned in Section 2. In addition to the specialization transformations discussed in Section 2, there is a third, expand base case, which replaces a call to a recursive predicate with its base case.

The system must determine how and when to apply the transformations. Features are selected one at a time, and transformations are chosen according to the following policy:

1. If the feature can be decomposed, it is, and no other transformations are allowed to apply to the feature. Because they generate features that usually produce a more refined gradient than the original feature, any subsequent transformations should be done to the resulting features rather than to the original.

2. Only if a feature is active will goal regression be applied to it. Goal regression usually produces features that are expensive, so it is only applied to features that have already proven their worth. If a feature

is inactive, it is unlikely that a feature recognizing its pre-image would have a greater contribution to concept accuracy.

3. If the feature is judged to be expensive, abstraction and specialization transformations are applied to it. An expensive feature is defined empirically as one whose cost exceeds 10% of the evaluation function limit.

4 Zenith's Feature Derivations

Zenith has been applied to two domains: the board game of OTHELLO, and telecommunications network management (TNM). This section focuses on the process by which Zenith creates features in both of these domains. A previous paper [Fawcett & Utgoff, 1991] discussed an earlier implementation of Zenith, and showed that Zenith's features were able to attain about 85% classification accuracy on an independent test set of OTHELLO instances.

4.1 OTHELLO

OTHELLO is a two-player game played on an 8×8 board. Players alternate moves, and there are usually 60 moves in a game. The search space contains approximately 10^{50} nodes. OTHELLO was chosen as a domain rather than checkers because many features are known for it [Mitchell, 1984]. OTHELLO is also attractive because it has been studied previously by researchers in artificial intelligence [Rosenbloom, 1982; Lee & Mahajan, 1988]. Zenith's opponent is an expert OTHELLO-playing program. The OTHELLO domain theory is available from the UCI database.

Figure 1 illustrates the derivation of some of the OTHELLO features that Zenith generated. Discussion of many of these features may be found in [Rosenbloom, 1982].

The domain theory for OTHELLO specifies the performance goal as `win(black)`. From this, an initial feature is created that measures whether the black player has won in a state. This feature is useless for directing search, but decomposing it yields features measuring the score for each player, and the existence of a legal move.

By regressing the conditions of Score (on the left-hand side of Figure 1) through the OTHELLO move operator, Zenith generated a feature that counts the number of semi-stable squares. Semi-stable squares are those squares that cannot be immediately acquired by the opponent. By removing a negation in this feature, Zenith generated the Axes feature, which measures the total potential for flipping opponent's pieces on the board.

The number of semi-stable squares is a useful feature of a state to measure, but it is expensive to test every square for semi-stability. Zenith specialized this expensive feature by looking for squares that are semi-stable; that is, by looking for squares that invariably satisfy the semi-stability property.[1] Because

[1] Proving invariance analytically is usually expensive and sometimes impossible. Instead,

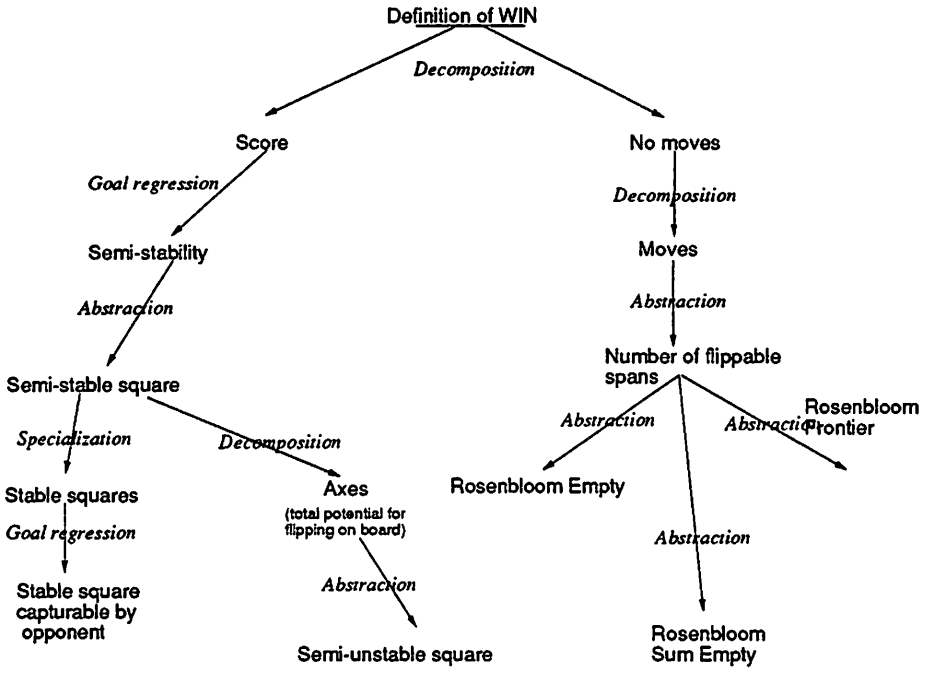

Figure 1. The derivation of some OTHELLO features generated by Zenith.

of the geometry of the OTHELLO board, the four corner squares are invariantly stable, but Zenith found about twenty squares to be frequently stable and created a feature for this set (of which the corner squares were members). The 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 made stable on the next move. This feature is related to such commonly known features as X Squares and C Squares.

Zenith also created a number of mobility features, shown on the right-hand side of Figure 1. From the initial feature, Zenith created a feature that measured the number of moves available. By expanding the definition of the OTHELLO move, a new feature was created that counted the number of squares involved in each move. By applying its abstraction transformation to this repeatedly, it created a number of simpler, less costly mobility features including three used by Rosenbloom (1982).

Zenith also generated features not shown in Figure 1. For example, by applying the regress-condition transformation to the Moves feature shown on the right-hand side of the figure, Zenith created a *future mobility* feature that detected moves that could become available in the next state. This feature was valuable to the performance element and was assigned a substantial weight by the LTU. To our knowledge, this is an original discovery; no such feature has

Zenith finds invariants of features empirically by sampling its database of training instances.

been published in any of the OTHELLO literature with which we are familiar.

4.2 Telecommunications Network Management

Zenith's second domain is that of telecommunications network management. A telecommunications network is a circuit-switched network in which calls are placed from one switch to another. Calls are routed in a distributed manner by the switches. Due to problems such as traffic congestion and equipment failure, calls may fail to complete causing network performance to degrade. A set of controls can be imposed on each switch to modify its routing behavior. The task of telecommunications network management (TNM) is to impose these controls so as to maximize the performance of the network according to some metric. Viewed as a state-space search, a state transition is the application of a control to a switch in the network. Zenith's task is to generate features to direct the problem solver in placing controls on the network, in order ultimately to improve network performance.

Several learning systems exist for TNM (Silver, et al. 1990), one of which creates decision trees from instances using the primitive (observable) features, but feature generation for this domain has not yet been explored. In contrast to OTHELLO, there is no body of features, either human or machine generated, against which Zenith's features can be compared.

Zenith does not perform a problem-solving episode in every cycle, but instead solves a set of four problems initially, from which it extracts 663 training instances. Each of the four represents a different network problem. Two controls were used: one that could reroute calls to a particular destination, and one that could block calls. A control can be applied to any switch in the network, and only alters the routing behavior of the switch to which it is applied. Combinations of these controls can effect large changes in routing behavior.

The performance goal of Zenith for TNM is to maximize the *throughput* of the network, defined as the number of calls completed divided by the number of calls placed. Zenith's initial feature was based on this goal, and was sufficient to classify the training instances with 80% accuracy. Zenith decomposed the feature into two features measuring the number of calls completed and the number of calls placed. Zenith regressed the conditions of a completed call through the reroute operator to yield a feature that measured a state's *rerouting potential*: the amount of traffic that could be successfully rerouted. This was a valuable but expensive feature, so it was transformed using specialization and abstraction into several other features, most of which were cheaper. Applications of expand base case yielded a feature that measured traffic that could be rerouted from one switch away. A number of useless features were also generated, such as one that measured the amount of traffic that would travel through a path if there was no traffic to use that path. However, most of the features were useful for discriminating the instances, and Zenith was

able to classify accurately 91% of an independent test set of instances.

5 Discussion and Conclusion

This paper presents a theory of feature generation for inductive concept learning. The theory comprises four general kinds of actions — decomposition, goal regression, abstraction and specialization — useful for generating features for a problem-solver. The Zenith system, an implementation of this theory, is based on a transformational model in which new features are created by transforming existing ones. Section 4 showed that Zenith is able to generate useful features in two very different problem-solving domains.

Because many features are known for OTHELLO, we have concentrated on that domain in evaluating our implementation. It is important to note that Zenith currently cannot generate some of the known OTHELLO features. For example, some features (Kierulf Weights, Corner Points) calculate the weighted sum of combinations of particular occupied squares. Zenith cannot create features that incorporate weights, although its LTU does assign weights to each feature. Other features (Exhaustive Moves, Move Levers) are based on variable-length sequences of problem-solving steps, rather than independent moves from a single state. By repeatedly regressing conditions Zenith can generate the preconditions of an operator sequence. However, Zenith does not reason about variable-length sequences of states, and so cannot produce such features. Future work should make clearer such limitations.

Zenith is an attempt to answer the question raised by Arthur Samuel over 30 years ago, *how can a system generate useful features automatically?* Specifically, it addresses the problem of feature generation for problem-solving systems, an area that has received little attention in constructive induction. The ultimate goal of this work is to generate, for any domain, features that are as good as those created by humans. By automating this process, we will increase the autonomy of learning systems and thereby make it less costly to build and maintain problem-solving systems.

Acknowledgements

This research was supported 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. Comments by Bernard Silver, Jamie Callan, Carla Brodley and Jeff Clouse improved the presentation.

References

Berliner, H. J. (1980). Backgammon computer program beats world champion. *Artificial Intelligence, 14*, 205-220.

Callan, J. P., & Utgoff, P. E. (1991). Constructive induction on domain knowledge. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 614-619). Anaheim, CA: MIT Press.

Fawcett, T. E., & Utgoff, P. E. (1991). A hybrid method for feature generation. *Machine Learning: Proceedings of the Eighth International Workshop* (pp. 137-141). Evanston, IL: Morgan Kaufmann.

Flann, N. S., & Dietterich, T. G. (1986). Selecting appropriate representations for learning from examples. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 460-466). Philadelphia, PA: Morgan Kaufmann.

Kittler, J. (1986). Feature selection and extraction. In Young & Fu (Eds.), *Handbook of pattern recognition and image processing.* New York: Academic Press.

Lee, K. F., & Mahajan, S. (1988). A pattern classification approach to evaluation function learning. *Artificial Intelligence, 36*, 1-25.

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

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

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

Mitchell, D. (1984). *Using features to evaluate positions in experts' and novices' othello games*, (Masters thesis), Evanston, IL: Department of Psychology, Northwestern University.

Mitchell, T, Keller, R, & Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning, 1*, 47-80.

Pagallo, G., & Haussler, D. (1990). Boolean feature discovery in empirical learning. *Machine Learning*, 71-99.

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*, 115-135.

Samuel, A. (1959). Some studies in machine learning using the game of Checkers. *IBM Journal of Research and Development, 3*, 211-229.

Samuel, A. (1963). Some studies in machine learning using the game of Checkers. In Feigenbaum & Feldman (Eds.), *Computers and Thought.* New York: McGraw-Hill.

Schlimmer, J. C., & Granger, R. H., Jr. (1986). Incremental learning from noisy data. *Machine Learning, 1*, 317-354.

Silver, B. (1986). Precondition analysis: Learning control information. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Silver, B., Frawley, W., Iba, G., Vittal, J., & Bradford, K. (1990). ILS: A framework for multi-paradigmatic learning. *Proceedings of the Seventh International Conference on Machine Learning* (pp. 348-356). Austin, TX: Morgan Kaufmann.

Utgoff, P. E. (1986). Shift of bias for inductive concept learning. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Utgoff, P. E., & Clouse, J. A. (1991). Two kinds of training information for evaluation function learning. *Proceedings of the Ninth National Conference on Artificial Intelligence* (pp. 596-600). Anaheim, CA: MIT Press.