VCS: VARIABLE CLASSIFIER SYSTEMS

Lingyan Shu
Jonathan Schaeffer

Computing Science Department,
University of Alberta,
Edmonton, Alberta
Canada T6G 2H1
shu@alberta.uucp
jonathan@alberta.uucp

ABSTRACT

Classifier systems (CS) have proven to be useful tools for the study of genetic algorithm based learning. Unfortunately, there are a number of difficulties with the formalization that limit the representational capabilities and, hence, its problem solving abilities and the speed at which it can learn. This paper introduces VCS - Variable Classifier Systems - that augment the traditional CS with the binding of constants in messages to variables in rule conditions. For a large class of problems, VCS allows for a more succinct representation of the solution space than is possible with CS, increasing the likelihood of a genetic search successfully solving the problem. Variables make it possible for these systems to represent information in ways similar to high level symbolic representations, narrowing the gap between classifier systems and conventional learning systems.

1. Introduction

Genetic learning systems have relatively simple representation structures. Usually, little prior knowledge is built into such systems. As a result, genetic learning models work well with incomplete information and noise. Also, they can be used with multiple domains at the same time, allowing the transfer of information from one domain to another [Hol86]. In contrast, the symbolic concept acquisition and the knowledge intensive, domain specific learning systems [MCM86] contain numerous built-in concepts, representation structures and domain specific constraints. It is easier for a human to communicate with such systems. These systems take advantage of a variety of rich knowledge sources (built-in knowledge, knowledge models, advice, etc.) and tend to be more efficient than the genetic learning approaches. Researchers have noticed that combining strategies, such as learning from advice, into genetic learning would provide a more powerful model in the sense of effectiveness and efficiency [BeF88, HHN86]. Representation issues are an important factor that affects the realization of this suggestion.

The prevalent representation framework for genetic algorithm based machine learning is the classifier system [Gol83, Gol88, HoR78, HHN86, Hol86, Rio86, Wil86]. Classifier systems are general purpose inductive machine learning systems which use genetic algorithms [Boo82, DeJ75, Hol75] and the bucket brigade algorithm as the learning mechanisms. Rules in a classifier systems are represented by strings of symbols. For example, most of the classifier systems use strings over the alphabet {0, 1, #} to represent rules, where "#" means either "0" or "1". This unified representation is amenable to the use of genetic operators as the learning mechanism. Also, low-level representations, such as bit strings, are capable of describing cognitive processes because many cognitive phenomena occur below the level represented by symbolic models [BeF88]. These characteristics make genetic learning more effective and general in many ways than symbolic concept acquisition and domain specific learning. However, because of the representation limitations, some knowledge which can be represented symbolically cannot be adequately represented in classifier systems. The simplistic representation scheme makes it hard to manipulate knowledge and to add built-in knowledge and models which can be used to describe features common to the problem domain. Compared with symbolic representations systems, it is difficult for classifier systems to abstract and explain events. They are incapable of representing information about commonly occurring patterns, such as what frames or semantic nets do in symbolic representations.

This paper introduces VCS - Variable Classifier Systems - in which classifier systems (CS) are evolved to include variables. Conditions and actions in rules are broken into fields, each of which can contain a constant(s) (using notation similar to the CS {0,1,#}), or a named variable. Variables can be used to ensure equality and move information between fields. The result is that variables allow some problems to have their solutions expressed in a succinct manner, reducing the amount of work required by genetic search to solve a problem. As well, variables provide a way of describing an abstract world, allowing for the building of model rules and knowledge structures as in high level symbolic representation systems. It thus becomes possible to combine the advantages of high and low level representations into one framework.

2. VCS Framework

VCS is a low-level representation framework, similar to a classifier system, in which the idea of a variable is introduced. Variables are used to describe abstract relations in a succinct manner, reducing the size of the solution set for many problems. Variables make VCS more powerful in expressiveness than CS. Some of the representational difficulties inherent in classifier systems [Sch88] can be overcome in the VCS framework.

VCS retains the structure of classifier systems. There are message and rule lists and, in each cycle, messages match rule conditions generating new messages from matching actions. Genetic operators and the bucket brigade operate as in CS. Rules can be negated and have multiple conditions. The difference between VCS and CS lies in the syntax and semantics of messages and rules.

The basic symbols in VCS are the alphabet {0, 1, ?, *} and a set of logical field placeholders. The question mark "?" sign replaces the "#" in classifier systems and represents the don't care or don't know situation. The "don't know" situation can be used as a prompt for the user to provide some information. This allows the system to communicate with the external world by providing useful feedback and prompting for useful information. The symbols "0" and "1" in a condition can be matched only by "0" and "1" in a message respectively. The "?" in a condition can be matched by "0", "1" or "?". A "?" in a message can only be matched by "?" in a condition, this being a major distinction between the "?" and the "#" of classifier systems.

Messages, conditions, and actions are implemented as fixed length bit strings. The semantics of this representation are that they are composed of a number of fields, each field representing some piece of information of the problem domain. To distinguish the implementation (unreadable strings of "1"s and "0"s) from the semantics, field identification placeholders are inserted in messages, conditions, and actions to identify the contents of each field. For readability, we use the notation $P_{contents}$ to name a field, where the identifier *contents* is meant as a meaningful interpretation of that field's contents. For brevity, we often name fields numerically $\{P_1, P_2, P_3, ..., P_n\}$. For example, consider the blocks world, an oft-cited example that many problem solving programs use. In the VCS representation, a rule condition could consist of two fields:

P_{name}: the name of a block

$P_{relation}$: relations between this block and other blocks

and the condition could be represented as

P_{name} `10` $P_{relation}$ `0110110`

where "10" is the name of a block and "0110110" are its relationships. Removing the field names yields the familiar classifier system notation. Field names are interpretation aids and do not affect the underlying implementation.

So far, there is little that is different from classifier systems. The major difference is the inclusion of the "*" operator into rule conditions and actions (but not messages). The "*" character in a field (anywhere in the field) indicates that the contents of the field should not be interpreted as a constant but as a variable. For example, a 3 bit field could contain *01, 00*, or *?*, any of which indicates that the field is to be treated as a variable. Again, it is important to separate the semantics from the implementation. The {0, 1, ?, *} notation is for the implementation. Instead, we prefer to use names for our variables, such as *x*, *y*, and *z* (instead of *01, 00*, *?*) to increase the clarity. Using the blocks world example again, the condition

$P_{name}\ x\ P_{relation}\ y$

indicates that the two fields are variables. For example, the condition

P_1 `*00` P_2 `?1` P_3 `0**0` P_4 `*00`

is equivalent to

$P_1\ x\ P_2$ `?1` $P_3\ y\ P_4\ x$

where P_1's x is different from P_4's x.

The semantics of matching messages with rule conditions are different from CS. A condition field containing only {0, 1, ?} is treated as a constant and matches the corresponding field in a message if all the characters match (recalling the slightly different semantics of the ? mentioned above). A condition field that is a variable matches the corresponding field in the message, *regardless of the contents of that field.* As such, we view each field as being a *parameter* of the rule; matching then becomes the binding of values (from the message) to parameters (conditions of the rule).

The user must define his messages in terms of the number of fields and the size of each field. Names do not have to be unique; two fields with the same name must be the same size. Fields with the same name have the ability to exchange information between them through variables. If the same variable appears in more than one condition field of the same name, then it must be bound to the same value in all cases, otherwise the matching fails (see example below). This feature allows the equality relationship to hold between fields in (multiple) conditions.

Variables can also appear as part of the action of a rule. The variable takes its value from that assigned in the condition. If the variable does not appear in the condition part, the variable in the action part will randomly take a value from the value domain of the field. Consider the rule in which the "/" is used to separate condition from action

P_1 `11` P_2 `*00` P_2 `*01 /`
P_1 `01` P_2 `*01` P_2 `*00`

with semantics

P_1 `11` $P_2\ x\ P_2\ y$ / P_1 `01` $P_2\ y\ P_2\ x$

Then the message

P_1 `11` P_2 `101` P_2 `111`

will match the rule, and result in the information exchange between the last two fields. The resulting message will be

```
P_1 01 P_2 111 P_2 101
```

Note that information exchange can only occur between the second and third fields since they have the same name. The exchange of values between fields is not possible in classifier systems, without enumerating all the possibilities.

When using variables, some rules with different appearances may have the same semantics. For example the rules

if v_i and v_{i+1} then v_{i+2}
if v_j and v_{j+1} then v_{j+2}

have the same meaning. To avoid this, our system enforces a *normalized* representation for rules. In this representation, each rule must use the minimum number of variables possible, and the variable names must be ordered and sequentially starting from a fixed name (e.g., v_1, v_2, v_3, ...). We say that the order of v_{i+1} is higher than the order of v_i. In a rule, a higher order variable name for a parameter can be used only after all the lower order variables of the same parameter have been used. Under this policy,

$P_1\, v_1\, P_1\, v_2$ / $P_1\, v_2\, P_1\, v_1$

is legal. But

$P_1\, v_2\, P_1\, v_1$ / $P_1\, v_1\, P_1\, v_2$

is not. As a result, there are many illegal rule combinations. The *semantics* of VCS insist that semantically illegal rules with respect to variable usage are not allowed to be created. This greatly reduces the search space of possible rules.

One last note on variables. A field may not be defined large enough to hold the name of all possible variables that could occupy it. For example, if there are *n* fields and a particular field is only 1 bit long, one cannot represent *n* variable names. VCS recognizes this and expands fields, if necessary, so they can properly accommodate all possible values. The extra bits are hidden from the user.

3. Properties of VCS

3.1. Simplicity

VCS inherits the simplicity of classifier systems. Every concept and situation can be generated by the genetic operators due to the uniform representation.

3.2. Expressiveness

VCS can be considered as a general parameter representation framework. To use the framework, the designers need only decide what fields are required to describe their problem (as in CS). Compared with CS, VCS is more expressive. One obvious example is that a classifier system can not represent relationships among fields without enumerating all possibilities. For example, as has already been shown, the symmetric relation "if R(x, y) then R(y, x)" is easy to do in VCS but not in CS. Another example is the transitive relation "if R(x, y) and R(y, z) then R(x, z)". Using multiple conditions in VCS (separated by commas), this relation could be represented as

$P_1\, x\, P_1\, y$, $P_1\, y\, P_1\, z$ / $P_1\, x\, P_1\, z$

In classifier systems, there is no way to enforce equality of fields between conditions.

3.3. Search Space

Classifier systems use strings on the alphabet {0, 1, #} yielding a search space of size 3^l, given *l* characters in a message. In VCS, the "*" augments the alphabet {0, 1, ?}, yielding an upper bound of 4^l on the search space. Normalizing the representation greatly reduces the number of semantically correct possibilities.

What is a bound on the search space, given normalized and semantically correct rules? An upper bound on the average search space is $S_{VCS} = 3^l + N_n * 3^{ma}$, where N_n is the number of all legal strings that contain variables, *n* is the number of fields, *m* is the average length of the fields, and *a* is the average number of the fields which are constants in a legal string that contains variables. Given *n* fields, it is possible to have up to *n* variables in a rule. Then $N_n = f_{(n,\,0)} + f_{(n,\,1)} + ... + f_{(n,\,n)}$, where $f_{(n,\,0)}$ is the number of strings with *n* fields that has no variable name in the last field; $f_{(n,\,i)}$ is the number of the strings containing variables and having variable v_i in the last field. Then

$$N_n = f_{(n,\,0)} + ... f_{(n,\,n)}$$
$$f_{(1,\,0)} = 1$$
$$f_{(1,\,1)} = 1$$
$$f_{(n,\,i)} = \sum_{j=i-1}^{n-1} f_{((n-1),\,j)},\ \ i = 1, \ldots, n$$
$$f_{(n,\,0)} = \sum_{j=0}^{n-1} f_{((n-1),\,j)},\ n = 1, 2, 3, \cdots$$

For most practical values of *n*, *m* and *l* ($l = n \times m$), S_{VCS} is much closer to 3^l than 4^l.

We can also look at the search space in another way. $Illegal = (n-2) \times (n+1)^{(n-1)} \times 3^{ma}$ is a lower bound for the number of the illegal patterns. Thus $4^l - Illegal$ is an upper bound for the average search space.

VCS still maintains the implicit parallelism of classifier systems. In CS, each string is a representative of 2^l schemata. In VCS, the "*" is used and any string could be an instance of a string containing a "*". Given that all possible combinations of {0, 1, ?, *} were legal, the search space would be 4^l and each string is a representative of 3^l schemata. However, normalization and semantics make these numbers smaller, similar to that for CS.

Using multiple conditions in a classifier has the potential to reduce the search space for some problems. In the fixed length single condition CS representation, the condition string must include all information necessary for solving the problem. For example, a representation for the game of tic-tac-toe might include all 9 squares on the board [Sch88]. However, VCS allows relationships to hold between multiple conditions. This permits smaller messages to be used, using multiple conditions to bind them together. For example, in the tic-tac-toe case, one could have each message representing the contents of one square. Then using three conditions for a rule, one can define *in a single rule* the information that 3 in a row wins. This is not possible in classifier systems.

Consider another example. The task is to check whether there are some terminals available in a room containing 6 terminals. We can use 2-bit strings to represent the state of terminal: occupied = 00, available = 01, and damaged = 10. Since there are 6 terminal, we use 3-bit strings to represent the terminal names: terminal 1 = 001, terminal 2 = 010, etc. We want to represent a usage constraint

```
if two or more terminals are available
then two or more people can use them.
```

In the CS representation, the condition of the rule would contain information on all 6 terminals, for example:

```
00 01 00 00 01 10
```

with the names of the terminals represented implicitly by their position in the rule. The usage constraint would then be represented as

```
00 00 ## ## ## ## / (terminals 1&2 available)
00 ## 00 ## ## ## / (terminals 1&3 available)
etc.
```

$O(N^2)$ rules are needed to represent all the conditions of the usage constraint, where N is the number of terminals.

In the VCS representation, the constraint can be represented as:

P_{name} x P_{state} `00,` P_{name} y P_{state} `00 /`
`(terminals` x&y `available)`

In the VCS representation, only 5 bits are needed for each condition; a total of 10 bits. In contrast, the solution space consists of only *one* rule . Note that CS can properly express the same multiple condition as in VCS, but it is not possible in CS to pass both terminal names to the action part.

Some problems can have their solutions expressed succinctly in VCS, allowing for a simpler solution and a correspondingly faster search.

3.4. Position Independent Property

In CS, representation inherits from the genetic example. Locus information is represented implicitly by the positions of the bits in one string. In VCS representation, locus information can be represented explicitly, making it possible to manipulate locus information. Assume n is the number of the fields in a string, m is the average length of a field and s is the average number conditions per rule (given the presence of multiple conditions). $\lceil \log_2 n \rceil$ extra bits may be needed to represent this information. If s is small enough and $2m > \lceil \log_2 n \rceil$, the actual string length will not increase.

3.5. Register Property

When genetic algorithms are used as tools of function optimization, no message passing is involved. When applied to learning, message passing is an important factor that affects both the effectiveness and the efficiency of a system. The message passing scheme of the CS is not effective in some situations (see the following examples). The effect of the VCS field identifications and variables provides the system with a register ability: a value stored in a variable can be used in a variety of places. The following examples show that in some situations, VCS can easily do what CS cannot.

Example 1

Variables allow values to move to different fields. Consider the rule

`if` x `is on` y `then` y `is below` x.

There is no way of representing this relation in CS, short of enumerating all possibilities. The VCS solution for the above problem is:

P_1 `01` P_2 `*0` P_2 `*1 /` P_1 `10` P_2 `*1` P_2 `*0`
`(on)  x  y  / (below)  y  x`

with the values of x and y in the condition being passed correctly to the action part.

Example 2

In CS, message passing can cause a performance bias. Consider the rule

```
if object is small and moving fast
then slow-down or ignore
```

A classifier representation might look like this:

```
   1       1    /   1                #
(small) (fast) / (tag) (slow-down or ignore)
```

Only message 11 can fire this rule. Through the message passing, the # always gets value 1. If value 0 means "ignore", then this case will never be chosen.

Consider the rule

```
if object is small or big
then slow-down or stop
```

The classifier representation could be:

```
      #           1    /        #           1
(small or big) (fast) / (slow or stop) (misc)
```

Under the classifier system message passing rules, this equates to:

```
   0       1    /    0      1
(small) (fast) / (slow) (misc)
              and
   1       1    /    1      1
 (big)  (fast) / (stop) (misc)
```

which is not the original meaning. The VCS solution for this rule is

P_1 `10` P_2 `10 /` P_3 `*1` P_1 `11`
`(small)  (fast) / (slow-down or ignore) (misc)`

Strictly speaking, the VCS solution is incomplete. If two messages exist stating that terminal 1 is available, then they could match the VCS rule, yielding the incorrect action "terminals 1 & 1 available". A solution could involve the use of default hierarchies, i.e. using an extra rule to eliminate duplicate entries. A condition such as "P_{name} x P_{state} 00, P_{name} x P_{state} 00" would find duplicates. This cannot be done in CS.

With register capabilities, we can group some properties together to form concepts. By assigning and changing the range of a field P_i, we actually changed the concept P_i. Also, the register property allows equations to be expressed in VCS which can not be done in CS.

The VCS framework has some other potentials for inductive learning. For example, by using a new identification sign P_{n+1} or by augmenting the value field of P_i, the problem space can be augmented. Thus, a four-block world can be enlarged to a five-block world without much change.

3.6. Build Structures by Building Abstract Relations

In classifier systems, existing knowledge can be incorporated into the system by an initial classifier pool or by the input messages. However, it is hard to incorporate into these systems the relations among the knowledge components. Structural knowledge cannot be easily built in and manipulated. The strength of the structural knowledge representations lies in the ability to represent relations among the knowledge components through the representation structure itself, providing an efficient way to manipulate these relations.

Frames and semantic networks are two popular structural knowledge representations. Super-ordinate (*super*) and sub-ordinate (*sub*) are two relations that appear in these high-level structural representations. Also, they are the relations among different levels of default hierarchies. These relations are partial orderings on their domains. By including rules such as

`if X` *super* `Y and Y` *super* `Z then X` *super* `Z`
`if X` *sub* `Y and Y` *sub* `Z then X` *sub* `Z`
`if X` *sub* `Y then Y` *super* `X`

we can build a knowledge structure into the system. By using abstract relations, we link the corresponding knowledge together. Then in default hierarchy situations, knowledge can be triggered in the same way as using a high level structural representation. Notice that the tagging method of representing inheritance in Belew and Forrest's work [BeF88] cannot activate concepts with sub-ordinate and super-ordinate relations in two directions at same time. This implies that the tagging classifier representation of the semantic network is not equivalent to the high level semantic network representation. However, by adding in the above three relation model rules, the representation is equivalent to the semantic network representation.

4. Problems in VCS

There are still some problems in VCS that need further study. The use of variables increases the complexity of the systems. Variable matching, binding, and normalization are extensions not occurring in classifier systems. Clearly, VCS represents a more complicated implementation.

Right now, VCS knows about equality of variables between fields. Inequality between variables is cumbersome, but can be achieved through extra rules that test for equality and have the rule negated (i.e. using default hierarchies). There are no provisions for other boolean or mathematical relationships between fields.

VCS, with a fixed size rule list and fixed size message lengths, is not a Turing machine. However, the succinctness of solutions increases the space of problems potentially solvable by VCS.

5. Conclusions and Further Work

VCS is a low-level representation framework with some high-level representation abilities. Compared with the variable scheme used in Smith's LS-1 [Gol88, Smi80], the VCS variable scheme is more general. The variable sign "*" in VCS is treated uniformly as other symbols. It is possible to implement learning mechanisms that are used in other frameworks in VCS. Knowledge structures can be built into VCS applications with some model relation rules. Future work includes examining how VCS would help establishing and maintaining rule associations such as chains and default hierarchies.

VCS also has promise for use in chunked learning. The traditional view of chunks is that they are the compact representations of several items. A schema is an example of a chunk. The more recent view of chunks is that they could also be data structures representing processes or procedures. From this point of view, functions in conventional programming languages are examples of chunks. The introduction of variables to classifier systems makes rules begin to look like functions. Values are bound to function parameters when a message matches a condition. The action part is just the return value(s) from the function call. Between the condition and action parts is a simple function body. If VCS continues to evolve, making the function describing the mapping of input to output values more sophisticated, then these systems become more like a conventional programming language. The only major difference is the method used for representing values and variables. Maybe this implies that the simple binary representations of CS and VCS should be the next area for evolution. The implementation of chunks as functions would make the knowledge being structured as a lattice rooted in a set of pre-existing primitives [RoN86]. How to detect, encode and decode a chunk in VCS, perhaps using techniques from traditional programming languages, would be an interesting research project.

The VCS system described in this paper has been designed, and is being implemented.

Acknowledgments

Joe Culberson, Renee Elio, and Randy Goebel helped clarify many of the ideas presented in this paper. Financial assistance from the Natural Sciences and Engineering Research Council of Canada is greatly appreciated.

References

[BeF88] R.K. Belew and S. Forrest, Learning and Programming in Classifier Systems, *Machine Learning 3*, (1988), 193-224, Kluwer Academic Publishers.

[Boo82] L.B. Booker, *Intelligent Behavior as an Adaptation in the Environment*, Ph.D. Thesis, Technical Report, University of Michigan, 1982.

[DeJ75] K.A. DeJong, *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Ph.D. Thesis, University of Michigan, 1975.

[Gol83] D.E. Goldberg, *Computer-Aided Gas Pipeline Operation Using Genetic Algorithms and Rule Learning*, Ph.D. Thesis, University of Michigan, 1983.

[Gol88] D.E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley Publishing Company, 1988.

[Hol75] J.H. Holland, *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, 1975.

[HoR78] J.H. Holland and J.S. Reitman, Cognitive Systems Based on Adaptive Algorithms, in *Pattern Directed Inference Systems*, D.A. Waterman and F. Hayes-Roth (ed.), Academic Press, New York, 1978, 313-329.

[HHN86] J.H. Holland, K.J. Holyoak, R.E. Nisbett and P.R. Thagard, *Induction: Processes of Inference, Learning, and Discovery*, MIT Press, Cambridge, 1986.

[Hol86] J.H. Holland, Escaping Brittleness: The Possibilities of General Purpose Learning Algorithms Applied to Parallel Rule-Based Systems, in *Machine Learning II*, R.S. Michalski, J.G. Carbonell and T.M. Mitchell (ed.), 1986.

[MCM86] Introduction, in *Machine Learning II*, R.S. Michalski, J.G. Carbonell and T.M. Mitchell (ed.), 1986.

[Rio86] R.L. Riolo, LETSEQ: An Implementation of the CFS-C Classifier System in a Task Domain that Predicts Letter Sequences, *Technical Report, University of Michigan*, 1986.

[RoN86] Paul S. Rosenbloom and Allen Newell, The Chunking of Goal Hierarchies : A Generalized Model of Practice, in *Machine Learning II*, R.S. Michalski, J.G. Carbonell and T.M. Mitchell (ed.), 1986.

[Sch88] D. Schuurmans, Representation and Selection Techniques for Genetic Learning Systems, M.Sc. Thesis, University of Alberta, 1988.

[Smi80] S.F. Smith, A learning system based on genetic adaptive algorithms, *Ph.D. Thesis, University of Pittsburgh*, 1980.

[Wil86] S.W. Wilson, Knowledge Growth in an Artificial Animal, in *Adaptive and Learning Systems: Theory and Applications*, K.S. Narendra (ed.), Plenum, New York, 1986, 255-264.