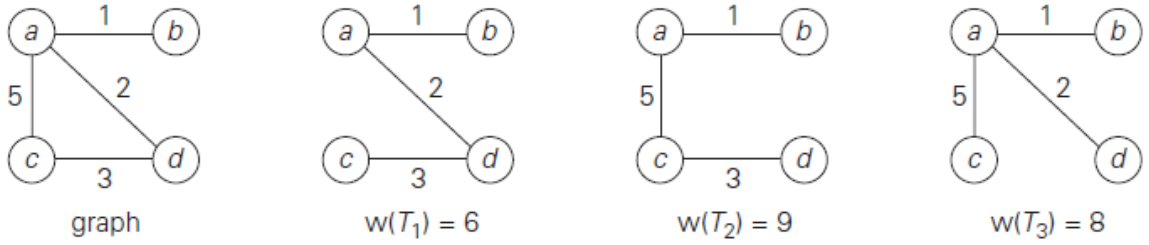
Text Book: Introduction to the Design and Analysis of Algorithms Author: Anany Levitin 2 nd Edition

Unit-4

7. Kruskal's's Algorithm

A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.



Graph and its spanning trees, with T1 being the minimum spanning tree.

Applications:

1. The MST has applications in many practical situations: given *n* points, connect them in the cheapest possible way so that there will be a path between every pair of points.

2. It has direct applications to the design of all kinds of networks— including communication, computer, transportation, and electrical—by providing the cheapest way to achieve connectivity. It identifies clusters of points in data sets.
3. It has been used for classification purposes in archeology, biology, sociology, and other sciences.
4. It is also helpful for constructing approximate solutions to more difficult problems such the traveling salesman problem

Kruskal's algorithm

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph $G = \langle V, E \rangle$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest. The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

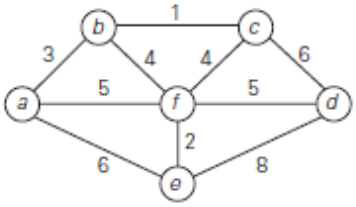
Algorithm

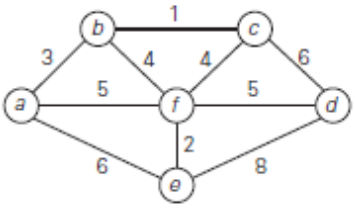
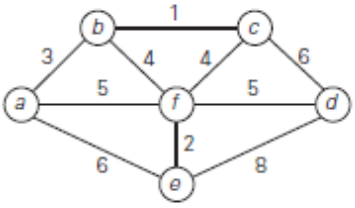
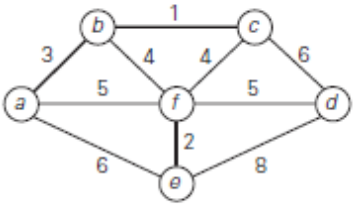
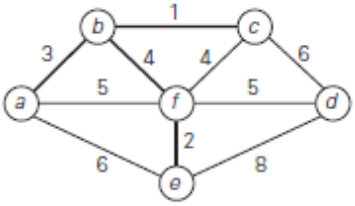
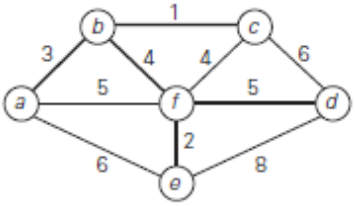
ALGORITHM *Kruskal(G)*

```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = ⟨V, E⟩
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights w(e_{i_1}) ≤ ··· ≤ w(e_{i_|E|})
E_T ← ∅;  ecounter ← 0        //initialize the set of tree edges and its size
k ← 0                         //initialize the number of processed edges
while ecounter < |V| − 1 do
    k ← k + 1
    if E_T ∪ {e_{i_k}} is acyclic
        E_T ← E_T ∪ {e_{i_k}};  ecounter ← ecounter + 1
return E_T
```

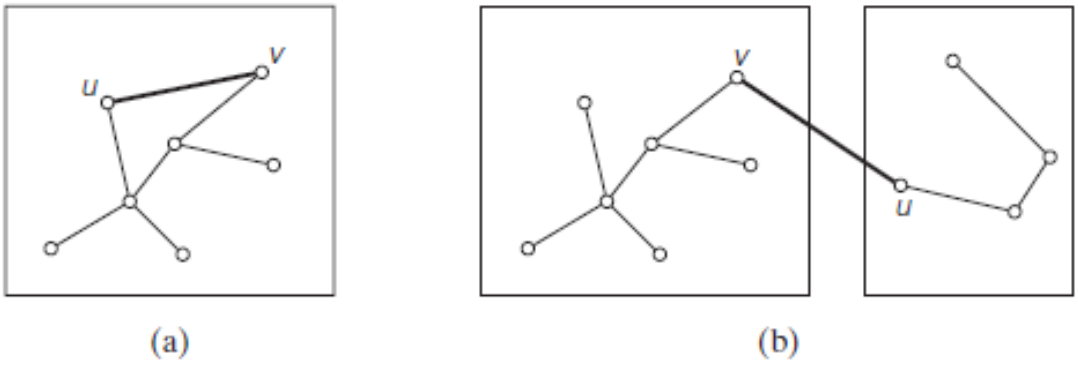
We can consider the algorithm's operations as a progression through a series of forests containing *all* the vertices of a given graph and *some* of its edges. The initial forest consists of $|V|$ trivial trees, each comprising a single vertex of the graph. The final forest consists of a single tree, which is a minimum spanning tree of the graph. On each iteration, the algorithm takes the next edge *(u, v)* from the sorted list of the graph's edges, finds the trees containing the vertices *u* and *v*, and, if these trees are not the same, unites them in a larger tree by adding the edge *(u, v)*.

Example:



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 5 6 6 8	
df 5		

Fortunately, there are efficient algorithms for doing so, including the crucial check for whether two vertices belong to the same tree. They are called ***unionfind*** algorithms. With an efficient union-find algorithm, the running time of Kruskal's algorithm will be dominated by the time needed for sorting the edge weights of a given graph. **Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in *O(|E| log |E|).***



GURE 9.6 New edge connecting two vertices may (a) or may not (b) create a cycle.

Disjoint Subsets and Union-Find Algorithms

Kruskal's algorithm is one of a number of applications that require a dynamic partition of some *n* element set *S* into a collection of disjoint subsets *S*1, *S*2, . . . , *Sk.*After being initialized as a collection of *n* one-element subsets, each containing a different element of *S*, the collection is subjected to a sequence of intermixed union and find operations.

An abstract data type of a collection of disjoint subsets of a finite set with the following operations:

makeset(x): creates a one-element set {*x*}. It is assumed that this operation can be applied to each of the elements of set *S* only once.

find(x): returns a subset containing *x*.

union(x, y): constructs the union of the disjoint subsets *Sx* and *Sy* containing *x* and *y,* respectively, and adds it to the collection to replace *Sx* and *Sy,* which are deleted from it.

For example, let S = {1, 2, 3, 4, 5, 6}. Then *makeset(i)* creates the set {*i*} and applying this operation six times initializes the structure to the collection of six singleton sets:

{1}, {2}, {3}, {4}, {5}, {6}.

Performing *union(*1, 4*)* and *union(*5, 2*)* yields

{1, 4}, {5, 2}, {3}, {6},

and, if followed by *union(*4, 5*)* and then by *union(*3, 6*),* we end up with the disjoint subsets

{1, 4, 5, 2}, {3, 6}.