On the Feasibility of Distributed Constraint Satisfaction

Zeev Collin
CS, Technion —
Israel Institute of Technology,
Haifa, 32000, Israel

Rina Dechter*
Information and CS, UCI,
Irvine, CA, 92717

Shmuel Katz
CS, Technion —
Israel Institute of Technology,
Haifa, 32000, Israel

Abstract

This paper characterizes connectionist-type architectures that allow a distributed solution for classes of constraint-satisfaction problems. The main issue addressed is whether there exists a **uniform** model of computation (where all nodes are indistinguishable) that guarantees convergence to a solution from every initial state of the system, whenever such a solution exists. We show that even for relatively simple constraint networks, such as rings, there is no general solution using a completely uniform, asynchronous, model. However, some restricted topologies like trees can accommodate the uniform, asynchronous, model and a protocol demonstrating this fact is presented. An **almost-uniform**, asynchronous, network-consistency protocol is also presented. We show that the algorithms are guaranteed to be self-stabilizing, which makes them suitable for dynamic or error-prone environments.

1 Introduction

Consider the distributed version of the graph coloring problem, where each node must select a color (from a given set of colors) that is different from any color selected by its neighbors. This coloring task, whose sequential version is known to be NP-complete, belongs to a class of **Constraint Satisfaction Problems (CSPs)** that present interesting challenges to distributed computation, particularly in the framework of connectionist architectures. We call the distributed versions of such problems **Network Consistency Problems (NCPs)**. We consider what types of distributed models admit a self-stabilizing algorithm (namely, one that converges to a solution from any initial state of the network), and present such algorithms when possible.

*This work was partially supported by the National Science Foundation, Grant #IRI-8821444 and by the Air Force Office of Scientific Research, Grant #AFOSR-90-0136.

Constraints are useful in programming languages, simulation packages and general knowledge representation systems, and the prospects of solving such problems by connectionist networks promise the combined advantages of parallelism, simplicity of design and error correction capabilities.

Indeed, many interesting problems attacked by researchers in neural networks are combinatorial and many involve constraint satisfaction [Ballard *et al.*, 1986, Dahl, 1987]. In fact, any discrete state connectionist network can be viewed as a type of constraint network, with each stable pattern of states representing a consistent solution. However, current connectionist approaches to CSPs lack theoretical guarantees of convergence (to a solution satisfying all constraints), and the terms on which such convergence can be guaranteed (if at all) have not been explored till now.

In this paper we show that widely used connectionist-type architectures in which all nodes run identical procedures cannot admit algorithms that guarantee convergence to a consistent solution, even if such a solution exists (Section 2). We then identify a distributed model that is close in spirit to the connectionist paradigm, for which such guarantees can be established (Section 3). Within this model, we characterize and provide algorithms for a restricted subclass of networks that can be solved uniformly (Section 4).

2 Model and Definitions

2.1 CSP definition

A network of binary constraints involves a set of n variables $X_1, \ldots, X_n$, each represented by its domain values, $D_1, \ldots, D_n$, and a set of constraints. A **binary constraint** R_{ij} between two variables X_i and X_j is a subset of the Cartesian product $D_i \times D_j$ that specifies which values of the variables are compatible with each other. A solution is an assignment of values to all the variables which satisfies all the constraints, and the constraint satisfaction problems associated with these networks are to find one or all solutions. A binary CSP can be associated with a **constraint-graph** in which nodes represent variables and arcs connect pairs of variables which are constrained explicitly. (General constraint satisfaction problems may involve constraints of any arity, but since network communication is only pairwise we focus on this subclass of problems.) Figure 1a presents a CSP constraint graph.

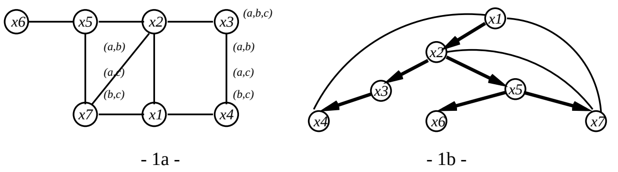

Figure 1: CSP - A constraint graph and its DFS-tree

2.2 The model

Our general communication model is known as the **shared memory multi-reader single-writer** model. A distributed network consists of n nodes, connected by shared communication registers, called **state registers**. The network can be viewed as a **communication graph** where nodes represent processors and arcs correspond to communication registers. The register $state_i$ is written only by node i, but may be read by several nodes (all of i's neighbors). The state register may have several fields, but it is regarded as one unit. A node can be modeled as a finite state-machine where its state is controlled by a **transition function** that is dependent on its current state and the states of its neighbors. In other words, an activated node performs an **atomic step** consisting of reading the states of all its neighbors, deciding whether to change its state and then moving to a new state (by writing in its state register)[1]. The collection of all transition functions is called a **protocol**. The processors are anonymous, i.e. , have no identities (we use the terms node i and processor P_i interchangeably and as a writing convenience only). A configuration c of the system is the state vector of all processors.

The execution of the system can be managed either by **a central demon** (scheduler) defined in [Dijkstra, 1974, Dolev *et al.*, 1990] or by **a distributed demon** defined in [Burns *et al.*, 1987, Dolev *et al.*, 1990]. The distributed demon activates a subset of the system's nodes at each step, while the central demon activates only one processor at a time (and thus can be viewed as a simplified version of the distributed demon). All activated nodes execute a single atomic step simultaneously.

The central demon means that an interleaving is sufficient for the analysis of the protocol. Nevertheless, on the implementation level, truly independent nodes can execute in parallel since they cannot affect each other. Only neighboring nodes in the constraint graph cannot execute at the same atomic step when a central demon is assumed.

We denote $c_1 \rightarrow c_2$ if c_2 is a configuration which is reached from configuration c_1 by some subset of processors simultaneously executing a single atomic step. An **execution** of the system is an infinite sequence of configurations $E = c_1, c_2 \ldots$ such that for every i, $c_i \rightarrow c_{i+1}$. An execution is considered **fair** if every node participates in it infinitely often.

2.3 Self-stabilization

A self-stabilizing protocol [Dijkstra, 1974] is one with a particular convergence property. The system configuration-space is partitioned into two classes — legal, denoted by L, and illegal. The protocol is self-stabilizing if in any infinite fair execution, starting from any initial configuration (and with any input values) and given "enough time", the system eventually reaches a legal configuration, and all subsequently computed configurations are legal. Thus a self-stabilizing protocol converges from any point in its configuration-space to a stable, legal region.

The legality of a configuration depends on the aim of the protocol. In our case, we wish to design a protocol for solving the network consistency problem. Thus, the set of legal configurations are those having a consistent assignment of values to all the nodes in the network, if such an assignment exists, and any set, otherwise. This definition allows the system to oscillate among various solutions, if more than one consistent assignment is possible. However, the protocols that are presented in this paper converge to one of the possible solutions.

2.4 The limits of uniform self-stabilization

A protocol is **uniform** if all the nodes are logically equivalent and identically programmed (i.e. have identical transition functions). Following an observation made by Dijkstra [Dijkstra, 1974] regarding the computational limits of a uniform model for performing the mutual exclusion task, we show that the network consistency problem cannot be solved using a uniform protocol. This is accomplished by presenting a specific constraint network and proving that its convergence cannot be guaranteed using **any** uniform protocol.

Consider the task of numbering a ring of processors in a cyclic ascending order — we call this CSP the **"ring ordering problem"**. The constraint-graph of the problem is a ring of nodes with the domains $\{0,\ 1,\ \ldots,\ n-1\}$. Every arc has the set of constraints $\{(i,\ (i+1) \bmod n) |\ 0 \le i \le n-1\}$ i.e., the left node is one smaller than the right one. A solution to this problem is a cyclic permutation of the numbers $0, \ldots, n-1$, which means that there are n possible solutions, and in all of them different nodes are assigned different values.

Theorem 1: No uniform, self stabilizing protocol can solve the ring ordering problem, in the presence of a

[1] In fact, a finer degree of atomicity, requiring only a **test-and-set** operation, is possible, but is not used here in order to simplify the arguments.

central demon.

Proof: In order to obtain a contradiction, assume that there exists a uniform self-stabilizing protocol for solving the problem. In particular, it would solve the ring-ordering problem for a ring having a composite number of nodes, $n = r \cdot q$ $(r, q > 1)$. Since convergence to a solution is guaranteed from any initial configuration, it also applies to one where all nodes are in identical states. We construct a fair execution for such a protocol for which the network never converges to a consistent solution, contradicting the self stabilization property of the protocol. Assume the following execution:

$$\begin{array}{lllll} P_0, & P_q, & P_{2q}, & \ldots, & P_{(r-1)q}, \\ P_1, & P_{q+1}, & P_{2q+1}, & \ldots, & P_{(r-1)q+1}, \\ \vdots & & & & \\ P_{q-1}, & P_{2q-1}, & P_{3q-1}, & \ldots, & P_{rq-1}, \\ \vdots & & & & \end{array}$$

Note that nodes P_0, P_q, P_{2q}, $\ldots$,$P_{(r-1)q}$, after their first activation, move to identical states because their inputs, initial states and transition functions are identical, and when each one of them is activated their neighbors are in identical states too. The same holds for any sequential activation of processors $\{P_{iq+j} | \ 0 \leq i < r, \ 0 \leq j < q\}$. Thus, cycling through the above schedule assures that P_0 and P_q, for instance, move to identical states over and over again an infinite number of times. Since a consistent solution requires their states to be different, the network will never reach a consistent solution, thus yielding a contradiction. □

Theorem 1 implies that it is generally impossible to guarantee convergence to a consistent solution using a uniform protocol. It also implies that such convergence cannot be guaranteed for a class of sequential algorithms using so called "repair" methods, such as in [Minton *et al.*, 1990]. It does not, however, exclude the possibility of existence of uniform protocols for restricted activation policies.

We can also show that, when using a distributed demon, convergence (to a solution) cannot be guaranteed even for **tree-networks**. Consider, for instance, the coloring problem in a tree-network constructed from exactly two connected nodes each having the domain {BLACK, WHITE}. Since the two nodes are topologically identical, If they start from identical initial states and both of them are activated simultaneously, they can never be assigned different values. Consequently, the network does not converge to a legal solution, although one exists. This counterexample can be extended to a large class of trees, where there is no possible way to distinguish between two internal nodes. We will show, however, (section 4) that for a central demon a uniform self stabilizing tree-network consistency protocol does exist.

Having proved that the network consistency problem cannot be solved using a uniform protocol, even with a central demon, we switch to a slightly more relaxed model of an "**almost uniform**" protocol, whereby all nodes but one are identical. We denote the special node as P_0.

3 Consistency-Generation Protocol

Our network consistency protocol is based on a sequential version of a **backtracking** algorithm, called **backjumping**. When implemented on a variable ordering generated by a **depth-first traversal** of the constraint graph, the technique enables a distributed implementation. A preliminary version of this protocol appears in [Collin and Dechter, 1990].

3.1 Sequential aspects of constraint satisfaction

The most common algorithm for solving a CSP is backtracking. In its standard version, the algorithm traverses the variables in a predetermined order, provisionally assigning consistent values to a subsequence $(X_1, \ldots, X_i)$ of variables and attempting to append to it a new instantiation of X_{i+1} such that the whole set is consistent. If no consistent assignment can be found for the next variable X_{i+1}, a deadend situation occurs; the algorithm "backtracks" to the most recent variable, changes its assignment and continues from there.

One useful improvement of backtracking, called **backjumping** [Dechter, 1990] consults the topology of the constraint graph to guide its "backward" phase. Specifically, instead of going back to the most recent variable instantiated it **jumps back** several levels to the first variable **connected** to the deadend variable. It turns out that when using a depth-first search (DFS) on the constraint graph (to generate a DFS tree) and then conducting backjumping in an inorder traversal of the DFS tree, [Even, 1979] the jump-back destination of variable X is the parent of X in the DFS tree.

The nice property of a DFS tree that allows a parallel implementation is that any arc of the graph which is not in the tree connects a node to one of its tree ancestors (i.e. along the path leading to it from the root). Consequently, the DFS tree represents a useful decomposition of the graph: if a variable X and all its ancestors are removed from the graph, the subtrees rooted at X will be disconnected (Figure 1b). This translates to a problem-decomposition strategy: if all ancestors of variable X are instantiated, then the solutions of all its subtrees are completely independent and can be performed in parallel [Freuder and Quinn, 1987].

3.2 General Network Consistency protocol

The network-consistency (NC) protocol is logically composed of two self-stabilizing subprotocols that can be executed interleaved (we divide the second subprotocol into two parts in order to simplify the explanation):

1. DFS-tree generation

2. (a) graph-traversal protocol
   (b) value-assignment

These subprotocols are unrelated to each other and, thus, can be independently replaced by any other version of implementation.

The basic idea of the protocol is to decompose the network into several independent subnetworks, according to the DFS-tree structure, and to instantiate these subnetworks in parallel. A proper order of value instantiation is guaranteed by the graph traversal protocol.

3.2.1 Neighborhoods and states

A self-stabilizing algorithm for generating a DFS-tree is presented in [Collin and Dolev, 1991] and will not be discussed here. This subprotocol is almost uniform and is the source of non-uniformity for the whole NC protocol. When the algorithm stabilizes each internal node, i, has one adjacent node, $parent(i)$, designated as its **parent** in the tree, and a set of **child** nodes denoted $children(i)$. Figure 2 indicates the environment of an internal node (2a), the root (2b), and a leaf (2c). The link leading from $parent(i)$ to i is called i's **inlink** while the links connecting i to its children are called i's **outlinks**. The set of its neighboring nodes along the path from the **root** to i are called i's **predecessors**. The role of the root is played by the special processor P_0. Each node i (representing variable X_i) has a list of possible values, denoted as $Domain_i$, and a pairwise relation R_{ij} with each neighbor j. The domain and the constraints may be viewed as a part of the system or as inputs that are always valid (though they can be changed during the execution, forcing the network to readjust itself to the changes).

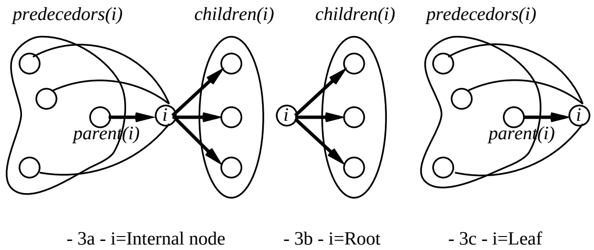

Figure 2: Node's neighborhood set

The state-register of each node contains the following fields:

1. A ***value*** field to which it assigns either one of its domain values or the symbol "$\star$" (to denote a dead-end).

2. A ***mode*** field indicating the node's "belief" regarding the status of the network. A node's mode is ON if its value or its ancestors' values were changed since the last time it was in a forward phase, or otherwise it is OFF . The modes of all nodes also give an indication of whether they have reached a consistent state (all in an OFF mode).

3. Two boolean fields called ***parent_tag*** and ***children_tag***, which are used by the graph-traversal protocol (Section 3.2.2).

3.2.2 Graph-traversal protocol

The graph-traversal protocol is handled by a self-stabilizing **privilege passing mechanism**. According to this protocol a node obtains a privilege to act, granted to it either by its parent or by its children. A node is allowed to change its state only if it is privileged.

Our privilege passing mechanism is an extension of a mutual exclusion protocol for two nodes called **balance-unbalance** [Dijkstra, 1974, Dolev *et al.*, 1990]. Once a DFS-tree is established, this scheme is implemented by having every state register contain two fields: *parent_tag*, referring to its inlink and *children_tag*, referring to all its outlinks. A link is **balanced**, if the *children_tag* and the *parent_tag* on its endpoints have the same value, and the link is **unbalanced** otherwise. A node, i, becomes privileged if its inlink is unbalanced and **all** its outlinks are balanced[2]. The privilege can be passed backwards to the parent by balancing the incoming link or forward to the children by unbalancing the outgoing links (i.e. by changing the *parent_tag* or the *children_tag* value accordingly). A node applies the NC-protocol only when it is privileged, otherwise it leaves its state unchanged.

Denote a **branch** to be a tree-path from the root to a leaf. The privilege-passing mechanism eventually converges to a set of **legally controlled** configurations, in which no more than one node is privileged on every branch. Figure 3 shows such a configuration (the *parent_tag* and the *children_tag* of every node are specified above and below the node respectively). This property assures that eventually a node and its ancestors cannot reassign their values simultaneously. The privileges travel along the branches backwards and forwards. We omit the proof due to space limitations.

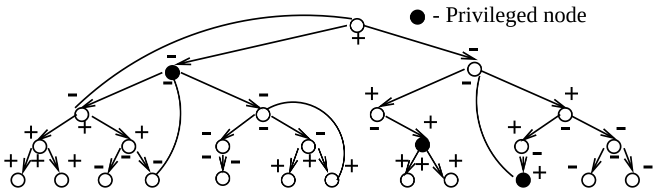

Figure 3: Legally controlled configuration

3.2.3 Value-assignment

The value-assignment has a **forward** and a **backward** phases, corresponding to the two phases of the sequential backtracking algorithm. During the forward phase,

[2]Note that this is well defined since we prove that eventually all siblings have the same parent-tag.

nodes in different subtrees assign themselves values consistent with their predecessors or verify the consistency of their assigned values. When a node senses a deadend it assigns its *value* field a "$\star$" and initiates a backward phase. When the network is consistent (all the processors are in an OFF *mode*) the forward and backward phases continue, where the forward phase is used to verify the consistency of the network and the backward phase just returns the privilege to the root to start a new forward wave. Once consistency verification is violated, the node sensing the violation initiates a new value-assignment. Since the root has no ancestors, it does not check consistency. It only assigns a new value at the end of each backward phase, when needed, and then initiates a new forward phase. A more elaborate description follows.

An internal node can be in one of three situations:

- **Node i is activated by its parent which is in an ON mode** (this is the forward phase of value assignments). In that case some change of value in one of its predecessors might have occurred. It, therefore, finds the first value in its domain that is consistent with all its predecessors, puts itself in an ON mode and passes the privilege to its children. If no consistent value exists, it assigns itself the "$\star$" value (a deadend) and passes the privilege to its parent (initiating a backward phase).

- **Node i is activated by its parent which is in an OFF mode.** In that case it verifies the consistency of its current value with its predecessors. If it is consistent it stays in an OFF mode and passes the privilege to its children. If not, it assigns itself a new value, moves to an ON mode, and passes the privilege to its children[3].

- **Node i is activated by its children** (backward phase). If one of the children has a "$\star$" value, the processor selects the next consistent value from its domain and passes the privilege to the children. If no consistent value is available, it assigns itself a "$\star$" and passes the privilege to its parent[4]. If all children have a consistent value, i passes the privilege to its parent.

The value-assignment protocol is uniform since each node has both the root's protocol and the non-root's protocol and will decide between them based on the role assigned to it by the DFS-tree protocol.

[3]A leaf, having no children, is always activated by its parent and always passes the privilege back to its parent (initiating a backward phase).

[4]Due to the privilege passing mechanism, when a parent sees one of its children in a deadend it has to wait until **all** of them have given him the privilege. This is done to guarantee that all subtrees have a consistent view regarding their predecessors' values.

The self-stabilization property of the NC protocol is inherited from its subprotocols; DFS-tree generation, privilege-passing and value-assignment. Once the self-stabilization of privilege-passing is established, it assures the adequacy of the control for distributedly implementing DFS-based backjumping, which guarantees the convergence of the network to a legal solution, if one exists, and if not it keeps checking all the possibilities over and over again.

3.3 Complexity analysis

A crude estimate of the time complexity of the NC protocol can be given by computing the maximal number of state changes from the time the privilege-passing has stabilized until final convergence. The search space that is generated by the sequential DFS-backjumping obeys the following recurrence: $T_m = 1+b\cdot k\cdot T_{m-1}$ with $T_0 = 1$, which yields $T_m = (b\cdot k)^{m+1}$ where T_m stands for the search space generated by sequential DFS-backjumping with depth m or less, b is the maximal branching degree and k bounds the domain sizes. Our bound improves the one presented in [Freuder and Quinn, 1987]. Note that since $b^m < n$ we get that $T_m = O(nk^{m+1})$. Clearly, in the worst parallel execution we get a sequential behavior with the same time complexity — exponential in the depth of the DFS-tree. However, often, due to parallel instantiations of different subtrees, the parallel protocol may have, on the average, a significant speedup over the sequential one. We believe that the speedup (of our protocol over the same sequential algorithm) is of $O(n/m)$. As an extreme example, consider problem instances that have a backtrack-free solution along the DFS ordering. These will be solved in $O(n)$ sequentially, while in $O(m)$ in parallel. To conclude, our protocol convergence can be achieved in polynomial time for networks with a bounded depth of the DFS-tree.

The average performance of the NC protocol can be further improved by adding to it a uniform self-stabilizing **arc-consistency** subprotocol [Mackworth and Freuder, 1985]. A network is said to be **arc-consistent** if for every value in each node's domain there is a consistent value in all its neighbors' domains. Arc-consistency can be achieved by a repeated execution of a "relaxation procedure", where each node reads its neighbors' domains and eliminates any of its own values for which there is no consistent value in one of its neighbors' domains. This protocol is clearly self-stabilizing.

4 Network Consistency for Trees

In the rest of the paper we discuss protocols for a restricted class of network topologies — trees. Our aim is to see whether such a restricted class of problems can be solved using the more relaxed, uniform, distributed model, and whether it can result in a more efficient protocol.

It is well known that the sequential network consistency problem on trees is tractable, and can be achieved in linear time [Mackworth and Freuder, 1985]. A special algorithm for this task is composed of an arc-consistency phase (that can be efficiently implemented on trees), followed by value assignment in an order created by some **rooted tree**. It has been shown that an arc-consistent tree enables backtrack-free value assignment with no deadends [Freuder, 1982]. Applying the general NC protocol together with the arc-consistency protocol to a tree will already result in an improved performance: when arc-consistency is established, one forward phase of the value-assignment protocol will assign values to all the nodes in linear time since no deadends will be encountered (see also [Dechter and Dechter, 1988]). Therefore, the almost-uniform NC protocol if applied to trees is guaranteed to converge in a polynomial number of steps.

Since, the DFS subprotocol of our general algorithm was the source for its non-uniformity, we reexamine the possibility that for trees, a rooted directed tree can be imposed via a uniform protocol. We already showed that when using a distributed demon, a uniform, network-consistency protocol for trees is not feasible. Therefore, the only avenue not yet explored is whether under a central demon such a protocol does exist. We next show that this conjecture is indeed correct.

4.1 A uniform tree-consistency protocol

In principle a uniform tree-consistency (TC) protocol can be extracted from the general NC protocol by only replacing the DFS-tree protocol with a **uniform rooted-tree** protocol, since any rooted-tree is also a DFS-tree. Since the arc-consistency protocol, the value-assignment protocol and the privilege-passing protocol are already uniform, the resulting TC protocol will be uniform, as desired. Nevertheless, we will show that for trees, the value-assignment protocol can be simplified as well, while there is no need to have a special privilege-passing mechanism at all.

The TC protocol consists of the following three subprotocols:

1. arc-consistency
2. tree-directing
3. tree-value-assignment

When the arcs are consistent and the tree has been directed, value assignment is eventually guaranteed by having each node follow the rule (of the **tree-value-assignment** protocol): "CHOOSE A VALUE CONSISTENT WITH YOUR PARENT'S ASSIGNMENT". Such a value must exist, since otherwise the value assigned by the parent would have been removed by the arc-consistency procedure. Since, as we will show, the tree-directing protocol is self-stabilizing, and since the arc-consistency protocol is self-stabilizing as well, the value-assignment protocol eventually converges to a consistent solution.

In order to **direct** the tree, we must break the symmetry reflected by the identical codes and the lack of identifiers, by exploiting the topology of the tree. For this task we use a distributed protocol for finding the **centers** of a tree [Korach *et al.*, 1984]. A center of a tree is a node whose maximal distance from the leaves is minimal. Consider a sequential algorithm that works in phases, so that in every phase the leaves of the previous phase are removed from the tree. In the last phase the tree has either one or two connected nodes left. These nodes are the centers of the tree.

Our protocol distributedly simulates the above algorithm. If only one center exists, it plays the role of a root and all the arcs are directed towards it. When two centers exist, the direction of the link that connects them remains ambiguous and both of them can be viewed as a root since all other links are directed towards them as before. In this case, each one of the two centers considers the other one to be its parent, and the tree-directing protocol, results in a **pseudo-rooted-tree** where the centers (one or two nodes) play the root role.

We claim that this ambiguity will not hurt the tree-value-assignment protocol at all. Note that the "first" center that applies the assignment protocol, assigns itself a value consistent with the other. When the other one is scheduled, it is supposed to assign itself a value that is consistent with the first one. However, its current assignment is already consistent (since the first one has taken care of that already) and thus it remains unchanged. All other nodes assign values that are consistent with their parents as before. The central demon policy assures that only one (neighboring) center will be scheduled each time.

This approach yields a relatively simple uniform tree-directing protocol that simulates the above description. Assume the number of nodes in the network[5] is n. Every node i has the following fields:

$N_i[0..\lfloor n/2 \rfloor]$ – a vector that counts the number of i's neighbors in each phase of the sequential algorithm. $N_i[j]$ records the number of neighbors of i in phase j. If $N_i[j] = 1$ it means that i becomes a leaf in the j-th phase (although it may be initialized incorrectly). $N_i[0]$ is repeatedly initialized to the number of i's neighbors in the network (so that $N_i[0] = 1$ means that i is a leaf in the original tree).

$parent_i$ – a variable assigned the value j if node j becomes the parent of i (the enumeration is local to i). When the network stabilizes, namely when all

[5] We can overcome the necessity of knowing the size of the network by using dynamic memory allocation. However, for the sake of the simplicity of the code we assume the knowledge of n.

the N-vectors converge, every node has one neighbor only that is eligible to be its parent, except a single center which has none, and no two nodes are parents of each other except, perhaps, the two centers.

The protocol works by having each node scan its neighbors' N-vectors and compute its own accordingly. Since the j-th entry of vector N_i represents the number of i's neighbors in the j-th phase, its value is recursively computed by decreasing the number of neighbors that became leaves from the entire number of neighbors in the previous phase. Each node chooses as its parent the neighbor that was not removed from the tree earlier than itself. Figure 4 presents a pseudo-code for the protocol. Recall that the code is repeated forever, although from some point on, the tree does not change.

A proper convergence of the N-vectors is guaranteed by the fact that $N_i[j]$ depends only on $N_i[j-1]$ and $\{N_k[j-1] | k \in neighbors(i)\}$, which are properly updated earlier. The base of this iterative convergence is applied by assigning to $N_i[0]$ the actual number of neighbors of i in the network.

The complexity of the tree protocol is clearly linear in the network's size since all its subprotocols are, and hence it equals the sequential time complexity. However, the parallel time can be further linearly bounded by the **diameter** of the tree where the diameter is the longest path between any two leaves of the tree.

5 Conclusions

The results presented in this paper establish theoretical bounds on the capabilities of connectionist architectures and other distributed approaches to constraint satisfaction problems.

The paper focuses on the feasibility of solving the network consistency problem using self-stabilizing distributed protocol, namely, guaranteeing a convergence to a consistent solution, if such exists, from any initial configuration. Such property is essential for dynamic environments, where unexpected changes could occur in some of the constraints.

We proved that a uniform protocol, one in which all nodes are identical, cannot solve the network consistency problem even if only one node is activated at a time. Consequently, although such protocols have obvious adventages and are closer in spirit to neural networks architectures, they cannot guarantee convergence to a solution. On the other hand, distinguishing one node from the others is sufficient to guarantee such a convergence even when sets of nodes are activated simultaneously. A protocol for solving the problem under such conditions is presented.

We then demonstrated that, when the network is restricted to trees, a uniform, self-stabilizing protocol for solving the problem does exist, but only under asynchronous control (one neighboring node is activated at a time).

It is still an open question whether a uniform protocol is feasible under some specific ordering of the asynchronous activation.

Regarding time complexity, we have shown that in the worse-case the distributed and the sequential protocols have the same complexity bound; exponential in the depth of the DFS tree. On the average, however, a linear speed up is feasible for bounded depth networks.

References

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

[Burns *et al.*, 1987] J. Burns, M. Gouda, and C. L. Wu. A self-stabilizing token system. In *Proceedings of the 20th Annual Intl. Conf. on System Sciences*, pages 218–223, Hawaii, 1987.

[Collin and Dechter, 1990] Z. Collin and R. Dechter. A distributed solution to the network consistency problem. In *Proceedings of the 5-th Intl. Symp. on Methodologies for Intelligent Systems.*, pages 242–251, Tennessee, USA, 1990.

[Collin and Dolev, 1991] Z. Collin and S. Dolev. A self stabilizing protocol for dfs spanning tree generation. in preparation, 1991.

[Dahl, 1987] E. D. Dahl. Neural networks algorithms for an np-complete problem: map and graph coloring. In *Proceedings of the IEEE first Internat. Conf. on Neural Networks*, pages 113–120, San Diego, 1987.

[Dechter and Dechter, 1988] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings AAAI-88*, St. Paul, Minnesota, August 1988.

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

[Dijkstra, 1974] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[Dolev *et al.*, 1990] S. Dolev, A. Israeli, and S. Moran. Self stabilization of dynamic systems assuming only read/write atomicity. In *Proceedings of PODC-90*, pages 103–118, Quebec City, August 1990.

[Even, 1979] S. Even. *Graph Algorithms.* Computer Science Press, Maryland, USA, 1979.

```
procedure tree-directing
Begin
1.  N_i[0] ← |neighbors(i)|
2.  j ← 0                        { j is a local counter }
3.  if N_i[0] ≠ 1 then           { when i is not a leaf }
4.      while N_i[j] > 1 do      { i is not yet a leaf at the j-th stage }
5.          j ← j + 1
            { the leaves of the (j − 1)-th stage are removed in the j-th stage }
6.          N_i[j] ← N_i[j − 1] − |{k | k ∈ neighbors(i) ∧ N_k[j − 1] = 1}|
        od
7.  if ∃k ∈ neighbors(i) s.t. N_k[j] ≥ 1 then { no more than one such k exists }
8.      parent_i ← k
9.  else                         { i is the only center }
10.     parent_i ← NONE
End.
```

Figure 4: Uniform tree-directing procedure for node i

[Freuder and Quinn, 1987] E. C. Freuder and M.J. Quinn. The use of lineal spanning trees to represent constraint satisfaction problems. Technical Report 87-41, University of New Hampshire, Durham, New Hampshire, 1987.

[Freuder, 1982] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, January 1982.

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

[Mackworth and Freuder, 1985] A. K. Mackworth and E.C. Freuder. The complexity of some polinomial network consistency algorithms for constraint satisfaction problem. *Artificial intelligence*, 25:65–74, 1985.

[Minton *et al.*, 1990] S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of AAAI-90*, Boston, 1990.