**Kruskal’s Algorithm**

**Utsav Rai 15BCE1352**

* 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

**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 OF KRUSKAL'S ALGORITHM**

**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 OF MINIMUM SPANNING TREE PROBLEM**

**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.

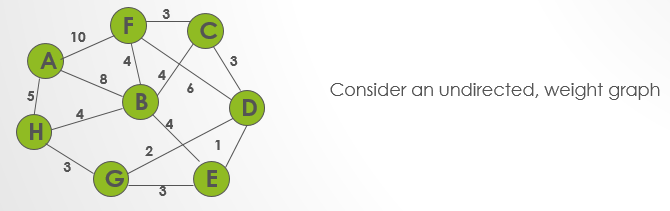
MST is fundamental problem with diverse applications

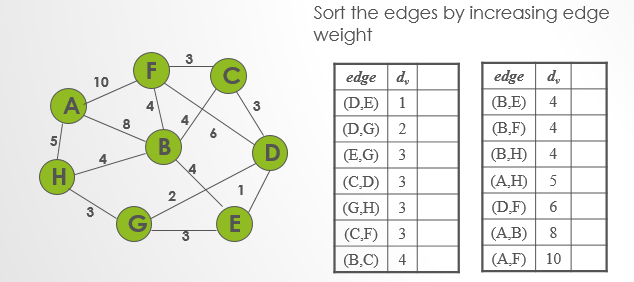
**Network design.**  
*– telephone, electrical, hydraulic, TV cable, computer, road*  
The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different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

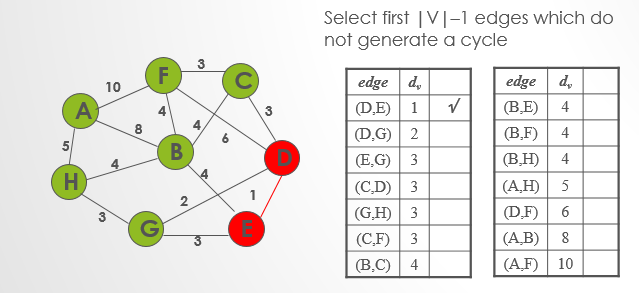
**Approximation algorithms for NP-hard problems.**  
*– traveling salesperson problem, Steiner tree*  
A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest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

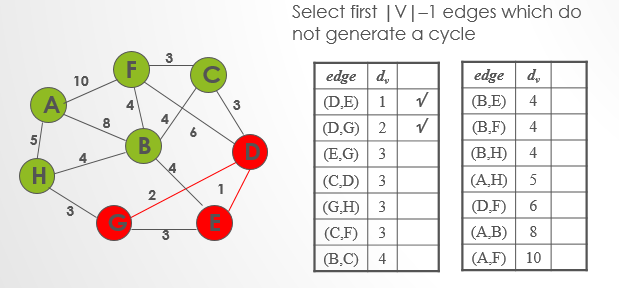
**Cluster analysis**  
k clustering problem can be viewed as finding an MST and deleting the k-1 most  
expensive edges.

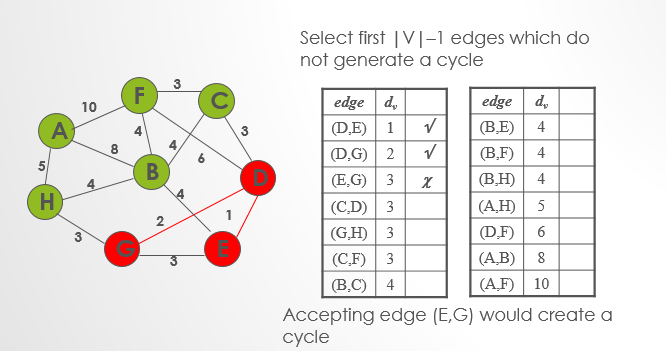
**WALK-THROUGH**

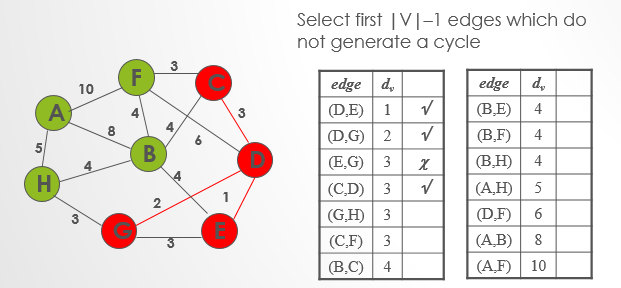


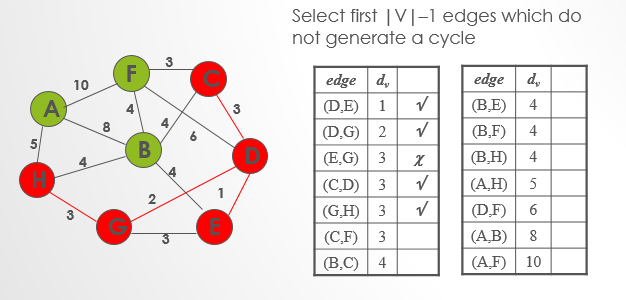


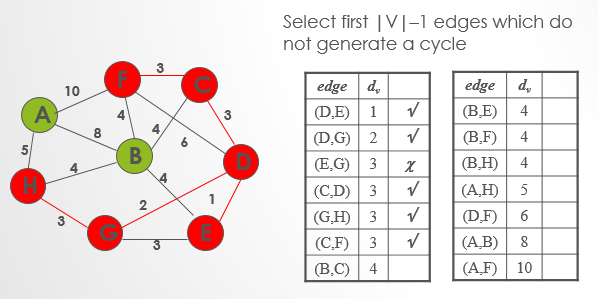


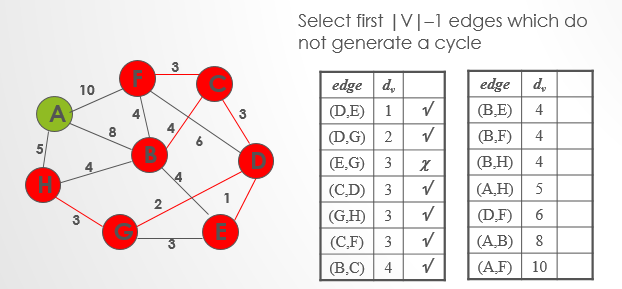


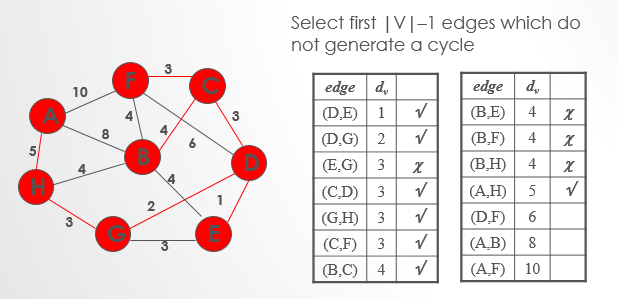


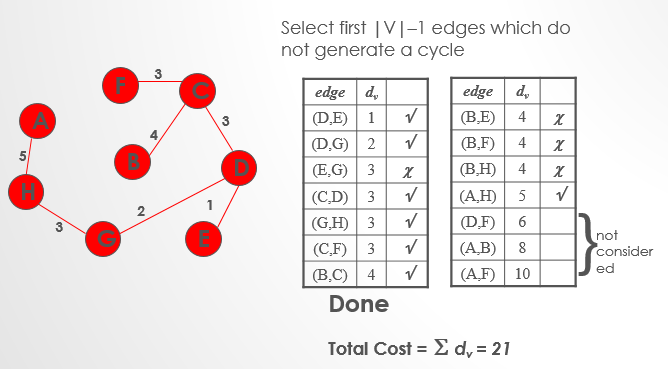












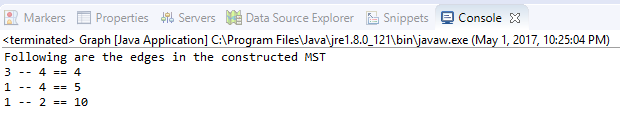
**Time Complexity**

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)

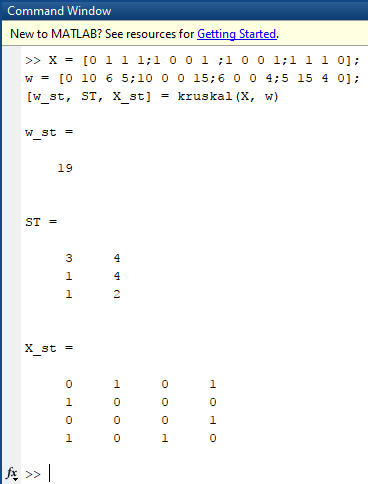
rnk(r\_j) = rnk(r\_j) + 1;

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:**

