Constructive Induction on Domain Information*

James P. Callan and **Paul E. Utgoff**

Department of Computer and Information Science,
University of Massachusetts · Amherst, Massachusetts 01003
callan@cs.umass.edu, utgoff@cs.umass.edu

Abstract

It is well-known that inductive learning algorithms are sensitive to the way in which examples of a concept are represented. Constructive induction reduces this sensitivity by enabling the inductive algorithm to create new terms with which to describe examples. However, new terms are usually created as functions of existing terms, so an extremely poor *initial* representation makes the search for new terms intractable.

This work considers inductive learning within a problem-solving environment. It shows that information about the problem-solving task can be used to create terms that are suitable for learning search control knowledge. The resulting terms describe the problem-solver's progress in achieving its goals. Experimental evidence from two domains is presented in support of the approach.

Introduction

One reason that inductive learning algorithms are not more widely used in problem-solving systems is their sensitivity to the representation of information. Some of the best-known successes in machine learning have been due in part to careful or fortunate choices of representations (e.g. AM [Lenat & Brown, 1984]). When the same algorithms are applied with less-carefully chosen vocabularies, they may fail to learn anything useful.

The difficulty of constructing an appropriate vocabulary is sometimes underestimated, because many tasks appear to have obvious representations. For example, one might expect to describe a chess board by listing the location of each piece. However, obvious representations are sometimes inadequate. Quinlan spent two person-months designing a vocabulary that enabled ID3 to classify certain chess positions [Quinlan, 1983]. In general, constructing a good vocabulary 'by hand' can take a long time, because it is essentially a manual search of the space of possible vocabularies.

One solution, known as *constructive induction*, is to enable the learning program to construct new terms for its vocabulary. Most constructive induction algorithms define new terms as Boolean or arithmetic functions of previously known terms. There are infinitely many such functions, so attention must be restricted to a small subset. Most systems search the space of possible vocabularies heuristically, using either the structure of the evolving concept (e.g. CITRE [Matheus & Rendell, 1989]) or the behavior of the learning algorithm (e.g. STAGGER [Schlimmer & Granger, 1986]) to guide search. In both cases, an extremely poor *initial* vocabulary makes the search intractable.

This paper addresses the problem of generating an initial vocabulary for inductive learning. The problem arises, and is naturally addressed, when inductive learning is considered in the context of a problem-solving system. Many problem-solving methods, for example Hillclimbing, require an evaluation function that maps each search state to a number. Such a function is not normally provided with a problem specification. This paper shows that available domain knowledge can be converted to a set of numeric features, for which an evaluation function can be trained via existing methods. It is assumed that such features will be relevant to the domain, because they are derived directly from domain knowledge.

Feature Construction

There is wide variation in what is considered domain knowledge. It is desirable to require a minimum of information, so that the resulting methods are widely applicable. This work requires that the domain knowledge describe, in first order predicate calculus, the problem-solving operators and the goal state. First order predicate calculus was chosen because it is general and well-known.

The specification of the goal state is neither the desired evaluation function nor a good vocabulary for learning the desired evaluation function, because it does not distinguish among states that are not goals.

*This research was supported by a grant from the Digital Equipment Corporation, and by the Office of Naval Research through a University Research Initiative Program, under contract N00014-86-K-0764.

One approach to creating a better vocabulary is to increase its resolution by decomposing the description of goal states into a set of functions, each of which maps search states to feature values. The additional functions, or *features*, may each change value at a different point, thereby making explicit the problem-solver's progress toward a goal. If this decomposition can be applied recursively, resolution is further increased. Ideally, the problem-solver's progress from state to state is reflected in a change of at least one feature value.

Statements in first order predicate calculus may or may not be quantified, so it makes sense to develop some transformations that operate only upon Boolean expressions, and other transformations that operate only upon quantifiers. A set of four transformations, developed as part of this research and defined below, is sufficient to cover each type of Boolean expression and each type of quantifier.

- *LE:* applies to Boolean expressions in which a logical operator ($\wedge, \vee, \neg$) has the highest precedence.
- *AE:* applies to Boolean expressions in which an arithmetic operator ($=, \neq, <, \leq, >, \geq$) has the highest precedence.
- *UQ:* applies to universally quantified expressions.
- *EQ:* applies to existentially quantified expressions.

LE, *AE*, and *UQ* transformations are presented below. An *EQ* transformation is under development.

The *LE* Transformation

The *LE* transformation applies to Boolean expressions in which a logical operator ($\wedge, \vee, \neg$) has the highest precedence. Its purpose is to transform a Boolean function into a set of Boolean functions, each of which represents some component of the original function. The resulting functions, called *LE* terms, map search states to Boolean values. *LE* terms are sometimes also called *subgoals*, or *subproblems* [Callan, 1989], because they represent part of the original problem specification.

The *LE* transformation creates new statements by eliminating the n-ary logical operator with the highest precedence. Its definition is:

$$LE\left((Q_i)^m(\vee_{j=1}^n b_j)\right) \Rightarrow \{(Q_i)^m b_1, \ldots, (Q_i)^m b_n\}$$
$$LE\left((Q_i)^m(\wedge_{j=1}^n b_j)\right) \Rightarrow \{(Q_i)^m b_1, \ldots, (Q_i)^m b_n\}$$

Q_i is one of m quantifiers. It matches either $\forall v_i \epsilon S_i$ or $\exists v_i \epsilon S_i$. b_j is one of n Boolean statements. After the new Boolean statements are created, they are optimized by removing unnecessary quantifiers.

For example, the statement $\exists a \epsilon A, \exists b \epsilon B, \exists c \epsilon C, p_1(a,b) \wedge p_2(a,c)$ is transformed into statements $\exists a \epsilon A, \exists b \epsilon B, p_1(a,b)$ and $\exists a \epsilon A, \exists c \epsilon C, p_2(a,c)$.

The statement to which the *LE* transformation is first applied is the specification of a goal state. The resulting statements can be considered subgoals. The *LE* transformation can be applied recursively, yielding subgoals of subgoals. If the transformation is applied wherever possible, the result is a subgoal hierarchy, with the goal at the root and atomic statements at the leaves. In general, a subgoal hierarchy contains between $c+1$ and $2c$ subgoals, where c is the number of connectives in the specification of the goal state.

The *LE* transformation explicitly represents subgoals, but it does not explicitly represent the dependencies among them. This characteristic occurs because variable bindings are not forced to be consistent across all subgoals. For example, the statement $\exists a \epsilon A, p_1(a) \wedge p_2(a)$ is decomposed into statements $\exists a \epsilon A, p_1(a)$ and $\exists a \epsilon A, p_2(a)$. The conjunction of the two statements is $\exists a \epsilon A, p_1(a) \wedge \exists a \epsilon A, p_2(a)$, which is *not* equivalent to the original statement. Thus the subgoals are represented as independent, when in fact they are interdependent. This characteristic is acceptable for two reasons. First, the dependency information remains available, although implicit, in the statement from which the subgoals were generated (i.e. subgoals that occur higher in the subgoal hierarchy). Second, given a set of *LE* terms that explicitly describe subgoals, the inductive learning algorithm can infer the existence of, and then represent explicitly, those dependencies necessary to learn the concept.

The *AE* Transformation

The *AE* transformation applies to Boolean expressions in which an arithmetic operator ($=, \neq, <, \leq, >, \geq$) has the highest precedence. The *AE* transformation creates a function, called an *AE* term, that calculates the difference between the operands of the arithmetic operator. If the original statement contains quantifiers, then the *AE* transformation calculates the *average* difference between the operands, over all permutations of variable bindings. The *AE* transformation could conceivably calculate other relationships between the operands, for example total difference, minimum difference or maximum difference. The *average* difference was chosen because it summarizes the relationships among a population of objects, rather than one relationship between a single pair of objects (as would the minimum or maximum difference).

If one operand of an expression is a constant, the resulting *AE* term measures the average distance to a threshold along some domain-dependent dimension. If neither operand is a constant, it measures the average distance between pairs of objects whose locations vary along the dimension. The problem-solver's task is either to drive the distance to zero (when the operator is $=$), to maintain a distance greater than zero (when the operator is $\neq$), or to enforce a particular ordering (when the operator is $>, \geq, <$, or $\leq$). In each case, an explicit representation of the average distance is useful.

For example, a subgoal for a circuit layout problem might require that all signal paths be less than one inch long ($\forall p \epsilon S, \text{length}(p) < 1$). The corresponding *AE* term indicates how close an average signal path is to the threshold.

The *UQ* Transformation

The *UQ* transformation applies to universally quantified expressions. Expressions with universal quantifiers require that every member of a set of objects satisfy the Boolean expression. The *UQ* transformation produces a function, called a *UQ* term, that calculates the percentage of permutations of variable bindings satisfying the Boolean expression.

UQ functions are useful because they indicate 'how much' of the subgoal is satisfied in a given state. For example, if a subgoal for a blocks-world problem requires that all blocks be on a table ($\forall b \epsilon B, \text{on}(b, \text{Table})$), then the corresponding *UQ* term indicates the percentage of blocks on the table.

Use of Problem-Solving Operators

The previous sections assume that all of the information about the goal state resides in its description, but that may not be the case in practice. It is usually desirable to move as many of the goal state requirements as possible into the preconditions of the problem-solving operators, so that fewer search states are generated. The advantage of doing so is that search speed may increase dramatically [Mostow & Bhatnagar, 1987].

The practice of moving goal state requirements into the preconditions of problem-solving operators requires that the previously described transformations also be applied to the preconditions of each operator. At worst, doing so allows the creation of some useless terms, many of which will be identified and deleted by the syntactic pruning rules (discussed below). The additional cost is offset by the guarantee that *all* of the information about the goal state will be found, whether it resides in the description of the goal state or in the preconditions of the operators.

Pruning

The *LE* transformation creates terms by extracting embedded expressions from a statement in first order predicate calculus. Sometimes taking an expression out of context causes it to become constant, or to duplicate other such expressions. Constant and redundant terms are undesirable because they needlessly increase the dimensionality of the space searched by the inductive algorithm. They also violate the assumption of linear independence among features, upon which some inductive algorithms depend [Young, 1984].

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

- **Duplicate pruning:** Deletes terms that are syntactically equivalent under canonical variable naming and quantifier ordering.
- **Quantified variable relation pruning:** Deletes expressions that contain only equality or inequality relations between quantified variables (e.g. $\forall v_1, v_2 \epsilon S, v_1 = v_2$).
- **Static relation pruning:** The adjacency relationship between squares of a chessboard is an example of a constant relation. Constant relations are currently identified manually, although an automatic identification algorithm is under development. Once constant relations are identified, terms that involve *only* constant relations are deleted automatically.

The remaining terms are potentially useful. They must be evaluated using other methods, for example based upon feedback from the learning algorithm.

Estimation of a Term's Cost

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

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

Results

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

The terms generated by CINDI have been tested in two domains. One domain, the blocks-world, was selected because its search space is small enough that the effectiveness of the terms could be determined precisely. The second domain, the game of OTHELLO, has a larger search space, so effectiveness in that domain was determined empirically.

In both experiments, qualitative information was used to train the evaluation functions. Qualitative information indicates that one state is preferred to another ($h(s_i) > h(s_j)$), but does not assign either state a value. The principle advantage of qualitative information is that it is more easily obtained than is precise information about a state's value.

The evaluation functions were implemented by single linear threshold units (LTUs) [Nilsson, 1965]. The LTU was chosen because it handles real-valued terms, and because of its well-known limitations [Minsky & Papert, 1972]. The intent was to study the effect of the *representation*, rather than the power of a particular learning algorithm. More complex learning algorithms use multiple hyperplanes to classify examples, while an LTU uses just one. Therefore the vocabularies that an LTU prefers might also be useful to more complex learning algorithms.

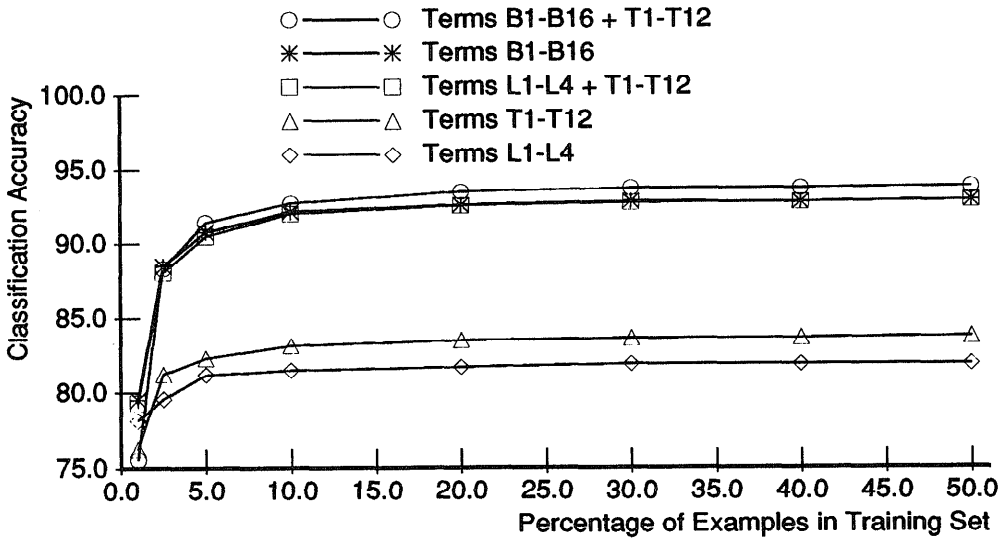

Figure 1: Learning curves for the 4 blocks problem.

Given two states s_i and s_j, each described by a vector of numeric features, and an LTU, described by a weight vector W, qualitative training information is encoded as follows:

$$h(\vec{s_i}) > h(\vec{s_j}) \quad (1)$$

$$\vec{W}^T \cdot \vec{s_i} > \vec{W}^T \cdot \vec{s_j} \quad (2)$$

$$\vec{W}^T \cdot (\vec{s_i} - \vec{s_j}) > 0 \quad (3)$$

$$h(\vec{s_i} - \vec{s_j}) > 0 \quad (4)$$

The Recursive Least Squares (RLS) learning rule [Young, 1984] was used to adjust the weight vector W, because it does not require that examples be linearly separable [Young, 1984]. However, the RLS algorithm does require that $h(\vec{s_i} - \vec{s_j})$ be assigned a value. In these experiments, $h(\vec{s_i} - \vec{s_j}) = 1$. We are considering a switch to a Thermal Perceptron [Frean, 1990], because it can use Equation 4 directly and it does not require linearly separable examples.

A Blocks-World problem

The blocks-world is a simple domain that is often studied in Artificial Intelligence. The problem studied here contained four blocks, labelled A through D, and a table. The problem-solver's goal was to stack A on B, B on C, C on D, and D on the table. The starting state was generated randomly.

CINDI generated ten *LE* terms from the specification of the goal state and the preconditions of the operators. The syntactic pruning rules eliminated two terms. No *AE* terms were generated, but four *UQ* terms were generated, for a total of twelve terms. The terms were arbitrarily labelled T1-T12.

For example, a precondition of the *stack-block* operator required that there be a block with an empty top. Two terms were created from that precondition. Term T6, an *LE* term, indicated whether some block had an empty top. Term T10, a *UQ* term, indicated the percentage of blocks with empty tops.

Terms T1-T12 were compared to two hand-coded representations, labelled B1-B16 and L1-L4. The hand-coded representations were intended to be the kind of representations that a problem-solver might use. Terms B1-B16 were Boolean terms that indicated the positions of the blocks. B1 indicated whether A was on B, B2 indicated whether A was on C, and so on. Terms L1-L4 were numeric. Each term indicated the position of one block. A term's value was 0 if its block was on top of A, 1 if its block was on B, and so on; its value was 5 if its block was on the table.

Figure 1 shows the percentage of examples on which an LTU has to be trained for it to reach a given level of accuracy in identifying which of two *randomly* selected search states is closer to a goal state. The values shown are accurate to within $\pm 1\%$, with 99% confidence. The figure shows the effects of both augmenting and replacing the problem-solver's vocabulary with the terms generated by CINDI. Performance improves when terms L1-L4 are replaced, but degrades when terms B1-B16 are replaced. However, performance improves when either the L1-L4 or B1-B16 representations are augmented with terms T1-T12. Representation L1-L4 shows more dramatic improvement.

Othello

OTHELLO is a two-player game that is easy to learn, but difficult to master. Players alternate placing discs on an 8x8 board, subject to certain restrictions, until neither player can place a disc. When the game is over,

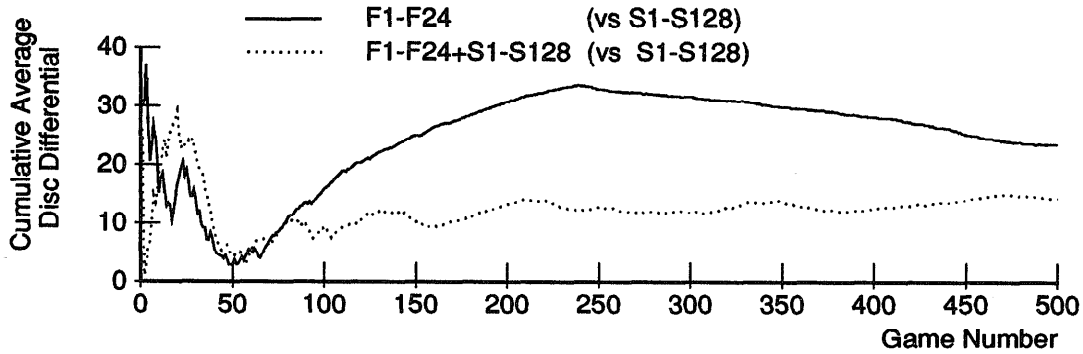

Figure 2: Cumulative average difference between scores of winners and losers in two OTHELLO tournaments.

the player with more discs has won.

CINDI generated 38 *LE* terms from the specification of the goal state. The syntactic pruning rules eliminated 15 terms. One *AE* term was created and 21 *UQ* terms were created, for a total of 45 terms. CINDI's cost estimates were used to eliminate any terms whose cost exceeded a manually-provided threshold. The remaining 24 terms were arbitrarily labelled F1-F24.

Several terms created by CINDI correspond to well-known OTHELLO concepts. One *LE* term indicated whether the player had *wiped out* (completely eliminated from the board) its opponent. The *AE* term measured the difference between the number of discs owned by each player (the *disc differential*); it can be used to play maximum and minimum disc strategies [Rosenbloom, 1982]. One *UQ* term measured the number of moves available to a player (*mobility*); it is highly correlated with winning positions [Rosenbloom, 1982].

Terms F1-F24 were compared with a set of terms, S1-S128, used in two independently developed, performance-oriented, OTHELLO programs. Terms S1-S64 are binary terms, one per square, that indicate whether the player owns the square. Terms S65-S128 are the corresponding terms for the opponent. A third representation, consisting of terms F1-F24 *and* terms S1-S128, was also considered.

Performance was measured by allowing LTUs with differing representations to compete for 500 games. After each move, an 'oracle' program[1] indicated the best move. The evaluation functions were then trained so that the oracle's move was preferred to all other moves.

Figure 2 shows the average disc-differential between pairs of players as the tournaments progressed. The disc-differential, a standard measure for OTHELLO, is the difference between the winner's score and the loser's score. Both tournaments show that the terms generated by CINDI are effective for learning OTHELLO evaluation functions. The tournament between the combined representation (F1-F24+S1-S128) and terms S1-S128 demonstrates that *augmenting* the problem-solver's vocabulary improves performance. The tournament between the new terms (F1-F24) and terms S1-S128 demonstrates that *replacing* the problem-solver's vocabulary results in a greater improvement.

The RLS algorithm may not converge if features are not linearly independent (i.e. if some features are linear functions of other features) [Young, 1984]. Terms S1-S128 are linearly independent, but terms F1-F24 are not. The decay in the evaluation function based upon terms F1-F24 is most likely caused by linear dependencies among the features. These dependencies probably also harm the performance of the evaluation function that uses the combined representation (F1-F24+S1-S128). However, in spite of this handicap, evaluation functions that use terms F1-F24 consistently outperform evaluation functions that use only terms S1-S128.

Related Work

Abstractions of an expression are generated by eliminating some combination of its predicates [Mostow & Prieditis, 1989; Pearl, 1984; Gaschnig, 1979]. The number of such expressions is combinatorially explosive, so human guidance is often used to restrict attention to a small subset of the possible abstractions. Each abstraction represents a simpler problem whose solution guides search in the original problem. However, unless care is taken, the additional cost of solving these simpler problems makes the overall approach more expensive than blind search [Valtorta, 1983; Mostow & Prieditis, 1989].

The *LE* transformation generates abstractions, but it does not generate the full set. It only generates between $c+1$ and $2c$ abstractions, where c is the number of connectives in the problem specification. Therefore it avoids both combinatorial explosion and the need for human guidance. Once created, *LE* terms are evaluated in the context of a single search state, which avoids the expense of searching for a solution that is associated with abstractions.

[1] A program that uses handcrafted features, a handcrafted evaluation function, and 3-ply search. It switches to exhaustive search in the last 12 moves of the game.

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

Conclusion

This paper describes a method for transforming easily available domain information into terms with which to train an inductive learning algorithm. The method, called *knowledge-based feature generation* [Callan, 1989], includes pruning rules, and lower bound cost estimates for each feature. The number of features produced is a linear function of the number of connectives in the domain knowledge, so the method will scale to complex problems. The method has been tested on two problems of differing size. In each case, the features it generated were more effective for inductive learning than were the problem-solver's features.

The features created by knowledge-based feature generation are intended to augment or replace the problem-solver's vocabulary as an *initial* vocabulary for inductive learning. Other methods of constructive induction can also be used, in conjunction with the inductive learning algorithm, to provide additional improvement in the vocabulary.

A number of topics remain for further research. One of these is the development of an *EQ* transformation, to apply to existential quantifiers. Another is an investigation into control rules for supplying features to an inductive learning algorithm. The subgoal hierarchy and the cost estimates enable one to create a variety of incremental algorithms for supplying terms to the inductive algorithm. None of these have been investigated. A final problem is to characterize the situations in which knowledge-based feature generation is likely to improve a problem-solver's vocabulary.

Acknowledgements

We thank Jeff Clouse for the oracle OTHELLO program, and Tom Fawcett for his OTHELLO domain theory. We thank John Buonaccorsi and Chengda Yang for assistance in the design of experiments. We thank Chris Matheus, Andy Barto, Tom Fawcett, Sharad Saxena, Carla Brodley, Margie Connell, Jeff Clouse, David Haines, David Lewis, and Rick Yee for their comments.

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.

Frean, M. (1990). *Small nets and short paths: Optimising neural computation.* Doctoral dissertation, Center for Cognitive Science, University of Edinburgh.

Gaschnig, J. (1979). A problem similarity approach to devising heuristics: First results. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp. 301-307). Tokyo, Japan.

Lenat, D. B., & Brown, J. S. (1984). Why AM and EURISKO appear to work. *Artificial Intelligence, 23,* 269-294.

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 R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Minsky, M., & Papert, S. (1972). *Perceptrons: An introduction to computational geometry (expanded edition).* Cambridge, MA: MIT Press.

Mostow, J., & Bhatnagar, N. (1987). Failsafe – A floor planner that uses EBG to learn from its failures. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 249-255). Milan, Italy: Morgan Kaufmann.

Mostow, J., & Prieditis, A. E. (1989). Discovering admissible heuristics by abstracting and optimizing: A transformational approach. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 701-707). Detroit, Michigan: Morgan Kaufmann.

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

Pearl, J. (1984). *Heuristics: Intelligent search strategies for computer problem solving.* Reading, Ma: Addison-Wesley.

Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess end games. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

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

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

Valtorta, M. (1983). A result on the computational complexity of heuristic estimates for the A* algorithm. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (pp. 777-779). Karlsruhe, West Germany: William Kaufmann.

Young, P. (1984). *Recursive estimation and time-series analysis.* New York: Springer-Verlag.