Feature Compilation

David J. Stracuzzi
Paul E. Utgoff

Technical Report 00-18
May 31, 2000

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Telephone: (413) 545-1985
Email:{stracudj|utgoff}@cs.umass.edu

1 Introduction **1**

2 Building in Knowledge **1**

3 Feature Specification Language **2**

4 Target Execution Language **3**

5 Compiler **4**

5.1 Enumeration 5

5.2 Mapping 6

6 TicTacToe **6**

6.1 Game Board Representation 6

6.2 Applying Features for Evaluation 6

6.3 Feature Specification Issues 8

6.4 Results 8

7 Qubic **9**

8 Related Work **10**

9 Discussion **10**

10 Summary **11**

Abstract

A fundamental issue in state evaluation is the production of features that enable state assessment. The features must identify the intrinsic properties of states so that they may be evaluated accordingly. Methods for feature design and construction currently range from automatic induction methods to simple "hand coding" of information into performance programs. Feature compilation allows experts to provide domain knowledge to a program without explicitly defining the representation. High level quantified statements may be compiled and expanded into an efficient representation. The details and benefits of feature compilation are presented along with a sample application.

1 Introduction

When building a performance program that learns to evaluate states, a fundamental problem is to design and construct a set of features that will facilitate the state assessment process. A good set of features allows performance programs to navigate efficiently through search and hypothesis spaces that would otherwise be intractable. The features must discriminate states that share one or more intrinsic properties from those that do not. By making available a set of such potentially overlapping features, it becomes possible to associate a value with the presence of each property, or combination of properties so that a state can be evaluated accurately.

Many methods exist for inferring values to associate with states, such as temporal difference learning. There are also many methods for adjusting the parameters of functional form over the features, such as gradient descent on the error function with respect to the adjustable parameters. Of great interest is how to design and construct the features on which these depend. A variety of approaches have been proposed, and many work well for certain kinds of problems. Feature compilation is presented as a method for adding expert domain knowledge into performance programs. We consider here only domains represented by Boolean attributes, although a feature compiler could be made to handle other representations.

2 Building in Knowledge

Creating a set of features is a crucial step that program designers take in providing a performance program with knowledge of the test domain. For simple domains, automatic feature construction methods may provide the necessary information. However, the skill level of a performance program is often improved when it is supplied with expert features. For example, the best version of Tesauro's (1992) program TD-Gammon requires hand-coded features. While automatic construction methods are very important and need to be pursued aggressively, it is also critical that human experts be able to specify features easily for use by the learning program. Indeed, we expect both approaches to be essential processes in building high performance decision making systems.

Our focus is on the latter problem of how to specify features. We would like a high level language for specification, and a low level language for execution, which suggests immediately a compilation process. The specification language should match the level at which a person may think and work with the domain. The low level representation should require as little time and space as possible. The feature designer should be able to write at a comfortable level, and then run the compiler to convert the source statements into efficient low level structures usable by a target performance program.

In general, useful domain features may be produced without necessarily producing a complete set

with respect to perfect state evaluation. Statements written in a high level language may describe whatever knowledge is available for a domain. Although the features produced will be incomplete with respect to perfect knowledge, they nevertheless enable a more accurate evaluation function. A partial set of features can form the foundation for learning or provide guidance to a search algorithm.

3 Feature Specification Language

For a game such as TicTacToe, there is no need to rediscover what is already known to the expert. Why should the program lose countless games in order to stumble onto the notion of three-in-a-row when we could simply tell it at the outset? It would be convenient to be able to make a statement such as, "For all triples of cells that form a row, each of the three cells must contain an X". From this, a feature that evaluates to 1 when X has three in a row and 0 otherwise would be constructed.

Notice the notion of quantification of variables over domains of objects. Happily, the domains are finite and in this case small. In general, a compiler can perform the enumerations indicated by the quantifications, and produce a set of features that recognize the important states whenever the variable domains are sufficiently small. This is the essence of the approach, and the details are provided in Section 5.

There are several important requirements of the feature specification language. The language must be at a sufficiently high level for feature specifications to be authored quickly and easily. Some domains may require that specifications be written in terms of previous specifications, so the language must also allow for structure. Quantification of variables is another key aspect of the feature specification language, as the compiler's actions are based on these variables. Each variable must have a defined finite domain. Related to the use of quantified variables is the issue of variable constraints, which are required in order to restrict the scope of a feature specification.

The language must also be able to differentiate situations in which variable ordering is important from those in which it is not. For example, when listing the three-in-a-row combinations described above, the order of the cells is not important and all permutations of the same three cells should be considered as one. There must also be a method for defining primitives on which the specifications will be built. If the domain attributes are represented by a set of Boolean variables, then the feature designer must be able to reference them through the language.

First order logic presents a good basis for a feature specification language. The language level is sufficiently high for describing the details of a domain without becoming distracted by implementation issues. Domain structure can be captured in first order logic through the layering of feature specifications. The use of variables in specifications is also built into the language. These three aspects form the heart of the feature specification language, although they are by no means complete.

Several notions must be added to first order logic in order to produce an adequate feature specification language. Mathematical operations in first order logic are based on a somewhat complex set of recursive definitions. Therefore, the evaluation of mathematical expressions, equations and inequalities is included in the language to provide an easy mechanism for constraining variables. Statements that describe the relationship between variables are also added to the language. This allows the importance of variable ordering to be described. Constructs are added to the language for defining constants and the domains of quantified variables. Finally, *primitives* are added to the language through special statements in the feature specifications. Primitives are defined in terms of the Boolean input variables, allowing the designer to use the input variables as a basis for feature

```
Define: X = 0;
Define: O = 1;
Define: E = 2;

Define: p1 = 0..8;
Define: p2 = 0..8;
Define: p3 = 0..8;

Set: p2->p1;
Set: p3->p2;
Sentence: XWON(p1, p2, p3) if |( &( p3=p2+1, p2=p1+1, p1%3=0, S(p1, X), S(p2, X), S(p3, X)),
                                &( p3=p2+3, p2=p1+3, S(p1, X), S(p2, X), S(p3, X)),
                                &( p3=p2+4, p2=p1+4, S(p1, X), S(p2, X), S(p3, X)),
                                &( p3=p2+2, p2=p1+2, p1=2, S(p1, X), S(p2, X), S(p3, X)));
Sentence: HASROW(X) if EXISTS(p1,p2,p3)[ XWON(p1,p2,p3) ];
```

Figure 1. Complete specification of "Does X have three in a row?"

specifications.

Using this logic-based language, a feature specification corresponds to defining a *sentence* over a set of quantified variables. A sentence is some combination of conjunctions, disjunctions, negations and quantifications of primitives, other sentences and variable constraints. Figure 1 shows the specification for "X has three in a row". First, constants are defined for X, O and empty. Next, the domains of the three variables used to represent cells of the board are quantified as zero through eight, so that each of the nine board cells may be referenced by each variable. The two following statements identify the ordering of the three variables as unimportant. All six permutations of three cells will be considered as one.

The sentence XWON, which describes the individual three-in-a-row combinations for X, is then defined as a disjunction over four conjunctions, where each conjunct defines the horizontal, vertical or diagonal rows of the board. The primitives such as $S(p1, X)$ reference the Boolean variables that describe the state of a TicTacToe board. For example, if $p1 = 4$ then $S(p1, X)$ would evaluate to `true` when square four of the board is occupied by X. Finally, HASROW is defined such that it will be true when any combination of variables is found to satisfy XWON.

4 Target Execution Language

For feature compilation to be a viable method for state evaluation, the resulting features must be both efficiently evaluated and compact. Compactness is particularly important for larger domains in which only a small part of the domain information may be explicitly represented. Many domains of interest possess some type of structure in which concepts of increasing complexity are built up from other, more simple concepts. Recognizing and using substructures is an integral step in creating an efficient representation. By allowing the feature language to reuse previously constructed information, both representation space and evaluation time can be saved. In order to take advantage of domain structure, the execution language must be able to combine concepts in a layered manner.

One possible execution language that meets the above requirements is an AND/OR graph. A *feature* in this execution language corresponds to a single node in the graph. Evaluating a feature with respect to a set of Boolean variables corresponds to evaluating the node in the graph based on the same Boolean inputs. Evaluation is performed in a bottom up manner, from the Boolean graph inputs to the output node of the desired feature. Thus, a feature is `true` with respect to a set of Boolean variables when the corresponding subgraph evaluates to `true`.

AND/OR graphs are equivalent to a set of first order logic statements in which the domain elements of each variable have all been enumerated. This allows for a tight mapping between the feature specification language and the execution language, which is integral to this approach. The equivalence between the two languages also makes the compiled representation intelligible to humans and verifiable with respect to a given source. This is an important property of AND/OR graphs that many other potential execution languages do not share.

An AND/OR graph addresses the requirements on structure and efficiency for an execution language. Compactness and evaluation efficiency are achieved through subgraph reuse. A feature of a particular problem will appear as a distinct subgraph. Since multiple appearances of a feature can be represented by multiple references to the corresponding subgraph, there is no duplication of information. A subgraph need only be evaluated once per graph evaluation, which saves time. Terminating evaluation at a node before all subgraphs have been evaluated provides an additional time savings when the graph is evaluated serially. If a child of any AND node evaluates to `false`, then the evaluations of the remaining children become irrelevant. Similarly, once any child of an OR node evaluates to `true`, the result of the entire node is known.

Note that while quantification is an important part of the specification language, it is not included in the execution language. Although quantification presents a powerful method for representing large or complex concepts, it also implies the use of search. Compiling into the AND/OR graph representation removes the search step from the evaluation process by finding and enumerating the specified states. These states are then known in advance and may be recognized without performing costly search during evaluation. By restricting the execution language to only AND, OR and NOT combinations, the input variables can be tested quickly for conditions of interest.

5 Compiler

Using domain descriptions written in the feature specification language as input, a logic compiler can construct a graph of features that describe the domain. Particularly important in this process is that the compiler take full advantage of the structure in a domain as explicitly defined in the feature specifications. Ideally, the compiler should also look for structure not explicitly defined in specification so that the resulting feature graph is as small as possible.

Two requirements must be met before a compiler can expand domain descriptions into a feature graph. First, each quantified variable used in the feature specification statements requires a fixed and finite domain. This allows the compiler to cycle over the domain of each variable and enumerate all valid instances of a sentence. Second, the compiler must be able to reference the Boolean input variables. The Boolean variables form the foundation of the feature graph and must be included in the execution representation. Once these conditions have been met, the quantified statements written in the feature specification language can be compiled into features.

In order for the compiler to reference the Boolean input variables and add them as inputs to the feature graph, the primitives defined in the feature specification must be expanded. Each primitive defined and used in the feature specification represents a reference to the Boolean input variables, so the compiler must generate the input nodes first in order to build the feature graph. The primitives become unique input nodes in the graph which evaluate to `true` exactly when the associated Boolean variables are `true`.

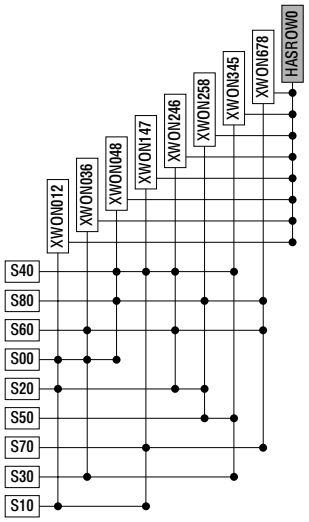

Figure 2: Circuit diagram for the feature "Does X have three in a row?". Inputs to the circuit appear along the left side and constructed features appear on the top. Shaded boxes represent OR nodes, clear boxes represent AND nodes, and dots show where connections are made.

5.1 Enumeration

One way to view the feature compilation process is as macro expansion. This is accomplished by removing the quantified variables through the enumeration of each variable's domain. Each variable used in a feature specification statement is enumerated over its entire domain, producing all possible instances of the sentence. This process is exponential in the number of quantified variables, so feature compilation can only be applied when variable domains are small enough to allow enumeration. However, most meaningful conditions of a domain will be highly constrained, which reduces the number of valid enumerations and keeps the number of actual features produced small.

Enumerating a feature specification and compiling the results into a feature graph requires several steps. The compiler must first read the input specification and translate it into a parse tree. The parse tree represents all of the constraints and sentence references from the specification and structures them according to the feature specification. Specifically, the layers of AND, OR and NOT combinations are now explicitly represented.

The compiler next enumerates the domains of all variables used in the feature specification and evaluates the parse tree in a bottom up manner according to each variable combination. Variable constraints are tested first. If satisfied, the necessary AND, OR, and NOT nodes are created to represent the specific logical statement represented by the parse tree with respect to the current values of the variables.

Consider again the specification for HASROW shown in Figure 1. The compiler would first expand the definition of XWON into a three layer parse tree. The first layer is an OR node attached to the four AND nodes of the second layer, and the third layer refers to the primitives and constraints for each of the AND nodes. If the variables are assigned as $p1 = 0$, $p2 = 1$ and $p3 = 2$, then only the constraints specifying a horizontal row of the board will be satisfied. An AND node is then created that tests whether cells zero, one and two of the board are occupied by X. Thus, the feature XWON012 has been created as shown in Figure 2. The eight remaining instances of XWON are similarly created as the quantified variables are enumerated over their respective domains. The

definition of HASROW can now be expanded into a disjunction over all eight instances of XWON as shown in Figure 2.

5.2 Mapping

One of the most important aspects of the feature compiler is the process of mapping specification sentences into the feature graph. A feature generated by the compiler represents one instance of a quantified feature specification, and a particular specification may expand into numerous features. Each feature is represented by a subgraph in the execution language.

During the process of building a feature, the compiler may also create several sub-features as a result of nested logic in a specification. The sub-features appear as subgraphs in the feature graph. Thus, in addition to features in the graph building on each other, they may also share a common sub-feature. In this way, the compiler attempts to find and take advantage of structure that is not explicitly defined in the feature specifications. This strict policy of reuse is the source of the compiled graph's efficiency.

In order to achieve this feature reuse policy, the compiler must track all of the nodes generated in the graph. As each specification statement is enumerated, the resulting nodes are assigned a unique tag based on their meaning. Then, during subsequent enumerations, if a node is generated with the same tag (and therefore meaning) as an existing node, the existing node is reused rather than recreated. The resulting graph is as dense as the input feature specification allows. A highly structured set of specifications will produce a graph with many layers that reuses many nodes.

6 TicTacToe

TicTacToe presents a simple domain on which to demonstrate the use of a feature compiler. The knowledge required for perfect play is relatively simple and easily expressed. The features generated by the specifications described below can be used in conjunction with a one-ply search in order to achieve perfect play for both X and O. Although TicTacToe is small enough to search completely, many other domains do not share this property. For these larger domains, eliminating many plies of search through fast evaluation provides a significant performance advantage.

6.1 Game Board Representation

The first step in defining a set of feature specifications for TicTacToe is to choose a representation for the game board. One way to describe a TicTacToe board using Boolean variables is to assign three variables to each cell of the board. Hence, one of the three variables is only true when X occupies its square, with another variable for each O and empty.

The triple for each of the nine board cells can then be referenced by an index between zero and eight by numbering the cells in row major order. Given this basic information, the domains of the quantified variables used for indexing the board can be specified and the compiler can generate nodes for each of the Boolean variables (primitives) used to describe the board.

6.2 Applying Features for Evaluation

After the board representation has been chosen and the necessary primitives have been defined, the domain features can be specified. In general, a good set of features for a domain will separate states according to the properties they share. For TicTacToe, a good set of features should discriminate whether a position is a win (value of +1), loss (-1) or draw (0) for the player on move.

When using a one-ply search, the feature specifications may be simplified by only considering

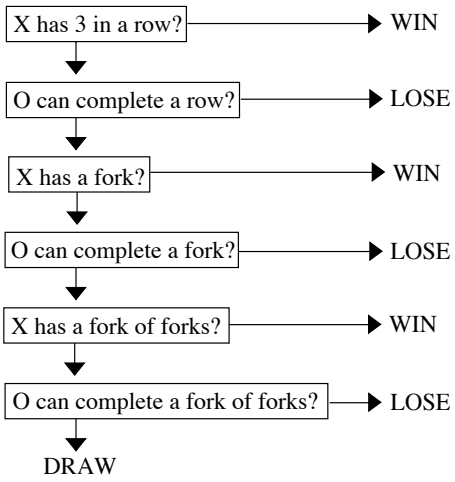

Figure 3. Decision list for evaluating TicTacToe positions.

board positions from the perspective of X when X has just moved. The reduced set of features can then be used to evaluate a game board from the perspective of O simply by complementing the position and negating the resulting evaluation. All cells owned by X become O, all cells owned by O become X as in negamax search.

Using the player-on-move view, six features, evaluated in a specific order, are sufficient for evaluating TicTacToe positions. A decision list for evaluating TicTacToe positions appears in Figure 3. If a position is evaluated using the perspective described above, then the first feature that must be tested is "X has three in a row". This corresponds to a terminal state of the game and, when true, causes the other five features to become irrelevant. The next feature that must be evaluated is "O can complete a row". Since it is now O's turn to move, this corresponds to a loss for X. The third feature tested is "X has a fork", which tests whether X has created a position where it now has two separate winning moves and therefore a forced win. Note that O cannot win in its next turn because the second feature has already evaluated to `false`.

Next, the feature that tests "O can complete a fork" is evaluated. As before, it is now O's turn, so if O can complete a fork, then O can force a loss for X. Continuing in this pattern, the fifth and sixth features in the decision list must check if "X has a fork of forks" and "O can complete a fork of forks" respectively. A fork of forks means that a player has reached a position in which it can build a fork regardless of the opponent's next move. Thus, this is a winning condition for whatever player achieves this position. Finally, if none of the above features has evaluated to `true`, then the current game position must be a draw.

There is an important relationship between the features described above and game tree search. The condition "X has three in a row" represents a terminal state of the game and can be recognized as a win with no additional search from the current state. However, "O can complete a row" is not a terminal state and can only be recognized as a loss for X with one ply of search. The condition "X has a fork" would then require two ply of additional search to discover that it is a win for X. O may make any move followed by a move for X that completes some row.

This pattern continues until the final condition of "O can complete a fork of forks", which represents a full five ply game tree search. Each feature in the decision list for TicTacToe represents a game tree search to a depth of one ply greater than the previous feature. Combining the above TicTacToe game features with a one ply search thus performs the equivalent of a six ply game tree

AND	OR	NOT	Input	Total
741	195	101	27	1064

Table 1. Number of nodes used in the complete TicTacToe feature graph.

X has a row	O can create a row	X has a fork	O can create a fork	X has a fork of forks	O can create a fork of forks
2	3	4	6	6	4

Table 2: Number of layers, excluding the Boolean inputs, for each feature used in the TicTacToe decision list.

search.

6.3 Feature Specification Issues

There are important issues to consider when designing a set of feature specifications for TicTacToe. The key to creating a small feature graph is to capture domain structure in the specifications. Each new specification preferably will build upon previous specifications, allowing the reuse of as many existing nodes in the compiled graph as possible. When new features cannot be built directly from existing features, they may still share common sub-features.

TicTacToe has just two important sub-features that are shared by most of the six decision list features. The first concerns row completion. The features for "O can complete a row", "X has a fork", "O can complete a fork" and "X has a fork of forks" all depend at least partially on whether one player occupies two of three cells in a row and the third is blank. The second important sub-feature tests whether a player occupies one cell in a row and the remaining two are blank. "O can complete a fork" and "X has a fork of forks" both rely heavily on this sub-feature. Four of the six decision list features are based on two simple sub-features, which leads to a compact decision graph.

A second issue that must be addressed when designing the feature specifications involves how the features should be enumerated. Consider the row completion sub-feature. The important aspect of row completion is the location of the empty cell. Without knowing the location of the empty cell, it is difficult to test whether a row can be completed in two separate locations, as in a fork.

A similar issue arises with the definition of "O can complete a fork of forks". Other features were enumerated according to the location of the important blank cells in order to be consistent with the sub-features. However, this creates an unnecessarily large and complex feature graph for "O can complete a fork of forks" due to the number of blank cells that must be tracked. Recall that this feature corresponds to a five ply search, meaning that there are five blank cells that must be filled under certain conditions. A more simple approach in this case is to enumerate according to the cells that have already been filled. A smaller feature graph is produced even though previously defined features are not used.

6.4 Results

The six features described above were compiled into a feature graph and applied to all 5477 TicTacToe game positions. Compilation on a Pentium 166 requires less than seven seconds to create a feature graph with a total of 1064 nodes. Table 1 shows a count of each type of node created by the compiler. Using the decision list described earlier, the six features correctly evaluated each of

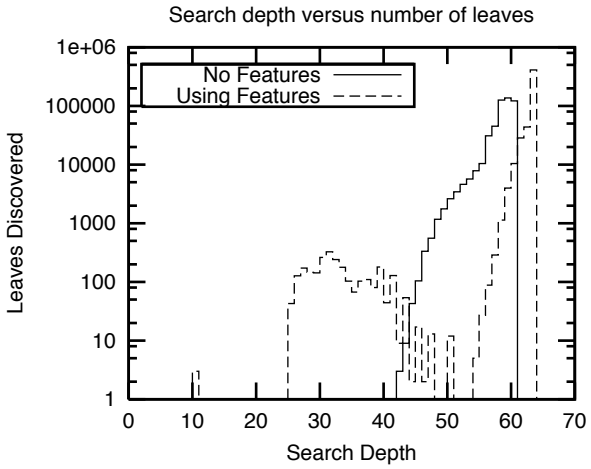

Figure 4: Comparison of Qubic search over 500,000 leaves with and without the compiled features.

the 5477 TicTacToe game positions as either a win, loss or draw for X.

Table 2 shows the number of layers for each of the decision list features. Note that each feature does not necessarily build on a previous feature. Features may build up directly from other features, share common subgraphs, or be completely independent of other features. The complete graph has a total of six layers excluding the Boolean inputs. Although a graph could be compressed into a two layer feed-forward network, this would increase the number of nodes in many cases.

7 Qubic

Qubic is a three dimensional version of TicTacToe played on a 4x4x4 board. The rules are unchanged except that a win now requires a player to occupy four squares in one of the 76 rows. Scaling the TicTacToe features up to Qubic requires no drastic changes, although the specifications become somewhat more complex in order to handle the extra dimension. While Qubic has been solved previously, features for perfect play are difficult to specify. However, features such as row completion, fork, and fork of forks may be compiled and used by a search algorithm to reduce the search space.

The features compiled for Qubic are analogous to the first three features created for TicTacToe. "X has a row", "O can create a row" and "X has a fork" were compiled into a feature graph with 3023 nodes, requiring approximately 33 minutes of CPU time. A decision list was then created and used to evaluate nodes in a brute force search. Note that in this case, the decision list does not perfectly evaluate all states.

As in TicTacToe, the Qubic features can recognize winning and losing positions a few ply earlier than an evaluator that simply tests for a completed row. Figure 4 shows a comparison between a brute force search that only recognizes four in a row as a win and a search that uses the decision list. Notice how the search with features finds leaf nodes (i.e. wins, losses and draws) significantly earlier in the search. A search may explore many unsuccessful lines of play very deeply before discovering the single winning line. The reduction in search space produced by the compiled features may then far exceed a simple two ply.

Recognizing a win or loss immediately allows the search to progress to other areas of the search space. Cutting off a line of search earlier prevents the search with features from discovering many

irrelevant leaves, such as those that result from following a non-optimal line of play. This explains the large number of draw positions (64^{th} ply) encountered by the search with features. It has progressed beyond the search without features into an area of the search space where ties are common.

8 Related Work

The KBANN algorithm of Shavlik and Towell (1989) creates an initial network according to a given domain theory, and then inductively adjusts it to fit training data. One advantage to this approach is that an incomplete or inaccurate domain theory is revised during the second stage of the algorithm. Conversely, KBANN supports only propositional domain theories. The network initialization algorithm, which relates closely to feature compilation, can only accommodate expressions with variable free antecedents and consequents. The first order based specification language presented here provides a more descriptive environment for describing domains.

Logic synthesis systems (Michel, Lauther & Duzy, 1992) are used to transform high level descriptions of logic behavior into a lower lever description or even hardware implementation. High level languages similar to programming languages are used to abstract specific behaviors or structures of a hardware system so that a designer may reason about them. The descriptions are then compiled into logical gate implementations or some other low level format.

Etzioni's STATIC (1993) builds a set of search control rules from a problem space definition. Using the problem definition based on operators and axioms to constrain legal states, STATIC creates an AND/OR graph representation of the problem space by backchaining the on the operators. The search control rules are then created from the AND/OR graph representation. Although the methods employed by STATIC are somewhat similar to those described for feature compilation, STATIC does not create a set of general purpose features for state evaluation. Rather, the control rules direct a program to solve a particular problem by moving from an initial state to a specific goal state.

Feature compilation also bears some similarity to the Prolog programming language (Clocksin & Mellish, 1994). This is particularly true when logic forms the basis for the feature specification language. Programming in Prolog is based on describing the known facts of a problem along with the rules and relationships required to infer new facts. Executing a program consists of carrying out the computation specified by logical declarations, and inferring facts from the given information. This is somewhat different from feature compilation, which simply builds a representation of given facts. The compiled feature representation can then be evaluated with respect to a specific instance, but no search for new facts or information is conducted.

9 Discussion

Two important factors are involved in making compilation a viable method for constructing a set of features. The first factor is the specification language, which must be high level, easy to use, and capable of expressing the structure of a domain. Writing feature specifications must be more efficient and at least as accurate as hand coding or learning with no prior knowledge. The second important factor in compiling features is the efficiency of the compiled representation. The size and evaluation cost of the compiled features must be competitive with both hand-coded and learned representations.

The logic compiler presented here addresses each of these demands. The extended version of first order logic discussed above is capable of producing a wide array of statements and descriptions. The AND/OR execution language maps closely to the logic statements and is fully capable of

representing any feature that may be specified. The tight mapping between languages also simplifies the compilation task. Structures are easily mapped from the high level language to the low level representation and structures that are not explicitly specified may be found and exploited.

Converting modified first order logic specifications into AND/OR an graph is only one example of feature compilation. Another, potentially more robust, feature compiler might map statements in a form of threshold logic to a graph of linear threshold units. Threshold logic would enjoy compilation and representation advantages similar to first order logic while possibly creating a smaller feature graphs. Other specification languages and representations are also possible, and may be designed to cater to various types of domains.

There are several advantages to using feature compilation. Feature compilation is an alternative to hard coding domain information directly into performance programs. Structures inherent to the domain can be made clear through the feature specifications, allowing the compiler to build a compact and efficient representation. By allowing human experts to specify known features for a domain, a great deal of training time can be also saved. Current learning algorithms can often spend large quantities of time rediscovering known features of a domain, instead knowledge can be quickly compiled into a useful form.

The ability to impart information to a learning program allows more time for discovering new knowledge. A neural network could probably learn the information contained in the TicTacToe features above, but consider the larger domain of Qubic. The large state space (3^{64}) in this problem causes the learning of high level concepts to become more difficult. Complicated features such as recognizing deeply nested forks become confounded by the need to discover more simple features, such as row completion. However, if a set of compiled features were made available, the network's learning problem would be significantly reduced. Discovery of new domain features becomes the primary network task rather than learning simple, well known aspects of the game.

Compiled features for a given domain need not be complete. Although the TicTacToe features described above were sufficient to evaluate correctly all of the game states, a useful set of features may be created even for domains where knowledge is incomplete. Knowledge in chess is incomplete for example, but features that are known to be important may be compiled and used to improve a performance program. In general, compiled features may be combined with search methods or expanded through learning in order to extend the available knowledge. Learning programs are thus provided with rules and constraints for a domain rather than forcing them to gather all necessary information from examples and mistakes.

10 Summary

Performance programs require a good set of features in order to evaluate accurately states in a domain. Feature compilation presents a method for experts to build knowledge into programs without the distraction of constructing an efficient low level representation of the features.

Feature specifications based on variables quantified over finite domains are first written in a high level language. A compiler then expands the specifications into a set of features by enumerating the variable domains. The low level representation created by the compiler is made both time and space efficient by exploiting the domain structure described in the feature specifications. The compiled features are thus capable of recognizing complex properties, such as those normally discovered through several plies of search, and may be used to reduce significantly the size and difficulty of a learning problem.

Acknowledgements

This work was supported by National Science Foundation Grant IRI-9711239. We thank Richard Cochran, Gang Ding and Margret Connell for helpful comments.

References

Clocksin, W. A., & Mellish, C. S. (1994). *Programming in Prolog.* Springer-Verlag.

Etzioni, O. (1993). Acquiring search-control knowledge via static analysis. *Artificial Intelligence, 62*, 255-301.

Michel, P., Lauther, U., & Duzy, P. (Eds.) (1992). *The synthesis approach to digital system design.* Kluwer Academic Publishers.

Shavlik, J. W., & Towell, G. G. (1989). Combining explanation-based learning and artificial neural networks. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 90-92). Ithaca, NY: Morgan Kaufmann.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning, 8*, 257-277.