Prioritized Rule Systems

A Thesis

Presented to the

Department of Computer Science

Brigham Young University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Brent W. Hughes

November 1989

This thesis, by Brent W. Hughes, is accepted in its present form by the Department of Computer Science of Brigham Young University as satisfying the thesis requirement for the degree of Master of Science.

____
Tony Martinez, Committee Chairman

____
Douglas Campbell, Committee Member

______________ ________________________________
Date Larry Christensen, Graduate Coordinator

ACKNOWLEDGEMENTS

The author wishes to thank Dr. Tony Martinez without whose help this work would never have been accomplished. He has provided motivation, assistance, and an open door providing the opportunity for many interesting and worthwhile discussions. My thanks go also to Dr. Douglas Campbell who has also extended many hours of his time in regards to this and other work. I am grateful for all those at Brigham Young University who have been "teachers", whether formally or informally. I am grateful for the spirit at BYU and for all those employees who understand the difference between the spirit of the law and the letter of the law, who are not just doing a job but who are truly striving to help.

Finally, I express my gratitude to my family — to my wife, Christine, and my children, Stephanie and Jody, without whom this would not have meant much.

Prioritized Rule Systems

1. Introduction

2. Prioritized Rule Systems - Informal Development

3. Prioritized Rule Systems - Formal Development

4. A PRS Architectural Model

5. Additional Capabilities

Chapter 1

Introduction

1.1 Overview

The purpose of this thesis is to present a processing model called Prioritized Rule Systems (PRS), an extension of ASOCS (adaptive self-organizing concurrent systems) (Martinez, 1986; Martinez, 1987). The new model has a number of advantages over the regular ASOCS model. These advantages are described later.

The new model inherits many of the attributes of ASOCS. A system based on these models is connectionist in nature. It consists of many simple processing elements operating independently and in parallel. The network maps Boolean inputs to Boolean outputs. The desired functional mapping is specified to the system in the form of if-then rules. The mapping is propositional in nature and any arbitrary mapping may be represented and learned by the system. The system is adaptable; it can learn new rules over time in an incremental fashion. Each new rule is learned with only one presentation of the rule. The system is not "programmed", but is self-organizing. Such a system has two operating modes. In the learning mode, it adapts to new rules (perhaps overriding older conflicting rules). In the processing mode, it acts much the same as a standard logic network mapping inputs to outputs. Both the learning of new rules and the processing of inputs takes place in $O(log(n))$ time where n is the number of nodes in the system. These systems have applications in adaptive logic, inferencing, robotics, remote systems, and adaptive control.

Prioritized Rule Systems have these advantages over ASOCS: Learning is simpler. Consistency of the system is maintained in a more straight forward manner. Fewer nodes are required to represent a set of rules. The number of nodes is bounded by the number of presented rules. A more comprehensive form of generalization is available in a PRS. In an architecture based on the PRS model, the position of nodes in the network is irrelevant; in current ASOCS architectures this is not true. This means that nodes can be used more efficiently in a PRS.

Prioritized Rule Systems do have some disadvantages over regular ASOCS. PRS nodes use more complex logic and additional information must be passed through the network.

We now provide some historical background as motivation for what follows.

1.2 Non-von Neumann Models

In his Turing Award lecture (1978), John Backus introduced the term "von Neumann computer" to describe the typical computers of today which are based on John von Neumann's ideas of 40 years ago. Although von Neumann is usually honored for those ideas, Backus's lecture served to emphasize a significant weakness of the von Neumann design which Backus referred to as the "von Neumann bottleneck". Backus spoke figuratively of a narrow tube between processor and memory through which all data must pass essentially one word at a time. Backus suggested that the time had come to overcome this bottleneck with new languages and new computer designs.

The idea that there might be an alternative to having a single processor work on memory one word at a time was not a new one, however. John von Neumann himself made significant contributions to the field of cellular automata, a very "non-von" type of computing machine with many simple processing units operating concurrently to modify many "words" of memory simultaneously.

A short history of some of the efforts in "non-von" computing from perceptrons to neural nets to ASOCS to Prioritized Rule Systems is now presented.

1.3 Perceptrons

In his book *Principles of Neurodynamics* (1962), Frank Rosenblatt discussed simple neuron-like learning networks which he called *perceptrons*. These "machines" could learn to classify certain sets of patterns depending on the configuration of the machine. A perceptron consists of a number of simple processing elements that operate in parallel. Although the exact details of the configuration of a perceptron are not needed in the discussion that follows, it is important to know that there are connections between certain of the "nodes" of the network that have modifiable "weights" or "strengths".

It is the modification of these weights that allows the perceptron to "learn".

One of the most important things to come out of Rosenblatt's work was the perceptron learning algorithm and its associated theorem. Stated simply, the theorem says that if a perceptron can represent a classification, then it can also "learn" that classification. Because of its importance, we will discuss this theorem in some detail.

The input to a perceptron can be thought of as a vector of two-state Boolean values representing some pattern. The output of the perceptron is a single two-state Boolean value indicating in which category the input pattern is placed. A set of vectors (patterns) can therefore be partitioned by the perceptron into two disjoint sets. The perceptron can be thought of as an implementation of a Boolean function of Boolean vectors. It maps each input vector into a 0 or a 1. If each vector consists of n Boolean values, then there are 2**n possible inputs to the network. (Two consecutive asterisks indicate exponentiation.) Since each input can be mapped to either of two outputs (0 or 1), there are 2**(2**n) possible mappings from n Boolean values to 0,1.

Can any perceptron configuration learn any of the 2**(2**n) possible mappings? No. A particular perceptron configuration may not be able to represent a particular mapping. That is, there may or may not be a setting of the weights that implements the specified mapping. However, if there does exist some setting of the weights that represents the mapping, then the perceptron learning algorithm, started with any settings on the weights, is guaranteed to be able to find a setting that does in fact represent the desired mapping.

Does there exist a specific perceptron configuration that can learn any of the 2**(2**n) possible mappings? Yes. However, it uses an exponential number of nodes. Does there exist a specific perceptron configuration that can learn any of the 2**(2**n) possible mappings using less than exponential resources? Information theory suggests that the answer is no. Many so called "random mappings", such as being able to classify random raster outputs from untuned TV screens, would indeed require exponential space or time computational abilities. However, the class of applications for which neural networks (and our work) is targeted have regularities and features which can be detected such that classification can be accomplished with much less than exponential demands (Martinez, 1989). Most patterns and decisions which we as humans efficiently classify fall into this realm.

Marvin Minsky and Seymour Papert published a book called *Perceptrons* (1969) in which they provided a rigorous examination of the perceptron. They proved a number of theorems which many saw as proving severe limitations on the perceptron model. In fact, much of the excitement concerning perceptron-like models faded away following the publication of Minsky and Papert's book.

1.4 Neural Nets

Some researchers felt that, although the perceptron itself, which used a single layer of modifiable weights, might be dead, an extension of the model to multiple layers might provide the additional power for which they were searching (Rumelhart & McClelland, 1986). However, for multi-layer perceptrons, no one had yet found a learning algorithm and an associated convergence theorem equivalent to that of the single layer perceptron. Further, in *Perceptrons*, Minsky and Papert expressed the opinion that no such algorithm and theorem would be found for multi-layer perceptrons. To this date, no one has proved them wrong. (Nevertheless, some practical and useful systems have been built without such a theorem.)

In spite of the lack of such learning guarantees, the field has recently been reborn. The new incarnation of this field is referred to as the field of neural nets in reference to the inspiration that such models receive from the network of neurons found in nervous systems (including the brain). Some interesting results have been obtained and some researchers feel that some of the models are very effective in a practical sense even without "guarantees" (Rumelhart & McClelland, 1986). Some others still feel, however, that the "successes" have only been realized with "toy" problems.

At this point, an attempt is made to put things in perspective.

1.5 Learning Networks

Information theory suggests that no system is capable of representing all possible mappings without an exponential amount of resources. The resources might take the form of number of nodes, number of connections, precision of the weights on connections, precision of timing, or precision in the functionality of nodes. If we want to learn an arbitrary mapping, we must pay with exponential resources. However, many of the theoretically possible mappings may not be of value in real world problems since many of the possible mappings are essentially "random". For the rest of the discussion, we will assume, therefore, that we are not starting with an exponentially sized, universal network.

We saw earlier that there are two major questions that can be asked about any learning network. Can the network represent the desired mapping? If yes, can it learn (converge to) the mapping? Each of these questions has subquestions. For example, in regard to the second question, is convergence guaranteed? If not, can we characterize when the system does not converge? We now look at several general types of adaptive (learning) networks as they relate to these questions.

The topology of a network can be represented by a graph where certain of the nodes receive the input vector and one or more nodes represent the output of the network. Many of the most well known neural net models start with a fixed topology and adapt only by modifying the weights on the links between nodes. Examples of this type of neural net are the perceptron, the back-propagation model, and the Boltzmann model. As has been mentioned, the perceptron learns by modifying a single layer of weights and can learn any mapping that it can represent, though it may have severe limitations on what it can represent. The back-propagation model (Rumelhart & McClelland, 1986) tries to achieve greater power using multiple layers of modifiable weights. Nevertheless, such a network may or may not be able to represent a specific mapping. And even when it can, the learning algorithm may be become stuck in a local "energy" minimum. (Nevertheless, back-propagation systems have found a number of significant uses.) The Boltzmann model (Hinton & Sejnowski, 1984) uses simulated annealing in an attempt to "shake" the model loose from local minima. However, there are still no guarantees that it finds the global minimum.

1.6 ASOCS

Some models change the topology of the network instead of changing weights. An example of this is the ASOCS (adaptive self-organizing concurrent systems) model (Martinez, 1986; Martinez, 1987; Martinez & Vidal, 1988). ASOCS assumes a pool of free nodes that can be acquired and placed in the network during adaptation. This model (as opposed to the previously discussed fixed topology models) is guaranteed to be able to learn any desired mapping (providing that enough free nodes are available and the appropriate connections can be made to place them in the appropriate part of the network). The nodes in specific ASOCS architectural models are more complex than in some neural nets, but ASOCS has the advantage of only requiring one pass through the learning set; some neural nets require many passes.

The ASOCS approach is based on an adaptive network of many simple computing elements which operate in a parallel asynchronous fashion. Both processing and control are distributed amongst the nodes. Figure 1.1 displays a high-level view of an ASOCS system. Rules are added incrementally and the system adapts to the changing rule base. Adaptation and data processing form two separate phases of operation. During processing the system acts as a parallel hardware circuit. The adaptation process is distributed amongst the computing elements and efficiently exploits parallelism.

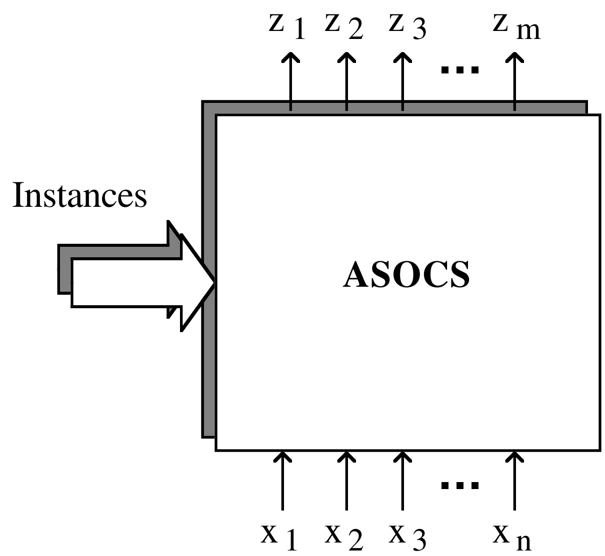


Figure 1.1 - Black box diagram

A number of different ASOCS models have been proposed, with three main classes having received the majority of research effort. These different models are labeled as adaptive algorithms and called AA1 (adaptive algorithm 1), AA2, and AA3 respectively. Although sharing the same functional goals, there mechanisms vary dramatically. We can, nevertheless, give a flavor for these architectures with one possible AA3 configuration shown in Figure 1.2.

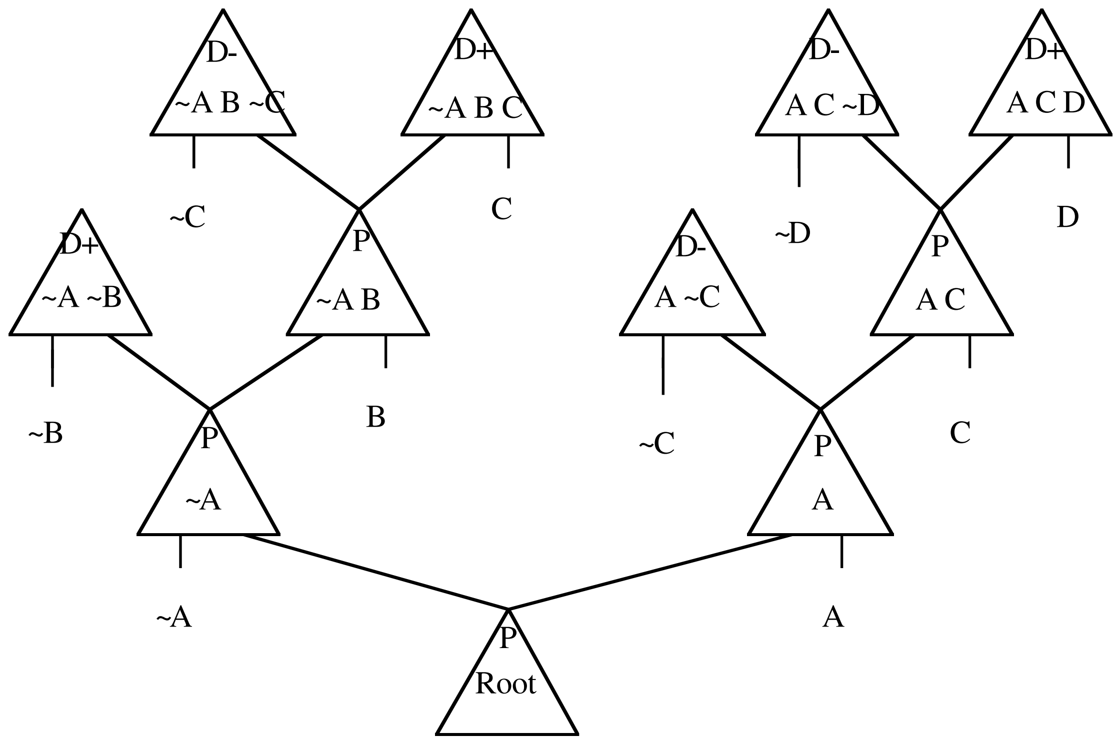


Figure 1.2 - Example network configuration

The figure shows both the hierarchichal and distributed characteristics of ASOCS architectures. We make no attempt to describe in detail how this network functions.

When new rules are added to the system, the system self-organizes in such a way as to remain consistent and optionally to achieve some degree of parsimony in the number of nodes. Consistency is maintained through a process called *discriminant variable addition* (DVA). By convention, new rules take precedence over older rules. If an old rule's output value conflicts with a new rule, and if both rules can match a given input, then DVA is performed. This consists of splitting the old rule (node) into a number of new rules (nodes), each of which contains at least one variable that discriminates it from the new rule. For example, if a new rule contains the variable v, then an old conflicting rule can be given the variable v'. This makes it impossible for both rules to match a given input. (The new Prioritized Rule Systems model elliminates the need for DVA with its use of priorites.)

We wish to emphasize one important point. Earlier we said that, as of yet, there are no guaranteed learning algorithms for multi-layer perceptrons where topology is fixed and only weights change.

ASOCS, with its modifiable topology, does provide learning guarantees in that it can both represent *and* learn all mappings.

This paper presents an extension of the ASOCS model that overcomes a number of weaknesses of ASOCS. The new model is called *Prioritized Rule Systems* .

1.7 High Level Overview of Prioritized Rule Systems

The following section presents a brief high level overview of the PRS model in order to give the reader an overall flavor of system processing and learning. How the system actually works, together with extensions to the simplistic model here shown, are the main thrust of the rest of this thesis.

A Prioritized Rule System is an ASOCS model which has the following attributes.

- A PRS processes inputs in the form of Boolean variables and output a Boolean result.
- A PRS is a set of rules specifying the mapping of inputs to output. For example, the rule (bcd',+) specifies that whenever b and c are both high and d is low, then the output should be high, whereas the rule (e,-) specifies that whenever e is high, the output should be low.
- A PRS can learn new rules over time.
- In a PRS, the input variables need not be predefined.
- A PRS can handle both specific and general rules. For example, (ab'cd'efgh,+) is very specific whereas (e,-) is very general.
- A PRS can automatically recognize and resolve rule conflicts. A conflict occurs when more than one rule matches a given input and the rules attempt to output different values.
- A PRS can automatically generalize in the sense of combining specific rules into more general rules where appropriate, thus using hardware more efficiently.
- Each rule in a PRS has an associated priority which helps to provide the capabilities mentioned above and others to be

mentioned later. For example, here is a representation of a full rule with associated priority: (ab'c,+,3).

We now give a brief example of a PRS architecture.

An implementation of a Prioritized Rule System can be thought of as a black box where inputs enter at one end and an output is produced at the other end.

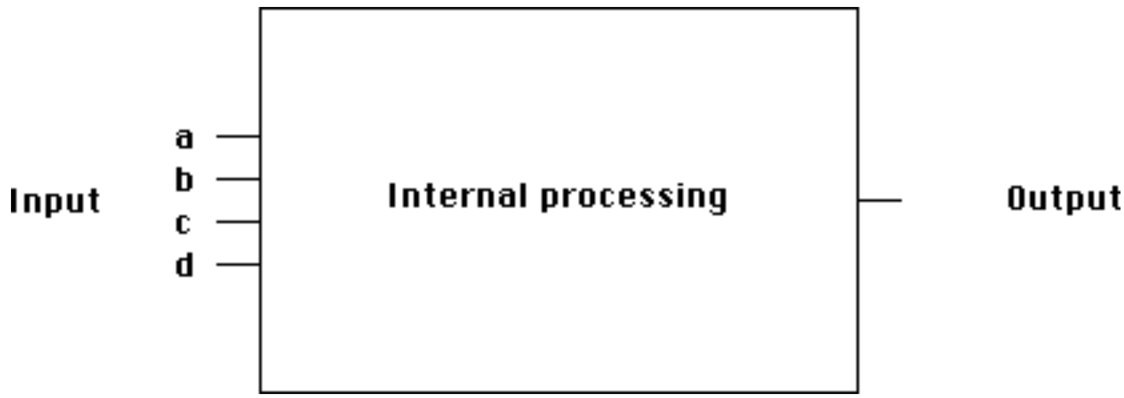


Figure 1.3 – PRS black box

Inside the black box are nodes (connected as a binary tree to provide logarithmic communication times). Each node represents a rule. Some nodes may match many different inputs (the represented rule is very general); other nodes may match few inputs (the represented rule is very specific).

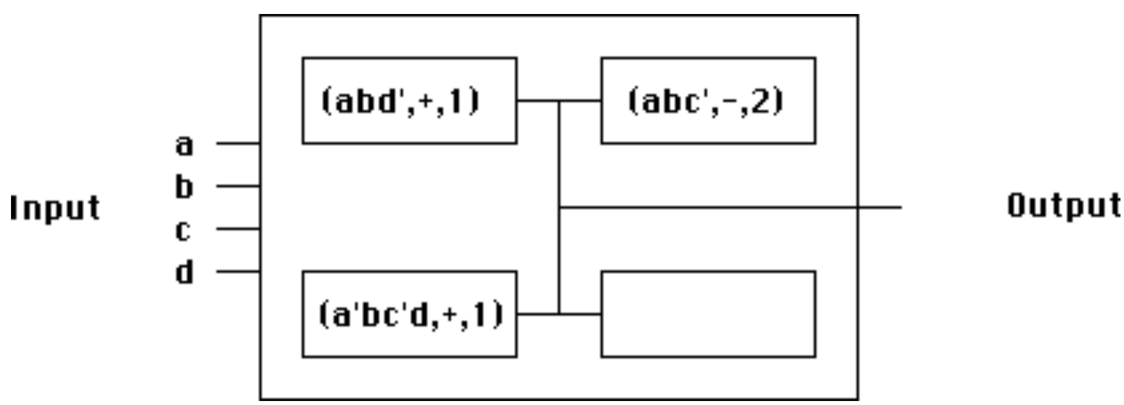


Figure 1.4 – The nodes represent rules

When an input is presented to the system, all matching nodes "raise their hands" and assert their associated output value. The node with highest priority "wins" and its output becomes the output of the system. In case of figure 1.5, the top two nodes match the input but the upper-right node wins.

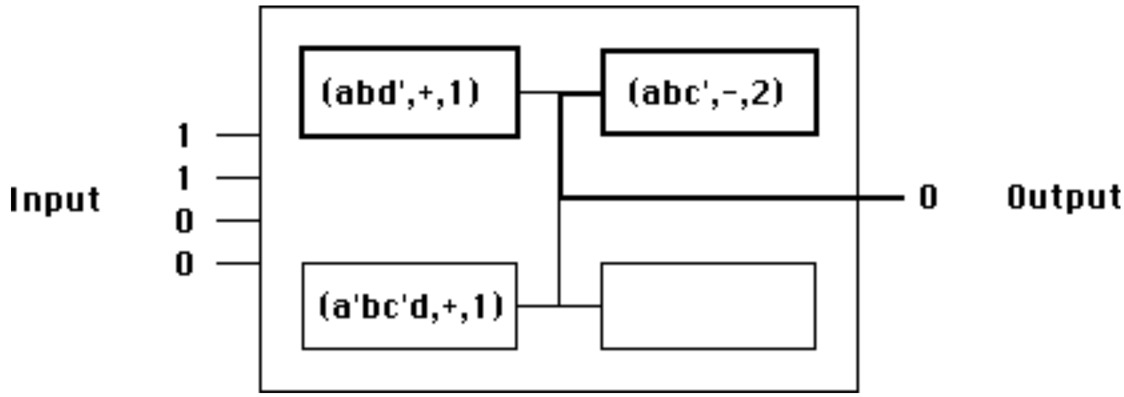


Figure 1.5 - Processing input

When a new rule is presented to the system for learning, it is given a higher priority than any other rule in the system, thus avoiding any possible conflicts. (For the present we choose that new rules take precedence over old rules.)

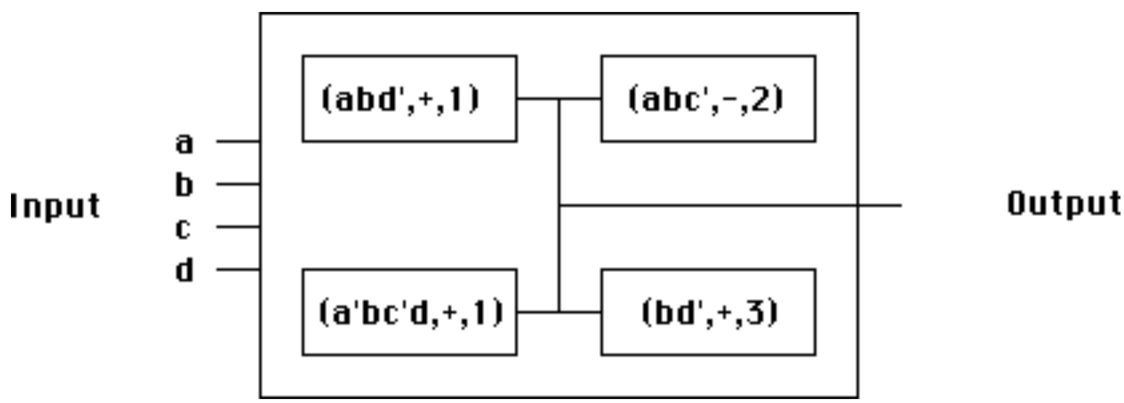


Figure 1.6 - Adding a new rule

After adding the new rule (bd',+,3) to figure 1.5 we produce figure 1.6. Each node (rule) independently and in parallel compares itself with the new rule and modifies itself accordingly. In the case of figure 1.6, the upper-left node deletes itself and the lower-left node makes itself more general producing figure 1.7.

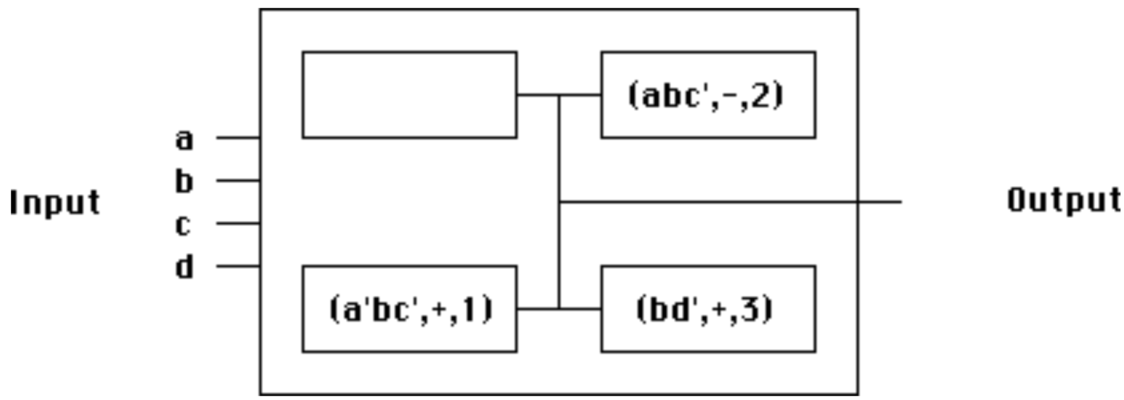


Figure 1.7 - After simplifying

We now conclude the list of attributes begun earlier:

- In a PRS, at most one node need be added to the system whenever a rule is learned.
- A PRS can learn any arbitrary set of valid rules.
- A PRS need only see each rule one time when learning a rule set.
- A PRS does not unlearn old rules when learning new rules, except in the case of conflict resolution.

This particular architectural instantiation has the following attributes:

- The PRS processes inputs in $O(log(n))$ time, where n is the number nodes in the network.
- The PRS learns individual rules in $O(log(n))$ time, where n is the number nodes in the network.

1.8 Advantages of Prioritized Rule Systems

Prioritized Rule Systems have the following advantages over ASOCS.

- The new model need never add more than one node per "instance". This follows from no longer requiring DVA expansion as in previous ASOCS models.

- The new model allows additional minimization strategies.

- The new model does not depend on the physical location of a node in the network.

- The new model uses nodes more "efficiently".

The new model retains the following strengths of other ASOCS models:

- It is capable of learning any Boolean function.

- It is capable of learning a rule set in a single pass.

- It is adaptable.

- It does not "unlearn" except in the case of conflict resolution.

- It processes inputs in log time.

- It learns in log time.

The new model has the following weaknesses compared to other ASOCS models:

- The individual nodes use more complex logic.

- The priority of each node must be stored, passed, and manipulated.

Chapter 2 discusses Boolean functions in preparation for a formal discussion of Prioritized Rule Systems. Chapter 3 provides the formal foundation of PRS, giving definitions, theorems, and an example of the use of the theorems. Chapter 4 describes an architectural model based on the theory of chapter 3. Chapter 5 discusses additional capabilities of the model. Chapter 6 describes a simulation of the architectural model. Chapter 7 compares PRS with ASOCS. Chapter 8 discusses extensions and future work. Chapter 9 provides a summary and conclusion.

Chapter 2

Prioritized Rule Systems
Informal Development

In this chapter we provide an informal introduction to the concepts that are formalized in chapter 3. The intent is to provide an intuitive feel for the material of chapter 3.

2.1 Partial Boolean Functions

Consider the following Boolean function:

$$f(x,y,z) = xy + y'z$$

The function f takes on the value 1 whenever either x and y are both 1 or whenever y is 0 and z is 1. We could write this function another way:

$$f(x,y,z) = \begin{cases} 1, & \text{when } xy = 1 \\ 1, & \text{when } y'z = 1 \\ 0, & \text{otherwise} \end{cases}$$

Suppose we specify a related function in the following way:

$$g(x,y,z) = \begin{cases} 1, & \text{when } xy = 1 \\ 1, & \text{when } y'z = 1 \\ ?, & \text{otherwise.} \end{cases}$$

The function g is undefined whenever the first two criteria are not met. We have switched from f, a total function, to g, a partial function. We may specify a third function:

$$h(x,y,z) = \begin{cases} 1, & \text{when } xy = 1 \\ 1, & \text{when } y'z = 1 \\ 0, & \text{when } x'z' = 1 \\ ?, & \text{otherwise.} \end{cases}$$

This specifies when the function h takes the value 0, when it takes the value 1, and that it is undefined under all other conditions. This function h can be expressed using a new notation:

xy +

$y'z$ +

$x'z'$ -

Each of the three lines above is referred to as a *rule*. Each can be thought of as an if-then rule. "If x and y are both 1, output a 1." "If y is 0 and z is 1, output a 1." "If x and z are both 0, output a 0." If none of the rules apply, the output is undefined. The set of all such rules at any given time is called a *rule set.* The rule set, then, represents a (partial) Boolean function.

2.2 Rule Systems

We would like our rule system to have the following attributes:

- We would like to modify the function from time to time by adding new rules.
- We would like the modified rule set to be consistent, that is, we don't want two rules to simultaneously output different values.

- We would prefer not to have many rules when fewer would do.
- We would like to embed this system in hardware.
- We would like both the processing and learning modes to be fast.

The rule system should be adaptive. That is, at any future time we can add another rule to the rule set, such as:

xy +

y'z +

x'z' -

z +

The rule system should allow for the removal of redundancy. In the previous example, the second rule, y'z +, is subsumed by the fourth rule, z +, and would, therefore, be removed producing the rule set:

xy +

x'z' -

z +

Conflicts can arise when adding new rules such as x'y'z -:

xy +

x'z' -

z +

x'y'z -

What value should h take on when x and y are both 0 and z is 1? The new (fourth) rule (x'y'z -) says that h should take on the value 0. But the third rule (z +) says that whenever z is 1 that h should take on the value 1. This issue is resolved in our rule systems by assuming that new rules always override old rules. The rule set is modified to reflect this assumption.

We first introduce the concept of a "discriminant variable". When considering a pair of rules, if there is a variable that is contained in both rules, and which is complemented in one of the rules but not in the other, then it is called a discriminant variable. It should be clear that such a variable makes it impossible for both rules to be "activated" at one time and, therefore, no conflict can occur. In our example above, the third rule (x'y'z -) and fourth rule (z +) are not discriminated and they attempt to set h to opposite values. Hence the conflict.

We remove a conflict by introducing discriminant variables into the old rule. In our example, the fourth rule (x'y'z -) contains the variables x and y that are not found in the third rule (z +). We therefore replace the third rule (z'y'z -) with two new rules each containing a variable that discriminates it from the fourth rule (z +). That is, we replace "z +" with both "xz +" and "yz +" as follows:

xy +

x'z' -

xz +

yz +

x'y'z -

If we were dealing with a system containing 100 variables, we might be required to replace a rule with 99 new rules. We call this the problem "DVA expansion". DVA stands for "discriminant variable addition".

At this point we introduce the notion of concordant and discordant rules. A rule that causes a function to take on the value 1 is called a "positive" rule. A rule that causes a function to take on the value 0 is called a "negative" rule. Two rules are "concordant" if they are both positive or both negative. Two rules are "discordant" if one is positive and the other is negative. Conflict can only occur between discordant rules.

There are actually several different types of redundancy reduction that can take place. Redundancy reduction only occurs between concordant rules in the standard Rule System model. Prioritized Rule Systems make it possible to also do redundancy reduction between discordant rules. Prioritized Rule Systems also do away with the need to handle DVA (discriminant variable addition) in order to keep the rule set consistent.

2.3 Prioritized Rule Systems

Under the Prioritized Rule System model, each time a new rule is added to the rule set, it is given a priority. For the moment, assume that each new rule is given a priority higher than all of the old rules and that whenever two rules conflict, the rule with the highest priority wins.

Consider the previous example with the following priorities:

xy + 1

x'z' - 2

z + 3

x'y'z - 4

Under the regular Rule System model, the third rule (z +) had to be replaced with two other rules using DVA. Under the Prioritized Rule System model, this is not necessary since the new rule (x'y'z -) has higher priority than the old conflicting rule. An alternative priority assignment is possible. A new rule need only be given a priority higher than the highest conflicting rule. The above rule set becomes:

xy + 1

x'z' - 1

z + 1

x'y'z - 2

We give another example of the added flexibility provided by Prioritized Rule Systems. Suppose we were to add the rule xy'z' +. It does not conflict with any discordant rules, so it is given priority 1.

xy + 1

x'z' - 1

z + 1

x'y'z - 2

xy'z' + 1

Using theorems proved in the next chapter, we could change the priority of the second rule (x'z' - 1) in the rule set to 2. We could then combine all of the positive rules into one single rule with priority 1 giving the following rule set:

x'z' - 2

x'y'z - 2

+ 1

The third rule in this new rule set (+ 1), having no variables at all, matches any input and tries to output a 1. However, if an input is also matched by either of the other two rules, then the third rule is overpowered by the greater priority of the first two rules, and the system outputs a 0.

This last simplification of the rule set may seem like magic to the reader. The next chapter should bring clarification.

Chapter 3

Prioritized Rule Systems Formal Development

3.1 Definitions

A *variable* represents a value. In our discussion, variables will represent Boolean values. We use lowercase letters to represent variables. Variables can take on the values 0 and 1. A variable can be complemented by following the variable with a single quote. For example, b' is the complement of b. When b has the value 0, b' has the value 1 and vice versa. We assume that the variables are taken from some fixed finite set of variables.

A *variable list* is a (possibly empty) set of distinct variables from the variable set (complemented or not). We usually write a variable list as the concatenation of its variables, for example: ab'c, bd'f'gh, c'd', a. Although a list is shown as a sequence of variables, the reader should remember that variable lists are sets.

An *input* is a set of Boolean values each associated with a variable in the variable set. For example: {(a,1),(b,0),(c,0),(d,1)}. If we accept a certain ordering of the variables, the input can be represented by a bit string. If, for example, we accept the standard alphabetic ordering of the variables, the input for the previous example can be written 1001.

A variable list *matches* an input if each uncomplemented variable in the list is matched by a 1 in the input and each complemented variable is matched by a 0 in the input. For example, a'b over the variable set {a,b,c,d} matches the inputs 0100, 0101, 0110, 0111, but it does not match any input beginning with 00, 10, or 11. Note that an empty variable list matches any input.

A *rule* is a variable list together with an associated polarity. The *polarity* (either *positive* or *negative*) of a rule represents the output of the rule and is written as a '+' or a '-'. A rule, then, is an ordered pair, the first element being a variable list, the second element being the polarity: (ab'c,+), (a',-). If the variable list of a rule matches an input, the rule outputs a 1 (if it's polarity is '+') or a 0 (if it's polarity is '-'). A rule matches an input if its associated variable list matches the input. When we want to indicate the variable list of a rule R, we write var(R); when we want to indicate the polarity of a rule R, we write po(R).

Two rules are *concordant* if they have the same polarity, otherwise, they are *discordant*. Two discordant rules are said to *conflict* if there exists at least one input that is matched by both rules.

The *matching set* of a variable list or rule is the set of all inputs that are matched by the variable list or rule. If R is a rule, then MS(R) represents the matching set of R. Rule R is *subsumed* by rule S (written R ≤ S) if MS(R) is a subset of MS(S). R is *properly subsumed* by S (written R < S) if MS(R) is a proper subset of MS(S). We write R = S if MS(R) = MS(S). Similar definitions apply to ≥, >, and ≠. Examples: (ab',+) < (a,+), (a'c'd'ef,-) < (a'd'e,+) < (d'e,+) < (e,-) ≤ (e,+). In the sequel, we will find the concept of subsumation to be a handy one. We will often be interested in pairs of rules where the variable list of one rule is a subset or superset of the variable list of the other rule, or where this is true except for a single variable that is not complemented in one of the variable lists but which is complemented in the other, for example (abf,+) and (abcdf',+).

Given a rule R, we write Rv to indicate the rule derived from R by adding to the variable list of R the variable represented by v. The variable v may be complemented or uncomplemented. Formally, Rv = (var(R) U {v}, po(R)).

A *rule set* is a set of rules. A rule set is *consistent* if, for every possible input, no two discordant rules can match the same input.

A *Rule System* is a consistent rule set. A Rule System can:

a. process inputs to produce an output.

b. incrementally add new rules.

When a Rule System is presented with an input, it outputs 0, 1, or '?'. It outputs a 0 if one or more rules with polarity '-' matches the input; it outputs a 1 if one or more rules with polarity '+' matches the input; it outputs a '?' if no rule matches the input.

(Since a rule set is consistent, rules of opposite polarity cannot match the same input.)

A Rule System represents a partial Boolean function.

A few comments need to be made with regards to the previous statement:

A. One might wonder what it means when the partial Boolean function represented by a Rule System is undefined. It means that, in terms of a Rule System, the Rule System has not been given enough information to decide the output (that is, in neural net terms, how the input should be classified). When the Rule System outputs a '?', it says "I don't know for sure", rather than "undefined" in the sense that division by zero is undefined.

B. The input to a Rule System occurs at one point in time and the output of the Rule System occurs at some later point in time. The usual concept of a mathematical function does not include this concept of "temporal logic".

When a rule is added to a Rule System, it takes precedence over all previous rules. (This is part of the definition of a Rule System and may be considered an arbitrary choice. Other types of Rule Systems may be defined differently.) If the rule conflicts with any old rule, the rule set must be modified in order to remain consistent. The Rule System may also wish to attempt certain types of reduction on the rule set.

A Rule System is *minimal* if no Rule System representing the same Boolean function has fewer rules. A Rule System that is not minimal can be *reduced* or *simplified* in the sense that its rule set can be replaced with another rule set containing fewer rules without changing the represented function. When implementing a real system, parsimony is a desirable goal, and we often replace two nodes with one, or free a redundant node. However, maintaining global minimality is a difficult problem (it is NP-complete).

One rule is *simpler* or *more general* than another if its variable list is a subset of that of the other rule. For example, (a,+) is simpler than (ab',+). We are interested in making rules simpler. For one thing, it may allow reductions at some future time. For another thing, greater generality may aid in the understanding of a particular system. However, there may be reasons for not attempting to keep every rule as simple as possible.

A *prioritized rule* is a 3-tuple, the first element being a variable list, the second element being a polarity, and the third element being a priority. A *prioritized rule set* is simply a set of prioritized rules. Here is an example of a prioritized rule set:

(a',+,3)

(ab'df,-,2)

(bcg,+,5)

When we want to indicate the priority of a rule R, we will usually write pr(R).

Given a specific input, let the priority of the highest-priority matching rule(s) be p. A prioritized rule set is consistent if, for every possible input, no two conflicting rules have the same highest priority p (the highest priority of matching rules).

A *Prioritized Rule System* (PRS) is a Rule System that uses a consistent prioritized rule set rather than a consistent regular rule set A PRS outputs a 0 if the polarity of the highest-priority matching rule(s) is negative; it outputs a 1 if the polarity of the highest-priority matching rule(s) is positive; it outputs a '?' if no rule matches the input.

We wish to emphasize the difference between a Rule System and a Prioritized Rule System. The rule set of a Rule System cannot contain conflicting rules, that is, it cannot contain discordant rules that can match the same input. The prioritized rule set of a Prioritized Rule System may have conflicting rules. However, the prioritized rule set cannot contain conflicting rules which, for some specific input, have the same priority and which priority is greater than or equal to the priority of any other matching rule. This requirement allows the output of the system to be well defined.

When there is no chance of confusion, we will refer to a prioritized rule simply as a rule and a prioritized rule set simply as a rule set.

The rule set of a PRS can be non-minimal, but it cannot be inconsistent. In section 3.2 we develop theorems that permit manipulation the rule set of a PRS while maintaining the consistency of the rule set.

The partial Boolean function represented by a PRS is called *the PRS function*.

A rule R is *covered* by a set C if for every input J matched by R, there is a rule S in C with pr(S) ≥ pr(R) that also matches J. If R is covered by C, we write R \ C. If a rule R is covered by the rule set of the PRS, then it is redundant. (See Theorem 7.)

A rule R is *strictly covered* by a set C if for every input J matched by R, there is a rule S in C with pr(S) > pr(R) that also matches J. If R is strictly covered by C, we write R \\ C. If a rule R is strictly covered by the rule set of the PRS, then it is redundant. (See Theorem 7.)

A rule R is (strictly) covered by a PRS if R is (strictly) covered by some subset of the rule set of the PRS.

3.2 Theorems

We now prove seventeen theorems for Prioritized Rule Systems. These theorems are used in chapter 4 for designing a PRS architecture and associated processing and learning algorithms.

Theorems 1 through 6 are preliminary theorems. Theorem 7 tells which rules may be removed from a PRS without changing the PRS function. Theorems 8 and 9 tell which rules may be added to a PRS. Theorems 10 through 12 provide replacement capabilities. Theorems 13 through 17 provide techniques for changing the priorities of existing rules.

In the following discussion, a statement of the form "a PRS with rule R" or "a PRS containing rule R" means that R is one of the rules contained in the rule set of the PRS, and that the rule set may contain other rules. When we say that a PRS function has not changed, we also mean that the rule set of the PRS is still consistent.

Theorem 1: Rule Rv is subsumed by rule R

In other words: $Rv \leq R$.

Proof: Any input whose appropriate bits are matched by the variables in Rv are also matched by the variables in R since the variables in R are just a subset of the variables in Rv. Therefore, by the definition of subsumation, Rv is subsumed by R.

Example: (a'bc,+,3) < (a'bcd',-,7)

Theorem 2: If rule R is strictly covered by a set C, then R is covered by C.

In other words: If R \\ C, then R \ C.

Proof: Since every element in C has priority > R, every element of C has priority ≥ R.

Theorem 3: If rule R is subsumed by rule S and pr(R) ≤ pr(S), then R is covered by any set containing S.

In other words: If R ≤ S and pr(R) ≤ pr(S), then R \ ({S} U W) for any set W.

Proof: Since R is subsumed by S, every input that matches R also matches S. Since pr(R) ≤ pr(S), R is covered by the set {S}. Therefore, R is covered by any set containing S.

Example: (a'bc,+,3) is covered by {(a'b,-,3)}.

Theorem 4: If rule R is subsumed by rule S and pr(R) < pr(S), then R is strictly covered by any set containing S.

In other words: If R ≤ S and pr(R) < pr(S), then R \\ ({S} U W) for any set W.

Proof: Identical to the proof of Theorem 3.

Example: (a'bc,+,3) is strictly covered by {(a'b,-,4)}.

Theorem 5: If pr(R) ≤ pr(Rv) and pr(R) ≤ pr(Rv'), then rule R is covered by any set containing Rv and Rv'.

In other words: If pr(R) ≤ pr(Rv) and pr(R) ≤ pr(Rv'), then for any set W, R \ ({Rv,Rv'} U W).

Proof: Any input matched by R is matched by either Rv or Rv'. Since pr(R) ≤ pr(Rv) and pr(R) ≤ pr(Rv'), R is covered by the set {Rv,Rv'}. It is therefore covered by any set containing Rv and Rv'.

Example: (ab,+,1) is covered by {(abc,-,1),(abc',+,2)}.

Theorem 6: If pr(R) < pr(Rv) and pr(R) < pr(Rv'), then rule R is strictly covered by any set containing Rv and Rv'.

In other words: If pr(R) < pr(Rv') and pr(R) < pr(Rv'), then for any set W, R \\ ({Rv,Rv'} U W).

Proof: Similar to the proof of Theorem 5.

Example: (ab,+,1) is strictly covered by {(abc,-,3),(abc',+,2)}.

Theorem 7: If R is covered by a PRS, then removing R from the PRS does not change the PRS function.

In other words: If R \ PRS, then PRS = PRS - {R}.

Proof: If R is strictly covered by the PRS, then by the definition of a covering, every input that matches R is also matched by some higher priority rule. Since R cannot effect the final output of the PRS, it can be removed.

If R cannot be strictly covered, but can be covered, then there is some input matched both by R and by some other rule S in the PRS of priority pr(R), and no other higher priority rule matches that input. The polarity of S must be equal to the priority of R, otherwise, the PRS would be inconsistent. Therefore, R has no effect on the output of the PRS that is not already covered by at least one other rule. Consequently, R can be removed from the PRS.

Example: Given a PRS with the rule set {(ab,+,1), (abc,+,1), (abd,-,2), (abc',-,2)}, the first rule may be dropped.

Corollary 7.1: If R is subsumed by some S in a PRS and pr(R) ≤ pr(S), then removing R from the PRS does not change the PRS function.

In other words: If there is an S in a PRS with R ≤ S and pr(R) ≤ pr(S), then PRS = PRS - {R}.

Proof: Since R ≤ S and pr(R) ≤ pr(S), then by Theorem 3, the rule R is covered by the PRS which contains S. By Theorem 7, removing R does not change the PRS. ˙

Example: (abc,+,1) may be removed from {(abc,+,1), (ab,+,1)}.

Corollary 7.2: Let a PRS contain rules R, Rv, and Rv'. If pr(R) ≤ pr(Rv) and pr(R) ≤ pr(Rv'), then removing R from the PRS does not change the PRS function.

In other words: If R, Rv and Rv' are in a PRS with pr(R) ≤ pr(Rv) and pr(R) ≤ pr(Rv'), then PRS = PRS - {R}.

Proof: By Theorem 5, R is covered by the PRS containing Rv and Rv'. By Theorem 7, removing R does not change the PRS. ˙

Theorem 8: If R is strictly covered by a PRS, then adding R to the PRS does not change the PRS function.

In other words: If R \\ PRS, then PRS = PRS U {R}.

Proof: By the definition of a strict covering, if J is an input matched by R, then there is a rule S in the PRS with pr(S) ≥ pr(R) that also matches J. Its higher priority overrules any output from R. Adding R to the PRS cannot change the output of the PRS. ˙

Example: (ab,+,1) may be added to {(abc,+,3), (abd,-,2), (abc',-,2)}.

Corollary 8.1: If R is subsumed by some S in a PRS and pr(R) < pr(S), then adding R to the PRS does not change the PRS function.

In other words: If for some S in a PRS, R ≤ S and pr(R) < pr(S), then PRS = PRS U {R}.

Proof: By Theorem 4, the rule R is strictly covered by the PRS. By Theorem 8, R can be added without changing the PRS function. ·

Example: (ab,+,1) may be added to {(a,-,2)}.

Corollary 8.2: Let a PRS contain the rules R, Rv, and Rv'. If pr(R) < pr(Rv) and pr(R) < pr(Rv'), then adding R to the PRS does not change the PRS function.

In other words: If R, Rv and Rv' are in a PRS with pr(R) < pr(Rv) and pr(R) < pr(Rv'), then PRS = PRS U {R}.

Proof: By Theorem 6, the rule R is strictly covered by the PRS. By Theorem 8, R can be added to the PRS without changing the PRS function. ·

Example: (a,+,1) may be added to {(ab,-,2), (ab',+,3)}.

Theorem 9: Let R be covered by a PRS. If there is a covering set C in the PRS such that for every rule S in C either pr(S) > pr(C) or po(S) = po(R), then adding R to the PRS does not change the PRS function.

In other words: If there exists a set C (subset of PRS) such that R \ C and for every S in C either pr(R) < pr(S) or po(R) = po(S), then PRS = PRS U R.

Proof: If an input matched by R is matched by a rule of higher priority, then the output of R cannot effect the output of the PRS. If an input matched by R is matched by a rule of the same polarity, then the output of R does not change the output of the PRS. ·

Example: (ab,+,1) may be added to {(abc,+,1), (abd,-,2), (abc',-,2)}.

Corollary 9.1: Let R be subsumed by some S in a PRS, if a) pr(R) < pr(S) or b) both pr(R) = pr(S) and po(R) = po(S), then adding R to the PRS does not change the PRS function.

In other words: If R ≤ S for some S in a PRS and either pr(R) < pr(S) or both pr(R) = pr(S) and po(R) = po(S), then PRS = PRS U {R}.

Proof: By Theorem 3, the rule R is covered by the PRS. By Theorem 9, R can be added to the PRS without changing the PRS function.

Examples: (ab,+,2) may be added to {(a,+,2)} or to {(ab,-,3)}.

Corollary 9.2: Let a PRS contain rules Rv and Rv', then adding R with either pr(R) < pr(Rv) or po(R) = po(Rv) and with either pr(R) < pr(Rv') or po(R) = po(Rv') to the PRS does not change the PRS function.

In other words: If Rv and Rv' are in a PRS, given an R with either pr(R) < pr(Rv) or po(R) = po(Rv) and with either pr(R) < pr(Rv') or po(R) = po(Rv'), then PRS = PRS U {R}.

Proof: By Theorem 5, the rule R is covered by the PRS. By Theorem 9, R can be added to the PRS without changing the PRS function.

Example: (a,+,2) may be added to {(ab,+,2), (ab',-,3)}.

Theorem 10: Let a PRS contain rules Rv and Rv' of the same polarity and priority. If both Rv and Rv' are replaced with R of polarity po(Rv) and priority pr(Rv), then the PRS function is unchanged.

In other words: If Rv and Rv' are in a PRS and if pr(R) = pr(Rv) = pr(Rv') and po(R) = po(Rv) = po(Rv'), then PRS = PRS U {R} - {Rv,Rv'}.

Proof: This is merely an application of Corollary 9.2 followed by Corollary 7.1.

Example: {(ab,+,2), (ab',+,2)} = {(a,+,2)}.

Theorem 11: Let a PRS contain rules Rv and RVv' satisfying pr(Rv) > pr(RVv'). Replacing RVv' with RV having the same polarity and priority as RVv' does not change the PRS function.

In other words: If Rv and RVv' are in a PRS and pr(Rv) > pr(RVv'), then PRS = PRS U {RV} - {RVv'}, where po(RV) = po(RVv') and pr(RV) = pr(RVv').

Proof: Corollary 8.1 is applied enough times to produce the rule RVv with the same polarity and priority as RVv'. Then Theorem 10 is applied to produce the desired rule RV. ▪

Example: {(ab,+,2), (ab'cde,-,1)} = {(ab,+,2), (acde,-,1)}.

Theorem 12: Let a PRS contain rules Rv and RVv' with pr(Rv) ≥ pr(RVv') and po(Rv) = po(RVv'). Replacing RVv' with RV having the same polarity and priority as RVv' does not change the PRS function.

In other words: If Rv and RVv' are in a PRS and pr(Rv) ≥ pr(RVv') and po(Rv) = po(RVv'), then PRS = PRS U {RV} - {RVv'}, where po(RV) = po(RVv') and pr(RV) = pr(RVv').

Proof: Similar to the proof of Theorem 11. ▪

Example: {(ab,+,2), (ab'cde,+,2)} = {(ab,+,2), (acde,+,2)}.

The next four theorems use these definitions: Let R be a rule with priority p in some PRS. Let Q be the set of all of the priorities of all of the rules in the PRS that conflict with R.

Theorem 13: If Q = Ø, then changing the priority of R does not change the PRS function.

Proof: Since no rule conflicts with R, no input matched by R can also be matched by a rule of opposite polarity. Changing the priority of R may cause some other rule of the same polarity to "win". Nevertheless, the output of the system is the same and therefore the PRS function has not changed. ▪

Theorem 14: Let $Q \neq Ø$ and let q be the largest element of Q. If $p > q$, then changing the priority of R from p to any value greater than q does not change the PRS function.

Proof: Since the priority of R is greater than that of any conflicting rule, R always wins any contest involving a conflicting rule. R will continue to win as long as the priority of R remains higher than any conflicting rule. Changing the priority of R may cause some other rule of the same polarity to win for some input. Nevertheless, the winner will output the same polarity as R, and therefore, the PRS function does not change. ▪

————————————————

Theorem 15: Let $Q \neq Ø$ and let q be the smallest element of Q. If $p < q$, then changing the priority of R from p to any value smaller than q does not change the PRS function.

Proof: Similar to the proof of Theorem 14. ▪

————————————————

The following theorem applies when none of the previous three apply, that is, when p is sandwiched between two elements of Q.

————————————————

Theorem 16: Let $Q \neq Ø$ and let q1 be the smallest element of Q and q2 be the largest element of Q. If $q1 < p < q2$, then changing the priority of R from p to any value between q1 and q2 does not change the PRS function.

Proof: Similar to the proof of Theorem 14. ▪

————————————————

Theorem 17: Given a PRS and integers k and m, let Q1 be the set of all rules of the PRS with priorities $\geq$ m. Adding k to the priorities of all of the rules of Q1 does not change the PRS function. Let Q2 be the set of all rule of the PRS with priorities $\leq$ m. Subtracting k from the priorities of all of the rules of Q2 does not change the PRS function.

Proof: Consider any two rules with priorities p and q. If $p = q$ before the change specified in the theorem, then they will be obviously be equal after the change. Likewise, if $p < q$ before the change, then $p < q$ after the change. And if $p > q$ before the change, then $p > q$ after the change. So the ultimate winner for any particular input will not change.

3.3 Using the Theorems

We use the previous theorems to manipulate the rule set of a PRS. For convenience, we write the rules without parentheses and commas. We begin with an empty rule set and add the following rules (which have no priority until entered into the rule set).

abc +
a'bcd' -
abcd -
ab'c +
a'bcd -
a'b' -

We adopt the following conventions:

- We always give a rule a priority that is a positive integer.
- Since the definition of a PRS requires that new rules have precedence over old rules, each new rule added to the PRS is initially given a priority one higher than that of any other rule in the PRS.
- We number rules in the rule set for reference purposes.

First, we add the rule "abc +" giving it a priority of 1.

(1) abc + 1

Next, we add "a'bcd' -" giving it a priority of 2.

(1) abc + 1
(2) a'bcd' - 2

Since the two rules do not conflict, we will use Theorem 13 to reduce the priority of rule 2 to 1.

(1) abc + 1
(2) a'bcd' - 1

Now we add "abcd -".

(1) abc + 1
(2) a'bcd' - 1
(3) abcd - 2

Since the new rule conflicts with rule 1, we can't reduce its priority. We now add "ab'c +".

(1) abc + 1
(2) a'bcd' - 1
(3) abcd - 2
(4) ab'c + 3

The new rule does not conflict with any other rule and we reduce its priority to 1.

(1) abc + 1
(2) a'bcd' - 1
(3) abcd - 2
(4) ab'c + 1

Using Theorem 10, we replace rules 1 and 4 with "ac + 1".

(2) a'bcd' - 1
(3) abcd - 2
(5) ac + 1

Next, we add "a'bcd -", immediately reducing its priority to 1.

(2) a'bcd' - 1
(3) abcd - 2
(5) ac + 1
(6) a'bcd - 1

We notice that rules 2 and 6 can be replaced with "a'bc - 1". We also notice that if rules 3 and 6 had the same priority, we could replace them with a rule of the form "acd -". Therefore, we cleverly use Corollary 9.1 to duplicate rule 6.

(2) a'bcd' - 1
(3) abcd - 2
(5) ac + 1
(6) a'bcd - 1
(7) a'bcd - 1

We replace rules 2 and 6 as mentioned before.

(3) abcd - 2
(5) ac + 1
(7) a'bcd - 1
(8) a'bc - 1

Using Theorem 13 we increment the priority of rule 7.

(3) abcd - 2
(5) ac + 1
(7) a'bcd - 2
(8) a'bc - 1

And now we combine rules 3 and 7.

(5) ac + 1
(8) a'bc - 1
(9) bcd - 2

Finally, we add "a'b' -" immediately lowering its priority to 1.

(5) ac + 1
(8) a'bc - 1
(9) bcd - 2
(10) a'b' - 1

We use Theorem 12 with rules 8 and 10 to replace rule 8 with "a'c - 1".

(5) ac + 1
(9) bcd - 2
(10) a'b' - 1
(12) a'c - 1

The addition of the rules is now complete.

Chapter 4

A PRS Architectural Model

In this section we consider an architectural model for Prioritized Rule Systems together with associated processing and learning algorithms. This is not done to give an exact description of a real implementation, but rather to show that such an implementation is possible. In presenting this model, we will not attempt to make it as efficient as possible; we only want to give the reader a feel for what can be done.

4.1 The Architecture

Consider the tree network structure shown in figure 4.1. Data and control information can flow both up and down the tree. Each rule in the rule set requires one node in the tree. Its position in the tree is immaterial. Such nodes are considered "active/in use". Other nodes, not yet representing rules, are considered "inactive/free". When a new rule needs to be added to the rule set, it acquires a free node which then becomes active.

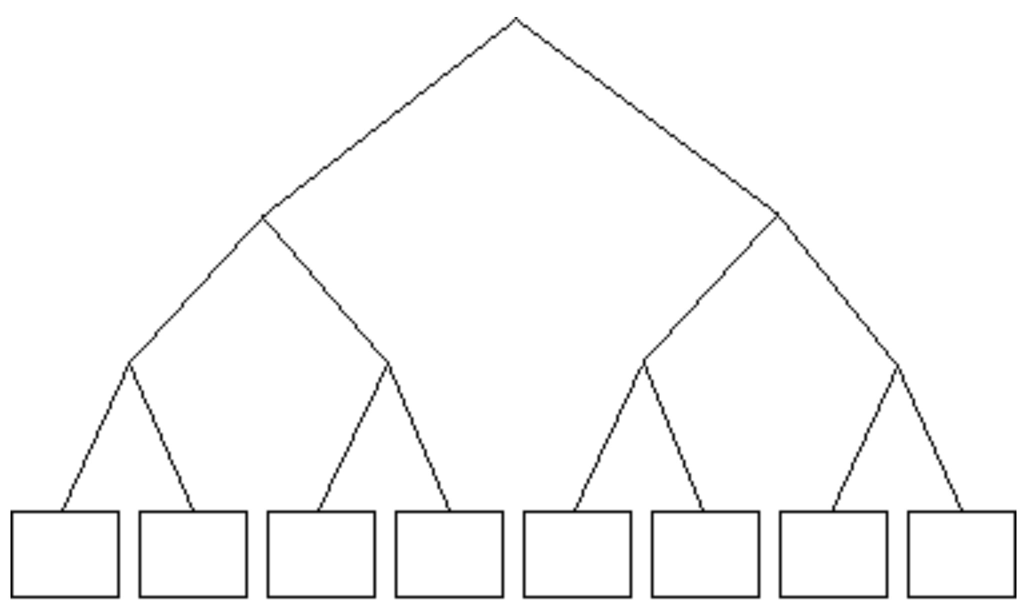


Figure 4.1

Each node contains several fields (see figure 4.2). One such field contains the priority of the node. If the node is free, it has a priority of zero; otherwise, it has a value greater than zero.

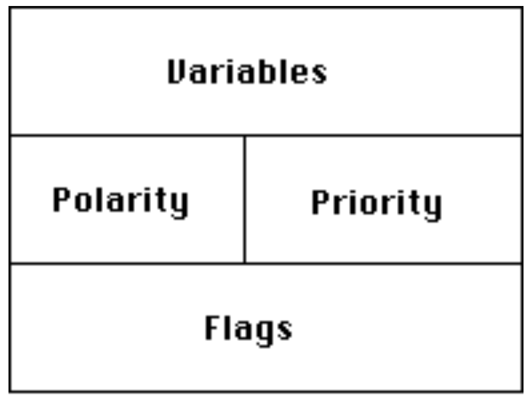


Figure 4.2

Each node has access to all of the variables in the environment (see figure 4.3). The value of each variable can be accepted as is, complemented, or ignored. The accepted variables (whether complemented or not) are "anded" together to produce the output value of the associated rule.

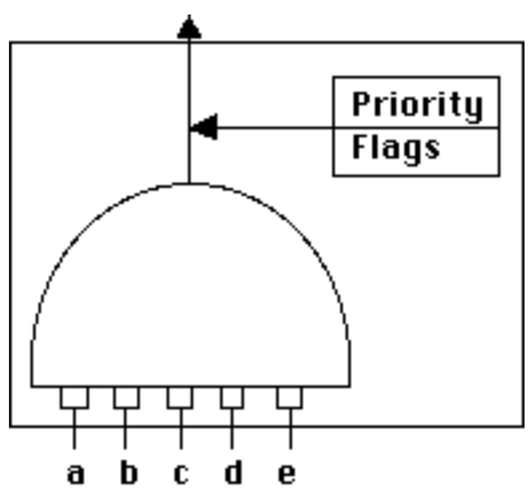


Figure 4.3

Various control signals can be broadcast to the nodes. All control signals and data are sent to every node. It is up to the individual nodes to respond in an appropriate manner.

Nodes can respond to control signals by modifying themselves and/or passing information up the tree. As an example, the controller may send a signal to the network asking all nodes that meet certain criteria to raise their hands. Each node that meets the criteria will set its hand-raised flag and then pass a message up the tree. The message consists of its priority and polarity (output). At each junction of two sub-trees, the message with higher priority is chosen and sent further up the tree; the losing message is thrown in the bit bucket.

The flag bits in each node are defined as follows:

Symbol	Meaning	Field
F	Free node	-
*	Hand raised	-
+	Concordant	1
-	Discordant	1
0	No discriminant variables	2
1	One discriminant variable	2
2	More than one discriminant variable	2
=	Equal	3
<	Old rule subsumed by new rule	3
>	New rule subsumed by old rule	3
≠	None of the previous three	3

4.2 Presenting a New Rule to the System

When a new rule is presented to the system, each node is instructed to set its flag bits in fields one through three according to the previous table. For example, suppose a particular active node N represents the rule (ab'c,-,3). If (b',+,4) is presented to the system, node N will set flags "-0<" to indicate that 1) it is discordant with the new rule, 2) it has no discriminant variable with respect to the new rule, and 3) it is subsumed by the new rule.

Notice that each active node sets exactly one flag bit in each of the three fields.

A flag code can be sent by the controller to each node in the network for matching purposes. A flag code consists of any combinations of bits in any of the fields. Here is one possible flag code: +-0<=. A node matches a flag code if each bit in each field of the node matches a corresponding bit in the flag code. For example, a node with flags "-0<" will match the flag code "+-0<=", but a node with flags "+1>" will not.

Note: There is one other special flag code. It consists of the flag F. All inactive (free) nodes match this flag code.

4.3 Raising Hands

One of the other functions that can be executed by the controller is called RaiseHands. The controller broadcasts a flag code to all nodes of the network instructing all matching nodes to raise their hands (set their HandRaised flag). This prepares the network for various other functions.

An example of another function is the Delete function. This function instructs all nodes that have their hands raised to delete themselves, that is, to make themselves inactive/free. The delete function provides a way of removing redundant nodes.

Before discussing how the system learns, we explain how it processes an input.

4.4 Processing Input

An input is presented to the system as a maximum sized variable list and arbitrary priority with every possible variable specified either in uncomplemented or complemented form. The polarity of a rule is needed only during learning, so we set the polarity to any value that is convenient. All nodes are instructed to set their flags. Next, the RaiseHands function is executed with the following flag code: +-0=>. All matching nodes raise their hands and the highest priority message bubbles its way to the top of the tree. The polarity of the winning message represents the output of the system; if the polarity is "+", the system outputs a 1, otherwise, it outputs a 0. If no node has raised its hand, the system outputs a "?".

4.5 Learning

We now present a number of functions that can be used in learning a prioritized rule set

4.5.1 Adding a New Rule

The simplest way to learn a new rule is to add it to the system with priority higher than that of any conflicting rule. The determination of the appropriate priority can be done in the following way (using C-like pseudo-code):

```
Message = RaiseHands("-0=<>≠");

NRPriority = Message.Priority + 1;
```

Theorem 14 let's us determine the lowest valid priority for the new rule. We find a free node and activate it with the appropriate initialization:

```
RaiseHands("F");

SelectSingleNode();

ActivateNode(NRPriority);
```

RaiseHands causes all free nodes to raise their hands. SelectSingleNode causes all but one of the nodes with their hands raised to lower their hands in the following fashion. The appropriate signal propagates down the tree; if both subtrees have hands raised at a junction, then the regular signal goes down one path and a LowerYourHands flag goes down the other; if only one subtree has hands raised, then the regular signal goes down that path and nothing goes down the other path. Eventually, all nodes but one lower their hands. ActivateNode causes all nodes with hands raised to take on the attributes of the currently broadcast rule together with the specified priority. In this case, of course, only one node will have its hand raised.

The previous discussion glosses over the possibility of no free nodes in the system. We can check for free nodes by looking at the message returned by RaiseHands("F"). If no hands are raised, then we can signal an error condition.

Before actually adding a new rule to the system (in the sense of putting it into a free node), there are other things that we may wish to do. Here is a sampling of such other functions.

4.5.2 Removing Subsumed Old Rules

Theorem 7 lets us remove subsumed old rules. The following code does this:

```
RaiseHands("+-0=<");

Delete();
```

4.5.3 Checking for New Rule Redundancy

It may be that the new rule does not change the PRS function and therefore needn't be added. By Corollary 7.1 we check if the new rule is covered in the rule set by any single rule with equal polarity and greater than or equal priority:

```
Message = RaiseHands("+0=>");

if (Message.Priority >= NRPriority)

    return True;

else

    return False;
```

4.5.4 Expanding Concordant Old Rules

Sometimes, an old rule with the same polarity as the new rule may be made more general. The expanded part of the old rule must overlap the new rule, that is, any new inputs that are matched by the old rule must also be matched by the new rule so that the functionality of the system is not changed by the expansion (Theorem 12). Here is how this can be done by asking all nodes that satisfy Theorem 12 to raise their hands. All such nodes are then instructed to remove their one discriminant variable as per Theorem 12.:

```
Message = RaiseHands("+1<");

RemoveDiscriminant();
```

4.5.5 Expanding Discordant Old Rules

Discordant rules with lower priorities by Theorem 11. Since the priority of the new rule can be made higher than any other rule, we can do something like this:

```
Message = RaiseHands("-1<=");

RemoveDiscriminant();

if (NRPriority <= Message.Priority)

    NRPriority = Message.Priority + 1;
```

The last two lines guarantee that the requirements of Theorem 11 are satisfied.

4.5.6 Checking for New Rule Subsumation

It may be that an old concordant rule may be expanded in such a way that it completely subsumes the new rule, in which case, the new rule is redundant and needn't be added to the rule set at all. We can check for this as follows:

```
Message = RaiseHands("+1=");

if (Message.Priority < NRPriority)

    return False;

DropLowerPriorities();

RemoveDiscriminant();

return True;
```

We ask all old rules, whose variable lists differ from that of the new rule by only one discriminant variable, to raise their hands. If none has priority higher than that of the new rule, then the new rule cannot be subsumed and we return False. Otherwise, we tell all nodes, whose priorities are lower than the new rule, to drop their hands. All those nodes whose hands are still raised are instructed to remove their discriminant variable. The system then returns True to indicate that the new rule has been subsumed and need not be added to the rule set. We have used Theorems 12, 1, and 7.

4.6 Comments

The techniques and algorithms of this chapter are designed more for pedagogy than efficiency. A more efficient approach follows.

Each of the above techniques is self-contained. Each causes certain nodes to "raise their hands". Those hand-raised nodes take appropriate action. Most of these techniques are applied in a sequential manner, that is, part of the global algorithm looks something like :

RaiseHands

Perform action

RaiseHands

Perform action

RaiseHands

Perform action

Etc.

We can combine these in the following way. The new rule is presented. Each node sets its flags accordingly. Depending on the setting of its flags, each node immediately takes an appropriate action such as deleting itself, or removing a discriminant variable. All of these actions take place independently and in parallel.

Chapter 5

Additional Capabilities

5.1 Multiple Outputs

Up to this point we have discussed Prioritized Rule Systems that have a single output. We extend PRS to multiple outputs. An extended PRS accepts a vector of binary values as input and produces a vector of three-state values (0,1,?) as output.

The obvious way to expand a PRS in this way is to have one independent PRS for each desired element of the output vector. Input from the environment is channeled to each PRS and the individual outputs from the separate PRSs are combined to form the output vector. For example, let's say the input to our system consists of the four variables a, b, c and d and that the desired output is a three element vector, the elements of which we will call x, y and z. Suppose that whenever b is high and d is low that we want to output the vector 101. We may write this as follows:

Rule for x-PRS: (bd',+,1)

Rule for y-PRS: (bd',-,1)

Rule for z-PRS: (bd',+,1)

These rules may also be written in the following manner:

(bd',x,1)

(bd',y',1)

(bd',z,1)

Each of the three rules belongs to its own independent PRS. The variable in the second slot in the 3-tuple indicates to which PRS the rule belongs and whether the output should be high or low. The first rule says that whenever b is high and d is low, that the output from the x-PRS should be high. The second rule says that whenever b is high and d is low, that the output from the y-PRS should be low. A similar comment applies to the third rule. Finally, we may rewrite the three rules as one simple rule:

(bd',xy'z,1)

This new rule contains all of the information contained in the other rules.

We now indicate another possible implementation of a multiple-output PRS. Within each node of the PRS architecture, we have a single list of input variables and a single set of the flags as before. However, we duplicate some of the other structures. For example, if the system is to output a vector of three elements, each node will have storage for three output values. We have a choice as to how many priorities are stored. We can choose to have just one priority that applies to all three outputs, or we can have maximum flexibility by providing storage for three separate priorities, one for each output. The choice we make determines how much storage is needed in each node and what algorithms can be used for the manipulation of such multi-rules.

We have indicated two different approaches to providing a PRS with multiple outputs. The decision as to which approach to use depends on the application.

5.2 Multi-Valued Variables

To this point, we have dealt only with two-state (0,1) input variables and three-state (0,1,?) output variables. Prioritized Rule Systems are able to handle multi-valued variables.

Suppose we have an input variable m that can take on eight different values. We can represent m with three normal binary variables a, b, and c. For example, when m has the value 5, we set a to 1, b to 0, and c to 1. In this way, we may represent any multi-valued input variable (with a finite number of states) with the appropriate number of two-state variables. Output variables are handled similarly.

We establish a notation for specifying multi-valued variables in rules. We write the name of the variable followed in parentheses by one of the values that the variable can take on. For example, b(5) represents the variable b having the value 5. We might also use names for the values that a variable can assume. For example, if the variable t represents several temperature ranges, we might see t(hot), t(cold), or t(warm). Our normal two-state variables can also be expressed in this notation, for example, b(0) and b(1) or b(false) and b(true). Here is a rule using this notation:

(b(true)t(warm),m(walk),2)

This rule might mean, if the sky is blue and the temperature is warm, then the means of travel should be walking.

5.3 Mutual Exclusion

We introduce the concept of mutually exclusive variables.

Suppose there are three input variables r, g, and b that represent the three colors red, green, and blue. If these are components of a single color, then any subset of the three variables could be set to high. However, if the variables represent exactly three colors that the environment can take on, then only one of the variables can be high at one time. The colors are mutually exclusive. If the environment is red, then it is not green and it is not blue. For the sake of parsimony, we may wish to represent the three colors using only one three-state pseudo-variable which would actually be implemented using two real two-state variables as described in section 5.2. In this case, we would not be saving much, but if the environment could take on any of 256 different values, it would mean the difference between using 256 variables and eight variables.

Regular ASOCS has a certain problem with mutually exclusive variables that will be described in a later chapter. PRS has essentially solved that problem.

5.4 The Don't-Know State

In certain systems, we may wish to take one action if a certain condition X in the environment is true, a second action if X is false, and a third different action if we are uncertain of the value of X. This can be represented with a multi-valued input variable. For example:

(m(true),g(true),1)

(m(false),g(false),1)

(m(?),p(true),1)

These rules might represent the following: If the mall is open, go there. If the mall is closed, stay home. If it is unknown whether or not the mall is open, make a phone call to find out.

It should be obvious that the don't-know state can be added to any input variable by creating a multi-state pseudo-variable as described in section 5.3.

5.5 Stable Output (State Variables)

The definition of Prioritized Rule Systems may be modified in a trivial way to provide for state variables. The normal output of a PRS is augmented with a "final" output. Whenever the normal output of the PRS changes to a non-"?" value, that value is copied to the final output. In this way, the final output of a PRS is never undefined (except perhaps immediately after the system is initialized).

Chapter 6

Simulation

6.1 Purpose

A program was written to simulate the architectural model described in chapter 4. It was written to:

- Help to uncover any "surprises" not anticipated in the written part of the thesis.
- Provide a visual demonstration of the workings of Prioritized Rule Systems.
- Provide a pedagogical tool for future use in neural net a parallel processing courses offered by the Computer Science Department.

6.2 Attributes

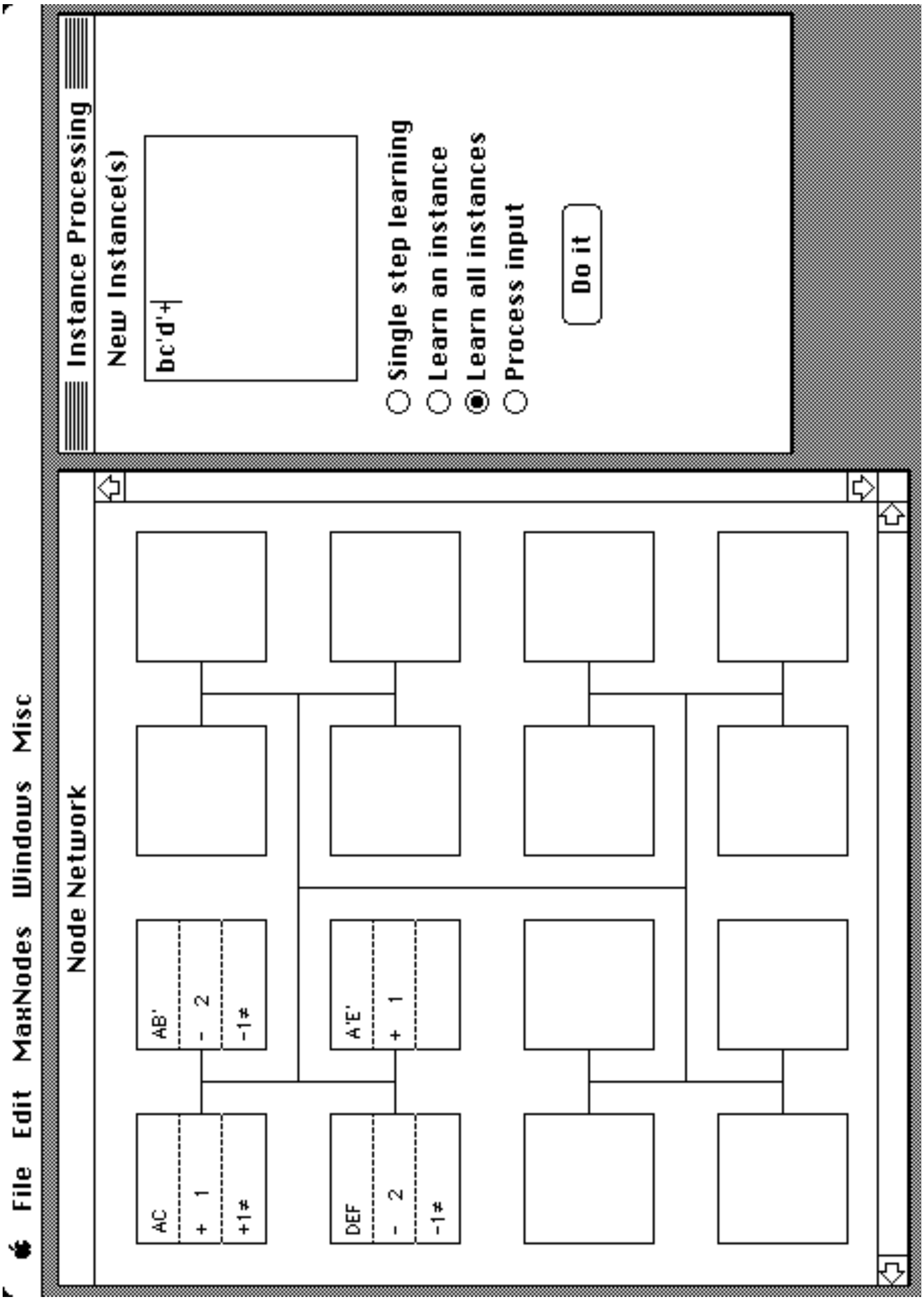
The program has the following attributes:

I. There are three windows.
   A. The Node Network window shows the binary tree in a two-dimensional grid.
      1. The network can have 4, 16, 64, or 256 nodes.
      2. The window can be moved to anywhere on the screen.
      3. The window can be made smaller or larger.
      4. The window can be scrolled to see other parts of the network.
      5. Each active node in the network shows its variable list, polarity, priority, and flag settings.
   B. The Legend window shows the meaning of the elements of each node.
      1. It tells what each field represents.
      2. It indicates the meaning of each flag bit.
      3. The window can be moved to anywhere on the screen.
      4. The window can be temporarily closed and later reopened.
   C. The Instance Processing window is where the main interaction with the user takes place.

1. The user may specify an input to be processed. Results are shown in this window and the "winning" node is highlighted in the Node Network window.
2. The user may enter new rules in this window. Individual rules are highlighted as they are being learned. The nodes in the Node Network window are updated during each stage of the learning.
3. Several modes of learning are available.
    a. All rules in the data entry window may be learned without further user interaction (batch mode).
    b. A single rule may be learned at once.
    c. A single rule may be learned in single step mode. In this mode, the user is informed of each function in the learning process that has been performed and the results thereof. The user clicks the mouse button to go from step to step. The individual nodes in the Node Network window reflect each step that is taken.

II. There is a menu which allows the following to be done.

A. The user may quit the program.
B. A file of rules may be opened and inserted into the data entry window of the Instance Processing window. The user may operate on these rules exactly as if he had entered them by hand.
C. The current network setting may be saved (as a list of rules) to a file.
D. The file may later be opened and read in as previously described.
E. The user may change the number of nodes in the network.
F. The user may select any of the three windows. This will bring the window to the front (handy if it was hidden by the other windows) and make it the active window.
G. Randomly generated tests may be run to check the integrity of the simulation


File
Edit
MaxNodes
Windows
Misc
Node Network
AC
+ 1
+1#
AB'
- 2
-1#
DEF
- 2
-1#
A'E'
+ 1
Instance Processing
New Instance(s)
bc'd'+
Single step learning
Learn an instance
Learn all instances
Process input
Do it

Chapter 7

Comparison With ASOCS

In chapter 3 we defined Rule Systems and then Prioritized Rule Systems. ASOCS is essentially a Rule System. The literature describes three architectural models based on ASOCS, namely AA1, AA2, and AA3 (Martinez, 1987). (The AA stands for "adaptive algorithm".) The following sections compare Prioritized Rule Systems and associated architectural models with ASOCS and associated architectural models.

7.1 Things in Common

Prioritized Rule Systems and ASOCS share the following strengths:

- They are capable of learning any Boolean function.
- They are capable of learning a rule set in a single pass.
- They are adaptable.
- They do not unlearn except when a contradiction occurs.
- They process input in log time.
- They learn in log time.

7.2 Advantages of PRS Over ASOCS

Prioritized Rule Systems have the following advantages over ASOCS:

- A PRS needs never add more than one node per rule. It doesn't suffer from "DVA expansion". This can be especially valuable when dealing with multi-valued and mutually exclusive variables described in a chapter 5.
- A PRS allows minimization between discordant rules.
- A PRS does not depend on the physical location of a node in the network.
- The new model uses nodes more "efficiently".

7.3 Disadvantages of PRS Compared to ASOCS

Prioritized Rule Systems have the following disadvantages compared to ASOCS:

- The individual nodes use more complex logic.
- The priority of each node must be stored, passed, and manipulated.

Chapter 8

Extensions and Future Directions

8.1 Generalization

One type of generalization is already built into the basic PRS model. Specific rules may be combined into more general rules. There is another type of generalization that can be handled in an extended version of PRS. (We will limit our discussion to binary-valued variables.) Whenever the normal output of a PRS is undefined ("?"), the system will actually output a 0 or 1 (and possibly set a flag indicating that this is an educated guess) to be determined in the following way. Assume that a new rule were added to the system that is absolutely specific; it matches only the current input. The rule could specify an output of a 0 or a 1. In each of those two cases, consider the smallest variable list that would cover that rule without also overlapping any rule with opposite output value. Whichever of the two cases has the smallest number of variables in its covering list "wins" and gets to output its value as the output of the system. In case of a tie, the system outputs a "?".

The generalizing method just mentioned is only one of a number of possibilities. A more thorough examination of the possibilities is recommended.

8.2 General Rules and Exceptions

Humans learn from specific examples; they tend to develop general rules with exceptions to those rules. For example, as we learn language, we unconsciously learn a general rule for forming the plural of nouns and the past tense of verbs. We also learn the exceptions to those rules. We may slowly learn the general rule that a noun is made plural by adding "s". An exception to that general rule is another less general rule that says to add "es". Finally, there may be some words that fit into no other rule and whose plural we must memorize. It could be desirable for Prioritized Rule Systems to develop in a similar way.

Consider the following prioritized rule set:

(abcd,-,1)

(abcd',+,1)

(abc'd,+,1)

(abc'd',+,1)

This rule set may be simplified in either of the following two ways:

(ab,+,1)	(ab,-,1)
(abcd,-,2)	(abc',+,2)
	(abd',+,2)

The first simplified rule set seems to be the most natural. It may seem even more so when the rule set is seen as a portion of a "Karnaugh map".

1	1
1	0

The more general lower-priority background rule (ab,+,1) specifies when the output should be high. (The rule specifies that all of the four cells in the box above should output a 1, although the 1 in the lower-right cell is hidden by the more specific rule (abcd,-,2).) The more specific higher-priority foreground rule (the exception) specifies when the output should be low. (The single lower-right cell outputs a 0, thus overriding the hidden 1 of the more general rule.)

Currently, a PRS may or may not "converge" to the simplified rule set that most naturally represents the general rule/exception concept just discussed (assuming there is such a "natural" representation). This may be a fertile area for further research.

8.3 Background Reduction

The types of rule reduction used in chapter 4 are guaranteed to be efficient; a new rule can be added in O(log(n)) time. The theorems of chapter 3 provide for more types of reduction than are actually used in chapter 4. Some of these other types of reduction are not as efficient. In fact, making the rule set absolutely minimal could require exponential time. There are, however, some techniques that would require only O(log(n)log(m)) time where n is the number of nodes and m is the number of variables. For many applications, these techniques could be included in the standard algorithms for adding new rules. For some other applications, the addition of new rules must be as fast as possible at the time the new rules are presented to the system. However, there may be other times when the system is not busy processing inputs or adding new rules. This time may be used to advantage for applying some of the more costly reduction techniques. This concept of background reduction needs to be more fully explored.

Chapter 9

Conclusion

We have introduced Prioritized Rule Systems and have described an architectural model based on the PRS abstract model. We have shown that this model has a number of significant advantages over the model known as ASOCS from which the PRS model was derived. A rigorous foundation has been established with the development of definitions and theorems. A simulation program has been written based on both the abstract PRS model and the described architectural model. Finally, some directions have been given for further research work. This work provides a stepping stone for the work that lies ahead.

Bibliography

Backus, J. (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, **21**, No. 8, 613-641.

Hinton, G. E. & Sejnowski, T. J. (1984) *Boltzmann machines: Constraint satisfaction networks that learn.* (Tech. Rep. No. CMU-CS-84-119), Pittspurgh, PA: Carnegie-Mellon University, Department of Computer Science.

Hopfield, J. J. (1982) Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, USA, **79**, 2554-2558.

Hopfield, J. J. (1984) Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences*, USA, 81, 3088-3092.

Martinez, T. R. (1986) *Adaptive Self-Organizing Logic Networks*. Ph.D. Dissertation, Technical Report - CSD 860093, University of California, Los Angeles, CA.

Martinez, T. R. (1987) Models of parallel adaptive logic. *Proceedings of the 1987 IEEE Systems Man and Cybernetics Conference*, October, 290-296.

Martinez, T. R. & Vidal, J. J. (1988) Adaptive parallel logic networks. *Journal of Parallel and Distributed Computing*, **5**, No. 1, 26-58.

Martinez, T. R. (1988) Digital neural networks. *Proceedings of the 1988 IEEE Systems Man and Cybernetics Conference*, August, 681-684.

Martinez, T. R. (1989) Neural network applicability: Classifying the problem space. *Proceedings of the IASTED International Symposium on Expert Systems and Neural Networks*, August.

Minsky, M. and S. Papert (1969) *Perceptrons, an Introduction to Computational Geometry*. MIT Press, Cambridge, MA.

Rosenblatt, F. (1962) *Principles of Neurodynamics*. Spartan Books, Washington D. C..

Rumelhart, D. and J. McClelland (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. **1**, MIT Press, Cambridge, MA.

Prioritized Rule Systems

Brent W. Hughes

Department of Computer Science

M. S. Degree, November 1989

Abstract

Non-von Neumann architectures attempt to overcome the "word-at-a-time" bottleneck of traditional computing systems. Neural nets are a class of non-von architectures whose goal is, among other things, to learn input-output mappings using highly distributed processing and memory. A neural net consists of many simple processing elements (nodes) with modifiable links between them, allowing for a high degree of parallelism. A typical neural net has fixed a topology. It learns by modifying the "weights" or "conductances" of the links between nodes.

Another model with similar goals, called ASOCS, learns by modifying its topology. Unlike a typical neural net, ASOCS is guaranteed to be able to represent and learn any desired mapping and to do so efficiently.

This thesis presents an extension of ASOCS called Prioritized Rules Systems. The PRS abstract model provides a foundation for various architectural models that have a number of advantages over other ASOCS models. One example of such an architectural model is presented in the thesis along with a description of a program written to simulate that model. The processing and learning algorithms of the architectural model are based on theorems proved in the thesis.

COMMITTEE APPROVAL:

____
Tony Martinez, Committee Chairman

____
Douglas Campbell, Committee Member

____
Larry Christensen, Graduate Coordinator