A SELF-STABILIZING ALGORITHM FOR FINDING DISTANCES IN A DISTRIBUTED SYSTEM

Tetz C. Huang and Ji-Cherng Lin

Department of Computer Engineering and Science, Yuan-Ze University
135 Yuan-Tung Rd., Chung-Li Taoyuan 32026, Taiwan
Email: cstetz@cs.yzu.edu.tw, csjclin@cs.yzu.edu.tw

ABSTRACT

In this paper, we propose a self-stabilizing algorithm for finding distances in a distributed system in which a central daemon is assumed. The correctness of the proposed algorithm is proved by using the bounded function technique.

I. Introduction

The notion of the self-stabilization in a distributed system was first introduced by E. W. Dijkstra in his classic paper [1] in 1974. According to him, a distributed system is self-stabilizing if regardless of any initial global state, the system can automatically adjust itself to eventually converge to a legitimate state and then stay in legitimate state thereafter unless it incurs a subsequent transient fault.

The main work in this paper is to provide a concise proof for the correctness of a commonly used algorithm for finding distances by employing the bounded function technique. The inspiration of using the bounded function in the proof comes from [2].

The rest of this paper is arranged as follows. In Section 2, the algorithm is proposed and the meaning of the legitimate state is explained. In Section 3, an example illustrates the execution of the algorithm. In Section 4, the correctness proof of the algorithm is given.

II. The Algorithm

As usual, we use a connected undirected graph $G = (V, E)$ to model a distributed system, with each node $i \in V$ representing a processor in the system and each edge $\{i, j\} \in E$ representing the bidirectional link connecting processors i and j. Following Dijkstra[1], the system assumes the presence of a *central daemon* who can randomly select one among all the privileged processors to make a move; the central daemon need not be fair in any sense. We should mention here that for the shortest path problem, it suffices to consider only simple graphs, i.e., graphs without any loop and multiple edge.

In the system, each edge $e = \{i, j\}$ is preassigned a *weight* (or *length*) $w(e) = w(i, j)$, which is a positive integer. If $L = (e_1, e_2, \ldots, e_t)$ is a path in G, the *weight* (or *length*) of L, $w(L)$, is defined to be $\sum_{k=1}^{t} w(e_k)$. For any two nodes i and j in V, a *shortest path* between i and j is a path of minimum weight which connects i and j; the weight of a shortest path between i and j is called the *distance* between i and j and is denoted by $d(i, j)$.

The problem of finding distances can be phrased as follows: Suppose a node r in G is specified as the source of the system. We want to find for each node i in G the distance between i and the source r.

The self-stabilizing algorithm for finding distances in the system is given below. Note that in the algorithm, $d(i)$ stands for a local variable of the node i and $N(i) = \{j \in V \mid \{i, j\} \in E\}$ denotes the set of all neighbors of i. The value of each local variable $d(i)$ is in the range $\{0, 1, 2, \ldots\}$.

Self-stabilizing algorithm for finding distances in a distributed system

```
{For the source r}
(R0)  $d(r) \neq 0 \rightarrow d(r) := 0$.
{For node $i \neq r$}
(R1)  $d(i) \neq \min_{j \in N(i)} (d(j) + w(i, j)) \rightarrow d(i) := \min_{j \in N(i)} (d(j) + w(i, j))$.
```

The legitimate states for the system is defined to be those states in which $d(r) = 0$ and $\forall i \neq r$, $d(i) = \min_{j \in N(i)} (d(j) + w(i, j))$. The meaning of the legitimate states can be seen from the following theorem.

Theorem 1: If the system $G = (V, E)$ is in any legitimate state, then $\forall i \in V$, $d(i) = d(i, r)$.

Proof: First, let each node $v \neq r$ selects a neighbor k with $d(k) + w(v, k) = \min_{j \in N(v)} (d(j) + w(v, j))$ to be its *predecessor*, denoted by $p(v)$. Since $d(p(v)) + w(v, p(v)) = \min_{j \in N(v)} (d(j) + w(v, j)) = d(v)$, we have $d(p(v)) + w(v, p(v)) = d(v)$ and $d(p(v)) < d(v)$ for any $v \neq r$. Let $i \neq r$ be any arbitrary node in V. If we trace predecessors from i on, we will get a sequence $(v_0, v_1, v_2, \ldots)$ with $v_0 = i$ and $p(v_k) = v_{k+1}$ for any $k = 0, 1, 2, \ldots$. If the tracing does not reach the source r at any point, then the tracing will continue indefinitely. That means the above sequence is infinite. Since $d(v_k) = d(v_{k+1}) + w(v_k, v_{k+1})$ for any k, we then have $d(i) = d(v_0) > d(v_1) > \cdots > 0$, i.e., we get infinitely many integers between $d(i)$ and 0, which is a contradiction. Therefore the tracing must reach the source r at a certain point and then terminates. Consequently, the sequence $(v_0, v_1, v_2, \ldots)$ is actually a finite one $(v_0, v_1, \ldots, v_t)$ which terminates at $v_t = r$. This sequence of nodes $(v_0, v_1, \ldots, v_t)$ defines a path from $v_0 = i$ to $v_t = r$. The weight (or length)

of the path equals
$w(v_0,v_1)+w(v_1,v_2)+\cdots+w(v_{t-1},v_t)$
$=w(v_0,v_1)+w(v_1,v_2)+\cdots+(w(v_{t-1},v_t)+d(v_t))$
$=w(v_0,v_1)+w(v_1,v_2)+\cdots+(w(v_{t-2},v_{t-1})+d(v_{t-1}))$
$=\cdots=w(v_0,v_1)+d(v_1)$
$=d(v_0)$
$=d(i)$
So we get a path from i to r which has the weight $d(i)$. Therefore, $d(i)\geq d(i,r)$.

Next, we need only to show that $d(i)\leq d(i,r)$ for any i. Let $\{d(i,r)\mid i\in V\}=\{d_0,d_1,\ldots,d_t\}$ with $0=d_0<d_1<\cdots<d_t$. For any node i with $d(i,r)=d_0$, i must be the source r and $d(i)=0$. Thus $d(i)\leq d(i,r)$. Let k be any integer, with $0\leq k<t$. Assume that for any node i with $d(i,r)\leq d_k$, $d(i)\leq d(i,r)$. Then consider any node i with $d(i,r)=d_{k+1}$. Let $(v_1,v_2,\ldots,v_s)$ be a shortest path connecting node i and the source r with $v_1=i$ and $v_s=r$. Then $(v_2,v_3,\ldots,v_s)$ is a shortest path connecting node v_2 and r and $d(v_2,r)<d(i,r)=d_{k+1}$. Therefore $d(v_2,r)\leq d_k$ and we have $d(v_2)\leq d(v_2,r)$ by the induction hypothesis. But then $d(i)=d(v_1)=\min_{j\in N(i)}(d(j)+w(i,j))\leq d(v_2)+w(i,v_2)=d(v_2,r)+w(i,v_2)=d(i,r)$. Thus, we have proved that for any $i\in V$, $d(i)\leq d(i,r)$. Consequently, $d(i)=d(i,r)$ for any $i\in V$. ■

Thus, as is obvious from the above theorem, there is actually only one legitimate state and when the system is in the legitimate state, our problem is solved.

III. An Illustration

Figure 1 illustrates the execution of the algorithm. There are six states in Figure 1. In each state, the shaded nodes represent privileged nodes whereas the shaded node with a darkened circle stands for the privileged node selected by the central daemon to make a move.

IV. Correctness Proof

For the sake of presentation, $(R1)$ is split into two rules:
$(R1\text{-a})$ $d(i)<\min_{j\in N(i)}(d(j)+w(i,j))\rightarrow d(i):=\min_{j\in N(i)}(d(j)+w(i,j))$ and
$(R1\text{-b})$ $d(i)>\min_{j\in N(i)}(d(j)+w(i,j))\rightarrow d(i):=\min_{j\in N(i)}(d(j)+w(i,j))$.

In view of the algorithm, the following Lemma 1 and Lemma 2 are obvious.

Lemma 2: (No deadlock) The system is deadlock-free in each illegitimate state.

Lemma 3: (Closure) No node is privileged when the system is in the legitimate state.

Next, we want to prove the convergence of the algorithm, that is, we want to show: Starting with any initial state, the system will converge to the legitimate state. So for the following discussion, we let the initial state of the system be fixed. For the sake of presentation in the following proofs, we define some terminologies and design three bounded functions. Since the system $G=(V,E)$ is a connected graph, a spanning tree T of G exists. If we choose the source r to be the root, then T becomes a rooted tree. For each node i in the system, let $d_{init}(i)$ be the $d(i)$ in the initial state and let the value $d_u(i)$ be defined recursively by
(1) $d_u(r)=d_{init}(r)$; and
(2) for $i\neq r$, $d_u(i)=\max\{d_{init}(i),\ d_u(p)+w(i,p)\}$, where p is the parent of i in T .

Lemma 4: For each node i in the system, $d(i)\leq d_u(i)$ at any time.

A node $i\neq r$ is called a *turn node* whenever $d(i)<\min_{j\in N(i)}(d(j)+w(i,j))$; otherwise, it is called a *non-turn node*. If i is a turn node and $d(i)=k$ then it is called a *k-turn node*. By definition, $A^{(k)}$ is the set of all k-turn nodes in the system and $t_k=\mid A^{(k)}\mid$ is the cardinality of $A^{(k)}$. Let $m=\max_{i\in V} d_u(i)$ and let $F_1=(t_0,t_1,\ldots,t_m)$, $F_2=\sum_{i\in V}d(i)$, and $F=(F_1,F_2)$. Note that all these functions including $A^{(k)}$, t_k, F_1, F_2 and F have a common domain, the set of all global states. We compare the F_1-values as well as the F-values by lexicographic order. Thus, for any two global states S_1 and S_2, $F_1(S_1)<F_1(S_2)$ if and only if there is a $k\in\{0,1,\ldots,m\}$ such that $t_j(S_1)=t_j(S_2)$ for any $j<k$ and $t_k(S_1)<t_k(S_2)$ whereas $F(S_1)<F(S_2)$ if and only if $F_1(S_1)<F_1(S_2)$ or $[F_1(S_1)=F_1(S_2)$ and $F_2(S_1)<F_2(S_2)]$. Obviously, all the F-values are bounded below by $(0,0,\ldots,0)$ and between any particular F-value and $(0,0,\ldots,0)$, there can be only finitely many F-values possible.

The following lemmas lead eventually to Theorem 12 which claims the convergence of the algorithm. Since the proofs of all these lemmas are of the same spirit, we presented here in details only that of Lemma 8.

Lemma 5: (1) A node which is a turn node right before the system makes a move cannot contribute to the increase of t_k, for any k, after the move of the system.
(2) The source r can not contribute to the increase of any t_k.

Lemma 6: F_1 does not increase each time after rule $R(0)$ is executed in the system.

Lemma 7: F_2 decreases each time after rule $R(0)$ is executed in the system.

Lemma 8: F_1 decreases each time after rule $R(1\text{-a})$ is executed in the system.

Proof: Let node i be the node of the system which executes the rule and let $d(i)=l$ right before the execution.

(1) Since i is an l-turn node right before the execution of rule $R(1\text{-a})$ and is a non-turn node after the execution, i contributes to the decrease of t_l by 1.
(2) By Lemma 4, the source and all those nodes which are turn nodes right before the execution do not contribute to the increase of any t_k after the execution.

(3) If node $j \in V - (N(i) \cup \{i\})$ is a non-turn node right before the execution, then by the same argument as in (2) in the proof of Lemma 5, j remains a non-turn node after the execution and therefore does not affect any t_k.
(4) If node $j \in N(i)$ is any non-turn node right before the execution, then $d(j) \geq \min_{k \in N(j)} (d(k) + w(j,k))$ right before the execution. If $d(i) + w(j,i) > \min_{k \in N(j)} (d(k) + w(j,k))$ right before the execution, then after the execution, $d(i)$ increases and hence $\min_{k \in N(j)} (d(k) + w(j,k))$ remains unchanged; and therefore, $d(j) \geq \min_{k \in N(j)} (d(k) + w(j,k))$ still and j remains a non-turn node. So in this case, j does not affect any t_k. If $d(i) + w(j,i) = \min_{k \in N(j)} (d(k) + w(j,k))$ right before the execution, then $d(j) \geq d(i) + w(j,i) > d(i) = l$; and therefore, after the execution, j either remains a non-turn node or becomes an s-turn node, where $s = d(j) > l$. So in this case, j either does not affect any t_k or can only contribute to the increase of some t_s with $s > l$.

From all above, we can see t_l decreases by at least 1 and t_k remains unchanged for any $k < l$ and therefore F_1 decreases after the execution of rule R(1-a). ■

Lemma 9: F_1 does not increase each time after rule R(1-b) is executed in the system.

Lemma 10: F_2 decreases each time after rule R(1-b) is executed in the system.

Theorem 11: F decreases each time after rule $R(0)$, R(1-a) or R(1-b) is executed in the system.
Proof: Obvious from above Lemmas 5-9. ■

Theorem 12: The algorithm is self-stabilizing.
Proof: The convergence property of the algorithm follows from Lemma 1, Theorem 2, the fact that F is bounded below by $(0, 0, \ldots, 0)$ and the fact that between the initial value and $(0, 0, \ldots, 0)$, there can only be finitely many F-values possible; and the closure property of the algorithm follows immediately from Lemma 2. ■

References

[1] E. W. Dijkstra, Self-stabilizing systems in spite of distributed control, *Communications of the Association of the Computing Machinery*, 17, 643-644, (1974).
[2] S. T. Huang and N. S. Chen, A self-stabilizing algorithm for constructing breadth-first trees, *Information Processing Letters*, 41, 109-117, (1992).

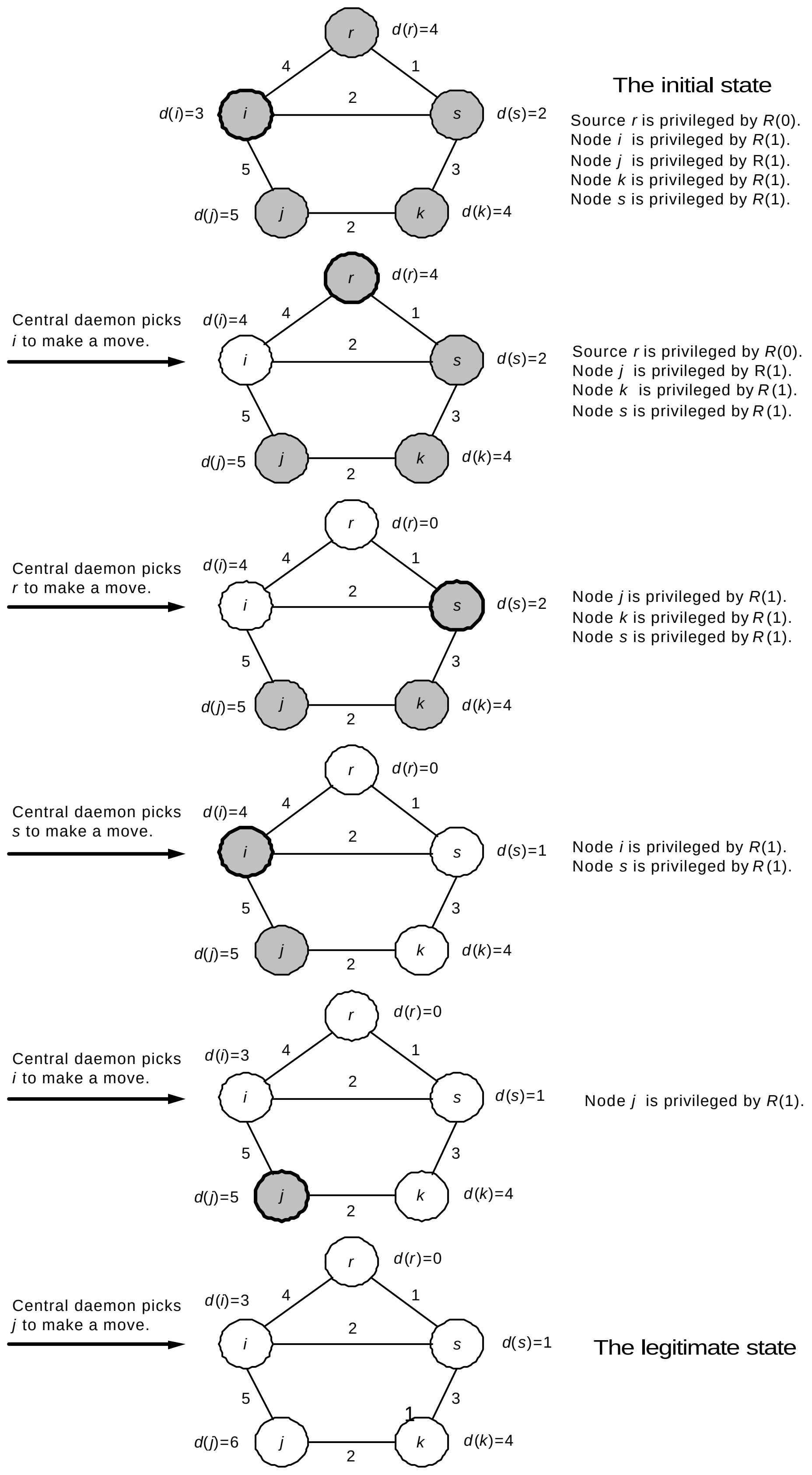

Figure 1. An example which illustrates the execution of the algorithm.