NETWORKS & COMMUNICATION (CSE 1004) PROJECT

ASHA S.

Kashish Miglani-15BCE1003

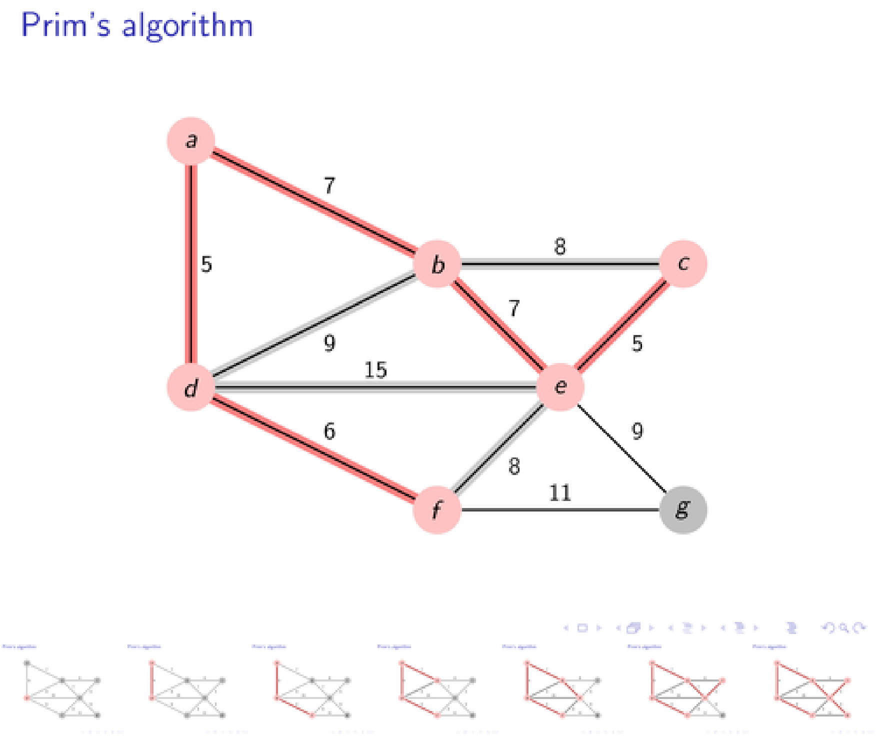
Vineet Kishore-16BCE1365

Siddharth Chandra-15BCE1286

Osho Agyeya-15BCE1326

Utsav Rai-15BCE1352

PRIM's ALGORITHM (OSHO AGYEYA)



USES OF PRIM's ALGORITHM

MAIN PURPOSE: FINDING MINIMUM SPANNING TREE

1.Distances between the cities for the minimum route calculation for transportation.

2.For Establishing the network cables these play important role in finding the minimum cables required to cover the whole region.

3. Prim Algorithm Approach to Improving Local Access Network in Rural Areas

4. AI (Artificial Intelligence)

5. Game Development

6. Cognitive Science

ALGORITHM

1. **Start** at ***any* node** in the **graph**

   - **Mark** the **starting node** as ***reached***
   - **Mark** all the **other nodes** in the **graph** as ***unreached***

   Right now, the **Minimum cost Spanning Tree (MST)** consists of the ***starting node***

   We **expand** the **MST** with the **procedure** given below....

---

2. **Find** an **edge *e*** with ***minimum cost*** in the **graph** that **connects**:

   - A ***reached* node *x*** to an ***unreached* node *y***

---

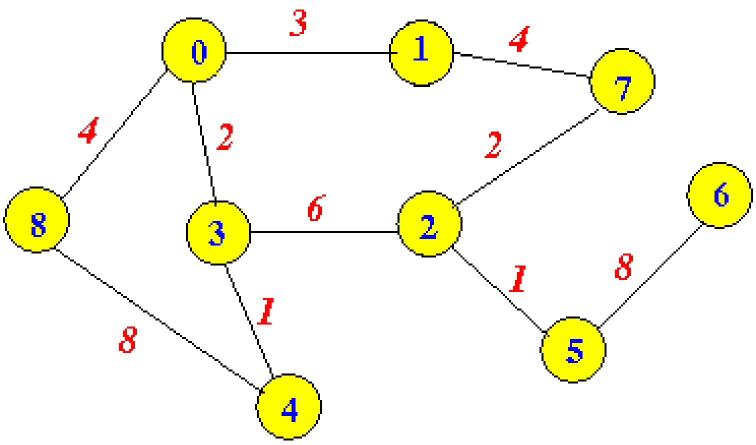
3. **Add** the **edge *e*** found in the **previous step** to the **Minimum cost Spanning Tree**

   Mark the ***unreached* node *y*** as ***reached***

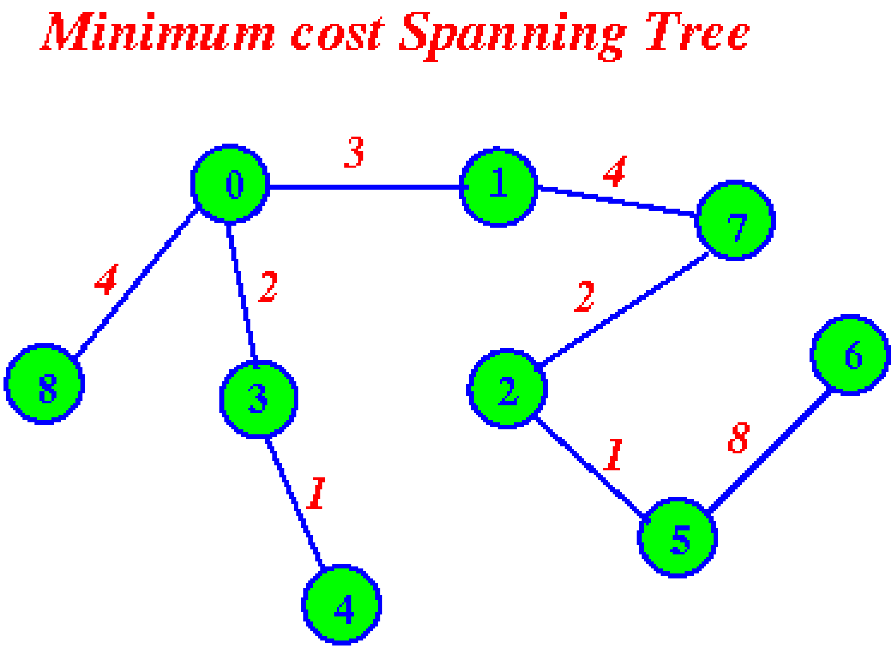
---

4. **Repeat** the steps **2** and **3** until ***all* nodes** in the **graph** have become ***reached***

INPUT



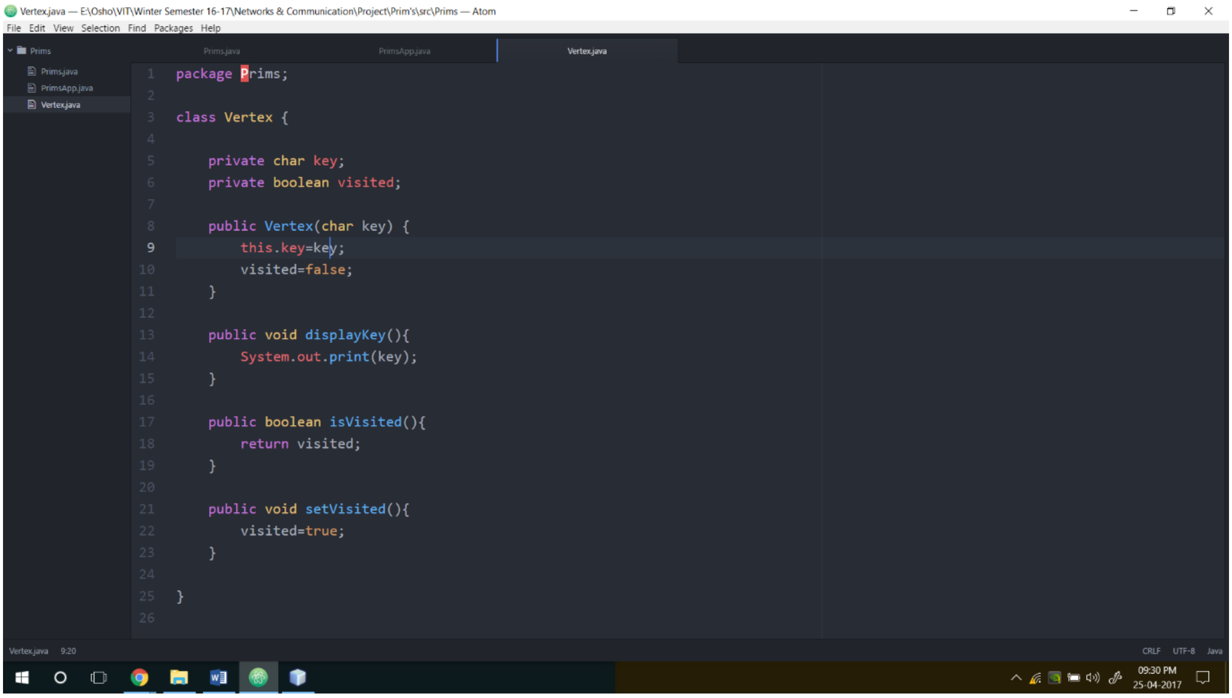
OUTPUT

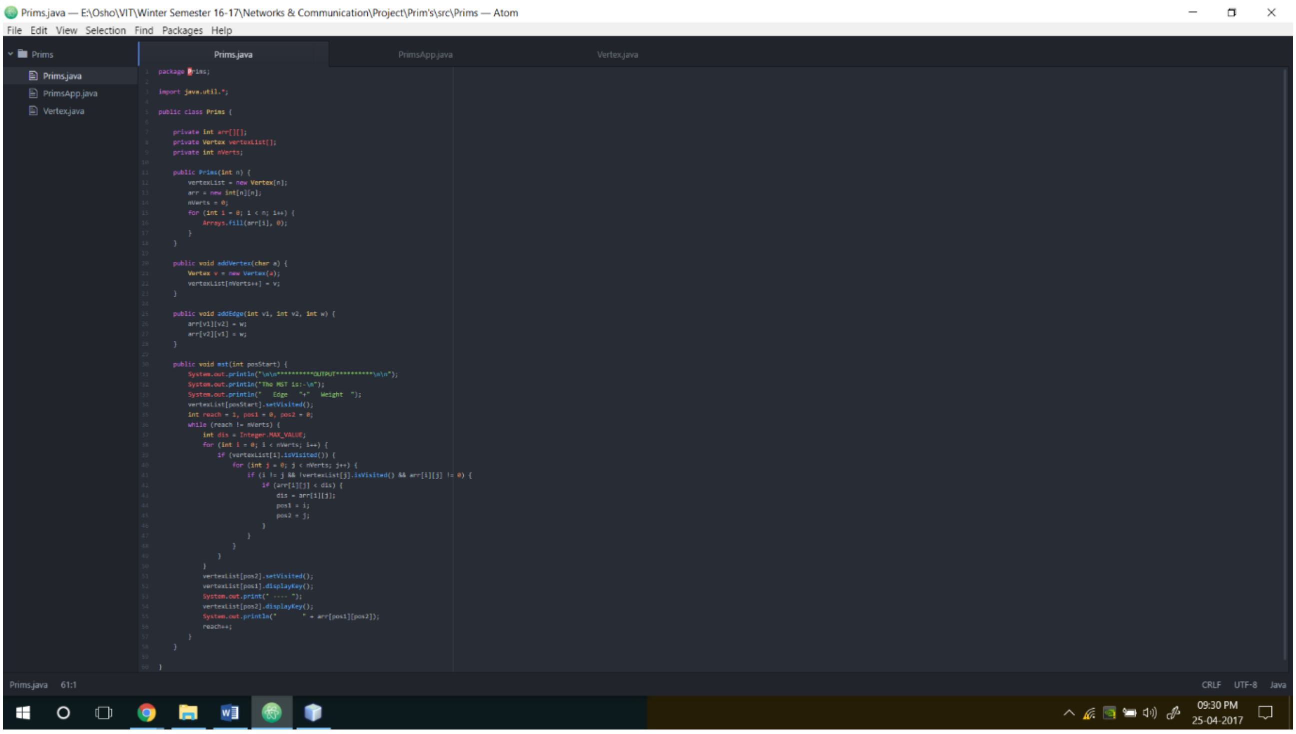


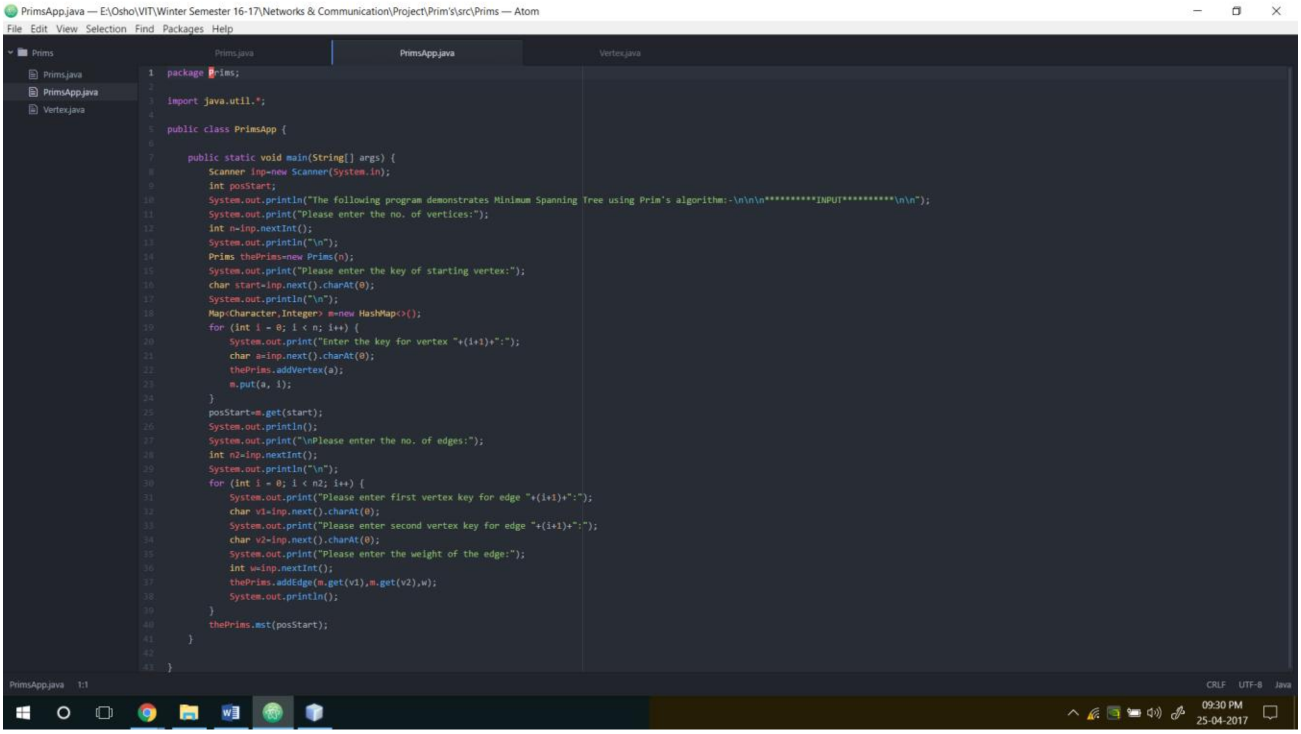
PSEUDO CODE

```
ReachSet = {0};
UnReachSet = {1, 2, ..., N-1};
   SpanningTree = {};

   while ( UnReachSet ≠ empty )
   {
      Find edge e = (x, y) such that:
       1. x ∈ ReachSet
       2. y ∈ UnReachSet
       3. e has smallest cost

      SpanningTree = SpanningTree ∪ {e};

      ReachSet    = ReachSet ∪ {y};
      UnReachSet = UnReachSet - {y};
   }
```

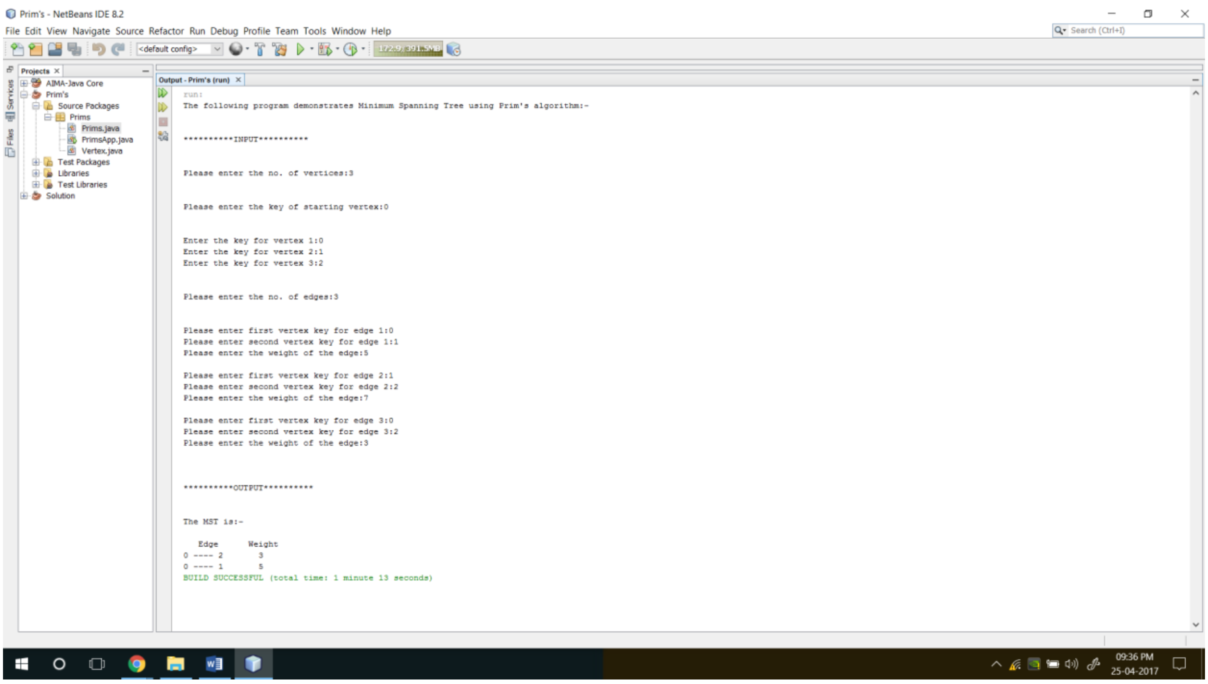
IMPLEMENTATION IN JAVA



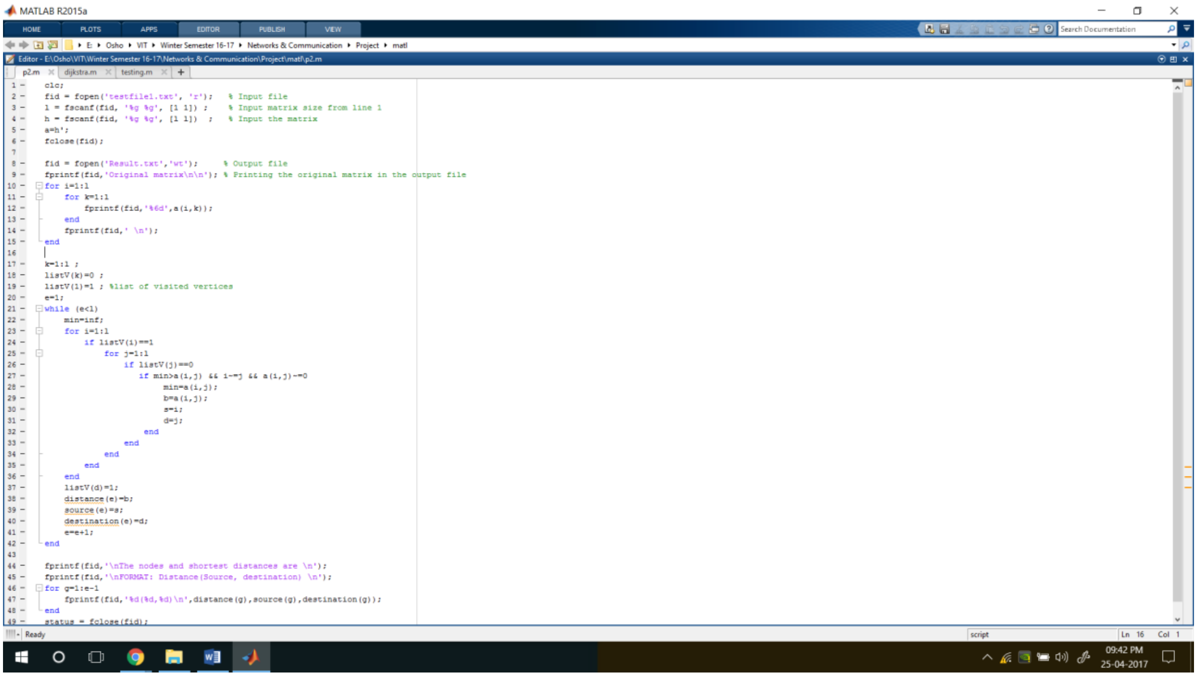



PrimsApp.java — E:\Osho\VIT\Winter Semester 16-17\Networks & Communication\Project\Prim's\src\Prims — Atom
File Edit View Selection Find Packages Help
Prims
Prims.java
PrimsApp.java
Vertex.java
package Prims;
import java.util.*;
public class PrimsApp {
public static void main(String[] args) {
Scanner inp=new Scanner(System.in);
int posStart;
System.out.println("The following program demonstrates Minimum Spanning Tree using Prim's algorithm:-\n\n\n**********INPUT**********\n\n");
System.out.print("Please enter the no. of vertices:");
int n=inp.nextInt();
System.out.println("\n");
Prims thePrims=new Prims(n);
System.out.print("Please enter the key of starting vertex:");
char start=inp.next().charAt(0);
System.out.println("\n");
Map<Character,Integer> m=new HashMap<>();
for (int i = 0; i < n; i++) {
System.out.print("Enter the key for vertex "+(i+1)+":");
char a=inp.next().charAt(0);
thePrims.addVertex(a);
m.put(a, i);
}
posStart=m.get(start);
System.out.println();
System.out.print("\nPlease enter the no. of edges:");
int n2=inp.nextInt();
System.out.println("\n");
for (int i = 0; i < n2; i++) {
System.out.print("Please enter first vertex key for edge "+(i+1)+":");
char v1=inp.next().charAt(0);
System.out.print("Please enter second vertex key for edge "+(i+1)+":");
char v2=inp.next().charAt(0);
System.out.print("Please enter the weight of the edge:");
int w=inp.nextInt();
thePrims.addEdge(m.get(v1),m.get(v2),w);
System.out.println();
}
thePrims.mst(posStart);
}
}
PrimsApp.java 1:1
CRLF UTF-8 Java
09:30 PM
25-04-2017

OUTPUT

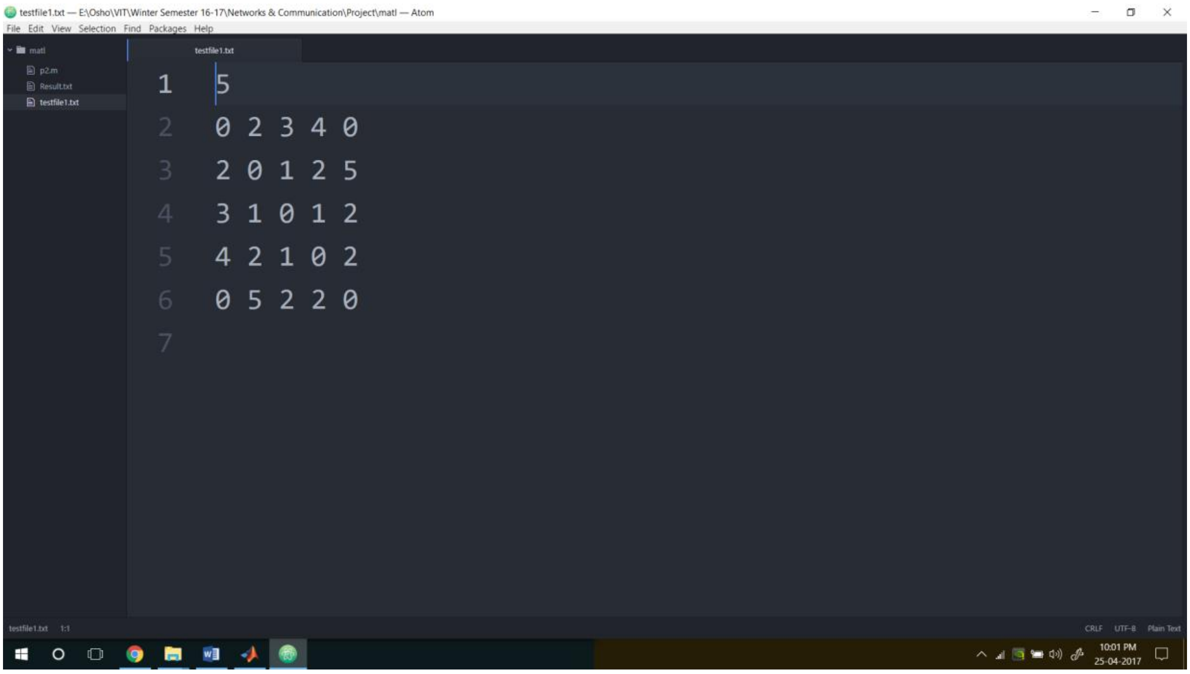

Prim's - NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Search (Ctrl+I)
Projects
AIMA-Java Core
Prim's
Source Packages
Prims
Prims.java
PrimsApp.java
Vertex.java
Test Packages
Libraries
Test Libraries
Solution
Output - Prim's (run)
run:
The following program demonstrates Minimum Spanning Tree using Prim's algorithm:-
**********INPUT**********
Please enter the no. of vertices:3
Please enter the key of starting vertex:0
Enter the key for vertex 1:0
Enter the key for vertex 2:1
Enter the key for vertex 3:2
Please enter the no. of edges:3
Please enter first vertex key for edge 1:0
Please enter second vertex key for edge 1:1
Please enter the weight of the edge:5
Please enter first vertex key for edge 2:1
Please enter second vertex key for edge 2:2
Please enter the weight of the edge:7
Please enter first vertex key for edge 3:0
Please enter second vertex key for edge 3:2
Please enter the weight of the edge:3
**********OUTPUT**********
The MST is:-
Edge Weight
0 ---- 2 3
0 ---- 1 5
BUILD SUCCESSFUL (total time: 1 minute 13 seconds)
09:36 PM
25-04-2017

IMPLEMENTATION IN MATLAB



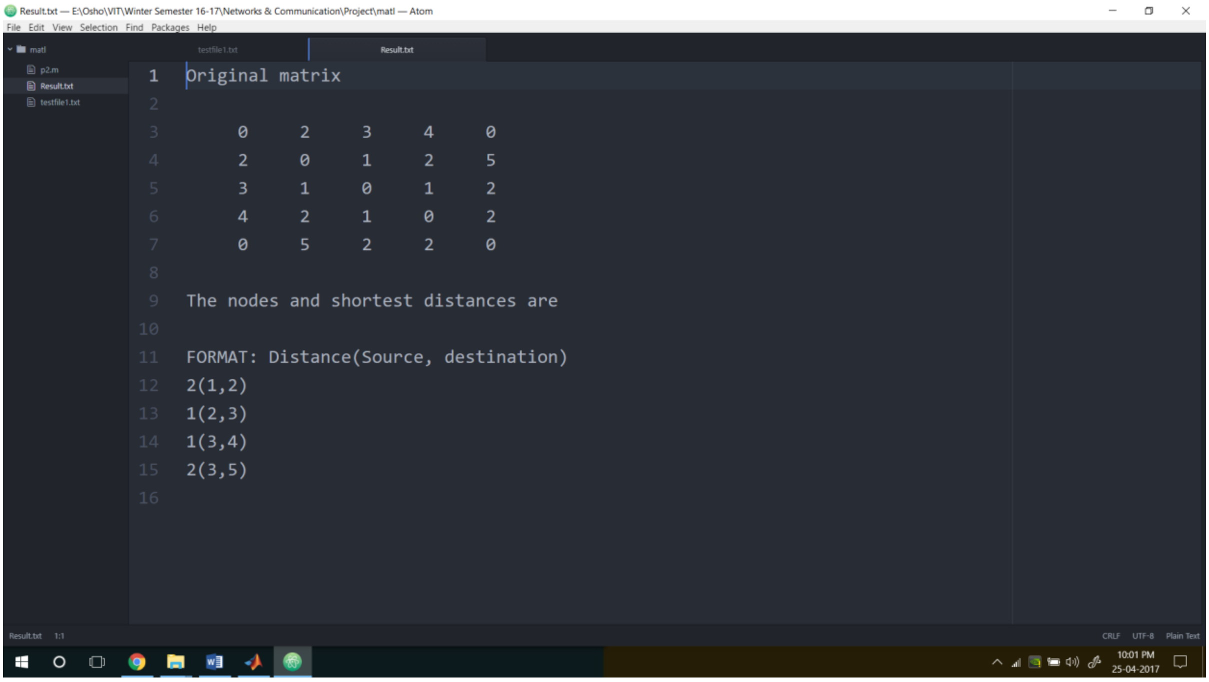
```
clc;
fid = fopen('testfile1.txt', 'r');    % Input file
l = fscanf(fid, '%g %g', [1 1]) ;     % Input matrix size from line 1
h = fscanf(fid, '%g %g', [l l])  ;    % Input the matrix
a=h';
fclose(fid);

fid = fopen('Result.txt','wt');      % Output file
fprintf(fid,'Original matrix\n\n'); % Printing the original matrix in the output file
for i=1:l
    for k=1:l
        fprintf(fid,'%6d',a(i,k));
    end
    fprintf(fid,' \n');
end

k=1:l ;
listV(k)=0 ;
listV(1)=1 ; %list of visited vertices
e=1;
while (e<l)
    min=inf;
    for i=1:l
        if listV(i)==1
            for j=1:l
                if listV(j)==0
                   if min>a(i,j) && i~=j && a(i,j)~=0
                        min=a(i,j);
                        b=a(i,j);
                        s=i;
                        d=j;
                   end
                end
            end
        end
    end
    listV(d)=1;
    distance(e)=b;
    source(e)=s;
    destination(e)=d;
    e=e+1;
end

fprintf(fid,'\nThe nodes and shortest distances are \n');
fprintf(fid,'\nFORMAT: Distance(Source, destination) \n');
for g=1:e-1
    fprintf(fid,'%d(%d,%d)\n',distance(g),source(g),destination(g));
end
status = fclose(fid);
```

INPUT



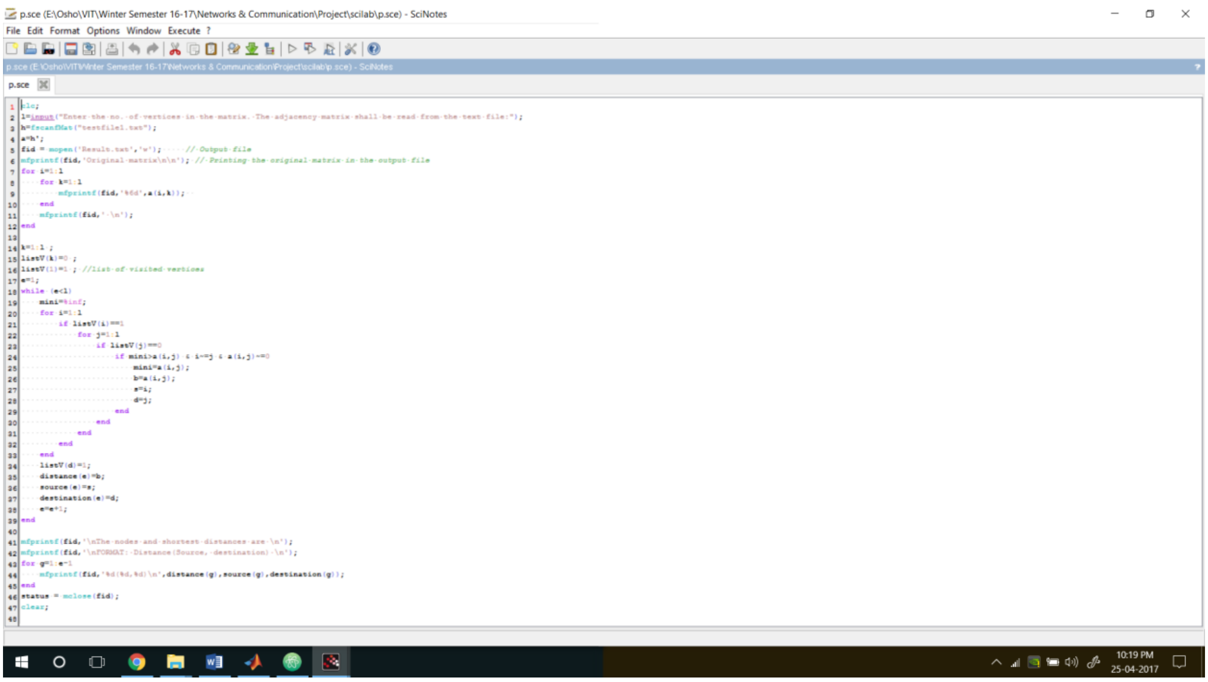
```
5
0 2 3 4 0
2 0 1 2 5
3 1 0 1 2
4 2 1 0 2
0 5 2 2 0
```

OUTPUT



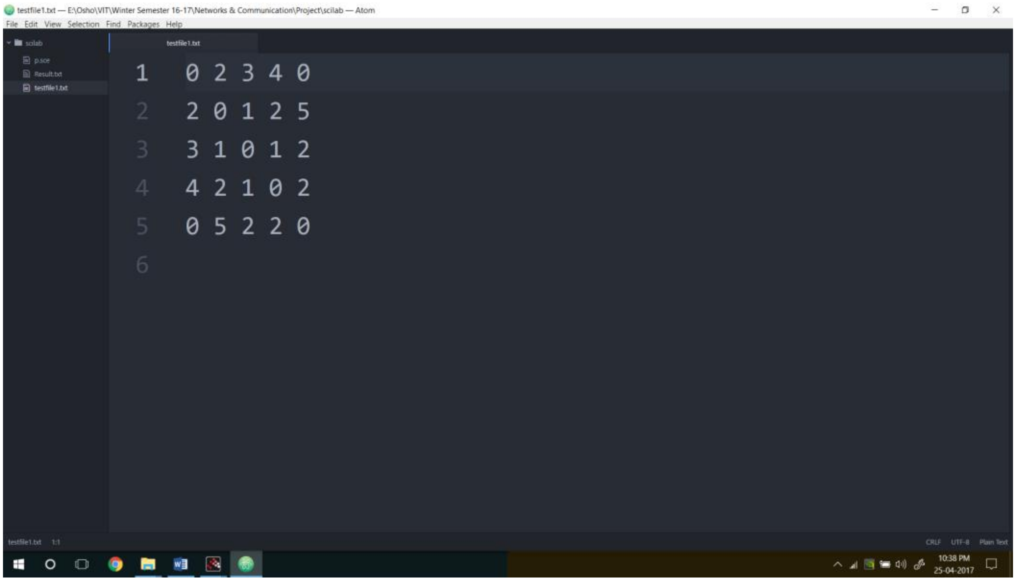
```
Original matrix

     0    2    3    4    0
     2    0    1    2    5
     3    1    0    1    2
     4    2    1    0    2
     0    5    2    2    0

The nodes and shortest distances are

FORMAT: Distance(Source, destination)
2(1,2)
1(2,3)
1(3,4)
2(3,5)
```

IMPLEMENTATION IN SCILAB

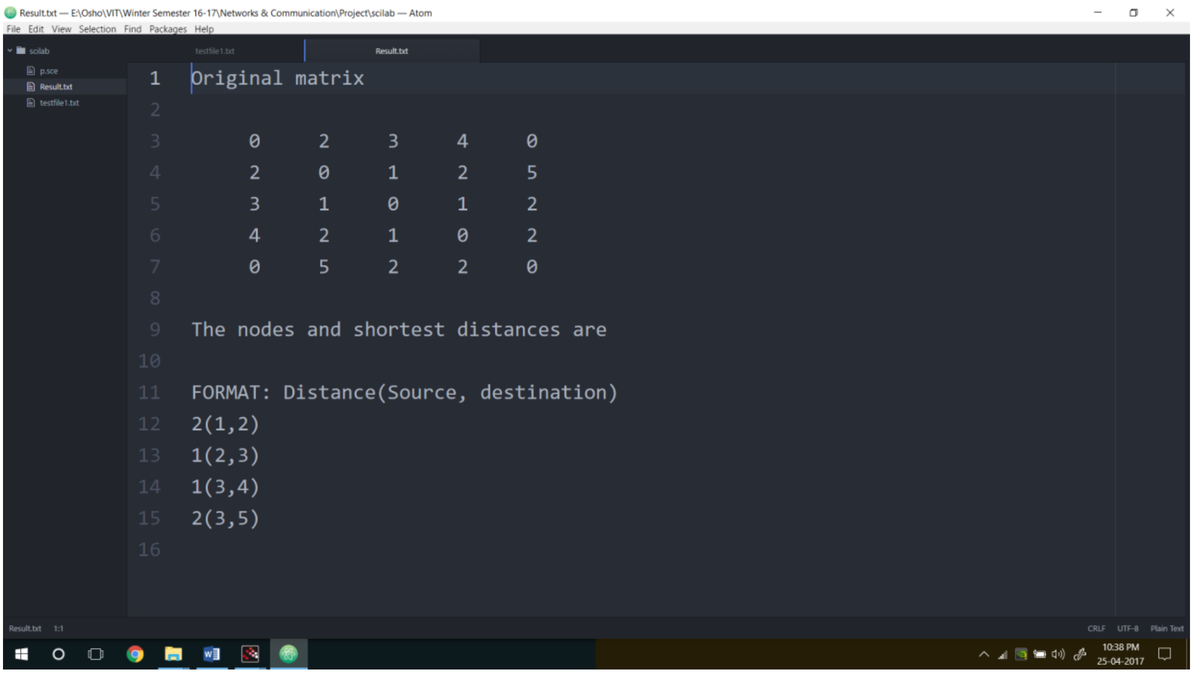


```
clc;
l=input("Enter the no. of vertices in the matrix. The adjacency matrix shall be read from the text file:");
h=fscanfMat("testfile1.txt");
a=h';
fid = mopen('Result.txt','w');     // Output file
mfprintf(fid,'Original matrix\n\n'); // Printing the original matrix in the output file
for i=1:l
    for k=1:l
        mfprintf(fid,'%6d',a(i,k));
    end
    mfprintf(fid,' \n');
end

k=1:l ;
listV(k)=0 ;
listV(1)=1 ; //list of visited vertices
e=1;
while (e<l)
    mini=%inf;
    for i=1:l
        if listV(i)==1
            for j=1:l
                if listV(j)==0
                    if mini>a(i,j) & i~=j & a(i,j)~=0
                        mini=a(i,j);
                        b=a(i,j);
                        s=i;
                        d=j;
                    end
                end
            end
        end
    end
    listV(d)=1;
    distance(e)=b;
    source(e)=s;
    destination(e)=d;
    e=e+1;
end

mfprintf(fid,'\nThe nodes and shortest distances are \n');
mfprintf(fid,'\nFORMAT: Distance(Source, destination) \n');
for g=1:e-1
    mfprintf(fid,'%d(%d,%d)\n',distance(g),source(g),destination(g));
end
status = mclose(fid);
clear;
```

INPUT



OUTPUT



KRUSKAL's ALGORITHM (UTSAV RAI)

- Kruskal's Algorithm and Prim's minimum spanning tree algorithm are two popular algorithms to find the minimum spanning trees.
- Kruskal's algorithm uses the greedy approach for finding a minimum spanning tree. Kruskal's algorithm treats every node as an independent tree and connects one with another only if it has the lowest cost compared to all other options available.
- Work with edges, rather than nodes

ALGORITHM

Two steps:

Sort edges by increasing edge weight

Select the first |V| – 1 edges that do not generate a cycle

Step to Kruskal's algorithm:

- Sort the graph edges with respect to their weights.
- Start adding edges to the minimum spanning tree from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which don't form a cycle—edges which connect only disconnected components.

Or as a simpler explanation,

Step 1 - Remove all loops and parallel edges

Step 2 - Arrange all the edges in ascending order of cost

Step 3 - Add edges with least weight

PSEUDOCODE

```
Let G = (V, E) be the given graph, with |V| = n
        {
            Start with a graph T = (V, ϕ) consisting of
only the
            vertices of G and no edges; /* This can be
viewed as n
            connected components, each vertex being one
connected component */
       Arrange E in the order of increasing costs;
        for (i = 1, i <=  n - 1, i + +)
        { Select the next smallest cost edge;
         if (the edge connects two different connected
components)
              add the edge to T;
        }
    }
```

PROOF OF CORRECTNESS

Theorem: Kruskal's algorithm finds a minimum spanning tree.

Proof: Let G = (V, E) be a weighted, connected graph. Let T be the edge set that is grown in Kruskal's algorithm.

The proof is by mathematical induction on the number of edges in T.

We show that if T is promising at any stage of the algorithm, then it is still promising when a new edge is added to it in Kruskal's algorithm

When the algorithm terminates, it will happen that T gives a solution to the problem and hence an MST.

Basis: T = ϕ is promising since a weighted connected graph always has at least one MST.

Induction Step: Let T be promising just before adding a new edge e = (u, v). The edges T divide the nodes of G into one or more connected components. u and v will be in two different components. Let U be the set of nodes in the component that includes u. Note that

U is a strict subset of V

T is a promising set of edges such that no edge in T leaves U (since an edge T either has both ends in U or has neither end in U)

e is a least cost edge that leaves U (since Kruskal's algorithm, being greedy, would have chosen e only after examining edges shorter than e)

The above three conditions are precisely like in the MST Lemma and hence we can conclude that the T Union {e} is also promising. When the algorithm stops, T gives not merely a spanning tree but a minimal spanning tree since it is promising.

APPLICATIONS

Minimum Spanning Tree (MST) problem: Given connected graph G with positive edge weights, find a min weight set of edges that connects all of the vertices.

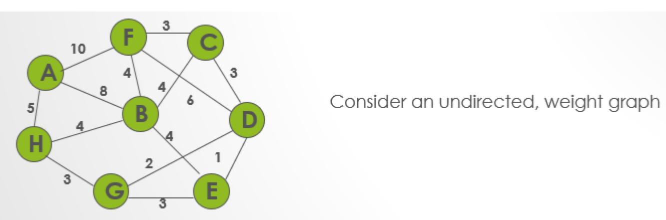
MST is fundamental problem with diverse applications

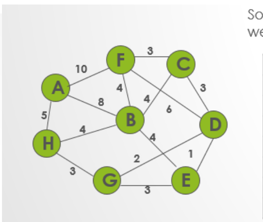
Network design.
– telephone, electrical, hydraulic, TV cable, computer, road
The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money.

Approximation algorithms for NP-hard problems.
– traveling salesperson problem, Steiner tree
A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once.

Cluster analysis
k clustering problem can be viewed as finding an MST and deleting the k-1 most
expensive edges.

WALK-THROUGH

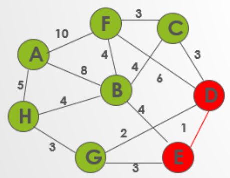




Sort the edges by increasing edge weight

edge	d_v	
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

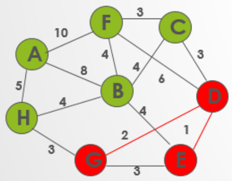
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

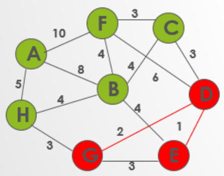
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

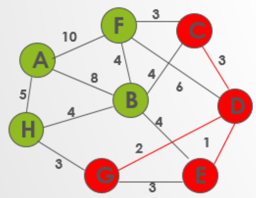


Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

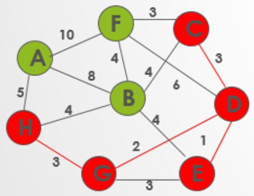
Accepting edge (E,G) would create a cycle



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	
(C,F)	3	
(B,C)	4	

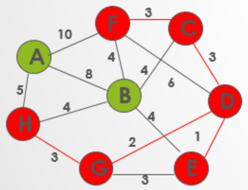
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	
(B,C)	4	

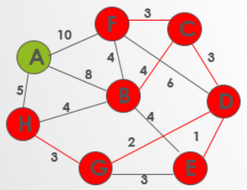
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	

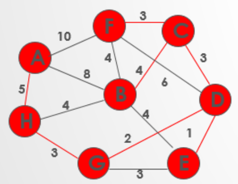
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

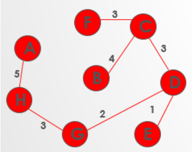
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

} not considered

Done

Total Cost = $\Sigma\, d_v = 21$

TIME COMPLEXITY

O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply find-union algorithm. The find and union operations can take atmost O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be atmost O(V2), so O(LogV) are O(LogE) same. Therefore, overall time complexity is O(ElogE) or O(ElogV).

JAVA PROGRAM

```java
//Java program for Kruskal's algorithm to find Minimum Spanning Tree
//of a given connected, undirected and weighted graph
        import java.util.*;
        import java.lang.*;
        import java.io.*;

class Graph
{
    // A class to represent a graph edge
    class Edge implements Comparable<Edge>
    {
        int src, dest, weight;

        // Comparator function used for sorting edges based on
        // their weight
        public int compareTo(Edge compareEdge)
        {
            return this.weight-compareEdge.weight;
        }
    };

    // A class to represent a subset for union-find
    class subset
    {
        int parent, rank;
    };

    int V, E;    // V-> no. of vertices & E->no.of edges
    Edge edge[]; // collection of all edges

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[E];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }
```

```java
// A utility function to find set of an element i
// (uses path compression technique)
int find(subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

        // If ranks are same, then make one as root and increment
        // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST()
{
    Edge result[] = new Edge[V];  // This will store the resultant MST
    int e = 0;  // An index variable, used for result[]
    int i = 0;  // An index variable, used for sorted edges
    for (i=0; i<V; ++i)
        result[i] = new Edge();

    // Step 1:  Sort all the edges in non-decreasing order of their
    // weight.  If we are not allowed to change the given graph, we
    // can create a copy of array of edges
    Arrays.sort(edge);

    // Allocate memory for creating V subsets
    subset subsets[] = new subset[V];
    for(i=0; i<V; ++i)
        subsets[i]=new subset();

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    i = 0;  // Index used to pick next edge

    // Number of edges to be taken is equal to V-1
    while (e < V - 1)
    {
        // Step 2: Pick the smallest edge. And increment the index
        // for next iteration
```

```
            Edge next_edge = new Edge();
            next_edge = edge[i++];

            int x = find(subsets, next_edge.src);
            int y = find(subsets, next_edge.dest);

            // If including this edge does't cause cycle, include it
            // in result and increment the index of result for next edge
            if (x != y)
            {
                result[e++] = next_edge;
                Union(subsets, x, y);
            }
            // Else discard the next_edge
        }

        // print the contents of result[] to display the built MST
        System.out.println("Following are the edges in the constructed MST");
        for (i = 0; i < e; ++i)
            System.out.println((result[i].src+1)+" -- "+(result[i].dest+1) +" == "+
                    result[i].weight);
    }

    // Driver Program
    public static void main (String[] args)
    {

     /*weighted graph
              10
         1--------2
         |   \      |
        6|    5\    |15
         |        \ |
         3--------4
             4        */
        int V = 4;  // Number of vertices in graph
        int E = 5;  // Number of edges in graph
        Graph graph = new Graph(V, E);

        // add edge 0-1
        graph.edge[0].src = 0;
        graph.edge[0].dest = 1;
        graph.edge[0].weight = 10;

        // add edge 0-2
        graph.edge[1].src = 0;
        graph.edge[1].dest = 2;
        graph.edge[1].weight = 6;

        // add edge 0-3
        graph.edge[2].src = 0;
        graph.edge[2].dest = 3;
        graph.edge[2].weight = 5;

        // add edge 1-3
        graph.edge[3].src = 1;
        graph.edge[3].dest = 3;
        graph.edge[3].weight = 15;

        // add edge 2-3
        graph.edge[4].src = 2;
        graph.edge[4].dest = 3;
        graph.edge[4].weight = 4;

        graph.KruskalMST();
    }
}
```

OUTPUT

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console
<terminated> Graph [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (May 1, 2017, 10:25:04 PM)
Following are the edges in the constructed MST
3 -- 4 == 4
1 -- 4 == 5
1 -- 2 == 10
```

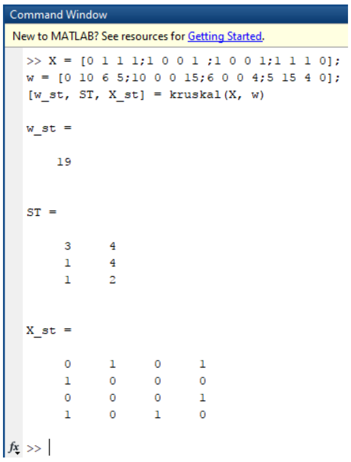
MATLAB CODE

```
function [w_st, ST, X_st] = kruskal(X, w)
% function [w_st, ST, X_st] = kruskal(X, w)
%
% This function finds the minimum spanning tree of the graph where each
% edge has a specified weight using the Kruskal's algorithm.
%
% Assumptions
% -----------
%      N:  1x1   scalar      -  Number of nodes (vertices) of the graph
%     Ne:  1x1   scalar      -  Number of edges of the graph
%    Nst:  1x1   scalar      -  Number of edges of the minimum spanning tree
%
% We further assume that the graph is labeled consecutively. That is, if
% there are N nodes, then nodes will be labeled from 1 to N.
%
% INPUT
%
%      X:  NxN logical      -  Adjacency matrix
%              matrix          If X(i,j)=1, this means there is directed
edge
%                              starting from node i and ending in node j.
%                              Each element takes values 0 or 1.
%                              If X symmetric, graph is undirected.
%
%   or     Nex2 double      -  Neighbors' matrix
%               matrix         Each row represents an edge.
%                              Column 1 indicates the source node, while
%                              column 2 the target node.
%
%      w:  NxN double       -  Weight matrix in adjacency form
%              matrix          If X symmetric (undirected graph), w has to
%                              be symmetric.
%
%   or     Nex1 double      -  Weight matrix in neighbors' form
%               matrix         Each element represents the weight of that
```

```
%                             edge.
%
%
% OUTPUT
%
%  w_st:    1x1 scalar     -  Total weight of minimum spanning tree
%    ST:  Nstx2 double     -  Neighbors' matrix of minimum spanning tree
%               matrix
%  X_st:  NstxNst logical  -  Adjacency matrix of minimum spanning tree
%                 matrix      If X_st symmetric, tree is undirected.
%
    isUndirGraph = 1;

    % Convert logical adjacent matrix to neighbors' matrix
    if size(X,1)==size(X,2) && sum(X(:)==0)+sum(X(:)==1)==numel(X)
        if any(any(X-X'))
            isUndirGraph = 0;
        end
        ne = cnvrtX2ne(X,isUndirGraph);
    else
        if size(unique(sort(X,2),'rows'),1)~=size(X,1)
            isUndirGraph = 0;
        end
        ne = X;
    end

    % Convert weight matrix from adjacent to neighbors' form
    if numel(w)~=length(w)
        if isUndirGraph && any(any(w-w'))
            error('If it is an undirected graph, weight matrix has to be
symmetric.');
        end
        w = cnvrtw2ne(w,ne);
    end
    N    = max(ne(:));    % number of vertices
    Ne   = size(ne,1);    % number of edges
    lidx = zeros(Ne,1);   % logical edge index; 1 for the edges that will be
                          % in the minimum spanning tree
    % Sort edges w.r.t. weight
    [w,idx] = sort(w);
    ne      = ne(idx,:);

    % Initialize: assign each node to itself
    [repr, rnk] = makeset(N);

    % Run Kruskal's algorithm
    for k = 1:Ne
        i = ne(k,1);
        j = ne(k,2);
        if fnd(i,repr) ~= fnd(j,repr)
            lidx(k) = 1;
            [repr, rnk] = union(i, j, repr, rnk);
        end
    end
    % Form the minimum spanning tree
    treeidx = find(lidx);
    ST      = ne(treeidx,:);

    % Generate adjacency matrix of the minimum spanning tree
    X_st = zeros(N);
```

```
    for k = 1:size(ST,1)
        X_st(ST(k,1),ST(k,2)) = 1;
        if isUndirGraph,  X_st(ST(k,2),ST(k,1)) = 1;  end
    end
    % Evaluate the total weight of the minimum spanning tree
    w_st = sum(w(treeidx));
end
function ne = cnvrtX2ne(X, isUndirGraph)
    if isUndirGraph
        ne = zeros(sum(sum(X.*triu(ones(size(X))))),2);
    else
        ne = zeros(sum(X(:)),2);
    end
    cnt = 1;
    for i = 1:size(X,1)
        v       = find(X(i,:));
        if isUndirGraph
            v(v<=i) = [];
        end
        u       = repmat(i, size(v));
        edges   = [u; v]';
        ne(cnt:cnt+size(edges,1)-1,:) = edges;
        cnt = cnt + size(edges,1);
    end
end
function w = cnvrtw2ne(w,ne)
    tmp = zeros(size(ne,1),1);
    cnt = 1;
    for k = 1:size(ne,1)
        tmp(cnt) = w(ne(k,1),ne(k,2));
        cnt = cnt + 1;
    end
    w = tmp;
end
function [repr, rnk] = makeset(N)
    repr = (1:N);
    rnk  = zeros(1,N);
end

function o = fnd(i,repr)
    while i ~= repr(i)
        i = repr(i);
    end
    o = i;
end
function [repr, rnk] = union(i, j, repr, rnk)
    r_i = fnd(i,repr);
    r_j = fnd(j,repr);
    if rnk(r_i) > rnk(r_j)
        repr(r_j) = r_i;
    else
        repr(r_i) = r_j;
        if rnk(r_i) == rnk(r_j)
            rnk(r_j) = rnk(r_j) + 1;
        end
    end
end
```

OUTPUT



```
Command Window
New to MATLAB? See resources for Getting Started.
>> X = [0 1 1 1;1 0 0 1 ;1 0 0 1;1 1 1 0];
w = [0 10 6 5;10 0 0 15;6 0 0 4;5 15 4 0];
[w_st, ST, X_st] = kruskal(X, w)

w_st =

    19


ST =

     3     4
     1     4
     1     2


X_st =

     0     1     0     1
     1     0     0     0
     0     0     0     1
     1     0     1     0

>> 
```

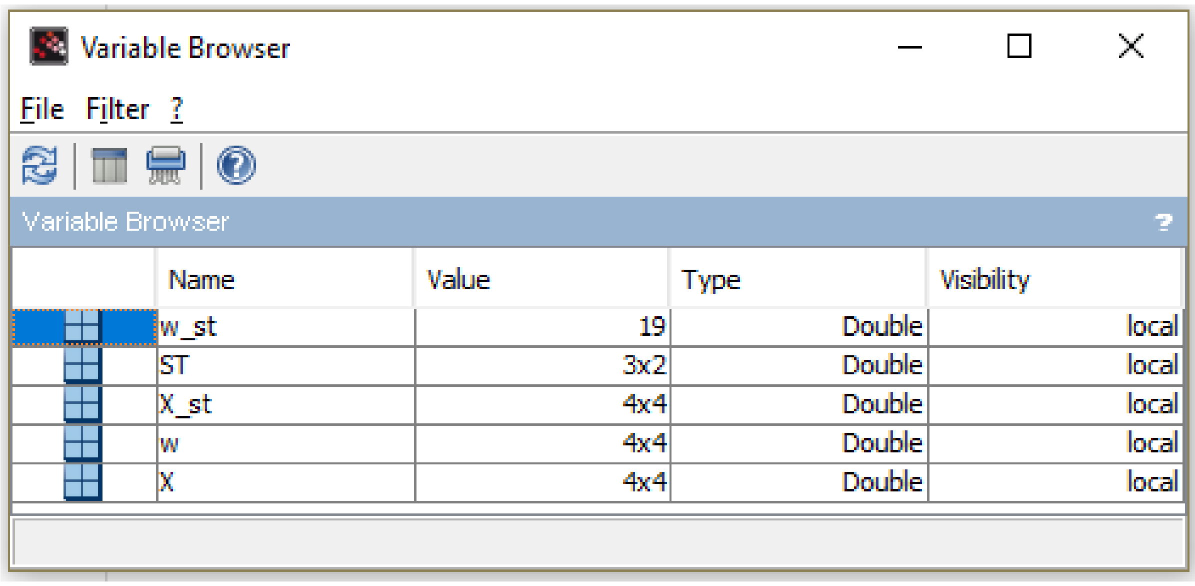
SCILAB CODE

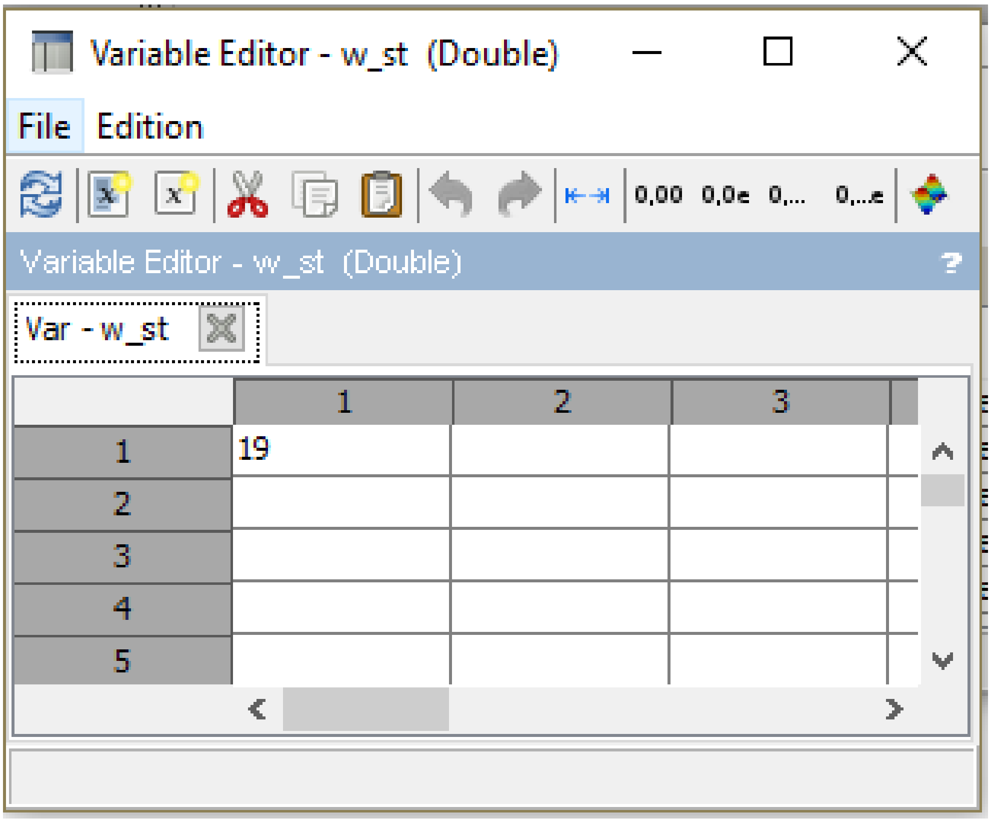
```
// This function finds the minimum spanning tree of the graph where each
// edge has a specified weight using the Kruskal's algorithm.

// Assumptions
// -----------
//    N:  1x1  scalar     -  Number of nodes (vertices) of the graph
//   Ne:  1x1  scalar     -  Number of edges of the graph
//  Nst:  1x1  scalar     -  Number of edges of the minimum spanning tree
//
// We further assume that the graph is labeled consecutively. That is, if
// there are N nodes, then nodes will be labeled from 1 to N.
//
// INPUT
//
//    X:  NxN logical     -  Adjacency matrix
//          matrix          If X(i,j)=1, this means there is directed edge
//                          starting from node i and ending in node j.
//                          Each element takes values 0 or 1.
//                          If X symmetric, graph is undirected.
//
//  or     Nex2 double    -  Neighbors' matrix
//          matrix          Each row represents an edge.
//                          Column 1 indicates the source node, while
//                          column 2 the target node.
//
//    w:  NxN double      -  Weight matrix in adjacency form
//          matrix          If X symmetric (undirected graph), w has to
//                          be symmetric.
//
//  or     Nex1 double    -  Weight matrix in neighbors' form
//          matrix          Each element represents the weight of that
//                          edge.
//
//
```

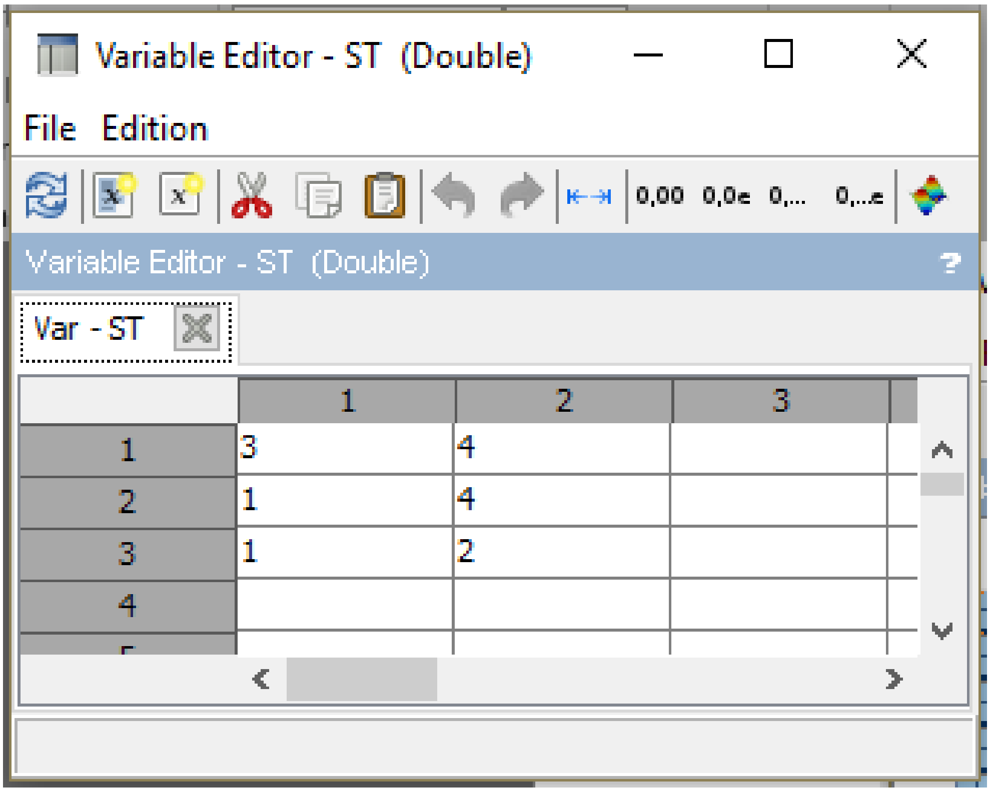
```
// OUTPUT
//
// w_st:   1x1 scalar     - Total weight of minimum spanning tree
//   ST:  Nstx2 double    - Neighbors' matrix of minimum spanning tree
//            matrix
// X_st:  NstxNst logical  - Adjacency matrix of minimum spanning tree
//              matrix      If X_st symmetric, tree is undirected.
//
// The above function gives us the minimum directed spanning tree.
funcprot(0)
function [w_st, ST, X_st]=kruskal(X, w)

  isUndirGraph = 1;
  if size(X,1)==size(X,2) & sum(X(:)==0)+sum(X(:)==1)==length(X)
    if or(or(X-X'))
      isUndirGraph = 0;
    end
    ne = cnvrtX2ne(X,isUndirGraph);
  else
    if size(unique(sort(X,2),'rows'),1)~=size(X,1)
      isUndirGraph = 0;
    end
    ne = X;
  end

  if length(w)~=max(size(w))
    if isUndirGraph & or(or(w-w'))
      error('If it is an undirected graph, weight matrix has to be symmetric.');
    end
    w = cnvrtw2ne(w,ne);
  end

  N    = max(ne(:));  // Number of vertices
  Ne   = size(ne,1);    //number of edges
  lidx = zeros(Ne,1); //  logical edge index; 1 for the edges that will be in the minimum spanning tree

  //Sort edges w.r.t. weight
  [w,idx] = gsort(w,'lr','i');
  ne      = ne(idx,:);

  // % Initialize: assign each node to itself
  [repr, rnk] = makeset(N);


  //  Run Kruskal's algorithm
  for k = 1:Ne
    i = ne(k,1);
    j = ne(k,2);
    if fnd(i,repr) ~= fnd(j,repr)
      lidx(k) = 1;
      [repr, rnk] = union(i, j, repr, rnk);
    end
  end

  // Form the minimum spanning tree
  treeidx = find(lidx);
  ST      = ne(treeidx,:);

  //Generate adjacency matrix of the minimum spanning tree
  X_st = zeros(N);
  for k = 1:size(ST,1)
    X_st(ST(k,1),ST(k,2)) = 1;
    if isUndirGraph,  X_st(ST(k,2),ST(k,1)) = 1;  end
  end
  //Evaluate the total weight of the minimum spanning tree
  w_st = sum(w(treeidx));
```

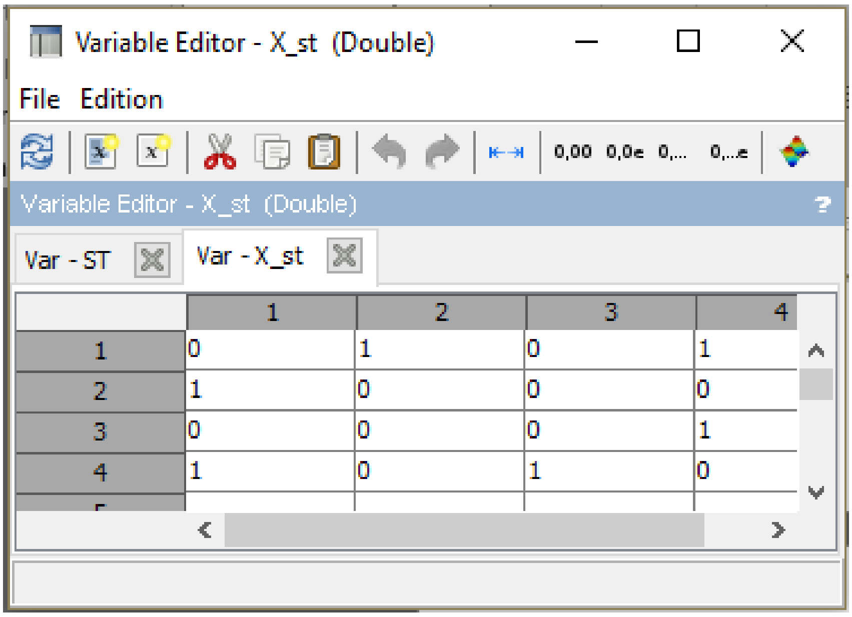
```
endfunction
funcprot(0)
function ne=cnvrtX2ne(X, isUndirGraph)
   if isUndirGraph
      ne = zeros(sum(sum(X.*triu(ones(X)))),2);
   else
      ne = zeros(sum(X(:)),2);
   end
   cnt = 1;
   for i = 1:size(X,1)
      v      = find(X(i,:));
      if isUndirGraph
         v(v<=i) = [];
      end
      u      = repmat(i, size(v));
      edges  = [u; v]';
      ne(cnt:cnt+size(edges,1)-1,:) = edges;
      cnt = cnt + size(edges,1);
   end
endfunction

function w=cnvrtw2ne(w, ne)
   tmp = zeros(size(ne,1),1);
   cnt = 1;
   for k = 1:size(ne,1)
      tmp(cnt) = w(ne(k,1),ne(k,2));
      cnt = cnt + 1;
   end
   w = tmp;
endfunction
funcprot(0)
function [repr, rnk]=makeset(N)
   repr = (1:N);
   rnk  = zeros(1,N);
endfunction

function o=fnd(i, repr)
   while i ~= repr(i)
      i = repr(i);
   end
   o = i;
endfunction
funcprot(0)
function [repr, rnk]=union(i, j, repr, rnk)
   r_i = fnd(i,repr);
   r_j = fnd(j,repr);
   if rnk(r_i) > rnk(r_j)
      repr(r_j) = r_i;
   else
      repr(r_i) = r_j;
      if rnk(r_i) == rnk(r_j)
         rnk(r_j) = rnk(r_j) + 1;
      end
   end
endfunction
X = [0 1 1 1;1 0 0 1 ;1 0 0 1;1 1 1 0];
w = [0 10 6 5;10 0 0 15;6 0 0 4;5 15 4 0];
[w_st, ST, X_st] = kruskal(X, w);
```

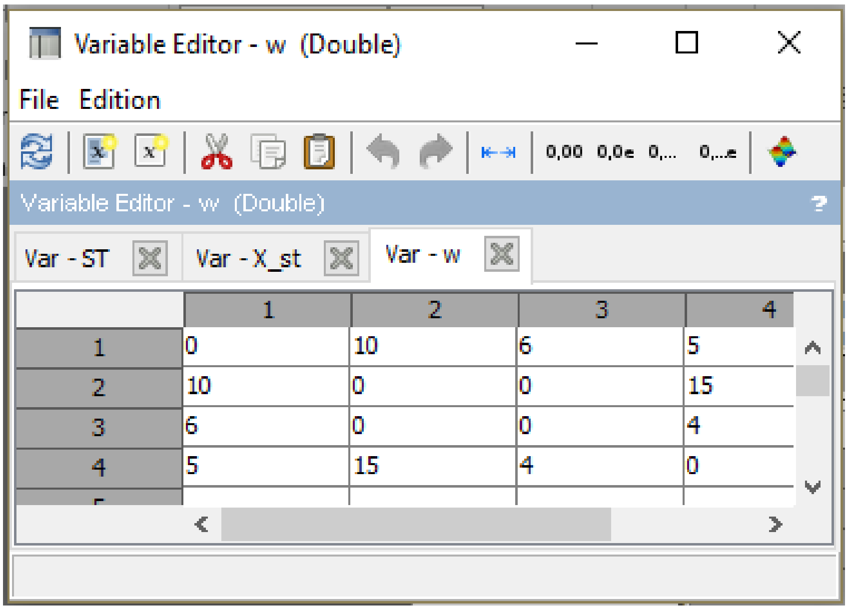
OUTPUT

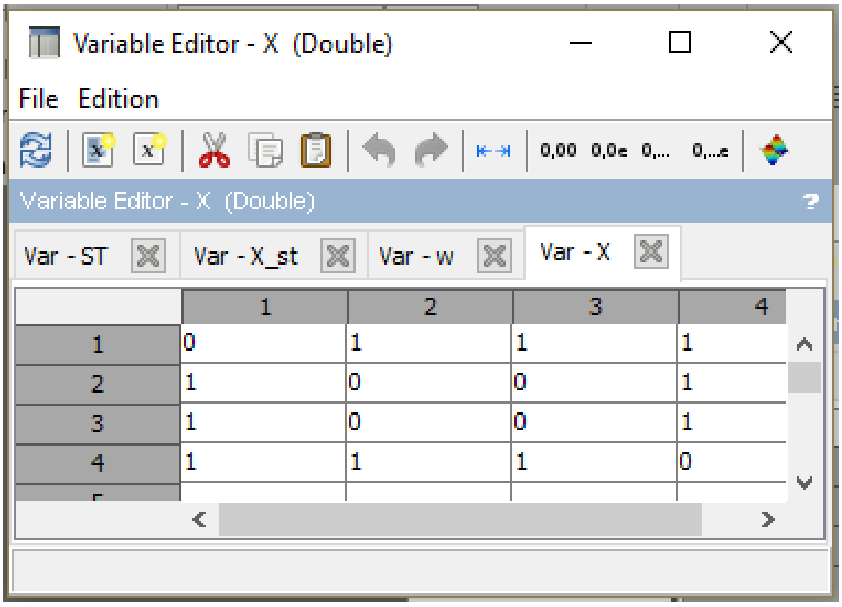

Variable Browser
File Filter ?
Variable Browser
Name
Value
Type
Visibility
w_st 19 Double local
ST 3x2 Double local
X_st 4x4 Double local
w 4x4 Double local
X 4x4 Double local


Variable Editor - w_st (Double)
File Edition
0,00 0,0e 0,... 0,...e
Variable Editor - w_st (Double)
Var - w_st
1 2 3
1 19
2
3
4
5


Variable Editor - ST (Double)
File Edition
0,00 0,0e 0,... 0,...e
Variable Editor - ST (Double)
Var - ST
1 2 3
1 3 4
2 1 4
3 1 2
4


Variable Editor - X_st (Double)
File Edition
0,00 0,0e 0,... 0,...e
Variable Editor - X_st (Double)
Var - ST
Var - X_st
1 2 3 4
1 0 1 0 1
2 1 0 0 0
3 0 0 0 1
4 1 0 1 0


Variable Editor - w (Double)
File Edition
0,00 0,0e 0,... 0,...e
Variable Editor - w (Double)
Var - ST
Var - X_st
Var - w
1 2 3 4
1 0 10 6 5
2 10 0 0 15
3 6 0 0 4
4 5 15 4 0


Variable Editor - X (Double)
File Edition
0,00 0,0e 0,... 0,...e
Variable Editor - X (Double)
Var - ST
Var - X_st
Var - w
Var - X
1 2 3 4
1 0 1 1 1
2 1 0 0 1
3 1 0 0 1
4 1 1 1 0

BELLMAN-FORD ALGORITHM (SIDDHARTH CHANDRA)

The Bellman-Ford algorithm is a graph search algorithm that finds the shortest path between a given source vertex and all other vertices in the graph. This algorithm can be used on both weighted and unweighted graphs.
Like Dijkstra's shortest path algorithm , the Bellman-Ford algorithm is guaranteed to find the shortest path in a graph. Though it is slower than Dijkstra's algorithm, Bellman-Ford is capable of handling graphs that contain negative edge weights, so it is more versatile. It is worth noting that if there exists a negative cycle in the graph, then there is no shortest path. Going around the negative cycle an infinite number of times would continue to decrease the cost of the path (even though the path length is increasing). Because of this, Bellman-Ford can also detect negative cycles which is a useful feature.
The Bellman-Ford algorithm, like Dijkstra's algorithm, uses the principle of relaxation to find increasingly accurate path length. Bellman-Ford, though, tackles two main issues with this process.

1. If there are negative weight cycles, the search for a shortest path will go on forever.
2. Choosing a bad ordering for relaxations leads to exponential relaxations.

The detection of negative cycles is important, but the main contribution of this algorithm is in its ordering of relaxations. Dijkstra's algorithm is a greedy algorithm that selects the nearest vertex that has not been processed. Bellman-Ford, on the otherhand, relaxes all of the edges.

PSEUDO-CODE

The pseudo-code for the Bellman-Ford algorithm is quite short.This is high level description of Bellman-Ford written with pseudo-code, not an implementation

```
for v in V:
     v.distance = infinity
     v.p = None
     source.distance = 0
     for i from 1 to |V| - 1:
          for (u, v) in E:
               relax(u, v)
```

The first for loop sets the distance to each vertex in the graph to infinity. This is later changed for the source vertex to equal zero. Also in that first for loop, the p value for each vertex is set to nothing. This value is a pointer to a predecessor
vertex so that we can create a path later.
The next for loop simply goes through each edge (u, v) in E and relaxes it. This process is done |V| - 1 times.

Relax Equation

```
relax(u, v):
     if v.distance > u.distance + weight(u, v):
          v.distance = u.distance + weight(u, v)
          v.p = u
```

Detecting Negative Cycle

```
for v in V:
     v.distance = infinity
     v.p = None
source.distance = 0
for i from 1 to |V| - 1:
     for (u, v) in E:
          relax(u, v)
for (u, v) in E:
     if v.distance > u.distance + weight(u, v):
          print "A negative weight cycle exists"
```

OUTPUT IN C++

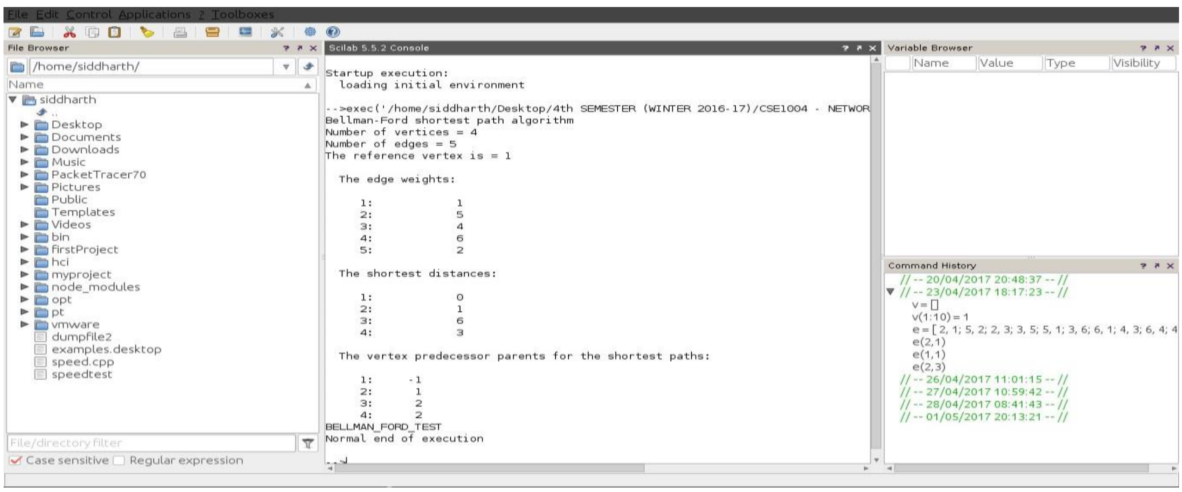
```
siddharth@siddharth-Inspiron-3541: ~/Desktop/4th SEMESTER (WINTER 2016-17)/CSE1
siddharth@siddharth-Inspiron-3541:~/Desktop/4th SEMESTER (WINTER 2016-17)/CSE100
4 - NETWORK AND COMMUNICATION/PROJECT$ g++ bell.cpp
siddharth@siddharth-Inspiron-3541:~/Desktop/4th SEMESTER (WINTER 2016-17)/CSE100
4 - NETWORK AND COMMUNICATION/PROJECT$ ./a.out
Enter the Number of Vertices -
4
Enter the Number of Edges -
5
Enter the Edges V1 -> V2, of weight W
1 2 1
2 3 5
3 1 4
2 4 2
4 3 6

The Adjacency List-
adjacencyList[1]  -> 2(1)
adjacencyList[2]  -> 3(5) -> 4(2)
adjacencyList[3]  -> 1(4)
adjacencyList[4]  -> 3(6)

Enter a Start Vertex -
1
No Negative Cycles exist in the Graph -
```

```
siddharth@siddharth-Inspiron-3541: ~/Desktop/4th SEMESTER (WINTER 2016-17)/CSE1
1 2 1
2 3 5
3 1 4
2 4 2
4 3 6

The Adjacency List-
adjacencyList[1]  -> 2(1)
adjacencyList[2]  -> 3(5) -> 4(2)
adjacencyList[3]  -> 1(4)
adjacencyList[4]  -> 3(6)

Enter a Start Vertex -
1
No Negative Cycles exist in the Graph -


Vertex    Shortest Distance to Vertex 1     Parent Vertex-
1         0                                 0
2         1                                 1
3         6                                 2
4         3                                 2
siddharth@siddharth-Inspiron-3541:~/Desktop/4th SEMESTER (WINTER 2016-17)/CSE100
4 - NETWORK AND COMMUNICATION/PROJECT$
```

IMPLEMENTATION IN SCILAB

```
function [ v_weight, predecessor ] = bellman_ford ( v_num, e_num, source, e, e_weight )
 r8_big = 1.0E+30;   v_weight = [];   predecessor = [];   v_weight(1:v_num) = r8_big;   v_weight(source) = 0;
predecessor(1:v_num) = -1;
 for i = 1 : v_num     for j = 1 : e_num       u = e(j,2);       v = e(j,1);
     t = v_weight(u) + e_weight(j);       if ( t < v_weight(v) ) then        v_weight(v) = t;        predecessor(v) = u;
     end     end   end
 for j = 1 : e_num     u = e(j,2);     v = e(j,1);
   if ( v_weight(u) + e_weight(j) < v_weight(v) )       mprintf ( '\n' );
     mprintf ( 'BELLMAN_FORD - Fatal error!\n' );       mprintf ( '  Graph contains a cycle with negative weight.\n' );
     error ( 'BELLMAN_FORD - Fatal error!' );     end   end
 return endfunction
function i4vec_print ( n, a, ti )
 mprintf ( '\n' );   mprintf ( '%s\n', ti );   mprintf ( '\n' );
 for i = 1 : n
   mprintf ( '%6d: %6d\n', i, a(i) );   end
 return endfunction
function r8vec_print ( n, a, ti )
 mprintf ('\n' );   mprintf ( '%s\n', ti );   mprintf ( '\n' );   for i = 1 : n
   mprintf ( '%6d: %12g\n', i, a(i) );   end
 return endfunction
function bellman_ford_test01 ( )
 e_num = 5;   v_num = 4;
 e = [ 2, 1; 3, 2; 1, 3; 3, 4; 4, 2];
 e_weight = [ 1;5;4;6;2];

 source = 1;   mprintf ( 'Bellman-Ford shortest path algorithm\n' );
 mprintf( 'Number of vertices = %d\n', v_num);   mprintf ( 'Number of edges = %d\n', e_num);   mprintf( 'The
reference vertex is = %d\n', source);   r8vec_print ( e_num, e_weight, '  The edge weights:' );
 [ v_weight, predecessor ] = bellman_ford ( v_num, e_num, source, e,e_weight );   r8vec_print ( v_num, v_weight, '  The
shortest distances:' );
 i4vec_print ( v_num, predecessor, '  The vertex predecessor parents for the shortest paths:' );
 return endfunction function bellman_ford_test ( )
 bellman_ford_test01;
 mprintf ( 'BELLMAN_FORD_TEST\n' );   mprintf ( 'Normal end of execution\n' );
 return endfunction bellman_ford_test;
```

OUTPUT



DIJKSTRA's ALGORITHM (KASHISH MIGLANI)

This is a single source shortest path algorithm . This algorithm helps in finding the shortest distance from one node to other nodes present in the graph. Generally we use it for finding the shortest distance of one node from all other node rather than finding the shortest distance between any two nodes . It is an extended application of the djikstra's algorithm.

This is one of the most reliable and fastest algorithm which helps in finding the shortest path.
We also use this algorithm for finding the path which we followed to obtain the lowest cost.

I have Implemented this Algorithm on 3 different platforms

APPLICATIONS

A) Telephone Network:In a telephone network the lines have bandwidth, BW. We want to route the phone call via the highest BW.

B) Flight Agenda:The agent wants to determine the earliest arrival time for the destination given an origin airport and start time.

C)Designate File Server: We consider that most of time transmitting files from one computer to another computer is the connect time. So we want to minimize the number of "hops" from the file server to every other computer on the network.In this case Djikstra's algorithm will be put into the use.

D) Google Maps:It uses more complex and efficient algorithms.. But dijkstras is the basis. It's also used in finding a shortest communication path between two nodes connected in a network

E)All this path finding algorithms are used in AI (Artificial Intelligence)

F)Game Development

G) Cognitive Science

PSEUDOCODE

Dijkstra's Pseudo Code

- u := Extract_Min(Q) searches for the vertex *u* in the vertex set *Q* that has the least *d*[*u*] value.
- That vertex is removed from the set *Q* and returned to the user.

```
1  function Dijkstra(G, w, s)
2     for each vertex v in V[G]                          // Initializations
3           d[v] := infinity
4           previous[v] := undefined
5     d[s] := 0
6     S := empty set
7     Q := V[G]
8     while Q is not an empty set                        // The algorithm itself
9           u := Extract_Min(Q)
10          S := S union {u}
11          for each edge (u,v) outgoing from u
12                 if d[v] > d[u] + w(u,v)                // Relax (u,v)
13                       d[v] := d[u] + w(u,v)
14                       previous[v] := u
```

If we are only interested in a shortest path between vertices *s* and *t*, we can terminate the search at line 9 if *u* = *t*.

25

Here initially I will get the number of the vertices and edges as input. Then after getting the input of all the edges along with their respective weight , i'll call the djikstra function.

Now in DJIKSTRA's function :

1)- I have one array named VIS , which if set true means the vertex with current index has been visited and vice-versa .

2)-Then I have an array named DIST , which will give the shortest path of a node From the chosen node at the index equals to the respective node's number.

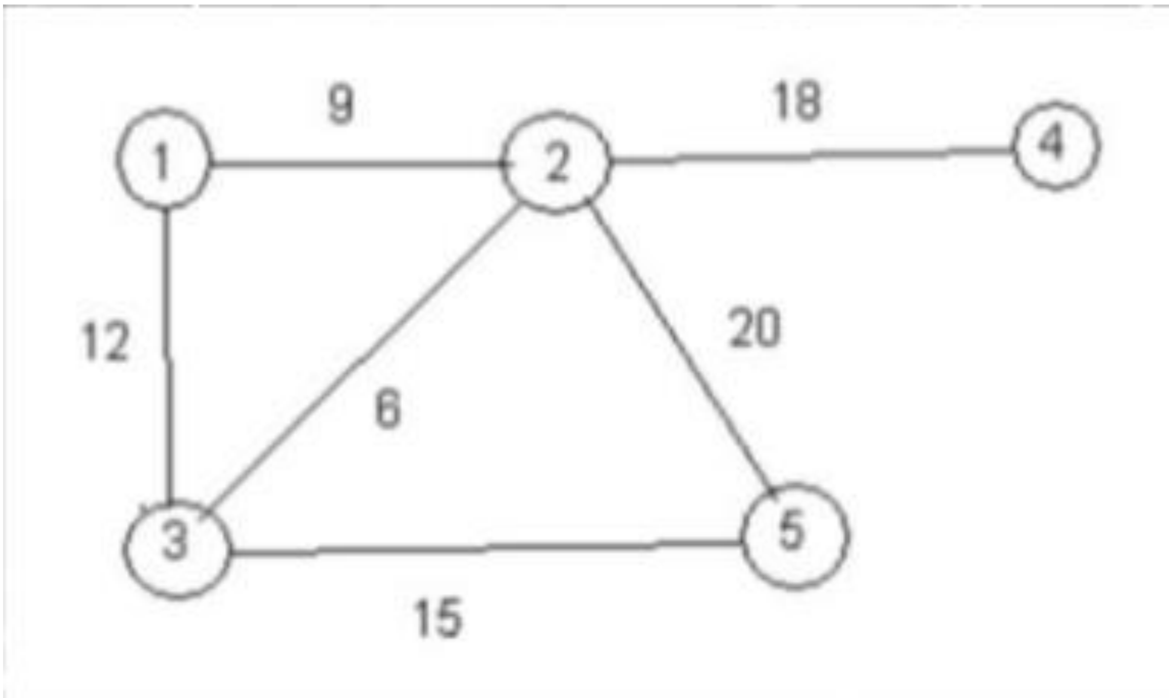
3)-Then I have used minimum priority queue which will pop-out the neighbour edge having the minimum weight.

4)-Initially we will push the vertex in the queue from which we want the shortest distance of all the nodes.

5)-Then on every iteration we will push the unvisited neighbours of the node on the top of the priority queue with side by side updation of distances stored in the DIST.

6)- Once the queue is empty we will stop the loop and will print the DIST array with the node number as the respective index.

INPUT



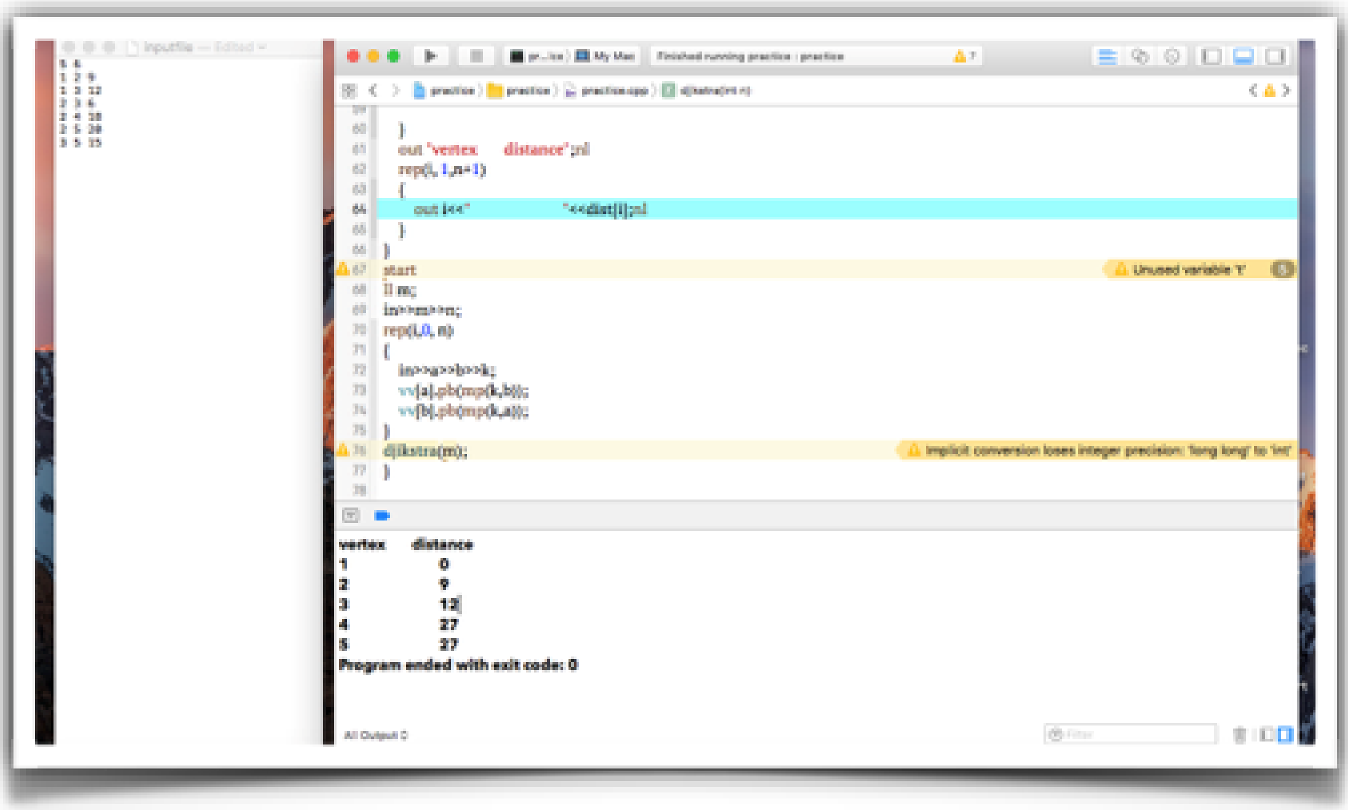
IMPLEMENTATION IN CPP

```
#include<bits/stdc++.h>
#include <fstream>
#define all(x) x.begin(),x.end()
#define rall(x) x.rbegin(),x.rend()
#define FILL(a,b) memset((a),(b),sizeof((a)))
#define countr(v,a) (int)count(v.begin(),v.end(),a)
#define err(v) v.erase(v.begin(),v.end());
#define fast
ios_base::sync_with_stdio(false),cin.tie(0),cout.tie(0);
```

```
#define ll long long
#define long_vec vector<ll>
#define nl cout<<endl;
#define out cout<<
#define print(v) repl(0,v.size()){out v[i]<<" ";}
#define rep(i,a,n) for(int i=a;i<n;i++)
#define repl(a,b) for(ll i=a;i<b;i++)
#define ret0 return 0;
#define sortv(v) sort(v.begin(),v.end())
#define start int main(){fast str s;int inp;ll
n,inpl,a,b,t,q=0,k;long_vec v;char c;ifstream in
("/Users/kashishmiglani/Desktop/iCloud Drive (Archive) -
1/Desktop/practice/inputfile") ;//ofstream
Output_in_file("/Users/kashishmiglani/Desktop/op1.txt");
#define str string
#define pb push_back
#define pll pair<ll,ll>
#define vec vector<int>
#define mp(a,b) make_pair(a,b)
#define vecp vector<pair<ll,ll>>
#define fi(it,a) for(auto it=a.begin();it!=a.end();it++)
#define MOD 1000000007
#define MAX 100000
using namespace std;
vector<pair<ll,ll>> vv[100000];
void djikstra(int n)
{
    bool vis[n+1];
    ll dist[n+1];
    rep(i, 0,n+1)
    {
        vis[i]=false;dist[i]=INT_MAX;
    }
    priority_queue<pll,vector<pll>,greater<pll>> q;
    q.push(mp(0,1));
    dist[1]=0;
    while(!q.empty())
    {
        ll w1=q.top().first;
        ll e1=q.top().second;
        q.pop();
        if(vis[e1])
            continue;
        vis[e1]=true;
        rep(i, 0,vv[e1].size())
        {
            if(dist[vv[e1][i].second] > dist[e1] + vv[e1][i].first )
            {
                dist[vv[e1][i].second] =dist[e1] + vv[e1][i].first;
                if(!vis[vv[e1][i].second])
                    q.push(vv[e1][i]);
            }
        }

    }
    out "vertex       distance";nl
```

```
    rep(i, 1,n+1)
    {
        out i<<"                          "<<dist[i];nl
    }
}
start
ll m;
in>>m>>n;
rep(i,0, n)
{
    in>>a>>b>>k;
    vv[a].pb(mp(k,b));
    vv[b].pb(mp(k,a));
}
djikstra(m);
}
```

OUTPUT

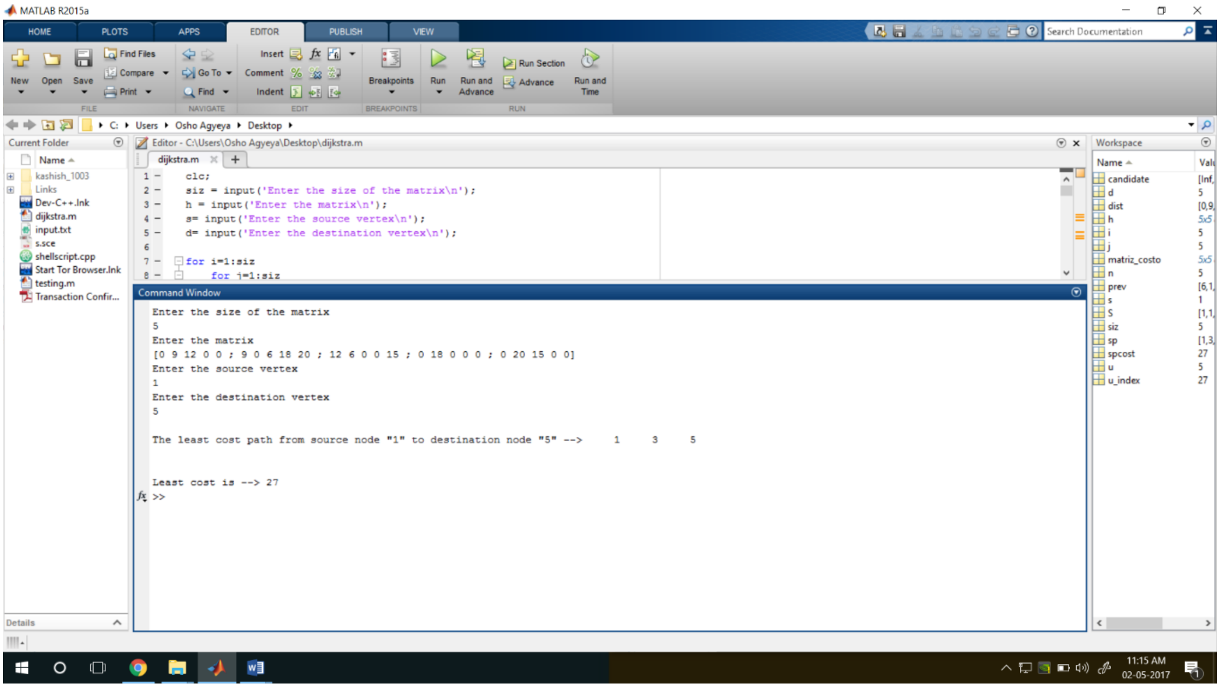


IMPLEMENTATION IN MATLAB

```
clc;
siz = input('Enter the size of the matrix\n');
h = input('Enter the matrix\n');
s= input('Enter the source vertex\n');
d= input('Enter the destination vertex\n');

for i=1:siz
    for j=1:siz
       if h(i,j)== 0
           h(i,j)=inf;
       end
    end
 end

matriz_costo=h;
%function [sp, spcost] = dijkstra(matriz_costo, s, d)

n=siz;
S(1:n) = 0;     %s, vector, set of visited vectors
dist(1:n) = inf;   % it stores the shortest distance between the source
node and any other node;
prev(1:n) = n+1;    % Previous node, informs about the best previous node
known to reach each  network node

dist(s) = 0;


while sum(S)~=n
    candidate=[];
    for i=1:n
        if S(i)==0
            candidate=[candidate dist(i)];
        else
            candidate=[candidate inf];
        end
    end
    [u_index u]=min(candidate);
    S(u)=1;
    for i=1:n
        if(dist(u)+matriz_costo(u,i))<dist(i)
            dist(i)=dist(u)+matriz_costo(u,i);
            prev(i)=u;
        end
    end
end


sp = [d];

while sp(1) ~= s
    if prev(sp(1))<=n
```

```
            sp=[prev(sp(1)) sp];
        else
            error;
        end
end
spcost = dist(d);
fprintf('\nThe least cost path from source node "%d" to destination node
"%d" -->',s,d);
disp(sp);
fprintf('\nLeast cost is --> %g',spcost);
fprintf('\n');
```

OUTPUT

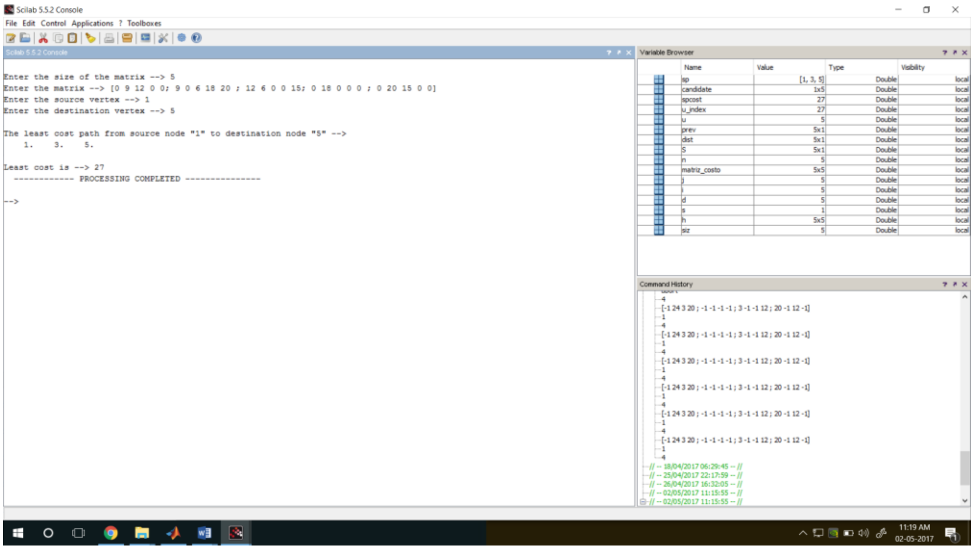


IMPLEMENTATION IN SCILAB

```
// Display mode
mode(0);

// Display warning for floating point exception
ieee(1);

clc;
siz = input("Enter the size of the matrix --> ");
h = input("Enter the matrix --> ");
s = input("Enter the source vertex --> ");
d = input("Enter the destination vertex --> ");
```

```
for i =1:siz
  for j =1:siz
    if h(i,j) ==(0) then
      h(i,j) = %inf;
    end;
  end;
end;

matriz_costo = h;
//function [sp, spcost] = dijkstra(matriz_costo, s, d)

n = siz;
S(1:n) = 0;//s, vector, set of visited vectors
dist(1:n) = %inf;// it stores the shortest distance between the source node and any other node;
prev(1:n) = (n+1);// Previous node, informs about the best previous node known to reach each  network node

dist(s) = (0);

while (sum(S) ~= (n))
  candidate = [];
  for i =1:n
    if S(i)==0 then
      candidate = [candidate,dist(i)];
    else
      candidate = [candidate,%inf];
    end;
  end;
  [u_index,u] = min(candidate);
  S(u) = 1;
  for i = 1:n
    if (dist(u)+(matriz_costo(u,i))) < dist(i) then
      dist(i) = dist(u)+(matriz_costo(u,i));
      prev(i) = u;
    end;
  end;
end;
sp = [d];
while (sp(1) ~= s )
  if (prev(sp(1)))<=n then
    sp = [prev(sp(1)) sp];
  else
    error;
  end;
end;
spcost = dist(d);
// L.56: No simple equivalent, so mtlb_fprintf() is called.
printf("\nThe least cost path from source node ""%d"" to destination node ""%d"" -->",s,d);
disp(sp);
// L.58: No simple equivalent, so mtlb_fprintf() is called.
printf("\nLeast cost is --> %g",spcost);
// L.59: No simple equivalent, so mtlb_fprintf() is called.
disp(" ------------ PROCESSING COMPLETED ---------------");
```

OUTPUT


Scilab 5.5.2 Console
File Edit Control Applications ? Toolboxes
Scilab 5.5.2 Console
Enter the size of the matrix --> 5
Enter the matrix --> [0 9 12 0 0; 9 0 6 18 20 ; 12 6 0 0 15; 0 18 0 0 0 ; 0 20 15 0 0]
Enter the source vertex --> 1
Enter the destination vertex --> 5
The least cost path from source node "1" to destination node "5" -->
1. 3. 5.
Least cost is --> 27
------------ PROCESSING COMPLETED ---------------
-->
Variable Browser
Command History

FLOYD–WARSHALL ALGORITHM (VINEET KISHORE)

In computer science, the Floyd–Warshall algorithm is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm. Versions of the algorithm can also be used for finding the transitive closure of a relation R, or (in connection with the Schulze voting system) widest paths between all pairs of vertices in a weighted graph.

ALGORITHM

The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with $O(|V|)^3$ comparisons in a graph. This is remarkable considering that there may be up to $\Omega(|V|^2)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

Consider a graph G with vertices V numbered 1 through N. Further consider a function shortestPath(i,j,k) that returns the shortest possible path from i to j using vertices only from the set {1,2,…,k} as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i to each j using only vertices in {1,2,…,k+1}.

For each of these pairs of vertices, the true shortest path could be either

(1) a path that only uses vertices in the set {1,2,…,k+1}.
or
(2) a path that goes from i to k+1 and then from k+1 to j.

We know that the best path from i to j that only uses

vertices 1 through k is defined by , and it is clear that if there were a better path from i to k+1 to j , then the length of this path would be the concatenation of the shortest path from i to k+1 (using vertices in {1,…,k}) and the shortest path from {k+1} to j (also using vertices in {1,…,k}).

If w(i,j) is the weight of the edge between vertices i and j, we can define shortestPath(I,j,k+1) in terms of the following recursive formula: the base case is

shortestPath(i,j,k+1) = w(i,j)

and the recursive case is

shortestPath(i,j,k+1) = min(shortestPath(i,j,k+1) , shortest(i,k+1,k) + shortestPath(k+1,j,k))

This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing shortestPath(i,j,k) for all (i,j) pairs for k=1, then k=2, etc. This process continues until k=N, and we have found the shortest path for all (i,j) pairs using any intermediate vertices.

PSEUDOCODE

```
let dist be a |V| × |V| array of minimum distances
initialized to ∞ (infinity)
for each vertex v
   dist[v][v] ← 0
for each edge (u,v)
   dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
for k from 1 to |V|
   for i from 1 to |V|
      for j from 1 to |V|
         if dist[i][j] > dist[i][k] + dist[k][j]
             dist[i][j] ← dist[i][k] + dist[k][j]
         end if
```

Path reconstruction

The Floyd–Warshall algorithm typically only provides the lengths of the paths between all pairs of vertices. With simple modifications, it is possible to create a method to reconstruct the actual path between any two endpoint vertices. While one may be inclined to store the actual path from each vertex to each other vertex, this is not necessary, and in fact, is very costly in terms of memory. Instead, the shortest-path tree can be calculated for each node in O(|E|) time using O(|V|) memory to store each tree which allows us to efficiently reconstruct a path from any two connected vertices.

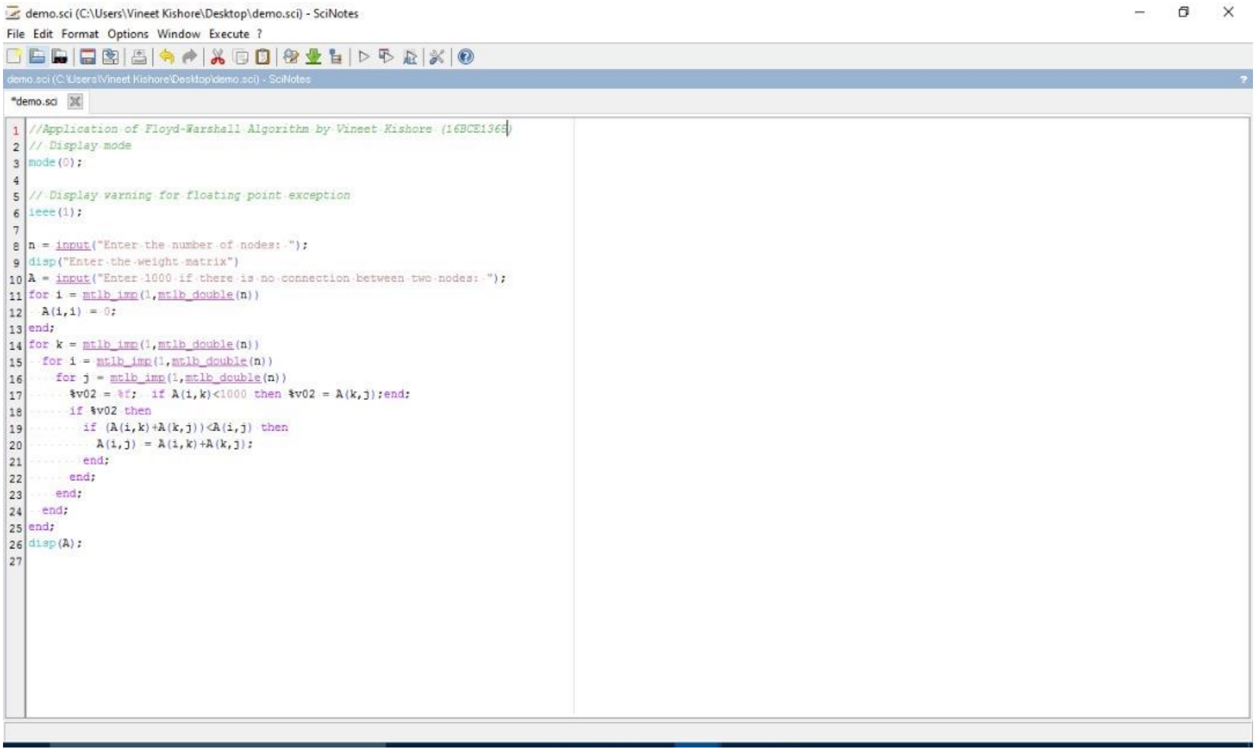
```
let dist be a |V|X|V| array of minimum distances
initialized to
(infinity)
let next be a |V|X|V| array of vertex indices initialized
to null

procedure FloydWarshallWithPathReconstruction ()
   for each edge (u,v)
      dist[u][v] ← w(u,v)  // the weight of the edge (u,v)
```

```
      next[u][v] ← v
   for k from 1 to |V| // standard Floyd-Warshall
implementation
      for i from 1 to |V|
         for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j] then
               dist[i][j] ← dist[i][k] + dist[k][j]
               next[i][j] ← next[i][k]

procedure Path(u, v)
   if next[u][v] = null then
       return []
   path = [u]
   while u ≠ v
       u ← next[u][v]
       path.append(u)
   return path
```

ANALYSIS

Let n be |V|, the number of vertices. To find all n^2 of shortestPath(i,j,k) (for all i and j) from those of shortestPath(i,j,k-1) requires $2n^2$ operations. Since we begin with shortestPath(i,j,0) and compute the sequence of n matrices shortestPath(i,j,1) , shortestPath(i,j,2), …, shortestPath(i,j,n), the total number of operations used is $n.2n^2$. Therefore, the complexity of the algorithm is $O(n^3)$.

IMPLEMENTATION IN SCILAB



OUTPUT