An 'All Pairs Shortest Paths' Distributed Algorithm Using $2n^2$ Messages

S. Haldar

RUU-CS-93-19
May 1993

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

An 'All Pairs Shortest Paths' Distributed Algorithm Using $2n^2$ Messages *

J. Tel

Department of Computer Science
University of Utrecht, P.O. Box 80.089
3508 TB Utrecht, The Netherlands

Abstract

In an execution of a distributed program, processes communicate among themselves by exchanging messages. The execution speed of the program could be expedited by a faster message delivery system, transmitting messages to their destinations through their respective shortest paths. Some distributed algorithms have been proposed in recent years for determining all pairs shortest paths for an arbitrary computer network. The best known algorithm uses $O(n^2 \log n)$ messages, where n is the number of computers in the network. This paper presents a new distributed algorithm for the same problem using $2n^2$ messages in the worst case. This algorithm uses a strategy quite different from those of the other algorithms for the same problem.

Index Terms: All pairs shortest paths, arbitrary weighted network, asynchronous, distributed algorithm, greedy approach, message passing, routing.

1 Introduction

A computer network is an interconnected collection of autonomous computers, referred to as *nodes*. In each computer network, resources (hardware, data, functions) are distributed

*This research is partially supported by NWO through NFI Project ALADDIN under Contract Number NF 62-376, and partially by ESPRIT through Project ALCOM II under Basic Research Action Number 7141.

ISSN: 0924-3275

among the nodes. An application program, also called distributed program, accesses resources from many nodes. To execute such a program, the network system assigns many processes, for the program, at different nodes. We define a *distributed algorithm* for a collection of processes to be a collection of *local algorithms*, one for each process. Each process executes its local algorithm and cooperates with the other processes to achieve the objective of the distributed program. To communicate among themselves, the processes exchange information through *messages*. An underlying message passing system (MPS, in short) handles the exchanging of messages between processes.

In wide area networks the nodes are connected by point-to-point communication links (also referred to as *edges*) for exchanging messages between two nodes, called *neighbors*. Providing edges between all pairs of nodes is not cost effective. Consequently, communication between some pairs is done through intermediate node(s), that is, some messages need to travel through many edges. We assume that both nodes and edges are reliable, that is, the nodes do not fail or show malicious behavior, and the edges do not garble or loose messages and deliver them within some finite delay. Transmitting a message through an edge incurs some cost, that is, each edge has some *weight* assigned to it.

Delivering messages between all pairs of nodes is a primary activity in any computer network. An underlying MPS shoulders this responsibility. Naturally, it is desirable that the MPS routes messages through shortest paths. (A shortest path between a pair of nodes is any path between them with the minimum weight.) Hence, the MPS should know all pairs shortest paths in advance. It uses a *routing table* that stores the all pairs shortest paths information in some coded form.

In this paper we investigate how to build a routing table for an arbitrary network efficiently. There are two broad approaches to building a routing table, namely, centralized and distributed. In the centralized approach, inputs from all nodes are collected in a single node, a sequential algorithm is executed on the inputs, and finally, outputs are dispersed to all the nodes. There are two lucrative strategies in developing sequential algorithms, namely the greedy method [5] and the dynamic programming [2]. The single node all shortest paths algorithm of Dijkstra [4] uses a greedy method, and requires $O(n^2)$ time units, where n is the number of nodes in the network. This algorithm can be executed separately for each node to obtain all pairs shortest paths in $O(n^3)$ time units. The all pairs shortest paths algorithm of Floyd and Warshall [6, 10] uses a dynamic programming

strategy, taking $O(n^3)$ time units. Although a centralized approach is simple, its message complexity is $\Theta(mn)$, where m is the number of edges in the network. (There are various complexity measures for a distributed algorithm: message complexity, bit complexity, time complexity, space complexity. We restrict our attention to message complexity, namely the number of messages exchanged in the algorithm in the worst case.)

In the distributed approach, each node builds its part of the routing table, and helps other nodes in building their parts. One such algorithm is presented by Toueg [9], based on the Floyd-Warshall sequential algorithm. It assumes edge weights to be positive, and each node knows identities of all the nodes, identities of its neighbors and weights of its incident edges. This algorithm exchanges $O(n)$ messages per edge, $O(mn)$ in total. Recently, Afek and Ricklin [1] have proposed a transformation scheme for distributed algorithms. It takes a distributed algorithm whose message complexity is $O(f \cdot m)$, where f is an arbitrary function of m and n, and produces a new *semi-distributed* algorithm for the same problem with $O(f \cdot n \log n + m \log n)$ message complexity. Applying this scheme (for example, to the Segall algorithm [7]) an all pairs shortest paths algorithm with upper bound of $O(n^2 \log n)$ message complexity can be obtained.

In this paper we present a new fully distributed algorithm that uses $2n^2$ messages in the worst case. It is based on the single node all shortest paths sequential algorithm of Dijkstra. We first modify his algorithm and make it more efficient by a constant factor, and then transform the modified algorithm into a distributed algorithm. In this distributed algorithm we allow communication only between neighbors, and this leads to lower message complexity. This algorithm uses a quite different strategy than those used by others for the same problem. In this algorithm, a node might finish building its part of the routing table much earlier than the others do; if the former node sends messages to any nodes, the messages are guaranteed to be delivered to their destinations through their respective shortest paths. This delivery is assured even if the routing table is partially built.

In Section 2 we define a model of the system under investigation. This section states some definitions and makes some assumptions. Dijkstra's sequential algorithm is presented in Section 3. His algorithm is modified in Section 4. We transform the modified algorithm into a distributed algorithm in Section 5. Section 6 concludes the paper.

2 Model, Definitions and Assumptions

We represent a computer network, where computers are connected by point-to-point communication links, by an *undirected weighted graph* $G = \langle V, E, W \rangle$, where V is a set of *nodes*, one for each computer; E is a set of *edges*, one for each link; and W is a *weight function* from E to non-zero positive real numbers. The number of nodes is denoted n, and the number of edges m. An edge in E is denoted by an unordered pair of nodes from V. If $u, v \in V$ are connected by an edge, we denote that edge as uv (and vu). An edge $e = uv \in E$ is called an *incident edge* of u (and of v), and u and v are *neighbors*, and the *weight* of e is $W[u, v]$. The weight of uv is equal to the weight of vu. A *path* between two nodes u and v is a sequence $p = \langle u = v_0, v_1, v_2, \cdots, v_k = v \rangle$ of nodes such that for each $0 \leq i < k$, $v_i v_{i+1} \in E$, and the weight of the path is $w(p) = \sum_{0 \leq i < k} W[v_i, v_{i+1}]$. The notation $u \overset{p}{\rightsquigarrow} v$ denotes that v is reachable from u through the path p. In this paper we assume that G is connected, that is, there exists a path between each pair of nodes. The *shortest path weight*, also called *distance*, between any pair of nodes u and v, denoted $dist(u, v)$, is the minimum weight of all possible paths between them, that is, $dist(u, v) = \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}$. A *shortest path* from a node u to another node v is defined as any path p with $w(p) = dist(u, v)$. We also like to determine the neighbor of u in p, denoted $via(u, v)$, through which messages for v will be routed.

We assume, as in [9], that each node u knows: (1) the graph size n, (2) identities of all nodes, $1, 2, \cdots, n$, (3) identities of its neighbors in *neighbor(u)* and (4) weights of its incident edges, $W[u, v]$, $uv \in E$. It is assumed that the weights of the edges do not change with time. Messages are delivered to their respective destinations within a finite delay, but they might be delivered out-of-order, that is, the edges are non FIFO. The distributed algorithm presented in this paper allows communication only between neighbors. We assume an asynchronous message passing system, that is, a sender of a message does not (in fact, must not) wait for the receiver to be ready to receive the message. Messages are stored in the edges until they are delivered.

The proposed algorithm uses a table called *routing table* (*RT*, in short) to store the final output of all pairs shortest paths and intermediate results of the computation in some coded form. The algorithm uses two types of messages (see later), thereby requiring one additional bit to distinguish them.

The following two properties hold for any graph $G = \langle V, E, W \rangle$.

Property 1 ([3], Lemma 25.1) *Subpaths of shortest paths are shortest paths.* □

Property 2 ([3], Corollary 25.2) *Let s and v be two nodes in V. Suppose a shortest path p from s to v can be decomposed into $s \overset{p'}{\rightsquigarrow} u \xrightarrow{uv} v$ for some node $u \in neighbor(v)$ and path p'. Then, the weight of a shortest path from s to v is $dist(s, v) = dist(s, u) + W[u, v]$.* □

3 Dijkstra's Single Node All Shortest Paths Algorithm

Shortest path algorithms typically exploit the property that a shortest path between two nodes contains other shortest paths within it (Property 1). This optimal substructure property invites the use of the dynamic programming [2] or the greedy approach [5]. The idea of a greedy approach is to make each choice in a locally optimal manner. However, it is not always easy to tell whether a greedy approach will be effective. In the case of the shortest paths problem, a greedy approach has been effective.

The Dijkstra algorithm for a graph $G = \langle V, E, W \rangle$ is given in Figure 1. For a pivotal node $s \in V$, it calculates the distance $dist(s, v)$ for all $v \in V$. For each node v, it maintains a distance estimate variable $d[v]$, initialized to ∞, which is an upper bound on the weight of a shortest path from s to v. The invariant $d[v] \geq dist(s, v)$ is maintained throughout the execution of the algorithm. The technique used by the Dijkstra algorithm is relaxation, a method that repeatedly decreases (relaxes) the upper bound until it becomes equal to the actual shortest weight. It maintains a set S of nodes whose distances from the pivotal node s have already been determined. That is, for all $v \in S$, we have $d[v] = dist(s, v)$. At the beginning, $d[s] = 0 = dist(s, s)$. It maintains a priority queue Q to contain all nodes in $V - S$. It repeatedly selects the node $u \in Q$ with the minimum distance estimate $d[u]$; inserts u in S; and relaxes the distance estimates for all the neighbors of u. That is, for each $uv \in E$, the distance estimate $d[v]$ is relaxed with respect to $d[u] + W[u, v]$. In his algorithm each edge is considered exactly twice to relax some distance estimates, that is, a total of $2m$ relaxations. The relaxations cause the distance estimates to decrease monotonically until the estimates become equal to the actual distances. In the Dijkstra algorithm, it is

guaranteed that when a node u is visited[1], at that time $d[u] = dist(s, u)$.

For correctness proof, please refer to [3](Theorem 25.10). The Dijkstra algorithm takes $O(n^2)$ time unit if Q is a linear array, $O(m \log n)$ time units if Q is a binary heap, and $O(n \log n + m)$ time units if Q is a Fibonacci heap.

4 Modified Algorithm

In the Dijkstra algorithm, when a node u is removed from Q and put in S, distance estimates of all the neighbors of u are relaxed. Let uv be an edge. If the node v is already in S, there is no need to relax $d[v]$ with respect to $d[u] + W[u, v]$, for the optimal distance $dist(s, v)$ has already been found. To avoid this relaxation, we attach a boolean tag $status[v]$ with each node v, initialized to *tentative*, and made *permanent* on reaching the shortest weight for v. At the time of visiting u, the edge uv will be considered only if v is *tentative*. The modified algorithm is given in Figure 2. In the modified algorithm Q represents a set of frontier nodes during the execution of the algorithm. All nodes in Q are *tentative*. At the beginning all incident edges of the pivotal node s are put in Q. For node v in Q, $d[v]$ is the tentative distance from s to v. The algorithm repeatedly selects the node $u \in Q$ with the minimum distance estimate $d[u]$; makes u permanent (i.e., visits u); and relaxes distance estimates of some neighbors of u. In the modified algorithm there are m relaxations and $2m$ boolean tests, whereas in the Dijkstra algorithm there are $2m$ relaxations. So, the modified algorithm speeds up the computation by a constant factor if the cost of boolean test is less than half the cost of addition and real comparison. Each node v is inserted in Q exactly once, that is, a total of n insertions in Q. In each iteration of the while loop, exactly one node u is removed from Q, and u is made *permanent*. (At that moment $d[u]$ is the shortest weight $dist(s, u)$.) So the loop terminates after n iterations. The variable $via[v]$ indicates the neighbor of s through which messages for v should be routed. The correctness proof of this algorithm is very similar to that of Dijkstra's, and omitted from the paper.

[1]By visiting u, we mean that the algorithm is inserting u in S and considering all its incident edges for some distant estimate relaxations.

5 Distributed 'All Pairs Shortest Paths' Algorithm

We first consider a sequential all pairs shortest paths algorithm. The modified algorithm given in Figure 2 could be executed for all nodes of G to obtain the all pairs shortest paths. But, we can do better using the following observation. Property 1 states that a shortest path contains other shortest paths within it. Let a shortest path p, between two nodes u and v, run through a node $v' \in neighbor(u)$. Then $dist(u,v) = W[u,v'] + dist(v',v)$ (by looking Property 2 from different angle), and we state this as Property 2'.

Property 2' *Suppose a shortest path p from a node u to another node v can be decomposed into $u \xrightarrow{uv'} v' \overset{p'}{\rightsquigarrow} v$ for some node $v' \in neighbor(u)$ in p and path p'. Then, the weight of a shortest path from u to v is $dist(u,v) = W[u,v'] + dist(v',v)$.* □

Now, instead of calculating both $dist(u,v)$ and $dist(v',v)$ separately and independently, we calculate $dist(v',v)$ first, and then $dist(u,v)$ by one addition operation. But, the question is, how does u know that p runs through v'? We answer this question in the following subsection. We present a sequential restricted version of the modified algorithm, for all pairs shortest paths in Section 5.1 first, and then, transform the Restricted algorithm into a distributed algorithm in Section 5.2. (We call the sequential algorithm 'Restricted' because in the algorithm each node accesses the table entries of only its neighbors.)

5.1 Restricted algorithm

Assume RT is the routing table to be built for a given graph $G = \langle V, E, W \rangle$. Each component of the table, $RT[u,v]$, for $u, v \in V$, has four fields: (1) *weight*, (2) *status*, (3) *via* and (4) *via-status*. Here, *weight* is distance estimate between u and v; *status* indicates whether u has visited (directly or indirectly) v; *via* is the neighbor of u in the path used to determine the *weight*; and *via-status* indicates u's knowledge of v's *status* in *via*. The algorithm is given in Figure 3. (In the text, a subcomponent of the routing table, $RT[u,v].field$ is denoted as $field[u,v]$.) The RT table is initialized as follows for all nodes u: (1) $weight[u,u] := 0$, $status[u,u] :=$ *permanent*, $via[u,u] := u$ and $via\text{-}status[u,u] :=$ *permanent*; (2) for all neighbors v of u, $weight[u,v] := W[u,v]$, $status[u,v] :=$ *tentative*, $via[u,v] := v$ and $via\text{-}status[u,v] :=$ *tentative*; and (3) for non adjacent nodes, *weights* are ∞, and *status* and *via-status* values are *tentative*. After the initialization, a repeat-until loop is executed. In each iteration of the loop the following steps are performed for all nodes u (one can

imagine that n processors, one for each u, are executing the algorithm synchronously).

Step 1: The node u selects a node v_u with minimum distance estimate $weight[u, v_u]$ from $RT[u, *]$ such that $status[u, v_u] = tentative$. If no tentative v_u is available, u has finished building up its part $RT[u, *]$ of the routing table, and hence, it does not do any effective thing. Otherwise, u executes Step 2.

Step 2: Let the node v_u be reached through a neighbor node $via_u = via[u, v_u]$. If $via\text{-}status[u, v_u]$ is *permanent* — that is, the node via_u has already visited the node v_u and calculated $dist(via_u, v_u)$, and this visit is also captured in $RT[u, *]$, then $weight[u, v_u] = dist(u, v_u)$ (by Property 2') — then u makes $status[u, v_u]$ *permanent*, and loops back to Step 1. Otherwise, it executes Step 3.

Step 3: It checks the table entry $RT[via_u, v_u]$. If $status[via_u, v_u]$ is not *permanent*, — the node via_u has not yet visited the node v_u — then u does nothing and loops back to Step 1. Otherwise, it copies the table entries $RT[via_u, *].\langle weight, status\rangle$ in a local variable $temp_u$ first, and then executes Step 4.

Step 4: The first thing u does is that it makes $status[u, v_u]$ and $via\text{-}status[u, v_u]$ *permanent*. (The node u indirectly visits v_u through the visit of v_u by via_u.) Finally, u updates its other table entries whose *status* are not *permanent*. Consider a node v'_u such that $status[u, v'_u] = tentative$. If u finds $weight[u, v'_u] > W[u, via_u] + temp_u[v'_u].weight$ (note that $temp_u[v'_u].weight$ is the distance estimate between the nodes via_u and v'_u), then the $weight[u, v'_u]$ is set to $W[u, via_u] + temp_u[v'_u].weight$, $via[u, v'_u]$ to via_u, and $via\text{-}status[u, v'_u]$ to $temp_u[v'_u].status$. Loops back to Step 1.

It is to be noted that we can use the same algorithm without *via-status* fields and Step 2. However, they are included to speed up the computation. Note that, in Step 2, when $via\text{-}status[u, v_u]$ is *permanent*, it indicates that the node via_u has visited the node v_u, and the effects of this visit is already captured in $RT[u, *]$. So, if u finds $via\text{-}status[u, v_u]$ *permanent*, it need not access the table entries in $RT[via_u, *]$ to update its table entries $RT[u, *]$, and hence executing the Step 2 would speed up the computation. The repeat-until loop is executed until no change in RT is observed in an iteration, that is, all nodes have built up their respective parts of RT. On termination the *via* fields indicate how messages should be routed in the graph G. The correctness proof is given in the next subsection.

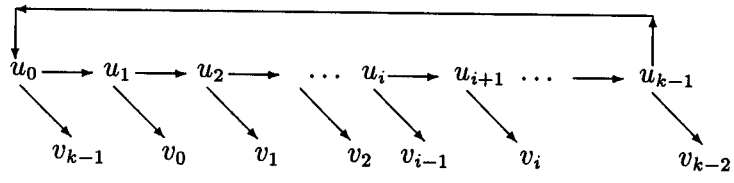
5.1.1 Correctness proof

We need to show that the Restricted algorithm satisfies two basic properties, namely, (1) *liveness*, that is, the algorithm terminates eventually, and (2) *safety*, that is, on termination of the algorithm, all pairs shortest paths are determined. To show the liveness property, we first show that the algorithm does not lead to any deadlocks, and then, the termination of the algorithm. The safety property is proved at last in Theorem 8.

Lemma 3 *There is no deadlock.*

Proof: On the contrary, let us assume that there is a deadlock when the algorithm is executed for a graph $G = \langle V, E, W \rangle$. Note that for all $uv \in E$, $W[u, v] > 0$.

Let $u_0, u_1, \cdots, u_k = u_0$ be a sequence of nodes involved in a deadlock. As the algorithm allows each node u to access table components for its neighbors only, then for $0 \leq i < k$, $u_i u_{i\oplus 1} \in E$, where $\oplus$ is modulo k addition operator. Let u_i be trying to make $status[u_i, v_i]$ *permanent* for some $v_i \in V$, and be waiting for $status[u_{i\oplus 1}, v_i]$ to become *permanent*. That is, the node u_i finds shortest path for v_i through $u_i u_{i\oplus 1}$. (See the following picture. Note that $\ominus$ is modulo k subtraction operator.)



As u_i is trying to make $status[u_i, v_i]$ *permanent* (and u_i has not made $v_{i\ominus 1}$ *permanent* yet in this deadlock situation), we have $weight[u_i, v_{i\ominus 1}] \geq weight[u_i, v_i]$. Note that u_i got the information of v_i through $u_{i\oplus 1}$. Hence, $weight[u_i, v_i] = W[u_i, u_{i\oplus 1}] + weight[u_{i\oplus 1}, v_i]$. Thus we get, $weight[u_0, v_{k-1}] \geq weight[u_0, v_0] = W[u_0, u_1] + weight[u_1, v_0]$. That is, $weight[u_0, v_{k-1}] \geq W[u_0, u_1] + weight[u_1, v_1]$, and hence, $weight[u_0, v_{k-1}] \geq W[u_0, u_1] + W[u_1, u_2] + weight[u_2, v_1]$; and so on. Finally, we have $weight[u_0, v_{k-1}] \geq W[u_0, u_1] + W[u_1, u_2] + \cdots + W[u_{k-1}, u_0] + weight[u_0, v_{k-1}]$. As W's are non-zero positive, we have $weight[u_0, v_{k-1}] \geq$ *some-positive-number* $+ weight[u_0, v_{k-1}]$. But, a number cannot be greater than itself, and hence, the sequence $u_0, u_1, \cdots, u_k = u_0$ cannot be in a deadlock. This proves the lemma. □

Corollary 4 *In each iteration of the repeat-until loop, at least one entry in the routing table RT is made permanent.* □

Lemma 5 *All table entries become permanent within $n^2 - n$ iterations.*

Proof: At the beginning there are $n^2 - n$ tentative entries in RT. By Corollary 4, at least one entry becomes permanent in each iteration. Hence, all entries become permanent within $n^2 - n$ iterations. □

Corollary 6 *The algorithm terminates within $n^2 - n$ iterations.* □

Now, before showing the safety property we prove an important lemma. If the algorithm selects p as a shortest path between two nodes u and v, then it also selects for all v' in p the subpath of p between u and v' as shortest path between u and v', and the subpath of p between v' and v as shortest path between v' and v.

Lemma 7 *Suppose a node u makes $status[u, v]$ permanent for a node v through a path p. Then, for all the nodes v' in p,*

(a) u has made $status[u, v']$ permanent through a subpath of p between u and v', and

(b) v' has made $status[v', v]$ permanent through a subpath of p between v' and v.

Proof:

(*a*) Suppose not. Let v' be a node in p such that $status[u, v']$ is not *permanent*. As v' lies in p, we have $weight[u, v'] < weight[u, v]$. Since, both $status[u, v]$ and $status[u, v']$ are tentative, u considers v' before v to make $status[u, v']$ *permanent*. The assertion follows.

(*b*) Let p be the sequence $\langle u = u_0, u_1, u_2, \cdots, u_k = v \rangle$ of nodes. The node u makes $status[u, v]$ *permanent* only if it finds $status[u_1, v] = permanent$; u_1 makes $status[u_1, v]$ *permanent* only if it finds $status[u_2, v] = permanent$; and so on. Ultimately, u_{k-1} makes $status[u_{k-1}, v]$ *permanent* only if it finds $status[v, v] = permanent$. At the beginning, in the initialization phase, the node v makes its $status[v, v]$ *permanent*. The assertion follows. □

Theorem 8 *When the algorithm terminates for a graph $G = \langle V, E, W \rangle$, for all $u, v \in V$, $weight[u, v] = dist(u, v)$.*

Proof: Suppose not. Consider any two nodes $u, v \in V$, for which, on termination, $weight[u,v] \neq dist(u,v)$. It is clear from the Restricted algorithm that for any two nodes $u, v \in V$, the $weight[u,v]$ is not changed after the entry $status[u,v]$ is made *permanent*. Then, $weight[u,v] \neq dist(u,v)$ at that time u makes $status[u,v]$ *permanent*.

Without loss of generality, we assume v is the first node whose $status[u,v]$ is made *permanent* by u when $weight[u,v] \neq dist(u,v)$. Let p be the path chosen by u to reach v. Let p be the sequence $\langle u = u_0, u_1, u_2, \cdots, u_k = v \rangle$ of nodes.

As $weight[u,v] \neq dist(u,v)$ at that time u makes $status[u,v]$ *permanent*, the path p is not a shortest path. Let $p' = \langle u = u'_0, u'_1, u'_2, \cdots, u'_{k'} = v \rangle$ be a shorter path from u to v. We have $w(p') < w(p)$. Then, $W[u,u'_1] = weight[u,u'_1] < weight[u,v]$, and hence, u must make $status[u,u'_1]$ *permanent* before making $status[u,v]$ *permanent*. When u makes $status[u,u'_1]$ *permanent*, it relaxes the distance estimate $weight[u,u'_2]$ with respect to $W[u,u'_1]+W[u'_1,u'_2]$. Thus, u becomes aware of the fact that $weight[u,u'_2] < weight[u,v]$, and hence, u must make $status[u,u'_2]$ *permanent* before making $status[u,v]$ *permanent*; and so on. Eventually, u makes $status[u,u'_{k'-1}]$ *permanent* before making $status[u,v]$ *permanent*. By virtue of the choice of v to be the first node whose $status[u,v]$ is made *permanent* when $weight[u,v] \neq dist(u,v)$, we have $weight[u,u'_{k'-1}] = dist(u,u'_{k'-1})$. Let $p'' = \langle u = u''_0, u''_1, u''_2, \cdots, u''_{k''} = u'_{k'-1} \rangle$ be the shortest path chosen by u to reach $u'_{k'-1}$. Then, by Lemma 7, for all u''_j in p'', $status[u,u''_j]$ and $status[u''_j,u'_{k'-1}]$ are permanent. The node u can make $status[u,u'_{k'-1}]$ *permanent* only if $status[u''_1,u'_{k'-1}]$ is *permanent*. The node u''_1 can make $status[u''_1,u'_{k'-1}]$ *permanent* only if $status[u''_2,u'_{k'-1}]$ is *permanent*; and so on. At the beginning, in the initialization phase, $status[u'_{k'-1},u'_{k'-1}]$ is made *permanent*. When $u''_{k''-1}$ makes $status[u''_{k''-1},u'_{k'-1}]$ *permanent*, it relaxes the distance estimate $weight[u''_{k''-1},v]$ with respect to $W[u''_{k''-1},u'_{k'-1}]+W[u'_{k'-1},v]$. When $u''_{k''-2}$ makes $status[u''_{k''-2},u'_{k'-1}]$ *permanent*, it relaxes the distance estimate $weight[u''_{k''-2},v]$ with respect to $W[u''_{k''-2},u''_{k''-1}] + weight[u''_{k''-1},v]$, that is, with respect to $W[u''_{k''-2},u''_{k''-1}] + W[u''_{k''-1},u'_{k'-1}] + W[u'_{k'-1},v]$; and so on. When u makes make $status[u,u'_{k'-1}]$ *permanent*, it relaxes the distance estimate $weight[u,v]$ with respect to $W[u,u''_1] + W[u''_1,u''_2] + \cdots + W[u''_{k''-1},u'_{k'-1}] + W[u'_{k'-1},v]$. So, u becomes aware of a shorter path than p to reach v. Hence u would not consider the path p, contradicting the assumption that it has chosen p, and hence $weight[u,v] = dist(u,v)$. The theorem follows. □

It has been a mirage for the author to determine the time complexity of the Restricted algorithm. However, doing that does not provide much insight about how to transform this algorithm into a distributed algorithm, and hence is omitted.

5.2 The distributed algorithm

It is to observe that in the Restricted algorithm a node u accesses the RT table entries of its neighbors, and not of any other node in the graph.

In the distributed algorithm, the routing table part $RT[u, *]$ is stored at node u, and the same Restricted algorithm is executed at each node u. The distributed algorithm is given in Figure 4. Here, one point is of particular interest. In the Restricted algorithm of Section 5.1, in Step 3 if a node u finds $status[via_u, v_u]$ is *tentative*, it does nothing; and in the very next iteration it checks the same *status* again. The node u does the checking until the *status* becomes *permanent*, and then, u takes a normal course of actions. In the distributed algorithm, to get the status information for v_u from the neighbor node via_u, u sends a message $\langle$*give-me your table-entries* for $v_u\rangle$ to the node via_u. Instead of sending a reply containing the $RT[via_u, *].\langle weight, status\rangle$ immediately back to u, the node via_u defers replying until $status[via_u, v_u]$ becomes *permanent*. Since the algorithm does not lead to a deadlock situation, as proved in Section 5.1.1, the node via_u eventually returns the reply message to u. On receiving the reply message from via_u, u makes $status[u, v_u]$ and *via-status*$[u, v_u]$ *permanent*, and updates the other tentative entries in $RT[u, *]$. Thus, to make one table entry status *permanent*, there is an exchange of at most two messages between two neighbors. At the beginning, the entire routing table has $(n^2 - n)$ tentative entries. Hence, the algorithm exchanges at most $2(n^2 - n)$ messages. On termination of the algorithm the *via* fields indicate how messages should be routed in the graph. The correctness proof of the distributed algorithm is very much similar to that of the Restricted algorithm, and hence omitted.

6 Conclusion

A fully distributed algorithm to find all pairs shortest paths for an arbitrary connected network has been presented. The algorithm produces the desired output within $2n^2$ messages, where n is the number of nodes in the network. Another achievement in the algorithm

is that the message complexity is insensitive to the number of edges in the network. The algorithm allows the communication only between neighbors, and uses two types of messages. The algorithm works with each edge's storage capacity of only two messages for each direction.

In an execution of the algorithm some node u could finish building up its part of the routing table much before the other nodes could do their parts, but they have partially built up their parts. Note that the shortest paths from u to other nodes form a spanning tree rooted at u. When u finishes its execution, the spanning tree for u is clearly identified through the *via* fields of the internal nodes in the spanning tree. It is guaranteed that the internal nodes will not change the respective *via* fields through which the messages from u will be transmitted. Hence, it is guaranteed that if u starts sending messages to any nodes, the messages will be delivered to their destinations through their respective shortest paths. This can be done even if some nodes have not finished executing the algorithm completely.

For the sake of simplicity of the presentation, a number of assumptions have been made in Section 2. Some of the assumptions could be relaxed, and the following properties can be obtained by modifying the algorithm a little. If the network size is known to the nodes, a node does not need to exchange messages with its neighbor when it is left with only one node to make *permanent*, leading to a total savings of $2n$ messages. The algorithm works even if the edge weights are different for different directions. The algorithm could easily be modified to correctly build up the routing table in case the network size is unknown to the nodes. The algorithm works even if the network is not connected.

Finally, it is conjectured that the message complexity is a tight bound, i.e., $\Theta(n^2)$, for the model considered in this paper.

Acknowledgment

I wish to thank Prof. Jan van Leeuwen who introduced me to the area of routing algorithms. Dr. Gerard Tel has been very cooperative during this work, and provided some useful comments on an earlier draft. His book [8] and the one in [3] have been very helpful in preparing this manuscript. I especially thank Prof. K. Vidyasankar who provided some useful comments and many suggestions to simplify the proof of the safety property of the Restricted algorithm.

References

[1] Y. Afek and M. Ricklin, 'Sparser: A paradigm for running distributed algorithms', Workshop on Distributed Algorithms, LNCS 647, 1992, 1–10. (Also in J. of Algorithms, Vol.14(2), 1993, 316–328.)

[2] R. Bellman, 'Dynamic programming', Princeton University Press, 1957.

[3] T.H. Cormen, C.E. Leiserson and R.L. Rivest, 'Introduction to algorithms', MIT Press, Second printing 1990 (Original 1989).

[4] E.W. Dijkstra, 'A note on two problems in connection with graphs', Numerische mathematik, Vol.1, 1959, 269–271.

[5] J. Edmonds, 'Matroids and the greedy algorithm', Mathematical Programming, Vol.1, 1971, 126–136.

[6] R.W. Floyd, 'Algorithm 97 (Shortest path)', Communications of the ACM, Vol.5(6), 1962, 345.

[7] A. Segall, 'Distributed network protocols', IEEE trans. on Information Theory, Vol.29(1), 1983, 23–35.

[8] G. Tel, 'Introduction to distributed algorithms', INF/DOC-92-05, Department of Computer Science, University of Utrecht, The Netherlands, 1992.

[9] S. Toueg, 'An all-pairs shortest-paths distributed algorithm', Tech Rep RC 8327, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA, 1980.

[10] S. Warshall, 'A theorem on boolean matrices', Journal of the ACM, Vol.9(1), 1962, 11–12.

```
Procedure Dijkstra(G = ⟨V, E, W⟩, s);
begin
    for all v ∈ V do d[v] := ∞;                          {initialize}
    d[s] := 0;
    S := {s};
    Q := V;
    while Q ≠ {} do
        remove u from Q with d[u] minimum;               {Extract-min from Q}
        S := S ∪ {u};
        for all v ∈ neighbor(u) do                       {relax distance estimates d[v]}
            if d[v] > d[u] + W[u, v] then
                d[v] := d[u] + W[u, v];
    endwhile;
end; {of procedure}
```

Figure 1: Dijkstra algorithm.

```
Procedure Modified-Dijkstra-Algorithm(G = ⟨V, E, W⟩, s);
begin
    for all v ∈ V do                                     {Initilize}
        d[v] := ∞;
        status[v] := tentative;

    d[s] := 0;
    status[s] := permanent;                              {visit s}
    via[s] := s;
    Q := {};

    for all v ∈ neighbor(s) do                           {insert all incident edges of s in Q}
        d[v] := W[s, v];
        via[v] := v;
        insert v in Q;

    {Q contains frontier tentative nodes to be visited}
    while Q ≠ {} do
        remove u from Q with d[u] minimum;               {Extract-min from Q}
        status[u] := permanent;                          {visit u; now d[u] = dist(s, u)}
        for all v ∈ neighbor(u), such that status[v] = tentative do   {expand frontier nodes}
            if d[v] > d[u] + W[u, v] then                {relax the edge uv}
                d[v] := d[u] + W[u, v];
                via[v] := via[u];
                if v ∉ Q then insert v in Q;
            endif;
        endfor;
    endwhile;
end; {of procedure}
```

Figure 2: Modified algorithm.