A Dynamic Self-Stabilizing Algorithm for Finding Strongly Connected Components *

Mehmet Hakan Karaata
Fawaz Al Anzi

Department of Electrical and Computer Engineering
Kuwait University
P.O. Box 5969 Safat-13060 Kuwait
E-mail: {karaata, alanzif}@eng.kuniv.edu.kw

January 27, 1999

Abstract

A self-stabilizing algorithm is presented in this paper that finds the strongly connected components of a connected directed graph on a distributed or network model of computation after $O(n)$ rounds. The algorithm is resilient to transient faults and does not require initialization. In addition, the proposed algorithm can withstand topology changes in the form of addition and/or removal of edges and vertices. A correctness proof of the algorithm is also provided. The paper concludes with some final remarks.

Key words: Directed graphs, distributed systems, self-stabilization, strongly connected components.

1 Introduction

Consider a connected directed graph $G = (V, E)$, where V is the set of n nodes, and E is the set of e edges. Without loss of generality, we assume V to be $\{1, 2, \cdots, n\}$. A *path* from node v_0 to node v_k in G is an alternating sequence

$$(v_0, (v_0, v_1), v_1, (v_1, v_2), \cdots, v_{k-1}, (v_{k-1}, v_k), v_k)$$

of nodes and edges that belong to V and E, respectively. Two nodes v and w in G are *path equivalent* if there is a path from v to w and a path from w to v. Path equivalence partitions V into maximal disjoint sets of path equivalent nodes. These sets are called

*A preliminaly version of this paper is to be presented at PODC'99 as a brief anouncement the *strongly connected components* of the graph. Finding strongly connected components is used in data flow analysis [16], and for finding transitive closures, disjoint paths and knots [14, 13]. In distributed systems, topological properties such as strongly connected components and disjoint paths are utilized for reliable communication, whereas knots are used for deadlock detection.

In [18], Tarjan presents an asymptotically optimal sequential algorithm for finding the strongly connected components, with time complexity $O(n+e)$. There also exist two other efficient sequential algorithm for finding the strongly connected components due to Nuutila and Soisalon-Suininen [14]. These algorithms are improved versions of Tarjan's algorithm. In addition to these algorithms, another linear algorithm is presented in [16] by Sharir. It requires one depth-first traversal of the input graph and another traversal of the graph obtained by reversing the edges of the input graph. More recently, in [10], Jiang proposed a new linear algorithm based on a traversal strategy that is the combination of depth-first and breadth-first traversal. In addition a distributed algorithm for finding strongly connected components is proposed by Mohanty and Bhattacharjee in [13]. All the existing algorithms use depth-first traversal which can be done in linear time, but can be expensive for large graphs. In addition, depth first traversal is sequential in nature, therefore, it is not appropriate for distributed computation. Our algorithm is the first algorithm not based on depth-first traversal. There are a number of sequential and distributed algorithms, however, to the best of our knowledge, no self-stabilizing algorithm for finding the strongly connected components is available in the literature.

In this paper, we present a dynamic self-stabilizing distributed algorithm for finding the strongly connected components. The proposed algorithm, referred to as SCC, can withstand *transient failures* and topology changes. We view a fault that perturbs the state of the system but not the program as a transient fault. The algorithm makes $O(n)$ rounds to identify all strongly connected components in a graph, where n is the number of nodes in G.

Due to the dynamic nature of distributed systems, protocols dealing with topology changes in the form of node and edge addition/removal in a transparent manner are desirable. Such protocols referred to as *dynamic protocols* provide a higher degree of fault tolerance. Another desirable property of fault tolerant distributed systems is the property of self-stabilization. A self-stabilizing system guarantees that, regardless of the current configuration, the system reaches a legal state in a bounded number of steps and the system state remains legal thereafter. Straightforward adaptation of any of the existing sequential algorithms for finding the strongly connected components to the self-stabilizing version of the algorithm seem to be difficult, since the self-stabilizing algorithms are inherently fault tolerant and have to satisfy additional properties.

Since the introduction of self-stabilization in distributed systems by Dijkstra [7], self-stabilizing algorithms for many fundamental problems in distributed systems have been proposed. For example, self-stabilizing mutual exclusion algorithms for various classes of networks have been presented in [3, 5, 7]; self-stabilizing algorithms for electing a leader appear in [9, 12]; self-stabilizing algorithms for various graph theoretic problems are presented in [4, 8, 17]; general techniques for constructing self-stabilizing algorithms are dealt with in [1, 2, 11]. Self-stabilizing algorithms are able to withstand *transient failures.* In addition, many self-stabilizing algorithms are capable of dealing with addition and deletion of vertices or edges in a transparent manner. A detailed survey on self-stabilization is available due to Schneider [15].

The paper is organized as follows. In Section 2, we present our model of computation. Section 3 contains some definitions and provides the basis of the algorithm. Section 4 presents the self-stabilizing algorithm. In Section 5, we provide a correctness proof of the proposed algorithm, a proof of the time complexity bound, and a proof of self-stabilization of the algorithm. We conclude the paper in Section 6 with some final remarks.

2 Computational Model

Let $G = (V, E)$ be an arbitrary graph with vertex set V and edge set E, where $|V| = n$. We assume that each vertex of G is a process, also called node. Each node maintains a set of local variables whose values can be updated by the node after inspecting its own local variables and the local variables of its neighbors. The program of a node i can be expressed as

$$* \; [\; G[1] \rightarrow M[1] \;\square\; G[2] \rightarrow M[2] \;\square\; \cdots \;\square\; G[k] \rightarrow M[k] \;],$$

where

- each *guard* $G[\;]$ is a boolean function of the variables of node i and the variables of its neighboring nodes;
- each *move* $M[\;]$ updates the variables of node i; and
- $*[S]$ corresponds to the repeated execution of the statement S until all guards are false.
- $\square$ is called the nondeterminism symbol. One of the guarded actions separated by them is selected nonderterministically in each iteration.

Note that each move denotes an atomic action. If a guard of node i evaluates to true, we say that the guard and node i are enabled. Furthermore, if two or more guards of the same node or of different nodes are enabled at the same time, then the *scheduler* arbitrarily selects a subset of the enabled guards and allows the execution of the corresponding moves to be completed. In the case of the *central scheduler*, the size of the subset is one and the execution of a move is completed before any guard is reevaluated; whereas in the case of the *distributed scheduler*, guards can be eveluated and actions can be taken at any time. We assume a central scheduler that is weakly fair for the sake of simplifying the proofs.

The *state* of a node is composed of the set of variables at the node. The *system state* is the cartesian product of the states of the nodes in the system. A *state* of the system is an element of the state space. The system state before the system is started is referred to as the *initial state*.

3 Basis of the Algorithm

The self-stabilizing algorithm for finding strongly connected components is based on the following important observation. This is presented in the form of the following lemma given in [6], whose proof is straightforward and hence omitted.

Lemma 1 *Let $G = (V, E)$ be a connected directed graph. Two nodes $i, j \in V$ belong to the same strongly connected component iff there exists a $i - j$ path in G and a $j - i$ path in G.*

A node i is said to be a predecessor of another node j, with respect to G, if there exists a directed path $i - j$ in G; alternatively, j is said to be a successor of i. Informally, the approach used by the algorithm can be described as follows. Each node $i \in V$ finds the set of its predecessors and successors. By the above lemma, we know that for each node i, the intersection of the set of predecessors and successors of i gives us the nodes in the strongly connected components containing i. Although the approach is rather simple, the challenge lies in making the algorithm dynamic and self-stabilizing. That is, no initialization is required, and topology and transient failures are dealt with in a transparent manner.

Prior to formally describing the SCC algorithm, we first present the basis of the algorithm . We associate a unique id with each node in G. In addition, each nodal process contains two variables P and S called P-set and S-set of process i and denoted by $P(i)$ and $S(i)$, respectively. Each contains a set of at most cn tuples of the form (id, d), where $id \in V$, c is a positive integer constant and d is a positive integer. We assume that each set cannot contain more than one tuple with the same node id. Upon termination, $P(i)$ contains the set of predecessors of node i. Similarly, upon termination, $S(i)$ contains the set of successors of node i. If (j, d) is in $P(i)$ after termination, then j is a predecessor of node i, and d denotes the length of the shortest directed path from node j to node i. Analogously, if (j, d) is in $S(i)$ after termination, then j is a successor of node i, and d denotes the length of the shortest directed path from node i to node j.

P-sets and S-sets are computed for nodes in V as follows. For each node $i \in V$, when node i discovers that its P-set does not contain i, it includes i in its P-set. Then, node id i propagates to all the successors of i, i.e., each successor of i includes i in its P-set. Analogously, each predecessor of node i includes i in its S-set. After this process is completed, upon termination of the algorithm, each node $i \in V$ identifies the set of nodes belonging to the strongly connected component node i is in as the intersection of the node id's in sets $P(i)$ and $S(i)$.

The above discussion on our approach to find strongly connected components does not include the details that make the algorithm distributed and self-stabilizing. We now describe these issues.

For each node $i \in V$, first, node i includes $(i, 0)$ in $P(i)$, then node i's children (immediate successors) include $(i, 1)$ in their P-sets and consequently node i's children's children include $(i, 2)$ in their P-sets, and so on. In this manner, node id i propagates to the successors of i. When these propagations are completed for node i, each successor of i contains i in its P-set. Such propagations originating from node i carrying node id i to its successors are referred to as *desirable propagations*. Observe that, in each step of a desirable propagation, when node id i is included in $P(j)$, the distance value associated with id i denotes the distance of node j from node i. In addition, tuple (i, d) included in $P(j)$ due to arbitrary initialization such that i is not a predecessor at distance d from j can also initiate propagations of node id i. Such propagations not originating from node i are referred to as *undesirable propagations*. In each step of an undesirable propagation, when node id i is included in $P(j)$, the distance value associated with id i does not denote the distance of node j from node i. Note that i may not be a valid node id.

Undesirable propagations need to be stopped and the tuples included in the sets due to these propagations and arbitrary initialization must be removed from the sets in order to make the approach self-stabilizing. It is easy to observe that when id i is included in the P-sets of all its successors and the distance values associated with these id's denote the distances from node i, propagations of i terminate. Whereas, an undesirable propagation may reach its origin which in turn trigger another undesirable propagation, and so on. Therefore, an undesirable propagation may continue indefinitely. Furthermore, if (i, d) is included in $P(j)$ by an undesirable propagation, then d can exceed n, where n is the number of nodes in the system.

In order to prevent an undesirable propagation from continuing indefinitely, we stop a propagation when the distance value carried with the propagation exceeds N, where N is a sufficiently larger number than n and on the order of n. That is, it is ensured that no tuple with a distance value more than N is included in any P-set. Since the distance values cannot be incremented further after reaching N, undesired propagations eventually

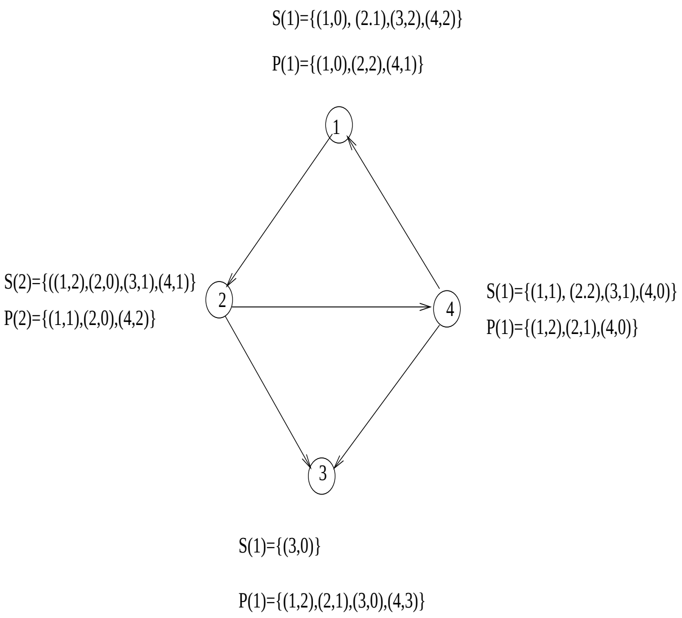

Figure 1: The state of a system after termination

stop. After an undesired propagation stops, the tuples included in the sets due to this propogation are removed as follows. If (i, d) is in $P(j)$, where $d \geq N$, (i, d) is removed from $P(j)$. Also, we remove a tuple (i, d) from $P(j)$, if j does not have a predecessor k such that $(i, d-1) \in P(k)$. It can readily be seen that after an undesired propagation is stopped, there exists at least one node j such that $(i, d) \in P(j)$, however, j does not have a predecessor k such that $(i, d-1) \in P(k)$. It is also easy to see that this continues to hold until all the tuples included by the undesirable propagation are removed. Therefore, all tuples with incorrect distance values are eventually removed from the P-sets.

Observe that S-sets are computed analogous to the way P-sets are computed. Therefore, the description of the approach used in computing the S-sets is omitted.

The above concepts are illustrated with the help of an example in Figure 1. In the figure, a directed graph on four nodes is shown where each directed edge is shown by an arrow. In addition, contents of P-sets and S-sets associated with each node after termination are shown. The graph contains a single strongly connected component. Nodes $1, 2, 4$ and edges $(1, 2), (2, 4)$ and $(4, 1)$ form the strongly connected component. Notice that for each node $i \in \{1, 2, 4\}$ in the strongly connected component, the intersection of the node id's contained in $P(i)$ and $S(i)$ is $\{1, 2, 4\}$. Therefore, each node i knows the set of nodes in the strongly connected component containing node i. Notice that the intersection of the node id's in sets $P(3)$ and $S(3)$ is 3 indicating that node 3 is not contained in any strongly connected component of G.

Notice that each node of G is required to know only the set of its predecessors and successors and accesses the states of its neighbors. Also, notice that each node is not required to know anything specific to the entire graph G such as the number of nodes in G. Therefore, when a topology change takes place, the set of predecessors and successors for some nodes may change. This change will be treated as a transient fault and hence the proposed algorithm identifies the strongly connected for the new topology.

4 Algorithm

This section presents the self-stabilizing algorithm for finding the strongly connected components implementing the strategy described above.

The self-stabilizing algorithm for finding the strongly connected components is given in Figure 2.

In the description of the algorithm, $(i,)$ denotes a tuple whose first element is i, but the second element can be an arbitrary positive integer.

Throughout the paper, we refer to the first guard of the algorithm as $G1$ and the corresponding move as $M1$, the second guard as $G2$ and the corresponding move as $M2$, and so on. Upon discovering that element $(i, 0)$ is not in $P(i)$ and $S(i)$ (see $G1$ and $G2$) node i includes $(i, 0)$ in $P(i)$ and $S(i)$, respectively (see $M1$ and $M2$). Then, node id i propagates to the successors of node i, see $G3$ and $M3$. In the process of propagating node id i, the second element of the two-tuple is changed reflecting the distance from node i. For instance, when the propagation reaches node j, node j includes (i, d) in $P(j)$, where d is the distance of node j from node i. Analogously, node id i propagates to the predecessors of node i, which is implemented by guard $G4$ and move $M4$. When a tuple is included in a set, if another tuple with the same node id exists in the set, it is removed first (see $M1$ through $M4$). In addition, absence of tuple (i, d) in the P-sets of all its immediate predecessors (parents) of node j initiates the removal of $(i, d+1)$ from $P(j)$, and eventually, all the successors of node j remove $(i, \_)$ from their P-sets, see $G5$ and $M5$. This is necessary because there may be some tuple included in the sets due to arbitrary initialization which need to be removed. Similarly, $G6$ and $M6$ are responsible for removal of the tuples from S-sets that have been included due to arbitrary initialization.

5 Correctness

In this section of the paper, we provide a correctness proof of the algorithm and the algorithm complexity. In this version of the paper, the proofs of the lemmas and theorems are omitted. In addition, the proof of self-stabilization is omitted.

Node i

VARIABLE $P(i), S(i)$: sets of at most cn tuples of the form (id, d), where id is a node in G, and c, d are positive integers.
FUNCTION $P^p(k)$: set of tuples in the P-sets of the predecessors of i whose first elements are k.
FUNCTION $S^s(k)$: set of tuples in the S-sets of the successors of i whose first elements are k.
FUNCTION $min(X)$: the lexicographically smallest tuple in X.
CONSTANT N : a sufficiently larger number than n and on the order of n, where n is the number of nodes in G.
PARAMETER (k, l) : a tuple, where $k \in V$ and l is a positive integer.
ACTIONS

$$
\begin{array}{lcl}
*\big[(i,0) \notin P(i) & \longrightarrow & P(i) = \{P(i) - \{(i,\_)\}\} \cup \{(i,0)\} \\
\Box\ (i,0) \notin S(i) & \longrightarrow & S(i) = \{S(i) - \{(i,\_)\}\} \cup \{(i,0)\} \\
\Box\ P\_include.(k,l) \wedge (k,l) \notin P(i) & \longrightarrow & P(i) = \{P(i) - \{(k,\_)\}\} \cup \{(k,l)\} \\
\Box\ S\_include.(k,l) \wedge (k,l) \notin S(i) & \longrightarrow & S(i) = \{S(i) - \{(k,\_)\}\} \cup \{(k,l)\} \\
\Box\ P\_remove.(k,l) & \longrightarrow & P(i) = P(i) - \{(k,l)\} \\
\Box\ S\_remove.(k,l) & \longrightarrow & S(i) = S(i) - \{(k,l)\}\big]
\end{array}
$$

where
$P\_include.(k,l)$: $(k, l-1) = min(P^p(k))$ and l is not larger than n.
$S\_include.(k,l)$: $(k, l-1) = min(S^s(k))$ and l is not larger than n.
$P\_remove.(k,l)$: (k,l) is in $P(i)$, but $(k, l-1)$ is not in $P^p(k)$ or l is larger than N.
$S\_remove.(k,l)$: (k,l) is in $S(i)$, but $(k, l-1)$ is not in $S^s(k)$ or l is larger than N.

Figure 2: The self-stabilizing algorithm for finding the strongly connected components (SCC Algorithm)

5.1 Partial Correctness

We need the following definition. Let $d(v,i)$ denote the length of the shortest directed path from v to i. Following lemmas establish the partial correctness of the algorithm.

Lemma 2 *If algorithm SCC terminates, after termination the following holds. For every node j in G, $(j,d) \in P(i)$ iff there exists a directed path with origin j and terminus i and d is the length of this path.*

Proof: We first prove that if there is a directed path with origin j and terminus i, then $(j,d) \in P(i)$, where d is the length of this path by induction on l, where l is the length of the directed path from j to i.

Basis: $l = 0$

If $l = 0$, then we know that $d(i,j) = 0$ and $i = j$. Observe that since $G1$ is disabled for i, $P(i)$ contains (i,j).

Induction Hypothesis: For each node $v \in V$ on the directed path from j to i such that $d(j,v) \leq l$, $(j,d) \in P(v)$, where $d = d(j,v)$.

Induction Step: Assume the induction hypothesis. To show that for each node $v \in V$ such that $d(j,v) = l+1$, $(j, l+1) \in V$. By the induction hypothesis, we know that for each predecessor of k of v such that k is on a shortest path from j to v, $(j,l) \in P(k)$ holds. Consequently, since $G2$ is disabled for v, we have that $(j, l+1) \in P(v)$.

Now, we show that if $(j,d) \in P(i)$, then there exists a directed path from j to i and the length of the shortest such path is d. Assume the contrary, i.e., $(j,d) \in P(i)$ but there does not exist a directed path from j to i or the length of the shortest such path is not d.

First, observe that since $(j,d) \in P(i)$ and $G3$ is disabled for i, there exists a predecessor v_{d-1} of node i such that $(j, d-1) \in P(v_{d-1})$. Similarly, since $(j, d-1) \in P(v_{d-1})$ and $G3$ is disabled for v_{d-1}, there exists a predecessor v_{d-2} of node v_{d-1} such that $(j, d-2) \in P(v_{d-2})$, and so on. Inductively, it can be shown that there exists a directed path $v_0, v_1, \cdots, v_d$, where $v_0 = j$, $v_d = i$, $v_t \in V$ for $0 \geq t \geq d$ and $(j,t) \in P(v_t)$ for $0 \geq t \geq d$. Hence, the proof follows. $\square$

The proof of the following lemma is analogous to that of Lemma 2, therefore omitted.

Lemma 3 *If algorithm SCC terminates, after termination the following holds. For every node j in G, $(j,d) \in S(i)$ iff there exists a directed path with origin i and terminus j and d is the length of this path.*

The following lemma establishing the partial correctness of the algorithm follows from Lemma 2, 3, and the definition of strongly connected components.

Lemma 4 *(Partial Correctness) After termination, $(j,d) \in P(i)$ and $(j,m) \in S(i)$ iff nodes j and i belong to the same strongly connected component.*

5.2 Algorithm Complexity and Termination

In this section, we present the worst case time complexity or the upper bound of the SCC algorithm.

We first classify the moves of the algorithm into two categories called *initial moves* and *non-initial moves*. The initial moves are the ones that are caused by arbitrary initialization and the non-initial moves are the ones that are caused by other moves in the system. We know that each move by node i causes a tuple x to be included in $P(i)$ (or $S(i)$) or removed from $P(i)$ (or $S(i)$). Tuple x is referred to as an *initial* tuple with respect to $P(i)$ if it has been in $P(i)$ since the start of the system. Otherwise, it is referred to as *non-initial* tuple. A move that includes tuple x in $P(i)$ is referred to as an *initial move* if

- $x = (i, 0)$, or
- x is an initial tuple with respect to the P-set of a parent of node i.

In addition, a move that removes edge id x from $P(i)$ is referred to as an *initial move*, if x is not contained in the P-set of any parent of node i since the start of the system. Otherwise, the move is referred to as a *non-initial* move.

Now, we show that the algorithm terminates. We first show the upper bounds on the number of initial moves.

Lemma 5 *The total number of initial moves is $O(n^2)$.*

Proof: We know that there exists $O(n^2)$ initial tuples in the system. We also know that each node $i \in V$ can make at most one move to include $(i, 0)$ in $P(i)$. By the definition of initial moves, $O(n^2)$ initial moves can be made. Hence, the proof follows. $\square$

An execution in a distributed system can be described as a sequence of moves $M_1, M_2 \cdots$, where M_j is a move made by a node in the system. Consider a move M_k by an arbitrary node i. We identify a unique parent j of node i and a unique move M_l where $l < k$, by node j to be the "cause" of move M_k defined as follows.

Define $cause()$ for initial moves

- $cause(M_k) = M_k$ if M_k is an initial move by node i.

Define $cause()$ for non-initial moves including tuple (t, d) in $P(i)$

- $cause(M_k) = M_l$ if M_l is the last move made by a parent of node i that included $(t, d-1)$ in its P-set.

Analogously, define $cause()$ for non-initial moves removing edge id (t, d) from $P(i)$

- $cause(M_k) = M_l$ if M_l is the last move made by a parent of node i that removed $(t, d-1)$ from its P-set.

We now state two useful properties related to the function $cause()$. The first property is that distinct moves by a node have distinct causes.

Proposition 6 *If M_p and M_q are distinct moves by node i then*

$$cause(M_p) \neq cause(M_q).$$

The next property that we establish is that the cause relationship is "acyclic" and the cause of a move by node i is always another move by a parent of node i. The following proposition follows from the fact that each node only reads its parent's variables to evaluate its guards, see Algorithm SCC in Figure 2.

Proposition 7 *Let M_k be a move by node i. If M_k is not the initial move by node i, then the move $cause(M_k)$ is made by a parent of node i.*

Now, based on the definition of $cause()$ we define the notion of the source of a move. We then show that distinct moves have distinct sources and use that to prove an upper bound on the total number of moves made by all the nodes in the system.

For each move M_k define

$$source(M_k) = \begin{cases} M_k & \text{if } M_k \text{ is an initial move} \\ source(cause(M_k)) & \text{if } M_k \text{ is a non-initial move} \end{cases}$$

Intuitively, $source(M_k)$ can be thought of as an initial move that causes M_k through a chain of moves.

The following proposition states a useful property of sources of moves. The proof of this proposition immediately follows from Proposition 6 and Proposition 7, hence omitted.

Proposition 8 *The source of each move of Algorithm SCC is a distinct initial move.*

Each initial move can trigger a bounded number of moves. The following lemma provides an upper bound on the number of moves caused by a single source.

Lemma 9 *Each distinct initial move can be the source of of a bounded number of moves of Algorithm SCC.*

Proof: We first show that each initial move changing a P-set can be the source of at most $O(n)$ moves changing P-sets.

We refer to a move that removes an element from a set is referred to as a *decreasing move*, similarly, a move including an element in a set as an *increasing move*.

We now show that each increasing move by an immediate predecessor (parent) of node $i \in V$ can cause at most one increasing move by i. We know that each move is caused by another move made by one of its parents. We also know that if the parent j of i includes (k, d) in $P(j)$ by move M, node i includes $(k, d+1)$ in $P(i)$ by move M' caused by move M by the definition of cause. Since $(k, d+1)$ is in $P(i)$ after move M, a second move by node i caused by M after move M' cannot include $(k, d+1)$ in $P(i)$ since it already exists. Thus, we have that each increasing move M can cause at most one increasing move move by the same node.

Analogously, it can be shown that a decreasing move by a parent of of i can cause at most one decreasing move by i.

Now, we show that each initial increasing move changing a P-set can be the source of a bounded number of increasing moves moves changing P-sets. If tuple (k, t) is included in

$P(i)$ such that i is included in a directed cycle of G, eventually, the tuple may propagate in the cycle and reach one of the parent of i. Recall that in each step of the propagation, the second element of the tuple is incremented by one. Therefore, when the propagation reaches a parent of i, the second element of the tuple contained in the P-set of the parent of i is larger than the tuple contained in $P(i)$. Note that all the aforementioned tuples have k as their first element. When node i finds the tuple in the P-set of one of its parents, it increments the second element of its tuple at least by the length of the cycle. Observe that since the second element cannot exceed N, the tuple can propagate through the cycles of G only a bounded number of times. From the above argoments, it can easily be observed that each initial move can be the source of a bounded number of moves.

We know that a decreasing move removing a tuple $(k, \_)$ by a parent of node i can only cause a decreasing move removing a tuple $(k, \_)$ from $P(i)$, where $(k.\_)$ refers to a tuple with k as the first element and the second tuple can be arbitrary. Since at most n nodes may contain $(k, \_)$ in their P-sets, each initial descreasing move can be the source of at most n decreasing moves. Thus, we have that each initial move changing a P-set can be the source of at most $O(n)$ moves changing P-sets.

Similarly, it can readily be shown that each initial move changing a S-set can be the source of a bounded number of moves changing S-sets. Hence, the proof follows.

□

The following lemma establishes the termination of the algorithm.

Lemma 10 *(Termination) Algorithm SCC terminates.*

Proof: The proof follows from Lemma 5, Lemma 9 and Proposition 8. □

As a consequence of Lemma 4 and Lemma 10, we now establish the total correctness of our algorithm.

Lemma 11 *(Total Correctness) Algorithm SCC identifies all strongly connected components of G after a bounded number of moves.*

We now provide the round complexity of our algorithm. A *round* refers to a minimum execution sequence in which each enabled action is taken at least once. The following lemma shows the round complexity of our algorithm.

Lemma 12 *Algorithm SCC terminates after n rounds, where n is the number of nodes in G.*

Proof: It can readily be observed that after the first round, each node $i \in G$ contains $(i, 0)$ in $S(i)$ (see $G1$ of the algorithm).

Now, in each one of the following rounds of the algorithm, for each node $i \in V$, node id i is guaranteed to propagate further. That is, after k rounds of the algorithm, for every node $j \in V$ such that $d(i,j) \leq k$, $(i,k) \in S(j)$ holds. Inductively, it can be shown that since the distance of the farthest node from i is n, after n rounds, the algorithm terminates. $\square$

Now, we show that algorithm SCC is self-stabilizing.

Let PRE be a predicate defined over SYS, the set of global states of the system. An algorithm ALG running on SYS is said to be *self-stabilizing* with respect to PRE if it satisfies:

Closure: If a global state q satisfies PRE, then any global state that is reachable from q using algorithm ALG, also satisfies PRE.

Convergence: Starting from an arbitrary global state, the distributed system SYS is guaranteed to reach a global state satisfying PRE in a finite number of steps of ALG.

Global states satisfying PRE are said to be *stable*. Similarly, a global state that does not satisfy PRE is referred to as an *instable state*. To show that an algorithm is self-stabilizing with respect to PRE, we need to show the satisfiability of both closure and convergence conditions. In addition, to show that an algorithm solves a certain problem, we need to either prove *partial correctness* or show that through transitions made by the algorithm among stable states the problem is solved.

We now show that algorithm SCC is self-stabilizing by establishing the convergence and the closure properties.

Let P be a predicate defined as follows. For every node i in G, $(k,l) \in P(i)$ and $(k,m) \in S(i)$ iff nodes k and i belong to the same strongly connected component, l is the distance of node i from node k, and m is the distance of node k from node i.

By Lemma 4 and Lemma 10, we know that predicate P is eventually satisfied establishing the convergence property. In addition, by Lemma 10, we know that when predicate P is satisfied, no guard is enabled. Therefore, the closure property is trivially satisfied. Hence, algorithm SCC is self-stabilizing.

From the above discussion, Lemma 4 and Lemma 10, we have the following theorem.

Theorem 13 *Algorithm SCC is self-stabilizing and it identifies all strongly connected components a bounded number of moves.*

6 Conclusions

On a distributed or network model of computation, we have presented a self-stabilizing algorithm that identifies the set of strongly connected components in a connected directed graph. Algorithm SCC assumes that every node knows only about its neighbor node numbers. Therefore, starting only with the neighbor information to each node of G, it is possible to find the strongly connected components of G after $O(n)$ rounds. We showed that the proposed algorithm is correct.

References

[1] Arora, A., Gouda, M. G.: Distributed reset. IEEE Transactions on Computers, 43 (1994), 1026–1038.

[2] Awerbuch, B., Varghese, G.: Distributed program checking: A paradigm for building self-stabilizing distributed protocols. in FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science, 1991, pp. 258–267.

[3] Brown, G. M., Gouda, M. G., Wu, C. L.: Token systems that self-stabilize. IEEE Trans. Comp., 38 (1989), 844–852.

[4] Bruell, S. C., Ghosh, S., Karaata, M. H., Pemmaraju, S. V.: Self-stabilizing algorithms for finding centers and medians of trees. To Appear in SIAM Journal on Computing.

[5] Burns, J. E., Pachl, J.: Uniform self-stabilizing rings. ACM Transactions on Programming Languages and Systems, 11 (1989), 330–344.

[6] Chartrand, G.: Introductory Graph Theory, Dover Books Publishing Co., Inc., 1985.

[7] Dijkstra, E. W.: Self-stabilizing systems in spite of distributed control. in EWD 391, In Selected Writings on Computing: A Personal Perspective, 1973, pp. 41–46.

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

[9] Huang, S. T.: Leader election in uniform rings. ACM Trans. Prog. Lang. Syst., 15 (1993), 563–573.

[10] Jiang, B.: I/O- and cpu optimal recognition of strongly connected components. Information Processing Letters, 45 (1989), 111–115.

[11] Katz, S., Perry, K.: Self-stabilizing extensions for message-passing systems. Distributed Computing, 7 (1993), 17–26.

[12] Lin, X., Ghosh, S.: Self-stabilizing maxima finding. in Proceedings of the 28th Annual Allerton Conference, 1991, pp. 662–671.

[13] Mohanty, H., Bhattacharjee, G. P.: Strongly connected components and knots in distributed systems. Computer Science and Informatics, 22 (1992), 28–31.

[14] Nuutila, E., Soisalon-Soininen, E.: On finding the strongly connnected components in a directed graph. Information Processing Letters, 49 (1994), 9–14.

[15] Schneider, M.: Self-stabilization. ACM Computer Surveys, 25 (1993), 45–67.

[16] Sharir, M.: A strong-connectivity algorithm and its application in data flow analysis. Comput. Math. Appl., (1981), 67–72.

[17] Sur, S., Srimani, P. K. : A self-stabilizing distributed algorithm for BFS spanning trees of a symmetric graph. Parallel Processing Letters, 2 (1992), 171–179.

[18] Tarjan, R. E.: Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1 (1972), 146–160.