Minimum Spanning Tree

By Jonathan Davis
&
George Bebis, Analysis of Algorithms,
Univeristy of Nevade, Reno

Spanning Trees

Greedy algorithm: makes locally best choice/decision ignoring effect on future.

Tree: connected acyclic graph.

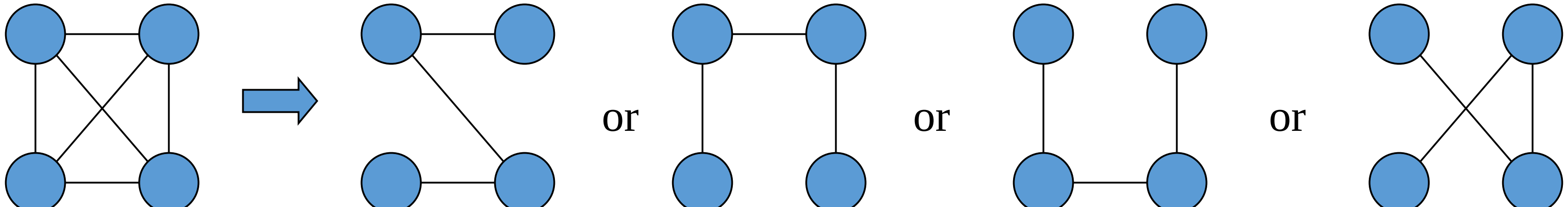
Spanning tree: a spanning tree of a graph G is a subset of edges of G that form a tree and reach all vertices of G.

A tree (i.e., connected, acyclic graph) which contains all the vertices of the graph.

A graph may have many spanning trees.

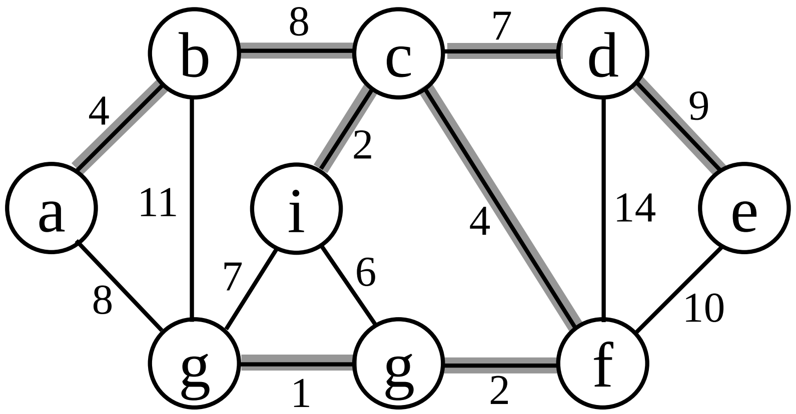
Graph A

Some Spanning Trees from Graph A



Spanning Trees

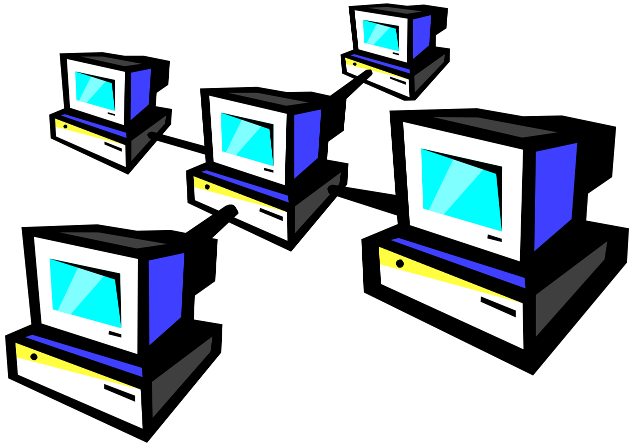
- Minimum Spanning Tree
  - Spanning tree with the **minimum sum of weights.**

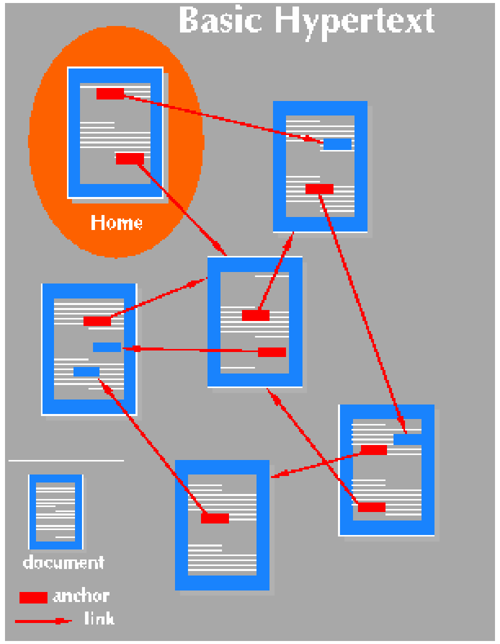


- Spanning forest
  - If a graph is not connected, then there is a spanning tree for each connected component of the graph

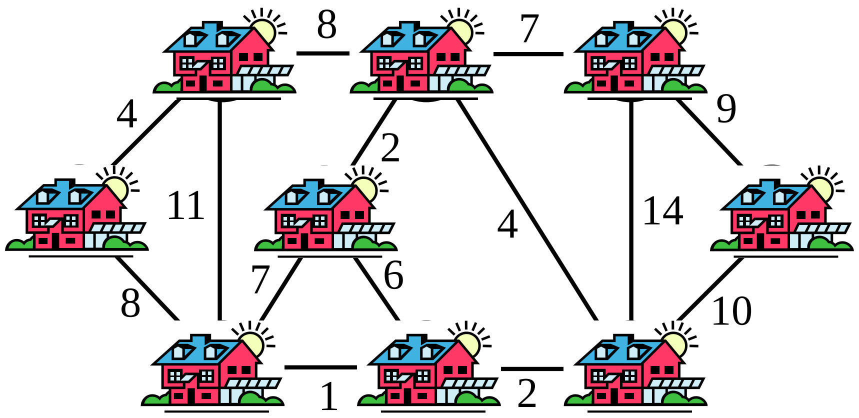
Applications to MST

– Find the least expensive way to connect a set of cities, terminals, computers, etc.





Example



Problem

- A town has a set of houses and a set of roads
- A road connects 2 and only 2 houses
- A road connecting houses u and v has a repair cost w(u, v)

Goal: Repair enough (and no more) roads such that:

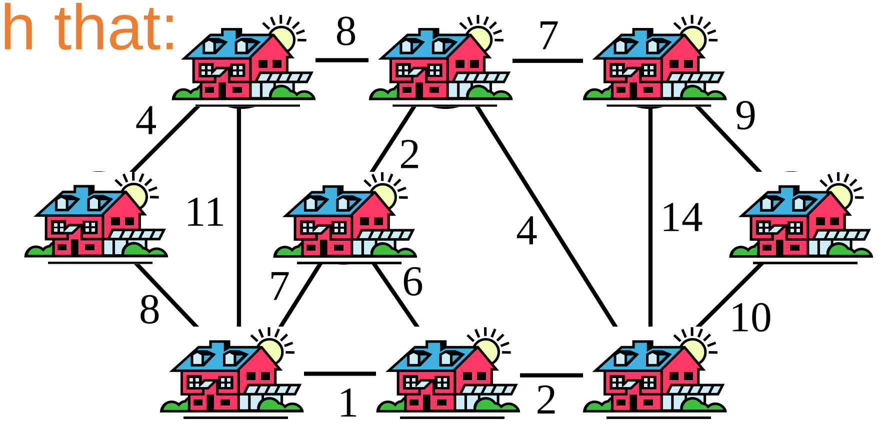
1. Everyone stays connected
   i.e., can reach every house from all other houses
2. Total repair cost is minimum

Minimum Spanning Trees

- A connected, undirected graph:
  - Vertices = houses, Edges = roads
- A **weight** $w(u, v)$ on each edge $(u, v) \in E$

Find a spanning tree $T \subseteq E$ such that:

1. T connects all vertices
2. $w(T) = \Sigma_{(u,v) \in T} w(u, v)$ is minimized

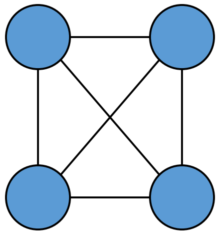


Properties of Minimum Spanning Trees

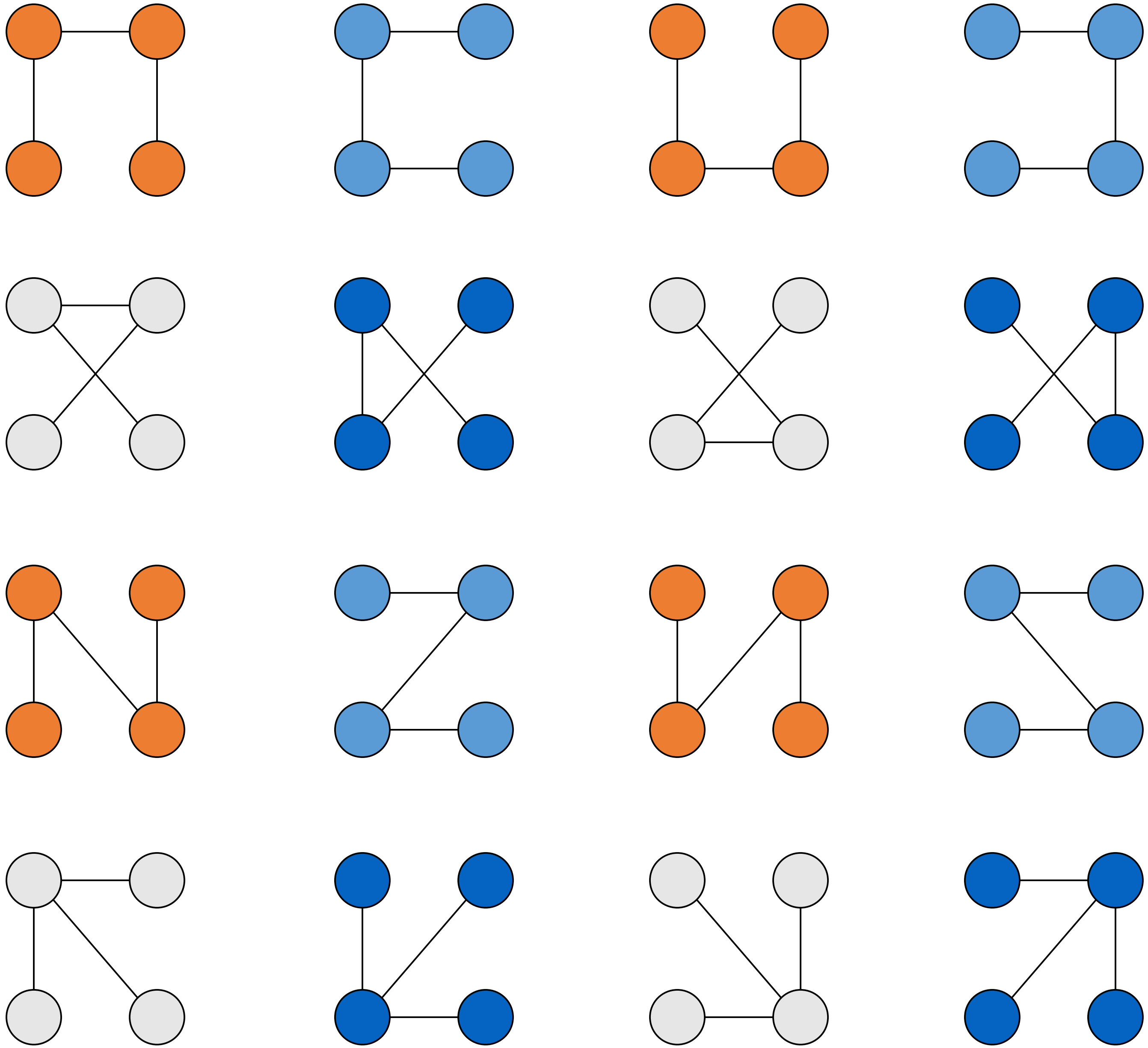
- Minimum spanning tree is **not** unique

- MST has no cycles – why?
  - We can take out an edge of a cycle, and still have the vertices connected while reducing the cost
- # of edges in a MST is |V| - 1

Complete Graph



All 16 of its Spanning Trees

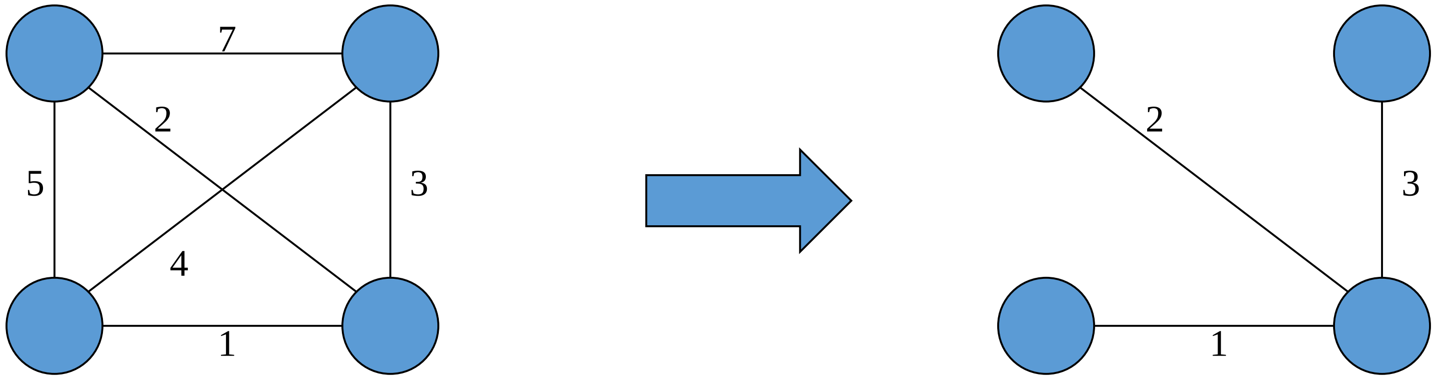


Minimum Spanning Trees

A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight

Complete Graph

Minimum Spanning Tree



Algorithms for Obtaining the Minimum Spanning Tree

- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm

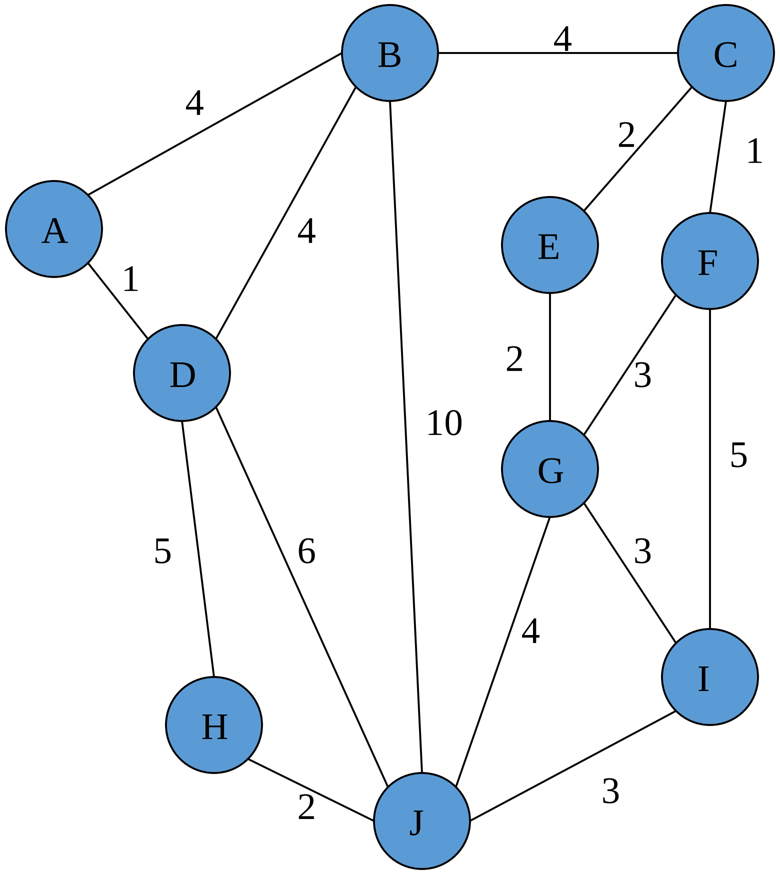
This algorithm starts with one node. It then, one by one, adds a node that is unconnected to the new graph to the new graph, each time selecting the node whose connecting edge has the smallest weight out of the available nodes’ connecting edges.

The steps are:

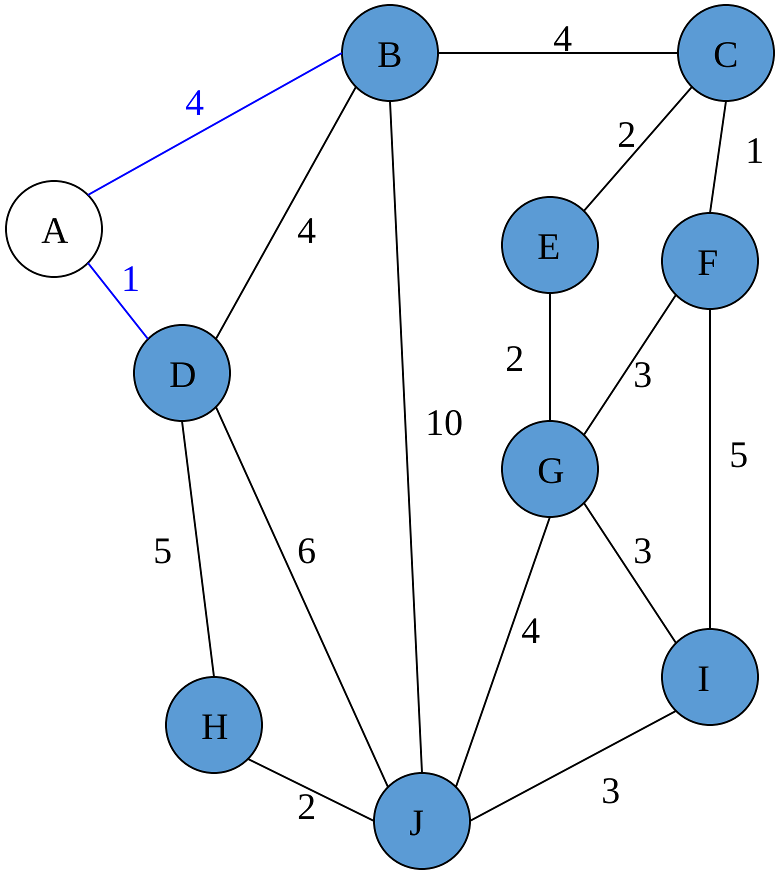
1. The new graph is constructed - with one node from the old graph.
2. While new graph has fewer than n nodes,
    1. Find the node from the old graph with the smallest connecting edge to the new graph,
    2. Add it to the new graph

Every step will have joined one node, so that at the end we will have one graph with all the nodes and it will be a minimum spanning tree of the original graph.

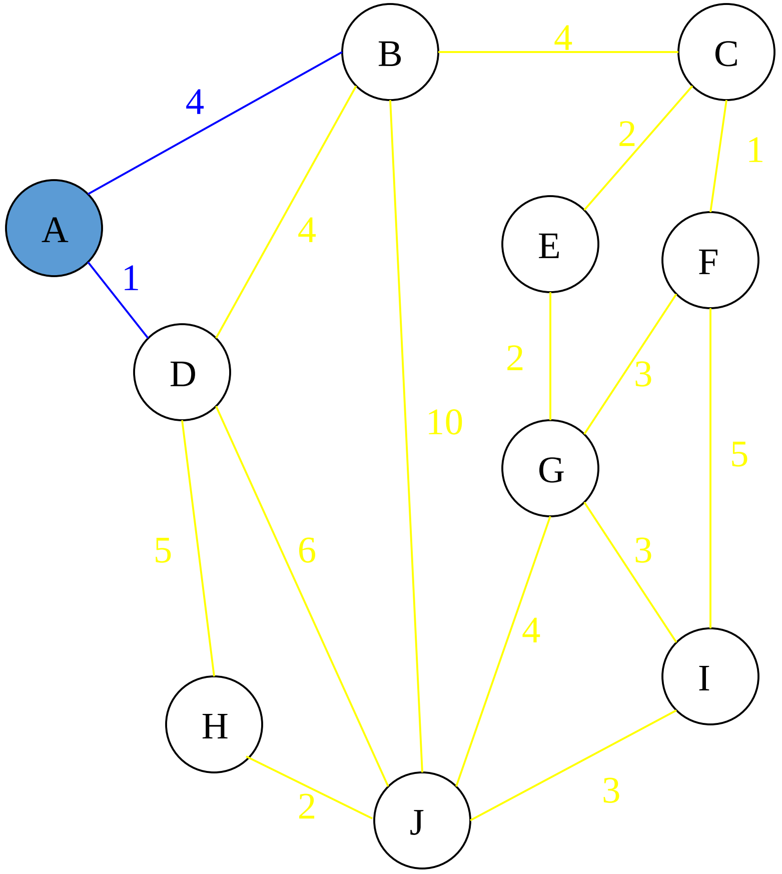
Complete Graph



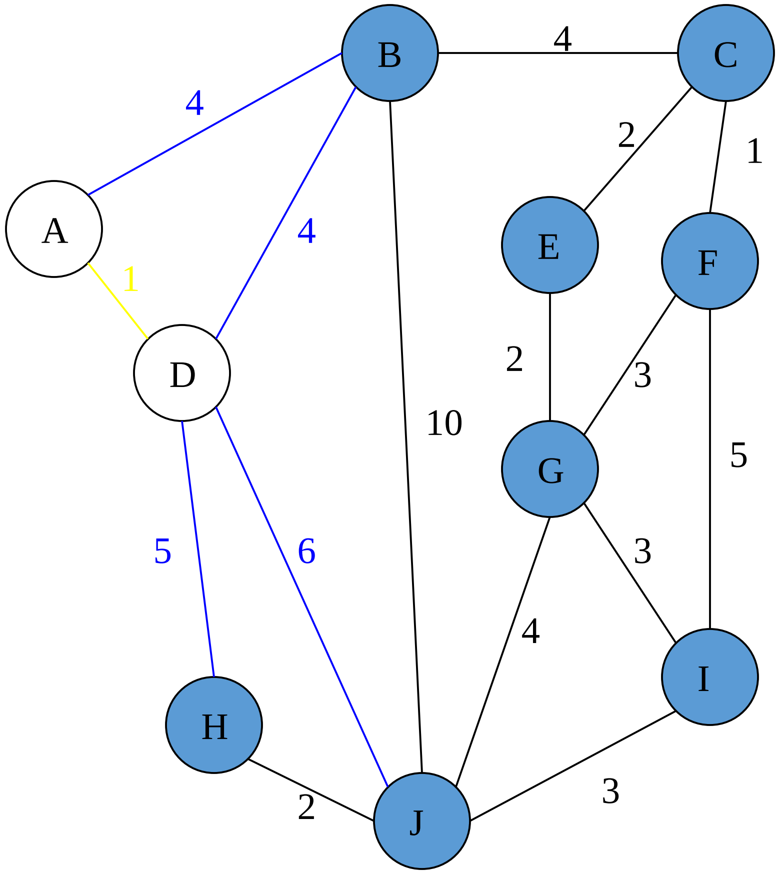
Old Graph



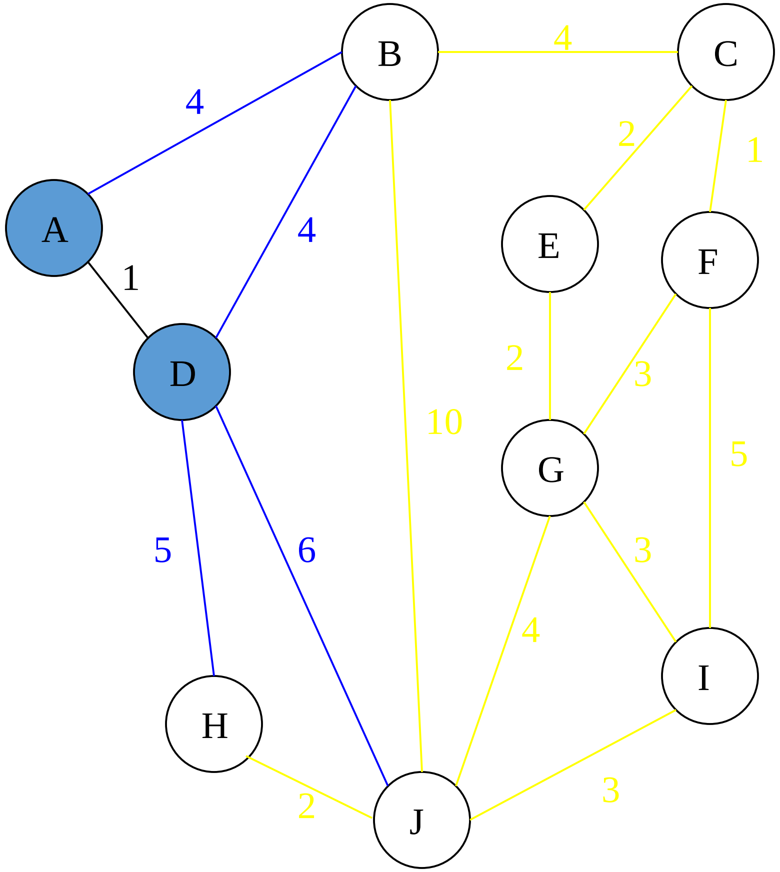
New Graph



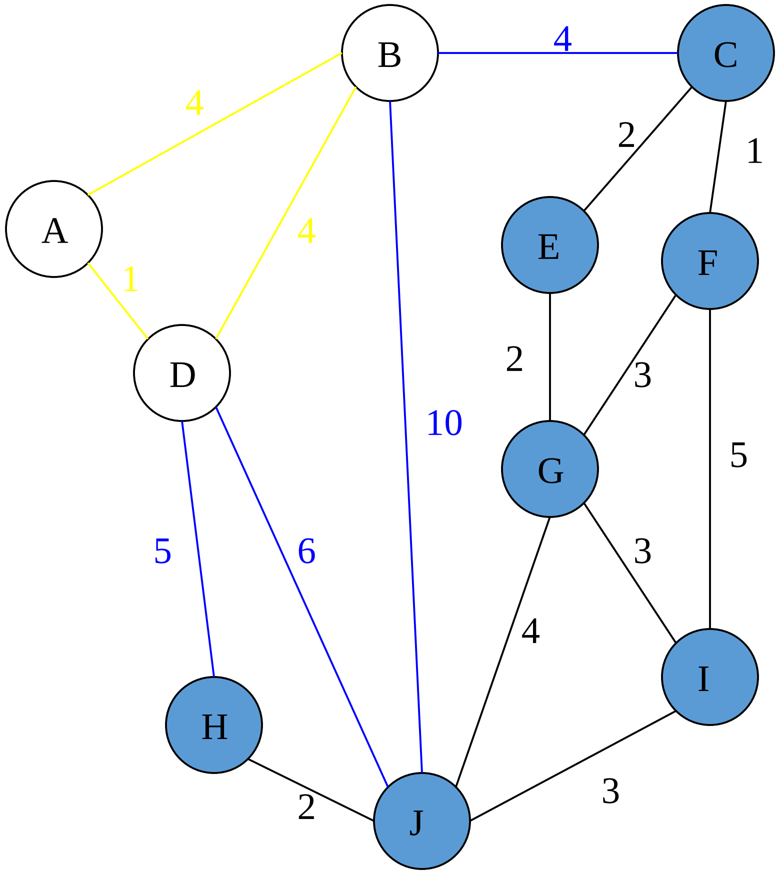
Old Graph



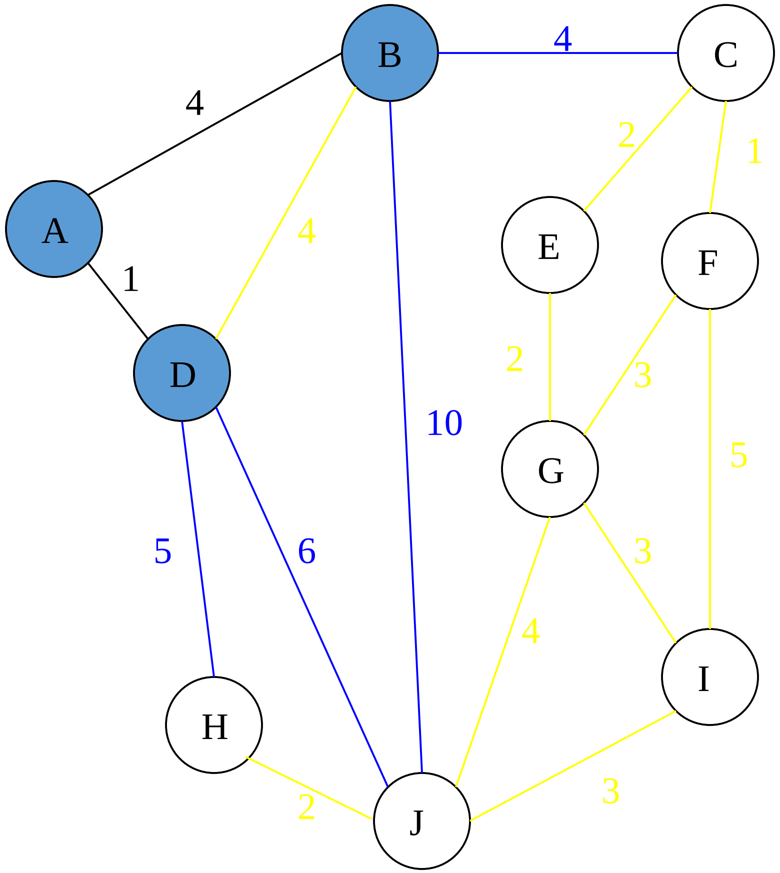
New Graph



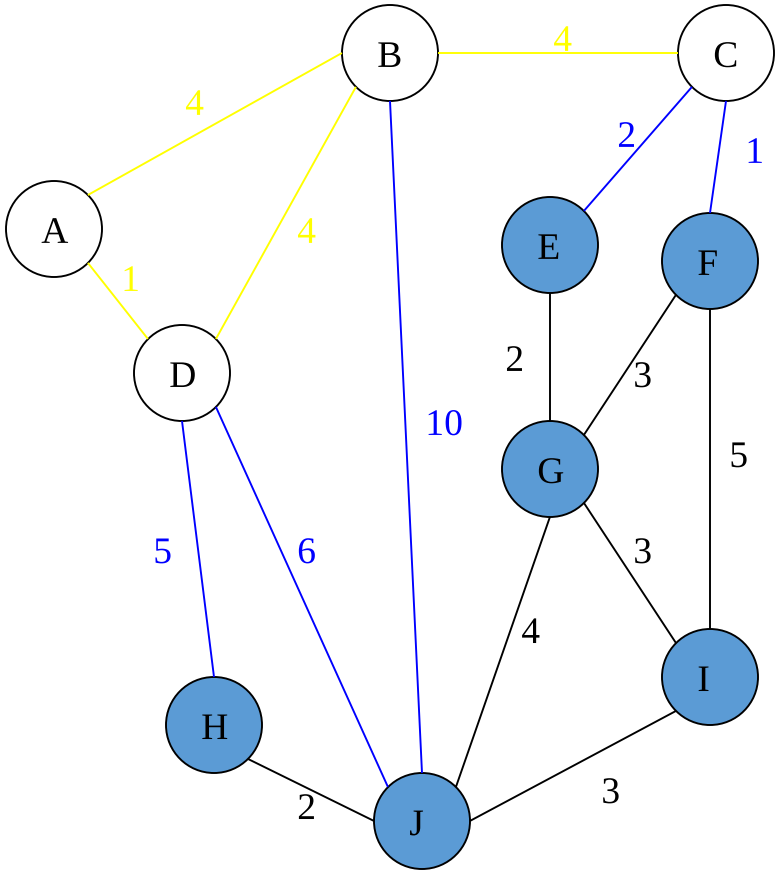
Old Graph



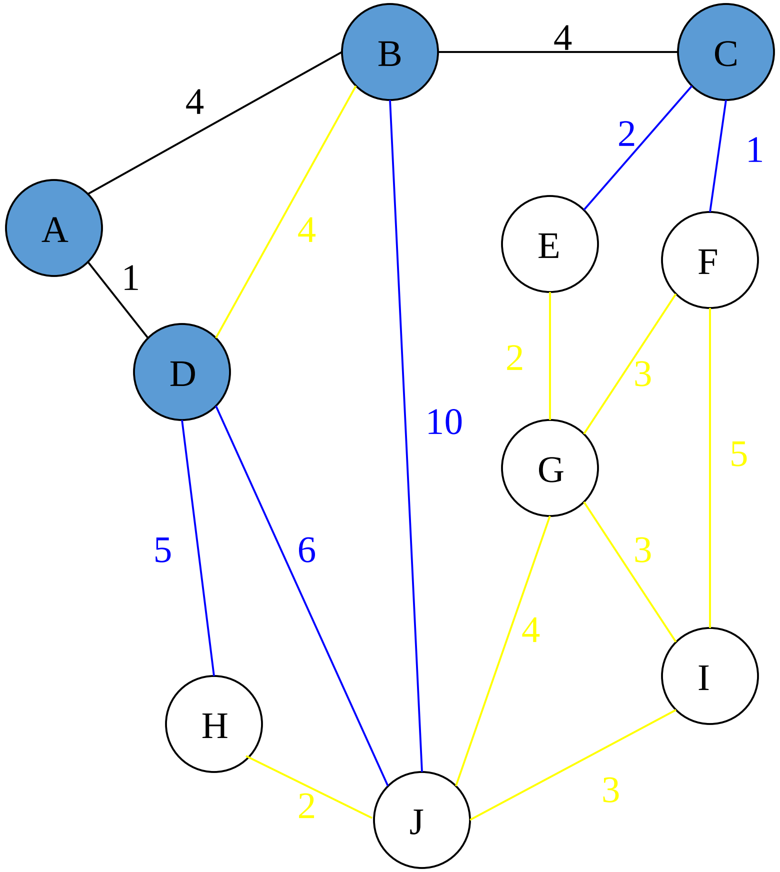
New Graph



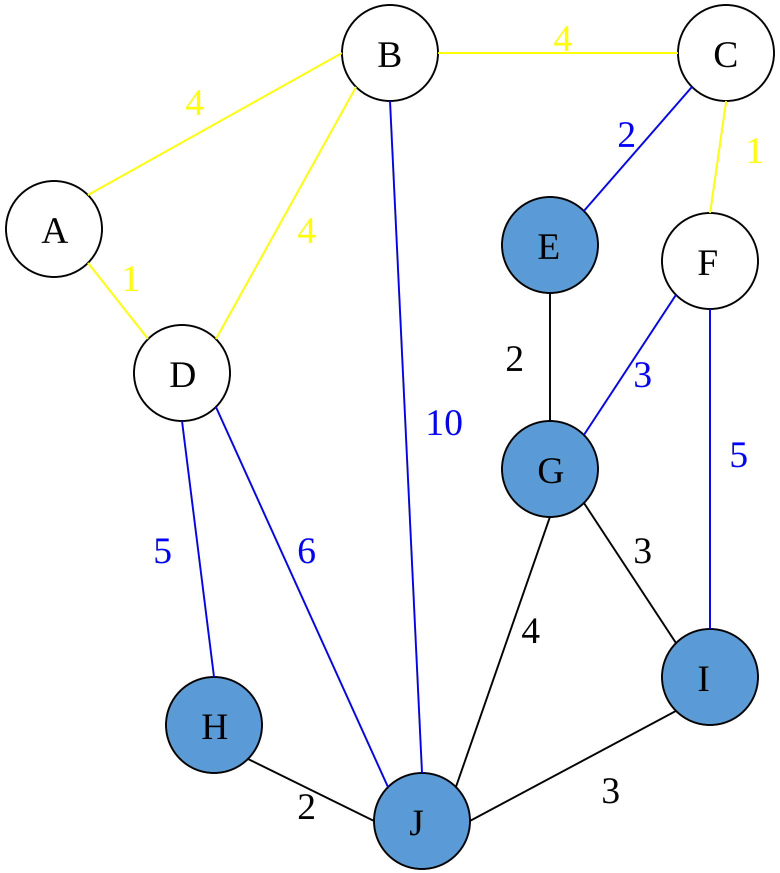
Old Graph



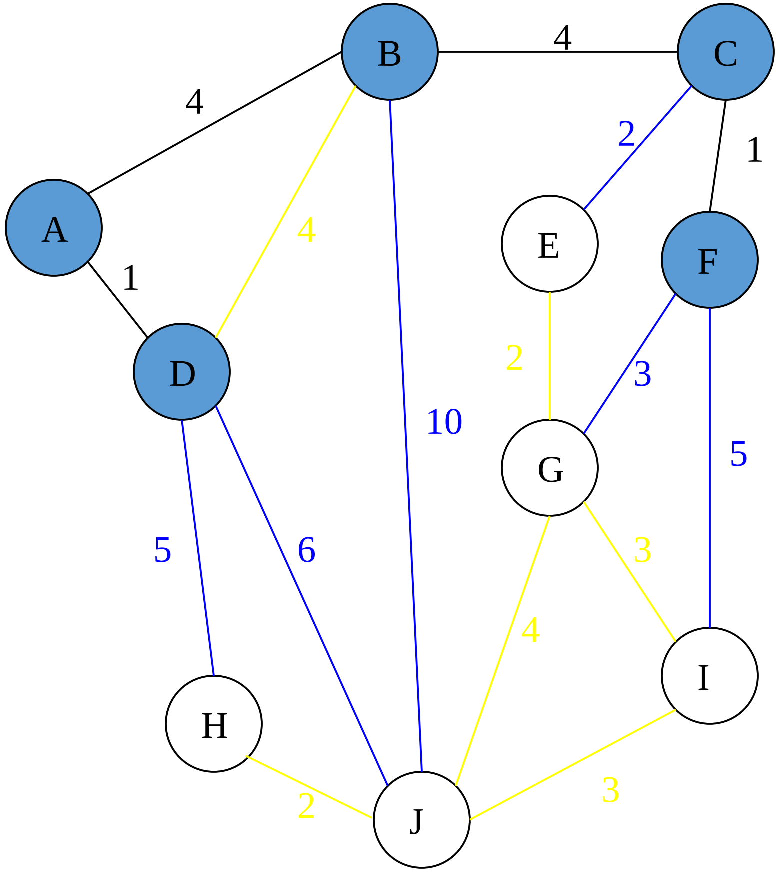
New Graph

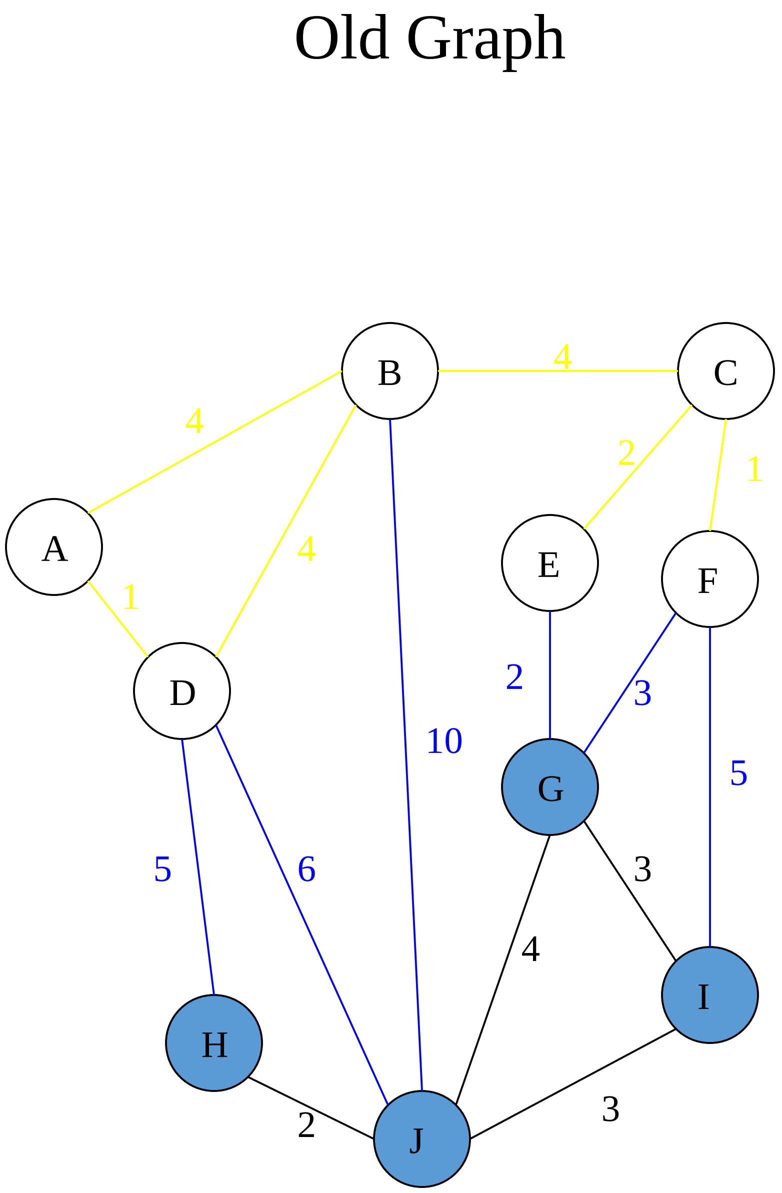


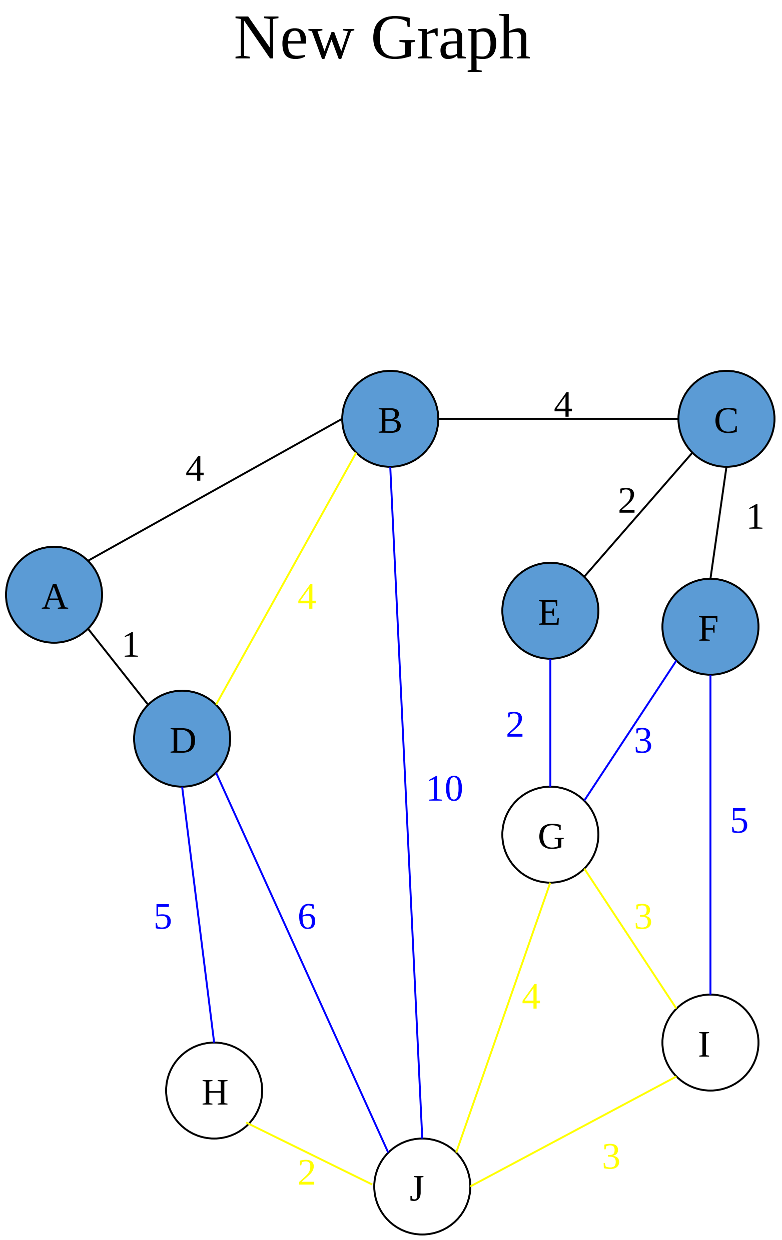
Old Graph



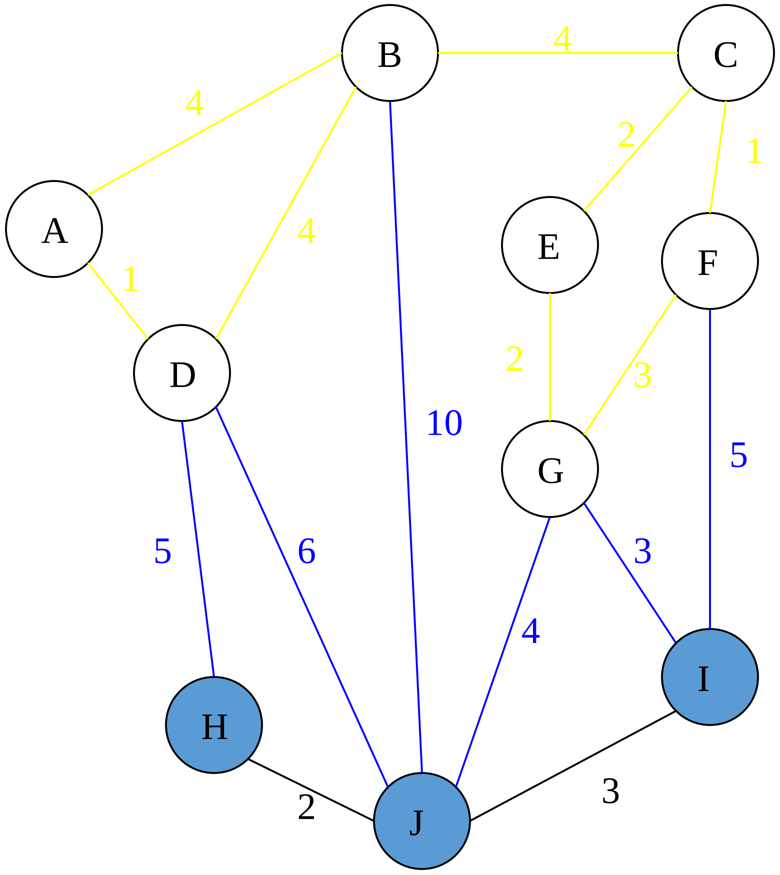
New Graph



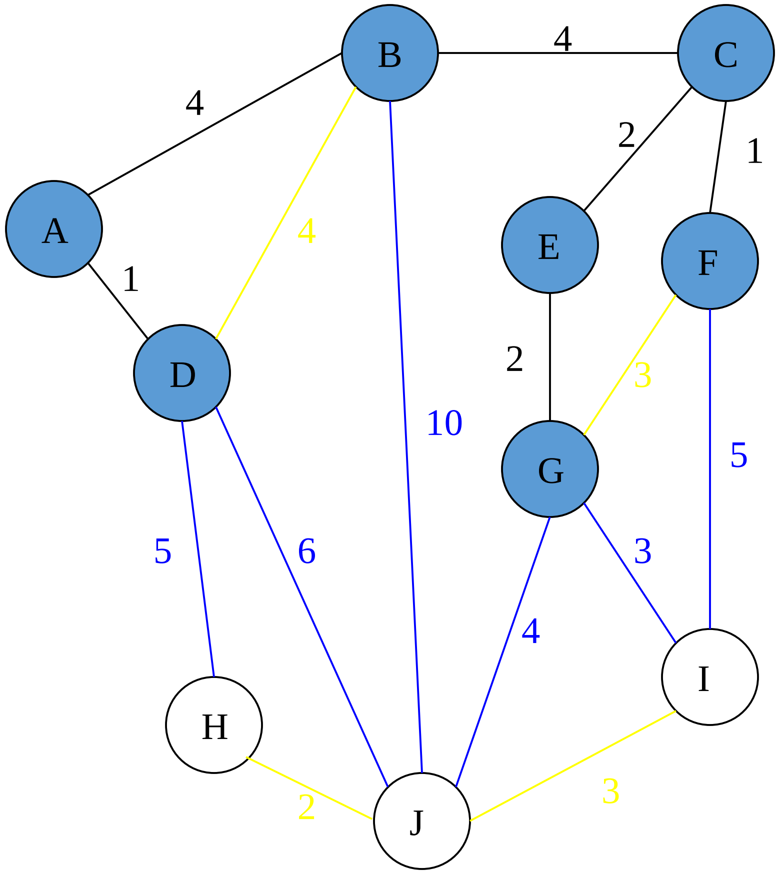

Old Graph
A
B
C
D
E
F
G
H
I
J
4
4
2
1
1
4
2
3
10
5
5
6
3
4
2
3


New Graph
A
B
C
D
E
F
G
H
I
J
4
4
2
1
1
4
2
3
10
5
5
6
3
4
2
3

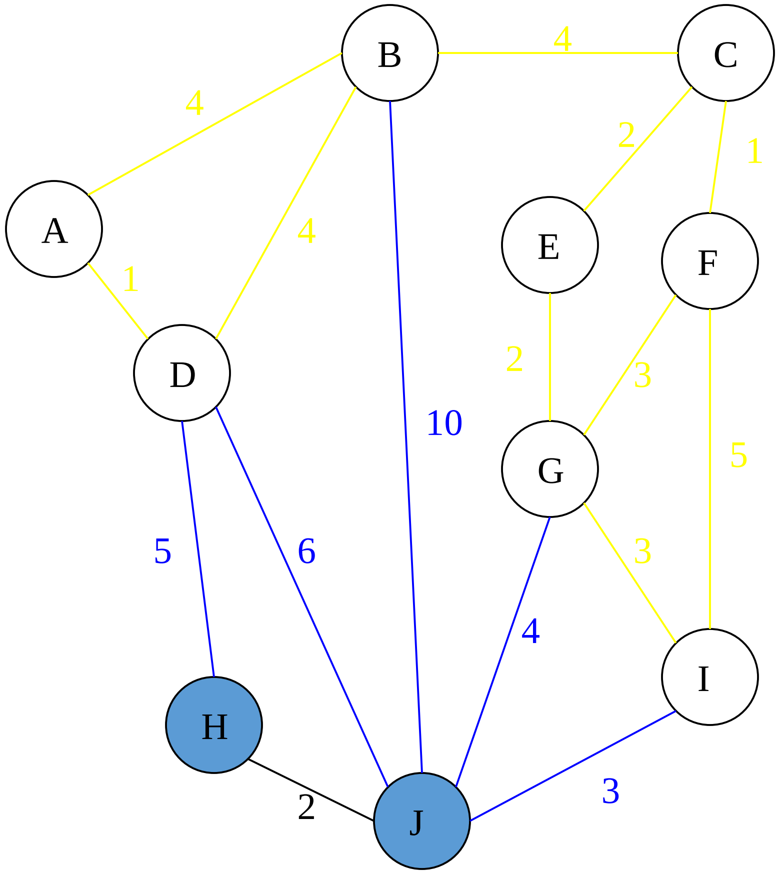
Old Graph



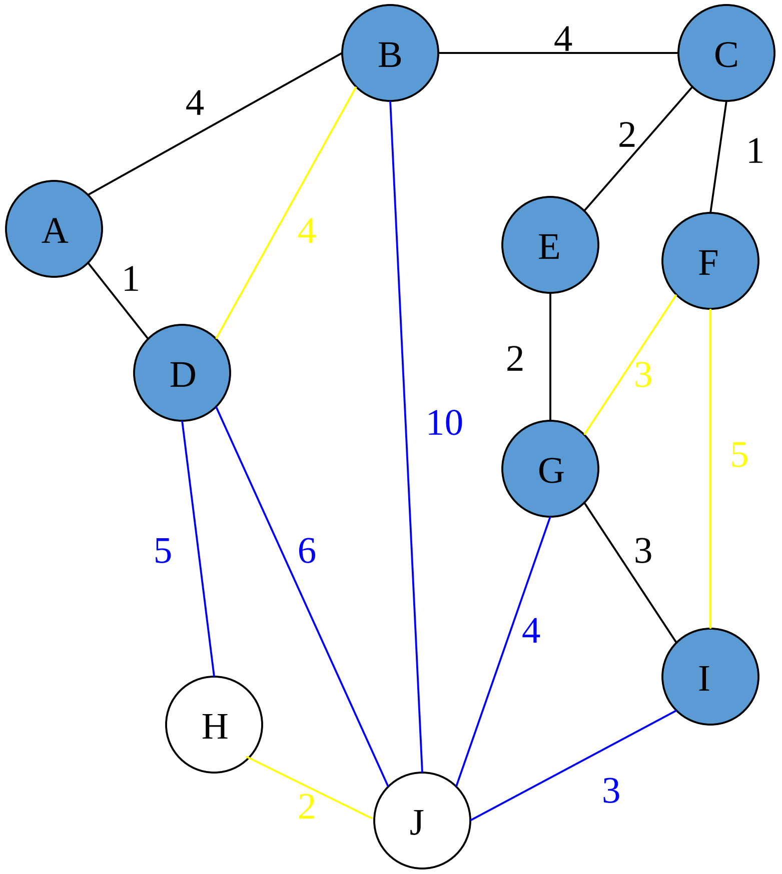
New Graph



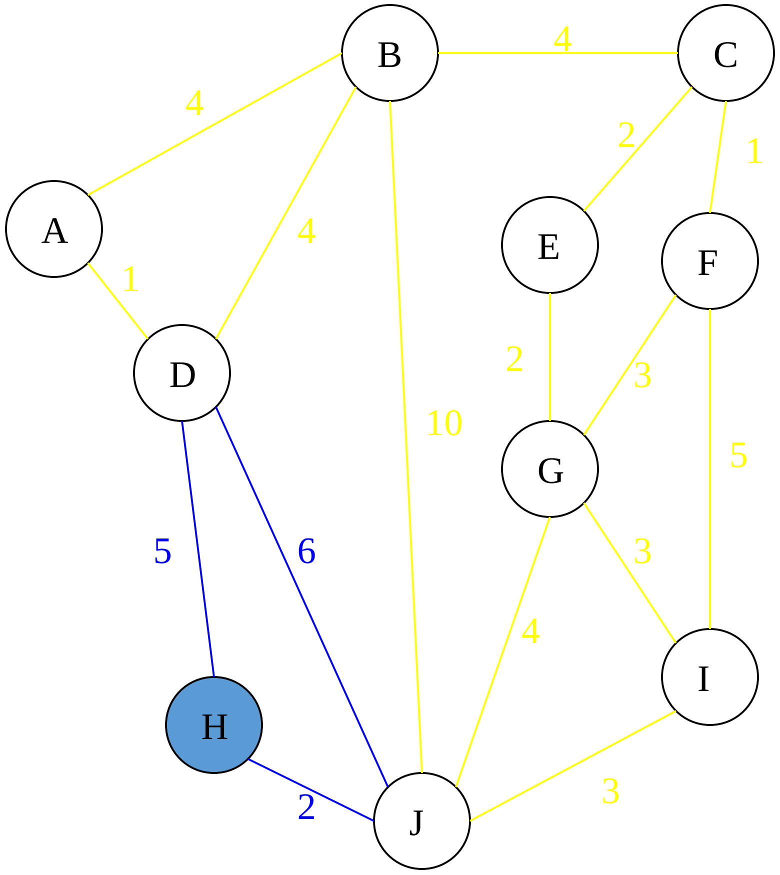
Old Graph



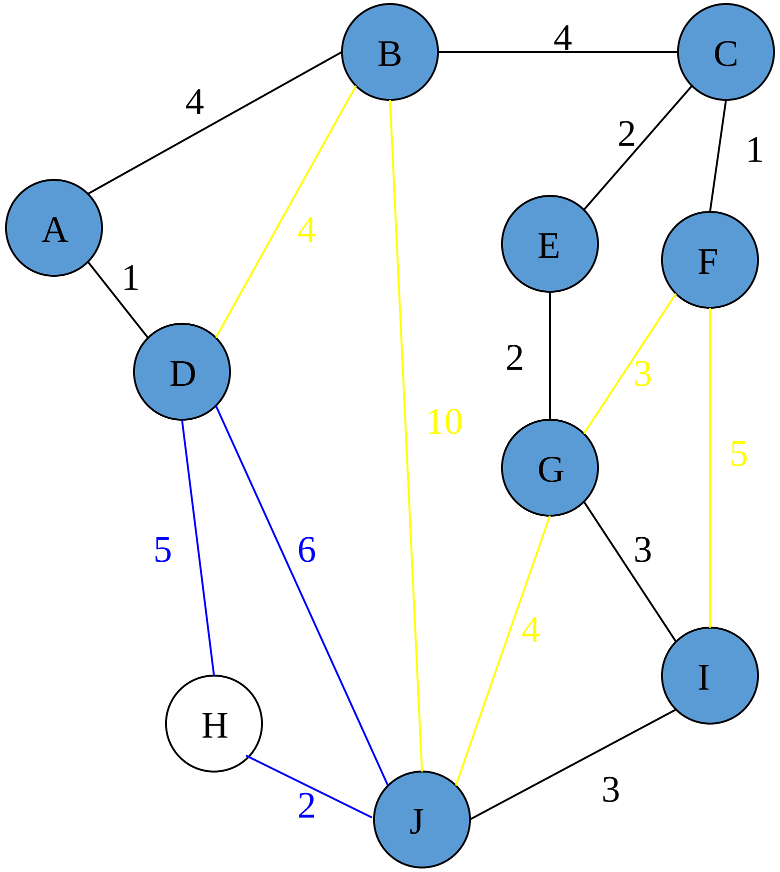
New Graph



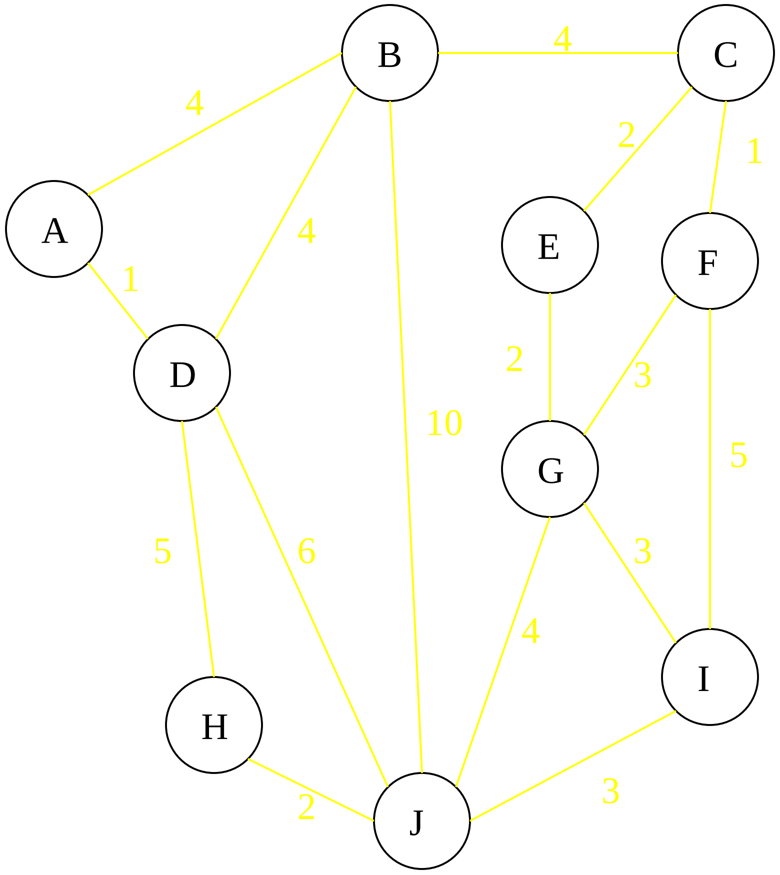
Old Graph



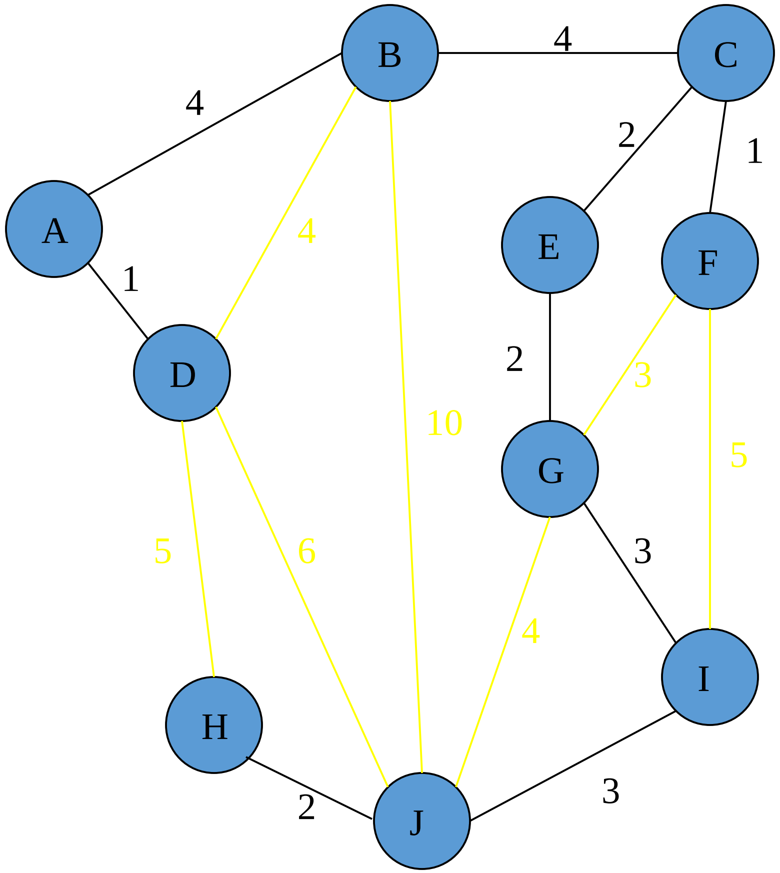
New Graph



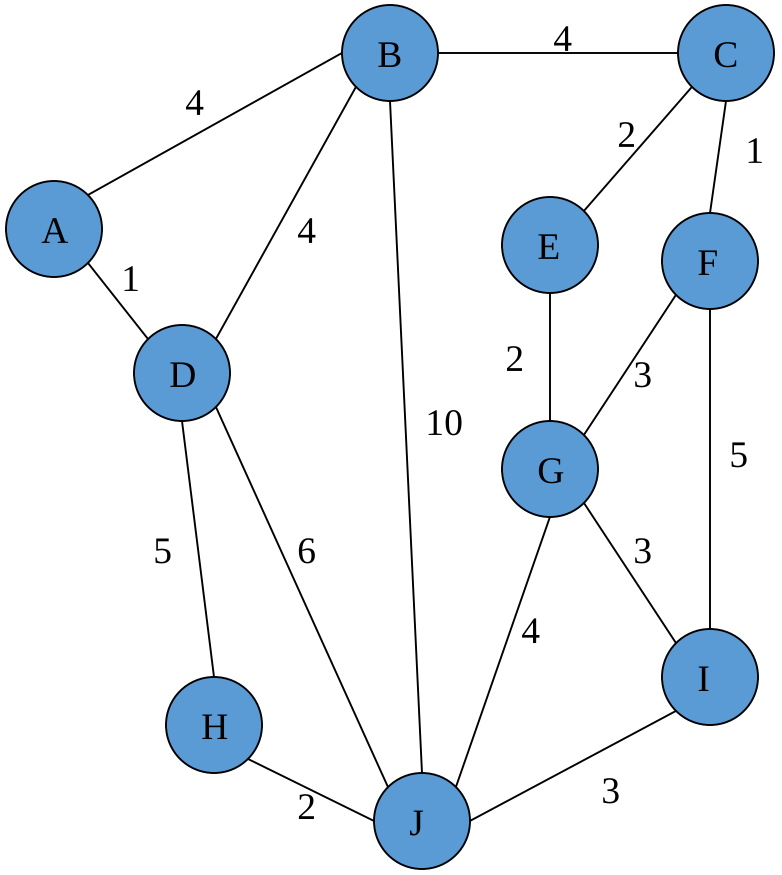
Old Graph



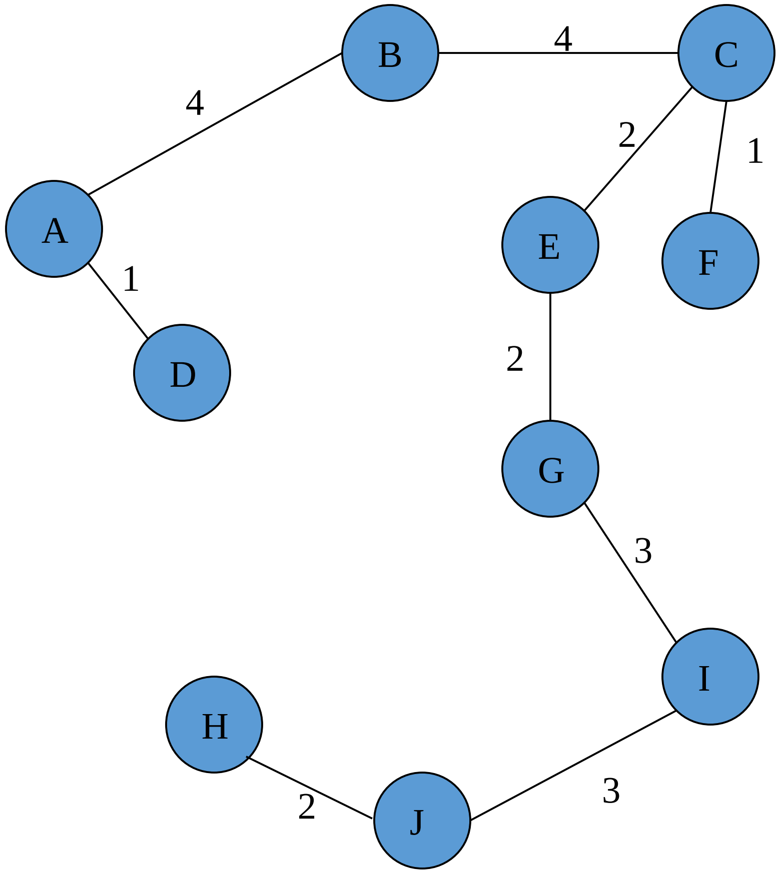
New Graph



Complete Graph



Minimum Spanning Tree



Analysis of Prim's Algorithm

Running Time = O(e + v log v)   (e = edges, v = nodes)

If a heap is not used, the run time will be O(n^2) instead of O(e + v log v). However, using a heap complicates the code since you’re complicating the data structure.

Unlike Kruskal’s, it doesn’t need to see all of the graph at once. It can deal with it one piece at a time. It also doesn’t need to worry if adding an edge will create a cycle since this algorithm deals primarily with the nodes, and not the edges.

For this algorithm the number of nodes needs to be kept to a minimum in addition to the number of edges. For small graphs, the edges matter more, while for large graphs the number of nodes matters more.

Kruskal's Algorithm

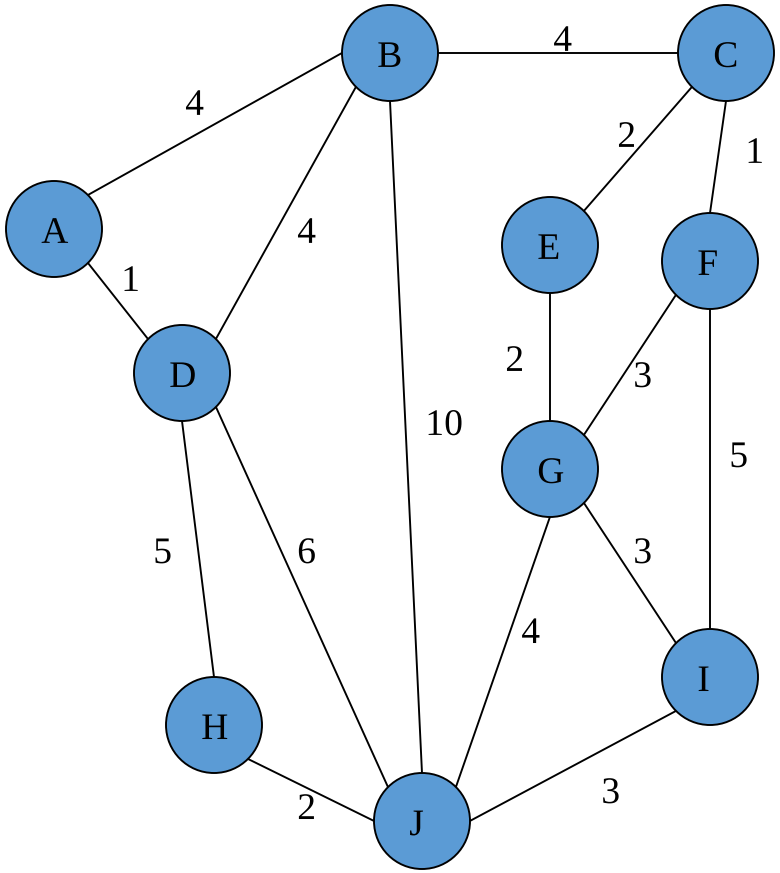
This algorithm creates a forest of trees. Initially the forest consists of n single node trees (and no edges). At each step, we add one edge (the cheapest one) so that it joins two trees together. If it were to form a cycle, it would simply link two nodes that were already part of a single connected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

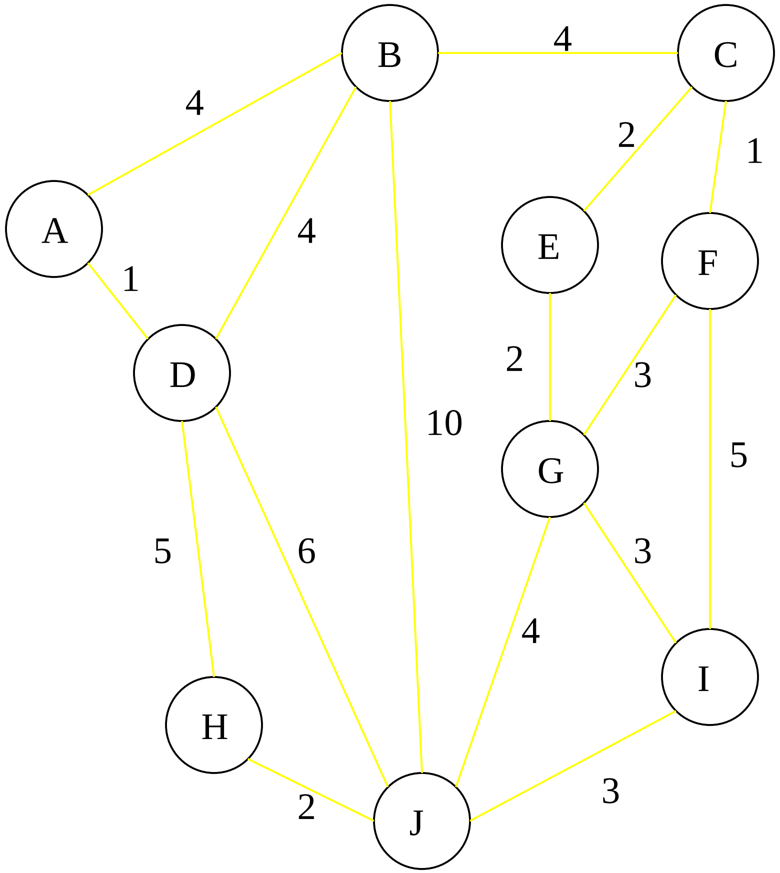
The steps are:

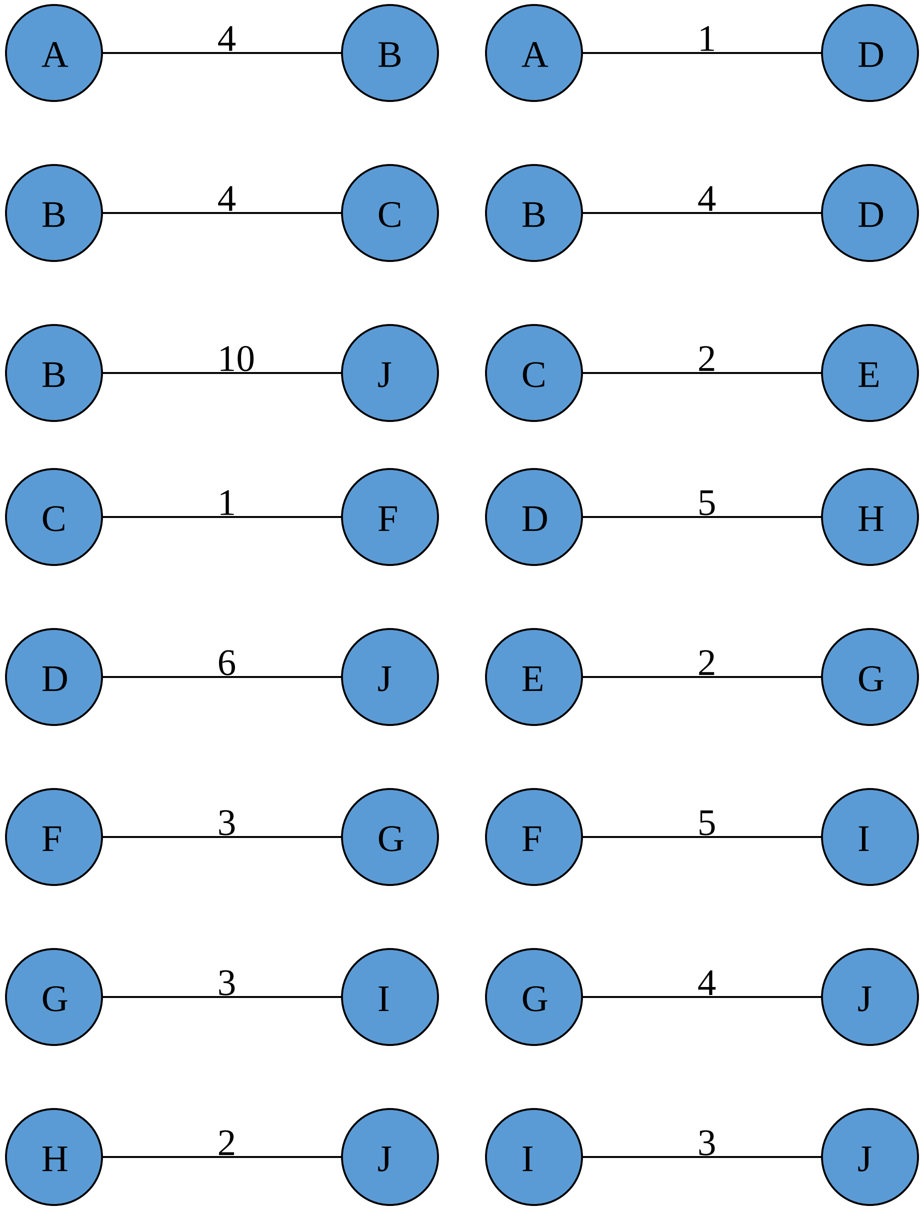
1. The forest is constructed - with each node in a separate tree.
2. The edges are placed in a priority queue.
3. Until we've added n-1 edges,
    1. Extract the cheapest edge from the queue,
    2. If it forms a cycle, reject it,
    3. Else add it to the forest. Adding it to the forest will join two trees together.

Every step will have joined two trees in the forest together, so that at the end, there will only be one tree in T.

Complete Graph

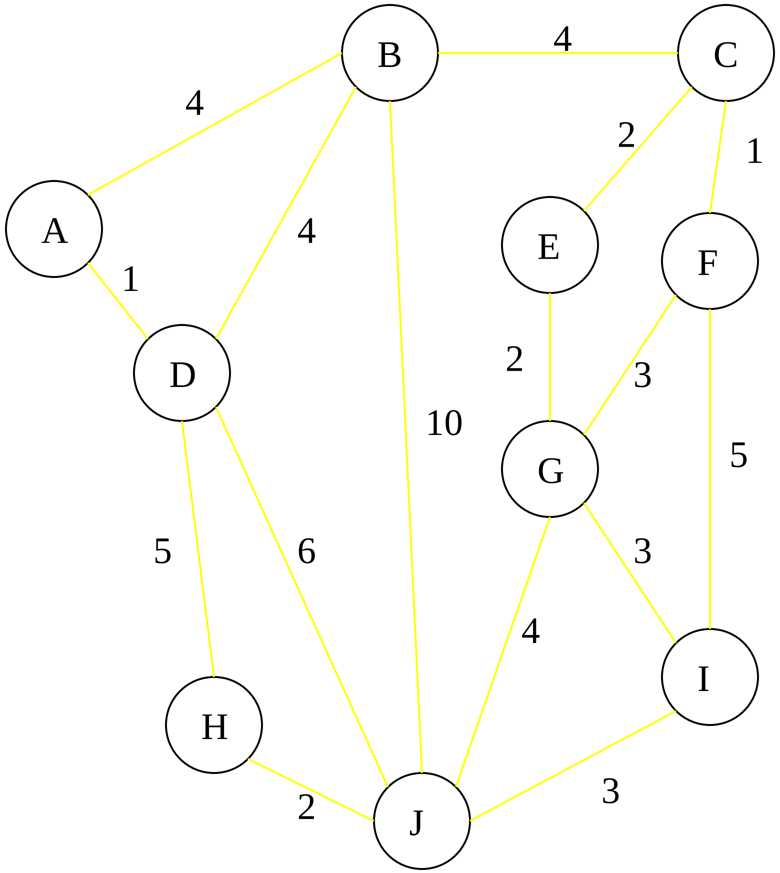


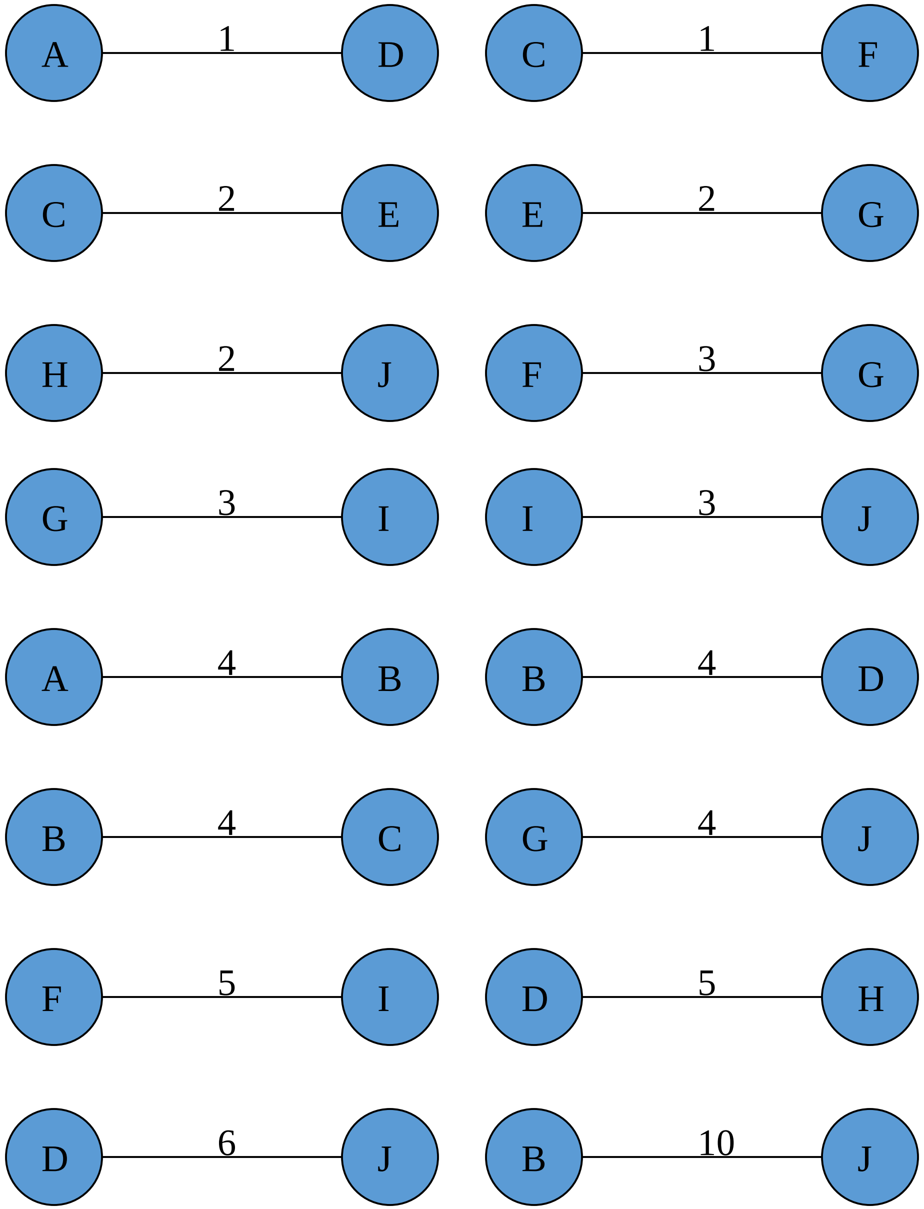

B
C
4
4
2
1
A
E
F
4
1
D
2
3
10
5
G
5
6
3
4
I
H
J
2
3


A 4 B
A 1 D
B 4 C
B 4 D
B 10 J
C 2 E
C 1 F
D 5 H
D 6 J
E 2 G
F 3 G
F 5 I
G 3 I
G 4 J
H 2 J
I 3 J

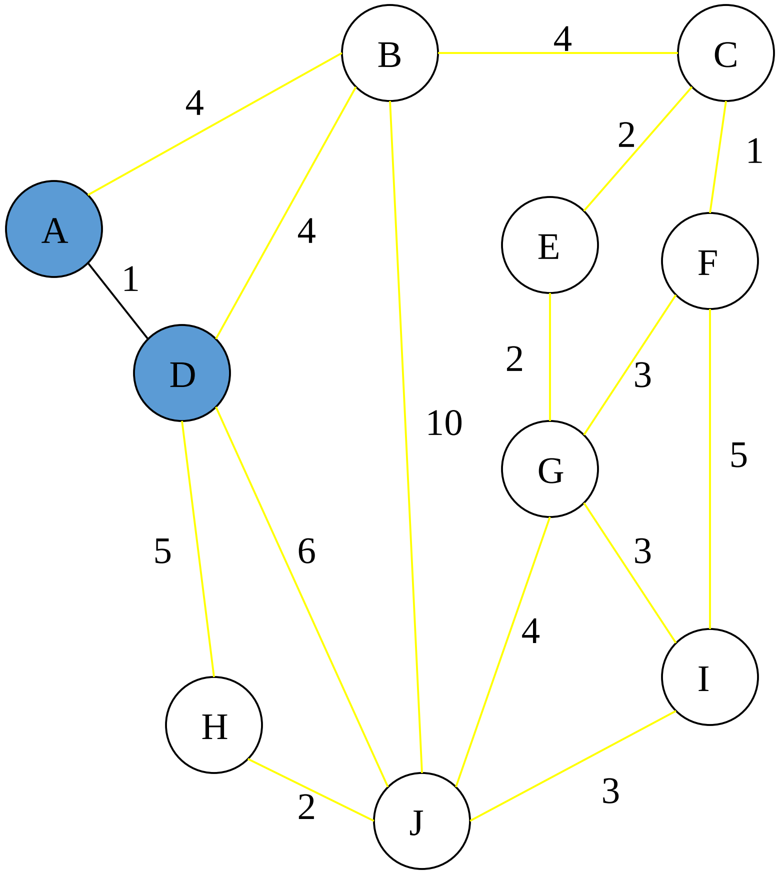
Sort Edges

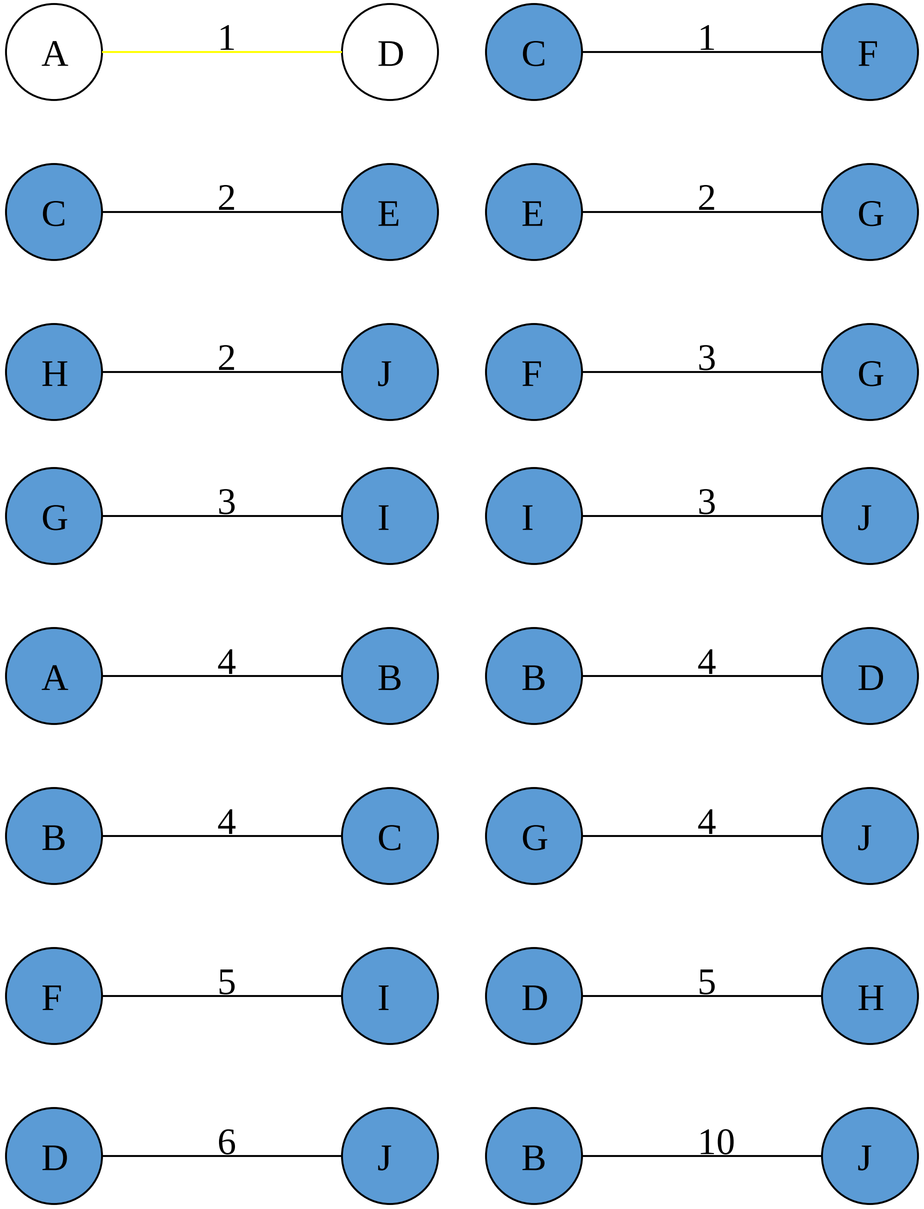
(in reality they are placed in a priority queue - not sorted - but sorting them makes the algorithm easier to visualize)



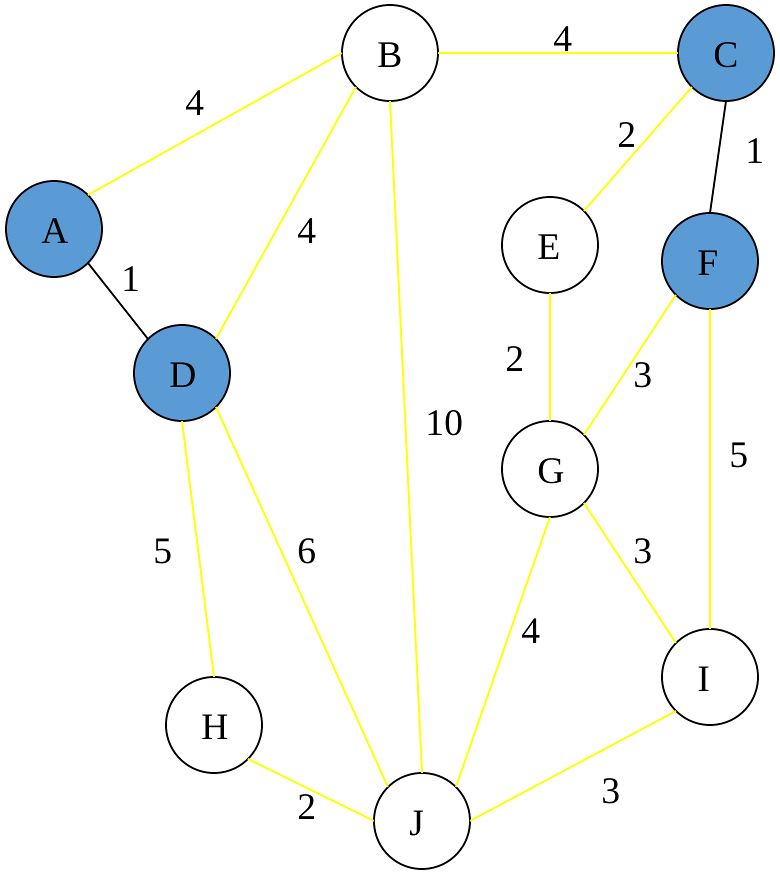


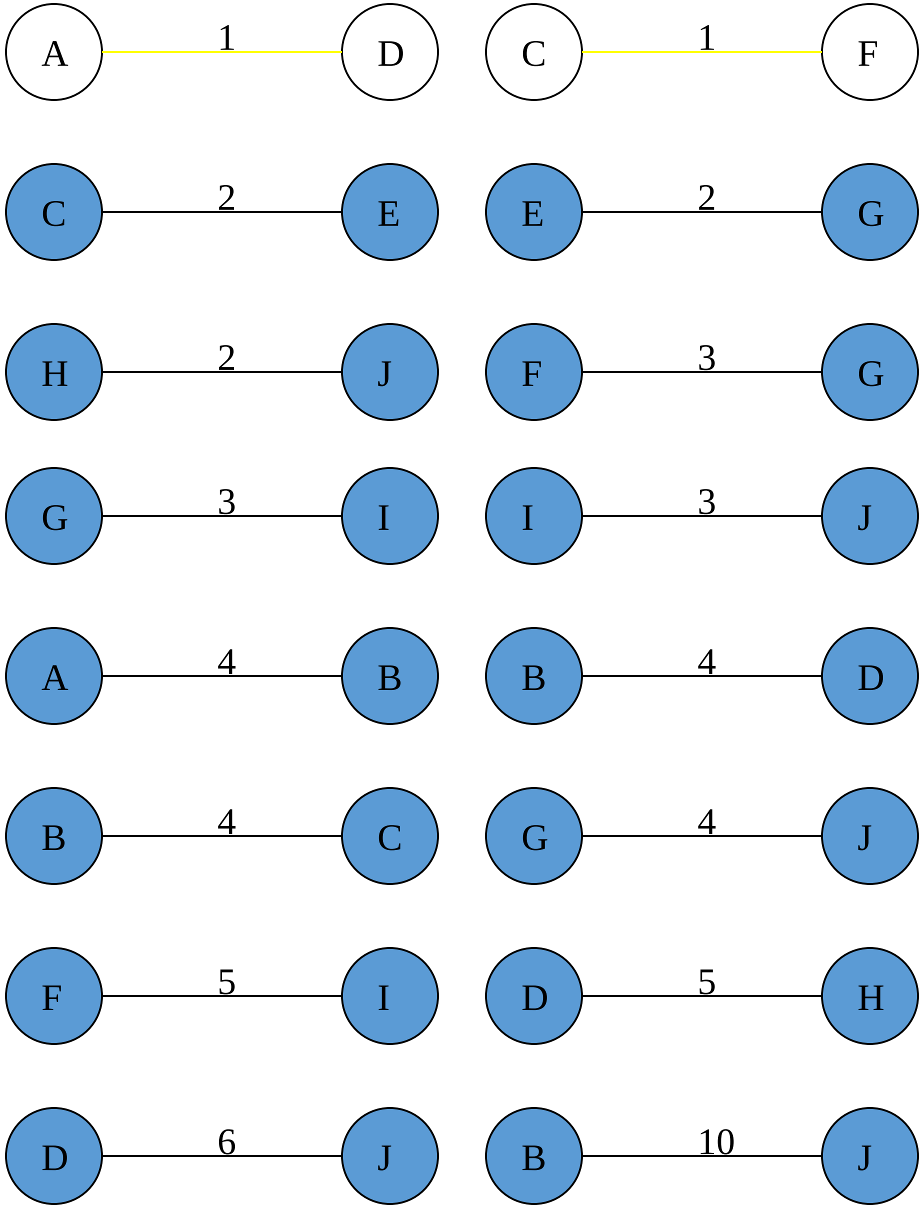
Add Edge


A
B
C
D
E
F
G
H
I
J
4
4
2
1
1
4
10
2
3
5
5
6
3
4
2
3

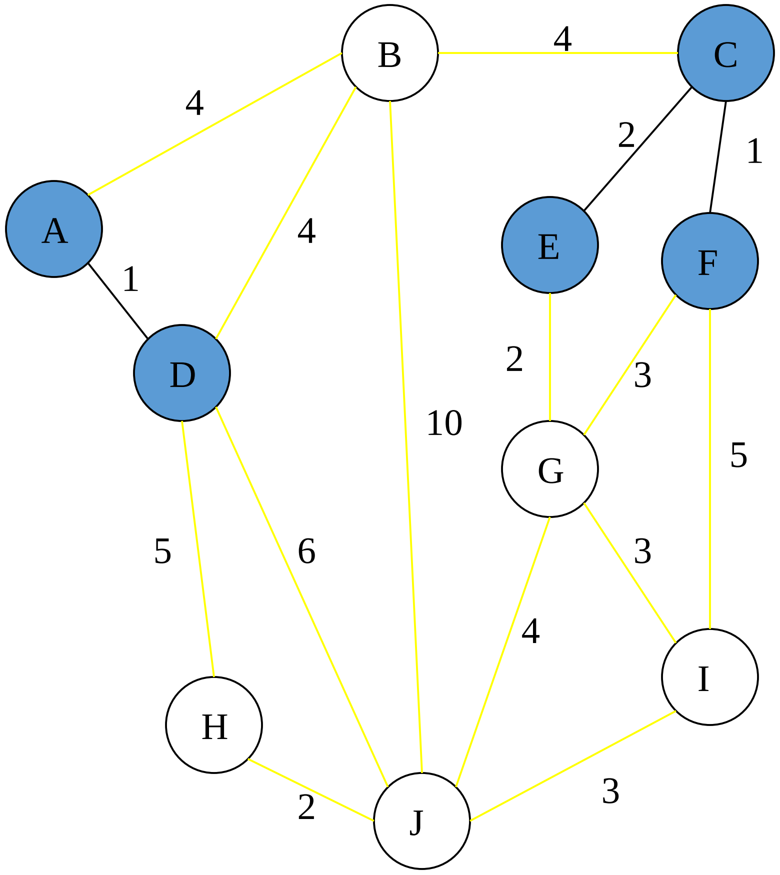

A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

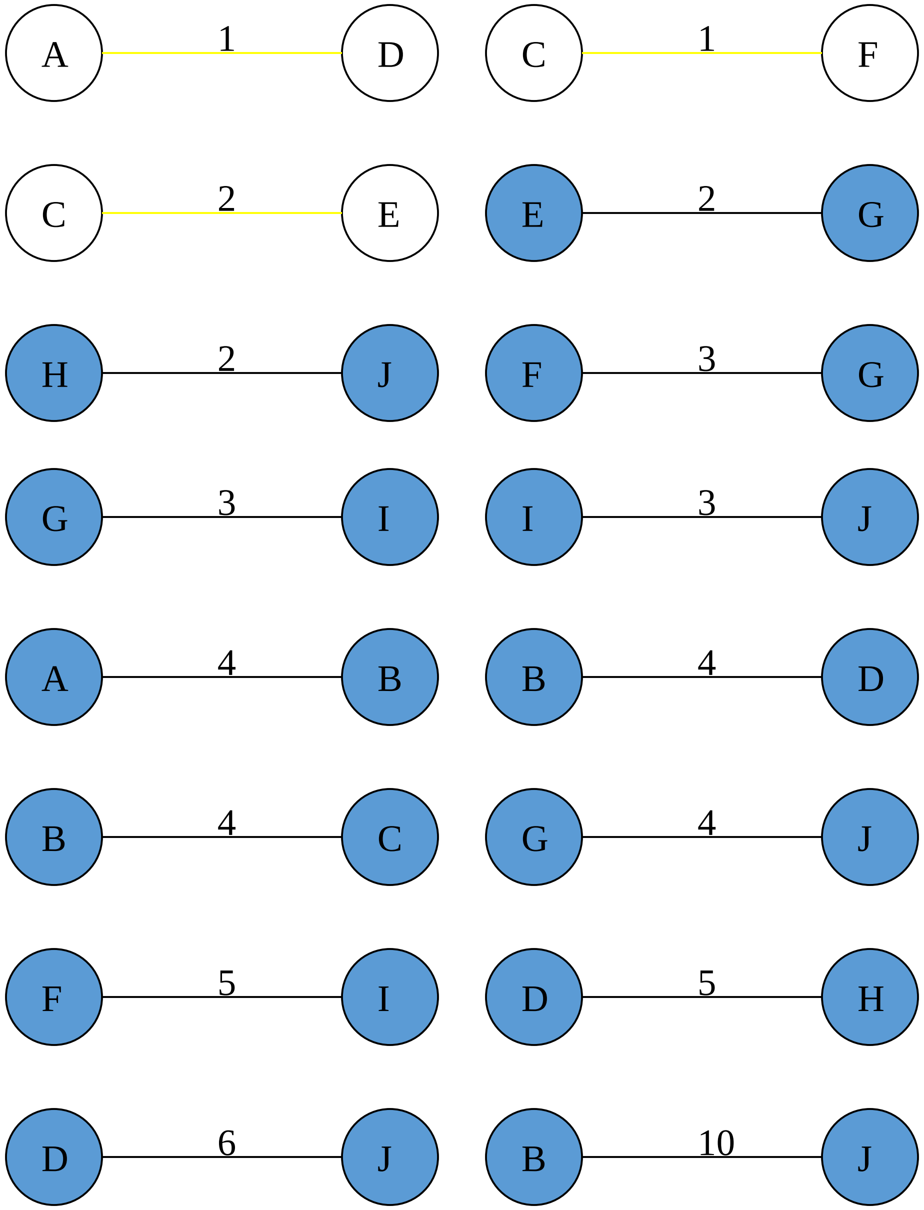
Add Edge


A
B
C
D
E
F
G
H
I
J
4
4
4
1
2
1
2
3
10
5
6
5
4
3
2
3

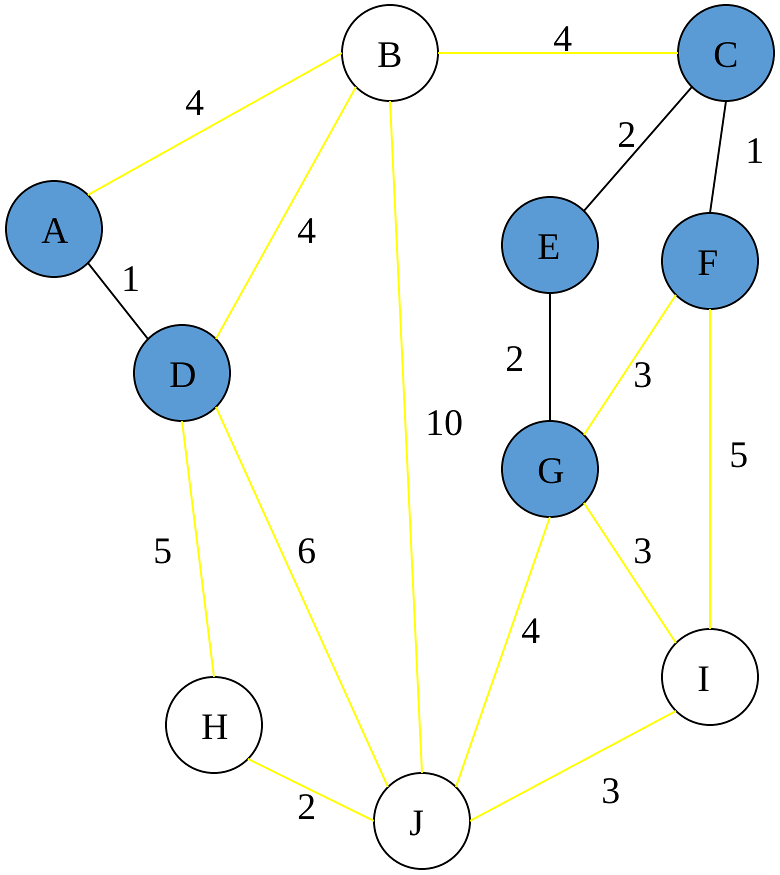

A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

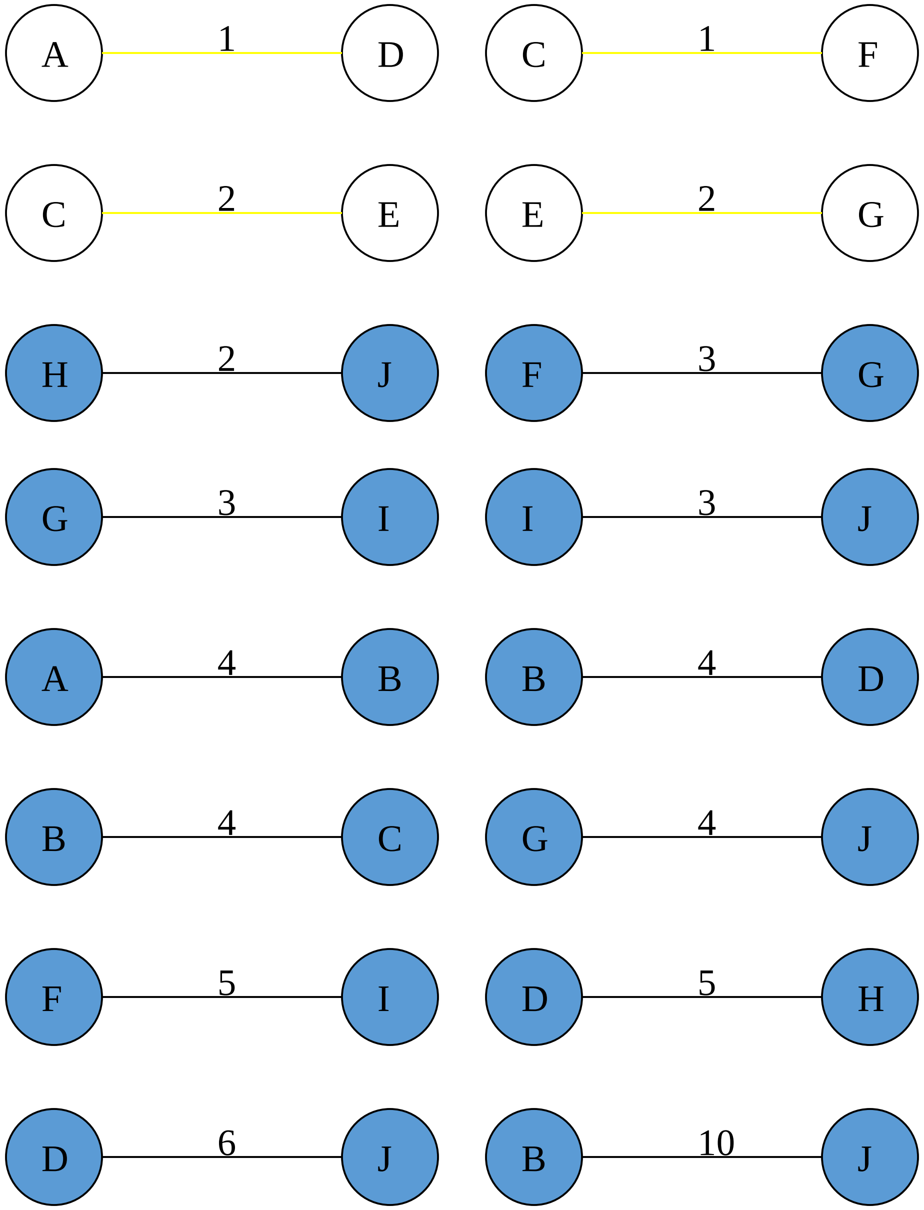
Add Edge


B
4
C
4
2
1
A
4
E
F
1
D
2
3
10
5
G
5
6
3
4
I
H
2
J
3

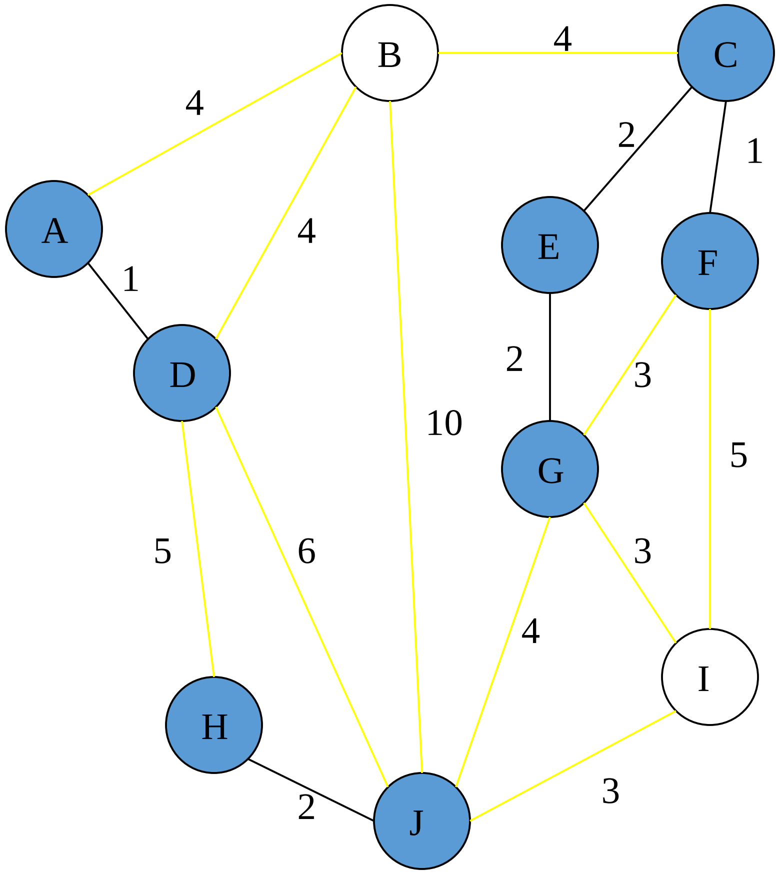

A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

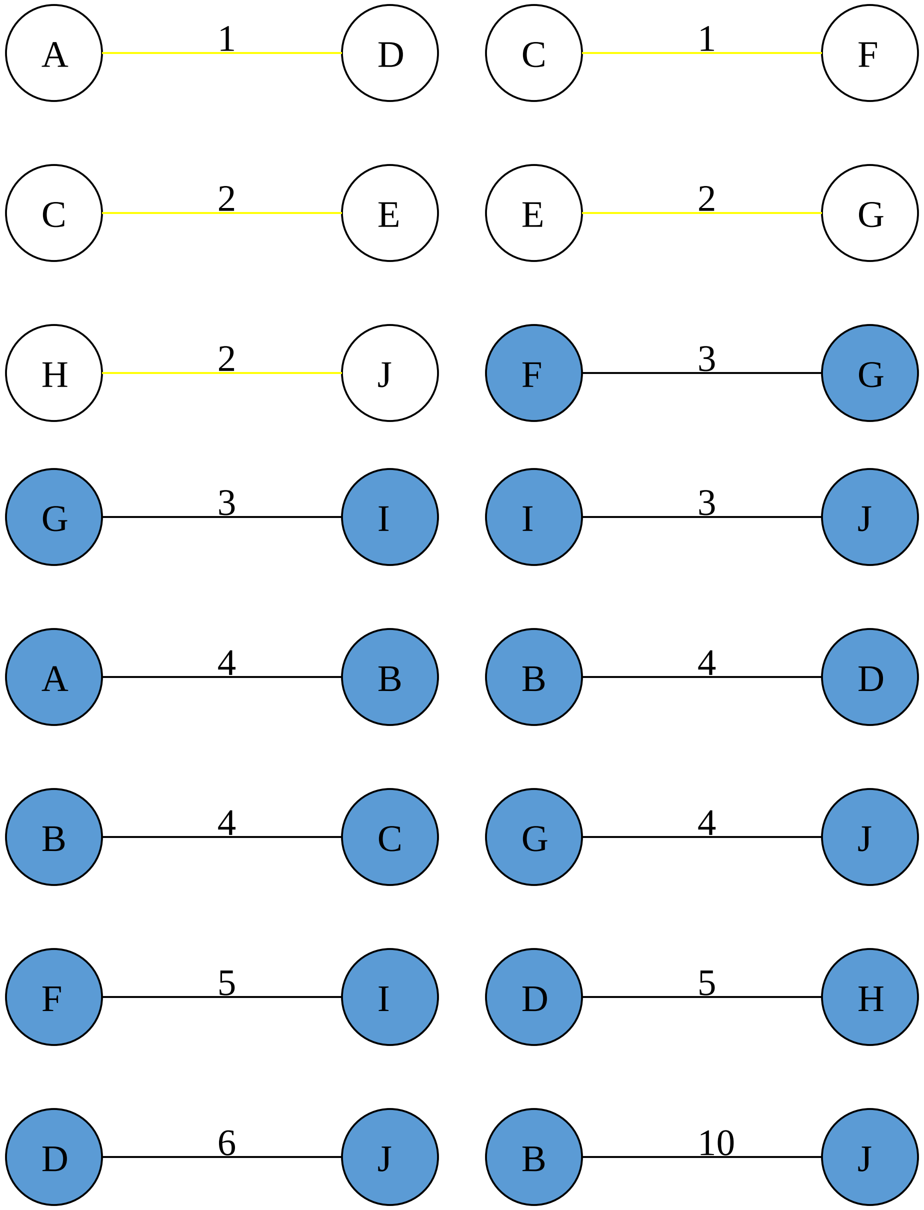
Add Edge


A
B
C
D
E
F
G
H
I
J
4
4
4
1
2
1
2
3
5
10
5
6
3
4
2
3


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

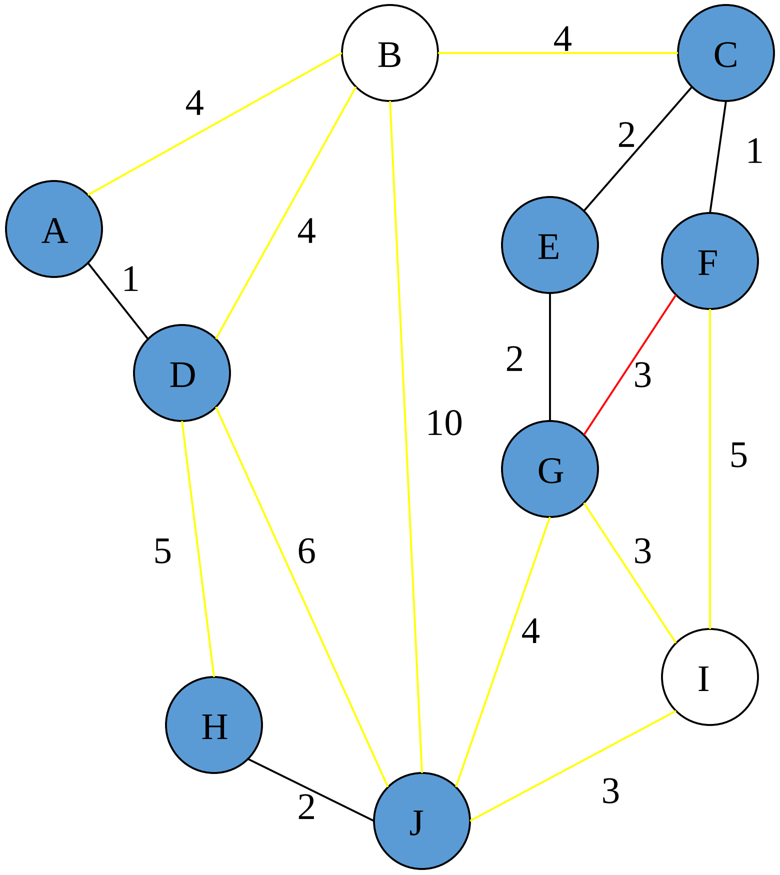
Add Edge

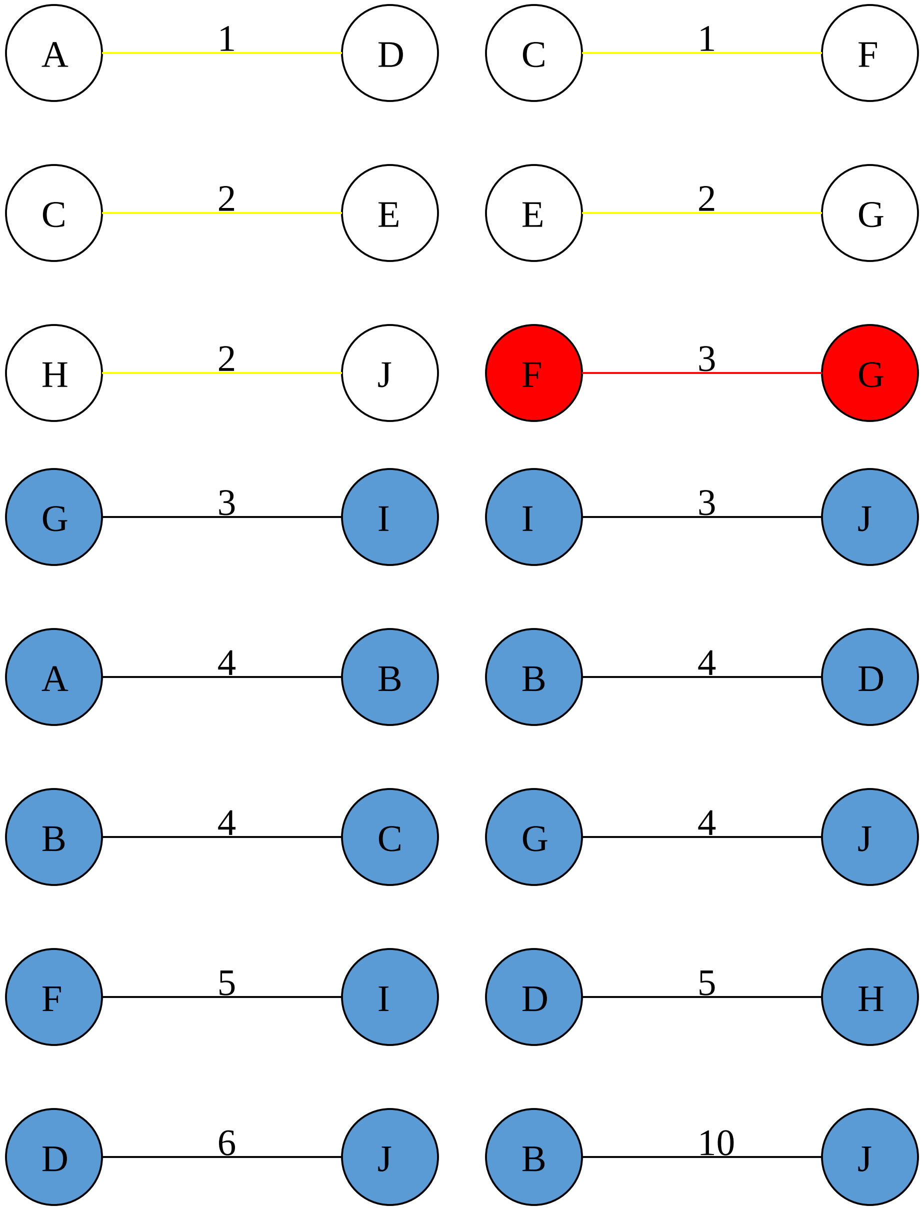

A
B
C
D
E
F
G
H
I
J
4
4
4
1
2
1
2
3
10
5
5
6
4
3
2
3


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

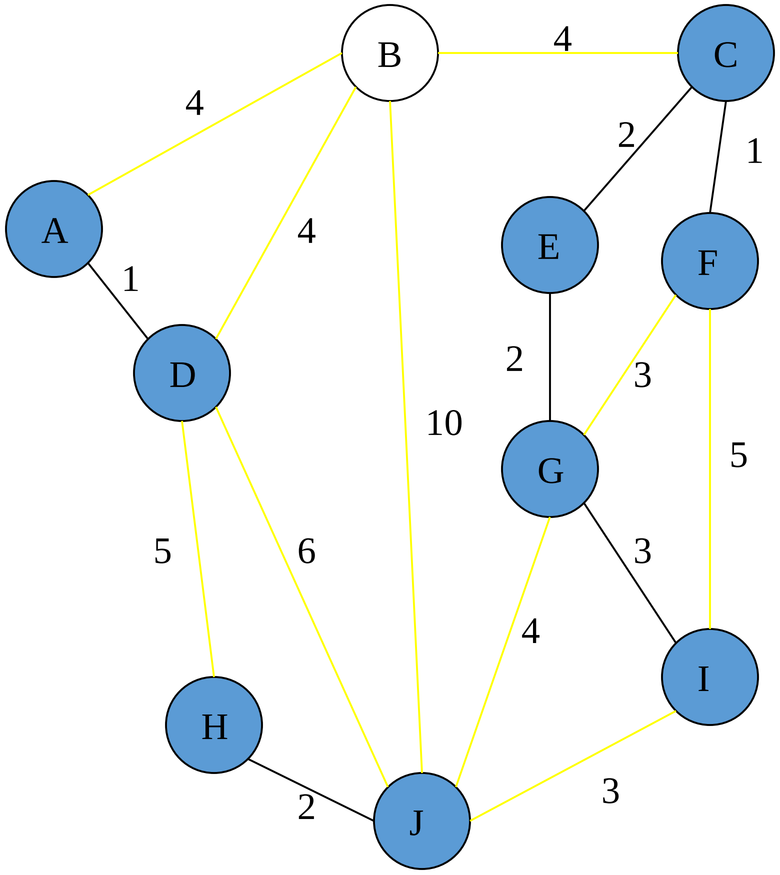
Cycle

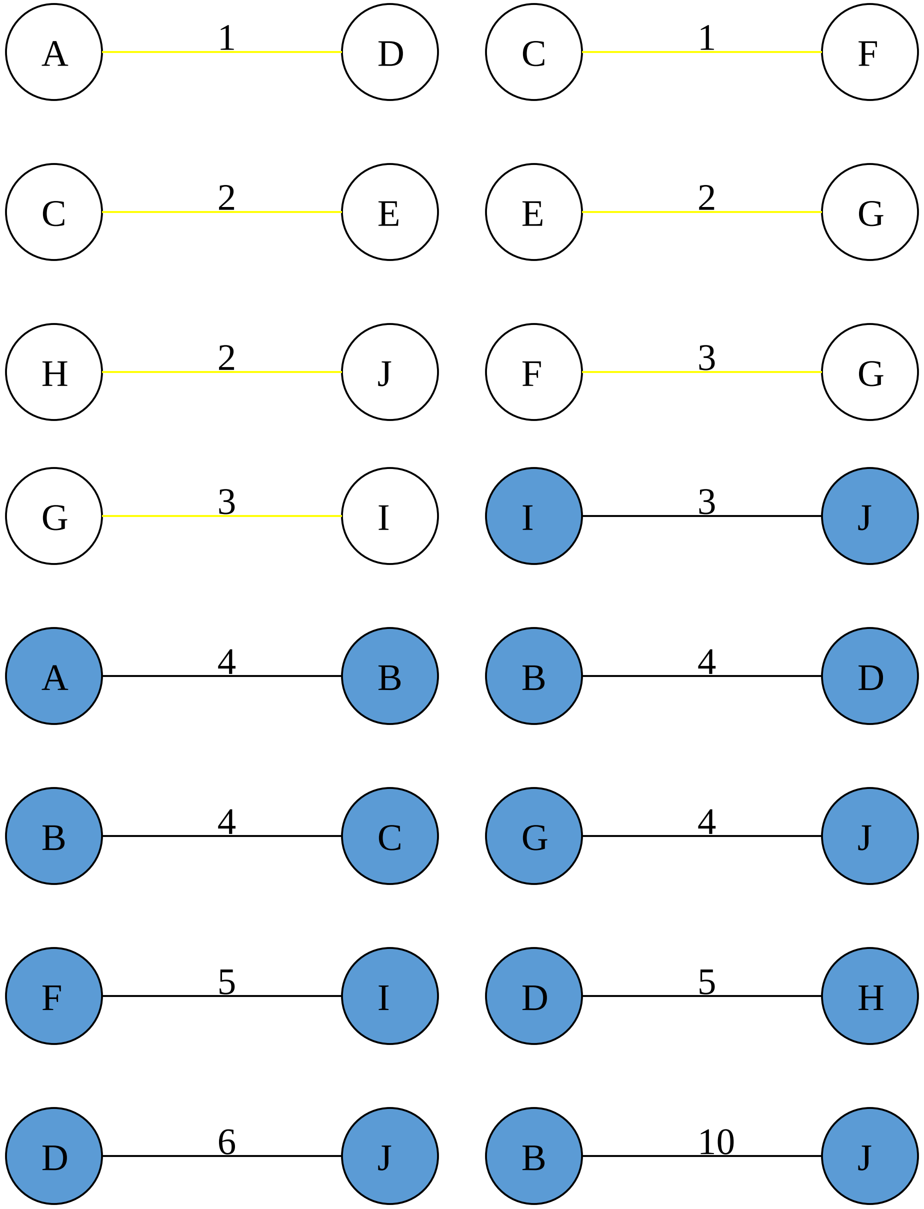
Don’t Add Edge



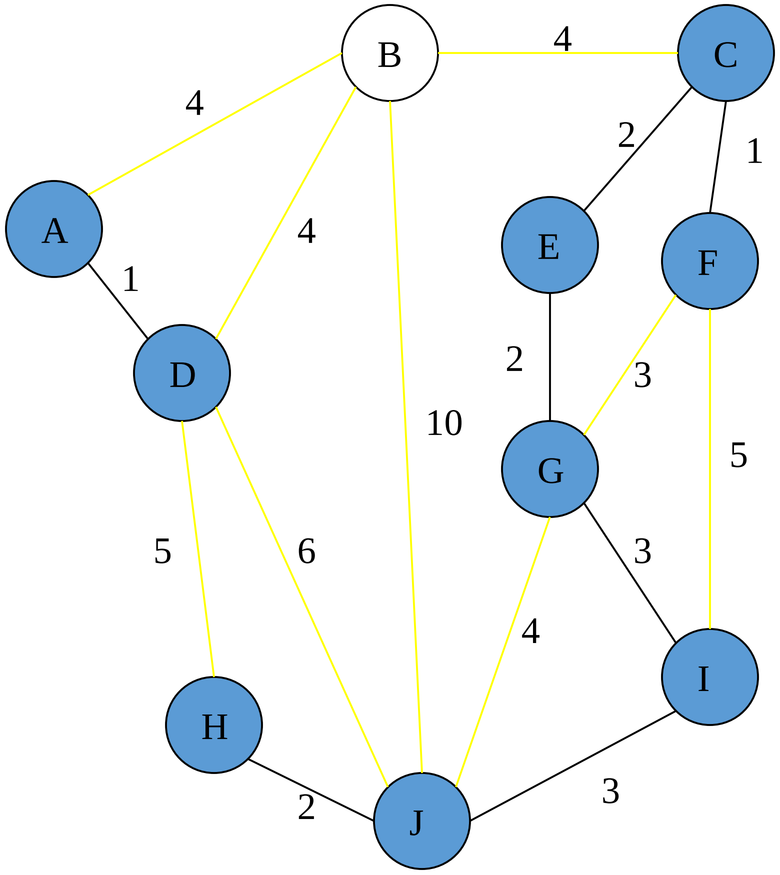


Add Edge

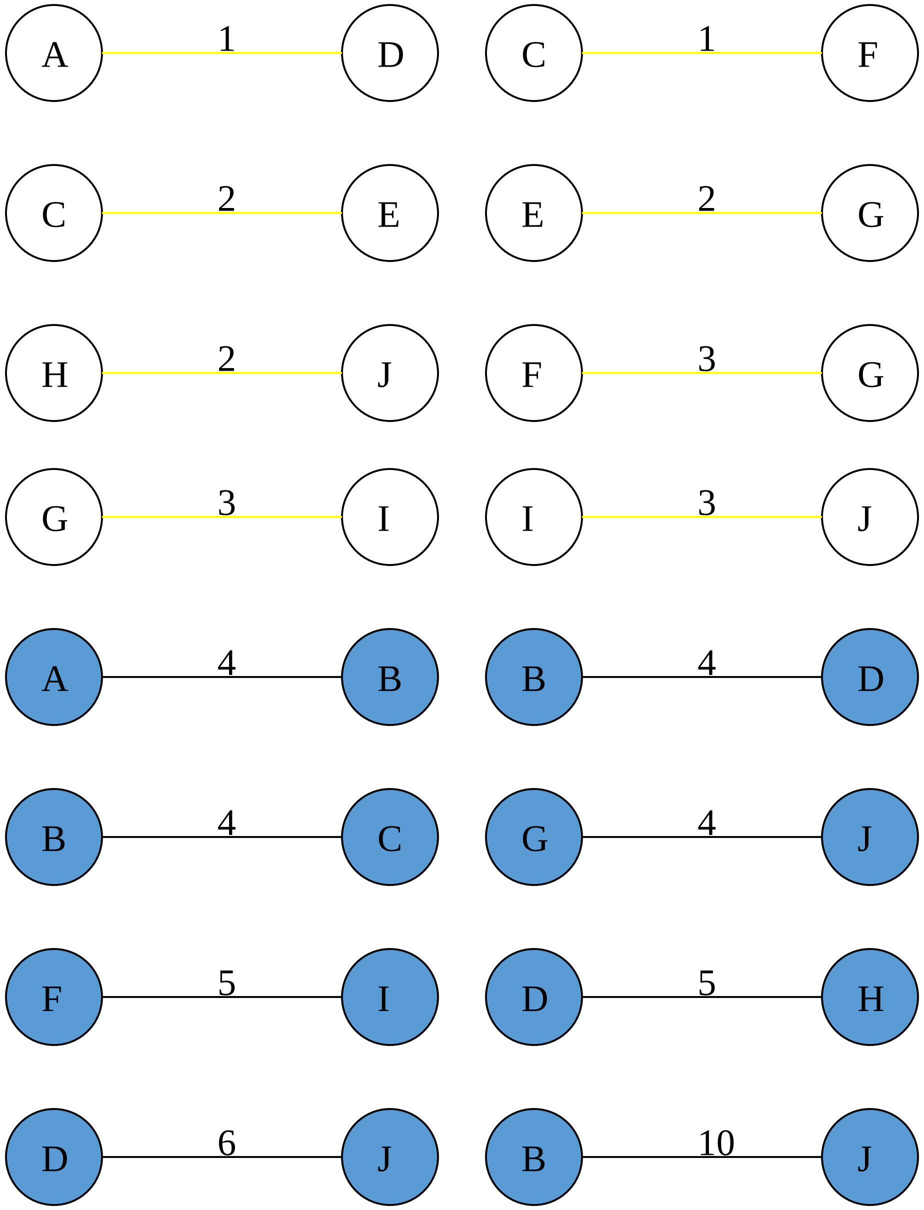

A
B
C
D
E
F
G
H
I
J
4
4
4
1
2
1
2
3
5
10
5
6
4
3
2
3


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

Add Edge

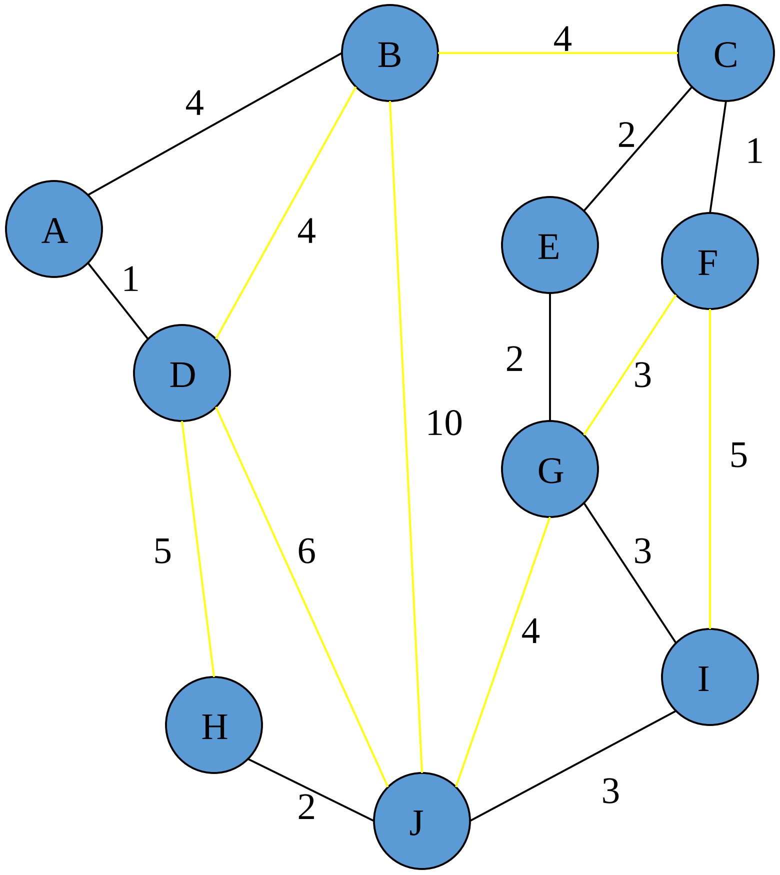


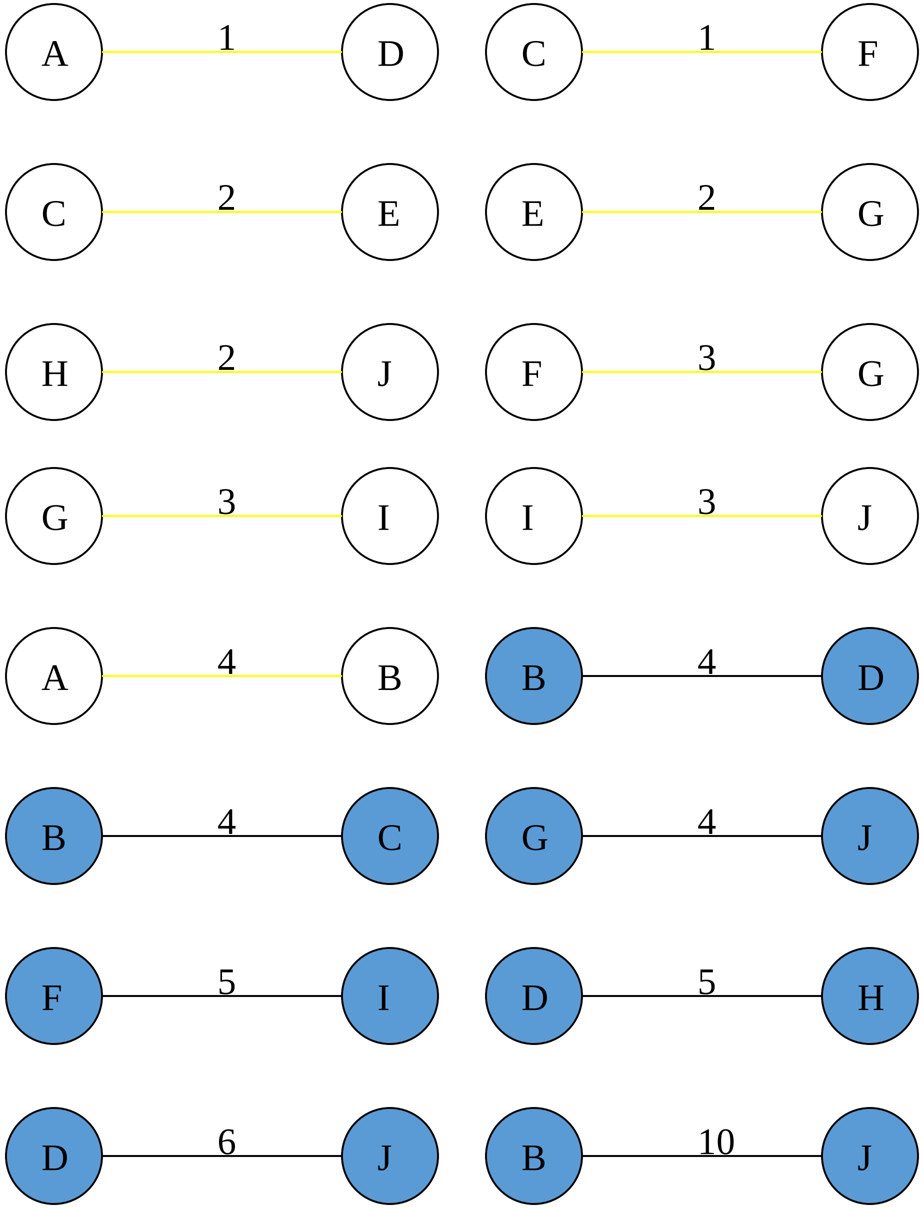
A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
3
10
5
5
6
3
4
2
3



A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

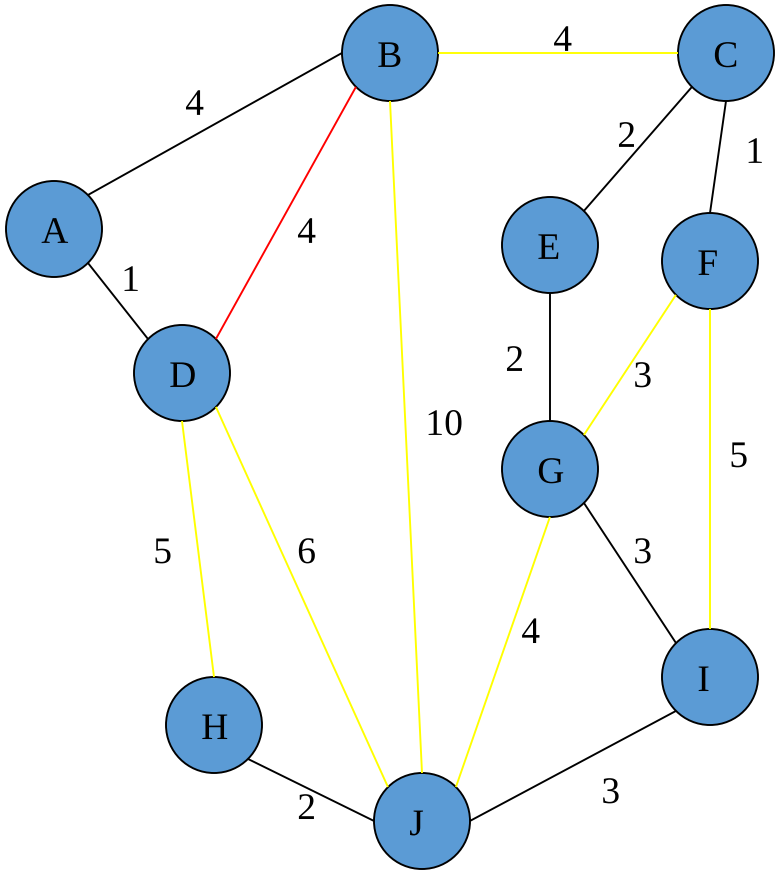
Add Edge

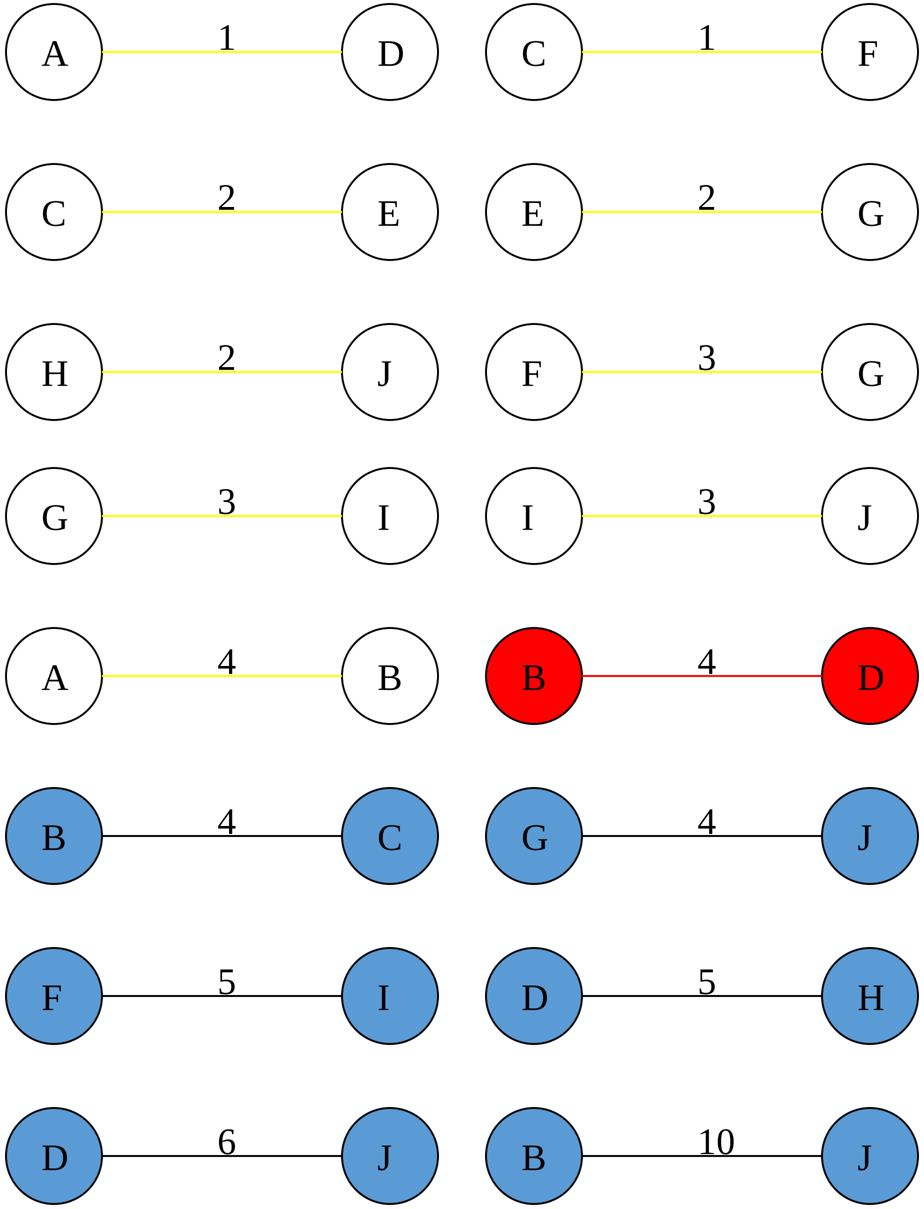

A
B
C
D
E
F
G
H
I
J
4
1
4
4
10
2
1
2
3
5
5
6
4
3
2
3


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

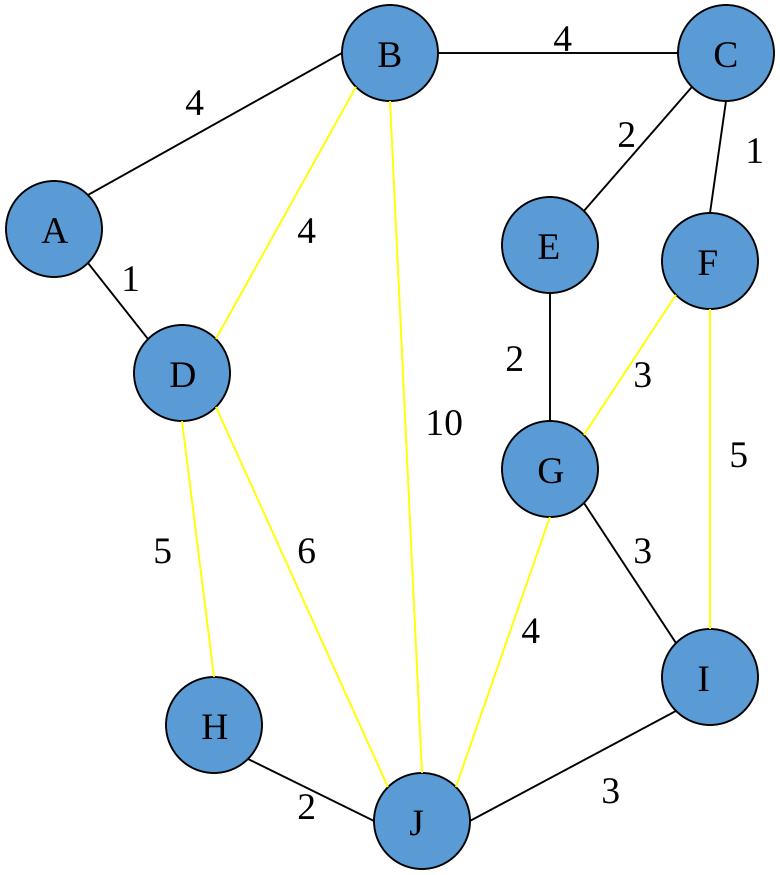
Cycle

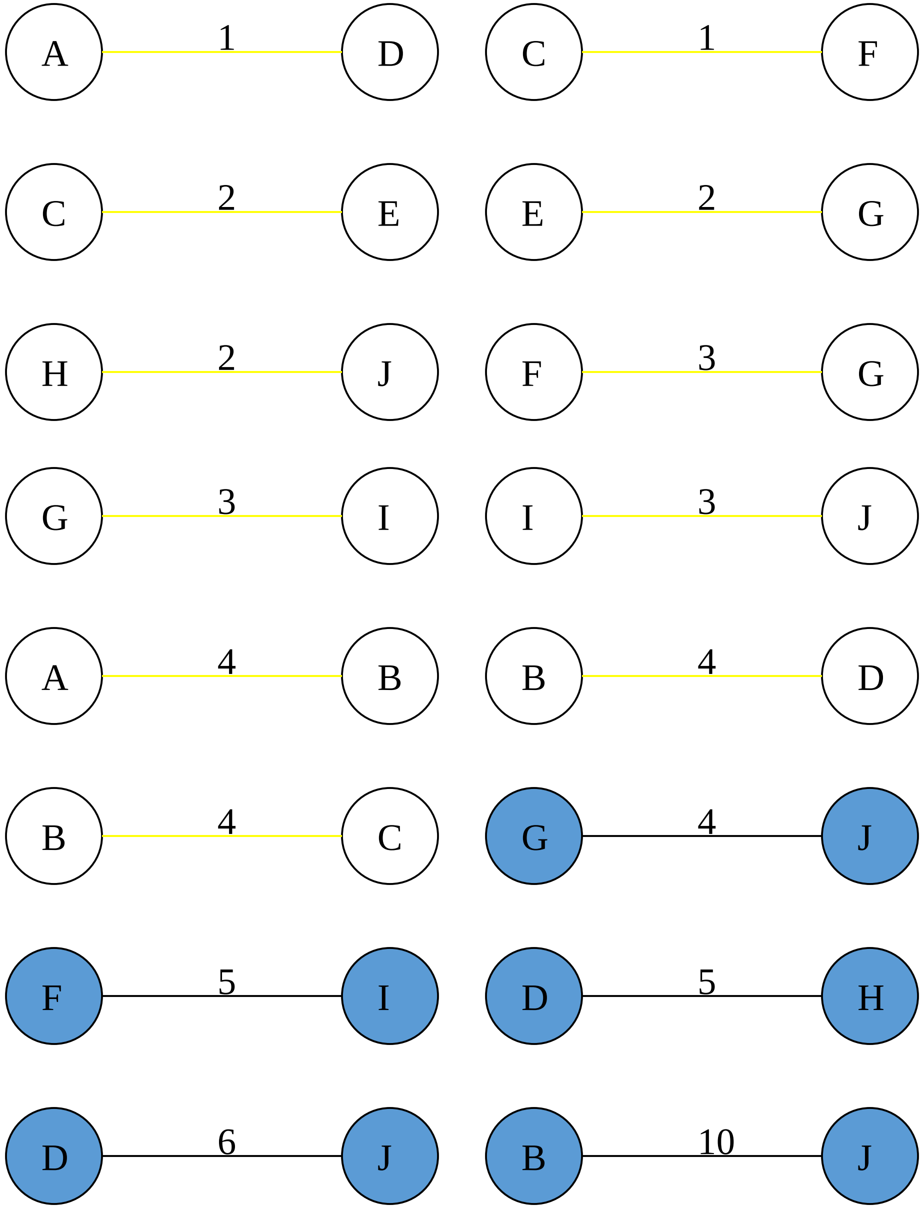
Don’t Add Edge



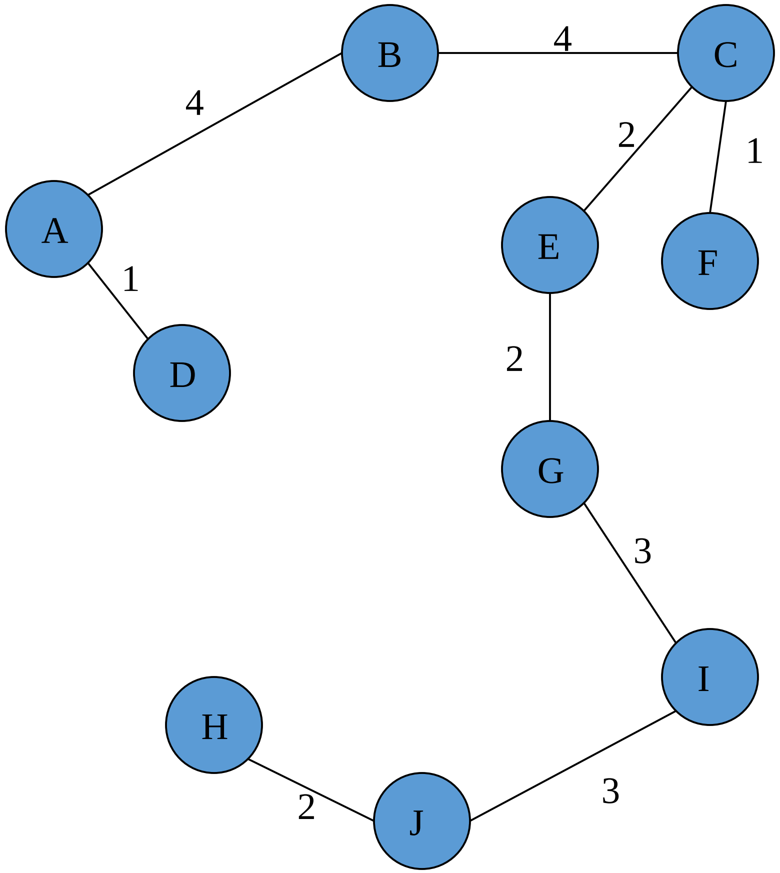


Add Edge

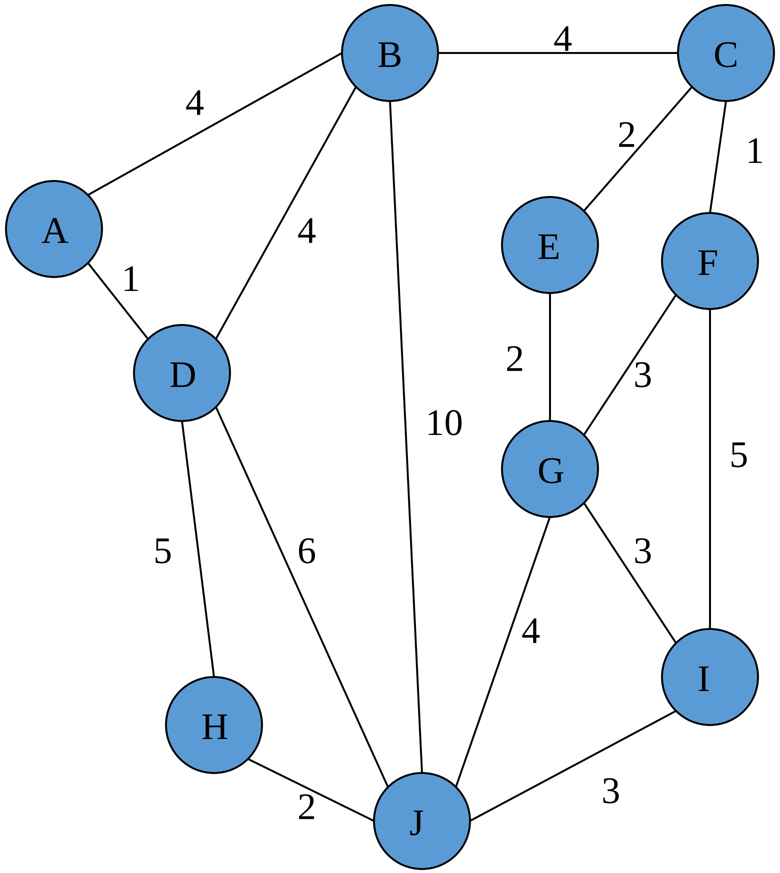

A
B
C
D
E
F
G
H
I
J
4
4
1
4
10
2
1
2
3
5
6
3
4
3
2
5
3


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

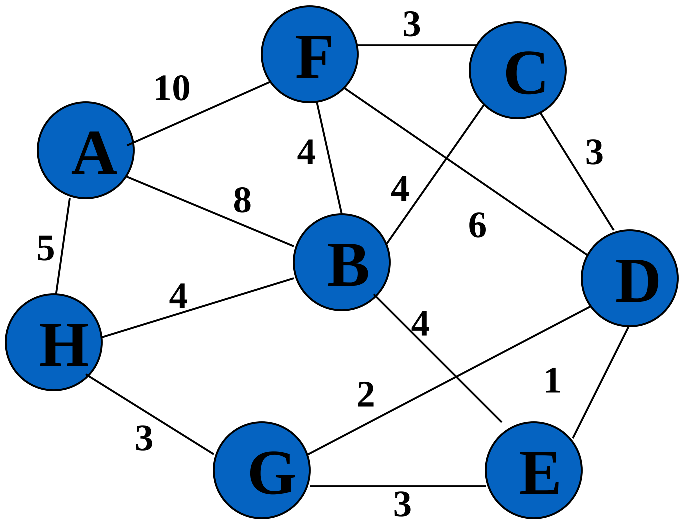
Minimum Spanning Tree



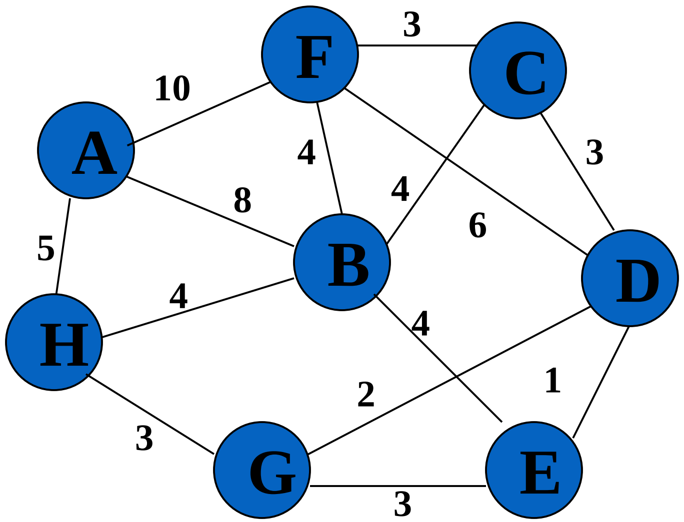
Complete Graph



Walk-Through



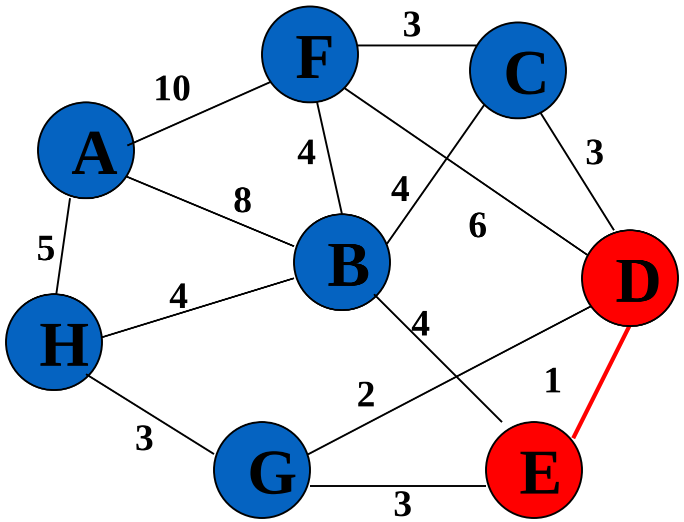
Consider an undirected, weight graph



Sort the edges by increasing edge weight

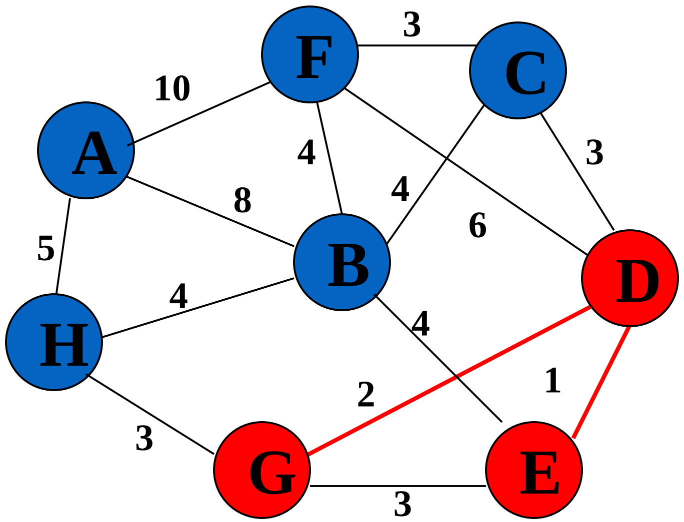
edge	d_v	
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



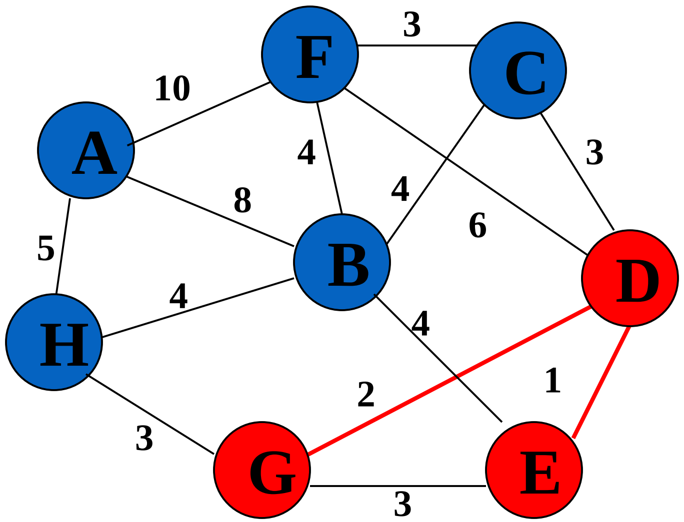
edge	d_v	
(D,E)	1	√
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

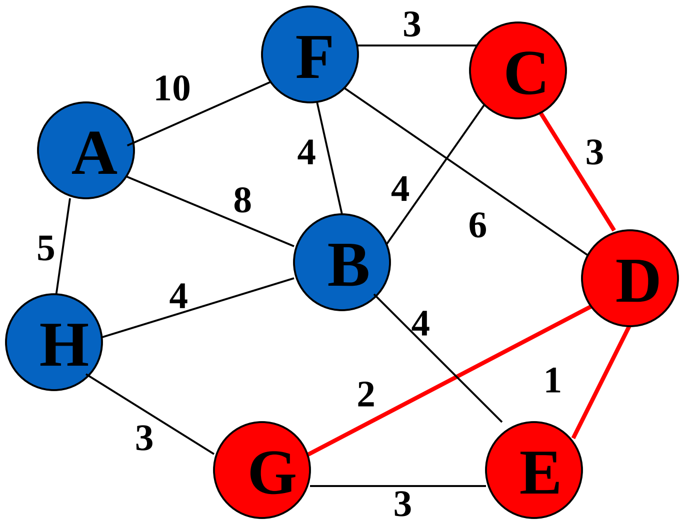
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

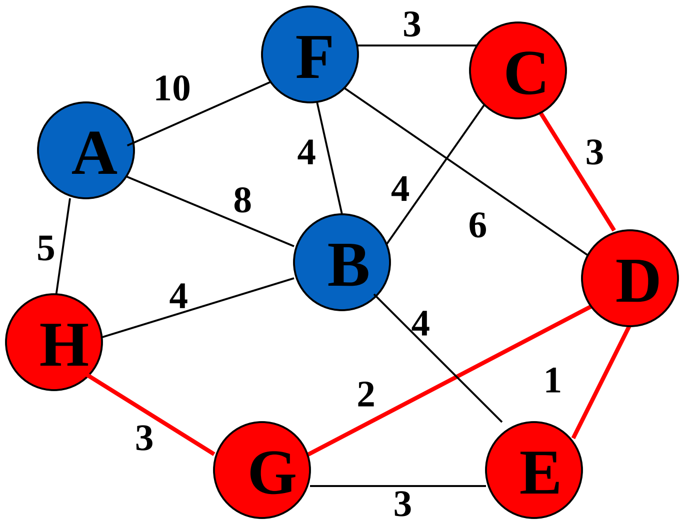
edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Accepting edge (E,G) would create a cycle



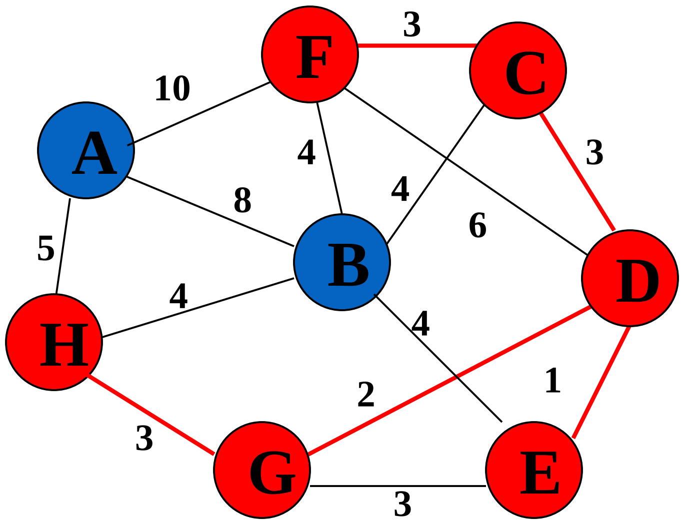
edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



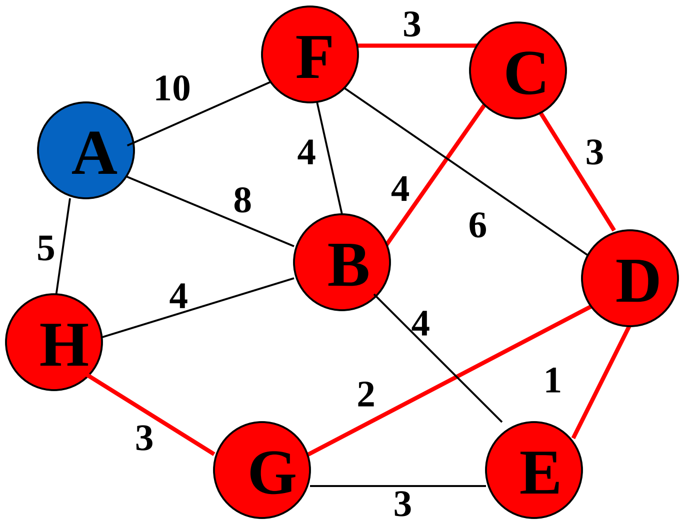
edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



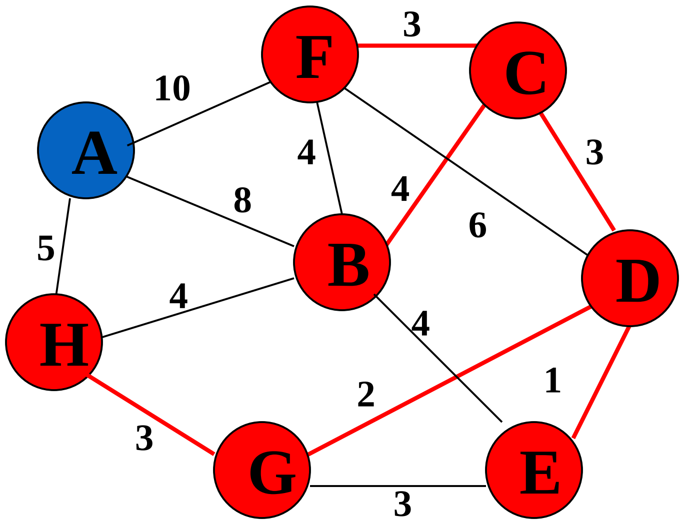
edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



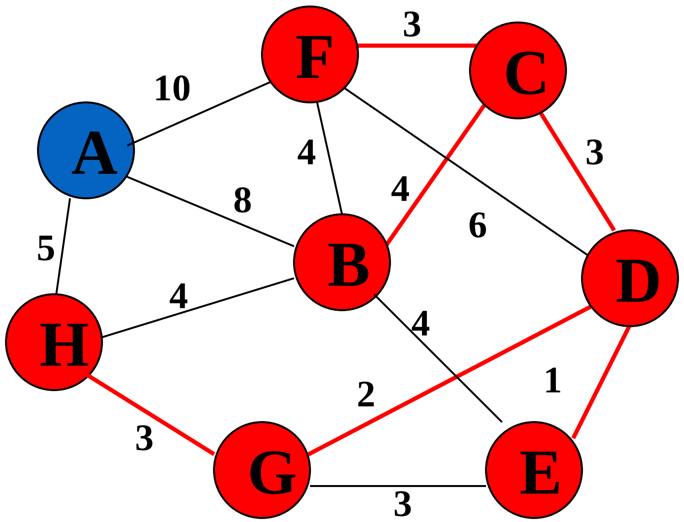
edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



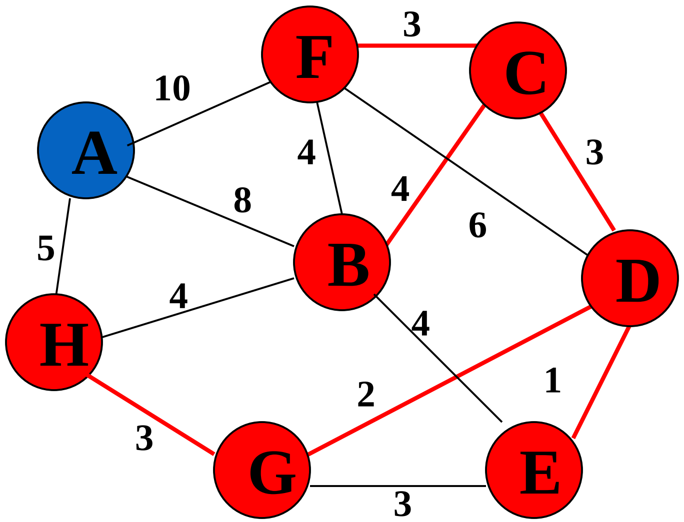
edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



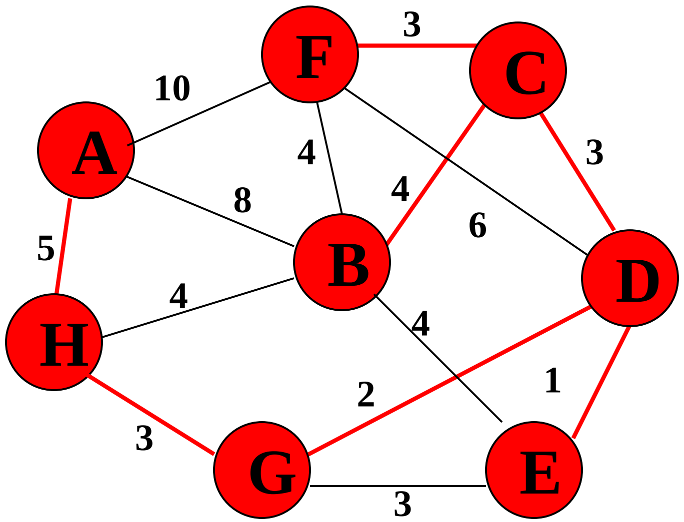
edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



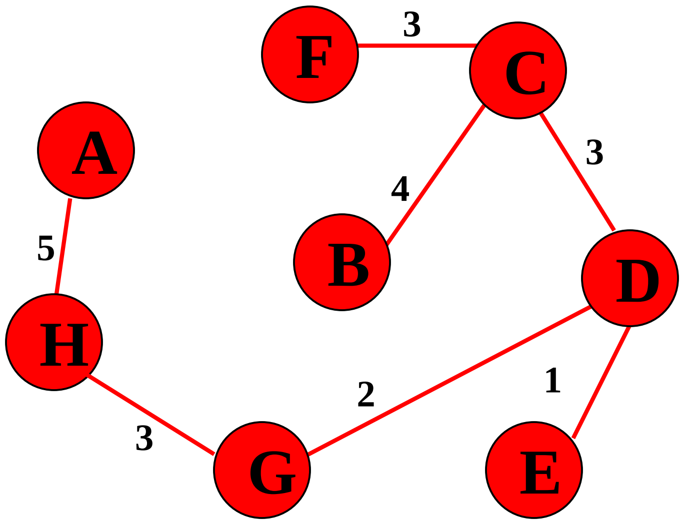
edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	



edge	d_v	
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

edge	d_v	
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

} not considered

Done

Total Cost = $\Sigma\, d_v$ = *21*

Analysis of Kruskal's Algorithm

Running Time = O(e log v)   (e = edges, v = nodes)

Testing if an edge creates a cycle can be slow unless a complicated data structure called a “union-find” structure is used.

It usually only has to check a small fraction of the edges, but in some cases (like if there was a vertex connected to the graph by only one edge and it was the longest edge) it would have to check all the edges.

This algorithm works best, of course, if the number of edges is kept to a minimum.