Kuwait J. Sci. Eng. **29**(2) 2002

A dynamic optimal stabilizing algorithm for finding strongly connected components*

MEHMET HAKAN KARAATA AND FAWAZ AL-ANZI

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

ABSTRACT

An optimal self-stabilizing algroithm is presented that finds the strongly connected components of a directed graph on an underlying network after $O(C)$ rounds, where C is the length of the longest cycle in the graph. Because the algorithm is self-stabilizing, it is resilient to transient faults and does not require initialization. The proposed algorithm can withstand topology changes in the form of addition or removal of edges and vertices. A correctness proof of the algorithm is provided.

Keywords: Directed graphs; distributed systems; self-stabilization; strongly connected components.

INTRODUCTION

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

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

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 the *strongly connected components* of the graph. Finding strongly connected components is used in data flow analysis (Sharir 1981), and for finding transitive closures, disjoint paths, and knots (Nuutila and Soisalon-Soininen 1994, Mohanty and Bhattacharjee 1992). 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.

* A preliminary version of this paper was presented at PODC'99 as a brief announcement.

There are a number of sequential and distributed algorithms for finding strongly connected components, however, to the best of our knowledge, no self-stabilizing algorithm has appeared in the literature. (Tarjan 1972) presents an asymptotically optimal sequential algorithm for finding the strongly connected components, with time complexity $O(n+e)$. (Nuutila and Soisalon-Suininen 1994) proposed two efficient sequential algorithms for finding strongly connected components, these algorithms are improved versions of Tarjan's algorithm. Another linear algorithm is presented in by (Sharir 1981). 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 (Jiang (1989) proposed a new linear algorithm based on a traversal strategy that is the combination of depth-first and breadth-first traversal. A distributed algorithm for finding strongly connected components was proposed by (Mohanty and Bhattacharjee (1992). All 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 and is not appropriate for distributed computation. Our algorithm is the first algorithm not based on depth-first traversal.

In this paper, we present a dynamic, self-stabilizing algorithm for finding the strongly connected components of a connected, directed graph. The proposed algorithm, which we 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 text, as a transient fault. The algorithm is optimal with respect to its round complexity and requires $O(C)$, where C is the length of the longest cycle in the graph, rounds to identify all strongly connected components in a graph.

Because of the dynamic nature of distributed systems, protocols that gracefully tolerate topology changes in the form of node and edge additions or removals are desirable. Such *dynamic protocols* are intrinsically fault tolerant. In the proposed algorithm, each node of G is required to know only the set of its parents and offsprings and only needs to access the states of its neighbors (parents and offsprings). Also, no node requires global knowledge of the entire graph G, other than N, where $N >> n$ holds, i.e., N is a lot larger than n, the number of nodes in G. When a topology change takes place, the set of parents and offsprings for some nodes may change. This change is treated as transient and the proposed algorithm automatically recomputes the values of the variables accordingly, and thus identifies the strongly connected for the new topology.

The property of self-stabilization is also a desirable property of fault tolerant distributed systems. A self-stabilizing system guarantees that, regardless of the initial state, the system will reach a legal state in a bounded number of steps, and remain in a legal state thereafter. The addition of the property of self-stabilization to any of the existing sequential algorithms for finding the strongly connected components seems to be difficult. The property of self-stabilization requires any such solution to satisfy additional properties.

Since the introduction of self-stabilization (Dijkstra 1973), primarily in distributed systems, self-stabilizing algorithms for many fundamental problems in distributed systems have been proposed. For example, self-stabilizing mutual exclusion algorithms for a variety of network classes have been presented (Brown *et al.* 1989, Burns and Pachl 1989, Dijkstra 1973). Self-stabilizing model transformers appear in Nesterenko and Arora (1999), and Karaata (2001). Self-stabilizing algorighms for a variety of graph theoretic problems are presented in Huang and Chen (1993), Datta *et al.* (2000), Petit and Villain (1997), Bruell *el al.* (1999), Karaata and Chaudhuri (1999), Karaata (1999, 2002), and Karaata and Chaudhuri (2000). General techniques for constructing stabilizing algorithms are dealt with (Arora and Gouda 1994, Awerbuch and Varghese 19911 Katz and Perry 1993). Self-stabilizing algorithms are able to withstand *transient failures*. In addition, many self-stabilizing algorithms are capable of gracefully dealing with the dynamic addition and deletion of vertices or edges. Dolev (2000) provides a detailed survey on self-stabilization.

This paper is organized as follows: In section 2, we present our computational model. Section 3 contains some basic definitions and some informal intuitions into our proposed solution. Section 4 presents the algorithm itself, and in Section 5 we prove the correctness of the algorithm, show its time complexity, and prove that it is self-stabilizing. We close with some observations in Section 6.

2. COMPUTATIONAL MODEL

Let $G = (V, E)$ be a simple directed connected graph with vertex set V and edge set E, where $|V| = n$. We assume that each vertex of G is a sequential process or node. For each directed edge in E, We assume a bidirectional communication link. Each node maintains a set of local variables, the values of which can be updated by the node after evaluating its local variables and the local variables of its neighbors incident on incoming and outgoing edges. The program of node i can be expressed as a nondeterministically executed sequence of statements:

$$^*[\, G[1] \rightarrow M[1] \,\square\, G[2] \rightarrow \ldots \,\square\, G[k] \rightarrow M[k]\,],$$

where

- each *guard* $G[\ ]$ is a boolean function of the variables of node i and the variables of the neighboring nodes (incident on incoming and outgoing edges);

- each *move* $M[\ \ ]$ atomically updates the variables of node *i*, and
- $^*[S]$ corresponds to the repeated execution of the statement S untill all guards are false, whereupon the program terminates;
- $\square$ is called the nondeterminism symbol. One of the guarded actions separated by them is selected nondeterministically in each iteration.

If a guard of node *i* evaluates to true, we say that the guard and node *i* are *enabled*. If two or more guards are enabled at the same time, then the *scheduler* arbitrarily selects a subset of statements with enabled guards and executes the corresponding moves. In the case of the *central scheduler*, a single statement is selected and its move is completed before any guard is reevaluated. In a *distributed scheduler*, guards can be concurrently evaluated and their moves taken at any time. To simplify our proofs we assume a weakly fair central scheduler.

The *state space* of a node is the cartesian product of all variables defined locally within the node. The *state* of a node is the value of all of its local variables at an instant, and is one element of the state space for the node. The *system state space* is defined as the cartesian product of the state spaces of all nodes in the system. A *state* of the system is defined by the state of all nodes at an instant, and is an element of the system state space. The system state before the system is started represents the *initial state* of the system.

3. BASIS OF THE ALGORITHM

The self-stabilizing algorithm for finding strongly connected components is based on the following important observation form Chartrand (1985):

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 path from i to j in G and a path from j to i in G.*

The proof immediately follows from the definition of a strongly connected component and is omitted.

A node *i* is said to be a predecessor of another node *j*, with respect to *G*, if there exists a directed path from *i* to *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. From Lemma 1, we know that for each node *i*, the intersection of the set of predecessors *P* and successors *S* of *i* gives us the nodes in the strongly connected component containing *i*. Although the approach is rather simple, the challenge lies in making the algorithm self-stabilizing.

Before describing the SCC algorithm formally, we establish a formal basis for the algorithm. Each process i defines two variables P and S called the P-set and S-set of i which we denote by $P(i)$ and $S(i)$, respectively. The P-set and S-set for a node contain at most N, where $N = O(n)$, ordered pairs of the form (id, d), where $id \in V$, and d is a nonnegative integer. The P-set (or S-set) of node $i \in V$ can contain at most one tuple with the same id. We allow N tuples in each set to accommodate dynamic addition of nodes to the graph on n nodes. The proposed algorithm allows the graph to contain at most N nodes. We assume that each set can contain no more than one ordered pair with a given 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 i discovers that its P-set does not contain an ordered pair with id i, it adds the ordered pair $(i, 0)$ to its P-set. Each successor of i includes the ordered pair (i, d) in its P-set and each predecessor of node i includes (i, d) in its S-set, where d represents a successor's or predecessor's distance from i. Upon termination of the algorithm, each node i can identify the set of nodes belonging to the strongly connected component node i by taking the intersection of the node id's in the ordered pairs of sets $P(i)$ and $S(i)$.

The above discussion on our approach to find strongly connected components does not include a discussion of how to make the algorithm distributed and self-stabilizing; we now discuss these issues. We refer to the immediate predecessors of a node as *parents* and the immediate successors as *offsprings*. For each node $i \in V$, i includes $(i, 0)$ in $P(i)$, then node i's offsprings (immediate successors) include $(i, 1)$ in their P-sets and consequently node i's offsprings' offsprings include $(i, 2)$ in their P-sets, and so on. In this manner, an ordered pair containing node id i propagates to all successors of i. Such propagations originating from node i carrying node id i to its successors represent *desirable propagations*. In each step of a desirable propagation, when node id i is included in $P(j)$, the distance value d associated with id i denotes the distance of node j from node i, i.e., d denotes the length of the shortest directed path from j to i. If the tuple (i, e) is included in $P(j)$ due to an arbitrary initialization such that i is not a predecessor at distance d from j, it can also result in propagations of node id i, and such propagations represent *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 is not the distance of node j from node i. The id i may not even represent a valid node id.

Clearly, undesirable propagations need to be stopped, and any associated tuples originating from the arbitrary initial state must be removed from the *P*-sets and *S*-sets of processes in order to make the approach self-stabilizing. 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* eventually terminate. Whereas, an undesirable propagation may reach its place of origin and continue its propagation indefinitly. If (i, d) is included in $P(j)$ by an undesirable propagation, then d can exceed the number of nodes in the system, n.

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 $N >> n$ and $N = O(n)$. Thus, no tuple with a distance value more than N is incremented further that N and undesired propagations are eventually terminated. After an undesired propagation stops, the tuples included in the sets due to this propagation are removed as follows. For any process j if there exists an ordered pair $(-, d)$ in $P(j)$, where $d > N$, then it is removed. Also, we remove a tuple (i, d) from $P(j)$, if there exists no parent k such that $(i, d-1) \in P(k)$ and $i \neq j$ holds. After an undesired propagation is stopped, there must exist at least on node j such that $(i, d) \in P(j)$ where j has no parent k such that $(i, d-1) \in P(k)$. The removal of undesirable tuples continues in this fashion until all tuples with incorrect distance values are eventually removed from all *P*-sets. The S-sets are computer analogously to *P*-sets by using the bidirectional property of the underlying communication link.

We illustrate these concepts with the help of an example. In Figure 1, a directed graph on four nodes is shown, where each directed edge is shown by an arrow. The

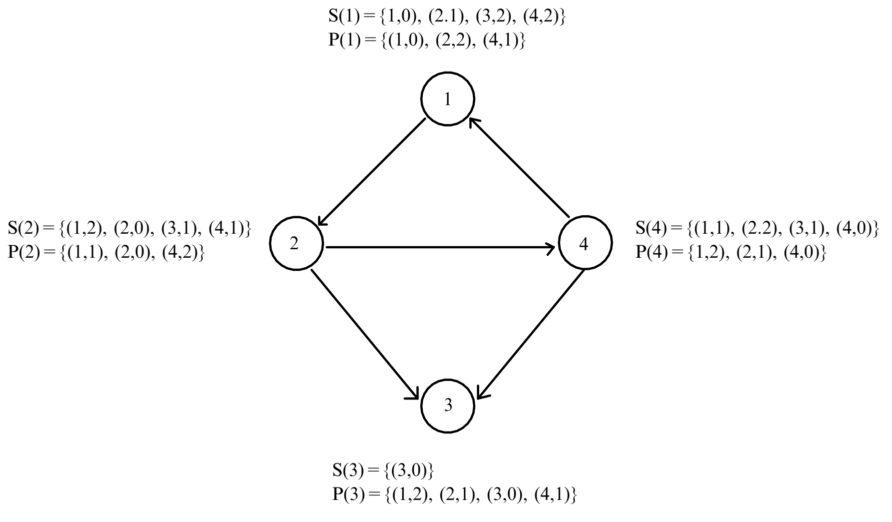
S(1) = {1,0), (2.1), (3,2), (4,2)}
P(1) = {(1,0), (2,2), (4,1)}
1
S(2) = {(1,2), (2,0), (3,1), (4,1)}
P(2) = {(1,1), (2,0), (4,2)}
2
4
S(4) = {(1,1), (2.2), (3,1), (4,0)}
P(4) = {1,2), (2,1), (4,0)}
3
S(3) = {(3,0)}
P(3) = {(1,2), (2,1), (3,0), (4,1)}

Figure 1: The state of a system after termination

contents of P-sets and S-sets associated with each node after termination are shown besidetheir respective node. The graph contains two strongly connected components, one of them consists of nodes 1,2,4 and edges (1,2), (2,4) and (4,1), and the other contains only node 3. 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 of all of nodes in the strongly connected component it is part of. For node 3, which is not part of a strongly connected component, 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.

4. ALGORITHM

In Figure 2, we present a self-stabilizing algorithm for finding the strongly connected components by implementing the strategy described above. The guards are evaluated for all combinations of k and l.

```
Node $i$

VARIABLE $P(i), S(i)$   :  sets of at most $N$ tuples of the form $(id, d)$, where $id$ is a node in $G$, and $d$
                           is a positive integer.

FUNCTION $P^p(k)$       :  set of tuples in the $P$-sets of the parents of $i$ whose first elements are $k$.

FUNCTION $S^s(k)$       :  set of tuples in the $S$-sets of the offsprings 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 non-negative integer.

ACTIONS

$*[(i, 0) \notin P(i)$                          $\rightarrow$  $P(i) = \{P(i) - \{(i, -)\}\} \cup \{(i, 0)\}$
$\square (i, 0) \notin S(i)$                    $\rightarrow$  $S(i) = \{S(i) - \{(i, -)\}\} \cup \{(i, 0)\}$
$\square P\_include.(k, l) \wedge (k, l) \notin P(i)$   $\rightarrow$  $P(i) = \{P(i) - \{(k, -)\}\} \cup \{(k, l)\}$
$\square S\_include.(k, l) \wedge (k, l) \notin S(i)$   $\rightarrow$  $S(i) = \{S(i) - \{(k, -)\}\} \cup \{(k, l)\}$
$\square P\_remove.(k, l)$                      $\rightarrow$  $P(i) = P(i) - \{(k, l)\}$
$\square S\_remove.(k, l)$                      $\rightarrow$  $S(i) = S(i) - \{(k, l)\}]$

where

  $P\_include.(k, l)$  :  $(k, l-1) = min(P^p(k)) \wedge l \leq N \wedge k \neq i$
  $S\_include.(k, l)$  :  $(k, l-1) = min(S^s(k)) \wedge l \leq N \wedge k \neq i$
  $P\_remove.(k, l)$   :  $((k, l) \in P(i) \wedge (k, l-1 \notin P^p(k) \wedge i \neq k) \vee 1 > N$
  $S\_remove.(k, l)$   :  $((k, l) \in S(i) \wedge (k, l-1) \notin S^s(k) \wedge i \neq k) \vee l > n$
```

Figure 2. Self-stabilizing algorithm for finding strongly connected components.

Throughout the paper, we refer to the first guard of each algorithm as $G1$, the second guard as $G2$, and so on. The notation $(i, -)$ denotes a tuple with node id i, but where the second element can be any positive integer. Upon discovering that element $(i, 0)$ is not in $P(i)$ and $S(i)$ node i includes $(i, 0)$ in $P(i)$ and $S(i)$, respectively (see $G1$ and $G2$). Then, node id i propagates to the successors of node i (see $G3$). In the process of propagating node id i, the second element of the two-tuple is changed reflecting the distance form 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 $G4$. 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 $G1$ throught $G4$). 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 $G5$). This is necessary because there may be some tuple included in the sets due to arbitrary initialization which need to be removed. Similarly, $G6$ is responsible for removal of the tuples from S-sets that have been included due to an arbitrary intialization.

5. CORRECTNESS

In this section of the paper, we provide a correctness proof of the algorithm and the algorithm complexity.

5.1 Partial Correctness

We need the following definition. Let $d(\nu, i)$ denote the length of a shortest directed path from ν to i. The following lemmas establish the partial correctness of the algorithm.

Lemma 2 *If algorithm SCC terminates, after its 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 $(i, 0)$.

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

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

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 parent 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 parent 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 $v_0, v_1, \ldots, v_d$, 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. □

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* $V, (j, d) \in S(i)$ *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 tuple 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. □

An execution in a distributed system can be described as a sequence of moves $M_1, M_2 \ldots$, 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_1 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 tupe (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*

$$\textit{cause}\,(M_p) \neq \textit{cause}\,(M_q).$$

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

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

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 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 as a *decreasing move*; similarly, we refer to 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 difinition 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 by the same node.

Analogously, it can be shown that a decreasing move by a parent 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 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 parents 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 hte second element cannot exceed N, the tuple can propagate through the cycles of G only a bounded number of times. From the above arguments, it can easily be observed that each initial move can be the source of a bounded number of moves.

We know that a devreasing 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 element can be arbitrary. Since at most n nodes may contain $(k, -)$ in their P-sets, each initial decreasing 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 ouralgorithm.

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. is the shortest prefix E'. Let E'' be the suffix of E that follows E'', $E = E' E''$. The second round of E is the first round of E'', and so on. The number of rounds in execution is used to measure the complexity of an algorithm.

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. □

We showed that our algorithm terminates after $O(n)$ rounds, where C is the length of the longest cycle in the graph. It is easy to see that N, the number of elements in each S and P-set, can be chosen as small as C, where l is the length of the longest cycle in G. Then, the round complexity of the algorithm can be given as $O(C)$. Note that by reducing N, we limit the number of nodes that can be added to the graph. Now, we sketch a proof that our algorithm is optimal with respect to its round complexity. We know that detecting cycles is an essential part of finding strongly connected components. We also know that in order to detect that i is in a cycle, the knowledge of node i being a predecessor (or successor) of itself is required as stated by the following lemma.

Lemma 13 *If processes know only their predecessors, process i is deadlocked iff i is included in the predecessor set of one of its predecessors.*

Proof: The proof follows from the definition of deadlock. □

Let $i \in V$ be a node contained in a cycle of G. Observe that i's existence in the cycle has to propagate to its successors and/or predecessors. It is easy to see that this propagation can reach a distance of one unit farther in each round, i.e., after k rounds the propagation can reach any successor or predecessor at distance k or less from i. Only after propagations to predecessors and successors encounter at some node, the existence of $i$in a cycle can be determined by some node in G. However, by the definition of our problem node i needs to know that it is contained in a cycle. Therefore, this knowledge of existence of a deadlock has to reach node i. Clearly, only after $O(C)$ rounds, each node $i \in V$ can determine that it is contained in a cycle, where C is the length of the longest cycle in G. Hence, our algorithm is optimal with respect to its round complexity.

Observe that the state space of our algorithm is $O(nlogn)$ bits for each node. We are to show that our algorithm is also state space optimal. That is, the particular problem we addressed in the paper cannot be solved using less than $O(nlogn)$ bits per node.

Recall that the aim of our algorithm is to guarantee that when a final state is reached, each node knows the set of nodes contained in the strongly connected component it is contained in. In addition, our algorithm ensures that a final stage is eventually entered. Since log_2n bits are required to encode each node id and in a strongly connected component as many as n nodes may exist, the minimum state space requirement for solving the problem is $O(nlogn)$ bits per process. Therefore, our algorithm is state space optimal.

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 Q 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 Q is eventually satisfied, establishing the convergence property. In addition, by Lemma 10, we know that when predicate Q 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 14 *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 $O(C)$ rounds. We showed that the proposed algorithm is correct. Note that the system model and both the space and the time complexities of the algorithm make it suitable for implementation. Therefore, the algorithm, its stability and distributed featues are ammendable to implementation.

ACKNOWLEDGEMENTS

We would like to thank the anonymous referees for their suggestions and constructive comments on an earlier version of the paper. Their suggestions have greatly enhanced the readability of the paper.

REFERENCES

Arora A., and Gouda M. G. 1994. Distributed reset. IEEE Transactions on Computers, **43:** 1026-1038.

Awerbuch B., and Varghese G. 1991. 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, pages 258-267.

Brown G. M., Gouda M. G., and Wu C. L. 1989. Token systems that self-stabilize. IEEE Trans. Comp. **38(6)**: 844-852.

Bruell Steven C., Ghosh Sukumar, Hakan Karaata Mehmet, and Pemmaraju Sriram V. 1999. Self-stabilizing algorithms for finding centers and medians of trees. SIAM Journal on Computing **29(2)**: 600-614.

Burns J. E., and Pachl J. 1989. Uniform self-stabilizing rings. ACM Transactions on Programming Languages and Systems **11**: 330-344.

Chartrand Gary 1985. Introductory Graph Theory. Dover Books Publishing Co., Inc.

Datta A. K., Johnen C., Petit F., and Villain V. 2000. Self-stabilizing depth-first token circulation in arbitrary rooted networks. Distributed Computing, **13(4)**: 207-218.

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

Dolev Shlomi 2000. Self-Stabilizations. MIT Press, Cambridge, MA.

Huang Shing-Tsaan, and Chen Nian-Shing 1993. Self-stabilizing depth-first token circulation on networks. Distributed Computing **7**: 61-66.

Jiang B. 1989. I/o- and cpu optimal recognition of strongly connected components. Information Processing Letters **45(3)**: 111-115.

Karaata Mehmet Hakan 1999. A self-stabilizing algorithm for finding articulation points. International Journal on Foundations of Computer Science **1**: 33-46.

Karaata Mehmet Hakan 2001. Self-stabilizing strong fairness under weak fairness. IEEE Transactions on Parallel and Distributed Systms **12(4)**: 337-345.

Karaata Mehmet Hakan 2002. A self-stabilizing algorithm for finding biconnected components. To Appear in the Journal of Parallel and Distributed Computing.

Karaata Mehmet Hakan and Chaudhuri Pranay 1999. A self-stabilizing algorithm for bridge finding. Distributed Computing **2**: 47-53.

Karaata Mehmet Hakan and Chaudhuri Pranay 2000. A dynamic self-stabilizing algorithm for constructing a transport net. To Appear in Computing 2002.

Katz S. and Perry K. J. 1993. Self-stabilizing extensions for message-passing systems. Distributed Computing **7**: 17-26.

Mohanty H. and Bhattacharjee G. P. 1992. Strongly connected components and knots in distributed systems. Computer Science and Informatics **22(1)**: 28-31.

Nesterenko M. and Arora A. 1999. Stabilization-preserving atomicity refinement. In DISC99 Distributed Computing 13th International Symposium, pages 254-268. Springer-Verlag.

Nuutila Esko and Soisalon-Soininen Eljas 1994. On finding the strongly connected components i a directed graph. Information Processing Letters **49**: 9-14.

Petit F. and Villain V. 1997. A space-efficient and slef-stabilizing depth-first token circulation protocol for asynchronous message-passing systems. In Euro-par’97 Parallel Processing, Proceedings LNCS: 1300 pages 476-479. Springer-Verlag.

Sharir M. 1981. A strong-connectivity algorithm and its application in data flow analysis. Comput. Math. Appl., **(7)**: 67-72.

Tarjan R. E. 1972. Depth-first search and linear graph algorithms. SIAM Journal on Computing **1(2)**: 146-160.

Submitted : *29 May 2001*
Revised : *1 April 2002*
Accepted : *2 April 2002*

خوارزمية ديناميكية مثلى ذاتية الاستقرار لإيجاد المكونات شديدة الترابط

مهمت هاكان كراتا و فواز العنزي

قسم هندسة الكمبيوتر - كلية الهندسة والبترول

جامعة الكويت، ص.ب 5969 الصفاة - الرمز البريدي 13060 الكويت

خلاصة

نقدم في هذا البحث خوارزمية ديناميكية مثلى ذاتية الاستقرار لإيجاد المكونات شديدة الترابط في الرسوم الاتجاهية باستخدام شبكة من الحاسابات المتصلة بدوائر من الحجم س، حيث يمثل الحجم س أطول دائرة موجود في الرسم الاتجاهي. وبما أن الخوارزمية المقدمة هي ذاتية الاستقرار فهي مرنة بالنسبة للأخطاء الانتقالية التي يمكن حصولها كما أنها لا تحتاج إلى تحضير قبل البدء والتشغيل. الخوارزمية المقترحة يمكنها تحمل التغيرات في شكل الرسوم الاتجاهية كإضافة أو حذف بعض النقاط في الرسم الاتجاهي. كما تحتوي المقالة على إثبات رياضي لصحة الخوارزمية.