Self-stabilizing network orientation algorithms in arbitrary rooted networks

Shivashankar Gurumurthy

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/1035

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
The Thesis prepared by

Shivashankar Gurumurthy

Entitled

Self Stabilizing Network Orientation Algorithms In Arbitrary Rooted Networks

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Examination Committee Chair

Dean of the Graduate College
Network orientation is the problem of assigning different labels to the edges at each processor, in a globally consistent manner. A self-stabilizing protocol guarantees that the system will arrive at a legitimate state in finite time, irrespective of the initial state of the system. Two deterministic distributed network orientation protocols on arbitrary rooted, asynchronous networks are proposed in this work. Both protocols set up a chordal sense of direction in the network. The protocols are self-stabilizing, meaning that starting from an arbitrary state, the protocols are guaranteed to reach a state in which every processor has a valid node label and every link has a valid edge label. The first protocol assumes an underlying depth-first token circulation protocol; it orients the network as the token is passed among the nodes and stabilizes in $O(n)$ steps after the token circulation stabilizes, where $n$ is the number of processors in the network. The second protocol is designed on an underlying spanning tree protocol
and stabilizes in $O(h)$ time, after the spanning tree is constructed, where $h$ is the height of the spanning tree. Although the second protocol assumes the existence of a spanning tree of the rooted network, it orients all edges—both tree and non-tree edges—of the network.
TABLE OF CONTENTS

ABSTRACT ................................................................................................................. iii

ACKNOWLEDGMENTS ............................................................................................... vii

CHAPTER I INTRODUCTION .................................................................................. 1
   Distributed System ............................................................................................... 1
   Fault Tolerance ................................................................................................... 2
   Network Orientation ............................................................................................ 3
   Related Work ........................................................................................................ 5
   Contributions ....................................................................................................... 5

CHAPTER II PRELIMINARIES ............................................................................. 8
   Self-Stabilizing System ....................................................................................... 8
     System ............................................................................................................... 8
     Programs .......................................................................................................... 9
     Predicates ....................................................................................................... 10
     Self-Stabilization ............................................................................................ 10
   Chordal Sense of Direction ............................................................................... 11
   Specification of the Network Orientation Protocol ......................................... 11

CHAPTER III NETWORK ORIENTATION USING DEPTH-FIRST TOKEN PASSING .... 13
   Algorithm $DFTNO$ ............................................................................................ 13
     Node Labeling ................................................................................................. 15
     Assigning edge labels at each processor ....................................................... 16
   Correctness of the Network Orientation Protocol $DFTNO$ ........................... 16
     Correctness of the Node Naming Phase ......................................................... 17
     Correctness of the Edge Labeling Phase ....................................................... 17
   Space and Time Complexity ............................................................................. 18

CHAPTER IV NETWORK ORIENTATION USING SPANNING TREE PROTOCOL .... 19
   Algorithm $STNO$ .............................................................................................. 19
     Node Labeling .................................................................................................. 23
     Edge Labeling ................................................................................................. 23
Correctness of the Network Orientation protocol $STNO$ ................................................... 24
Correctness of the Node Naming Phase .......................................................... 24
Correctness of the Edge Labeling Phase .................................................. 25
Space and Time Complexity .................................................................. 25

CHAPTER V CONCLUSION ........................................................................ 26

BIBLIOGRAPHY ......................................................................................... 28

VITA ........................................................................................................... 31
ACKNOWLEDGMENTS

First and foremost I wish to thank my advisor Dr. Ajoy Datta, whose help and guidance cannot be quantified with mere words. He has been a great inspiration throughout and has strived to derive the best efforts from me. His valuable inputs at critical junctures have never failed to overcome bottlenecks. I also like to thank Dr. Kazem Taghva, Dr. John Minor and Dr. Ashok Iyer for serving on my thesis committee. Special thanks are due to Dr. Vincent Villain and Dr. Franck Petit for their valuable ideas towards improving the design of the algorithms. I wish to thank the University of Nevada, Las Vegas and especially the Computer Science Department for providing me an excellent education and financial support for the most rewarding two years of my life. I am indebted to Dr. Yfantis and Dr. Gewali, for sharing their abundant knowledge and experience with me, for supporting me through times of crisis, for encouraging me to achieve greater heights and also for being great friends. I wish to thank all my friends here who have given me an ambient and cordial environment to work in, and being supportive during my stay in Vegas. Lastly, I wish to thank my parents and God together for all their good wishes and blessings that have been instrumental in bringing me this far.
CHAPTER 1

INTRODUCTION

1.1 Distributed System

A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software. Distributed system software enables computers to coordinate their activities and to share the resources of the system - hardware, software, and data. Users of well-designed distributed systems should perceive a single, integrated computing facility even though it may be implemented by many computers in different locations.

The key characteristics of a distributed system are: support for resource sharing, openness, concurrency, scalability, fault-tolerance and transparency. Resource sharing denotes the multiple access or usage of components such as disks, printers, files, databases and other data objects. Openness of a distributed system determines the adaptability of the system to modular extensions. When several processes exist in a single computer, we say that they are executed concurrently. Scalability of a distributed system denotes the efficiency of the software under increased load. Transparency is defined as the concealment from the user, the separation of the individual components in the system, so that it is perceived as one whole unit rather than a collection of components. Finally, fault-tolerance is defined as the ability of the distributed system to recover from faults. We discuss this last point in detail, in the following section.
1.2 Fault Tolerance

Design of fault tolerant system is based on two approaches, both of which must be deployed to handle each fault: hardware redundancy - use of redundant components, software recovery - the design of programs to recover from faults.

To produce systems that are tolerant to hardware failures, two interconnected computers are often employed for a single application, one of them acting as a standby machine for the other. This is a costly solution. Allocation of redundant hardware required for fault-tolerance can be designed such that hardware is exploited for non-critical activities when no faults are present. By such techniques, tolerance to some types of hardware faults can be provided in a distributed system at a relatively low cost.

Software recovery involves designing of software, such that the state of the permanent data can be recovered or 'rolled-back', when a fault is detected. Many software mechanisms have been devised that enables files and other persistent data to be restored to the state they were in before the failed program began its execution.

One of the most inclusive and unified approaches to fault-tolerance distributed system design is *Self-stabilization*. A self-stabilizing system guarantees that regardless of the current state, the system recovers to a legal configuration in a finite number of steps, and remains in a legal state until another fault occurs. There are many advantages in designing such a self-stabilizing system. No startup or initialization procedures are necessary since the system converges to legal state from any arbitrary state. A fault occurring at a process may cause a illegal global state, but the system will detect such a state, and correct itself in finite time. The ability of the system to detect errors and correct itself without external intervention makes a self-stabilizing system more reliable and more powerful than non-stabilizing systems.

Self-stabilization is defined as an exercise for achieving global convergence through
local actions. It is an easy and complete model for fault-tolerance. Every processor participates in the algorithm and executes only according to its local code under all situations. Other models may allow for a subset of processors to fail with correctness only guaranteed for the non-faulty processors. In self-stabilization all processors can start at faulty local state before converging to a global legal state.

Self-stabilization and its impact on distributed systems has been studied in detail since its conception by Dijkstra [11]. Common approach in this area has been focussed on various aspects such as models of self-stabilization, methods for developing self-stabilizing systems, limitations and costs of self-stabilization, and proof techniques. Many distributed algorithms have been modified to include self-stabilization, like mutual exclusion, leader election, network message passing protocols and routing protocols, and various graph algorithms.

Several token-passing mutual exclusion algorithms on different topologies have been proposed. [4, 7, 11, 17, 13] for ring; [5, 15, 26] for linear array of processors. Huang and Chen [18] present a token circulation protocol for network in nondeterministic DFS order and Dolev, Israeli and Moran [12] give a mutual exclusion algorithm on a tree network. Datta et al. [10] present a self-stabilizing token-passing protocol on a rooted network, which has an extremely small state requirement.

Apart from this, other distributed algorithms such as spanning tree construction [1, 3, 8, 12, 16], PIF on tree networks, and shortest paths problem in a graph have been self-stabilized.

1.3 Network Orientation

The network orientation problem concerns the assignment of different labels or directions to the edges of each processor, in a globally consistent manner. The choice of labeling function varies with the application and the graph topology on hand. The
label of an edge indicates which direction in the network this edge leads to. The labels can be used in many applications, such as routing and traversal in networks. Every node assigns a unique label to each of its incident edge.

Various classes of labeling schemes have been proposed. A cartographic edge labeling uses properties of an embedding of the graph in a plane. Coordinate labeling, an instance of the cartographic edge labeling class, is one which labels the edge \( <u, v> \) at \( u \) by the relative coordinates of \( v \) from \( u \). In a chordal labeling, a cyclic ordering of the nodes is fixed and each incident link in a node is labeled by the distance of the connecting node in the above cycle. Note that in a chordal labeling, the edge is labeled as an integer value and its inverse modulo \( N \), by the nodes at either end of the edge. In a neighboring labeling scheme, all edges ending in the same node are labeled with the same label at the connecting nodes.

A labeling of a network is said to be in Local Orientation if it is an injective function. A labeling is said to have Edge Symmetry, if knowledge of the label on one side allows to derive the label on the other side if the edge. A labeling is a Locally Symmetric Orientation when it in Local Orientation with Edge Symmetry.

Any node which labels its edges consistently has a much clearer local view of the network. Such information could be used in various applications such as routing and message passing protocols. Another very important fact is that the assignments of labels to edges have a dramatic effect on the communication complexity [21, 25]. Though many papers have assumed a network orientation as an underlying protocol and have come up with better algorithm complexities, setting up the node and edge labels in the network have not been dealt with clearly.
1.4 Related Work

It was demonstrated by Santoro [21] that the availability of an orientation decreases the message complexity of important computations of several topologies. Many subsequent papers have assumed oriented networks in order to reduce the algorithm complexity. Surprisingly, there are very few papers that have addressed the question how orientations can be computed in networks where no orientation is available. Korfhage and Gafni [20] have presented an algorithm to orient directed tori. The orientation problem for tori was also studied by Syrotiuk et al. [22]. There has also been considerable interest in the problem of orienting a ring network [19, 23, 9]. [24] is a recent major work in this area, which studies orientation of cliques, hypercubes, and tori in both anonymous and non-anonymous networks.

Another related area of research is the Sense Of Direction (SoD), which allows processors to communicate efficiently, by exploiting the topological properties of the network algorithmically. [14] provides a formal definition of SoD, and gives a relationship among three factors: the labeling, the topological structure, and the local view that an entity has of the system. [25] shows how election problem on rings, hypercubes, and cliques can be solved more efficiently in presence of the SoD.

1.5 Contributions

In this paper we propose the first self-stabilizing protocols for orienting an arbitrary rooted network. We present two protocols which are self-stabilizing, and assign names and edge labels at the nodes in the network. In the first protocol, called DFTNO, a token is used to assign a name to a node as it passes through the network. We assume an underlying depth-first token circulation protocol maintained on an arbitrary network, for our implementation. Huang and Chen [18] have presented a token circulation protocol for a connected arbitrary network using a non-deterministic
depth-first-search order. Dolev, Israeli, Moran [12] gave a mutual exclusion protocol on a tree network. In a more recent work [10], Datta, et. al., proposed a self-stabilizing depth-first token passing protocol in a connected arbitrary graph. This protocol has better space complexity than the previous algorithms. In this work, the protocol of [10] is used as an underlying protocol maintaining the token circulation. This protocol does not require an underlying tree to be maintained for the token circulation.

A single token is maintained and circulated in the network in a deterministic depth-first search order. The node holding the token has the privilege of naming itself with a unique name. The idea is to prevent two nodes in the network from assigning duplicate names. A token which acts as a counter while traversing the network accomplishes this task of unique node labeling.

In the second protocol, called \( STNO \), we assume that an underlying protocol maintains a spanning tree of the rooted network. The protocol \( STNO \) runs on the spanning tree, but assigns labels to all edges of the network. There exist many self-stabilizing spanning tree construction algorithms (e.g.,[1, 2, 8, 12]). Any of these can be used as the underlying protocol.

A spanning tree algorithm classifies the network nodes as root, leaf and internal nodes. The leaf node initiates the round by calculating its weight. Every internal node and the root use the weight value of all its child nodes to compute its weight. The root then initiates a labeling phase in which each node gets a name from its parent in the spanning tree, and assigns names to its children based on their respective weight values.

Both algorithms presented in this work have the same space complexity of \( O(\Delta \times \log N) \), but the \( STNO \) requires \( O(\Delta \times \log \Delta) \) more bits to maintain the spanning tree. The \( DFTNO \) requires only \( O(\log \Delta) \) more bits for the underlying token circulation.

The remainder of the documentation is organized as follows: Chapter 2 discusses
the preliminary model of the system used in this work, along with some important definitions. Chapter 3 presents the proposed DFTNO algorithm along with the proof of correctness. Chapter 4 presents the proposed STNO algorithm along with the proof of correctness. Finally, conclusions and some future research directions are discussed in Chapter 5.
CHAPTER 2

PRELIMINARIES

In this chapter, we define the distributed systems and programs considered in this work, and state what it means for a protocol to be self-stabilizing. We then define the problem of network orientation in arbitrary rooted networks.

2.1 Self-Stabilizing System

2.1.1 System

A distributed system is an undirected connected graph, $S = (V, E)$, where $V$ is a set of processors ($|V| = N$, $N$ is the upper bound of the number of processors in the network) and $E$ is the set of bidirectional communication links. We consider networks which are asynchronous and rooted, i.e., all processors, except the root are anonymous. We denote the root processor by $r$. A communication link $(p, q)$ exists iff $p$ and $q$ are neighbors. We denote the set of incident edges on a processor $p$ as $E_p$, and the edge connecting processor $p$ with $q$ as $E_{p,q}$. Each processor $p$ maintains its set of neighbors, denoted as $N_p$. We assume that $N_p$ is a constant and is maintained by an underlying protocol. The degree of $p$ is denoted by $\Delta_p$ and is equal to $|N_p|$. The ancestor of processor $p$ ($p \neq r$) is denoted by $A_p$ and is maintained by the underlying protocol (the depth-first token passing protocol in the DFTNO algorithm and the spanning tree protocol in the STNO algorithm).
2.1.2 Programs

The program consists of a set of shared variables (henceforth referred to as variables) and a finite set of actions. A processor can only write to its own variables and can only read its own variables and variables owned by the neighboring processors. So, the variables of p can be accessed by p and its neighbors.

Each action is uniquely identified by a label and is of the following form:

\(< \text{label} > :: \langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle\)

The guard of an action in the program of p is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates zero or more variables of p. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed: the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step. The atomic execution of an action of p is called a step of p.

The state of a processor is defined by the values of its variables. The state of a system is a product of the states of all processors (∈ V). In the sequel, we refer to the state of a processor and system as a (local) state and configuration, respectively. Let a distributed protocol P be a collection of binary transition relations denoted by \(\rightarrow\), on \(C\), the set of all possible configurations of the system. A computation of a protocol P is a maximal sequence of configurations \(e = (\gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...)\), such that for \(i \geq 0\), \(\gamma_i \rightarrow \gamma_{i+1}\) (a single computation step) if \(\gamma_{i+1}\) exists, or \(\gamma_i\) is a terminal configuration. Maximality means that the sequence is either infinite, or it is finite and no action of P is enabled in the final configuration. All computations considered in this paper are assumed to be maximal. During a computation step, one or more processors execute a step and a processor may take at most one step. This execution model is known as the distributed daemon [6]. We use the notation Enable \((A, p, \gamma)\)
to indicate that the guard of the action $A$ is true at processor $p$ in the configuration $\gamma$. A processor $p$ is said to be enabled at $\gamma$ ($\gamma \in C$) if there exists an action $A$ such that $\text{Enable}(A, p, \gamma)$. We assume a weakly fair daemon, meaning that if a processor $p$ is continuously enabled, then $p$ will be eventually chosen by the daemon to execute an action.

The set of computations of a protocol $P$ in system $S$ starting with a particular configuration $\alpha \in C$ is denoted by $E_\alpha$. The set of all possible computations of $P$ in system $S$ is denoted as $E$. A configuration $\beta$ is reachable from $\alpha$, denoted as $\alpha \leadsto \beta$, if there exists a computation $e = (\gamma_0, \gamma_1, \ldots, \gamma_i, \gamma_{i+1}, \ldots) \in E_\alpha (\alpha = \gamma_0)$ such that $\beta = \gamma_i (i \geq 0)$.

2.1.3 Predicates

Let $X$ be a set. $x \vdash P$ means that an element $x \in X$ satisfies the predicate $P$ defined on the set $X$. A predicate is non-empty if there exists at least one element that satisfies the predicate. We define a special predicate $\text{true}$ as follows: for any $x \in X, x \vdash \text{true}$.

2.1.4 Self-Stabilization

We use the following term, attractor in the definition of self-stabilization.

**Definition 2.1.1 (Attractor)** Let $X$ and $Y$ be two predicates of a protocol $P$ defined on $C$ of system $S$. $Y$ is an attractor for $X$ if and only if the following condition is true:

$$\forall \alpha \vdash X : \forall e \in E_\alpha : e = (\gamma_0, \gamma_1, \ldots) :: \exists i \geq 0, \forall j \geq i, \gamma_j \vdash Y.$$ 

We denote this relation as $X \triangleright Y$.

**Definition 2.1.2 (Self-stabilization)** The protocol $P$ is self-stabilizing for the specification $SP_P$ on $E$ iff there exists a predicate $L_P$ (called the legitimacy predicate)
defined on C such that the following conditions hold:

1. \( \forall \alpha \vdash L_{P} : \forall e \in E_{\alpha} :: e \vdash SP_{P} \) (correctness).

2. true \( \vdash L_{P} \) (closure and convergence).

2.2 Chordal Sense of Direction

Both protocols discussed in this paper set up a chordal sense of direction in the un-oriented network. We use a formal definition for the chordal sense of direction as given in [14]. A chordal sense of direction in a connected undirected graph \( S = (V, E) \) with \(|V| = N\), is defined by fixing a cyclic ordering of the nodes and labeling each link by the distance in the above cycle.

Let \( \psi : V \rightarrow V \) be a successor function defining a cyclic ordering of the nodes of \( S \) and let \( \psi^{k}(p) = \psi^{k-1}(\psi(p)) \) for \( k > 0 \). Let \( \delta : V \times V \rightarrow \{0, \ldots, N - 1\} \) be the corresponding distance function, i.e., \( \delta(p, q) \) is the smallest \( k \) such that \( \psi^{k}(p) = q \). The labeling \( \pi \) is a chordal labeling iff, \( \forall (p, q) \in E_{p} :: \pi_{p}(p, q) = \delta(p, q) \).

Note that \( \psi \) is the function defining the cyclic ordering of the nodes, and the different chordal labeling functions arise from different \( \psi \)'s. Further note that, if the link between \( p \) and \( q \) is labeled by \( d \) at node \( p \), it is labeled by \( N - d \) at node \( q \). In other words, the edge label at \( p \) is the inverse modulo \( N \) of the edge label at \( q \). It is assumed that each node is aware of the total number of nodes that constitute the network. Also note that, in order to avoid ambiguity, the edge labels assigned locally should be unique.

2.3 Specification of the Network Orientation Protocol

A labeling of the network is an assignment in every node of different labels from the set \( 1, \ldots, N - 1 \) to the edges incident to that node. An orientation of the network is a labeling, for which each node \( p \) can be assigned a unique name \( \eta_{p} \) from the set
0, 1, ..., N - 1, such that the edge connecting node \( p \) to node \( q \) is labeled \((\eta_p - \eta_q) \mod N\) at node \( p \). Since the node labels are unique, the edge labels computed as above are also locally unique, as both \( \eta_p \) and \( N \) are constants, and \( \eta_q \) is already unique. The real problem hence, is to ensure that the node labels assigned to each node in the network are globally unique. The specific solution of this problem may vary, but once the unique node labels are set up, the edge labels can be computed by the respective nodes in a globally consistent manner.

We define a specification, \( SP_{NO} \) for the Network orientation problem \( (NO) \). We consider a computation \( e \) of the network orientation problem, \( NO \), to satisfy the specification, \( SP_{NO} \), if the following conditions are true:

\( (SP1) \) Every node in the network has a unique name \( \eta_p \) in the range \( 0, \ldots, N - 1 \).

\( (SP2) \forall p \in V : \forall l \in E_{p,q} :: \pi_p[l] = (\eta_p - \eta_q) \mod N. \)

Note that \( SP1 \) (the unique naming of nodes) guarantees that the edge labels assigned satisfying \( SP2 \) are locally oriented, i.e., will be unique locally at the node.
NETWORK ORIENTATION USING DEPTH-FIRST TOKEN PASSING

In this section, we propose a self-stabilizing network orientation algorithm using depth-first token circulation protocol. We first present the data structure used by the algorithm, followed by the algorithm $DFTNO$. We then explain the protocol for setting up the orientation in the network. Finally, we present the correctness proof for $DFTNO$ algorithm. We assume that the self-stabilizing depth-first token circulation protocol of [10] maintains a single token circulating in a deterministic depth-first order.

3.1 Algorithm $DFTNO$

The algorithm depends on an underlying token circulation algorithm. The network is assumed to have exactly one token which is passed from one node to another in a DFS order. No node gets the token more than once during a round. Also, according to the fairness property, every node has to get the token exactly once during a single round of token circulation. It is assumed that there is no deadlock in the network, and a node receiving a token, releases it to another node in finite time.

The circulating token is used as a counter which is incremented every time the token is passed on to an unvisited node. Every node that gets the token, is assigned a name by the token counter, in this case, a number denoting the position of the node in DFS tree of the network that the token traverses. The sequence of events from the
time the token is generated by the root node, to the time when the root node cannot
pass on the token to any other node, constitutes a round of token circulation. Thus
in a round, the token passes from node to node, until it has visited every node in the
network exactly once. At the end of the round, the value of the node is clearly the
total number of nodes in the system. We use this fact to prove the correctness of our
protocol. Having stated the algorithm informally, we now get down to the specific
details of the algorithm.

We denote the descendant relationship of a processor by a variable $D_p$ ($D_p \in
\mathcal{N}_p \cup \bot$). Every processor $p$ has a variable $\eta_p$ (where $\eta_p \in 0, \ldots, N - 1$) which main-
tains the label of the node corresponding to processor $p$. Another variable $\pi_p$ (where
$\forall l \in E_{p,q} : \pi_p[l] = \eta_p - \eta_q \mod N$) holds the edge labels for every incident link
on processor $p$. Every node maintains a variable $\text{Max}$ which contains the maximum
node name that the node is aware of.

Algorithm 3.1.1 ($DFTNO$)  Network Orientation using Depth-First Token Circu-
alation.

\begin{align*}
\text{Macro} & \quad \text{UpdateMax}_p = \{ \text{Max}_p := \text{Max}_{D_p} \} \\
\text{Nodelabel}_p & = \{ \eta_p := 0; \text{Max}_p := 0 \text{ if } (p = r) \} \\
\quad \text{otherwise} \quad \text{Max}_p := \eta_p \\
\text{Edgelabel}_p & = \{ \forall l \in E_{p,q} : \pi_p[l] \neq \eta_p - \eta_q \mod N : \pi_p[l] := \eta_p - \eta_q \mod N \}
\end{align*}

Predicate

$\text{InvalidEdgelabel}(p) \equiv \exists l \in E_{p,q} : \pi_p[l] \neq \eta_p - \eta_q \mod N$

Actions

$\text{Forward}(p) \rightarrow \text{Nodelabel}_p$
$\text{Backtrack}(p) \rightarrow \text{UpdateMax}_p$
$\sim \text{Forward}(p) \wedge \sim \text{Backtrack}(p) \wedge \text{InvalidEdgelabel}(p) \rightarrow \text{Edgelabel}_p$

Algorithm $DFTNO$ is shown as Algorithm 3.1.1. The macro $\text{Nodelabel}_p$ is used
to name the node after consulting the parent for the current maximum node value.
The edge labeling is done only when the node does not have a token and at least one
of the edge labels is inconsistent. The macro $Edgelabel_p$ corrects those edge labels that have been found to be incorrect.

A node is said to hold a token if the following predicate holds:

$$Token(p) \equiv Forward(p) \lor Backtrack(p)$$

$Forward(p)$ is enabled at processor $p$, when it receives a token for the first time from its parent $A_p$. On the other hand, $Backtrack(p)$ is enabled each time the token is backtracked to processor $p$ from its descendant $D_p$. For a more detailed description of $Forward(p)$ and $Backtrack(p)$, refer to [10].

### 3.1.1 Node Labeling

The depth-first token circulation protocol guarantees that every node, during a single round, will have its $Forward(p)$ enabled exactly once, i.e., it will have the token at least once. When a node has a token for the first time, it assigns the next lowest available name, as its node label, after consulting its parent. The node then passes the token to the next node (descendant), if any. Otherwise, it backtracks the token to its parent along with the current maximum value. The process of node labeling is shown in Figure 3.1.1.

Consider the example in Figure 3.1.1. In Step (ii), the root generates the token, names itself as node 0, and notes the current max value as 0. In Step (iii), node b gets the token and names itself as 1 (i.e., $max_{a_p} + 1$). In Step (iv), node d gets the token, names itself 2, and sets its max value to 2. In step (v), node c is named 3 and its max value is set to 3. In step (vi) the token is backtracked to d along with the information that the max value is now 3. So d sets its max value to 3 too. In Step (vii), the node b has $max := 3$ and backtracks the token and max to the root $r$. In step(ix) the root forwards the token to node a, which names itself 4 i.e. $max_{a_p} + 1$. In Step (x) the token is backtracked to the root, which now prepares to initiate the
3.1.2 Assigning edge labels at each processor

Once the node names have been assigned, each node shares its name with its neighbors to enable the edge labeling process. Starting from an arbitrary state where the nodes and edges are labeled arbitrarily, by the time the token completes one full round in the network, both the node labels and edge labels get properly assigned and the network becomes oriented. Every time a node holds the token, it is assigned a legitimate name which is consistent with the global state of the system. Once the node labels are correct, the edge labels are corrected to complete the network orientation.

3.2 Correctness of the Network Orientation Protocol $DFTNO$

As mentioned before, the $DFTNO$ algorithm is written on top of the depth-first token circulation protocol of [10]. We assume that the legitimacy predicate for the
protocol of [10] is denoted by $L_{TC}$. We define the legitimacy predicates, $L_{NL}$ and $L_{NO}$ for the node labeling and edge labeling phase, respectively, as follows:

$$L_{NL} \equiv L_{TC} \land SP1$$
$$L_{NO} \equiv L_{NL} \land SP2$$

### 3.2.1 Correctness of the Node Naming Phase

Every processor, on receiving the Forward token, assigns a name to itself. We can visualize the token as a counter, which is incremented by one every time it is passed from one node to another. The name of the node is assigned as the lowest available value as indicated by the token.

**Lemma 3.2.1** *At the end of node naming phase, every node has a unique $\eta \in (0, \ldots, N - 1)$.*

**Proof:** The proof follows from the macro *NodeLabel* and the fact that there is exactly one token in the network which starts from the root with a node name 0 and traverses all nodes in the network following a consistent order, the depth-first order. □

**Theorem 3.2.1** $L_{TC} \geq L_{NL}$.

**Proof:**
- **Closure:** Follows from the *DFTNO* algorithm.
- **Convergence:** Follows from Lemma 3.2.1. □

### 3.2.2 Correctness of the Edge Labeling Phase

**Lemma 3.2.2** *Edge labels are in local orientation (i.e., no two local edge labels can be identical) and the DFTNO algorithm sets up a chordal sense of direction.*

**Proof:** Follows from the *DFTNO* algorithm and macro *EdgeLabel*. □
Theorem 3.2.2 \( L_{NL} \supset L_{NO} \)

Proof: Closure: Follows from the \( DFTNO \) algorithm.
Convergence: Follows from Theorem 3.2.1 and Lemma 3.2.2. \( \square \)

The following theorem follows from Theorems 3.2.1 and 3.2.2.

Theorem 3.2.3 Algorithm \( DFTNO \) is self-stabilizing.

3.2.3 Space and Time Complexity

The time to stabilize for the \( DFTNO \) algorithm depends on the underlying depth-first token circulation protocol. After the token circulation protocol stabilizes, the \( DFTNO \) takes \( O(n) \) steps to stabilize.

Every node holds three variables: \( \eta_p, \pi_p, \) and \( Max_p \). \( Max_p \) and \( \eta_p \) can hold values in \( 0, \ldots, N - 1 \), which requires \( 2 \times \log N \) bits, where \( N \) is the upper bound of the number of processors in the network. \( \pi_p \) has \( \Delta_p \) values, each in \( 1, \ldots, N - 1 \), requiring \( \Delta_p \times \log N \) bits. Thus, the total space complexity for the \( DFTNO \) algorithm is \( O(\Delta \times \log N) \) bits.
CHAPTER 4

NETWORK ORIENTATION USING SPANNING TREE PROTOCOL

In this chapter we propose a self-stabilizing network orientation protocol that uses an underlying spanning tree protocol to orient the network. We assume that the underlying spanning tree protocol deterministically maintains a spanning tree of the graph.

4.1 Algorithm STNO

The algorithm STNO assumes that a spanning tree of the original graph is maintained such that the role of the nodes in the graph are classified into one of the following three types: root, leaf and internal node. The main idea of the algorithm is to assign unique labels to each node in the network and allow the respective nodes to assign locally unique edge labels.

The unique node labels could be in the range 0, . . . , N − 1. For the STNO, a different labeling scheme is proposed, based on the weights of the subtrees for the respective nodes in the spanning tree. The child nodes pass on the information regarding the number of nodes in the subtree rooted at it, to their parents. The parent node controls the actual labeling of its child nodes, and allocates a range of values to each child based on the information it received from them regarding their weights.

The proper labeling phase begins at the leaf nodes, which notifies its parent that
its weight is 1. Every internal or root node sums all such weight values that it receives from its child nodes, adds one to the value (to include itself) and returns the computed weight to its parent node. This proceeds in a bottom-up towards the root node. Finally, the root node computes the total count of all the nodes in the network as its weight.

Once this information is collected, the root initiates the actual node labeling phase by assigning the value 0 to itself. Then it assigns node labels to each of its child nodes in the spanning tree, leaving appropriate gaps in the node labels for naming nodes in each subtree rooted at its child nodes. Thus, each internal node effectively receives a range of values based on the weight of its subtrees. The node then assigns the lowest value in the range as its node label and repeats the process of allocating non-overlapping ranges of values to each of its child nodes in the spanning tree. This proceeds top-down from the root towards the leaf, in the process all the nodes assigning a node label to themselves. Finally, the leaf nodes receive a single value and assign that value as its node label.

The important criteria here is to ensure that there is no overlap of the range of values that a node receives. This criteria is met, because the allocation is controlled by the parent node which allocates non-overlapping sets of values to each of its child nodes. Hence duplicate node labeling is eliminated and the nodes have unique names at the end of the round. Once the unique labels have been assigned the edge labeling routine assigns locally unique edge labels to all incident edges (tree and non-tree edges) on the nodes. This completes the process of network orientation. With this general idea, the following paragraphs explain the details of the actual implementation of the algorithm.
Algorithm 4.1.2 (STNO)  Network Orientation using Spanning Tree.

Macros

\[
\text{Distribute}_p \quad = \quad \text{given} := \eta_p; \quad \forall q \in D_p \{ \text{Start}_p[q] := \text{given} + 1; \quad \text{given} := \text{given} + \text{Weight}_q \}
\]

\[
\text{Edgelabel}_p \quad = \quad \{ \forall q \in N_p : \forall l \in E_{p,q} : \pi_p[l] \neq \eta_p - \eta_q \mod N \cdot \pi_p[l] := \eta_p - \eta_q \mod N \}
\]

\[
\text{CalcWeight}_p \quad = \quad \{ \forall q \in D_p \quad : \quad \text{Weight}_p := 1 + \sum_q \text{Weight}_q \}
\]

Variables

\[
\text{Start}_p[q] \quad (q \in D_p) \in \{1, \ldots, N - 1\}.
\]

\[
\text{Weight}_p \quad (p \in V) \in \{1, \ldots, N\}.
\]

\[
\eta_p \quad (p \in V) \in \{0, \ldots, N - 1\}.
\]

\[
\pi_p[l] \quad (l \in E_p) \in \{1, \ldots, N - 1\}.
\]

Predicates

\[
\text{InvalidNodelabel}(p) \equiv \{ \eta_p \neq \text{Start}_A[p] \}
\]

\[
\text{InvalidEdgelabel}(p) \equiv \{ \forall q \in N_p : \exists l \in E_{p,q} : \pi_p[l] \neq \eta_p - \eta_q \mod N \}
\]

\[
\text{InvalidWeight}(p) \equiv \{ \forall q \in D_p \quad : \quad \text{Weight}_p \neq 1 + \sum_q \text{Weight}_q \}
\]

Actions

\[
\{ \text{For the internal processors} \}
\]

\[
\begin{align*}
\text{IN} & \quad : \\
\text{InvalidNodelabel}(p) & \quad \rightarrow \eta_p := \text{Start}_A[p]; \\
& \quad \text{Distribute}_p; \\
& \quad \text{Edgelabel}_p;
\end{align*}
\]

\[
\begin{align*}
\text{IE} & \quad : \quad \sim \text{InvalidNodelabel}(p) \land \text{InvalidEdgelabel}(p) \quad \rightarrow \quad \text{Edgelabel}_p; \\
\text{IW} & \quad : \quad \text{InvalidWeight}(p) \quad \rightarrow \quad \text{CalcWeight}_p;
\end{align*}
\]

\[
\{ \text{For the root} \}
\]

\[
\begin{align*}
\text{RN} & \quad : \\
\eta_p & \quad \neq 0 \quad \rightarrow \quad \eta_p := 0; \\
& \quad \text{Distribute}_p; \\
& \quad \text{Edgelabel}_p;
\end{align*}
\]

\[
\begin{align*}
\text{RE} & \quad : \\
\eta_p & \quad := 0 \land \text{InvalidEdgelabel}(p) \quad \rightarrow \quad \text{Edgelabel}_p; \\
\text{RW} & \quad : \\
\text{InvalidWeight}(p) & \quad \rightarrow \quad \text{CalcWeight}_p;
\end{align*}
\]

\[
\{ \text{For the leaf processors} \}
\]

\[
\begin{align*}
\text{LN} & \quad : \\
\text{InvalidNodelabel}(p) & \quad \rightarrow \quad \eta_p := \text{Start}_A[p]; \\
& \quad \text{Edgelabel}_p;
\end{align*}
\]

\[
\begin{align*}
\text{LE} & \quad : \quad \sim \text{InvalidNodelabel}(p) \land \text{InvalidEdgelabel}(p) \quad \rightarrow \quad \text{Edgelabel}_p \\
\text{LW} & \quad : \\
\text{Weight}_p & \quad \neq 1 \quad \rightarrow \quad \text{Weight}_p := 1;
\end{align*}
\]
Algorithm STNO, shown in Algorithm 4.1.2, sets up a network orientation using the weights of the subtrees to label the nodes. Every processor has a variable Start which holds the starting index for each outgoing link from the processor. The variable Weight\_p maintains the size (or count of the nodes) of the subtree rooted at p. \( \eta_p \) holds the assigned name for the node and \( \pi_p \) records the edge labels for each edge incident on p. Each processor p maintains a set of children in the spanning tree denoted by \( D_p \), maintained by the underlying spanning tree protocol. Note that for the node labeling we use only the tree edges that belong to the spanning tree, but all edges, both the tree and non-tree edges are labeled by the edge labeling algorithm.

The predicate InvalidWeight is true if the node detects that its Weight value is incorrect. The node corrects the Weight variable using the CalcWeight macro. The predicate InvalidNodelabel is true when the node detects its name variable (\( \eta \)) to be incorrect. The predicate InvalidEdgelabel is true when there exists at least one edge label in the node that is inconsistent with the definition of the chordal sense of direction. In this case, the node corrects the edge labels using the macro Edgelabel\_p. The macro Distribute\_p assigns the names to all the descendant nodes of node p.

Network orientation is initiated by the leaf processors which set their Weight variable to 1. Each internal node does a summation of the Weight variables of all its descendents and its own to compute its Weight value. This proceeds bottom-up on the tree until the root node computes the Weight value from its subtrees. The root node now initiates the naming phase in which it distributes the names to each of its descendants according to the Weight variable at each descendant. This proceeds top-down, until the parent of the leaf nodes assigns a name to the leaf nodes. Once the leaves are named, every node in the network has a valid node label. Then the node simply reads the assigned name of its neighboring nodes and computes a chordal sense of direction to set up the edge labeling and network orientation.
4.1.1 Node Labeling

The node labeling is shown in Figure 4.1.1. The leaf nodes initiate the node labeling phase, by setting its Weight variable always to 1 because the leaves do not have any descendants (Figure 4.1.1 (i)). The parent of a leaf node, i.e., an internal node, having its IW action enabled, corrects its Weight variable using the CalcWeight macro (Figure 4.1.1(ii)). This propagates to the root, which does a similar computation to correct its Weight variable (Figure 4.1.1(iii)). In this configuration, for each node \( p \in V \), the Weight\(_p\) reflects the actual weight of the subtree rooted at node \( p \). Similar to the leaf nodes, the root writes the value 0 to its \( \eta \) variable, thus initiating the actual node labeling procedure, which propagates top-down on the tree (Figure 4.1.1 (iv)). The root assigns the number 0 to itself, and distributes the remainder of the set \( \{0, \ldots, N - 1\} \) over its children, where each child receives as many numbers as there are nodes in its subtree (Figure 4.1.1 (v)). Each node, upon receipt of an interval of numbers from its parent, assigns itself the smallest number and distributes the remainder of the interval over its children in a similar manner. Thus when all the leaves have been assigned a name, each node in the network has a unique \( \eta \in \{0, \ldots, N - 1\} \) (Figure 4.1.1 (vi)).

4.1.2 Edge Labeling

Once the node names have been assigned, each node shares its name with its neighbors to enable the edge labeling. In this particular algorithm, as soon as the node has an assigned name, and detects an invalid edge label, the node computes an edge label for every incident edge on the node that had an invalid label. Thus, once the node labels are consistent with the global state, the edge labels are corrected to complete the network orientation.
Figure 4.1.1: Assigning names on the spanning tree of the network.

4.2 Correctness of the Network Orientation protocol \textit{STNO}

Algorithm \textit{STNO} is written on top of a self-stabilizing spanning tree protocol. We denote the legitimacy predicate of the underlying spanning tree protocol by $L_{ST}$. The legitimacy predicates for the node labeling and edge labeling phase, $L_{NL}$ and $L_{NO}$, respectively, are as follows:

\[
L_{NL} \equiv L_{ST} \wedge SP1 \\
L_{NO} \equiv L_{NL} \wedge SP2
\]

4.2.1 Correctness of the Node Naming Phase

Every processor consults its descendants to calculate its weight and gets a name assigned by the parent node.

\textbf{Lemma 4.2.1} All nodes have a unique legitimate name in $O(h)$ steps.
Proof: It takes $O(h)$ steps for the root to compute its Weight (the size of the tree). The variable Weight at each node now holds the true weight of the subtree rooted at that node. Starting from this configuration, it takes another $O(h)$ steps to assign the correct names to the nodes from the root to the leaves. □

**Theorem 4.2.1** $L_{ST} \succ L_{NL}$.


4.2.2 Correctness of the Edge Labeling Phase

We get the following result as in Section 3.2.2.

**Theorem 4.2.2** $L_{NL} \succ L_{NO}$

The following theorem follows from Theorems 4.2.1 and 4.2.2.

**Theorem 4.2.3** Algorithm STNO is self-stabilizing.

4.2.3 Space and Time Complexity

The time to stabilize the STNO algorithm depends on the underlying spanning tree construction protocol. After the spanning tree protocol stabilizes, the STNO takes another $O(h)$ steps to stabilize.

Weight$_p$ and $\eta_p$, each require log $N$ bits, while Start$_p$ and $\pi_p$ each require $\Delta_p \times \log N$ bits. Hence the total space complexity for the STNO protocol is $O(\Delta \times \log N)$ bits.
CHAPTER 5

CONCLUSION

We presented two self-stabilizing algorithms for the network orientation problem. Both algorithms work on an asynchronous, anonymous, and arbitrary network, thus being applicable for a wide range of network structures. $DFTNO$ works on an arbitrary network and assumes an underlying depth-first token circulation protocol which runs using a fair daemon. The algorithm takes $O(n)$ steps to stabilize after the underlying token circulation protocol stabilizes. $STNO$, on the other hand, requires an underlying protocol which maintains a spanning tree of the network with an unfair daemon. The stabilization time for the $STNO$ is $O(h)$ steps after the spanning tree protocol has stabilized.

The idea behind both algorithms is to assign unique names to each node in the network followed by a simple computation of the locally unique edge labels at the respective nodes. For this purpose, the first algorithm uses the token principle whereas the second algorithm uses the weight information of the nodes in the underlying spanning tree maintained on the network. An interesting observation here is that if the spanning tree maintained in the $STNO$ is a DFS tree of the graph, then the naming could be similar for both algorithms, provided the respective ordering at individual nodes is the same.

Both the $DFTNO$ and $STNO$ algorithms require the same amount of space which is $O(\Delta \times \log N)$ bits. But, the $STNO$ is required to maintain the descendants in the
spanning tree which requires an extra space of $O(\Delta \times \log \Delta)$ bits. The DFTNO, on the other hand, requires only $O(\log \Delta)$ bits as it does not maintain the spanning tree.

Self-stabilizing network orientation has far-reaching implications in networks. An important extension of the network orientation, which has become an attractive area of research in the recent years, is the Sense Of Direction (SoD). An important property of SoD is that it allows processors to refer to the other processors by locally unique names, which are derived from the shortest path between the processors and can be translated from one processor to the other. It was shown by Flocchini et al. [FMS95] that in arbitrary graphs, any sense of direction has a dramatic effect on the complexity of several important distributed problems like broadcast, depth-first traversal, leader election, and spanning tree construction. A future topic of research would be to design self-stabilizing SoD algorithms, which maintain a sense of direction in the network.
BIBLIOGRAPHY


VITA

Graduate College
University of Nevada, Las Vegas

Shivashankar Gurumurthy

Local Address:
4236 Grove Circle #1
Las Vegas, NV 89119

Home Address:
# 20 North Mada Street
Nungambakkam
Madras 600034, India.

Degrees:
Bachelor of Engineering, Computer Science and Engineering, 1997
University of Madras, Madras, India

Thesis Title:
Self-stabilizing Network Orientation Algorithms
in arbitrary rooted networks.

Thesis Examination Committee:
Chairperson, Dr. Ajoy Kumar Datta, Ph.D.
Committee Member, Dr. John Minor, Ph.D.
Committee Member, Dr. Kazem Taghva, Ph.D.
Graduate Faculty Representative, Dr. Ashok Iyer, Ph.D.