Graph Algorithms

Definitions

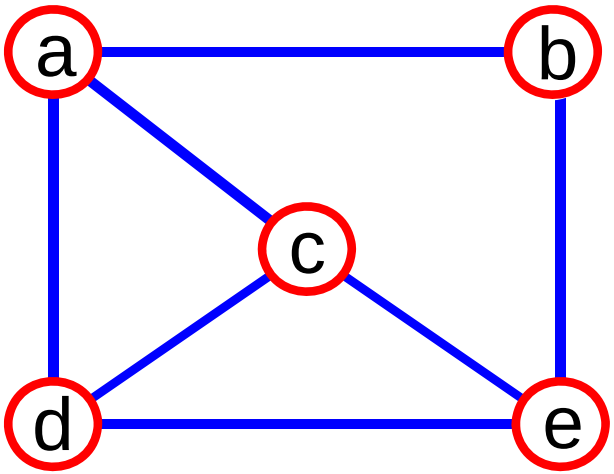
- A graph G = (V,E) is composed of:

  V: Finite, non-empty set of vertices

  E: set of edges connecting the vertices in V

  = Subset of VxV

- An edge e = (u,v) is a pair of vertices.



V= {a, b, c, d, e}

E= {(a,b), (a,c), (a,d), (b,e), (c,d), (c,e), (d,e)}

Definitions

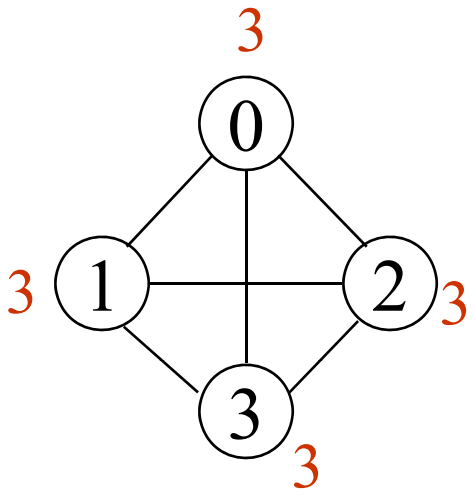
- An **undirected graph** is one in which the pair of vertices in a edge is **unordered**, $(v_0, v_1) = (v_1, v_0)$
- A **directed graph** is one in which each edge is a **directed** pair of vertices, $\langle v_0, v_1 \rangle$!= $\langle v_1, v_0 \rangle$
- If (v_0, v_1) is an edge in an undirected graph,
  - v_0 and v_1 are *adjacent*
  - The edge (v_0, v_1) is incident on vertices v_0 and v_1
- If $\langle v_0, v_1 \rangle$ is an edge in a directed graph
  - v_0 is *adjacent to* v_1, and v_1 is *adjacent from* v_0

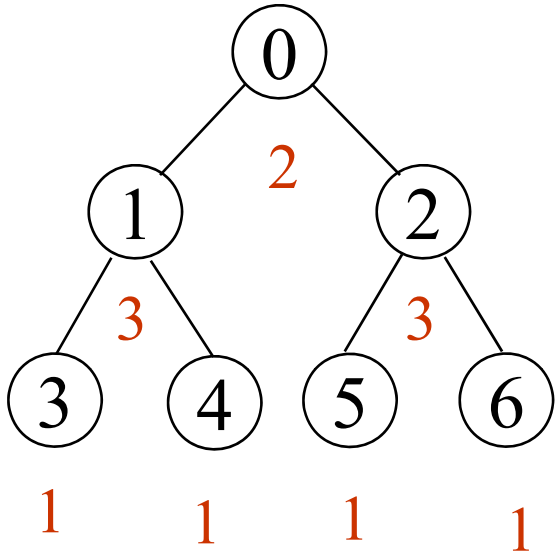
Definitions

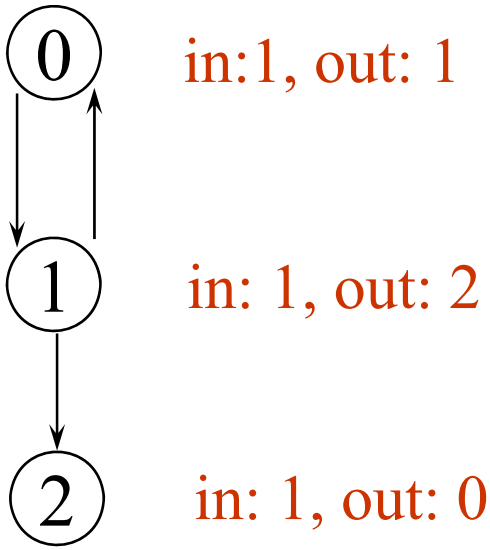
- The degree of a vertex is the number of edges incident to that vertex
- For directed graph,
  - the in-degree of a vertex *v* is the number of edges that have *v* as the end vertex
  - the out-degree of a vertex *v* is the number of edges that have *v* as the start vertex
  - if d_i is the degree of a vertex *i* in a graph *G* with *n* vertices and *e* edges, the number of edges is

$$e = \left(\sum_{0}^{n-1} d_i \right) / 2$$

Definitions

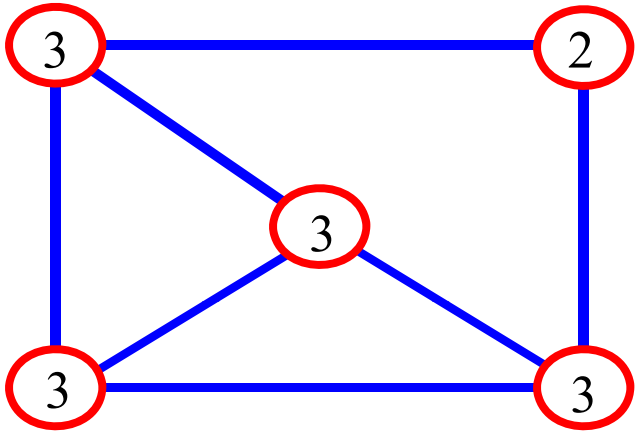


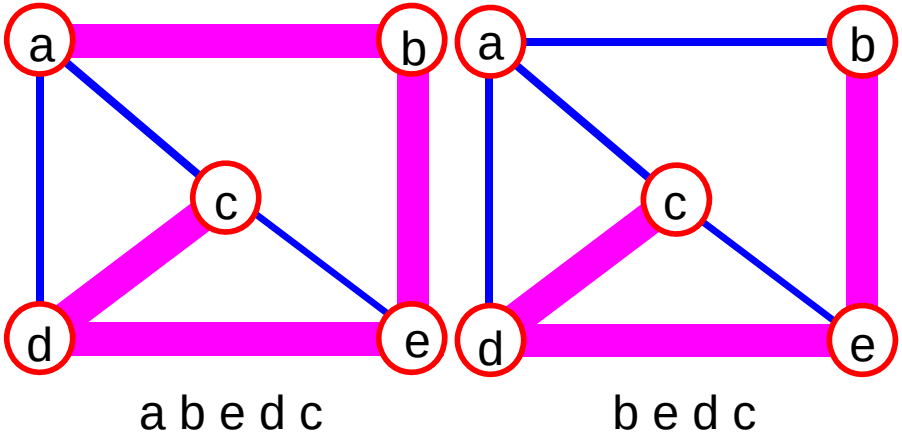




Definitions

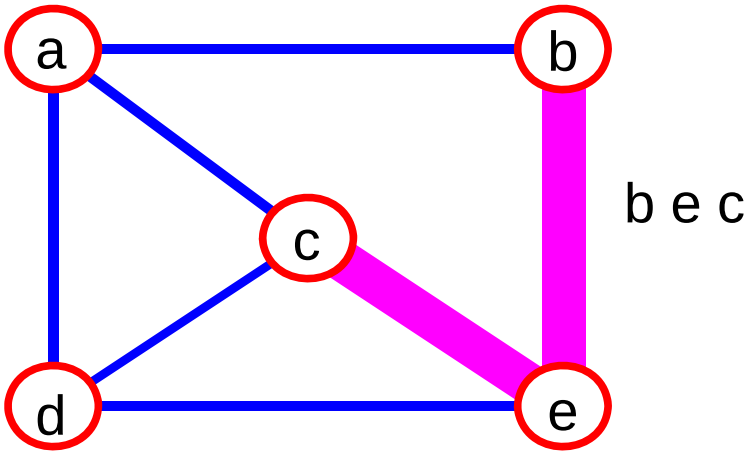
- path: sequence of vertices $v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.



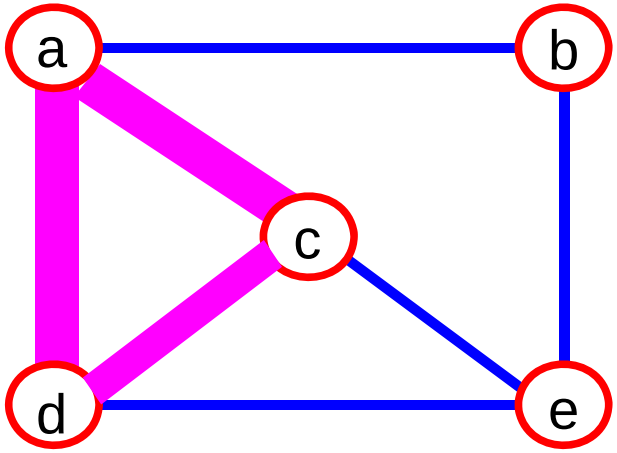


Definitions

- simple path: no repeated vertices
- cycle: simple path, except that the last vertex is the same as the first vertex



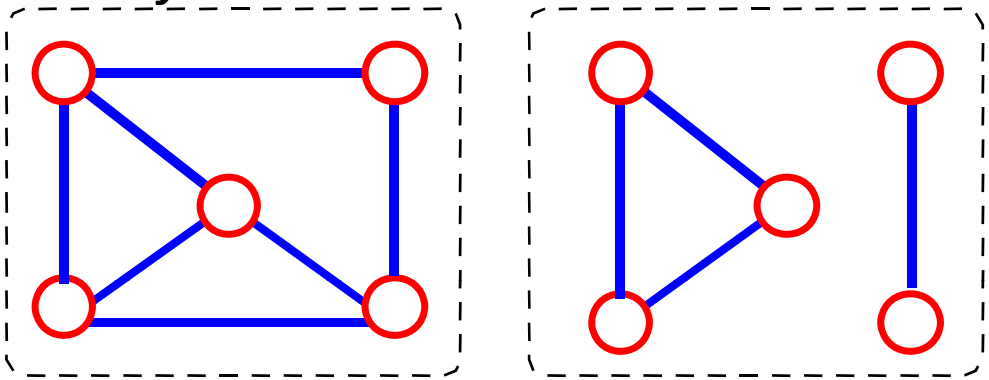
b e c



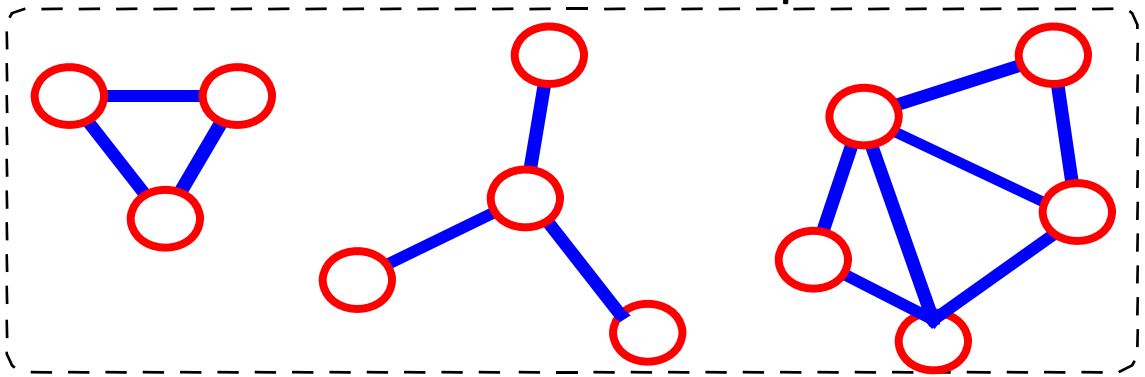
a d c a

Definitions

- connected graph: any two vertices are connected by some path

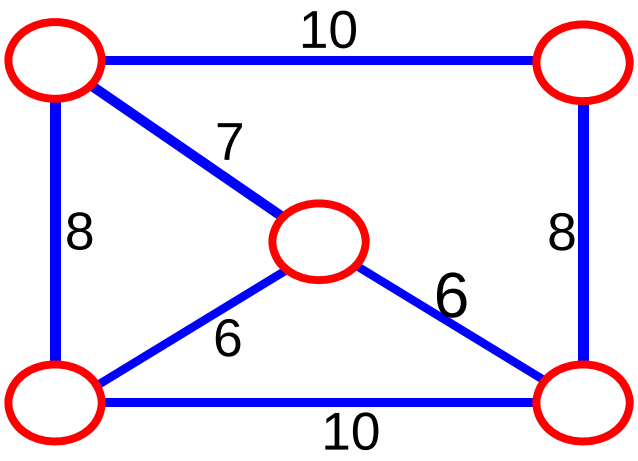


- subgraph: subset of vertices and edges forming a graph
- connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.



Definitions

- A *weighted graph* associates weights with the edges
  - e.g., a road map: edges might be weighted with distance



Definitions

- We will typically express running times in terms of |E| and |V|
  - If $|E| \approx |V|^2$ the graph is *dense*
  - If $|E| \approx |V|$ the graph is *sparse*

- If you know you are dealing with dense or sparse graphs, different data structures may make sense

Graph Representation

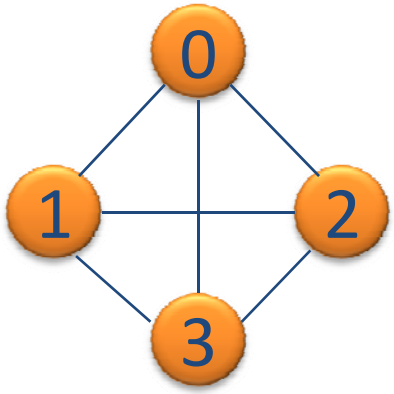
- Adjacency Matrix
- Adjacency Lists
- Incidence Matrix

Adjacency Matrix

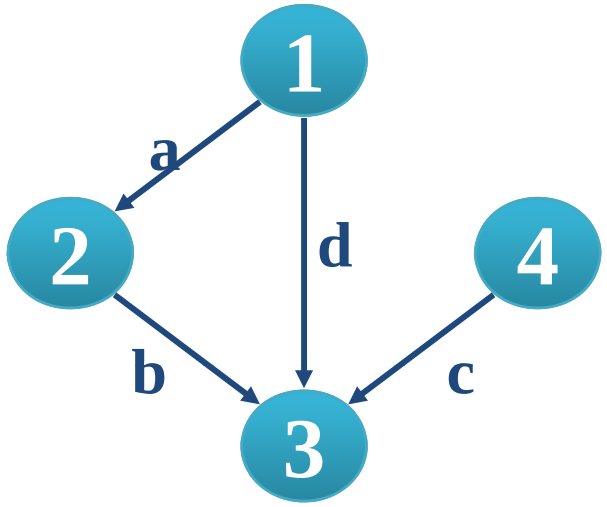
- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional nxn array, say adj_mat
  - ✓ If the edge (v_i, v_j) is in E(G), adj_mat[i][j]=1
  - ✓ If there is no such edge in E(G), adj_mat[i][j]=0

- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a directed graph need not be symmetric

Adjacency Matrix



A	0	1	2	3
0	0	1	1	1
1	1	0	1	1
2	1	1	0	1
3	1	1	1	0



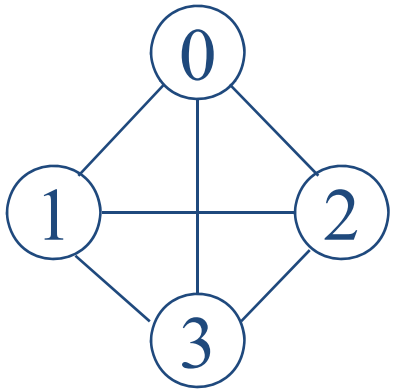
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

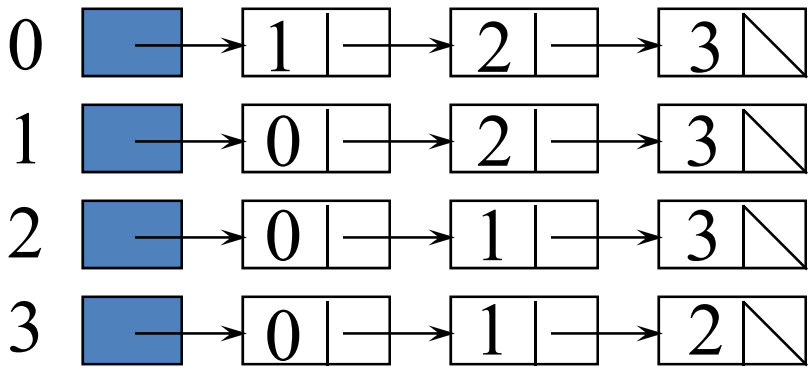
Adjacency Matrix

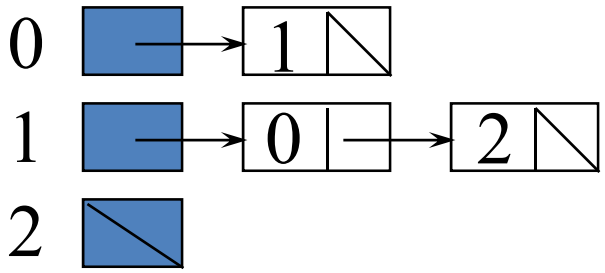
- From the adjacency matrix, to determine the connection of vertices is easy
  - The degree of a vertex i is the number of 1's in i^{th} row
  - For a directed graph, the number of 1's in i^{th} row is the out_degree, while the number of 1's in i^{th} column in_degree.
- **Time:** to list all vertices adjacent to u: O(V).
- **Time:** to determine if (u, v) $\in$ E: O(1).
- **Space:** O(V^2).
  - Not memory efficient for large graphs.
- ***Parallel edges*** cannot be represented
- Can store weights instead of bits for weighted graph.

Adjacency Lists

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to v







Adjacency Lists

```
typedef struct adjvertex
{
        int vertex;
        struct node *next;
}adjvertex;
```

```
typedef struct graph
{
        int no_of_Vertices;
        adjvertex *adjlist [100];
}graph;
```

Adjacency Lists

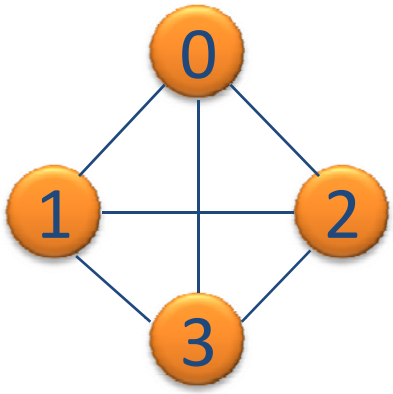
- How much storage is required?
  - The *degree* of a vertex *v* = # incident edges
    - Directed graphs have in-degree, out-degree
  - For directed graphs, # of items in adjacency lists is
    $$\Sigma \text{ out-degree}(v) = |E|$$
    takes O(V + E) storage
  - For undirected graphs, # items in adjacency lists is
    $$\Sigma \text{ degree}(v) = 2\ |E|$$
    also O(V + E) storage
- So: Adjacency lists take O(V+E) storage

Incidence Matrix

- Consider a matrix A = (a_{ij}), rows corresponds to vertices, column corresponds to edges.
- For undirected graph:
  - a_{ij}= 1 if e_j is incedent to v_i
    = 0 otherwise
- For directed graph:
  - a_{ij}= 1 if e_j is incedent out of v_i
    = -1 if e_j is incedent into v_i
    = 0 otherwise

Bhaskar Sardar, Information Technology Department, *Jadavpur University, India*

Incidence Matrix



A	(0,1)	(0, 2)	(0, 3)	(1, 2)	(1, 3)	(2, 3)
0	1	1	1	0	0	0
1	1	0	0	1	1	0
2	0	1	0	1	0	1
3	0	0	1	0	1	1

Graph Traversal

- Given: a graph G = (V, E), directed or undirected
- Goal: systematically explore every vertex and every edge
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree

Graph Traversal

- Depth First Search
  - Once a possible path is found, continue the search until the end of the path
  - Think of a Stack
- Breadth First Search
  - Start several paths at a time, and advance in each one step at a time
  - Think of a Queue

Depth First Search

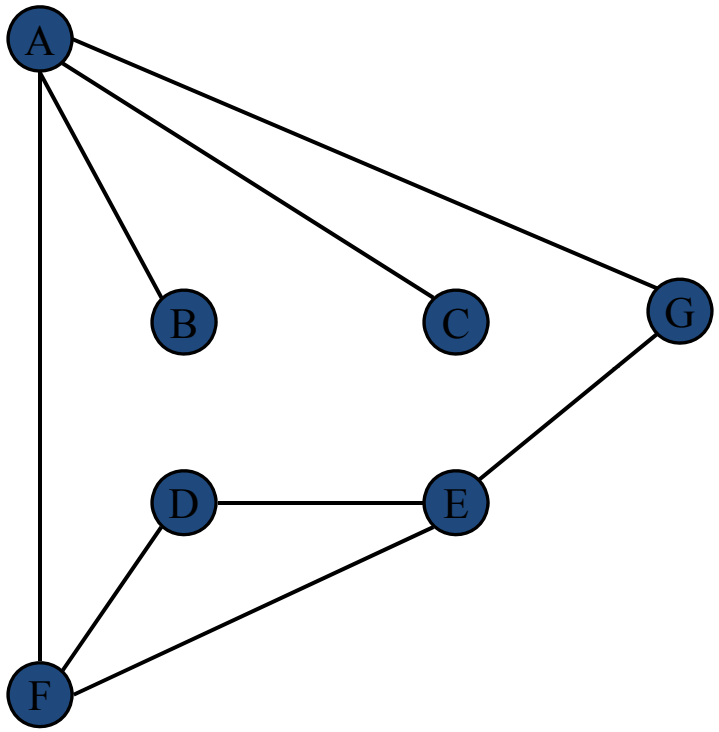
- We start at vertex ***s***, and mark ***s*** "visited". Next we label ***s*** as our current vertex called ***u***.
- Now we travel along an arbitrary edge (***u***, ***v***).
- If edge (***u***, ***v***) leads us to an already visited vertex ***v*** we return to ***u***.
- If vertex ***v*** is unvisited, we move to ***v***, mark ***v*** "visited", set ***v*** as our current vertex, and repeat the previous steps.

Depth First Search

```
void DFS (int start)
{
        int v;
        adjvertex *adj;
        visited [start] = 1;
        printf ("%d", start);
        adj = g→adjlist [start];
        while (adj ! = NULL)
        {
                v = adj→vertex;
                if (! visited [v])
                        DFS (v);
                adj = adj →next;
        }
}
```

Total running time: $O(2e)=O(e)$

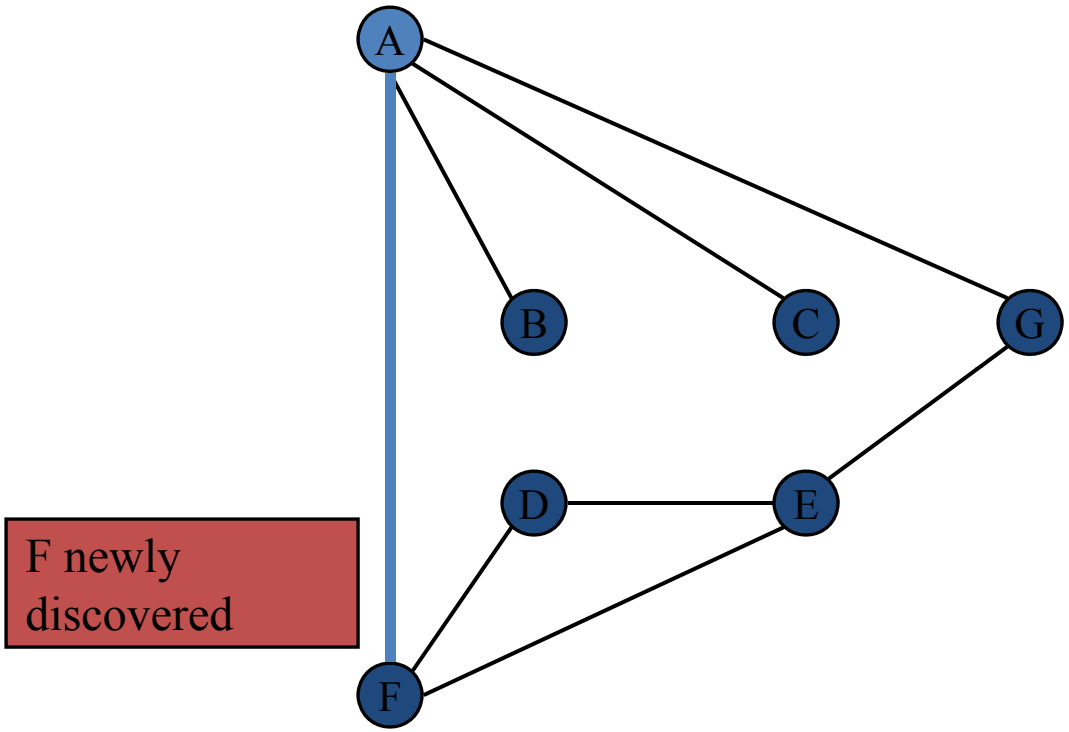
Depth First Search



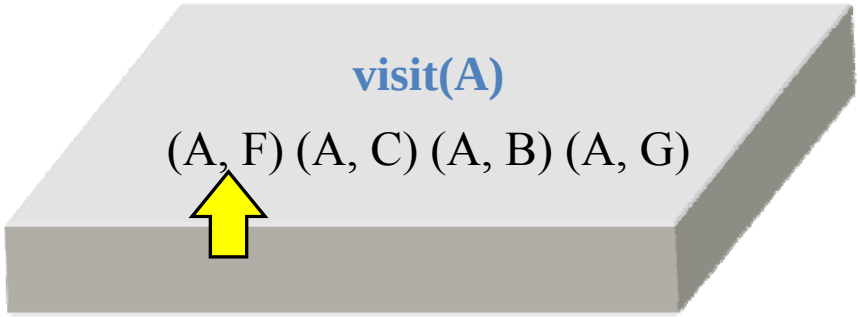
Adjacency Lists

A: F C B G
B: A
C: A
D: F E
E: G F D
F: A E D
G: E A

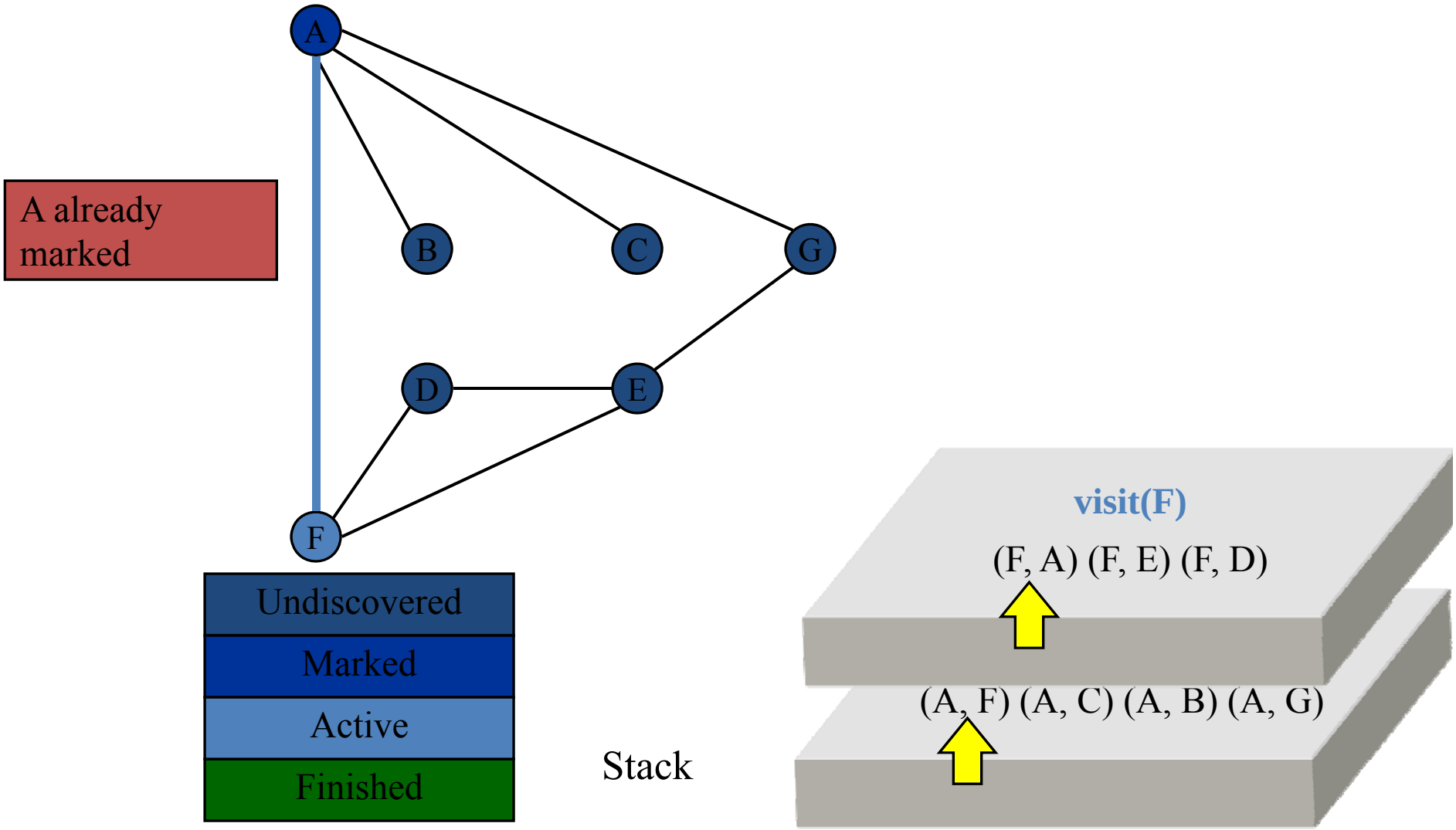
Depth First Search



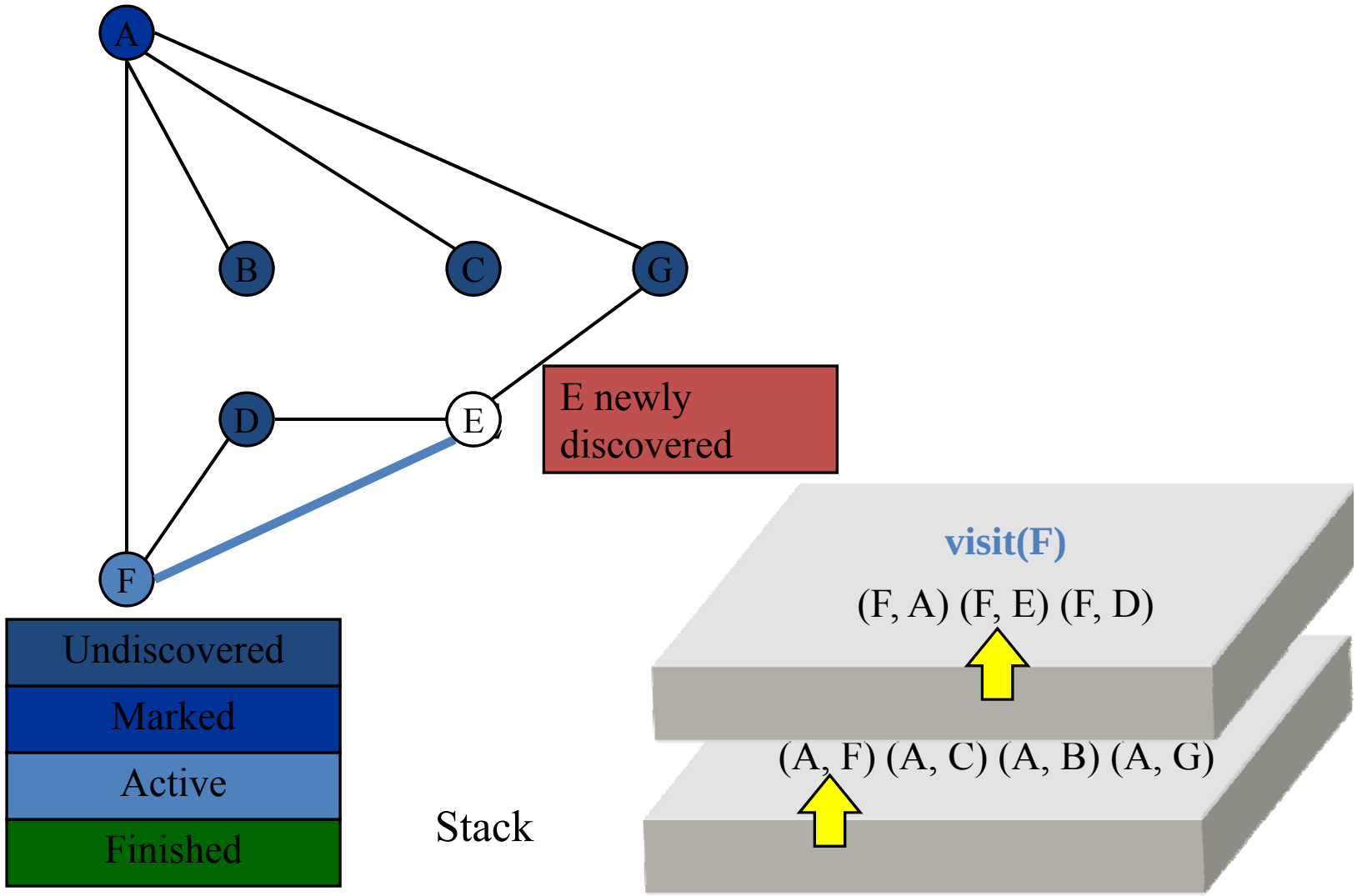
Stack



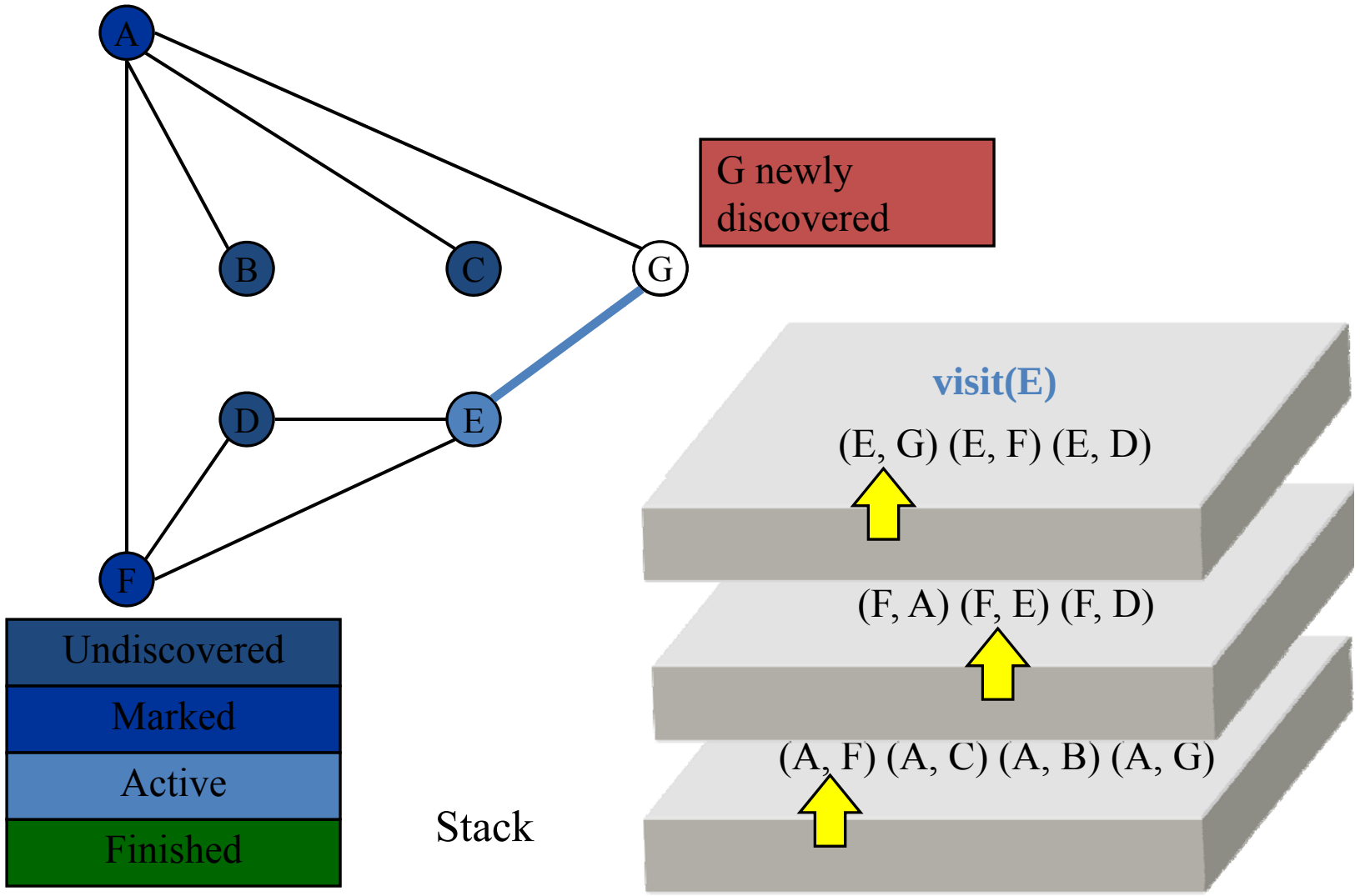
Depth First Search



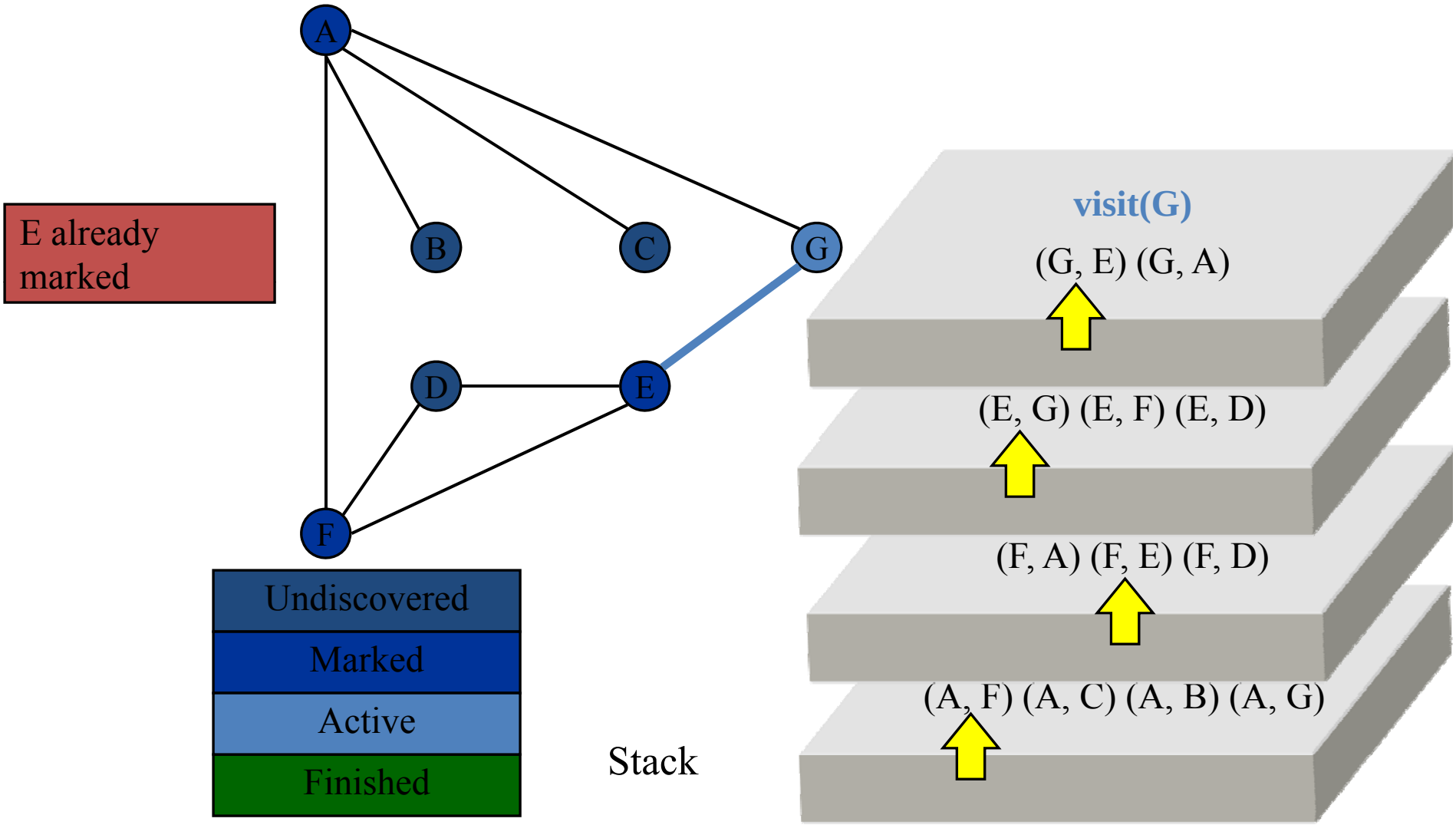
Depth First Search



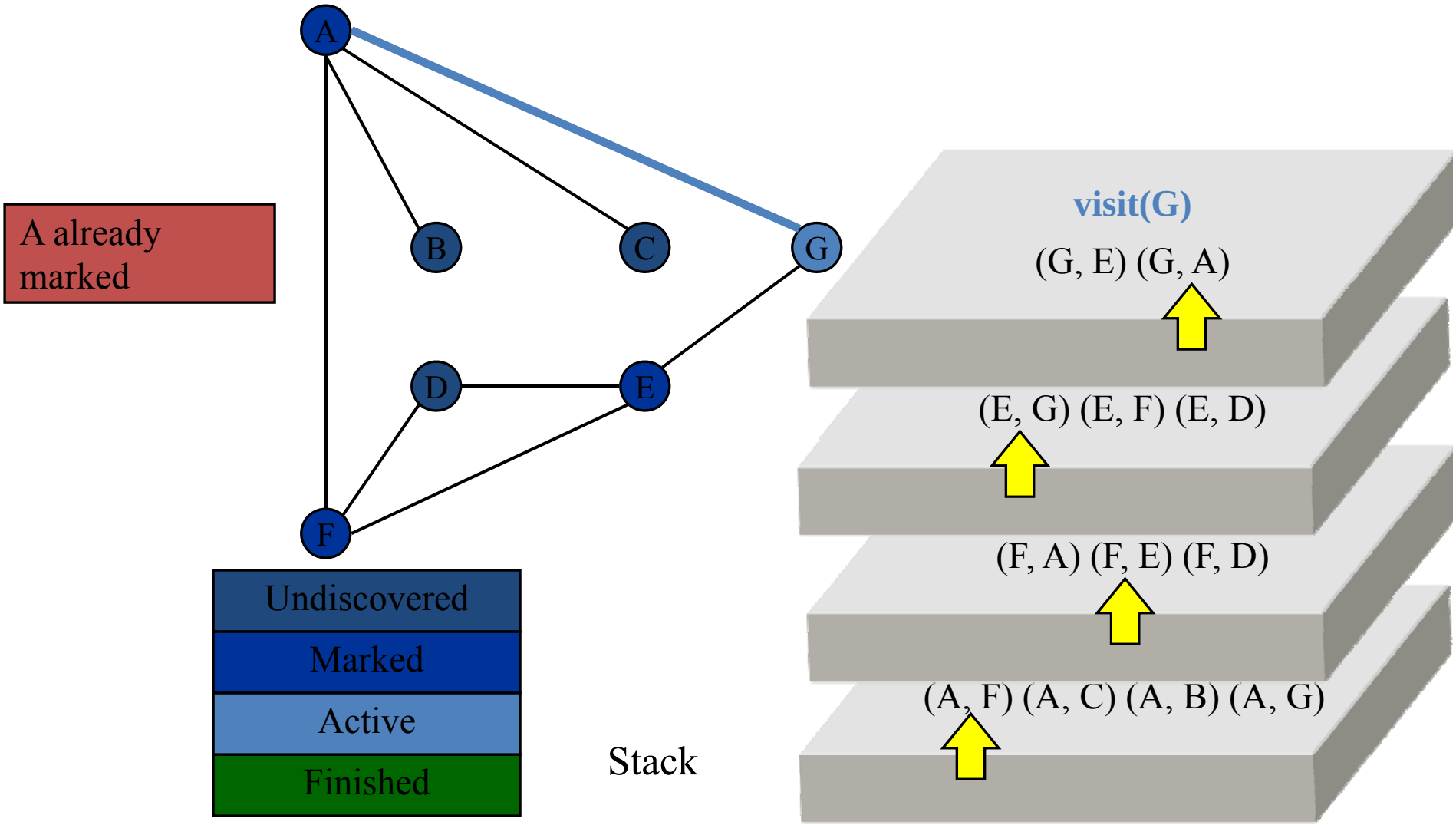
Depth First Search



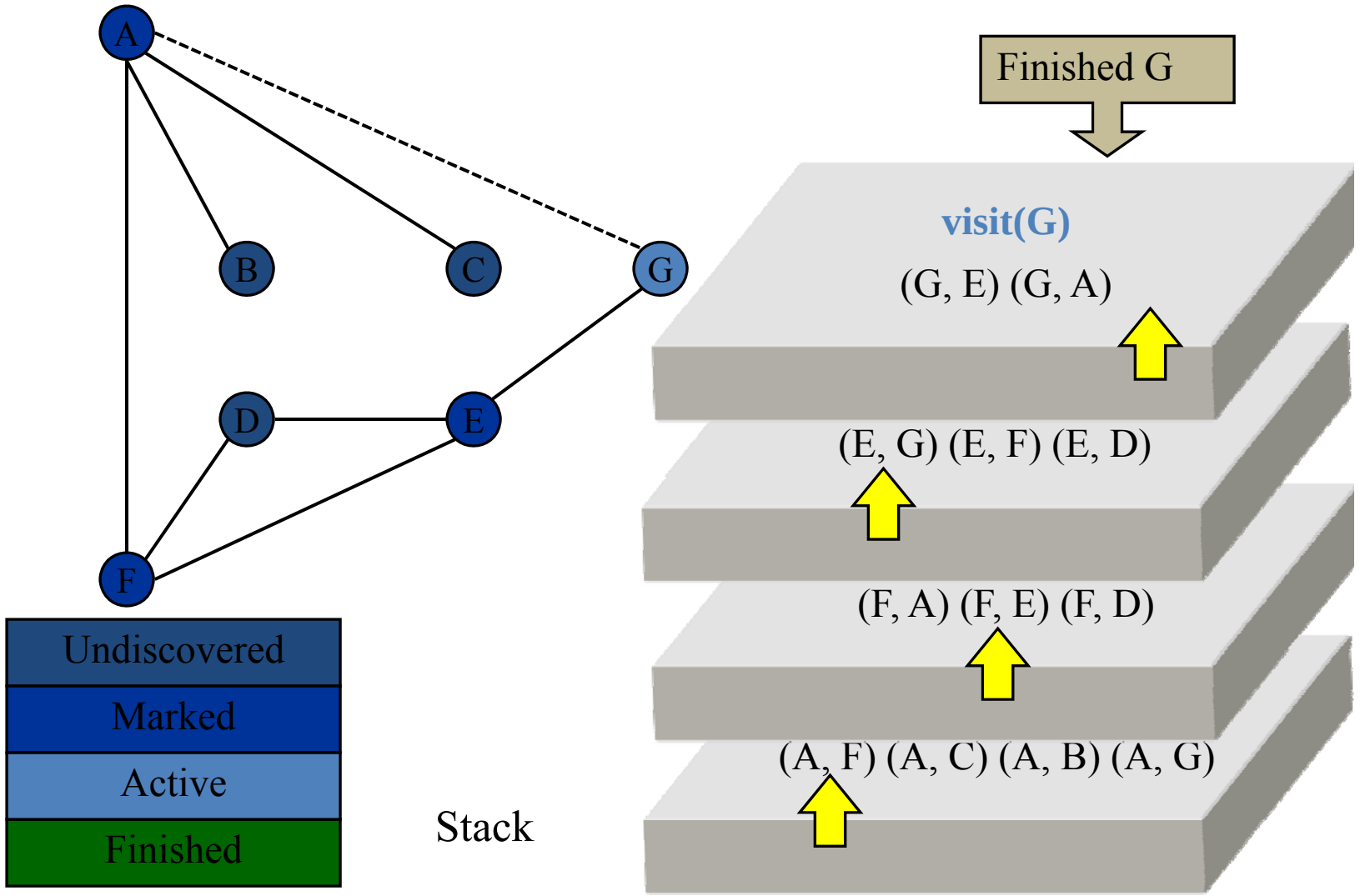
Depth First Search



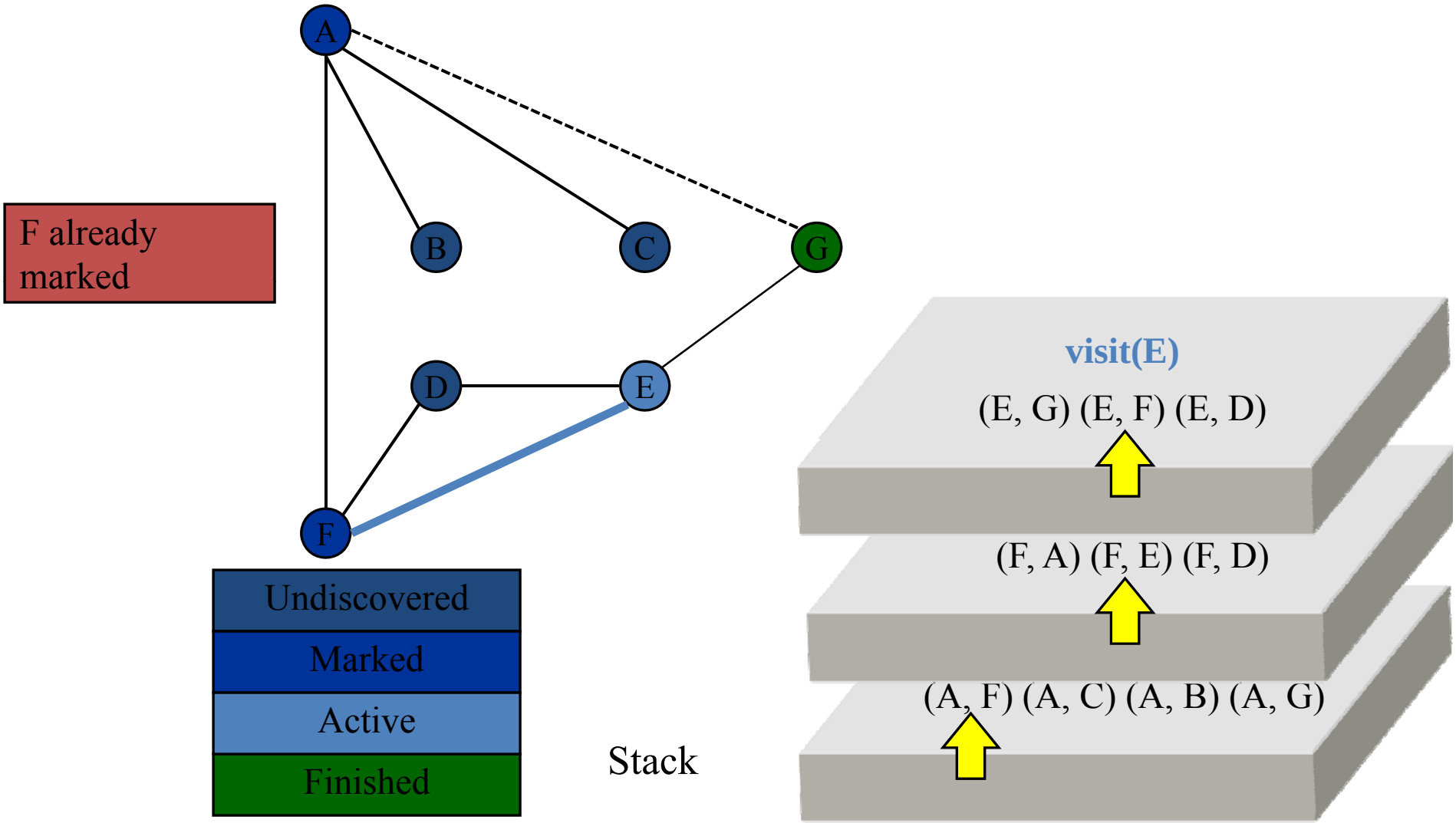
Depth First Search



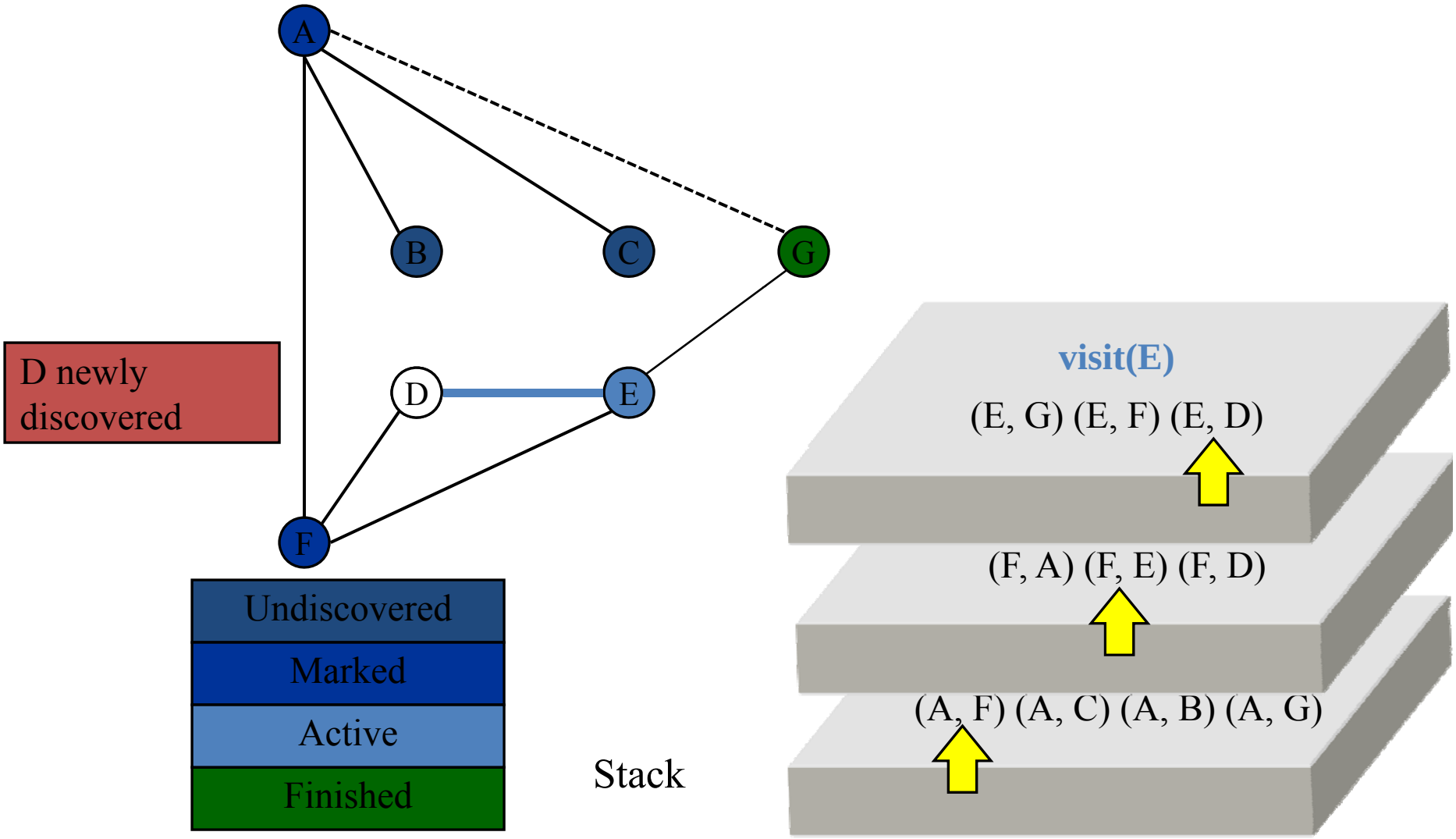
Depth First Search



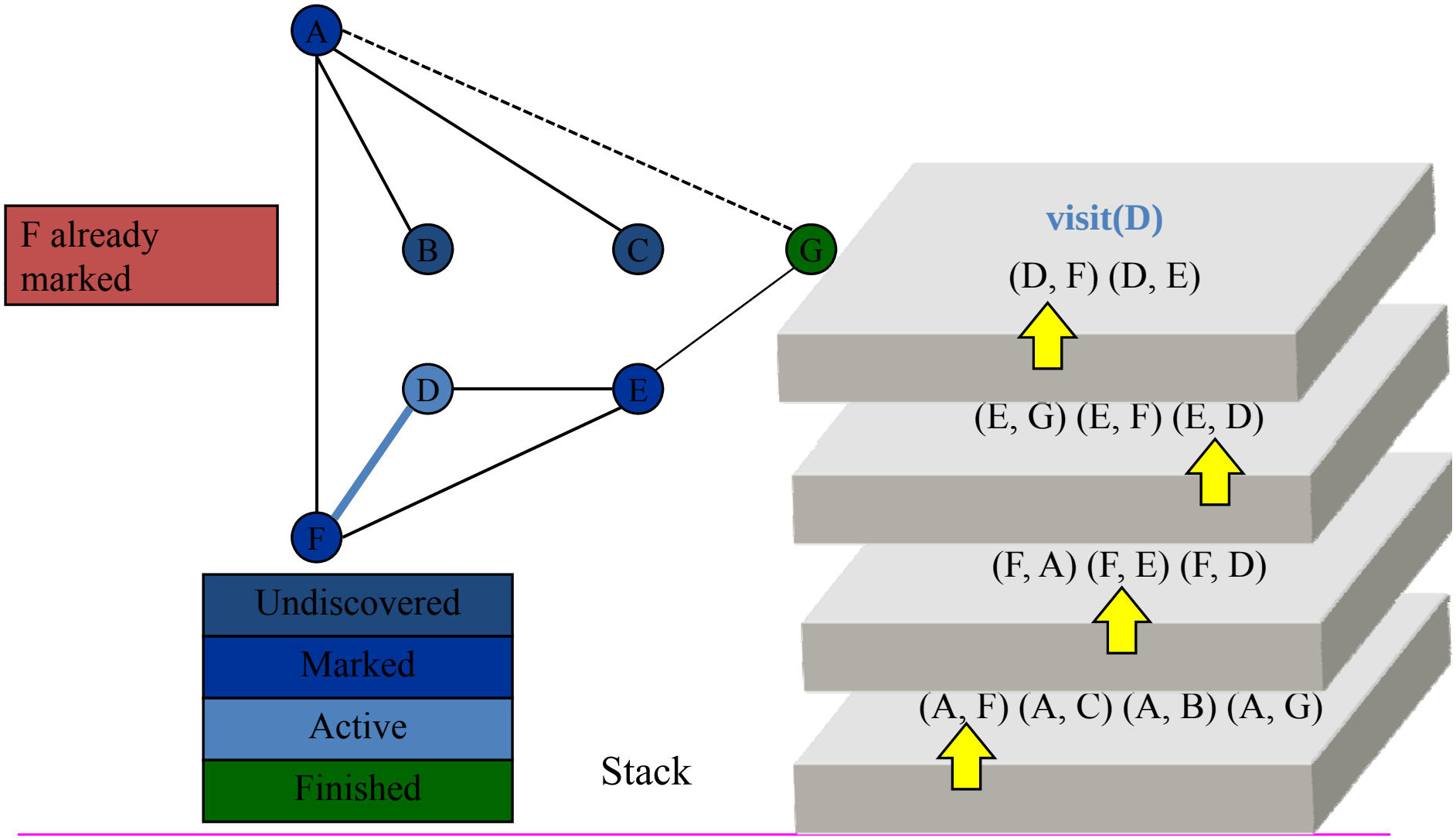
Depth First Search



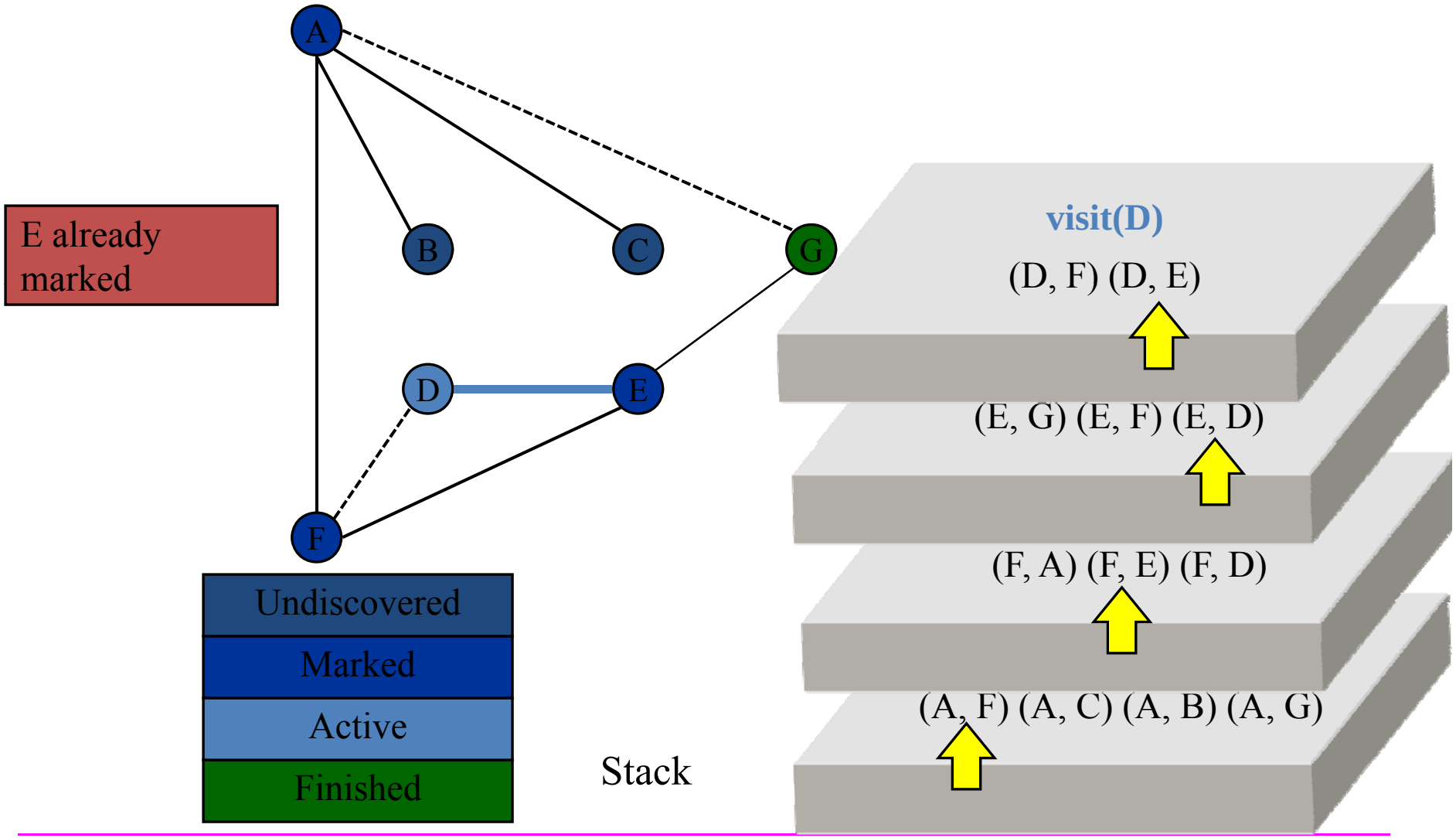
Depth First Search



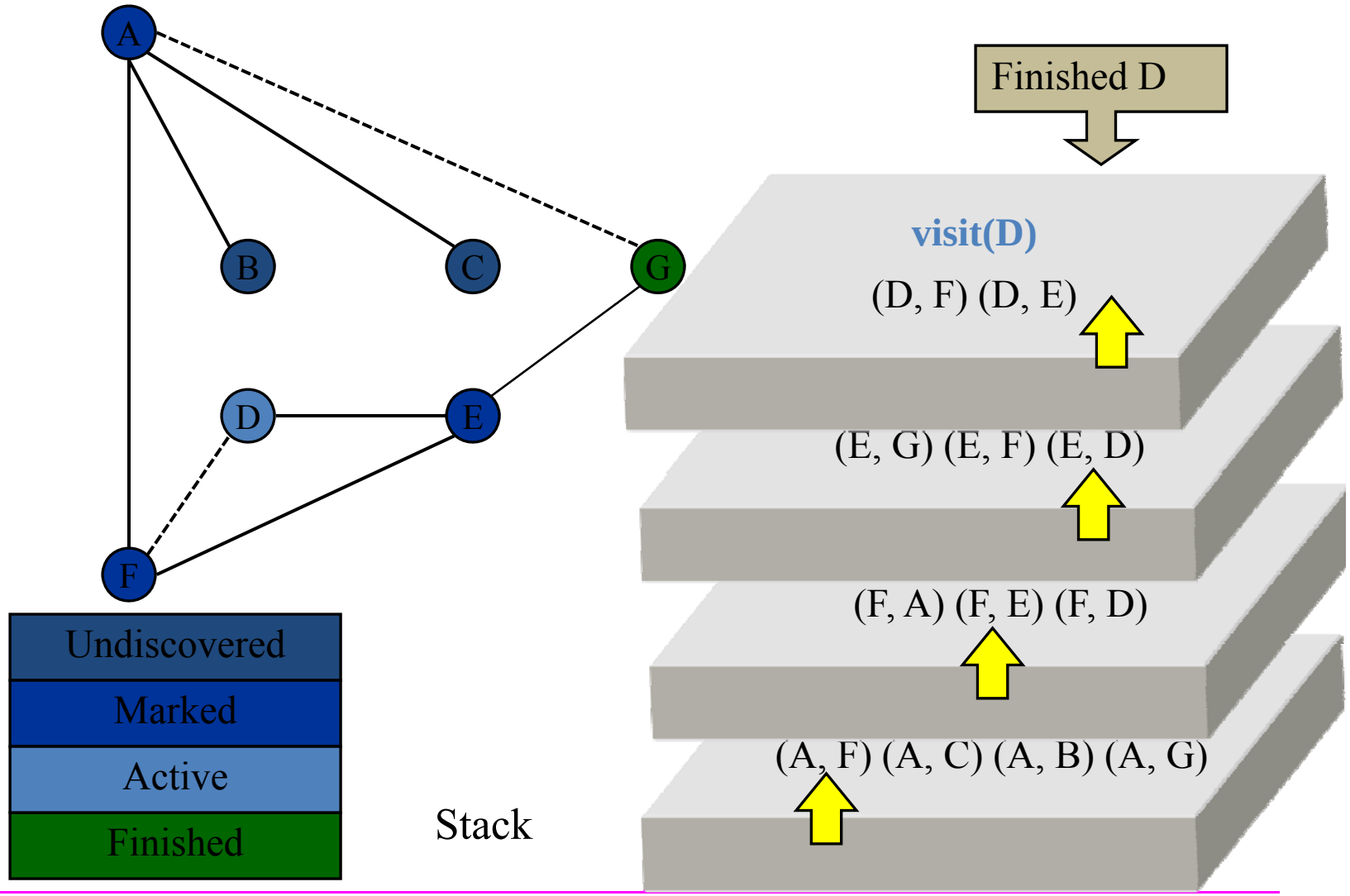
Depth First Search



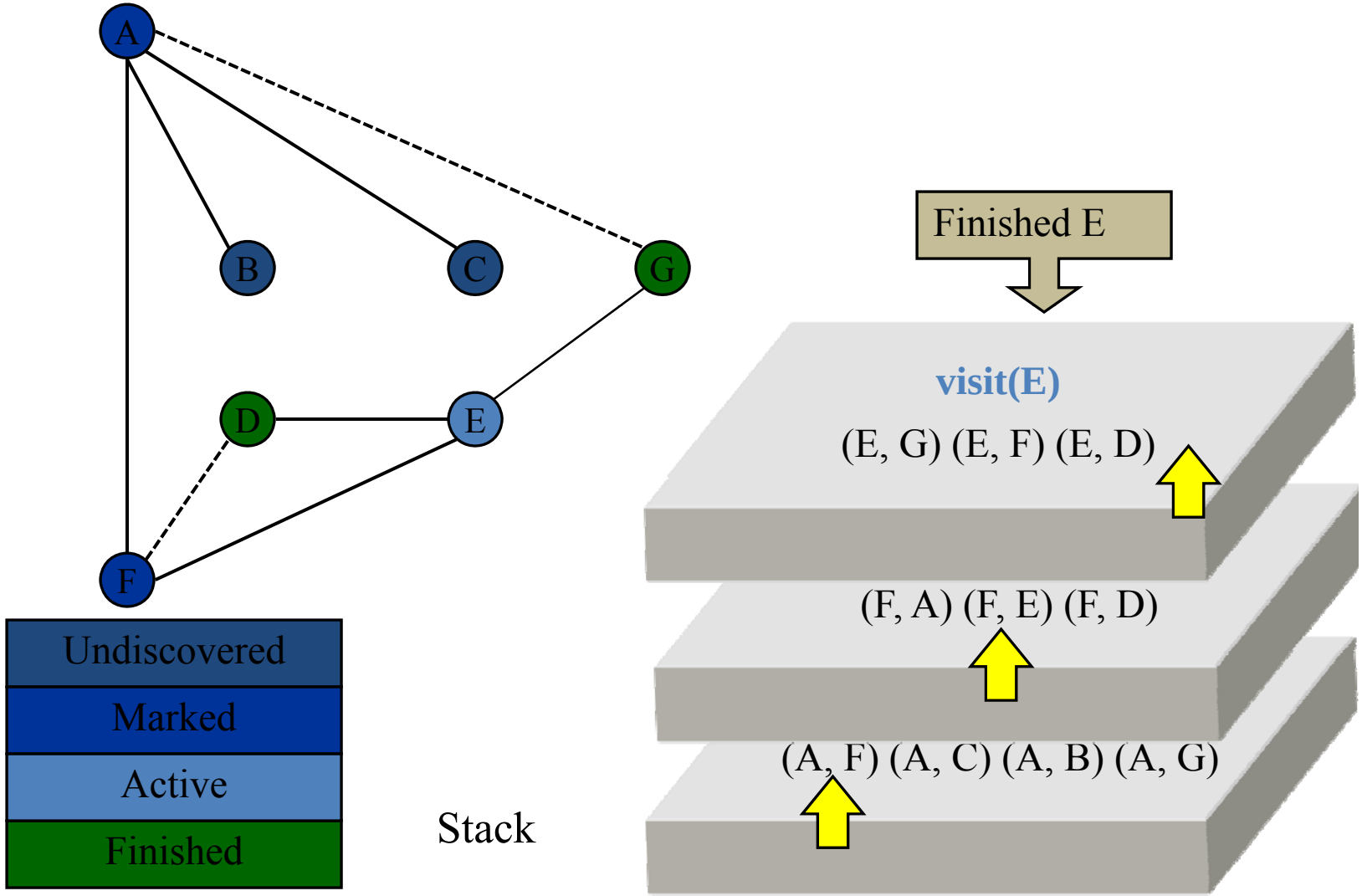
Depth First Search



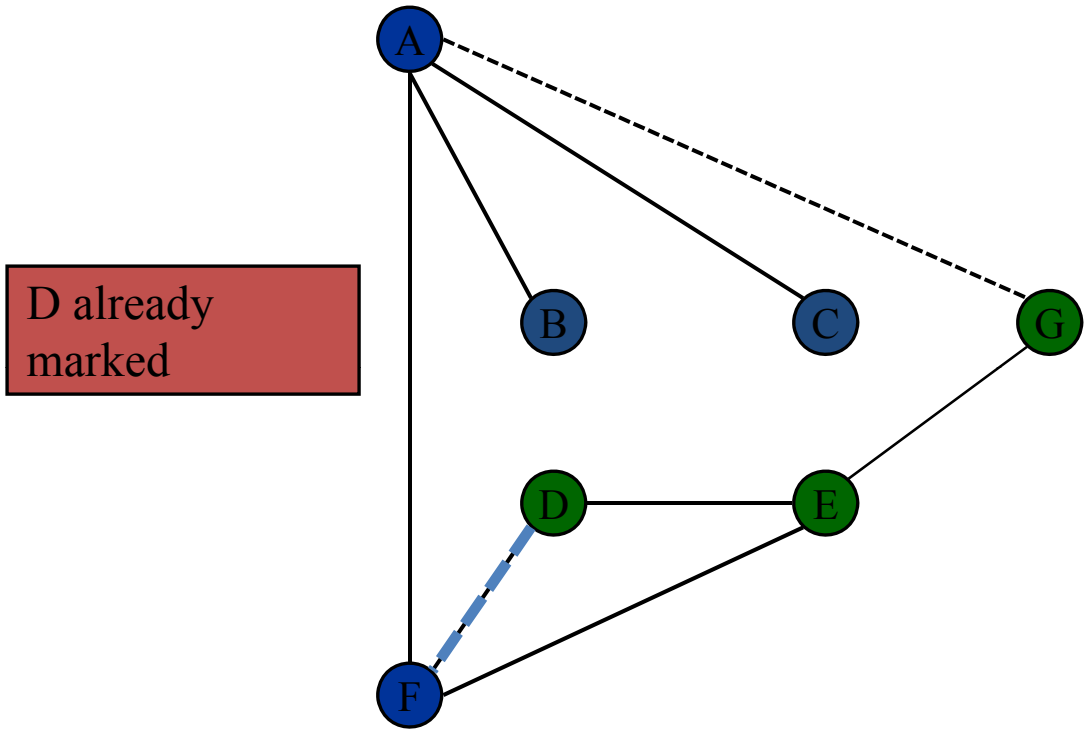
Depth First Search



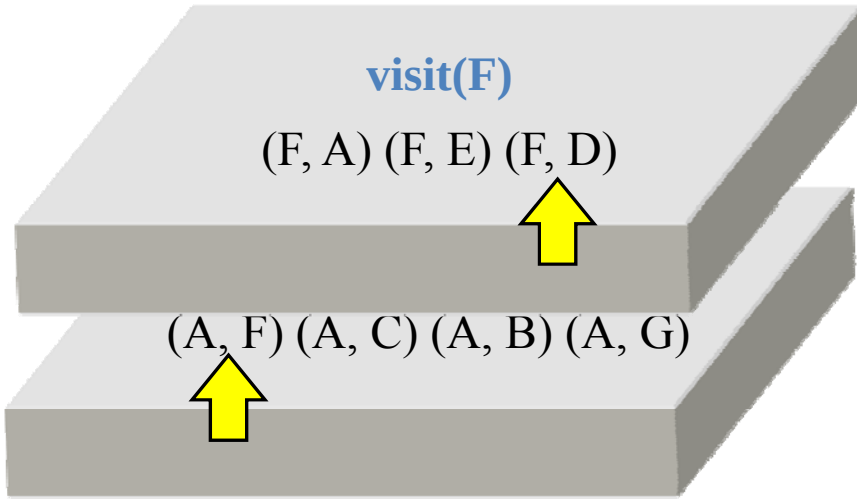
Depth First Search



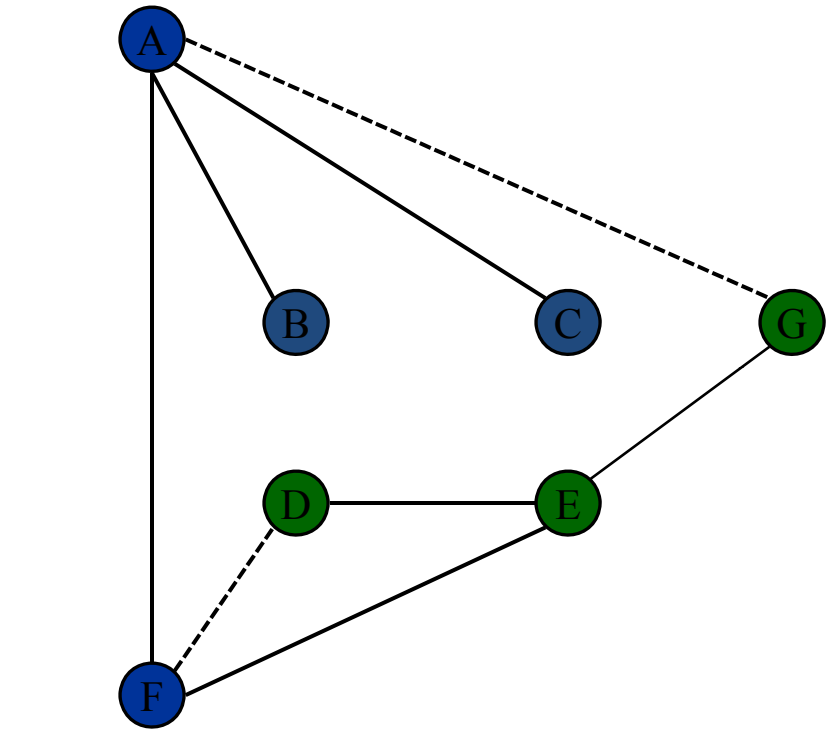
Depth First Search



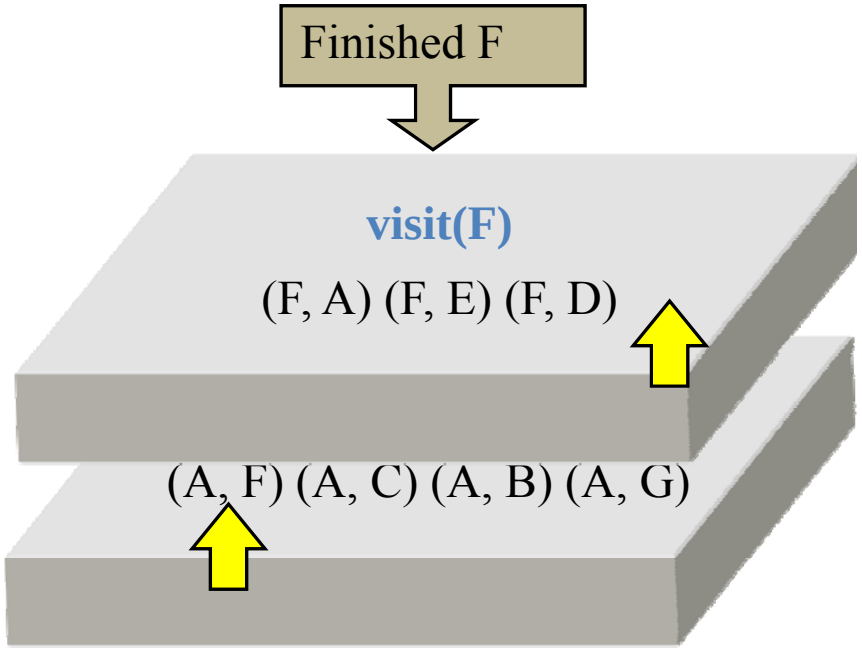
Stack



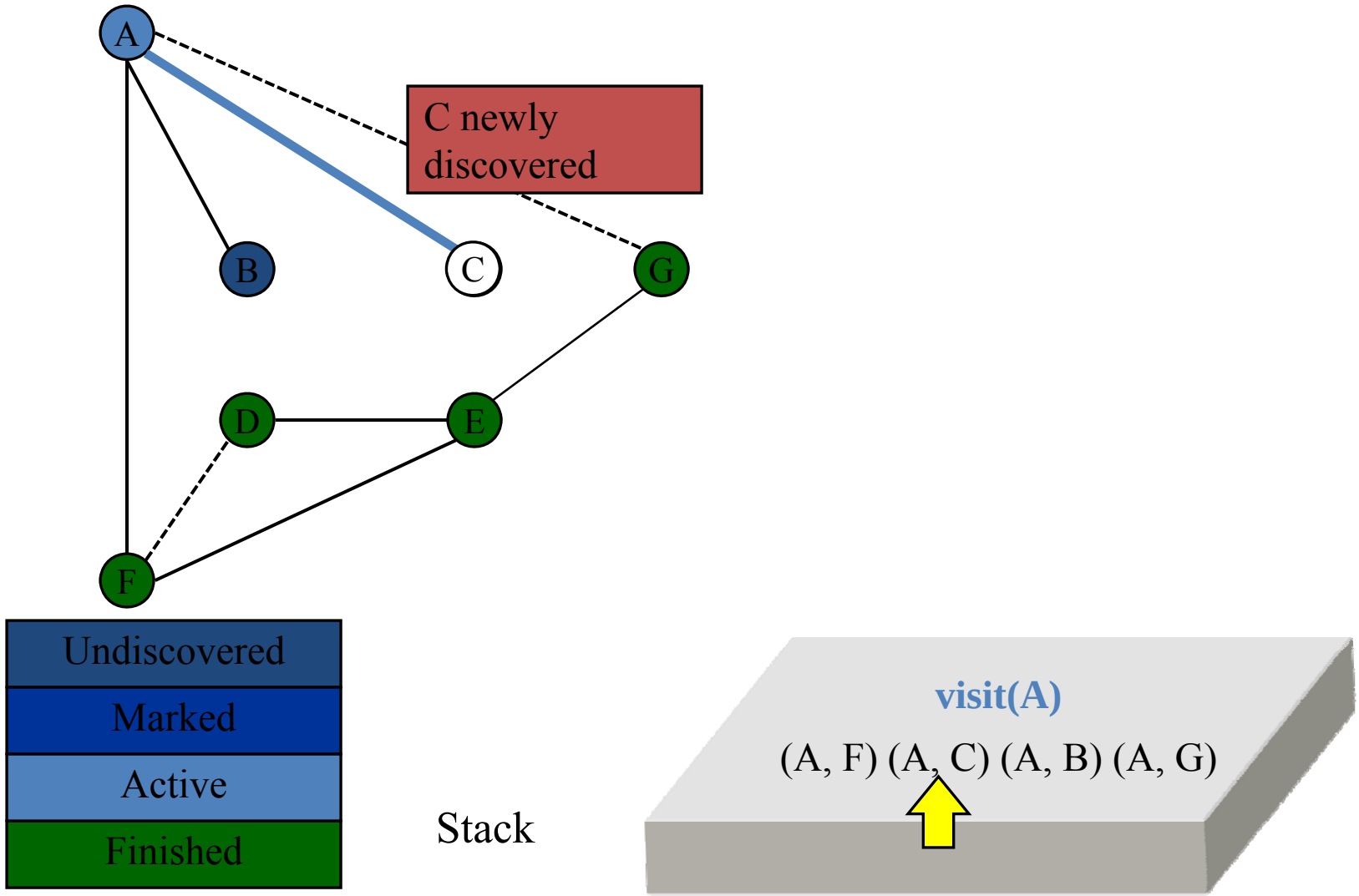
Depth First Search



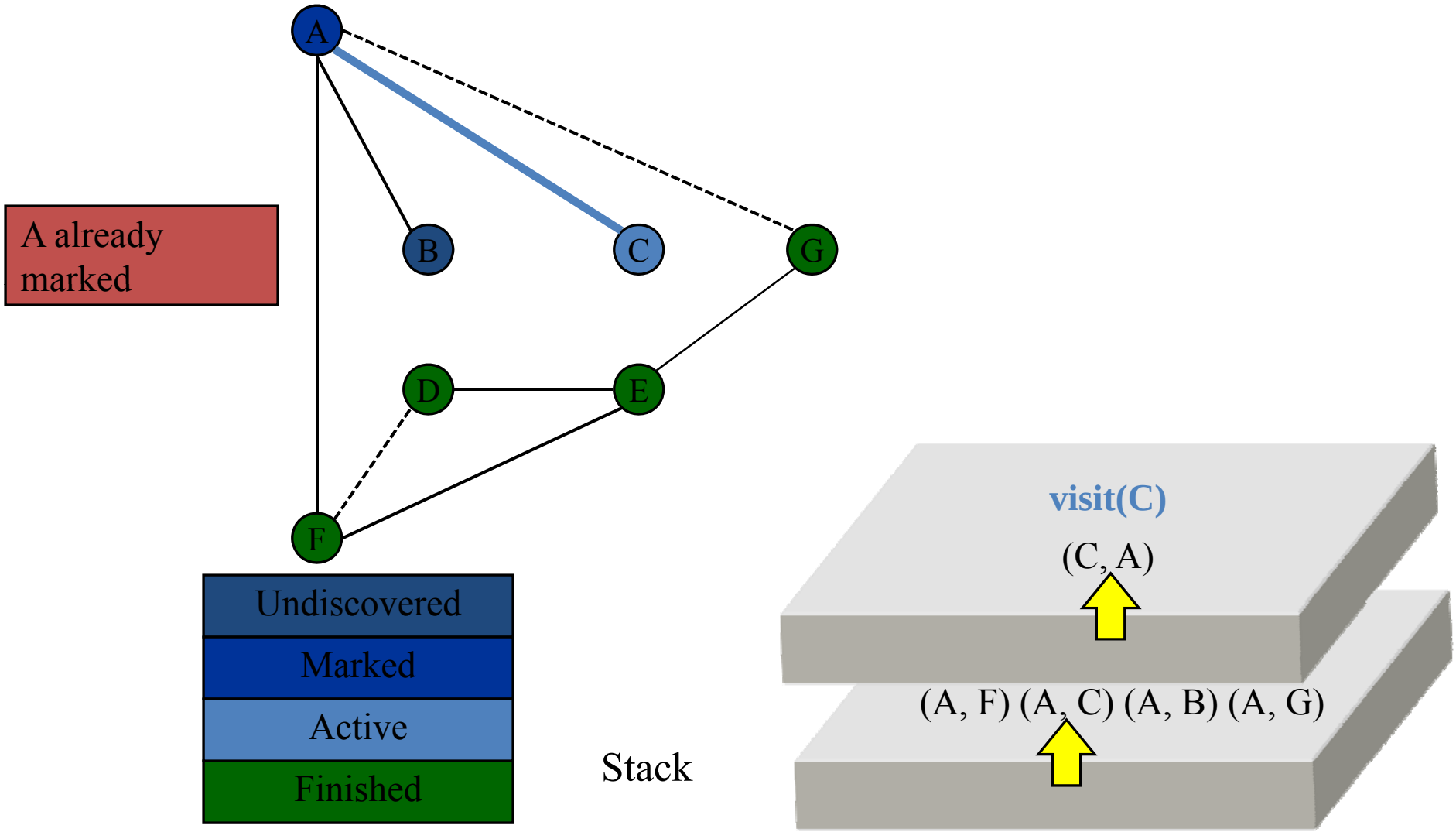
Stack



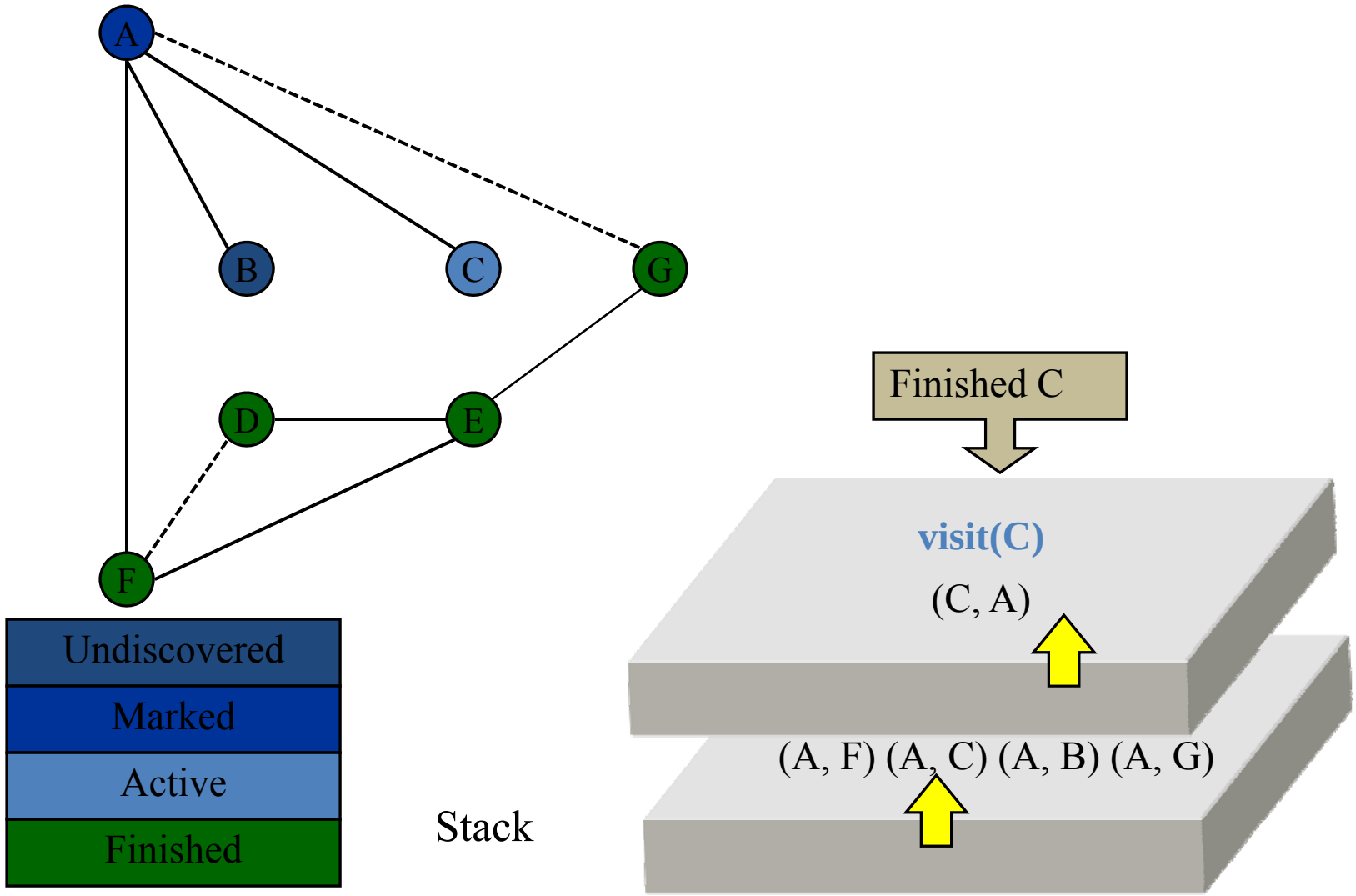
Depth First Search



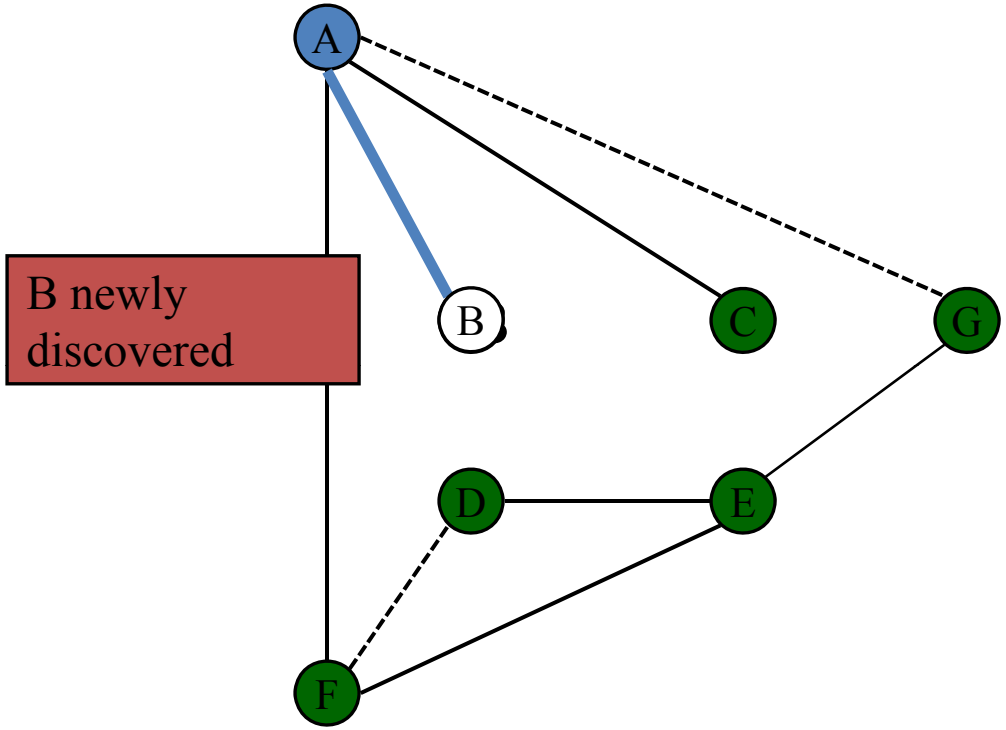
Depth First Search



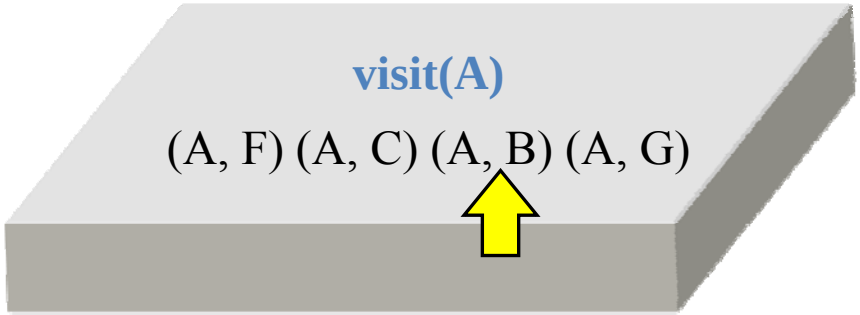
Depth First Search



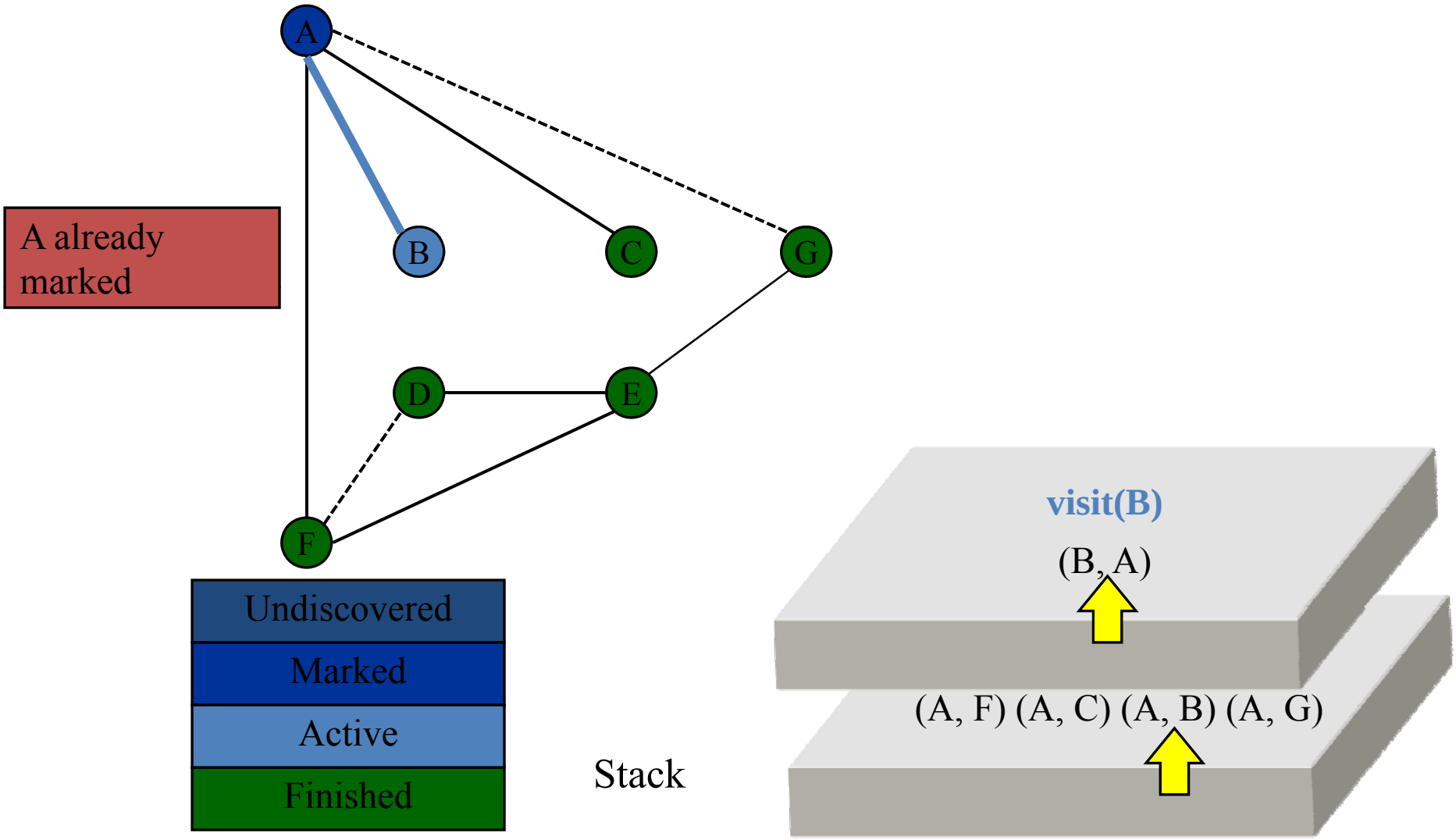
Depth First Search



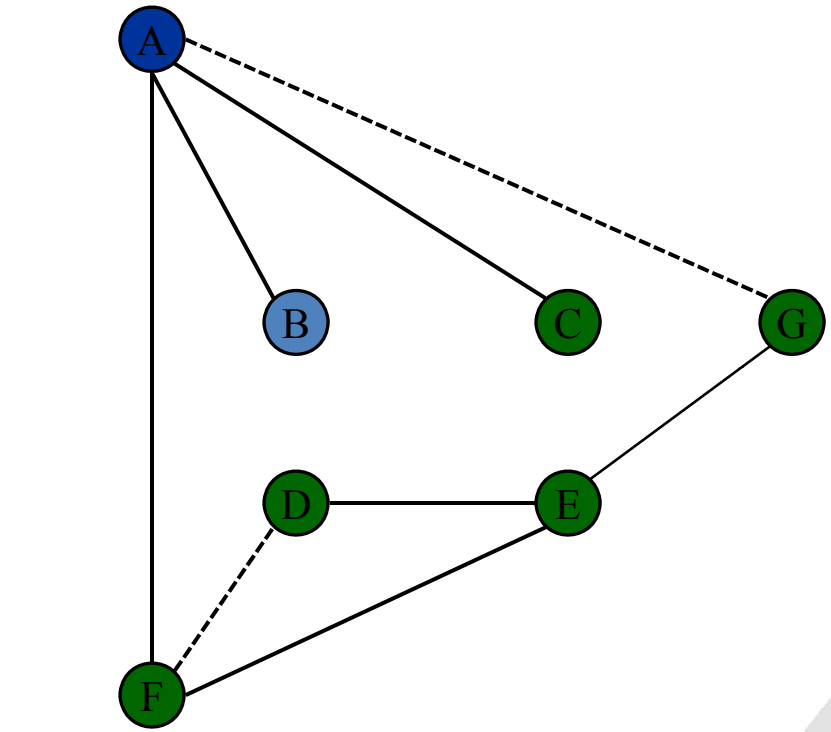
Stack



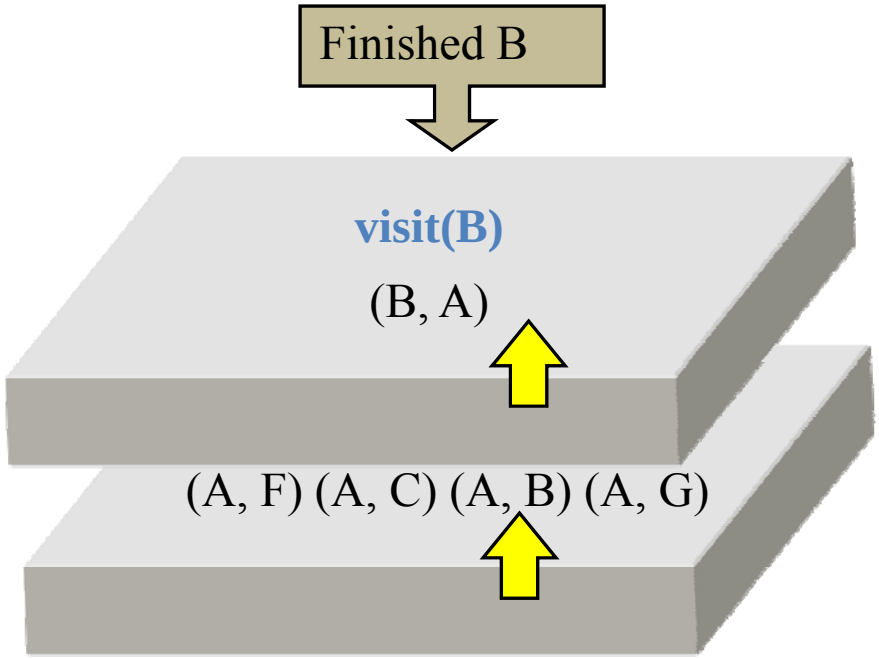
Depth First Search



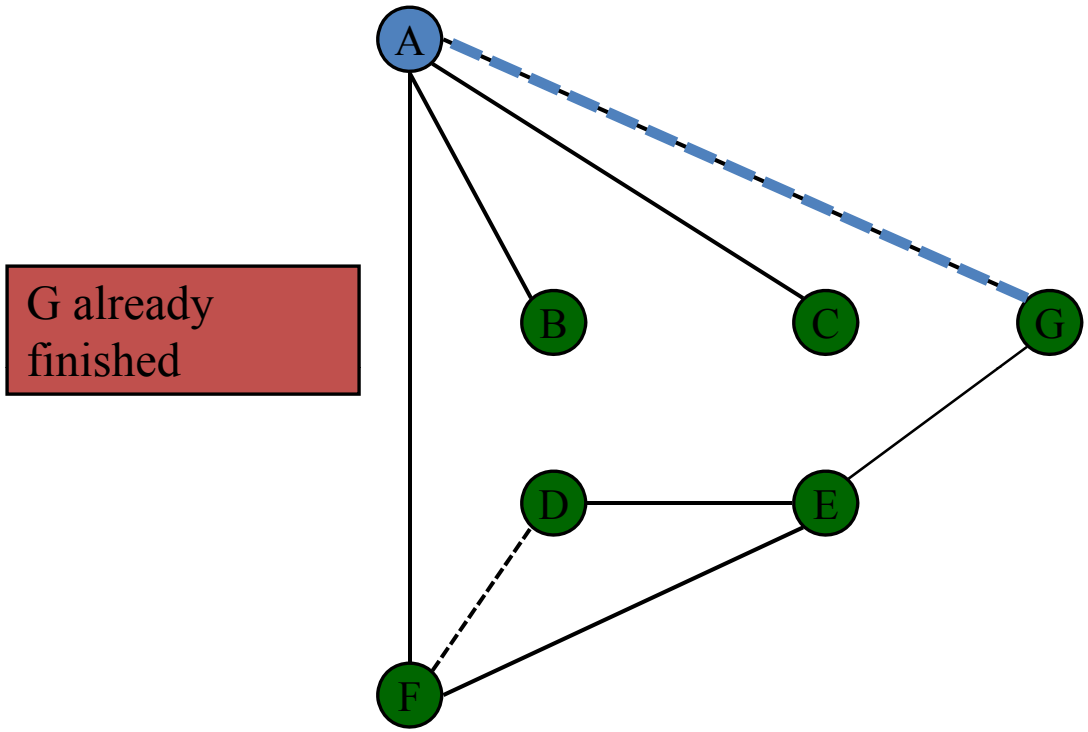
Depth First Search



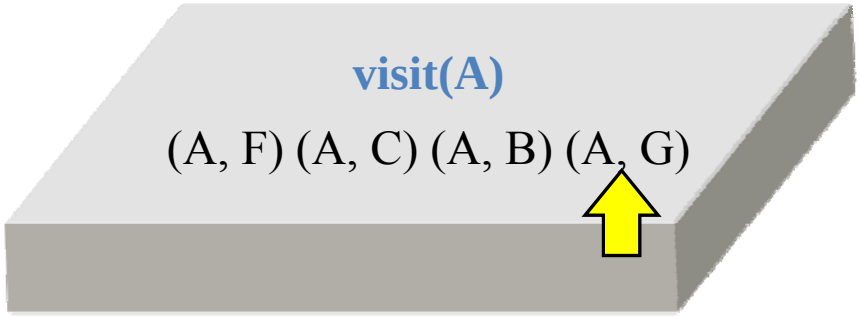
Stack



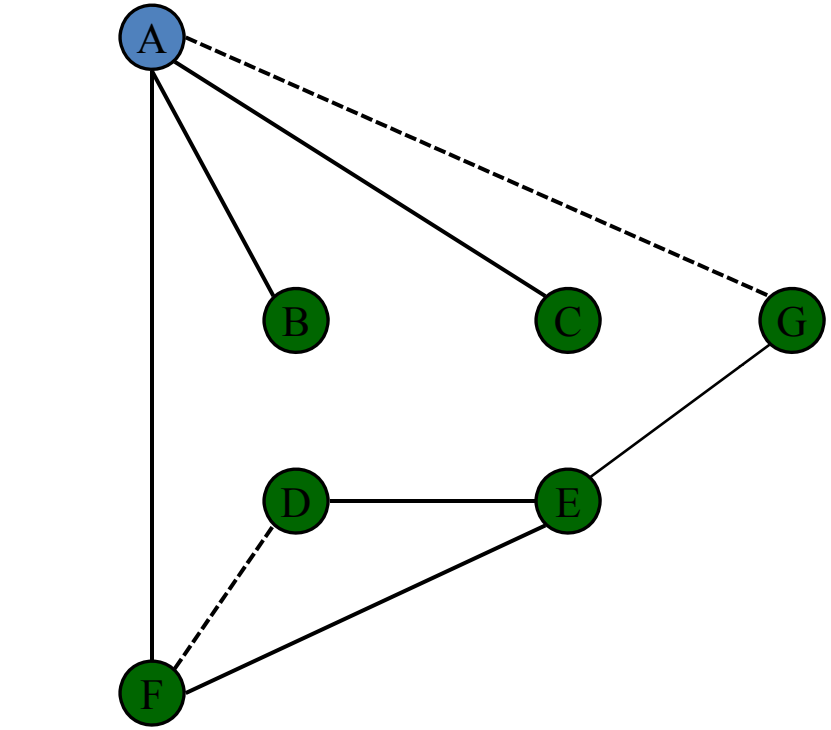
Depth First Search



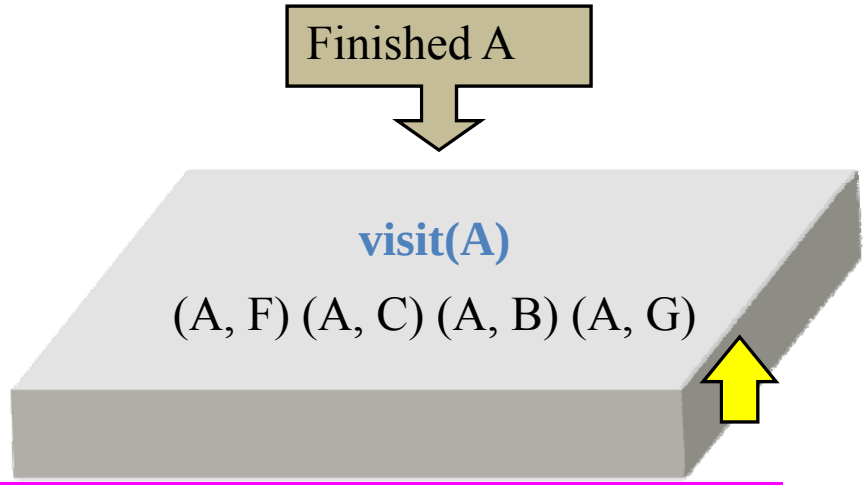
Stack



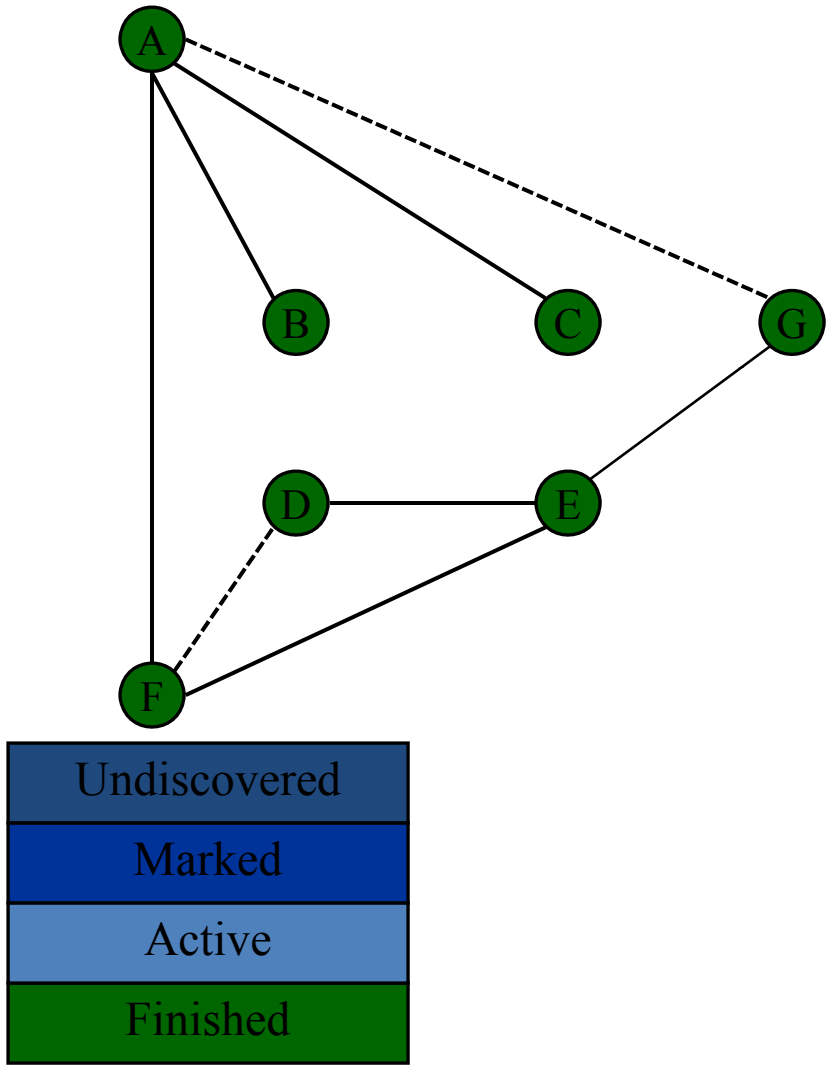
Depth First Search



Stack



Depth First Search



Breadth First Search

- Breadth-First Search (BFS) traverses a graph, and in doing so defines a tree with several useful properties.
- The starting vertex *s* has level 0, and, as in DFS, defines that point as an "anchor."
- In the first round, all of the edges that are only one edge away from the anchor are visited.
- These vertices are placed into level 1;
- In the second round, all the new edges from level 1 that can be reached are visited and placed in level 2.
- This continues until every vertex has been assigned a level.
- The label of any vertex ***v*** corresponds to the length of the shortest path from ***s*** to ***v***.

Breadth First Search

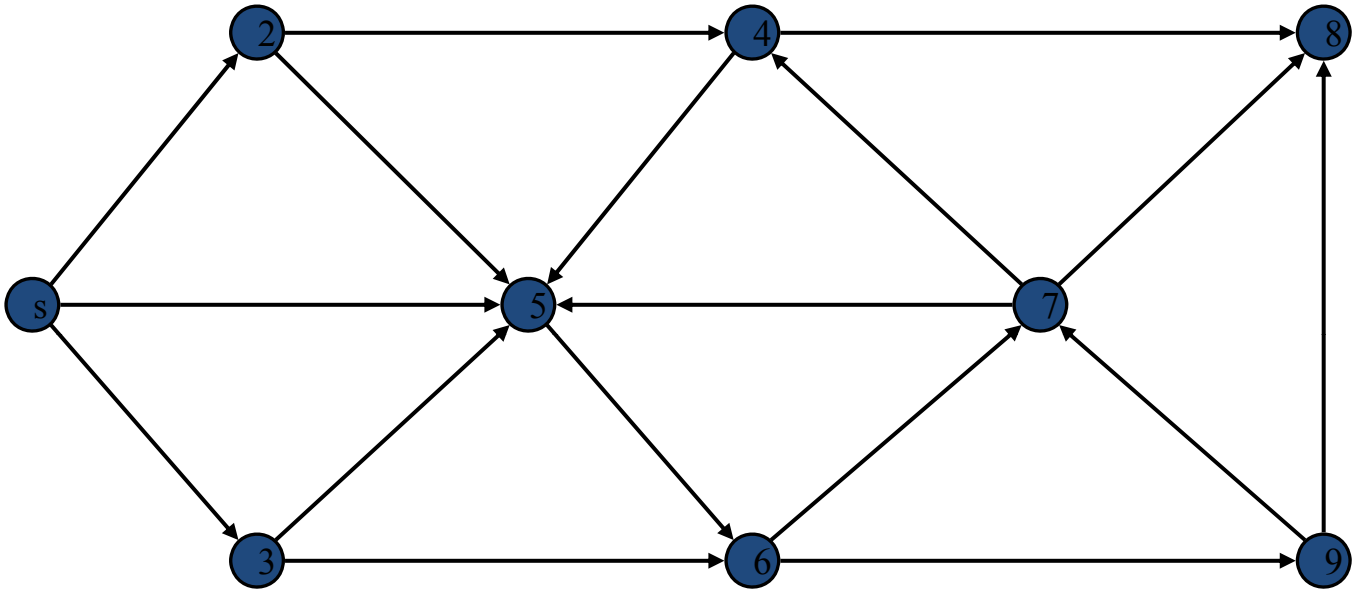
```
void BFS (int start)  {
    int v, result;      queue q;              adjvertex *adj;
    visited [start] = 1;                      enqueue (start, &q);
    while ((result = dequeue (&v, &q)) != -1)                    {
                        printf ("%d", v);     adj = g→adjlist [v];
                        while (adj != NULL) {
                                    if (! visited [adj→vertex])
                                    {
                                                visited [adj→vertex] = 1;
                                                enqueue (adj→vertex, &q);
                                    }
                                    adj = adj→next;
                        }           }           }
```

Nodes dequeued

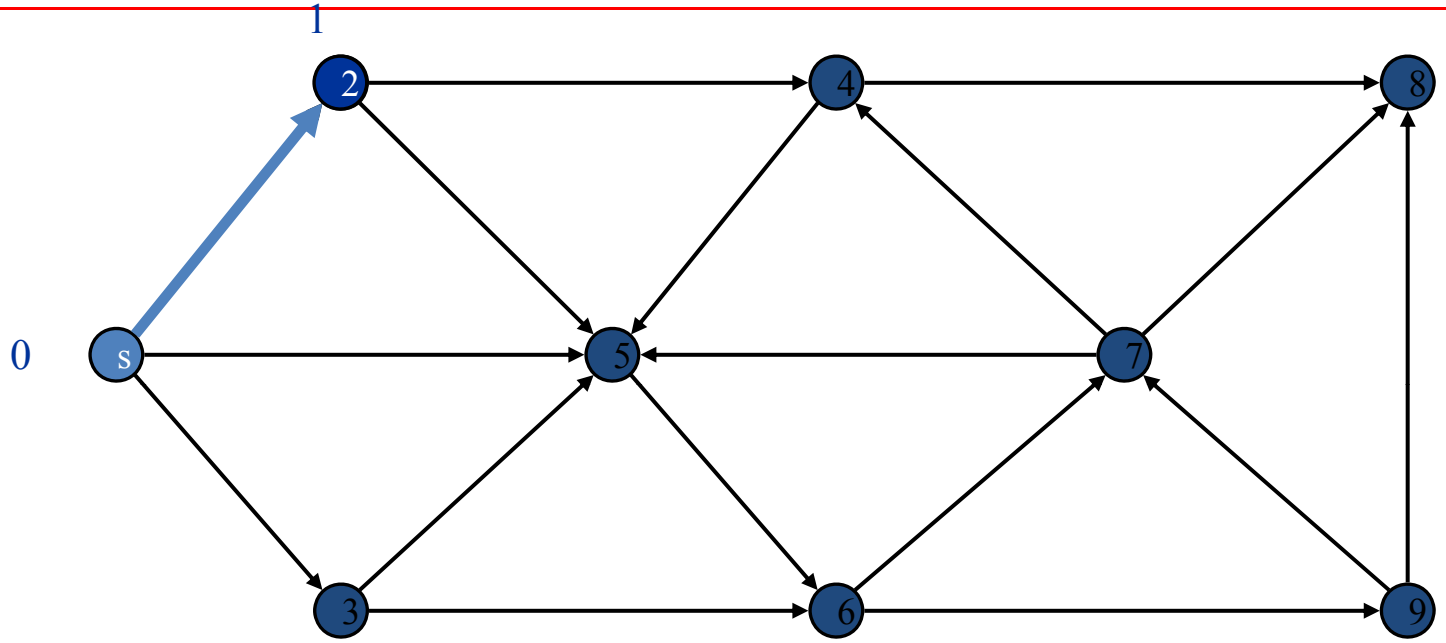
Nodes enqueued exactly once

Total running time: $O(2e)=O(e)$

Breadth First Search



Breadth First Search



Undiscovered

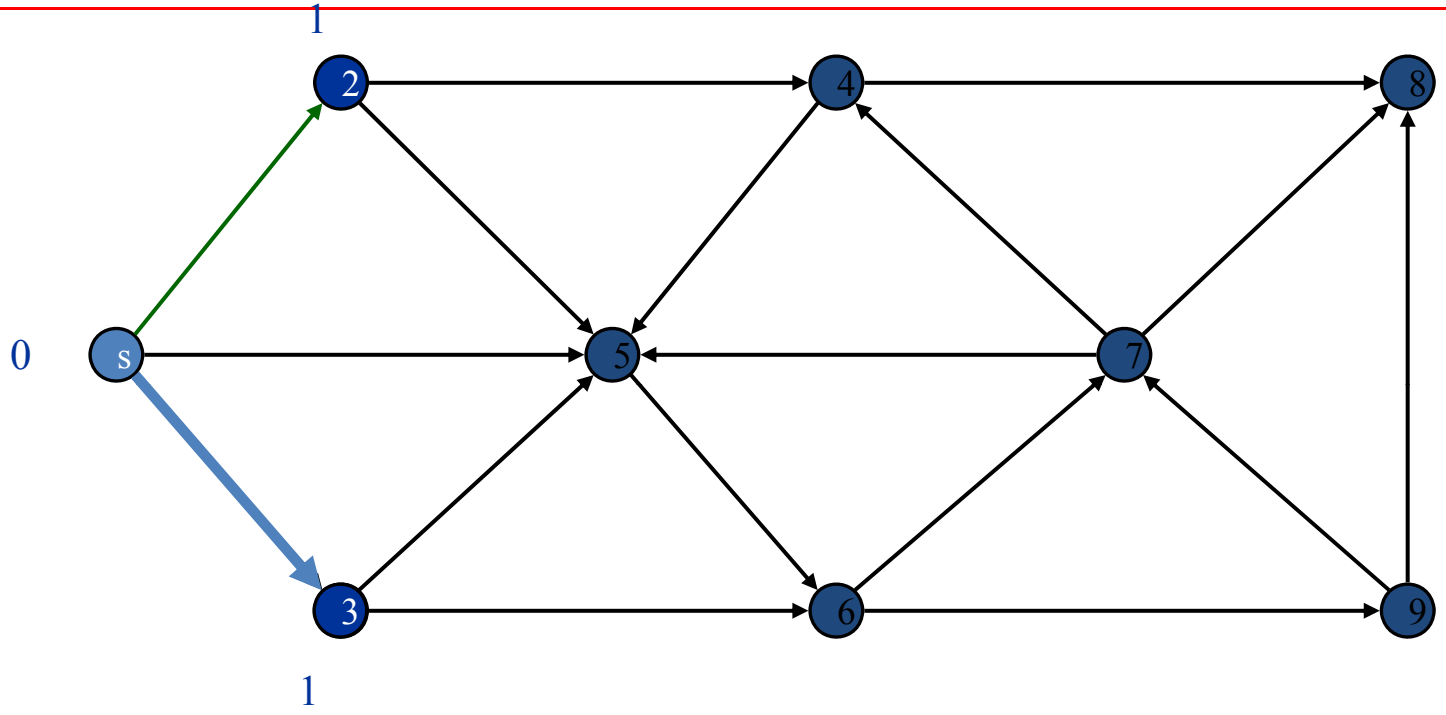
Discovered

Top of queue

Finished

Queue: s

Breadth First Search



Undiscovered

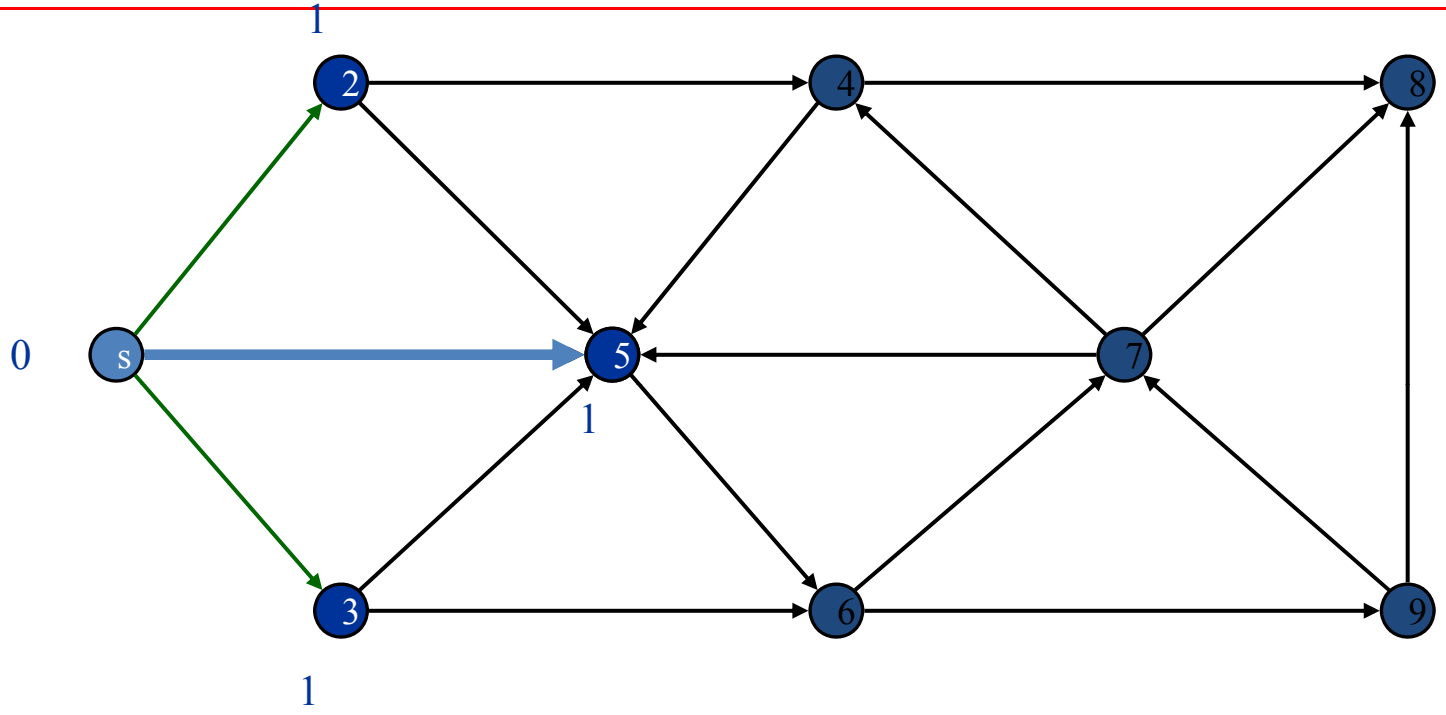
Discovered

Top of queue

Finished

Queue: s 2

Breadth First Search



Undiscovered

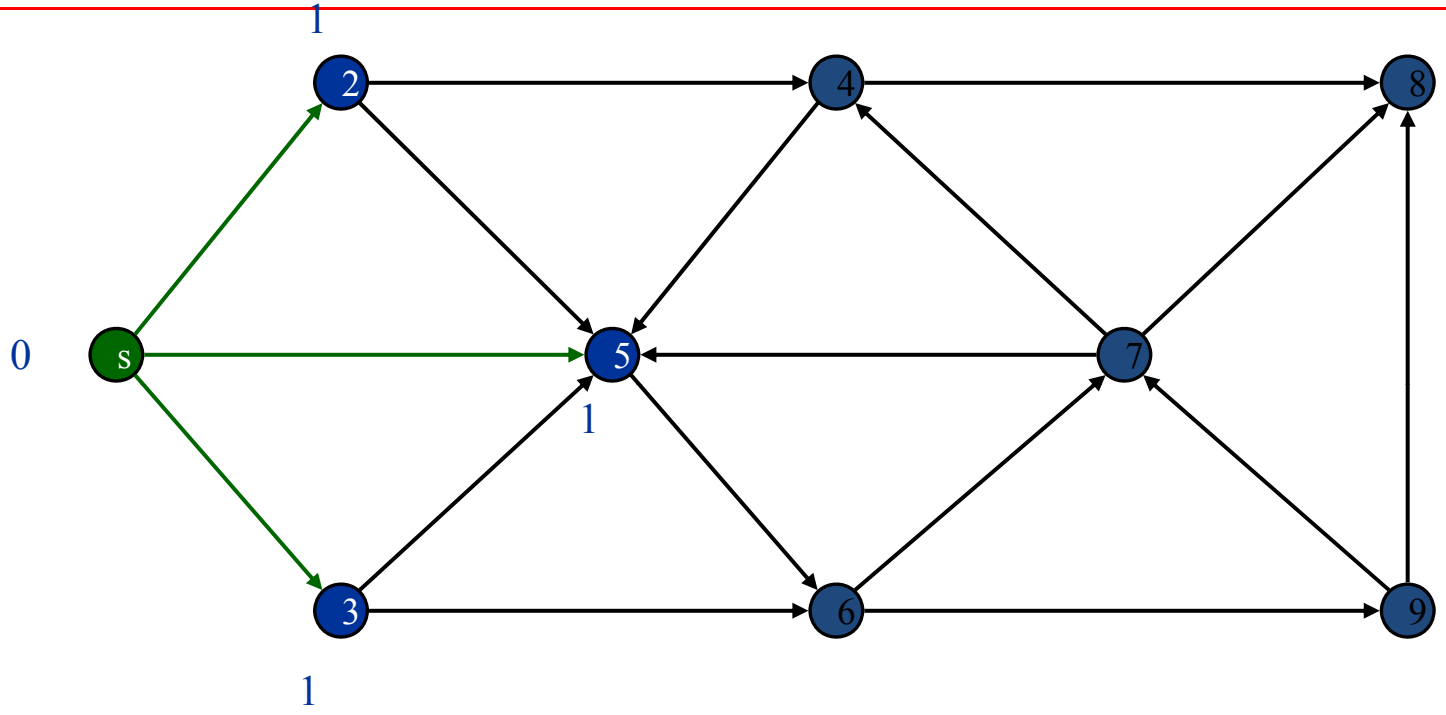
Discovered

Top of queue

Finished

Queue: s 2 3

Breadth First Search



Undiscovered

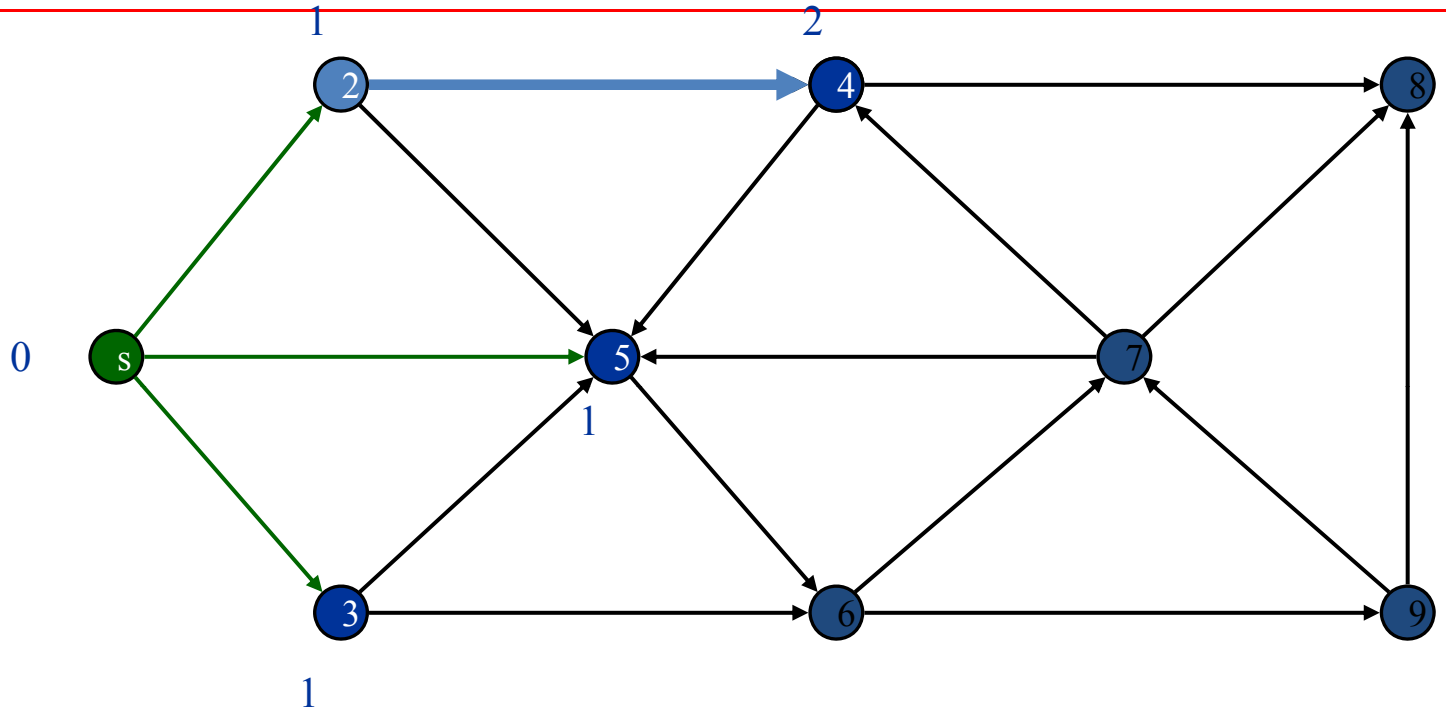
Discovered

Top of queue

Finished

Queue: 2 3 5

Breadth First Search



Undiscovered

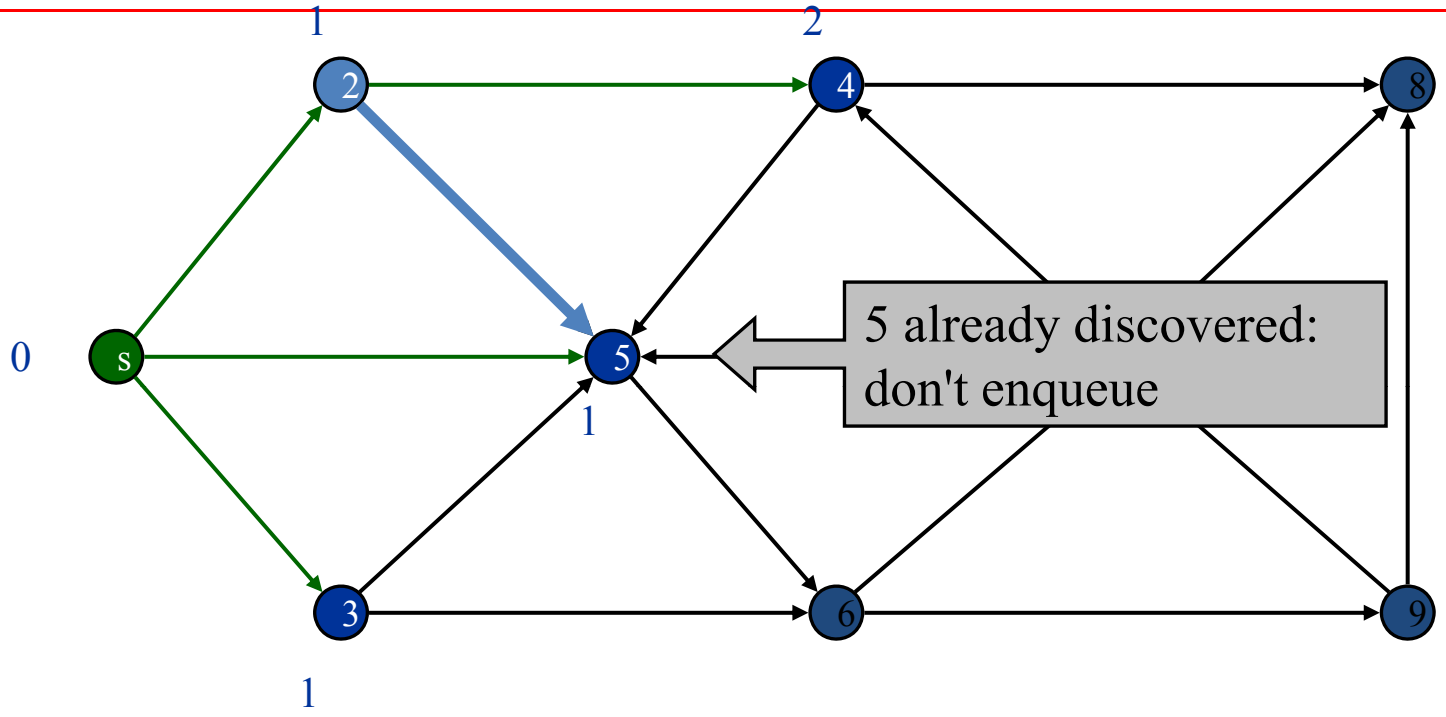
Discovered

Top of queue

Finished

Queue: 2 3 5

Breadth First Search



Undiscovered

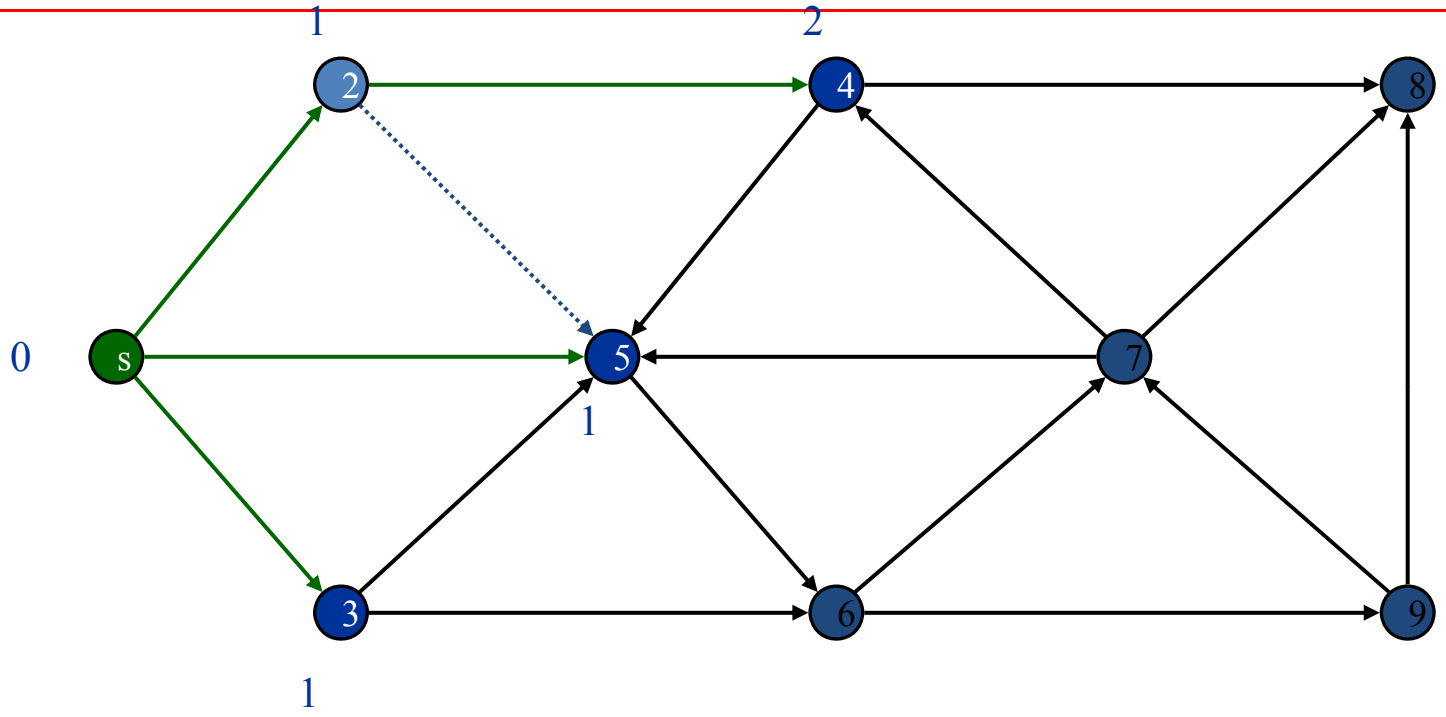
Discovered

Top of queue

Finished

Queue: 2 3 5 4

Breadth First Search



Undiscovered

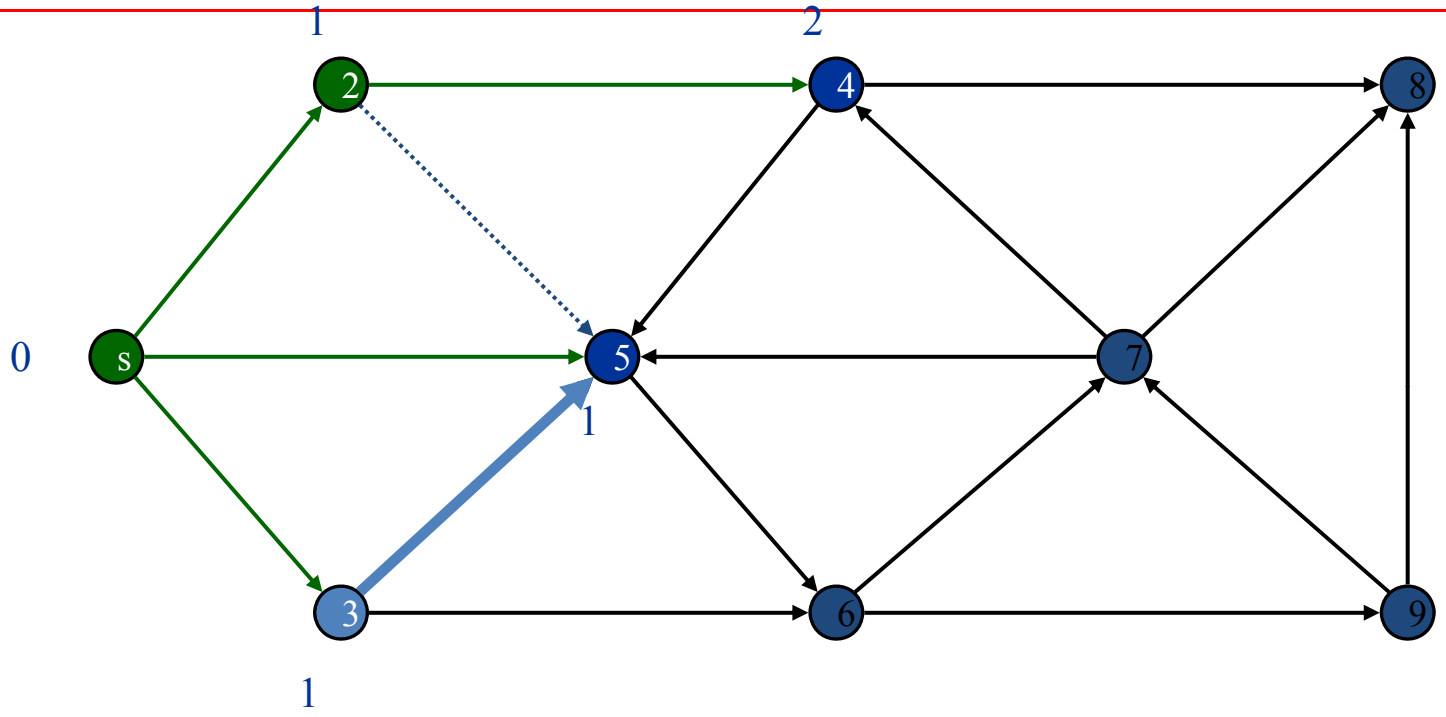
Discovered

Top of queue

Finished

Queue: 2 3 5 4

Breadth First Search



Undiscovered

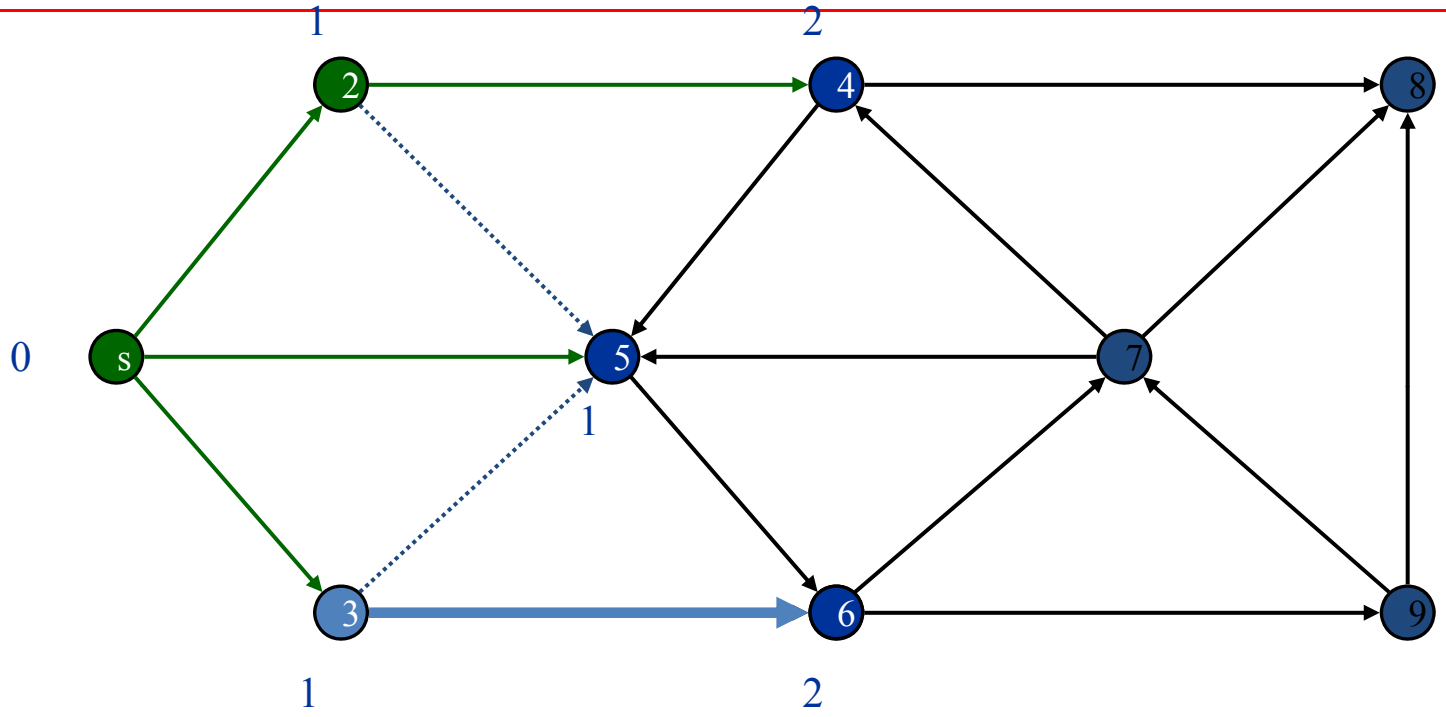
Discovered

Top of queue

Finished

Queue: 3 5 4

Breadth First Search



Undiscovered

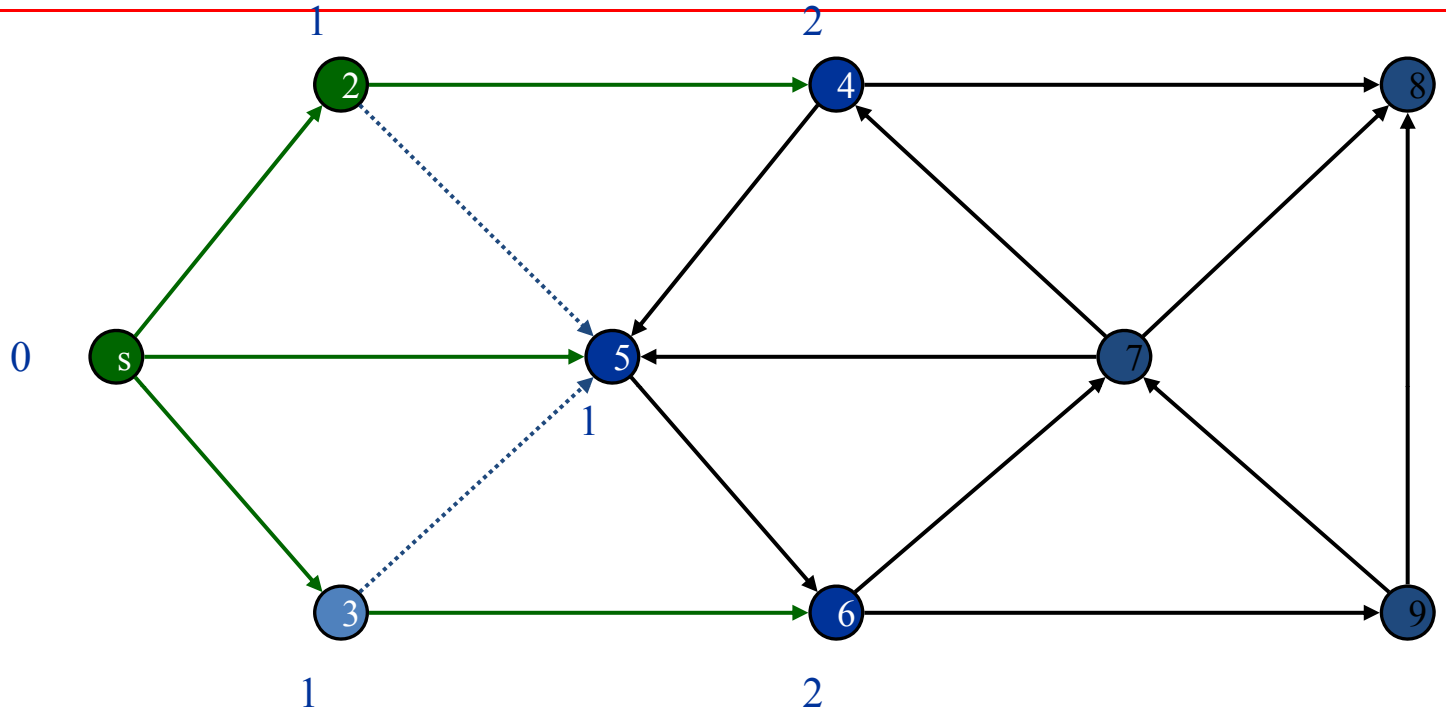
Discovered

Top of queue

Finished

Queue: 3 5 4

Breadth First Search



Undiscovered

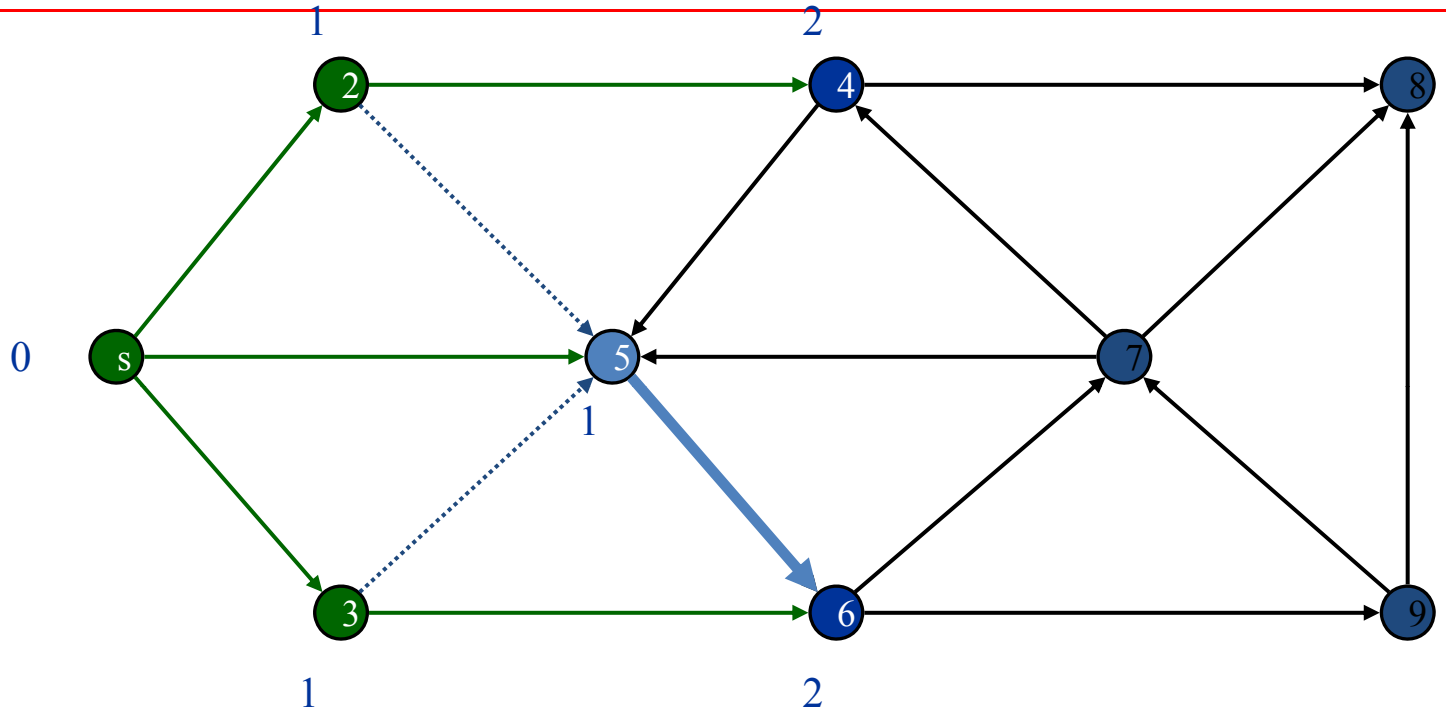
Discovered

Top of queue

Finished

Queue: 3 5 4 6

Breadth First Search



Undiscovered

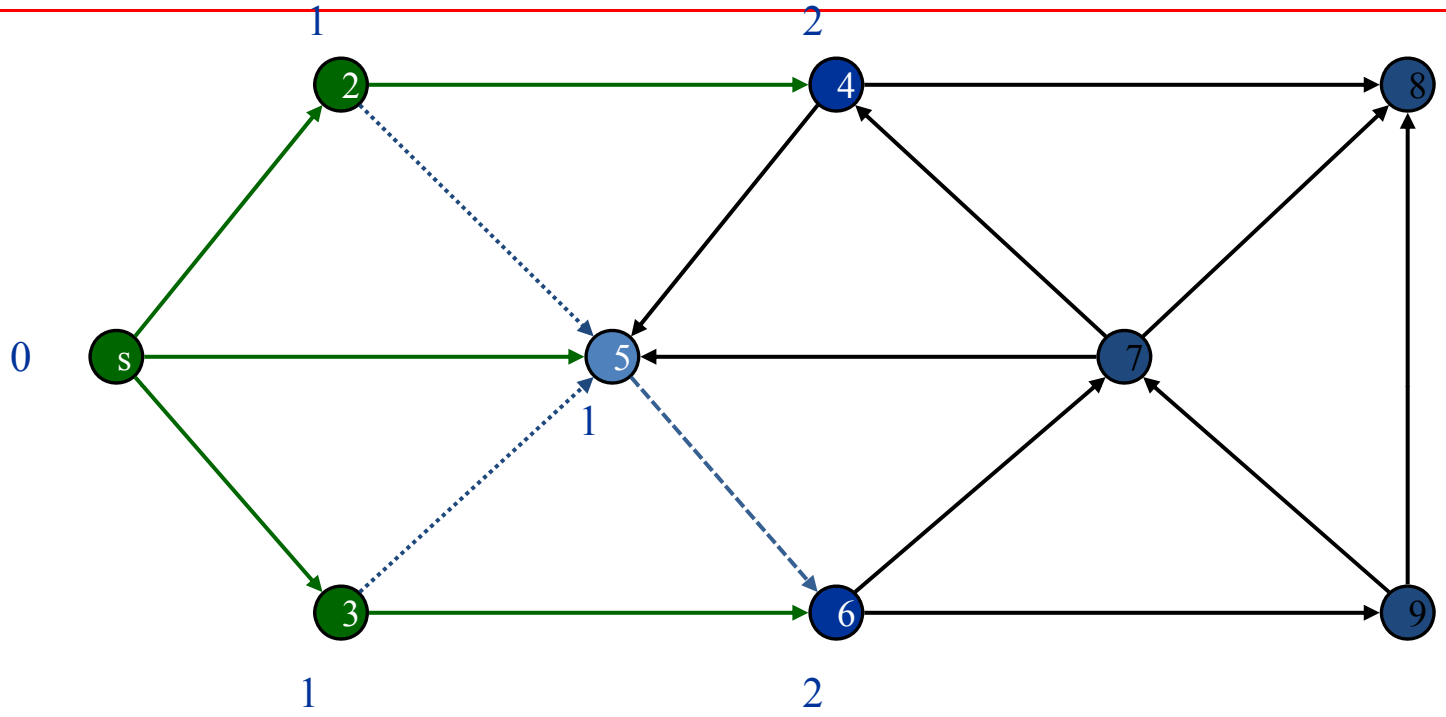
Discovered

Top of queue

Finished

Queue: 5 4 6

Breadth First Search



Undiscovered

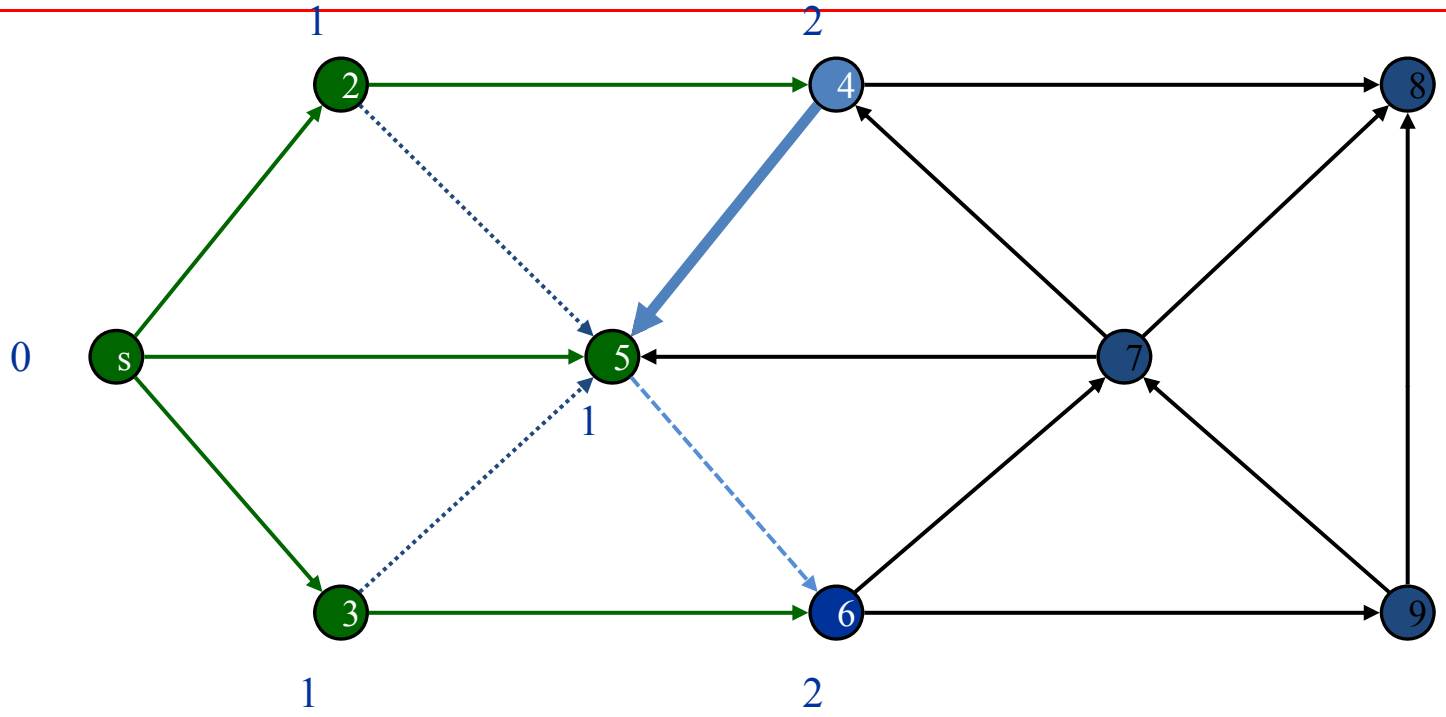
Discovered

Top of queue

Finished

Queue: 5 4 6

Breadth First Search



Undiscovered

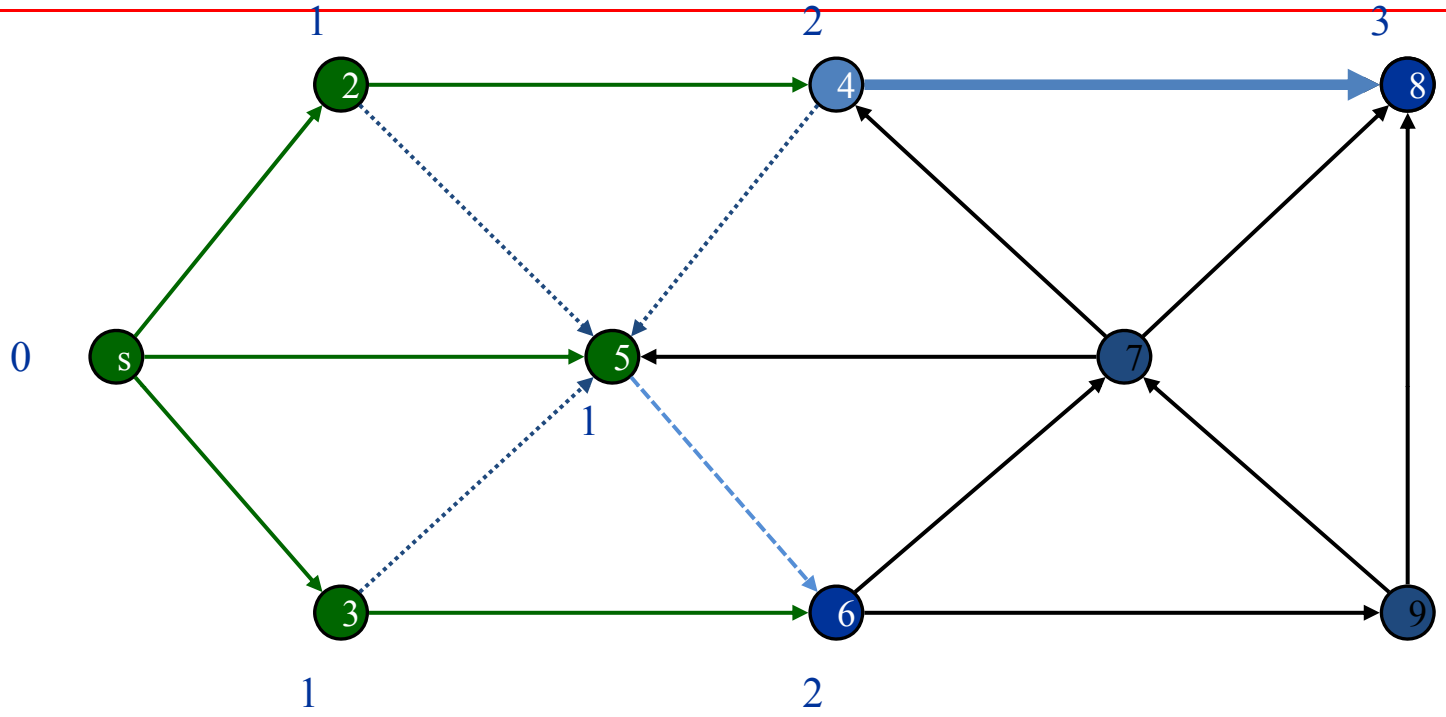
Discovered

Top of queue

Finished

Queue: 4 6

Breadth First Search



Undiscovered

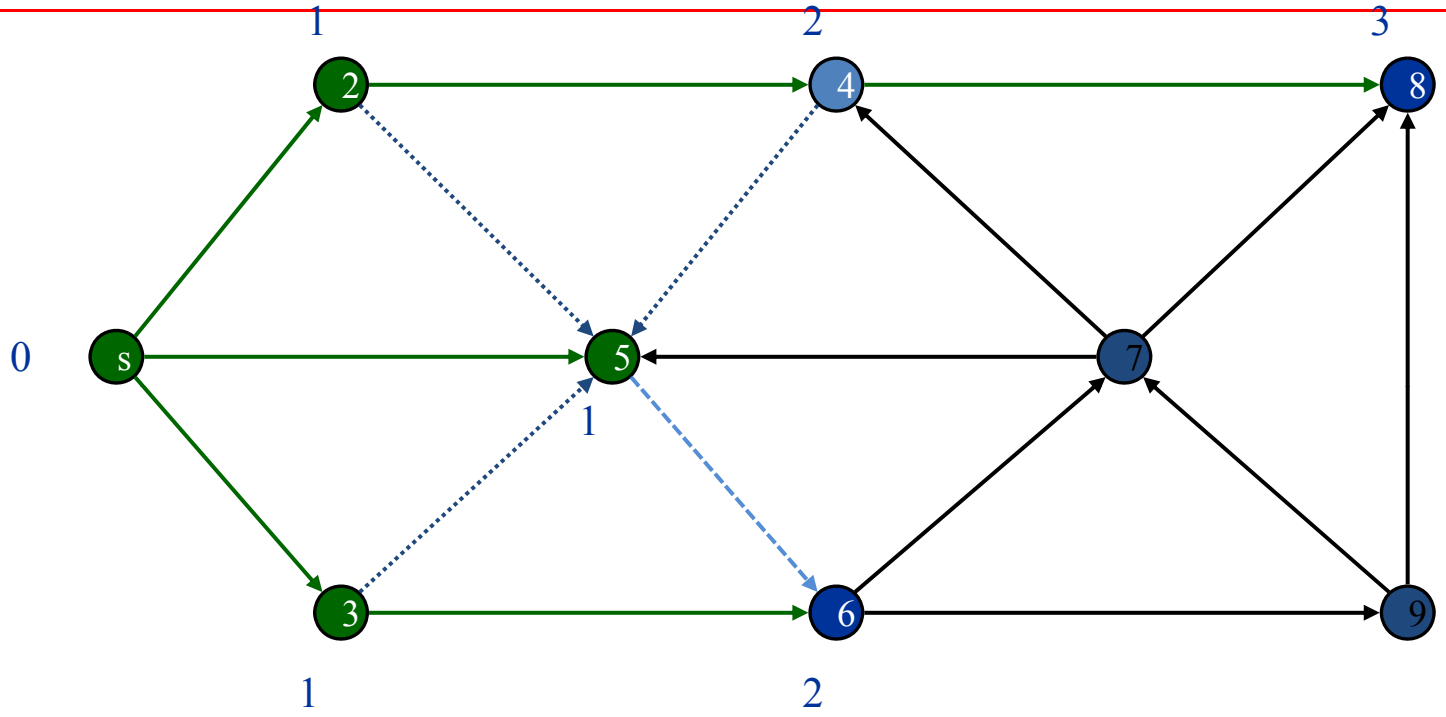
Discovered

Top of queue

Finished

Queue: 4 6

Breadth First Search



Undiscovered

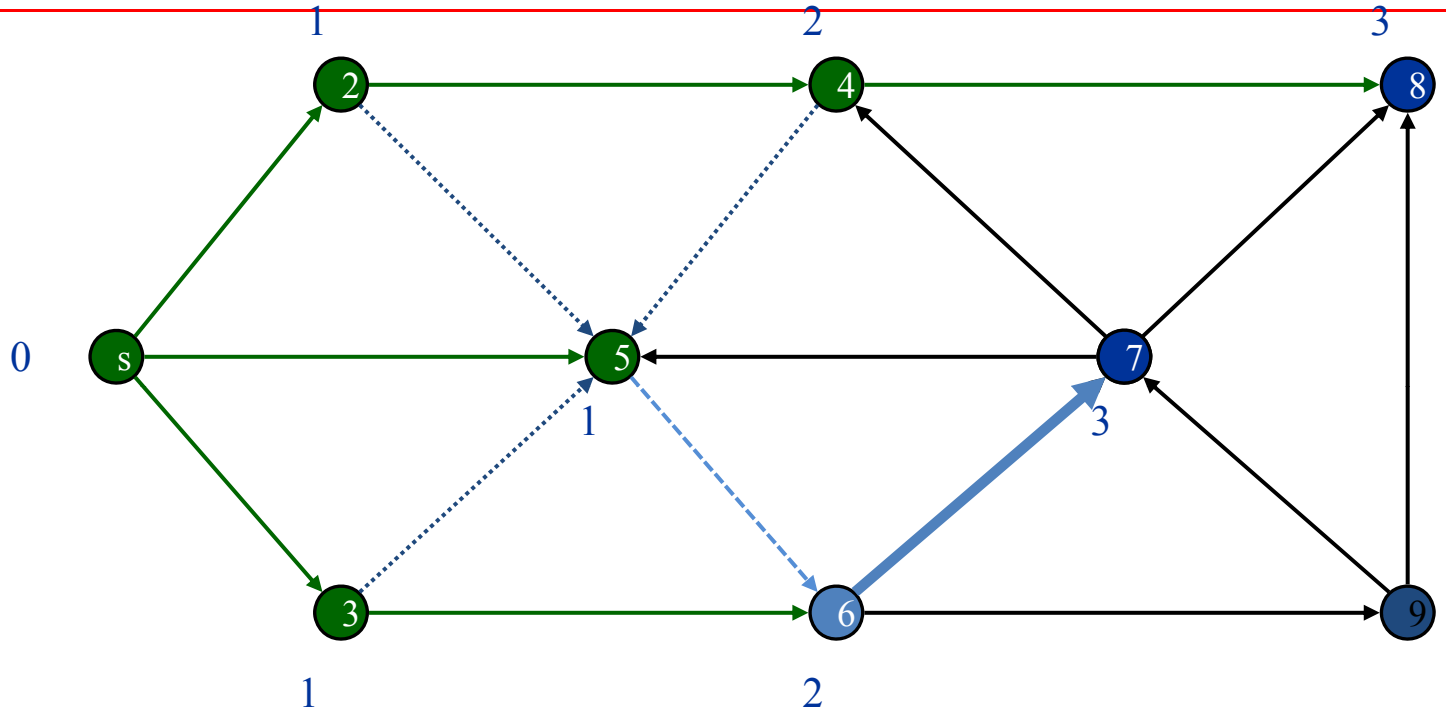
Discovered

Top of queue

Finished

Queue: 4 6 8

Breadth First Search



Undiscovered

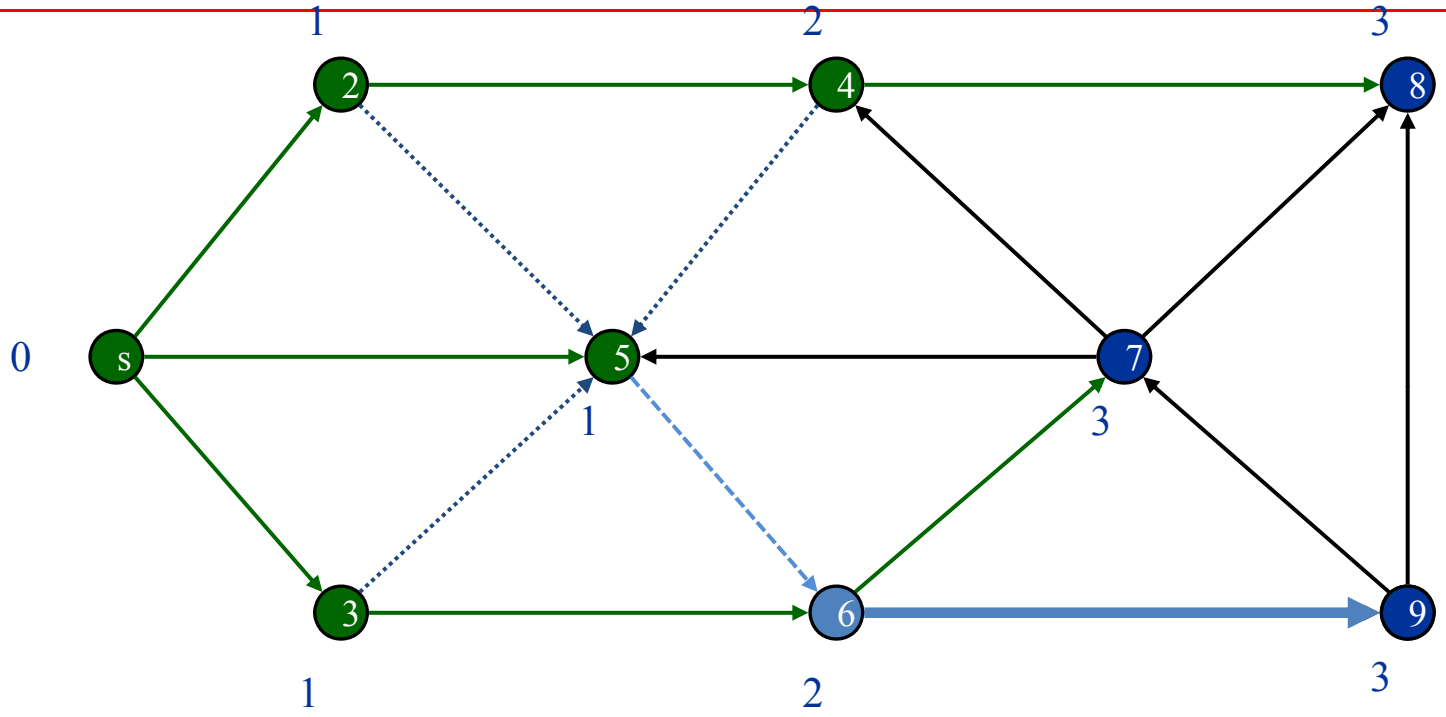
Discovered

Top of queue

Finished

Queue: 6 8

Breadth First Search



Undiscovered

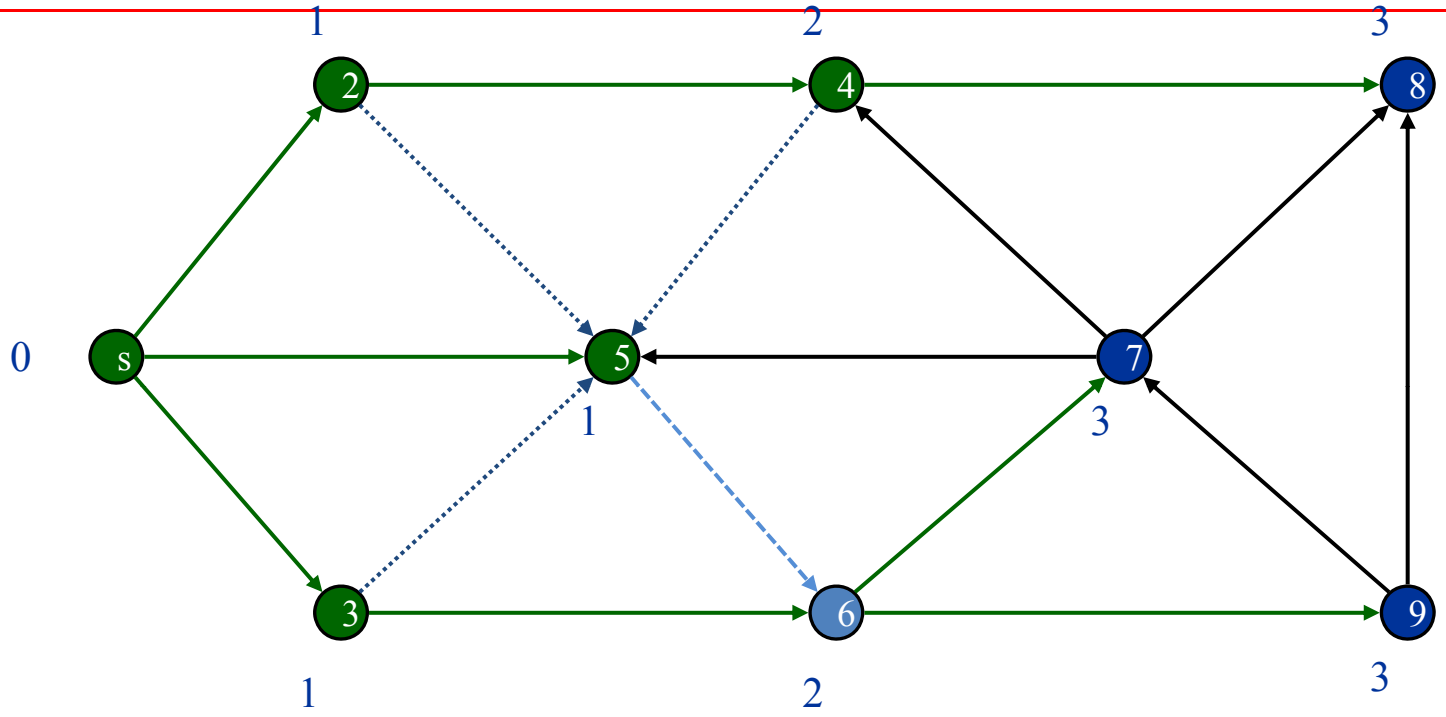
Discovered

Top of queue

Finished

Queue: 6 8 7

Breadth First Search



Undiscovered

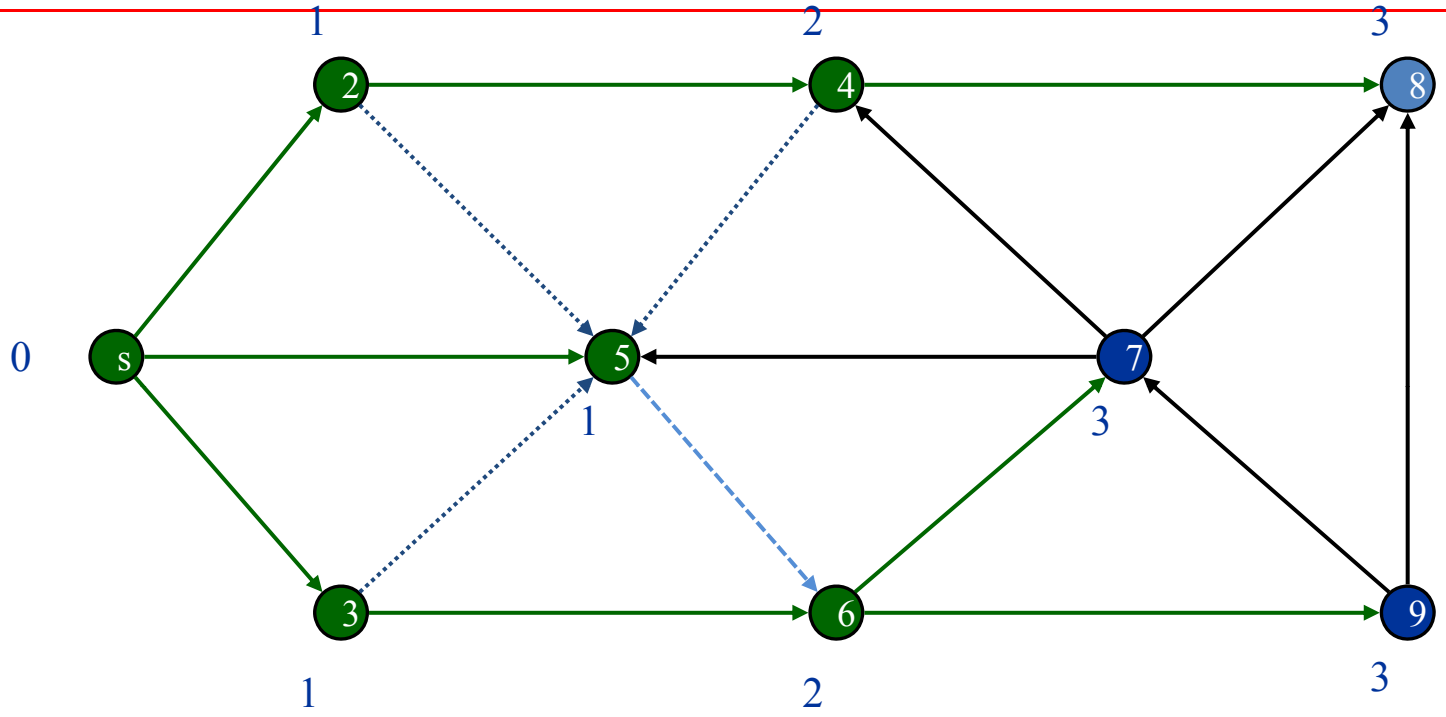
Discovered

Top of queue

Finished

Queue: 6 8 7 9

Breadth First Search



Undiscovered

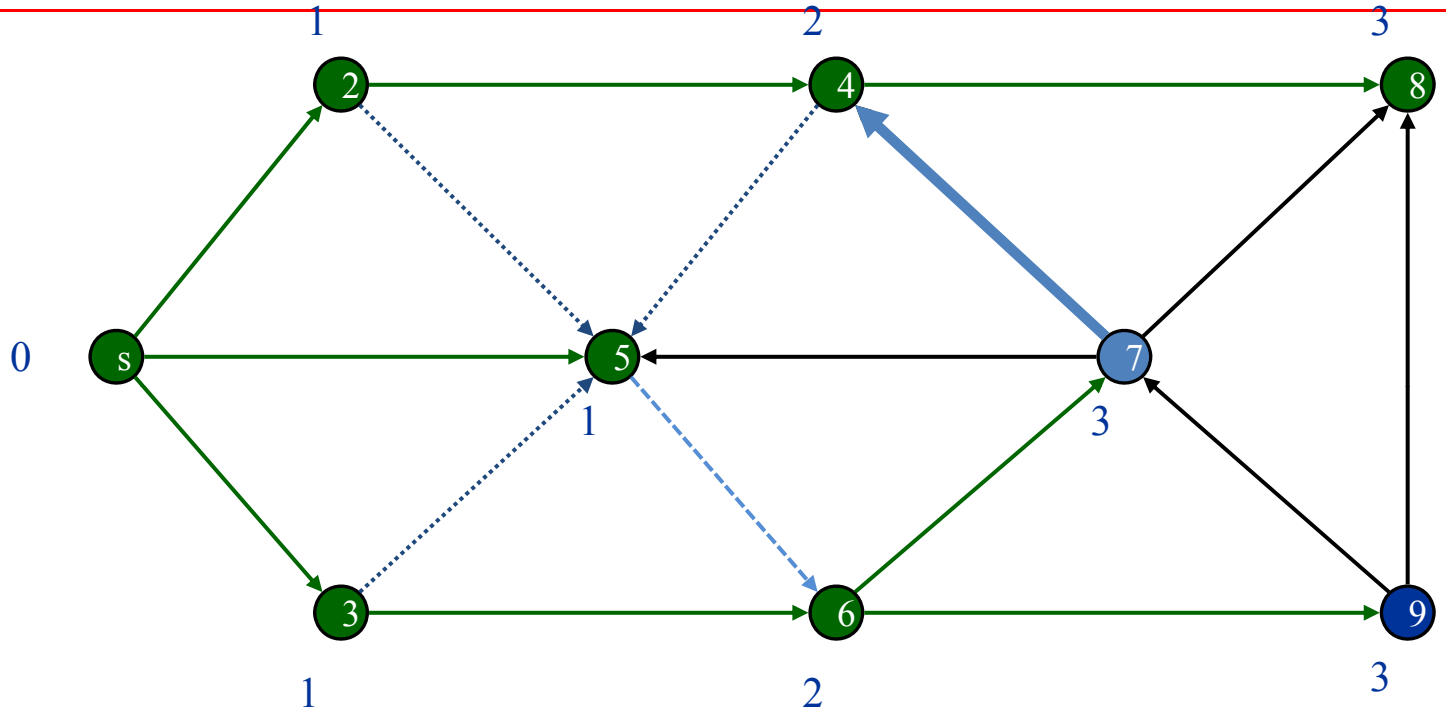
Discovered

Top of queue

Finished

Queue: 8 7 9

Breadth First Search



Undiscovered

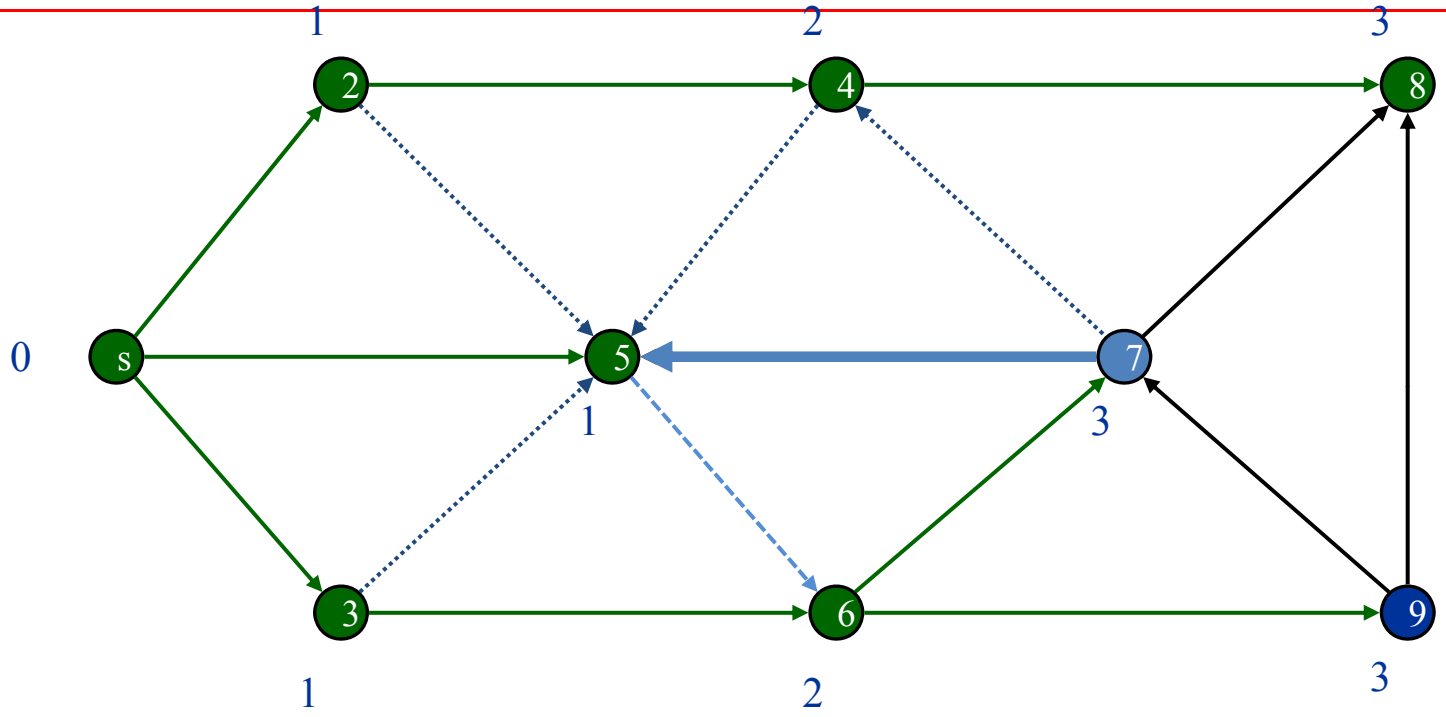
Discovered

Top of queue

Finished

Queue: 7 9

Breadth First Search



Undiscovered

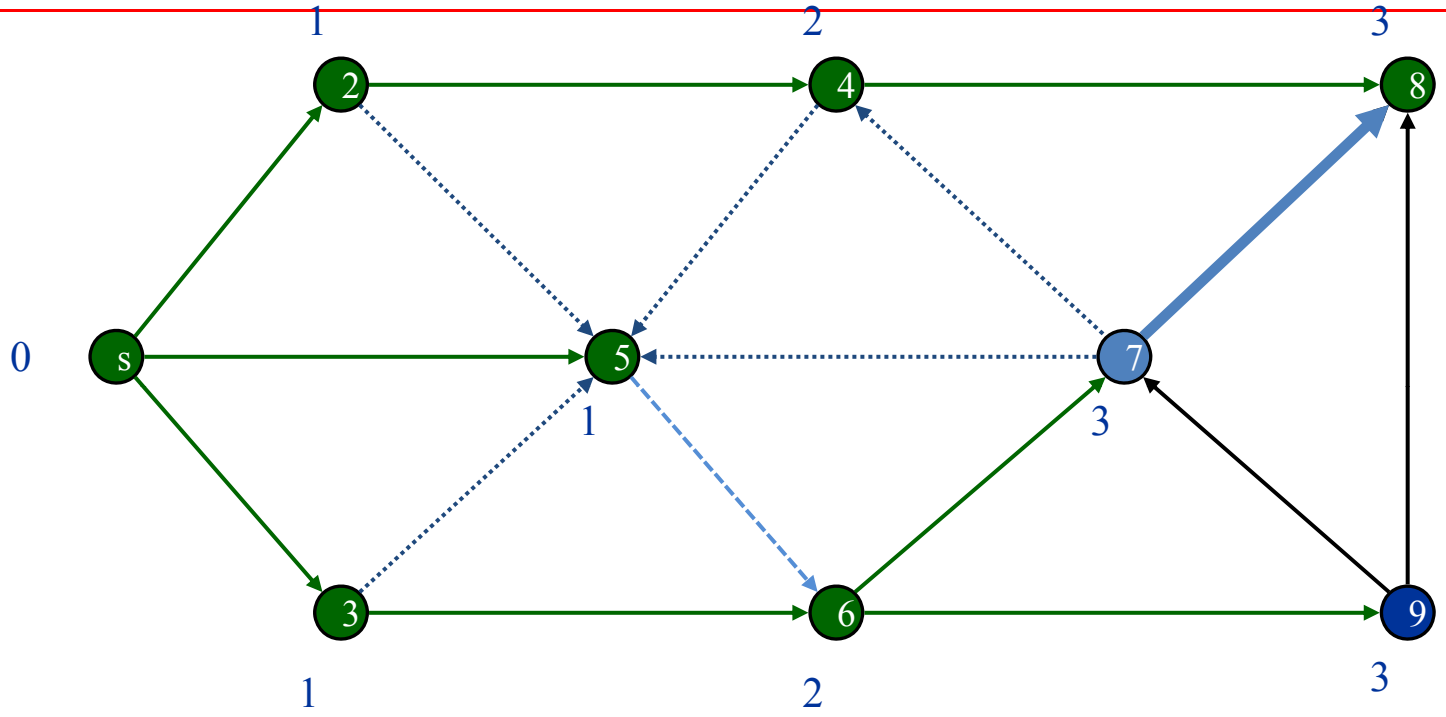
Discovered

Top of queue

Finished

Queue: 7 9

Breadth First Search



Undiscovered

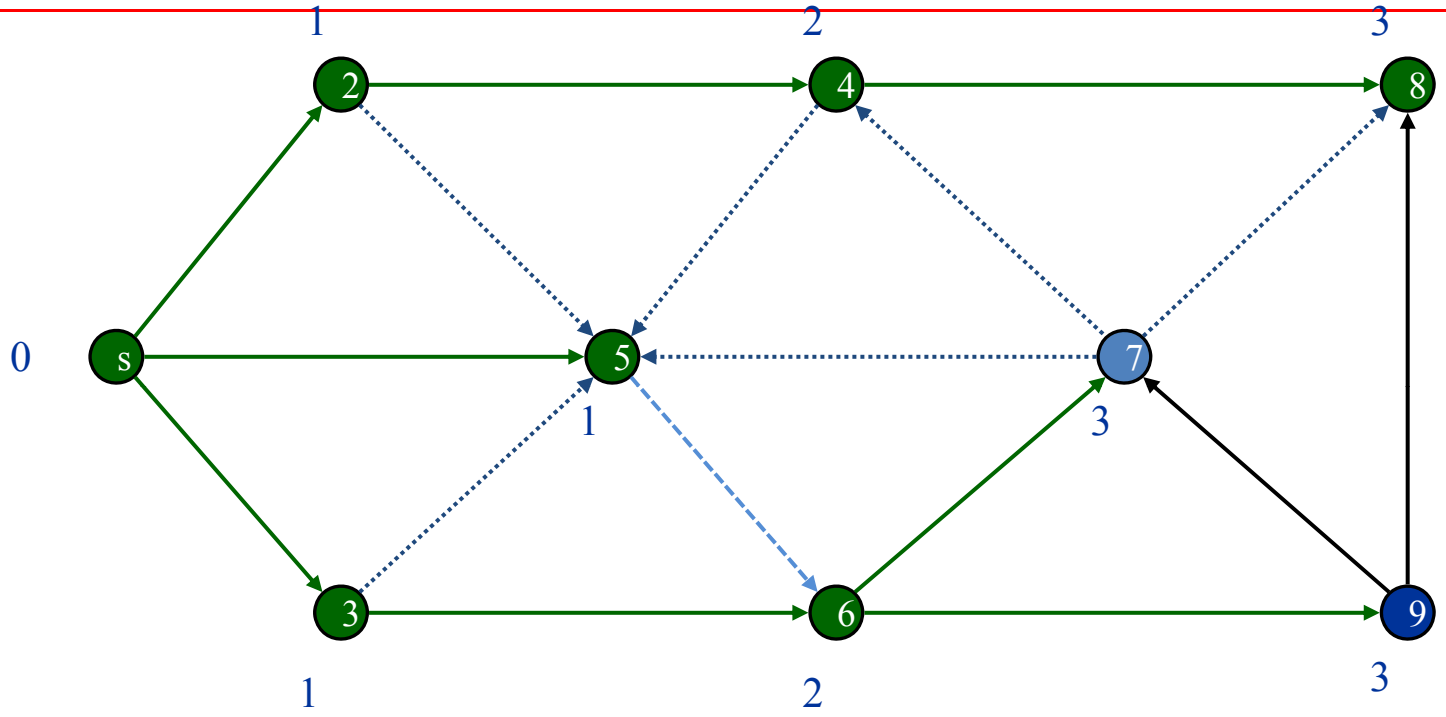
Discovered

Top of queue

Finished

Queue: 7 9

Breadth First Search



Undiscovered

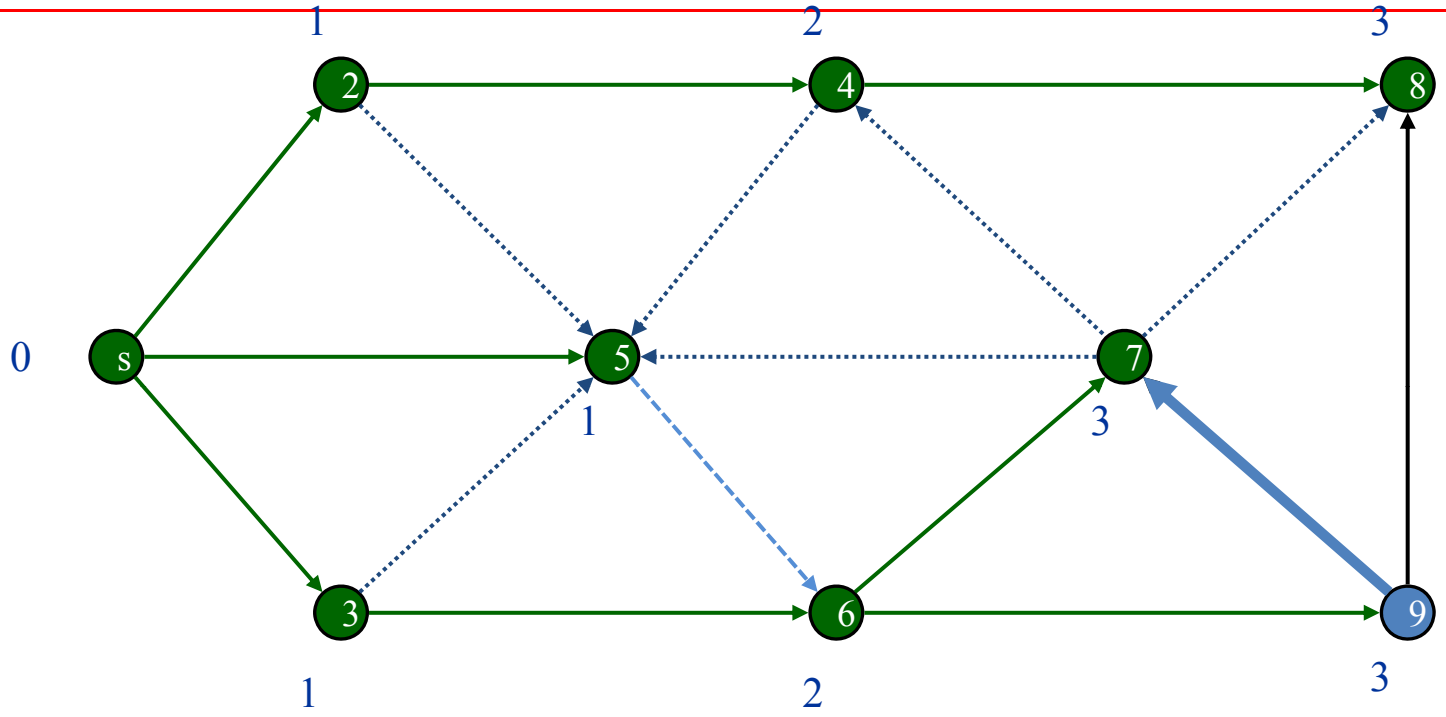
Discovered

Top of queue

Finished

Queue: 7 9

Breadth First Search



Undiscovered

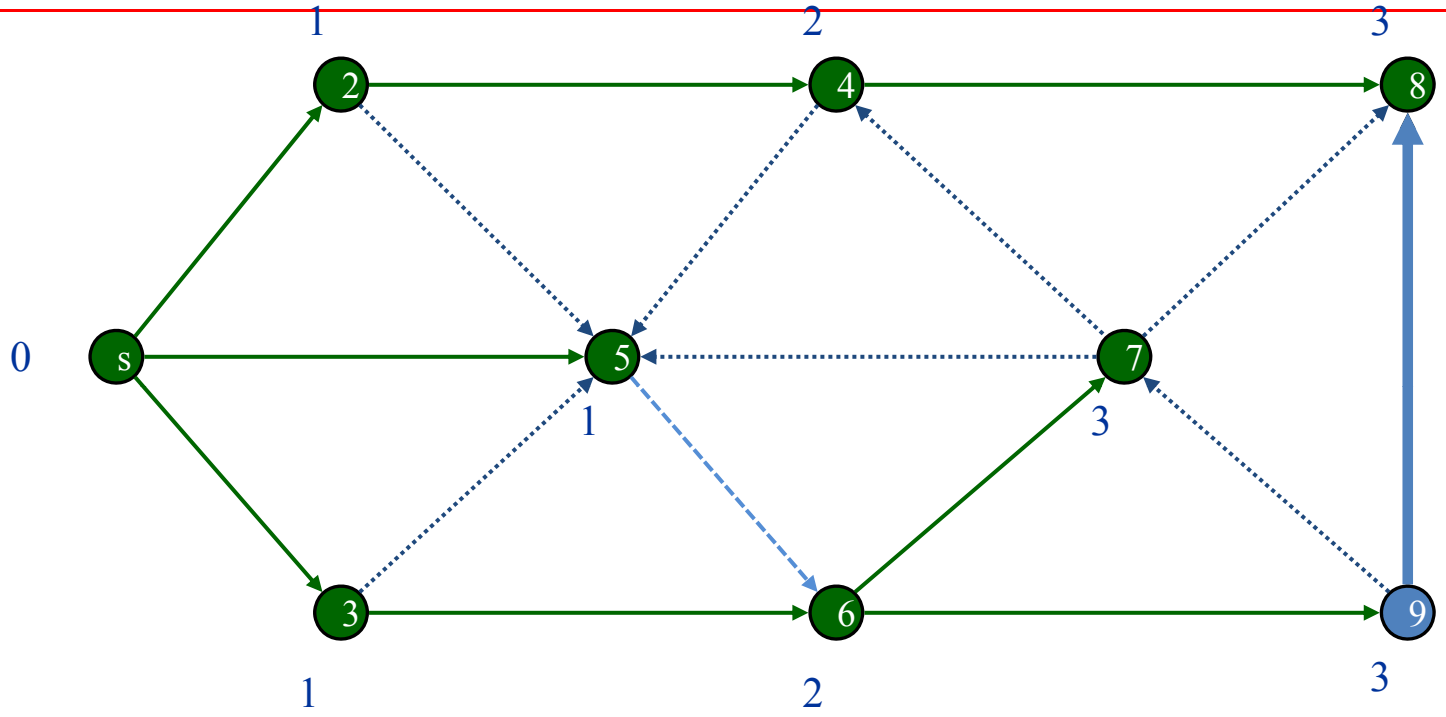
Discovered

Top of queue

Finished

Queue: 9

Breadth First Search



Undiscovered

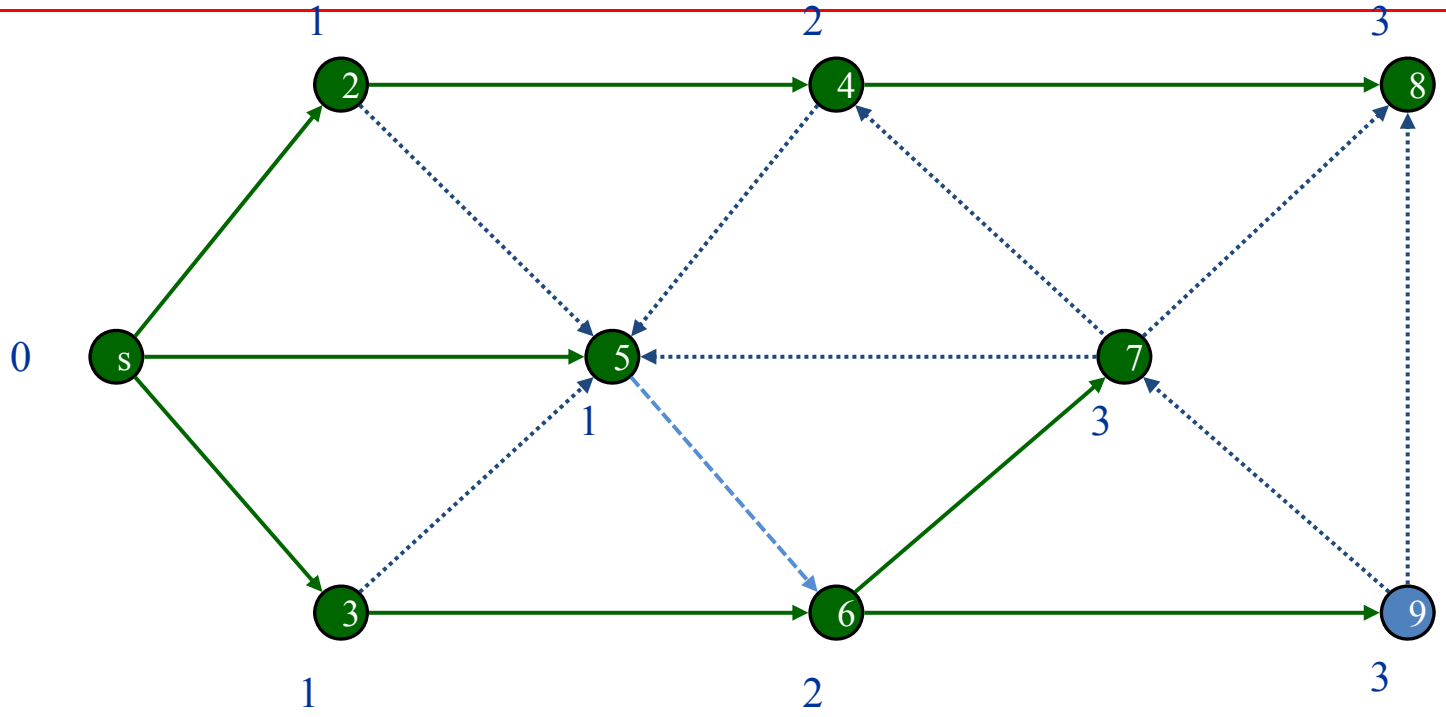
Discovered

Top of queue

Finished

Queue: 9

Breadth First Search



Undiscovered

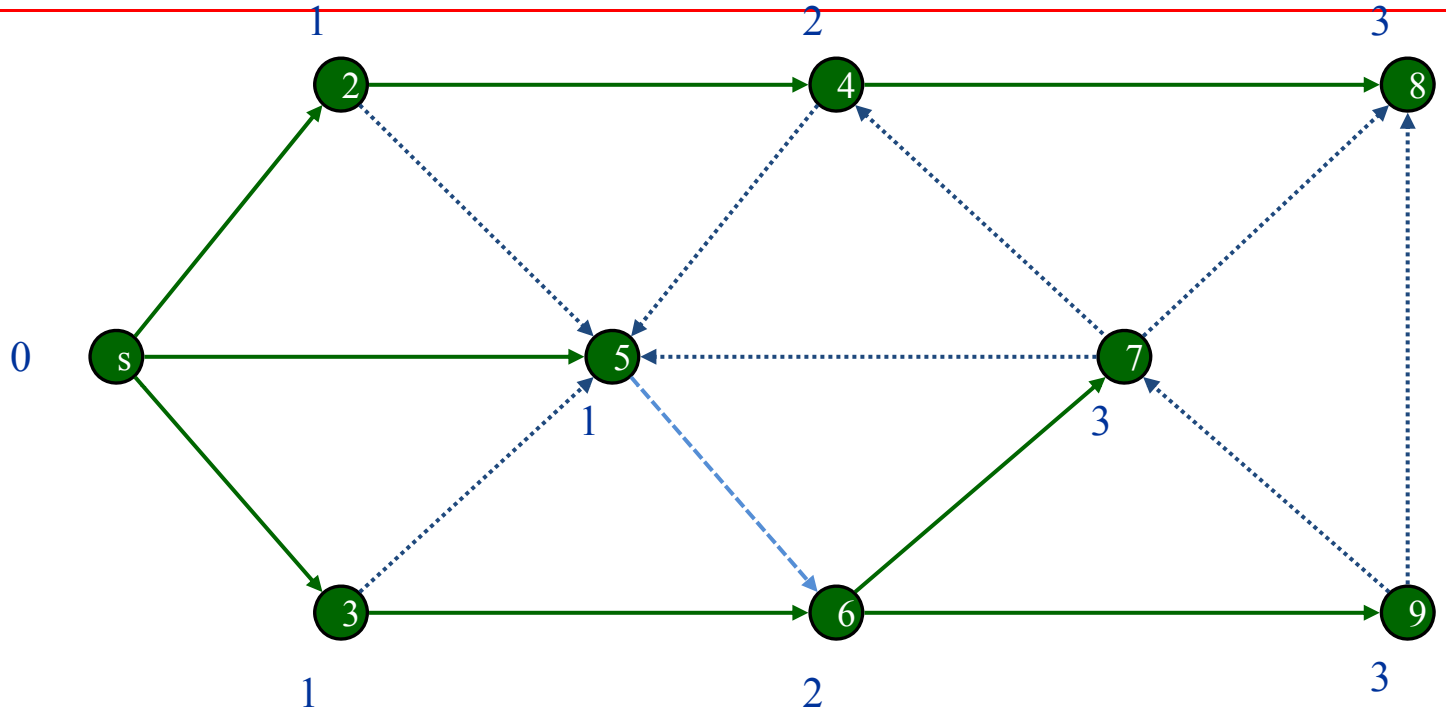
Discovered

Top of queue

Finished

Queue: 9

Breadth First Search



Undiscovered

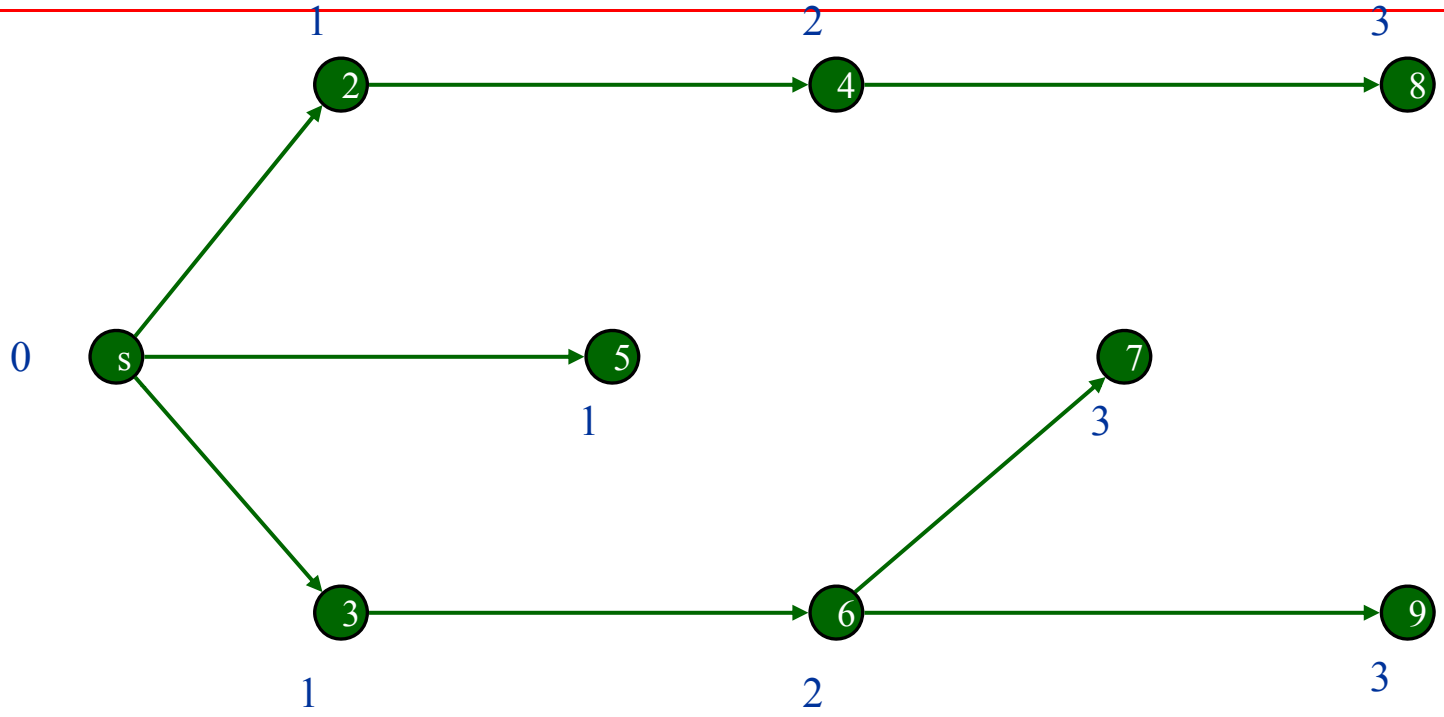
Discovered

Top of queue

Finished

Queue:

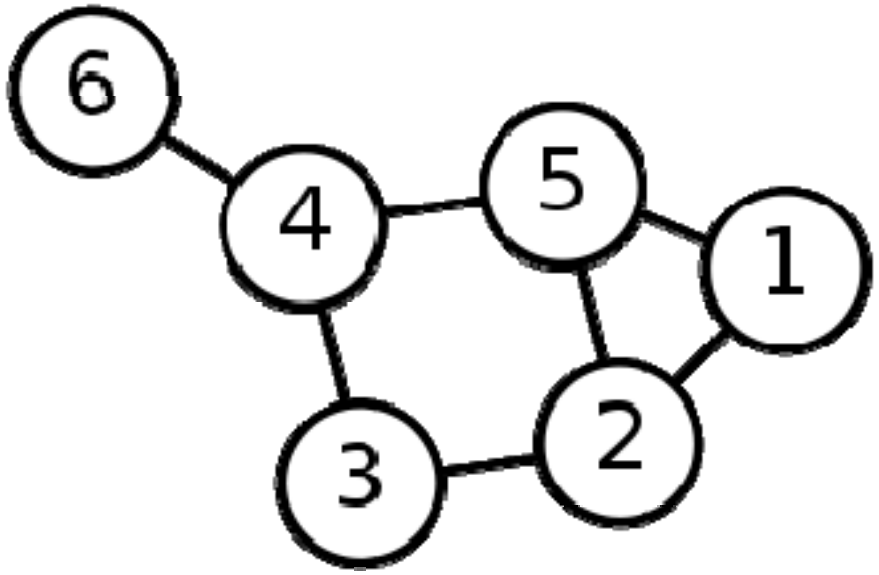
Breadth First Search



BFS Tree

Single Source Shortest Path Problem

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



Dijkstra's Algorithm

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.

Approach: Greedy : makes local optimum choice in each step hoping to reach global optimum.

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

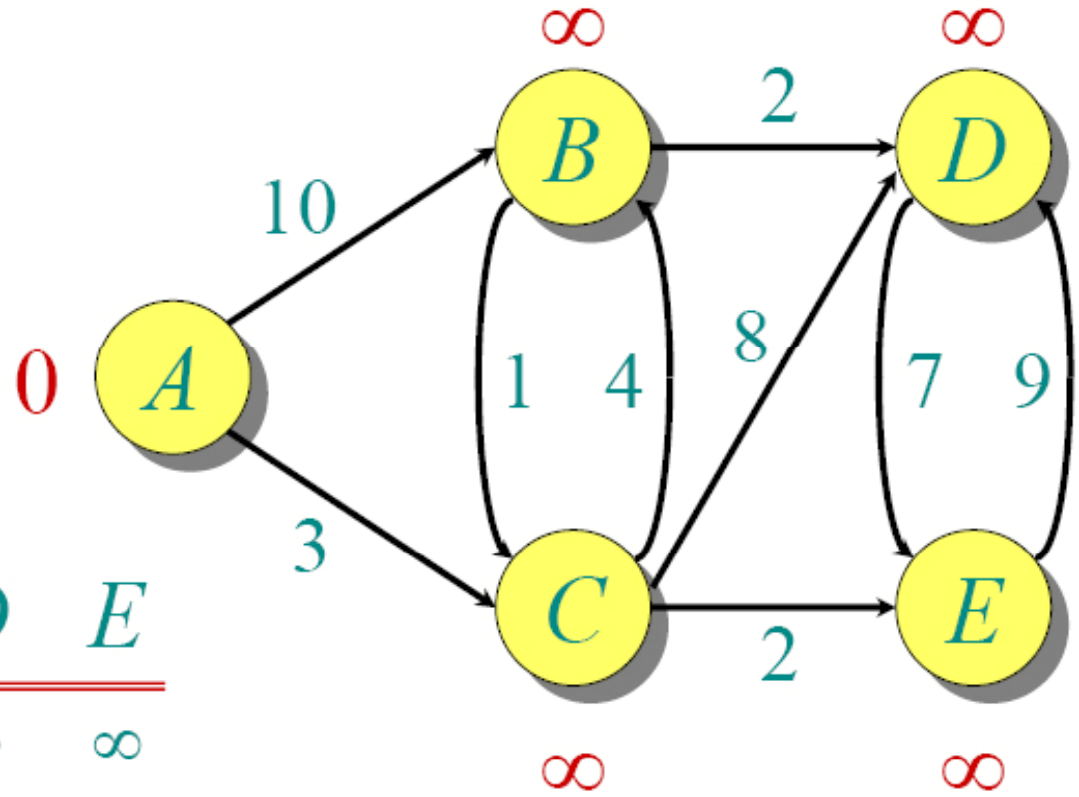
Dijkstra's Algorithm

```
dist[s] ←0                                  (distance to source vertex is zero)
for  all v ∈ V–{s}
      do  dist[v] ←∞                        (set all other distances to infinity)
S←∅                                         (S, the set of visited vertices is initially empty)
Q←V                                         (Q, the queue initially contains all vertices)
while Q ≠∅                                  (while the queue is not empty)
do   u ← mindistance(Q,dist)                (select the element of Q with the min. distance)
     S←S∪{u}                                (add u to list of visited vertices)
     for all v ∈ neighbors[u]
          do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                then     d[v] ←d[u] + w(u, v)   (set new value of shortest path)
                   return dist
```

Total running time: $O(n^2)$

Dijkstra's Algorithm

Initialize:

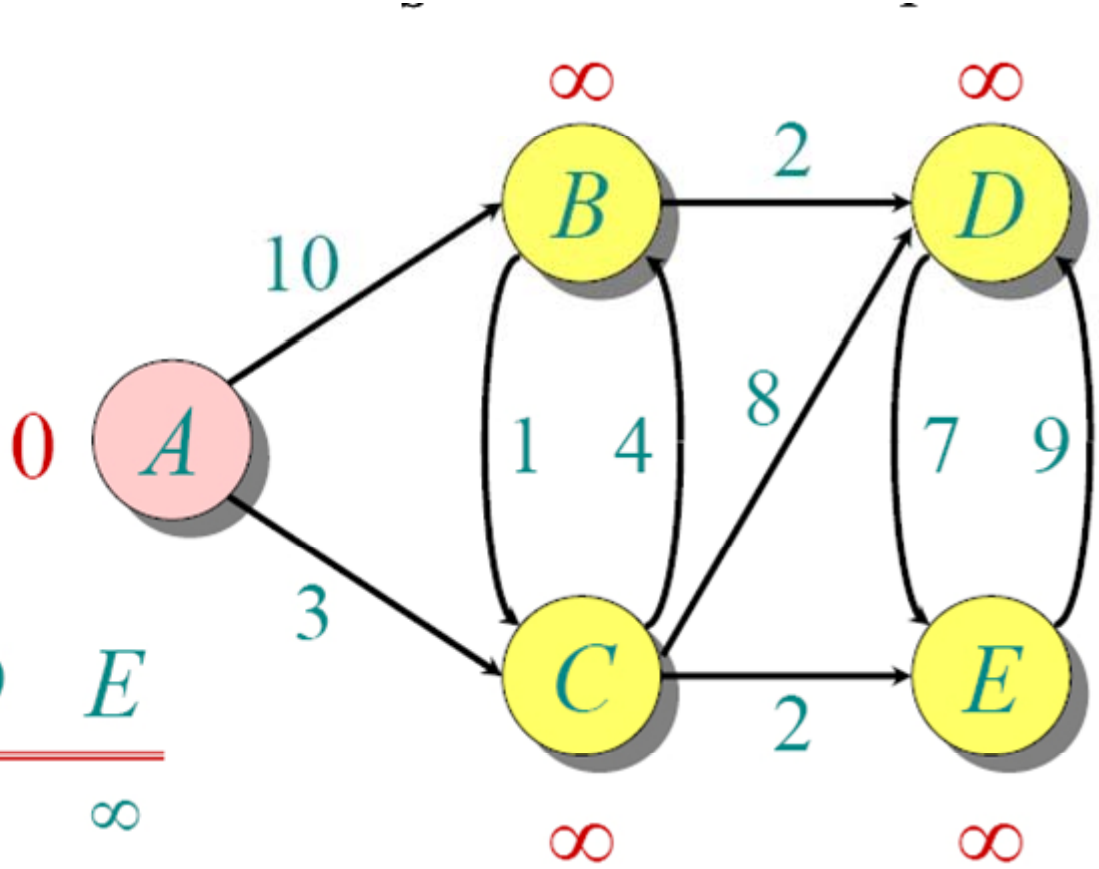


Q: A B C D E

0	∞	∞	∞	∞

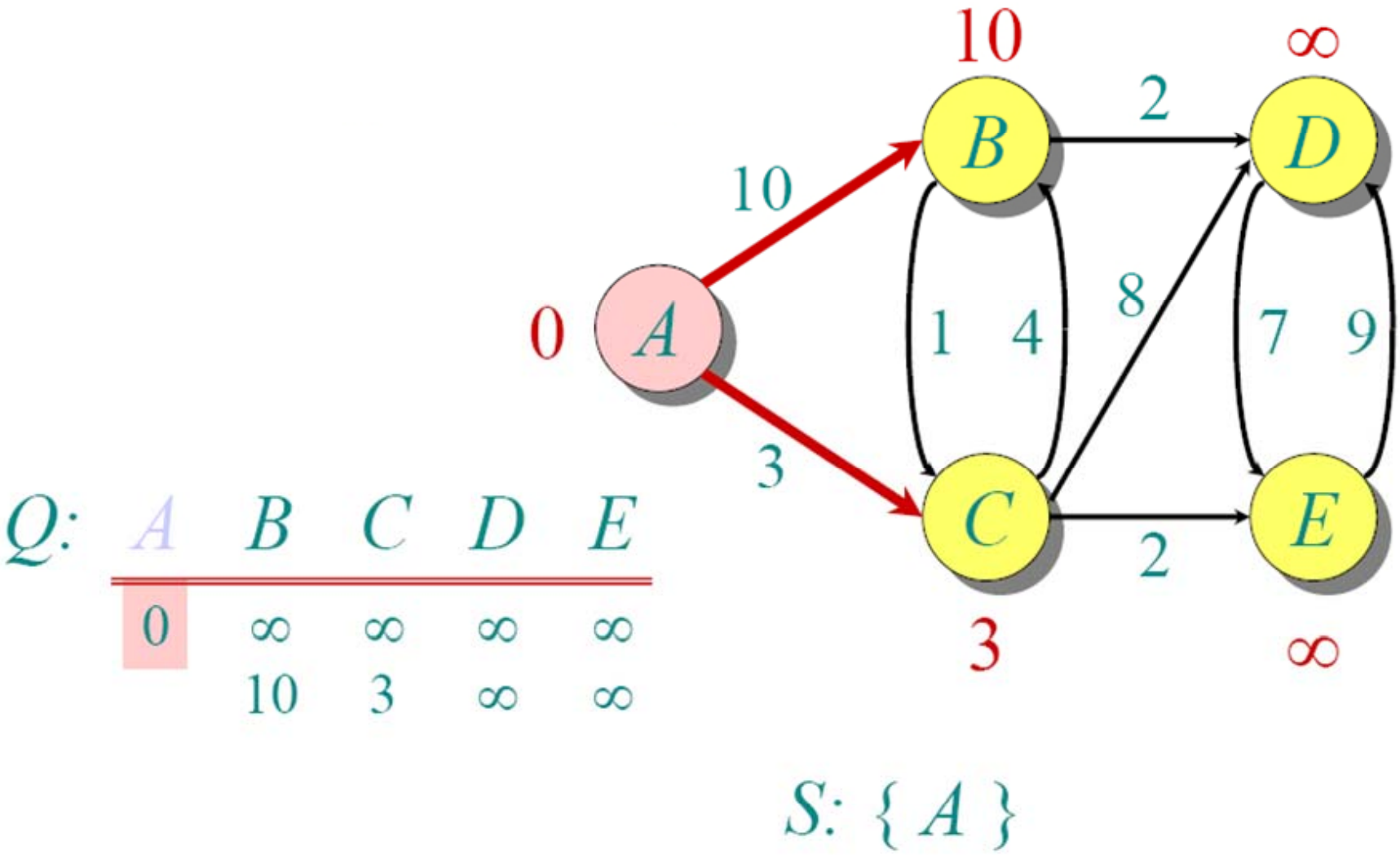
S: {}

Dijkstra's Algorithm

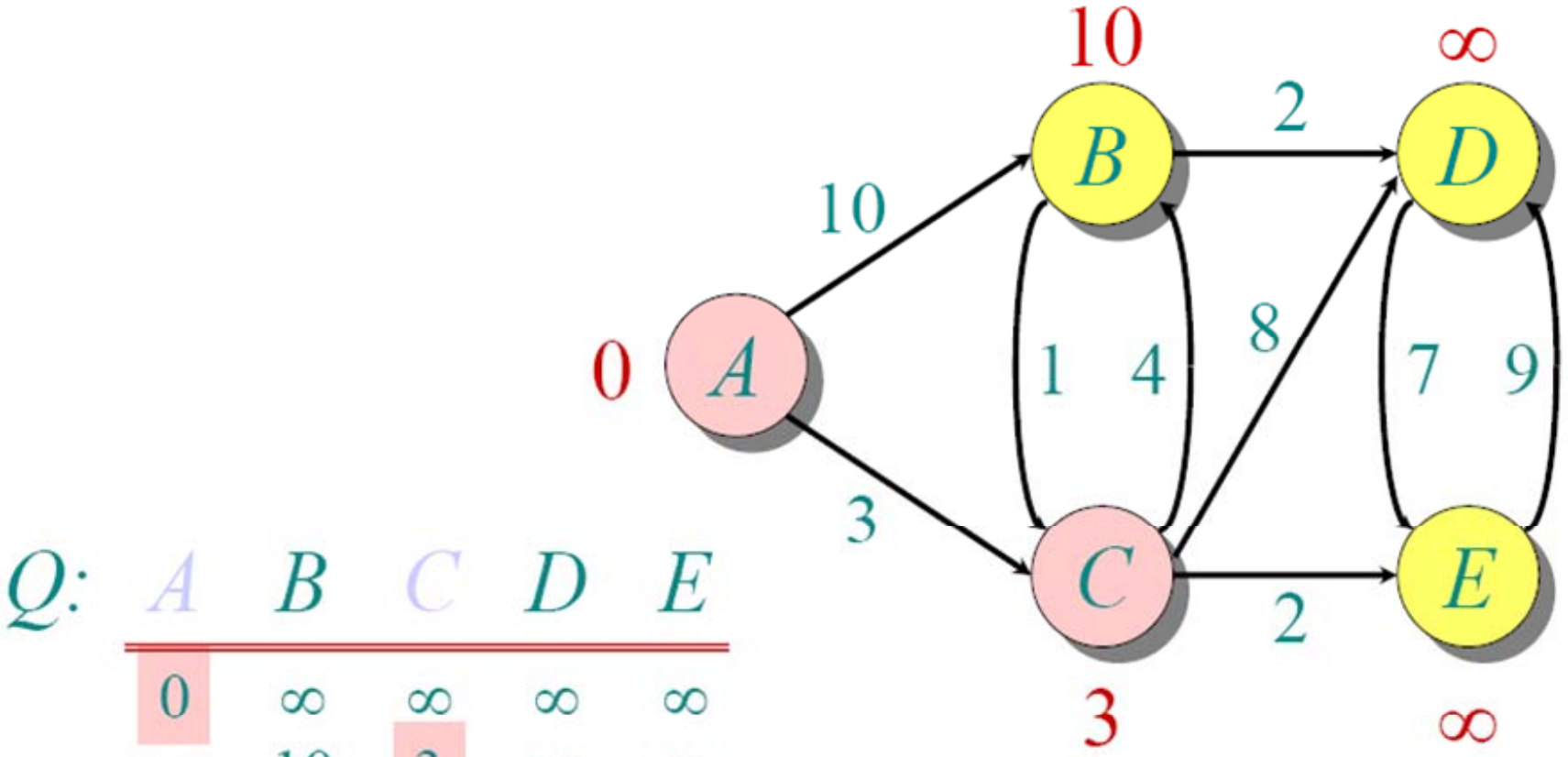


Q:	A	B	C	D	E
	0	∞	∞	∞	∞

Dijkstra's Algorithm



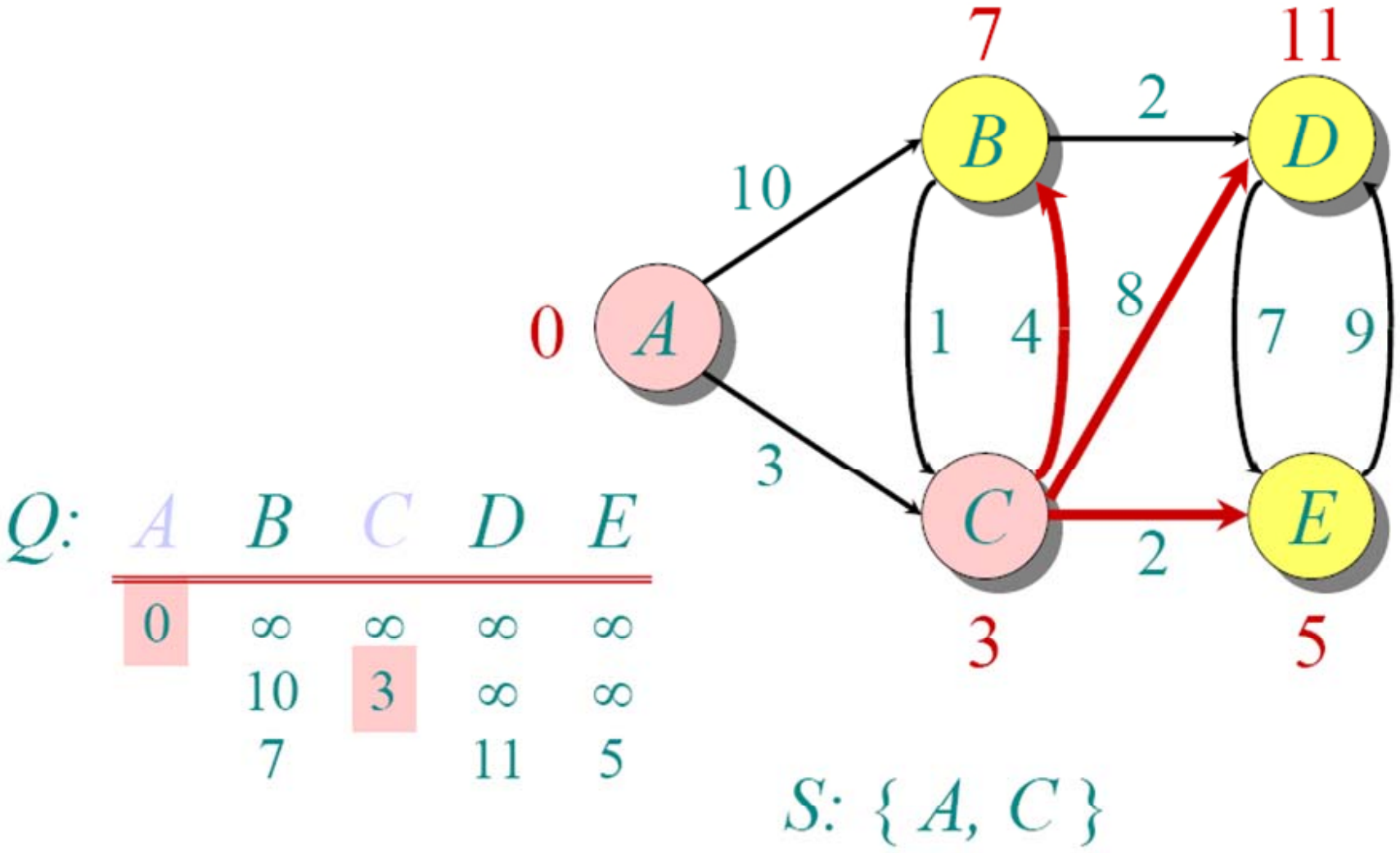
Dijkstra's Algorithm



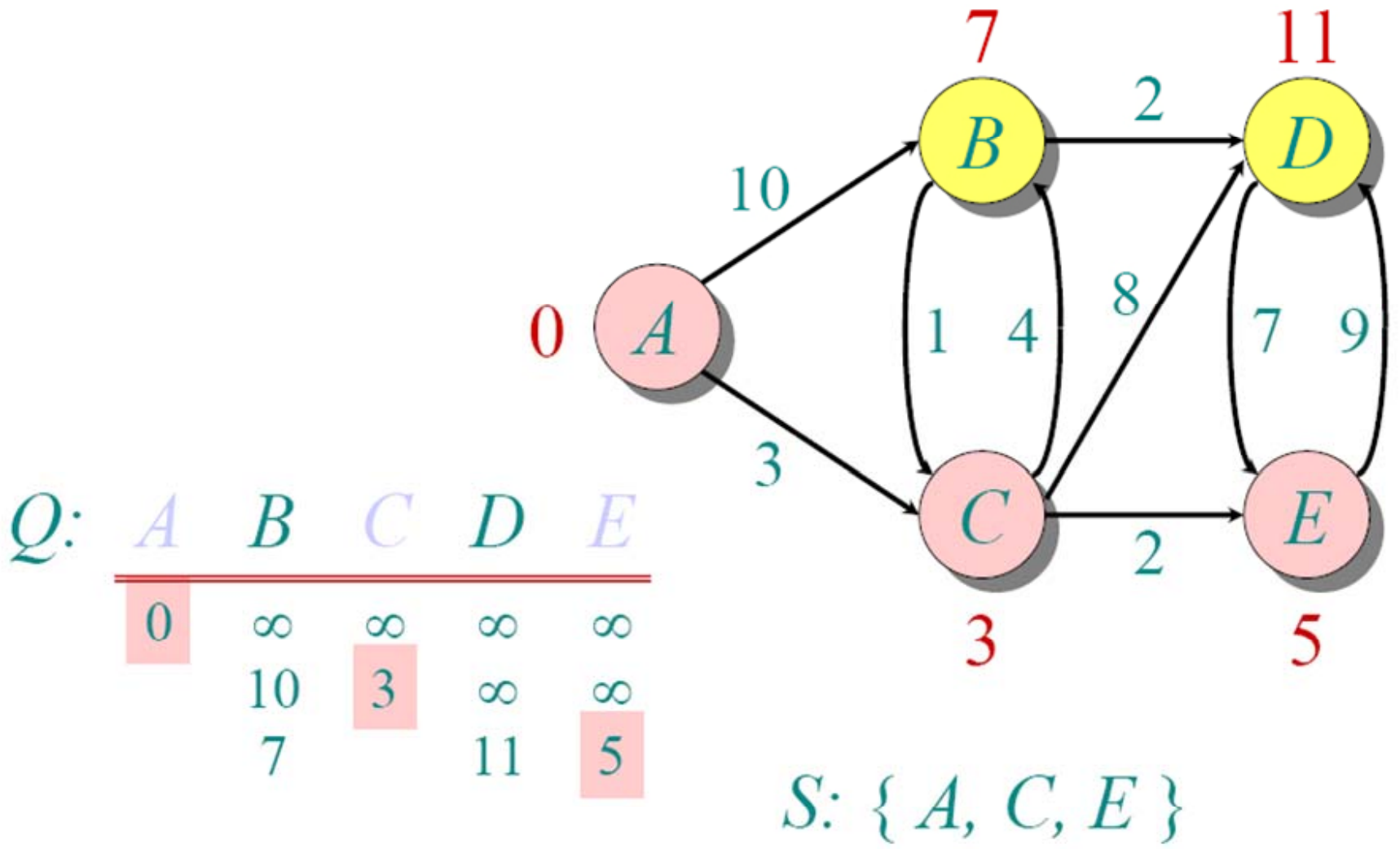
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A, C \}$

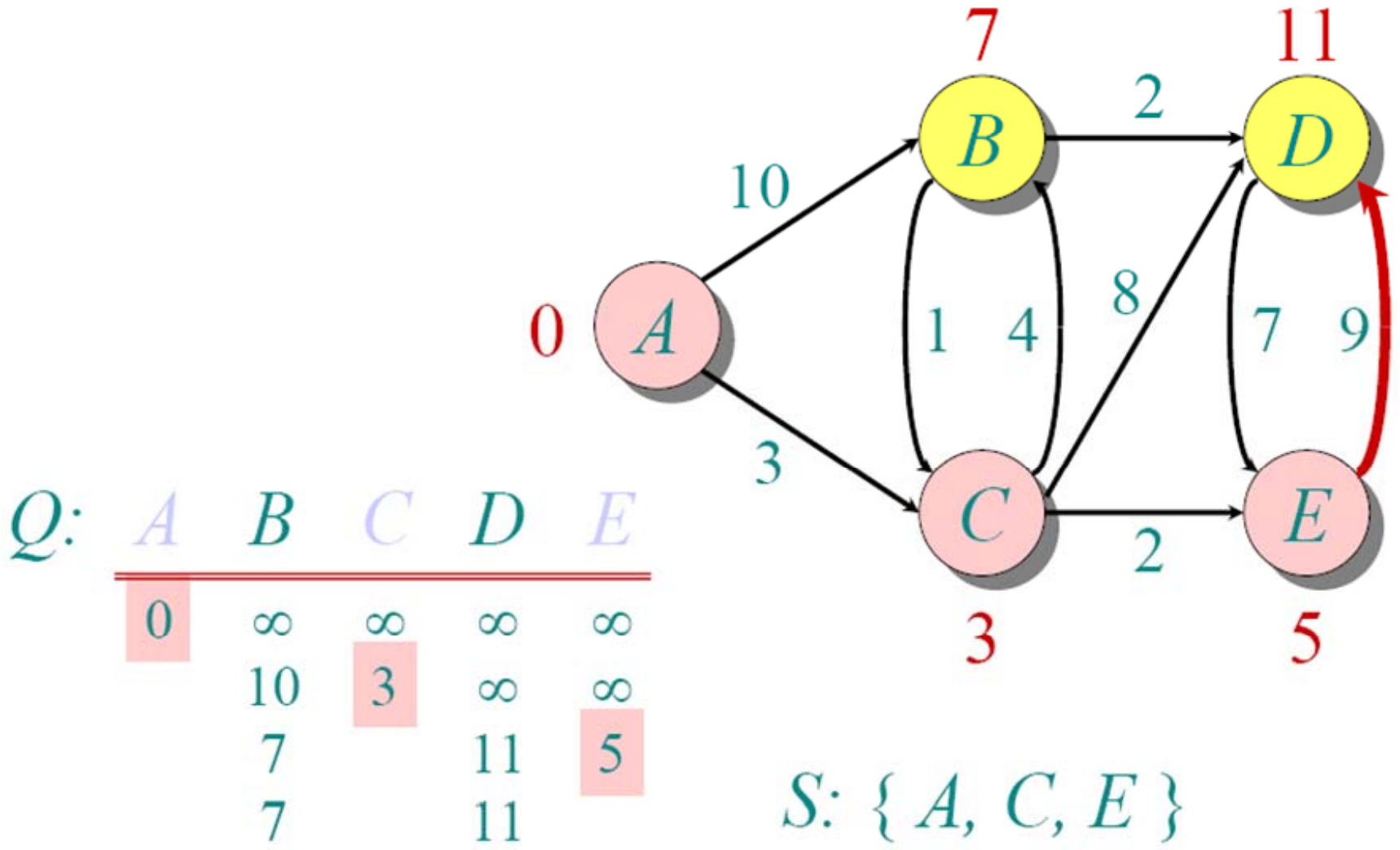
Dijkstra's Algorithm



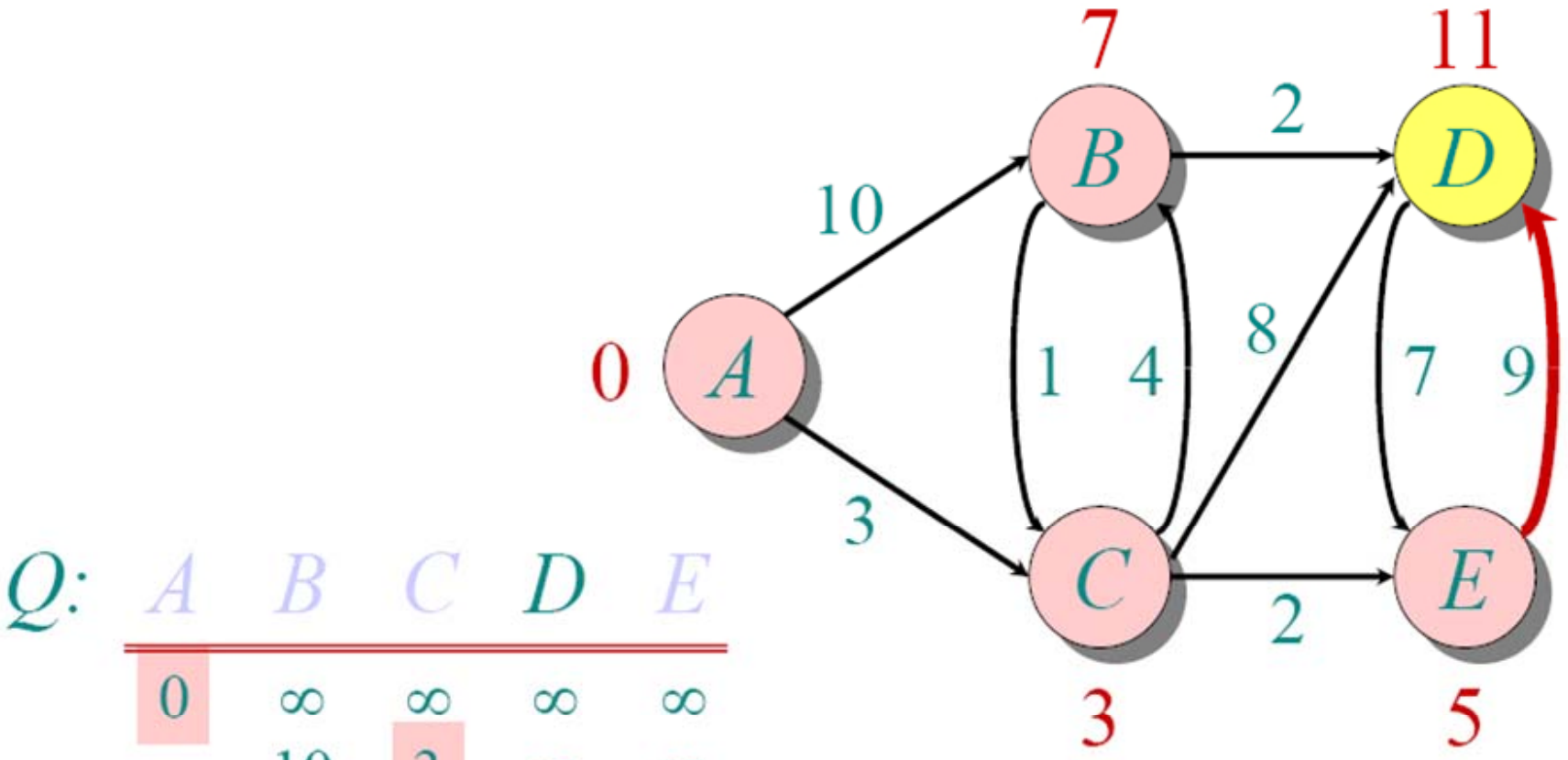
Dijkstra's Algorithm



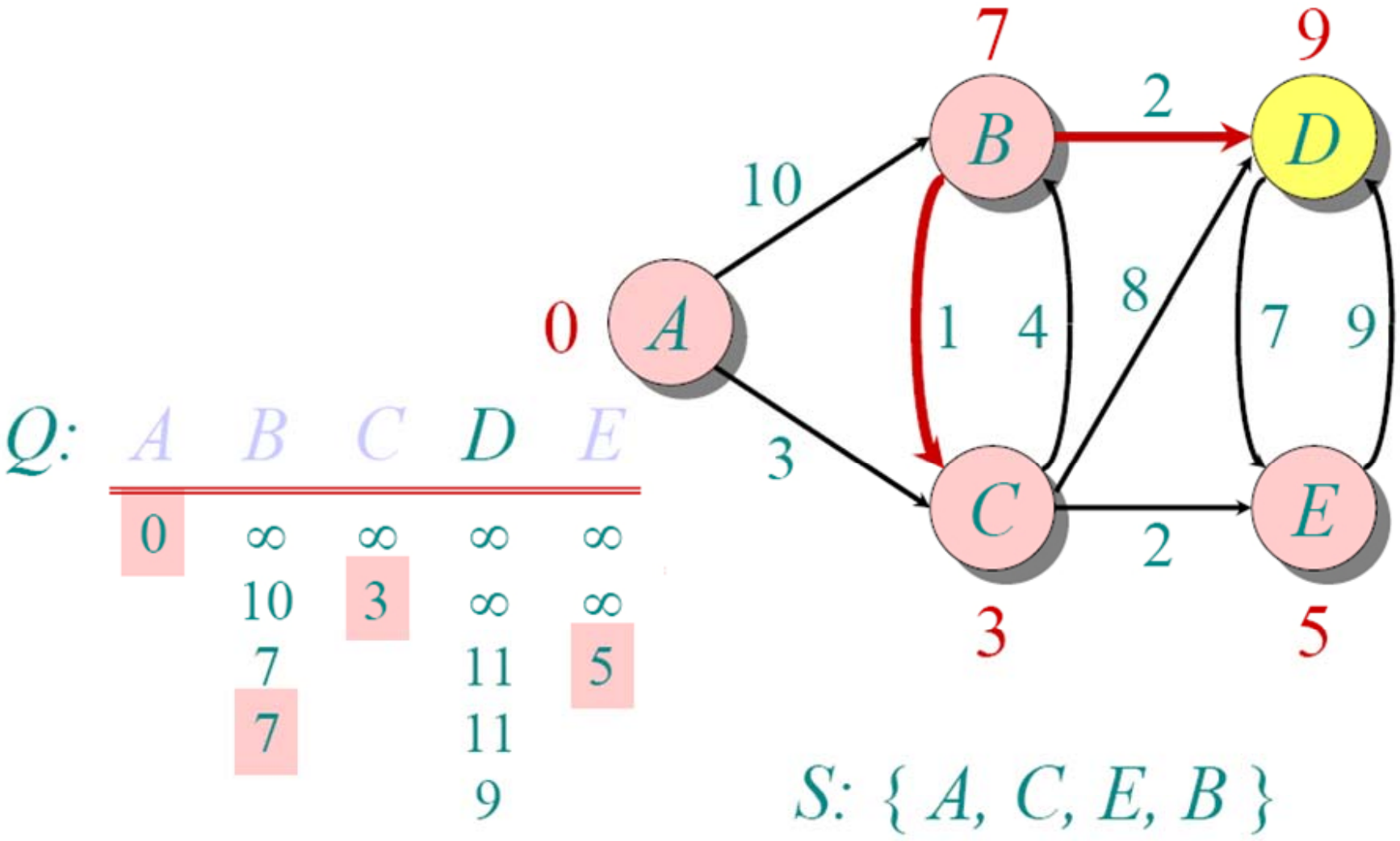
Dijkstra's Algorithm



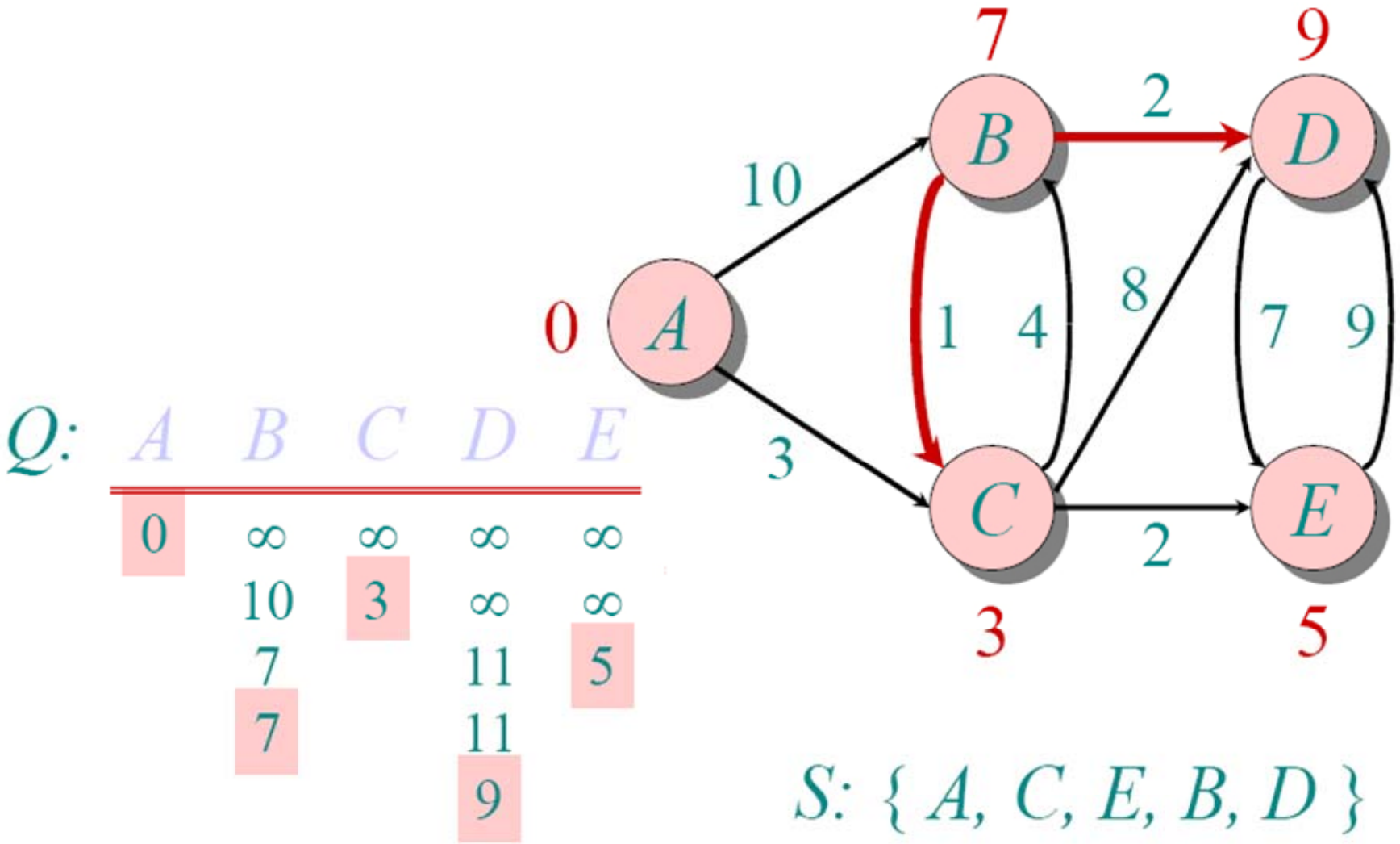
Dijkstra's Algorithm



Dijkstra's Algorithm



Dijkstra's Algorithm



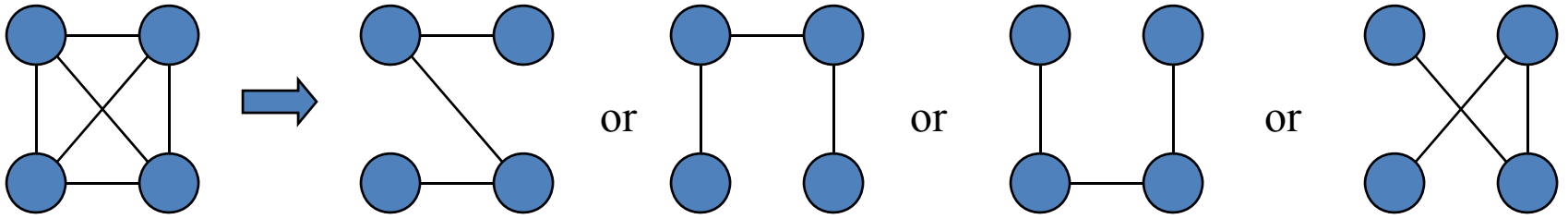
Spanning Trees

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

A graph may have many spanning trees.

Graph A

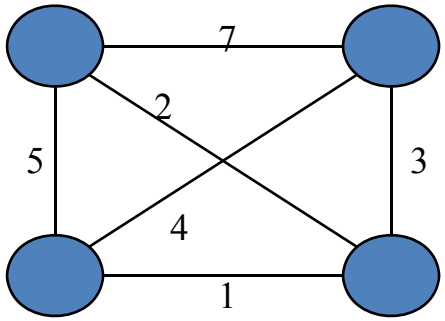
Some Spanning Trees from Graph A



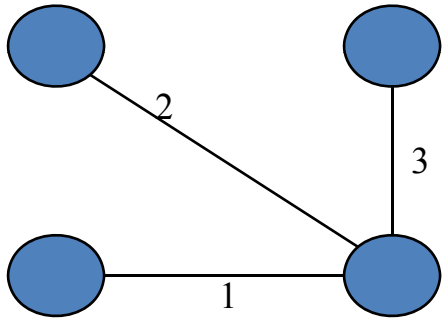
Minimum Spanning Trees

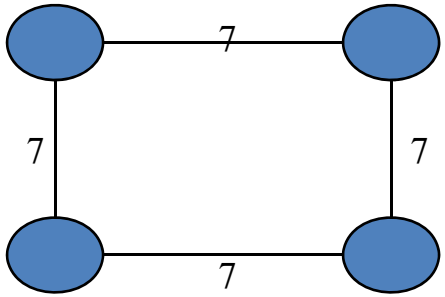
The Minimum Spanning Tree for a given graph is the Spanning Tree of minimum cost for that graph.

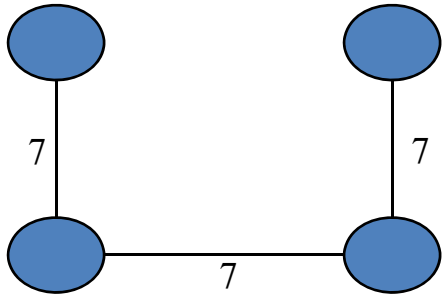
Graph



Minimum Spanning Tree







Algorithms for Obtaining the Minimum Spanning Tree

- Kruskal's Algorithm
- Prim'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.

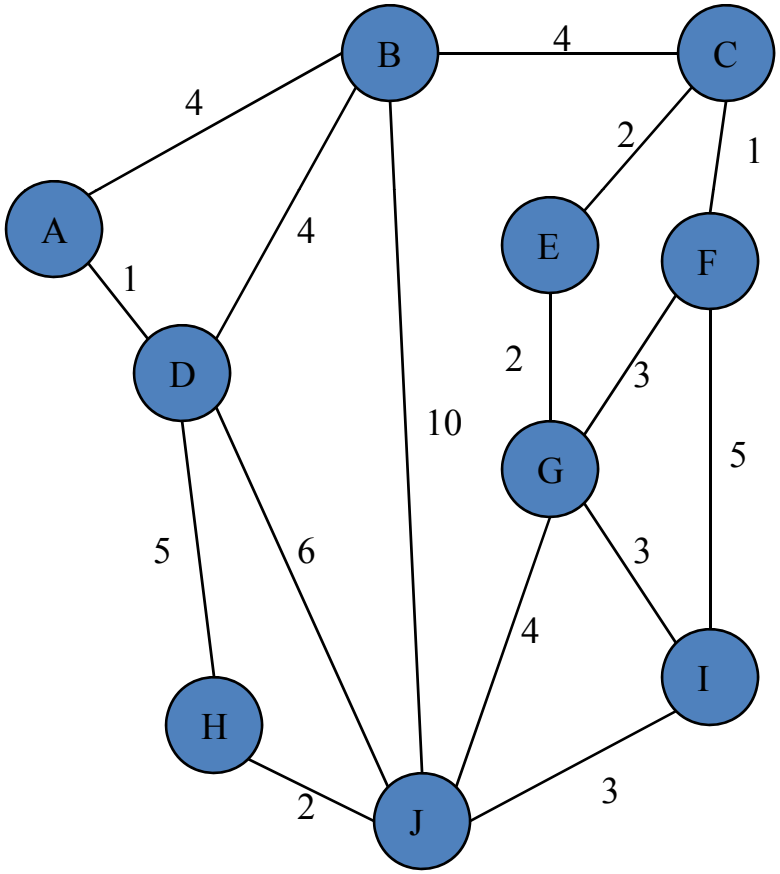
Kruskal's Algorithm

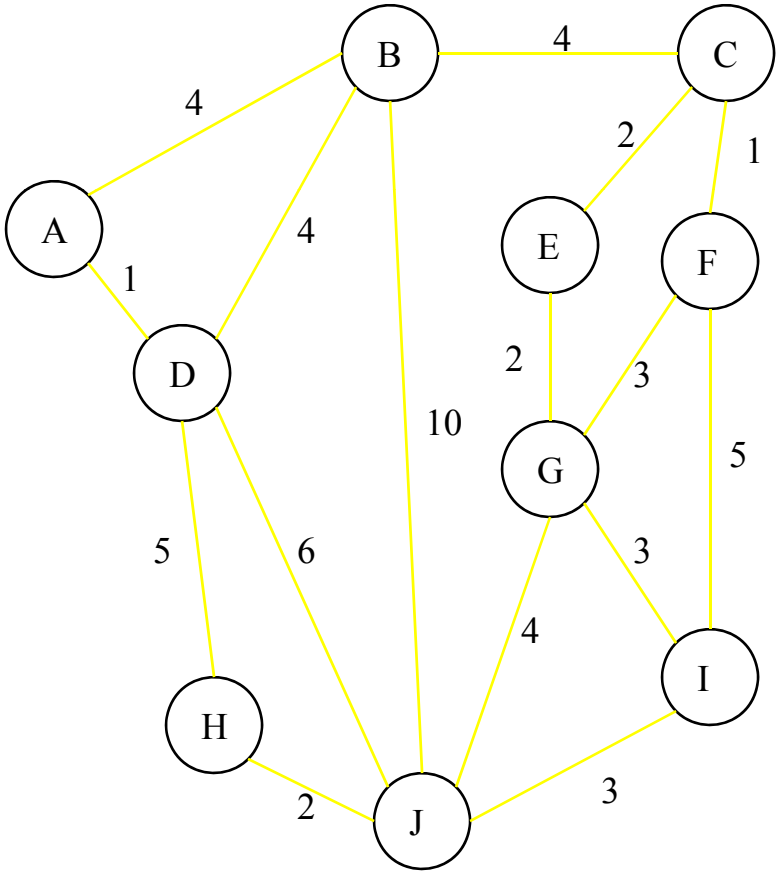
The steps are:

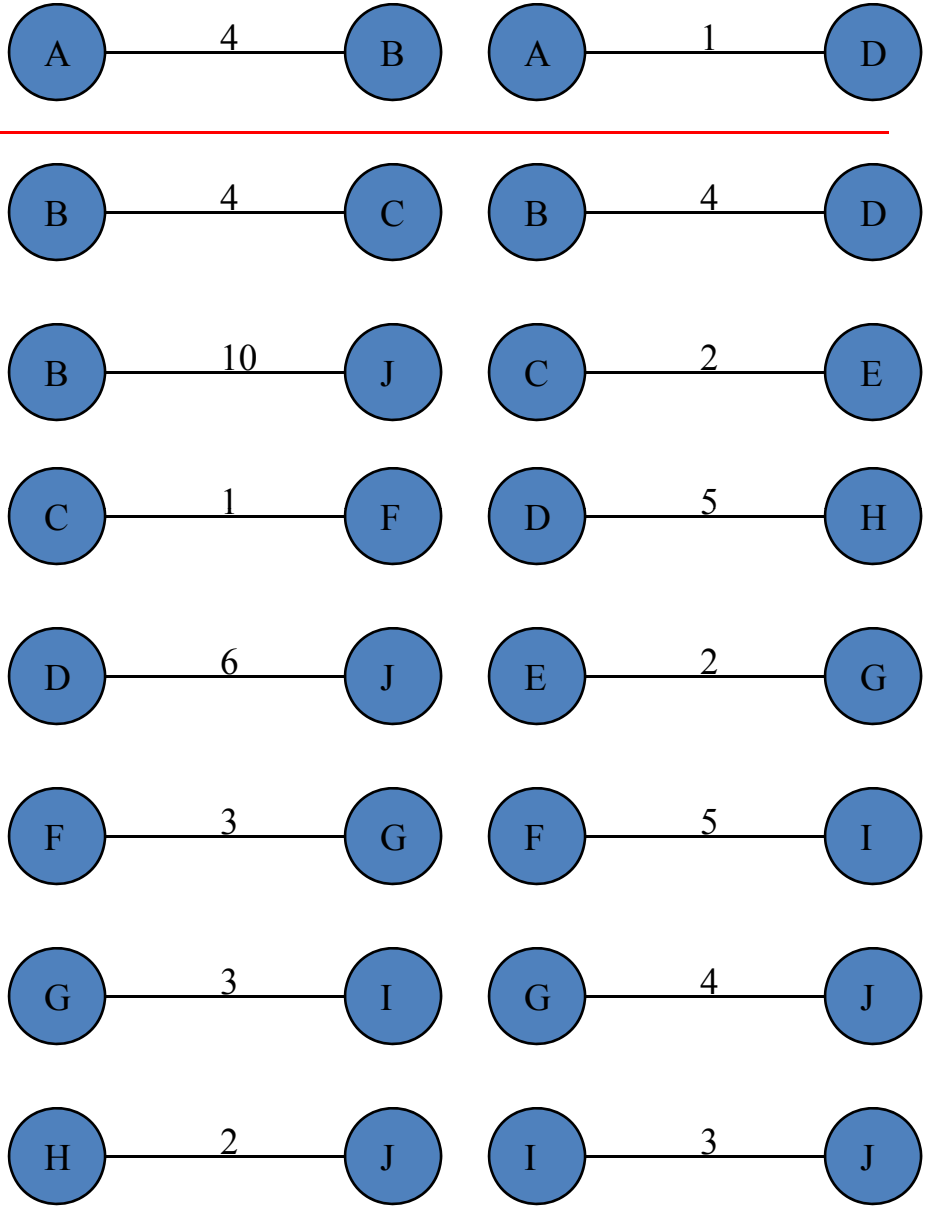
1. The forest is constructed - with each node in a separate tree.
2. Sort the edges.
3. Until we've added n-1 edges,
    3.1. Extract the next cheapest edge.
    3.2. If it forms a cycle, reject it.
    3.3. Else add it to the forest. Adding it to the forest will join two trees together.

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

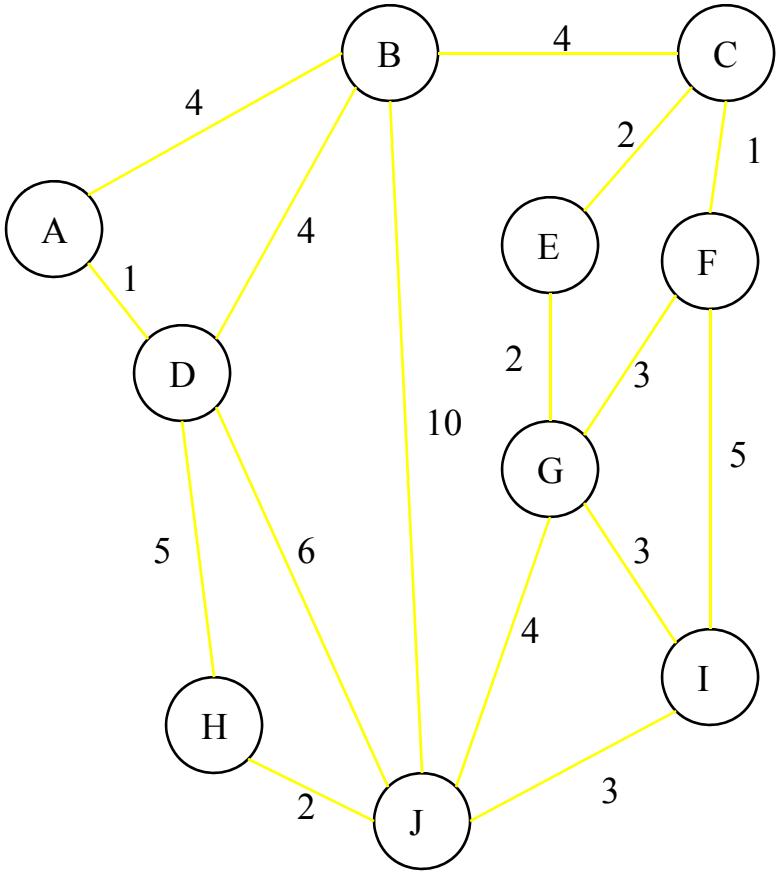
Example Graph

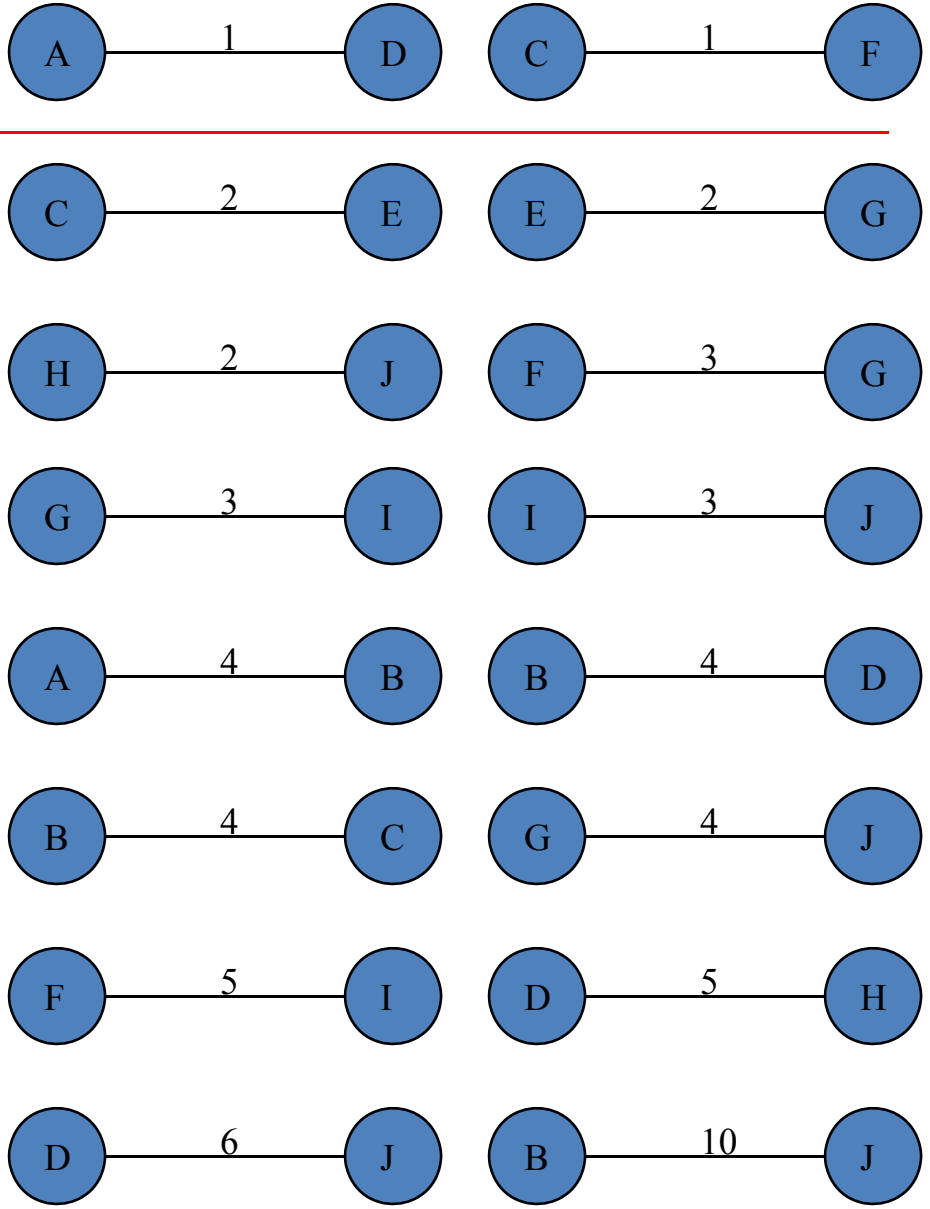



A
B
C
D
E
F
G
H
I
J
4
4
4
1
2
1
10
2
3
5
5
6
3
4
3
2
3

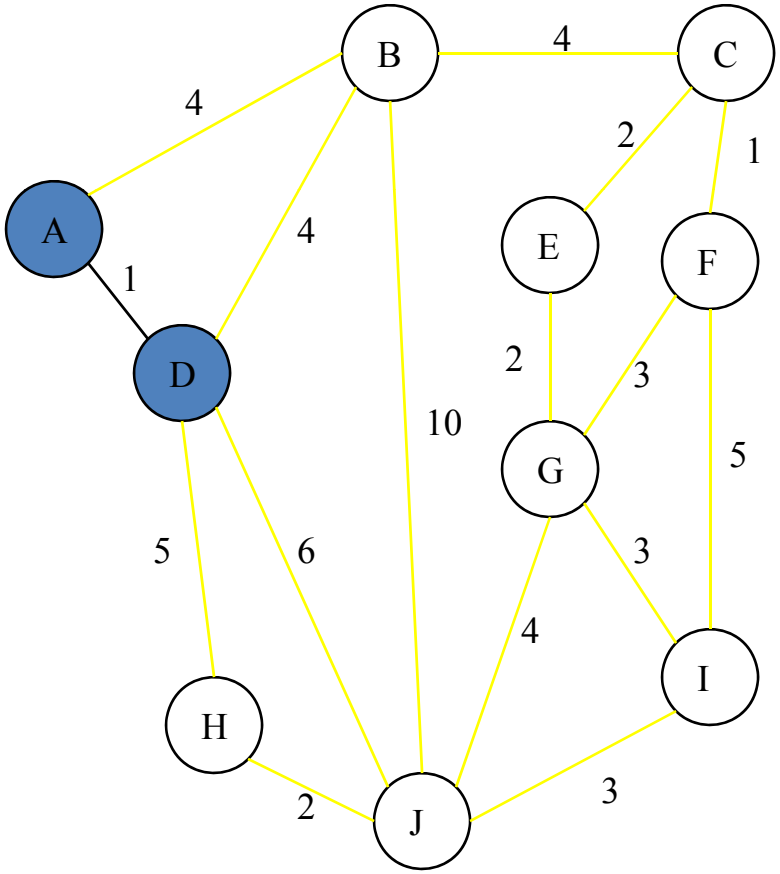

A 4 B
A 1 D
B 4 C
B 4 D
B 10 J
C 2 E
C 1 F
D 5 H
D 6 J
E 2 G
F 3 G
F 5 I
G 3 I
G 4 J
H 2 J
I 3 J

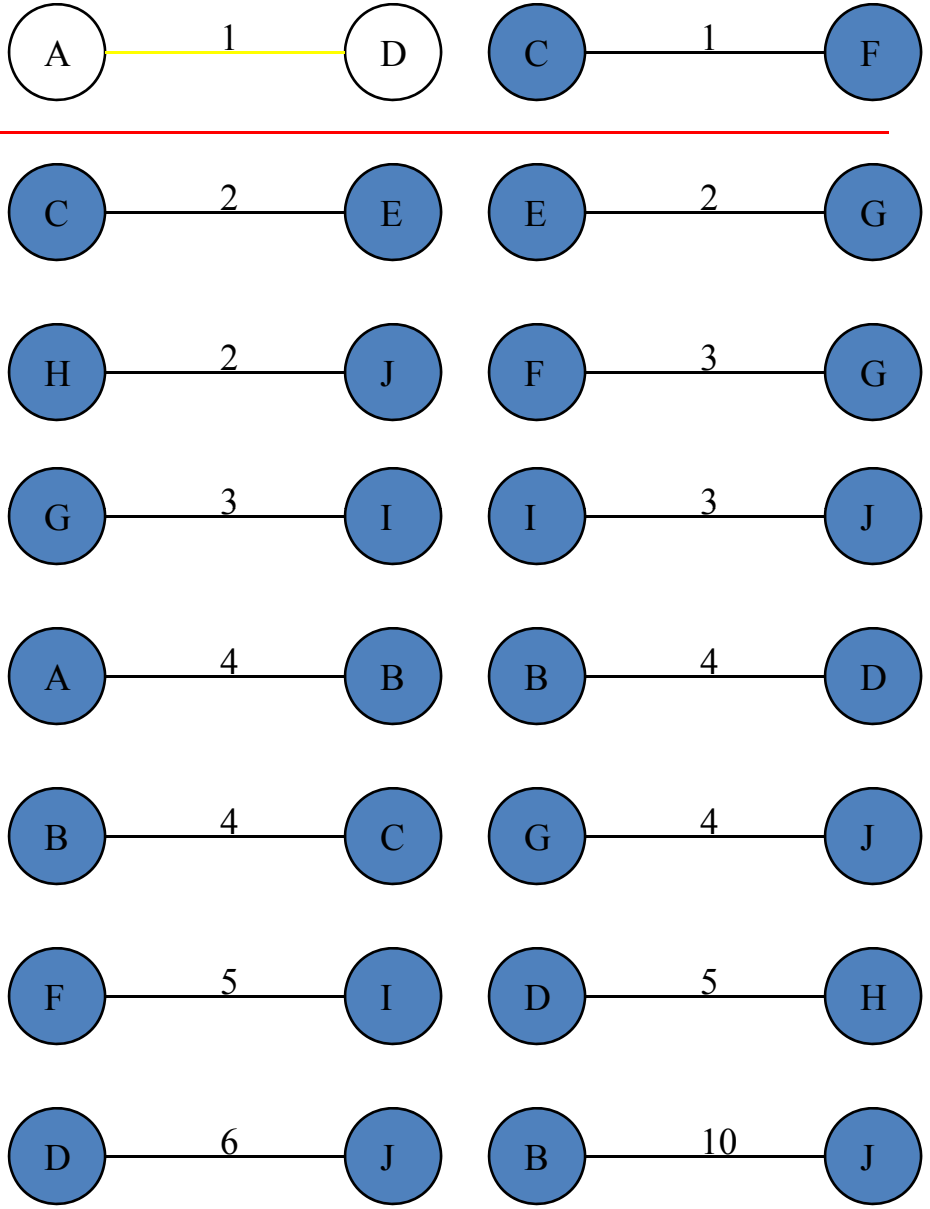
Sort Edges


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

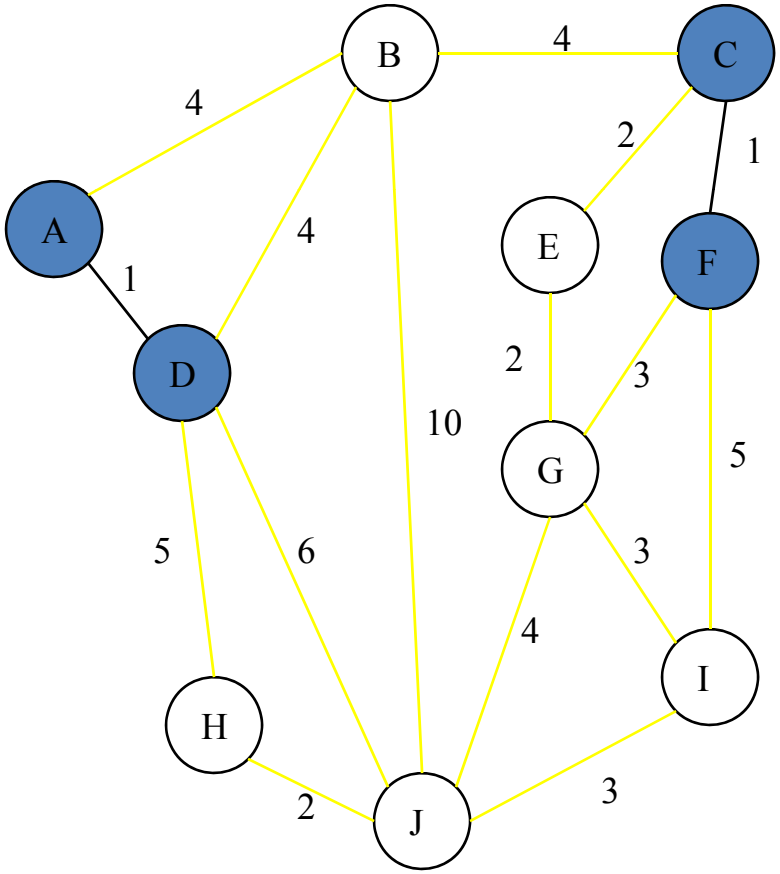

A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

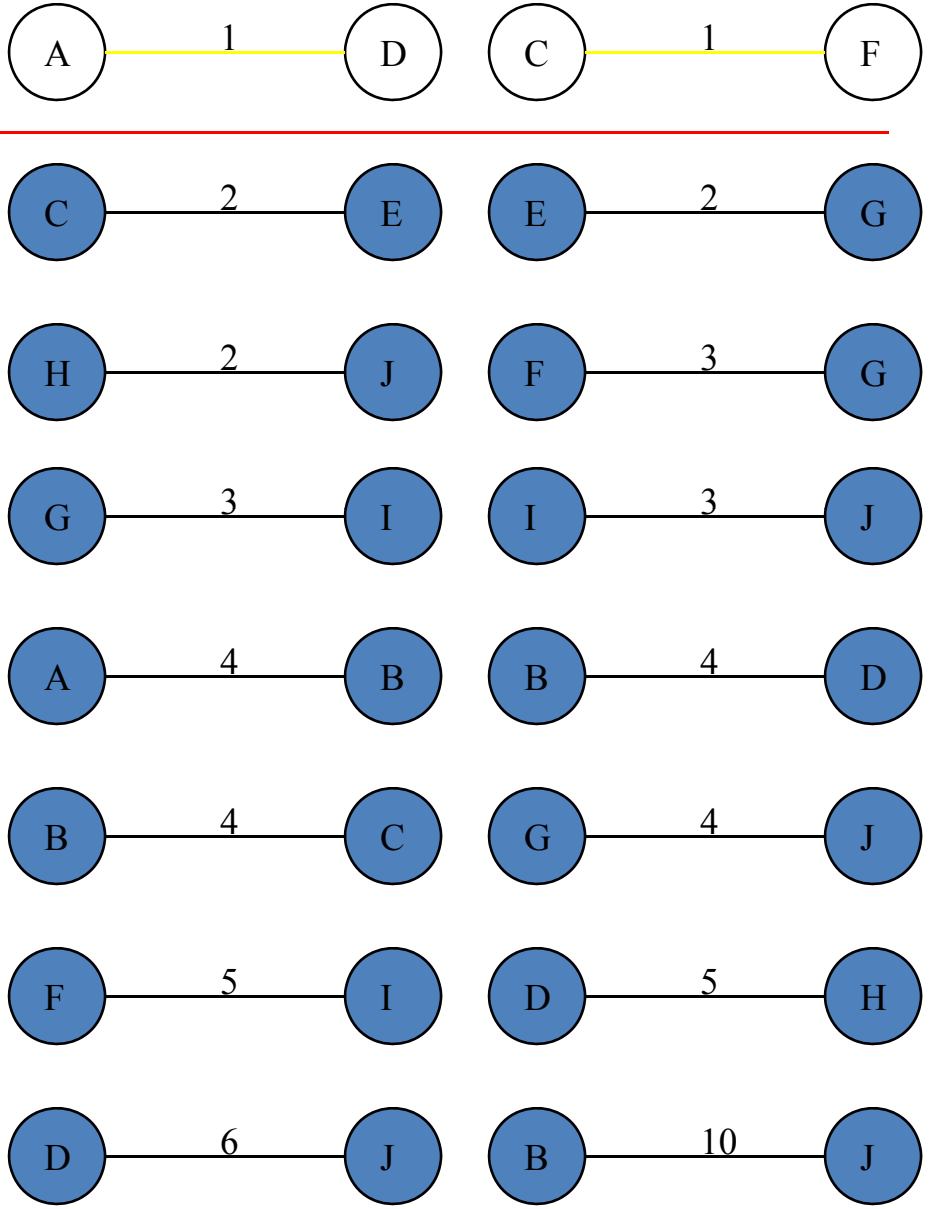
Add Edge


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

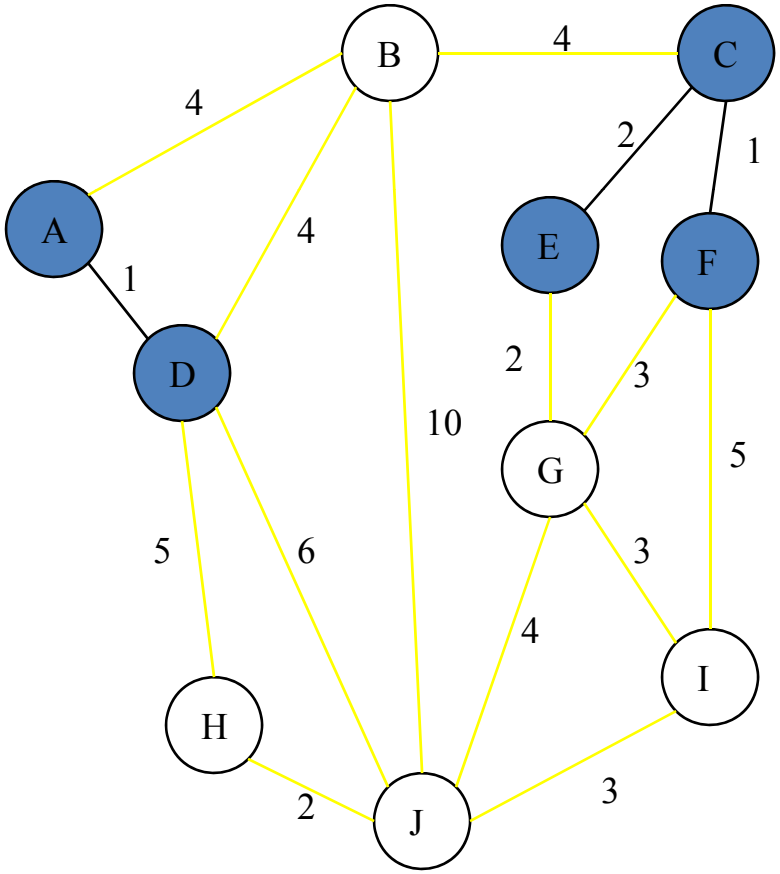

A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

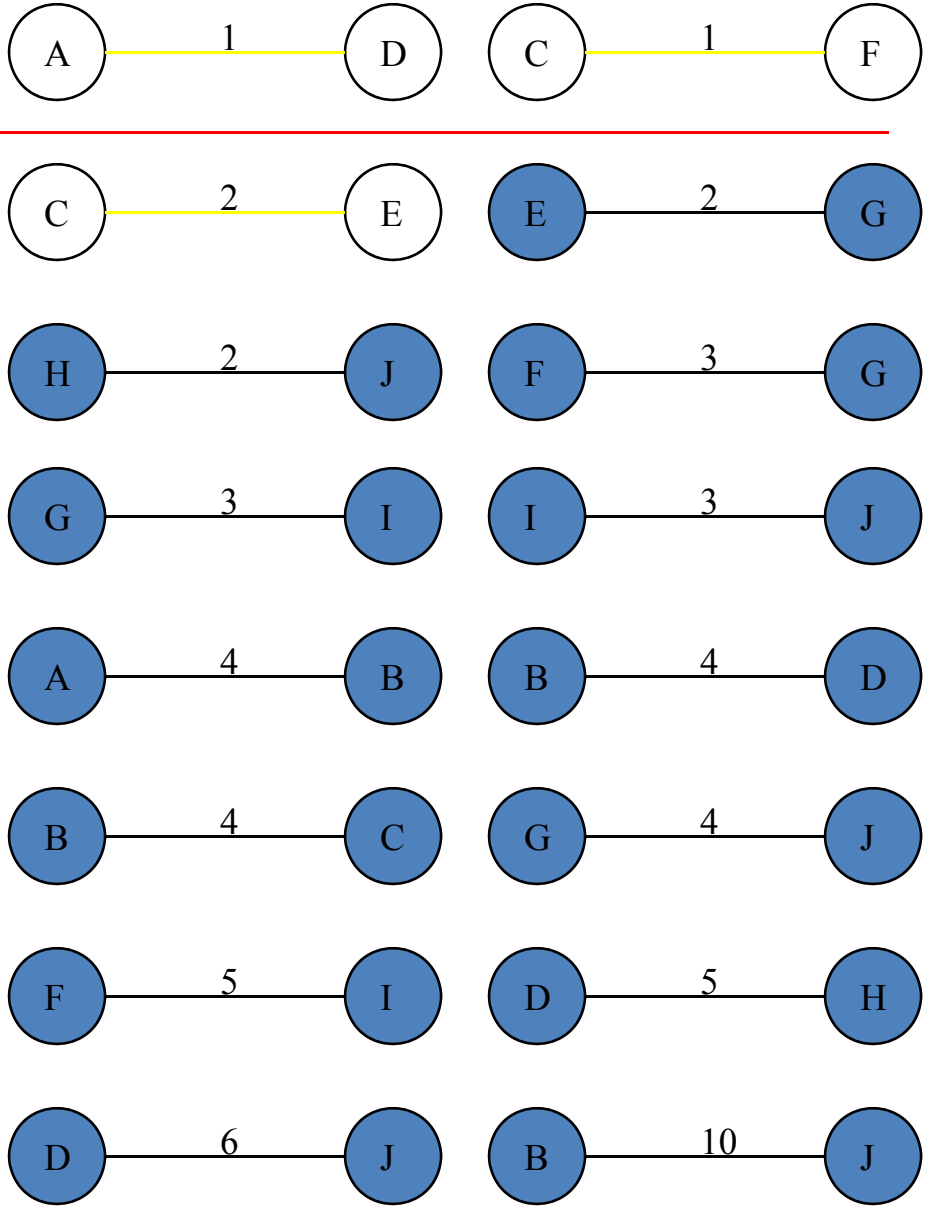
Add Edge


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

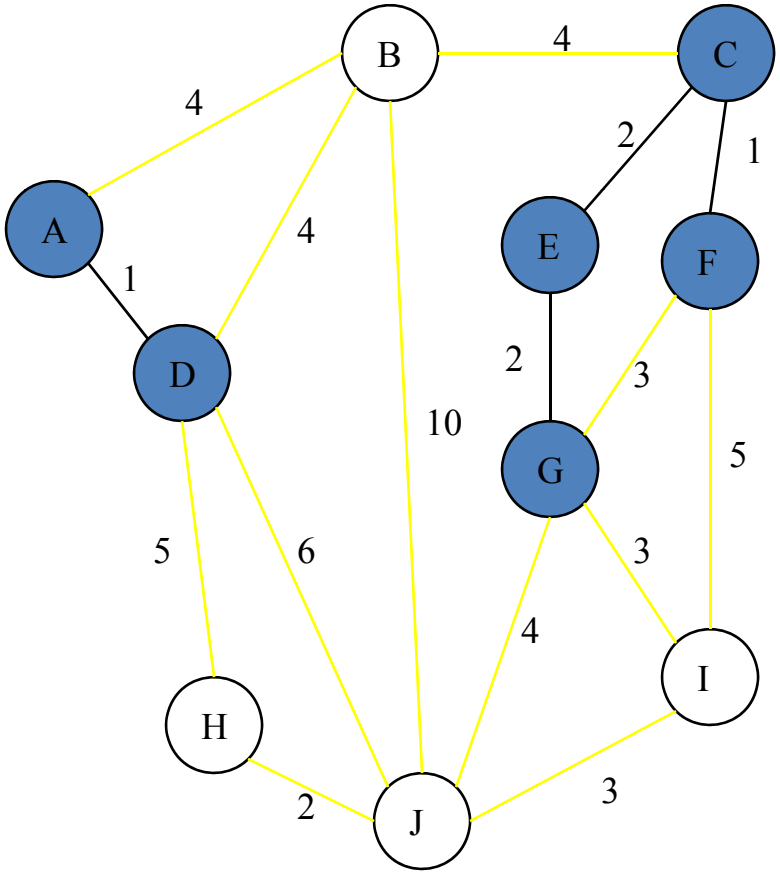

A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

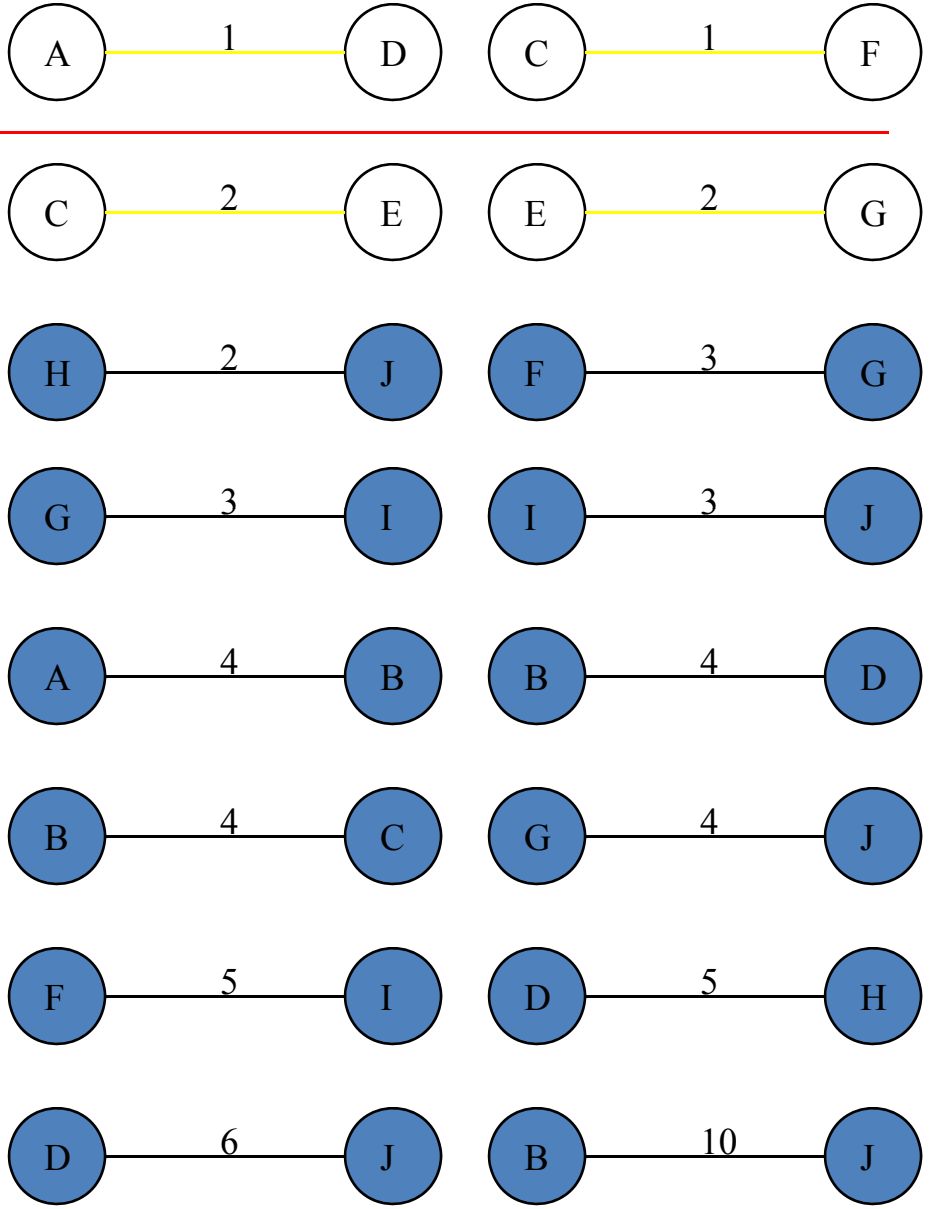
Add Edge


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

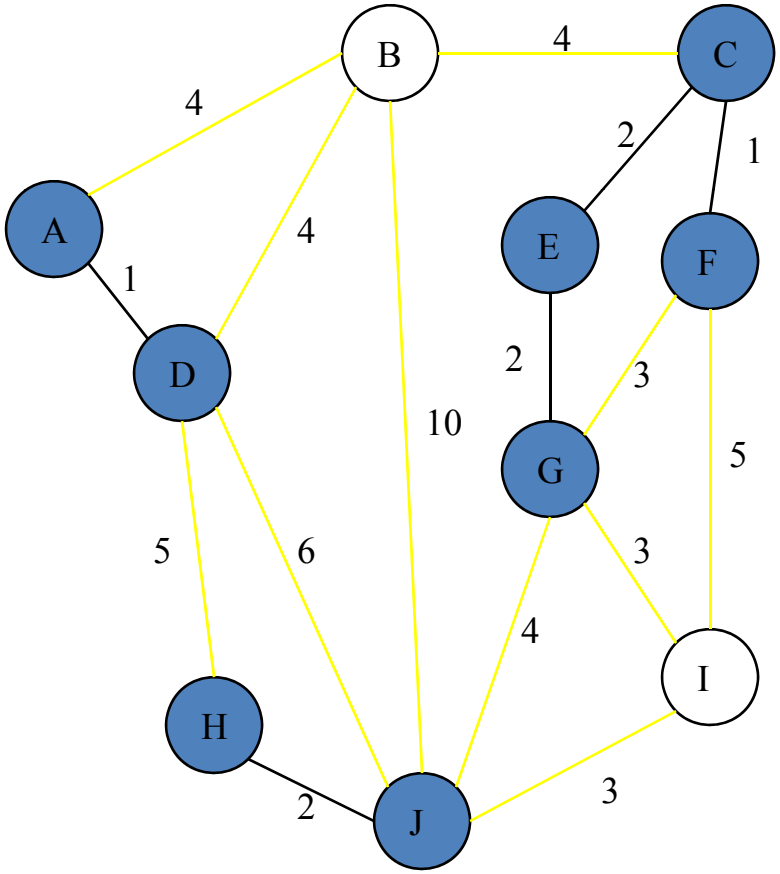

A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

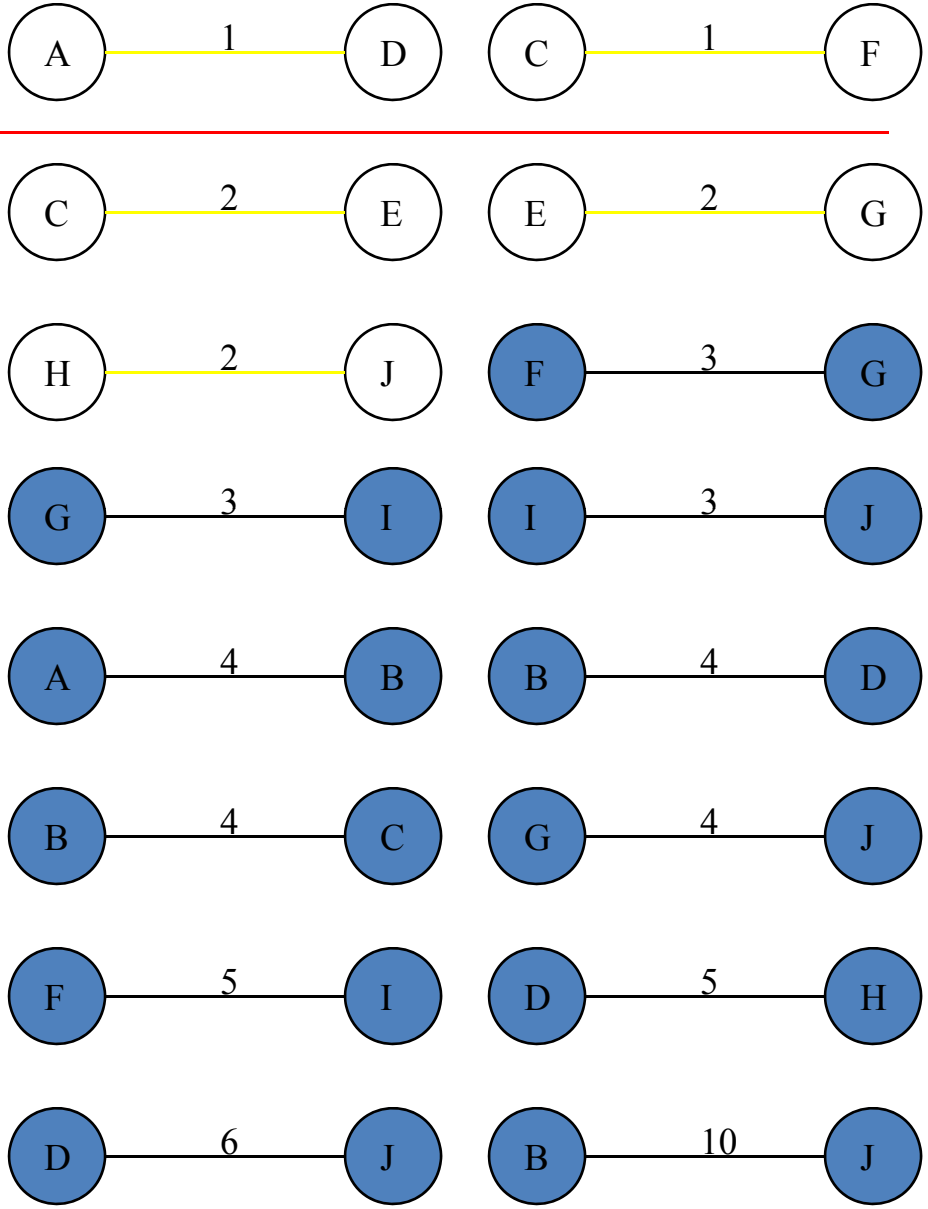
Add Edge


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

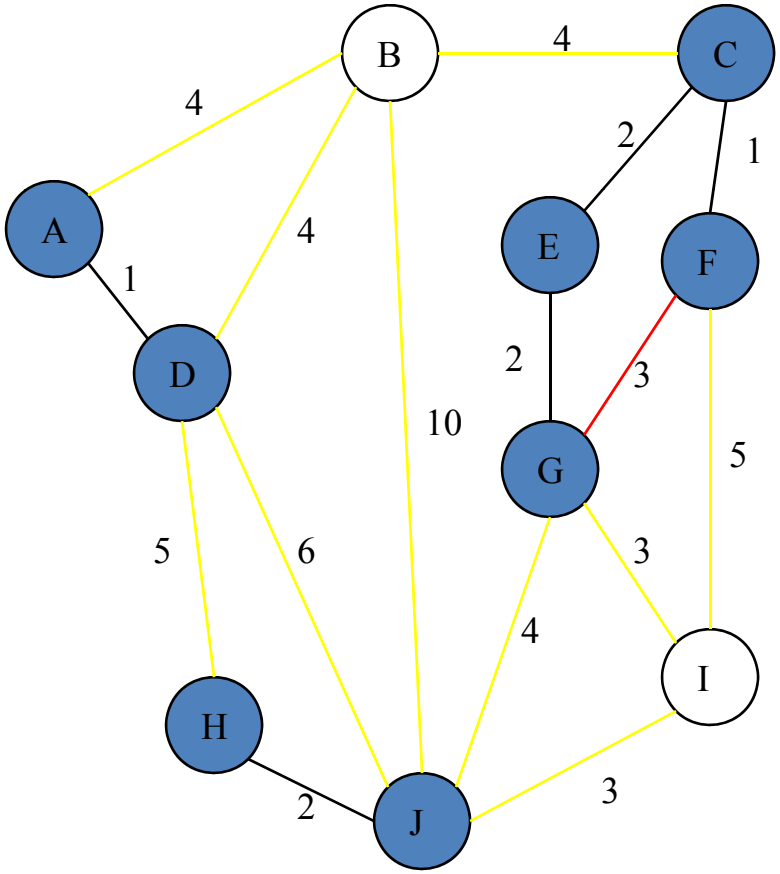
Add Edge

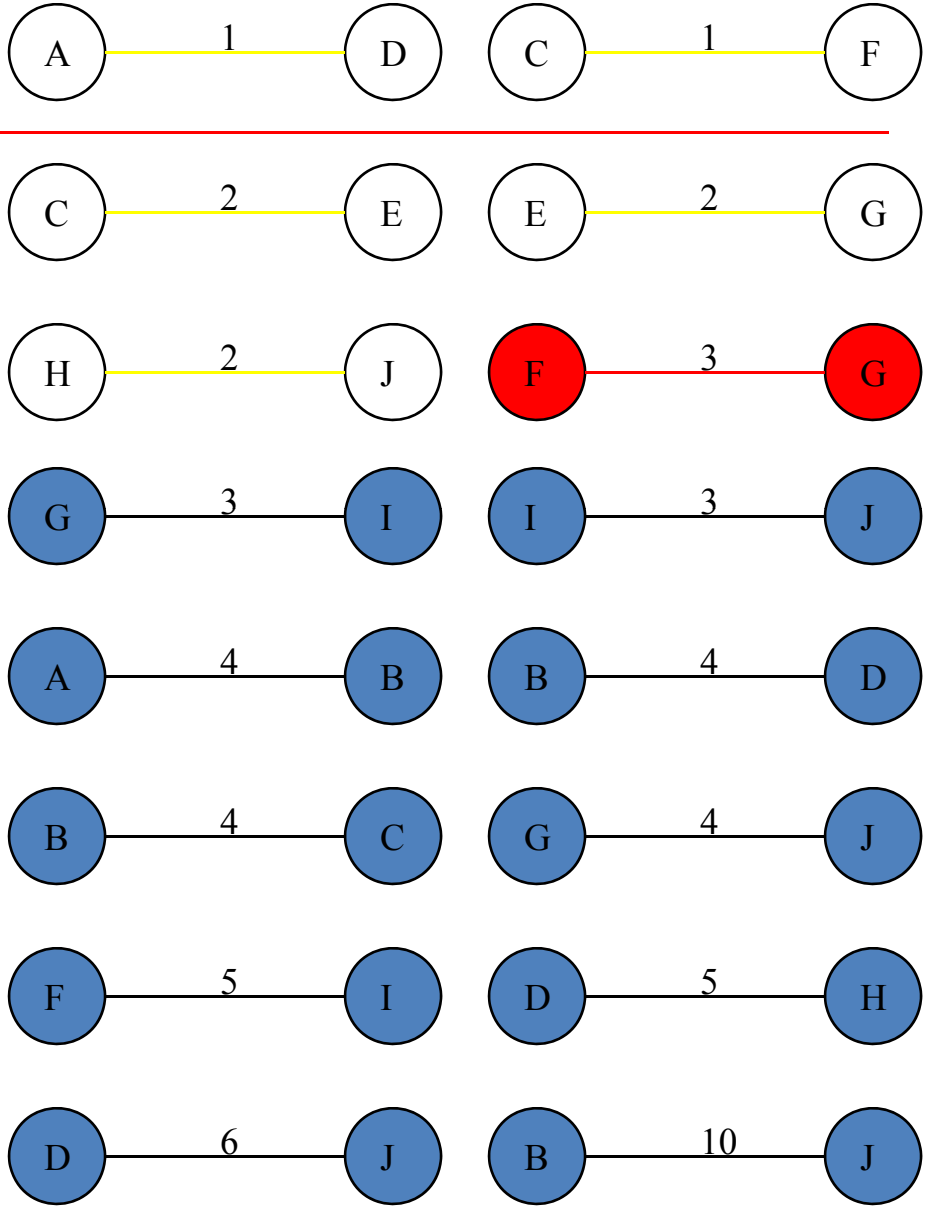

A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
10
3
5
5
6
3
4
3
2
3


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

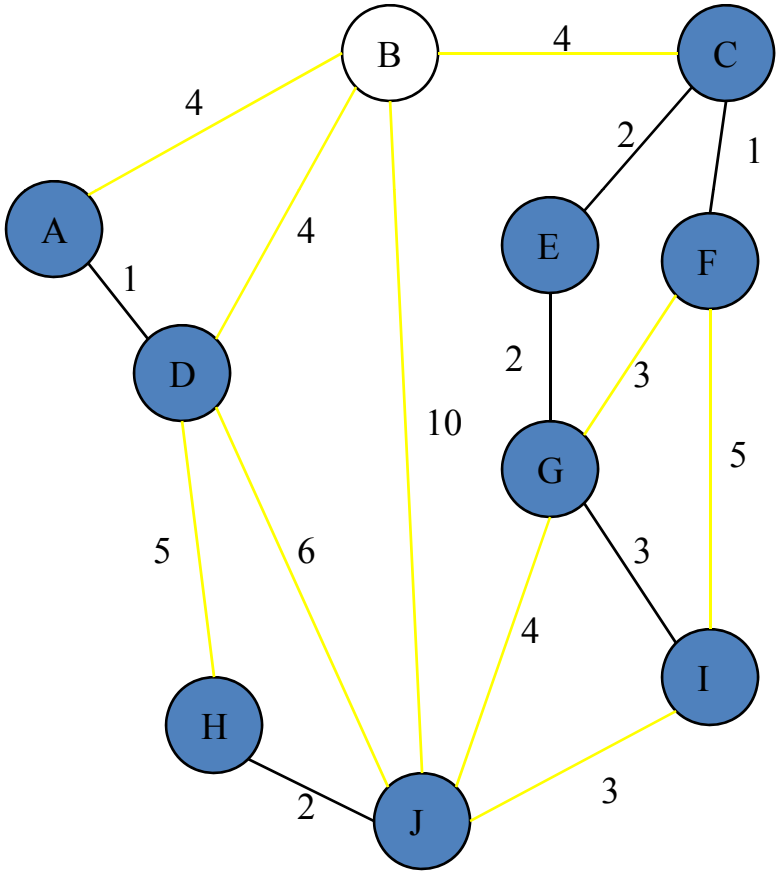
Cycle

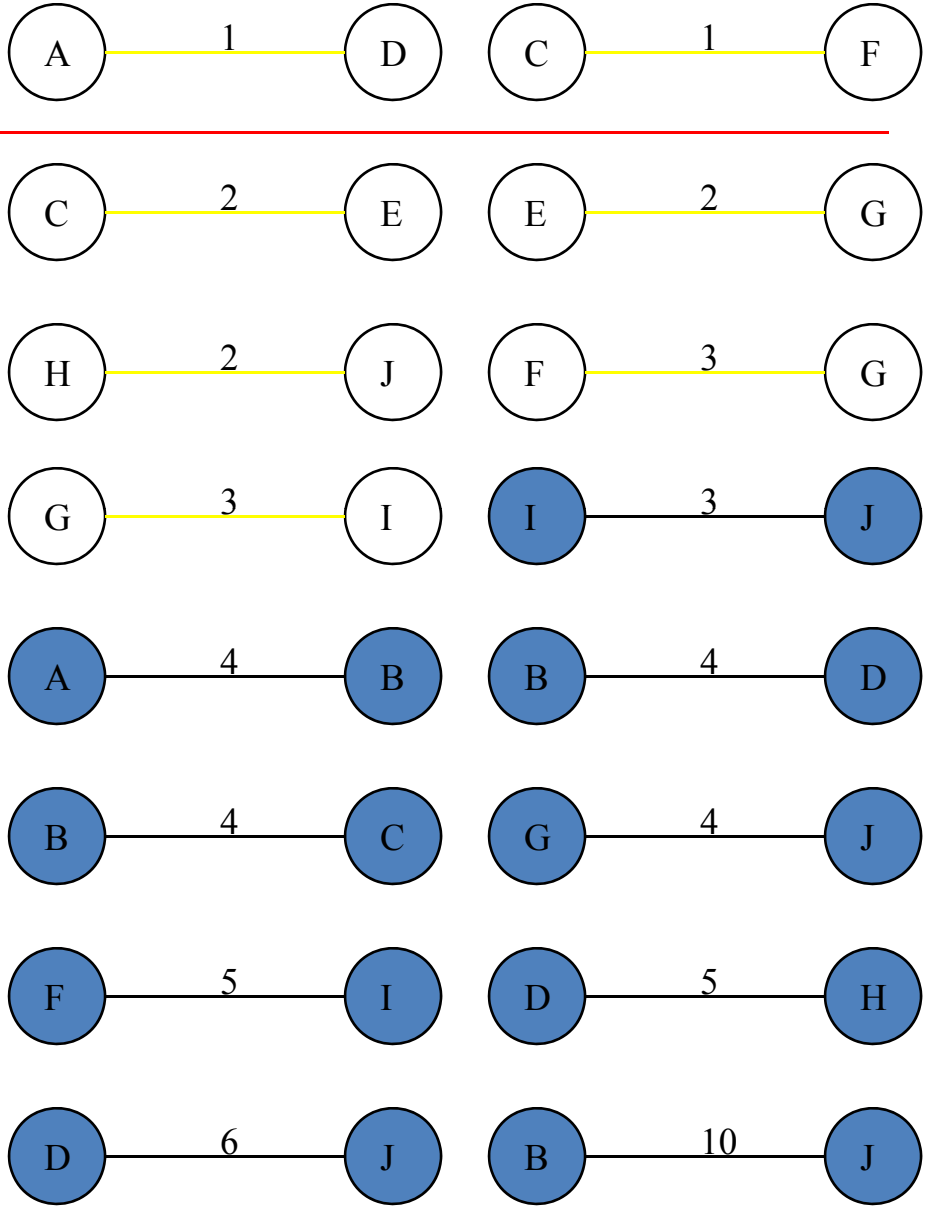
Don't Add Edge



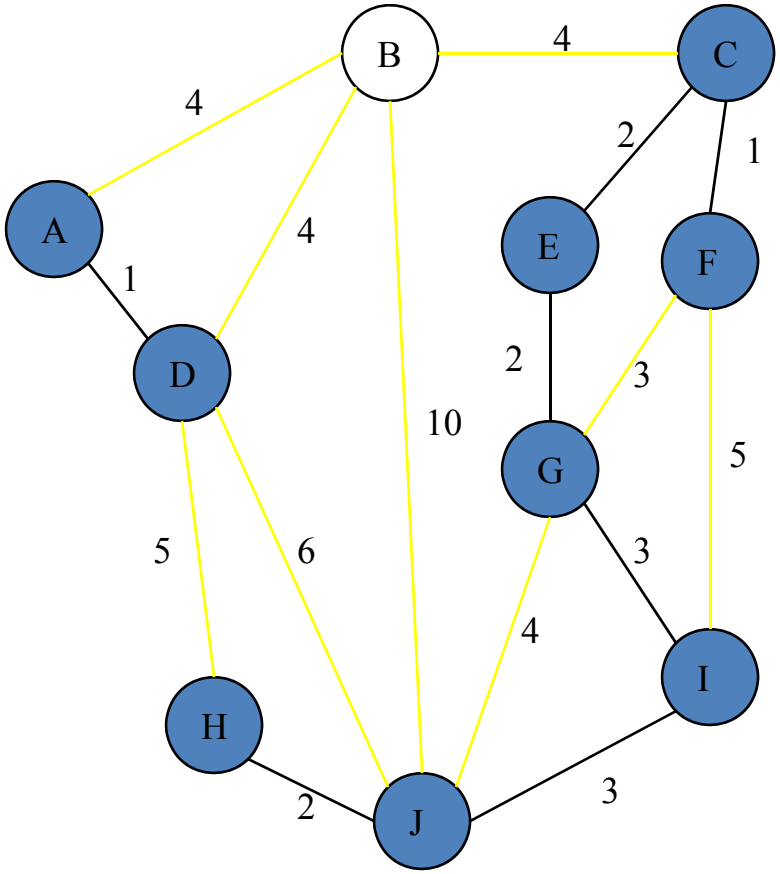


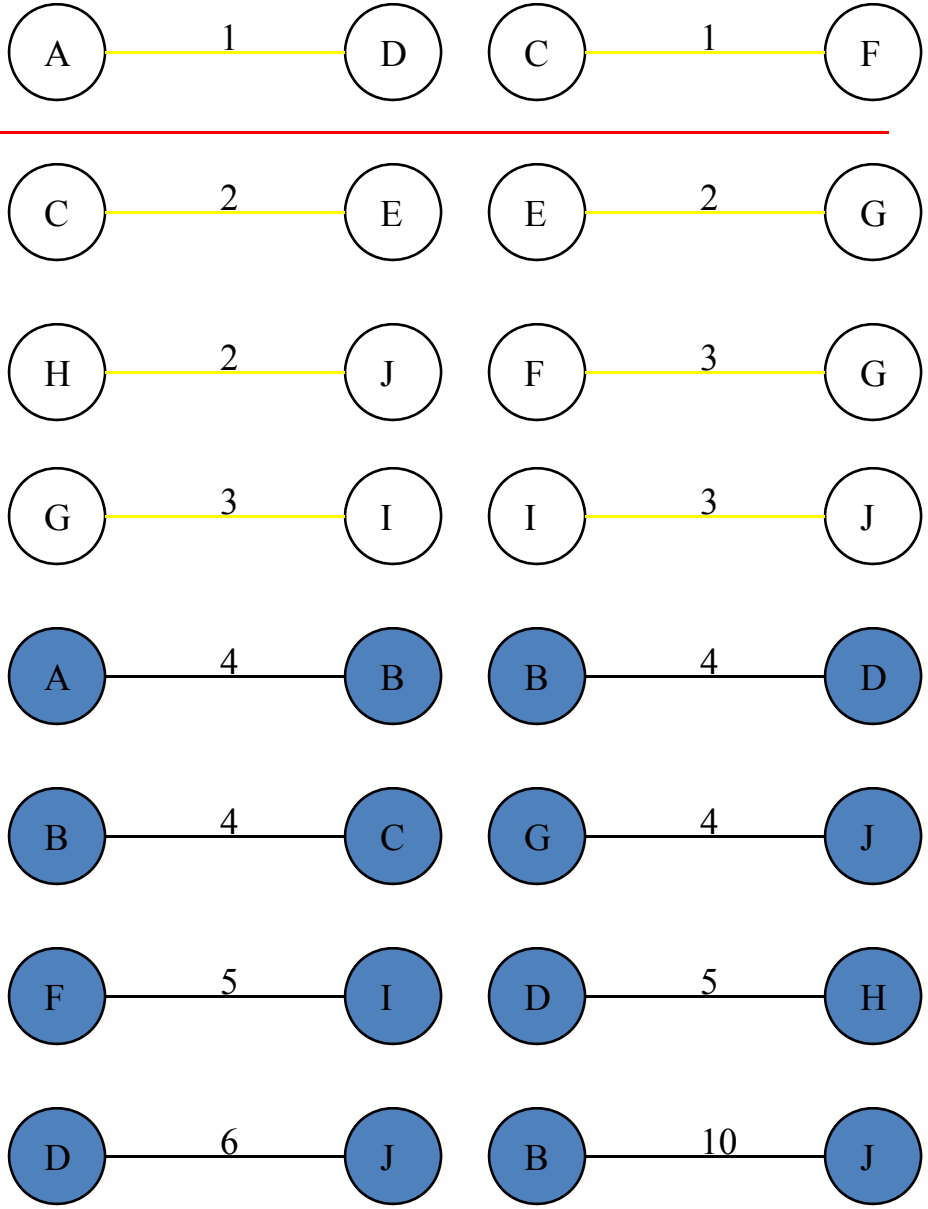
Add Edge


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

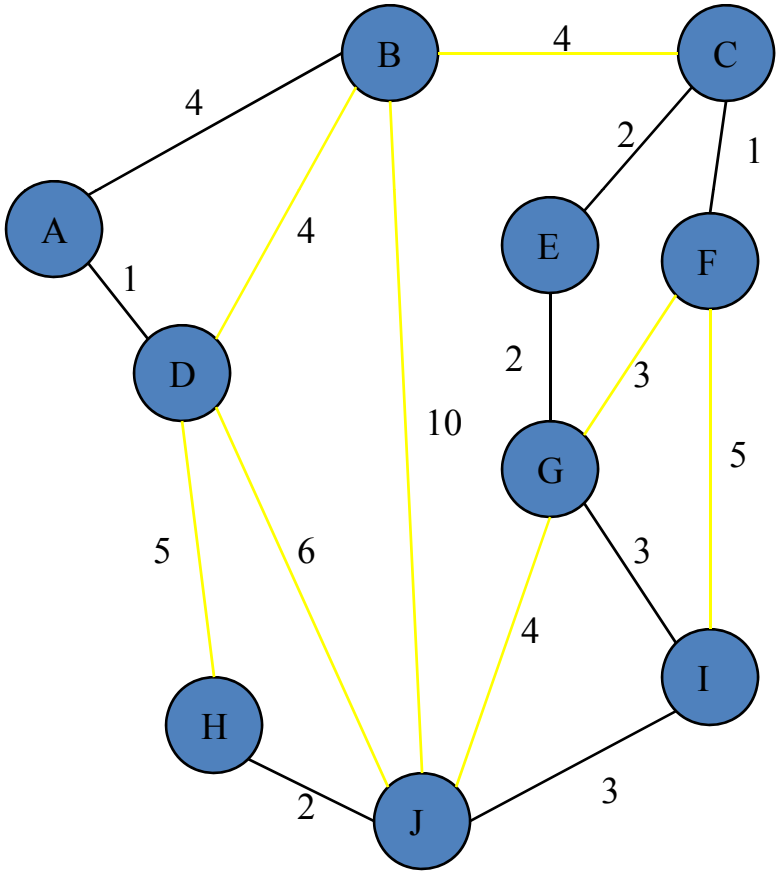

A 1 D
C 1 F
C 2 E
E 2 G
H 2 J
F 3 G
G 3 I
I 3 J
A 4 B
B 4 D
B 4 C
G 4 J
F 5 I
D 5 H
D 6 J
B 10 J

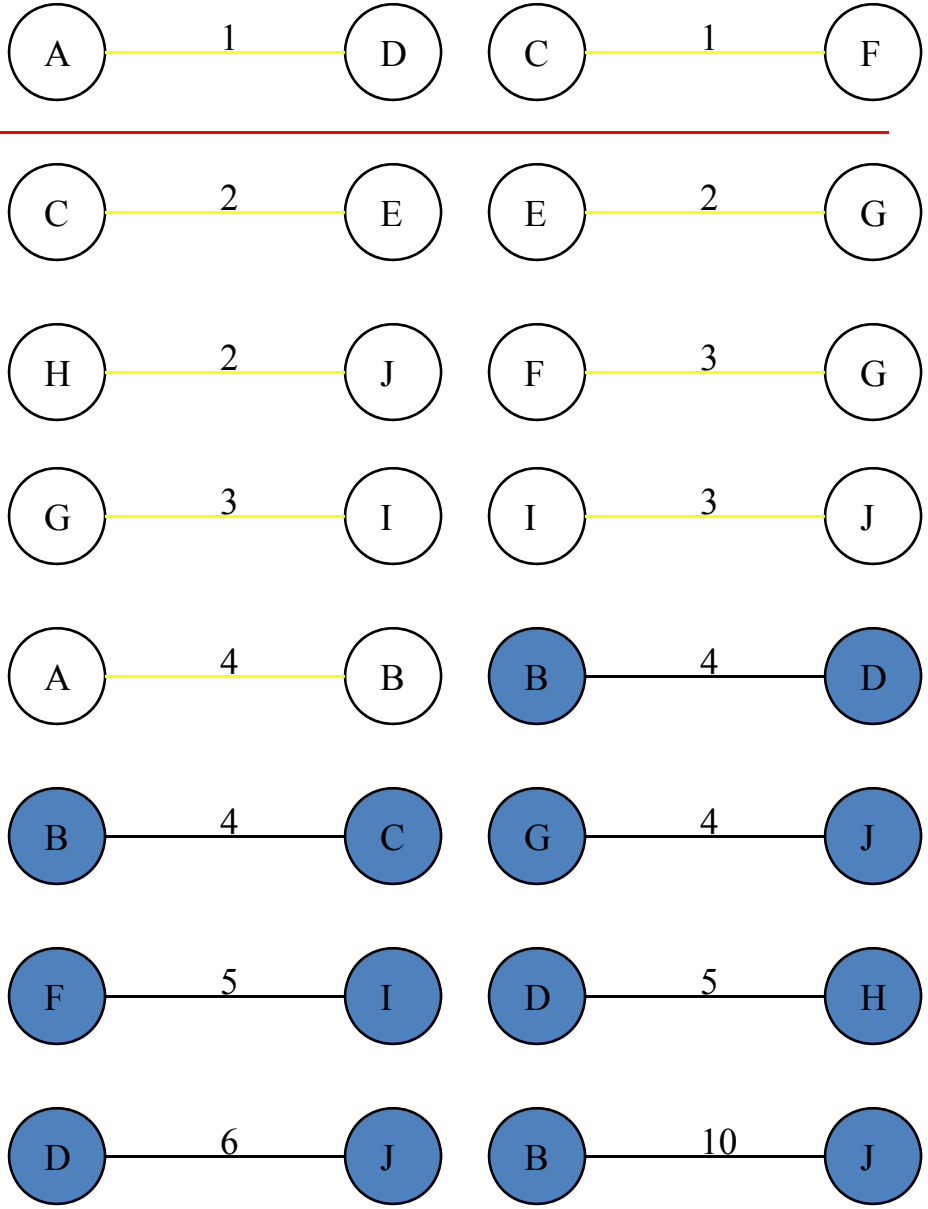
Add Edge


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

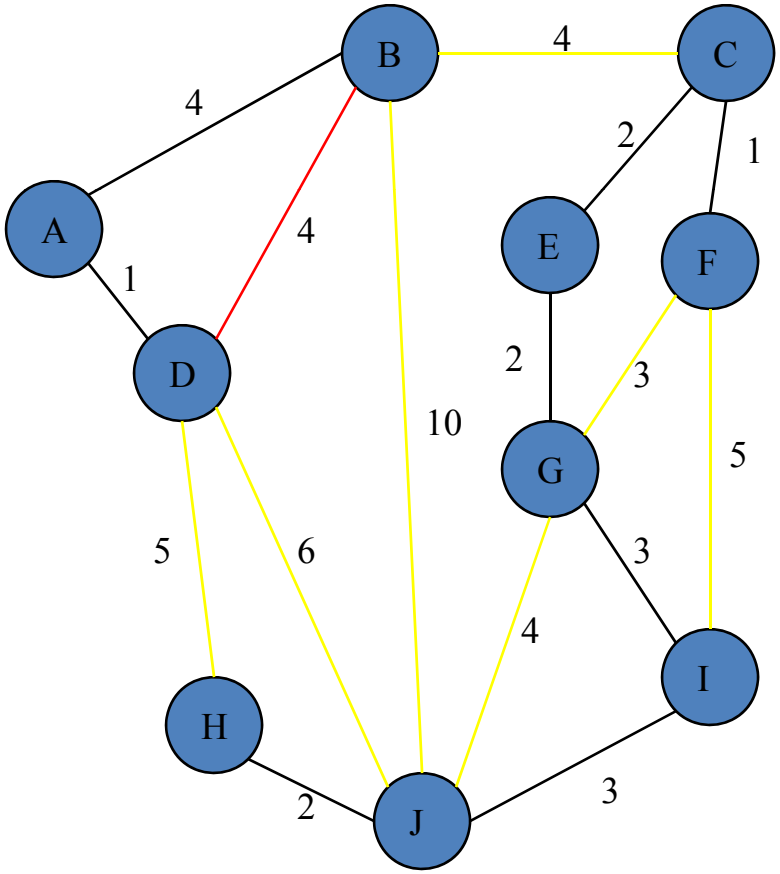
Add Edge

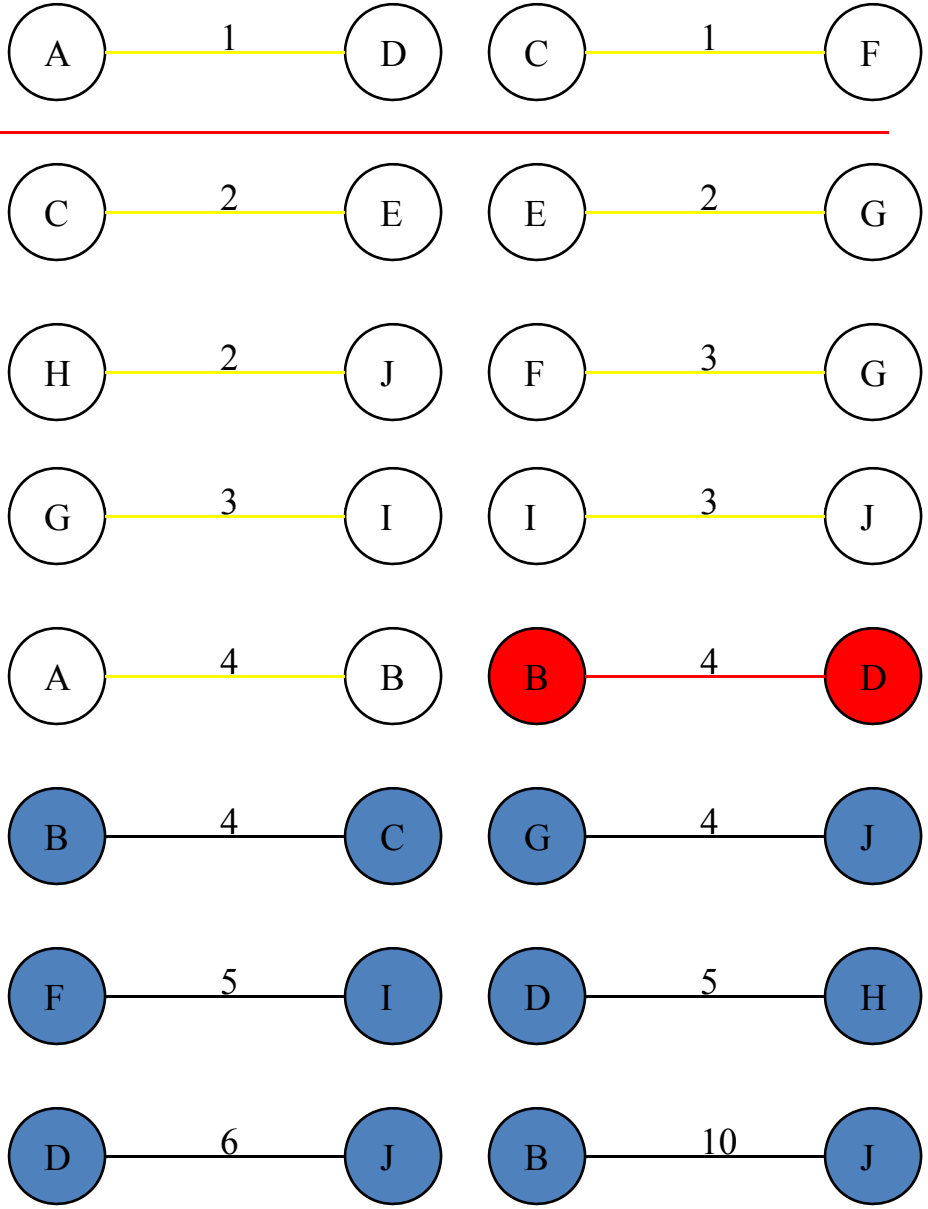

A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
10
3
5
5
6
3
4
3
2
3


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

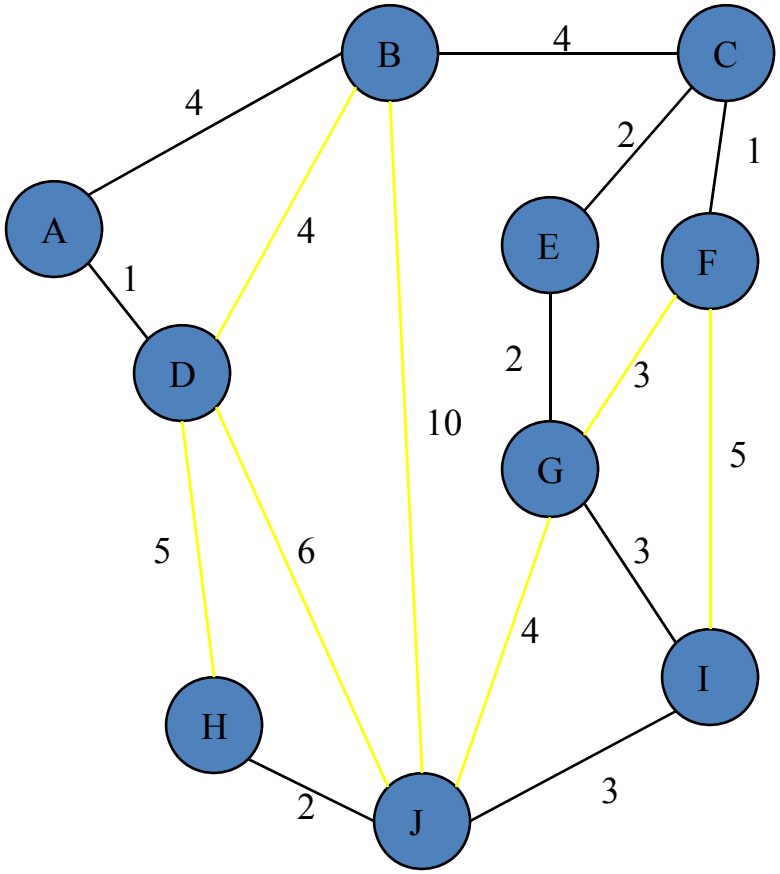
Cycle

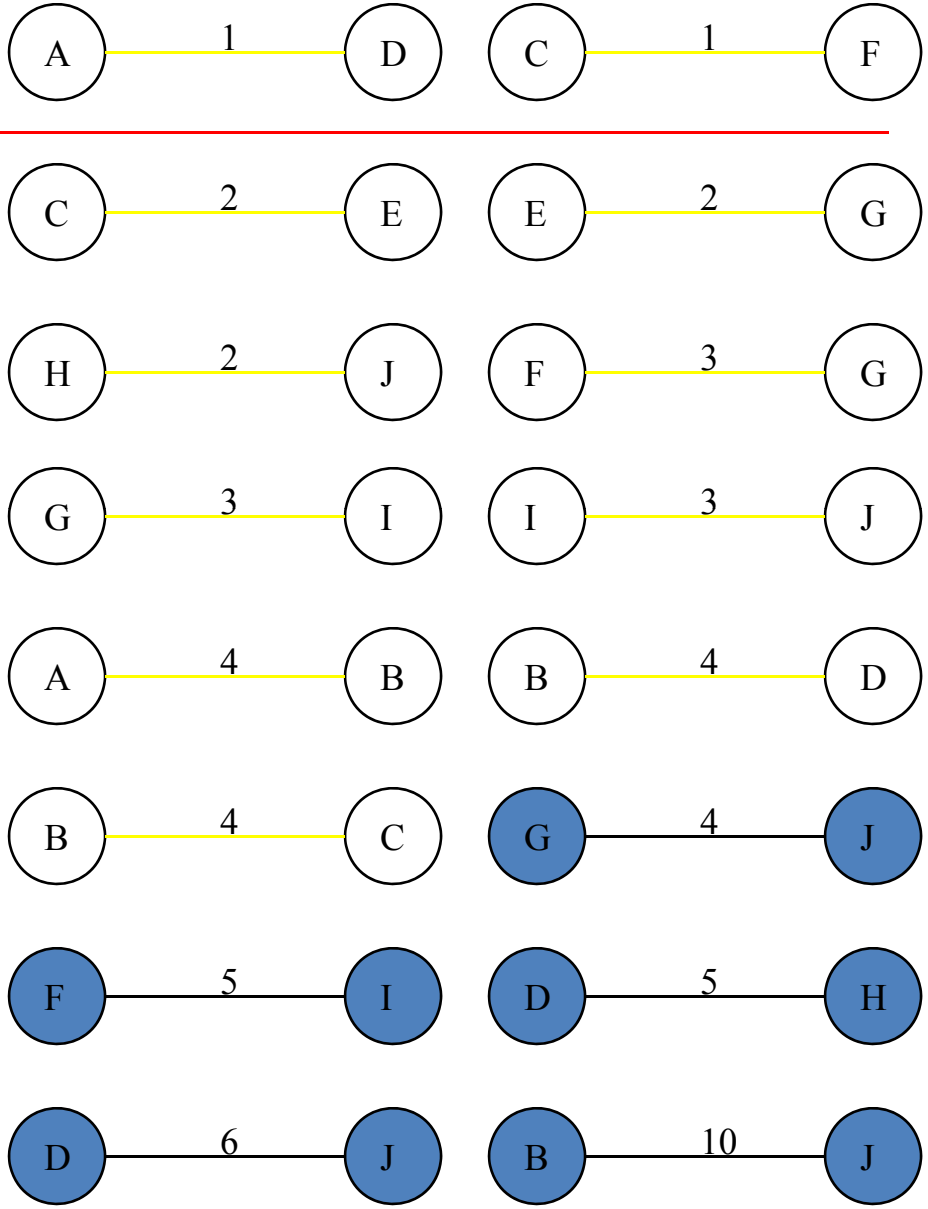
Don't Add Edge



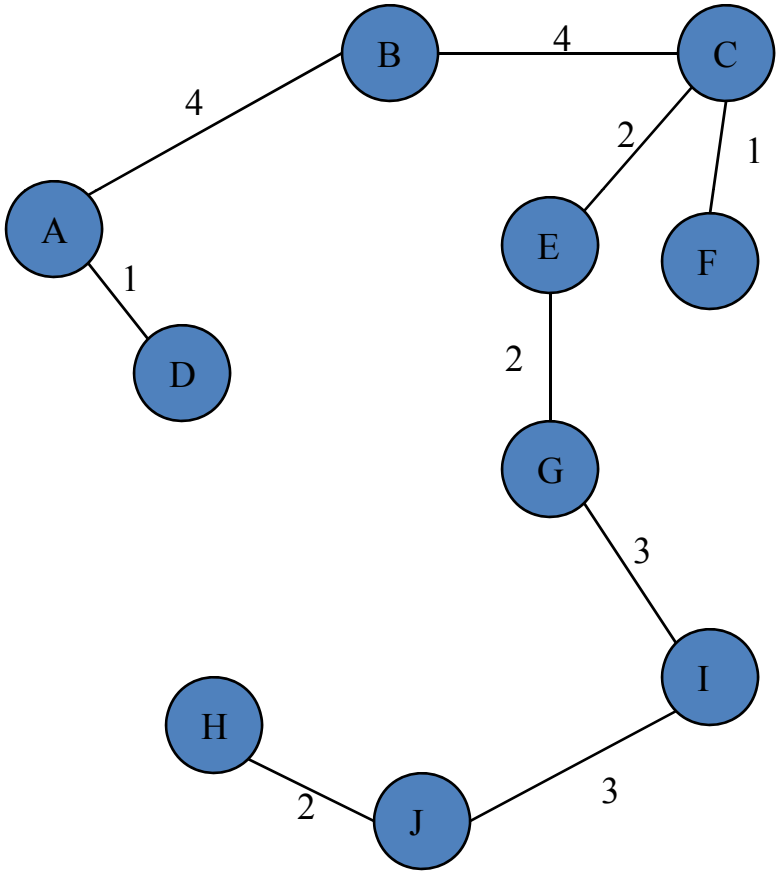


Add Edge

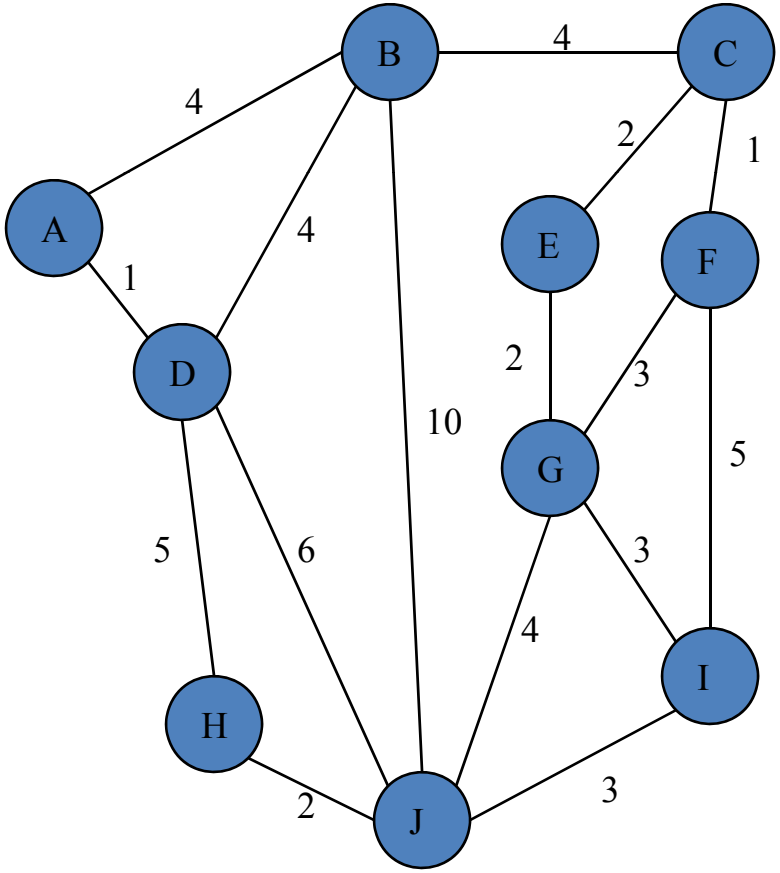

A
B
C
D
E
F
G
H
I
J
4
4
1
4
2
1
2
10
3
5
5
6
3
4
3
2
3


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

Minimum Spanning Tree



Example Graph



Analysis of Kruskal's Algorithm

The algorithm starts with sorting edges (O (E log E)).

We consider all E edges.

For each edge we check for possibility of cycle (O(logn))

Total running time is O(ElogE) + O (Elogn)

Prim's Algorithm

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

Prim's Algorithm

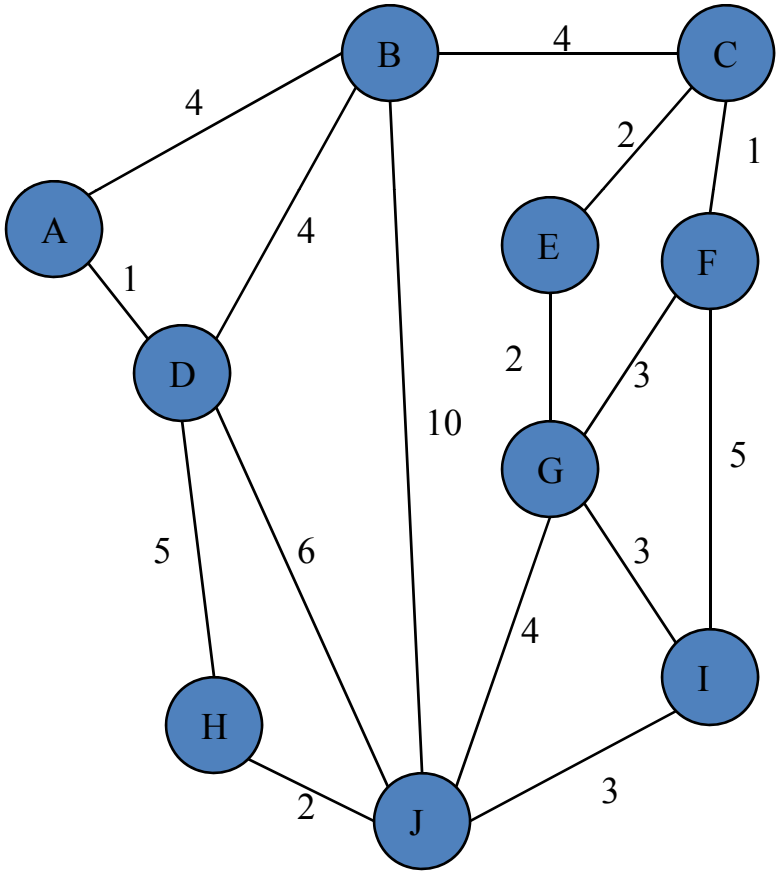
The steps are:

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

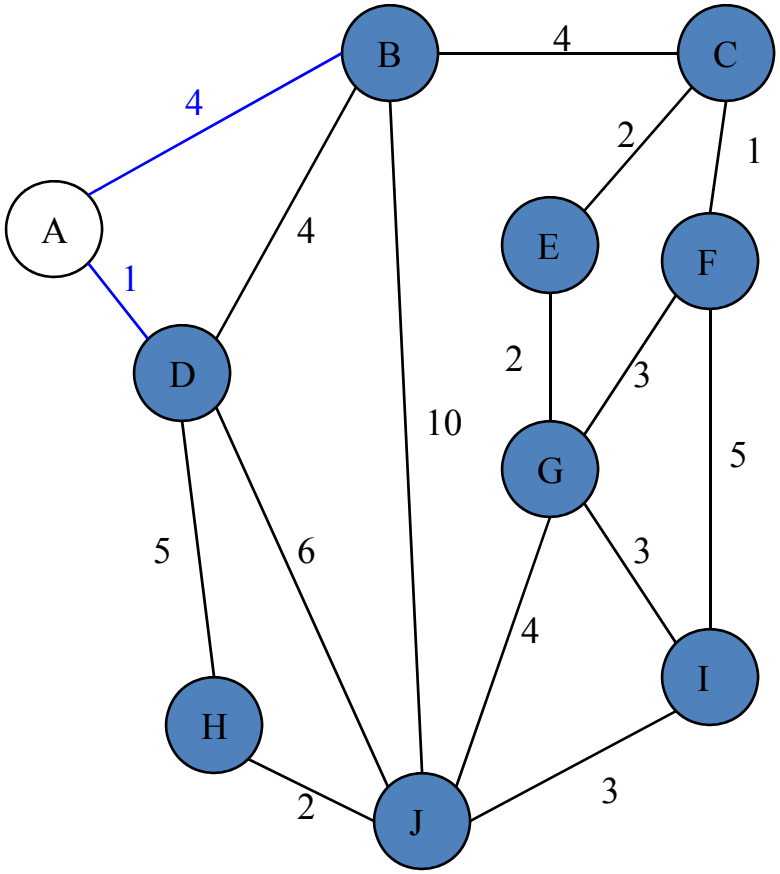
Complexity: $O(n^2)$

In every step one node is added, 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.

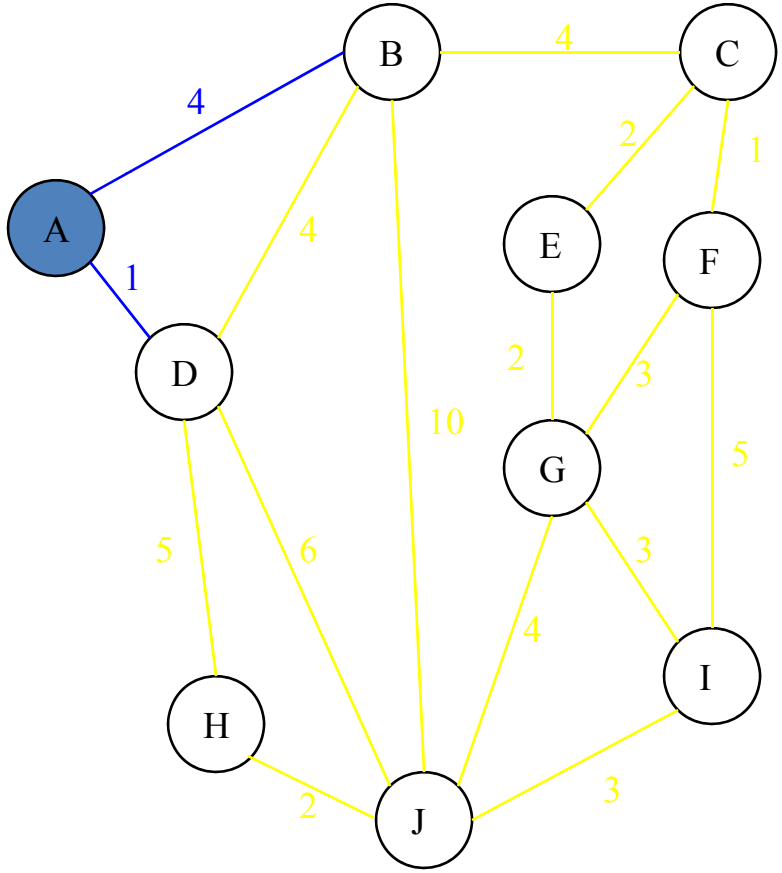
Example Graph



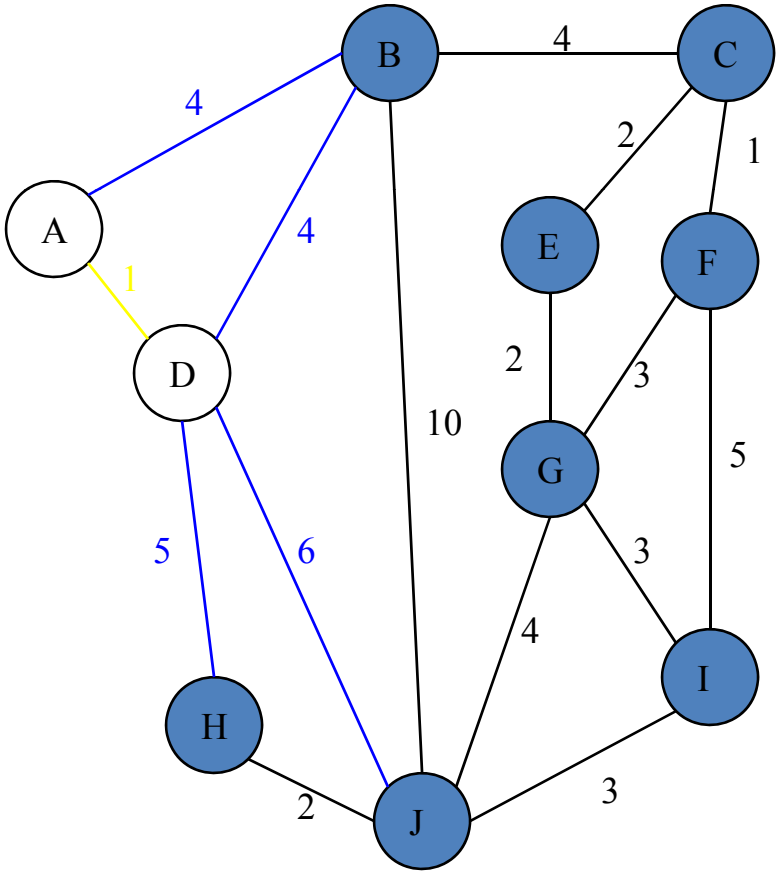
Example Graph



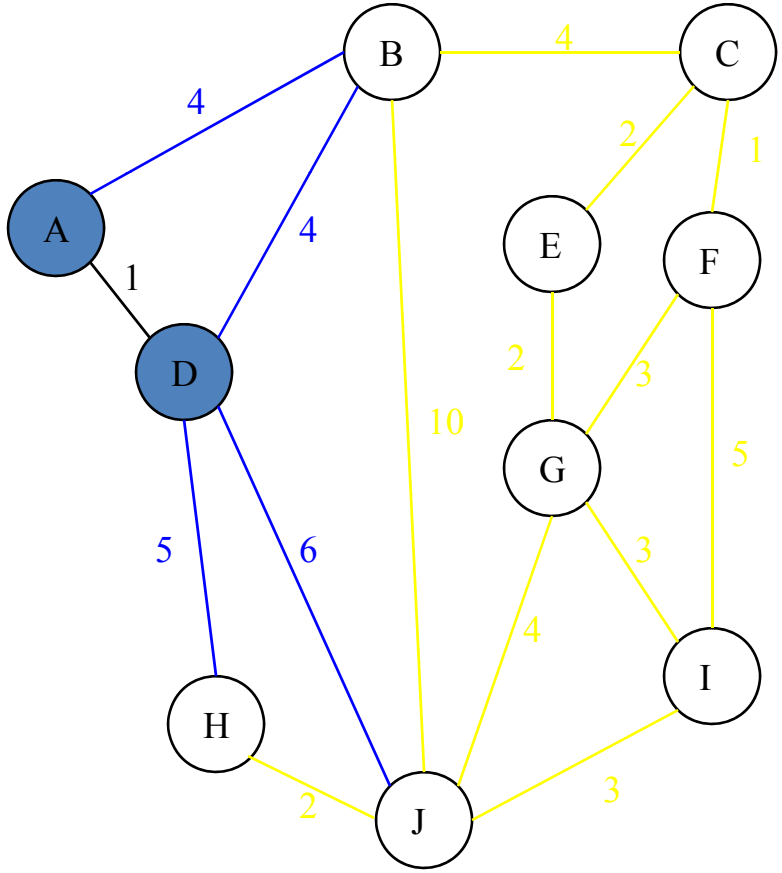
New Graph



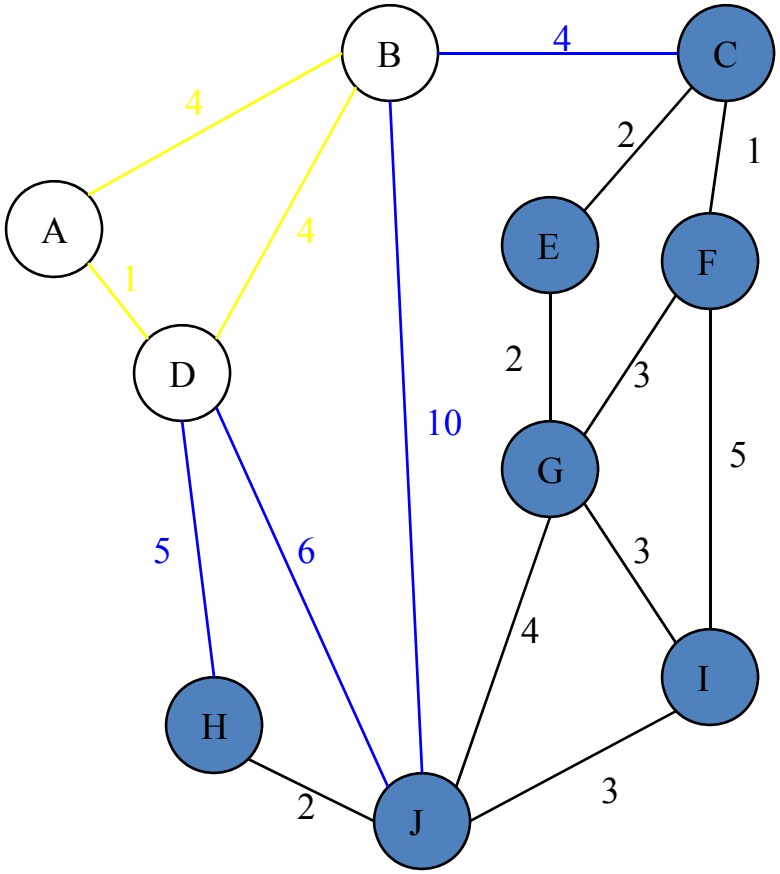
Example Graph



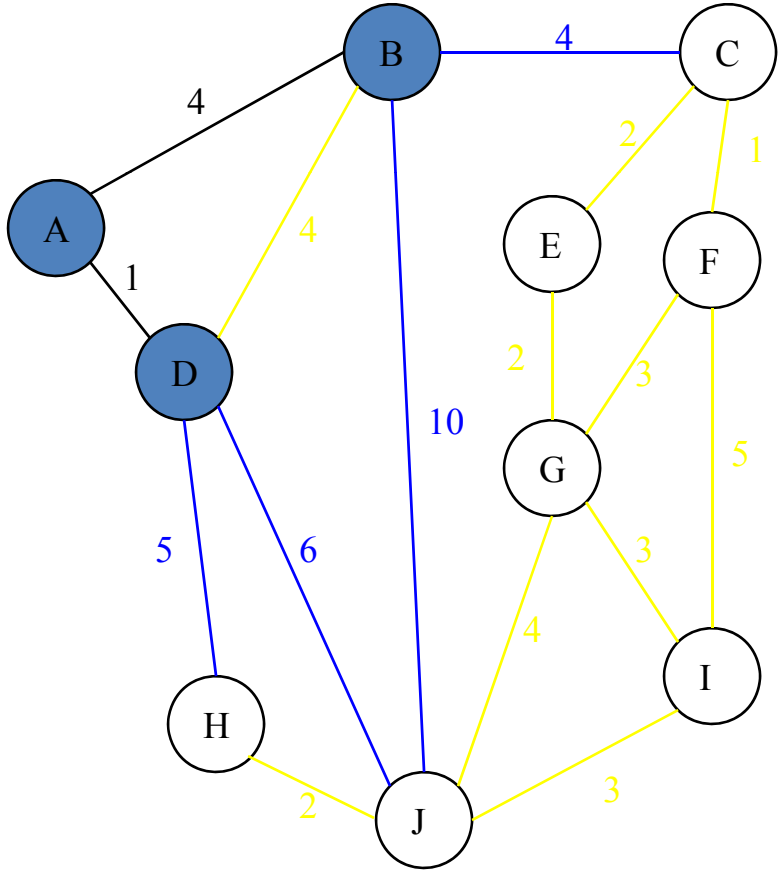
New Graph



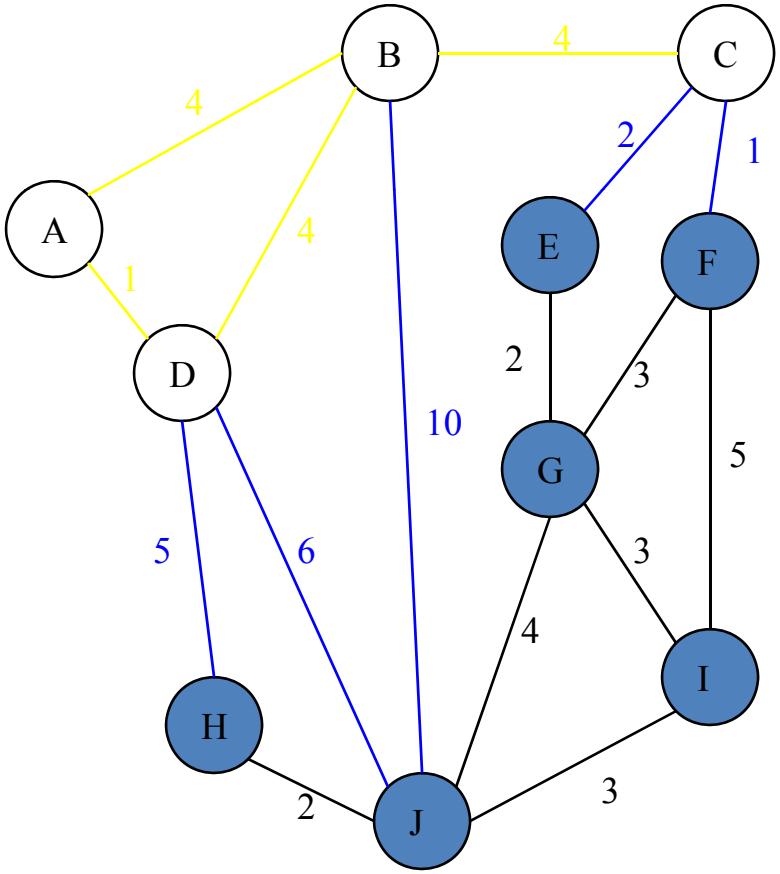
Example Graph



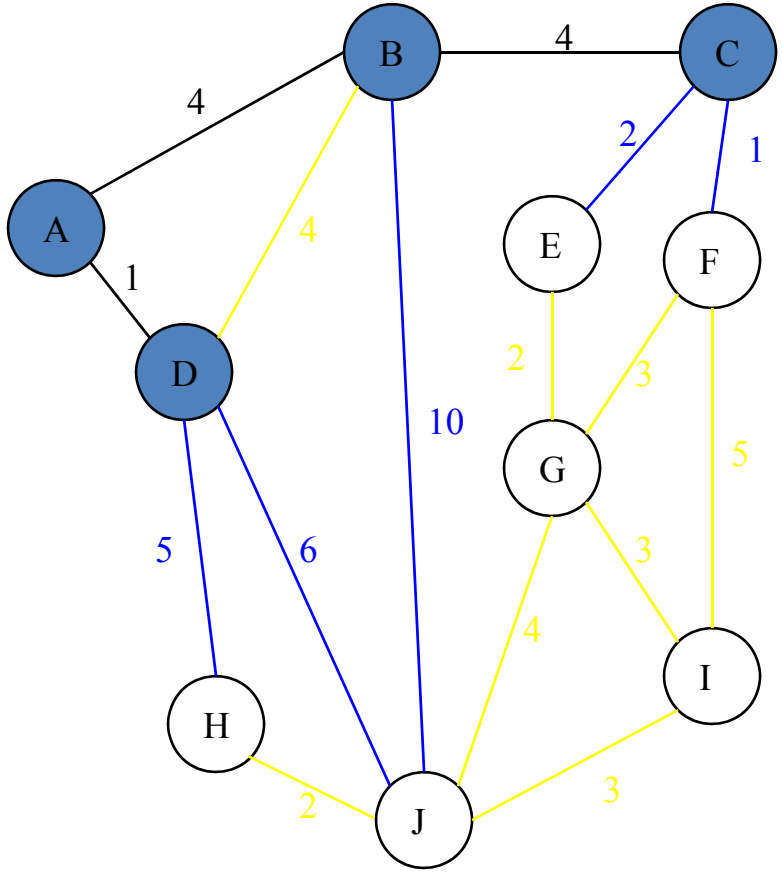
New Graph



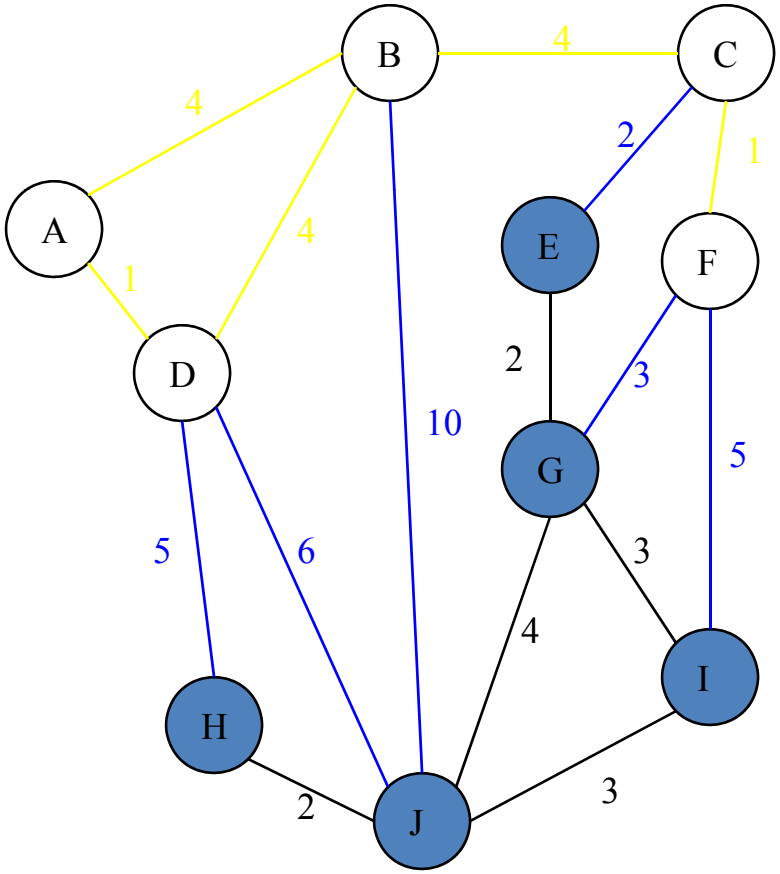
Example Graph



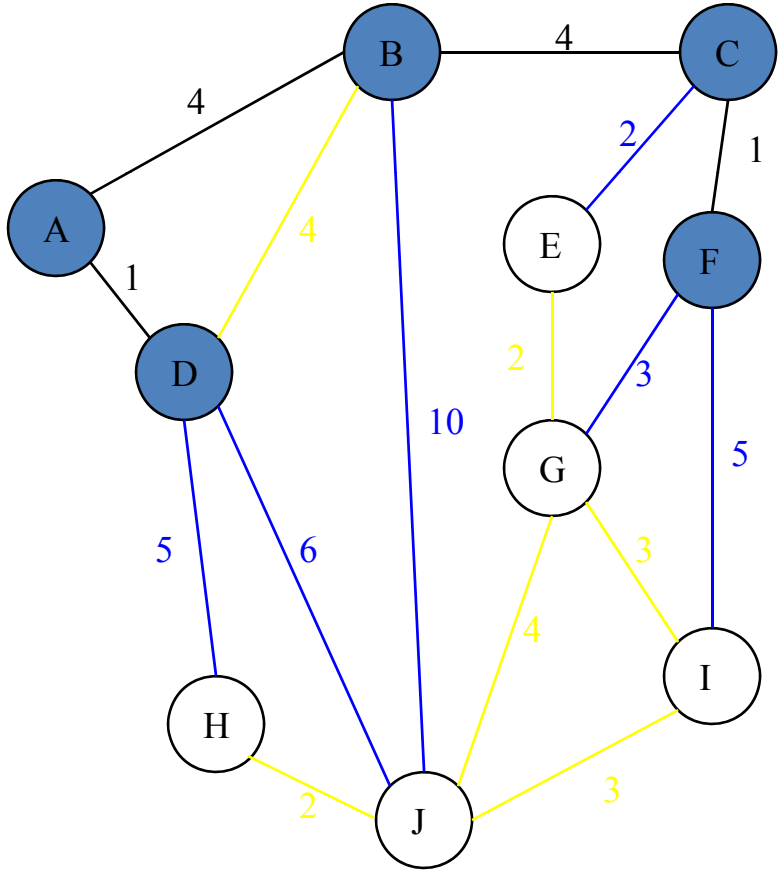
New Graph



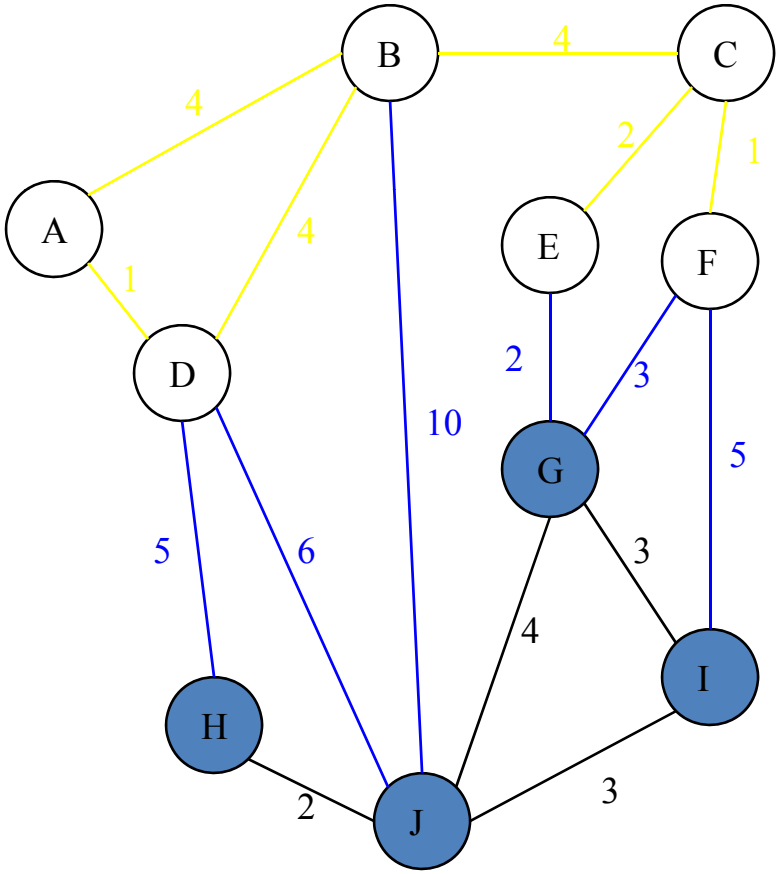
Example Graph



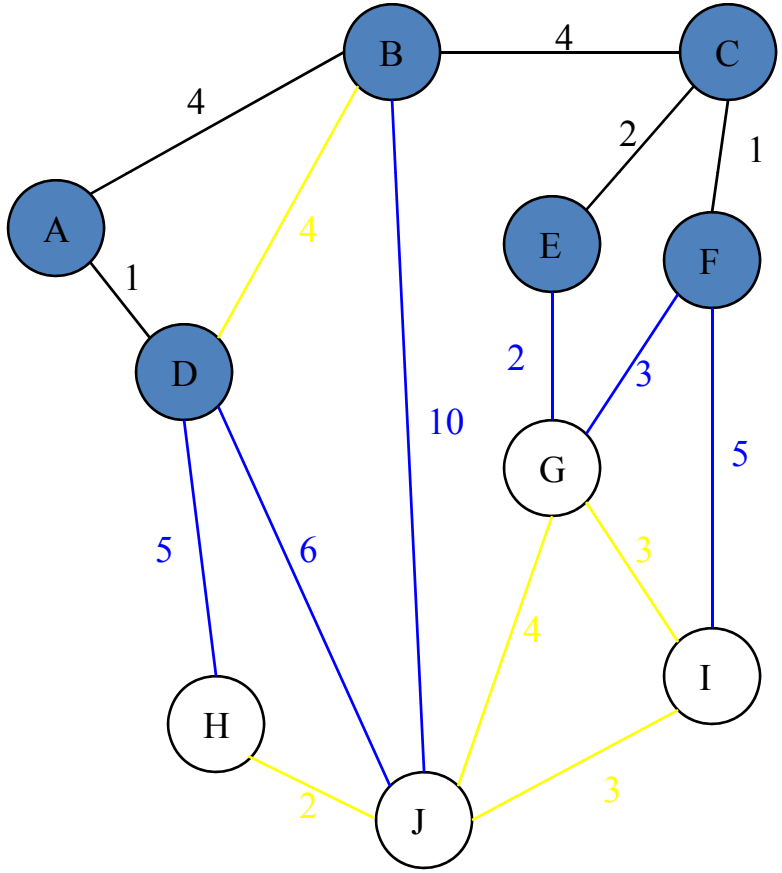
New Graph



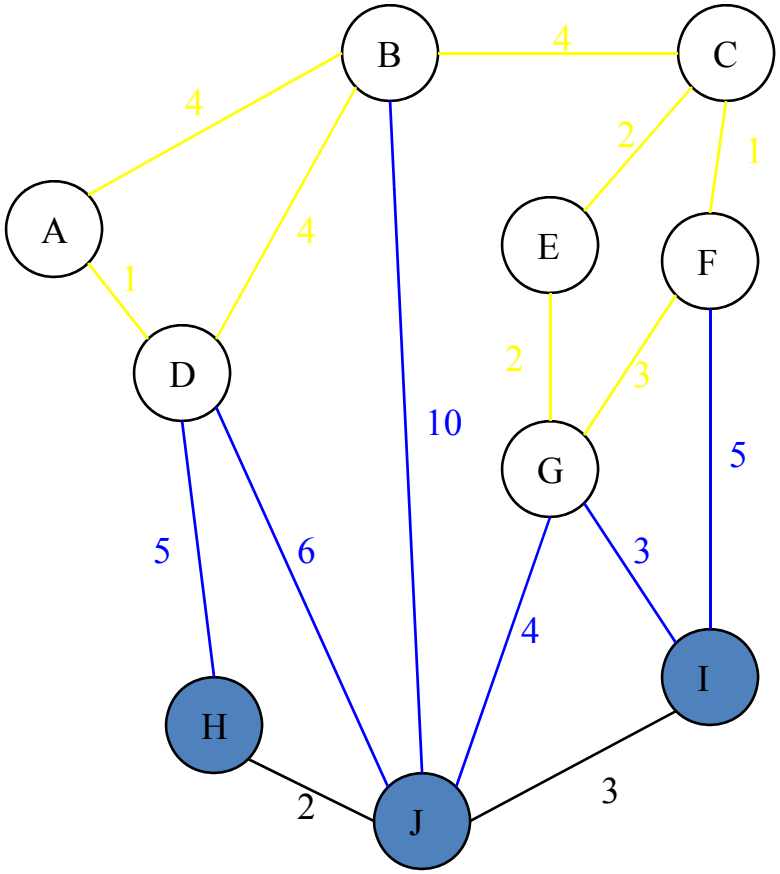
Example Graph



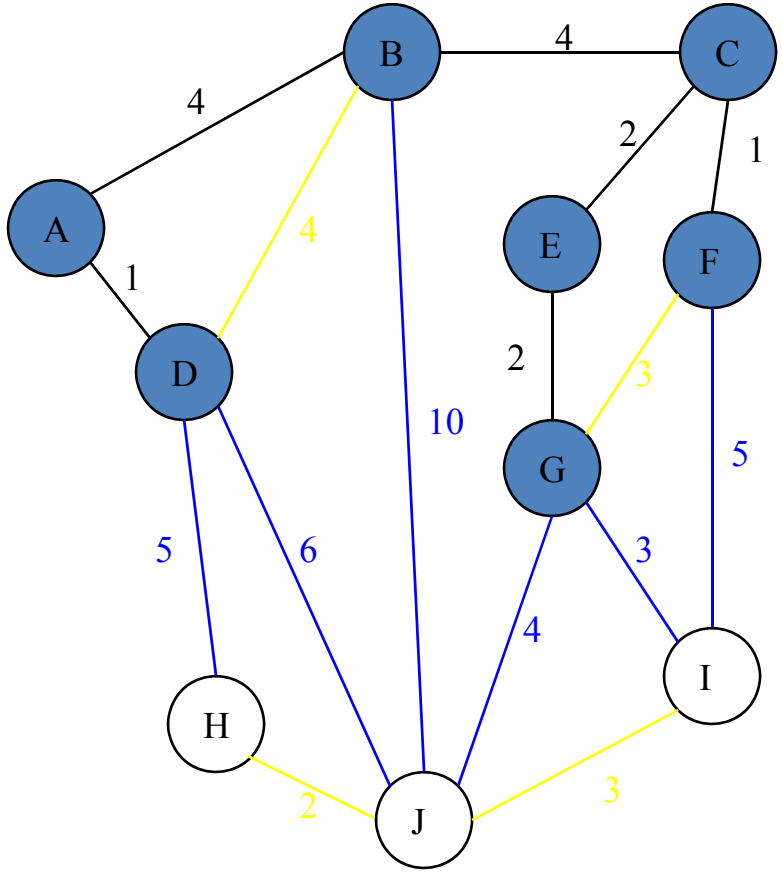
New Graph



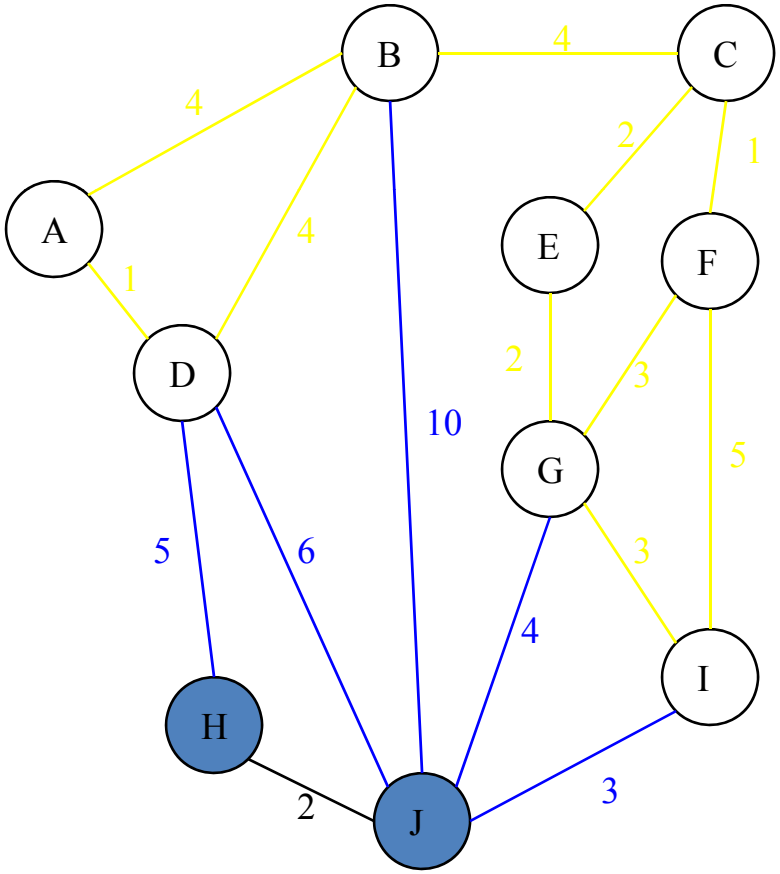
Example Graph



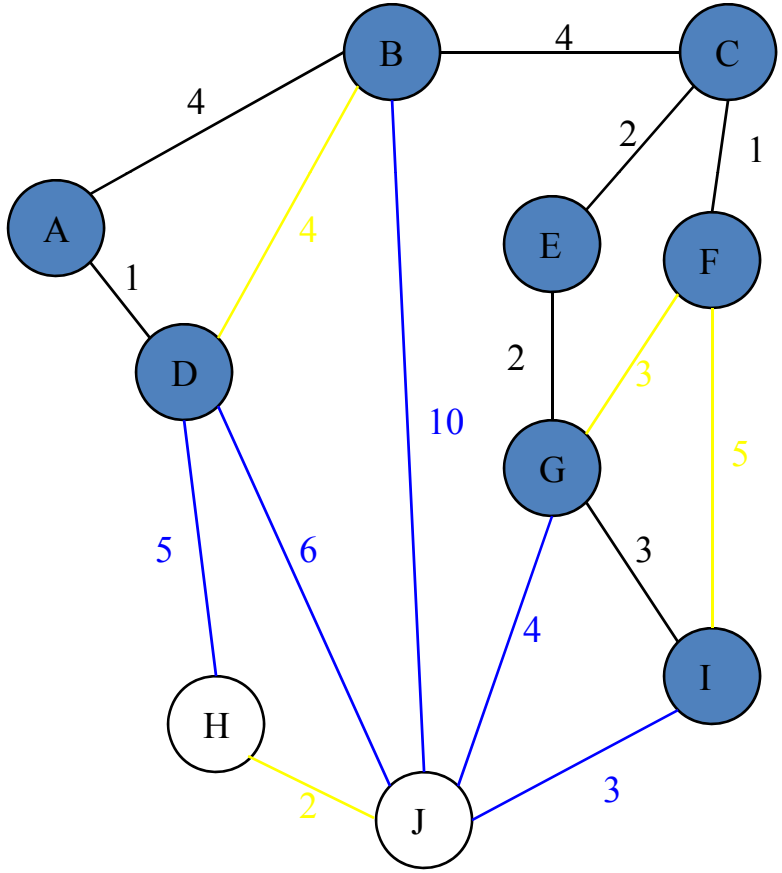
New Graph



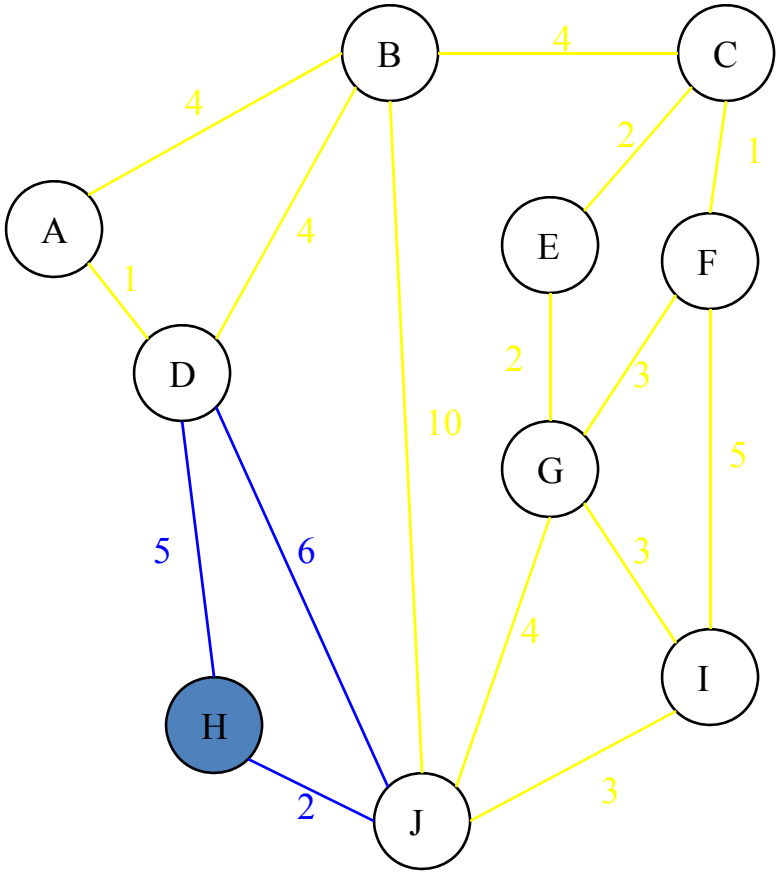
Example Graph



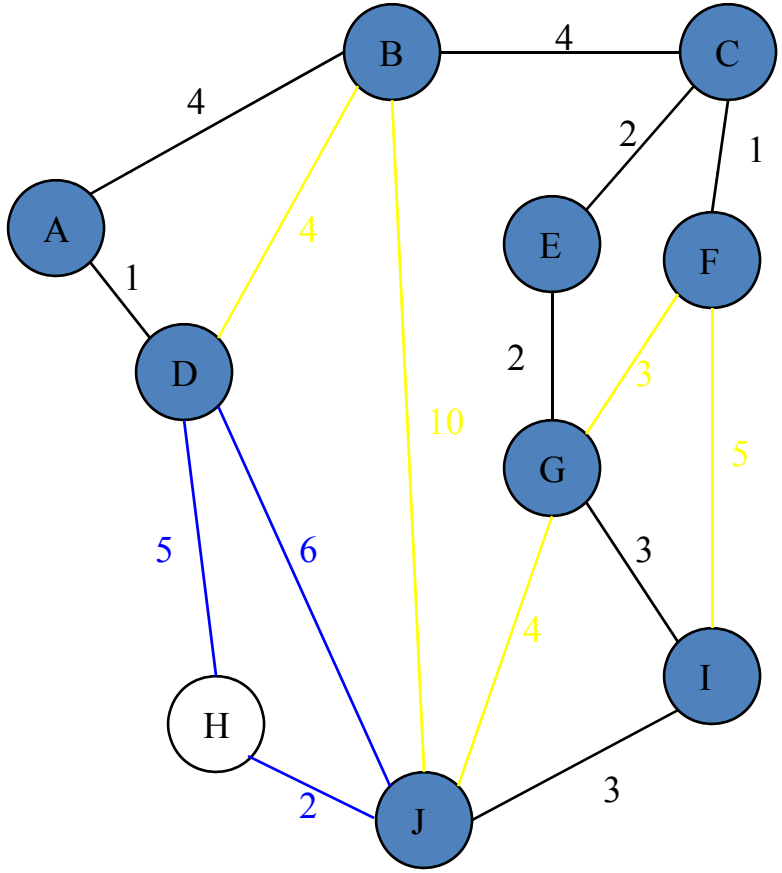
New Graph



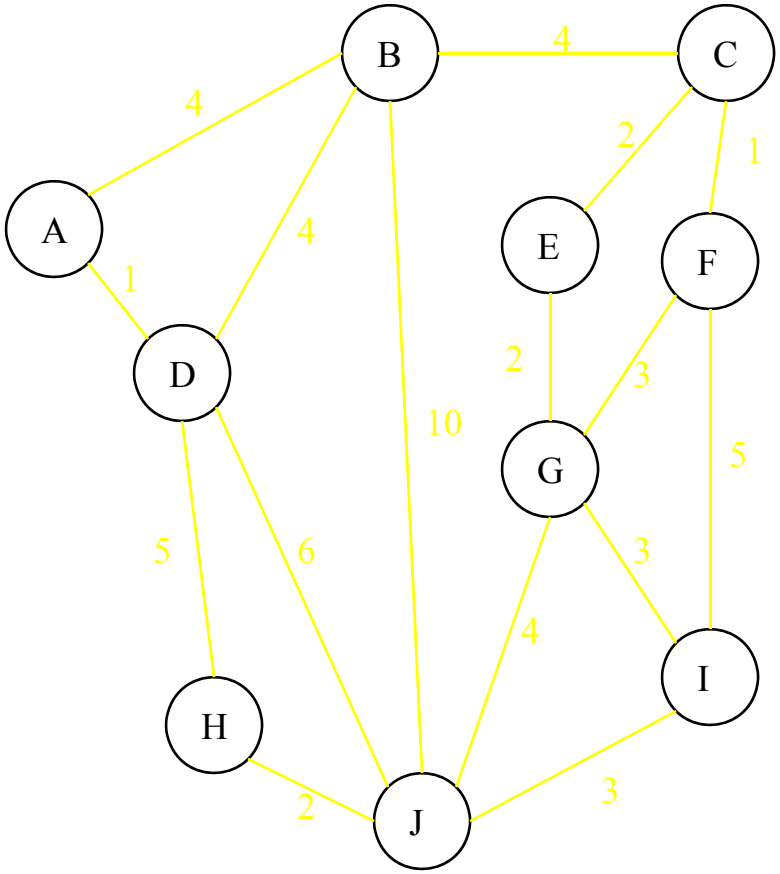
Example Graph



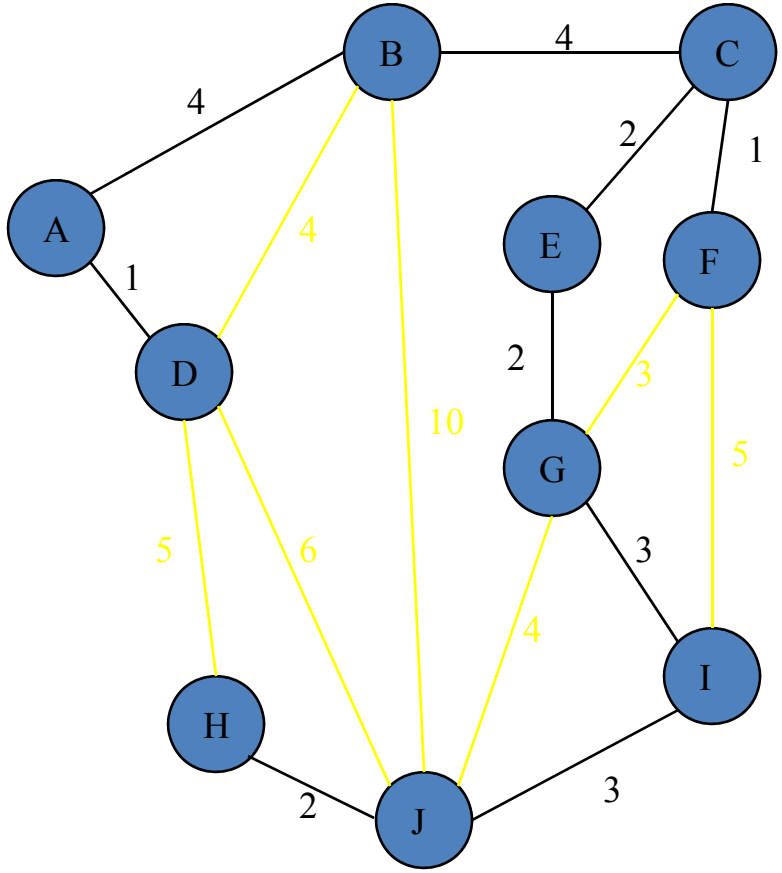
New Graph



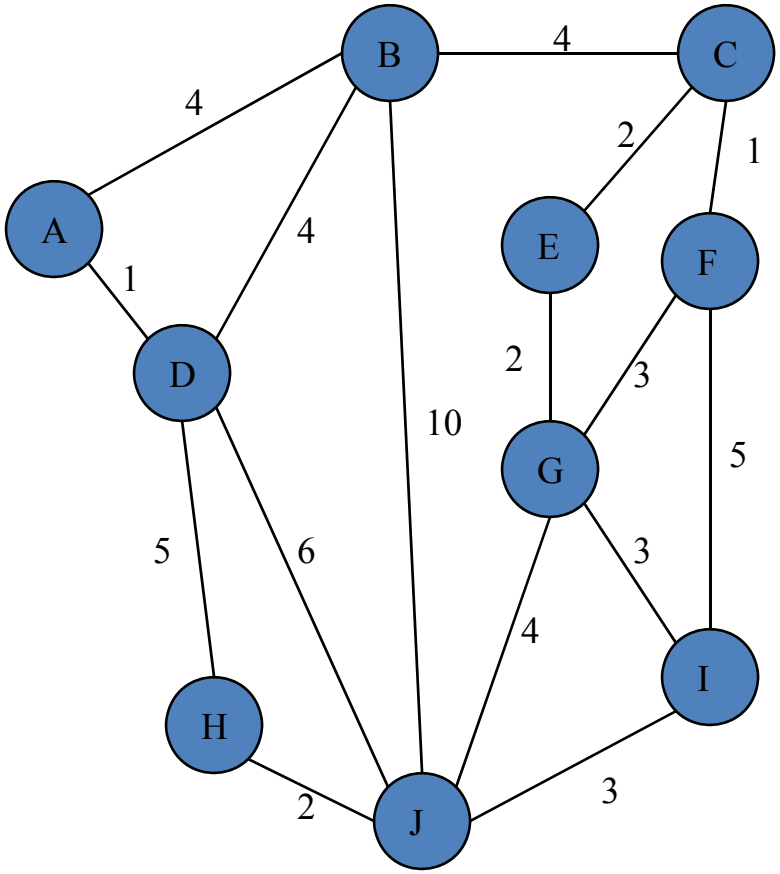
Example Graph



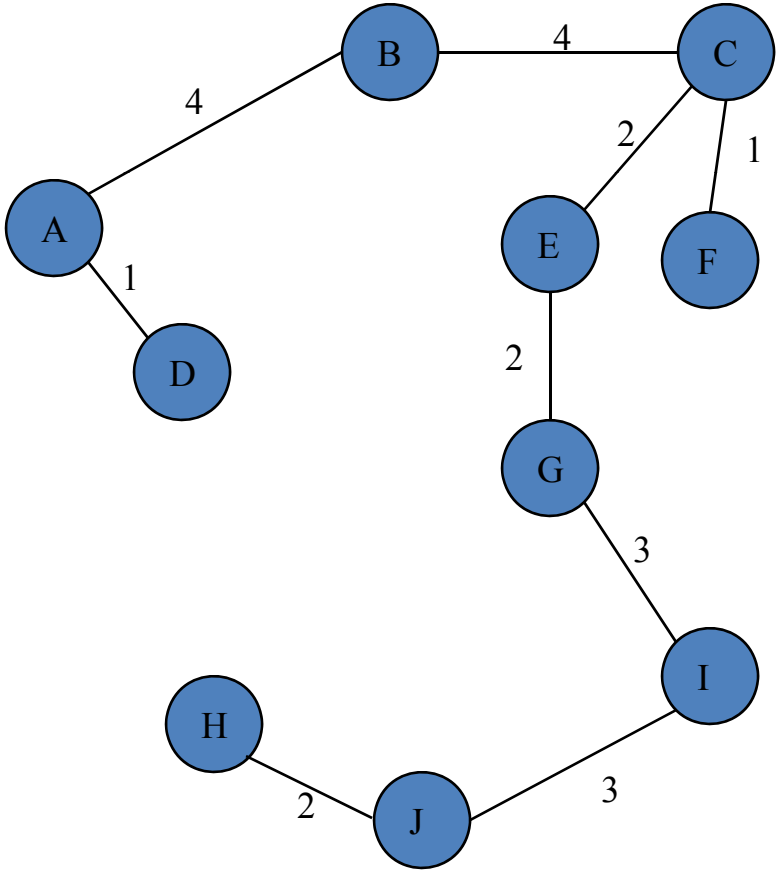
New Graph



Example Graph



Minimum Spanning Tree



Any Doubt ?

- Please feel free to write to me:

bhaskargit@yahoo.co.in