Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A DISTRIBUTED SOLUTION TO THE NETWORK CONSISTENCY PROBLEM

Zeev Collin
Rina Dechter

July 1991
CSD-910039

Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A DISTRIBUTED SOLUTION TO THE NETWORK CONSISTENCY PROBLEM

Zeev Collin
Rina Dechter

July 1991
CSD-910039

Technical Report
R-148
March 1990

A DISTRIBUTED SOLUTION TO THE NETWORK CONSISTENCY PROBLEM

Zeev Collin & Rina Dechter (1)

Computer Science Department
Technion -- Israel Institute of Technology
Haifa, Israel, 32000

ABSTRACT

In this paper we present a distributed algorithm for solving binary constraint satisfaction problem. Unlike approaches based on connectionist type architectures, our protocol is guaranteed to be self stabilized, namely it converges to a consistent solution, if such exists, from any initial configuration. An important quality of this protocol is that it is self-stabilizing, a property That renders our method suitable for dynamic or error prone environments.

Topic: Automated reasoning
Subtopic: Distributed AI

(1) This research was supported in part by NSF grant #IRI-8821444 and by Air Force grant #AFSOR 880177 while the second author was visiting the cognitive systems lab at UCLA.

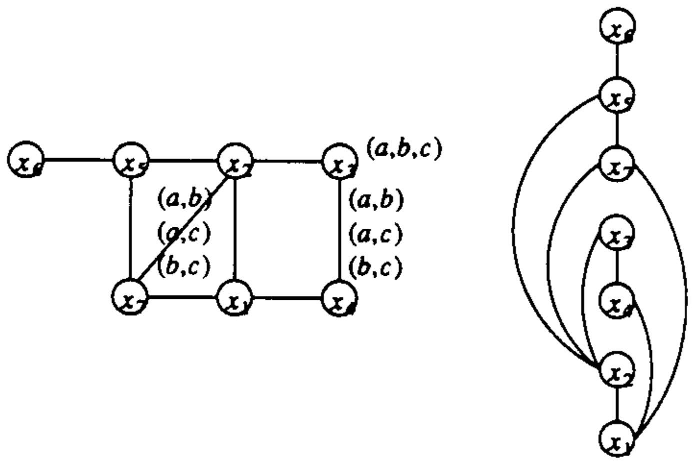
1. Introduction

Consider the distributed version of the graph coloring problem, where each processor 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 (i.e. graph coloring) is known to be NP-complete, belongs to a large class of combinatorial problems known as **Constraint Satisfaction Problems (CSPs)** which present interesting challenges to distributed computation, particularly to connectionist architectures. We call the distributed version of the problem the **network consistency problem**. Since the problem is inherently intractable, the interesting questions for distributed models are those of feasibility rather than efficiency. The main question we wish to answer in this paper is: What types of distributed models would admit a self-stabilizing algorithm, namely, one that converges to a solution, if such exists, from any initial state of the network.

The motivation for addressing this question stems from attempting to solve constraint satisfaction problems within a "connectionist" type architecture. Constraints are useful in programming languages, simulation packages and general knowledge representation systems because they permit the user to state declaratively those relations that are to be maintained, rather than writing the procedures for maintaining the relations. The prospects of solving such problems by connectionist networks promises the combined advantages of massive parallelism and simplicity of design. Indeed, many interesting problems attacked by neural networks researchers involve constraint satisfaction [1, 16, 3], and, in fact, any discrete state connectionist network can be viewed as a type of constraint networks, with each stable pattern of states representing a consistent solution. However, whereas current connectionist approaches to CSPs lack theoretical guarantees of convergence (to a solution satisfying all constraints), the distributed model which we use here is the closest in spirit to the connectionist paradigm for which such guarantees have been established. Other related attempts for solving CSPs distributedly were either restricted to singly connected networks [4, 15], or, were based on general constraint propagation, thus not guarantee convergence to a consistent solution [12].

Our distributed model consists of a network of interconnected processors in which an activated processor reads the states of all its neighbors, decides whether to change its state and then moves to a new state. The activation of a processor is determined by its current state and the states of its neighbors. We also assume that all processors but one are identical (this assumption is not part of classical connectionist models, but is necessary for our protocol). Under these architectural restrictions the network consistency problem is formulated as follows: Each processor has a pre-determined set of values and a compatibility relation indicating which of its neighbors' values are compatible with each of its own. Each processor must select a value that is compatible with the values selected by its neighbors. We shall first review the sequential variant of this problem and then develop a distributed self-stabilizing solution.

A **Network of binary constraints** involves a set of n variables $X_1,...,X_n$, 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 (CSP)** associated with these networks 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. Figure 1a presents a constraint network where each node represents a variable having values $\{a, b, c\}$ and each link is associated with a strict lexicographic order (where $X_i < X_j$ iff $i < j$). (The domains and the constraints explicitly indicated on some of the links.)



(a) (b)

Figure 1: An example of a binary CN

General constraint satisfaction problems may involve constraints of any arity, but since network communication is only pairwise we focus on this subclass of problems.

The rest of this paper is organized as follows: Section 2 provides the sequential algorithm for solving a CSP that is the basis for our distributed protocol. Section 3 introduces the distributed model and the requirements for self-stabilization, section 4 provides the self-stabilizing distributed protocol for solving the network consistency problem, while section 5 presents some worst-case analysis of the performance of our protocol.

2. Sequential Algorithms for Constraint Satisfaction

2.1 Backtracking

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. A backtracking algorithm for finding one solution is given below. It is defined by two recursive procedures, **Forward** and **Backword**. The first extends a current partial assignment if possible, and the second handles deadend situations. The procedures maintain lists of candidate values (C_i) for each variable X_i. The procedure **compute-candidates**$(x_1, \ldots, x_i, X_{i+1})$ selects all values in the domain of X_{i+1} which are consistent with the previous assignments.

```
Forward ( $x_1, \ldots, x_i$)
Begin
  1. if $i = n$ exit with the current assignment.
  2. $C_{i+1} \leftarrow$ Compute-candidates($x_1, \ldots, x_i, X_i$)
  3. if $C_{i+1}$ is not empty then
  4.     $x_{i+1} \leftarrow$ first element in $C_{i+1}$, and
  5.     remove $x_{i+1}$ from $C_{i+1}$, and
  6.     Forward($x_1, \ldots, x_i, x_{i+1}$)
  7. else
  8.     Backword($x_1, \ldots, x_i$)
End.
```

```
Backword( $x_1, \ldots, x_i$)
Begin
  1. if $i$=0, exit   { No solution exists }
  2. if $C_i$ is not empty then
  3.     $x_i \leftarrow$ first in $C_i$, and
  4.     remove $x_i$ from $C_i$, and
  5.     Forward($x_1, \ldots, x_i$)
  6. else
  7.     Backword($x_1, \ldots, x_{i-1}$)
End
```

2.2 Backjumping

Many enhancement schemes were proposed to overcome the inefficiency of "naive" backtracking [14, 13, 9, 11]. One particularly useful technique, called **backjumping** [5] consults the topology of the constraint graph to guide its "backword" 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.

Consider again the problem in figure 1a. If variables are instantiated in the order $X_1, X_2, X_4, X_3, X_7, X_5, X_6$ (see figure 1b), then when a dead-end occurs at X_7 the algorithm will jump back to variable X_2; since X_7 is not connected to either X_3 or X_4 they cannot be responsible for the deadend. If the variable to which the algorithm retreats has no more values, it backs-up further, to the most recent variable connected either to the original or to the new deadend variables, and so on.

2.3 Depth first search with backjumping

Whereas the implementation of backjumping in an arbitrary variable ordering requires a careful maintenance of each variable's parents set [5], some orderings facilitate a specially simple implementation. If we use a depth-first search (DFS) on the constraint graph (to generate a DFS tree) and then conduct backjumping in an inorder traversal of the DFS tree [8], finding the jump-back destination amounts to following a very simple rule: if a deadend occurred at variable X, go back to the parent of X in the DFS tree. Consider once again our example of figure 1. A DFS tree of this graph is given in figure 2, and an inorder traversal of this tree is $(X_1, X_2, X_3, X_4, X_5, X_6, X_7)$. Hence, if a deadend occur at node X_5 the algorithm retreats to it parent X_2.

The nice property of a DFS tree which makes it particularly suitable for 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). Namely, 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. This translates to a useful problem-decomposition strategy: if all ancestors of variable X are instantiated then the solution of each of its subtrees are completely independent and can be performed in parallel. The idea of using a DFS tree traversal for backtracking and its potential for parallel implementation is not new. It was introduced by Freuder and Quinn [11]. However the parallel algorithm they present

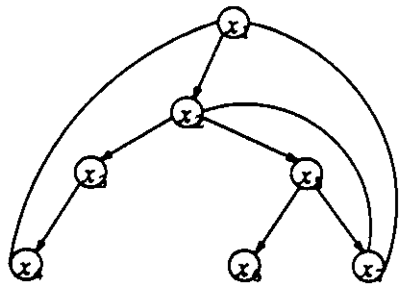


Figure 2 : A DFS tree

assumes a message passing model, is not self-stabilized and it is targeted for implementation on a multiprocessor [10]. We believe that the use of DFS-based backjumping for a connectionist type architecture and its self-stabilizing property is novel to this work.

3. Basic Definitions for Distributed Computations

3.1 The model

Our general communication model is similar to the one defined in [7]. A distributed system consists of n processors, $P_0, P_1, \cdots P_{n-1}$, connected by bidirectional communication links. It can be viewed as a **communication graph** where nodes represent processors and arcs corresponds to communication links. We use the terms **node** and **processor** interchangeably. Some (or all) edges of the graph may be directed, meaning that the two linked processors (called a **child** and a **parent** respectively), are aware of this direction (this, though, is unrelated to communication flow). Neighbors communicate using shared communication registers, called **state registers**, and $state_i$ is the register written only by node i, but may be read by several processors (all i's neighbors). The state register may have a few fields, but it is regarded as one unit. This method of communication is known as **shared memory multi-reader single-writer** communication. The processors are anonymous i.e. have no identities. (We use the term node i or processor P_i as a writing convenience only). A configuration C of the system is the state vector of all processors.

Processor's activity is managed by **distributed demon** defined in. [2,7] In each activation the distributed demon activates a subset of system's processors, all of which execute a single atomic step simultaneously. That is, they read the states of their neighbors, decide whether to change their state and move to their new state. An **execution** of the system is an infinite sequence of configurations $E = c_1 , c_2 \cdots$ such that for every i c_{i+1} is a configuration reached from configuration c_i by a single atomic step executed by any subset of processors simultaneously. We say that an execution is **fair** if any node participates in it infinitely often.

A processor can be modeled as a state-machine, having a predetermined set of states. The state transition of a processor is controlled by a **decision function**, f_i, which is a function of its input, its state and the states of its neighbors. The collection of all decision functions is called **protocol**.

A **uniform protocol** is a protocol in which all nodes have identical decision functions. Following Dijkstra's observation [6] regarding mutual exclusion task, we can show that solving the network consistency problem using a uniform protocol is impossible. We, therefore, adopt the model of "**almost uniform protocol**" namely, all processors but one are identical and have identical decision functions. We denote the special processor as P_0.

3.2. Self stabilization

Our requirements from self stabilizing protocol are similar to those in [6]. A self stabilizing protocol should demonstrate legal behavior of the system, namely when starting from any initial configuration (and with any input values) and given enough time, the system should eventually converge to a legal set of configurations for any fair execution. The legality of a configuration depends on the aim of the protocol. Formally, let L be the set of legal configurations. A protocol for the system is **self stabilizing** with respect to L if every infinite fair execution, E, eventually satisfies the following two properties:

1. E enters a configuration c_i that belongs to L.

2. For any $j > i$ and $c_i, c_j \in E$, if $c_i \in L$ then $c_j \in L$ (i.e. once entering L it never leaves it).

In our case a legal configuration is a consistent assignment of values to all the nodes in the network if one exists and, if not, any configuration is legal.

4. A Distributed Consistency-Generation Protocol

This section presents a self stabilizing protocol for solving the network consistency problem. It consists of two subprotocols; one simulates the sequential backjumping on DFS (section 4.3), and the other facilitates the desired activation mechanism (section 4.2).

4.1 Neighborhoods and states

We assume the existence of a self-stabilizing algorithm for generating a DFS tree, as a result of which each internal processor, P_i, has one adjacent processor, *parent* (P_i), designated as its parent, and a set of children nodes denoted *children* (P_i). The link leading from *parent* (P_i) to P_i is called **inlink** while the links connecting P_i to its children are called **outlinks.** The rest of P_i's neighbors are divided into two subsets: *ancestors* (P_i), consisting of all neighbors that reside along the path from the root to P_i and the set of its successors. For our algorithm a processor can disregard its successors (which are not its children) and observe only the three subsets of neighbors as indicated by figure 3a (for internal nodes) and figure 3c (for leaves). **The root**, having no parent is played by the special processor P_0 (figure 2b).

We assume that processor P_i (representing variable X_i) has a list of possible values, denoted as ***Domain***$_i$, one of which will be assigned to its state (i.e. to its *value* field in the state register), and a pairwise relation R_{ij} with each neighbor P_j.

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

1. A **value** field to which it assigns either one of its domain values or the symbol "*" (to denote a deadend).

2. A **mode** field indicating the processor's "belief" regarding the status of the network. A processor changes the mode from "*on*" to "*off* and vice-versa in accordance with the pol-

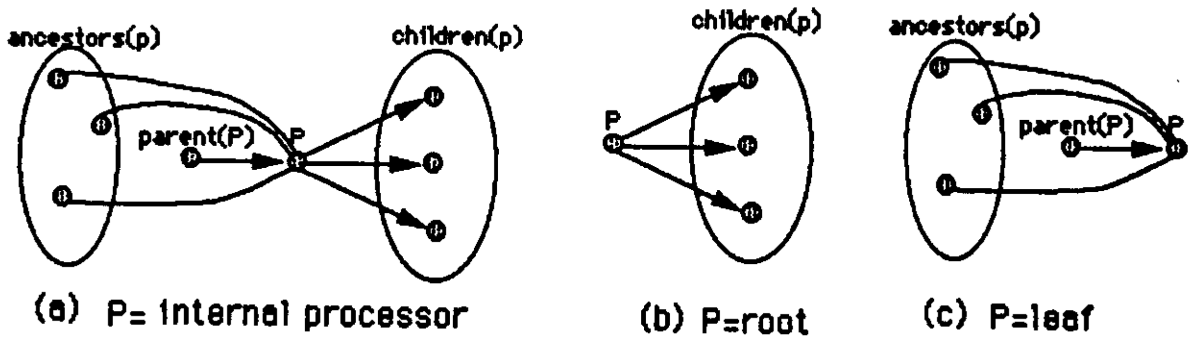


Figure 3: A processor's neighborhood set.

icy described in section 4.3. The modes of all processors also give an indication whether all processors have reached a consistent state (all being in "*off*" mode).

3. Two boolean fields called **parent_tag** and **children_tag**, which are used to control the activity of the processors (section 4.2.)

Additionally, each processor has an ordered domain list which is controlled by a local **domain pointer** (to be explained later), and a local **direction** field indicating whether the algorithm is in its forward or backward phase (to be discussed in section 4.3).

4.2 Activation Mechanism

The control protocol is handled by a self-stabilizing activation mechanism. According to this protocol a processor can get a privilege to act, granted to him either by its parent or by its children. A processor can change its state only if it is privileged.

Our control mechanism is based on a mutual exclusion protocol for two processors called **balance/unbalance**. The balance/unbalance mechanism is a simplified version of Dijkstra's protocol for directed ring [6, 7], and is summarized next.

Consider a system of two processors, P_0 and P_1, each being in one of two states "0" or "1". P_0 changes its state if it equals P_1's state, while P_1 changes its state if it differs from P_0's state. We call a processor that is allowed to change its state **privileged**. In other words, P_0 becomes privileged when the link between the processors is **balanced** (i.e. the states on both its endpoints are identical). It then unbalances the link and P_1 becomes privileged, (the link is unbalanced). P_1 in its turn balances the link. It is easy to see that in every possible configuration there is one and only one privileged processor. Hence this protocol is self stabilizing for the mutual exclusion task and the privilege is passed infinitely often between the two processors. We next extend the balance/unbalance protocol to our needs, assuring, for instance that a node and its ancestor will not be allowed to change their values simultaneously.

Given a DFS spanning tree, every state register contains two fields: **parent_tag**, referring to the inlink and **children_tag**, referring to all the outlinks. A node, *i*, becomes privileged if its inlink is unbalanced a d all its outlinks are balanced, namely if the following two conditions are satisfied:

1. for $j = parent(i) : parent\_tag_i \neq children\_tag_j$ (the inlink is unbalanced)

2. $\forall k \in children(i) : children\_tag_i = parent\_tag_k$(1) (the outlinks are balanced)

A node applies its decision function (described in section 4.3), only when it is privileged (otherwise it leaves its state unchanged), and upon its execution, it passes the privilege accordingly. The privilege can be passed backwards to the parent by balancing the incoming link, i.e. by changing the *parent_tag* value, or forward to the children by unbalancing the outgoing links, i.e. by changing the *children_tag* value.

We define the set of legal configurations to be those in which exactly one processor is privileged on every path from the root to a leaf. Figure 4 shows such a configuration. Note how the privilege splits on its way "down". We claim that the control mechanism is self stabilizing, with respect to these legal configurations. (The proof is presented in the extended paper.)

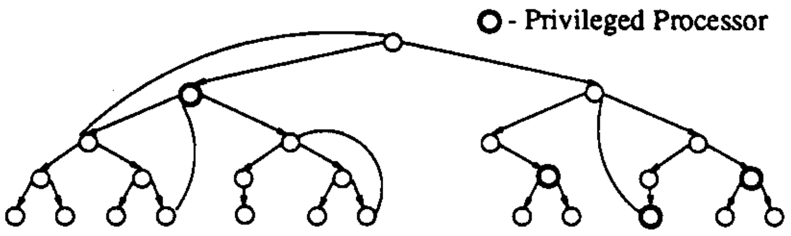


Figure 4: An example for a legal privileges configuration

Once got privileged, a processor cannot tell where the privilege came from (i.e. from its parent or from its children). Thus, a processor uses its **direction** field to indicate the source of its privilege. Since in a stable period exactly one processor is privileged on every path from the root to a leaf, the privileges travel along their paths backwards and forwards. The direction field of each processor indicates the direction that the privilege was recently passed by this processor. When passing the privilege to its parent, the processor assigns its direction field the "*backward*" value, while when passing the privilege to its children it assigns the "*forward*" value. Thus, upon receiving the privilege again, it is able to recognize the direction it came from: if *direction* = "*forward*", the privilege was recently passed towards the leaves and therefore it can come only from its children; if *direction* = "*backward*", the privilege was recently passed towards the root and therefore it can come only from its parent. Following are the procedures for privilege passing by P_i.

```
procedure pass-privilege-to-parent
Begin
  1. $parent\_tag_i \leftarrow children\_tag_{parent(i)}$        { balance inlink }
  2. $direction_i \leftarrow$ "backward"
End.

procedure pass-privilege-to-children
Begin
  1. for $k \in children(i)$ $children\_tag_i \leftarrow \neg parent\_tag_k$  { unbalance outlinks }
  2. $direction_i \leftarrow$ "forward"
End.
```

(1) Note that this is well defined since we can prove that all siblings have the same parent-tag.

4.3 Protocol description

The protocol has a forward and a backward phases, corresponding to the two phases of the sequential algorithm. During the forward phase processors in different subtrees assign consistent values (in parallel) or verify the consistency of their assigned values. When a processor realizes a deadend it assigns its value field a "*" and initiates a backward phase. When the network is consistent (all processors are in an "*off*" mode) the forward and backward phases continue, whereby 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 offending processor moves to an "*on*" mode and continues from there.

A processor can be in one of three situations:

1. **Processor P_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 ancestors might have occurred. It, therefore, resets the domain pointer to point to the beginning of the domain list, finds the first value in its domain that is consistent with all its ancestors, put itself in an "*on*" mode and passes privilege to its children. If no consistent value exists, it assigns itself the "*" value (a deadend) and passes privilege to its parent (initiating backward phase).

2. **Processor P_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 ancestors. If it is consistent it stays in an "*off*" mode and moves privilege to its children. If not, it assigns itself a new value (after resetting the domain pointer to start), moves to an "*on*" mode and passes privilege to children.

3. **Processor P_i is activated by its children** (backward phase). If one of the children has a "*" value, the processor selects the next consistent value (after the current pointer) from its domain, resets its domain pointer to point to the assigned value and passes the privilege to children. If no consistent value is available, it assigns itself a "*" and passes privilege to its parent (1). If all children has a consistent value, P_i passes privilege to its parent.

In the following we present the algorithms performed by processor P_i, $i \neq 0$, (see figure 5) and the root (figure 6).

The procedure **compute-next-consistent-value** (in figure 5) tests each value which is located after the domain pointer, for consistency. Namely the value is checked against each *ancestor*(P_i)'s values and the first consistent value is returned. The pointer's location is readjusted accordingly (i.e., to the found value). If no legal value was found the value returned is "*" and the pointer is reset to the beginning of the domain. The procedure **verify-consistency** checks the consistency of current value with ancestors and returns a truth-value (i.e. "*true*" if it is consistent and "*false*" otherwise). For details see the extended paper.

The algorithm performed by the root, P_0, (see figure 6) is slightly different and in a way simpler. The root does not check consistency. All it does is assigning a new value at the end of each backward phase, when needed, then initiating a new forward phase.

(1) 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 privilege. This is done to guarantee that all subtrees have a consistent view regarding their ancestor's values. A more relaxed privilege passing mechanism is presented in the extended paper.

```
procedure update-state (for any processor except the root)
Begin
 1. read $parent(P_i)$ and $children(P_i)$
 2. if $direction$ = "$backward$" then    { privilege came from parent }
 3.   if parent's mode is "$on$" then
 4.     $mode \leftarrow$ "$on$"
 5.     $value \leftarrow$ compute-next-consistent-value
 6.     if $value$ = "*" then
 7.       pass-privilege-to-parent
 8.     else                    { there is a legal consistent value }
 9.       pass-privilege-to-children
10.   else                      { parent's mode is "$off$" }
11.     if verify-consistency = "$true$" then
12.       $mode \leftarrow$ "$off$"; pass-privilege-to-children
13.     else                    { verify-consistency = "$false$" }
14.       $mode \leftarrow$ "$on$"
15.       $value \leftarrow$ compute-next-consistent-value
16.       if $value$ = "*" then
17.         pass-privilege-to-parent
18.       else                  { there is a legal consistent value }
19.         pass-privilege-to-children
20. else                        { $direction$ = "$forward$" i.e. privilege came from children }
21.   if $\exists k \in children(P_i)\ value_k$ = "*" then
22.     $mode \leftarrow$ "$on$"
23.     $value \leftarrow$ compute-next-consistent-value
24.     if $value$ = "*" then   { no consistent value was found }
25.       pass-privilege-to-parent
26.     else                    { there is a legal consistent value }
27.       pass-privilege-to-children
28.   else                      { all children are consistent }
29.     pass-privilege-to-parent
End.
```

Figure 5: The decision function of P_i , $i \neq 0$.

```
procedure root-update-state
Begin
 1. read $children(P_0)$
 2. if $\exists k \in children(P_0)\ value_k$ = "*" then
 3.   $mode \leftarrow$ "$on$"
 4.   $value \leftarrow$ next-value
 5. else                        { all children are consistent }
 6.   $mode \leftarrow$ "$off$"
 7. pass-privilege-to-children
End.
```

Figure 6: The decision function of P_0.

The procedure **next-value** returns the value pointed by the domain pointer and readjusts the pointer's location. If the end of the domain list is reached, the pointer is reset to the beginning.

In the extended paper we will prove the correctness of our protocol. The self-stabilization property of our activation mechanism assures an adequate control activation mechanism for distributedly implementing backtracking. Having this property we can prove the self-stabilization of the "consistency-generation" protocol, namely that eventually the network converges to a legal solution, if one exists, and if not it keeps checking all the possibilities over and over again.

5. Complexity Analysis

The precise time complexity of the protocol has yet to be formally analyzed. However, a crude estimate can be given of the maximal number of state changes from the time the activation mechanism has stabilized until final convergence. The worst-case number of states changes depends on the worst-case time of the sequential backjump algorithm. We will present a bound on the search space explored by the sequential algorithm and show that the same bound applies to the number of state changes of our protocol. Our bound improves the one presented in [11].

Let $T_{m,}$ stands for the search space generated by DFS-backjumping when the depth of the DFS tree is m or less. Let b be the maximal brunching degree in the tree and let k bounds the domain sizes. Since any assignment of a value to the root node generates b subtrees of depth $m-1$ or less that can be solved independently, T_m obeys the following recurrence:

$$T_m = k \cdot b \cdot T_{m-1} \tag{1}$$

with $T_0 = k$. Solving this recurrence yields

$$T_m = b^m k^{m+1} \tag{2}$$

(Note that when the tree is balanced we get that $T_m = nk^{m+1}$.)

It is easy to show that the **number of state changes** of our protocol satisfies exactly the same recurrence. The reason is as follows. Any sequential DFS-backjumping produces a search space smaller or equal to the number of state changes of the distributed protocol, however there is exactly one run of the sequential algorithm whose search space is identical to the number of state changes in the protocol, thus the two worst-case are identical.

6. Conclusions

We have shown that the network consistency problem can be solved distributedly within a connectionist type architecture. The protocol we presented is self-stabilizing, namely its convergence to a consistent solution is guaranteed. The self-stabilizing property renders our model suitable for solving CSPs in dynamic environments. For instance, unexpected changes of some of the domains or some of the constraints will trigger a transient perturbation in the network but it will eventually converge to a new solution. Similarly, the protocol can adjust to changes in network's topology. Such changes demand the generation of a new DFS spanning tree. A self stabilized DFS protocol will be presented in the extended paper.

Although we are attacking an NP-complete problem, we have shown that our protocol's complexity is polynomial in networks of bounded DFS depth. Thus the DFS depth can be regarded as a crucial parameter that determines the rate of convergence in our model. It will be interesting to explore whether the speed of convergence in connectionist networks is related to a similar network parameter.

References

[1] Ballard, D. H., P. C., Gardner, and M. A. Srinivas, ''Graph problems and connectionist architectures.,'' University of Rochester, Rochester, NY, Tech. Rep. TR-167, 1986.

[2] Burns, J., M. Gouda, and C. L. Wu, ''A Self stabilizing token system,'' in *20th Annual Intl. Conf. on System Sciences*, Hawaii: 1987, pp. 218-223.

[3] Dahl, E. D., ''Neural networks algorithms for an NP-complete problem: map and graph coloring,'' in *IEEE first intl. Conf. on Neural Networks*, San Diego: 1987, pp. 113-120.

[4] Dechter, R. and A. Dechter, ''Belief maintenance in dynamic constraint networks,'' in *Proceedings AAAI-88*, St. Paul, Minesota: 1988.

[5] Dechter, R., ''Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition,'' *Artificial Intelligence*, Vol. 41, No. 3, 1990.

[6] Dijkstra, E. W., ''Self stabilizing systems in spite of distributed control,'' *Communication of the ACM*, Vol. 17, No. 11, 1974.

[7] Dolev, S., A. Israeli, and S. Moran, ''Self stabilization of dynamic systems assuming only read/write atomicy,'' Technion, Haifa, Israel, 1990.

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

[9] Freuder, E.C., ''A sufficient condition of backtrack-free search.,'' *Journal of the ACM*, Vol. 29, No. 1, 1982, pp. 24-32.

[10] Freuder, E. C. and M. J. Quinn, ''Parallelism in algorithms that take adventage of stable sets of variables to solve constraint satisfaction problems.,'' University of New Hampshire, Durham, New Hampshire., Tech. Rep. 85-21 , 1985.

[11] Freuder, E. C. and M. J. Quinn, ''The use of lineal spanning trees to represent constraint satsfaction problems,'' University of New Hampshire, Durham, New Hampshire., Tech. Rep. 87-41 , 1987.

[12] Gusgen, H.W. and J. Hertzberg, ''Some fundamental properties of local constraint propagation,'' *Artificial Intelligence Journal*, Vol. 36, No. 2, 1988, pp. 237-247.

[13] Haralick, R, M. and G.L. Elliot, ''Increasing tree search efficiency for constraint satisfaction problems,'' *AI Journal*, Vol. 14, 1980, pp. 263-313.

[14] Mackworth, A.K., ''Consistency in networks of relations,'' *Artificial intelligence*, Vol. 8, No. 1, 1977, pp. 99-118.

[15] Rossi, F and U. Montanari, ''Exact solution of networks of constraints using perfact relaxation,'' in *First Intl. Conf. on Principles of Knowledge Representation and Reasoning*, Toronto, Canada: 1989.

[16] Tagliarini, G.A. and E. W. Page, ''Solving constraint satisfaction problems with neural networks,'' in *IEEE first intl. Conf. on Neural Networks*, San Diego, CA: 1987, pp. 741-747.