To appear in *International Journal of Neural Systems*

LIA: A Location-Independent Transformation for ASOCS Adaptive Algorithm 2

George L. Rudolph and Tony R. Martinez
Computer Science Department, Brigham Young University, Provo, Utah 84602

Abstract—**Most Artificial Neural Networks (ANNs) have a fixed topology during learning, and often suffer from a number of short-comings as a result. ANNs that use dynamic topologies have shown ability to overcome many of these problems. Adaptive Self Organizing Concurrent Systems (ASOCS) are a class of learning models with inherently dynamic topologies. This paper introduces Location-Independent Transformations (LITs) as a general strategy for implementing learning models that use dynamic topologies efficiently in parallel hardware. A LIT creates a set of location-independent nodes, where each node computes its part of the network output independent of other nodes, using local information. This type of transformation allows efficient support for adding and deleting nodes dynamically during learning. In particular, this paper presents the Location-Independent ASOCS (LIA) model as a LIT for ASOCS Adaptive Algorithm 2. The description of LIA gives formal definitions for LIA algorithms. Because LIA implements basic ASOCS mechanisms, these definitions provide a formal description of basic ASOCS mechanisms in general, in addition to LIA.**

1. INTRODUCTION

Hardware support for ANNs is important for enabling real-time handling of large, complex problems, in particular for enabling real-time learning. Learning times can exceed tolerable limits for applications of large size and complexity using conventional computing schemes. As hardware is becoming cheaper and easier to design, viable hardware mechanisms that directly support neural computation, especially ANNs with dynamic topologies, can be explored and implemented.

The *Location-Independent Transformation* (LIT) is a general implementation strategy for ANNs that overcomes several weaknesses of current hardware implementation methods. This strategy maps an ANN onto a parallel, hierarchical network, providing bounded logarithmic broadcast and gather times for an ANN with either a static or dynamic topology (see below). The purpose of LIT is not to invent new or improved learning algorithms, but rather to support existing algorithms in an efficient way for hardware implementation.

This paper presents an LIT for ASOCS (Adaptive Self-Organizing Concurrent Systems) model AA2 [12-14], which has an inherently dynamic topology. A complete, formal description of an LIT transformation includes the following:

- giving the mapping from the original network onto the LIT architecture,
- providing formal execution and learning algorithms (for those interested in potential implementation and for formal verification of the mapping), and
- giving a complexity analysis of the formal algorithms.

Initial LIT implementation research for ASOCS precedes that of other LIT models (see below). However, a complete, formal description of the LIT for AA2 had not previously been published, due to the constraints of on-going research. This paper provides this formal treatment.

Unlike ASOCS models, most ANNs use static topologies—the topology is fixed initially, and remains the same throughout learning. ANNs with static topologies often suffer from the following short-comings:

- sensitivity to user-supplied parameters: learning rate(s), etc.,
- local error minima during learning,
- there is no mechanism for deciding on an effective initial topology (number of nodes, number of layers, etc.).

Current research is demonstrating the use of dynamic topologies in overcoming these problems [2-5], [10-11], [14], [16-17], [19]. A *dynamic topology* is one which allows adding and deleting both nodes and weighted connections during learning.

† This research is funded in part by grants from Novell Inc. and Word Perfect Corp.

Early ANN hardware implementations are model-specific, and are intended to support static topologies [6-7], [16]. More recent *neurocomputer* systems have specialized neural hardware, and seek to support more general classes of ANNs [8], [18], [23]. Although some neurocomputers could potentially support dynamic topologies more directly in hardware, rather than in software, they currently do not. Of course, general parallel machines, like the Connection Machine [9] and the CRAY [1], can simulate the desired dynamics in software, but these machines are not optimized for neural computation. LIT supports general classes of ANNs *and* dynamic topologies in an efficient parallel hardware implementation.

LIT maps the original network into a hierarchical, parallel network of Location-Independent nodes (LI-nodes). A *LI-node* contains enough information locally to compute its function independent of any other node in the network. The set of LI-nodes is then embedded in a uniform tree topology. Figure 1 gives a general view of LIT. Execution (running the network) and learning are based on efficient broadcast/gather operations in the tree. Because the nodes are location-independent, they can be placed anywhere in the tree, and dynamic topologies can be efficiently and easily supported: Adding nodes is as simple as allocating free nodes in the tree; deleting nodes is as simple as de-allocating the desired nodes.

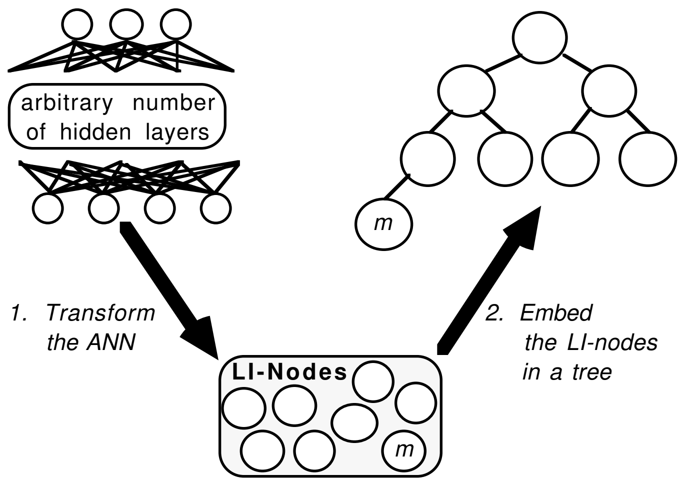


Figure 1. General LIT Transformation

As suggested above, LIT is a generalization of the authors' work on a strategy for implementing in hardware the ASOCS class of models [12-14], in particular AA2 [14], which have inherently dynamic topologies. The initial results of that work were the Location-Independent ASOCS model (LIA) [20] and the Priority ASOCS model [15], [24]. While the underlying mechanisms of ASOCS may differ significantly from those common among ANNs, the overall goals of ASOCS are similar to other such learning models, as is the need to support dynamic topologies. Hence, the authors have applied and expanded the initial strategy to more common ANNs. This led to the development of LITs for counterpropagation, competitive learning [21], and backpropagation with an arbitrary number of layers [22]. LITs can potentially support a broad set of ANNs, thus allowing one efficient implementation strategy to support both inherently dynamic ANNs and variations of ANNs with dynamic extensions (such as backpropagation).

This paper presents the LIA model as a LIT for AA2, developing the initial ASOCS results within the context of the expanded LIT research. Section 2 presents a general overview of LIT. Section 3 gives the LIT transformation from AA2 to LIA. Section 4 presents formal definitions of the LIA execution and learning algorithms. Section 5 gives a complexity analysis of the algorithms in section 4. Section 6 is the conclusion.

2. THE TRANSFORMED NETWORK

A transformation maps the original network onto an LIT network such that each node (in the LIT network) contains enough information locally to compute its part of the network output, independent of any other node. A network whose nodes have this property is said to be *location-independent.* The nodes also are location-independent—regardless of the physical location of any node in the network, the relative order in which they compute results, or the order in which those results are gathered, *the individual computations are the same, and the network output is the same*. Furthermore, because a node's information is local, adding or deleting nodes from the network can be done without affecting any other nodes. Thus, location-independence allows efficient support for dynamic topologies. Intuitively, the transformation is applied to the initial topology of the original network, before any learning has taken place; however, it can be viewed as being applied to some "snapshot" of the network, at any time, and the process is still the same.

In this paper, the term *Control Unit* refers to a mechanism that broadcasts inputs to a network, and gathers results from it. The term *original* refers to a network before it is transformed, and the term *transformed* refers to a network after it has been transformed. These terms apply similarly to the nodes as well. The number of layers refers to the number of *weight layers*.

LIT is a two-step process:

1. Construct a set of location-independent nodes based on the original model.
2. Embed the nodes in a tree.

This process is outlined in figure 1. Based on an original ANN model (left), a set of LI-nodes is constructed (middle), and the nodes are embedded in a tree (right). The construction of LI-nodes is essentially determined by how the computations and behaviors of the original network can be executed efficiently in the transformed network. Consequently, the details of the construction/transformation differ across models. It seems helpful, in determining the efficacy of a proposed mapping, to express the original equations in an equivalent form that reflects the resulting structure and behavior of the transformed network. Formal definitions of such "transformed" equations also aid in precisely specifying network behaviors for implementation. An example transformation of an AA2 network to an equivalent LIA network is given in the first part of section 3. Transformed equations are given in section 4.

A communication topology can be chosen as a matter of speed, cost and reliability. The same topology can be used with many models. A binary tree topology is specified in the second step in figures 1 and 2. It is a simple, fast, regular, hierarchical topology that naturally supports broadcast and gather operations. It provides a basis for applying the transformation and explaining the associated execution and learning algorithms, without obscuring the details with a complex interconnect. However, a hypercube, or some other more fault-tolerant topology, could also be used. Hierarchical topologies allow adding or deleting nodes with time logarithmic in the number of nodes in the network, while limiting connectivity among nodes. The connectivity of a binary tree does not explode because each node is connected to one parent and zero to two children. The algorithms for the transformed network show explicitly how the required communication can take place. The example AA2 transformation in the next section provides an example of how LIT can support dynamic topologies.

3. TRANSFORMATION OF AA2 NETWORKS TO LIA NETWORKS

This section begins with a description of ASOCS ideas and AA2. Due to space constraints, this description is brief; the reader may refer to the ASOCS literature for further detail, if desired. The ASOCS description is followed by the informal discussion of the transformed network, and then by the formal algorithms.

Assume a logical universe composed of two disjoint sets of Boolean variables: 1) input variables and 2) one or more output variables. (Feedback variables are discussed in the ASOCS literature, but not in this paper.) An *instance* specifies a mapping of some subset of the input variables onto a single output variable. (Multiple output variables may be specified in a network, however each variable is assumed to be independent.) The semantics of an instance are that when the conjunction of input variables is true, the output variable specified in the instance should be set to the value specified by the instance. For example, the instance

$AB'C \Rightarrow Z$

asserts that when A is high, B is low (negation is denoted by the prime " ' ") and C is high, Z should be set high.

The basic operational features of an ASOCS system are summarized as follows:

- An ASOCS network (including LIA) learns new instances incrementally, as they are presented to the network, over time.
- The network adapts in a self-organizing fashion to represent a modified instance set, which is consistent with the new instance being added and parsimonious, whenever a new instance causes modifications to be necessary.
- During execution mode, the network takes the full input environment and computes the appropriate output for those inputs, based on the instances stored in the learned instance set.

Consistency is maintained by adding discriminant variables to "stored" instances that conflict with the new instance being added or, if discrimination is not possible, entirely deleting the stored instance. Potential conflicts exist if a stored instance and the new instance specify different outputs for some set of input states. A *discriminant variable* guarantees that a stored instance and the new instance are matched by mutually exclusive sets of input states.

Parsimony is maintained by eliminating redundant instances from the stored instance set. Redundancy is determined by pairwise comparison of instances. Pairwise minimizations are relatively fast approximations to finding globally minimal representations. Algorithms for achieving more minimal instance set representations (such as using background minimization) are a subject for future research, perhaps in connection with on-going research in the area of generalization algorithms.

Where ASOCS models, such as AA1, AA2, AA3, LIA and PASOCS, differ is in how each one represents an instance set internally (in a network), and consequently, in the details of how each network learns or adapts. However, all ASOCS models implement the concepts of instances and instance sets in some fashion, as outlined above.

Figure 2 shows an example of an AA2 [14] network transformed to an equivalent LIA network. The numbers accompanying particular nodes are for reference only. The AA2 network is composed of two types of nodes, 1) PNodes, and 2) DNodes. PNodes are the interior nodes in the network, the building blocks from which representations of instances are built in a hierarchical fashion. Each PNode represents the conjunction of its two inputs (represented by the letters inside each node in figure 2). The two inputs to a PNode may be either of two types: an input variable (or its negation), or the output of another PNode. The output of a PNode is Boolean, and the output of any given PNode may be connected (and thus shared) as input to one or more other PNodes or DNodes.

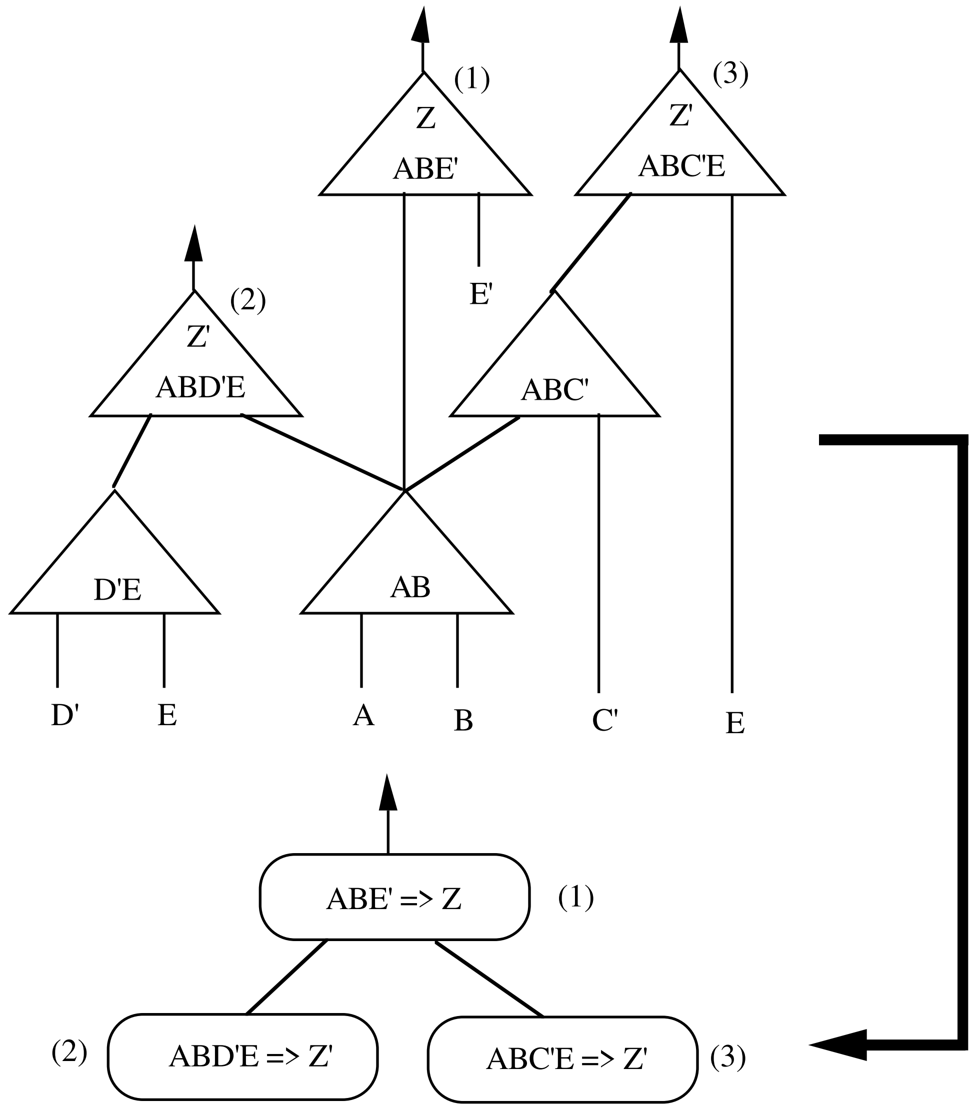


Figure 2. An Example AA2 network transformed to LIA

DNodes are the leafnodes in an AA2 network (i.e. nodes 1, 2 and 3). Like a PNode, each DNode represents the conjunction of its inputs, and the two inputs of a DNode may be either a variable (or its negation) or the output of a PNode. Logically, however, a DNode also represents one of the instances in the instance set currently stored in the network. In other words, there is one DNode for each instance stored in the network, and each Dnode determines the output of the network when the instance it represents is matched. The output value of the network for a DNode is indicated by the Z or Z', respectively, at the top of each Dnode in figure 2. The output of a DNode is Boolean—the actual output of the network is determined by whether the output of the DNode is connected to the positive collector (think of each collector as an OR-plane) or negative collector of the output variable. Z therefore represents the positive collector for output variable Z, while Z' represents the negative collector for variable Z.

This brief overview of AA2 concludes with two further observations: First, a network outputs a ? ("don't know" value) in the case where none of the instances is matched, and therefore every DNode outputs a 0, and therefore both the positive and negative collectors output a 0. Second, the instance set stored in a network is changed by adding and deleting PNodes and

Dnodes, and by altering connections between PNodes, Dnodes and the connections to output collectors.

There is one node in the LIA network for each Dnode in the AA2 network. Each transformed node stores the instance represented by the corresponding AA2 Dnode (as shown in figure 2). Another way of looking at the transformation is this: A transformed node can store a complete instance; each node in the transformed network stores one instance in the instance set represented by the AA2 network. So, LIA can be seen either as a LIT transformation (implementation) of AA2, or as a different way of representing an instance set, whichever is convenient.

Suppose that a new instance

AB'F ⇒ Z'

is to be added to the AA2 network of figure 2. None of the possible conjunctions AB', AF or B'F already exist as PNodes, so a new PNode representing one of them, AB' for example, would be added to the network. In that case, a new DNode with inputs from the new PNode and the variable F would also be added to the network. The output of the DNode would be connected to the negative collector for Z.

The same new instance is added to the LIA network of figure 2 by allocating a free node in the network, the left child of node 2 for example, and storing the new instance in that node.

Now suppose the instance

ABD'E ⇒ Z'

is to be deleted from the AA2 network of figure 2. DNode 2 is deleted, along with its connections to the negative collector and any PNodes or variables to which it is directly connected—in this case PNodes representing D'E and AB. In addition, since the PNode representing D'E is not shared by any remaining instances, it is deleted from the network.

The same instance is deleted from the LIA network by marking node 2 as "free". No other nodes in the LIA network are affected.

A 2-input AA2 node (either Pnode or Dnode) may or may not actually store its variable list(s) at each node, depending on the specific implementation. An LIA node, on the other hand, stores two lists, or vectors of variables: One is the left-hand side of an instance $\boldsymbol{V_j}$. Each element v_{ij} has the value *, or the value of the corresponding variable of the instance that the node stores. The other vector is the right-hand side of an instance $\boldsymbol{R_j}$. It has ? values in every element r_{kj} except the one corresponding to the output variable for that instance, in which case the value is 1 or 0, depending on the polarity of that variable. Even though each instance only has one right-hand side variable, it is easier to visualize (and perhaps handle) how multiple output variables are gathered in parallel if the output of each node is a vector rather than a scalar. Figure 3 shows a conceptual view of an LIA node.

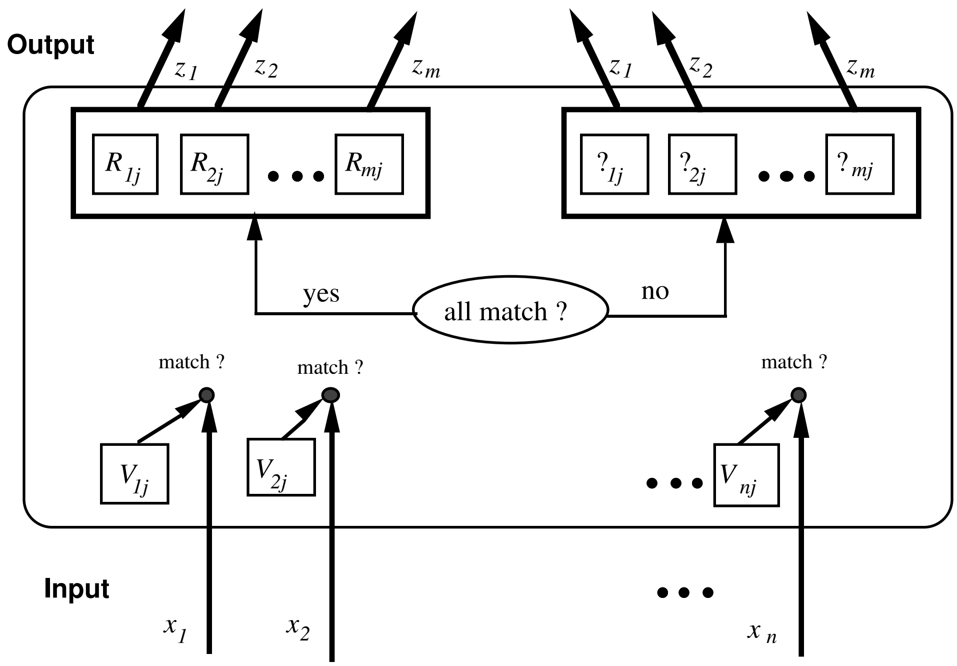


Figure 3. Conceptual view of LIA node.

3.1. LIA Execution Mode

The goal of execution mode is to determine what is the output of the network for a given set of inputs. More precisely, it is to determine which nodes in Ω match a given state or set of input values $\boldsymbol{X}$, and thus determine the output of the network. This is done in two steps, broadcast and gather. During the broadcast step, $\boldsymbol{X}$ is broadcast down the tree to the nodes and each node j compares each element x_i of $\boldsymbol{X}$ with the corresponding element v_{ij} of $\boldsymbol{V_j}$ (the left-hand-side of β_j the instance stored at node j) to see if the two vectors "match". In this matching function, a 1 matches a 1, a 0 matches a 0, and * ("don't care") matches either 1 or 0. The domain of x_i is Boolean, while the domain of v_{ij} is Boolean and *. A node that is matched will assert the right-hand side $\boldsymbol{R_j}$ of its β_j as output $\boldsymbol{O_j}$, whereas a node that is not matched (there exists at least one mismatch) will assert a vector of ? ("don't know") values as output $\boldsymbol{O_j}$. The latter is equivalent to the node itself not responding to some set of input values.

During the gather step, each node combines the elements of its asserted output $\boldsymbol{O_j}$ with the corresponding values received from its children, and passes the result $\boldsymbol{Y_j}$ up to its parent. The parent of the root node is the CU, and thus $\boldsymbol{Y_j}$ from the root node becomes the output of the network $\boldsymbol{Z}$.

The underlying purpose of the gather function is one of transmission—transmitting output values from the nodes to the CU. Thus, the general sense of the gather function at each node is that of an *OR*, namely *OR*ing together the respective output elements from all the nodes to determine a single value for each output variable, and the $\vee$ symbol is used to denote this operation. However, the semantics of the gather function are somewhat different from *OR* as it is generally understood, even beyond the fact that the gather function here is extended to handle ? values. These semantics can be explained in terms of a dyadic operation using one output variable o_{kj} and the corresponding output variable from one child y_{kjh}, however the operation extends to any number of variables (or children). The gather operation is defined as follows:

o_{kj}	y_{kj}	result
?	?	?
?	0	0
?	1	1
0	?	0
0	0	0
0	1	impossible case
1	?	1
1	0	impossible case
1	1	1

Table 1. Definition of gather operation

The idea is that the indefinite value ? is replaced by a definite 1 or 0 (as the case may be) when such is asserted as the output of the network for a variable. Note that although more than one node may match a given $\boldsymbol{X}$, the nodes will never have to combine the cases $0 \vee 1$ or $1 \vee 0$ for any variable, because the network is consistent—if any one node asserts a definite value for some output variable, no other node will contradict that value. (In a physical implementation, such a case might occur only if one or more nodes has been corrupted, which results in a system error in any case.)

Three cases can occur for execution mode for any single variable, given some input vector $\boldsymbol{X}$ and some network that stores instance set Ω: 1) no instances match, 2) exactly one instance matches, 3) more than one instance matches. In the first case, the network will output a vector of ? values. In the second case, the output of the network is clearly the output of that matched instance for that variable, since there is only one instance. The third case arises from the fact that two or more concordant instances may match the same $\boldsymbol{X}$, even though Ω is minimized, because such instances may overlap in their coverage of states.

In the AA2 literature, an AA2 node is represented as a 2-input AND-gate with a node CU controlling the inputs and output of the node. In execution mode, the network can be viewed essentially as a hired-wired hierarchy of 2-input AND-gates whose outputs flow into an OR-plane (the collectors). Likewise, a LIA node can be viewed as an n-input AND-gate with a node CU controlling the inputs and output(s) of the node. These views are conceptually correct, intending primarily to show how the nodes implement conjunctions of variables and the required mapping of inputs to outputs. However, Stout *et. al.* found these high-level views to be too simplistic and somewhat misleading, when attempting to design actual implementations [24]. The critical issue is that such a view ignores the control circuitry (switches, memory, etc.) needed to “set up” or “preprocess” and maintain the inputs to the AND-gates. Considering the functionality required by the overall AA2 system (in which the network is embedded) and the local CU at each node to compute with instances in an AA2 network, the true overall function is more akin to the equivalence (XNOR) function. The "matching" function used in LIA and PASOCS models is essentially an equivalence function extended to handle * values in the instances. This view was more convenient for the VLSI implementation of PASOCS in [24].

3.2. LIA Learning Mode

Let Ω_0 be the null, or empty instance set, which corresponds to a network that stores no instances. Let α_t be the *t*-th new instance added to a network which stores instance set Ω_{t-1} before

α_t is presented for learning, and stores Ω_t after learning α_t. For purposes of the discussing modifications that occur during learning, consider the initial value of Ω_t to be Ω_{t-1}—all potential modifications are discussed with reference to Ω_t, rather than Ω_{t-1}.

The goal of learning is to add α_t in such a way that Ω_t includes the mapping specified by α_t, is consistent with α_t, and parsimonious. The behavior of the network in learning mode is motivated by the search for β_j that are inconsistent with α_t the search for β_j that can be involved in minimizing Ω_t (assuming that α_t is added to Ω_t) and necessary adaptive actions of the network based on the results of these searches.

These ideas can be summarized by the following semi-formal definition of instance set learning:

$$\Omega_0 = \varnothing$$

$$\Omega_t = \begin{cases} \Omega_t, \textit{ iff } \exists\ \beta_j \textit{ that covers } \alpha_t \\ \Omega_{t-1} \cup \alpha_t \cup \{\textit{all inconsistent } \beta_j \textit{ made consistent}\} - \{\textit{all redundant } \beta_j\} - \{\textit{all inconsistent } \beta_j\} \end{cases}$$

The searches and resulting actions are based around the fact that α_t, partitions Ω_t into disjoint subsets based on the relationship of each β_j to α_t. The right-hand side of any β_j must be Concordant, Discordant, or Notrelated with respect to the right-hand side of α_t. By definition, each of these subsets of Ω_t is mutually exclusive. The left-hand side of any β_j must be Subset, Equal, Superset, Overlap, or Discriminated with respect to the left-hand side of α_t. By definition, each of these subsets of Ω_t is also mutually exclusive. Thus, the combination of left-hand side and right-hand side relationships results in a partition.

There are 15 subsets in the partition of Ω_t, but only seven of these cause adaptive actions and are therefore of interest during learning. The eight which do not are those which contain any β_j that is Notrelated, is Discriminated, or is Concordant Overlap. With the exception of Notrelated, which should be obvious, the others are discussed in more detail in [12] and other ASOCS literature. Of the remaining seven subsets: four would contain any possible inconsistent β_j, those being Discordant Subset, Equal, Superset or Overlap; three would contain any instance that could possibly be used to minimize Ω_t, those being Concordant Subset, Equal or Superset. Note that minimization as it is used here refers to simple, fast, pairwise minimizations only. Other minimizations are briefly discussed in section 5.3.

The adaptive actions for each of these seven subsets are as follows:

Concordant Subset or Equal: If either of these partitions are non-empty, then there exists at least one β_j that covers all possible input states that α_t covers, and possibly more, and has the same output as α_t. Consequently, adding α_t to Ω_t would be redundant. Furthermore, if this is true, all other partitions are empty, otherwise Ω_t is not consistent and minimal to begin with. Hence α_t is not added, and learning terminates for this instance.

Concordant Superset: If this partition is non-empty, then there exists at least one β_j which is redundant with α_t because α_t covers all the states that β_j covers and more, and they have the same output. Any such β_j can self-delete, and α_t is added to Ω_t.

Discordant Equal or Superset: If either of these partitions are not empty, then there exists at least one β_j which completely contradicts α_t because α_t covers all the states that β_j covers and more, and they have opposite output. The contradiction can be properly resolved by removing β_j. Thus, any such β_j can self-delete, and α_t is added to Ω_t.

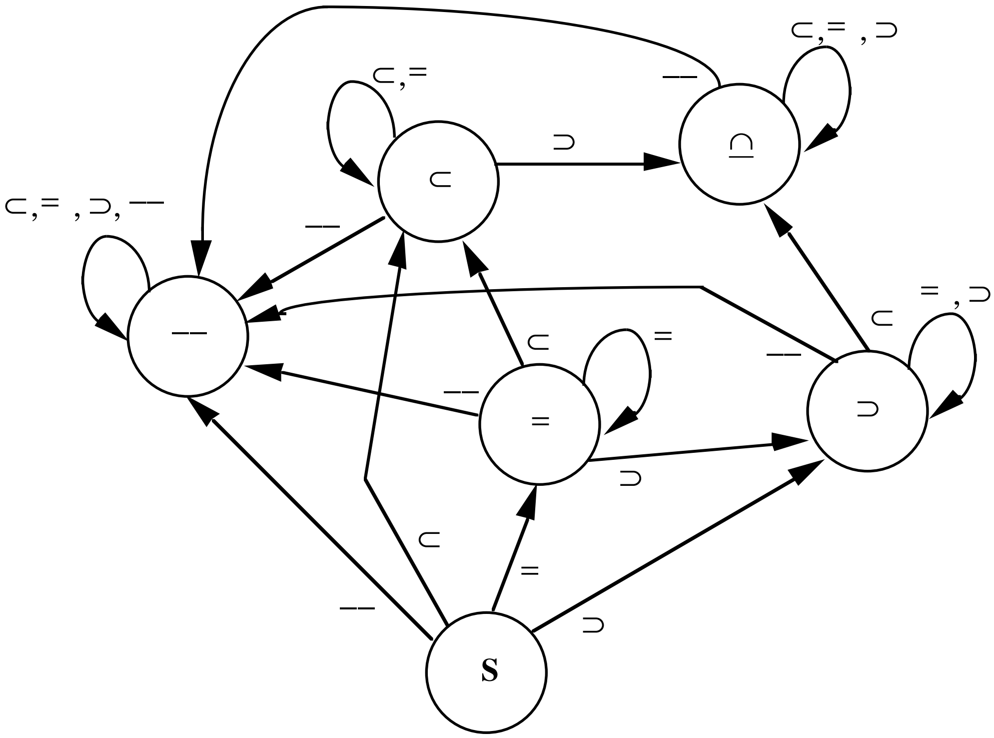
Discordant Subset or Overlap: If either of these partitions are not empty, then there exists at least one β_j which is partially contradicted by α_t: Either β_j covers all the states that α_t covers and more, or β_j covers some states that α_t covers and some states that α_t does not, and they have opposite output. The contradiction can be properly resolved by specializing each contradictory β_j so that it no longer covers contradictory states, but still covers all non-contradicted states as before. This specialization process is called *Discriminant Variable Addition* (DVA).

By definition, inconsistencies exist between α_t and β_j because their left-hand sides contain no discriminant variable(s). The most logical way to specialize β_j with discriminant variables is to find all the variables in $\boldsymbol{V}$ (the left-hand side of α_t) that are not in $\boldsymbol{V_j}$, invert their values (so they discriminate from α_t), add each one to the left-hand side of a copy of the original β_j, delete the original β_j, and add these modified instances to Ω_t. α_t is also added to Ω_t.

The following list summarizes the relationships that cause adaptive actions and their associated actions:

Relationship	**Action**
Concordant Subset	exit (already learned)
Concordant Equal	exit (already learned)
Concordant Superset	node self-deletes
Discordant Superset	node self-deletes
Discordant Equal	node self-deletes
Discordant Subset	DVA
Discordant Overlap	DVA

Learning assumes that the instance set Ω_t is consistent and parsimonious before α_t is presented for learning. As α_t is broadcast to the nodes, each node receives the broadcast, passes it on to its children (if any) and computes its relationship to α_t. Figure 4 shows a sequential state-flow diagram that indicates how the relationship between the left-hand sides of two instances can be determined. The diagram uses only single-variable comparisons, one comparison per transition. The comparison can also be done in parallel blocks depending on the width of the data path, as in [24].



S is the starting state; all other symbols represent states and /or single-variable comparisons as follows:

⊂ is subset
= is equal
⊃ is superset
$\underline{\cap}$ is overlap
— is discriminated

Figure 4. A sequential view of how the relationship between two instances can be determined using only single variable comparisons.

When all nodes have determined the relationships, the CU begins checking for the adaptive actions as outlined above. The CU broadcasts a "covered" flag, for which each node computes a 1 if it covers α_t (the node's β_j is subset or equal) and is concordant, and a 0 otherwise. Since the result is Boolean, the results can be gathered at each node using the Boolean OR, and the result passed up to each node's parent. If the CU receives a 1, then there exists at least one node (there may be relatively minimal overlapping nodes that respond) which properly covers α_t. In other words, α_t has already been learned. Since Ω_t is already consistent and parsimonious, no further modification is needed to generate Ω_t, therefore the network can proceed to the next instance. If the algorithm does not exit at this point, then α_t will be added to Ω_t once any and all other adaptive actions have been completed.

If α_t has not been learned, the next step is to check for nodes that can self-delete. The CU broadcasts the self-deletion flag. Any node that can self-delete (i.e. any superset instance and any equal discordant instance) does so, and returns a 1, while all others return a 0. As before, each node combines (Boolean OR) its result with those of its children, and passes the result up to its parent. The result of this particular gather is not necessarily needed, except to indicate whether any node(s) in fact self-deleted. The reason for doing self-deletion at this point in the algorithm is that any nodes which self-delete may potentially be re-allocated during DVA, if necessary. This makes more efficient use of the nodes. This differs from the AA2 algorithm of [12]. There, self-deletion takes place *after* DVA, in order to take advantage of any pre-existing Pnode hierarchies for node combination in adding properly discriminated instances to the network. Note that self-deletion effectively deletes redundant instances and totally contradicted instances.

The next step is to check for partially contradicted instances (i.e. subset or overlap discordant instances), and execute DVA if needed, as outlined above. This is accomplished as follows: The CU broadcasts the DVA flag. Each partially contradicted node returns a 1, while all others return a 0. If the CU receives a 1 from the gather of these results, it then broadcasts a flag which permits the required nodes to execute DVA. Each such node j now computes the list of discriminant variables S from α_t, one of which it adds to its left-hand side $\boldsymbol{V_j}$ (this is equivalent to deleting the original β_j and replacing it with one of the specialized instances), deleting it from S. If S is not empty after this, the node splits S in half and sends a copy of the original β_j with half of S to the right, and a copy of the original β_j with the other half of S to the left (if S is odd, the "extra" variable can be included in either the left or right half of S). If a child node is *not* empty when it receives this DVA list, it will split the S it receives and simply pass these on, with copies of the original β_j, as did the original parent. If however, a child is free, it will allocate itself, store the original β_j, and add one variable from the S it receives to its left-hand side, deleting that variable from S. The node then proceeds with the original β_j and its remaining S as did the original parent. DVA terminates at the node where there are no more variables to send. DVA terminates for the network when all instances created by DVA have been stored in a node.

DVA can be executed in parallel [with complexity $O(\log m)$, where m is the number of nodes in the network] in the tree because, for any two nodes at the same level of the tree, DVA lists are sent down mutually disjoint subtrees. This remains true as long as it is assumed that there are always enough nodes in the branches of the subtree of the original contradictory node to store instances required by DVA. If there are not enough nodes in a subtree, then either DVA cannot be completed until more nodes are added where required, or the DVA lists must be passed back up the tree and rerouted down subtrees where free nodes exist.

There are several methods for handling the case where there are insufficient nodes in a particular subtree (but sufficient nodes in the network). Sequentially selecting and adding each remaining instance has complexity $O(k\log m)$, where m is the number of nodes in the network and k is the number of instances remaining to be added as a result of DVA. Other, potentially more parallel and more localized rerouting algorithms could be used instead of the sequential method. Such parallel methods would seem to have faster average times (than the sequential method), but also risk a worst case search that is linear in the number of nodes in the tree.

Once contradictions have been resolved (if there were any), a free node is selected from any remaining free nodes in the network, and α_t is stored in it. The learning of α_t is complete, and the network may proceed to the next instance.

4. FORMAL EXECUTION AND LEARNING ALGORITHMS

4.1 Execution Algorithm

Definitions

- = used either as assignment operator or as the equivalence operator, as appropriate [see equation (4)].
- * "don't care" value. Only occurs in the left-hand-side V_j of an instance.
- ? "don't know" value. Only occurs in the right-hand side of an instance $\boldsymbol{R_j}$, and thus possibly in $\boldsymbol{O_j}$, $\boldsymbol{Y_j}$ and $\boldsymbol{Z}$.
- *n* number of currently defined inputs to the network, also the size of $\boldsymbol{X}$. If "don't care" variables are counted in each node, it is also the size of $\boldsymbol{V}$.

p	number of currently defined outputs of the network, also the size of $\boldsymbol{R_j}$, $\boldsymbol{O_j}$, $\boldsymbol{Y}$ and $\boldsymbol{Z}$, if "don't know" variables are counted. For most ASOCS descriptions, p=1. In this paper, $p\geq 1$.
w_j	the number of children of node j.
$\boldsymbol{X}$	input pattern presented to the network.
x_i	ith Boolean component of $\boldsymbol{X}$.
$\boldsymbol{V_j}$	left-hand-side of the instance stored in node j, has size n if "don't care" values are counted.
v_{ij}	ith component of $\boldsymbol{V_j}$. It has domain {0,1,*}.
$\boldsymbol{R_j}$	right-hand-side of the instance stored in node j, has size p if "don't know" values are counted.
r_{kj}	kth component of $\boldsymbol{R_j}$. It is also the output of node j with respect to output variable z_k. It has domain {0,1,?}.
$\boldsymbol{O_j}$	individual output vector of node j—it is either $\boldsymbol{R_j}$ or "don't know", depending on the value of $match_j$.
o_{kj}	kth component of $\boldsymbol{O_j}$. It has domain {0,1,?}.
m	number of D-nodes/instances in the original AA2 network and the number of nodes in the LIA network.
$match_j$	Boolean value for node j that is 1 if $\boldsymbol{V_j}$ covers $\boldsymbol{X}$, and 0 otherwise.
$\boldsymbol{Y_j}$	vector of intermediate network output values computed by node j.
y_{kj}	kth component of $\boldsymbol{Y}$. It has domain {0,1,?}.
$\boldsymbol{Z}$	vector of network output values.
z_k	real-valued (always between 0 and 1) kth component of $\boldsymbol{Z}$. It corresponds in the original network to the output O_k of output node k.

<u>Equations</u>

$$y_{kj} = o_{kj} \vee \left(\bigvee_{h=1}^{w_j} y_{kjh} \right) \tag{1}$$

$$o_{kj} = ? \vee (match_j \wedge r_{ij}) \tag{2}$$

$$match_j = \bigwedge_{i=1}^{n} [(v_{ij} = *) \vee (v_{ij} = x_i)] \tag{3}$$

The steps of the execution algorithm are as follows:

1. Broadcast $\boldsymbol{X}$ to the nodes.
2. Each node does the following:
   - 2.1 Compute $\boldsymbol{O_j}$.
     - 2.1.1 Determine if $\boldsymbol{V_j}$ matches $\boldsymbol{X}$.
     - 2.1.2 $match_j$ is 1 if all variables match, 0 otherwise.
     - 2.1.3 The node contributes $\boldsymbol{R_j}$ to the output of the network if and only if $match_j$ is 1.
   - 2.2 Compute $\boldsymbol{Y_j}$.
   - 2.3 Send $\boldsymbol{Y_j}$ to the parent node.
3. $\boldsymbol{Y_j}$ from the root node is the output $\boldsymbol{Z}$ of the network.

4.2 Learning Algorithm

<u>Additional Definitions</u>

α_t — the t-th new instance presented for learning.

$\boldsymbol{V}$ — the left-hand side of α_t.

$\boldsymbol{R}$ — the right-hand side of α_t.

r_k — the kth component of $\boldsymbol{R}$, corresponding to z_k.

r_{kj} — the kth component of $\boldsymbol{R_j}$.

β_j — the instance stored at a node j, or "old instance". It has left-hand side $\boldsymbol{V_j}$ and right-hand side $\boldsymbol{R_j}$.

$\subset$ — proper subset

$=$ — equal sets

$\supset$ — proper superset

$\underline{\cap}$ — overlap

—— — discriminated

$covered_{kj}$ — flag computed at node j relative to α_t with right hand side z_k, which indicates whether node j covers α_t; it is 1 if so, 0 otherwise.

$delete_{kj}$ — flag computed at node j relative to α_t with right hand side z_k, which indicates whether node j should self-delete; it is 1 if so, 0 otherwise.

dva_{kj} — flag computed at node j relative to α_t with right hand side z_k, which indicates whether node j should execute DVA; it is 1 if so, 0 otherwise.

S — the set of variables in $\boldsymbol{V}$ which are not a node's $\boldsymbol{V_j}$; the variables in S are computed and used only for DVA.

$rconc_{kj}$ — an intermediate Boolean variable used to make the equations below more readable; it is 1 iff β_j is related to α_t for output variable z_k and they are concordant—otherwise, it is 0. (See equation 4.)

$rdisc_{kj}$ — an intermediate Boolean variable used to make the equations below more readable; it is 1 iff β_j is related to α_t for output variable z_k and they are discordant—otherwise, it is 0. (See equation 5.)

<u>Equations</u>

$$rconc_{kj} = (r_{kj} \neq ?) \wedge (r_{kj} = r_k) \tag{4}$$

$$rdisc_{kj} = (r_{kj} \neq ?) \wedge (r_{kj} \neq r_k) \tag{5}$$

$$\mathrm{covered}_{kj} = \begin{cases} 1, \text{ if } \mathrm{rconc}_{kj} \wedge (\mathbf{V_j} \subset \mathbf{V}) \\ 0, \text{ otherwise} \end{cases} \tag{6}$$

$$delete_{kj} = \begin{cases} 1, \text{ if } (r_{kj} \neq ?) \wedge (\boldsymbol{V_j} \supset \boldsymbol{V}) \\ 1, \text{ if } rdisc_{kj} \wedge (\boldsymbol{V_j} = \boldsymbol{V}) \\ 0, \text{ otherwise} \end{cases} \tag{7}$$

$$dva_{kj} = \begin{cases} 1, \text{ if } rdisc_{kj} \wedge [(\boldsymbol{V_j} \subset \boldsymbol{V}) \vee (\boldsymbol{V_j} \underline{\cap} \boldsymbol{V})] \\ 0, \text{ otherwise} \end{cases} \tag{8}$$

The steps of the learning algorithm are given below.

1. α_t is broadcast to the network.

2. Each node computes its relationship to α_t.
3. Determine if α_t is already covered by Ω_t.
   3.1 The CU broadcasts the covered flag.
   3.2 Each node j returns a 1 if β_j is *Concordant Equal*, or *Concordant Subset*, a 0 otherwise.
   3.3 If the CU receives a 1 from the network, exit.
4. Delete redundant and wholly contradicted information.
   4.1 The CU broadcasts the self-delete flag.
   4.2 Any node whose β_j is *Discordant Equal*, *Concordant Superset* or *Discordant Superset* self-deletes.
5. Resolve partial contradictions, if any.
   5.1 The CU broadcasts the DVA flag.
   5.2 Each node j returns a 1 if β_j is *Discordant Subset* or *Discordant Overlap*, and returns 0 otherwise.
   5.3 If the CU receives a 0, skip to step 6.
   5.4 Each node j that returns a 1 does the following:
      5.4.1 Compute the set difference $S = \boldsymbol{V} - \boldsymbol{V_j}$
      5.4.2 Choose a v in S, and delete it from S.
      5.4.3 Add $\sim v$ to the original $\boldsymbol{V_j}$.
      5.4.4 For each remaining v in S do the following:
         a. Choose a v and delete it from S.
         b. Allocate a free node in the network.
         c. Make the new node's variable list the original $\boldsymbol{V_j}$ concatenated with $\sim v$.
         d. The new node's R is the same as $\boldsymbol{R_j}$.
      end
6. Add α_t to the network.
   6.1 Allocate a free node.
   6.2 Store α_t in the new node.

end (learning)

4.3 Optional Network Minimizations

As in [14], the learning algorithm above omits optional minimization procedures that may lead to a smaller network. These procedures are optional because they do not affect the correctness, or consistency, of the network, but rather the size in terms of the number of defined input variables and the number of instances in the network. Furthermore, minimization requires rebroadcasting modified instances to the network, which adds to the complexity of learning and potentially causes further minimizations, affecting the speed of learning. This section provides details similar to those of sections 3.2 and 4.2 for readers who may be interested—further details can be found in the ASOCS literature. This section begins with definitions for one type of pairwise minimizations based on certain one-difference concordant instances, which were defined for the original ASOCS models. The definition of the term "one-difference" differs here and in [24] from previous ASOCS usage in slight, but fairly significant, ways. Following this, the idea of "background" minimization is briefly discussed as an alternative to minimization during active learning.

β_j is *one-difference* with respect to α_t iff $\boldsymbol{V_j}$ and $\boldsymbol{V}$ share exactly one discriminant variable δ_{kj}. For example, ABC' $\Rightarrow$ Z is one-difference with respect to ABC $\Rightarrow$ Z. These instances can be replaced with the single, shorter instance AB $\Rightarrow$ Z. Likewise, AB'C $\Rightarrow$ Z is one-difference with respect to AB $\Rightarrow$ Z. These two can be replaced by the pair AC $\Rightarrow$ Z and AB $\Rightarrow$ Z.

Relating this back to the discussion of section 3.2, the one-difference relationship is a subset of the discriminated left-hand side relationship/partition. To be precise, the previous discriminated relationship can be divided into two mutually exclusive relationships, one-difference (denoted by 1d-) and two-difference (denoted by 2d-). As before, none of the 2d-relationships cause adaptive actions (and hence minimization), and so are not of interest here. The 1d- relationship, however, may be divided into four mutually exclusive subsets: 1d-subset, 1d-equal, 1d-superset and 1d-overlap. While this adds 12 subsets to Ω_t (because they can be concordant, discordant, or notrelated on the right-hand side), only three of these subsets cause minimization: concordant 1d-subset, concordant 1d-equal and concordant 1d-superset. Of the other nine, four are notrelated, four are discordant discriminated and one, the concordant 1d-overlap, also cannot be pairwise minimized with any other instances.

Note that both 1d-overlap concordant and discordant relationships never induce adaptive actions, just like the 2d- instances. Therefore, there is no real point to distinguishing the 1d-overlap relationship from the 2d- ones. The state-flow diagram of figure 4, which can be modified to include 1d- relationships, helps illustrate this idea. From the overlap state, the "machine" can move directly to the discriminated state whenever a discriminated input comparison is encountered, as is currently shown—hence, an extra state for 1d-overlap relationships is not needed.

Concordant 1d-equal: If this partition is non-empty, then there exists at least one β_j such that the set of states covered by ($\beta_j \wedge \alpha_t$) is also covered by α_t with $\sim\delta_{ikj}$ dropped from $\boldsymbol{V}$. α_t with $\sim\delta_{ikj}$ dropped from $\boldsymbol{V}$ is rebroadcast to the network as if it were the original instance—this causes the now redundant β_j to self-delete, and allows any other potential minimizations to take place.

Concordant 1d-subset: If this partition is non-empty, then there exists at least one β_j such that the following is true: Choosing exactly one such β_j, there exists a subset P of the states covered by β_j for which the output of the network is the same as α_t and for which there is a state covered by α_t that differs only in the value of δ_{ikj}. For the states in P and their counterparts, the discriminant variable is therefore not needed. α_t with $\sim\delta_{ikj}$ dropped from $\boldsymbol{V}$ covers the same states as α_t and additionally the states in P. α_t with $\sim\delta_{ikj}$ dropped is rebroadcast to the network as if it were the original instance—β_j is now concordant overlap with respect to α_t, but any further potential minimizations can take place.

Concordant 1d-superset: This case is analogous to the 1d-subset case, but with α_t and β_j reversed. If this partition is non-empty, then there exists at least one β_j such that the following is true: For each β_j , there exists δ_{ikj} and a respective subset P of the states covered by β_j for which the output of the network is the same as α_t and for which there is a state covered by α_t that differs only in the value of δ_{ikj}. For the states in each P and their counterparts, the discriminant variable is therefore not needed. β_j with δ_{ikj} dropped from $\boldsymbol{V_j}$ covers the same states as β_j and additionally the states in P. Each modified β_j with its δ_{ikj} dropped is removed from Ω_t and rebroadcast to the network, potentially causing further minimizations. (α_t is not modified when it is added to the network as the new instance in this case.)

The following summarizes the 1d- relationships and their associated adaptive actions:

Relationship	**Action**
Concordant 1d-subset, or	

Concordant 1d-equal	if any such nodes, pick exactly one such β_j (*not* one from both partitions!) delete $\sim\delta_{ikj}$ from $\boldsymbol{V}$ rebroadcast the modified α_t to the network
Concordant 1d-superset	for all such nodes, delete $\sim\delta_{ikj}$ from $\boldsymbol{V_j}$ remove β_j from Ω_t rebroadcast the modified β_j to the network

<u>minimization definitions</u>

$-\subset$	one-difference (proper) subset.
-=	one-difference equal.
$-\supset$	one-difference (proper) superset.
δ_{ikj}	one-difference variable v_{ij} in β_j relative to α_t with right-hand side z_k.
min_ni_var$_{ikj}$	flag computed at node j relative to α_t with right hand side z_k; it is the one-difference variable δ_{ikj} if α_t can be minimized by dropping $\sim\delta_{ikj}$, ? otherwise.
min_is$_{ikj}$	flag computed at node j relative to α_t with right hand side z_k; it is 1 if β_j can be minimized by dropping δ_{ikj}, 0 otherwise.

<u>minimization equations</u>

$$min\_ni\_var_{kj} = \begin{cases} \delta_{ikj}, \text{ if } rconc \wedge ((\boldsymbol{V_j} - \subset \boldsymbol{V}) \vee (\boldsymbol{V_j} -= \boldsymbol{V})) \\ ?, \text{ otherwise} \end{cases} \tag{9}$$

$$min\_is_{kj} = \begin{cases} 1, \text{ if } rconc \wedge (\boldsymbol{V_j} - \supset \boldsymbol{V}) \\ 0, \text{ otherwise} \end{cases} \tag{10}$$

5. COMPLEXITY AND PERFORMANCE

The complexity of the algorithms in section 4 can be characterized by pipelined broadcast and gather in a tree of nodes. The width of a broadcast block is assumed to be the size of a single variable. The tree has m nodes, one for each instance in Ω_t, and the nodes operate in parallel. Assume that each node has n inputs and p outputs. (Each node only has one output variable, but with multiple output variables the network as a whole has p output variables. Each node outputs a ? value for every variable except the one specified by the instance stored at the node). Given these assumptions, analysis gives an overall complexity of $O(n+p+\log m)$ for both execution and learning, for a single pattern.

The complexity of the steps of the execution algorithm is summarized, followed by elaboration where necessary. The numbers correspond to the steps from the algorithms in section 5. The complexity of the learning algorithm is presented in the same format.

<u>Execution Algorithm</u>

1.	Broadcast	$O(n+\log m)$
2-2.2.	Node Computation	$O(n)$
2.3-3.	Gather	$O(p+\log m)$
total		$\boldsymbol{O(n+p+\log m)}$

Broadcast or gather for a single variable is clearly $O(\log m)$. For n variables, one might expect a complexity of $O(n\log m)$, but the pipelined tree reduces this to $O(n+\log m)$, since different levels of the tree operate on different variables simultaneously. The node computation is an $O(n)$ operation for n variables, but can be overlapped with broadcast. The gather is $O(p)$, one for each output variable, but this is overlapped with the gather of variables up the tree.

Learning Algorithm

1.	Broadcast α_t	$O(n+p+\log m)$
2.	Compute Relationships	$O(n+p)$
3.	Check for already learned	$O(\log m)$
4.	Check for self-delete	$O(\log m)$
5.	Check for DVA	$O(n+p+\log m)$
6.	Add α_t to network	$O(n+p+\log m)$
total		$\boldsymbol{O(n+p+\log m)}$

Steps 1 and 2 can actually overlap. The broadcast is $O(n+p)$ under the assumption that values for all n input variables and all p output variables are sent, even though many of the inputs may be * values and all but one of the output variables are ?. Combining this with the pipelined broadcast down the tree [$O(\log m)$ for one variable] yields $O(n+p+\log m)$. Computation of left- and right-hand side relationships can proceed as each value is received, and hence can overlap with broadcast. Steps 3 and 4 are both $O(\log m)$ because a single flag is broadcast, and a single value is gathered. Step 6 is $O(\log m)$ because a free node needs to be located and chosen (this can be done with a single broadcast and gather). It is $O(n+p)$ because the left- and right-hand sides need to be sent to the node for storage once the free node is located. The combined complexity is thus $O(n+p+\log m)$ for step 6.

The complexity of step 5 given above assumes that DVA can take place in parallel, and that there are always enough free nodes in the subtree of any node to accommodate DVA growth. The worst-case scenario is that the deepest leaf node in the tree must add n-1 instances to the instance set. In that case, the depth of the tree will be increased by $O(\log n)$, for $n > 1$, but on the average $n+p+(n-1)/2$ variables must be passed to the children at each level. It takes $O(\log m)$ time for the DVA flag to reach the deepest leaf node. The $O(\log n)$ growth factor will always be overshadowed by the $O(n+p)$ time required to pass variables (instances) between nodes. This leads to an overall DVA complexity of $O(n+p+\log m)$ if DVA takes place in parallel.

The analysis in this section shows that the complexity of execution and learning for the algorithms of section 5 is $O(n+p+\log m)$.

6. CONCLUSION

ANNs that use a static topology, i.e. a topology that remains fixed throughout learning, suffer from a number of short-comings. Current research is demonstrating the use of dynamic topologies in overcoming some of these problems. The Location-Independent Transformation (LIT) is a general strategy for implementing variations of ANNs that use dynamic topologies during learning.

This paper presented the LIA model, which is an LIT for ASOCS AA2 model. AA2, like other ASOCS models, has an inherently dynamic topology. In addition to high-level definitions for execution and learning algorithms, more formal definitions are presented for those interested in hardware (and software) implementations. Analysis gives overall complexities for both execution and learning as $O(n+p+\log m)$, where n is the number of inputs, p is the number of outputs, and m is the number of instances or Dnodes in original network.

Current on-going work includes the following:

- LITs for other ANNs, aside from those for which transformations have been developed.
- VLSI design and fabrication of LIT models.

- Investigation of appropriate applications, generalization algorithms and global minimization algorithms for ASOCS/LIA class models.

REFERENCES

[1] G. S. Almasi, A. Gottlieb, *Highly Parallel Computing*, The Benjamin/Cummings Publishing Company, Inc, Redwood City, CA, 1989.

[2] P. T. Baffes, J. M. Zelle, “Growing Layers of Perceptrons: Introducing the Extentron Algorithm”, *Proceedings of 1992 IEEE/INNS International Joint Conference on Neural Networks—Baltimore*, Vol. 2, pp 4979-4984, 1992.

[3] T. Denoeux, R. Lengelle, “Initializing Back Propagation Networks With Prototypes” *Neural Networks*, Vol. 6, #3, pp 351-363, Pergamon Press Ltd, New York, 1993.

[4] S. Fahlmann, “Faster-Learning Variations on Back-Propagation: An Empirical Study”, *Proceedings of the 1988 Connectionist Models Summer School*, pp 38-51, 1988.

[5] S. Fahlmann, C. Lebiere, “The Cascade-Correlation Learning Architecture”, *Advances in Neural Information Processing 2*, pp 524-532, Morgan Kaufmann Publishers: Los Altos, CA.

[6] Farhat, N., D. Psaltis, A. Prata, and E. Paek. Optical Implementation of the Hopfield Model. *Applied Optics, Vol. 24, #10.* pp.1469-1475 . 1985.

[7] Graf, H., L. Jackel, W. Hubbard. VLSI Implementation of a Neural Network Model. In *Artificial Neural Networks: Electronic Implementations*, Nelson Morgan, Ed. pp. 34-42. 1990.

[8] D. Hammerstrom, W. Henry, M. Kuhn. Neurocomputer System for Neural-Network Applications. In *Parallel Digital Implementations of Neural Networks*. K. Przytula, V. Prasanna, Eds. Prentice-Hall, Inc. 1991.

[9] W. D. Hillis. *The Connection Machine.* Cambridge, Mass.: MIT Press, 1985.

[10] E. Karnin, “A Simple Procedure for Pruning Back-Propagation Trained Neural Networks”, *IEEE Transactions On Neural Networks*, Vol. 1, #2, pp 239-242, June, 1990.

[11] C. Lee, “A Simple Procedure for Pruning Back-Propagation Trained Neural Networks”, *Neural Networks*, Vol. 6, #3, pp 385-392.

[12] T. Martinez, J.J. Vidal. “Adaptive Parallel Logic Networks”. *Journal of Parallel and Distributed Computing,* Vol. 5, #1. pp. 26-58. 1988.

[13] T. Martinez, D. Campbell, “A Self-Organizing Binary Decision Tree for Incrementally Defined Rule Based Systems”. *IEEE Trans. on Systems, Man and Cybernetics,* Vol. 21, #5. pp. 1231-1237, 1991.

[14] T. Martinez, D. Campbell, “A Self-Adjusting Dynamic Logic Module”, *Journal of Parallel and Distributed Computing*, Vol. 11, #4, pp 303-313. 1991.

[15] T. Martinez, D. Campbell, B. Hughes. “Priority ASOCS”, *Journal of Artificial Neural Networks*, Vol. 1, #3. pp. 403-429. 1994.

[16] C. Mead, *Analog VLSI and Neural Systems*, Addison-Wesley Publishing Company, Inc, 1991.

[17] S. Odri, D. Petrovacki, G. Krstonosic, “Evolutional Development of a Multilevel Neural Network”, *Neural Networks*, Vol. 6, #4. pp 583-595. Pergamon Press Ltd, New York, 1993.

[18] Ramacher, U., W. Raab, J. Anlauf, U. Hachmann, J. Beichter, N. Brüls, M. Weißling, E. Schneider, R. Männer, J. Gläß. Multiprocessor and Memory Architecture of the

Neurocomputer SYNAPSE-1. *Proceedings, World Congress on Neural Networks 1993, Vol. 4.* pp. 775-778. INNS Press, 1993.

[19] D. Reilly, L. Cooper, C. Erlbaum, "A Neural Model Category Learning", *Biological Cybernetics*, Vol. 45, pp 35-41, 1982.

[20] G. Rudolph, T. Martinez, "An Efficient Static Topology for Modeling ASOCS", *Artificial Neural Networks*, Editors, Kohonen et al, pp 279-734, Elsevier Publishers, North Holland, 1991.

[21] G. Rudolph, T. Martinez, "A Transformation for Implementing Localist Neural Networks", To appear in *Journal of Neural, Parallel and Scientific Computations*, 1995.

[22] Rudolph G., Martinez, T. R. "A Transformation for Implementing Multilayer Distributed FeedForward Neural Networks", Submitted, 1995.

[23] Shams, S. Dream Machine—A Platform for Efficient Implementation of Neural Networks with Arbitrarily Complex Interconnect Structures. *Technical Report CENG 92-23*. PhD Dissertation, USC, 1992.

[24] Stout, M., G. Rudolph, T.R. Martinez, L. Salmon, "A VLSI Implementation of a Parallel, Self-Organizing Learning Model", Proceedings of the *International Conference on Pattern Recognition* (1994) 373-376.