Minimum Spanning Tree

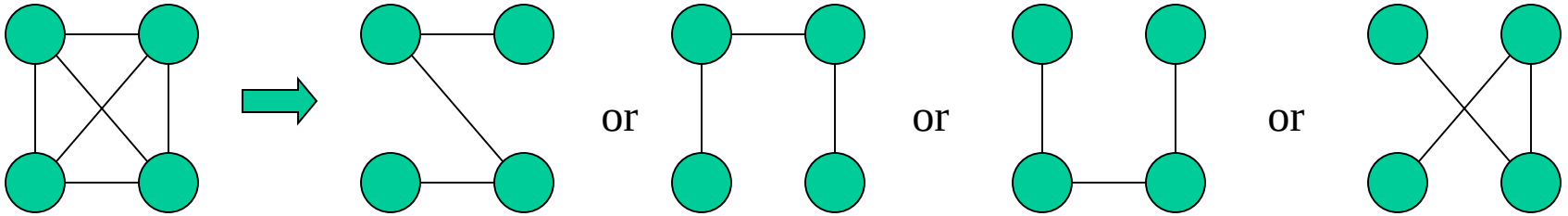
Spanning Trees

A spanning tree of a graph is just a subgraph that contains all the vertices and is a tree.

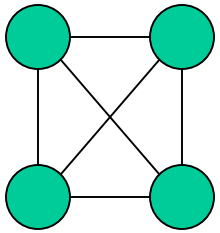
A graph may have many spanning trees.

Graph A

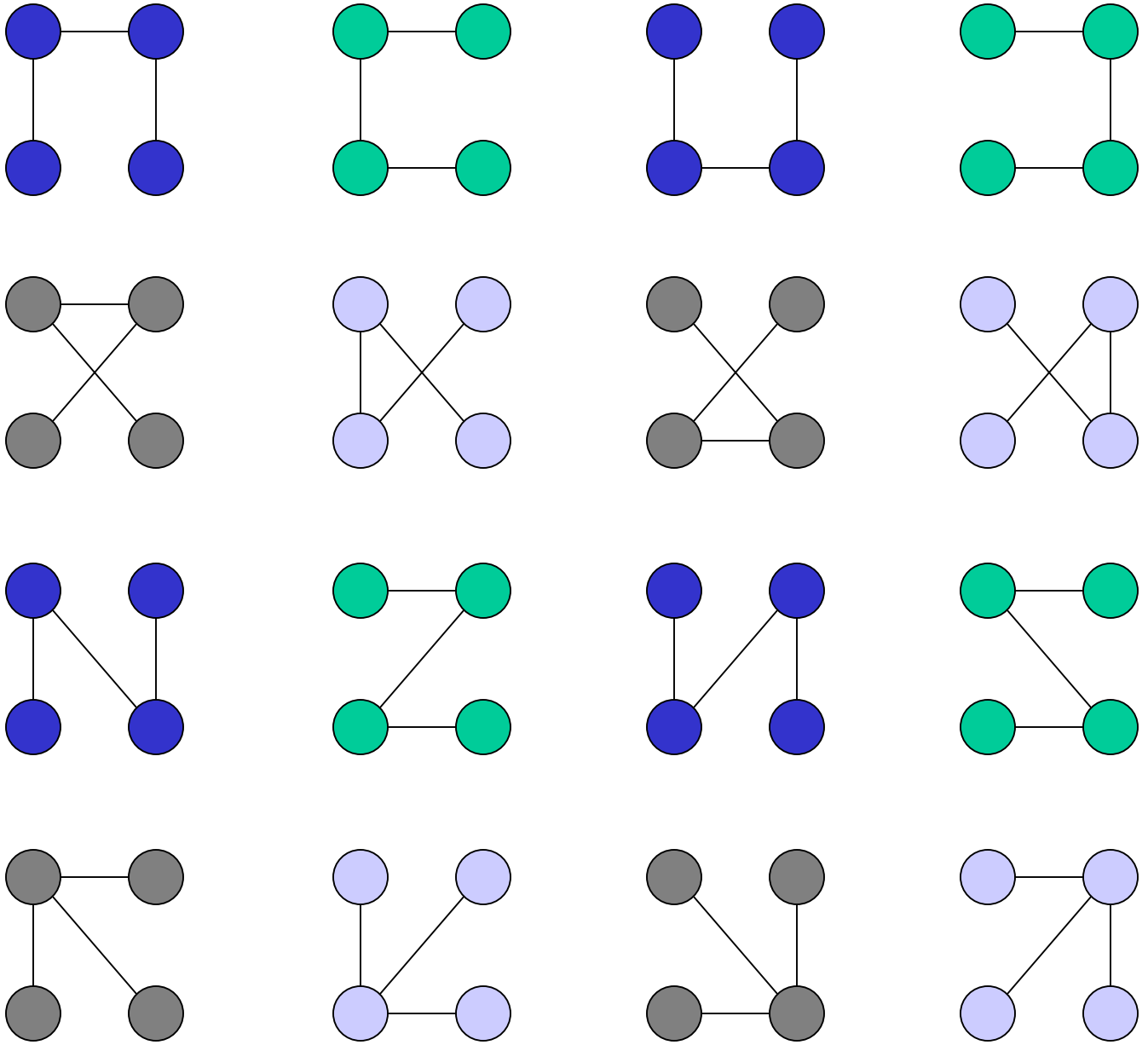
Some Spanning Trees from Graph A



Complete Graph



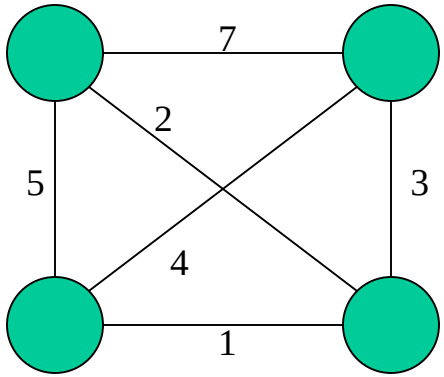
All 16 of its Spanning Trees



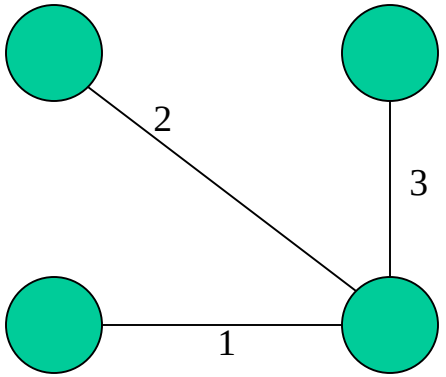
Minimum Spanning Trees

The Minimum Spanning Tree for a given graph is the Spanning Tree of minimum cost for that graph.

Complete Graph



Minimum Spanning Tree



Algorithms for Obtaining the Minimum Spanning Tree

- Kruskal's Algorithm
- Prim's Algorithm
- Boruvka's Algorithm

Kruskal's Algorithm

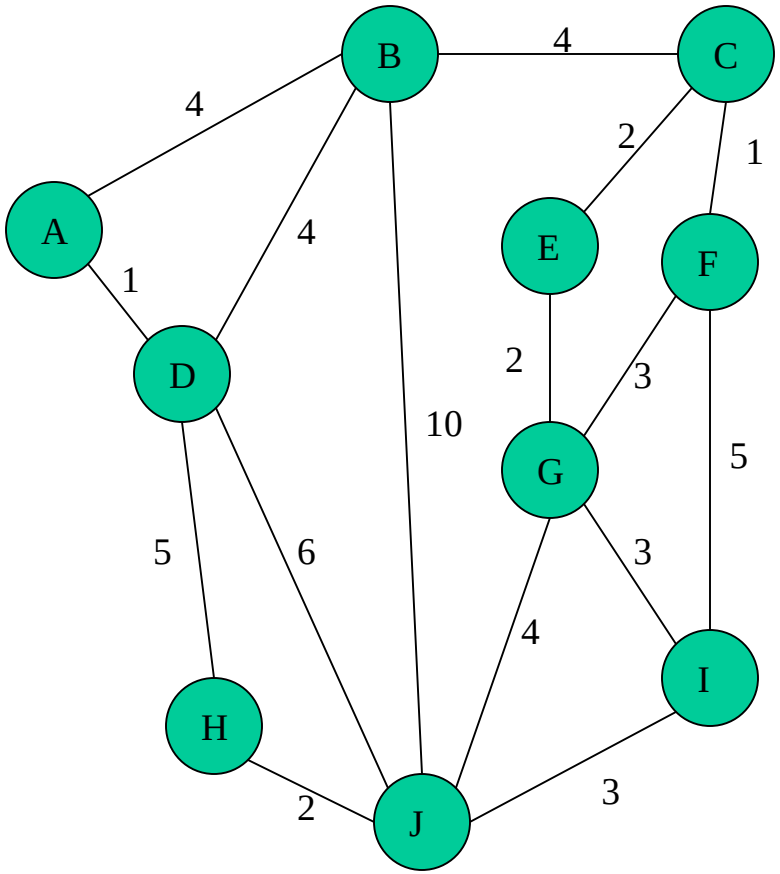
This algorithm creates a forest of trees. Initially the forest consists of n single node trees (and no edges). At each step, we add one edge (the cheapest one) so that it joins two trees together. If it were to form a cycle, it would simply link two nodes that were already part of a single connected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

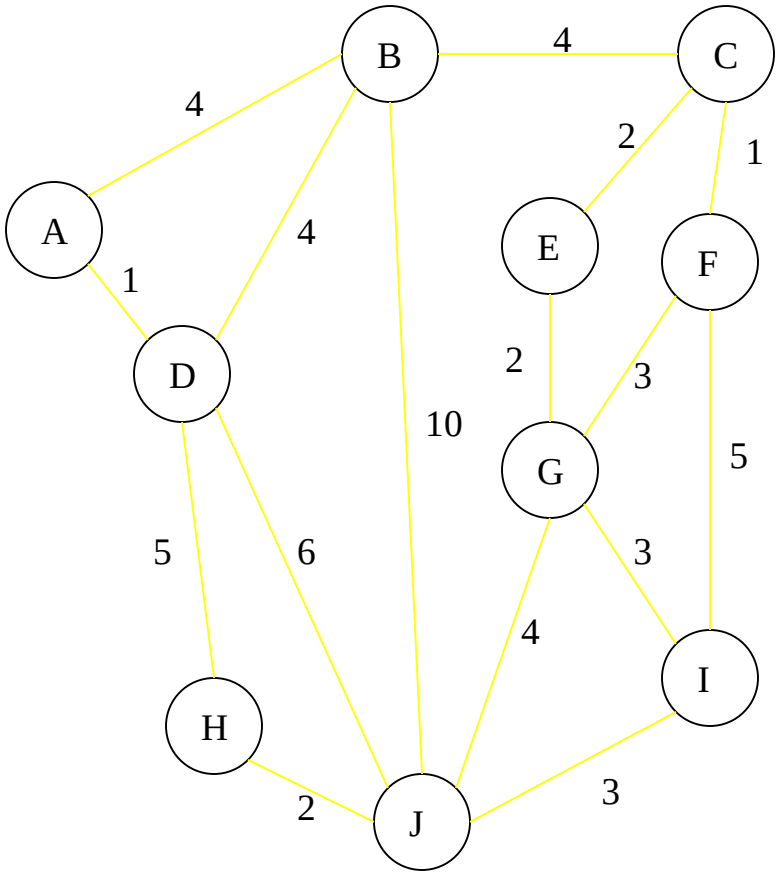
The steps are:

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





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

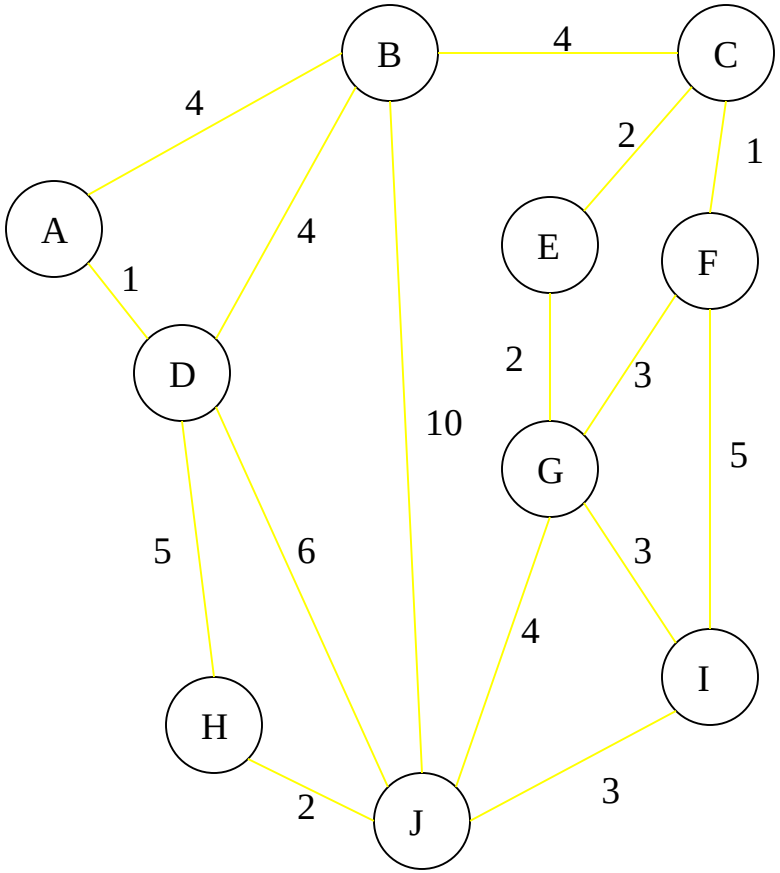
G — 4 — J

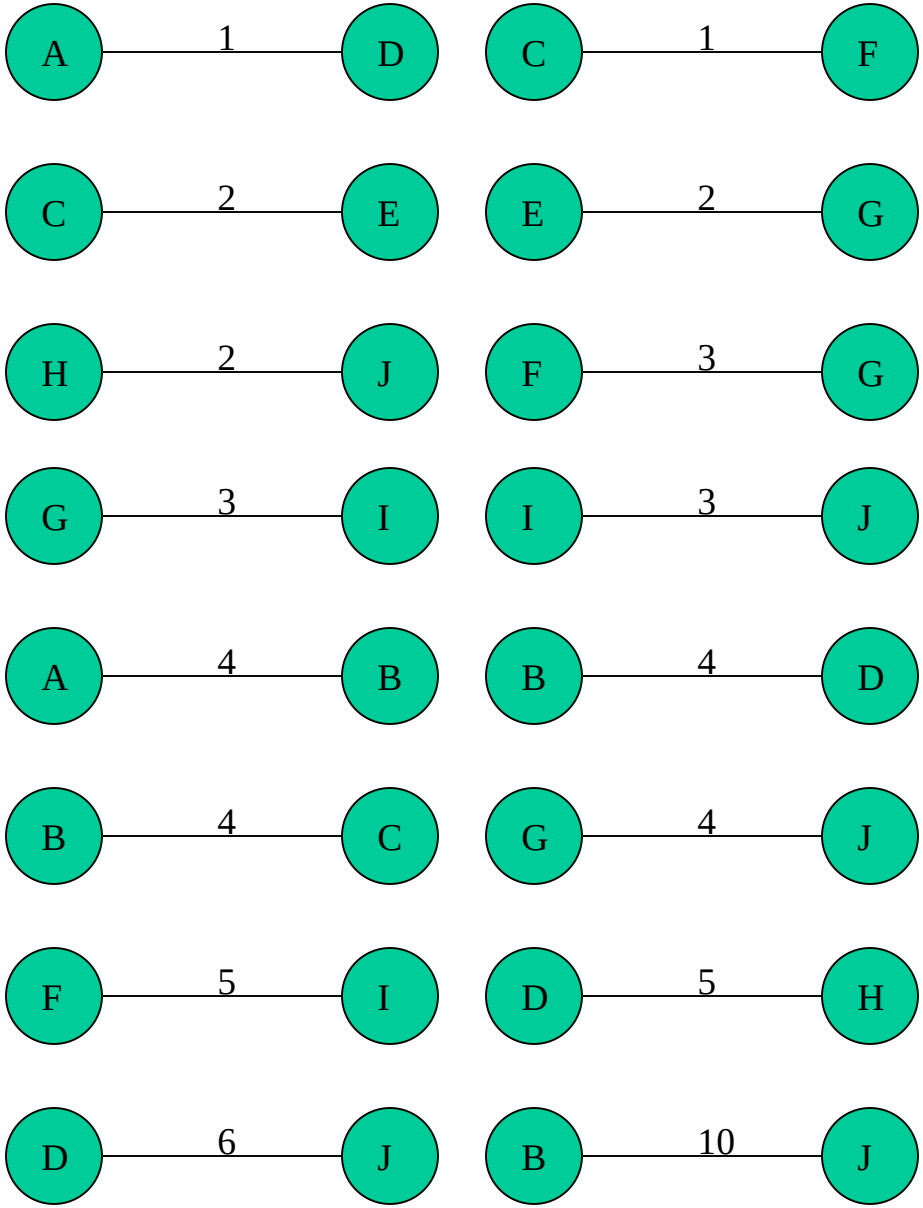
H — 2 — J

I — 3 — J

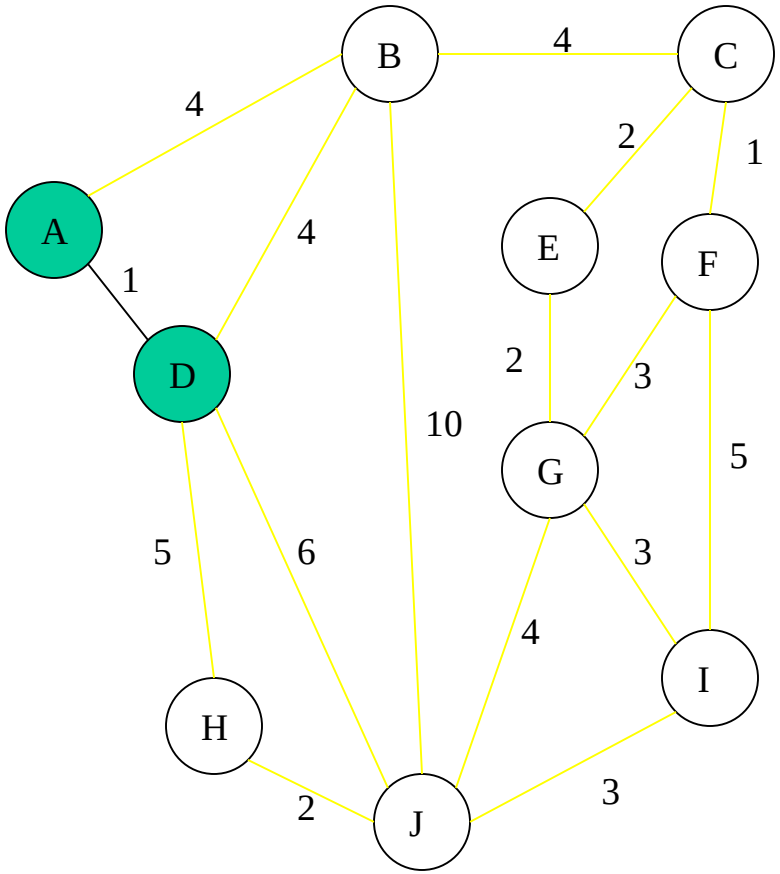
Sort Edges

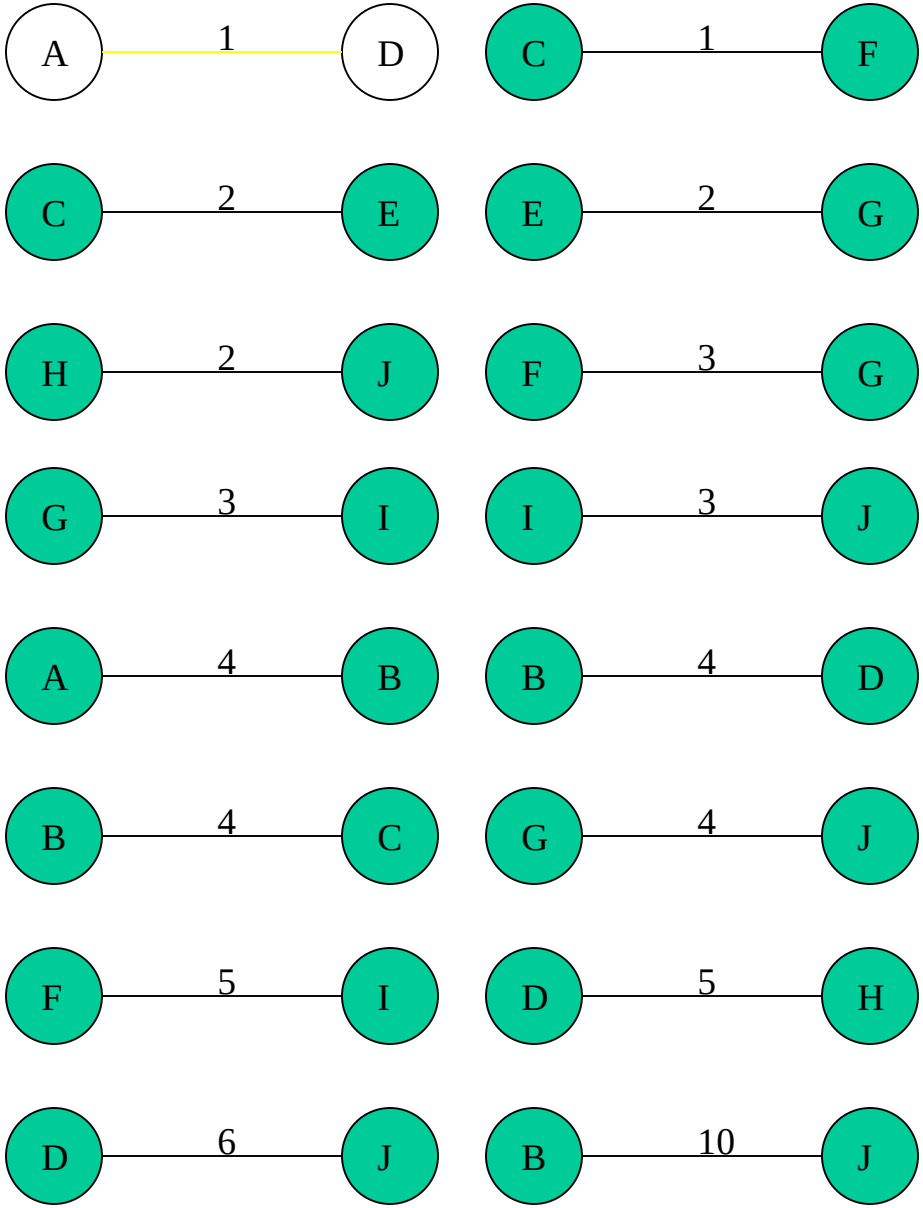
(in reality they are placed in a priority queue - not sorted - but sorting them makes the algorithm easier to visualize)



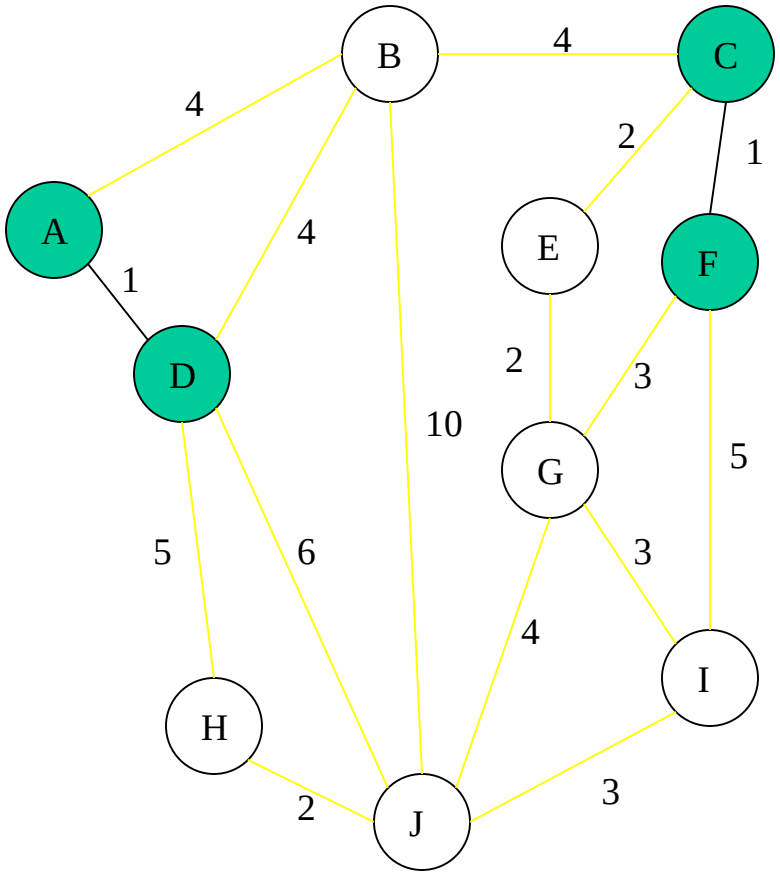


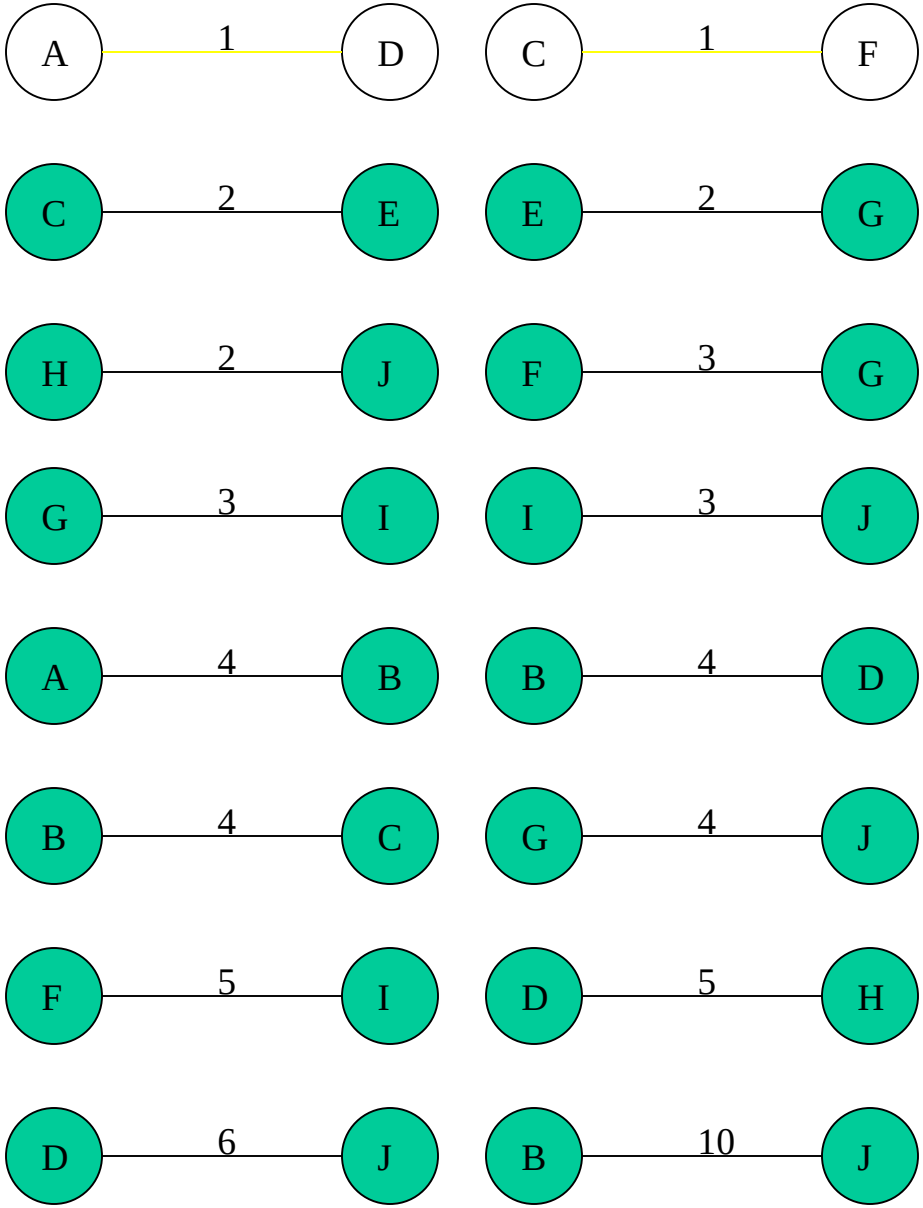
Add Edge


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

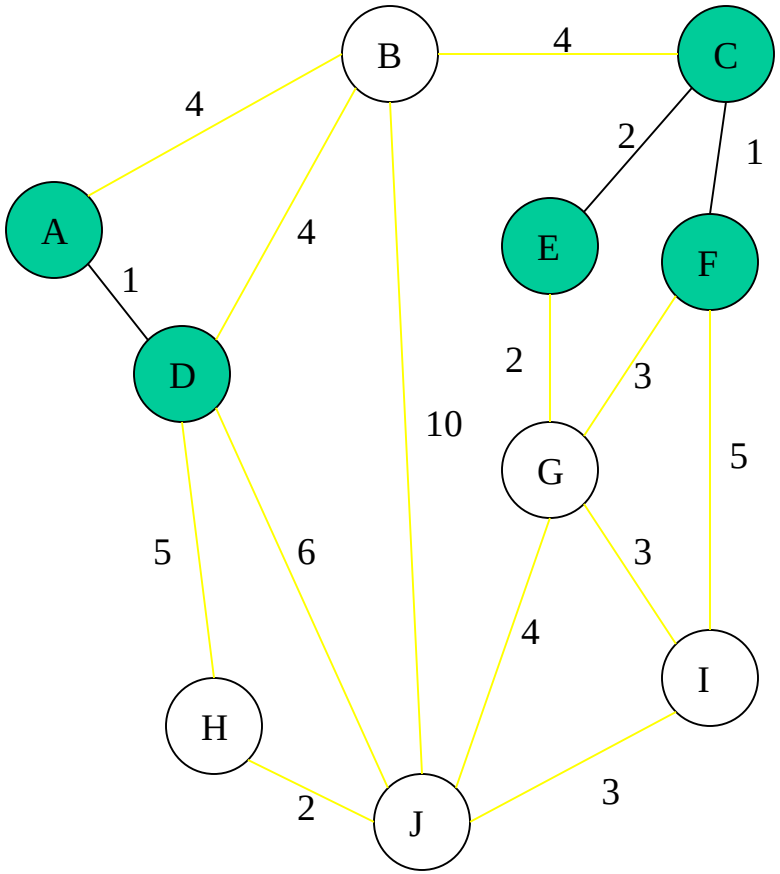

A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

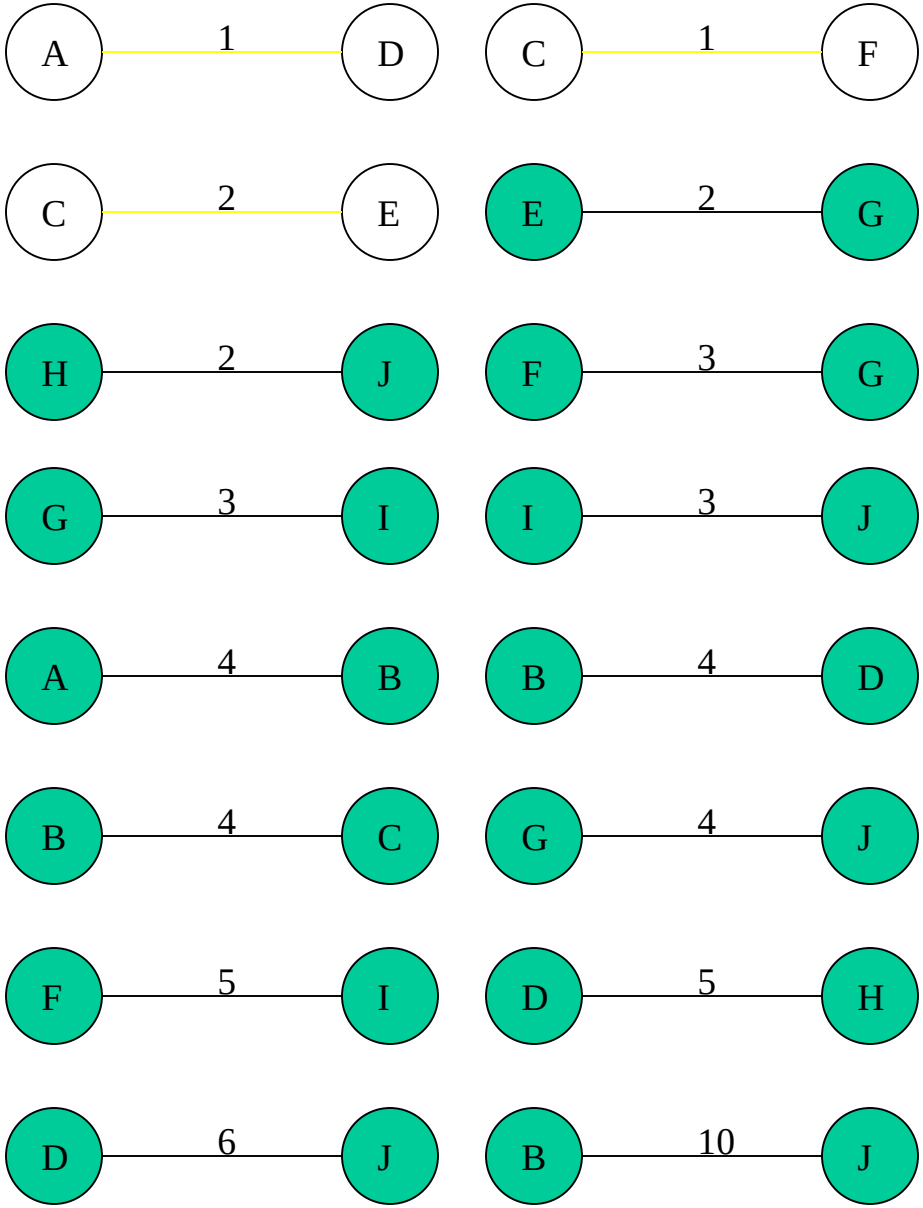
Add Edge


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

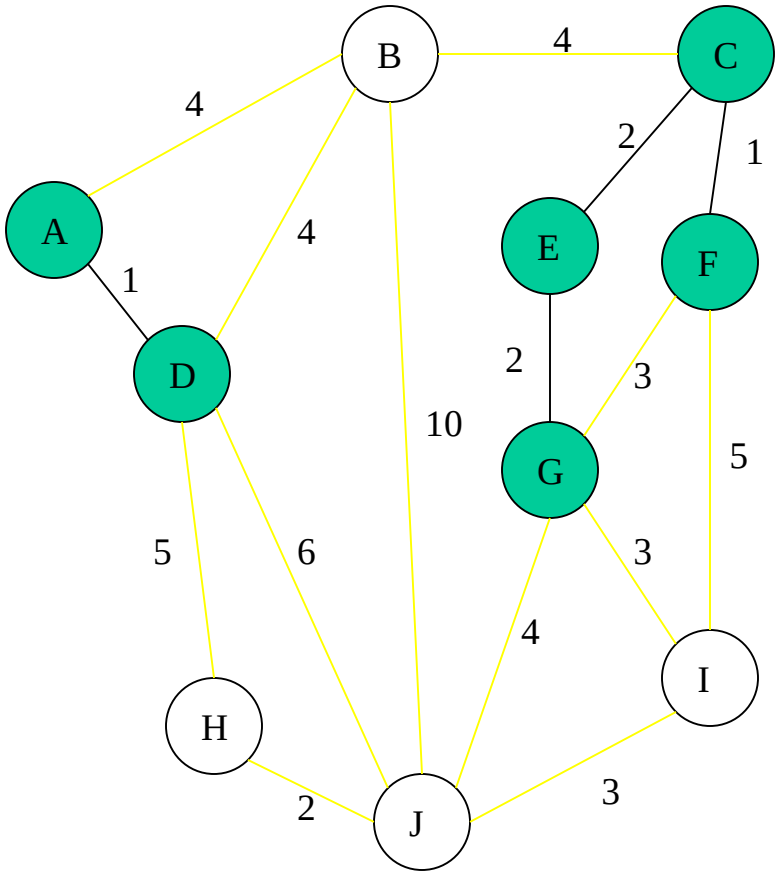

A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

Add Edge


A
B
C
D
E
F
G
H
I
J
4
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 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

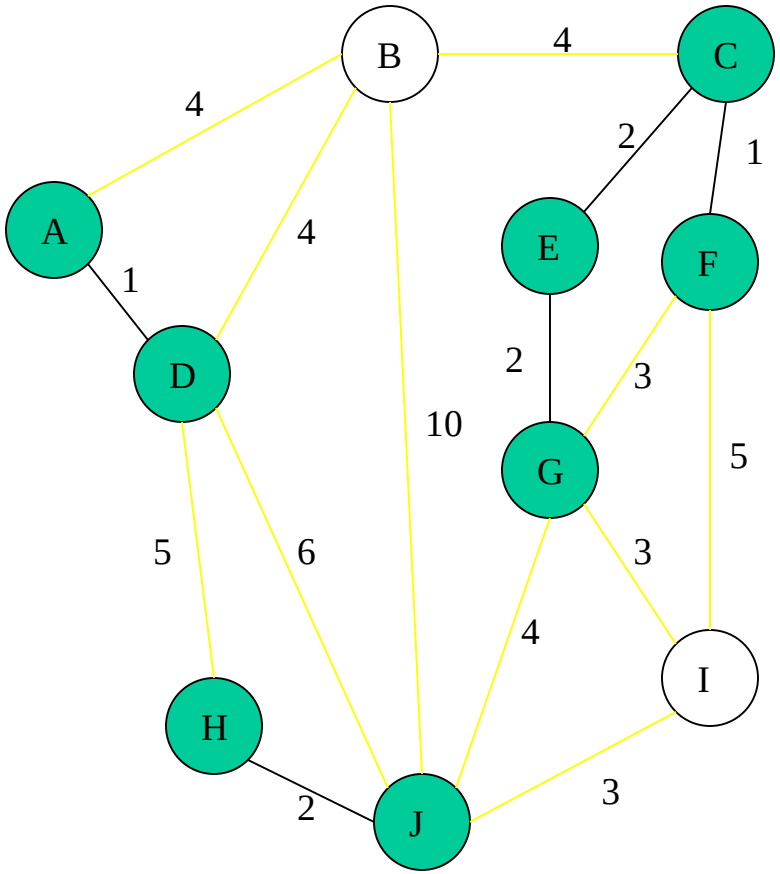
F 5 I

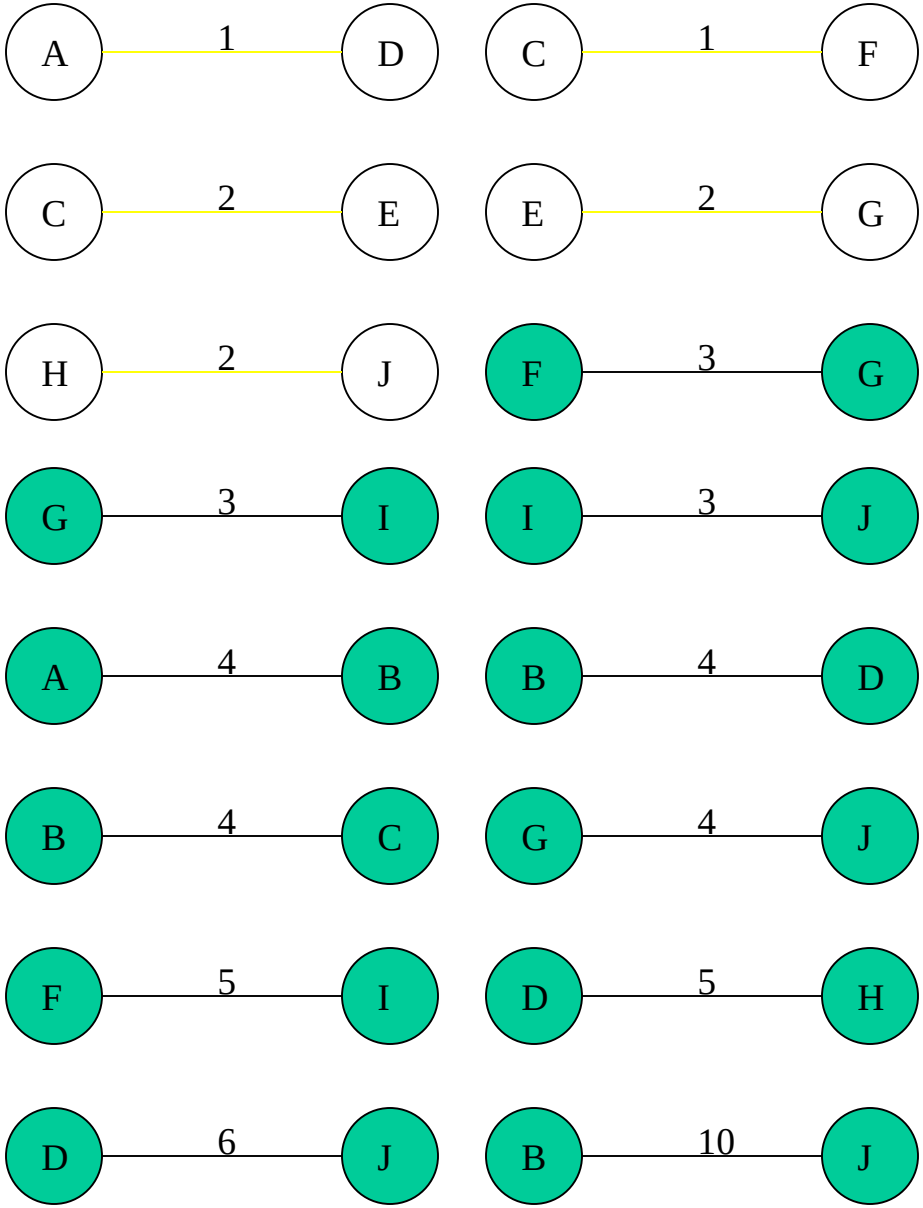
D 5 H

D 6 J

B 10 J

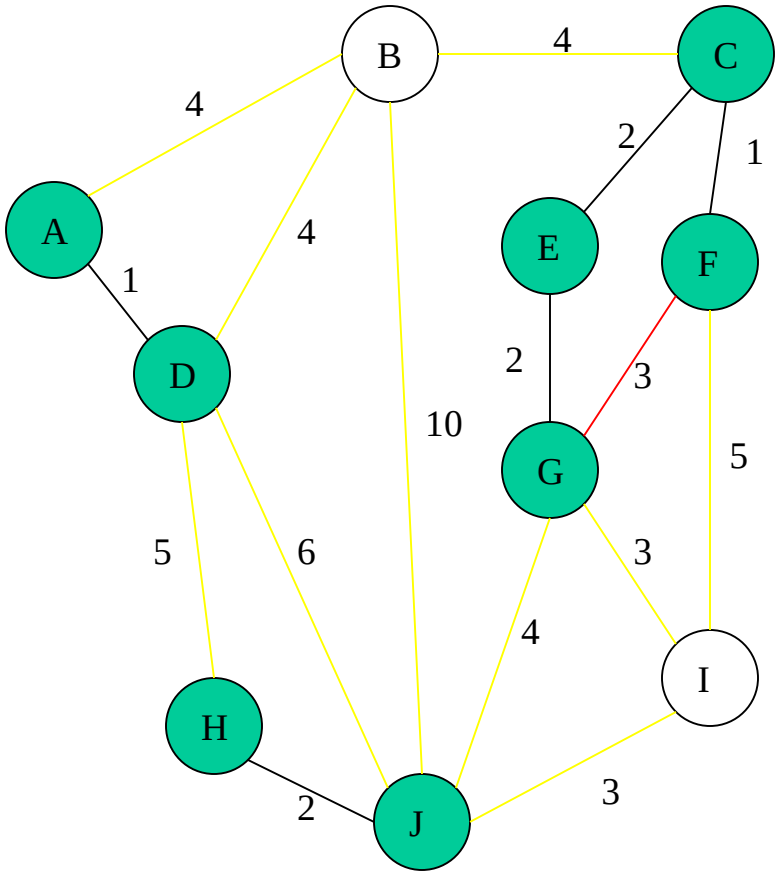
Add Edge

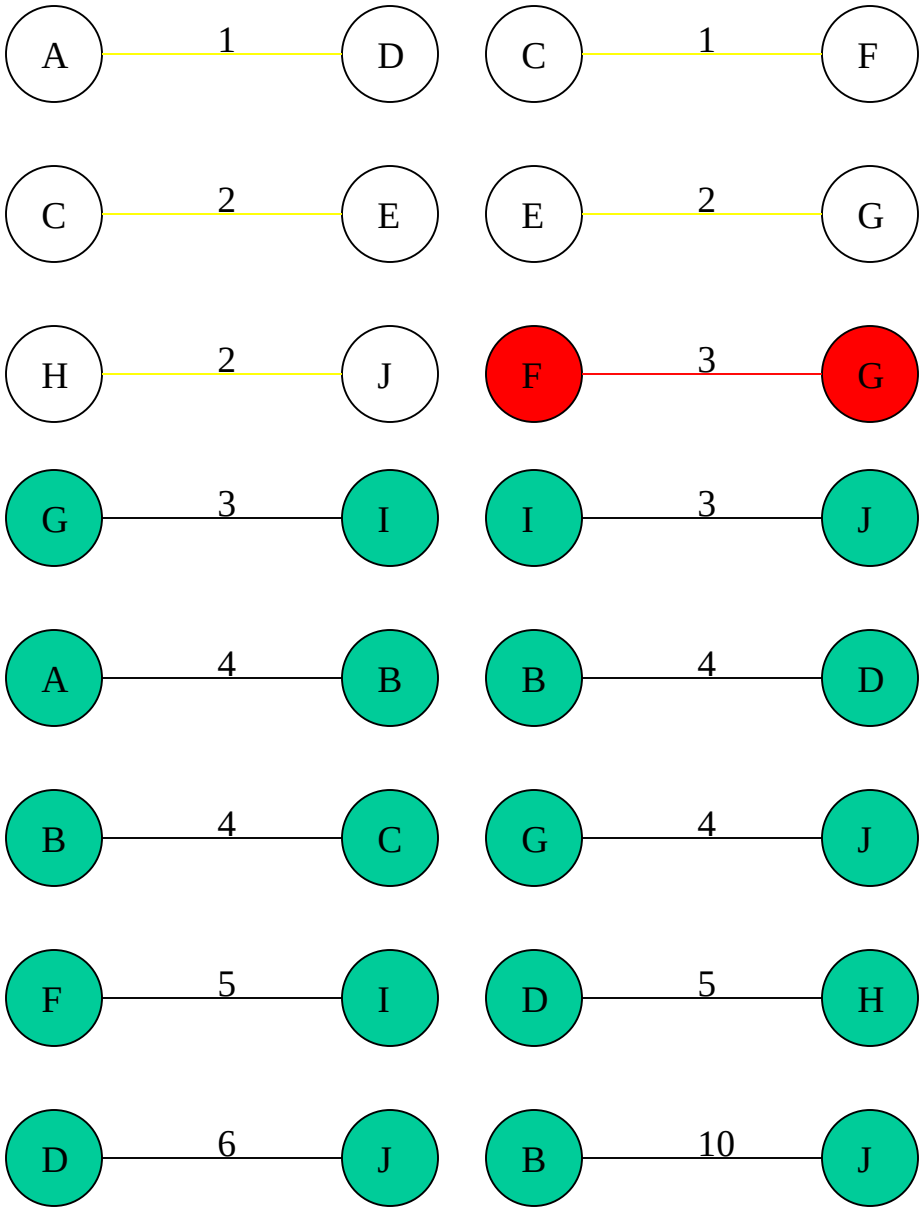

A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
3
10
5
5
6
3
4
2
3


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

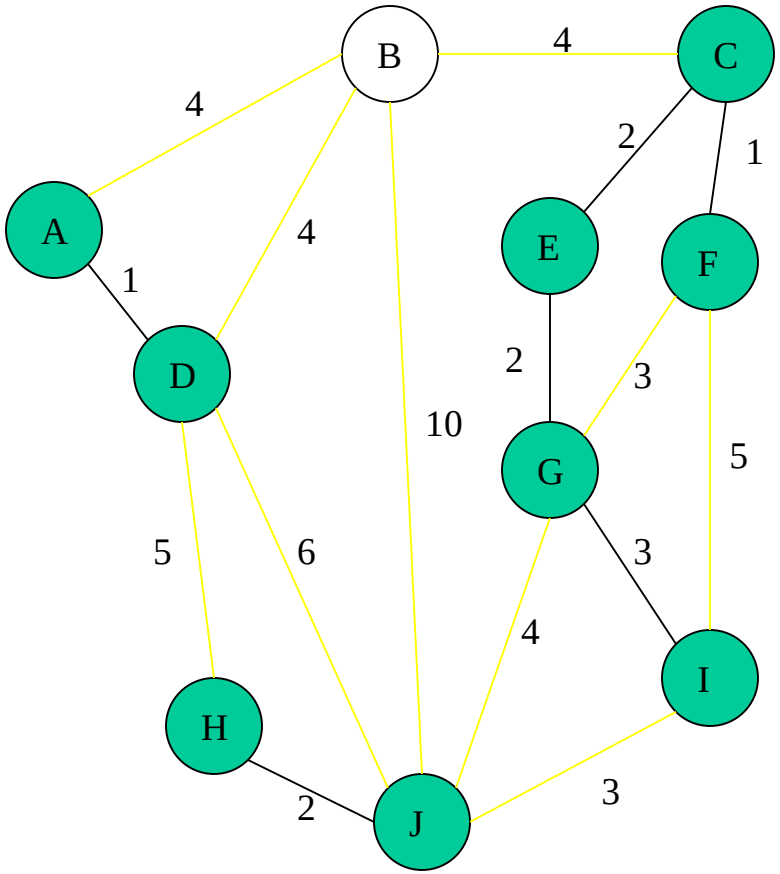
Cycle

Don’t Add Edge





Add Edge



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

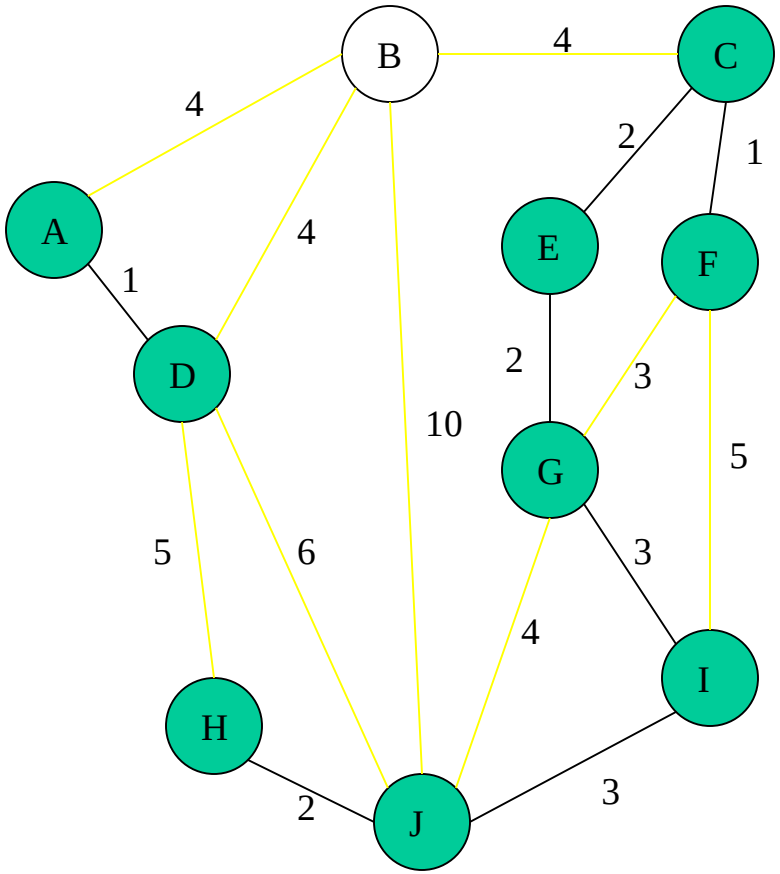
F 5 I

D 5 H

D 6 J

B 10 J

Add Edge



A 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

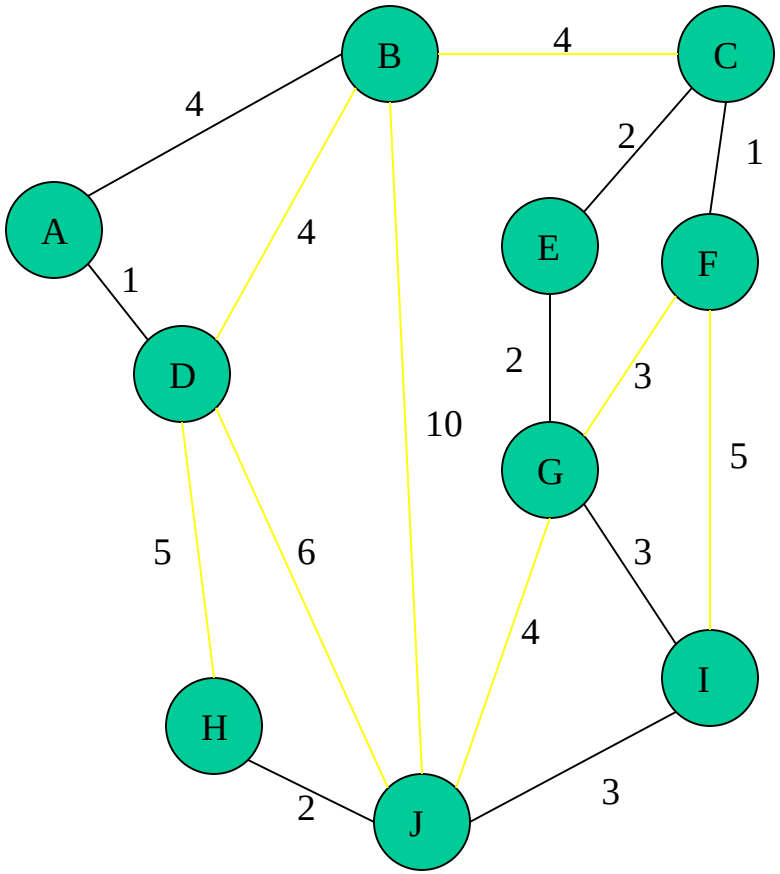
F 5 I

D 5 H

D 6 J

B 10 J

Add Edge



A 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

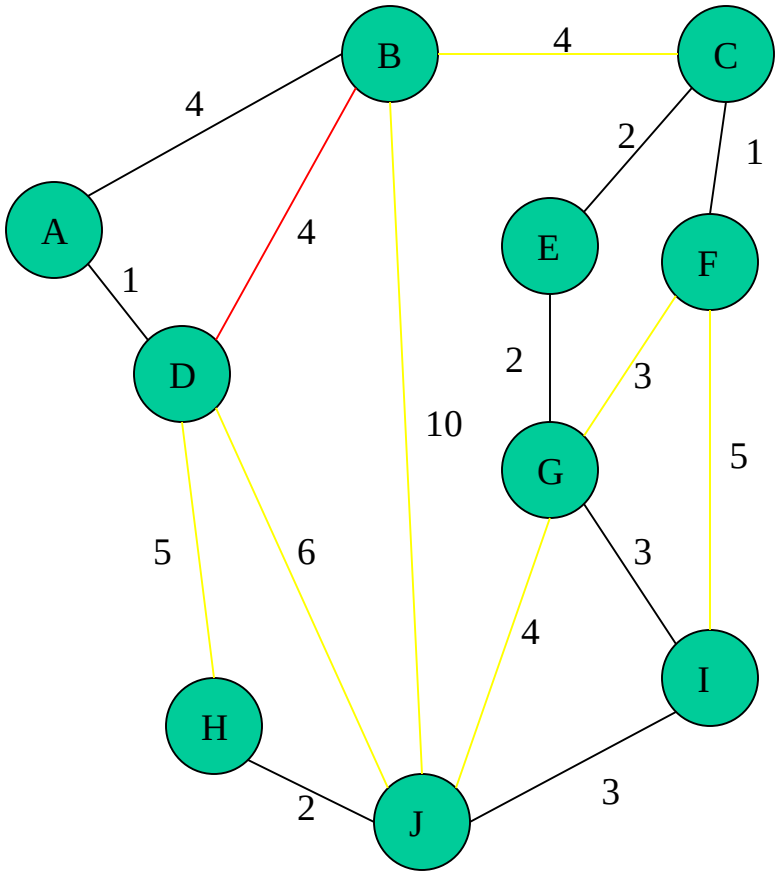
D 5 H

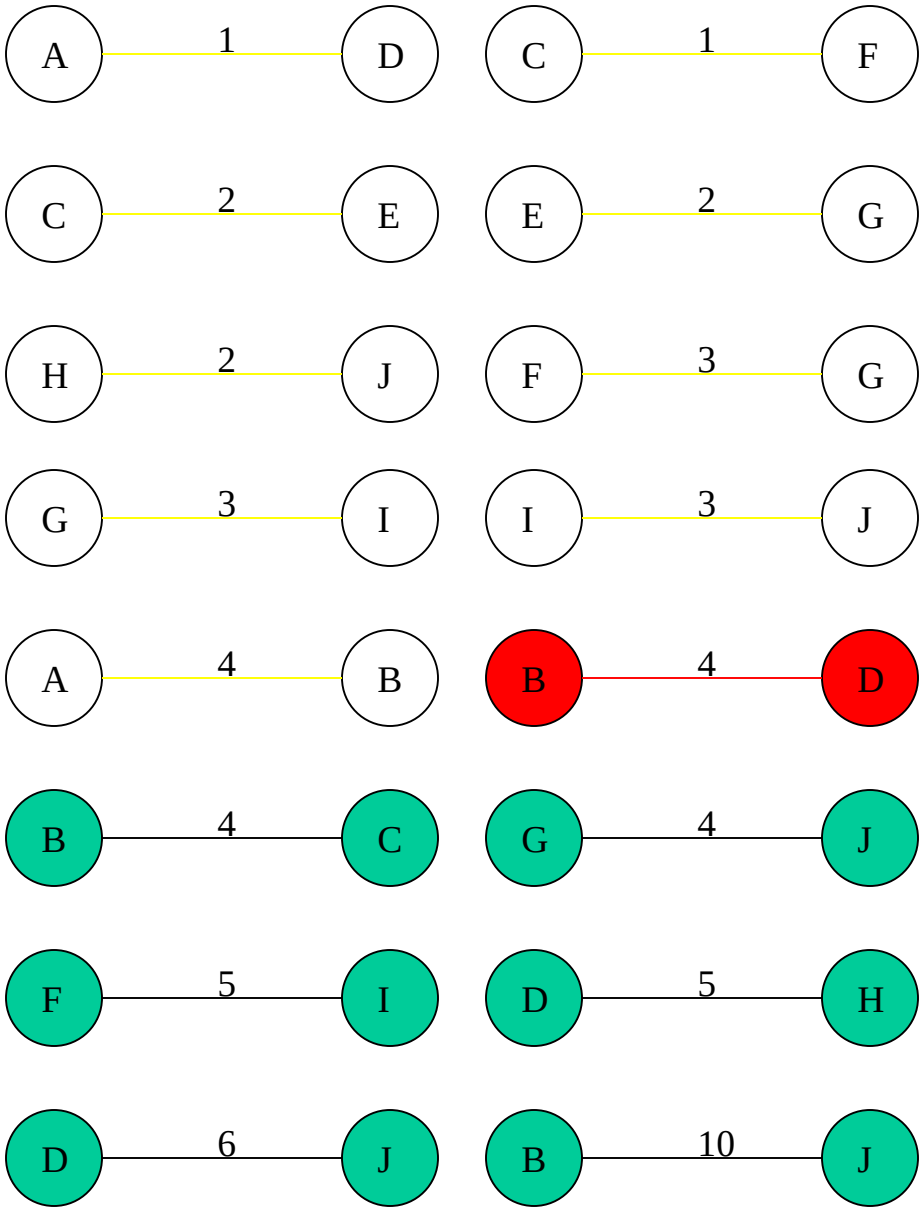
D 6 J

B 10 J

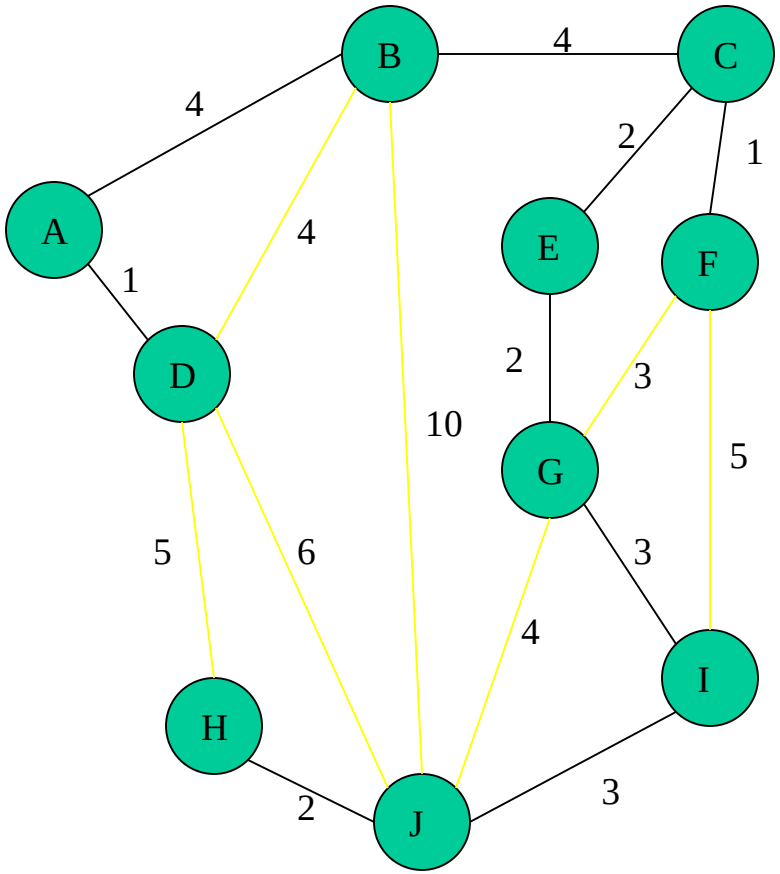
Cycle

Don’t Add Edge



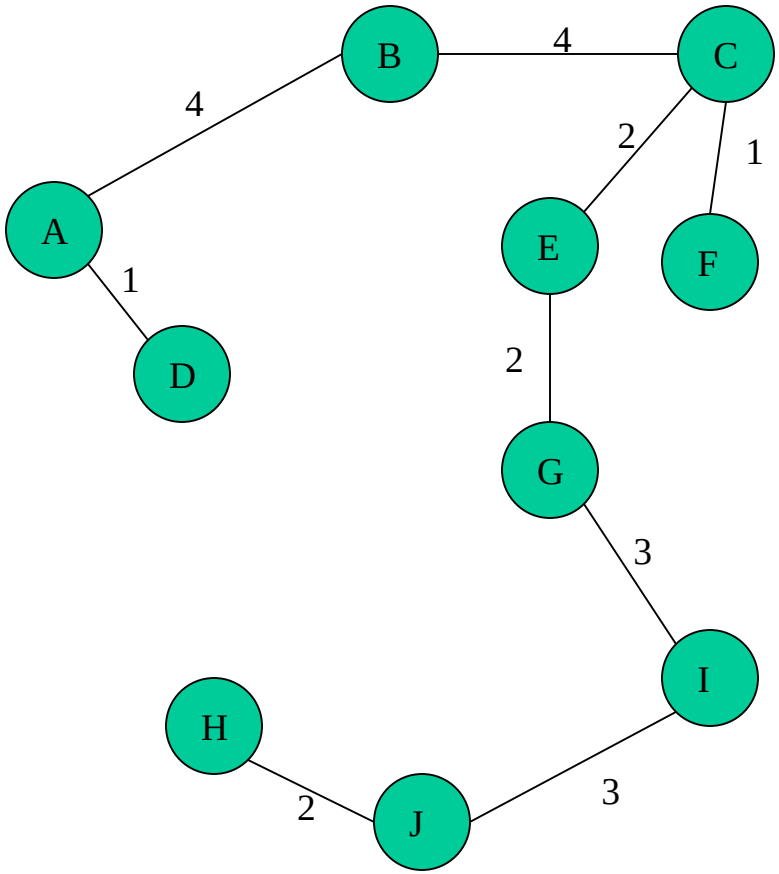


Add Edge

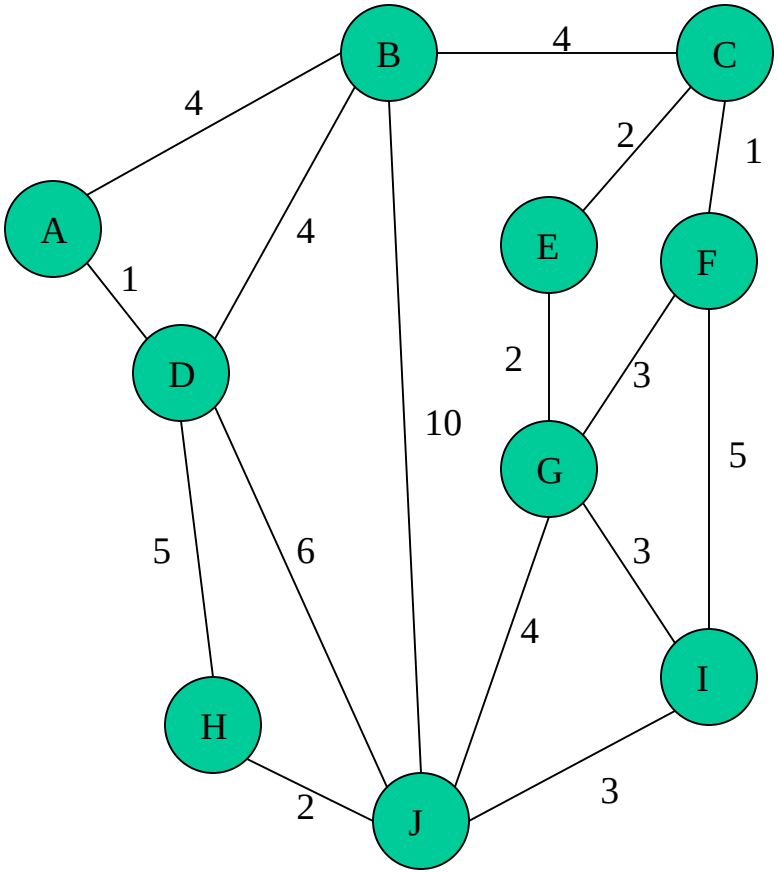


A	1	D	C	1	F
C	2	E	E	2	G
H	2	J	F	3	G
G	3	I	I	3	J
A	4	B	B	4	D
B	4	C	G	4	J
F	5	I	D	5	H
D	6	J	B	10	J

Minimum Spanning Tree



Complete Graph



Analysis of Kruskal's Algorithm

Running Time = $O(m \log n)$ (m = edges, n = 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.

Prim's Algorithm

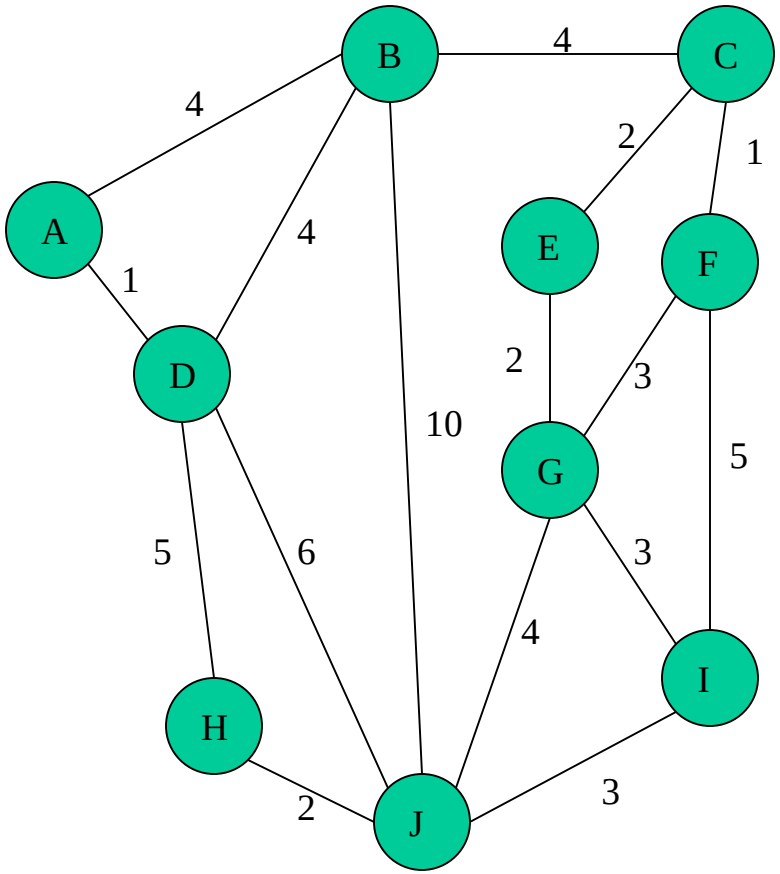
This algorithm starts with one node. It then, one by one, adds a node that is unconnected to the new graph, each time selecting the node whose connecting edge has the smallest weight out of the available nodes' connecting edges.

The steps are:

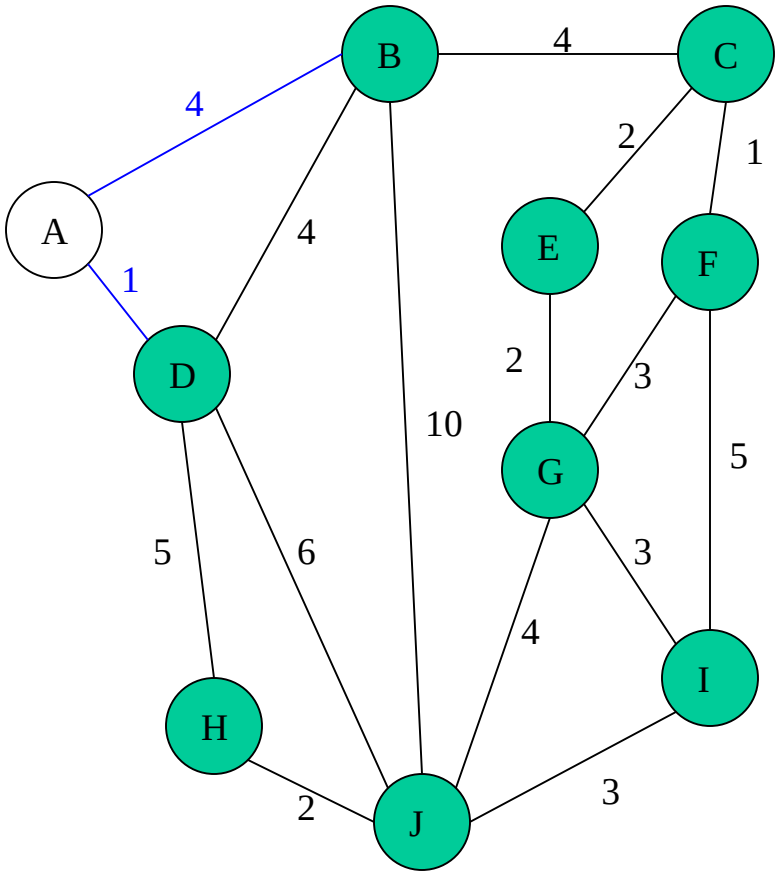
1. The new graph is constructed - with one node from the old graph.
2. While new graph has fewer than n nodes,
    1. Find the node from the old graph with the smallest connecting edge to the new graph,
    2. Add it to the new graph

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.

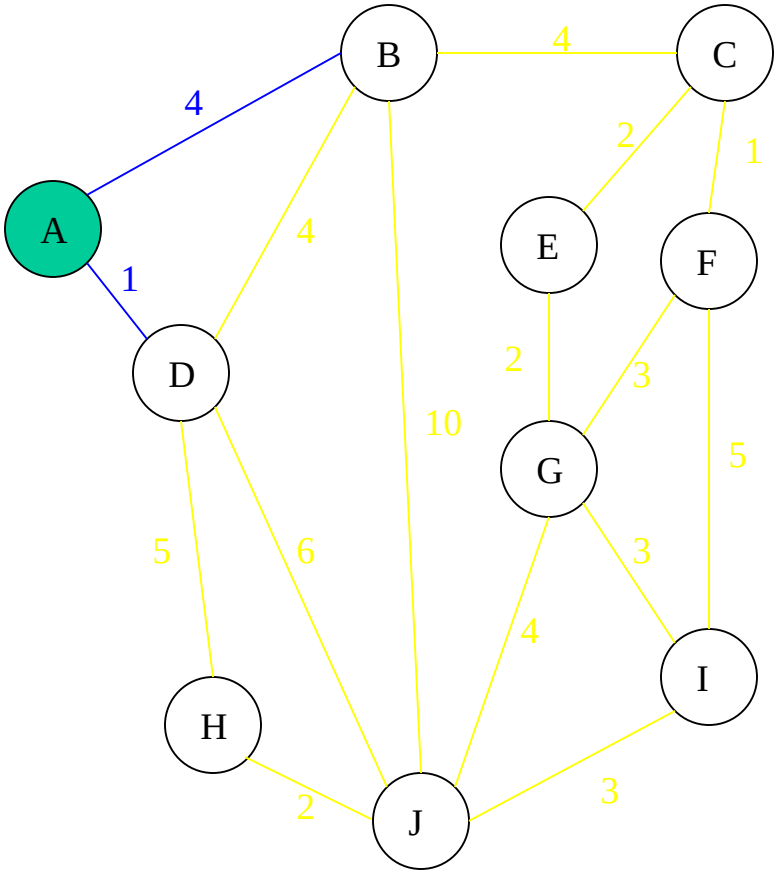
Complete Graph



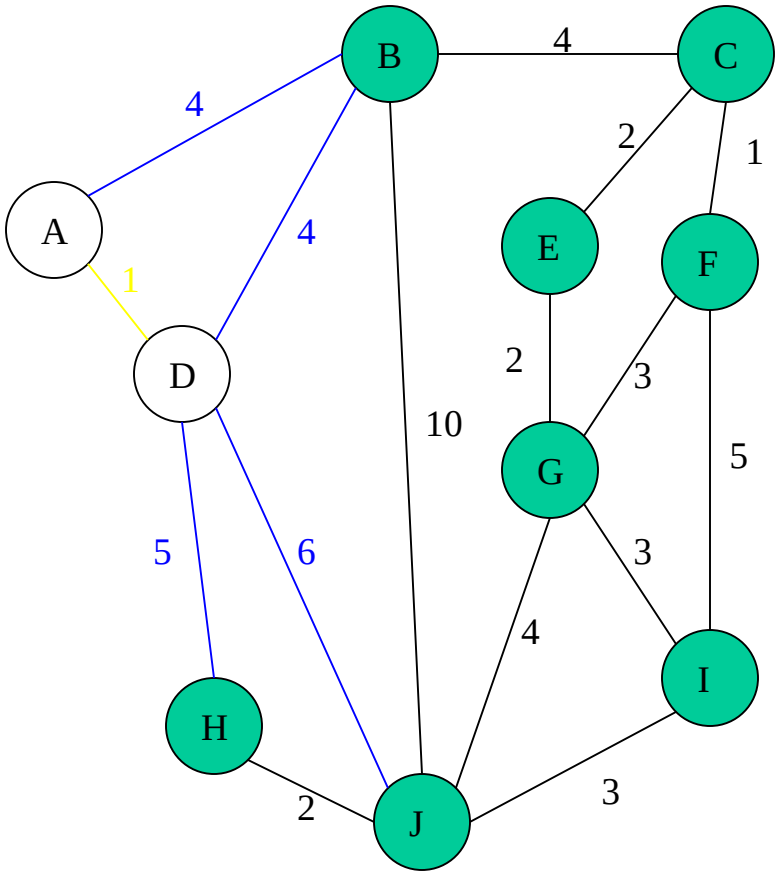
Old Graph



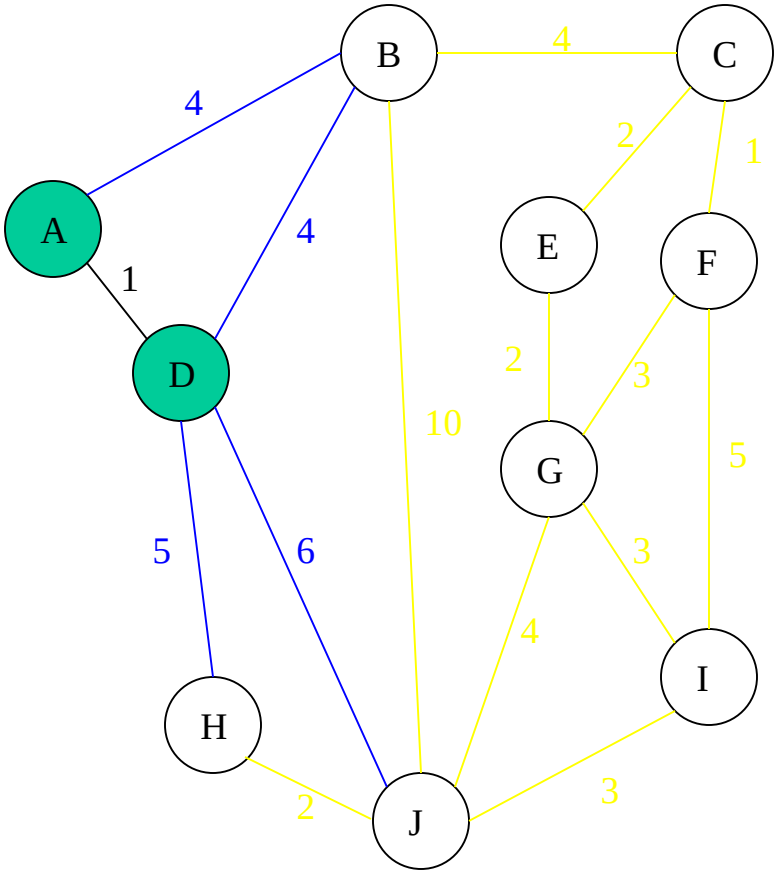
New Graph



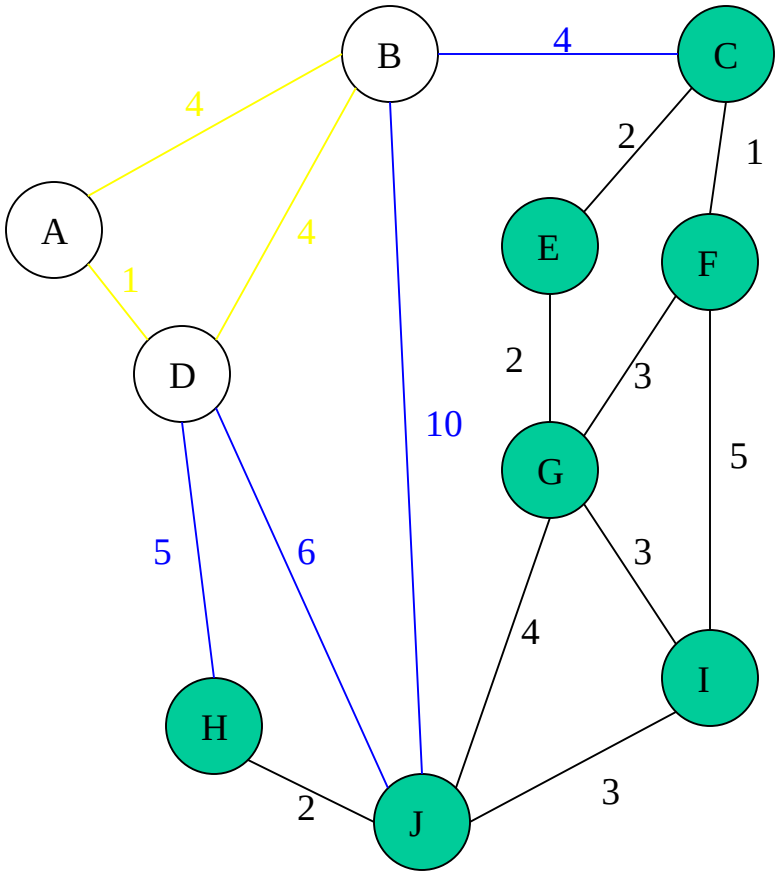
Old Graph



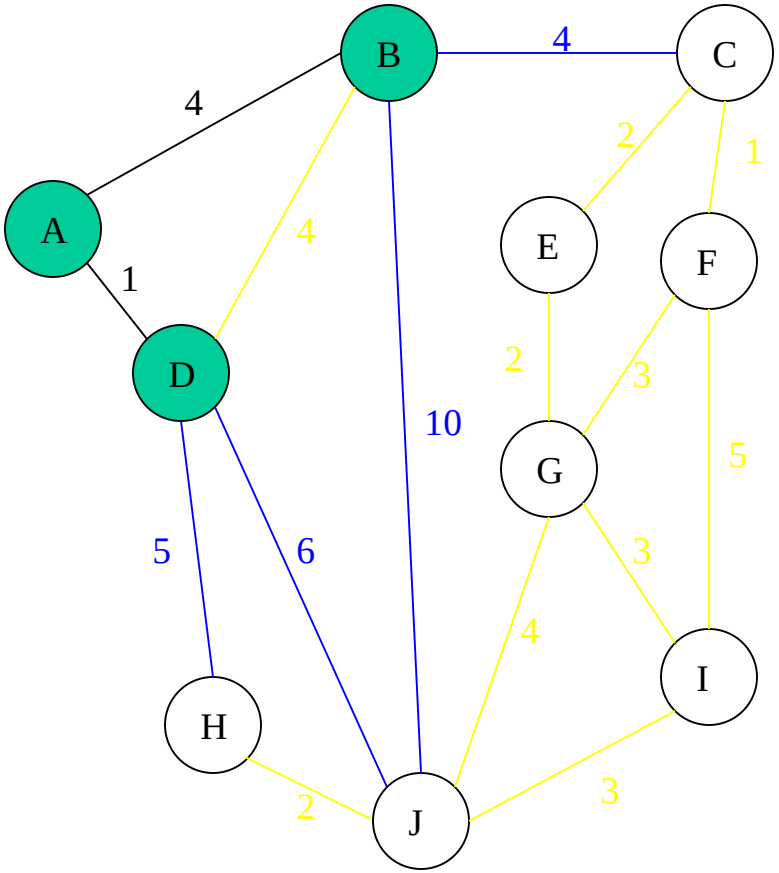
New Graph



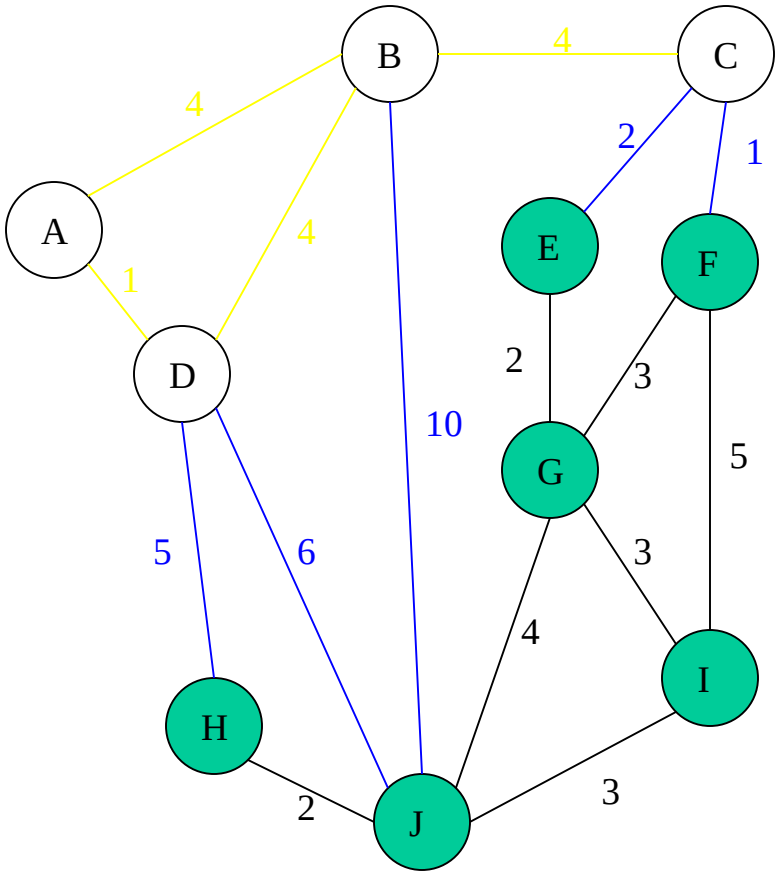
Old Graph



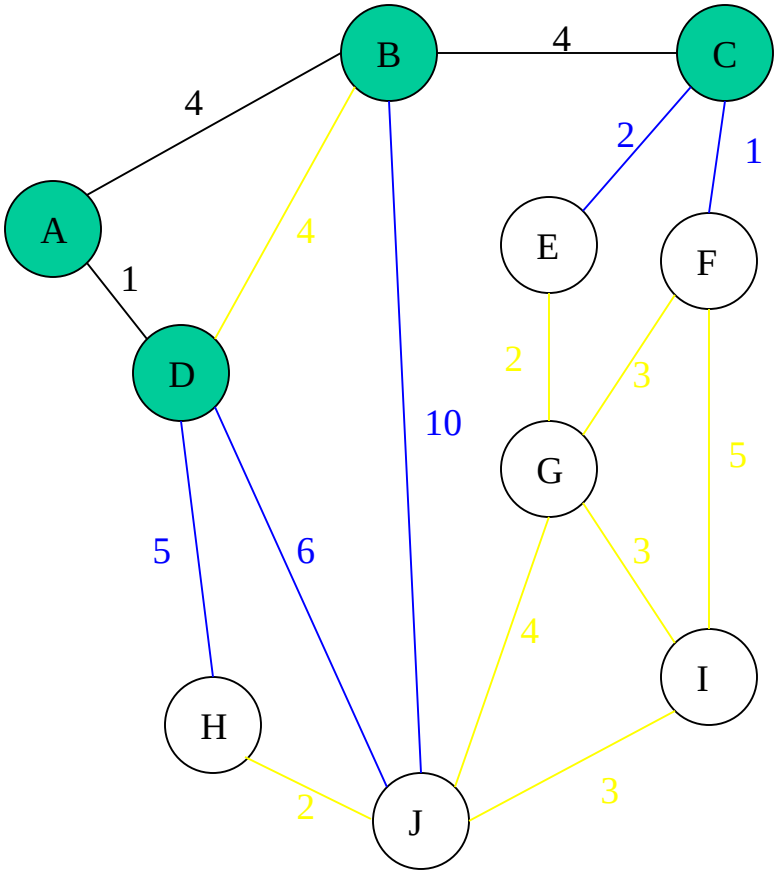
New Graph



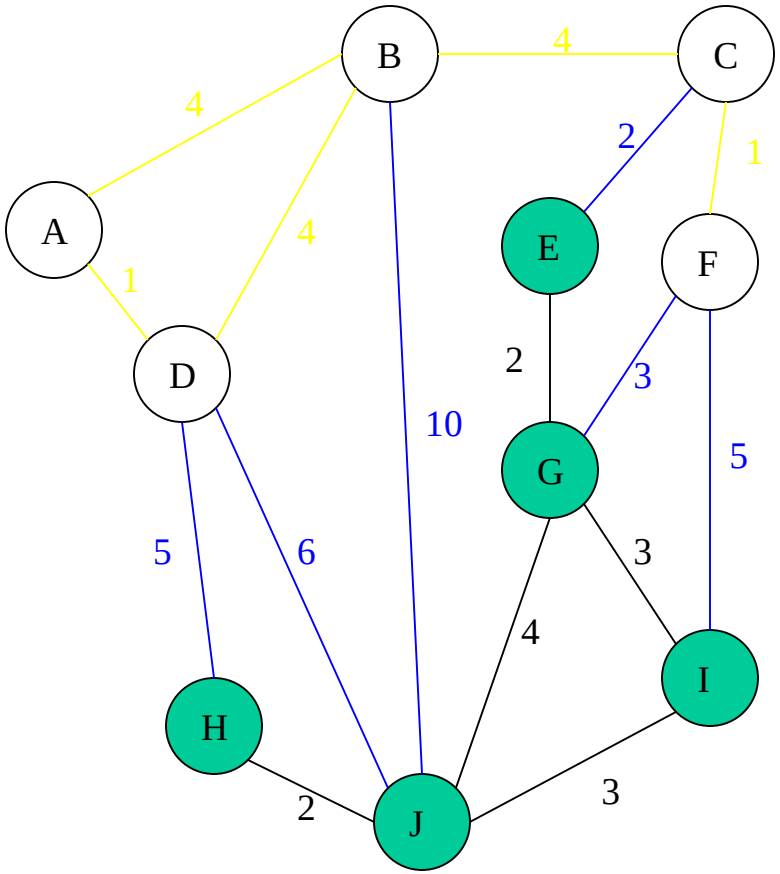
Old Graph



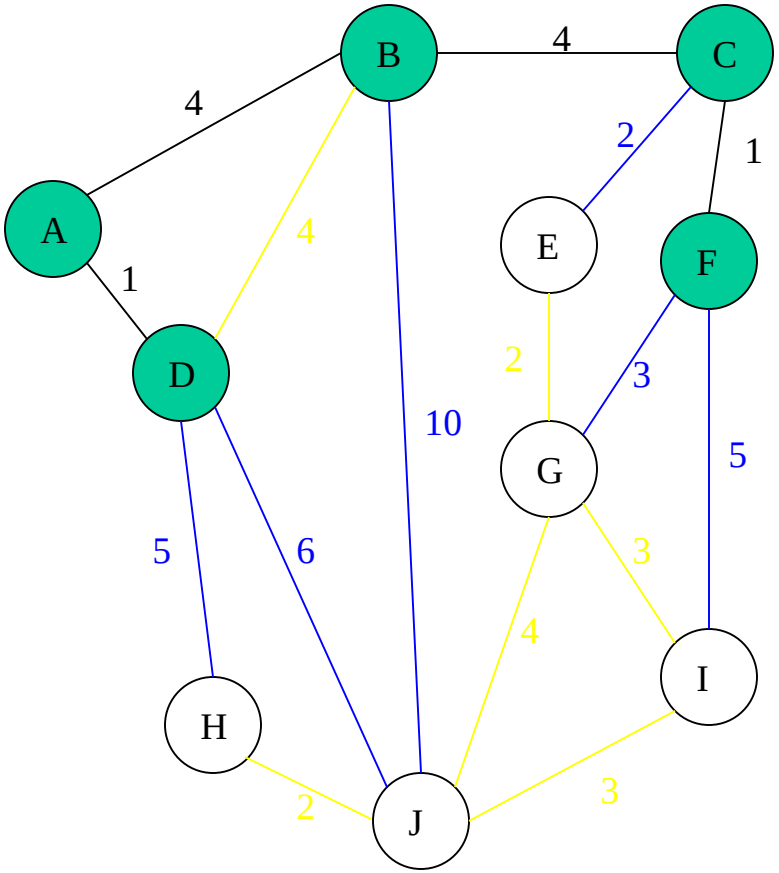
New Graph



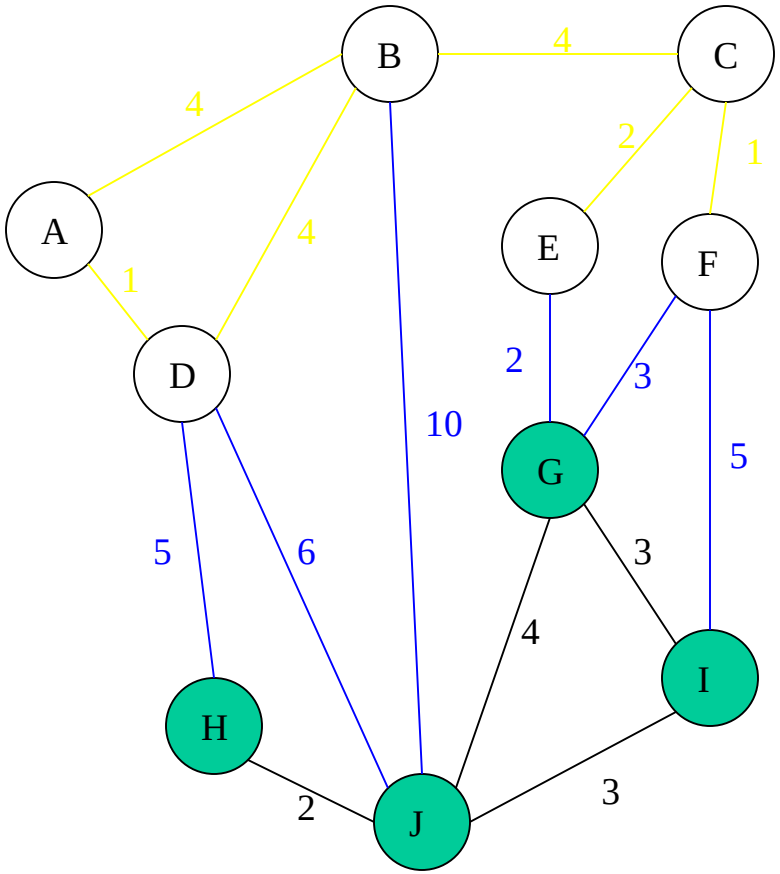
Old Graph



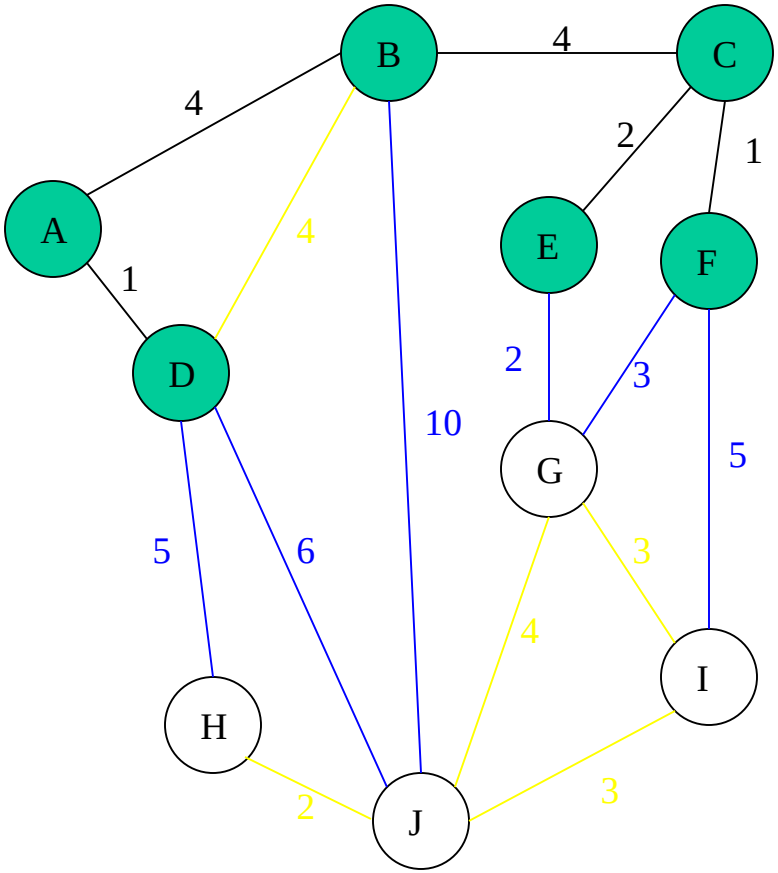
New Graph



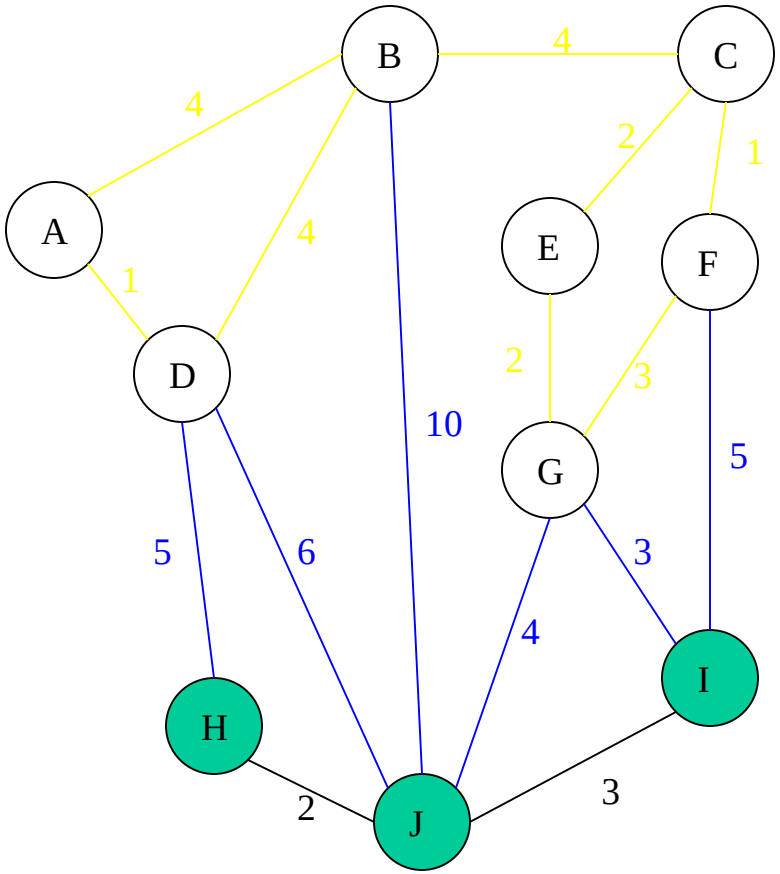
Old Graph



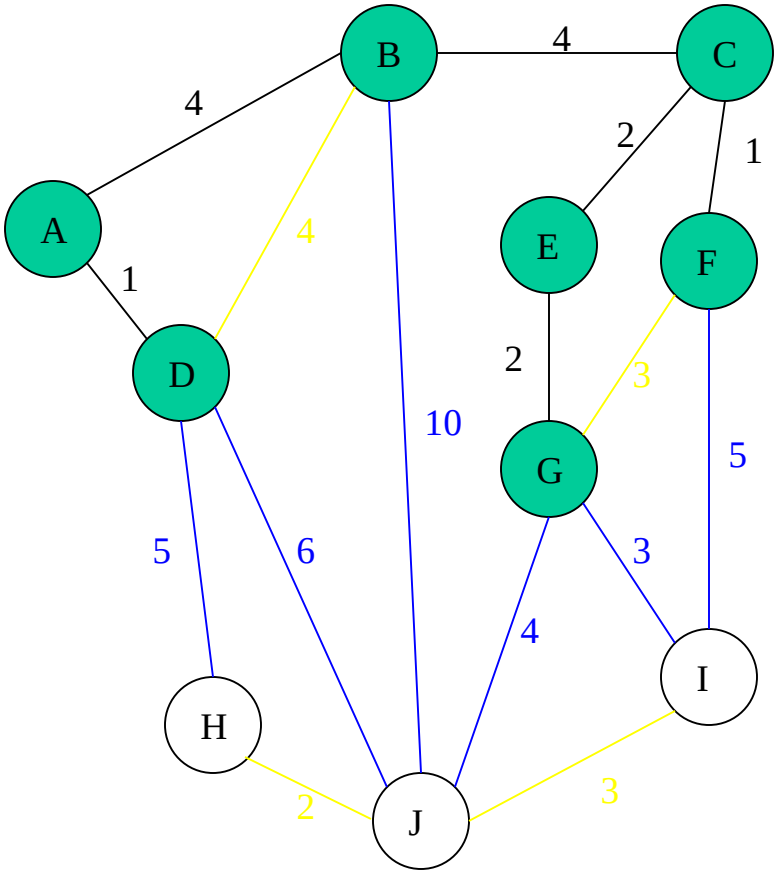
New Graph



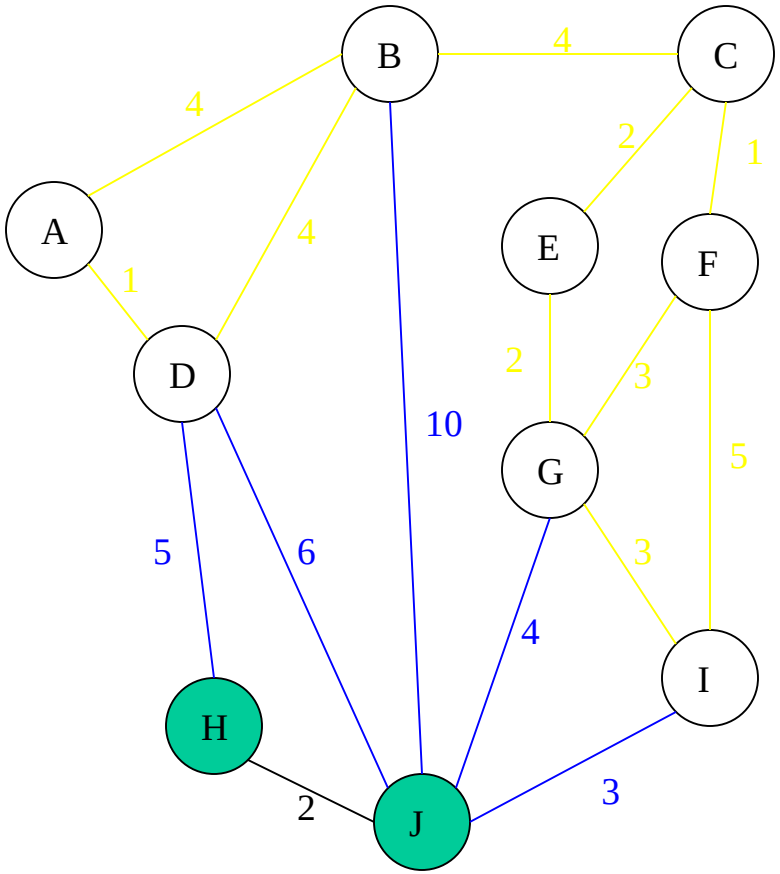
Old Graph



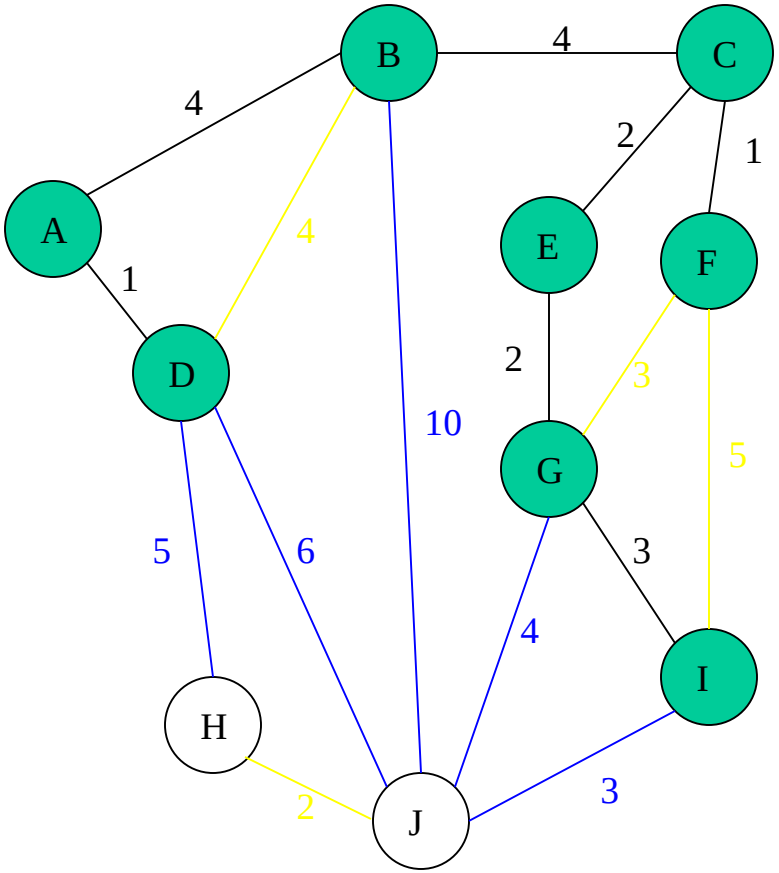
New Graph



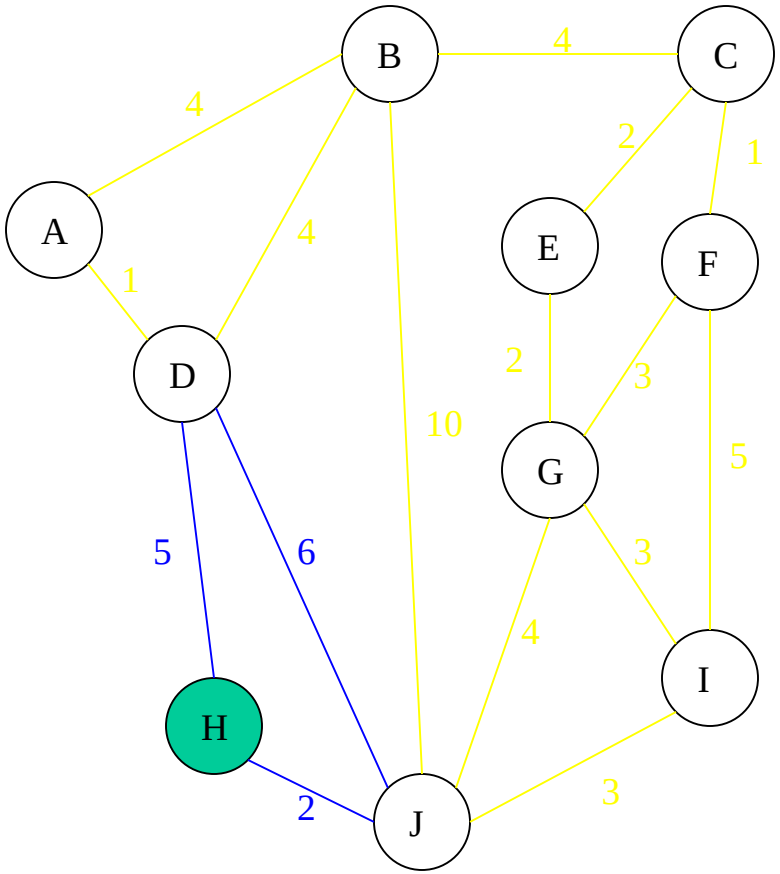
Old Graph



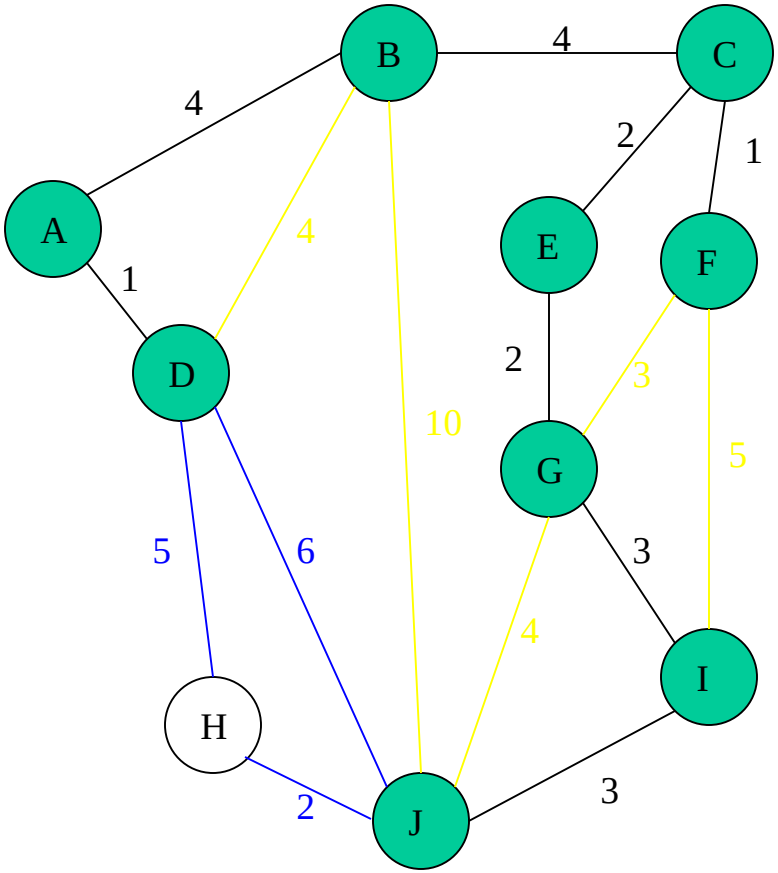
New Graph



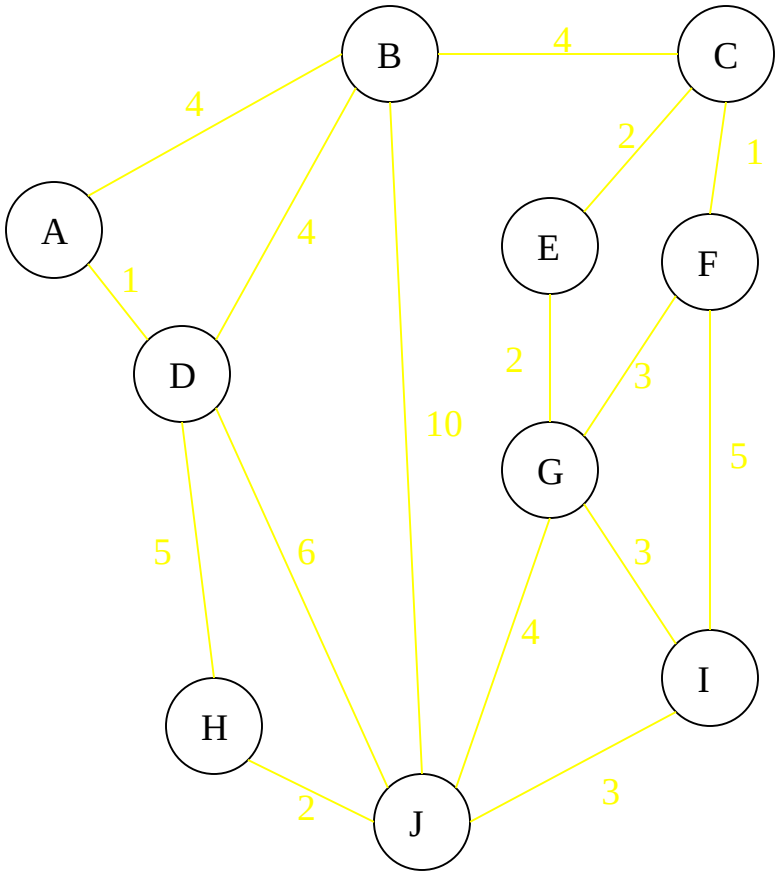
Old Graph



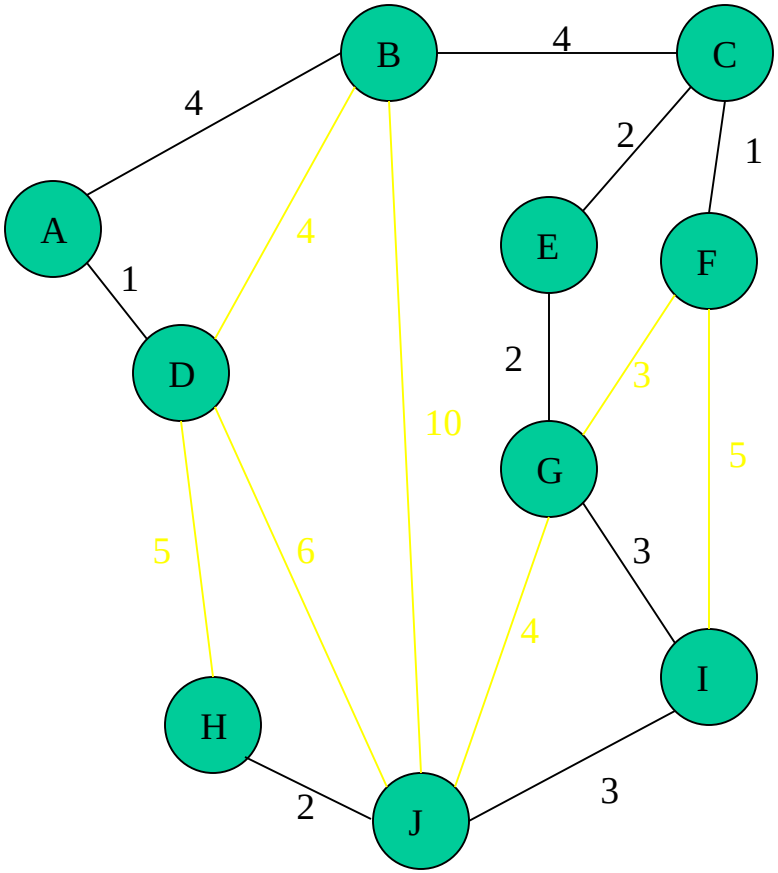
New Graph



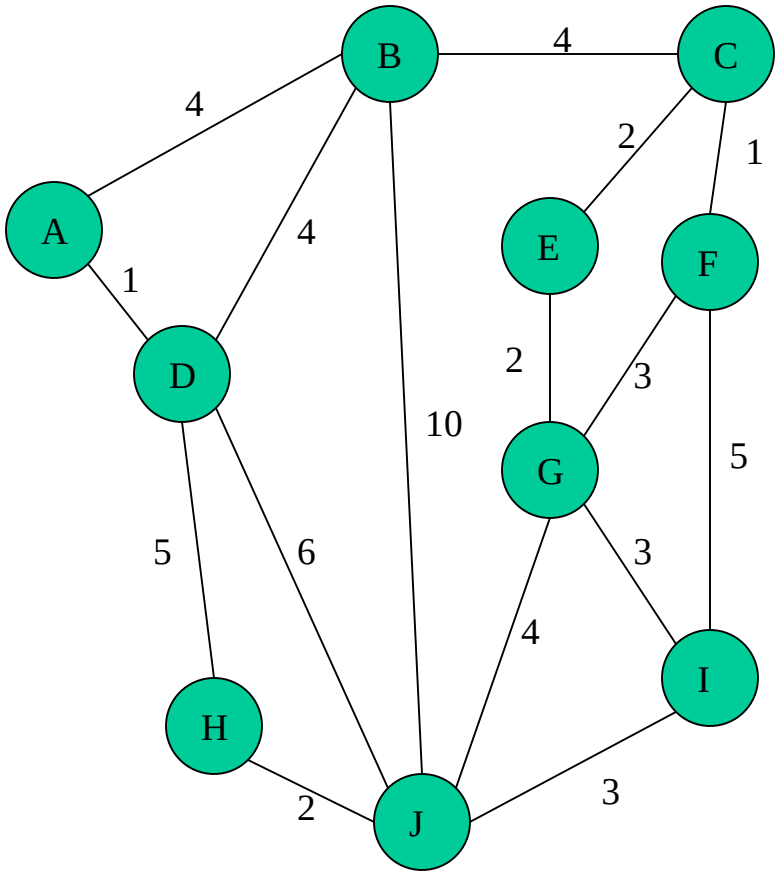
Old Graph



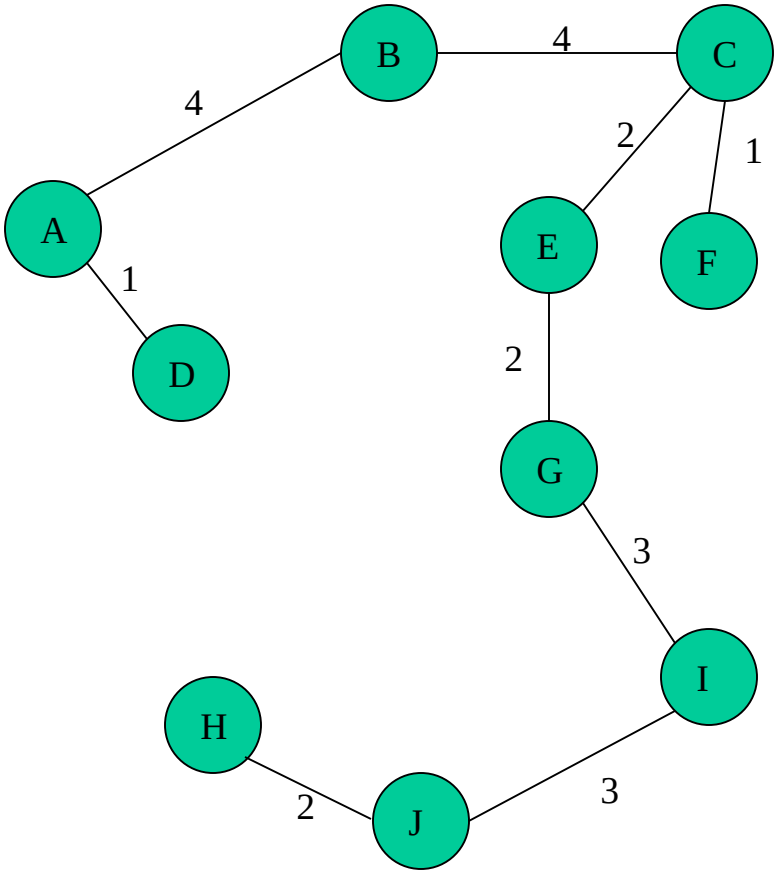
New Graph



Complete Graph



Minimum Spanning Tree



Analysis of Prim's Algorithm

Running Time = O(m + n log n) (m = edges, n = nodes)

If a heap is not used, the run time will be O(n^2) instead of O(m + n log n). However, using a heap complicates the code since you're complicating the data structure. A Fibonacci heap is the best kind of heap to use, but again, it complicates the code.

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.

Boruvka's Algorithm

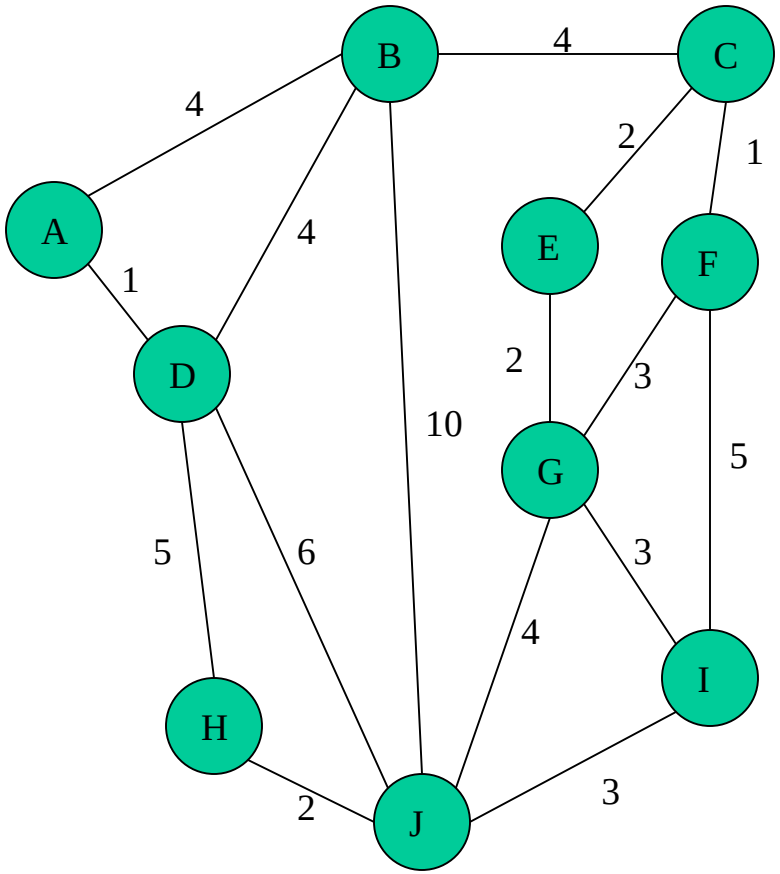
This algorithm is similar to Prim's, but nodes are added to the new graph in parallel all around the graph. It creates a list of trees, each containing one node from the original graph and proceeds to merge them along the smallest-weight connecting edges until there's only one tree, which is, of course, the MST. It works rather like a merge sort.

The steps are:

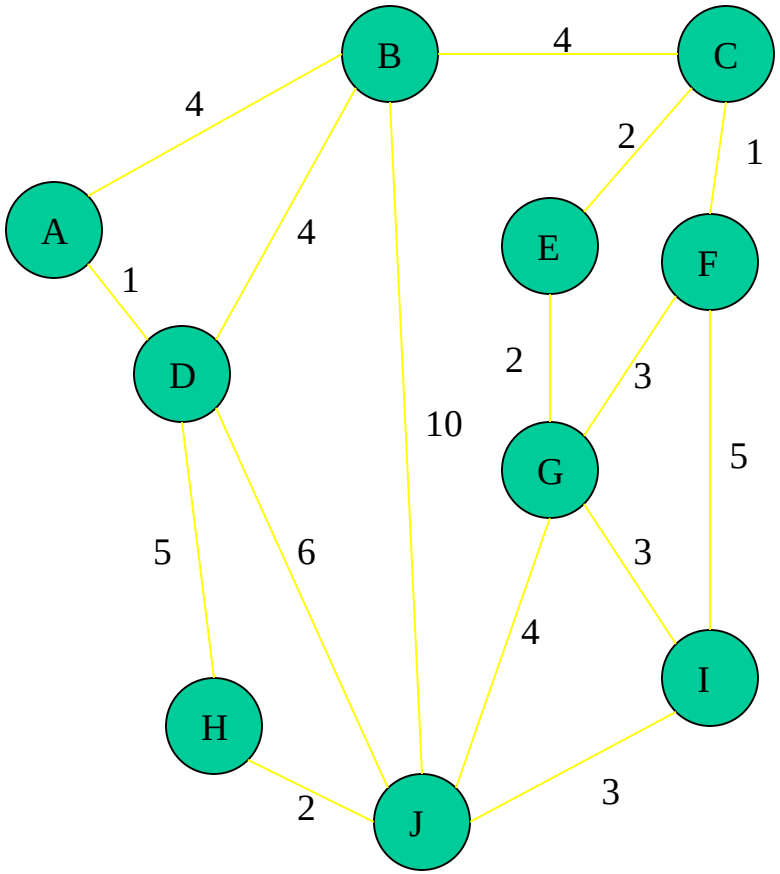
1. Make a list of n trees, each containing a single node
2. While list has more than one tree,
    1. For each tree in the list, find the node not connected to the tree with the smallest connecting edge to that tree,
    2. Add all the edges found to the new graph, thus creating a new set of trees

Every step will have joined groups of trees, until only one tree remains.

Complete Graph



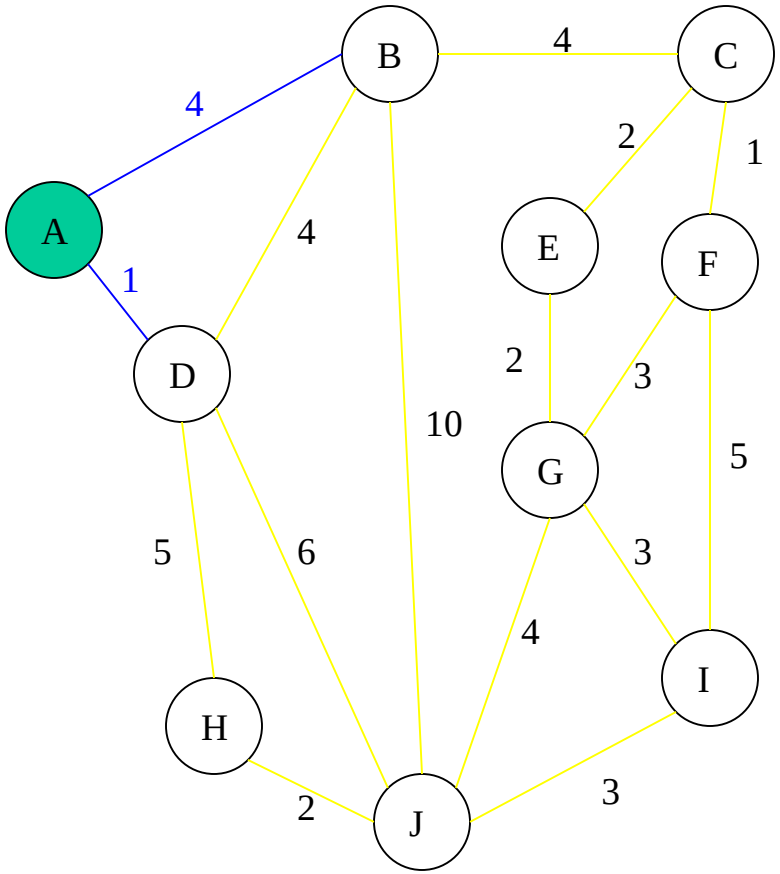
Trees of the Graph at Beginning of Round 1



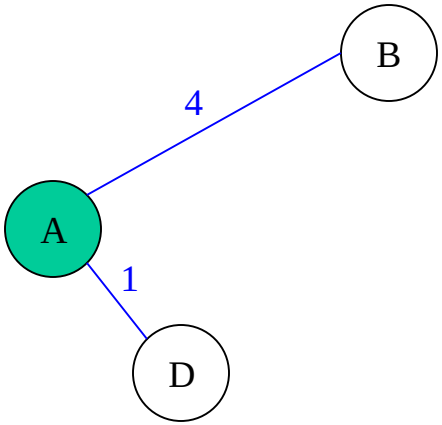
List of Trees

- A
- B
- C
- D
- E
- F
- G
- H
- I
- J

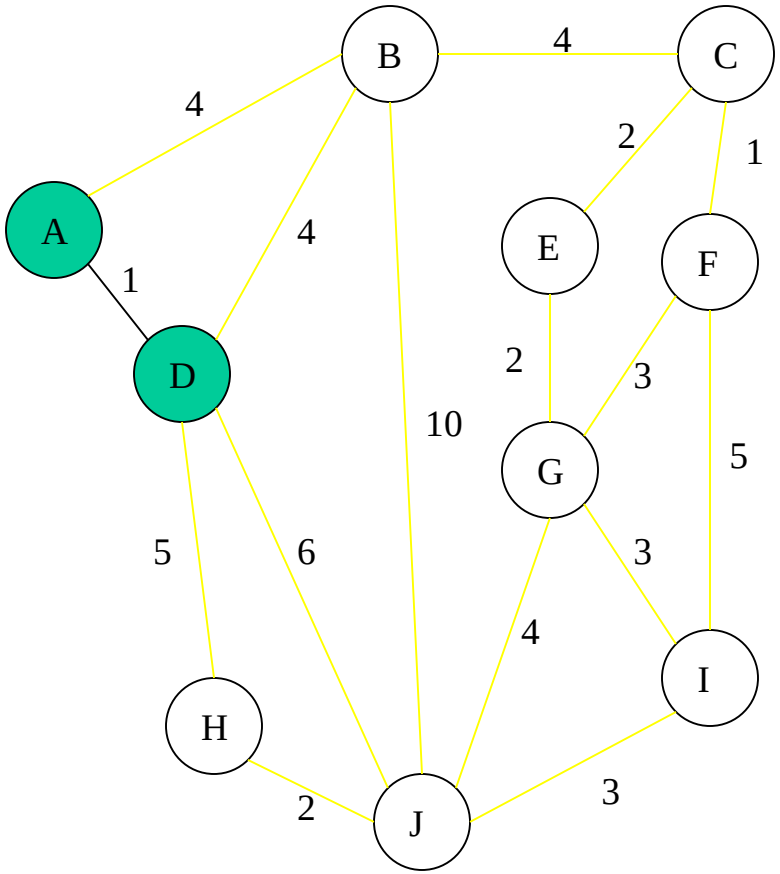
Round 1



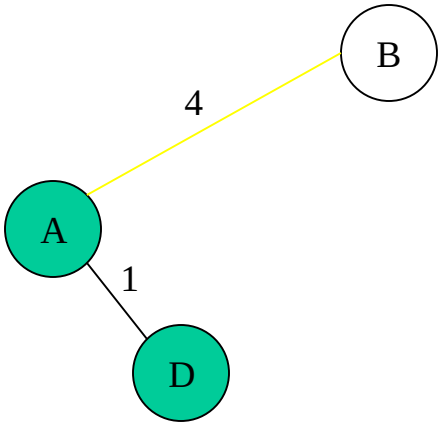
Tree A



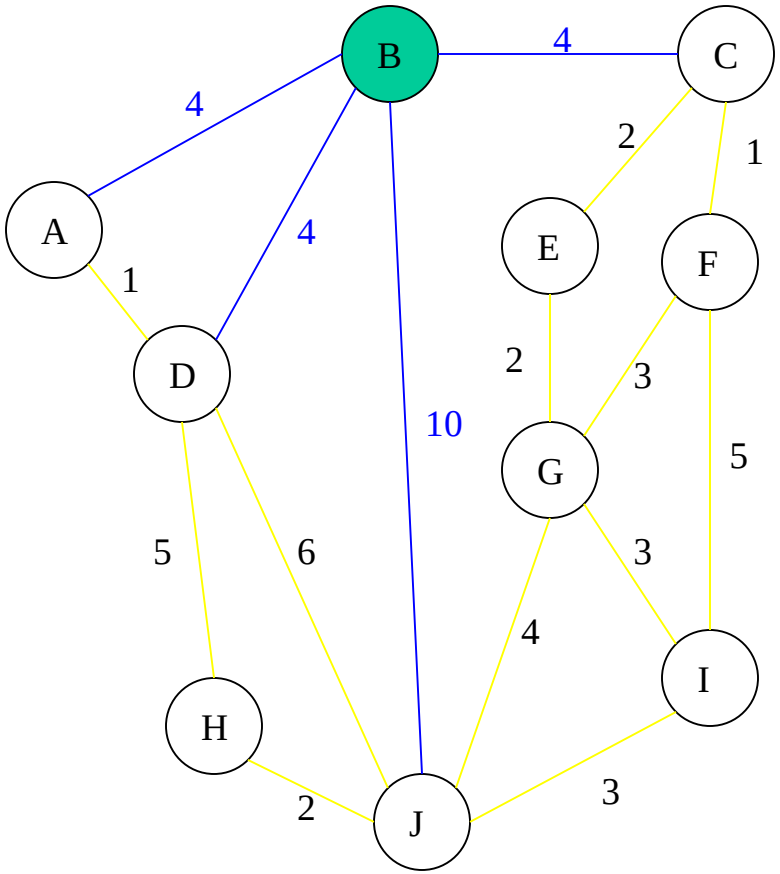
Round 1



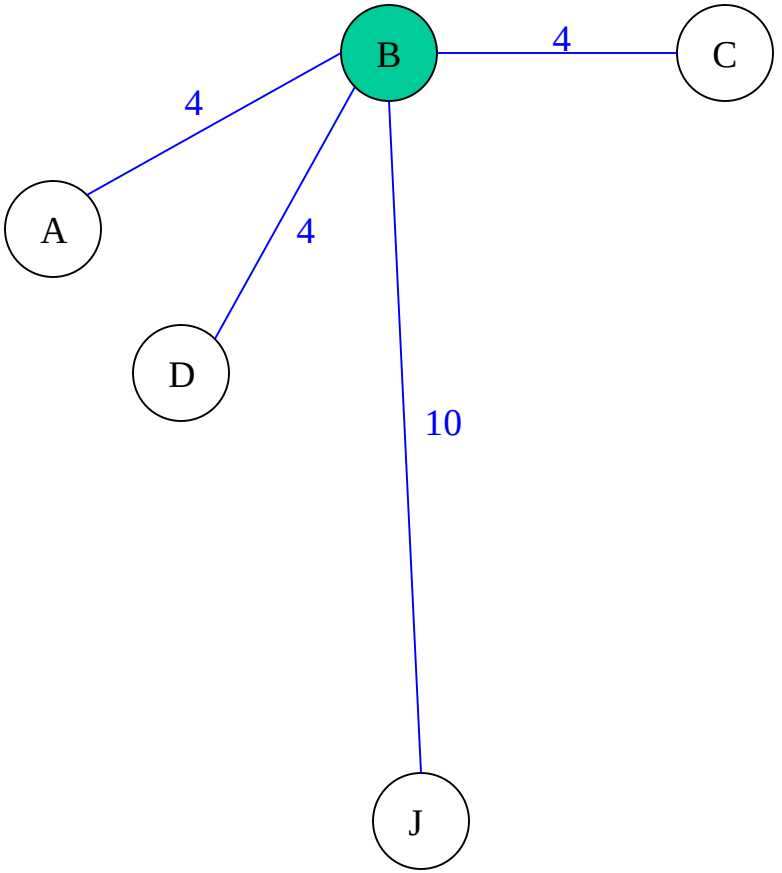
Edge A-D



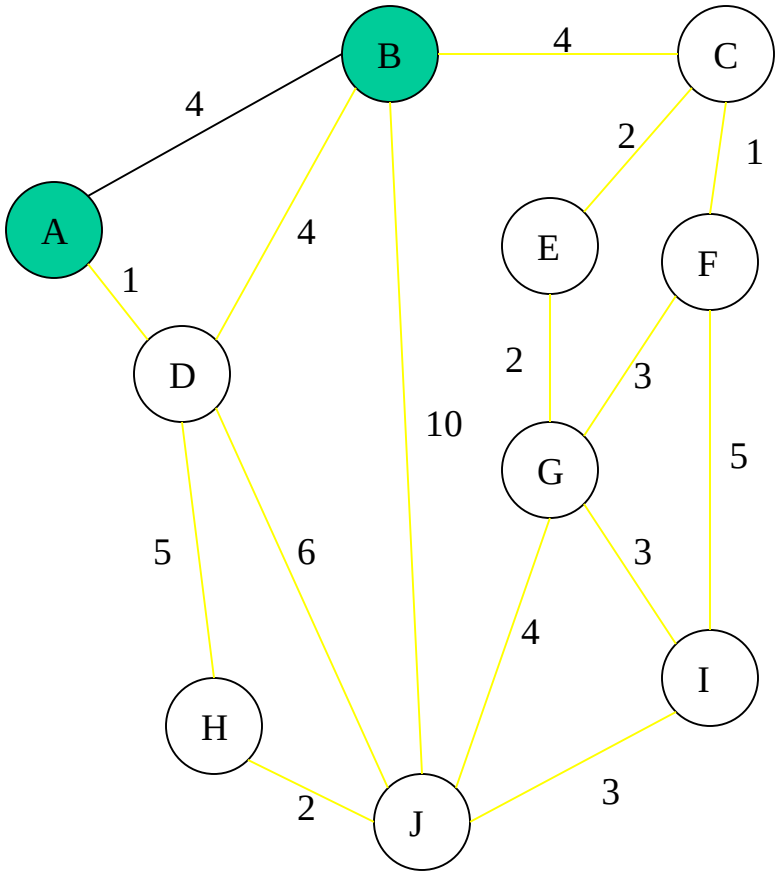
Round 1



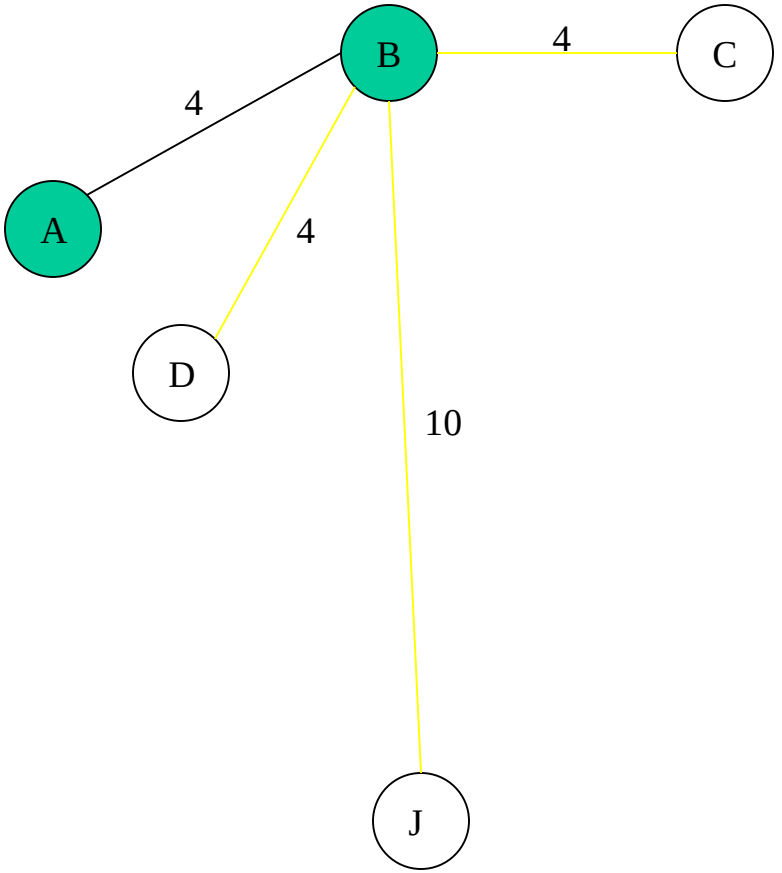
Tree B



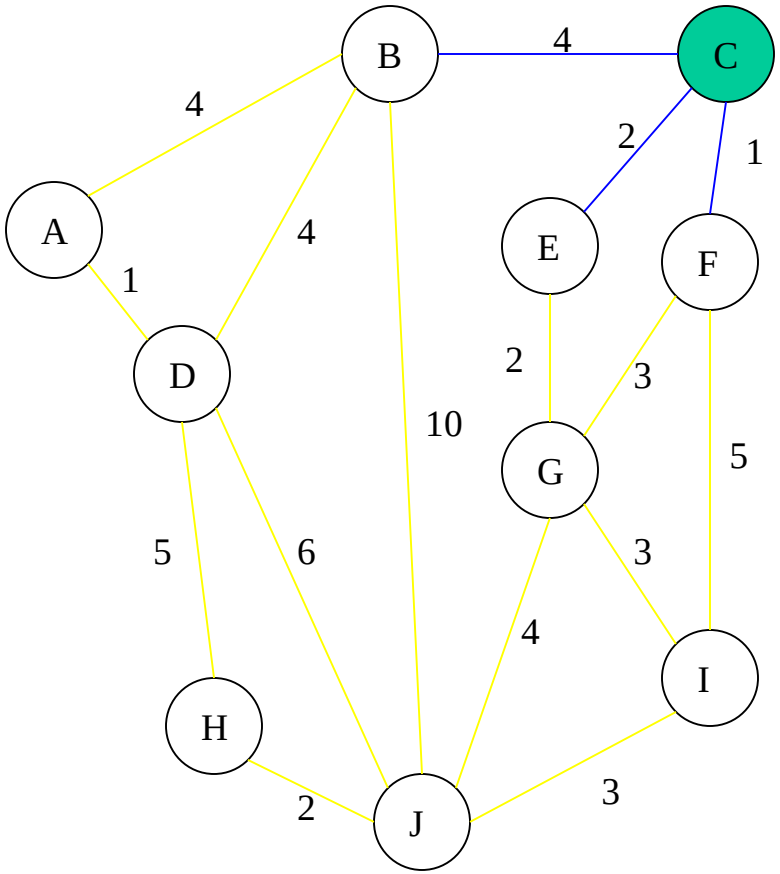
Round 1



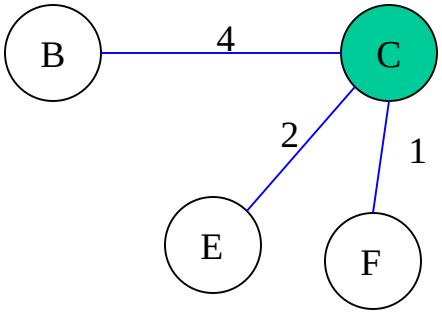
Edge B-A



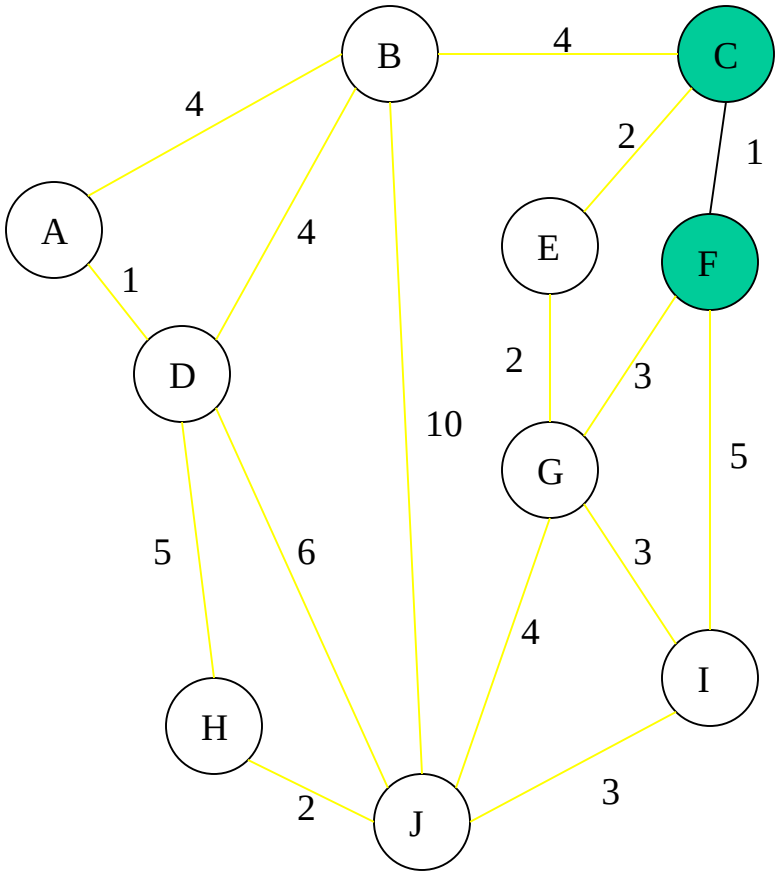
Round 1



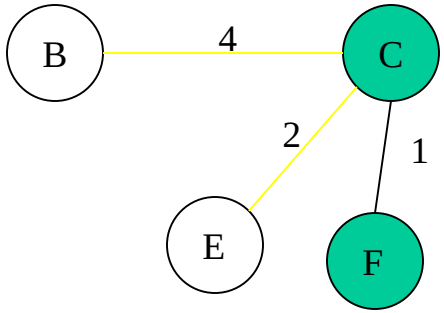
Tree C



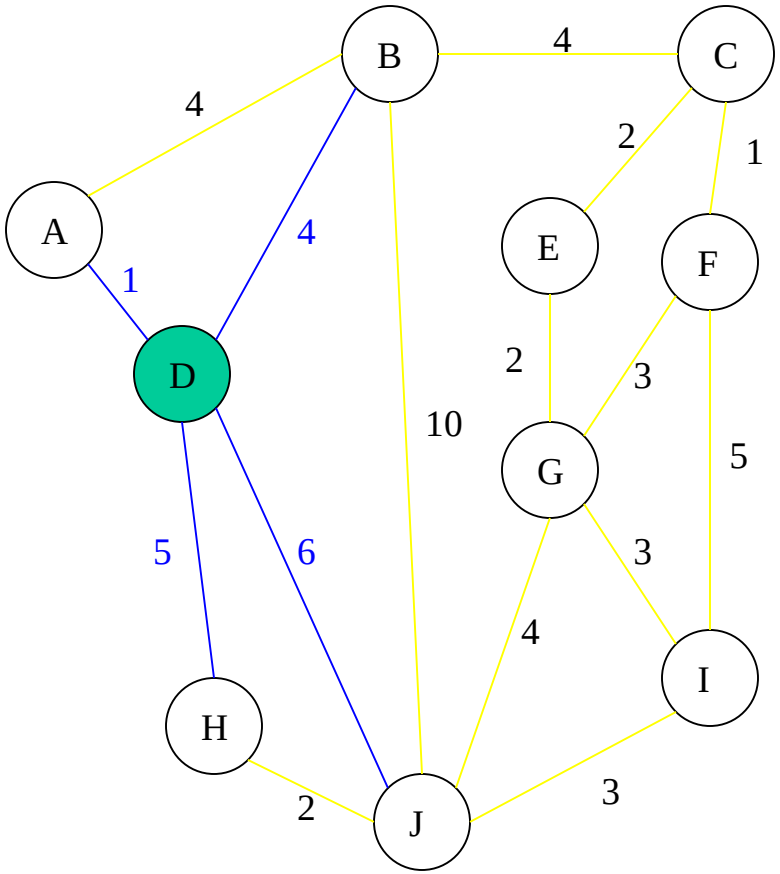
Round 1



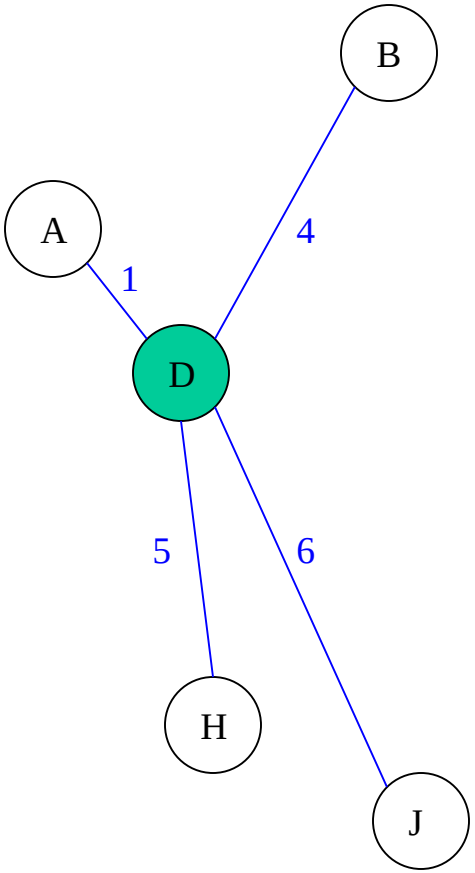
Edge C-F



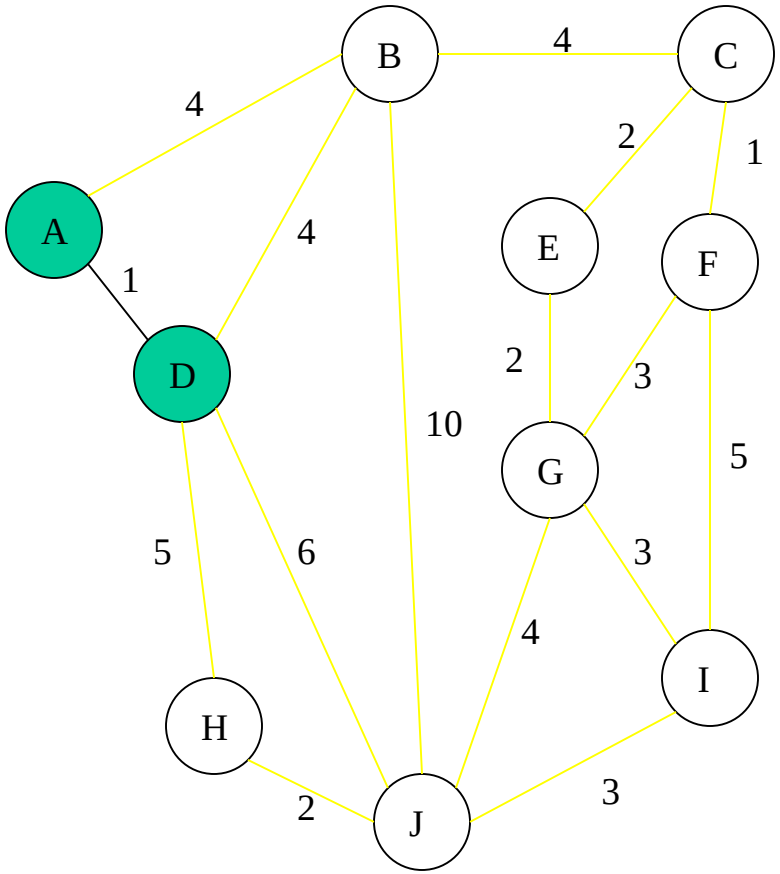
Round 1



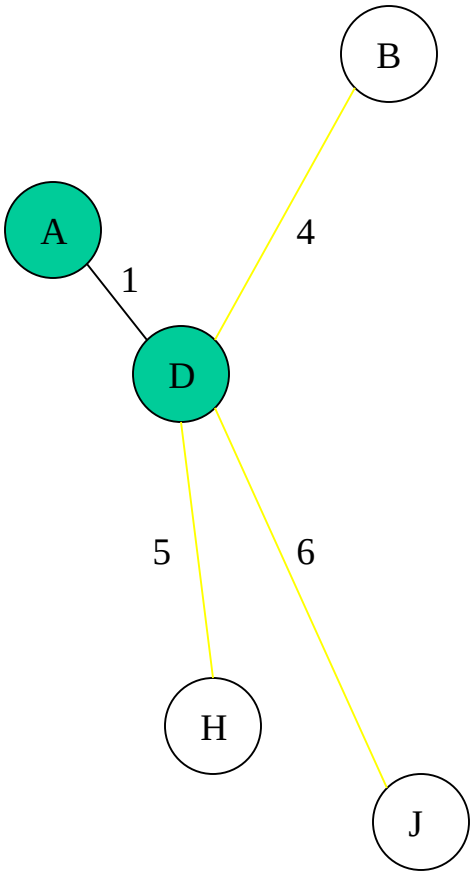
Tree D



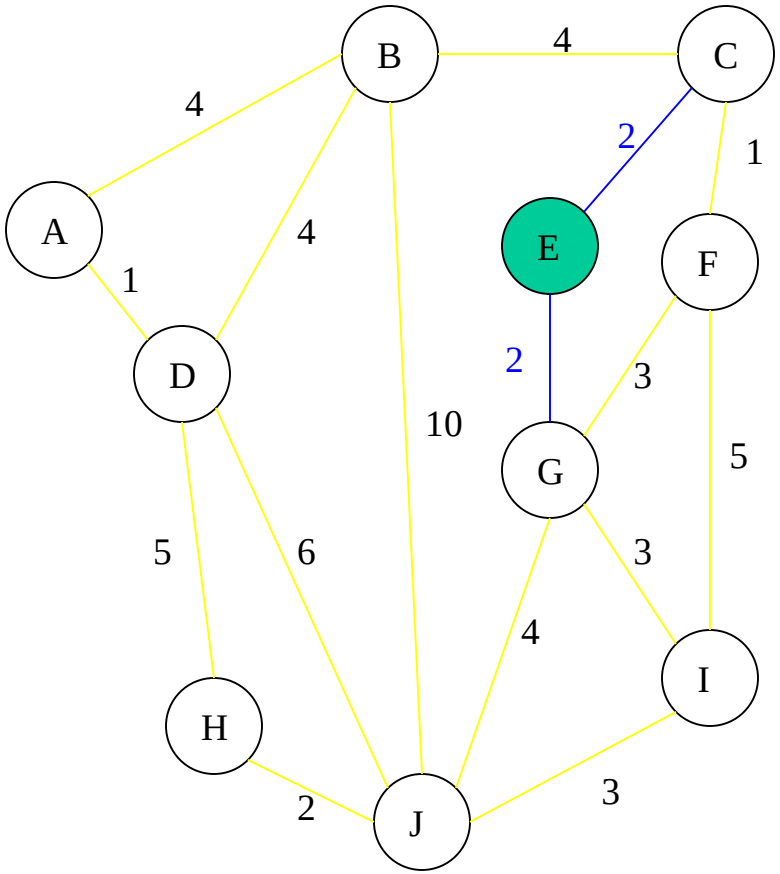
Round 1



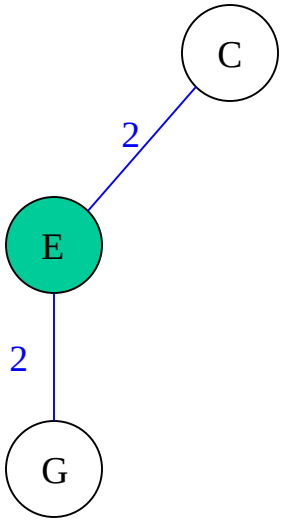
Edge D-A



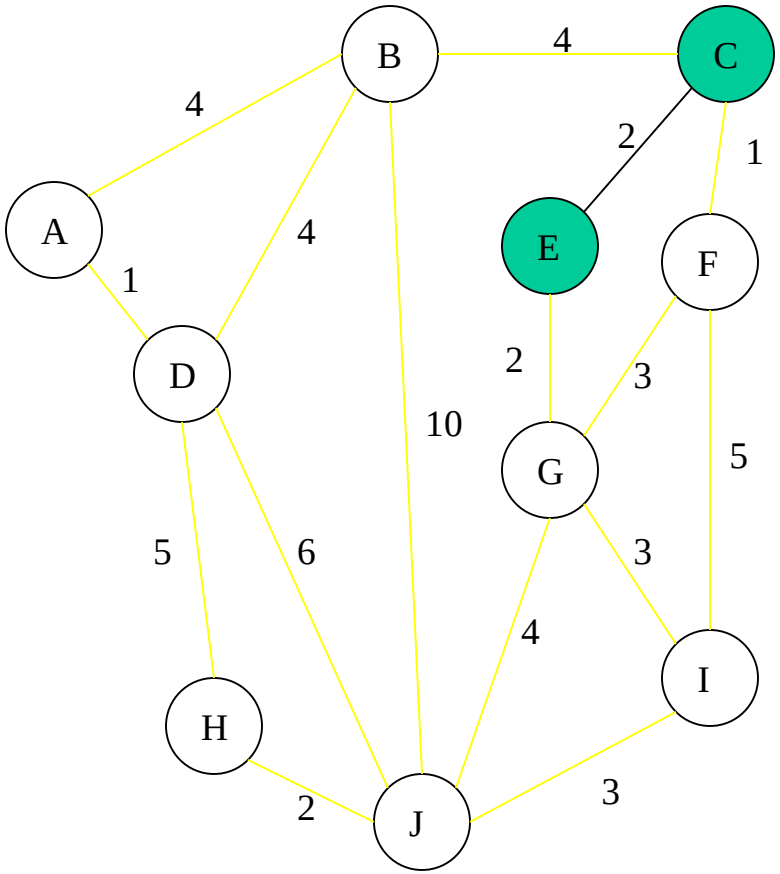
Round 1



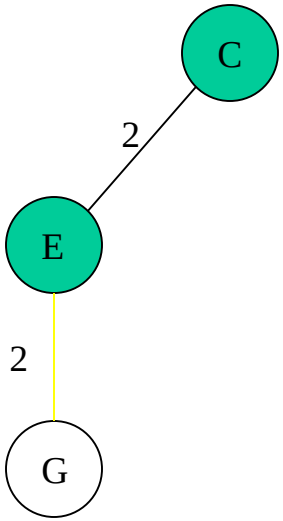
Tree E



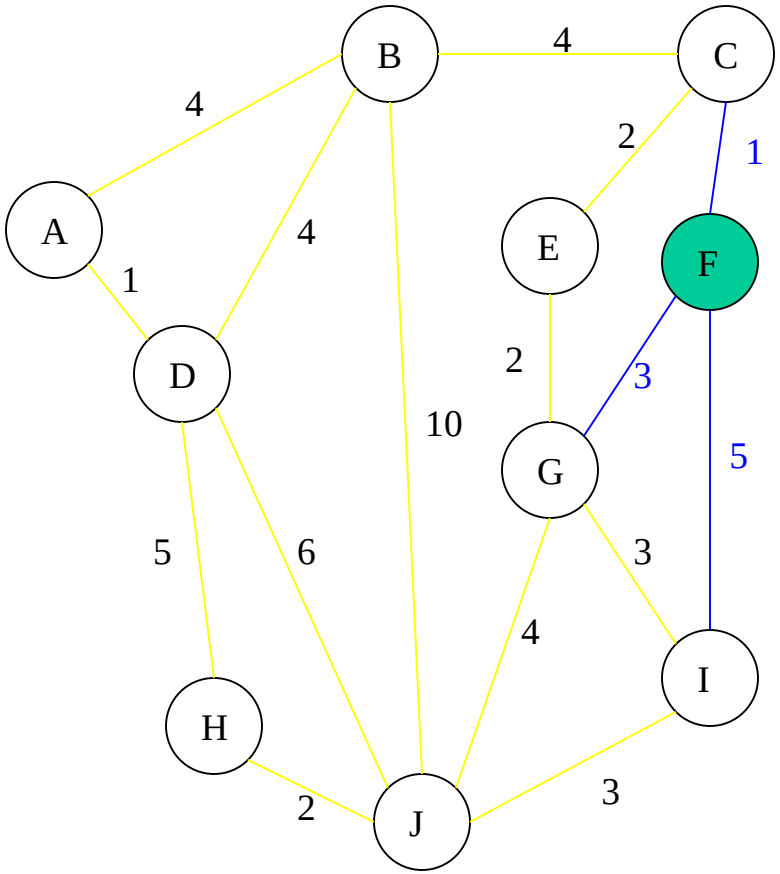
Round 1



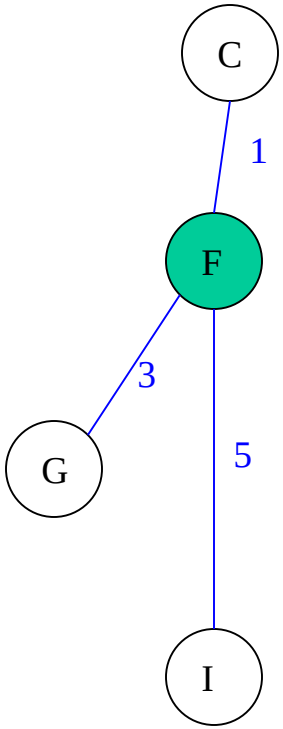
Edge E-C



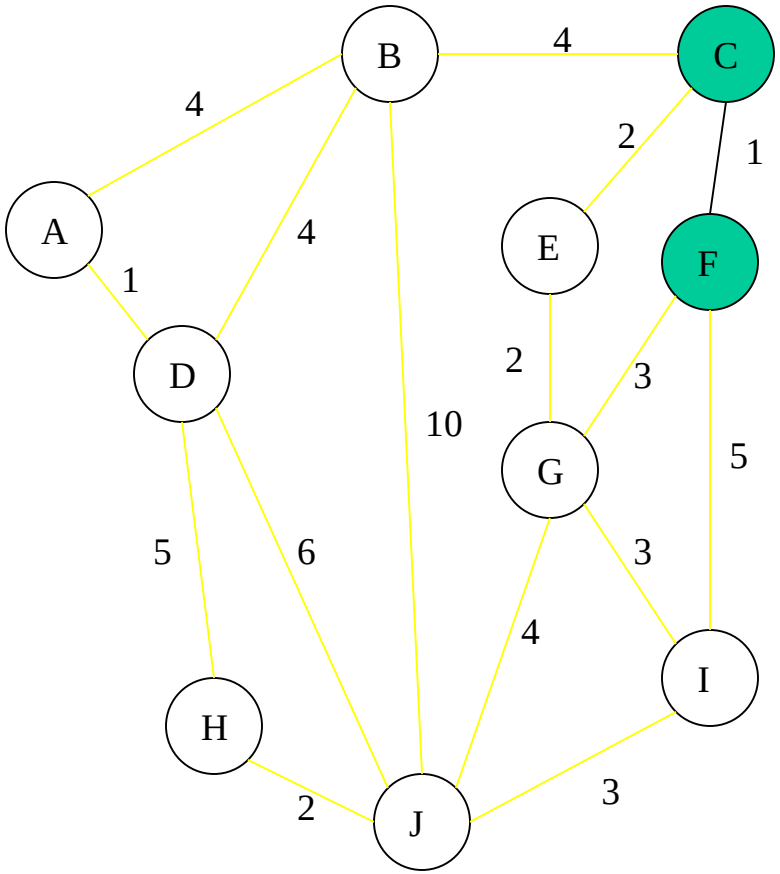
Round 1



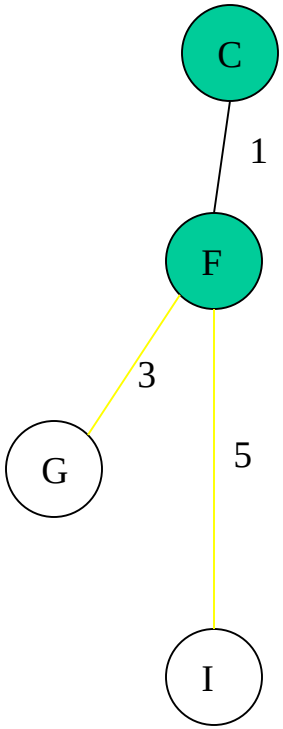
Tree F



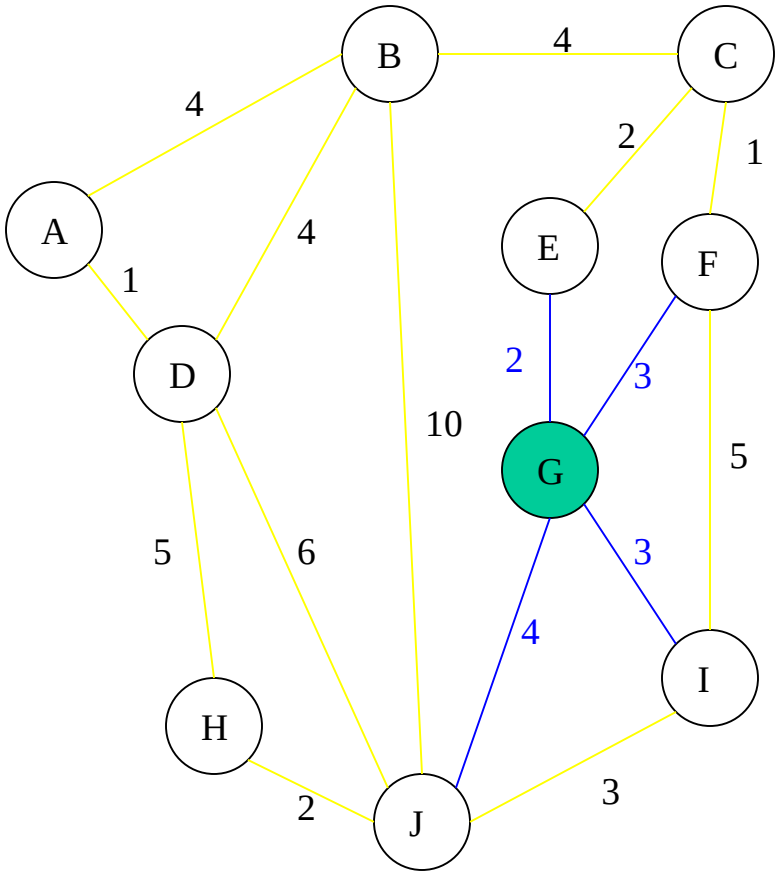
Round 1



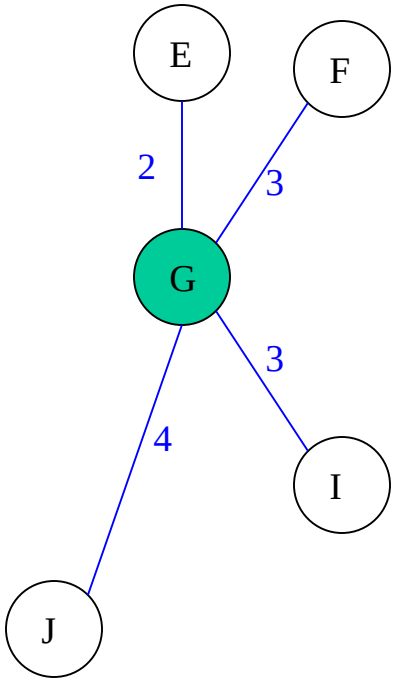
Edge F-C



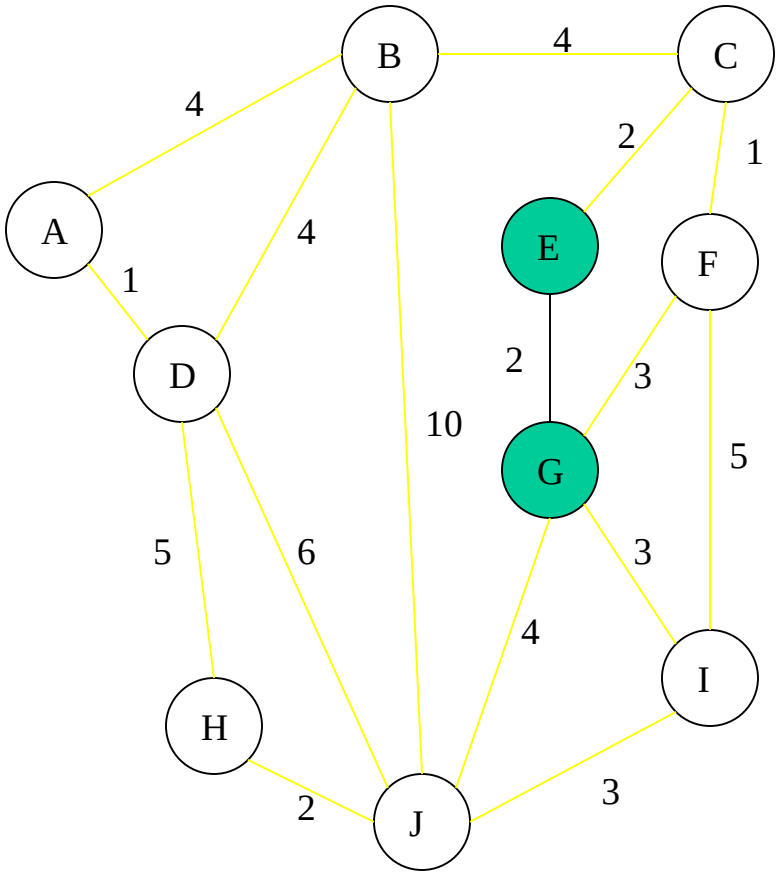
Round 1



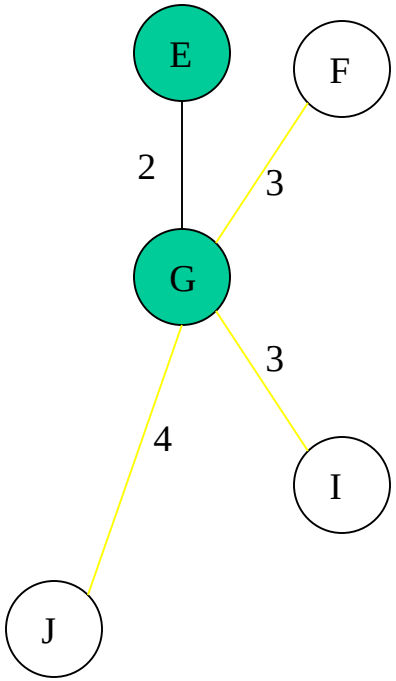
Tree G



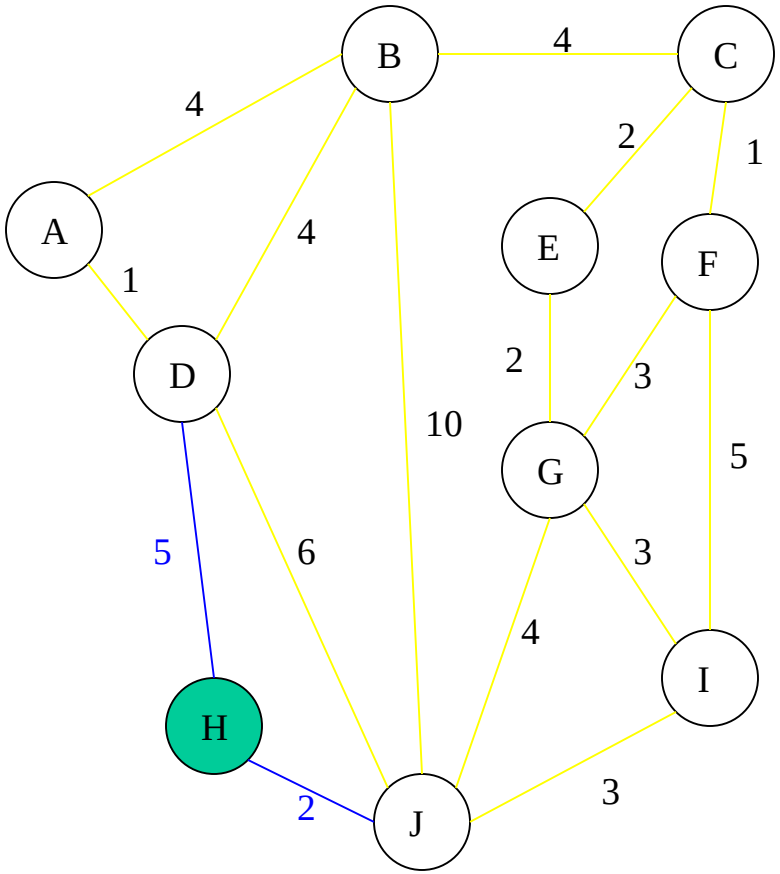
Round 1



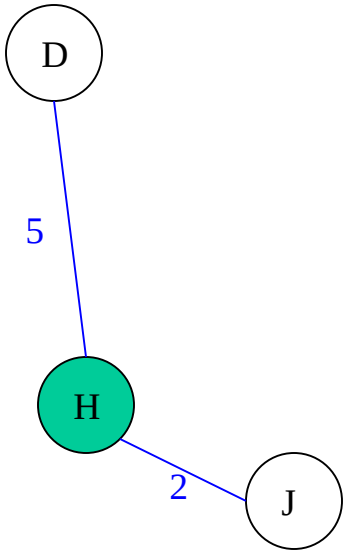
Edge G-E



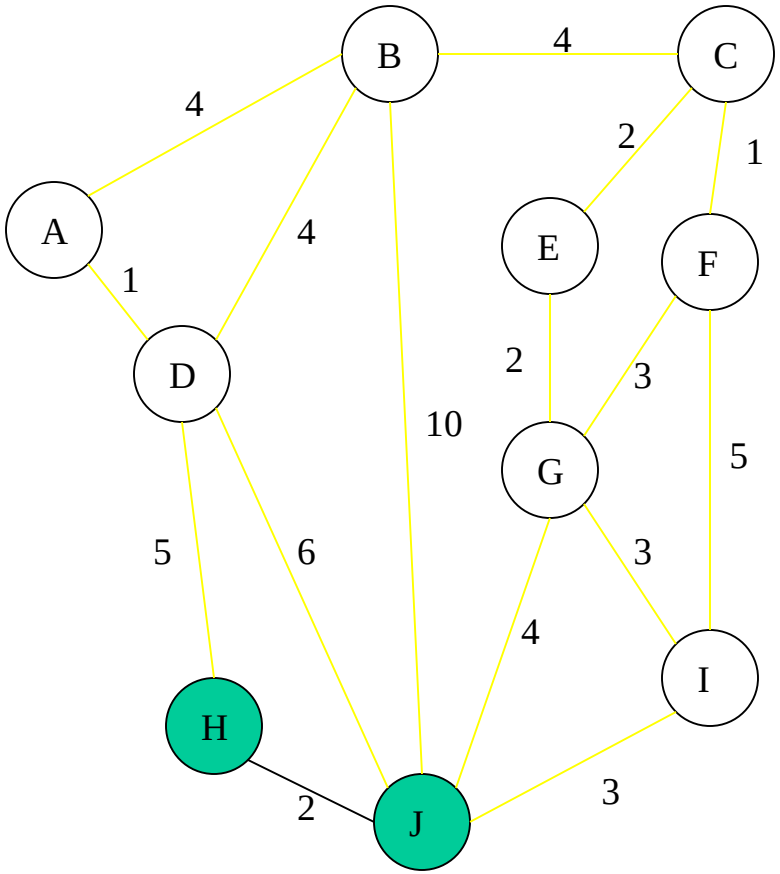
Round 1



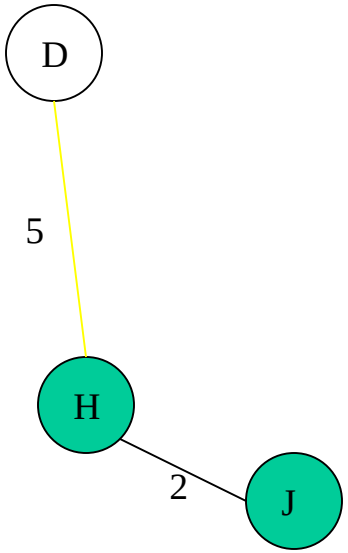
Tree H



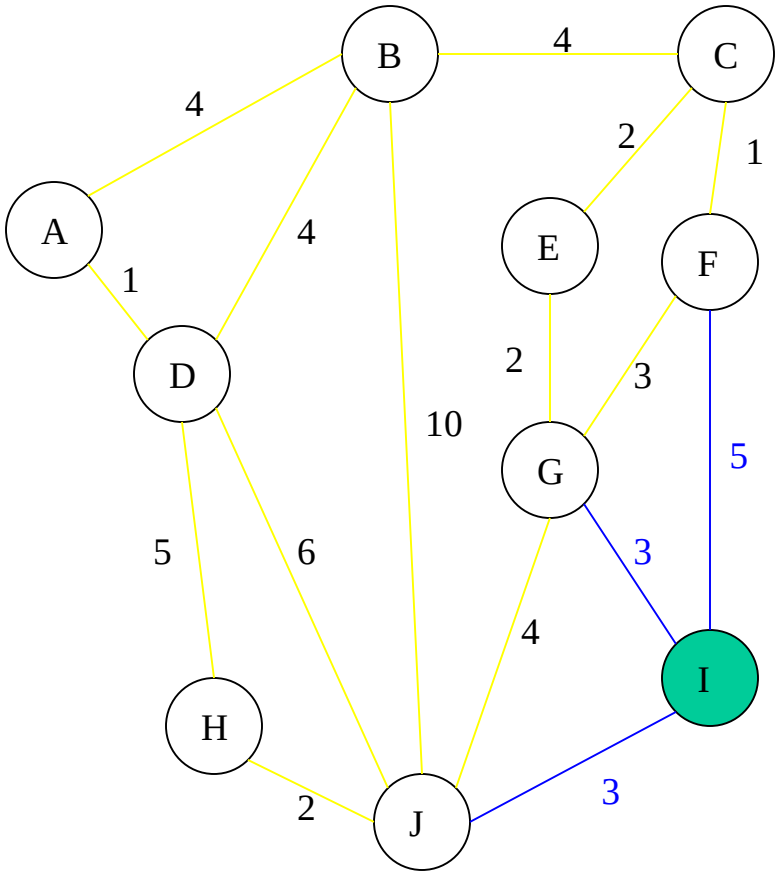
Round 1



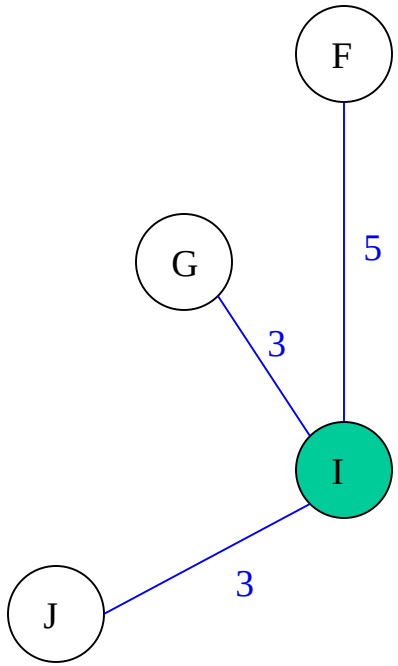
Edge H-J



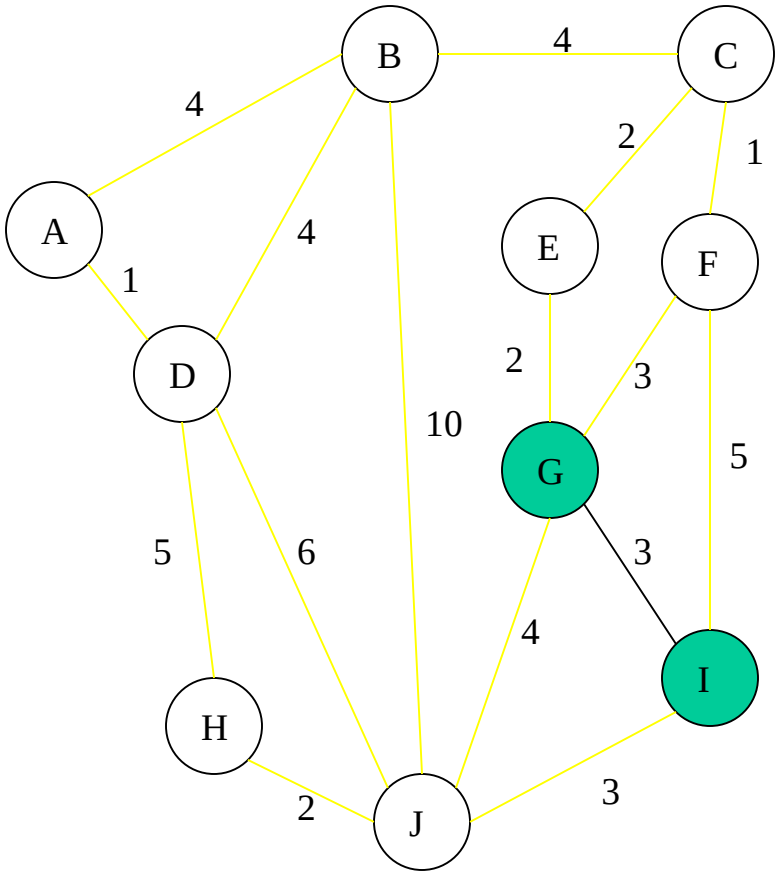
Round 1



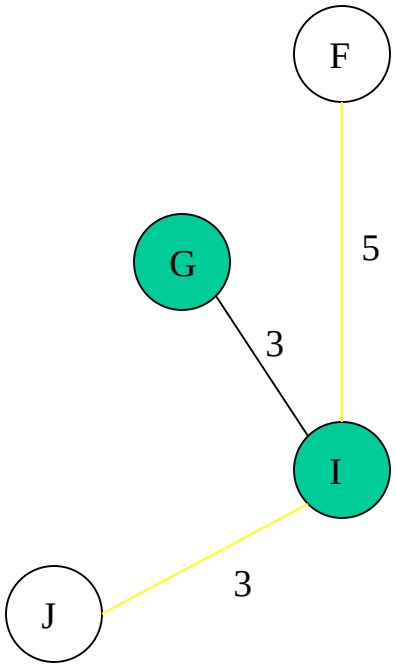
Tree I



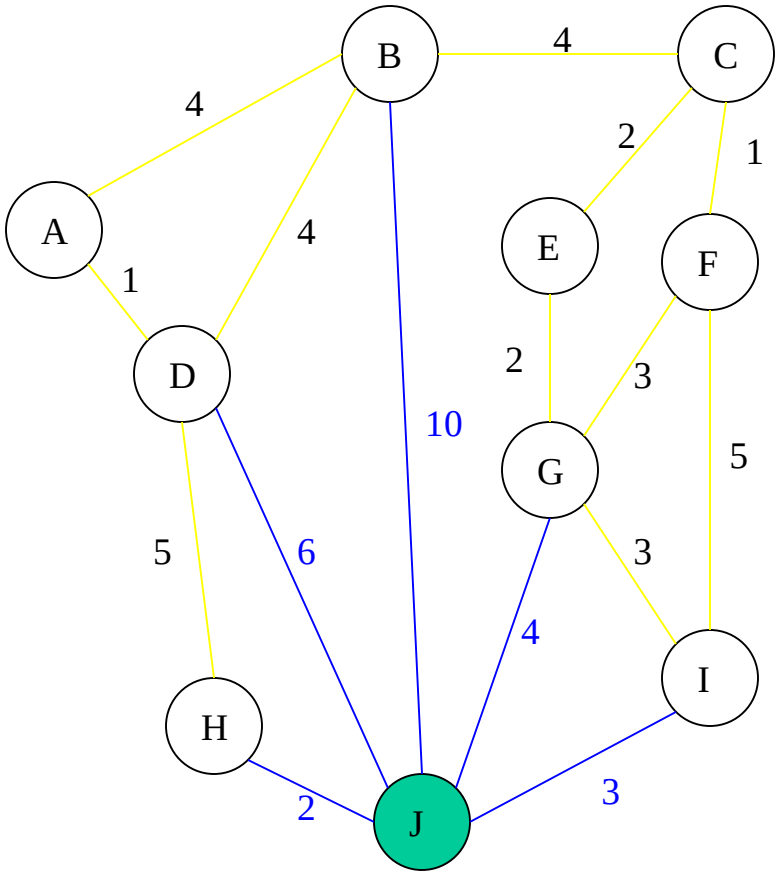
Round 1



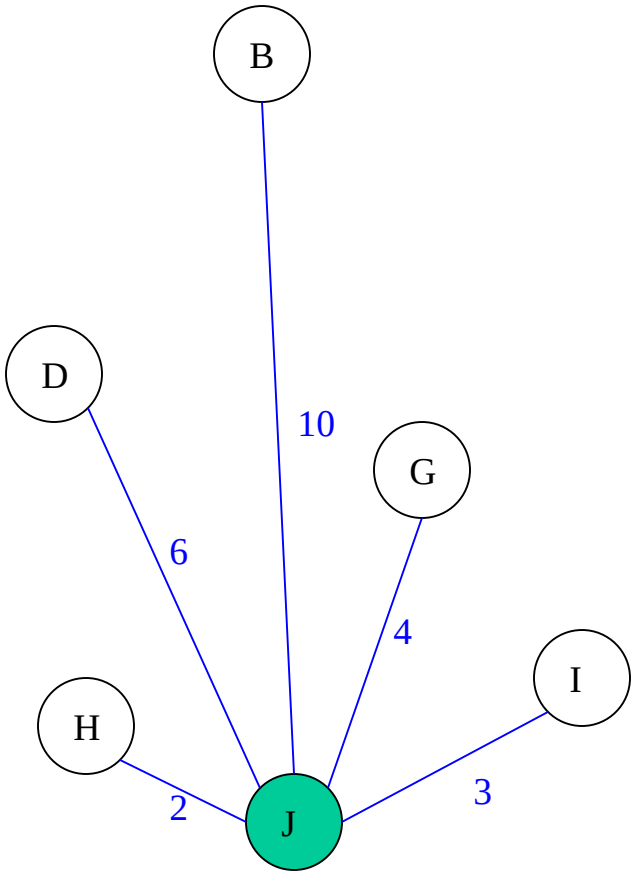
Edge I-G



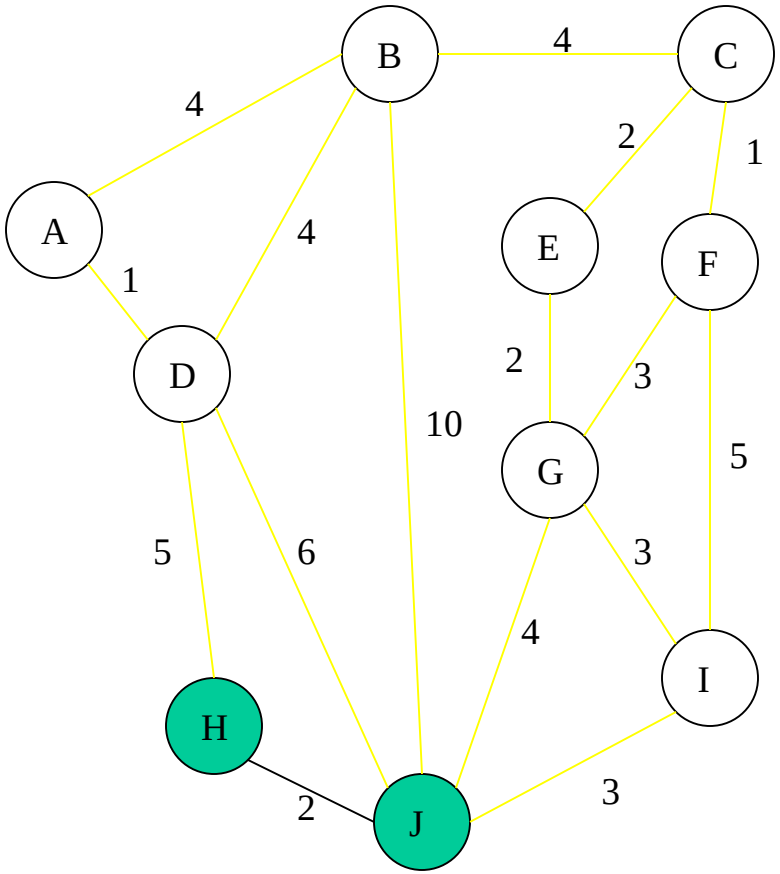
Round 1



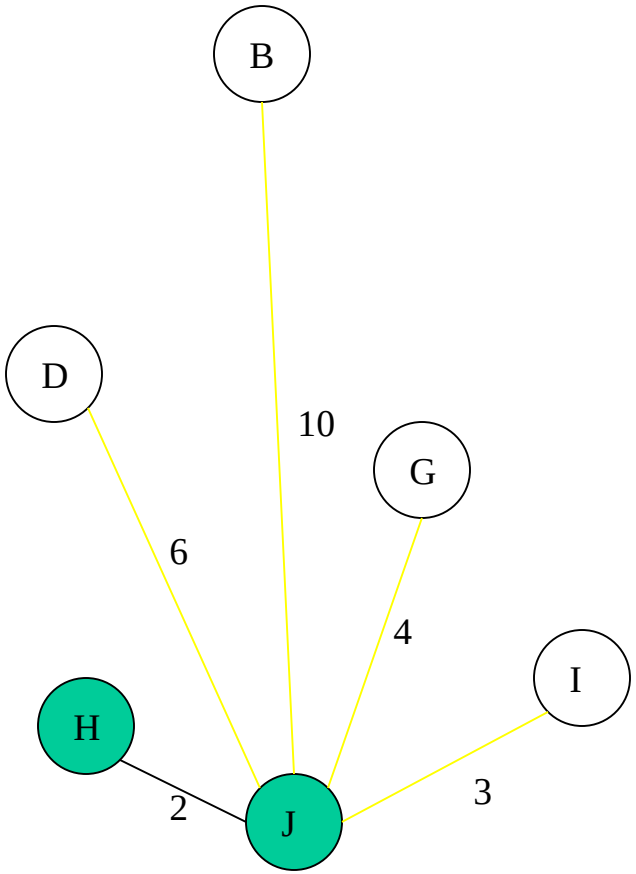
Tree J



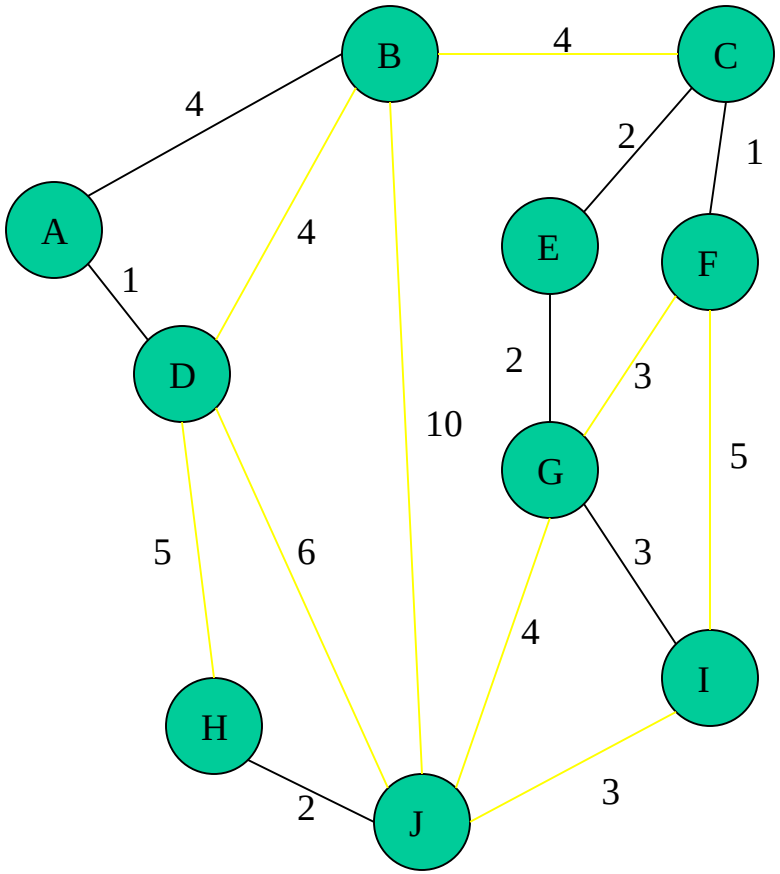
Round 1



Edge J-H



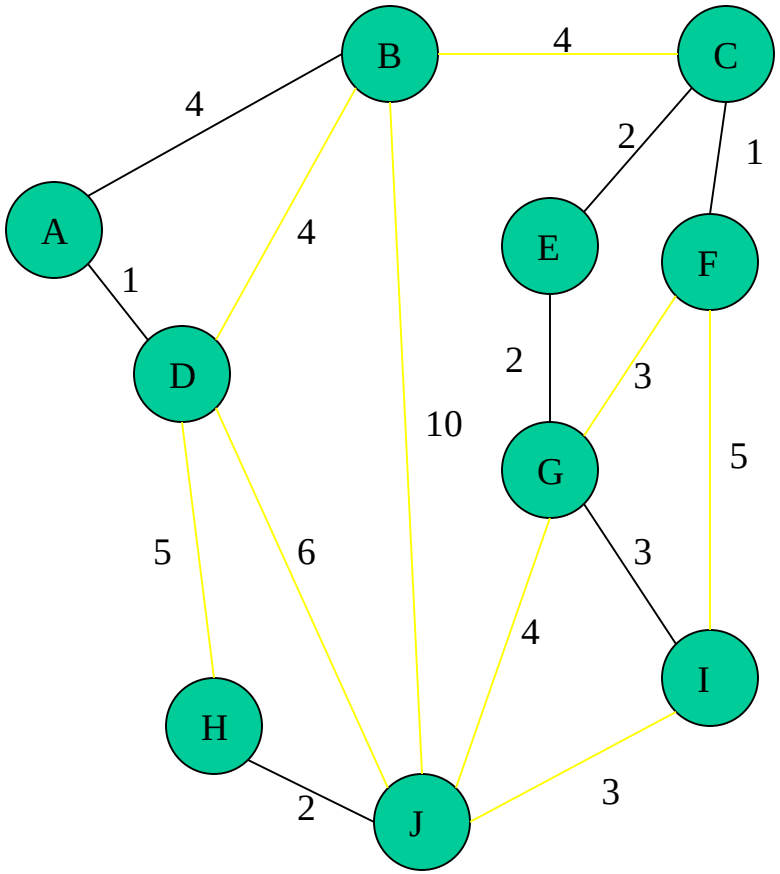
Round 1 Ends - Add Edges



List of Edges to Add

- A-D
- B-A
- C-F
- D-A
- E-C
- F-C
- G-E
- H-J
- I-G
- J-H

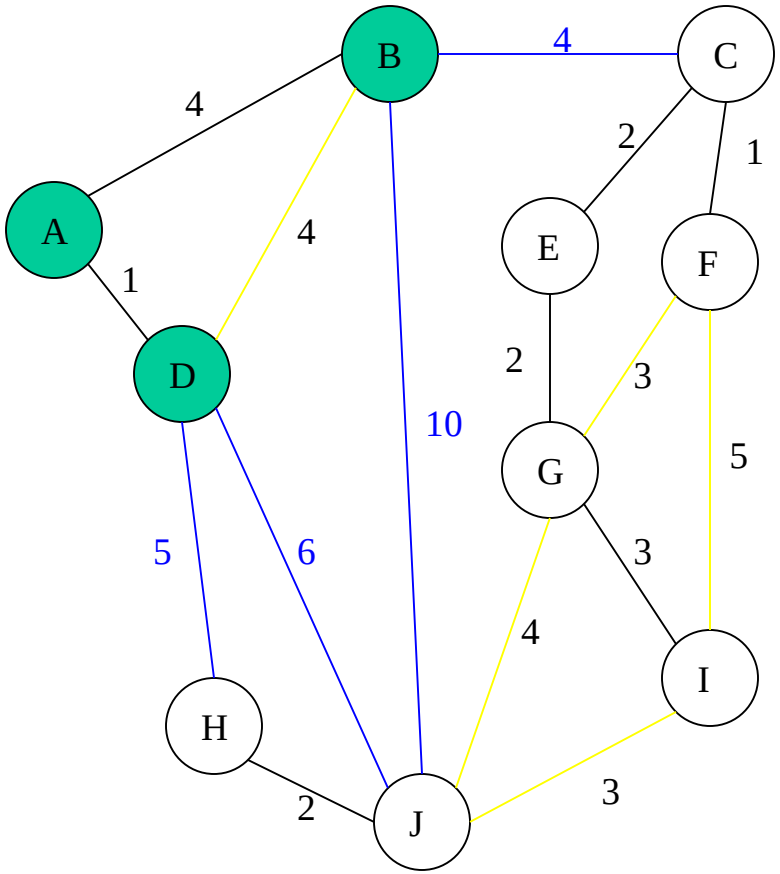
Trees of the Graph at Beginning of Round 2



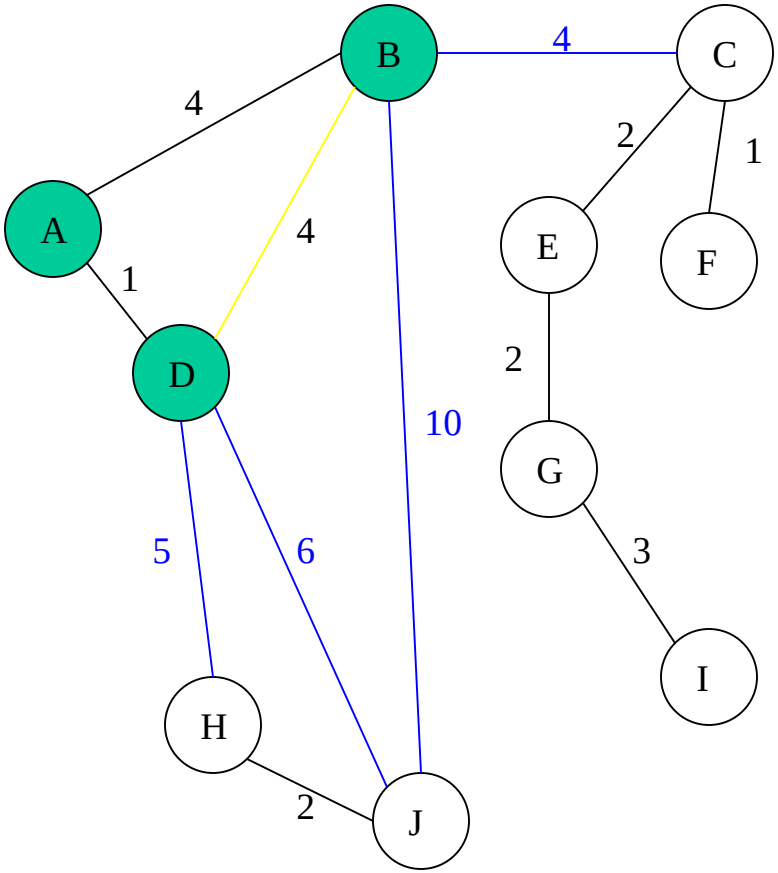
List of Trees

- D-A-B
- F-C-E-G-I
- H-J

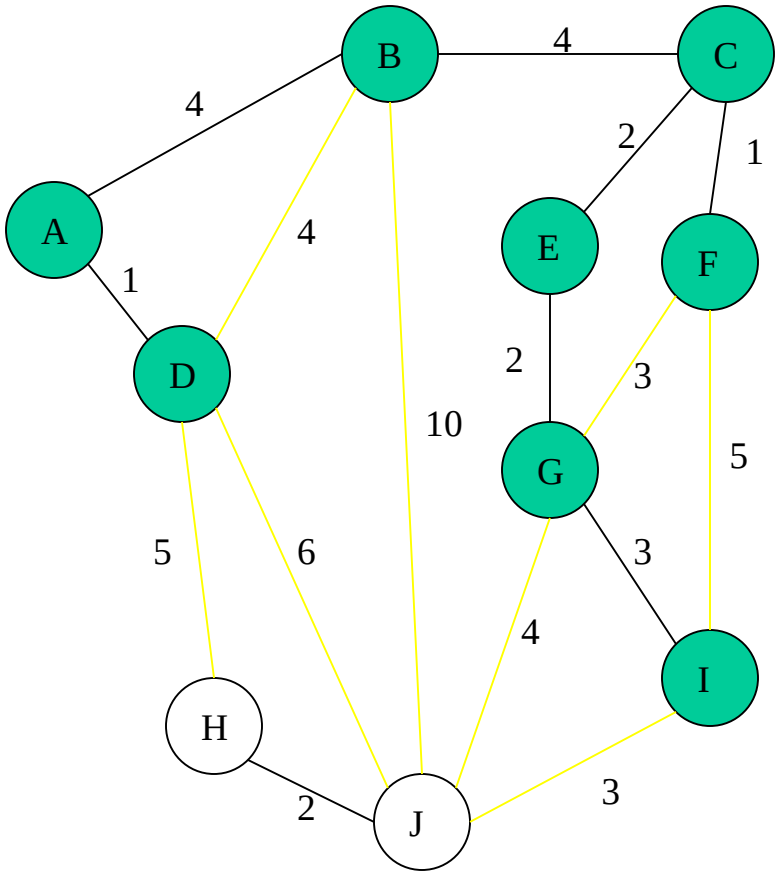
Round 2



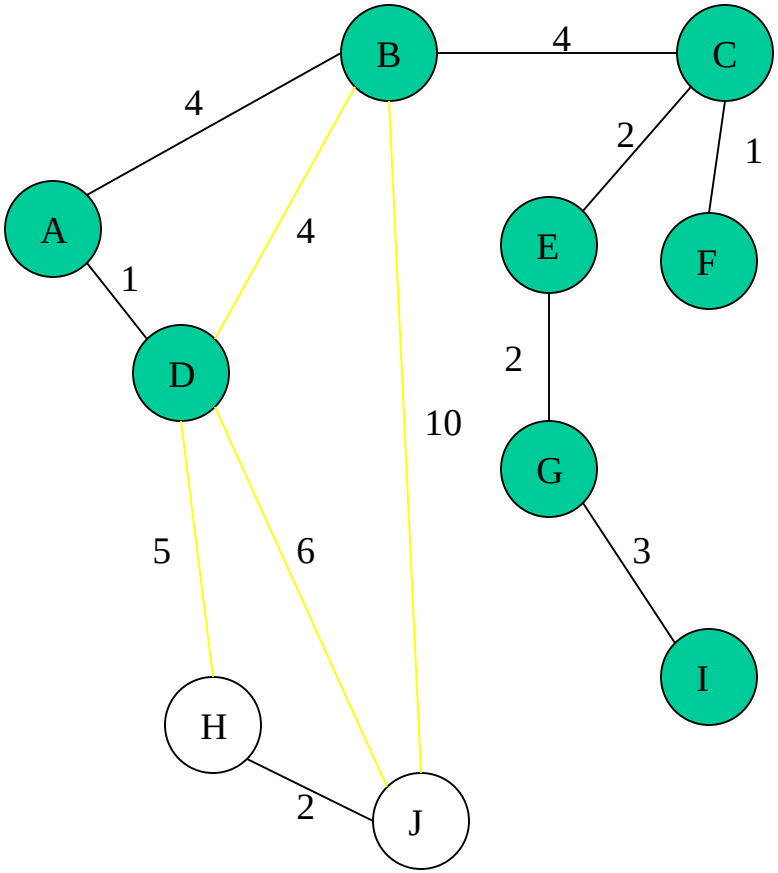
Tree D-A-B



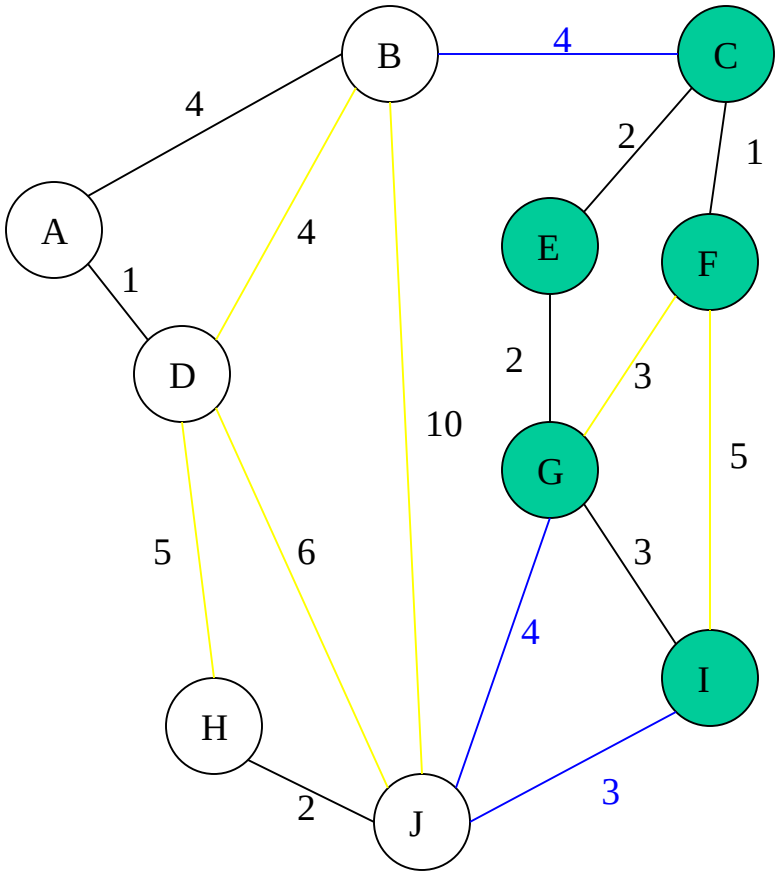
Round 2



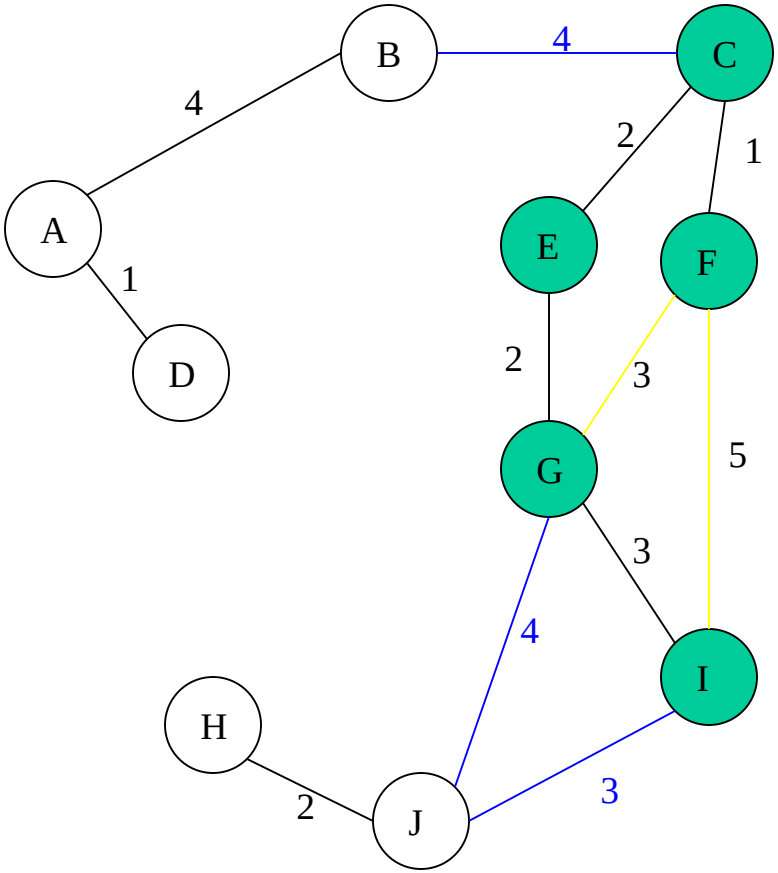
Edge B-C



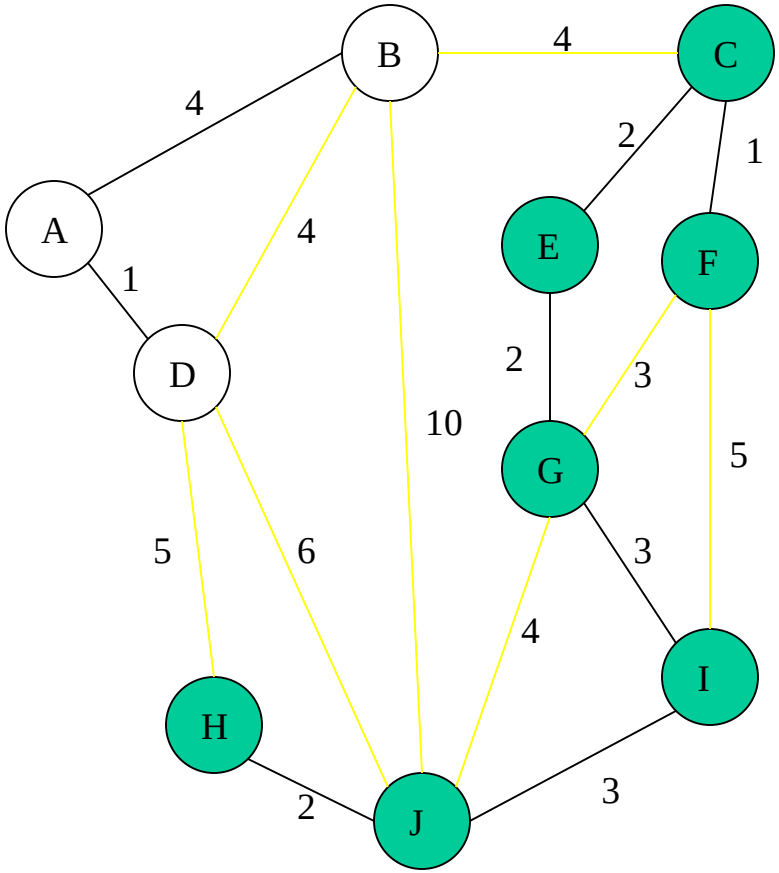
Round 2



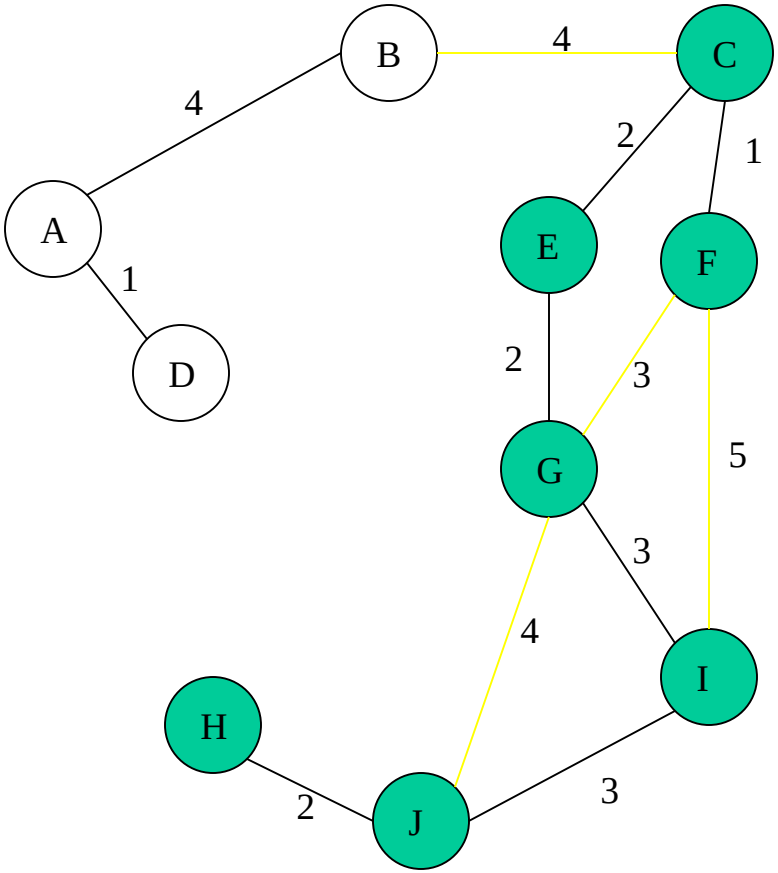
Tree F-C-E-G-I



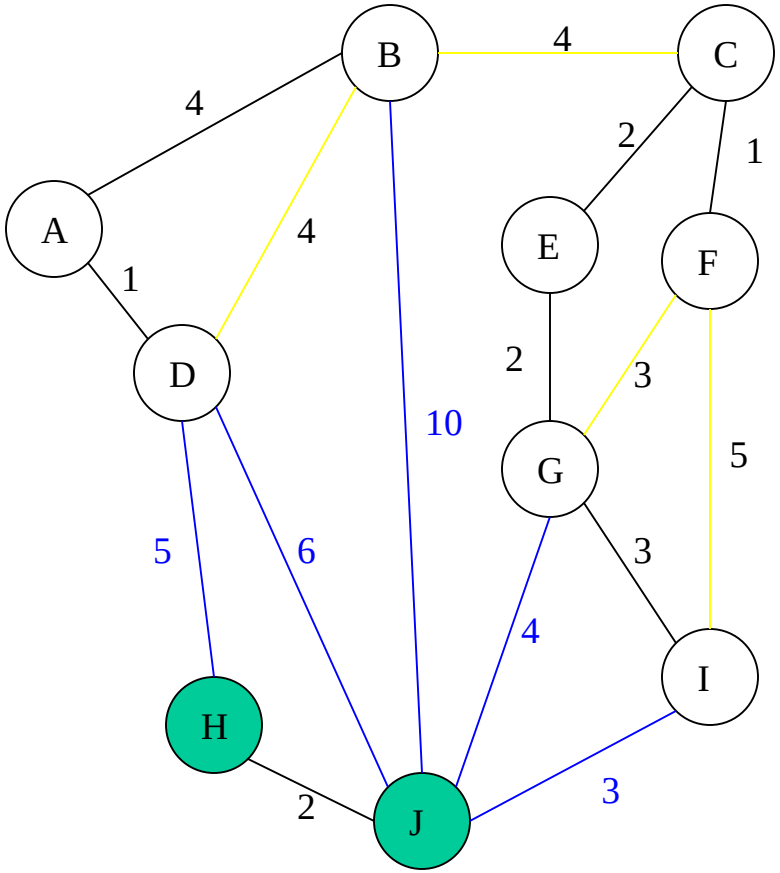
Round 2



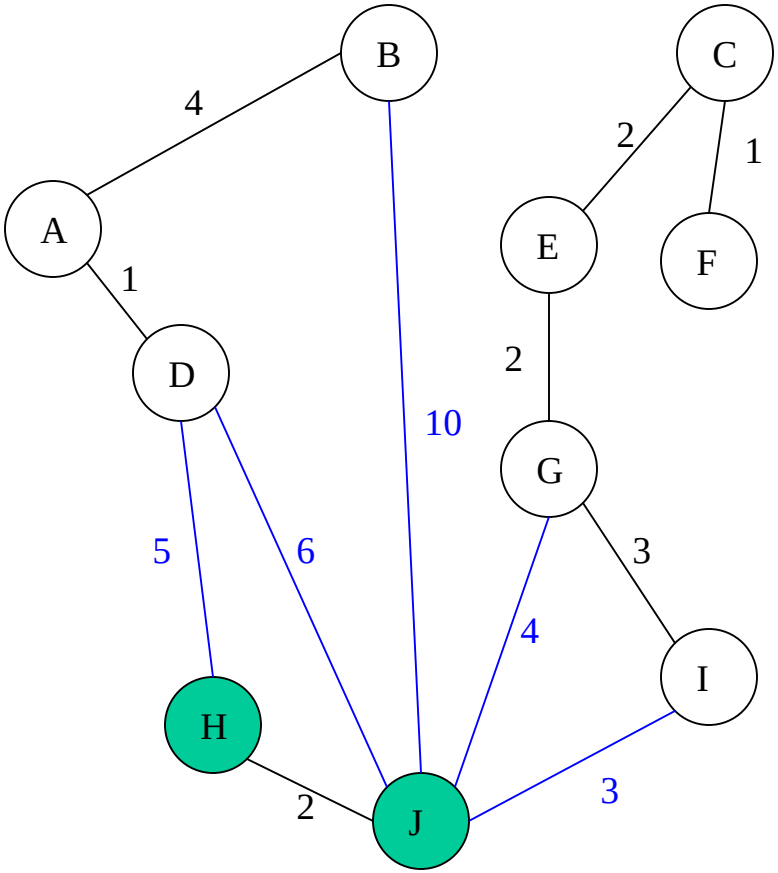
Edge I-J



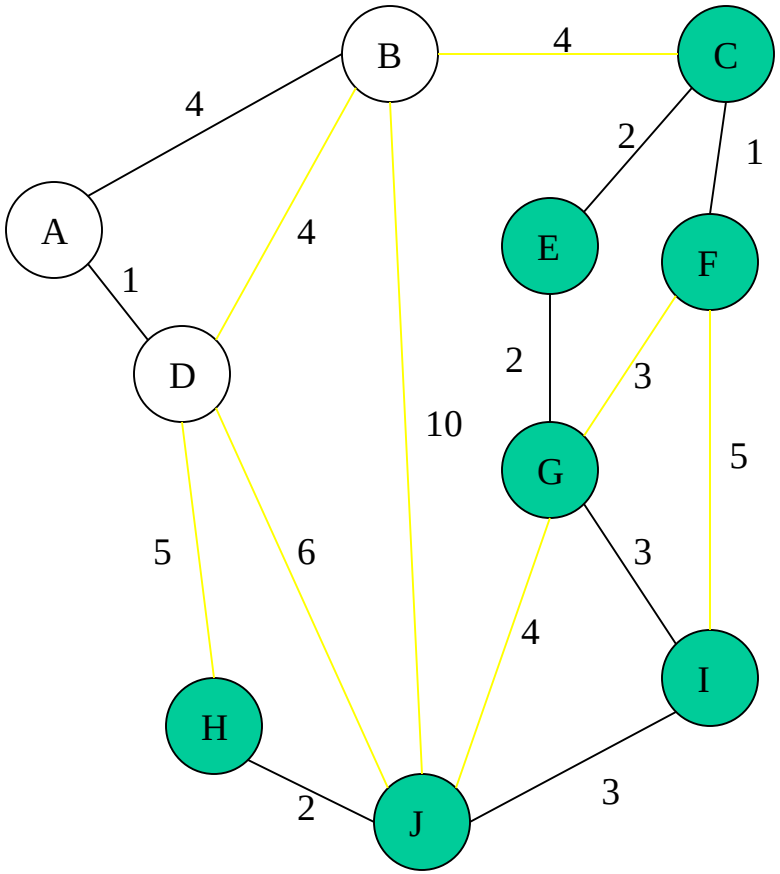
Round 2



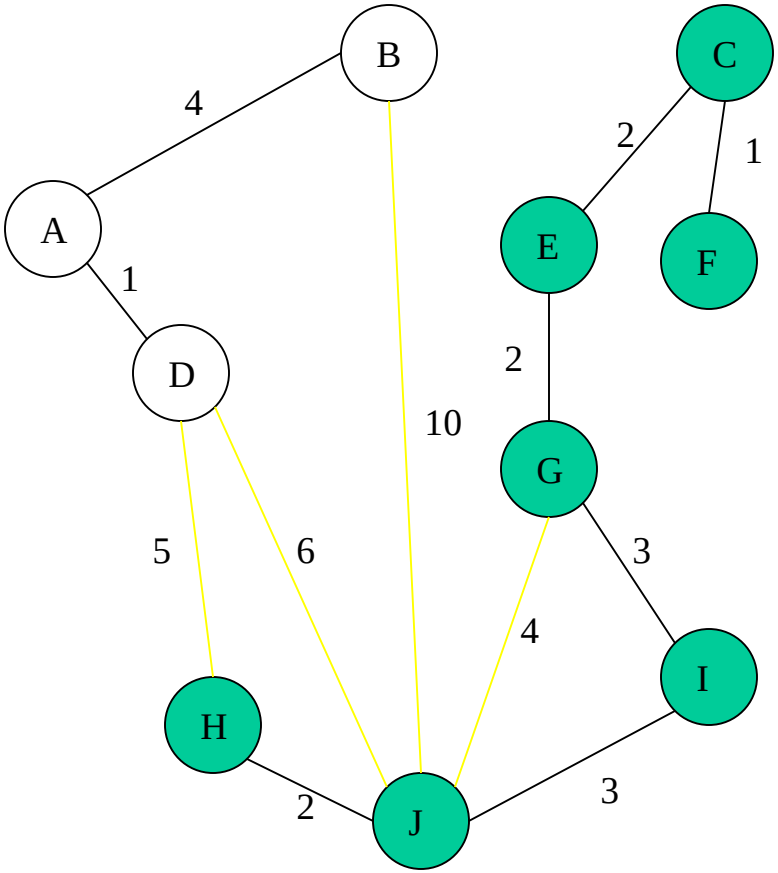
Tree H-J



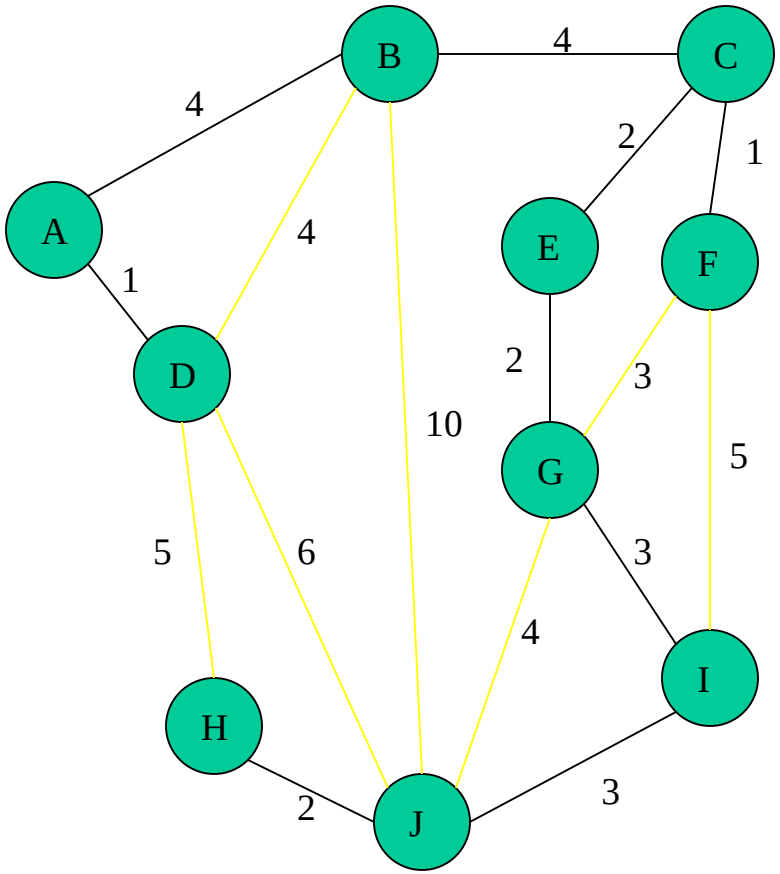
Round 2



Edge J-I



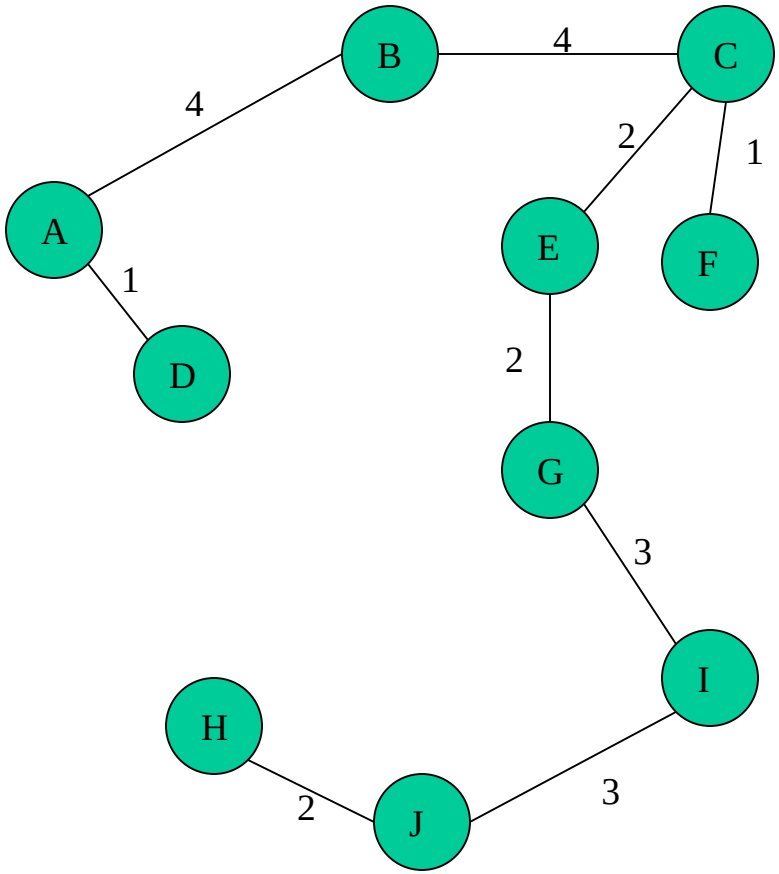
Round 2 Ends - Add Edges



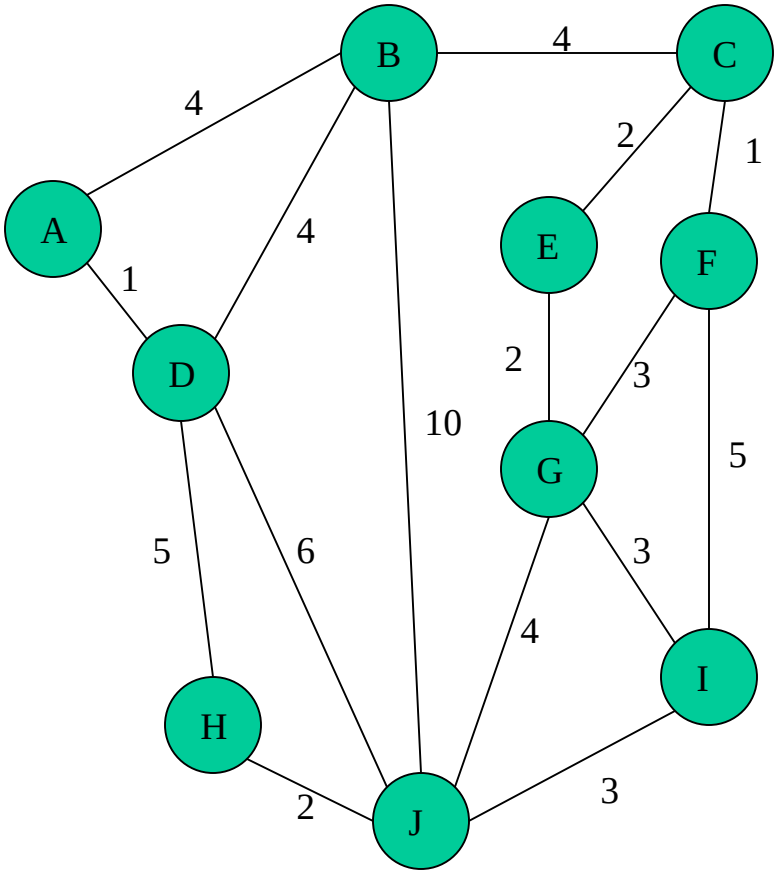
List of Edges to Add

- B-C
- I-J
- J-I

Minimum Spanning Tree



Complete Graph



Analysis of Boruvka's Algorithm

Running Time = O(m log n) (m = edges, n = nodes)

Although this algorithm is difficult to explain, unlike the two preceding algorithms, it does not require a complicated data structure.

Like Prim's, it does not need to worry about detecting cycles. It does, however, need to see the whole graph, but it only examines pieces of it at a time, not all of it at once.

Like Kruskal's it is best if edges are kept to a minimum (though it doesn't hurt to keep the nodes to a minimum as well).

Conclusion

Kruskal's and Boruvka's have better running times if the number of edges is low, while Prim's has a better running time if both the number of edges and the number of nodes are low.

Boruvka's avoids the complicated data structures needed for the other two algorithms.

So, of course, the best algorithm depends on the graph and if you want to bear the cost of complex data structures.

The best algorithm that I know of is a hybrid of Boruvka's and Prim's, which I did not examine here. It does O(log log n) passes of Boruvka's and then switches to Prim's, resulting in a running time of O(m log log n). So, it's the fastest algorithm, but would, of course, require the Fibonacci heap for Prim's which Boruvka's avoids when used by itself. However, in order to keep things simple, I did not explore it here.