Regular Correspondence

A Self-Organizing Binary Decision Tree for Incrementally Defined Rule-Based Systems

Tony R. Martinez and Douglas M. Campbell

Abstract—**This paper presents an adaptive self-organizing concurrent system (ASOCS) model for massively parallel processing of incrementally defined rule systems in such areas as adaptive logic, robotics, logical inference, and dynamic control. An ASOCS is an adaptive network composed of many simple computing elements operating asynchronously and in parallel. This paper focuses on adaptive algorithm 3 (AA3) and details its architecture and learning algorithm. It has advantages over previous ASOCS models in simplicity, implementability, and cost. An ASOCS can operate in either a data processing mode or a learning mode. During the data processing mode, an ASOCS acts as a parallel hardware circuit. In learning mode, rules expressed as Boolean conjunctions are incrementally presented to the ASOCS. All ASOCS learning algorithms incorporate a new rule in a distributed fashion in a short, bounded time.**

I. Introduction

This paper presents an adaptive self-organizing concurrent system (ASOCS) architecture [6], [7], [10] that guarantees learning for Boolean rule-based systems in bounded time. This particular ASOCS uses adaptive algorithm 3 and has significant simplicity, implementability, and cost advantages over previous ASOCS models [8], [9]. Target applications include rule and example based systems for logical inference, robotics, adaptive logic, fault-recovery, and real-time dynamic control.

The search for fast and robust computation has increased research in highly parallel systems with both traditional [2], [4] and connectionist [5], [14] views. Researchers of massively parallel systems seek speed both during processing and learning. However, programming and updating massively parallel systems incur tremendous overhead and complexity.

The goal of ASOCS is to train (program) a parallel digital network to solve problems defined by rule-based propositional logic. A system is trained (programmed) through the incremental input of rules expressed as conjunctions of Boolean variables. Real world applications using rule-based propositional logic are forthcoming [3].

An ASOCS is an adaptive network of many simple computing elements operating in a parallel, asynchronous fashion. ASOCS can operate in both data processing and data learning modes.

During data processing the system acts as a parallel hardware circuit; it asynchronously maps input data to output data in $O(\max(d, \log n))$ time, where d is the maximum depth (longest path) of the network, and n is the number of network nodes, as is typical for hardware circuits.

During learning the system reconfigures itself in a distributed manner to accommodate new (and perhaps conflicting) rules. ASOCS potential comes from its ability to guarantee adaptation in $O(\log(n))$ time for any new rule. Through its learning process the system discovers the critical variables; it uses these to generalize and classify large input spaces.

Manuscript received October 12, 1989; revised April 12, 1991.z

The authors are with the Computer Science Department, Brigham Young University, Provo, UT 84602.

IEEE Log Number 9101213.

The majority of ASOCS research on adaptive algorithms has focused on adaptive algorithm 1, adaptive algorithm 2, and adaptive algorithm 3. Details for AA1 can be found in [8]; details for AA2 can be found in [9]. These three algorithms vary dramatically, although AA3 shares some similarity to AA2.

ASOCS arose from reexamining perceptron [12] related ideas. The basic building block, however, is that of digital programmable nodes, an idea spawned by the notion of a universal logic module (ULM) [16]. Verstraete [15] sought methods of programming fixed ULM structures to solve arbitrary Boolean mappings. ASOCS departs from these efforts by using a nonpassive network that adapts in a self-organizing fashion [6], [10]. This technique has led to models promising parallel inference, high-speed adaptation, and internal consistency control. Proof of concept VLSI fabrication of ASOCS devices has been completed [1] and other implementations are currently underway. Formal proof of AA3 is found in [17].

Although ASOCS research was initiated with a neural network emphasis, the proposed mechanisms differ extensively from standard neural network paradigms. The authors make no claim to be modeling neural functionality with this model. Rather, distributed, parallel, and self-organizing paradigms are used in order to attain an improved *computational* mechanism, offering speed, fault tolerance, and ease of use.

The outline of the paper follows. Section II defines the mechanism of knowledge input. Section III described AA3 during processing mode. Section IV describes AA3 during learning mode. Section V works in detail a concrete example of AA3 learning. Section VI extends the model to multiple outputs. Section VII discusses the advantages of ASOCS. Sections VIII and IX overview simulation results, comparisons with AA1 and AA2, and current research efforts.

II. Knowledge Input

The atomic knowledge element is the *instance*. Each instance is a (partial) function from a set of Boolean variables to a Boolean variable. Thus each instance is a propositional production rule.

Following are examples of instances:

I) $\sim A \sim B \Rightarrow C$
II) $A \sim BC \Rightarrow \sim A$
III) $\sim A \sim B \Rightarrow \sim C$.

Instance I forces C to become true whenever A and B are false. Instance II forces A to become false whenever A and C are true and B is false. Instance III forces C to become false whenever A and B are false. Instances I and III are *inconsistent* with each other; when A and B are false, instance I tries to set C to true while instance III tries to set C to false. When the variables of an instance are not matched, the instance says nothing about the output of the function. (An implementation may set the output variable to true, to false, or even to don't know.) Instances are incomplete and partial functions by definition.

The union of a set of instances defines another partial function. It is a (partial) function defined by a set of rules, a functional rule base. A set of instances is called an *instance set*. An instance with a nonnegated variable on its right-hand side is called a *positive* instance; an instance with a negated variable on its right-hand side is called a *negative* instance. Thus each instance has either negative or positive *polarity*.

A Boolean variable occurring in one instance and occurring in its complemented form in another instance is said to be a *discriminant variable* for the two instances.

An instance set S is *consistent* if S does *not* contain any two instances X and Y where X is a positive instance, Y is a negative instance and 1) they have the same right-hand variable and 2) there is a set of Boolean values which can simultaneously match the left-hand sides of X and Y. We require an instance set to be well defined (*consistent*). Thus, no two instances have a common set of Boolean values for which the function produces inconsistent values.

Two fundamental types of inconsistencies exist. The first type is obvious; the same variables define the output inconsistently as in $AC \Rightarrow B$ and $AC \Rightarrow {\sim}B$. The second type is more subtle. Although the rules $AD \Rightarrow B$ and $C \Rightarrow {\sim}B$ do not have identical left-hand variables, when A, D, and C are all true, B becomes both true and false.

Fortunately, there is a basic computational tool for inconsistency. Its proof is omitted.

Lemma: Two instances are inconsistent if and only if they are of opposite polarities and have no discriminant variable.

I) $AB \Rightarrow C$
II) $BC \Rightarrow {\sim}C$
III) ${\sim}BC \Rightarrow {\sim}C$.

For example, Instance I above is inconsistent with II because there is no discriminant variable. Instance I is consistent with III because of the discriminant variable B. Instances II and III cannot be inconsistent because they are of the same polarity.

In the ASOCS model, instances are input incrementally. Let NI be a new instance and S be an instance set. Either NI is consistent with S or else there is at least one instance of S with which NI is inconsistent.

An NI may contain new information or be redundant to information already contained in the instance set.

By definition, a NI is redundant with respect to an instance set S, if the partial function defined by the NI is already contained in the partial function defined by S. For example, let the instance set be

$$D \Rightarrow C$$
$$B \Rightarrow C$$

and let the new instance be $AB \Rightarrow C$. Clearly, the fact that AB forces C is already contained by the second instance of the original system. Since the new instance adds no information, parsimony suggests that we delete it.

By definition, an NI contains new information with respect to an instance set S if the partial function defined by the NI extends the partial function defined by S. For example, let the instance set S be

$$A \Rightarrow C$$
$$B \Rightarrow C$$

and let the new instance be ${\sim}A{\sim}B \Rightarrow C$. In this case, the new instance tells us to set C to true when A and B are false, an extension of the partial function defined by S.

The principle of parsimony may also apply to new information. For example, let the instance set be

$$AB \Rightarrow C$$
$$A{\sim}C \Rightarrow C$$
$$B \Rightarrow C$$

and let the NI be $A \Rightarrow C$. The new instance contains new information; it tells us what to do when A is true and B is false, as well as what to do when A is true and C is false. But more than that, the original instances $AB \Rightarrow C$ and $A{\sim}C \Rightarrow C$ are now redundant as they are but special cases of $A \Rightarrow C$. If we can detect such redundancies quickly, then parsimony suggests they be deleted. However, perhaps more important than parsimony, is that by removing "don't care" information, the system can find *critical* features that can aid in generalizing to a good output when the system receives input for which no training has taken place.

If a new instance is inconsistent with an instance set, then we give precedence to the newer instance and *remove the contradicted portion of the old instance.*

Theorem: Let $S1 \Rightarrow Z$ be a new instance and $S2 \Rightarrow {\sim}Z$ be an old instance. Suppose there is no discriminant variable for the new and old instance. If $S1$ is a subset of $S2$, then every part of the old instance contradicts the new instance.

Proof: Since $S2$ is a subset of $S1$, every set of variables that realizes $S2$ (and forces Z to be false by the old instance) extends to variables that realize $S1$ (which forces Z to be true by the new instance). Therefore, every part of the partial function defined by $S1 \Rightarrow Z$ is a rewrite of $S2 \Rightarrow {\sim}Z$, that is, the new instance contradicts *everything* for which the old rule stood. Q.E.D.

Theorem: Let $S1 \Rightarrow Z$ be a new instance and $S2 \Rightarrow {\sim}Z$ be an old instance. Suppose there is no discriminant variable for the new and old instance. Suppose $S1$ is not a subset of $S2$. Let $S3$ be those variables in $S1$ that are not in $S2$. Then the part of the old instance that is not contradicted by the new instance is given by the following set of instances $\{{\sim}IS2 \Rightarrow {\sim}Z$: I is in $S3\}$.

Proof: Since $S1$ is not a subset of $S2$, there is a variable in $S1$ that is not in $S2$. Thus, the set $S3$ is not empty. Let I belong to $S3$. The instance $S2 \Rightarrow {\sim}Z$ is the union of the two rules: $IS2 \Rightarrow {\sim}Z$ and ${\sim}IS2 \Rightarrow {\sim}Z$. Since I belongs to $S1$ and since there is no discriminant variable for $S1$ and $S2$, we can find Boolean values for $S1$, I, and $S2$ so that all hold, forcing Z to be set inconsistently. On the other hand, $S1 \Rightarrow Z$ and ${\sim}IS2 \Rightarrow {\sim}Z$ now share the discriminant variable I and are therefore not inconsistent. That is, we may save the ${\sim}IS2 \Rightarrow {\sim}Z$ part of the old instance for each variable in $S3$. This concludes the proof of the theorem. Q.E.D.

For example, consider the new instance $ACD \Rightarrow {\sim}Z$ confronting the old instance $AB \Rightarrow Z$. Since ACD is not a subset of AB we form the variables in the new instance that are not in the old instance, namely C and D. The theorem says to replace $AB \Rightarrow Z$ by the pair of instances ${\sim}CAB \Rightarrow Z$ and ${\sim}DAB \Rightarrow Z$.

We may therefore take any consistent instance set and any new instance and produce a new instance set that contains as much of the old instance set as can be saved with the new instance being given precedence. The number of instances in the new instance set may be greater than, less than, or equal to the number of instances in the original instance set (depending on the amount of redundancy or contradiction). In the new instance set, all instances have equal priority and order is again inconsequential.

Instances may come from human intervention or automated mechanisms. Typical learning has more general instances (those with fewer antecedent variables) entered first with refinement through more specific instances (those with more antecedent variables).

In the AA3 ASOCS implementation of the next section, the system maintains consistency in a manner invisible to the user. By dynamically modifying the instance set, the system discovers which variables are critical in making decisions. This leads to natural algorithmic *generalization* through critical variables when the system receives novel inputs. In the AA3 ASOCS model, the system does not explicitly store the instance set; instead, the system *stores the information implicitly in a distributed fashion.*

III. The ASOCS AA3 Model

In this section we show how the AA3 ASOCS dynamic parallel

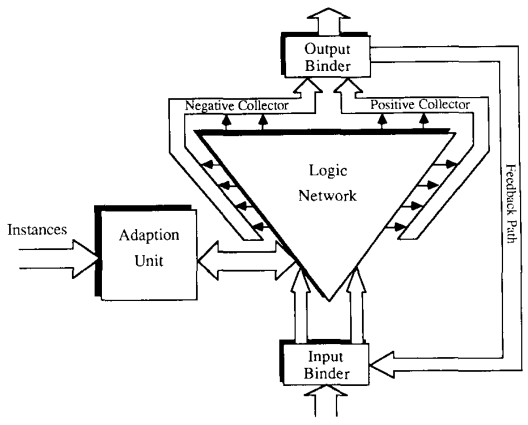

Fig. 1. AA3 architecture.

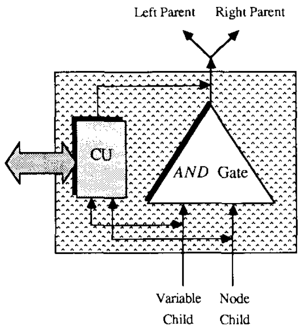

Fig. 2. AA3 network node.

network models a consistent instance set. We say that an AA3 ASOCS model *fulfills* a consistent instance set if whenever an input forces the instance set to logically output Z (or $\sim Z$), then the AA3 dynamic parallel network physically outputs the corresponding Z (or $\sim Z$).

Fig. 1 gives the architecture of the AA3 ASOCS model. This section only discusses those parts of the architecture needed for execution. In Section IV we discuss how the model knows how to represent itself (how it learns).

During execution mode, only the logic network is active. The input data flows asynchronously through the network with only propagation delays. The feedback path allows the system to use an output variable as an input variable. (If a clocked register is added at the output binder, then the system is similar to a dynamically adaptable finite state machine.)

Each node within the logic network has the structure of Fig. 2. During execution mode, only the dyadic AND gate is active. Each node has two inputs called *children*: the *node* child and the *variable* child.

The node child receives its input from a node; the node child's value represents a conjunction of other variables.

The variable child is connected directly to an input variable. The conjunction of the node child and variable child is available for direct output as well as for further processing by another node. If a node outputs to another node, then it must output to exactly two nodes: its *left parent*, and its *right parent*. Such parent nodes are said to be *siblings* with respect to each other.

Each node control unit has memory for a 3-state polarity flag

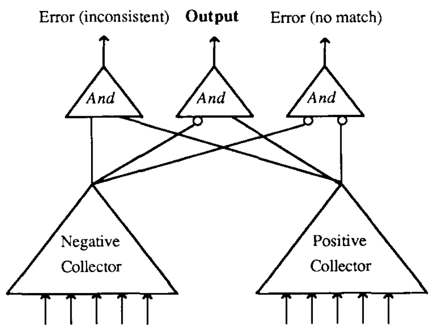

Fig. 3. Collector structure.

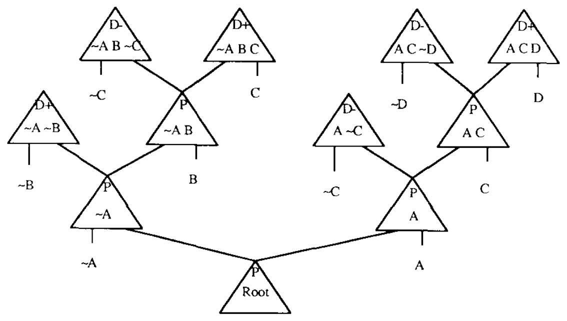

Fig. 4. Example network configuration.

with value: P, $D+$, or $D-$. The symbol P signifies the node is a Primitive node (*P*node for short). $D+$ signifies the node is a positive Discriminant node (positive *D*node); $D-$ signifies the node is a negative Discriminant node (negative *D*node).

Each node also contains a *variable list*, which is the total set of variables over which the node does a logical conjunction.

The overall structure of the AA3 network is that of a binary decision tree as in Fig. 4. A node is a *D*node if and only if it is a (top) leaf in this structure (has no parents). A node is a *Pnode* if and only if it is an internal node in the tree (has parents).

The output of all positive *D*nodes is sent to the *positive* collector; the output of all negative *D*nodes is sent to the negative collector. Each collector performs a logical *OR* of its received values. Since the overall network is a binary decision tree, *exactly one Dnode will be active for any input.* A three node structure handles the collector outputs in Fig. 3. The middle node is the output node; it outputs Z if the positive collector is active (a positive *D*node is active); it outputs $\sim Z$ if the negative collector is active (a negative *D*node is active). If *both* positive and negative collector are inactive, or both collectors are active, then the network has an error. An error will not happen in the model as described, but could occur if there is a hardware fault in a physical implementation of the model.

The bottom node of the network is the *root* node; it is has no inputs, its logic gate always outputs true, and it is initially a negative *D*node.

The *maximum depth of the network* is equal to the number of bound input and feedback variables plus the root node.

Fig. 4 illustrates a possible AA3 configuration. Each node represents a conjunction of a set of variables. The symbol at the top of the node is the polarity and node indicator. The 16 possible outputs of the network in Fig. 4 are give in Table I. Note that although an instance

TABLE I
NETWORK FUNCTION REPRESENTATION

A	B	C	D	output
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

set represents a partial function, an actual AA3 network represents a total function on those inputs that are part of the network.

IV. THE AA3 LEARNING ALGORITHM

In this section we first discuss the architecture that supports the learning algorithm and then the learning algorithm itself.

A. Architecture

When the system receives a new instance that contains a new input variable, then the input binder of Fig. 1 allocates an input line for the new variable.

When the system receives a new instance the adaption unit of Fig. 1 broadcasts to the logic network the variables and polarity of the new instance. This information allows all nodes within the network to work cooperatively and in a distributed fashion.

The node of Fig. 2 contains a *control unit* able to execute the learning and deletion algorithm. The control unit is also able to send messages to its node child and to its siblings (if they exist).

We emphasize that the network does not store the original instance set. Indeed, as the example in Section V shows, it is usually logically impossible to reconstruct the instance set from the network.

B. The Learning Algorithm

We describe the AA3 learning algorithm that tells how a consistent network reconfigures itself when faced with a new instance. At the completion of the AA3 learning algorithm (and at the completion of its deletion algorithm) the network is still consistent and represents the *functionality* of the previous network coupled with new information from the new instance.

Each node has a type (*D*node or *P*node), polarity, and variable list, labeled as *node.type, node.polarity* and *node.variables* respectively. Node.polarity is null for a *P*node.

When the system receives a new instance, the AU broadcasts to each node the *instance.variables* and the *instance.polarity* of the new instance. Each node then independently follows the following learning algorithm.

```
(01) if (node.type = D) and
(02)       (instance.polarity < > node.polarity) and
(03)       there is no discriminant variable between
           instance.variables and node.variables then
(04)     if (node.variables is a proper superset of
         instance.variables) then
(05)           polarity_inversion(node)
         else begin
(06)           S := instance.variables − node.variables
(07)           DVA (node,S)
         end;
(08)     Self-Deletion ( )
     end
```

C. Remarks

Line (01)–(03): Only a *D*node that has opposite polarity to the new instance and that cannot be discriminated from the new instance ever "learns" (is modified).

Line (04)–(05): If a node has the same or more variables than the new instance, then the new instance directly contradicts the node for all active states. The contradiction will disappear if the node changes its polarity to that of the new instance. Polarity inversion flips the polarity of the node and redirects the node's output to the opposite collector.

Line (06): Since instance.variables is always non-empty and since it failed line (04), some instance.variable is not a node.variable. Therefore, the set S is not empty. For each variable in S, the new instance contradicts the node for some state. Therefore, for each such variable we add two nodes to the network to resolve this contradiction. We do this recursively through the procedure DVA.

Line (07): Informally, the recursive procedure DVA takes a variable V from S, wires in two new parent nodes for S, deletes V from S, changes the old node to a *P*node, and recursively calls itself for the concordant parent node if there still are some variables left. More formally,

```
Procedure DVA (node, S);
(09) Allocate (node 1); Allocate (node 2);
(10) Set V to an element of S;
(11) node1.polarity  := node.polarity.
     node1.child     := complement of V
     node1.type      := D;
(12) node2.polarity  := instance.polarity,
     node2.child     := V;
     node2.type      := D;
(13) node.type       := P;
(14) if |S| > 1, then DVA (node2, S − V);
```

D. Remarks

Line (09): Node1 and node2 are the new parent nodes.

Line (10): The order of variables chosen is inconsequential

Line (11–12): The new parents become *D*nodes of opposite polarity, each differing in node.variables only by the discriminant variable V.

Line (13): Since node is no longer a *D*node, change its type to a *P*node.

Line (14): If S has more than one element, then recursively call DVA for node2 and $S - V$.

We give an example of DVA. Consider the positive *D*node of Fig. 5 with variables $A{\sim}B$ confronting the new instance $A{\sim}BCD \Rightarrow {\sim}Z$. The node executes the learning algorithm. The node is a *D*node, has opposite polarity to the new instance, and has no discriminant variable with respect to the new instance. Since $A \sim B$ is not a superset of $A{\sim}BCD$, the set of instance.variables−node.variables is $\{C, D\}$. DVA is called twice. Assuming the discriminant variable C is chosen first, Fig. 6 shows the modification after one DVA recursion, and Fig. 7 the final reconfiguration after both.

All DVA modifications take place independently and in parallel.

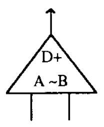

Fig. 5. Initial positive *D*node.

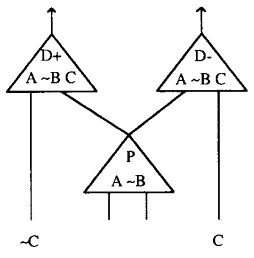

Fig. 6. After one DVA.

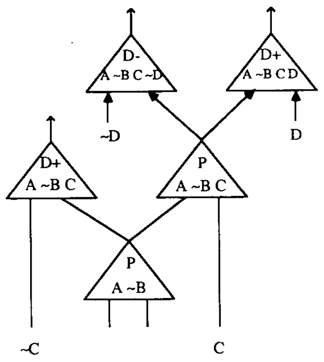

Fig. 7. After complete DVA.

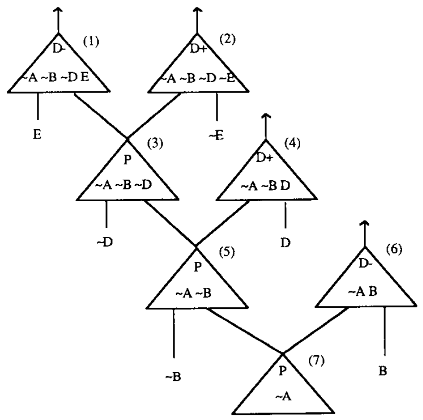

Fig. 8. Initial network section.

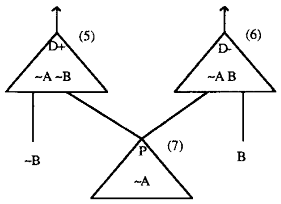

Fig. 9. Final network section after self-deletion.

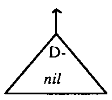

Fig. 10. Initial network configuration.

The final step of the algorithm is self-deletion. Each *D*node has a unique *sibling*. At the end of the learning algorithm, each node independently executes the following deletion algorithm.

```
(15) Self-Deletion
(16)    If (node.type = $D$) and
(17)    (node.polarity = sibling.polarity) then begin
(18)          Inform_child(node)
(19)          Self-Delete(node);
              Self-Delete(sibling).
        End;
     End;
```

Lines (15)–(19). Sibling *D*nodes that have the same polarity are superfluous; if their child node fires, then one of the two nodes must fire with their common polarity. Therefore, delete both nodes and make their child node a *D*node of the same polarity as the parents. This procedure is then recursively initiated by the new *D*node with its sibling.

For example, assume initially the section of network shown in Fig. 8.

Assume that the new instance $\sim A \sim B E \Rightarrow Z$ is broadcast. Nodes 1 and 6 are the only discordant *D*nodes. Node 6 is discriminated by variable B. Node 1 is superset so it does polarity inversion, becoming a positive *D*node. Nodes 1 and 2 are both positive *D*nodes and they self-delete, causing node 3 to become a positive *D*node. At this point nodes 3 and 4 are both positive *D*nodes and they self-delete causing node 5 to become a positive *D*node. Since the sibling of node 5 (node 6) is a negative *D*node, no more self-deletion occurs in this section of the network. The final network section after these modifications is shown in Fig. 9.

Like DVA, self-deletion occurs independently and in parallel throughout the network.

V. Illustration of the Distributed Mechanism

In this section we illustrate each aspect of AA3 with an example. The example demonstrates that the original instance set may be distributed throughout the system and that the original instance set need not be recoverable from the network.

We start the network in the null state of Fig. 10. We describe the evolution of the network as seven instances are input.

Instance I — $A \sim BC \Rightarrow \sim Z$ (negative): The root node begins as a negative *D*node. Since the initial network has no positive *D*nodes, no matching occurs. The network remains as given in Fig. 10.

Instance II — $\sim ABDE \Rightarrow \sim Z$ (negative): Since Fig. 10 has no positive *D*nodes, the network remains unchanged.

Instance III — $BDE \Rightarrow \sim Z$ (negative): Since there still are no positive *D*nodes in Fig. 10, the network remains unchanged.

Instance IV — $ABC \Rightarrow Z$ (positive): Since the root node has no variables, there is no discriminant variable for the root node and the new instance. Since ABC is a superset of the variables of the

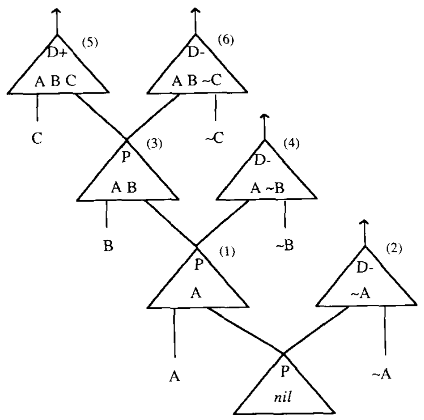

Fig. 11. Modified network.

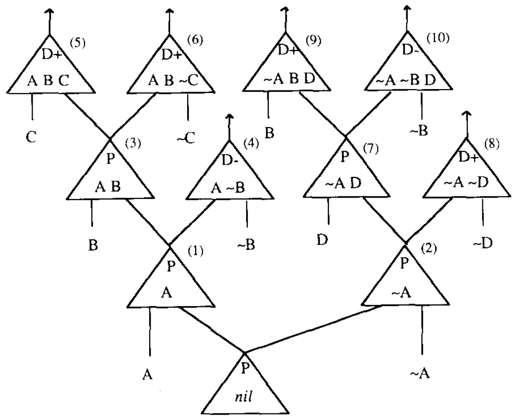

Fig. 13. After polarity inversion and DVA.

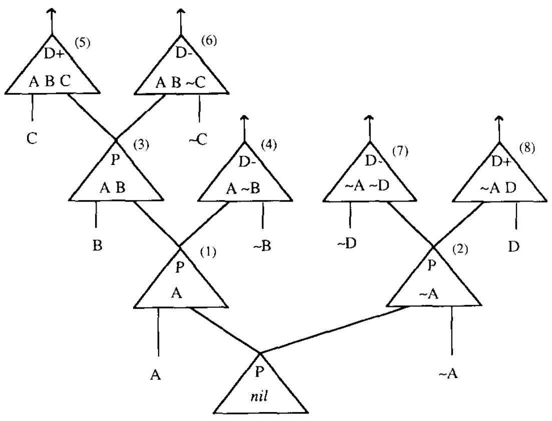

Fig. 12. Modified network.

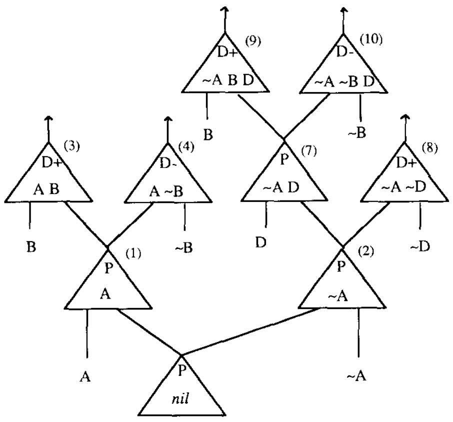

Fig. 14. Final network.

root node, a DVA is done for each of the variables A, B, C. If the DVA is done in alphabetical order, then Fig. 11 is produced. Since no self deletion is possible, the final network configuration remains as in Fig. 11.

Instance V — $\sim AD \Rightarrow Z$ (positive): The variables $\sim AD$ are compared with the variables of the three negative *D*nodes (2), (4), (6). A is a discriminant variable for node (4) and for node (6). Node (2) and the NI do not share a discriminant variable. The variable list $\sim A$ of node (2) is a subset of the variable list $\sim AD$ of the NI. Therefore, DVA is done at node 2 with the variable D. Fig. 12 gives the network. Since no self-deletion is possible the network remains as in Fig. 12.

Instance VI — $A \sim B \sim C \Rightarrow \sim Z$ (negative): The variables $A \sim B \sim C$ are compared with the variables of the positive *D*nodes (5) and (8). C is a discriminant variable for node (5); A is a discriminant variable for node (8). Since there are no positive *D*node matches, the network remains unchanged. The NI is already fulfilled by the network.

Instance VII — $B \Rightarrow Z$ (positive): The variable B is compared with the variables of the negative *D*nodes (4), (6), and (7). B is a discriminant variable for node (4). Neither nodes (6) nor node (7) have a discriminant variable for $B \Rightarrow Z$. The variables of node (6), $AB \sim C$, are a superset of the NI. Node (6) therefore undergoes polarity inversion. Since the variables of node (7) are not a superset of the NI, DVA must be done. DVA is done for each variable in $B \Rightarrow Z$ that does not appear in $A \sim D$, namely for B alone. Fig. 13 gives the network. Self-deletion is now possible. Nodes (5) and (6) are siblings of the same (positive) polarity; they delete themselves; their child, node (3), becomes a positive *D*node as in Fig. 14. Self-deletion continues. Nodes (3) and (4) are siblings, but have opposite polarity. Nodes (9) and (10) are siblings, but have opposite polarity. There are no other siblings; the self-deletion stops. The final network is that of Fig. 14.

Note that the network fulfills the instance set with a distributed mechanism. Indeed, *no single node is responsible for the final instance* $B \Rightarrow Z$. The responsibility is spread across the three positive *D*nodes (3), (8), and (9). One of these nodes is active whenever B is true, depending on the value of the other inputs, thus fulfilling the instance.

VI. Multiple Outputs

Having discussed the architecture and learning and deletion algorithm for a single output, we extend to multiple outputs. We first discuss the changes to the architecture and then the changes to the algorithms.

Architecturally, a node has an additional polarity flag for each output variable for which it is a *D*node. Each output variable has its own positive and negative collector. A node could be a positive

*D*node for one variable, a negative *D*node for another variable, and a *P*node for a third variable.

During any learning cycle, the AU sends three items: the polarity of the instance, its variables, and the output variable name. Each node must check whether it is a *D*node for the output variable of the new instance. During DVA, a new *D*node must also set the corresponding polarity flag for the current output variable. If a node is being used for other variables, then when told to self-delete it simply sets its current output variable flag to nil. If a node is no longer being used for other variables, then when told to self-delete it can do a true node deletion.

The real power and efficiency of the system becomes evident with multiple outputs; sharing of network nodes can be done with multiple outputs, leading to more efficient use of hardware.

VII. Freedom from Exponential Complexity

The majority of current connectionist models assume an initial fixed number of nodes and a static interconnect where the dynamic aspect of the network is in the weights on links between nodes. The ASOCS model assumes a dynamic structure where the number of nodes may increase or decrease and links between nodes are also dynamic. (Efficient implementation technologies to allow dynamic structure are mentioned below.)

The main advantage of a dynamic structure is the potential for the model to solve complex functions without necessity of exponential space or time demands. For example, assume a static binary decision tree (BDT) that is *universal* for 20 input variables. By universal it is meant that it will be able to solve any of the possible Boolean functions of 20 variables. There are 2^{2^n} functions of n variables and in this case the BDT would have to have 2^n or 1 000 000 nodes in order to guarantee representation of an arbitrary function of 20 inputs. As can be seen, the number of nodes grows exponentially. It appears that any static neural network model that guarantees arbitrary function learning will require an exponential amount of space whether it be in number of nodes, number of links, resolution of weights, etc. Use of a dynamically structured neural network allows models that guarantee learning of complex functions without requiring exponential space.

There are random functions that appear to always require exponential demands, such as recognizing white noise on a raster screen. However, these types of applications are expressly outside of the application domain targeted by both artificial and natural neural systems [11]. The application space for which neural networks and ASOCS are targeted feature input/output mappings where generalization and minimization can take place, thus allowing a parsimonious network solution. However, since the specific mappings are not known *a priori*, it requires a dynamic network structure in order to take advantage of this occurrence.

For example, in representing a Boolean function of 20 variables with a BDT, some branches of the tree may be of length 20, while the majority, when minimized, may be much shorter. The dynamic structure accommodates those parts of the network requiring the full complexity, while requiring only sufficient nodes for the rest of the network.

There are a number of ways to implement a logically dynamic network. One model of AA3 uses small BDT's, with a maximum depth of say 10. Complex connections exceeding 10 variables pass through a dynamic router into another 10-depth AA3 module. Extra modules are thus only used where required. Another class of mechanisms uses a logically independent network with a single network node representing a *D*node, with the nodes connected to a broadcast topology [13], i.e., tree, mesh, optical, etc. This allows dynamic logical structure while maintaining parsimony of node usage at an implementation level.

VIII. Simulation and Comparison with Two Other Alogrithms

Software simulation of AA3 indicates that each instance generates two nodes on the average [6] for a single variable output. As discussed in Section VI, this statistic should improve with multiple variable outputs.

AA3 improves AA1 [8] in two ways.

1) *Adaption unit functionality*: In AA1 the adaption unit must store and maintain a consistent instance set. In AA3 the adaption unit merely broadcasts the instance to the network; the network handles storage and consistency in a distributed manner.

2) *Memory requirements*: In AA1 the memory requirement per node is proportional to the product of the instance set size and the number of output variables. In AA3 a single variable list is stored at each node, and 2-bits are required for each output variable that a *D*node defines.

Although AA3 is similar to AA2 [6], [11], it improves AA2 in two significant ways.

1) *Simpler learning algorithm*: AA2 does comparison with all nodes, building a node that exactly matches each new instance. AA3 matches only with discordant *D*nodes, and requires only that the network can *fulfill* the instance set.

2) *Ease of implementation*: AA2 requires dynamic interconnect between network nodes. AA3 uses a fixed interconnect model, with improved potential for VLSI design.

IX. Conclusion

This paper introduces adaptive algorithm 3 (AA3) for adaptive self-organizing concurrent systems. It features the following:

1) guaranteed mapping of arbitrary Boolean functions,
2) bounded linear learning time (logarithmic with the number of nodes)
3) stability of previously learned patterns,
4) ability to extract critical features from a large environmental input.

AA3 implements an ASOCS model with:

5) negligible memory requirements,
6) distributed storage and maintenance of the knowledge base,
7) a simple learning algorithm,
8) a fixed interconnect.

New models that allow simpler implementation and a more robust instance mechanisms are now being proposed. In order to give the model expanded utility in real-world applications, current research thrusts include:

A) extension of ASOCS data and functional primitives to higher order structures (including analog and multistate variables),
B) investigation and enhancements of the ASOCS feedback mechanism to allow temporal dynamics and sequential algorithms,
C) new models with improved speed, programmability, and fault tolerance,
D) improved generalization mechanisms for handling inputs for which no explicit training has taken place.

References

[1] J. Chang and J. J. Vidal, "Inferencing in hardware," in *Proc. MCC-Univ. Res. Symp.*, Austin, TX, July 1987.

[2] L. S. Haynes, R. Lau, D. Siewiorek, and D. Mizell, "A survey of highly parallel computing," *Computer*, vol. 15, no. 1, pp. 9–24, 1982.

[3] J. J. Helly, W. V. Bates, and S. Kelem, "A representational basis for the development of a distributed expert system for space shuttle flight control," NASA Tech. Memo. 58258, May 1984.

[4] K. Hwang, "Advanced parallel processing with supercomputer architectures," *Proc. IEEE*, vol. 75, pp. 1348–1379, 1987.
[5] T. Kohonen, *Self-Organization and Associative Memory*. New York: Springer-Verlag, 1984.
[6] T. R. Martinez, "Adaptive self-organizing logic networks," Ph.D. dissertation, Tech. Rep. CSD 860093, Univ. Calif., Los Angeles, May 1986.
[7] ——, "Adaptive self-organizing concurrent systems," in *Progress in Neural Networks*, vol. 1, O. Omidvar, Ed. Norwood, NJ: Ablex, , ch. 5, pp. 105–126, 1990.
[8] T. R. Martinez and J. J. Vidal, "Adaptive parallel logic networks," *J. Parallel and Distributed Computing*, vol. 5, no. 1, pp. 26–58, 1988.
[9] T. R. Martinez and D. G. Campbell, "A self-adjusting dynamic logic module," *J. Parallel and Distributed Computing*, vol. 11, no. 4, pp. 303–313, 1991.
[10] T. R. Martinez, "Self-organizing binary decision trees: Towards VLSI connectionist computing," *Int. J. Computer-Aided VLSI Design*, to be published.
[11] ——, "Neural network applicability: Classifying the problem space," in *Proc. IASTED Int. Symp. Expert Systems and Neural Networks*, pp. 41–44, Aug. 1989.
[12] F. Rosenblatt, *Principles of Neurodynamics*. Washington, DC: Spartan Books, 1962.
[13] G. Rudolph and T. R. Martinez, "DNA: Towards an implementation of ASOCS," in *Proc. IASTED Int. Symp. Expert Syst. and Neural Networks*, pp. 12–15, Aug. 1989.
[14] D. Rumelhart and J. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. I. Cambridge, MA: MIT Press, 1986.
[15] R. A. Verstraete, "Assignment of functional responsibility in perceptrons," Ph.D. dissertation, Comput. Sci. Dept., Univ. Calif., Los Angeles,, June 1986.
[16] S. S. Yau and C. K. Tang, "Universal logic circuits and their modular realizations," in *AFIPS Conf. Proc.*, vol. 32, pp. 297–305, 1968.
[17] C. Barker and T. R. Martinez, "Proof of correctness for ASCOS AA3 networks," to be published.

Decisions with Probabilities over Finite Product Spaces

Michael Pittarelli

***Abstract*—Techniques for decision making with probabilities over finite product spaces are discussed. In general, the type of decision problem generated by the available probabilistic information is one of decision under partial uncertainty: the probability distribution over the event space for the problem is determined only to the extent that it is contained in a convex polyhedron of distributions. The structure of this set makes computationally feasible the application of any of the various criteria for decision making with indeterminate probabilities that have appeared in the literature. Algorithms are developed for economically reducing the size of sets guaranteed to contain the unknown distribution over the event space for a given problem, thereby improving the quality of the decision made using any criterion.**

I. Introduction

Over the past 30 years, many authors have found it useful to classify decision problems into three types: decision under risk, decision under uncertainty, and decision under partial uncertainty. Partial uncertainty is usually characterized as involving knowledge of a set of distributions K such that any element p of K is potentially the "actual" but unknown probability distribution over the set of events (states, conditions) in terms of which the problem is defined. However, the probabilistic information available for decisions under risk and uncertainty may also be characterized in this way. For decision under risk, the set K is a singleton:

$$K = \{p\}.$$

For decision under uncertainty, K is the entire simplex of probability distributions over the event space of the problem. This perspective facilitates a unified approach to decision making using various forms of probabilistic information.

Here, decision problems are considered with the following components: a set of possible acts $A = \{a_1, \cdots, a_m\}$, a set of possible states (or events) $S = \{s_1, \cdots, s_n\}$, and a utility function $u : S \times A \to \mathbf{R}$, where S is the Cartesian product of a (finite) set of finite variable domains:

$$S = \mathrm{dom}(V_o) = \times_{\nu \in V_o} \mathrm{dom}(\nu).$$

Probabilities $p(s_j)$ are determined only to the extent to which they may be inferred from a given collection of marginal probability distributions defined on Cartesian products of the domains of subsets of some set V of variables such that, usually, $V_o \subseteq V$.

In general, the marginals determine a nonsingleton proper subset K of P_{V_o}, the set of all possible distributions over S (unconstrained by compatibility with any given set of marginals), any member of which could be the actual distribution. Thus, the type of decision problem under consideration is one of decision making under partial uncertainty proper, although it may in special cases (due either to the structure over which probabilities are available and its relation to the set S or to the quantities involved) reduce to one of decision making under risk or under uncertainty.

The restriction that the set S of relevant events be a Cartesian product of finite variable domains is not as severe as it might at first seem. The variables $\nu \in V_o$ may represent attributes in terms of which a collection of entities is classified, as in the relational model of data [12] or in statistical analysis of categorical variables [5].

The collections of marginals available for a given decision problem are themselves treated as *probabilistic databases* [2] here. They are manipulated via an algebra analogous to the relational algebra for relational databases in order to reduce the size of the set K of distributions over the set V_o compatible with the given marginals, while at the same time keeping the computational expense reasonably low. The effect of the reduction in size of K is to increase the likelihood that a single best action (relative to many different types of criteria) will be identified, thereby overcoming to the extent possible what may be viewed as the "imperfection," relative to the given decision problem, of the available data [14].

A simple algebra of probability over finite product spaces is presented in Section II.

II. Notation and Algebra

A set of variables $V = \{\nu_1, \cdots, \nu_k\}$ over which a probability distribution p is defined is referred to as the *scheme* for p.

A collection $P = \{p_1, \cdots, p_m\}$ of probability distributions over schemes of finite variables will be referred to as a (probabilistic) *database* [2]. The database $P = \{p_1, \cdots, p_m\}$ has the *structure* $X = \{V_1, \cdots, V_m\}$, where V_i is the scheme for distribution p_i. Let $\mathrm{dom}(X) = \mathrm{dom}(V_1 \cup \cdots \cup V_m)$. (As in relational database theory, some arbitrary but fixed linear ordering on the variables is assumed.)

Manuscript received April 1, 1990; revised April 12, 1991.

The author is with the Department of Computer and Information Science, State University of New York Institute of Technology, Utica, NY 13504-3050.

IEEE Log Number 9101212.