Lecture-07

Graph Algorithm

Sharad Hasan

Lecturer

Department of Computer Science and Engineering

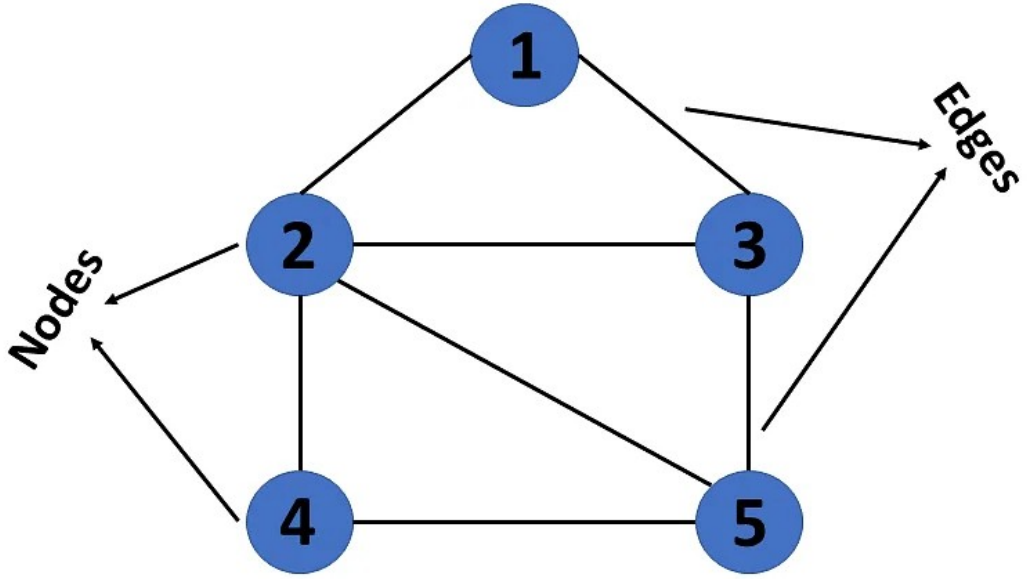
Sheikh Hasina University, Netrokona.

Introduction to Graphs

Graphs in data structures are non-linear data structures made up of a finite number of nodes or vertices and the edges that connect them. Graphs in data structures are used to address real-world problems in which it represents the problem area as a network like telephone networks, circuit networks, and social networks. For example, it can represent a single user as nodes or vertices in a telephone network, while the link between them via telephone represents edges.

Introduction to Graphs

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(V, E).



This graph has a set of vertices V= { 1,2,3,4,5} and

a set of edges E= { (1,2),(1,3),(2,3),(2,4),(2,5),(3,5),(4,50 }.

Introduction to Graphs

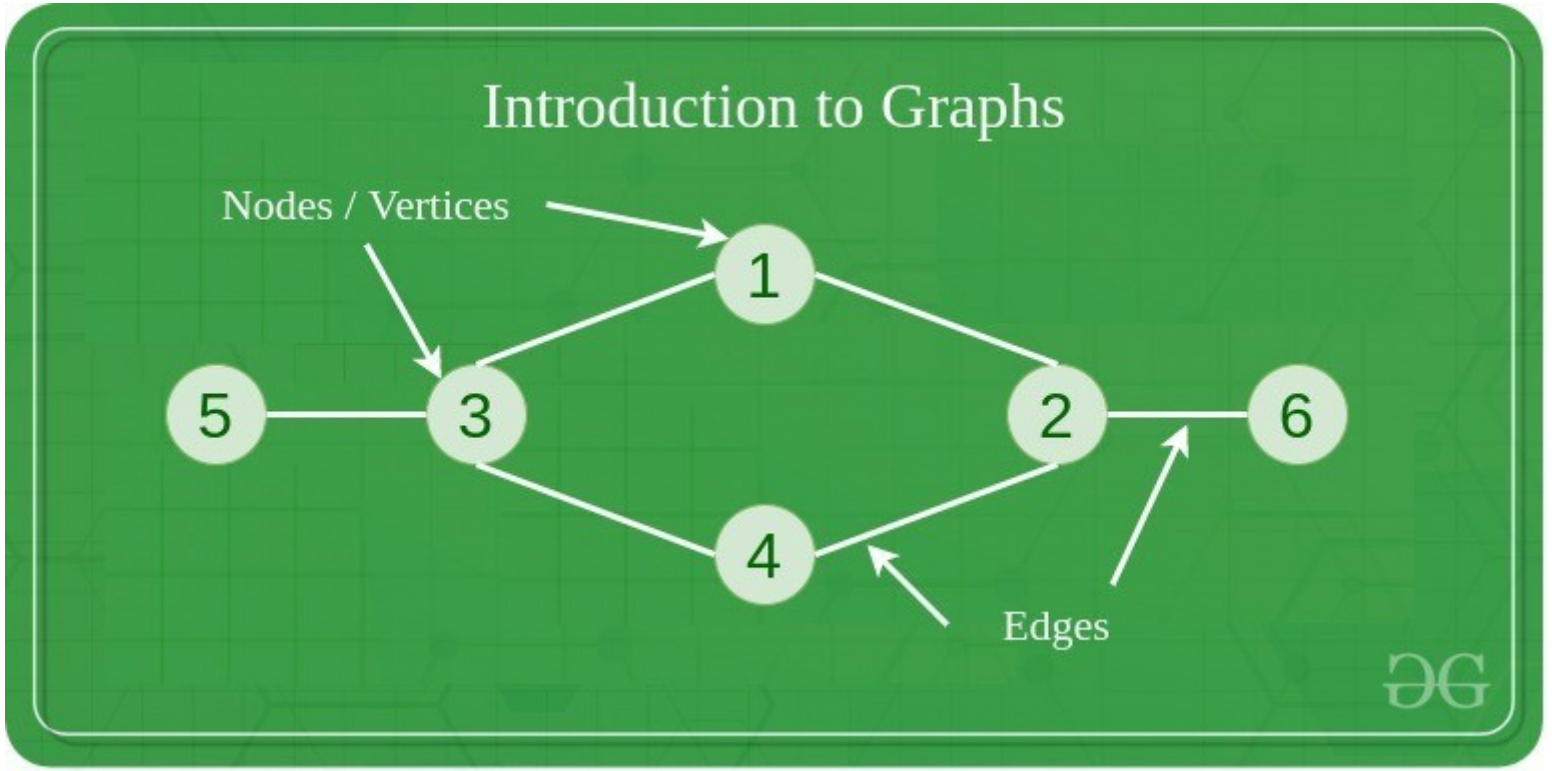
Graph data structures are a powerful tool for representing and analyzing complex relationships between objects or entities. They are particularly useful in fields such as social network analysis, recommendation systems, and computer networks. In the field of sports data science, graph data structures can be used to analyze and understand the dynamics of team performance and player interactions on the field.

Imagine a game of football as a web of connections, where players are the nodes and their interactions on the field are the edges. This web of connections is exactly what a graph data structure represents, and it's the key to unlocking insights into team performance and player dynamics in sports.

Components of a Graph

Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabeled.

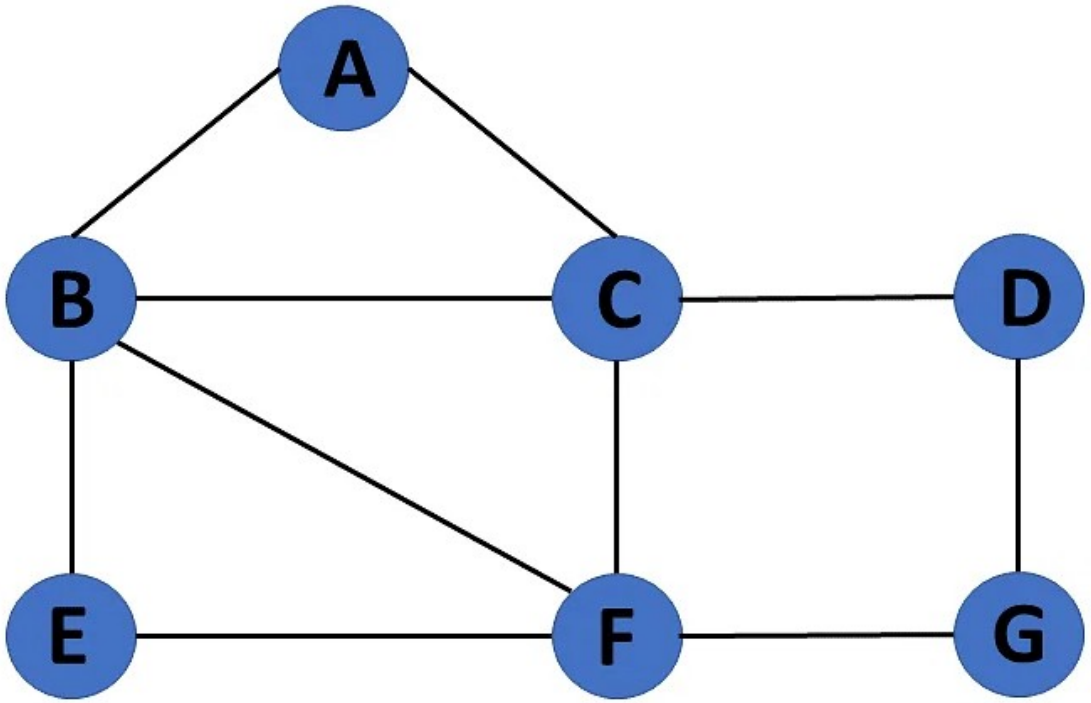
Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabeled.



Types of Graphs in Data Structures

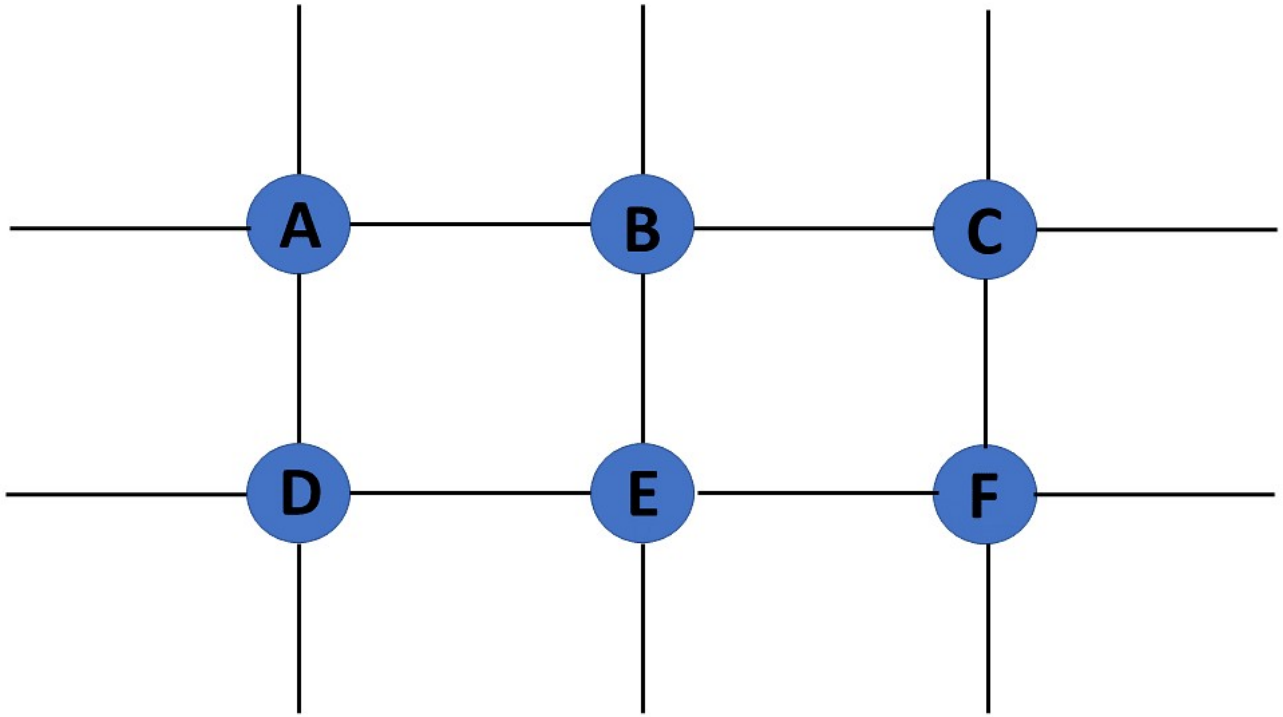
1. Finite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is limited in number



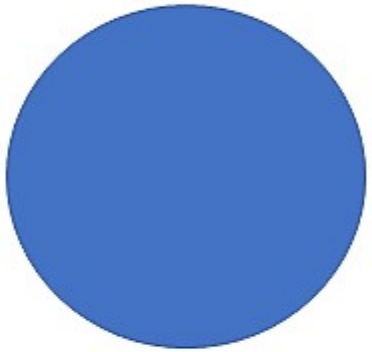
2. Infinite Graph

The graph G=(V, E) is called a finite graph if the number of vertices and edges in the graph is interminable.



3. Trivial Graph

A graph G= (V, E) is trivial if it contains only a single vertex and no edges.



Graph Traversal Algorithm

Breadth First Search

Breadth First Search

The Breadth First Search (BFS) algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.

Relation between BFS for Graph and Tree traversal

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree.

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

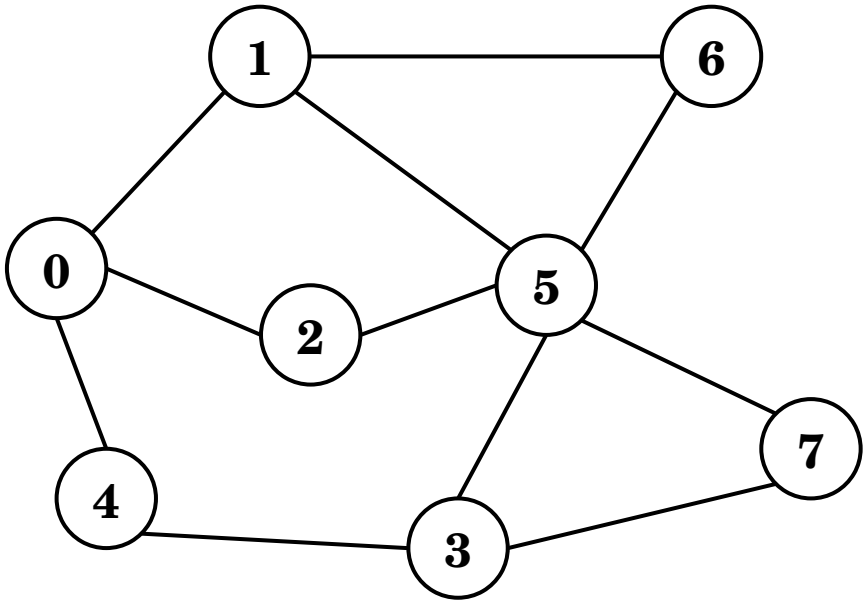
- Visited and
- Not visited.

A Boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a queue data structure for traversal.

How does BFS work?

Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.

To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.



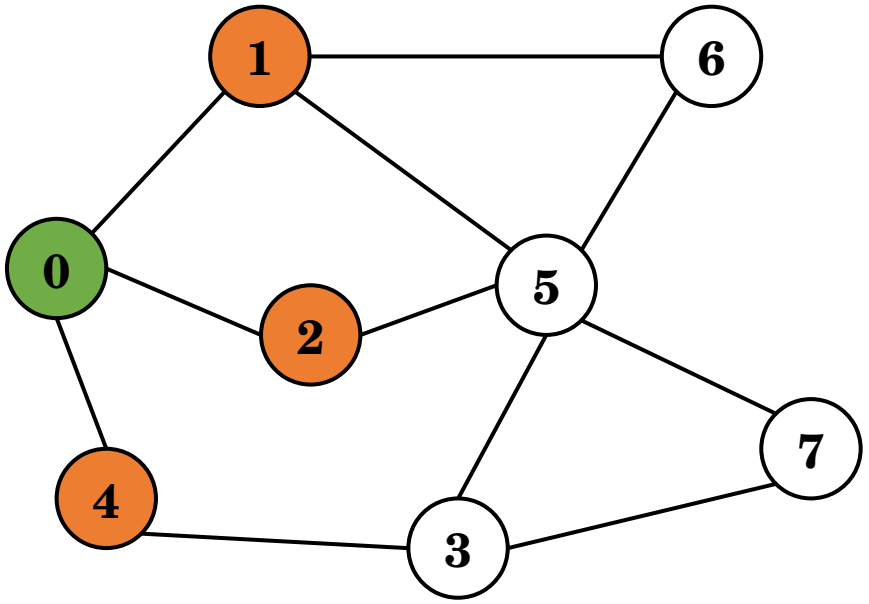
Explored	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7

Queue								

Output								

Initial Steps:

1. Visit ***node (0)*** [make it current], insert it into ***Output*** and change its explored status to ***1***.
2. Find the neighbors of ***node (0)*** and enqueue them into ***Queue***
3. Change the explored status of those vertices which are just included into the ***Queue***



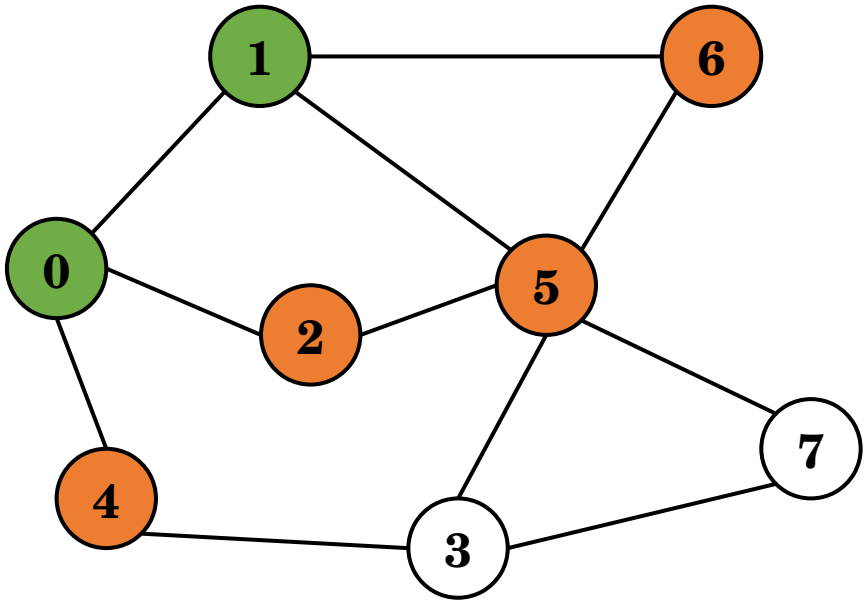
Explored	1	1	1	0	1	0	0	0
	0	1	2	3	4	5	6	7

Queue	1	2	4					

Output	0							

<u>Repeating Steps (do until queue is not empty):</u>

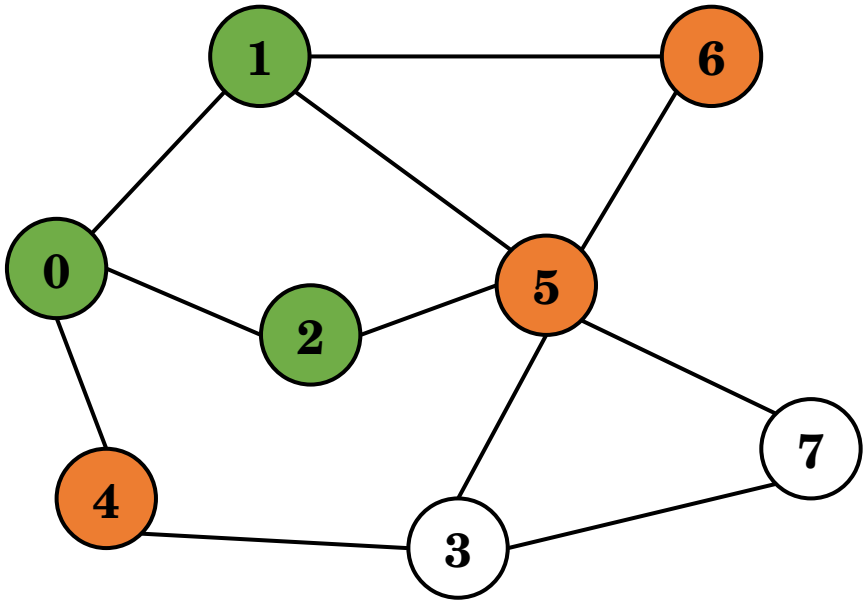
1. Visit the first node (let's say ***node (x)***) of the ***Queue*** [make it current]**,** dequeue it and insert it into ***Output*** Find the neighbors of ***current*** and enqueue them into ***Queue*** if their explored status is ***0***
2. Change the explored status of those vertices which are recently enqueued into ***Queue***.



Explored	1	1	1	0	1	1	1	0
	0	1	2	3	4	5	6	7

Queue	2	4	5	6				

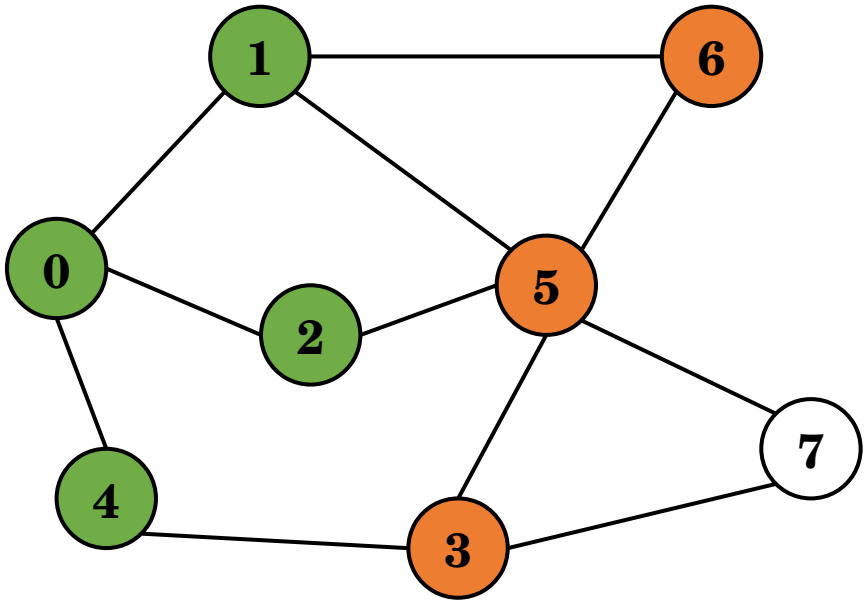
Output	0	1						



Explored	1	1	1	0	1	1	1	0
	0	1	2	3	4	5	6	7

Queue	4	5	6					

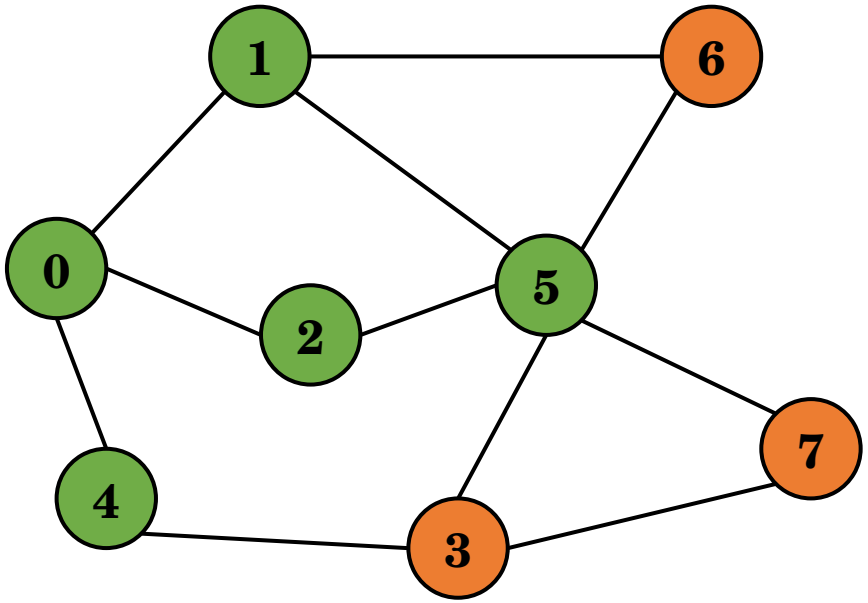
Output	0	1	2					



Explored	1	1	1	1	1	1	1	0
	0	1	2	3	4	5	6	7

Queue	5	6	3					

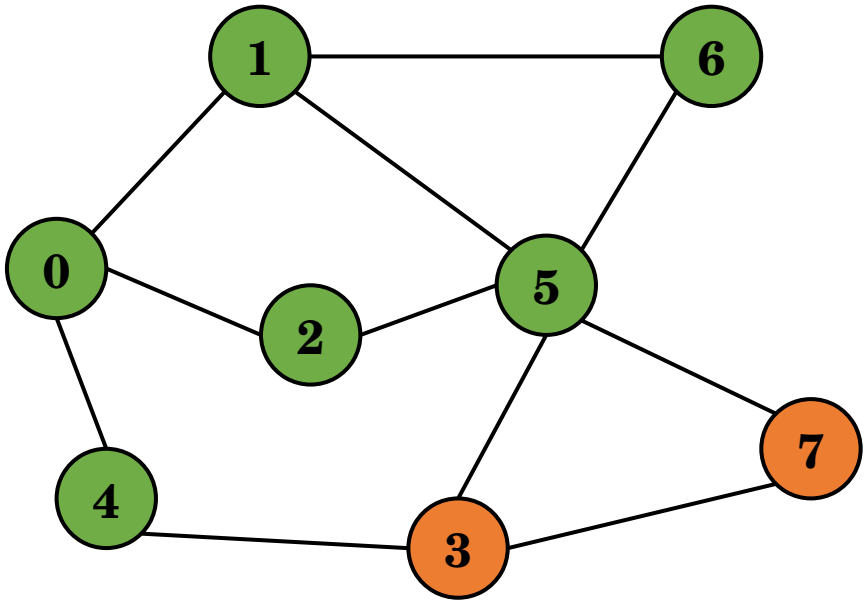
Output	0	1	2	4				



Explored	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7

Queue	6	3	7					

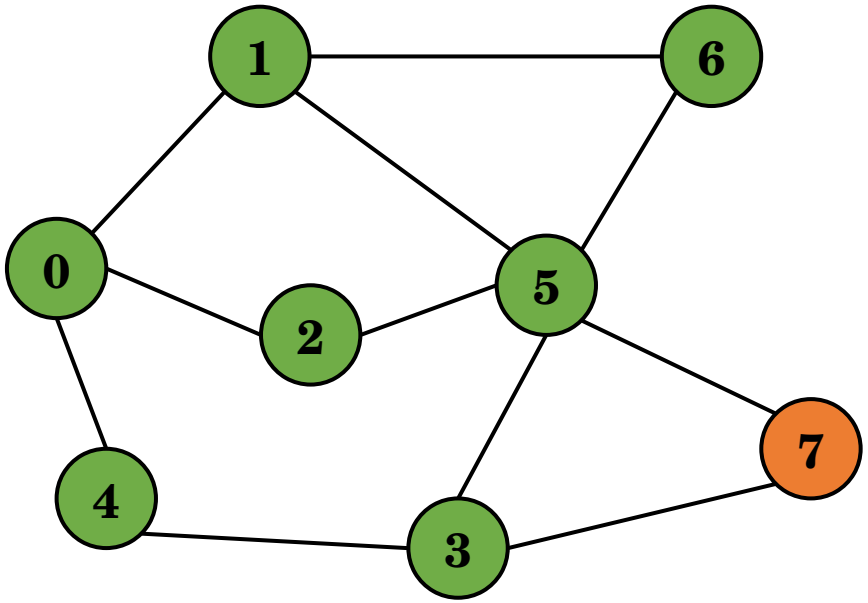
Output	0	1	2	4	5			



Explored	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7

Queue	3	7						

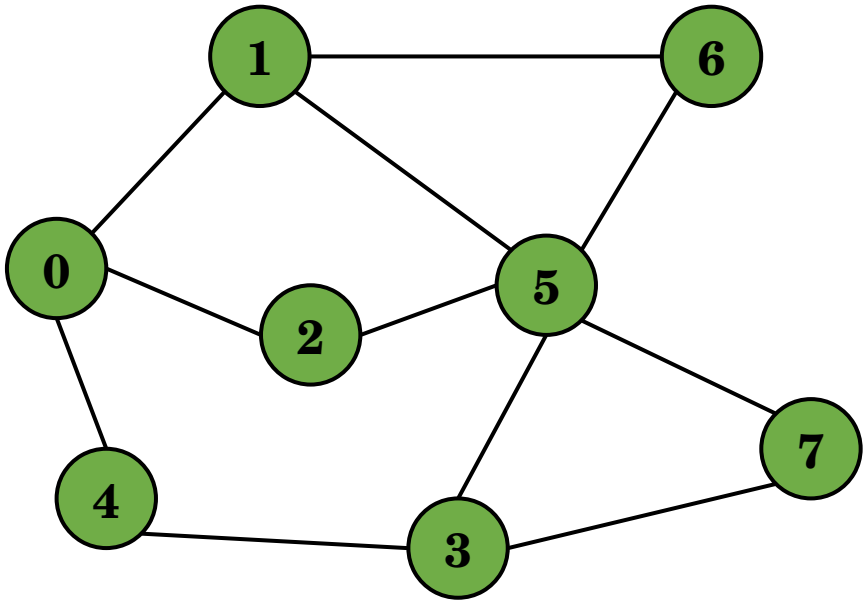
Output	0	1	2	4	5	6		



Explored	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7

Queue	7							

Output	0	1	2	4	5	6	3	



Explored	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7

Queue								

Output	0	1	2	4	5	6	3	7

Applications of Breadth First Search

1. Shortest Path and Minimum Spanning Tree for unweighted graph:

In an unweighted graph, the shortest path is the path with the least number of edges. With Breadth First, we always reach a vertex from a given source using the minimum number of edges. Also, in the case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.

2. Minimum Spanning Tree for weighted graphs:

We can also find Minimum Spanning Tree for weighted graphs using BFT, but the condition is that the weight should be non-negative and the same for each pair of vertices.

Applications of Breadth First Search

3. Peer-to-Peer Networks: In Peer-to-Peer Networks like BitTorrent, Breadth First Search is used to find all neighbor nodes.

4. Crawlers in Search Engines:

Crawlers build an index using Breadth First. The idea is to start from the source page and follow all links from the source and keep doing the same. Depth First Traversal can also be used for crawlers, but the advantage of Breadth First Traversal is, the depth or levels of the built tree can be limited.

5. Social Networking Websites:

In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.

Applications of Breadth First Search

6. GPS Navigation systems: Breadth First Search is used to find all neighboring locations.

7. Broadcasting in Network: In networks, a broadcasted packet follows Breadth First Search to reach all nodes.

8. In Garbage Collection: Breadth First Search is used in copying garbage collection using Cheney's algorithm. Breadth First Search is preferred over Depth First Search because of a better locality of reference.

9. Cycle detection in undirected graph: In undirected graphs, either Breadth First Search or Depth First Search can be used to detect a cycle. We can use BFS to detect cycle in a directed graph also.

Applications of Breadth First Search

10. Ford–Fulkerson algorithm In Ford – Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces the worst-case time complexity to $O(VE^2)$.

11. To test if a graph is Bipartite: We can either use Breadth First or Depth First Traversal.

12. Path Finding: We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.

13. Finding all nodes within one connected component: We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

Applications of Breadth First Search

14. AI: In AI, BFS is used in traversing a game tree to find the best move.

15. Network Security: In the field of network security, BFS is used in traversing a network to find all the devices connected to it.

16. Connected Component: We can find all connected components in an undirected graph.

17. Topological sorting: BFS can be used to find a topological ordering of the nodes in a directed acyclic graph (DAG).

18. Image processing: BFS can be used to flood-fill an image with a particular color or to find connected components of pixels

Applications of Breadth First Search

19. Recommender systems: BFS can be used to find similar items in a large dataset by traversing the items' connections in a similarity graph.

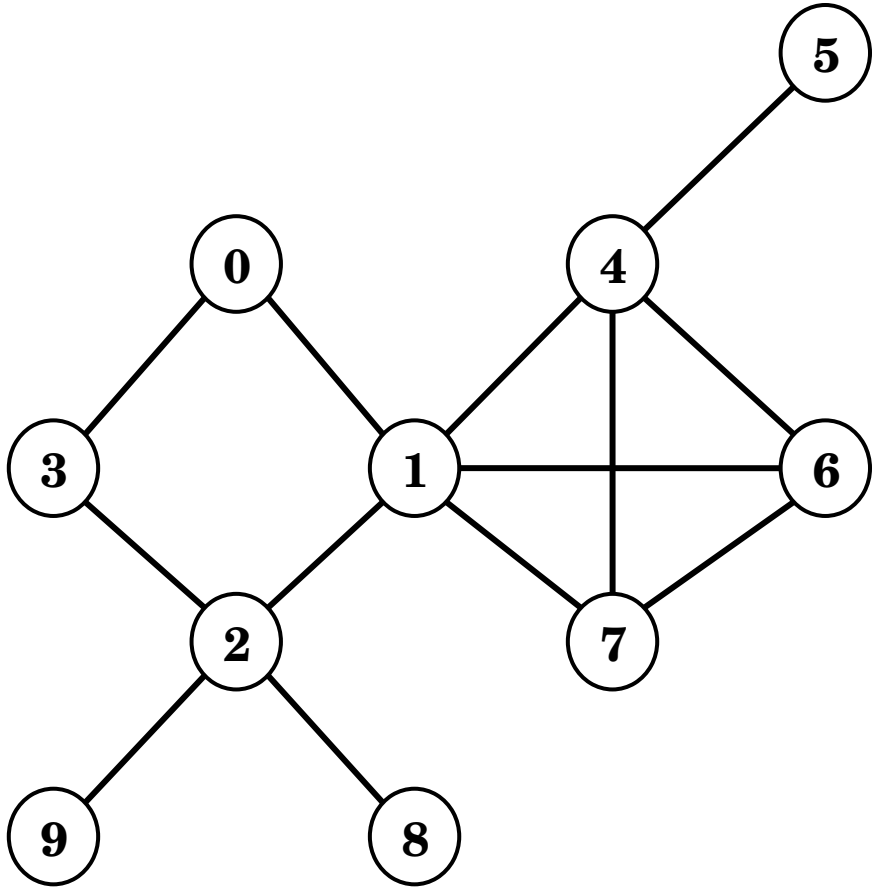
20. Other usages: Many algorithms like Prim's Minimum Spanning Tree and Dijkstra's Single Source Shortest Path use structures similar to Breadth First Search.

Advantages of BFS

19. Recommender systems: BFS can be used to find similar items in a large dataset by traversing the items' connections in a similarity graph.

20. Other usages: Many algorithms like Prim's Minimum Spanning Tree and Dijkstra's Single Source Shortest Path use structures similar to Breadth First Search.

Depth First Search



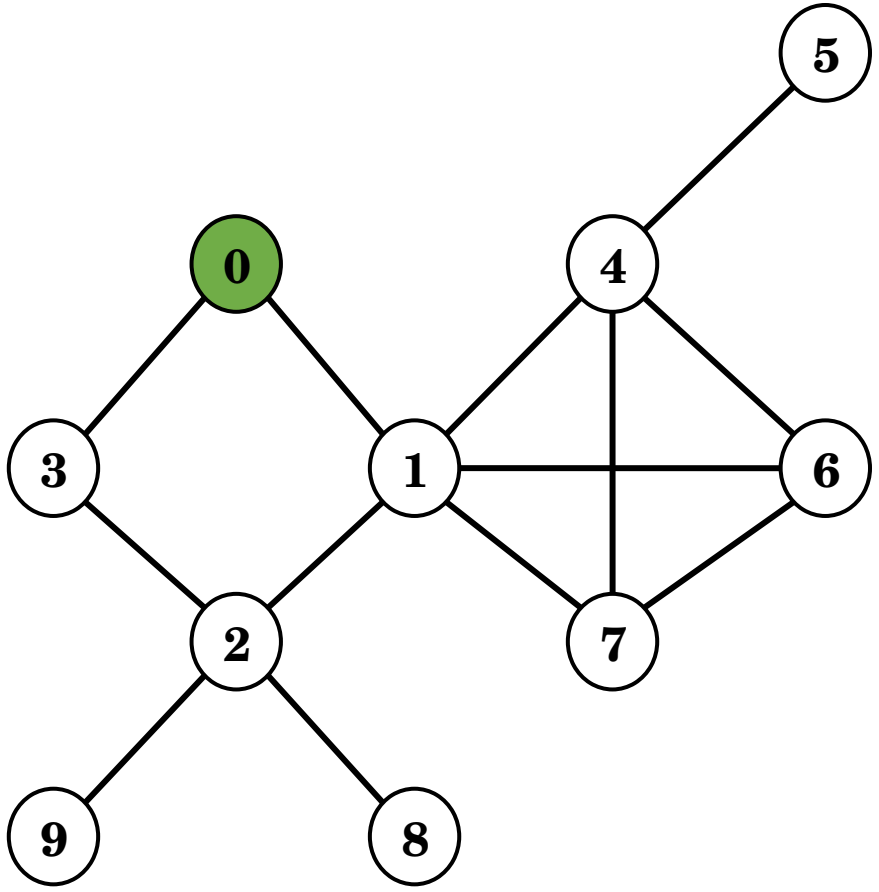
Initial Steps:

1. Visit ***node (0)*** [make current =0]
2. Insert ***node (0)*** into output
3. Change the visited status of ***node (0)*** into 1
4. Push ***node (0) into the stack***

Visited	0	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9

Output										

Stack



Repeating steps (do until stack is empty)

1. Look for unvisited neighbors of ***current*** node. If there is any unvisited neighbors of current node, perform step 2. Otherwise perform step 3.

2. From the unvisited neighbors of ***current*** node, select anyone and make it new ***current***.

Insert current into ***output***. Change its visited status. Push it into the stack. Repeat step 1.

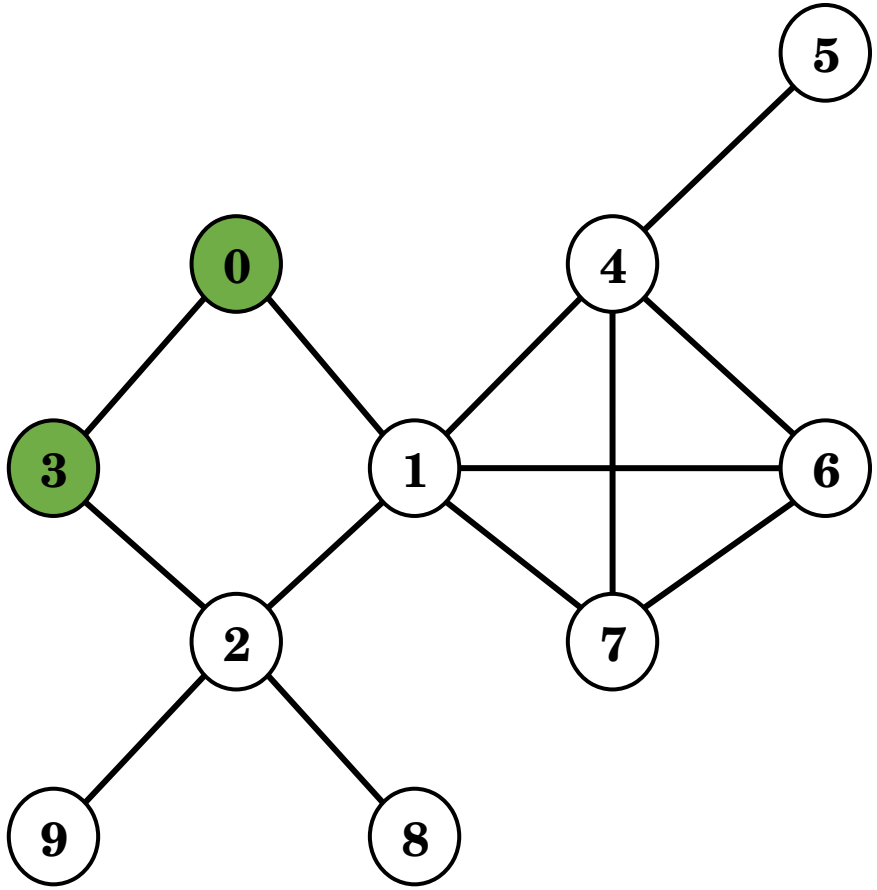
3. If there is no unvisited neighbor of the current node, then pop from the stack, after popping make the top element as new ***current*** and repeat step 2.

Visited	1	0	0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9

Output	0									

0

Stack

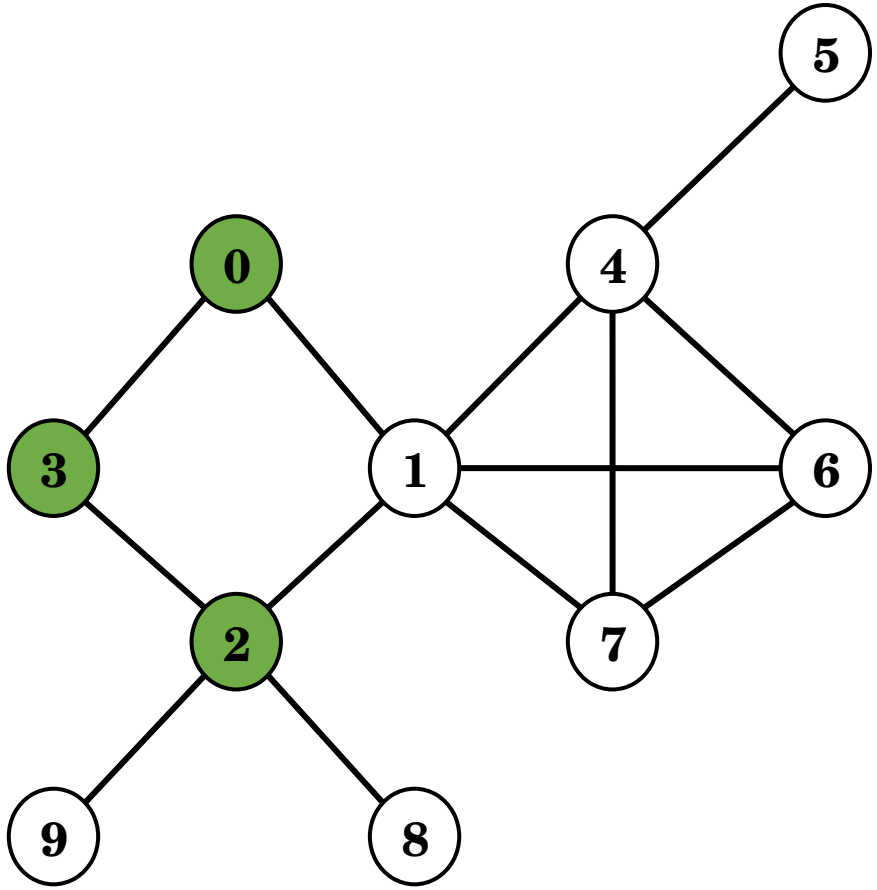


Visited	1	0	0	1	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9

Output	0	3								

3
0

Stack

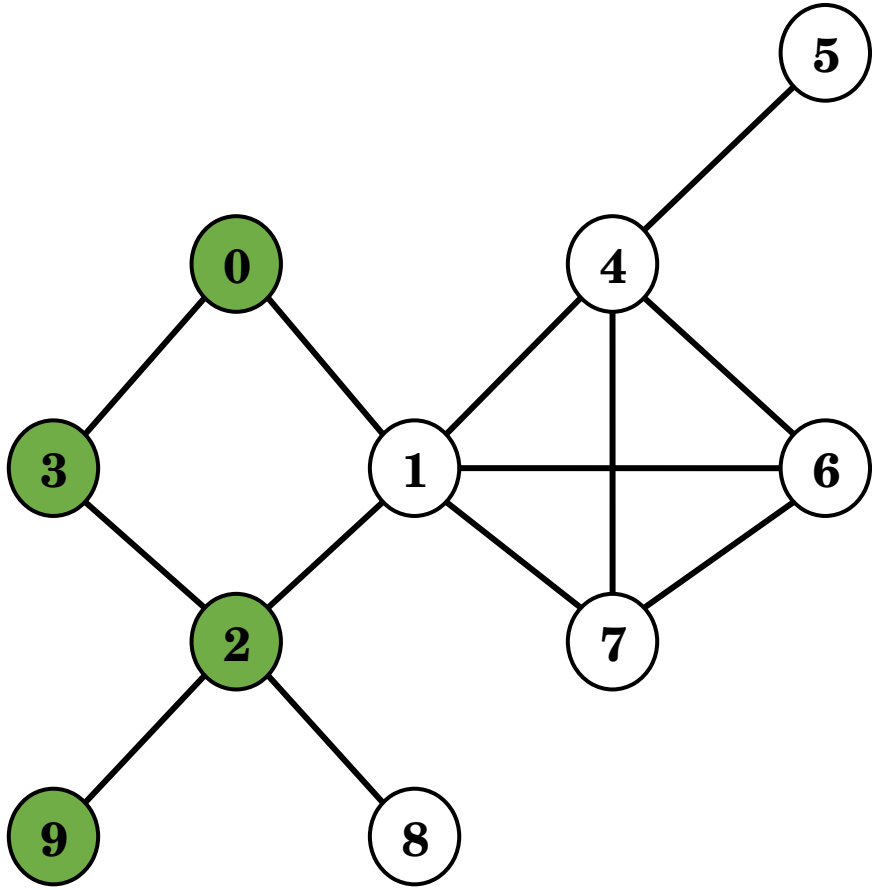


Visited	1	0	1	1	0	0	0	0	0	0
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2							

2
3
0

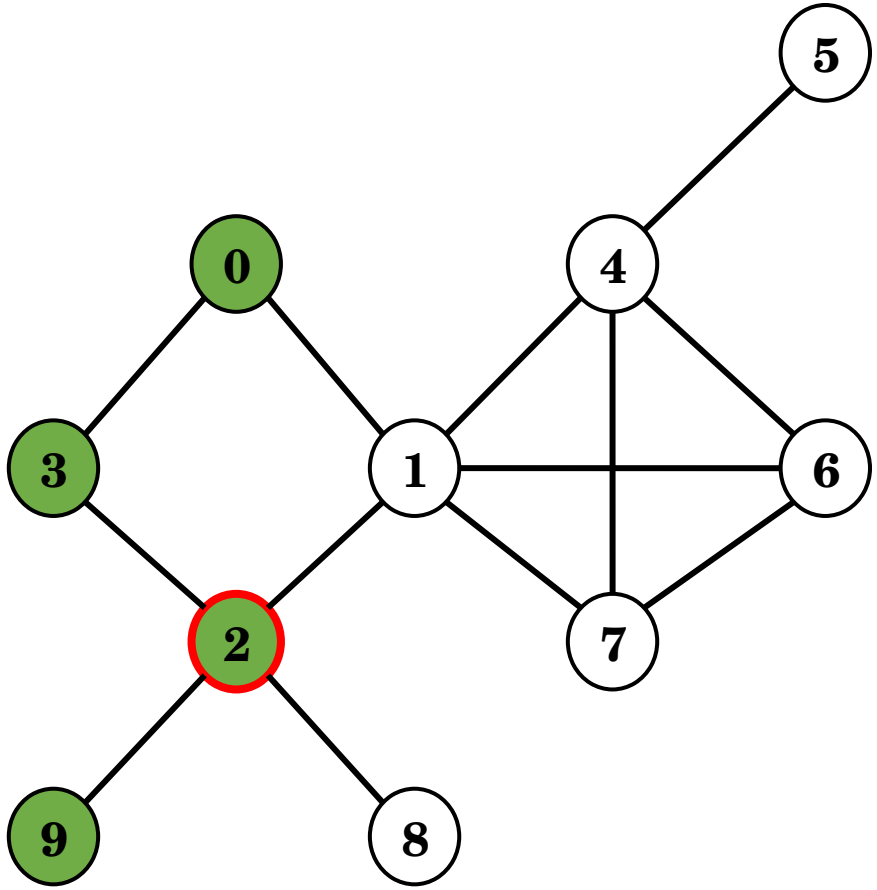
Stack



	0	1	2	3	4	5	6	7	8	9
Visited	1	0	1	1	0	0	0	0	0	1

Output										
	0	3	2	9						

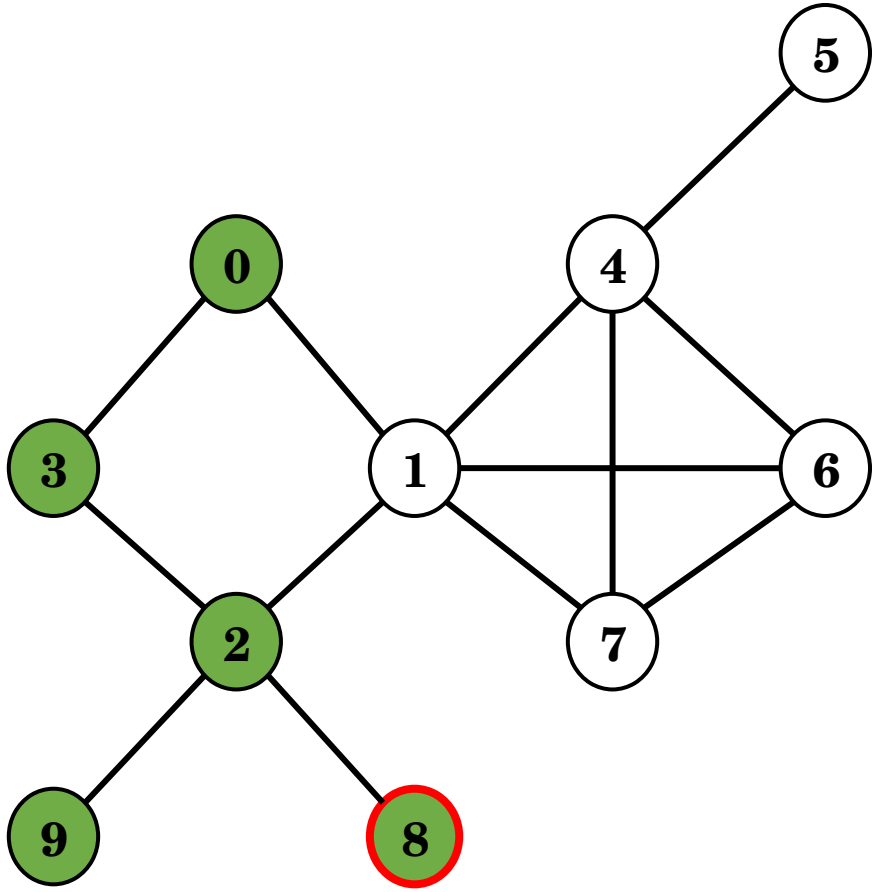
Stack
9
2
3
0



	0	1	2	3	4	5	6	7	8	9
Visited	1	0	1	1	0	0	0	0	0	1

Output	0	3	2	9						

Stack
2
3
0

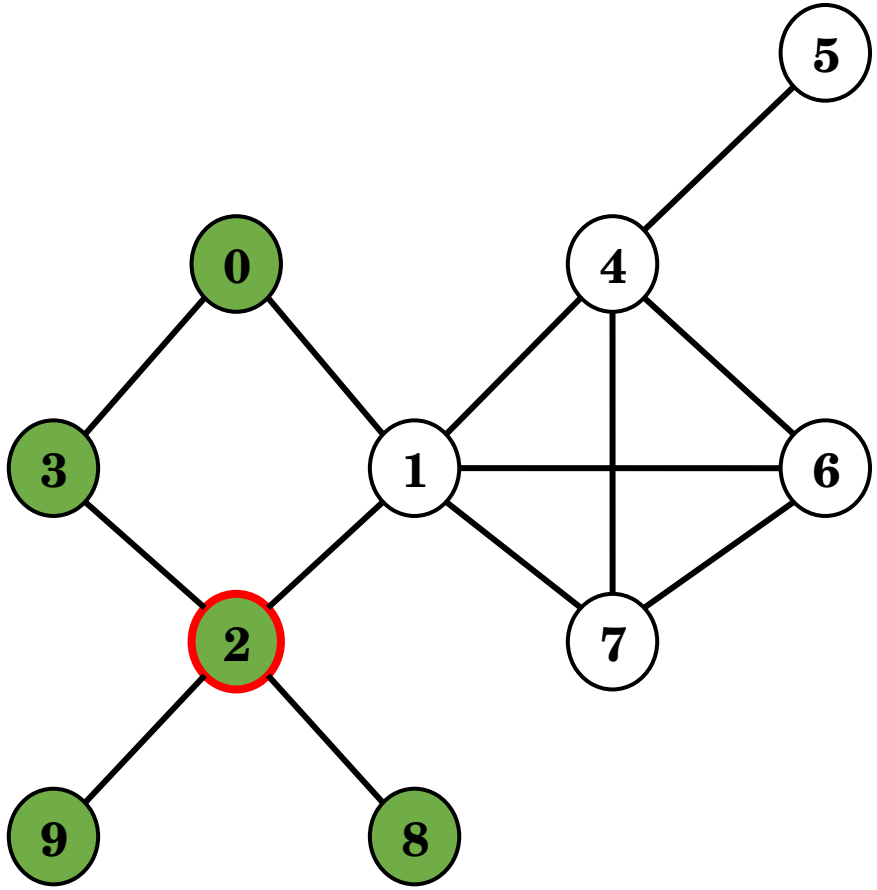


Visited	1	0	1	1	0	0	0	0	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8					

8
2
3
0

Stack

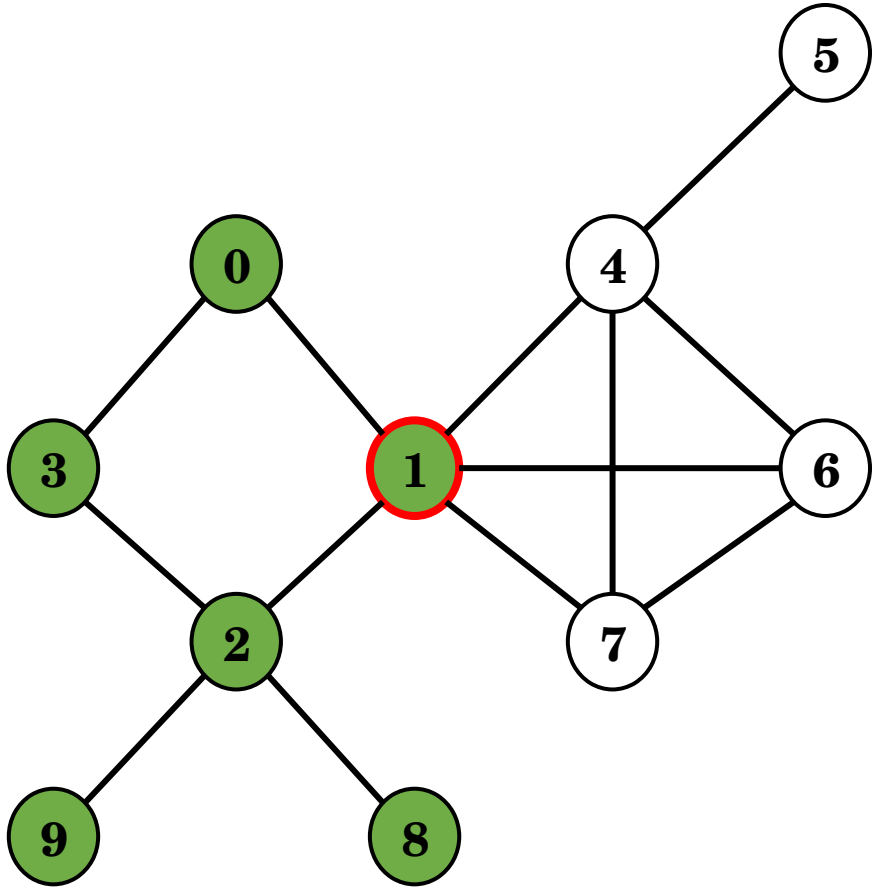


Visited	1	0	1	1	0	0	0	0	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8					

2
3
0

Stack

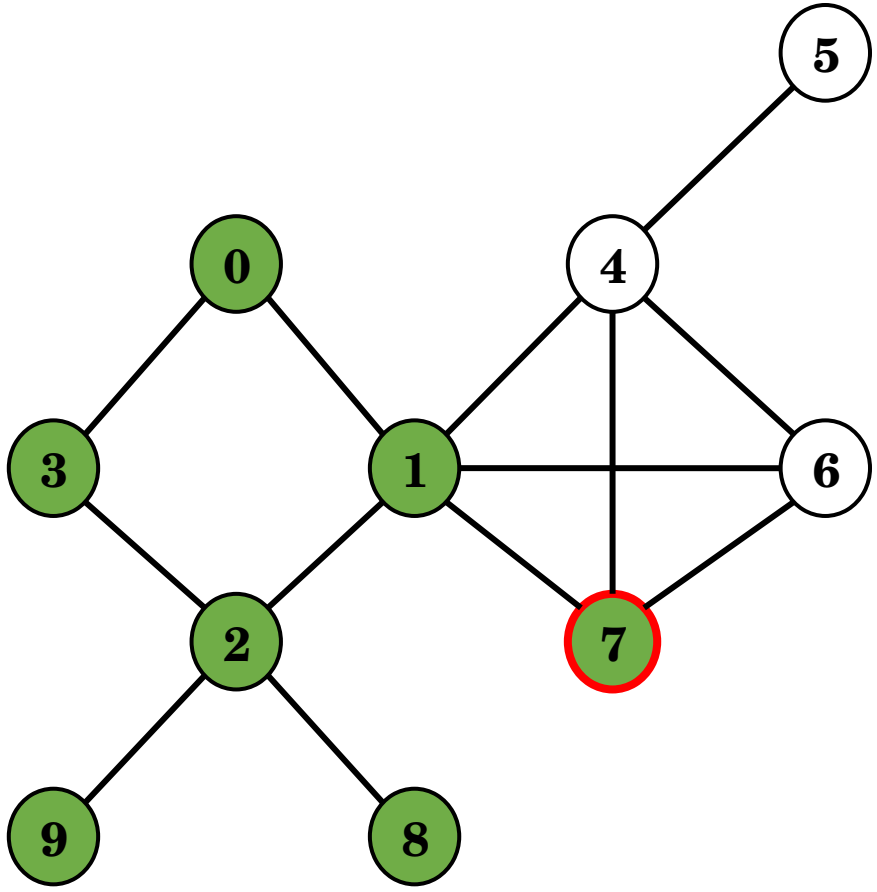


	0	1	2	3	4	5	6	7	8	9
Visited	1	1	1	1	0	0	0	0	1	1

Output	0	3	2	9	8	1				

1
2
3
0

Stack

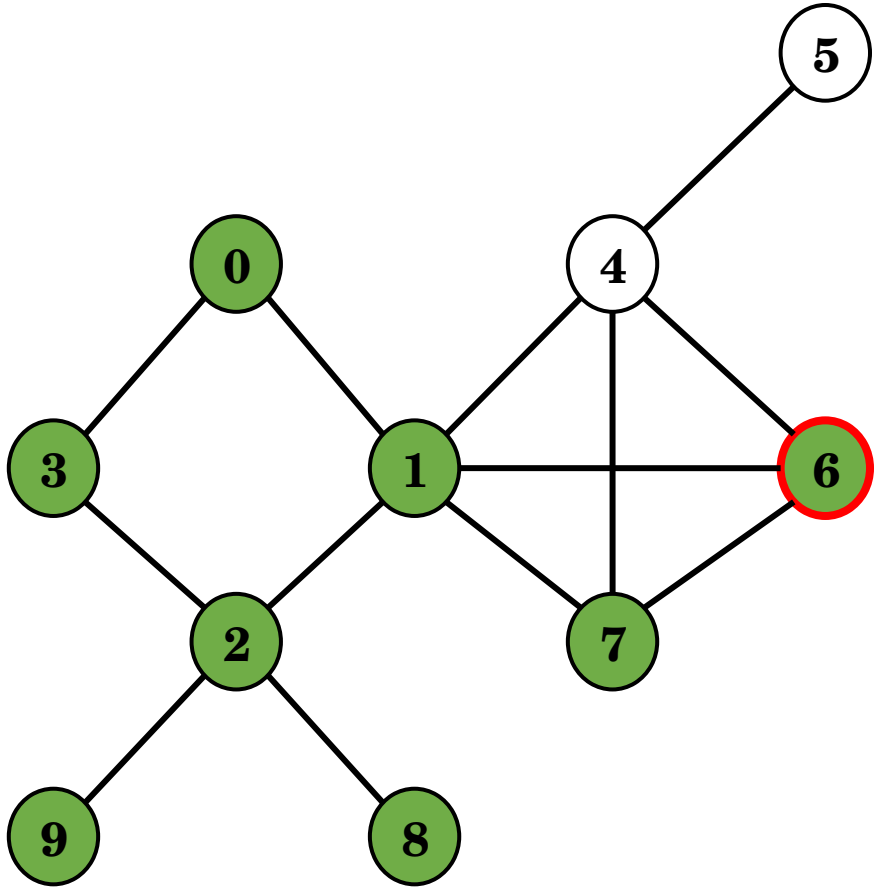


Visited	1	1	1	1	0	0	0	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7			

7
1
2
3
0

Stack

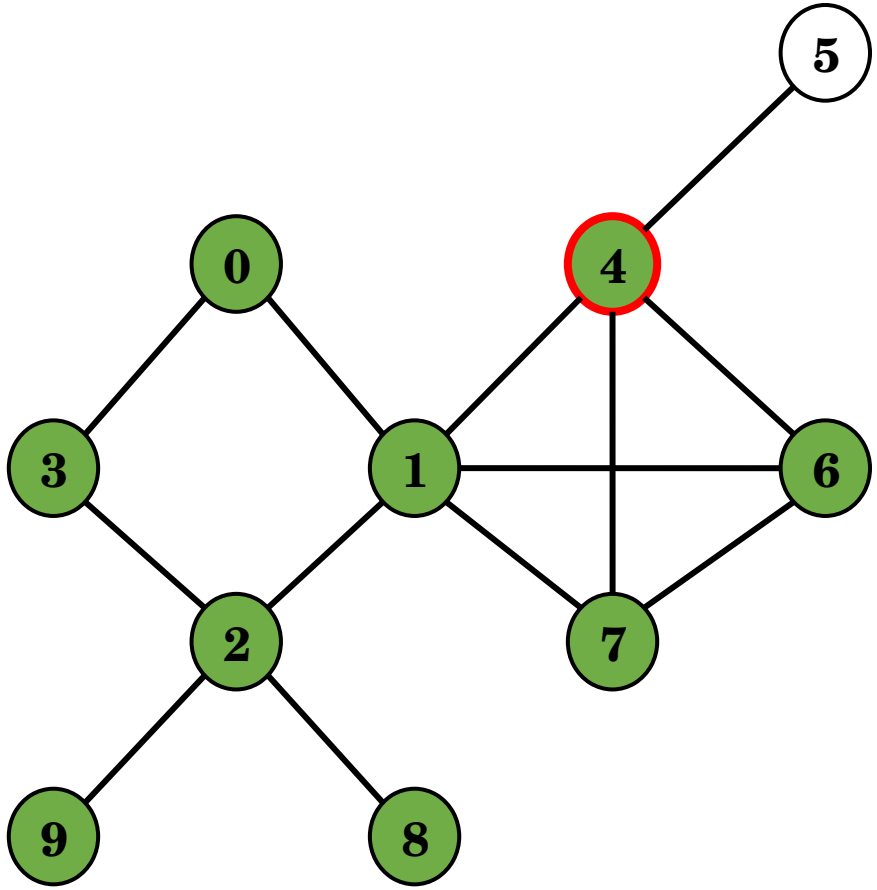


Visited	1	1	1	1	0	0	1	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7	6		

6
7
1
2
3
0

Stack

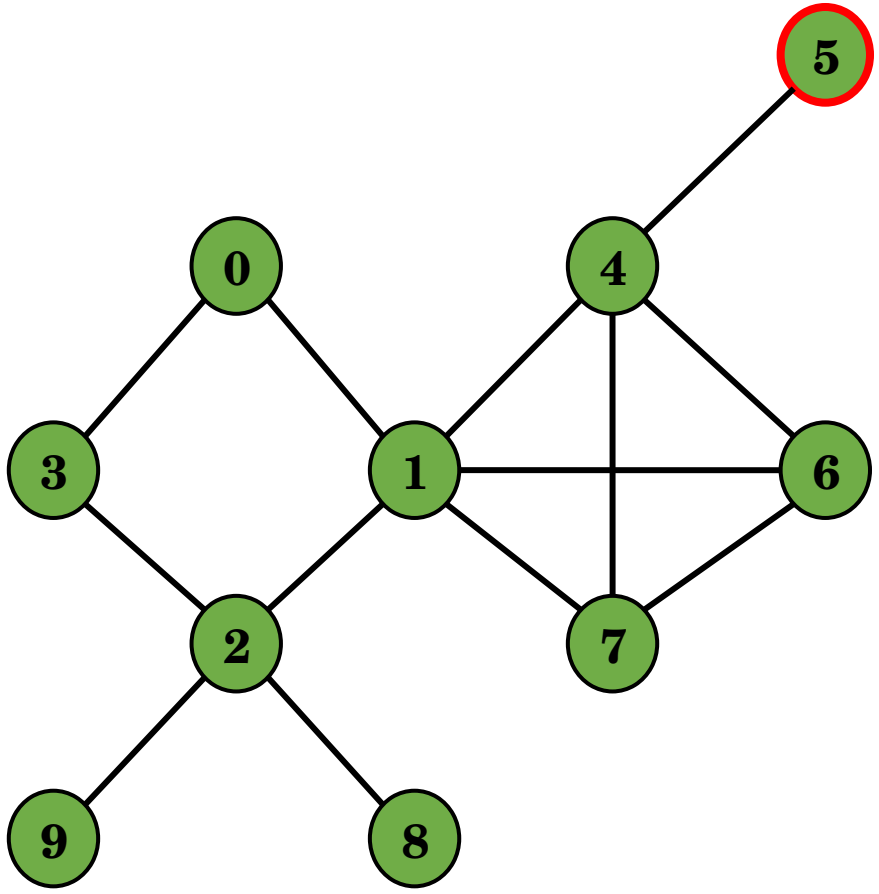


Visited	1	1	1	1	1	0	1	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7	6	4	

4
6
7
1
2
3
0

Stack

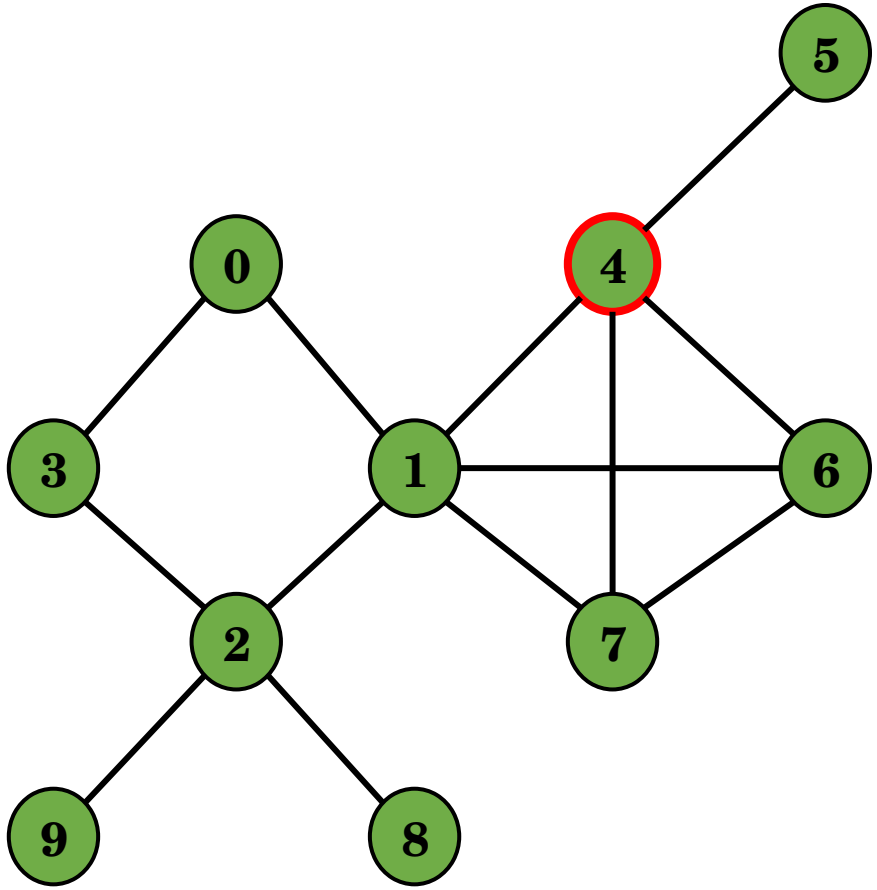


Visited	1	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7	6	4	5

5
4
6
7
1
2
3
0

Stack

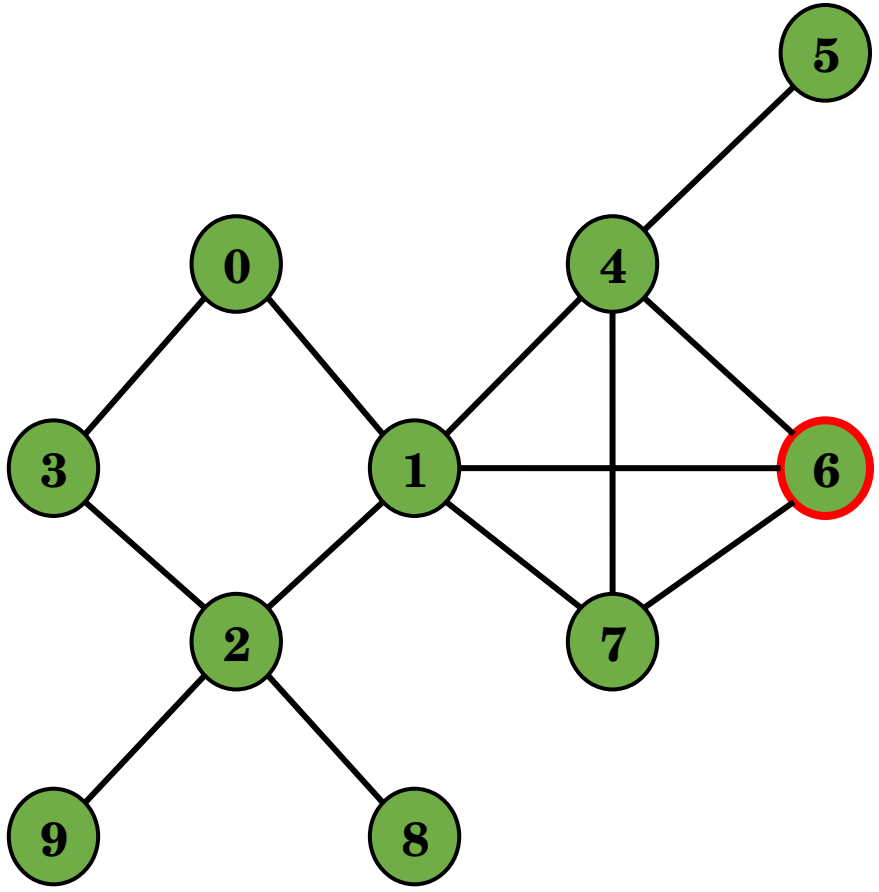


Visited	1	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7	6	4	5

4
6
7
1
2
3
0

Stack

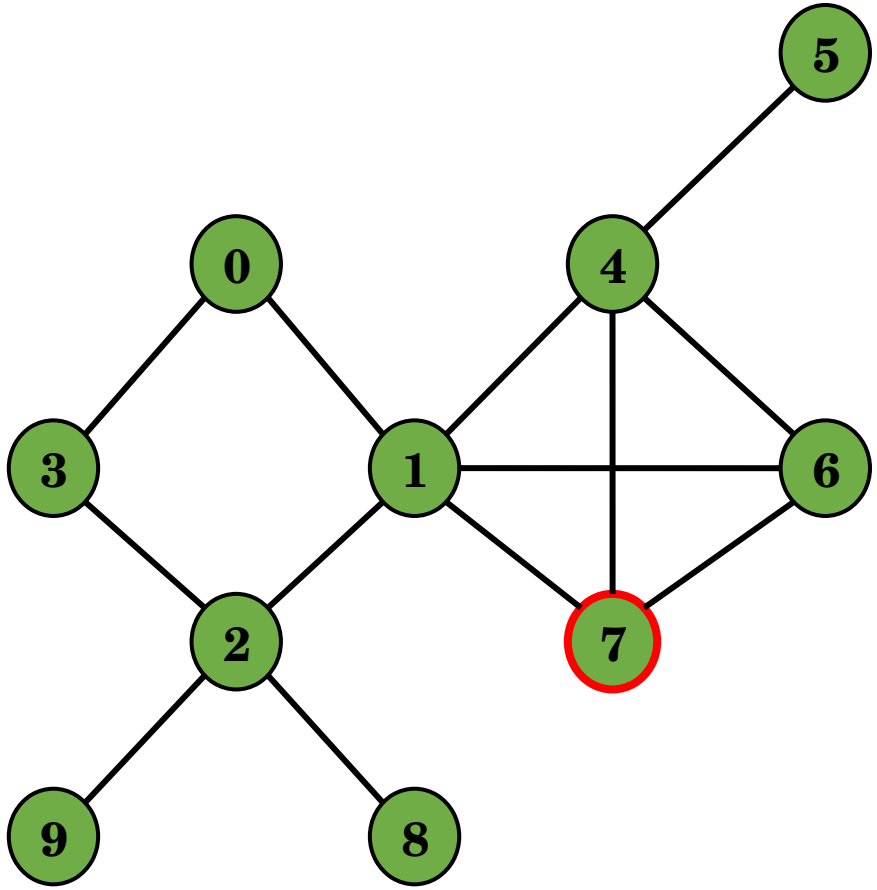


Visited	1	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7	6	4	5

6
7
1
2
3
0

Stack

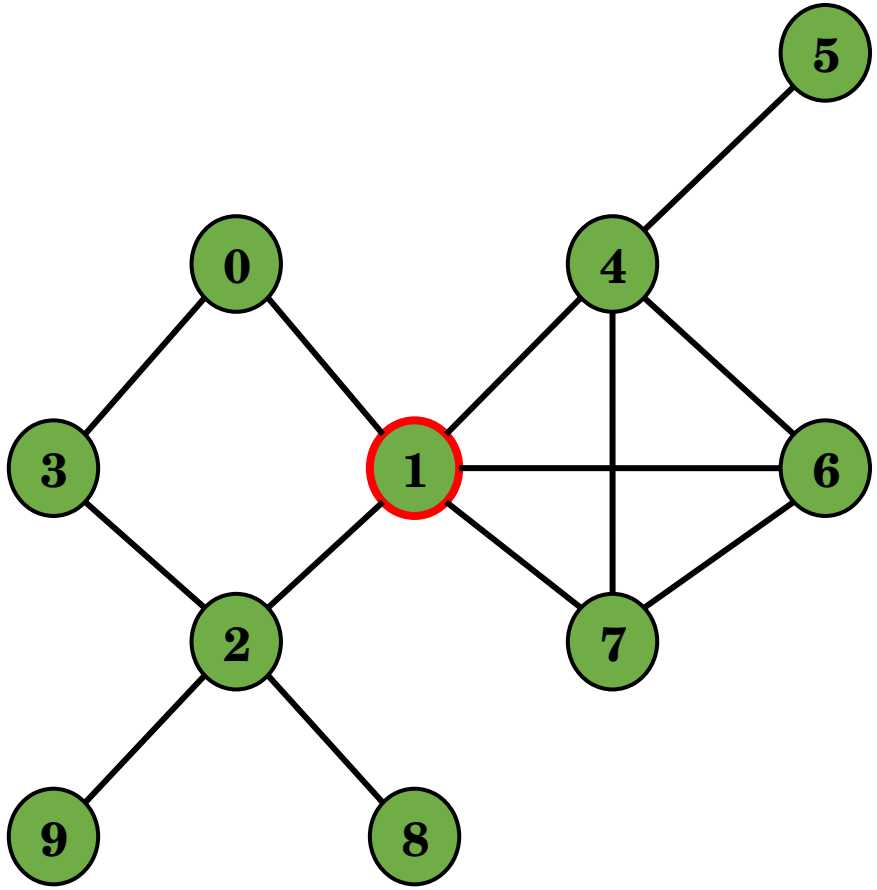


Visited	1	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7	6	4	5

7
1
2
3
0

Stack

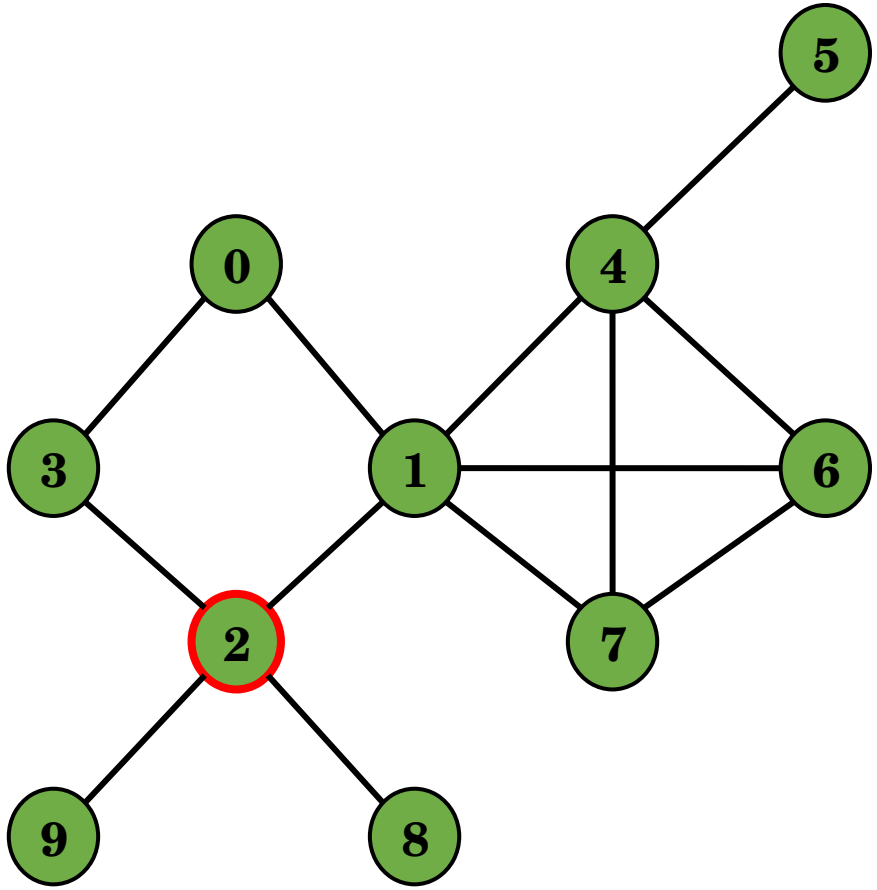


Visited	1	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7	6	4	5

1
2
3
0

Stack

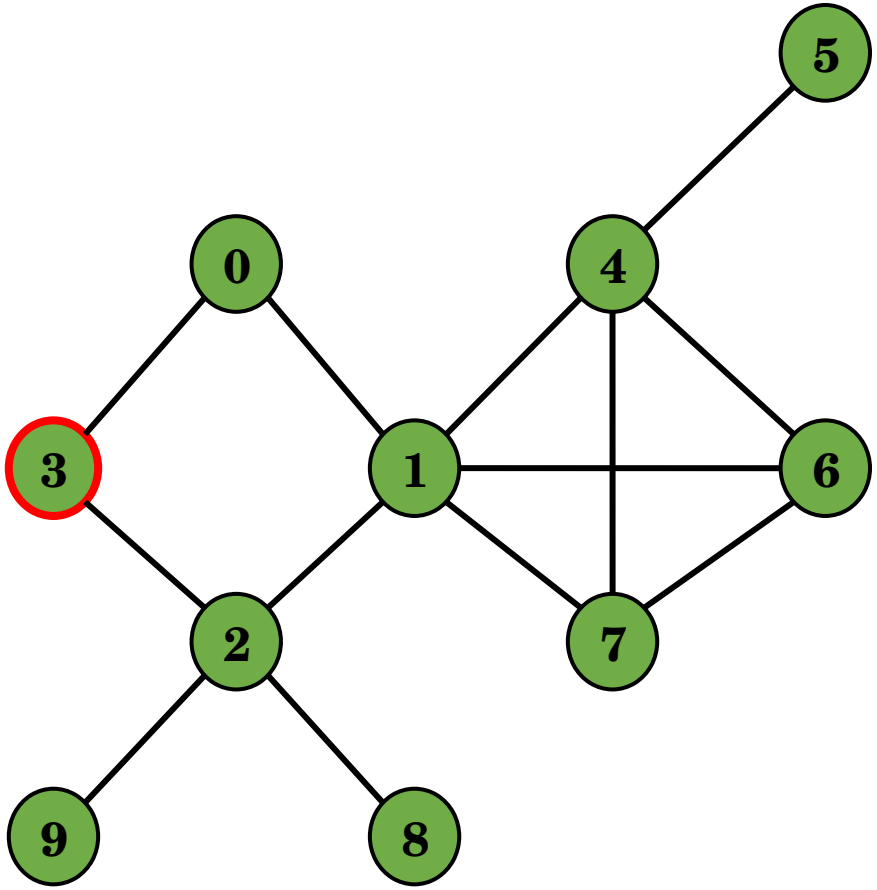


Visited	1	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7	6	4	5

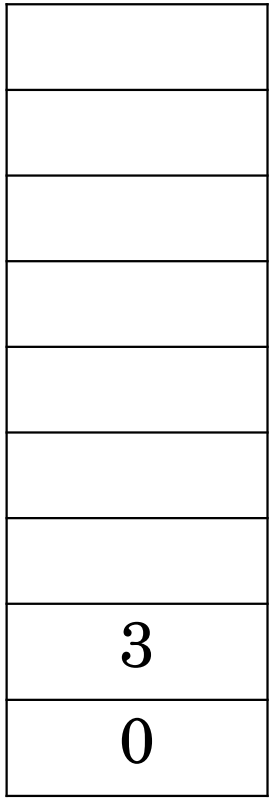
2
3
0

Stack

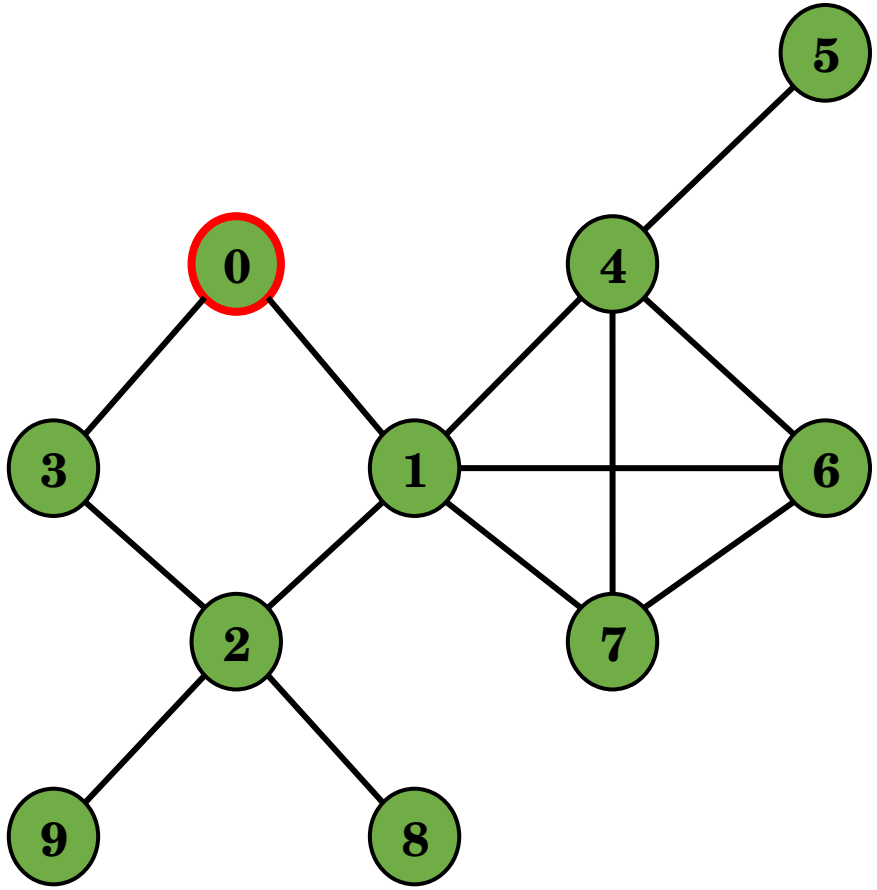


Visited	1	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7	6	4	5

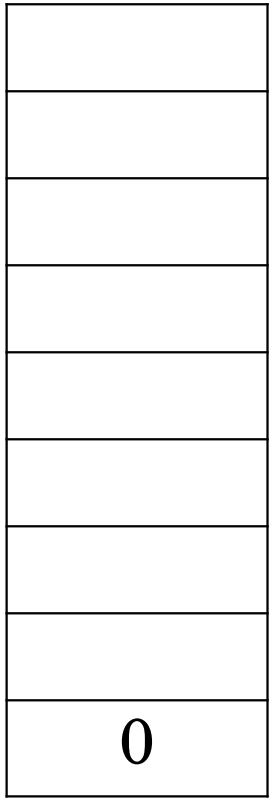


Stack

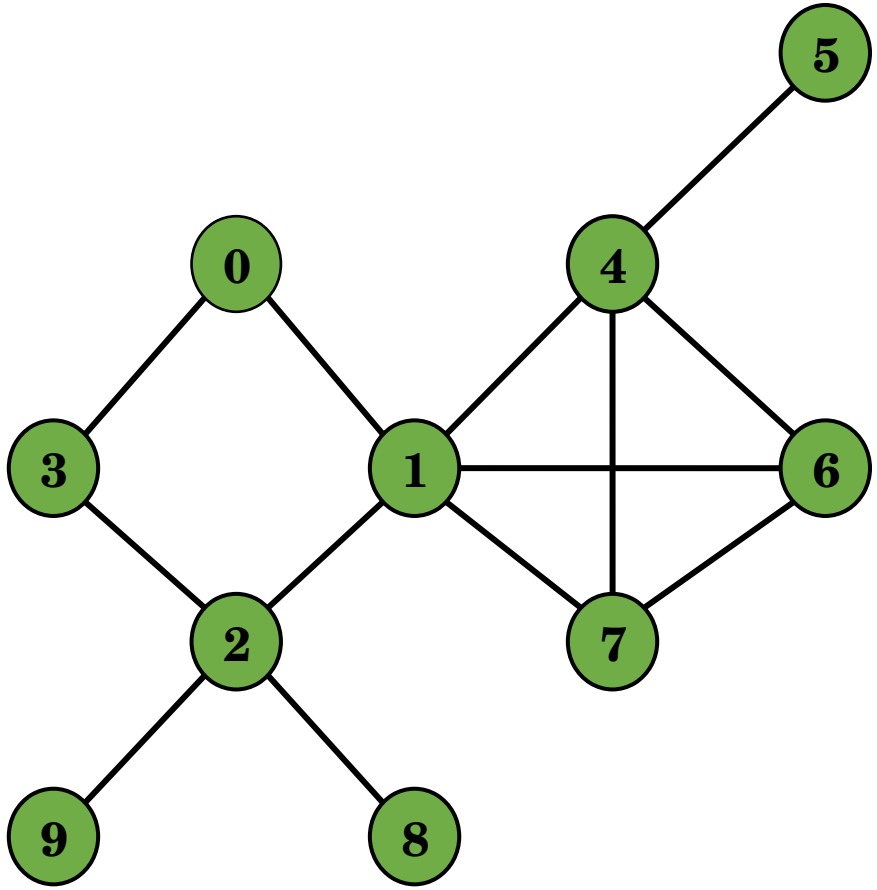


Visited	1	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7	6	4	5

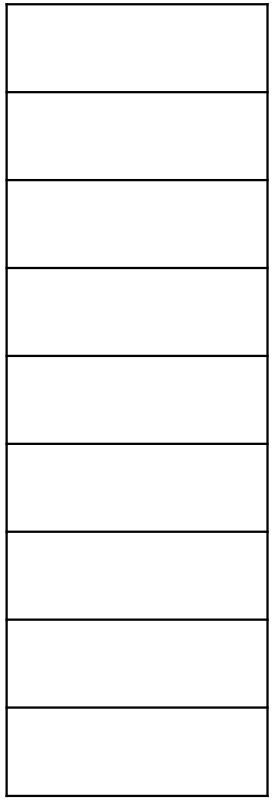


Stack



Visited	1	1	1	1	1	1	1	1	1	1
	0	1	2	3	4	5	6	7	8	9

Output	0	3	2	9	8	1	7	6	4	5



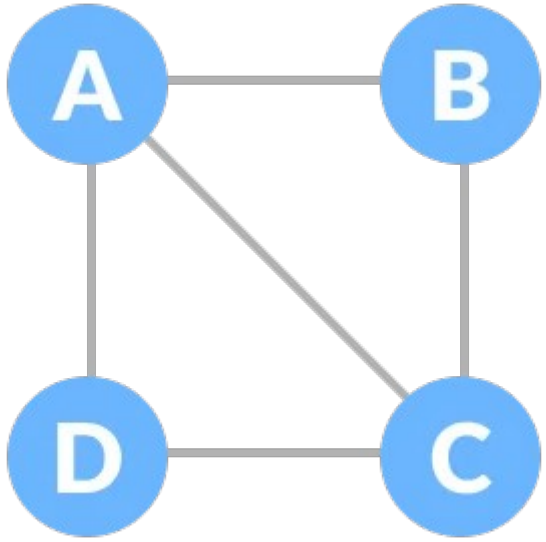
Stack

Minimum Spanning Tree

Undirected and Connected Graph

Before we learn about spanning trees, we need to understand two graphs: undirected graphs and connected graphs.

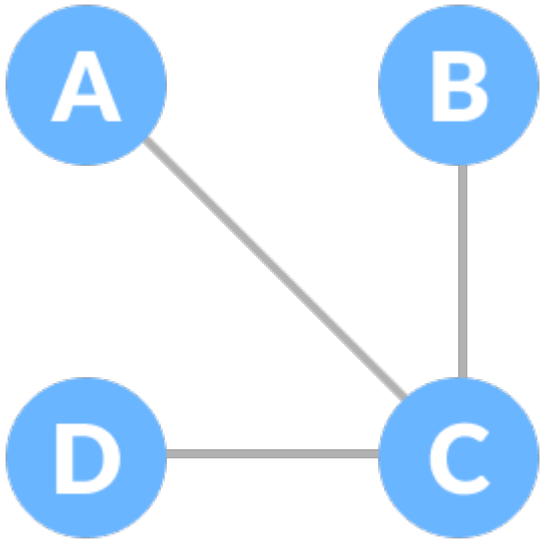
An ***undirected graph*** is a graph in which the edges do not point in any direction (i.e.. the edges are bidirectional).



Undirected Graph

Undirected and Connected Graph

A ***connected graph*** is a graph in which there is always a path from a vertex to any other vertex.

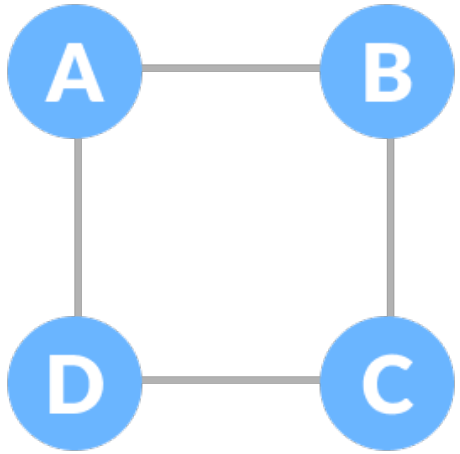


Connected Graph

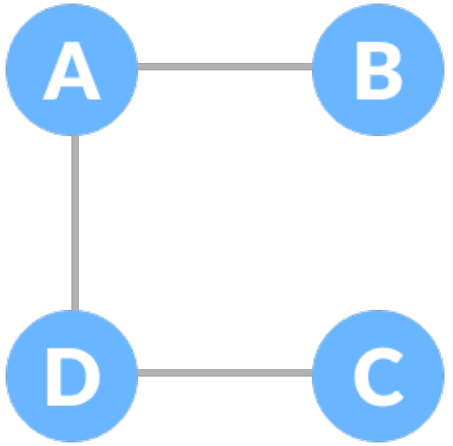
Spanning Tree

- A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree. The edges may or may not have weights assigned to them.
- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.
- By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

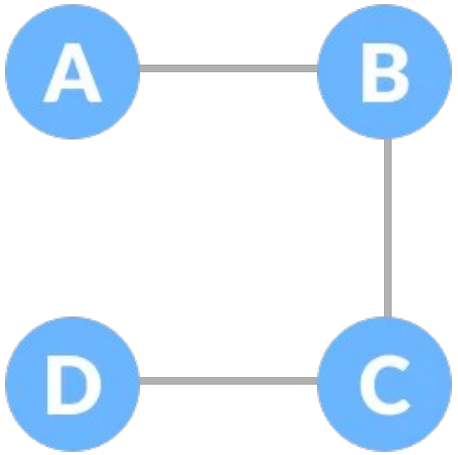
Spanning Tree Example



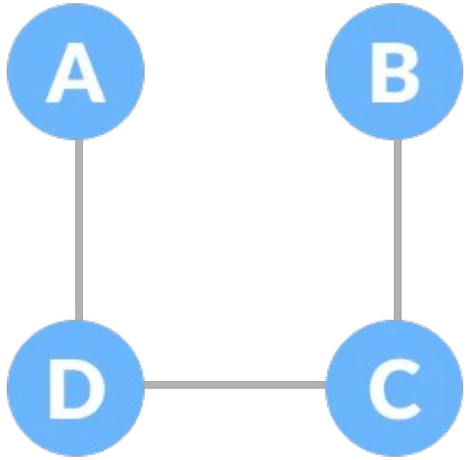
Normal Graph



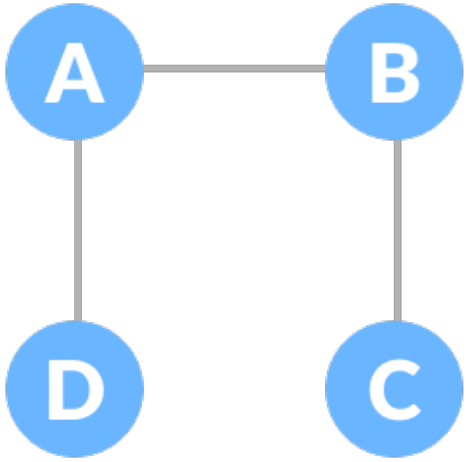
Spanning Tree 1



Spanning Tree 2

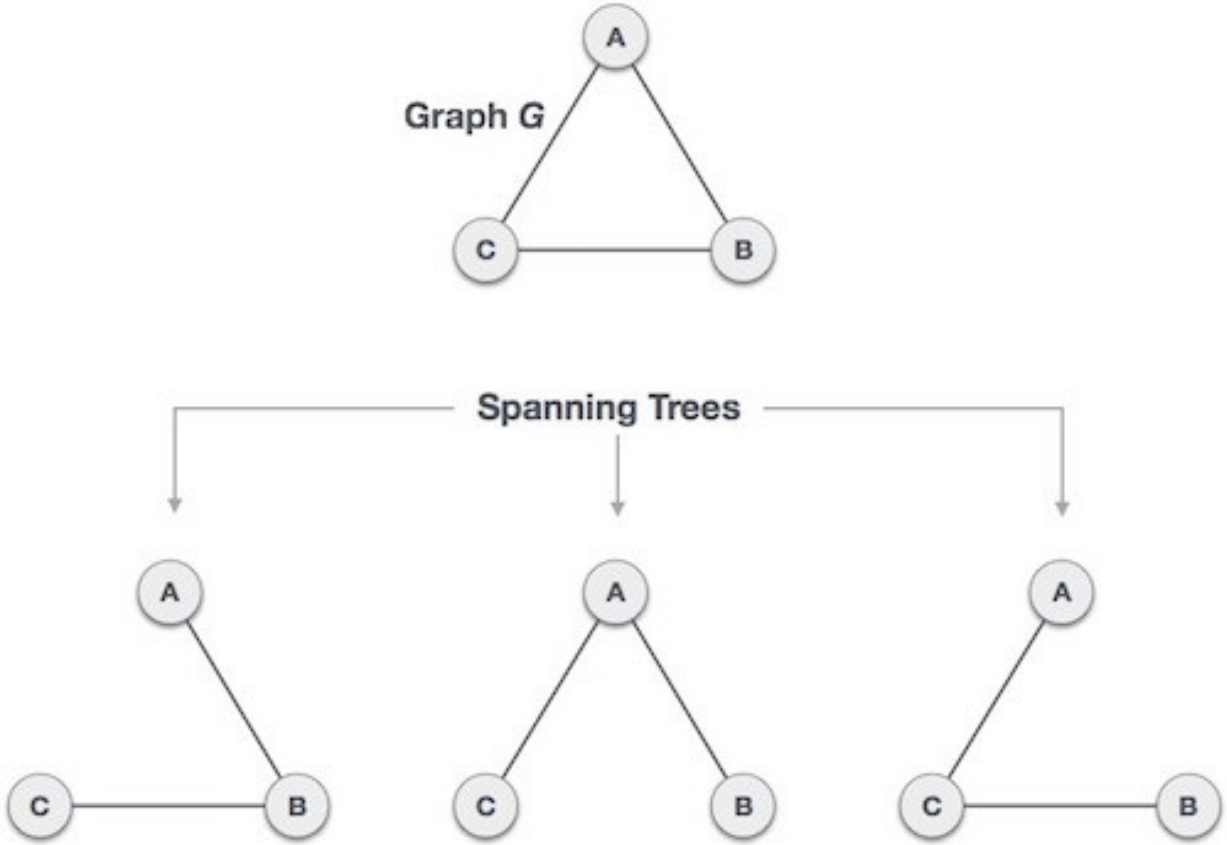


Spanning Tree 3



Spanning Tree 4

Spanning Tree Example



We found three spanning trees off one complete graph. A complete undirected graph can have maximum $n^{(n-2)}$ number of spanning trees, where n is the number of nodes. In the above addressed example, n is 3,
hence $3^{(3-2)}$= 3 spanning trees are possible.

General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

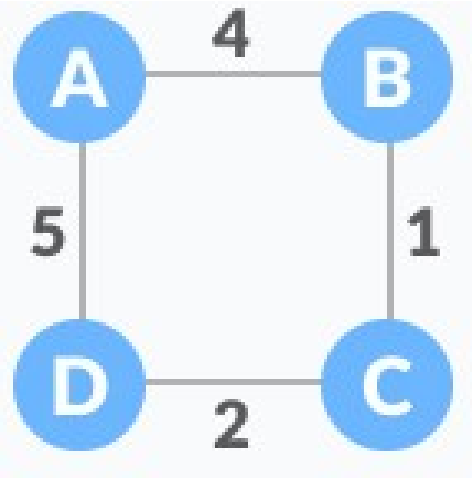
Mathematical Properties of Spanning Tree

- Spanning tree has ***n-1*** edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
- A complete graph can have maximum $n^{(n-2)}$ number of spanning trees.

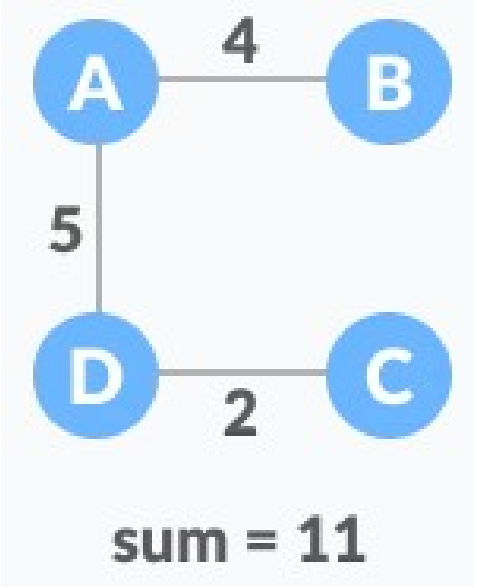
Minimum Spanning Tree

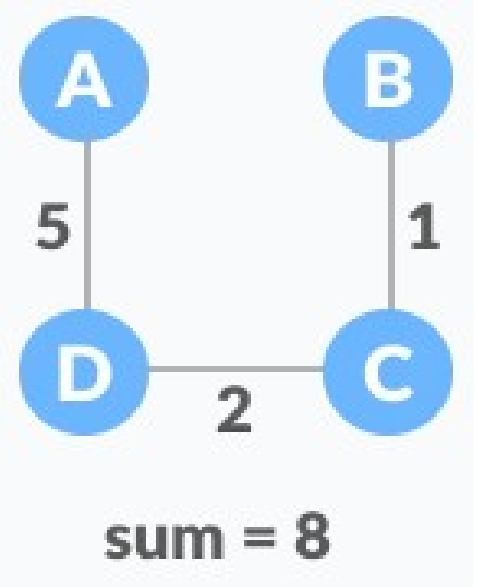
A minimum spanning tree is a spanning tree in which the sum of the weight of the edges is as minimum as possible.

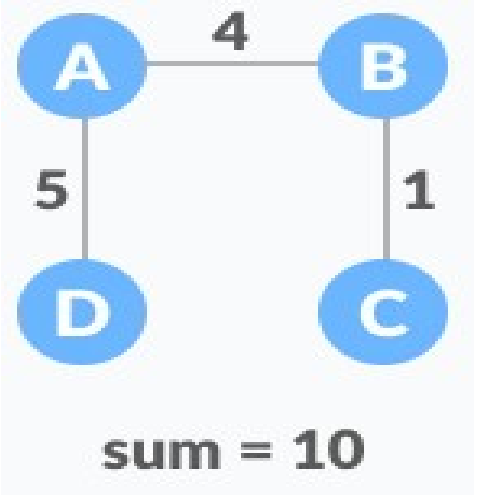
Example of a Minimum Spanning Tree

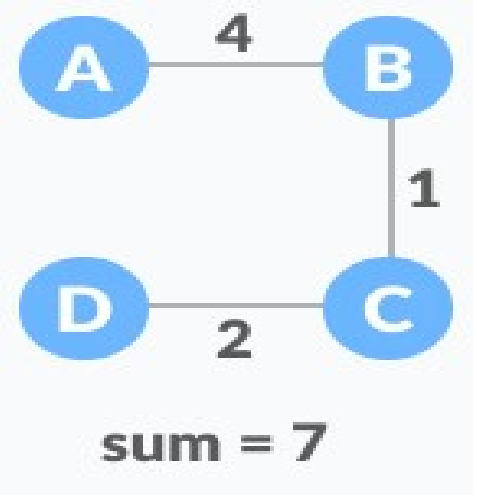


Initial Graph





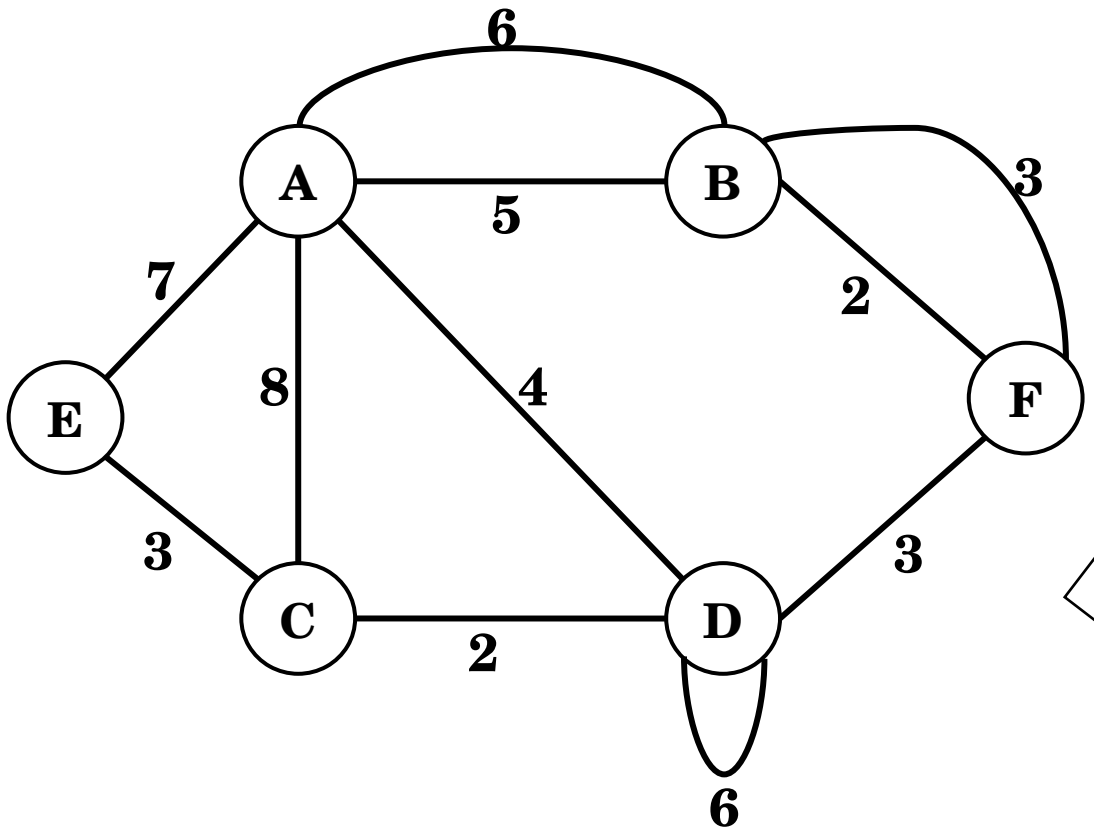




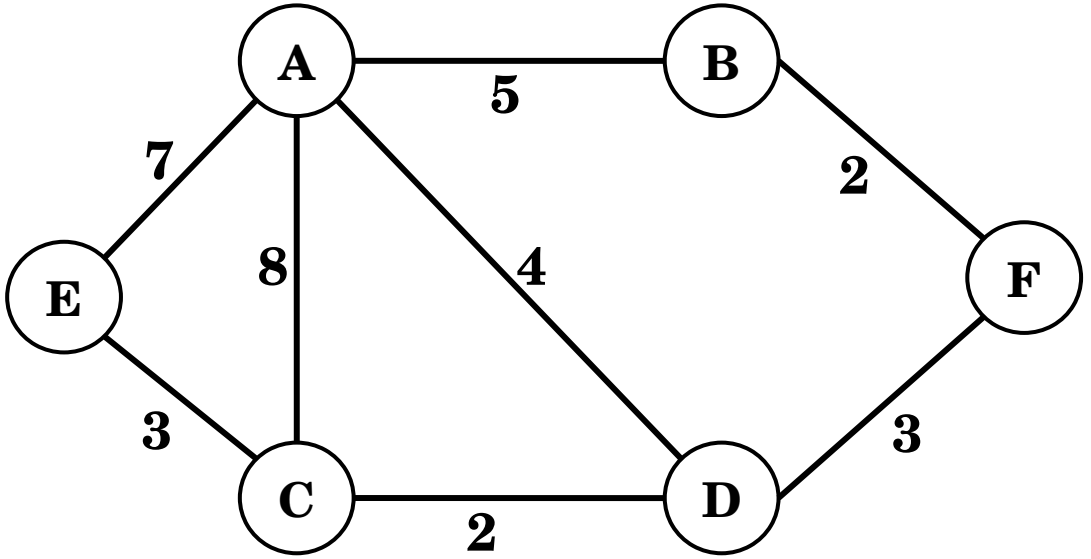
Kruskal's Algorithm

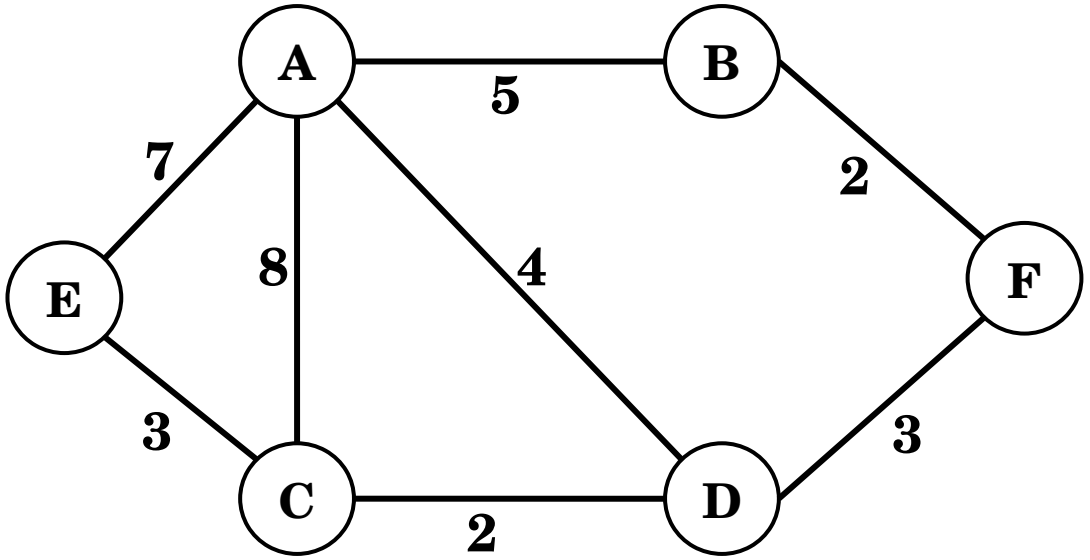
Step's of Kruskal's Algorithm

1. Remove loops and multiple edges (if any) from the given graph. In case of removing multiple edge, remove the edges with higher weights. Leave the edge having minimum weight among multiple edges.
2. Sort the edges in increasing order according to their weight.
3. From the ***sorted edge list,*** insert edges one by one into the solution graph. If any edge creates a cycle in the solution graph, then skip that edge.



Step 1





Step 2:

BF □ 2

CD □ 2

DF □ 3

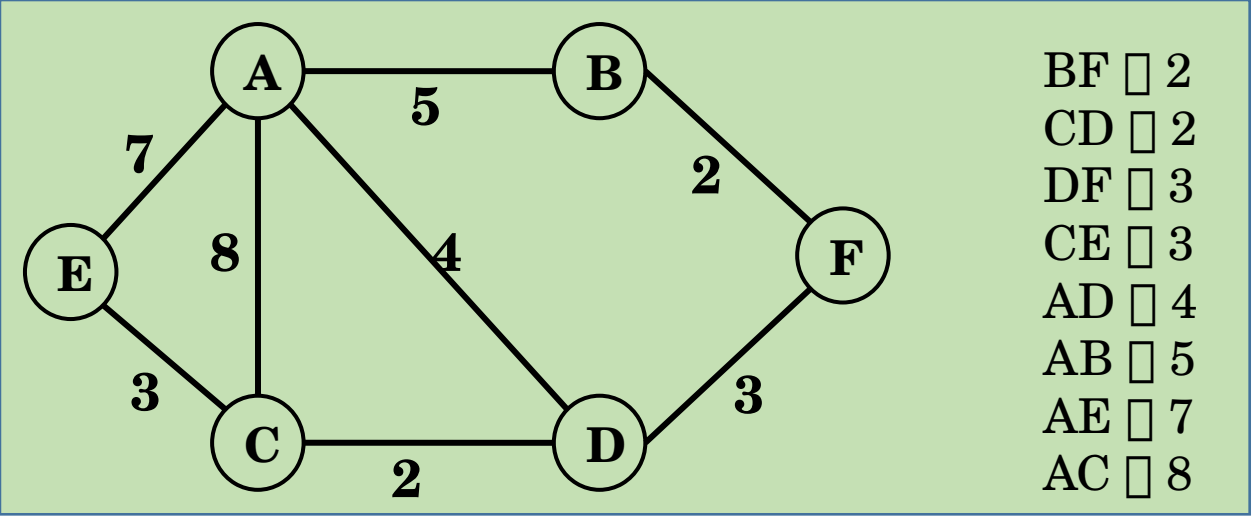
CE □ 3

AD □ 4

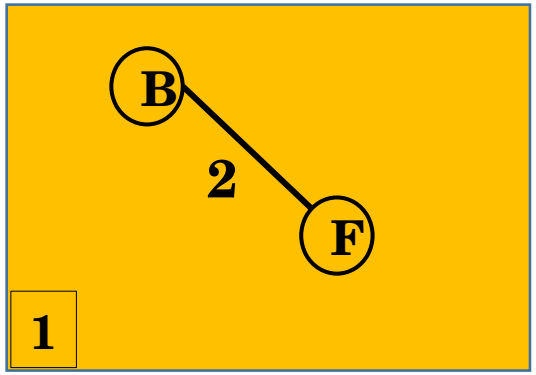
AB □ 5

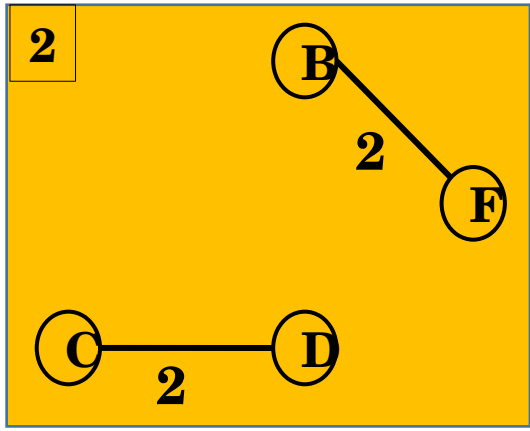
AE □ 7

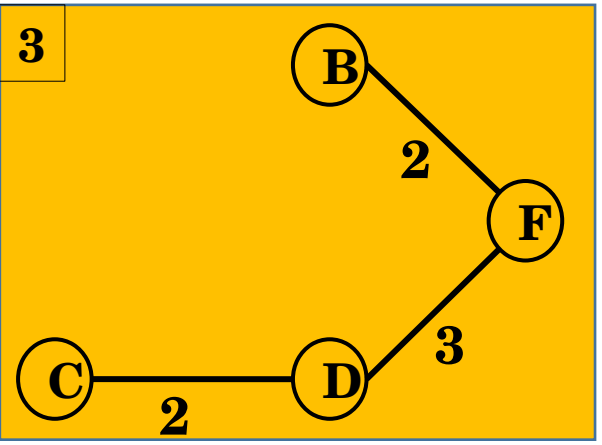
AC □ 8

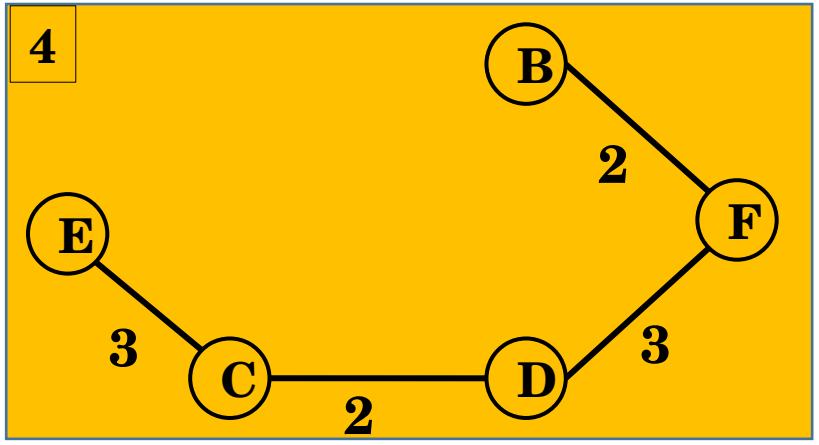


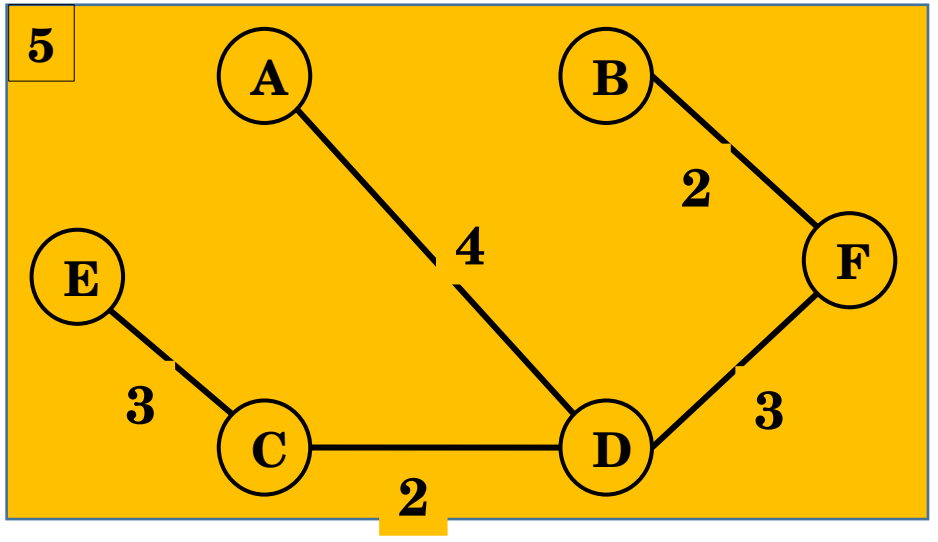
BF □ 2
CD □ 2
DF □ 3
CE □ 3
AD □ 4
AB □ 5
AE □ 7
AC □ 8



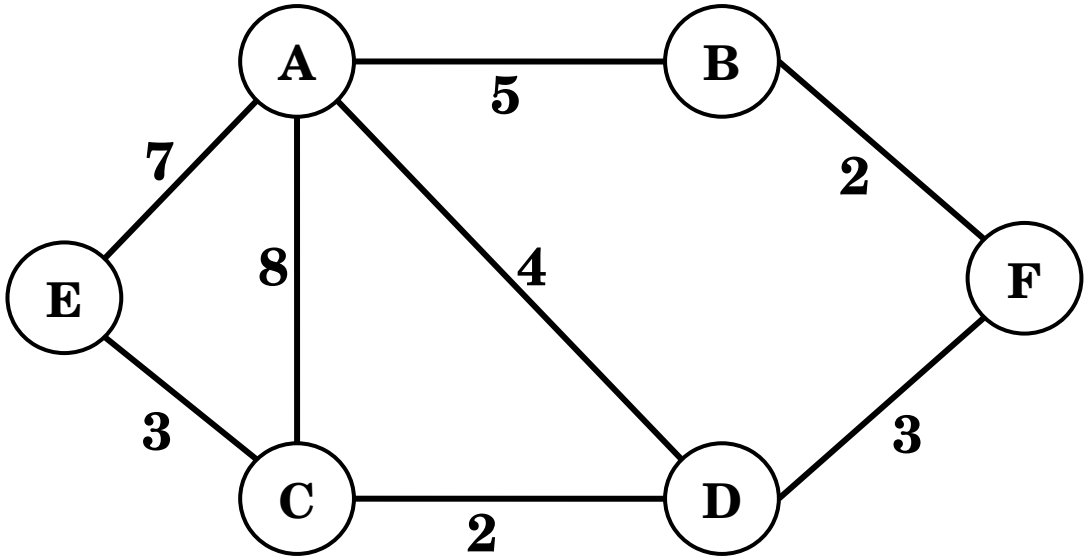








Prim’s Algorithm


A
B
E
C
D
F
5
7
2
8
4
3
3
2

Shortest Path Algorithms

Dijkstra's Shortest Path Algorithm

What is Dijkstra’s Algorithm

- Dijkstra's algorithm is a Single Source Shortest Path (SSSP) algorithm for graphs with non-negative edge weights.
- Depending on how the algorithm is implemented and what data structures are used the time complexity is typically ***O((V+E) * Log (V))*** which is competitive against other shortest path algorithms.

Prerequisite of Dijkstra’s Algorithm

- One constraint for Dijkstra's algorithm is that the graph must only contain ***non-negative edge weights***. This constraint is imposed to ensure that once a node has been visited its optimal distance cannot be improved.
- This is property is especially important because it enables Dijkstra's algorithm to act in a greedy manner by always selecting the next most promising node.

Purpose and Use Cases

- With Dijkstra's Algorithm, you can find the shortest path between nodes in a graph. Particularly, you can find the shortest path from a node (called the "source node") to all other nodes in the graph, producing a shortest-path tree.
- This algorithm is used in GPS devices to find the shortest path between the current location and the destination. It has broad applications in industry, specially in domains that require modeling networks.
- Ever wondered how does Google Maps finds the shortest and fastest route between two places? Well, the answer is Dijkstra's Algorithm.

Purpose and Use Cases

- Dijkstra's Algorithm is a Graph algorithm that finds the shortest path from a source vertex to all other vertices in the Graph (single source shortest path).
- It is a type of Greedy Algorithm that only works on Weighted Graphs having positive weights. The time complexity of Dijkstra's Algorithm is $O(V^2)$ with the help of the adjacency matrix representation of the graph.
- This time complexity can be reduced to $O((V + E) \log V)$ with the help of an adjacency list representation of the graph, where V is the number of vertices and E is the number of edges in the graph.

Fundamentals of Dijkstra's Algorithm

- Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
- The process continues until all the nodes in the graph have been added to the path. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Understanding the Working of Dijkstra's Algorithm

- A graph and source vertex are requirements for Dijkstra's Algorithm. This Algorithm is established on Greedy Approach and thus finds the locally optimal choice (local minima in this case) at each step of the Algorithm.
- Each Vertex in this Algorithm will have two properties defined for it:
  - Visited Property
  - Path Property

Let us understand these properties in brief.

Understanding the Working of Dijkstra's Algorithm

Visited Property:

1. The 'visited' property signifies whether or not the node has been visited.
2. We are using this property so that we do not revisit any node.
3. A node is marked visited only when the shortest path has been found.

Path Property:

4. The 'path' property stores the value of the current minimum path to the node.
5. The current minimum path implies the shortest way we have reached this node till now.
6. This property is revised when any neighbor of the node is visited.
7. This property is significant because it will store the final answer for each node.

Understanding the Working of Dijkstra's Algorithm

- Initially, we mark all the vertices, or nodes, unvisited as they have yet to be visited. The path to all the nodes is also set to infinity apart from the source node. Moreover, the path to the source node is set to zero (0).
- We then select the source node and mark it as visited. After that, we access all the neighboring nodes of the source node and perform relaxation on every node. Relaxation is the process of lowering the cost of reaching a node with the help of another node.
- In the process of relaxation, the path of each node is revised to the minimum value amongst the node's current path, the sum of the path to the previous node, and the path from the previous node to the current node.

Understanding the Working of Dijkstra's Algorithm

Let us suppose that p[n] is the value of the current path for node n, p[m] is the value of the path up to the previously visited node m, and w is the weight of the edge between the current node and previously visited one (edge weight between n and m).

In the mathematical sense, relaxation can be exemplified as:

p[n] = minimum(p[n], p[m] + w)

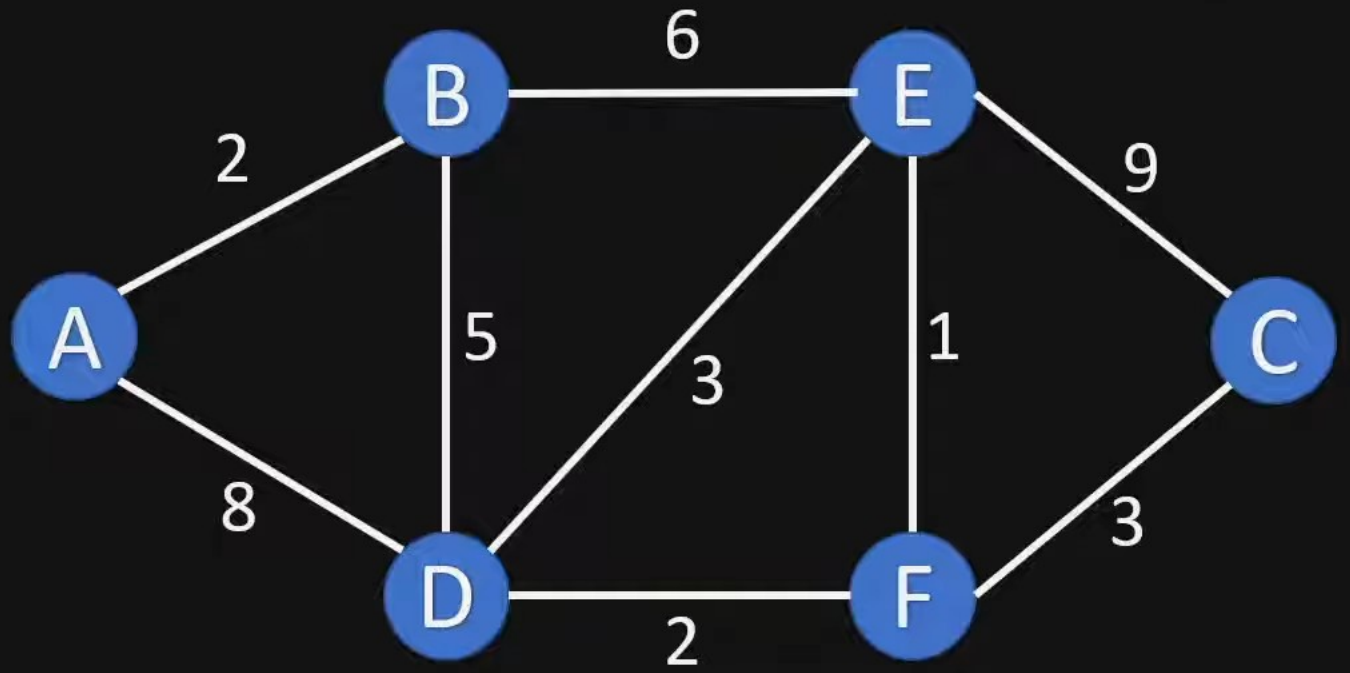
We then mark an unvisited node with the least path as visited in every subsequent step and update its neighbor's paths.

We repeat this procedure until all the nodes in the graph are marked visited.

Whenever we add a node to the visited set, the path to all its neighboring nodes also changes accordingly.

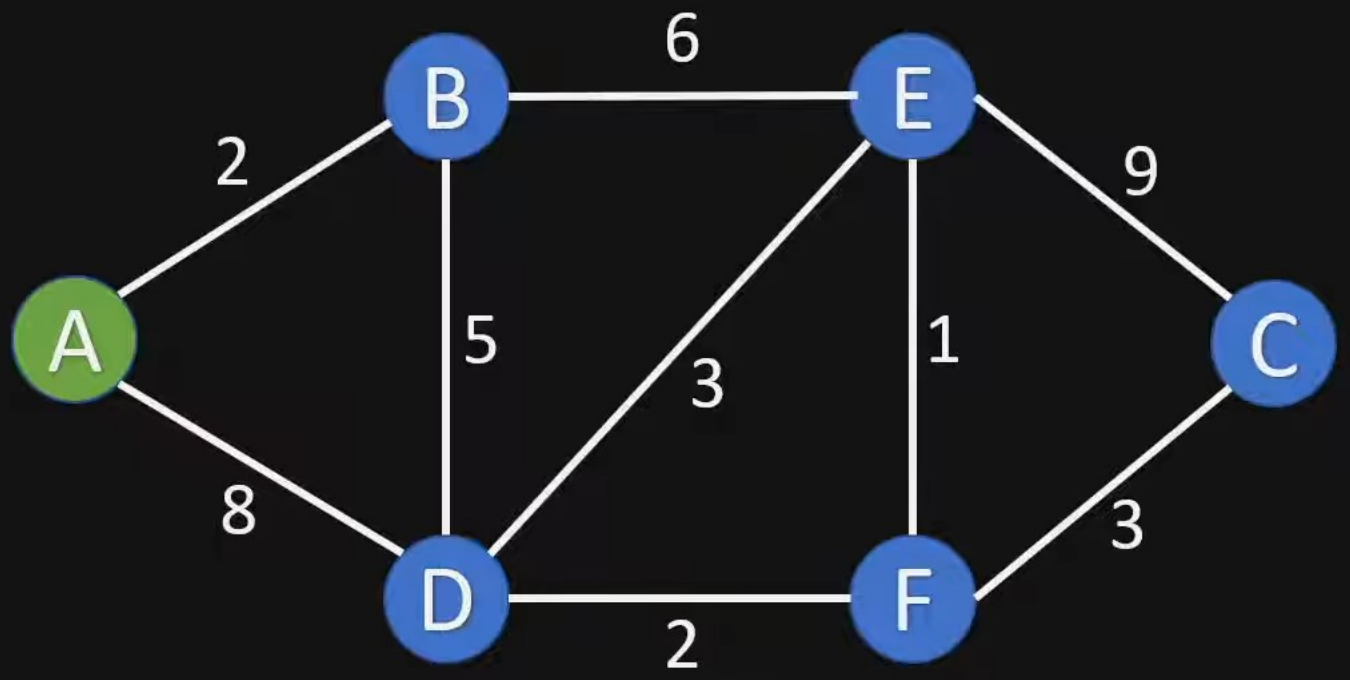
If any node is left unreachable (disconnected component), its path remains 'infinity'. In case the source itself is a separate component, then the path to all other nodes remains 'infinity'.

Dijkstra's Shortest Path Algorithm



- Shortest path from a fixed node to every other node
- e.g. Cities and routes between them

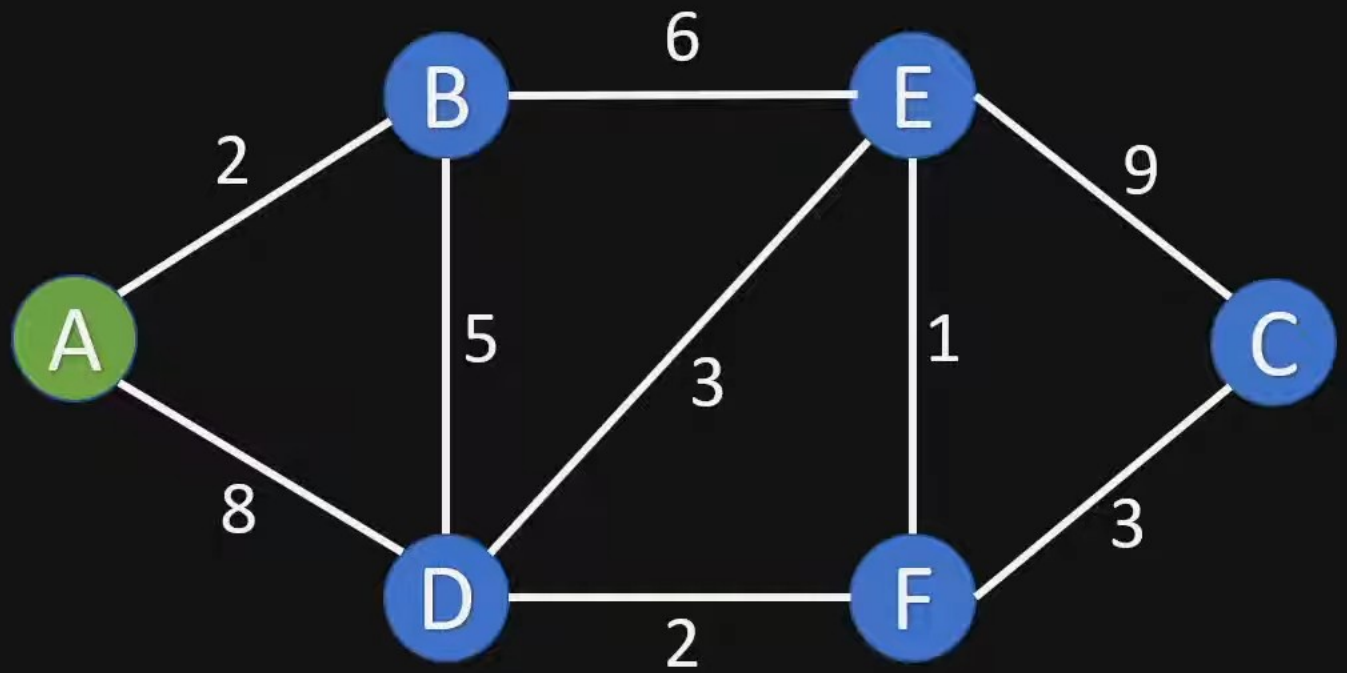
1. Mark all nodes as unvisited



Visited Nodes: []

Unvisited Nodes: [A, B, C, D, E, F]

2. Assign to all nodes a tentative distance value



Node	Shortest Distance	Previous Node
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	

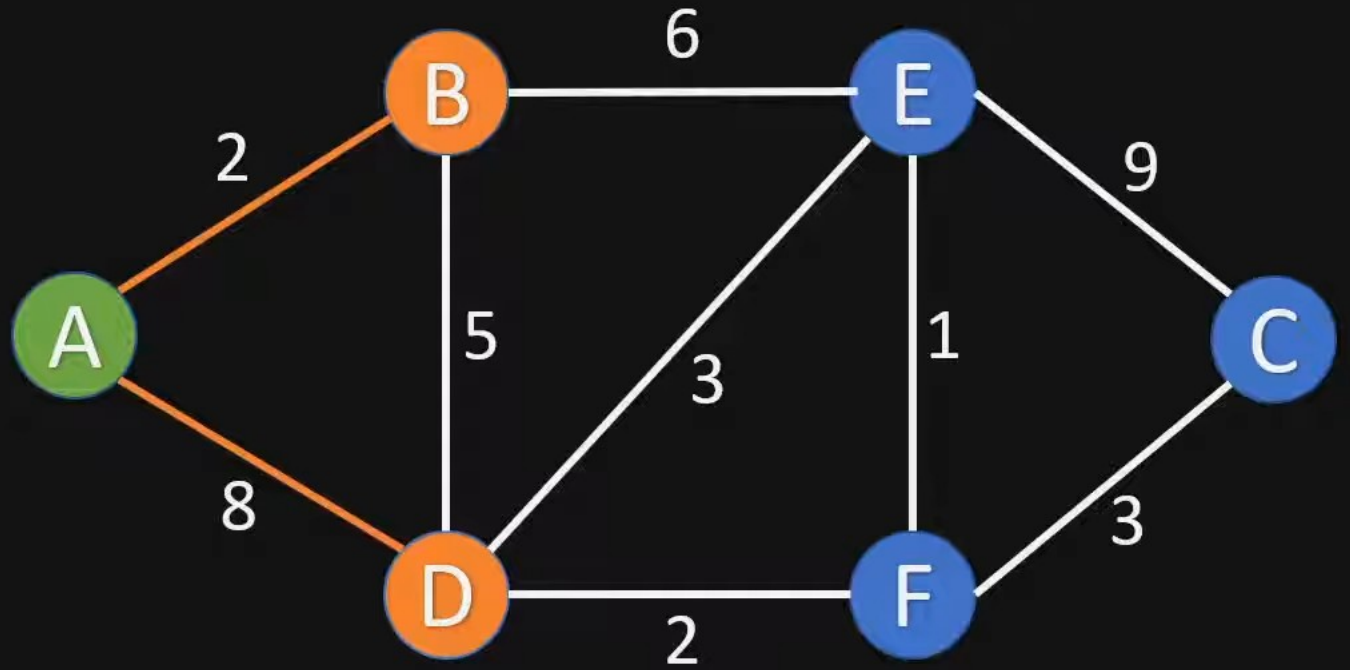
Visited Nodes: []

Unvisited Nodes: [A, B, C, D, E, F]

Dijkstra's Algorithm

3. For the current node calculate the distance to all unvisited neighbours

3.1. Update shortest distance, if new distance is shorter than old distance



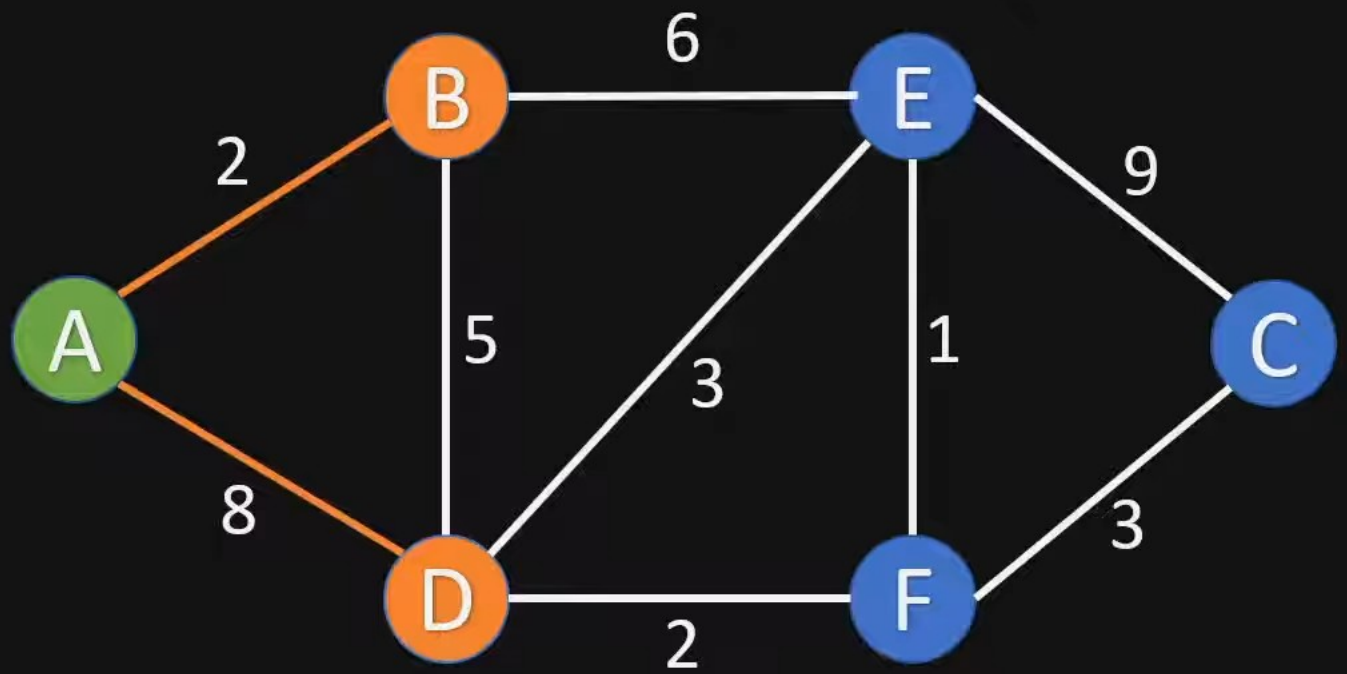
Node	Shortest Distance	Previous Node
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	

Visited Nodes: []

Unvisited Nodes: [A, B, C, D, E, F]

d[v] = d[u]+c(u,v)
where,
u is the current node
	if	(d[u]+c(u,v) < d[v]){
		d[v] = d[u]+c(u,v)
		}

Dijkstra's Algorithm



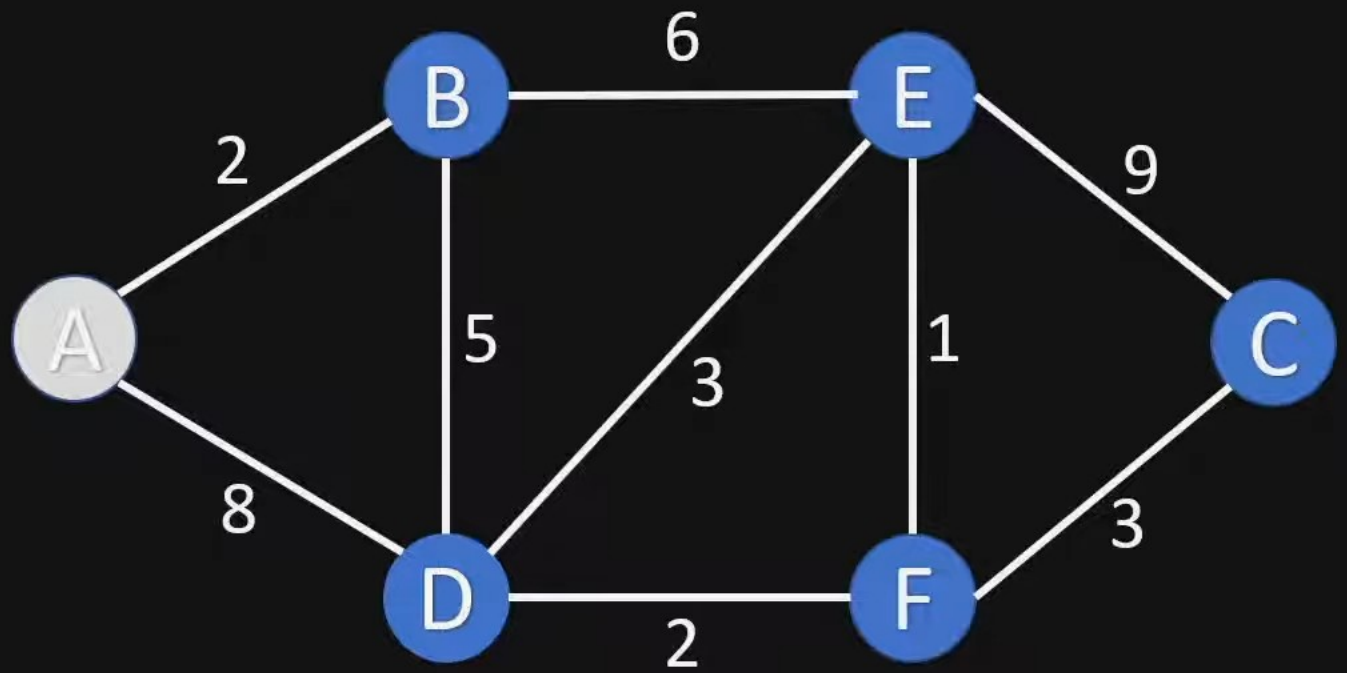
Visited Nodes: []

Unvisited Nodes: [A, B, C, D, E, F]

Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

Dijkstra's Algorithm

4. Mark current node as visited

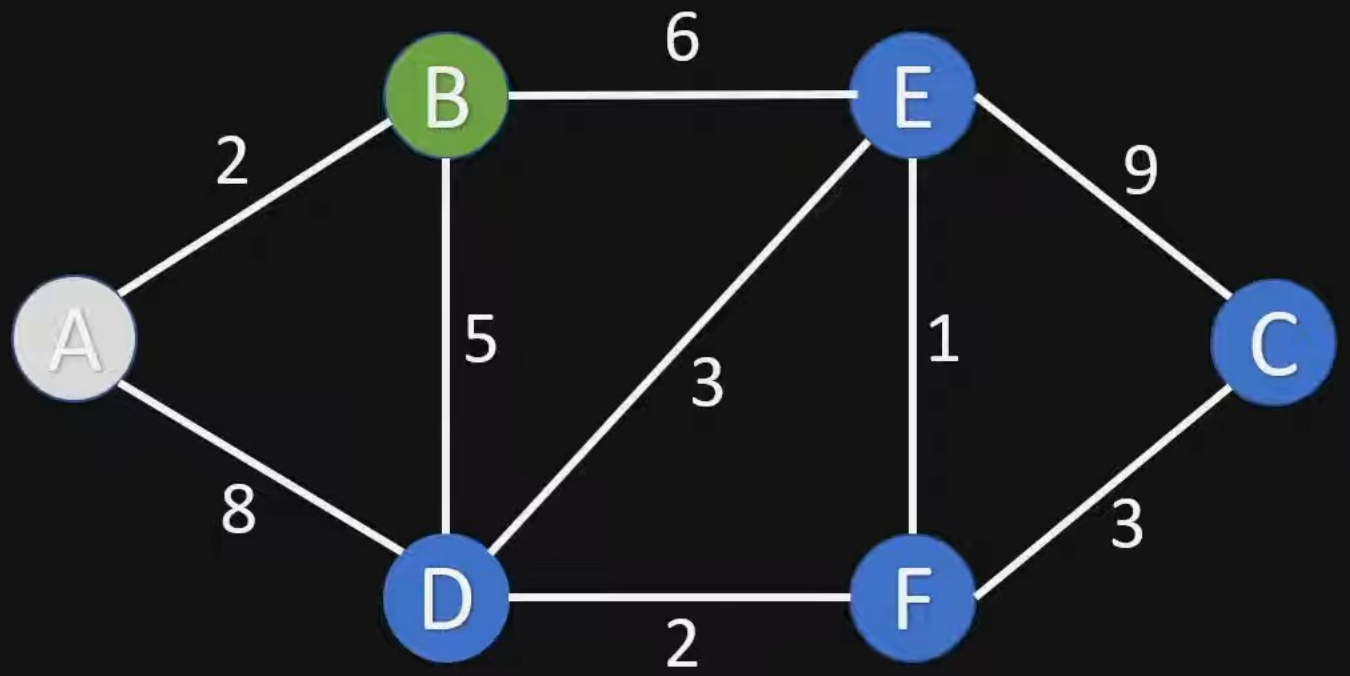


Node	Shortest Distance	Previous Node
A	0	
B	2	A
C	∞	
D	8	A
E	∞	
F	∞	

Visited Nodes: [A]

Unvisited Nodes: [B, C, D, E, F]

5. Choose new current node from unvisited nodes with minimal distance



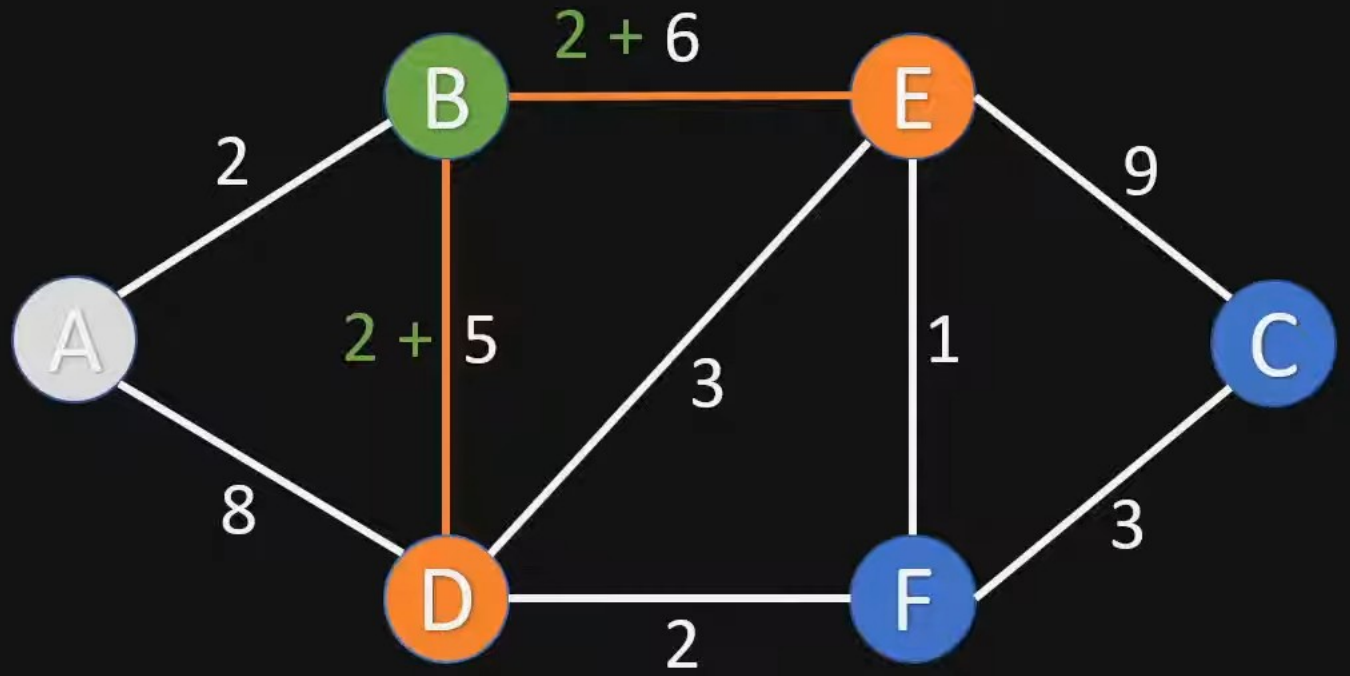
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

Visited Nodes: [A]

Unvisited Nodes: [B, C, D, E, F]

3. For the current node calculate the distance to all unvisited neighbours

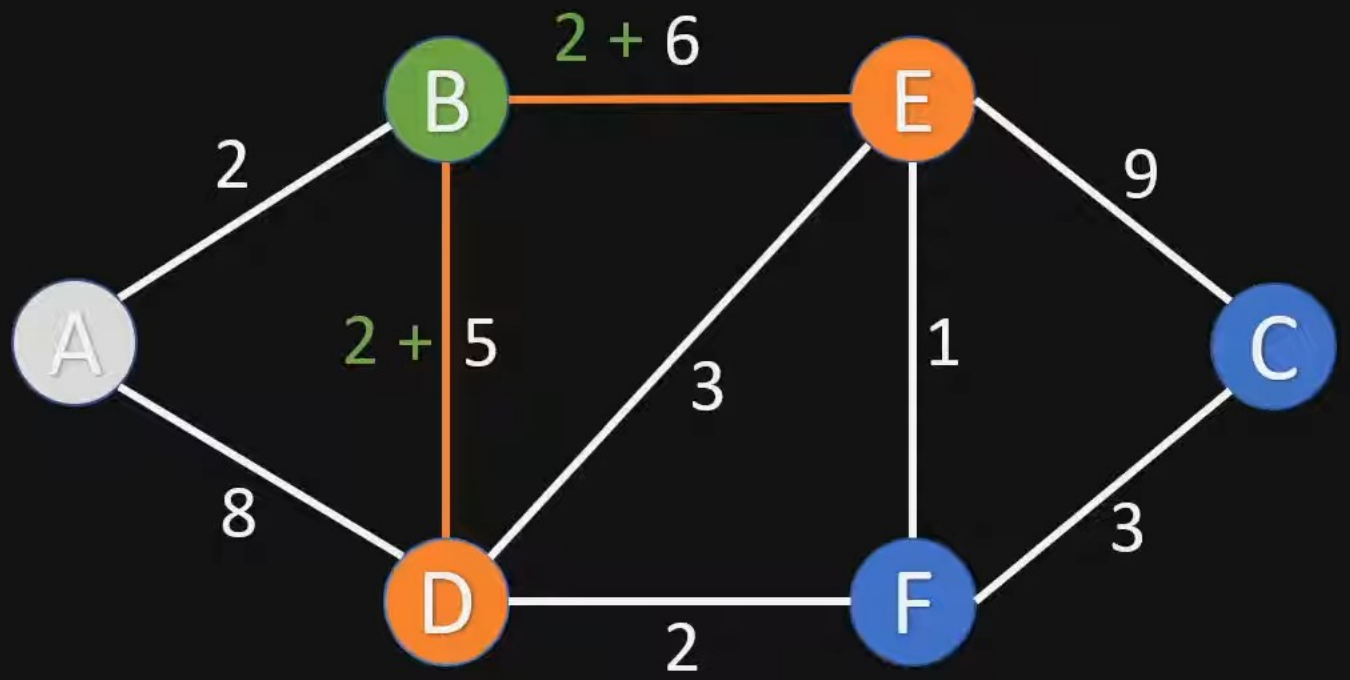
3.1. Update shortest distance, if new distance is shorter than old distance



Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**8**	A
E	∞	
F	∞	

Visited Nodes: [A]

Unvisited Nodes: [B, C, D, E, F]



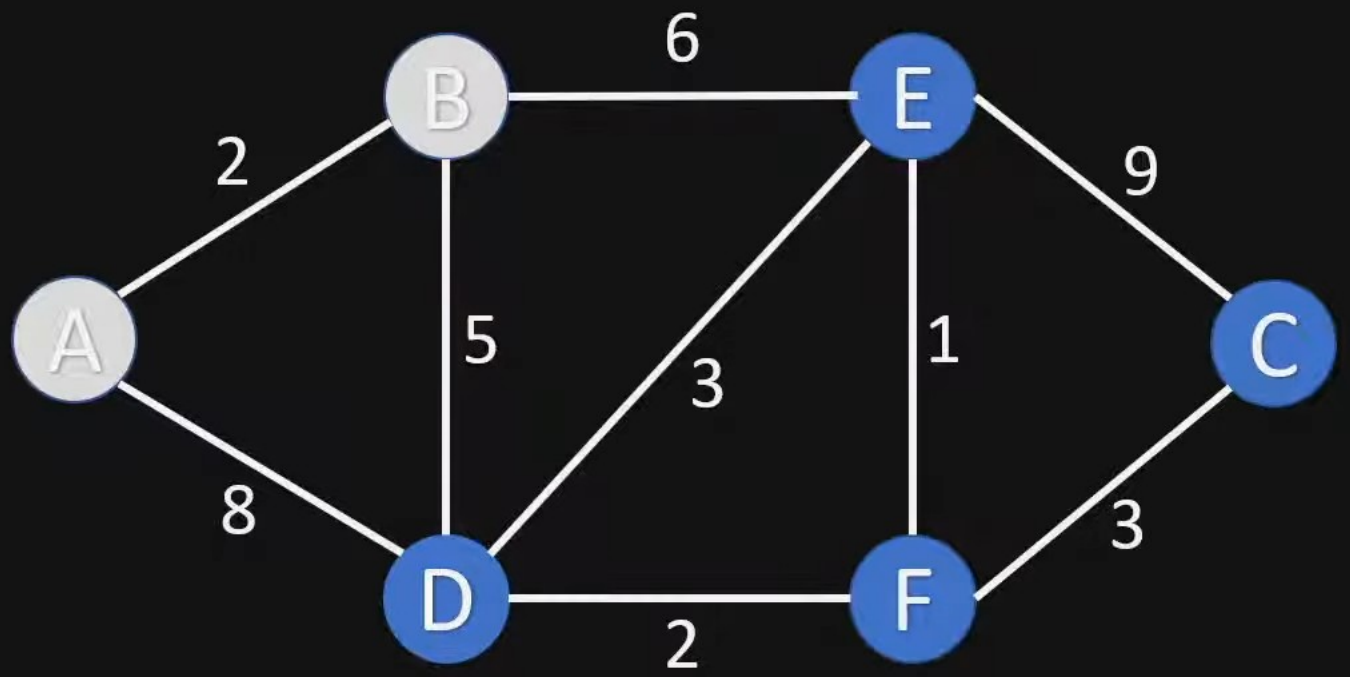
Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**∞**	
D	**7**	B
E	**8**	B
F	**∞**	

Visited Nodes: [A]

Unvisited Nodes: [B, C, D, E, F]

Dijkstra's Algorithm

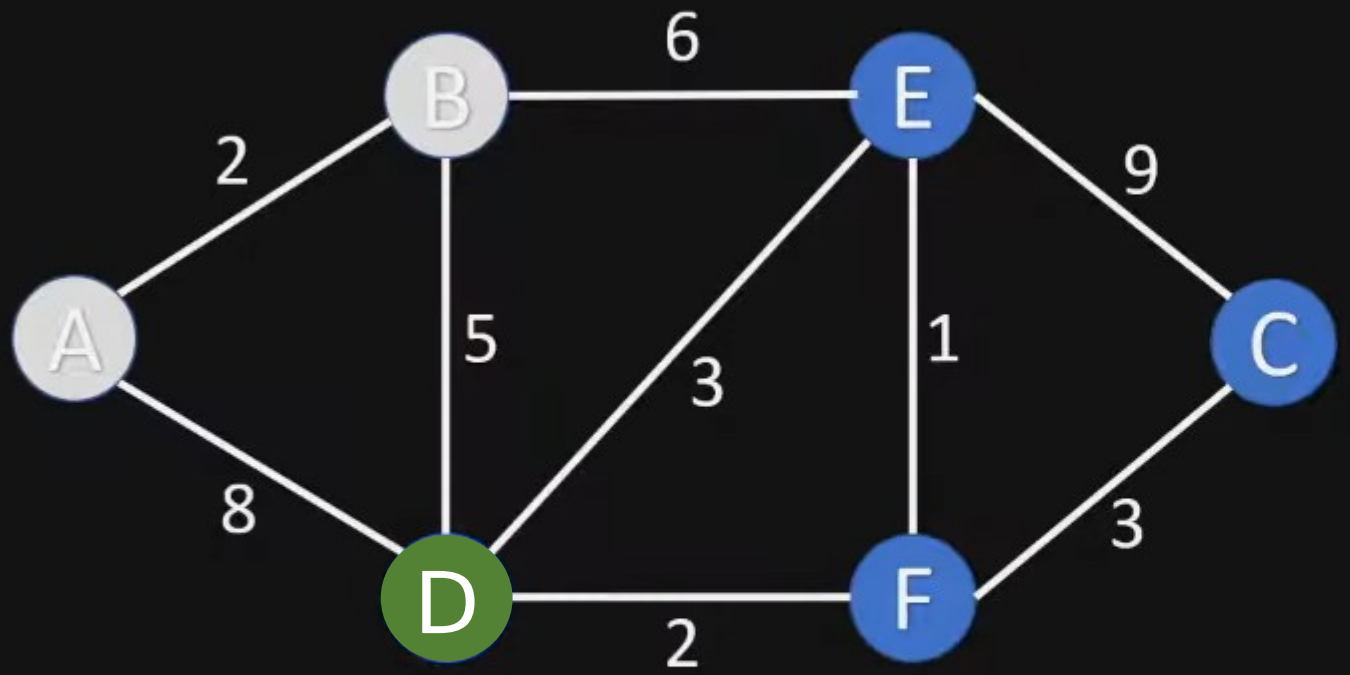
4. Mark current node as visited



Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**7**	B
E	**8**	B
F	∞	

Visited Nodes: [A, B] Unvisited Nodes: [C, D, E, F]

. Choose New Current Node from unvisited nodes with minimal distance



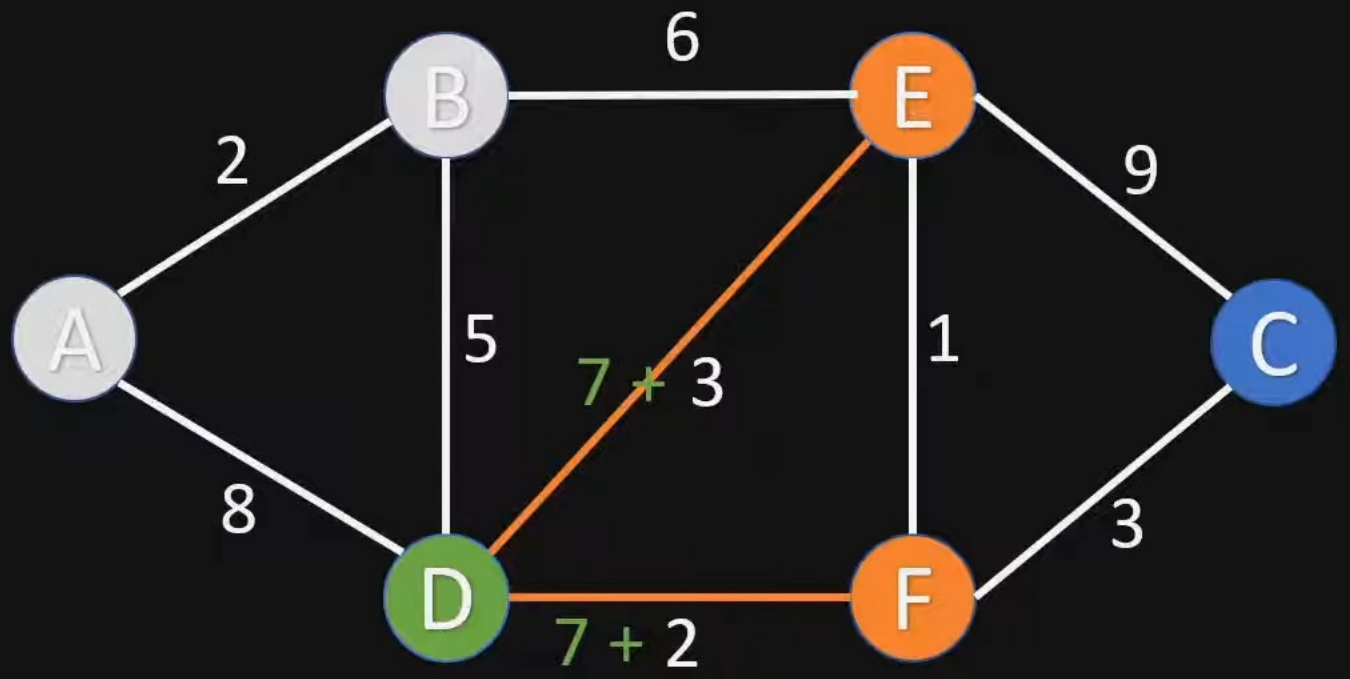
Visited Nodes: [A, B] Unvisited Nodes: [C, D, E, F]

Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	∞	
D	**7**	B
E	**8**	B
F	∞	

Dijkstra's Algorithm

3. For the current node calculate the distance to all unvisited neighbours

3.1. Update shortest distance, if new distance is shorter than old distance

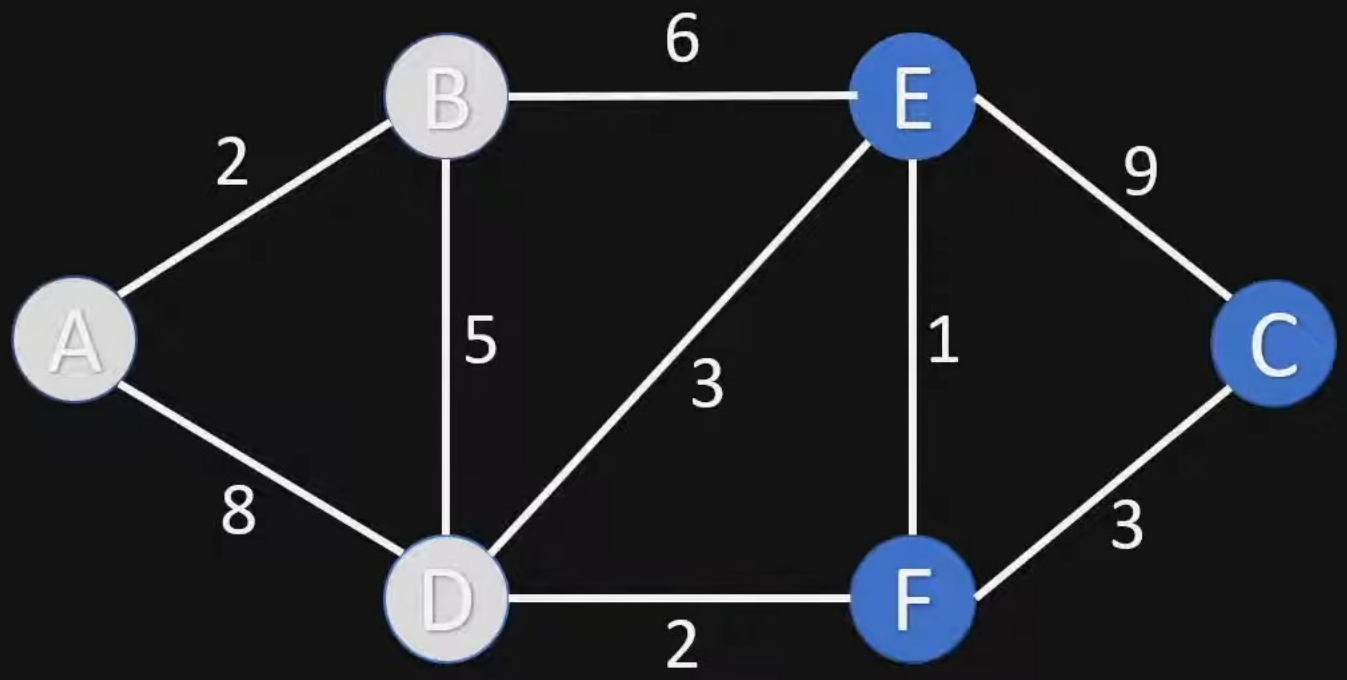


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**∞**	
D	**7**	B
E	**8**	B
F	**9**	D

Visited Nodes: [A, B] Unvisited Nodes: [C, D, E, F]

Dijkstra's Algorithm

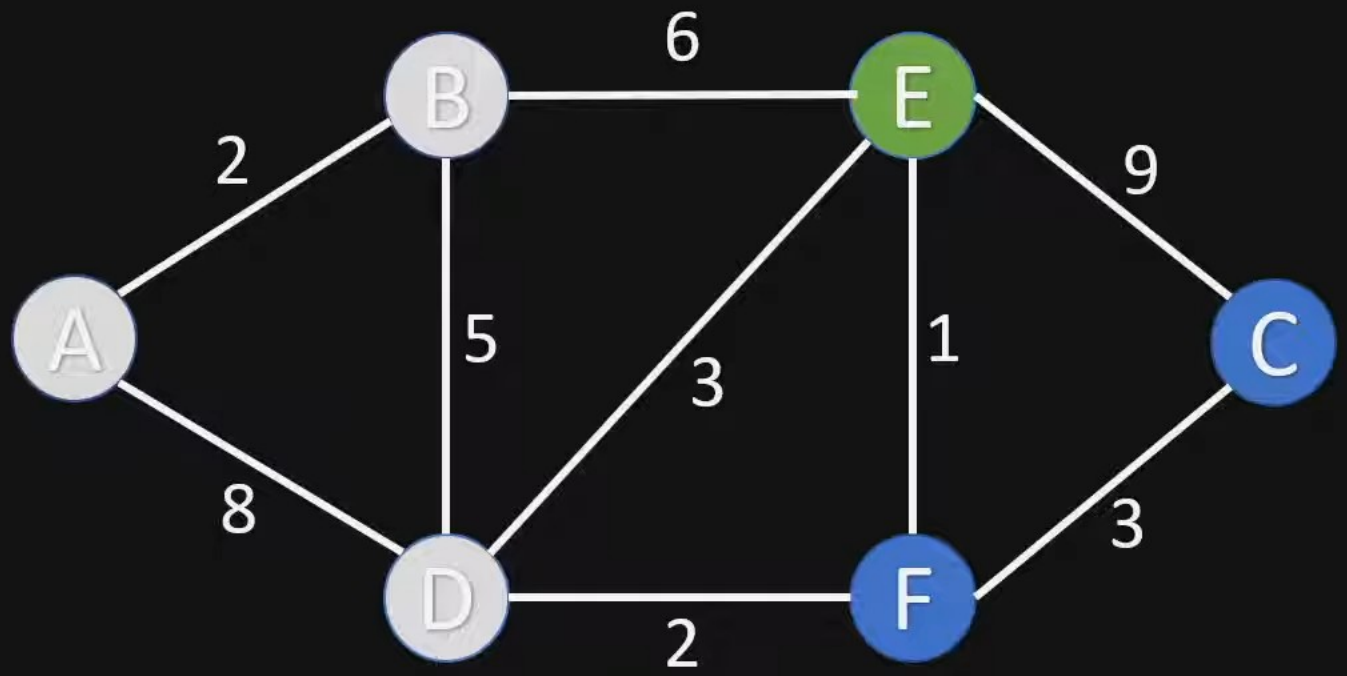
4. Mark current node as visited



Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**∞**	
D	**7**	B
E	**8**	B
F	**9**	D

Visited Nodes: [A, B, D] Unvisited Nodes: [C, D, E, F]

5. Choose new current node from unvisited nodes with minimal distance

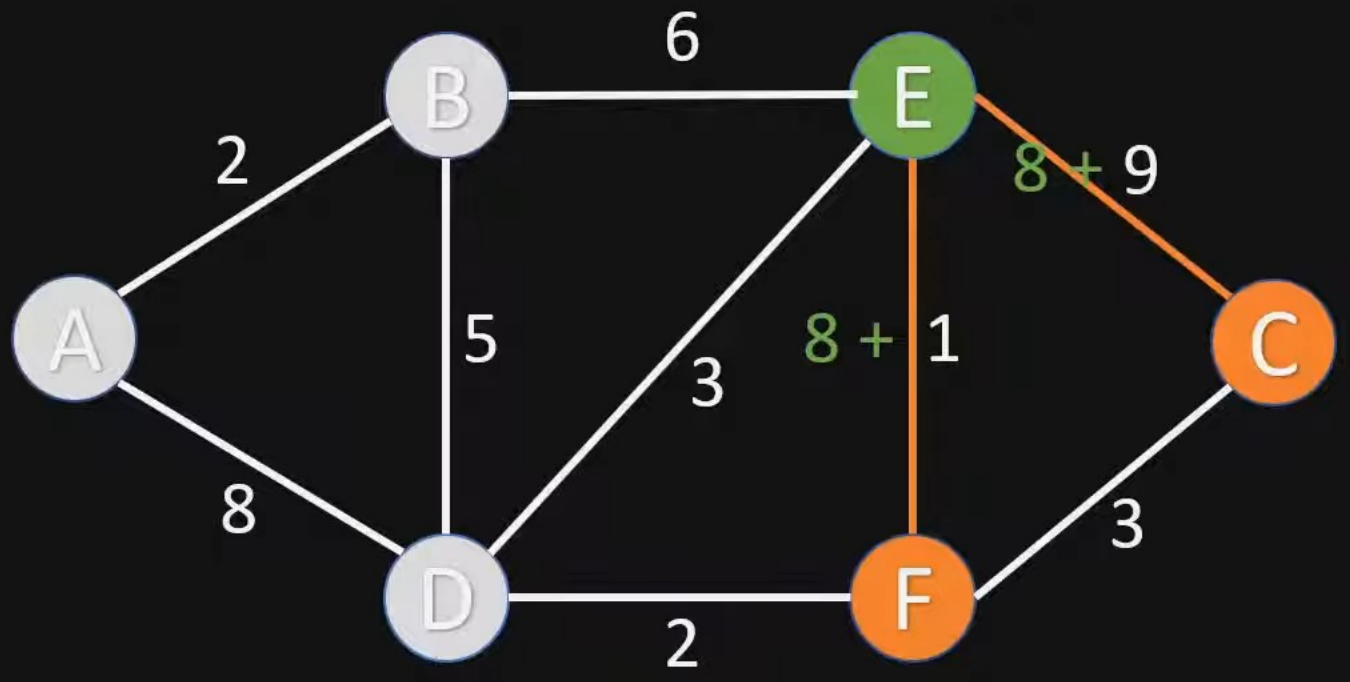


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**∞**	
D	**7**	B
E	**8**	B
F	**9**	D

Visited Nodes: [A, B, D] Unvisited Nodes: [C, E, F]

3. For the current node calculate the distance to all unvisited neighbours

3.1. Update shortest distance, if new distance is shorter than old distance

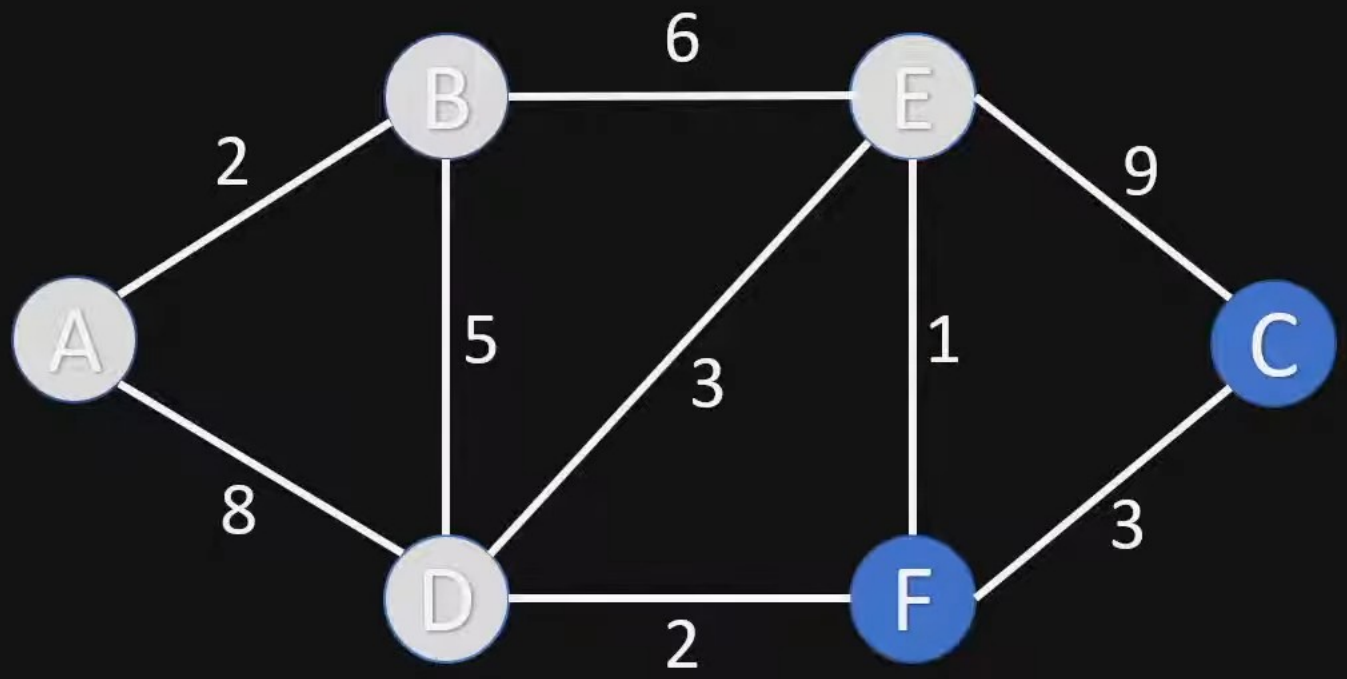


Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**17**	E
D	**7**	B
E	**8**	B
F	**9**	D

Visited Nodes: [A, B, D] Unvisited Nodes: [C, E, F]

Dijkstra's Algorithm

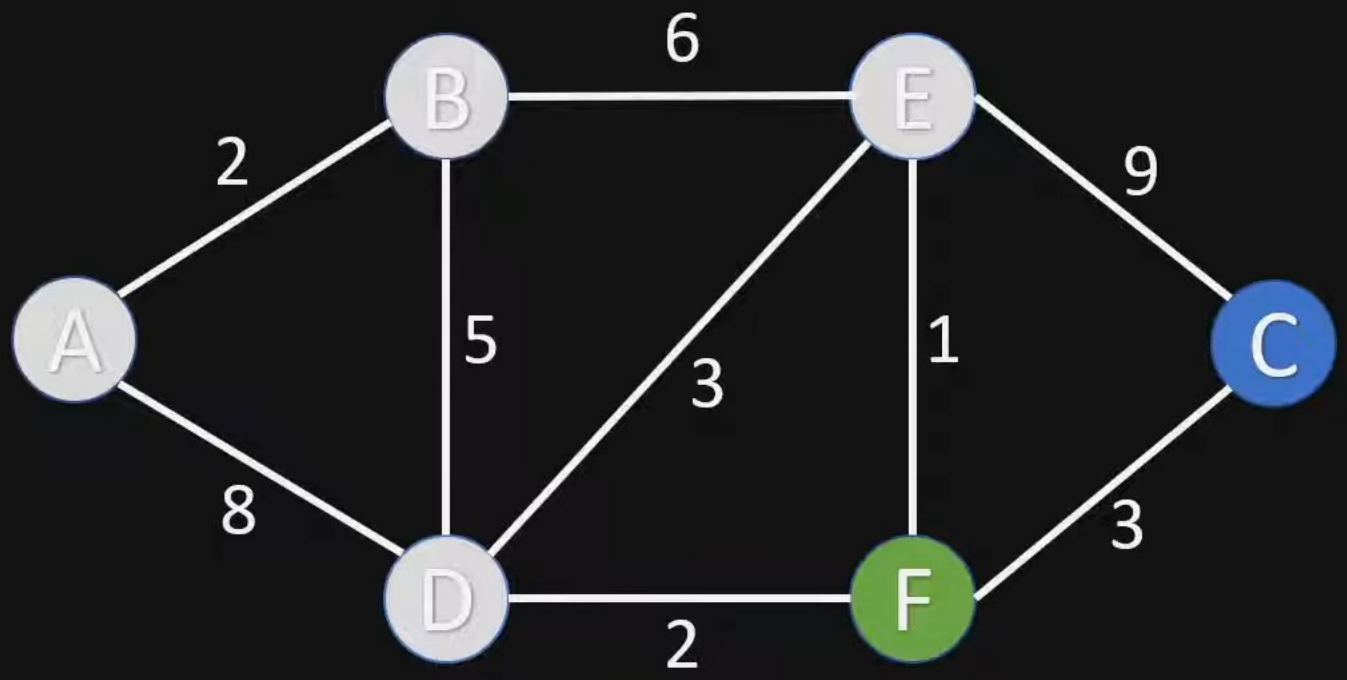
4. Mark current node as visited



Visited Nodes: [A, B, D, E] Unvisited Nodes: [C, F]

Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**17**	E
D	**7**	B
E	**8**	B
F	**9**	D

5. Choose new current node from unvisited nodes with minimal distance

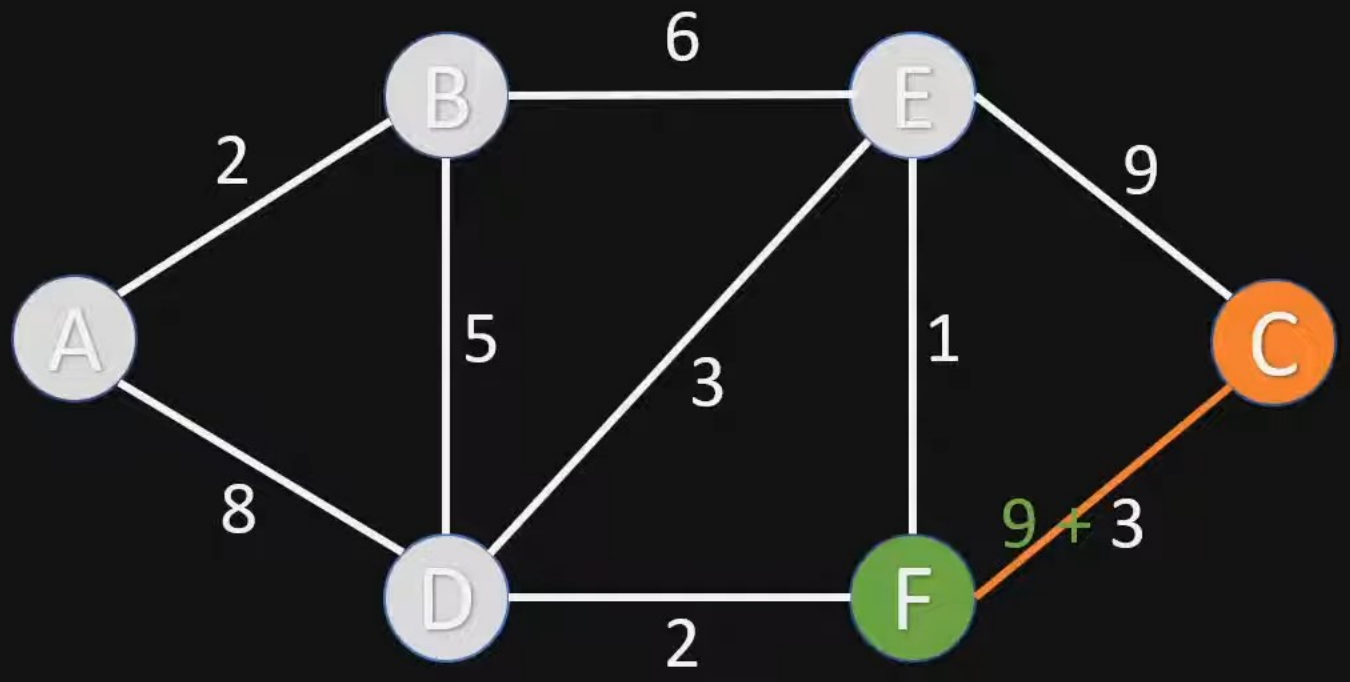


Visited Nodes: [A, B, D, E] Unvisited Nodes: [C, F]

Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**17**	E
D	**7**	B
E	**8**	B
F	**9**	D

3. For the current node calculate the distance to all unvisited neighbours

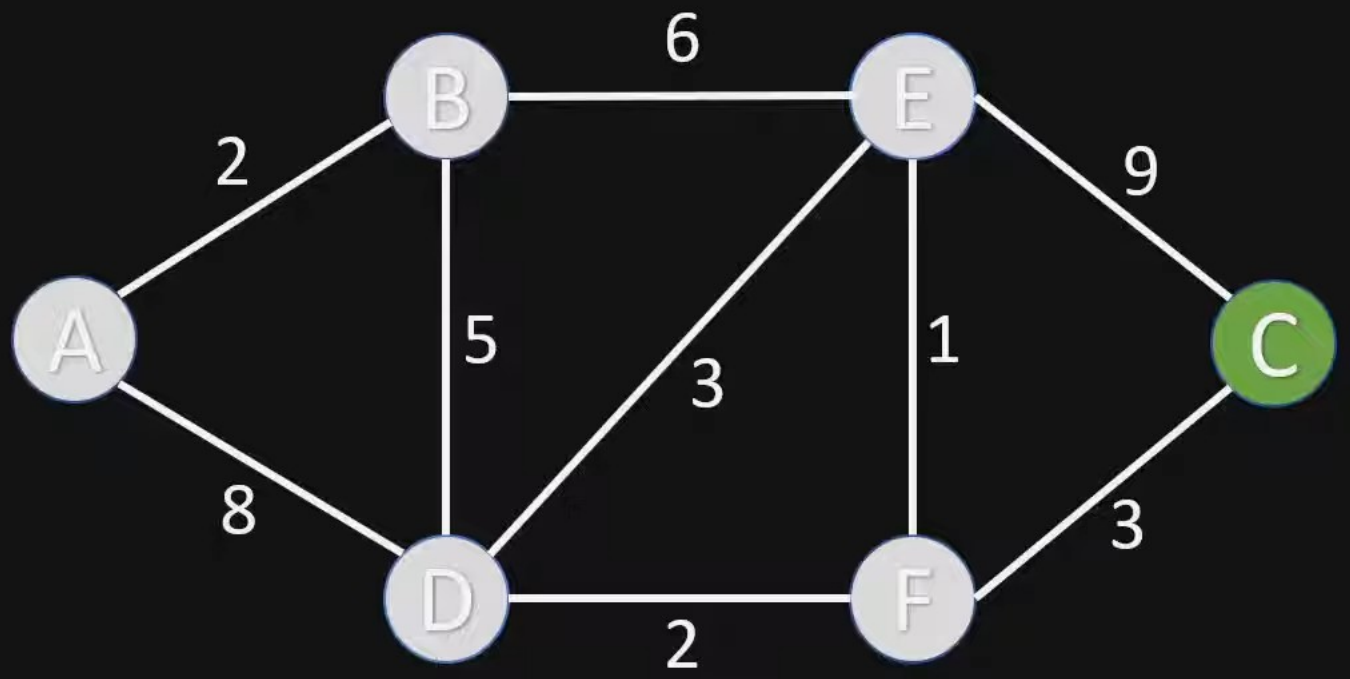
3.1. Update shortest distance, if new distance is shorter than old distance



Visited Nodes: [A, B, D, E] Unvisited Nodes: [C, F]

Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

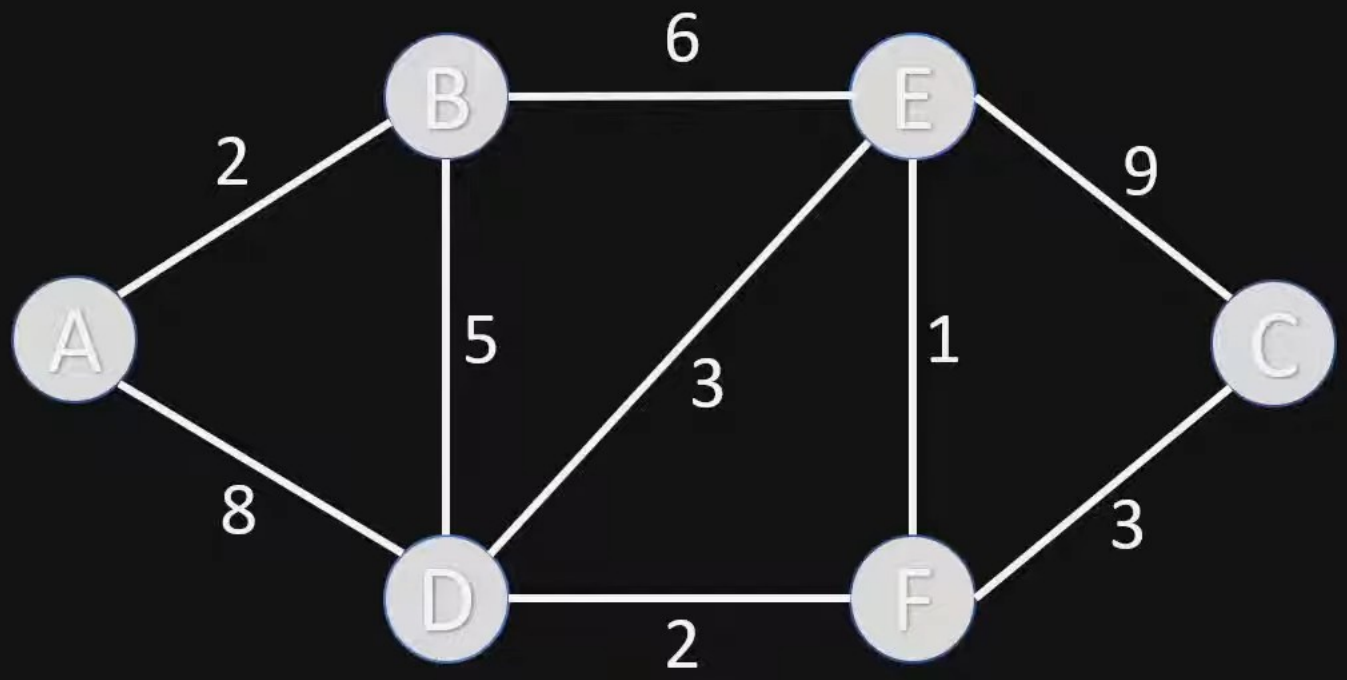
5. Choose new current node from unvisited nodes with minimal distance



Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Visited Nodes: [A, B, D, E, F] Unvisited Nodes: [C]

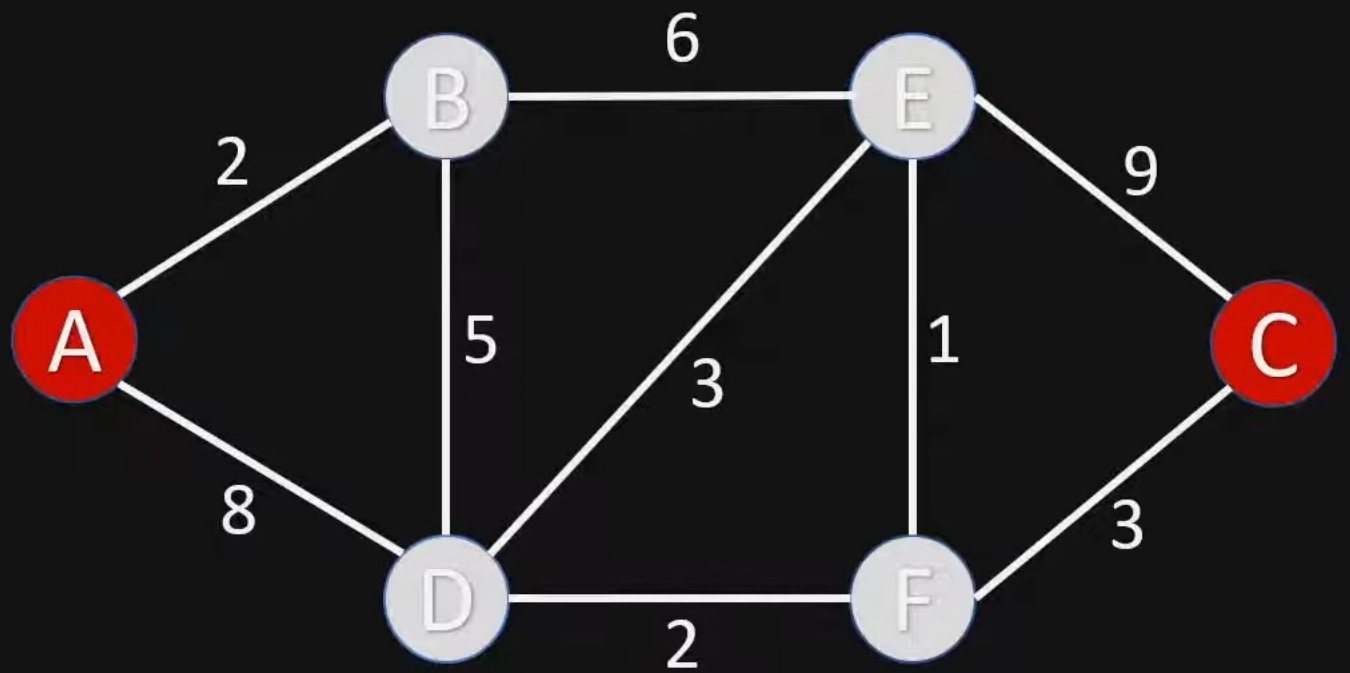
4. Mark current node as visited



Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Visited Nodes: [A, B, D, E, F, C] Unvisited Nodes: []

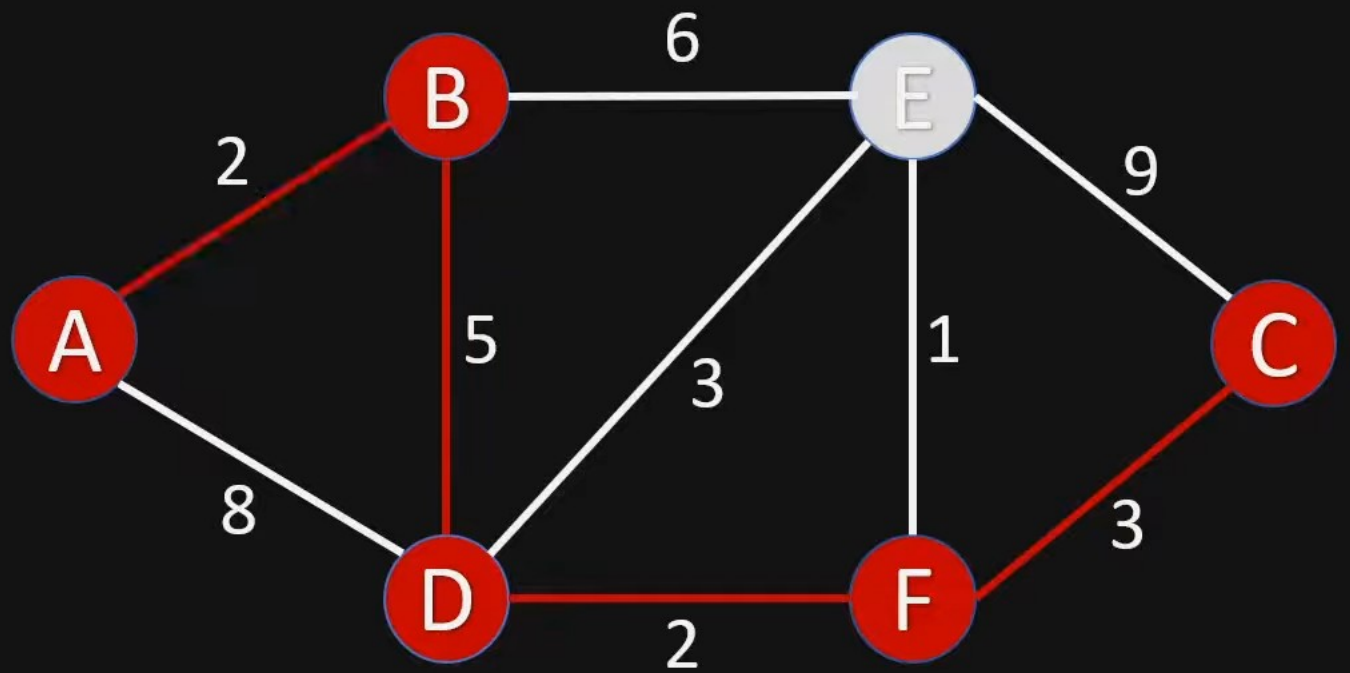
Get shortest path from A to C



Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

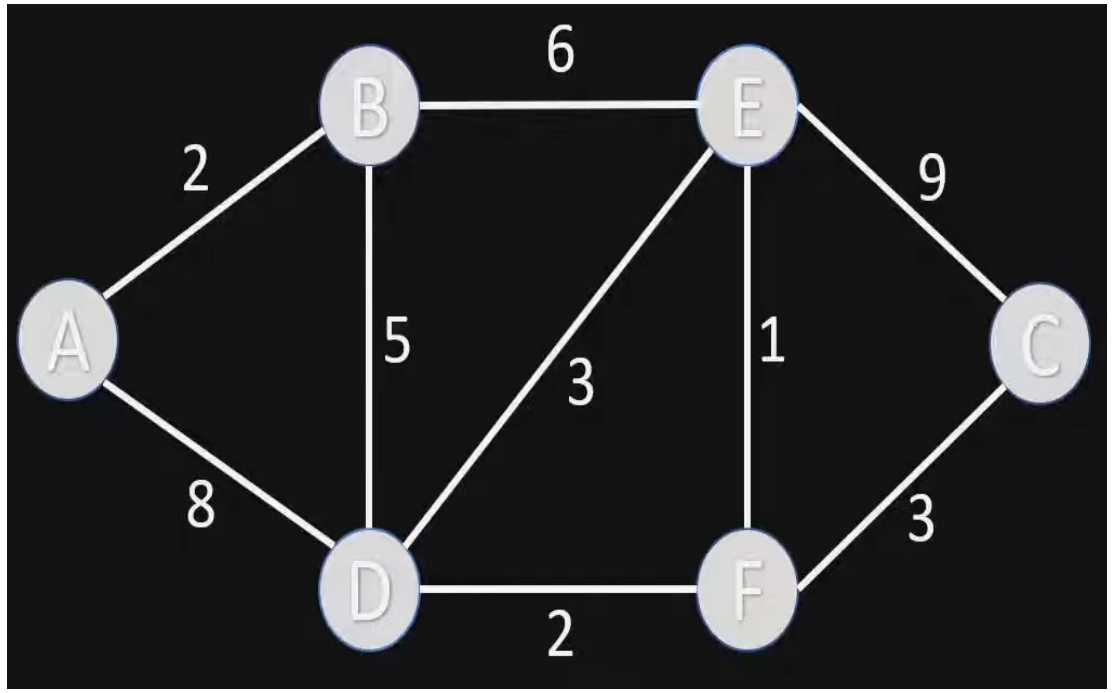
Dijkstra's Algorithm

Get shortest path from A to C



Node	Shortest Distance	Previous Node
A	0	
B	**2**	A
C	**12**	F
D	**7**	B
E	**8**	B
F	**9**	D

Dijkstra's Algorithm



Node	Shortest Distance	Previous Node
A	0	
B	2	A
C	12	F
D	7	B
E	8	B
F	9	D

	Shortest Distance	Path
A to B	2	A □ B
A to C	12	A □ B □ D □ F □ C
A to D	7	A □ B □ D
A to E	8	A □ B □ E
A to F	9	A □ B □ D □ F

Dijkstra Algorithm Time and Space Complexity

Complexity analysis for Dijkstra's algorithm with adjacency matrix representation of graph. Time complexity of Dijkstra's algorithm is $O(V^2)$ where V is the number of vertices in the graph.

It can be explained as below:

1. First thing we need to do is find the unvisited vertex with the smallest path. For that we require O(V) time as we need check all the vertices.
2. Now for each vertex selected as above, we need to relax its neighbors which means to update each neighbors path to the smaller value between its current path or to the newly found. The time required to relax one neighbor comes out to be of order of O(1) (constant time).
3. For each vertex we need to relax all of its neighbors, and a vertex can have at most (V-1) neighbors, so the time required to update all neighbors of a vertex comes out to be [O(V) * O(1)] = O(V)

Dijkstra Algorithm Time and Space Complexity

So now following the above conditions, we get:

Time for visiting all vertices = O(V)

Time required for processing one vertex = O(V)

Time required for visiting and processing all the vertices = O(V) * O(V) = $O(V^2)$

So the time complexity of Dijkstra's algorithm using adjacency matrix representation comes out to be ***O(V²).***

- Space complexity of adjacency matrix representation of graph in the algorithm is also $O(V^2)$ as a V*V matrix is required to store the representation of the graph. An additional array of V length will also be used by the algorithm to maintain the states of each vertex but the total space complexity will remain $O(V^2)$.

Dijkstra Algorithm Time and Space Complexity

- The time complexity of Dijkstra's algorithm can be reduced to O((V+E) log V) using adjacency list representation of the graph and a min-heap to store the unvisited vertices, where E is the number of edges in the graph and V is the number of vertices in the graph.
- With this implementation, the time to visit each vertex becomes O(V+E) and the time required to process all the neighbors of a vertex becomes O(log V).

Time for visiting all vertices =O(V+E)

Time required for processing one vertex=O(log V)

Time required for visiting and processing all the vertices

= O(V+E) * O(log V) = O ((V+E) * log V)

- The space complexity in this case will also improve to O(V) as both the adjacency list and min-heap require O(V) space. So the total space complexity becomes

O(V) + O(V) = O(2V) = O(V)

Limitation of Dijkstra’s Algorithm

- The main limitation of Dijkstra's algorithm is that it does not work correctly with graphs that have negative edge weights. In fact, if there are negative weights in a graph, Dijkstra's algorithm can give incorrect results or even go into an infinite loop.
- The reason for this is that Dijkstra's algorithm assumes that the path with the smallest weight found so far is also the shortest path overall. However, when there are negative edge weights, it is possible that there may be a path with a larger weight that is actually shorter in length.
- To handle negative edge weights, there are other algorithms such as the Bellman-Ford algorithm and the A* algorithm that can be used.

Bellman Ford Algorithm: Shortest path and Negative Weight Cycle

Bellman-Ford Algorithm

- Bellman-Ford is a single source shortest path algorithm that determines the shortest path between a given source vertex and every other vertex in a graph. This algorithm can be used on both weighted and unweighted graphs
- A Bellman-Ford algorithm is also guaranteed to find the shortest path in a graph, similar to Dijkstra's algorithm. Although Bellman-Ford is slower than Dijkstra's algorithm, it is capable of handling graphs with negative edge weights, which makes it more versatile.
- The shortest path cannot be found if there exists a negative cycle in the graph. If we continue to go around the negative cycle an infinite number of times, then the cost of the path will continue to decrease (even though the length of the path is increasing). As a result, Bellman-Ford is also capable of detecting negative cycles, which is an important feature.

Why would one ever have edges with negative weights in real life?

- Negative weight edges might seem useless at first but they can explain a lot of phenomena like cashflow, the heat released/absorbed in a chemical reaction, etc.
- For instance, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption.
- If we want to find the set of reactions where minimum energy is required, then we will need to be able to factor in the heat absorption as negative weights and heat dissipation as positive weights.

The idea behind Bellman Ford Algorithm

The Bellman-Ford algorithm's primary principle is that it starts with a single source and calculates the distance to each node. The distance is initially unknown and assumed to be infinite, but as time goes on, the algorithm relaxes those paths by identifying a few shorter paths. Hence it is said that Bellman-Ford is based on ***Principle of Relaxation***

The idea behind Bellman Ford Algorithm

The Bellman-Ford algorithm's primary principle is that it starts with a single source and calculates the distance to each node. The distance is initially unknown and assumed to be infinite, but as time goes on, the algorithm relaxes those paths by identifying a few shorter paths. Hence it is said that Bellman-Ford is based on ***Principle of Relaxation***

Relaxation means

Suppose, Source vertex is ***A***

Distance from ***A*** to any vertex ***v = d[v]***

Current vertex is ***u.***

Distance from ***A*** to vertex ***u = d[u]***

Cost of going to vertex ***v*** from ***u = c[u,v]***

```
if (d[u]+c(u,v) < d[v]){
        d[v] = d[u]+c(u,v)
}
```

Principle of Relaxation of Edges for Bellman-Ford

- It states that for the graph having N vertices, all the edges should be relaxed N-1 times to compute the single source shortest path.
- In order to detect whether a negative cycle exists or not, relax all the edge one more time and if the shortest distance for any node reduces then we can say that a negative cycle exists. In short if we relax the edges N times, and there is any change in the shortest distance of any node between the N-1th and Nth relaxation than a negative cycle exists, otherwise not exist.

Why Relaxing Edges N-1 times, gives us Single Source Shortest Path?

- In the worst-case scenario, a shortest path between two vertices can have at most N-1 edges, where N is the number of vertices. This is because a simple path in a graph with N vertices can have at most N-1 edges, as it's impossible to form a closed loop without revisiting a vertex.
- By relaxing edges N-1 times, the Bellman-Ford algorithm ensures that the distance estimates for all vertices have been updated to their optimal values, assuming the graph doesn't contain any negative-weight cycles reachable from the source vertex. If a graph contains a negative-weight cycle reachable from the source vertex, the algorithm can detect it after N-1 iterations, since the negative cycle disrupts the shortest path lengths.

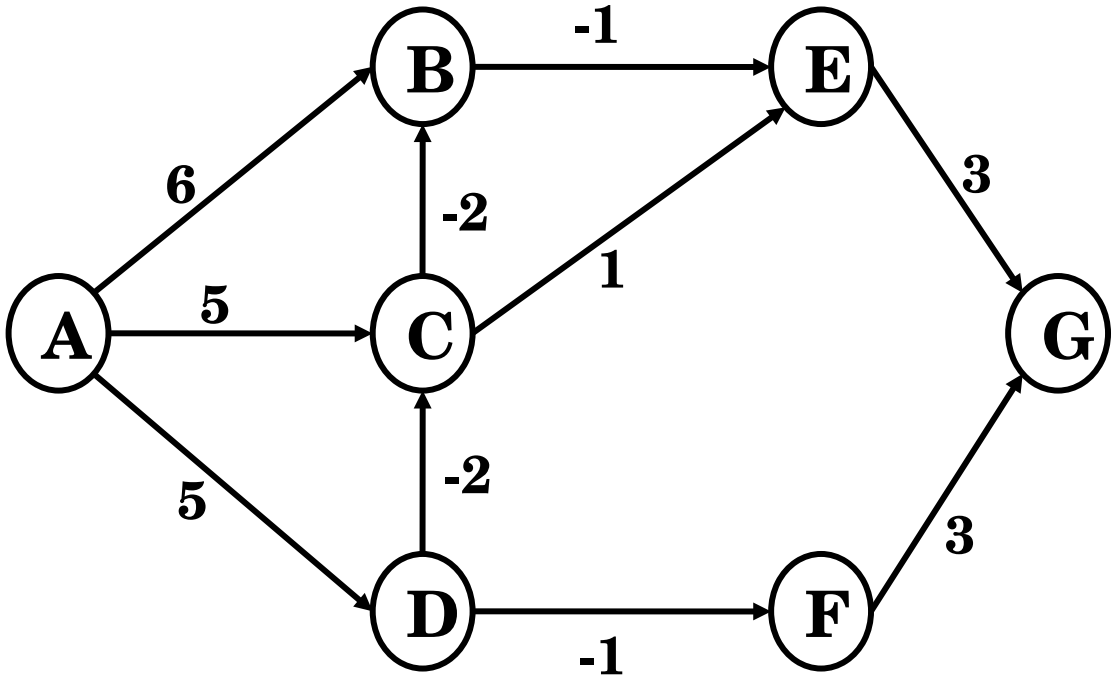
Why Relaxing Edges N-1 times, gives us Single Source Shortest Path?

- In summary, relaxing edges N-1 times in the Bellman-Ford algorithm guarantees that the algorithm has explored all possible paths of length up to N-1, which is the maximum possible length of a shortest path in a graph with N vertices. This allows the algorithm to correctly calculate the shortest paths from the source vertex to all other vertices, given that there are no negative-weight cycles.

Why Does the Reduction of Distance in the n'th Relaxation Indicates the Existence of a Negative Cycle?

- As previously discussed, achieving the single source shortest paths to all other nodes takes N-1 relaxations. If the N'th relaxation further reduces the shortest distance for any node, it implies that a certain edge with negative weight has been traversed once more. It is important to note that during the N-1 relaxations, we presumed that each vertex is traversed only once. However, the reduction of distance during the N'th relaxation indicates revisiting a vertex.

Example



There are 7 vertices.

n = 7

According to Bellman Ford algorithm we have to ***Relax*** each edge

(n-1) = 7-1 = 6 times.

Iteration 1

(A, B)

(A, C)

(A, D)

(B, E)

(C, B)

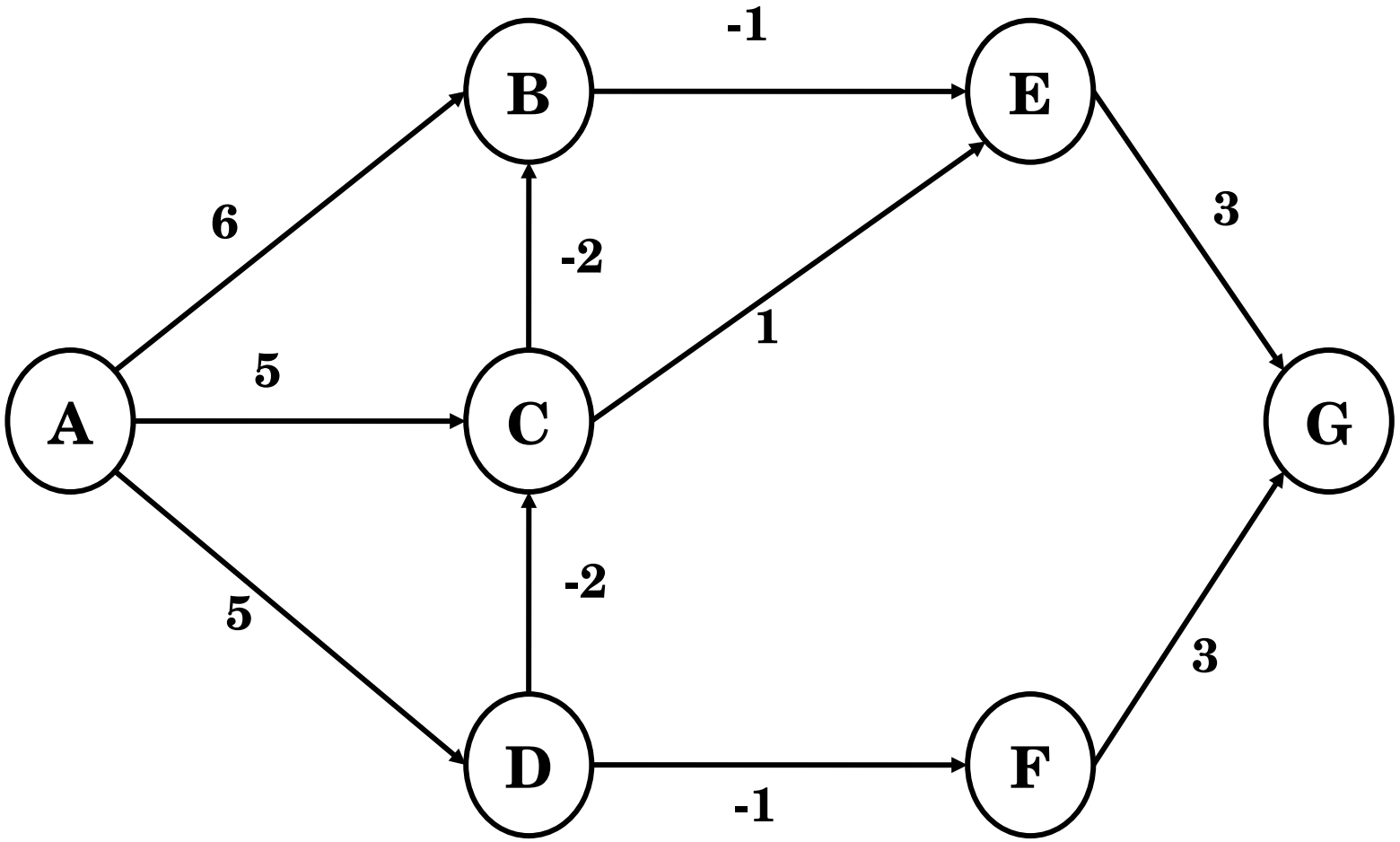
(C, E)

(D, C)

(D, F)

(E, G)

(F, G)



Iteration 1

(A, B)

(A, C)

(A, D)

(B, E)

(C, B)

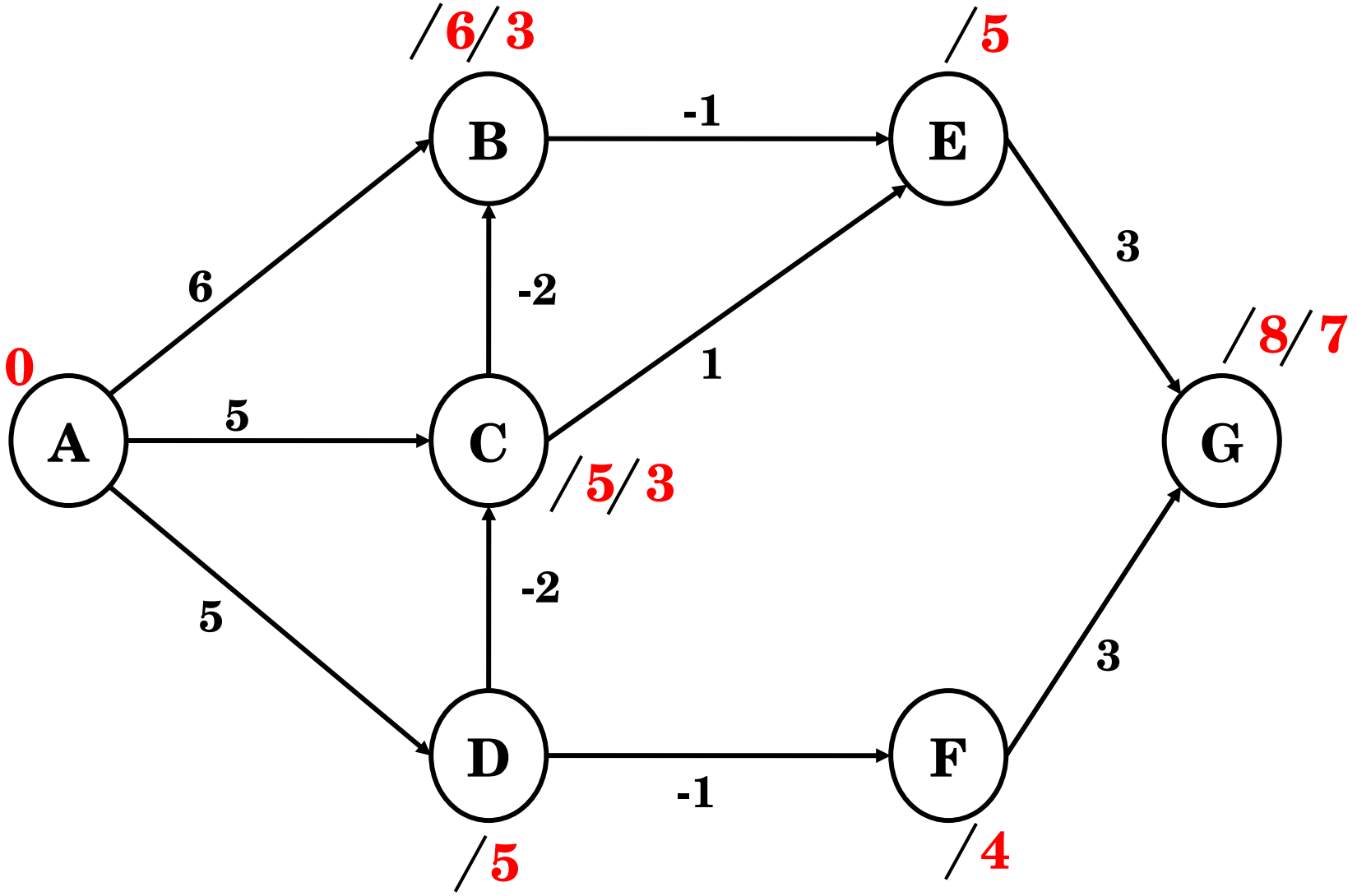
(C, E)

(D, C)

(D, F)

(E, G)

(F, G)



Iteration 2

(A, B)

(A, C)

(A, D)

(B, E)

(C, B)

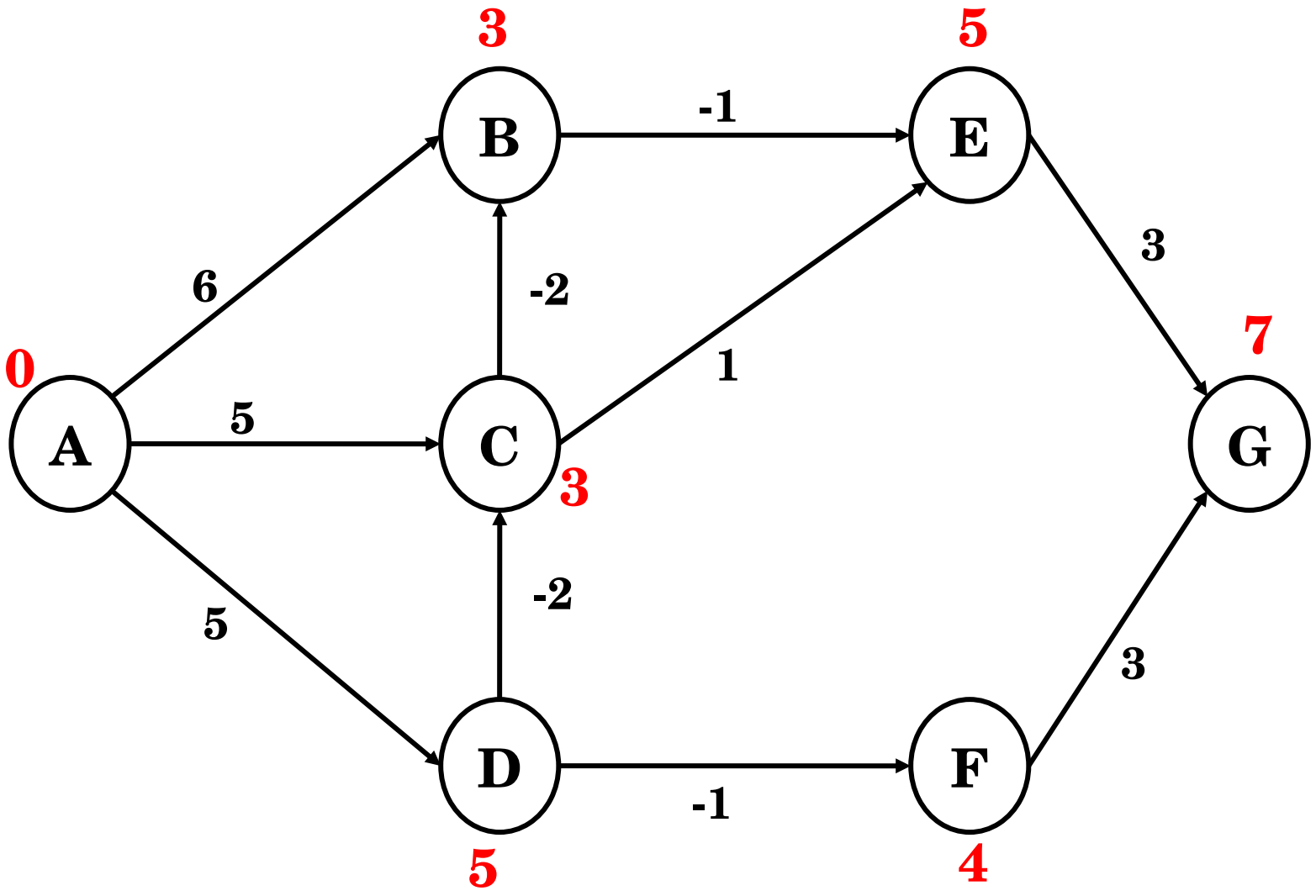
(C, E)

(D, C)

(D, F)

(E, G)

(F, G)



Iteration 2

(A, B)

(A, C)

(A, D)

(B, E)

(C, B)

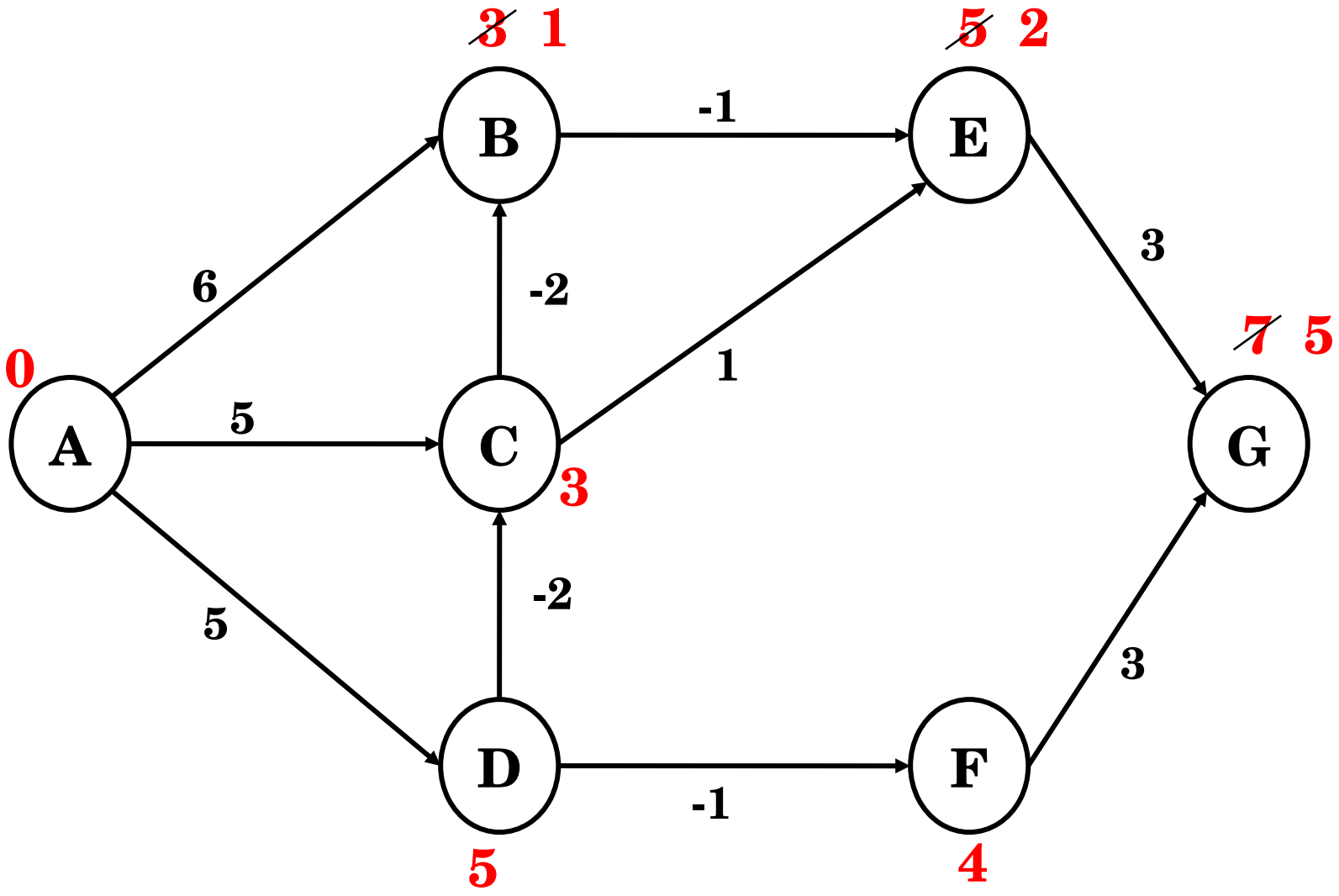
(C, E)

(D, C)

(D, F)

(E, G)

(F, G)



Iteration 3

(A, B)

(A, C)

(A, D)

(B, E)

(C, B)

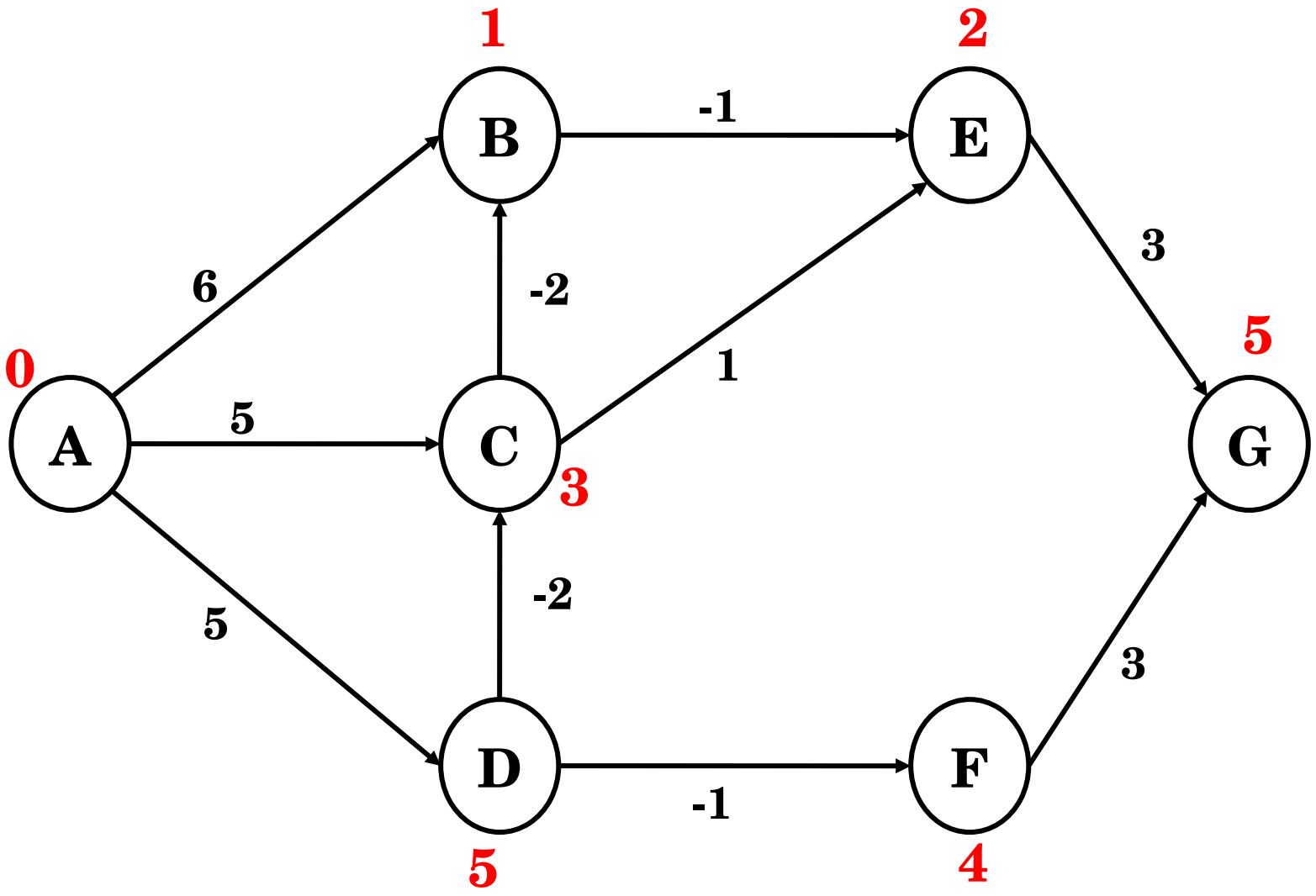
(C, E)

(D, C)

(D, F)

(E, G)

(F, G)



Iteration 3

(A, B)

(A, C)

(A, D)

(B, E)

(C, B)

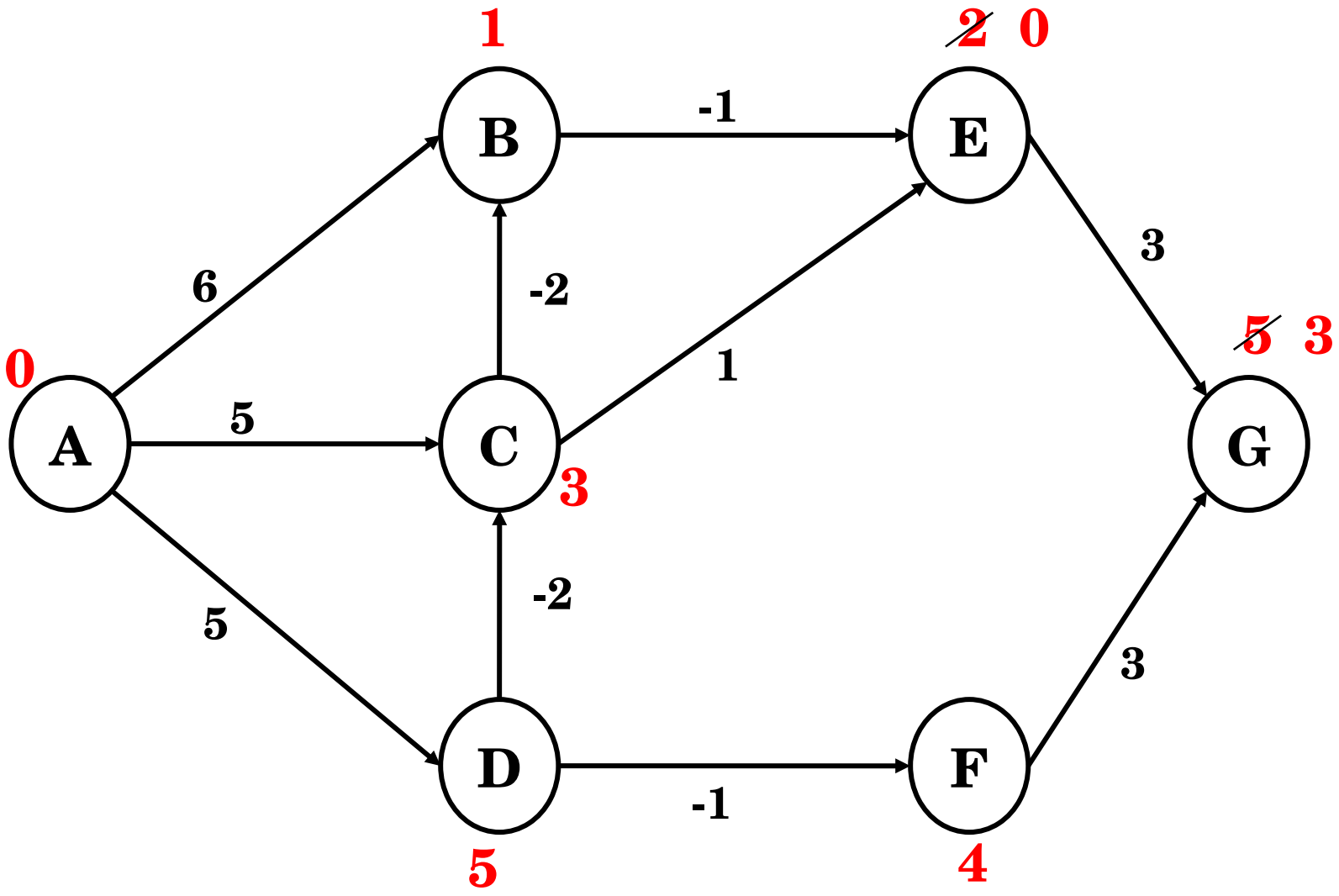
(C, E)

(D, C)

(D, F)

(E, G)

(F, G)



Iteration 4

(A, B)

(A, C)

(A, D)

(B, E)

(C, B)

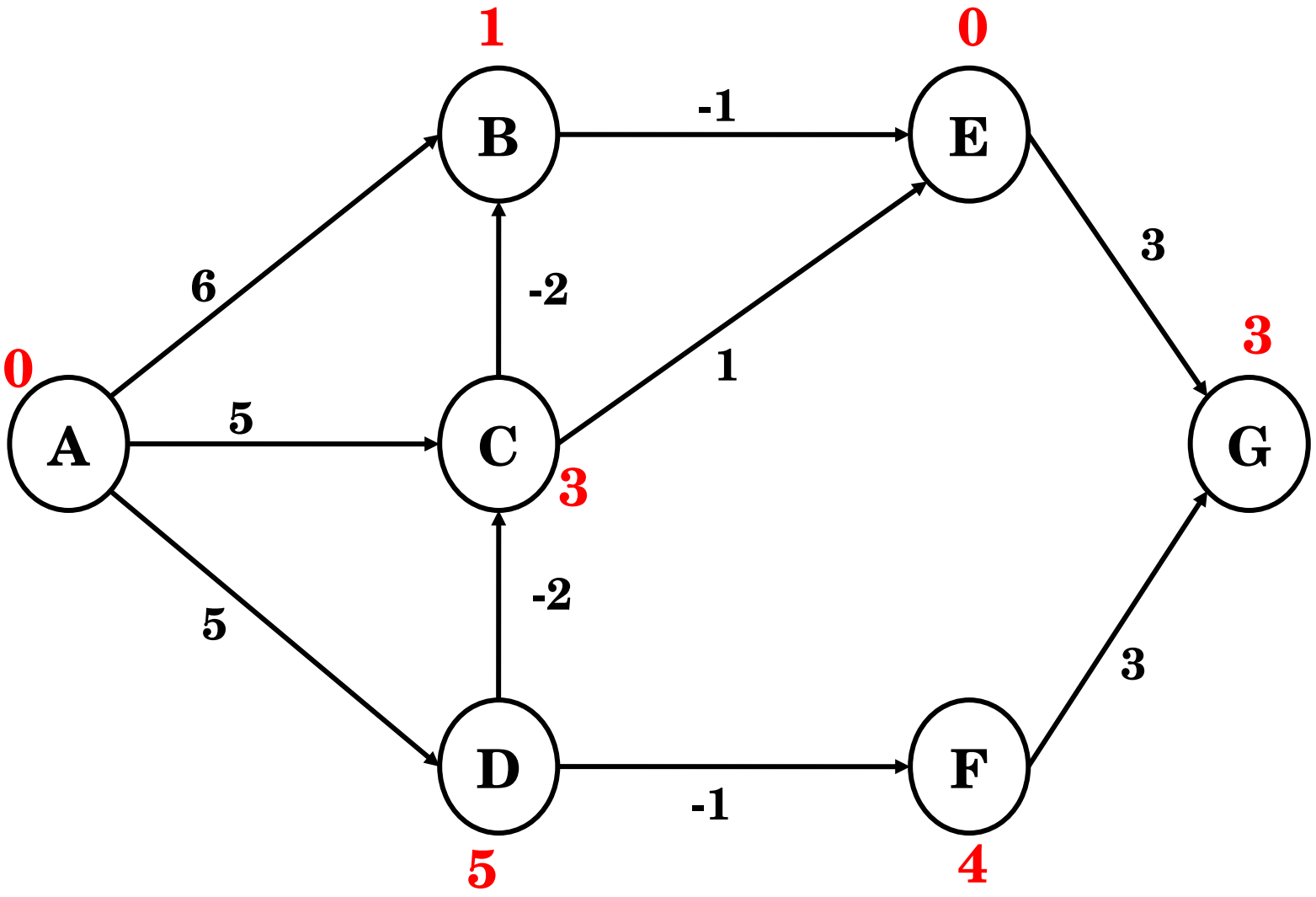
(C, E)

(D, C)

(D, F)

(E, G)

(F, G)



Iteration 4

(A, B)

(A, C)

(A, D)

(B, E)

(C, B)

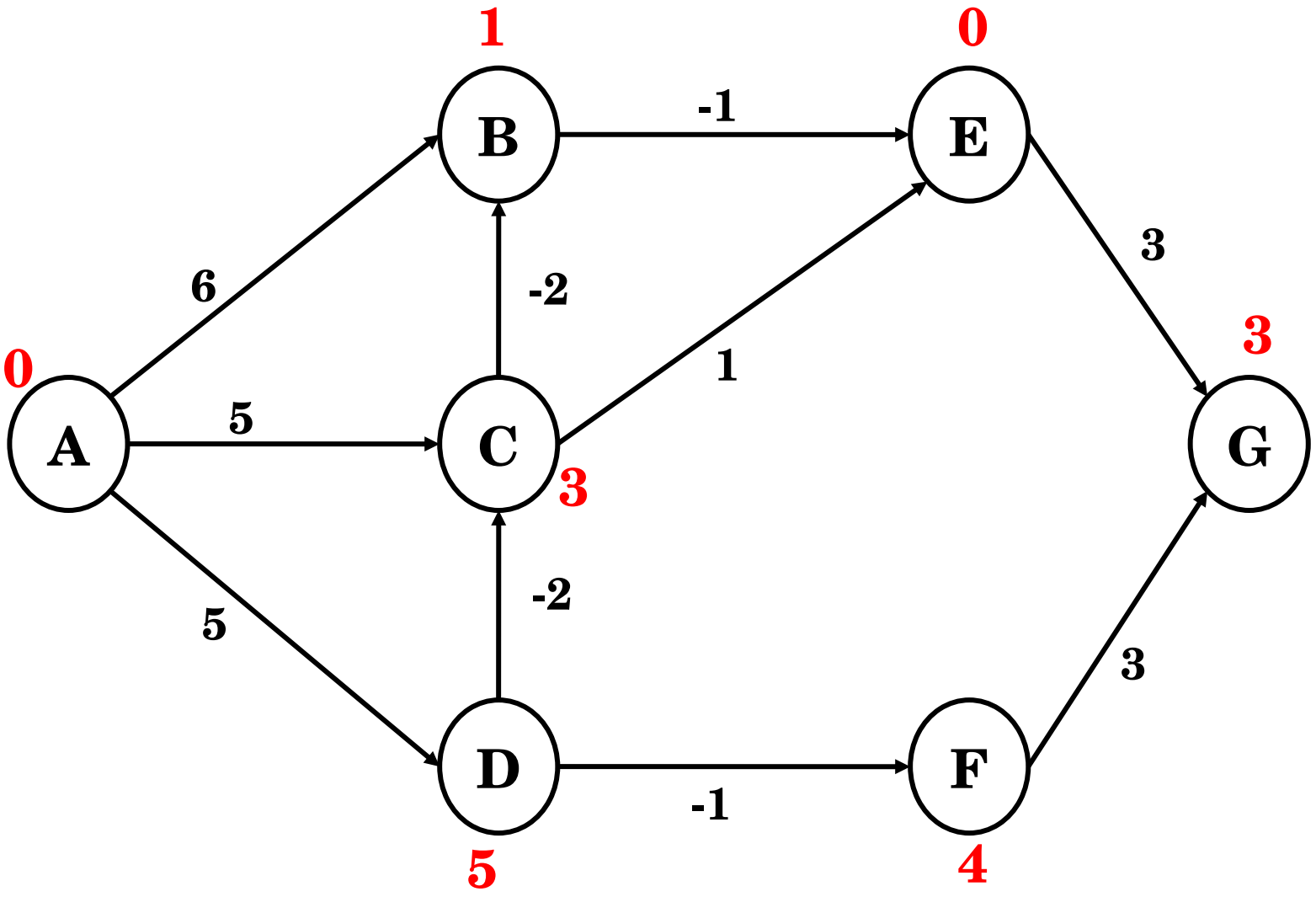
(C, E)

(D, C)

(D, F)

(E, G)

(F, G)



Iteration 5

(A, B)

(A, C)

(A, D)

(B, E)

(C, B)

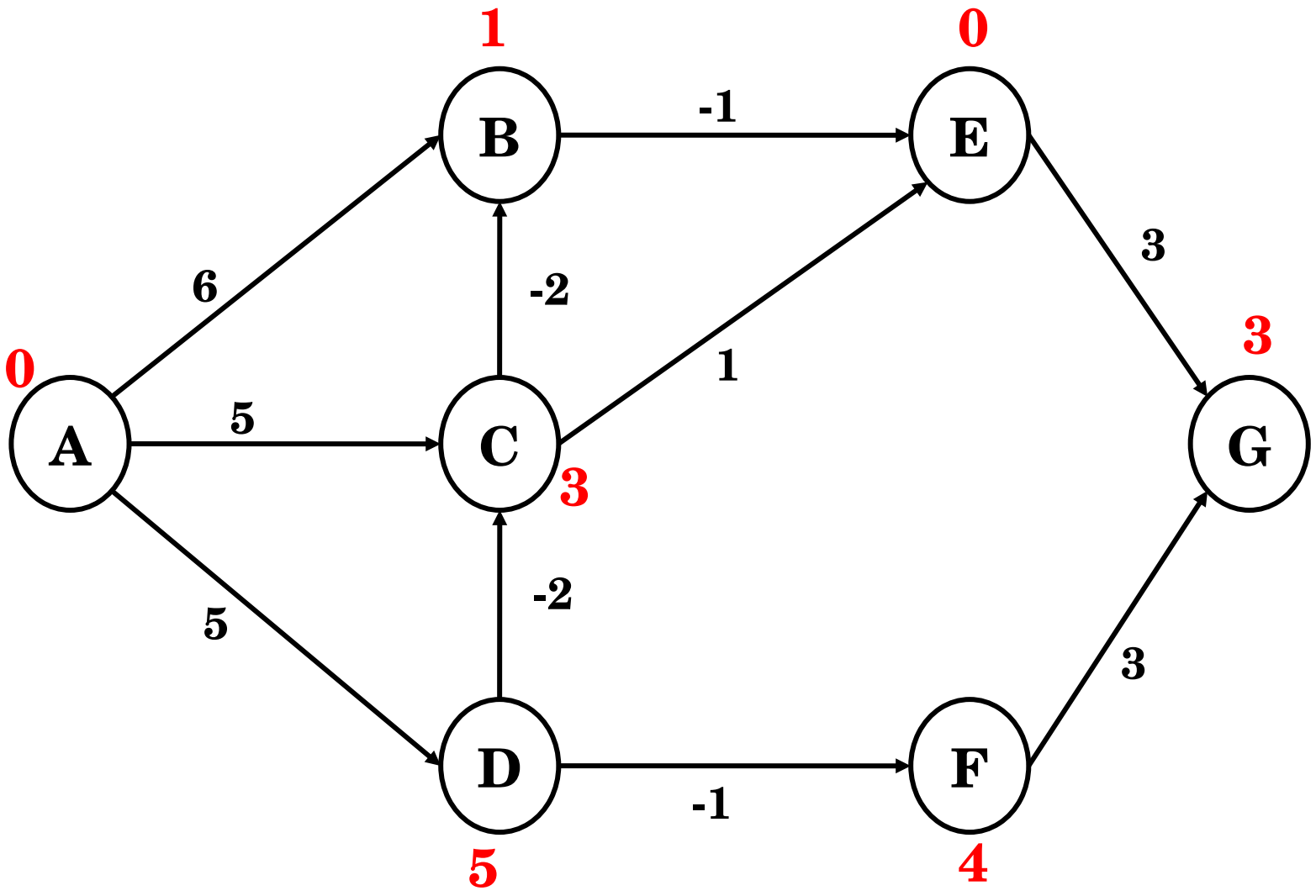
(C, E)

(D, C)

(D, F)

(E, G)

(F, G)



Iteration 6

(A, B)

(A, C)

(A, D)

(B, E)

(C, B)

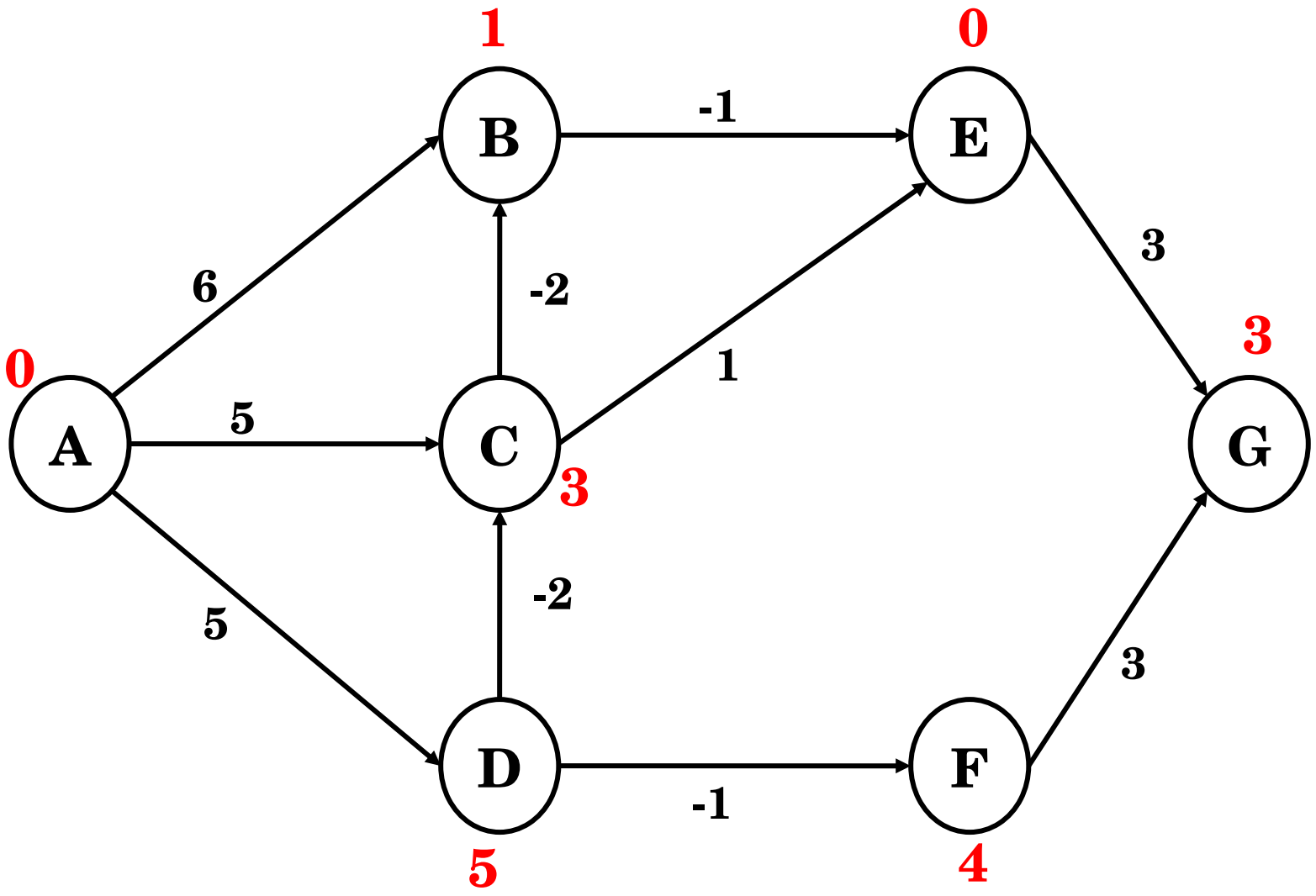
(C, E)

(D, C)

(D, F)

(E, G)

(F, G)



Time and Space Complexity of Bellman Ford Algorithm

Suppose,

No. of vertices = V

No. of edges = E

At each iteration, E number of edges are getting relaxed. There are (V-1) number of iteration. Total number of relaxation will be: E * (V-1)

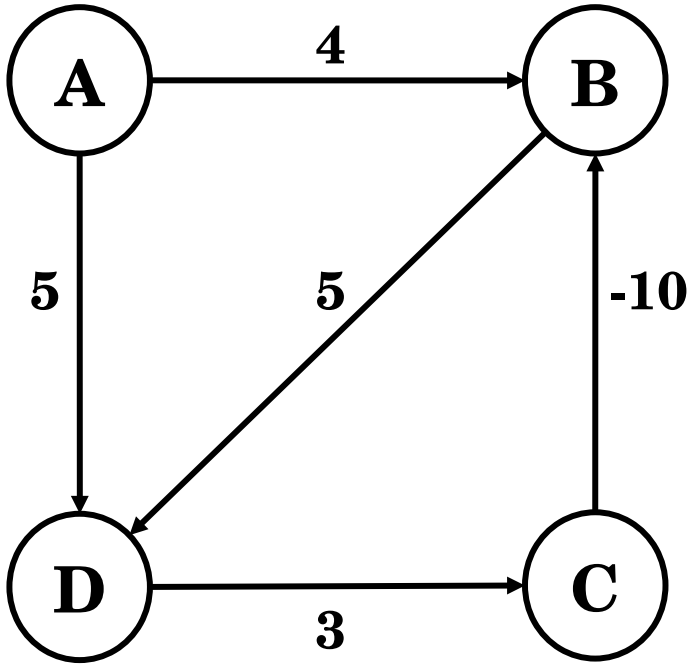
Time complexity is O (E * V)

If the graph is a complete graph, number of edges will be

Total number of relaxation will be: * (V-1)

Time complexity is O (V^3)

Limitation of Bellman Ford Algorithm



There are 4 vertices.

n = 4

According to Bellman Ford algorithm we have to ***Relax*** each edge

(n-1) = 4-1 = 3 times.

Iteration 1

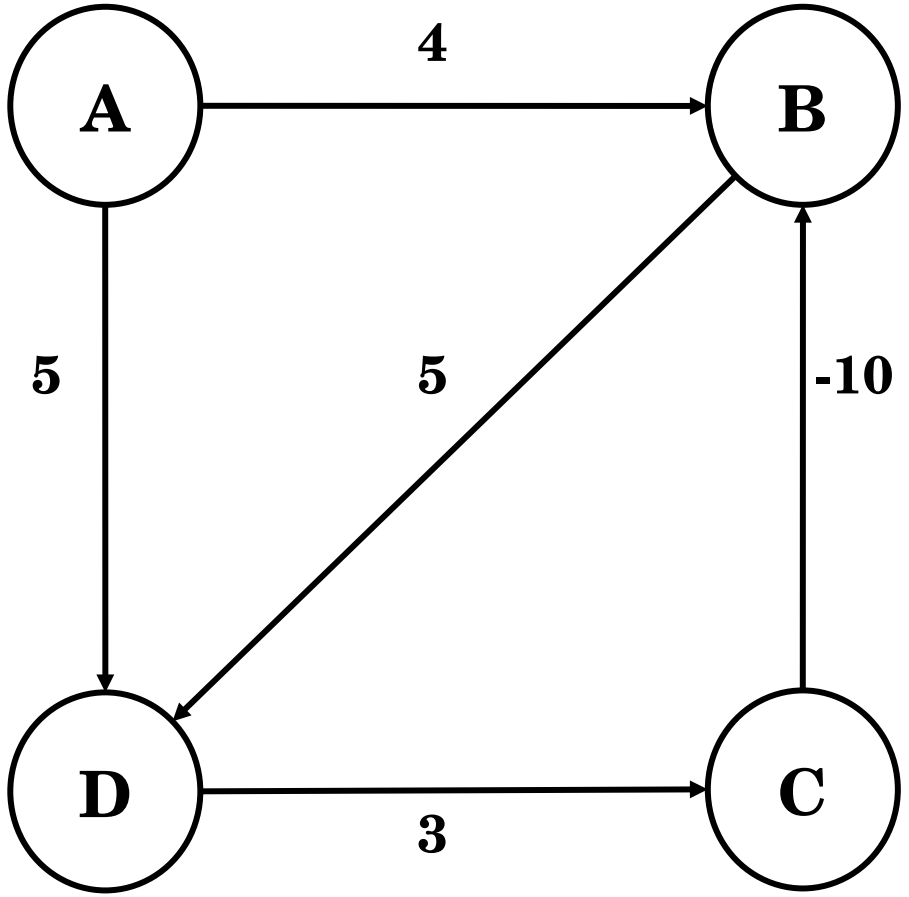
(A, B)

(A, D)

(B, D)

(C, B)

(D, C)



Iteration 1

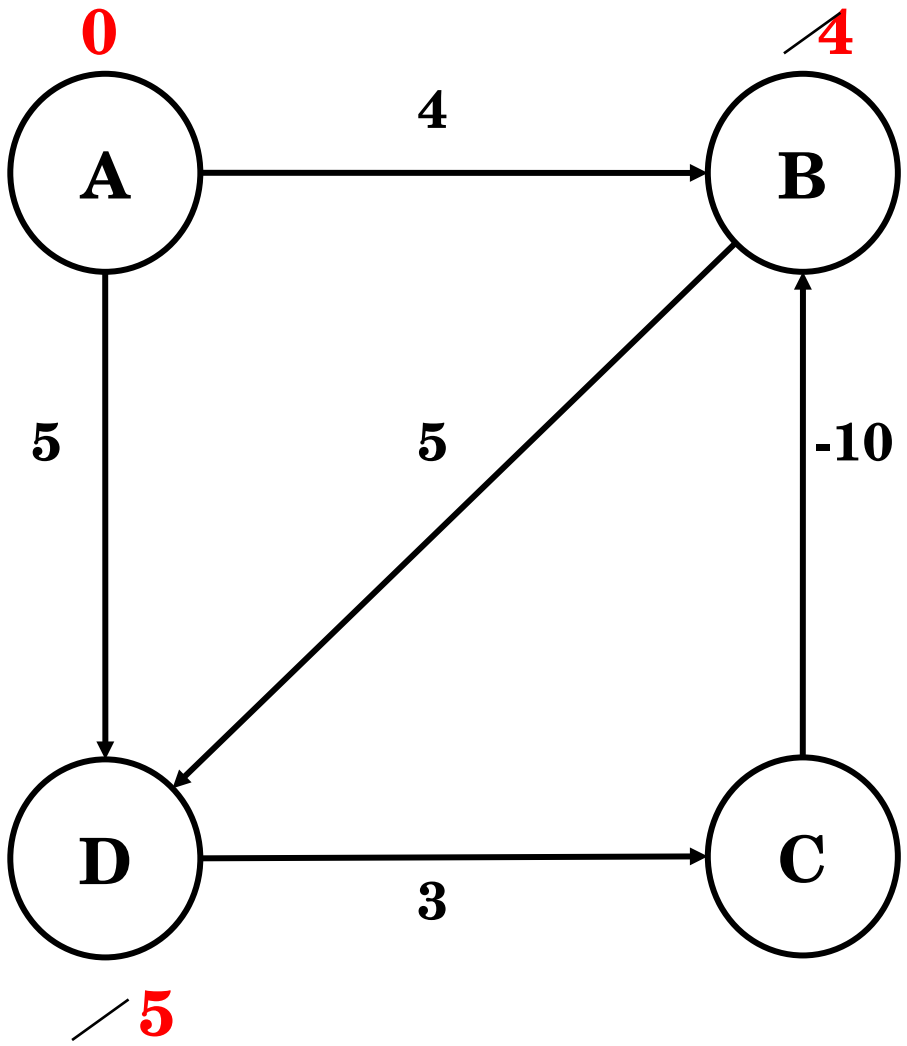
(A, B)

(A, D)

(B, D)

(C, B)

(D, C)



Iteration 2

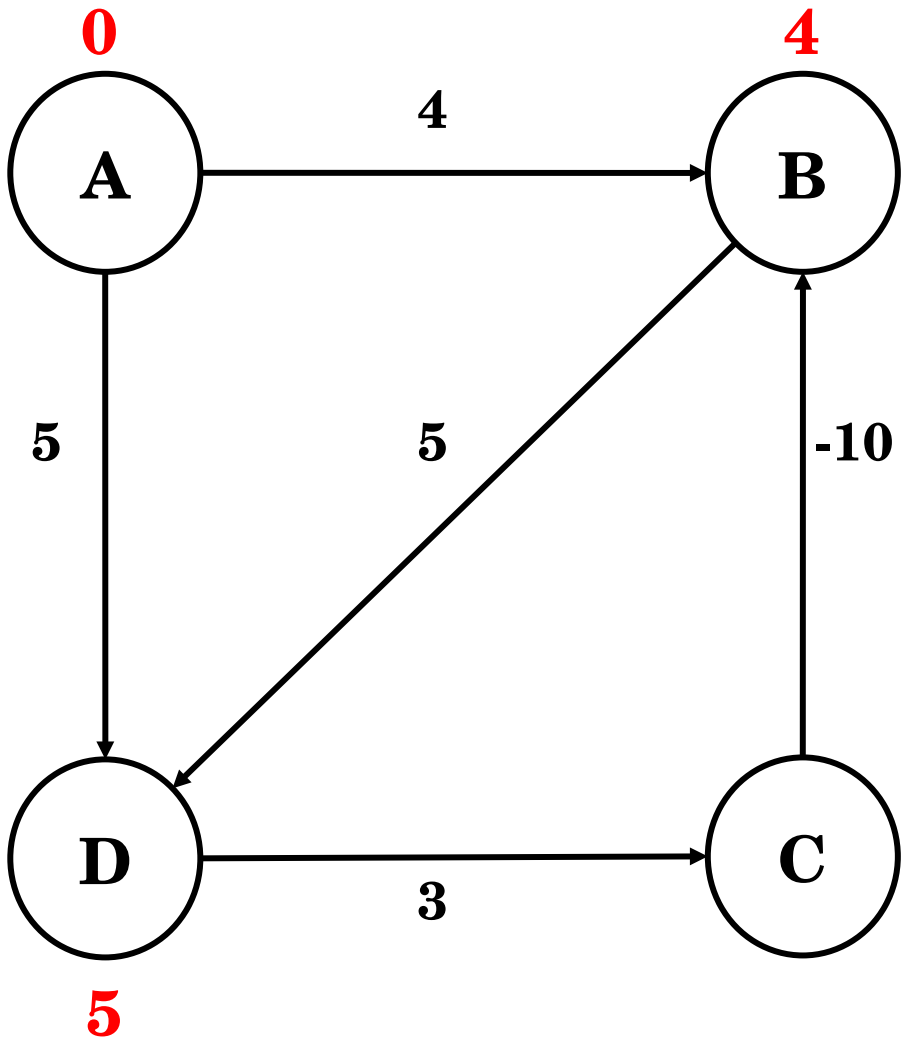
(A, B)

(A, D)

(B, D)

(C, B)

(D, C)



Iteration 2

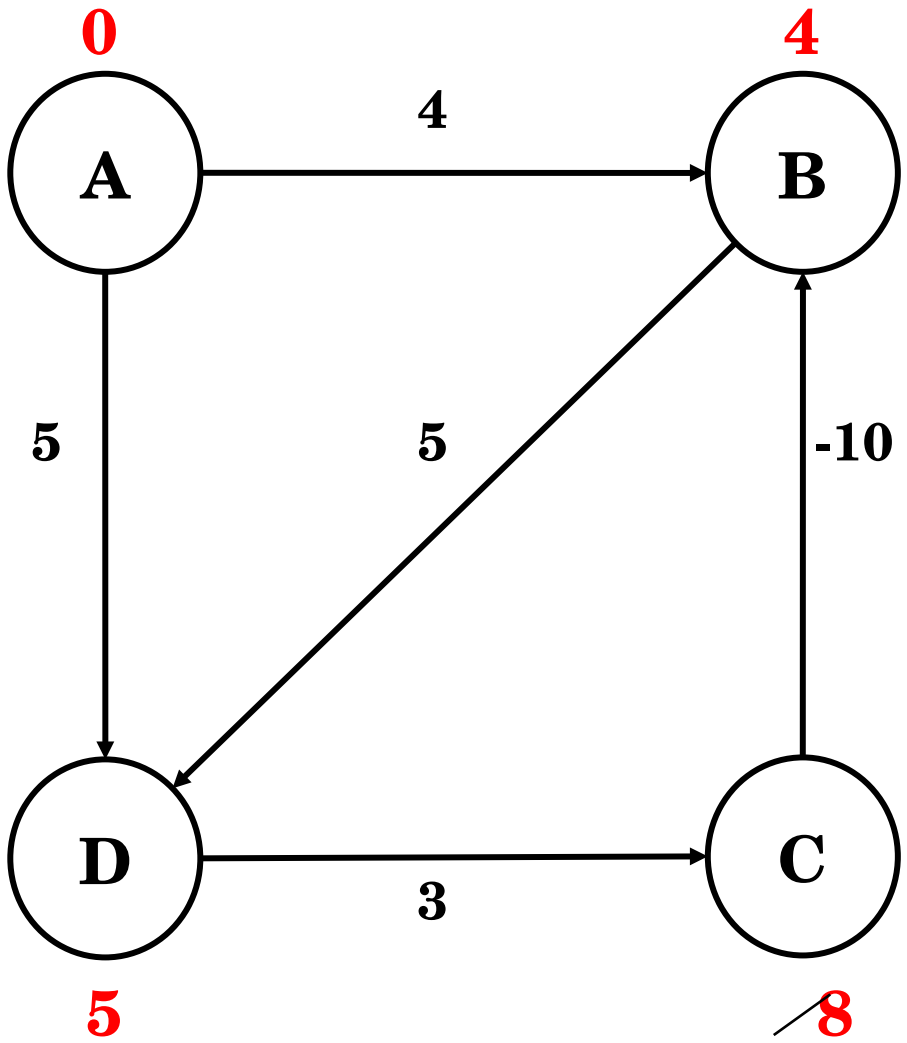
(A, B)

(A, D)

(B, D)

(C, B)

(D, C)



Iteration 3

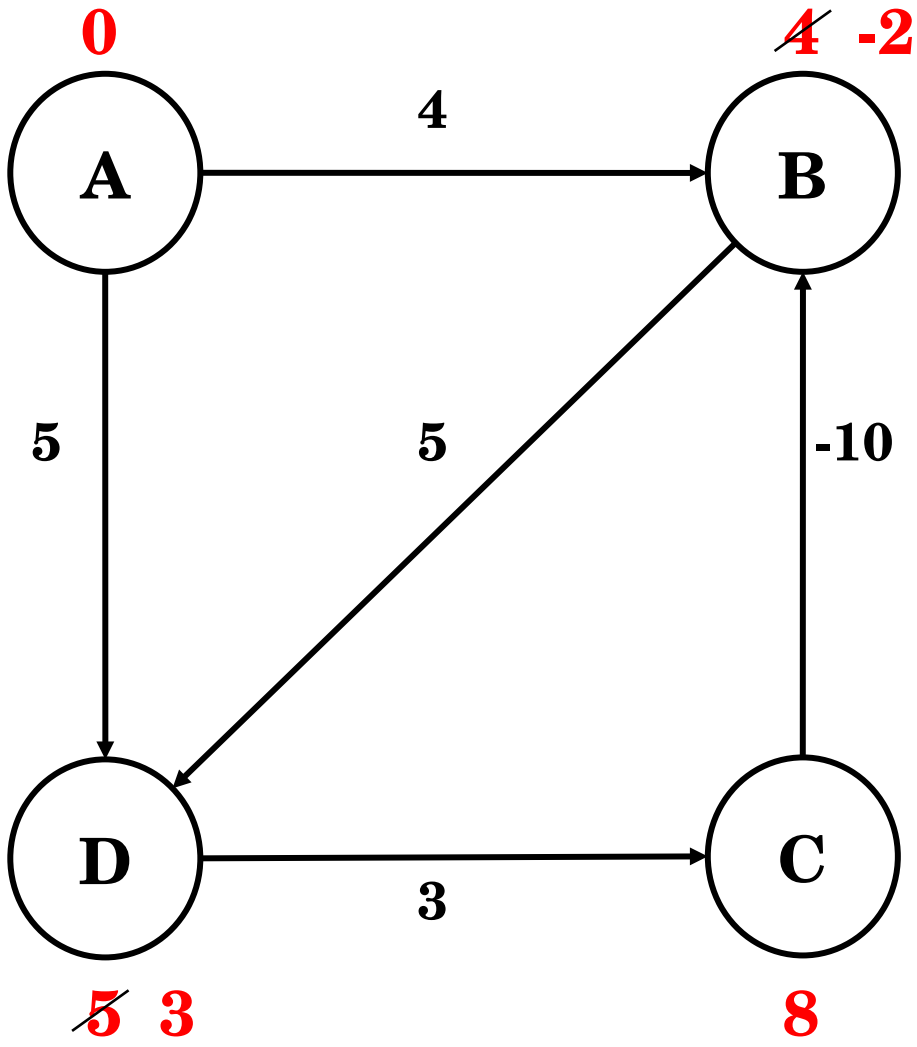
(A, B)

(A, D)

(B, D)

(C, B)

(D, C)



Iteration 4

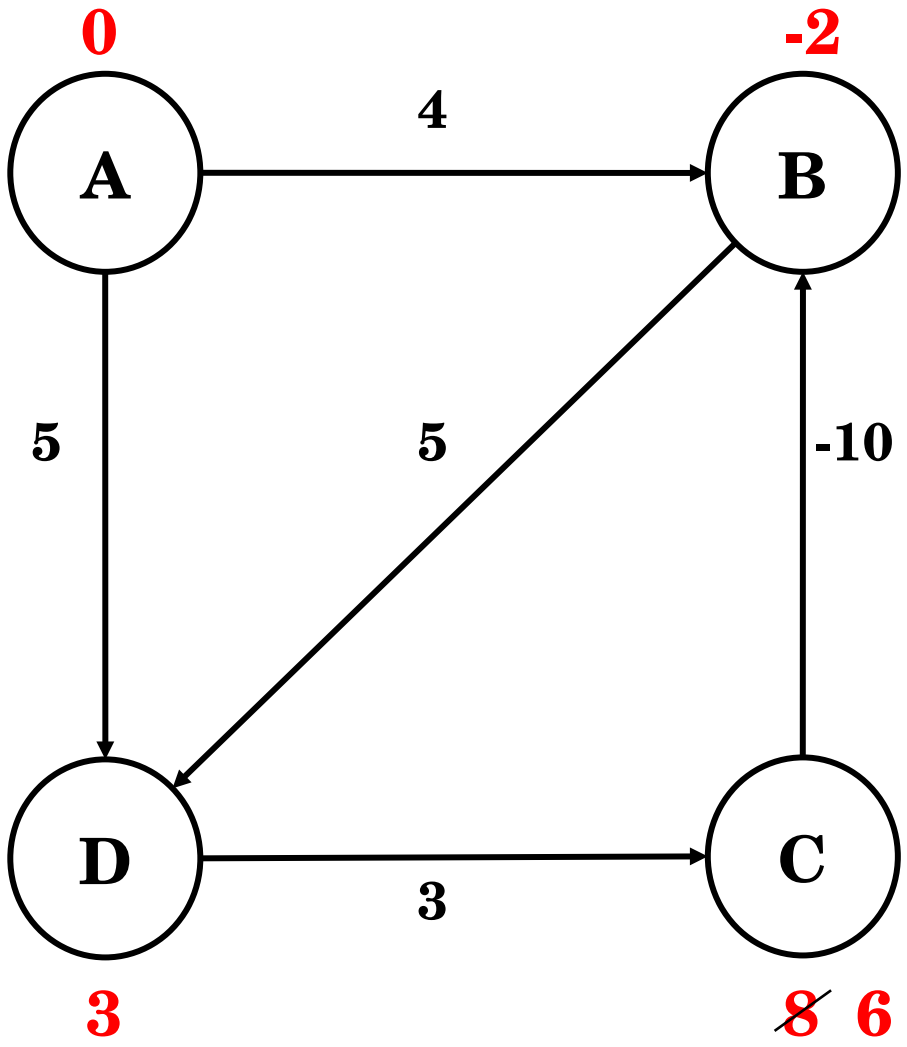
(A, B)

(A, D)

(B, D)

(C, B)

(D, C)



Reason

The distances were supposed to stop changing after iteration 3. but in this case, the distance is still changing. This is because there is a negative weight cycle:

B □ D □ C

5 + 3 – 10

= -2

The limitation is: Bellman Ford Algorithm can’t give us correct result is there is a negative edge cycle present in the graph.

The limitation can be used as a feature

In Bellman Ford algorithm, the distance continue changing even after (V-1) iterations if a negative weight cycle is present. We can use this property to check whether is a negative edge cycle in the graph or not.


Maximum Flow Algorithms
Ford-Fulkerson Algorithm
Edmonds-Karp algorithm

Ford-Fulkerson Algorithm

Flow network

In graph theory, a flow network is defined as a directed graph involving a source(S) and a sink(T) and several other nodes connected with edges. Each edge has an individual capacity which is the maximum limit of flow that edge could allow.

Flow in the network should follow the following conditions:

- For any non-source and non-sink node, the input flow is equal to output flow.
- For any edge(E_i) in the network, $0 \leq flow(E_i) \leq capacity\ (E_i)$
- Total flow out of the source node is equal total to flow in to the sink node.
- Net flow in the edges follows skew symmetry i.e. $F(u,v) = -F(u,v)$, where $F(u,v)$ is flow from node u to node v. This leads to a conclusion where you have to sum up all the flows between two nodes(either directions) to find net flow between the nodes initially.

Flow network

A good analogy for a flow network is the following visualization: We represent edges as water pipes, the capacity of an edge is the maximal amount of water that can flow through the pipe per second, and the flow of an edge is the amount of water that currently flows through the pipe per second. This motivates the first flow condition. There cannot flow more water through a pipe than its capacity. The vertices act as junctions, where water comes out of some pipes, and then, these vertices distribute the water in some way to other pipes. This also motivates the second flow condition. All the incoming water has to be distributed to the other pipes in each junction. It cannot magically disappear or appear. The source $\boldsymbol{s}$ is origin of all the water, and the water can only drain in the sink $\boldsymbol{t}$.

Maximum Flow

It is defined as the maximum amount of flow that the network would allow to flow from source to sink. Multiple algorithms exist in solving the maximum flow problem. Two major algorithms to solve these kind of problems are Ford-Fulkerson algorithm and Edmond Karp Algorithm.

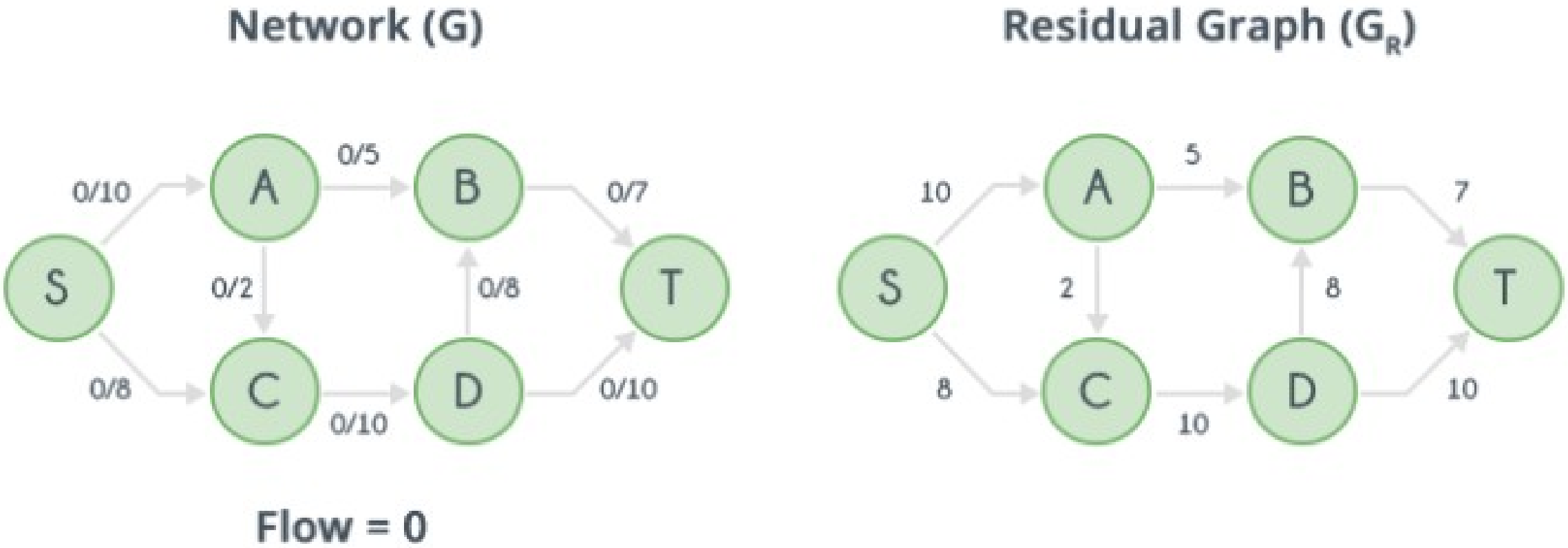
Algorithm

Inputs required are network graph ***G***, source node ***S*** and sink node ***T***

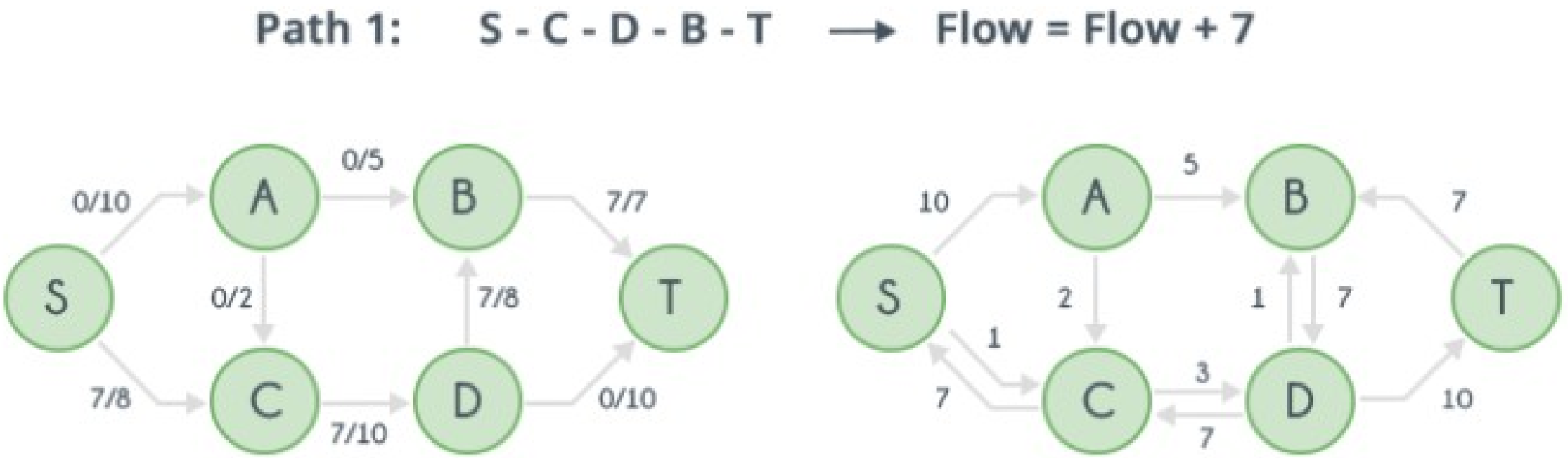
```
function: FordFulkerson(Graph G,Node S,Node T):
    Initialise flow in all edges to 0
    while (there exists an augmenting path(P) between S and T in residual network graph):
        Augment flow between S to T along the path P
        Update residual network graph
    return
```

An **augmenting path** is a simple path from source to sink which do not include any cycles and that pass only through positive weighted edges. A residual network graph indicates how much more flow is allowed in each edge in the network graph. If there are no augmenting paths possible from ***S*** to ***T***, then the flow is maximum. The result i.e. the maximum flow will be the total flow out of source node which is also equal to total flow in to the sink node.

Example

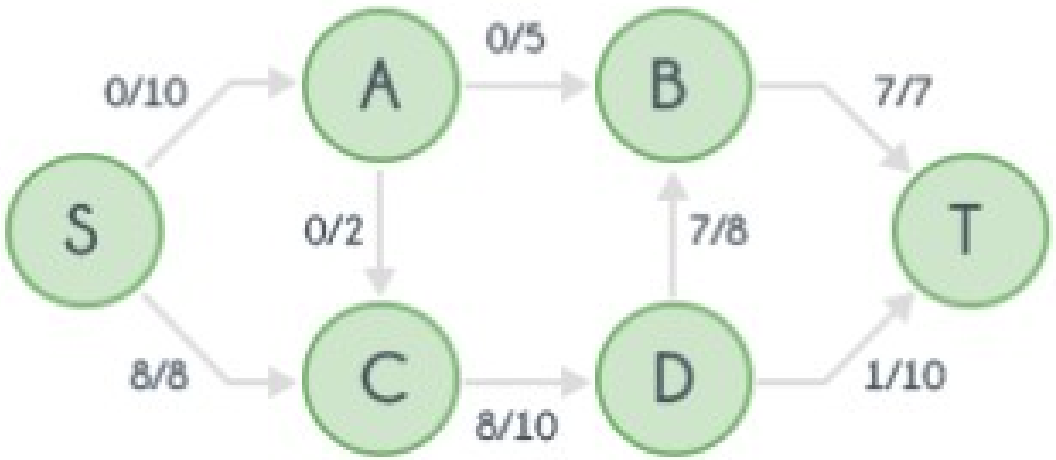

Network (G)
S
A
B
C
D
T
0/10
0/5
0/7
0/2
0/8
0/8
0/10
0/10
Flow = 0
Residual Graph (G_R)
S
A
B
C
D
T
10
5
7
2
8
8
10
10

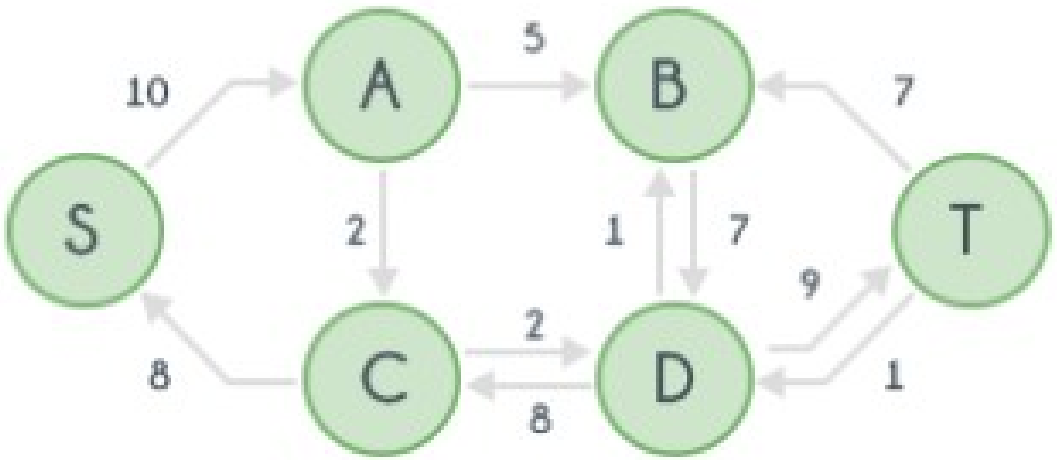
Example


Path 1: S - C - D - B - T ⟶ Flow = Flow + 7
S
A
B
C
D
T
0/10
0/5
7/7
0/2
7/8
7/8
7/10
0/10
S
A
B
C
D
T
10
5
7
2
1
7
1
7
3
7
10

Example

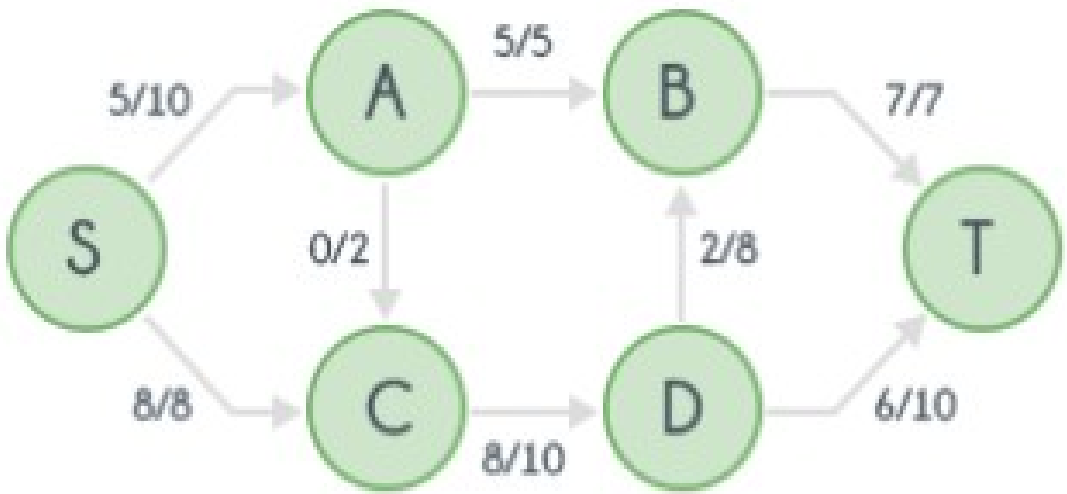
Path 2: S - C - D - T → Flow = Flow + 1

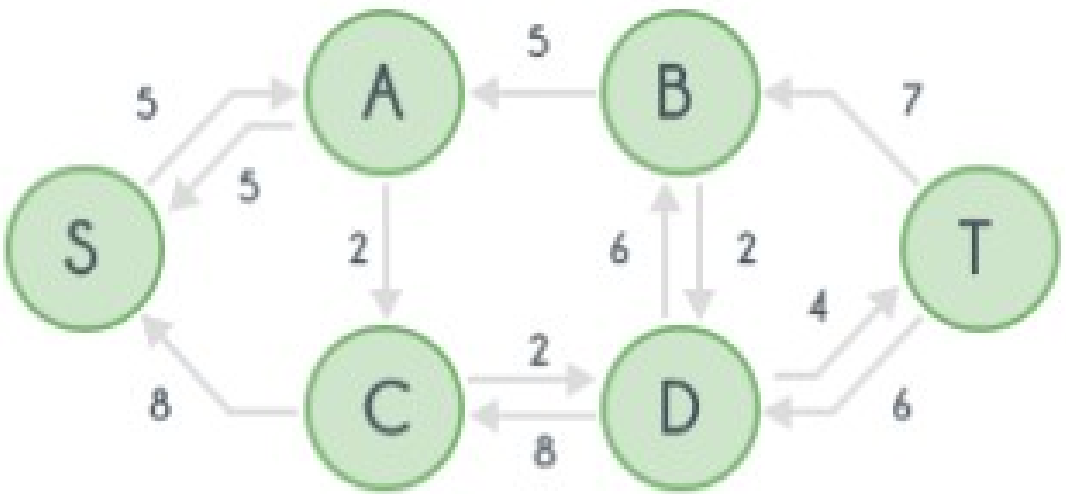




Example

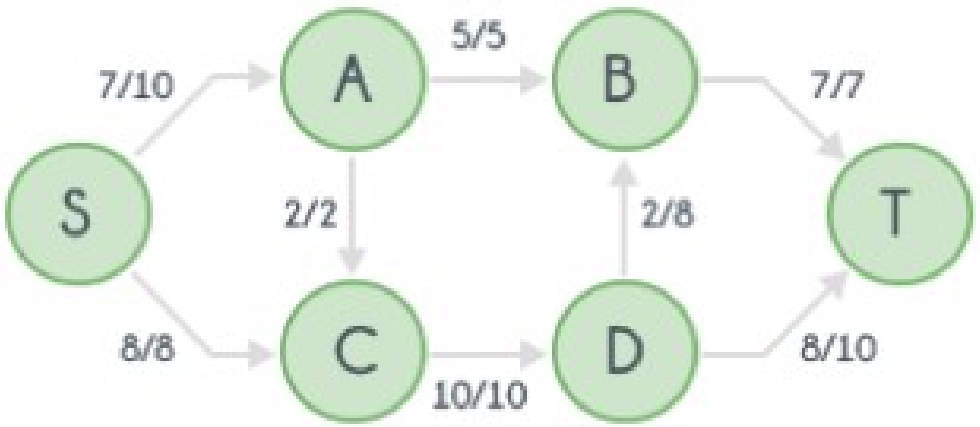
Path 3: S - A - B - T → Flow = Flow + 5

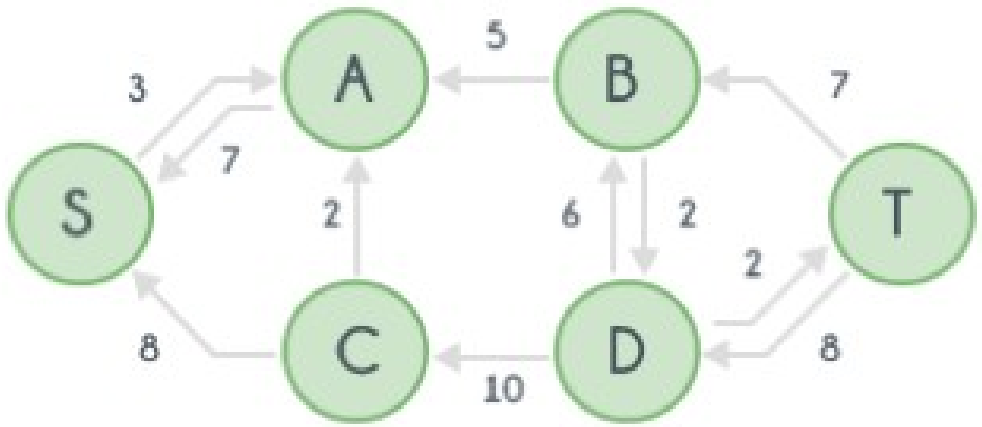




Example

Path 4: S - A - C - D - T ⟶ Flow = Flow + 2





No More Paths Left

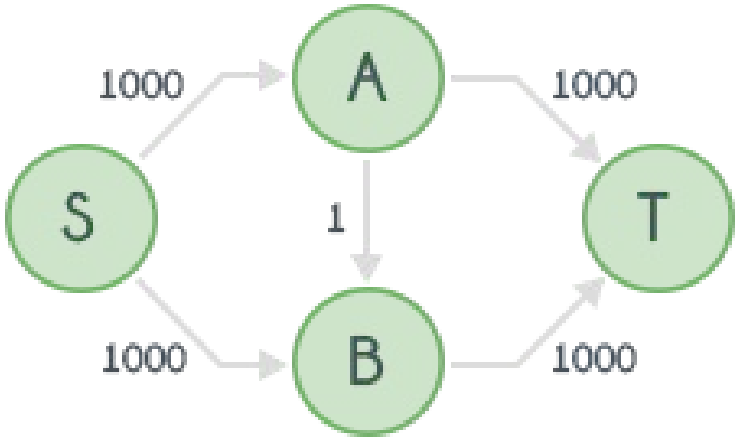
Max Flow = 15

Implementation

- An augmenting path in residual graph can be found using DFS or BFS.
- Updating residual graph includes following steps: (refer the diagrams for better understanding)
  - For every edge in the augmenting path, a value of minimum capacity in the path is subtracted from all the edges of that path.
  - An edge of equal amount is added to edges in reverse direction for every successive nodes in the augmenting path.

Limitation of Ford-Fulkerson Algorithm

The complexity of Ford-Fulkerson algorithm cannot be accurately computed as it all depends on the path from source to sink. For example, considering the network shown below, if each time, the path chosen are $S \rightarrow A \rightarrow B \rightarrow T$ and $S \rightarrow B \rightarrow A \rightarrow T$ alternatively, then it can take a very long time.

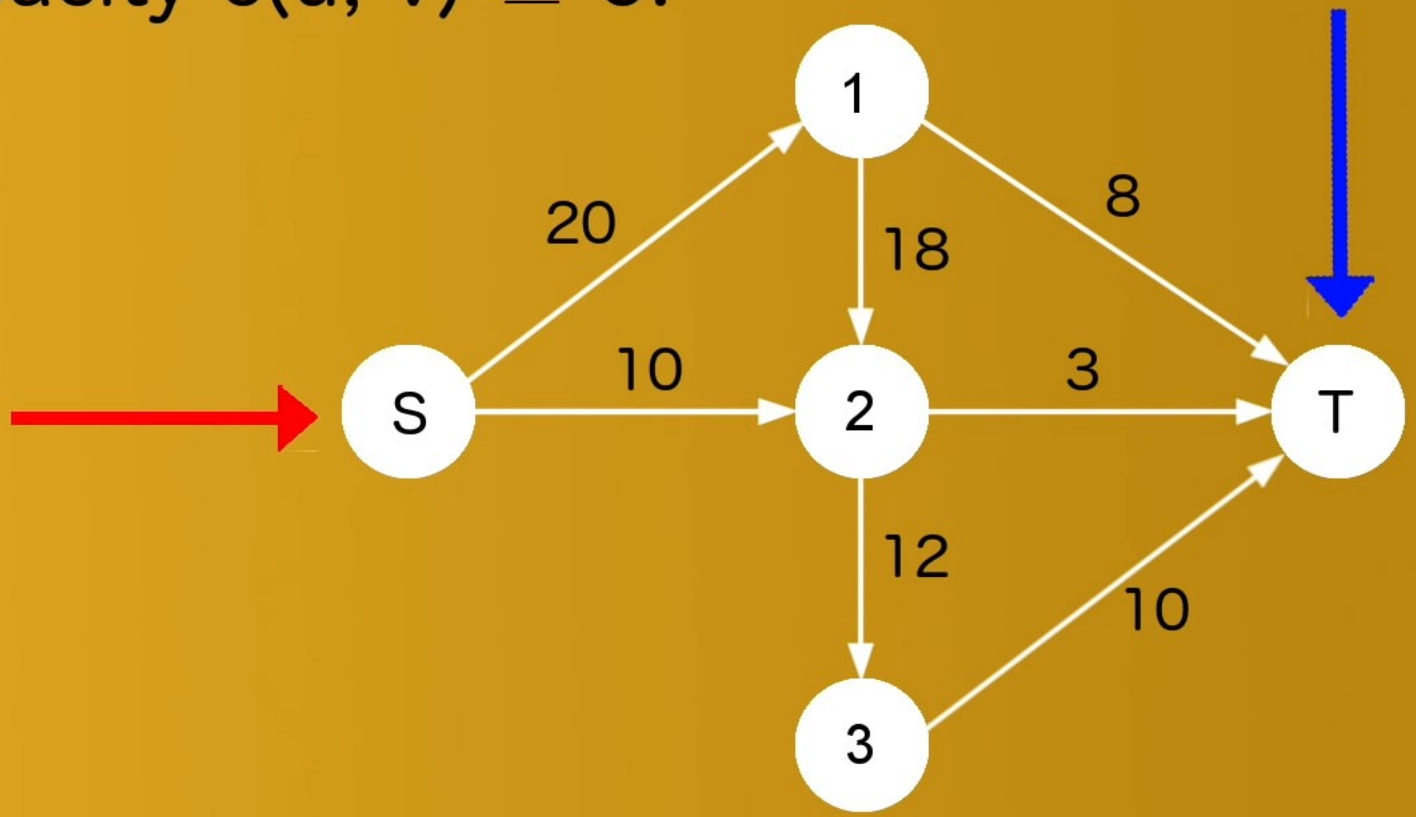


Instead, if path chosen are only S → A → T and S → B → T, it also generate maximum flow in just 2 steps.

Edmonds-Karp Algorithm

Flow Network

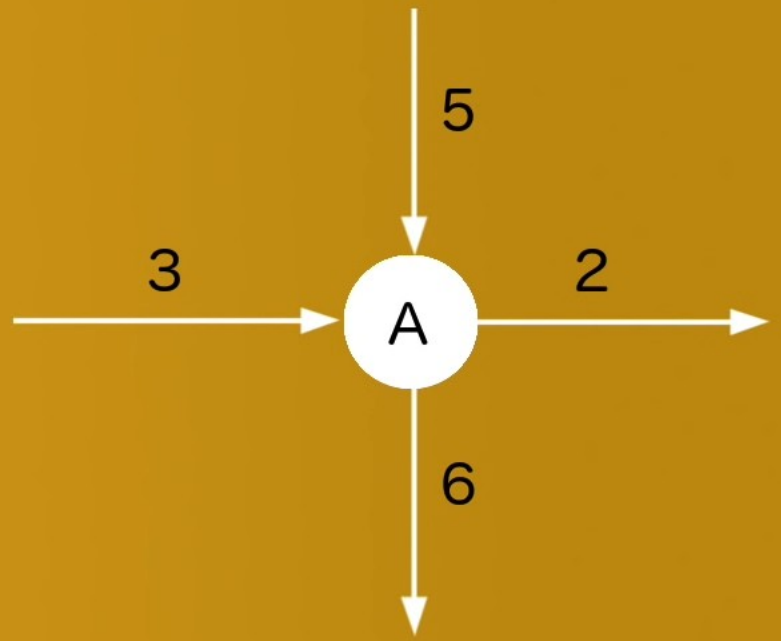
A flow network G = (V, E) is a directed graph in which each edge (u, v) ∈ E has a nonnegative capacity c(u, v) ≥ 0.



Flow Network

$$\forall u, v \in V \Rightarrow 0 \leq f(u, v) \leq c(u, v)$$

$$\forall u \in V - \{s, t\} \Rightarrow \sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$$

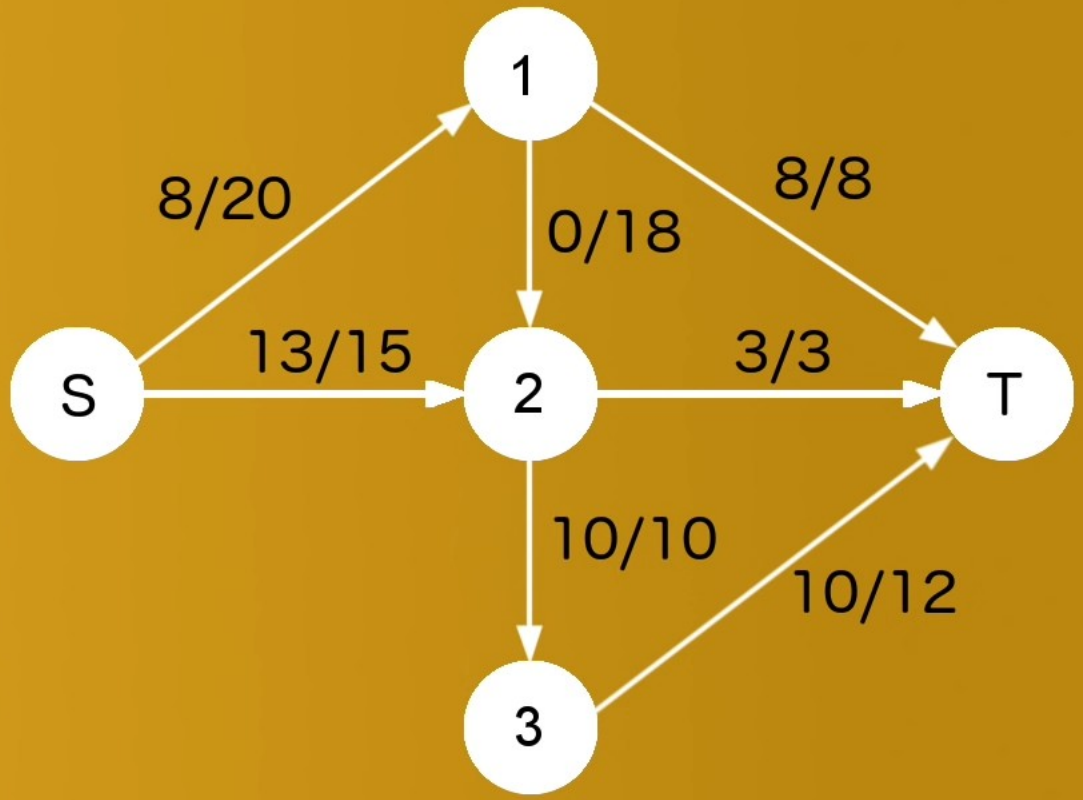


Flow

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

$$|f| = f(s,1) + f(s,2) = 21$$

The max flow (f*) is the maximum flow through a given network



Residual Networks

Given a flow network G=(V,E) and a flow f. The residual network of G induced by f is $G_f = (V,E_f)$

$E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$

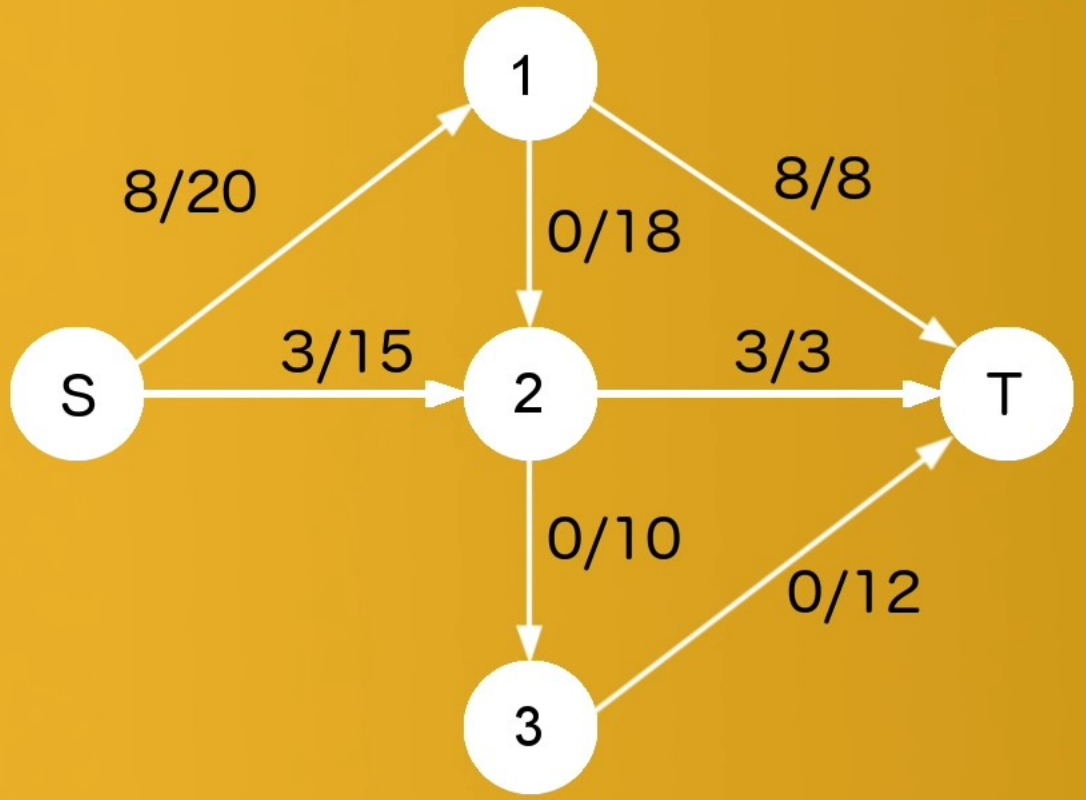
where

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

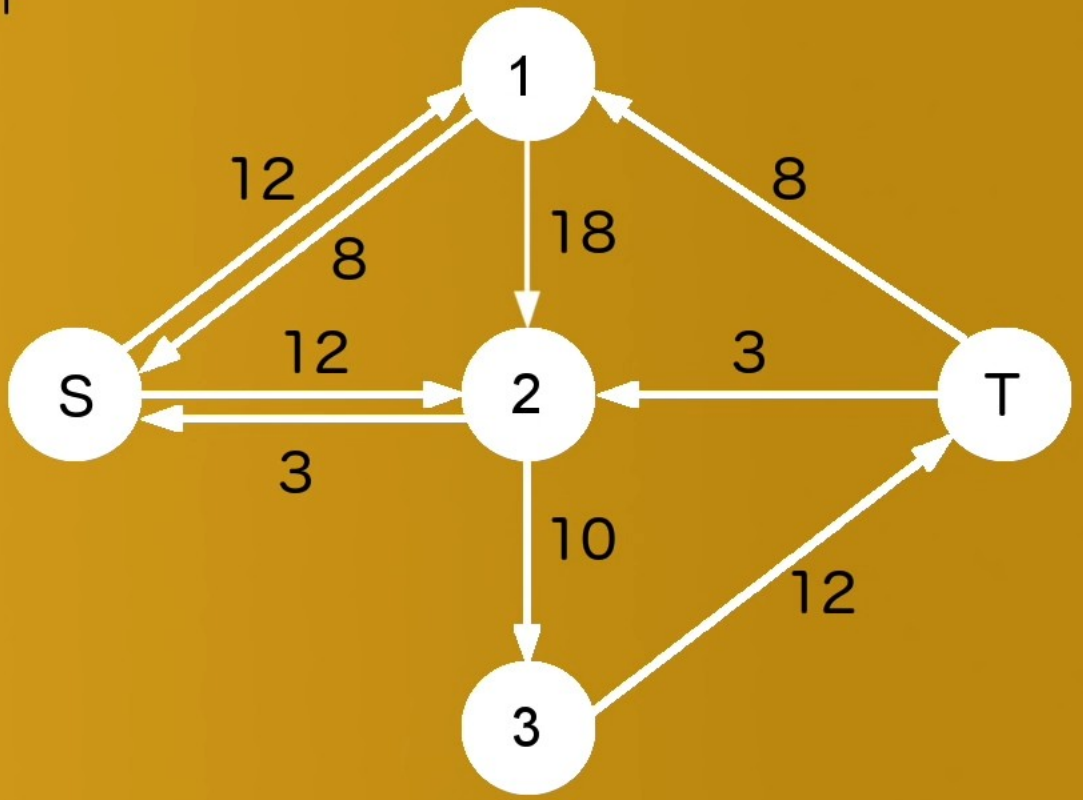
We say an edge is critical if the flow through it is equal to its maximum capacity.

Residual Networks example

f



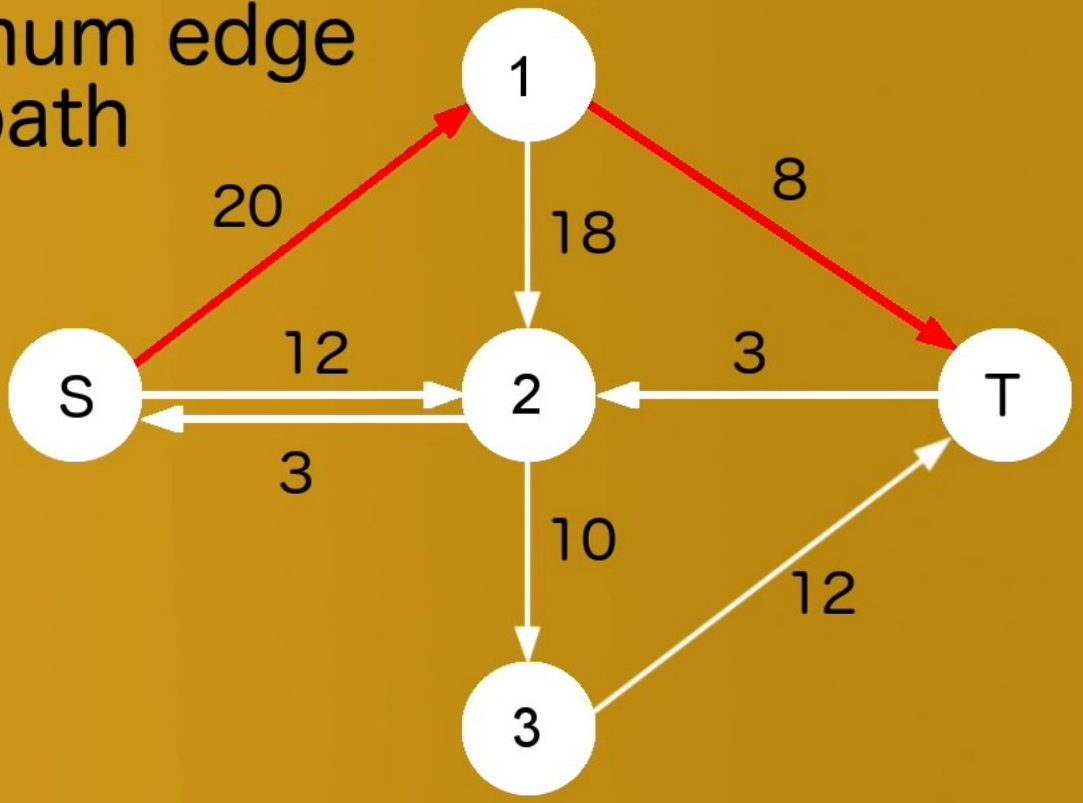
G_f



Augmenting path

Augmenting path is a path from source to sink on residual network

Residual capacity is the minimum edge capacity on the augmenting path



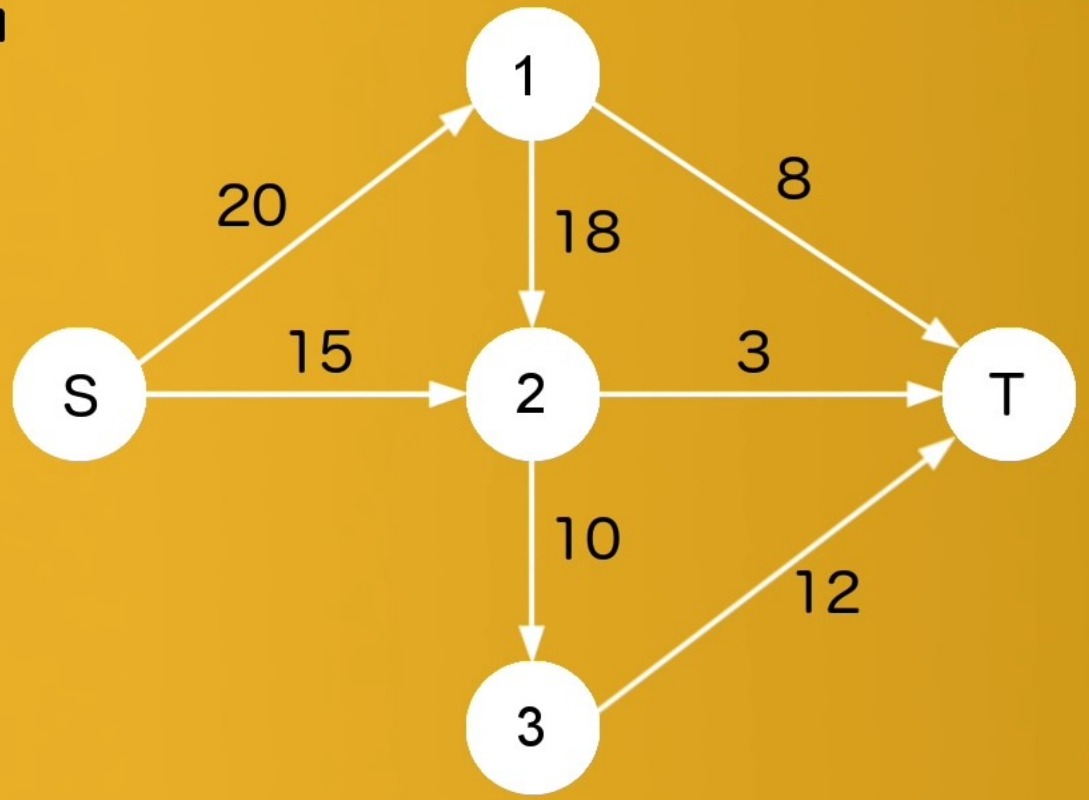
Edmonds-Karp Algorithm

EDMONDS-KARP ALGORITHM(G,s,t)

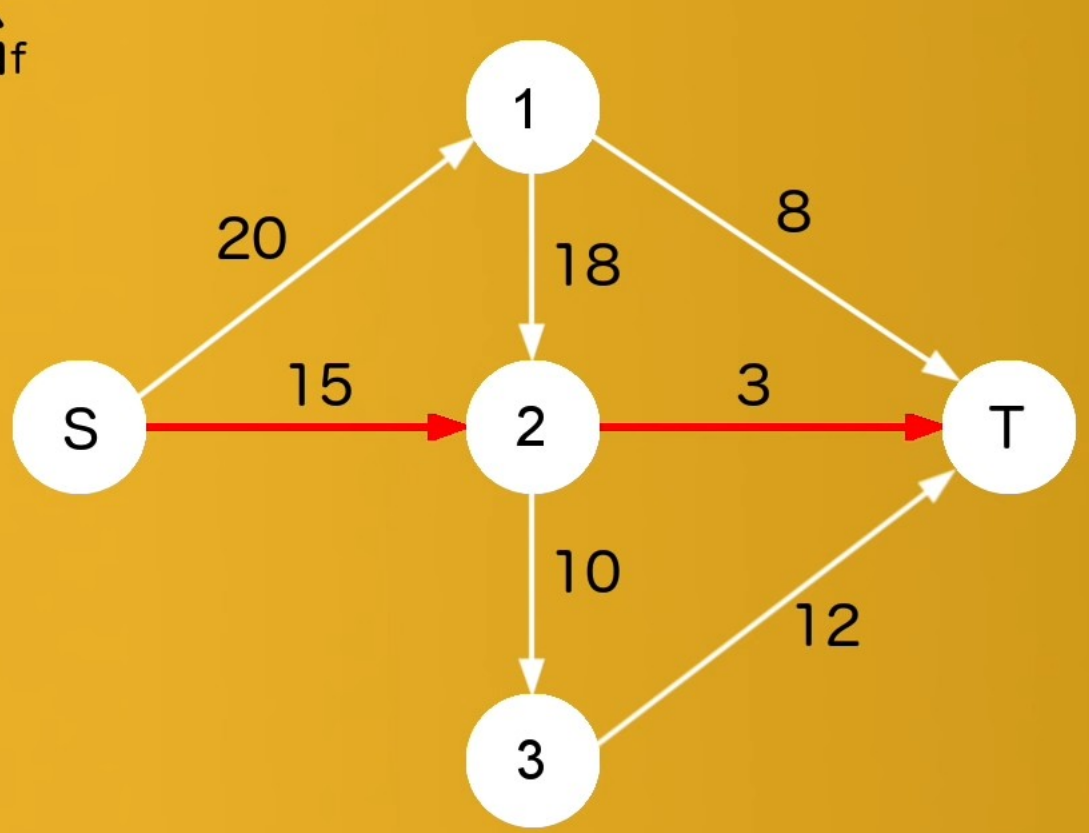
begin
  initialise flow f to 0
  while there exists a shortest augmenting path p in the residual network G_f **do**
    augment flow f along p
  end
end

Example 1

G

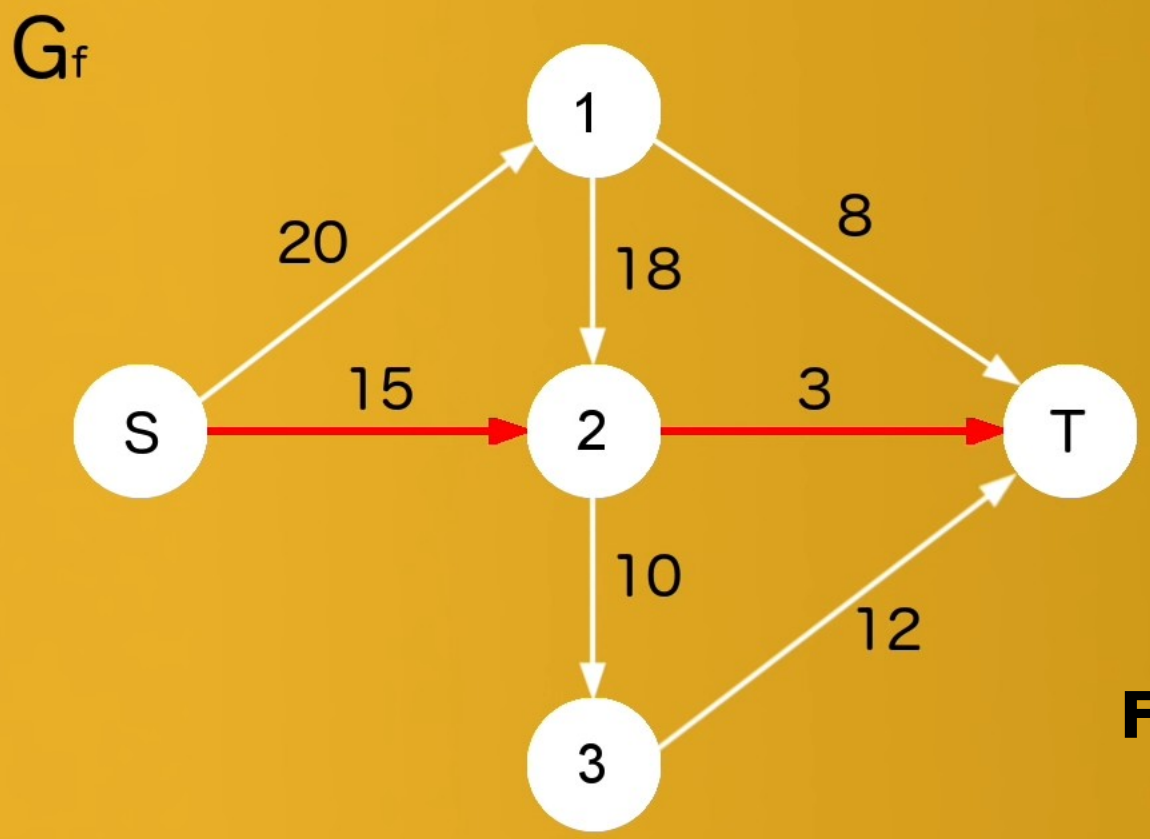


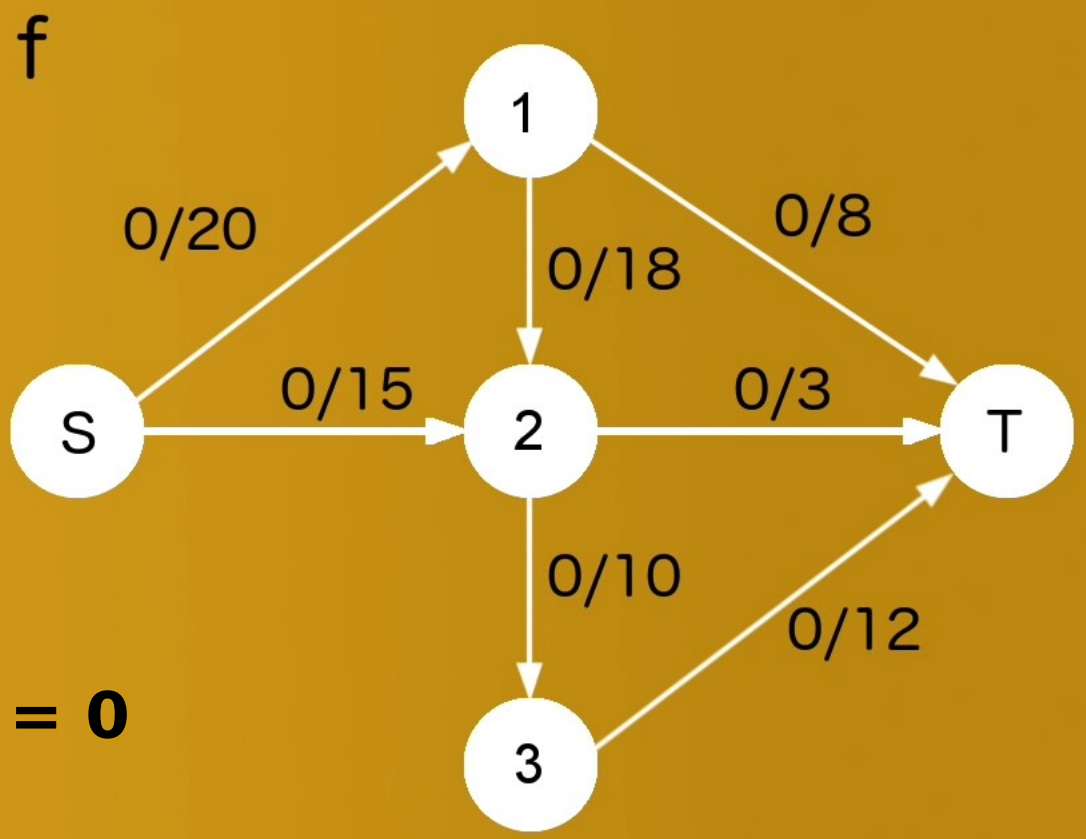
Example 1



.. Choose shortest augmented path using BFS

Example 1

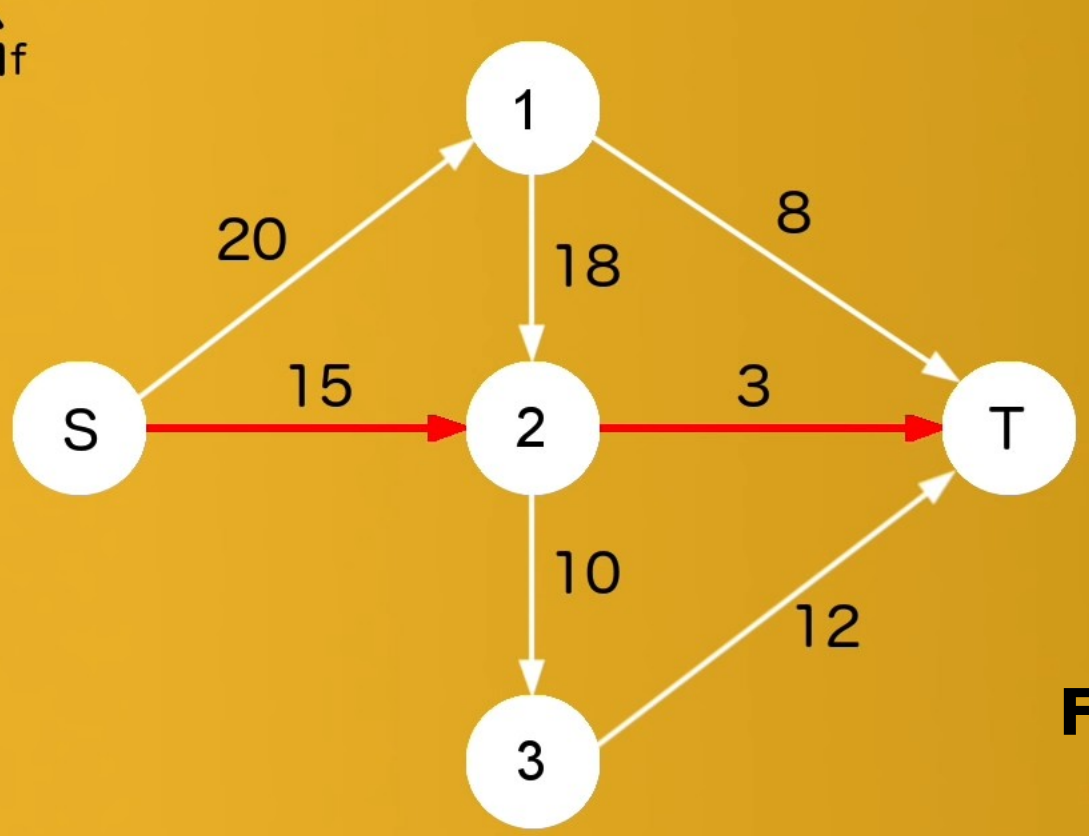


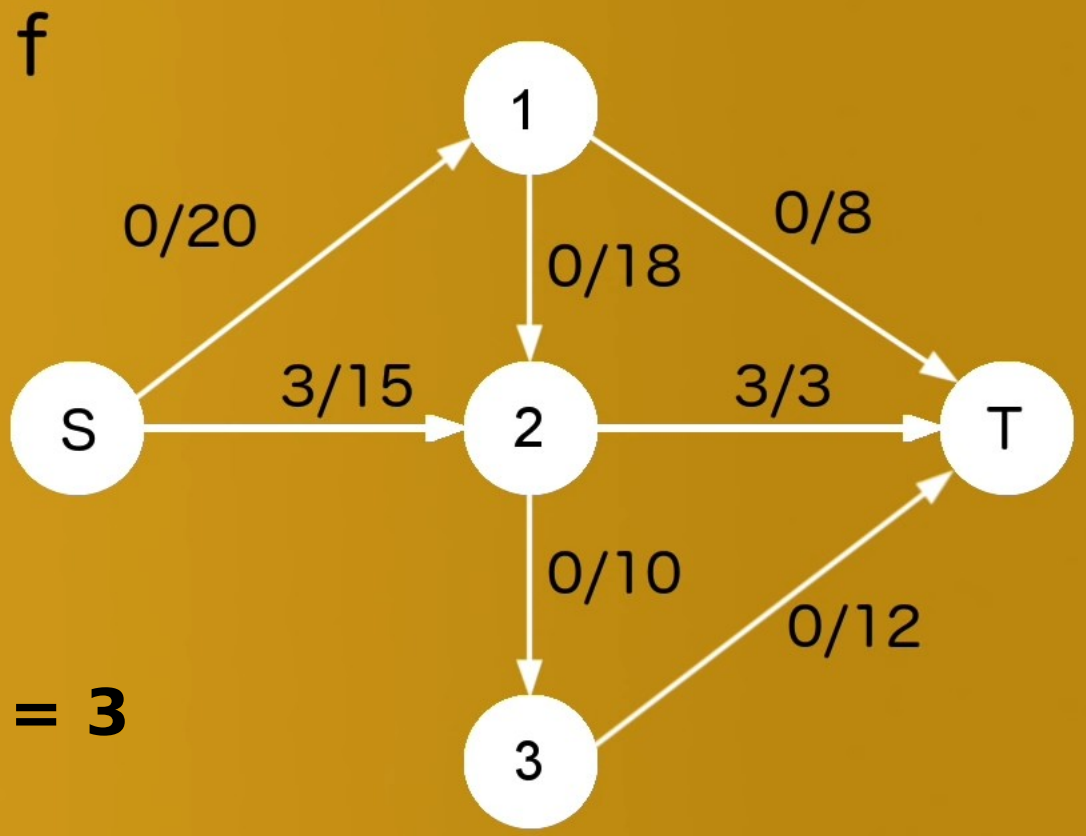


Flow = 0

2. Make *flow = 0* to all edges initiall

Example 1



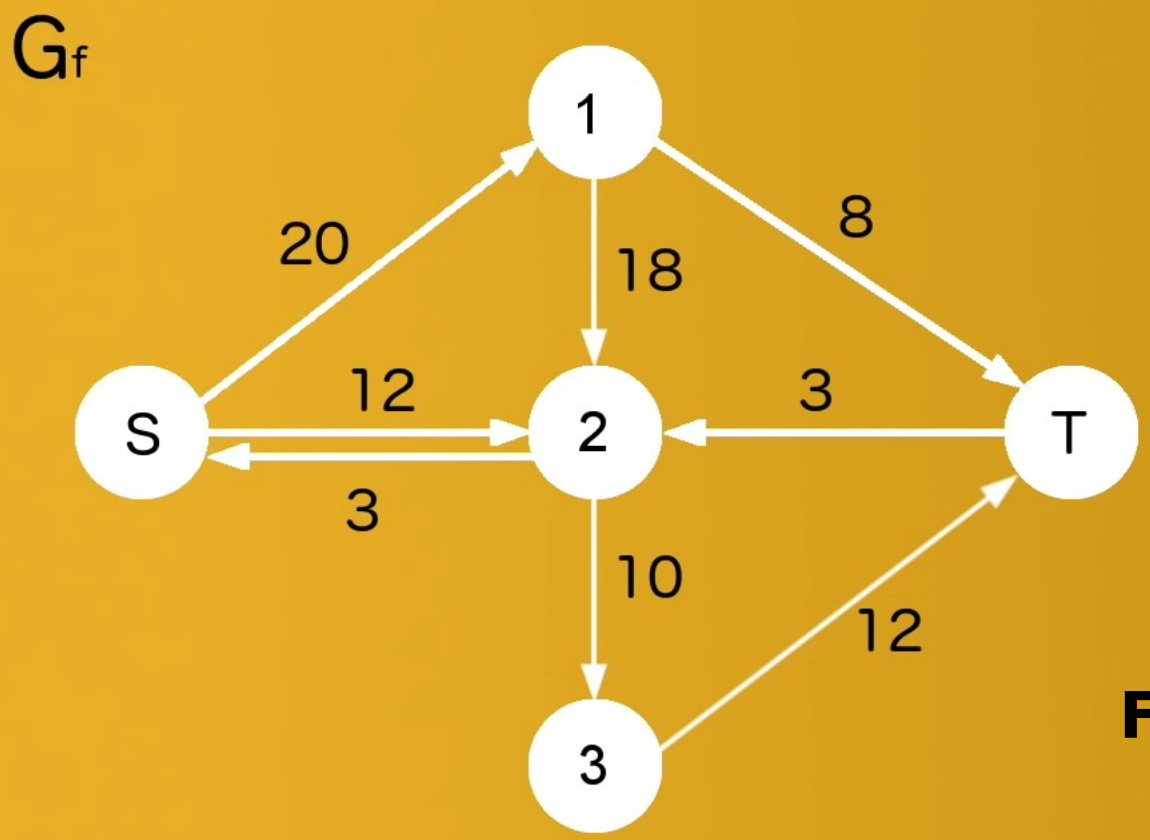


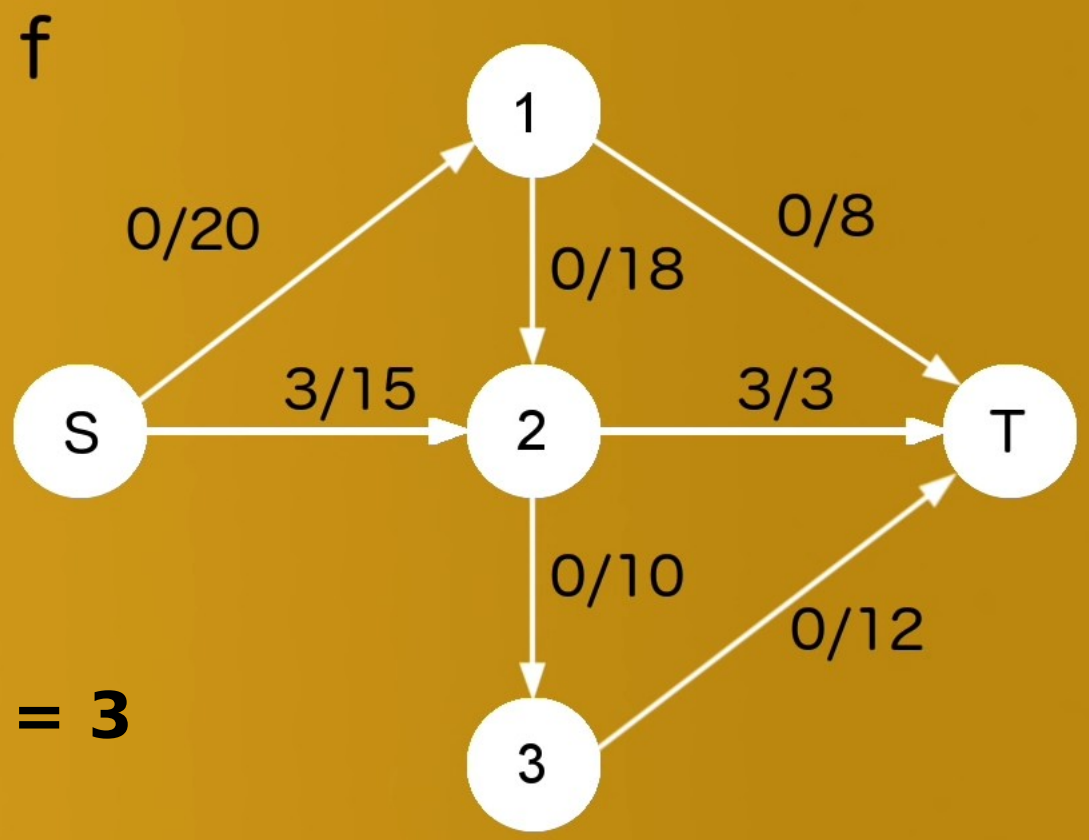
Flow = 3

3. In the chosen augmenting path, bottlene

Make *flow* = *3* to the chosen augmenting pa

Example 1

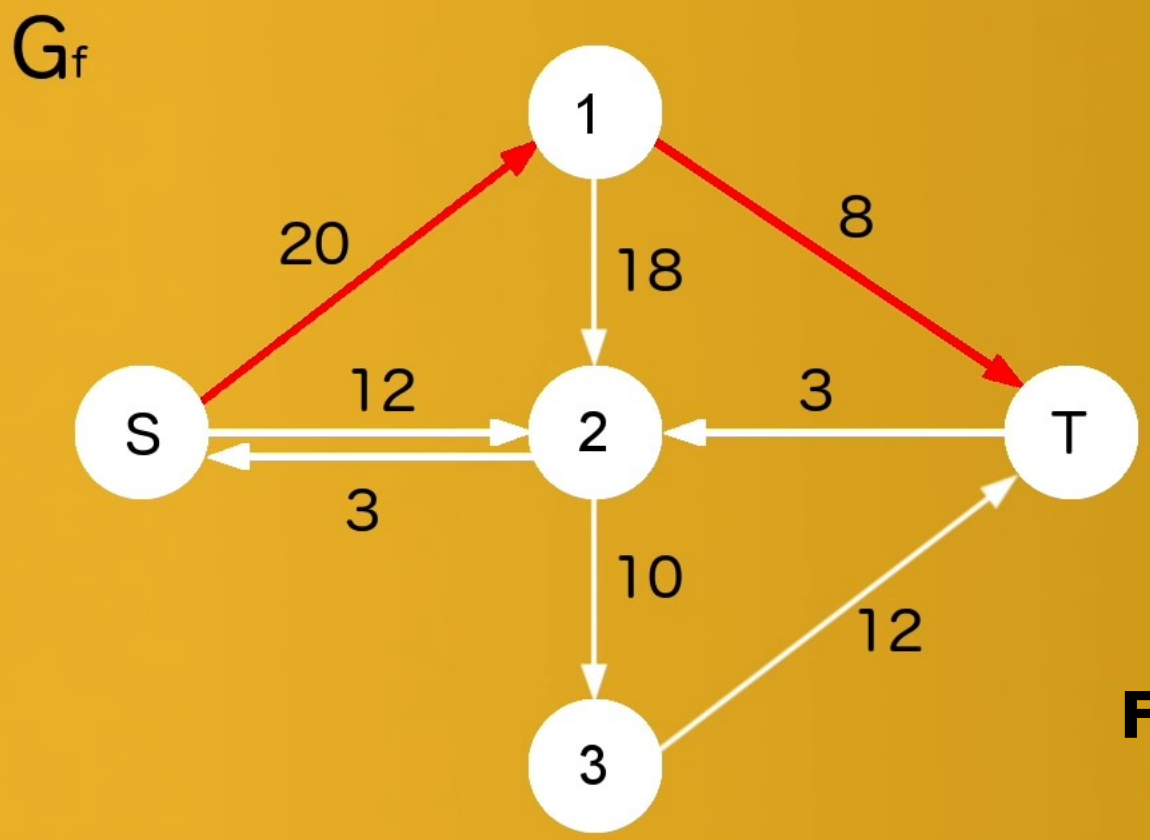


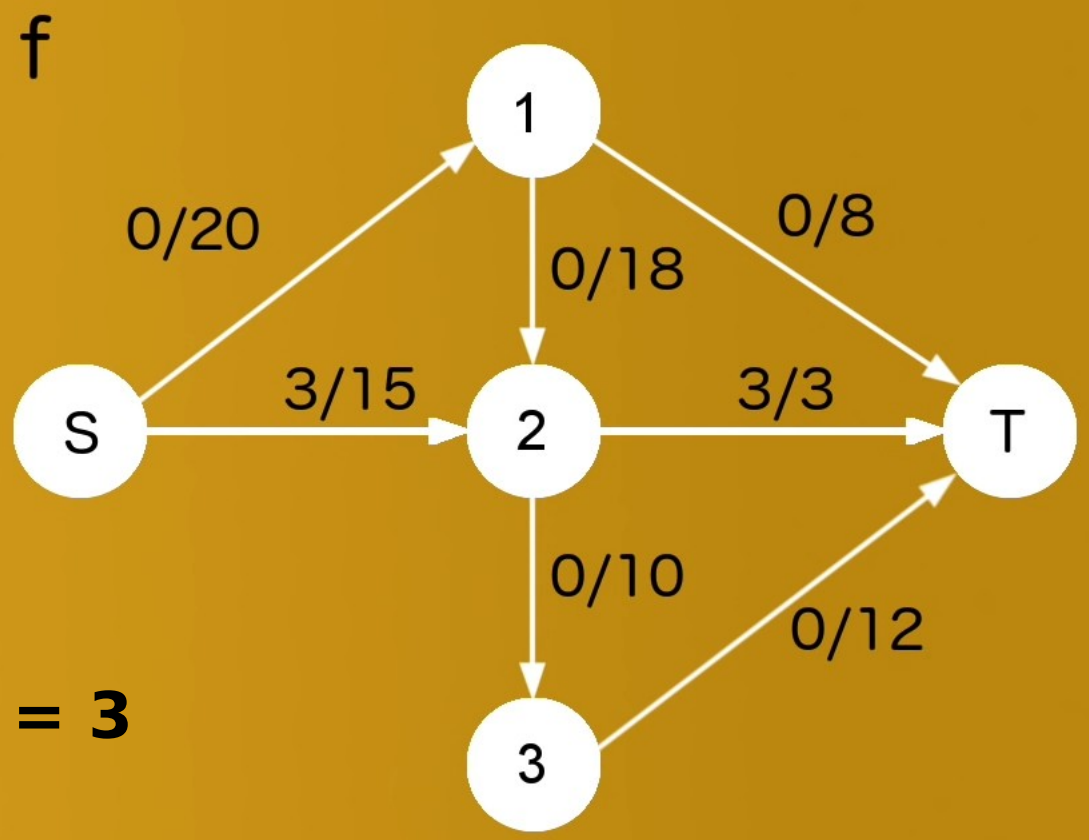


Flow = 3

4. Update the residual Graph

Example 1

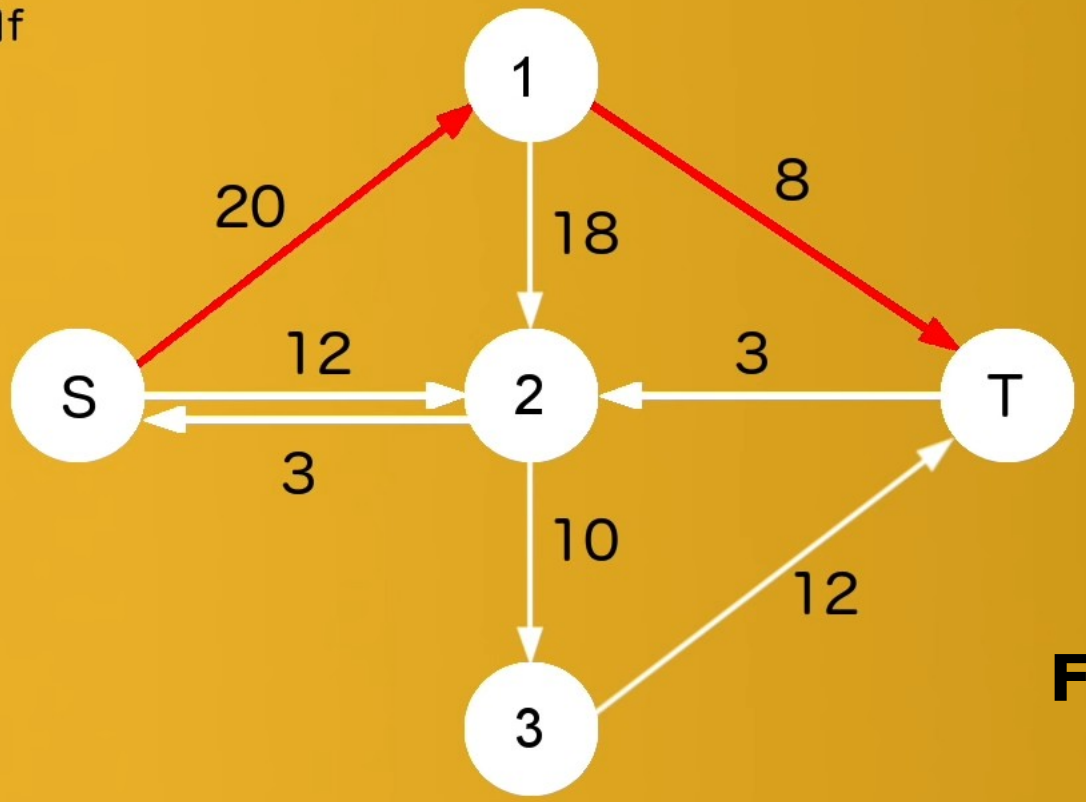


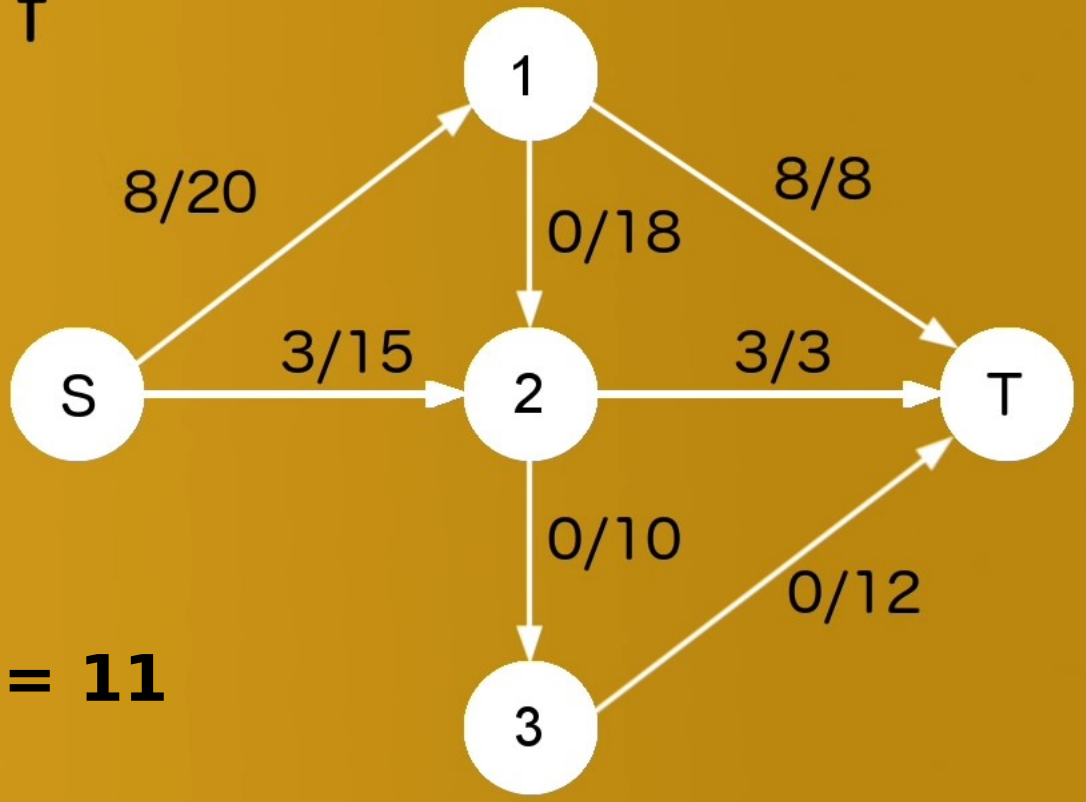


Flow = 3

5. Find next shortest augmented path using BFS

Example 1



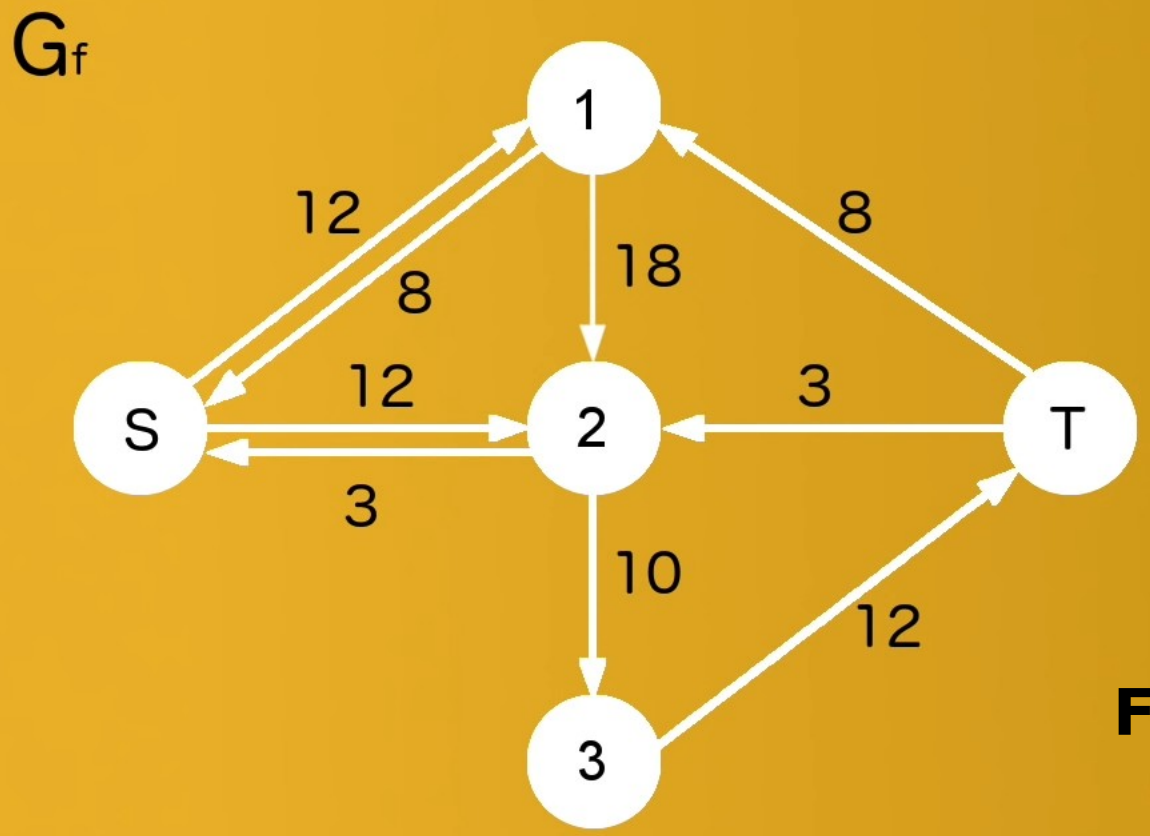


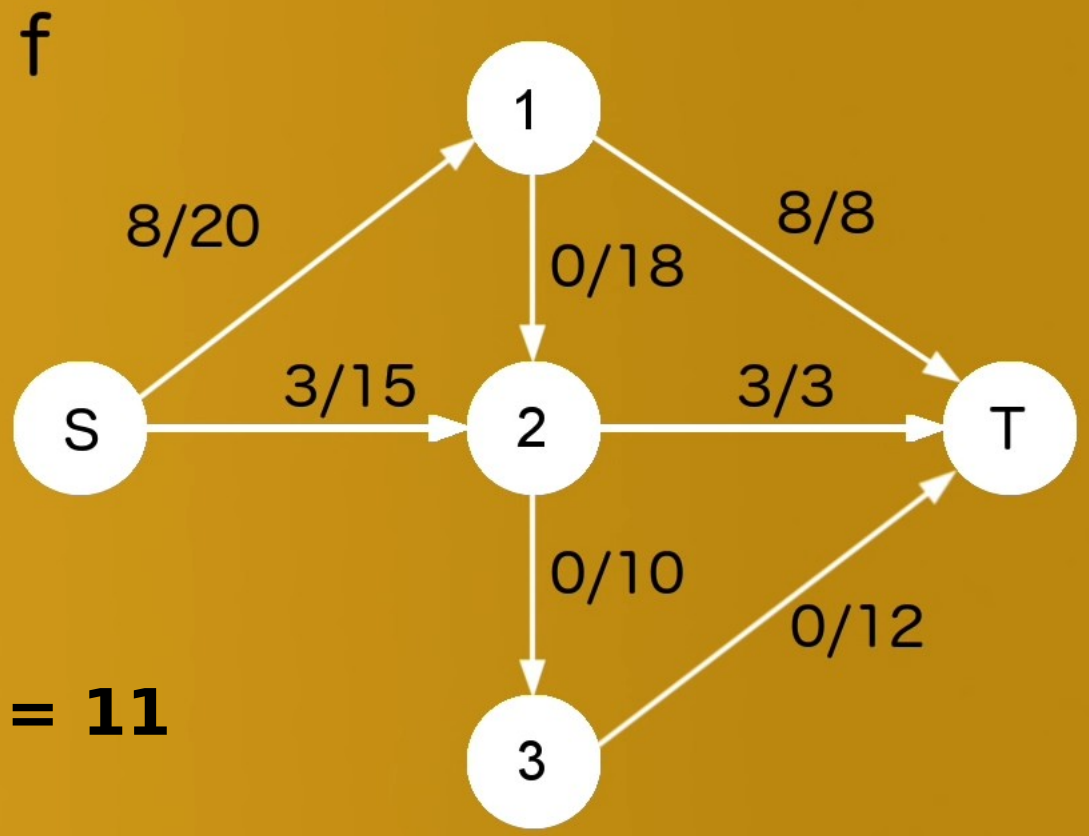
Flow = 11

6. In the chosen augmenting path, bottlene

Make *flow* = *8* to the chosen augmenting pa

Example 1

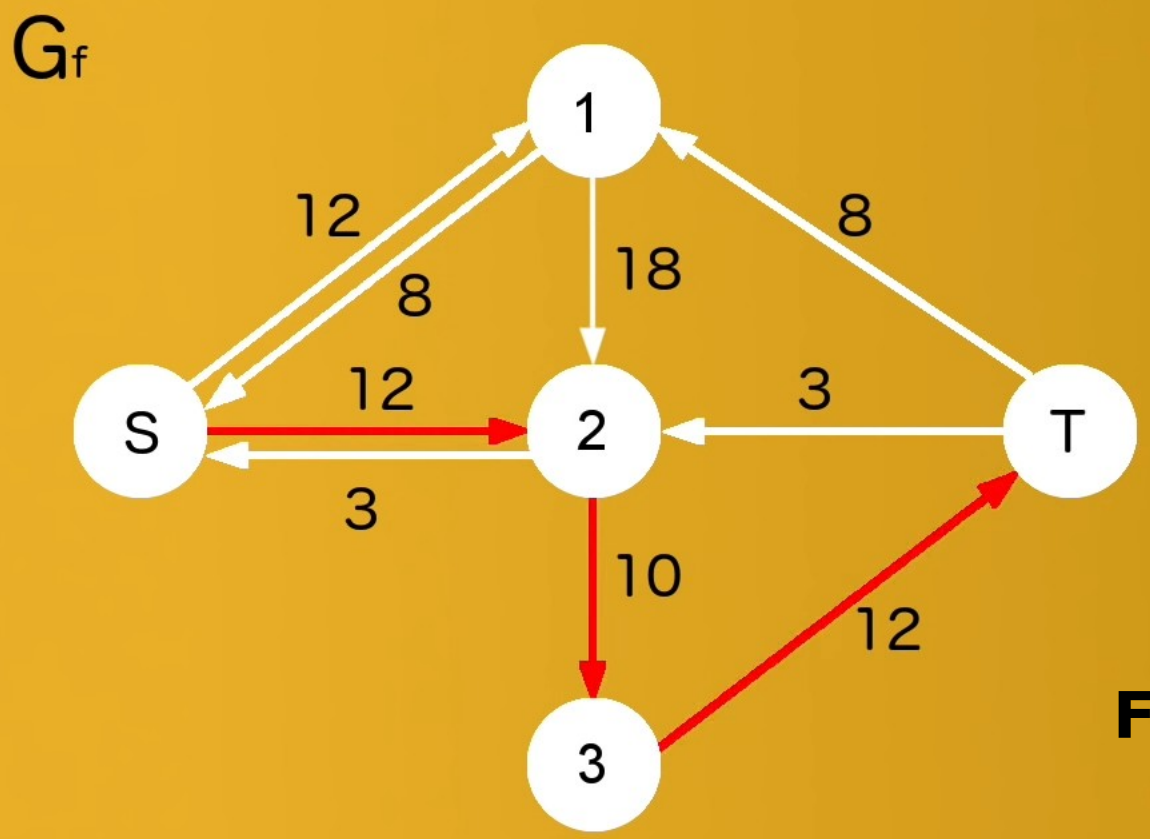


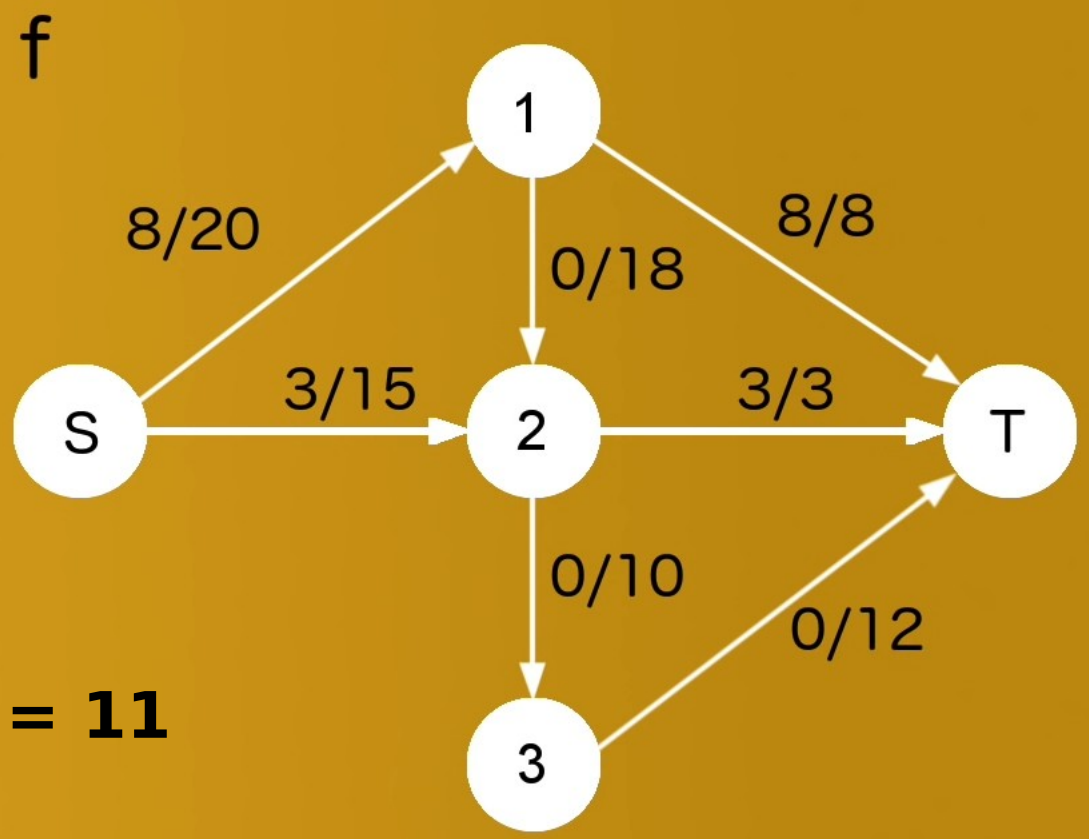


Flow = 11

7. Update the residual Graph

Example 1

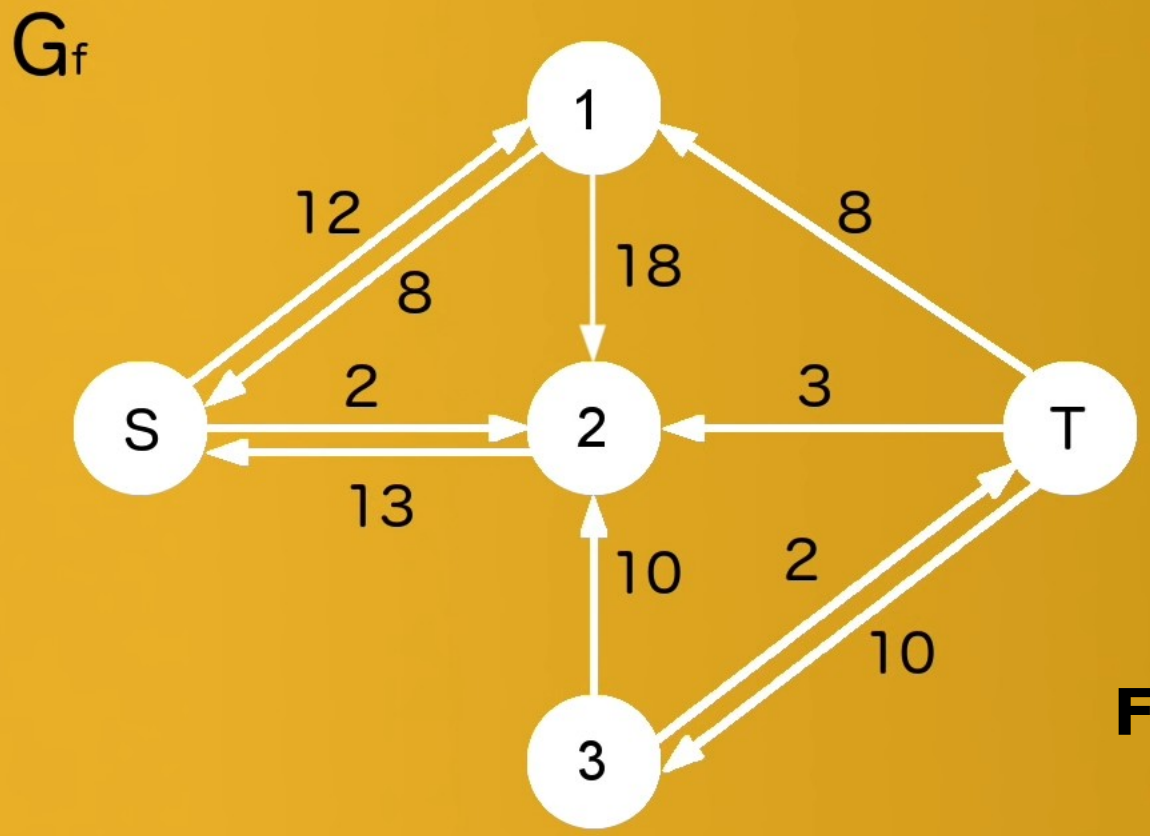


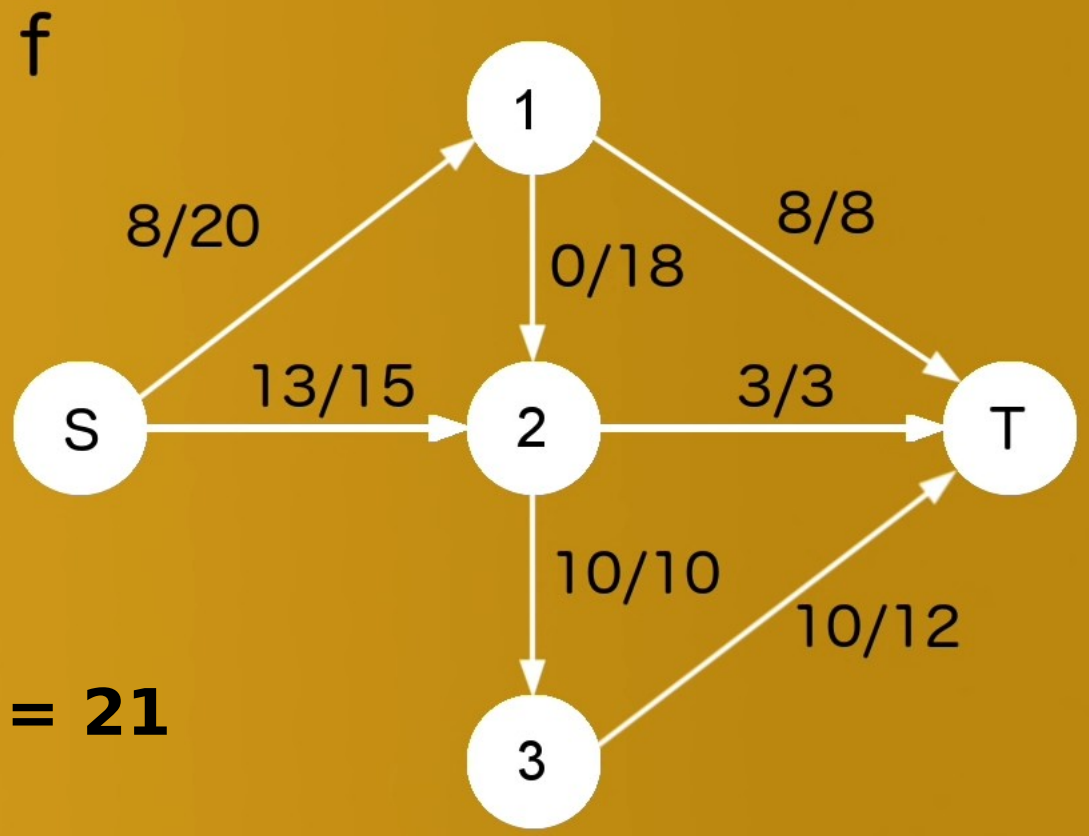


Flow = 11

8. Find next shortest augmented path using BFS

Example 1





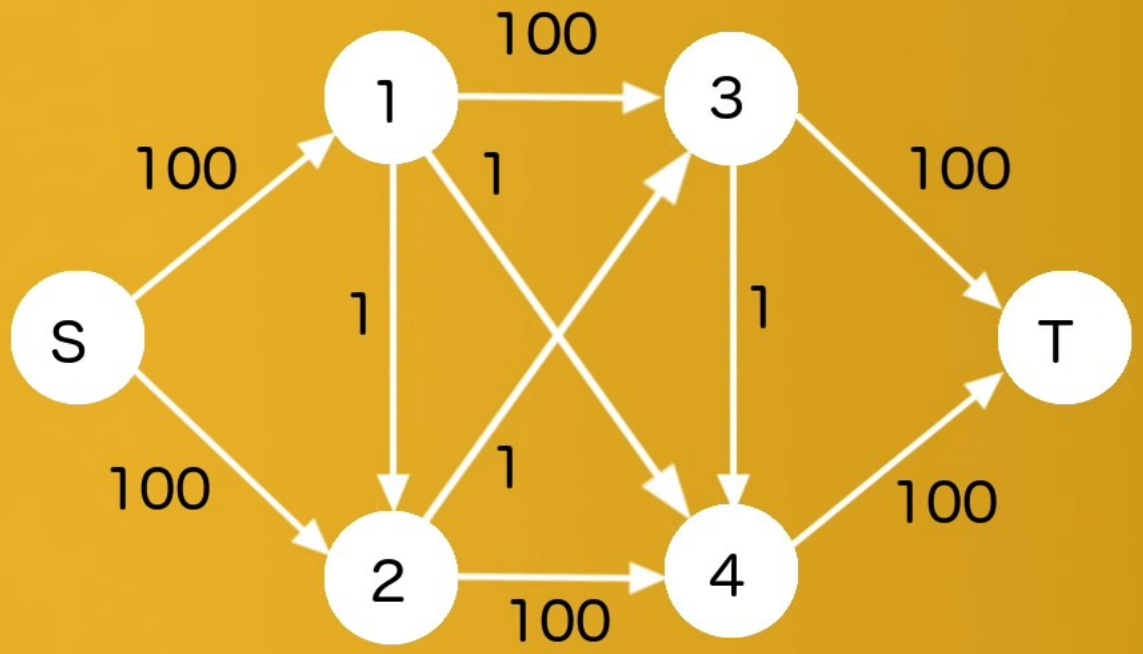
Flow = 21

9. Update the residual Graph

Example 2 - Strength of Algorithm

Choosing augmenting path randomly

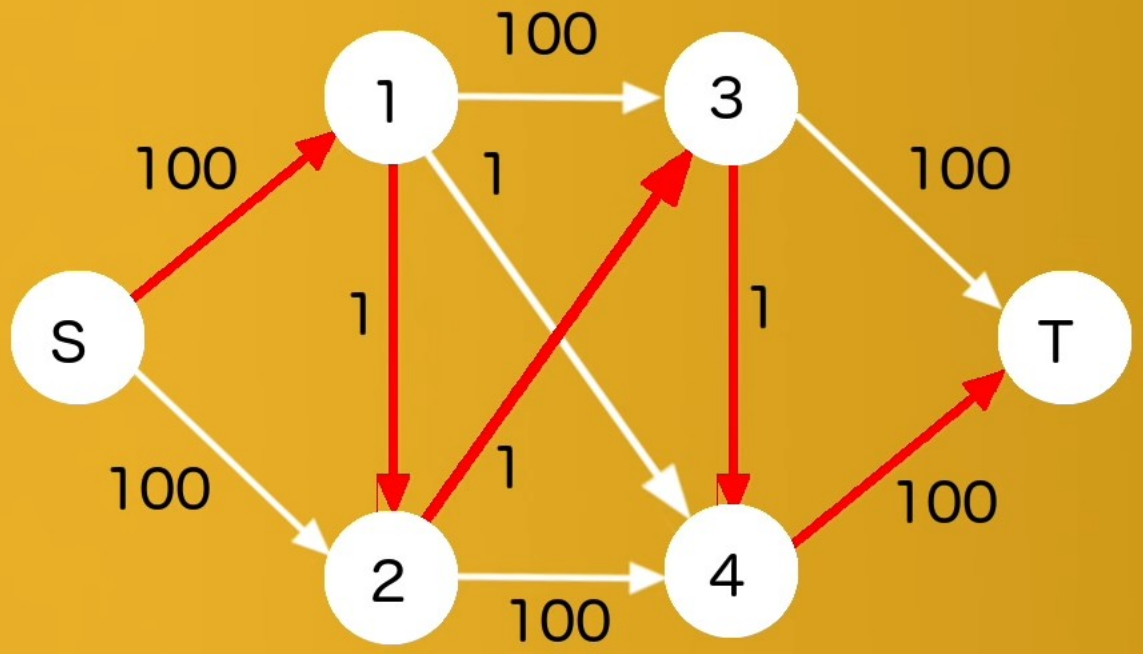
G



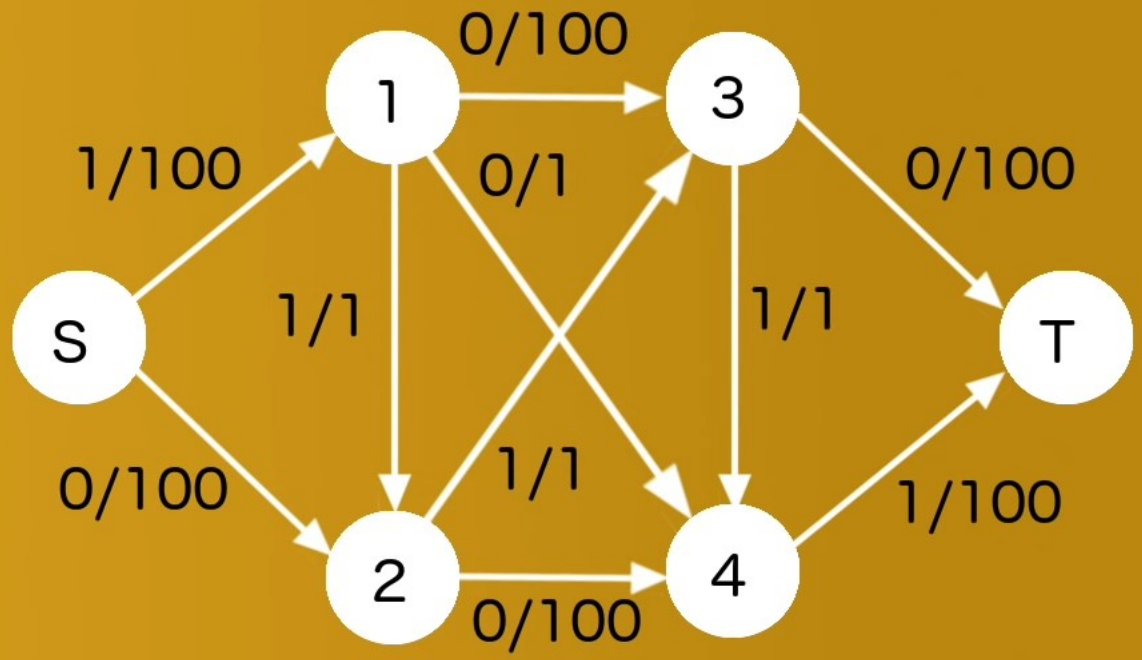
Example 2 - Strength of Algorithm

Choosing augmenting path randomly

G_f



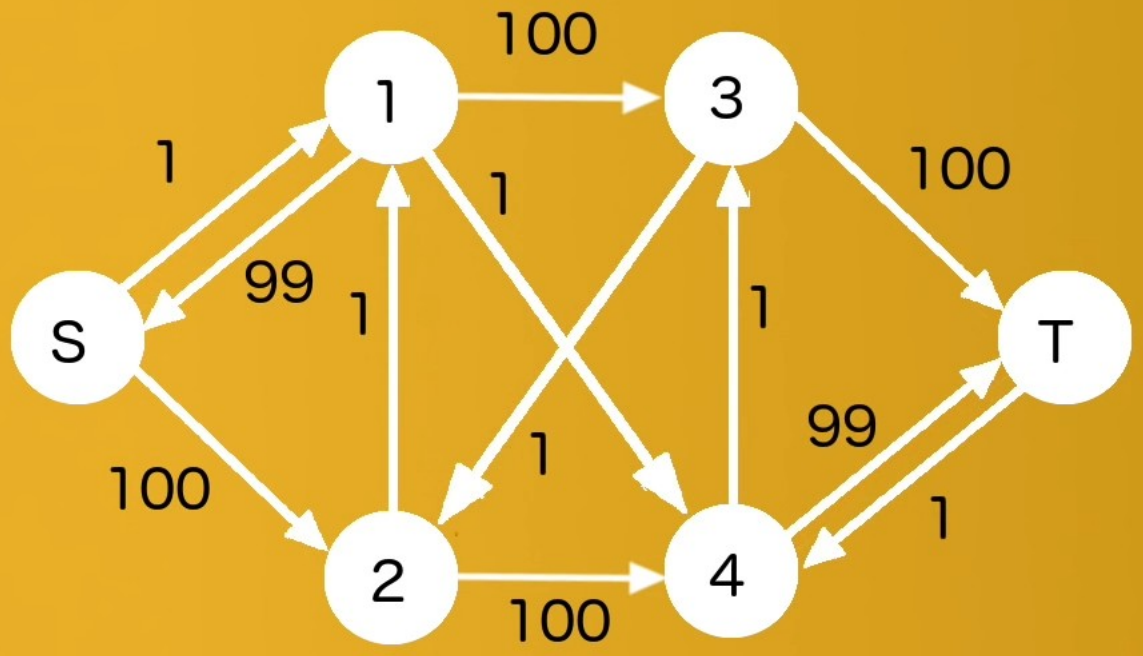
f



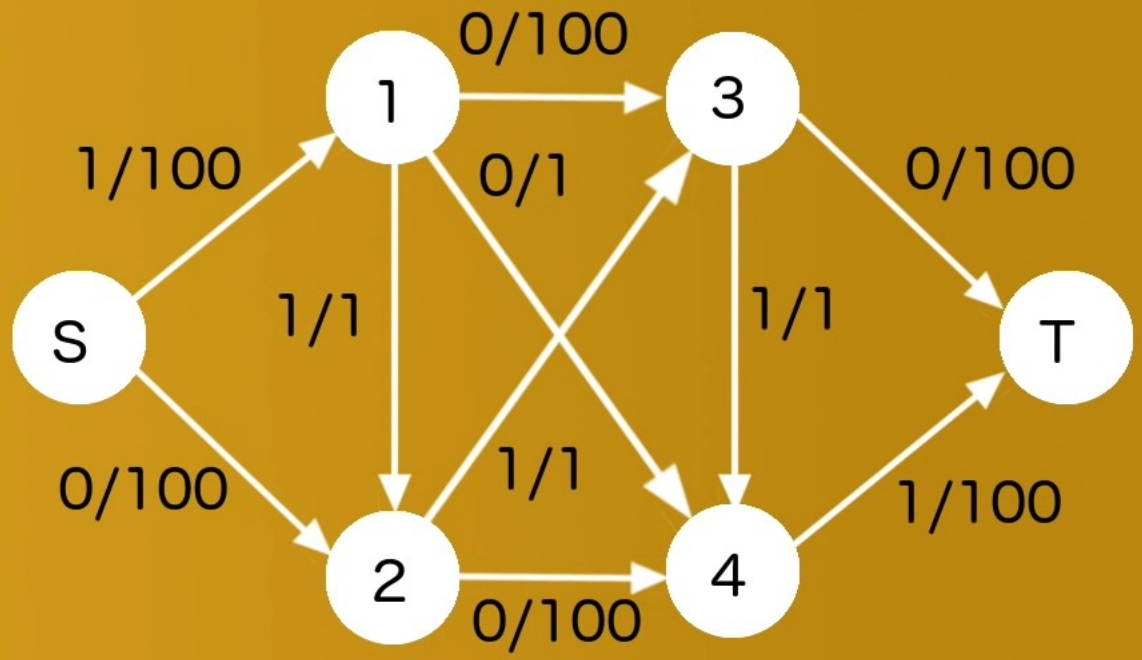
Example 2 - Strength of Algorithm

Choosing augmenting path randomly

G_f



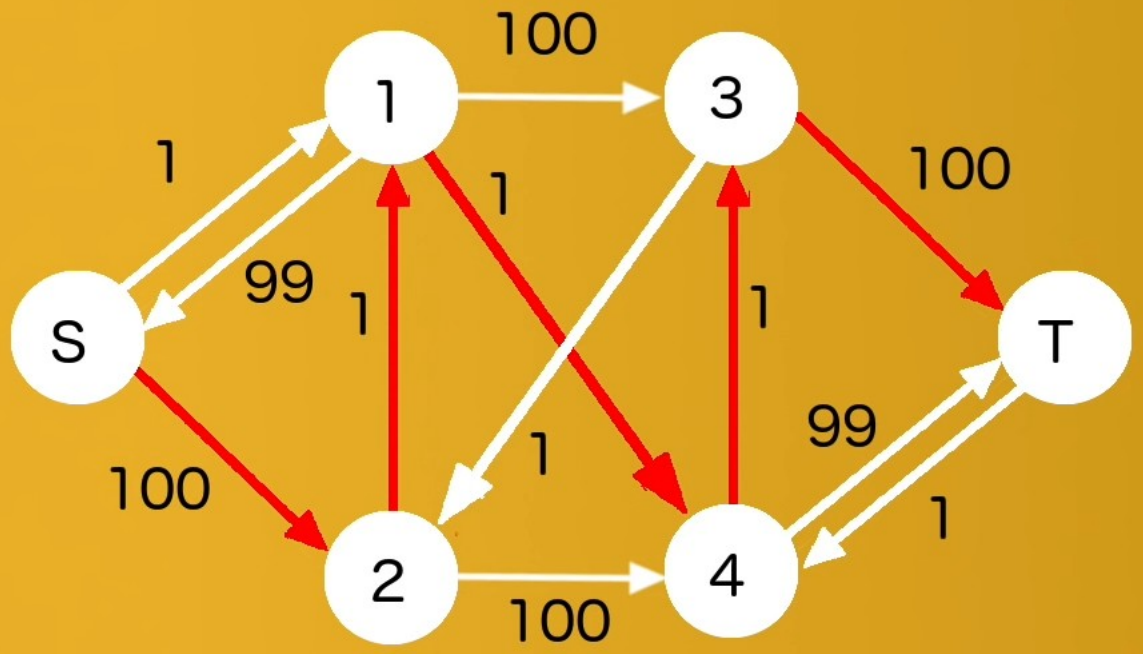
f



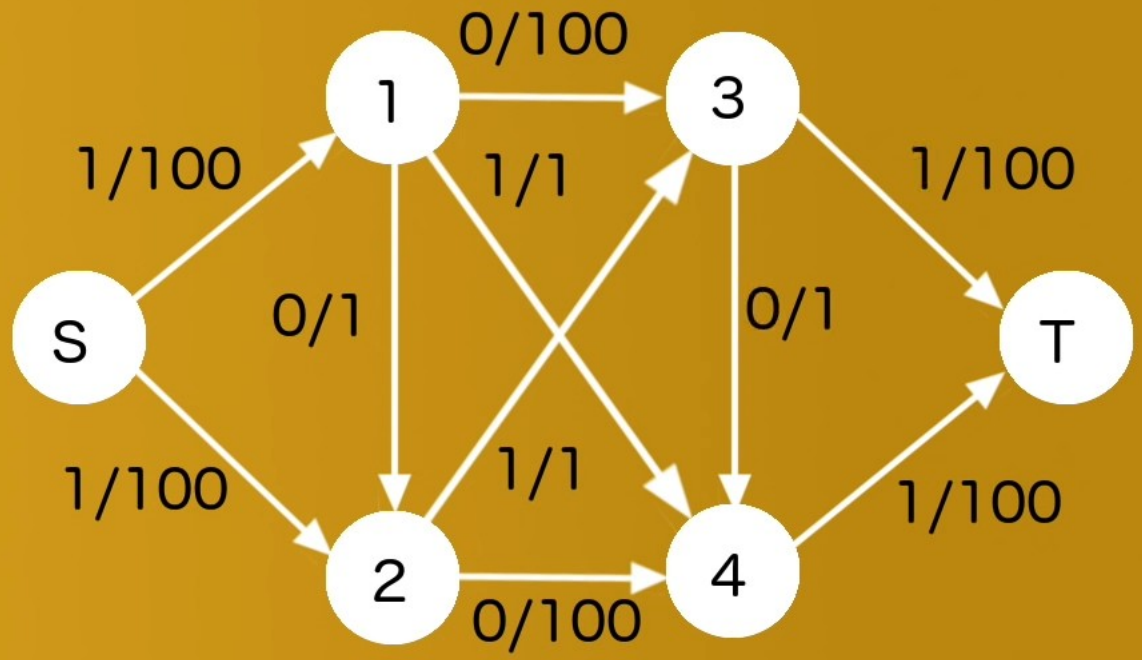
Example 2 - Strength of Algorithm

Choosing augmenting path randomly

G_f



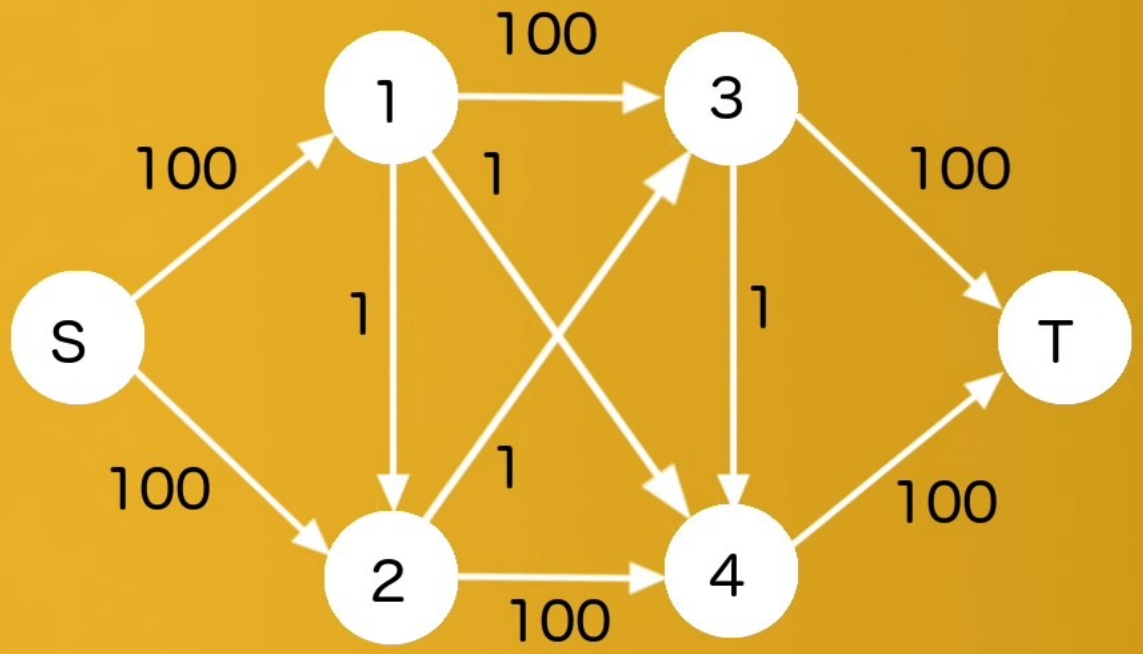
f



Example 2 - Strength of Algorithm

Choosing augmenting path using BFS (as shortest path)

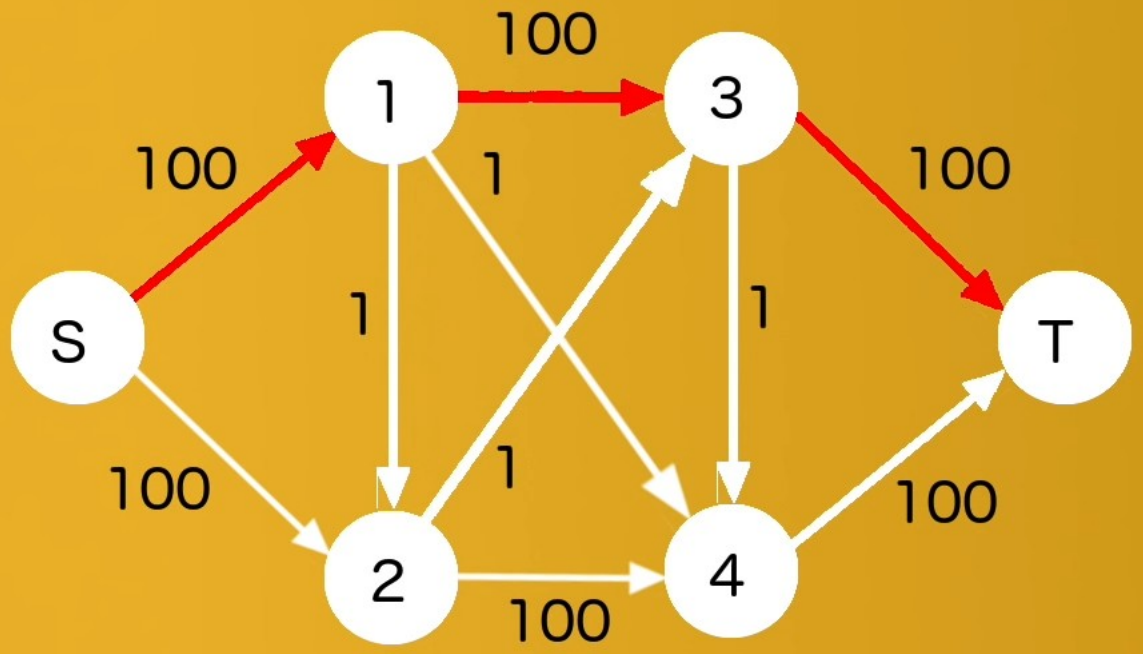
G



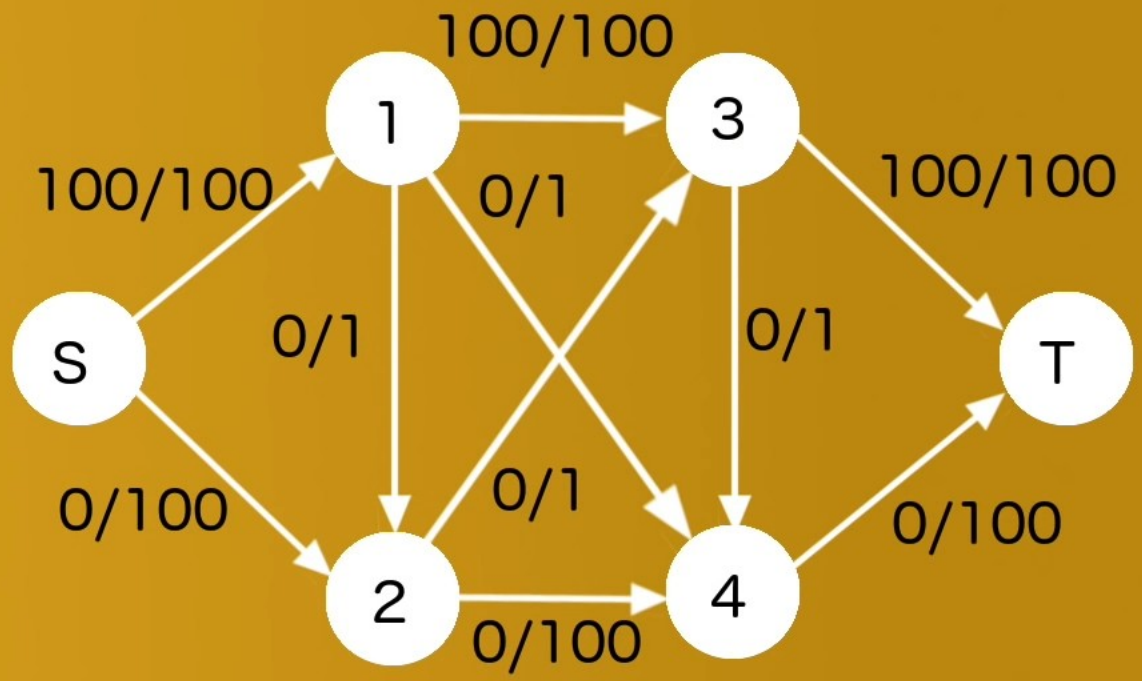
Example 2 - Strength of Algorithm

Choosing augmenting path using BFS (as shortest path)

G_f



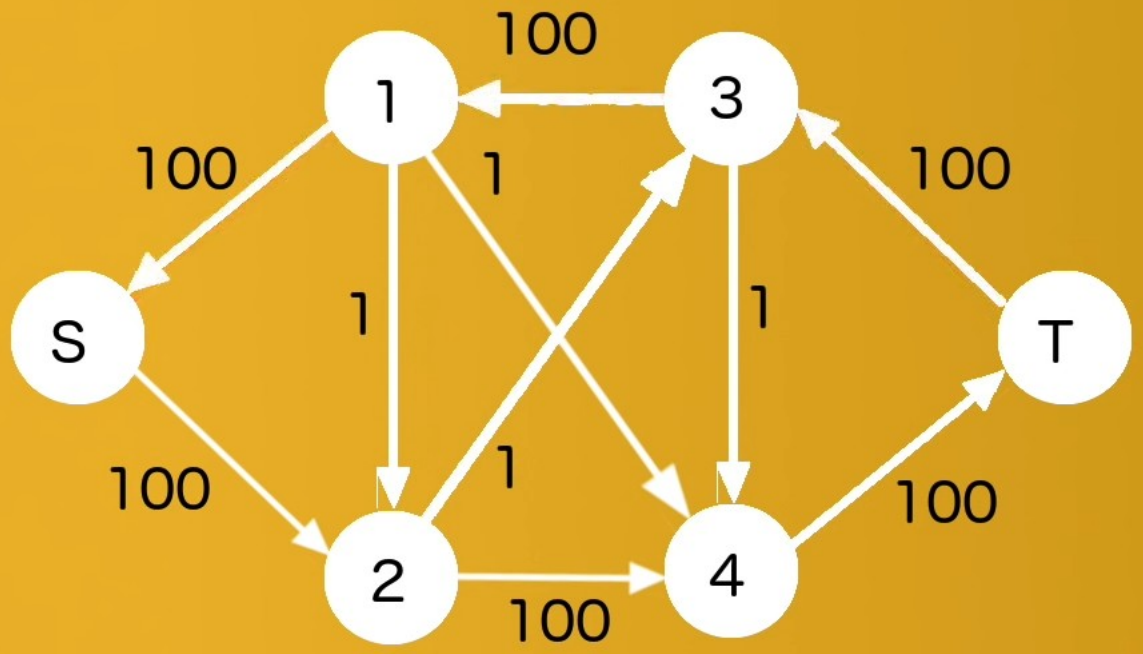
f



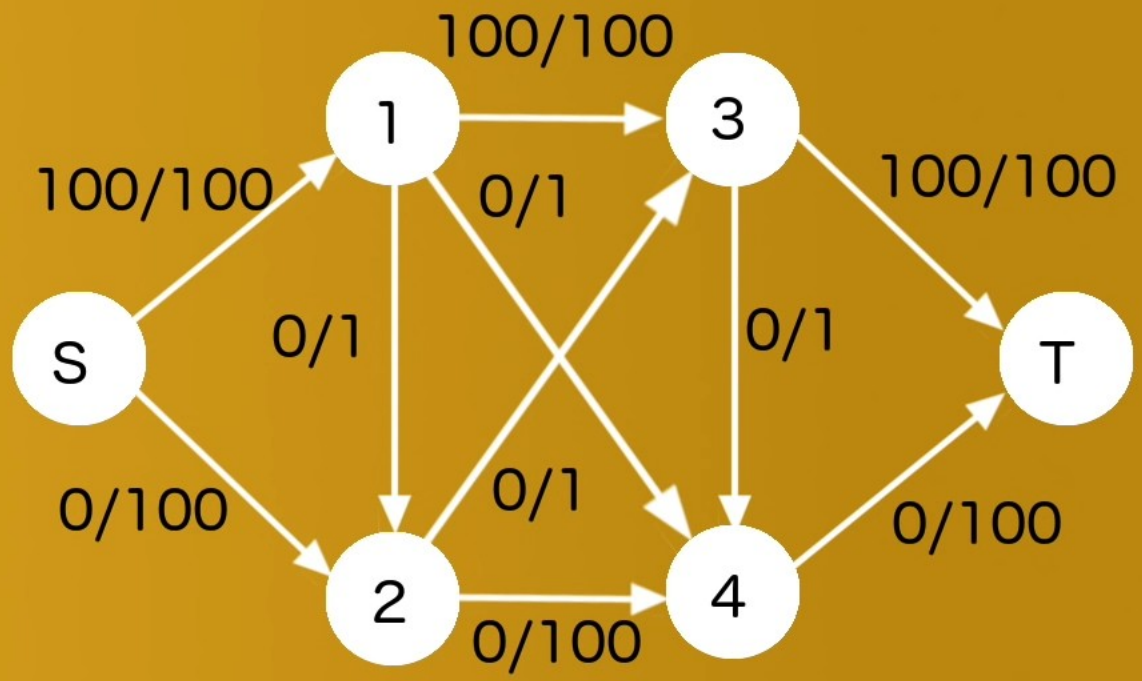
Example 2 - Strength of Algorithm

Choosing augmenting path using BFS (as shortest path)

G_f



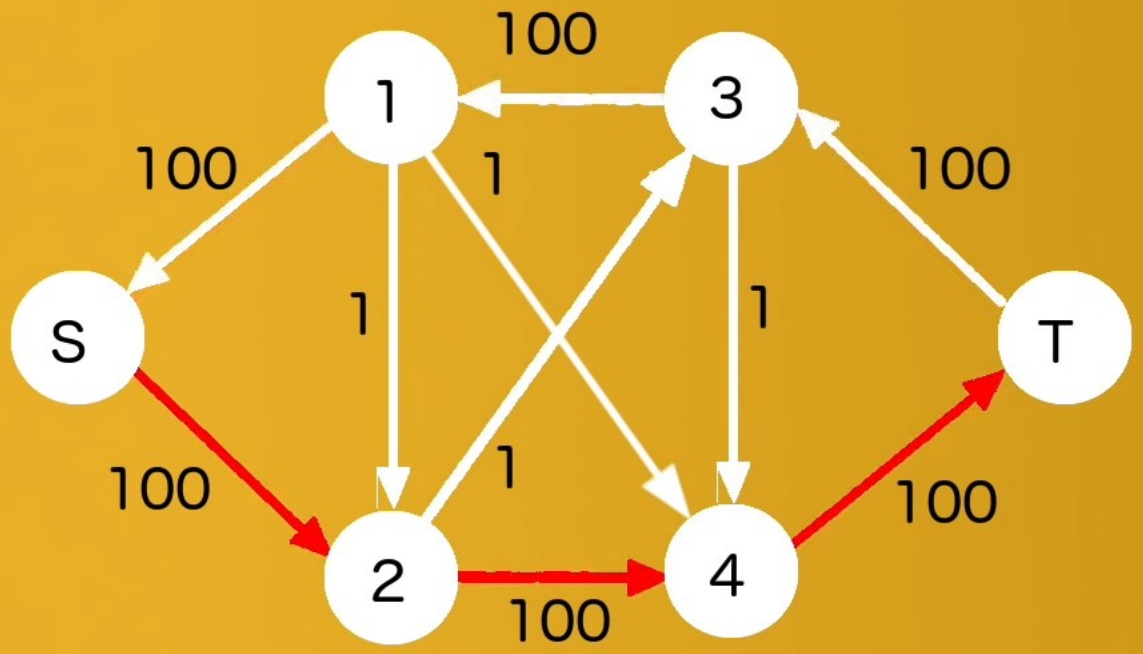
f



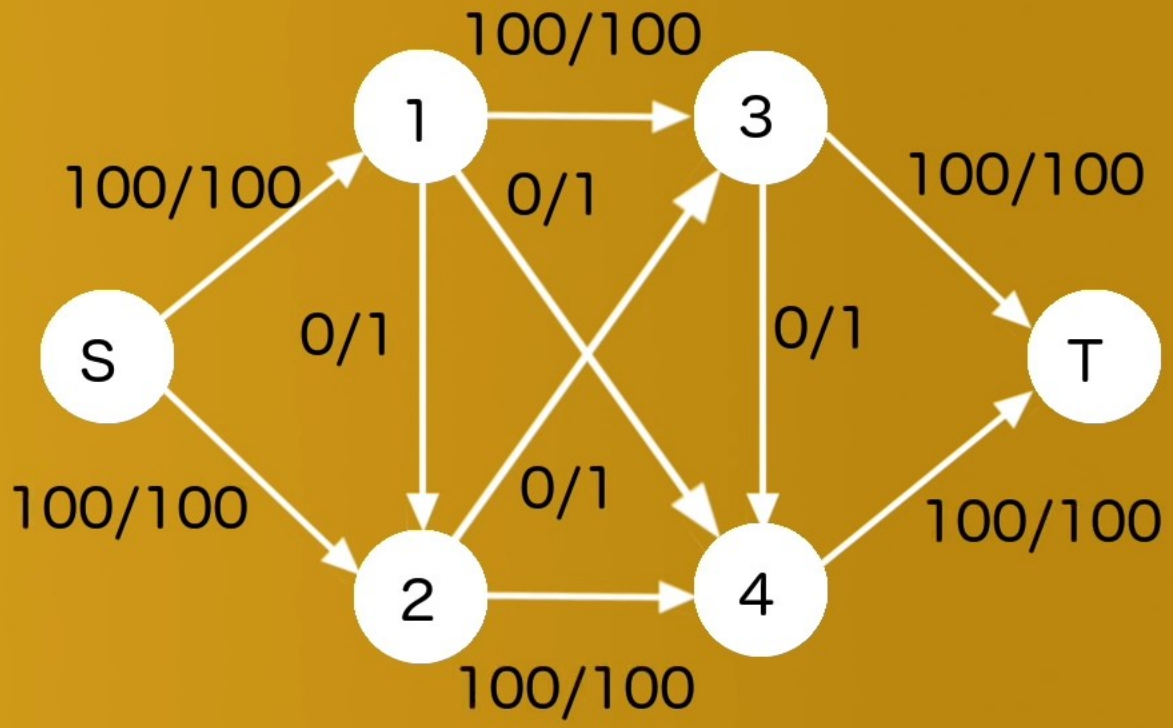
Example 2 - Strength of Algorithm

Choosing augmenting path using BFS (as shortest path)

G_f



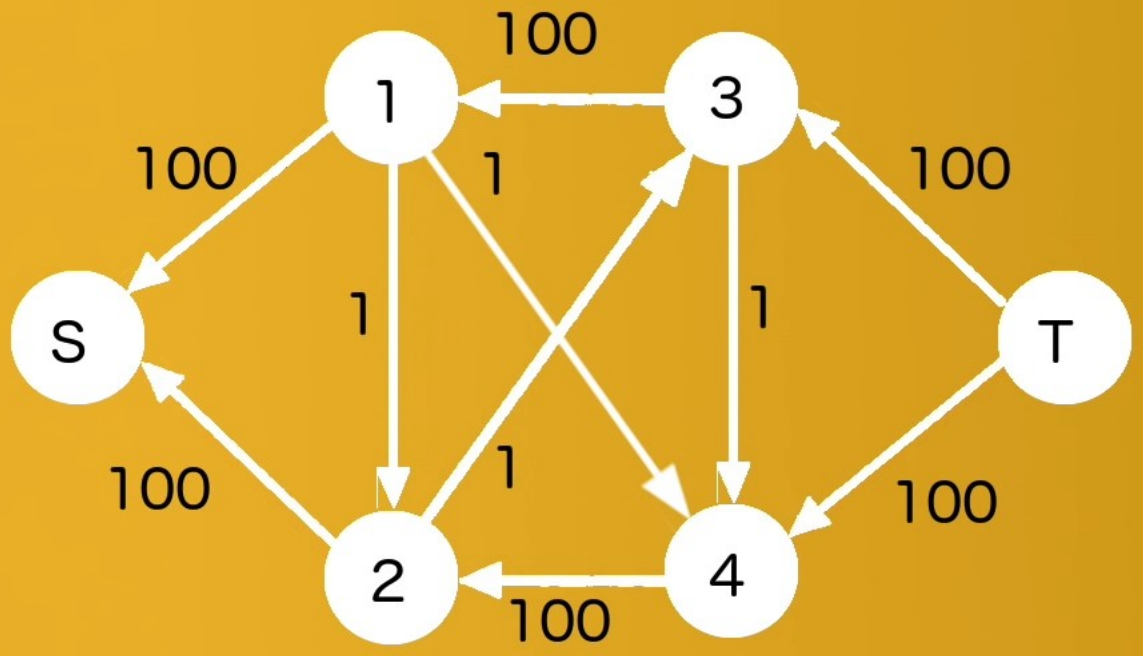
f



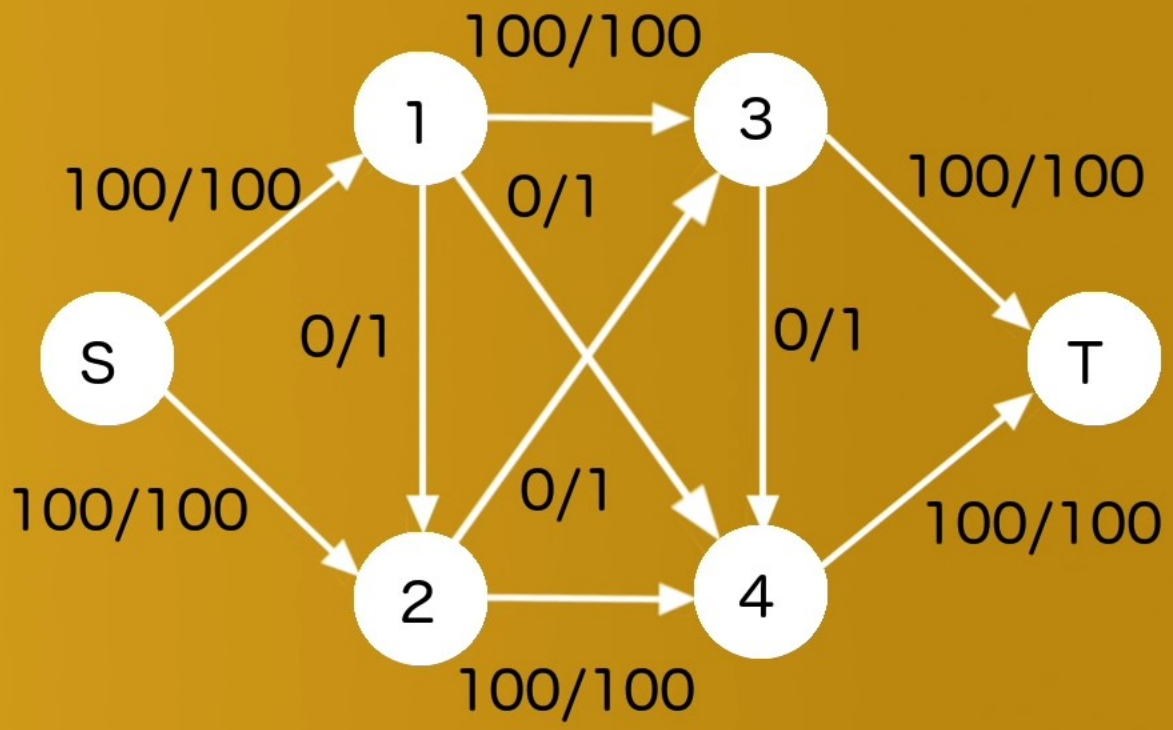
Example 2 - Strength of Algorithm

Choosing augmenting path using BFS (as shortest path)

G_f



f



Time Complexity

- Every iteration of the while loop, the shortest path algorithm must be performed on the residual network. This takes O(E) time using breadth first search.

```
EDMONDS-KARP ALGORITHM(G,s,t)

begin
   initialise flow f to 0
   while there exists a shortest augmenting path p in
   the residual network Gf do
      augment flow f along p
   end
end
```

Time Complexity

- What is the maximum amount of times the while loop can be executed?
- At every iteration one of the edges must become critical.
- From the time it is critical to the time it is critical again the distance to it must increase.
- It can increase at most V times as this is the longest path possible.
- This can happen to all E edges. Therefore, in the worst case E edges can become critical V times. So the loop can run VE times max.

Time Complexity

Therefore, running a loop that takes O(E) time at most O(VE) times means the algorithm has a runtime of

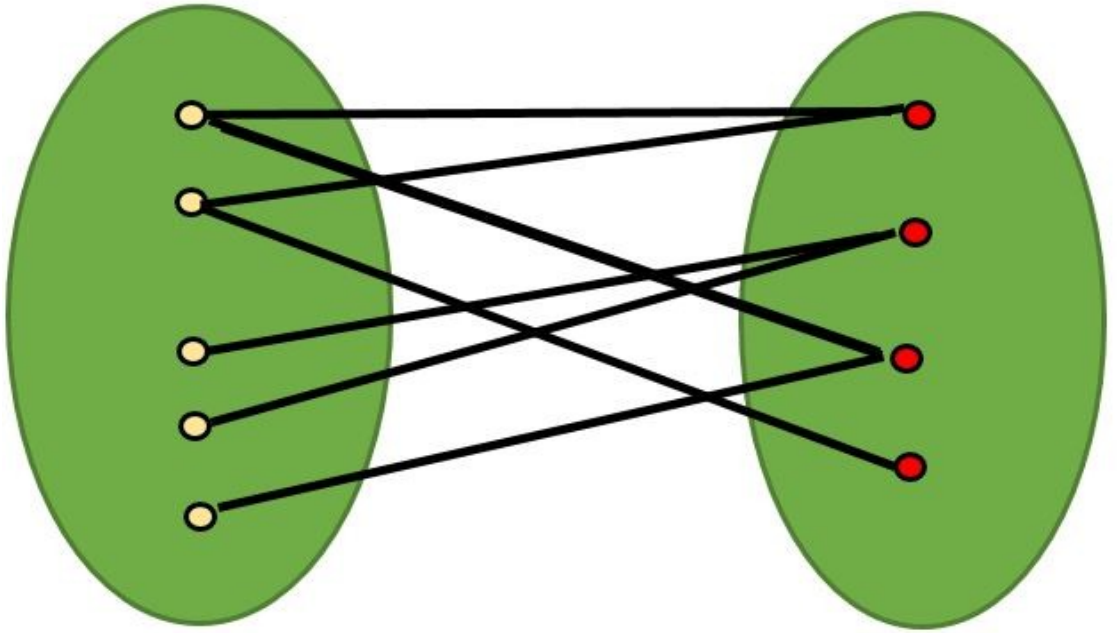
$$O(VE^2)$$

Maximum Bipartite Matching

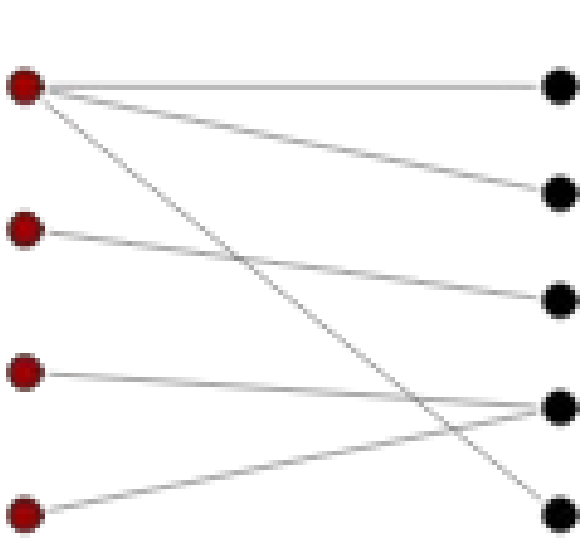
(An application of Max-Flow Algorithm)

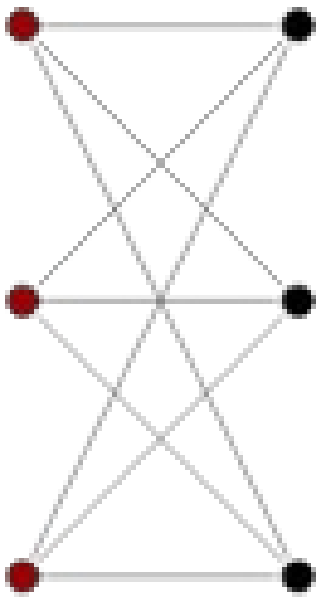
Bipartite Graph

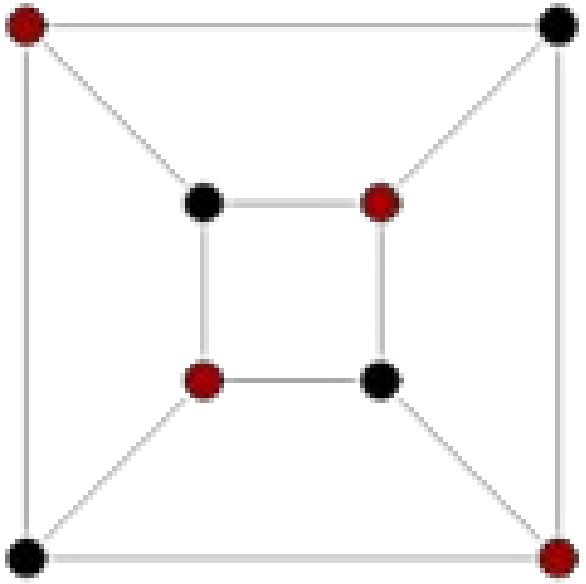
A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.



More examples of Bipartite Graph







Application of Bipartite Graph

- **Matching problems:** Bipartite graphs are commonly used to model matching problems, such as matching job seekers with job vacancies or assigning students to project supervisors. The bipartite structure allows for a natural way to match vertices from one set to vertices in the other set.
- **Social networks:** Bipartite graphs, where the nodes in one set represent users and the nodes in the other set reflect interests, groups, or communities, can be used to simulate social networks. The bipartite form makes it simple to analyze the connections between users and interests.
- **Web Search engine:** The query and click-through data of a search engine can be defined using a bipartite graph, where the two sets of vertices represent queries and web pages.

Maximum Bipartite Matching

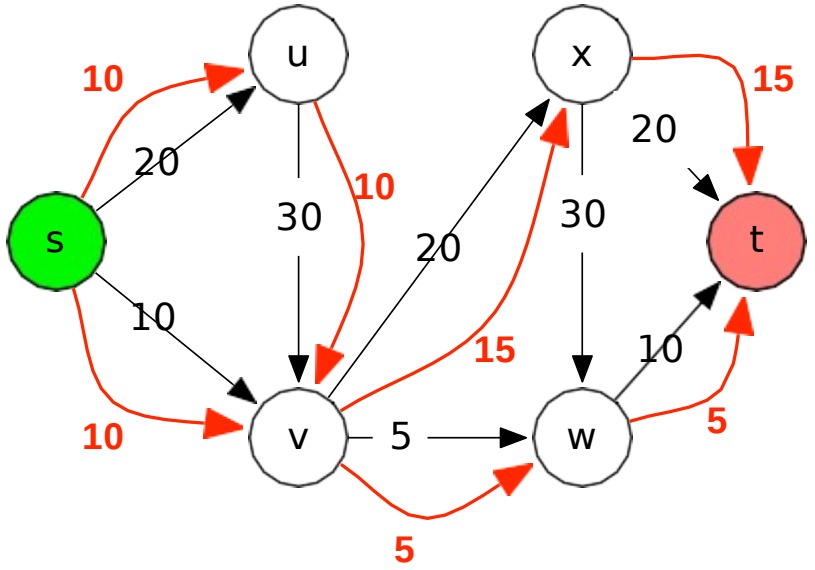
A matching in a Bipartite Graph is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matchings for a given Bipartite Graph.

Why do we care?

There are many real world problems that can be formed as Bipartite Matching. For example, consider the following problem:

“There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. Find an assignment of jobs to applicants in such that as many applicants as possible get jobs.”

Network Flows

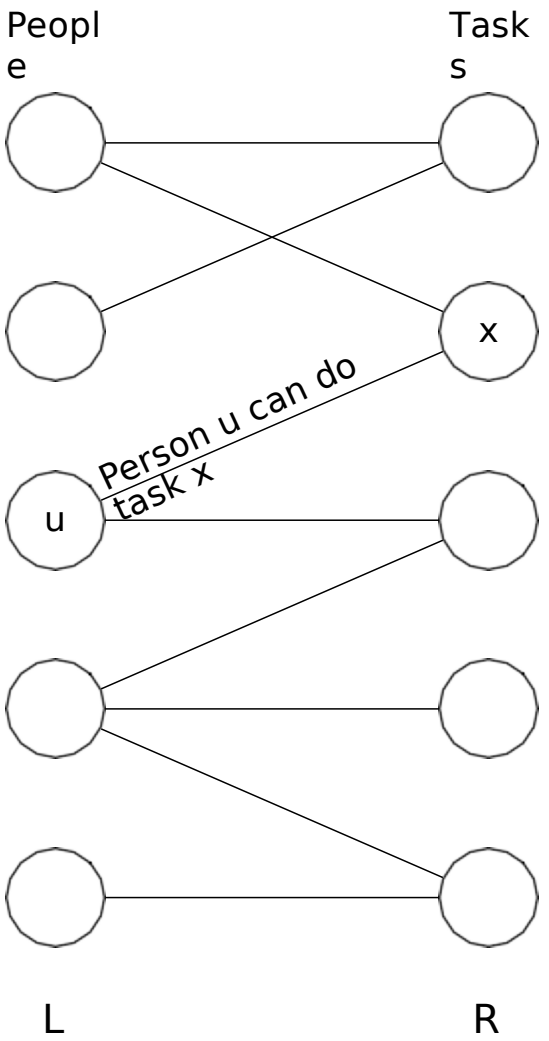


The network flow problem is itself interesting.

But even more interesting is how you can use it to solve many problems that don't involve flows or even networks.

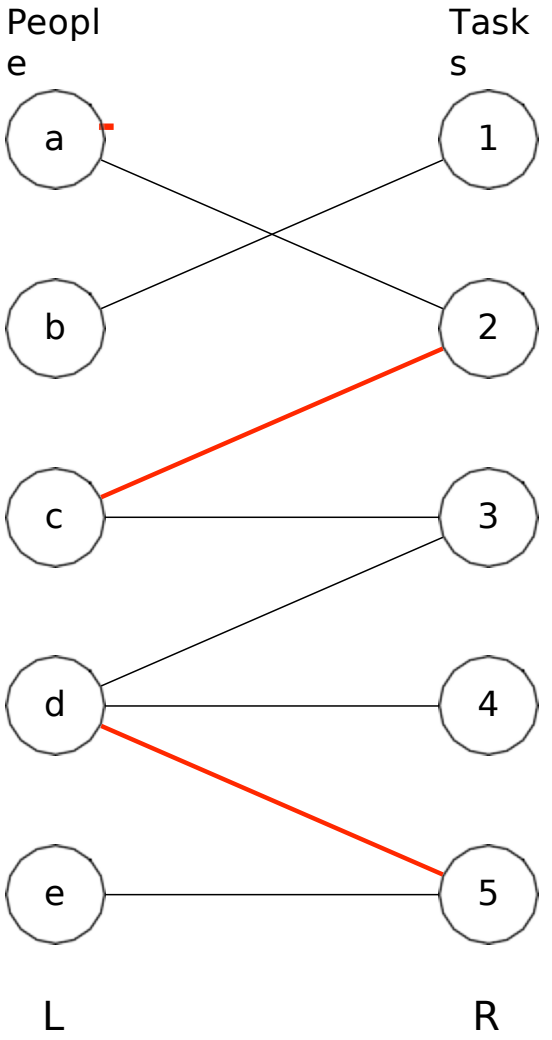
Bipartite Graphs

- Suppose we have a set of people L and set of jobs R.
- Each person can do only some of the jobs.
- Can model this as a bipartite graph →



Bipartite Matching

- A **matching** gives an assignment of people to tasks.
- Want to get as many tasks done as possible.
- So, want a **maximum matching**: one that contains as many edges as possible.
- (This one is not maximum.)



Maximum Bipartite Matching

Maximum Bipartite Matching

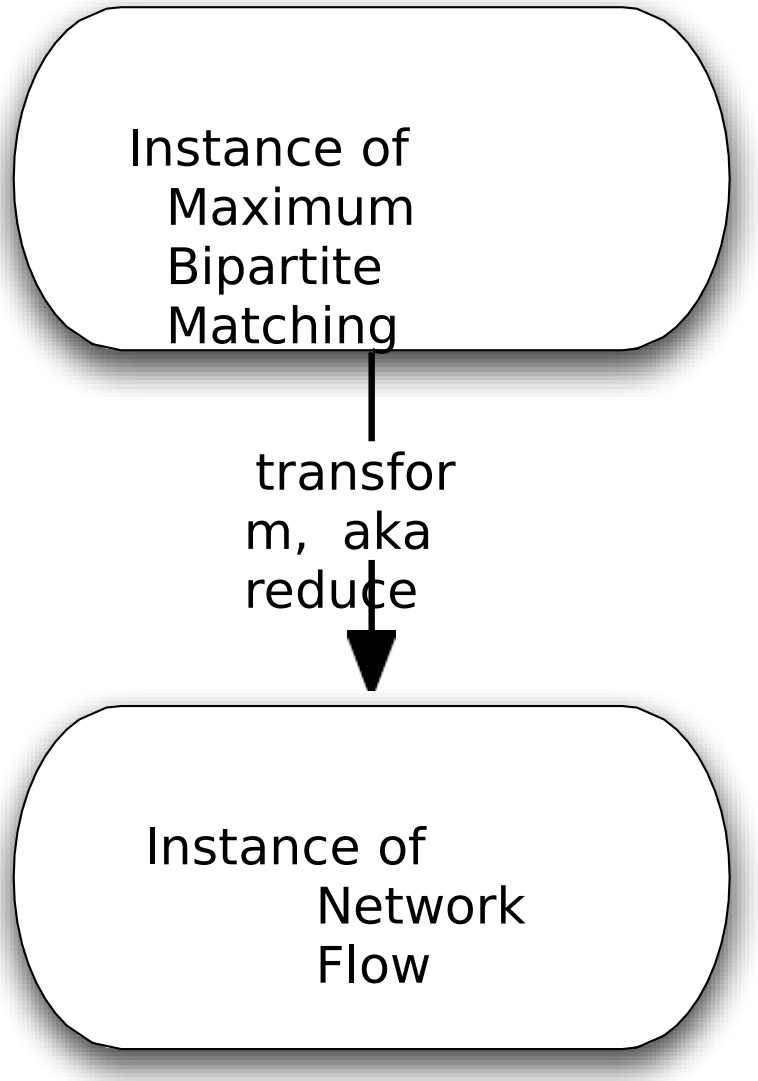
Given a bipartite graph $G = (A \cup B, E)$, find an $S \subseteq A \times B$ that is a matching and is as large as possible.

Notes:

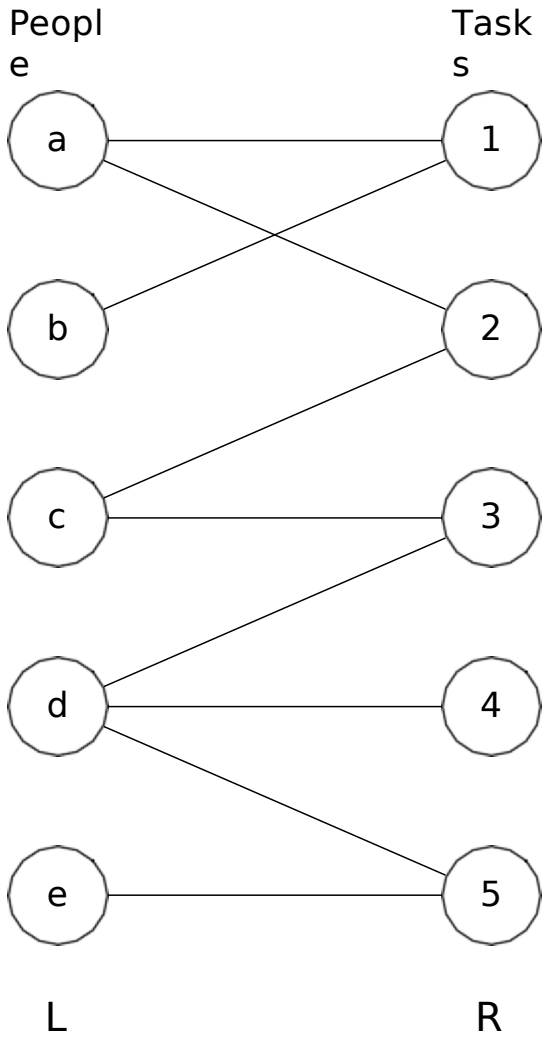
- We're given A and B so we don't have to find them.
- S is a **perfect matching** if every vertex is matched.
- *Maximum* is not the same as *maximal*: greedy will get to maximal.

Reduce

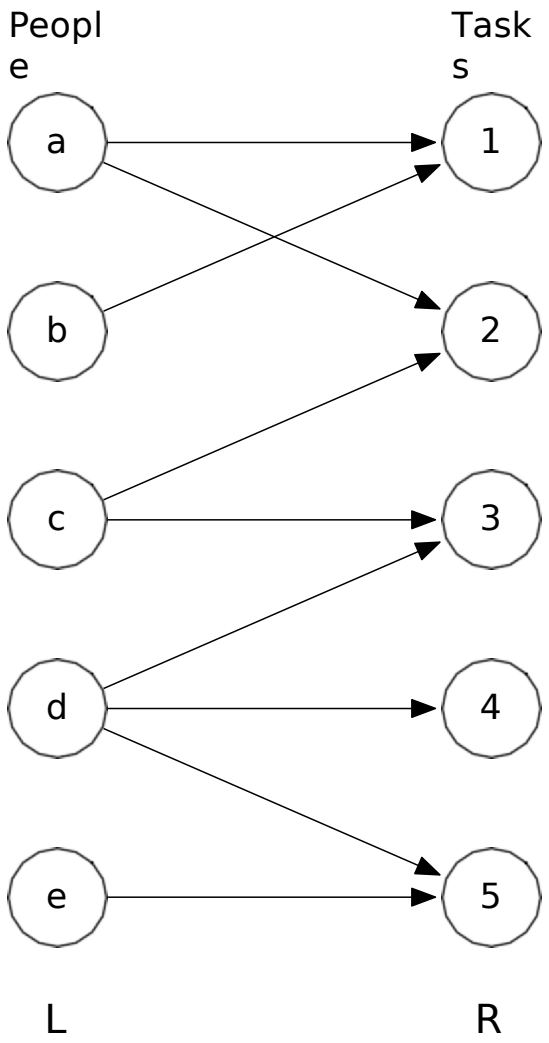
- Given an instance of bipartite matching,
- Create an instance of network flow.
- Where the solution to the network flow problem can easily be used to find the solution to the bipartite matching.



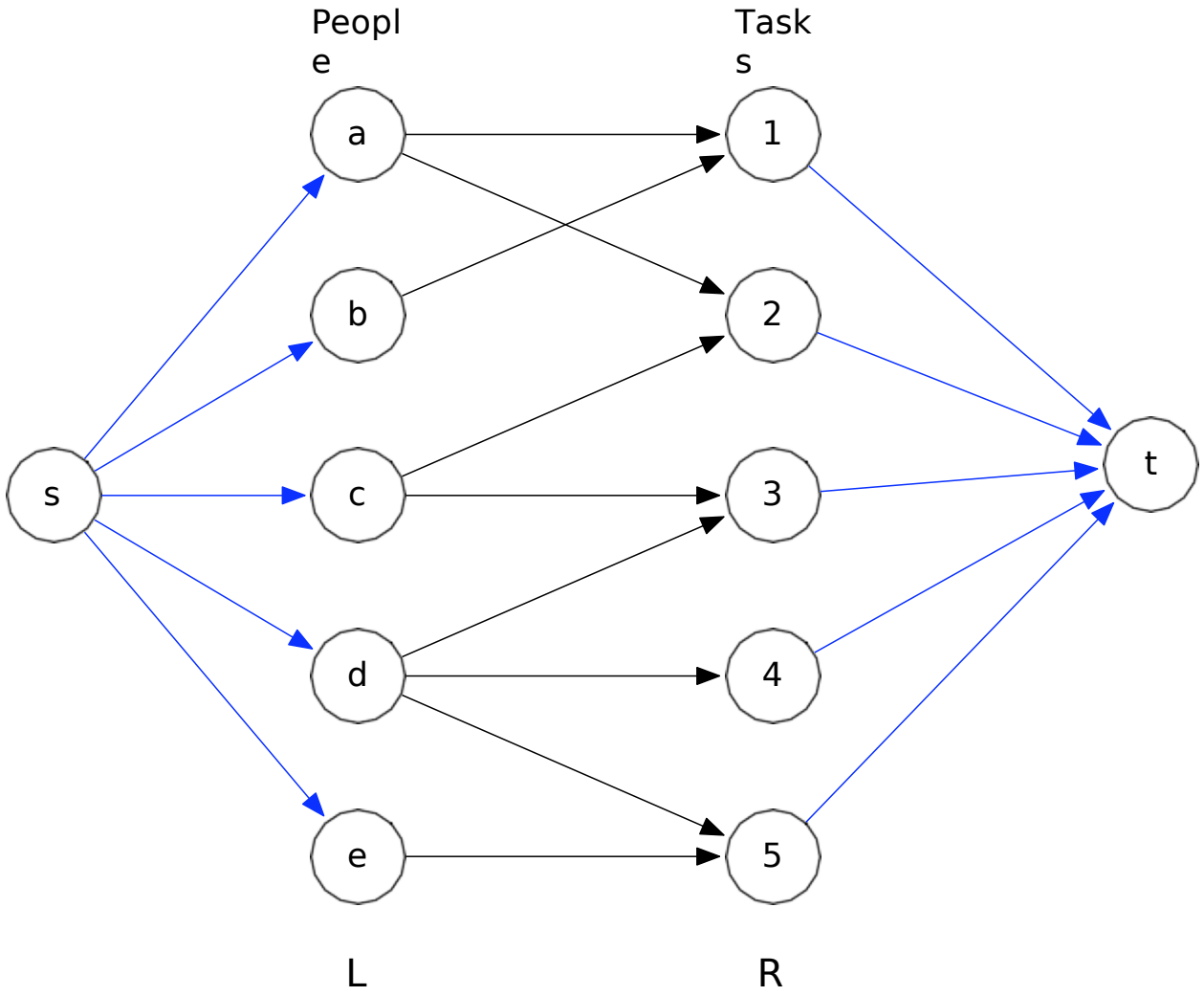
Reducing Bipartite Matching to Net Flow



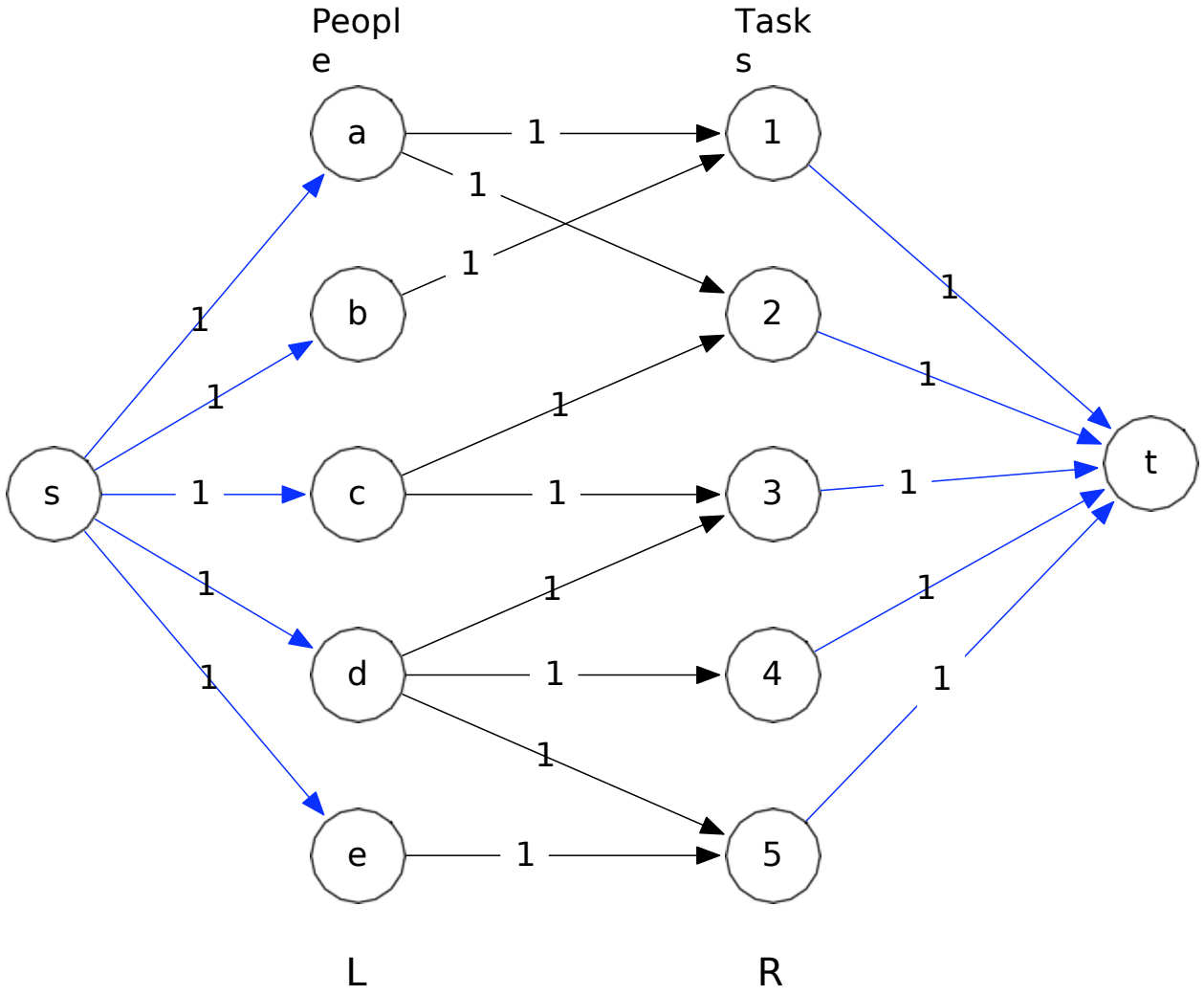
Reducing Bipartite Matching to Net Flow



Reducing Bipartite Matching to Net Flow



Reducing Bipartite Matching to Net Flow



Using Net Flow to Solve Bipartite Matching

To Recap:

1. Given bipartite graph $G = (A \cup B, E)$, direct the edges from A to B.
2. Add new vertices s and t.
3. Add an edge from s to every vertex in A.
4. Add an edge from every vertex in B to t.
5. Make all the capacities 1.
6. Solve maximum network flow problem on this new graph G^J.

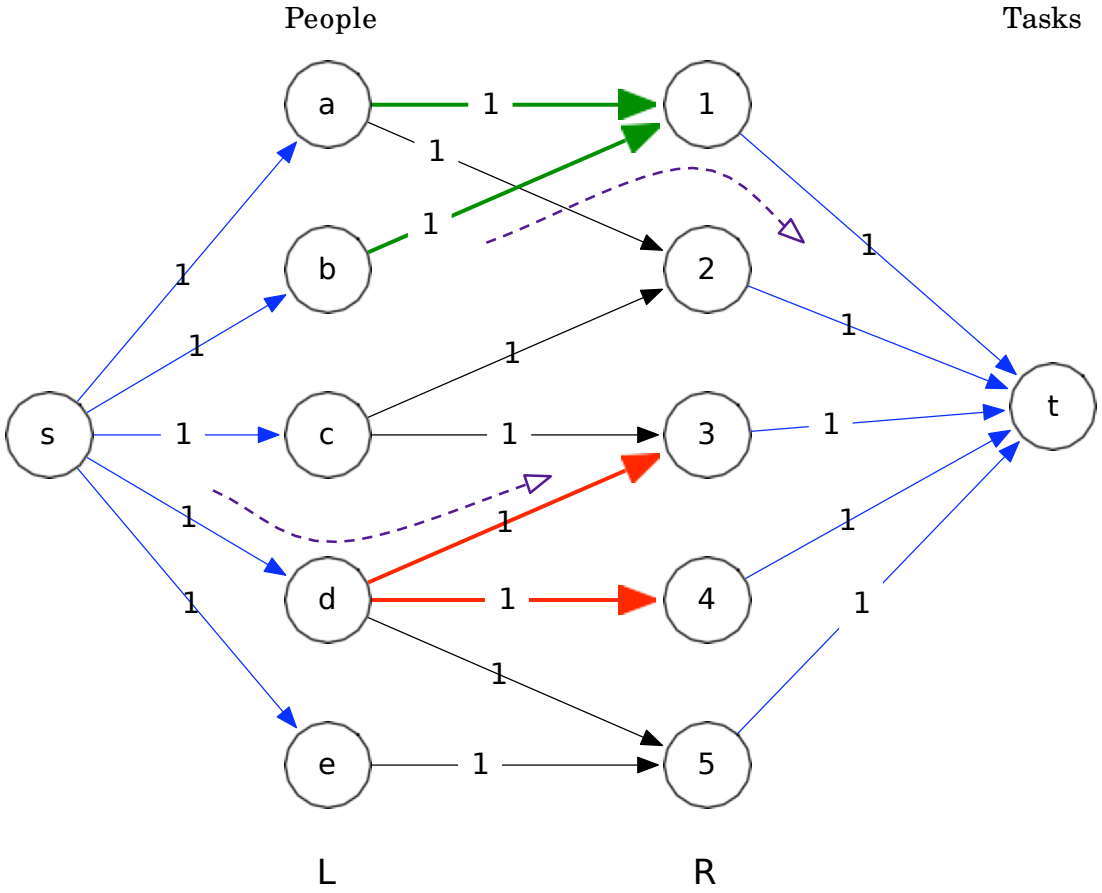
The edges used in the maximum network flow will correspond to the largest possible matching!

Analysis, Notes

- Because the capacities are integers, our flow will be integral.
- Because the capacities are all 1, we will either:
  - use an edge completely (sending 1 unit of flow) or
  - not use an edge at all.
- **Let M be the set of edges going from A to B that we use**.
- We will show that
  1. M is a matching
  2. M is the largest possible matching

M is a matching

We can choose at most one edge leaving any node in A.
We can choose at most one edge entering any node in B.

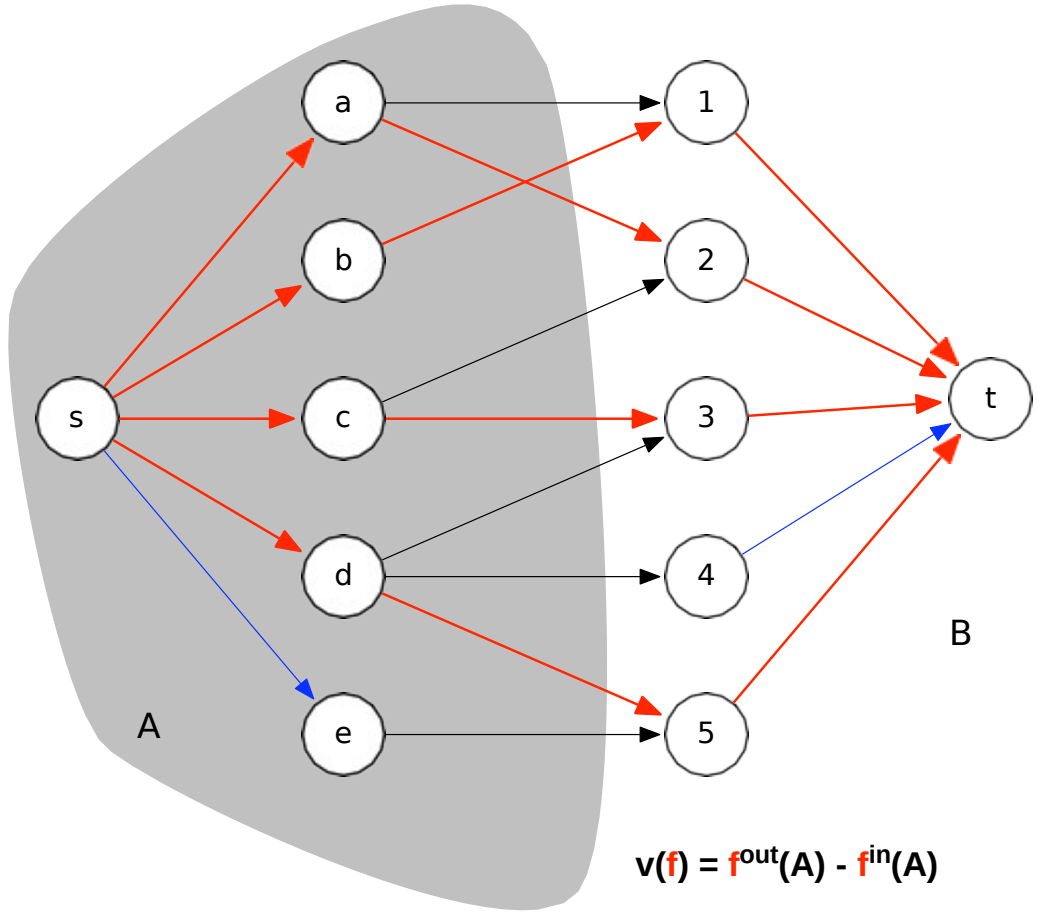


If we chose more than 1, we couldn't have balanced flow.

Correspondence between flows and matchings

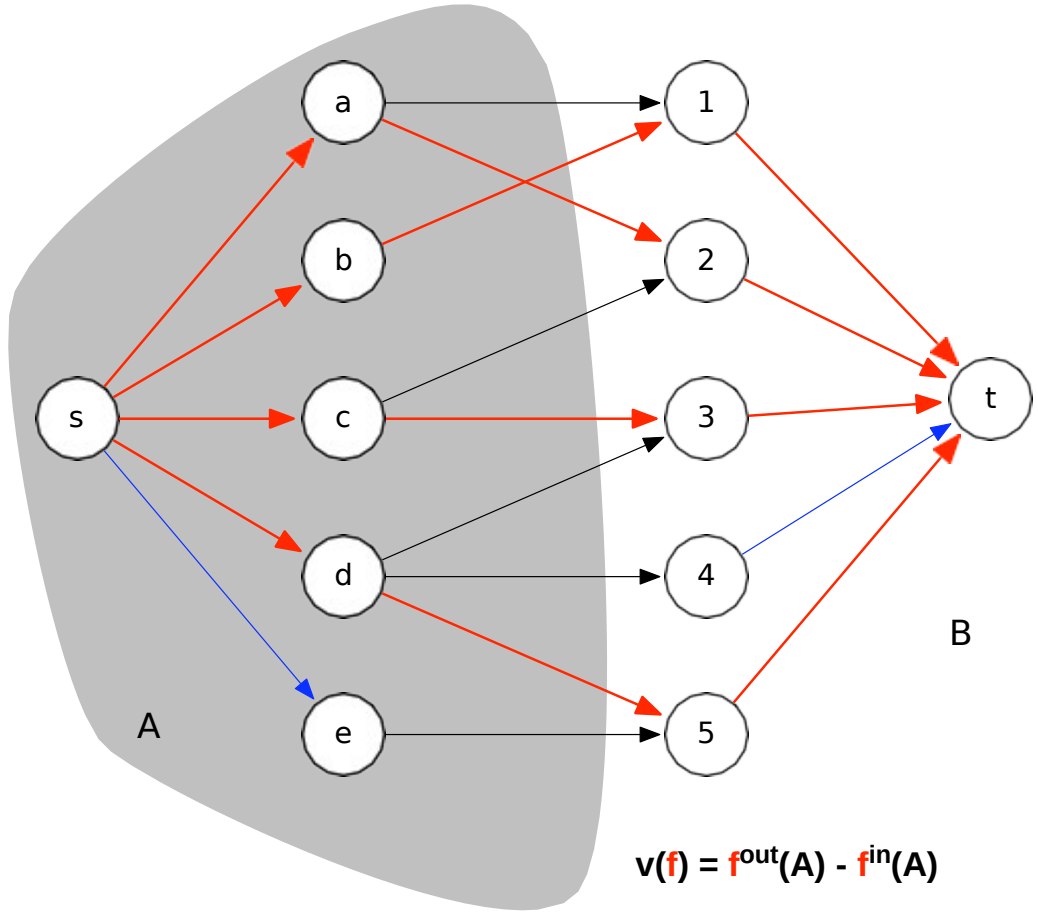
- If there is a matching of k edges, there is a flow f of value k.

- If there is a flow f of value k, there is a matching with k edges.



Correspondence between flows and matchings

- If there is a matching of k edges, there is a flow f of value k.
  - f has 1 unit of flow across each of the k edges.
  - $\leq$ 1 unit leaves & enters each node (except s, t)
- If there is a flow f of value k, there is a matching with k edges.



M is as large as possible

- We find the **maximum** flow f (say with k edges).
- This corresponds to a matching M of k edges.
- If there were a matching with $> k$ edges, we would have found
- a flow with value $> k$, contradicting that f was maximum.
- Hence, M is maximum.

Running Time

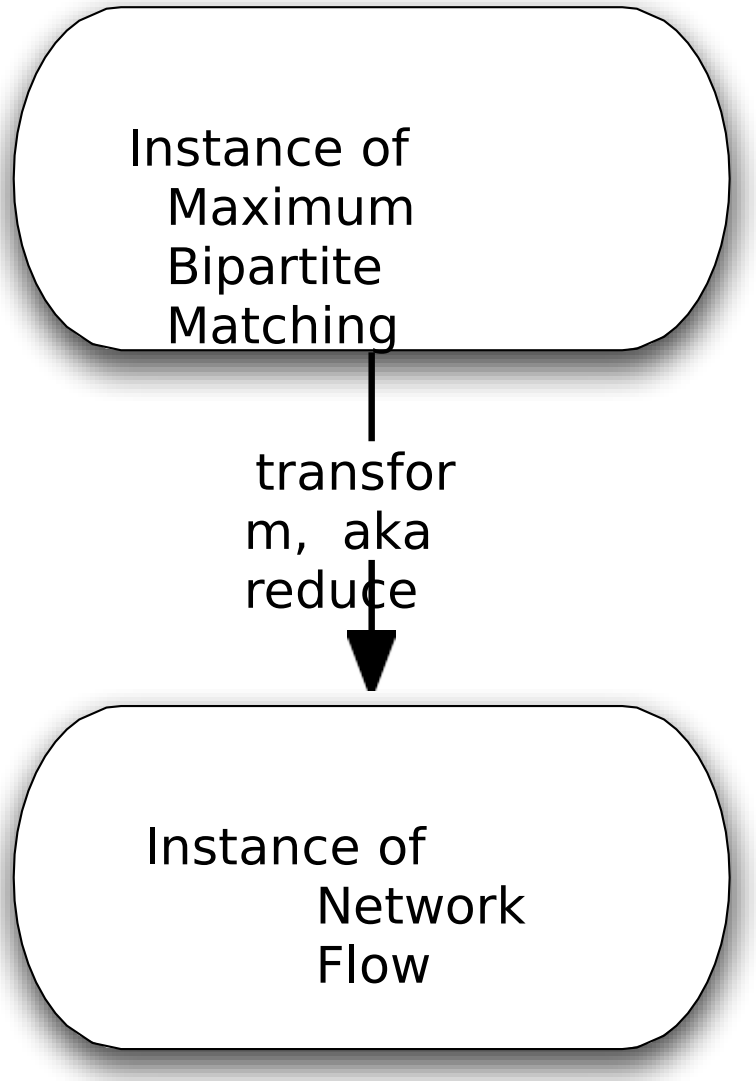
- How long does it take to solve the network flow problem on G'?
- The running time of Ford-Fulkerson is $O(m'C)$ where m' is the number of edges, and $C = \sum_{e \text{ leaving } s} c_e$.
- $C = |A| = n$.
- The number of edges in G' is equal to number of edges in G (m) plus $2n$.
- So, running time is $O((m + 2n)n) = (mn + n^2) = O(mn)$

Theorem

We can find maximum bipartite matching in $O(mn)$ time.

Summary: Bipartite Matching

- Fold-Fulkerson can find a maximum matching in a bipartite graph in $O(mn)$ time.
- We do this by **reducing** the problem of maximum bipartite matching to network flow.



Topological Sorting

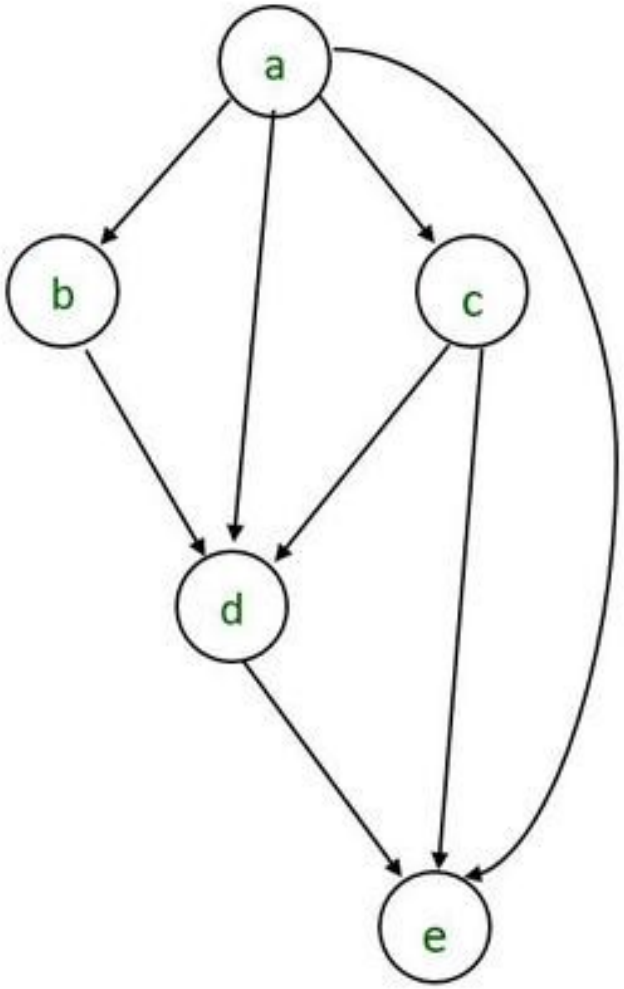
Directed Acyclic Graph (DAG)

In computer science and mathematics, a directed acyclic graph (DAG) refers to a directed graph which has no directed cycles

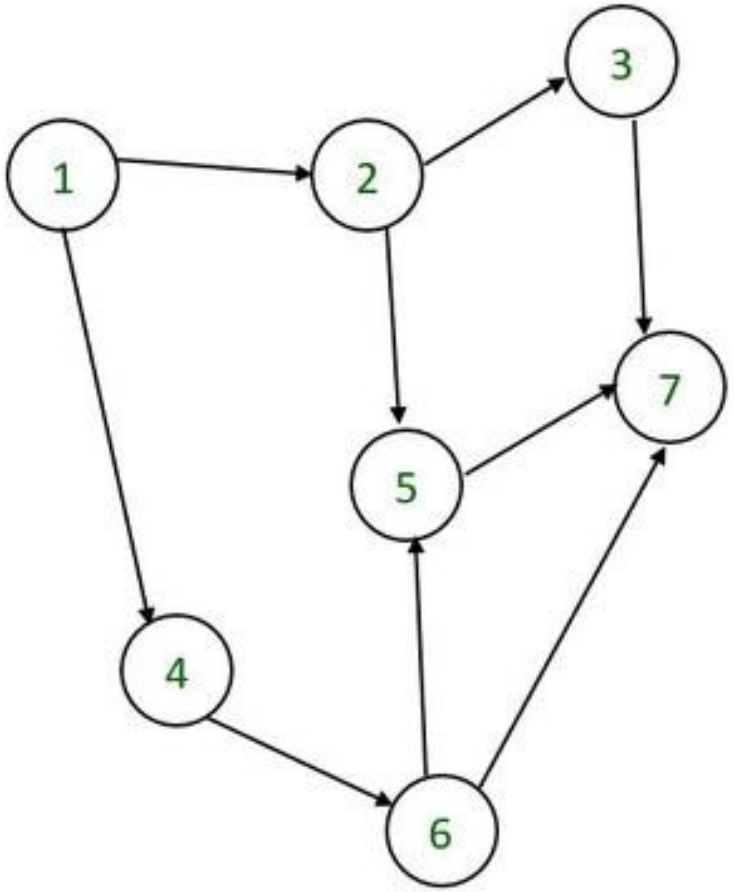
In graph theory, a graph refers to a set of vertices which are connected by lines called edges. In a directed graph or a digraph, each edge is associated with a direction from a start vertex to an end vertex. If we traverse along the direction of the edges and we find that no closed loops are formed along any path, we say that there are no directed cycles. The graph formed is a directed acyclic graph.

A DAG is always topologically ordered, i.e. for each edge in the graph, the start vertex of the edge occurs earlier in the sequence than the ending vertex of the edge.

Example

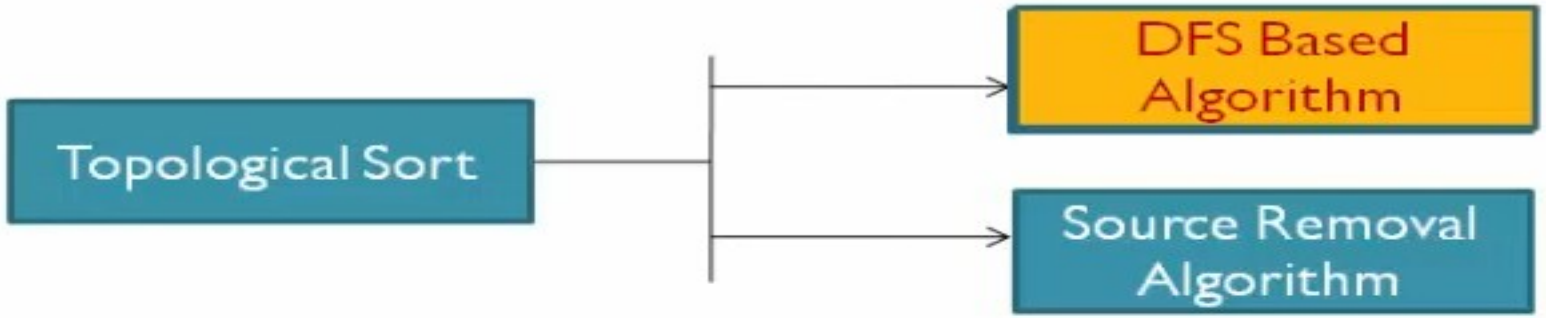


(A)



(B)

Different methods for Topological Sort



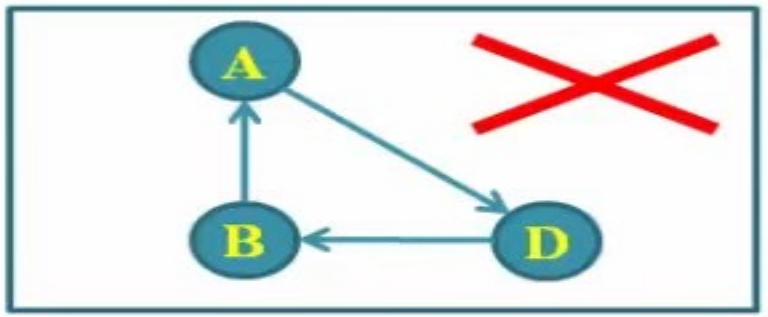
DFS Based Algorithm

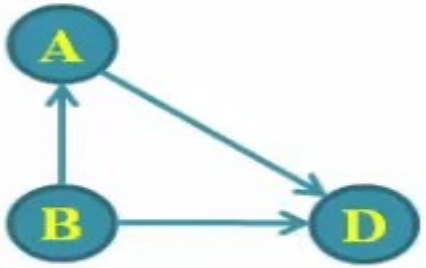
1) Perform DFS algorithm.
2) Store POP Off content of Stack separately.
3) At the end reverse the Pop off Content. It will give one topological sort.

Note : Same graph may have multiple Topological Sort.

Topological Sort

- Topological Sort is a process of assigning a linear ordering to the vertices of a DAG (Directed Acyclic Graph), so that if there is an edge from **vertex i to vertex j** then **i appears before j in linear ordering.**
- **Example :**





= B - D - A

Topological Sort
= B - A - D

- **Use of Topological Sort :**

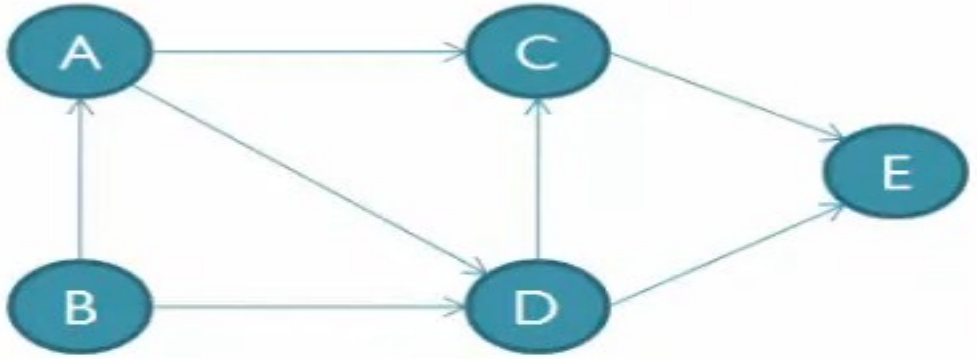
  Suppose the vertices of the graph may represent tasks to be performed, and the edges may represent constraints that one task must be performed before another,

  In this situation, a topological sort is just a valid sequence for the tasks.

One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.



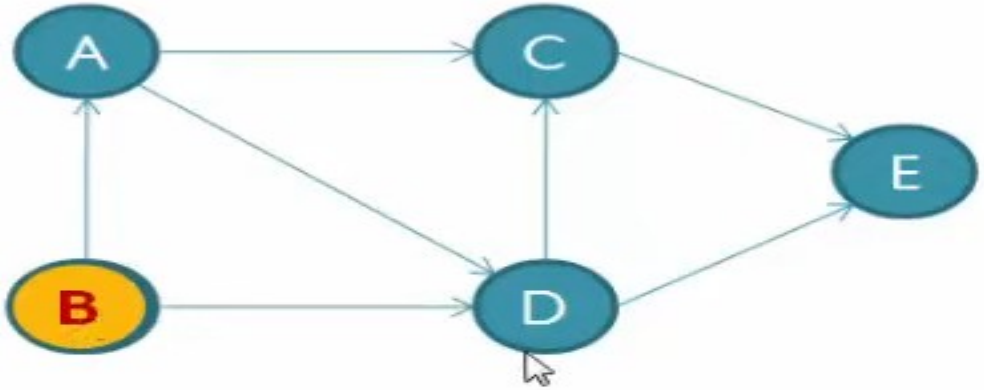
Pop-off contents of Stack

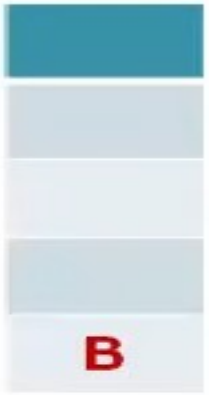
In DFS, we can start traversing from any node.
But, in case of topological sorting, we have to start from a node that has indegree = 0.
In this example, B is such a node. So, we will start from B

One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.



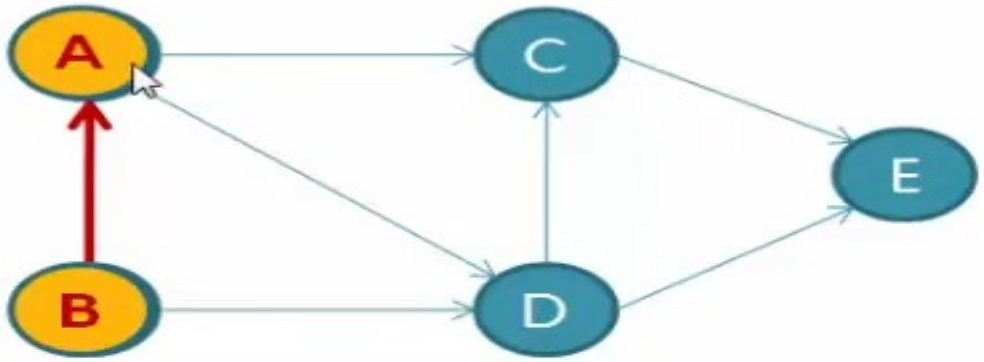


Pop-off contents of Stack

One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.

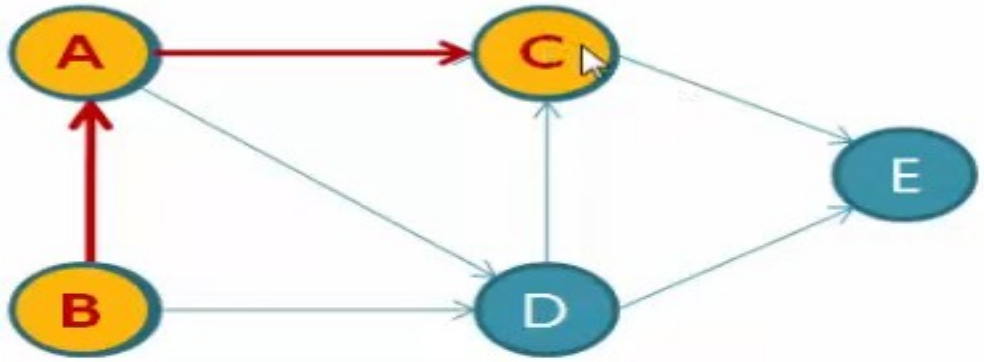


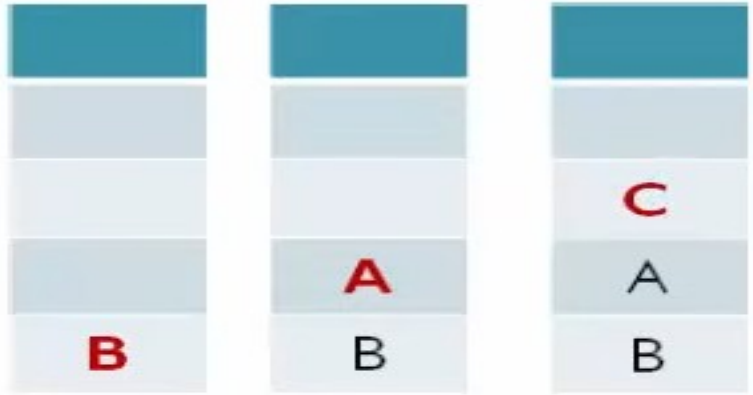
Pop-off contents of Stack

One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.



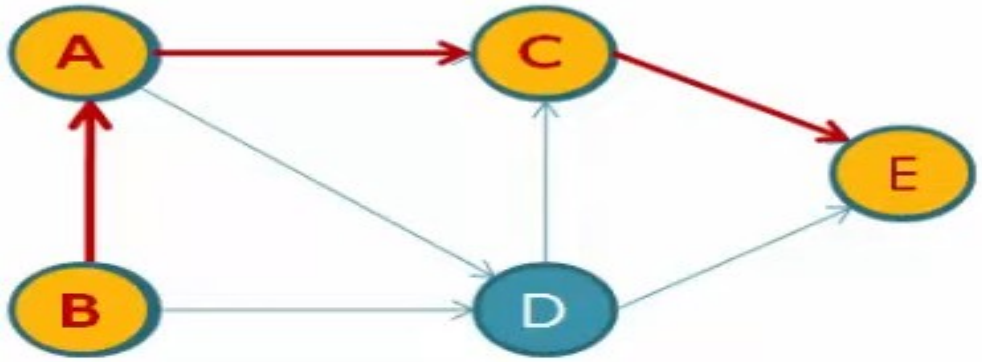


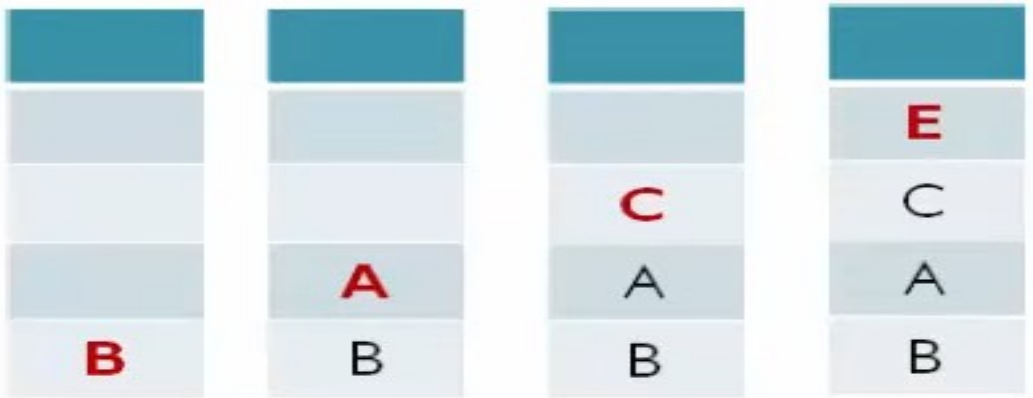
Pop-off contents of Stack

One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.



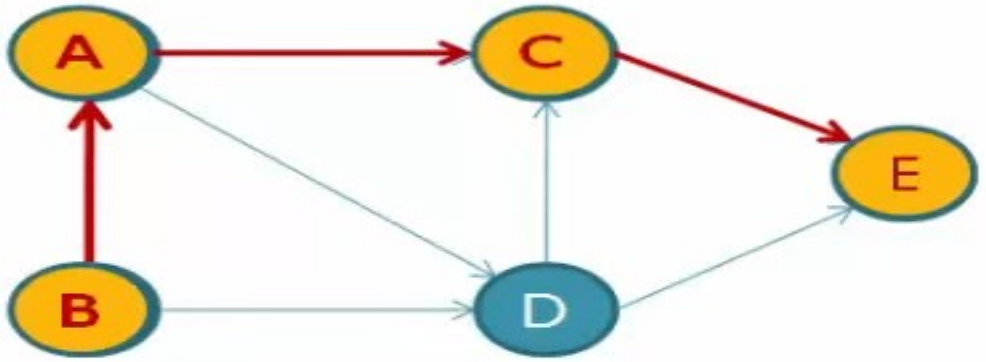


Pop-off contents of Stack

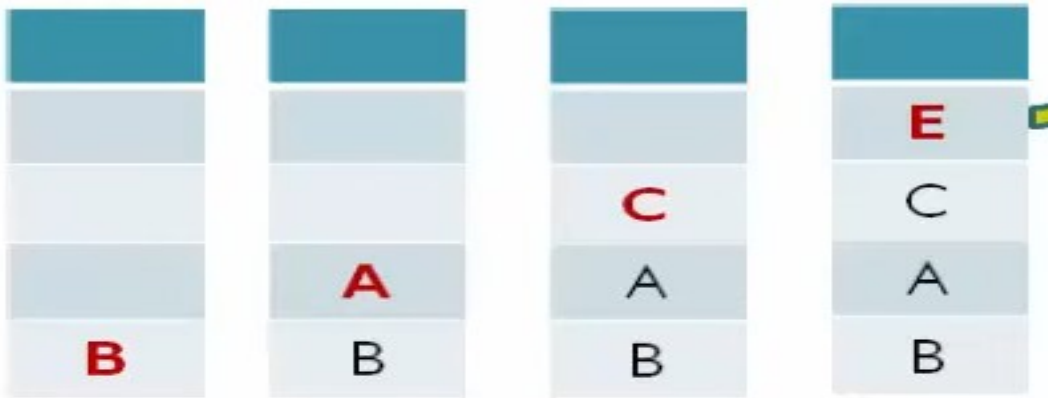
One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.



E has no any unvisited neighbor node.
So **POP(E)**

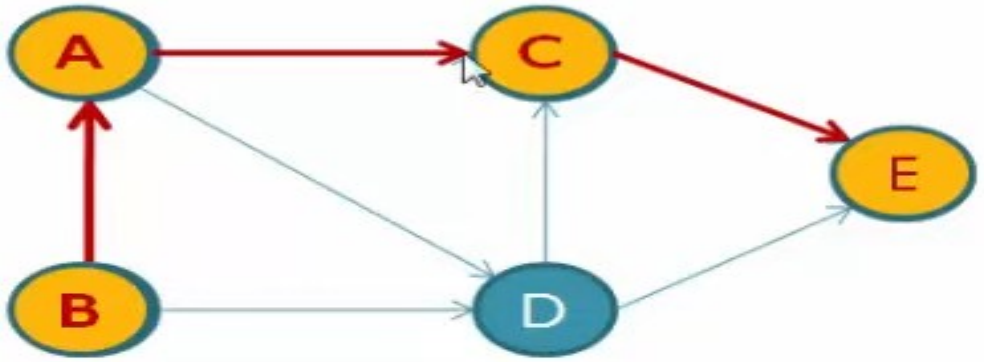


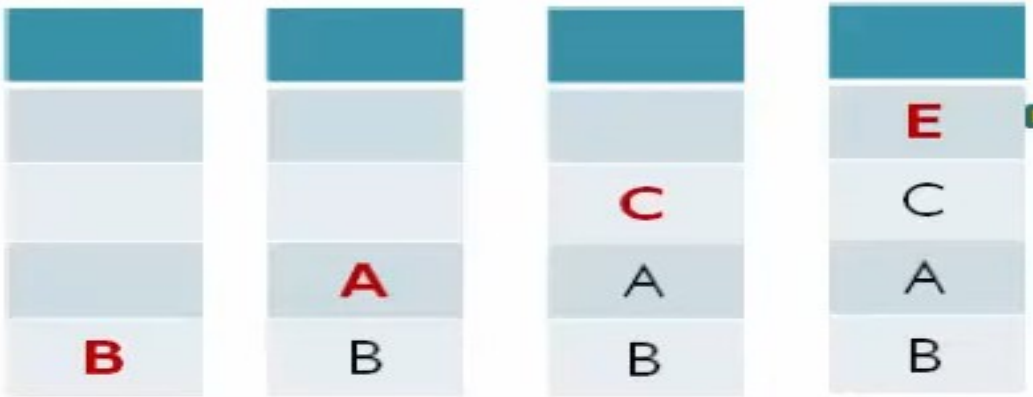
Pop-off contents of Stack

One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.





E has no any unvisited neighbor node.
So **POP(E)**

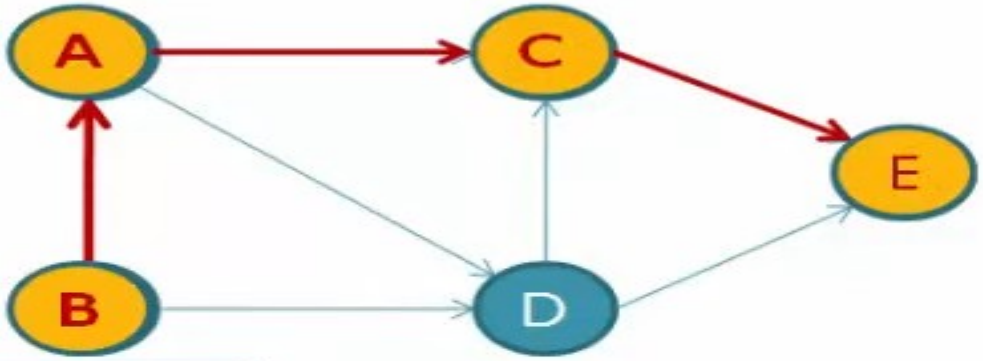
Pop-off contents of Stack

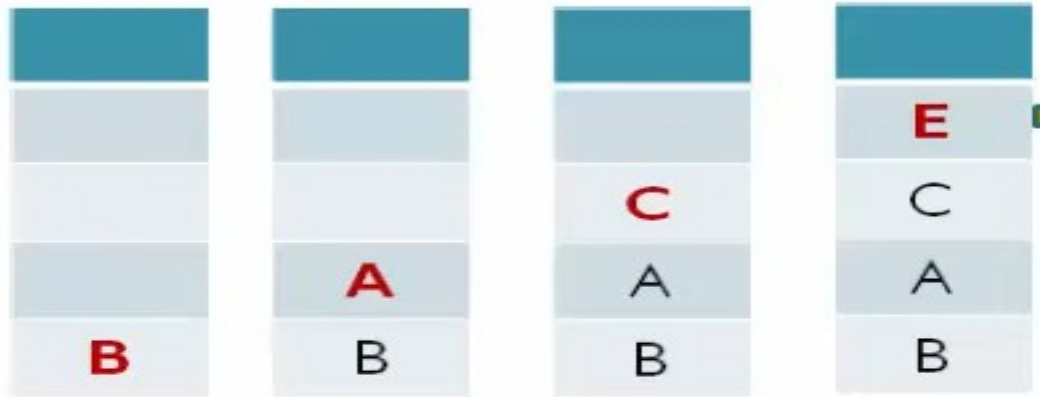
E

One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.





E has no any unvisited neighbor node.
So POP(E)

No unvisited neighbor node.
So POP(C)

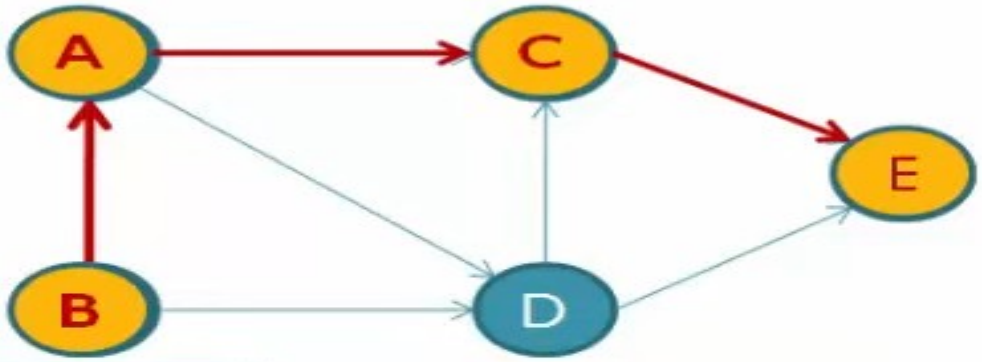
Pop-off contents of Stack

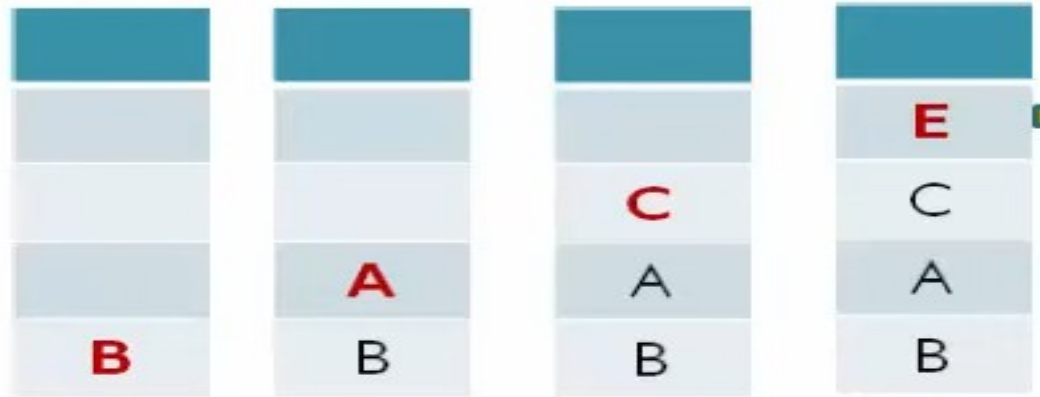
E

One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.





E has no any unvisited neighbor node.
So POP(E)

No unvisited neighbor node.
So POP(C)

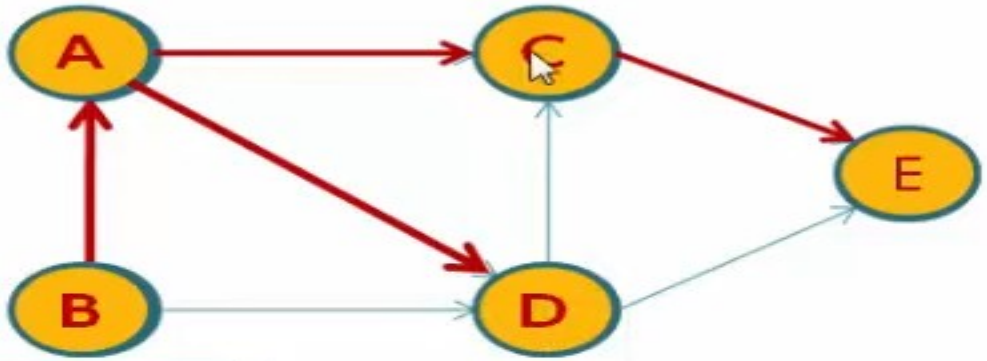
Pop-off contents of Stack

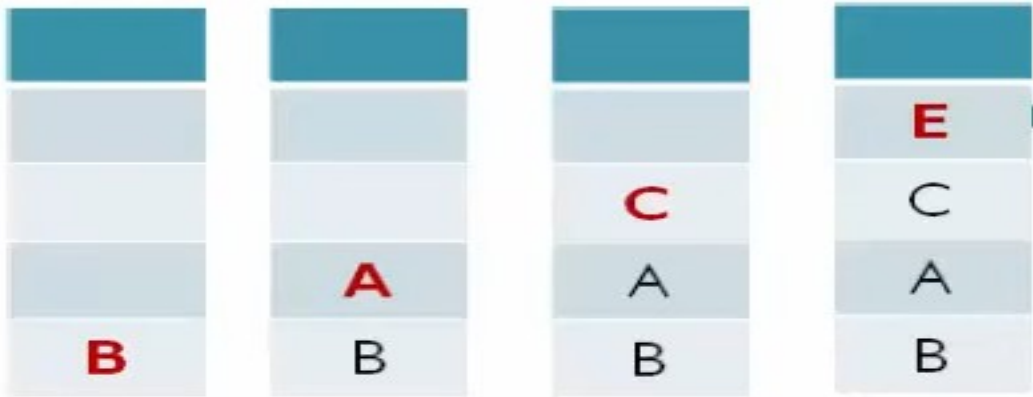
E C

One graph may have many topological order .

Example

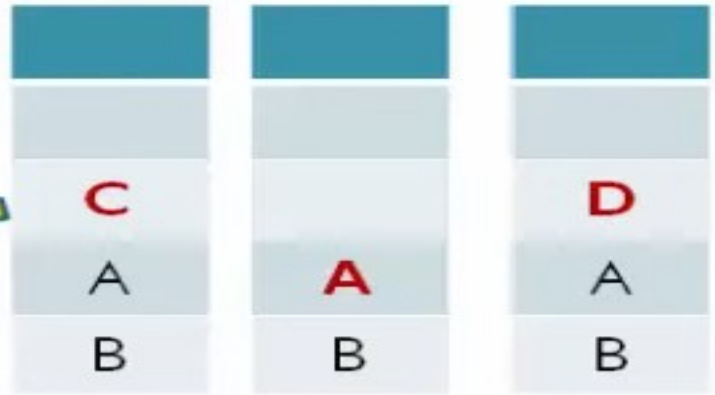
Sort the digraph for topological sort using DFS based algorithm.





E has no any unvisited neighbor node.
So POP(E)

No unvisited neighbor node.
So POP(C)



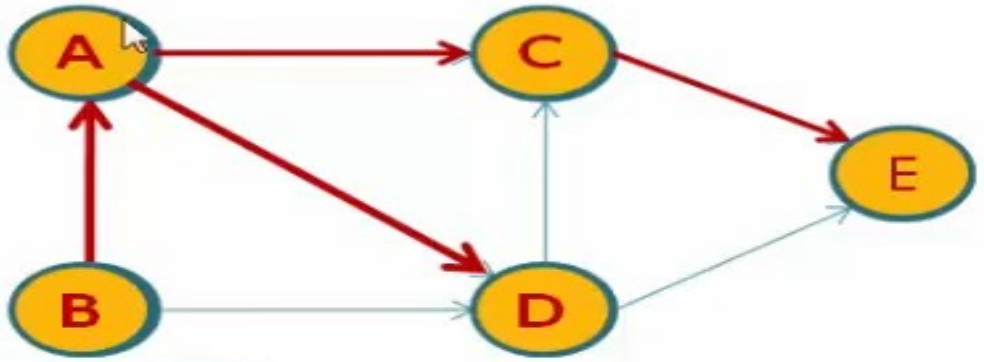
Pop-off contents of Stack

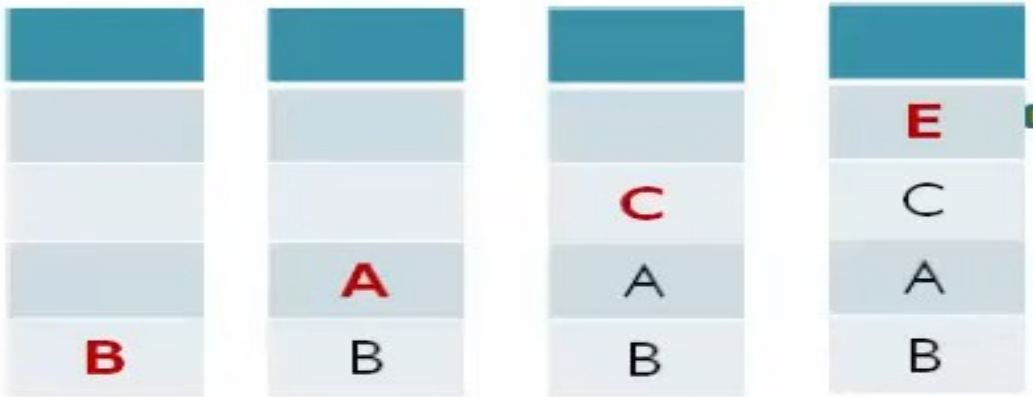
E C

One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.





E has no any unvisited neighbor node.
So POP(E)

No unvisited neighbor node.
So POP(C)

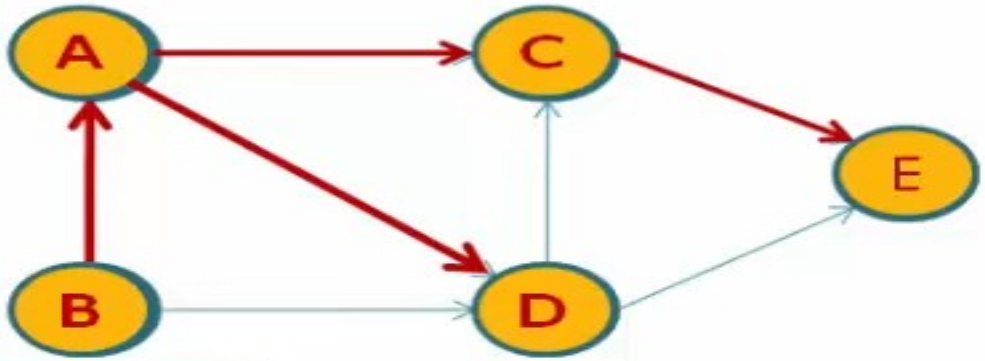
Pop-off contents of Stack

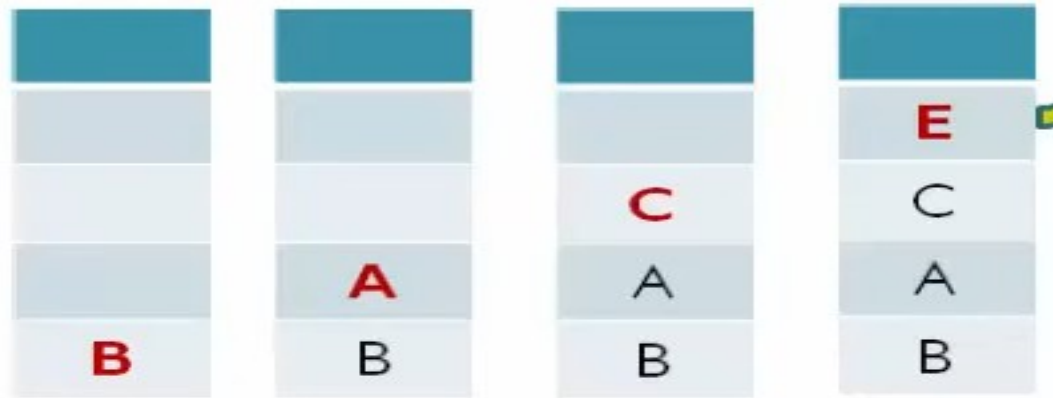
E C D

One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.





E has no any unvisited neighbor node.
So POP(E)

No unvisited neighbor node.
So POP(C)

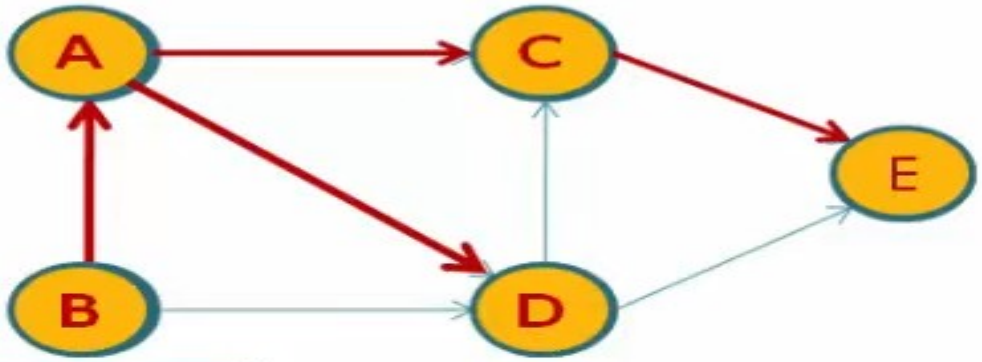
Pop-off contents of Stack

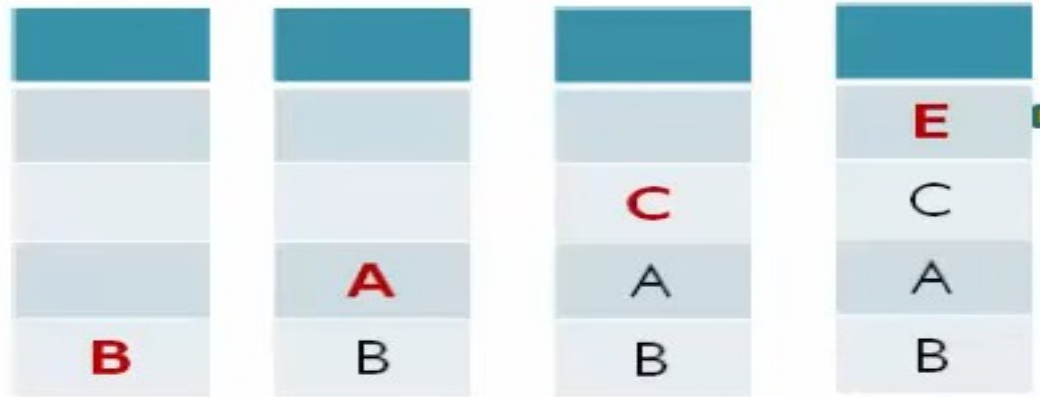
E C D A

One graph may have many topological order .

Example

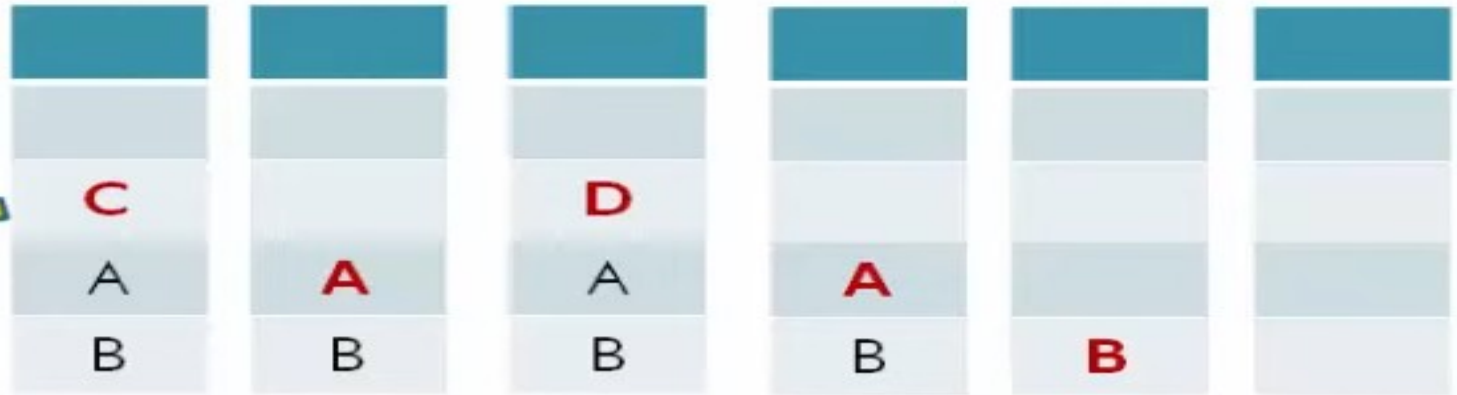
Sort the digraph for topological sort using DFS based algorithm.





E has no any unvisited neighbor node.
So POP(E)

No unvisited neighbor node.
So POP(C)



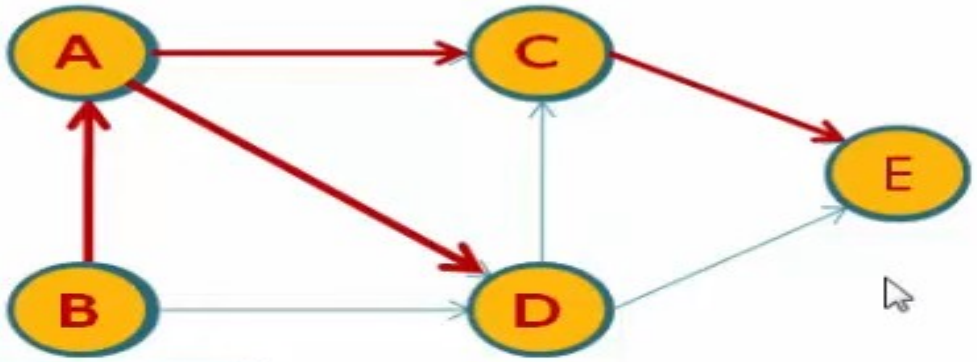
Pop-off contents of Stack

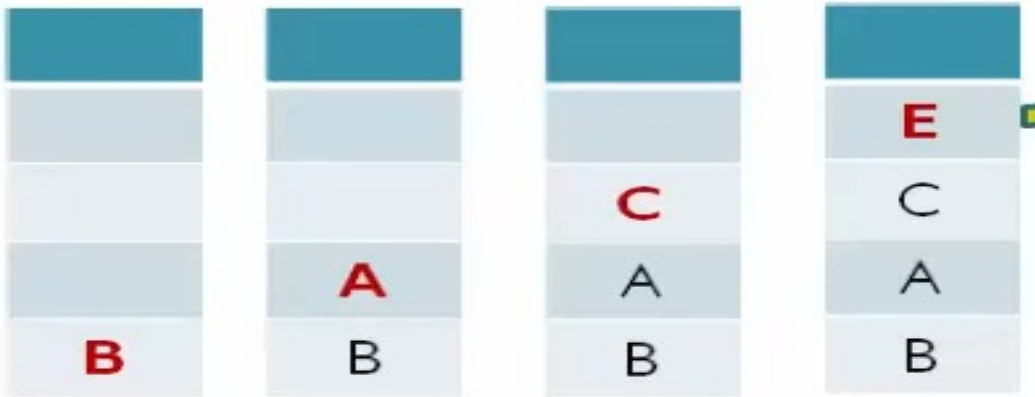
E C D A B

One graph may have many topological order .

Example

Sort the digraph for topological sort using DFS based algorithm.





E has no any unvisited neighbor node.
So POP(E)

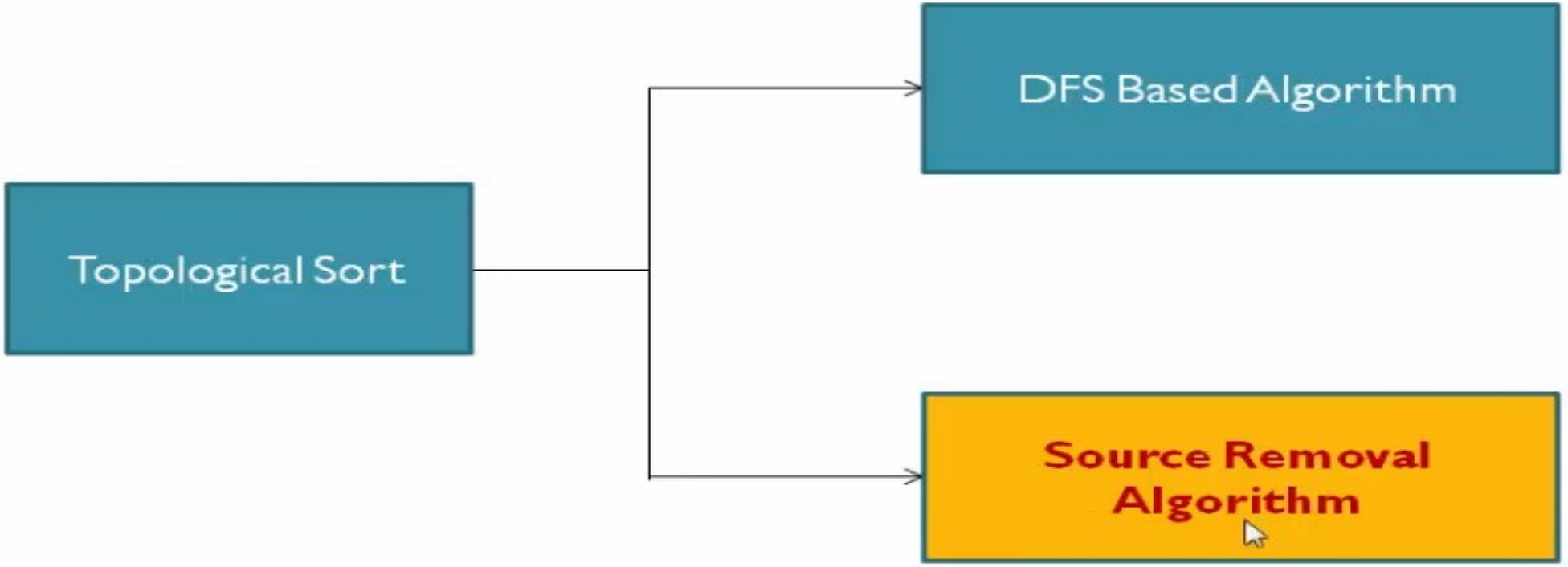
No unvisited neighbor node.
So POP(C)

Pop-off contents of Stack

E C D A B

Reverse the content.
Topological Sort
B A D C E

Different methods for Topological Sort



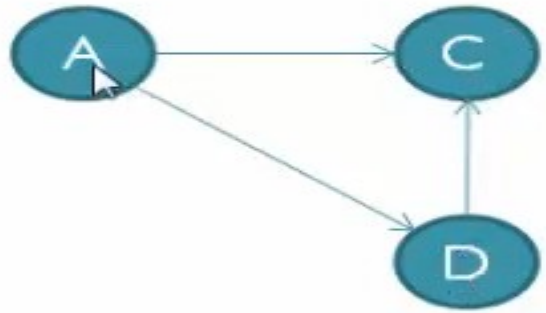
(Kahn's Algorithm)

Note : Same graph may have multiple Topological Sort.

Steps for Source Removal Algorithm

- This is a direct implementation of decrease and conquer method. Following are the steps to be followed in this algorithm.

1. From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it. If there are more than one such vertices then break the tie randomly.



Steps for Source Removal Algorithm

- This is a direct implementation of decrease and conquer method. Following are the steps to be followed in this algorithm.

1. From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it. If there are more than one such vertices then break the tie randomly.

Steps for Source Removal Algorithm

- This is a direct implementation of decrease and conquer method. Following are the steps to be followed in this algorithm.

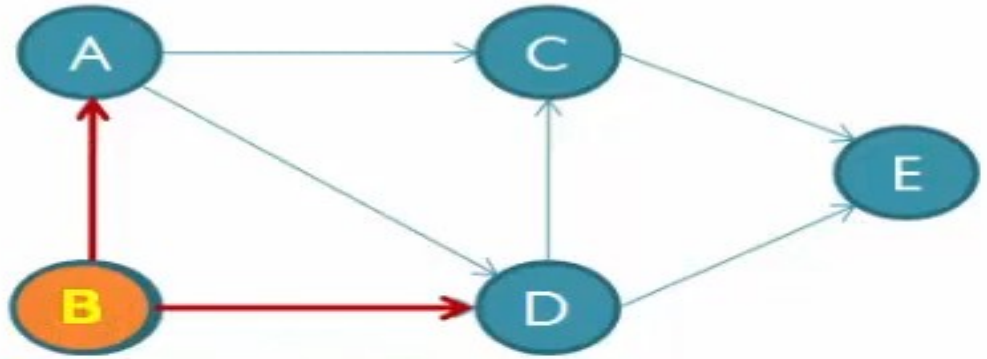
1. From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it. If there are more than one such vertices then break the tie randomly.

2. Note the vertices that are deleted.

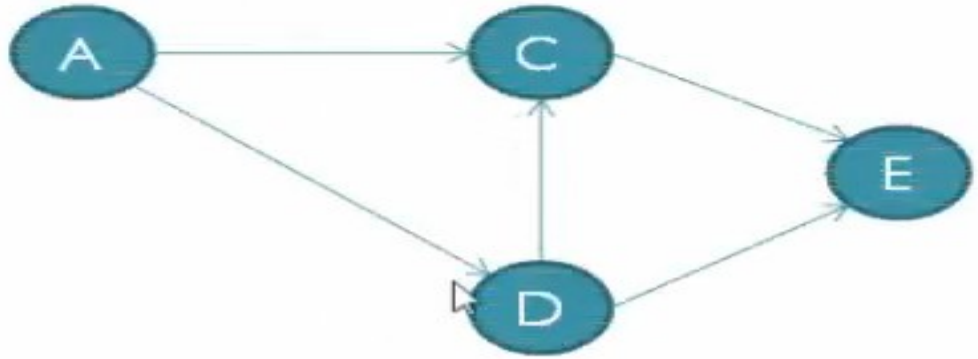
3. All these recorded vertices give topologically sorted list.

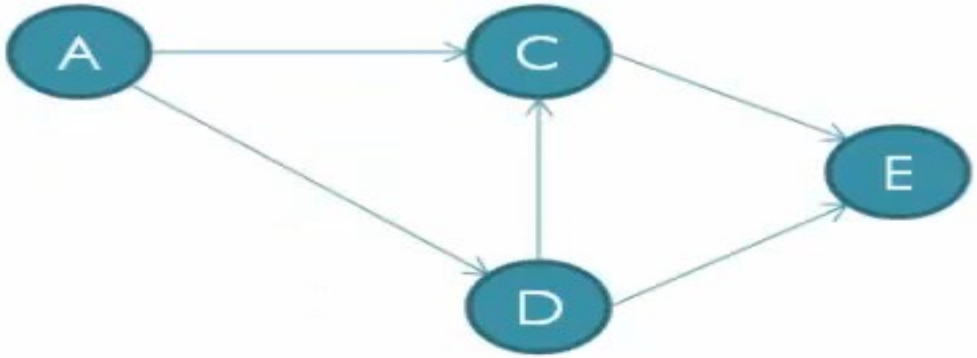
Example

- Sort the digraph for topological sort using source removal algorithm.

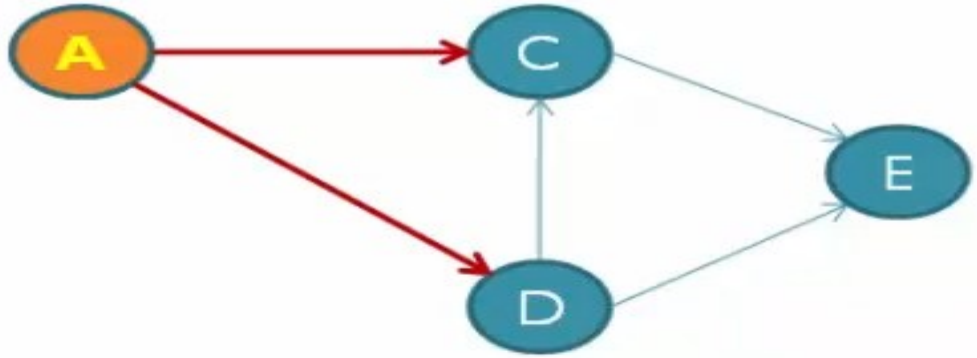


From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it.



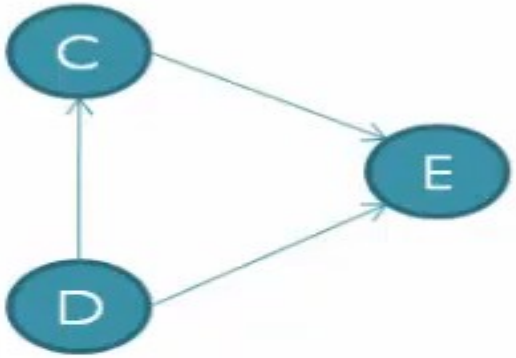


B

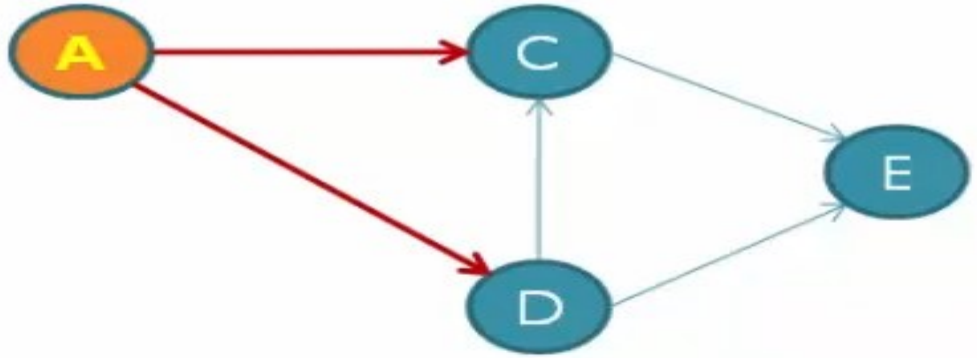


B

Again From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it.

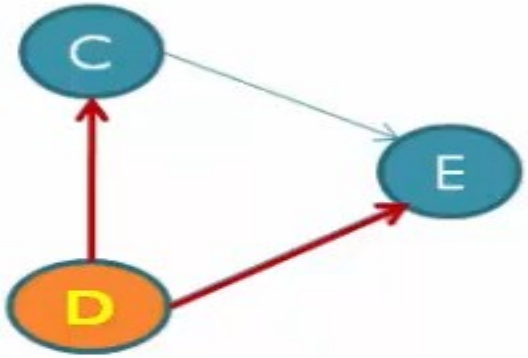


B , A



B

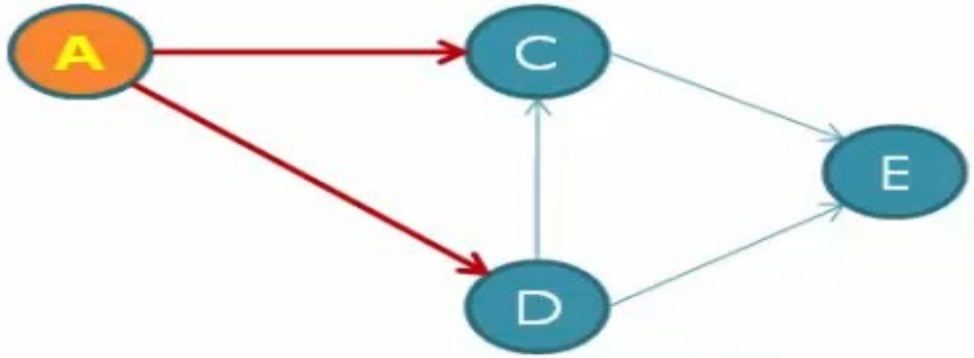
Again From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it.



B , A

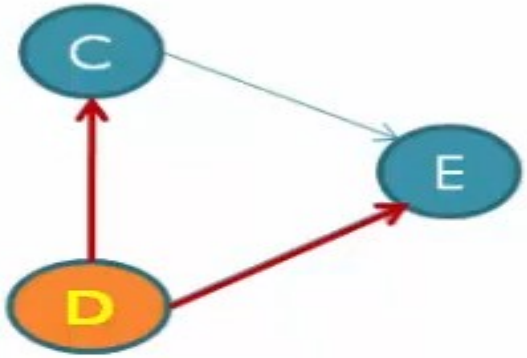
Again From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it.

B , A , D



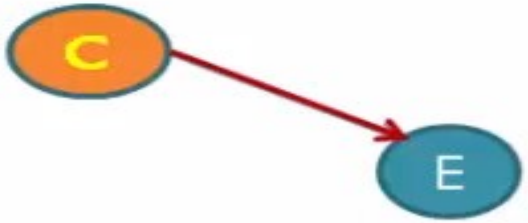
B

Again From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it.



B , A

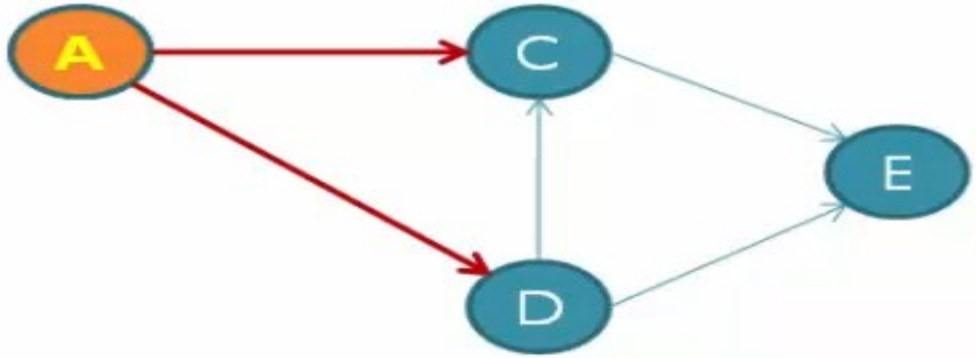
Again From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it.



B , A , D

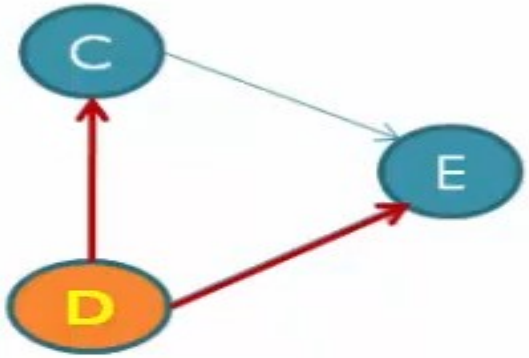
Again From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it.

B , A , D , C



B

Again From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it.



B , A

Again From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it.

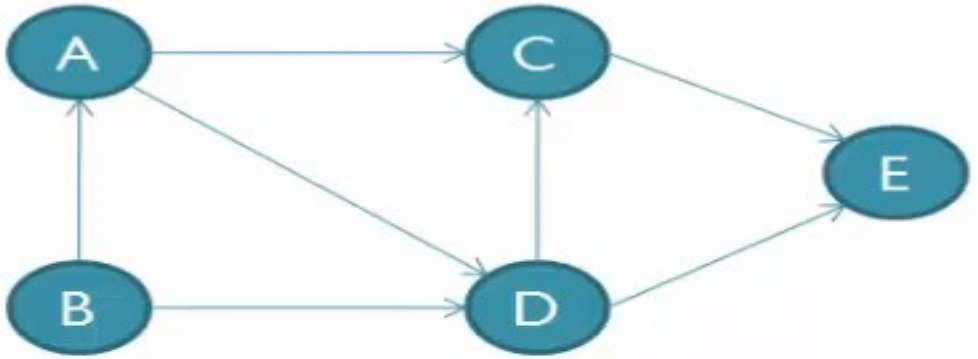
B , A , D

Again From a given graph find a vertex with no incoming edges. Delete it along with all the edges outgoing from it.

B , A , D , C

Final Topological Sort

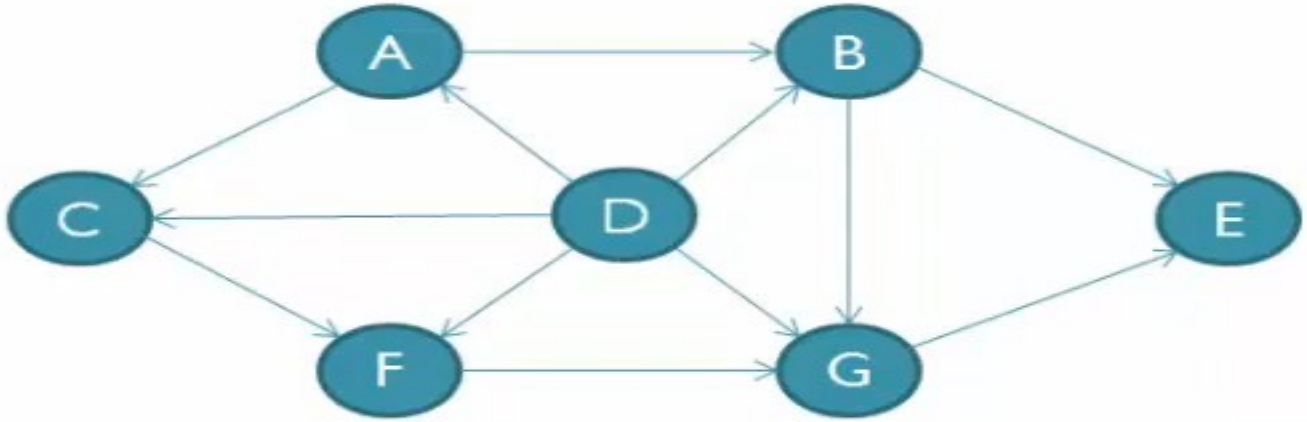
B , A , D , C , E



B , A , D , C , E

This is one topological sort of the given graph.
But this sort is not ‘only one’ or unique.
One DAG may have multiple valid topological orderings.

Find Topological Sort of following Graph using Source Removal Algorithm **and DFS based algorithm**



D , A , B , C , F , G , E

D , A , C , B , F , G , E

Running Time of Topological Sort

Time complexity for Kahn's algorithm:

Lets assume that we are ordering a graph with V vertices and E edges using Topological Sort.

- While traversing the nodes, when we come across a node (Let it be X), we need to decrease the indegree of all the nodes which have the edges from the node X. So time complexity for decreasing all the indegree of the connected nodes is O(|E|).
- In Topological sort, we run across all edges, which takes O(|V|) time. Hence overall time complexity becomes O(|V|+|E|).

Time complexity for Topological sort through DFS:

Since it is modification of DFS, time complexity doesn't alter. Time complexity is O(|V|+|E|).

Space Complexity of Topological Sort

Space complexity for Kahn's Algorithm:

While enqueuing a node, we need some extra space to store temporary values. Other than that, the ordering can be done in-place. Hence space complexity is $O(|V|)$.

Space complexity for Topological Sort through DFS:

Since we need to store the sequence of nodes into a stack, we need some extra space. Other than that, the ordering can be done in-place. Hence space complexity is $O(|V|)$.

Applications of Topological Sort

1. Task Scheduling:

In project management and task scheduling, topological sort helps determine the optimal order of tasks with dependencies. Each node in the directed acyclic graph (DAG) represents a task, and directed edges represent dependencies between tasks. By performing a topological sort, you can schedule tasks in the correct sequence to ensure that dependent tasks are completed before their dependents.

2. Software Dependency Resolution:

When managing software projects with multiple modules or libraries, topological sort assists in resolving dependencies between software components. The algorithm ensures that dependencies are resolved in the correct order during compilation or deployment, preventing issues arising from missing dependencies.

Applications of Topological Sort

3. Building Makefiles:

Makefiles in software development specify the sequence of commands required to build an application from its source code. Topological sort can be used to determine the correct order in which source files need to be compiled, taking into account their interdependencies.

4. Compiler Optimizations:

In compilers, topological sort aids in optimizing the order in which code is generated for different program segments. This optimization ensures that variables are allocated in the correct order and that loops are compiled with appropriate loop unrolling techniques

Applications of Topological Sort

5. Dependency Analysis:

In software engineering, topological sort is used to analyze dependencies between modules or components. This helps in understanding the relationships between different parts of a software system, enabling better maintenance and modification.

6. Deadlock Detection:

Topological sort can be used in resource allocation systems to detect deadlocks, which occur when processes are waiting for resources that are blocked by other processes. By creating a resource allocation graph and performing a topological sort, you can identify cycles that indicate potential deadlocks.

Applications of Topological Sort

7. Course Scheduling:

In educational institutions, topological sort can help create efficient course schedules that ensure prerequisites are met. Each course is represented as a node, and prerequisites are represented by directed edges.

8. Event Management:

In event scheduling systems, topological sort assists in determining the optimal order of events or tasks to ensure that no event starts before its prerequisites are completed.