Self-Stabilizing Depth First Search

Zeev COLLIN* Shlomi DOLEV†

March 14, 1994

Abstract

A distributed algorithm is *self-stabilizing* if it can be started from *any possible* global state. Once started, the algorithm converges to a consistent global state by itself. This paper presents a distributed self-stabilizing Depth First Search (DFS) spanning tree algorithm, whose output is a DFS spanning tree of the communication graph, kept in a distributed fashion.

keywords: distributed computing, fault tolerance.

*Department of Computer Science, The Technion, Haifa 32000, Israel.

†Department of Computer Science, Texas A&M University, College Station, TX 77843, Contact Author: shlomi@cs.tamu.edu. Supported in part by TAMU Engineering Excellence funds and NSF Presidential Young Investigator Award CCR-91-58478.

1 Introduction

A distributed algorithm is *self-stabilizing*, if it can be started from *any possible* global state and once started, the algorithm regains consistency by itself. The self-stabilization property is very useful for systems in which processors may crash and then spontaneously recover in an arbitrary state. When the intermediate period between one recovery and the following crash is long enough, the algorithm stabilizes.

The study of self-stabilizing algorithms started with the fundamental paper of Dijkstra, [Dij-74]. Recently there has been great interest in the subject e.g., [La-86], [BP-88], [KP-90], [DIM-90], [DIM-91a]. Finding a DFS spanning tree is one of the common tasks in the literature of distributed algorithms e.g., [Aw-85], [Ch-83]. In this paper we present a distributed self-stabilizing Depth First Search (DFS) spanning tree algorithm [Ev-79], the output of which is a DFS spanning tree of the communication graph, kept in a distributed fashion. Distributed algorithms for constructing a DFS spanning tree are widely used e.g., [CD-90], [CDK-91a] and [CDK-91b] quote a preliminary draft of this paper and use our self-stabilizing DFS algorithm. Independently, in [He-91] a self-stabilizing DFS algorithm is presented. Our algorithm uses a refined model with respect to the one of [He-91][1].

Our DFS algorithm uses lexicographic order among paths (as described in the sequel). This technique could be easily extended to obtain other self-stabilizing algorithms for different types of trees.

This paper is organized as follows: Section 2 briefly presents the distributed system and the requirements from self-stabilizing algorithms. The self-stabilizing DFS spanning tree algorithm is given in Section 3. Finally, some concluding remarks are given in Section 4.

2 Self-Stabilizing Distributed System

We adopt the model introduced in [DIM-90]. A *distributed system* consists of n processors denoted by $P_1, P_2, \cdots, P_n$. Processor P_1 is called *special*. All other processors are called *regular*. Regular processors have no distinct identities, the subscripts $2, ..., n$ are used for ease of notation only. Each processor can communicate with some other processors, called its *neighbors*. The system's *communication graph* is the graph formed by representing each processor as a node and connecting every pair of neighbors by an edge. Communication between neighbors is carried out using *shared communication registers* (called *registers* throughout this work). Each register is serializable with respect to read and write operations.

Any processor, P_i, writes in one register, r_i, and may read from the register of any of its neighbors. Thus, each pair of neighbors, P_i and P_j, are connected by an *edge* $e = (P_i, P_j)$ that supports two-way communication. Each processor, P_i, orders its edges by some arbitrary

[1] In [He-91] it is assumed that during the time a processor reads, makes local computations and writes, no other processor is active.

ordering α_i. For any edge $e = (P_i, P_j)$ let $\alpha_i(j)$ ($\alpha_j(i)$, respectively) be the *edge index* of e according to α_i (α_j, respectively). We assume that for every processor P_i and any edge $e = (P_i, P_j)$, P_i knows the value of $\alpha_j(i)$[2].

We consider a processor and its register to be a single entity, thus the state of a processor fully describes the value stored in its register. Denote by S_i the set of states of P_i. A *configuration* $c \in (S_1 \times S_2 \times \cdots S_n)$ of the system is a vector of states, one for each processor. Processor activity is managed by a scheduler. At any given configuration the scheduler activates a single processor which executes a single *atomic step*. An atomic step of a processor consists of an internal computation followed by either a read or write action, but not both. An execution of the system is a finite or infinite sequence of configurations $E = (c_1, c_2, \cdots)$, such that for $i = 1, 2, \cdots$ the configuration c_{i+1} is reached from c_i by a single atomic step of some processor. A *fair execution* is an infinite execution in which every processor executes atomic steps infinitely often.

A *task* is defined by a set of executions which are called *legal executions*. An algorithm is *self-stabilizing* for a specific task if any fair execution of the algorithm has a suffix that is within the set of legal executions of that task.

Although the description of the system is similar to [DIM-90] the protocols we present work also in message passing systems by the use of a self-stabilizing simulation of shared memory described in (the full version of) [DIM-91a].

3 Self-Stabilizing DFS Spanning Tree

This section presents a self-stabilizing algorithm for constructing a DFS spanning tree. The root of the tree is the special processor P_1.

We use the following notations to define the environment of a processor in a DFS spanning tree. Each processor, P_i, has (at most) one adjacent processor, designated as its *parent* in the tree, and a set of *child processors*. A processor that resides along the path from the root to P_i in the DFS spanning tree is an *ancestor* of P_i, while the *descendants* of P_i are the processors such that P_i is their ancestor.

The adjacent edges of every processor P are categorized as follows:

1. *tree-edges* – the edges that belong to the DFS spanning tree.
   (a) *incoming edge* – the edge that connects P to its parent. Every non-root processor has one such edge and the root has none.
   (b) *outgoing edges* – the edges that connect P to its children.
2. *non tree-edges* – edges that do not belong to the DFS spanning tree.

[2]This is an abstraction of the model used in [DIM-90] where it is assumed that P_i writes in a register r_{ij} to each of its neighbors P_j. In such a model, P_i may write $\alpha_i(j)$ in r_{ij}. Once P_j reads r_{ij} it knows the value of $\alpha_i(j)$.

(a) *backward edges* – the edges that connect P with Q, where Q is an ancestor of P (note that the edge (P, Q) is not a DFS tree-edge, hence Q is not P's father).

(b) *forward edges* – the edges that connect P with R, where R is a descendant of P. (Similarly, (P, R) is not a DFS tree-edge, hence R is not P's child).

The output of a distributed algorithm is called a *DFS-marked* graph if it defines a DFS spanning tree and if every processor in the system can identify the category of each of its adjacent edges with respect to this DFS spanning tree.

The output of our DFS spanning tree algorithm, after the network has converged, is a DFS-marked graph maintained in a distributed fashion. In the next subsection we will formally define how the DFS-marked graph is marked upon the system graph.

3.1 DFS Representation

Consider a graph with edge indices α_i for each processor P_i (as defined in Section 2). We define the DFS tree of such a graph using the edge indices of the processors in the following way.

Definition 3.1 *The* first DFS tree *of a graph is the DFS spanning tree that is obtained by performing a (centralized) DFS algorithm [Ev-79], visiting the adjacent edges of every processor P_i in the order induced by α_i.*

For a given graph with a special processor P_1 (that plays the role of the root), each simple (loopless) path from P_1 is denoted by $\perp$ followed by a sequence of the indices of the (outgoing) edges that define the path (e.g. $(\perp, 5, 3)$ defines the path that begins with the root, continues through the edge with index 5 in α_1, reaches a processor, say Q, through this edge and ends with a processor, say R, that is reached through the third edge of Q). A processor Q is associated with the set of simple paths from the root to Q. We define a lexicographical order "$\prec$", over the representation of the above set of paths, where $\perp$ is the minimal character[3]. The smallest path (w.r.t "$\prec$") in the set of the simple paths of a processor P_i is called the *first path* of P_i, and is denoted by f_i. The father of P_i in f_i is the last processor before P_i in the path defined by f_i.

Observation 3.1 *The graph obtained by marking, for every $1 < i \leq n$, the edge between P_i and P_i's father in f_i is the first DFS tree.*

3.2 DFS Spanning Tree Algorithm

Informally, the DFS spanning tree algorithm works as follows: the register of any processor P_i consists of a *path* field denoted by $path_i$. During the execution of the algorithm the special processor P_1 repeatedly writes the path $\perp$ in $path_1$. Any regular processor P_i repeatedly reads

[3] E.g. $(\perp,1) \prec (\perp,1,1) \prec (\perp,2) \prec (1)$.

the registers of its neighbors. The path $path_j$, read by P_i from the neighbor P_j, derives a path for P_i simply by concatenating $path_j \circ \alpha_j(i)$ (recall that the value of $\alpha_j(i)$ is known to P_i). P_i chooses its path to be the minimal path among the paths derived from its neighbors' paths.

We assume that for any processor P_i, $path_i$ contains either a sequence of at most N edge indices or a sequence that begins with a $\perp$ followed by at most $N \perp 1$ edge indices, where $N \geq n$ is an upper bound on the number of processors in the graph. Note that by the above definition $path_i$ may describe the longest simple path that is possible in the graph. Assigning a too long sequence to a *path* field causes an overflow that results in losing the most significant items of the sequence[4]. This is a generalization of the common overflow bit mechanism, which is used to simplify the presentation and the correctness proof of the algorithm. The notation $|x|_w$ refers to the sequence of the w least significant items of x. Figure 1 presents the algorithm for a root processor and a non-root processor that has δ neighbors $Q_1, Q_2, \cdots, Q_\delta$.

root P_1**:**
1. do forever
2. $\quad path_1 := \perp$
3. od

non-root P_i**:**
1. do forever
2. $\quad$ for $j := 1$ to δ do $read\_path_j := read(path_j)$ od
3. $\quad write\ path_i := \min\{|read\_path_j \circ \alpha_j(i)|_N$, such that $1 \leq j \leq \delta\}$
4. od

Figure 1: DFS Algorithm

A processor P_i (P_j) can indicate that it is a parent (child) of P_j (P_i) when $path_i = path_j \circ \alpha_i(j)$. We consider the edges of the tree to be directed from parent toward the child. We prove that in this algorithm the paths of all the processors (namely, the values of the $path_i$ variables) eventually converge to the first paths and hence, define the first DFS tree.

Theorem 3.2 *Every fair execution has a suffix such that for every configuration in that suffix and for every processor* P_i $path_i = f_i$.

Proof: We prove the theorem by induction on the depth of the processor in the first DFS tree. We claim that every fair execution has a suffix such that for every configuration and for every processor P_i at depth d or less, $path_i = f_i$. Then we prove the above for $d + 1$.
Base: $d = 0$. The root assigns the value $\perp$ to $path_1$, which is f_1 by definition.

[4]E.g. assigning the value $(\perp, x_1, x_2, \cdots, x_N)$ in $path_i$ results in $path_i = (x_1, x_2, \cdots, x_N)$.

Step: Assume that there is a suffix of the execution during which the *path* fields of all processors whose depth in the first DFS tree is smaller than or equal to d have converged to their first path. Consider a processor P_i whose depth in the first DFS tree is $d+1$. We prove that there is a suffix of the execution in which the following assertion is satisfied: the *path* fields of all the processors in the subtree, T_i, rooted at P_i in the first DFS tree are $\succeq f_i$.

For a given configuration c, let the *minimal read value* in T_i be the minimal value (w.r.t. $\succ$) of the *read_path* variables in T_i. Let P_f be the father of P_i according to f_i. We now show that there exists a suffix of the execution in which for every configuration the minimal read value is $\succeq f_f$.

By the induction assumption all the ancestors of P_i have their first path in their *path* variable. Thus, any edge that is connected from an ancestor of P_i to a descendant of P_i (note that all the processors in T_i are descendant of P_i) derives a *path* value that is greater than f_f. Thus, after the first read through each of those edges the *read_path* that corresponds to those reads is greater than f_f. Let c' be the configuration in which all the above read variables are greater than f_f. Let $min(c)$ be the minimal value of a *path* stored in either the *path* field or the *read_path* variable of a processor of T_i in c. We claim that if $f_f \succ min(c')$ then there is a suffix of the execution (that follows c') such that in any of its configurations, c'', $min(c'') \succ min(c')$.

During the execution of the algorithm whenever a processor, P, assigns a value to its *path* that begins with $\perp$ this value is greater than at least one *read_path* variable of P, thus greater than $min(c')$ (note that since $f_f \succ min(c')$, $min(c')$ begins with $\perp$). Thus, eventually every processor in T_i writes a value that is greater than $min(c')$ in its *path*. Once every processor reads the new values, c'' is reached.

Repeated application of the above claim yields that there exists a suffix of the execution in which all the *paths* in T_i are not smaller than f_f. Following the first time, during this suffix, when P_i reads the *paths* of its neighbors, P_i finds that the *path* of its parent in the first DFS tree yields the minimal *path* and assigns $path_i = f_i$. ■

The output of the above algorithm is indeed a DFS-marked graph. Every processor P_i after reading the stabilized *paths* of its neighbors, is able to identify the type of any of its incident edges relative to the first DFS tree. The edge $e = (P_i, P_j)$ is:

1. an *incoming edge* – iff $path_i = path_j \circ \alpha_j(i)$.
2. an *outgoing edge* – iff $path_j = path_i \circ \alpha_i(j)$.
3. a *backwards edge* – iff $path_j$ is a prefix of $path_i$ and e is not an incoming edge.
4. a *forward edge* – iff $path_i$ is a prefix of $path_j$ and e is not an outgoing edge.

4 Concluding Remarks

In this paper we further extend the research on self-stabilizing graph algorithms. We present a self-stabilizing algorithm for a basic distributed task, namely, constructing a DFS tree. Our

DFS algorithm uses a lexicographic order relation on the path representation. We believe that this scheme is general and could be used to design self-stabilizing algorithms for other graph algorithm tasks using different order relations (e.g., BFS could be constructed using a function on the length of the path representation [DIM-90]).

The memory requirement for our algorithm is $O(n \log \Delta)$ where Δ is an upper bound on the degree of a node. The time complexity is measured in *rounds*. The first round of an execution, E, is finished immediately after each processor has executed one atomic step; the second round is finished after each processor has executed one atomic step following the termination of the first round, and so on and so forth. The time complexity of our algorithm is $O(dn\Delta)$ rounds, where d is the diameter of the communication graph.

Self-stabilizing leader election protocols (e.g. [AG-90], [DIM-91a]) can be composed with our algorithm using the fair protocol composition method of [DIM-90]. The resulting composition is a self-stabilizing DFS algorithm for a system of processors with distinct identifiers (when composed with [AG-90]) or a system with anonymous processors (when composed with the randomized protocol of [DIM-91a]).

Acknowledgment: Many thanks to Jennifer L. Welch and the anonymous referees for a careful reading of a preliminary version of this paper.

References

[AG-90] A. Arora and M. Gouda, "Distributed Reset", *Proc. of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 316-331, 1990.

[Aw-85] B. Awerbuch, "A new distributed depth-first-search algorithm", *Information Processing Letters*, 20:147-150, 1985.

[BP-88] J.E. Burns and J. Pachl, "Uniform Self-Stabilizing Rings", *Aegean Workshop On Computing, 1988, Lecture Notes in Computer Science* 319, pp. 391-400.

[CD-90] Z. Collin and R. Dechter. "A distributed solution to the network consistency problem", *Proc. of the 5th Intl. Symp. on Methodologies for Intelligent Systems*, pp. 242-251, 1990.

[CDK-91a] Z. Collin, R. Dechter and S. Katz. "On the feasibility constraint satisfaction", *Proc. of IJCAI-91*, Sydney, Australia, 1991.

[CDK-91b] Z. Collin, R. Dechter and S. Katz. "Self-Stabilizing Distributed Constraint Satisfaction", TR-709 1991, Technion, Israel Institute of Technology, Dept. of Computer Science.

[Ch-83] T. Cheung, Graph traversal techniques and the maximum flow problem in distributed computation, *IEEE Trans. Software Engineering* SE-9 (4) (1983) 504-512.

[Dij-74] E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control", *Communications of the ACM* 17,11 (1974), pp. 643-644.

[DIM-90] S. Dolev, A. Israeli and S. Moran, "Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity", *Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computation*, Montreal, August 1990, pp. 103-117.

[DIM-91a] S. Dolev, A. Israeli and S. Moran, "Resource Bounds for Self-Stabilizing Message Driven Protocols", *Proc. of the Tenth Annual ACM Symposium on Principles of Distributed Computation*, Montreal, August 1991, pp. 281-294.

[DIM-91b] S. Dolev, A. Israeli and S. Moran, "Uniform Dynamic Self-Stabilizing Leader Election", *Lecture Notes in Computer Science 579: Distributed Algorithms, Proc. of the 5th International Workshop on Distributed Algorithms*, Delphi, Greece, October 1991, S. Toueg, P.G. Spirakis and L. Kirousis, Editors, pp. 163-180, Springer Verlag, 1992.

[Ev-79] S. Even, *Graph Algorithms*. Computer Science Press, Maryland, 1979.

[He-91] T. Herman, "Adaptivity through Distributed Convergence", Ph.D. dissertation, University of Texas at Austin, 1991.

[KP-90] S. Katz and K. J. Perry, "Self-stabilizing extensions for message-passing systems", *Proc. of the Ninth Annual ACM Symposium on Principles of Distributed Computation*, Montreal, August 1990, pp. 91-101.

[La-86] L. Lamport, "The Mutual Exclusion Problem: Part II - Statement and Solutions", *Journal of the Association for Computing Machinery* , Vol. 33 No. 2 (1986), pp. 327-348.