9-11-1991

Optimization algorithms for shortest path analysis

Susan M. Hojnacki

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Optimization Algorithms for Shortest Path Analysis

by

Susan M. Hojnacki

A Thesis, submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering

Approved by:

____________________________
Tony H. Chang, Ph.D. (Committee Chairman)
Department of Computer Engineering

____________________________
Ronald G. Matteson, Ph.D. (Committee Member)
Department of Computer Engineering

____________________________
James E. Heliotis, Ph.D. (Committee Member)
School of Computer Science and Information Technology

____________________________
Roy S. Czernikowski, Ph.D. (Department Head)
Department of Computer Engineering

September 11, 1991

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York
# TABLE OF CONTENTS

TABLE OF FIGURES ......................................................................... iii
TABLE OF TABLES ........................................................................ iv
ABSTRACT ..................................................................................... v

1 Networks and Graph Theory .......................................................... 1
  1.1 Introduction and Background .................................................. 1
  1.2 Classification of Digraphs ....................................................... 5

2 Network Matrices and Data Structures ........................................... 9
  2.1 Matrix Representations of Networks ....................................... 9
  2.2 Other Network Representations - Data Structures ................. 13
  2.3 Transitive Closure ............................................................... 16

3 Description of Algorithms ............................................................ 18
  3.1 Dijkstra’s Shortest Path Algorithm ....................................... 18
  3.2 Bellman-Ford Shortest Path Algorithm .................................. 23
  3.3 Floyd-Warshall Algorithm ................................................... 26

4 Analysis and Results ................................................................... 30
  4.1 Dijkstra’s Shortest Path Algorithm ....................................... 31
  4.2 Bellman-Ford Shortest Path Algorithm .................................. 34
  4.3 Floyd-Warshall Algorithm ................................................... 35
  4.4 Time Trials ........................................................................... 37
    4.4.1 Sparse Network .............................................................. 37
    4.4.2 Complete Network ........................................................ 38

5 Conclusions ................................................................................ 41
REFERENCES .................................................................................................................. 44

APPENDIX A: PROGRAM LISTINGS ............................................................................. 46
  algdl.c ................................................................. 46
  algd2.c ................................................................. 50
  algd3.c ................................................................. 55
  algd4.c ................................................................. 59
  algbf1.c ................................................................. 63
  algbf2.c ................................................................. 68
  algfw.c ................................................................. 73
  spath.h ................................................................. 76

APPENDIX B: Program Instructions and Examples ..................................................... 77
  Example #1: All Arcs Have Positive Weights ....................................................... 77
  Example #2: Some Arcs With Negative Weights .................................................. 80
  Example #3: A Network With A Negative Cycle ............................................... 82
  Dijkstra’s Shortest Path Algorithm .................................................................. 83
    Instructions for Running algdl.c ................................................................. 83
    Instructions for Running algd2.c ................................................................. 87
    Instructions for Running algd3.c ................................................................. 91
    Instructions for Running algd4.c ................................................................. 94
  Bellman-Ford Algorithm ................................................................................. 97
    Instructions for Running algbf1.c ................................................................. 97
    Instructions for Running algbf2.c ................................................................. 102
  Floyd-Warshall Algorithm ............................................................................. 115
    Instructions for Running algfw.c .................................................................. 115

APPENDIX C: Evaluation of Simulation Software ..................................................... 127

INDEX .......................................................................................................................... 128
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Graph Representation of a Network</td>
<td>2</td>
</tr>
<tr>
<td>Figure 1.2a</td>
<td>Example of a self loop</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.2b</td>
<td>Example of parallel arcs</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.2c</td>
<td>Example of a cycle</td>
<td>3</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Example of a Reflexive Digraph</td>
<td>5</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>Example of a Symmetric Digraph</td>
<td>5</td>
</tr>
<tr>
<td>Figure 1.5</td>
<td>Example of a Transitive Digraph</td>
<td>6</td>
</tr>
<tr>
<td>Figure 1.6</td>
<td>Example of an Irreflexive Digraph</td>
<td>6</td>
</tr>
<tr>
<td>Figure 1.7</td>
<td>Example of an Asymmetric Digraph</td>
<td>7</td>
</tr>
<tr>
<td>Figure 1.8</td>
<td>Example of an Antisymmetric Digraph</td>
<td>7</td>
</tr>
<tr>
<td>Figure 1.9</td>
<td>Example of an Equivalence Digraph</td>
<td>8</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>Node-Arc Incidence Matrix</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Digraph With Weights</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Weight Matrix</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Adjacency Matrix</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.5</td>
<td>Linked Adjacency List Representation</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2.6</td>
<td>Forward Star Representation</td>
<td>16</td>
</tr>
<tr>
<td>Figure B.1</td>
<td>Digraph for Example #1</td>
<td>78</td>
</tr>
<tr>
<td>Figure B.2</td>
<td>Digraph for Example #2</td>
<td>81</td>
</tr>
<tr>
<td>Figure B.3</td>
<td>Digraph for Example #3</td>
<td>83</td>
</tr>
</tbody>
</table>
TABLE OF TABLES

Table 4-1  Comparison of Dijkstra and Bellman-Ford Algorithms for a Sparse Network ......................................................... 37
Table 4-2  Comparison of Dijkstra, Bellman-Ford, and Floyd-Warshall Algorithms for a Sparse Network ............................................................ 38
Table 4-3  Comparison of Dijkstra and Bellman-Ford Algorithms for a Complete Network ................................................................. 39
Table 4-4  Comparison of Dijkstra, Bellman-Ford, and Floyd-Warshall Algorithms for a Complete Network ................................................................. 39
ABSTRACT

Discrete optimization algorithms exist for analysis of network optimization of flow problems. Computer programs written from these algorithms can be used for local area network analysis of point-to-point computer networks, transportation networks, resource allocation, distribution, and production scheduling. One aspect of a network that can be optimized using discrete optimization algorithms is the length of the path that data will take when traveling through the network. One node in the network signifies the source node and a second node is the sink or destination. The object is to find the shortest path between the two nodes.

The definition of shortest path depends on the quantity analyzed in the network. "Shortest path" can represent the fastest path, most cost-efficient path, most fuel-efficient path, etc. Also, different levels of computation may be required. It may be necessary to find the shortest path between two nodes in a network, the shortest path between a source node and all other nodes in a network, or the shortest path between all pairs of nodes in a network.

The complexity, performance, and results of different optimization methods can be compared using a series of network models. A comparison of the algorithms researched and results of the computer analysis will be shown.
Chapter 1

Networks and Graph Theory

1.1 Introduction and Background

A network consists of nodes (vertices) and arcs (edges). The arcs represent means of transport in the network and the nodes are locations connected by the arcs. Some examples of networks include: a local area network, a telephone communications network, and a transportation network made up of roads and highways. Networks can be modeled and analyzed using graphs and graph theory. This chapter presents some of the basics of graph theory necessary to understand the discrete optimization algorithms that will be presented.

The arcs in a network can have a direction associated with them, in which case they are called directed arcs. A weight or capacity is associated with each arc. The weight can be represented as a lower or upper bound. Distance, cost, penalties, time, or any other quantity to be minimized can be used for arc weights in a network. Some networks will have negative arc weights. Negative arc weights can represent costs in a network, while positive ones can represent profits.

A network can be described by a labeled drawing or a matrix. A labeled drawing, also called a graph, contains circles and lines segments. The circles represent nodes in the network, and the lines segments represent network arcs. If the arcs are directed arcs,
then the line segments have directional arrows and the graph is called a directed graph, or digraph. Figure 1.1 is a digraph with six nodes and nine arcs. The nodes are labeled from 0 to 5, and the arcs are labeled from 0 to 8.

A directed arc from node \( n_1 \) to node \( n_2 \) is represented by the ordered pair \( (n_1, n_2) \). Arc number 1 in Figure 1.1 can be represented by the ordered pair \( (0,5) \). Arcs can also be labeled with numbers or letters. A graph in which none of the arcs have a direction associated with them is called an undirected graph.

The number of arcs leaving a node is called the out-degree or out-valence. The number of arcs entering a node is called the in-degree or in-valence. The sum of all in-degrees should equal the sum of all out-degrees in a network.
A self loop is an arc whose source and destination nodes are the same. In Figure 1.2a, node 1 has a self loop. Parallel arcs occur when two arcs have the same direction and also have the same source and destination node. In Figure 1.2b, arcs 1 and 2 are parallel arcs. A path through a network is a series of successive arcs in which no node is repeated. A cycle, or directed circuit, is a path through a directed graph that starts and ends at the same node. A negative cycle is a cycle in which the sum of the arc weights is negative. A network that does not contain any cycles is called acyclic. In Figure 1.2c, the sequence of arcs (1, 2, 4) is a cycle.

![Diagram of self loop, parallel arcs, and cycle](image)

**Figure 1.2**
Example of (a) a self loop, (b) parallel arcs, and (c) a cycle

A simple directed graph is a graph that does not contain any self loops or parallel arcs. All of the algorithms researched operate on simple directed graphs only. A graph that is not simple can be made simple by replacing each set of parallel arcs with the shortest arc of the set (arc with the smallest weight). Self loops can be removed from the graph for the analysis. In addition, if the graph is not a digraph, it can be made a digraph by Chapter 1
replacing each undirected arc with two directed arcs (in opposite directions) with equal weights. However, if an undirected arc has a negative weight associated with it, replacing the undirected arc with two directed arcs will create a negative cycle.

The shortest path problem is an optimization problem. An optimization problem may have many solutions, but only one solution (or a set of solutions) is the best solution. This solution is called optimal (although there may be more than one optimal solution). The shortest path problem is defined as finding the minimum weight path from a source node to a destination node, for any given node pair in the network, based on arc weights. The shortest path does not contain any repeated nodes. There can be more than one optimal solution to the shortest path problem if there is more than one path from source to destination with the same minimum path weight.

There are three cases of shortest path problems:

1. All arc weights are non-negative.
2. Some arc weights are negative, but there are no negative cycles.
3. There is at least one negative cycle in the network.

The third case cannot be solved. However, there exist algorithms that can find the negative cycles in the network.
1.2 Classification of Digraphs

1. **simple**
   There are no self loops or parallel arcs. The digraph in Figure 1.1 is a simple digraph.

2. **reflexive**
   Every node in the network has a self loop.

   ![Figure 1.3](image)
   Example of a Reflexive Digraph

3. **symmetric**
   For every arc in one direction between two nodes, there is also an arc in the opposite direction between the same two nodes.

   ![Figure 1.4](image)
   Example of a Symmetric Digraph
4. transitive If there is an arc from node $a_i$ to node $a_j$, and an arc from node $a_j$ to node $a_k$, then there is an arc from node $a_i$ to node $a_k$.

Figure 1.5
Example of a Transitive Digraph

5. irreflexive There are no self loops in the network.

Figure 1.6
Example of an Irreflexive Digraph
6. asymmetric  There is at most one directed arc between a pair of nodes, no two nodes have arcs in both directions. Self loops are not permitted.

7. antisymmetric  There is at most one directed arc between a pair of nodes.

Figure 1.7
Example of an Asymmetric Digraph

Figure 1.8
Example of an Antisymmetric Digraph
8. equivalence  A digraph that is reflexive, symmetric, and transitive.

Figure 1.9
Example of an Equivalence Digraph
Chapter 2

Network Matrices and Data Structures

2.1 Matrix Representations of Networks

One way to model a network is with a matrix. This representation is ideal for performing calculations on network information. Given a network with n nodes numbered 0 to n - 1, and m arcs numbered 0 to m - 1, the first matrix depiction examined can be defined as:

\[
a_{ij} = \begin{cases} 
  +1 & \text{if arc } j \text{ is directed away from node } i \\
  -1 & \text{if arc } j \text{ is directed towards node } i \\
  0 & \text{if arc does not exist}
\end{cases}
\]

for \( i = 0,1,\ldots, n-1 \)

\( j = 0,1,\ldots, m-1 \)

where \( a_{ij} \) is the element in the \( i^{th} \) row and \( j^{th} \) column of matrix A. The resulting matrix will be \( i \) rows by \( j \) columns, \( i \) corresponding to nodes, and \( j \) corresponding to arcs. This matrix is called the node-arc incidence matrix\(^1\), also called the vertex-arc incidence matrix\(^2\).


A node-arc incidence matrix for the digraph in Figure 1.1 is shown below.

\[
\begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 1 & 1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & -1 \\
0 & -1 & 0 & -1 & 0 & 0 & 0 & 1 & 1 \\
\end{bmatrix}
\]

Figure 2.1
Node-Arc Incidence Matrix

It can be seen that each column of the node-arc incidence matrix will have exactly two non-zero entries, a single +1 entry, and a single -1 entry, the reason being that each arc can enter only one node and can leave only one node.

Another matrix representation of a network is the weight matrix representation. In this representation, a network with \( n \) nodes could be represented by an \( n \) by \( n \) matrix, called \( W \), defined by:

\[
w_{ij} = \begin{cases} 
\text{weight} & \text{if arc } (i,j) \text{ exists} \\
\infty & \text{if arc } (i,j) \text{ does not exist} 
\end{cases}
\]

for \( i = 0,1,\ldots, n-1 \)

\( j = 0,1,\ldots, n-1 \)
Figure 2.2 shows the results of adding arbitrary weights (for purpose of example) to the digraph in Figure 1.1. The weights are displayed in boxes, next to the arc they describe.

The entries on the diagonal of the matrix are set to 0, making the weight matrix irreflexive. Parallel arcs and self loops cannot be represented in a weight matrix. The weight matrix for the digraph in Figure 2.2 is as follows:

\[
\begin{bmatrix}
0 & 14 & \infty & \infty & \infty & 7 \\
\infty & 0 & 18 & \infty & \infty & 25 \\
\infty & \infty & 0 & 9 & 11 & \infty \\
\infty & \infty & \infty & 0 & \infty & \infty \\
\infty & \infty & \infty & 15 & 0 & \infty \\
\infty & \infty & 33 & \infty & 20 & 0 \\
\end{bmatrix}
\]
A third matrix representation of a digraph is the *adjacency matrix* representation. This representation is similar to the weight matrix, but instead of describing weights in the matrix, an arc's presence between two nodes is represented. Parallel arcs cannot be represented with an adjacency matrix.

The adjacency matrix, \( X = [x_{ij}] \), where

\[
x_{ij} = \begin{cases} 
1 & \text{if there is an arc directed from node } i \text{ to node } j \\
0 & \text{if an arc does not exist between node } i \text{ and node } j 
\end{cases}
\]

for \( i = 0,1,\ldots, n-1 \)

\( j = 0,1,\ldots, n-1 \)

Self loops can be represented in an adjacency matrix by a 1 at \( x_{ii} \) (meaning node \( i \) has a self loop). There are six nodes in Figure 2.2, so the adjacency matrix for Figure 2.2 will be a 6 by 6 matrix. The adjacency matrix for Figure 2.2 is as follows:

\[
\begin{bmatrix}
0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 \\
\end{bmatrix}
\]

Figure 2.4
Adjacency Matrix
2.2 Other Network Representations - Data Structures

Let \( m \) represent the number of arcs in a network, and let \( n \) represent the number of nodes. A network where the number of \( m \) arcs is much less than \( n \cdot (n - 1) \) (the total of all possible edges) is called a sparse network. Using a matrix representation for a sparse directed network would not be efficient because it would take up \( n^2 \) words of storage, few of which would be used to store entries corresponding to existing arcs. A more practical way to represent a sparse network would be to use a data structure that lists the arcs of the network. One such representation is called the adjacency list or list of edges. This representation consists of three one-dimensional arrays, origin stores the originating node of the arc, target stores the destination node of the arc, and weight stores the length of the arc. Arc \( i \) starts at \( \text{origin}_i \), ends at \( \text{target}_i \), and has a weight of \( \text{weight}_i \). The amount of computer storage required for this network representation is \( 3m \) words.

For example, the network in Figure 2.2 is a sparse network. There are 9 arcs and 6 nodes. The following arrays describe the network displayed in Figure 2.2:

\[
\begin{align*}
\text{origin} &= (0, 0, 1, 1, 2, 2, 4, 5, 5) \\
\text{target} &= (1, 5, 2, 5, 3, 4, 3, 2, 4) \\
\text{weight} &= (14, 7, 18, 25, 9, 11, 15, 33, 20)
\end{align*}
\]

The amount of storage required is 27 words, compared to the matrix representation for this network which would require 36 words of storage. The storage savings can be large depending on the size of the network and its sparseness.
A second data structure for network representation is called the *linked adjacency list*\(^3\). There are \(n\) linked lists, one for each node in the network. All arcs radiating from the \(i\)th node are linked together. Each element in the linked list defines an arc in the network and is a structure containing the node number of the target of the arc, and the weight of that arc. This type of data structure requires \(n + 3m\) words of storage.

The linked adjacency list for the network in Figure 2.2 is shown in Figure 2.5. The amount of storage required for this network is 33 words.

---

Another network representation is called the forward star\(^4\). This data structure consists of three arrays. The first array contains a pointer or index for each node in the network. This pointer points to a location in the second and third arrays. The second and third arrays describe the arcs. The second array contains the arcs’ target node number, and the third array contains the weight of that arc. The information in the second and third arrays for the arcs of the ith node are stored sequentially. The forward star of node i consists of all the arcs radiating from node i. The amount of storage required is \(n + 1 + 2m\) words.

For example, the forward star data structure of the network shown in Figure 2.2 is displayed in Figure 2.6. Node 6 does not exist. Its pointer points to the next available slot in the target and weight arrays. This is necessary in order to compute the first and last index into the target and weight arrays for node 5. The amount of storage required for this network is 25 words.

The linked adjacency list and the forward star data structures are convenient for programming purposes since they implicitly define the originating node of each arc. The linked adjacency list is especially suitable for applications where nodes are to be added and deleted from the network.

2.3 Transitive Closure$^5$

$\mathbf{G}$ is an $n$-node digraph and $\mathbf{M}$ is the weight matrix for $\mathbf{G}$. To compute the transitive closure of graph $\mathbf{G}$, construct another $n$-node digraph by adding arcs to $\mathbf{G}$ as follows: add an arc $(i,j)$ directed from node $i$ to node $j$ if and only if there is a directed path (of

any length 2,3,..., n-1) from i to j in G. Examine all arcs incident to a node before examining the next node. While examining a node, the arcs incident to the node can be chosen in any order. The resulting digraph is called the transitive closure of G. The weight matrix, M, describing G is transformed by overwriting directly on M (the intermediate matrices are not saved).\textsuperscript{6}

Similar notation will be used to describe each of the algorithms studied. The total number of nodes in the network is equal to \( n \). The nodes are numbered from 0 to \( n - 1 \). The number of arcs in the network is equal to \( m \) and they are numbered from 0 to \( m - 1 \). In a weight matrix, the weight of a non-existent arc is represented by \text{INFINITY}. The source node is called \( s \) and the destination node is called \( d \). Also, for any real number \( r \), \( r + \text{INFINITY} = \text{INFINITY} \).

### 3.1 Dijkstra's Shortest Path Algorithm

An algorithm which computes the shortest path from one node in a network to another node in a network is \textit{Dijkstra's Shortest Path Algorithm}. This algorithm is considered a "greedy algorithm" because it always picks the choice that looks the best at the moment. Greedy algorithms do not \textit{always} result in optimal solutions, but this algorithm is an exception.\(^7\)

Dijkstra's algorithm uses a label-setting method. Each node in the network has a label, permanent or temporary, and a weighted value which represents the distance, cost, or some other value to be optimized from that node to the source node. If a node has been permanently labeled, then the node's weighted value represents the shortest weight (distance) from that node to the source node. If a node has a temporary label, then its' weighted value represents an upper bound on the weight (distance) between that node and the source node.

For this algorithm to work correctly, all the arc lengths in the net must be positive. This algorithm uses n - 1 iterations for the worst case (when the destination node is the farthest node from the source node).

Initialization:
Label all nodes in the network with distance = INFINITY (some large number for practical purposes). Set distance = 0 for the source node (distance from the source node to itself is zero). The source node is the only permanently labeled node at this point.

Iteration:
Label each immediate successor node (i) of the source node (s) with a temporary label equal to the weight of the arc from s to i. The node with the smallest temporary label among all immediate successors is the node closest to s (has the
smallest weight). Ties can be broken arbitrarily. Make this nodes' label permanent and call it \textit{recent}. It is the most recent node to be permanently labeled, and the closest node to the source node. Next, examine all the immediate successors of node \textit{recent} and shorten their temporary labels if the path from the source node to any of them is shorter by going through \textit{recent} than by not going through \textit{recent}. Then, from all the temporarily labeled nodes, pick the one with the smallest label. Make its label permanent and call \textit{it recent}. Node \textit{recent} is the second closest node to the source node.

Continue looping, decreasing the values of the temporary labels when possible by choosing a shorter path through the most recently permanently labeled node. Then select the node with the smallest temporary label and make it permanent. Terminate the loop when the destination node, \textit{d}, gets a permanent label.

The shortest path from node \textit{s} to node \textit{d} is identified by tracing the permanently labeled nodes between node \textit{s} and node \textit{d}.

The network representation used in this algorithm is a weight matrix. The C program written to implement this algorithm is called \texttt{algdl.c}. The program listing is displayed in Appendix A. Instructions for running the program and analyzing the results are in Appendix B. An example is also shown in Appendix B.
The network is made to look like it contains all arcs possible by assigning arc lengths of INFINITY to non-existent arcs. Therefore, the time it takes to find the shortest path depends on the number of nodes, and not the number of arcs in the network. The time that can be expected for a complete network is \( O(n^2) \).

If the network is sparse, as it is in most practical applications, the computation time for this algorithm remains the same, \( O(n^2) \). Therefore, this algorithm is not efficient for finding a shortest path from source node to destination node in a sparse network when a weight matrix is used to describe the network. The computation time for finding the shortest path in a sparse network can be reduced by using a different matrix representation (or data structure). A linked adjacency list representation can be used. Another way to shorten the computation time for this algorithm is in the iteration phase, instead of looking at ALL the temporarily labeled nodes (to see which one should be permanently labeled), only look at the nodes which are successors of the most recently permanently labeled node. This can be done by keeping an array, 0..n, of all the nodes in the network, with the first \( m \) nodes in the array temporarily labeled, and the nodes from \( m \) to \( n \) permanently labeled. However, this modification will require some computation time to maintain this array.
algd1.c has been modified to use the linked adjacency list structure instead of the weight matrix representation. The modified C program is called algd2.c, and is displayed in Appendix A. The instructions for running this program are in Appendix B. Also, an example is shown in Appendix B.

Dijkstra's Shortest Path Algorithm can also be used to find the shortest path between one node and all other nodes in the network. To do this, the algorithm is run until EVERY node is permanently labeled. This is accomplished by changing the iteration phase so that instead of iterating until the destination node is permanently labeled, include a "for loop" so the iteration phase is completed for each node in the network. Then the distances in the distance array will correspond to the shortest path distance between the source node and all other nodes. The time required to run is still $O(n^2)$. A C program showing this modification is called algd3.c and is displayed in Appendix A. The instructions for running this program and an example are shown in Appendix B.

To find the shortest path from every node to all other nodes, modify Dijkstra's algorithm to run once for each node in the network. This is done by adding a "for loop" around the initialization and iteration phases. The time required to run this version is $O(n^3)$. The C program that performs this modification is called algd4.c and is presented in Appendix A. The instructions for running this version and an example are in Appendix B.
If the distances between the nodes are all equal to one, then the weight matrix will equal the adjacency matrix, and the resulting shortest path distance will represent the number of nodes that have to be traversed.

3.2 Bellman-Ford Shortest Path Algorithm

Bellman-Ford Shortest Path Algorithm can be used to find the distance from one node in a network to every other node in the network. This algorithm uses a label-correcting method. This algorithm will work correctly even if there are arcs with negative lengths in the network, but will not work if there are negative cycles in the network.

Initialization:

The label of a node represents the distance from the source node to that node. Initially, the label of the source node is set to zero and the labels of all other nodes are set to INFINITY (a very large number for practical purposes). A queue of nodes, queue, is maintained throughout this algorithm. This queue contains a list of the nodes that need to be further inspected. Initially, the source node is the only node in the queue.
Iteration:

During the iteration phase, the distance label of a node, \( y \), is updated to the value of the present distance from the source node to \( y \). Also, the new predecessor of \( y \) (tracing back to the source node) is found.

A node is taken from the head of the queue during each iteration. Call this node \( x \). All of the successors of node \( x \) are inspected. If the distance from the source node, \( s \), to a successor of node \( x \), say node \( y \), is lessened by going through \( x \), then the distance label for node \( y \) is reset to the smaller value. The predecessor for node \( y \) is set to node \( x \). Node \( y \) is added to the queue to be inspected later. Therefore, if a node’s distance label is reduced, that node is placed in the queue.

An improvement of this algorithm was suggested by d’Esopo and refined by Pape. Instead of always placing the node at the tail of the queue, a node can be added to the tail \textit{or} head of the queue. It is placed at the tail of the queue if it has never been in the queue before. However, if a node has been in the queue before, it is placed at the head of the queue, allowing it to be reinspected immediately, so that the labels of all the nodes that were reached by means of that node can be shortened first. This shortens the time required to execute the algorithm by decreasing the number of times nodes enter the queue.
If there are negative cycles in the network, this algorithm will not work. It will try to continually reduce the distance by going around and around the negative cycle.

The network representation for this algorithm is a forward star, making it ideal for sparse networks from a memory viewpoint. A C implementation of this algorithm is called algbf1.c and can be found in Appendix A. The instructions for running this program and analyzing the results are in Appendix B. An example is also displayed in Appendix B.

This algorithm can easily be modified to find the shortest paths between all pairs of nodes. The algorithm is run n times: once for each node as the source node. This is accomplished by putting a "for loop" around the initialization and iteration phases. The C program that performs this modification is called algbf2.c and is shown in Appendix A. The instructions for running this version and an example are in Appendix B.

It is difficult to estimate the number of times a node will enter and leave the queue for a general network model. The worst case is represented by a complete network.
3.3 Floyd-Warshall Algorithm

In some cases it is necessary to compute the shortest path between all pairs of nodes, that is, between all \( n \times (n - 1) \) ordered pairs of nodes in the network. *Floyd-Warshall Algorithm* will be used in this case. This algorithm is a dynamic programming algorithm which means the problem is solved by dividing it into smaller subproblems, solving the subproblems, and saving the answers in a tabular format. The combination of all the solutions of the subproblems aids in solving the main problem.

This algorithm works even if there are negative arc weights in the network. In addition, it will find any negative cycles. The computational complexity of this algorithm is \( O(n^3) \).

The Floyd-Warshall Algorithm is based on Warshall's algorithm for computing transitive closure\(^8\) of a matrix for a network. The input to the algorithm is an \( n \times n \) weight matrix. Starting with this weight matrix, \( n \) different matrices are built sequentially. The algorithm inserts a node into a path whenever it is beneficial to that path to do so.

---

Initialization:

Given weight matrix, $\text{MATRIX}$, initialize $\text{MATRIX}_{ii}$ to 0 for $i = 0, 1, \ldots, n-1$. $\text{PATH}$ is an $n \times n$ matrix in which the $i,j$th element is the node that is the predecessor of node $i$. Initialize $\text{PATH}_{ij}$ such that

$$
\text{PATH}_{ij} = \begin{cases}
  i & \text{if } \text{MATRIX}_{ij} \neq \infty \\
  -1 & \text{if } \text{MATRIX}_{ij} = \infty
\end{cases} \quad (1)
$$

for $i = 0, 1, \ldots, n-1$

$j = 0, 1, \ldots, n-1$

Iteration:

The input matrix is the weight matrix: $\text{MATRIX}^0$. Compute $\text{MATRIX}^1$ from $\text{MATRIX}^0$, $\text{MATRIX}^2$ from $\text{MATRIX}^1$, and so on, until $\text{MATRIX}^n$ has been computed. The rule used for computing the sequence of matrices is:

$$
\text{MATRIX}_{ij}^0 = \text{MATRIX}_{ij} \quad (2)
$$

$$
\text{MATRIX}_{ij}^h = \min( \text{MATRIX}_{ij}^{h-1}, \text{MATRIX}_{ik}^{h-1} + \text{MATRIX}_{kj}^{h-1}) \quad (3)
$$

$k = h - 1$

for $h = 1, 2, \ldots, n$ and $k = 0, 1, \ldots, n-1$
In the first iteration, the matrix contains direct distances and equation (2) is used. In iteration k, node k is inserted into the path from node i to node j if the distance from node i to node k plus the distance from node k to node j is less than the current distance from node i to node j. This action is represented by the minimization operation in equation (3). Thus, the distance from node i to node j is reduced by inserting node k into the path.

The PATH matrix is updated during each iteration. In iteration k, if node k is inserted into the path from node i to node j, PATH$_{ij}$ is replaced by the current value of PATH$_{kj}$.

The i,jth element of MATRIX$^n$ corresponds to the shortest distance from node i to node j in the network. If there are no negative cycles, then the non-diagonal entries in the final matrix, MATRIX$^n$, represent the shortest distances. If the network does have some negative cycles, then there will be some negative values in the matrix entries on the diagonal, and the non-diagonal entries will not represent the shortest distances. The reason there will be negative values in the entries on the diagonal is as follows. The algorithm works by inserting a node, k, into the path from node i to node j, if the weight of arc (i,k) plus the weight of arc (k,j) is less than the weight of the current path from node i to node j (from the
weight matrix). If \( i = j \) and the weight of arc \((i,k)\) plus the weight of arc \((k,j)\) results in a negative value, then this value will be less than the weight of the path from \( i \) to \( j \) when \( i = j \) (which is always 0 in a weight matrix). Therefore, there is a negative weight path from node \( i \) to node \( k \) and back to node \( i \) (node \( j \)). This forms a negative cycle that includes nodes \( i \) and \( k \), and may include other nodes in the network also.

The non-diagonal entries will not be accurate because a path from the source node to a destination node that is part of a negative cycle cannot be a shortest path because a path with a smaller weight can always be found by continuing around the negative cycle.

The C program written to implement this algorithm is called \texttt{algfw.c}. This implementation tests for a negative cycle and halts when one is found. The program listing is displayed in Appendix A. Instructions for running the program and analyzing the results are in Appendix B. Two examples are also displayed in Appendix B, one including a negative cycle.
Chapter 4

Analysis and Results

The algorithms researched were analyzed using three considerations: storage requirements, computational complexity, and actual processing time. The number of nodes in a given network is $n$ and the number of arcs is $m$.

Computational complexity is the study of the number of operations required by an algorithm. Most shortest path algorithms consist of two operations: addition and minimization. Both operations are represented in the following equation:

$$\text{WEIGHT}_i = \min(\text{WEIGHT}_v, \text{WEIGHT}_{\text{recent}} + \text{MATRIX}_{\text{recent},i})$$

for each successor node, $i$, of node recent

The equation is used to determine if the "distance" from the source node to node $i$ can be reduced by going through the most recently permanently labeled node (which then would become node $i$'s predecessor node). $\text{MATRIX}_{\text{recent},i}$ is the distance between the most recently permanently labeled node and node $i$. $\text{WEIGHT}_i$ represents the current distance between the source node and node $i$ (which may be reduced throughout the course of the program). $\text{WEIGHT}_{\text{recent}}$ represents the distance between the source node and the most recently permanently labeled node.
For some algorithms, the computational complexity will not change for a general directed graph. However, for the Bellman-Ford Algorithm, the number of operations will vary depending on the network. For this case, the worst possible number of operations has to be computed.

4.1 Dijkstra's Shortest Path Algorithm

The network is made to look like it contains all arcs possible by assigning arc lengths of INFINITY to non-existent arcs. Therefore, the time it takes to find the shortest path depends on the number of nodes, and not the number of arcs in the network.

One row of the weight matrix is inspected for each iteration of the outermost loop of the algorithm. The outermost loop is executed \( n - 1 \) times for the worst case (when the destination node is permanently labeled last). As more nodes get permanently labeled, the number of additions and minimizations needed to modify the temporary labels decreases. The inner loop is used to find the smallest temporarily labeled node for the \( i \)th row of the matrix. The first time the outer loop is executed, \( n - 2 \) additions and \( n - 2 \) minimizations are required to examine all the successors of node \( i \). Also, \( n - 2 \) minimizations (comparisons) are required to find the smallest value among all temporary labels. The second time
through the outer loop, \( n - 3 \) additions and \( n - 3 \) minimizations are required to examine all the successors of node \( i \). Also, \( n - 3 \) minimizations are needed to find the smallest temporarily labeled node. The third time through the loop, \( n - 4 \) additions and \( n - 4 \) minimizations are needed... and so on. Therefore, the total number of additions required by this method is \( (n - 1)(n - 2) / 2 \). The total number of minimizations required is \( (n - 1)(n - 2) \). This results in a computation time of approximately \( 3/2 \times (n^2 - 3n + 2) \). This is on the order of \( n^2 \). The actual running time of the algorithm depends on the proximity of the destination node to the source node. If the destination node is very close to the source node, it will be permanently labeled sooner than if it is the farthest node from the source node.

The second version of Dijkstra's Algorithm uses a linked adjacency list structure. The number of operations is not lessened, but the number of tests is lessened. Because instead of looping for all nodes in the network to find the successors of the most recently permanently labeled node, the program steps through a linked list of successors. To illustrate this point, consider a 100-node network with a source node having only 5 immediate successors. Version 1 of Dijkstra's Algorithm (algd1.c) would have to test each of 99 nodes in the first iteration to check if any are successors of the source node \textit{before} the program can determine if the temporary label can be shortened. However, version 2 of Dijkstra's Algorithm (algd2.c) steps through the linked list of 5 successors, without having to do a test to check if they \textit{are} successors. Complexity is still \( O(n^2) \).
For version 3 of Dijkstra's Algorithm, the version that was modified to find the shortest path between the source node and all other nodes (algd3.c), the computation time is still proportional to \( n^2 \). This is the same as the worst case for the single source/single destination version of Dijkstra's Algorithm, because each node must be permanently labeled.

In version 4, Dijkstra's original algorithm is repeated once for each node as the source node. To find the shortest path between all pairs of nodes in the network for this version, (algd4.c) the number of additions and minimizations will increase \( n \) times. The computation time will be proportional to \( n^3 \).

If the graph is not weighted, the weight matrix of the network equals the adjacency matrix. In this case, the logical operations, and and or, can be substituted for addition and minimization to save some computation time.

The storage required for the Dijkstra Shortest Path Algorithm, versions 1, 3, and 4, consists of memory for the weight matrix and the three arrays used for keeping track of permanently labeled nodes, node distances from the source, and predecessor nodes. The resulting storage requirement is \( n^2 + 3n \). The storage required for version 2 is \( n + 3m \) for the linked adjacency list, and \( 3n \) for the three arrays used for storing the queue, node distances from the source, and predecessor nodes. Therefore, the total storage required for version 2 is \( 4n + 3m \).
4.2 Bellman-Ford Shortest Path Algorithm

The difference between Dijkstra's Shortest Path Algorithm and the Bellman-Ford Shortest Path Algorithm is that in Dijkstra's algorithm, nodes are given a permanent label once. In the first version of the Bellman-Ford Algorithm a node is inspected many times and its label may be corrected each time. For this reason, it can be used for networks with negative arc weights, where Dijkstra's algorithm cannot. Each node can enter the queue (be labeled) as many as n - 1 times. This requires a computational complexity approximately proportional to n^3 at worst. However, for most cases, it will be less than this. Version 2, the version of the Bellman-Ford Shortest Path Algorithm modified to compute the shortest paths between all pairs of nodes, requires a computational complexity, at worst, n times greater than the original Bellman-Ford algorithm. Again, for most cases, it will be less than this.

The storage required consists of memory for the forward star structure (pointer, target, and weight arrays) and the distance, queue, and predecessor arrays. The resulting storage requirement is 4n + 2m.

If the network is sparse, m << n * (n - 1), the Bellman-Ford Algorithm is more efficient to use than the version of Dijkstra's Algorithm to find the shortest path from one node to all other nodes in the network.
4.3 Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm uses an $n \times n$ weight matrix. A total of $n$ matrices are computed: $\text{MATRIX}^0$, $\text{MATRIX}^1$, ..., $\text{MATRIX}^{n-1}$. Since each matrix consists of $n^2$ elements, a total of $n^3$ elements must be computed throughout the course of the algorithm. The computation of each element requires one addition and one minimization. Therefore, the computational complexity of the basic algorithm is proportional to $2n^3$. However, the actual processing time can be reduced by noting that for:

$$\text{MATRIX}_{ij}^k = \min \left[ \text{MATRIX}_{ij}^{(k-1)}, \left( \text{MATRIX}_{ik}^{(k-1)} + \text{MATRIX}_{kj}^{(k-1)} \right) \right]$$

$i \neq j$, $i \neq k$, and $j \neq k$. By including a test for these cases in the for loops of the program for the Floyd-Warshall Algorithm, the number of additions can be reduced to $n(n - 1)(n - 2)$ and the number of minimizations can be reduced to $n(n - 1)(n - 2)$. The resulting computational complexity is $2n(n - 1)(n - 2)$, which is still proportional to $2n^3$. However, this modification can only be made for networks without negative cycles, because the entries on the diagonal of the matrix are used to tell if a negative cycle has been found. This modification would exclude the processing of the entries on the diagonal.

Chapter 4
The execution time can be reduced even further for sparse graphs by including a test for the case where $\text{MATRIX}_{ik} = \text{INFINITY}$ and $\text{MATRIX}_{lj} = \text{INFINITY}$. This will eliminate the addition and minimization operations for these cases, at the expense of the time needed to execute the extra tests. However, the computation time is still $O(n^3)$.

Storage is required for one $n \times n$ matrix, which the algorithm keeps writing over for each iteration. Also, an $n \times n$ matrix can be used to store the predecessor array. Therefore, a total of $2n^2$ units of storage is required.

If the graph is not weighted, the weight matrix of the network equals the adjacency matrix. In this case, the logical operations, and and or, can be substituted for addition and minimization to save some computation time.

A very small amount of additional storage may be needed for each algorithm, over and above what has been stated, for counters, flags, and so on. This will depend on how the algorithm is implemented and what computer language is used to implement it.
4.4 Time Trials

The implementations of the algorithms that are compared in this section are shown in Appendix A. Comparisons are made first with a sparse network, in which the number of arcs is much less than \( n \ast (n - 1) \). Next, comparisons are made with a complete network, in which there are \( n \ast (n - 1) \) arcs.

4.4.1 Sparse Network

The first set of comparisons were made using a sparse, 100 node network. There were 191 arcs in this network. The time trials were run on an IBM PS/2 Model 70 (20 MHz). Dijkstra’s Shortest Path Algorithm (version 3) was compared to the Bellman-Ford Algorithm (version 1) for the case where the shortest path between a single source node and all other nodes in the network is computed. The following table displays the results for each program.

<table>
<thead>
<tr>
<th>Dijkstra</th>
<th>Bellman-Ford</th>
</tr>
</thead>
<tbody>
<tr>
<td>algd3.c</td>
<td>algbf1.c</td>
</tr>
<tr>
<td>.09 sec</td>
<td>&lt; .01 sec</td>
</tr>
</tbody>
</table>

Table 4-1 Comparison of Dijkstra and Bellman-Ford Algorithms for a Sparse Network

The Bellman-Ford Algorithm was faster than Dijkstra’s Algorithm for the sparse network.
The algorithms were again compared using the sparse 100 node network. This time the case where the shortest paths between all pairs of nodes was computed using all three algorithms (version 4 of the Dijkstra Algorithm and version 2 of the Bellman-Ford Algorithm were used). Table 4-2 displays the results.

<table>
<thead>
<tr>
<th>Dijkstra</th>
<th>Bellman-Ford</th>
<th>Floyd-Warshall</th>
</tr>
</thead>
<tbody>
<tr>
<td>algd4.c</td>
<td>algbf2.c</td>
<td>algfw.c</td>
</tr>
<tr>
<td>9.39 sec</td>
<td>0.17 sec</td>
<td>1.00 sec</td>
</tr>
</tbody>
</table>

Table 4-2 Comparison of Dijkstra, Bellman-Ford, and Floyd-Warshall Algorithms for a Sparse Network

Again, the Bellman-Ford Algorithm proved to be the fastest at finding the shortest path for the sparse network.

4.4.2 Complete Network

The second set of timing comparisons was made using a complete, 60-node network. There were 3540 arcs in this network. The timing comparisons were run on an IBM PS/2 Model 70 (20 MHz). Dijkstra’s Shortest Path Algorithm (version 3) was compared to the Bellman-Ford Algorithm (version 1) for the case where the shortest path between a single source node and all other nodes in the network is computed. The results for each program are shown in Table 4-3.

Chapter 4
For the complete network, it can be seen that the speed of the Dijkstra Shortest Path Algorithm is comparable to the speed of the Bellman-Ford Algorithm.

Next, the speed of the algorithms was compared for the case where the shortest path between all pairs of nodes is computed. Table 4-4 displays the timing results.

<table>
<thead>
<tr>
<th>Dijkstra</th>
<th>Bellman-Ford</th>
<th>Floyd-Warshall</th>
</tr>
</thead>
<tbody>
<tr>
<td>algd4.c</td>
<td>algbf2.c</td>
<td>algfw.c</td>
</tr>
<tr>
<td>2.69 sec</td>
<td>3.36 sec</td>
<td>2.24 sec</td>
</tr>
</tbody>
</table>

Table 4-4 Comparison of Dijkstra, Bellman-Ford, and Floyd-Warshall Algorithms for a Complete Network
The Floyd-Warshall Algorithm proved to be the fastest algorithm for this case, followed closely by the Dijkstra Algorithm.

NOTE: The results for the complete network timing example cannot be accurately compared to the results of the sparse network timing example because the sparse network had 100 nodes and the complete network had 60 nodes.
Chapter 5

Conclusions

A precise comparison of different shortest path algorithms using a generalized network cannot be performed accurately because one algorithm may work better than another in a particular situation, or may depend on where the destination node is in relation to the source node. Also, various limitations may be present in different situations, i.e., limited computer storage resources, computing power, etc. These limitations may support the use of one algorithm over another for a specific circumstance.

A dense network is a network in which the number of arcs is close to or equal to \( n^* (n - 1) \), the number of arcs in a complete network. A sparse network is a network in which the number of arcs is much less than \( n^* (n - 1) \).

Any of the algorithms that use a forward star or linked adjacency list network representation are best to use if the network is sparse or small, and computer storage is at a minimum. The forward star and linked adjacency list network representations for input files take up far less computer storage than a weight matrix representation for cases when the number of arcs is much less than \( n^* (n - 1) \).
If computer storage is not at a premium, then the speed of the algorithms can be compared to be used as a basis for determining the best algorithm. From the analysis of computational complexity, and actual processing time presented in Chapter 4, the following additional conclusions can be made.

First, if the shortest path between a single source and single destination needs to be computed, Dijkstra’s Shortest Path Algorithm (version 1 or version 2) is the only algorithm presented that will compute this.

If the network is sparse, and the shortest paths from a single source to all other nodes need to be computed, or the shortest paths between all pairs of nodes need to be computed, the fastest algorithm to use is the Bellman-Ford Algorithm (version 1).

If the shortest paths from a single source node to all other nodes need to be computed in a dense network with non-negative arc weights, Dijkstra’s Algorithm (version 4) and the Bellman-Ford Algorithm are equally as fast.

If the shortest paths between all pairs of nodes need to be computed in dense networks with negative OR non-negative arc weights, then the Floyd-Warshall Algorithm is the best algorithm to use.
Additional shortest path algorithms have been developed by Hu, Dantzig, and others using slightly different methods than those used in the algorithms presented. However, similar results were obtained.

Also, variants of the shortest path algorithm exist. For example: to compute the shortest path through specified nodes; to find the second shortest path, the third shortest path, ... , the \( k \)th shortest path; and to find the shortest path through a network when constraints have been imposed (i.e. an upper bound placed on the weight of arcs in the resulting shortest path).
REFERENCES

Chapter 1:

Chapter 2:

Chapter 3 and Chapter 4:
Dijkstra's Shortest Path Algorithm

**Bellman-Ford Algorithm**

**Floyd-Warshall Algorithm**

**Additional Reference:**
APPENDIX A

PROGRAM LISTINGS

algdl.c
/* algdl.c */
/* Susan Hojnacki */
/* April 1991 */
/* Dijkstra's Shortest Path Algorithm to find the shortest path between */
/* a source node and a single destination node in a network. */
/* The network must be described by a weight matrix and stored in a file. */
/* Nodes in the network should be numbered starting with zero. */
/* Syntax: algdl [-v] */
#define dprintf if (verbose) printf
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "spath.h"

int matrix[MAXNODES][MAXNODES];
int *weight;
int *permanent;
int *pred;
int s, d, n, recent, path, verbose;
void trace_path(int);

main(argc, argv)
int argc;
char **argv;
{
    int i, j, k;
    char oneline[LINELENGTH], fname[12], ch, seps[] = " ,\t\n";
    char *number;
    FILE *mtxfile;

    if (**argv != NULL && **argv == '\n') {
        ch = **argv;
        if (ch == '\n') verbose 1;
    }

    printf("DIJKSTRA'S SHORTEST PATH ALGORITHM (algdl.c) \n\n");;

    /* Prompt user to enter the file name of file matrix is stored in. */
    printf("Enter the weight matrix file name: \n");
    scanf("%s", fname);

    /* Open the file and read in the weight matrix. */
    if ( (mtxfile = fopen(fname, "r")) != NULL ) {
        n 0;
        /* Read in one line at a time from the file, and put it into the */
        while (fgets(oneline, LINELENGTH, mtxfile) != NULL) {
            k 0;
            /* Use strtok() function to read in one number at a time from */
            /* the oneline buffer. */
            number = strtok(oneline, seps);
            while (number != NULL) {
                /* Convert the item read in to an integer and store it in */
                /* the matrix array. */
                }
matrix[n][k]  atoi(number);
k++;  
number  strtok(NULL, seps);
}
n++;
else  
printf("ERROR: Cannot open %s\n\n",fname);
exit(-1);

/* Prompt user to enter the source node and the destination node.*/
printf("Enter the node numbers of the source node and destination node: \n");
scanf("%d %d", &s, &d);

printf("NETWORK: \n");
printf("Total number of nodes in the network = %d.\n\n", n);

if (verbose)  
for (i = 0; i < n; i++)  
for (j = 0; j < n; j++)  
printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);

*/
Dynamically allocate the weight, permanent, and pred arrays (global). */
weight  (int *) malloc(n * sizeof(int));  
permanent  (int *) malloc(n * sizeof(int));  
pred  (int *) malloc(n * sizeof(int));
init();
iterate();

printf("RESULTS: \n");

/* Print out the results in the three arrays if -v was specified. */
if (verbose)  
/* Print out the results in the permanent array. */
printf("permanent[0..%d] = [",n-1);  
for (i = 0; i < n; i++)  
if (i < n - 1)  
printf("%d, ", permanent+i));  
else  
printf("%d\n\n", *(permanent+i));
}

/* Print out the results in the weight array. */
printf("weight[0..%d] = [",n-1);  
for (i = 0; i < n; i++)  
if (i < n - 1)  
printf("%d, ", *(weight+i));  
else  
printf("%d\n\n", *(weight+i));
}

/* Print out the results in the pred array. */
printf("pred[0..%d] = [",n-1);  
for (i = 0; i < n; i++)  
if (i < n - 1)  
printf("%d, ", *(pred+i));  
else  
printf("%d\n\n", *(pred+i));
}

if (path)  
printf("The shortest path has been found.\n");
trace_path(d);
}
else printf("A path could not be found from %d to %d.
", s, d);

/* Free the memory that was allocated with malloc(). */
free(permanent);
free(weight);
free(pred);
}

init()
{
int i;
dprintf("\nInitialization Phase Starting.\n");
for (i = 0; i < n; i++) {
    *(weight + i) = INFINITY;
    *(permanent + i) = 0;
    *(pred + i) = -1;
    dprintf("weight[%d] = %d, permanent[%d] = %d, pred[%d] = %d \n",
            i, *(weight + i), i, *(permanent + i), i, *(pred + i));
}
*(weight + s) = 0;
*(permanent + s) = 1;
recent = s;
path = 1;
dprintf("Initialization Phase Complete.\n");
}

iterate()
{
int i, j, temp1, temp2, newlabel;
dprintf("Iteration Phase Starting.\n");
/* Loop until the destination node is permanently labeled. */
while(*(permanent + d) == 0) {
    /* Look at each node in the network and find the successors of */
    /* the most recently permanently labeled node (that have not */
    /* been permanently labeled themselves). */
    dprintf("Examining the immediate successors of node %d: \n",
            recent);
    for (i = 0; i < n; i++) {
        if (matrix[recent][i] < INFINITY && !*(permanent + i)) {
            dprintf("Node %d is an immediate successor of %d.\n", i, recent);
            /* Examine all the successors and shorten their temporary */
            /* labels if the path from the source node to any of them */
            /* is shorter by going through the predecessor than by not */
            /* going through the predecessor. */
            newlabel = *(weight + recent) + matrix[recent][i];
            if (newlabel < *(weight + i)) {
                dprintf("Node %d’s temporary label is reduced from %d to %d.\n",
                        i, *(weight + i), newlabel);
                *(weight + i) = newlabel;
                *(pred + i) = recent;
            }
        }
    }
    temp1 = INFINITY;
    dprintf("\nExamining all temporary labels to find the shortest.\n");
    for (i = 0; i < n; i++) {
        if (!*(permanent + i) && *(weight + i) < temp1) {
            dprintf("Node %d has a temporary label of %d.\n", i, *(weight + i));
            temp2 = i;
            temp1 = *(weight + i);
        }
    }
}

Appendix A
if (temp1 < INFINITY) {
    *(permanent+temp2) = 1;
    recent temp2;
    dprintf("Node %d has been permanently labeled with %d.\n", temp2, *(weight + temp2));
} else {
    path = 0;
    *(permanent+d) = 1;
}
}
dprintf("Iteration Phase Complete.\n\n");
}

void trace_path(int d){
    printf("Distance (weight) from node %d to node %d %d\n", s, d, *(weight + d));
    printf("Shortest Path traced backwards from node %d to node %d: \n", d, s);
    while (1) {
        printf(" %d", d);
        if (*(pred+d) == s) {
            printf(" , %d\n", s);
            break;
        } else {
            *(pred+d);
            printf(" , ");
        }
    } /* while (1) */
}
/* algd2.c */
/* Susan Hojnacki */
/* April 1991 */
/* Dijkstra's Shortest Path Algorithm to find the shortest path between */
/* a source node and a single destination node in a network. */
/* This version expects the network to be described by an arc list */
/* stored in a file. Nodes in the network should be numbered starting */
/* with zero. */
/* Syntax: algd2 [-v] -v = verbose */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "spath.h"
#define dprintf if (verbose) printf

/* Structure for a linked list. */
struct node {
  int successor;
  int weight;
  struct node *next;
};

struct node *nodes[MAXNODES];
int *weight;
int *permanent;
int *pred;
int s, d, n, recent, path, verbose;
void trace_path(int);

verbose = 0;

main(argc, argv)
int argc;
char **argv;
{
  int i, j, k, r, num, m;
  char oneline[LINELength], fname[12], ch, seps[] = " ,t\n";
  char *number;
  FILE *mtxfile;
  struct node *temp;
  struct node *append(struct node *, struct node *);
  struct node *get_node(struct node *);
  void print_matrix(struct node *);

  if (**argv != NULL && **argv == '-') {
    /* Get the character after the dash. */
    ch = *argv;
    if (ch == 'v') verbose = 1;
  }

  printf("DIJKSTRA'S SHORTEST PATH ALGORITHM (algd2.c)\n\n");

  /* Prompt user to enter the file name of file the network is stored in.*
  printf("Enter the arc data file name: \n");
  scanf("%s", fname);

  /* Prompt user to enter the source node and the destination node.*
  printf("\nEnter the node numbers of the source node and destination node,\n");
  printf("and the total number of nodes: \n");
  scanf("%d %d %d", &s, &d, &n);

  /* Dynamically allocate the weight, permanent, and pred arrays (global). */
  weight = (int *) malloc(n * sizeof(int));
  permanent = (int *) malloc(n * sizeof(int));

Appendix A 50
pred = (int *) malloc(n * sizeof(int));

/* Initialize the nodes array of pointers to NULL for each node. */
for (i = 0; i < n; i++) {
    nodes[i] = NULL;
}

/* Open the file and read in the matrix. */
if ((mtxfile = fopen(fname, "r")) != NULL) {
    /* Read in one line at a time from the file, and put it into the */
    /* one line buffer. */
    m = 0;
    while (fgets(oneline, LINELENGTH, mtxfile) != NULL) {
        /* Use strtok() function to read in one number at a time from */
        /* the oneline buffer. */
        if ((number = strtok(oneline, seps)) != NULL) {
            num = atoi(number);
            number = strtok(NULL, seps);
            if (number != NULL) {
                /* Store successor node number in the structure. */
                temp = get_node(temp);
                temp->successor = atoi(number);
                number = strtok(NULL, seps);
                if (number != NULL) {
                    /* Store weight in the structure. */
                    temp->weight = atoi(number);
                    temp = append(temp, nodes[num]);
                    m++;
                    } /* if number != NULL ... */
                } /* if number == NULL ... */
        }/* while */
    }
} else {
    printf("ERROR: Cannot open %s.\n",fname);
    exit(-1);
}

printf("\nNETWORK: \n");
printf("Total number of arcs in the network = %d.\n", m);
printf("Total number of nodes in the network = %d.\n", n);

/* Print the network out to confirm it was read in properly if the */
/* -v option was specified on the command line. */
if (verbose) {
    for (i = 0; i < n; i++) {
        printf("Node %d: \n", i);
        print_matrix(nodes[i]);
    }
}

init();
iterate();

/* Print the results in the three arrays if -v was specified. */
if (verbose) {
    /* Print out the results in the permanent array */
    printf("permanent[0..%d] = [{\n", n-1);
    for (i = 0; i < n; i++) {
        if (i < n - 1)
            printf("%d, ", permanent+i);
        else
            printf("%d]\n", permanent+i);
    }
}

/* Print out the results in the weight array. */
printf("weight[0..%d] = [", n-1); 
for (i = 0; i < n; i++) {
    if (i < n) 
        printf("%d", *(weight+i)); 
    else 
        printf("%d\\n", *(weight+i)); 
} 

/* Print out the results in the pred array. */ 
printf("pred[0..%d] = [", n-1); 
for (i = 0; i < n; i++) {
    if (i < n) 
        printf("%d", *(pred+i)); 
    else 
        printf("%d\\n", *(pred+i)); 
} 

if (path) { 
    printf("The shortest path has been found.\n"); 
    trace_path(d);
} else printf("A path could not be found from %d to %d.\n", s, d);

/* Free the memory that was allocated with malloc(). */ 
free(permanent); 
free(weight); 
free(pred);
}

init() 
{ 
    int i; 
    dprintf("\nInitialization Phase Starting.\n"); 
    for (i = 0; i < n; i++) {
        *(weight + i) = INFINITY; 
        *(permanent + i) = 0; 
        *(pred + i) = -1; 
        dprintf("weight[%d] = %d, permanent[%d] = %d, pred[%d] = %d \n", 
                i, *(weight+i), i, *(permanent+i), i, *(pred+i)); 
    }
    *(weight + s) = 0; 
    *(permanent + s) = 1; 
    recent = s; 
    path = 1; 
    dprintf("Initialization Phase Complete.\n\n"); 
}

iterate() 
{ 
    int i, j, r, tempi, temp2, newlabel; 
    struct node *list; 
    dprintf("Iteration Phase Starting.\n"); 
    
    /* Loop until the destination node is permanently labeled. */ 
    while (* (permanent+d) == 0) {
        /* Look at each node in the network and find the successors of */ 
        /* the most recently permanently labeled node (that have not */ 
        /* been permanently labeled themselves). */ 
        dprintf("Examining the immediate successors of node %d: \n", recent);
        if (nodes[recent] != NULL) 
            list = nodes[recent]; 
            while (1) {
                /* Step through the adjacency list to examine only successors */ 
                } 
            } 
        } 
    } 
    } 

Appendix A
if (!*(permanent+list->successor)) {
    dprintf("Node %d is an immediate successor of %d.\n", list->successor, recent);
    /* Examine all the successors and shorten their temporary */
    /* labels if the path from the source node to any of them */
    /* is shorter by going through the predecessor than by not*/
    /* going through the predecessor. */
    newlabel = *(weight+recent) + list->weight;
    if (newlabel < *(weight+list->successor)) {
        dprintf("Node %d's temporary label is reduced from %d to %d.\n",
                 list->successor, *(weight+list->successor), newlabel);
        *(weight+list->successor) = newlabel;
        *(pred+list->successor) = recent;
    }
    if (list->next == NULL)
        break;
    else
        list = list->next;
}

for (i = 0; i < n; i++) {
    if (!*(permanent+i) && *(weight+i) < temp1) {
        dprintf("Node %d has a temporary label of %d.\n", i, *(weight+i));
        temp2 = i;
        temp1 = *(weight+i);
    }
}

if (temp1 < INFINITY) {
    *(permanent+temp2) = 1;
    recent = temp2;
    dprintf("Node %d has been permanently labeled with %d.\n\n",
            i, *(weight + temp2));
} else {
    path = 0;
    *(permanent+d) = 1;
}

/* allocate space for and initialize a structure; return a pointer to it */
struct node *get_node( struct node *item)
{
    void *malloc (size_t);
    /* allocate enough space to store a successor node. */
    item = (struct node *) malloc(sizeof (struct node));
    /* if allocated, initialize the structure */
    if (item != NULL) {
        item->successor = 0;
        item->weight = 0;
        item->next = NULL;
    } else
        printf("Nothing allocated.\n");
    /* return the pointer */
return (item);
}

/* add node to which new points to the end of the linked list to which */
/* list points. */
struct node *append(struct node *new, struct node *list)
{
    if (list == NULL) {
        list = new;
        return(list);
    }
    else {
        /* return results of most recent search */
        list->next = append(new, list->next);
        return(list);
    }
}

/* display the contents of the matrix */
void print_matrix(struct node *list)
{
    if (list != NULL) {
        printf(" %d %d\n", list->successor, list->weight);
        if (list->next != NULL) print_matrix(list->next);
    }
}

void trace_path(d)
int d;
{
    printf("Distance (weight) from node %d to node %d %d\n", s, d, *(weight + d));
    printf("Shortest Path traced backwards from node %d to node %d:\n",d, s);
    while (1) {
        printf(" %d",d);
        if (*(pred + d) == s) {
            printf(" %d\n\n",s);
            break;
        }
        else {
            d = *(pred + d);
            printf(".\"");
        }
    } /* while (1) */
}
algd3.c

/* algd3.c */
/* Susan Hojnacki */
/* April 1991 */
/* Dijkstra's Shortest Path Algorithm */
/* Modified to find shortest paths from source node to all other nodes. */
/* The network must be stored as a weight matrix in a file. Nodes in */
/* the network should be numbered starting with zero. */
/* Syntax: algd3 [-v] */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "spath.h"
#define dprintf if (verbose) printf

int matrix[MAXNODES][MAXNODES];
int *weight;
int *permanent;
int *pred;
int s, n, recent, verbose;
void trace_path(int);

verbose = 0;

main(argc, argv)
int argc;
char **argv;
{
    int i, j, k;
    char oneline[LINELENGTH], fname[12], ch, seps[] = " ,\t\n";
    char *number;
    FILE *mtxfile;
    if (**argv != NULL && **argv == '\') {
        /* Get the character after the dash. */
        ch = ***argv;
        if (ch == 'v') verbose = 1;
    }

    printf("DIJKSTRA'S SHORTEST PATH ALGORITHM (algd3.c)\n");
    printf("Version to find shortest path from source node to all other nodes.\n\n");

    /* Prompt user to enter the file name of file matrix is stored in. */
    printf("Enter the weight matrix file name: \n");
    scanf("%s", fname);

    /* Open the file and read in the weight matrix. */
    if ( (mtxfile = fopen(fname, "r")) != NULL) {
        j = 0;
        /* Read in one line at a time from the file, and put it into the */
        /* oneline buffer. */
        while (fgets(oneline, LINELENGTH, mtxfile) != NULL) {
            k = 0;
            /* Use strtok() function to read in one number at a time from */
            /* the oneline buffer. */
            number = strtok(oneline, seps);
            while(number != NULL) {
                /* Convert the item read in to an integer and store it in */
                /* the matrix array. */
                matrix[j][k] = atoi(number);
                k++;
                number = strtok(NULL, seps);
            }
            j++;
        }
    }

Appendix A  55
else
    printf("ERROR: Cannot open %s\n",fname);
    exit(-1);
}

/* Determine n (total number of nodes in the network). */

/* Prompt user to enter the source node. */
printf("Enter the node number of the source node.\n");
scanf("%d",&s);

printf("NETWORK:\n");
printf("Total number of nodes in the network %d.\n",n);

/* Print the matrix out to confirm it was read in properly if the */
/* -v option was specified on the command line. */
if (verbose)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            printf("matrix[%d][%d] = %d \n", i,j,matrix[i][j]);

/* Dynamically allocate the weight, permanent, and pred arrays (global). */
weight = (int *) malloc(n * sizeof(int));
permanent = (int *) malloc(n * sizeof(int));
pred = (int *) malloc(n * sizeof(int));

init();
iterate();

printf("RESULTS:\n");
/* Print the results in the three arrays if -v was specified. */
if (verbose)
    /* Print out the results in the permanent array. */
    printf("permanent[0..%d] [\n",n - 1);
    for (i = 0; i < n; i++)
        if (i < n - 1)
            printf("%d,\n", *(permanent+i));
        else
            printf("%d\n", *(permanent+i));

    /* Print out the results in the weight array. */
    printf("weight[0..%d] [\n",n - 1);
    for (i = 0; i < n; i++)
        if (i < n - 1)
            printf("%d,\n", *(weight+i));
        else
            printf("%d\n", *(weight+i));

    /* Print out the results in the pred array. */
    printf("pred[0..%d] [\n",n - 1);
    for (i = 0; i < n; i++)
        if (i < n - 1)
            printf("%d,\n", *(pred+i));
        else
            printf("%d\n", *(pred+i));

    /* Print out the paths. */
    for (i = 0; i < n; i++)
        if (s != i && *(pred + i) != -1) trace_path(i);
        if (s != i && *(pred + i) == -1) printf("No path from %d to %d.\n",s,i);

Appendix A
init()

    init()
    {
        int i;
        dprintf("nInitialization Phase Starting.\n");
        for (i = 0; i < n; i++) {
            *(weight + i) = INFINITY;
            *(permanent + i) = 0;
            *(pred + i) = -1;
            dprintf("weight[%d] = %d, permanent[%d] = %d, pred[%d] = %d \n",
                     i, *(weight+i), i, *(permanent+i), i, *(pred+i));
        }
        *(weight + s) = 0;
        *(permanent + s) = 1;
        recent = s;
        dprintf("Initialization Phase Complete.\n");
    }

iterate()

    iterate()
    {
        int i, j, d, tempi, temp2, newlabel;
        dprintf("Iteration Phase Starting.\n");
        /* Loop until every node is permanently labeled. */
        for (d = 0; d < n; d++) {
            /* Look at each node in the network and find the successors of */
            /* the most recently permanently labeled node (that have not */
            /* been permanently labeled themselves). */
            dprintf("Examining the immediate successors of node %d: \n", recent);
            for (i = 0; i < n; i++) {
                if ((matrix[recent][i] < INFINITY) & *(permanent+i)) {
                    dprintf("Node %d is an immediate successor of %d.\n", i, recent);
                    /* Examine all the successors and shorten their temporary */
                    /* labels if the path from the source node to any of them */
                    /* is shorter by going through the predecessor than by not */
                    /* going through the predecessor. */
                    newlabel = *(weight+recent) + matrix[recent][i];
                    if (newlabel < *(weight+i)) { 
                        dprintf("Node %d's temporary label is reduced from %d to %d.\n", 
                                i, *(weight+i), newlabel);
                        *(weight+i) = newlabel;
                        *(pred+i) = recent;
                    }
                }
            }
        }
        /* Examine all temporary labels to find the shortest: */
        for (i = 0; i < n; i++) {
            if (! *(permanent+i) & *(weight+i) < temp1) {
                dprintf("Node %d has a temporary label of %d.\n", i, *(weight+i));
                temp2 = i;
                temp1 = *(weight+i);
            }
        }
    }

Appendix A

57
if (temp1 < INFINITY) {
    *(permanent + temp2) = 1;
    recent = temp2;
    dprintf("Node %d has been permanently labeled with %d.\n\n", temp2, *(weight + temp2));
} else {
    *(permanent + d) = 1;
}

dprintf("Iteration Phase Complete.\n\n");

void trace_path(d)
int d;
{
    printf("Distance (weight) from node %d to node %d = %d\n", s, d, *(weight + d));
    printf("Shortest Path traced backwards from node %d to node %d:\n", d, s);
    while (1) {
        printf(" %d", d);
        if (*(pred + d) == s) {
            printf("", d\n\n", s);
            break;
        } else {
            d = *(pred + d);
            printf(".");
        }
    } /* while (1) */
}
/* algd4.c */
/* Susan Hojnacki */
/* April 1991 */
/* Dijkstra's Shortest Path Algorithm */
/* Modified to find shortest paths between all pairs of nodes. */
/* The network must be described as a weight matrix and stored in a file. */
/* Nodes in the network should be numbered starting with zero. */
/* Syntax: algd4 [-v] */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "spath.h"
#define dprintf if (verbose) printf

int matrix[MAXNODES][MAXNODES];
int *weight;
int *permanent;
int *pred;
int s, d, n, recent, path, verbose;
void trace_path(int);

int main(argc, argv)
int argc;
char *argv[]
{
int i, j, k;
char oneline[LINELENGHT], fname[12], ch, seps[] = " ,\t\n";
char *number;
FILE *mtxfile;

if (**argv != NULL & **argv == ' ')
/* Get the character after the dash. */
  ch = **argv;
  if (ch == 'v') verbose = 1;
}

printf("DIJKSTRA'S SHORTEST PATH ALGORITHM (algd4.c)\n");
printf("Version to find shortest paths between all pairs of nodes.\n\n");

/* Prompt user to enter the file name of file matrix is stored in. */
printf("Enter the weight matrix file name: \n");
scanf("%s", fname);

/* Open the file and read in the weight matrix. */
if ( (mtxfile = fopen(fname, "r")) != NULL )
  j 0;
/* Read in one line at a time from the file, and put it into the */
/* oneline buffer. */
while (fgets(oneline, LINELENGHT, mtxfile) != NULL) {
  k 0;
/* Use strtok() function to read in one number at a time from */
/* the oneline buffer. */
  number = strtok(oneline,seps);
  while (number != NULL) {
    /* Convert the item read in to an integer and store it in */
    /* the matrix array. */
    matrix[j][k] = atoi(number);
    k++;
    number = strtok(NULL, seps);
  }
  j++;
}

Appendix A 59
else {
    printf("ERROR: Cannot open %s.\n\n", fname);
    exit(-1);
}

/* Determine n (total number of nodes in the network). */
n = j;

printf("\nNETWORK:\n\n");
printf("Total number of nodes in the network %d.\n\n", n);

/* Print the matrix out to confirm it was read in properly if the */
/* -v option was specified on the command line. */
if (verbose)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            printf("matrix[%d][%d] %d\n", i, j, matrix[i][j]);

/* Dynamically allocate the weight, permanent, and pred arrays (global). */
weight (int *) malloc(n * sizeof(int));
permanent (int * ) malloc(n * sizeof(int));
pred - (int *) malloc(n * sizeof(int));

for (s = 0; s < n; s++) {
    printf("\n\n\nSOURCE NODE %d.\n\n", s);
    init();
    iterate();

    /* Print the results in the three arrays if -v was specified. */
    if (verbose) {
        /* Print out the results in the permanent array. */
        printf("permanent[0..%d] [", n 1);
        for (i 0; i < n; i++) {
            if (i < n 1)
                printf("%d,
\n", *(permanent+i));
            else
                printf("%d\n\n", *(permanent+i));
        }

        /* Print out the results in the weight array. */
        printf("weight[0..%d] [", n 1);
        for (i 0; i < n; i++) {
            if (i < n 1)
                printf("%d,
\n", *(weight+i));
            else
                printf("%d\n\n", *(weight+i));
        }

        /* Print out the results in the pred array. */
        printf("pred[0..%d] = [", n 1);
        for (i 0; i < n; i++) {
            if (i < n 1)
                printf("%d,
\n", *(pred+i));
            else
                printf("%d\n\n", *(pred+i));
        }
    }

    /* Print out the paths. */
    for (i 0; i < n; i++) {
        if (s != i && *(pred + i) != -1) trace_path(i);
        if (s != i && *(pred + i) == -1) printf("No path found from %d to %d.\n\n", s, i);
    }

Appendix A
} /* for... loop for each node */

/* Free the memory that was allocated with malloc(). */
free(persistent);
free(weight);
free(pred);
}

init()
{
    int i;
    dprintf("\nInitialization Phase Starting.\n");
    for (i = 0; i < n; i++) {
        *(weight + i) = INFINITY;
        *(permanent + i) = 0;
        *(pred + i) = -1;
        dprintf("weight[%d] = %d, permanent[%d] = %d, pred[%d] = %d \n".
            i, *(weight+i), i, *(permanent+i), i, *(pred+i));
    }
    *(weight + s) = 0;
    *(permanent + s) = 1;
    recent = s;
    path = 1;
    dprintf("Initialization Phase Complete.\n\n");
}

iterate()
{
    int i, j, temp1, temp2, newlabel;
    dprintf("Iteration Phase Starting.\n");

    /* Loop until every node has been permanently labeled. */
    for (d = 0; d < n; d++) {
        /* Look at each node in the network and find the successors of */
        /* the most recently permanently labeled node (that have not */
        /* been permanently labeled themselves). */
        dprintf("Examining the immediate successors of node %d:\n", recent);
        for (i = 0; i < n; i++) {
            if ((matrix[recent][i] < INFINITY) && *(permanent+i)) {
                dprintf("Node %d is an immediate successor of %d.\n", i, recent);
                /* Examine all the successors and shorten their temporary */
                /* labels if the path from the source node to any of them */
                /* is shorter by going through the predecessor than by not*/
                /* going through the predecessor. */
                newlabel = *(weight + recent) + matrix[recent][i];
                if (newlabel < *(weight+i)) {
                    dprintf("Node %d's temporary label is reduced from %d to %d.\n","n",
                        i, *(weight+i), newlabel);
                    *(weight+i) = newlabel;
                    *(pred+i) = recent;
                }
            }
        }
        temp1 = INFINITY;
        dprintf("\nExamining all temporary labels to find the shortest:\n");
        for (i = 0; i < n; i++) {
            if (! *(permanent+i) && *(weight+i) < temp1) {
                dprintf("Node %d has a temporary label of %d.\n", i, *(weight+i));
                temp2 = i;
                temp1 = *(weight+i);
            }
        }
    }
}
if (temp1 < INFINITY) {
    *(permanent+temp2) 1;
    recent temp2;
    dprintf("Node %d has been permanently labeled with \%d.\n\n", 
        temp2, *(weight + temp2));
} else {
    path = 0;
    *(permanent+d) 1;
}

if (path) printf("A shortest path has been found.\n");
dprintf("Iteration Phase Complete.\n\n");

Appendix A
algbfl.c

/* algbfl.c */
/* Susan Hojnacki */
/* April 1991 */
/* Bellman-Ford (refined by Pape) Shortest Path Algorithm to find the */
/* shortest path between a source node and all other nodes in the */
/* network. This version expects the network to be described by a */
/* forward star matrix stored in a file as an arc list. Nodes in the */
/* network should be numbered starting with zero. */
/* Syntax: algbfl [-v] -v = verbose */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "spath.h"
define dprintf if (verbose) printf

int *pointer;
int target[MAXARCS], weight[MAXARCS];
int *distance;
int *queue;
int s, x, n, tail, verbose;
void trace_path(int);
verbose = 0;

main(argc, argv)
int argc;
char **argv;
{
    int i, j, num, m, index, first, last;
    char oneline[LINELength], fname[12], ch, seps[] = " ,\t\n";
    char *number;
    FILE *mtxfile;

    if (**argv != NULL && **argv == '.') {
        /* Get the character after the dash. */
        ch = ***argv;
        if (ch == 'v') verbose = 1;
    }

    printf("BELLMAN-FORD SHORTEST PATH ALGORITHM (algbfl.c)\n");
    printf("Finds shortest paths between a source node and all other nodes.\n");

    /* Prompt user to enter the file name of file the network is stored in.*/
    printf("Enter the arc data file name: \n");
    scanf("%s", fname);

    /* Prompt user to enter the source node and the total number of nodes */
    /* in the network. */
    printf("\nEnter the node number of the source node and the total number of nodes: \n");
    scanf("%d %d", &s, &n);

    /* Dynamically allocate the pointer, distance, queue, and pred arrays. */
    pointer = (int *) malloc((n+1) * sizeof(int));
    distance = (int *) malloc(n * sizeof(int));
    queue = (int *) malloc(n * sizeof(int));
    pred = (int *) malloc(n * sizeof(int));

    /* Open the file and read in the matrix. */
    if ( (mtxfile = fopen(fname, "r")) != NULL ) {
        /* Read in one line at a time from the file, and put it into the */
        /* oneline buffer. */
        /* index = index into target and weight arrays */
    }
index = 0;
/* Initialize all elements in pointer array to -1 (no arcs). */
for (i = 0; i < n; i++)
  *(pointer + i) = -1;

/* i is the index into the pointer array */
i = 0;
*(pointer + i) = 0;
m = 0;
while (fgets(oneline, LINELENGTH, mtxfile) != NULL) {
  /* Use strtok() function to read in one number at a time from */
  /* the oneline buffer. */
  number = strtok(oneline, seps);
  if (number == NULL)
    num = atoi(number);
  i = num;
  *(pointer + i) = index;
}
number = strtok(NULL, seps);
if (number == NULL) {
  /* Store successor node number in the target array. */
  target[index] = atoi(number);
  number = strtok(NULL, seps);
  if (number == NULL) {
    /* Store distance in the weight array. */
    weight[index] = atoi(number);
  }
}
m++;
/* if number strtok .... */
} /* while */
/* Assign index values for any remaining nodes. */
i++;
while (i <= n) {
  *(pointer + i) = index;
i++;
}
/* Change any -1 pointers to the first index of the next node. */
for (i = 0; i <= n; i++) {
  if (*(pointer + i) == -1) {
    *(pointer + i) = *(pointer + i + 1);
    dprintf("There are no arcs emanating from node %d.\n", i);
  }
}
else {
  printf("ERROR: Cannot open %s.\n", fname);
  exit(-1);
}

printf("\nNETWORK: \n");
printf("Total number of arcs in the network %d.\n", m);
printf("Total number of nodes in the network %d.\n", n);
/* Print the network out to confirm it was read in properly if the */
/* -v option was specified on the command line. */
if (verbose) {
  first = *(pointer - 1);
  last = *(pointer + i + 1) + 1;
  printf("Node %d.\n", i);
  for (j = first; j <= last; j++) {
    printf("\%d %d\n", target[j], weight[j]);
  }
}
}
/* Make sure arcs and nodes exist. */
if (m > 0 && n > 0) {
    init();
    iterate();
}

printf("RESULTS:\n");
/* Print the results in the three arrays if -v was specified. */
if (verbose) {
    /* Print out the results in the queue. */
    printf("queue[0..%d] = [",n-1);
    for (i = 0; i < n; i++) {
        if (i < n - 1)
            printf("%d, ", *(queue+i));
        else
            printf("%d\n", *(queue+i));
    }
    /* Print out the results in the distance array. */
    printf("distance[0..%d] = [",n-1);
    for (i = 0; i < n; i++) {
        if (i < n - 1)
            printf("%d, ", *(distance+i));
        else
            printf("%d\n", *(distance+i));
    }
    /* Print out the results in the pred array. */
    printf("pred[0..%d] = [",n-1);
    for (i = 0; i < n; i++) {
        if (i < n - 1)
            printf("%d, ", *(pred+i));
        else
            printf("%d\n", *(pred+i));
    }
}
/* Print the paths. */
for (i = 0; i < n; i++) {
    if (i != s && *(pred + i) != -1) trace_path(i);
    else if (i != s && *(pred + i) == -1) printf("No path from %d to %d.\n", s, i);
}
/* Free the memory that was allocated with malloc(). */
free(pointer);
free(queue);
free(distance);
free(pred);
}

init()

int i;
dprintf("\nInitialization Phase Starting.\n");
/* Set node labels to INFINITY initially. */
*(distance + i) = INFINITY;
/* Initialize queue values to -1 = node never in the queue. If */
/* queue value == 0, node was in queue previously, but is not in */
/* the queue now. If queue value > 0, node is currently in queue.*/
*(queue + i) = -1;
/* Initialize predecessor of node i to -1. */
*(pred + i) = -1;
dprintf("distance[%d] %d, queue[%d] = %d, pred[%d] %d \n", i, *(distance+i), i, *(queue+i), i, *(pred+i));
/* Label of source node is initially set to 0. */

Appendix A
iterate()
{
    int i, j, y, newlabel, first, last, next;
    dprintf("Iteration Phase Starting.\n");
    /* Loop while there are still nodes in the queue to be examined. */
    while (x != INFINITY) {
        next = *(queue + x);
        /* Pull node x off of the queue. */
        *(queue + x) = 0;
        dprintf("Node %d has been removed from the queue. The next node in the queue is
%d.\n", x, next);  
        dprintf("Examining successors of node %d.\n", x);
        first = *(pointer + x);
        last = *(pointer + x + 1)
            1;
        /* Examine all the successors of node x. */
        for (j = first; j <= last; j++) {
            /* y is the successor of node x */
            y = target[j];
            dprintf("\nNode %d is a successor of node %d.\n", y, x);
            newlabel = *(distance + x) + weight[j];
            dprintf("Node %d's current distance from node %d is %d.\n",
                y, s, *(distance + y));
            dprintf("Node %d's distance from node %d, by going thru node %d, is %d.\n",
                y, x, newlabel);
            /* Compare the current distance from node y to the source node with */
            /* the new distance computed by going from node y to the source node */
            /* THRU node x. */
            if (newlabel < *(distance + y)) {
                dprintf("Node %d's label is reduced from %d to %d.\n",
                    y, *(distance + y), newlabel);
                /* Reduce the label of node y, because it is shorter to go from the */
                /* source node to node y thru node x. */
                *(pred + y) = x;
                *(distance + y) = newlabel;
                if (*(queue + y) < 0) {
                    dprintf("Node %d was never in the queue, so it is placed at the tail
end.\n", y);
                    /* Put the node at the tail of the queue. */
                    *(queue + tail) = y;
                    tail y;
                    *(queue + tail) = INFINITY;
                    if (next == INFINITY) next y;
                }
            } else {
                if (*(queue + y) == 0) {
                    dprintf("Node %d was in queue before, but is not now.\n", y);
                    dprintf("Node %d was placed at the head of the queue.\n", y);
                    /* Put the node at the head of the queue. */
                    *(queue + y) next;
                    next y;
                } else {
                    dprintf("Node %d is already in queue.\n", y);
                }
            }
        }
    }
    /* Print out what's in the queue. */
    dprintf("Currently: queue[0..%d] = [%", n - 1);
    for (i = 0; i < n; i++) {
...

Appendix A
if (i < n - 1)
        dprintf("%d, "*(queue+i));
    else
        dprintf("%d\n", *(queue+i));
}
/* Print out what's in the distance array. */
dprintf("Currently: distance[0..%d[","n-1);
for (i = 0; i < n; i++) {
    if (i < n - 1)
        dprintf("%d, "*(distance+i));
    else
        dprintf("%d\n", *(distance+i));
}
} /* if (newlabel < *(distance + y)) */
} /* for loop */
x = next;
dprintf("\n********** The node at the head of the queue is %d.\n", x);
} /* while loop */
dprintf("Iteration Phase Complete.\n\n");
} /* iterate() */

Appendix A

void trace_path(d)
int d;
{
    printf("Distance from node %d to node %d %d\n", s, d, *(distance + d));
    printf("Shortest Path traced backwards from node %d to node %d:\n", s, d);
    while (1) {
        printf(" %d",d);
        if (*(pred + d) == s) {
            printf("%d\n", s);
        }
        else {
            d = *(pred + d);
            printf(".");
        }
    } /* while (1) */
}

algbf2.c

/** algbf2.c */
/** Susan Hojnacki */
/** April 1991 */
/** Bellman-Ford (refined by Pape) Shortest Path Algorithm to find the */
/** shortest path between all pairs of nodes in the network. This version */
/** expects the network to be described by a forward star matrix stored in */
/** a file as an arc list. Nodes in the network should be numbered */
/** starting with zero. */
/** Syntax: algbf2 [-v] */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "spath.h"
define dprintf if (verbose) printf

int *pointer;
int target[MAXARCS], weight[MAXARCS];
int *distance;
int *queue;
int s, x, n, tail, verbose;
void trace_path(int);
verbose 0;

main(argc, argv)
int argc;
char **argv;
{
    int i, j, num, m, index, first, last;
    char oneline[LINELength], fname[12], ch, seps[] " ,\t\n";
    char *number;
    FILE *mtxfile;

    if (**argv != NULL && **argv == '\n') {
        /* Get the character after the dash. */
        ch = **argv;
        if (ch == 'v') verbose 1;
    }

    printf("BELLMAN-FORD SHORTEST PATH ALGORITHM (algbf2.c)\n");
    printf("Finds shortest paths between all pairs of nodes.\n");

    /* Prompt user to enter the file name of file the network is stored in. */
    printf("Enter the arc data file name: \n");
    scanf("%s", fname);

    /* Prompt user to enter the total number of nodes in the network. */
    printf("Enter the total number of nodes in the network: \n");
    scanf("%d", &n);

    /* Dynamically allocate the pointer, distance, queue, and pred arrays. */
    pointer (int *) malloc((n+1) * sizeof(int));
    distance (int *) malloc(n * sizeof(int));
    queue (int *) malloc(n * sizeof(int));
    pred (int *) malloc(n * sizeof(int));

    /* Open the file and read in the matrix. */
    if ( mtxfile = fopen(fname, "r") != NULL ) {
        /* Read in one line at a time from the file, and put it into the */
        /* oneline buffer. */
        /* index into target and weight arrays */
        index 0;

    }
/* Initialize all elements in pointer array to -1 (no arcs). */
for (i = 0; i < n; i++)
  *(pointer + i) = -1;

/* i is the index into the pointer array */
i = 0;
*(pointer + i) = 0;
m = 0;
while (fgets(oneline, LINELENGTH, mtxfile) != NULL) {
  /* Use strtok() function to read in one number at a time from */
  /* the oneline buffer. */
  if (number = strtok(oneline, seps)) != NULL) {
    num = atoi(number);
    if (num != i) {
      i = num;
      *(pointer + i) = index;
    }
  }
  number = strtok(NULL, seps);
  if (number = NULL) {
    /* Store successor node number in the target array. */
    target[index] = atoi(number);
    number = strtok(NULL, seps);
    if (number = NULL) {
      /* Store distance in the weight array. */
      weight[index] = atoi(number);
      index++;
    }
  }
  m++;
  i++;
} /* while */
/* Assign index values for any remaining nodes. */
i++;
while (i <= n) {
  *(pointer + i) = index;
  i++;
}
/* Change any -1 pointers to the first index of the next node. */
for (i = 0; i <= n; i++) {
  if (*(pointer + i) == -1) {
    *(pointer + i) = *(pointer + i + 1);
    dprintf("There are no arcs emanating from node %d\n", i);
  }
}
}
else {
  printf("ERROR: Cannot open %s.\n",fname);
  exit(-1);
}

printf("\nNETWORK:\n");
printf("Total number of arcs in the network %d.\n",m);
printf("Total number of nodes in the network = %d.\n",n);

/* Print the network out to confirm it was read in properly if the */
/* -v option was specified on the command line. */
if (verbose) {
  for (i = 0; i < n; i++) {
    first = *(pointer + i);
    last = *(pointer + i + 1) - 1;
    printf("Node %d:\n", i);
    for (j = first; j <= last; j++) {
      printf("%d %d\n",target[j], weight[j]);
    }
  }
}
/* Make sure nodes and arcs exist. */
if (m > 0 && n > 0) {

Appendix A

for (s = 0; s < n; s++) {
    printf("\n\nSOURCE NODE %d\n", s);

    init();
    iterate();

    printf("\n\nRESULTS: \n\n");
    /* Print the results in the three arrays if -v was specified. */
    if (verbose) {
        /* Print out the results in the queue. */
        printf("queue[0..%d] = [",n-1);
        for (i = 0; i < n; i++) {
            if (i < n - 1)
                printf("%d, ", *(queue+i));
            else
                printf("%d] \n", *(queue+i));
        }
        /* Print the results in the distance array. */
        printf("distance[0..%d] = [",n-1);
        for (i = 0; i < n; i++) {
            if (i < n - 1)
                printf("%d, ", *(distance+i));
            else
                printf("%d] \n", *(distance+i));
        }
        /* Print the results in the pred array. */
        printf("pred[0..%d] = [",n-1);
        for (i = 0; i < n; i++) {
            if (i < n - 1)
                printf("%d, ", *(pred+i));
            else
                printf("%d] \n", *(pred+i));
        }
        /* Print the paths. */
        for (i = 0; i < n; i++) {
            if (i != s && *(pred + i) != -1) trace_path(i);
            if (i != s && *(pred + i) == -1) printf("No path from %d to %d\n", s, i);
        }
    }

    /* Free the memory that was allocated with malloc(). */
    free(pointer);
    free(queue);
    free(distance);
    free(pred);
}
Appendix A

/* Label of source node is initially set to 0. */
*(distance + s) = 0;
/* The source node is the only node in the queue initially. */
tail = s;
/* Node x is the node currently being examined. */
x = s;
*(queue + tail) = INFINITY;
dprintf("Initialization Phase Complete.\n\n")

iterate()
{
  int i, j, y, newlabel, first, last, next;
dprintf("Iteration Phase Starting.\n");

  /* Loop while there are still nodes in the queue to be examined. */
  while (x != INFINITY) {
    next = *(queue + x);
    /* Pull node x off the queue. */
    *(queue + x) = 0;
    dprintf("Node \(x\) has been removed from the queue. The next node in the queue is \(x\).\n\n")
    dprintf("Examining successors of node \(x\).\n\n")
    first = *(pointer + x);
    last = *(pointer + x + 1);
    /* Examine all the successors of node x. */
    for (j = first; j <= last; j++) {
      /* y is the successor of node x. */
      y = target[j];
      dprintf("\(\text{Node} \(x\) is a successor of node \(x\).\n\n")
      newlabel = *(distance + x) + weight[j];
      dprintf("\(\text{Node} \(x\)'s current distance from node \(x\) is \(y\).\n\n")
      y, s, *(distance + y);
      dprintf("\(\text{Node} \(x\)'s distance from node \(y\), by going thru node \(x\)\), is \(y\).\n\n")
      y, s, x, newlabel);
      /* Compare the current distance from node y to the source node with the */
      /* new distance computed by going from node y to the source node THRU */
      /* node x. */
      if (newlabel < *(distance + y)) {
        dprintf("Node \(y\)'s label is reduced from \(y\) to \(newlabel\).\n\n")
        y, *(distance + y), newlabel;
        /* Reduce the label of node y, because it is shorter to go from */
        /* the source node to node y thru node x. */
        *(distance + y) = newlabel;
        if (*(queue + y) < 0) {
          dprintf("Node \(y\) was never in the queue, so it is placed at the tail\n\n")
          /* Put the node at the tail of the queue. */
          *(queue + tail) = y;
          tail = y;
          *(queue + tail) = INFINITY;
          if (next == INFINITY) next = y;
        }
      } else {
        if (*(queue + y) == 0) {
          dprintf("Node \(y\) was in queue before, but is not now.\n\n")
          dprintf("Node \(y\) is placed at the head of the queue.\n\n")
          /* Put the node at the head of the queue. */
          *(queue + y) next;
          next = y;
        }
      } else {
        dprintf("Node \(y\) is already in queue.\n\n")
      }
    }
  }
}
Print out what's in the queue.

dprintf("Currently: queue[0..%d] = [",n-1);
for (i = 0; i < n; i++) {
    if (i < n - 1)
        dprintf("%d,", *(queue+i));
    else
        dprintf("%d\n", *(queue+i));
}

Print out what's in the distance array.

dprintf("Currently: distance[0..%d] = [",n-1);
for (i = 0; i < n; i++) {
    if (i < n - 1)
        dprintf("%d,", *(distance+i));
    else
        dprintf("%d\n\n", *(distance+i));
}

/* if (newlabel < *(distance + y)) */
*/
/* for loop */
x = next;
dprintf("\n*********** The node at the head of the queue is %d.\n", x);
/* while loop */
dprintf("Iteration Phase Complete.\n\n");

/* iterate() */

void trace_path(d)
int d;
{
    printf("Distance from node %d to node %d = %d\n",s, d, *(distance + d));
    printf("Shortest Path traced backwards from node %d to node %d: %n",d,s);
    while (1) {
        printf("%d",d);
        if (*(pred + d) == s) {
            printf("%d\n",s);
            break;
        }
        else {
            d = *(pred + d);
            printf("%d\n",d);
        }
    } /* while (1) */
}
% algfw.c
% /* algfw.c */
% /* Susan Hojnacki */
% /* April 1991 */
% /* Floyd-Warshall Shortest Path Algorithm */
% /* Finds the shortest paths between all pairs of nodes. The network */
% /* must be stored as a weight matrix in a file. Nodes in the network */
% /* should be numbered starting with zero. */
% /* Syntax: algfw [-v] */
% ttinclude <stdio.h>
% ttinclude <stdlib.h>
% ttinclude <string.h>
% ttinclude <math.h>
% ttdefine dprintf
% if (verbose) printf
% ttinclude "spath.h"
% #define dprintf if (verbose) printf

int matrix[MAXNODES][MAXNODES];
int path[MAXNODES][MAXNODES];
int n, ncycle, verbose;
verbose 0;

main(argc, argv)
int argc;
char **argv;
{
    int i, j, k;
    char oneline[LINELength], fname[12], ch, seps[] " ,	\n"
    char *number;
    FILE *mtxfile;

    if (**argv != NULL && **argv == ' -') {
        /* Get the character after the dash. */
        ch = *+++argv;
        if (ch == 'v') verbose 1;
    }

    printf("FLOYD-WARSHALL SHORTEST PATH ALGORITHM (algfw.c)\n");
    printf("Finds the shortest path between all pairs of nodes.\n\n");
    printf("Enter the weight matrix file name: \n");
    scanf("%s", fname);

    /* Open the file and read in the weight matrix. */
    if ( (mtxfile = fopen(fname, "r")) != NULL ) {
        FILE *mtxfile;
        /* Read in one line at a time from the file, and put it into the */
        /* one line buffer. */
        while (fgets(oneline, LINELength, mtxfile) != NULL) {
            k = 0;
            /* Use strtok() function to read in one number at a time from */
            /* the one line buffer. */
            number = strtok(oneline, seps);
            while(number != NULL) {
                /* Convert the item read in to an integer and store it in */
                /* the matrix array. */
                matrix[j][k] = atoi(number);
                k++;
                number = strtok(NULL, seps);
            }
            j++;
        } else {
            printf("ERROR: Cannot open %s.\n\n", fname);
        }
    }

Appendix A 73
exit(-1);

/* Determine n (total number of nodes in the network). */
n = j;

printf("\nNETWORK:\n");
printf("Total number of nodes in the network = %d.\n",n);

/* Print the matrix out to confirm it was read in properly if the */
/* -v option was specified on the command line. */
if (verbose)
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            printf("matrix[%d][%d] \n", i,j,matrix[i][j]);

init();
iterate();

printf("RESULTS:\n");
/* Print out the results in the path array. */
printf("WEIGHT ARRAY:\n");
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        printf("matrix[%d][%d] \n", i,j,matrix[i][j]);

/* Print out the results in the path array. */
printf("PATH ARRAY:\n");
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        printf("path[%d][%d] \n", i,j,path[i][j]);

init()
{
    int i, j;
    dprintf("\nInitialization Phase Starting.\n");
    /* Initialize negative cycle flag to false. */
    ncycle = 0;
    /* Initialize path matrix element ij to i (the predecessor node) */
    /* if the weight from i to j is not INFINITY. */
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (matrix[i][j] != INFINITY)
                path[i][j] = i;
            else
                path[i][j] = 0;

    dprintf("Initialization Phase Complete.\n\n");
}

iterate()
{
    int i, j, k, newdist;
    dprintf("Iteration Phase Starting.\n");

    /* Loop for each node, but stop if a negative cycle is found. */
    /* IF there are NO negative cycles in the network, this algorithm */
    /* can be speeded up slightly by removing the check for a negative*/
    /* cycle, and uncommenting the two IF statements that check if the*/
    /* matrix row number == the matrix column number (self loop). */
    for (k = 0; k < n && !ncycle; k++)
        for (i = 0; i < n; i++)
            if (i == k) continue;
            /* Don't try to insert node k into path from node i to node j */

Appendix A
if arc from node i to node k has a weight of infinity. */
if (matrix[i][k] != INFINITY) {
    for (j = 0; j < n; j++) {
        /* if arc from node i to node m has a weight of infinity. */
        if ((j == k) || (j == i)) continue; /*
        /* Don't try to insert node k into path from node i to node */
        /* j if arc from node k to node j has a weight of infinity. */
        if (matrix[k][j] != INFINITY) {
            newdist = matrix[i][k] + matrix[k][j];
            if (newdist < matrix[i][j]) {
                dprintf("The weight from node \%d to node \%d has been reduced from \%d to \%d.\n",
                i, j, matrix[i][j], newdist);
                /* Reduce the weight from node i to node j. */
                matrix[i][j] = newdist;
                dprintf("Node \%d has been inserted into the path from node \%d to node \%d.\n",
                        k, i, j);
                /* Insert node k into the path from node i to node j. */
                path[i][j] = path[k][j];
            }
        }
    } /* for j */
} /* for k */
/* Check for a negative-weight cycle. */
ncycle (ncycle || matrix[i][i] < 0);
if (ncycle)
    printf("A negative cycle has been found with node \%d.\n", i);
} /* for i */
} /* for k */
dprintf("Iteration Phase Complete.\n\n");
}
spath.h

#define INFINITY 10000
#define LINELENGTH 2000
#define MAXNODES 101
#define MAXARCS 3550
APPENDIX B

Program Instructions and Examples

The data for running each program must be stored in an ASCII file. Two different types of data files are used with the C programs, depending on which data structure is used by the algorithm. If a weight matrix is used, the data file contains the weight matrix, with each element in the matrix separated by at least one space. If the linked adjacency list or forward star representation is used by the algorithm, a list of the arcs in the network must be stored in a file. The list has one row for each arc (numbered starting with 0, not 1). Each row contains the originating node of the arc, the target node of the arc, and the weight of the arc. Each value is separated by at least one space. Each arc in the network must be on a separate row in the data file.

The following examples are used in the program instructions presented in this appendix.

Example #1: All Arcs Have Positive Weights

This example will be used to illustrate the Dijkstra Shortest Path Algorithm and its variations. A computer network is used to transfer data between different sites of a large company. A node is called an IMP. Each IMP keeps information about the network in tables. One piece of information that an IMP stores is the transmission delay.
The transmission delay is measured in msec. Each IMP periodically sends out a test packet to determine the transmission delay for each arc in the network. The path from the source IMP to the destination IMP that has the smallest transmission delay will be the fastest ("shortest") path. There are 12 nodes and 26 arcs in this example. A directed graph depicting this network is shown in Figure B.1.

![Digraph for Example #1](image)

Figure B.1
Digraph for Example #1
The weight matrix for this network is:

\[
\begin{array}{cccccccccccccccc}
0 & 22 & 10000 & 10000 & 10000 & 24 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000

10000 & 0 & 10000 & 10000 & 10000 & 28 & 10000 & 10000 & 10000 & 10000 & 10000

10000 & 35 & 0 & 10000 & 42 & 10000 & 25 & 10000 & 10000 & 10000 & 10000 & 10000

10000 & 10000 & 36 & 0 & 35 & 10000 & 10000 & 24 & 10000 & 10000 & 10000 & 10000

10000 & 10000 & 10000 & 10000 & 10000 & 0 & 10000 & 10000 & 38 & 10000 & 10000 & 10000

10000 & 10000 & 10000 & 10000 & 10000 & 0 & 23 & 10000 & 10000 & 32 & 10000 & 10000

32 & 10000 & 10000 & 33 & 10000 & 10000 & 0 & 27 & 10000 & 10000 & 44 & 10000

10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 0 & 36 & 10000 & 10000 & 21 & 10000

10000 & 10000 & 10000 & 10000 & 10000 & 41 & 10000 & 10000 & 10000 & 0 & 10000 & 10000 & 40

34 & 10000 & 10000 & 10000 & 10000 & 10000 & 27 & 10000 & 10000 & 0 & 38 & 10000

10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 0 & 20

10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 35 & 10000 & 31 & 10000 & 0
\end{array}
\]

The contents of the data file that is used for a linked adjacency list or forward star data structure are:

\[
\begin{array}{cccc}
0 & 1 & 22 \\
0 & 5 & 24 \\
1 & 6 & 28 \\
2 & 1 & 35 \\
2 & 4 & 42 \\
2 & 6 & 25 \\
3 & 2 & 36 \\
3 & 4 & 35 \\
3 & 7 & 24 \\
4 & 7 & 38 \\
5 & 6 & 23 \\
5 & 9 & 32 \\
6 & 0 & 32 \\
6 & 3 & 33 \\
6 & 7 & 27 \\
6 & 10 & 44 \\
7 & 8 & 36 \\
7 & 10 & 21 \\
8 & 4 & 41 \\
8 & 11 & 40 \\
9 & 0 & 34 \\
9 & 6 & 27 \\
9 & 10 & 38 \\
10 & 11 & 20 \\
11 & 7 & 35 \\
11 & 9 & 31 \\
\end{array}
\]
Example #2: Some Arcs With Negative Weights

A salesman wants to travel from Seattle to New York City to visit some important clients. He would also like to visit some other customers along his way. In advance, the salesman can compute an expected commission from each customer in each city he visits. He can also compute the expected traveling cost of each leg of the trip. Cities are represented by nodes in the graph in Figure B.2. The weight of each arc represents the net cost (traveling expenses - estimated commission) of taking that route. Therefore, an arc weight will be negative whenever the salesman expects to have a commission larger than the cost of his traveling expenses for that leg of the trip. The salesman wants to minimize the total net cost of his trip.

This example will be used to illustrate the Bellman-Ford Algorithm and the Floyd-Warshall Algorithm.
The weight matrix for this network is:

\[
\begin{array}{cccccccccccc}
0 & -173 & 10000 & -147 & 10000 & 641 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 \\
10000 & 10000 & 0 & -166 & -597 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 \\
10000 & 10000 & 10000 & 0 & -1216 & -159 & 56 & 16 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 \\
10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 10000 & 0 \\
\end{array}
\]
The contents of the data file that is used for a linked adjacency list or forward star data structure are:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>-173</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>47</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>641</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-397</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>-30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>166</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>-597</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>-1216</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>-159</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>56</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>-155</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>-695</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>185</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>458</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>-709</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>-942</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>-225</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>-805</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>211</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>86</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>-227</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td>-2333</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>447</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>13</td>
<td>-2145</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>-195</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>-2908</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>-2612</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example #3: A Network With A Negative Cycle

This example will be used to illustrate the effect of a negative cycle on the Floyd-Warshall Algorithm. The network in Figure B.3 has five nodes and eight arcs. A negative cycle can be traced from node 1, to node 3, to node 2, and back to node 1. When the weights of this cycle are summed, the result is -2.

![Figure B.3 Digraph for Example #3](image)

The weight matrix for this network is:

```
0  7 10000 10000 10000
10000 0 10000 5 10000
10000 -4 0 10000 -1
10000 10000 -3 0 2
6 4 10000 10000 0
```
I. Dijkstra's Shortest Path Algorithm

Instructions for Running algd1.c

This program finds the shortest path between a single source node and a single destination node in a network using Dijkstra's Algorithm. Before running the C program, you must have a weight matrix describing the network stored in ASCII format in a file. The weight matrix has one row for each node (numbered starting with 0, not 1). Each row of the matrix must be on a separate row in the file and each column of the matrix must be separated by at least one space.

To run the program, type:

```
algd1 [-v]
```

and press ENTER. The -v option may be passed to the program on the command line. This is the verbose option. When the -v switch is used, informational messages explaining the progress of the algorithm are printed as the program executes.

The program will prompt you to enter the name of the file in which the matrix is stored.

`Enter the weight matrix file name:`
Include the directory path if the file is not in the current directory. Next, the program will prompt you to enter the source and destination nodes.

Enter the node numbers of the source node and destination node:

Type the node numbers of the source and destination node separated by at least one space. For example, if the source node is node 0 and the destination node is node 11 (see Example #1), type:

0 11

and press ENTER.

The total number of nodes in the network will be printed out. Also, the program will print informational messages as it runs if the -v option was specified on the command line. Additionally, if the -v option was specified, the results of three arrays will be displayed when the program is finished. These arrays are called permanent, weight, and pred. Each array has one element per node and is indexed by node number. The array called permanent tells you which nodes were permanently labeled. Each element can be a 1 or a 0. A 1 in the permanent array represents a permanent label for that node.

The second array is called weight. It lists the shortest weights (distances) found from the source node to the other nodes in the network. This number is accurate for the destination node, as well as all the nodes that are permanently labeled.
when the program halts. A value of \texttt{INFINITY} (as defined in the \#define preprocessor statement at the top of the program to be 10000) in the \texttt{weight} array for a particular node signifies that there isn't any path between the source node and that particular node.

The third array listed is called \texttt{pred} and it is the array used to store the predecessors during program execution. The shortest path from source to destination can be found by tracing backwards through the array from destination to source.

\[ d, \ pred[d], \ pred[pred[d]], \ldots, \ pred[...], \ s \]

For example, if the source node is 0, the destination node is 11, and the \texttt{pred} array is:

\[
\begin{array}{c|cccccccccccc}
\text{node} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\hline
\text{pred}\{0..11\} & -1 & 0 & 3 & 6 & 3 & 0 & 5 & 6 & 7 & 5 & 6 & 10 \\
\end{array}
\]

the path can be traced backwards from the destination node as follows:

\[ 11, \ pred[11], \ pred[pred[11]] \ldots \]

\[ 11, \ 10, \ 6, \ 5, \ 0 \]

So the path from source to destination would be 0, 5, 6, 10, 11. The function \texttt{trace_path()} determines the shortest path and displays the results at the end of the output listing.
Whether or not the \texttt{-v} option is specified on the command line, the following results will be displayed. If a shortest path can be found, the distance (total path weight) from source to destination is displayed. If no path can be found, the element corresponding to the destination node in the weight array will be equal to \texttt{INFINITY}. In this case, a message will be printed at the end of the output listing stating that no path was found.

The following listing is the result of running \texttt{algdl.c} with Example \#1.

\begin{verbatim}
DIJKSTRA'S SHORTEST PATH ALGORITHM (algdl.c)
Enter the weight matrix file name: mat12s.mtx
Enter the node numbers of the source node and destination node: 0 11
NETWORK: Total number of nodes in the network 12.
RESULTS: The shortest path has been found.
Distance (weight) from node 0 to node 11 111
Shortest Path traced backwards from node 11 to node 0: 11, 10, 6, 5, 0
\end{verbatim}
Instructions for Running algd2.c

This program finds the shortest path between a single source node and a single destination node using Dijkstra’s Algorithm. Before running this C program, you must have an ASCII file describing the network. The file has one row for each arc (numbered starting with 0, not 1). Each row contains the originating node of the arc, the target node of the arc, and the weight of the arc. Each arc in the network must be on a separate row in the file and each value on each row must be separated by at least one space.

To run the program, type:

```
algd2 [-v]
```

and press ENTER. The -v option may be passed to the program on the command line. This is the verbose option. When the -v switch is used, informational messages explaining the progress of the algorithm are printed as the program executes.

The program will prompt you to enter the name of the file in which the arc data is stored.

Enter the arc data file name:

Include the directory path if the file is not in the current directory. Next, the program will prompt you to enter the source node, destination node, and total number of nodes.
Enter the node numbers of the source node and destination node, and the total number of nodes:

Type the node number of the source, the node number of the destination node, and the total number of nodes in the network. Separate the values by at least one space. For example, if the source node is node 0, the destination node is node 11, and the total number of nodes in the network is 12, type:

0 11 12

and press ENTER.

The total number of arcs and the total number of nodes in the network will be printed out. The program will print informational messages as it runs if the -v option was specified on the command line. Additionally, if the -v option was specified, the results of three arrays will be displayed when the program is finished. These arrays are called permanent, weight, and pred. Each array has one element per node and is indexed on node number. The array called permanent tells you which nodes were permanently labeled. Each element in the permanent array can be either a 1 or a 0. A 1 in the permanent array represents a permanent label for that node.
The second array is called weight. It lists the shortest weights (distances) found from the source node to the other nodes in the network. This number is accurate for the destination node, as well as all the nodes that are permanently labeled when the program halts. A value of INFINITY (as defined in the #define preprocessor statement at the top of the program to be 10000) in the weight array for a particular node signifies that there isn’t any path between the source node and that particular node.

The third array listed is called pred and it is the array used to store the predecessors during program execution. The shortest path from source to destination can be found by tracing backwards through the array from destination to source:

d, pred[d], pred[pred[d]], ..., pred[...], s

For example, if the source node is 0, the destination node is 11, and the pred array is:

```
node:    0 1 2 3 4 5 6 7 8 9 10 11
pred[0..11] = [-1, 0, 3, 6, 3, 0, 5, 6, 7, 5, 6, 10]
```

the path can be traced backwards from the destination node as follows:

```
11, pred[11], pred[pred[11]] ...
11, 10, 6, 5, 0
```
So the path from source to destination would be 0, 5, 6, 10, 11. The function `trace_path()` determines the shortest path and displays the results at the end of the output listing.

Whether or not the -v option is used, the following results will be displayed. If a shortest path can be found, the distance (total path weight) from source to destination is displayed. If no path can be found, the element corresponding to the destination node in the weight array will be equal to INFINITY. In this case, a message will be printed at the end of the output listing stating that no path was found.

The following listing shows the results of running algd2.c with Example #1.

```
DIJKSTRA'S SHORTEST PATH ALGORITHM (algd2.c)

Enter the arc data file name: adj12s.mtx

Enter the node numbers of the source node and destination node, and the total number of nodes: 0 11 12

NETWORK:
Total number of arcs in the network = 26.
Total number of nodes in the network = 12.

The shortest path has been found.
Distance (weight) from node 0 to node 11 = 111
Shortest Path traced backwards from node 11 to node 0: 11, 10, 6, 5, 0
```
Instructions for Running `algd3.c`

This program finds the shortest path between a single source node and every node in the network using Dijkstra's Algorithm. Before running the C program, you must have a weight matrix describing the network stored in ASCII format in a file. The weight matrix has one row for each node (numbered starting with 0, not 1). Each row of the matrix must be on a separate row in the file and each column of the matrix must be separated by at least one space.

To run the program, type:

```
algd3 [-v]
```

and press ENTER. The `-v` option may be passed to the program on the command line. This is the `verbose` option. When the `-v` switch is used, informational messages explaining the progress of the algorithm are printed as the program executes.

The program will prompt you to enter the name of the file in which the matrix is stored.

*Enter the weight matrix file name:*

Include the directory path if the file is not in the current directory. Next, the program will prompt you to enter the source node.
Enter the node number of the source node:

Type the node number of the source node. For example, if the source node is node 0, type:

0

and press ENTER.

The total number of nodes in the network will be printed out. The program will print informational messages as it runs if the -v option was specified on the command line. Additionally, if the -v option was specified, the results of three arrays will be displayed when the program is finished. These arrays are called permanent, weight, and pred. Each array has one element per node and is indexed on node number. The array called permanent tells you which nodes were permanently labeled. An element in the pred array can be a 1 or a 0. A 1 in the permanent array represents a permanent label for that node.

The second array is called weight. It lists the shortest weights (distances) found from the source node to the other nodes in the network. This number is accurate for all nodes when the program halts. A value of INFINITY (as defined in the #define preprocessor statement at the top of the program to be 10000) in the weight array for a particular node signifies that there isn’t any path between the source node and that particular node.
The third array listed is called pred and it is the array used to store the predecessors during program execution. The shortest path from source to destination can be found by tracing backwards through the array from destination to source:

d, pred[d], pred[pred[d]], ..., pred[...], s

For example, if the source node is 0, the destination node is 11, and the pred array is:

node: 0 1 2 3 4 5 6 7 8 9 10 11
pred[0..11] = [-1, 0, 3, 6, 3, 0, 5, 6, 7, 5, 6, 10]

the path can be traced backwards from the destination node as follows:

11, pred[11], pred[pred[11]] ...

11, 10, 6, 5, 0

So the path from source to destination would be 0, 5, 6, 10, 11. The function trace_path() determines the shortest path and displays the results at the end of the output listing.

Whether or not the -v options is specified, the following results will be displayed. If a shortest path can be found, the distance (total path weight) from source to destination is displayed. If no path can be found, the element corresponding to the destination node in the weight array will be equal to INFINITY. In this case, a message will be printed stating that no path was found.
Instructions for Running algd4.c

This program finds the shortest path between all pairs of nodes in the network using Dijkstra's Algorithm. Before running the C program, you must have a weight matrix describing the network stored in an ASCII file. The weight matrix has one row for each node (numbered starting with 0, not 1). Each row of the matrix must be on a separate row in the file and each column of the matrix must be separated by at least one space.

To run the program, type:

```
algd4 [-v]
```

and press ENTER. The -v option may be passed to the program on the command line. This is the verbose option. When the -v switch is used, informational messages explaining the progress of the algorithm are printed as the program executes.

The program will prompt you to enter the name of the file in which the matrix is stored.

*Enter the weight matrix file name:*

Include the directory path if the file is not in the current directory.
The total number of nodes in the network will be printed out. The program will print informational messages as it runs if the -v option was specified on the command line. Additionally, if the -v option was specified, the results of three arrays will be displayed when the program is finished processing a node as the source node. These arrays are called permanent, weight, and pred. Each array has one element per node and is indexed on node number. The array called permanent tells you which nodes were permanently labeled. A 1 in the permanent array represents a permanent label for that node.

The second array is called weight. It lists the shortest weights (distances) found from the source node to the other nodes in the network. This number is accurate for all nodes when the program halts. A value of INFINITY (as defined in the #define preprocessor statement at the top of the program to be 10000) in the weight array for a particular node signifies that there isn’t any path between the source node and that particular node.

The third array listed is called pred and it is the array used to store the predecessors during program execution. The shortest path from source to destination can be found by tracing backwards through the array from destination to source:

\[ d, \text{pred}[d], \text{pred}[\text{pred}[d]], \ldots, \text{pred}[\ldots], s \]
For example, if the source node is 0 and the \texttt{pred} array is:

\begin{verbatim}
node:      0 1 2 3 4 5 6 7 8 9 10 11
pred[0..11] = [-1, 0, 3, 6, 3, 0, 5, 6, 7, 5, 6, 10]
\end{verbatim}

the path can be traced backwards from node 11 to node 0 as follows:

11, \texttt{pred[11]}, \texttt{pred[pred[11]]} ...

11, 10, 6, 5, 0

So the path from source to node 11 would be 0, 5, 6, 10, 11. The function \texttt{trace_path()} determines the shortest path and displays the results at the end of the output listing for each source node.

The following results will be displayed whether or not the \texttt{-v} is specified on the command line. If a shortest path can be found, the distance (total path weight) from source to destination is displayed. If no path can be found, the element corresponding to the destination node in the \texttt{weight} array will be equal to \texttt{INFINITY}. In this case, a message will be printed stating that no path was found.
II. Bellman-Ford Algorithm

Instructions for Running algbfl.c

This program finds the shortest path between a single source node and every node in the network using the Bellman-Ford Algorithm. Before running this C program, you must have an ASCII file describing the network. The file has one row for each arc (numbered starting with 0, not 1). Each row contains the originating node of the arc, the target node of the arc, and the weight of the arc. Each arc of the network must be on a separate row in the file and each value on each line must be separated by at least one space.

To run the program, type:

`algbfl [-v]`

and press ENTER. The -v option may be passed to the program on the command line. This is the verbose option. When the -v switch is used, informational messages explaining the progress of the algorithm are printed as the program executes.

The program will prompt you to enter the name of the file in which the arc data is stored.

*Enter the arc data file name:*
Include the directory path if the file is not in the current directory. Next, the program will prompt you to enter the source node and the total number of nodes.

*Enter the node number of the source node and the total number of nodes:*

Type the node number of the source node and the total number of nodes in the network. Separate the values by at least one space. In Example #2, the source node is 0 and there are 14 nodes in the network, so type:

```
0 14
```

and press ENTER.

The total number of arcs and the total number of nodes in the network will be printed out. The program will print informational messages as it runs if the -v option was specified on the command line. Additionally, if the -v option was specified, the results of three arrays will be displayed when the program is finished. These arrays are called queue, distance, and pred. Each array has one element per node and is indexed on node number. The array called queue lists the elements in the queue after the program has terminated.

The second array is called distance. It lists the shortest distances found from the source node to the other nodes in the network. A value of INFINITY (as defined in the #define preprocessor statement at the top of the program to be 10000) in

Appendix B
the distance array for a particular node signifies that there isn't any path between
the source node and that particular node.

The third array listed is called pred and it is the array used to store the
predecessors during program execution. The shortest path from source to
destination can be found by tracing backwards through the array from destination
to source.

d, pred[d], pred[pred[d]], ..., pred[...], s

For example, for source node equal to 0 in the example, the pred array is:

node: 0 1 2 3 4 5 6 7 8 9 10 11 12 13
pred[0..13] = [-1, 0, 1, 2, 3, 3, 4, 6, 7, 7, 9, 10, 8, 11]

and the path can be traced backwards from node 13 to node 0 as follows:

13, pred[13], pred[pred[13]] ...
13, 11, 10, 9, 7, 6, 4, 3, 2, 1, 0

So the path from source to node 13 would be 0, 1, 2, 3, 4, 6, 7, 9, 10, 11, 13. The
function trace_path() determines the shortest path and displays the results at the
deck of the output listing.
The following results will be displayed whether or not the \(-v\) option was specified on the command line. If no path can be found, the element corresponding to the destination node in the distance array will be equal to INFINITY. In this case, a message will be printed at the end of the output listing stating that no path was found for that source/destination combination.

The results below were obtained when the algbfl.c program was run with Example #2.

**BELLMAN-FORD SHORTEST PATH ALGORITHM** (algbfl.c)
Finds shortest paths between a source node and all other nodes.

Enter the arc data file name:
adj14s.mtx

Enter the node number of the source node and the total number of nodes:
0 14

**NETWORK:**
Total number of arcs in the network = 28.
Total number of nodes in the network = 14.

**RESULTS:**
Distance from node 0 to node 1 = -173
Shortest Path traced backwards from node 1 to node 0:
   1, 0

Distance from node 0 to node 2 = -570
Shortest Path traced backwards from node 2 to node 0:
   2, 1, 0

Distance from node 0 to node 3 = -736
Shortest Path traced backwards from node 3 to node 0:
   3, 2, 1, 0

Distance from node 0 to node 4 = -1952
Shortest Path traced backwards from node 4 to node 0:
   4, 3, 2, 1, 0

Distance from node 0 to node 5 = -895
Shortest Path traced backwards from node 5 to node 0:
   5, 3, 2, 1, 0

Distance from node 0 to node 6 = -2107
Shortest Path traced backwards from node 6 to node 0:
   6, 4, 3, 2, 1, 0

Distance from node 0 to node 7 = -2816
Shortest Path traced backwards from node 7 to node 0:
7, 6, 4, 3, 2, 1, 0

Distance from node 0 to node 8 = -3041
Shortest Path traced backwards from node 8 to node 0:
8, 7, 6, 4, 3, 2, 1, 0

Distance from node 0 to node 9 = -3621
Shortest Path traced backwards from node 9 to node 0:
9, 7, 6, 4, 3, 2, 1, 0

Distance from node 0 to node 10 = -3174
Shortest Path traced backwards from node 10 to node 0:
10, 9, 7, 6, 4, 3, 2, 1, 0

Distance from node 0 to node 11 = -3369
Shortest Path traced backwards from node 11 to node 0:
11, 10, 9, 7, 6, 4, 3, 2, 1, 0

Distance from node 0 to node 12 = -3268
Shortest Path traced backwards from node 12 to node 0:
12, 8, 7, 6, 4, 3, 2, 1, 0

Distance from node 0 to node 13 = -6277
Shortest Path traced backwards from node 13 to node 0:
13, 11, 10, 9, 7, 6, 4, 3, 2, 1, 0
Instructions for Running algbf2.c

This program finds the shortest paths between all pairs of nodes in the network using the Bellman-Ford Algorithm. Before running this C program, you must have an ASCII file describing the network. The file has one row for each arc (numbered starting with 0, not 1). Each row contains the originating node of the arc, the target node of the arc, and the weight of the arc. Each arc of the network must be on a separate row in the file and each value on each line must be separated by at least one space.

To run the program, type:

    algbf2 [-v]

and press ENTER. The -v option may be passed to the program on the command line. This is the verbose option. When the -v switch is used, informational messages explaining the progress of the algorithm are printed as the program executes.

The program will prompt you to enter the name of the file in which the arc data is stored.

    Enter the arc data file name:

Include the directory path if the file is not in the current directory. Next, the program will prompt you to enter the total number of nodes.
Enter the total number of nodes:

Type the number of nodes in the network. In Example #2, the total number of nodes in the network is 14, so type:

14

and press ENTER.

The total number of arcs and the total number of nodes in the network will be printed out. The program will print informational messages as it runs if the -v option was specified on the command line. Additionally, if the -v option was specified, the results of three arrays will be displayed when the program is finished processing a node as the source node. These arrays are called queue, distance, and pred. Each array has one element per node and is indexed on node number. The array called queue lists the elements in the queue after the program has terminated.

The second array is called distance. It lists the shortest distances found from the source node to the other nodes in the network. A value of INFINITY (as defined in the #define preprocessor statement at the top of the program to be 10000) in the distance array for a particular node signifies that there isn’t any path between the source node and that particular node.
The third array listed is called *pred* and it is the array used to store the predecessors during program execution. The shortest path from source to destination can be found by tracing backwards through the array from destination to source.

\[ d, \text{pred}[d], \text{pred}[	ext{pred}[d]], \ldots, \text{pred}[\ldots], s \]

For example, for source node equal to 0 in the example, the *pred* array is:

<table>
<thead>
<tr>
<th>node:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>pred[0..13]</td>
<td>-1, 0, 1, 2, 3, 3, 4, 6, 7, 7, 9, 10, 8, 11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

and the path can be traced backwards from node 13 to node 0 as follows:

\[ 13, \text{pred}[13], \text{pred}[	ext{pred}[13]] \ldots \]

\[ 13, 11, 10, 9, 7, 6, 4, 3, 2, 1, 0 \]

So the path from source to node 13 would be 0, 1, 2, 3, 4, 6, 7, 9, 10, 11, 13. The function *trace_path()* determines the shortest path and displays the results at the end of the output listing.

The following results will be displayed whether or not the -v option was specified on the command line. If no path can be found, the element corresponding to the destination node in the distance array will be equal to INFINITY. In this case, a message will be printed at the end of the output listing stating that no path was found for that source/destination combination.
The following listing of results are printed out when the algbf2.c program is run with Example #2.

**BELLMAN-FORD SHORTEST PATH ALGORITHM** (algbf2.c)
Finds shortest paths between all pairs of nodes.

Enter the arc data file name:
adj14s.mtx

Enter the total number of nodes in the network:
14

**NETWORK:**
Total number of arcs in the network = 28.
Total number of nodes in the network = 14.

**SOURCE NODE = 0.**

**RESULTS:**
Distance from node 0 to node 1 = -173
Shortest Path traced backwards from node 1 to node 0:
1, 0

Distance from node 0 to node 2 = -570
Shortest Path traced backwards from node 2 to node 0:
2, 1, 0

Distance from node 0 to node 3 = 736
Shortest Path traced backwards from node 3 to node 0:
3, 2, 1, 0

Distance from node 0 to node 4 = -1952
Shortest Path traced backwards from node 4 to node 0:
4, 3, 2, 1, 0

Distance from node 0 to node 5 = -895
Shortest Path traced backwards from node 5 to node 0:
5, 3, 2, 1, 0

Distance from node 0 to node 6 = -2107
Shortest Path traced backwards from node 6 to node 0:
6, 4, 3, 2, 1, 0

Distance from node 0 to node 7 = -2816
Shortest Path traced backwards from node 7 to node 0:
7, 6, 4, 3, 2, 1, 0

Distance from node 0 to node 8 = -3041
Shortest Path traced backwards from node 8 to node 0:
8, 7, 6, 4, 3, 2, 1, 0

Distance from node 0 to node 9 = -3621
Shortest Path traced backwards from node 9 to node 0:
9, 7, 6, 4, 3, 2, 1, 0

Distance from node 0 to node 10 = -3174
Shortest Path traced backwards from node 10 to node 0:
10, 9, 7, 6, 4, 3, 2, 1, 0
Distance from node 0 to node 11 = -3369
Shortest Path traced backwards from node 11 to node 0:
   11, 10, 9, 7, 6, 4, 3, 2, 1, 0

Distance from node 0 to node 12 = -3268
Shortest Path traced backwards from node 12 to node 0:
   12, 8, 7, 6, 4, 3, 2, 1, 0

Distance from node 0 to node 13 = -6277
Shortest Path traced backwards from node 13 to node 0:
   13, 11, 10, 9, 7, 6, 4, 3, 2, 1, 0

SOURCE NODE 1.
RESULTS:
No path from 1 to 0.

Distance from node 1 to node 2 = -397
Shortest Path traced backwards from node 2 to node 1:
   2, 1

Distance from node 1 to node 3 = -563
Shortest Path traced backwards from node 3 to node 1:
   3, 2, 1

Distance from node 1 to node 4 = -1779
Shortest Path traced backwards from node 4 to node 1:
   4, 3, 2, 1

Distance from node 1 to node 5 = -722
Shortest Path traced backwards from node 5 to node 1:
   5, 3, 2, 1

Distance from node 1 to node 6 = -1934
Shortest Path traced backwards from node 6 to node 1:
   6, 4, 3, 2, 1

Distance from node 1 to node 7 = -2643
Shortest Path traced backwards from node 7 to node 1:
   7, 6, 4, 3, 2, 1

Distance from node 1 to node 8 = -2868
Shortest Path traced backwards from node 8 to node 1:
   8, 7, 6, 4, 3, 2, 1

Distance from node 1 to node 9 = -3448
Shortest Path traced backwards from node 9 to node 1:
   9, 7, 6, 4, 3, 2, 1

Distance from node 1 to node 10 = -3001
Shortest Path traced backwards from node 10 to node 1:
   10, 9, 7, 6, 4, 3, 2, 1

Distance from node 1 to node 11 = -3196
Shortest Path traced backwards from node 11 to node 1:
   11, 10, 9, 7, 6, 4, 3, 2, 1

Distance from node 1 to node 12 = -3095
Shortest Path traced backwards from node 12 to node 1:
   12, 8, 7, 6, 4, 3, 2, 1
Distance from node 1 to node 13 = -6104
Shortest Path traced backwards from node 13 to node 1:
   13, 11, 10, 9, 7, 6, 4, 3, 2, 1

SOURCE NODE = 2.
RESULTS:
No path from 2 to 0.
No path from 2 to 1.
Distance from node 2 to node 3 = -166
Shortest Path traced backwards from node 3 to node 2:
   3, 2
Distance from node 2 to node 4 = -1382
Shortest Path traced backwards from node 4 to node 2:
   4, 3, 2
Distance from node 2 to node 5 = -325
Shortest Path traced backwards from node 5 to node 2:
   5, 3, 2
Distance from node 2 to node 6 = -1537
Shortest Path traced backwards from node 6 to node 2:
   6, 4, 3, 2
Distance from node 2 to node 7 = -2246
Shortest Path traced backwards from node 7 to node 2:
   7, 6, 4, 3, 2
Distance from node 2 to node 8 = -2471
Shortest Path traced backwards from node 8 to node 2:
   8, 7, 6, 4, 3, 2
Distance from node 2 to node 9 = -3051
Shortest Path traced backwards from node 9 to node 2:
   9, 7, 6, 4, 3, 2
Distance from node 2 to node 10 = -2604
Shortest Path traced backwards from node 10 to node 2:
   10, 9, 7, 6, 4, 3, 2
Distance from node 2 to node 11 = -2799
Shortest Path traced backwards from node 11 to node 2:
   11, 10, 9, 7, 6, 4, 3, 2
Distance from node 2 to node 12 = -2698
Shortest Path traced backwards from node 12 to node 2:
   12, 8, 7, 6, 4, 3, 2
Distance from node 2 to node 13 = -5707
Shortest Path traced backwards from node 13 to node 2:
   13, 11, 10, 9, 7, 6, 4, 3, 2

SOURCE NODE = 3.
RESULTS:
No path from 3 to 0.
No path from 3 to 1.
No path from 3 to 2.

Distance from node 3 to node 4 = -1216
Shortest Path traced backwards from node 4 to node 3: 4, 3

Distance from node 3 to node 5 = -159
Shortest Path traced backwards from node 5 to node 3: 5, 3

Distance from node 3 to node 6 = -1371
Shortest Path traced backwards from node 6 to node 3: 6, 4, 3

Distance from node 3 to node 7 = -2080
Shortest Path traced backwards from node 7 to node 3: 7, 6, 4, 3

Distance from node 3 to node 8 = -2305
Shortest Path traced backwards from node 8 to node 3: 8, 7, 6, 4, 3

Distance from node 3 to node 9 = -2885
Shortest Path traced backwards from node 9 to node 3: 9, 7, 6, 4, 3

Distance from node 3 to node 10 = -2438
Shortest Path traced backwards from node 10 to node 3: 10, 9, 7, 6, 4, 3

Distance from node 3 to node 11 = -2633
Shortest Path traced backwards from node 11 to node 3: 11, 10, 9, 7, 6, 4, 3

Distance from node 3 to node 12 = -2532
Shortest Path traced backwards from node 12 to node 3: 12, 8, 7, 6, 4, 3

Distance from node 3 to node 13 = -5541
Shortest Path traced backwards from node 13 to node 3: 13, 11, 10, 9, 7, 6, 4, 3

SOURCE NODE  4.
RESULTS:
No path from 4 to 0.
No path from 4 to 1.
No path from 4 to 2.
No path from 4 to 3.
No path from 4 to 5.

Distance from node 4 to node 6 = -155
Shortest Path traced backwards from node 6 to node 4: 6, 4

Distance from node 4 to node 7 = -864
Shortest Path traced backwards from node 7 to node 4: 7, 6, 4

Distance from node 4 to node 8 = -1089
Shortest Path traced backwards from node 8 to node 4: 8, 7, 6, 4

Distance from node 4 to node 9 = -1669
Shortest Path traced backwards from node 9 to node 4: 9, 7, 6, 4

Distance from node 4 to node 10 = -1222
Shortest Path traced backwards from node 10 to node 4: 10, 9, 7, 6, 4

Distance from node 4 to node 11 = -1417
Shortest Path traced backwards from node 11 to node 4: 11, 10, 9, 7, 6, 4

Distance from node 4 to node 12 = -1316
Shortest Path traced backwards from node 12 to node 4: 12, 8, 7, 6, 4

Distance from node 4 to node 13 = -4325
Shortest Path traced backwards from node 13 to node 4: 13, 11, 10, 9, 7, 6, 4

SOURCE NODE = 5.

RESULTS:
No path from 5 to 0.

No path from 5 to 1.

No path from 5 to 2.

No path from 5 to 3.

No path from 5 to 4.

No path from 5 to 6.

No path from 5 to 7.

Distance from node 5 to node 8 = 185
Shortest Path traced backwards from node 8 to node 5: 8, 5

No path from 5 to 9.

No path from 5 to 10.

Distance from node 5 to node 11 = 271
Shortest Path traced backwards from node 11 to node 5: 11, 8, 5
Distance from node 5 to node 12 = -42
Shortest Path traced backwards from node 12 to node 5: 12, 8, 5

Distance from node 5 to node 13 = -2654
Shortest Path traced backwards from node 13 to node 5: 13, 12, 8, 5

SOURCE NODE = 6.
RESULTS:
No path from 6 to 0.
No path from 6 to 1.
No path from 6 to 2.
No path from 6 to 3.
No path from 6 to 4.
No path from 6 to 5.

Distance from node 6 to node 7 = -709
Shortest Path traced backwards from node 7 to node 6: 7, 6

Distance from node 6 to node 8 = -934
Shortest Path traced backwards from node 8 to node 6: 8, 7, 6

Distance from node 6 to node 9 = -1514
Shortest Path traced backwards from node 9 to node 6: 9, 7, 6

Distance from node 6 to node 10 = -1067
Shortest Path traced backwards from node 10 to node 6: 10, 9, 7, 6

Distance from node 6 to node 11 = -1262
Shortest Path traced backwards from node 11 to node 6: 11, 10, 9, 7, 6

Distance from node 6 to node 12 = -1161
Shortest Path traced backwards from node 12 to node 6: 12, 8, 7, 6

Distance from node 6 to node 13 = -4170
Shortest Path traced backwards from node 13 to node 6: 13, 11, 10, 9, 7, 6

SOURCE NODE = 7.
RESULTS:
No path from 7 to 0.
No path from 7 to 1.
No path from 7 to 2.
No path from 7 to 3.
No path from 7 to 4.
No path from 7 to 5.
No path from 7 to 6.

Distance from node 7 to node 8 = -225
Shortest Path traced backwards from node 8 to node 7:
\[ 8, 7 \]

Distance from node 7 to node 9 = -805
Shortest Path traced backwards from node 9 to node 7:
\[ 9, 7 \]

Distance from node 7 to node 10 = -358
Shortest Path traced backwards from node 10 to node 7:
\[ 10, 9, 7 \]

Distance from node 7 to node 11 = -553
Shortest Path traced backwards from node 11 to node 7:
\[ 11, 10, 9, 7 \]

Distance from node 7 to node 12 = -452
Shortest Path traced backwards from node 12 to node 7:
\[ 12, 8, 7 \]

Distance from node 7 to node 13 = -3461
Shortest Path traced backwards from node 13 to node 7:
\[ 13, 11, 10, 9, 7 \]

SOURCE NODE = 8.
RESULTS:
No path from 8 to 0.
No path from 8 to 1.
No path from 8 to 2.
No path from 8 to 3.
No path from 8 to 4.
No path from 8 to 5.
No path from 8 to 6.
No path from 8 to 7.
No path from 8 to 9.
No path from 8 to 10.

Distance from node 8 to node 11 = 86
Shortest Path traced backwards from node 11 to node 8:
\[ 11, 8 \]
Distance from node 8 to node 12 = -227
Shortest Path traced backwards from node 12 to node 8:
   12, 8

Distance from node 8 to node 13 = -2839
Shortest Path traced backwards from node 13 to node 8:
   13, 12, 8

SOURCE NODE = 9.
RESULTS:
No path from 9 to 0.
No path from 9 to 1.
No path from 9 to 2.
No path from 9 to 3.
No path from 9 to 4.
No path from 9 to 5.
No path from 9 to 6.
No path from 9 to 7.
No path from 9 to 8.
Distance from node 9 to node 10 = 447
Shortest Path traced backwards from node 10 to node 9:
   10, 9

Distance from node 9 to node 11 = 252
Shortest Path traced backwards from node 11 to node 9:
   11, 10, 9

No path from 9 to 12.

Distance from node 9 to node 13 = -2656
Shortest Path traced backwards from node 13 to node 9:
   13, 11, 10, 9

SOURCE NODE = 10.
RESULTS:
No path from 10 to 0.
No path from 10 to 1.
No path from 10 to 2.
No path from 10 to 3.
No path from 10 to 4.
No path from 10 to 5.
No path from 10 to 6.
No path from 10 to 7.
No path from 10 to 8.
No path from 10 to 9.

Distance from node 10 to node 11 = -195
Shortest Path traced backwards from node 11 to node 10:
   11, 10

No path from 10 to 12.

Distance from node 10 to node 13 = -3103
Shortest Path traced backwards from node 13 to node 10:
   13, 11, 10

SOURCE NODE = 11.
RESULTS:
No path from 11 to 0.
No path from 11 to 1.
No path from 11 to 2.
No path from 11 to 3.
No path from 11 to 4.
No path from 11 to 5.
No path from 11 to 6.
No path from 11 to 7.
No path from 11 to 8.
No path from 11 to 9.
No path from 11 to 10.
No path from 11 to 12.

Distance from node 11 to node 13 = -2908
Shortest Path traced backwards from node 13 to node 11:
   13, 11

SOURCE NODE = 12.
RESULTS:
No path from 12 to 0.
No path from 12 to 1.
No path from 12 to 2.
No path from 12 to 3.
No path from 12 to 4.
No path from 12 to 5.
No path from 12 to 6.
No path from 12 to 7.
No path from 12 to 8.
No path from 12 to 9.
No path from 12 to 10.
No path from 12 to 11.

Distance from node 12 to node 13 = -2612
Shortest Path traced backwards from node 13 to node 12:
13, 12

SOURCE NODE = 13.
RESULTS:
No path from 13 to 0.
No path from 13 to 1.
No path from 13 to 2.
No path from 13 to 3.
No path from 13 to 4.
No path from 13 to 5.
No path from 13 to 6.
No path from 13 to 7.
No path from 13 to 8.
No path from 13 to 9.
No path from 13 to 10.
No path from 13 to 11.
No path from 13 to 12.
III. Floyd-Warshall Algorithm

Instructions for Running algfw.c

This program finds the shortest path between all pairs of nodes in the network using the Floyd-Warshall Algorithm. Before running the C program, you must have a weight matrix describing the network stored in an ASCII file. The weight matrix has one row for each node (numbered starting with 0, not 1). Each row of the matrix must be on a separate row in the file and each column of the matrix must be separated by at least one space.

To run the program, type:

```
algfw [-v]
```

and press ENTER. The -v option may be passed to the program on the command line. This is the verbose option. When the -v switch is used, informational messages explaining the progress of the algorithm are printed as the program executes.

The program will prompt you to enter the name of the file in which the matrix is stored.

```
Enter the weight matrix file name:
```

Include the directory path if the file is not in the current directory.
The total number of nodes in the network will be printed out. In Example #2, the following line will be printed:

*Total number of nodes in the network = 14.*

The program will print informational messages as it runs if the -v option was specified on the command line.

After the program has terminated, all the elements in two matrices will be printed out. The first is *MATRIX* and the second is *PATH*. To determine the shortest path between node i and node j, look at the i,jth element in the *MATRIX* array. To determine the actual path, use the *PATH* array. Element i,j in the *PATH* array represents the predecessor node of node j in the path from node i to node j. If a negative cycle is found, the program will halt and a message will be printed. The following listing of results is printed out when algfw.c is run with the weight matrix of Example #2 as input. There is a brief summary of how to interpret these results for Example #2 at the end of the listing.

FLOYD-WARSHALL SHORTEST PATH ALGORITHM (algfw.c)
Finds the shortest path between all pairs of nodes.

Enter the weight matrix file name:
mat14s.mtx

NETWORK:
Total number of nodes in the network = 14.
RESULTS:
DISTANCE ARRAY:
matrix[0][0] = 0
matrix[0][1] = -173
matrix[0][2] = -570
matrix[0][3] = -736
matrix[0][4] = -1952
matrix[0][5] = -895
matrix[0][6] = -2107
matrix[0][7] = -2816
matrix[0][8] = -3041
matrix[0][9] = -3621
matrix[0][10] = -3174
matrix[0][11] = -3369
matrix[0][12] = -3268
matrix[0][13] = -6277
matrix[1][0] = 10000
matrix[1][1] = 0
matrix[1][2] = -397
matrix[1][3] = -563
matrix[1][4] = -1779
matrix[1][5] = -722
matrix[1][6] = -1934
matrix[1][7] = -2643
matrix[1][8] = -2868
matrix[1][9] = -3448
matrix[1][10] = -3001
matrix[1][11] = -3196
matrix[1][12] = -3095
matrix[1][13] = -6104
matrix[2][0] = 10000
matrix[2][1] = 10000
matrix[2][2] = 0
matrix[2][3] = -166
matrix[2][4] = -1382
matrix[2][5] = -325
matrix[2][6] = -1537
matrix[2][7] = -2246
matrix[2][8] = -2471
matrix[2][9] = -3051
matrix[2][10] = -2604
matrix[2][11] = -2799
matrix[2][12] = -2698
matrix[2][13] = -5707
matrix[3][0] = 10000
matrix[3][1] = 10000
matrix[3][2] = 10000
matrix[3][3] = 0
matrix[3][4] = -1216
matrix[3][5] = -159
matrix[3][6] = -1371
matrix[3][7] = -2080
matrix[3][8] = -2305
matrix[3][9] = -2885
matrix[3][10] = -2438
matrix[3][11] = -2633
matrix[3][12] = -2532
matrix[3][13] = -5541
matrix[4][0] = 10000
matrix[4][1] = 10000
matrix[4][2] = 10000
matrix[4][3] = 10000
matrix[4][4] = 0
matrix[4][5] = 10000
matrix[4][6] = -155
matrix[4][7] = -864
class matrix[4][8] = -1089
matrix[4][9] = -1669
matrix[4][10] = -1222
matrix[4][11] = -1417
matrix[4][12] = -1316
matrix[4][13] = -4325
matrix[5][0] = 10000
matrix[5][1] = 10000
matrix[5][2] = 10000
matrix[5][3] = 10000
matrix[5][4] = 10000
matrix[5][5] = 0
matrix[5][6] = 10000
matrix[5][7] = 10000
matrix[5][8] = 185
matrix[5][9] = 10000
matrix[5][10] = 10000
matrix[5][11] = 271
matrix[5][12] = -42
matrix[5][13] = -2654
matrix[6][0] = 10000
matrix[6][1] = 10000
matrix[6][2] = 10000
matrix[6][3] = 10000
matrix[6][4] = 10000
matrix[6][5] = 10000
matrix[6][6] = 0
matrix[6][7] = -709
matrix[6][8] = -934
matrix[6][9] = -1514
matrix[6][10] = -1067
matrix[6][11] = -1262
matrix[6][12] = -1161
matrix[6][13] = -4170
matrix[7][0] = 10000
matrix[7][1] = 10000
matrix[7][2] = 10000
matrix[7][3] = 10000
matrix[7][4] = 10000
matrix[7][5] = 10000
matrix[7][6] = 10000
matrix[7][7] = 0
matrix[7][8] = -225
matrix[7][9] = -805
matrix[7][10] = -358
matrix[7][11] = -553
matrix[7][12] = -452
matrix[7][13] = -3461
matrix[8][0] = 10000
matrix[8][1] = 10000
matrix[8][2] = 10000
matrix[8][3] = 10000
matrix[8][4] = 10000
matrix[8][5] = 10000
matrix[8][6] = 10000
matrix[8][7] = 10000
matrix[8][8] = 0
matrix[8][9] = 10000
matrix[8][10] = 10000
matrix[8][11] = 86
matrix[8][12] = -227
matrix[8][13] = -2839
matrix[9][0] = 10000
matrix[9][1] = 10000
matrix[9][2] = 10000
matrix[9][3] = 10000
matrix[9][4] = 10000
matrix[9][5] = 10000
matrix[9][6] = 10000
matrix[9][7] = 10000
matrix[9][8] = 10000
matrix[9][9] = 0
matrix[9][10] = 447
matrix[9][11] = 252
matrix[9][12] = 10000
matrix[9][13] = -2656
matrix[10][0] = 10000
matrix[10][1] = 10000
matrix[10][2] = 10000
matrix[10][3] = 10000
matrix[10][4] = 10000
matrix[10][5] = 10000
matrix[10][6] = 10000
matrix[10][7] = 10000
matrix[10][8] = 10000
matrix[10][9] = 10000
matrix[10][10] = 0
matrix[10][11] = -195
matrix[10][12] = 10000
matrix[10][13] = -3103
matrix[11][0] = 10000
matrix[11][1] = 10000
matrix[11][2] = 10000
matrix[11][3] = 10000
matrix[11][4] = 10000
matrix[11][5] = 10000
matrix[11][6] = 10000
matrix[11][7] = 10000
matrix[11][8] = 10000
matrix[11][9] = 10000
matrix[11][10] = 10000
matrix[11][11] = 0
matrix[11][12] = 10000
matrix[12][0] = 10000
matrix[12][1] = 10000
matrix[12][2] = 10000
matrix[12][3] = 10000
matrix[12][4] = 10000
matrix[12][5] = 10000
matrix[12][6] = 10000

Appendix B
matrix[12][7] = 10000
matrix[12][8] = 10000
matrix[12][9] = 10000
matrix[12][10] = 10000
matrix[12][11] = 10000
matrix[12][12] = 0
matrix[12][13] = -2612
matrix[13][0] = 10000
matrix[13][1] = 10000
matrix[13][2] = 10000
matrix[13][3] = 10000
matrix[13][4] = 10000
matrix[13][5] = 10000
matrix[13][6] = 10000
matrix[13][7] = 10000
matrix[13][8] = 10000
matrix[13][9] = 10000
matrix[13][10] = 10000
matrix[13][11] = 10000
matrix[13][12] = 10000
matrix[13][13] = 0

PATH ARRAY:
path[0][0] = 0
path[0][1] = 0
path[0][2] = 1
path[0][3] = 2
path[0][4] = 3
path[0][5] = 3
path[0][6] = 4
path[0][7] = 6
path[0][8] = 7
path[0][9] = 7
path[0][10] = 9
path[0][11] = 10
path[0][12] = 8
path[0][13] = 11
path[1][0] = 0
path[1][1] = 1
path[1][2] = 1
path[1][3] = 2
path[1][4] = 3
path[1][5] = 3
path[1][6] = 4
path[1][7] = 6
path[1][8] = 7
path[1][9] = 7
path[1][10] = 9
path[1][11] = 10
path[1][12] = 8
path[1][13] = 11
path[2][0] = 0
path[2][1] = 0
path[2][2] = 2
path[2][3] = 2
path[2][4] = 3
path[2][5] = 3
path[2][6] = 4
path[2][7] = 6
path[2][8] = 7
path[2][9] = 7
path[2][10] = 9
path[2][11] = 10
path[2][12] = 8
path[2][13] = 11
path[3][0] = 0
path[3][1] = 0
path[3][2] = 0
path[3][3] = 3
path[3][4] = 3
path[3][5] = 3
path[3][6] = 4
path[3][7] = 6
path[3][8] = 7
path[3][9] = 7
path[3][10] = 9
path[3][11] = 10
path[3][12] = 8
path[3][13] = 11
path[4][0] = 0
path[4][1] = 0
path[4][2] = 0
path[4][3] = 0
path[4][4] = 4
path[4][5] = 0
path[4][6] = 4
path[4][7] = 6
path[4][8] = 7
path[4][9] = 7
path[4][10] = 9
path[4][11] = 10
path[4][12] = 8
path[4][13] = 11
path[5][0] = 0
path[5][1] = 0
path[5][2] = 0
path[5][3] = 0
path[5][4] = 0
path[5][5] = 5
path[5][6] = 0
path[5][7] = 0
path[5][8] = 5
path[5][9] = 0
path[5][10] = 0
path[5][11] = 8
path[5][12] = 8
path[5][13] = 12
path[6][0] = 0
path[6][1] = 0
path[6][2] = 0
path[6][3] = 0
path[6][4] = 0
path[6][5] = 0

Appendix B
path[6][6] = 6
path[6][7] = 6
path[6][8] = 7
path[6][9] = 7
path[6][10] = 9
path[6][11] = 10
path[6][12] = 8
path[6][13] = 11
path[7][0] = 0
path[7][1] = 0
path[7][2] = 0
path[7][3] = 0
path[7][4] = 0
path[7][5] = 0
path[7][6] = 0
path[7][7] = 7
path[7][8] = 7
path[7][9] = 7
path[7][10] = 9
path[7][11] = 10
path[7][12] = 8
path[7][13] = 11
path[8][0] = 0
path[8][1] = 0
path[8][2] = 0
path[8][3] = 0
path[8][4] = 0
path[8][5] = 0
path[8][6] = 0
path[8][7] = 0
path[8][8] = 8
path[8][9] = 0
path[8][10] = 0
path[8][11] = 8
path[8][12] = 8
path[8][13] = 12
path[9][0] = 0
path[9][1] = 0
path[9][2] = 0
path[9][3] = 0
path[9][4] = 0
path[9][5] = 0
path[9][6] = 0
path[9][7] = 0
path[9][8] = 0
path[9][9] = 9
path[9][10] = 9
path[9][11] = 10
path[9][12] = 0
path[9][13] = 11
path[10][0] = 0
path[10][1] = 0
path[10][2] = 0
path[10][3] = 0
path[10][4] = 0
path[10][5] = 0
path[10][6] = 0
path[10][7] = 0
path[10][8] = 0
path[10][9] = 0
path[10][10] = 10
path[10][11] = 10
path[10][12] = 0
path[10][13] = 11
path[11][0] = 0
path[11][1] = 0
path[11][2] = 0
path[11][3] = 0
path[11][4] = 0
path[11][5] = 0
path[11][6] = 0
path[11][7] = 0
path[11][8] = 0
path[11][9] = 0
path[11][10] = 0
path[11][11] = 11
path[11][12] = 0
path[11][13] = 11
path[12][0] = 0
path[12][1] = 0
path[12][2] = 0
path[12][3] = 0
path[12][4] = 0
path[12][5] = 0
path[12][6] = 0
path[12][7] = 0
path[12][8] = 0
path[12][9] = 0
path[12][10] = 0
path[12][11] = 0
path[12][12] = 12
path[12][13] = 12
path[13][0] = 0
path[13][1] = 0
path[13][2] = 0
path[13][3] = 0
path[13][4] = 0
path[13][5] = 0
path[13][6] = 0
path[13][7] = 0
path[13][8] = 0
path[13][9] = 0
path[13][10] = 0
path[13][11] = 0
path[13][12] = 0
path[13][13] = 13
In Example #2, the salesman wanted to find the "shortest path" in terms of net cost from Seattle to New York City, or node 0 to node 13. Element [0][13] in the MATRIX array equals -6277. This represents the smallest net cost attainable by a trip from Seattle (node 0) to New York City (node 13). Since the net cost represents traveling expenses minus profit, the salesman can expect a profit of 6277. To find the path that the salesman should take, look at element [0][13] in the PATH array, since the salesman wants to go from node 0 to node 13. The value of PATH[0][13] is 11. Node 11 is the predecessor of node 13 on the path from node 0 to node 13. Next look at PATH[0][11]. Its value is 10. Node 10 is the predecessor of node 11 on the path from node 0 to node 13. Thus far, the path from node 0 to node 13, traced backwards, is:

13, 11, 10, ...

The next node in the path can be obtained by looking at PATH[0][10]. Therefore, the next node is 9. This procedure is repeated until the source node is reached. The resulting shortest path from node 0 to node 13 is:

0, 1, 2, 3, 4, 6, 7, 9, 10, 11, 13
Example #3: The outcome of running the Floyd-Warshall Algorithm on a network with a negative cycle can be seen when the algfw.c program is run with Example #3. The following output listing results:

FLOYD-WARSHALL SHORTEST PATH ALGORITHM  (algfw.c)
Finds the shortest path between all pairs of nodes.

Enter the weight matrix file name:
mat5b.mtx

NETWORK:
Total number of nodes in the network = 5.
A negative cycle has been found with node 3.

RESULTS:
WEIGHT ARRAY:
matrix[0][0] = 0
matrix[0][1] = 7
matrix[0][2] = 10000
matrix[0][3] = 12
matrix[0][4] = 10000
matrix[1][0] = 10000
matrix[1][1] = 0
matrix[1][2] = 10000
matrix[1][3] = 5
matrix[1][4] = 10000
matrix[2][0] = 10000
matrix[2][1] = -4
matrix[2][2] = 0
matrix[2][3] = 1
matrix[2][4] = -1
matrix[3][0] = 10000
matrix[3][1] = -7
matrix[3][2] = -3
matrix[3][3] = -2
matrix[3][4] = -4
matrix[4][0] = 6
matrix[4][1] = 4
matrix[4][2] = 10000
matrix[4][3] = 9
matrix[4][4] = 0

PATH ARRAY:
path[0][0] = 0
path[0][1] = 0
path[0][2] = 0
path[0][3] = 1
path[0][4] = 0
path[1][0] = 0
path[1][1] = 1
path[1][2] = 0
path[1][3] = 1
path[1][4] = 0
The program halts when it finds a negative cycle, signaled by a negative value for any of the entries on the diagonal in the \textit{MATRIX} array. In this example, \texttt{MATRIX[3][3]} is equal to -2. This reveals that node 3 is part of a negative cycle. Since a negative cycle exists in the network, the distances in the resulting \texttt{MATRIX} array will not be correct. The negative cycle can be found by tracing backwards through the \texttt{PATH} array:

\texttt{3, 1, 2, 3}

Which results in the following cycle (when the nodes are reversed to put them in the proper order):

\texttt{3, 2, 1, 3}
APPENDIX C

Evaluation of Simulation Software

I studied a network simulation software package to see if it could provide a comparison for the analytical methods I used. The software package is called NETWORK II.5 Release 6.0, made by CACI Products Company. The software allows the user to model a local area network. The main objective of the software is to help a network designer model several network configurations and simulate efficiency and percent utilization of those configurations. Conflicts between devices in the network can be examined also.

The software package was extremely flexible and very interesting to me. However, it did not provide a way to model shortest paths in a network simulation, or have any optimization capabilities relating to shortest paths. Therefore, the software could not be used as part of this thesis.
INDEX

acyclic ................................................................. 3
algbf1.c ............................................................... 25, 37, 39
  instructions ....................................................... 98-101
  output listing ...................................................... 101, 102
  program listing .................................................... 63-67
algbf2.c ............................................................... 25, 38, 39
  instructions ....................................................... 103-106
  output listing ...................................................... 106-115
  program listing .................................................... 68-72
algd1.c ............................................................... 20, 22, 32
  instructions ....................................................... 84-87
  output listing ...................................................... 87
  program listing .................................................... 46-49
algd2.c ............................................................... 22, 32
  instructions ....................................................... 88-91
  output listing ...................................................... 91
  program listing .................................................... 50-54
algd3.c ............................................................... 22, 33, 37, 39
  instructions ....................................................... 92-94
  program listing .................................................... 55-58
algd4.c ............................................................... 22, 33, 38, 39
  instructions ....................................................... 95-97
  program listing .................................................... 59-62
algfw.c ............................................................... 29, 38, 39
  instructions ....................................................... 116, 117, 125-127
  output listing 1 .................................................. 117-124
  output listing 2 .................................................. 126, 127
  program listing .................................................... 73-75
antisymmetric ........................................................ 7
arc ................................................................. 1, 3, 5, 6, 9, 11-18, 21, 23, 31, 37, 38, 41, 77, 88, 89, 98, 99, 103, 104
directed ............................................................ 1, 2, 4, 7
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>undirected</td>
<td>4</td>
</tr>
<tr>
<td>arcs</td>
<td></td>
</tr>
<tr>
<td>parallel</td>
<td>3, 5, 11, 12</td>
</tr>
<tr>
<td>asymmetric</td>
<td>7</td>
</tr>
<tr>
<td>Bellman-Ford Algorithm</td>
<td>23, 31, 34, 37-39, 42, 80, 98, 103</td>
</tr>
<tr>
<td>circuit</td>
<td></td>
</tr>
<tr>
<td>directed</td>
<td>3</td>
</tr>
<tr>
<td>computational complexity</td>
<td>26, 30-32, 34, 35, 42</td>
</tr>
<tr>
<td>cycle</td>
<td>3</td>
</tr>
<tr>
<td>negative</td>
<td>3, 4, 23, 25, 26, 28, 29, 35, 83, 126, 127</td>
</tr>
<tr>
<td>Dantzig</td>
<td>43</td>
</tr>
<tr>
<td>digraph</td>
<td>2, 3, 5, 10-12, 16, 17</td>
</tr>
<tr>
<td>antisymmetric</td>
<td>7</td>
</tr>
<tr>
<td>asymmetric</td>
<td>7</td>
</tr>
<tr>
<td>equivalence</td>
<td>8</td>
</tr>
<tr>
<td>irreflexive</td>
<td>6</td>
</tr>
<tr>
<td>reflexive</td>
<td>5</td>
</tr>
<tr>
<td>simple</td>
<td>5</td>
</tr>
<tr>
<td>symmetric</td>
<td>5</td>
</tr>
<tr>
<td>transitive</td>
<td>6</td>
</tr>
<tr>
<td>Dijkstra's Shortest Path Algorithm</td>
<td>18, 19, 22, 31-34, 37-40, 42, 77, 84, 88, 92, 95</td>
</tr>
<tr>
<td>equivalence</td>
<td>8</td>
</tr>
<tr>
<td>Floyd-Warshall Algorithm</td>
<td>26, 35, 40, 42, 80, 83, 116, 126</td>
</tr>
<tr>
<td>forward star</td>
<td>15, 16, 25, 41, 77, 79, 82</td>
</tr>
<tr>
<td>storage</td>
<td>15</td>
</tr>
<tr>
<td>graph</td>
<td>1</td>
</tr>
<tr>
<td>directed</td>
<td>2, 3, 31, 78</td>
</tr>
<tr>
<td>simple directed</td>
<td>3</td>
</tr>
<tr>
<td>undirected</td>
<td>2</td>
</tr>
<tr>
<td>greedy algorithm</td>
<td>18</td>
</tr>
<tr>
<td>Hu</td>
<td>43</td>
</tr>
<tr>
<td>in-degree</td>
<td>2</td>
</tr>
<tr>
<td>in-valence</td>
<td>2</td>
</tr>
<tr>
<td>irreflexive</td>
<td>6, 11</td>
</tr>
</tbody>
</table>

Index 130
<table>
<thead>
<tr>
<th>term</th>
<th>pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>label</td>
<td>23, 24</td>
</tr>
<tr>
<td>correcting method</td>
<td>23</td>
</tr>
<tr>
<td>permanent</td>
<td>19-22, 30-34, 85, 89, 90, 93, 96</td>
</tr>
<tr>
<td>setting method</td>
<td>19</td>
</tr>
<tr>
<td>temporary</td>
<td>19-21, 31, 32</td>
</tr>
<tr>
<td>list</td>
<td>13</td>
</tr>
<tr>
<td>adjacency</td>
<td>13</td>
</tr>
<tr>
<td>linked adjacency</td>
<td>14, 16, 21, 22, 32, 33, 41, 77, 79, 82</td>
</tr>
<tr>
<td>of edges</td>
<td>13</td>
</tr>
<tr>
<td>storage</td>
<td>13, 14</td>
</tr>
<tr>
<td>matrix</td>
<td>1, 9, 13, 17, 26, 28, 36</td>
</tr>
<tr>
<td>adjacency</td>
<td>12, 23, 33, 36</td>
</tr>
<tr>
<td>node-arc incidence</td>
<td>9, 10</td>
</tr>
<tr>
<td>vertex-arc incidence</td>
<td>9</td>
</tr>
<tr>
<td>weight</td>
<td>10, 11, 18, 20-23, 26, 27, 29, 31, 33, 35-36, 41, 77, 79, 81, 83, 84, 92, 95, 98, 103, 116</td>
</tr>
<tr>
<td>minimization</td>
<td>28, 30-33, 35, 36</td>
</tr>
<tr>
<td>network</td>
<td>1-4, 9, 10, 13-16, 18, 19, 21-23, 25, 26, 30, 31, 41, 77, 84, 85, 88, 92, 95, 98, 103, 116</td>
</tr>
<tr>
<td>complete</td>
<td>21, 37-41</td>
</tr>
<tr>
<td>dense</td>
<td>41, 42</td>
</tr>
<tr>
<td>example</td>
<td>2, 11, 77, 80, 83</td>
</tr>
<tr>
<td>sparse</td>
<td>13, 21, 25, 34, 37, 38, 40-42</td>
</tr>
<tr>
<td>ordered pair</td>
<td>2, 26</td>
</tr>
<tr>
<td>out-degree</td>
<td>2</td>
</tr>
<tr>
<td>out-valence</td>
<td>2</td>
</tr>
<tr>
<td>recent</td>
<td>20, 30</td>
</tr>
<tr>
<td>reflexive</td>
<td>5, 8</td>
</tr>
<tr>
<td>self loop</td>
<td>3, 5-7, 11, 12</td>
</tr>
<tr>
<td>simple</td>
<td>5</td>
</tr>
<tr>
<td>spath.h</td>
<td>76</td>
</tr>
<tr>
<td>program listing</td>
<td></td>
</tr>
<tr>
<td>symmetric</td>
<td>5, 8</td>
</tr>
<tr>
<td>time trials</td>
<td>37</td>
</tr>
<tr>
<td>transitive</td>
<td>6, 8</td>
</tr>
<tr>
<td>Term</td>
<td>Pages</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>closure</td>
<td>16</td>
</tr>
<tr>
<td>transitive closure</td>
<td>16, 17</td>
</tr>
<tr>
<td>variants</td>
<td>43</td>
</tr>
<tr>
<td>weight</td>
<td>1, 4</td>
</tr>
<tr>
<td>arc</td>
<td>3, 11, 19, 28, 29, 77, 98, 103</td>
</tr>
<tr>
<td>array</td>
<td>13, 15, 30, 85-87, 89-91, 93, 94, 96, 97</td>
</tr>
<tr>
<td>minimum</td>
<td>4</td>
</tr>
<tr>
<td>negative</td>
<td>1, 4, 26, 34, 42, 80</td>
</tr>
<tr>
<td>non-existent arc</td>
<td>18</td>
</tr>
<tr>
<td>non-negative</td>
<td>4, 42</td>
</tr>
<tr>
<td>upper bound</td>
<td>43</td>
</tr>
</tbody>
</table>