1.1 Introduction:

Various number of optimization problems are mathematically equivalent to solve shortest-paths in a graph. The shortest path algorithms have been worked thoroughly than any other algorithms in Graph Theory. For a good comparative study of various shortest-path algorithms we refer to the work of Dreyfus[5]. Generally, shortest-paths can be calculated through weight of a path in a weighted graph as its length. The minimum weight of a path will be called the distance between the two vertices or points. The basic assumption is that all the weights are positive. If the weight of the edge is zero, its ends doesn’t have connections. This means there is no path. If an edge is not present in that graph then we define the weight of the edge to be infinite.

There are different types of shortest path problems the following algorithms appear frequently.

1. Shortest-path between two specified vertices.
2. Shortest-path between all pairs of vertices
3. Shortest-path from a specified vertex to all others.
4. Shortest-path between specified vertices that passes through specified vertices.
The algorithm for solving the problem of shortest-path from a specified vertex to another specified vertex was discovered by Dijkstra[4] and independently, by whiting and Hiller[15]. It finds not only a shortest-path but shortest-path from source vertex to all other vertices of Graph.

In this chapter we discuss the algorithm of finding the shortest path from a specified vertex to another specified vertex. A simple weighted digraph \( G \) of \( m \) vertices is described by an \( m \) by \( m \) matrix is \( D = [d_{pq}] \)

where

\[ d_{pq} = \text{Length or distance or weight of the directed edge from vertex } 'p' \text{ to vertex } q, \]
\[ d_{pq} \geq 0 \]
\[ d_{pp} = 0 \]
\[ d_{pq} = \infty, \text{ if there is no edge from } p \text{ to } q. \]

We know that, \( d_{pq} \neq d_{qp} \) and the triangle inequality need not be satisfied. That is \( d_{pq} + d_{qr} \) may be less than \( d_{pr} \). In fact, if the triangle inequality is satisfied for every \( p, q \) and \( r \), the problem would be trivial because the direct edge \((x,y)\) would be the shortest path from vertex \( x \) to vertex \( y \). The distance of a directed path \( A \) is defined to be the sum of the length of the edges in \( A \). The problem is to find the shortest possible path and its length from a starting vertex \( s \) to a terminal vertex \( t \).
1.2 Preliminaries:

With each edge $e$ of a graph $G$, let there be associated a real number $w(e)$ called its weight. Then $G$, together with these weights on its edges is called a weighted Graph. Weighted Graphs occur frequently in applications of Graph Theory. In the friendship Graph, for example, weights might indicate intensity of friendship. In the communications Graph, they could represent the construction or maintenance costs of the various links.

If $H$ is a subgraph of a weighted graph, the weight $w(H)$ is the sum of the weights $\sum_{e \in H} w(e)$ on its edges. In a weighted Graph, a path of minimum weight connecting two specified vertices $u$ and $v$.

For solving the shortest-path problem, we refer to the weight of a path in a weighted Graph as its length. The minimum weight of a $(u,v)$-path will be called the distance between $u$ and $v$ and denoted by $d(u,v)$.

We use the following terminology while discussing this algorithm.

1. Path of a Graph: An open walk in which no vertex appears more than once is called a path (simple path). The number of edges in a path is called the length of a path.

2. Tree: A Tree is a connected graph without any circuits. A tree has to be a simple Graph that is having neither a self loop nor parallel edges.

![Tree Diagram](Fig: (1.1) Tree)
3. Spanning Tree: A Tree $T$ is said to be a spanning tree of a connected Graph $G$ if $T$ is a subgraph of $G$ and $T$ contains all vertices of $G$. For instance, the subgraph in heavy lines is a spanning tree of the graph.

4. Circuit: A closed walk in which no vertex appears more than once is called a circuit. Then we have that every vertex in a Circuit is of degree two. If the Circuit is a subgraph of another Graph, one must count degrees contributed by the edges in the circuit only. A circuit is also called a cycle, elementary cycle, circular path and Polygon.

Figure (1.3) Three different Circuits
Arborescence: A digraph $G$ is said to be an Arborescence if

i. $G$ contains no circuit - neither directed nor semicircuit.

ii. In $G$ there is precisely one vertex $v$ of zero in-degree.

iii. This vertex $v$ is called the root of the arborescence.

Fig. (1.4): Arborescence
**Subgraph of G** Any collection of edges in G

**Walk** A non-edge-retracing sequence of edges in G

A non-intersecting open-walk in G

**Path in G**

**Circuit in G** A non-intersecting closed walk in G

**Fig 1.5**: Walk, Paths and circuits as subgraph

**Connected Graph**: A Graph G is said to be Connected if there is at least one path between every pair of vertices in G.
1.3 Dijkstra's Algorithm:

Dijkstra's Algorithm has the following steps to find the shortest-paths from one vertex to all the remaining vertices in the graph.

1. Set \( l(u_0) = 0 \), \( l(v) = \infty \) for \( v = u_0 \), \( s_0 = \{ u_0 \} \) and \( i = 0 \).

2. For each \( v \in s_i \), replace \( l(v) \) by \( \min \{ l(v), l(u) + w(u, v) \} \)

   Compute \( \min \{ l(v) \} \) and let \( u_i + 1 \) denote a vertex for which this minimum is attained.

   \( s_{i+1} = s_i \cup \{ u_i + 1 \} \).

3. If \( i = v - 1 \), stop. If \( i < v - 1 \), replace \( i \) by \( i + 1 \) and goto step 2.

When the algorithm terminates, the distance from \( u_0 \) to \( v \) is given by the final value of the label \( l(v) \). A Flow-Chart explaining and deleting this algorithm is shown in Fig 1.

Dijkstra's Algorithm determines only the distance from \( u_0 \) to all the other vertices, and not actually shortest-paths. This Shortest-Path can however be easily determined by keeping track of the predecessors of vertices in the Tree.

Dijkstra's Algorithm takes a Shortest-Path from the starting vertex \( p_0 \) to each of the other vertices which are accessible from source vertex, then the union of these paths will be an arborescence \( T \) at source vertex. Every path in \( T \) from source is the unique shortest path in the digraph. Such a Tree is called the shortest distance arborescence in an undirected graph and it is not the shortest spanning tree.
In Dijkstra's algorithm, more vertices acquire permanent labels the number of additions and comparisons needed to modify the temporary labels continues to decrease. In this case where every vertex gets permanently labeled, we need \( \frac{a(a-1)}{2} \) and \( 2a(a-1) \) comparisons. Thus the computation time is proportional to \( a^2 \).

If the given Graph \( G \) is not weighted, every edge in \( G \) has a weight of one, and matrix \( D \) is the same as the adjacency matrix. For this we perform logical operations rather than real arithmetic.

We have assumed the distance \( d_{pq} \) are all non-negative numbers shortest-path algorithms have however been provided the sum of all \( d_{pq} \) around every direct circuit is positive. The computation time of the algorithms that can handle \( d_{pq} \) is \( a^3 \) and not \( a^2 \).
Fig (1.6): Shortest distance from slot
1.4 Algorithm:

/* This algorithm takes n the number of nodes in the Graph and D the adjacency matrix of the graph as input. 's' and 't' are the source and destination or terminal nodes. Label is a record, vect, s, m, p and c are integer variables, z is an integer array */

1. Read n, the number of nodes in the graph, source and the destination nodes 's' and 't' and 'd' as the adjacency matrix, Vect is assigned to zero.

2. Repeat step (4) for all 'n' nodes.

3. m= infinity (Large value)

4. c = s, vect[j] = 1, j=j+1

5. Repeat through step(6) for i=1 to n

6. if (label(i,j)=0) then label[j]=dij+label(i)


8. Repeat step(9) for j=1 to n

9. If (label(j) > m) then M « label(j)

10. i=p; vect(p) = t; j=j+1   i = p & do step (1) to 17 until p=t
11. Repeat through step(13) for i=1 to n

12. If ((i not in p) and (label(s)= 0)) then label(j) = z

13. If z< label(j)+dij then label(j) = z

14. m=∞

15. for i = 1 to n do step6

16. if (label(j)>m then m = label(j) and p=j)

17. label(j)=m then j=j+1

18. print label(t) for j=1 to n also shortest path distance.
1.5 Illustration

For vertex to vertex shortest path algorithm for directed weighted graphs we give the input in the form of adjacency matrix. ['m' by 'm' matrix] i.e. $D = [d_{pq}]$ where $d_{pq} =$ length or distance or weight of the directed edge from vertex 'p' to 'q', $d_{pq} \geq 0$

$d_{pp} = 0$, which explains vertex it self a self loop.

$d_{pq} = \infty$, if there is no edge from 'p' to 'q'.

First we decide, for which vertices the shortest-path is to be found. After that we consider the directed weights of the graph and the minimum directed weighted edge is considered as the shortest route edge. Like wise, for the next vertex again the least weight edge is considered and process is continued to the terminal vertex. But if two weighted edges are equal then arbitrarily we can choose any one of them.

Example :1

Consider the following Directed Weighted Graph

![Diagram](image.png)

Fig (1.7) : Simple weighted Digraph
For this Graph, the input is in the form of Adjacency Directed Weighted Graph.

\[
\begin{array}{ccccccc}
A & B & C & D & E & F & G \\
\hline
A & 0 & \infty & \infty & 8 & 2 & \infty & \infty \\
B & 7 & 0 & 1 & \infty & \infty & \infty & \infty \\
C & 3 & \infty & 0 & \infty & 4 & 3 & \infty \\
D & 2 & \infty & \infty & 0 & 1 & \infty & \infty \\
E & \infty & \infty & \infty & \infty & 0 & \infty & 2 \\
F & \infty & \infty & \infty & 10 & 4 & \infty & 7 \\
G & \infty & \infty & \infty & 2 & \infty & \infty & 0 \\
\end{array}
\]

Here the shortest-path in the form Adjacency matrix and directed graph is i.e vertex to vertex B to G

\[
\text{Cost of Shortest path} = 11 \\
\text{Root is B} \rightarrow C \rightarrow F \rightarrow G
\]
Example: 2

Consider the following Directed Weighted Graph

For this graph, the input is in the form of Adjacency Directed weighted Graph.

\[
\begin{array}{cccc}
  v_1 & v_2 & v_3 & v_4 \\
  v_1 & 0 & 2 & \infty & \infty \\
  v_2 & \infty & 0 & 3 & 2 \\
  v_3 & \infty & 4 & 0 & 5 \\
  v_4 & 1 & 0 & 6 & 0 \\
\end{array}
\]

Here the shortest-path in the form adjacency matrix and directed graph

Cost of Shortest - path = 5
Root is \( v_1 \rightarrow v_2 \rightarrow v_3 \)
We have designed a program based on the algorithm discussed earlier obtained the outputs in the desired way. We present the details here under.

1.6 PROGRAM:
/* To find the shortest path of a graph using Dijkstra's Algorithm, from one vertex to another vertex : */
#include <stdio.h>
#include <limits.h>
/* Maximum number of Nodes in a graph */
#define MAXNODE 10
#define PERM 1
#define TENT 2
#define infinity INT-MAX
typedef struct NODELABEL
{
    int predecessor;
    int length; /* optimal distance from source */
    int label; /* label is tentative or permanent */
} NODELABEL;
/* Function : short path
prototype : int shortpath (a,n,s,t,path,dist)

Input : a - Adjacency matrix describing the graph
        n - Number of nodes in the graph
        s - source node
        t - Target or sink node

Output : path - list of optimal path from source to sink
         dist - minimum distance between source and sink */
Return: 0 - if there is no path
count - indicating the number of nodes along the optimal path,
otherwise */

int shortpath (a,n,s,t,path,dist)
int a[MAXNODE][MAXNODE], n,s,t,path[MAXNODE], *dist;
{
    NODELABEL state[MAXNODE];
    int i,k,min,count;
    int rpath[MAXNODE];
    *dist = 0;
    /* Initialize all nodes as tentative nodes */
    for ( i=1; i<=n; i++)
    {
        state[i].predecessor = 0;
        state[i].length = infinity;
        state[i].label = TENT;
    }
    /* make source node as permanent */
    state[s].predecessor = 0;
    state[s].length = 0;
    state[s].label = PERM;
    /* start from source node */
    k = s;
    do {
        /* check all paths from kth node and find their distance k node */
        for (i=1; i<=n; i++)
/* -ve if no direct path, 0 if to the same, otherwise direct path */
if (a[k][i] > 0 && state[i].label == TENT )
{
    if (state[k].length + a[k][i] < state[i].length )
    {
        state[i].predecessor = k;
        state[i].length = state[k].length + a[k][i];
    }
}

/* find the tentatively labeled node with smaller cost */
min = infinity;
k = 0;
for (i=1; i<=n; i++)
{
    if (state[i].label == TENT && state[i].length < min)
    {
        min = state[i].length;
        k = i;
    }
}

/* is source or sink node is isolated */
if (k == 0)
    return(0);
state[k].label = PERM;
} while (k != t);

/* store optimal path */
k = t;
count = 0;
do {
    count = count + 1;
    rpath [count] = k;
    k = state[k].predecessor;
} while (k != 0);

/* reverse nodes since algorithm stores path in reverse direction */
for (i=1; i<= count; i++)
    path[i] = rpath[count - i + 1];
for (i=1; i<= count; i++)
    *dist += a[path[i]][path[i+1]]; 
return(count);
}

void main()
{
    int a[MAXNODE][MAXNODE], i,j;
    int path[MAXNODE];
    int from, to, dist, count, n;
    printf("How many nodes ?");
    scanf("%d", &n);
    printf("%dB4i); 
    for (i=1; i<=n; i++)
    {
        printf("\n enter node %d connectivity : " i);
        for (j=1; j<=n; j++)
        {
            scanf("%d", &a[i][j]);
            printf("%d", a[i][j]);
        }
    }
printf("\n from to where ? ");
scanf("%d %d", &from, &to);
printf("%d %d", from, to);
count = shortpath(a, n, from, to, path, &dist);
if (dist)
{
    printf("\n shortest path: ");
    printf("%d", path[i]);
    for (i = 2; i <= count; i++)
        printf("-> %d", path[i]);
    printf("\n minimum distance = %d\n", dist);
}
else
    printf("\n path does not exists \n");
}
Demonstration with an example:
Consider the following directed weighted graph

For this Graph, the input is in the form of Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
For this Graph, weighted adjacency matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>3</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>∞</td>
<td>0</td>
<td>10</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>F</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>6</td>
<td>0</td>
<td>∞</td>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>G</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>H</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>I</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>4</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>

Here the shortest-path is in the form of Adjacency matrix and weighted directed graph is

Shortest-path from A to E

Cost of shortest-path = 6
Root is A → H → G → E
1.8 Result

How Many Nodes ? = 7
Enter Node 1 Connectivity : 0 0 8 2 0 0
Enter Node 2 Connectivity : 7 0 1 0 0 0
Enter Node 3 Connectivity : 3 0 0 4 3 0
Enter Node 4 Connectivity : 2 0 0 1 0 0
Enter Node 5 Connectivity : 0 0 0 0 0 2
Enter Node 6 Connectivity : 0 0 10 4 0 7
Enter Node 7 Connectivity : 0 0 0 2 0 0

From To Where ? 2 7

Shortest path : 2 -> 3 -> 5 -> 7
Minimum Distance = 7
How Many Nodes ? = 7
Enter Node 1 Connectivity : 0 0 0 8 2 0 0
Enter Node 2 Connectivity : 7 0 1 0 0 0 0
Enter Node 3 Connectivity : 3 0 0 0 4 3 0
Enter Node 4 Connectivity : 2 0 0 0 1 0 0
Enter Node 5 Connectivity : 0 0 0 0 0 0 2
Enter Node 6 Connectivity : 0 0 0 10 4 0 7
Enter Node 7 Connectivity : 0 0 0 2 0 0 0

From To Where ? 1 6
Path does not exist
How Many Nodes? = 4
Enter Node 1 Connectivity: 0 0 0 7
Enter Node 2 Connectivity: 0 0 0 0
Enter Node 3 Connectivity: 0 0 3 0
Enter Node 4 Connectivity: 0 0 9 0
From To Where? 1 3

Shortest Path: 1-> 4 -> 3
Minimum Distance = 16

How Many Nodes? = 5
Enter Node 1 Connectivity: 0 85 80 20 0
Enter Node 2 Connectivity: 0 0 20 0 95
Enter Node 3 Connectivity: 70 20 0 80 0
Enter Node 4 Connectivity: 0 75 0 0 75
Enter Node 5 Connectivity: 70 10 20 80 0
From To Where? 1 5

Shortest Path: 1-> 4 -> 5
Minimum Distance = 95