REPRESENTATIONAL DIFFICULTIES WITH CLASSIFIER SYSTEMS

Dale Schuurmans
Jonathan Schaeffer

Computing Science Department,
University of Alberta,
Edmonton, Alberta
Canada T6G 2H1

dale@ai.toronto.edu
jonathan@alberta.uucp

ABSTRACT

Classifier systems are currently in vogue as a way of using genetic algorithms to demonstrate machine learning. However, there are a number of difficulties with the formalization that can influence how knowledge is represented and the rate at which the system can learn. Some of the problems are inherent in classifier systems, and one must learn to cope with them, while others are pitfalls waiting to catch the unsuspecting implementor. This paper identifies some of these difficulties, suggesting directions for the further evolution of classifier systems.

1. Introduction

In the last five years, genetic algorithms (GA) have become an expanding area of research in computational models of learning. These models are motivated by concepts from evolutionary biology and population genetics. A recent result of this work has been the development of classifier systems (CS), a simple representational and computational paradigm which uses GAs [Gol89, HHN86]. Already a number of CS implementations have been built demonstrating the potential of this paradigm for machine learning (for example, [Boo89, RoR89]).

At a first glance, classifier representations appear to be a simple, effective method for implementing computational systems that are well suited to manipulation by GAs. However, upon implementing even a simple classifier learning system, many subtleties arise within the representation which can have a strong influence on the learning capabilities of the system.

This paper presents and discusses some of the representational weaknesses inherent in the classifier system methodology which would appear, in general, to hinder genetic search processes. These include search bias, limited disjunction, positional semantics, and parameterization. The intention here is to constructively criticize classifier representation schemes, from the perspective of genetic search algorithms, illustrating the inherent limitations and pitfalls that one is likely to encounter. These observations are useful in suggesting directions for the further evolution of classifier systems.

2. An Illustrative Problem Domain

The representation of a tic-tac-toe board will be used to illustrate many of the observations made in this paper about classifier systems. This problem was chosen for the simplicity and clarity with which we can illustrate our points. We are not addressing the problems of having a CS learn to play perfect tic-tac-toe (a problem that is known to be difficult). None of the points raised in this paper are restricted to our sample domain.

A tic-tac-toe board can be represented as a 2-dimensional grid, with each square being addressed by a row and column coordinate as follows:

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

Each square will contain exactly one of the marks {*X*, *O*, *B* (blank)}. A simple classifier representation of a tic-tac-toe board would encode the current state of the board as a single input message. Squares can be represented using 2 bits with, for example, the meanings X = 01, O = 00, and B = 11. Thus, the input message will be 18 bit long. For example, the message "011100110100111111" would be used to denote the following board configuration:

X	B	O
B	X	O
B	B	B

Note that with this representation, the square number is implicitly encoded into the classifier.

The action string will refer to an address of a square on the board. Were this part of a tic-tac-toe playing program, this square would be the location of the program's next move. This means that we only need to consider 4 bits of the action string. The first 2 bits specify the row and the next 2 bits specify the column. For example, the classifier

111111111111111111 / 0101

means that if every square is empty, then place a mark in the center square. (Strictly speaking, conditions and actions are the same size. For brevity in our examples, we prefer this shorter notation.)

Finally, note that there are some bit values for which there is no attached meaning. This occurs both in our representation of a square (row or column coordinates of 11) and the contents of a square (value of 10).

3. Difficulties with Classifier Systems

Behind the deceptive simplicity of the classifier representation there are a number of subtleties arising in the representation of knowledge which influence the rates and capabilities of learning demonstratable by a CS.

3.1. Search Bias

We have necessarily encoded certain classifier patterns as being illegal. Clearly, in this type of representation, based upon fixed length bit strings, we will always have the problem of illegal patterns (unless, of course, we are fortuitous enough to have 2^k possible values for each field). This property of classifier representations can lead to a number of subtle difficulties in a learning task when we begin to search the space of possible classifiers.

Consider the representation for the contents of a square used in the previous section. Although initially the system will contain many classifiers with illegal values, after time, most of them will disappear (survival of the fittest). Consider performing mutation in a mature population where illegal values have been removed. Mutating a bit on a square containing an X will result in either a B or O. Mutating an O will yield a value of X or I (illegal). Mutating a B results in either X or I. Given that there is an equal probability of a square containing an X, O, or B, then there is a 33% chance of mutating to an X, 16% O, 16% B, and 33% I. Not only is there a bias towards X over O, one in 3 mutations will result in illegal square values, producing a new classifier that is useless.

Even if our representation did not have illegal values, the problems of search bias can still exist. The only time it will not be a problem is when the probabilities of occurrence of field values are equal. For example, if a field should have a value of α 80% of the time and β only 20%, then the mutation operator is more likely to discover an α in that position and mutate it to a β than the other way around. Clearly, this will not result in the proper distribution of α and β.

Finally, the search bias arguments for mutation also apply to cross-over and inversion.

3.2. Limited Disjunction

In any rule-learning task, the object of the system is to distinguish between world states where it is desirable to effect a particular action from those where it is undesirable. It is fundamental that we be able to represent a condition for effecting an action which includes all desirable world states and excludes undesirable ones. That this condition be representable with a small number of rules is a useful property. Clearly, without the ability to in some way encode *generalized* rules, we would be left with the task of simply enumerating the desirable states with their corresponding action.

The presence of the "don't care" symbol (#) in classifier representations is important because it allows us to generalize classifiers by expressing *disjunction* in the condition parts of our classifiers. Disjunction allows conditions to match any one of a set of possible values, thus, reducing the number of classifiers required to implement desired behaviors. Despite its obvious usefulness, "don't care"s only provide us with a limited mechanism for representing disjunction. In fact, most of the possible disjunctive combinations are unrepresentable in a single classifier.

Consider a field of information that can take on any one of n values. For a classifier representation, typically, we would encode this field with $k = \lceil \log_2 n \rceil$ bits. Given three placeholders in conditions {0, 1, #}, we can represent $3^k \approx 3^{\log_2 n} \approx n^{\ln 3/\ln 2}$ possible combinations of the n values. But there are 2^n possible combinations in total. Clearly then, for linearly increasing n there is an exponentially increasing number of unrepresentable disjunctive combinations.

It has been suggested that by utilizing illegal combinations or adding extra bits to the field we could overcome this limitation. To represent all combinations, we would need k placeholders in our conditions, where $3^k \approx 2^n$. Thus, $k \approx \frac{\ln 2}{\ln 3} n$, meaning that to be able to represent every possible disjunctive combination of n values, we require $O(n)$ bits. Therefore, if we want every disjunctive combination represented, we can do little better than to allocate a single bit for each distinct possible value!

The consequences of having unrepresentable disjunctive combinations can range from providing us with beneficial search biases to pruning any reasonable solutions from the search space. What follows is a rather simple problem which demonstrates how limited disjunction, if not taken properly into account, can effectively remove any possibility of expressing a reasonable solution.

Example: Assume the representation X = 01, O = 10, and B = 00. Now consider the problem of having our system learn how to recognize a full board. That is, we want our system to learn to respond with a special message, say "1111", exactly when the board is full (i.e. when no square contains a B), no matter how we arrange the Xs and Os.

Initially, consider only square 0,0. Here, we simply want to express that whenever square 0,0 contains either an X or an O (it does not contain a B) then we should respond with "1111". But notice that given the way we have designed our classifier representation it is impossible to represent the condition *NOT B* (equivalently *X OR O*) using # symbols. Considering that X = 01 and O = 10, we can see that the only pattern matching both is ##, but this pattern necessarily matches B as well! So with our present representation, the solution to this sub-problem requires two distinct classifiers:

01################ / 1111
10################ / 1111

Now, extending this result to consider the whole board requires that we use $2^9 = 512$ classifiers in total (all of which must be discovered by the system)!

If we had shown better luck (or insight) we would have chosen a slightly different representation which would yield a solution requiring *only a single classifier*. Let us alter the representation scheme back to the original one in section 2: X = 01, O = 00, and B = 11. Now *X OR O* can be represented by the pattern 0#, yielding a single classifier which solves the problem:

0#0#0#0#0#0#0#0#0# / 1111

Obviously the above example has been contrived to illustrate our point as blatantly as possible. By changing the bit pattern assignments we wish to use, we may dramatically alter the size of the solution set that the system must discover. This difficulty, we call the *pattern assignment problem*, can be stated as follows:

> Given a field that can take on any one of *n* values, find a bit pattern representation for each value that:
> 1) minimizes the number of bits used to encode all *n* values (reduce the size of the search space), and
> 2) minimizes the number of classifiers needed to express a solution (reduce the size of the solution set).

The mere existence of limited disjunction exposes us to the possibility of requiring a solution set which is combinatorially larger than the minimum size attainable by appropriately assigning bit patterns. To avoid such a threat, a CS designer is forced to consider the potential effect of any pattern assignment upon the representation of a successful solution, meaning that the *designer* would have to know the form of successful solutions *a priori*!

Of course, limited disjunction could be thought of as beneficial in some respects. Any way in which we reduce the size of the search space without removing any desirable solution states benefits us by reducing the required search effort.

In conclusion, utilizing a representation scheme which prohibits certain disjunctive combinations from occurring in the search space can be either a blessing or a curse, depending upon whether or not an optimal (or adequate) solution remains representable. However we would like to emphasize that in utilizing such a representation to our benefit we are, in reality, supplying the learning system with *implicit* domain specific knowledge. That is, we are actually indicating beforehand which combinations are likely to be useful and ruling out others as possible candidates. It is preferable that any knowledge of the task domain that we supply to the learning system beforehand be *explicated.*

3.3. Positional Semantics

In our simple classifier representation for playing tic-tac-toe, we can see that the semantics of the representation are position dependent. That is, to know which particular square a segment of a condition string is referring to, we must know the exact position of the segment in its condition string. In the basic CS/GA framework there is no effective mechanism by which important information can be exchanged between classifier positions. Thus, there is no way important generalizations can be made across positions in a classifier string. If an important pattern has been discovered for one particular position in a classifier string, this same pattern will have to be *independently discovered for each separate position* where that pattern may be important. This can be illustrated with a simple example from our tic-tac-toe domain:

Example: Consider a simple learning task which requires our CS to learn how to make legal moves, and suppose that the system has already learned that it is legal to place an X in square 0,0 whenever square 0,0 presently contains a B:

11################ / 0000

For a complete solution we need to have this information generalized to all squares on the tic-tac-toe board. But notice that to represent this generalization the system will require 9 distinct classifiers, one for each square; and notice further, that the presence of the above classifier *in no way* helps the CS discover this identical "11" pattern for each of the remaining squares .

In keeping with the theme of the paper, we do not wish to remedy this problem by proposing some new *ad hoc* genetic operator which transfers bit patterns between classifier positions. The belief is that the root of this

---

Note that some of the classifier system literature includes the provisions for the *negation* of condition strings [Rio86]. This additional capability does allow for an effective solution to our particular full board problem in an obvious way. However, this does not solve the problem of unrepresentable disjunctive combinations in general and, hence, does not rid us of the pattern assignment problem (discussed shortly).

If this particular example is not convincing, consider the game of *go*. Here, given a similar CS construction we would require 19×19 = 361 distinct classifiers just to express what the legal moves are!

weakness lies in they way in which we have chosen to represent the domain and not so much within the basic search mechanisms.

How can we effectively avoid the pitfall of positional semantics? If we want a CS that can effectively generalize its classifiers along some particular dimension of the problem domain, we must ensure that references to instances of that dimension are made explicitly and not through some implicit, position-dependent mapping. So, in our tic-tac-toe board example, we can see that our initial representation scheme was rather naive in this respect (although this was not obvious initially). To provide for the effective generalization of acquired knowledge along the various board positions, we must make all references to board positions explicit in the representation of our classifiers. This necessitates a change in our formulation of a classifier system representation for a tic-tac-toe board:

Example: Instead of having one input message represent the entire board configuration, we will use 9 distinct input messages, one for each square, each representing the contents of their respective squares. Thus, each message will consist of the two fields: *square* and *contents*, requiring 4+2 = 6 bits per message. Using the pattern assignments from section 2, the set of messages:

{000001, 000111, 001000, 010011, 010101,
011000, 100011, 100111, 101011}

would be used to denote the board configuration given in section 2. The action is encoded using 6 bits, the first 4 bits indicating the address of the square in which to place the mark and the last 2 bits set to "01" indicating that an X is to be placed. For example, the classifier "010100 / 010101 " says that if the center square is empty then place an X in the center square. Now we can express the concept of a legal move with the single classifier

####11 / ####01.

By avoiding position dependent references in our representation we greatly enhance the CS's capabilities to express succinctly important generalizations.

The reader may have realized that even this new representation, while avoiding one pitfall, has introduced a new problem. Consider the case where the action string of a tic-tac-toe board classifier yields the address of a square where a winning move can be made. Here it is necessary, in order to know which squares result in three-in-a-row, that we consider the contents of more than one square at a time when making our decisions. But with the representation just described there is apparently no way for a particular classifier to consider more than a single square at a time. How can we extend the capabilities of our classifiers so that they may consider many squares simultaneously without reintroducing position dependent semantics to our representation? There are two ways in which these types of limitations are overcome in CS implementations: by introducing the possibility of *multiple conditions*, and through coordinated sequences of internal actions (*chaining*). Both of these mechanisms present many new issues relevant to searching the classifier space, and each introduces new problems and limitations.

3.4. Parameterization

We can extend the basic form of a classifier to include the possibility of an arbitrary number of condition strings. Now we consider the total condition of a classifier to be satisfied only if each of the condition strings are satisfied (i.e. the total condition is a *conjunction* of the condition strings). Multiple conditioned classifiers are actually common to many CS implementations [HHN86, Rio86]. Typically, the message that matches the first of the conditions is used by the "pass through" mechanism to construct the action message.

With this additional capability we can utilize information about more than one square in a straight-forward manner for our tic-tac-toe example:

Example: Consider the following classifier:

000001,000101,001011 / 001001

It says that if square 0,0 and square 0,1 contain Xs and square 0,2 contains a B, then place an X into square 0,2 (a good move!). Now consider trying to generalize this particular classifier in a useful way. You may have noticed that we were trying to be clever in the way in which the squares are addressed. That is, by separating the references to rows and columns the hope was to be able to effectively generalize useful information along these dimensions. So an obvious and apparently useful generalization of the previous classifier would be:

##0001,##0101,##1000 / ##1001

which is intended to say that no matter what one particular row, if there is an X in columns 0, 1 and a B in column 2, then place an X into the square in column 2. But unfortunately, this is **not** what the classifier actually does. For example, the set of messages:

{000001, 100101, 101000}

will satisfy the classifier condition, but the resulting action will not necessarily be a good move. In fact, it is impossible in this formulation to express the general concept of a winning move without resorting to enumerating all of the possibilities which, in fact, is all that the first formulation provided us. So we haven't gained anything, with respect to representing the concept of a winning move, by introducing multiple conditions.

We call this particular difficulty the *parameterization problem*, and it arises whenever we jointly introduce # (don't care) symbols and multiple condition strings. The problem is that there exists no mechanism in classifier representations by which we can enforce the equality or inequality of the message bits which match the identical # positions in separate condition strings. Therefore we cannot adequately parameterize solutions which require multiple conditions and some form of agreement between the different conditions about matched messages.

3.5. Chaining

We have argued that the problem of positional semantics arises from a poor choice of problem representation. If so, then this is a pitfall that the inexperienced designer must avoid. Multiple conditions are not the answer. Can chaining be any better?

Let us briefly deviate from our tic-tac-toe example to consider teaching a classifier system to compare two 10-bit numbers, determining their (in)equality. If a message is 20 bits, 10 for each number, it is not possible in this framework to generalize the system to compare, for example, 30-bit numbers. Further more, even if we allowed messages large enough to encompass this problem, it is not clear that knowing how to compare 10-bit numbers will in any significant way be of benefit in learning how to compare 30-bit numbers. The classifier system does not know how to generalize the *dimension* of this problem since, in this example, the dimension is implicitly encoded as the length of the message. Because of the problems of positional semantics in our representation, the solution space is $O(2^n)$, n being the length of the numbers.

A different way of looking at the problem is as a series of ordered 1-bit comparisons. The bits can be compared from high-order to low-order until the solution is found. Because of the implied ordering of comparisons, a different representation of the problem could build a chain to compare successive bits. Consider representing the comparison of n-bit numbers with n messages. The messages would consist of 3 parts: the bit position in the number, the value of that bit in the first number, and the value in the second number. The system would compare the high-order bits and if the answer is not obtained, introduce a new message into the system to match the next order bit. Note that this solution requires a successor/predecessor function. The system can either learn to count itself, or be an additional capability of the system (as, for example, in [Rio86]).

By using these smaller messages, the dimension is now implicitly encoded by the number of messages that enter the system. Further, by reducing the size of the messages, it appears that the search space has been reduced. Instead of using rules n conditions, we can try solving the problem using chains. That is, we utilize the *coordinated* action of a number of rules to realize the same behavior. However, to compare n bit numbers we will require chains that are $O(n)$ in length. Building chains in a classifier system is widely seen as a strength of the formalization, but in practice we believe it to be difficult to achieve.

The above solution has reduced the number of rules in the solution space to $O(n)$. However, the solution is of a repetitive nature and, because of the limitations of classifier systems, must have a rule for each possible iterate. In other words, it is impossible to arrive at a solution requiring the number of rules independent of n.

But is it desirable that a simple problem such as comparing numbers have a solution requiring $O(n)$ rules? With a more powerful formalization, this problem can be solved using a *constant* number of rules. To break the problem into a series of 1-bit comparisons requires a variable looping construct not found in classifier systems. Classifier systems as defined by Holland have only the capabilities for expressing boolean expressions (and, or, not) and nested if-then-else statements (chains). Chains must be explicit and of finite length. There is no facility for parameterizing a rule to create a variable length chain.

3.6. Effect of solution size on search

An often discussed point in the literature is that classifier systems are computationally universal [Gol89, HHN86]. Given this, one cannot say that there are solutions which exist for some problems that a CS cannot express. Obviously given any rule based system that is computationally universal, we can always compensate for limitations in our representation of individual rules through the coordinated actions of many rules. However, we must keep in mind that we intend to automatically discover solutions with some form of mechanized search over the space of possible rules (classifiers).

Throughout the paper a recurrent theme has been to focus on those properties of classifier representations which impair our ability to *succinctly* express solutions to problems. The effectiveness of a genetic search relies on the fact that a classifier which has demonstrated a high level of fitness has many sub-patterns (called schemata [Hol75]) in common with an optimal classifier. These sub-patterns constitute the *building blocks* from which an optimal classifier can be constructed [De85, De87]. Consider that we have some solution that is only expressible with, say, n classifiers. This implies that each of the classifiers has some important sub-pattern that is distinct from the corresponding sub-patterns of the other classifiers. So, at some stage of the overall search, our efforts must be divided among n distinct searches, where progress towards discovering any solution classifier in no way contributes towards finding the other solution classifiers.

By utilizing simplified representations (and, hence, limiting our capabilities for expressing arbitrary rule forms) we *do not* inherently exclude any problem solutions from being expressible, but we *do* expose ourselves to the possibility of a combinatorial explosion in the number of rules required to express these solutions. We would claim further that such an explosion in the solution size necessarily manifests itself as an explosion in the search effort required to discover that entire solution. This creates a problem that we, as "learning system" designers, can only avoid with certainty by having sufficient knowledge about the solution *a priori*!

It is interesting to note that all of these problems seem to disappear if we consider using a sufficiently general representation scheme (for example, OPS type production systems [For81]), but this comes at the expense of vastly increasing the size of the search space. At present, there exists no detailed analysis from the machine learning perspective of this trade-off between solution set size and search space size. A better understanding of this issue would contribute a great deal towards constructing effective, domain independent learning machines.

4. Conclusions

Classifier systems are an important step in the evolution of genetic algorithms. One cannot underestimate their contribution as an analytic tool for research into adaptive systems. These representations provide an elegant, simple model which has been used to produced a wide range of important results, extending our basic knowledge of adaptive processes. In terms of implementing real, practical learning systems, however, we believe that more principled and general representational methodologies are needed.

There are a number of limitations that restrict the ability of classifier systems, enhanced with genetic capabilities, to learn. In this direction, we believe that capabilities such as parameterized chaining must be available to the system [ShS89]. As well, the system should have access to other knowledge sources. These enhancements would only endow classifier systems with more of the functionality found in production systems.

Acknowledgments

This work evolved out of discussions with a number of people including Robert Chai, Renee Elio, Keith Fenske, and Lingyan Shu. David Goldberg provided us with valuable feedback. This work was supported in part by the Canadian Natural Sciences and Engineering Research Council.

References

[Boo89] L.B. Booker, Classifier Systems that Learn Internal World Models, *Machine Learning 3*, 2/3 (1989), 161-192.

[De85] K. De Jong, Genetic Algorithms: A 10 Year Perspective, *ICGA*, 1985, 169-177.

[De87] K. De Jong, On Using Genetic Algorithms to Search Program Spaces, *ICGA*, 1987, 210-216.

[For81] C.L. Forgy, OPS5 User's Manual, Department of Computer Science, Carnegie-Mellon University, 1981.

[Gol89] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

[Hol75] J.H. Holland, in *Adaptation in Natural and Artificial Systems*, 1975.

[HHN86] J.H. Holland, K.J. Holyoak, R.E. Nisbett and P.R. Thagard, in *Induction: Process of Inference, Learning, and Discovery*, 1986, 124-126.

[Rio86] R. Riolo, CFS-C: A Package of Domain Independent Subroutines for Implementing Classifier Systems in Arbitrary, User-Defined Environments, Logic of Computers Group, Division of Computer Science and Engineering, University of Michigan, 1986.

[RoR89] G.G. Robertson and R.L. Riolo, A Tale of Two Classifier Systems, *Machine Learning 3*, 2/3 (1989), 139-160.

[ShS89] L. Shu and J. Schaeffer, VCS - Variable Classifier Systems, *ICGA*, 1989. To appear.