DESIGN AND ANALYSIS OF ALGORITHMS
Course Code: 19CS4DCDAA

Module 4

GREEDY TECHNIQUE: Introduction, Prim’s Algorithm, Kruskal’s algorithm, Dijkstra’s Algorithm, The Bellman–Ford algorithm. An activity-selection problem, Huffman codes.

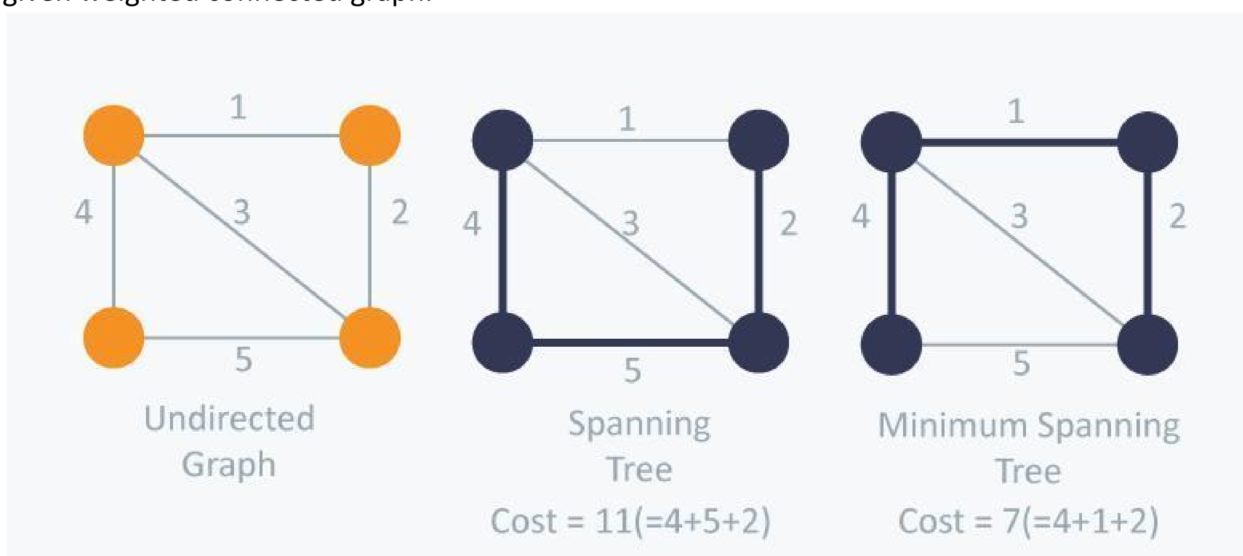
Limitations of Algorithm Power: Lower-Bound Arguments, Decision Trees, P, NP, and NP-Complete Problems, NP-Complete Problems

GREEDY TECHNIQUE

A greedy method is a problem solving technique that always tries to find the best solution that works in stages considering one input at a time with the hope that we get an optimal solution.

A spanning tree is a tree in which all nodes are connected without forming a cycle.

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.

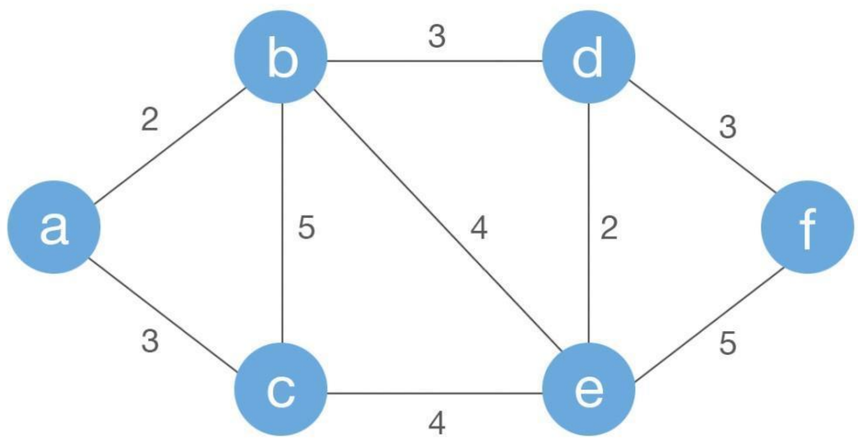


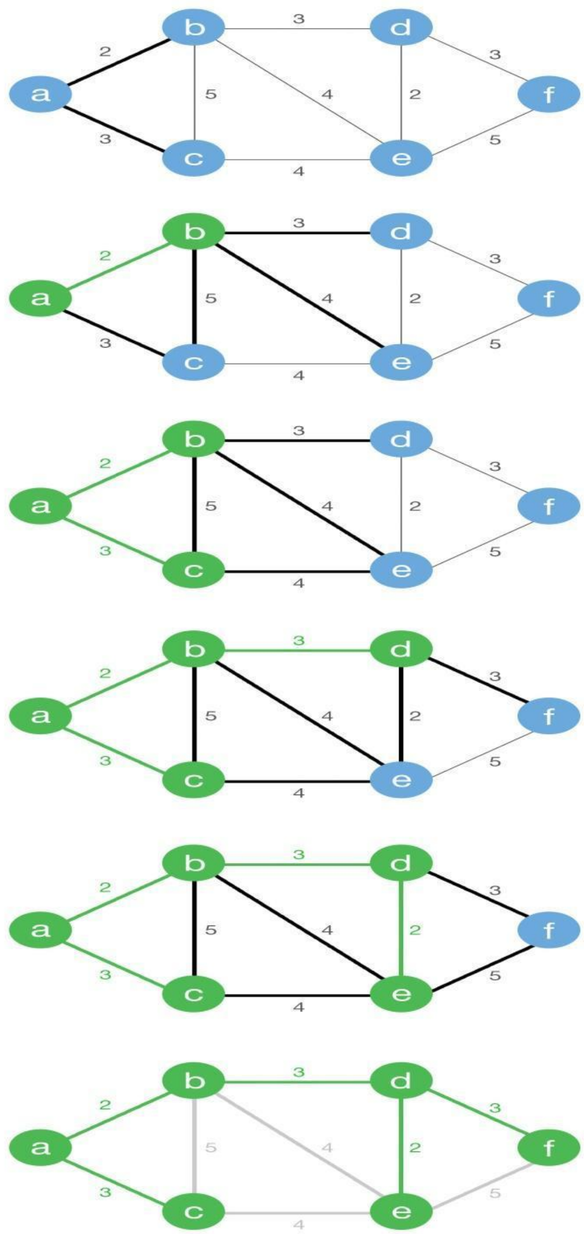
Prim's Algorithm

ALGORITHM *Prim(G)*

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
  find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
  such that v is in V_T and u is in $V - V_T$
  $V_T \leftarrow V_T \cup \{u^*\}$
  $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

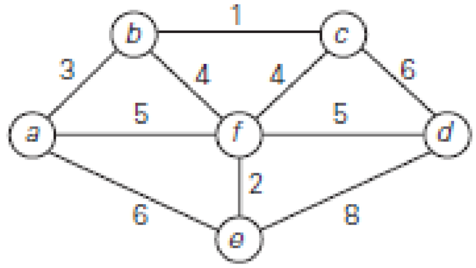
Example: Apply prims algorithm



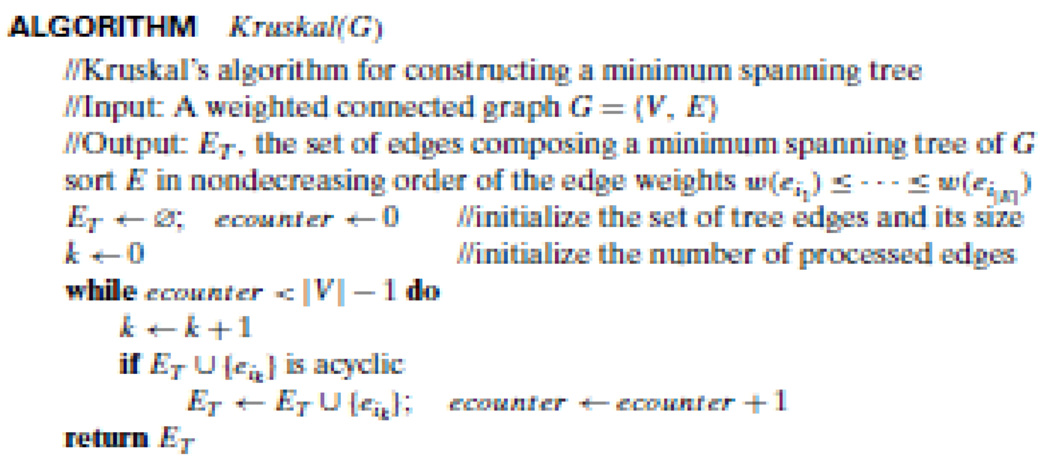


Edge weight total = 13

Ex:



Kruskal's Algorithm



ALGORITHM *Kruskal(G)*

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //initialize the set of tree edges and its size
$k \leftarrow 0$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
$k \leftarrow k + 1$
if $E_T \cup \{e_{i_k}\}$ is acyclic
$E_T \leftarrow E_T \cup \{e_{i_k}\}$; $ecounter \leftarrow ecounter + 1$
return E_T

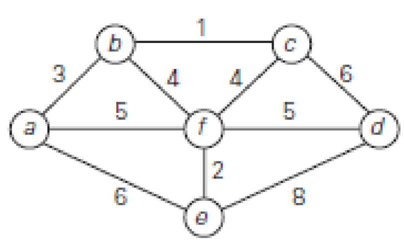
Below are the steps for finding MST using Kruskal's algorithm

1. *Sort all the edges in non-decreasing order of their weight.*

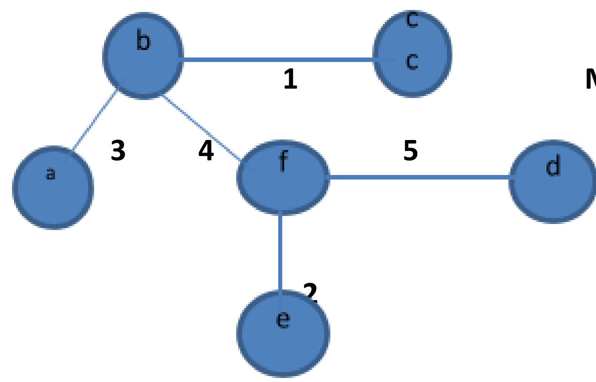
2. *Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.*

3. *Repeat step#2 until there are (V-1) edges in the spanning tree.*

Eg: Apply Kruskal Algorithm



Edges	**(b,c)**	**(f,e)**	**(a,b)**	**(b,f)**	**(c,f)**	**(a,f)**	**(f,d)**	**(c,d)**	**(a,e)**	**(e,d)**
weight	**1**	**2**	**3**	**4**	**4**	**5**	**5**	**6**	**6**	**8**
selected	**yes**	**yes**	**yes**	**yes**	**no**	**no**	**yes**	**no**	**no**	**no**

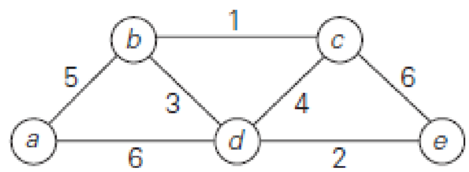


Min cost=15

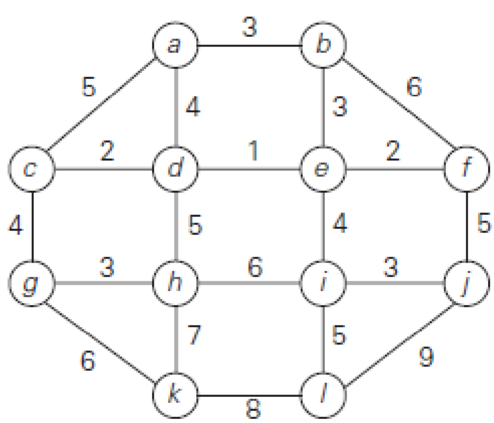
Eg:

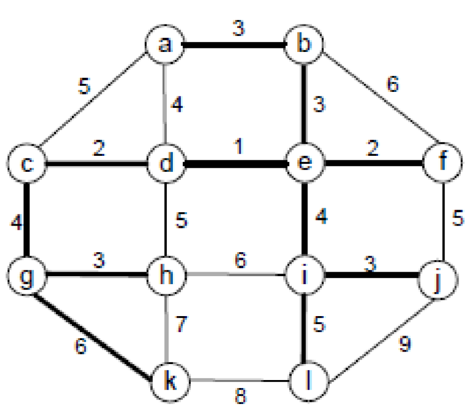
1. Apply Kruskal's algorithm to find a minimum spanning tree of the following graphs.

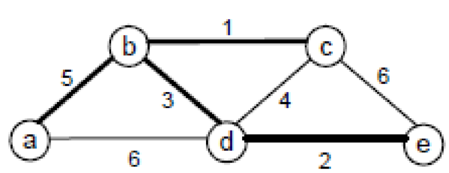
   a.



b.

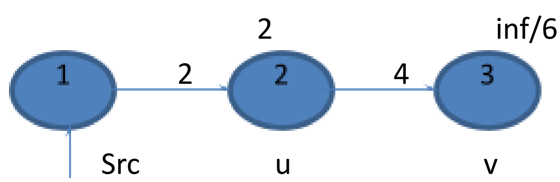






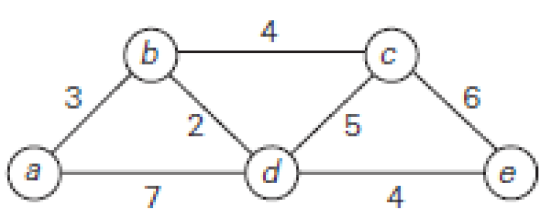
Dijkstra’s Algorithm (Single Source Shortest Path Algorithm)

- Single source shortest path problem is the one where we compute the shortest distance from a given source vertex V to all other vertices in the graph.
- Applicable for directed and undirected graphs.
- Not applicable with negative weights.
- Relaxation of edge
  If(d[u]+c(u,v)<d[v])
  d[v]=d[u]+c(u,v)



d[3]=min(d[3],d[2]+c(2,3))
min(inf, 2+4)
d[3]=6

Apply Dijkstra’s Algorithm
Source= a



d[a]	**0**	**0**	**0**	**0**
d[b]	**3**	**3**	**3**	**3**
d[c]	**inf**	**7**	**7**	**7**
d[d]	**7**	**5**	**5**	**5**

d[e]	inf	inf	9	9
Distance table				

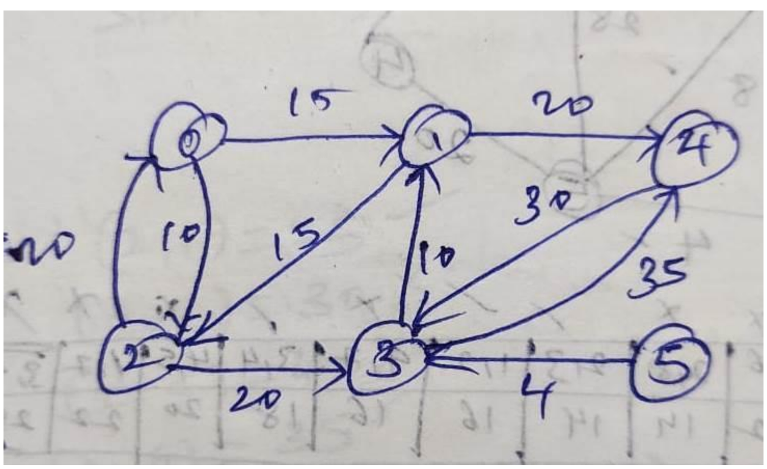
		a	b	c	d	e
a		0	3	inf	7	inf
b		3	0	4	2	inf
c		inf	4	0	5	6
d		7	2	5	0	4
e		inf	inf	6	4	0

d[v]=min(d[v],d[u]+c[u][v])

Src	UNVISITED NODES	d[v]=min(d[v],d[u]+c[u][v])	U	d[U]
a	b,c,d,e	----	b	3
a,b	c,d,e	d[c]=min(inf, 3+4)=7 d[d]= min(7, 3+2)=5 d[e]= min(inf, 3+inf)=inf	d	5
a,b,d	c,e	d[c]=min(7, 5+5)=7 d[e]= min(inf, 5+4)=9	c	7
a.b,d,c	e	d[e]= min(9, 7+6)=9	e	9
a.b,d,c,e				

Apply Dijkstra's Algorithm

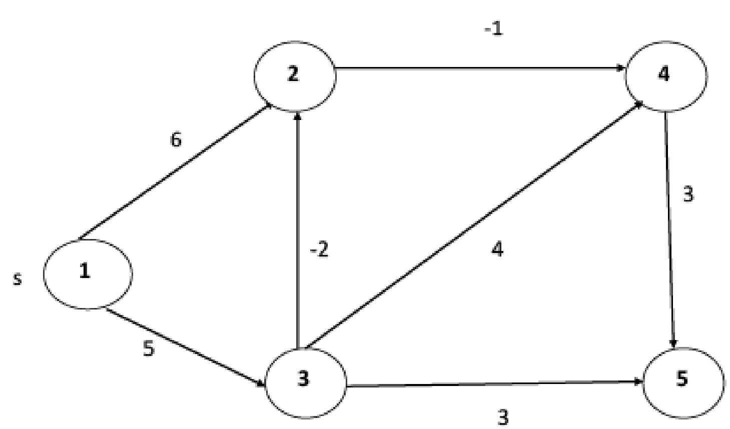
Source= 5



Bellman Ford Algorithm

- Given a graph and a source vertex *src* in graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.
- *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs.*
- **Algorithm**
  Following are the detailed steps.
  Input: Graph and a source vertex *src*
  Output: Shortest distance to all vertices from *src*. **If there is a negative weight cycle, then shortest distances are not calculated**, negative weight cycle is reported.
  1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
  2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
  …..**a)** Do following for each edge u-v
  ………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]
  ………………….dist[v] = dist[u] + weight of edge uv
  3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
  If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"
  The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle.

  Example:



Step 1: List the edges **Initially**

Edges	Cost
1-2	6
1-3	5
2-4	-1
3-2	-2
3-4	4
3-5	3
4-5	3

Vertices	1(src)	2	3	4	5
Distance	0	inf	inf	inf	inf
Predecessor Vertex	-	-	-	-	-

d[v]=min(d[v],d[u]+c[u][v])

Total No of Iterations=n-1=5-1=4

1st Iteration

Vertices	1(src)	2	3	4	5
Distance	0	3	5	5	8
Predecessor Vertex	-	3	1	2	3

2nd Iteration

Vertices	1(src)	2	3	4	5
Distance	0	3	5	2	5
Predecessor Vertex	-	3	1	2	4

3rd Iteration

Vertices	1(src)	2	3	4	5
Distance	0	3	5	2	5
Predecessor Vertex	-	3	1	2	4

4^{th} Iteration

Vertices	**1(src)**	**2**	**3**	**4**	**5**
Distance	**0**	**3**	**5**	**2**	**5**
Predecessor Vertex	**-**	**3**	**1**	**2**	**4**

Huffman Codes:

Huffman's algorithm

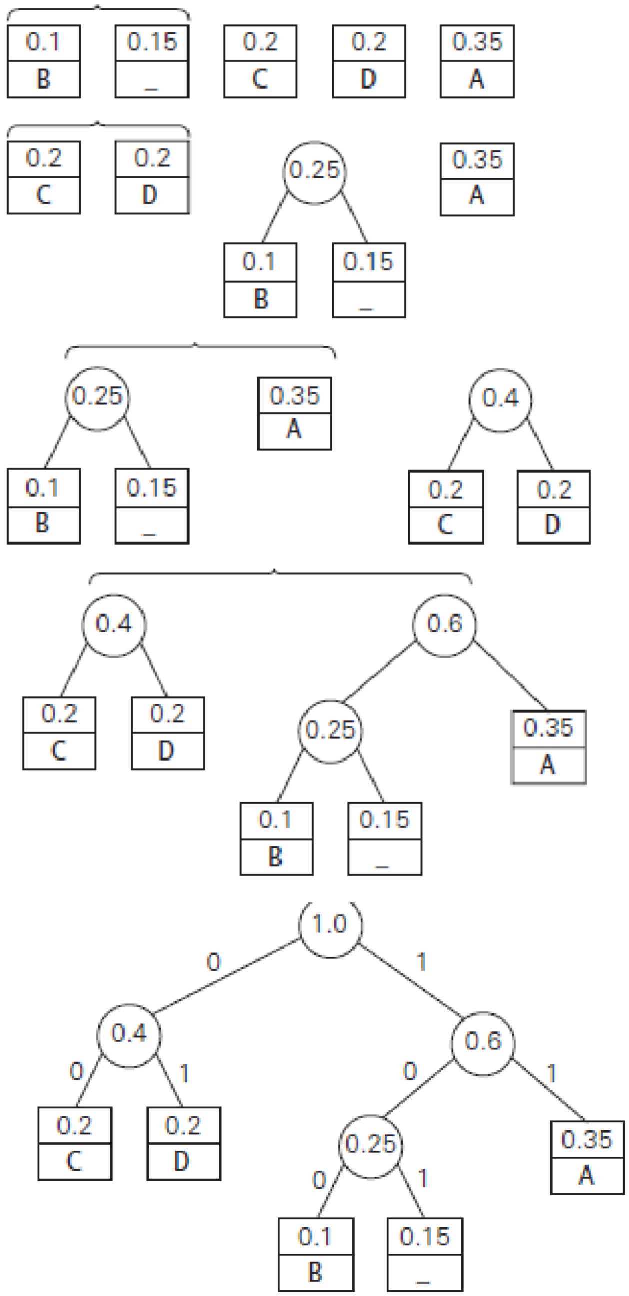
Step 1 Initialize *n* one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's ***weight***. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight . Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called a ***Huffman tree***. It defines—in the manner described above—a ***Huffman code***.

EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15



The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD.