A Hybrid Theory of Feature Generation

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

COINS Technical Report 91-8
June 1991

Abstract

Existing approaches to constructive induction have been largely empirical, starting with structural features and combining them successively into higher level features. This thesis proposal presents a hybrid analytical/empirical theory of feature generation for inductive concept learning. From this theory a transformational method of feature generation is developed. The goal of the method is to generate useful features to improve problem solving performance, given only a theory of the domain and the ability to perform in the domain. An implementation of this method is able to generate useful features for the OTHELLO board game, given just the rules of the game and the ability to interact with an opponent. Additional research and experiments are described that are expected to provide support for the method, as well as evidence for its generality.

Contents

1 Introduction

One of the central concerns of machine learning is 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 generalize inductively 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 threshold units [Nilsson, 1965] represent the concept predicate as an inequality involving a threshold and a weighted sum of example features. Many concept-learning systems [Michalski & Chilausky, 1980; Dietterich & Michalski, 1983] represent the concept as different forms of boolean expressions.

1.1 Induction and Representation

Regardless of the formalism, inductive concept learning methods are sensitive to the set of features used to describe the examples. This set of features strongly influences not only the form, size and accuracy of the final concept learned, but the speed of convergence of the learning method as well. In some cases, as with linear threshold units, the learning method may never converge if the instance description vocabulary is inappropriate. As a result, a human is usually responsible for determining whether the example representation must be changed. If the learned concept is unacceptably inaccurate, or its form unacceptably large, the human guesses what new features would help the concept learning algorithm, adds the new features, then runs the algorithm again. The human is therefore part of the learning cycle, so the process is not automatic.

The problem of devising new terms for an induction task has been given many names. It has been called *constructive induction* [Michalski, 1983] and the *new term problem* [Dietterich, London, Clarkson & Dromey, 1982], to emphasize the fact that the induced concept is formed from newly-constructed terms, rather than from those that were initially given to it. The set of representable concepts has been called a *bias* of the learning method, so the process of adding new terms has been called *shift of bias* [Utgoff & Mitchell, 1982]. In multi-layer connectionist networks, the internal (hidden-layer) nodes can adapt themselves to recognize recurring input patterns; this phenomenon has been called *learning internal representations* [Rumelhart & McClelland, 1986].

Regardless of the names and details of the processes, there are certain aspects common to all of them. One way of viewing the role of new features in induction is in the layers illustrated in Figure 1.

- Examples are expressed using a vocabulary called the **input representation**, or instance-level features. The input representation consists of functions and predicates on the example, as well as relations among parts of the example. For example, in the domain of chess an example chess state might be represented by listing the pieces occupying each board square, expressed as occupies(piece,square), along with the neighbor

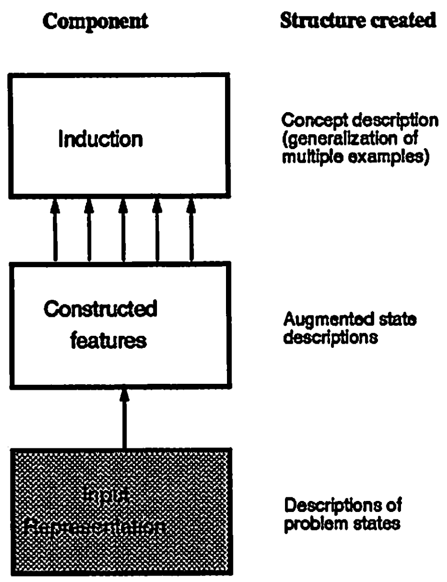
Component
Structure created
Induction
Concept description (generalization of multiple examples)
Constructed features
Augmented state descriptions
Input Representation
Descriptions of problem states

Figure 1: The role of feature creation in induction

relations among the board squares, as neighbor(square1,square2,direction). The input representation is immutable.

- The **constructed feature** layer augments the input representation by adding descriptors to the examples. For example, in chess a constructed feature for a fork might add a description like fork(fsquare,tsquare1,tsquare2), indicating that the piece at square fsquare is involved in a fork threatened by tsquare1 and tsquare2. The descriptions added by features may also be numeric; for example, mobility(white,23). Constructive induction operates by adding new features at this intermediate level.

- The **induction** layer performs induction over the augmented example descriptions and forms a final concept. That is, it performs a search for an acceptable concept description based on the set of example descriptions provided by the layers below it. The induction process can be as simple as the absolute correction rule for a linear threshold unit [Nilsson, 1965] or as complicated as an algorithm like AQ11 [Dietterich & Michalski, 1981] that performs induction over structured, symbolically represented examples.

1.2 Existing Approaches to Feature Generation

Constructive induction may then be seen as a process that changes the features computed in the second layer. Generally, feature learning involves a separate process that is able to examine the result of a concept learning algorithm and suggest new useful features that will improve it. There are many existing systems that create new features, many of which are reviewed in Section 2.

Two general approaches have been taken. The predominant approach has been data-driven, in which new features are created by combining existing features in various ways. Systems of this type, such as CITRE [Matheus & Rendell, 1989] and STAGGER [Schlimmer & Granger, 1986], start with the input representation features and build up more complex ones, for example by creating the conjunction of two features. Typically, feedback from the learned concept is used to suggest plausible combinations. Usually, new features compete with old ones and the best are retained. This process continues until the learned concept is acceptable, which is usually defined in terms of size or accuracy. Data-driven techniques are limited in the amount of improvement that they can accomplish because they start with the example features and combine features one step at a time. If useful features are complex combination of the example features, a data-driven system will need to generate and test prohibitively many features before a useful one is derived. As a result, data-driven systems have only been applied to toy domains in which useful features are simple combinations of example features.

The second approach is analytical, and makes use of a domain theory to construct features. Systems of this type, such as STABB [Utgoff, 1986a], use information about the domain to deduce appropriate new features, rather than combine example features in new ways. Because such systems exploit information about the domain, they are able to create complex features in one step, and can guarantee that the features will be useful to the concept learner. However, such systems can only create features that follow deductively from the domain theory. Many real-world domains require useful features that are not deducible from the domain theory, and these analytical systems are incapable of deriving them. In addition, these analytical systems place strong restrictions on the domain theories to which they can apply.

Because of their deficiencies, neither of these approaches is acceptable as a general model of feature creation. Performance of feature discovery systems is far behind human capabilities. People are able to learn useful features of a domain given little more than a theory of how the domain works. For example, through experience people can become excellent chess players given just the rules of chess and some background knowledge about two-person games. Programs should be able to develop features as sophisticated as those that people use, given the same information that people are given.

Specifically, given information about a domain and the ability to perform in the domain, a system should be able, unaided, to generate useful features. When utilized by an inductive learning method, these features should enable the system to improve its performance steadily, and ultimately to exhibit human-level performance in the domain.

1.3 A Hybrid Approach

This proposal offers a different approach to feature generation, the central thesis of which is that

> *Useful features can be derived from abstractions and combinations of abstractions of the domain theory.*

Abstractions are created by relaxing conditions specified in a domain theory, so the approach is a hybrid of the data-driven (bottom-up) and theory-driven (top-down) approaches. The use of a domain theory allows the system to start with complex initial features that are sensitive to the goals and operators of the domain, rather than simply starting with the instance-level features. The domain theory also allows the system to perform goal regression, a powerful goal-based feature transformation. However, the system must also be able to generalize and combine features as well. Generalizing a feature increases its applicability, and combining features can increase discriminability by excluding negative instances.

The purpose of this research is to investigate a theory of constructive induction based on this hybrid approach and to explore its strengths and weaknesses. The theory will be supported empirically with an implemented system, and the limits of the theory will be determined by characterizing the features that it can and cannot discover. The primary contribution of this research will be to demonstrate that aspects of both data-driven and theory-driven learning are necessary for the generation of useful features in non-toy domains. Because the method creates useful abstractions from an intractable domain theory, it will also address the intractable domain theory problem and the problem of automatic generation of abstractions.

The rest of the proposal is organized as follows. Section 2 discusses three classes of related work in constructive induction, and concludes by analyzing their strengths and weaknesses. Section 3 presents a theory of constructive induction, along with the model on which the theory is based, and an example derivation of several features. Section 4 describes the proposed work of the thesis, including a discussion of how the thesis will be demonstrated, and how the work will affect research in machine learning.

2 A Brief Overview of Constructive Induction

Existing work in constructive induction (CI) may be divided into three categories according to the *purpose* for which new features are created. This division stresses the relationship between the concept formation component and the feature generation component.

1. **Supporting complementary representations:** The features use a representation different from the concept's; therefore, the space searched by the CI component is fundamentally different from that searched by the induction component.

2. **Changing search bias:** In this case, the features use the *same* representation as the concept description; therefore, the CI component is not expanding the representation space of the induction component, but may be changing the order in which concepts are considered. This is the idea behind CITRE [Matheus & Rendell, 1989] and FRINGE [Pagallo, 1989], both of which create new features that are conjunctions extracted from the concept representation itself.

3. **Increasing conciseness:** Some systems create new terms not to increase accuracy directly, but to increase conciseness of the domain theory. This conciseness may either be an end in itself (as in DUCE [Muggleton & Buntine, 1988]), or as a way to increase the efficiency of the domain theory (as in RINCON [Wogulis & Langley, 1989]). Many clustering systems (e.g. COBWEB [Fisher, 1987]) have conciseness as a primary goal, with the assumption that conciseness will aid accuracy.

The next three sections review related work in constructive induction organized by these categories.

2.1 Systems that Support Complementary Representations

2.1.1 STABB (Utgoff)

STABB [Utgoff, 1986a; Utgoff, 1986b] is the constructive induction component of LEX, a system that acquires problem-solving heuristics in the domain of symbolic integration. LEX contains a problem solver, a critic to extract training instances from a problem solution trace, and a generalizer that takes instances from the critic and produces generalized heuristics. LEX has a set of operators that perform algebraic manipulations and integration operations. Each operator has one or more associated concepts, each a *heuristic*, that represent the problem forms to which its operator may usefully be applied. The learning goal of LEX is to refine these heuristics as much as possible, in order to reduce search in applying operators to solve integration problems.

LEX represents incompletely learned heuristics using Mitchell's version space method [Mitchell, 1977], by which the most general and most specific boundaries (versions) of a concept are explicitly stored. These boundaries may be adjusted by the generalizer after every training instance involving the heuristic. Version spaces depend on a generalization hierarchy appropriate for the learning task; LEX's hierarchy contains mathematical terms such as *transcendental function*, *integer*, $+$, and *monomial*.

STABB generates new terms for LEX's hierarchy, and is invoked when the version space of an operator collapses (when there are no terms left in the region between its general and specific boundaries). STABB is given the sequence of operators that solved the integration problem, and the operator in the sequence whose version space is now empty as a result of generalization. STABB has two methods for creating new terms: **constraint back-propagation** and **least disjunction.**

Least disjunction is a goal-free method that creates a new term as a disjunction of existing terms. The new term is guaranteed to be sufficient to match all of the positive examples and none of the negative examples. Least disjunction is the default method of STABB, and will always derive a new term. However, because it is insensitive to the goal, it is much weaker than constraint back-propagation. That is, it will create the minimal disjunction necessary to encompass two terms, whereas constraint back-propagation will derive a term that characterizes every state for which the operator sequence may be applied.

Constraint back-propagation (CBP) generates new terms by propagating the set of all solved states back through an operator sequence that yielded the original solved trace. Given an operator sequence $OP_1, OP_2, \cdots, OP_n$, and a solution state S, CBP computes the operator preimages $OP_1^{-1}, OP_2^{-1}, \cdots, OP_n^{-1}$ to S in reverse order. The expression S, backed up through an operator preimage OP_k^{-1}, yields a sufficient condition (call it S_k) for which OP_k will yield a solution. Therefore, if OP_k is the operator whose version space is now empty, CBP generates S_k, adds it as a new term to LEX's hierarchy, and uses it to regenerate the version space for OP_k.

In summary:

- STABB is a knowledge-based constructive induction component for LEX. STABB's CBP method is able to create new terms that are exact sufficient conditions of usefulness for LEX's operators.

- STABB's CBP is an *analytical* method closely related to explanation-based generalization [Mitchell, Keller & Kedar-Cabelli, 1986]. CBP's use of operator preimages accounts for both its strength (that of creating terms that are exactly sufficient) and also its weakness (that of needing computable operator preimages).

- LEX's generalization hierarchy (concept description language) represents an initial strong bias used for induction. STABB is able to alter that bias on concept failure by composing conditions from LEX's operator language. Two assumptions underly this relationship:

  1. STABB's CBP method is assumed to be too expensive to be used in place of induction, and should only be used when necessary (defined by concept failure).
  2. The terms added by CBP, while created expressly for a single operator, are assumed to be useful for the domain in general. By placing these terms in the

hierarchy, other operators' heuristics can take advantage of them when their version spaces are adjusted.

2.1.2 STAGGER (Schlimmer)

STAGGER [Schlimmer & Granger, 1986] integrates weight learning with boolean function learning in a principled manner. Subsequent work [Schlimmer, 1987] added numeric attribute value partitioning (i.e., learning ranges).

In STAGGER, a concept description is a set of numerically weighted features, called *elements*. Each element is a boolean function of attribute values, e.g. (SIZE=MEDIUM) AND (COLOR=RED). Each of these elements has two weights associated with it: a measure of its logical necessity (LN) and logical sufficiency (LS). These weights are computed as:

$$LS = \frac{p(matched|example)}{p(matched|\neg example)} \qquad LN = \frac{p(\neg matched|example)}{p(\neg matched|\neg example)}$$

Whenever a new instance is presented to STAGGER, each element is matched against the instance and its weights are used to make a prediction about whether the instance is a positive or negative example of the concept being learned. After the prediction is made, the LS and LN of each element are updated by recomputing the conditional probabilities above.

Without the ability to learn new elements, STAGGER is effectively a single-layer connectionist network. As such, it has the well-known limitation inherent in a linear threshold unit: it can only represent concepts that are linearly separable (although these include purely conjunctive and purely disjunctive concepts).

Feature learning in STAGGER is triggered by concept prediction failure. This is different from (and less exact than) concept failure in LEX, which is defined by there being no concept description consistent with positive and negative examples. In STAGGER, prediction failure may also be caused by inappropriate weights.

New features are formed by applying the boolean operators AND, OR and NOT to existing features according to a set of heuristics. For example, if STAGGER predicted a negative example to be positive (an error of commission), it might conjoin two features having large LN values to form a new, more specific feature. This new feature would then compete with the others, and if its predictive accuracy fell below that of its component features, the new feature would be removed.

Thus, the concept learning of STAGGER allows it to learn linear combinations of features, and the feature learning in turn employs feedback from the weights to direct it. Since STAGGER only generates new features upon prediction failure, it may be seen as having a strong bias toward linearly separable concepts. Its feature generation component serves to shift this bias, using boolean operators, when it is inadequate.

2.1.3 MIRO (Drastal, Czako and Raatz)

MIRO [Drastal, Czako & Raatz, 1989] is related to both constructive induction and explanation-based learning, in that it generates new terms for induction from concepts (descriptors) implied by deductive rules from a domain theory. The authors refer to this as induction over an abstraction space, since the new terms are presumably more abstract than the instance features. The authors also characterize the process as using a strong form of deductive (and thus justifiable) bias.

MIRO takes as input a set of positive and negative instances, each expressed as a set of features. The features can be positive or negative literals. MIRO has a domain theory, composed of Horn clauses, whose inferences must be representable as a finite acyclic AND/OR graph. Negations (negative features) can occur in the right-hand side of a clause, but the left-hand side must assert a positive literal.

At the center of MIRO's operation is its construction of an *abstract concept description language*, $L_{\mathcal{A}}$, based on the domain theory provided. The definition of $L_{\mathcal{A}}$ is in turn based on the notion of a *maximally proven descriptor*. Informally, if a descriptor x (the conclusion on the right-hand side of a rule) can be deduced from some example e, but no other descriptors may be deduced using x, then x is said to be a maximally proven descriptor. MIRO assumes that the domain theory's rules deduce abstract descriptors from less abstract descriptors, thus x can be thought of as a maximally abstract descriptor of e. The set of all such maximally proven descriptors, computed from each positive and negative example, constitutes the abstract concept description language $L_{\mathcal{A}}$.

After this language is defined, MIRO uses a greedy algorithm to induce a concept description from it. Similar to the AQ11 algorithm [Dietterich & Michalski, 1981], MIRO repeatedly selects a seed from the set of positive examples and creates a partial concept description that covers the seed and none of the negative examples. The partial concept description is created using a one-sided variant of the candidate-elimination algorithm [Mitchell, 1978]. The one-sided candidate-elimination method computes a G set, in the abstract language $L_{\mathcal{A}}$, that is consistent with the seed and the set of all negative examples. MIRO heuristically selects an element of this G set as a "cover" of the seed, which is a partial concept description of the target concept. MIRO then chooses another seed, and continues until all positive examples are covered. Finally the partial concept descriptions are disjoined into a complete concept description, which MIRO returns.

It may happen that MIRO cannot produce a consistent concept description because the abstract language $L_{\mathcal{A}}$ is insufficient to discriminate the positive and negative examples. In this case, MIRO is able to decrease the inductive bias by incrementally adding non-maximally proved descriptors into $L_{\mathcal{A}}$. After every change to $L_{\mathcal{A}}$, the examples are reprocessed. MIRO continues to extend $L_{\mathcal{A}}$ as long as $L_{\mathcal{A}}$ is insufficient to discriminate the examples. Eventually, MIRO will add the instance language descriptors into $L_{\mathcal{A}}$; if these are still insufficient, MIRO will declare failure.

MIRO is one of the few models of constructive induction that incorporate both deduction and induction. There are a number of weaknesses of MIRO as a general model of constructive induction:

- MIRO is limited in the kind of domain theories it can use. The domain theory must be representable as a finite acyclic AND/OR graph, so recursive rules are not allowable. Furthermore, the experiments in [Drastal, Czako & Raatz, 1989] use an even more restrictive propositional logic domain theory; although MIRO could theoretically employ any domain theory consisting of first-order (but non-acyclic) Horn clauses, the authors admit that substantial problems arise when such domain theories are allowed.

- MIRO makes the assumption that rules in the domain theory make inferences from less abstract descriptors to more abstract descriptors. Careful engineering of the domain theory may be required to assure that this condition is met. MIRO is currently incapable of being used in a domain in which rules specify transitions from one state to another, rather than from a description to a more abstract description.

2.1.4 LIVE (Shen)

LIVE [Shen & Simon, 1989] learns predictive rules of an environment, given some basic information about the environment and the ability to perform experiments in the environment. Specifically, LIVE is given a set of *actions* that it can execute, a set of *features* that it can perceive in the environment, a set of *constructors* for creating new features, and a *goal* to achieve in the environment. By performing experiments in the environment, LIVE is able to create rules (STRIPS-style operators with pre- and post-conditions) describing the conditions under which an action will have a certain effect.

LIVE's learning can be characterized as difference-based specialization of rules. It starts with overly general rules that describe the effects of its actions. When these rules are violated, new, more specialized rules are created to account for the violating event. LIVE specializes the rules by discriminating between a state in which its rule predicted correctly and the state that violated the rule.

LIVE learns new features when its existing features are unable to discriminate between these "correct" and "surprising" states. It creates features using its given set of constructors. For example, one of LIVE's tasks was that of deriving Mendel's Law (that the dominant gene determines an attribute) given the ability to simulate the artificial fertilization of garden peas, and a set of constructors. In this task, LIVE's goal is to predict the color of peas that result from cross-breeding peas of various colors. In the process of deriving these laws, LIVE used its constructor functions to generate features to discriminate states upon which the action had different effects. In deriving the notion of dominant gene, LIVE first generated the feature of *different* parent's genes, then of one gene's representation being *greater than* another. After another false prediction, LIVE was forced to examine the grandparent peas' colors in order to find some distinguishing characteristic that would account for the differences in the grandchildren peas' colors. The final rule that LIVE produces expresses the color of the offspring of a union as the bitwise-OR of the inherited genes. This works because there were two genes, represented as 1 and 0, with the dominant gene represented as 1.

If the rule preconditions are viewed as a concept, then LIVE is performing concept learning. LIVE is unusual as a concept learner in that it generates concepts to differentiate only two states at a time. Its features — the new terms required to discriminate rule preconditions — are generated by its constructors when the existing features are inadequate for doing this differentiation.

The main shortcoming of LIVE's feature generation is that the set of constructor functions given to it represents a bias that may not be correct for the features that are needed. In the example of [Shen & Simon, 1989], the "bitwise-OR" function is not obviously useful for predicting pea color, and indeed only works if the dominant color is represented as 1, and the recessive color represented as 0. Other problems exist that are common to data-driven feature generation methods. The constructors may produce a very large feature space, depending on their number and the order in which they are searched. There is also the problem of finding the *right* feature: LIVE stops as soon as it finds some feature that can account for a difference between two states, but there may be many such features, and LIVE may not be able to recover if the wrong choice is made. Other feature generation systems are less sensitive to this problem because they only demand that the features be predictive of the concept, not that they uniquely determine it.

2.1.5 BACKPROP

The simplest connectionist architecture is the one-layer network, called the linear threshold unit (LTU) [Nilsson, 1965]. An LTU maintains a set of weights, and given a set of numeric feature values, produces a thresholded linear sum of the weights as output. Geometrically, the weights of the LTU represent a hyperplane. In general, n weights constitute an $(n-1)$-dimensional hyperplane that splits the n-dimensional feature space. Given an instance feature vector, if the output of the LTU is less then zero, the instance represented lies on one side of the hyperplane; if the output is greater than zero, it lies on the other side.

The LTU has the well-known restriction that it can only discriminate instances that are linearly separable. This restriction can be eliminated by using multi-layer connectionist networks, in which there is a *hidden layer* that can compute intermediate results that may then be used by the top-level (output) node. The generalized delta rule [Rumelhart & McClelland, 1986, Chapter 8], commonly called the BACKPROP algorithm, can learn weights for a network of LTUs by back-propagating error signals from the output node down into the hidden layer. Through training, the hidden-layer LTUs can learn to compute useful intermediate features of the input nodes for use in the output LTU, and this is a form of constructive induction. Every distinct set of weight vectors for the hidden layer is a separate representational bias.

Rumelhart *et al* (1986) have used the generalized delta rule in a number of multilayer connectionist networks applied to known non-linearly separable problems. For example, BACKPROP was applied to the parity problem (determining whether a set of n bits has even or odd parity) using a network with n input and n intermediate nodes. BACKPROP created features that counted the number of inputs that were on. That is, intermediate node k became active when k or more of the input lines were on. The output LTU learned positive links to each of the odd-numbered intermediate nodes, and negative links to each of the even-numbered intermediate nodes. This resulted in the output LTU becoming active only when an odd number of input nodes were on.

It is impressive that such useful features can be created virtually *ex nihilo* by a simple, uniform algorithm; however, there are some drawbacks to BACKPROP:

- The algorithm is purely empirical, and incorporating background knowledge into it is an open problem. The method of [Towell, Shavlik & Noordewier, 1990] can build a connectionist network from a domain theory, but the method is limited to domain theories expressed in propositional logic.

- Empirically, BACKPROP requires significantly more presentations than symbolic algorithms. Two recent empirical studies, [Mooney, Shavlik, Towell & Gove, 1989] and [Fisher & McKusick, 1989], observed that BACKPROP takes substantially longer than ID3 on the same task (this conclusion was consistent over the six tasks total studied in the two research projects). The former study concluded that BACKPROP needed one to two orders of magnitude more presentations than ID3 to achieve the same level of accuracy, although BACKPROP was slightly more noise-tolerant.

  It should be mentioned that measuring the training time by the number of presentations may be misleading. Connectionist networks can typically process examples faster than symbolic algorithms (e.g., ID3). However, they are usually not an order of magnitude faster.

- The design of connectionist networks has not been completely automated because several aspects of network architecture are not automatically determinable. The initial settings of a network's weights have a strong influence on both the learning speed and the generalizing ability of the resulting network [Towell, Shavlik & Noordewier, 1990]. Also, the number of intermediate nodes needed for a network to solve a given problem varies a great deal, and is generally not determinable [Fisher & McKusick, 1989]. If too few are used, the network will not attain 100% accuracy on the problem; if too many are used, the network will "memorize" the inputs and not form useful features, and thus not generalize well to unseen inputs. It is possible to use a dynamic network architecture whereby more intermediate nodes are added when necessary [Ash, 1989], but this trial-and-error approach increases the number of presentations required.

2.2 Systems that Change Search Bias

2.2.1 CITRE (Matheus)

CITRE [Matheus & Rendell, 1989] creates new terms for (and from) decision trees, and has been applied to the domain of tic-tac-toe. The initial attributes are structural: they are the board position $pos_{11}, pos_{12}, pos_{23}$, etc., which can take on values x, o and blank. Thus, a feature is an expression like equal(pos11,x), which can be tested in a decision tree.

CITRE creates new terms when there is more than one positive-valued leaf in a decision tree (and thus, when the concept is disjunctive). It has only one feature-creation operator, described as *and*(_, _), which conjoins two features. CITRE considers every positively-labelled branch of the decision tree and applies its operator to every pair of features on the path. It also uses two domain-specific filtering heuristics (one filters out features of different piece types, the other filters out features involving non-adjacent squares) in the selection process. It then generalizes its features by changing common constants to variables. Finally, it evaluates the resulting features using an information-theoretic "utility" metric, which measures the discriminability of each feature individually. CITRE keeps only 27 features at any one time: the initial 9 primitive features, which are never discarded, and the 18 highest-ranked constructed features (no justification has been given for these numbers).

In later work [Matheus, 1990], Matheus added heuristics that are able to generalize newly-created features using domain-dependent methods. For the domain of tic-tac-toe, Matheus used heuristics that performed spacial translations of patterns. Using both the filtering heuristics and the generalization heuristics, CITRE is able to increase the accuracy of the learned concept about 15% over what ID3 can achieve with the instance features alone [Matheus, 1990, Figure 6].

In summary,

- CITRE is *cyclical* and *concept-based*, in that the examples are processed using an existing set of features, then new features are created from the induced concept. After 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.

- CITRE does not change the representation of the produced concepts, because CITRE creates only conjunctions of existing features, and a decision tree can implicitly represent conjunctions anyway. The performance increase comes about because CITRE is changing the search bias of the decision tree algorithm (ID3). Because ID3 considers only one attribute at a time in choosing an attribute on which to split, CITRE may be seen as constructing promising attribute conjunctions. If two attributes A and B correlate poorly with the concept, but their conjunction $A \wedge B$ correlates highly, then by creating this conjunctive feature CITRE will probably increase the accuracy of the resulting decision tree.

- CITRE is able to use some domain information, but it must be programmed in as a filtering or generalization heuristic. CITRE's feature composing operator, *and*(_, _), exploits no domain information in selecting features for arguments.

2.2.2 FRINGE (Pagallo)

FRINGE [Pagallo, 1989; Pagallo & Haussler, 1990] was designed to address the *replication problem*, in which some portion of a decision tree occurs more than once in the tree. FRINGE is very similar to CITRE, in that it acquires features by examining decision trees, but the feature forming heuristic is slightly different. FRINGE forms conjunctions of two features that occur at the fringes (the leaf nodes) of a decision tree. That is, given a positive leaf node, FRINGE creates a new feature by conjoining the features that are immediately above the leaf on the path.

FRINGE was applied to five different concept domains, including small random DNF functions, a multiplexor, and 4- and 5-bit parity. By creating new features from conjunctions, FRINGE was able consistently to outperform a decision tree algorithm that had access to only the instance-level features. Both classification accuracy and average tree size were better with FRINGE.

FRINGE has also been applied to more extensive machine learning domains, such as the mushroom classification domain [Schlimmer, 1986] and the hyperthyroid data [Quinlan, 1987]. The decision trees generated by FRINGE were more concise and at least as accurate as those generated by ID3.

2.3 Systems that Increase Conciseness

2.3.1 DUCE (Muggleton and Buntine)

The goal of DUCE [Muggleton & Buntine, 1988] is to produce a small, concise domain theory. DUCE begins with a domain theory in propositional logic and repeatedly applies rule-collapsing operators to it with the goal of reducing the total number of symbols in the theory. The operators all work by finding common subexpressions of the theory, and factoring them out into new rules. For example, one of the operators is **absorption**. Given the two rules:

$$\begin{aligned} X &\leftarrow A \wedge B \wedge C \wedge D \wedge E \\ Y &\leftarrow A \wedge B \wedge C \end{aligned}$$

the **absorption** operator will generate the rules:

$$\begin{aligned} X &\leftarrow Y \wedge D \wedge E \\ Y &\leftarrow A \wedge B \wedge C \end{aligned}$$

The other operators are similar. Note that, in isolation, absorption constitutes simple truth-preserving factoring, but in the context of a larger set of rules this factoring may introduce generalization, because other rules may assert Y.

DUCE depends upon an oracle (the user) to determine whether a given generalization is valid, and to name the intermediate concepts that it creates. Thus DUCE is by nature an interactive system, although it does remove some of the burden from the user by searching intelligently for the transformations it proposes. It does a best-first search based on the locally maximum reduction transformation; it proposes this transformation, and if the user accedes, performs it. DUCE continues this cycle until no further reduction is possible.

Note that the goal of DUCE is only to produce the smallest, most concise domain theory; this is an end in itself. There is no explicit link of this goal to performance accuracy (as in CITRE or FRINGE), or performance efficiency (as in RINCON below).

2.3.2 RINCON (Wogulis and Langley)

RINCON [Wogulis & Langley, 1989] is very similar to DUCE in that its explicit goal is to increase the conciseness of the domain theory, and that it performs a hill-climbing search for the transformations that allow it to do so. RINCON, however, is incremental, and its implicit goal is to increase the classification *efficiency*, rather than accuracy.

In RINCON, a domain theory is represented as a directed acyclic graph (tangled hierachy) of concepts, organized from general to specific. Each of the concepts may have one or more definitions, expressed as Horn clases. An instance is an instantiated clause, e.g.:

$$\begin{aligned} \mathit{uncle}(\mathit{walter}, \mathit{jim}) \leftarrow\ & \mathit{male}(\mathit{walter}) \wedge \mathit{sibling}(\mathit{walter}, \mathit{carol}) \wedge \\ & \mathit{sibling}(\mathit{walter}, \mathit{carol}) \wedge \mathit{mother}(\mathit{carol}, \mathit{jim}). \end{aligned}$$

For every instance given to RINCON, if the instance can be classified (i.e., proved) by the domain theory, no learning is done. Otherwise, RINCON begins a two-step process to learn and assimilate intermediate concepts:

1. RINCON finds the most specific concepts in the theory that match the instance, and the most general concepts in the theory that *do not* match the instance. The latter are used to rewrite the instance, so that the instance is effectively re-expressed with the highest-level matching concepts[1]. The resulting instance is added to the domain theory.

2. The instance is generalized with the lowest-level concepts that did *not* match. RINCON chooses as its generalization that which can allow it to re-express the greatest number of concepts in its domain theory. This generalization step is similar to DUCE's **absorption** operator, and its metric for choosing the generalization is similar to DUCE's metric as well.

Since RINCON is an incremental system, it performs these re-expressions and generalizations after every instance seen. Thus RINCON is susceptible to local minima effects, depending on example order. Furthermore, its purpose is not just to produce a more concise domain theory, but to improve the efficiency of matching instances. However, the authors point out that in some cases, a "flat" domain theory (one without intermediate concepts) is more efficient than one with intermediate concepts.

[1]This is similar to MIRO's use of maximally-proven descriptors (see Section 2.1.3).

2.4 Limitations of Existing Work

The feature creation systems surveyed above were categorized by the purpose for which the features are created; that is, the relationship between the concept formation and feature generation components. In discussing their limitations, it is also useful to place them into two general classes according to the amount of background (domain-dependent) information the feature generation component uses.

One class of feature generation methods may be termed *data-driven*, in that the methods begin with instance-level features, and combine them progressively to form more complex features. They start with the instance-level representation and build upwards, based at every step on the performance of the existing features. This is the predominant approach in constructive induction, and most of the systems surveyed above (STAGGER, BACKPROP, CITRE, FRINGE, DUCE and RINCON) are data-driven.

The main limitation of this data-driven approach is the size of the space that must be searched. It is assumed that the complexity of the useful features of a domain increases with the complexity of the domain. Therefore, the space of possible features, some portion of which a constructive induction method must search, may become prohibitively large as the complexity of the domain increases. Unfortunately, in none of the work above has the size of the feature space been measured, nor the effect of the search strategy in guiding the method through the space; therefore it is difficult to quantify the limitations of these data-driven systems.

Data-driven methods are also sensitive to the choice of functions used to construct new features. Most of the methods surveyed above employ very general constructors, such as *and*, *or* and *not*. This generality gives the data-driven methods domain independence, since these constructors are sufficient for generating any binary function of the input features. However, their generality leads to the large search space problem just mentioned. It is possible to employ more specific constructors, as LIVE does. This may reduce the search space, but for a given domain it may not be obvious whether a given set of constructors is sufficient for deriving useful features. This choice of constructors may be seen as a second-order bias problem: if the constructors are ill-chosen they may prevent the feature generation component from generating useful features, which will prevent the concept formation component from deriving an acceptable concept from the examples.

The second general class of methods may be termed *theory-driven*, in that they employ some theory about their domain to guide feature generation. MIRO and STABB are theory-driven systems, although their domain theories are very different: STABB assumes a set of operators specifying legal state transitions, and MIRO assumes a set of rules that deduce abstractions.

Theory-driven methods can overcome the limitations of data-driven methods. The feature search space in these systems is considerably smaller: STABB performs no feature-space search at all, and MIRO searches the abstract space L_A, which is assumed to be much smaller than the set of all descriptors.

The primary limitation of theory-driven methods is that they rely too heavily on deduction, and they are restricted to creating features that follow deductively from the domain theory. This strong statement should be qualified, however. MIRO does perform induction over descriptors deduced using a domain theory, but MIRO can only use features that are logically implied by the domain theory. It cannot combine features, as CITRE does, or alter existing features, as BACKPROP can. Furthermore, MIRO places strong restrictions on the domain theories that it can use: they must be representable as a finite acyclic AND/OR graph, so neither recursive rules nor state-changing operators are allowed. STABB is able to combine features using its method of least disjunction, but STABB cannot alter features that have been derived deductively.

There are a number of reasons a system would benefit from being able to perform both deductive and inductive operations.

- Features are known to exist that are strongly predictive of a concept, but do not strictly imply it. The X SQUARE feature of OTHELLO, discussed in Appendix A, and the GUARD feature of checkers [Samuel, 1959] are examples of these. The significance of these features can be informally justified using the domain theory, but not proven. Therefore no purely deductive system could derive them. It is also very unlikely that they could be derived by a purely inductive system that generalized from instance features using no knowledge of the rules of their respective games.

- Whether created by constraint back-propagation or by feature combination, features can become arbitrarily complex. It is necessary for a constructive induction system to be able to simplify and generalize features so as to reduce cost. While some simplification and generalization can be done deductively (such as in Prodigy's compression phase [Minton, 1988]), eventually cost can only be reduced by sacrificing deductive soundness. Sacrificing deductive soundness is an inherently inductive operation.

- Simplification of features may be desirable for other reasons, such as gaining additional generality. MetaLEX [Keller, 1987] demonstrated the benefits of assuming, for the purpose of simplification, that a condition is always true or false, even when it is not. Simplifying a feature that is too expensive is an alternative to discarding it.

- A purely deductive method cannot work with an intractable domain theory. By definition, an intractable domain theory is one in which solution sequences are so long that conditions *cannot* be propagated along their length, because the complexity of the resulting conditions become prohibitive. Any feature generation method using goal regression with an intractable domain theory must be able to regress goals through operator sequences that are shorter than full solution paths. In doing so, the method is compromising the truth-preserving nature of deduction.

In summary, most of the approaches taken to constructing new features have been data-driven, and have not taken advantage of domain information. Therefore, the approaches are mostly limited to domains in which the instance-level features are already appropriate for the concept learning. MIRO and STABB both make better use of domain knowledge, but they have shortcomings as general models of constructive induction because they rely heavily (almost exclusively) upon deduction. In contrast, the approach presented in the next section combines aspects of both analytical and empirical learning.

3 A Theory of Constructive Induction

The purpose of this research is to propose a theory of constructive induction that combines theory-driven and data-driven approaches to feature generation. The next section will present the model on which the theory is based.

3.1 The Model

The model comprises a set of general assumptions and a set of specific components that are assumed to exist within an induction system.

3.1.1 General assumptions

Most of the current work in constructive induction assumes that the features are being used for concept learning, but makes no assumption about the purpose or context of the concept learning. The following assumptions specify the model of the learning context in which the theory will hold. These should be considered general assumptions that both constrain the problem and specify the information available to be exploited.

- **A1:** The purpose of concept learning is to improve some problem-solving process.
- **A2:** The purpose of the problem-solving process is to find a solution to the goal; or if the solution quality can be measured, the highest quality solution possible.

Two assumptions are made about how the problem solver searches for solutions, and how the concept learner improves problem solving:

- **A3:** The problem solver pursues a goal by performing a search within a state-space. The problem solver generates a set of successor states and evaluates the set, choosing the best.
- **A4:** The problem solver uses the learned concept to determine the best of a set of states. No assumptions are made about the method that it uses for choosing the best. The concept learner may evaluate each state independently, or evaluate subsets of states.

Assumption A4 will be elaborated upon later, when the concept learner is presented. Two assumptions are made about feature discovery:

- **A5:** The purpose of feature discovery is to aid the concept learner in evaluating sets of states; specifically, in distinguishing states that are closer to the goal from those that are further away.
- **A6:** The feature discovery component has access to a theory of the domain. The domain theory is expressed declaratively, and includes:

- The goal of the performance element.
- The definitions of operators in the domain, including code for computing both pre- and post-images of conditions with respect to operators.
- Definitions of ancillary functions and predicates used in the goals and operators. The ancillary functions and predicates are defined down to the operational (instance-level) predicates.
- Meta-information about the functions and predicates; specifically, which are *operational* and which are *state-dependent.*

  An operational predicate is one that matches surface (input representation) features. In a board game, the predicates $owns(Player, Square)$, $neighbor(Square1, Square2)$ and $blank(Square)$ would be operational; whereas $won(Player)$ and $legal\_move(Player, Square)$ would not be.

  A state-dependent predicate is one whose truth value can be directly or indirectly changed by an operator, and thus depends on the state. For example, $owns(Player, Square)$ and $won(Player)$ are state-dependent, but $neighbor(Square1, Square2)$ is not.

The specific components of the model are based on the assumptions above. The components are depicted in Figure 2, and are described in the next four subsections.

3.1.2 The performance component

Assumptions A3 and A4 establish that the problem-solver pursues a goal by performing a state-space search, and that the concept learner aids the problem-solver by determining which node to expand next. In order to make these choices, the performance component uses a *preference predicate* [Utgoff & Saxena, 1987] based on the 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 because it can be learned from state-preference 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.

It is assumed that the performance system has access to this qualitative information. Specifically, it is assumed that at the end of a "performance episode", the performance component can extract a solution path. For example, if the performance task is a two-person game, then the set of moves made by the winner can be used as such a solution path.

3.1.3 The concept learning component

The concept learning component is the "induction" component of Figure 1 on page 2. It learns a concept representing the preference predicate used in problem solving. The preference predicate could be represented in many different ways: a set of rules, a connectionist

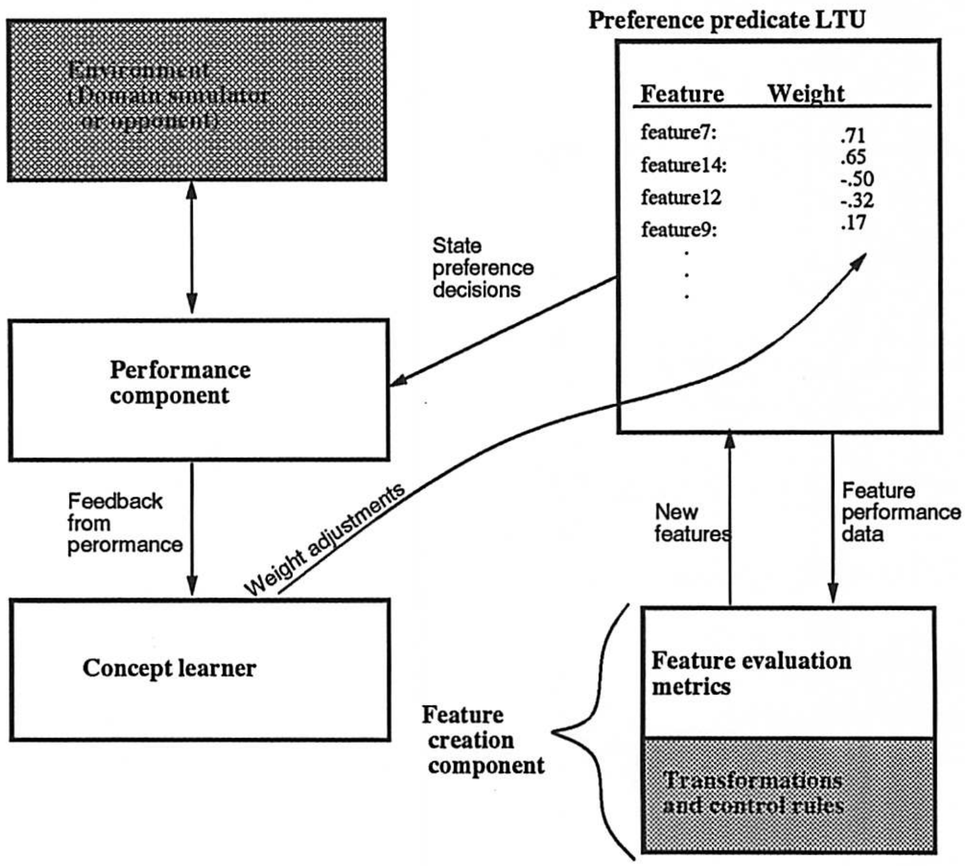

Figure 2: The components of the model

network, or a decision tree. One of the simplest possible concept forms, a linear threshold unit, was chosen for several reasons. An LTU is very fast both to train and to use. 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.

Additionally, the weights of the LTU are scaled and 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, and therefore the more useful the feature is to the problem solver. This use of the LTU's weights is not critical, however, and other measures could be adopted that would evaluate the discriminability of features independently of the LTU.

After the performance component completes a problem-solving episode, the correct solution path is passed to the concept learning system, which then creates preference pairs from it with which to train the LTU. 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 best choice in state S_p is S_b, then for every alternate choice resulting in a state S_a a preference pair $\langle S_b, S_a \rangle$ is generated. These pairs are then used to train the LTU, which is in turn used by the performance component to influence node selection.

3.1.4 The feature set

A feature is a function from a state to a real number. Every feature is scaled so that its range is between -1 and $+1$, for the purpose of comparing weights. Every feature has associated with it static information such as its definition and the transformations that created it. It is possible that a feature can be created by more than one sequence of transformations; if this is the case, the additional sequences are recorded as well.

Each feature also has performance data associated with it, to be used by the feature discovery system. These data include:

- The range of the feature before scaling.
- 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.
- An estimate of the cost of the feature. This could be either the average time cost, measured empirically, or a zeroeth-order estimate of the amount of computation performed.
- A composite "worth" score, based on the feature's weight and cost. This will initially be computed simply as $|weight|/cost$.

Although features are not discarded, if a feature's worth drops below a certain threshold, it is declared inactive and is not used in the performance system's evaluation function. This is similar to the strategy used in Samuel's checker player [Samuel, 1959], with two differences.

Samuel's system kept a count of how many times a feature had the lowest value, and would only deactivate a feature after its count exceeded 32. When Samuel's system deactivated a feature, it was dropped into a reserve queue and could eventually make its way to the front and become active again. Samuel's approach is more conservative than the simpler one used here, though a more elaborate approach like Samuel's may be adopted if necessary.

3.2 A Theory of Feature Discovery

The feature creation component is responsible for creating, evaluating and improving the feature set, and will be the primary contribution of this thesis.

A body of rules called **transformations** creates new features from existing features. The IF portion of the rule determines whether the transformation should be applied, and the THEN portion actually creates the new feature and adds it to the feature set (and performs the attendant bookkeeping). The transformations are biased to consider features of higher before lower worth.

Conceptually, the IF portion of a transformation may be separated into two sets of conditions: *necessary* conditions, which must be satisfied in order for a new feature to be created; and *recommended* or *control* conditions, which should be satisfied in order for the new feature to be useful. This distinction is similar to that between LEX's operator preconditions and operator heuristics. The recommended conditions represent search control knowledge. The transformations and their associated control conditions are discussed separately, although they are integrated in the implementation.

3.2.1 Classes of transformations

The classes of transformations are categorized below. The theory will specify the general effect of each transformation, but the particular choices will be considered details of the implementation. In these descriptions, destructive changes are implied (e.g., "dropping the condition of a feature") for convenience. However, it should be emphasized that *features are not changed* by these transformations; the result of a transformation is a *new* feature with the changes made. As in STAGGER, features are modified constructively so that they may compete with their unmodified constituents.

- **Goal-to-feature transformations.** These produce an initial set of features from the goal expression, for use in the evaluation function, and so are only used once. The goal-to-feature transformations will be similar to those used by Callan (1989). For example, if the performance goal is to achieve the relationship $f(s) > g(s)$, then one such transformation will produce features measuring $f(s)$ and $g(s)$ independently.

  The justification for these transformations is that they create features that provide information on the degree to which a performance goal has been satisfied in a state.

- **Definition-expansion transformations.** If a feature uses a non-operational predicate, and the domain theory contains a definition for the predicate, then it is possible to expand (unfold) the definition into the feature. There are a number of reasons for doing this. In general, expanding a predicate allows other transformations (e.g., condition-dropping and condition-regression transformations) to apply. It may also enable truth-preserving simplifications described below to apply. For example, if a feature contains the conditions $A \wedge B \wedge C$, and the predicate B can be expanded into $\neg A \wedge D$, then the simplifier can reduce the condition to *false* and eliminate it.

In general, definition-expansion transformations do not change the semantics of features. However, one such transformation replaces a call to a recursively-defined predicate with the predicate's base case. This transformation thus can change the semantics of a feature by specializing it.

- **Condition-regression transformations.** If a feature is of high worth, its conditions are matching an aspect of states that are useful for the performance system to distinguish. It may also be useful to look for the conditions under which such an aspect will arise, earlier in the problem-solving process. In other words, if a feature is useful, then it may also be useful to recognize states one step away from when the feature would change.

  Feature regression transformations produce new features by back-propagating (regressing) the conditions of an existing binary feature through the domain operator(s). For example, given a feature that recognizes a catastrophic event, a regression transformation can produce a feature that is activated when the performance system is one step away from the catastrophe. Regressing the conditions of a feature allows the performance system effectively to extend its lookahead for high-worth features.

- **Condition-dropping transformations.** These transformations can apply to both conjunctions and disjunctions. Dropping a condition of a conjunction generalizes it; dropping a condition from a disjunction specializes it. Generalizing a feature increases its applicability while decreasing its cost; specializing a feature can increase its discriminability by excluding negative instances. For example, assume a feature contains the condition $\neg(A \wedge B \wedge C)$. One condition-dropping transformation might look for a negation in such a condition and remove one of the conjuncts, producing $\neg(B \wedge C)$.

  It is important to note that no condition-*adding* transformations are included. This is justified because features are created (by goal-to-feature and condition-regression transformations) using exact conditions from the domain theory. It is not necessary to add conditions back in after they have been removed.

- **Feature-combining transformations.** These consist of transformations for conjoining and disjoining features. Such transformations are common in the data-driven systems reviewed in Section 2. An example of when this is useful is the C OR X SQUARE feature of OTHELLO. C and X squares are two different kinds of squares in OTHELLO. Although they are not mentioned as part of the game, they are both strategically significant for the same reason, although they may be discovered independently. A disjunction creating transformation may propose to join two features discovered independently but through similar means.

After a new feature is created by a transformation, the feature is given to a simplifier that tries to apply standard truth-preserving simplifications to it. For example, the simplifier will remove multiple occurrences of the same condition, and will transform $A \wedge \neg A$ to *false*. The simplifier also performs elementary partial evaluation; for example, if a feature contains an expression like $A = B$, and A and B are both constants, then the evaluation is done by the simplifier and the result is used in place of the original expression. When the simplifier is finished, the resulting feature is added to the feature set.

These transformations combine aspects of both analytical and empirical learning, and this is a primary difference between this theory and previous work. The goal-from-feature, definition-expansion and condition-regression transformations use the domain theory in generating new features, and so are able to perform analytical transformations of the existing features. In addition, the condition-regression transformations can apply to any feature, not just those derived from the goal, and so are more general than those used in STABB and MIRO. The condition-dropping and feature-combining transformations perform inductive, data-driven alterations to features that are common in previous contructive induction approaches.

3.2.2 Issues with the choice of transformations

The theory will specify the general action and effect of each transformation, but the particular choices will be considered details of the system. Nevertheless, the general classes leave much latitude in the design of these transformations, and the first main research issue is:

What is a good, parsimonious set of general feature discovery operators?

A related issue is that of the appropriate *grainsize* of the transformations. For example, one could imagine the following condition-removing transformations:

- Delete any single condition.
- Delete a state-dependent condition.
- Create partitions of the feature's conditions based on their relations, then remove conditions randomly until only one condition remains within each partition.

There is a tradeoff between generality and tractability in the choice of grainsize. The smaller the grainsize, the more general the transformations, and the greater the number of features that can be created. However, the more general the transformations are, the more of them there are that are applicable throughout the derivation, and the longer each derivation becomes. Thus, increasing the generality increases both the depth and branching factor of the search tree, and the need to control such transformations becomes a critical issue, discussed in the next section.

However, designing more specific transformations leads to the danger of ending up with transformations appropriate only for a single domain (or worse, transformations that can only discover the features that the system was designed to discover).

3.2.3 Issues with the control strategy

The discussion of the transformations concentrated on what the transformations do, but little mention was made of when or why each transformation is invoked. This leads to the second main research issue:

What is a good, general strategy for controlling the system?

It should be emphasized that there are actually *two* control issues: how to control the components of the *entire system*, and how to control the transformations within the *feature generation* component.

System control. It is assumed that the entire system is cyclic, performing the following steps in order:

1. Solve a problem using the LTU (trained on the current set of features) to direct search. When done, extract training instances from the search tree and add them to the global set of examples.
2. Train the LTU on the global set of examples. This establishes weights for the features.
3. If the LTU accuracy is less than 100%, invoke the feature discovery component to suggest new features. Add resulting features to the feature set.

This simple control strategy for the entire system suffices for the current implementation, and nothing more complicated should be needed.

Transformation control. The control strategy for the transformations is linked to the choice of transformations. If the transformations are general, they will be applicable at many points, causing a large branching factor in the feature search space. This places the burden on the control rules, which must be judicious in applying the transformations. On the other hand, if transformations are more specific, control is less of a problem, but the generality of the system is reduced.

The current feature generation control strategy is very simple, and may be expressed as follows:

- Features are ordered by *worth*. When the feature generation component is invoked, transformations are applied to the highest-worth feature, then the second highest-worth features, etc. All transformations that apply are fired. The feature generation component stops when n new features have been generated. n is a system parameter.

- After new features have been generated, the n lowest-worth old features are deactivated. An inactive feature is not used by the LTU, and is not considered in feature generation. Pruning occurs *after* generation so that all active features are considered in feature generation.
- A transformation will not generate a new feature if that feature would be identical up to variables to an existing (either active or inactive) feature. This rule prevents features from being generated more than once.

A good control strategy for the transformations is one of the main issues of this research, and the policy above should be seen as a base onto which more control rules can be added. Experiments with the initial implementation indicate that search control should be improved: with the n parameter set to infinity, every feature generation step creates about 15 or 20 new features, and many of these turn out to be redundant or useless. Approximately 250 features are generated in the course of deriving the features in Section 3.3. It should be noted that generating this many features does not make the current strategy infeasible — given 24 hours of CPU time, all of these features could be generated and tested — but a more sophisticated control strategy should be developed, if only to increase the ratio of "good" to "bad" features.

This generate-and-compete control strategy has been used in other AI systems, the earliest of which was probably Selfridge's (1959) Pandemonium system. Selfridge also talked about the "worth" of demons in Pandemonium, but this measure did not take into account the cost of their computations. Lenat's (1983) AM system for mathematical discovery maintained two separate rating systems. Each concept had an *interestingness* value, which was an integer assigned by a set of heuristics. Items on AM's agenda, which proposed new operations to be performed, competed with each other according to a *worth* measure. The worth of an agenda item was a complex function of the concept and operation involved, as well as the reasons for performing the operation. The agenda items competed directly with one another on the agenda, and they were strongly influenced by the concepts involved; new agenda items would be suggested based upon the worth of existing concepts. Holland's (1986) classifier system ranked classifiers by their *strength*, with the weakest classifiers in the population being replaced by new offspring. The strength of a classifier is calculated by Holland's bucket-brigade algorithm, and is based upon the participation of the classifier in the generation of solutions.

Sources of information available. In developing a more sophisticated control strategy, several sources of information can be exploited. Theoretically, any of the other components could provide information:

- In most constructive induction systems the **concept learner** provides feedback in the form of the concept itself. The model used by this proposal specifies that the concept learner exploits the weight of a feature in estimating its worth. CITRE and FRINGE both use the form of the learned concept to guide feature creation, but this makes their approaches applicable only to decision trees.

- The **performance element** is typically not used to provide feedback because most constructive induction systems do not assume the presence of one. However, it may be possible to use performance information in selecting features and transformations. For example, the concept of *fork* exists in many board games, and may be defined as a search node at which all child nodes have significantly lower evaluations than the parent node. Such search space information, along with specific feature values, could be used to create a feature to detect the fork.

- The **instances themselves** are used by STABB to direct its back-propagation, and so may be useful in goal-regression transformations. Unfortunately, the model does not use STABB's well-defined indication of concept failure (version space collapse) which enables STABB to concentrate on a single example, so it may be difficult to exploit the instances in this manner.

3.2.4 Issues with evaluating features

As mentioned earlier, the features are kept in a set ordered by their *worth*, the worth being some composite measure of cost and benefit. Since this worth determines which features are used in the concept, as well as which will serve as the basis for new features, the third research issue is:

How should the usefulness of features be evaluated?

There are three subdivisions of this question.

1. How should feature **benefit** be measured? There must be some policy for determining the contribution of an individual feature to the accuracy of a concept. If features are evaluated independently, as in STAGGER and CITRE, then problems arise because combinations of the features may be strongly predictive of a concept while the individual features are poor predictors. On the other hand, if the concept learner is an LTU, multi-colinearity becomes an issue: if several features are co-linear, the assigned weights would be less than if any of the features were used individually.

2. How should feature **cost** be measured? Every feature incurs some cost in its application; but although most constructive induction systems measure the discriminability of their features in some way, none measures cost. A first approach is simply to measure empirically the amount of time a feature takes to evaluate, averaged over many states. This method should probably suffice, but a more sophisticated analytical technique can be adopted if necessary.

3. How should cost and benefit be combined into **worth**? There is a tradeoff between efficiency and discriminability that must be recognized. Very useful features exist that can perform complex calculations (and essentially look ahead one or more moves), but the computation that implicitly performs this lookahead is very expensive and in many

cases is not worth the discriminability gained [Berliner, 1984]. One the other hand, there are some features, such as those in the OTHELLO board game that are concerned with X and C squares, that are based on lookahead but are both efficient and strongly predictive of move quality. As a first approach, the simple ratio of $benefit/cost$ will be used.

3.3 An Example Derivation

This section demonstrates the derivation of several features useful in the game of OTHELLO. OTHELLO is a two-person game played on an 8×8 board. The rules of OTHELLO and some of the related concepts are explained in detail in Appendix A. OTHELLO was chosen as a domain for a number of reasons. It has fairly simple rules, but the strategy required to play it well can be surprisingly complex. There are several computer programs that can play OTHELLO well and whose details have been published [Rosenbloom, 1982; Lee & Mahajan, 1988]. In addition, Donald Mitchell has written a Master's thesis [Mitchell, 1984] containing information on virtually all known OTHELLO features. The thesis contains not only feature definitions, but the correlation of each feature with winning for novices and experts in beginning, middle and late game play. Having such a catalog of features is a great advantage in investigating their discovery; furthermore, new discovered features can be evaluated by comparing their correlations with those given in Mitchell's thesis.

In this derivation, little information about the control strategy — *why* a particular step was taken — will be given. The derivation should be viewed as an illustration of the transformations and the feature representation, rather than as support for the theory. Section 3.3.4 is a short discussion of the implementation.

3.3.1 Some Prolog background

In order to explain the following feature derivation, a few details about Prolog and the feature representation must be explained. Variables begin with upper-case letters, literals are in lower-case, and a set of terms separated by commas is a conjunction of the terms. The notation ***conditions*** will be used to indicate some conjunction of conditions, and ***vars*** to indicate a list of variables. In Prolog rules, the left- and right-hand sides are separated by a "left arrow", written as ": –". Lists are enclosed in square brackets, e.g. $[a, b, c]$.

Each feature is of the form:

$$feature_i(Value) : -\text{count}(\boldsymbol{vars}, \boldsymbol{conditions}, Value)$$

Notice that the board is not an argument to a feature; there is always an implicit "current board" to which a feature applies. The value computed by $feature_i$ is bound to $Value$. Count is a predicate that collects the set of all values of ***vars*** that satisfy ***conditions*** and binds $Value$ to the size of the resulting set[2]. This is best explained by example:

[2]Those familiar with Prolog will recognize the close similarity of count to setof, upon which it is based.

$$\mathbf{count}([X], \textit{conditions}, \textit{Value})$$

is equivalent to the mathematical expression:

$$\textit{Value} = |\{X \mid \textit{conditions}\}|$$

and

$$\mathbf{count}([X, Y, Z], \textit{conditions}, \textit{Value})$$

is equivalent to:

$$\textit{Value} = |\{(X, Y, Z) \mid \textit{conditions}\}|$$

That is, all unique tuples of (X, Y, Z) are counted.

The variable list *vars* can also be empty, and this form is used in a *binary feature*:

$$\mathbf{count}([], \textit{conditions}, \textit{Value})$$

so Value will be bound to 1 if *conditions* can be satisfied at all, else 0.

This feature representation is similar to that of Michalski's (1983) counting arguments rules. It 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. With an empty *vars* list, a feature is binary; adding variables that occur in *conditions* into *vars* has the effect of refining it.

3.3.2 The derivation of MOVES

Initially, the system has a complete domain theory of OTHELLO, sufficient for a human to play the game. The domain theory is coded in Prolog, and contains information about the rules of the game (encoded as the MOVE operator, along with its preimages), the topology of the board (using SQUARE and NEIGHBOR predicates) and a given board's configuration (using OWNS and BLANK). Meta-knowledge is assumed about the operationality and state-dependence of the predicates in the domain theory. The definitions of WIN, LOSE, END_OF_GAME and other supporting predicates are available. There are two players, x and o, corresponding to black and white, the goal is to win the game for player x.

The first step is to apply its goal transformations to the WIN definition to create the initial set of features. Although any predicate can be trivially converted into a binary feature, such features provide little information that can be used to direct search. Since the WIN predicate is a conjunction, it is broken apart and heuristics are reapplied to the conjuncts.

From the original goal specification, two significant features can be created: one that counts the number of discs owned by x (from the WIN definition), and one that counts the number of moves available for x (from the END_OF_GAME definition). The definition of the latter is:

```
feature7(Value) :-
        count( [Square], legal_move(Square,x),  Value).
```

Square is the square of the move, so this form counts the number of distinct moves for player x. This is the definition of the MOVES feature [Mitchell, 1984, page 47]. This feature, apart from being useful in its own right, forms the basis for a number of other useful mobility features.

3.3.3 The derivation of other mobility features

Since legal_move is defined by the domain theory, it can be expanded, yielding feature8:

```
feature8(Value) :-
        count( [Square,Bracket],
                       (square(Square),
                        bs(Square,Bracket,x)),
                Value).
```

If the definition of bs is expanded, feature9 is derived:

```
feature9(Value) :-
        count( [Square,Spanbegin,Opp,Spanend,Dir,Bracket],
                              (square(Square),
                               blank(Square),
                               opponent(x,Opp),
                               direction(Dir),
                               neighbor(Square,Dir,Spanbegin),
                               span(Spanbegin,Spanend,Dir,Opp),
                               neighbor(Spanend,Dir,Bracket),
                               owns(x,Bracket)),
               Value).
```

This set of conditions defines a pattern consisting of a blank square at Square, a span of opponent's pieces from Spanbegin to Spanend, then a final bracketing opponent's piece at Bracket. The last predicate, owns(x,Bracket), is the bracketing condition. By removing this condition, the system can generate feature13:

```
feature13(Value) :-
        count( [Square,Spanbegin,Opp,Spanend,Dir,Bracket],
                              (square(Square),
                               blank(Square),
                               opponent(x,Opp),
                               direction(Dir),
                               neighbor(Square,Dir,Spanbegin),
                                span(Spanbegin,Spanend,Dir,Opp),
                                neighbor(Spanend,Dir,Bracket))
                 Value).
```

This feature now counts the number of *un*bracketed spans emanating from each blank square. Finally, if we substitute the span call with its "base case", we get:

```
feature14(Value) :-
      count( [Square,Spanbegin,Opp,Dir,Bracket],
                               (square(Square),
                                blank(Square),
                                opponent(x,Opp),
                                direction(Dir),
                                neighbor(Square,Dir,Spanbegin),
                                owns(Opp,Spanbegin),
                                neighbor(Spanbegin,Dir,Bracket))
                 Value).
```

This feature is the basis for three mobility-related features used by Rosenbloom in his world-championship-level program [Rosenbloom, 1982]. If the count form is restricted to counting only the blank squares of this pattern — if the variable list of the count form is [Square] — then the resulting feature is equivalent to ROSENBLOOM EMPTY [Mitchell, 1984, page 49], which counts the number of empty squares that have an opponent's piece as a neighbor.

If the count form is restricted instead to the opponent's pieces (the variable list being [Spanbegin] in this case), then the resulting feature is ROSENBLOOM FRONTIER [Mitchell, 1984, page 112], which counts the number of pieces for each player that have at least one empty neighbor.

If instead the distinct *pairs* of B and C are counted — the variable list being [Square,Spanbegin] — then the resulting feature is ROSENBLOOM SUM EMPTY [Mitchell, 1984, page 113], which counts the number of empty squares next to each opponent's piece.

The derivations of these four mobility features, along with a number of stability features, are depicted in Figure 3.

3.3.4 A note on the implementation

This theory has been implemented in a system called Zenith, which is written in a combination of C and Prolog and runs on a Sun-4 Sparcstation. Its opponent is WYSTAN, an OTHELLO-playing program written by Jeff Clouse. Zenith contains eight transformations so far, from each the classes listed above except feature-combination. Zenith can generate all of the mobility-related features mentioned in Section 3.3, and all of the stability-related features in Figure 3 except for CORNER SQUARES and its descendents (on the left-hand side of the figure).

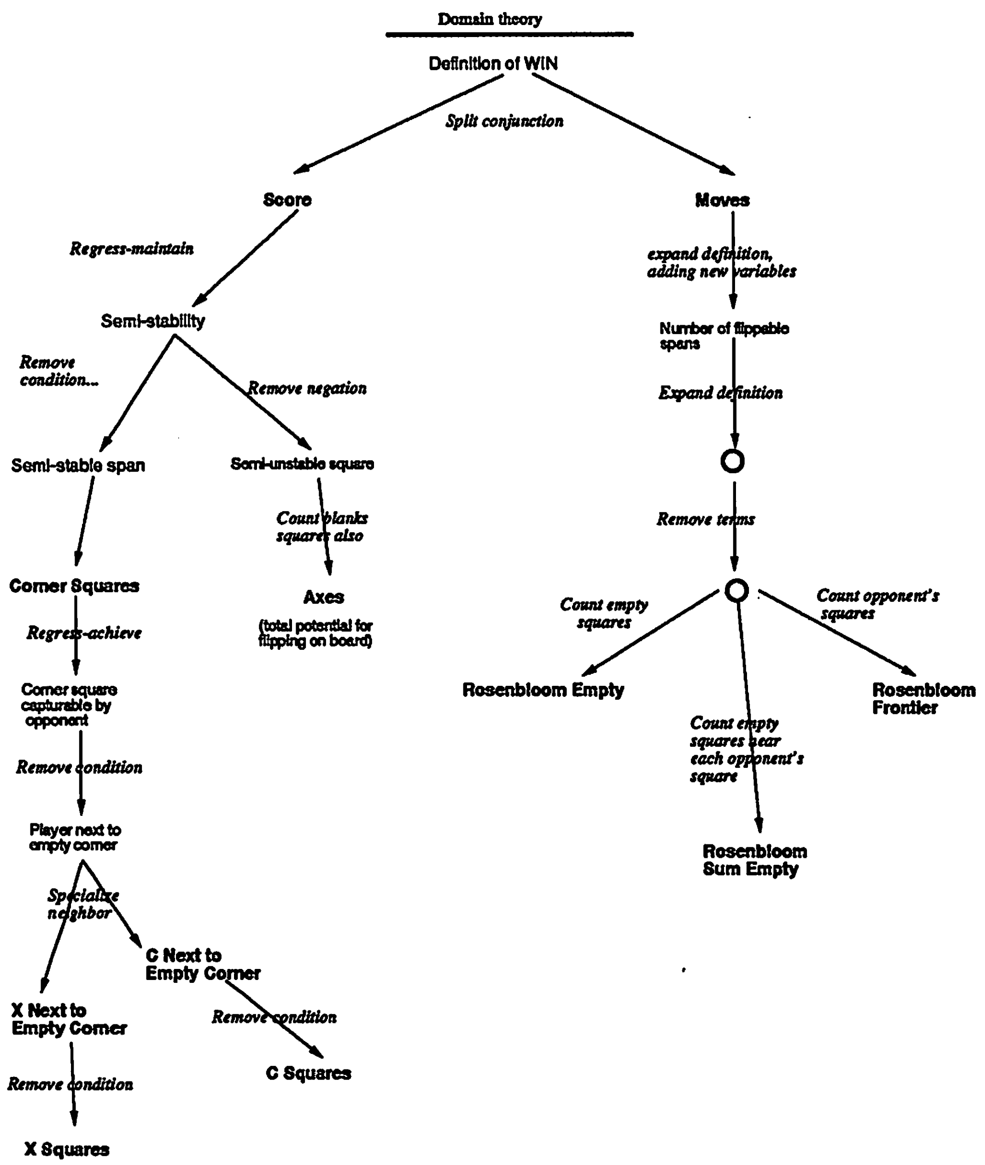

Figure 3: The derivation of some OTHELLO features from the original OTHELLO domain theory. Names of commonly known features are in boldface; descriptions of operations are in italics.

4 Proposed Work

This work on feature discovery should be considered exploratory, in that little work has been done on the combination of empirical and analytical techniques to support an inductive process. The investigation of this theory will therefore be empirical, and the result will be a characterization of the kinds of features that it is able and unable to create. This in turn will enable an analysis of the abilities and limitations of the approach. Comparing it to existing constructive induction systems will establish its relative performance, and will make a compelling argument for the necessity of a hybrid approach.

4.1 Domains

In order to demonstrate generality of the theory, Zenith will be applied to two different domains. The first will be the game of OTHELLO, already discussed, and the second will be the task of telecommunications network management (discussed below). The two domains are different enough to provide convincing evidence for the generality of the approach.

The tests will entail applying the system to the theories of the domain and examining the features generated and their derivations. Where applicable, existing feature generation techniques will be applied to the same domains to compare the performance of the features they may derive, along with the computational effort involved in the derivation.

The second domain, that of telecommunications network management, also has a fairly concise but intractable domain theory; however, it is much different from OTHELLO. Telecommunications networks and their management are discussed in Appendix B. The goal is to maximize an evaluation function by individual moves, rather than to beat an opponent. Viewed as a state-space search, a state transition is the application of a control to the network, a control being a local policy change of the way in which calls are routed.

One example of a control is a *destination re-route* (DRR), which is placed on a switch to force it to route all calls, bound for a given destination, via a particular communications channel. The DRR control takes as arguments the switch to which it is applied, the destination involved (another switch), and the communications channel. An average network has about ten switches, each of which is connected to about three others, so a DRR has about $10 \times 10 \times 3 = 300$ legal argument values. If there are four other controls, the branching factor is about 1500. Solution paths are typically not very long (≈ 6 moves, where one move is the application of a control). Thus the search space is "branchy and short" rather than "thin and deep". An additional difference is that many features in OTHELLO are "pattern-based", since moves in OTHELLO are essentially pattern transformations. In contrast, the flow of call traffic through a network is based primarily on channel capacities and network connectivity.

One of the disadvantages of this second domain is that, while a simulator and several problem-solving systems do exist for it [Frawley, Fawcett & Bradford, 1988; Silver, Frawley, Iba, Vittal & Bradford, 1990; Silver, Vittal, Frawley, Iba & Bradford, 1990], the features and important concepts in it are not well known. It will be possible to measure performance improvement based on discovered features, but there is no comprehensive catalog of existing features that can be used for comparison.

4.2 Experiments

Because there are many components to the Zenith architecture, many experiments can be done to test the sensitivity of the approach to the choice of these components.

- The **domain.** This is probably the most significant variable. The choice of domains was discussed in Section 4.1.

- The **representation of the domain theory**. It is expected that the representation of the domain theory will have some effect on the efficiency of Zenith's feature discovery. The representation may determine how many transformation applications it takes to derive a feature, the order in which features are derived, and possibly whether a given feature can be derived at all. In order to investigate the sensitivity of Zenith to domain theory representation, I plan to apply Zenith to two OTHELLO domain theories: one written by me and one written by another person, someone who is unfamiliar with OTHELLO features and the transformations used in Zenith.[3]

- The choice of **concept learning** component. It is possible that the features discovered by Zenith are useful only to linear threshold units. This could be tested by replacing the LTU with another inductive concept learning method; for example, ID3 with numeric attribute partitioning [Quinlan, 1986].

- The **feature set size**. If the feature set is of unbounded size, Zenith will be able to generate any feature in the transitive closure of its transformation set. Realistically, Zenith's feature set must be of some small bounded size. How sensitive is the discovery process to this size? One would expect that the smaller the size of the feature set, the more critical the search control becomes, because Zenith can afford fewer "bad" features. Decreasing the size is also a test of hill-climbing by feature weight: if a high-worth feature can only be generated from a low-worth one,the low-worth feature must survive long enough for the generation to occur.

  One would hope that Zenith's approach is robust with respect to feature set size; otherwise, the quality of the search control is brought into question. In order to test this, a group of derived features (such as those in Figure 3 will be chosen as a basis. The feature set size will be varied, and observations made of the number of features derivable, the length of the derivations of each, and the overall accuracy of the feature set at each step.

[3]This approach was taken by Nicholas Flann to provide evidence for the generality of his IOE algorithm [Flann & Dietterich, 1989].

4.3 Tentative Schedule

The implementation should take approximately six months. The experiments should take four months once the implementation is complete. Intermediate writing, of conference papers, grant proposals, etc., should take two months. The writing of the final dissertation should take another five months. Thus the total investigation should take approximately a year and a half.

4.4 Conclusion

It has long been recognized that inductive learning is very sensitive to the representation used to express the examples. The ability to determine "good" features automatically has been a goal in machine learning for nearly as long.

There have been many systems that generate new features automatically. Most are data-driven, and build up complex features from the instance-level representation. The feature construction is done with domain-independent operators using little or no domain knowledge in the process. While these approaches have the advantages of being very general and making few assumptions about the learning task, they are restricted in what they can achieve. Most of them work only if the instance-level representation is already appropriate for concept learning. Furthermore, the domains used in such research — tic-tac-toe, parity problems, n-bit multiplexors — raise doubts as to how well the methods scale up.

The theory presented here integrates theory-driven and data-driven methods for feature discovery. It is motivated by the desire to increase the power and range of constructive induction by going beyond the limitations of the two approaches. The successful demonstration of this theory will affect the machine learning community in several ways.

First, it will constitute a significant advance in constructive induction as one of the few systems to produce features for a "difficult" domain; that is, a domain with a very large search space and an intractable domain theory.

Second, it will demonstrate to researchers of the data-driven approach in contructive induction that domain knowledge is useful for generating features for non-toy domains. It will also demonstrate to researchers in the analytical approach that generalizing and combining features is useful for generating features for non-toy domains. It should make the same point about rules to researchers in explanation-based learning. To the extent that neither kind of system can generate features for such a domain, it will demonstrate the necessity of a hybrid approach.

Third, because it will learn successfully in a domain using an intractable theory, it will constitute a solution to the intractable theory problem, a problem that is being pursued by researchers in explanation-based learning [Mitchell, Keller & Kedar-Cabelli, 1986; Mostow & Fawcett, 1987; Tadepalli, 1989].

Fourth, because features are abstractions of states, the system constitutes an automatic transformation-based abstraction system. All other such systems require a human to control the application of the transformation. This research, using feedback from an inductive concept learner to control the transformations automatically, will constitute an advance in this field [Mostow & Fawcett, 1987; Ellman, 1988; Mostow & Prieditis, 1989].

An implementation of this theory has been undertaken, and the preliminary results are very encouraging. Feature generation *can* be automated in intractable domains. The successful completion of this work will make a significant advance in the state of the art of feature generation, and will bring us much closer to an ideal of automatic concept learning from examples.

Acknowledgements

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

I wish to thank my advisor, Paul Utgoff, for his numerous comments on the organization and content of this proposal. Paul originally suggested viewing feature generation as a discovery process. Discussions with Bernard Silver, Jamie Callan, Andy Barto, Edwina Rissland and Nick Flann were useful in developing these ideas. Further comments by Jamie Callan and Sharad Saxena helped to improve the organization of the proposal.

References

Ash, T. (1989). *Dynamic node creation in backpropagation networks* (ICS Report 8901). San Diego, CA: University of California, Institute for Cognitive Science.

Berliner, H. J. (1984). Search vs knowledge: An analysis from the domain of games. In A. Elithorn, & R. Banerji (Eds.), *Artificial and Human Intelligence.* New York: Elsevier Science Publishers.

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. (1981). Inductive learning of structural description. *Artificial Intelligence, 16*, 257-294.

Dietterich, T. G., London, B., Clarkson, K., & Dromey, G. (1982). Learning and inductive inference. In P. R. Cohen, & E. A. Feigenbaum (Eds.), *The Handbook of Artificial Intelligence: Volume III.* San Mateo, CA: 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.

Drastal, G., Czako, G., & Raatz, S. (1989). Induction in an abstraction space: A form of constructive induction. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 708-712). Detroit, Michigan: Morgan Kaufmann.

Ellman, T. (1988). Approximate Theory Formation: An Explanation-Based Approach. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 95-99). Saint Paul, MN: Morgan Kaufmann.

Fisher, D. H. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine Learning, 2*, 139-172.

Fisher, D. H., & McKusick, K.B. (1989). An empirical comparison of ID3 and backpropagation. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 788-793). Detroit, Michigan: Morgan Kaufmann.

Flann, N., & Dietterich, T. (1989). A Study of Explanation-Based Methods for Inductive Learning. *Machine Learning, 4*, 187-226.

Frawley, W., Fawcett, T., & Bradford, K. (1988). *NETSIM: An Object-Oriented Simulation of the Operation and Control of a Circuit-Switched Network* (Technical Note TN 88-506.1). Waltham, MA: GTE Laboratories, Inc., Computer and Information Systems Laboratory.

Holland, J. H. (1986). Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R. S. Michalski, J. G. Carbonell, &

T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Keller, R. M. (1987). Concept learning in context. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 91-102). Irvine, CA: Morgan Kaufmann.

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

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). Constructive induction on decision trees. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 645-650). Detroit, Michigan: Morgan Kaufmann.

Matheus, C. J. (1990). Adding domain knowledge to SBL through feature construction. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 803-808). Boston, MA: 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. (Special issue on knowledge acquisition and induction)

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.

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. M. (1977). Version spaces: A candidate elimination approach to rule learning. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (pp. 305-310). Morgan Kaufmann.

Mitchell, T. M. (1978). *Version spaces: An approach to concept learning.* Doctoral dissertation, Department of Electrical Engineering, Stanford University, Palo Alto, CA. (also Stanford CS report STAN-CS-78-711, HPP-79-2)

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

Mooney, R., Shavlik, J., Towell, G., & Gove, A. (1989). An experimental comparison of symbolic and connectionist learning algorithms. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 775-780). Detroit, Michigan: Morgan Kaufmann.

Mostow, J., & Fawcett, T. (1987). *Approximating intractable theories: a problem space model* (Technical Report ML-TR-16). New Brunswick, NJ: Rutgers University.

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.

Muggleton, S., & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 339-352). Ann Arbor, MI: Morgan Kaufman.

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.

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

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

Quinlan, J. R. (1987). Simplifying decision trees. *Internation Journal of Man-machine Studies, 27,* 221-234.

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

Rumelhart, D. E., & McClelland, J. L. (1986). *Parallel Distributed Processing.* Cambridge, MA: MIT Press. (2 volumes)

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

Selfridge, O. G. (1959). Pandemonium: A paradigm for learning. *Proceedings of the Symposium on the Mechanization of Thought Processes* (pp. 513-526). Teddington, England: National Physical Laboratory, H.M. Stationary Office, London. (Also published in *Computers and Thought,* edited by Feigenbaum and Feldman. New York: McGraw-Hill Book Company, 1963, pp. 251-268.)

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

Schlimmer, J. C. (1986). Concept Acquisition through representational adjustment. *Machine Learning, 1*, 81-106.

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

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.

Silver, B., Vittal, J., Frawley, W., Iba, G., & Bradford, K. (1990). ILS: A Framework for Integrating Multiple Heterogeneous Learning Agents. *Proceedings of Second Generation Exper Systems, 10th International Workshop on Expert Systems and Their Applications* (pp. 301-313).

Tadepalli, P. (1989). Lazy Explanation-Based Learning: A Solution to the Intractable Theory Problem. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 694-700). Detroit, Michigan: Morgan Kaufmann.

Towell, G., Shavlik, J., & Noordewier, M. (1990). Refinement of Approximate Domain Theories by Knowledge-Based Neural Networks. *Proceedings of the Eighth National Conference on Artificial Intelligence* (pp. 861-866). Boston, MA: 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. (1986a). Shift of bias for inductive concept learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Utgoff, P. E. (1986b). *Machine learning of inductive bias.* Hingham, MA: Kluwer. (reviewed in IEEE Expert, Fall 1986)

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.

Wogulis, J., & Langley, P. (1989). Improved Efficiency by Learning Intermediate Concepts. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 657-662). Detroit, Michigan: Morgan Kaufmann.

A The Game of OTHELLO

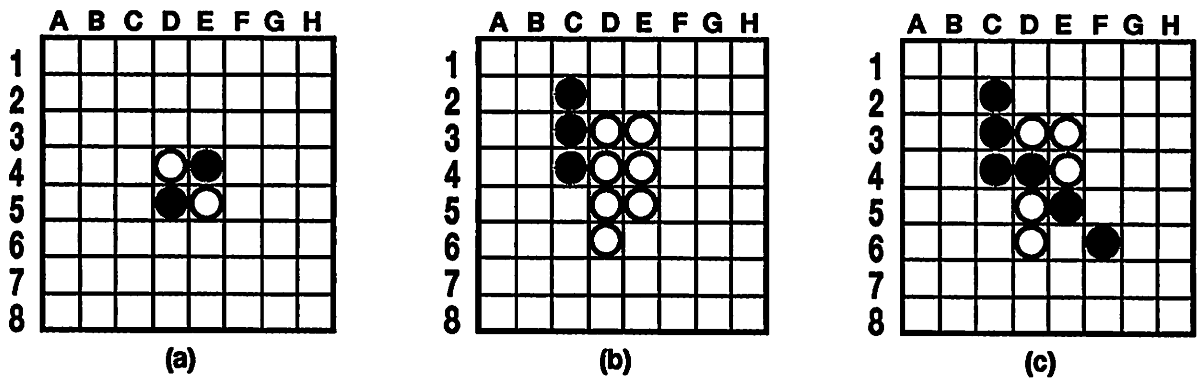

Figure 4: OTHELLO boards: (a) Initial OTHELLO board (b) Board in mid-game (c) After Black plays f6 on (b).

OTHELLO[4] is a two-person game played on an 8×8 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 4a. 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 4b, 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 4b, the board in Figure 4c 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 the number of points by which the player has won is simply the difference between the two piece counts. For analysis, an OTHELLO game is often broken up into three segments: the early game (from 4 to 16 pieces), the middle game (from 17 to 32 pieces), and the late game (from 48 pieces to the end) [Mitchell, 1984].

There are approximately 7.5 legal moves from every state[5], although this number varies quite a bit. There are usually 60 moves in an OTHELLO game, since the game usually does not end until the board is full. This yields a space of approximately $7.5^{60} \approx 10^{50}$ legal boards.

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

[5]This figure was Calculated from Figure 3.5 of [Rosenbloom, 1982].

Corner squares are inherently stable, since once they are occupied there is no sequence of moves that will flip the pieces in them. Therefore, gaining control of corner squares is a key strategy in Othello. There are two sets of distinguished squares adjacent to the corners that are significant in corner square control. The squares along the edges immediately adjacent to the corners are called C squares (they are A2, B1, G1, H2, A7, B8, H7 and G8). X squares (B2, G2, B7 and G7) are diagonally adjacent to the corners. X and C squares are both considered dangerous to own because they can allow the opponent to move into the corner. C squares are somewhat less vulnerable than X squares because it is relatively difficult to move onto an edge. X squares are more vulerable because it is easier to occupy a square on the diagonal behind an X square, and from there to move into the corner.

B Telecommunications Network Management

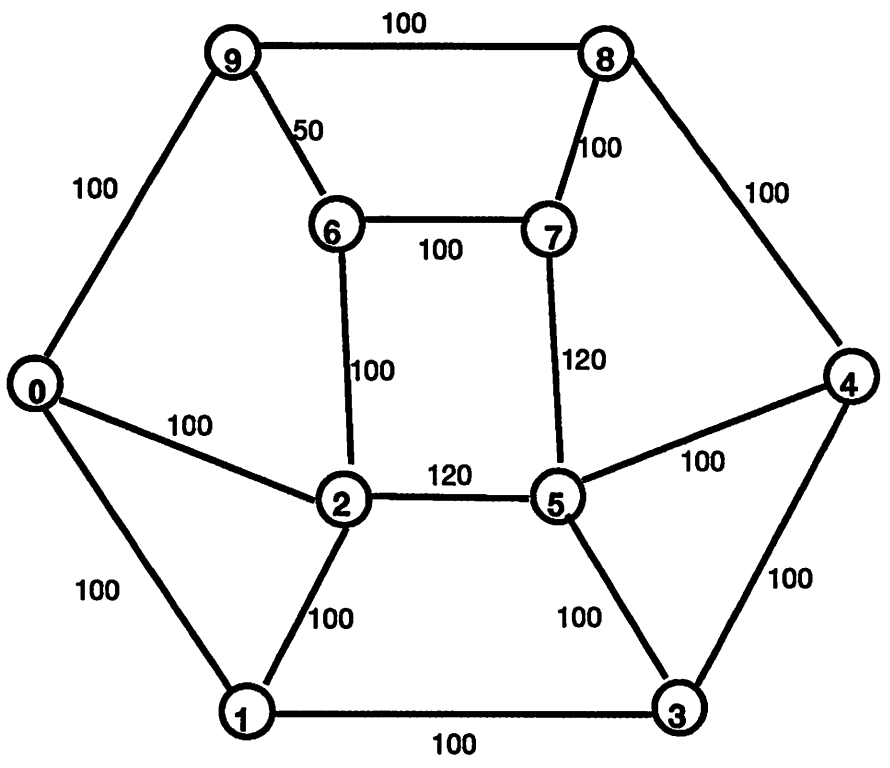

Figure 5: The ILS-10 telecommunications network. The circles represent switches, the lines between them represent trunks, and a number next to a trunk is that trunk's capacity.

The domain of telecommunications network management concerns the control of traffic in a circuit-switched network. The goal of network management is to maximize the performance of the network, according to some metric.

A telecommunications network consists of a set of switches connected by trunk lines. A simple telecommunictions network, called ILS-10, is shown in Figure 5. The goal of a telecommunications network is to allow calls between switches, by finding paths from their origins to their destinations. A call is generated outside the network with a given origin and destination. The call starts at its originating switch. The switch allocates a trunk line for the call, based on its destination, and passes the call over the trunk line to the next switch. This next switch repeats the process: it examines the call's destination, allocates a trunk line for the call, then sends the call onward over the line. When the call reaches its destination, it is said to have *completed*[6]; there is now a trunk line path from its origin to its destination through which communication can occur. For example, in the ILS-10 network

[6]This terminology may be confusing. A completed call is one that has just begun.

a call from switch 9 to switch 3 might be routed along the path $[9, 8, 4, 3]$. The trunk lines will remain allocated until the call terminates. If at some point in the routing process a switch cannot route the call, then the call is said to have *failed* at the switch, and the call is dropped completely from all trunk lines (ie, no failure recovery is attempted).

The most important part of the process is how each switch decides to route a call; that is, how each switch decides to which switch the call should next pass. Each switch has a *routing table*, which specifies for every destination an ordered list of trunks. When a call arrives at the switch, the table is consulted, and the first trunk in the list is tried. If the trunk is full or the connected switch is down, the switch tries the next trunk in the routing table. When the routing table entries are exhausted, the call fails. It should be noted that routing tables, along with the topology of the network itself, are designed very carefully so as to minimize the possibility of cycling.

Several points should be emphasized about the routing process. Each switch decides only what the next step in a call's path should be; it does not decide what path the call should take. Each switch must make this decision based solely on the call's destination, as no other information is passed along. For example, a switch cannot determine which other switches may already have routed the call, so it cannot detect cycles in the routing, nor can it detect that the call is taking an overly circuitous path to its destination. Also, the lifetime of a call is never known; once a call has completed, it stays in place, occupying trunk lines, as long as neither end breaks the connection.

Because trunks have finite capacity, traffic patterns change, and because network components sometimes fail, a network may become congested and need to have its routing behavior changed. Rather than change the routing tables, which are considered fixed, there are controls that can be placed on a switch to alter its routing behavior. Examples of these are:

- The Destination Re-Route (DRR) control, which overrides a routing table and forces a switch to re-route to another switch all calls that are going to a given destination. This is useful, for example, when a switch is known to be overloaded with calls; a DRR can be applied to adjacent switches to route traffic around to other switches.

- The Immediate Re-Route (IRR) control, which forces a switch to re-route all calls that would be have been routed onto a given trunk. This is useful when a trunk is known to have decreased capacity because it is damaged or completely inoperative.

- The Cancel-To (CANT) control, which cancels some portion of the traffic that would be routed onto a given trunk. This is useful when, for example, a trunk is damaged and traffic that would go through it cannot be routed any other way.

These are typical of the kinds of controls available. Switches typically have ten to twelve controls; newer switches have fifteen or twenty. Most controls take an extra argument, the percentage of calls to which the control applies. For example, a CANT of 50% specifies that only 50% of the calls to which it applies should be cancelled; the remainder will be processed exactly as if the control did not exist.

The goal of telecommunications network management is to place controls so as to maximize some evaluation function of the network's behavior. Typically the evaluation function is fairly simple, like the number of completed calls divided by the number of calls attempted. Existing problem-solving systems first try to diagnose the problem (eg, as being due to equipment failure or unusually high traffic into a single node) before suggesting plans to remedy it.

There are a number of simplifying assumptions made by existing systems for telecommunications network management [Silver, Frawley, Iba, Vittal & Bradford, 1990; Silver, Vittal, Frawley, Iba & Bradford, 1990] in order to make it a tractable domain:

- It is not always possible or desirable to maximize the evaluation function, since doing so may involve imposing and removing controls so quickly that true network performance is obscured by the traffic transients created by the controls. More typically, network controls are imposed only when the network evaluation functions drops below a threshold for a certain length of time.

- Any network with wildly varying traffic patterns can defy improvement, simply because of the delay between diagnosing the problem and imposing the controls. Therefore, it is commonly assumed that traffic patterns will remain "reasonably stable" for the amount of time it takes to diagnose a problem and impose controls. This assumption is not always satisfied in real-world networks, but humans also fail when it is violated.

- A network that is overloaded (operating at 95% capacity or more) can experience unavoidable call failures simply from slight variations in traffic. Therefore, it is assumed that the network is operating within some comfortable margin of its capacity.

Using these assumptions, existing problem solving systems coupled with machine learning techniques [Silver, Frawley, Iba, Vittal & Bradford, 1990; Silver, Vittal, Frawley, Iba & Bradford, 1990] are able to diagnose network problems and improve call completion averages.