On Improving Connectionist Energy Minimization

Gadi Pinkas [*] Rina Dechter [†]

November 28, 1994

Abstract

Symmetric networks designed for energy minimization such as Boltzman machines and Hopfield nets are used frequently for optimization, constraint satisfaction and approximation of NP-hard problems. Nevertheless, finding a global solution (i.e., a global minimum for the energy function) is not guaranteed and even a local solution may take an exponential number of steps. We propose an improvement to the standard local activation function used for such networks. The improved algorithm guarantees that a *global* minimum is found in *linear* time for tree-like subnetworks. The algorithm is uniform and *does not* assume that the network is tree-like. It can identify tree-like subnetworks even in cyclic topologies (arbitrary networks) and avoid local minima along these trees. For acyclic networks, the algorithm is guaranteed to converge to a global minimum from any initial state of the system (self-stabilization) and remains correct under various types of schedulers. For general (cyclic) topologies, we show how our tree-like algorithm can be extended using the cycle-cutset idea. The general algorithm optimizes tree-like subnetworks and has some performance guarantees that are related to the size of the network's cycle-cutset. In any case, the algorithm performs no worse than the standard algorithms. On the negative side, we show that in the presence of cycles, no uniform algorithm

[*]Center for Optimization and Semantic Control, Washington University, Box 1045, St Louis, MO 63130, pinkas@cs.wustl.edu

[†]Department of Information and Computer Science, University of California, Irvine, CA 92717

exists that guarantees optimality even under a sequential synchronous scheduler. In addition, no uniform algorithm exists to optimize even *acyclic* networks when the scheduler is asynchronous.

1 Introduction

Symmetric networks such as Hopfield networks, Boltzmann machines, mean-field and Harmony networks are frequently used for optimization, constraint satisfaction and approximation of NP-hard problems [Hopfield 82], [Hopfield 84], [Hinton, Sejnowski 86], [Peterson, Hartman 89], [Smolensky 86], [Brandt et al. 88]. These models are characterized by a symmetric matrix of weights and a quadratic energy function that should be minimized. Usually, each unit computes the gradient of the energy function and updates its own activation value so that the free energy decreases gradually. Convergence to a *local* minimum is guaranteed although in the worst case it is exponential in the number of units [Kasif et al. 89], [Papadimitriou et al. 90].

In many cases the problem at hand is formulated as a minimization problem and the best solutions (sometimes the *only* solutions) are the global minima [Hopfield, Tank 85],[Ballard et al. 86], [Pinkas 90b]. The desired algorithm is therefore one that manages to reduce the impact of shallow local minima and improve the chances of finding a global minimum. Some models such as Boltzmann machines and Harmony nets, use simulated annealing to escape from local minima. These models asymptotically converge to a global minimum, meaning that if the annealing schedule is slow enough, a global minimum is found. Nevertheless, such a schedule is hard to find and therefore, in practice, these networks are not guaranteed to find a global minimum even in exponential time.

In this paper we look at the *topology* of symmetric neural networks. We present an algorithm that finds a global minimum for acyclic networks and optimizes a tree-like subnetwork in linear time. We then extend it to general topologies by dividing the network into fictitious tree-like subnetworks.

Partially, the algorithm is based on a dynamic programming algorithm that belongs to the family of nonserial dynamic programming methods [Bertelé, Brioschi 72]. This dynamic programming method was adapted also in [Dechter et al 90] for constraint optimization.

Our adaptation is connectionist in style; i.e., the algorithm can be stated as a simple, uniform activation function [Rumelhart et al. 86], [Feldman, Ballard 82] and it can be executed in parallel architectures using synchronous or asynchronous scheduling policies. It does not assume the desired topology (acyclic) and performs no worse than the standard local algorithms for all topologies. In fact, it may be integrated with many of the standard algorithms in such a way that the new algorithm out-performs the standard algorithms by avoiding local minima along the identified tree-like subnetworks.

Our algorithm is also applicable to a recent class of greedy algorithms called *local repair algorithms*, that are applied to optimization and constraint satisfaction problems. Usually (in local repair techniques), the problem at hand is formulated as a minimization of a distance function that measures the distance between the current state and the goal state (the solution). The algorithm picks a setting for the variables and then repeatedly changes the variable that causes the maximal decrease in the distance function. For example, a commonly used distance function for constraint satisfaction problems is a function that counts the number of violated constraints. A local repair algorithm may be viewed as an energy minimization network: the problem variables are the nodes, whereas, a node is connected to variables whose values are needed to determine the effect of changing the node's value on the distance function. The distance function plays the role of the energy function. Local repair algorithms, are sequential though, and they use a greedy scheduling policy whereby the next node to be activated is the one leading to the largest change in the distance (i.e., energy). Recently, such local repair algorithms were successfully used on various large-scale hard problems such as 3-SAT, n-queen, scheduling and constraint satisfaction [Minton et al 90], [Selman et al. 92].

Since local repair algorithms may be viewed as sequential variations on the energy paradigm, it is reasonable to assume that improvement to energy minimization will also be applicable to local-repair algorithms.

Our negative results on energy minimization involve conditions on the parallel model of execution and are applicable only to the parallel versions of local repair (where each problem variable is allocated a processing unit). This paper is a revised version and an extension of [Pinkas, Dechter 92].

The paper is organized as follows: Section 2 discusses connectionist energy minimization. Section 3 presents the new algorithm and gives an example where it out-performs the standard local algorithms. Section 4 discusses negative results, convergence under various schedulers and self-stabilization. Section 5 extends the approach to general topologies and suggests future research. Section 6 discusses applications that produce networks that are mostly cycle-free. Section 7 summarizes.

2 Connectionist energy minimization

Given a quadratic energy function of the form:

$$E(X_1, ..., X_n) = -\sum_{i<j}^{n} w_{i,j} X_i X_j - \sum_{i}^{n} +\theta_i X_i.$$

Each of the variables X_i may have a value of zero or one called the activation value, and the task is to find a zero/one assignment to the variables $X_1, ...X_n$ that minimizes the energy function. To avoid confusion with signs, we will consider the equivalent problem of maximizing the goodness function:

$$G(X_1, ..., X_n) = -E(X_1, ..., X_n) = \sum_{i<j} w_{i,j} X_i X_j + \sum_{i} \theta_i X_i \tag{1}$$

In connectionist approaches, we look at the network that is generated by assigning a node (i) for every variable (X_i) in the function, and by creating a weighted arc (with weight $w_{i,j}$) between node i and node j, for every term $w_{i,j} X_i X_j$. Similarly, a bias θ_i is given to unit i, if the term $\theta_i X_i$ is in the function. For example, figure 2-a shows the network that corresponds to the goodness function $E(X_1, ..., X_5) = 3X_2X_3 - X_1X_3 + 2X_3X_4 - 2X_4X_5 - 3X_3 - X_2 + 2X_1$. Each of the nodes is assigned a processing unit and the network collectively searches for an assignment that maximizes the goodness. The algorithm that is repeatedly executed in each unit/node is called the *activation function*. An algorithm is *uniform* if it is executed by all the units.

We give examples for two of the most popular activation functions for connectionist energy minimization: the discrete Hopfield network [Hopfield 82] and the Boltzmann machine [Hinton, Sejnowski 86].

In the discrete Hopfield model, each unit computes its activation value using the formula:

$$X_i = \begin{cases} 1 & \text{iff } \sum_j w_{i,j} X_j \geq -\theta_i \\ 0 & \text{otherwise} \end{cases}$$

In Boltzmann machines the determination of the activation value is stochastic and the probability to set the activation value of a unit to one is: $P(X_i = 1) = 1/(1 + e^{-(\sum_j w_{i,j} X_j + \theta_i)/T})$, where T is the annealing temperature.

Both approaches may be integrated with our topology-based algorithm; i.e., nodes that cannot be identified as parts of a tree-like topology use one of the standard local algorithms.

3 The algorithm

3.1 Key idea

We assume that the model of communication between neighboring nodes is a shared memory, multi-reader, single-writer model. We also assume that scheduling is done with a central scheduler (synchronous) and that execution is fair. In a *shared memory, multi-reader single-writer* each unit has a shared register called the activation register. A unit may read the content of the registers of all its neighbors but write only its own. *Central* scheduler means that the units are activated one at a time in an arbitrary order.[1] An execution is said to be *fair* if every unit is activated infinitely often. We do not require *self-stabilization* initially. Namely, algorithms may have an initialization step and can rely on initial values. Later we will examine the conditions under which the algorithm is also self-stabilized.

The algorithm identifies parts of the network that have no cycles (tree-like subnetworks), and optimizes the free energy on these subnetworks. Once a tree is identified, it is optimized using a dynamic programming method that propagates values from leaves to a root and back.

Let us assume first that the network is acyclic; any such network may be directed into a rooted tree. The algorithm is based on the observation that given an activation value (0/1) for a node in a tree, the optimal assignments for all its adjacent nodes are independent of each other. In particular, the optimal assignment to the node's descendants are independent of the assignments for its ancestors. Therefore, each node i in the tree may compute two values: G_i^1 is the maximal goodness contribution of the subtree rooted at i, including the connection to i's parent whose activation is *one*. Similarly, G_i^0 is the maximal goodness of the subtree, including the connection to i's parent whose activation value is *zero*. The acyclicity property will allow us to compute each node's G_i^1 and G_i^0 as a a simple function of its children's values, implemented as a propagation algorithm initiated by the leaves.

Knowing the activation value of its parent and the values G_j^0, G_j^1 of all its children, a node can compute the maximal goodness of its subtree. When the information reaches the root, it can assign a value (0/1) that maximizes the

[1]Standard algorithms need to assume the same condition in order to guarantee convergence to a *local* minimum (see [Hopfield 82]). This condition can be relaxed by restricting that only adjacent nodes are not activated at the same time (mutual exclusion).

goodness of the whole network. The assignment information propagates now toward the leaves. Knowing the activation value of its parent, a node can compute the preferred activation value for itself. At termination (at stable state), the tree is optimized.

The algorithm has 3 basic steps:
1) **Directing a tree:** knowledge is propagated from leaves toward the center so that after a linear number of steps, every unit in the tree knows its parent and children.
2) **Propagation of goodness values:** the values (G_i^1 and G_i^0), are propagated from leaves to the root. At termination, every node knows the maximal goodness of its subtree and the appropriate activation value it should assign given that of its parent. In particular, the root can now decide its own activation value so as to maximize the whole tree.
3) **Propagation of activation values:** starting with the root, each node in turn determines its activation value. After O(depth of tree) steps, the units are in a stable state which globally maximizes the goodness.

Each unit's *activation register* consists of the following fields: X_i: the activation value; G_i^0 and G_i^1: the maximal goodness values; and $(P_i^1, .., P_i^j)$: a bit for each of the j neighbors of i that indicates i's parent.

3.2 Directing a tree

The goal of this algorithm is to inform every node of its role in the network and its child-parent relationships. Nodes with a single neighbor identify themselves as leaves first and then identify their neighbor as a parent (point to it). A node identifies itself as a root when all neighbors point toward it. When a node's neighbors but one point toward it, the node selects the one as a parent. Finally, a node that has at least two neighbors *not* pointing toward it, identifies itself as being outside the tree.

The problem of directing a tree is related to the problem of selecting a leader in a distributed network, and of selecting a center in a tree [Korach et. al, 84]. Our problem differs (from general leader selection problems) in that the network is a tree. In addition we require our algorithms to be self-stabilized. A related self-stabilizing algorithm appears in [Collin et al. 91]. That algorithm is based on finding a center of the tree as the root node and therefore creates more balanced trees. The advantage of the algorithm presented here is that it

Figure 1: Directing a tree: a) A tree b) A cyclic network with a tree-like subnetwork.

3.3 Propagation of goodness values

In this phase every node i computes its goodness values G_i^1 and G_i^0, by propagating these two values from the leaves to the root (see figure 2).

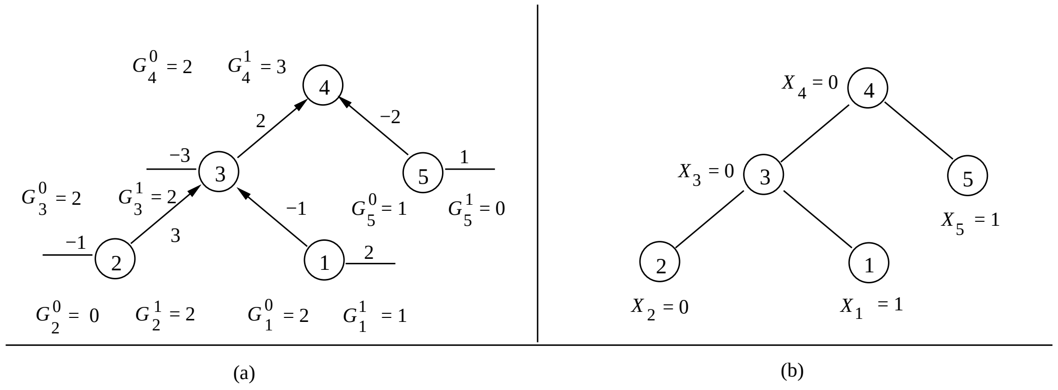

Figure 2: a) Propagating goodness values. b) Propagating activation values.

Given a node X_i, its parent X_k and its children, $children(i)$ in the tree, it can be shown, based on the energy function (1), that the goodness values obey the following recurrence:

$$G_i^{X_k} = max_{X_i \in \{0,1\}}\{\sum_{j \in children(i)} G_j^{x_i} + w_{i,k}X_iX_k + \theta_i X_i\}$$

Consequently a nonleaf node i computes its goodness values using the goodness values of its children as follows: If $X_k = 0$, then i must decide between setting $X_i = 0$, obtaining a goodness of $\sum_j G_j^0$, or setting $X_i = 1$, obtaining a goodness of $\sum_j G_j^1 + \theta_i$. This yields:

$$G_i^0 = max\{\sum_{j \in children(i)} G_j^0, \sum_{j \in children(i)} G_j^1 + \theta_i\}$$

Similarly, when $X_k = 1$, the choice between $X_i = 0$ and $X_i = 1$, yields:

$$G_i^1 = max\{\sum_{j \in children(i)} G_j^0, \sum_{j \in children(i)} G_j^1 + w_{i,k} + \theta_i\}$$

The initial goodness values for leaf nodes can be obtained from the above (no children). Thus, $G_i^0 = max\{0, \theta_i\}$, $G_i^1 = \{0, w_{ik} + \theta_i\}$.

For example: If unit 3 in figure 2 is zero then the maximal goodness contributed by node 1 is $G_1^0 = max_{X_1 \in \{0,1\}}\{2X_1\} = 2$ and is obtained at $X_1 = 1$. Unit 2 (when $X_3 = 0$) contributes $G_2^0 = max_{X_2 \in \{0,1\}}\{-X_2\} = 0$ obtained at $X_2 = 0$, while $G_2^1 = max_{X_2 \in \{0,1\}}\{3X_2 - X_2\} = 2$ is obtained at $X_2 = 1$. As for nonleaf nodes, if $X_4 = 0$, then when $X_3 = 0$, the goodness contribution will be $\sum_k G_k^0 = 2 + 0 = 2$, while if $X_3 = 1$, the contribution will be $-3 + \sum_k G_k^1 = -3 + 1 + 2 = 0$. The maximal contribution $G_3^0 = 2$ is achieved at $X_3 = 0$.

3.4 Propagation of activation values

Once a node is assigned an activation value, all its children can activate themselves so as to maximize the goodness of the subtrees they control. When such value is chosen for a node, its children can evaluate *their* activation values, and the process continues until the whole tree is assigned.

There are two kinds of nodes that may start the process: a root which will choose an activation value to optimize the entire tree, and a non-tree node which uses a standard activation function.

When a root X_i is identified, if the maximal goodness is $\sum_j G_j^0$, it chooses the value "0." If the maximal goodness is $\sum_j G_j^1 + \theta_i$, it chooses "1." In summary, the root chooses its value according to:

$$X_i = \begin{cases} 1 & \text{iff } \sum_j G_j^1 + \theta_i \geq \sum_j G_j^0 \\ 0 & \text{otherwise} \end{cases}$$

In figure 2 for example, $G_5^1 + G_3^1 + 0 = 2 < G_5^0 + G_3^0 = 3$ and therefore $X_4 = 0$.

An internal node whose parent is k chooses an activation value that maximizes $\sum_j G_j^{x_i} + w_{i,k}X_iX_k + \theta_i X_i$. The choice therefore, is between $\sum_j G_j^0$ (when $X_i = 0$) and $\sum_j G_j^1 + w_{i,k}X_k + \theta_i$ (when $X_i = 1$), yielding:

$$X_i = \begin{cases} 1 & \text{iff } \sum_j G_j^1 + w_{i,k}X_k + \theta_i \geq \sum_j G_j^0 \\ 0 & \text{otherwise} \end{cases}$$

As a special case, a leaf i chooses $X_i = 1$ iff $w_{i,k}X_k \geq -\theta_i$, which is exactly the discrete Hopfield activation function for a node with a single neighbor. For example, in figure 2, $X_5 = 1$ since $w_{4,5}X_4 = 0 > -\theta_5 = -1$, and $X_3 = 0$ since $G_1^1 + G_2^1 + 2X_4 + \theta_3 = 1 + 2 + 0 - 3 = 0 < G_2^0 + G_1^0 = 2$. Figure 2-b shows the activation values obtained by propagating them from the root to the leaves.

3.5 A complete activation function

Interleaving the three algorithms described earlier achieves the goal of identifying tree-like subnetworks and maximizes their goodness. In this subsection we present the complete algorithm, combining the three phases while simplifying the computation. The algorithm is integrated with the discrete Hopfield activation function demonstrating how similar the formulas are.

Let i be the executing unit, j a non-parent neighbor of i and k the parent of i:

Algorithm activate: Optimizing on Tree-like Subnetworks (unit i):

```
1. Initialization: If first time, then $(\forall j)\ P_i^j = 0$; /*Clear pointers (cyclic nets)*/

2. Tree directing: If there exists a single neighbor $k$, such that $P_k^i = 0$,
       then $P_i^k = 1$ and for all other neighbors $j$, $P_i^j = 0$;
       else, for all neighbors $P_i^j = 0$;

3. Computing goodness values:
   $G_i^0 = max\{\sum_{j \in children(i)} G_j^0 P_j^i, \sum_{j \in children(i)} G_j^1 P_j^i + \theta_i\}$;
   $G_i^1 = max\{\sum_{j \in children(i)} G_j^0 P_j^i, \sum_{j \in children(i)} (G_j^1 P_j^i + w_{i,j} P_i^j) + \theta_i\}$;

4. Assigning activation values:
   If at least two neighbors are not pointing to $i$, then /*Not in tree: use Hopfield*/
       $X_i = \begin{cases} 1 & \text{if } \sum_j w_{i,j} X_j \geq -\theta_i \\ 0 & \text{otherwise} \end{cases}$
   else,    /* Node in a tree (including root and leaves) */
       $X_i = \begin{cases} 1 & \text{if } \sum_j ((G_j^1 - G_j^0) P_j^i + w_{i,j} X_j P_i^j) \geq -\theta_i \\ 0 & \text{otherwise} \end{cases}$
```

The algorithm's properties will be discussed in section 4.

3.6 Goodness computation without tree-directing

When restricted to acyclic networks, the tree-directing procedure is not needed for computing the goodness values, but merely for assigning activation after the goodness values had converged. The idea is that each node compute for each of its neighbors a pair of goodness values dedicated to it. It operates as if each neighbor is its "parent".

Let i be the executing unit, j an arbitrary neighbor of i and k be the neighbor to which the goodness values are directed:

```
Algorithm activate1: Optimizing goodness on Tree-like Subnetworks (unit $i$):

    1. Computing goodness values for neighbor $k$:
       $G_i^0 = max\{\sum_{j \in neighbors(i), j \neq k} G_j^0, \sum_{j \in neighbors(i); j \neq k} G_j^1 + \theta_i\}$;
       $G_i^1 = max\{\sum_{j \in neighbors(i); j \neq k} G_j^0, \sum_{j \in neighbors(i); j \neq k}(G_j^1 + w_{i,k}) + \theta_i\}$;
```

It is easy to show that:

Theorem 1 *: The goodness values are guaranteed to converge to the correct googness values.*

The task of "assigning activation value" still needs to be done in a coordinated manner using the structure of the tree. Each node will be assigned a value relative to the value that its parent has, as before.

3.7 An example

The example illustrated in figure 3 demonstrates a case where a local minimum of the standard algorithms is avoided. Standard algorithms may enter such local minimum and stay in a stable state that is clearly wrong.

The example is a variation on a Harmony network example [Smolensky 86] (page 259), and an example from [McClelland et al. 86] (page 22). The task of the network is to identify words from low-level line segments. Certain patterns of line segments excite units that represent characters, and certain patterns of characters excite units that represent words. The line strokes used to draw the characters are the input units: L1,..., L5. The units "N," "S," "A" and "T" represent characters. The units "able", "nose", "time" and "cart" represent words, and Hn, Hs, Ha, Ht, H1...,H4 are hidden units required by the Harmony model. For example, given the line segments of the character S, unit L4 is activated (input), and this causes units Hs and "S" to be activated. Since "NOSE" is the only word that contains the character "S", both H2 and the unit "nose" are also activated and the word "NOSE" is identified.

The network has feedback cycles (symmetric weights) so that ambiguity among characters or line-segments may be resolved as a result of identifying a word. For example, assume that the line segments required to recognize the word "NOSE" appear, but the character "N" in the input is blurred and therefore the setting of unit L2 is ambiguous. Given the rest of the line segments

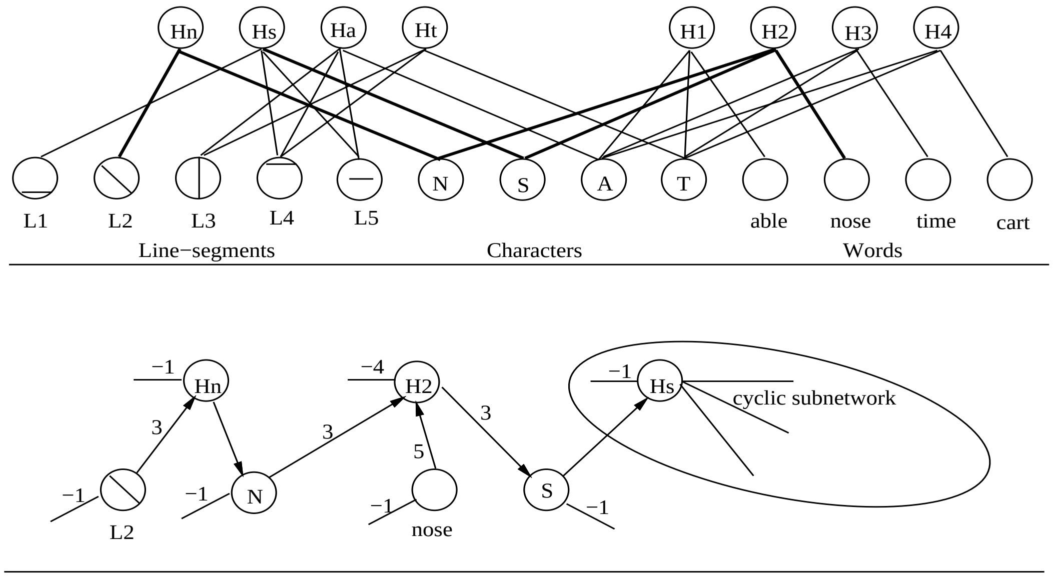

Figure 3: A Harmony network for recognizing words: local minima along the subtrees are avoided.

(e.g., those of the character "S"), the network identifies the word "NOSE" and activates units "nose" and H2. This will cause unit "N" and all of its line segments to be activated. Thus the ambiguity of L2 is resolved.

The network is designed to have a global minimum when L2, Hn, "N", H2 and "nose" are all activated. However, standard connectionist algorithms may fall into a local minimum when all these units are zero, generating goodness of $5 - 4 = 1$. The correct setting (global minimum) is found by our tree-optimization algorithm (with goodness: 3-1+3-1+3-1+5-1-4+3-1+5=13). The thick arcs in the upper network of figure 3 mark the arcs of a tree-like subnetwork. This tree-like subnetwork is drawn with pointers and weights in the lower part of the figure. Node "S" is not part of the tree and its activation value is set to one because the line-segments of "S" are activated. Once "S" is set, the units along the tree are optimized (by setting them all to one) and the local minimum is avoided.

4 Feasibility, convergence, and self-stabilization

So far we have shown how to enhance the performance of connectionist energy minimization networks without losing much of the simplicity of the standard approaches. The simple algorithm presented is limited in three ways, however. First, it assumes unrealistically that a central scheduler is used; i.e., a scheduler that activates the units one after the other synchronously.[2] We would like the network to work correctly under a *distributed scheduler*, where any subset of units may be activated for execution at the same time asynchronously. Second, the algorithm guarantees convergence only for tree-like sub-networks. We would like to find an algorithm that converges to correct solutions even if cycles are introduced. Finally, we would like the algorithm to be *self-stabilizing*. It should converge to a legal, stable state given enough time, even after noisy fluctuations that cause the units to execute arbitrary program states and the registers to have arbitrary content. Formally, an algorithm is *self-stabilizing* if in any fair execution, starting from any input configuration and any program state (of the units), the system reaches a valid stable configuration.

In this section, we illustrate two negative results regarding the first two problems; i.e., that it is not feasible to build uniform algorithms for trees under a distributed scheduler, and that such an algorithm is not feasible for cyclic networks even under a central scheduler. We then show how to weaken the conditions so that convergence is guaranteed (for tree-like subnetworks) in realistic environments and self-stabilization is obtained.

A scheduler can generate any specific schedule consistent with its definition. Thus the central scheduler can be viewed as a specific case of the distributed scheduler. We say that a problem is *impossible* for a scheduler, if for every possible algorithm there exists a fair execution generated by such a scheduler that does not find a solution to the problem. Since all the specific schedules generated by a central scheduler can also be generated by a distributed scheduler, what is impossible for the central scheduler is also impossible for the distributed scheduler.

[2]The same results are obtained if the steps of the algorithm executes as one atomic operation or if neighbors are mutually excluded.

4.1 Negative results for uniform algorithms

In [Collin et al. 91] following [Dijkstra 74] negative results are presented regarding the feasibility of distributed constraint satisfaction. Since constraint satisfaction problems can be formulated as energy minimization problems, these feasibility results apply also for computing the global minimum of energy functions. For completeness we now adapt their results for a connectionist computation of energy minimization.

THEOREM **4.1** *No deterministic*[3] *uniform algorithm exists that guarantees a global minimum under a distributed scheduler, even for simple chain-like trees, assuming that the algorithm needs to be insensitive to initial conditions.*

Proof:
Consider the network of Figure 4. There are two global minima possible : (11...1101...11) and (11...1011...11) (when the four centered digits are assigned to units, $i-1, i, i+1, i+2$). If the network is initialized such that all units have the same register values, and all units start with the same program state, then there exists a fair execution under a distributed scheduler such that in every step all units are activated. The units left of the center $(1, 2, 3, ...i)$ "see" the same input as those units right of the center $(2i, 2i-1, 2i-2, ..., i+1)$ respectively. Because of the uniformity and the determinism, the units in each pair $(i, i+1), (i-1, i+2), ..., (1, 2i)$ must transfer to the same program state and produce the same output on the activation register.Thus, after every step of that execution, units i and $i+1$ will always have the same activation value and a global minimum (where the two units have different values) will never be obtained. $\square$

This negative result should not discourage us in practice since it relies on an obscure infinite sequence of executions which is unlikely to occur under a random scheduler. Despite the negative result, one can show that the algorithm presented will optimize the energy of tree-like subnetworks under a distributed scheduler if at least one of the following cases holds (see the next section for details):

1. If step 2 of algorithm *activate* in Section 3.5 is atomic;

[3]The proof of this theorem assumes determinism and does not apply to stochastic activation functions.

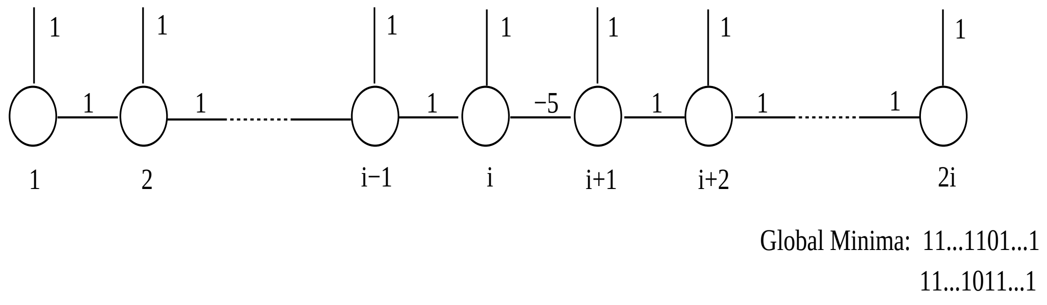

Figure 4: No uniform algorithm exists to optimize chains under distributed schedulers.

2. if for every node i and every neighbor j, node i is executed without j infinitely often (fair exclusion);[4]

3. if one node is unique and acts as a root, that is, does not execute step 2 (an almost uniform algorithm);

4. if the network is cyclic (one node will be acting as a root).[5]

Another negative result similar to [Collin et al. 91] is given in the following theorem.

THEOREM **4.2** *If the network is cyclic, no deterministic uniform algorithm exists that guarantees a global minimum, even under a central scheduler, assuming that the algorithm needs to be insensitive to initial conditions.*

This may be proved even for cyclic networks as simple as rings.
Proof:
In Figure 5, we see a ring-like network whose global minima are (010101) and (101010). Consider a fair execution under a central scheduler that activates the units 1,4,2,5,3,6 in order and repeats this order indefinitely. Starting with the same program state and same inputs, the two units in every pair of (1,4), (2,5), (3,6) "see" the same input, therefore they have the same output, and transfer to the same program state. As a result, these units never output different values and a global minimum is not obtained. $\square$

[4]Case one is a special case of case two.

[5]Global solutions are not guaranteed to be found but all tree-like subnetworks will be optimized.

Note that any tree-like subnetwork of a cyclic network will be optimized even under a distributed scheduler (since nodes that are part of a cycle are identified as roots and the algorithm acts as an almost uniform algorithm).

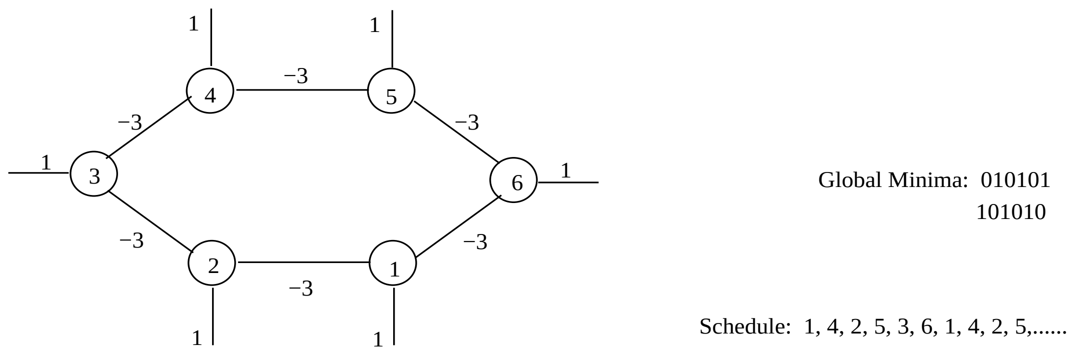

Figure 5: No uniform algorithm exists that guarantees to optimize rings even under a central scheduler.

4.2 Convergence and self-stabilization

In the previous subsection we proved that under a pure distributed scheduler there is no hope for a uniform network algorithm. In addition, we can easily show that the algorithm is not self-stabilizing when cycles are introduced. For example, consider the configuration of the pointers in the ring of Figure 6. It is in a stable state, although, clearly not a valid tree.[6]

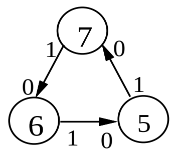

Figure 6: The uniform algorithm is not self-stabilizing in cyclic networks.

[6]Such configuration will never occur if all units start at the starting point; i.e., clearing the bits of P_i. It may only happen due to some noise or hardware fluctuations.

In this subsection we weaken the requirements allowing our algorithm to converge to correct solutions and to be self-stabilizing under realistically weaker distributed schedulers.

We will not use the notion of a *pure* distributed scheduler; instead, we will ask our distributed scheduler to have the *fair exclusion* property.

DEFINITION **4.1** A scheduler has the *fair exclusion* property if for every two neighbors, one is executed without the other infinitely often.

Intuitively, a distributed scheduler with fair exclusion will no longer generate infinite sequences of the pathological execution schedules used in the previous subsection to prove the negative results. Instead, it is guaranteed that from time to time, every two neighboring units will not execute together.

In addition we might weaken the requirement on the uniformity of the algorithm (that all nodes execute the same procedure). An *almost uniform* algorithm is when all the nodes perform the same procedure except one node that is marked unique. In the almost uniform version of algorithm *activate*, the root of the tree is marked and executes the procedure of section 3.5 as if all its neighbors are pointing to it; i.e., it constantly sets P_i^j to zero.

THEOREM **4.3** *Algorithm* activate *of section 3.5 has the following properties: 1. It converges to a global minimum and is self-stabilizing in networks with tree-like topologies under a distributed scheduler with fair exclusion. 2. The algorithm also converges in tree-like subnetworks (but is not self-stabilizing) when the network has cycles. 3. It is self-stabilizing for any topology if an* almost *uniform algorithm is applied, even under a* pure *distributed scheduler.*

For proof see appendix.

5 Extensions to arbitrary networks

The algorithm we present in section 3 is limited in that it is restricted to nodes of tree-like subnetworks only. Nodes that are part of a cycle execute the traditional activation function which may lead to the known drawbacks of local energy minima and slow convergence. In this section we suggest generalizations of our algorithms to nodes that are parts of cycles, that will work well for sparse networks. In the following paragraphs we will discuss the principles underlying this method. A full account of this extension is deferred for future work.

Two well-known complementary schemes for extending tree algorithms to non-tree networks, are *tree clustering*, and *cycle-cutset decomposition* [Dechter 90] used both in Bayes networks and constraint networks. In tree clustering the idea is to combine subsets of variables into higher level *meta* variables until the interaction between the meta level variables is tree-like. The cycle-cutset decomposition, on the other hand, is based on the fact that an instantiated variable cuts the flow of information on any path on which it lies and therefore changes the effective connectivity of the network. Consequently, when the group of instantiated variables cuts all cycles in the graph, (e.g., a cycle-cutset), the remaining network can be viewed as cycle-free and can be solved by a tree algorithm. The complexity of the cycle-cutset method can be bounded exponentially in the size of the cutset set in each nonseperable component of the graph [Dechter 90].

In this section we show how to extend our distributed energy minimization tree algorithm to arbitrary networks using the cycle-cutset idea.

5.1 Using cutset nodes and values

Recall that the energy minimization task is to find a zero/one assignment to the variables $X = \{X_1, ..., X_n\}$ that maximizes the goodness function. Define, $Gmax(X_1, ..., X_n) = max_{X_1,...,X_n} G(X_1, ..., X_n)$. The task is to find an activation level $X_1, ..., X_n$ satisfying

$$Gmax(X_1, ...X_n) = max_{X_1,...,X_n}(\sum_{i<j} w_{i,j} X_i X_j + \sum_i \theta_i X_i). \qquad (2)$$

Let $Y = \{Y_1, ..., Y_k\}$ be a subset of the variables $X = \{X_1, ..., X_n\}$. The maximum can be computed in two steps. First we compute the maximum energy conditioned on a fixed assignment $Y = y$, then maximize the resulting function over all possible assignments to Y. Let $Gmax(X|Y = y)$, the maximum goodness value of G conditioned on $Y = y$, be defined by:

$$Gmax(X|Y = y) = max_{X=x|x_Y=y}\{G(X)\},$$

where, x_Y is the zero/one value assignments in the instantiation x that are restricted to the variable subset Y. Clearly,

$$Gmax(X) = max_{Y=y} Gmax(X|Y = y).$$

If the variables in Y form a cycle-cutset, then the conditional maxima of $Gmax(X|Y = y)$ can be computed efficiently using a tree algorithm. The overall maxima is achieved by enumerating over all possible assignments to Y. As a matter of fact, enumeration can be restricted to each nonseparable component and to each of its own cutset since the hyper structure between components is tree-like [Dechter 90], [Even, 79]. Obviously, this scheme is effective only when the cycle-cutset is small. In section 5.2 we discuss some steps toward implementing this idea in a distributed environment.

Given a network with a set of nodes $X = \{X_1, ..., X_n\}$, let us first assume (1) that there is a known set of cutset variables $Y = \{Y_1, ..., Y_k\}$, (2) that each node knows whether or not it is designated as a cutset node, and (3) that a cutset node has already had an assigned activation value of '0' or '1'.[7] We denote the cutset node assignments, $Y = y$. Under these assumptions we show that the network can compute the conditional maxima energy, $Gmax(X|Y = y)$, using a simple modification to the tree algorithm *activate*. The modified procedure, called *activate-with-cycle* is given next.

Algorithm *activate-with-cutset-values* is guaranteed to compute the maximum energy of the network conditioned to $Y = y$.

[7]Later we'll see that the assumptions above need not hold. We can design a uniform algorithm which obtains a cutset using randomization and determines the cutset values using a local algorithm. In addition, we can design an algorithm (with exponential complexity) that computes a global solution ($GMAX$) by enumerating all possible cutset value-combinations.

Algorithm activate-with-cutset-values: Optimizing with cycle-cutset, Assumptions: 1) The cutset (Y), removes all cycles. 2) The values of the cutset nodes (y) are known a priori.

```
1. Initialization: If first time, then $(\forall j)\ P_i^j = 0$;

2. Tree directing:
   If $i$ is a cutset node, then for every neighbor $(j)$ If $P_j^i = 0$, then $P_i^j = 1$;   (A
   cutset variable selects each of its neighbors as a parent if that neighbor doesn't
   point to it.)
   else (regular node), if there exists a single neighbor $k$, such that $P_k^i = 0$,
      then $P_i^k = 1$ and for all other neighbors $j$, $P_i^j = 0$;
   else, for all neighbors $P_i^j = 0$;

3. Assigning activation values:
   If $i$ is a cutset node, then $X_i = y_i$, where $y_i$ is the cutset (known) value.
   else (regular tree nodes)
      $X_i = \begin{cases} 1 & \text{if } \sum_j((G_j^1 - G_j^0)P_j^i + w_{i,j}X_jP_i^j) \geq -\theta_i \\ 0 & \text{otherwise} \end{cases}$

4. Computing goodness values:
   If (cutset node)
      (For each neighbor $(j)$, $G_i^0, G_i^{j1}$ are goodness values for neighbor $(j)$ ).
      $G_i^0 = y_i(\theta_i)$,
      $G_i^{j1} = y_i(\theta_i) + w_{ij}$
   else (a regular tree node),
      $G_i^0 = max\{\sum_{j \in children(i)} G_j^0 P_j^i, \sum_{j \in children(i)} G_j^1 P_j^i + \theta_i\}$;
      $G_i^1 = max\{\sum_{j \in children(i)} G_j^0 P_j^i, \sum_{j \in children(i)}(G_j^1 P_j^i + w_{i,j}P_i^j) + \theta_i\}$;
      (note that, if $(j)$ is a cutsetnode, then $G_j^1 = G_j^{i1}$)
```

The basic idea is to let each cutset node behave as a leaf node. One can view a cutset node as if it duplicates itself for each neighbor. In the above algorithm we assume that all the nodes are on the tree, some of which are cutset nodes that act as leaves. The change to the tree-directing part is in that a cutset variable makes each one of its neighbors (for whom it is not a parent), its own parent. Thus a cutset variable may have several parents and zero or more part-of-a-tree neighbors (which point to it). Considering again the example network in figure 1b and assuming node (7) is a cutset variable, a tree-directing may now change so that node (7) points both to (5) and to (6), (6) points to (5) and (5) remains the root. Note that with this modification all arcs are directed and the resulting graph is an acyclic directed graph.

Once the graph is directed, each regular non-cutset node has exactly the same view as before. It has one parent (or no parent) and perhaps a set of child nodes, some of which may be cutset nodes. It then computes goodness values and activation values almost as before. A cutset variable will compute its goodness values for each of its neighbors separately based on the weighted arc with the parent (the G_i^0's are identical for all parents but the G_i^1 may differ for each parent). In addition, if one of the children of node i is a cutset variable, i should be using the goodness values of that child that is designated to it when computing its own goodness values. The reader should note that a solution found in this method is a global energy minimum (goodness maximum) conditioned on the values of the cutset variables. It is not guaranteed however, to be even a *local* minimum of the original energy function. Some of the cutset nodes may be unstable and may flip their value if they use the Hopfield activation function after the convergence of the tree.

In the next subsections, we will extend the cutset approach in two directions. The first is to compute an *unconditioned* global solution by enumerating all possible cutset values. The second direction is to combine a standard local activation algorithm with the cutset technique. This combination produces a *uniform*, connectionist-style algorithm that although not guaranteed to find a global solution, is still more powerful than both standard and cutset methods alone.

5.2 Unconditioned global minimum

In order to extend this algorithm for computing unconditional maxima we have to address the issue of enumerating all conditional energy minimizations over all instantiations of Y. One option we propose is to use the brute-force approach of computing all conditional goodness values in parallel as follows. Once the tree-directing part is accomplished, a node computes a collection of goodness values, each indexed by a conditioning assignment $Y = y$. In actuality, a node will only condition its values on a subset of the cutset variables that reside in the directed subtree for which it is the root. Nodes will compute goodness values associated with each assignment to the cutset variable. In step (4) of the algorithm the goodness values of a node that are associated with the cutset assignment $Y = y$ will be computed using the goodness values of child nodes that are also associated with the same assignment $Y = y$. The maximum number of goodness values each node may need to carry is exponential in the cutset size.[8] Upon convergence, the roots of the trees obtained from the cutsets will compare the goodness value associated with all its indices, select an assignment $Y = y$ that maximize the overall goodness value and then choose a corresponding activation value. Subsequently, it will propagate this information down the tree. Nodes will compute their activation using those goodness values that are associated with the maximizing assignment. Cutset variables will switch to the activation value associated with the selected assignment. All the above steps can be executed using one activation function.

The main issues that still must be addressed in this approach are where cutset variables come from and how they will be selected. It is known that computing the minimal cutset is NP-hard. We will therefore settle for any heuristic method that tries to compute a small cutset. One simple answer is that the determination of the cutset variable is performed centrally and announced to each node. This clearly violates the spirit of connectionist computation but may nevertheless be a practical solution for real networks. Alternatively, we should be looking for a uniform algorithm for finding a cycle-cutset. Clearly the problem is unsolvable for completely uniform networks. For an almost uniform network, when just one node is allowed to execute a different algorithm, one can envision a algorithm that may simulate one of the approximation algorithms for cycle-cutset [Becker, Geiger, 1994]. This is one of the open issues that we leave for future work.

5.3 A local search with dynamic cutset values

Another approach is to use the cutset method to improve *local* search without trying to enumerate all possible cutset values. In this approach, cutset nodes are either known (a priori) or randomly selected; however, their values aren't given and are computed using standard local techniques (e.g., Hopfield). The rest of the nodes use the tree algorithm (or Hopfield if they are not marked as parts of trees).

[8]A careful implementation can assure that the maximum number of goodness values that must be carried by each node is exponential only in the cutset size of its own nonseparable component.

Algorithm activate-with-cutset: Optimizing with cycle-cutset when values are not given,
Assumption: The cutset nodes are given a priori.

```
1. Initialization: If first time, then $(\forall j)\ P_i^j = 0$;

2. Tree directing:
   If $i$ is a cutset node, then for every neighbor $(j)$, if $P_j^i = 0$, then $P_i^j = 1$;
      (neighbors become parents unless they already point to it)
   else (regular tree node), if there exists a single neighbor $k$, such that $P_k^i = 0$,
      then $P_i^k = 1$ and for all other neighbors $j$, $P_i^j = 0$;
   else (root or non-tree node), for all neighbors $P_i^j = 0$;

3. Assigning activation values:
   If all neighbors of $i$ point to it except maybe one (i.e., it is part of a tree) then,
      $X_i = \begin{cases} 1 & \text{if } \sum_j ((G_j^1 - G_j^0)P_j^i + w_{i,j}X_jP_i^j) \geq -\theta_i \\ 0 & \text{otherwise} \end{cases}$
   else (a cutset node or a node that is not yet part of any tree), Compute Hopfield:
      $X_i = \begin{cases} 1 & \text{if } \sum_j w_{i,j}X_j \geq -\theta_i \\ 0 & \text{otherwise} \end{cases}$

4. Computing goodness values:
   If $i$ is a cutset node, then for each neighbor $j$,
      $G_i^0 = X_i(\theta_i)$,
      $G_i^{j1} = X_i(\theta_i) + w_{ij}$    ($G_i^0, G_i^{j1}$ are goodness values for neighbor $j$ ).
   else (a regular tree node),
      $G_i^0 = max\{\sum_{j \in children(i)} G_j^0 P_j^i, \sum_{j \in children(i)} G_j^1 P_j^i + \theta_i\}$;
      $G_i^1 = max\{\sum_{j \in children(i)} G_j^0 P_j^i, \sum_{j \in children(i)} (G_j^1 P_j^i + w_{i,j}P_i^j) + \theta_i\}$;
```

As before, the network computes a minimum of the energy, conditioned on the values of the cutset variables. This time, however, the cutset values are not given a priori and instead, are computed constantly using a standard local algorithm such as Hopfield. In addition, nodes that are not yet part of a tree, or nodes that are still part of a cycle (if the cutset is not perfect) use Hopfield as well.

A node flips its value either as a result of a Hopfield step (cutset and non-tree nodes) or in order to optimize a tree. In both steps, the energy does not increase and as a result the algorithm is guaranteed to converge on a stable local or global minimum. This is summarized in algorithm *activate-with-cutset*. The algorithm's idea is that if performed sequentially it would iterate between the following two steps: 1. finding a local minima using Hopfield activation function 2. finding global minima conditioned on the cutset values determined in the previous step, via the tree algorithm. In the connectionist algorithm these two steps are not synchronized. As a result we get an algorithm that converges on a solution that is both a local minima relative to Hopfield algorithm as well as a conditional global minima relative to the cutset variables.

The following example demonstrates how the algorithm finds a better minimum than what is found by the standard Hopfield algorithm when there are cycles:

Consider the energy function: $energy = 50AB - 200BC - 100AC - 3AD - 3DE - 3AE + 0.1A + 0.1B + 0.1C + 4E + 4D$. The associated network consists of two cycles: A, B, C and A, D, E. If we select node A as a cutset node, the network would then be cut into two acyclic (tree-like) sub-networks. Assume that the network starts with a setting of zeros ($A, B, C, D, E = 0$). This is a local minimum (energy $= 0$) of the Hopfield algorithm. Our activate-with-cutset algorithm breaks out of this local minimum by optimizing the acyclic sub-network A, B, C conditioned on $A = 0$. The result of the optimization is the assignment $A = 0, B = 1, C = 1, D = 0, E = 0$ with $energy = -199.7$. It is not a stable state because A obtains an excitatory sum of inputs (50) and therefore flips its value to $A = 1$ using its Hopfield activation algorithm. The new state $A, B, C = 1, D, E = 0$ is also a local minimum of the Hopfield paradigm ($energy = -249.7$). However, since nodes A, D, E form a tree, the activate-with-cutset algorithm also manages to break out of this local minimum. It finds a global solution conditioned on $A = 1$ which happens to be the global minimum $A, B, C, D, E = 1$ with $energy = -250.97$. The new algorithm was capable of finding the only global minimum of the energy function and managed to escape two of the local minima that trapped the Hopfield algorithm.

In the next subsection we examine a uniform extension of the cutset idea that does not assume knowing the cutset nodes a priori.

5.4 A local search with dynamic cutset variables

The activate-with-cutset algorithm is not a uniform algorithm. The cutset nodes have to be known prior to execution and special logic has to be associated with these nodes. When the cutset nodes are not given a priori, or when we wish to have a uniform algorithm, we might select our cutset nodes by randomization. A random cutset selection may be obtained using a local heuristics that is designed to improve the chances for selecting good cutset nodes.

The activate-with-random-cutset algorithm does not assume known cutset nodes. Instead, a node that is not part of any tree (e.g., part of a cycle), may turn into a cutset node with probability $P = f()$. The function $f()$ is a heuristics that assigns high probabilities to nodes with potential to become "good" cutset nodes. Similarly, a cutset node may turn into a non-cutset node if it becomes part of a tree or if the the random process decides so (with probability $P = 1 - f()$. The rest of the algorithm is very similar to activate-with-cutset; we just have to be aware that the randomly selected cutset is not perfect and that there might be too many or too few cutset nodes. Fewer cutset nodes will leave some cycles around, while too many cutset nodes may cause the solution to be conditioned on more nodes than is needed. Cutset nodes that haven't been identified yet as parts of trees and nodes that are parts of cycles compute the standard Hopfield activation function.[9] Redundant cutset nodes that are not needed are going to be de-selected in time and optimized as parts of trees.

The algorithm needs to differentiate between three types of nodes, when none is known a priori: 1) Nodes that are currently selected as cutset nodes; 2) nodes that are parts of trees; and 3) nodes that are neither cutset nodes or are parts of trees (have potential to become cutset nodes or to become part of a tree).

A node may mark itself as a cutset node by setting an indication ($CUTSET_i$= 1). A node is considered to be part of a tree if all its neighbors (except maybe one) point to it. The rest of the nodes may become cutset nodes using randomization. Similarly, only cutset nodes may be de-selected and transformed back into non-cutset kinds of nodes. The dynamic and randomized selection/de-selection process allows the search for the best cutset nodes to continue with

[9]Boltzmann, (with simulated annealing) or Meanfield (deterministic approximation to simulated annealing) activation functions may be used instead of Hopfield for the cutset nodes, thus increasing the chances of finding the "right" value.

non-increasing energy. As long as there are cycles, cutset nodes will be selected. At the same time, nodes functioning too long as cutset nodes are de-selected thus reducing the chances for redundant cutset nodes while continuously exploring the space of possible cutsets.

These ideas are incorporated into algorithm *activate-with-random-cutset*.

```
Algorithm activate-with-random-cutset:
Optimizing with cycle-cutset that is selected randomly with heuristics.

  1. Initialization: If first time, then $(\forall j)\ P_i^j = 0$; $CUTSET_i = 0$;

  2. Cutset setting:
     If all its neighbors except maybe one, point to it (part of a tree), then
        $CUTSET_i = 0$;
     else (not part of tree), if $CUTSET_i = 0$ (not yet a cutset node), then
        with probability $P = f()$ make $i$ a cutset node: $CUTSET_i = 1$.
     else (a previously selected cutset node) with probability $P = 1 - f()$ de-select node
     $i$
        and make it a non-cutset node: $CUTSET_i = 0$.

  3. Tree directing:
     If $CUTSET_i = 1$ (a cutset node), then for every neighbor $(j)$,
        if $p_j^i = 0$ then, $P_i^j = 1$;
     else (not a cutset node) if there exists a single neighbor $k$, such that $P_k^i = 0$ then,
           (part of a tree but not a root) $P_i^k = 1$; for all other neighbors $j$, $P_i^j = 0$;
     else, for all neighbors $P_i^j = 0$;

  4. Assigning activation values:
     If all neighbors of $i$ point to it except maybe one then (part of a tree),
        $X_i = \begin{cases} 1 & \text{if } \sum_j((G_j^1 - G_j^0)P_j^i + w_{i,j}X_jP_i^j) \geq -\theta_i \\ 0 & \text{otherwise} \end{cases}$
     else (a cutset node or a node that is not part of any tree), Compute Hopfield:
        $X_i = \begin{cases} 1 & \text{if } \sum_j w_{i,j}X_j \geq -\theta_i \\ 0 & \text{otherwise} \end{cases}$

  5. Computing goodness values: (only nodes in trees need goodness values)
     If $i$ is a cutset node, then for each neighbor $j$,
        $G_i^0 = X_i(\theta_i)$,
        $G_i^{j1} = X_i(\theta_i) + w_{ij}$
     else (a regular node),
        $G_i^0 = max\{\sum_{j \in children(i)} G_j^0 P_j^i, \sum_{j \in children(i)} G_j^1 P_j^i + \theta_i\}$;
        $G_i^1 = max\{\sum_{j \in children(i)} G_j^0 P_j^i, \sum_{j \in children(i)}(G_j^1 P_j^i + w_{i,j}P_i^j) + \theta_i\}$;
```

The heuristic function f may increase the cutset probability of a node based on the following guidelines:

1. Increase probability to nodes that have not been cutset nodes for a long time. Also, a node that has been a cutset node long enough should have a higher probability to become a non cutset node.

2. Increase probability to nodes that have not flipped their value for a long time. The more time that has passed, the more chances a node has to become a good cutset candidate. Also, more time gives more opportunities for the node to become part of a tree, thus avoiding premature decision.

3. Increase the probability of nodes with high connectivity; the more connections a node has, the more effective cutset node it may become (since it may cut several cycles at once).

We can start execution by assigning small probabilities to all nodes that are not part of a tree. As a result only few nodes would become cutset nodes, giving opportunities to other nodes to become part of a tree. With time, as we still have nodes that are not part of a tree, we may increase these probabilities, thus allowing more cutset nodes. Gradually increasing the probability will lead to solutions that are conditioned on fewer cutset nodes. At the same time, nodes that act too long as cutset nodes will become regular nodes giving opportunities to better cutset nodes to affect the process. During the whole process, energy never increases.

Further investigation and experimentation with the activate-with-random-cutset algorithm are left for future research.

6 Potential applications

Clearly, our improved algorithm will work better than the standard methods in networks that have large tree-like subnetworks or have only a few cycles. In this section we discuss shortly two domains that are most likely to produce sparse, near-tree networks. Assumptions and conditions are mentioned that cause the relevant networks to be mostly cycle free.

6.1 Inheritance networks

Inheritance is a straightforward example of an application, where translations of symbolic rules into energy terms form networks that are mostly cycle free.

Each arc of an inheritance network, such as A ISA B or A HAS B is modeled by the energy term $A - AB$. The connectionist network that represents the complete inheritance graph is obtained by summing the energy terms that correspond to all the ISA and HAS relationships in the graph.

Nonmonotonicity can be expressed if we add penalties to arcs and use the semantics discussed in [Pinkas 94]. Nonmonotonic relationships may cause cycles in both the inheritance graph and the connectionist network (eg. Penguin ISA Bird; Bird ISA FlyingAnimal; Penguin ISA not(FlyingAnimal)). Multiple inheritance may cause cycles as well, even when the rules are monotonic (eg., Dolphin ISA Fish; Dolphin ISA Mammal; Fish ISA Animal; Mammal ISA Animal).

Arbitrary constraints on the nodes of the graph may be introduced in this model. Constraints may be represented as proposition logic formulas and then translated into energy terms using the method in [Pinkas 90b].

In a "pure" inheritance network that has no multiple inherited nodes and no nonmonotonic relationships, the network is cycle-free. If we allow multiple inheritance, nonmonotonicity, or arbitrary propositional constraints, we may introduce cycles into the network that are generated.

Nevertheless, it is reasonable to assume that in large practical inheritance and taxonometric domains, the majority of the network is based on pure inheritance. Cycles (multiple inheritance, nonmonotonicity and arbitrary constraints) are scarcely introduced and the few that exist may be handled by our extension using the cycle-cutset idea.

6.2 Diagnosis

Another potential application that will generate mostly cycle-free subnetworks is diagnosis.

6.2.1 Formalism

Let $X_1, X_2, ...X_n$ be True(1)/false(0) propositions that represent symptoms and hypotheses. In a diagnosis application we may have the following sets of rules (formal semantics is discussed later):

- $(\alpha_1 X1, \alpha_2 X_2, ..., \alpha_m X_m >\!\!\rightarrow \beta X)$ i.e., the symptoms $X_1, ..., X_m$ with importance factors $\alpha_1, ..., \alpha_m$, suggest the hypothesis X with sensitivity β. A subset of the symptoms may be enough to suggest the hypothesis if the sum of the importance factors of the active symptoms is larger than the sensitivity β. Intuitively, the larger the sum of the factors, the larger the support for the hypothesis. By determining the support needed, the sensitivity β encourages parsimonious explanations of the symptoms. The corresponding energy function is $\sum_i^m -\alpha_i X_i X + \sum_i^m \alpha_i X_i + \beta X$.

- $(\alpha\varphi)$ where φ is a constraint expressed as a well formed propositional formula, and α is the importance of the constraint. Examples of such constraints are:

  - $(X \rightarrow X_i)$ i.e., if the hypothesis X holds, so does the symptom X_i.
  - $(X_1 \rightarrow (\neg X_2 \wedge \neg X_3) \wedge X_2 \rightarrow (\neg X_1 \wedge \neg X_3) \wedge X_3 \rightarrow (\neg X_1 \wedge \neg X_2))$ i.e., only one of the propositions X_1, X_2, X_3 can be true (mutual exclusion).

  Any propositional logic formula is allowed and nonmonotonicity may be expressed using conflicting constraints (augmented with importance factors). Quadratic energy function may be generated from arbitrary propositional constraints by introducing hidden variables (see [Pinkas 90b] for details).

Note that any variable X may be used as a symptom in some rules and as a hypothesis in other rules. This observation will be used later in our discussion of independence.

6.2.2 Semantics

We can use a formal logic and semantics called penalty logic [Pinkas 94], [Pinkas 91] that is based on ranked models [Shoham 88], [Pearl 90], [Lehmann, Magidor 88]. In this semantics, models are ranked according to a score computed by summing the penalties associated with violating the rules. Propositions that are held in all the models of minimal penalty are concluded.

For a diagnosis rule $(\alpha_1 X_1, ..., \alpha_k X_k \rightarrow \beta X)$, a model is penalized by adding $(\sum_{j=1}^k \alpha_{i_j})$ if the symptoms $X_{i_1}, ..., X_{i_k}$ are active and the hypothesis X inactive. To encourage parsimonious explanations, β is always added when the corresponding hypothesis (X) is activated. For violating a constraint $\alpha\varphi$, a model is penalized by adding an extra α.

A set of rules such as the above can be translated into an energy function and be implemented as a connectionist network. Penalty logic, its semantics, and the algorithms for translating formulas into energy functions are discussed in [Pinkas 94].

6.2.3 Cycles, trees and independence assumption

Assume for now that our knowledge base consists only of diagnosis rules ($\alpha_1 X_1, ..., \alpha_m X_m \rightarrow \beta X$) and (optional) causal constraints of the type ($X \rightarrow X_i$).

We say that the symptoms of hypothesis X are *independent* of each other, when all its direct symptoms (neighbors) (of X), given a value for X, are not affected by the value of any other direct symptom of X.

When the symptoms of any hypothesis in a knowledge base are independent of each other there are no cycles in the network and the tree algorithm converges to a global maximum in linear time.

When we start adding symptoms that affect a hypothesis through different paths; e.g., $X_1 \rightarrow X$, and $X_1 \rightarrow X_2 \rightarrow ... \rightarrow X$, or when we start adding arbitrary constraints, cycles may be added.

Independency assumptions of this kind are quite common in actual implementations of Bayes networks, influence diagrams [Pearl 88], and certainty propagation of rule-based expert systems [Shortliffe 76].

7 Summary

We have shown a uniform self-stabilizing connectionist activation function that is guaranteed to find a global minimum of acyclic symmetric networks in linear time under a realistically weakened distributed scheduler. It can also be used to improve local repair techniques when the connectivity of the problem variables form tree-like subnetworks. For general (cyclic) topologies we show how our tree-like algorithm can be extended using cutset nodes that force a network to become a collection of tree-like sub-networks. The algorithm optimizes tree-like subnetworks within general networks but is not self-stabilizing when used in cyclic topologies. Variations of the cutset algorithm allow the cutset nodes to be selected a priori or to be selected randomly with local heuristics and use a uniform activation algorithm. Another variation allows a global minimum to be found by enumerating all combinations of cutset values.

Two domains (inheritance and diagnosis) are brought as examples for applications that will benefit in particular from the new optimization algorithm. Networks of these domains are likely to have near tree topologies having only few cycles.

We stated two negative results: 1) Under a pure distributed scheduler no *uniform* algorithm exists to globally optimize even simple chain-like networks. 2) No uniform algorithm exists to globally optimize simple cyclic networks (rings) even under a central scheduler. We conjecture that these negative results are not of significant practical importance since in realistic schedulers the probability of having pathological scheduling scenarios approaches zero. We show that our algorithm converges correctly (on tree-like subnetworks) when the demand for pure distributed schedulers is somewhat relaxed (adding either fair exclusion, almost uniformity or cycles). Similarly, self-stabilization is obtained in acyclic networks or when the requirement for a uniform algorithm is relaxed (adding almost uniformity).

The negative results apply to connectionist algorithms as well as to parallel versions of local repair techniques. The positive results suggests improvements both to connectionist activation functions and to local repair techniques.

8 Appendix

Proof sketch: of theorem 4.3:
The second and third phases of the algorithm are adaptations of an existing dynamic programming algorithm [Bertelé, Brioschi 72], and their correctness is therefore not proved here. The self-stabilization of these steps is obvious because no variables are initialized. The proof therefore concentrates on the tree directing phase.

Let us first assume that the scheduler is distributed with fair exclusion and that the network is a tree. We now prove the first part of the theorem. We want to show that the algorithm converges, that it is self-stabilizing and that the final stable result is that the pointers P_i^j represent a tree. We will prove the other parts of the theorem toward the end.
Definitions:
A node is *legal* if it is either a root (i.e., all its neighbors are legal, point to it and it doesn't point to any of them), or an intermediate node (i.e., it points to

one of the neighbors and the rest of its neighbors are all legal and point back). A node is a *candidate* if it is an illegal node and has all its neighbors but one pointing to it and legal.
We must show:
1) The property of being legal is stable; i.e., once a node becomes legal it will stay legal.
2) A state where the number of illegal nodes is $k > 0$, leads to a state where the number of illegal nodes is less than k; i.e., the number of illegal nodes decreases and eventually all nodes turn legal.
3) If all the nodes are legal then the graph is marked as a tree.
4) The algorithm is self-stabilizing for trees.
5) The algorithm converges even if the graph has cycles (part 2 of the theorem).
6) The algorithm is self-stabilizing in arbitrary networks if an almost uniform version is used, even under a distributed scheduler (part 3 of the theorem).
Proof:
1) Show that a legal state is stable. Assume a legal node i becomes illegal. It is either a root node and one of its children became illegal, or an intermediate node whose one of its children became illegal (it cannot be that its parent suddenly points to i or that one of the children stopped pointing and still is legal). Therefore, there must be a chain of $i_1, i_2, ..., i_k$ of nodes that became illegal. Since there are no cycles, there must be a leaf that was legal and turned illegal. This cannot occur since a leaf does not have children, leading to a contradiction.

2) Show that if there are illegal nodes, their number is reduced. To prove this claim we need three steps: 2.1) Eventually a state is reached where if there is at least one illegal node then there is also a candidate node among the illegal nodes. 2.2) This candidacy is stable. 2.3) eventually the candidate will become legal (therefore the number of illegal nodes is reduced).

2.1) Because of the fair execution, eventually a state is reached where each node has been executed at least once. Assume that at least one node is illegal, and all the illegal nodes are not candidates. If a node is illegal and not a candidate, then either it is a root-type (all point to it) but at least one of its children is illegal, or there are at least two of its neighbors that are illegal. Suppose there are no root-type illegal nodes. Then all illegal nodes have at least two illegal neighbors. Therefore there must be a cycle that connects illegal nodes (contradiction). Therefore, one of the illegal nodes must be root-type. Suppose i is a root-type illegal node. It must have a neighbor j which is illegal. Consider the subtree of j that does not include i: it must contain illegal nodes. If there are no root-type illegal nodes we get a contradiction again. However, if there is a root-type node, we eliminate it and look at the subtree of some illegal j' that does not include j. Eventually, since the network is finite, we obtain a subtree with no root-like illegal nodes but which includes other illegal nodes. This leads to a contradiction. The conclusion is that there must be candidates if there are illegal nodes.

2.2) Show that a candidate is stable unless it becomes legal.
If a node i is a candidate, all its legal children remain legal. There are three types of candidate nodes (node j is an illegal neighbor of i): 1) node j points to i; 2) the pointer goes in both directions; 3) there is no pointer from i to j or vice-versa. All possible changes in the pointers P_i^j or P_j^i will cause i to remain a candidate or to turn legal (the rest of the pointers will not be changed).

2.3) Show that every candidate node will eventually turn legal: assume j is the illegal neighbor of the candidate i. In the next execution of i without j, if $P_j^i = 0$ then i becomes legal by pointing to j; otherwise, i becomes a root-type candidate (all its neighbors point to it) but j is illegal. We will prove now that if an illegal node j points to i then eventually a state is reached where either j is legal or $P_j^i = 0$, and that this proposition is stable once it holds. If this statement is true then i is executed eventually: if j is legal then all of $i's$ neighbors are legal and therefore i turns legal. If j is illegal then $P_j^i = 0$, and i will point to it ($P_i^j = 1$) making itself legal.

To prove: if j is an illegal node pointing to i then there will be a state where either j is legal or $P_j^i = 0$, and this state is stable.
We prove it by induction on the size of the subtree of j that does not include i.
Base: If j is a leaf and j points to i then if at the time j is executed (without i) $P_i^j = 0$, then node j points to i and turns legal; otherwise, j updates $P_j^i = 0$. This status is stable because the legal state is stable and since a leaf will point to a node only if it turns legal.
Induction step: Assume hypothesis is true for trees of size less than n. Suppose j is the illegal neighbor if i. Node j points to i and it has $j_1, ..., j_k$ other

neighbors. Because we assume that all nodes were executed at least one time, since j points to i we assume that at the last execution of j all the other neighbors $j_1, ..., j_k$ pointed to j. The subtrees rooted by j_l (not including j) are of size n and therefore by the hypothesis there will be a state where all the nodes $j_1, ..., j_k$ are either legal or $P^j_{j_l} = 0$. This state is stable, so when eventually j is executed, it will either point to i turning legal (if all $j_1, ..., j_k$ are pointing to it), or it will make $P^i_j = 0$ (if some of its neighbors do not point to it). Since the status of $j_1, ..., j_k$ is stable at that point, whenever j is executed it will either become legal or its pointers become zero.

3) Show that if all the nodes are legal then the graph is marked as a tree: If a node is legal, then all its children are legal and point to it. Therefore each node represents a subtree (if not a leaf) and has one parent at the most. To show that there is only one root we make the following argument. If several roots exist, then because of connectivity, there is one node that is shared between at least two subtrees and therefore has two parents (contradiction).

4) The algorithm is self-stabilizing for cycle-free networks since no initialization is needed (in the proof we haven't use the first initialization step; i.e., $P^j_i = 0$). In the case where no cycles exist we do not need this step. The pointers can get any initial values and the algorithm still converges.

5) The algorithm (with $P^j_i = 0$ initialization) converges even if the graph has cycles. Since all the nodes start with zero pointers, a (pseudo) root of a tree-like subnetwork will never point toward any of its neighbors (since it is part of a cycle and all of its neighbors but one must be legal).

6) Show that the algorithm is self-stabilizing in arbitrary networks if an almost uniform version is used, even under a distributed scheduler. We need to show that a candidate will eventually turn legal even if its neighbors are executed in the same time.
Suppose node i is a candidate and node j is its illegal neighbor:
1) If j is a root, then it will never point to i, and therefore i will eventually turn legal by pointing to j.
2) If i is the root, then $P^j_i = 0$, and if j becomes legal it will point to i making i legal. Node j will turn eventually legal using the following induction (on the size of the subtree of j).

Hypothesis: In a subtree without a node that acts as a root, all illegal nodes will eventually turn legal.
Base: If j is a leaf, it will point eventually to its neighbor i which in its turn will make j legal by $P_i^j = 0$.
Induction: If $j_1, ..., j_k$ are other neighbors of j, then they will eventually turn legal (induction hypothesis) while pointing to j. Eventually j is executed and also turns legal.
3) Suppose neither i nor j are roots, but one of them is not part of a cycle (and therefore is part of a subtree that does not include a node marked as a root). Using the above induction, all the nodes in the subtree will eventually turn legal. As a result either i or j eventually turns legal, and therefore i will eventually turn legal as well.
$\square$

References

[Ballard et al. 86] D. H. Ballard, P. C. Gardner, M. A. Srinivas, "Graph problems and connectionist architectures," University of Rochester, Technical Report 167, 1986.

[Becker, Geiger, 1994] A. Becker and D. Geiger, "Approximation algorithms for loop cutset problems" *Proceedings of the 10th conference on Uncertainty in Artificial Intelligence (UAI-94)*, pp. 60–68, Seattle Washington, August, 1994.

[Bertelé, Brioschi 72] U. Bertelé, F. Brioschi, "Nonserial dynamic programming," Academic Press, 1972.

[Brandt et al. 88] R. D. Brandt, Y. Wang, A. J. Laub, S. K. Mitra, "Alternative networks for solving the traveling salesman problem and the list-matching problem," in *IEEE International Conference on Neural Networks,* Vol 2,pp. 333–340, 1988.

[Collin et al. 91] Z. Collin, R. Dechter, S. Katz, "On the feasibility of distributed constraint satisfaction," *Proceedings of IJCAI*, Sydney, 1991.

[Dechter 90] Dechter, R., "Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition," *Artificial Intelligence, Vol. 41(3)*, pp. 273–312, January 1990.

[Dechter 90] Dechter, R., "Constraint networks." In *Encyclopedia of Artificial Intelligence*, 2nd ed., John Wiley & Sons, Inc., pp 276–285, 1992.

[Dechter et al 90] R. Dechter, A. Dechter, J. Pearl, "Optimization in constraint networks," in R.M. Oliver, J.Q. Smith, Influence diagrams, belief nets and decision analysis, John Wiley and Sons, 1990.

[Dijkstra 74] E.W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM 17,* no. 11, pp. 643–644, 1974.

[Even, 79] S. Even "Graph Algorithms" Computer Science Press, 1979.

[Feldman, Ballard 82] J.A Feldman, D.H. Ballard, "Connectionist models and their properties," *Cognitive Science* 6, 1982.

[Hinton, Sejnowski 86] G.E Hinton and T.J. Sejnowski, "Learning and relearning in Boltzmann Machines," in J. L. McClelland and D. E. Rumelhart, *Parallel Distributed Processing: Explorations in The Microstructure of Cognition I,* pp. 282–317, MIT Press, 1986.

[Hopfield 82] J. J. Hopfield "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences 79,* pp. 2554–2558, 1982.

[Hopfield 84] J. J. Hopfield "Neurons with graded response have collective computational properties like those of two-state neurons," *Proceedings of the National Academy of Sciences 81*, pp. 3088–3092, 1984.

[Hopfield, Tank 85] J.J. Hopfield, D.W. Tank "Neural Computation of Decisions in Optimization Problems," *Biological Cybernetics 52*, pp. 144–152.

[Kasif et al. 89] S. Kasif, S. Banerjee, A. Delcher, G. Sullivan "Some results on the computational complexity of symmetric connectionist networks," Department of Computer Science, The John Hopkins University, Technical Report JHU/CS-89/10, 1989.

[Korach et. al, 84] K. Korach, D. Rotem, and N. Santoro "Distributed algorithms for finding centers and medians in networks" *ACM transaction on Programming Languages and Systems* 6(3):380–401, July 1984.

[Lehmann, Magidor 88] D. Lehmann, M. Magidor, "Rational logics and their models: A study in cumulative logic", Technical Report, TR-86-16, Leibnitz Center for Computer Science, Hebrew University, Jerusalem, 1988.

[McClelland et al. 86] J. L. McClelland, D. E. Rumelhart, G.E Hinton, "The appeal of PDP," in J. L. McClelland and D. E. Rumelhart, *Parallel Distributed Processing: Explorations in The Microstructure of Cognition I*, MIT Press, 1986.

[Minton et al 90] S. Minton, M. D. Johnson, A. B. Phillips, "Solving large scale constraint satisfaction and scheduling problems using a heuristic repair method,", *Proceedings of the Eighth Conference on Artificial Intelligence*, pp. 17–24, 1990.

[Papadimitriou et al. 90] C. Papadimitriou, A. Shaffer, M. Yannakakis, "On the complexity of local search," *ACM Symposium on the Theory of Computation,* pp. 438–445, 1990.

[Pearl 88] J. Pearl "Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference" Morgan Kaufmann Publishers, San Mateo, California, 1988.

[Pearl 90] J. Pearl, "System Z: A natural ordering of defaults with tractable applications to nonmonotonic reasoning," in *Proceedings of the Workshop on Theoretical aspects of Knowledge Representation* (TARC), M. Vardi (ed.), pp. 121–135, Morgan Kaufmann, 1990.

[Peterson, Hartman 89] C. Peterson, E. Hartman, "Explorations of mean field theory learning algorithm," *Neural Networks 2,* no. 6, 1989.

[Pinkas 90b] G. Pinkas, "Energy minimization and the satisfiability of propositional calculus," *Neural Computation 3,* no. 2, 1991.

[Pinkas 91] G. Pinkas, "Propositional non-monotonic reasoning and inconsistency in symmetric neural networks," *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pp. 525–530, Sydney, 1991.

[Pinkas, Dechter 92] G. Pinkas, R. Dechter, "A new improved activation function for energy minimization," *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI),* pp. 434–439, San Jose, 1992.

[Pinkas 94] G. Pinkas, "Reasoning, nonmonotonicity and learning in connectionist networks that capture propositional knowledge," to appear in *Artificial Intelligence Journal.*

[Rumelhart et al. 86] D.E. Rumelhart, G.E Hinton, J.L. McClelland, "A general framework for parallel distributed processing," in J. L. McClelland and D. E. Rumelhart, *Parallel Distributed Processing: Explorations in The Microstructure of Cognition I*, MIT Press, 1986.

[Selman et al. 92] B. Selman, H. Levesque, D. Mitchell, "A new method for solving hard satisfiability problems," *Proceedings of the Tenth National Conference on Artificial Intelligence,* pp. 440–446, 1992.

[Shoham 88] Y. Shoham, *Reasoning about Change,* MIT Press, Cambridge, 1988.

[Shortliffe 76] E. H. Shortliffe, *Computer-based medical consultation, Mycin*, Elsevier, New York, 1976.

[Smolensky 86] P. Smolensky, "Information processing in dynamical systems: Foundations of harmony theory," in J.L.McClelland and D.E.Rumelhart, *Parallel Distributed Processing: Explorations in The Microstructure of Cognition I* , MIT Press, 1986.