CS 2009 – Design and Analysis of Algorithms

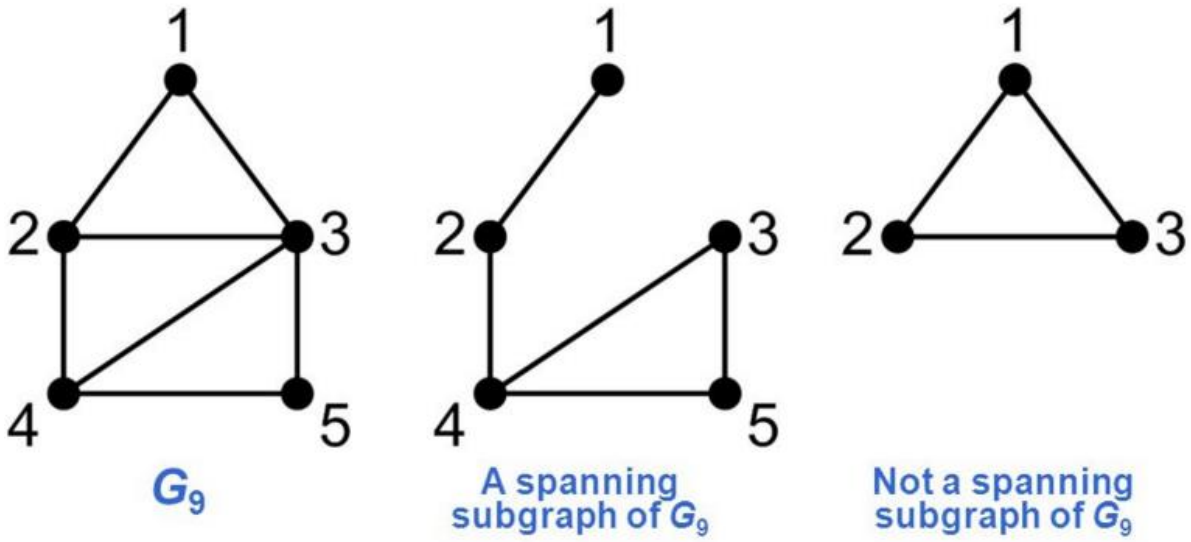
Minimum Spanning Trees

Rizwan Ul Haq

Minimum Spanning Tree

- A tree is a connected graph without cycles.
- A subgraph that spans (reaches out to) all vertices of a graph is called a spanning subgraph.
- A subgraph that is a tree and that spans (reaches out to) all vertices of the original graph is called a spanning tree.
- Among all the spanning trees of a weighted and connected graph, the one (possibly more) with the least total weight is called a minimum spanning tree (MST).

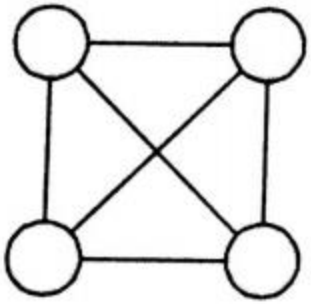
sub graph


1
2
3
4
5
G_9
1
2
3
4
5
A spanning subgraph of G_9
1
2
3
Not a spanning subgraph of G_9

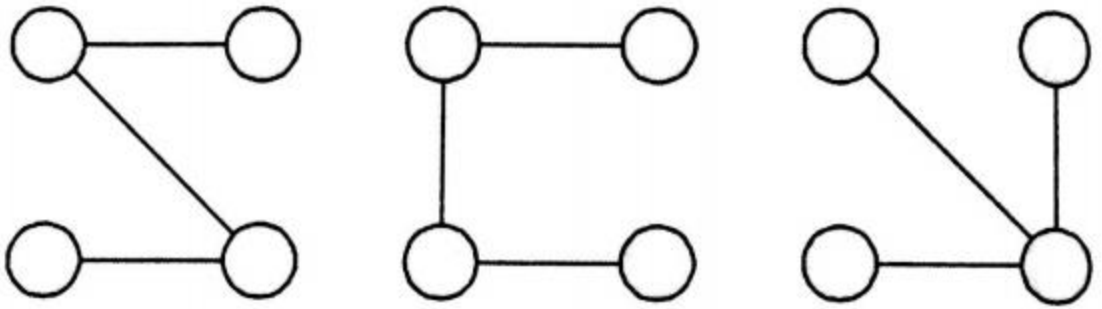
Example

A graph *G*:

G:

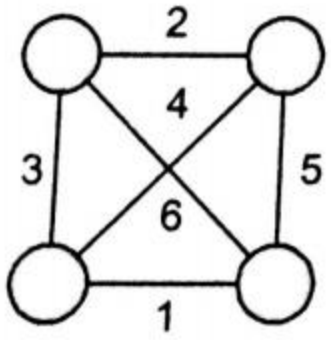


Three (of the many possible) spanning trees from graph *G*:

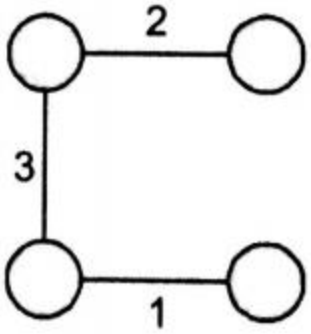


A weighted graph *G*:

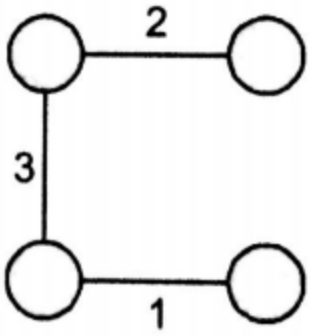
G:



The minimum spanning tree from weighted graph *G*:



- Length of spanning tree :
  - Sum of all weights of edges of spanning tree
  - In previous example it is 6

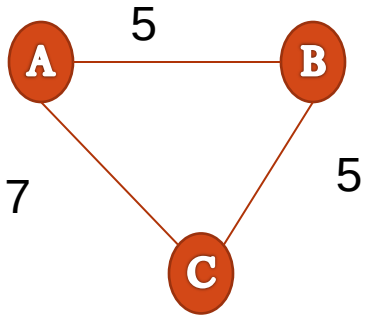


Application

- Minimum spanning trees have direct applications in the design of networks, including computer networks, telecommunications networks, transportation networks, and water supply networks

Shortest Path vs Spanning Tree

- In MST, requirement is to reach each vertex once (create graph tree) and total (collective) cost of reaching each vertex is required to be minimum among all possible combinations.
- In Shortest Path, requirement is to reach destination vertex from source vertex with lowest possible cost (shortest weight). So here we do not worry about reaching each vertex instead only focus on source and destination vertices and that's where lies the difference.



Prim's Algorithm

- An algorithm for determining minimal spanning tree in connected undirected weighted graph

Two ways

- Two ways to build a minimum spanning tree.
  - PRIM'S ALGORITHM
  - KRUSKAL'S ALGORITHM

Prim's Algorithm

- In this case we start with a source node, say 's'. We maintain an array of the distance of each node from the MST being built.
- We also maintain an array of the predecessor of each node. When a node has not yet been selected into the MST it's predecessor is NULL.
- Initially we set all distances as infinity except that of the source which is set to zero.

Steps:

- We now start the iterations.
- We select the node that has the least distance, say ‘u’.
  - In the first loop, the source is selected because its distance is zero.
  - In subsequent iterations this process ensures that the minimum weight edge is selected. We now include the node in the tree.
- Then we relax all the nodes connected to the node.
  - This means that we see for each node v connected to u whether the distance of v is greater than the weight of edge (u,v). If it is, we assign w(u,v) to the distance of v.
  - This step ensures that after each new node is added to the tree, the distance of the remaining nodes from the tree is updated.
  - Thus when we have all the edges in the tree, our minimum spanning tree is ready.

PRIM'S Algorithm

```
PRIM(G, w, r)
1.   for each u belongs to V [G]
2.       do key[u] ← ∞
3.          π[u] ← NIL
4.   key[r] ← 0
5.   Q ← V [G] // build min heap
6.   while Q ≠ Ø
7.     do u ← EXTRACT-MIN(Q)
8.        for each v belongs to Adj[u]
9.        do if v belongs to Q and w(u, v) < key[v]
10.            then π[v] ← u
11.                 key[v] ← w(u, v)
                  // modify node and maintain heap property
```

Analysis

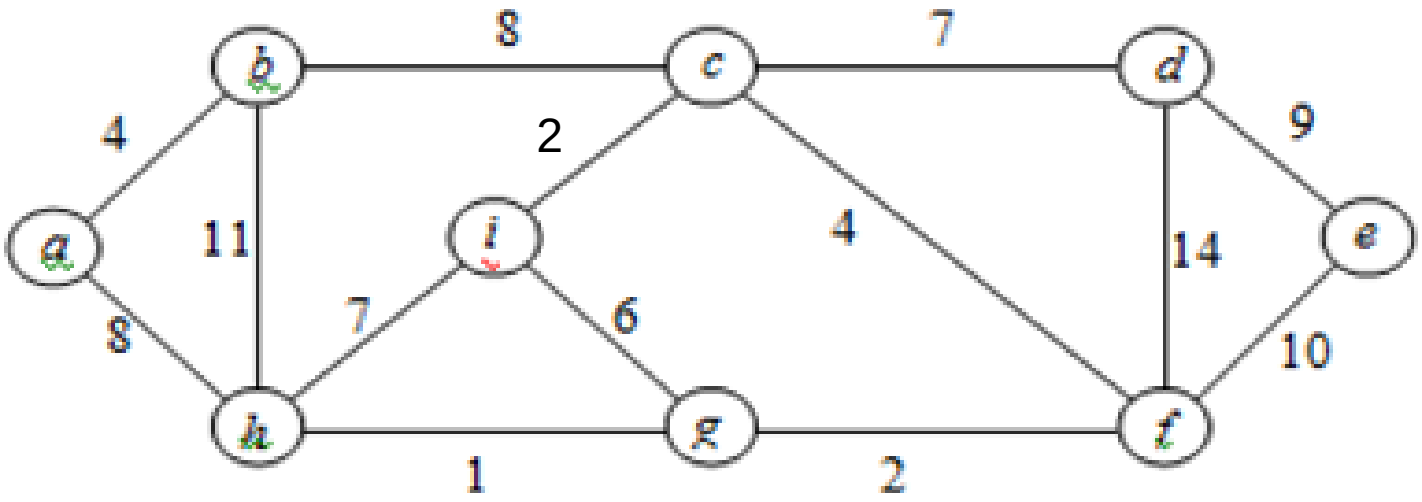
- O(E logV)
  - 1-3 => O(V)
  - 7-10 =>(O(E logV + V log V)

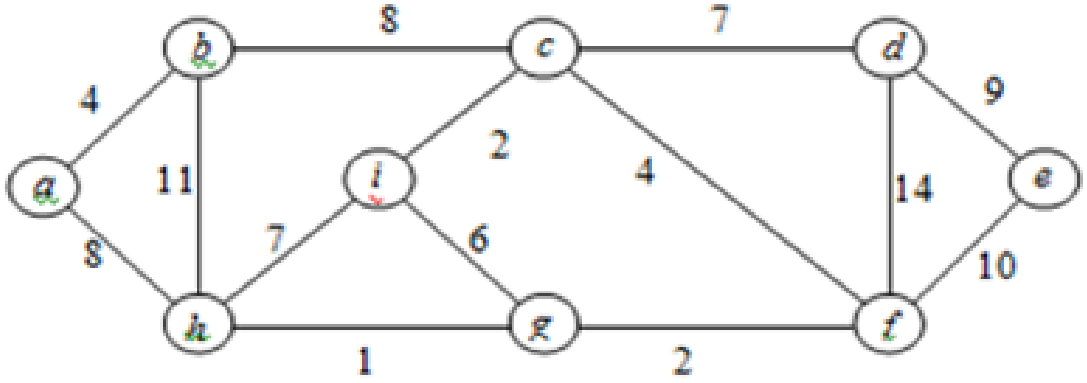
```
PRIM(G, w, r)
1.  for each u belongs to V [G]
2.  do key[u] ← ∞
3.      π[u] ← NIL
4.  key[r] ← 0
5.  Q ← V [G] // build min heap
6.  while Q ≠ Ø
7.  do u ← EXTRACT-MIN(Q)
8.      for each v belongs to Adj[u]
9.       do if v belongs to Q and w(u, v) < key[v]
10.           then π[v] ← u
11.                key[v] ← w(u, v)
                 // modify node and maintain heap property
```

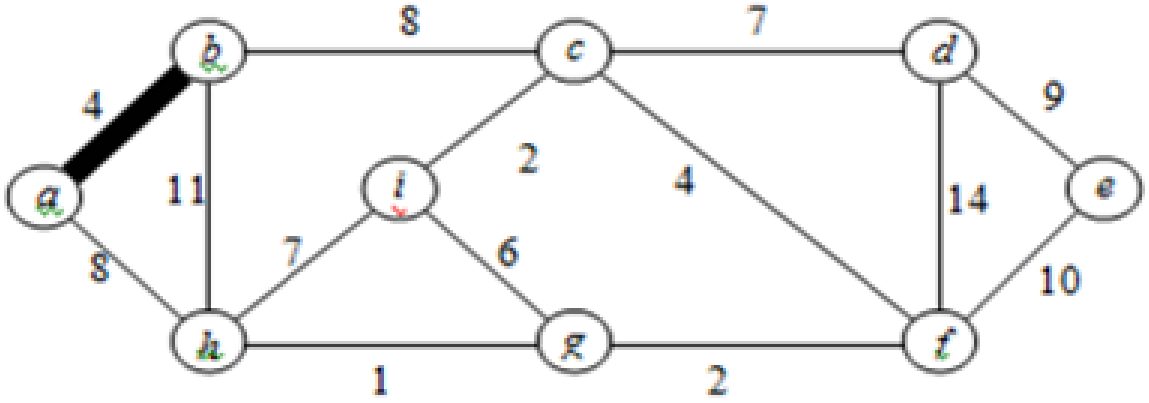
- Time required for one call to EXTRACT-MIN(Q)=O(log V)[using min priority queue].
- The while loop is executing total V times.so EXTRACT-MIN(Q) is called V times.so total time required for EXTRACT-MIN(Q)=O(VlogV).
- The for loop is executing total 2E times(as total length of all the adjacency lists=2E for undirected graph).Time required for executing line 11=O(log v)[using DECREASE_KEY operation on the min heap).Line 11 is executing total 2E times.so total time required to execute line 11=O(2Elog V)=O(Elog v).
- The for loop at line 1 will be executed for V times. using BUILD_HEAP procedure, to perform lines 1 to 5,it will require O(V) times
- Total time complexity O(VlogV+ElogV+V)=O(ElogV)

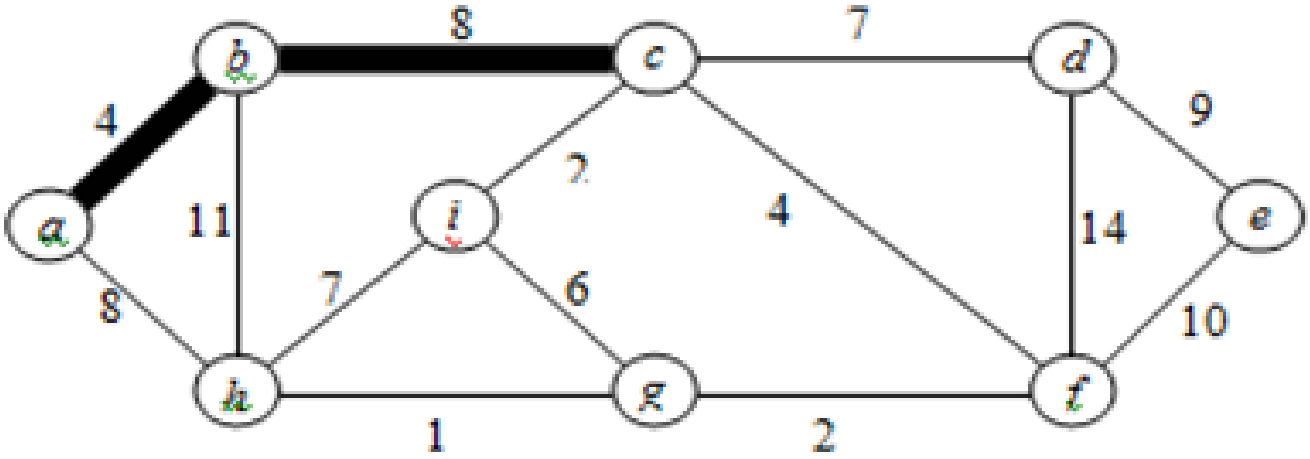
- **NOTE: Prims work fine for negative weight edges**

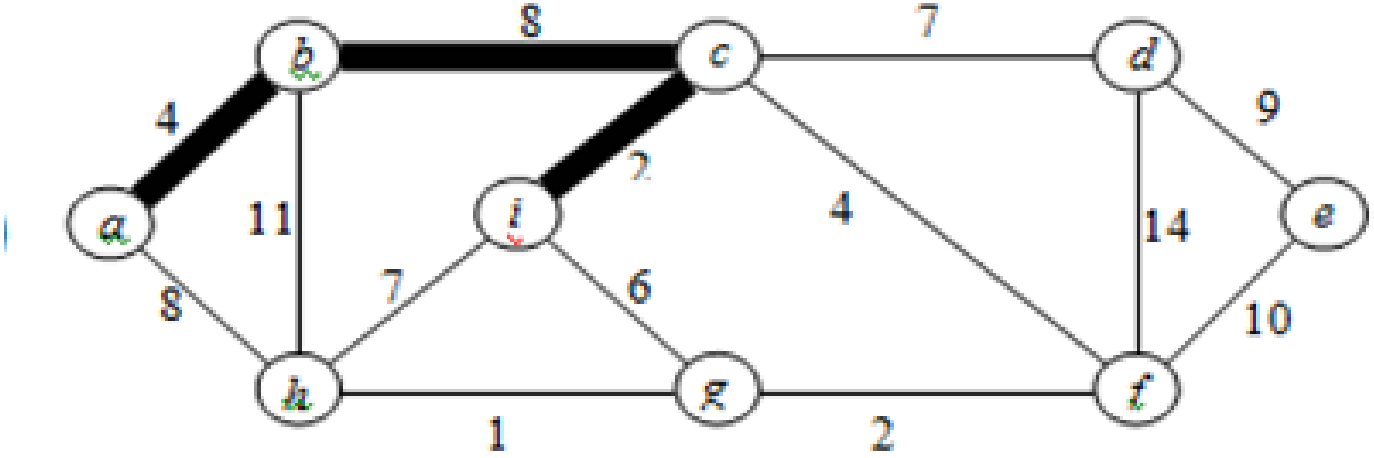
Example

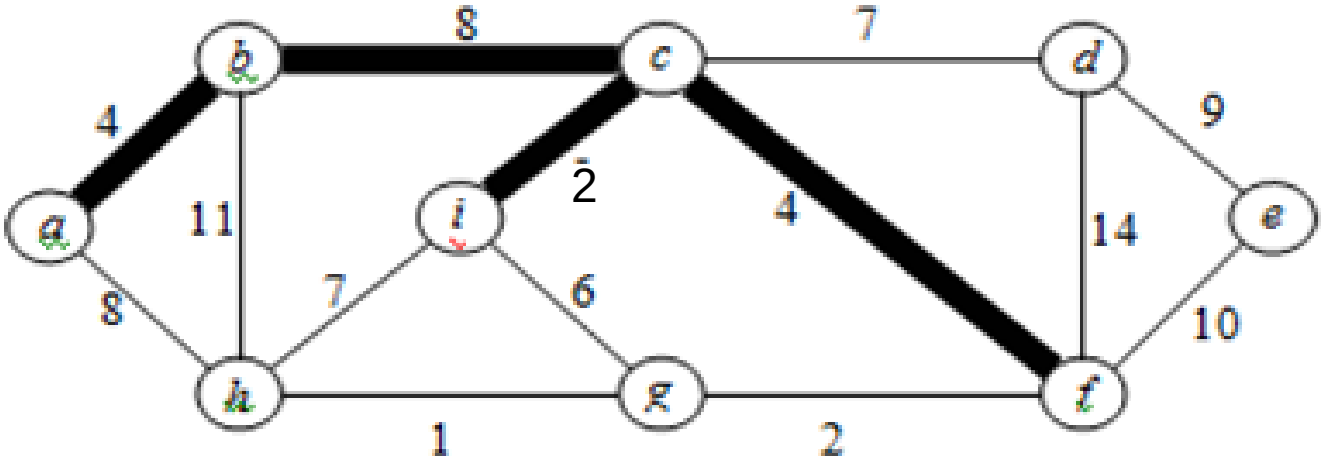


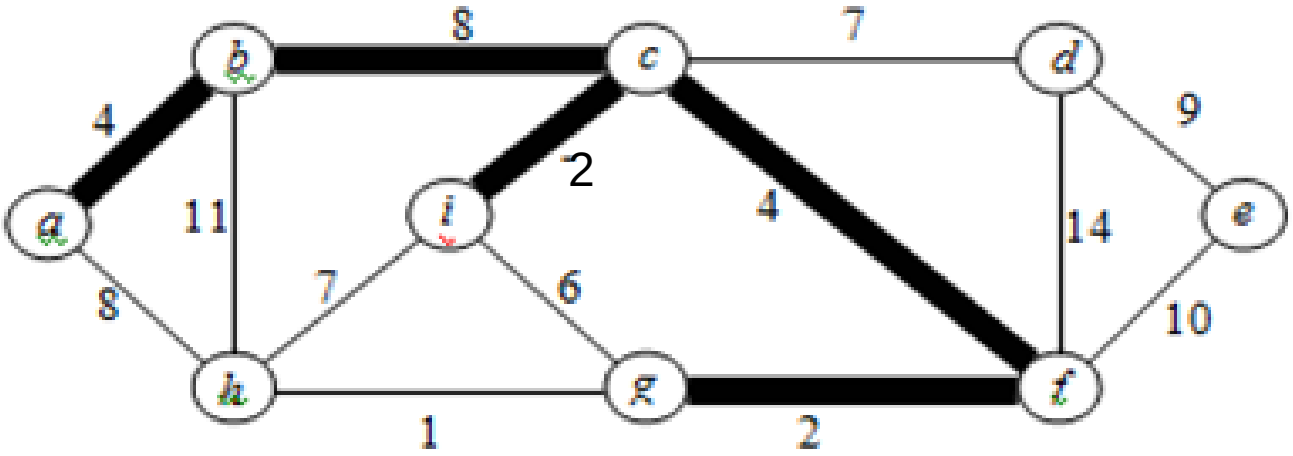

b
c
d
a
i
e
h
g
f
8
7
4
9
11
2
4
14
8
7
6
10
1
2

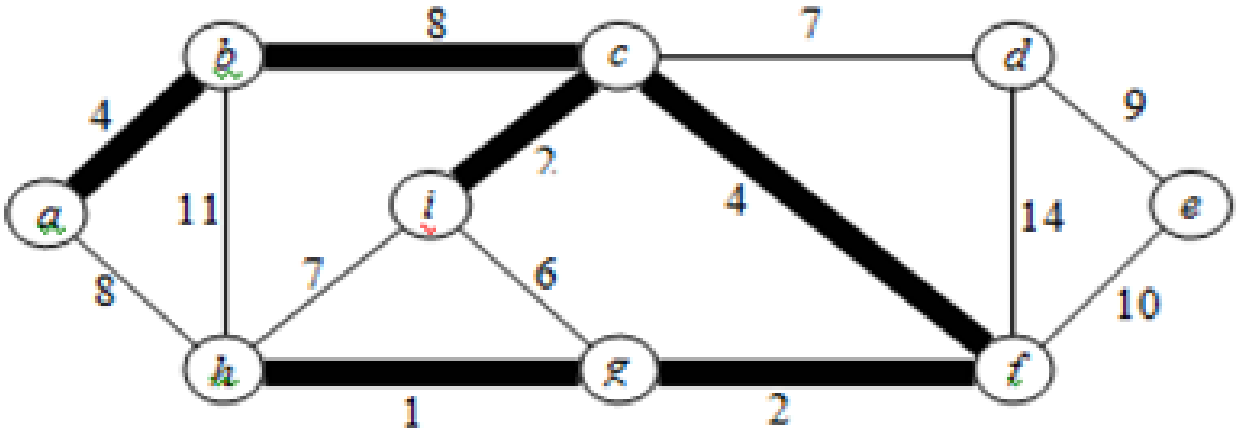

b
c
d
a
i
e
h
g
f
8
7
4
9
11
2
4
14
8
7
6
10
1
2

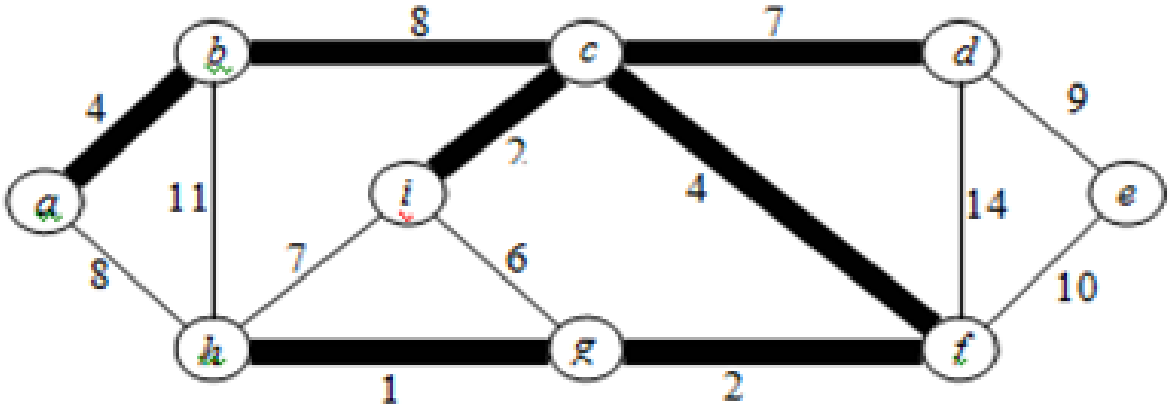

a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
14
9
10
1
2


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
6
7
14
9
10
1
2

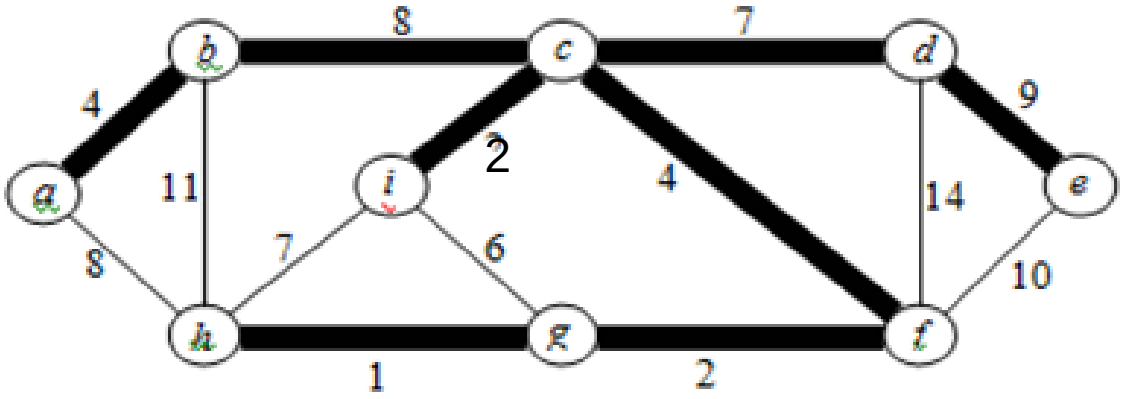

a
b
c
d
e
f
g
h
i
4
8
11
8
8
7
2
4
6
7
14
9
10
1
2


a
b
c
d
e
f
g
h
i
4
8
11
8
8
7
2
4
6
7
14
9
10
1
2

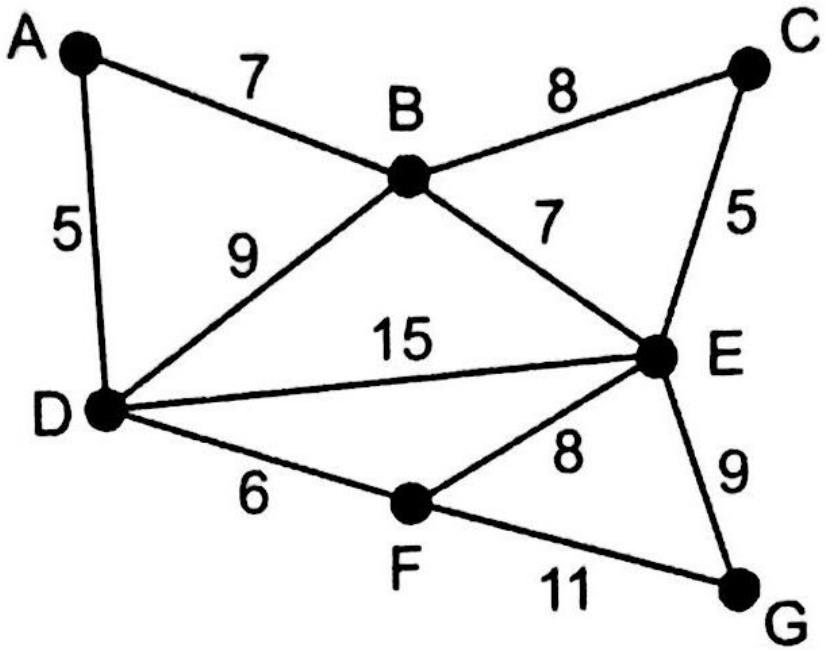

a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
7
6
14
9
10
1
2


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
7
6
14
9
10
1
2

Safe Edges



Task :



- Is MST Unique???
  - Yes, if edge lengths are distinct
  - No Otherwise

- In MST there is a single path between any two vertices??? Is that true.
  - Yes, because there is no cycle

Kruskal's algorithm

- In graph theory this algorithm finds a minimum spanning tree for a connected weighted undirected graph.

- This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

Kruskal's algorithm

- This algorithm creates a forest of trees.
- Initially the forest consists of n single node trees (and no edges).
- At each step, we add one edge (the cheapest one) so that it joins two trees together.
- If it were to form a cycle, it would simply link two nodes that were already part of a single connected tree, so that this edge would not be needed.

MST-KRUSKAL(G)

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}     //edge is added in A
8            UNION(u, v)          Merge two tree/sets i.e take union of both sets
9  return A
```

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0
```

```
FIND-SET(x)
1  if x ≠ x.p
2       x.p = FIND-SET(x.p)
3  return x.p
```

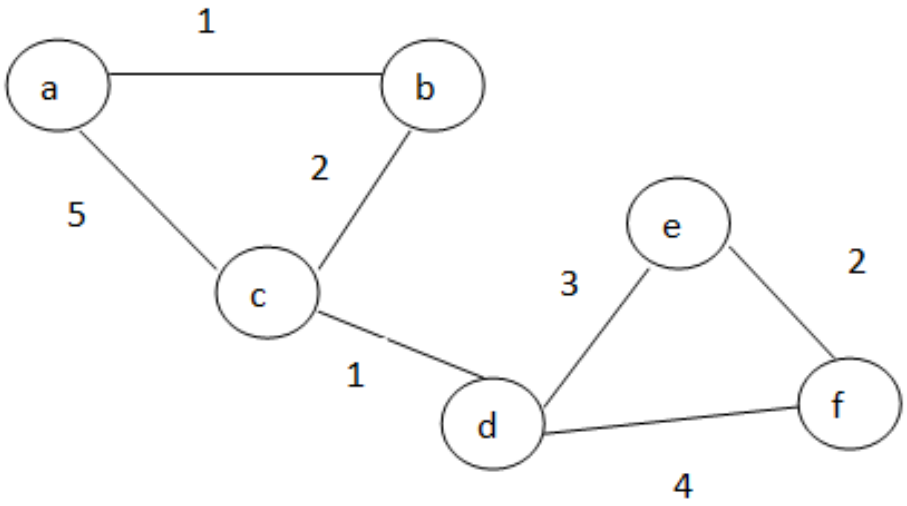
```
UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2        y.p = x
3  else x.p = y
4        if x.rank == y.rank
5              y.rank = y.rank + 1
```

Analysis

- O (E log (E))
  - Line 1 : O(1)
  - Line 2-3: (O(V))
  - Line 4: Sorting algorithm O(E log E)
  - Line 5-8 :O(E log V)

```
MST-KRUSKAL(G, w)
1  A = ∅
2  for each vertex v ∈ G.V
3       MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6       if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u, v)}
8            UNION(u, v)
9  return A
```

Example



```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0
```

```
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
```

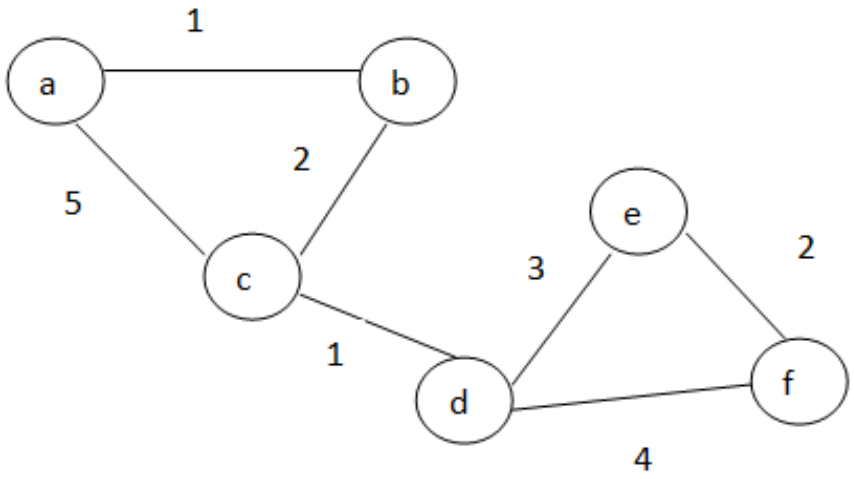
- {a} => Rank=0; P/ root =a
- {b}=> Rank=0; P/ root =b
- {c}=> Rank=0 ; P/ root =c
- {d}=> Rank=0; P/ root =d
- {e}=> Rank=0; P/ root =e
- {f}=> Rank=0; P/ root =f

Rank of x= Height of x

P of x= Parent of x (here P represent root of respective tree)

4 sort the edges of $G.E$ into nondecreasing order by weight w

(a,b) = 1
(c,d) = 1
(b,c) = 2
(e,f) = 2
(d,e) = 3
(d,f) = 4
(a,c) = 5



```
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
```

(a,b)

Find-Set(a) = a

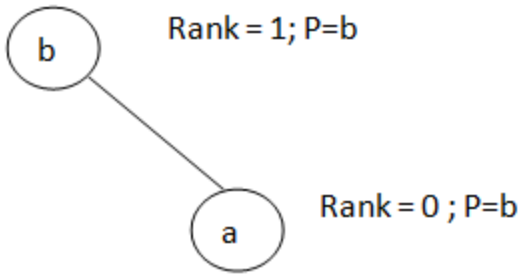
Find-Set(b) =b

a ≠b

Union (a,b)

a.P=b

a.Rank=b.Rank=> b.Rank=0+1




```
UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))
```

```
LINK(x, y)
1  if x.rank > y.rank
2      y.p = x
3  else x.p = y
4      if x.rank == y.rank
5          y.rank = y.rank + 1
```

```
FIND-SET(x)
1  if x ≠ x.p
2      x.p = FIND-SET(x.p)
3  return x.p
```

a	b	0
b	b	1
c	c	0
d	d	0
e	e	0
f	f	0

```
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
```

(c,d)

Find-Set(c) = c

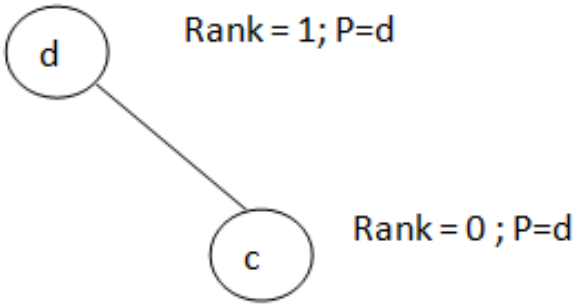
Find-Set(d) =d

c≠d

Union (c,d)

c.P=d

c.Rank=d.Rank=> d.Rank=0+1




```
UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2      y.p = x
3  else x.p = y
4      if x.rank == y.rank
5          y.rank = y.rank + 1
```

V	P	R
a	b	0
b	b	1
c	d	0
d	d	1
e	e	0
f	f	0

```
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
```

(b,c)

Find-Set(b) = b

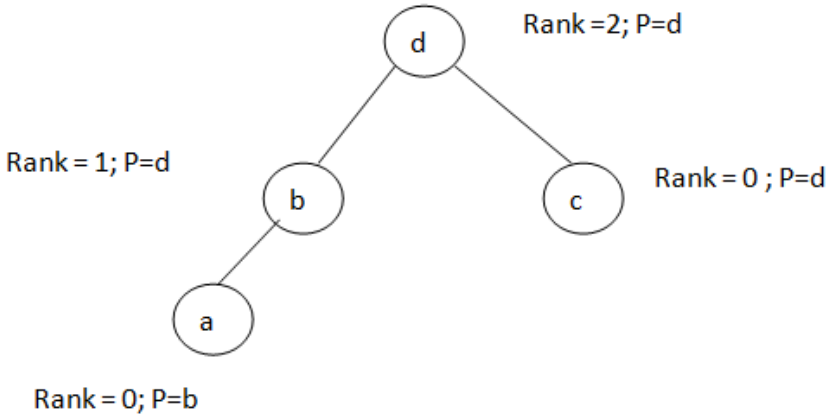
Find-Set(c) =d

b≠d

Union (b,d)

b.P=d

b.Rank=d.Rank=> d.Rank=1+1=2




```
UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2      y.p = x
3  else x.p = y
4      if x.rank == y.rank
5          y.rank = y.rank + 1
```

V	P	R
a	b	0
b	d	1
c	d	0
d	d	2
e	e	0
f	f	0

```
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
```

(e,f)

Find-Set(e) = e

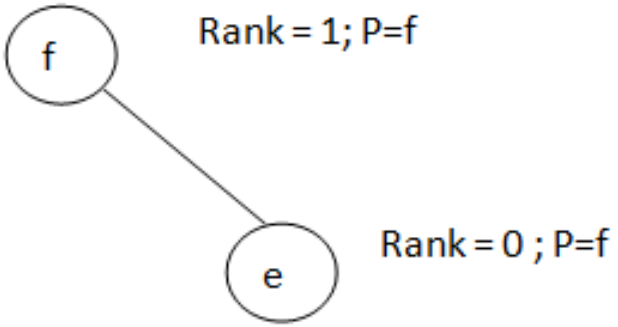
Find-Set(f) =f

e≠f

Union (e,f)

e.P=f

e.Rank=f.Rank=> f.Rank=0+1



V	P	R
a	b	0
b	d	1
c	d	0
d	d	2
e	f	0
f	f	1

```
UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2      y.p = x
3  else x.p = y
4      if x.rank == y.rank
5          y.rank = y.rank + 1
```

```
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
```

(d,e)

Find-Set(d) = d

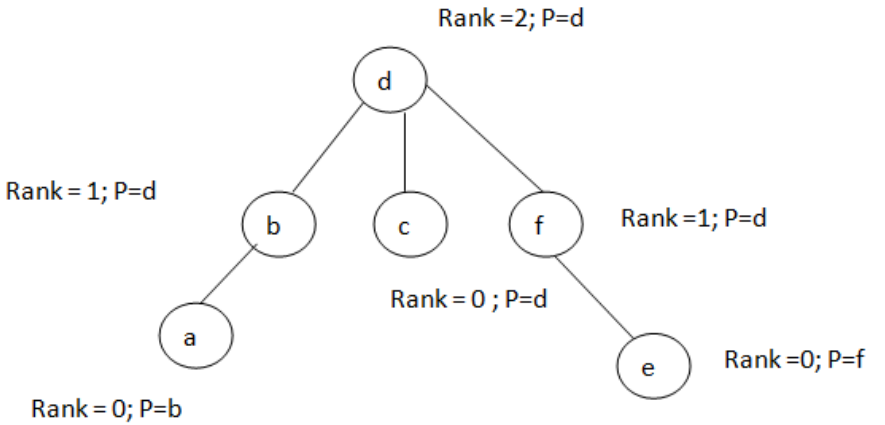
Find-Set(e) =f

d≠f

Union (d,f)

d.Rank>f.Rank

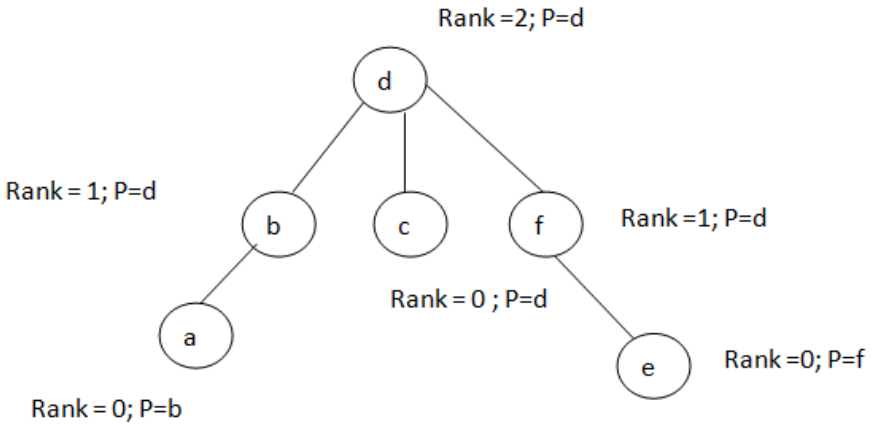
f.P=d



V	P	R
a	b	0
b	d	1
c	d	0
d	d	2
e	f	1
f	d	1

```
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
```

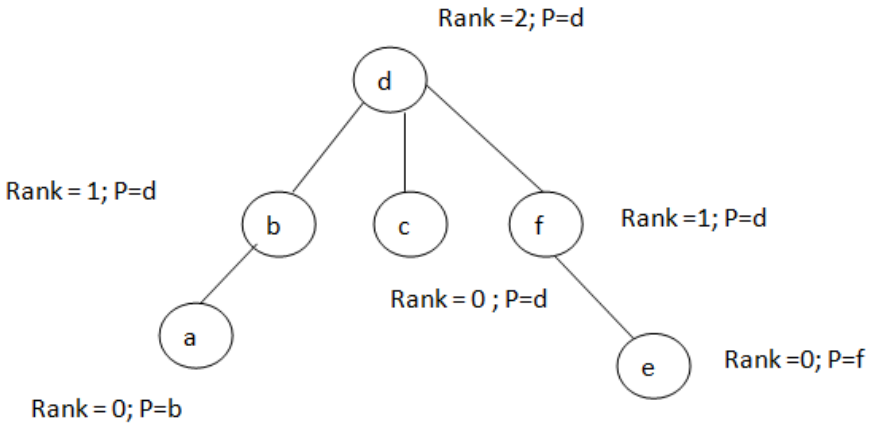
(d,f)
Find-Set(d) = d
Find-Set(f) =d
d=d
//discard edge



V	P	R
a	b	0
b	d	1
c	d	0
d	d	2
e	f	1
f	d	0

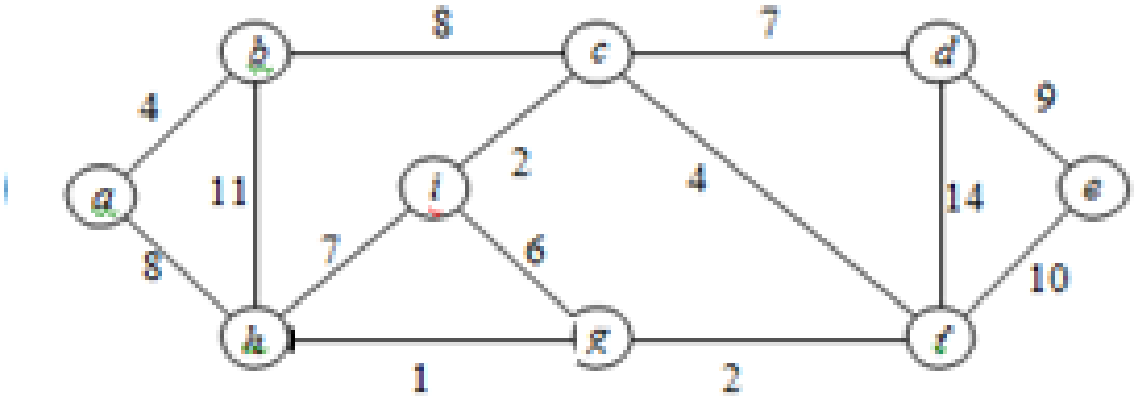
```
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
```

(a,f)
Find-Set(a) = d
Find-Set(f) =d
d=d
//discard edge



V	P	R
a	b	0
b	d	1
c	d	0
d	d	2
e	f	1
f	d	0

Example

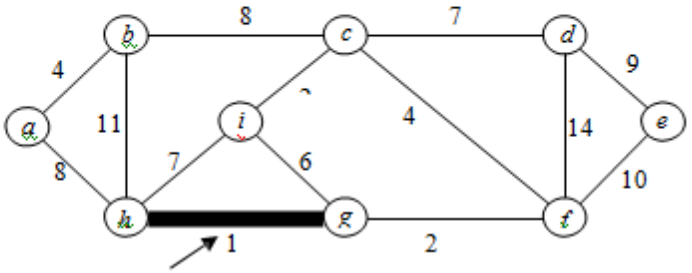


Edge Processed	Connected Components / Collection of Disjoint Sets
Initial Sets	{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}
(g,h)	{a}, {b}, {c}, {d}, {e}, {f}, {g,h}, {i}
(c,i)	{a}, {b}, {c,i}, {d}, {e}, {f}, {g,h}
(f,g)	{a}, {b}, {c,i}, {d}, {e}, {f,g,h}
(a,b)	{a,b} {c,i}, {d}, {e}, {f,g,h}
(c,f)	{a,b}, {c,f,g,h,i}, {d}, {e}
(g,i)	discarded
(c,d)	{a,b}, {c,d,f,g,h,i}, {e}
(h,i)	discarded
(a,h)	{a,b,c,d,f,g,h,i}, {e}
(b,c)	discarded
(d,e)	{a,b,c,d,e,f,g,h,i}
(f,e)	discarded
(b,h)	discarded
(d,f)	discarded

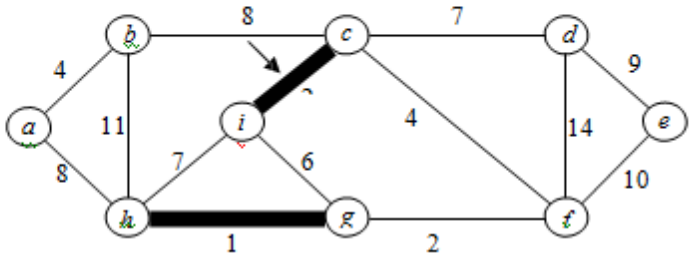
A = {(g,h), (c,i), (f,g), (a,b), (c,f), (c,d), (a,h), (d,e)}

Execution of Kruskal's Algorithm

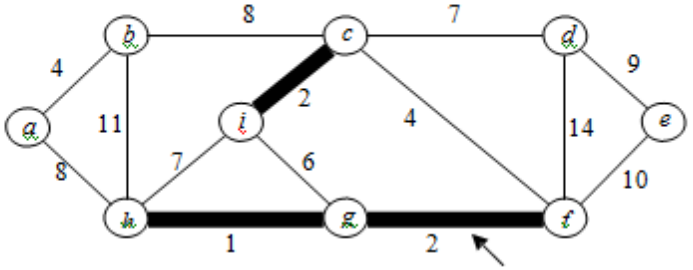
(a)



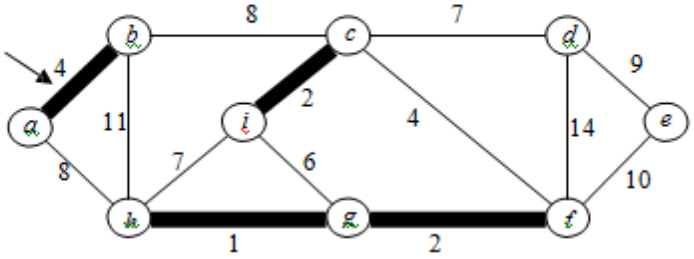
(b)



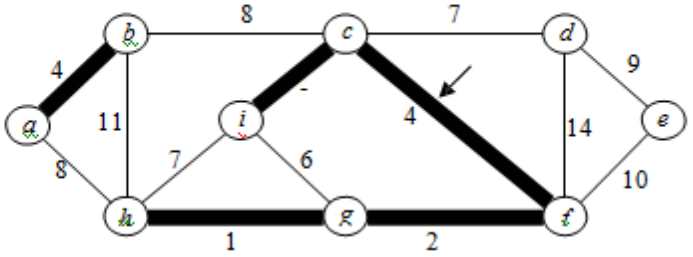
(c)



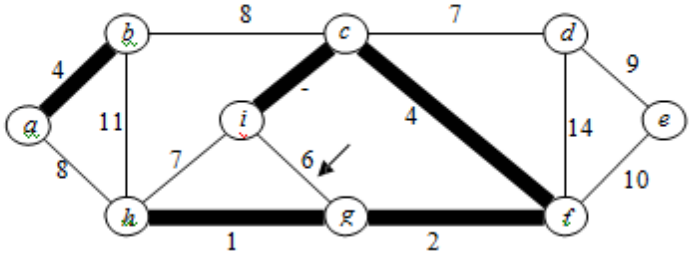
(d)



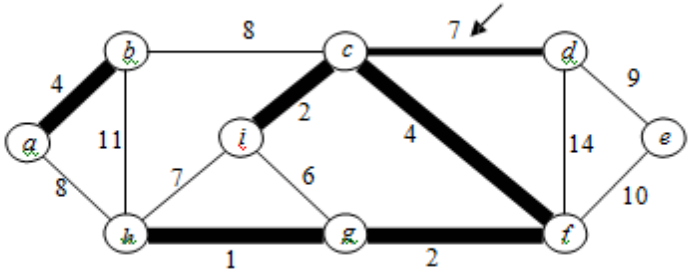
(e)



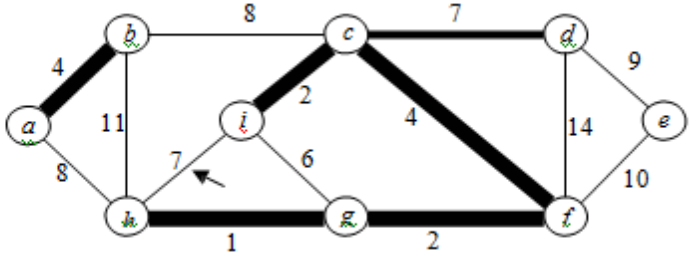
(f)



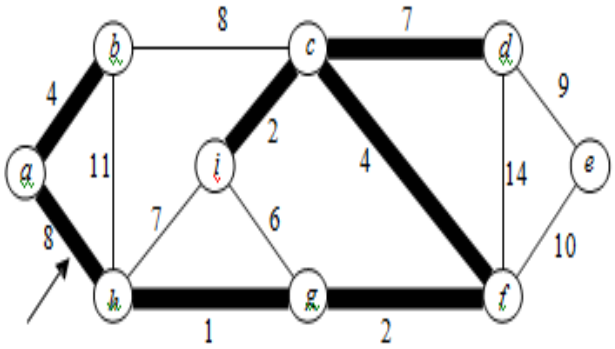
(g)



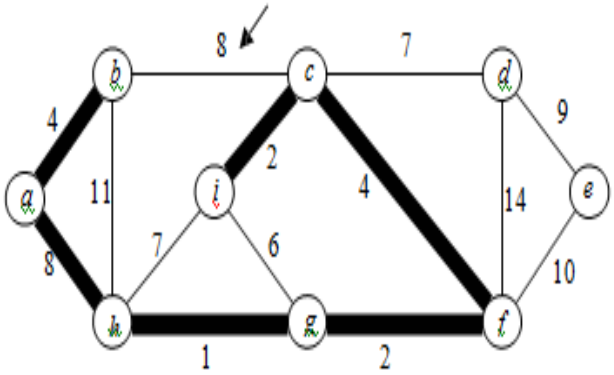
(h)



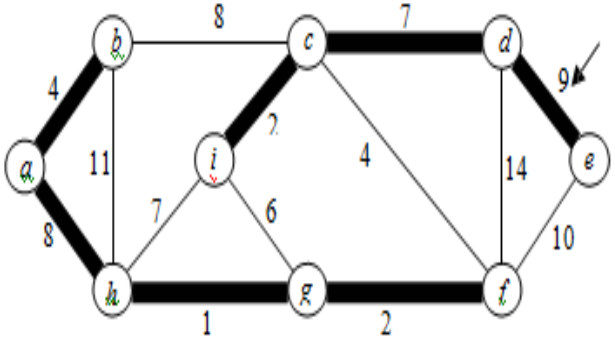
(i)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
7
6
14
9
10
1
2

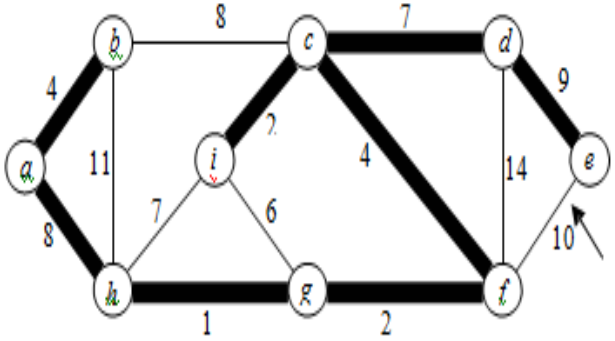
(j)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
7
6
14
9
10
1
2

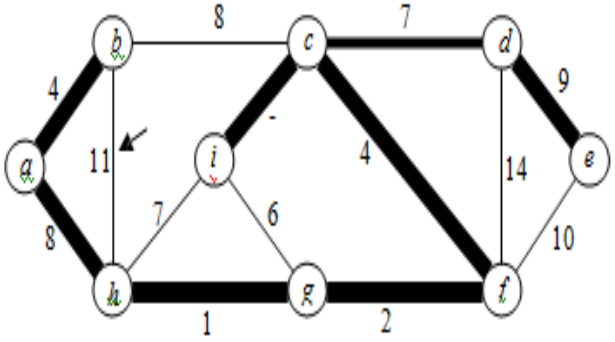
(k)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
7
6
14
9
10
1
2

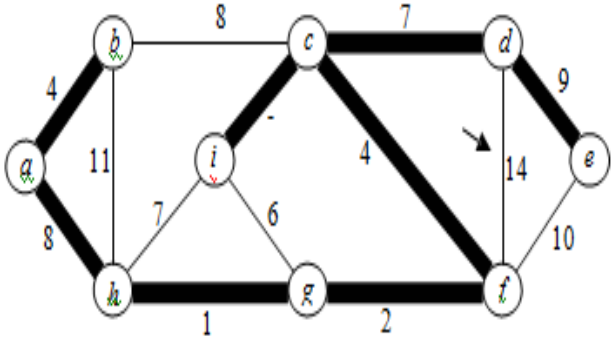
(l)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
4
7
6
14
9
10
1
2

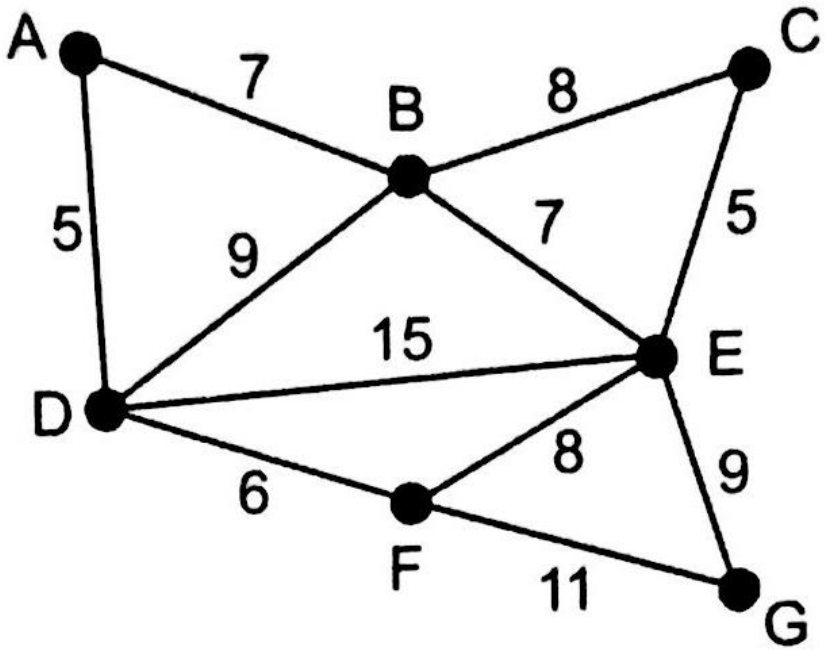
(m)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
-
4
7
6
14
9
10
1
2

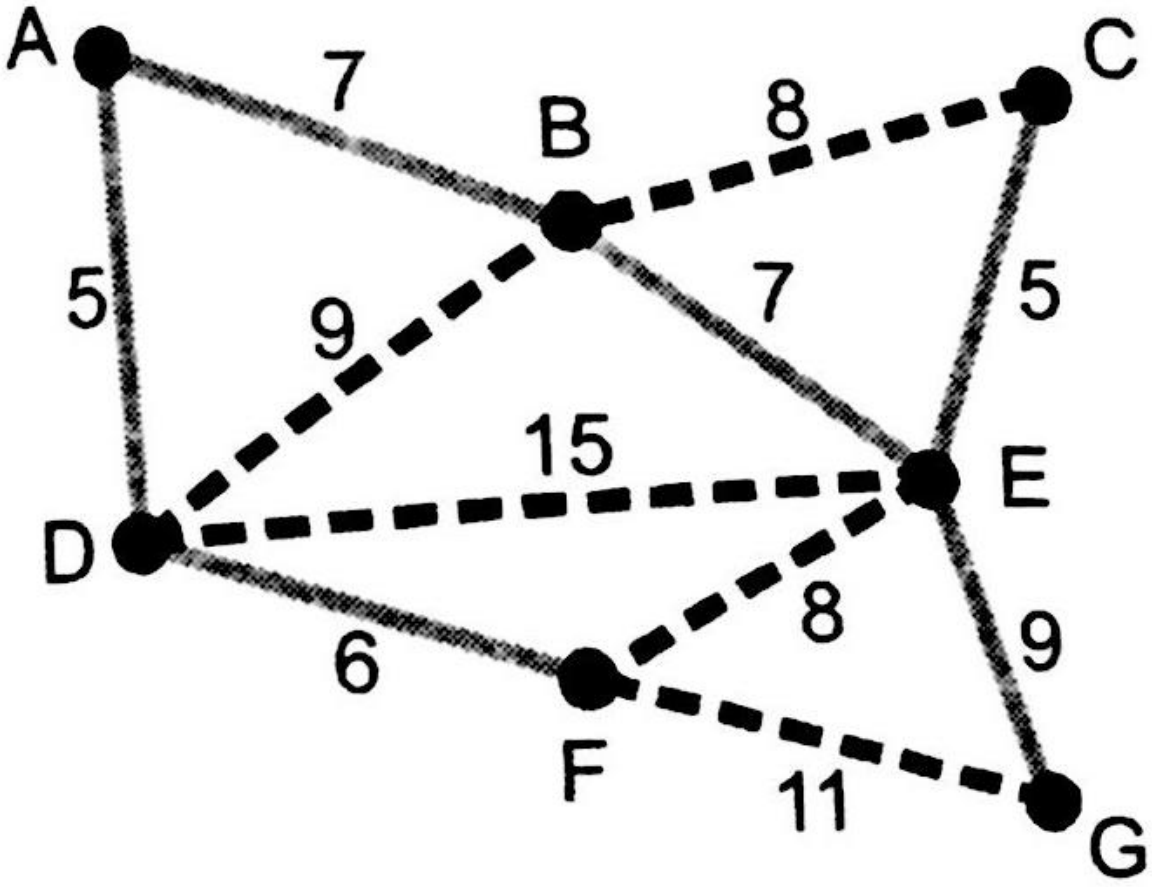
(n)


a
b
c
d
e
f
g
h
i
4
8
8
11
7
-
4
7
6
14
9
10
1
2

Task



Home task



- Prim's algorithm is significantly faster in the limit when you've got a really dense graph with many more edges than vertices. Kruskal performs better in typical situations (sparse graphs) because it uses simpler data structures.
- If we stop the algorithm in middle prim's algorithm always generates connected tree, but Kruskal on the other hand can give disconnected tree or forest
- The most basic form of Prim's algorithm only finds minimum spanning trees in connected graphs. However, running Prim's algorithm separately for each connected component of the graph