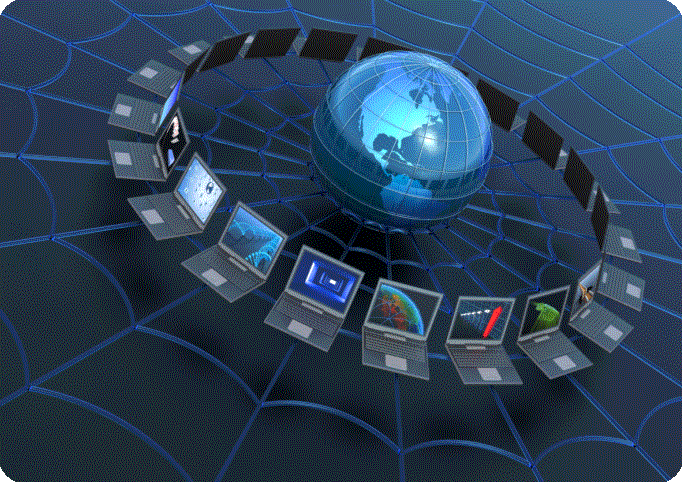
NETWORKS & COMMUNICATION (CSE 1004) PROJECT

ASHA S.

LAB 04/02/17



Kashish Miglani-15BCE1003

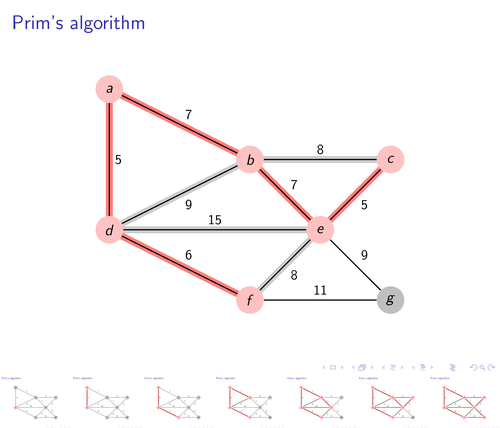
Vineet Kishore-16BCE1365

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.

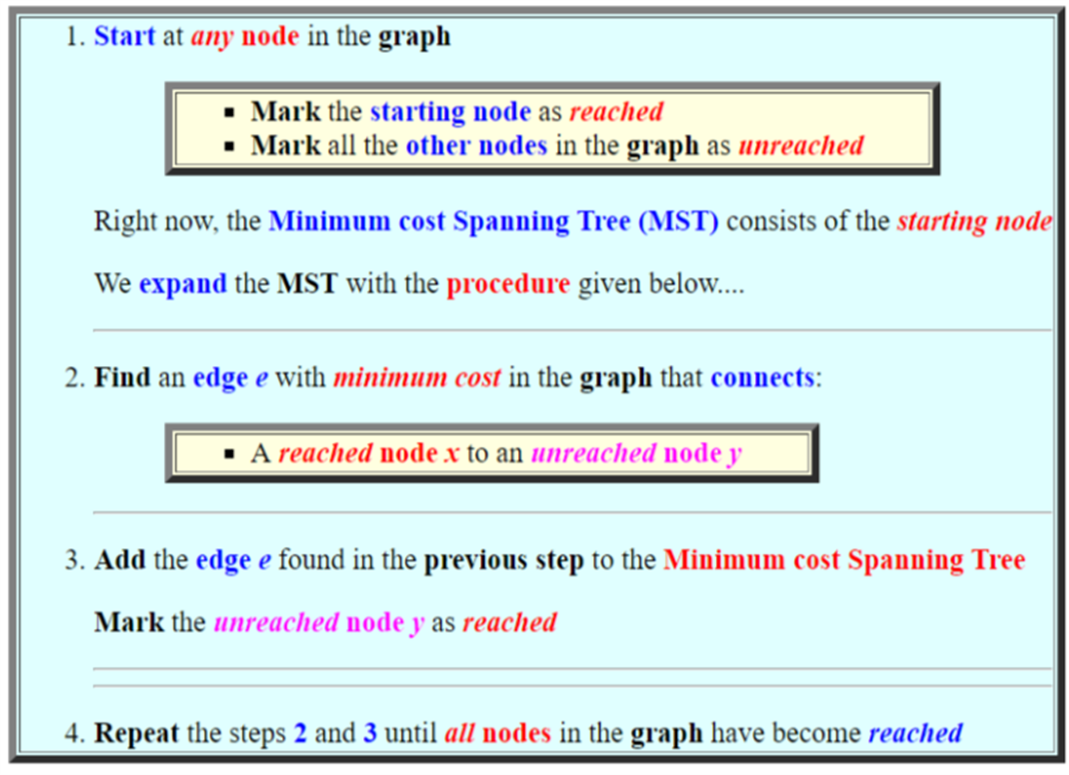
3. Prim Algorithm Approach to Improving Local Access Network in Rural Areas

4. AI (Artificial Intelligence)

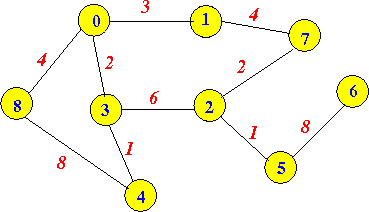
5. Game Development

6. Cognitive Science

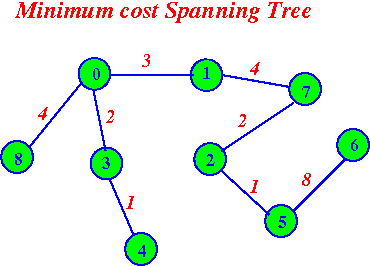
ALGORITHM



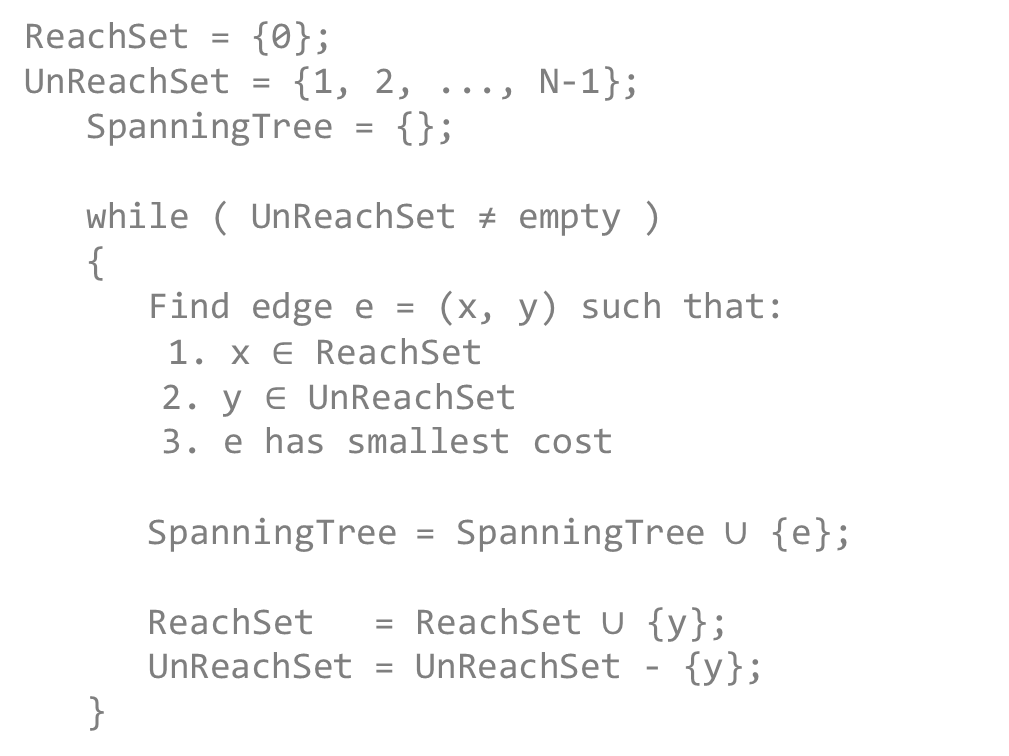
INPUT



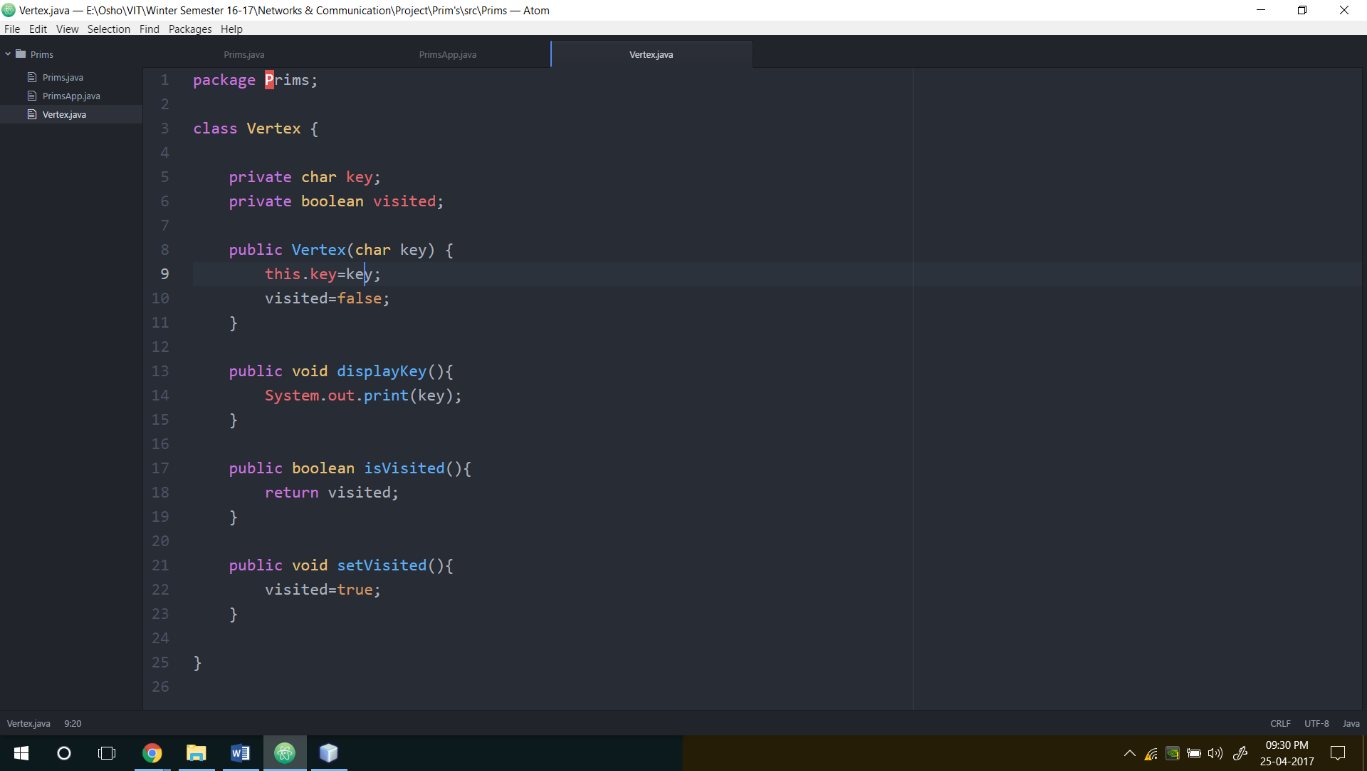
OUTPUT

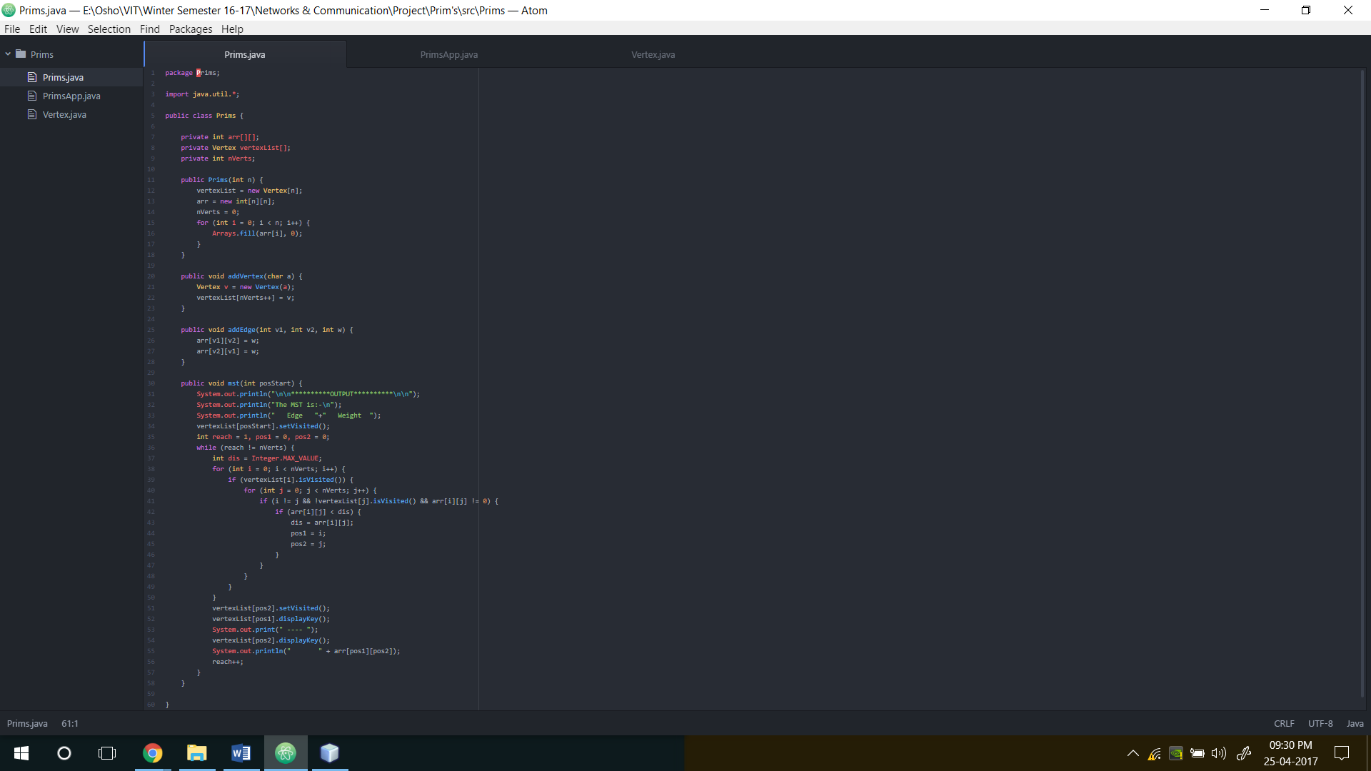


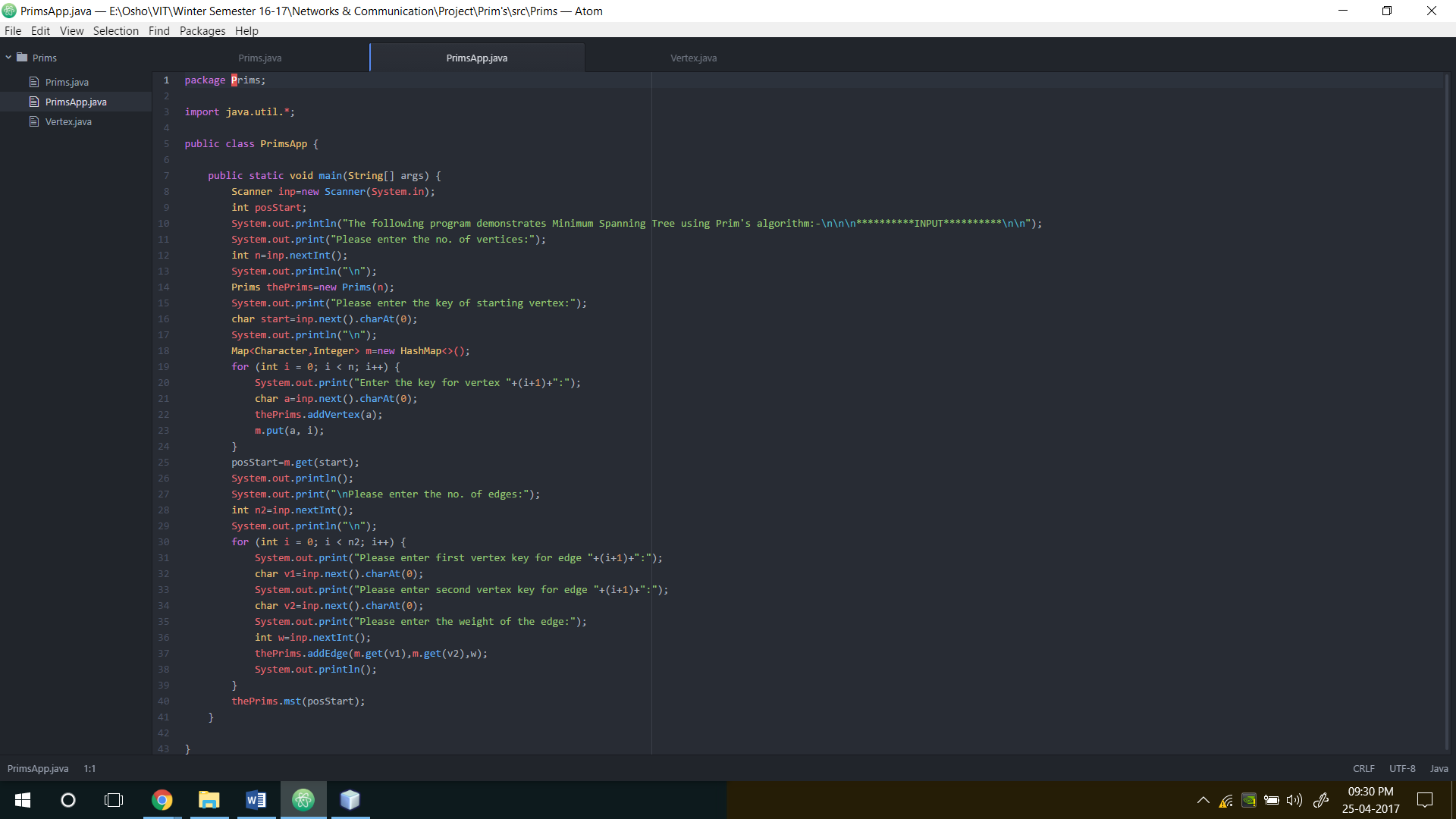
PSEUDO CODE



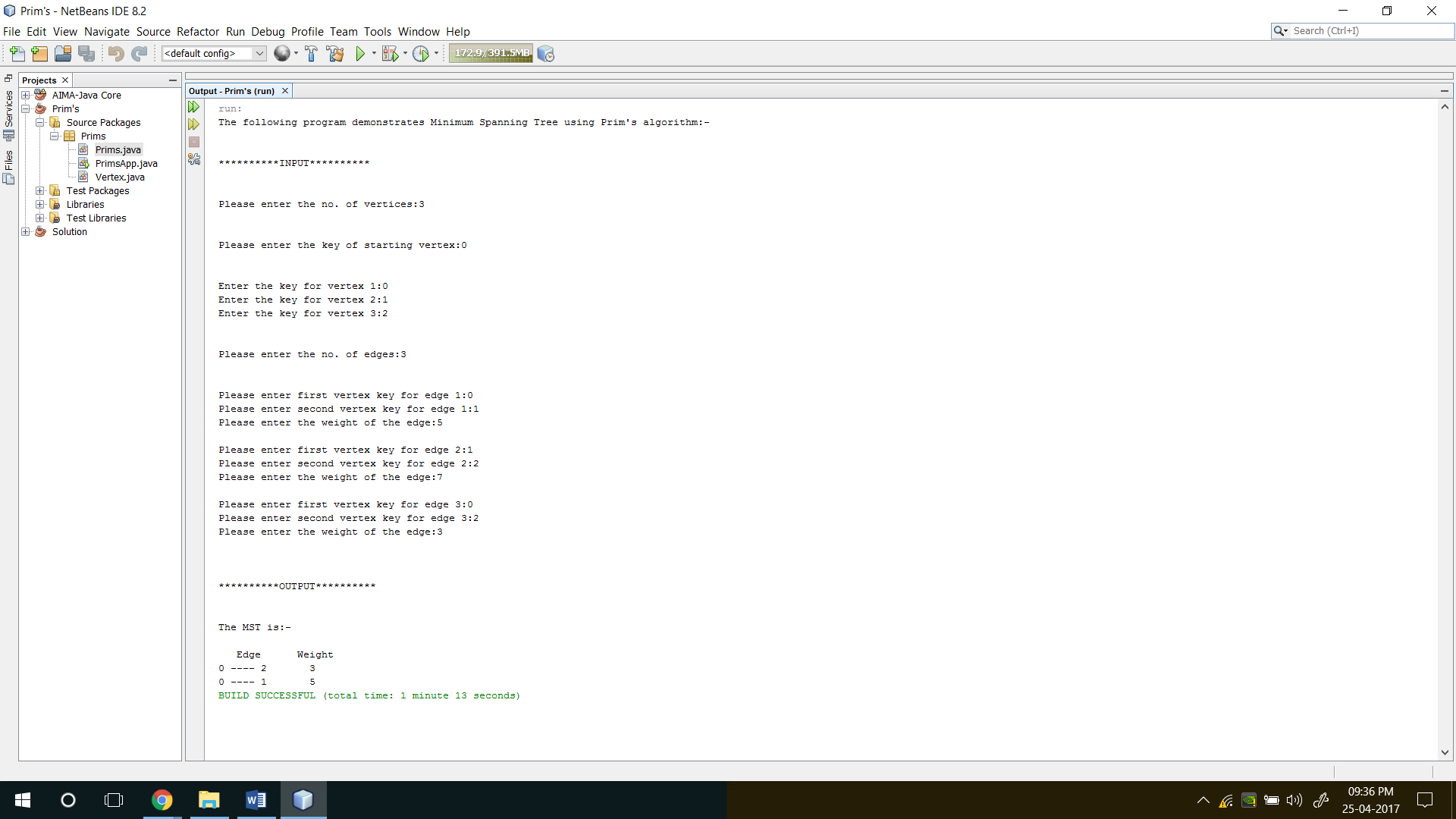
IMPLEMENTATION IN JAVA



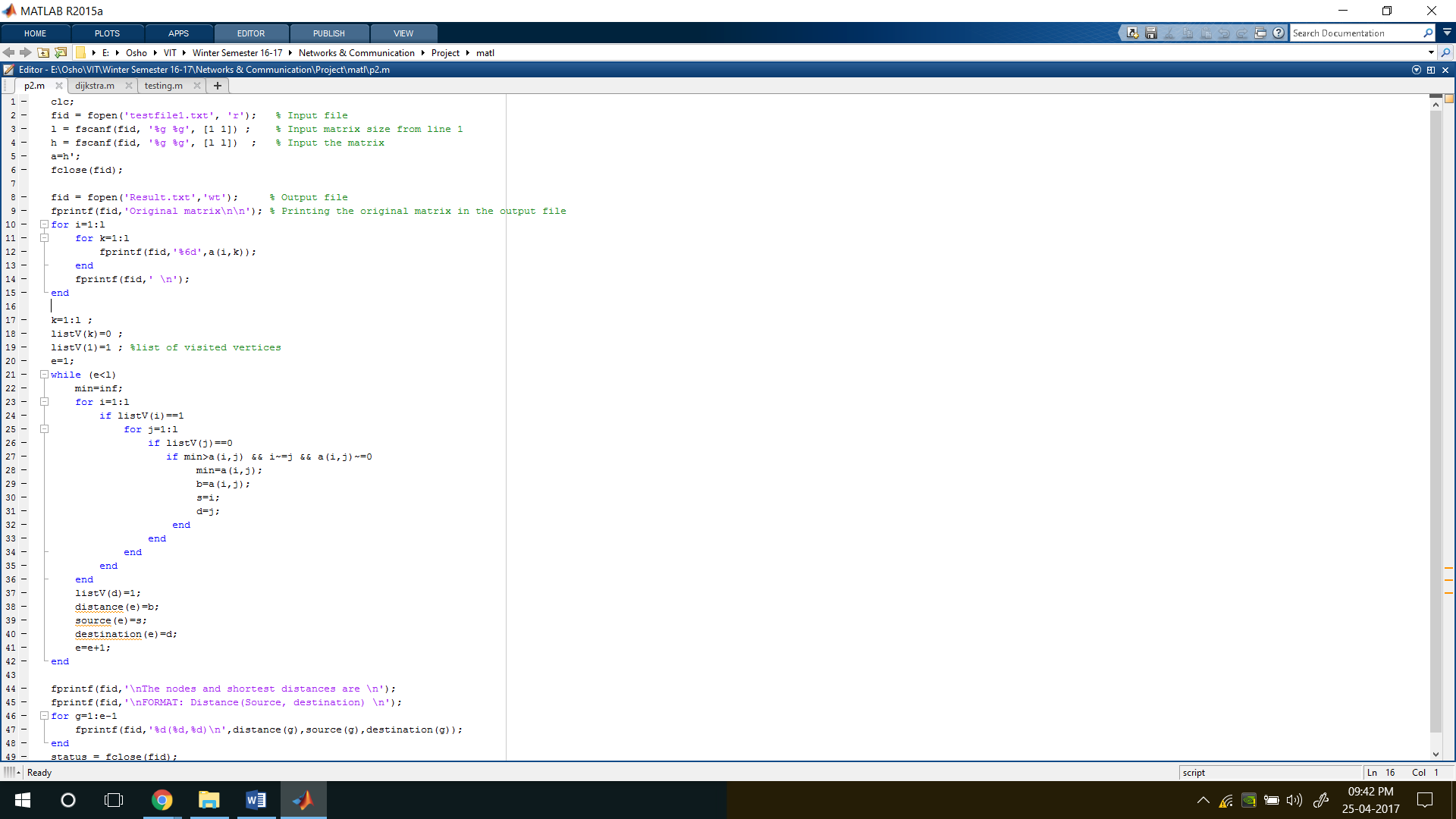




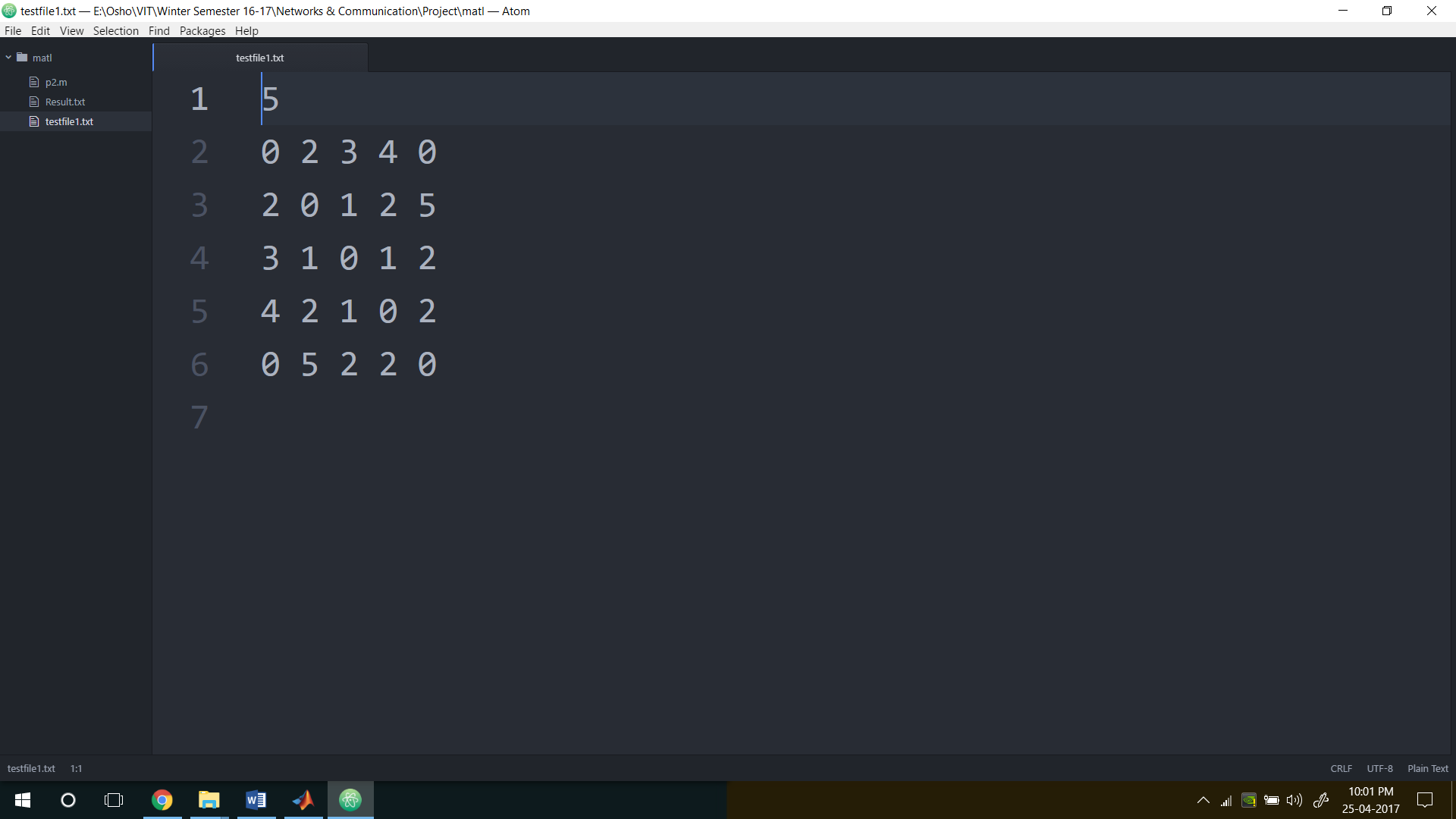
OUTPUT



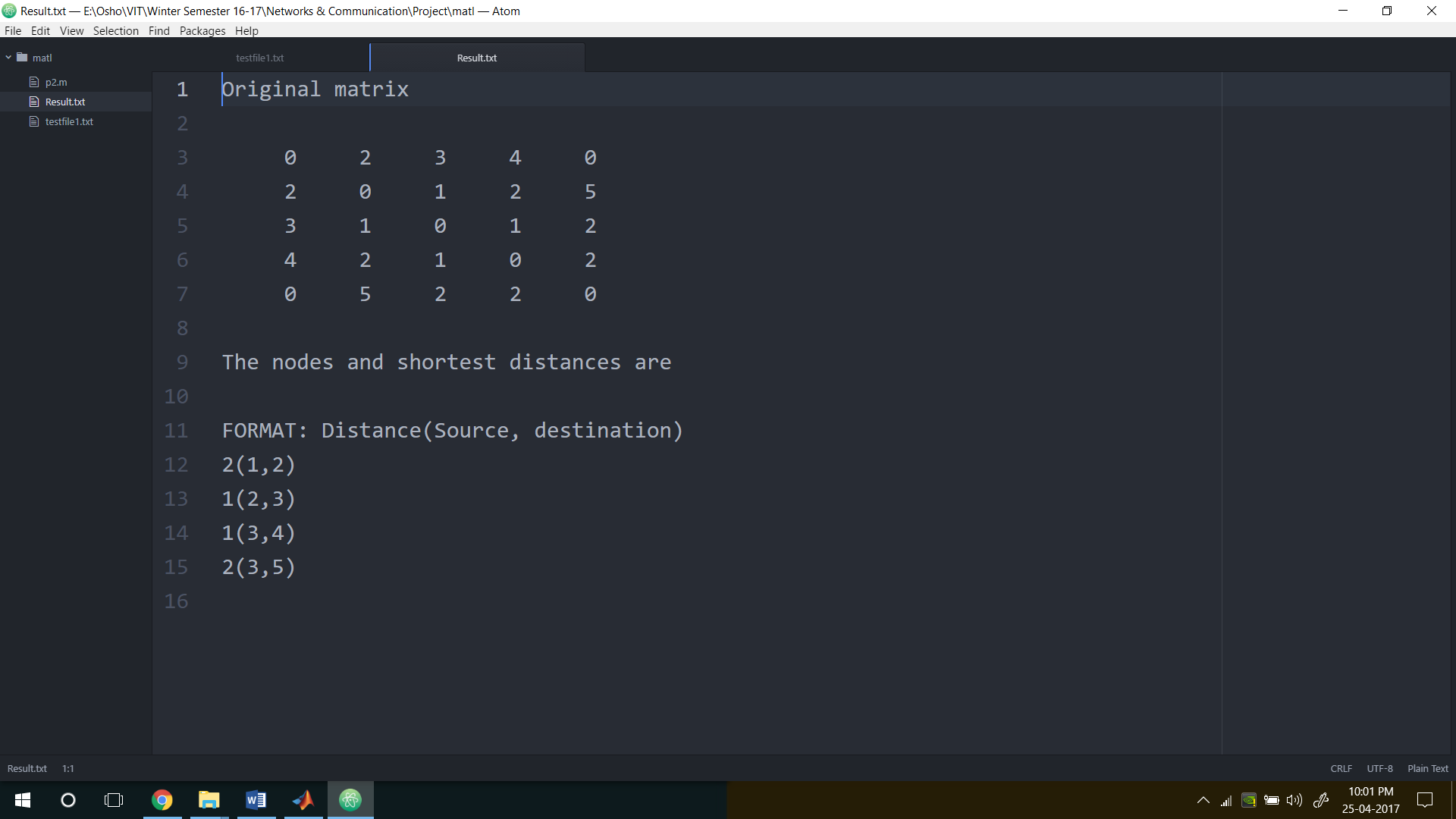
IMPLEMENTATION IN MATLAB



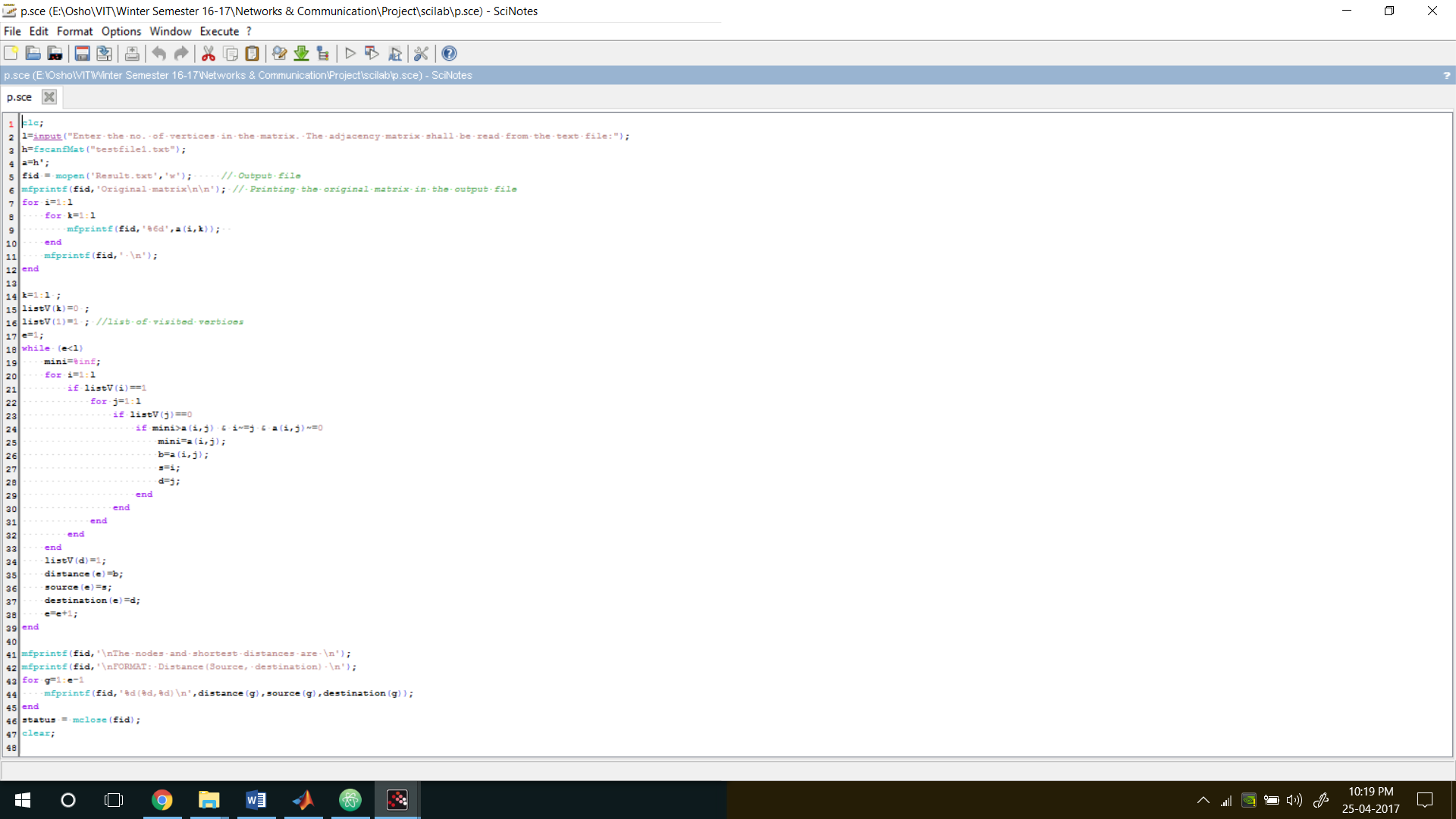
INPUT



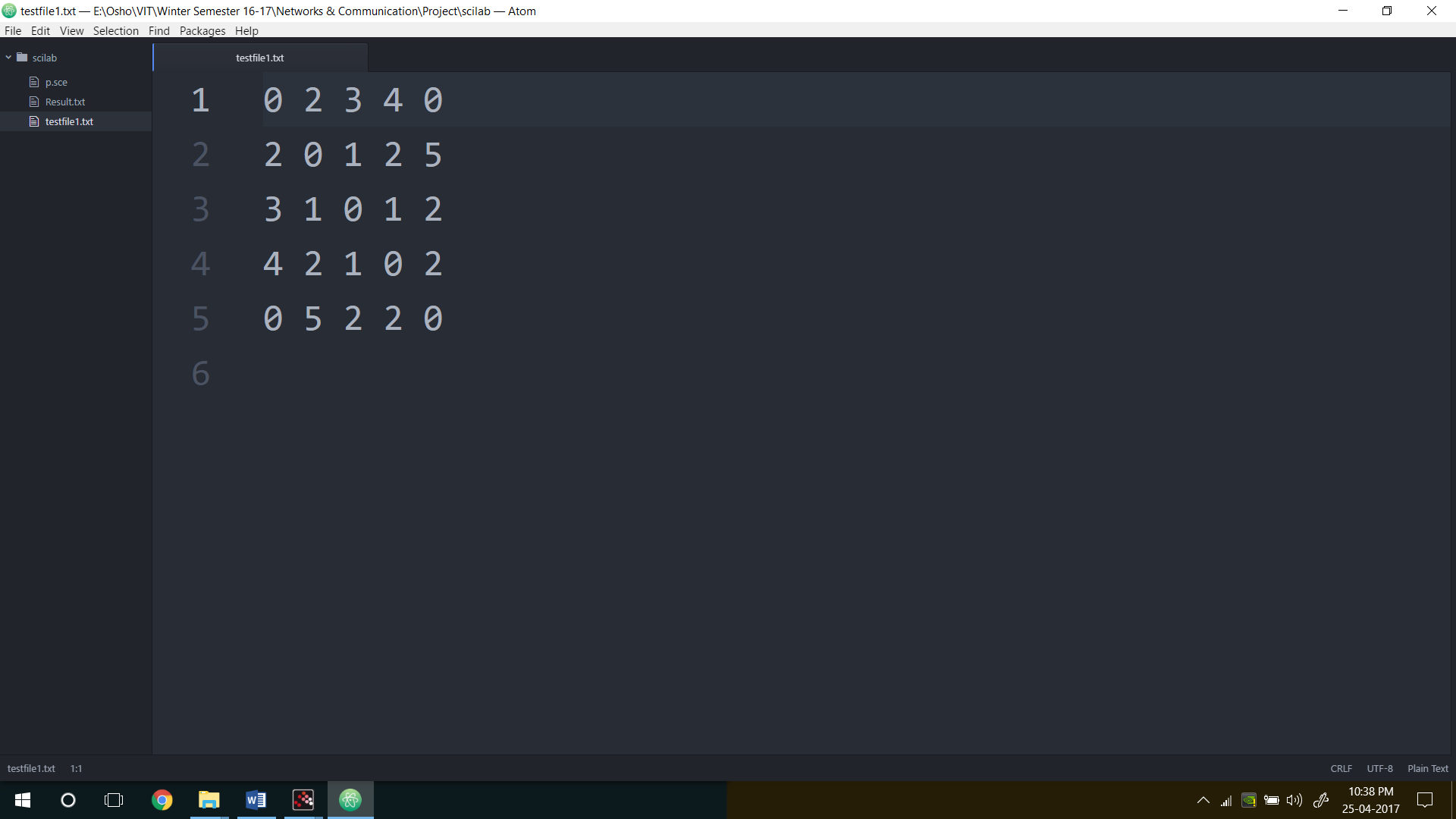
OUTPUT



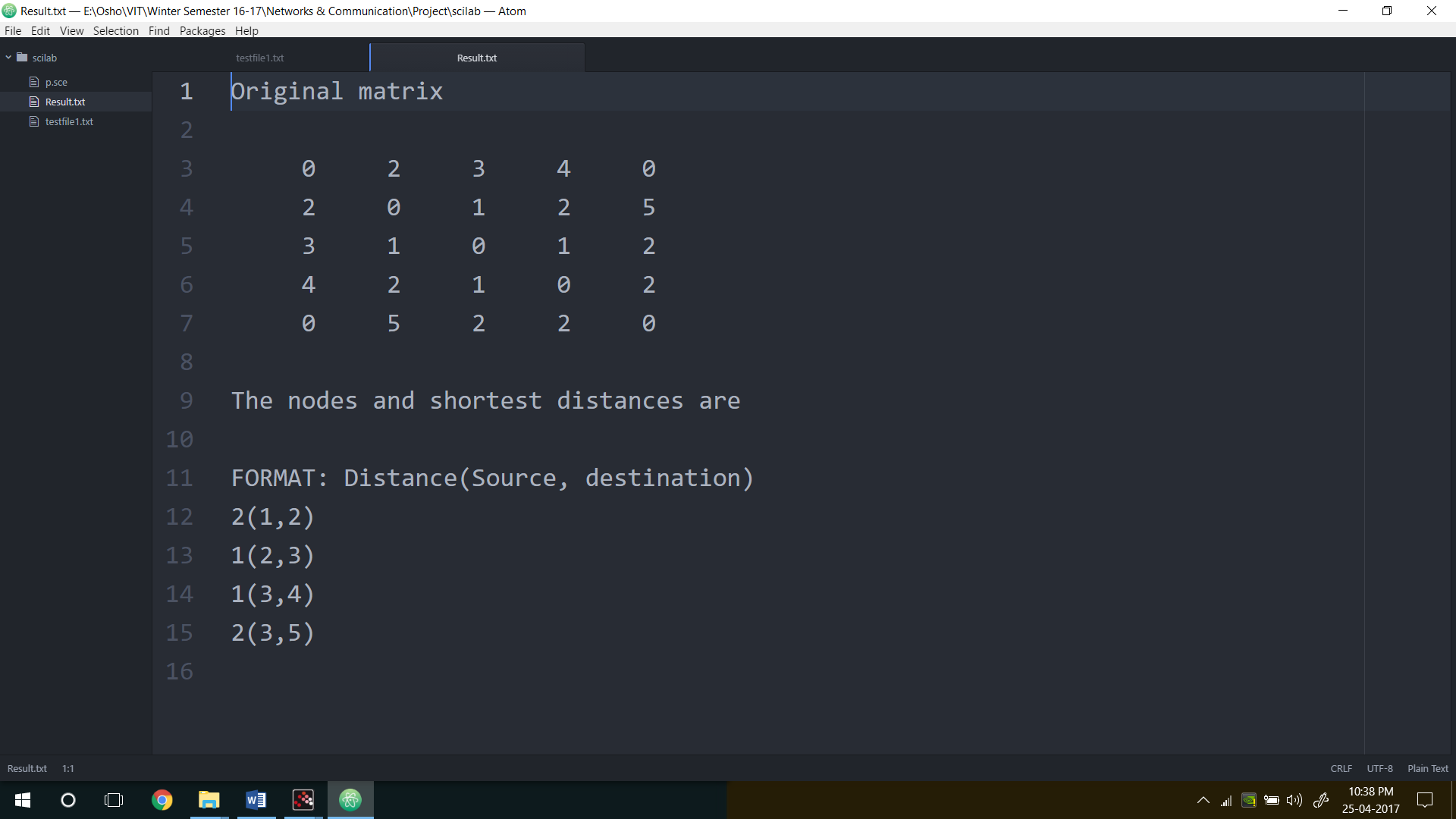
IMPLEMENTATION IN SCILAB



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, ф) 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 = ф 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.

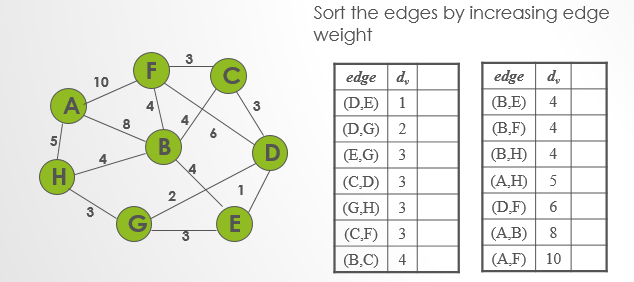
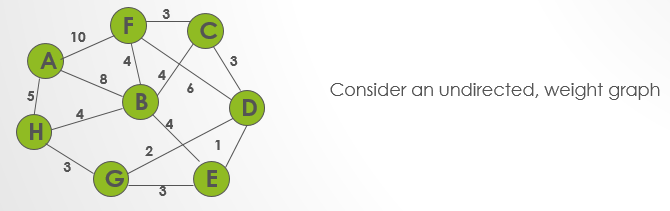
MST is fundamental problem with diverse applications

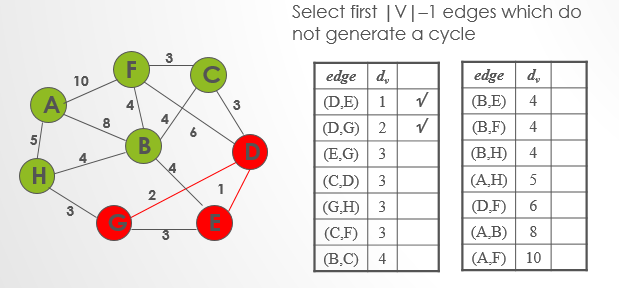
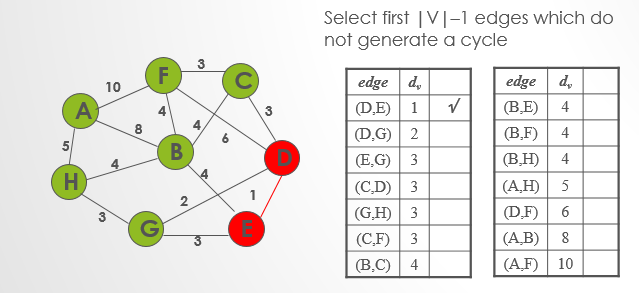
Network design.  
– telephone, electrical, hydraulic, TV cable, computer, road  
The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts 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.

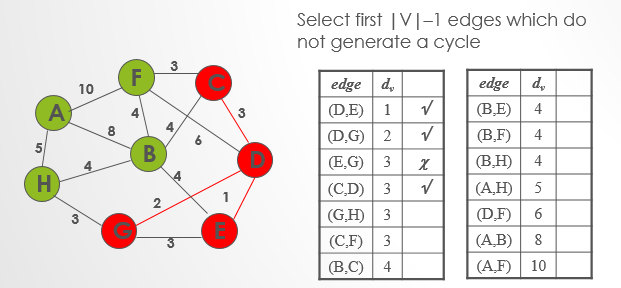
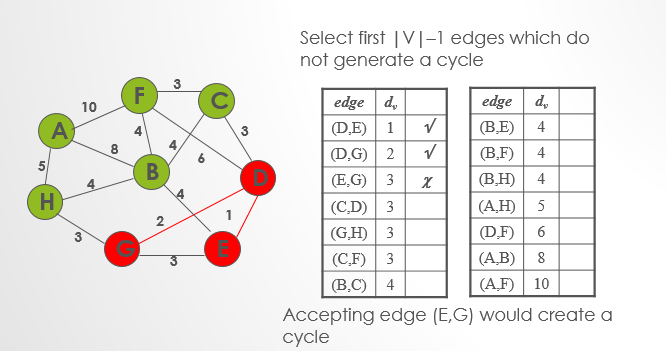
Approximation algorithms for NP-hard problems.  
– traveling salesperson problem, Steiner tree  
A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path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

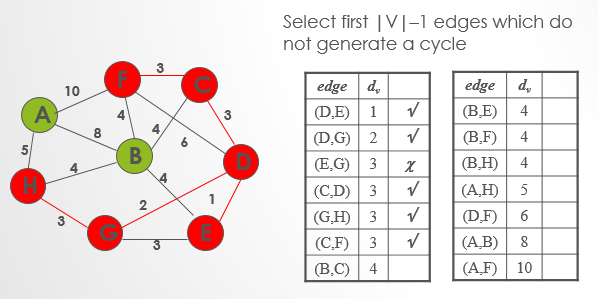
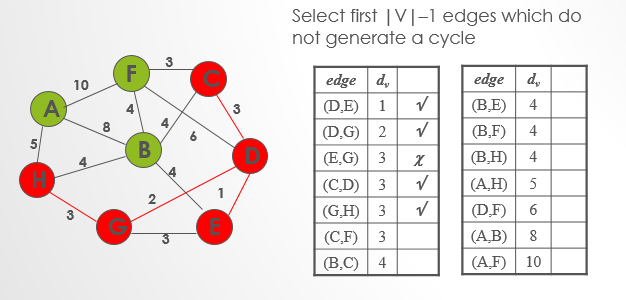
Cluster analysis  
k clustering problem can be viewed as finding an MST and deleting the k-1 most  
expensive edges.

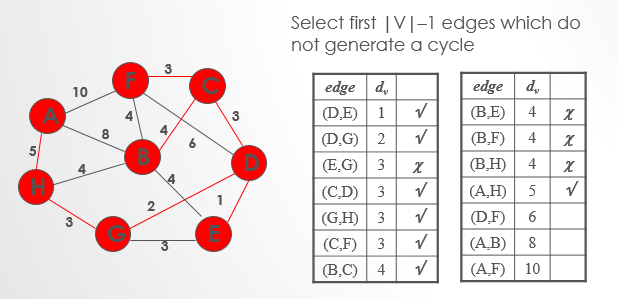
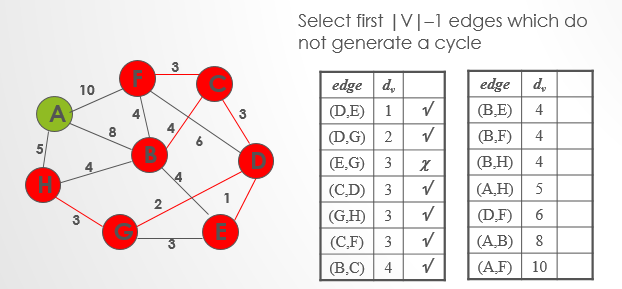
WALK-THROUGH

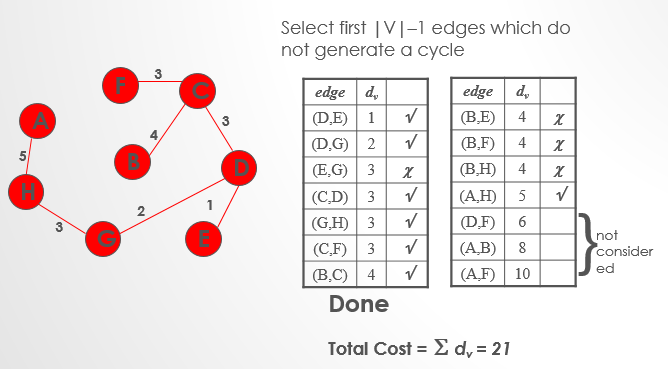












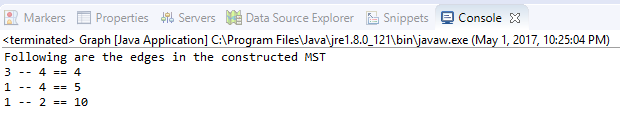
TIME COMPLEXITY

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 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();  
 }  
  
 *// 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



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)

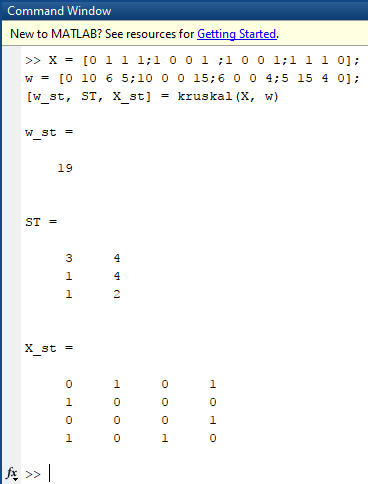
rnk(r\_j) = rnk(r\_j) + 1;

end

end

end

OUTPUT



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

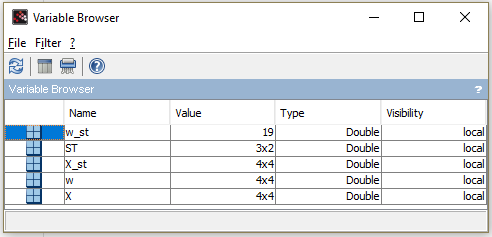
endfunction

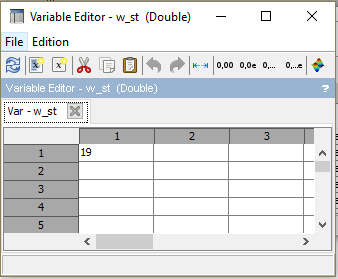
X = [0 1 1 1;1 0 0 1 ;1 0 0 1;1 1 1 0];

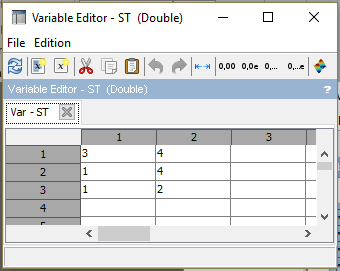
w = [0 10 6 5;10 0 0 15;6 0 0 4;5 15 4 0];

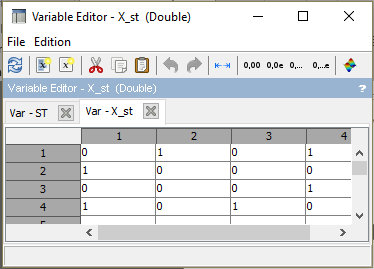
[w\_st, ST, X\_st] = kruskal(X, w);

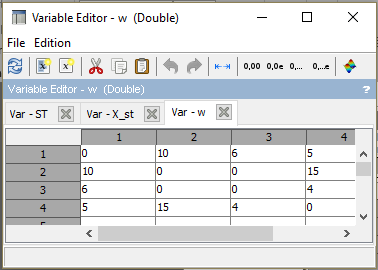
OUTPUT

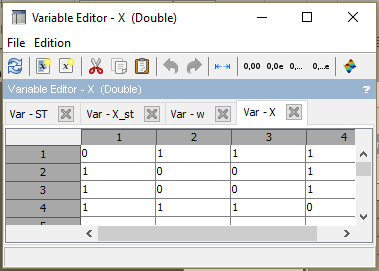












BELLMAN-FORD ALGORITHM (SIDDHARTH CHANDRA)

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:

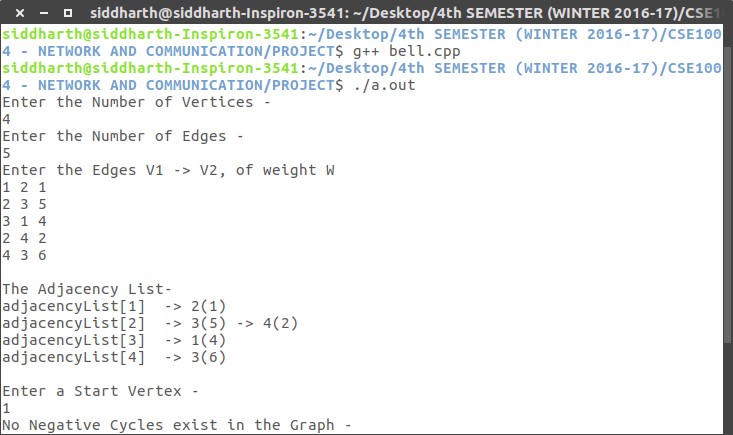
relax(u, v)

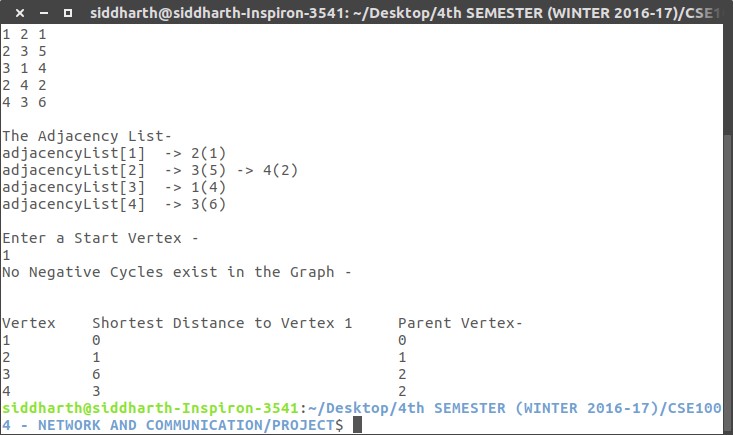
for (u, v) in E:

if v.distance > u.distance + weight(u, v):

print "A negative weight cycle exists"

OUTPUT IN C++





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:' );

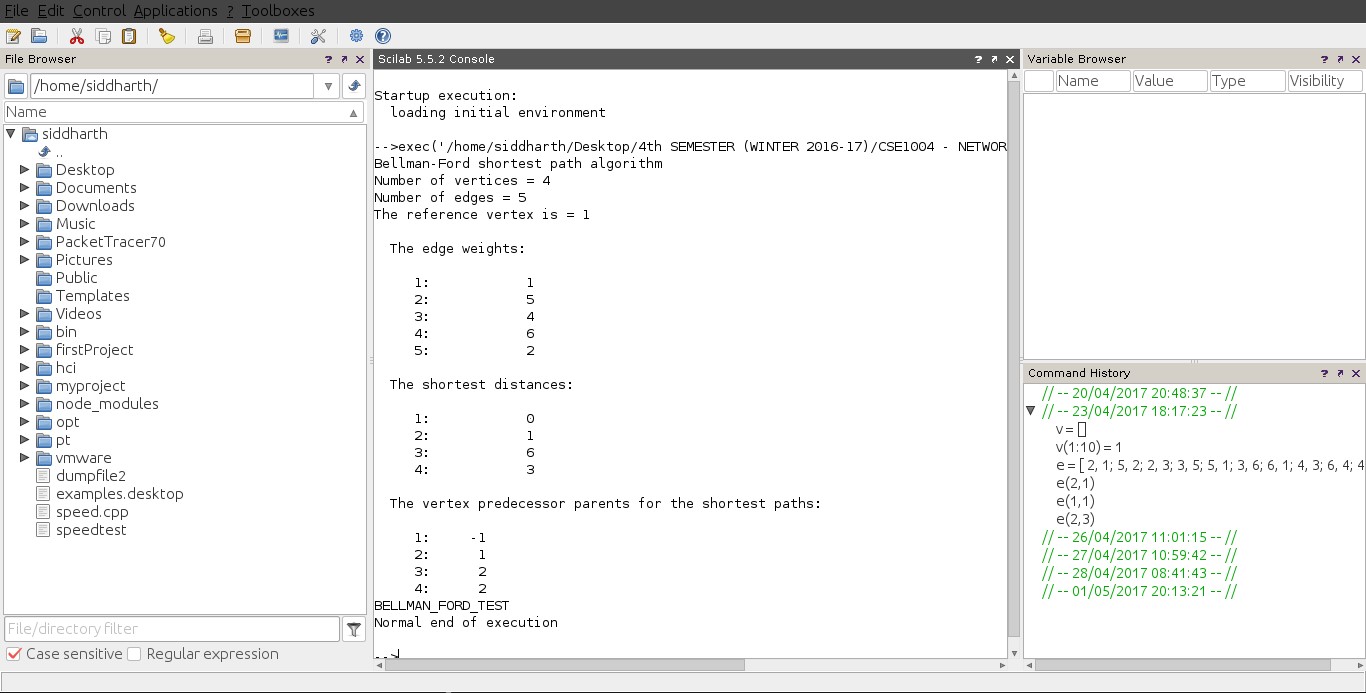
return endfunction function bellman\_ford\_test ( )

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.

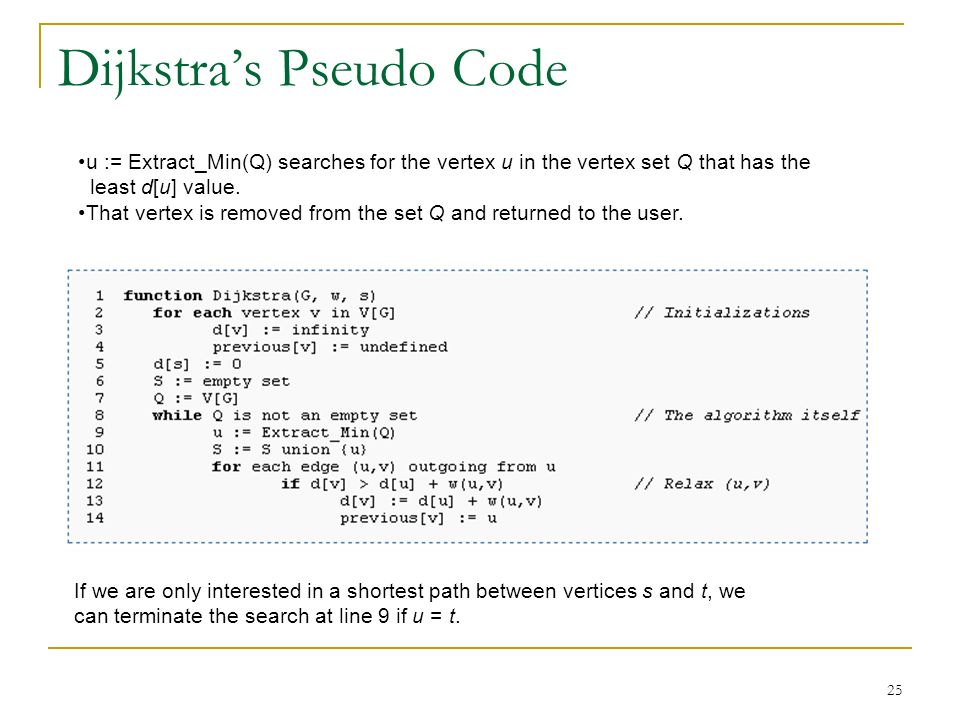
D) Google Maps:It uses more complex and efficient algorithms.. But dijkstras is the basis. It's also used in finding a shortest communication path between two nodes connected in a network

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

F)Game Development

G) Cognitive Science

PSEUDOCODE



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.

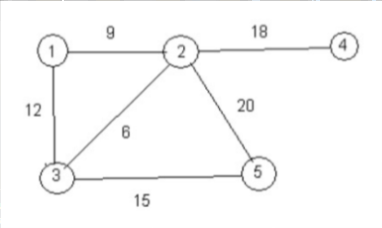
3)-Then I have used minimum priority queue which will pop-out the neighbour edge having the minimum weight.

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));

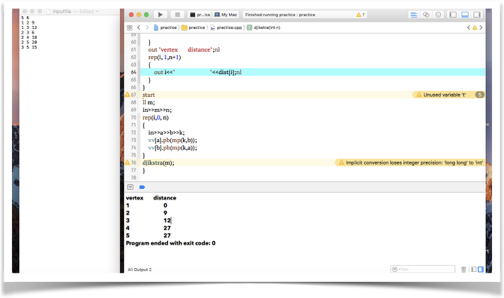
vv[b].pb(mp(k,a));

}

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);

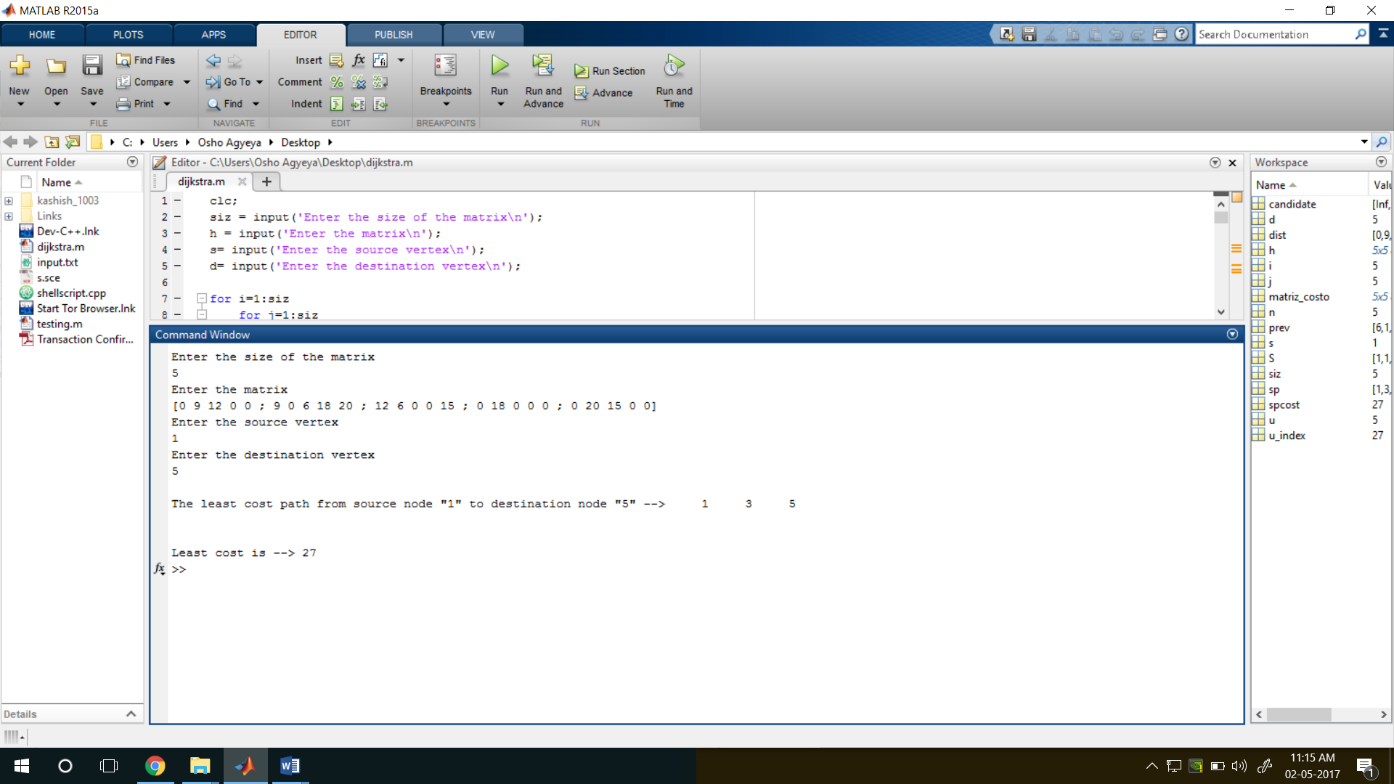
fprintf('\nThe least cost path from source node "%d" to destination node "%d" -->',s,d);

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);

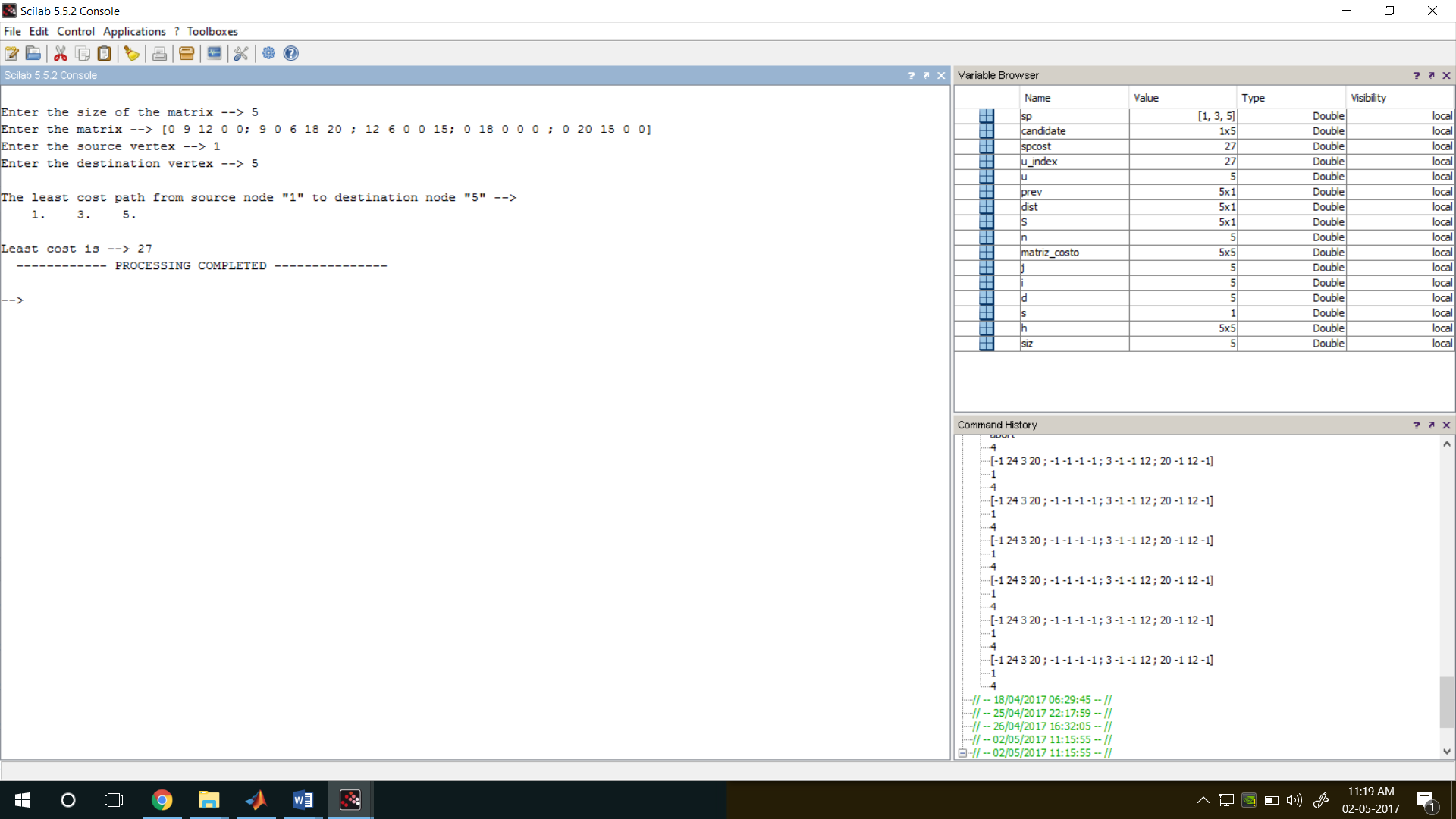
*// L.58: No simple equivalent, so mtlb\_fprintf() is called.*

printf("\nLeast cost is --> %g",spcost);

*// L.59: No simple equivalent, so mtlb\_fprintf() is called.*

disp(" ------------ PROCESSING COMPLETED ---------------");

OUTPUT



FLOYD–WARSHALL ALGORITHM

(VINEET KISHORE)

In [computer science](https://en.wikipedia.org/wiki/Computer_science), the Floyd–Warshall algorithm is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for finding [shortest paths](https://en.wikipedia.org/wiki/Shortest_path_problem) in a [weighted graph](https://en.wikipedia.org/wiki/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](https://en.wikipedia.org/wiki/Transitive_closure) of a relation R, or (in connection with the [Schulze voting system](https://en.wikipedia.org/wiki/Schulze_method)) [widest paths](https://en.wikipedia.org/wiki/Widest_path_problem) 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 Θ ( | V | 3 ) {\displaystyle \Theta (|V|^{3})} O(|V|)3 comparisons in a graph. This is remarkable considering that there may be up to Ω ( | V | 2 ) {\displaystyle \Omega (|V|^{2})} Ω(|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 {\displaystyle G} G with vertices V {\displaystyle V} V numbered 1 through  N {\displaystyle N} N. Further consider a function shortestPath(i,j,k) s h o r t e s t P a t h ( i , j , k ) {\displaystyle \mathrm {shortestPath} (i,j,k)} shortestthat returns the shortest possible path from i i {\displaystyle i} iito j j {\displaystyle j} using vertices only from the set {1,2,…,k} { 1 , 2 , … , k } {\displaystyle \{1,2,\ldots ,k\}} {1,2,….as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i i {\displaystyle i} to each j j {\displaystyle j} jusing only vertices in {1,2,…,k+1}.{ 1 , 2 , … , k + 1 } {\displaystyle \{1,2,\ldots ,k+1\}} {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}.{ 1 , … , k } {\displaystyle \{1,\ldots ,k\}} {1ds

or

(2) a path that goes from i i {\displaystyle i} ito k+1 k + 1 {\displaystyle k+1} and then from k+1 k + 1 {\displaystyle k+1} to jj {\displaystyle j}.

We know that the best path from i i {\displaystyle i} to j j {\displaystyle j} that only uses vertices 1 through k k {\displaystyle k} is defined by s h o r t e s t P a t h ( i , j , k ) {\displaystyle \mathrm {shortestPath} (i,j,k)} , and it is clear that if there were a better path from i i {\displaystyle i} ito k+1 k + 1 {\displaystyle k+1} to j j {\displaystyle j}, then the length of this path would be the concatenation of the shortest path from i i {\displaystyle i} to k+1 k + 1 {\displaystyle k+1} (using vertices in {1,…,k{ 1 , … , k } {\displaystyle \{1,\ldots ,k\}}}) and the shortest path from {k+1} { k + 1 } {\displaystyle \{k+1\}} to j j {\displaystyle j} (also using vertices in {1,…,k} { 1 , … , k } {\displaystyle \{1,\ldots ,k\}}).

If w(i,j) w ( i , j ) {\displaystyle w(i,j)} is the weight of the edge between vertices i i {\displaystyle i} and jj {\displaystyle j}, we can define shortestPath(I,j,k+1) s h o r t e s t P a t h ( i , j , k + 1 ) {\displaystyle \mathrm {shortestPath} (i,j,k+1)} in terms of the following [recursive](https://en.wikipedia.org/wiki/Recursion) formula: the base case is

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

s h o r t e s t P a t h ( i , j , 0 ) = w ( i , j ) {\displaystyle \mathrm {shortestPath} (i,j,0)=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))

s h o r t e s t P a t h ( i , j , k + 1 ) = {\displaystyle \mathrm {shortestPath} (i,j,k+1)=} m i n ( s h o r t e s t P a t h ( i , j , k ) , {\displaystyle \mathrm {min} {\Big (}\mathrm {shortestPath} (i,j,k),} s h o r t e s t P a t h ( i , k + 1 , k ) + s h o r t e s t P a t h ( k + 1 , j , k ) ) {\displaystyle \mathrm {shortestPath} (i,k+1,k)+\mathrm {shortestPath} (k+1,j,k){\Big )}}

This formula is the heart of the Floyd–Warshall algorithm. The algorithm works by first computing shortestPath(i,j,k) s h o r t e s t P a t h ( i , j , k ) {\displaystyle \mathrm {shortestPath} (i,j,k)} for all (i,j) ( i , j ) {\displaystyle (i,j)}pairs for k=1k = 1 {\displaystyle k=1}, then k=2k = 2 {\displaystyle k=2}, etc. This process continues until k=Nk = N {\displaystyle k=N}, and we have found the shortest path for all (i,j) ( i , j ) {\displaystyle (i,j)}pairs using any intermediate vertices.

PSEUDOCODE

1 **let** dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)

2 **for each** vertex *v*

3 dist[*v*][*v*] ← 0

4 **for each** edge (*u*,*v*)

5 dist[*u*][*v*] ← w(*u*,*v*) *// the weight of the edge (*u*,*v*)*

6 **for** *k* **from** 1 **to** |V|

7 **for** *i* **from** 1 **to** |V|

8 **for** *j* **from** 1 **to** |V|

9 **if** dist[*i*][*j*] > dist[*i*][*k*] + dist[*k*][*j*]

10 dist[*i*][*j*] ← dist[*i*][*k*] + dist[*k*][*j*]

11 **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](https://en.wikipedia.org/wiki/Shortest-path_tree) can be calculated for each node in O(|E|) Θ ( | E | ) {\displaystyle \Theta (|E|)} time using O(|V|) Θ ( | V | ) {\displaystyle \Theta (|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|

|

V

|

×

|

V

|

{\displaystyle |V|\times |V|}

array of minimum distances initialized to

∞

{\displaystyle \infty }

(infinity)

**let** next be a

|

V

|

×

|

V

|

{\displaystyle |V|\times |V|}

|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]

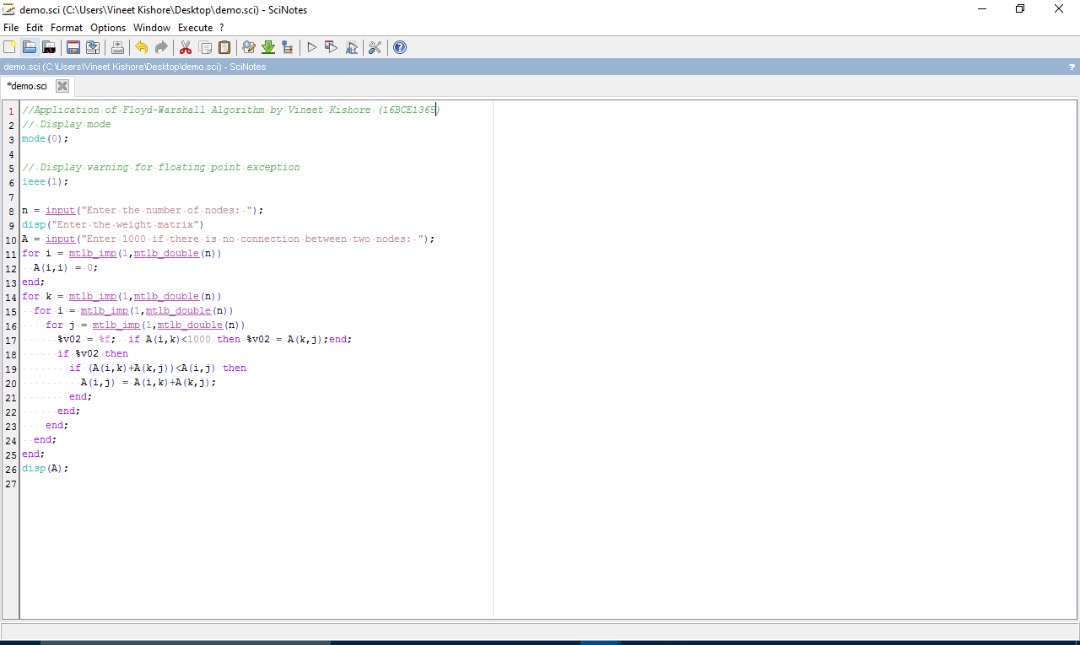
path.append(u)

**return** path

ANALYSIS

Let n n {\displaystyle n} be |V|| V | {\displaystyle |V|}, the number of vertices. To find all n2 n 2 {\displaystyle n^{2}} of shortestPath(i,j,k) s h o r t e s t P a t h ( i , j , k ) {\displaystyle \mathrm {shortestPath} (i,j,k)} (for all i i {\displaystyle i} and jj {\displaystyle j}) from those of shortestPath(i,j,k-1) s h o r t e s t P a t h ( i , j , k − 1 ) {\displaystyle \mathrm {shortestPath} (i,j,k-1)} requires 2n2 2 n 2 {\displaystyle 2n^{2}} operations. Since we begin with shortestPath(i,j,0) s h o r t e s t P a t h ( i , j , 0 ) = e d g e C o s t ( i , j ) {\displaystyle \mathrm {shortestPath} (i,j,0)=\mathrm {edgeCost} (i,j)} and compute the sequence of n n {\displaystyle n} matrices shortestPath(i,j,1) s h o r t e s t P a t h ( i , j , 1 ) {\displaystyle \mathrm {shortestPath} (i,j,1)}, shortestPath(i,j,2)s h o r t e s t P a t h ( i , j , 2 ) {\displaystyle \mathrm {shortestPath} (i,j,2)}, …… {\displaystyle \ldots }, shortestPath(i,j,n)s h o r t e s t P a t h ( i , j , n ) {\displaystyle \mathrm {shortestPath} (i,j,n)}, the total number of operations used is n.2n2 n ⋅ 2 n 2 = 2 n 3 {\displaystyle n\cdot 2n^{2}=2n^{3}}. Therefore, the [complexity](https://en.wikipedia.org/wiki/Computational_complexity_theory) of the algorithm is O(n3).

IMPLEMENTATION IN SCILAB



OUTPUT

