Research Report

CCS 322

IMPROVING FLOW MANAGEMENT AND CONTROL VIA IMPROVING SHORTEST PATH ANALYSIS

by

Darwin Klingman*
John Mote**
David Whitman***

August 1978

* Professor of Operations Research and Computer Sciences, The University of Texas at Austin, BEB 608, Austin, TX 78712

** Systems Analyst, Center for Cybernetic Studies, The University of Texas at Austin, Austin, TX 78712

*** Systems Analyst, Center for Cybernetic Studies, The University of Texas at Austin, Austin, TX 78712

This research was supported by Department of Transportation Contract DOT-OS-70074.

CENTER FOR CYBERNETIC STUDIES
A. Charnes, Director
BEB 203E
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1821

ABSTRACT

Shortest and/or longest path analysis is a major analytical component of quantitative models for improving flow management and control. The importance of shortest path analysis in all phases of these operations has given rise to intensive research and software development for solving shortest path problems. However, many of the significant real-world applications for shortest path analysis involve problems that are far larger than the current software can handle on existing computers. This results from the fact that shortest path methodology and computer implementation has focused on procedures that keep all problem data in central memory, without using external storage devices. Accordingly, the purpose of this project is to conduct research on effective mathematical methods and computer implementations for solving large-scale shortest path problems. Specifically, we will explore the development of efficient solution procedures and computer programming software that exhibit both in-core and out-of-core storage and transfer capabilities. We expect our research to produce procedures capable of solving shortest path problems having millions of variables and several thousand equations on current computer hardware with reasonable solution times. Further, we anticipate these procedures can be used to greatly increase the solution capabilities of medium size computer systems which have a small amount of central computer memory.

1.0 INTRODUCTION

The primary purpose of this paper is to report the development and computational testing of in-core out-of-core shortest path algorithms and codes. This type of procedure is distinguished from previous codes in that not all of the data resides in central computer memory simultaneously; thus, they are referred to as in-core out-of-core procedures. The major advantages of such a code over in-core procedures are (1) they can solve problems which the latter cannot due to central memory limitations; (2) even for problems which the latter can solve the former requires less central memory which is often critical for fast job processing on multi-processing computer systems; and (3) codes developed for such procedures can also be used as an in-core code without significant performance degradation. Given the importance of shortest path analysis, it is surprising that the research and software development efforts have historically focused on solution procedures which keep all problem data in central computer memory [3, 4, 5, 6, 7, 9, 10, 11, 15, 16, 17].

The design of such procedures presents numerous computational difficulties in selecting the kinds of information and data structures to use in order to minimize central processing time as well as peripheral processing time. One of the most difficult problems in such a design is overcoming or accommodating the fact that the most efficient [4, 7] shortest path algorithms require random access of original problem data. These difficulties with current in-core procedures and the potential benefits of overcoming them have led distinguished researchers to stress the need for a major research effort to determine the most efficient in-core out-of-core algorithms for shortest path problems. Our study extends the excellent computational studies by Dial et al [4] and Gilsinn and Witzgall [7] on in-core algorithms.

This report specifically focuses on characterizing and comparing different

algorithms for calculating the shortest paths from one node to all other nodes in a directed network. The study discloses that alternative list structures and labeling methods exert a remarkably powerful influence on solution efficiency and solution capabilities.

In general, our conclusion is that the performance of in-core out-of-core shortest path algorithms is dominated by the amount of data transfer that must be performed. Consequently, if there is a sufficient amount of central memory available the best type of algorithm for in-core out-of-core processing is the label setting algorithm. Specifically, the Dijkstra address calculation sort algorithm (Dial [3], ACM Algorithm 360) suitably modified for in-core out-of-core processing, is the most efficient for city transit problems and the Dantzig address calculation sort algorithm is best for random networks. The identity of the best of these methods also depends upon the computer, its I/O subsystem and its billing routine. For example, an elementary label-correcting algorithm, which is one of the slowest algorithms in an in-core mode, appears to be the most efficient for city transit problems on the CDC 6600 at The University of Texas, but not on the DEC System 10. The study additionally shows that the performance of in-core out-of-core algorithms is not very sensitive to the range of the arc length coefficients and grid rectangularities. Finally, this study demonstrates the feasibility of solving shortest path problems with millions of variables or more modest size problems on medium-size computer systems which have a small amount of central computer memory.

Shortest and/or longest path analysis is a major analytical component of numerous quantitative transportation and communication models [2, 3, 6, 7, 9, 10, 11, 16, 17]. 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 a special case of shortest path problem, namely, an acyclic 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.

Due to the demonstrated value of shortest path analysis in diverse practical applications and the increased reliance on this type of analysis in recent years, the number and complexity of shortest path models has been steadily growing. This is true not only for problems that can be directly handled within a shortest (longest) path framework, but also for many problems with key components that can be treated within such a framework. As greater realism and more interacting considerations are included in the models for these problems, the size of these models also inevitably grows. Consequently, one is faced with a need to solve large scale problems that promises to increase as these models continue to become more realistic (and more useful).

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 [7] and the Dial et al [4] study, departing only to make distinctions and refinements not anticipated in the previous works.

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 [4, 7, 10], and the manner in which it is represented directly affects the performance of algorithms applied to the network.

The fastest in-core algorithms [4, 7, 16] store a network with $|N|$ nodes and $|A|$ arcs using a linked list structure. In this method, all of the arcs that begin at the same node are stored together and each 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.

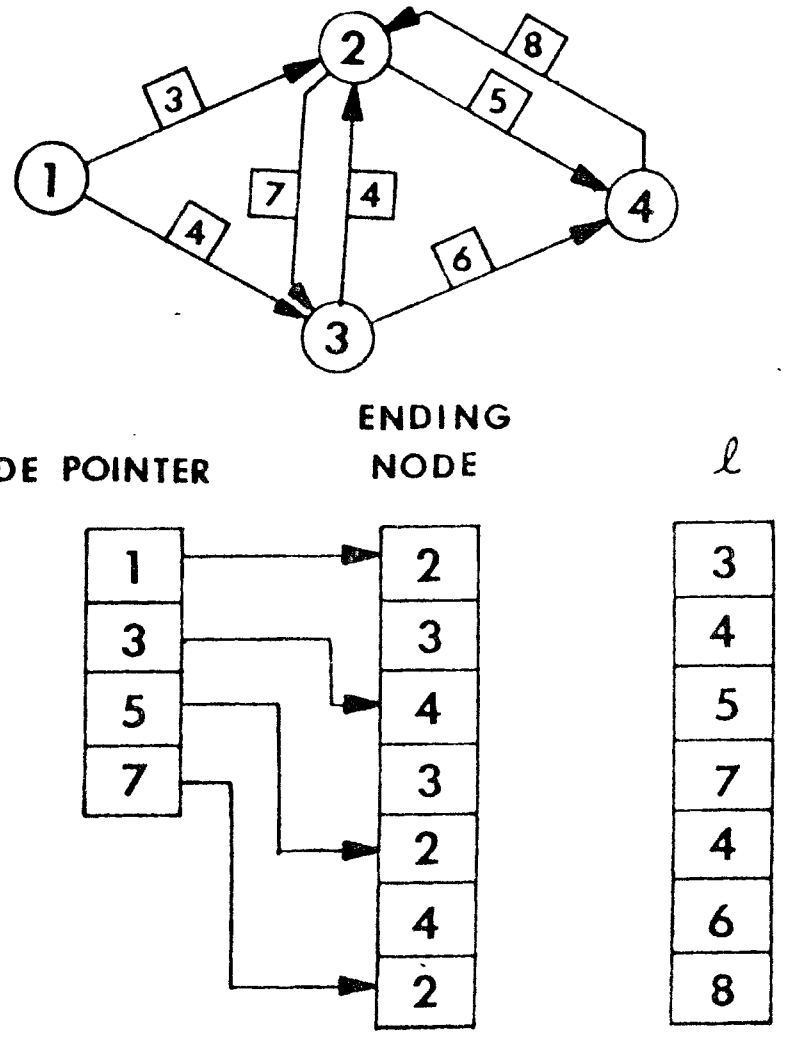
A rooted tree T is called a *minimum tree* or *shortest path tree* of a larger network G(N,A) if T contains all nodes of G accessible from r, and if for each node v in N_T, the unique path P from r to v is a shortest path from r to v in the network G.

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

Figure 1 - Sorted Forward Star Form



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 predecessor 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 over-estimate 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.

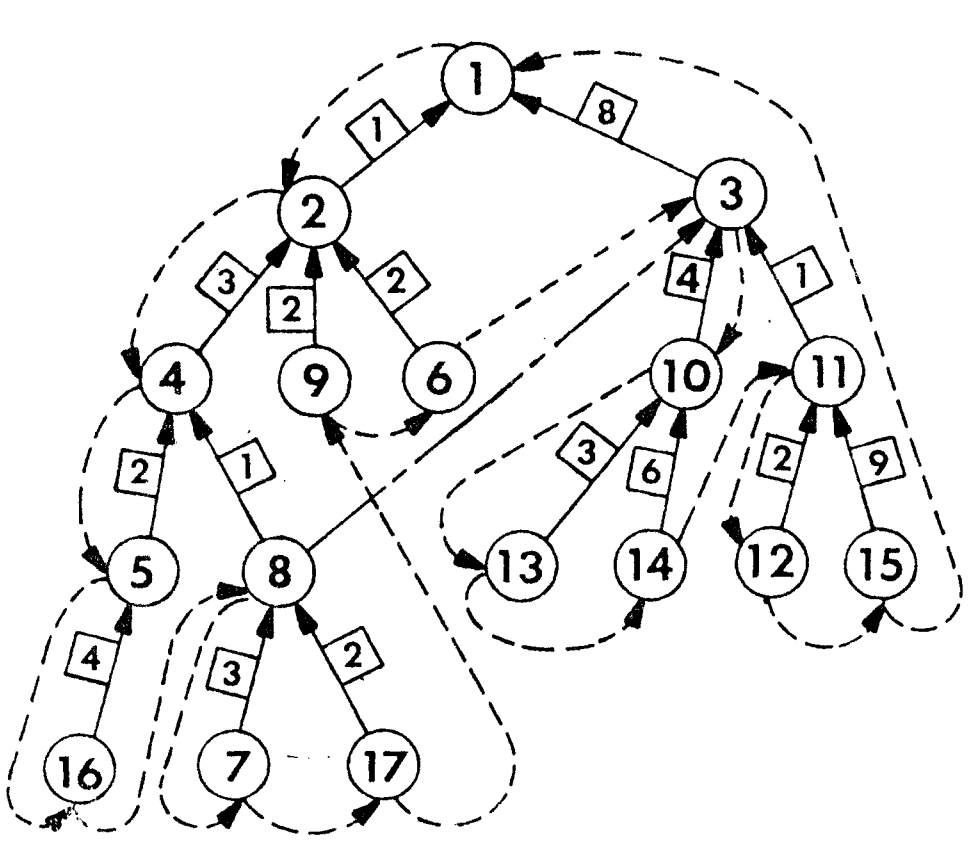
In Figure 2 the number in the square on each arc indicates the length of the arc. The entries in the d array identify the length of the unique path from the root to each node. The other functions illustrated in Figure 2 will now be described.

The first of these functions, the *thread* function [1, 8], is denoted by t(x). This function is a downward pointer through the tree. As illustrated in 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:

Figure 2 - Tree Labeling Techniques

Predecessor	**p (x)**
Node potential	**d (x)**
Thread	**t (x)**
Reverse thread	**rt (x)**
Depth	**dh(x)**



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

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 $r, t(r), \ldots, t^{k-1}(r)$ 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.)

Note that both the domain and the range of each of the above discrete functions are 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.

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 [4, 7]. 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 $u \in N_T$, $v \notin N_T$, 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.

General Label-Setting Method

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 $d(u) + \ell(u,v) = \underset{(p,q)\in S}{\text{minimum}} (d(p) + \ell(p,q))$. 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 non-negative 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).

General Label-Correcting Method

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 $(u,v) \in A$ such that $d(u) + \ell(u,v) < d(v)$. Otherwise, for such an arc, redefine

$$N_T := N_T \cup \{v\}$$

$$A_T := A_T - \{(s,v) \in A_T\} \cup \{(u,v)\}$$

$$p(v) := u$$

$$d(v) := d(u) + \ell(u,v)$$

3. Repeat Step 2.

4. Stop. $T(N_T,A_T)$ is a minimum tree and for each node $v \in N$, $d(v)$ 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 computers (a CDC 6600 and DEC System 10 (KI10)), the same FORTRAN compilers (MNF and FORTRAN-10) and all the codes were executed during time periods when the demand for computer use was comparable. Further, all of the codes were implemented by the same systems analysts and no attempt was made to exploit any of the unique hardware characteristics of the CDC 6600 or DEC 10.

Even with these safeguards, minor differences between the solution times of any two codes for a single test run of each must be regarded as being of questionable significance. For this reason, each test problem was solved three times (i.e., for three different roots) and the average solution time reported. Each code makes use of a real-time clock routine supplied by the computer vendors. Such routines 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.

5.1 Test Problems

The problem set consists of shortest path problems from two distinct

topological groups. One set consists of *city transit* networks. A city transit network consists of a *rectangular grid* network which has been augmented by non-grid arcs and a set of *terminal* nodes with terminal arcs. This class of problems was examined because they are one of the primary types of large scale shortest path problems. These problems were generated using a random problem generator. (See Table I for problem specifications.) The problem generator creates problems as follows:

(1) First, a p x q rectangular grid network is generated. A p x q grid problem 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 (henceforth called *grid* nodes) and 4pq - 2p - 2q grid arcs. The grid arc lengths are generated by using a uniform probability distribution (u.p.d.) and the user specified maximum grid arc length.

(2) Additional nodes are, then, randomly added to the network. These nodes, called *terminal* nodes, are distinct nodes, but their locations are considered to correspond to one of the grid nodes. Arcs are randomly generated out of terminal nodes to grid nodes using a u.p.d. The number of arcs for each terminal node is determined by using a u.p.d. and a user specified average number of arcs out of a terminal node. (See Table I.) The arc length on terminal arcs is randomly generated using a u.p.d., a user specified maximum city block length, and the rectangular distance (L_1 norm or city block norm) between two nodes. Note, however, that no arc length is allowed to exceed a user

TABLE I

CITY TRANSIT PROBLEM SPECIFICATIONS

Problem No.	Number of Terminals	Grid Size	Total No. of Nodes	Total No. of Arcs	Ave. Total No. of arcs Out of Grid Nodes	Ave. No. of Arcs Out of Terminal Nodes	Max. Grid Arc Lengths	Max Length Per City Block	Max Arc Length
1	10	50 x 50	2510	15,000	5	250	30	20	1000
2	10	50 x 50	2510	27,500	10	250	30	20	1000
3	10	25 x 100	2510	15,000	5	250	30	20	1000
4	10	25 x 100	2510	27,500	10	250	30	20	1000
5	20	50 x 100	5020	30,000	5	250	30	20	1000
6	20	50 x 100	5020	55,000	10	250	30	20	1000
7	20	40 x 125	5020	30,000	5	250	30	20	1000
8	20	40 x 125	5020	55,000	10	250	30	20	1000
9	30	75 x 100	7530	45,000	5	250	30	20	1000
10	30	75 x 100	7530	82,500	10	250	30	20	1000
11	30	50 x 150	7530	45,000	5	250	30	20	1000
12	30	50 x 150	7530	82,500	10	250	30	20	1000

specified maximum arc length. Thus, any arc length which exceeds this number is eliminated.

(3) Finally, the grid network is augmented by additional arcs, called *express* arcs. These arcs are randomly generated using a u.p.d. so that the grid nodes would have on the average a user specified average number of total arcs directed out of each node. (See Table I.) The express arc lengths are generated in the same fashion as terminal arc lengths.

Table II contains the CDC 6600 solution statistics and Table III contains the DEC 10 solution statistics on the city transit problems specified in Table I for the alternative implementations to be discussed. The statistics in these tables will be explained 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 20,000, 40,000, 60,000, 80,000, or 100,000 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 10,000. Again the arc lengths were randomly selected using a uniform probability distribution. Table IV contains the computational statistics on the random network problems. Because of computer budget limits, the random problems were not solved on the DEC 10.

TABLE II

MEAN SOLUTION TIMES IN SECONDS ON THE CDC 6600
FOR CITY TRANSIT PROBLEMS

Problem No.		1	2	3	4	5	6	7	8	9	10	11	12
Code													
C2	CP	1.1	CNR	1.1	CNR	CNR	CNR	CNR	CNR	CNR	CNR	CNR	CNR
	TM	1.1		1.1									
C2V10	CP	1.7	3.5	1.7	3.7	9.0	13.3	9.0	12.7	11.6	21.4	17.9	21.1
	TM	28.5	93.4	28.5	96.2	325.4	484.8	314.6	461.1	758.3	841.1	753.7	825.6
C2V13	CP	1.7	3.5	1.7	3.7	10.1	16.3	9.0	14.6	20.9	19.4	18.8	16.3
	TM	28.5	90.1	28.5	85.4	314.9	561.7	302.5	494.6	805.8	904.3	756.2	821.3
C5V3	CP	1.5	4.1	1.7	4.5	10.5	11.4	9.5	11.2	CNR	CNR	CNR	CNR
	TM	28.3	139.2	28.5	156.8	437.8	411.5	396.5	410.2				
C5V6	CP	1.6	3.8	1.7	4.3	CNR	CNR	CNR	CNR	CNR	CNR	CNR	CNR
	TM	28.6	119.1	28.5	135.9								
C5V7	CP	5.5	4.9	5.6	4.9	15.3	13.5	15.7	13.8	20.2	19.3	21.0	19.5
	TM	11.9	20.3	12.1	21.1	38.1	65.6	39.6	67.5	54.6	95.0	87.8	94.7
S1V1	CP	1.9	2.2	1.9	2.3	3.9	4.6	3.8	4.6	5.7	6.8	5.8	6.9
	TM	82.3	82.6	82.3	82.6	164.7	165.4	164.6	165.3	247.0	248.0	247.0	248.2
S1V2	CP	.6	.9	.6	.9	2.1	3.7	2.1	3.7	4.8	6.4	4.8	6.5
	TM	.9	3.3	1.0	3.4	59.9	109.8	59.7	109.5	184.5	215.4	184.8	215.3
S1V3	CP	.6	.9	.6	1.0	2.3	3.9	2.3	3.9	5.5	7.0	5.4	6.9
	TM	1.0	3.1	1.0	3.3	59.5	109.6	59.4	109.3	184.5	215.5	184.6	215.2
S2	CP	.5	CNR	DNR	CNR	CNR	CNR	CNR	CNR	CNR	CNR	CNR	CNR
	TM	.5											
S2A	CP	.8	CNR	DNR	CNR	CNR	CNR	CNR	CNR	CNR	CNR	CNR	CNR
	TM	.8											
S2V1	CP	.8	1.5	.8	1.7	DNR	DNR	DNR	DNR	DNR	DNR	DNR	DNR
	TM	3.2	28.1	3.3	29.2								
S2V2	CP	1.1	3.7	1.1	3.8	4.9	DNR	DNR	DNR	DNR	DNR	DNR	DNR
	TM	1.2	4.5	1.3	4.7	164.7							
S2V3	CP	.8	3.3	.8	3.4	DNR	DNR	DNR	DNR	DNR	DNR	DNR	DNR
	TM	2.6	23.0	2.5	23.5								
S2V4	CP	1.2	1.3	1.2	1.3	7.2	6.9	7.3	6.8	DNR	DNR	28.6	DNR
	TM	3.7	4.6	3.6	4.7	35.8	40.0	36.1	39.8			213.7	

DNR - did not run
CNR - could not run

TABLE III

SYSTEM BILLING TIME ON THE
DEC 10 FOR CITY TRANSIT PROBLEMS

Problem No.	5	6
Code \ Times	TM	TM
C2V10	204.1	316.9
C5V3	328.3	331.4
C5V7	104.3	201.7
S1V2	85.8	153.6
S2V4	177.4	211.2

TABLE IV

MEAN SOLUTION TIMES ON THE CDC 6600
FOR 1000 NODE RANDOM NETWORKS

Arc Length Range 1-200									
Code	C2V10	C2V13	C5V3	C5V6	C5V7	S1V2	S1V3	S2V6	S2V8
No. of Arcs	CP TM	CP TM	CP TM	CP TM	CP TM	CP TM	CP TM	CP TM	CP TM
20000	1.0 11.6	1.0 11.6	1.1 11.7	1.2 11.8	2.8 19.6	.6 3.2	.6 3.2	.6 4.5	.7 4.5
40000	2.7 47.1	2.7 40.3	2.8 60.7	2.9 59.5	5.4 40.9	1.2 12.2	1.2 12.2	.8 4.7	.8 4.7
60000	4.9 75.7	4.4 60.3	3.9 70.4	4.1 64.2	8.5 72.1	1.9 21.9	1.9 21.9	.7 4.5	.7 4.5
80000	5.0 70.6	4.8 66.1	3.9 60.4	3.9 56.4	8.3 71.4	2.3 25.8	2.4 25.9	.7 4.5	.7 4.5
100000	6.7 88.0	6.0 76.6	5.0 66.0	5.2 64.1	10.4 88.4	2.9 29.1	DNR	.7 4.6	.7 4.5
Arc Length Range 1-10000									
20000	DNR	DNR	DNR	DNR	DNR	.7 3.4	DNR	.7 3.9	.7 3.9
40000	"	"	"	"	"	1.4 15.8	DNR	.9 4.3	.9 4.3
60000	"	"	"	"	"	2.0 23.9	DNR	.7 3.9	.7 4.0
80000	"	"	"	"	"	2.5 27.5	DNR	.6 3.9	.6 3.9
100000	"	"	"	"	"	3.1 30.3	DNR	.7 3.9	.7 4.0

DNR - Did Not Run

5.2 Measurements and Evaluation of Code Efficiency

Charges for a computer job are *in some manner* based on a weighted aggregate of the central processing (CP) time, the amount of time (I/O) spent on transferring data to and from peripheral storage (often called I/O time), and the amount of central memory used (CR) multiplied by the time of central memory occupancy (CT). That is, a cost function of the form $a_1(CP) + a_2(I/O) + a_3(CR)(CT)$ is used, where a_1, a_2, and a_3 are constant unit charges. In our testing the total amount of central memory (CR) utilized by all codes was kept the same. On the CDC 6600 and DEC 10 this was 70,000 words (except for two codes which will be discussed subsequently).

Most billing systems utilize a function to compute I/O time which involves the number of peripheral accesses, the volume of data transferred DT, CT and CR. The I/O function is very important and plays a crucial role in the evaluation of in-core out-of-core shortest path algorithms. For our purposes, the most important aspect is the way in which DT is measured. Specifically, the user is billed for the amount of data that the operating system buffers into central memory which is not necessarily equal to the amount of data requested. In particular, on the CDC 6600 at the University of Texas the user is billed for Max {512 words, AR} per access, where AR is the amount of data requested. Thus, if the user requests 128 words per access, the system bills for 512 words per access. The most efficient algorithms in the shortest path literature require randomly accessing forward star data. Consequently when these algorithms are coded for in-core out-of-core processing, their I/O accesses are to obtain all arc data for

the forward star of node u (FS(u)). In city transit problems, most FS(u) consist of less than 40 words of data; thus, such algorithms incur large DT values because of the billing system.

The billing system on the DEC 10 is similar to that on the CDC 6600 except that the user is billed for the Max {128 words, AR}. Due to the peculiarity of billing systems, we computed a number of statistics for each code which we initially planned to report in order to provide the reader with a list of statistics which could be used to calculate the best algorithm to use for a given billing system. After talking to several commercial time sharing service and university computation centers we abandoned this approach. The primary reasons are (1) data on billing systems are not available for public release and (2) as with most multi-criteria problems the presentation of so many statistics becomes extremely difficult for a reader to assimilate without extensive explanation and analysis.

In Tables II, III, and IV, we report for the CDC 6600, CP time and TM time. TM time is the value computed by the billing system on the CDC 6600. The computer job statistics reported on the DEC 10 is simply TM time.

For in-core codes CP time is a reasonable statistic to use for evaluation purposes as long as the central memory requirements of the codes are clearly stated. In this case the amount of data transfer DT can be assumed to be equal for all in-core codes. The data in Tables II, III, and IV clearly demonstrate that CP time does not accurately measure the performance of an in-core out-of-core code because DT can vary substantially with changes in procedural rules without notably altering CP time.

The above discussion indicates that it is, indeed, a difficult task to determine the "best" in-core out-of-core algorithm; in fact, it may be impossible to conclude that a particular algorithm is best across different billing systems. Thus studies of this type run the risk of obtaining statistics which do not allow any discrimination to be drawn between algorithms. Fortunately, as will be seen subsequently, our test results on two very different computers and operating systems do provide some useful insights and conclusions. Nevertheless, it is important to recognize that our conclusions and findings could be substantially different for some computers and billing systems.

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 potential 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 litera-

ture regarding this search. The most rudimentary observation is that if the arcs are sequentially examined, it is not necessary to examine any arc $(u,v) \in A$ 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 $(u,v) \in FS(u)$ 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; however, to utilize this observation requires randomly accessing forward star data. As will be seen, the order in which forward stars of nodes are examined plays a major role in the efficiency of the algorithm because of the way billing systems calculate DT.

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.

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 pre-

viously, 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. It has been shown [4, 7, 16] that the way in which the sequence list is managed has major ramifications for the efficiency of in-core algorithms. We now describe in detail the codes whose solution statistics are indicated for city transit and random networks in Tables II, III, and IV.

The study [4] showed that the best in-core label-correcting algorithm uses a two way sequence list. The resulting code was called C2. Below, we describe how this algorithm and code are modified to operate as an in-core out-of-core procedure. In total, thirteen versions of this general algorithm were investigated. For brevity, only two versions are described.

Code C2V10 utilizes the predecessor and node potential functions and a two-way sequence list. The two-way sequence list is implemented as suggested by Pape [16]. That is, the sequence list is a node length array, called CL, indexed by node numbers, such that

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

Code C2V10, as will be true of some of the other codes to be discussed, keeps the original network stored in forward star form on a random access disk file as follows. All of the arcs in each forward star (i.e., their ending node and arc length) are recorded on sectors (physical record unit or block) of the disk file. The storage of arc data for each forward star is stored sequentially; however, the first arc in each forward star begins on a new sector. This allows the code to keep in central computer memory a node length pointer array, called SECTOR, which points to the sector where the forward star for node i begins. SECTOR is essentially equivalent to the NODE POINTER array in Figure 1.

The arc data are brought (paged) into central computer memory, a forward star at a time, in accordance with the management of the two way sequence list and stored in a buffer B_A. The size of this buffer is specified by the user, but must be as large as the largest forward star. Normally the buffer is large enough to hold several forward stars. Thus another node length pointer array, called the core pointer array CPP, is kept in central memory. This array is used to indicate if the forward star of a node is currently in-core (i.e., residence in B_A) and if it is, where its forward star begins in B_A. Another node length array is kept in central computer memory to indicate the number of arcs in each forward star, called NUMOUT. This information is used to determine if there is space in B_A to fit the next desired forward star and also to locate the arcs in a forward star once they are resident in B_A. In addition, the p and f functions and sequence list are kept in central computer memory. Code C2V10 then examines forward

stars in accordance with the management of the sequence list. It first checks to see if the forward star of the node to be examined is in-core; if not, its forward star is paged into central memory. Thus, the size of B_A directly affects solution time. In our testing the size of B_A is determined by the total amount of central memory (70K) to be used less the amount of central memory used by the node length arrays and the program itself. Thus the size of B_A differs by problem and code. Table V contains the buffer sizes used. The reason for this variation is that we adopted the philosophy that all codes should have the same amount of total central computer memory in order to eliminate this term from the billing function discussed in Section 5.2. Thus, as other implementations require more node length arrays to remain resident in central memory, the size of their B_A is reduced.

Code C2V13 is identical in structure to code C2V10 except that the management of the two way sequence list is different. In particular, the two way sequence list is partitioned into two node sets. One set consists of nodes on the sequence list whose forward stars are in B_A and the other set consists of the other nodes on the sequence list. These two sets are managed as before, i.e., nodes are added to both the front and back of each set and only deleted from the front. However, nodes are only removed from the set of nodes whose forward stars are not in B_A when the other set is empty.

The solution times in Table II indicate that the efficiency of these codes is essentially the same on transit problems. This result indicates

that B_A seldom contains information for nodes on the sequence list. Further observe that the TM time of these codes is extremely large. This is particularly interesting in view of the fact that C2 was found to be the best in-core code.

Problems 1 and 3 of Table II were run in-core on these codes (i.e., all problem data fit in the buffer B_A). The comparison of their solution times (Table II) to the original in-core version of this code, C2, shows that CP time increases by approximately 50 percent. This is due to the fact that C2V10 and C2V13 have to perform a number of node data transfers into B_A in order to get all problem data into central memory.

The solution times on random networks in Table IV indicate that C2V13 performs better than C2V10; thus, for the random problems it is useful to partition the sequence list. This is because the random problems have fewer nodes and the size of B_A is larger. These two factors substantially increase the likelihood that B_A contains data for nodes on the sequence list.

In [4] 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 $\Psi = dh(u) - dh(v) + 1$. The algorithm terminates when the sequence list is empty. We developed seven in-core out-of-core versions of this algorithm.

Three of these versions will be discussed. The first version, called C5V3, simply modifies C5 to keep the original network data on a disk file. It uses the same disk file problem structure as C2V10, keeping the same arrays CPP, NUMOUT, and SECTOR in central memory.

Code C5V6 differs from C5V3 in that it partitions the FIFO sequence list into two sets of nodes. Like C3V13, one set contains all nodes on the sequence list whose forward stars are in B_A. Each set is, then, maintained in FIFO fashion and nodes are always removed from the set of nodes in B_A first. The results in Table II indicate that the performance of these codes is very similar and further, that these codes cannot be used to solve problems which have a large number of nodes. This is a result of the fact that they utilize several node length arrays. The results in Table IV further show that C5V6 is better than C5V3 in terms of TM time, but not CP time, on random problems. Further investigation shows that partitioning the sequence list and using this to make maximum use of the data in B_A reduces DT. Additionally the results in Table IV indicate that as the number of arcs increases, the best algorithm in terms of TM time shifts from C2V13 to C5V6. Thus, when the number of arcs gets sufficiently large, it becomes worthwhile to keep correct node potentials.

Due to the large TM times of the previous label-correcting codes (which is primarily due to the fact that they are transferring small amounts of data per disk access but being billed for larger amounts), we implemented a less sophisticated label-correcting code C5V7. Like C5V3 and C5V6, it uses the functions p, d, t, rt, and dh; but, it does not use a sequence list and the arrays NUMOUT and CPP. Further, it uses a different disk file

problem structure. In particular, C5V7 reads several forward stars (called a page) into B_A at once. The size of a page is equal to the size of B_A. The original problem data is then stored on the disk file in pages. Given the page size, the code places in the first page complete forward stars starting at node 1 for as many consecutive nodes as will fit in the page. If the first page contains forward stars for nodes 1 to n_1, then the second page contains complete forward stars for consecutive nodes $n_1 + 1$ to $n_1 + n_2$. The next page contains complete forward stars for nodes $n_1 + n_2 + 1$ to $n_1 + n_2 + n_3$, etc. An array PP is kept in central memory which indicates which position of B_A (a page) contains the first arc out of each node. This array is also used to flag the nodes that need to be scanned. C5V7 sets the pointer in PP negative when a node's potential is changed and positive when this node's forward star is scanned. C5V7 sequentially examines pointers in PP until it finds a negative pointer. At this point, it brings this page into B_A and examines all forward stars in this page until PP contains no negative pointers for this page. Next the code continues to scan pointers in PP starting at the pointers for the page following the one just examined.

The primary advantages of this implementation are that it always performs large data transfers if the page size is large, and the central memory requirement on this code is smallest of all codes tested. In testing this code 70,000 words of central memory were not used. The size of B_A in our testing (see Table V) was large enough to overcome the billing system

TABLE V

SIZE OF BUFFER B_A IN WORDS ON CDC 6600 and DEC 10

Code	Transit with 2510 Nodes	Transit with 5020 Nodes	Transit with 7530 Nodes	Random with 1000 Nodes
C2V10	48,000	33,000	18,000	57,000
C2V13	48,000	33,000	13,000	57,000
C5V3	38,000	13,000	CNR (-12,000)	53,000
C5V6	34,000	7,000	CNR (-20,000)	51,000
C5V7	2,000	2,000	2,000	2,000

CNR - could not run

peculiarities but small enough in order to keep the number of forward stars on a page reasonably small. The disadvantage of this code is that it cannot make effective use of a sequence list.

Testing of analogous in-core codes show that this type of processing is substantially inferior to the in-core label-correcting codes C2 and C5. The results in Tables II and III show that this is not completely true for in-core out-of-core codes. These results indicate that the CP time of C5V7 is always inferior to the other label-correcting codes; but its TM time is substantially (in some cases 10 times) better than the other label-correcting codes. Note that this is only true for city transit problems. This follows from the fact that most of the nodes in these problems have very few arcs in their forward stars and thus the codes C2V10, C2V13, C5V3 and C5V6 are being billed heavily for data transfers. Note that the times on the DEC 10 are closer because its billing system penalizes small data transfers less heavily. Analysis of billing results showed that the label-correcting codes which used a sequence list were billed for as much as four times more DT then C5V7 on the city transit problems.

Basically, it appears that the best in-core out-of-core label-correcting code for city transit problems, grid problems, and any other network problems which have very few arcs in their forward stars is C5V7. The best in-core out-of-core label-correcting code for random network problems which have "many" arcs in their forward stars changes from C2V13 to C5V6 as the size of the forward stars increases. (See Table III.)

The city transit problems allow the reader to obtain some insight into the questions (1) how sensitive are solution times to the number of arcs since consecutive pairs of problems primarily differ only in their number of arcs, and (2) how sensitive are solution times to the rectangularity of the underlying grid problem since the pairs of every other problems (i.e., problems 1 and 3, 2 and 4, etc.) primarily differ only in their grid rectangularity. The solution statistics in Table II indicate that the algorithms are not sensitive to grid rectangularity but are affected by the number of arcs. In particular, it is interesting to observe that CP on C5V7 usually decreases as the number of arcs increase for a given rectangularity while TM strictly increases as the number of arcs increases. For the other label-correcting codes on both computers, both CP and TM tend to increase as the number of arcs increases. The statistics on the random problems in Table IV also show that CP and TM tends to increase as the number of arcs increases; but, it is noteworthy to observe the non-monotonic character of this relationship. (The random networks with an arc length range of 1-10000 were not solved by the label-correcting codes because preliminary testing, as well as previous in-core code testing, showed that the solution times of such codes are not sensitive to arc length ranges.)

7.0 IMPLEMENTATION TECHNIQUES FOR THE LABEL-SETTING METHOD

In this section several implementations of the general label-setting method are discussed. From an algorithmic viewpoint, the primary differences between these implementations are the way in which the minimum in Step 3 of the algorithm description is found and the handling of original

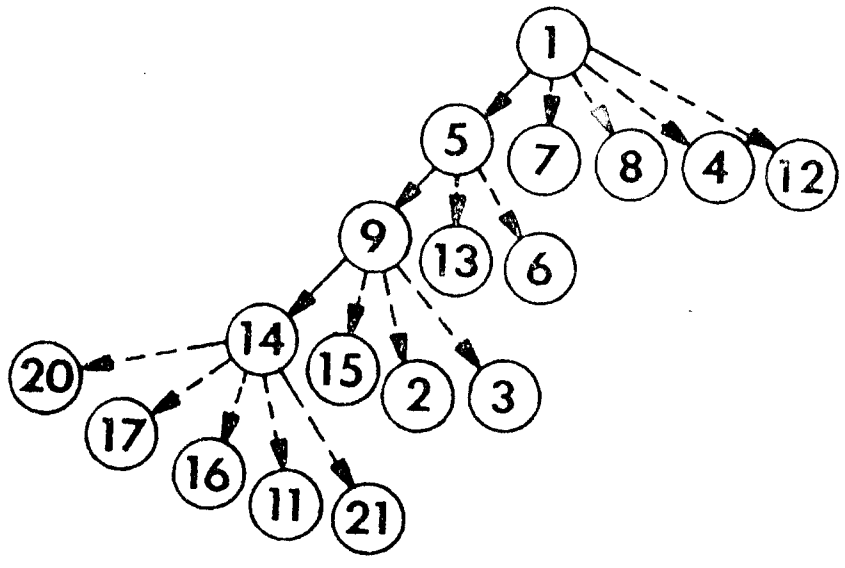
problem data. These implementations are evaluated by solving the same test problems using the same computers and compilers.

A naive 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-

Figure 3 - Label-Setting Iteration



setting stems from this property of the algorithm.

In the following subsections we discuss 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 assignment of one or more temporary node potentials to a node in N_E, (3) the representation of the original network on the external file, and (4) the buffering of arc data into central memory.

7.1 Dijkstra Address Calculation Sort

The first implementation to be discussed is the one originally developed by Dial [3]. In [4] the in-core code developed for this algorithm is called code S1. Several studies [9, 23] of shortest path algorithms have concluded that code S1 is the fastest in-core code, superior to all other label-setting and label-correcting implementations. However, the study [4] showed

that code S1 is dominated on grid networks by the in-core label-correcting code C2 and on random networks by an in-core label-setting code S2 which is based on an algorithm proposed by Dantzig [2].

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 $d(v) = \underset{u \in N_T}{\text{minimum}} (d(u) + \ell(u,v))$ 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 prodecessor.

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 $\ell_{max} = \underset{a \in A}{\text{maximum}} \ell(a)$. 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 $u \in N_T$ and thus for $t \in N_E$ $d(v) \leq d(t) \leq d(v) + \ell_{max}$. 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 $(\ell_{max} + 1)$.

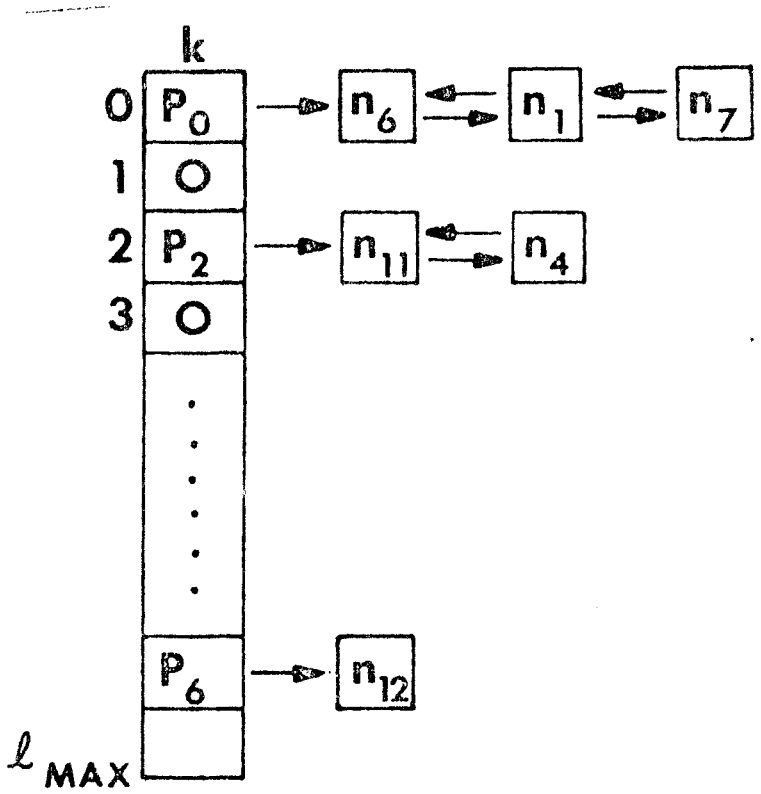
To find the minimum by this procedure, it is convenient to use a computer array k of size $\ell_{max} + 1$ where

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

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.

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.

Figure 4 - Address Calculation Sort



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 removed from the two-way chained list and sequentially 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 $(\ell_{max} + L)$. (If $d(q) = \infty$ then node v has never been added to k and thus no steps are 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 [3]). This algorithm, as noted earlier, was found by authors of several studies, to be the most efficient shortest path method for problems with nonnegative arc lengths.

Two attractive features of this algorithm, in addition to its efficiency for an in-core out-of-core implementation, 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 accessible from the root must have a minimum potential at some step, but never more than once, thus only the arcs starting at accessible nodes are examined at all. The major disadvantages of this algorithm are the computer memory required to store k and the random access required of arc data. Different ways of coping with these limitations are discussed subsequently.

Code S1 was converted to an in-core out-of-core code called code S1V1 by simply representing the original network data in a packed forward star format on an external disk file. In this format the elements of the ending node and ℓ arrays of Figure 1 are written pairwise on the disk file and storage of each forward star starts where the preceding one stops (i.e., forward stars do not necessarily begin on a new disk sector). An array, called IFSP, equivalent to the NODE POINTER array of Figure 1, is kept in central memory. An in-core buffer LTEMP, whose size is equal to the space required to store the largest forward star, is kept in central memory. As the algorithm picks a node whose forward star is to be scanned, code S1V1 pages its forward star into LTEMP and then scans it. As would be expected, the solution times of this code (shown in Table II) are heavily dependent on the number of nodes in the network. That is, this code requires one access of the problem file for

each node accessible from the root.

To reduce the number of accesses two other versions of S1, S1V2 and S1V3, were developed for city transit problems. These codes initially load B_A with the forward stars of the first M grid nodes. Since a forward star of a grid node has only a few arcs, B_A will often contain a large portion of the arc data for all grid nodes. When forward star data are required for a node not in B_A, its data are paged into LTEMP.

S1V2 and S1V3 differ only in their management of LTEMP. S1V2 simply manages LTEMP in a manner analogous to the way in which S1V1 manages it. S1V3, on the other hand, utilizes LTEMP as an auxiliary B_A array, i.e., whenever the forward star of a node not in B_A is required, then the code checks if its forward star is currently residing in LTEMP. If it is not, then its forward star is paged into LTEMP, replacing the previous contents of LTEMP.

This approach was designed to exploit the topology of transit networks. More precisely, the shortest arcs out of each grid node are generally connected to its grid node neighbors. Thus it is quite likely that one of its neighbors will be the next node whose forward star is to be scanned. Since LTEMP is filled with arc data on the disk file which is consecutive to the required data, it will normally contain arc data on the grid nodes preceding and following the grid node whose forward star is being scanned due to the way in which grid nodes are numbered. The likelihood is further increased by the fact that forward stars of grid nodes contain very few arcs; thus, LTEMP may contain the forward stars of several grid nodes. The advantage of this approach is that it should, on the average, require fewer

paging requests; i.e., S1V2 requires one paging request for each node not in B_A which is accessible from the root while S1V3 may require fewer requests. The statistics in Tables II and IV indicate that the performance of S1V2 and S1V3 are essentially the same in terms of TM time; thus the special handling of LTEMP does not appear to be worthwhile and the fact that S1V3 has larger CP times indicates that it is actually a disadvantage to handle LTEMP in this fashion.

In general the data in Table II indicates that it is extremely advantageous to place forward star data in B_A. Further, the TM times indicate that as the size of B_A is reduced the performance deteriorates to that of S1V1. (See Table VI for the buffer size on each problem.) The statistics in Table II also provide evidence that S1V2 and S1V3 are not sensitive to the rectangularity of the grid, but are sensitive to the number of arcs. The times further indicate that the solution times of these codes, like the label-correcting codes, are quite dependent on the number of nodes in the problem.

The solution times in Table IV indicate the surprising result that S1V2 and S1V3 are not sensitive to arc length ranges. The solution times in [4] show that the performance of the in-core code S1 is highly dependent on the maximum arc length.

7.2 Dantzig Address Calculation Sort

The study of [4] shows that a major time consuming task of the S1 type implementation involves inserting and deleting nodes in the two-way linked array when their node potentials are reduced. One way to reduce the effort of inserting and removing nodes on the two-way linked list of the address

TABLE VI

NUMBER OF FORWARD STARS INITIALLY IN B_A
FOR S1 CODES
CITY TRANSIT PROBLEMS

PROBLEM \ CODE	S1V1	S1V2 AND S1V3
1	-	2500
2	-	2438
3	-	2500
4	-	2436
5	-	3221
6	-	1709
7	-	3224
8	-	1719
9	-	1926
10	-	1009
11	-	1919
12	-	1016

RANDOM PROBLEMS

	1-200 ARC LENGTH RANGE	1-10000 ARC LENGTH RANGE
PROBLEM \ CODE	S1V2	S1V2
1	1000	1000
2	777	653
3	496	419
4	388	323
5	302	253

calculation sort 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*. Dantzig [2] 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. Indeed, for a "one-shot" solution of the shortest path problem, the effort devoted to organizing the data in a sorted forward star form may outweigh the advantages to be gained. However, it is important to recognize that the construction of a large transportation network, as must commonly be done for a large city, may cost 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 transportation

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 as long as possible.

The steps of the algorithm basically operate in the manner previously described for S1 except that: (A) The two-way linked list is replaced by a one-way linked list. (B) The forward star of each node u in N_T is scanned *until an improving node v is found*, whereupon v is placed on the linked

list with its predecessor u, and p(v) is set to u and d(v) is set to d(p(v)) + ℓ(p(v),v). (Node p(v) is not scanned again until the ordered pair (p(v),v) is removed from the linked list.) In some of the in-core out-of-core implementations to be discussed subsequently, this procedure is modified so that the forward star is *not* scanned until an improving node is found. Instead, the end node of the next arc in the forward star is added to the linked list whether it is improving or not. The list 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 B. Code S2 of [4] embodies this implementation.

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 keeps less information on k and potentially avoids adding dominated values to k.

Because of (1) it is not necessary for k to be empty; consequently, when all nodes are accessible from the root, it may *not* be 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.

Eight in-core out-of-core versions of S2 were implemented. Four of these versions were designed for city transit networks and the others for random networks.

The simpler versions of the city transit network codes represent the original network data in packed sorted forward star form on a disk file and use the IFSP array of code S1V1. These codes, S2V1 and S2V2, utilize two in-core arc buffer arrays, LTEMP and B_A. These codes initially place in B_A the first KSIZE arcs in each forward star of terminal nodes and the first ISIZE arcs in each forward star of grid nodes. When all of the arcs in B_A associated with some node u have been examined and the algorithm requires that additional arcs for node u be examined, new arcs for node u are added to B_A in the following manner. Using the IFSP array whose pointer is updated to indicate implicitly the disk sector containing the next arc of node u to be examined, the information in this disk sector and the immediately following disk sector are paged into LTEMP. (Thus the

size of LTEMP is equal to two disk sectors.) These arcs are then scanned in LTEMP until an improving arc is found for node u. The improving arc is then added to the linked list in the manner described earlier and the next ISIZE (KSIZE) arcs for node u are moved from LTEMP to B_A if node u is a grid (terminal) node.

The difference between S2V1 and S2V2 is the way in which arcs are examined in B_A when a node u (arc (p(u),u)) is removed from the linked list. S2V1, like S2, examines arcs in B_A until improving nodes (arcs) are found for node p(u) and node u. (See Step B of the algorithm description.) S2V2, on the other hand, simply places the ending node of the next arc for each node p(u) and u on the linked list in the manner described before.

In an in-core code it is better to continue examining arcs until an improving one is found because this avoids adding and removing items from the linked list and avoids the manipulation of the forward star pointer to find the next arc to examine. To demonstrate this fact we modified the in-core S2 to perform in both fasions. S2 in Table II continues examining arcs until an improving arc is found and S2A does not. The times show S2 is faster. (See Table VII for parameter values.)

In an in-core out-of-core code, however, we felt that it might be better to postpone the examination of arcs as long as possible because, as noted earlier, this algorithm may stop before examining all arcs and the examination of arcs until an improving arc is found, can require substantial computer time due to the fact that arcs may have to be paged in central memory. The statistics in Table II indicate that S2V2 is superior to S2V1 for exactly this reason, i.e., the statistics show that S2V2 performs fewer data transfers into central memory and fewer arcs are examined.

TABLE VII

VALUES OF USER SPECIFIED PARAMETERS USED FOR S2 CODES

TRANSIT PROBLEMS

PROBLEM 1	ISIZE	JSIZE	KSIZE	LTEMP SIZE
S2V1	10	-	70	128
S2V2	10	-	70	128
S2V3	10	6	45	512
S2V4	10	6	45	512
PROBLEM 2				
S2V1	10	-	70	128
S2V2	10	-	70	128
S2V3	10	6	45	512
S2V4	10	6	45	512
PROBLEM 5				
S2V2	3	-	90	128
S2V4	3	3	70	512
PROBLEM 6				
S2V4	3	6	70	512
PROBLEM 11				
S2V4	1	5	8	512

* ALL FOUR CODES SOLVED THE PROBLEM USING THE ARCS INITIALLY IN B_A; NO PAGING REQUESTS WERE REQUIRED.

RANDOM PROBLEMS

ON ALL 1-200 ARC LENGTH RANGE PROBLEMS
CODES: S2V6 AND S2V8
ISIZE: 31
JSIZE: 9
LTEMP SIZE: 128 (S2V2)
512 (S2V4)

ON ALL 1-10000 ARC LENGTH RANGE PROBLEMS
CODES: S2V6 AND S2V8
ISIZE: 26
JSIZE: 9
LTEMP SIZE: 128 (S2V2)
512 (S2V4)

The other city transit versions of code S2, S2V3 and S2V4, utilize a different representation of the original network data and modify the utilization of LTEMP. The codes represent the original problem in two disk files. Disk file 1 contains the entire network represented in a packed sorted forward star form at like codes S2V1 and S2V2. Disk file 2 contains only a portion of the original network. Specifically this file contains the arcs numbered from KSIZE + 1 to KSIZE + JSIZE in each forward star of terminal nodes and from ISIZE + 1 to ISIZE + JSIZE in each forward star of grid nodes. These codes, like S2V1 and S2V2, place the first KSIZE and ISIZE arcs of each forward star in B_A for terminal and grid nodes, respectively. S2V3 examines arcs in B_A like S2V1; i.e., it looks for an improving one. S2V4 examines arcs in B_A like S2V2.

S2V3 and S2V4, however, replenish arcs in B_A differently from S2V1 and S2V2. The size of the LTEMP array is a user specified parameter. This buffer is completely refilled with arc data each time arc data is paged into central memory. If data for node u must be paged into LTEMP, arc data is obtained from disk file 2 if it contains arcs for node u which have not been examined; otherwise, disk file 1 is used. In either case, the disk sector of the appropriate file containing the next arcs to be examined for node u is paged into LTEMP and, at the same time, as many of the immediately following disk sectors as will fit in LTEMP are paged into central memory. Arcs for node u are then examined and moved into B_A as in S2V1 and S2V2. Note that LTEMP should contain arc data for several different nodes, since the size of LTEMP is hopefully large and the amount of data for each node on disk file 2 is "small" (only JSIZE arcs per node). Because of this, *any* arc data in LTEMP which can be legitimately moved to B_A is moved. This

processing of LTEMP is the reason for creating the two disk files; i.e., disk file 2 allows LTEMP to contain data for more nodes.

The statistics in Table II demonstrate that S2V4 is superior to S2V3, thus showing that it is better not to search for an improving arc. Further, this data indicates that S2V4 is the best label-setting for solving small and medium size city transit problems (problems 1-8) on the CDC 6600. The performance of S2V4 deteriorates on the large problems (problems 9-12) due to the fact that all codes were not allowed to use more than 70,000 words of central memory. This forced us to set ISIZE equal to one on the large problems. However, on the smaller problems, when ISIZE was at least three, S2V4 performed better than all other in-core out-of-core codes. On the large city transit problems C5V7 performed best on the CDC 6600. (The results in Table III indicate that the above conclusions may not be valid on the DEC System 10 since it can be seen that S1V2 dominates S2V4 and C5V7 on the medium size problems. These apparent inconsistencies are discussed in the next section.) Again, the time in Table II indicates that the S2 codes are not affected by grid rectangularity but are affected by the number of arcs and nodes. The statistics, however, indicate that S2V2 and S2V4 are not affected as much as other codes by the number of arcs. This is primarily because the S2 type codes do not have to examine all of the arcs to terminate.

Four in-core out-of-core versions of S2 were developed for random problems--S2V5, S2V6, S2V7, and S2V8. On a conceptual level, codes S2V5, S2V6, S2V7, and S2V8 are equivalent to codes S2V1, S2V2, S2V3, and S2V4, respectively except that KSIZE = ISIZE; i.e., the same amount of information is kept in B_A for each node. On a coding level, however, these codes

are somewhat different.

Solution times are only reported for S2V6 and S2V8 because preliminary testing showed S2V5 and S2V7 to be substantially inferior; i.e., in an in-core out-of-core mode, it simply is not worthwhile to scan for improving arcs if it requires paging data into central memory. The statistics in Table IV show that S2V6 and S2V8 completely dominate all other codes and furthermore, large arc length ranges do not harm solution times. If anything, large arc ranges improved times. This was not true for the in-core code S2; but, it should be noted that S2 looks for improving arcs and this may not be advisable for large arc length ranges. Our statistics indicate that on the large arc length ranges S2V6 and S2V8 examined fewer arcs before terminating on many of these problems. More precisely, on the problems with a 1-200 arc length range, S2V4 examined 7639, 9316, 7863, 8018, and 7931 arcs for the problems with 20,000, 40,000, 60,000, 80,000, and 100,000 arcs, respectively. On the problem with a 1-10,000 arc length range, S2V4 examined 7549, 9809, 7929, 7218, and 7722 arcs for the problems with 20,000, 40,000, 60,000, 80,000, and 100,000 arcs, respectively. The most surprising part of these statistics is the fact that S2V8 examined so few arcs before terminating. In general, it appears that S2V8 is the best in-core out-of-core code for random networks; however, this may not be true for problems with a large number of nodes because in this case ISIZE will have to be small due to central memory limitations.

8.0 EVALUATION SUMMARY

8.1 Solution Statistics

The study demonstrates that conclusions based on the study of in-core

algorithms [4, 7] do not directly extend to in-core out-of-core algorithms. For example, the fastest in-core code for grid networks (city transit problems) is the label-correcting algorithm C2; however, in an in-core out-of-core mode, this algorithm's performance is dominated both by a simpler label-correcting algorithm C5V7 and by the label-setting algorithms S1V2 and S2V4.

In distinction to their in-core counterparts, the label-correcting and label-setting algorithms in an in-core out-of-core mode exhibit little dependence on grid rectangularity and arc length ranges. However, the study confirms that these algorithms are also sensitive to network topology.

The results in Table II indicate that code S2V4 is better than the other codes for solving small and medium size city transit problems. The code C5V7 is better on large city transit problems for the CDC 6600. The results in Table III lead us to believe that code S1V2 is better on large city transit problems on the DEC 10. These results can be explained by examining the billing systems and I/O subsystems of the two computers. For example, the CDC 6600 is capable of performing a minimum data transfer of 64 words but the billing system charges for a minimum of 512 words for each data transfer initiated. The DEC System 10, on the other hand, transfers a minimum of 128 words, but charges only for the amount transferred.

Looking at the results in Table II, the TM times reported for S1V2 include an inflated charge for data transfer (by a factor of approximately four) since this algorithm transferred the minimum amount of data required to capture one complete forward star, requiring in most cases only 128 words. If the TM times in Table II for S1V2 were adjusted to eliminate the "over charging" of the CDC 6600 billing system, the adjusted S1V2 times would ex-

hibit the same relationship to the times for the other algorithms as on the DEC System 10 (Table III). Consequently, it is our conclusion that the S1V2 algorithm is, in general, the best algorithm to use for city transit problems. Further, these results indicate that the user of in-core out-of-core algorithms must be aware of the I/O and billing system characteristics of the computer being used.

In general, it appears that the Dantzig address calculation sort algorithm, S2V8, is the best in-core out-of-core algorithm for random networks. However, a major limitation of the Dantzig algorithm in an in-core out-of-core mode is its reliance on sufficient computer memory to store a few (ISIZE $\geq$ 6) arcs per node in central memory. For this reason, it is conjectured that it is not a good algorithm for mini-computers because this would severely limit the size of problems which could be solved even in an in-core out-of-core mode. In this case, the algorithms S1V2 (which becomes S1V1 when $B_A = \emptyset$) or C5V7 would be preferable. Another limitation of the S2V4 and S2V8 algorithms is their reliance on a more complex data structure (two problem files) for storing the problem data.

8.2 Memory Requirements

Table VIII contains the computer array requirements of each code. It is interesting to note that the most efficient label-correcting algorithm in terms of TM time is also the most efficient in terms of computer memory utilization. The most efficient label-setting code in terms of computer memory is S1V1 and is only marginally worse than C5V7 if ℓ_{max} is not large.

TABLE VIII

STORAGE REQUIREMENTS

CODE	NODE LENGTH ARRAYS	BUFFERS SMALL	BUFFERS LARGE	BUFFERS OTHER
C2V10	6	-	B_A*	-
C2V13	6	-	B_A*	-
C5V3	10	-	B_A*	-
C5V6	11	-	B_A*	-
C5V7	6	B_A**	-	-
S1V1	6	LTEMP**	-	$(\ell_{MAX} + 1)$
S1V2 S1V3	6	LTEMP**	B_A*	$(\ell_{MAX} + 1)$
S2V1 S2V2	7	LTEMP**	2N(ISIZE)*	$(\ell_{MAX} + 1)$
S2V3 S4V4	7	LTEMP***	2N(ISIZE)*	$(\ell_{MAX} + 1)$

WHERE N IS EQUAL TO THE NUMBER OF NODES.

* THE SIZE OF THIS BUFFER MUST BE LARGE ENOUGH TO STORE THE LARGEST FORWARD STAR BUT SHOULD BE MADE AS LARGE AS POSSIBLE TO IMPROVE EFFICIENCY.

** THE SIZE OF THIS BUFFER MUST BE LARGE ENOUGH TO STORE THE LARGEST FORWARD STAR, BUT NEED NOT BE MADE LARGE.

*** THE SIZE OF THIS BUFFER MUST BE LARGE ENOUGH TO STORE THE LARGEST FORWARD STAR BUT SHOULD BE MADE "SOMEWHAT" LARGER.

8.3 Limitations

Some apparent limitations of this study in the selection of test problems and the extent of testing performed should be clarified. Due to a lack of available alternatives only randomly generated problems were examined. In addition, no pure grid networks were tested because of computer budget limitations and it appeared that they would not exhibit any significant differences from the city transit networks. Budget restrictions also limited our testing on the DEC System 10 and on random networks.

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, The University of Texas at Austin, 1976.

2. G. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey, 1963.

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

4. R. Dial, F. Glover, D. Karney, and D. Klingman, "A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees," Research Report CCS 291, Center for Cybernetic Studies, The University of Texas at Austin, 1977. To appear in *Networks*.

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

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

7. 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.

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

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

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

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

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

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

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

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

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

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