Use of Domain Knowledge in Constructive Induction

James P. Callan
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

COINS Technical Report 90-95
August 16, 1990

Abstract

One of the important problems in integrating inductive learning algorithms with problem-solving systems is determining how they communicate. It is well-known that inductive algorithms are sensitive to the vocabulary with which examples of a concept are described. It is also known that a vocabulary can be acceptable for problem-solving but cause the inductive algorithm to learn slowly or inaccurately. It remains an open question how best to choose a language with which a problem-solving system and an inductive algorithm communicate.

This thesis proposal considers the representation problem for a class of architectures in which the problem-solver is a search procedure and the inductive learning algorithm is a source of heuristic guidance. It presents a solution in which the problem-solver uses its domain knowledge to generate automatically a set of features suitable for learning search control knowledge. Each new feature describes the problem-solver's progress in achieving one of its goals. Experimental evidence from two domains supports the claim that this solution results in faster and more accurate learning. A program of additional research and experiments is described that is expected to provide further support for the solution.

Contents

1 Introduction

One goal of Artificial Intelligence (AI) is to endow machines with the ability to learn from experience. Research toward this goal has produced a variety of learning algorithms (e.g. the Perceptron [Minsky & Papert, 1972], ID3 [Quinlan, 1986], backpropagation [Rumelhart & McClelland, 1986a]) and some impressive performance programs (e.g. Samuel's checker program [Samuel, 1959], MetaDendral [Buchanan & Mitchell, 1978], LEX [Mitchell, Utgoff & Banerji, 1983]). However, in spite of these successes, few of today's problem-solving systems have the ability to learn.

One reason that inductive learning algorithms are not more widely used in problem-solving systems is their sensitivity to the way in which information is represented. Some of the best-known successes in machine learning have been due in part to careful or fortunate choices of representations (e.g. AM [Lenat & Brown, 1984]). When the same algorithms are applied with less-carefully chosen vocabularies, they may fail to learn anything useful. As a result, machine learning techniques are impossible to apply without confronting one of the oldest and most pervasive problems in Artificial Intelligence: how to choose an appropriate vocabulary, or set of *attributes*, to describe objects in the domain.

The difficulty of constructing an appropriate vocabulary is sometimes underestimated, because many tasks appear to have 'obvious' representations. For example, one might expect to describe a chess board by listing the location of each piece. However, 'obvious' representations are frequently inadequate for inductive learning. It took Quinlan two man-months to design a vocabulary that enabled ID3 to learn certain chess positions [Quinlan, 1983]. In general, constructing a good vocabulary 'by hand' can take a long time, because it is essentially a manual search of the space of possible vocabularies. Even when a manual search works, its reliance upon human intuition makes it more of an art than a science; results will obviously vary from person to person.

This problem is exacerbated by the discovery that a vocabulary that is appropriate for one AI technique may not be appropriate for machine learning [Flann & Dietterich, 1986]. Since no single vocabulary will suffice, the designers of large problem-solving systems must address the representation issue at least twice: Once for the machine learning component, and again for the other components of the system.

One solution to this problem is to enable the machine-learning program to choose its own vocabulary. This approach is usually called *constructive induction* [Michalski, 1983], although it is also called the *new term problem* [Dietterich, London, Clarkson & Dromey, 1982], *feature extraction* [Kittler, 1986] and *feature generation* [Rendell, 1985]. Most implementations are incremental; whenever the program is not making reasonable progress in achieving some goal, domain-independent heuristics are used to change the vocabulary. Constructive induction may be faster than a manual search of the space of possible vocabularies, but it is still a search. The *initial* vocabulary remains important, because it determines where the search starts, and how large the branching factor will be.

Creation of the initial vocabulary has remained a manual process, partly because creating such vocabularies requires knowledge that is not normally available to an inductive learning

algorithm. This proposal presents an alternative solution. It shows how a problem-solver can use what it knows about a search problem to generate an initial vocabulary for describing search states to an inductive learning algorithm. Existing techniques for constructive induction could then be used to adjust the vocabulary, as necessary.

The following section reviews the previous work on constructive induction. Section 3 proposes a new approach, one that uses domain knowledge to generate an adequate vocabulary. The technique described in that section is the basis for the thesis proposal. Section 4 contains the thesis proposal itself. It describes the issues that the dissertation is expected to address, and the approaches that will be used. Finally, Section 5 summarizes the preceding sections and discusses the significance of the work.

2 A Brief Overview of Constructive Induction

The problem of inadequate vocabularies is not new; the first programs to adjust their own vocabularies automatically were developed in the 1950's [Minsky, 1963]. Those early programs were described as creating *properties*, while more recent programs are described as creating *variables, predicates,* and *functions* [Dietterich & Michalski, 1983], *features* [Rendell, 1985], and *terms* [Utgoff & Mitchell, 1982]. All of these different names refer to terms in the learning algorithm's vocabulary. While they can be used interchangeably, this paper will generally refer to them as *terms* or *features*.

The simplest approach to fixing an inadequate vocabulary is to add every term that might possibly be useful. This approach is effective in domains in which the number of possible terms is small. However, in most domains the number of possible terms is too large. The dimensionality of the space searched by an inductive learning algorithm is determined by the size of the vocabulary, so large vocabularies result in large search spaces. In general, large vocabularies cause slower learning; in some cases, they also cause less accurate learning [Saxena, 1989b]. Therefore, it is preferable to add only the terms that facilitate finding a concept description.

The distinction between discovering a concept and creating a new term is not, unfortunately, as clear-cut as one might expect. The definitions of *concept* and *term* are clear: A *concept* describes a set of examples, or *training instances*, while a *term* is part of a single training instance. The confusion arises because membership in a particular set might be an important characteristic of a training instance. For example, the concept *conservative* might describe a set of individuals with similar views on education, religion, crime and abortion. However, *conservative* can also be used as a term in the description of another individual, to indicate the individual's membership in a particular set. Therefore, an algorithm that builds a description of a set of *conservative* individuals can be viewed as either learning a concept or creating a term.

Confusion over the difference between learning concepts and creating terms is problematic in a review of constructive induction. Some systems are described by their authors as doing constructive induction, while similar systems are described as learning concepts. It quickly becomes clear that the distinction between the two is largely arbitrary; the only difference

is how the results are used.

This thesis proposal defines constructive induction as any process that creates new terms *in order to improve future learning performance.* Therefore, UNIMEM [Lebowitz, 1987] does constructive induction, because it forms new terms (or concepts) in order to improve performance on future classification tasks; AQ^{11} [Michalski & Chilausky, 1980], which is similar to UNIMEM, does not do constructive induction because the resulting concepts (or terms) are not used in future learning tasks.

Even with this restriction, there are many systems containing some module or subsystem that fits the definition of constructive induction. Quite a variety of techniques have been developed during the last 35 years. The general approach taken by each can be classified as belonging to one of five categories.

- **Recombination:** New attributes are functions of old attributes.
- **Partitioning:** New nominal or Boolean attributes are created by partitioning the range of a numeric attribute.
- **Clustering:** New attributes are created to characterize groups of training instances.
- **Transformation:** New attributes are found by applying transformations to one or more training instances.
- **Constraint back-propagation:** New attributes are found by back-propagating a goal state through a sequence of operators.

These five categories are intended to group algorithms according to their overall approach to constructive induction.

The following subsections survey systems that exemplify each approach. Each subsection considers two aspects of each system: The set of operators for constructing new terms, and the set of heuristics that guide the search for new terms. Finally, this section concludes with a discussion of an open problem in constructive induction.

2.1 Recombination

The recombination approach to constructive induction can be characterized as creating new attributes out of existing, or already known, attributes. It was first suggested by Selfridge, for use in his Pandemonium model of learning [Selfridge, 1959]; since then it has become the most commonly used approach to constructive induction. Recombination is popular because it requires very little information, it is easy to implement, and it is often sufficient. The major distinctions among different systems that do recombination are the operators used to combine attributes, and the heuristics used to control combinatorial explosion.

The set of operators for combining attributes is partially determined by the data types used in the representation. If the representation consists of Boolean or nominal attributes, then the operators are normally some subset of the Boolean operators $\wedge$, $\vee$ and $\neg$. Samuel's

checker program [Samuel, 1959], PLS0 [Rendell, 1985], STAGGER [Schlimmer & Granger, 1986], FRINGE [Pagallo, 1989], and CITRE [Matheus & Rendell, 1989] are examples of systems that use Boolean operators. If the representation consists of numeric attributes, then the operators are usually some subset of the arithmetic operators $+$, $-$, $*$, and $/$. Pandemonium, GMDH [Ivakhnenko, 1971], BACON [Langley, Bradshaw & Simon, 1983], and ABACUS [Falkenhainer & Michalski, 1986] are examples of systems that use arithmetic operators. PET also uses arithmetic operators, but its set is slightly different; it uses $+, *, =, **$, *successor*, and *derivative* [Porter & Kibler, 1984]. Both Boolean and numeric operators can be used when the representation consists solely of Boolean attributes, because Boolean attributes can be encoded as {TRUE, FALSE} or as $\{0, 1\}$. Mehra's framework for constructive induction depends upon this ability [Mehra, Rendell & Wah, 1989].

The unconstrained application of either Boolean or arithmetic operators yields a combinatorially explosive number of new attributes; it is therefore important how the search through the space of possible new attributes is organized. Samuel's checker program used an incremental breadth-first search of just one ply, while BACON used an ordered depth-first search. Both breadth-first and depth-first search are complete; they will eventually find every useful new attribute. However, because they are relatively unguided, they may explore a large part of the search space before finding any useful new attributes.

Exhaustive search can be avoided with the addition of pruning. The Pandemonium model of learning does a random search, but periodically prunes any 'subdemons' (terms) that contribute little to the evolving concept. Then it generates new 'subdemons' (terms) by applying randomly to the survivors its operators for combining terms. While powerful, this approach requires that the language for defining terms be chosen carefully so that random changes result in meaningful (although not necessarily useful) new terms. Like exhaustive search, this approach may explore a large part of the search space before finding any useful new attributes.

A more common approach is to combine a systematic search techique with pruning. GMDH, an algorithm that builds 'connectionist' networks, takes this approach. At a given layer i, GMDH considers as 'variables' (new terms) quadratic equations of *every* pair of 'variables' (terms) at layer $i - 1$; coefficients are determined by regression. GMDH then prunes any of these terms whose utility falls below a threshold. A term's utility is determined by calculating the mean-squared error of its predictive ability on an independent test set. Thesholds are also determined dynamically, using the mean-squared error of the evolving concept on the test set. The utility calculations and the dynamic utility threshold enable GMDH to conduct a kind of beam search for new terms.

The PET system uses a beam search to find arithmetic relations between sets of expressions; it is guided by semantic constraints and a set of heuristics that specify the desired characteristics of the resulting relations, for example that they have maximal coverage and minimal complexity. The ABACUS system is also a good example of this technique. ABACUS finds arithmetic equations to explain sets of data, much as BACON did. However, ABACUS uses arithmetic and physical constraints to identify irrelevant variables and to prune out meaningless combinations of relevant variables. It is much more sharply focussed

on its task than BACON was.

Another alternative, found in STAGGER, is to use the behavior of the learning algorithm to guide the search for new attributes. STAGGER creates new terms whenever it misclassifies a concept. A false positive calls for a more specific attribute; it is created by applying the $\wedge$ operator to two highly rated attributes. A false negative calls for a more general attribute; it is created by applying the $\vee$ operator to two highly rated attributes. Either type of misclassification also causes STAGGER to invert a poorly rated attribute, by applying the $\neg$ operator to it.

Finally, the search for new attributes can be guided by the structure of the learned, or partially learned, concept, as it is in FRINGE and CITRE. Both represent a concept as a decision tree; both construct a decision tree from the training instances. FRINGE constructs new attributes by combining the last pair of attributes on each path to a positive leaf node; this approach is sometimes called *constructive compilation* [Matheus & Rendell, 1989], because it simply records combinations already found in the data. CITRE is similar to FRINGE, except that it combines *every* pair of attributes on each path to a positive leaf node, and it *generalizes* the combinations that it encounters. CITRE generalizes an attribute by changing constants to variables. For example, the TicTacToe attribute (pos11=X $\wedge$ pos12=X) is transformed into (pos11=v $\wedge$ pos12=v) [Matheus & Rendell, 1989]. The value of the generalized attribute is either FALSE, or the binding that satisifies its definition. For example, the generalized TicTacToe attribute can take on the values X, O, and FALSE.

2.2 Partitioning

The partitioning approach to constructive induction can be characterized as creating new Boolean or nominal attributes from existing, or already known, numeric attributes. Its purpose is to make numeric information available to systems that were designed for Boolean or nominal data types. Partitioning is often added to a system after its design is essentially complete; it is an easy extension that doesn't change fundamental design decisions. The major distinction among different partitioning techniques is how a partition is constructed, i.e. how the numeric range is divided into discrete intervals.

STAGGER's technique for partitioning real-valued ranges [Schlimmer, 1987] is a good example. STAGGER's set of potential partition points is the set of attribute values occurring in its training instances. For each value, a two by two record stores the number of positive and negative examples with values greater than and less than this potential partition point. The *utility* of each partition point t is calculated by the following equation.

$$Utility(t) = \prod_{i=1}^{|classes|} \frac{p(class_i)}{1 - p(class_i)} \times \frac{p(class_i|\text{an attribute value} < t)}{p(class_i|\text{an attribute value} > t)}$$

The points whose utility values are locally maximal are used to partition the numeric range.

One advantage of STAGGER's approach is that it is simple. Utility values for each point can be calculated incrementally, as each new training instance is seen. It has two disadvantages. The first is that it requires a human to specify the number of partitions.

The second is that it uses the attribute values as potential partition points. An example illustrates how these two characteristics interact.

Assume that the user has granted the system one partition point, and that the values for the attribute tend to fall around either 2.0 or 5.0. The system might initially select either 2.0 or 5.0 as partition points; assume that it chooses 2.0. Subsequent training instances with an attribute value of 2.1 will be misclassified as belonging with the 5.0 partition. When STAGGER makes an error, it adjusts the partition, so that ultimately the partition point will get pushed out, perhaps to 2.4. However, no matter how far it gets pushed out, there is always the chance that some future instance will contain an attribute value that is just slightly beyond the partition point. Therefore, STAGGER's judgment about the *unseen* portions of a numeric range will be wrong 50% of the time, on average.

Quinlan suggests a slightly different approach for ID3 that avoids this problem [Quinlan, 1986]. If the values that attribute A takes on are sorted into order $v_1, v_2, \ldots, v_k$, each pair of values (v_i, v_{i+1}) suggests $\frac{v_i+v_{i+1}}{2}$ as a partition point. The use of the midpoint avoids the problem found in STAGGER. However, unlike STAGGER, ID3 does not specifically evaluate the utility of its partition points. Instead, new Boolean attributes are created to indicate whether attribute A's value falls above or below *each* partition point. If attribute A takes on many distinct values, ID3 will generate a large number of attributes, many of which may be unnecessary.

The technique that seems to be the most appropriate is found in a genetic algorithm called ARGOT [Shaefer, 1988]. ARGOT begins by dividing a numeric range into a set of partitions of equal size. The exact number is arbitrary and unimportant. Then, as training occurs, ARGOT keeps statistics on how often the attribute value falls within each range. Partitions that have 'too many'[1] hits are further subdivided, while partitions that have 'too few' hits are merged into adjacent partitions. Ultimately, the number of partitions and their ranges should reflect the distribution of the training instances, without any human intervention.

2.3 Clustering

The clustering approach to constructive induction creates attributes that represent *groups* of training instances. It is based on the observation that a set of instances cannot have a common classification unless they are somehow similar. The major differences among systems that perform clustering are their requirements for group membership, and their techniques for converting groups into new attributes.

Competitive learning, a technique for training hidden units in connectionist networks, does not contain an explicit similarity metric. Instead, it requires that each hidden unit in a layer of the network compete for the right to respond to each input pattern [Rumelhart & Zipser, 1986b]. The competition is organized as a winner-take-all event, so that only one unit can respond to each input pattern. The unit that wins has its weights redistributed among its input lines, strengthening its response to similar patterns. If the input patterns

[1]The exact definitions of 'too many' and 'too few' have not been published.

are highly structured, competitive learning converges upon a set of stable clusters. However, one disadvantage of competitive learning is that the network architecture determines the number of clusters into which each layer of the network will group patterns. Therefore one cannot say that competitive learning produces units that reflect the structure of the input alone; they also reflect the structure of the network.

Fu's technique for finding the intermediate nodes in a concept hierarchy determines dynamically how many clusters to create [Fu & Buchanan, 1985]. It begins with the assumption that each training instance is classified into one of a set of classes $\{c_1, c_2, \ldots, c_n\}$. An 'average' member of each class is created by averaging the attribute values of each instance in the class. The *dissimilarity* between two classes c_j and c_k can be calculated as the sum of the differences between the attribute values of their average members. For example, if the average members of classes c_j and c_k are described by attribute vectors $\vec{a_j}$ and $\vec{a_k}$, then the dissimilarity between the two classes is $\Sigma_{i=1}^{n}(a_{j_i} - a_{k_i})$. The technique is easily extended to a weighted sum of attributes, although the weights must be determined manually. Classes whose dissimilarity falls below some user-defined threshold can be grouped together into a higher-level class. New classes are named arbitrarily, and each new class name becomes a new attribute.

Fu's technique was designed to create intermediate nodes in a concept hierarchy in which the lowest and highest level nodes are known. It would not be effective in a domain in which there are few classes, nor would it be effective on problems that are not linearly separable.

UNIMEM [Lebowitz, 1987] applies similar techniques to training instances, instead of classes. The dissimilarity measure is similar to Fu's, except that it also handles nominal attributes; the dissimilarity of two nominal attributes is 1 if they differ, and 0 if they do not. UNIMEM also does not consider all combinations of instances, since it is designed for large numbers of training instances. Instead, as each training instance is received, a generalization hierarchy enables it to locate previous training instances that are similar. When the similarity of two training instances exceeds some threshold, the average of their attribute vectors becomes a new attribute. Coincidental matches will result in many useless new attributes, so UNIMEM also evaluates the attributes it creates by keeping track of whether or not they match future training instances strongly enough.

There are several advantages to the approach adopted in UNIMEM. It is incremental, it requires no user intervention beyond setting the initial parameters, and it creates what appear to be meaningful concept hierarchies. The only disadvantage is that the first few training instances have a disproportionate influence on what types of attributes UNIMEM will subsequently learn. UNIMEM builds and reorganizes its concept hierarchy incrementally, but it never completely reorganizes the hierarchy. The concept hierarchy determines which instances are checked for similarity, and similarity determines which attributes are formed. If the distribution of the initial training instances is not representative of the entire training set, it is possible that UNIMEM will not learn the most appropriate concept hierarchy.

Fisher's COBWEB [Fisher, 1987] is quite similar to UNIMEM in both its design and characteristics. The major difference between the two systems is their dissimilarity measures. COBWEB uses *category utility*, which was originally developed to predict the basic-level in

human classification hierarchies [Gluck & Corter, 1985]. Basic level categories are those categories (or *classes*, or *concepts*) within a hierarchy that are most natural for people to use. Category utility is applied to a partition of the examples (i.e. all of the classes in a hierarchy with a common parent), and is defined as:

$$CU(\{C_1, C_2, \ldots, C_n\}) = \frac{\Sigma_{k=1}^{n} P(C_k)\,[\Sigma_i \Sigma_j P(A_i = V_{ij}|C_k)^2 - \Sigma_i \Sigma_j P(A_i = V_{ij})^2]}{n}$$

In this definition, A_i represents attribute i, V_{ij} represents the j'th value of A_i, and C_k represents category k.

COBWEB's use of category utility allows one to argue that the hierarchies it produces are more valid psychologically than hierarchies produced by UNIMEM. However, like UNIMEM, COBWEB is affected by the order in which examples are presented. When COBWEB was run on Stepp's soybean data (a subset of Michalski's soybean data [Michalski & Chilausky, 1980]), it required three iterations through the examples before converging on the desired concept hierarchy [Fisher, 1987].

Rendell's PLS0 [Rendell, 1985] is a more complicated system that does not have this limitation. It is designed for domains like Checkers, where the structural attributes are quite similar to each other. While each square on a Checker board has its own characteristics, many of the important configurations in checkers are relatively independent of where they occur on the board. PLS0's task is to identify those configurations, wherever they may occur. It begins by projecting the training instances into lower dimensional subspaces. Each projection has an associated utility descriptor that partitions the subspace and associates with each partition the percentage of the projected instances that fall within the + class. Projections are then clustered, based on the similarity of their utility descriptors. Finally, the description of each cluster is generalized, using transformations like rotation, translation, and mirroring.

While PLS0 clearly performs clustering, clustering is just one element of what makes it powerful. The ability to project training instances into lower dimensions focusses its attention on a particular class of new attributes (e.g. attributes that describe pairwise interactions of checkers). Within that class, clustering is used to construct *every* useful attribute; utility descriptors are expected to discriminate among them. There is an underlying assumption that different attributes (e.g. piece adjacency, and pieces separated by one square) will have different correlations with winning states. While probably true, this assumption has apparently not been tested.

The most serious problem with PLS0 is its computational cost. The cost of creating every projection of dimensionality d is $O(\frac{n!}{d!(n-d)!})$ for a *single* training instance with n attributes. PLS0 lacks a mechanism for selecting just one value for d, but it clearly can not try them all since that requires creating every combination of attributes for each training instance; the cost would be prohibitive. In practice, PLS0 is usually used with $d = 2$, which reduces the projection cost to $O(n^2)$ for each training instance[2].

[2]Chris Matheus, personal communication.

2.4 Transformation

The transformation approach to constructive induction can be characterized as creating new attributes by applying predefined transformations to one or more training instances. It was suggested by Selfridge in the 1950's, for use in visual pattern recognition [Minsky, 1963]. It has received only a limited amount of attention since then, perhaps because the problem of selecting the 'right' transformations is as difficult as the problem of selecting an appropriate vocabulary.

The technique proposed by Selfridge consists of two types of transformations. *Basic transformations* change a pattern into another, different, pattern. Some examples include filling a polygonal region, removing points that are not on the boundary, and removing points that are not on a vertex. *Terminal operations* on a pattern yield a numeric value. One example is counting the number of points in a pattern. New attributes, or *properties*, are initially generated by randomly picking sequences of transformations that end with a terminal operation. Later, after some useful attributes are discovered, the set of attributes is improved by creating variations of existing useful attributes.[3]

One weakness of Selfridge's approach is that it is initially a random search of an infinite space. A second weakness is that the transformations he proposes are domain-dependent; choosing the 'right' transformations for a domain is itself a difficult problem.

STABB's least disjunction procedure [Utgoff, 1986] has neither of these weaknesses. The least disjunction procedure begins with sets of positive and negative training instances, and an attribute hierarchy; its goal is to create an attribute that applies only to the positive training instances. In the first step, it creates a description that is the disjunction of all of the positive training instances. In the second step, it generalizes each disjunct in every way that does not cover a negative instance. (The attribute hierarchy makes this possible.) In the third step, it throws away any disjuncts that are subsumed by other generalizations, and in the fourth it removes any subexpressions that are common to every generalization. The resulting generalized disjunction defines a new attribute that covers all of the positive instances and none of the negative instances.

The least disjunction procedure is domain-independent, efficient, and very effective. Its use of positive and negative training instances focuses its attention on attributes that will be immediately useful, and the generalization steps increase the likelihood that the new attributes will remain useful in the future. Its only limitations are that it requires an attribute hierarchy, and it does not apply to real-valued attributes.

2.5 Constraint Back-propagation

STABB's constraint back-propagation procedure is the most goal-oriented approach to constructive induction. It is performed by taking a goal concept and back-propagating its description through a sequence of operators [Utgoff, 1986]. The result is a set of ordered

[3]This search takes place in an infinite search space, so some sort of stopping criterion is necessary. Minsky does not indicate whether Selfridge suggested criteria for stopping the search.

pairs, (o_i, S_i); each o_i is an operator which, if applied to a state $s \epsilon S_i$, leads towards the goal. The descriptions of each set S_i form the basis of constructive induction. Each part of a description is checked against the current vocabulary. Expressions that match the definitions of existing attributes are replaced by those attributes. Expressions that do not match the definition of any existing attributes define new attributes.

The advantage of constraint back-propagation in constructive induction is that it produces exactly the set of new attributes needed to describe the states along the solution path. Its power comes from its access to a solution sequence, its ability to back-propagate the description of a set of states through the operators, and its access to an attribute hierarchy. However, this advantage is offset by one serious disadvantage. If operators are complex, it may not be obvious how to propagate descriptions of sets of states backwards through operators; not all operators have clearly defined counterparts that operate in the backwards direction. This characteristic limits the domains in which STABB can be applied.

2.6 The Initial Vocabulary

Most of the systems described in this section conduct a heuristic search through a space of possible vocabularies (the *vocabulary space*). In the vocabulary space, each state represents a set of terms; some of the terms may have been in the initial vocabulary, and some may have been created through constructive induction.

The initial vocabulary determines the starting state for the search, and it also affects the branching factor. Successor states are generated by adding new terms to the vocabulary at a given state.[4] Most algorithms create new terms by applying operators to one or more of the existing terms, so the size of the initial vocabulary determines the number of operands available for creating new terms. A good initial vocabulary produces a small space in which the initial state is not too far from states containing useful new terms. A poor initial vocabulary prevents even the best of the available constructive induction algorithms from finding useful new terms.

Constructive induction algorithms are powerful because they reduce the sensitivity of the inductive learning algorithm to the initial vocabulary. However, that sensitivity can not be completely eliminated, because the constructive induction algorithms are themselves sensitive to the initial vocabulary. Therefore, construction of a good initial vocabulary remains an important part of using inductive learning algorithms.

3 Knowledge-Based Feature Generation

This proposal began by observing that inductive learning algorithms would be easier to use if it were not so difficult to create a good initial vocabulary. The need to create a good initial vocabulary is even more of a problem when an inductive learning algorithm is used in conjunction with a problem-solving system. If the problem-solver's vocabulary is not

[4]Successor states can also be generated by deleting terms from the vocabulary.

appropriate for inductive learning, the developer must create, fine-tune, and integrate two separate vocabularies for a single task (e.g. WYL [Flann & Dietterich, 1986]). It would be more desirable if the problem-solver could use its knowledge about the task to automatically create the initial vocabulary for inductive learning.

This section describes a new approach, called knowledge-based feature generation [Callan, 1989], that enables a problem-solver to create an initial vocabulary for an inductive learning algorithm. It begins with a general model of the problem-solver, the induction algorithm, and their relationship, in order to define what information is available and how it can be used. The model is followed by a theory, knowledge-based feature generation, that states how the available information can be used to create an initial vocabulary for inductive learning. The theory is then illustrated with an example, and supported by a set of experimental results.

3.1 The Model

There are many different problem-solvers, with many different architectures; what they know, and in what detail they know it, varies. However, one very general and accepted view is that all problem-solvers conduct a search through some problem space [Newell & Simon, 1972]. While different problem-solvers may conduct the search quite differently, this paradigm provides a convenient framework for describing the problem-solver's knowledge and intentions. These are assumed to be as follows:

- **A1:** The problem-solver has declarative knowledge about its goal(s) and the operator(s) available for transforming search states.

- **A2:** The problem-solver intends to reach a goal state along some low-cost path.

A problem-solver that satisfies assumption A1 is said to have a declarative specification of a search problem, or a *search problem specification* for short. A search problem specification is a weak form of domain knowledge because it does not state how to select the search state to explore next. Therefore, the problem-solver is assumed to use the inductive learning algorithm for that purpose.

- **A3:** The inductive learning algorithm is intended to identify which of a pair of states is closer to a goal state.

The problem-solver can not assume that its own vocabulary for describing search states is appropriate for inductive learning. Instead, it must be prepared to translate its descriptions of search states into a vocabulary that is more appropriate for inductive learning. The extra overhead introduced by this translation process should be offset by improved learning performance.

- **A4:** The problem-solver must translate its descriptions of search states into a more appropriate vocabulary for inductive learning.

These four assumptions define a particular model of how a problem-solver and an inductive algorithm can be integrated. The model's assumptions are restricted to the interface between the problem-solver and the inductive learning algorithm, in order to maintain its generality. In particular, it makes no assumptions about how the problem-solver and the inductive algorithm carry out their tasks. It only defines what those tasks are, and what information is minimally available to carry them out.

The model is intended to be easy to satisfy, and three of the four assumptions are indeed easy to satisfy. However, assumption A4 requires the creation of a good vocabulary for inductive learning, given only the problem-solver's original vocabulary and a search problem specification. The problem of creating a good vocabulary from limited domain information has received little study, and is considered an open problem. The next section discusses a possible solution.

3.2 The Theory

The problem-solver must create a vocabulary that enables an inductive algorithm to recognize states that lead to a goal state, but its only source of information is the search problem specification. The description of the goal state provides a single Boolean description of any state, i.e. whether or not it is a goal state. This description does not distinguish among states that are not goal states, so it is an inadequate term with which to learn the desired concept. How, then, can the problem-solver create an appropriate vocabulary?

The desired concept is intended to enable the problem-solver to *hillclimb* towards a goal state. Hillclimbing depends upon the ability to accurately identify which of a set of states is closest to a goal state. Therefore one can argue that the desired concept incorporates, at least implicitly, a gradient whose height increases (or decreases) monotonically as one approaches a goal state. An appropriate vocabulary for learning is one that facilitates the construction of such a gradient.

The specification of the goal states can be treated as a step function,[5] since its value is 0 (False) for all states leading to a goal, and 1 (True) for the goal. This step function is a very gross approximation to the desired gradient, because it changes value at just one point. As a vocabulary for describing search states, it is impoverished, because it rarely reflects the problem-solver's progress towards a goal state. One approach to creating a better vocabulary is to decompose this single step function into a set of step functions, each of which can change value at a different point. The additional step functions make explicit the problem-solver's progress towards a goal state, and may facilitate the construction of better approximations to the desired gradient. If this process can be applied recursively, it may yield additional step functions whose values change at even more points. Ideally, the problem-solver's progress from state to state is reflected in a change to the value of at least one of the step functions.

[5]The phrase *step function* is defined differently by the mathematics and engineering communities. In mathematics, a function $f(x)$ over the interval $[a, b]$ is a step function iff f is defined for all x in the interval and f is not continuous. In engineering, f is further restricted to take on just two values. This proposal uses the broader definition from mathematics.

These observations suggest an approach to creating new terms that is founded upon the following hypotheses.

- **H1:** A representation for learning search control knowledge should explicitly represent the progress in problem-solving at each state.
- **H2:** Terms that express the progress in problem-solving at each state can be constructed automatically from a search problem specification.

Hypothesis H1 concerns the desirability of describing search states with step functions whose values change at many different points, while hypothesis H2 concerns the ability to create such functions out of a search problem specification.

Although hypothesis H1 is loosely supported by the previous discussion, and may seem obviously or intuitively true, it is a hypothesis that has been largely unexplored. None of the approaches to constructive induction described in Section 2 reflect this outlook. It is stated here precisely because it is an unusual approach to constructive induction. One goal of the research is to determine whether or not such representations are indeed desirable.

Even if such representations are desirable, it remains to be shown that they can be created automatically from a search specification. The approach, as suggested above, is to transform the description of the goal state into a set of step functions whose values can change at different points. The description of how to accomplish this transformation requires that one more assumption be made about the problem-solver.

- **A5:** The specification of the search problem is represented in first-order predicate calculus.

This assumption was not included in Section 3.1 because it is not necessary to the model. It is not, strictly speaking, necessary to the theory either. Section 4.3 discusses how it can be relaxed. It is included here largely for convenience.

Statements in first order predicate calculus consist of a possibly empty set of quantifiers and a Boolean expression over literals and/or variables. Because quantifiers may or may not be present, it makes sense to develop some transformations that operate only upon Boolean expressions, and other transformations that operate only upon quantifiers. A set of four transformations, developed as part of this research and defined below, is sufficient to cover each type of Boolean expression and each type of quantifier.

- **LE:** applies to logical expressions, i.e. Boolean expressions in which a logical predicate ($\wedge, \vee, \neg$) has the highest precedence.
- **AE:** applies to arithmetic expressions, i.e. Boolean expressions in which an arithmetic predicate ($=, \neq, <, \leq, >, \geq$) has the highest precedence.
- **UQ:** applies to universal quantifiers.
- **EQ:** applies to existential quantifiers.

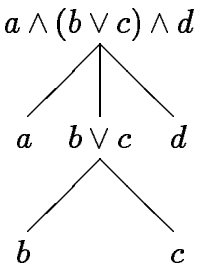

Figure 1: The subgoal hierarchy for the goal $a \wedge (b \vee c) \wedge d$.

LE, *AE*, and *UQ* transformations are presented in the following subsections. An *EQ* transformation remains to be developed as part of the thesis research; this subject is discussed in Section 4.2.

3.2.1 The *LE* Transformation

The *LE* transformation applies to Boolean expressions in which a logical predicate $(\wedge, \vee, \neg)$ has the highest precedence. Its purpose is to transform a Boolean step function into a set of Boolean step functions, each of which represents some component of the original step function. Each of the resulting step functions produces a value, T or F, when applied to a search state. Therefore each new step function defines a new term. This proposal usually refers to these terms as *LE* terms, because they are created by the *LE* transformation. However, they are also called subgoals, or subproblems [Callan, 1989], because they represent part of the original problem specification.

The *LE* transformation creates new statements by eliminating the predicates with the highest precedence. Its definition is:

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

Q_i is meant to stand for either $\forall v_i \epsilon S_i$ or $\exists v_i \epsilon S_i$, as in [Enderton, 1972]; b_j is any boolean statement. After the new Boolean statements are created, they are optimized by removing unnecessary quantifiers.

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

Normally the statement to which the *LE* transformation is first applied is the specification of a goal state (although see section 3.2.4). Therefore the resulting statements can be considered subgoals. The *LE* transformation can be applied to any subgoals that contain logical expressions, yielding subgoals of subgoals. If the transformation is applied wherever possible, the result is a subgoal hierarchy, with the goal at the root and atomic statements

at the leaves. For example, the subgoal hierarchy for $a \wedge (b \vee c) \wedge c$ is shown in figure 1. In general, a subgoal hierarchy contains between $c+1$ and $2c$ subgoals, where c is the number of logical predicates in the specification of the goal state.

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

The *LE* transformation, while new, produces expressions that are subsets of Gaschnig's *edge subgraphs* [Gaschnig, 1979], Guida and Somalvico's *semantic graphs* [Guida & Somalvico, 1979], Pearl's *relaxed models* [Pearl, 1984], and Mostow & Prieditis' *abstractions* [Mostow & Prieditis, 1989]. These other methods are more comprehensive than the *LE* transformation, because they can eliminate any combination of predicates, regardless of where they occur in the statement. However, the number of resulting expressions is combinatorially explosive, so all of these other methods currently require manual guidance.

There is also an important difference in how the resulting expressions are used. The expressions that result from the *LE* transformation are treated as step functions that describe search states. The expressions that result from these other methods are treated as simpler, or less constrained, versions of the original problem. The distances to goal states along various paths in the less constrained problem are used as estimates of the distances along similar paths in the original problem. This can be characterized as a *lookahead* approach, because it requires searching ahead for a solution to a subgoal. This additional search effort usually makes the cost of the heuristic more expensive than blind search [Valtorta, 1983], although recent work shows that it can be made acceptable if search states can be pruned or merged [Mostow & Prieditis, 1989]. This cost problem does not affect the methods reported in this proposal, because they are not based upon searching for solutions to subgoals.

3.2.2 The *AE* Transformation

The *AE* transformation applies to Boolean expressions in which an arithmetic predicate $(=, \neq, <, \leq, >, \geq)$ has the highest precedence. Its purpose is to transform a statement representing a Boolean step function into a single step function having multiple steps. The resulting step function produces a numeric value when applied to a search state. Therefore each new step function defines a new term. This proposal refers to these terms as *AE* terms, because they are created by the *AE* transformation.

The AE transformation creates a function that calculates the difference between the operands of the arithmetic predicate. If the original statement contains quantifiers, then the AE transformation calculates the *average* difference between the two operands. Its definition for the $\neq$ arithmetic predicate is:

$$\begin{aligned} AE\,(f_1(l_1) \neq f_2(l_2)) &\Rightarrow |f_1(l_1) - f_2(l_2)| \\ AE\,(Q_1, Q_2, f_1(v_1) \neq f_2(v_2)) &\Rightarrow \frac{\Sigma_{v_1 \epsilon S_1} \Sigma_{v_2 \epsilon S_2} |f_1(v_1) - f_2(v_2)|}{2} \end{aligned}$$

Q_i is meant to stand for either $\forall v_i \epsilon S_i$ or $\exists v_i \epsilon S_i$, as in [Enderton, 1972]; f_i is any arithmetic function. Similar definitions can be constructed for other arithmetic predicates ($=, <, \leq, >, \geq$) by substituting them for $\neq$ in the definitions above.

For example, the statement $\forall v_1 \epsilon S_1, \forall v_2 \epsilon S_2, f(v_1) \neq f(v_2)$ is transformed into:

$$\frac{\Sigma_{v_1 \epsilon S_1} \Sigma_{v_2 \epsilon S_2} |f(v_1) - f(v_2)|}{2}$$

.

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

The AE transformation could conceivably calculate other relationships between the two operands, for example total difference, minimum difference or maximum difference. The *average* difference is used throughout this proposal because it summarizes the relationships among a population of objects, rather than one relationship between a single pair of objects (as would the minimum or maximum difference). The *median* difference also summarizes relationships among a population of objects, and might be a more appropriate choice; this possibility will be investigated as part of the thesis research. If possible, just one of these measures (mean or median) will be used, in order to minimize the proliferation of features.

One problem with the AE transformation, as it is currently conceived, is its use of the absolute value to compute the difference between the operands. While useful for the $=$ and $\neq$ predicates, it destroys the ordering information necessary for the $<, \leq, >$, and $\geq$ predicates. The absolute value avoids the problems that arise when calculating the average difference for a goal like $\forall v_1, v_2 \epsilon \{4, 5\}, f(v_1) \neq f(v_2)$; without the absolute value, the average difference is always zero, because $[(f(5) - f(4)) + (f(4) - f(5)))/2$ is zero. The absolute value solves this problem; unfortunately it introduces another problem. One solution is to create two transformations, one that uses the absolute value and one that does not. However, this solution is guaranteed to create one useless new term. A better solution is to make the AE transformation sensitive to which predicate the expression contains. This will be pursued as part of the thesis research.

3.2.3 The *UQ* Transformation

The *UQ* transformation applies to universal quantifiers ($\forall$). Its purpose is to transform a Boolean step function into a single step function having multiple steps. The resulting step function produces a value between 0.0 and 1.0 when applied to a search state. Therefore each new step function defines a new term. This proposal refers to these terms as *UQ* terms, because they are created by the *UQ* transformation.

Subgoals with universal quantifiers require that a set of objects satisfy the Boolean expression. The *UQ* transformation produces a function that calculates the percentage of permutations of variable bindings satisfying the Boolean expression. Its definition is:

$$UQ\left((\forall v_i \epsilon S_i)^m p(v_1, \ldots, v_n)\right) \Rightarrow \frac{(\Sigma_{v_i \epsilon S_i})^m \begin{cases} 1 & \text{if } p(v_1, \ldots, v_n) \\ 0 & \text{otherwise} \end{cases}}{\Pi_{i=1}^{m} |S_i|}$$

p is meant to stand for any predicate.

For example, the statement $\forall v \epsilon S, p(v)$ is transformed into the function:

$$\frac{\Sigma_{v \epsilon S} \begin{cases} 1 & \text{if } p(v) \\ 0 & \text{otherwise} \end{cases}}{|S|}$$

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

UQ functions are a subset of the terms produced by Michalski's *counting arguments* rules [Michalski, 1983]. The counting arguments rules are applied to concept descriptions that contain quantified variables. Each rule counts the permutations of objects that satisfy some condition. However, the source of the conditions is not specified, suggesting that the source is a human. In the method proposed here, subgoals act as the source of the conditions, so human intervention is not necessary.

3.2.4 Use of Search Space Operators

The previous sections have assumed that all of the information about the goal state can be found in its description. However, that may not be the case in practice. It is usually desirable to move as many of the goal state requirements as possible into the preconditions of the search space operators, so that fewer search states are generated. The advantage of doing so is that the search speed may increase dramatically; the disadvantage is that information about the goal state becomes dispersed throughout the problem specification.

For example, suppose the problem-solver's task is to sort the contents of a file. Two of the possible specifications for the task are:

S1:

operator: exchange any two adjacent records.

goal: every pair of adjacent records is in the desired order.

S2:

operator: if two adjacent records are out of order, exchange them.

S1 is the more 'traditional' specification, because it clearly separates the problem-solver's intentions from the actions with which it searches. However, S1 allows the problem-solver to apply its operator to states that are already in sorted order; without further heuristics or control information, S1 results in a very inefficient sort. In contrast, specification S2 mixes the problem-solver's intentions with its actions so completely that the goal state does not need to be explicitly described; every terminal state is a goal state. S2 does not allow the problem-solver to apply its operator to records that are already ordered, so the sort is more efficient than the sort performed with specification S1.

The practice of moving goal state requirements into the preconditions of search space operators presents a problem for any method that considers only the goal state. This problem disappears if the transformations are also applied to the preconditions of each search space operator. The resulting expressions may or may not correspond to what a human observer would call a subgoal. However, treating them as subgoals does no harm; at worst, it allows the creation of a few more useless terms. This additional cost is offset by the guarantee that *all* of the information about the goal state will be found, whether it resides in the description of the goal state or in the preconditions of the operators.

3.2.5 Control Strategies

Although knowledge-based feature generation is presented above as an unordered set of transformations, it is most easily controlled in two phases. In the first phase, the LE transformation is applied repeatedly, yielding a set (or hierarchy, if preferred) of Boolean step functions (LE terms). In the second phase, the remaining transformations are applied to the Boolean step functions, yielding numeric step functions (numeric terms). This two phase process can be controlled with a variety of strategies, yielding algorithms that explore the space of vocabularies differently. The possibilities include both 'batch' and incremental control strategies.

A 'batch' control strategy is realistic because knowledge-based feature generation yields a relatively small number of terms. In the first phase, the LE transformation yields between $c+1$ and $2c$ Boolean step functions, where c is the number of logical predicates in the search problem specification. In the second phase, the remaining two transformations are applied to each Boolean step function. Although it is possible that both transformations will apply to each Boolean step function, it is also possible that neither of them will. Therefore, knowledge-based feature generation yields between $c+1$ and $6c$ new terms. These numbers

are small enough that one might choose to create and use all 6c terms to describe training instances.

An alternate approach is to augment the 'batch' control strategy with a pruning heuristic, to eliminate terms that are unlikely to increase learning performance. This approach, while appealing, is not currently possible because of the requirements imposed by the algorithms that evaluate representation quality. Saxena's method of rating sets of terms requires all of the training instances [Saxena, 1989a], while Mehra's method requires a small set of 'representative' training instances [Mehra, Rendell & Wah, 1989]. Neither of these requirements can be satisfied under the model proposed in Section 3.1. However, Saxena's recent work on an incremental algorithm with fewer requirements suggests that this approach will become both possible and practical in the near future.[6]

Incremental control strategies are easily created by dynamically deciding which of the possible new terms are actually used to describe search states. One could, for example, order the use of new terms by evaluation cost. The complexity of a new term depends upon the complexity of the subgoal from which it was generated. The complexity of a subgoal is determined by the number of quantified variables and the sizes of their ranges. It is therefore possible to determine the evaluation cost of a new term without actually evaluating it. One might choose to begin with the least-complex terms; if they do not result in accurate learning after some period of time, more complex terms could be considered.

The decision to use a 'batch' or incremental strategy for new terms depends largely upon the learning algorithm being used. Most connectionist algorithms (e.g. DNC [Ash, 1989]) could easily be modified to handle the addition and deletion of terms in training instances. However, STAGGER [Schlimmer & Granger, 1986] is the only existing symbolic learning algorithm that clearly has this property. The other symbolic learning algorithms must start over, forgetting what they have learned, whenever a term is added or deleted (e.g. ID3 [Quinlan, 1983]). Until this problem is overcome, the cost of repeatedly forgetting and relearning may make an incremental strategy more expensive than a 'batch' strategy for these algorithms.

3.3 An Example

The previous sections have described how to create new terms to describe search states. This section illustrates the approach on the Eight Queens problem. The Eight Queens problem is a constraint satisfaction problem that is commonly used to study the effectiveness of heuristics for search control [Nilsson, 1980; Pearl, 1984; Nadel, 1987]. It requires a problem-solver to find a placement of eight queens on a chess board that prevents any queen from attacking another queen. One of the 92 solutions is shown in Figure 2. The problem is difficult because the placement of any single queen affects the placement of every other queen. The Relaxation and Divide-and-Conquer heuristics that are often used in constraint satisfaction problems are ineffective on the Eight Queens problem.

[6] Sharad Saxena, personal communication.

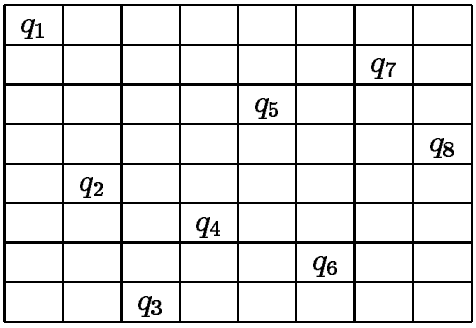

Figure 2: A solution to the Eight Queens problem.

There are many specifications for the Eight Queens problem. One such specification is given in Figure 3. No attempt has been made to 'tune' this specification for the purposes of creating new terms. Any equivalent specification for the Eight Queens problem could be used, although it might result in a different set of new terms. Several different specifications, appropriate for a constraint-satisfaction problem-solver, are given in [Nadel, 1987].

The example that follows is based upon the two phase control strategy described above. In the first phase, the *LE* transformation is used to generate a set of subgoals from the specification of the goal state. Each of these subgoals can be used directly as a new term. In the second phase, the *AE* and *UQ* transformations are applied to subgoals.

3.3.1 Phase 1: The *LE* Transformation

The first phase in generating new terms from the specification in Figure 3 is to decompose the specification of the goal state into a set of subgoal specifications. The result of applying the *LE* transformation to the specification in Figure 3 is the following set of subgoal specifications.

$$\forall q_i \epsilon Q, (\text{row}(q_i) \neq \text{NIL}) \tag{1}$$
$$\forall q_i \epsilon Q, (\text{column}(q_i) \neq \text{NIL}) \tag{2}$$
$$\begin{aligned}\forall q_i \epsilon Q, \neg\exists q_j \epsilon Q,\ & (q_i \neq q_j) \wedge \\ & ((\text{row}(q_i) = \text{row}(q_j)) \vee \\ & \ (\text{column}(q_i) = \text{column}(q_j)) \vee \\ & \ (\text{row}(q_i) + \text{column}(q_i) = \text{row}(q_j) + \text{column}(q_j)) \vee \\ & \ (\text{row}(q_i) - \text{column}(q_i) = \text{row}(q_j) - \text{column}(q_j)))\end{aligned} \tag{3}$$
$$\forall q_i \epsilon Q, \neg\exists q_j \epsilon Q, q_i \neq q_j \tag{4}$$
$$\begin{aligned}\forall q_i \epsilon Q, \neg\exists q_j \epsilon Q,\ & ((\text{row}(q_i) = \text{row}(q_j)) \vee \\ & \ (\text{column}(q_i) = \text{column}(q_j)) \vee \\ & \ (\text{row}(q_i) + \text{column}(q_i) = \text{row}(q_j) + \text{column}(q_j)) \vee \\ & \ (\text{row}(q_i) - \text{column}(q_i) = \text{row}(q_j) - \text{column}(q_j)))\end{aligned} \tag{5}$$

Initial State:	$Q = \{q_1, q_2, \ldots, q_8\}$ $R = \{1, 2, \ldots, 8\}$ $C = \{1, 2, \ldots, 8\}$ $\forall q \epsilon Q, (\text{row}(q) = \text{NIL}) \wedge (\text{column}(q) = \text{NIL})$		
Operators:	assign-row	condition:	$\exists q \epsilon Q, \exists r \epsilon R, (\text{row}(q) = \text{NIL})$
		action:	$\text{row}(q) := r$
	assign-column	condition:	$\exists q \epsilon Q, \exists c \epsilon C, (\text{column}(q) = \text{NIL})$
		action:	$\text{column}(q) := c$
Goal State:	$\forall q_i \epsilon Q, \quad (\text{row}(q_i) \neq \text{NIL}) \wedge (\text{column}(q_i) \neq \text{NIL}) \wedge$ $\neg \exists q_j \epsilon Q, \quad (q_i \neq q_j) \wedge$ $((\text{row}(q_i) = \text{row}(q_j)) \vee$ $(\text{column}(q_i) = \text{column}(q_j)) \vee$ $(\text{row}(q_i) + \text{column}(q_i) = \text{row}(q_j) + \text{column}(q_j)) \vee$ $(\text{row}(q_i) - \text{column}(q_i) = \text{row}(q_j) - \text{column}(q_j)))$		

Figure 3: A specification for the Eight Queens problem. The row and column of a location are summed to identify its diagonal in one direction; the difference identifies its diagonal in the other direction.

$$\forall q_i \epsilon Q, \neg \exists q_j \epsilon Q, (\text{row}(q_i) = \text{row}(q_j)) \quad (6)$$
$$\forall q_i \epsilon Q, \neg \exists q_j \epsilon Q, (\text{column}(q_i) = \text{column}(q_j)) \quad (7)$$
$$\forall q_i \epsilon Q, \neg \exists q_j \epsilon Q, (\text{row}(q_i) + \text{column}(q_i) = \text{row}(q_j) + \text{column}(q_j)) \quad (8)$$
$$\forall q_i \epsilon Q, \neg \exists q_j \epsilon Q, (\text{row}(q_i) - \text{column}(q_i) = \text{row}(q_j) - \text{column}(q_j)) \quad (9)$$

In general, the *LE* transformation generates between $c + 1$ and $2c$ subgoal specifications from a goal specification with c logical predicates. In this example, the specification of the goal state contained six logical predicates; nine subgoal specifications were generated from it. Each subgoal specification defines a Boolean term that describes a search state (an *LE* term).

3.3.2 Phase 2: The Remaining Transformations

The second phase in generating new terms from the specification in Figure 3 is to apply the remaining transformations to the various subgoals. The different types of transformations have been described in previous sections; each type should be applied, if possible, to each of the nine subgoals above. This section illustrates their application to subgoal 7, which states that no two queens should occupy the same column.

None of the transformations are designed to handle negated quantifiers. Therefore, the first step is to convert the negated existential quantifier in subgoal 7 to a universal quantifier,

using deMorgan's laws. The result is subgoal 7′.

$$\forall q_i, q_j \epsilon Q, (\text{column}(q_i) \neq \text{column}(q_j)) \tag{7′}$$

UQ terms are numeric measures of the degree to which the subgoal is satisfied. The value of the *UQ* term for subgoal 7′ is determined by evaluating the following expression.

$$\frac{\Sigma_{q_i \epsilon Q} \Sigma_{q_j \epsilon Q} \begin{cases} 1 & \text{if column}(q_i) \neq \text{column}(q_j) \\ 0 & \text{otherwise} \end{cases}}{|Q|^2}$$

This expression indicates the number of queens that are assigned to unique columns. It is created by converting the universal quantifiers into summation operators, and by encoding the truth of the condition as a numeric value.

AE terms represent the distance between two objects along some domain-dependent dimension. The value of the *AE* term for subgoal 7′ is determined by evaluating the following expression.

$$\frac{\Sigma_{q_i \epsilon Q} \Sigma_{q_j \epsilon Q} \begin{cases} 0 & \text{if } (\text{column}(q_i) = \text{NIL}) \vee (\text{column}(q_j) = \text{NIL}) \\ |\text{column}(q_i) - \text{column}(q_j)| & \text{otherwise} \end{cases}}{\Sigma_{q_i \epsilon Q} \Sigma_{q_j \epsilon Q} \begin{cases} 0 & \text{if } (\text{column}(q_i) = \text{NIL}) \vee (\text{column}(q_j) = \text{NIL}) \\ 1 & \text{otherwise} \end{cases}}$$

This expression indicates the average number of columns between any two queens that have been assigned to columns. It is created by converting the universal quantifiers into summation operators, by converting the inequality into a difference statement, and by dividing the result by the number of permutations of quantified variables for which columns have been assigned.

UQ terms can be created for each of the subgoals. *AE* terms can only be created for subgoals 1, 2, 6, 7, 8 and 9. They cannot be created for the other subgoals because those subgoals do not contain arithmetic predicates.

3.4 Preliminary Results

The theory presented in the preceding section has been implemented in a computer program called CINDI. The input to CINDI is a problem specification, expressed in first order predicate calculus. The output from the program is a set of specifications for the new terms.

Testing the quality of the resulting terms is difficult, particularly because there is no commonly accepted method. The approach adopted in this study was to measure how many of the different examples of a concept needed to be seen by a single linear threshold unit (LTU) [Nilsson, 1965] for it to reach a particular level of accuracy in identifying which of

two randomly selected search states is closer to a goal state. The LTU was chosen because it handles real-valued terms and because of its well-known limitations [Minsky & Papert, 1972]. The intent was to study the effect of the *representation*, rather than the power of a particular learning algorithm. More complex learning algorithms use multiple hyperplanes to classify examples, while an LTU uses just one. Therefore the vocabularies that an LTU prefers would also be useful to more complex learning algorithms.

LTUs are often associated with linearly separable examples, but linear separability is not strictly necessary. When the examples are linearly separable, the Perceptron learning rule is guaranteed to find a hyperplane that separates the positive examples from the negative examples, yielding 100% classification accuracy. When the examples are *not* linearly separable, the least-mean squares (LMS) learning rule is a better choice, because it tends to find hyperplanes that minimize the mean-squared error [Duda & Hart, 1973]; these hyperplanes provide good, but not perfect, classification. The experiments reported in the following sections were conducted with the LMS learning rule, because the examples were not linearly separable.

In each experiment, training and testing were alternated until the LTU reached a specified level of accuracy in classifying examples. Each training phase was performed on only a very small set of randomly chosen examples, but testing was performed on every possible example. This approach provides perfect measurements of how accurate a LTU is in classifying examples, but it is time-consuming. As a result, only two problems have been examined in any detail; they are discussed below.

3.4.1 The Eight Queens problem

As described in Section 3.3, the Eight Queens problem is commonly used to study the effectiveness of heuristics for search control. CINDI generated nine subgoals from the specification of the goal state. It generated another subgoal from the specification of the *assign-column* operator. Two of the ten subgoals were syntactically equivalent; one of two identical subgoals was manually discarded,[7] leaving nine subgoals (nine LE terms). All nine subgoals satisfied the requirements enabling UQ terms to be created, but only six satisfied the requirements enabling AE terms. The resulting terms were arbitrarily labeled Q1-Q24. Three of them are described above, in Section 3.3.

The performance of a single LTU using terms Q1-Q24 was compared with its performance using a hand-coded representation, labeled C1-C8. The hand-coded representation was intended to be the kind of representation that a problem-solver might use internally; it is one of the representations suggested in [Nadel, 1987]. Terms C1-C8 were numeric terms that indicated the columns to which each queen was assigned. C1 indicated the column for queen 1, C2 indicated the column for queen 2, and so on. No attempt was made to improve this representation, and no other representations were tried.

The learning curve in Figure 4 shows the number of different examples on which the LTU

[7]The code necessary to discard automatically all but one of a set of syntactically identical subgoals has not yet been implemented. However, it is not expected to pose any special difficulty.

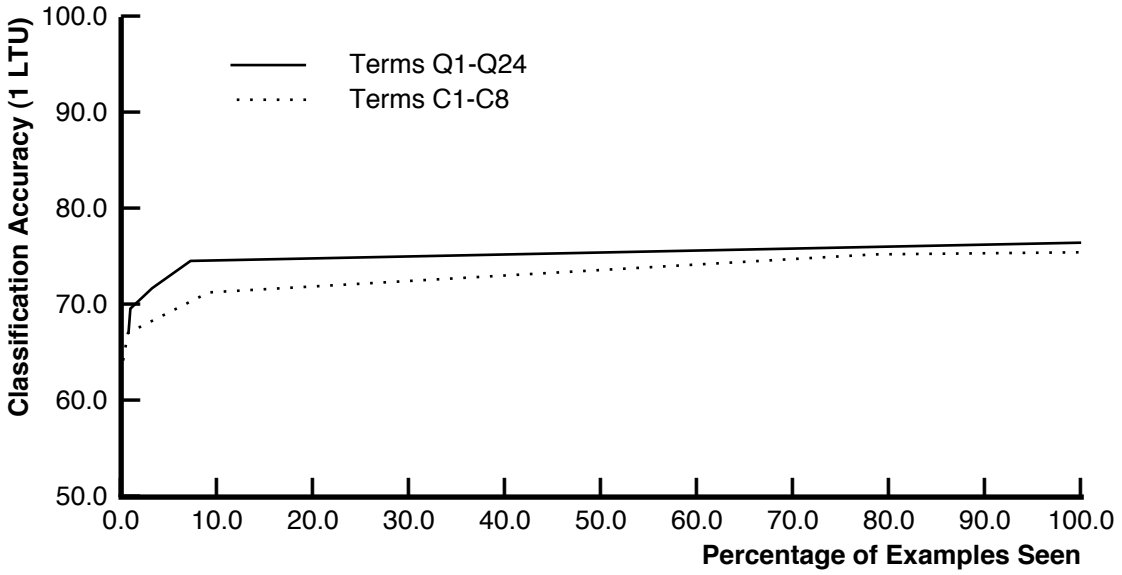

Figure 4: Learning curves for the 8 queens problem.

had to be trained for it to reach a given level of accuracy in identifying which of two randomly selected search states is closer to a goal state.[8] The set of terms generated automatically by CINDI clearly outperformed the hand-coded representation when the training set consisted of less than 10% of the different examples.

3.4.2 A Blocks-World problem

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

CINDI generated four subgoals from the specification of the goal state. It generated another six subgoals from the specification of the *stack-block* and *move-block-to-table* operators. Three of the ten subgoals were syntactically equivalent; two of three identical subgoals were manually discarded, leaving eight subgoals (eight LE terms). Four subgoals satisfied the requirements enabling UQ terms to be created, and none satisfied the requirements enabling AE terms. The resulting terms were arbitrarily labeled T1-T12.

For example, one of the subgoals created from the *stack-block* operator required that there be a block with an empty top. Two terms were created from that subgoal. Term T6

[8]Each measurement is the average of 100 experiments. The number 100 was chosen arbitrarily. Subsequent experiments, performed with the Recursive Least Squares training rule [Young, 1984] and conducted until values were known to within $\pm 1\%$ with 99% confidence, confirmed the relative utility of each representation.

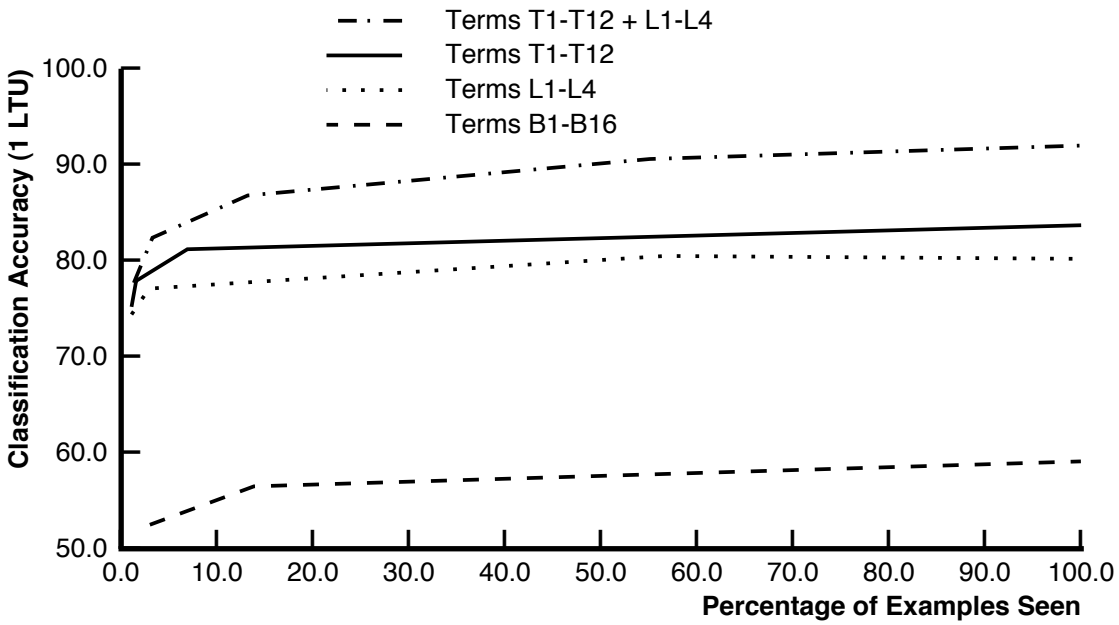

Figure 5: Learning curves for the 4 blocks problem.

was a *LE* term that indicated whether or not the subgoal was satisfied. Term T10 was a *UQ* term that indicated the percentage of blocks that had empty tops.

The performance of a single LTU using terms T1-T12 was compared with its performance using two hand-coded representations, labeled B1-B16 and L1-L4. The hand-coded representations were intended to be the kind of representations that a problem-solver might use internally. Terms B1-B16 were Boolean terms that indicated the positions of the blocks. B1 indicated whether A was on B, B2 indicated whether A was on C, and so on. Terms L1-L4 were numeric encodings of the positions of the blocks. The blocks were numbered 0 (for A) through 3 (for D), with 5 representing the table. Term L1 indicated block A's position, term L2 indicated block B's position, and so on. No attempt was made to improve any of the representations, and no other representations were tried.

The learning curve in Figure 5 shows the number of different examples on which the LTU had to be trained for it to reach a given level of accuracy in identifying which of two randomly selected search states is closer to a goal state.[9] The set of terms generated automatically by CINDI clearly outperformed both of the hand-coded representations. A combination of two representations performed best of all.

[9]Each measurement is the average of 100 experiments. The number 100 was chosen arbitrarily. Subsequent experiments, performed with the Recursive Least Squares training rule [Young, 1984] and conducted until values were known to within $\pm 1\%$ with 99% confidence, confirmed the relative utility of each representation *except* the B1-B16 representation. Further analysis revealed that the LMS rule would favor the B1-B16 representation if the experimental parameters were adjusted slightly.

4 Research Proposal

The preceding section describes a set of transformations that generate new terms for search problem specifications. However, this proposal should not be judged solely on the basis of those transformations, because it is likely that the details of the transformations will change as research progresses. Therefore the hypotheses that motivate the transformations should also be evaluated. These hypotheses are restated here.

- **H1:** A representation for learning search control knowledge should explicitly represent the progress in achieving subgoals at each state.
- **H2:** Terms that express the progress in achieving subgoals at each state can be constructed automatically from a search problem specification.

These hypotheses are more general than the transformations that they motivate, and are the foundations of this thesis proposal.

Hypotheses H1 and H2 are stated as universal truths, but it is more likely that they are only true for some class of problems. One goal of the research is to understand the conditions under which they hold. While the two hypotheses can be studied independently, they are more interesting when considered together. H1 provides the motivation for studying H2. If H1 is false, then the truth of H2 is of only theoretical interest. If H1 is true but H2 is false, then some other source of information must be found to guide constructive induction. If both H1 and H2 are true for a class of problems, then progress has been made towards solving the representation problem in machine learning.

This proposal describes preliminary work on testing hypotheses H1 and H2. The testing methodology has been to develop transformations that generate new terms from search problem specifications (validating H2), and then test their efficacy for inductive learning (validating H1). Preliminary results, as described in Section 3.4, suggest that both hypotheses are indeed true.

While the preliminary results are encouraging, they do not provide enough support for the hypotheses. Not all of the terms that the transformations generate are useful; these terms must either be eliminated, or be shown to fall outside of hypothesis H1. The thesis must address this problem before it can be considered complete.

Some of the constraints that hypotheses H1 and H2 place upon the research may not be obvious. The hypotheses both concern a specific kind of term; they make no claims about the utility of other kinds of terms in inductive learning, or the ability to generate them. As a result, there are many kinds of terms about which the thesis will make no statement. The hypotheses are also intended to be independent of any particular learning algorithm. Committing to a particular learning algorithm would make certain problems easier (e.g. recognizing useless terms), but would make the results less general. As a result, I will not to commit to a particular learning algorithm.

The next section explains how the research will be conducted. It is followed by Section 4.2, which explains the expected results of the thesis. Section 4.3 covers some possible extensions of the thesis topic.

4.1 Research Methodology

Machine learning is a field that has historically been empirical; researchers construct programs, experiment with them, and publish the results. This methodology sometimes makes it difficult to evaluate a learning algorithm, since even its author may not understand why it works (e.g. AM [Lenat & Brown, 1984]. As a result, researchers are trying to be more careful about understanding not only how the algorithm works, but *why* it works.

In keeping with this trend in machine learning, the thesis research will occur along two dimensions. First, the hypotheses H1 and H2 will be validated empirically. H2 will be validated with a set of algorithms that construct terms, and H1 will be validated by examining the effect of these terms on inductive learning algorithms. Second, I will attempt to show *why* H1 and H2 are true.

The focus of the thesis will be to show that the hypotheses H1 and H2 are correct. However, it is also expected to address a variety of ancillary problems concerning the efficiency and practicality of the approach. My underlying goal is to offer an alternative to manually constructing initial vocabularies for inductive learning. The algorithms that I propose will only be used if they are clear, clearly useful, and able to produce terms of reasonable quality.

The theory about how domain information can be used in constructive induction (section 3.2) has been implemented in a program called CINDI. The model of the interface between the problem-solver and a learning algorithm (section 3.1) has been implemented in a program called LCSP. It consists of a problem-solver, an interface to CINDI, and an interface for learning algorithms. Several learning algorithms are available, including LTU's [Nilsson, 1965], ID5R [Utgoff, 1989], the Perceptron Tree algorithm [Utgoff, 1988], the DNC algorithm [Ash, 1989], and the Taylor algorithm for learning polynomial evaluation functions[10]; each of these can be used with LCSP and CINDI. This suite of programs enables rapid testing of any change to the theory of knowledge based feature generation.

4.2 Expected Results

The expected results of the thesis can be broadly classified as validation of the hypotheses H1 and H2. The hypotheses themselves are not expected to change significantly.

Validation of the hypotheses will occur in three forms. First, an *EQ* transformation will be developed, to generate new terms from statements containing existential quantifiers. The completed set of transformations will validate hypothesis H2. Second, the resulting new terms will be shown to improve inductive learning in several domains, validating hypothesis H1. Third, an explanation will show when H1 and H2 can be relied upon, and why.

The thesis will also show that the resulting algorithms are of practical significance. In particular, it will show that the algorithms are efficient, that the cost of evaluating the resulting terms is not prohibitive, and that the results scale up to reasonably large domains.

The thesis will be considered complete when each of these tasks has been accomplished. The following sections explain each of them in more detail.

[10]Developed by Paul Utgoff.

4.2.1 The *EQ* Transformation

An *EQ* transformation is necessary to transform statements with an existential quantifier ($\exists$) into step functions having multiple steps. The resulting step function produces a value between 0.0 and 1.0 when applied to a search state. Therefore each new step function defines a new term. This proposal refers to these terms as *EQ* terms, because they are created by the *EQ* transformation. There are two approaches to developing an *EQ* transformation. Both will be explored as part of the thesis research.

The first approach is to apply the *UQ* transformation to both universal and existential quantifiers. The transformation is applicable, and would yield a meaningful result. The resulting function would indicate the percentage of permutations that satisfy the Boolean expression; states with higher values would leave the problem-solver more freedom in deciding how to satisfy the condition. The difficulty with this approach is its expense. The cost is reasonable for univeral quantifiers, because every permutation must be seen in order to know whether the subgoal is satisfied; counting them at the same time adds relatively little overhead. However, this same cost seems rather high for existential quantifiers; normally it is not necessary to see every permutation to know whether an existential quantifier is satisfied.

A second approach is to develop a cheaper transformation, i.e. one in which it is not necessary to examine every permutation. I am not sure what such a transformation might look like, but the possibility of cheaper EQ functions justifies some effort in this direction.

4.2.2 Generality: Other Domains

The theory about how domain knowledge can be used in constructive induction was developed by studying the Eight Queens problem. Therefore, while results for the Eight Queens problem are suggestive, they cannot be considered proof that the theory is effective on other problems. The theory can only be considered useful when it is shown to work in several different domains. Therefore, the theory will need to be applied to at least two, and possibly several, other domains.

Establishing that a set of new terms improves learning is surprisingly difficult to do in the general case. However, even if one is willing to settle for empirical results, the problem is far from straightforward. The first obstacle is finding an accepted benchmark against which to test. I am not aware of any accepted benchmark, with published results, for search algorithms. Even if accepted benchmarks were available, the representation of the problem would have a pronounced effect on the quality of the new terms that are generated. Ideally, one would use benchmark representations as well.

Since there are no accepted benchmarks, they will need to be developed as part of the research. First, a set of problems will be selected that are representative of classes of problems. Some of the candidates are discussed below. Second, the problems will be represented in predicate calculus. When possible, existing representations will be used. When representations do not exist, I will attempt to develop 'obvious' representations. Finally, the problem-solver will be run with, and without, knowledge-based feature generation, so that performance can be compared. The performance metrics will include total time to find a

solution, number of nodes expanded, and classification accuracy.

The theory will be applied to the placement of integrated circuits ('chips') on a printed circuit board. This problem has been chosen because it is known to be difficult, it will show the ability of the algorithms to scale to large problems, and because it is of interest to a large engineering community. If the theory is effective in this domain, the results will be of great practical significance.

The theory will also be applied to the game of Othello.[11] This problem was chosen because move selection in Othello is an intractable search problem, and because two performance programs,[12] a learning program [Fawcett, 1990], and a catalog of sophisticated features [Mitchell, 1984] are all available for comparison.

The theory will be applied to at least one other problem. This will provide a set of five domains in which to test the theory. Two of the domains are classic constraint satisfaction problems (Eight Queens, blocks world), one is a well-studied search problem (Othello), and one is a large constraint satisfaction problem of practical importance (chip layout).

4.2.3 Description of When it is Expected to Work

Knowledge-based feature generation may not be appropriate for all problem-solving situations. Instead, it may work in some domains and not work in others. A theory is most useful if it includes a way of predicting when it will work and when it will not. Therefore, one important goal for the thesis is to describe or characterize the class of domains in which this approach to creating terms is effective.

It is not yet clear how such a characterization will be produced. One approach is to apply the theory to several domains, as described above, and then generalize the results. For example, if it appears effective on constraint satisfaction problems but not games, then the two domains can be compared to see what they do and do not have in common. After a characterization is constructed, new problems can be tried that do or do not exhibit the necessary properties, in order to test the predictiveness of the characterization.

Another approach is to work forward, from the model of the environment that motivates the theory. Since the theory is designed for a particular model, it may be that the theory is effective for *any* problem that can be solved within that model. This would be quite a different result; it would argue that knowledge-based feature generation is effective for a particular class of problem-solving architectures. This result would be exciting, because it would show that the theory is fairly general. I do not expect to find such a result; my intuition is that the theory is most effective for certain classes of problems. However, this approach will also be explored.

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

[12] One written by Jeff Clouse, and one written by Paul Utgoff.

4.2.4 Explanation of Why it Works

Strictly speaking, the theory about how to use domain information in constructive induction can be useful without an understanding of *why* it improves learning performance. This type of result is actually quite common in constructive induction; the only justification for most methods of constructive induction is "It works [for a particular learning algorithm [on a particular problem]]." Unfortunately, such a justification is unsatisfying because it leaves many questions unanswered. An understanding of *why* it works might suggest ways of improving the method, or it might suggest different methods. Without such an understanding, other researchers have more difficulty evaluating the contribution of a particular method.

I would like to be able to explain *why* this particular approach to constructive induction improves learning performance. I am not sure how to make such an explanation, but I do have some ideas. One possibility is that search control knowledge requires functional concepts, like the number of placed queens, whereas most methods of constructive induction create structural concepts. Another possibility is to strengthen the argument that terms that describe the search space are more useful than terms that are formed from domain-independent (and search-space-independent) heuristics.

It may not be possible to explain how the theory improves learning without making more assumptions about the learning algorithm. I will try to avoid this approach. However, it might be possible to show that this type of term is particularly useful for algorithms that search the space of concepts in a particular manner. I would only take this approach if the other approaches fail.

4.2.5 Scale Up

Many ideas work well on small problems, but fail to work when they are scaled up for use on larger problems. One might therefore wonder whether the same is true of the method of knowledge-based feature generation. Knowledge-based feature generation could fail to scale up for three reasons. First, the cost of generating features might become prohibitive. Second, the cost of evaluating features might become prohibitive. Finally, the number of generated features might overload the learning algorithm, causing its performance to deteriorate. The first two of these problems are not expected to arise; the remainder of this Subsection explains why. The last problem, too many features, is not specifically addressed. However, the next Subsection (Subsection 4.2.6) addresses the problem of screening out features that are not useful for learning. If there are too many *useful* features, those that are more expensive to evaluate can be pruned away.

The method of knowledge-based feature generation consists of three parts. The first part involves the decomposition of the specification of the goal state into subgoal specifications. The heuristic procedure that performs the decomposition (the LE transformation) operates in $O(c)$ time and space, where c is the number of connectives in the specification of the goal state. This procedure scales linearly, so it will be effective on larger problems as well.

The second part involves the transformation of the subgoal specifications into specifications for arithmetic terms (the AE and UQ transformations). This proposal does not clearly

specify how each transformation takes place, but each of them has been implemented, and each takes either $O(1)$ or $O(q)$ time and space, where q is the number of quantifiers in the subgoal. Therefore the transformation of subgoal specifications into specifications for new arithmetic terms will also scale up to larger problems.

Finally, a value must be assigned to each new term. This step is important because a term's *cost* is the time necessary to assign it a value; its *benefit* is the time it saves through reduced search effort. If the cost outweighs the benefit, then the new term hinders the problem-solving effort [Berliner, 1984].

It is easy to calculate the cost of terms produced by knowledge-based feature generation. A term's cost depends upon the number of quantifiers in its definition, and the sizes of the sets to which they apply. Therefore, as mentioned in Section 3.2.5, it is possible to recognize the high cost of evaluating a term *before* the term is evaluated.[13] Unfortunately, there is no equally straightforward method of calculating a term's benefit, so one cannot dynamically decide whether a term is worth its cost. However, it is possible to generate terms incrementally, beginning with the cheapest, or to avoid terms who's cost exceeds some threshold.

4.2.6 Identifying Useful Terms

One unfortunate result of most methods of constructive induction, including the proposed method, is that the new terms are not equally useful for discriminating between positive and negative training instances. Terms that are useless for this purpose are undesirable because they needlessly increase the size of the space being searched by the learning algorithm. Therefore, it is important to ask of any method of constructive induction how the useful new terms can be distinguished from the useless new terms.

The task of recognizing the useful new terms is often left to the learning algorithm, because most learning algorithms are already designed to identify and use the most relevant terms. It is straightforward, if somewhat costly, to make use of the existing abilities of the learning algorithm. Experiments with CITRE show that eliminating the less useful terms improves performance dramatically. Concepts are found more quickly, are more accurate, and are stated more concisely with a pruned set of terms than with a larger set of terms [Matheus, 1990]. This result is supported by Saxena's experiments with ID3 [Saxena, 1989b]

My strategy for shielding the learning algorithm from useless terms involves several different components. The first step is to reduce the number of useless new terms that are generated. One weakness of the proposed method is the *LE* transformation that decomposes the description of the goal state into subgoals. Since it is a domain-independent heuristic, it sometimes produces irrelevant subgoals; each irrelevant subgoal can cause several useless new terms. Preliminary results suggest that some of the irrelevant subgoals can be recognized syntactically. If this is so, it would greatly reduce the number of useless terms that are generated. This hypothesis will be explored in more detail as part of the thesis research.

[13] The production of expensive terms is caused by the complexity of the problem, not its size. Very complex problems may produce expensive terms. Simple problems, of whatever size, will produce cheap terms.

A second approach to handling useless new terms is to measure each term's utility. Most learning algorithms perform a measurement of this type; it makes no sense to duplicate what the learning algorithm would do anyway. However, the constructive induction module has an additional source of information that the learning algorithm does not have: It knows what the term is intended to measure. It may therefore be able to monitor the term during learning to see how well it is doing its job. For example, if a term measures the average difference in an inequality, the standard deviation may indicate how useful the term is; if the deviation is large, the term may be less useful than if it is small. The thesis research will investigate the possibility of developing similar utility measures for each type of new term.

The utility of meaningful terms ultimately depends upon the learning algorithm. A term might be useful to one learning algorithm while being useless to another, perhaps because of differences in how they search the space of possible concepts. Meaningful terms that are not useful to the learning algorithm cannot be recognized without more knowledge of the learning algorithm; such knowledge is unavailable, given the model of the environment described in Section 3.1. Therefore, while efforts to identify and eliminate useless terms will be an important part of the thesis research, the learning algorithm cannot be completely shielded from them. The learning algorithm must also be capable of deciding for itself which of the terms it receives are actually useful to it.

4.3 Desired Results

The thesis topic, as defined in the previous sections, investigates the efficacy of using syntactic transformations of search problem specifications to create terms that describe how a search state satisfies subgoals. It is possible to expand the topic in several different directions, but it is not yet clear whether these are minor extensions or major problems. This section discusses one of these directions.

This section of the proposal is called Desired Results to emphasize that, while this direction will be explored as part of the thesis research, it may not yield results. The thesis is not dependent upon anything mentioned in this section. However, one can assume that the failure to achieve any of the desired results will shed light on the limitations of using syntactic transformations of search problem specifications to create terms.

4.3.1 Extension to Horn Clauses

The theory of knowledge-based feature generation has been presented as applying only to first order predicate calculus. However, section 3.2 states that this assumption is more for expository than theoretical reasons. Therefore, one possible extension to the thesis topic is an exploration of other languages to which it might apply.

Perhaps the most natural extension would be to Horn clause logic. Horn clause logic is more restricted than first order predicate logic, but in many ways they are quite similar. Specifically, both language contain existential quantifiers, universal quantifiers, arithmetic predicates, and logical predicates. Therefore, the *EQ*, *UQ*, *AE* and *LE* transformations should apply to problems specified in Horn clause logic.

The extension to Horn clause logic is desirable because Horn clause logic is the basis for the Prolog language. Prolog is used by many researchers in Machine Learning, particularly those working on Explanation-Based Learning (EBL) (e.g. [Kedar-Cabelli & McCarty, 1987]). The EBL community has largely ignored the work on constructive induction, but that may change. Many EBL researchers are now adding inductive learning to their systems, in order to handle incomplete domain theories. It seems only a matter of time until they also face the problem of incomplete representations. When they do, a method of constructive induction that uses both domain knowledge and Horn clause logic should be well-received.

5 Conclusion

There are many problem-solving systems that could benefit from inductive learning capabilities. On the surface it might appear that adding inductive learning to a problem-solving system is easy, since the only input to inductive learning algorithms is a set of training instances. However, the vocabulary used to describe the training instances is crucial to the success of inductive learning; if the vocabulary is appropriate, learning occurs quickly, otherwise it occurs more slowly, with lower accuracy, or not at all. Finding an appropriate initial vocabulary is problematic because there are no algorithms, or even good heuristics, for developing one. As a result, the process of adding inductive learning to a problem-solving system is more of an art than a science.

Constructive induction is intended to solve this problem, by providing a source of new terms with which to describe training instances. The existing techniques for constructive induction show the power of domain-independent heuristics, but the exploitation of easily available domain information has been largely unexplored. This proposal shows that domain knowledge can be used in constructive induction to create a new class of terms that the existing techniques cannot create. The results of applying this method to the Eight Queens and blocks-world problems are encouraging. They show that the resulting new terms can be used to express useful search control knowledge. However, a number of questions remain to be answered. The proposed research is necessary to answer these questions.

If it works, knowledge-based feature generation offers several advantages. One advantage is that new terms can be created before any training instances are available. None of the existing methods offer this ability. Another advantage is that the value of each new term can be computed as soon as the training instance is available; some methods, for example conceptual clustering, must wait to see a corpus of training instances before new term values can be computed. A third advantage is that the method creates a useful type of term that the other methods cannot intentionally create. The final advantage is that it is intended to complement, rather than replace, other constructive induction algorithms; other algorithms may offer capabilities (e.g. handling feedback from the learning algorithm) that knowledge-based feature generation does not offer.

Very little research has been done on how domain knowledge can be applied to constructive induction. This proposal advocates one way of addressing the problem. The preliminary results are encouraging, as are the potential benefits. If successful, it will make inductive

learning much easier to include in problem-solving systems.

Acknowledgements

This research was supported by a grant from the Digital Equipment Corporation, and by the Office of Naval Research through a University Research Initiative Program, under contract N00014-86-K-0764. I thank Paul Utgoff and Andy Barto for their comments on drafts of this proposal. I also thank Tom Fawcett and Carla Brodley for their comments on Section 2, and Chris Matheus for his comments on drafts of section 3.2.

Bibliography

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 Elithorn & Banerji (Eds.), *Artificial and Human Intelligence*. New York: Elsevier Science Publishers.

Buchanan, B. G., & Mitchell, T. M. (1978). Model-directed learning of production rules. In Waterman & Hayes-Roth (Eds.), *Pattern-directed inference systems*. New York: Academic Press.

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. G., London, B., Clarkson, K., & Dromey, G. (1982). Learning and inductive inference. In Cohen & 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 Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach*. San Mateo, CA: Morgan Kaufmann.

Duda, R. O., & Hart, P. E. (1973). *Pattern classification and scene analysis*. New York: Wiley & Sons.

Enderton, H.B. (1972). *A mathematical introduction to logic*. New York: Academic Press.

Falkenhainer, B. C., & Michalski, R. S. (1986). Integrating quantitative and qualitative discovery: The ABACUS system. *Machine Learning, 1*, 367-401.

Fawcett, T. E. (1990). *Feature discovery for inductive concept learning*, (Coins Technical Report 90-15), Amherst, MA: University of Massachusetts, Department of Computer and Information Science.

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

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

Fu, L. M., & Buchanan, B. G. (1985). Learning intermediate concepts in constructing a hierarchical knowledge base. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 659-666). Los Angeles, CA: Morgan Kaufmann.

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

Gluck, M. A., & Corter, J. E. (1985). Information, uncertainty, and the utility of categories. *Program of the Seventh Annual Conference of the Cognitive Science Society* (pp. 283-287). Irvine, CA: Lawrence Erlbaum Assoc.

Guida, G., & Somalvico, M. (1979). A method of computing heuristics in problem solving. *Information Sciences, 19*, 251-259.

Ivakhnenko, A. G. (1971). Polynomial theory of complex systems. *IEEE Transactions on Systems, Man, and Cybernetics, SMC-1*, 364-378.

Kedar-Cabelli, S.T., & McCarty, L.T. (1987). Explanation-based generalization as resolution theorem proving. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 383-389). Irvine, CA: Morgan Kaufmann.

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

Langley, P., Bradshaw, G., & Simon, H. (1983). Rediscovering Chemistry with the BACON system. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

Lebowitz, M. (1987). Experiments with incremental concept formation: UNIMEM. *Machine Learning, 2*, 103-138.

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

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

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.

Mehra, P., Rendell, L. A., & Wah, B. W. (1989). Principled constructive induction. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 651-656). Detroit, Michigan: Morgan Kaufmann.

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

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

Minsky, M. (1963). Steps towards artificial intelligence. In Feigenbaum & Feldman (Eds.), *Computers and thought.* New York: McGraw-Hill.

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

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., Utgoff, P. E., & Banerji, R. B. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In Michalski, Carbonell & Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* San Mateo, CA: Morgan Kaufmann.

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

Nadel, B. (1987). *Representation selection for constraint satisfaction problems: A case study using n-queens*, (Technical Report DCS-TR-208), New Brunswick, NJ: Rutgers University, Department of Computer Science.

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, NJ: Prentice-Hall.

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

Nilsson, N. J. (1980). *Principles of artificial intelligence.* Palo Alto, CA: Tioga.

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.

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

Porter, B. W., & Kibler, D. F. (1984). Learning operator transformations. *Proceedings of the Fourth National Conference on Artificial Intelligence* (pp. 278-282). Austin, TX: William Kaufmann.

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

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

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

Rumelhart, D. E., & McClelland, J. L. (1986a). *Parallel distributed processing.* Cambridge, MA: MIT Press.

Rumelhart, D. E., & Zipser, D. (1986b). Feature discovery by competitive learning. In Rumelhart & McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition.* Cambridge, MA: MIT Press.

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

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

Saxena, S. (1989b). *The input representation problem in machine learning*, (unpublished thesis proposal), Amherst, MA: University of Massachusetts, Computer and Information Science Department.

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.

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

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

Shaefer, C. (1988). The ARGOT strategy. *Proceedings of the First International Workshop on Change of Representation and Inductive Bias.* New York: Philips Laboratories.

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

Utgoff, P. E. (1988). Perceptron trees: A case study in hybrid concept representations. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 601-606). Saint Paul, MN: Morgan Kaufmann.

Utgoff, P. E. (1989). Incremental induction of decision trees. *Machine Learning, 4*, 161-186.

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

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