Research Report CCS 291

A COMPUTATIONAL ANALYSIS OF ALTERNATIVE ALGORITHMS AND LABELING TECHNIQUES FOR FINDING SHORTEST PATH TREES

by

Robert Dial*
Fred Glover**
David Karney***
Darwin Klingman****

April 1977

*Director of Planning Methodology and Technical Support Division, UMTA/DOT, Washington, D.C.

**Professor of Management Science, University of Colorado, Boulder, CO 80302

***Director of Computer Research, Analysis, Research and Computation, Inc., P.O. Box 4067, Austin, TX 78765

****Professor of Operations Research and Computer Sciences, University of Texas and Director of Computer Science Research, Center for Cybernetic Studies, BEB 608, Austin, TX 78712

This research was partly supported by DOT contract DOT-UT-60054 with Analysis, Research and Computation, Inc., P.O. Box 4067, Austin, TX 78765, and by ONR Contracts N00014-75-C-0569 and N00014-75-C-0616 with the Center for Cybernetic Studies, The University of Texas. Reproduction in whole or in part is permitted for any purpose of the United States Government.

CENTER FOR CYBERNETIC STUDIES

A. Charnes, Director
Business-Economics Building, 203E
The University of Texas
Austin, TX 78712
(512) 471-1821

ABSTRACT

This paper examines different algorithms for calculating the shortest path from one node to all other nodes in a network. More specifically, we seek to advance the state-of-the-art of computer implementation technology for such algorithms and the problems they solve by examining the effect of innovative computer science list structures and labeling techniques on algorithmic performance.

The study shows that the procedures examined indeed exert a powerful influence on solution efficiency, with the identity of the best dependent upon the topology of the network and the range of the arc distance coefficients. The study further discloses that the shortest path algorithm previously documented as the most efficient is dominated for all problem structures by the new methods, which are sometimes an order of magnitude faster.

1.0 INTRODUCTION

Shortest and/or longest path analysis is a major analytical component of numerous quantitative transportation and communication models [4,9,13,15,16,20, 23]. These mathematical models seek to improve efficiency and service by increasing capacity, reducing travel time, minimizing congestion, reducing the cost of transportation service, improving vehicle routing, or reducing energy utilization. Such models usually utilize a network to represent the transportation system (which may consist of road segments, railroad tracks, and other common carrier transportation routes) where one desires to find a numerical value of the *minimum* time, cost, distance, energy usage, etc., or *maximum* capacity between several pairs of points in the network. The former problems are often called *shortest path* problems while the latter are called *longest path* problems.

Finding these values in many applications often requires finding the shortest or longest path from one point (called a *root node*) to all other points (nodes) in the network, where nodes can be road intersections, railroad junction points, airplane terminals, and so forth. Further, such information is often successively required for several different root nodes and for a large number of different criterion functions (time, distance, cost, etc.). Additionally, applications often involve iterative determination of the shortest or longest paths for several different values of each criterion function's coefficients during sensitivity analysis. For many applications the networks are very large, containing several thousand nodes and arcs (segments or links).

The longest path problem is often applied to schedule major projects such as: phased network capacity improvement programs; maintenance, overhaul, and leasing of large-scale transportation equipment; resource leveling; research and development programs; and the market introduction of a new production service. The longest path problem is the central component of critical path scheduling, often designated by a variety of acronyms such as CPS, CPM, and PERT. Regardless of the name used, it is very important to realize that the longest path problem is mathematically identical to a shortest path problem. Thus, the algorithms in this paper apply to such problems and henceforth we will use the term shortest path problem to refer to both problems.

The above discussion illustrates the remarkable pervasiveness and significance of shortest path problems and the importance of algorithms to handle these problems efficiently. Because of this, a number of algorithms have been developed for finding the shortest paths from one node to all other nodes in large directed networks. Dreyfus [8] has written an excellent paper classifying the types of algorithms and giving theoretical computational bounds for each class.

While the literature contains many shortest path algorithms, it is important to observe that there are only a handful of *general* methods for solving shortest path problems. Each general algorithm has within it *subalgorithms*. That is, there are special subproblems or sets of operations that must be handled in order to execute the general algorithm; e.g., finding the minimum of a set, breaking a loop, reconnecting subtrees, carrying out computations over the nodes and arcs of subtrees, etc. The literature basically contains descriptions of a large number of different ways to handle these subproblems; unfortunately, many of these alternatives are referenced in the literature as different algorithms rather than as variants of the small class of general algorithms.

Historically these "algorithms" were developed and published because researchers devised ingenious ways of handling one or more of the subproblems in a mathematically efficient manner; i.e., the developer was able to show that his algorithm would require in the *worst case* fewer addition and/or comparison operations than another algorithm.

The use of digital computers has shown, however, that algorithms which have excellent worst case bounds are not necessarily the most efficient (in terms of computer time) for solving real-world problems. This is partially due to the fact that real-world problems have unique features (e.g., only a fraction of the total number of possible arcs, special network or grid structures, small distance coefficient values, etc.) which are often not reflected in the worst case bounds. More importantly, many of the "good" (polynomially bounded) algorithms assume that certain information is available or updated after each iteration at no computational expense; however, when using a digital computer to execute the algorithm, the maintenance of such information actually requires non-trivial computer storage, retrieval, and comparison operations. Therefore, mathematically efficient algorithms do not necessarily result in efficient computer solution procedures.

This has, consequently, spawned an important interface between mathematics and computer science, called *computer implementation technology* [14]. Computer implementation technology is an essential and often neglected component of the study of classes of algorithms. It is in fact a major practical tool for dealing with the ubiquitous issue of *computational complexity*, since no analysis of computational complexity can be truly meaningful without reference to the technology by which solution systems are implemented.

Computer implementation technology involves the design of special procedures to carry out subalgorithms of a general method efficiently on a digital computer. Typically, this requires *research* to determine: (1) the kinds of information to keep on hand for executing certain operations most effectively, (2) the kinds of data structures in which to express this information, and (3) the actual methods for processing these data structures to make the desired information available when it is needed. Effective use of such research further involves *design by feedback*, iteratively amending and integrating component procedures by reference to computational analysis and performance.

The evolution of efficient methods for network flow and shortest path problems [1,2,3,6,9,10,11,12,13,14,22,24] uniquely demonstrates the power of computer implementation technology, properly applied, to yield gains that were not previously suspected. For example, 2000 node 7000 arc minimum cost network flow problems that required several minutes to solve in 1968 can now be solved in only 20 seconds, using the same general algorithm, computer, and compiler [3]. Similarly, Gilsinn and Witzgall [9] found that improved implementation technology caused solution times for shortest path problems to drop from one minute to slightly more than one second, using the same general shortest path algorithm, computer, and compiler.

In the past, due to the lack of attention to developing systematized principles and concepts, it was common for people to attribute variations in a general algorithm to the skill (*art*) of the computer programmer. Recently, an awareness has developed within many of the science disciplines, and particularly within operations research, that the design of efficient computer programs for solving mathematical problems is subject to the enunciation of key methodological and analytical principles, and therefore is primarily a *science rather than an art.*

The excellent study by Gilsinn and Witzgall [9] pioneered this awareness in application to shortest path algorithms and provides a unified structure for describing such algorithms. The purpose of this paper is to extend this work, to evaluate procedures not investigated in the Gilsinn and Witzgall study, and to further demonstrate the importance of computer implementation technology by the exposition of new procedures that are superior to those previously documented.

This paper specifically focuses on characterizing and comparing different algorithms for calculating the shortest paths from one node to all other nodes in a directed network. This study shows that alternative list structures and labeling methods indeed exert a remarkably powerful influence on solution efficiency, and that the identity of the best of these methods depends upon the topology of the network and the range of the arc length coefficients. An additional significant result of the study is the disclosure that the new implementation methods are sometimes an order of magnitude faster than the previously fastest method.

2.0 NETWORK TERMINOLOGY AND STORAGE

This section contains formal definitions of the terms used to describe shortest path problems and algorithms. In order to unify the literature in shortest path methods and their implementation, we will largely use the terminology of the Gilsinn and Witzgall study, departing only to make distinctions and refinements not anticipated in previous work.

A *directed network* or simply a network $G(N,A)$ consists of a finite set N of *nodes* and a finite set A of *arcs*, where each arc $a \in A$ may be denoted as an ordered pair (u,v), referring to the fact that the arc is conceived as beginning at a node $u \in N$ and terminating at a different node $v \in N$.

A *directed path* or *path* is a finite sequence of arcs $P = \{a_1, a_2, \ldots a_n\}$ such that for each $i = 2, \ldots n$, arc a_i begins at the end of arc a_{i-1}. P is called a path from node u to node v if a_1 starts at node u and arc a_n terminates at node v. If a network contains a path from node u to node v, then v is called *accessible* from u. A path P from u to v is called a *circuit* if $u = v$. A path for which $a_i \neq a_j$ for $i \neq j$ is called *arc-simple.*

Let $\ell(a)$ or $\ell(u,v)$ denote a *nonnegative* length associated with arc $a = (u,v)$ of a network. Then we define the length of path P to be $d(P) = \sum_{i=1}^{n} \ell(a_i)$. Path P from one particular node to another node is called a *shortest path* if d(P) is the minimum length of any path between these nodes.

A network may be represented in a computer in several ways and the manner in which it is represented directly affects the performance of algorithms applied to the network. Three basic ways of representing a network with $|N|$ nodes and $|A|$ arcs are:

1. Use an $|N| \times |N|$ matrix $C = (C_{ij})$, where element $C_{ij} = \ell(i,j)$. This value is treated as "infinity" (in practice, some very large number) if the arc does not exist. This representation has two shortcomings. First, it assumes that the network does not contain multiple arcs for the same node pair. Second, if the network is *sparse* (that is, most $C_{ij} = \infty$ or equivalently $|A| / |N|^2$ is small) then computer storage is not effectively utilized.

Matrix representation is normally used with matrix methods for solving shortest path problems. Such methods [15] are normally used to find the shortest path between all pairs of nodes simultaneously. Because of their large storage requirements, their application is restricted to relatively small networks and will not be considered in this paper.

2. Another way of representing a network is to list all of the arcs in the network by keeping for each arc its beginning node, ending node, and length. This requires $3|A|$ computer memory locations, which is generally superior to the matrix representation, but is not well suited to the implementation of certain network processing operations. The next representation to be described has more attractive memory requirements and is also more amenable to processing.

3. The most popular way of storing a network is to use a linked list structure. In this method, all of the arcs that begin at the same node are stored together and eacn is represented by recording only its ending node and length. A pointer is then kept for each node (heading) which indicates the block of computer memory locations for the arcs beginning at this node. The set of arcs emanating from node u is called the *forward star* of node u and denoted by FS(u); i.e., FS(u) = $\{(u,j) \in A\}$. If the nodes are numbered sequentially from 1 to $|N|$ and the arcs are stored consecutively in memory such that the arcs in the forward star of node i appear immediately after the arcs in the forward star of node i-1, then this method, called the *forward star form*, requires only $|N| + 2\,|A|$ units of memory.

Throughout this paper we will assume that the network is represented in forward star form. In some cases we will further assume that the arcs of the forward star of each node are ordered by ascending length; this will be called a *sorted* forward star form. Figure 1 illustrates the storage of a network in a sorted forward star form. The number in the square attached to an arc of the network diagram is the arc length.

The forward star forms are commonly used with special algorithms called *labeling methods* for implementing shortest path and network flow solution procedures. In general, labeling methods are the most widely used methods for industrial and governmental applications, and constitute the primary focus of this paper because such methods are especially effective in application to large sparse networks. Next we define some terms commonly used in describing labeling algorithms.

3.0 TREE TERMINOLOGY AND LABELING TECHNIQUES

In the context of directed networks, a *rooted tree*, or simply a *tree*, is a network $T(N_T,A_T)$ together with a node r (called the *root node*), such that each node of N_T, except r, is accessible from r by a unique arc-simple path in T.

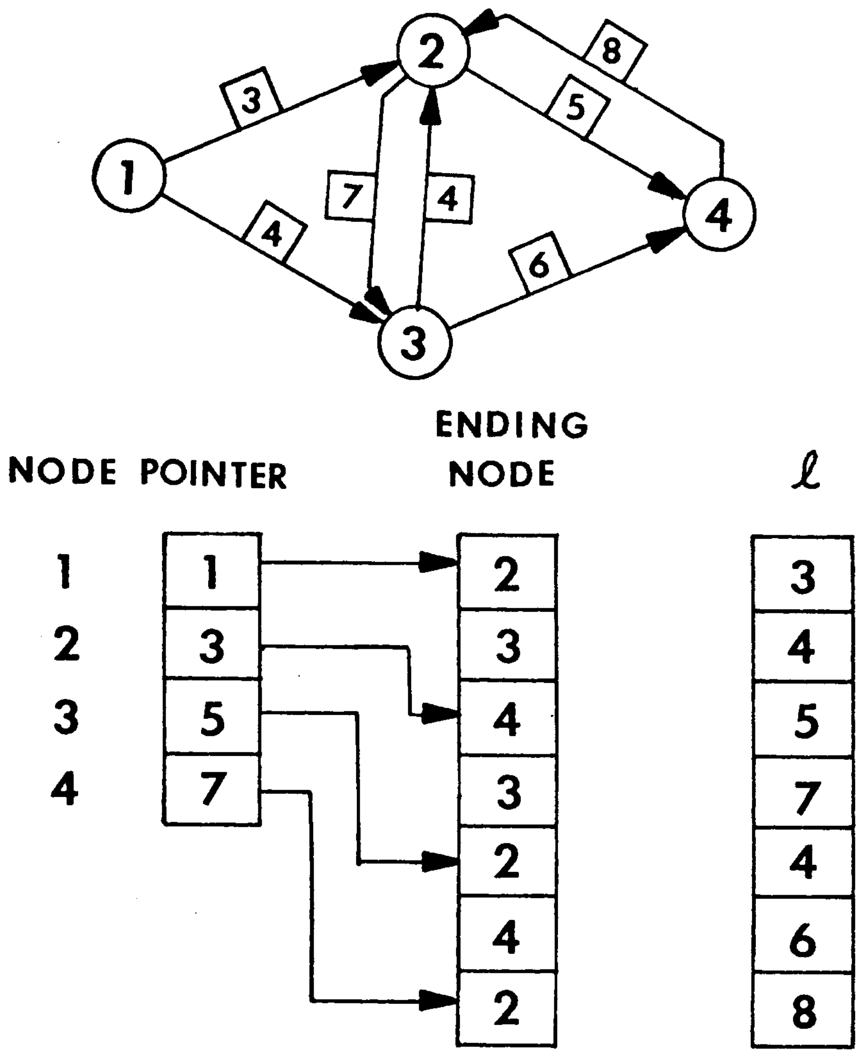


FIG. 1 - SORTED FORWARD STAR FORM

A rooted tree T is called a *minimum tree* or *shortest path tree* of a larger network G(N,A) if T contains all nodes of G accessible from r, and if for each node v in N_T, the unique path P from r to v is a shortest path from r to v in the network G.

Labeling algorithms typically start with a tree, T, consisting only of the root node r and seek to enlarge and modify T until it becomes a shortest path tree of a larger network G. Thus, an important computer implementation component of such algorithms involves properly handling T and storing G.

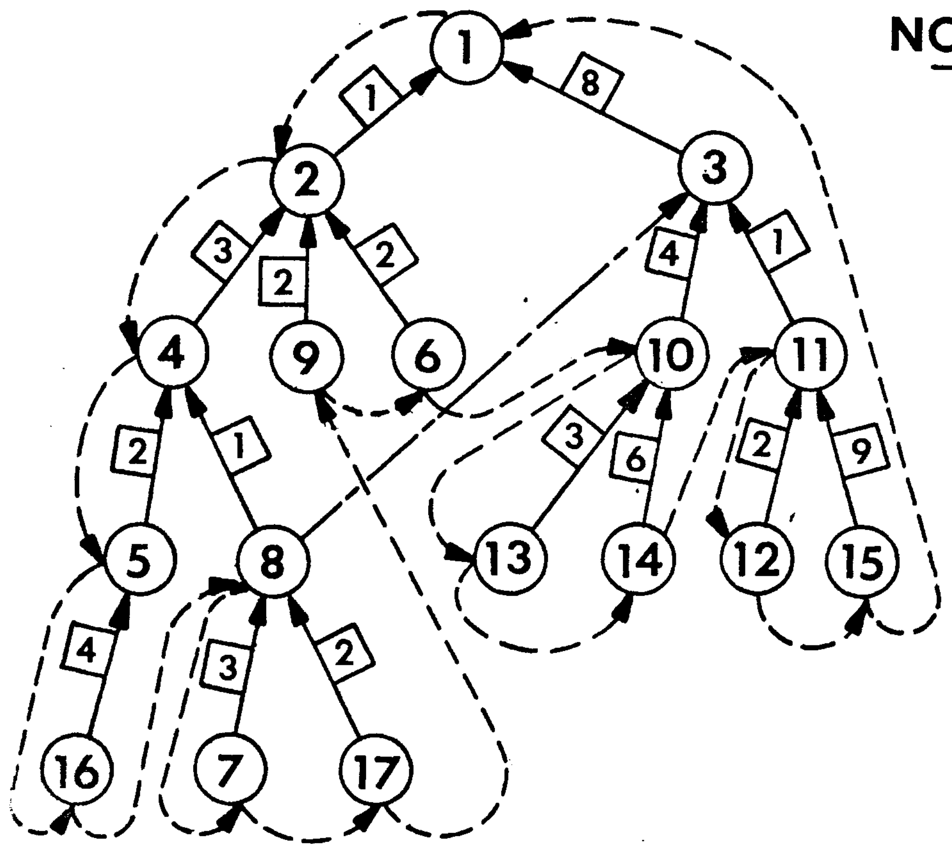
A common way of representing a tree in a computer is to think of the root node as the highest node in the tree and all the other nodes hanging below the root. The tree is then represented by keeping a pointer list which contains for each node $w \neq r$ in the tree, the starting node v of the single arc in the tree terminating at w. This upward pointer is called the *predecessor* of node w and will be denoted by p(w). Further, node w is called an *immediate successor* of node v. For convenience, we will assume that the predecessor of the root, p(r), is zero. Figure 2 illustrates a tree rooted at node 1, the predecessors of the nodes, and other functions to be described subsequently. The predecessor of a node is identified in the p array. For example, the precedessor of node 16 is node 5.

Most labeling algorithms keep another list indexed by the node numbers and associated with the tree T. This list contains for each node v a label d(v), whose value is the length of the unique path from r to v in T. (In some implementations, d(v) is not always the correct length but an overestimate that gradually converges to the correct length.) Henceforth d(v) will be called the *node potential* of node v. Nodes not in T may or may not be labeled with a node potential value; usually they are given the label ∞, indicating that they are not yet reached by the tree. The root r has a node potential of zero.

In Figure 2 the number in the square on each arc indicates the length of the arc. The entries in the d array identify the length of the unique path from the root to each node. Figure 2 illustrates additional tree information expressed as node functions, which will be used in the computer implementation procedures to be discussed subsequently.

The first of these functions, the *thread* function [1,12], is denoted by t(x). This function is a downward pointer through the tree. As illustrated in

Predecessor	**p(x)**
Node potential	**d(x)**
Thread	**t(x)**
Reverse thread	**rt(x)**
Depth	**dh(x)**
Cardinality	**c(x)**
Last node in subtree	**f(x)**



NODE	p	d	t	rt	dh	c	f
1	0	0	2	15	0	17	15
2	1	1	4	1	1	9	6
3	1	8	10	6	1	7	15
4	2	4	5	2	2	6	17
5	4	6	16	4	3	2	16
6	2	3	3	9	2	1	6
7	8	8	17	8	4	1	7
8	4	5	7	16	3	3	17
9	2	3	6	17	2	1	9
10	3	12	13	3	2	3	14
11	3	9	12	14	2	3	15
12	11	11	15	11	3	1	12
13	10	15	14	10	3	1	13
14	10	18	11	13	3	1	14
15	11	18	1	12	3	1	15
16	5	10	8	5	4	1	16
17	8	7	9	7	4	1	17

FIG. 2 - TREE LABELING TECHNIQUES

Figure 2 by the dotted line, function t may be thought of as a connecting link (thread) which passes through each node exactly once in a top to bottom, left to right sequence, starting from the root node. For example, in Figure 2, t(1) = 2, t(2) = 4, t(4) = 5, t(5) = 16, t(16) = 8, etc.

Letting n denote the number of nodes in T (N_T, A_T), the function t satisfies the following inductive characteristics:

a) The set $\{r, t(r), t^2(r), \ldots, t^{n-1}(r)\}$ is precisely the set of nodes of the rooted tree, where by convention $t^2(r) = t(t(r))$, $t^3 = t(t^2(r))$, etc. The nodes r, t(r), ..., $t^{k-1}(r)$ will be called the *antecedents* of node $t^k(r)$.

b) For each node i other than node $t^{n-1}(r)$, t(i) is one of the nodes such that p(t(i)) = i, if such nodes exist. Otherwise, let x denote the first node in the predecessor path of i to the root which has an immediate successor y and y is not an antecedent of node i. In this case, t(i) = y.

c) $t^n(r) = r$; that is, the "last node" of the tree threads back to the root node.

The *reverse thread* function, rt(x), is simply a pointer which points in the reverse order of the thread. That is, if t(x) = y, then rt(y) = x. Figure 2 also lists the reverse thread function values.

The *depth* function, dh(x), indicates the number of nodes in the predecessor path of node x to the root, not counting the root node itself. If one conceives of the nodes in the tree as arranged in levels where the root is at level zero and all nodes "one node away from" the root are at level one, etc., then the depth function simply indicates the level of a node in the tree. (See Figure 2.)

The *cardinality* function, c(x), specifies the number of nodes contained in the subtree associated with node x in the tree. By the nodes in the subtree associated with node x, we mean the set of all nodes $w \in N_T$ such that the predecessor path from w to the root contains x. (See Figure 2.)

The *last node in a subtree* function, f(x), specifies that last node in the subtree of x that is encountered when traversing the nodes of this subtree in "thread order." More precisely, $f(x) = y$ where y is the unique node in the subtree of x such that t(y) is not also a node in the subtree of x. (See Figure 2.)

Note that both the domain and the range of each of the above discrete functions consist of the set of nodes and thus are independent of the number of arcs. Since $|N|$ is the maximum number of nodes that could be in T, a one dimensional array of size $|N|$, called a *node length array*, is allocated to each function during computer implementation. The procedures for updating the values of the functions when the tree is reconfigured will be detailed subsequently.

4.0 SHORTEST PATH PROBLEM AND LABELING METHODS

By means of the foregoing terminology, the problem of finding the shortest paths from a given node r to all other nodes in network G(N,A) may be stated as that of finding a minimum tree $T(N_T,A_T)$ of G rooted at node r.

Labeling methods for computing such a minimum tree have been divided into two general classes, label-setting and label-correcting methods. Both methods typically start with a tree $T(N_T,A_T)$ such that $N_T = \{r\}$ and $A_T = \emptyset$. A label-setting method then augments N_T and A_T respectively, by one node $v \in N$ and one arc $(u,v) \in A$ at each iteration in such a manner that $u \in N_T$, $v \in N_T$, and the unique path from r to v in T is a shortest path. A label-setting method terminates when all arcs in A which have their starting endpoints in N_T also have their ending endpoints in N_T.

A label-correcting method, on the other hand, always exchanges, augments, or updates arcs in A_T in a manner that replaces or shortens the unique path from r to v in T, but does not guarantee that the new path is a shortest path (until termination occurs). Using the notation defined in the previous section, we now give a precise description of each of these *general* methods.

<u>General Label-Setting Method</u>

1. Initialize a tree $T(N_T,A_T)$ such that $N_T = \{r\}$ and $A_T = \emptyset$. Further, set $p(t): = 0$, $t \in N$; $d(t): = \infty$, $t \in N - \{r\}$; and $d(r): = 0$. (The notation $a: = b$ sets a equal to b.)
2. Let $S = \{(u,v): u \in N_T; v \in N - N_T, (u,v) \in A\}$. If $S = \emptyset$, go to step 4. Otherwise proceed.
3. Let $d(u) + \ell(u,v) = \min\limits_{(p,q)\varepsilon S} (d(p) + \ell(p,q))$. Redefine

$$N_T: = N_T \cup \{v\}$$
$$A_T: = A_T \cup \{(u,v)\}$$
$$p(v): = u$$
$$d(v): = d(u) + \ell(u,v)$$

   and repeat step 2.
4. Stop. $T(N_T,A_T)$ is a minimum tree and for each node $v \in N$, $d(v)$ is the length of a shortest path from r to $v \neq r$.

It is worth noting that a label-setting method only works for nonnegative arc lengths. A label-correcting method, however, works for negative arc lengths as long as there are no circuits of negative length in the network G(N,A).

<u>General Label-Correcting Method</u>

1. Initialize a tree $T(N_T,A_T)$ such that $N_T = \{r\}$ and $A_T = \emptyset$. Further, set $p(t): = 0$, $t \in N$; $d(r): = 0$; and $d(t): = \infty$, $t \in N - \{r\}$.

2. Go to step 4 if there does not exist an arc $(u,v) \varepsilon A$ such that $d(u) + \ell(u,v) < d(v)$. Otherwise, for such an arc, redefine

$$N_T: = N_T \cup \{v\}$$
$$A_T: = A_T - \{(s,v) \varepsilon A_T\} \cup \{(u,v)\}$$
$$p(v): = u$$
$$d(v): = d(u) + \ell(u,v)$$

3. Repeat step 2.

4. Stop. $T(N_T,A_T)$ is a minimum tree and for each node $v \varepsilon N$, $d(v)$ is the length of a shortest path from r to $v \neq r$. Further, if a shortest path from r to v exists (i.e., if $p(v) \neq 0$), then it may be constructed by successively examining the predecessors of v until the root node r is encountered.

5.0 EXPERIMENTAL DESIGN

Alternative implementation methods are evaluated in this study by solving a diverse set of randomly generated shortest path problems using the same computer (a CDC 6600), the same compiler (a FORTRAN RUN compiler), and executing the codes during time periods when the demand for computer use was comparable. Further, all of the codes were implemented by the same systems analyst and no attempt was made to exploit any of the unique hardware characteristics of the CDC 6600.

Even with these safeguards, minor differences between the solution times of any two codes for a single test run of each must be regarded of questionable significance. For this reason, each test problem was solved 100 times (i.e., for 100 different roots) and the average solution time reported. Each code makes use of a real-time clock routine supplied by CDC. This routine can be employed using a FORTRAN subroutine call and is generally accurate to two decimal places. The reported times include only the elapsed time after input of the shortest path problem and prior to output of its solution. This includes the time required to initialize the function arrays.

The problem set consists of shortest path problems from two distinct topological groups. One set consists of rectangular *grid* networks. A p x q rectangular grid network may be envisioned as having its nodes arranged in p parallel rows each containing q nodes. Each node connects by arcs only to the four nodes (if present) to its right and left and above and below. Thus a p x q grid network has pq nodes and 4 pq - 2p - 2q arcs. It is important to note, however, that the arc lengths are randomly generated. Thus, arc lengths are not necessarily symmetric and the triangle inequality may not hold.

The grid network test problems all have 2500 nodes with rectangularities of 50 x 50, 25 x 100, 10 x 250, and 5 x 500. These problems were generated using a unifrom probability distribution with two unique distance ranges for the arc lengths; the first range of arc lengths lies between 1 and 100 and the second between 1 and 10000. Table 1 describes all of these grid problems and contains solution times on the alternative implementations to be discussed subsequently.

The second topologically distinct set of problems consists of *random* networks. A random network is one in which two nodes are selected randomly to form a new arc to add to the network. The nodes are selected using a uniform probability distribution, subject to the restrictions that the two nodes are not the same and arcs are not allowed to be duplicated. The random network test problems all have 1000 nodes and contain either 5000, 10000, 15000, 20000, 25000, or 30000 arcs. For each of these problem sizes, two problems were generated, one with arc lengths between 1 and 200 and the other with arc lengths between 1 and 10000. Again the arc lengths were randomly selected using a

Table I

SOLUTION TIMES ON GRID NETWORKS
(SECS/TREE, AVERAGED FOR 100 TREES)

Rectangularity	Nodes	Arcs	Arc Length Range	C1	C2	C3	C4	C5	S1	S2	S3	S4
50 X 50	2500	9800	1-100	.66	.25	.60	.48	.41	.37	.40	DNR	.81
25 X 100	2500	9250	1-100	.99	.25	.58	.51	.40	.37	.40	DNR	.75
10 X 250	2500	9480	1-100	2.96	.26	.49	.44	.38	.38	.43	DNR	.65
5 X 500	2500	8990	1-100	3.70	.24	.43	.39	.34	.42	.47	DNR	.57
50 X 50	2500	9800	1-10000	.68	.25	.60	.49	.41	1.70	1.67	.57	.82
25 X 100	2500	9750	1-10000	1.03	.25	.57	.50	.41	2.27	2.24	.57	.76
10 X 250	2500	9480	1-10000	3.00	.26	.50	.43	.37	2.89	2.87	.61	.66
5 X 500	2500	8990	1-10000	4.13	.24	.39	.37	.27	3.11	2.91	.57	.58

DNR--Did not run.

Table II

SOLUTION TIMES IN SECONDS ON A CDC 6600 FOR RANDOM NETWORKS
(SECS/TREE, AVERAGED FOR 100 TREES)

Nodes	Arcs	Arc Length Range	C1	C2	C3	C4	C5	S1	S2	S3	S4
1000	5000	1-200	.15	.13	.42	.28	.20	.21	.23	DNR	.34
1000	10000	1-200	.31	.28	.63	.42	.35	.33	.32	DNR	.50
1000	15000	1-200	.44	.43	.72	.58	.47	.42	.39	DNR	.61
1000	20000	1-200	.59	.59	.90	.70	.61	.52	.47	DNR	.72
1000	25000	1-200	.80	.80	1.17	.88	.77	.62	.55	DNR	.81
1000	30000	1-200	.91	.91	1.31	1.01	.90	.70	.62	DNR	.90
1000	5000	1-10000	.16	.13	.43	.28	.20	.50	.53	.30	.34
1000	10000	1-10000	.32	.29	.61	.43	.35	.51	.45	.40	.47
1000	15000	1-10000	.43	.43	.71	.59	.46	.59	.51	.49	.58
1000	20000	1-10000	.65	.64	.89	.71	.63	.68	.59	.62	.66
1000	25000	1-10000	.85	.85	1.21	.90	.79	.76	.61	.69	.70
1000	30000	1-10000	.97	.96	1.32	1.03	.91	.88	.70	.86	.81

DNR--Did not run.

uniform probability distribution. Table II contains the computational results on the random network problems.

To provide researchers with reproducible benchmarks, the appendix contains FORTRAN listings of the problem generators and the two computer codes found to be the best in this study.

6.0 IMPLEMENTATION TECHNIQUES FOR THE LABEL-CORRECTING METHOD

In this section we discuss a sequence of implementations of the general label-correcting algorithm which successively utilize more and more information (as embodied in the node functions) to determine the effect of this information on the efficiency of the algorithm. The merits of these alternative implementations are then evaluated by solving the test problems.

6.1 Implementations Using Only p and d Functions

Step 2 of the general label-correcting method involves finding any arc a which can be added to (or updated in) the tree with a resultant decrease in the node potnetial of its ending node. One of the fundamental subalgorithms of this general method involves searching for such an arc in an intelligent manner. Several observations have been made in the literature regarding this search. The most rudimentary observation is that if the arcs are sequentially examined, it is not necessary to examine any arc $(u,v) \varepsilon A$ whose beginning node has an infinite node potential since $d(u) + \ell(u,v) < d(v)$ will never be satisfied for nonnegative arc lengths.

This observation extends quite naturally as follows. If each arc $(u,v) \varepsilon FS(u)$ has been examined and found to satisfy the condition $d(u) + \ell(u,v) \geq d(v)$, then it is unnecessary to re-examine these arcs until the node potential of u decreases. This observation is one of the primary motivating factors for storing the network in a forward star form. As will be seen, the order in which forward stars of nodes are examined plays a major role in the efficiency of the algorithm.

Based on the preceding observation, it is convenient to keep a *sequence list* of nodes whose node potentials have decreased since their forward stars were last examined. That is, nodes are added to the sequence list whenever their node potentials are decreased and deleted from the list upon examining their forward stars. By not allowing a node to appear more than once on this list, it is possible to restrict the size of this list to a node length array. One simple way to guarantee that a node is not duplicated on the sequence list is to complement the forward star pointer of the node when it is added to (or deleted from) the list. Using this technique, the sign of a node's forward star pointer is checked before adding the node to the sequence list. If its sign is positive, the node is added to the list; otherwise, it is already on the list.

The sequence list can be managed in a variety of ways. In particular, if the forward stars are examined in the order in which their identifying nodes are placed on the sequence list, the list is said to be managed in a FIFO (First-in, First-out) manner; if the forward star of the latest node added to the list is examined before that of a node placed on the list previously, it is said to be managed in a LIFO (Last-in, First-out) manner. Yet another way to manage the sequence list is to pick the node at the front of the list to examine next as in the FIFO procedure, but to add nodes at either the front or the back of the list; that is, to handle the sequence list as a two-way sequence list adding to either end but always deleting from the front. As will be seen, the way in which the sequence list is managed has major ramifications for the efficiency of the algorithm. We now describe in detail the codes whose solution times are indicated for grid networks and random networks in Tables I and II.

Code C1 employs a FIFO sequence list, and the predecessor and node potential functions. The list is processed by using two pointers, s and e, where s points to the entry whose forward star is to be examined next and e is the position of the last node added.

Code C2 utilizes the predecessor and node potential functions and a two-way sequence list. The two-way sequence list is implemented as suggested by Pape [22]. That is, the sequence list is a node length array, called CL, identified by node numbers, such that

$$CL(x) = \begin{cases} -1 & \text{if node x was previously on the list but is no longer on the list} \\ 0 & \text{if node x has never been on the list} \\ +y & \text{if node x is on the list and y is the next node of the list} \\ +\infty & \text{if node x is on the list and x is the last node on the list} \end{cases}$$

In addition, the start and end pointers, s and e, are kept. (See the listing of code C2 in the appendix.)

The solution times in Table II are very similar for codes C1 and C2. Thus for random networks, the management of the sequence list does not seem to affect solution speed. The results in Table I, on the other hand, show that this is not true for grid networks. Code C2 is dramatically superior in this case.

This surprising difference can be explained as follows. The minimum tree and also most of the intermediate trees are very narrow and deep in grid networks, due to the fact that only one or two tree arcs emanate from each node. This causes the subtree of an arbitrary node v, in general, to exhibit the "narrow and deep" property. Moreover, if the node potential of node v is decreased by an amount δ then the node potentials of all nodes in the subtree of v must ultimately be decreased by δ (unless the subtree later becomes restructured, in which case some node potentials will decrease by an even greater amount). To illustrate, suppose arc (8,3) of Figure 2 is to be added to the tree and $d(3)$ is set to $d(8) + \ell(8,3) = 5 + 1 = 6$ (hence $d(3)$ is decreased by 2). Then the length of the unique path from the root to each node in the subtree of node 3 is reduced. Consequently, the node potentials in this subtree should be decreased.

The FIFO sequence list postpones updating these node potentials since node v is added to the back of the list. In contrast, the two-way sequence list adds v to the front of the list (if it is not already on the list). Thus, loosely speaking, nodes in the subtree of v tend to be updated before other nodes are examined.

This updating sequence helps to eliminate unnecessary node potential corrections that are dominated by the δ correction that should be transmitted through the subtree. That is, an arc (p,q) may satisfy the condition $d(p) + \ell(p,q) < d(q)$ only because $d(q)$ has not been reduced by δ. The occurrence of such unnecessary corrections can have a cumulatively deleterious effect. In particular, each one causes a new node to be added to the sequence list which has an "erroneous" (i.e., dominated) node potential value. Each time such a node is then selected from the list (if it has not in the meantime received a "corrected" node potential value), a correspondingly erroneous value is transmitted to still other nodes. The difficulties of the process are thus perpetuated.

The effects of generating and transmitting erroneous node potentials, just discussed, raise the question of whether a label-correcting method can be implemented by means of more sophisticated list structures and processing techniques with a net gain in computational efficiency. We now consider implementations designed to respond to this question.

6.2 Implementations Using p, d, t, c, and f Functions

The thread function, as observed earlier, provides an efficient way of locating each node in the subtree of any node in N_T. Thus, if the node potential of node v is decreased by δ, the thread function can be used to update all node potentials in node v's subtree. As shown in [1], the last node and cardinality functions can be used efficiently to update t.

We have designed two codes, C3 and C4, to test the major implementation alternatives. Code C3 uses the p, d, t, c, and f functions as follows. The code starts with $N_t = \{r = \text{root}\}$, $A_t = \emptyset$ and initializes $p(v) = 0$, $v \in N$; $t(r) = r$; $t(v) = 0$, $v \in N - \{r\}$; $d(r) = 0$; $d(v) = \infty$, $v \in N - \{r\}$; $c(r) = 1$; $c(v) = 0$, $v \in N - \{r\}$; $f(r) = r$; $f(v) = 0$, $v \in N - \{r\}$. Code C3 also uses a logical node array e to indicate if a node's forward star requires scanning. In particular, for $v \in N$, $e(v) = 1$ if the node potential of v has changed since v was last examined and $e(v) = 0$, otherwise. This array is initialized by setting $e(r) = 1$ and $e(v) = 0$, $v \in N - \{r\}$. Using e, the algorithm then searches for an arc $(u,v) \in A$ such that $\delta = -d(u) + d(v) - \ell(u,v) > 0$, whereupon d(v) is reset to $d(v) := d(v) - \delta$ and the node potentials of all other nodes in the subtree of node v are decremented by δ. The algorithm terminates when $e(v) = 0$, $v \in N$. (Since each element of e has only two states, it is not necessary to use a separate computer array for this function.)

The p, t, c, and f functions are updated by the following set of operations where (u,v) denotes the arc to be added to A_T. (The reader may find it helpful to perform these operations using Figure 2 and letting $(u,v) = (8,3)$.)

Step 1: Identify the node y such that $t(y) = v$. Then set $t(y) := t(f(v))$. (Note that the identification of y may be efficiently done by first letting $y' = p(v)$. Second, if $t(y') = v$ then $y = y'$ and the process stops. Otherwise, let $y' = f(t(y'))$ and repeat the second step.)

Step 2: Identify the first node x (lowest node) common to the predecessor paths for u to r and v to r. Then set $c(i) := c(i) + c(v)$ for each node i in the predecessor path from u to x (excluding node x) and set $c(i) := c(i) - c(v)$ for each node i in the predecessor path from p(v) to x (excluding node x).

Step 3: Let $w = p(t(f(v)))$. If $w = 0$, then set $w = r$. Set $f(i) = y$ (i.e., the node y determined in step 1) for those nodes i on the predecessor path from p(v) to w, excluding w itself if $p(t(f(v))) \neq 0$.

Step 4: Set $p(v) := u$.

Step 5: Set $t(f(v)) := t(u)$.

Step 6: Set $t(u) := v$.

The second code, C4, based on the more sophisticated node functions is a simple modification of C3 in which the e array is replaced with a FIFO sequence list.

6.3 A Primal Simplex Method Interpretation of the Label-Correcting Algorithm

The preceding implementations of the general label-correcting algorithm may be viewed as specialized variants of the *primal simplex* algorithm where the basic variables correspond to the arcs in A_T, augmented by artificial arcs which start at the root r and end at node i for each $i \in N - N_T$ such that $\ell(r,i) = \infty$. The interpretation is especially direct for the codes C3 and C4, which insure that the node potentials always satisfy complementary slackness, i.e., $-d(u) + d(v) = \ell(u,v)$, $(u,v) \in A_T$ and $-d(r) + d(i) = \ell(r,i)$, $i \in N - N_T$. Extending this interpretation, the process of selecting an improving arc (i,j) corresponds to searching for an arc which violates dual feasibility. The process of adding such an arc (t,s) to N_T and deleting an arc (p(s),s) from A_T is equivalent to a simplex basis exchange. (Note that if $p(s) = 0$ then arc (p(s),s) corresponds to an artificial arc and is not a member of A_T.) The update of the node potentials after performing this basis exchange simply maintains complementary slackness.

From this point of view, the replacement of the e array of C3 with the FIFO sequence of C4 corresponds simply to the use of different pivot selection rules. Tables I and II show that this change of pivot selection strategy strictly improves solution time.

The previous codes C1 and C2, on the other hand, correspond to a deferred updating version of the primal simplex algorithm in the sense that a basis exchange is performed each time an arc is added to A_T, but the full set of updated node potentials in a subtree are not immediately determined. In particular, codes C1 and C2 differ from the codes C3 and C4 by requiring that complementary slackness be maintained only "locally" rather than globally. The times in Tables I and II demonstrate that it is not necessarily beneficial to maintain complementary slackness after each iteration. Code C2, while postponing the update of the dual variable (node potential) values, appears to balance the distortion caused by using locally updated dual variable values with the work required to maintain globally updated values.

6.4 Additional Implementations Using Alternative Pivot Strategies

As a result of the interpretations of these codes as variants of the primal simplex method, we undertook to test variations of C3 and C4 that used other types of pivot strategies. First, code C4 was modified by scanning the forward star of a node removed from the FIFO list multiple times. Each time the forward star is scanned, the arc violating dual feasibility by the largest amount is selected for the basis exchange. This pivot criterion was tested because it has been shown in other network flow applications to be more effective than simply pivoting the arcs in a "random" order [5,10,11,24]. The times for this variant of C4 are not shown in Tables I and II because, contrary to the results for other types of network flow problems, the solution times were uniformly 10% to 15% slower than for the "unordered" selection procedure.

Following this, we tested a number of other more sophisticated pivot criteria. Mulvey [2] has shown that an excellent pivot criterion for large transportation and transshipment problems derives from the use of an *arc candidate list*. Mulvey's approach involves two parameters r and s, where r specifies the maximum number of arcs on the list and s specifies the maximum number of pivots to be made before revising the elements on the list. The candidate list is created by sequentially examining the forward star of nodes with an e value of 1 in code C3 and selecting arc (u,v) in each forward star which violates $-d(u) + d(v) \leq \ell(u,v)$ by the largest amount (if one exists) for inclusion on the list (accumulating at most r such arcs). Each time the list is revised, the search for arcs is initiated at the node following the node where the search was stopped when building the previous list. If r eligible arcs cannot be found, the size of r is reduced to the number actually encountered.

The candidate list approach was incorporated into code C3 and tested for several different list sizes. The outcome, again surprisingly, yielded solution times inferior to those of code C4.

We then designed another variant of the candidate list approach, which made use of the sequence list of code C4. In particular, the first r nodes were taken from the sequence list to form a *node candidate list*. Several different strategies were tested for picking nodes off this candidate list. First, the nodes were selected in increasing order of their cardinality function value, and the forward star of the selected node was scanned.

The logic behind this pivot selection strategy is that nodes with larger cardinality function values are likely to be closest to the root node, indicating an increased attractiveness for being examined first. Several different list sizes were tested, but none reduced solution times. Similarly, tests were conducted for the strategy of selecting the nodes in increasing order of their node potential values. This also failed to reduce solution times.

These results strongly suggested that more sophisticated versions of special purpose simplex codes using globally updated node potentials are not competitive with the simpler label-correcting code C2. Before submitting completely to this conclusion, however, we decided to test a different implementation of the simplex method where the c and f functions are replaced by the reverse thread, rt, and depth, dh, functions. The primary motivation underlying this implementation is that these functions can be updated more easily than the previous functions in the setting of shortest path problems. (This is not true, however, in the setting of other network flow problems.)

6.5 Primal Simplex Implementations Using p, d, t, rt, and dh Functions

The implementations based on the reverse thread and depth functions, like the preceding implementations, use the thread function to find and update all node potentials in a subtree. The rt and dh functions are used to update t, replacing the c and f functions in this task.

First, a code C5 was implemented using p, d, t, rt, and dh functions initialized such that $p(v) = 0$, $v \in N$; $t(r) = rt(r) = r$; $t(v) = rt(v) = 0$, $v \in N - \{r\}$; $d(r) = 0$; $d(v) = \infty$, $v \in N - \{r\}$; $dh(v) = 0$, $v \in N$. Additionally, code C5 uses a FIFO sequence list to locate an arc $(u,v) \in A$ such that $\delta = -d(u) + d(v) - \ell(u,v) > 0$, whereupon all nodes in the subtree of node v are decremented by δ and "added to" the sequence list. Simultaneously, the depth of each node in this subtree is incremented by $\Psi = d(u) - d(v) + 1$. The algorithm terminates when the sequence list is empty.

The p, t, and rt functions are updated by the following steps where (u,v) denotes the arc to be added to A_T. (The reader may find it helpful to perform these steps using Figure 2 and letting (u,v) = (8,3).)

Step 1: Identify the first $k > 1$ such that $dh(t^k(v)) \leq dh(v)$. (Note that the identification of k should be done simultaneously with the updating of d and dh since identifying k requires tracing out the nodes in v's subtree. In fact, $t^{k-1}(v)$ is the last node in the subtree of v.) Set $t(t^{k-1}(v)) := t(u)$, $rt(t(u)) := t^{k-1}(v)$, $t(rt(v)) := t^k(v))$, and $rt(t^k(v)) := rt(v)$.

Step 2: Set $t(u) := v$, $rt(v) := u$, and $p(v) := u$.

The solution times in Tables I and II indicate that the special purpose simplex code C5 is 25% to 30% faster than the equivalent method using the p, d, t, c, and f functions, code C4. The results also clearly show the importance of matching data structures with algorithmic steps. Further, the results indicate that the code C5 is usually the first or second fastest label-correcting code. The code most often superior to C5 is code C2. As with code C4, we modified code C5 to perform a number of candidate list pivot strategies. None of these variants improved solution times, however.

These results raise the question as to why the "quasi-simplex" code C2, which defers the complete updating of node potentials is generally superior to the full simplex codes C3, C4, and C5 (and to their dominated variants whose times are not reported in the tables). Analysis of the computational data for these codes discloses that the number of pivots (label-correcting iterations) made by code C2 typically is on the order of 1 1/2 to 2 times the number of nodes in the problem. Observe that any label-correcting method which starts with a tree consisting of only the root node, must make at least as many pivots as the number of problem nodes if each node in G(N,A) is accessible from the root. Thus, the margin for improvement in the number of pivots made by C2 is small.

In fact, the simplex codes C3, C4, and C5 do achieve some of this theoretically available improvement. In particular, the average number of pivots made by these codes is approximately 1 3/8 to 1 1/2 times the number of nodes. The augmentations of codes by more sophisticated pivot rules still require pivots ranging from 1 1/4 to 1 3/8 times the number of nodes. However, these small gains in the number of pivots do not represent corresponding gains in solution time, and in fact lead to net losses. It appears that the additional overhead involved in maintaining and updating the extra functions (plus possibly maintaining a pivot candidate list), simply overshadows the gain achieved in reducing the number of pivots for *sparse shortest path networks*. However, as density increases in the shortest path networks, the pivot reduction achieved by the full simplex codes over the quasi-simplex code C2 becomes effective. In particular, the results in Table II indicate that the simplex code C5 becomes comparable to C2 at 20000 arcs and is the fastest label-correcting code for the 1000 node random networks with 25000 and 30000 arcs. Thus, it appears that as the networks become sufficiently dense, it is worthwhile maintaining complementary slackness.

7.0 IMPLEMENTATION TECHNIQUES FOR THE LABEL-SETTING METHOD

In this section we discuss several implementations of the general label-setting method. The primary difference between these implementations is the way in which the minimum in step 3 of the algorithm description is found. As for alternative implementations of the label-correcting method, these implementations are evaluated by solving the same test problems using the same computer and compiler.

A naïve implementation of the general label-setting method would be to find the set S of step 2 by examining all arcs in A and then calculating and discarding node potentials to find the minimum of step 3. This involves examining all arcs during every execution of step 2, as well as performing many unnecessary node potential calculations in step 3. The implementations described in this section make use of temporarily retained node potentials in such a way that each arc in A is examined *at most once*, thereby avoiding extensive recalculation.

As a basis for understanding these implementations, it is useful to observe that steps 2 and 3 of the label-setting method simply find an arc from a tree node to a non-tree node which yields the minimum distance extension. Figure 3 illustrates one way of viewing these steps at some iteration where the tree $T(N_T,A_T)$ consists of the *solid line* arcs and their associated nodes. The *dashed line* arcs and their ending nodes N_E indicate possible tree extensions. (Note that $N-N_T$ may not be equal to N_E.)

By reference to this diagram, it may be seen that steps 2 and 3 can be performed by keeping a temporary node potential and predecessor for each node v in N_E such that $d(v) = \underset{u \in N_T}{\text{minimum}} (d(u) + \ell(u,v))$ and the predecessor of v is set to a node u which yields the minimum node potential for v. Thus, if $p(v) = u$ then $-d(u) + d(v) = \ell(u,v)$. Step 3 then adds a node v in N_E with the smallest temporary node potential to N_T and correspondingly adds its arc $(p(v),v)$ to A_T. After performing this step, node v's potential will never change (i.e., it is assigned a *permanent node potential* at this time) and arc $(p(v),v)$ is permanently assigned to the tree. The name label-setting stems from this property of the algorithm.

In the following subsections we discuss four alternative implementations for carrying out steps 2 and 3 in this manner. These implementations differ in the way they handle the following fundamental operations: (1) the computation and updating of temporary node potentials, (2) the determination of the minimum temporary node potential, and (3) the assignment of one or more temporary node potentials to a node in N_E.

7.1 Interpretation of the Label-Setting Method as a Primal Simplex Method

Before discussing these implementations, it is interesting to observe that the label-setting method may be viewed as a special purpose primal simplex method where the basic variables correspond to the arcs permanently assigned to A_T, augmented by artificial arcs which start at the root r and end at node i for each $i \in N - N_T$ such that $\ell(r,i) = \infty$. The node potentials clearly satisfy complementary slackness at each iteration; i.e., $-d(u) + d(v) = \ell(u,v)$, $(u,v) \in A_T$ and $-d(r) + d(i) = \ell(r,i)$, $i \in N - N_T$. Further, the process of selecting an improving arc (i,j) to enter the basis corresponds to searching (in some fashion) for an arc which violates dual feasibility (i.e., $-d(i) + d(j) \not\leq \ell(i,j)$) by the *largest amount*. The process of adding such an arc (t,s) to A_T and deleting the artificial arc (r,s) from this basis is equivalent to a simplex basis exchange. The setting of the node potential of node s after performing this basis exchange simply maintains complementary slackness.

Thus, the label-correcting and label-setting methods are both simply variants of the same general algorithm. More specifically, they are both special purpose primal simplex methods which use different pivot strategies. It is well known in linear programming literature that searching for the variable which violates dual feasibility by the largest amount at each iteration to enter the basis does not usually produce good solution times. In fact, such an approach normally results in unusually large solution times. However, in the case of shortest path problems with nonnegative arc lengths, the following subsections demonstrate that various researchers have devised ingenious ways of exploiting the topology of the problem so that such a pivot strategy can be performed by examining *each variable at most once*.

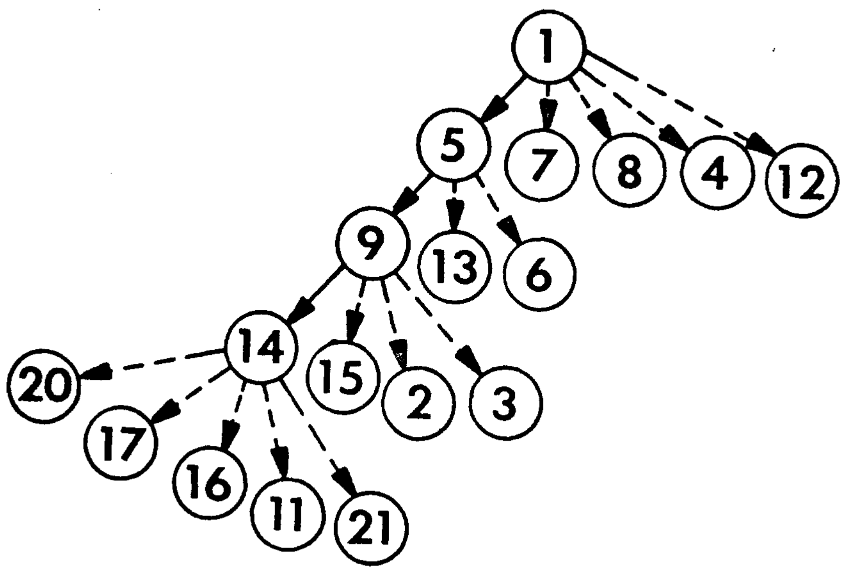


FIG. 3 - LABEL-SETTING ITERATION

7.2 <u>Dijkstra Address Calculation Sort</u>

The first implementation to be discussed is the one originally developed by Dial [6], called code S1. Several studies [9,23] of shortest path algorithms have concluded that code S1 is the fastest code, superior to all other label-setting and label-correcting implementations.

The Dial code operates in accordance with the previous observations by keeping a unique temporary node potential and predecessor for each node v in

N_E such that $d(v) = \underset{u \, \varepsilon \, N_T}{\text{minimum}} \, (d(u) + \ell(u,v)$, and maintaining $p(v) = u$ for a node u satisfying $d(v) = d(u) + \ell(u,v)$. Likewise, at each iteration, a node v in N_E with the minimum temporary node potential is added to N_T and its arc $(p(v),v)$ is added to A_T.

The chief feature of code S1 is the manner in which temporary node potentials are updated and their minimum is identified. In particular, after adding node v to N_T, the updating is accomplished simply by scanning the forward star of node v. The new candidate values for node potentials imputed by these arcs are then calculated and compared with their current temporary node potentials, retaining the smaller one with its corresponding predecessor.

The Dial implementation then identifies the minimum temporary node potential using the following observation. Each temporary node potential equals a permanent node potential plus the length of some arc. Consequently, temporary node potential values may be *uniquely* represented modulo $(\ell_{max} + 1)$ where $\ell_{max} = \underset{a \, \varepsilon \, A}{\text{maximum}} \, \ell(a)$. That is, if $d(p) \neq d(q)$, where $d(p)$ and $d(q)$ are *temporary node potentials*, then $d(p)$ modulo $(\ell_{max} + 1) \neq d(q)$ modulo $(\ell_{max} + 1)$.

To see this, suppose that node v has the minimum temporary node potential at the current iteration. Then $d(u) \leq d(v)$ for $u \, \varepsilon \, N_T$ and thus for $t \, \varepsilon \, N_E$ $d(v) \leq d(t) \leq d(v) + \ell_{max}$. In other words, at each iteration all temporary node potentials are bracketed on the lower side by $d(v)$ and on the upper side by $d(v) + \ell_{max}$. Thus it is possible from one iteration to the next to uniquely represent all temporary node potentials modulo $(\ell_{max} + 1)$.

To find the minimum by this procedure, it is convenient to use a computer array k of size $\ell_{max} + 1$ where

$$k(i) = \begin{cases} 0 \text{ if } i \neq d(v) \text{ modulo } (\ell_{max} + 1), \text{ for any } v \in N_E \\ p_i \text{ if } i = d(q) \text{ modulo } (\ell_{max} + 1), \text{ for some } q \in N_E, \end{cases}$$

where p_i is a pointer which points to all nodes in N_E that have a modulo temporary node potential value of i. The nodes in N_E that have the same modulo temporary node potential value (and thus, on any given iteration, the same temporary node potential value) are identified by chaining the nodes by a two-way linked list. Thus, every node with the same temporary potential value is linked to an antecedent and a successor node (which may be dummies at the "ends" of the list). When a node's temporary potential changes, the node is disconnected from the chain simply by re-linking its antecedent and successor to each other. This array achieves an "automatic sort" of the nodes in N_E relative to their temporary node potentials. Figure 4 illustrates the sort structure induced by the k array and the two-way linked lists, representing node names by the symbol n_i.

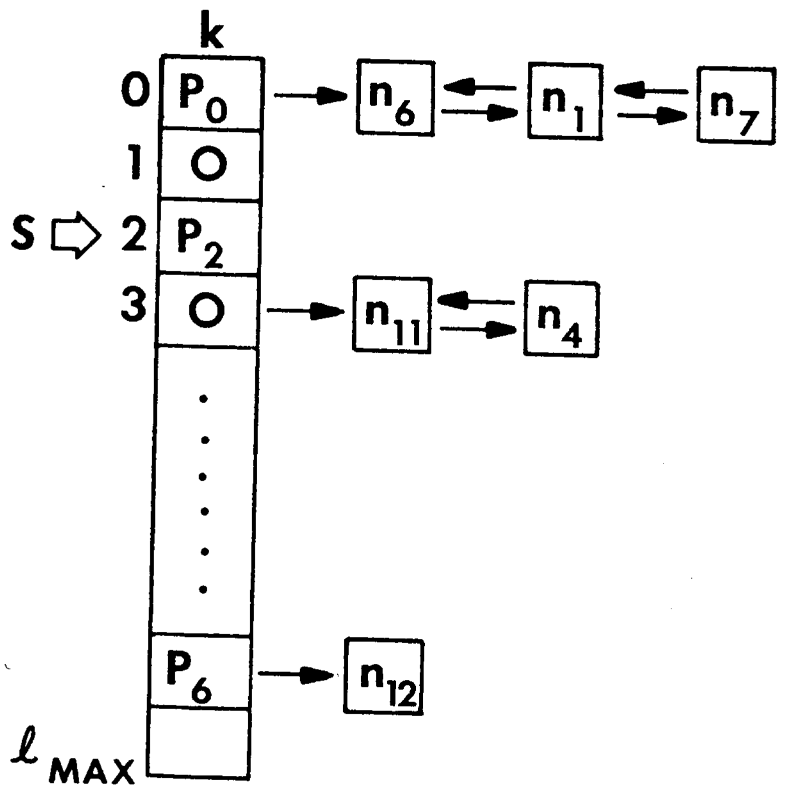


FIG. 4 - ADDRESS CALCULATION SORT

The current minimum temporary node potential is found by sequentially examining the elements of k in a wrap around fashion. Each time a nonzero element of k is encountered, the current minimum node potential is that of the nodes associated with this element, and examination of k resumes at the next nonzero element of k on the next iteration.

To describe the implementation of this algorithm, it is convenient to define the following terms:

1. The *imputed node potential value of node q*, relative to the forward star of v, denoted by $d_v(q)$, is $d(v) + \ell(v,q)$.
2. An *improving imputed node potential* $d_v(q)$ is one such that $d_v(q) < d(q)$; i.e., $d_v(q)$ is smaller than the current minimum temporary node potential of node q.
3. Node q is an *improving node* relative to FS(v) if it has an improving imputed node potential.
4. A node v is *scanned* by examining FS(v) and updating d(q) and p(q) for each improving node $q \in FS(v)$; i.e., $d(q) := d_v(q)$ and $p(q) = v$.

To implement this approach, the algorithm initializes $p(v) = 0$, $v \in N$; $d(r) = 0$ and $d(v) = \infty$, $v \in N - \{r\}$; and $k(i) = 0$, $0 \leq i \leq \ell_{max}$. The root node r is then scanned and the improving nodes of FS(r) are "added to" the appropriate elements of k. The first pass of the k list starts at k(0), examining the elements of k in sequence until the first nonzero element is encountered. Each node v associated with this nonzero element is then sequentially removed from the two-way chained list and scanned. Any improving node q located during the scan of v is removed from "its current position" in k and moved to its new position $d_v(q)$ modulo $(\ell_{max} + 1)$. (If $d(q) = \infty$ then node v has never been added to k and thus no step is required to remove it.)

At each subsequent iteration, the examination of array k resumes where it left off (and wraps around if necessary) to find the first nonzero entry. This entry identifies a node with the new minimum temporary node potential. All chained nodes with this temporary node potential are then removed from k and scanned in the manner previously indicated. The algorithm stops when a complete pass of k is made without finding a nonzero entry.

This approach is called an *address calculation sort* because the insertion and deletion of an item from the list simply involves calculating an address in a convenient and straight forward manner. Its application to shortest path implementations, as proposed and coded by Dial, is known in the literature as CACM Algorithm 360 (see [6]). This algorithm, as noted earlier, was found by Gilsinn and Witzgall [9], as well as by authors of several unpublished studies, to be the most efficient shortest path method for problems with non-negative arc lengths.

Two attractive features of this algorithm, in addition to its efficiency, are its simplicity and the structuring which assures that each arc is examined at most once. This latter feature, which is independent of the use of the address calculation sort, follows from the fact that an arc is scanned in a given iteration if and only if its starting node has a minimum node potential at that iteration. Every node "reachable" from the root must have a minimum potential at some step, but never more than once, thus only the arcs starting at reachable nodes are examined at all.

This implementation has two major time consuming tasks: (1) inserting and deleting nodes in the two-way linked array when their node potentials are reduced, and (2) examining the elements of k to find the next minimum. The time required by the first task is partially illustrated by the increasing solution times of Table II when the number of arcs is increased. The effort of the second task is dramatically shown by comparing the times in Tables I and II for the different arc length ranges.

These solution times generally show that the algorithm's performance depends on the maximum arc lengths, number of nodes, and number of arcs. Each of these items has a direct or indirect influence on the two main computational tasks. More specifically, the maximum arc length directly affects the sparseness of the k array (as measured by $t/(\ell_{max} + 1)$ where t equals the number of nonzero entries in k). As the sparseness of k increases, more elements of k must be examined at each iteration to find the new minimum.

The number of nodes and the number of arcs in the network both indirectly affect the sparseness of k since these parameters influence the number of nodes with temporary node potentials. Additionally, these parameters affect the number of nodes whose node potentials decrease from iteration to iteration and thus require relocation in the two-way linked lists.

Another limitation of this implementation stems from its computer memory requirements. In particular, the k array is of size $\ell_{max} + 1$ which can be prohibitive for large arc lengths. Different ways of coping with these limitations are discussed subsequently.

7.3 Dantzig Address Calculation Sort

One way to reduce the effort of inserting and removing nodes on the two-way linked list is to postpone adding nodes to the list. This can be done by observing that it is unnecessary to scan the entire forward star of the node v when it is assigned a permanent node potential. In particular, only the endpoint of a minimum length arc in such a forward star needs to be considered for addition to k. This follows from the fact that all temporary node potentials determined from node v will be greater than or equal to the node potential determined for the endpoint of a minimum length arc of FS(v). We now describe an approach designed to exploit this observation.

In order to limit the nodes considered for addition to k by selecting a minimum length arc from FS(v), it is convenient to store the network G(N,A) in a *sorted forward star form*. George Dantzig [4] was the first to suggest this type of scheme, and thus we refer to it as the Dantzig address calculation sort.

At first glance, the Dantzig address calculation sort appears to incur substantial pre-processing work--a fact that has apparently discouraged other researchers from pursuing this approach. Indeed, for a "one-shot" solution of the shortest path problem, the effort devoted to organizing the data in a sorted forward star form outweighs the advantages to be gained. However, it is important to recognize that the construction of a large transporation network, as must commonly be done for a large city, costs hundreds of thousands of dollars. Further, once this data base is constructed, it is used again and again to find shortest path trees for alternative root nodes. These repeated applications can all be based on a single pre-processing effort.

Additionally, changes to the data base of such large transporation networks generally involve only a small portion of the overall configuration (adding or deleting certain arcs, or changing the lengths of others). Thus, minimal additional work is required to amend the sorted forward star form to accommodate the effect of such changes.

It is possible to take advantage of a network in sorted forward star form by modifying the code S1 in the following principal way. The improving nodes of the forward star of each node in N_T are sequentially added to the two-way linked list (the two-way linked list is actually replaced by a one-way linked list in this implementation) as the previous node of N_T is removed. Thus, the one-way linked list contains at most as many nodes as nodes in N_T.

Additionally, each time a node n_i is added to the one-way linked list, the predecessor of n_i at the time it is added (i.e., the forward star node which put node n_i on the list) is paired with n_i and added to the list. That is, each item on the one-way linked list is a pair which consists of a node and its predecessor. This has several advantages. First, it allows a node to appear more than once on the one-way linked list and thus eliminates the need to move nodes when their temporary node potentials are decreased. This, in turn, postpones the removal of a duplicate node from the one-way linked list until the temporary node potential imputed to this node by its paired predecessor is a minimum. This correspondingly postpones the scan of this predecessor to identify its next improving node as long as possible.

The algorithm basically operates in the manner previously described for S1 except that: (1) The two-way linked list is replaced by a one-way linked list. (2) The forward star of each node v in N_T is scanned until an improving node u is found, whereupon u is placed on the linked list with its predecessor v, and p(v) is set to v and d(v) is set to $d(p(v)) + \ell(p(v),v)$. (Node p(v) is not scanned again until the ordered pair (v,p(v)) is removed from the linked list.) (3) k is sequentially searched for the next minimum as before.

It should be noted in this implementation, however, that the next nonzero element of k may not point to the next minimum, as was the case for S1. Thus when a node v is removed from the linked list, it is discarded if its paired predecessor differs from its current predecessor in array p, since this implies that v has already been assigned a permanent node potential. In any event, the predecessor paired with v is scanned for its next improving node. If an improving node is found, it is added to the linked list in the manner already described.

In the case that v's paired predecessor is equal to its current predecessor p(v), then v's temporary node potential is a minimum and v is assigned a permanent potential and added to N_T. Further, node v is scanned as described in step 2. Code S2 embodies this implementation. A precise description of the implementation is given by the listing of the code in the appendix.

The advantages of this implementation are: (1) the algorithm can be terminated when all nodes are permanently labeled; (2) a node is never moved on the linked list when its node potential is improved; and (3) the postponement of adding temporary node potentials to k deeps less information on d and potentially avoids adding dominated values to k.

Because of (1) it is not necessary for k to be empty; consequently, even when all nodes are reachable from the root, it is *not* necessary to examine each arc once. The strategy of (2) could have been applied in the Dial implementation, but is not, because in the Dial implementation if a node is duplicated on the linked list, the number of nodes on the linked list could be as large as the number of arcs. This is normally prohibitive because of computer memory space. However, in the S2 implementation, the number of nodes on the linked list will never exceed the number of nodes in the problem since there is at most one node on the linked list for each node in N_T.

The computational results in Tables I and II reflect these advantages. The results in Table II indicate that the code S2 strictly dominates code S1 on problems with 10,000 or more arcs (i.e., problems with an average of 10 or more arcs per node). A thorough analysis of these results indicates that this dominance results primarily from advantage (1) above. Namely, on problems with 10 or more arcs per node, S2 examines only a subset of the arcs before stopping. This indicates that the superiority of code S2 should become more pronounced on denser problems. In addition, the results in Table II indicate that code S2 is the fastest code for problems with 10,000 or more arcs in the 1-200 arc length range and for problems with 15,000 or more arcs in the 1-10,000 arc length range.

The results in Table I, however, indicate that code S2 is inferior to code S1 for grid problems. This is due to the fact that code S2 has to examine almost every arc on these sparse problems. Dantzig in [4] suggests pre-ordering the arc lengths in each forward star before solving the problem. Thus, we called the above code the Dantzig address calculation sort. Next we briefly discuss a number of our attempts to improve this method.

7.4 Improvements to the Dantzig Address Calculation Sort

Recall that code S2 keeps at most one entry on k for each node with a permanent node potential. Thus for problems with 1000 nodes and with arcs in the 1-10,000 arc length range, k is very sparse. As a result, a lot of time is spent searching for the next nonzero entry of k.

In an effort to reduce this search time, we tried two different implementation strategies. The first was simply to partition k into segments of equal length and to keep counters of the number of nonzero entries in each segment. This was done for segment sizes of 16, 32, 64, 128, and 256. The algorithm then examined the counters to determine if any of their associated elements contained a nonzero entry. If not, all the elements of the segment could be skipped without being submitted to examination. The results of this testing are not shown in Table II because this procedure did not improve solution times.

This testing did disclose an interesting piece of information, however. Namely, the tests indicated that the nonzero entries of k are approximately uniformly distributed in R. (Note this is probably due to the fact that the arc lengths were generated using a uniformly distributed probability distribution.

Thus, the above results may not hold for problems whose arc lengths do not satisfy this property.) Due to the sparseness of k, this implies that each counter value is small and thus each segment of k contains very few nonzero entries.

To take advantage of this finding, we aggregated the segments of k. That is, rather than chaining together nodes with the same temporary node potentials, we chained together all nodes in each segment. We then linearly sorted the elements of a segment at the point at which it was selected for examination. This type of sort is called a *single radix sort* [19] and the radix r is the size of each segment. Code S3 is a modification of code S2 and uses a single radix sort.

The results in Tables I and II indicate that code S3 dramatically dominates codes S1 and S2 on grid problems. Further, code S3, in contrast to codes S1 and S2, is very stable as rectangularity varies. Similarly, the results in Table II indicate that code S3 strictly dominates code S1 and dominates code S2 on the sparser random networks. As density reaches 20 arcs per node, code S2 dominates code S3.

Besides its computational improvement, the single radix sort has an additional advantage: It requires less computer memory. The size of the k array is reduced from $(\ell_{max} + 1)$ to $(\ell_{max} + 1)/r$.

However, better computational bounds (based on worst case analysis) are available for balanced and unbalanced binary sort procedures [16,19] than for the single radix sort procedure. Consequently, we developed a code, S4, based on the Dantzig approach using an unbalanced binary sort to test whether the better theoretical worst case bounds might supply a practical advantage. Tables I and II indicate that S4 is slower than S1 and S2. We did not use a balanced binary sort, which has a still better bound (i.e., logarithmic bound) than the unbalanced binary sort, because the Gilsinn and Witzgall study [9] as well as other unpublished studies found the Dijkstra algorithm using a balanced binary sort to be slower than code S1.

Without going into great detail, an unbalanced binary sort works by keeping a binary tree of numbers (nodes) with a root number (node). A number is added to the list by comparing the number with the root. If the number is smaller, it is moved downward to the left and compared next with the number in that position. If the number is larger, it is moved downward to the right and compared next with the number in that position. This type of comparison and movement continues until the bottom of the tree is reached along some path. At this point, the number is hung to the left if it is smaller than the last tree number to which it was compared. Otherwise, it is hung to the right. The minimum is always the left-most node in the tree. The tree is called an unbalanced binary tree because the depth of the bottom nodes in the binary tree may vary greatly.

8.0 EVALUATION SUMMARY

8.1 Solution Times

The results in Tables I and II indicate that the code S1 previously believed to be the fastest code for calculating the shortest path from one to all other nodes in a network is dominated by codes C2, S2, and S3. Further, the study shows that the most efficient solution procedure depends on the topology of the network and the range of the arc length coefficients. On grid networks and sparse random networks code C2 is the fastest. In fact, this code is sometimes an order of magnitude faster than S1. As density increases, code S2 dominates C2. This dominance depends both on density and the range of the arc length coefficients. For example, for a problem whose arc lengths are in the 1-200 range, code S2 dominates C2 when the average number of arcs per node exceeds 10; however, for a problem whose arc lengths are in the 1-10,000 range, code S2 does not dominate C2 until the average number of arcs per node exceeds 15.

8.2 Memory Requirements

Table III contains the computer array requirements of each code. Code C2 not only computationally dominates the other codes on grid and sparse network problems, but also dominates them in terms of computer memory requirements. Table II indicates the paradox involved in using the label-setting codes to solve large shortest path problems. In particular, code S2 is the fastest of all the codes (including C2) on dense problems but requires substantial computer memory which often would prohibit using it to solve such problems.

8.3 Limitations

This study has examined the efficiency of algorithms when all problem data is kept in fast access main computer memory. It is exceedingly important to the realm of ultra large-scale applications, which are arising with increasing frequency, to similarly examine design principles for efficient computer codes and to determine the best algorithmic rules for the situation in which problem data is exchanged between main computer memory and peripheral storage.

The creation and testing of methods with ultra large-scale capabilities to identify the precise trade-offs of mathematical and computational considerations in an environment where data must be allocated and transferred between different types of memory will require substantial research. It is our belief, based on the present study, that the best implementation principles to emerge from such research will be based on the design of code S3. This belief may seem paradoxical since code S3 is clearly dominated by other codes. The belief largely rests on the fact that the use of peripheral storage will make it impractical to randomly access arc data. All other codes require random access of arc data. Further, if random access is not used then we feel that updating the node potentials in the manner accomplished in code C3 will prove extremely valuable.

Table III

COMPUTER ARRAY SPACE

Code	Node Length	Arc Length	Node Length Logical	Other
C1	4	2	1	
C2	4	2	1	
C3	6	2		
C4	7	2	1	
C5	7	2	1	
S1	5	2	1	$1\ (\ell_{max} + 1)$
S2	6	2		
S3	6	2		$1 \left(\frac{\ell_{max} + 1}{r} \right)$
S4	10	2		

Where r is the size of the radix.

Acknowledgments

This paper has greatly benefited from discussion with leading researchers. In particular, we wish to thank Dr. Ellis Johnson for pointing out a number of potential advances of the Dantzig algorithm. Further, we wish to acknowledge the apt comments and direct assistance of Professor Uwe Pape in developing code S2. The assistance of Fernando Palacios-Gómez in developing the Dantzig binary sort code S4 is sincerely acknowledged.

The authors also wish to acknowledge the cooperation of the staff at The University of Texas Computation Center and the editorial assistance of Dr. John Hultz.

REFERENCES

1. R. Barr, F. Glover, and D. Klingman, "Enhancements of Spanning Tree Labeling Procedures for Network Optimization," Research Report CCS 262, Center for Cybernetic Studies, University of Texas at Austin, 1976.

2. G. Bradley, G. Brown, and G. Graves, "Design and Implementation of Large Scale Primal Transshipment Algorithms," Technical Report NPS55BZBW76091, Naval Postgraduate School, Monterey, California, 1976.

3. A. Charnes, F. Glover, D. Karney, D. Klingman, and J. Stutz, "Past, Present, and Future of Development, Computational Efficiency, and Practical Use of Large-Scale Transportation and Transshipment Computer Codes," Computers and O.R., 2 (1975).

4. D. Dantzig, Linear Programming and Extensions, Princeton University Press, Princeton, New Jersey, 1963.

5. J. Dennis, "A High-Speed Computer Technique for the Transportation Problem," JACM, 8 (1958), 132-153.

6. R. Dial, "Algorithm 360 Shortest Path Forest with Topological Ordering," Communications of the ACM, 12 (1969), 632-633.

7. E. Dijkstra, "A Note on Two Problems in Connexion with Graphs," Numerical Mathematics, 1 (1959), 269-271.

8. S. Dreyfus, "An Appraisal of Some Shortest-Path Algorithms," Operations Research, 17 (1969), 395-412.

9. J. Gilsinn and C. Witzgall, "A Performance Comparison of Labeling Algorithms for Calculating Shortest Path Trees," NBS Technical Note 772, U.S. Department of Commerce, 1973.

10. F. Glover, D. Karney, and D. Klingman, "Implementation and Computational Study on Start Procedures and Basis Change Criteria for a Primal Network Code," Networks, 4 (1974), 191-212.

11. F. Glover, D. Karney, D. Klingman, & A. Napier, "A Computational Study on Start Procedures Basis Change Criteria, and Solution Algorithms for Transportation Problems," Management Science, 20 (1974), 793-813.

12. F. Glover, D. Klingman, and J. Stutz, "The Augmented Threaded Index Method for Network Optimization," INFOR, 12 (1974), 293-298.

13. B. Golden, "Shortest Path Algorithms: A Comparison," Research Report OR 044-75, Massachusetts Institute of Technology, 1975.

14. R. Helgason, J. Kennington, and H. Lall, "Primal Simplex Network Codes: State-of-the-Art Implementation Technology," Technical Report IEOR 76014, Department of Industrial Engineering and Operations Research, Southern Methodist University, Dallas, Texas, 1976.

15. T. Hu, "Revised Matrix Algorithms for Shortest Paths," SIAM Journal of Applied Mathematics, 15 (1967), 207-218.

16. E. Johnson, "On Shortest Paths and Sorting," Proceedings of the ACM 25th Annual Conference, (1972), 510-517.

17. D. Karney and D. Klingman, "Implementation and Computational Study on an In-Core Out-of-Core Primal Network Code," Operations Research, 24 (1976).

18. D. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1973.

19. D. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

20. E. Moore, "The Shortest Path Through a Maze," Proceedings of the International Symposium on the Theory of Switching, 1957.

21. J. Mulvey, "Column Weighting Factors and Other Enhancements to the Augmented Threaded Index Method for Network Optimization," Joint ORSA/TIMS Conference, San Juan, Puerto Rico, (1974).

22. U. Pape, "Implementation and Efficiency of Moore-Algorithms for the Shortest Route Problem," Mathematical Programming, 7 (1974), 212-222.

23. D.W. Robinson, "Analysis of a Shortest Path Algorithm for Transportation Applications," Control Analysis Corporation, Technical Report, March 1976.

24. V. Srinivasan and G. Thompson, "Accelerated Algorithms for Labeling and Relabeling of Trees with Applications for Distribution Problems," JACM, 19 (1972), 712-726.

A P P E N D I X

```
      SUBROUTINE S2
C
C     S2 FINDS THE SHORTEST PATHS TO ALL NODES FROM THE ROOT R
C     FOR THE NETWORK DEFINED BY THE ARRAYS A, N, AND L.  IT RETURNS
C     THE SHORTEST PATH TREE IN THE ARRAYS P AND D.
C     ALL VARIABLES ARE TYPE INTEGER.
C
C     DEFINITION OF VARIABLES
C
C     VARIABLE  LENGTH IF
C     NAME      AN ARRAY                  MEANING
C     --------  ---------                 -------
C
C     NODE                                NUMBER OF NODES IN THE NETWORK
C     LMAX                                MAXIMUM ARC LENGTH PLUS ONE
C     INFIN                               POSITIVE CONSTANT GREATER THAN ANY
C                                           PATH (I.E., +INFINITY)
C     R                                   THE ROOT NODE
C     N         NO. OF ARCS               LIST OF TO-NODES FOR THE ARCS IN THE
C                                           NETWORK IN ORDER BY FROM NODE
C     L         NO. OF ARCS               ARC LENGTHS FOR THE ARCS IN N
C     A         NODE+1                    ENTRY POINTS INTO THE N AND L LISTS
C     P         NODE                      PREDECESSORS OF THE NODES IN THE
C                                           SHORTEST PATH TREE
C     D         NODE                      LENGTH OF PATH FROM ROOT (POTENTIAL)
C     IASAVE    NODE+1                    ARRAY FOR SAVING ARRAY A
C     S         NODE+1                    SUCCESSOR ARRAY FOR SORT AND FREE LIST
C                                           ENTRIES
C     K         LMAX                      POINTER TO FIRST ENTRY OF SORT LISTS
C     T         NODE+1                    NODES TO BE SCANNED
C     H         NODE+1                    PREDECESSORS OF THE NODES IN T WHEN
C                                           NODE WAS ADDED TO SORT LIST
C     IFREE                               HEAD OF FREE STORAGE LIST
C     NSET                                NUMBER OF NODES LABELED
C     Z                                   NEXT SORT LIST TO BE EXAMINED
C
C     ENTRY POINTS
C
C     S2 HAS TWO ENTRY POINTS
C       S20 MUST BE REFERRED TO IN THE FIRST CALL AND
C       S2 MUST BE USED IN ALL SUBSEQUENT CALLS
C
        COMMON NODE,LMAX,R,INFIN,IFREE
        COMMON N( 1200),L( 1200),D( 101),P( 101),A( 101),
     1    S( 101),T( 101),K( 250),H( 101),IASAVE( 101)
        INTEGER E,TES,Y,A,U,Z,AUX,P,S,V,R,T,N,D,TEM,X,MEM,H
C     ENTRY FOR ALL CALLS AFTER THE FIRST
C     RESTORE A ARRAY FROM IASAVE
        NODEP=NODE+1
        DO 70 I=1,NODEP
70      A(I)=IASAVE(I)
        GO TO 75
C     ENTRY POINT FOR FIRST CALL
        ENTRY S20
C     SORT ARCS FOR EACH NODE BY INCREASING ARC LENGTH
C       SORT USES A SHELL SORT WITH INTERVALS 4 AND 1
        DO 77 I=1,NODE
        I1=A(I)
        I2=A(I+1)
        NN=I2-I1
        IF(NN-1) 65,64,65
```

```
86       A(I)=0
         GO TO 77
85       IF(NN.GE.13) GO TO 78
         ID=1
         GO TO 83
78       ID=4
83       NN=NN+I1-1
79       IDP=ID+I1
         DO 80 J=IDP,NN
         KKID=J
         KK=KKID-ID
         JN=N(J)
         JL=L(J)
82       IF(JL.GE.L(KK)) GO TO 81
         N(KKID)=N(KK)
         L(KKID)=L(KK)
         KKID=KK
         KK=KK-ID
         IF(KK.GE.I1) GO TO 82
81       N(KKID)=JN
         L(KKID)=JL
80       CONTINUE
         IF(ID.EQ.1) GO TO 84
         ID=1
         GO TO 79
84       N(I2-1)=-N(I2-1)
77       CONTINUE
C     SAVE ARC DATA ENTRY POINT ARRAY A IN IASAVE
         NODEP=NODE+1
         DO 76 I=1,NODEP
76       IASAVE(I)=A(I)
75       CONTINUE
C     INITIALIZE DATA
C        SET SHORTEST DISTANCES TO INFINITY
C        SET UP FREE STORAGE LIST IN S
C        CLEAR OTHER ARRAYS
         DO 1 U=1,NODE
         D(U)=INFIN
         H(U)=0
         S(U)=U+1
         T(U)=0
         P(U)=0
1        CONTINUE
         IFREE=1
         S(NODE+1)=0
         DO 2 Z=1,LMAX
         K(Z)=0
2        CONTINUE
C
C     START ALGORITHM
         MEM=0
         LEV=0
         Z=1
C     LABEL ROOT NODE R WITH DISTANCE ZERO AND SCAN AS PREDECESSOR NODE
         D(R)=0
         MSET=1
         U=R
         GO TO 38
C
C     SET FINAL SHORTEST DISTANCE (LABEL) FOR NODE U
3        CONTINUE
C     EXIT IF ALL NODES HAVE BEEN LABELED
```

```
      NSET=NSET+1
      IF(NSET.EQ.NODE) GO TO 15
12    E=A(U)
C   CHECK IF ALL ARCS OUT OF U HAVE BEEN SCANNED
      IF(E.EQ.0) GO TO 58
C   SET W EQUAL TO THE TO-NODE AND UPDATE POINTER A(U)
      W=N(E)
      IF(W.GT.0) GO TO 39
C   THIS IS THE LAST ARC OUT OF U
      W=-W
      A(U)=0
      GO TO 40
39    A(U)=E+1
C   TES IS THE NEW DISTANCE TO W FROM U VIA THE ARC (U,W)
40    TES=L(E)+D(U)
C   CHECK IF ARC IS IMPROVING. IF NOT GET NEXT ARC OUT OF U.
      IF(D(W).LE.TES) GO TO 12
C   GET TES MOD(LMAX)
5     Y=TES-LEV
9     IF(Y.LE.LMAX) GO TO 10
      Y=Y-LMAX
      GO TO 9
C   ARC (U,W) IS AN IMPROVING ARC WITH REDUCED SHORTEST DISTANCE
C   TES AND Y = TES MOD(LMAX)
C   PUT THIS ARC IN FIRST FREE LOCATION AND ADD TO BEGINNING OF
C   LIST Y AND UPDATE PREDECESSOR AND DISTANCE
10    X=K(Y)
      I=IFREE
      IFREE=S(I)
      S(I)=X
      T(I)=W
      H(I)=U
      K(Y)=I
11    D(W)=TES
      P(W)=U
      MEM=1
C   END OF SCAN OF NEWLY LABELED NODE
C
C   GET NEXT ARC OUT OF PREDECESSOR NODE
58    U=IU
38    E=A(U)
C   CHECK IF ALL ARCS OUT OF U HAVE BEEN SCANNED
      IF(E.EQ.0) GO TO 52
C   SET W EQUAL TO THE TO-NODE AND UPDATE POINTER A(U)
      W=N(E)
      IF(W.GT.0) GO TO 53
C   THIS IS THE LAST ARC OUT OF U
      W=-W
      A(U)=0
      GO TO 54
53    A(U)=E+1
C   TES IS THE NEW DISTANCE TO W FROM U VIA THE ARC (U,W)
54    TES=L(E)+D(U)
C   CHECK IF ARC IS IMPROVING. IF NOT GET NEXT ARC OUT OF U.
      IF(D(W).LE.TES) GO TO 38
C   GET TES MOD(LMAX)
      Y=TES-LEV
55    IF(Y.LE.LMAX) GO TO 56
      Y=Y-LMAX
      GO TO 55
C   ARC (U,W) IS AN IMPROVING ARC WITH REDUCED SHORTEST DISTANCE
C   TES AND Y = TES MOD(LMAX)
```

```
C    PUT THIS ARC IN FIRST FREE LOCATION AND ADD TO BEGINNING OF
C    LIST Y AND UPDATE PREDECESSOR AND DISTANCE
56     X=K(Y)
       I=IFREE
       IFREE=S(I)
       S(I)=X
       T(I)=W
       H(I)=U
       K(Y)=I
       D(W)=TES
       P(W)=U
       MEM=1
C    END OF SCAN OF PREDECESSOR NODE
C
52     CONTINUE
C    GET NEXT NODE TO BE SCANNED OFF OF LIST Z
13     I=K(Z)
       IF(I.NE.0) GO TO 14
       Z=Z+1
       IF(Z.LE.LMAX) GO TO 13
C    EXIT IF NO IMPROVING ARCS FOUND ON THIS PASS OF LISTS
       IF(MEM.EQ.0) GO TO 15
C    SET UP FOR NEXT PASS OF LISTS
       MEM=0
       LEV=LEV+LMAX
       Z=1
       GO TO 13
C    REMOVE ARC (H(I),T(I)) FOR LIST Z AND RETURN LOCATION I TO FREE
C    LIST
14     K(Z)=S(I)
       S(I)=IFREE
       IFREE=I
       U=T(I)
       IU=H(I)
C    IF SHORTEST DISTANCE HAS BEEN DECREASED (AND PREDECESSOR CHANGED),
C    SCAN ONLY THE PREDECESSOR NODE
       IF(IU.EQ.P(U)) GO TO 3
       GO TO 58
15     CONTINUE
       RETURN
       END
```

```
      SUBROUTINE C2
C
C    C2 FINDS THE SHORTEST PATHS TO ALL NODES FROM THE ROOT R
C    FOR THE NETWORK DEFINED BY THE ARRAYS A, N, AND L.
C    IT RETURNS THE SHORTEST PATH TREE IN THE ARRAYS P AND D.
C    ALL VARIABLES IN C2 ARE TYPE INTEGER.
C
C    DEFINITION OF VARIABLES
C
C    VARIABLE  LENGTH IF
C    NAME      AN ARRAY             MEANING
C    --------  ---------            -------
C
C    NODE                      NUMBER OF NODES IN THE NETWORK
C    R                         THE ROOT NODE
C    INF                       POSITIVE CONSTANT GREATER THAN ANY
C                                PATH LENGTH (I.E. +INFINITY)
C    N         NO. OF ARCS     LIST OF TO-NODES OF THE ARCS IN THE
C                                NETWORK IN ORDER BY FROM NODE
C    L         NO. OF ARCS     ARC LENGTHS FOR THE ARCS IN N
C    A         NODE+1          ENTRY POINTS INTO N AND L
C                                A(I) POINTS TO THE FIRST ARC OUT OF
C                                  NODE I FOR I=1,2,...,NODE
C                                A(N+1) IS 1 PLUS THE NUMBER OF ARCS
C    P         NODE            PREDECESSORS OF THE NODES IN THE SHORTEST
C                                PATH TREE
C    D         NODE            LENGTHS OF THE SHORTEST PATHS
C    CL        NODE            QUEUE OF NODES TO BE SCANNED
C
      COMMON NODE,R,N( 1200),L( 1200),A( 101),D( 101),CL( 101)
      COMMON P( 101),INF
      INTEGER D,A,CL,P,R
C
C    INITIALIZATION
      DO 100 I=1,NODE
C    SET DISTANCES TO INFINITY, CLEAR PREDECESSORS AND QUEUE
      D(I)=INF
      P(I)=0
 100  CL(I)=0
C    SET DISTANCE OF ROOT TO ZERO
      D(R)=0
C    SET QUEUE TO CONTAIN ONLY ROOT R
      CL(R)=INF
      I=R
      NT=R
C
C    MAIN LOOP OF ALGORITHM
C
C    SCAN THE ARCS OUT OF NODE I - ARCS A(I) TO A(I+1)-1
 120  IA=A(I+1)-1
      ID=D(I)
      IA1=A(I)
      IF (IA1.GT.IA) GO TO 201
      DO 200 IR=IA1,IA
      K=N(IR)
C    COMPUTE DISTANCE TO NODE K USING ARC (I,K)
      KD=ID+L(IR)
C    CHECK FOR DECREASE IN SHORTEST DISTANCE
C    IF NOT IMPROVING, GET NEXT ARC
      IF (KD.GE.D(K)) GO TO 200
C    DECREASE OF SHORTEST DISTANCES WITH ARC (I,K)
```

```
C    UPDATE PREDECESSOR AND SHORTEST DISTANCE
       P(K)=I
       D(K)=KD
C    CHECK QUEUE STATUS OF NODE K
       IF (CL(K)) 16Ø,14Ø,2ØØ
C    NODE K HAS NEVER BEEN SCANNED
C    ADD NODE K TO END OF QUEUE AFTER NODE NT
 14Ø  CL(NT)=K
C    MAKE NODE K NEW END OF QUEUE NT AND FLAG AS END OF QUEUE
       NT=K
       CL(K)=INF
       GO TO 2ØØ
C    NODE K HAS ALREADY BEEN SCANNED
C    ADD NODE K TO BEGINNING OF QUEUE JUST AFTER NODE I
 16Ø  CL(K)=CL(I)
       CL(I)=K
C    NODE K IS ON QUEUE
 2ØØ  CONTINUE
C    GET NEXT NODE I FOR BEGINNING OF QUEUE
 2Ø1  ICL=CL(I)
C    FLAG NODE I AS HAVING BEEN SCANNED
       CL(I)=-1
       I=ICL
       IF (I.LT.INF) GO TO 12Ø
C
C      END OF THE ALGORITHM
C
       RETURN
       END
```

```
C
C    GRID NETWORK SHORTEST PATH TEST PROBLEM GENERATOR
C
C    GENERATES MULTIPLE GRID NETWORKS WITH SPECIFIED DIMENSIONS AND
C    ARC LENGTH RANGE.  EACH PROBLEM IS FOLLOWED BY A LIST OF NODES
C    WHICH CAN BE USED AS THE ROOT NODE.
C    PROBLEMS ARE GENERATED ON TAPE 2.
C
C    INPUT    (FROM STANDARD SYSTEM INPUT DEVICE)
C
C    RECORD 1 - RANDOM NUMBER SEED (F10.10)
C    RECORD 2 - (4I5)
C                 MM - NUMBER OF ROWS IN GRID
C                 NN - NUMBER OF COLUMNS IN GRID
C                 NAVC - AVERAGE ARC LENGTH, ARC LENGTHS WILL BE BETWEEN
C                        1 AND 2*NAVC
C                 NROOT - NUMBER OF NODES IN THE ROOT NODE LIST
C    RECORD 3 THRU RECORD K - NODES TO BE INCLUDED IN THE ROOT NODE
C                 LIST.  FROM 0 TO NROOT NODES CAN BE SPECIFIED.  IF
C                 FEWER THAN NROOT NODES ARE ENTERED, THE INPUT LIST IS
C                 TERMINATED BY ENTERING A ZERO.  NODES ARE GENERATED
C                 RANDOMLY TO COMPLETE THE LIST OF NROOT NODES.
C                 FORMAT IS (I5).
C
C    RECORDS 2 THRU K CAN BE REPEATED FOR AS MANY PROBLEMS AS DESIRED.
C    THE LAST PROBLEM IS FOLLOWED BY A BLANK RECORD.
C
C
C    PROBLEM FORMAT
C
C    ALL RECORDS ARE WRITTEN WITH A (16I5) FORMAT
C
C    THE PROBLEM FILE CAN CONTAIN MORE THAN ONE PROBLEM.  WITHIN EACH
C    PROBLEM,
C
C      CARD 1  CONTAINS  THE TOTAL NUMBER OF NODES, THE NUMBER OF ROWS
C                        IN THE GRID, THE NUMBER OF COLUMNS IN THE
C                        GRID, THE TOTAL NUMBER OF ARCS, THE AVERAGE
C                        ARC LENGTH, NUMBER OF NODES IN THE LIST OF
C                        ROOT NODES
C
C    THE ARC DATA FOLLOWS IN SETS OF CARD IMAGES, EACH SET DESCRIBING
C    THE ARCS OUT OF A GIVEN NODE.  THE SETS GIVE THE ARCS OUT OF
C    NODE 1, NODE 2, ..., NODE N, IN ORDER.  THE LAST SET IS FOLLOWED
C    BY A BLANK RECORD.
C    WITHIN THE SET OF CARD IMAGES DEFINING THE ARCS OUT OF A NODE,
C      CARD 1  GIVES THE NUMBER OF ARCS OUT OF THE NODE
C      CARD 2  GIVES THE TO NODES FOR THESE ARCS
C      CARD 3  GIVES THE ARC LENGTHS.
C     CONTAINS ZERO, CARDS 2 AND 3 ARE OMITTED.
C    FOLLOWING THE ARC DATA IS THE LIST OF ROOT NODE CANDIDATES, ONE
C    PER RECORD WITH THE LAST ENTRY BEING FOLLOWED BY A BLANK RECORD.
C
C    THE LAST PROBLEM IN THE FILE IS FOLLOWED BY A SECOND BLANK RECORD.
C
      COMMON ITO(4),IDIST(4)
      REWIND 2
C    ENTER RANDOM NUMBER SEED
      READ(1,101) SEED
101   FORMAT(F10.10)
C    ENTER PROBLEM PARAMETERS - NUMBER OF ROWS AND COLUMNS IN GRID,
C      AVERAGE ARC LENGTH, NUMBER OF ROOTS TO BE GENERATED
C      BLANK RECORD TERMINATES INPUT
10        READ(1,100) MM,NN,NAVC,NROOT
100       FORMAT(16I5)
          CALL RANF(SEED)
C      NODES IS TOTAL NUMBER OF NODES
          NODES=MM*NN
C      NARC IS TOTAL NUMBER OF ARCS
          NARC=4*NODES-2*(MM+NN)
C      WRITE PROBLEM HEADER
C      NODES = 0 IS TRAILER
          WRITE(2,100) NODES,MM,NN,NARC,NAVC,NROOT
          IF(MM.EQ.0) STOP
C      REPLACE AVERAGE LENGTH BY TWICE AVERAGE LENGTH
C      ALL ARC LENGTHS WILL FROM 1 TO NAVC
          NAVC=2*NAVC
C      NARC COUNTS THE TOTAL NUMBER OF ARCS
          NARC=0
C      GENERATE ARCS OUT OF NODE AT GRID POSITION (M,N) BEGINNING WITH
C         (1,1) AND GOING ACROSS ROWS
C      I IS THE NODE NUMBER (1 TO MM*NN) FOR THE NODE AT (M,N)
          M=N=I=1
2         NA=0
          K=M-1
C      FOR NODE (M,N) GENERATE ARCS TO NODES ABOVE AND BELOW
          DO 1 J=1,2
C      SKIP ARC IF TO NODE IS OFF GRID
          IF(K.LT.1 .OR. K.GT.MM) GO TO 1
C      NA COUNTS ARCS OUT OF NODE (M,N)
          NA=NA+1
C      ITO IS TO NODE
          ITO(NA)=(K-1)*NN+N
C      IDIST IS ARC LENGTH BETWEEN 1 AND NAVC
          IDIST(NA)=NAVC*RANF(0)+1
1         K=K+2
          K=N-1
C      FOR NODE (M,N) GENERATE ARCS TO NODES TO LEFT AND RIGHT
          DO 3 J=1,2
          IF(K.LT.1 .OR. K.GT.NN) GO TO 3
          NA=NA+1
          ITO(NA)=(M-1)*NN+K
          IDIST(NA)=NAVC*RANF(0)+1
3          K=K+2
C      WRITE OUT NUMBER OF ARCS
          WRITE(2,100) NA
          NARC=NARC+NA
          IF(NA.EQ.0) GO TO 4
C      WRITE OUT TO NODES AND LENGTHS OF ARCS OUT OF NODE I
C      SKIP IF NONE
          WRITE(2,100) (ITO(K),K=1,NA)
          WRITE(2,100) (IDIST(K),K=1,NA)
C      ADVANCE TO NEXT NODE
4         I=I+1
C      CHECK IF DONE
          IF(I.GT.NODES) GO TO 5
C      MOVE TO NEXT COLUMN
          N=N+1
          IF(N.LE.NN) GO TO 2
C      MOVE TO NEXT ROW
          N=1
          M=M+1
          GO TO 2
```

```
5       LR=0
C     READ NEXT NODE TO BE INCLUDED IN ROOT NODE LIST
7       READ(1,100) NR
C     ZERO TERMINATES INPUT OF SPECIFIED ROOTS IF FEWER THAN NROOT
C     NODES ARE ENTERED.
        IF(NR.EQ.0) GO TO 6
C     NODE ENTERED MUST BE DISTINCT FROM PRECEDING ROOT IN LIST
        IF(NR.EQ.LR) GO TO 7
        WRITE(2,100) NR
        LR=NR
        NROOT=NROOT-1
        GO TO 7
C     STOP AFTER NROOT NODES HAVE BEEN ENTERED
6       IF(NROOT.LE.0) GO TO 8
C     GENERATE REMAINING ROOTS RANDOMLY.
9       I=RANF(0)*NODES+1
        IF(I.GT.NODES) GO TO 9
C     NODE GENERATED MUST BE DISTINCT FROM PRECEDING ROOT IN LIST
        IF(I.EQ.LR) GO TO 9
        WRITE(2,100) I
        LR=I
        NROOT=NROOT-1
        GO TO 6
C     WRITE OUT NEXT ROOT NODE
C     NR=0 TERMINATES LIST
8       WRITE(2,100) NR
        PRINT 100,NODES,NARC,MM,NN,NAVC,NROOT
        GO TO 10
        END
```

```
C     RANDOM NETWORK SHORTEST PATH TEST PROBLEM GENERATOR
C
C     GENERATES MULTIPLE RANDOM NETWORKS WITH SPECIFIED DENSITIES AND
C     ARC LENGTH RANGES.  EACH PROBLEM IS FOLLOWED BY A LIST OF NODES
C     WHICH CAN BE USED AS THE ROOT NODE.
C     PROBLEMS ARE GENERATED ON TAPE 1.
C
C     INPUT    (FROM STANDARD SYSTEM INPUT DEVICE)
C
C     RECORD 1 - RANDOM NUMBER SEED (F10.10)
C     RECORD 2 - (4I5)
C                N - NUMBER OF NODES IN PROBLEM, ZERO TERMINATES INPUT
C                NAN - AVERAGE NUMBER OF ARCS LEAVING EACH NODE, EACH
C                      NODE WILL HAVE FROM 0 TO 2*NAN ARCS.
C                      NAN MUST NOT BE GREATER THAN N / 2.
C                LAVE - AVERAGE ARC LENGTH, ARC LENGTHS WILL BE BETWEEN
C                       1 AND 2*LAVE
C                NROOT - NUMBER OF NODES IN THE ROOT NODE LIST
C     RECORD 3 THRU RECORD K - NODES TO BE INCLUDED IN THE ROOT NODE
C                LIST.  FROM 0 TO NROOT NODES CAN BE SPECIFIED.  IF
C                FEWER THAN NROOT NODES ARE ENTERED, THE INPUT LIST IS
C                TERMINATED BY ENTERING A ZERO.  NODES ARE GENERATED
C                RANDOMLY TO COMPLETE THE LIST OF NROOT NODES.
C                FORMAT IS (I5).
C
C     RECORDS 2 THRU K CAN BE REPEATED FOR AS MANY PROBLEMS AS DESIRED.
C     THE LAST PROBLEM IS FOLLOWED BY A BLANK RECORD.
C
C
C     PROBLEM FORMAT
C
C     ALL RECORDS ARE WRITTEN WITH A (16I5) FORMAT
C
C     THE PROBLEM FILE CAN CONTAIN MORE THAN ONE PROBLEM.  WITHIN EACH
C     PROBLEM,
C
C       CARD 1  CONTAINS  THE NUMBER OF NODES, AVERAGE LENGTH OF EACH
C                         ARC, AVERAGE NUMBER OF ARCS LEAVING EACH NODE,
C                         NUMBER OF NODES TO BE USED AS ROOT NODES.
C
C     THE ARC DATA FOLLOWS IN SETS OF CARD IMAGES, EACH SET DESCRIBING
C     THE ARCS OUT OF A GIVEN NODE.  THE SETS GIVE THE ARCS OUT OF
C     NODE 1, NODE 2, ..., NODE N, IN ORDER.  THE LAST SET IS FOLLOWED
C     BY A BLANK RECORD.
C     WITHIN THE SET OF CARD IMAGES DEFINING THE ARCS OUT OF A NODE,
C       CARD 1  GIVES THE NUMBER OF ARCS OUT OF THE NODE
C       CARD 2  GIVES THE TO NODES FOR THESE ARCS
C       CARD 3  GIVES THE ARC LENGTHS.
C     IF CARD 1 CONTAINS ZERO, CARDS 2 AND 3 ARE OMITTED.  IF CARD 1
C     INDICATES MORE THAN 16 ARCS, CARDS 2 AND 3 ARE REPEATED AS
C     NECESSARY.
C     FOLLOWING THE ARC DATA IS THE LIST OF ROOT NODE CANDIDATES, ONE
C     PER RECORD WITH THE LAST ENTRY BEING FOLLOWED BY A BLANK RECORD.
C
C     THE LAST PROBLEM IN THE FILE IS FOLLOWED BY A SECOND BLANK RECORD.
C
        LOGICAL DUP
        DIMENSION NLIST(500),NDIST(500),DUP(2500)
        REWIND 1
C     ENTER RANDOM NUMBER SEED
```

```
      READ600,SEED
600   FORMAT(F10.10)
1     CALL RANF(SEED)
C   ENTER NUMBER OF NODES, AVERAGE NUMBER OF ARCS PER NODE, AVERAGE
C   ARC LENGTH, AND NUMBER OF ROOTS.
      READ 500,N,NAN,LAVE,NROOT
      IF(N .LE.0) GO TO 200
      K=0
      KK=K
      WRITE(1,500) N,NAN,LAVE,NROOT
C   GENERATE ARC DATA
      KK=0
      DO 40 I=1,N
C   DUP IS A LOGICAL ARRAY SUCH THAT DUP(J) = .TRUE. IF AN ARC FORM
C       NODE I TO NODE J HAS ALREADY BEEN GENERATED AND .FALSE.
C        OTHERWISE.
      DO 41 J=1,N
41    DUP(J)=.FALSE.
      MM=2*RANF(0)*NAN+0.5
      WRITE(1,500) MM
C   IF THERE ARE NO ARCS OUT OF NODE I, SKIP TO NODE AND ARC LENGTH
C   GENERATION
      IF(MM .EQ. 0) GO TO 40
      K=MM
      KK=KK+K
C   GENERATE ARC LENGTHS FOR THE K ARCS OUT OF NODE I
      DO 50 J=1,K
      NDIST(J)=2*LAVE*RANF(0)+1
50    CONTINUE
      DO 80 L=1,K
81    Y=RANF(0)
C   LL IS NEXT TO NODE
      LL=N*Y+1
      IF(LL.GT.N) LL=N
C   CHECK FOR ARC DUPLICATION
      IF(DUP(LL)) GO TO 81
      DUP(LL)=.TRUE.
      NLIST(L)=LL
   80 CONTINUE
   40 CONTINUE
C    WRITE OUT TO NODES AND LENGTHS OF ARCS OUT OF NODE I
      WRITE(1,500) (NLIST(J),J=1,K)
      WRITE(1,500) (NDIST(J),J=1,K)
   40 CONTINUE
  500 FORMAT(16I5)
C    K IS TOTAL NUMBER OF ARCS
      K=KK
      PRINT 501,N,K,NROOT,LAVE
  501 FORMAT(1H0,4I10)
      LR=0
C   READ NEXT NODE TO BE INCLUDED IN ROOT NODE LIST
    5 READ 500,NR
      I=NR
      IF(NR.NE.0) GO TO 92
C   ZERO TERMINATES INPUT OF SPECIFIED ROOTS IF FEWER THAN NROOT
C   NODES ARE ENTERED.
C   GENERATE REMAINING ROOTS RANDOMLY.
94    I=RANF(0)*N+1
      IF(I.GT.N) GO TO 94
C   NODE GENERATED MUST BE DISTINCT FROM PRECEDING ROOT IN LIST
92    IF(I.EQ.LR) GO TO 93
      WRITE(1,500) I
      LR=1
      NROOT=NROOT-1
C     STOP AFTER NROOT NODES HAVE BEEN ENTERED
      IF(NROOT.LE.0) GO TO 95
93    IF(NR .GT.0) GO TO 5
      GO TO 94
C     WRITE TRAILER RECORD FOR ROOT NODE LIST
95    I=0
      WRITE(1,500) I
      GO TO 1
C     WRITE TRAILER RECORD FOR FILE
200   WRITE(1,500) N
      STOP
      END
```