

COMMUNICATION COMPLEXITY OF DISTRIBUTED SHORTEST
PATH ALGORITHMS

by

Daniel U. Friedman

S.B., Rice University, Houston, Texas (1976)

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December, 1978

Signature of Author.....
Department of Electrical Engineering and Computer
Science, December 20, 1978

Certified by.....
Thesis Supervisor

Accepted by.....
Chairman, Departmental Committee on Graduate Students

COMMUNICATION COMPLEXITY OF DISTRIBUTED SHORTEST PATH
ALGORITHMS

by

Daniel U. Friedman

Submitted to the Department of Electrical Engineering and
Computer Science in Partial Fulfillment of the Requirements
for the Degree of Master of Science, December 20, 1978.

ABSTRACT

One routing strategy frequently used in computer networks assigns traffic dependent distances to the links of the network and then increases the traffic flow on shortest paths. If a central facility monitors all network traffic, classical algorithms can be readily employed to compute shortest paths. If traffic is only locally monitored, we wish to have distributed procedures in which the nodes begin with only local information and compute shortest paths by communicating with one another. In this thesis, we present several such distributed shortest path algorithms and analyze their communication cost. Since the transmission of control information required for network operation reduces the bandwidth available to users, we concentrate on finding algorithms that use a minimum of information exchange.

Thesis Supervisor: Robert G. Gallager

Title: Professor of Electrical Engineering and Computer Science
and Associate Director of Laboratory for Information and
Decision Systems

ACKNOWLEDGEMENTS

I take this opportunity to thank Prof. Robert Gallager for introducing me to this thesis topic and for supervising the thesis work. His willingness to listen and think about the problems made the research rewarding. I also express my appreciation to Prof. Pierre Humblet for his interest, advice, and willingness to discuss many problems at length. In addition, I wish to acknowledge several helpful discussions that I had with Prof. Jim Massey while he was at MIT, and with my fellow graduate student Mr. Paris Kanellakis.

The manuscript was typed by Ms. Fifa Monserrate, and I thank her for her patience and a job well done.

The research was carried out the MIT Laboratory for Information and Decision Systems with support from the John and Fannie Hertz Foundation and the Vinton Hayes Fund of MIT, and I gratefully acknowledge their aid.

Finally, I take this opportunity to thank my parents for their encouragement and support. Without this, I probably would have had a job by now. Wovon man nicht sprechen kann, darüber muss man schweigen.

TABLE OF CONTENTS

I. INTRODUCTION	1
I.1 Motivation	1
I.2 Notation and Definitions	4
I.3 Problem Formulation	7
I.4 Shortest Path Trees	9
II. CENTRALIZED ALGORITHMS FOR THE SHORTEST PATH PROBLEM	13
II.1 Single Source Problem	13
II.2 All Pairs Problem	23
II.3 Lower Bounds	25
III. DISTRIBUTED SHORTEST PATH ALGORITHMS	27
III.1 More Notation and Assumptions	27
III.2 Minimum Hop Path Algorithms	30
III.3 Broadcasting All Arclengths	41
III.4 Worst Case Analysis of Some Distributed Shortest Path Algorithms	45
III.5 Preprocessing and Message Encoding	58
IV. AVERAGE COMMUNICATION COST ANALYSIS	62
IV.1 Motivation	62
IV.2 Average Computation Analysis of Spira's Algorithm	66
IV.3 Applications to Distributed Algorithms	73

TABLE OF CONTENTS

(continued)

V. SUGGESTIONS FOR FURTHER RESEARCH 88

APPENDIX A 90

BIBLIOGRAPHY 92

I. INTRODUCTION

I.1 Motivation

One routing strategy frequently used in computer networks assigns traffic dependent lengths to the links of the network and then apportions the traffic among the various paths depending on their relative distances. (The distance of a path is the sum of the lengths of its links). Periodically, the lengths are updated to reflect changes in the traffic, path distances are recomputed, and traffic is appropriately rerouted. Thus, the strategy is somewhat adaptive. Implementation of this strategy requires that four basic subproblems be addressed.

- 1) Dependence of link lengths on the traffic
- 2) Computation of path distances
- 3) Allocation of traffic
- 4) Frequency of updating.

Parts 1) and 3) are generally handled by choosing some cost function, assigning to each link a length that reflects the cost of using it, and then routing traffic so as to minimize the total cost. Methods for doing these things involve considerations of multicommodity flow problems and the statistics of network traffic. They present formidable problems because

- a) it is not always obvious how to choose a meaningful (in terms of network performance) cost function and,
- b) given a cost function, it is not easy to see how to assign link distances in an appropriate manner and,

- c) given the link distances, the optimal traffic allocation pattern can be difficult to find.

Much of the previous work done on routing has focused on these problems and the reader is referred to [1] and [2] for representative studies. Limitations of the computational resources directly affect 2) and hence the whole strategy. Since the total number of paths can grow exponentially with the number of nodes, it is often not feasible to compute all path distances. Frequently, only the shortest paths are computed and the amount of traffic flow allocated to these paths is increased relative to the old values. The frequency with which updates can be performed depends on how much of the network's resources the updating procedure uses.

If a central facility monitors all network traffic, 2) is purely a computational problem, and there are several well known algorithms for computing shortest paths. If traffic is only locally monitored, these algorithms can still be used provided all other nodes send their local information to the central computer. This is a communication problem. In either case, the network is completely dependent on the central facility for purposes of routing. Hence, it is desirable to have procedures in which the nodes begin with only local information and compute path distances by communicating with one another. If a node or link is not functioning, it is automatically excluded from the update, but other nodes proceed with the remaining links.

This thesis presents and analyzes several such distributed shortest path algorithms, and hence bears on part 2) of this strategy. The link lengths are assumed to be already assigned and are taken as part of the input. Each node initially knows the lengths of its outgoing arcs and possibly some things about the topology of the network. It finds shortest paths to other nodes by communicating with them. It is assumed that the arc lengths and topology are all fixed during execution. The problem of routing in the presence of failures is indeed significant, but here we are only concerned with the amount of information that the nodes must exchange in order to find shortest paths in a fixed graph. The problem is discussed abstractly in that protocol issues are ignored. It is assumed that there is a protocol enabling nodes to communicate reliably with one another, and we count only the information relevant to the algorithm. For example, if one node sends to a neighbor the length of some link, we count only the number of bits needed to encode this length, even though this "message" will have protocol bits appended. This abstraction of communication cost is analogous to that made in the study of centralized algorithms. Elementary operations such as addition or comparison are often assigned unit cost since they only require some bounded number of machine language instructions to implement. Similarly, we view protocol as a fixed overhead and are interested in how the abstract communication cost grows as a function of the size of the network.

I.2 Notation and Definitions

For our purposes, a network is represented by a directed graph or digraph $G=(N,A)$, where N is a set of vertices or nodes labelled $1, \dots, n$ and $A \subseteq N \times N$ is a set of arcs. An arc from node i to node j is denoted by (i,j) . We assume all arcs are nontrivial, i.e. $(i,j) \in A \Rightarrow i \neq j$. The set $\{j \mid (i,j) \in A\}$ is termed the adjacency list of i , $AL(i)$, and a node $j \in AL(i)$ is called a neighbor of i . The cardinality of $AL(i)$, $|AL(i)|$, is called the outdegree of i , $OD(i)$. Similarly, the indegree of node i , $ID(i)$, is defined as $|\{(k,i) \mid (k,i) \in A\}|$. Note $\sum_i OD(i) = \sum_i ID(i) = |A|$. G can be naturally associated with an undirected graph, $\bar{G}=(N,L)$, where N is the same set of nodes as in G , and L is a set of undirected edges or links, $\langle i,j \rangle$. We take $\langle i,j \rangle \in L$ if either $(i,j) \in A$ or $(j,i) \in A$ inclusively. If $(i,j) \in A$ and $(j,i) \in A$, we still take only one copy of $\langle i,j \rangle$. The degree of node i in \bar{G} , $D(i)$, is defined as $|\{\langle i,j \rangle \mid \langle i,j \rangle \in L\}|$. Note that $\langle i,j \rangle$ and $\langle j,i \rangle$ are the same element. $\sum_i D(i) = 2|L|$ since a link is counted at both endpoints.

A path in G from i to j , $P[i,j]$ is a finite sequence of vertices $[i=i_1, i_2, \dots, i_k=j]$, such that $(i_\ell, i_{\ell+1}) \in A$, $1 \leq \ell < k-1$. The arcs $(i_\ell, i_{\ell+1})$ are said to be in the path. If $i=j$ and all other nodes are distinct and different from i then $P[i,j]$ is a cycle. A path $P[r,t] = [r=j_1, \dots, j_s=t]$ is a subpath of $P[i,j]$ if

- 1) $P[r,t]$ is a subsequence of $P[i,j]$ and
- 2) (x,y) is an arc in $P[r,t]$ only if (x,y) is an arc in $P[i,j]$.

Path can be analogously defined for undirected graphs in an obvious manner. A directed graph G is called strongly connected if there is a path from i to j , $\forall i \neq j$. G is called weakly connected or just connected if there is a path in \bar{G} from i to j , $\forall i \neq j$. Note that in \bar{G} , if there is a path from i to j then there is also one from j to i since the edges have no direction. This is not true in G , and hence we distinguish strongly connected from connected. The diameter of \bar{G} , $D(\bar{G})$ is defined as $\max(\min \# \text{ of links in a path from } i \text{ to } j \text{ in } \bar{G})$. This means that for any i, j , there is a path from i to j in \bar{G} using at most $D(\bar{G})$ edges.

A weighted digraph $G=(N,A,\ell)$ is a digraph together with a function $\ell: N \times N \rightarrow (-\infty, \infty]$, and $\ell(i,j)$ is called the length of arc (i,j) . We take $\ell(i,i)=0 \forall i \in N$, $\ell(i,j) < \infty, \forall (i,j) \in A$, and $\ell(i,j) = \infty \forall (i,j) \notin A$. That is, for shortest path problems, we consider those arcs in the graph to have finite length and missing arcs to have infinite length. The length of a path $P[i,j]=[i_1, \dots, i_k]$ is defined as $\sum_{s=1}^{k-1} \ell(i_s, i_{s+1})$.

We will frequently consider asymptotic rates of growths of functions. To express this notion in compact form, we introduce the "big-oh" notation. A function $f(n)$ is said to be $O(g(n))$ (order $g(n)$ or big-oh of $g(n)$) if there exists an n_0 and a constant c such that

$$|f(n)| \leq |c.g(n)|, \forall n \geq n_0.$$

Additionally, we will often denote the cardinality of a set S , $|S|$, simply by S if it is clear that the reference is to a numerical quantity and not to S as a set.

I.3 Problem Formulation

(Much of the material in the remainder of this chapter and in the next chapter is drawn from [3] and [4]. They are excellent references for the reader who is unfamiliar with graphs or the shortest path problem).

Given a weighted digraph there are 3 shortest path problems we consider.

- 1) All Pairs (AP) - Find the length of a shortest path from i to j , $\forall i \neq j$.
- 2) Single Source (SS) - Find the length of a shortest path from a source node i to all other nodes.
- 3) Single Pair (SP) - Find the length of a shortest path from a given source to a given destination.

In practice, one may actually want the path or just the length. We tend to such details when necessary.

Assume for the moment that G is strongly connected. If G has cycles of negative length, then there is no shortest path from i to j for any choice of i and j . Otherwise, all of the above problems have well defined solutions, and there is a cycle free shortest path between any two nodes. (Cycles of zero length can be deleted). One of the basic observations concerning the shortest path problem is the following "Principle of Optimality":

If $P[i,j]$ is a shortest path from i to j then any subpath $P[r,t]$ must also be a shortest path from r to t .

From this principle, we readily obtain the following necessary conditions for the shortest path path lengths in a weighted digraph having no negative length cycles.

Let $D(i,j)$ = length of a shortest path from i to j .

Then

$$D(i,i) = 0$$

$$D(i,j) = \min_{k \neq j} (D(i,k) + \ell(k,j))$$

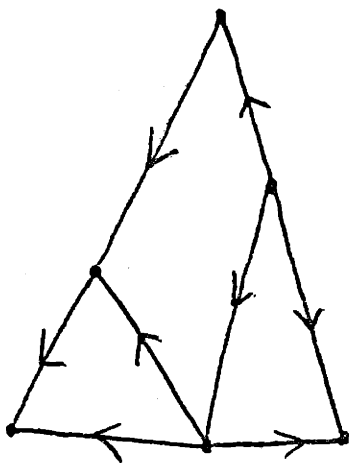
These are called Bellman's Equations (BE). It is shown in [4], that if G has no nonpositive cycles, then BE are uniquely satisfied by the shortest path lengths. If G has zero length cycles, then BE may have solutions other than the shortest path lengths. In practice however, computational procedures find the shortest path lengths solutions even if G has zero length cycles.

I.4 Shortest Path Trees

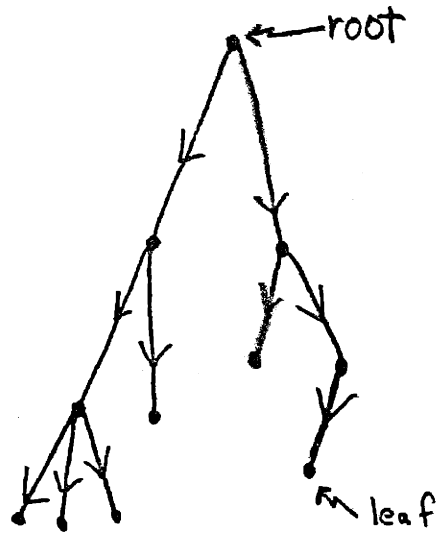
A digraph containing no cycles is called a directed acyclic graph or dag for short. A rooted tree, $T=(M,B)$, is a dag satisfying

- 1) There is exactly one vertex called the root with indegree = 0.
- 2) Every other vertex has indegree = 1.
- 3) There is a path from the root to every other vertex.

Note that 2) implies that the path in 3) is unique, and one can also show that $B=M-1$. Because T is acyclic, there is at least one vertex having outdegree = 0, and any such vertex is a leaf. The depth of a vertex j in T is defined as the number of arcs in the path from the root to j . The depth of the root is 0, and the depth of the tree is defined as $\max(\text{depth}(j) | j \in M)$. Examples of trees and dags are shown below.

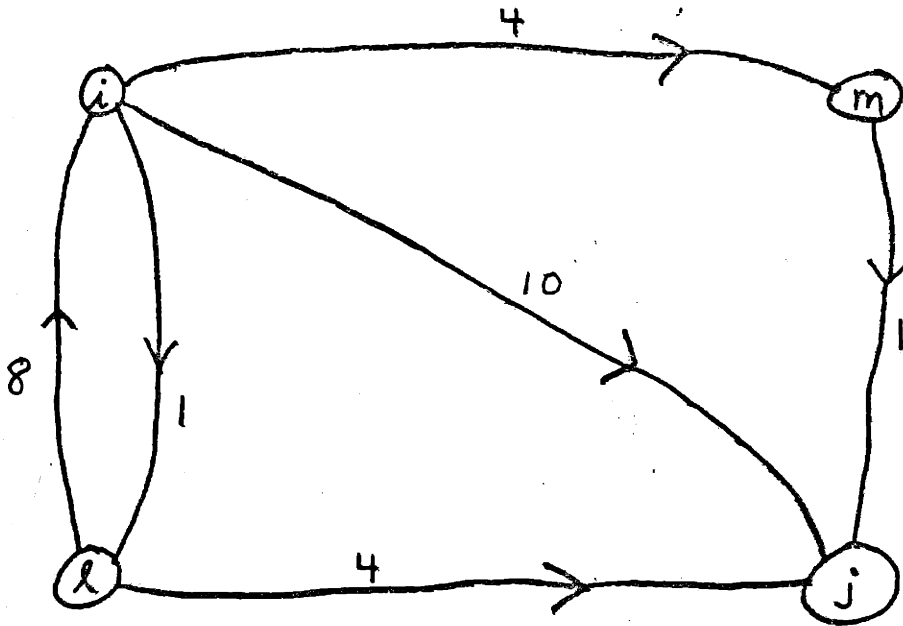


DAG

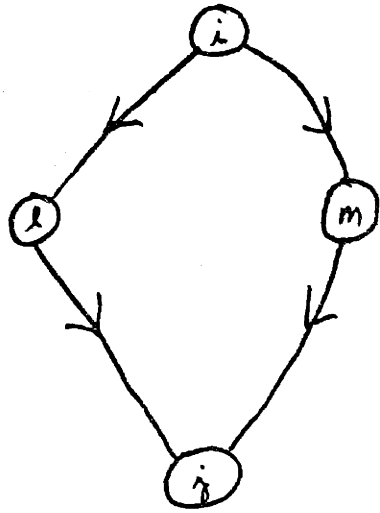


TREE

Now suppose $G=(N,A)$ is a weighted digraph in which every cycle has positive length. Define $DO(i)$ to be the set of arcs such that $(p,q) \in DO(i)$ iff (p,q) is an arc in a shortest path in G from i to some other vertex. Let $DSPO(i)$ be the graph $(N,DO(i))$. By using the principle of optimality and the fact that all cycles in G have positive length, one can show that the graph $DSPO(i)$ is a dag. ($DSPO(i)$ stands for dag of shortest paths out of i). $DSPO(i)$ is not always a tree because there can be two or more different paths from i to some other vertex that have equal lengths.

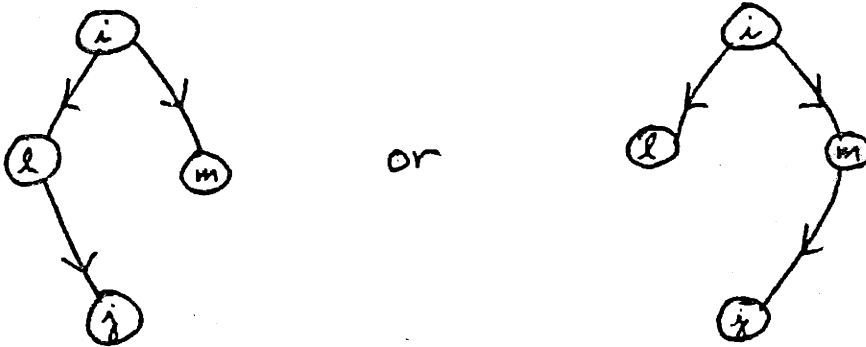


G



DSPO(i)

By arbitrarily choosing only one of the paths (in the example, delete either (l,j) or (m,j)) we can change the graph into a tree.



If we do this for all vertices in $DSPO(i)$ having indegree >1 , we can construct a tree rooted at i in which the path from i to j in the tree is a shortest path from i to j in G . We call this a tree of shortest path out of i , $TSPO(i)$. By considering the set of arcs in G that are in shortest paths from all other vertices to a vertex i , one obtains analogous structures - shortest path trees and dags into i . We call these $DSPI(i)$ and $TSPI(i)$ ($TSPI(i)$ is a graph satisfying the definition of rooted tree with the roles of indegree and outdegree reversed. Thus it is a reverse rooted tree, but we still call it a tree). If

$l(i,j)=1 \quad \forall (i,j) \in A$ then a shortest path is called a minimum hop path for obvious reasons. In this special case, we call these trees and dags MHT(D)I(O)(i) for minimum hop tree (dag) into (out of) i. These tree and dag structures play an important role in shortest path algorithms.

II. CENTRALIZED ALGORITHMS FOR THE SHORTEST PATH PROBLEM

II.1 Single Source Problem

It is perhaps a curious fact that there is no known algorithm for solving the SP problem that in effect doesn't solve the SS problem at the same time. In this section, three algorithms for solving the SS problem are presented.

Dijkstra's Algorithm

This algorithm works under the assumption that $\ell(i,j) \geq 0 \forall (i,j) \in A$.

We take node 1 as the source.

$$D(1,1) \leftarrow 0$$

$$D(1,j) \leftarrow \ell(1,j)$$

$$P \leftarrow \{1\},$$

$$T \leftarrow \{2, \dots, n\}$$

Step 1: Designation of the set P of Permanent Labels

$$\text{Find } \hat{k} \in T \text{ s.t. } D(1, \hat{k}) = \min\{D(1,j) \mid j \in T\}$$

$$T \leftarrow T - \{\hat{k}\}$$

$$P \leftarrow P \cup \{\hat{k}\}$$

If $T = \emptyset$ stop. Else Go To Step 2.

Step 2: Revision of Tentative Distances to Nodes in T

$$\forall j \in T \cap A(\hat{k}) \text{ do: } D(1,j) \leftarrow \min\{D(1,j), D(1,\hat{k}) + \ell(\hat{k},j)\}$$

Go To Step 1.

The m^{th} time step 1 is executed, at most $n-m-1$ comparisons are needed to find the minimum. Thus step 1 requires a total of $O(n^2)$ comparisons.

Each arc in A appears at most once as part of an addition. Thus the total cost is $O(n^2 + A)$, n^2 comparisons and $O(A)$ additions.

Proof of Correctness

Claim: Let $T_m = \{j_1, \dots, j_{n-m}\}$ and P_m be the sets T and P at the beginning of the m^{th} iteration. Let $D_m(1, j_k)$ be the value of $D(1, j_k)$ at that time, $1 \leq k \leq n-m$. Assume (without loss of generality) that node j_1 is the node in T_m chosen in the first line of step 1. Then

- 1) $D_m(1, j_k)$ is the length of a shortest path from 1 to j_k s.t. all nodes except j_k are in P_m ; for all $1 \leq k \leq n-m$.
- 2) $D_m(1, j_1)$ is in fact a shortest path length (unconstrained).

Proof:

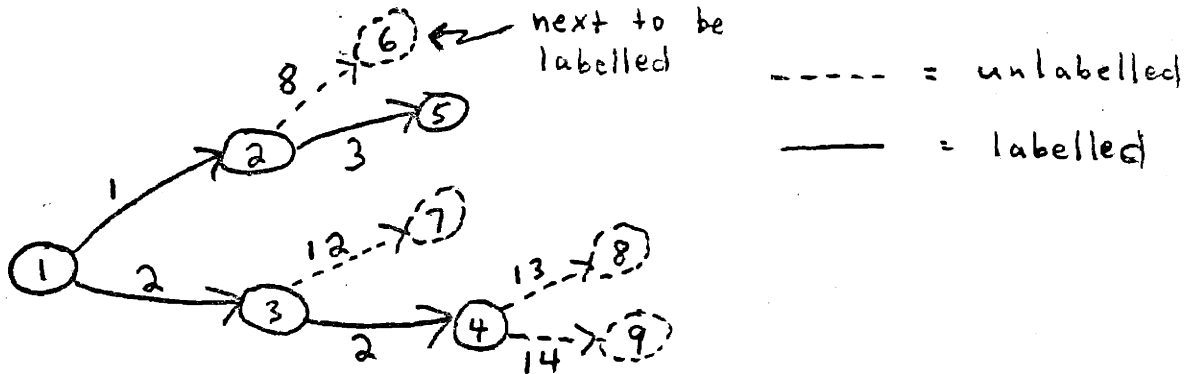
Part 1) follows immediately from the fact that at each iteration, $D(1, j)$, is updated with the smaller of the old value of $D(1, j)$ and the path length $D(1, \hat{k}) + \ell(\hat{k}, j)$ where \hat{k} was just marked permanent.

To prove part 2), assume that there is some other path $\tilde{P}[1, j_1]$ of length $\tilde{D}(1, j_1)$ s.t. $\tilde{D}(1, j_1) < D_m(1, j_1)$. Let node x be the first node in $\tilde{P}[1, j_1]$ that is in T_m (note $j_1 \in T_m \Rightarrow x$ exists, and node 1 $\in P_m \Rightarrow x \neq 1$). Let $\tilde{P}[1, x]$ be the subpath of $\tilde{P}[1, j_1]$ that goes from 1 to x and let the length of $\tilde{P}[1, x]$ be $\tilde{D}(1, x)$. Then we have

$D_m(1,x) \leq \tilde{D}(1,x)$ by part 1 of the claim since all nodes in $\tilde{P}[1,x]$ except x are in P_m
 $\leq \tilde{D}(1,j_1)$ since all arclengths are ≥ 0 .
 $< D(1,j_1)$ by assumption.

Hence we have $D_m(1,x) < D_m(1,j_1)$. If $x=j_1$, this is a contradiction. If $x \neq j_1$, this means j_1 would not have been chosen in step 1. Again contradiction. ||

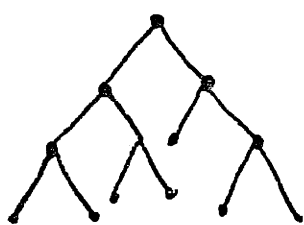
Thus Dijkstra's algorithm works by growing a shortest path tree. It finds such a tree for nodes i_1, \dots, i_k , say, such that $D(1,i_1) \leq D(1,i_2) \leq \dots \leq D(1,i_k)$ and shortest paths to remaining nodes have lengths $\geq D(1,i_k)$. It then considers one arc extensions of this tree and labels the closest unlabelled nodes. (See example below):



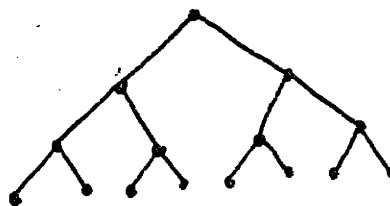
Observe, however, that it considers all one arc extensions from a certain node as soon as it is labelled. The set of tentative distances, therefore, need not be in any order, and so it can take $n-m-1$ comparisons to find the

minimum in the worst case at the m^{th} stage. By considering shortest one arc extensions only one at a time, we can achieve an $O(A \log n)$ comparisons algorithms. This is better than $O(n^2)$ if $A \ll n^2$, i.e. sparse graphs. This modification of Dijkstra's algorithm is due to Spira [5].

To explain this procedure, we introduce the notion of played binary trees. An undirected graph \bar{T} is a binary tree if it is the undirected graph associated with a rooted tree T having the property that all vertices in T , other than the leaves, have outdegree = 2. If all leaves are at the same depth the binary tree is complete.



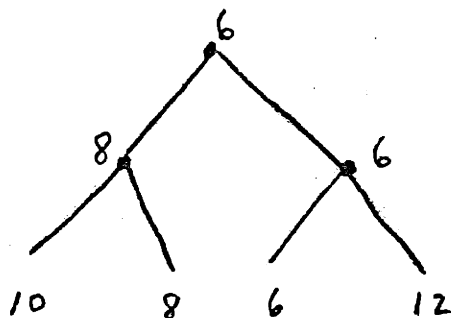
not complete



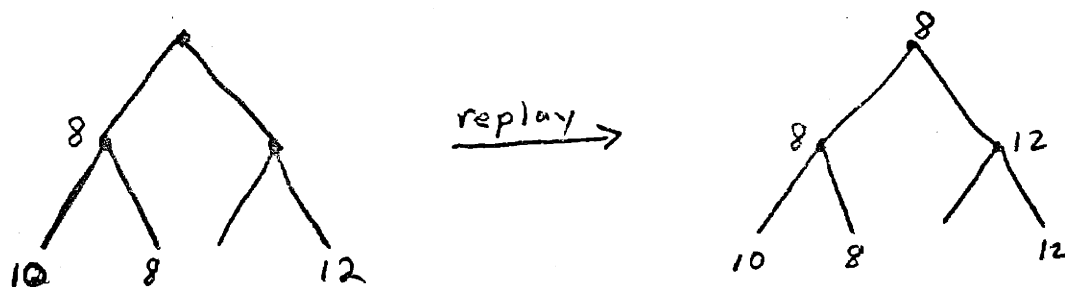
Complete

Notice that a binary tree with n leaves must have at least depth $\lceil \log_2 n \rceil$.

Binary trees are useful for extracting minima of sets. To find the minimum of a set of n numbers, first construct a complete binary tree with $2^{\lceil \log_2 n \rceil}$ leaves. Place the numbers on various leaves and perform successive comparisons going up the tree.



This is called playing the tree and $n-1$ comparisons are required to find the minimum. After doing this, we can find the next smallest element in $O(\log n)$ comparisons. We erase the path taken by the winner and then replay that part of the tree. Since it has depth $O(\log n)$, only $O(\log n)$ comparisons are required.



We obtain a savings over the naive method (performing $n-2$ comparisons on the $n-1$ remaining elements) because we only have to redo comparisons among elements that lost to the minimum the first time. This leads us to the following lemma.

Lemma: There is an algorithm to find the k smallest elements of a set in $n-1 + (k-1) \lceil \log n \rceil$ comparisons.

Spira's algorithm uses a variation of this idea.

Lemma: Let $\{S_i\}, 1 \leq i \leq k$ be a family of sets such that

- 1) $|S_1| = 1$
- 2) $S_i = S_{i-1} - \{\min S_{i-1}\} + \text{one or two new elements.}$

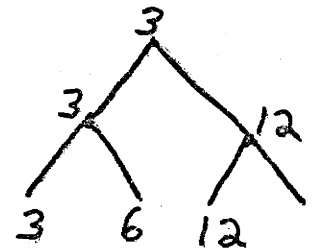
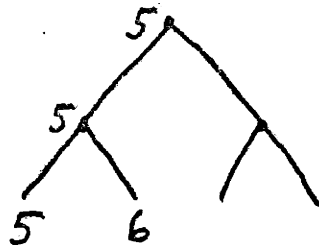
Then there is an algorithm to successively find the minimum of S_1, S_2, \dots, S_k in $2k \lceil \log n \rceil$ comparisons provided $|S_k| \leq n$.

We sketch the proof since it should be pretty obvious. Construct a complete binary tree with $2^{\lceil \log n \rceil}$ leaves. Place the element in S_1 at a leaf and play the tree ($\lceil \log n \rceil$ steps). Erase the path taken by this element and place one new element in S_2 at the leaf where the element of S_1 was. If there is another new element in S_2 , place it at some other leaf. Replay the tree. The procedure continues in an obvious manner. ||

$$S_1 = \{1\}$$

$$S_2 = \{5, 6\}$$

$$S_3 = \{6, 3, 12\}$$



Now we present the algorithm. We assume that we are given an arclength list for each node i , and we call this list ALLST(i). Node 1 is taken to be the source.

Dijkstra-Spira Algorithm

1. For $i=1$ to n DO: SORT ALLST(i) into increasing order.
2. For $j=2$ to n DO: LABEL(j) = TENTATIVE
3. $T \leftarrow \{2, \dots, n\}$
4. $D(1,1) \leftarrow 0$
5. $S_1 \leftarrow \{D(1,1) + \min \text{ALLST}(1)\}$.

COMMENT:

The sets S_i will consist of distances of various paths from node 1 to other nodes. Each such distance will be stored in the form $D(1,p) + \ell(p,k)$ since we will need the identity of the node p which is the next to last vertex in the path from 1 to k having distance $D(1,p) + \ell(p,k)$. Though we don't explicitly give the data structure for doing this, we observe that storing a distance in this two-part manner only adds a constant amount of work over just storing the distance as one number.

6. $i \leftarrow 1$
7. Let $D(1,\hat{p}) + \ell(\hat{p},\hat{k}) = \min\{D(1,p) + \ell(p,k) \mid (D(1,p) + \ell(p,k)) \in S_i\}$
8. $\text{ALLST}(\hat{p}) \leftarrow \text{ALLST}(\hat{p}) - \{\min \text{ALLST}(\hat{p})\}$.

COMMENT ON 8: The minimum of $\text{ALLST}(\hat{p})$ at this point is just $\ell(\hat{p},\hat{k})$ and since we examined it in 7, we remove it.

$$9. \quad S_{i+1} \leftarrow S_i - \{D(l, \hat{p}) + \ell(\hat{p}, \hat{k})\} \cup \{D(l, \hat{p}) + \min \text{ALLST}(\hat{p})\}$$

COMMENT ON 9: We revise our set of distances by deleting the one just examined in 7 and adding the distance of the next best path that is a one arc extension from \hat{p} .

10. IF LABEL(\hat{k}) = TENTATIVE THEN

DO ;

LABEL(\hat{k}) = PERMANENT

T \leftarrow T - { \hat{k} }

D(l, \hat{k}) \leftarrow D(l, \hat{p}) + $\ell(\hat{p}, \hat{k})$

$S_{i+1} \leftarrow S_{i+1} \cup \{D(l, \hat{k}) + \min \text{ALLST}(\hat{k})\}$

END;

COMMENT ON 10: If the distance in 7 was to a tentative node, then it is a shortest distance. We mark the node as permanent and add to the set of candidate distances the distance of the next best path that is a one arc extension from \hat{k} .

11. $i \leftarrow i+1$

12. If T $\neq \emptyset$ GO TO 7

13. ELSE STOP.

The proof of correctness of this procedure should be clear from previous discussions. It is basically Dijkstra's algorithm except extensions of paths from labelled nodes are considered one at a time. This allows the minimum to be found in an efficient manner in step 7.

Sorting a list of q numbers can be done in $c \cdot q \log q$ comparisons for some constant c . Thus, the arclength list $ALLST(i)$ can be sorted in $c \cdot OD(i) \log(OD(i))$ comparisons \Rightarrow step 1 requires $c \sum_{i=1}^n OD(i) \log(OD(i))$

comparisons. Now $OD(i) \leq n$ and so step 1 requires at most

$$c \cdot \log n \sum_{i=1}^n OD(i) = c \cdot A \cdot \log n \text{ comparisons. Step 7 requires } O(\log n)$$

comparisons to find the minimum because of the lemma, and it is executed at most once for each arc in A . Thus the overall cost is $O(A \log n)$.

Bellman-Ford Method

If there are negative arclengths then the inductive character of Dijkstra's algorithm breaks down since a path can have a smaller length than its subpaths. However, as long as there are no negative length cycles, there will exist cycle free shortest paths. The following procedure solves the SS problem when negative arclengths are allowed provided there are no negative length cycles. It is $O(n^3)$, however, since in effect it considers extensions of paths from all nodes rather than extensions of known optimal paths. Node 1 is again taken as the source.

$$D(1,1,1) = 0$$

$$D(1,j,1) = \ell(1,j) \text{ (Recall } \ell(1,j) = \infty \text{ if } j \notin AL(1))$$

$$m \leftarrow 1$$

Step 1: $D(1,j,m+1) \leftarrow \min[D(1,j,m), \min_{k \neq j} \{D(1,k,m) + \ell(k,j)\}]$

Step 2: IF $D(1,j,m+1) = D(1,j,m) \forall j$ then STOP.
ELSE DO: $m \leftarrow m+1$ Go to Step 1.

An intuitive proof of correctness can be seen in the following interpretation of $D(1,j,m)$:

$D(1,j,m)$ = the length of a shortest path from 1 to j that uses m or fewer arcs.

(the reader can easily verify this interpretation). Since there is a shortest path using $n-1$ or fewer arcs, $D(1,j,n-1)$ must be a shortest distance, $\forall j$. If the test in step 2 yields a true result, for $m < n-1$, then we are fortunate to have early convergence. To analyze the cost, observe that each execution of step 1 requires $O(n^2)$ comparisons and additions. Since step 1 can be executed $O(n)$ times in the worst case, the algorithm requires $O(n^3)$ comparisons and additions.

We observe that the algorithm can be modified so that in step 1, only the additions $D(1,k,m) + \ell(k,j)$ for $(k,j) \in A$ are performed. In this case, each execution of step 1 requires a total of $O(A)$ additions and comparisons and overall the algorithm is $O(An)$. This of course can be $O(n^3)$.

II.2 All Pairs Problem

Let P and Q be two real $n \times n$ matrices. Define the min/plus product $*$ of P and Q , $R = P * Q$ by

$$R_{ij} = \min_k \{p_{ik} + q_{kj}\}$$

If L is a matrix of arclengths $\ell(i,j)$ then a little work shows that

$$L * (L * (\dots * L) \dots)$$

m times

is a matrix of shortest path lengths subject to the constraint that the paths have m or fewer arcs. One can show that $*$ is associative, and therefore, L^{n-1} , a matrix of shortest path lengths, can be computed by performing $\log_2(n-1)$ successive squaring operations. Each squaring can be done in $O(n^3)$ additions and comparisons and so we obtain an $O(n^3 \log n)$ algorithm.

There is in fact a clever labelling algorithm due to Floyd and Warshall that solves the AP problem in $O(n^3)$ comparisons and additions.

Let $D(i,j,m)$ = the length of a shortest path from i to j subject to the constraint that all nodes other than i and j are not in the set $\{m, m+1, \dots, n\}$. Then $D(i,j,n+1)$ is a shortest distance between i and j . (Again we assume no negative length cycles). Now a shortest path from i to j that has no intermediate nodes in the set $\{m+1, \dots, n\}$ either a) does not visit m , in which case $D(i,j,m+1) = D(i,j,m)$ or b) visits m in which case $D(i,j,m+1) = D(i,m,m) + D(m,j,m)$. Hence we can compute

$D(i,j,n+1)$ as follows.

1. $D(i,j,1) = \ell(i,j)$
2. $m \leftarrow 1$
3. $D(i,j,m+1) \leftarrow \min(D(i,j,m), D(i,m,m) + D(m,j,m))$
4. $m \leftarrow m+1$
5. IF $m=n+1$ STOP. ELSE GO TO 3.

Note, negative arclengths are allowed as long as there are no negative length cycles. There are exactly $n(n-1)(n-2)$ equations to solve, $i \neq j$ and $m \neq i$ and $m \neq j$ and hence the algorithm is $O(n^3)$. This is the same order of complexity as the Bellman-Ford algorithm yet here we find all $n(n-1)$ shortest paths. The AP problem has an algebraic flavor (matrices) that is not present in the SS problem. Observe, though, that if $\ell(i,j) \geq 0 \forall i \neq j$ then we can also achieve an $O(n^3)$ algorithm for the AP problem by applying Dijkstra's algorithm n times, once for each node as a source.

II.3 Lower Bounds

An analytic tree program is a program that at each step evaluates some analytic function of the input and then branches to one of two successive steps. Dijkstra's algorithm is such a procedure since at each step it computes a linear function of the input distances (addition) and then branches based on a comparison. Spira and Pan [6] have shown that any analytic tree program which verifies that a weighted rooted tree with n vertices is a shortest path tree (with respect to some weighted graph containing this tree) must use $O(n^2)$ comparisons in the worst case. Thus Dijkstra's algorithm is essentially optimal in the class of algorithms that use comparisons among sums of arclengths to compute a shortest path tree. An interesting open problem is to find an $O(A)$ algorithm for the SS problem when the graph is sparse, i.e. $A \ll n^2$.

It has also been shown that computing the matrix product $*$ is of the same order of complexity as solving the AP problem, when the matrix entries and arclengths are nonnegative. That is, computing $P*Q$ is of the same difficulty as computing L^{n-1} for nonnegative reals. The reader is referred to [3] for a proof, references to the original work, and a more general formulation of the minimum cost path problem and its relationship to computing $*$. The problem of detecting negative cycles is discussed in [4]. Kerr [7] has shown that if the only permissible operations are $p+q$ and $\min(p,q)$ then $O(n^3)$ operations are required to compute $P*Q$ in the worst case.

However, Fredman [8] has found an $O\left(\frac{n^3 \log \log n}{\log n}\right)$ algorithm that of course uses other operations. In addition to [3] and [4], the reader is also referred to a doctoral thesis by D.B. Johnson [9] for a discussion of the shortest path problem.

III. DISTRIBUTED SHORTEST PATH ALGORITHMS

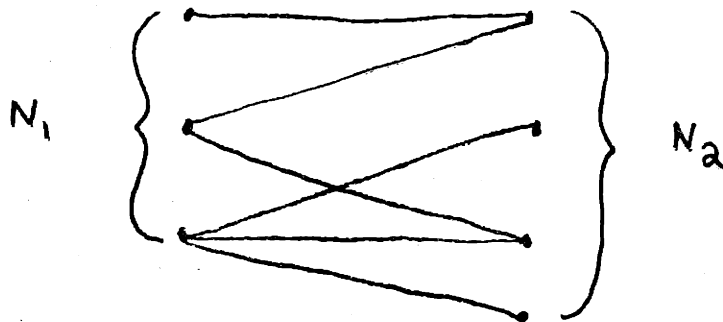
III.1 More Notation and Assumptions

Since G represents a data network, we hereafter assume, (unless otherwise noted)

- 1) $(i,j) \in A$ implies $(j,i) \in A$. (links are duplex)
- 2) \bar{G} is connected
- 3) $\ell(i,j) > 0$ $(i,j) \in A$. (Conventional routing algorithms generally assign positive costs to the links).

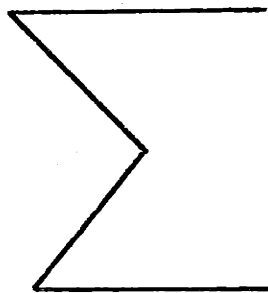
A directed graph satisfying condition 1 is termed symmetric. However, it is not assumed that $\ell(j,i) = \ell(i,j)$, because in general the cost of sending a certain flow over the path $[i,j]$ will not be the same as sending an equal amount of flow over the path $[j,i]$. Note that 1) and 2) imply that the directed graph G is strongly connected.

An undirected graph $\bar{G}=(N,L)$ is called bipartite if $\exists N_1$ and N_2 such that $N_1 \cap N_2 = \phi$, $N_1 \cup N_2 = N$ and $\langle i,j \rangle \in L$ implies either $i \in N_1, j \in N_2$, or $i \in N_2, j \in N_1$. Intuitively, a bipartite graph is one that can be made to look like this



for some appropriate partition of the set N into N_1 and N_2 . Our interest in bipartite graphs will become apparent in III.2.

We assume that at each node of the network, there is a computer capable of performing the usual operations such as addition, subtraction, comparison, branching, and data storage and retrieval. Additionally, we assume that there is a protocol enabling reliable node to node communication. The communication cost of an algorithm will be taken as



message cost in bits x # arcs traversed
by message

all messages
occurring while
algorithm executes

In general, messages will consist of node identities and arclengths. An element of the set of integers $\{1, \dots, M\}$ can be encoded into a binary number with $\lceil \log_2(M) \rceil$ bits. Note that even if we wish only to encode the number 1 as an element of this set, $\lceil \log_2 M \rceil$ bits are still required since we must specify the end of this message. Thus a node identity can be encoded with $\lceil \log_2 n \rceil$ bits. If arclengths are all integers, then one arclength can be encoded with $\lceil \log_2 (\ell - \max) \rceil$ bits

where $\ell\text{-max} = \max\{\ell(i,j) \mid (i,j) \in A\}$. If arclengths are allowed to be arbitrarily large integers or rational numbers with arbitrary precision, then the cost of representing them is clearly unbounded. However, this is misleading since in some sense, we wish to consider the transmission of an arclength from some node to one of its neighbors as 1 message. In conventional network, the arclengths tend to be relatively small integers anyway. The communication cost of a distributed algorithm X will be denoted by $CC(X)$.

The algorithms presented in the remainder of the thesis will require that the nodes process and communicate various arclengths and path distances. For notational convenience, we will denote arclengths and path distances simply by $\ell(i,j)$ and $d(i,j)$ when presenting the algorithms. However, we will assume that whenever a node stores an arclength $\ell(i,j)$, it does so in a manner that allows it to retrieve the identities i and j at some later time, (This can be done by either directly storing the identities i and j together with the length or using a data structure that allows i and j to be uniquely determined from the position of the arclength in the data structure, e.g. mapping the pair (i,j) into some index in a 1-1 manner.) In particular, a node will be able to supply the identities i and j as well as the length if some other node requests this information. The same will hold for distances $d(i,j)$.

III.2 Minimum Hop Path Algorithms

We first consider the problem of finding minimum hop paths since it illustrates the salient features of the operation and analysis of distributed algorithms. Each node i initially knows only its own identity, and at the completion of the algorithm knows all neighbors j_k through which it has minimum hop paths to k , for each $k \neq i$. It doesn't know the entire path, but then this is not necessary. With all distributed algorithms, there is the problem of beginning the procedure. That is, how do the nodes know when to start. For the time being, we ignore this problem, and assume that, somehow, the nodes receive a signal to start. Later, we will deal with the problem of executing these algorithms in an asynchronous environment.

Algorithm MH1: (Gallager)

Each node i executes

Step 0: Broadcast the identity " i " to all neighbors and receive transmissions from them.

Step ℓ : 1) Record newly discovered identities and the neighbors $\ell > 0$ from which they were received.

2) If no new identities were received, broadcast the message "done" and stop.

3) Otherwise, to each neighbor j , broadcast the identities of all nodes not previously received from or broadcast to j . If there are no such identities to broadcast to j , send the message "nothing new".

4) Receive transmissions from all active neighbors.

One can see that i has an $(\ell+1)$ hop minimum hop path to a node k through neighbor j iff

1) i first heard about k at step ℓ and

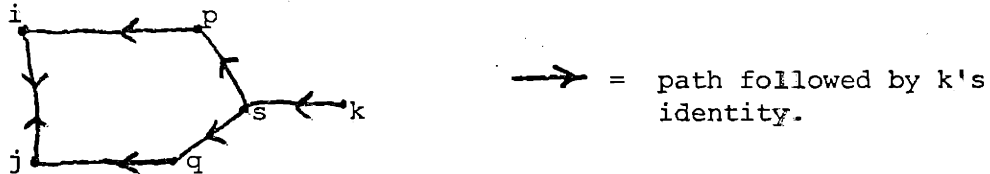
2) i heard about k from j at step ℓ .

Communication Cost:

Each node identity traverses each of the L duplex links in at least one direction. Since each identity is encoded into $\log n$ bits, we have

$$CC(MH1) \geq Ln \log n \text{ bits}$$

An identity k , will traverse a link $\langle i, j \rangle$ in both directions iff i and j are connected to k via equidistant minimum hop paths.



In the above graph $i(j)$ will hear about k from $p(q)$ at the same step and then tell the other about k at the next step. If there exist such i, k, j , then if in fact there is an s , such that i, j , and s are part of an odd elementary cycle in \bar{G} . (A cycle $[i_1 \dots i_k = i_1]$ in \bar{G} is termed elementary, if there is no other cycle in \bar{G} containing exactly a proper subset of the nodes $[i_1 \dots i_k]$.) One can show that \bar{G} has no odd length elementary cycles iff \bar{G} has no odd length cycles iff \bar{G} is bipartite. This corresponds to the fact that in a bipartite graph, for every link $\langle i, j \rangle$ and node k , either i has a minimum hop path to k through j or j has a minimum hop path to k through i . Note that in the example, neither i nor j has a minimum hop path to k through the other. Of course the graph is not bipartite.

To upperbound $CC(MH1)$, observe that node i transmits its identity to $D(i)$ neighbors and the identity of node $j \neq i$ to at most $D(i)-1$ neighbors, where $D(i)$ = degree of node i . Thus we have

$$\begin{aligned} CC(MH1) &\leq \left[\sum_i D(i) + \sum_i \sum_{j \neq i} (D(i)-1) \right] \log n \text{ bits} \\ &= [2Ln - n(n-1)] \log n \text{ bits.} \end{aligned}$$

There is a simple way to get around the odd cycle problem (at the expense of doubling the number of steps taken by MH1) and achieve an algorithm that uses $Ln \log n$ bits for all networks with n nodes and L duplex links. Each pair i, j s.t. $\langle i, j \rangle \in L$ arbitrarily choose one of the nodes to be HI and the other LO. Then MH1 is executed with HI and LO nodes broadcasting on alternate steps.

Algorithm MH2:

Step 0: Node pairs exchange identities and choose HI and LO.

Step ℓ : All HI nodes broadcast to their corresponding LO neighbors, ℓ odd as in MH1, new identities learned at Step $\ell-1$.

Step ℓ : All LO nodes broadcast to corresponding HI neighbors new ℓ even identities learned at Step $\ell-2$.

Termination is as in MH1

Note that a node can be HI relative to one neighbor and LO relative to another neighbor and it will broadcast to them on alternate steps. In fact there must be at least one such node unless \bar{G} is bipartite.

Algorithm MH1 takes $D(\bar{G})$ steps whereas algorithm MH2 takes $2D(\bar{G})$ steps. In both of them each node must perform a constant number of operations for each identity it receives - determining if the identity is old or new and storing it if it's new. Node i therefore performs $O(D(i) \cdot n)$ computations.

At the end of either MH1 or MH2, each node i knows all neighbors j_k through which it has a minimum hop path to k , $\forall k \neq i$. After MH1, even more is known. Each node i also knows its position in MHDI(k) relative to its neighbors. The rule for determining this is as follows:

We say that i is upstream (downstream) from a neighbor j in a dag if the arc (i,j) ((j,i)) is in the dag. Then there are three cases:

- 1) i has a minimum hop path to k through j . Then i is upstream from j in MHDI(k).
- 2) i and j told each other about k at the same step. They are not related, i.e. neither (i,j) nor (j,i) is in MHDI(k).
- 3) Otherwise i is downstream from j in MHDI(k).

The same works for MHDO(k) with the roles of upstream and downstream reversed. This rule doesn't quite work with MH2 because of the alternation. Consider the case in which i and j have equidistant minimum hop paths to k . If i is HI and j is LO, then i tells j about k , but j doesn't tell i about k . Algorithm MH2 uses less communication than MH1 because it resolves such ambiguities in one direction only. That is, j knows that i and j are not related in MHDI(k), but i only knows that j is not downstream. There is a way to resolve these ambiguities in L_n bits. Since j knows the precise relationships, it can communicate this to i . At the end of MH2, j sends i either

- 1) a stream of n bits with a 1 in position k if i and j are not related in $MHDI(k)$. Otherwise this bit is a 0. or
- 2) a list of such nodes k .

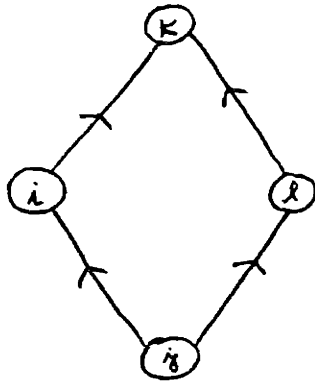
If there are α such nodes k , j chooses 1) iff $n < \alpha \log n$. At the end of $MH2$, i knows those k for which j is downstream from i in $MHDI(k)$.

For the other nodes, it only knows that either i and j are not related or that j is upstream in the appropriate $MHDI$. Using the information described above, i can resolve those ambiguities. Notice that this procedure must be done at most once for each $\langle i, j \rangle \in L$. The total cost of this is therefore Ln bits. So using $MH1$, it takes $[2Ln - n(n-1)] \log n$ bits to find the $MHDI(k)$. Whereas, using $MH2$ and the above information, it takes only $Ln[\log n + 1]$ bits.

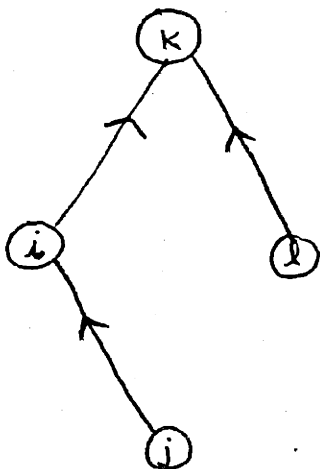
The previous discussion illustrates the significance of being able to wait and encode identities efficiently. $MH1$ resolves odd cycle ambiguities one at a time at a cost of $\log n$ bits/identity. With $MH2$, the LO nodes can wait and encode all ambiguities in approximately the minimum of $(n, \alpha \log n)$ bits. If $n < \alpha \log n$, then the augmented $MH2$ is better. Otherwise $MH1$ and $MH2$ have the same cost for finding the $MHDI(K)$, since these α identities would go from j to i in $MH1$ anyway. The number α depends on the number of odd cycles in \bar{G} . It appears difficult to make a precise statement about how α depends on L and n . As an example, for any n , even, and $n-1 \leq L \leq \frac{n^2}{4}$, \exists a connected bipartite graph with n

vertices and L edges, and bipartite graphs have no odd cycles.

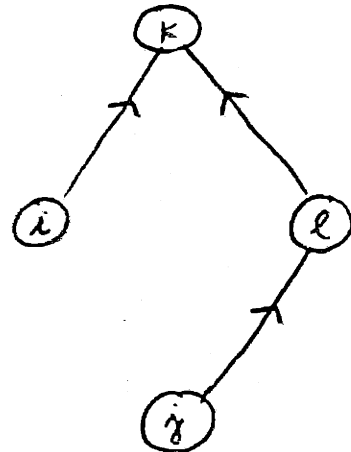
We now consider the problem of extracting minimum hop trees from minimum hop dags. A node i knows if a neighbor j is upstream in $MHDI(k)$. However i doesn't know if j is upstream from any other nodes.



In the above example, there are two possible trees.



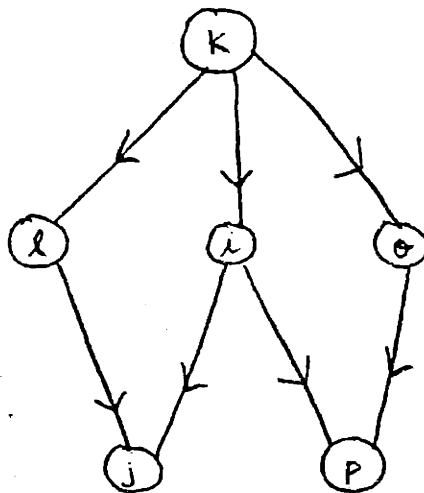
or



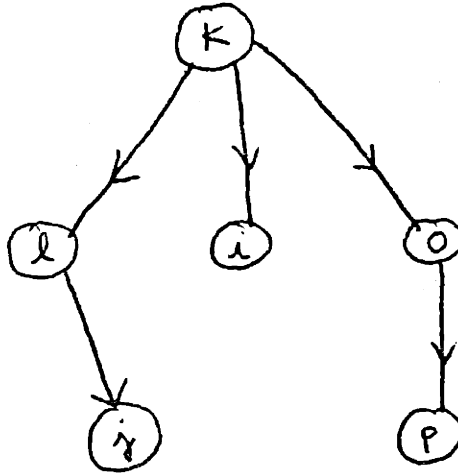
Node j is in a position to choose either i or l as its downstream neighbor in $MHTI(k)$. So once the nodes have determined their relative positions in $MHDI(k)$, $\forall k$, they can extract minimum hop trees as follows: Each node j must do:

For each $k \neq j$, choose a downstream neighbor j_k in $MHDI(k)$. There is at least one such node. Tell this neighbor j_k that you have chosen it as your downstream neighbor in $MHDI(k)$.

Note that even if j only has one downstream neighbor j_k in $MHDI(k)$, it must tell this neighbor, because j_k might not know it is a unique downstream neighbor. There are a total of $n(n-1)$ such messages, since each j must broadcast once about each $k \neq j$. Thus this costs $n(n-1) \log n$ bits, since the message to neighbor j_k consists essentially of the identity k . Once relative positions in $MHTI(k)$ have been chosen, the nodes immediately know relative positions in a $MHTO(k)$, k . As an example consider the following $MHDO(k)$.



If j chooses l and p chooses 0 , we get



At the beginning of this section, we mentioned the problem of starting these distributed procedures. The algorithms were presented with the assumption that each node somehow receives a starting signal. In fact, these algorithms can operate totally asynchronously, and any node can choose to begin. One method for coordinating the nodes is the following:

Suppose node i , and only node i , wishes to begin an update. It does so by sending its identity to its neighbors. Consider some neighbor of i , say j . When j receives node i 's identity, j "wakes up" and sends its own identity to all its neighbors, including i . Consider a neighbor of j , say k . k is either asleep (and so wakes up) or

has been awakened by another node, when it receives j 's probe. Thus k either sends or has already sent its identity to j . A node pair x, y s.t. $\langle x, y \rangle \in L$ is thus initiated into the update when they exchange identities. Once node $i(j)$ has learned the identities of all its neighbors, it sends this information to $j(i)$. After nodes i and j have done this with all neighbors, each knows two hop minimum hop paths, and then they exchange this information. In general, an arbitrary node P will at some point have all information about m hop minimum hop paths. Node p sends this information to all its neighbors and waits to hear from them. When it has received the m hop information from all its neighbors, it can compute its own $m+1$ hop minimum hop paths, and the process is repeated.

Observe that this method also works even if any number of nodes independently decide to begin an update. In general node p will be awakened by probes from one or more of its neighbors. Node p then sends its identity to all its neighbors and the process continues as above. The point is that once a node pair x, y s.t. $\langle x, y \rangle \in L$ exchange identities, they are initiated into the procedure and are coordinated from then on. The alternation feature of MH2 can also be implemented in this manner. When a node pair exchange identities, they can then decide which is HI and which is LO and proceed as usual.

It is desirable for a distributed algorithm to operate correctly in an asynchronous environment. If one wishes to design routing algorithms that work in the presence of link and node failures, it is important for the algorithms to allow a node to recognize some local failure and then independently initiate some procedure to inform the other nodes.

III.3 Broadcasting All Arclengths

It was mentioned in the introduction that classical shortest path algorithms can be used by one computer if it has all arclengths in memory. In this section, we present algorithms for broadcasting all arclengths to all nodes and for transmitting all arclengths to one node and then having this node send shortest path information back to the other nodes. It is assumed that each node i knows

- 1) its identity
- 2) its upstream and downstream neighbors in the minimum hop trees, $MHTO(k)$ and $MHTI(k)$, $\forall k$.
- 3) $\ell(i,j), \forall j \in AL(i)$.

The second assumption is not entirely impractical since the nodes need learn this information only once, and these trees can be used repeatedly for communication purposes as the arclengths change. One can view this as the distributed equivalent of preprocessing.

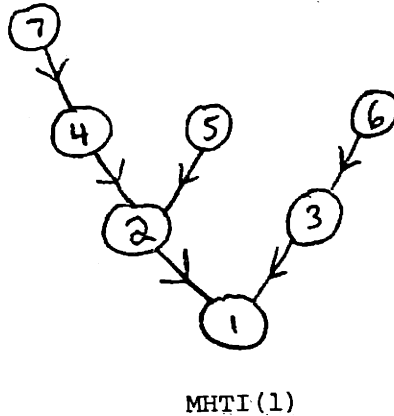
Now consider the following procedure for broadcasting all arclengths to a single destination j .

Algorithm BASD

Node j decides to do an update and begins by broadcasting a "start" message to its neighbors in $MHTO(j)$. These nodes in turn broadcast the "start" message to their neighbors in $MHTO(j)$. The message keeps propagating down the tree until it reaches the leaves. When a leaf receives the start, it broadcasts its arclength list to its downstream neighbor in $MHTI(j)$. This procedure continues until all arclengths reach the root j . That is, when a node i has received

information from all of its upstream neighbors in $MHTI(j)$, it broadcasts this information together with its own arclength list to its downstream neighbor in $MHTI(j)$.

Example:



Node 1 is the destination. $MHTI(1)$ is shown but other arcs in the network are not shown.

7 sends $ALLST(7)$ to 4.

4 sends $ALLST(7)$ and $ALLST(4)$ to 2.

5 sends $ALLST(5)$ to 2.

2 sends $ALLST(7)$, $ALLST(4)$, and $ALLST(5)$ to 1.

Other arclength lists propagate similarly.

Notice that the algorithm operates completely asynchronously. The "start" message can be interpreted as a ready command, and receipt of arclength lists from an upstream neighbor is an acknowledgement from that neighbor.

An arclength $\ell(i,k)$ will traverse each arc in the tree path from i to the root j exactly once. The maximum number of arcs in such a path is just the depth of $MHTI(j)$. An arclength can be

specified as a triple $(i, j, \ell(i, j))$ in $2\log n + \log(\ell - \max)$ bits.

Since $\max_j \text{Depth}(\text{MHTI}(j)) = D(\bar{G})$, the communication cost of this procedure is upperbounded over all possible destination nodes by

$$2LD(\bar{G}) [2\log n + \log(\ell - \max)] \text{bits.}$$

Once the destination node j has computed shortest paths for all node pairs, it can send this information back to the other nodes via minimum hop paths. Each node $i \neq j$ requires a neighbor m_{ik} through which it has a shortest path to k , $\forall k \neq i$. The identity of each such best route neighbor can be sent from j to i via a minimum hop path having at most $D(\bar{G})$ arcs. Accounting for all i , we see that node j must send the identities of $(n-1)^2$ such best route neighbors. Since each identity traverses at most $D(\bar{G})$ arcs and can be encoded with $\log n$ bits, the communication cost of this procedure is

$$O(D(\bar{G})(n-1)^2 \log n) \text{bits.}$$

Hence the overall cost of sending all arclengths to one node and then having this node send shortest path information back to the other nodes is

$$O[LD(\bar{G})(\log n + \log(\ell - \max)) + D(\bar{G})n^2 \log n] \text{bits.}$$

We now consider the problem broadcasting all arclengths to all destinations.

Algorithm BAAD

1. Each node i begins by sending its identity and arclength list to its neighbors in $MHTO(i)$, i.e. in its own tree.
2. When a node receives such an arclength list, it examines the source and then sends this list on to its downstream neighbors in the MHTO for that source.

Each node k receives $\ell(r,s)$ for $r \neq k$ exactly once. Thus

$$CC(\text{BAAD}) = 2L(n-1) (\text{arclength messages}).$$

where an arclength message uses $O(\log n + \log(\ell - \max))$ bits.

Again, observe that the algorithm is completely asynchronous, i.e., any node can start the update, and transmission delays can be arbitrary. If some node x is "sleeping" (i.e. not involved in the update) when it receives a transmission from one or more neighbors, it "wakes up", sends its own arclength list down its own tree, and then sends other subsequent arclength lists down the appropriate trees.

Observe that a variation of this algorithm can be used to send the topology of the whole network to all nodes, in $O(Ln \log n)$ bits. The nodes first perform a minimum hop algorithm to establish minimum hop trees. Then algorithm BAAD is executed with nodes simply sending adjacency list rather than arclength lists.

III.4 Worst Case Analysis of Some Distributed Shortest Path Algorithms

In this section, we assume that each node i initially knows

- 1) its own identity and the identities of its neighbors
- 2) the number of nodes in the network, n
- 3) $\ell(i,j), \forall j \in AL(i)$.

Once again, we temporarily ignore the problem of beginning the algorithm, and methods for coordinating asynchronous operation will be discussed later.

Algorithm SP1: (Distributed Bellman-Ford)

(The reader may wish to refer to Section II.1 for the discussion of the centralized Bellman Ford Algorithm).

Arclengths are not assumed to be positive. Each node i executes:

1. $m \leftarrow 0$
2. $\forall j \neq i$ do: $d(i,j,m) \leftarrow \ell(i,j)$.
3. If $d(i,j,m) < \infty$, broadcast $d(i,j,m)$ to all neighbors.
4. Receive transmission from neighbors.
5. Update distances as follows:
$$d(i,j,m+1) \leftarrow \min\{d(i,j,m), \min_{k \in AL(i)} \{d(k,j,m) + \ell(i,k)\}\}$$
6. $m \leftarrow m+1$
7. If $m \geq n-1$ then stop. Else go to step 3.

Although the details have been omitted, it is clear that each node can easily keep track of which neighbor is associated with a shortest distance, i.e. a neighbor k s.t. $d(i,j,\text{optimal}) = \ell(i,k) + d(k,j, \text{optimal})$.

Each node i broadcasts at most n times about each of the $n-1$ other nodes in the worst case. Thus node i broadcasts at most $D(i)n^2$ distances, where $D(i)$ is the degree of i . Summing over all i , we obtain

$$CC(SP1) \leq 2Ln^2 \text{ (distance messages).}$$

Observe that the following improvements can be made in practice:

- a) A node does not initially have to know the number of nodes in the network, but can learn this as the algorithm is executed. This is because if after the m^{th} iteration, a node i has heard about x other nodes, with $x < n-1$, then node i must hear at least one new identity at the $(m+1)^{\text{st}}$ iteration. Thus, if node i learns no new identities at the $(m+1)^{\text{st}}$ iteration, it knows the identity of (but not necessarily a shortest path to) every other node in the network.
- b) Node i need only broadcast $d(i,j,m+1)$ to a neighbor k if $d(i,j,m+1) < d(i,j,m)$ and $d(i,j,m+1) < \min_{\ell < m} \{d(k,j,\ell)\}$ received from k .
- c) If $d(i,j,m+1) = d(i,j,m), \forall j \neq i$, then node i can stop.

- d) If $\ell(i,j) \geq 0 \forall (i,j) \in A$, then at each iteration node i can mark the smallest unmarked distance as being a shortest distance. Initially, node i marks any distance $\ell(i, \hat{j})$ s.t. $\ell(i, \hat{j}) = \min\{\ell(i,j) \mid j \in AL(i)\}$ as being a shortest distance. The proof of correctness of this is completely analogous to the proof of correctness of Dijkstra's algorithm. (see II.1) Thus at iteration m , node i will have at most $n-m+1$ possible improved distances to broadcast. This reduces the communication cost by approximately a factor of 2 since $\sum_{j=1}^n n-j \approx \frac{n^2}{2}$ rather than n^2 .

Notice that if $\ell(i,j) = a$ positive constant $\forall (i,j) \in A$, then algorithm SP1 with improvements essentially reduces to the minimum hop algorithm MH1 (see III.2) and only $O(Ln)$ distance messages are transmitted.

Unfortunately, the $O(Ln^2)$ upperbound is tight even if $\ell(i,j) > 0, \forall (i,j) \in A$. In particular, the algorithm is neither $O(L^2)$ nor $O(n^3)$, either of which is better than $O(Ln^2)$. Examples to demonstrate the tightness of the $O(Ln^2)$ upperbound are presented in Appendix A. One could conceivably use other modifications or pre-processing (for example, assume the nodes know the topology of the network) to further reduce the constant factor so that $CC(SP1) \leq cLn^2$, with $c < 2$, or to efficiently encode distances. However, such discussion has been omitted since there are asymptotically more efficient algorithms for graphs with positive arclengths. In III.5, we will discuss methods for making these algorithms as efficient as possible

in terms of a bit communication cost rather than just a "message" cost.

We now turn to a discussion of two Dijkstra-like procedures that can be used if all arclengths are positive.

Algorithm SP2:

Recall that N is the set of nodes in the network and that $ALLST(k)$ is the arclength list of node k .

Each node i executes:

1. $T \leftarrow N - \{i\}$
2. $\forall j \in AL(i)$ do: $[d(i,j), NT(j)] \leftarrow [\ell(i,j), j]$
3. $\forall j \notin AL(i)$ do: $[d(i,j), NT(j)] \leftarrow [\infty, \text{blank}]$

Comment: $NT(j)$ is the identity of a neighbor through which i has a path to j of distance $d(i,j)$. Initially, $NT(j)=j$ if $j \in AL(i)$ and $NT(j)$ is undefined (blank) if $j \notin AL(i)$.

4. Let \hat{k} be such that $d(i, \hat{k}) = \min\{d(i, k) \mid k \in T\}$.
5. $T \leftarrow T - \{\hat{k}\}$
6. If $ALLST(\hat{k})$ is not in memory, send a request to \hat{k} for its arclength list. The request traverses the minimum hop path from i to \hat{k} and is answered by the first node along the path having the information.
7. $\forall s \in AL(\hat{k})$ do:
 If $\ell(\hat{k}, s) + d(i, \hat{k}) < d(i, s)$ then
 $[d(i, s), NT(s)] \leftarrow [d(i, \hat{k}) + \ell(\hat{k}, s), NT(\hat{k})]$.

8. If $T=\phi$ then stop. Else go to 4.

In parallel node i executes the following communication process:

1. When a probe of the form {Request ALLST(\hat{k})} arrives from a neighbor j do:
 - a) If ALLST(\hat{k}) is in memory, send it to j .
 - b) If ALLST(\hat{k}) is not in memory do.

Record the fact that j wants ALLST(\hat{k}). If node i has not requested ALLST(\hat{k}), send a request to the downstream neighbor in MHTI(\hat{k}).
2. When ALLST(k) is received, send it to all neighbors that have requested it.

Node i receives each arclength $\ell(r,s)$ for $r \neq i$ at most once. Further, each node i requests every other arclength list at most once. Thus

$$CC(SP2) \leq 2L(n-1) \text{ arclength messages} + n(n-1) \text{ request messages.}$$

We also observe that the algorithm can be executed completely asynchronously, and the coordination required is analogous to that of the previous asynchronous algorithms.

From a computational point of view, algorithm SP2 is exactly Dijkstra's algorithm. Since nodes only begin with local information though, they must request ALLST(\hat{k}) after \hat{k} has been labelled in order to find the next shortest path. Communication takes place over the minimum hop trees in order to insure that each arclength list is received only once by each node. Once a node j has

received ALLST(\hat{k}), a subsequent request for ALLST(\hat{k}) made by a node i that is upstream from node j in MHTI(\hat{k}), can be answered by node j . Thus from a communications point of view, algorithm SP2 is essentially equivalent to the procedure for broadcasting all arclengths to all nodes (Algorithm BAAD - see III.3) in that ALLST(\hat{k}) travels to all other nodes via paths in MHTO(\hat{k}). The notion of a request mechanism was introduced to interleave the communication and computation. However, the communication and computation structures are not coordinated since the minimum hop trees and shortest paths trees need not be the same. Thus, algorithm SP2 does not fully exploit the distributed character of the shortest path problem that arises from the optimality principle. This principle implies that if the path $[i_2, \dots, i_k]$ has been examined by node i_2 and found to be non-optimal, then the path $[i_1, i_2, \dots, i_k]$ need not be examined by i_1 since it cannot be optimal. Thus if the path $[i_1, i_2, \dots, i_{k-1}]$ is a shortest path and node i_1 asks node i_2 for ALLST(i_{k-1}), node i_2 should not give the arclength $\ell(i_{k-1}, i_k)$ to node i_1 since the path $[i_1, i_2, \dots, i_k]$ is not optimal. Essentially, node i_2 can "prune" a part of node i_1 's potential shortest path tree. We now present the algorithm formally.

Algorithm SP3: [10]

Each node i executes:

1. $T \leftarrow N - \{i\}$

2. $P \leftarrow \{i\}$
3. $\forall j \in AL(i) \text{ do: } [d(i,j), NT(j)] \leftarrow [\ell(i,j), j]$
4. $\forall j \notin AL(i) \text{ do: } [d(i,j), NT(j)] \leftarrow [\infty, \text{blank}]$
5. CANDIDATES $(i,i) \leftarrow ALLST(i)$
6. $\forall s \neq i \text{ do: } CANDIDATES (i,s) \leftarrow \emptyset$
7. $\forall j \in N - \{i\} \text{ do: } FATHER(j) \leftarrow i$

Comment:

CANDIDATES (i,s) is the set of arclengths $\ell(s,k)$ that are useful to node i , that is, those arclengths that correspond to arcs which are in shortest paths or potentially in shortest paths from node i to other nodes. Since node i initially knows only its own arclengths, CANDIDATES $(i,i) \leftarrow ALLST(i)$ and CANDIDATES $(i,s) \leftarrow \emptyset$ for $s \neq i$.

FATHER (i,s) is the immediate predecessor of node s in the path from i to s having a length equal to the present value of $d(i,s)$,

FATHER (i,s) must be remembered in order to do pruning.

8. $m \leftarrow 1$
9. Let $\hat{k}_1, \dots, \hat{k}_{b(m)}$ be such that $d(i, \hat{k}_j) = \min\{d(i,k) \mid k \in T\}$ for $1 \leq j \leq b(m)$.

Comment:

The nodes $\hat{k}_1, \dots, \hat{k}_{b(m)}$ are the unlabelled nodes that are closest to i . The number $b(m)$ will possibly vary on each iteration. We have previously presented Dijkstra's algorithm with only one node

at a time being processed, that is, a node i would select any node that minimized tentative distances. However, it is certainly correct and potentially faster to label all nodes $\hat{k}_1, \dots, \hat{k}_{b(m)}$ at once.

10. For $j=1$ to $b(m)$ do:

$T \leftarrow T - \{\hat{k}_j\}$

$P \leftarrow P \cup \{\hat{k}_j\}$

If ALLST(\hat{k}_j) is not in memory, request it from NT(\hat{k}_j)
end;

11. For $j=1$ to $b(m)$ do:

For each arclength $\ell(\hat{k}_j, s)$ received do:

If $d(i, \hat{k}_j) + \ell(\hat{k}_j, s) < d(i, s)$ then do:

CANDIDATES(i, \hat{k}_j) \leftarrow CANDIDATES(i, \hat{k}_j) \cup $\{\ell(\hat{k}_j, s)\}$

If $d(i, s) < \infty$ then

CANDIDATES($i, \text{FATHER}(s)$) \leftarrow CANDIDATES($i, \text{FATHER}(s)$) -
 $\{\ell(\text{FATHER}(s), s)\}$

$[d(i, s), \text{NT}(s)] \leftarrow [d(i, \hat{k}_j) + \ell(\hat{k}_j, s), \text{NT}(\hat{k}_j)]$

FATHER(s) \leftarrow \hat{k}_j

end;

end;

end;

Comment:

Pruning is done in step # 11. If $d(i, s) < \infty$ then $d(i, s)$ is the length of an actual path in the graph going from i to s through FATHER(s). If $d(i, \hat{k}_j) + \ell(\hat{k}_j, s) < d(i, s)$ then the path from i to s through \hat{k}_j is better and so we delete $\ell(\text{FATHER}(s), s)$ from CANDIDATES($i, \text{FATHER}(s)$).

12. For $j=1$ to $b(m)$ do:

If there are any requests for $ALLST(\hat{k}_j)$, send

$CANDIDATES(i, \hat{k}_j)$

end ;

13. If $T=\emptyset$ stop. Else do: $m \leftarrow m+1$

Go to 9.

In parallel, node i executes the following communication process.

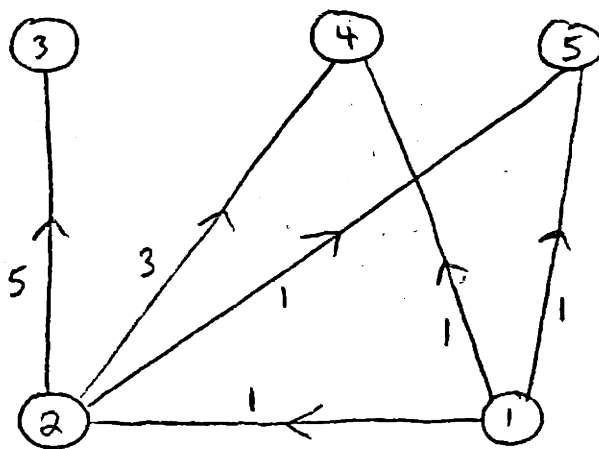
1. If a neighbor j requests $ALLST(r)$ then do:

If $r \in P$, send j $CANDIDATES(i, r)$.

Else record the fact that j has requested $ALLST(r)$.

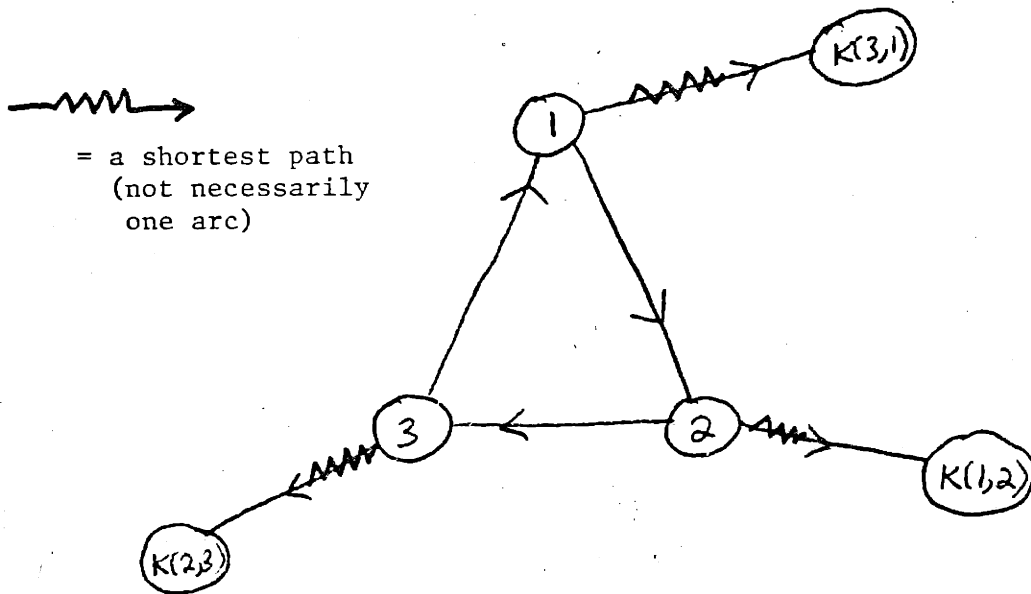
Observe that as is the case with algorithm SP2, each node i can learn the number of nodes in the network as it executes the procedure. We further observe that algorithm SP3 can be executed completely asynchronously. Any node may choose to begin an update and "awaken" its neighbors. These nodes in turn begin their updates and wake up their neighbors etc. However, there is a potential for deadlock that does not arise with SP2. There, a node i immediately forwards any request for $ALLST(\hat{k})$ on to its downstream neighbor in $MHTI(\hat{k})$, if node i does not have the information. The request can keep propagating down the tree, but can eventually be answered since node \hat{k} knows $ALLST(\hat{k})$ by assumption. With SP3 though, a node j passes on $ALLST(k)$ to some neighbor i that requested it, only after j has processed it (by process, we mean execute step 11 of SP3 on the

arclengths received). In fact, the whole point of pruning is that if i has a shortest path to \hat{k} through j , j should process $ALLST(\hat{k})$ before i does. If transmission delays are arbitrary, j may not have processed $ALLST(\hat{k})$ by the time that i wants it. This point is illustrated by the following example:



Node 1 initially labels nodes 2,4, and 5 and node 2 initially labels 5. After node 1 requests, receives, and processes the arclengths from 2,4, and 5 (by process we mean execute step 11 of algorithm SP3 on the arclengths received from 2,4, and 5), it realizes that the path $P[1,3]=[1,2,3]$ is the next best shortest path, and it asks

node 2 for ALLST(3). If node 5 is slow in responding to 2's request for ALLST(5), node 2 will not have processed ALLST(3) by the time node 1 requests it, i.e. 2 is waiting on 5 before it proceeds. This leads one to suspect that there is the possibility of having a cycle $[i_1, \dots, i_k = i_1]$ such that node i_ℓ has requested some information from node $i_{\ell+1}$, but $i_{\ell+1}$ does not have it, $1 \leq \ell < k-1$. In this case, there will be a deadlock. Fortunately, this cannot occur if all arclengths are positive. We show this for a 3 node cycle.



In the above figure, $k(1,2)$ is the node whose arclength list has been requested by 1 from 2, that is $d(1, k(1,2)) = \ell(1,2) + d(2, k(1,2))$. $k(2,3)$ and $k(3,1)$ are analogously defined. Since algorithm SP3 processes nodes in order of increasing distance from the source, the

fact that 2 has not processed (i.e. executed step 11) the arclengths of $k(1,2)$ means that $d(2,k(2,3)) \leq d(2,k(1,2))$. If the deadlock exists this inequality must hold for the other node pairs as well, and we obtain

$$\begin{aligned}d(1,k(3,1)) &\geq d(1,k(1,2)) \\d(2,k(1,2)) &\geq d(2,k(2,3)) \\d(3,k(2,3)) &\geq d(3,k(3,1))\end{aligned}\tag{III.4-1}$$

But if $\ell(i,j) > 0, \forall (i,j) \in A$ we also have

$$\begin{aligned}d(1,k(1,2)) &> d(2,k(1,2)) \\d(2,k(2,3)) &> d(3,k(2,3)) \\d(3,k(3,1)) &> d(1,k(3,1))\end{aligned}\tag{III.4-2}$$

Combining III.4-1 and III.4-2 we see $d(1,k(3,1)) > d(1,k(3,1))$ which is a contradiction. We also see that in fact we only need one of the inequalities in III.4-2 to be strict, i.e. only one of $\ell(1,2), \ell(2,3), \ell(3,1)$ need be strictly positive. This argument clearly extends to an arbitrary cycle and so we conclude that if $\ell(i,j) \geq 0$ and there are no zero length cycles, then in any cycle of requests, there must be at least one node that can answer the request made of it without waiting for its own request to be answered.

Communication Cost:

Each node i requests $ALLST(j)$ for $j \neq i$ and receives each arc in $ALLST(j), j \neq i$, at most once. Thus $CC(SP3) \leq 2L(n-1)$ arclengths + $n(n-1)$ request messages. Unfortunately, we have been unable to

quantify the effectiveness of pruning. This is because pruning depends very heavily on the topology of the graph, and for given values of n and $|L|$, there are many graphs on n nodes and $|L|$ links that have distinctly different topologies. One can construct graphs in which one node, even with pruning, will receive a significant number of the arclengths. This does not mean all nodes will receive a lot of arclengths. It is just difficult to quantify the interactive effects of pruning. Thus, the above bound on $CC(SP3)$ basically states that the total worst case communication cost is just n times the worst case amount of information that one node must receive. One useful "rule of thumb" is that pruning tends to be relatively more effective in those graphs that have relatively long (in terms of number of arcs) shortest paths, because there are relatively more intermediate nodes in such paths to do the pruning. Algorithms $SP2$ and $SP3$ also have an advantage over algorithm $SP1$ in that only arclengths and not path distances must be transmitted. This reduces the bit cost of a message by approximately $\log n$ bits since a path length can be $(n-1)(\ell-\max)$ which requires approximately $\log n + \log(\ell-\max)$ bits to encode.

III.5 Preprocessing and Message Encoding

In III.3, we assumed that each node knows its position in the various minimum hop trees for purposes of broadcasting all arclengths, and we viewed this as a distributed version of preprocessing. In this section, we investigate some improvements to algorithm SP3 that can be made by assuming that all nodes know the topology of the network. More specifically, we assume that each adjacency list is assigned some order and that all nodes know all ordered adjacency lists.

Firstly, we observe that for each $\langle x, y \rangle$ in L , the source node i needs to know at most one of $\ell(x, y)$ and $\ell(y, x)$. If x is labelled before y is, then the arc (x, y) is potentially in a shortest path from i through x to y , and so node i can use $\ell(x, y)$. When y is labelled, however, the arc (y, x) is of no use since node i already has a shortest path to x that doesn't visit y . If x and y are labelled at the same iteration, then neither the arclength $\ell(x, y)$ nor the arclength $\ell(y, x)$ is of any use to node i . The problem is that if the shortest path from i to x goes through a neighbor of i , say j_x , and the shortest path from i to y goes through j_y , with $j_y \neq j_x$, then j_y may not necessarily know that i has already labelled x and hence j_y will give i the arclength $\ell(y, x)$. Thus we need an efficient way for i to tell its neighbors which arclengths are potentially useful. To do this, we introduce the notion of candidate bit vectors. Suppose that the ordered adjacency list

of a node p is $\{p_1, \dots, p_{D(p)}\}$ where $D(p)$ is the degree of p . Then, when i labels p and requests the arclengths of node p from a neighbor of i , say j , node i gives j a vector of $D(p)$ bits with a 1 in the k^{th} position iff node p is unlabelled by i , i.e. iff i does not yet know a shortest path to p_k . The information in the bit vector allows j to know which arcs are still candidates as far as i is concerned. We further note that node i may even be able to use fewer than $D(p)$ bits.

Suppose that

- 1) $p_k \in AL(p)$ and $p_k \in AL(s)$.
- 2) The shortest paths from i to both s and p go through a neighbor j .
- 3) Node i labelled nodes p_k, s, p in that order and s was the first node that i labelled after p_k such that the shortest path from i to s went through j .

Then, when i requested the arclengths of s from j , i gave a bit to j that indicated p_k was already labelled. Hence when i labels p and asks j for the arclengths of p , i does not even have to give a bit for the arc (p, p_k) since j already knows it cannot be useful to i . To accomplish this, nodes i and j must both maintain a list, say P_{ij} , of those nodes that j knows i has labelled. Node i of course maintains its own list, say P_i of all nodes it has labelled. When i labels p and asks j for the arclengths of p , it constructs the following bit vector:

Suppose $AL(p) = \{p_1, \dots, p_{D(p)}\}$, and $p_k \in AL(p)$. There are 3 cases.

- 1) $p_k \in P_{ij}$. Node i does not need to give a bit since j knows i has labelled p_k .
- 2) $p_k \in P_i - P_{ij}$. The corresponding bit is 0 since i has labelled p_k and so doesn't need the arclength $\ell(p, p_k)$.
- 3) $p_k \in N - P_i$ where $N =$ set of nodes in the network. In this case, the arclength $\ell(p, p_k)$ is possibly useful and the bit is 1.

Now j receives a vector of bits b_1, \dots, b_s ($s \leq |AL(p)|$) where a bit b_r corresponds to a node p_{t_r} such that p_{t_r} is the r^{th} node in $AL(p)$ that is not in P_{ij} . If $b_r = 0$, j knows i has labelled p_{t_r} since i last requested some arclengths from j . In this case j adds p_{t_r} to P_{ij} . If $b_r = 1$, j knows that $\ell(p, p_{t_r})$ is potentially useful to i and so gives the arclength to i unless it (i.e., j) already pruned it.

(Recall that by the optimality principle, whatever is not useful to j cannot be useful to i). The exact number of bits, which is $|AL(p)| - |AL(p) \cap P_{ij}|$, depends very much on the topology, and so we

are unable to make any general statement. We can only say that a source node i will give at most $\sum_{\substack{s=1 \\ s \neq i}}^n |AL(s)|$ bits in total, i.e.

one bit vector of at most $|AL(s)|$ bits when it requests the arclengths of s , for each $s \neq i$. Hence all nodes use at most $2L(n-1)$ bits for all candidate bit vectors. With this scheme, each node i will receive at most $L - |AL(i)|$ arclengths and so overall, at most $L(n-2)$

arclengths are transmitted. As before, any other pruning that occurs will reduce this cost even further. If an arclength message uses b bits, then this scheme uses the fraction

$$\frac{2L(n-1) + L(n-2)b}{2L(n-1)b} \approx \frac{b+2}{2b} \quad \text{as } L, n \rightarrow \infty$$

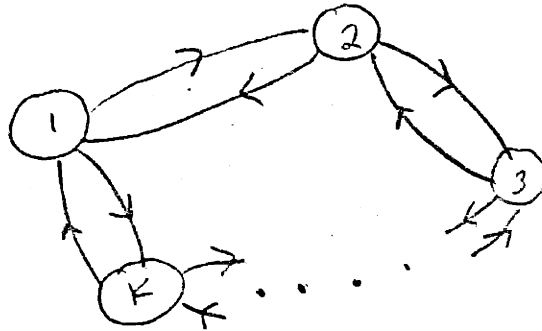
of the number of bits used by algorithm SP3.

Another improvement that can be made with preprocessing is in the encoding of arclength messages. We have previously said that an arclength can be specified as a triple $(x, y, \ell(x, y))$ in $2\log n + \log(\ell - \max)$ bits. Now if node i requests the arclengths of x from j , node j clearly does not have to specify x in the triple since i knows x . We further observe that if each node knows the topology, then node j need only specify the identity of y relative to other nodes in $AL(x)$. This can be done in $\log(D(x))$ bits, where $D(x)$ = degree of x . For sparse graphs, i.e. $D(x) \ll n$, this results in an improvement by a factor $\frac{\log(D(x))}{\log n}$ bits and may be significant.

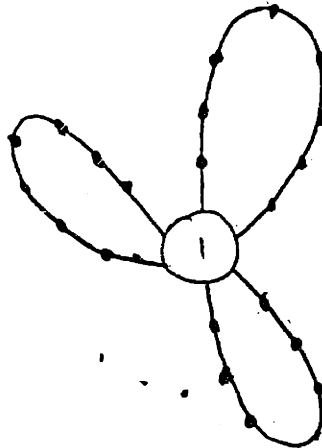
IV. AVERAGE COMMUNICATION COST ANALYSIS

IV.1 Motivation

Thusfar, we have developed an [Ln arclength messages + $2Ln$ bits] distributed shortest path algorithm and a $2Ln$ arclength messages algorithm for broadcasting all arclengths to all nodes. (Algorithm BAAD - Section III.3). Observe however, that the latter procedure has a constant cost over all networks with n nodes, L links, and any arclength assignment, whereas the pruning of algorithm SP3 may reduce the cost of that algorithm to be asymptotically smaller than $O(Ln)$, at least for certain topologies. Still, it is somewhat disappointing that we have not found a shortest path algorithm whose worst case communication cost is provably less than that of algorithm BAAD. More specifically, is there is a distributed shortest path algorithm whose communication cost is upperbounded by αn^2 arclength messages for some constant α ? We note that for classes of sparse graphs (say the degree of each node \leq a constant β that is independent of n), the $O(Ln)$ bound is $O(n^2)$. However the constant multiplying the n^2 term increases with β . The following "plausibility argument" is meant to indicate to the reader some of the reasons that will make it difficult to ever find an $O(n^2)$ algorithm for sparse graphs, if we have each node execute the same single source algorithm. (We term such algorithms homogeneous because all nodes execute the same procedure). Consider the following ring network.



It is plausible that for this graph, node 1 must examine at least $k-1$ arclengths in the worst case, even if node 1 knows the topology. Hence, it is plausible that for the following graph, node 1 must examine $O(L)$ arclengths in the worst case.



Even with pruning and knowledge of the topology, node 1 must in the worst case resolve all potential ties created by elementary cycles. Since our techniques do not really enable us to distinguish one source from another, we can only say that the worst case total is just n times the worst case for one node. This yields the $O(Ln)$ bound.

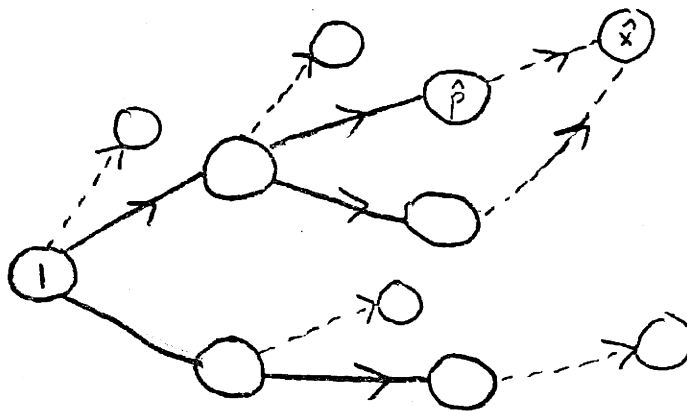
As we see, the crux of the problem lies in precisely determining the interactions of the various single source problems, solely in terms of n and L . This appears to be difficult in view of the large number of different topologies that can exist. The properties of homogeneity and topology independence are useful in practice since they permit the same program to be used at all nodes, regardless of topology. While one could conceivably design a more efficient algorithm for a particular network, this approach seems to be somewhat impractical since this algorithm may be very inefficient on some other network. Because the addition or deletion of a few choice nodes can significantly alter some topologies, one may have to develop a new reoptimized algorithm for each new graph. In our algorithms, the nodes may indeed know the topology, but nothing special is assumed about it, i.e. the topology is part of the input.

With this in mind, we consider the communication cost of a distributed shortest path algorithm when averaged over random arclengths. This analysis is motivated by the fact that the lengths

assigned to the links of data networks for purposes of routing may be appropriately modelled as random variables. Since shortest path updates may be performed relatively frequently, our average case analysis will perhaps be relevant to the average communication cost of performing these updates.

IV.2 Average Computation Analysis of Spira's Algorithm

We briefly review Spira's algorithm. (See II.1 for a detailed presentation.) Spira's procedure is similar to Dijkstra's except that arcs from a particular node are examined one by one in order of increasing arclength. Recall that this enables us to find the next best path in only $O(\log n)$ comparisons using played binary trees. Suppose node 1 is the source. At each stage of the algorithm, there is a tree (rooted at node 1) of shortest paths to j other labelled nodes. In addition, from each labelled node p , there is a one arc extension which is the last arc in the next best path that is a one arc extension of the path from 1 to p .



—————> Permanent Arcs
-----> Tentative Arcs

We find the shortest of all such paths say $[1, \dots, \hat{p}, \hat{x}]$, where \hat{p} is in the tree and (\hat{p}, \hat{x}) is the one arc extension. If \hat{x} is unlabelled (new) then we have found a shortest path from 1 to \hat{x} , and the tree grows by one node. If \hat{x} is already labelled, this path is of no use since we already have a shortest path to \hat{x} . Thus the number of iterations (i.e. # of times we play the tree to find the next best path) depends on how often we are unlucky and find a path to a labelled node. Fortunately, the average number of such unlucky trials can be computed. More precisely we have:

Lemma: [5]

Let $G = (N, A, \ell)$ be a weighted digraph on n nodes such that

- 1) $A = \{(i, j) \mid i \neq j\}$
2. The arclengths are independent, identically distributed nonnegative random variables.

Then the average number of iterations Spira's algorithm makes to solve the single source problem for any source node i is $O(n \log n)$ provided ties are broken randomly when the arclength lists are sorted and when the binary tree is played to find the next best path.

Remark: Property 1) simply means that G contains all possible $n(n-1)$ arcs. G is called a clique on n nodes. The random tie breaking

rule is important. In [11], it is shown that for a certain "plausible" deterministic, tie breaking rule, $O(n^2)$ iterations may be used on the average for certain probability distributions on the arclengths.

Proof: Let z_j be a $\{0,1\}$ valued random variables s.t. $z_j=1$ iff the path we examine leads to a new node given that we have found shortest paths to j nodes. Now we claim that

$$p_r [z_j=1] \geq \frac{n-j}{n} \quad \text{IV.2-1}$$

Let p be one of the j labelled nodes. The set of unexamined arcs of p can be partitioned into two subsets.

UAO = {unexamined arcs leading to old nodes}

UAN = {unexamined arcs leading to new nodes}

Now we observe that $|UAO| + |UAN| \leq n-1$ (we may already have examined and discarded some arcs and in this case $|UAO| + |UAN| < n-1$) and $|UAN| = n-1-j$ since every node has $n-1$ outgoing arcs and only j nodes are labelled. Because all arclengths are i.i.d. random variables and ties are broken randomly, the next one arc extension from p is equally likely to go to any node that is the sink of an unexamined arc. Hence the probability that this one arc extension from p goes

to a new node is just $\frac{|UAN|}{|UAO| + |UAN|} \geq \frac{n-1-j}{n-1} \geq \frac{n-j}{n}$. Now this

argument holds for any of the labelled nodes and so IV.2-1 follows.

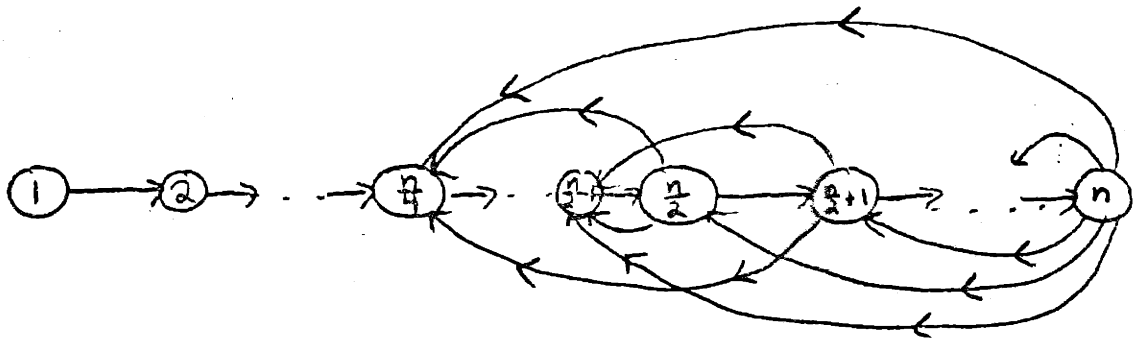
Hence the expected number of paths we must examine before we find a shortest path to a new node, given that we have labelled j nodes is $\leq \frac{n}{n-j}$. Summing over all j we obtain that the overall average number

$$\text{of iterations is } \leq \sum_{j=1}^{n-1} \frac{n}{n-j} = O(n \log n)$$

Since the graph is a clique, this argument clearly holds for any source i . ||

Because the bound holds for any source i and because the number of comparisons on each iteration is $O(n \log n)$ (using played binary trees), we conclude that the all pairs problem for a clique with random nonneg. arclengths can be solved as n single source problems on the average in $O(n^2 \log^2 n)$ comparisons + # comparisons needed to sort the arclength lists. Recall that this latter quantity is just $O(n^2 \log n)$. Even for a single source problem we must sort all arclength lists and so in this case, the $O(n^2 \log n)$ sorting cost dominates the $O(n \log^2 n)$ cost of performing the rest of the algorithm for that source. In the all pairs problem, the $O(n^2 \log^2 n)$ cost is dominant, and the sorting is worthwhile. In his paper, Spira also shows that the variance in the number of iterations for a single source problem is at most $3n^2$. However, the number of iterations required for the various single source problems need not be independent. Thus, he can only bound the variance of the total number of iterations for all sources by $3n^4$.

We now observe that, unfortunately, this $O(n \log n)$ results does not necessarily hold if G is not a clique. There are two basic problems. Firstly, since every node does not necessarily have an arc to every other node, $\frac{|UAN|}{|UAN|+|UAO|}$ is not necessarily lower bounded by $\frac{n-j}{n}$. The exact value depends on the topology. In fact, for the following graph, the expected number of iterations to solve the SS problem is $O(n^2)$, when each of the nodes $1, \dots, n/4$ is a source. Thus the overall number of iterations is $O(n^3)$, for the AP problem.



In this graph, the sets of arcs is

$$\{(i, i+1) \mid 1 \leq i \leq n-1\} \cup \{(i, j) \mid \frac{n}{4} < j < i, \quad i \geq \frac{n}{2}\} .$$

For a source k , $k < \frac{n}{4}$, the nodes will initially be labelled in the order $k, k+1, \dots, n/2$. Once we reach $\frac{n}{2}$ however, only $O(\frac{1}{n})$ of the arcs lead to new nodes. Hence, we will examine $O(n)$ arcs out of

$\frac{n}{2}$, on the average. The same clearly holds once any $j \geq \frac{n}{2}$ is labelled.

So, for a source k , we will examine $\frac{n}{2} \cdot O(n)$ arcs $= O(n^2)$ arcs. The second problem is that the average will not necessarily be the same for all sources in general. In the above example, if we take node n as the source, we immediately have many arcs to new nodes and thus the average will be different from that when node 1 is a source.

Spira's result does hold however, if we average over random graphs, because the averaging enables us to make a statement about the probability that a certain arc leads to a new node. More precisely we have:

Lemma: Let n and γ be given positive integers with $0 < \gamma \leq n(n-1)$, and let LENGTHS be a finite set of nonnegative real numbers. Let G be the collection of all weighted cliques on n nodes s.t. for any $G \in G$, there is a subset A_G of the arcs of G satisfying

- 1) $|A_G| = \gamma$
- 2) $(i,j) \in A_G \Rightarrow \ell(i,j) \in \text{LENGTHS}, (i,j) \notin A_G \Rightarrow \ell(i,j) = \infty$

(Note for $G, G' \in G$ A_G is not necessarily equal to $A_{G'}$, but $|A_G| = |A_{G'}|$. Also $|\text{LENGTHS}| < \infty \Rightarrow |G| < \infty$). Assign the (discrete) uniform probability measure to G . Then under the assumption of random tie breaking rules for sorting arclength lists and playing

binary trees, Spira's algorithm uses an average of $O(\min(\gamma, n \log n))$ iterations to solve the single source problem for any source.

Proof: The assumptions of random tie breaking rules and a uniform probability measure on G imply that for any node i , all possible orderings of the destinations of arcs leaving i (where the ordering is in terms of arclengths) are equiprobable. This implies IV.2-1 holds and the $n \log n$ part follows as before. Now, observe that if the length of the shortest one arc extension from a certain node is infinite, Spira's algorithm can be modified to consider that node as being "blocked". In particular, on a clique of n nodes in which only γ arclengths are finite Spira's algorithm need only perform γ iterations. At that point the length of a shortest path to each node that has not been labelled must be ∞ . Thus the average number of iterations is $O(\min(\gamma, n \log n))$. ||

The previous lemma effectively embeds the collection of random graphs with γ arcs and random arclengths into the collection of random weighted cliques that have exactly γ finite arclengths.

IV.3 Applications to Distributed Algorithms

We now present a distributed implementation of Spira's algorithm. Our discussion will only informally outline the basic iteration since the details of the initialization and data structures are similar to those of previous algorithms. Again, $NT(x)$ denotes the neighbor through which the source has a shortest path to x , T is the set of nodes to which the source has not found shortest paths, and for a source i , $CANDIDATES(i,x)$ is the ordered set of arclengths of x that are useful or potentially useful to i . The ordering, \succ , on $CANDIDATES(i,x)$ is defined as follows: $\ell(x,y) \succ \ell(x,z)$ iff either $\ell(x,y) > \ell(x,z)$, or $\ell(x,y) = \ell(x,z)$ and i received $\ell(x,y)$ after it received $\ell(x,z)$. When a node initially sorts its own arclength list, it breaks ties randomly, i.e. if $\ell(x,x_1) = \dots = \ell(x,x_k)$, then x randomly (uniformly) chooses any one of the $k!$ possible orderings.

Algorithm SP4

Each node i executes:

1. Let $[i, \dots, r, x]$ be the next smallest path as determined by playing the binary tree of path distances. (Hereafter, we will say that node i examines the path $[i, \dots, r, x]$).
2. If $x \in T$ then do:
 - a. $T \leftarrow T - \{x\}$
 - b. If $r=i$ then $NT(x) \leftarrow x$.
 - c. Else $NT(x) \leftarrow NT(r)$

- d. Ask nodes $NT(r)$ and $NT(x)$ for the next smallest arclengths of r and x respectively.
 - e. end
3. Else do:
- a. $CANDIDATES(i,r) \leftarrow CANDIDATES(i,r) - \{\ell(r,x)\}$
 - b. Ask node $NT(r)$ for the next smallest arc out of r .
 - c. end
4. When the requested arclength(s) arrive, add it (them) to the appropriate CANDIDATE list (s). If there are any outstanding requests for the arclengths for r and/or x , send the appropriate information.
5. If $T = \emptyset$ then stop. Else go to 1.

In parallel, node i executes the following communication process:

When a neighbor j requests the next smallest arclengths of k do:

Suppose $\ell(k,x)$ is the arclength that i most recently gave j .

- a. If there is an arclength $\ell(k,y)$ in $CANDIDATES(i,k)$ s.t. $\ell(k,y) > \ell(k,x)$, then send j the smallest (in terms of $>$) such arclength $\ell(k,y)$.
- b. Otherwise, record the fact that j has requested the next smallest arclength of k .

Comment: Step a) of the communication process requires that node i know the arclength $\ell(k,x)$, which is the arclength of k that i most recently sent

to j . Either node i can remember this, or node j can supply this information as part of the request. Also, if there are no more useful arcs out of a certain node, then the next smallest arclength is effectively infinite, and node i can send a message to this effect.

Deadlock Problems:

As was the case with algorithm SP3 (See III.3), one may suspect that algorithm SP4 may deadlock. Again, we will prove that if all arclengths are positive, then for any cycle $[i_1, i_2, \dots, i_s = i_1]$ such that i_ℓ has requested some information from $i_{\ell+1}$, at least one of the nodes will be able to supply its neighbor with the requested information. That this is sufficient to guarantee that the algorithm is deadlock free can be seen as follows. At any instant of (real) time, the set of nodes N can be partitioned into three subsets

COMP = {nodes that are computing}

WAIT = {nodes that are waiting for new information}

FIN = {nodes that have finished}.

If there is never a time when the nodes that are in N-FIN are all waiting for new information from other nodes in N-FIN, then the fact that Spira's algorithm terminates correctly after finitely many iterations together with the fact that any node in FIN can answer any request

without waiting for more information implies that all nodes will eventually finish. We also observe that if node i examines some path $[i, j]$ and requests the next arclengths, then this request can be answered without waiting since by assumption i and j know their own arclengths.

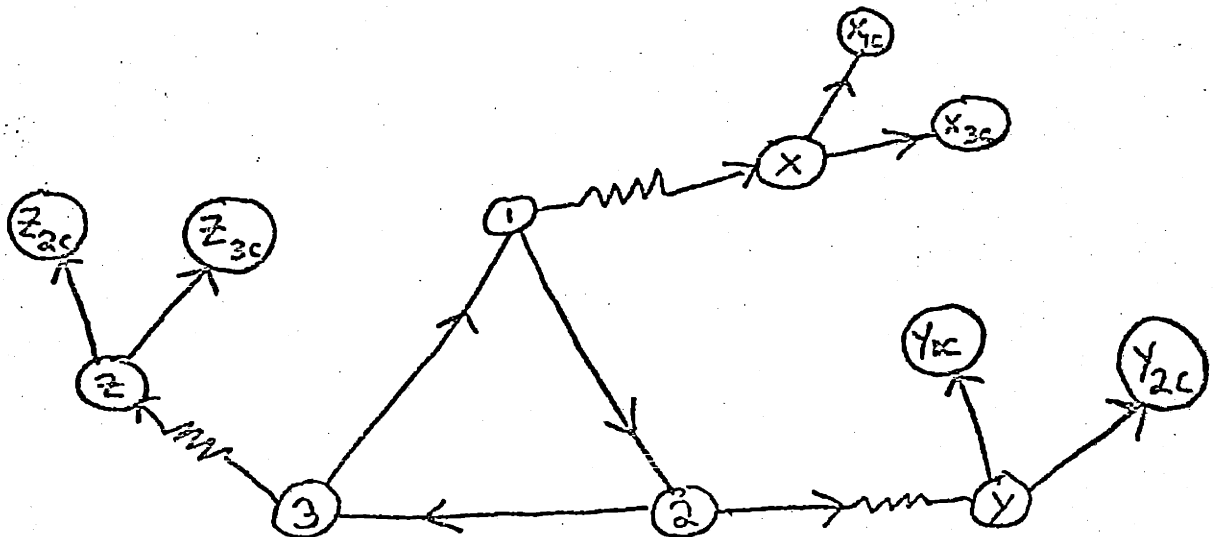
Thus a deadlock can occur only if there is a time when $COMP = \phi$, $|WAIT| \geq 2$, and each node i in $WAIT$ is waiting to receive a new arclength of r_i from $NT(r_i)$, where $NT(r_i) \in WAIT$, $NT(r_i) \neq i$, and $NT(r_i) \neq r_i$. Now since $|WAIT| < \infty$, the "pigeon hole" principle implies that there must be a cycle of requests $[i_1, \dots, i_s = i_1]$. So assuming the result that at least one of the nodes in the cycle can answer, we conclude that at least one of the nodes in the cycle will receive the information it requested and will move from $WAIT$ to $COMP$. (We note that the previous argument is valid only if transmission delays are all finite. They can be arbitrary, but they must be finite). So now we turn to proving the result about a cycle of requests.


We first observe that in algorithm SP3, (III.3), if a node i chooses a shortest path $P[i, x] = [i, NT(x), \dots, x]$ from itself to x , then node $NT(x)$ must in fact choose the subpath of $P[1, x]$ from itself to x as a shortest path. This fact can be proved by induction on the number of arcs in $P[1, x]$, and we say that algorithm SP3, computes consistent shortest paths. However, algorithm SP4 need not compute consistent shortest paths if ties are broken randomly. Recall that this random tie breaking rule was used in obtaining the average time bounds of section IV.2. While it may be possible to devise a deterministic tie breaking rule which insures that

SP4 computes consistent shortest paths, the proofs of IV.2 will no longer be correct as they are. However, we note that the proof of deadlock freedom for SP3 did not use the consistency property. Fortunately, we will also not need it to prove the result on cycles of requests for algorithm SP4. It has been mentioned only because one might be tempted to invoke it and because it may bear on the manner in which the general routing strategy uses the shortest paths.

As in III.3, we will prove our result for a "canonical" three node cycle. The generalization to an arbitrary cycle should be clear.

Consider the following situation



 = a shortest path (not necessarily one arc)

in which:

- node 1 has a shortest path to y through 2
- node 2 has a shortest path to z through 3
- node 3 has a shortest path to x through 1

and

x_{1c} is the current one arc extension from x in node 1's tree
 x_{3c} " " " " " 3's tree
 y_{1c} " " " y " 1's tree
 y_{2c} " " " " " 2's tree
 z_{3c} " " " z " 3's tree
 z_{2c} " " " " " 2's tree

and

1 transitions into the WAIT state after examining the path through y to y_{1c}
 2 " " " " " " z " z_{2c}
 3 " " " " " " x " x_{3c}

Now because 1 receives arclengths of y from 2

" 2 " z from 3
 " 3 " x from 1

the following inequalities must hold

$$\begin{aligned}
 \ell(x, x_{1c}) &\geq \ell(x, x_{3c}) \\
 \ell(y, y_{2c}) &\geq \ell(y, y_{1c}) \\
 \ell(z, z_{3c}) &\geq \ell(z, z_{2c})
 \end{aligned}
 \tag{IV.3-1}$$

We claim in fact that at least one of the inequalities in IV.3-1 must be a strict inequality. Suppose that they are all equalities. Then because the nodes examine paths in order of nondecreasing length, it must be that

$$\begin{aligned}
 d(1, x) + \ell(x, x_{1c}) &\geq d(1, y) + \ell(y, y_{1c}) = d(1, y) + \ell(y, y_{2c}) \\
 d(2, y) + \ell(y, y_{2c}) &\geq d(2, z) + \ell(z, z_{2c}) = d(2, z) + \ell(z, z_{3c}) \\
 d(3, z) + \ell(z, z_{3c}) &\geq d(3, x) + \ell(x, x_{3c}) = d(3, x) + \ell(x, x_{1c})
 \end{aligned}
 \tag{IV.3-2}$$

where $d(i, j)$ is the shortest distance from i to j . Now if all arclengths are positive, the fact that 1 (2,3) has a shortest path to $y(z, x)$ through 2 (3,1) implies

$$d(1,y) > d(2,y)$$

$$d(2,z) > d(3,z)$$

IV.3-3

$$d(3,x) > d(1,x)$$

Combining IV.3-2 and IV.3-3 we obtain $l(x, x_{1c}) > l(x, x_{1c})$. This is a contradiction, and so we conclude that one of the inequalities in IV.3-1 must be strict. Without loss of generality assume $l(x, x_{1c}) > l(x, x_{3c})$. Then this means that node 1 does have a new arclength of x to give to 3. We are not quite done, however, because if node 3 determines that the path from itself through 1 to x and x_{3c} is a new shortest path, it also needs a new arclength out of x_{3c} . The fact that node 1 has an arclength $l(x, x_{1c})$ s.t. $x_{1c} \neq x_{3c}$ implies that node 1 examined the path from itself through x to x_{3c} on a previous iteration. If the path $P[3, x_{3c}]$ examined by 3 is a shortest path for 3, the optimality principle implies that the subpaths 3 of $P[3, x_{3c}]$ that go from 1 to x and x_{3c} must also be shortest paths. Now even if 1's shortest path to x , $\tilde{P}[1, x]$, is not a subpath of $P[3, x_{3c}]$, it must have the same length as the subpath of $P[3, x_{3c}]$ that goes from 1 to x . Hence the path $\tilde{P}[1, x_{3c}] = (\tilde{P}[1, x]$ followed by $[x, x_{3c}])$ must be a shortest path. When 1 examined $\tilde{P}[1, x_{3c}]$ either

- a) it already had another shortest path to x_{3c} or
- b) it determined $\tilde{P}[1, x_{3c}]$ is a shortest path and waited for an arclength of x_{3c} before doing its next iteration.

In either case, node 1 must now also now have an arclength of x_{3c} . Thus node 1 can supply node 3 with all the information that 3 needs to perform its next iteration. ||

Remarks:

- a) One can now see why it is not necessary for algorithm SP4 to compute consistent shortest paths in order to be deadlock free. Even if 1 does not choose a shortest path to x that is consistent with the shortest path to x chosen by 3, the strict inequality $d(3,x) > d(1,x)$ must still hold if $\ell(3,1) > 0$.
- b) The previous proof works even with zero length arcs as long as there are no zero length cycles.

Worst Case Communication Cost:

Each node requests and receives each arclength at most once, Therefore, $CC(SP4) \leq 2Ln[\text{arclength messages} + \text{request messages}]$. The exact bit cost of a message depends on the details of the implementation (see comment after description of SP4). In any case, the cost of each message will be $O(\log n + \log(\ell - \max))$ bits.

Application of Results of IV.2

There is an almost exact correspondence between the number of arclengths a node receives and the number of iterations of Spira's algorithm it makes. (More precisely, # arclengths received \leq # iterations + n. A node may receive some arclengths but finish before it uses them.) Therefore, the results of Section IV.2 apply to the average communication cost. However, one must be careful when interpreting their significance, and there are several points that merit elaboration.

The first result of IV.2 stated that for a clique of n nodes with i.i.d. random arclengths, a source performs $O(n \log n)$ iterations (average). We showed, though, that this need not apply to an arbitrary graph. Since many networks are not cliques (in fact connecting every pair of nodes by a link is not only often economically unfeasible, but also defeats the purpose of having a network to begin with), this result is of limited practical value. However, we observe that our counterexample relied heavily on a graph with $O(n)$ nodes of large outdegree and small indegree. It is not clear if a similar counterexample exists for symmetric, connected graphs (data networks).

The second result stated that if we average over random digraphs with γ arcs, then a similar $O(n \log n)$ average case bound holds. This result is also of limited practical value. The class of random digraphs with γ arcs includes many graphs that are not symmetric. In fact, the symmetric digraphs comprise approximately only the fraction

$$\frac{B\left(\frac{n(n-1)}{2}, \frac{\gamma}{2}\right)}{B(n(n-1), \gamma)}$$

where $B(n, k)$ is the binomial coefficient on n, k , of the total number of digraphs on γ arcs. For large n and γ this is a very small number. Thus, it is not clear if the same average case bound holds for the smaller class of symmetric graphs, and it appears to be more difficult to compute the average in this case. Our proof of the bound for general digraphs used the fact that all possible orderings (in terms of arclengths) of the

destinations of arcs leaving a node are equiprobable. In particular, knowing that $l(i,j) < \infty$ tells us nothing about the value of $l(j,i)$ relative to other arclengths of j . If we attempt to extend this result, by embedding the class of random symmetric digraphs with γ arcs into the class of random cliques with γ finite arclengths s.t. $l(r,s) < \infty \Rightarrow l(s,r) < \infty$ then given that $l(i,j) < \infty$, it is not clear that all orderings of arclengths of j are equiprobable since $l(j,i)$ must also be finite.

Even if we were able to obtain an average case bound for random symmetric graphs, the result would be of limited practical significance because the topology of a network remains relatively fixed while many shortest path updates are performed. The significant averaging occurs only over the arclength ensemble.

We now outline a distributed algorithm that exploits the properties of statistical averaging over the arclengths only, in any fixed topology. The procedure will make use of candidate bit vectors, and so we again assume that each adjacency list is assigned some order, and all nodes know all ordered adjacency lists. (It is important to realize that the orderings of the adjacency lists are entirely arbitrary and must be agreed upon by all nodes only once beforehand. Their sole purpose is to facilitate the use of bit vectors, and they can be repeatedly used on successive shortest path updates.)

On each iteration, a source node i examines a path $P[i,x]$. If $P[i,x]$ is a new shortest path, we term the event a success. So given that node i has found shortest paths to nodes n_1, \dots, n_j , the

(conditional) probability of success can be expressed as

$$\Pr[\text{success}] = \sum_{s=1}^j \Pr[P[i,x] \rightarrow n_s] \Pr[\text{success} \mid P[i,x] \rightarrow n_s]$$

where $P[i,x] \rightarrow n_s$ means that the path $P[i,x]$ is a one arc extension of the known shortest path to n_s . In general, the probabilities in each term of the sum depend upon the topology and the particular nodes n_1, \dots, n_j , and hence appear difficult to compute. However, whatever the probabilities $\{\Pr[P[i,x] \rightarrow n_s]\}$ are, they sum to one, and thus, if we are able to lower bound $\Pr[\text{success} \mid P[i,x] \rightarrow n_s]$ by a constant α that is independent of s , we obtain $\Pr[\text{success}] \geq \alpha$. Our proof of the first lemma in IV.2, in fact, used such a lower bound $\alpha = \frac{n-j}{n}$.

Now we will use candidate bit vectors to insure that whenever a node i receives the next arclength of a node j , the probability that this arc leads to a new node is $\geq \frac{1}{2}$ most of the time. Because of deadlock problems that will be discussed later, node i will have to communicate directly with node j via a minimum hop path (rather than communicate with node $NT(j)$ as in algorithms SP3 and SP4) when i wants the next smallest arclengths of j . In order to use bit vectors, each node j must also maintain a list A_{ij} of those nodes in $AL(j)$ which j knows i has labelled, for each $i \neq j$. (Node j can do this with $n |AL(j)|$ bits of storage.) Note that node j may not have to know all nodes i has labelled since i will only ask j for arclengths of j .

So the basic procedure is to have each node i execute Spira's algorithm and request new arclengths directly via minimum hop paths. However, whenever i makes a request it does the following:

Suppose i examines a path $P[i,x] = [i, \dots, r, x]$ and asks node r for its next best arclength. If more than half the nodes in $AL(r) - A_{ir}$ are unlabelled by i , i sends the request as is. Otherwise i sends r a bit vector indicating which nodes in $AL(r) - A_{ir}$, i has labelled since it last gave r a bit vector. (Node i does similarly with A_{ix} if it also needs an arclength from x .) Node r will respond with the smallest arclength $l(r,t)$ s.t. $t \in AL(r) - A_{ir}$.

Let us analyze the average communication cost under the assumption that for each node, all possible orderings of the destinations of arcs leaving that node (ordering is by arclength with random tie breaking rules) are equiprobable and independent of orderings at other nodes.

a) Bits in bit vectors:

Each time i sends r a bit vector, r learns that i has labelled at least half the nodes that remain in $AL(r) - A_{ir}$. Thus i sends r at most

$$|AL(r)| + \frac{|AL(r)|}{2} + \frac{|AL(r)|}{4} + \dots + 1 = 2|AL(r)|$$

bits in bit vectors, and so i sends at most $2 \sum_r |AL(r)| = 4L$ bits in total for bit vectors.

b) # of arclengths received by i:

As previously mentioned, (# arclengths received by i) \leq (# iterations of Spira's algorithm i performs + n). Thus we compute the average # of iterations. Consider the computation phase of some iteration, say m, in which i examines a path [i, ..., r, x]. Node i received the arclength $\ell(r, x)$ from r during the communication phase of some previous iteration $m' < m$. (Iteration m' is the most recent iteration prior to m in which i examined a path that is a one arc extension from r.) Now when i received $\ell(r, x)$, x was equally likely to be any node in $AL(r) - A_{ir}$. The independence assumption implies, however, that x is still equally likely to be any node in $AL(r) - A_{ir}$ at the beginning of the m^{th} iteration (note: A_{ir} at the beginning of iteration m is the same as A_{ir} at the end of iteration m') regardless of which other paths i examined in between iterations m' and m. What has changed of course is the probability that x is an unlabelled node. Because of the use of bit vectors though, there can be at most $\lceil \log_2 |AL(r)| \rceil$ iterations involving r for which $\text{Pr}[x \text{ is unlabelled}]$ is $\leq \frac{1}{2}$. Since this holds for any r, there can be at most $\sum_{\substack{r=1 \\ r \neq i}}^n \lceil \log_2 |AL(r)| \rceil$ iterations for which $\text{Pr}[\text{success}]$ is $\leq \frac{1}{2}$. For other iterations, i finds a new shortest path with probability $\geq \frac{1}{2}$. Hence the average number of these other iterations is $\leq 2n$. Combining all this with the fact that # arclengths received \leq # iterations + n, we conclude that i

receives on the average at most

$$\sum \lceil \log_2 |AL(r)| \rceil + 3n \text{ arclengths.}$$

Since this analysis holds for any source i and since every message or bit vector traverses a minimum hop path having at most $D(\bar{G})$ arcs we obtain an average total communication cost upperbounded by

$$4 \ln D(\bar{G}) \text{ bits} + n D(\bar{G}) \left[\sum \lceil \log_2 |AL(r)| \rceil + 3n \right]$$

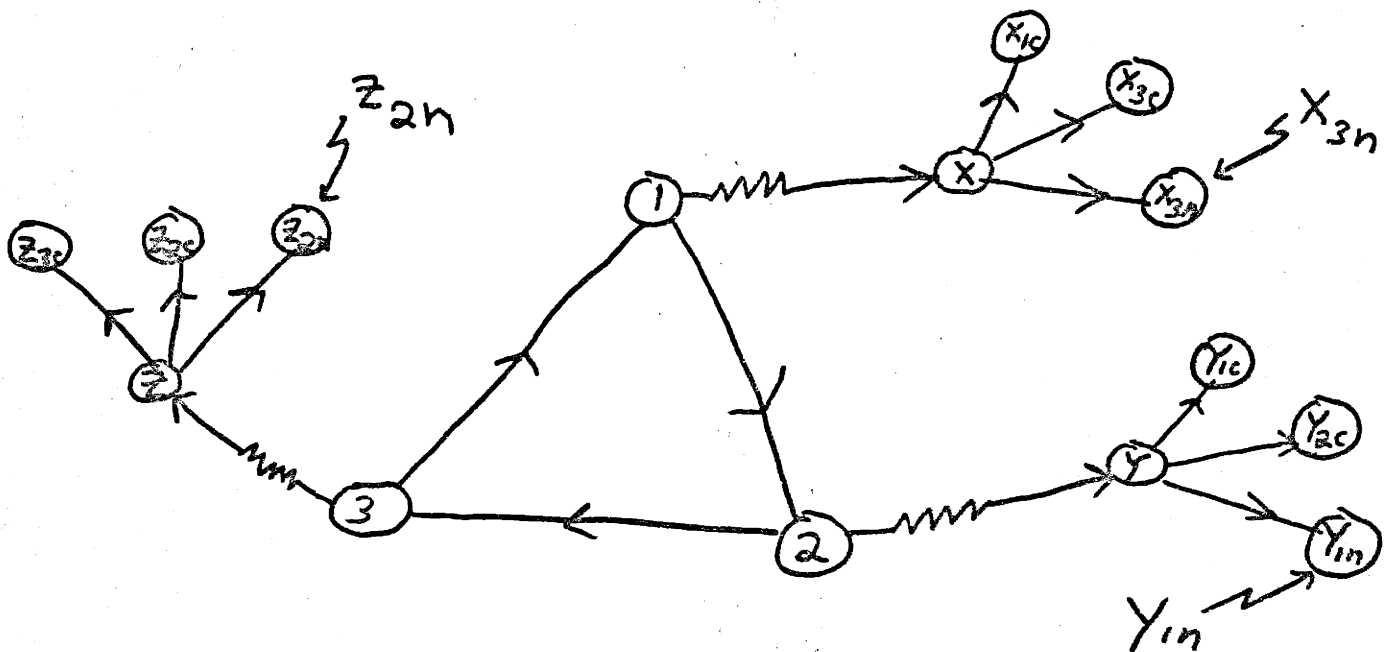
[arclength messages + request messages].

Because \log is a convex \cap function, $\sum \lceil \log_2 |AL(r)| \rceil \leq n \left(\log_2 \frac{2L}{n} + 1 \right)$

and so the expression further simplifies to

$$4 \ln D(\bar{G}) + n^2 D(\bar{G}) \left[\log_2 \frac{2L}{n} + 3 \right].$$

The reader may now ask why this procedure cannot be modified so that node i communicates with node $NT(r)$ for the arclengths of r , and thereby eliminate the $D(\bar{G})$ factor. He or she can be certain that we have tried, but as previously indicated, a serious deadlock problem arises. We describe the problem by means of another three node example:



The above example is identical to the previous example of this section with the exceptions of three new nodes x_{3n} , y_{1n} , and z_{2n} and the corresponding arcs. Now, unfortunately, one can show that the following can occur:

Node 1 examines a path $[1,2,\dots,y,y_{1c}]$ and asks 2 for the next smallest arc of y . Node 2 has a new arc, i.e. $y_{2c} \neq y_{1c}$, but node 1 already has a shortest path to y_{2c} (this path is not shown), and the next arc of y which is useful to 1 is in fact (y,y_{1n}) whose length node 2 does not yet have. Analogous situations can also hold for $(2,3,z)$ and $(3,1,x)$, and a deadlock exists.

Our previous result on cycles of requests for algorithm SP4 only guarantees that at least one of y_{2c} , z_{3c} , x_{1c} must be different from one of y_{1c} , z_{2c} , x_{3c} respectively. It does not guarantee that one of the nodes has a new arc that is also potentially useful to the requesting nodes. Were we able to eliminate the $D(\bar{G})$ factor (by some yet unknown technique), we would obtain an algorithm that uses $O(Ln)$ bits + $O(n^2 \log \frac{L}{n})$ arclength messages of communication on the average, and this represents at least a modest improvement over our $O(Ln)$ arclength messages algorithms.

V. SUGGESTIONS FOR FURTHER RESEARCH

1. New Algorithms and Better Analysis of Existing Algorithms

It would be satisfying to find a distributed shortest path algorithm which uses asymptotically less communication than algorithm BAAD (broadcasting all arclengths to all destinations). Thus far, our approach has been to decompose the problem into n interacting single source problems, and our single source algorithms have in some sense mimicked centralized algorithms (though nontrivial deadlock questions arise if one attempts to devise intelligent interactions). It is certainly possible that an entirely new approach will be needed in order to beat the $O(Ln)$ arclengths bound. The reader should be aware that centralized lower bounds apply to distributed algorithms which can be simulated on one computer, in the following sense. Suppose the distributed algorithm uses $O(X)$ computations + $O(Y)$ communications, and that it can be simulated in such a way that one communication requires only a constant number of computations in the simulation. Then if X is less than some centralized lower bound, Y must be greater than that bound. Thus the basic idea is to decrease Y at the expense of increasing, X , i.e. make the nodes "smarter". In some sense, candidate bit vectors (III.5) do just that, because a node may use $O(n)$ computations in interpreting the meaning of one bit in a bit vector.

It would also be nice to have better analyses of existing algorithms. In particular, can the effectiveness of the pruning heuristic of

algorithm SP3 be more precisely characterized, and can one find average case bounds similar to those of IV.2 for symmetric connected graphs?

2. Simulation

If tighter analytical bounds are not obtainable, simulations may provide some insight into the average behavior of existing algorithms and into the construction of worst case examples for the pruning heuristic.

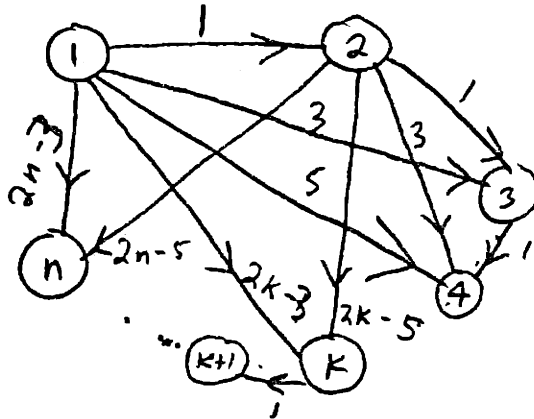
3. Models of Distributed Algorithms and Lower Bounds

Since a lower bound is only valid within the context of some model, we first need to develop reasonable models of the computation - communication processes of distributed algorithms before searching for such bounds. This appears to be a difficult task. However, we suspect that an $O(Ln)$ "somethings" worst case lower bound does exist, where the "somethings" (perhaps bits) is to be determined by the model.

APPENDIX A

Worst Case Examples of Algorithm SP1 (III.4)

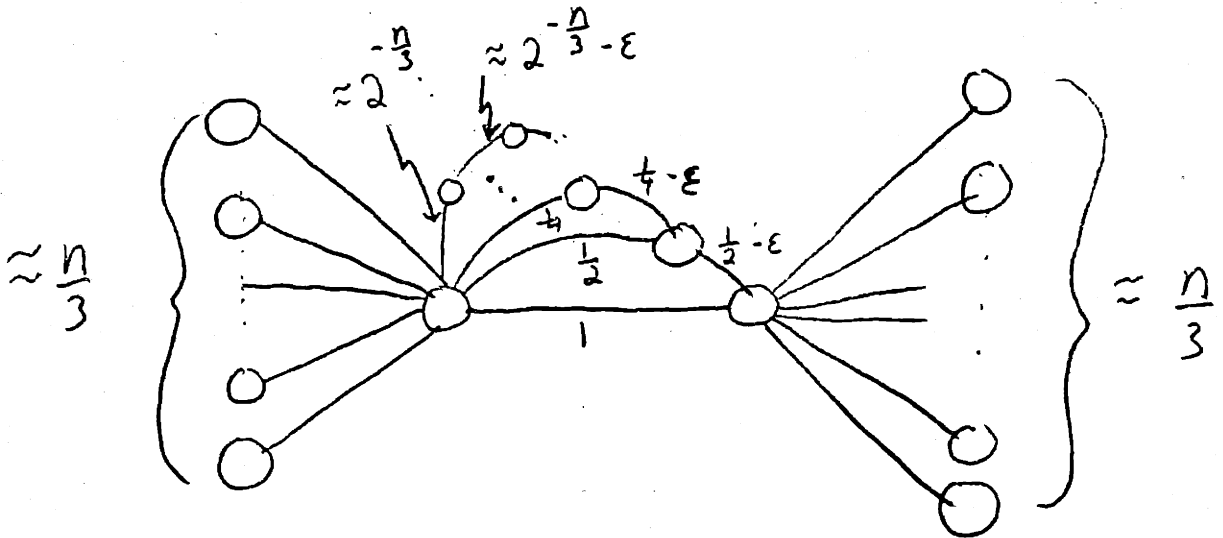
I. This shows $CC(SP1)$ is not $O(n^3)$



and fill in other arcs analogously, i.e. $l(i,i+1)=1$, $l(i,i+2)=3$
 $l(i,i+3)=5$, etc.

By symmetry, the total communication cost is just n times the number of distances received by node 1. A little work will show that node 1 will hear i times about node k from node $k-i$, for $i < k$. Thus node 1 hears $O(k^2)$ times about node k and so receives $O(n^3)$ distances. Hence the total cost is $O(n^4)$.

II. This shows $CC(SP1)$ is not $O(L^2)$.



Each of the $\frac{n}{3}$ nodes on the left will hear approximately $\frac{n}{3}$ times about each of the $\frac{n}{3}$ nodes on the right. This yields $O(n^3)$. However $|L| < 2n$ and so $O(L^2)$ is $O(n^2)$. Thus the algorithm cannot be $O(L^2)$.

BIBLIOGRAPHY

1. D. Cantor and M. Gerla, "Optimal Routing in a Packet Switched Computer Network," IEEE Trans. on Comput., Oct. 1974.
2. R.G. Gallager, "A Minimum Delay Routing Algorithm Using Distributed Computation," IEEE Trans. on Comm., Jan. 1977.
3. A. Aho, J. Hopcroft, J. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, June 1976.
4. E. Lawler, Combinatorial Optimization: Networks and Matroids, Holt, Rinehart, Winston, 1976.
5. P.M. Spira, "A New Algorithm for Finding All Shortest Paths in a Graph of Positive Arcs in Average Time $O(n^2 \log^2 n)$ ", SIAM J. on Computing, March 1973.
6. P. Spira and A. Pan, "On Finding and Updating Shortest Paths and Spanning Trees," SIAM J. on Computing, Sept. 1975.
7. L.R. Kerr, "The Effect of Algebraic Structure on the Computational Complexity of Matrix Multiplications," Ph.D. Thesis, Cornell University, Ithaca, N.Y., 1970.
8. M.L. Fredman, "New Bounds on the Complexity of the Shortest Path Problem," SIAM J. Computing, March 1976.
9. D.B. Johnson, "Algorithms for Shortest Paths," Ph.D Thesis, Cornell University, Ithaca, N.Y., 1973.
10. The basic idea of this algorithm is due to R.G. Gallager.
11. A. Meyer, P. Bloniarz, M. Fischer, "A Note on the Average Time to Compute Transitive Closure", Unpublished Memorandum, MIT Laboratory for Computer Science, July 1976