Chicago Journal of Theoretical Computer Science

The MIT Press

Volume 1999, Article 10

Self-stabilizing Distributed Constraint Satisfaction

ISSN 1073–0486. MIT Press Journals, Five Cambridge Center, Cambridge, MA 02142-1493 USA; (617)253-2889; *journals-orders@mit.edu, journals-info @mit.edu*. Published one article at a time in LaTeX source form on the Internet. Pagination varies from copy to copy. For more information and other articles see

- *http://mitpress.mit.edu/CJTCS/*
- *http://www.cs.uchicago.edu/publications/cjtcs/*
- *ftp://mitpress.mit.edu/pub/CJTCS*
- *ftp://cs.uchicago.edu/pub/publications/cjtcs*

The *Chicago Journal of Theoretical Computer Science* is abstracted or indexed in *Research Alert®, SciSearch®, Current Contents®/Engineering Computing & Technology*, and *CompuMath Citation Index®*.

The *Chicago Journal of Theoretical Computer Science* is a peer-reviewed scholarly journal in theoretical computer science. The journal is committed to providing a forum for significant results on theoretical aspects of all topics in computer science.

Self-stabilizing Distributed Constraint Satisfaction

Zeev Collin
Computer Science Department
The Technion
Haifa, Israel
zeev@cs.technion.ac.il

Rina Dechter
Department of Information and Computer Science
University of California
Irvine, California, USA
dechter@ics.uci.edu

Shmuel Katz
Computer Science Department
The Technion
Haifa, Israel
katz@cs.technion.ac.il

31 December 1999

Abstract

Distributed architectures and solutions are described for classes of constraint satisfaction problems, called *network consistency problems*. An inherent assumption of these architectures is that the communication network mimics the structure of the constraint problem. The solutions are required to be self-stabilizing and to treat arbitrary networks, which makes them suitable for dynamic or error-prone environments. We first show that even for relatively simple constraint networks, such as rings, there is no self-stabilizing solution that guarantees convergence from every initial state of the system using a completely uniform, asynchronous model (where all processors are identical). An *almost uniform*, asynchronous, network consistency protocol with one specially designated node is shown and proven correct. We also show that some restricted topologies such as trees can accommodate the uniform, asynchronous model when neighboring nodes cannot take simultaneous steps.

1 Introduction

Consider the distributed version of the graph coloring problem, where each node must select a color (from a given set) that is different from any color selected by its neighbors. This NP-complete problem belongs to the class of *constraint satisfaction problems* (csp's), which present interesting challenges to distributed computation.[1]

We call the distributed versions of this class of problems *network consistency problems*. Since constraint satisfaction is inherently intractable for the general case, the interesting questions for distributed models are those of feasibility rather than efficiency. We wish to answer the following main question: What types of distributed models admit a self-stabilizing algorithm, namely, one that converges to a solution, if such exists, from any initial state of the network? For models that do admit solutions, we present and prove the correctness of appropriate algorithms that converge to a solution for any specific problem.

The motivation for addressing this question stems from attempting to solve constraint satisfaction problems in environments that are inherently distributed. For instance, an important family of constraint satisfaction problems in telecommunications involves scheduling transmissions in radio networks [RL92]. A radio network is a set of n stations that communicate by broadcasting and receiving signals. Typical examples of radio networks are packet radio networks and satellite networks. Every station has a transmission radius. The *broadcast scheduling problem* involves scheduling broadcast transmissions from different stations in an interference-free way. In the time-division multiplexing version, there is a fixed set of time slots, and each

[1] Obviously, there is no connection to Hoare's CSP (communicating sequential processes) notation [Hoa85].

station must assign itself one time slot such that any two stations that are within the transmission radius of each other are assigned different time slots. The frequency-division multiplexing version is similar, with the fixed set of time slots replaced by a fixed set of frequencies. One can easily see that the problem translates naturally to a graph coloring problem where broadcasting radios are the nodes in the graph and any two nodes are connected if and only if their transmission radii overlap.

Solving the broadcasting scheduling problem autonomously and distributedly by the radio stations themselves is highly desirable because the environment is inherently distributed: In some applications (e.g., in a military setting, when the radios are moving) no central control is feasible or practical. Moreover, a self-stabilizing distributed solution has the important virtue of being fault tolerant.

Another motivating area, distributed artificial intelligence (DAI), has become very popular in recent years [Les90]. The issue addressed is problem solving in a multiagent environment where agents have to cooperate to solve a problem and to carry out its solution in a distributed manner. As in the broadcast scheduling problem, the distributed environment is a physical necessity rather than a programmer's design choice.

One possible architecture considered for solving the network consistency problem is neural networks. Such networks perform distributed computation with uniform units and are normally self-stabilizing. However, current connectionist approaches to constraint problems [BGS86, Dah87] lack theoretical guarantees of convergence to a solution, and the conditions under which such convergence can be guaranteed (if at all) have not been systematically explored. This paper aims to establish such guarantees by studying the feasibility of solving a csp in uniform, self-stabilizing distributed architectures.

Intuitively, a distributed algorithm is *self-stabilizing* [Dij74] if it converges to a solution from any initial state of both the network and the algorithm. Our aim is to determine what types of distributed models admit self-stabilizing algorithms and, whenever possible, to present such an algorithm. Self-stabilization is a desirable property since it yields robustness in the face of dynamically changing environments. This is especially true if the solution treats any network configuration and, thus, within some time after a change to the network, converges to a solution. Some changes can be imposed externally (e.g., adding new components to the problem, changing the values of variables or buffers); others may occur to the system internally, by errors in the implementing hardware. The accepted model we use for self-stabilization [Dij74] treats the part of the computation from after a perturbation to the next perturbation as a normally executing algorithm with abnormal initial values, including control locations. The implicit assumption is that perturbations occur infrequently, so that the system stabilizes and does most of its work in consistent states.

In this paper, we characterize architectures that allow a self-stabilizing distributed solution for classes of constraint satisfaction problems, and we present algorithms when possible. As noted above, the self-stabilization can model dynamically changing constraint networks as well as transient network perturbations, thus increasing the robustness of the solutions. Following some background on constraint networks and self-stabilization (Section 2) we show that *uniform* networks, in which all nodes run identical procedures and any scheduling policy is allowed, cannot admit algorithms that guarantee convergence to a consistent solution from any initial state using an asynchronous model (Section 3). In Section 4, we depart slightly from uniformity by allowing one unit to execute a different algorithm. We call this model *almost uniform* and use it to present an asynchronous, network consistency protocol. It combines a subprotocol to generate a depth-first search (DFS) spanning tree, with a value-assignment subprotocol. The value assignment exploits the potential parallelism of the spanning tree, using a special form of backtracking appropriate for constraint problems. In Section 5, we show that some restricted topologies, such as trees, can accommodate the uniform, asynchronous model. Preliminary versions of some of these results first appeared in [CDK91].

2 Background

2.1 Constraint networks

A csp is defined over a *constraint network* that consists of a finite set of *variables*, each associated with a *domain* of values, and a set of *constraints*. A *solution* is an assignment of a value to each variable from its domain such that all the constraints are satisfied. Typical constraint satisfaction problems are to determine whether a solution exists, to find one or all solutions, and to find an optimal solution relative to a given cost function. An example of a constraint satisfaction problem is the k-colorability problem mentioned in the introduction. The problem is to color, if possible, a given graph with k colors only, such that any two adjacent nodes have different colors. A constraint satisfaction formulation of this problem associates the nodes of the graph with variables, the possible colors are their domains, and the inequality constraints between adjacent nodes are the constraints of the problem. Each constraint of a csp may be expressed as a relation, defined on some subset of variables, denoting their legal combinations of values. In addition, constraints can be described by mathematical expressions or by computable procedures.

The structure of a constraint network is depicted by a constraint graph whose nodes represent the variables and in which any two nodes are connected if the corresponding variables participate in the same constraint. In the k-colorability formulation, the graph to be colored is the constraint graph. Constraint networks have proven successful in modeling mundane cognitive tasks such as vision, language comprehension, default reasoning, and abduction, as well as in applications such as scheduling, design, diagnosis, and temporal and spatial reasoning. In general, constraint satisfaction tasks are computationally intractable (NP-hard).

Techniques for processing constraints can be classified into two categories [Dec91]: (1) search and (2) consistency inference, and these techniques interact. Search algorithms traverse the space of partial instantiations, while consistency-inference algorithms reason through equivalent problems. Search is either systematic and complete, or stochastic and incomplete. Likewise, consistency-inference has complete solution algorithms (e.g., variable-elimination) or incomplete versions in the form of *local consistency* algorithms. Let us state this formally.

Definition 1 *A* network of binary constraints *is a set of n variables $X = \{X_1, \ldots, X_n\}$, a domain D_i of possible values for each variable X_i, $1 \leq i \leq n$, and a set of constraints $R_{S_1}, \ldots, R_{S_t}$, where $S_i \subseteq X$. A* binary constraint, *denoted R_{ij} over two variables X_i and X_j, is a subset of the product of their domains, $R_{ij} \subseteq D_i \times D_j$. A* solution *is an assignment of a value to each variable, $x \doteq (x_1, \ldots, x_n)$, $x_i \in D_i$, such that $\forall i, j$, $1 \leq i, j \leq n$, $(x_i, x_j) \in R_{ij}$. A* constraint graph *has a node for each variable and links connecting pairs of variables that appear in the same constraint.*

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, where each node represents a variable having values $\{a, b, c\}$, and each link is associated with a strict lexicographic order ($X_i \prec X_j$ in the lexicographic order if and only if $i < j$). The domains are explicitly indicated on the nodes X_3 and X_4, and the constraints are explicitly listed near the link between them, where a pair (m, n) corresponds to possible values for X_3 and X_4, respectively. This means that solutions can have $(X_3 = a \wedge X_4 = b)$, $(X_3 = a \wedge X_4 = c)$, or $(X_3 = b \wedge X_4 = c)$. For a survey of sequential solution methods for csp's, see [Mac91, Dec91].

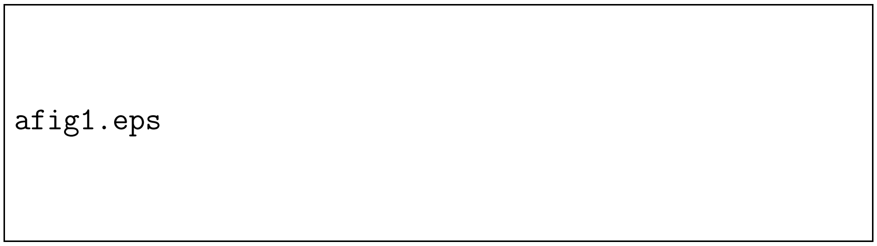

Figure 1: csp constraint graph and a DFS spanning tree

2.2 The communication model

The model consists of n processors arranged in a network architecture. Each node (processor) can communicate only with its neighbors via the communication links. The network can be viewed as a *communication graph* where nodes represent processors and links correspond to communication registers. The communication link between two neighboring processors, i and j, is implemented by two communication registers at both ends of the link. In each register, one processor writes and the other reads. The communication register denoted r_{ij} is written into by node i and may be read only by neighbor j. We also define a *general* communication register, that is written into only by node i but may be read by all of i's neighbors, as a shorthand for a set of communication registers between i and its neighbors that are always assigned the same value. A communication register may have several fields but is regarded as one unit. We expect that the amount of memory used by every processor is relatively small, thus limiting the communications. This eliminates solution schemes that require massive data storage [Mor93] or transmit the whole problem to one processor to solve. Instead of all of the constraints in the system, any one node needs only the constraints between itself and its neighbors. A detailed analysis of the space requirements of our solution is in Section 4.5.2. We assume that the communication and the constraint graphs are identical, and thus two nodes communicate if and only if they are constrained.

A node can be modeled as a finite state-machine whose 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 (if necessary), deciding whether to change its state, and then moving to a new state.[2] A state of the processor encodes the values of its communication registers and its internal variables. A *configuration* c of the system is the state vector of all nodes.

Let c_1, c_2 be two configurations. We write $c_1 \rightarrow c_2$ if c_2 is a configuration that 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 $E = c_0, c_1, \ldots$ such that for every i, $c_i \rightarrow c_{i+1}$. The *initial configuration* is denoted c_0. An execution is considered *fair* if every node participates in it infinitely often.

We present the transition functions as a sequential code in each process. Assuming that the "program counter" is one of the local variables encoded by the state, an execution of the code step-by-step is equivalent to a sequence of state transitions. The collection of all transition functions is called a *protocol*. The processors are anonymous, that is, they have no identities. (We use the terms "node i" and "processor P_i" interchangeably and as a writing convenience only.) This assumption is crucial throughout the paper, since it assures that the processors are truly identical and cannot use their identities to differentiate among sections of code.

We consider two types of scheduling policies. The execution of the system can be managed either by a *central scheduler* (also called an asynchronous *demon*) defined in [Dij74, DIM93] or by a *distributed scheduler* defined in [BGW87, DIM93] (also called a synchronous demon). A distributed scheduler activates a subset of the system's nodes at each step, while a central scheduler activates only one node at a time. All activated nodes execute a single atomic step simultaneously. The central/distributed scheduler can generate any specific schedule (also called an execution) consistent with its

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

definition. Thus, the central scheduler can be viewed as a specific case of the distributed one, since its executions are included in the executions of the distributed scheduler. We say that a problem is *impossible* for a scheduler if for every possible protocol there exists a fair execution generated by such a scheduler that does not find a solution to the problem even if such exists. Note that for different protocols the scheduler can generate different kinds of specific schedules, as long as the condition that defines the type of scheduler is maintained. 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 impossible also for the distributed one.

When a central scheduler is assumed, an interleaving of single operations is sufficient for the analysis of the protocol. Nevertheless, nonneighboring nodes can actually take atomic steps at the same time, even when a central scheduler is assumed, because every such execution can be shown to be equivalent to one where the operations are interleaved (done one-by-one; see [KP90, KP92]).

2.3 Self-stabilization

A self-stabilizing protocol [Dij74] is one with a particular convergence property. The system configurations are 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 self-stabilization model is inherently suited for adapting to changes in the environment and to being fault tolerant. When the protocol can be applied to any network, failure of a link is viewed as self-stabilization of an initial configuration without that link. For instance, in the broadcast transmission problem, the topology of the network may change continuously if the radios are not stationary (such as in war scenarios). A self-stabilizing algorithm may, in some cases, adapt to such changes automatically, without an external control. Moreover, adaptation to local changes may be quick in many cases. In the worst case, though, a local change may require processing through the whole network. Clearly, the legality of a configuration depends on the aim of the protocol.

2.4 The network consistency problem

The *network consistency problem* is defined by mapping a binary constraint network onto a distributed communication model where the variables are associated with processors and communication links with explicit binary constraints. Consequently, the constraint graph and the communication graph are identical.

Since we wish to design a protocol for solving the network consistency problem, 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.

2.5 Related work

In the context of constraint satisfaction, most closely related to our work are attempts to represent csp's and optimization problems on symmetric neural networks, with the hope that the network converges to a full solution [Hop82, HT85, BGS86, Dah87]. Typically, the problem at hand (e.g., a graph coloring) is formulated as an energy-minimization problem on a Hopfield-like network in which the global minima represent the desired solutions. A general method for translating any csp into such a network is presented in [Pin91]. Unfortunately, since the network may converge to a local minimum, a solution is not guaranteed. Another class of algorithms inspired by the connectionist approach is the class of "repair methods" [MJPL90, SLM92, Mor93], also known as *stochastic local search* (SLS).

Additional related work is in the literature on parallel search for constraint satisfaction. Most of that work differs from ours in that the parallel models do not lend themselves easily to distributed communication. Specifically, those models either are not self-stabilized, are nonuniform, or deal with a restricted class of problems [KD94, KR90, FM87, ZM94, YDIK92, Yok95, DD88].

In the self-stabilizing literature, many specific algorithms could be framed as constraint satisfaction problems or treat subtasks useful for constraint satisfaction. Thus a specific algorithm for coloring planar graphs is in [GK93], while self-stabilizing dynamic programming on trees is seen in [GGKP95]. The basic approach for achieving self-stabilization in tree-structured systems is introduced in [Kru79], while one of many algorithms to construct self-stabilizing spanning trees is in [CYH91] with a breadth-first version in [HC92]. Work on local adjustments for self-stabilization [DH97, GGHP96] is also relevant to our solution of constraint systems. Additional details on modeling self-stabilization for dynamic systems is found in [DIM93]. In [AVG96], constraints are used in a very different context than here: They are logical predicates associated with a program (e.g., to describe invariants), and actions are provided by the user that reestablish a constraint once violated.

3 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 [Dij74] 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. A crucial point in the proof is that an algorithm that solves a problem relative to a scheduler must solve it for any specific schedule realizable by the scheduler.

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, each with the domain $\{0, 1, \ldots, n-1\}$. Every link has the set of constraints $\{(i, (i+1) \bmod n) | 0 \leq i \leq n-1\}$, that is, the left node is smaller by 1 than the right. 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 2 *No uniform, self-stabilizing protocol can solve the ring ordering problem with a central scheduler.*

Proof 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, the protocol also converges when initially all nodes are in identical states. We construct a fair execution of such a protocol that satisfies the restriction on the scheduler but for which the network never converges to a consistent solution, contradicting the self-stabilization property of the protocol. Assume the following execution:

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

Note that nodes $P_0, P_q, P_{2q}, \ldots, P_{(r-1)q}$ move to identical states after their first activation, because their inputs, initial states, and transition functions are identical, and when each one of them is activated its neighbors are in identical states too. The same holds for any sequential activation of processors $\{P_{iq+j} | \ 0 \le i < r, \ 0 \le j < q\}$. 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 never reaches a consistent solution, thus yielding a contradiction. Figure 2 demonstrates such a counterexample execution for a ring with six nodes. The indicated nodes are scheduled in each configuration. Different colors refer to different states. □

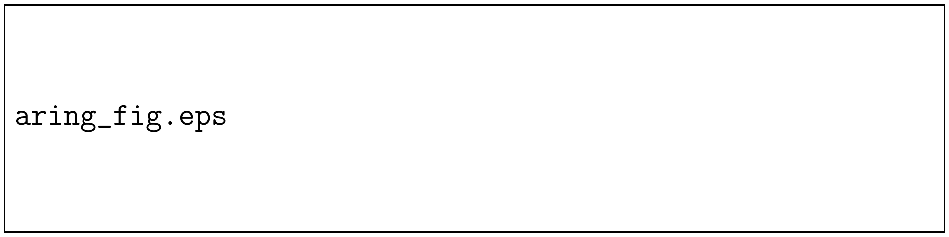

Figure 2: The ring ordering problem ($n = 6$)

Theorem 2 is proven above for a centralized scheduler, but as noted earlier it holds also for a distributed scheduler. This theorem means that it is generally impossible to guarantee convergence to a consistent solution using a uniform protocol if no additional restrictions are made on the possible executions. In particular, convergence cannot be guaranteed for a class of sequential algorithms using "repair" methods, as in [MJPL90], if a completely random order of repair is allowed. It does not, however, exclude the possibility of uniform protocols for restricted scheduling policies or for special network topologies (not including a ring). In practice it is fair to assume that adversarial scheduling policies are not likely to occur or can be deliberately avoided.

When using a distributed scheduler, convergence (to a solution) cannot be guaranteed even for *tree networks*. Consider, for instance, the coloring problem in a tree network constructed from 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 and will never converge to a legal solution, although one exists. This trivial counterexample can be extended to a large class of trees, where there is no possible way to distinguish between two internal nodes. However, we show (Section 5) that for a central scheduler, a uniform self-stabilizing tree network consistency protocol does exist.

Having proven that the network consistency problem cannot always be solved by using a uniform protocol, even with a central scheduler, we switch to a slightly more relaxed model of an *almost uniform* protocol, in which all nodes but one are identical. We denote the special node as node 0 (or P_0). Note that such a special processor can be determined by finding a leader in a distributed network. Thus, if a leader could be elected uniformly, it could be used to convert our almost uniform protocol to a uniform one. Since we cannot solve the consistency problem for a central scheduler with an almost uniform protocol, it follows from our impossibility result (as is well known [Ang80]) that a leader cannot be elected in an arbitrary anonymous (where all processors are identical) network. However, randomization can be used to break the symmetry and to elect a leader in uniform networks [AM94].

4 Consistency-generation protocol

This section presents an almost uniform, self-stabilizing network consistency protocol. The completeness of this protocol (i.e., the guarantee of converging to a solution if one exists) stems from the completeness of the sequential constraint satisfaction algorithm it simulates. It can accommodate changes to constraints, as long as the resulting graph is connected and includes the special node (or one can be elected using randomization). We briefly review some basic sequential techniques for constraint satisfaction.

4.1 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 (*forward* phase). If no consistent assignment can be found for the next variable X_{i+1}, a dead-end situation occurs; the algorithm backtracks to the most recent variable (*backward* phase), changes its assignment, and continues from there.

A useful improvement of backtracking, *graph-based backjumping* [Dec90], 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 dead-end variable. If the variable to which the algorithm retreats has no more values, it backs up further, to the most recent variable among those connected either to the original variable or to the new dead-end variable, and so on.

It turns out that when using a depth-first search (DFS) on the constraint graph (to generate a DFS spanning tree) and then conducting backjumping in a preorder traversal of the DFS tree [Eve79], the jump-back destination of variable X is the parent of X in the DFS spanning tree [Dec91].

The nice property of a DFS spanning tree that allows a parallel implementation is that any arc of the graph that is not in the tree connects a node to one of its tree ancestors (i.e., to a node residing along the path leading to it from the root). Consequently, the DFS spanning tree represents a useful decomposition of the graph: If a variable X and all its ancestors in the tree are removed from the graph, the remaining subtrees rooted at the children of X are disconnected. Figure 1b presents a DFS spanning tree of the constraint graph presented in Figure 1a. Note that if X_2 and its ancestor X_1 are removed from the graph, the network becomes two disconnected trees rooted at X_3 and X_5. 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 [FQ87].

4.2 General protocol description

The distributed version of the binary csp is called the network consistency (NC) problem. Our network consistency protocol is based on a distributed version of graph-based backjumping implemented on a variable ordering generated by a depth-first traversal of the constraint graph.

The NC protocol is logically composed of two self-stabilizing subprotocols that can be executed interleaved, as long as one self-stabilizes for any configuration, and then establishes a condition guaranteeing the self-stabilization of the second [DIM93]. The subprotocols are as follows:

1. DFS spanning tree generation

2. value assignment (using the graph traversal mechanism)

The second subprotocol assumes the existence of a DFS spanning tree in the network. However, the implementations of these subprotocols are unrelated to each other and, thus, can be independently replaced by any other implementation.

Until the first subprotocol establishes a DFS spanning tree, the second subprotocol executes but, in all likelihood, does not lead to a proper assignment of values. We use a self-stabilizing DFS spanning tree protocol that is described in [CD94]. The spanning tree protocol is not affected by the interleaved second subprotocol, and thus the existence of a DFS spanning tree is eventually guaranteed. The convergence of the second subprotocol is also guaranteed starting from any configuration and assuming the existence of a DFS spanning tree (which is not changed by continued operation of the first subprotocol). Therefore, the combination is guaranteed to converge properly after the DFS spanning tree is completed.

The basic idea of the second subprotocol is to decompose the network (problem) logically into several independent subnetworks (subproblems), according to the DFS spanning tree structure, and to instantiate these subnetworks (solve the subproblems) in parallel. Proper control over value instantiation is guaranteed by the graph traversal mechanism presented in Section 4.4.

We would like to emphasize that using graph-based backjumping rather than naive backtracking as the sequential basis for our distributed implementation is crucial for the success of our protocol. First, naive backtracking leads naturally to algorithms in which only one processor executes at a time.

Second, it requires a total ordering of the processors generally encoded in their identifiers. Moreover, unless the nodes that are consecutive in the backtracking order are connected in the communication graph, passing activation from one node to another when going forward or upon a dead end seems feasible only when using node identifiers.

Algorithm graph–based backjumping uses a partial order that is derived from the DFS spanning tree, thus providing more opportunities for parallelism and eliminating the need for node identifiers.

4.3 Self-stabilizing DFS spanning tree generation protocol

First we describe an almost uniform, self-stabilizing protocol for generating a DFS spanning tree. The full algorithm was described in [CD94] and, thus, is not described in detail or proven here. Alternative self-stabilizing algorithms for generating a DFS spanning tree could be used instead and may yield a better space complexity, as discussed in Section 4.5.2.

This subprotocol is the source of nonuniformity for the whole NC protocol. The root of the generated tree is the distinguished node 0 (P_0).

Each processor, P_i, has (at most) one adjacent processor, *parent*(i), designated as its *parent* in the tree, and a set of *child* processors, *children*(i). The set of the processors that reside along the path from the root to P_i in the DFS spanning tree is denoted by *ancestors*(i), while *descendants*(i) is the set of processors P_j so that P_i is in *ancestors*(j). The set of P_i's neighboring processors that are in *ancestors*(i), except the parent of P_i, are called P_i's predecessors and are denoted by *predecessors*(i). Figure 3 indicates the environment of an internal processor. The *ancestors*(i) set is empty if i is the root, while the *descendants*(i) set is empty if i is a leaf.

The links of every processor P are divided into the following categories (see Figure 3):

1. *Tree-edges* are the edges that belong to the DFS spanning tree.
    (a) *Inlink* is the edge that connects P to its parent. Every nonroot processor has one such link and the root has none.
    (b) *Outlinks* are the edges that connect P to its children.
2. *Backward edges* are the edges that connect P to its predecessors.

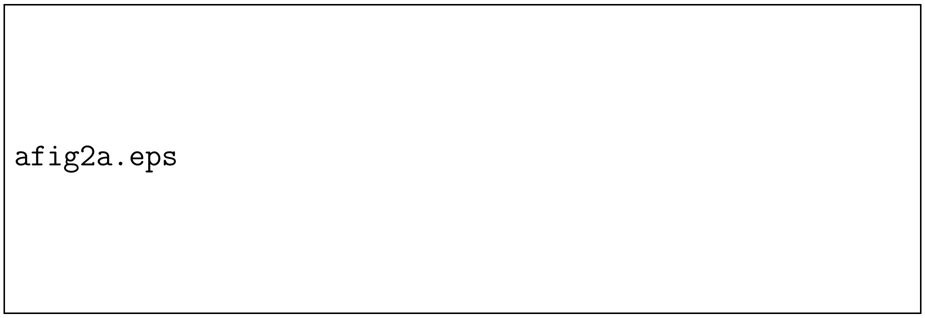

Figure 3: The neighborhood set of i in a DFS-marked graph

3. *Descendant edges* are the edges that connect P to its descendants, excluding its children.

A distributed graph is called *DFS-marked* if there is a DFS spanning tree over the graph such that every processor in the system can identify the category of each of its adjacent edges with relation to this DFS spanning tree.

Each node has a local numbering (ranking) of its edges. During the execution of the protocol the special node 0, which plays the role of the root, repeatedly assigns itself the label $\perp$ (the minimal element) and suggests labels to its neighbors. The label suggested to j by its neighbor i is constructed by concatenating i's ranking of the edge from i to j to the right of i's own label. Thus a label is a sequence of elements, each of which is the minimal element or an edge rank. Labels are compared using lexicographic order. The sequence in a label has a fixed maximal length, and the concatenation can lead to "overflow" where the leftmost element is removed (and this is needed for convergence). Every nonroot node chooses the smallest label suggested to it to be its label and the suggesting neighbor to be its parent, and suggests labels to its neighbors in a similar way.

The communication register between i and j contains the following fields used by the DFS spanning tree generation protocol:

$r_{ij}.mark$ contains the label that is "suggested" to j by i.

$r_{ij}.par$ is a boolean field that is set to TRUE if and only if i chooses j as its parent.

The output of the DFS spanning tree algorithm, after the network has converged, is a DFS-marked graph maintained in a distributed fashion.

4.4 Value assignment subprotocol

The second subprotocol assumes the existence of a DFS spanning tree in the network; namely, each nonroot node has a designated parent, children, and predecessors among its neighboring nodes (see Figure 3). When the DFS spanning tree generation subprotocol stabilizes, each node has a minimal label and a designated parent. Using this information, each node can compute its children set, *children*(i), by selecting the neighbors whose $r_{ji}.par$ field is true, and its predecessors set, *predecessors*(i), by selecting the neighbors whose minimal label ($r_{ji}.mark$ without the last character) is a prefix of its own. This means that we can traverse the directed tree either toward the leaves or toward the root.

The value assignment subprotocol presents a graph traversal mechanism that passes control to the nodes in the order of the value assignment of the variables (in DFS order), without losing the parallelism gained by the DFS structured network. Section 4.4.1 presents the basic idea of *privilege passing* that implements the graph traversal mechanism, while Section 4.4.2 presents the value assignment strategy that guarantees convergence to a solution.

Each node i (representing the 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).

The state register of each node contains the following fields:

$value_i$ is a field to which it assigns one of its domain values or the symbol "$\star$" (to denote a dead end).

$mode_i$ is a field indicating the node's "belief" regarding the status of the network. A node's mode is ON if the value assignment of itself or one of its ancestors has changed since the last time it was in a forward phase (to be explained in Section 4.4.2); 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).

parent_tag and *children_tag* are two boolean fields used for the graph traversal (Section 4.4.1).

Additionally, each node has a sequence set of domain values that is implemented as an ordered list and 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.

4.4.1 Graph traversal using privileges

The graph traversal is handled by a self-stabilizing *privilege passing mechanism*, according to which a node obtains the 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* [Dij74, DIM93]. Once a DFS spanning 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 becomes privileged if its inlink is unbalanced and *all* its outlinks are balanced. In other words, the following two conditions must be satisfied for a node i to be privileged:

1. $parent\_tag_i \neq children\_tag_{parent(i)}$ (the inlink is unbalanced)
2. $\forall k \in children(i) : children\_tag_i = parent\_tag_k$ (the outlinks are balanced)

By definition, we consider the inlink of the root to be unbalanced and the outlinks of the leaves to be balanced.

A node applies the value assignment subprotocol (described in Section 4.4.2) only when it is privileged; otherwise it leaves its state unchanged. As part of the execution of the subprotocol, the node passes the privilege. The privilege can be passed backward to the parent by balancing the inlink or forward to the children by unbalancing the outlinks (i.e., by changing the value of *parent_tag* or *children_tag*, respectively).

We use the following notation to define the set of configurations that are legally controlled relative to the graph traversal:

- Denote a *chain* to be a maximal sequence of unbalanced links, $e_1, e_2 \ldots,$ e_n, such that the following are true:

    1. The inlink of the node whose outlink is e_1 is balanced, unless the node is the root.
    2. Every adjacent pair of links e_i, e_{i+1} $(1 \leq i < n)$ is an inlink and an outlink, respectively, of a common node.
    3. All the outlinks of the node whose inlink is e_n are balanced.

- The chain begins at the node with e_1 as one of its outlinks, denoted as the *chain head*, and ends at the node with the inlink e_n, denoted as the *chain tail*.

- Denote a *branch* to be a path from the root to a leaf.

- A branch *contains* a chain (or a chain is *on* the branch) if all the links of the chain are in the branch.

- A configuration is *legally controlled* if it does not contain any nonroot chain heads, namely, every branch of the tree contains no more than one chain and its chain head is the root.

Figure 4 shows a legally controlled configuration. The DFS spanning tree edges are directed, and the values (+ or −) of the *parent_tag* and the *children_tag* of every node are specified above and below the node, respectively. The privileged nodes are black. In a legally controlled configuration, a node and its ancestors are not privileged at the same time and therefore cannot reassign their values simultaneously. The privileges travel backward and forward along the branches. We prove (Section 4.4.3) that, using the graph traversal mechanism, the network eventually converges to a set of legally controlled configurations that are also legal with respect to the network consistency task.

Once it has become privileged, a node cannot tell where the privilege came from (i.e., from its parent or from its children). Thus, a node uses its *direction* field to indicate the source of its privilege. Since during the legally controlled period no more than one node is privileged on every branch, the privileges travel along the branches backward and forward. The *direction* field of each node indicates the direction of the next expected wave. When afig3.eps

Figure 4: Legally controlled configuration

passing the privilege to its children, the node assigns its *direction* field the BACKWARD value, expecting to get the privilege back during the next backward wave, while when passing the privilege to its parent it assigns the FORWARD value, preparing itself for the next forward wave. Thus, upon receiving the privilege again, it is able to recognize the direction it came from: if *direction* = BACKWARD, the privilege was recently passed toward the leaves, and therefore it can come only from its children; if *direction* = FORWARD, the privilege was recently passed toward the root, and therefore it can come only from its parent. The value of the *direction* field can be improper upon the initialization of the system. However, after the first time a node passes the privilege, its *direction* field remains properly updated. Figure 5 presents the privilege passing procedures for node i.

```
procedure pass-privilege-to-parent
Begin
1.  $direction_i \leftarrow$ FORWARD                                 { prepare for the next wave }
2.  $parent\_tag_i \leftarrow children\_tag_{parent(i)}$             { balance inlink }
End.

procedure pass-privilege-to-children
Begin
1.  $direction_i \leftarrow$ BACKWARD                                { prepare for the next wave }
2.  $children\_tag_i \leftarrow \neg parent\_tag_{k \in children(i)}$  { unbalance outlinks }
End.
```

Figure 5: Privilege passing procedures

4.4.2 Value assignment

The value assignment has forward and backward phases, corresponding to the two phases of the sequential backtracking algorithm. During the forward phase, nodes in different subtrees assign themselves (in parallel) values consistent with their predecessors or verify the consistency of their assigned values. When a node senses a dead end (i.e., it has no consistent value to assign), it assigns its *value* field a $\star$ and initiates a backward phase. Since the root has no ancestors, it does not check consistency and is never dead-ended. It only assigns a new value at the end of a backward phase, when needed, and then initiates a new forward phase.

When the network is consistent (all the nodes are in an OFF mode), the forward and backward phases continue, where the forward phase is used to verify the consistency of the network, while the backward phase just returns the privilege to the root to start a new forward wave. Once consistency is violated, the node sensing the violation relative to its predecessors moves to an ON mode and initiates a new value assignment. 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 ON mode, and passes the privilege to its children. If no consistent value exists, it assigns itself the $\star$ value (a dead end) and passes the privilege to its parent (initiating a backward phase).

- Node i is activated by its parent, which is in an OFF mode (this is the forward phase of consistency verification). In that case, it verifies the consistency of its current value with its predecessors. If it is consistent, it stays in (or moves to) an OFF mode and passes the privilege to its children. If not, it tries to find the next value in its domain that is consistent with all its predecessors, and continues as in the previous case. A leaf, having no children, is always activated by its parent and always passes the privilege back to its parent (initiating a backward phase).

- Node i is activated by its children (backward phase). If one of the children has a $\star$ value, i selects the next value in its domain that is

> consistent with all its predecessors, and passes the privilege back to its children. If no consistent value is available, it assigns itself a $\star$ and passes the privilege to its parent. If all children have a consistent value, i passes the privilege to its parent.

Due to the privilege passing mechanism, when a parent sees one of its children in a dead end, it still has to wait until *all* of them have given it the privilege. This is done to guarantee that all subtrees have a consistent view regarding their predecessors' values. This requirement limits the amount of parallelism considerably. It can be relaxed in various ways to allow more parallelism.

The algorithms performed by a nonroot node ($i \neq 0$) and the root once they become privileged and after reading the neighbors' states are presented in Figures 6 and 7.

The procedure *compute-next-consistent-value* (Figure 7) tests each value located after the domain pointer for consistency. More precisely, the domain value is checked against each of *predecessors*(i)'s values, and the next domain value consistent with the values of the predecessors is returned. The pointer's location is readjusted accordingly (i.e., to the found value), and the mode of the node is set to ON. If no consistent value is found, the value returned is $\star$ and the pointer is reset to the beginning of the domain. The predicate *consistent(val*, *set_of_nodes)* is TRUE if the value of *val* is consistent with the *value* fields of *set_of_nodes* and none of them is dead-ended (has the value $\star$).

The algorithm performed by the root P_0, when it is privileged, is simpler. The root does not check consistency. All it does is assign a new value at the end of each backward phase, when needed, and then initiate a new forward phase. The procedure *next-value* increments the domain pointer's location and returns the value indicated by the domain pointer. If the end of the domain list is reached, the pointer is reset to the first (smallest) value.

The value assignment subprotocol can be regarded as uniform, since each node may have both the root's protocol and the nonroot's protocol and decide between them based on the role assigned to it by the DFS spanning tree protocol.

4.4.3 Proof of self-stabilization

To prove the correctness of our NC protocol, we first prove that the graph traversal is self-stabilizing, namely, that the system eventually reaches a

```
root 0:
Begin
if $\exists k \in children(0) :\ value_k = \star$ then
    if $\neg$consistent($value_0, children(0)$) then
        $mode \leftarrow$ ON
        $value \leftarrow$ next-value
    else                                  { all children are consistent }
        $mode \leftarrow$ OFF
    pass-privilege-to-children
End.

nonroot $i$:
Begin
    if $direction =$ FORWARD then                  { forward phase }
        if $mode_{parent(i)} =$ ON then            { a change in a value assignment occurred }
            $pointer \leftarrow 0$                 { reset domain pointer }
        else                                       { parent's mode is OFF}
            if consistent($value_i, predecessors(i)$) then
                $mode \leftarrow$ OFF
    if $(pointer = 0) \vee \neg$consistent($value_i, predecessors(i)$) $\vee$
                $\vee(direction =$ BACKWARD $\wedge\ \neg$consistent($value_i, children(i)$)) then
        $value \leftarrow$ compute-next-consistent-value
                                                   { privilege passing }
    if leaf($i$) $\vee\ (value = \star) \vee (direction =$ BACKWARD $\wedge$ consistent($value_i, children(i)$))
        then pass-privilege-to-parent
        else pass-privilege-to-children
End.
```

Figure 6: Value assignment subprotocols for root and nonroot nodes

```
procedure compute-next-consistent-value
Begin
$mode_i \leftarrow$ ON
while $pointer \leq$ endof $Domain_i$ do
    $pointer \leftarrow pointer + 1$
    if consistent($Domain_i[pointer]$, $predecessors(i)$) then
        return $Domain_i[pointer]$          { a consistent value was found }
$pointer \leftarrow 0$
return $\star$          { no consistent value exists }
End.
```

Figure 7: Consistency procedure

legally controlled configuration (even if the values in the nodes are not yet consistent), and from that point it remains legally controlled. Assuming the system is legally controlled, we show that if a legal assignment exists, it is eventually reached and thereafter remains unchanged. Thus the system reaches a legal set of configurations and stays there—and therefore is self-stabilizing.

Note that a nonroot node is privileged when its inlink is unbalanced (thus it is on a chain) and all its outlinks are balanced. In other words, a nonroot node is privileged if and only if it is a chain tail. The root is privileged if and only if it is not a chain head. Also note that passing the privilege by a node affects only the chains on the branches containing that node, because it has no interaction with other branches.

To prove the self-stabilization of the privilege passing mechanism, we first prove some of its properties.

Lemma 3 *In every infinite fair execution, every nonroot node that is on a chain eventually passes the privilege to its parent.*

Proof We prove the lemma by induction on the node's height, h (i.e., its distance from the nearest leaf), and on the possible value assignments from the domain of the node.

Base: $h = 0$. The node is a leaf and, therefore, when activated, can pass the privilege only to its parent.

Step: Assume node i, whose height is $h > 0$, is on a chain. Node i eventually becomes privileged because, if any of i's outlinks are unbalanced, then the corresponding children are on chains and the induction hypothesis holds for them, namely, they eventually pass the privileges to i. Note that a node that passes its privilege to the parent (to i in our case) does not become privileged again unless its parent had become privileged first and passed the privilege to its children, since outlinks are unbalanced by a privileged node only.

If, when becoming privileged, i passes the privilege to its parent, the claim is proven. Otherwise, whenever i passes the privilege to its children the same argument holds, so i eventually becomes privileged again. Moreover, i's domain pointer is advanced every time it passes the privilege to its children. Therefore, after a finite number of such passings, bounded by the size of $domain_i$, the domain pointer reaches a $\star$ and then, following the code in Figures 6 and 7, i passes the privilege to its parent. □

Theorem 4 *The graph traversal mechanism is self-stabilizing with respect to the set of legally controlled configurations. Namely, it satisfies the following assertions:*

1. *Reachability: Starting from any initial configuration, the system eventually reaches a legally controlled configuration.*
2. *Closure: If c is a legally controlled configuration and $c \rightarrow c'$, then c' is also a legally controlled configuration.*

Proof We prove the theorem by showing that all the nonroot chain heads in the network eventually disappear. Note that passing a privilege by the root makes the root a chain head but does not increase the number of nonroot chain heads. Passing a privilege by any nonroot node does not create any chain head. When a nonroot node passes the privilege, it is a chain tail (its outlinks are balanced). Thus, if the privilege is passed to the node's children, none of them become a chain head since their parent is still on the same chains. On the other hand, if the privilege is passed to its parent, the node balances its inlink, which cannot possibly create a new chain head. Thus the number of nonroot chain heads in the network never increases. Moreover, Lemma 3 implies that every nonroot node that is on a chain, and particularly any nonroot chain head, eventually passes the privilege to its parent and stops being a chain head. Therefore, the number of nonroot chain heads steadily decreases until no nonroot chain heads are left, and hence the network is legally controlled. Since no nonroot chain heads are ever created, the network remains legally controlled forever. □

The self-stabilization property of the NC protocol is inherited from its subprotocols: DFS spanning tree generation and value assignment. Once the self-stabilization of privilege passing is established, it assures that the control is a correct, distributed implementation of DFS-based backjumping, which guarantees the convergence of the network to a legal solution, if one exists, and if not, it repeatedly checks all the possibilities.

4.5 Complexity analysis

4.5.1 Time complexity

A worst-case complexity estimate of the value assignment subprotocol, once a DFS tree already exists, can be given by counting the number of state changes until the network becomes *legally controlled* and by adding to it the number of state changes before a legal *consistent configuration* is reached. These two counts bound the sequential performance of the value assignment protocol and thus, also, its worst-case parallel time. The bound is tight since it can be realized when the depth of the DFS tree equals n. In that case only one node is activated at a time. We next bound the sequential time and the parallel time, as a function of the depth of the DFS tree, m. We show that in some cases an optimal speedup is realizable.

Let T^1_m stand for the maximal number of privilege passings in the subnetwork with a DFS spanning subtree of depth m, before its root completes a full round of assigning all of its domain values (if necessary) and passing the privilege forward to its children for every assigned value (for a nonroot node it is the number of privilege passings in its subtree before the node passes the privilege backward). Let b be the maximal branching degree in the DFS spanning tree, and let k bound the domain sizes. Since every time the root of a subtree becomes privileged, it either tries to assign a new value or passes the privilege backward, T^1_m obeys the following recurrence:

$$\begin{aligned} T^1_m &= k \cdot b \cdot T^1_{m-1}, \\ T^1_0 &= k. \end{aligned}$$

Solving this recurrence yields $T_m^1 = b^m k^{m+1}$, which is the worst-case number of privilege passings before reaching the legally controlled configuration where only the root is privileged.

The worst-case number of additional state changes toward a consistent solution (if one exists) is bounded by the worst-case time of sequential graph-based backjumping on the DFS tree-ordering. Let T_m^2 stand for the maximal number of value reassignments in the subnetwork with a DFS spanning subtree of depth m, before it reaches a solution. This equals the search space explored by the sequential algorithm. 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^2 obeys the same recurrence as T_m^1 and results in the same expression: $T_m^2 = b^m k^{m+1}$.

Thus, the overall worst-case time complexity of the value assignment subprotocol is $T_m = T_m^1 + T_m^2 = O\left(b^m k^{m+1}\right)$. Note that when the tree is balanced, we have $T_m = O\left((n/b) \cdot k^{m+1}\right)$ since $n = O(b^{m+1})$.

Next, we evaluate the parallel time of the value assignment subprotocol, assuming that the privilege passing mechanism has already stabilized into legally controlled configurations (namely, the root is activated). For this analysis we assume that the DFS tree is balanced. Clearly, when the tree is not balanced, the speedup reduces as a function of b. Consider now two cases. Assume that the network is backtrack-free. In that case, since there are no dead ends, the sequential time obeys the recurrence

$$\begin{aligned} T_m^2 &= k + b \cdot T_{m-1}^2, \\ T_0^2 &= 1, \end{aligned}$$

yielding

$$\begin{aligned} T_m^2 &= b^m + k \cdot (b^{m+1} - 1)/(b-1) \\ &= O(n/b \cdot k). \end{aligned}$$

The parallel time obeys the recurrence

$$\begin{aligned} T_m^2 &= k + T_{m-1}^2, \\ T_0^2 &= 1, \end{aligned}$$

yielding $T_m^2 = m \cdot k + 1$. The overall speedup in this case is bounded by $n/(b \cdot m)$ where $m = \log_b n$.

Consider now the general case while still assuming that the tree is balanced. The sequential complexity, as shown earlier, is $O(n/b \cdot k^{m+1})$. In that case, since the subtrees are traversed in parallel, the parallel complexity obeys the recurrence

$$\begin{aligned} T_m^2 &= k \cdot T_{m-1}^2, \\ T_0^2 &= k, \end{aligned}$$

yielding $T_m^2 = k^{m+1}$. In this case a speedup of (n/b) seems realizable.

In summary, we have shown that, as in the sequential case, our protocol's complexity is exponentially proportional to the depth of the DFS spanning tree; that is, the system has a better chance for a "quick" convergence when the DFS spanning tree is of a minimal depth. There is no gain in speedup when the depth of the tree is n. However, for balanced trees having a depth of $m = log_b n$, the speedup lies between n/b and $n/(b \cdot m)$.

4.5.2 Space complexity

Each node needs to have the information about the constraints with its neighbors. Assuming that the maximum degree in the graph is d and since a constraint can be expressed in $O(k^2)$, each node needs space for $O(d \cdot k^2)$ values. Among the subprotocols, the DFS subprotocol requires the most space, $O(n \cdot \log d)$ bits. Thus there is an overall space requirement for each processor of $O(n \cdot \log d + d \cdot k^2)$ values, using our subprotocols. In [DJPV98] a DFS spanning tree can be accomplished using only $O(\log d)$ space per node, but the time needed for stabilization may be longer.

In order to store the whole network information, a processor needs space for $O(n^2 \cdot k^2)$ values, so our distributed treatment is clearly superior to a centralized solution. Note that the log encoding common in the analysis of space requirements may not be feasible in practice for this context, because the communication registers are fixed, values must be communicated, and constraints must be changed during execution.

4.5.3 Speedup of incremental change

Our model allows adaptation to change without external intervention. Moreover, in many cases a change may be locally, and thus quickly, stabilized. For example, suppose, after the network has converged to a solution, that the domain of a variable is made smaller by some external event or that a new constraint is introduced. If these newly enforced restrictions are consistent with the current assignment, there is no change to the solution. Alternatively, if a current assignment does not satisfy the new constraint, at least one processor detects the problem, since it repeatedly checks consistency, and tries to choose an alternative value that satisfies its local environment. If it succeeds and if the neighbors are happy with the new assignment, change stops. It is clear, however, that in the worst case, local changes may cause a complete execution of the algorithm that may be time exponential.

4.5.4 Adding arc consistency

The average performance of the NC protocol can be further improved by adding to it a uniform self-stabilizing *arc consistency* subprotocol [MF85]. 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, since the domain sizes are finite, and they can only shrink or be left intact by each activation. As a result, after a finite number of steps all the domains remain unchanged.

Since arc consistency can be applied in a uniform and self-stabilizing manner, it suggests that various constraint propagation methods can be incorporated to improve backjumping [FD94], while maintaining properties of self-stabilization. In particular, the well-known technique of forward checking can be used along with arc consistency during the value assignment subprotocol. If, as a result, a future variable's domain becomes empty, that information can be propagated back to the current privileged variable. The details and impact of such improvements remain to be worked out.

5 Network consistency for trees

It is well known that the sequential network consistency problem on trees is tractable and can be achieved in linear time [MF85]. A special algorithm for this task is composed of an arc consistency phase (explained in the previous section) that can be efficiently implemented on trees, followed by a backtrack-free value assignment in an order created by some *rooted tree.*

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

In principle a uniform tree consistency (TC) protocol can be extracted from the general NC protocol by replacing the DFS spanning tree protocol with a uniform rooted tree protocol to direct an undirected tree, since in trees any rooted tree is a DFS tree. Since the arc consistency protocol and the value assignment protocol are already uniform, the resulting TC protocol is uniform. Nevertheless, we show that for trees, the value assignment protocol can be simplified as well, while there is no need for a special privilege-passing mechanism. The proposed TC protocol consists of the three subprotocols: tree directing, arc consistency, and tree value assignment.

When the variables' domains are arc 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 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.

To direct the tree uniformly, we must exploit the topology of the tree to break the symmetry reflected by the identical codes and the lack of identifiers. For this task we use a distributed protocol for finding the *centers* of a tree [KRS84, KPBG94]. 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. Note that if we regard a center as the root of the tree, the children of every node are removed from the tree in earlier phases than the node itself (except in the case of two centers that are removed in the same, last, phase of the algorithm), which means that the removal phase of a node is greater than the removal phase of any of its children, and the removal phase of its parent is greater than (or, in the case of two centers, equals) its own. We denote the removal phase of a node in this sequential algorithm as its *level*. The level of the leaves is 0. Another way to define the level of a node is its maximal distance to a leaf without passing through a center.

Our protocol distributedly simulates the above algorithm. If only one center exists, it declares itself as a root, and all the incident edges are directed toward it. When two centers exist, one of them becomes a root and the link that connects them is directed accordingly. The choice of which center becomes the root is not deterministic and depends on the scheduling order and the initial values.

This approach yields a simple, uniform, tree-directing protocol that simulates the above description. Every node i has the following fields:

l_i is the level of i, a variable that eventually indicates the phase of the sequential algorithm in which i is removed from the tree.

$root_i$ is a boolean field that indicates whether i is the root.

$parent_i$ is a variable assigned the number of the edge leading to the neighbor that becomes the parent of i.

The protocol works by having each node repeatedly compute its own l-value by adding 1 to the second-largest value among the l-values of its neighbors. Each node except the centers ultimately chooses as its parent the neighbor to which it is still connected whenever it becomes a leaf, namely, that neighbor whose level is greater than its own.

A node views itself as a center when one of the following two conditions is satisfied:

1. Its l-value is greater than all of its neighbors', which means that it is a single center.

2. Its l-value is equal to that of its largest neighbor—the other center. In this case, the node checks whether the other center is already the root. If so, it chooses the other center to be its parent; otherwise it becomes the root.

Because the l-values converge to the correct values of the level of the node, the above interpretation is ultimately accurate. Clearly, once the l-values converge, each node properly chooses its parent, and the direction of the tree is completed. Therefore, it is sufficient to prove the convergence of the

l-values to the levels of the nodes. A proper convergence of the l-values can be proved by induction on the levels of the nodes. For more details of the tree-directing algorithm and its proof see [CDK94].

The parallel time can be linearly bounded by the *diameter* (dim) of the tree, where the diameter is the longest path between any two leaves of the tree. Since in the worst case the diameter of a tree equals the number of nodes, n, the space that is required in this case to hold the level of each node is $O(\log dim)$. A different self-stabilizing tree-directing algorithm is presented in [PD92]. In that algorithm, any node of the tree may become the root, depending on the initial configuration and the schedule. Although that algorithm is usually better in its space requirements than the one presented above, forcing a center to become the root, as is done here, yields a more balanced tree.

6 Conclusions

The results presented in this paper establish theoretical bounds on the capabilities of distributed self-stabilizing architectures to solve constraint satisfaction problems. The paper focuses on the feasibility of solving the network consistency problem using self-stabilizing distributed protocols, namely, guaranteeing convergence to a consistent solution, if such exists, from *any* initial configuration.

We have proved that, when any scheduler is allowed, 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 (i.e., when using a central scheduler). Consequently, although such protocols have obvious advantages, they cannot guarantee convergence to a solution. On the other hand, distinguishing one single node from the rest 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. Note that the negative results are established under a model requiring convergence for every central or distributed schedule and is not applicable to many cases where the schedule is restricted. We demonstrated that when the network is restricted to trees, a uniform, self-stabilizing protocol exists for solving the problem with any central scheduler, where only one neighboring node is activated at a time.

Note also that the restriction to a central scheduler is not as severe as it might at first appear. Any protocol that works with a central scheduler can also be implemented with a distributed scheduler that obeys the restriction that two neighboring nodes are never activated together. It is still an open question whether a uniform protocol is feasible for general graphs under restricted scheduling policies (e.g., round-robin).

Regarding time complexity, we showed that in the worst 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 speedup between n/b and $n/(b \cdot m)$ is possible, where n is the number of nodes and m is the DFS's tree depth. We also argued that when the environment undergoes local change, the solution to the network consistency problem can often be repaired quickly (but not for all cases), due to the inherent local computation of the distributed architecture.

Acknowledgments

We thank Shlomo Moran for help in the proof of the root-finding protocol and Arun Jagota for his advice on related work and for useful discussions.

Acknowledgment of support

Dechter was partially supported by the National Science Foundation, grant number IIS9610015, and by the Air Force Office of Scientific Research, grant number AFOSR-90-0136. Katz was partially supported by the Argentinian Research Fund at the Technion-Israel Institute of Technology.

References

[AM94] Y. Afek and Y. Matias. Elections in anonymous networks. *Information and Computation*, 113:312–330, 1994.

[Ang80] D. Angluin. Local and global properties in networks of processes. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 82–93. ACM Press, 1980.

[AVG96] A. Arora, G. Varghese, and M. Gouda. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. *Journal of High-Speed Networks*, 5:1–14, 1996.

[BGS86] D. H. Ballard, P. C. Gardner, and M. A. Srinivas. Graph problems and connectionist architectures. Technical Report 167, University of Rochester, Rochester, N.Y., March 1986.

[BGW87] J. Burns, M. Gouda, and C. L. Wu. A self-stabilizing token system. In *Proceedings of the 20th Annual International Conference on System Sciences*, pages 218–223, Hawaii, 1987. Western Periodicals.

[CD94] Z. Collin and S. Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49:297–301, 1994.

[CDK91] Z. Collin, R. Dechter, and S. Katz. On the feasibility of distributed constraint satisfaction. In *Proceedings of IJCAI-91*, Sydney, Australia, 1991. Morgan Kaufmann.

[CDK94] Z. Collin, R. Dechter, and S. Katz. Self-stabilizing distributed constraint satisfaction. Technical report, Information and Computer Science (R15b), 1994. www.ics.uci.edu/∼dechter/publications/R15.

[CYH91] N. S. Chen, H. P. Yu, and S. T. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[Dah87] E. D. Dahl. Neural network algorithms for an np-complete problem: Map and graph coloring. In *Proceedings of the IEEE First International Conference on Neural Networks*, pages 113–120, San Diego, 1987.

[DD88] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings AAAI-88*, pages 37–42, St. Paul, Minn., August 1988. Morgan Kaufmann.

[Dec90] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence Journal*, 41(3):273–312, January 1990.

[Dec91] R. Dechter. Constraint networks. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 276–285. Wiley, New York, 1991.

[DH97] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 3(4), 1997.

[Dij74] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[DIM93] D. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[DJPV98] A. K. Ditta, C. Johnen, F. Petit, and V. Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. In *Proceedings of SIROCCO'98, International Colloquium on Structural Information and Communication Complexity*, 1998.

[Eve79] S. Even. *Graph Algorithms*. Computer Science Press, Rockville, Md., 1979.

[FD94] D. Frost and R. Dechter. In search of best search: An empirical evaluation. In *AAAI-94: Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 301–306, Seattle, Wash., 1994. Morgan Kaufmann.

[FM87] R. Finkel and U. Manber. Scalable parallel formulations of depth-first search. *ACM Transactions on Programming Languages and Systems*, 9:235–256, 1987.

[FQ87] 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, N.H., 1987.

[GGHP96] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54. ACM Press, 1996.

[GGKP95] S. Ghosh, A. Gupta, M. H. Karaata, and S. V. Pemmaraju. Self-stabilizing dynamic programming algorithms on trees. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 11.1–11.15, 1995.

[GK93] S. Ghosh and M. H. Karaata. A self-stabilizing algorithm for coloring planar graphs. *Distributed Computing*, 7:55–59, 1993.

[HC92] S. T. Huang and N. S. Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41:109–117, 1992.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hop82] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Science*, volume 79, pages 2554–2558, 1982.

[HT85] J. J. Hopfield and D. W. Tank. Neural computation of decisions in optimization problems. *Biological Cybernetics*, 52:144–152, 1985.

[KD94] S. Kasif and D. Delcher. Local consistency in parallel constraint networks. *Artificial Intelligence*, 69(1–2):307–327, 1994.

[KP90] S. Katz and D. Peled. Interleaving set temporal logic. *Theoretical Computer Science*, 75:263–287, 1990.

[KP92] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.

[KPBG94] M. Karaata, S. Pemmaraju, S. Bruell, and S. Ghosh. Self-stabilizing algorithms for finding centers and medians of trees (brief announcement). In *Proceedings of PODC'94, Thirteenth ACM Symposium on Principles of Distributed Computing*, page 374. ACM Press, 1994.

[KR90] V. Kumar and V. N. Rao. Scalable parallel formulations of depth first search. In V. Kumar, P. S. Gopalakrishnan, and L. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 1–41. Springer-Verlag, 1990.

[KRS84] 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.

[Kru79] H. S. M. Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8:91–95, 1979.

[Les90] V. R. Lessr. An overview of DAI: Viewing distributed AI as distributed search. *Japanese Society for Artificial Intelligence*, 5(4), 1990.

[Mac91] A. Mackworth. Constraint satisfaction. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–292. Wiley, New York, 1991.

[MF85] A. K. Mackworth and E. C. Freuder. The complexity of some polinomial network consistency algorithms for constraint satisfaction problems. *Artificial intelligence*, 25:65–74, 1985.

[MJPL90] 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*, volume 1, pages 17–24, Boston, 1990. Morgan Kaufmann.

[Mor93] P. Morris. The breakout method for escaping from local minima. In *AAAI-93*, pages 40–45, San Jose, Calif., 1993. Morgan Kaufmann.

[PD92] G. Pinkas and R. Dechter. An improved connectionist activation for energy minimization. In *AAAI-92*, pages 434–439, San Jose, Calif., 1992. Morgan Kaufmann.

[Pin91] G. Pinkas. Energy minization and the satisfiability of propositional calculus. *Neural Computation*, 3(2):282–291, 1991.

[RL92] R. Ramanathan and E. L. Lloyes. Scheduling algorithms for multi-hop radio networks. In *Proceedings of the SIGCOMM'92, Communication Architectures and Protocols*, pages 211–222, New York, 1992. ACM Press.

[SLM92] B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *AAAI-92*, pages 440–446, San Jose, Calif., 1992. Morgan Kaufmann.

[YDIK92] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *IEEE 12th International Conference on Distributed Computing Systems*, pages 614–621, 1992.

[Yok95] M. Yokoo. Asynchronous weak commitment search for solving distributed constraint satisfaction problems. In *First International Conference on Constraint Programming*, France, 1995. Springer-Verlag.

[ZM94] Y. Zhang and A. Mackworth. Parallel and distributed finite constraint satisfaction: Complexity, algorithms and experiments. In L. N. Kanal, V. Kumar, H. Kitano, and C. B. Suttner, editors, *Parallel Algorithms for Machine Intelligence and Vision*, pages 1–41. Elsevier, Amsterdam, 1994.