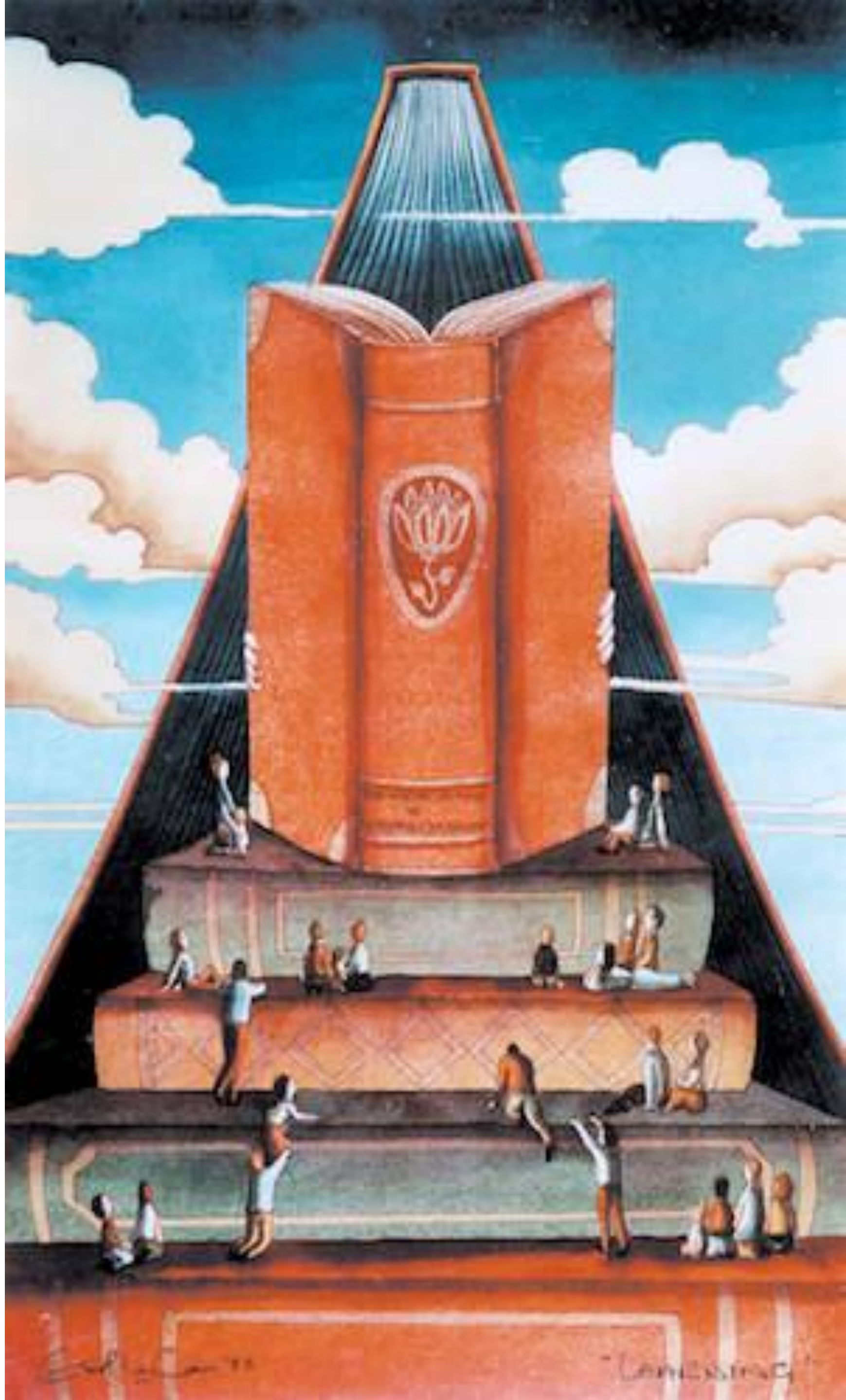
GREEDY ALGORITHM



Outlines

- Introduction
- Knapsack Problem
- Job Sequencing with Deadlines
- Minimum-Cost Spanning Trees
  - Prim’s Algorithm
  - Kruskal’s Algorithm
- Single-Source Shortest Paths
  - Dijkstra’s Algorithm

Introduction

- A greedy algorithm always makes the choice that looks best at the moment.
- That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- The greedy approach does not always lead to an optimal solution.
- The problems that have a greedy solution are said to posses the greedy-choice property.
- The greedy approach is also used in the context of hard (difficult to solve) problems in order to generate an approximate solution.

Knapsack Problem

- We are given n objects and a knapsack or bag
- Object i has a weight w_i and the knapsack has a capacity W
- If a fraction x_i, $1 \le x_i \le 1$, of object i is placed into the knapsack, the a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Thus, the problem is as follows:

$$\left.\begin{aligned} &\max \textstyle\sum_{i=1}^{n} p_i x_i \\ &\text{s.t. } \textstyle\sum_{i=1}^{n} w_i x_i \le W \\ &\quad 0 \le x_i \le 1, 1 \le i \le n \end{aligned}\right\}$$

A feasible solution is any set $(x_1,\ldots,x_n)$ satisfying the constraints. An optimal solution is a feasible solution for which the objective is maximized

Knapsack Problem (Cont..)

- Example: Consider the instance of a knapsack problem: n=3, W=20, $(p_1,p_2,p_3)=(25,24,15)$ and $(w_1,w_2,w_3)=(18,15,10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

- Of these four feasible solutions, solution 4 yields the maximum profit.

Knapsack Problem (Cont..)

```
Greedy_Knapsack(W, n)
    for i = 1 to n do
        x[i] = 0
    end for
    M = W;
    for i = 1 to n do
        if w[i] > M then
            break;
        end if
        x[i] = 1.0; M = M – w[i]
    end for
    if i ≤ n then x[i] = M/w[i] end if
End  Greedy_Knapsack
```

//p[1:n] and w[1:n] contain the profit and weights respectively of the n objects s.t. p[i]/w[i]≥ p[i+1]/w[i+1]. W is the knapsack size and x[1:n] is the solution vector

Job Sequencing with Deadlines

- We are given a set J of n jobs
- Job i is associated with an integer deadline $d_i \geq 0$ and a profit $p_i > 0$
- P_i is earned iff the job is completed by its deadline
- To complete a job, one has to process the job on a machine for one unit of time and one machine is available for processing jobs
- The feasible solution for this problem is $J' \subseteq J$ that can be completed within its deadline
- The value is $\sum_{i \in J'} p_i$
- The optimal solution is a feasible solution with maximum value

Job Sequencing with Deadlines (Cont..)

- Example: let n=4, $(p_1,p_2,p_3,p_4)=(100,10,15,27)$ and $(d_1,d_2,d_3,d_4)=(2,1,2,1)$. The feasible solution s and their values:

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

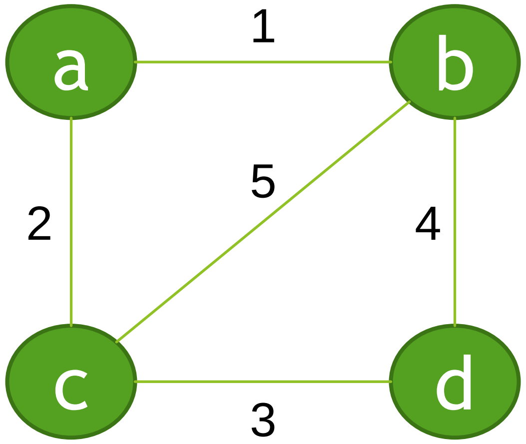
Solution 3 is optimal

Job Sequencing with Deadlines (Cont..)

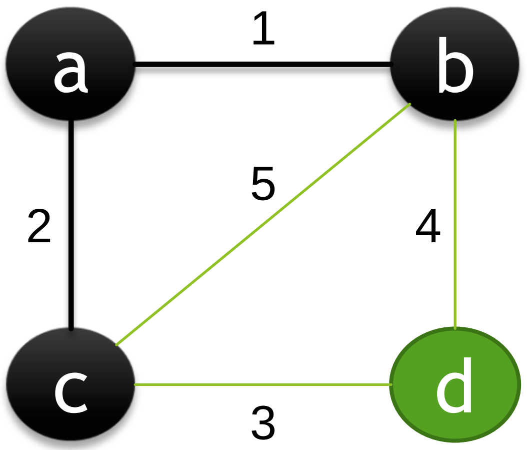
```
Algorithm JS(d, j, n)
// d[i] ≥ 1, 1 ≤ i ≤ n are the deadlines, n ≥ 1. The jobs
// are ordered such that p[1] ≥ p[2] ≥ ··· ≥ p[n]. J[i]
// is the ith job in the optimal solution, 1 ≤ i ≤ k.
// Also, at termination d[J[i]] ≤ d[J[i + 1]], 1 ≤ i < k.
{
    d[0] := J[0] := 0; // Initialize.
    J[1] := 1; // Include job 1.
    k := 1;
    for i := 2 to n do
    {
        // Consider jobs in nonincreasing order of p[i]. Find
        // position for i and check feasibility of insertion.
        r := k;
        while ((d[J[r]] > d[i]) and (d[J[r]] ≠ r)) do r := r − 1;
        if ((d[J[r]] ≤ d[i]) and (d[i] > r)) then
        {
            // Insert i into J[ ].
            for q := k to (r + 1) step −1 do J[q + 1] := J[q];
            J[r + 1] := i; k := k + 1;
        }
    }
    return k;
}
```

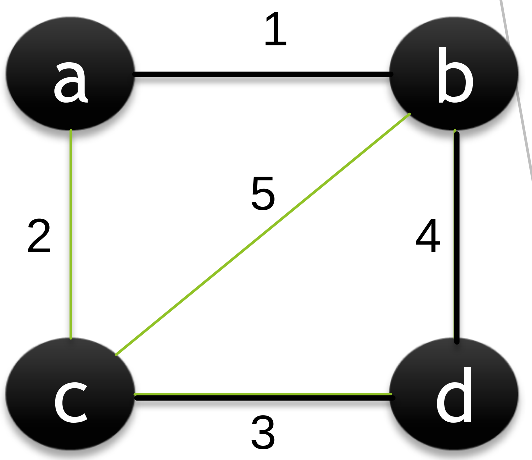
Minimum cost spanning tree (MCST)

- Tree: No cycles; equivalently, for each pair of nodes u and v, there is only one path from u to v
- Spanning: Contains every node in the graph
- Minimum cost: Smallest possible total weight of any spanning tree



MCST (Cont..)



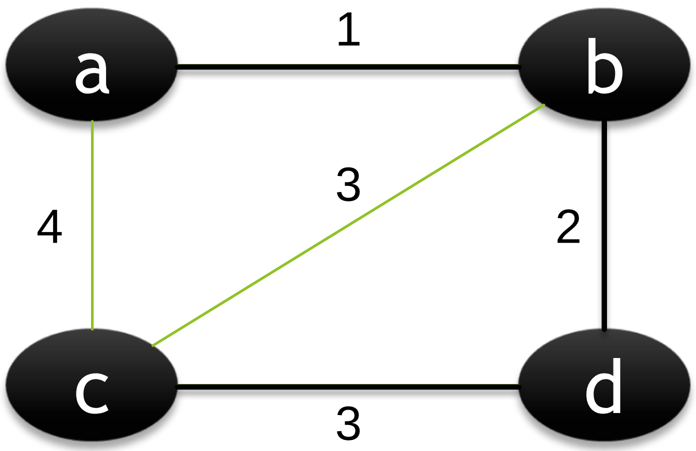


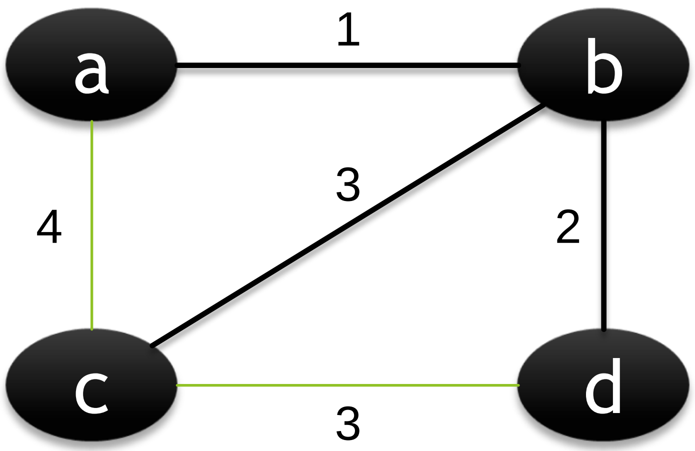
- Black edges and nodes are in T. Is T a MCST?
- Not spanning; d is not in T.

- Black edges and nodes are in T. Is T a MCST?
- Not minimum cost; can swap edges 4 and 2

MCST (Cont..)

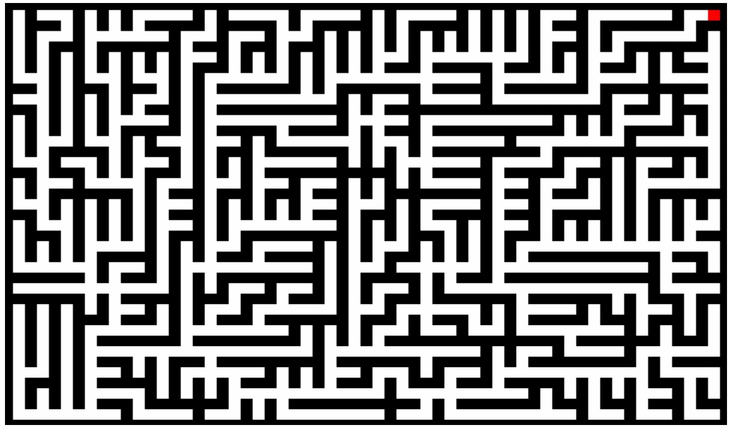
- Which edges form a MCST?





Application of MCST

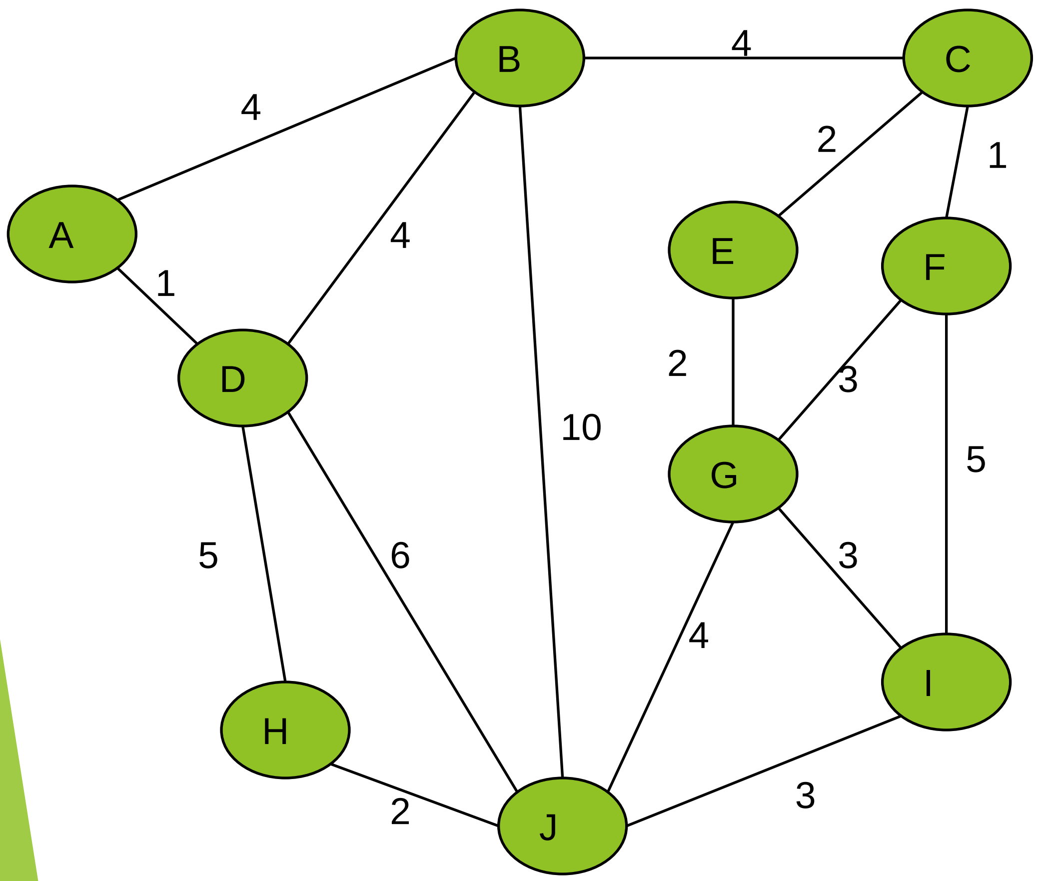
- Electronic circuit designs
- Planning how to lay network cable to connect several locations to the internet
- Planning how to efficiently bounce data from router to router to reach its internet destination
- Creating a 2D maze (to print on cereal boxes, etc.)



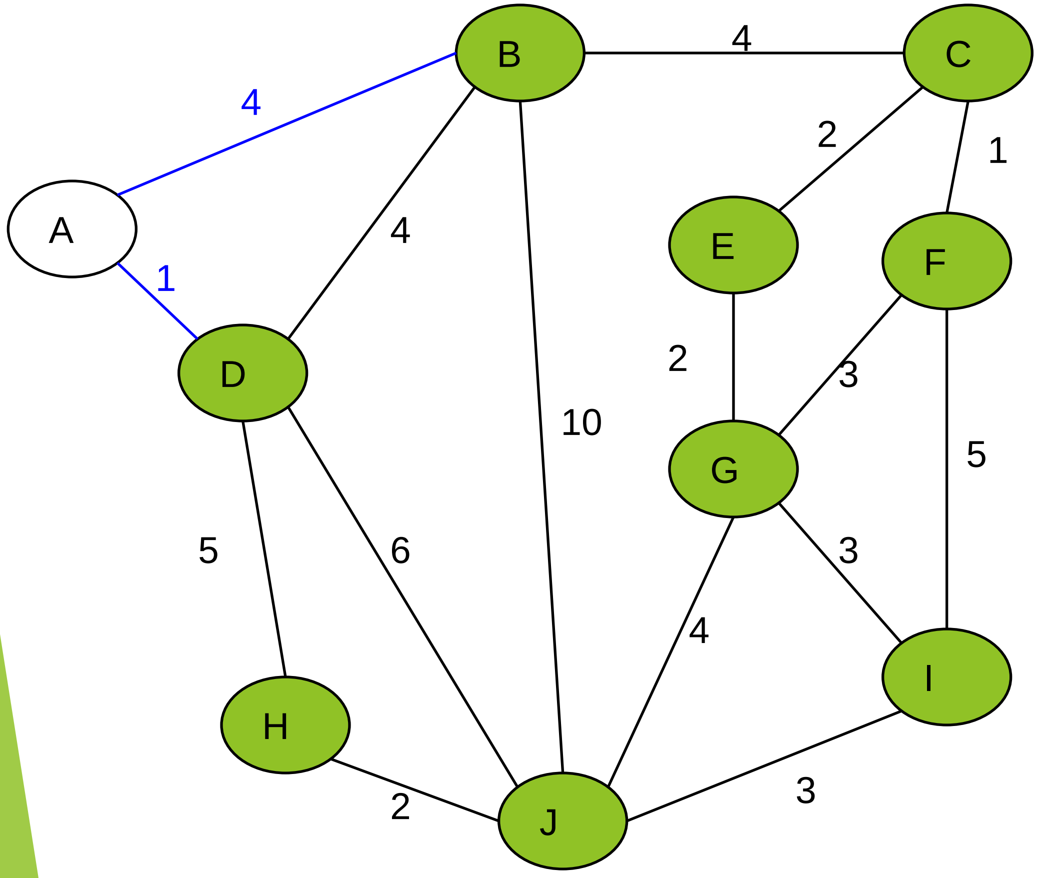
Prim's Algorithm

- Prim’s algorithm takes a graph *G=(V, E)* and builds an MCST *T*
- Major steps of the algorithm:
  - Pick an arbitrary node r from V
  - Add r to T
  - While T contains < |V| nodes
    - Find a minimum weight edge (u, v) where $u \in T$ and $v \notin T$
    - Add node v to T

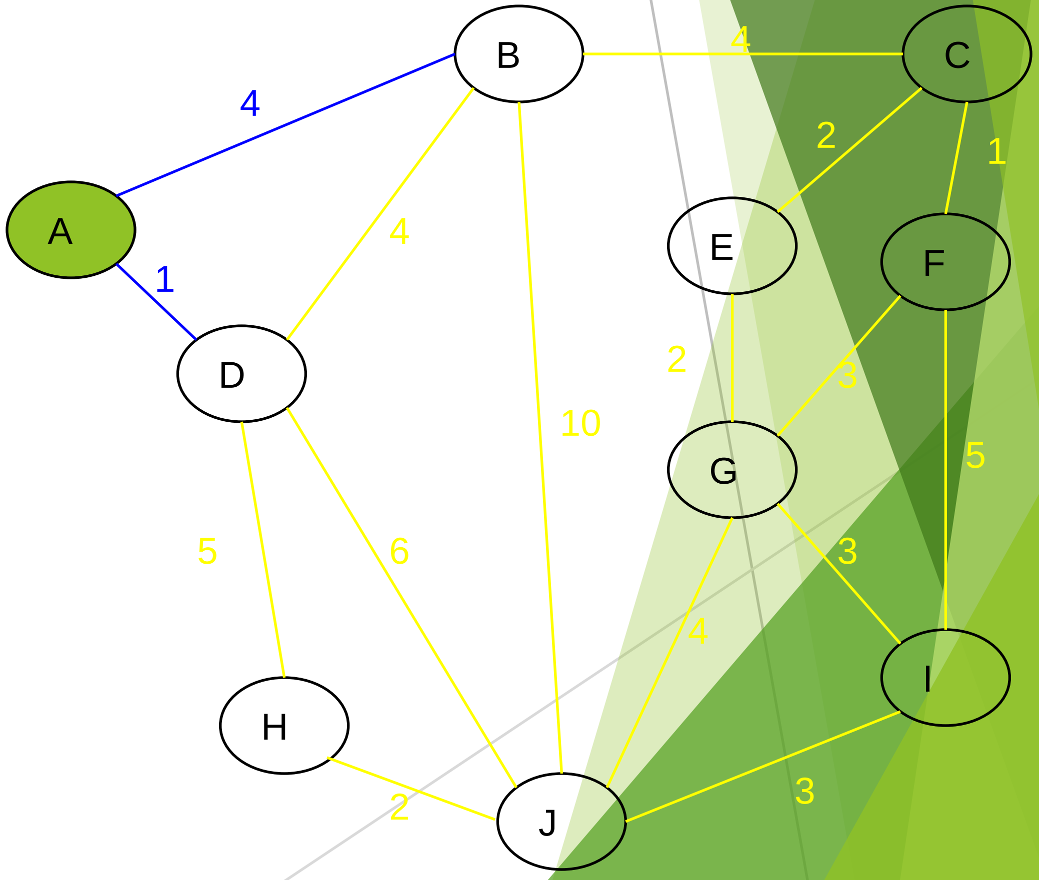
Complete Graph



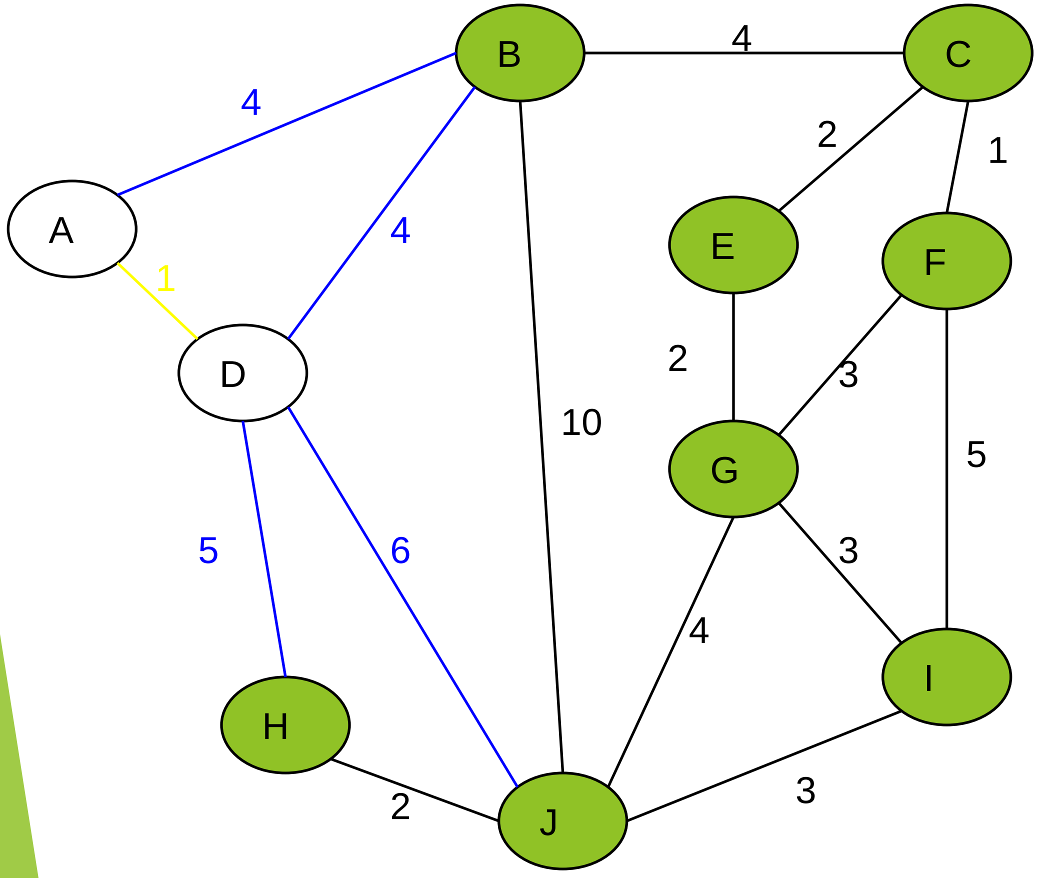
Old Graph



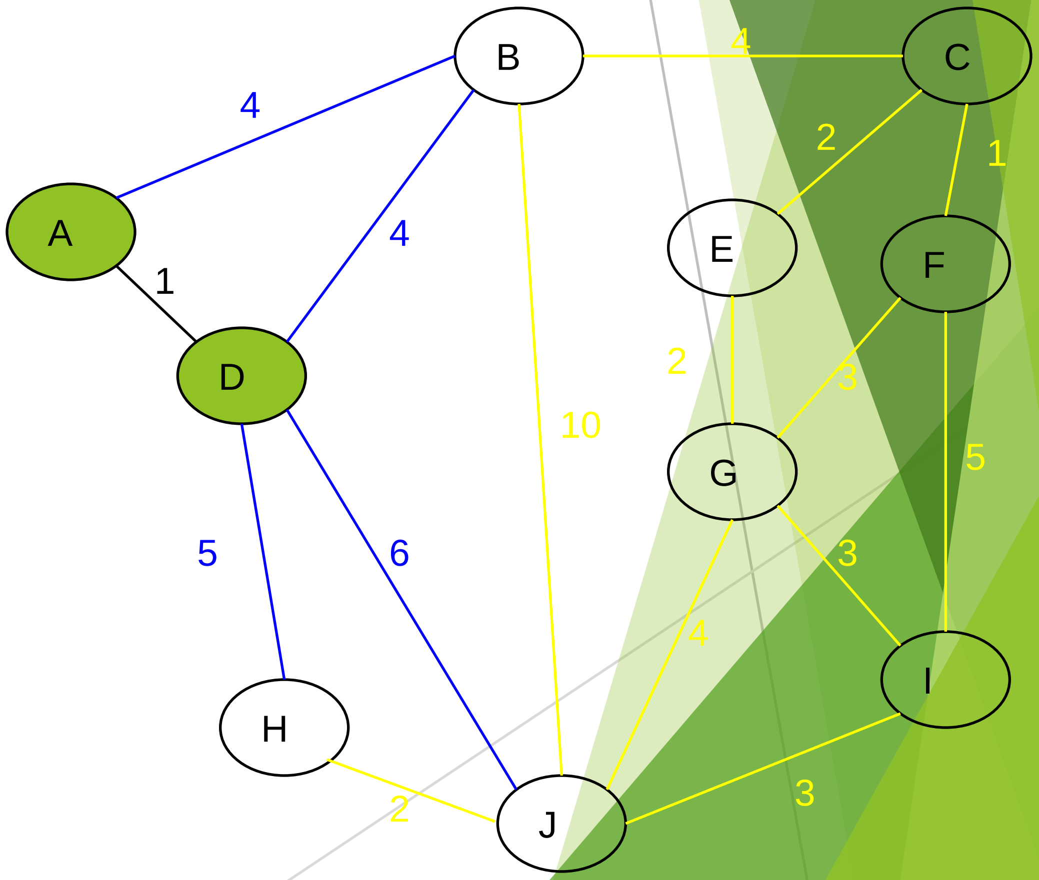
New Graph



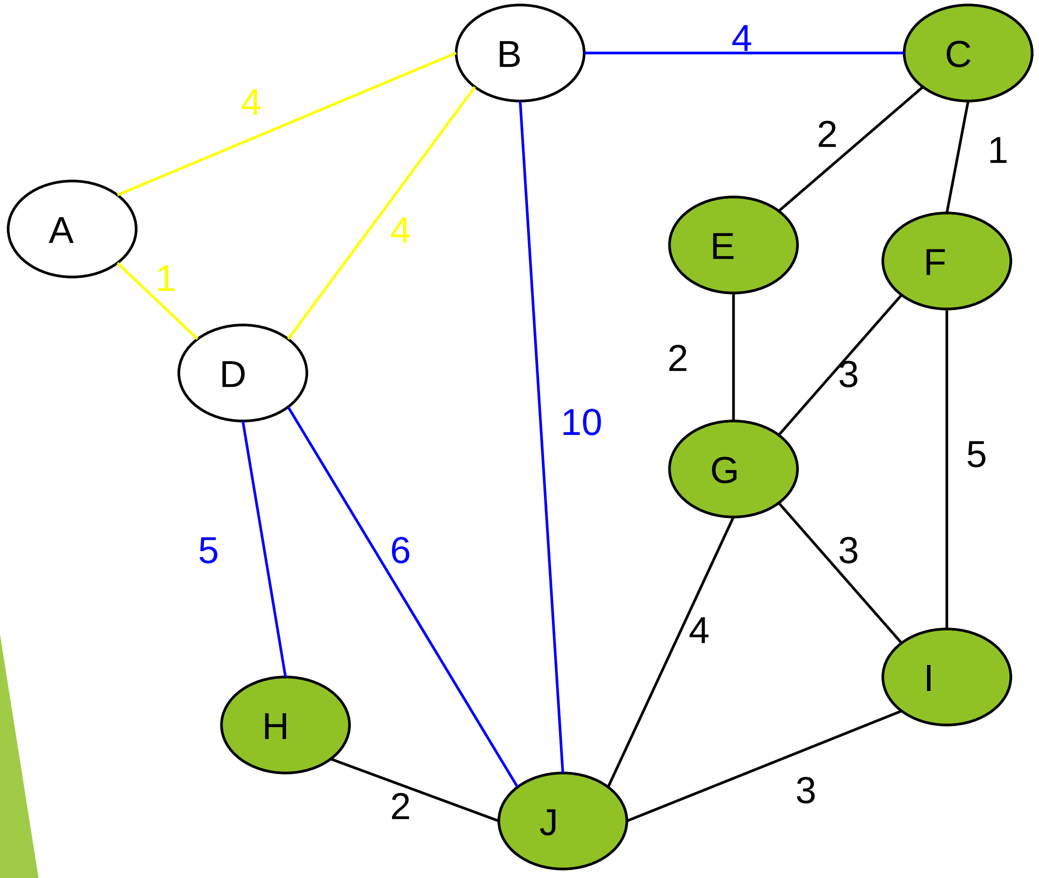
Old Graph



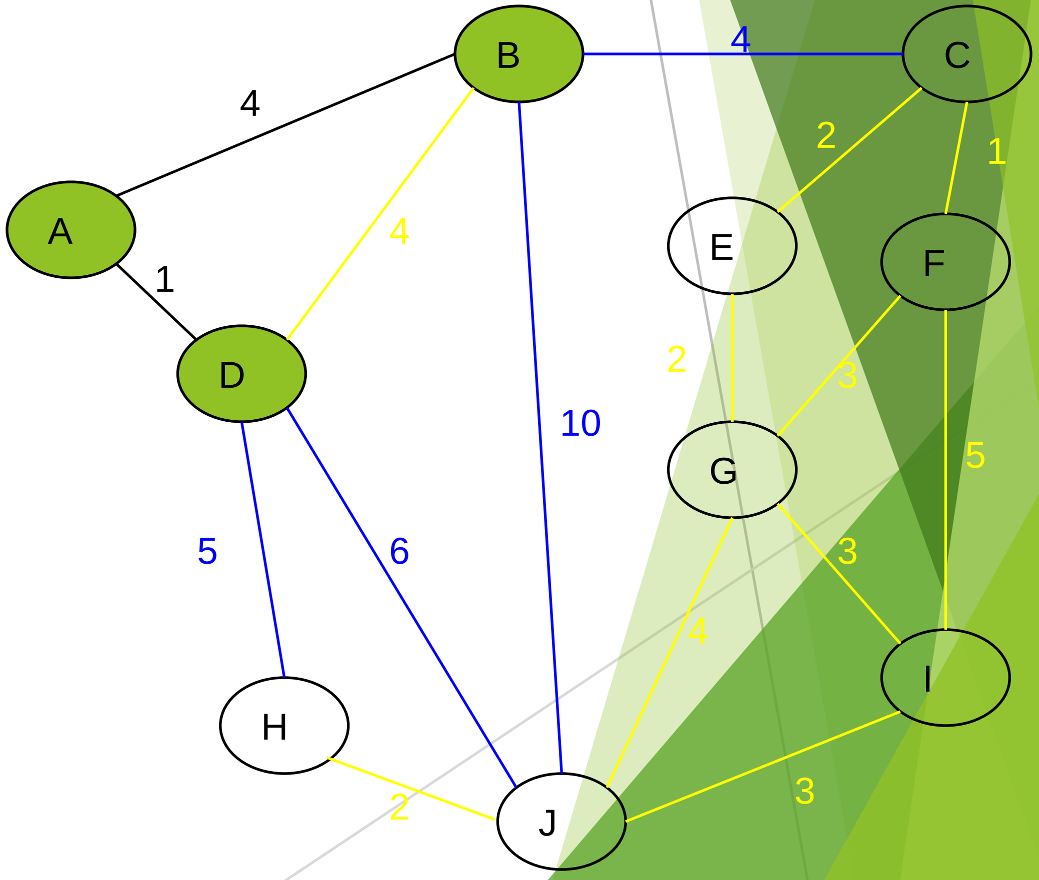
New Graph



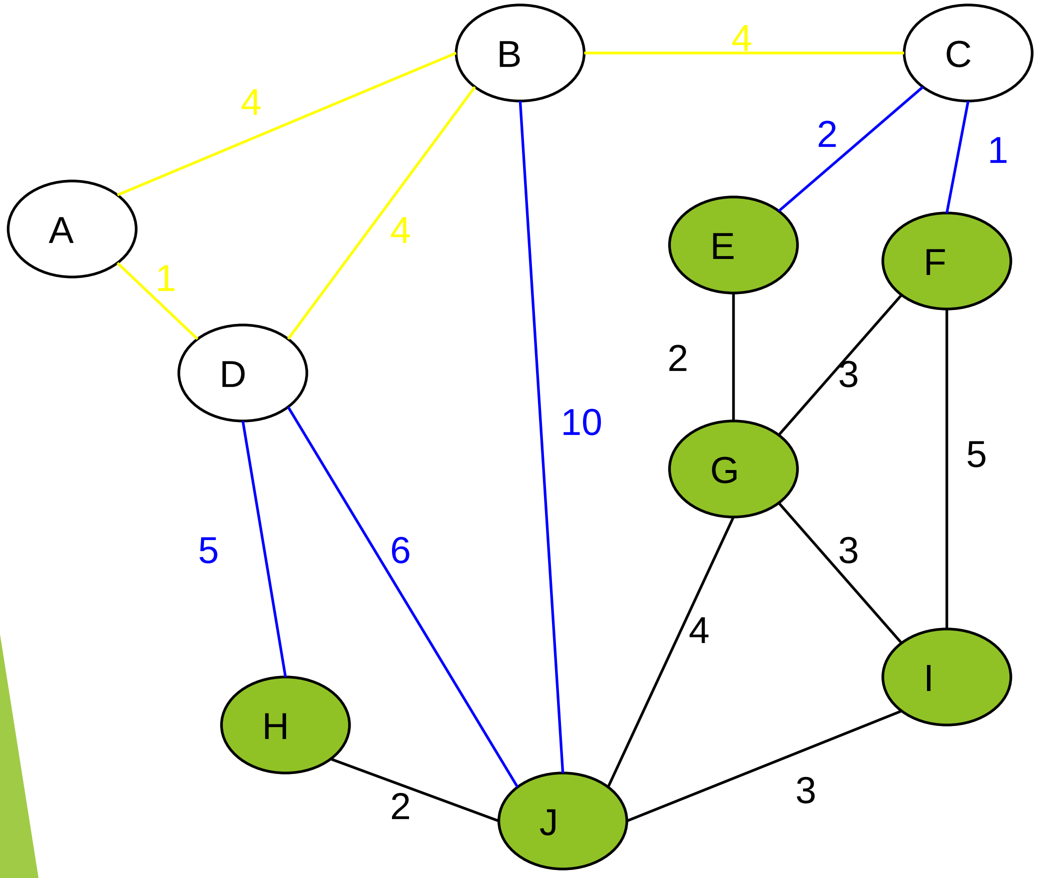
Old Graph



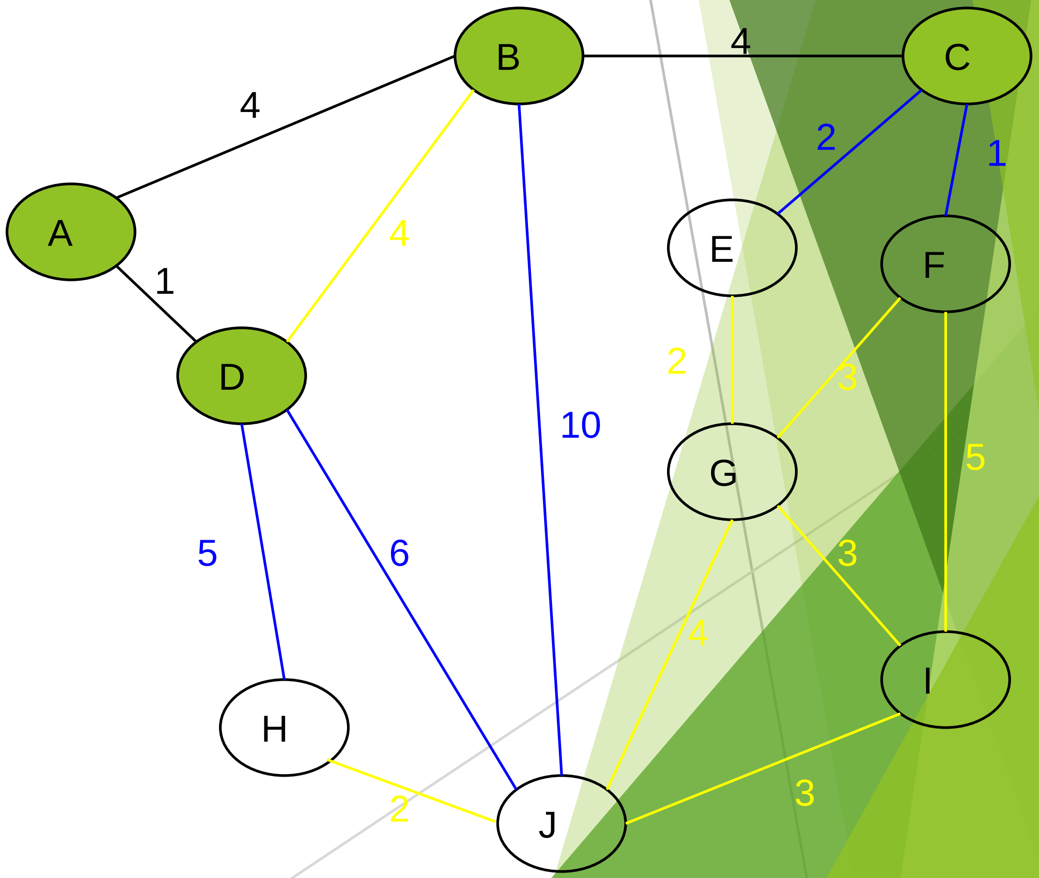
New Graph



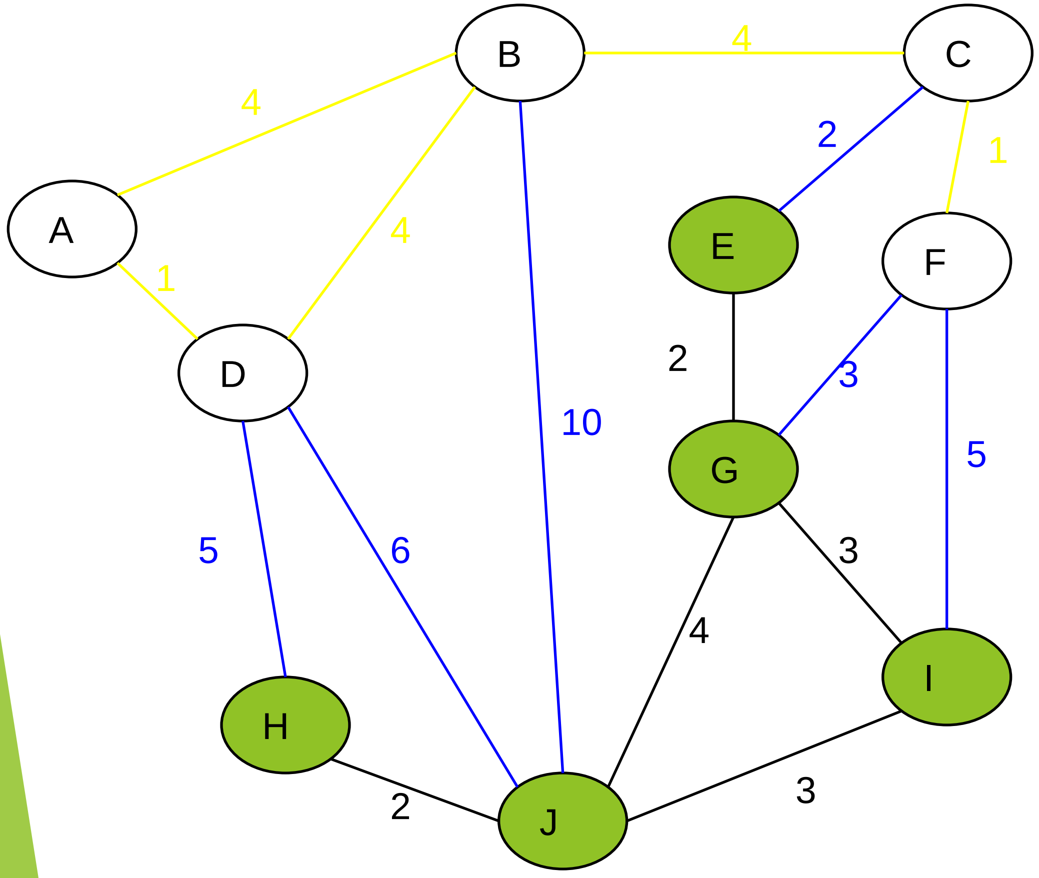
Old Graph



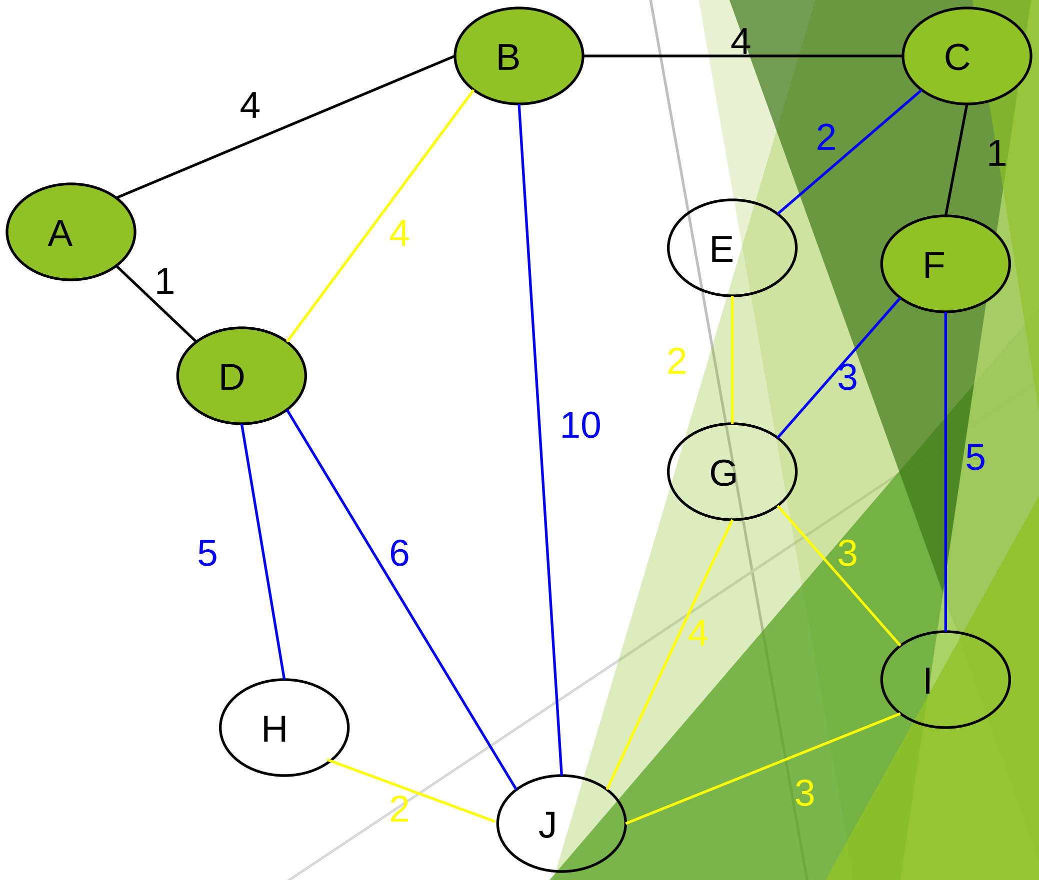
New Graph



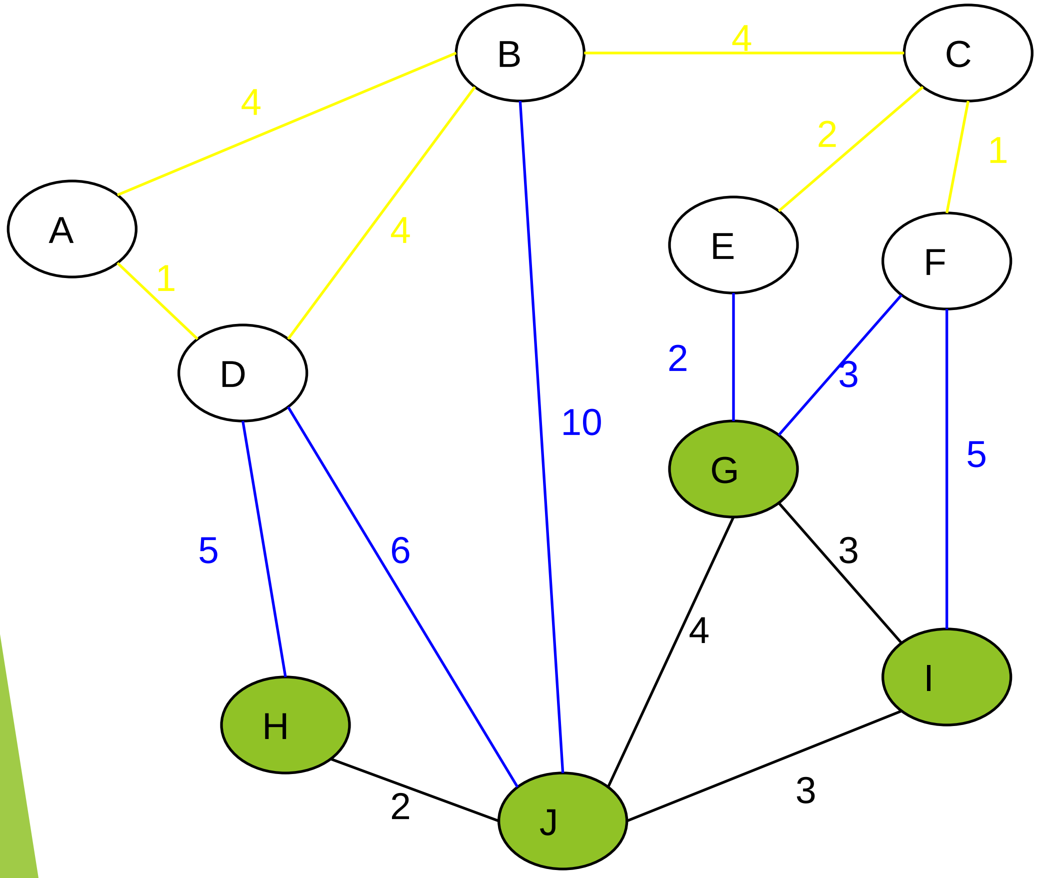
Old Graph



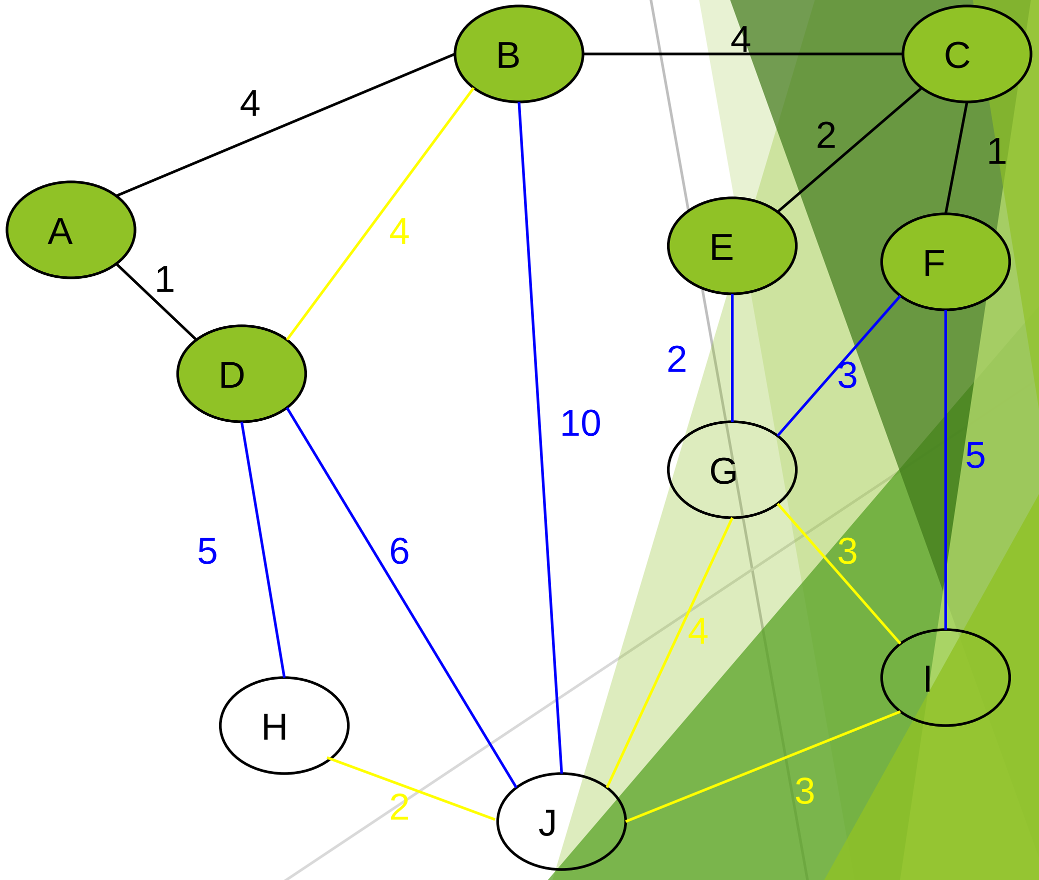
New Graph



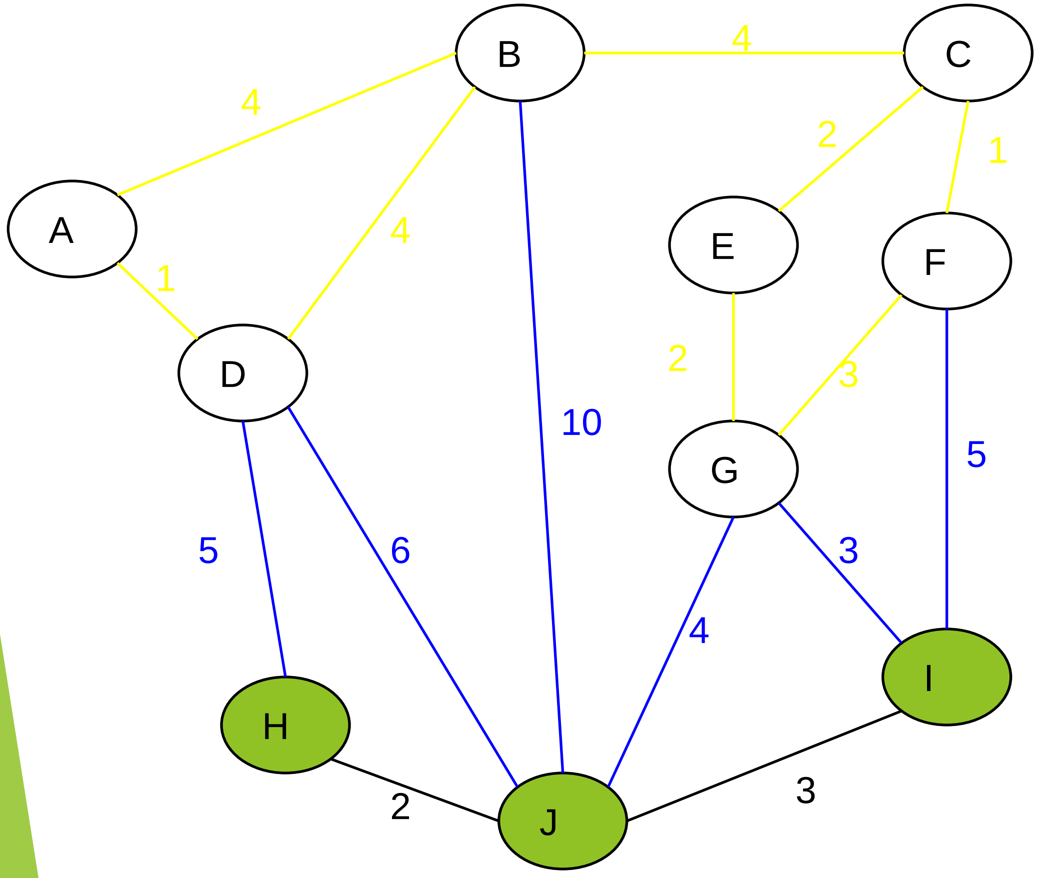
Old Graph



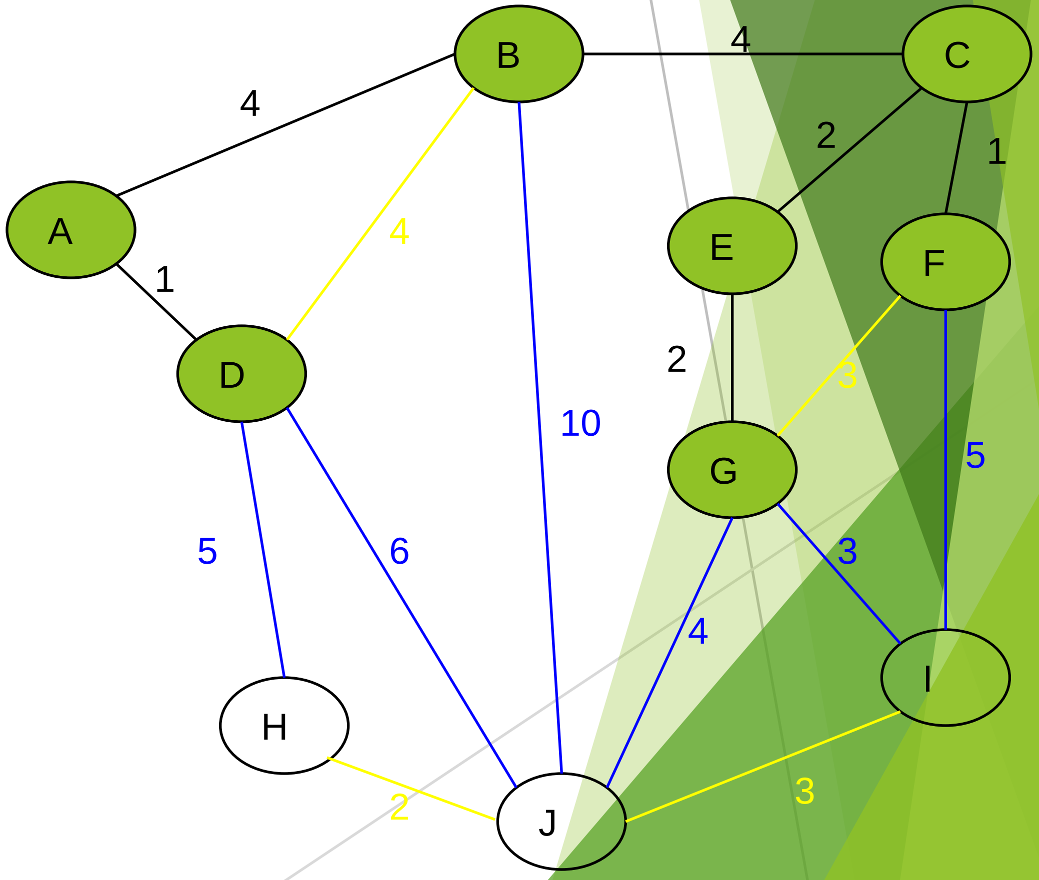
New Graph

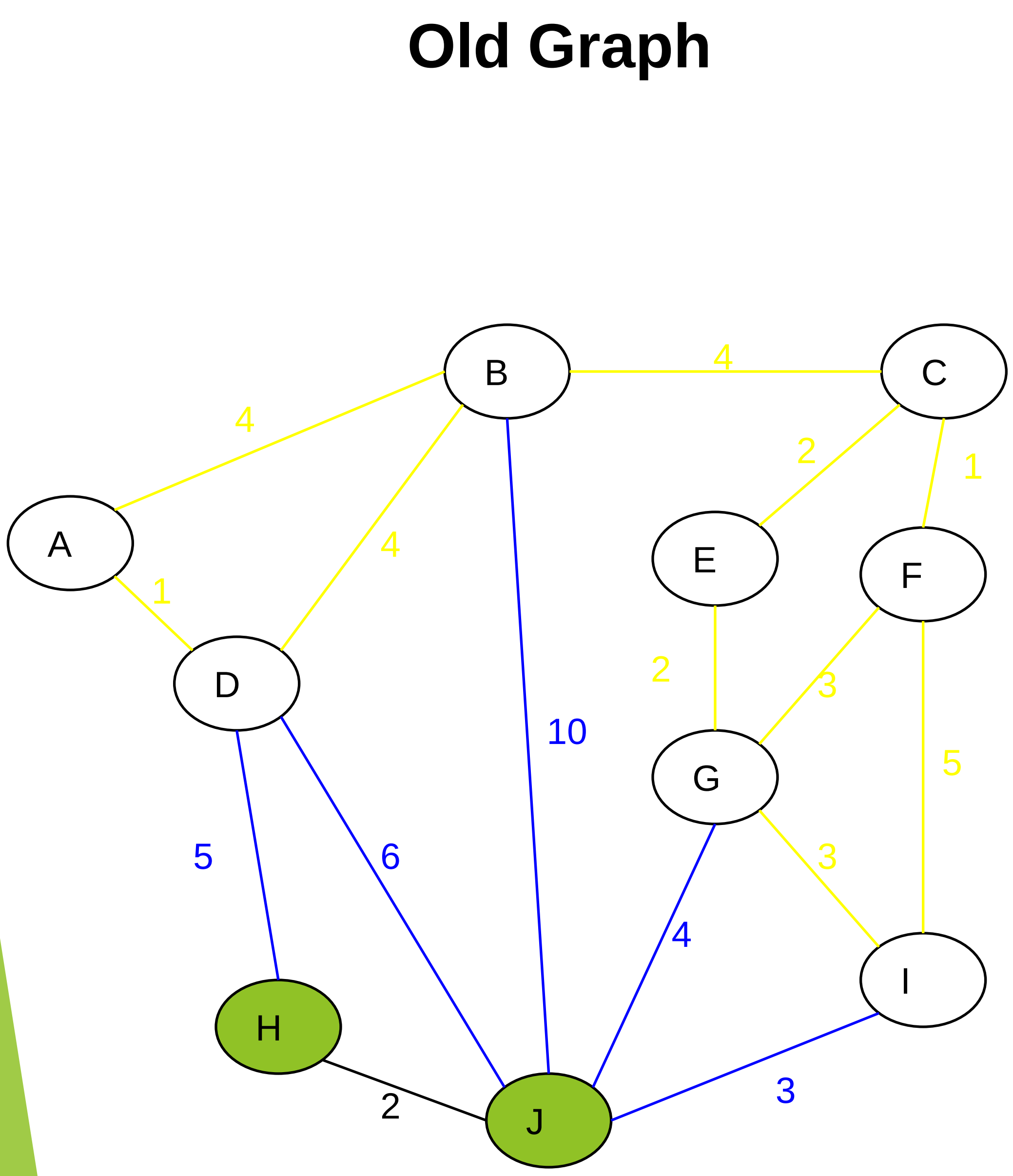


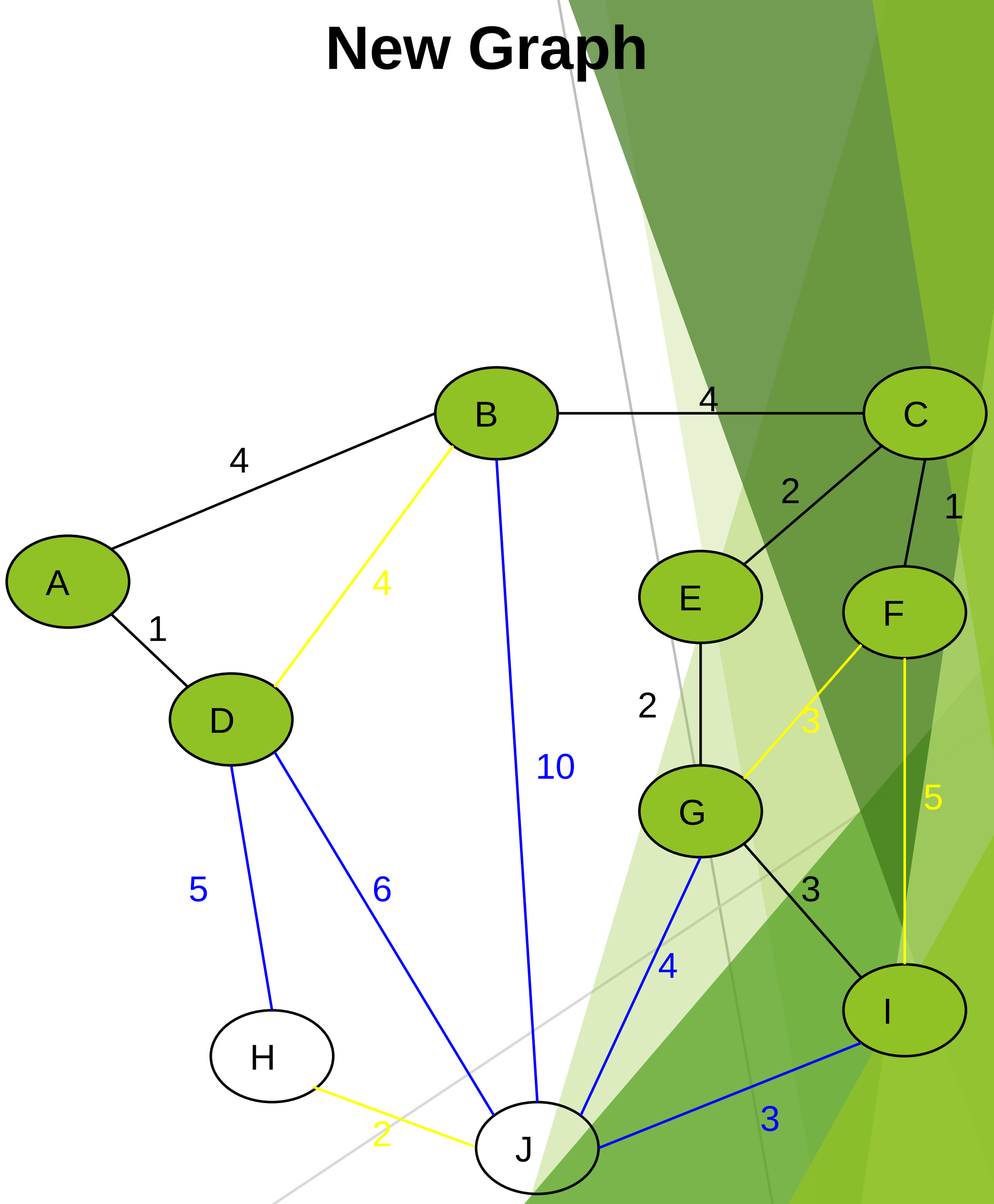
Old Graph



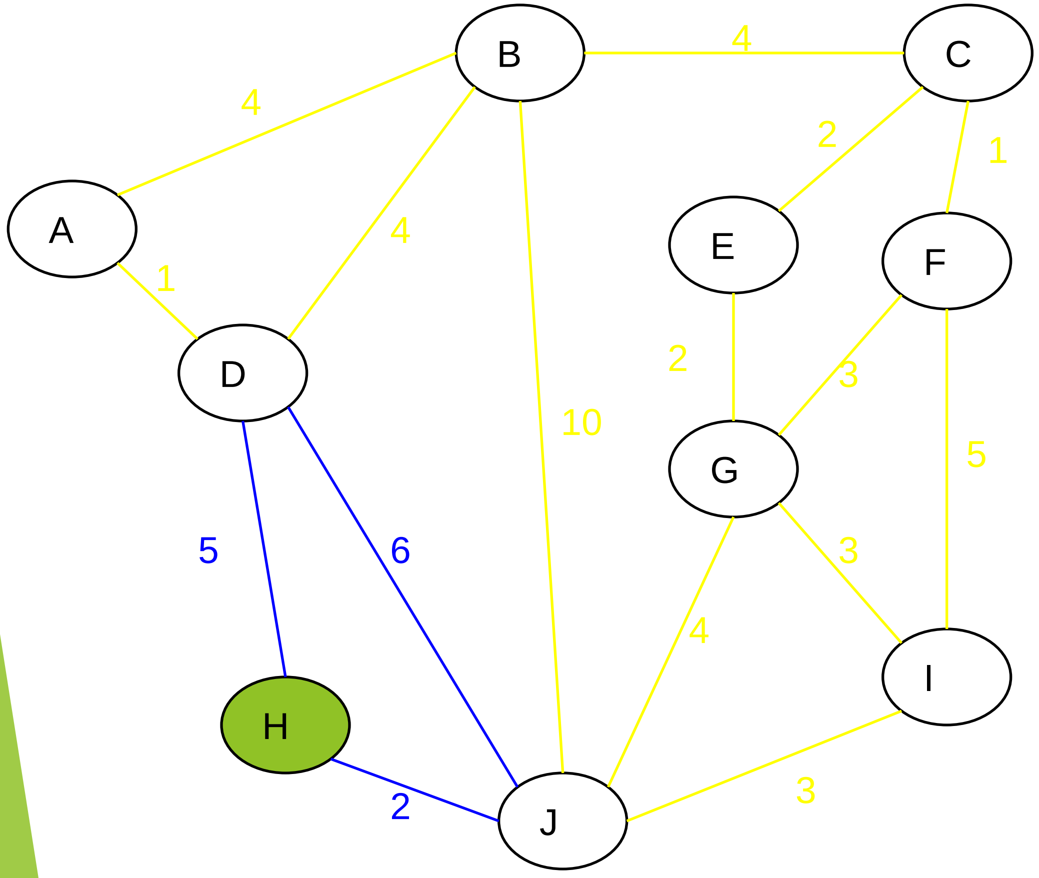
New Graph



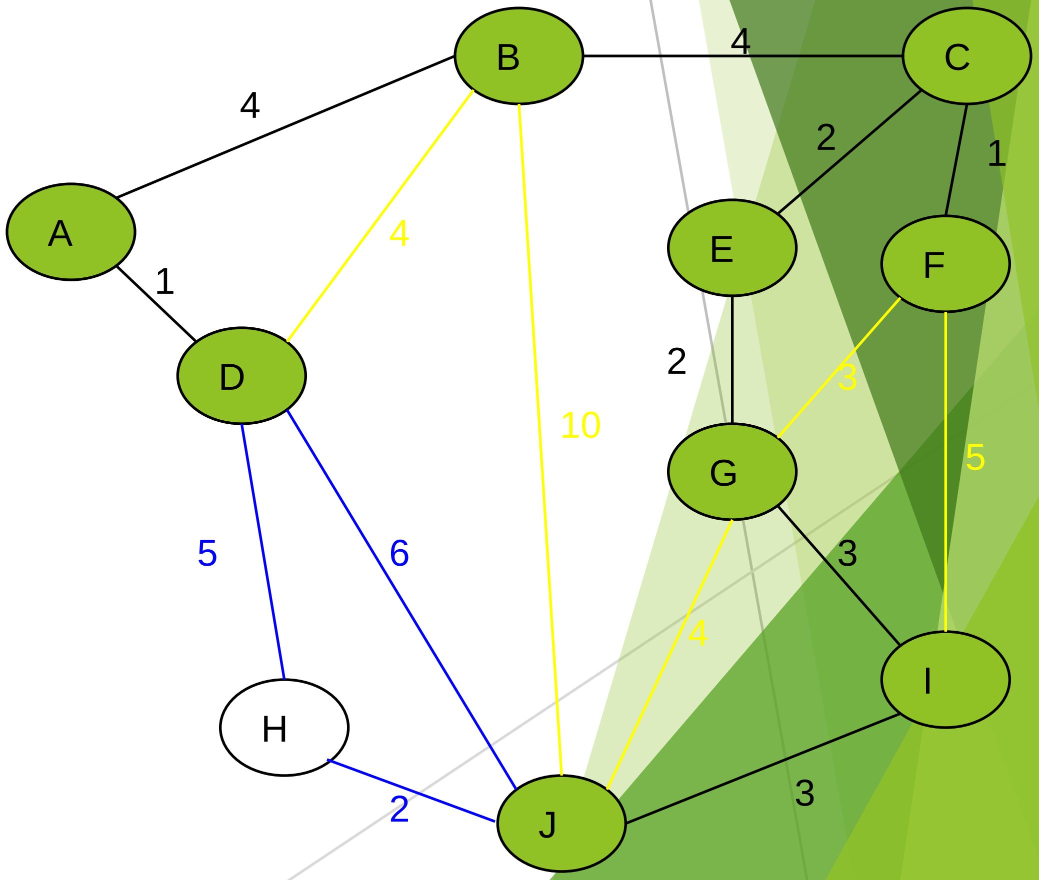

Old Graph
A
B
C
D
E
F
G
H
I
J
4
4
4
1
2
1
2
3
3
5
5
6
10
4
2
3


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

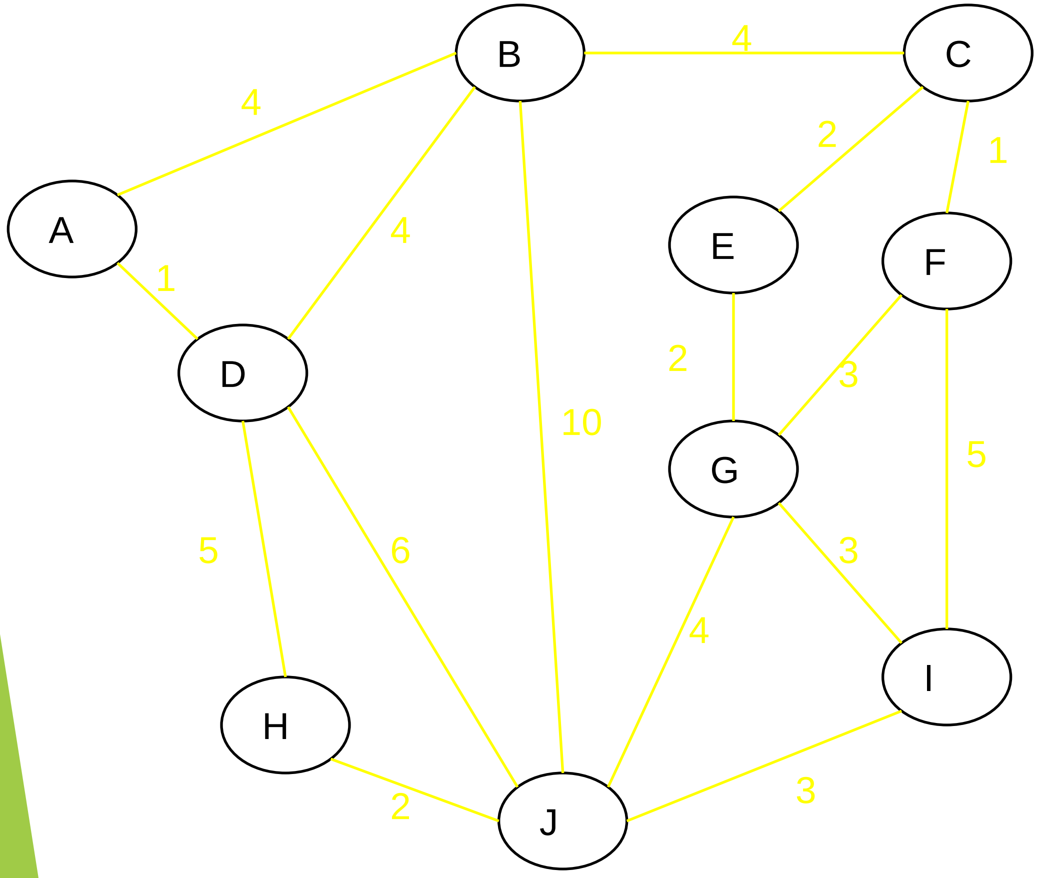
Old Graph



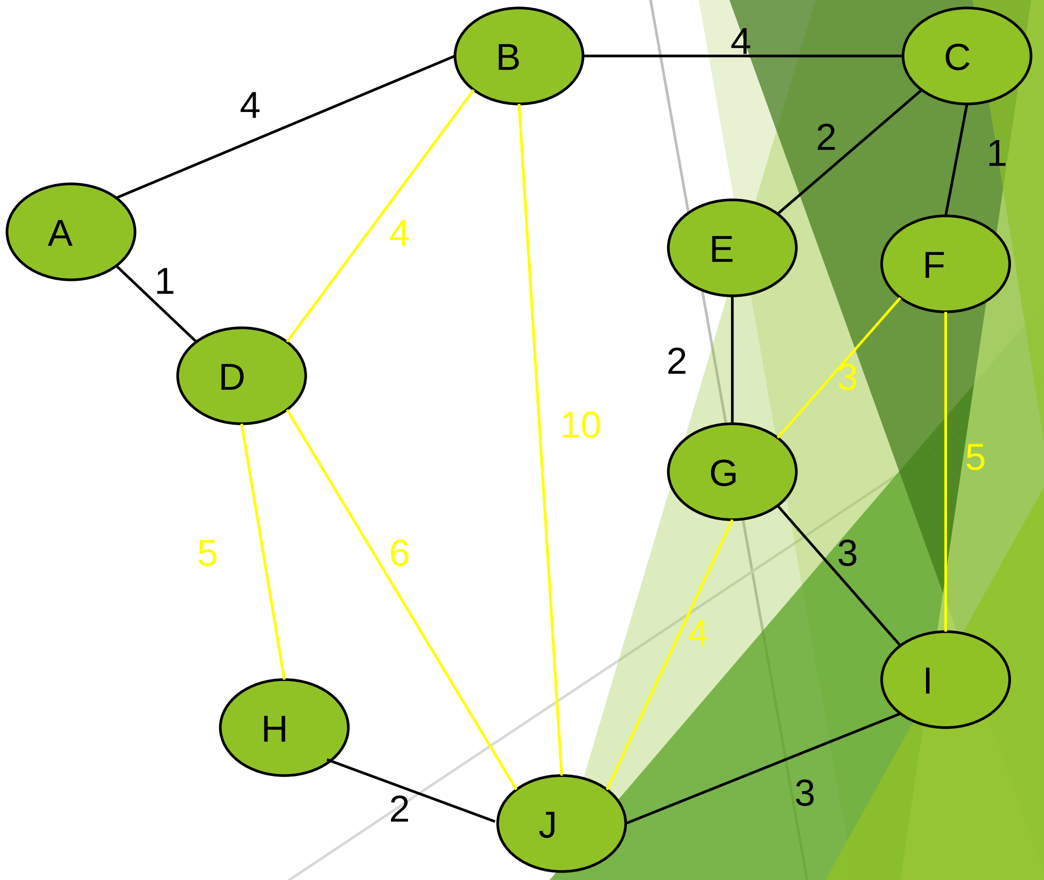
New Graph



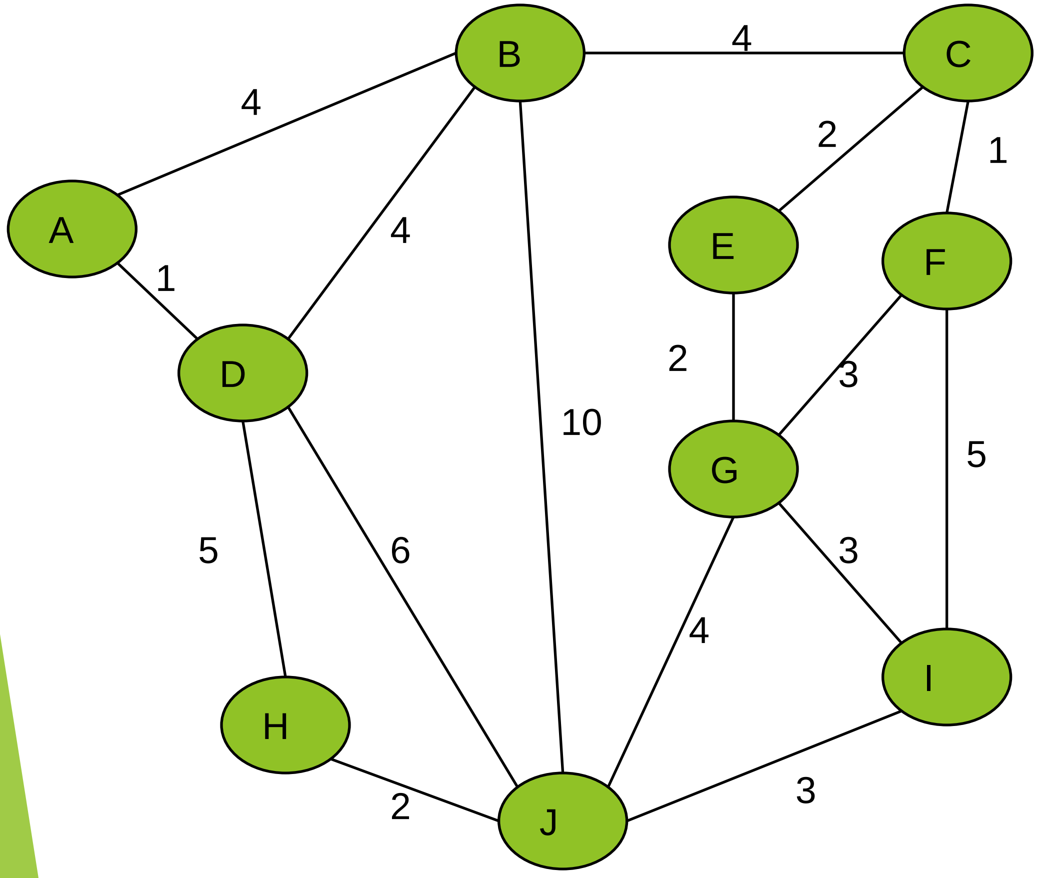
Old Graph



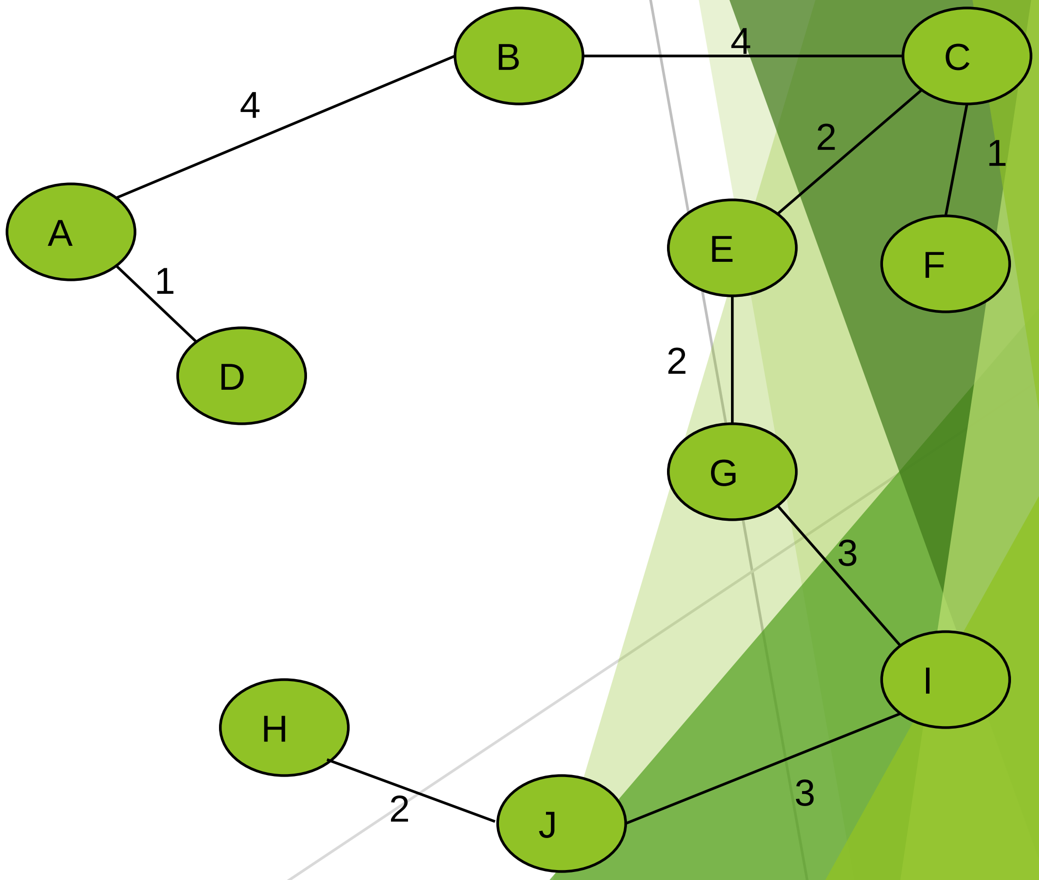
New Graph



Complete Graph



Minimum Spanning Tree



```
Algorithm Prim(E, cost, n, t)
// E is the set of edges in G. cost[1 : n, 1 : n] is the cost
// adjacency matrix of an n vertex graph such that cost[i, j] is
// either a positive real number or ∞ if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t[1 : n − 1, 1 : 2]. (t[i, 1], t[i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
    Let (k, l) be an edge of minimum cost in E;
    mincost := cost[k, l];
    t[1, 1] := k; t[1, 2] := l;
    for i := 1 to n do  // Initialize near.
        if (cost[i, l] < cost[i, k]) then near[i] := l;
        else near[i] := k;
    near[k] := near[l] := 0;
    for i := 2 to n − 1 do
    { // Find n − 2 additional edges for t.
        Let j be an index such that near[j] ≠ 0 and
        cost[j, near[j]] is minimum;
        t[i, 1] := j; t[i, 2] := near[j];
        mincost := mincost + cost[j, near[j]];
        near[j] := 0;
        for k := 1 to n do // Update near[ ].
            if ((near[k] ≠ 0) and (cost[k, near[k]] > cost[k, j]))
                then near[k] := j;
    }
    return mincost;
}
```

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

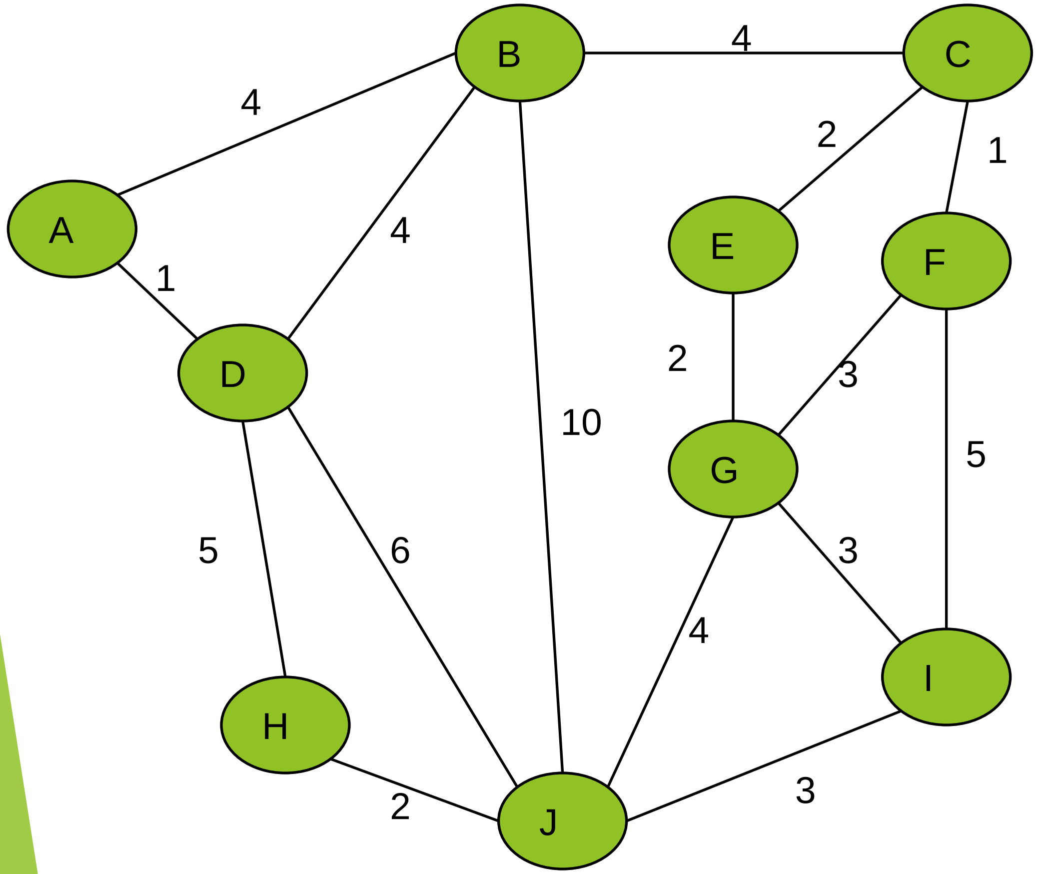
Kruskal's Algorithm

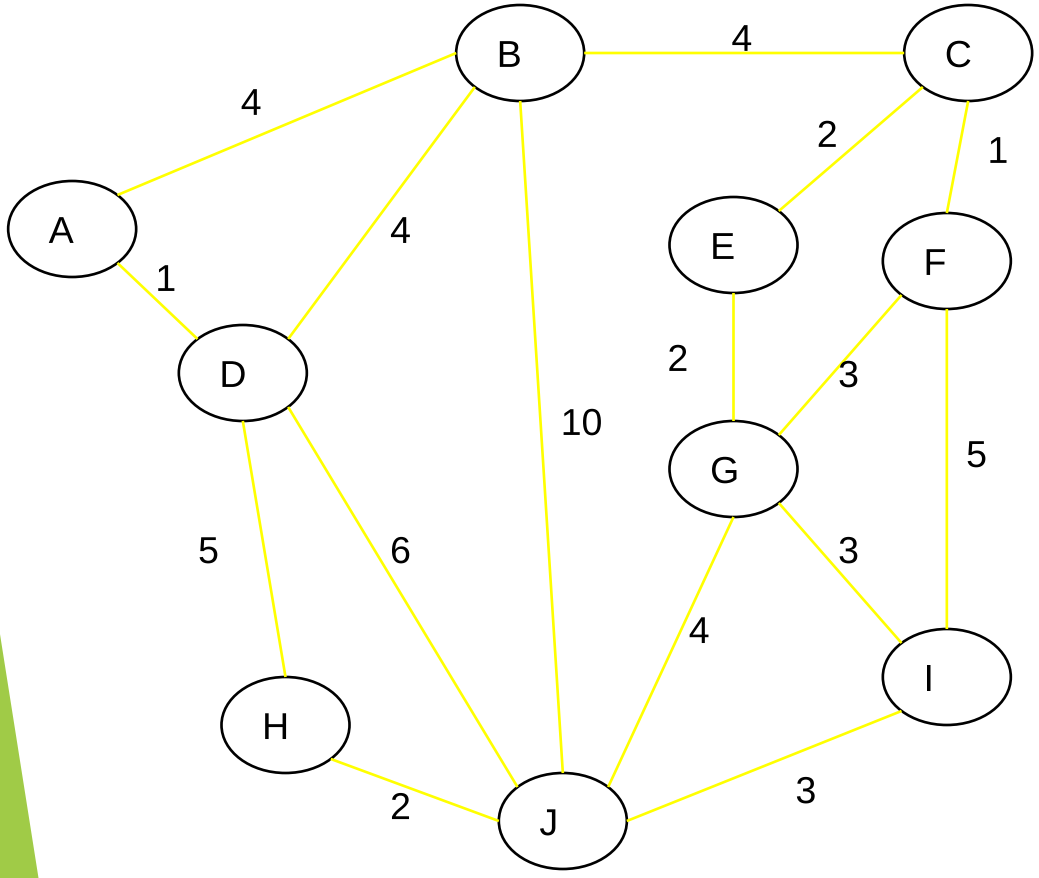
- This algorithm creates a forest of trees.
- Initially the forest consists of n single node trees (and no edges).
- At each step, we add one edge (the cheapest one) so that it joins two trees together.
- If it were to form a cycle, it would simply link two nodes that were already part of a single connected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

Kruskal's Algorithm (Steps)

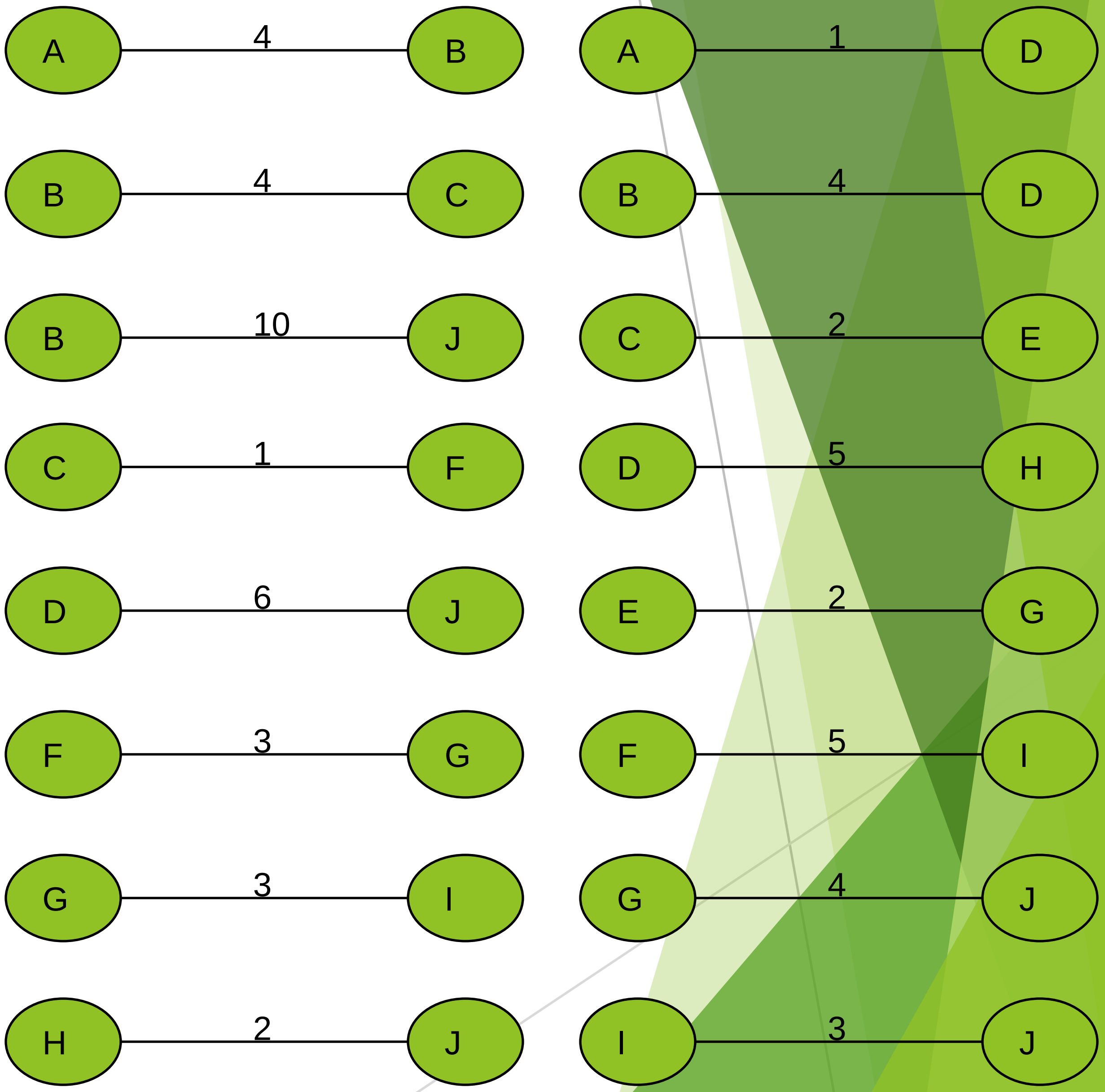
- The forest is constructed - with each node in a separate tree.
- The edges are placed in a priority queue.
- Until we've added n-1 edges,
  - Extract the cheapest edge from the queue,
  - If it forms a cycle, reject it,
  - Else add it to the forest. Adding it to the forest will join two trees together.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





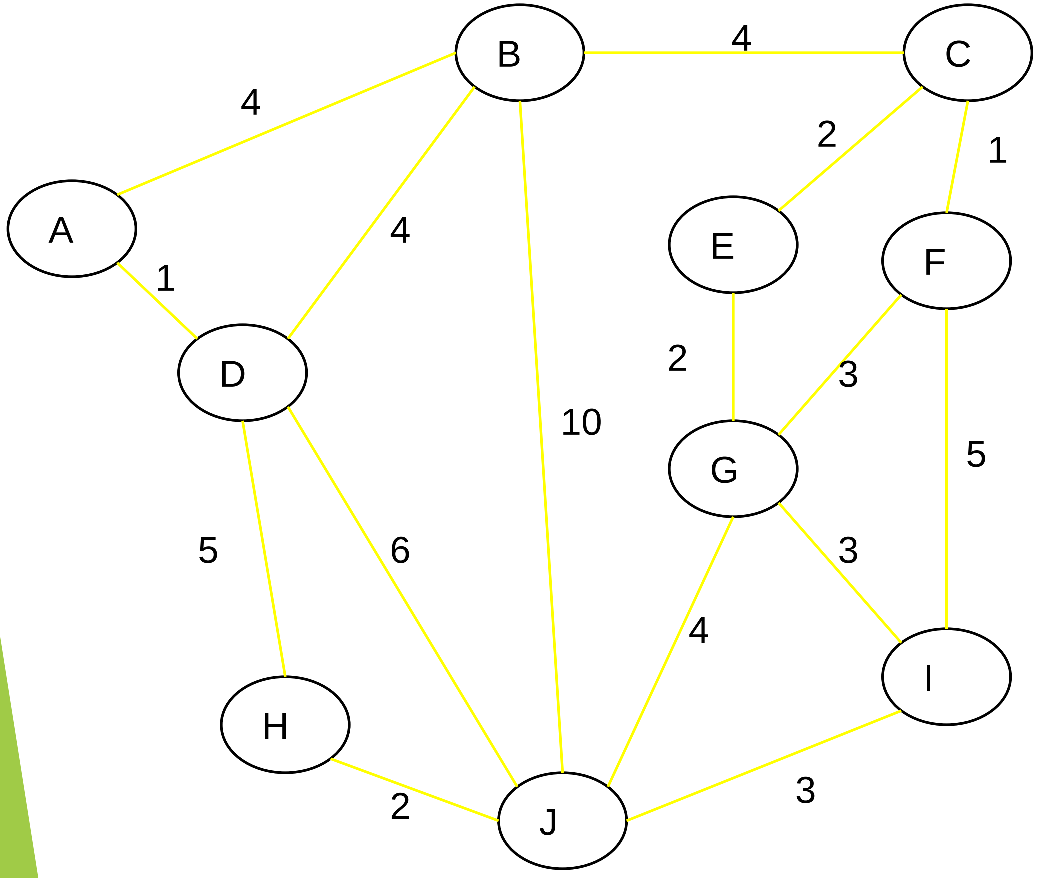
A
B
C
D
E
F
G
H
I
J
4
4
2
1
1
4
10
2
3
5
5
6
3
4
2
3



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

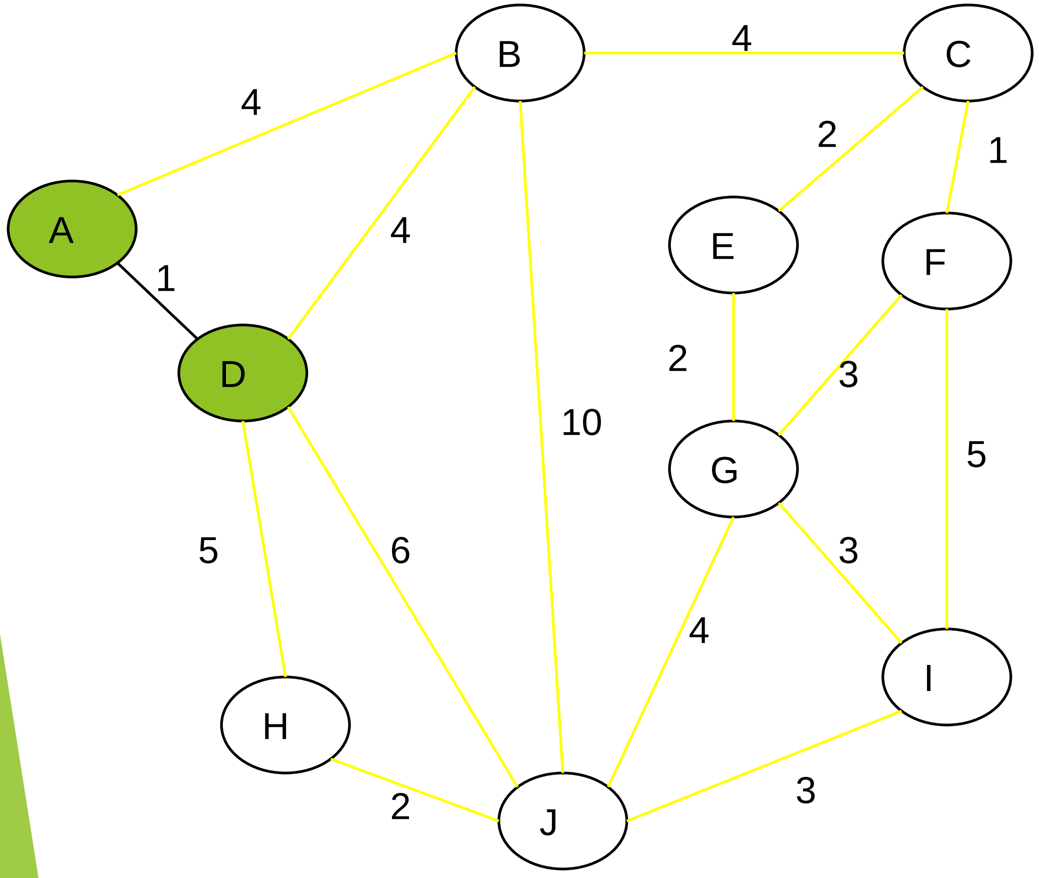
Sort Edges

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

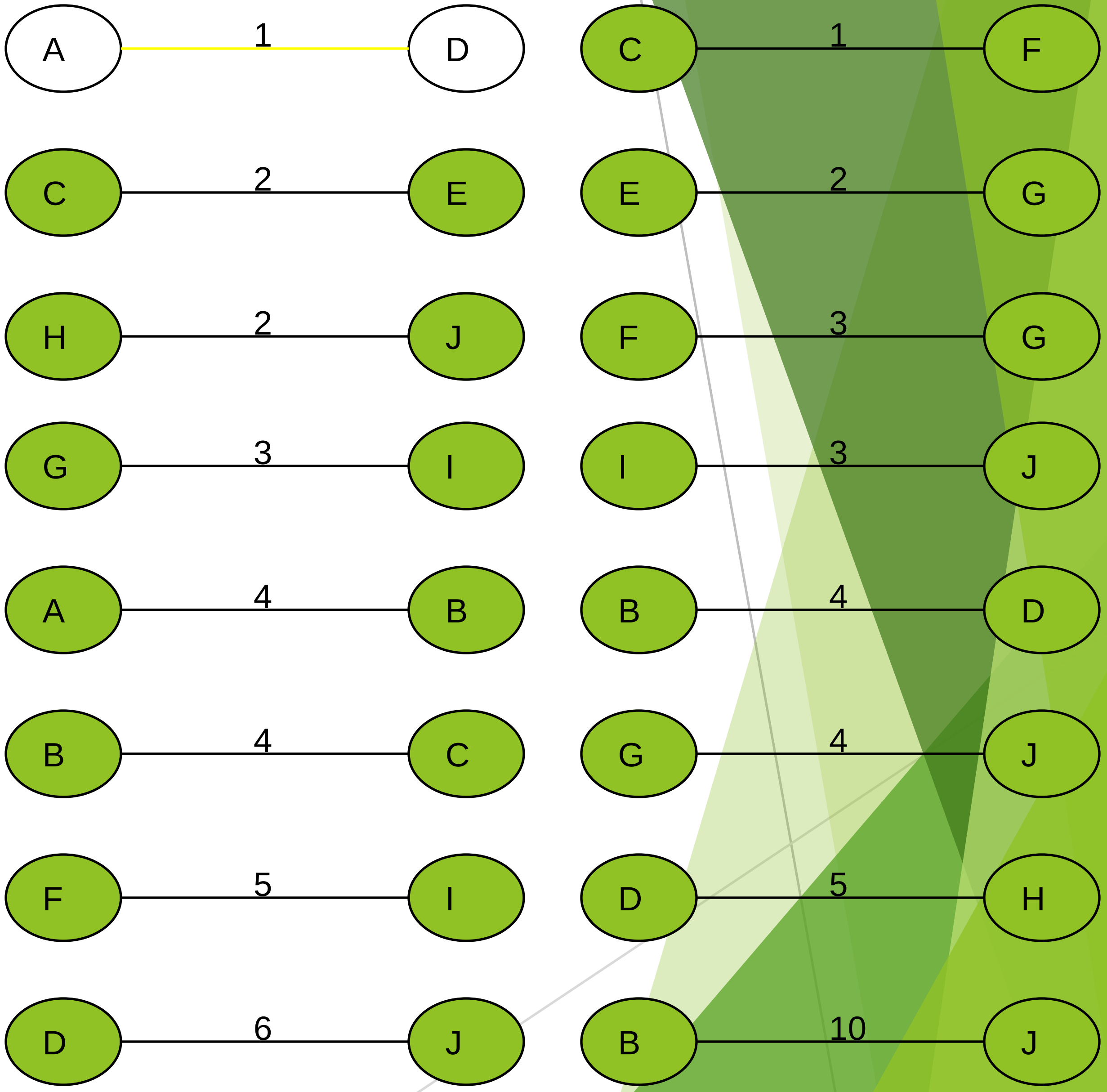


A	1	D
C	1	F
C	2	E
E	2	G
H	2	J
F	3	G
G	3	I
I	3	J
A	4	B
B	4	D
B	4	C
G	4	J
F	5	I
D	5	H
D	6	J
B	10	J

Add Edge

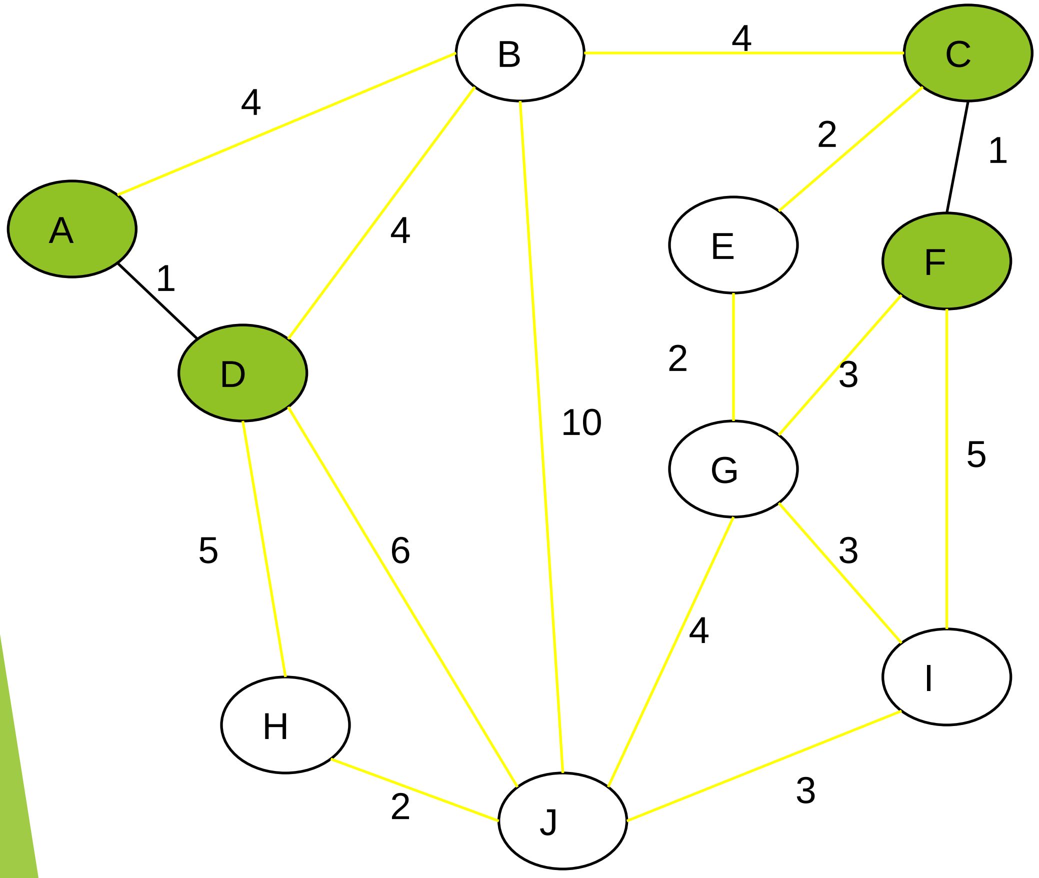


B
4
C
4
2
1
A
4
E
F
1
D
2
3
10
G
5
5
6
3
4
I
H
2
J
3

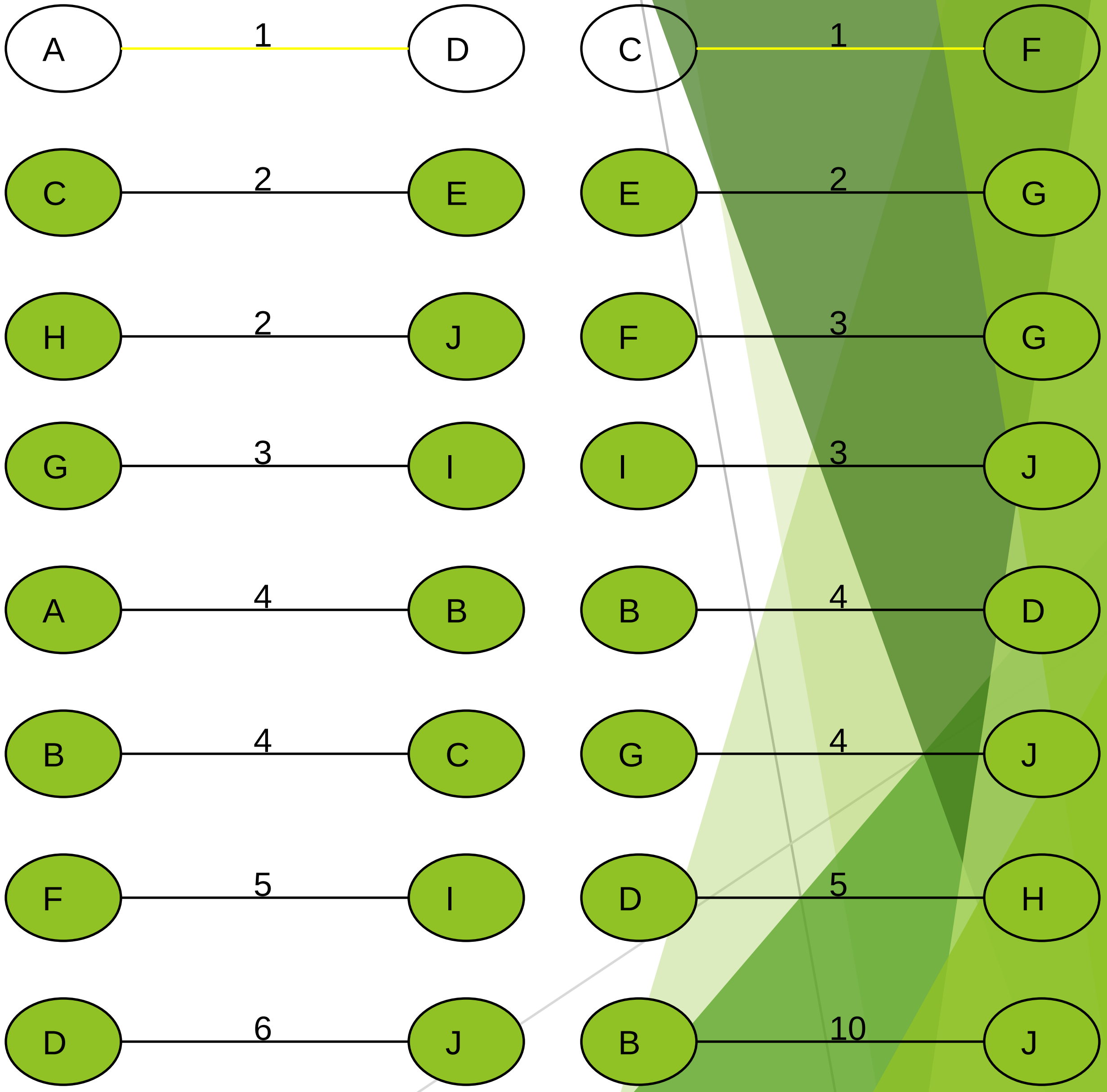


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

Add Edge

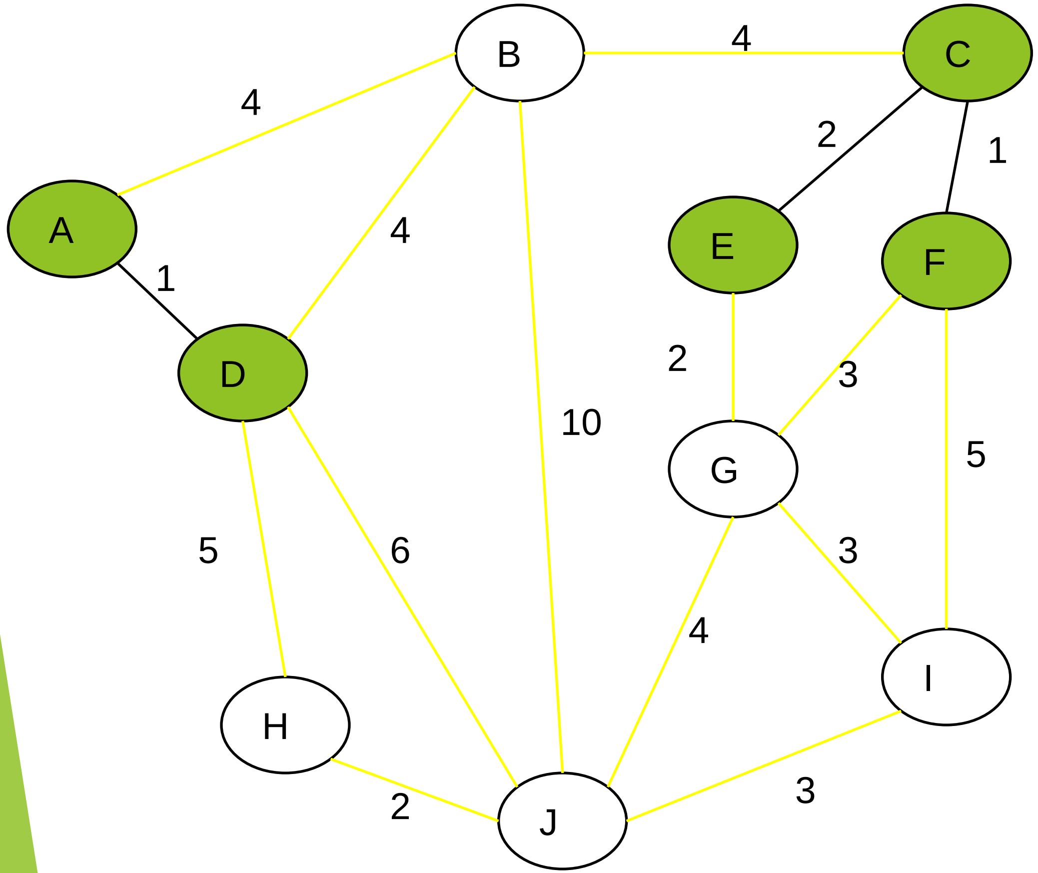


A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
10
2
3
5
5
6
3
4
2
3

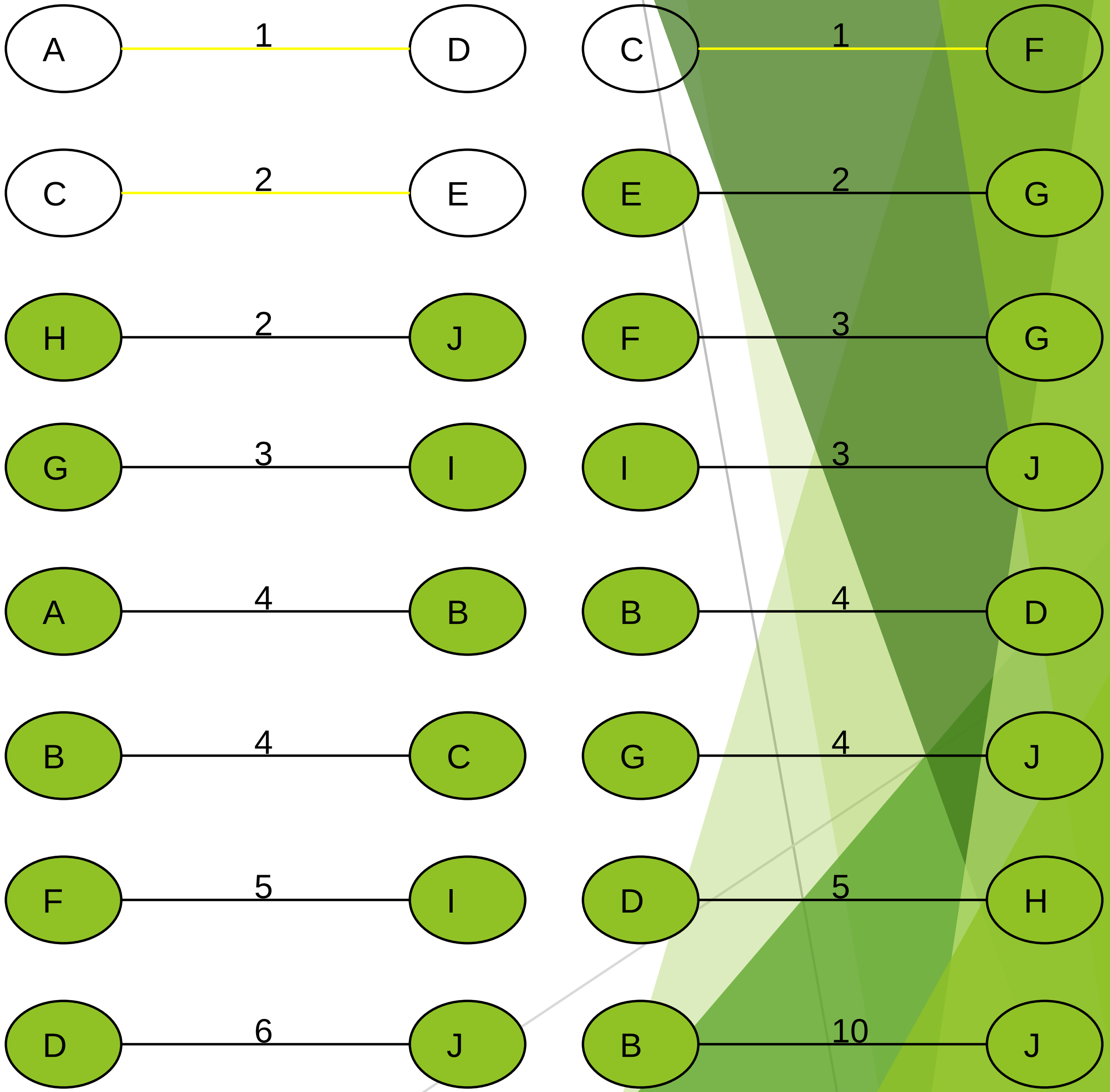


A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

Add Edge

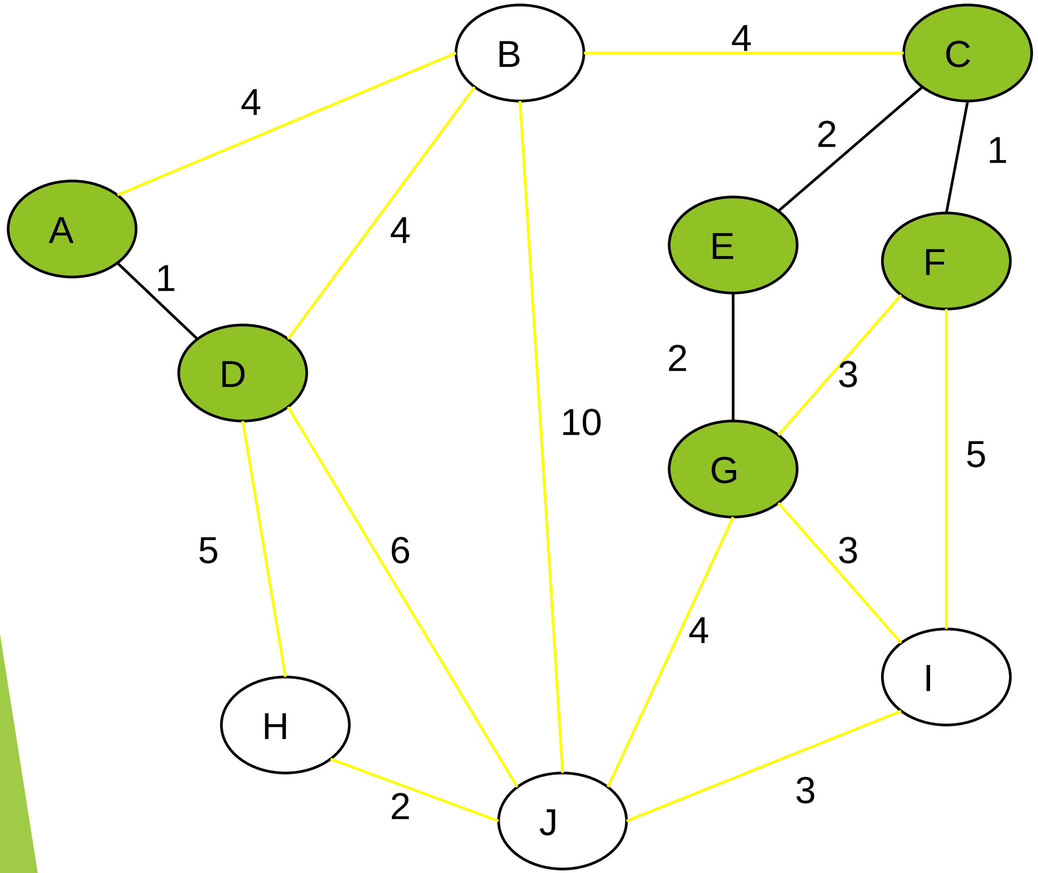


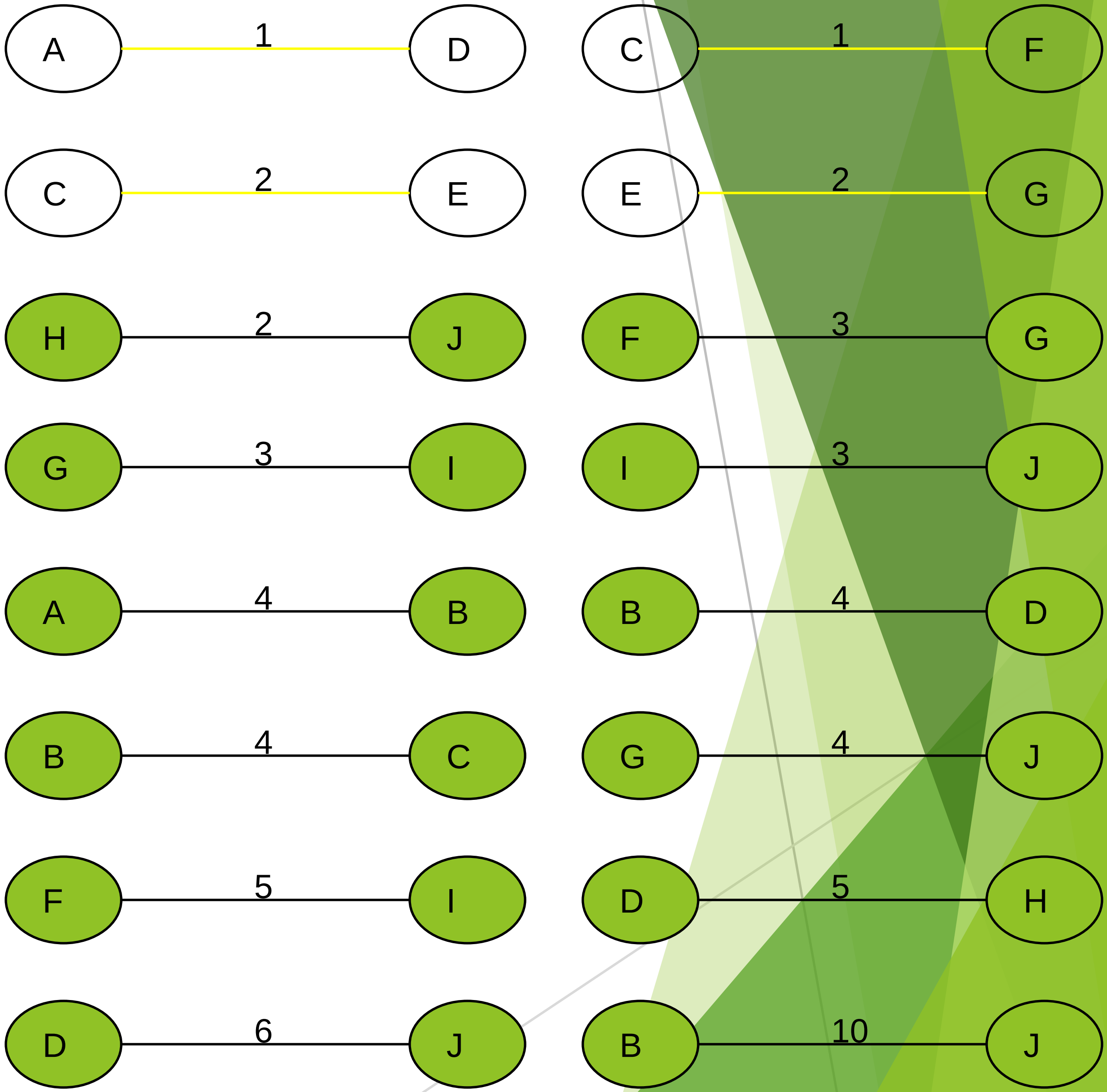
A
B
C
D
E
F
G
H
I
J
4
4
4
1
2
1
2
3
10
5
5
6
3
4
3
2
3



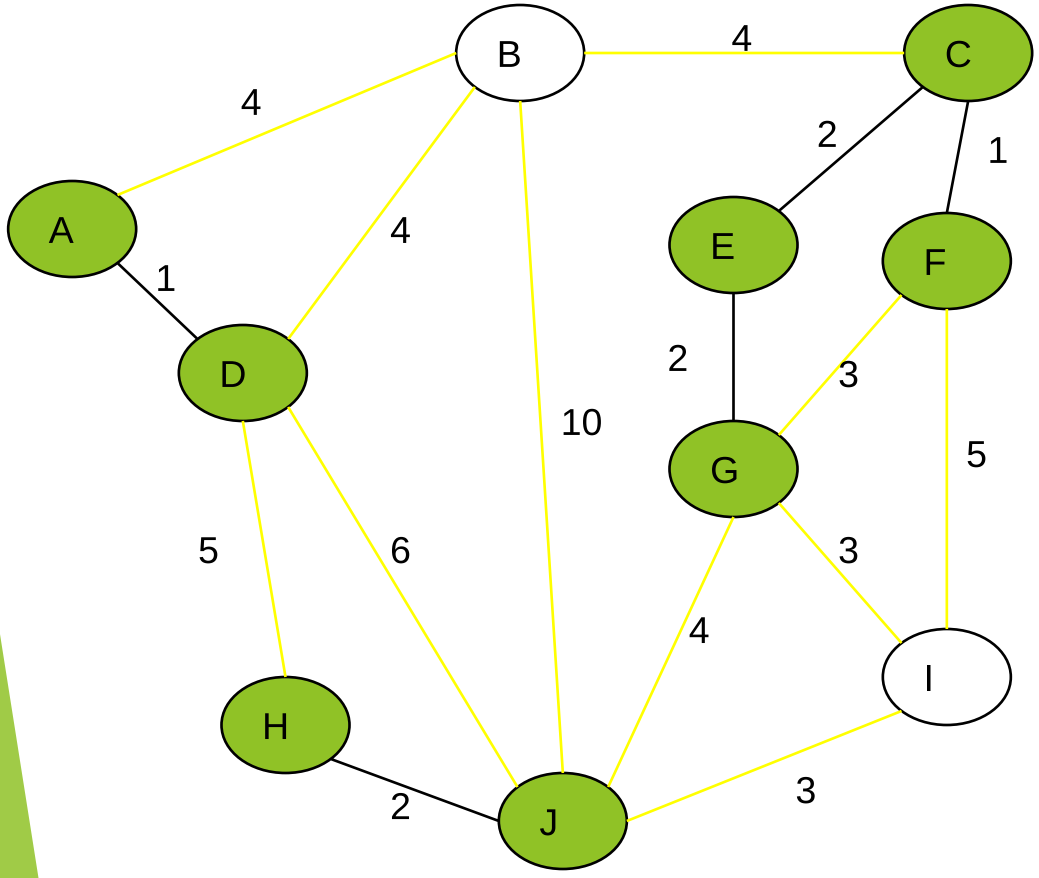
A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

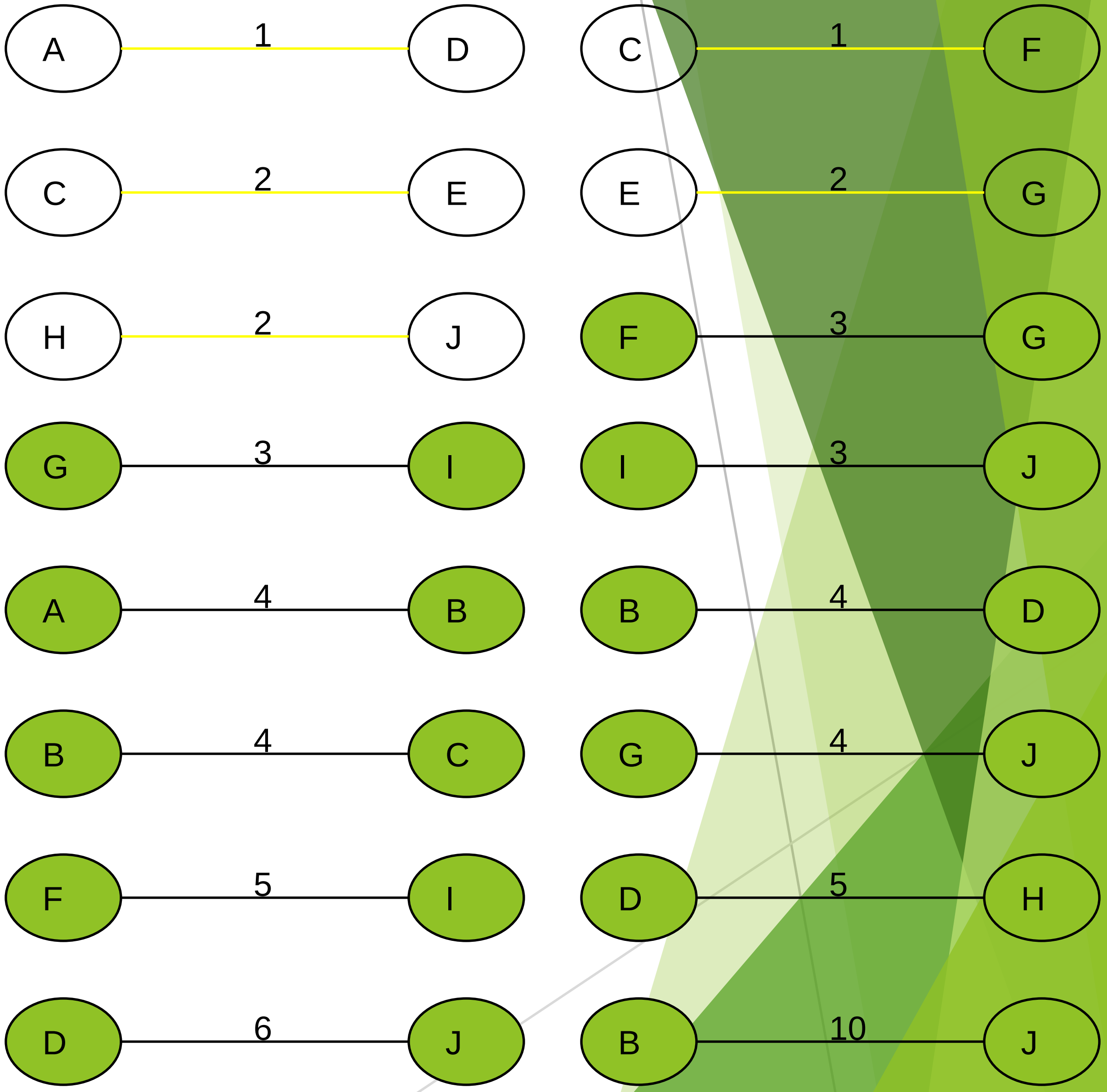
Add Edge


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

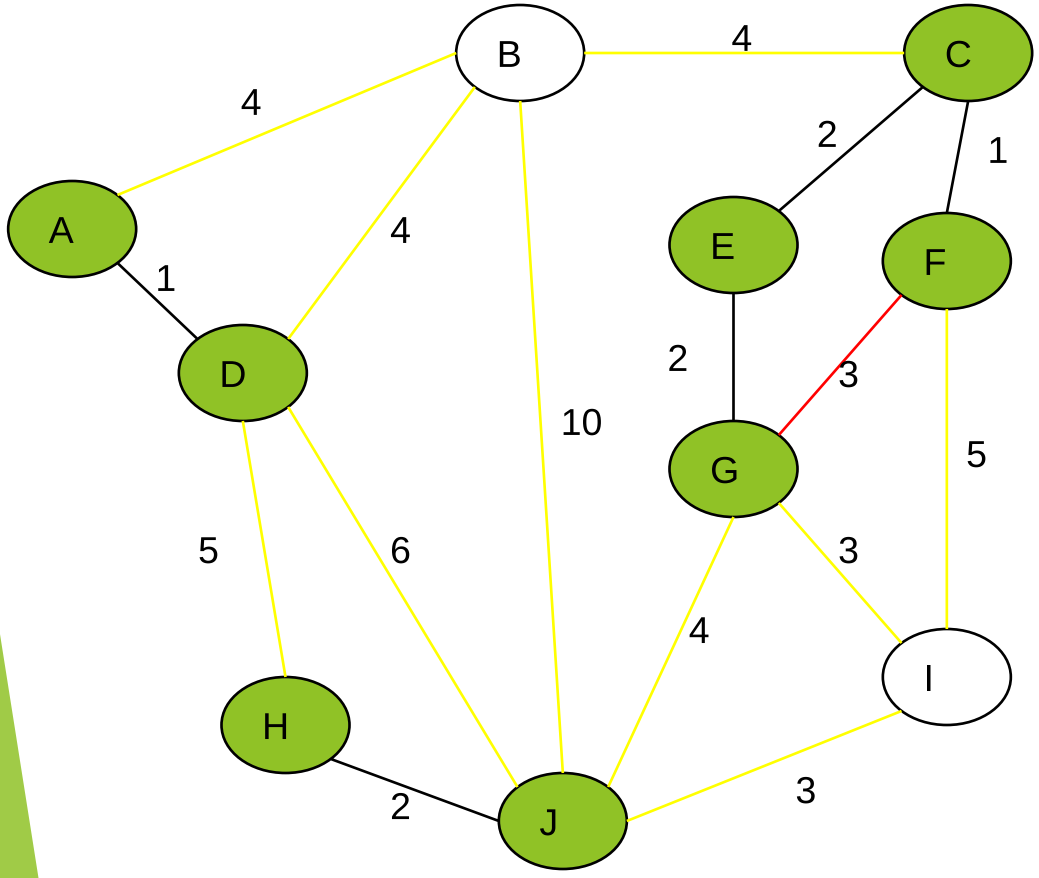
Add Edge


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

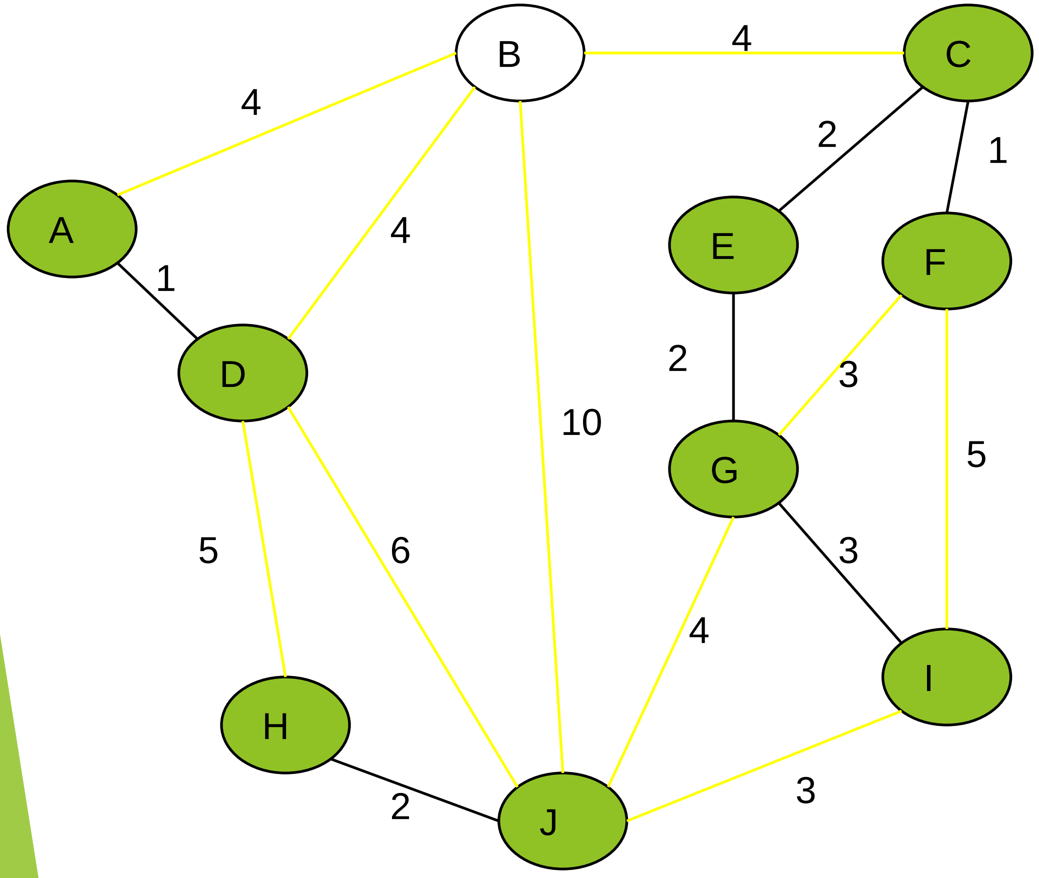
Cycle

Don’t Add Edge

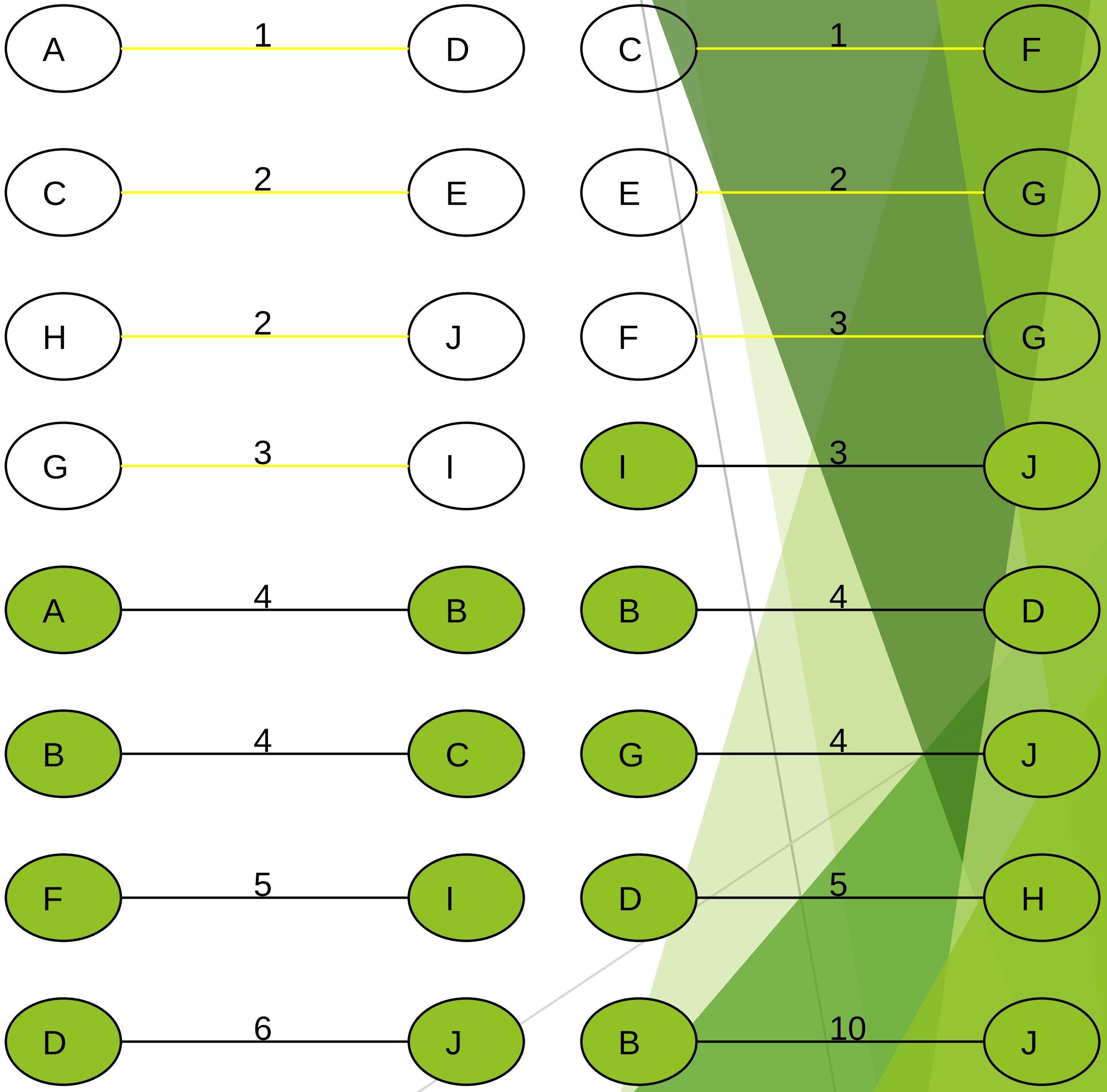


A	1	D
C	2	E
H	2	J
G	3	I
A	4	B
B	4	C
F	5	I
D	6	J
C	1	F
E	2	G
F	3	G
I	3	J
B	4	D
G	4	J
D	5	H
B	10	J

Add Edge

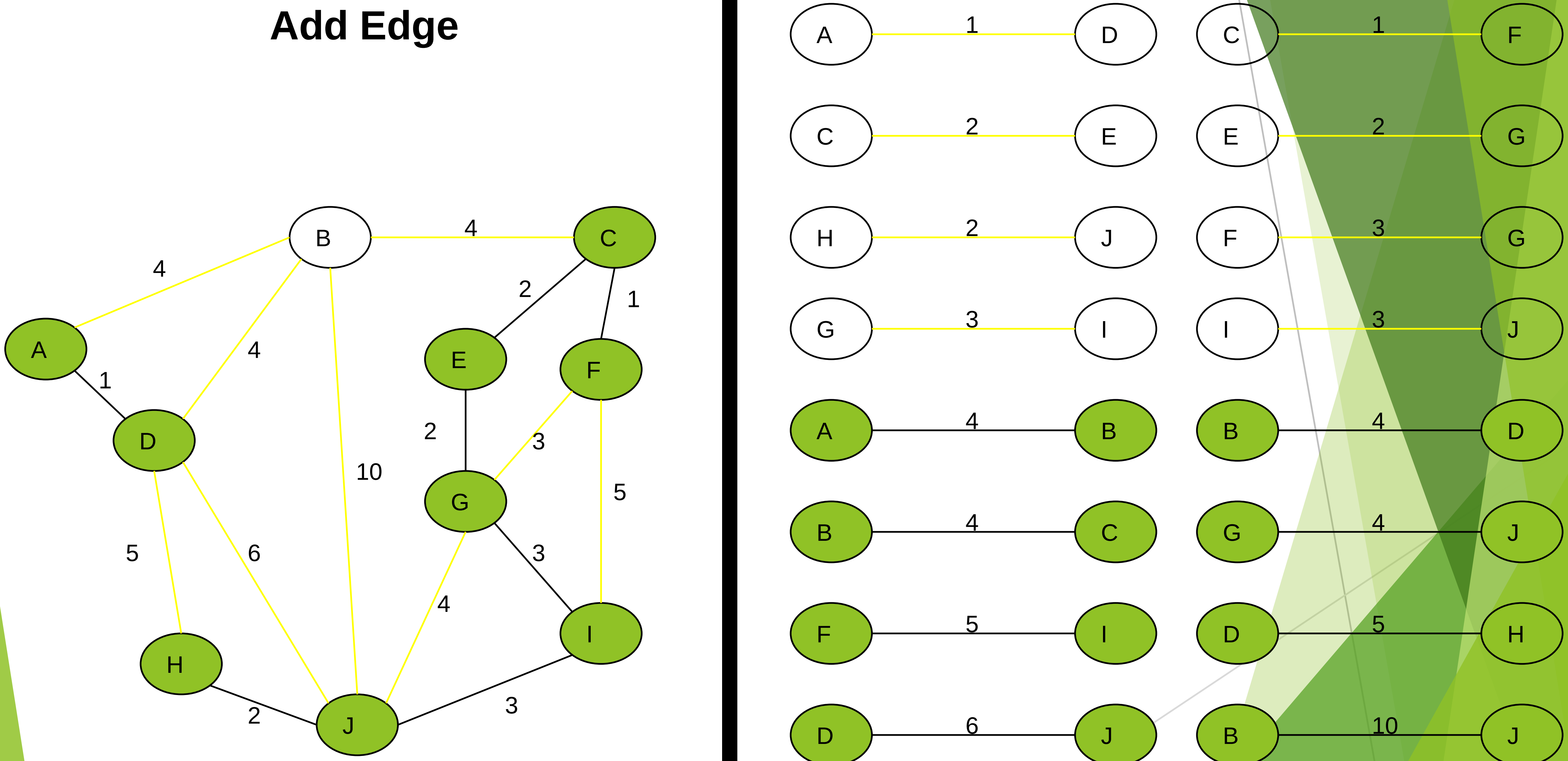


A
B
C
D
E
F
G
H
I
J
4
4
4
1
2
1
10
2
3
5
5
6
3
4
2
3



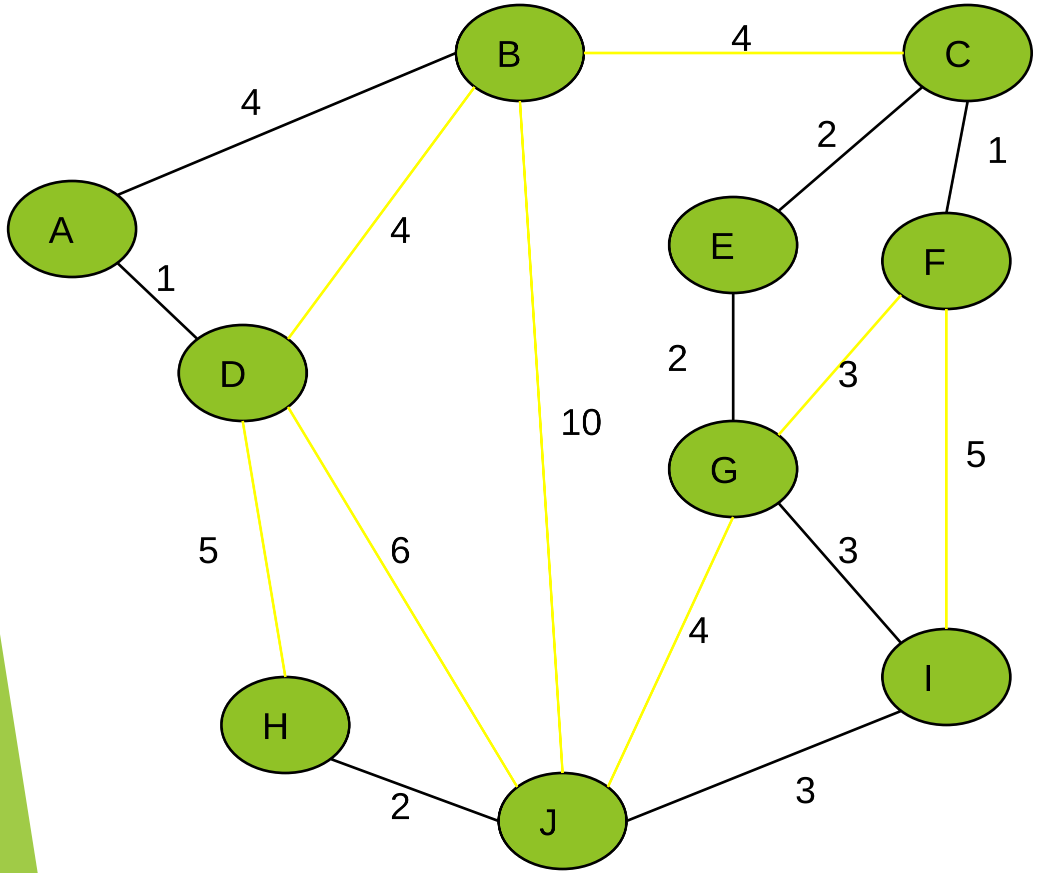
A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

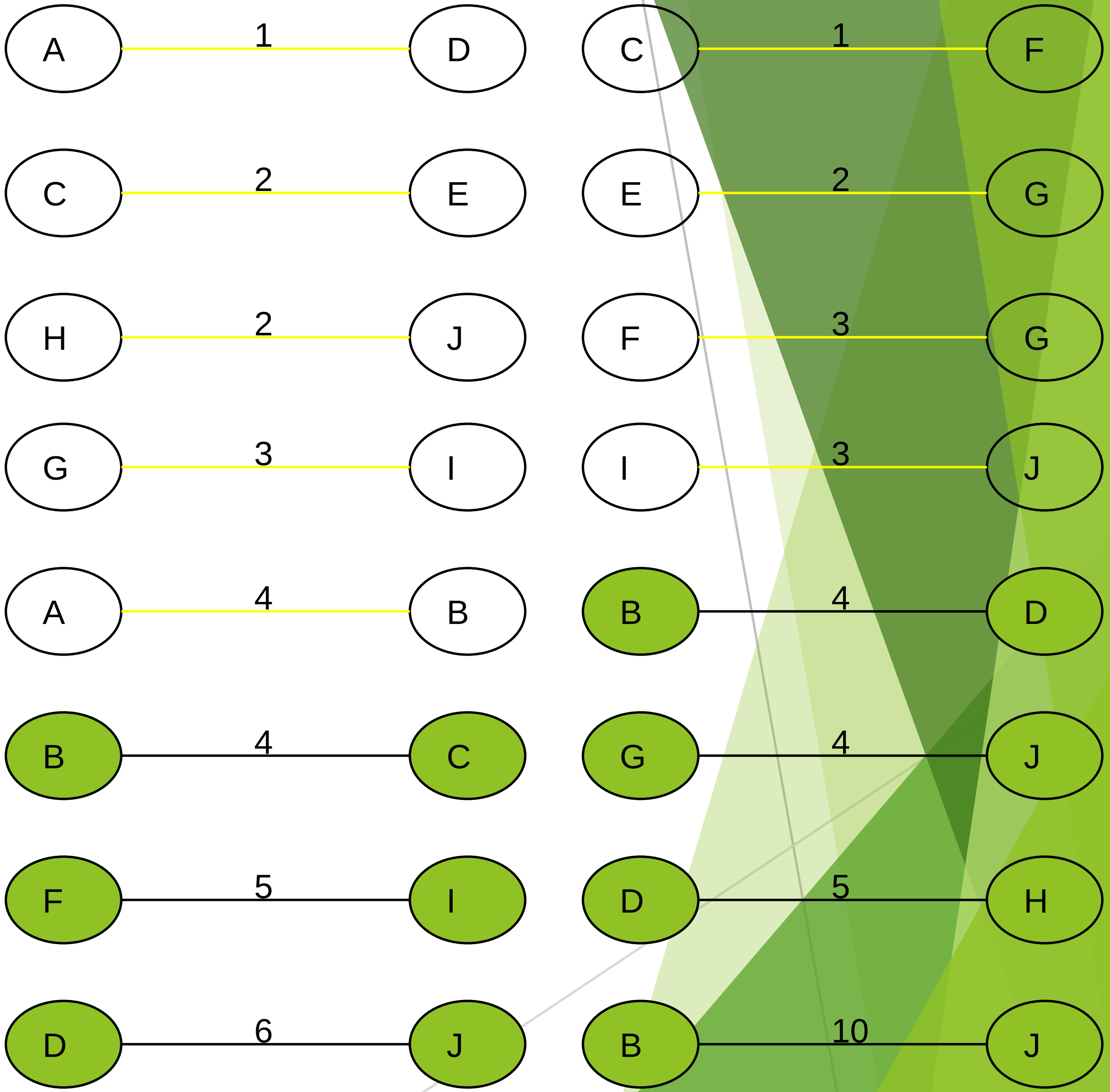
Add Edge



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

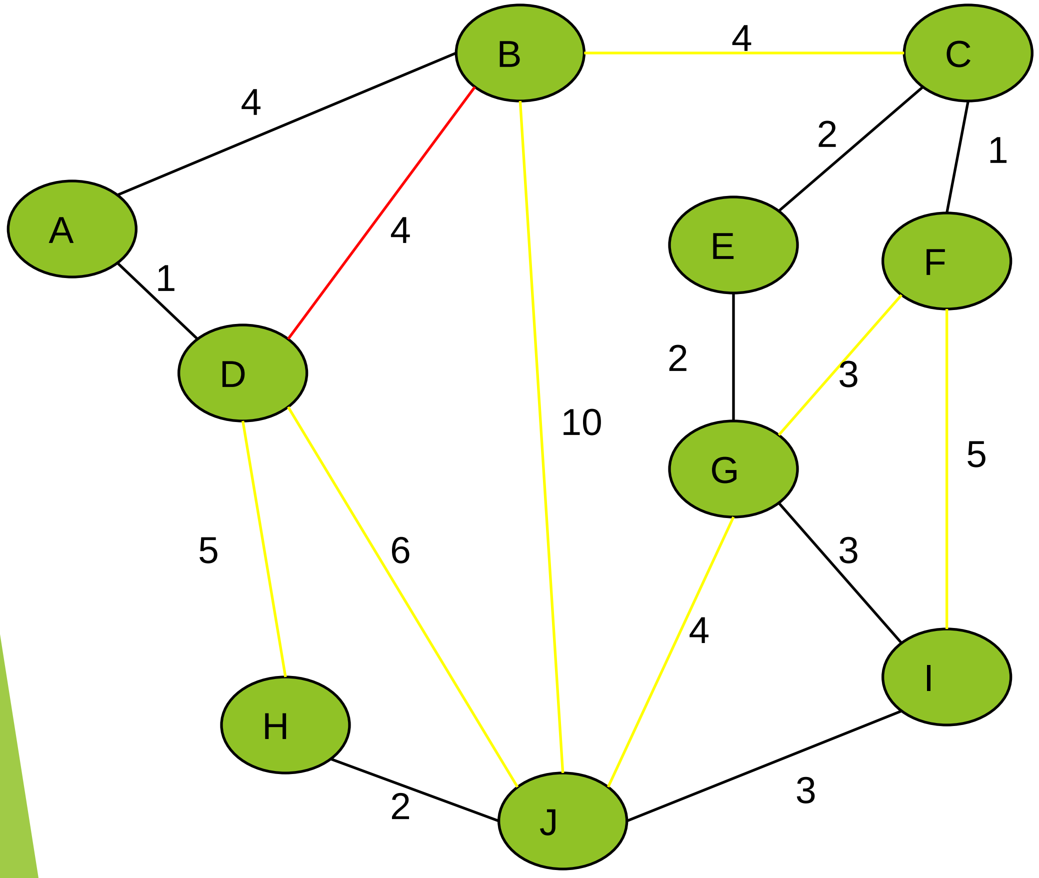
Add Edge

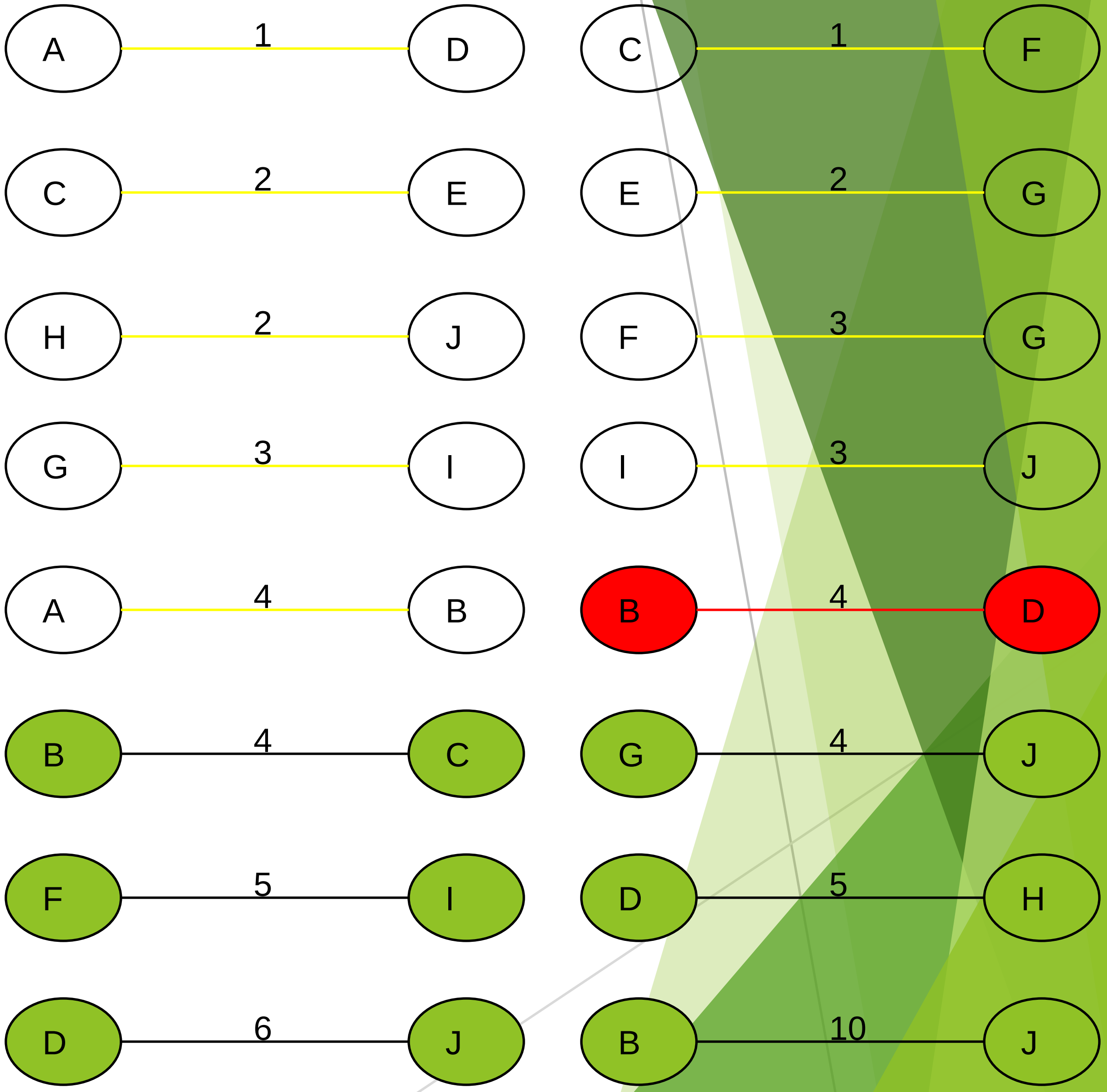

A
B
C
D
E
F
G
H
I
J
4
1
4
10
4
2
1
2
3
5
5
6
4
3
2
3


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

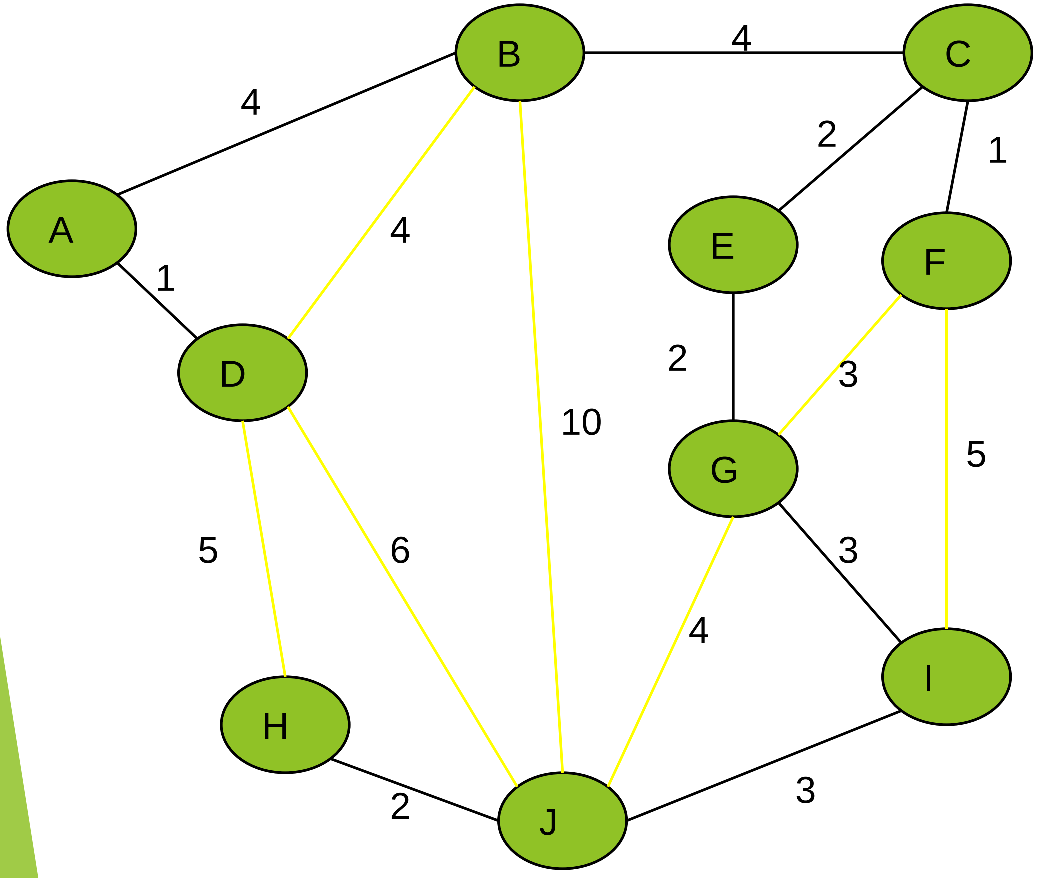
Cycle

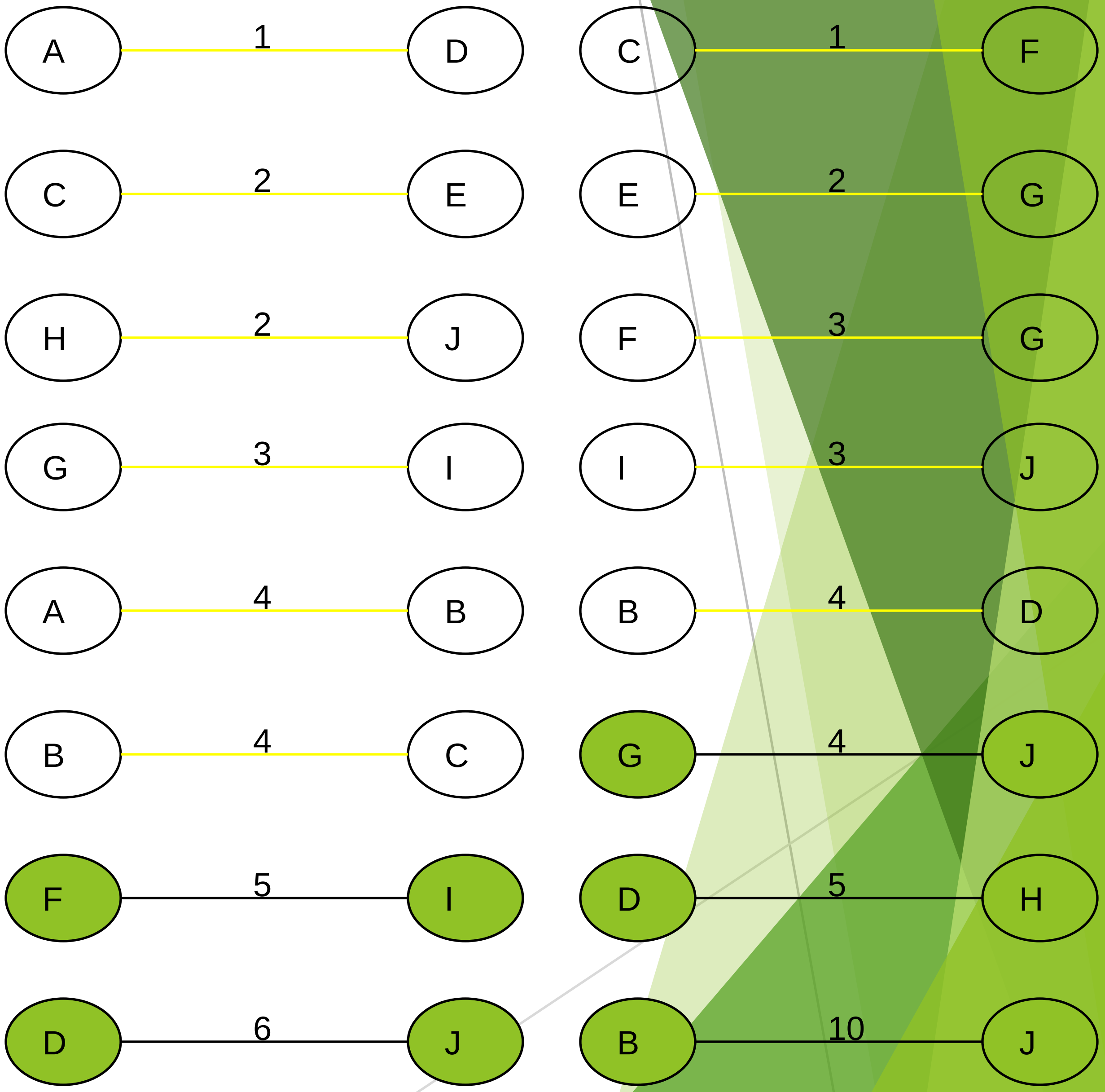
Don't Add Edge



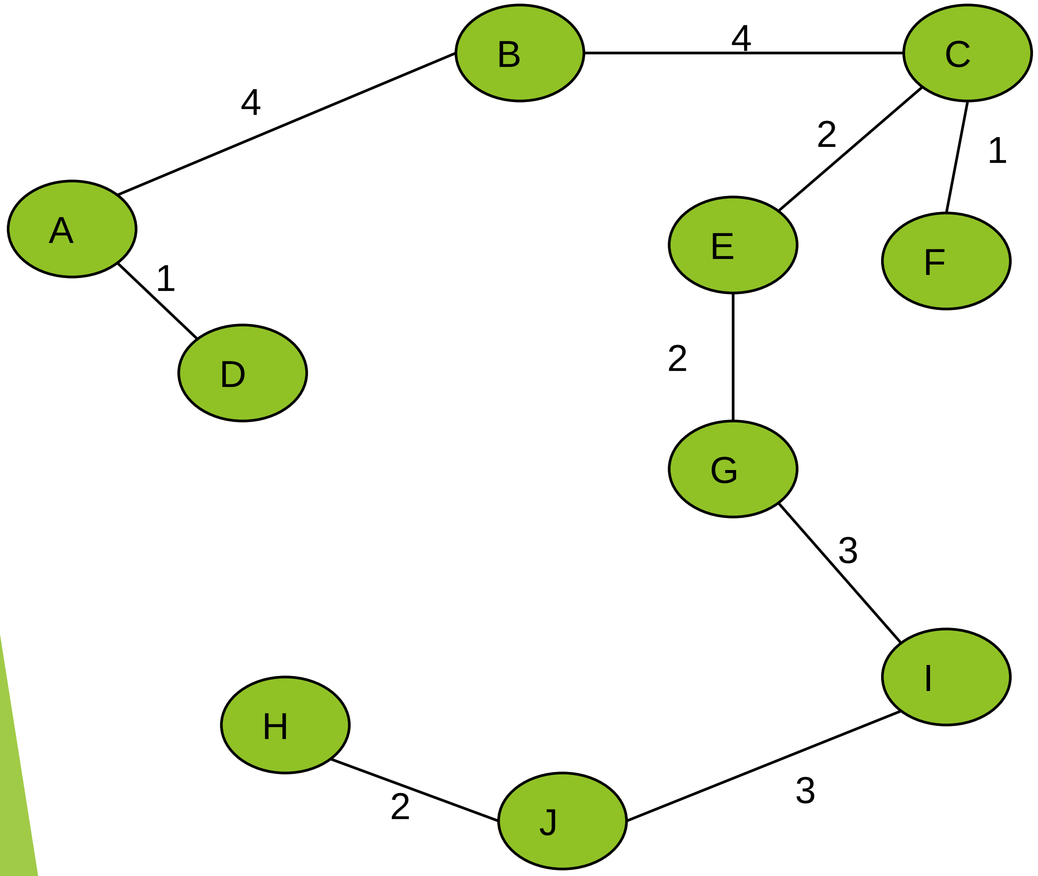


Add Edge

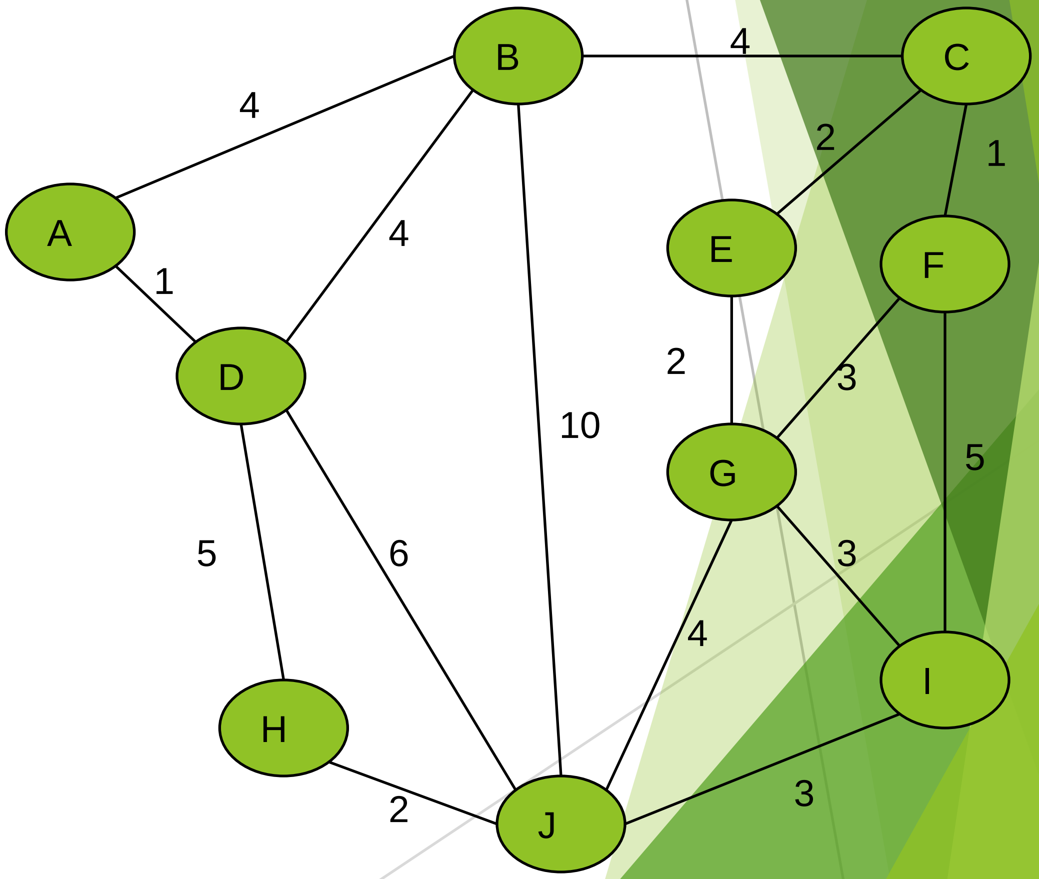

A
B
C
D
E
F
G
H
I
J
4
4
2
1
1
4
2
3
10
5
5
6
3
4
2
3


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



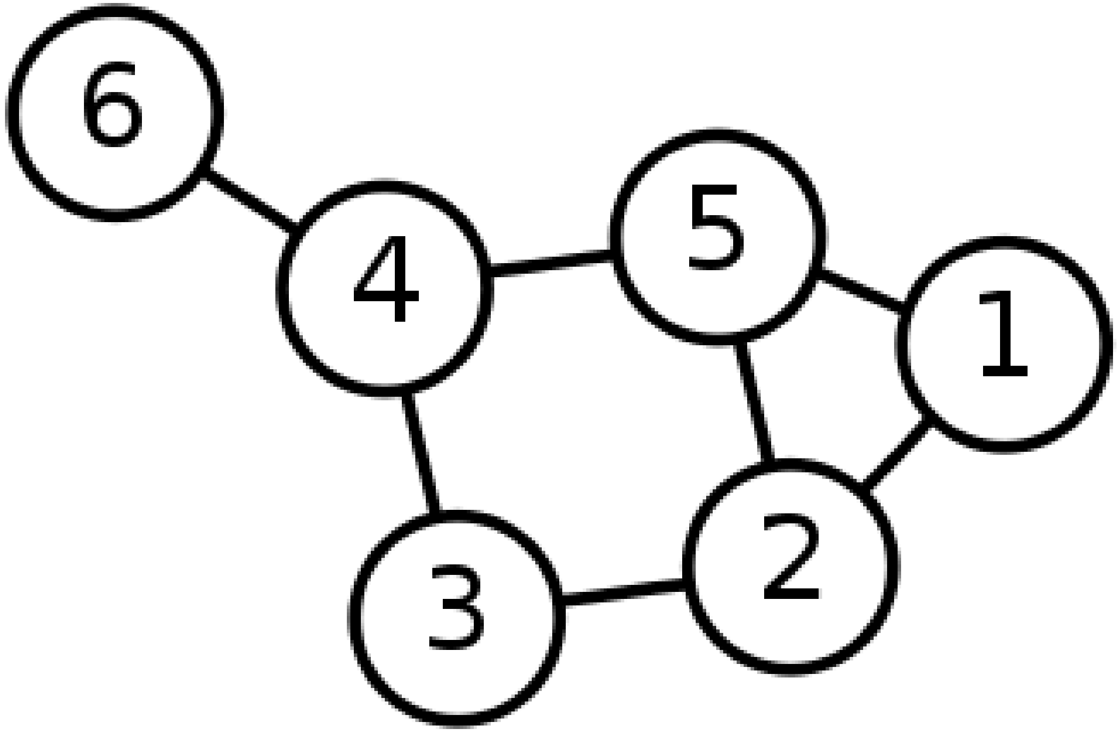
```
Algorithm Kruskal(E, cost, n, t)
// E is the set of edges in G. G has n vertices. cost[u, v] is the
// cost of edge (u, v). t is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for i := 1 to n do parent[i] := -1;
    // Each vertex is in a different set.
    i := 0; mincost := 0.0;
    while ((i < n - 1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u, v) from the heap
        and reheapify using Adjust;
        j := Find(u); k := Find(v);
        if (j ≠ k) then
        {
            i := i + 1;
            t[i, 1] := u; t[i, 2] := v;
            mincost := mincost + cost[u, v];
            Union(j, k);
        }
    }
    if (i ≠ n - 1) then write ("No spanning tree");
    else return mincost;
}
```

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

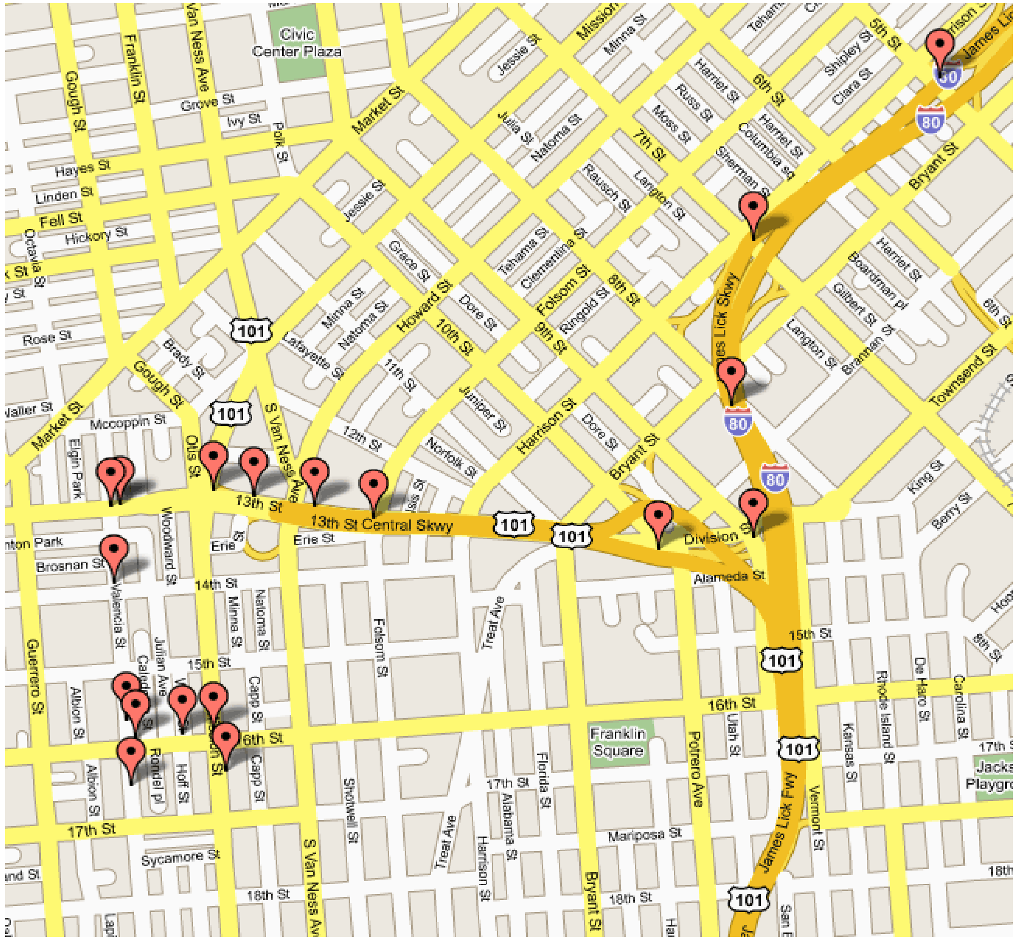
Single-Source Shortest Paths

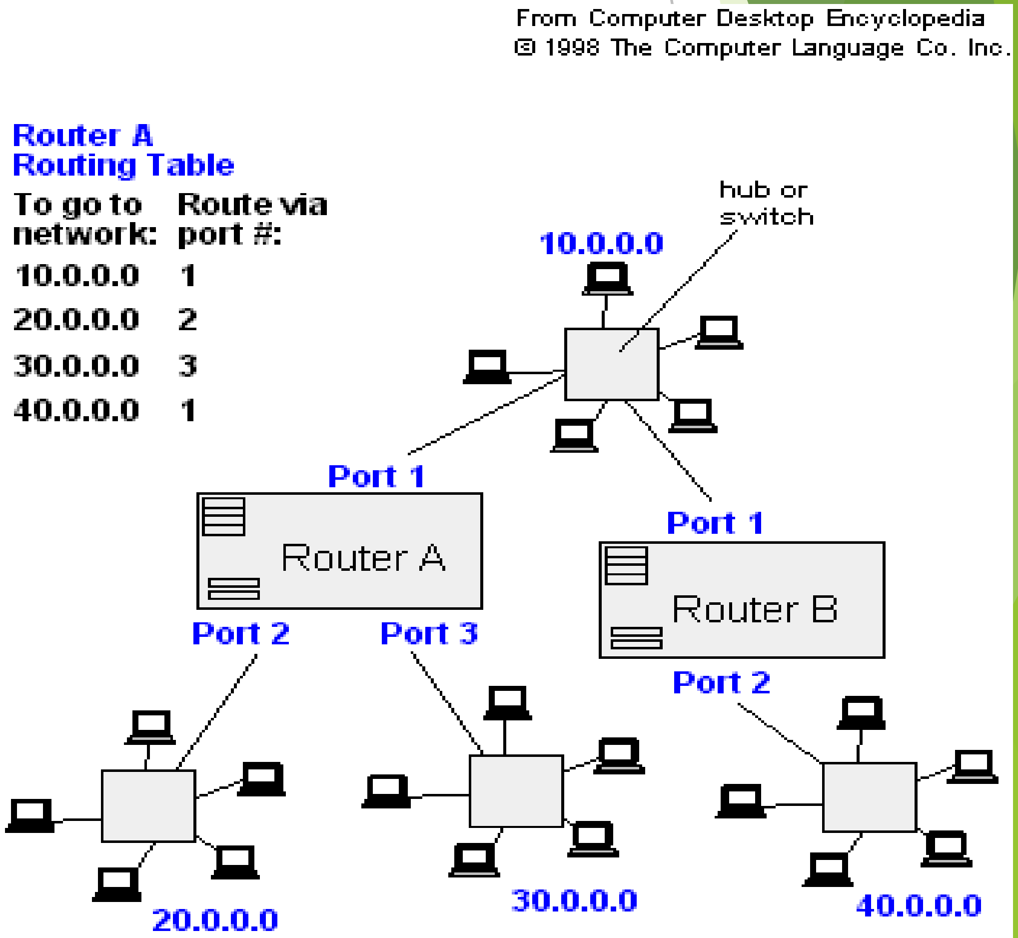
- The problem of finding shortest paths from a source vertex v to all other vertices in the graph.



Application of SSSP

- Maps (Map Quest, Google Maps)
- - Routing Systems





Dijkstra's Algorithm

- This is a solution to the single-source shortest path problem in graph theory.

- Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

- Input: Weighted graph $G=\{E,V\}$ and source vertex $v \in V$, such that all edge weights are nonnegative

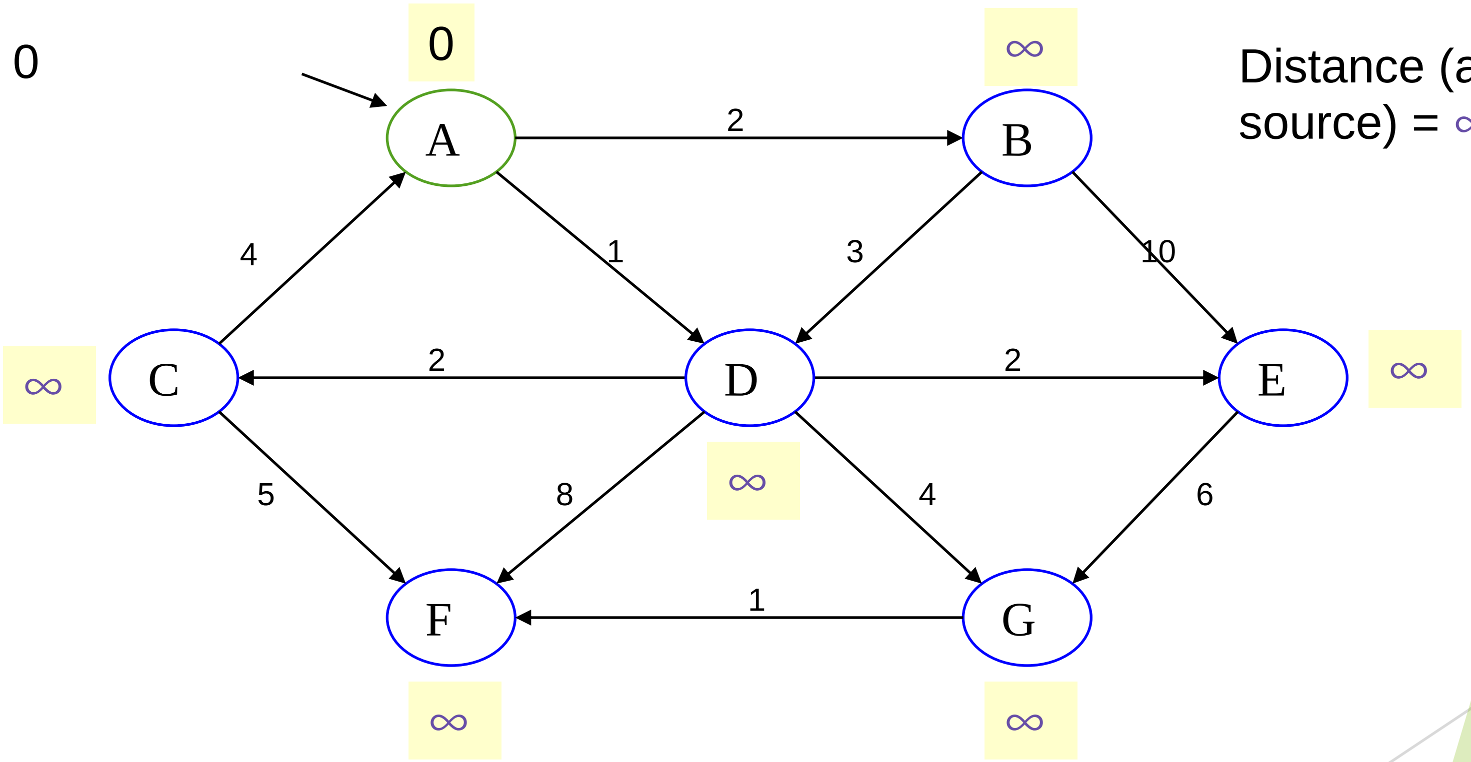
- Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Approach

- The algorithm computes for each vertex u the distance to u from the start vertex v, that is, the weight of a shortest path between v and u.
- The algorithm keeps track of the set of vertices for which the distance has been computed, called the cloud C
- Every vertex has a label D associated with it. For any vertex u, D[u] stores an approximation of the distance between v and u. The algorithm will update a D[u] value when it finds a shorter path from v to u.
- When a vertex u is added to the cloud, its label D[u] is equal to the actual (final) distance between the starting vertex v and vertex u.

Example: Initialization

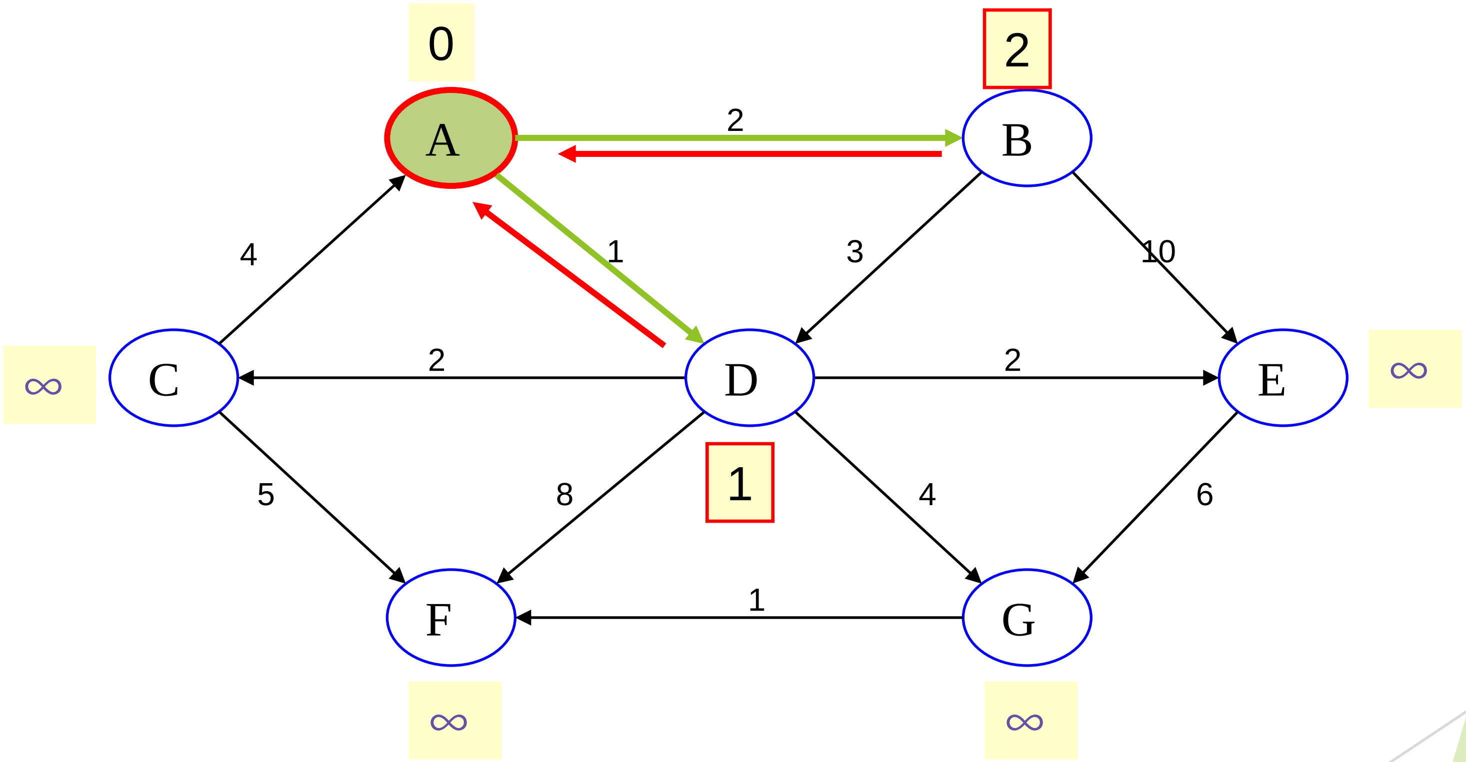
Distance(source) = 0



Distance (all vertices but source) = ∞

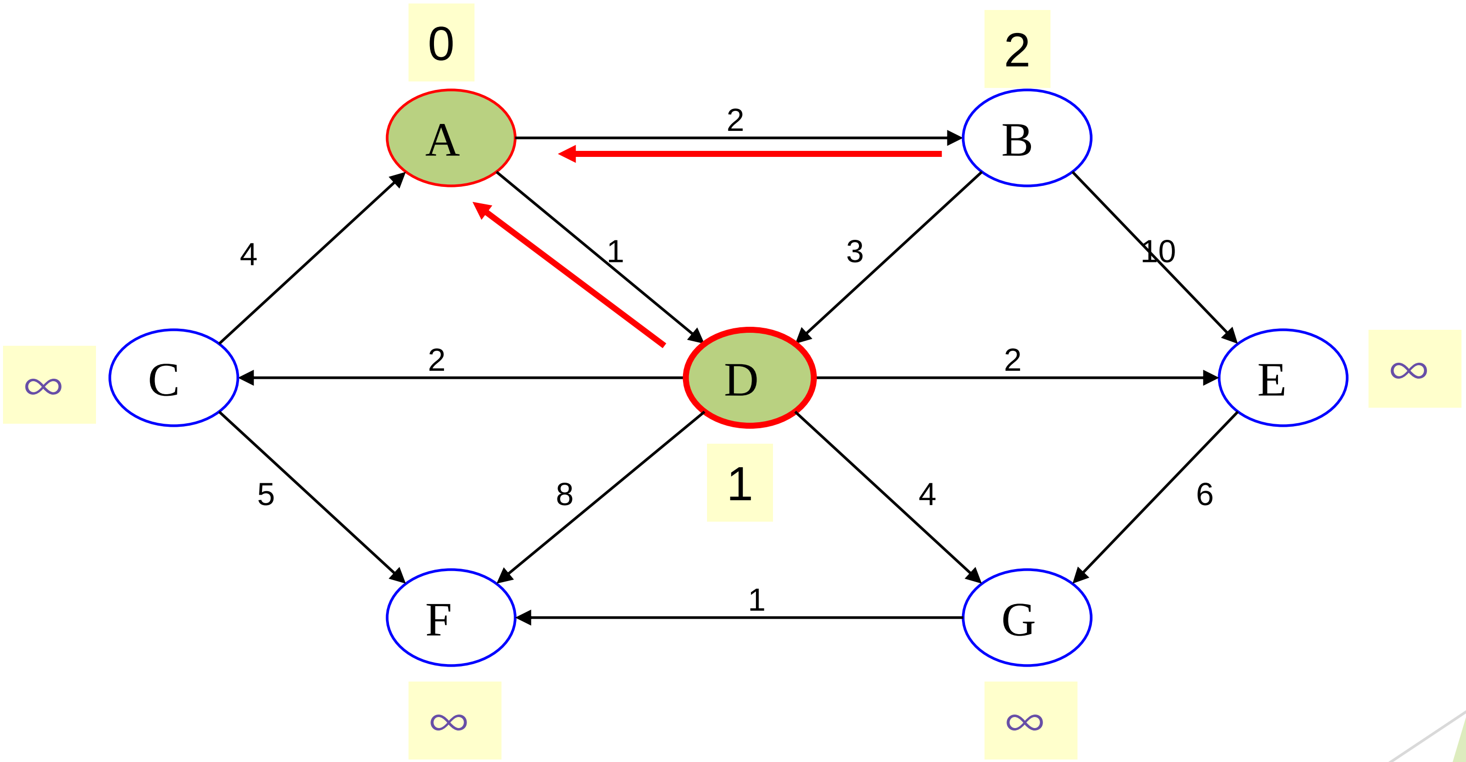
Pick vertex in List with minimum distance.

Example: Update neighbors' distance



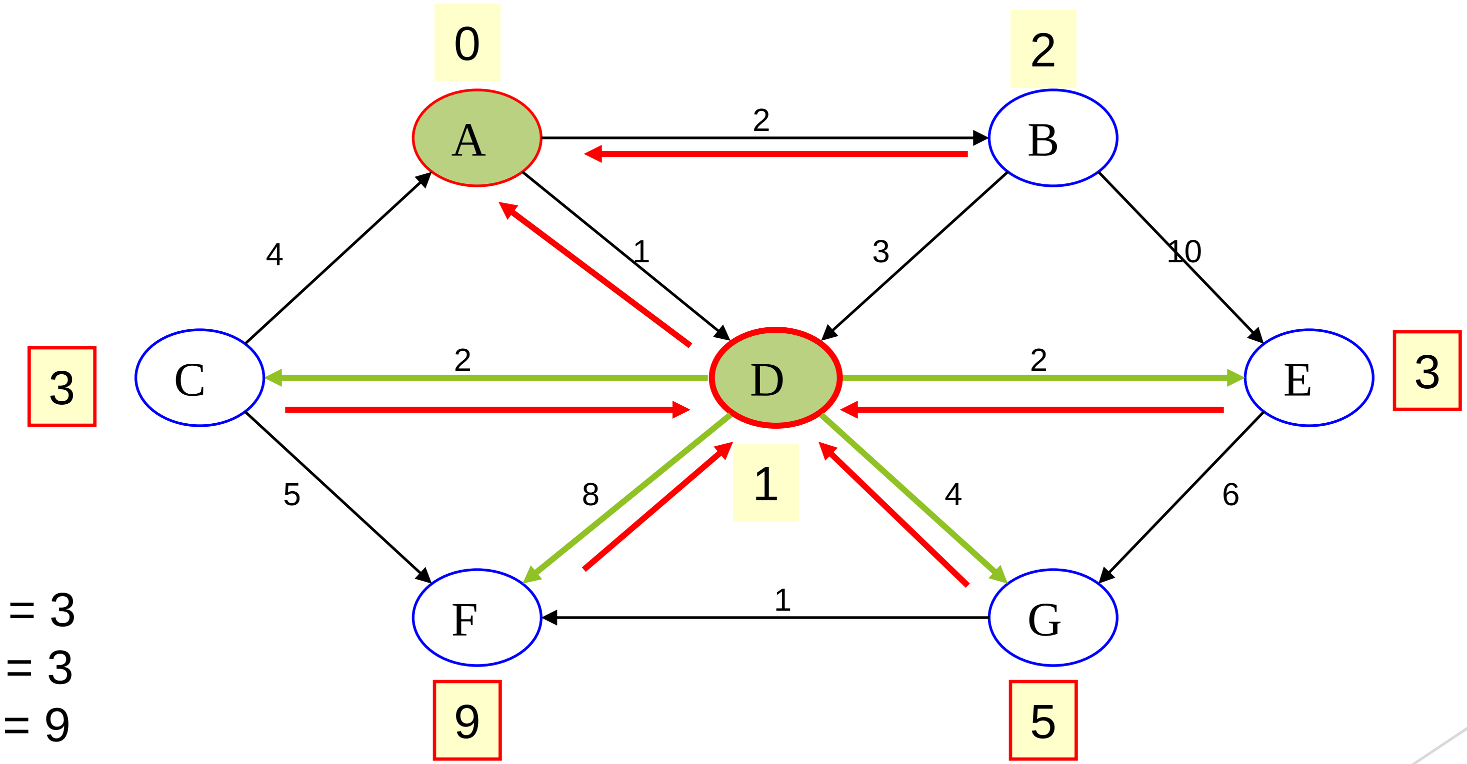
Distance(B) = 2
Distance(D) = 1

Example: Remove vertex with minimum distance



Pick vertex in List with minimum distance, i.e., D

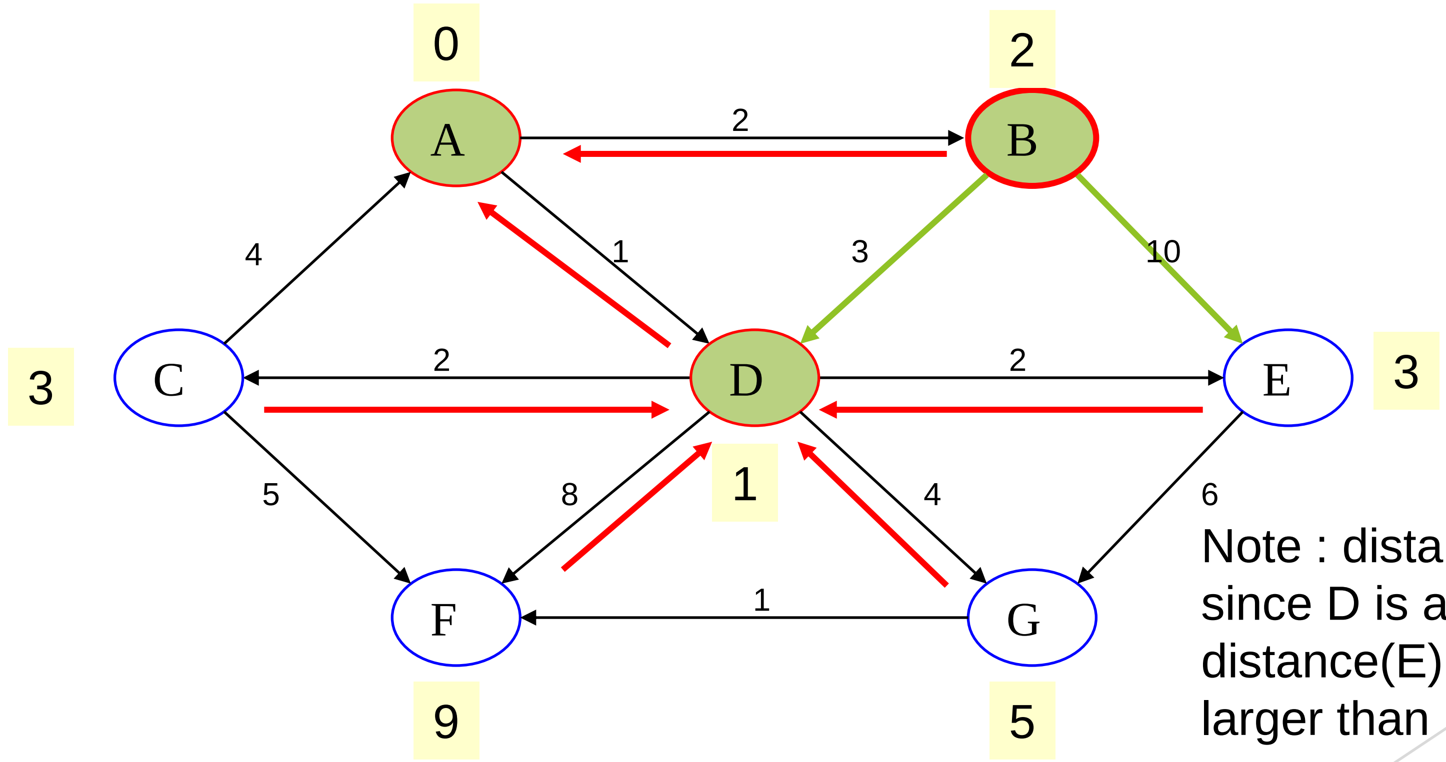
Example: Update neighbors



Distance(C) = 1 + 2 = 3
Distance(E) = 1 + 2 = 3
Distance(F) = 1 + 8 = 9
Distance(G) = 1 + 4 = 5

Example: Continued...

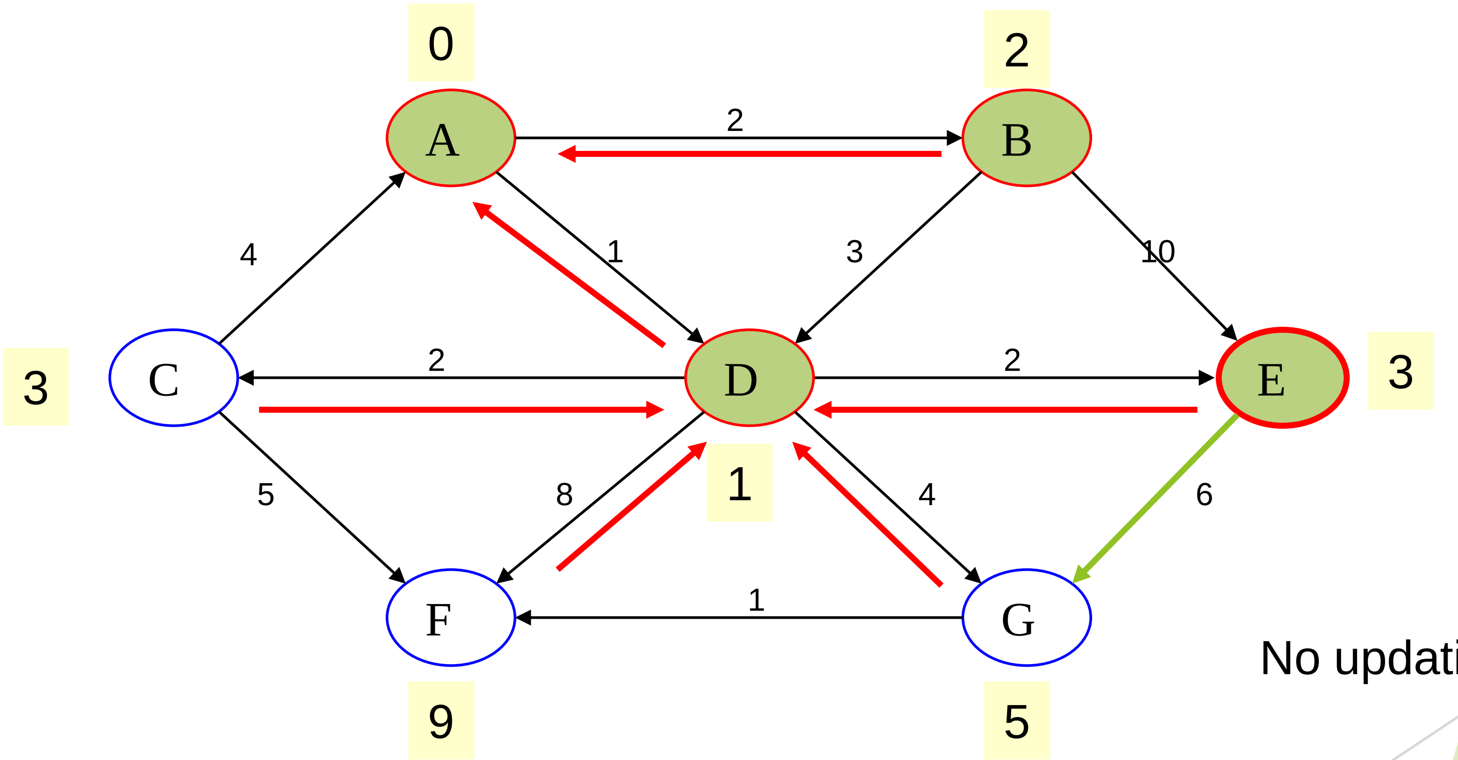
Pick vertex in List with minimum distance (B) and update neighbors



Note : distance(D) not updated since D is already known and distance(E) not updated since it is larger than previously computed

Example: Continued...

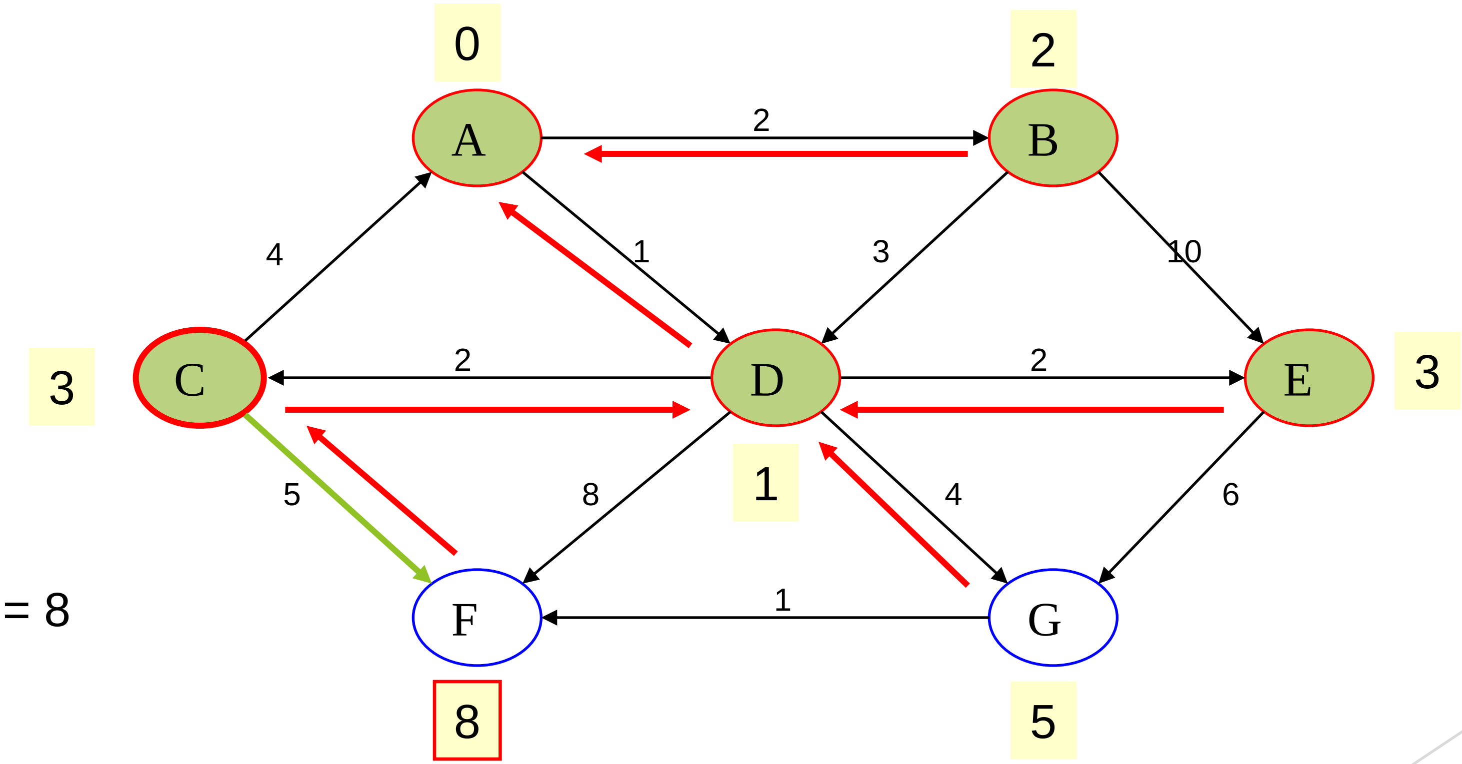
Pick vertex List with minimum distance (E) and update neighbors



No updating

Example: Continued...

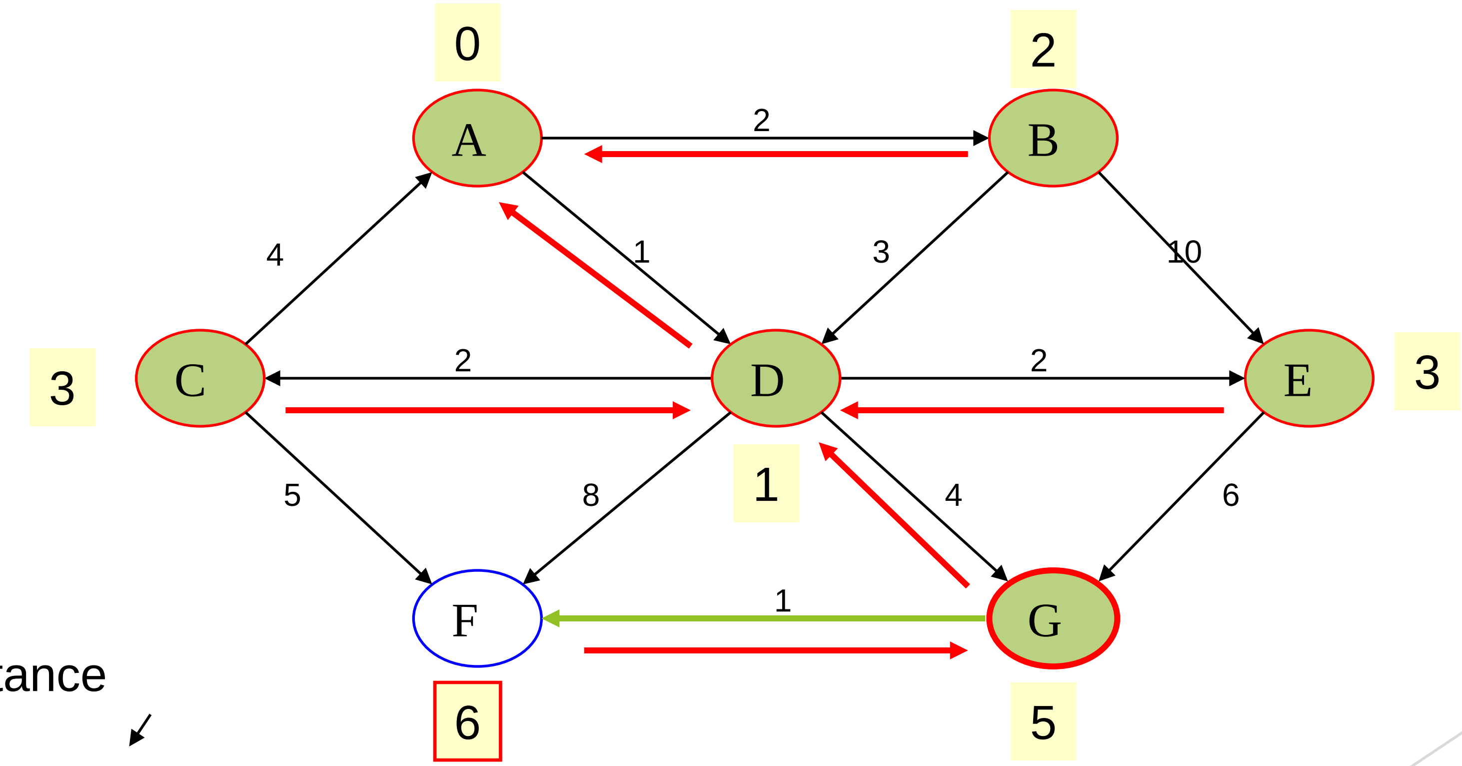
Pick vertex List with minimum distance (C) and update neighbors



Distance(F) = 3 + 5 = 8

Example: Continued...

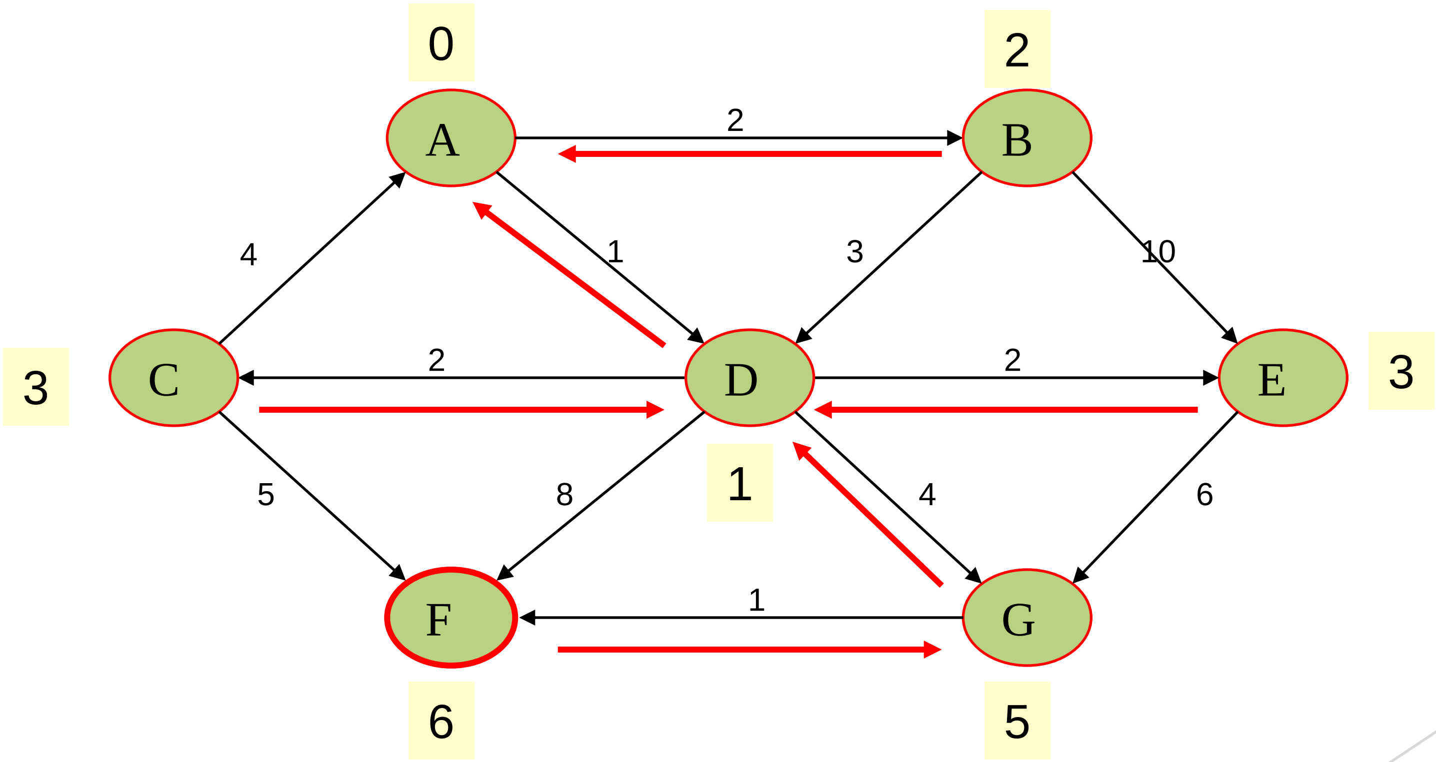
Pick vertex List with minimum distance (G) and update neighbors



Previous distance

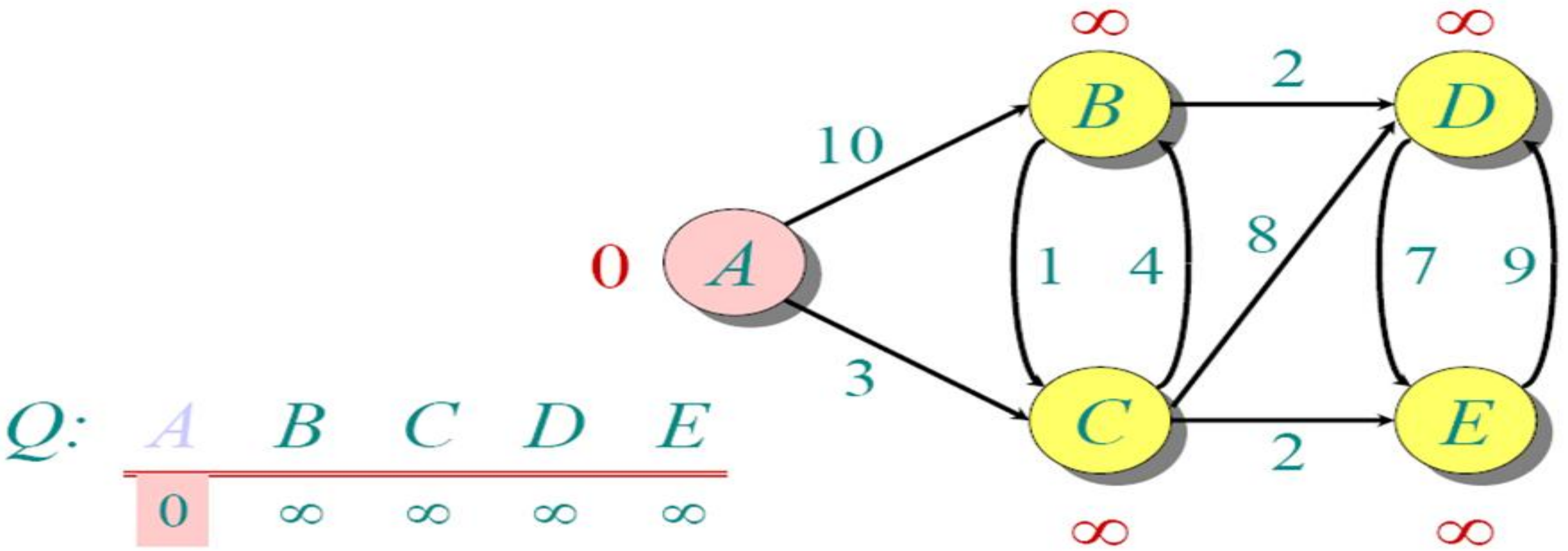
Distance(F) = min (8, 5+1) = 6

Example (end)

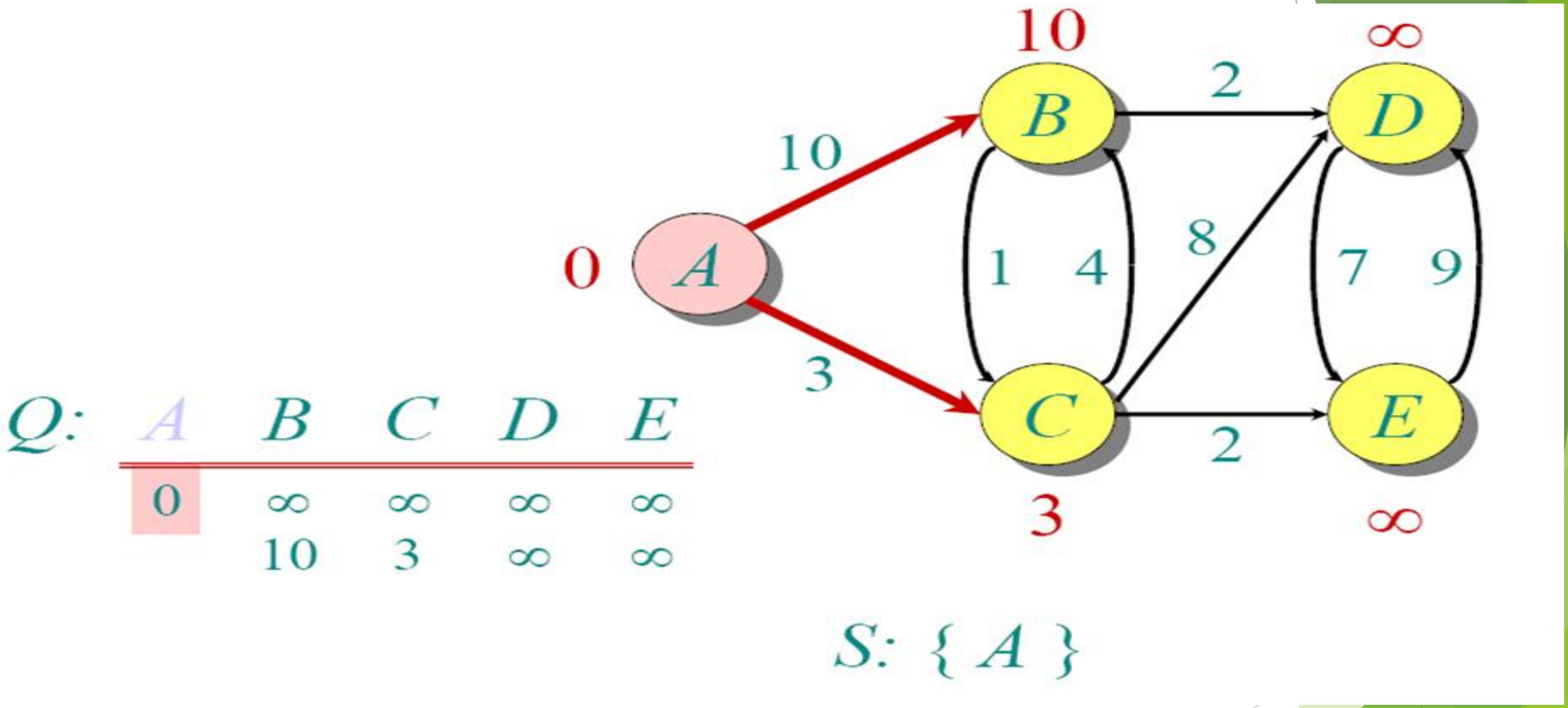


Pick vertex not in S with lowest cost (F) and update neighbors

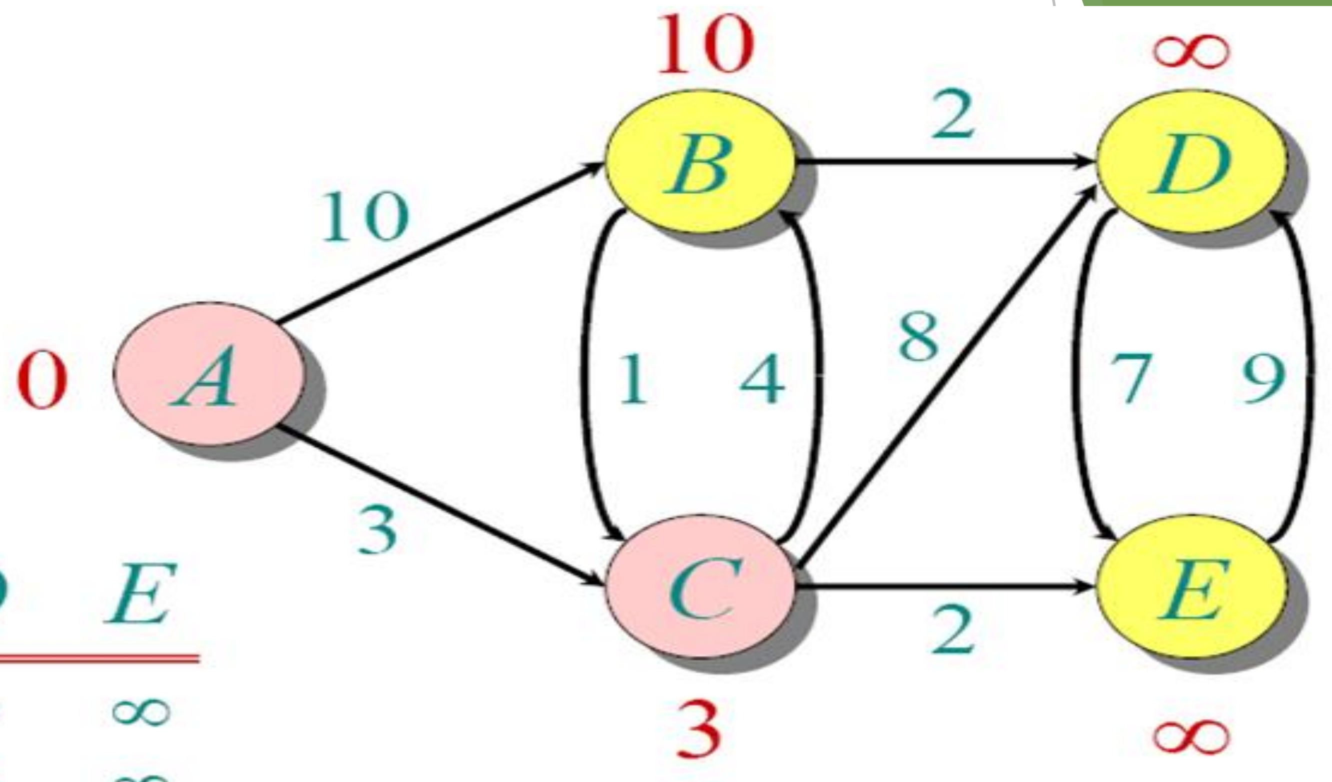
Another Example



Another Example



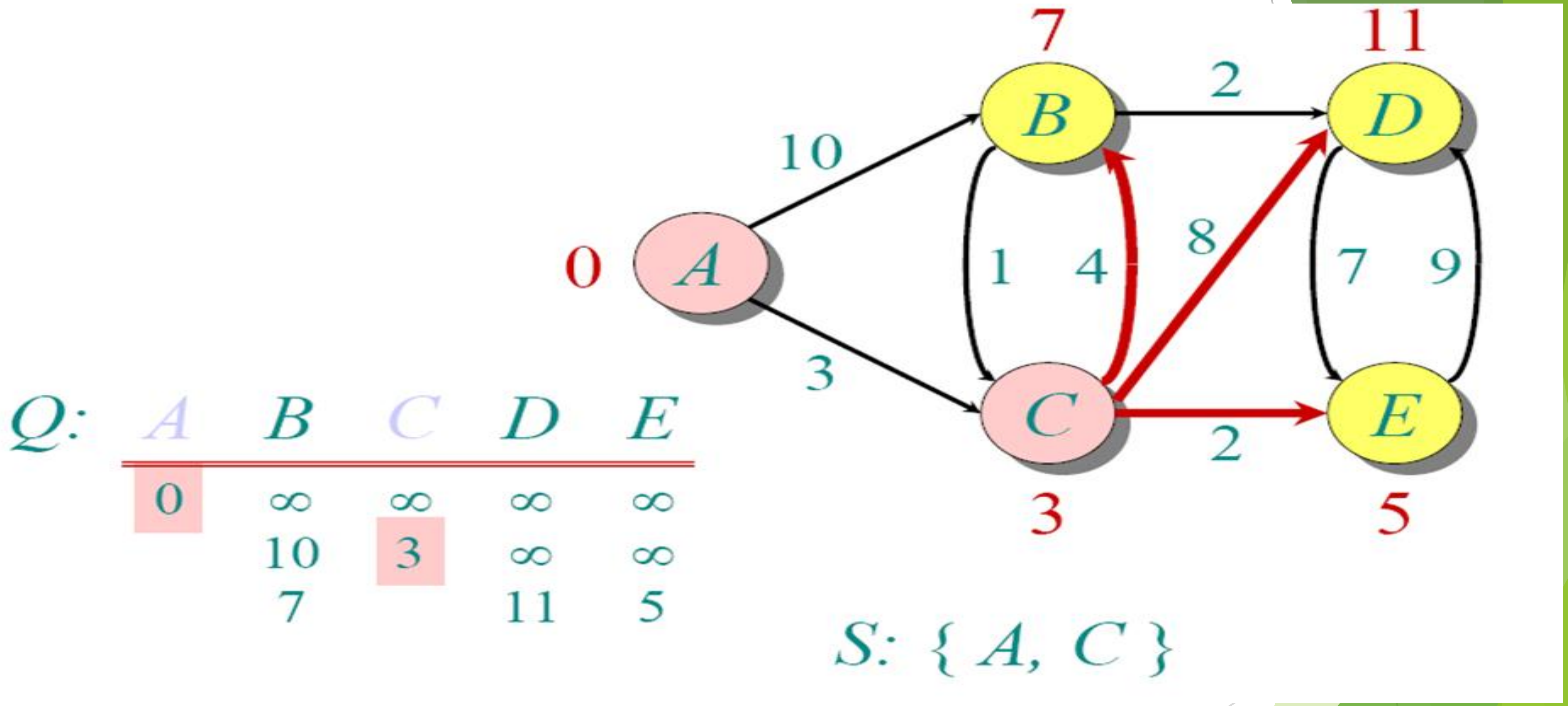
Another Example



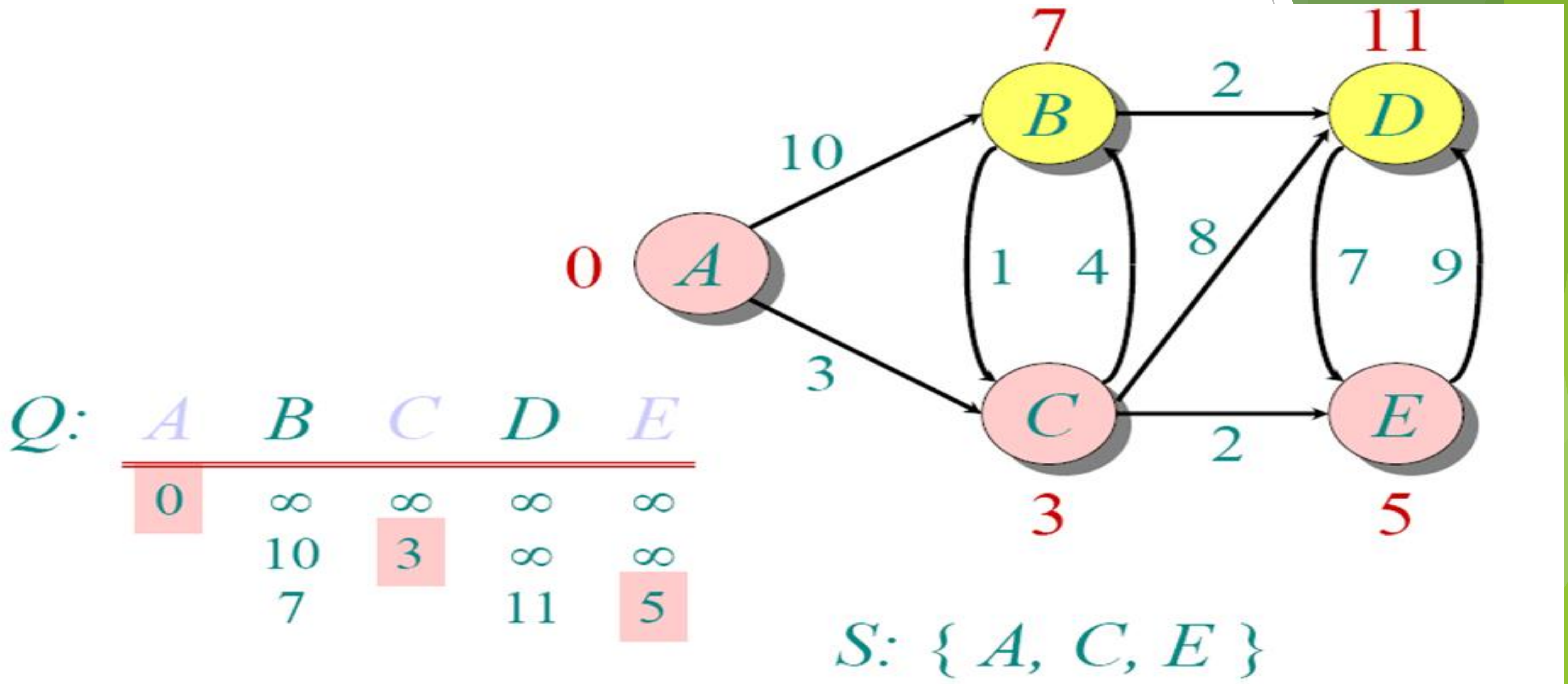
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

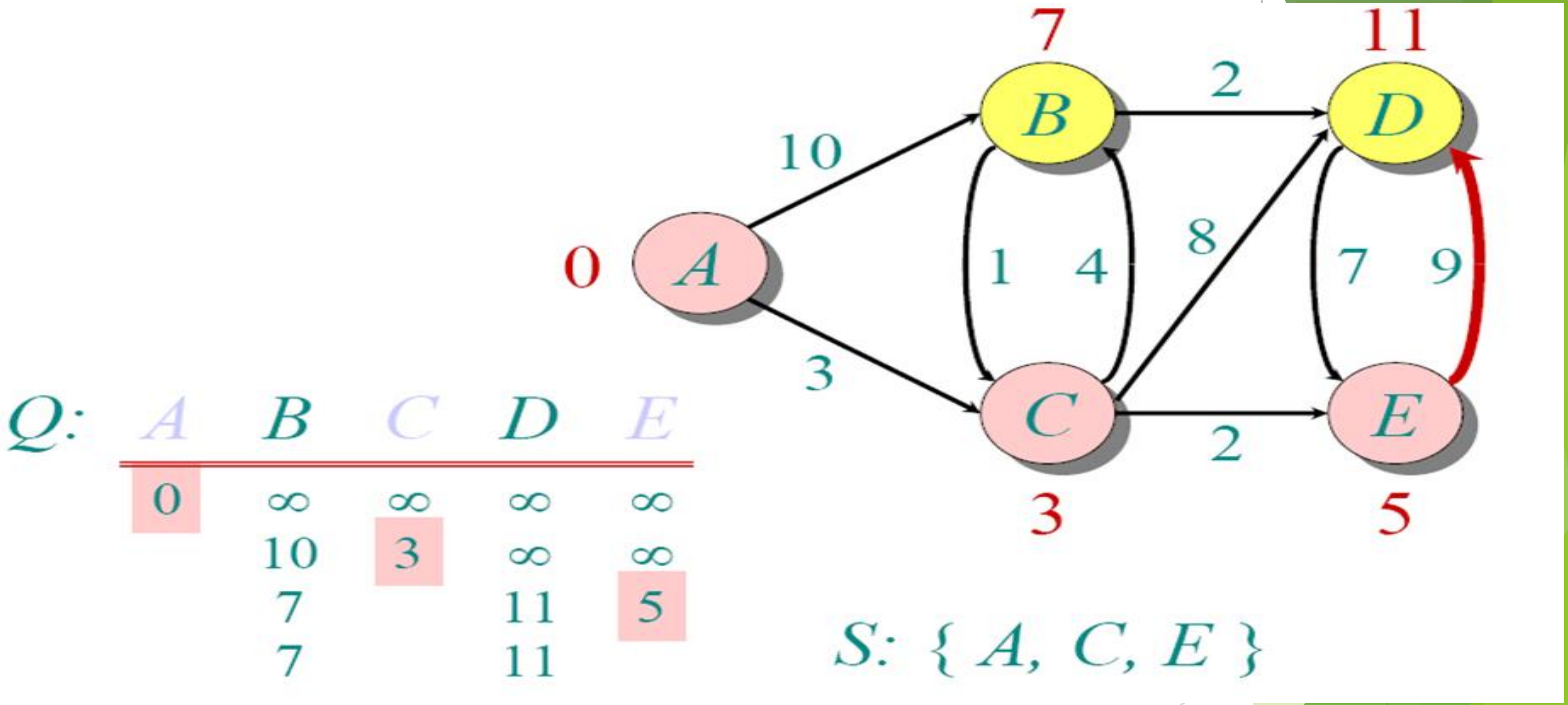
Another Example



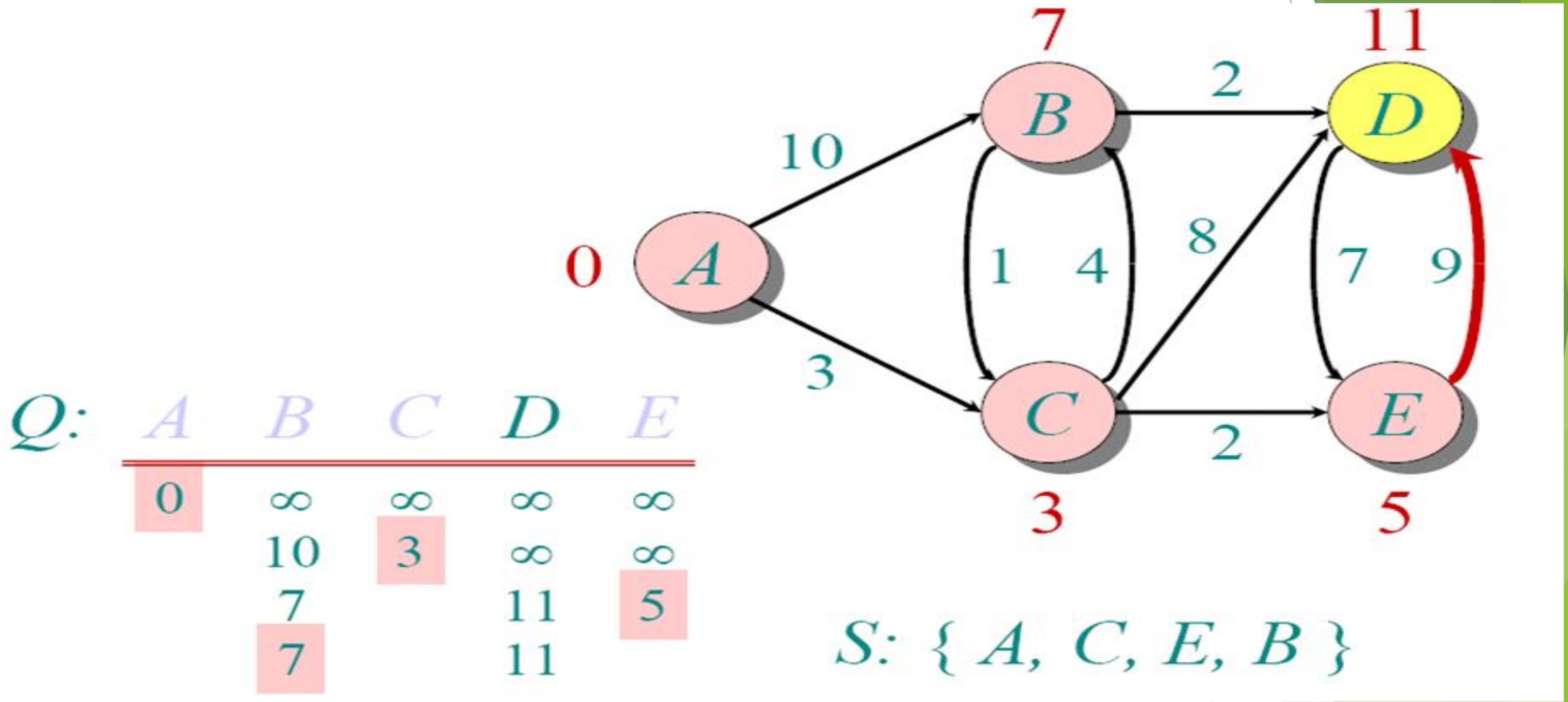
Another Example



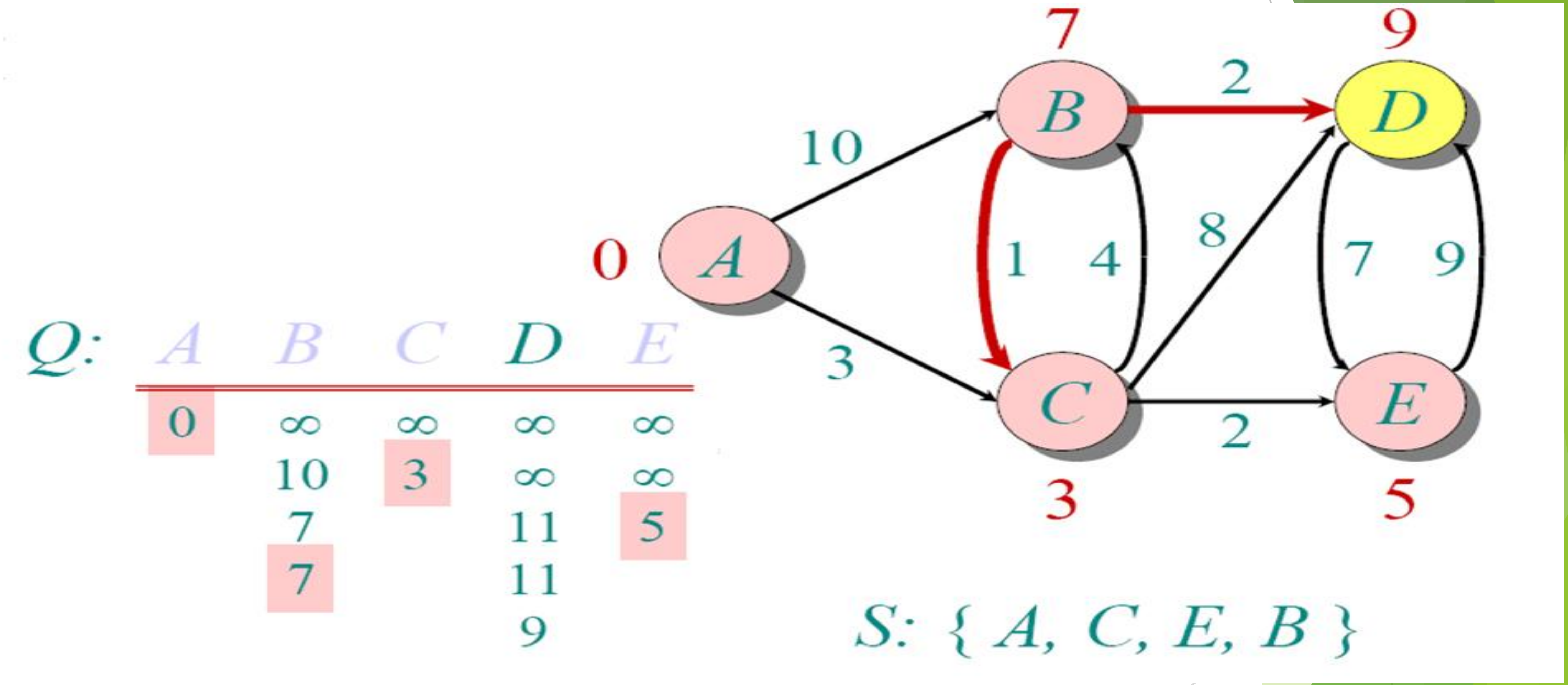
Another Example



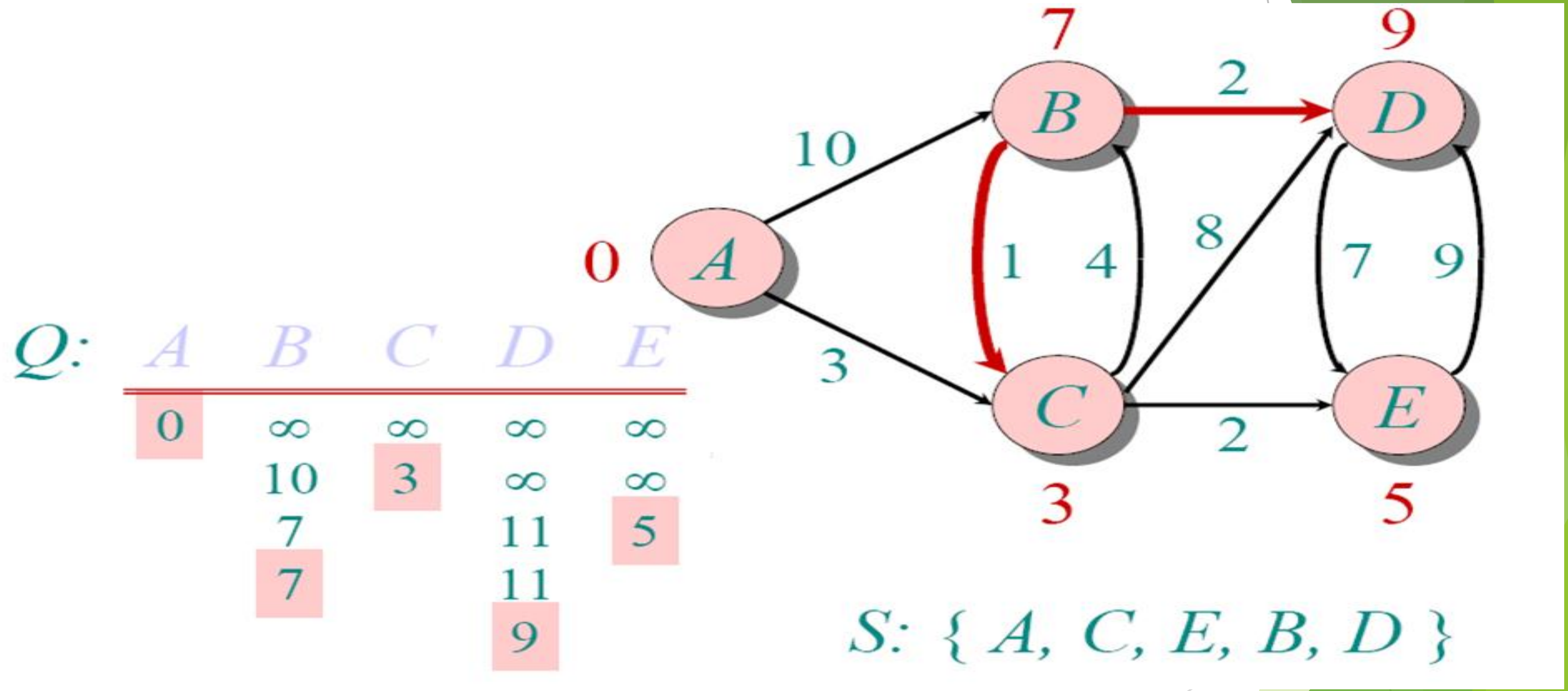
Another Example



Another Example



Another Example



Dijkstra's Pseudo Code

- Graph *G*, weight function *w*, root *s*

DIJKSTRA(G, w, s)

for each $v \in V$
    do $d[v] \leftarrow \infty$
$d[s] \leftarrow 0$
$S \leftarrow \emptyset$ $\triangleright$ Set of discovered nodes
$Q \leftarrow V$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN(Q)
        $S \leftarrow S \cup \{u\}$
        for each $v \in Adj[u]$
            do if $d[v] > d[u] + w(u, v)$
                then $d[v] \leftarrow d[u] + w(u, v)$

relaxing edges

Thanks for your Attention

Q&A