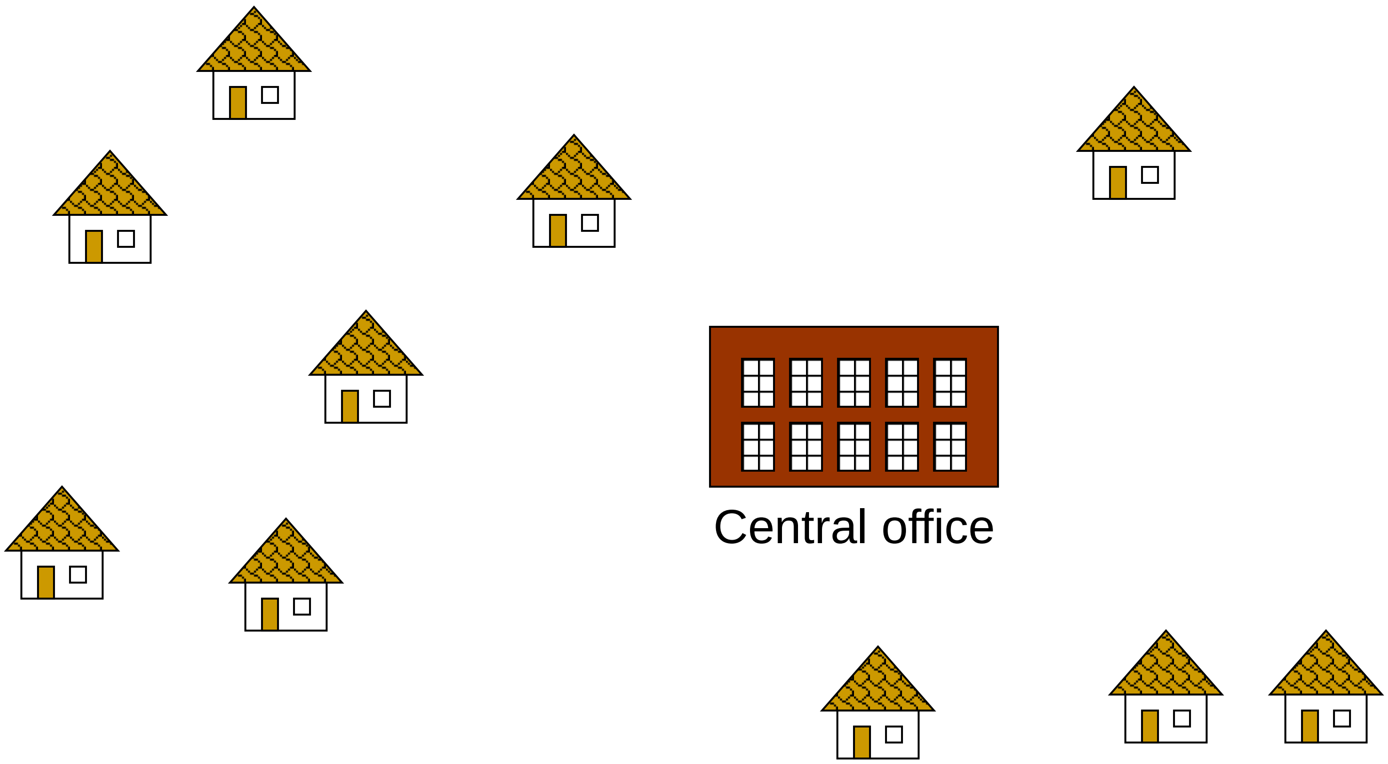
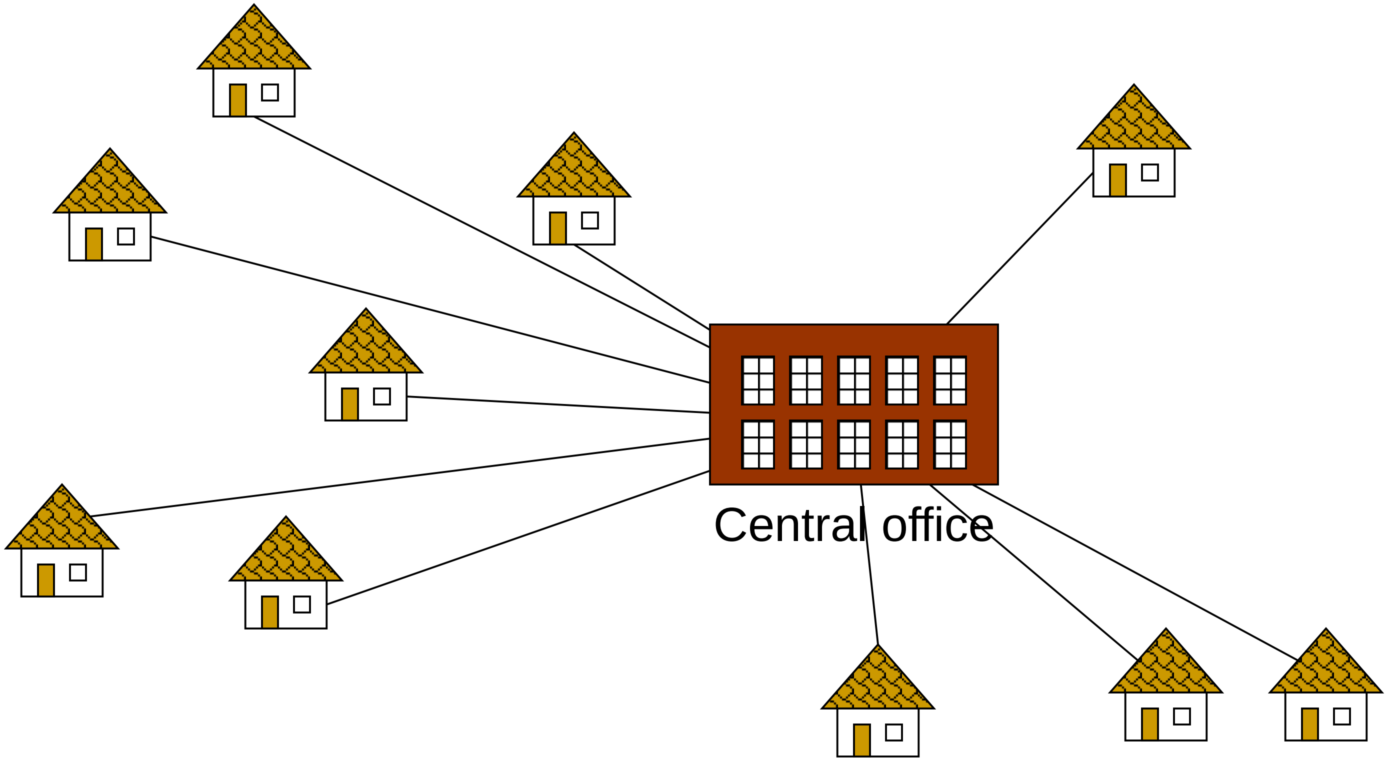
Minimum Spanning Tree in Graph

Slides by Si Dong, M.T. Goodrich and R. Tamassia

Problem: Laying Telephone Wire

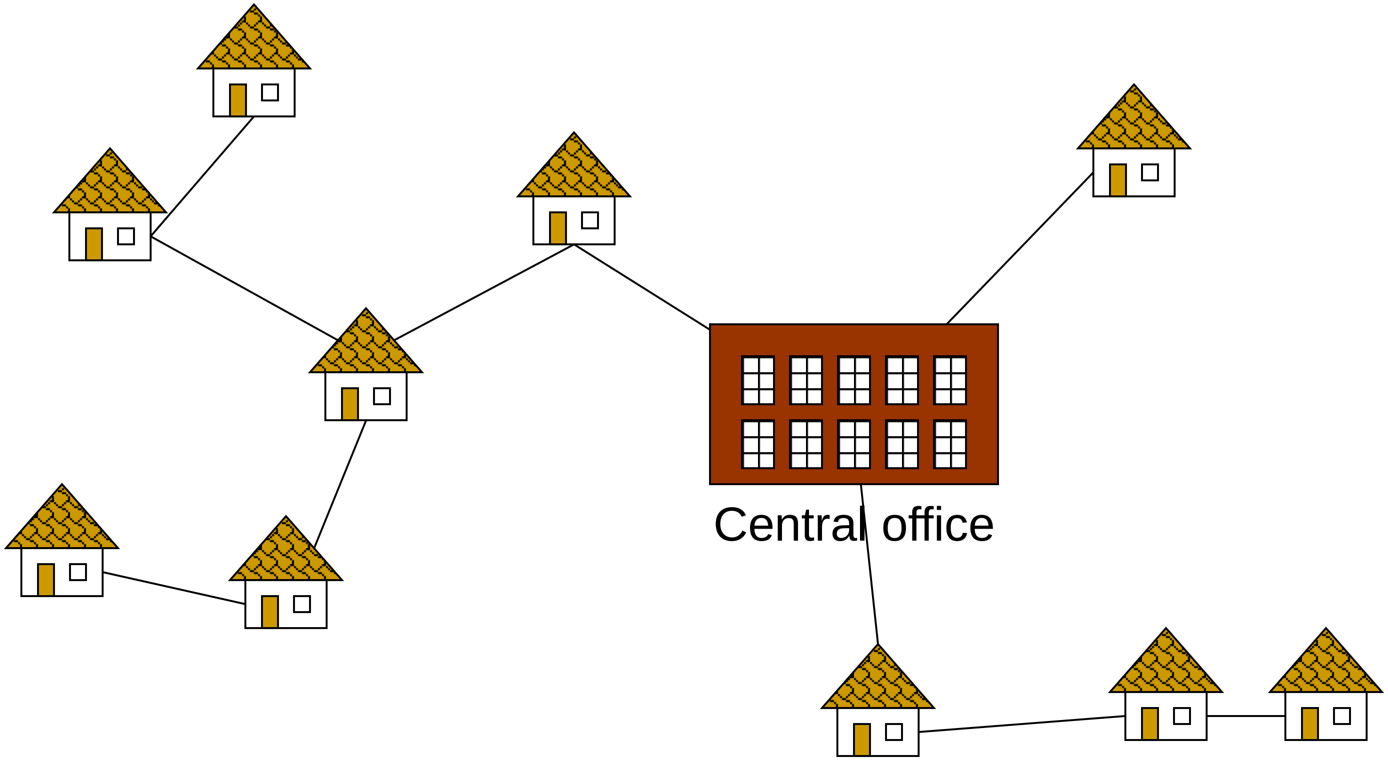


Wiring: Naive Approach



Expensive!

Wiring: Better Approach

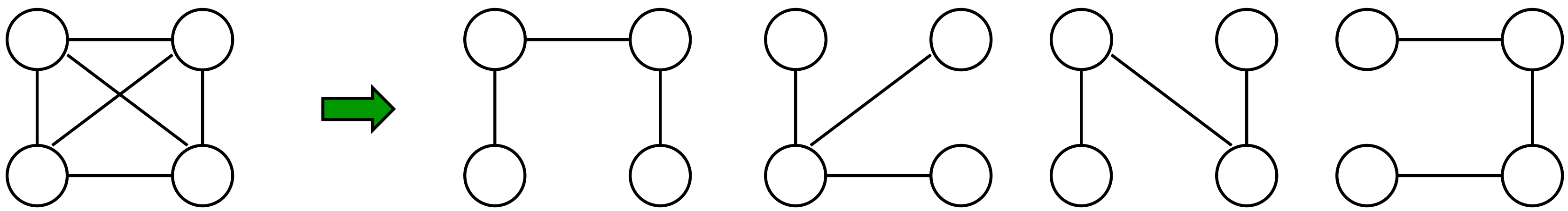


Minimize the total length of wire connecting **ALL** customers

Spanning trees

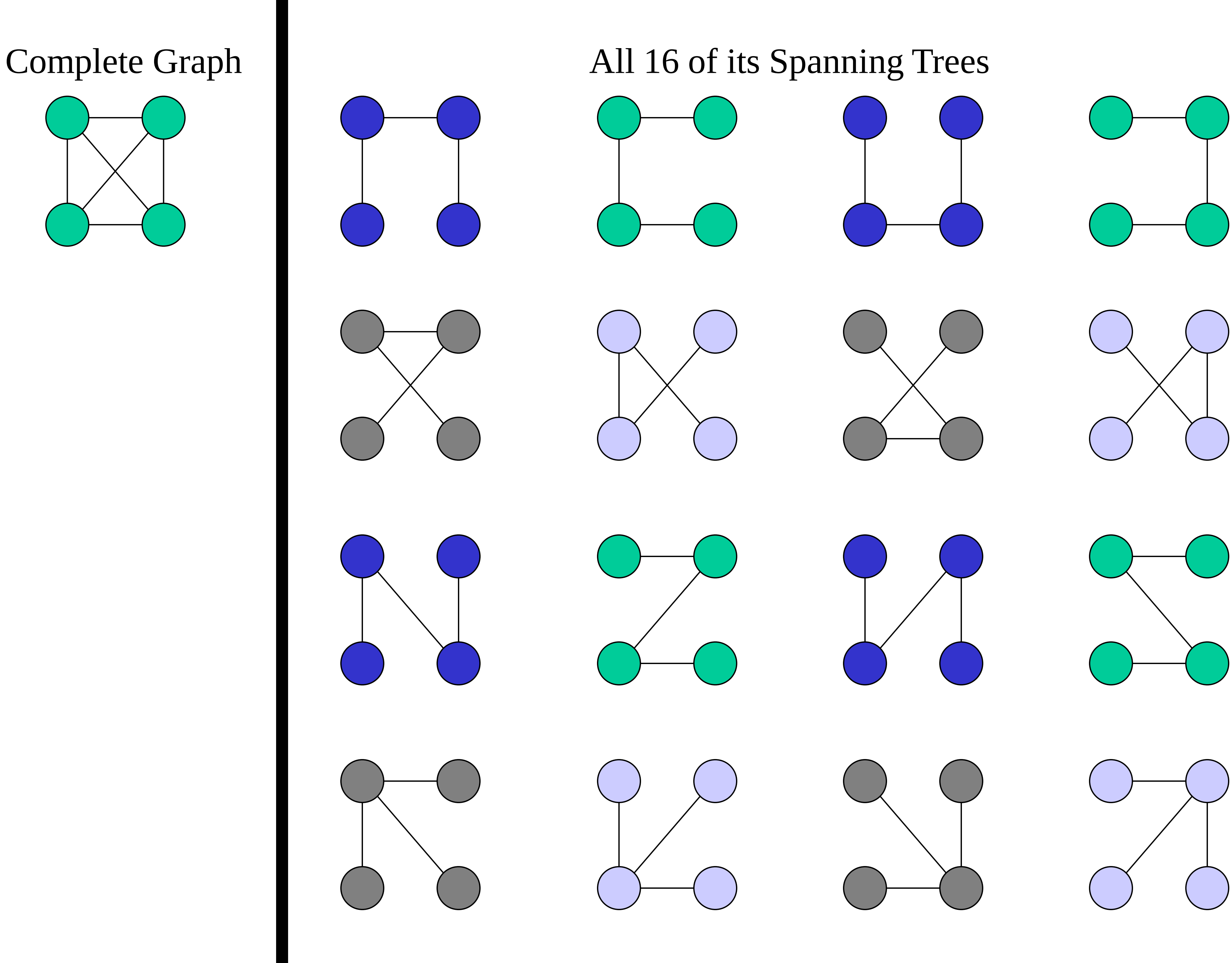
- Suppose you have a **connected undirected** graph:
  - Connected: every node is reachable from every other node
  - Undirected: edges do not have an associated direction

- ...then a spanning tree of the graph is a connected subgraph which contains all the vertices and has no cycles.



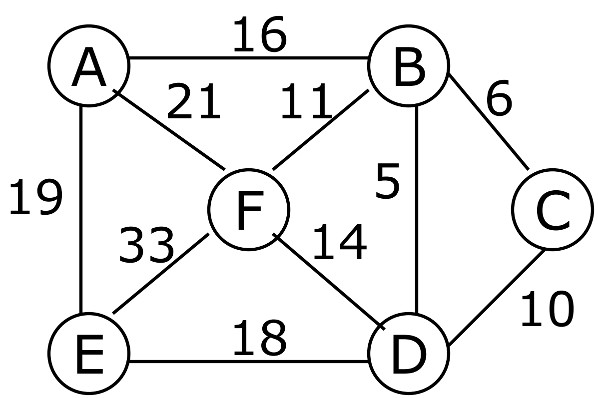
A connected, undirected graph

Four of the spanning trees of the graph

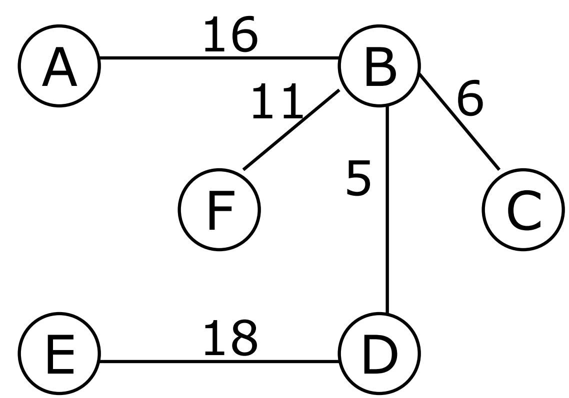

Complete Graph
All 16 of its Spanning Trees

Minimum-cost spanning trees

- Suppose you have a connected undirected graph with a weight (or cost) associated with each edge.
- The cost of a spanning tree would be the sum of the costs of its edges.
- A minimum-cost spanning tree is a spanning tree that has the lowest cost.



A connected, undirected graph



A minimum-cost spanning tree

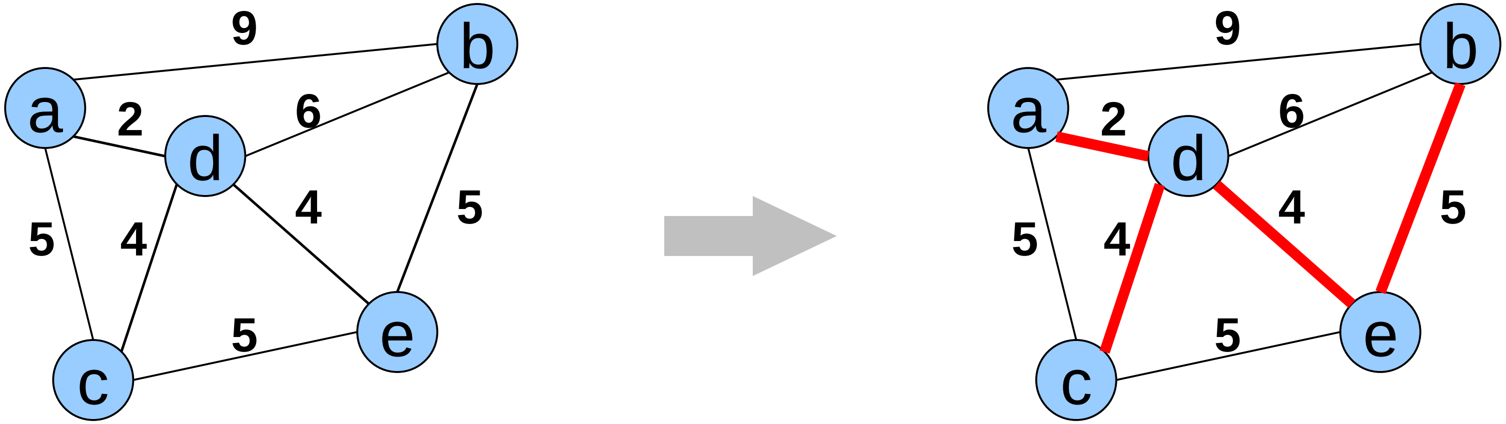
Minimum Spanning Tree (MST)

A **minimum spanning tree** is a subgraph of an undirected weighted graph ***G***, such that

Tree = connected graph without cycles

- it is a tree (i.e., it is acyclic)
- it covers all the vertices ***V***
  - contains $|V| - 1$ edges
- the total cost associated with tree edges is the minimum among all possible spanning trees
- not necessarily unique.

How Can We Generate a MST?



Finding minimum spanning trees

- There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms.

- **Kruskal's algorithm**: Start with *no* nodes or edges in the spanning tree, and repeatedly add the cheapest **edge** that does not create a cycle
  - Here, we consider the spanning tree to consist of edges only

- **Prim's algorithm**: Start with any *one node* in the spanning tree, and repeatedly add the cheapest edge, and the **node** it leads to, for which the node is not already in the spanning tree.
  - Here, we consider the spanning tree to consist of both nodes and edges

Kruskal’s algorithm

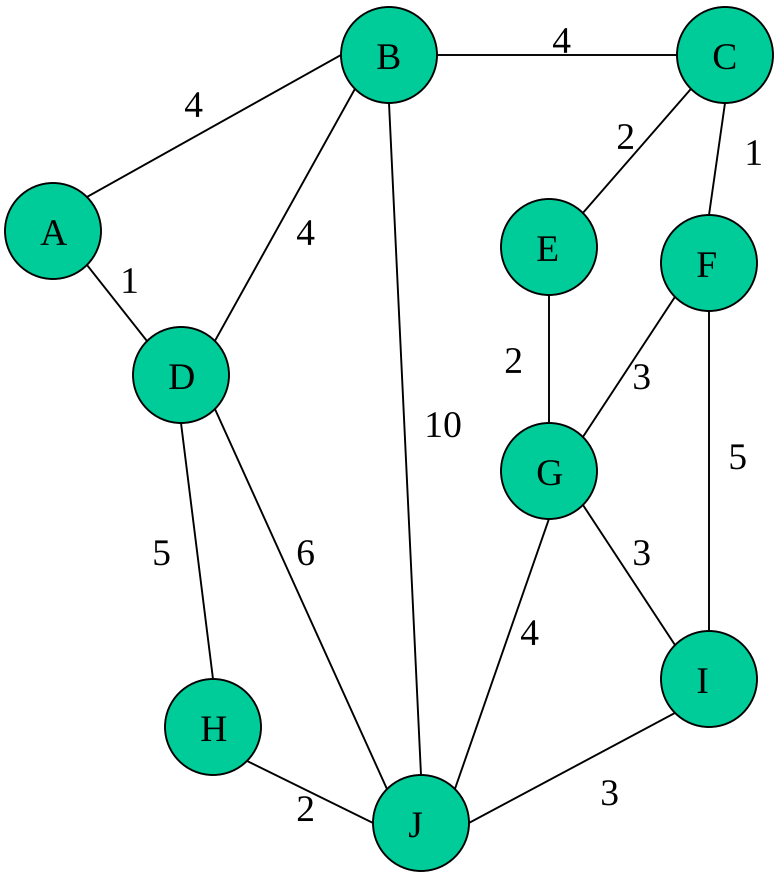
The steps are:

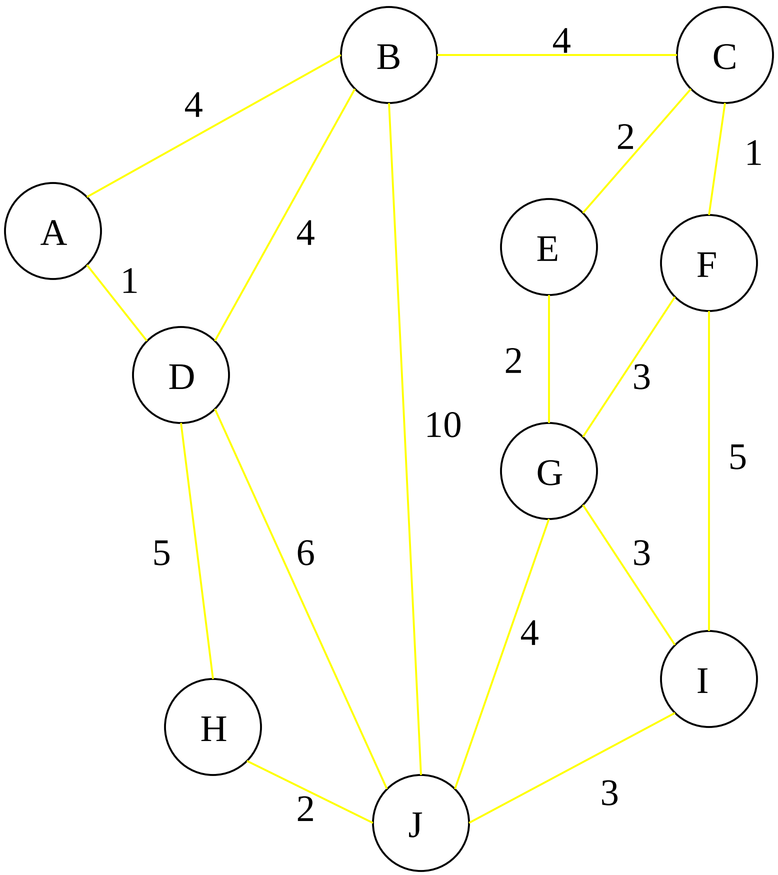
1. The forest is constructed - with each node in a separate tree.
2. The edges are placed in a priority queue.
3. Until we've added n-1 edges,
    (1). **Extract the cheapest edge from the priority queue**,
    (2). **If it forms a cycle, reject it**. **Else add it to the forest**.

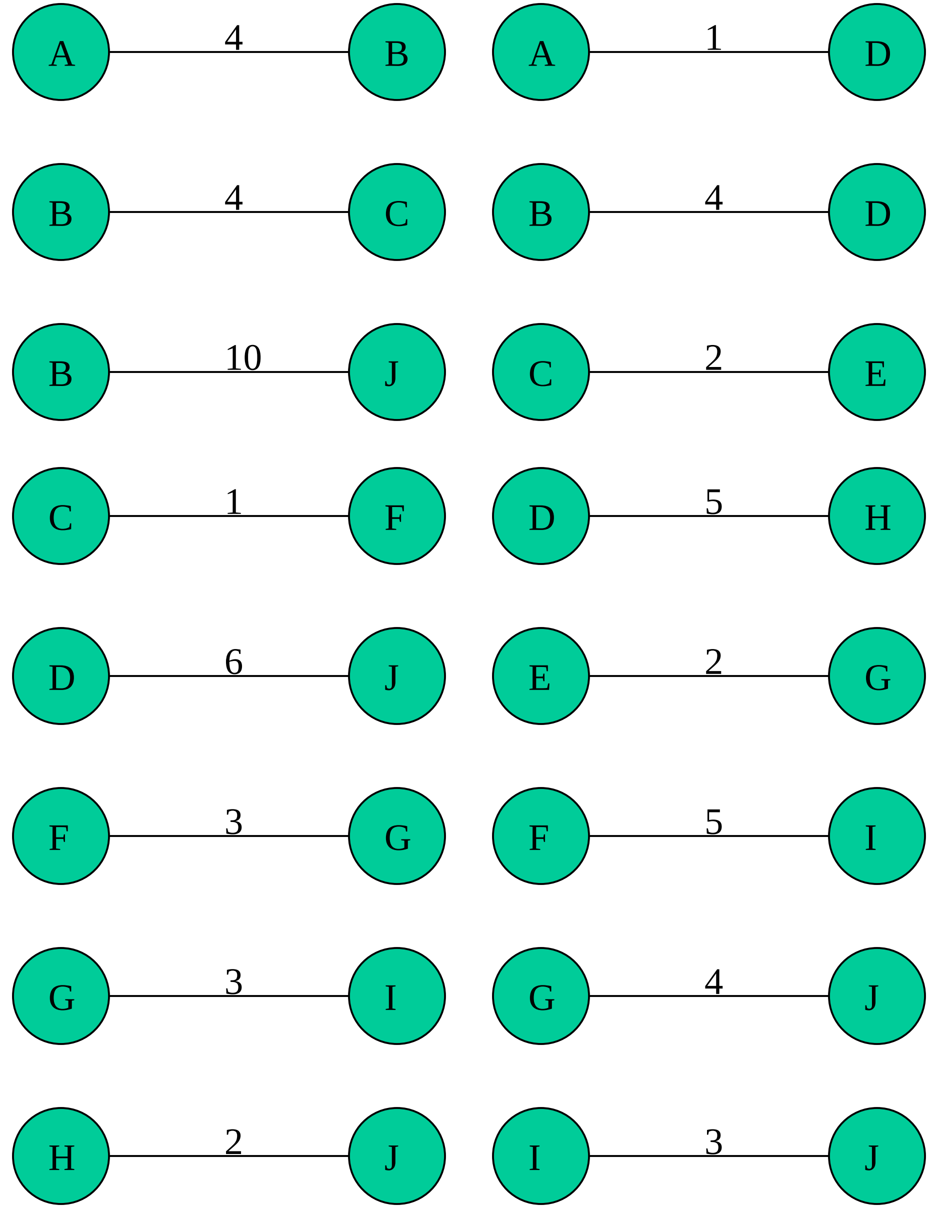
Adding it to the forest will join two trees together.

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

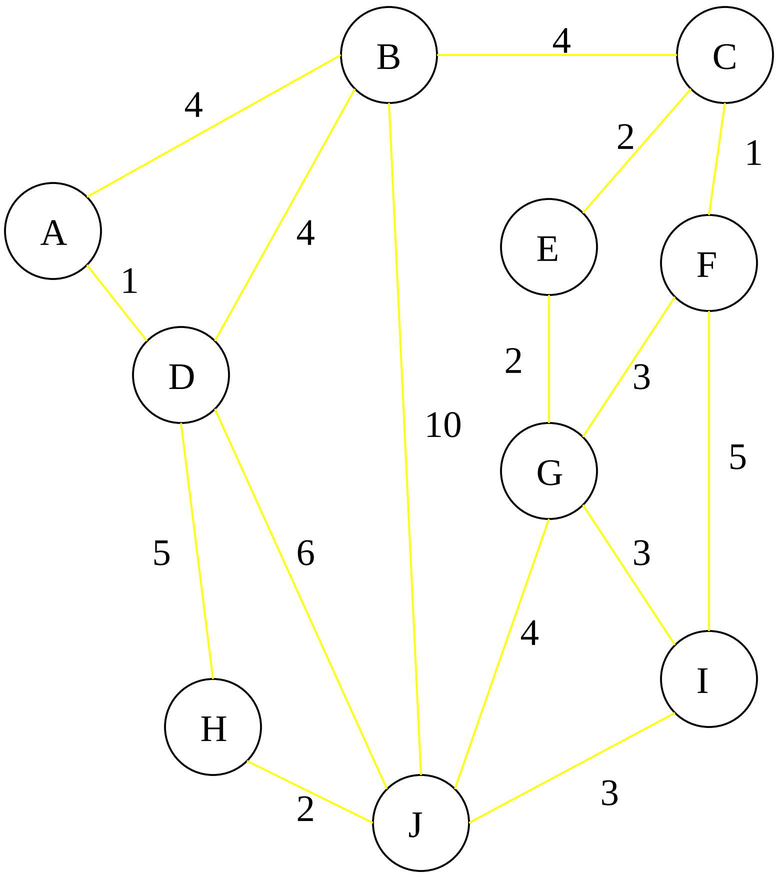


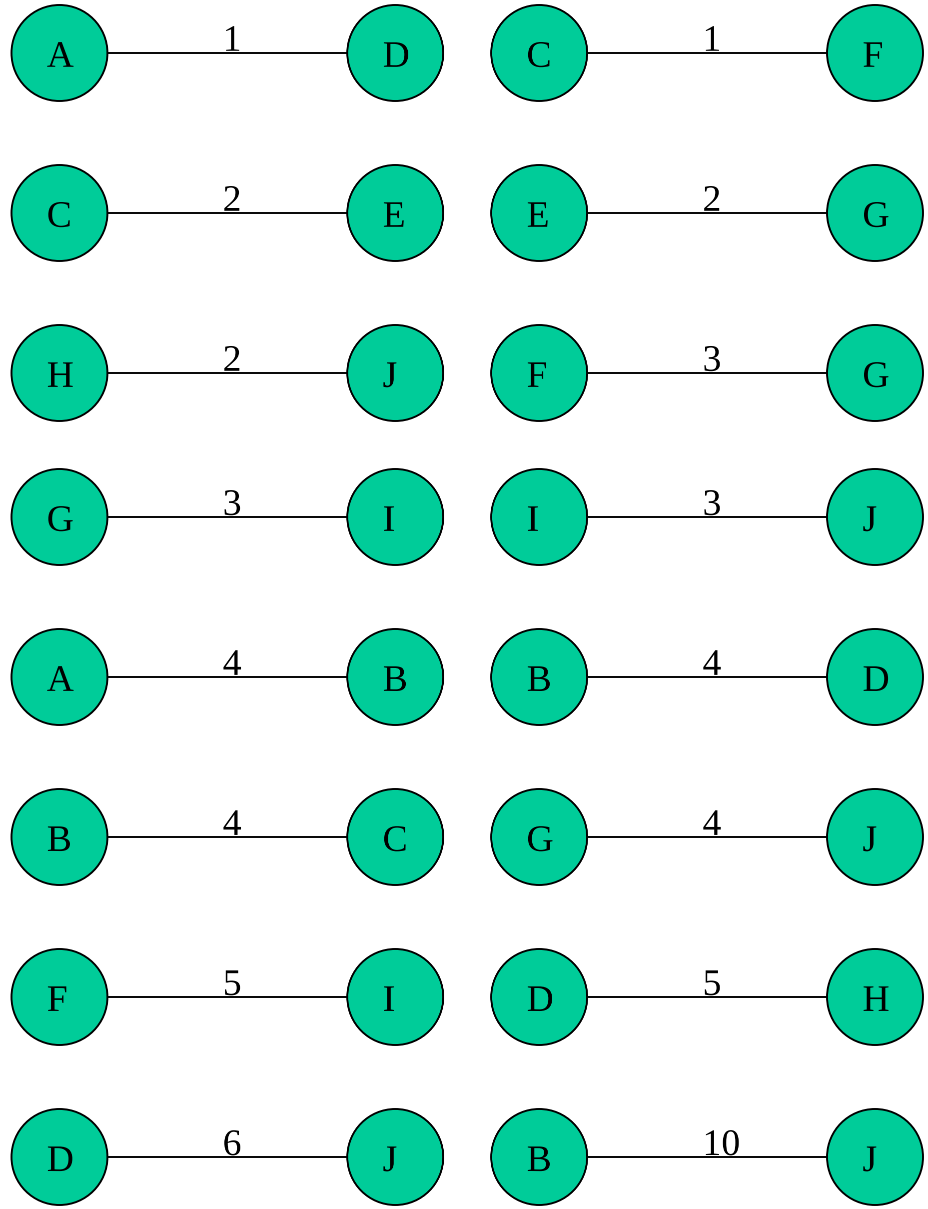

A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
10
2
3
5
5
6
4
3
2
3


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

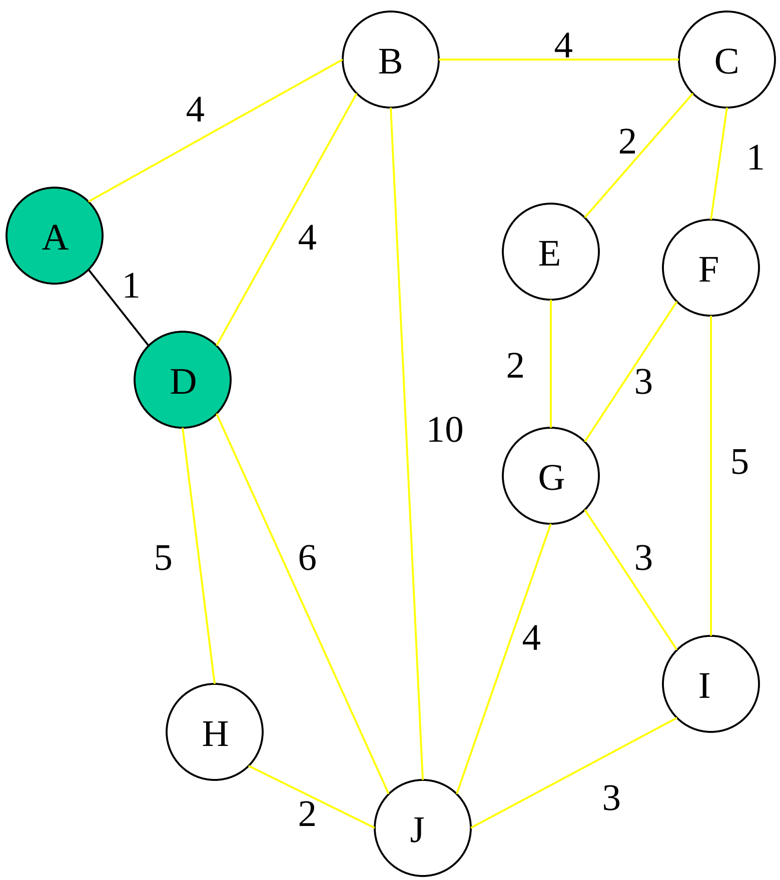
Sort Edges

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

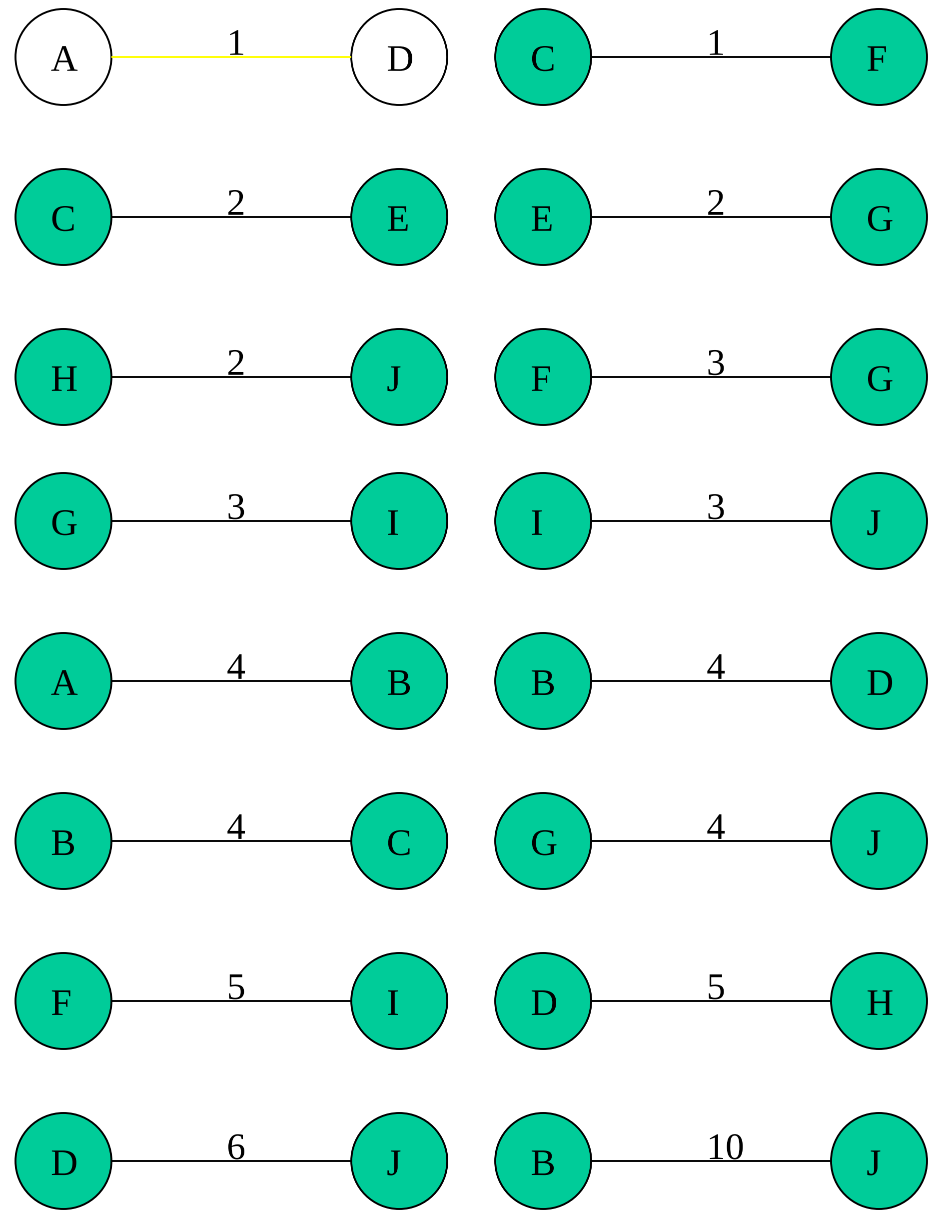




Add Edge

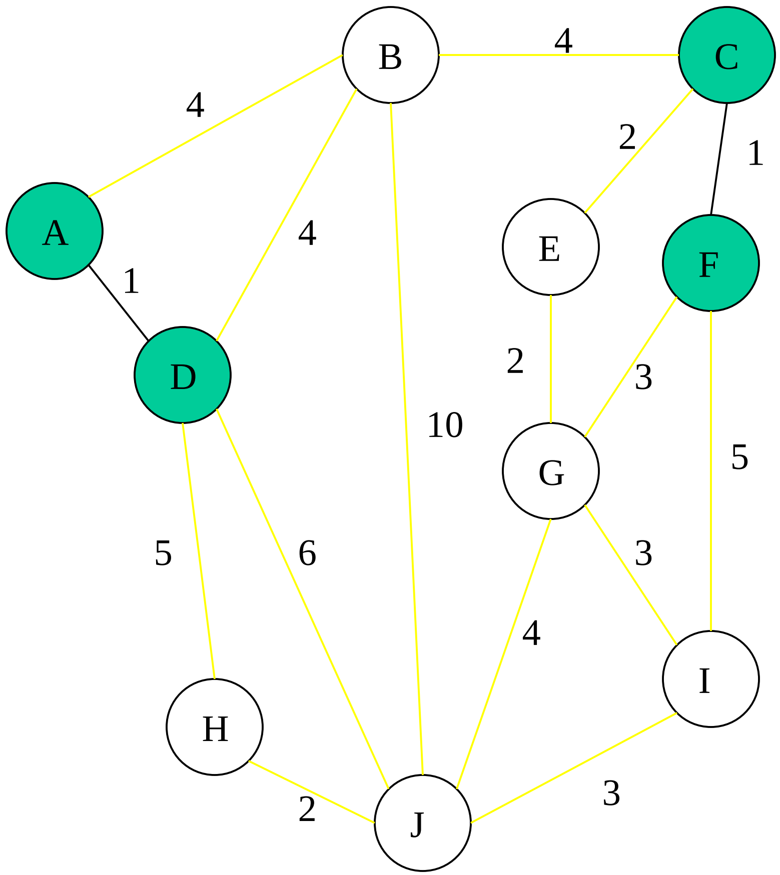


A
B
C
D
E
F
G
H
I
J
4
4
2
1
4
1
10
2
3
5
5
6
3
4
2
3

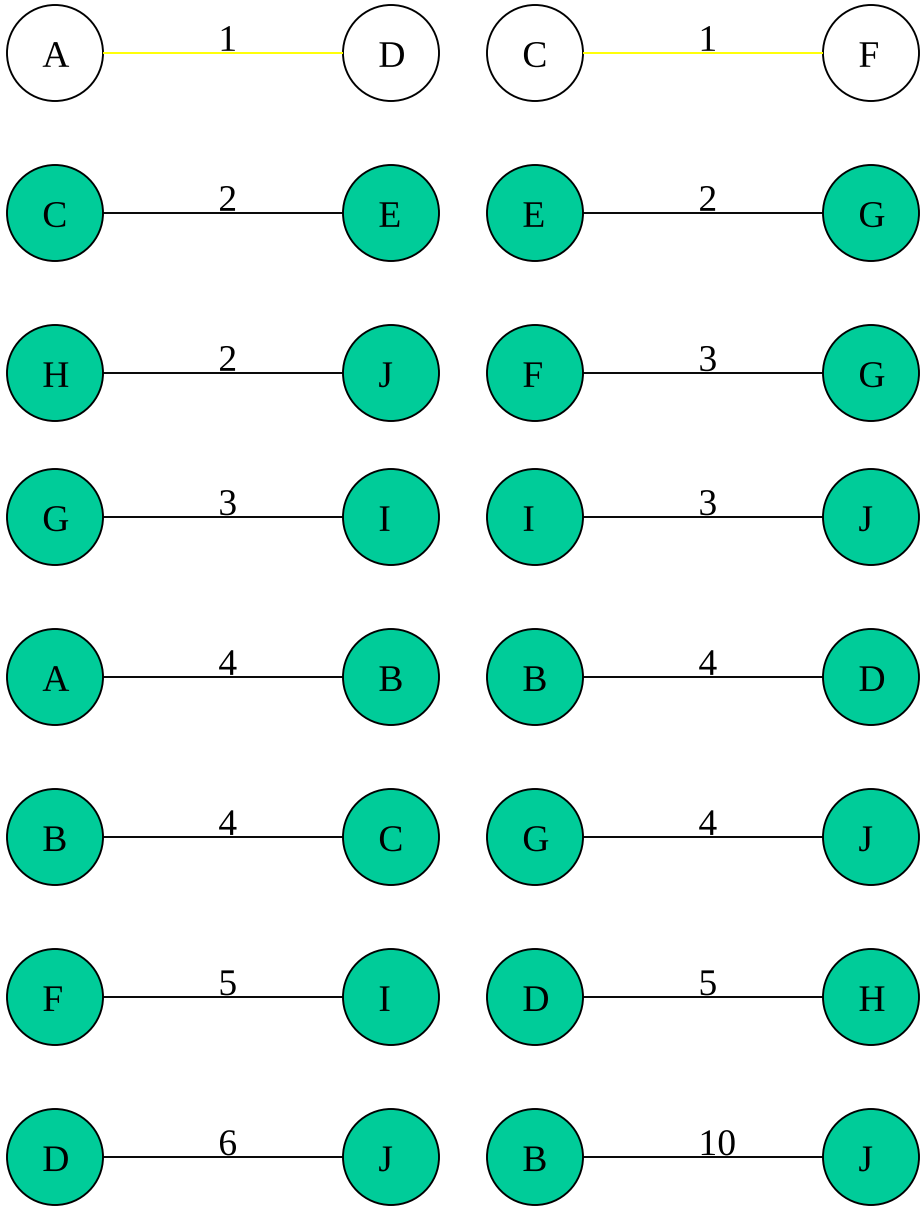


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

Add Edge

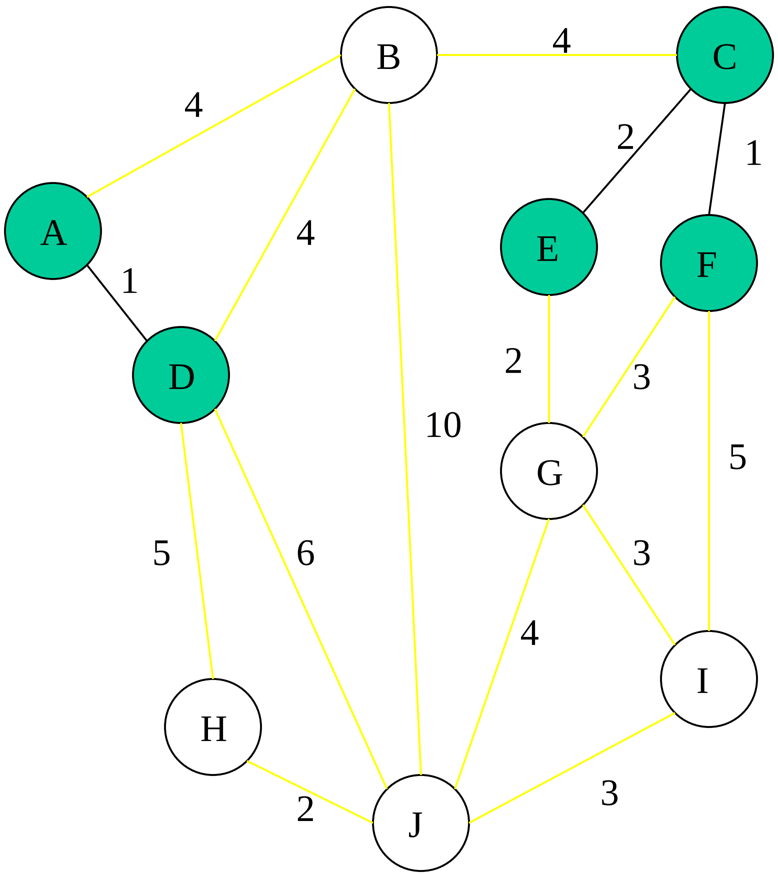


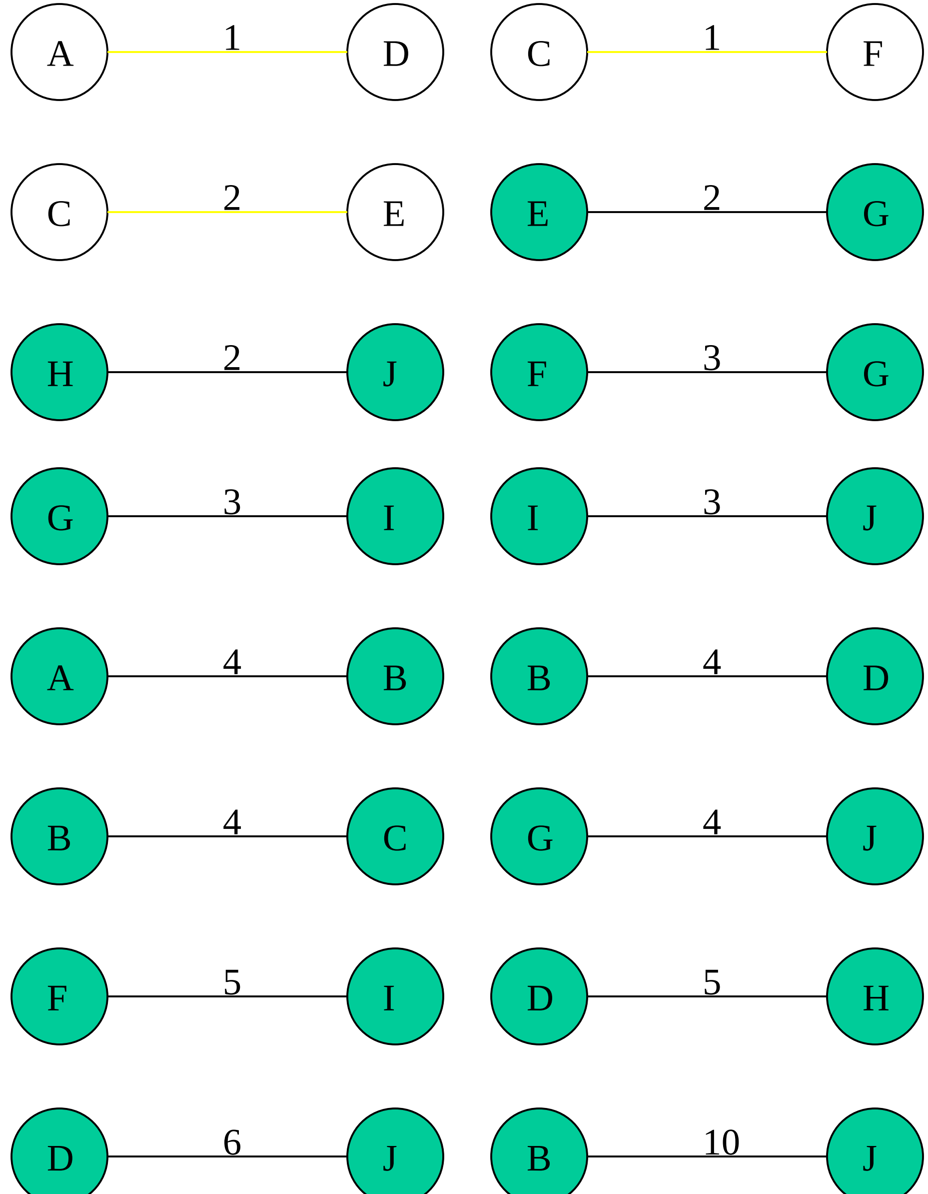
A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
3
10
5
6
5
4
3
3
2
3



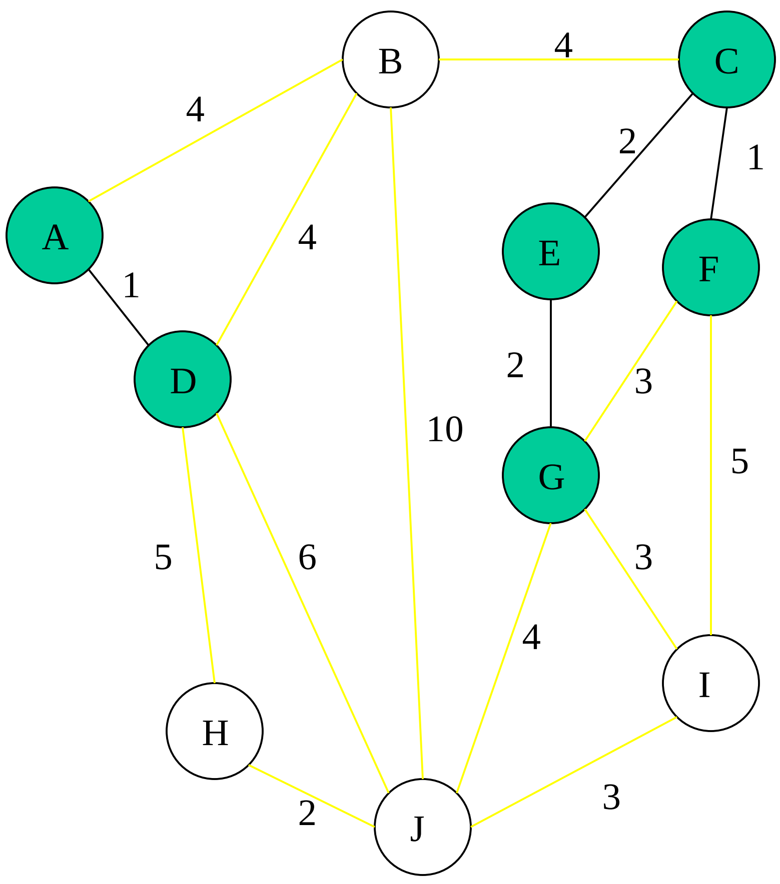
A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

Add Edge

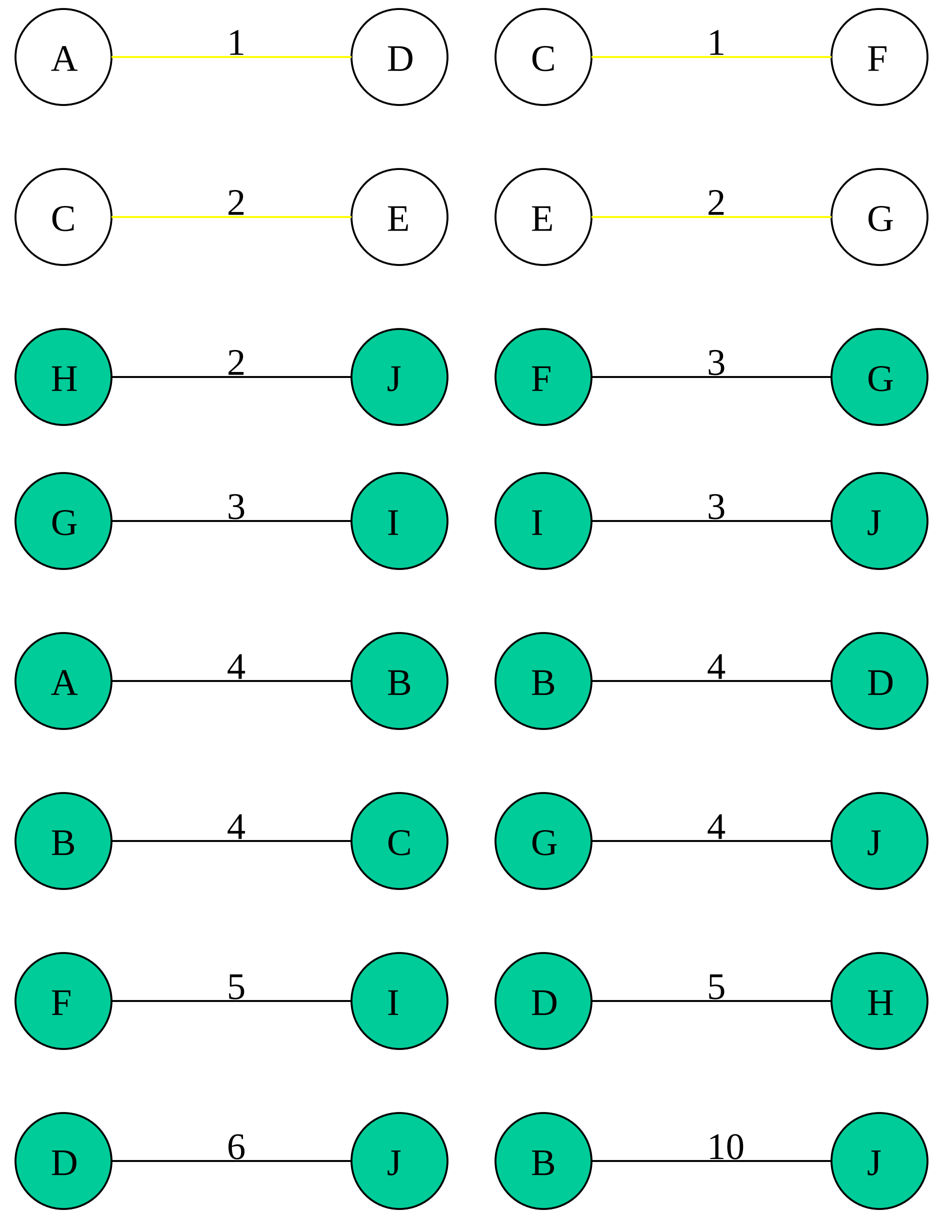

A
B
C
D
E
F
G
H
I
J
4
4
4
1
2
1
2
3
10
5
6
5
3
4
2
3


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

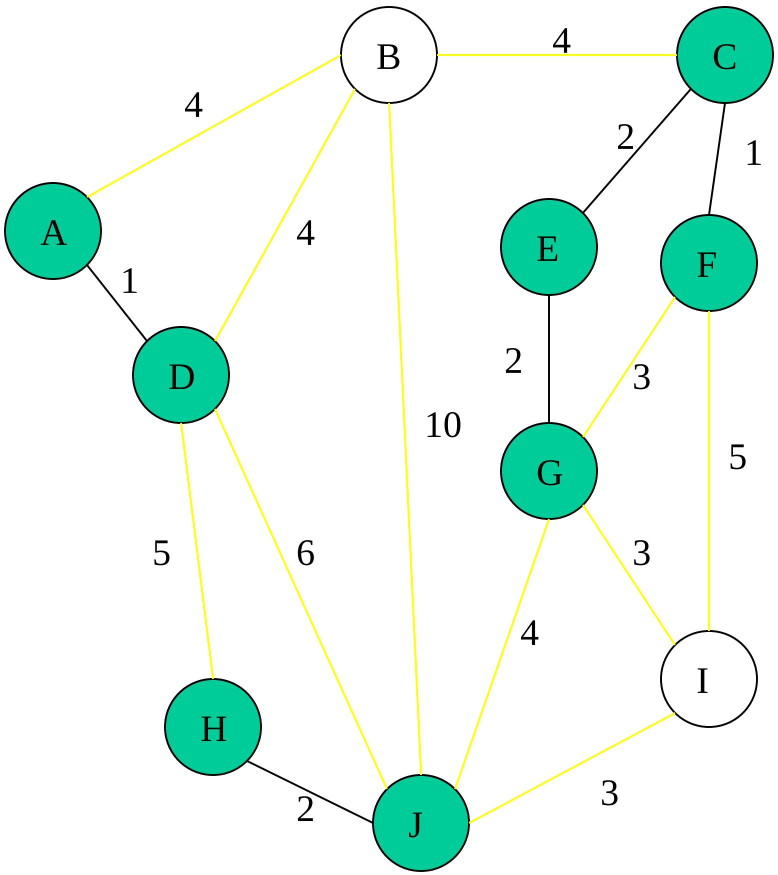


A
B
C
D
E
F
G
H
I
J
4
4
2
1
1
4
2
3
10
5
5
6
3
4
2
3

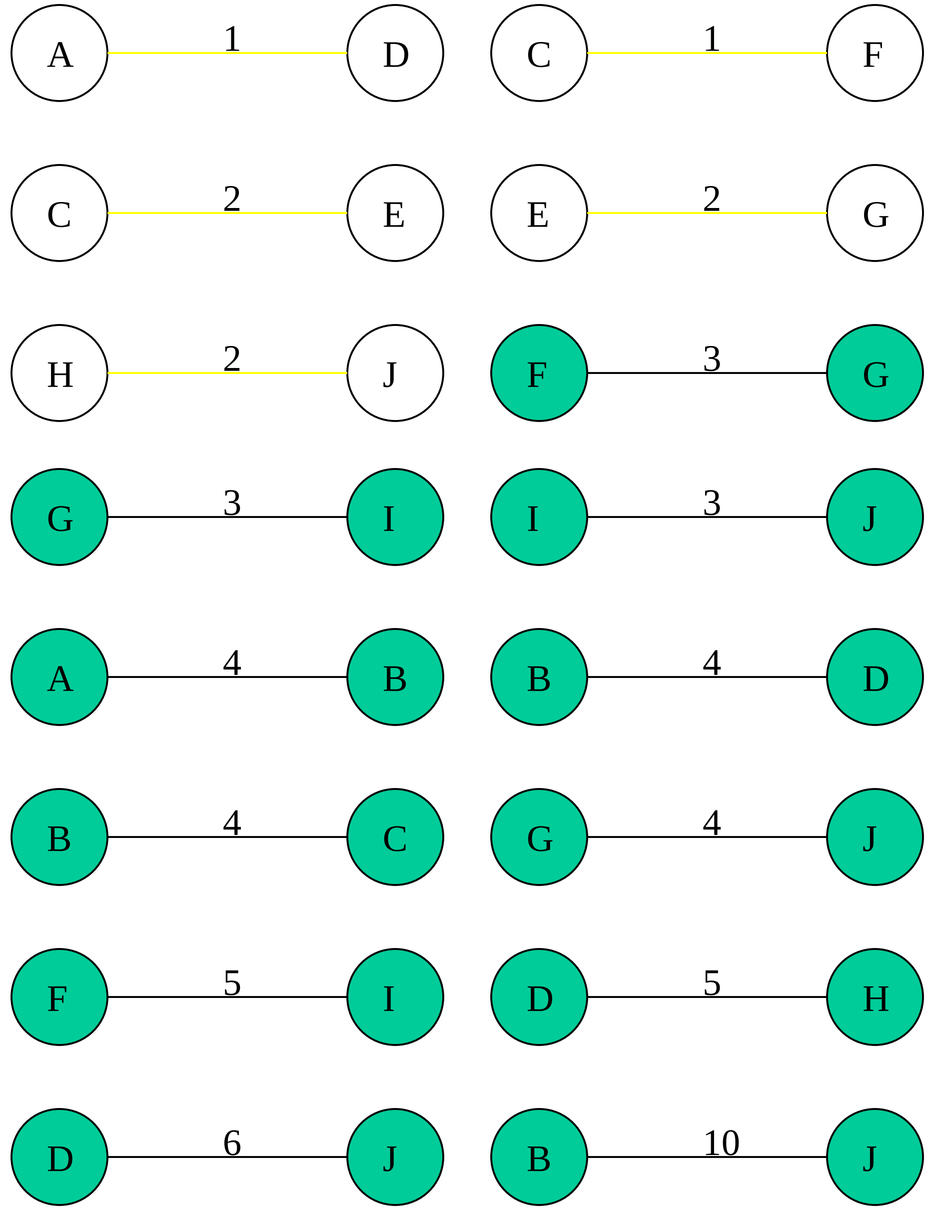


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

Add Edge



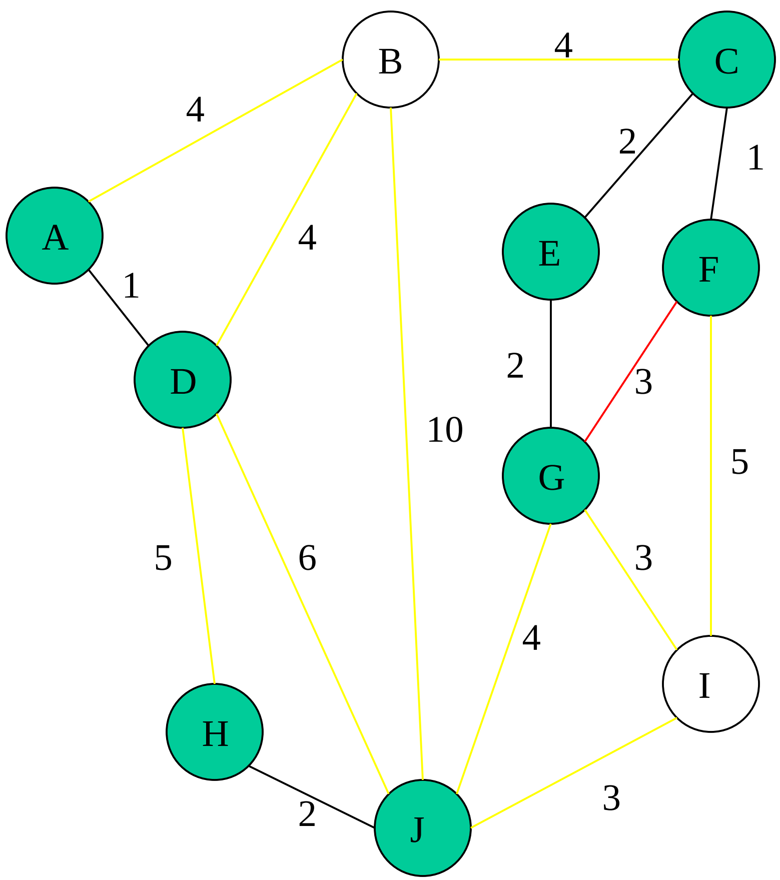
B
C
A
E
F
D
G
H
I
J
4
4
2
1
1
4
2
3
10
5
5
6
3
4
2
3

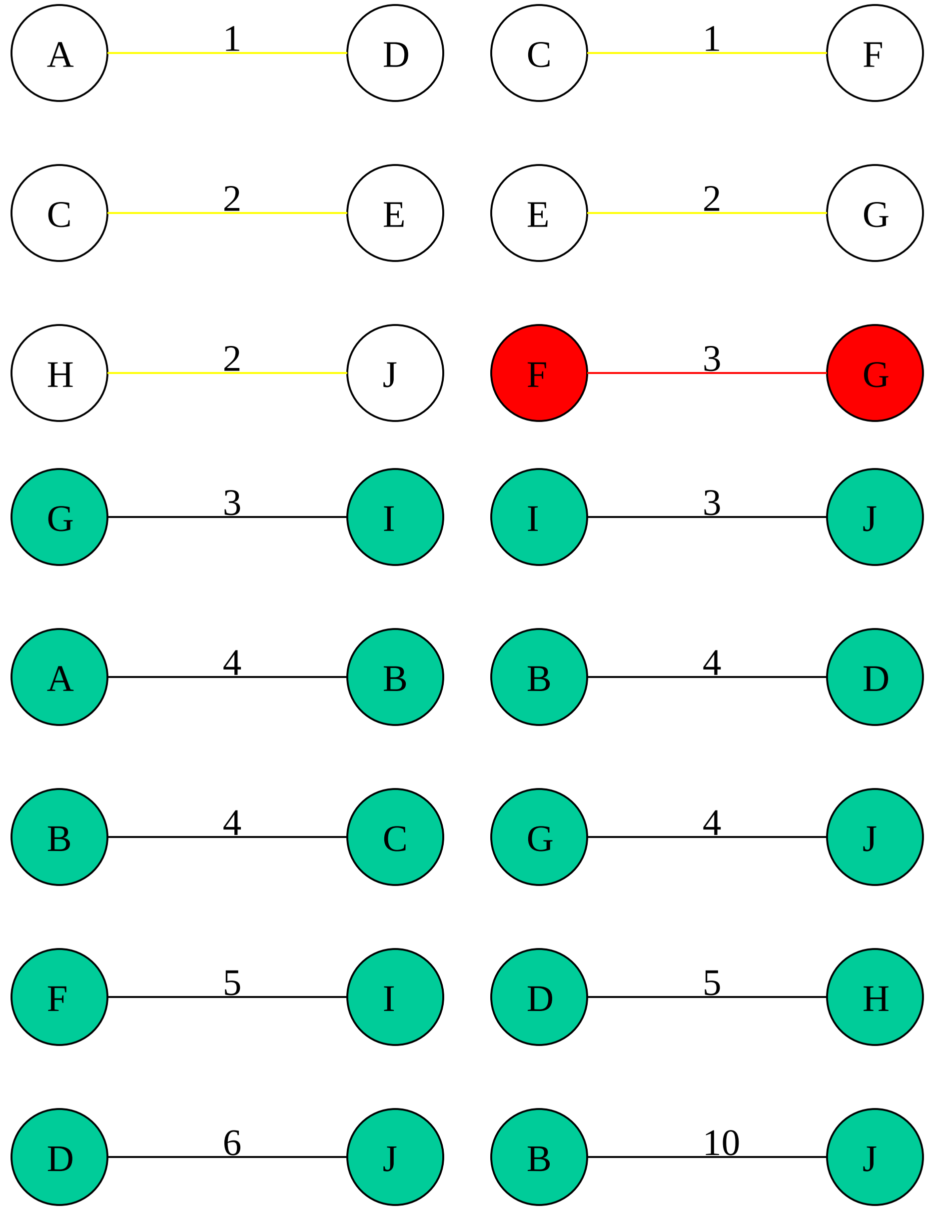


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

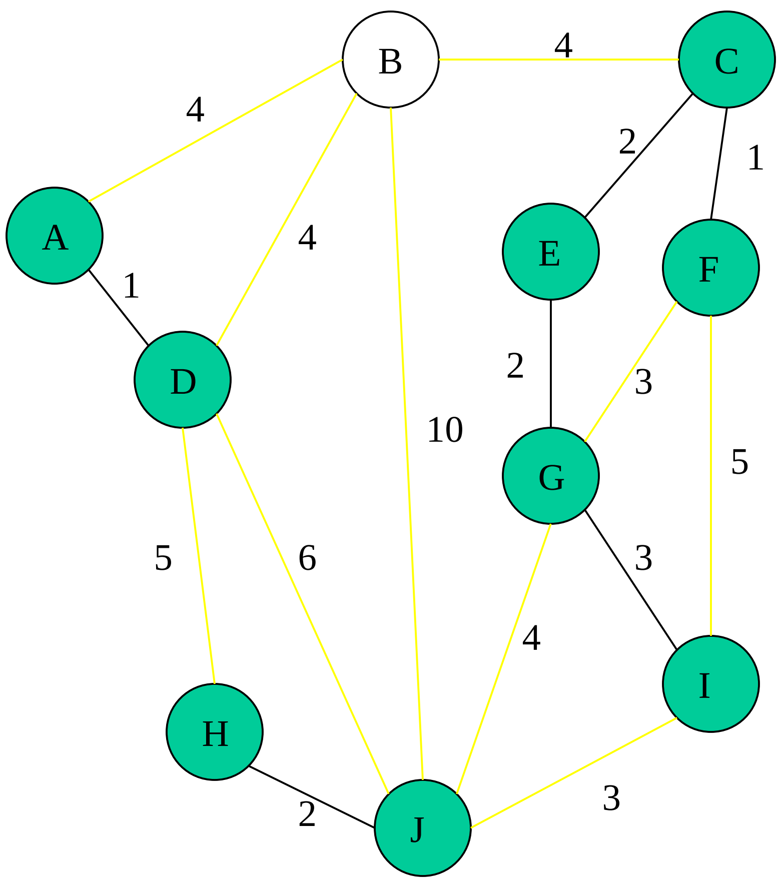
Cycle

Don't Add Edge

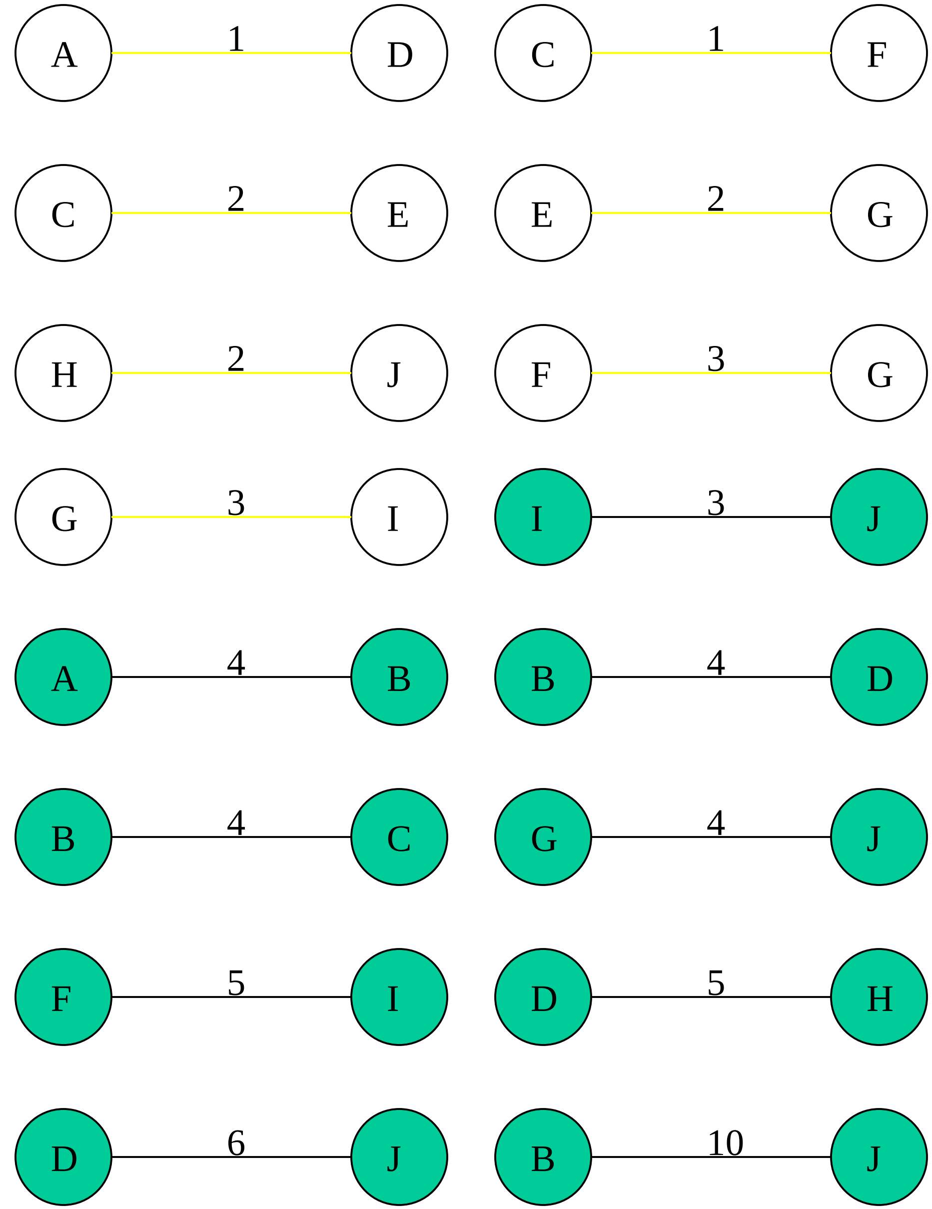




Add Edge

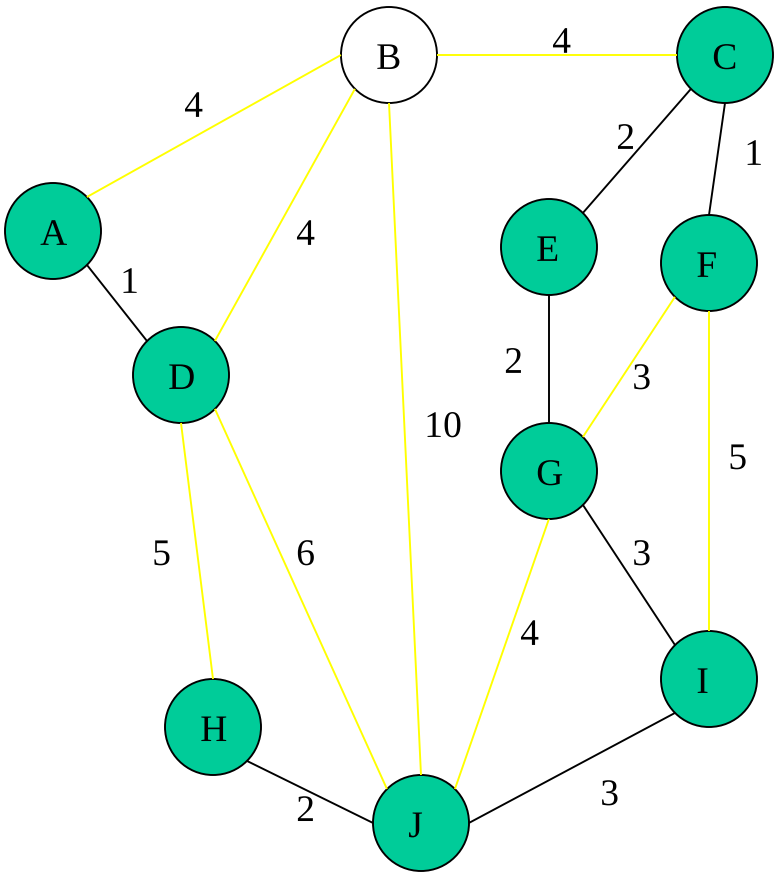


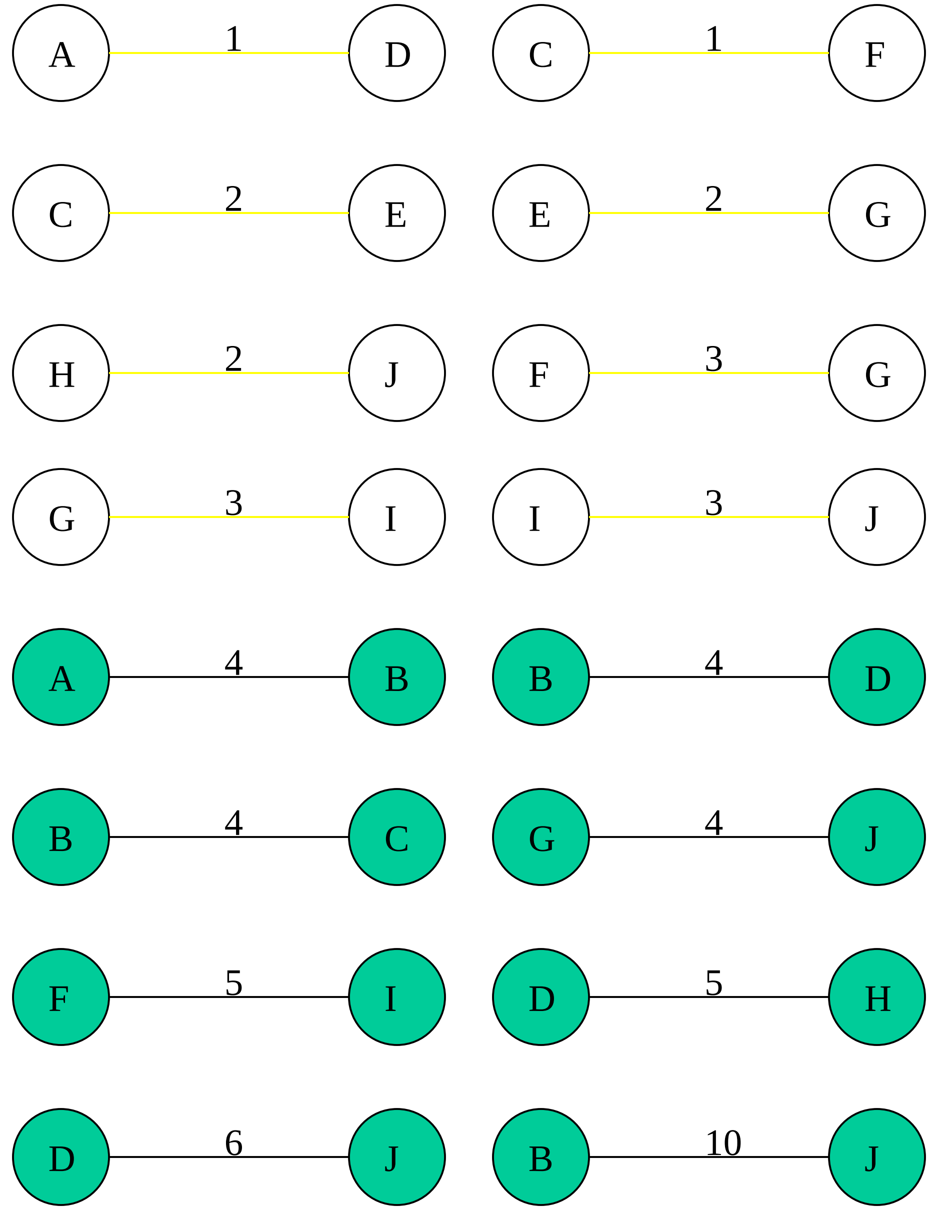
A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
3
5
10
5
6
3
4
2
3



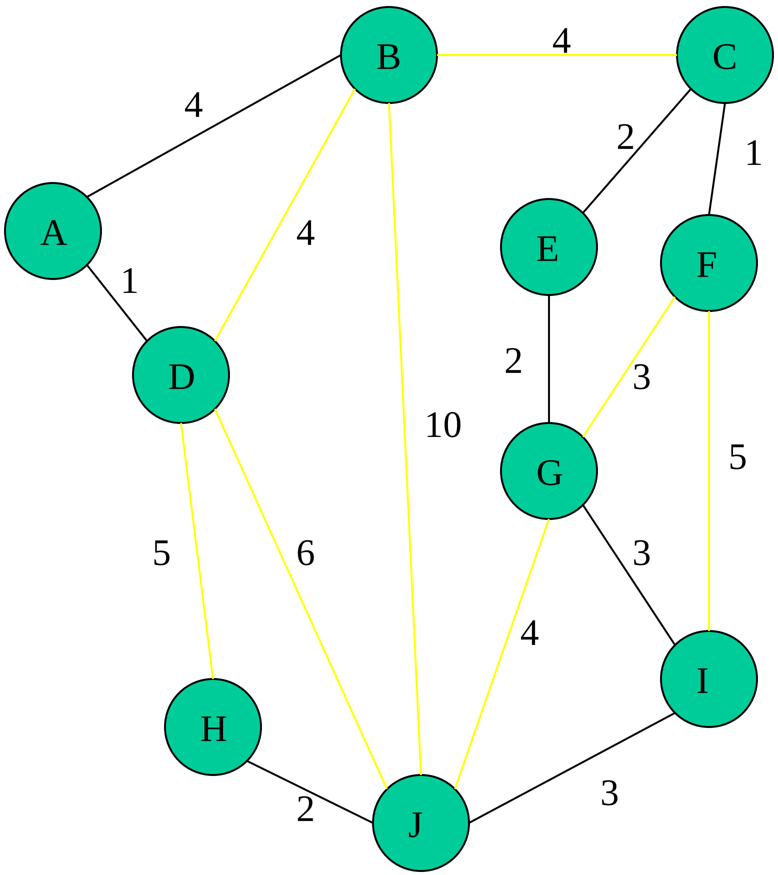
A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

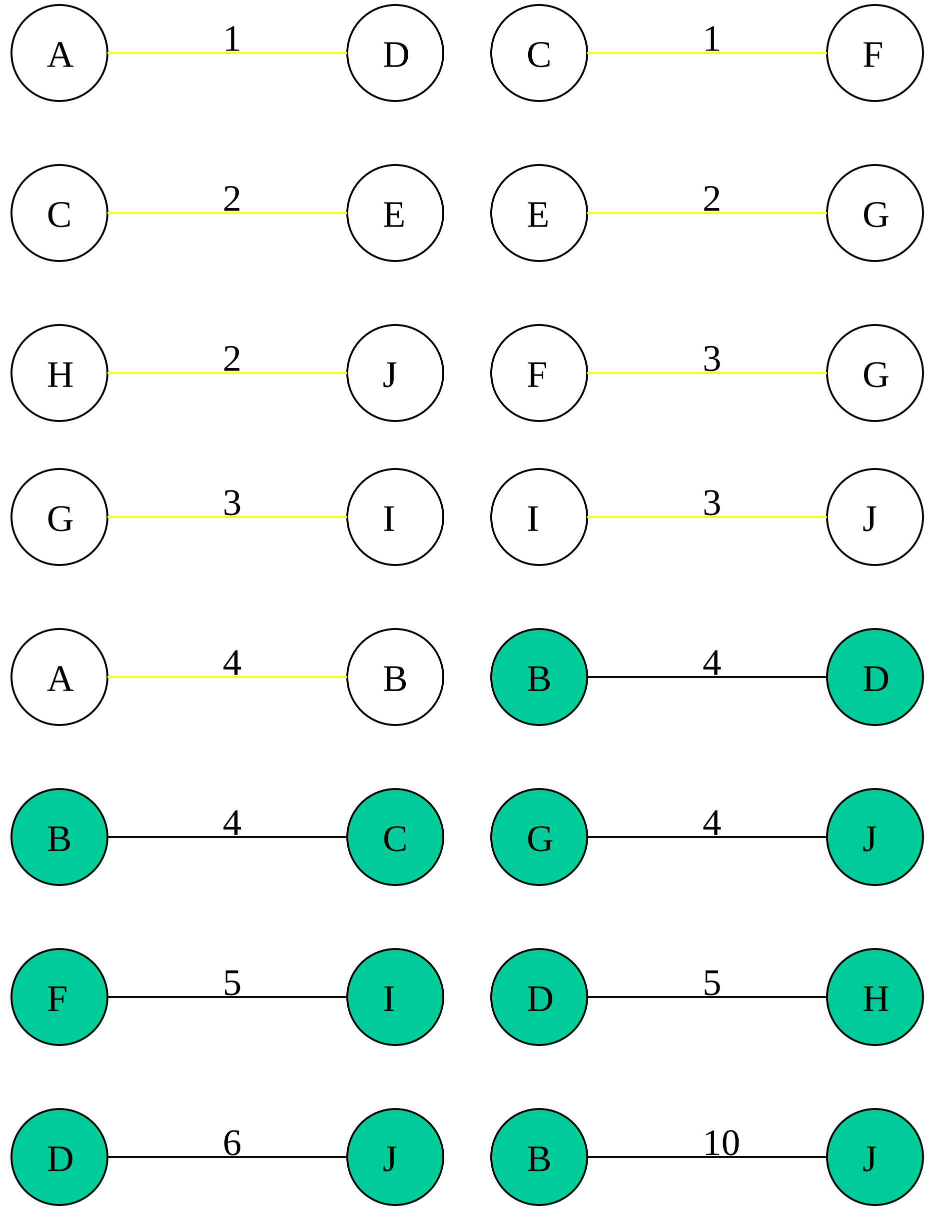
Add Edge


A
B
C
D
E
F
G
H
I
J
4
4
1
4
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

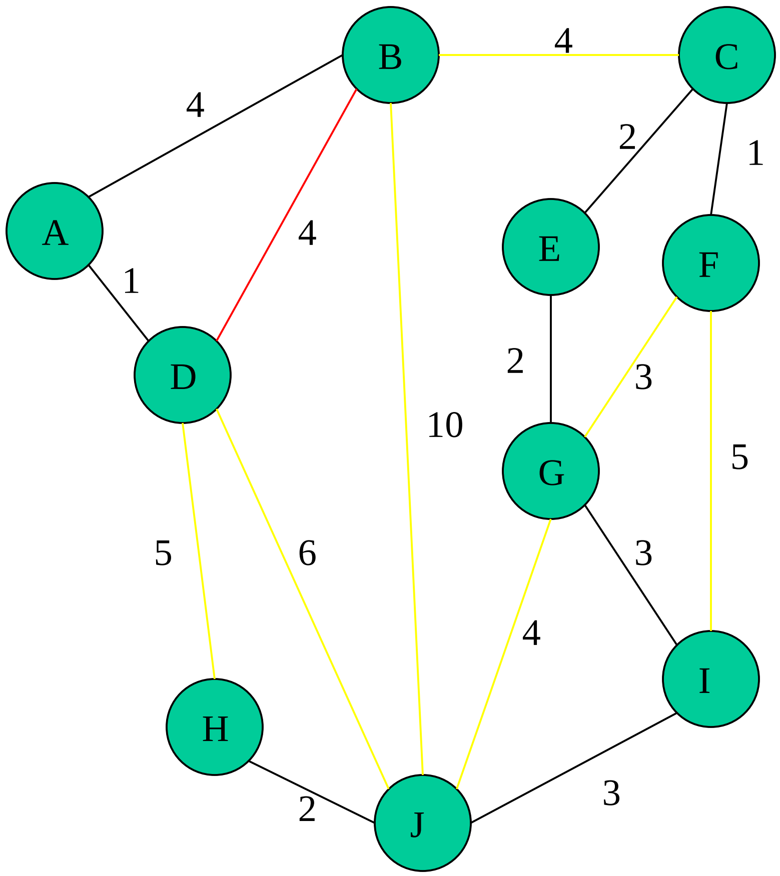
Add Edge

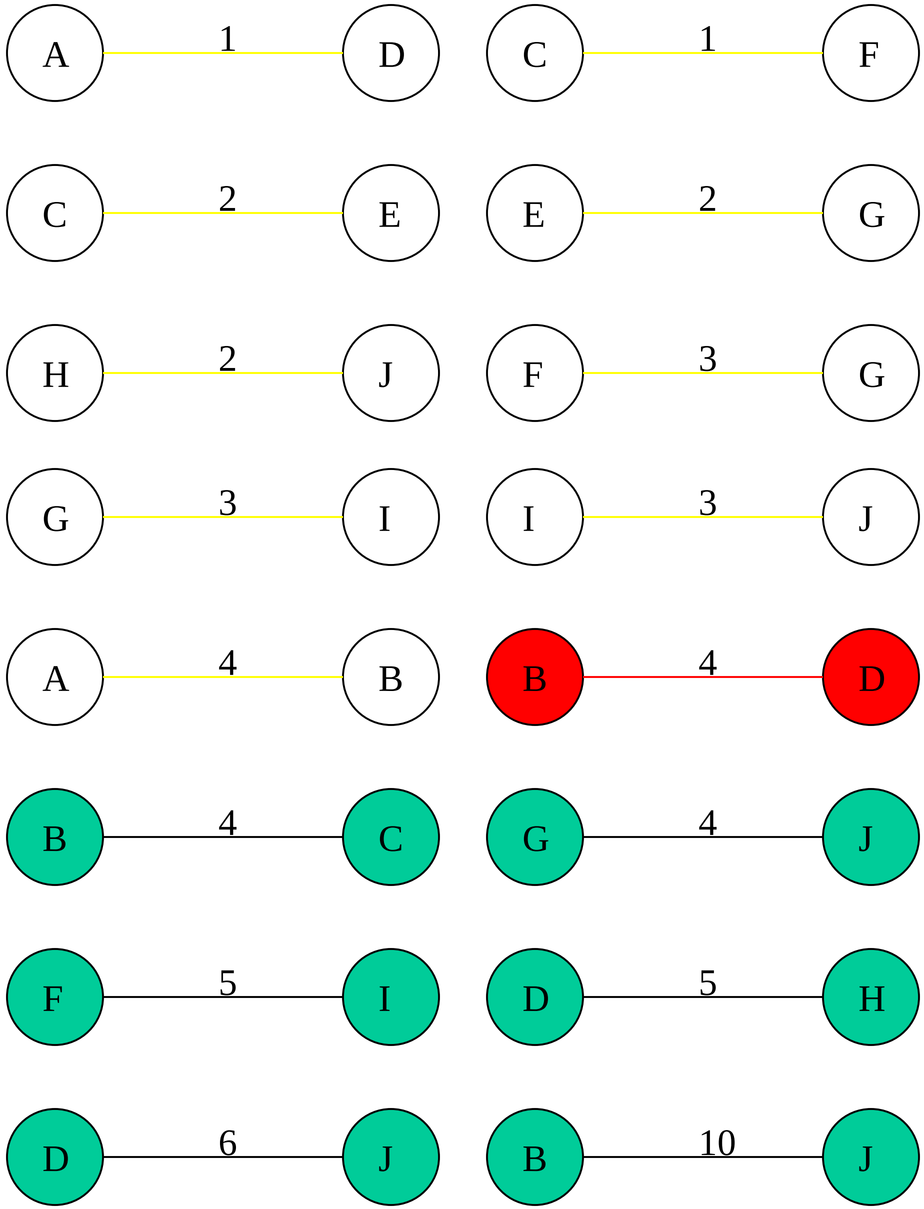

A
B
C
D
E
F
G
H
I
J
4
4
4
1
2
1
2
3
10
5
5
6
3
4
2
3


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

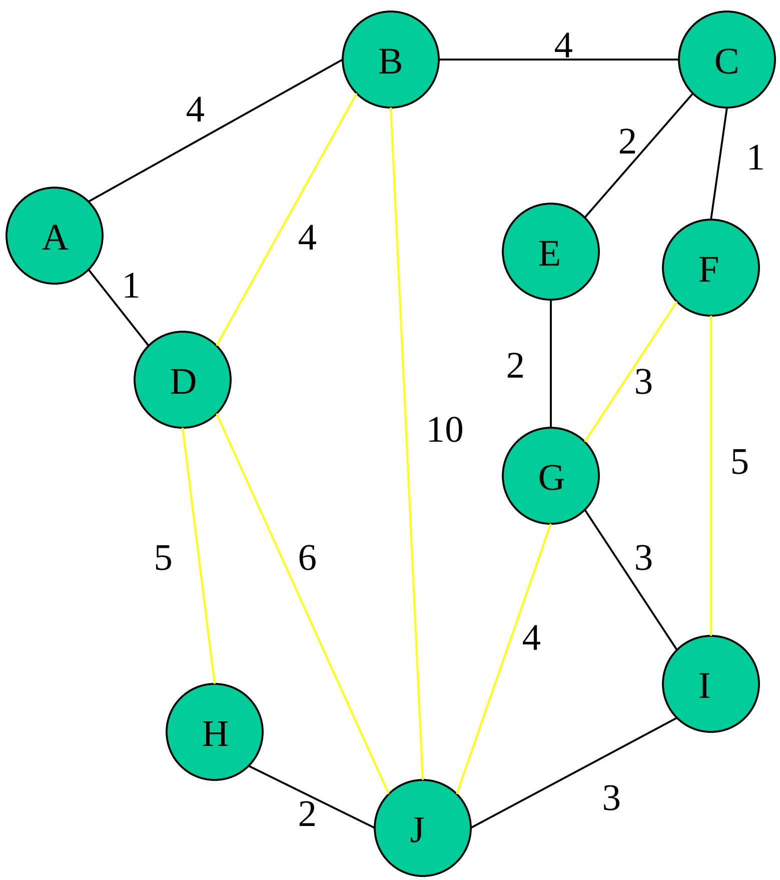
Cycle

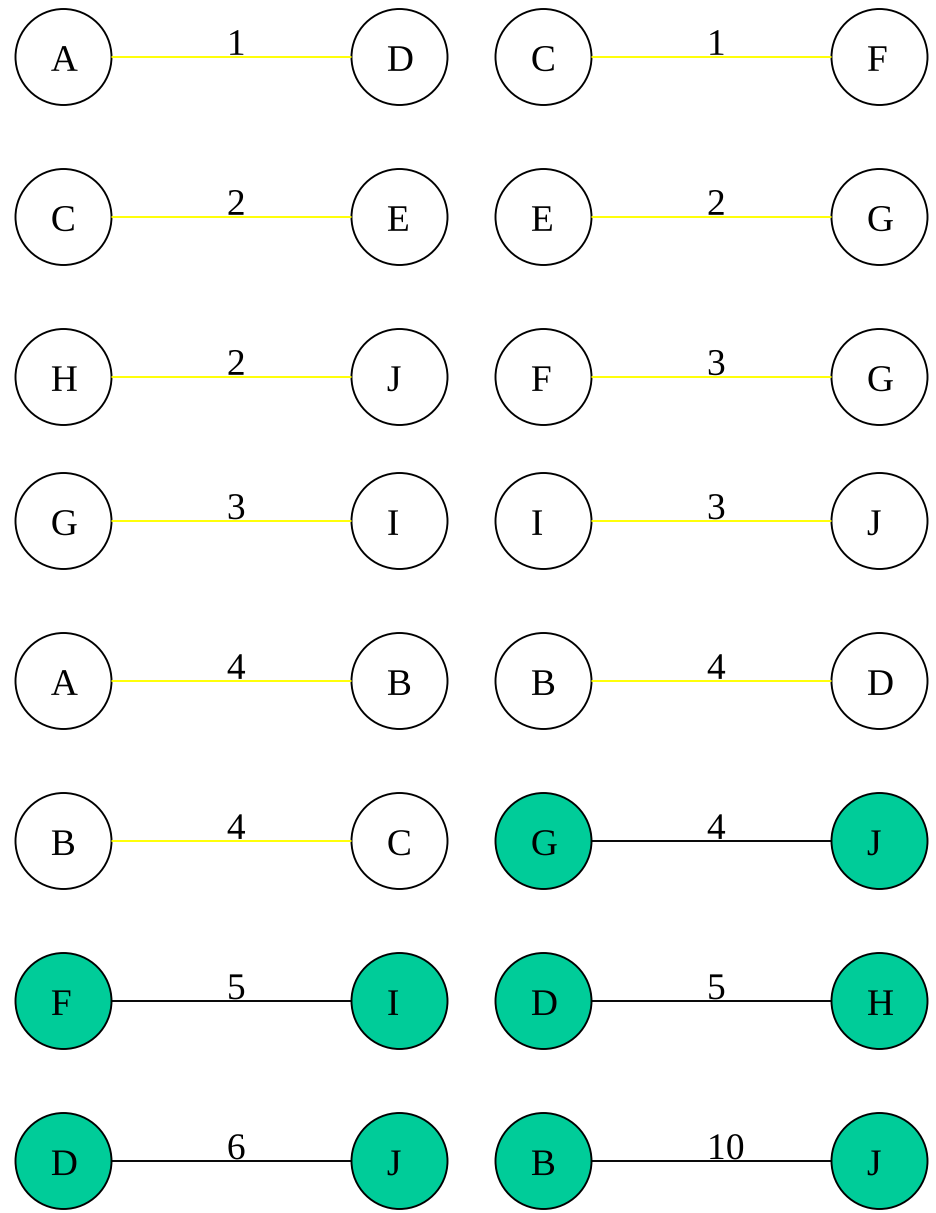
Don’t Add Edge



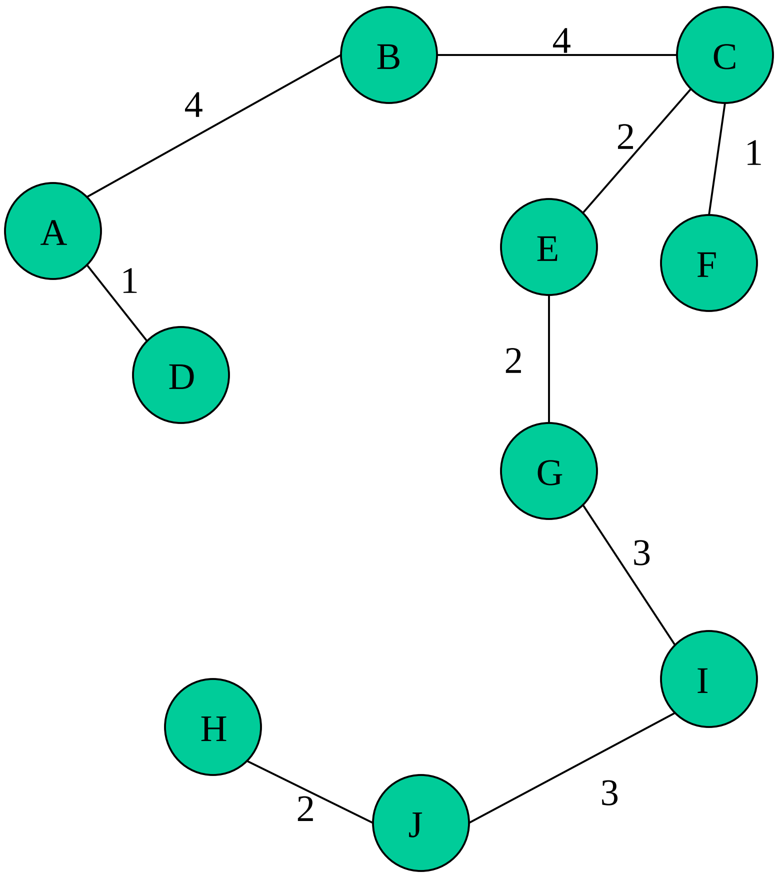


Add Edge

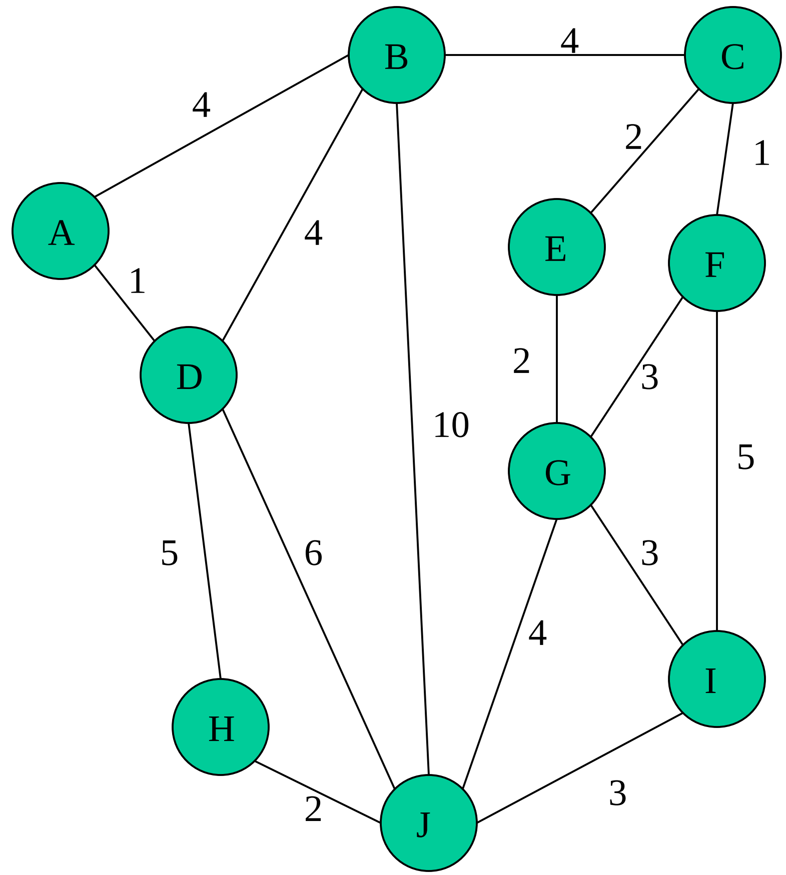

A
B
C
D
E
F
G
H
I
J
4
4
2
1
1
4
2
3
10
5
5
6
3
4
2
3


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

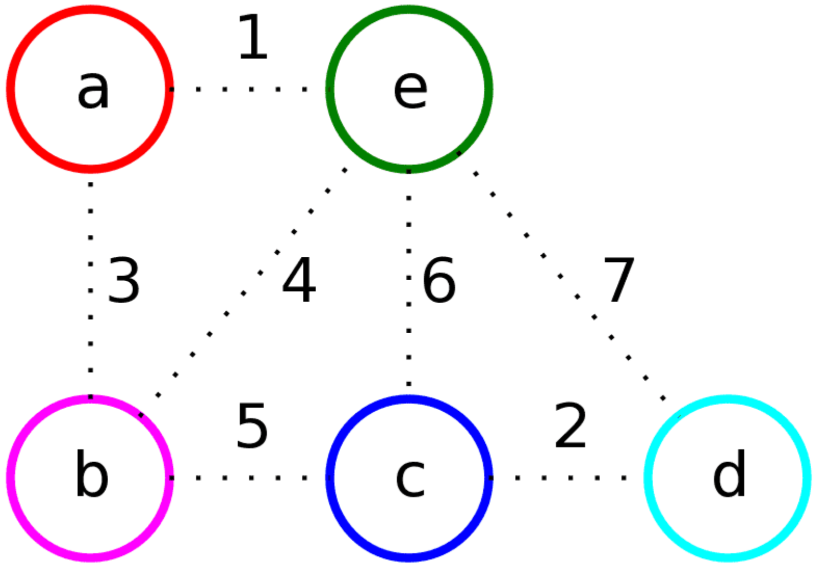


Complete Graph



Visualization of Kruskal's algorithm

Edge	ab	ae	bc	be	cd	ed	ec
Weight	3	1	5	4	2	7	6



"repeatedly add the cheapest **edge** that does not create a cycle"

Time complexity of Kruskal's Algorithm

Running Time = $O(E \log E)$ (E = # edges)

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

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

We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time; this allows the step "remove an edge with minimum weight from S" to operate in constant time. Next, we use a disjoint-set data structure (Union&Find) to keep track of which vertices are in which components. We need to perform $O(V)$ operations, as in each iteration we connects a vertex to the spanning tree, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform $O(V)$ operations in $O(V \log V)$ time. Thus the total time is $O(E \log E) = O(E \log V)$.

Proof of Kruskal's Algorithm

Theorem. After running Kruskal's algorithm on a connected weighted graph G, its output T is a minimum weight spanning tree.

Proof. First, T is a spanning tree. This is because:

- T **is a forest.** No cycles are ever created.
- T **is spanning.** Suppose that there is a vertex v that is not incident with the edges of T. Then the incident edges of v must have been considered in the algorithm at some step. The first edge (in edge order) would have been included because it could not have created a cycle, which contradicts the definition of T.
- T **is connected.** Suppose that T is not connected. Then T has two or more connected components. Since G is connected, then these components must be connected by some edges in G, not in T. The first of these edges (in edge order) would have been included in T because it could not have created a cycle, which contradicts the definition of T.

Proof of Kruskal's Algorithm 2

Second, T is a spanning tree of minimum weight. We will prove this using induction. Let T^* be a minimum-weight spanning tree. If $T = T^*$, then T is a minimum weight spanning tree. If $T \neq T^*$, then there exists an edge $e \in T^*$ **of minimum weight** that is not in T. Further, $T \cup e$ contains a cycle C such that:

a. Every edge in C has weight less than $\mathrm{wt}(e)$. (This follows from how the algorithm constructed T.)

b. There is some edge f in C that is not in T^*. (Because T^* does not contain the cycle C.)

Consider the tree $T_2 = T \setminus \{e\} \cup \{f\}$:

a. T_2 is a spanning tree.

b. T_2 has more edges in common with T^* than T did.

c. And $\mathrm{wt}(T_2) \geq \mathrm{wt}(T)$. (We exchanged an edge for one that is no more expensive.)

We can redo the same process with T_2 to find a spanning tree T_3 with more edges in common with T^*. By induction, we can continue this process until we reach T^*, from which we see

$$\mathrm{wt}(T) \leq \mathrm{wt}(T_2) \leq \mathrm{wt}(T_3) \leq \cdots \leq \mathrm{wt}(T^*).$$

Since T^* is a minimum weight spanning tree, then these inequalities must be equalities and we conclude that T is a minimum weight spanning tree.

Prim's algorithm

The steps are:

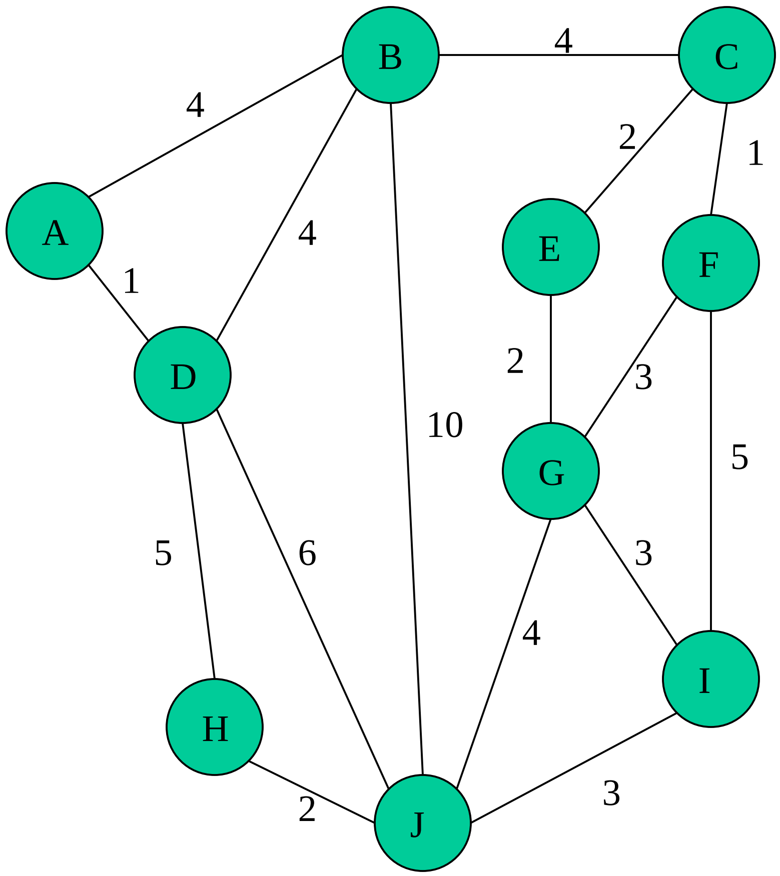
1. Initialize a tree with a single node arbitrarily chose from graph.
2. Repeat until all nodes are in the tree:

    (1). **Find the node** from the graph **with the smallest connecting edge** to the tree,

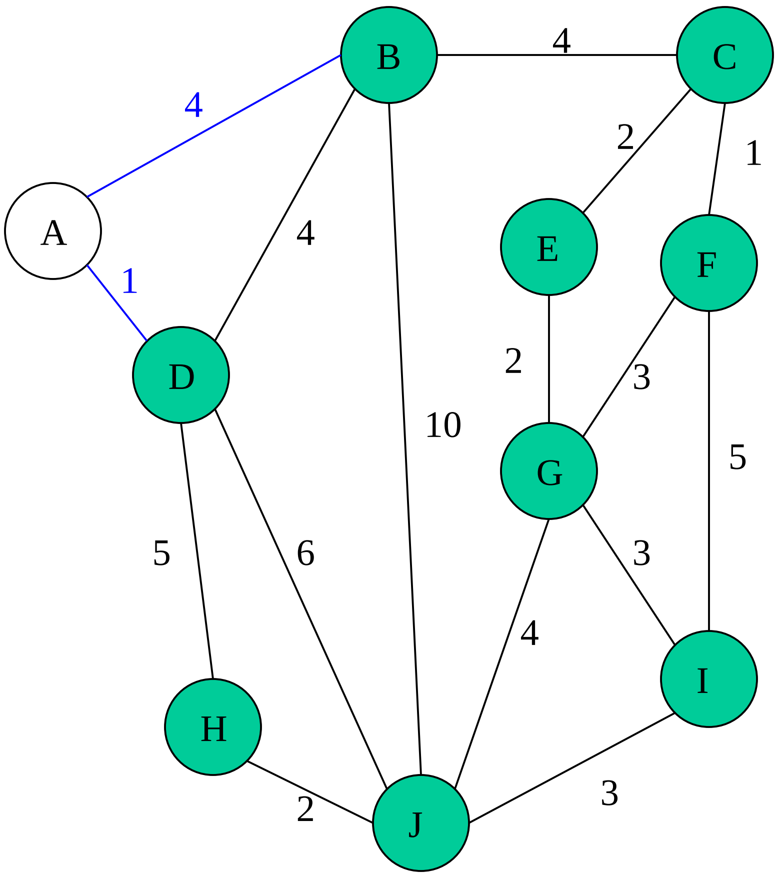
    (2). **Add it to the tree**

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

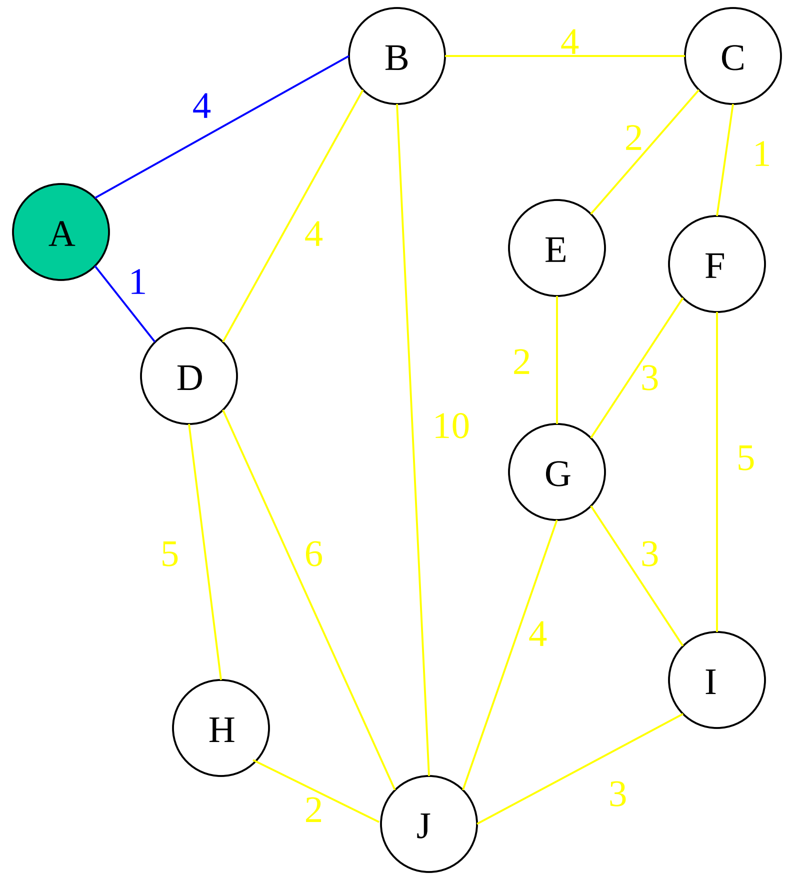
Complete Graph



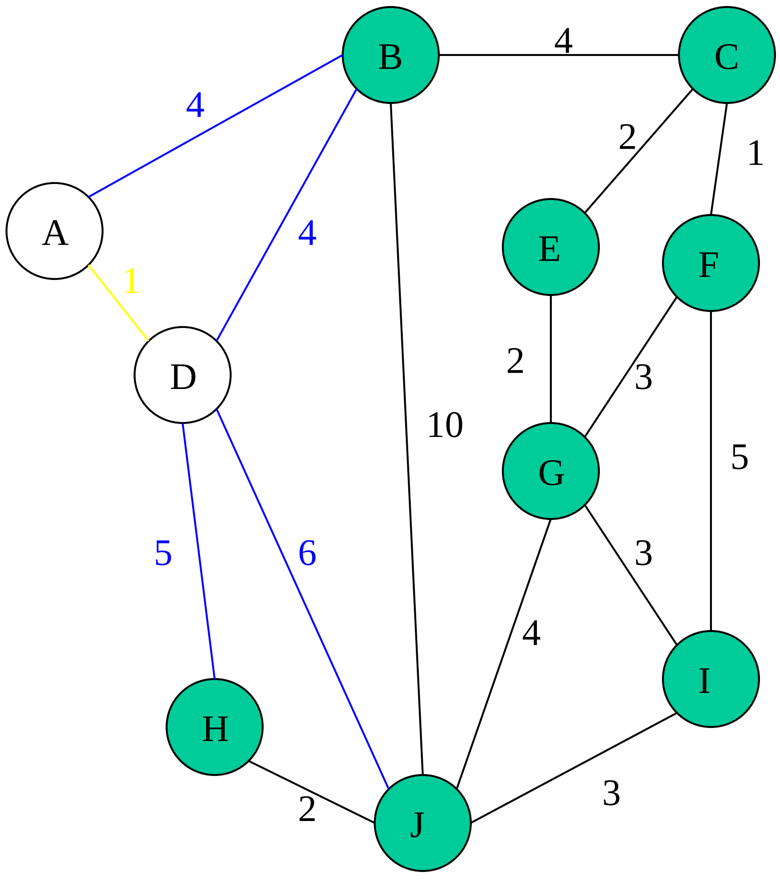
Old Graph



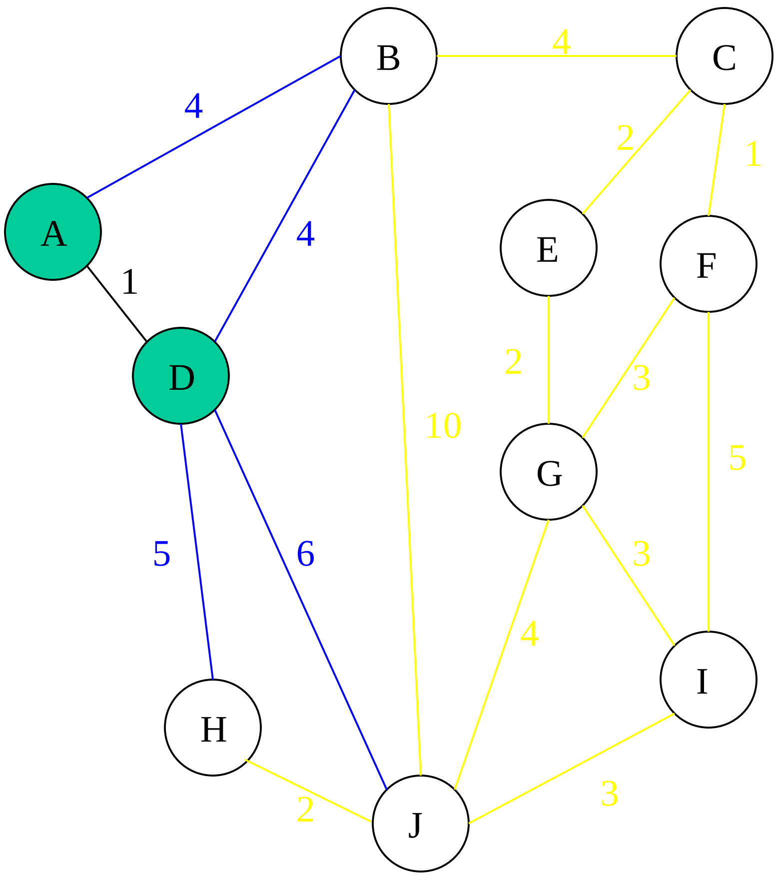
New Graph



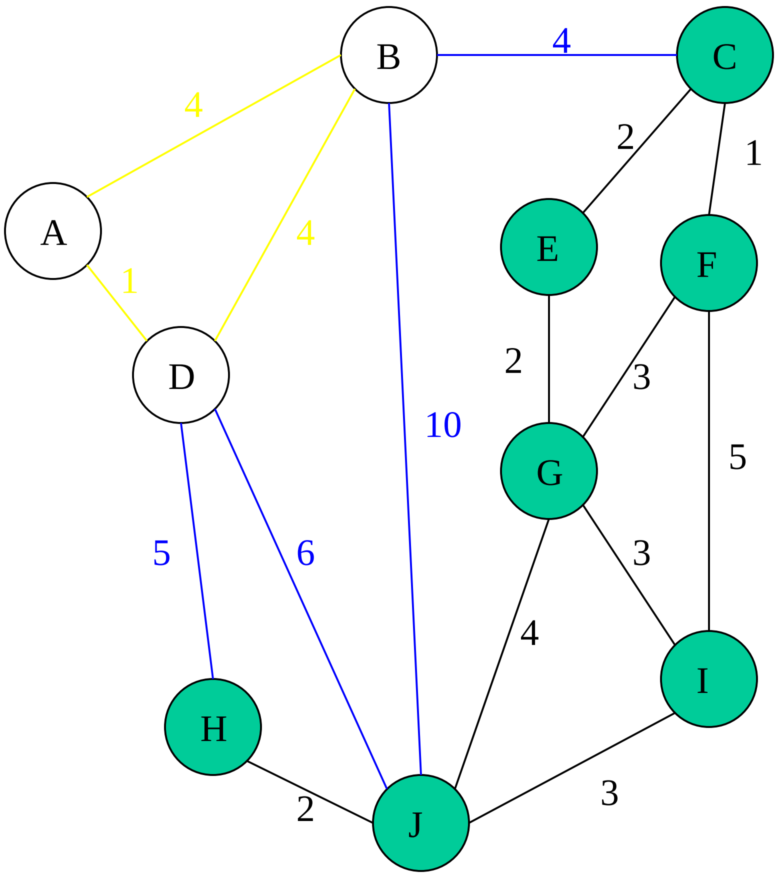
Old Graph



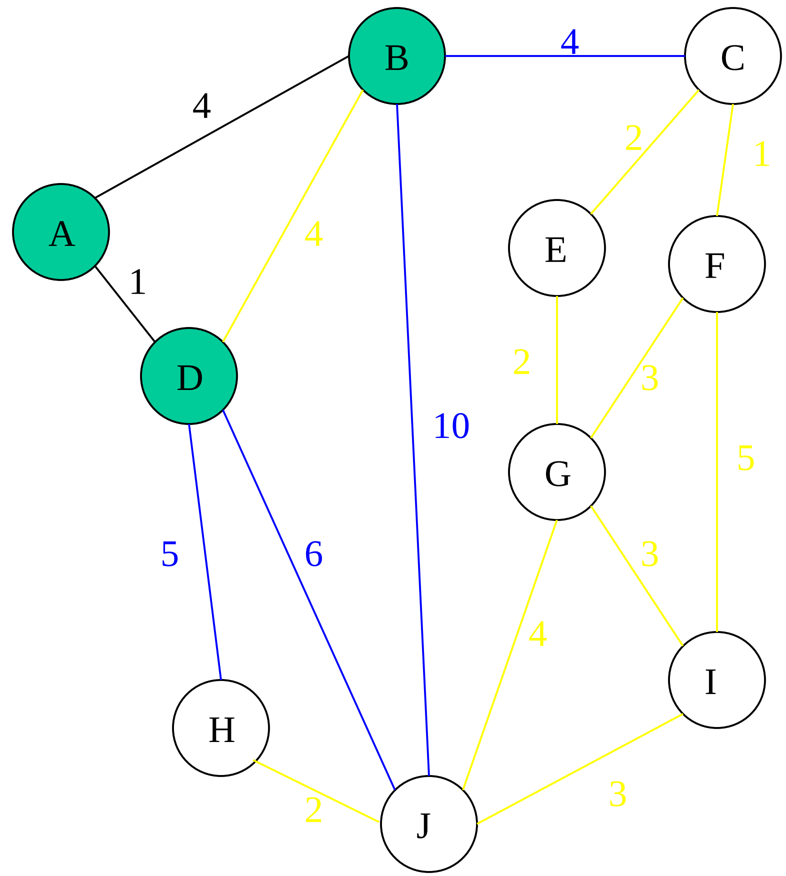
New Graph



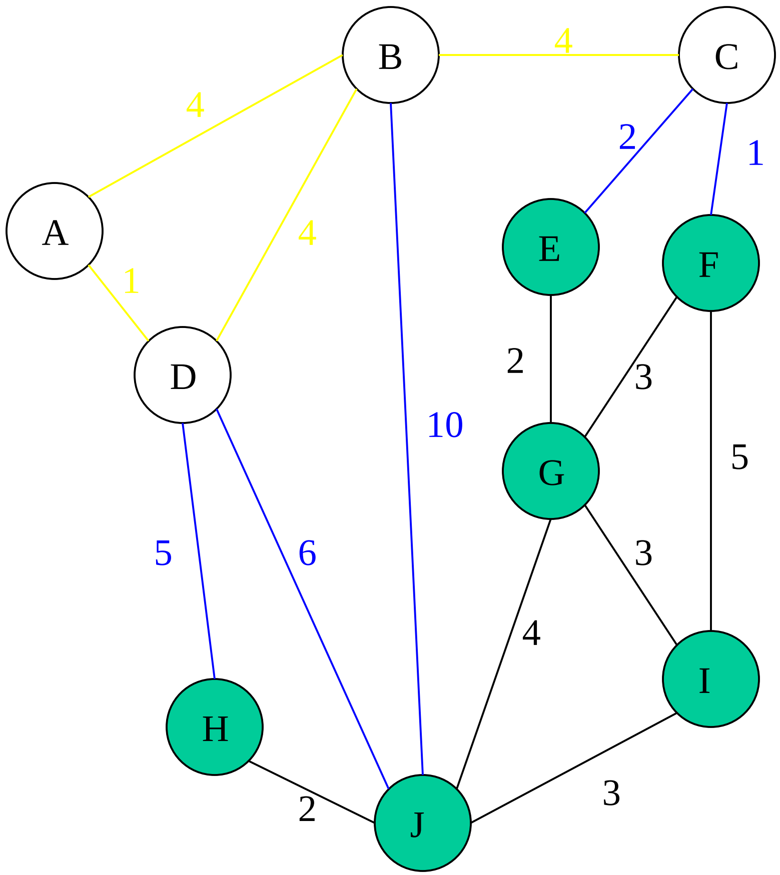
Old Graph



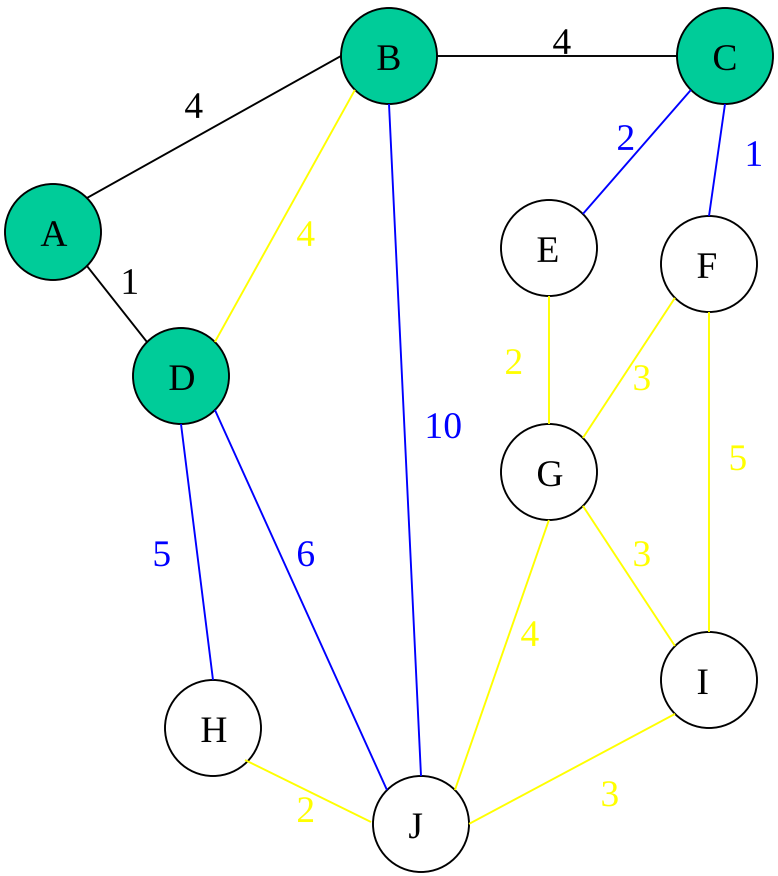
New Graph



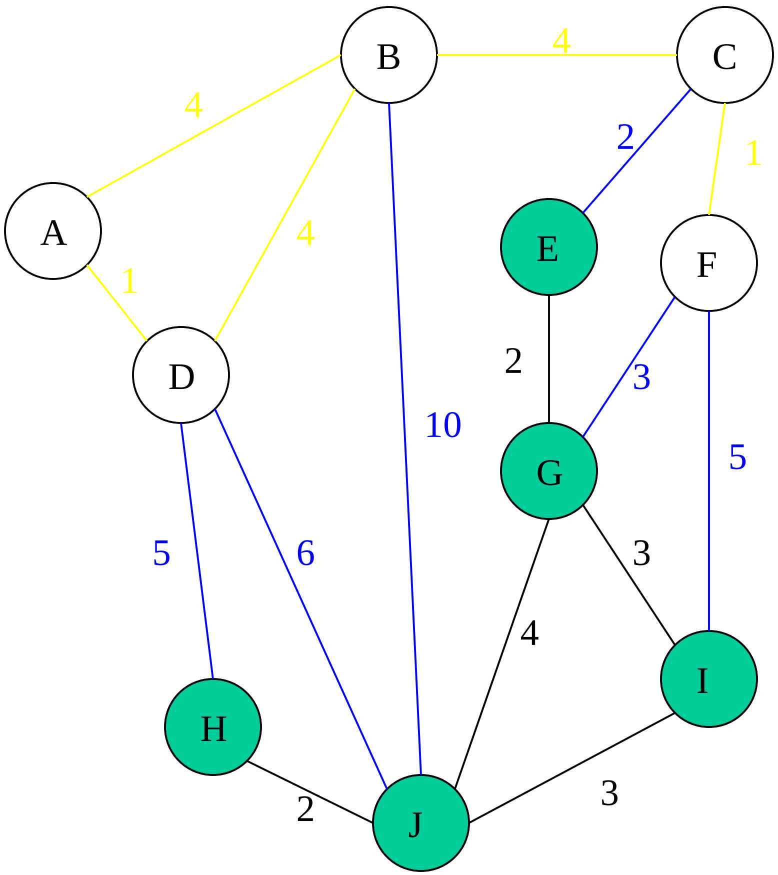
Old Graph



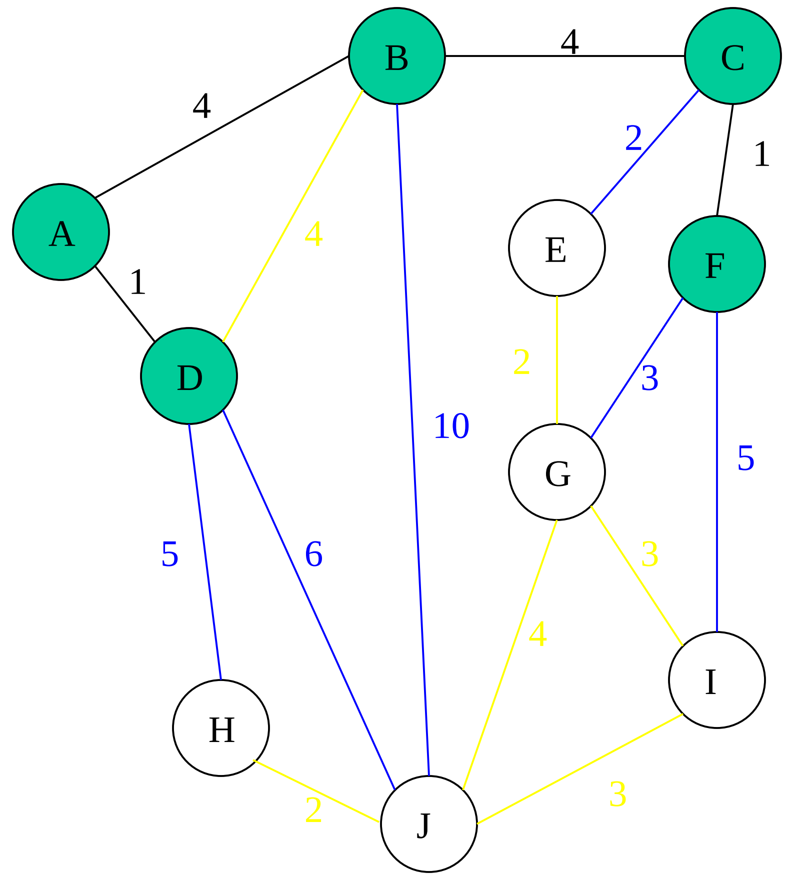
New Graph



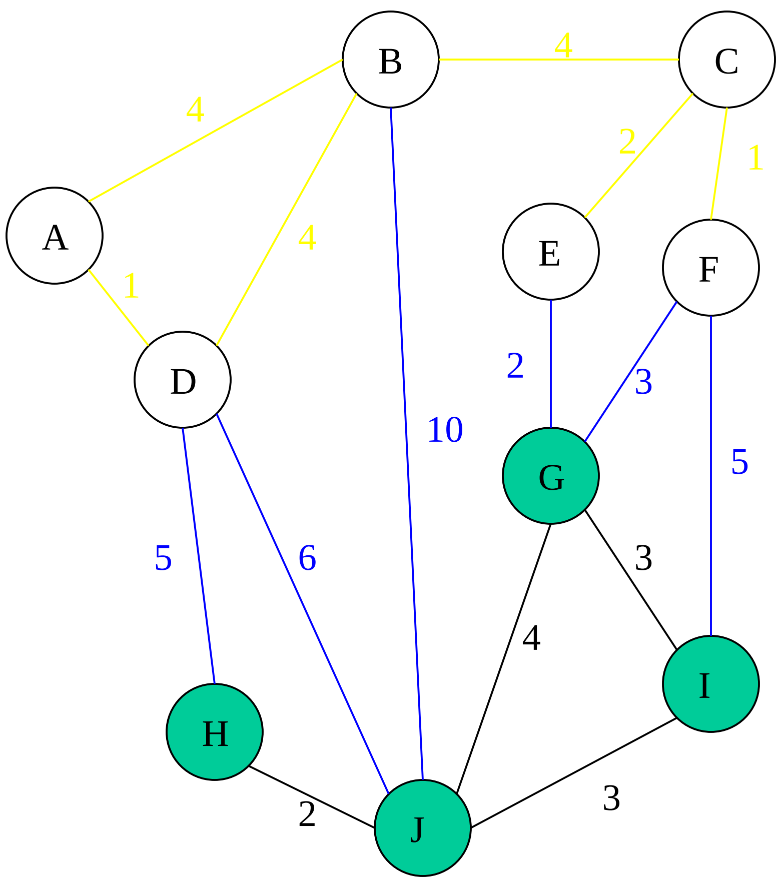
Old Graph



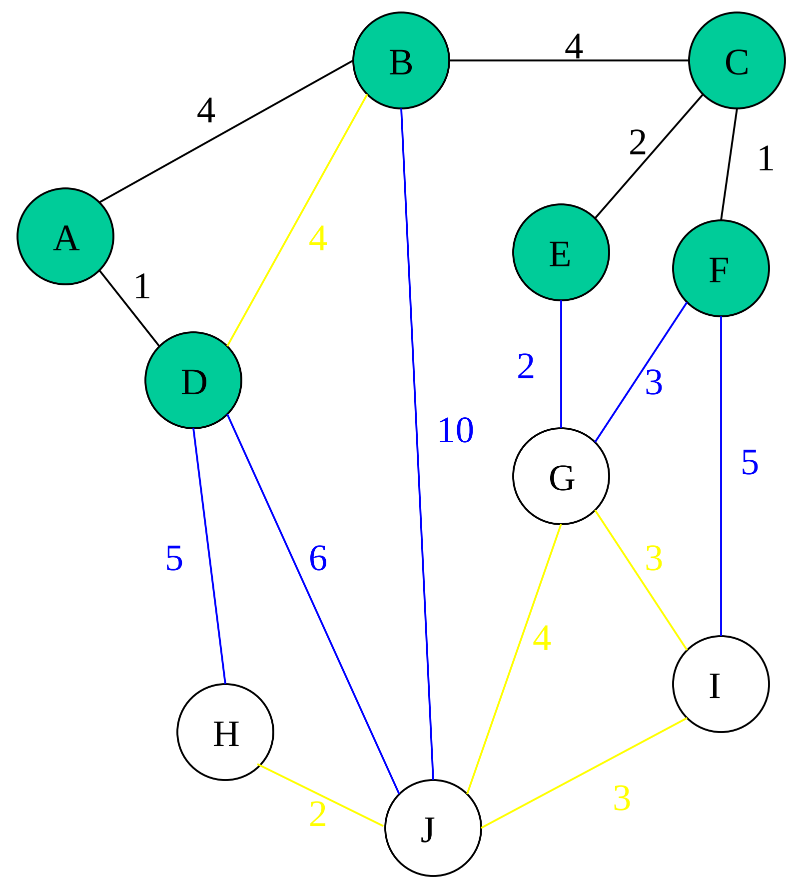
New Graph



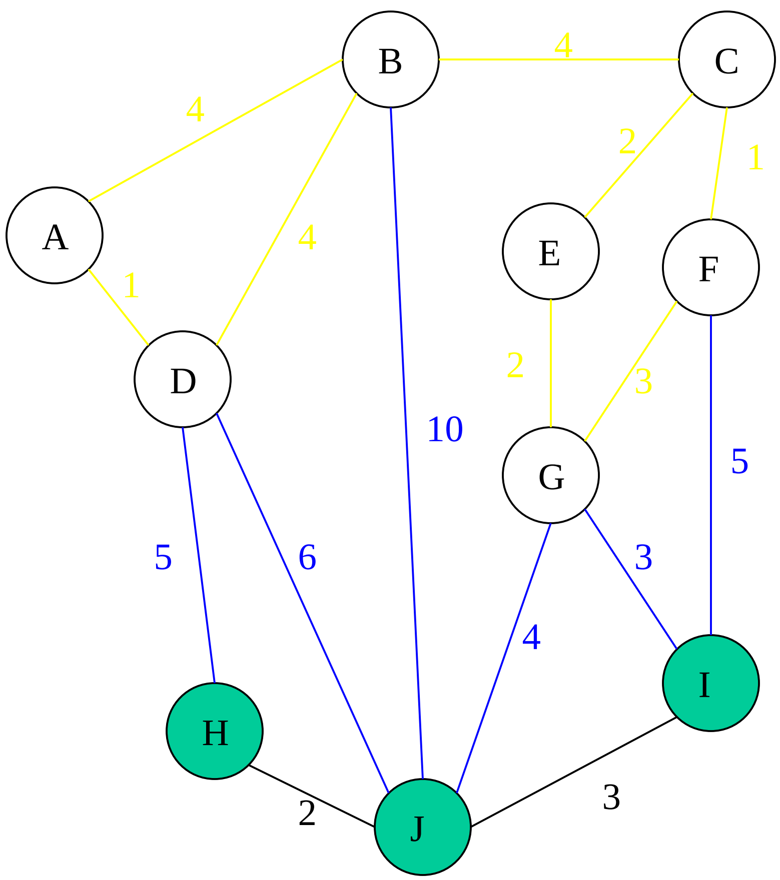
Old Graph



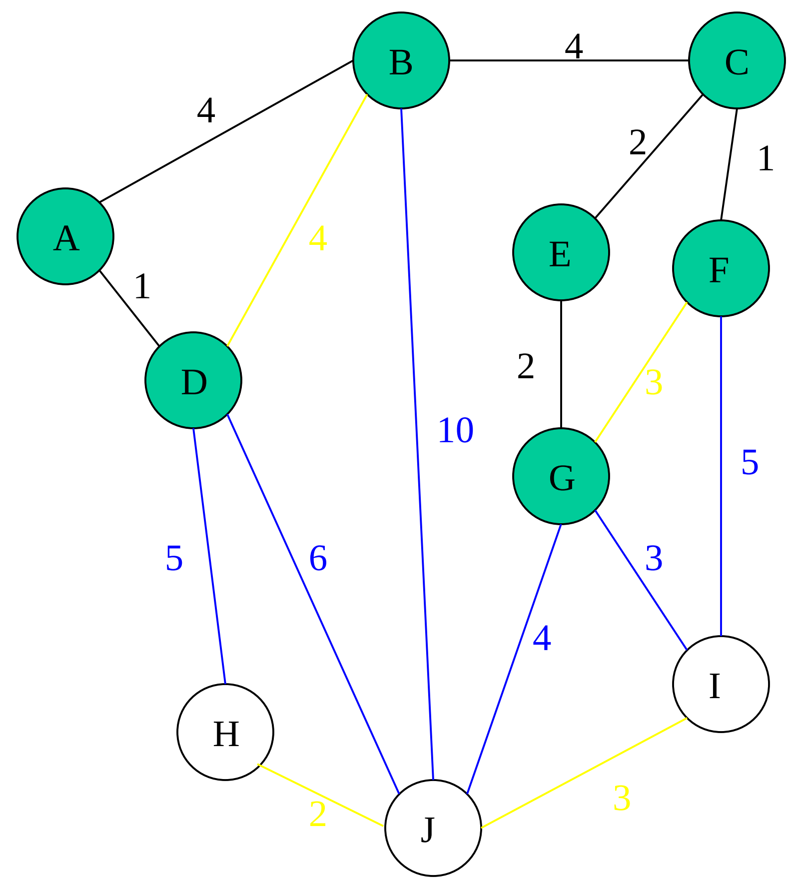
New Graph



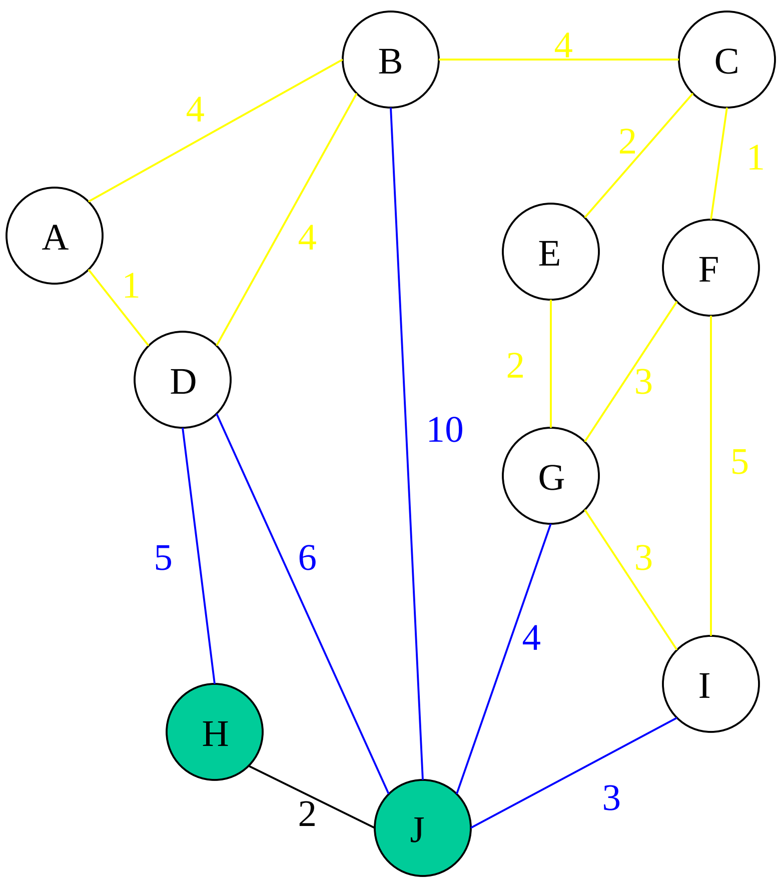
Old Graph



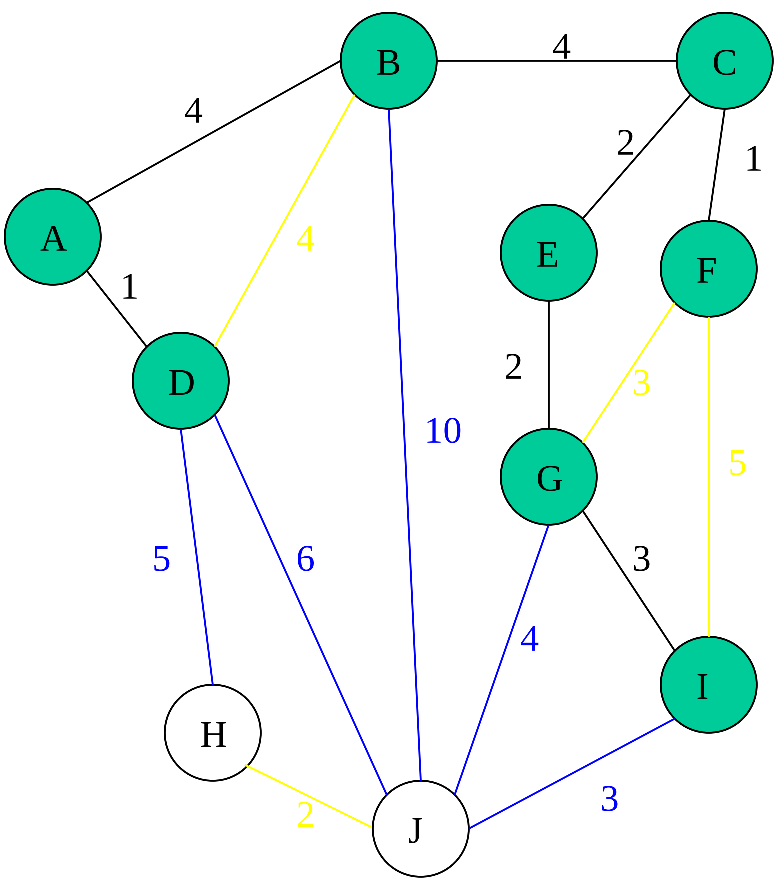
New Graph



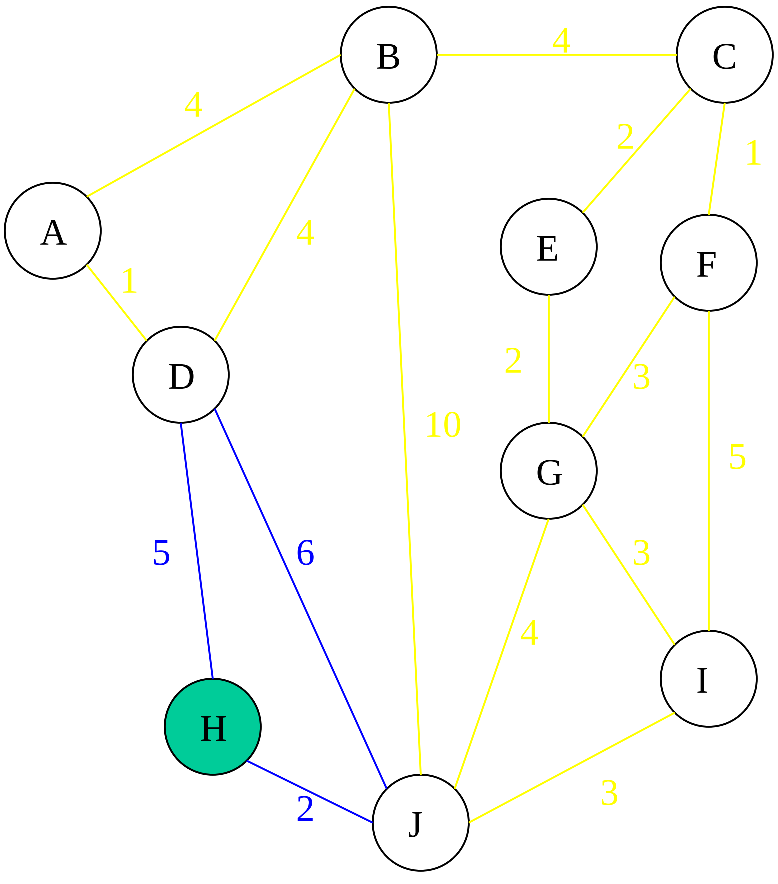
Old Graph



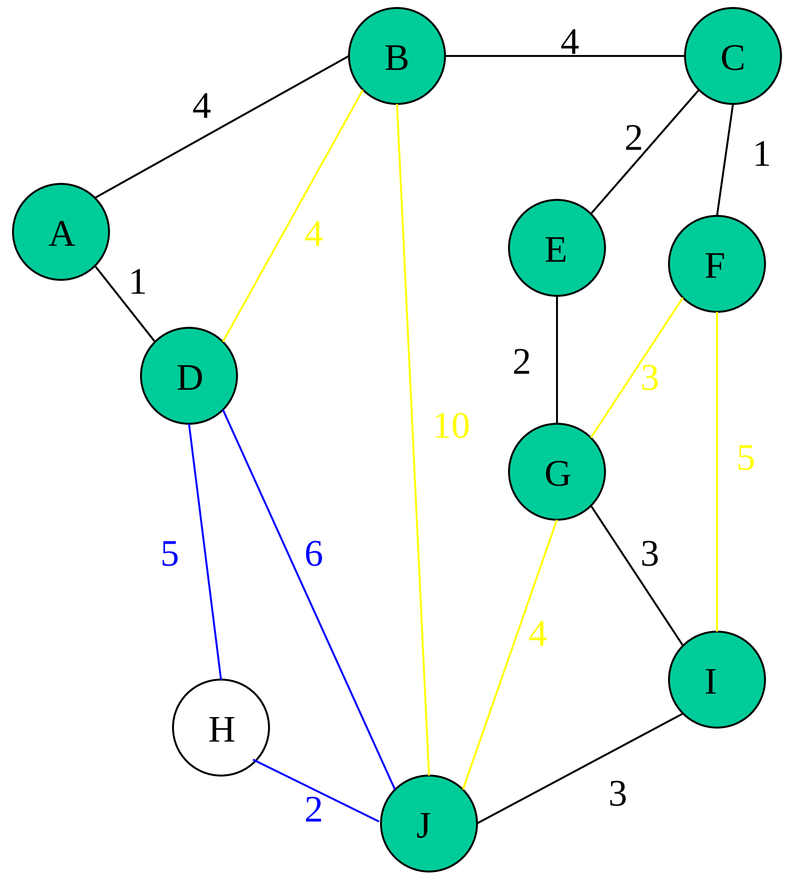
New Graph



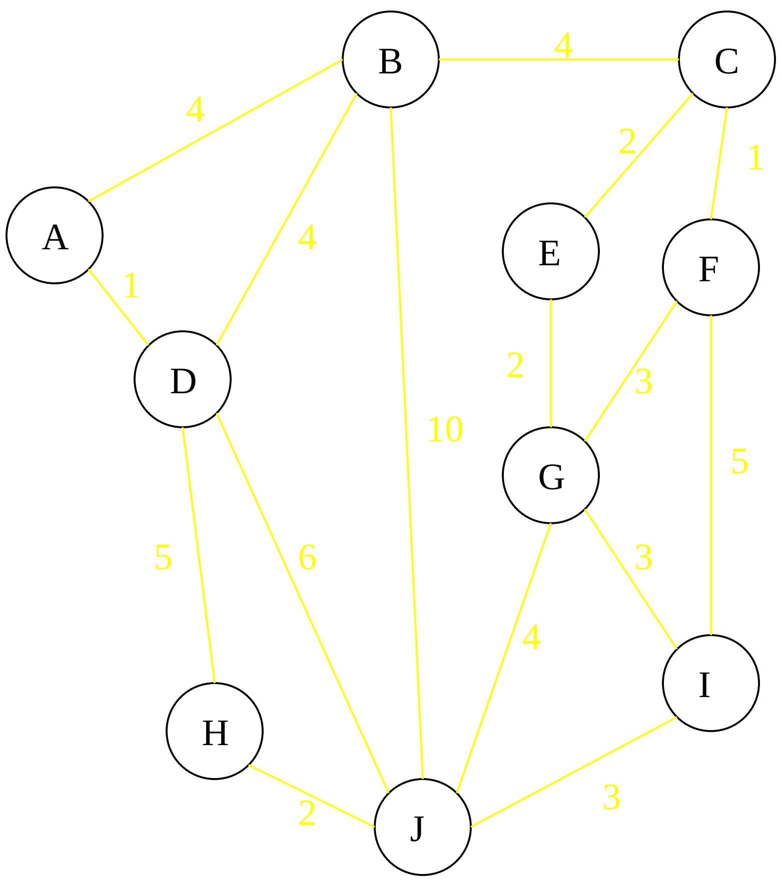
Old Graph



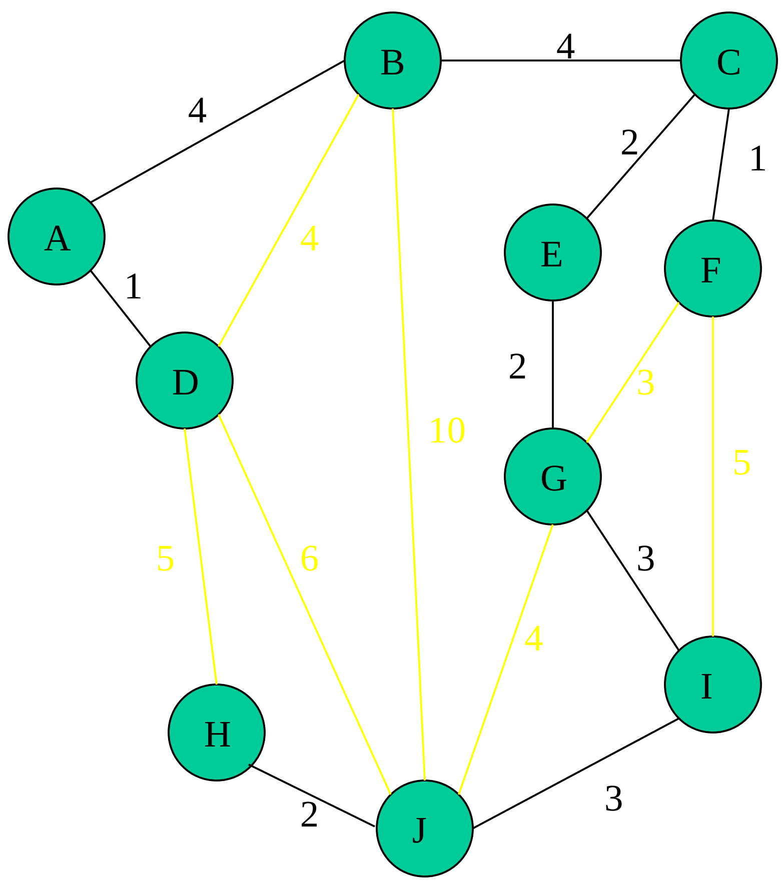
New Graph



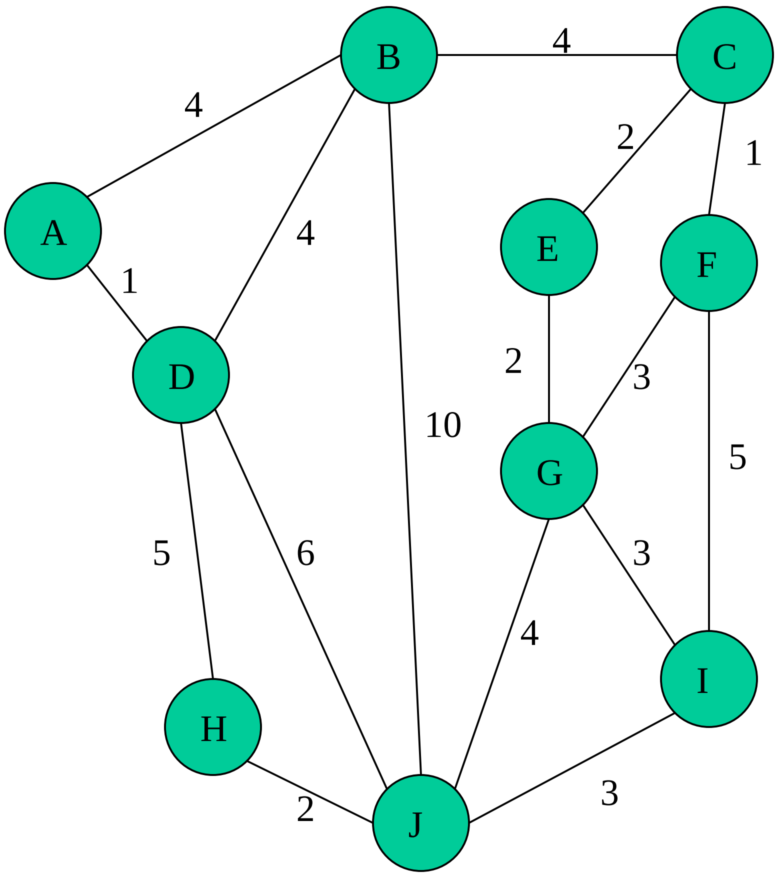
Old Graph



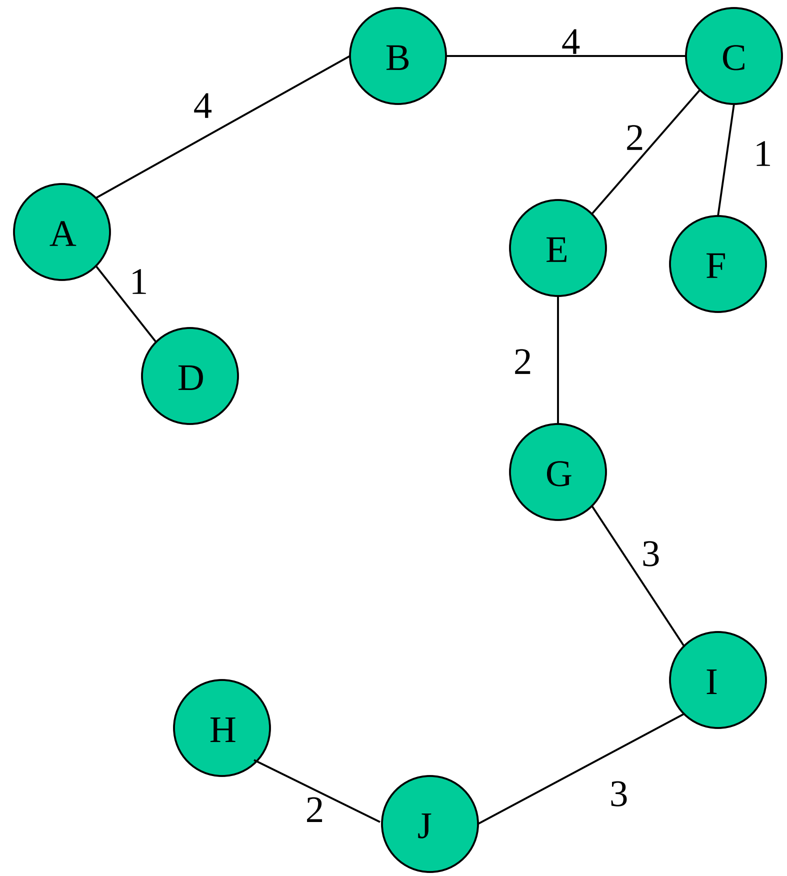
New Graph



Complete Graph



Minimum Spanning Tree



Summary of Prim's Algorithm

- Unlike Kruskal's, Prim's algorithm **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.

Time Complexity Review

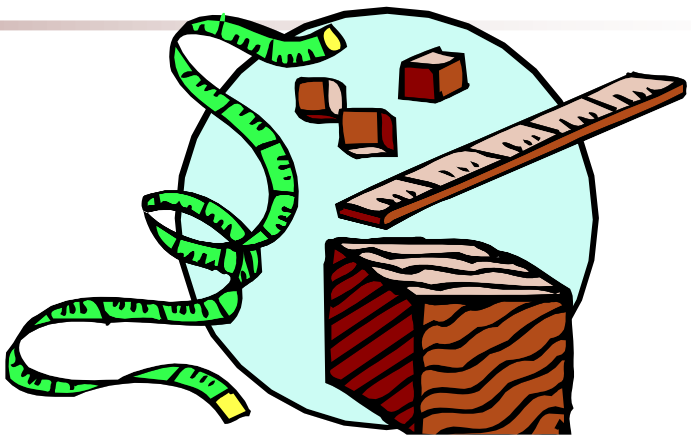
- Kruskal's algorithm: $O(e \log v)$
- Prim's algorithm: $O(e + v \log v)$

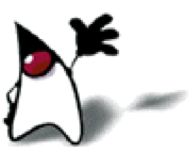
- Kruskal's algorithm is preferable on **sparse graphs**, i.e., where *e* is very small compared to the total number of possible edges.

- Prim's algorithm is easy to implemented, but the number of vertices needs to be kept to a minimum in addition to the number of edges.

Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Approximation Algorithms

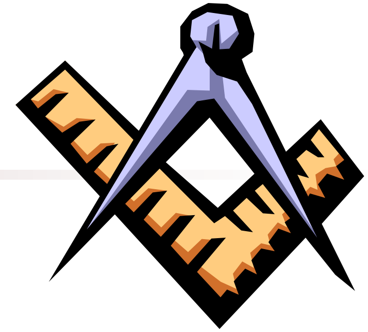




TSP: Traveling Salesperson Problem

Given a number of cities and the costs of traveling from any city to any other city, what is the cheapest round-trip route that visits each city exactly once and then returns to the starting city?

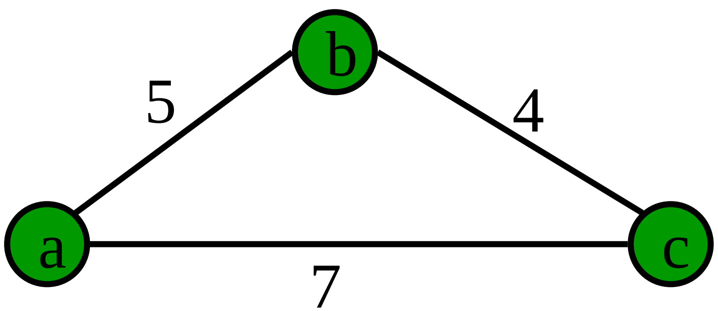
Approximation Ratios



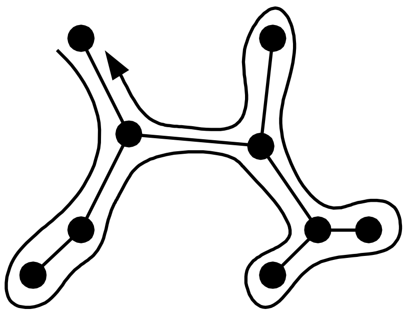
- Optimization Problems
  - We have some problem instance x that has many feasible "solutions".
  - We are trying to minimize (or maximize) some cost function c(S) for a "solution" S to x. For example,
    - Finding a minimum spanning tree of a graph
    - Finding a smallest vertex cover of a graph
    - Finding a smallest traveling salesperson tour in a graph
- An approximation produces a solution T
  - T is a **k-approximation** to the optimal solution OPT if $c(T)/c(OPT) \leq k$ (assuming a min. prob.; a maximization approximation would be the reverse)

Special Case of the Traveling Salesperson Problem

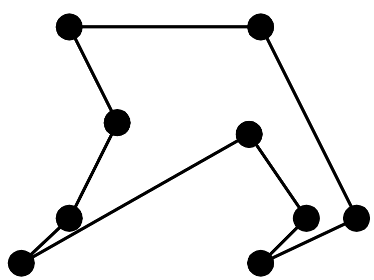
- OPT-TSP: Given a complete, weighted graph, find a cycle of minimum cost that visits each vertex.
  - OPT-TSP is NP-hard
  - Special case: edge weights satisfy the triangle inequality (which is common in many applications):
    - w(a,b) + w(b,c) $\geq$ w(a,c)



2-Approximation for TSP Special Case



Euler tour P of MST M



Output tour T

Algorithm *TSPApprox*(G)

Input weighted complete graph G, satisfying the triangle inequality

Output a TSP tour T for G

$M \leftarrow$ a minimum spanning tree for G

$P \leftarrow$ an Euler tour traversal of M, starting at some vertex s

$T \leftarrow$ empty list

for each vertex v in P (in traversal order)

  if this is v' s first appearance in P **then**

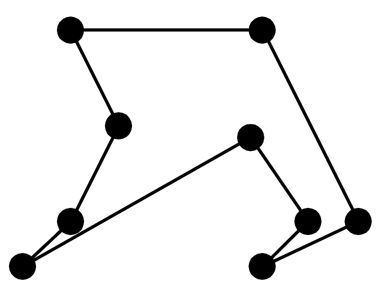
    T.*insertLast*(v)

T.*insertLast*(s)

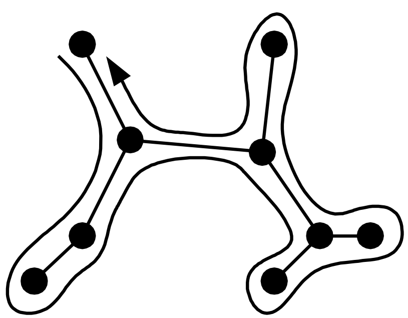
return T

2-Approximation for TSP Special Case - Proof

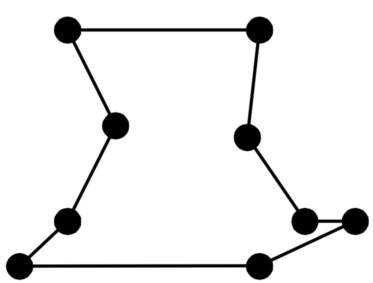
- The optimal tour is a spanning tour; hence $|M| \le |OPT|$.
- The Euler tour P visits each edge of M twice; hence $|P| = 2|M|$
- Each time we shortcut a vertex in the Euler Tour we will not increase the total length, by the triangle inequality ($w(a,b) + w(b,c) \ge w(a,c)$); hence, $|T| \le |P|$.
- Therefore, $|T| \le |P| = 2|M| \le 2|OPT|$



Output tour *T*
(at most the cost of *P*)



Euler tour *P* of MST *M*
(twice the cost of *M*)



Optimal tour *OPT*
(at least the cost of MST *M*)