UNIVERSITY OF TORONTO

REFRESHER ON

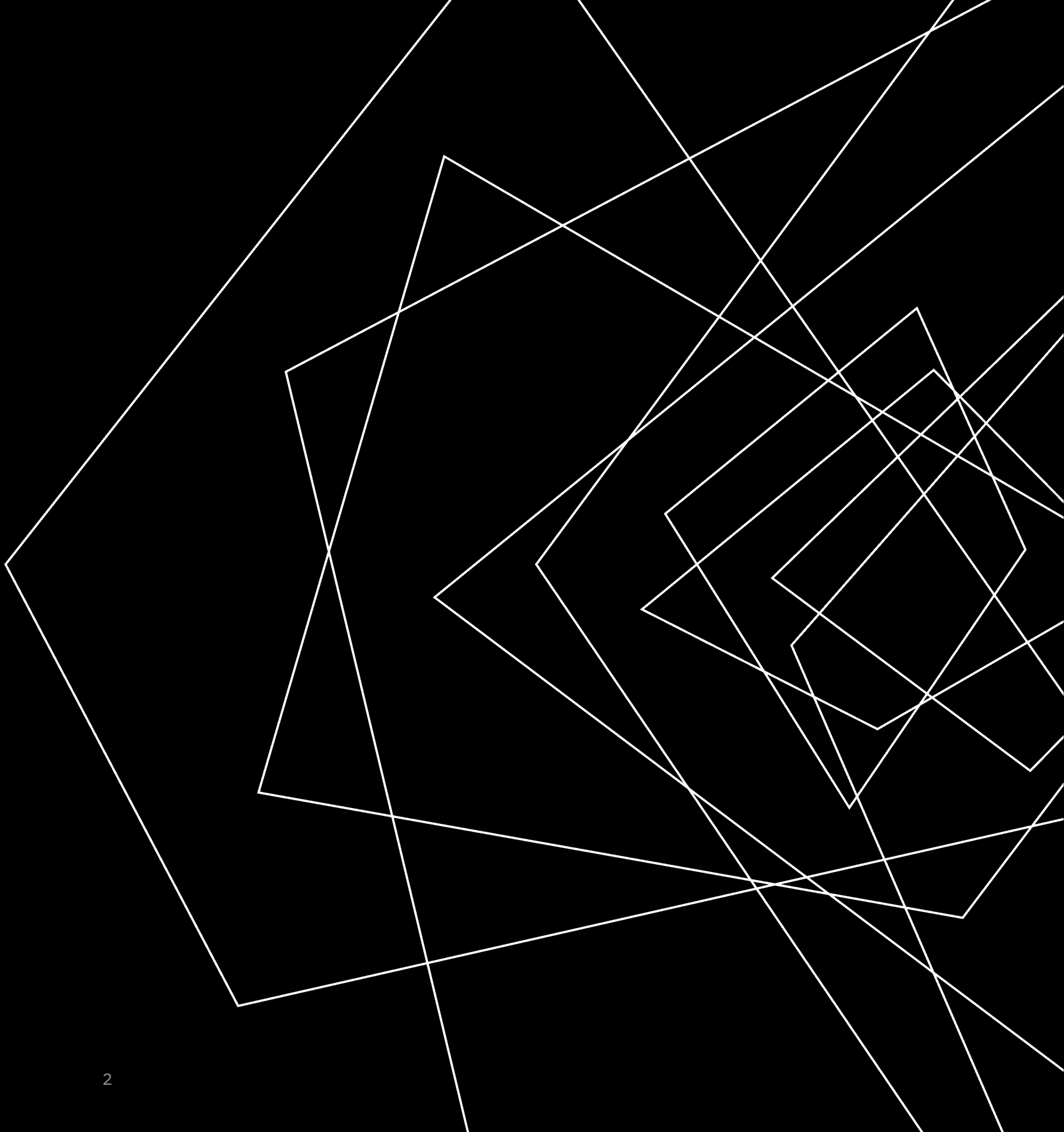
GRAPH ALGORITHMS

Keerat Kaur Guliani

TUSK Machine Learning Operations Course

AGENDA FOR TODAY

- Minimum Spanning Tree – Kruskal’s algorithm
- Shortest Path – Dijkstra’s algorithm
- Maximum Flow – Ford Fulkerson algorithm



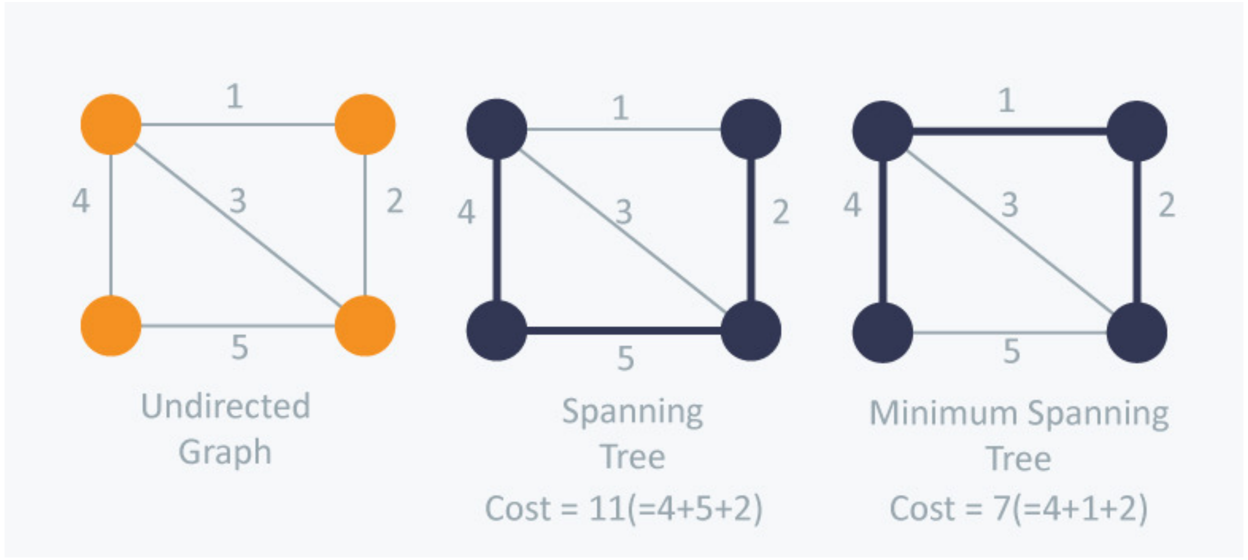
MINIMUM SPANNING TREE

What is a spanning tree?

A connected subgraph S of graph G (V, E) is said to be spanning iff:

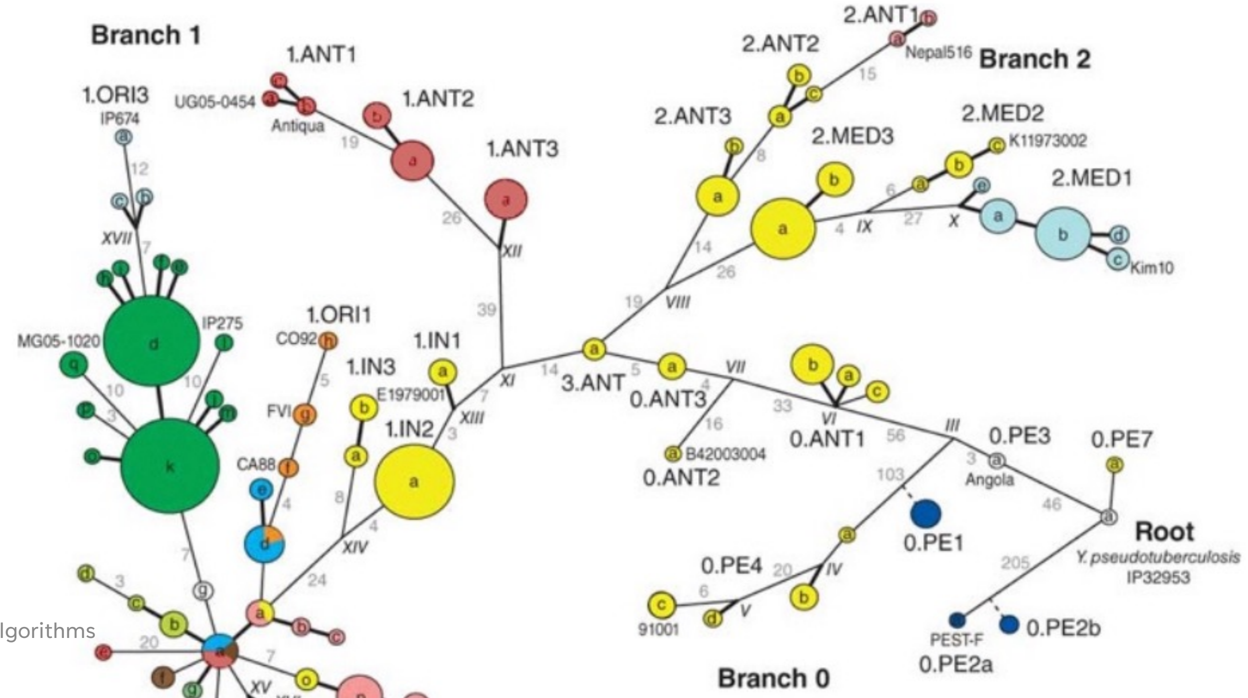
- S should contain all vertices of G
- S should contain (|v|-1) edges

A minimum spanning tree is a tree of minimum cost that connects all of its vertices.



Why MSTs?

- Network design
  - Connecting cities with roads/electricity/telephone/...
- cluster analysis
  - eg, genetic distance
- image processing
  - eg, image segmentation
- Useful primitive
  - for other graph algs



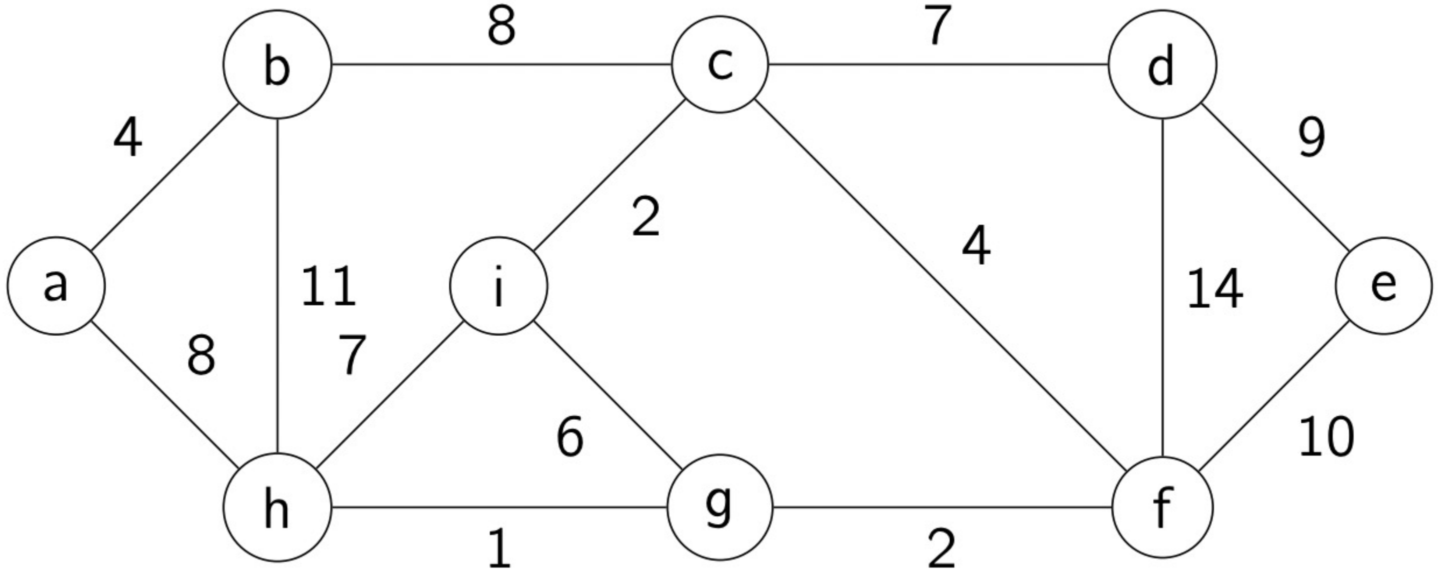
KRUSKAL’S ALGORITHM FOR MST - PSEUDOCODE

In the graph G (V, E):

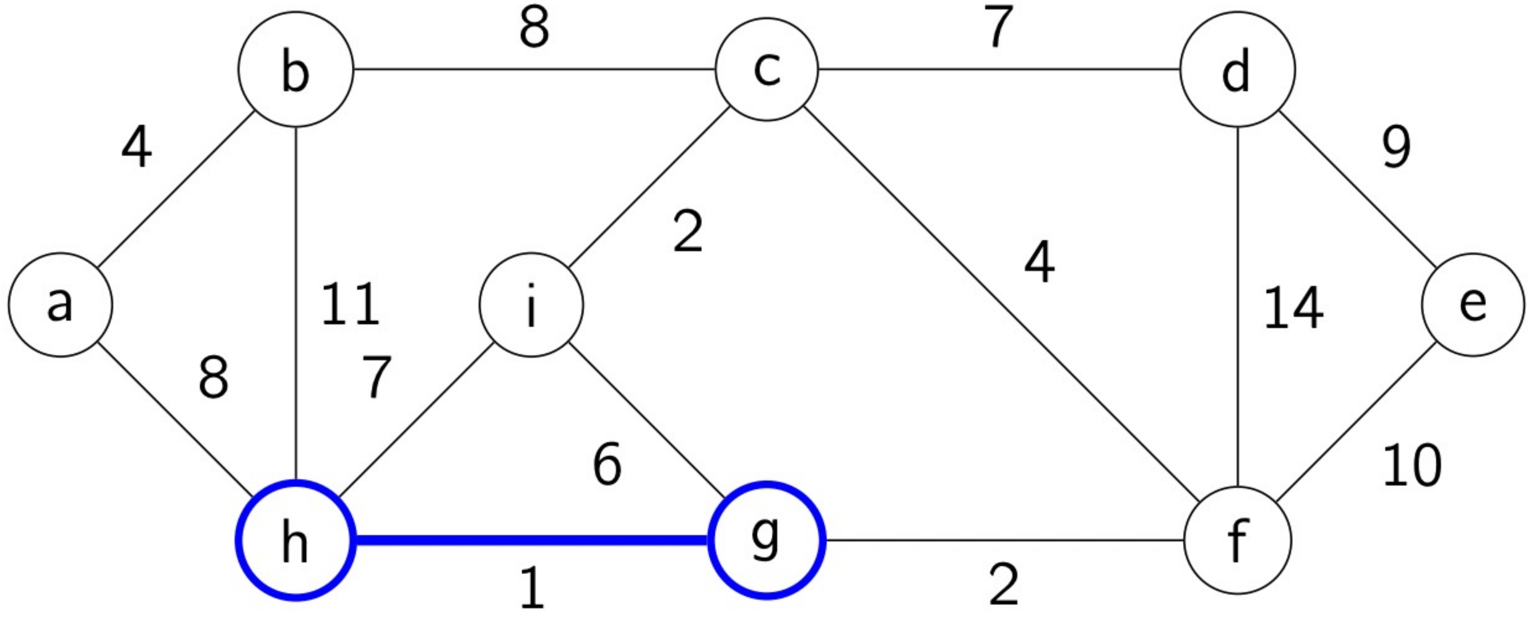
1. Construct a min heap with E edges.
2. Take the edge at the root node one by one and add to the spanning tree. Cycles should not be created.

Best case: (n-1) edges for heapify costing log E -> O(n log E)
Worst case: all E edges -> O (E log E)

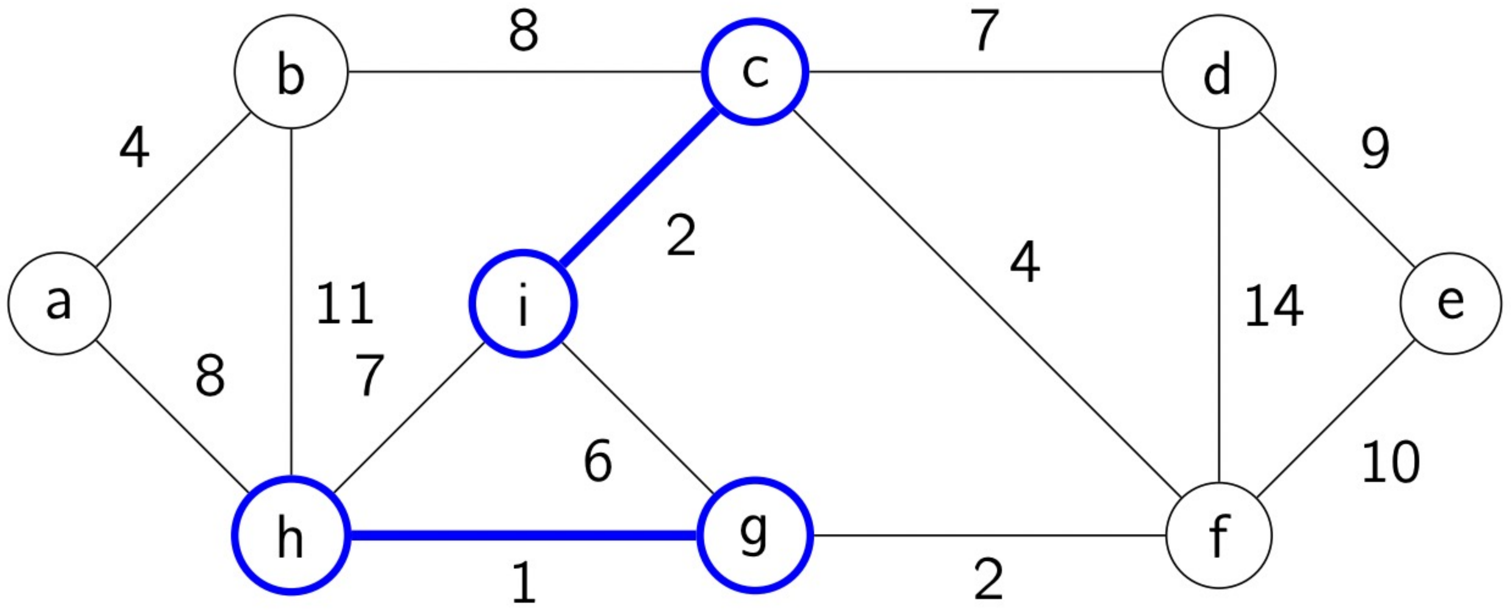
FOR EXAMPLE,



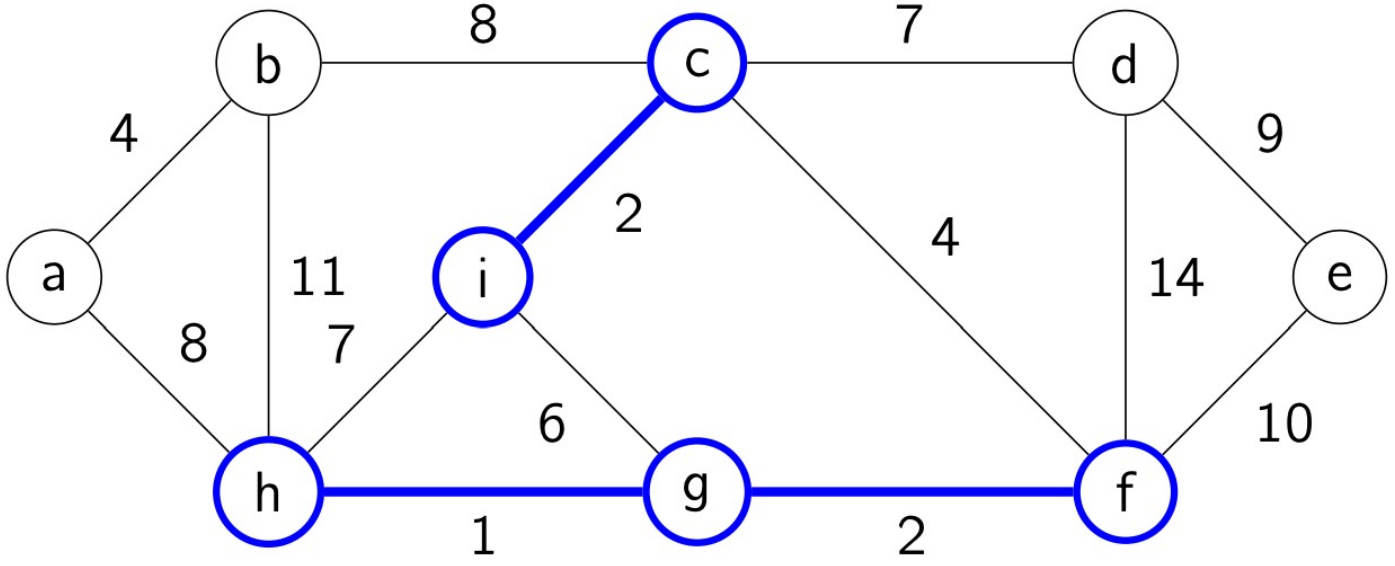
Example from Stanford's CS161 W23 lecture notes

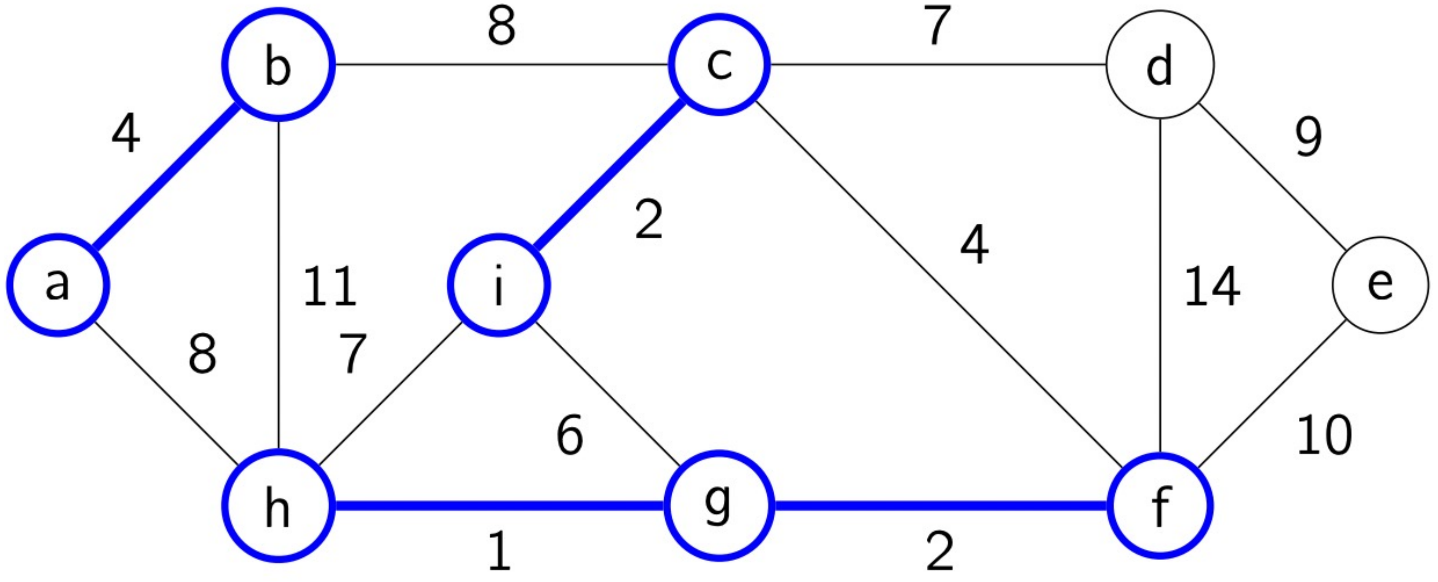


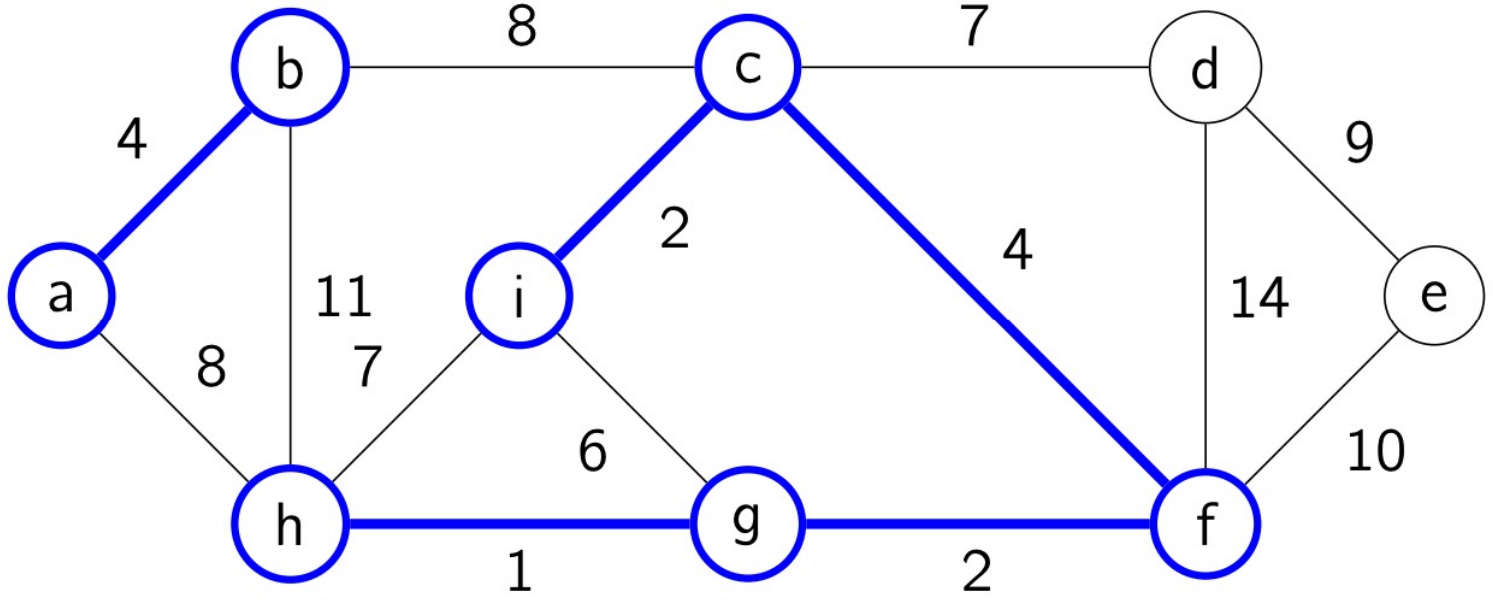
a
b
c
d
e
f
g
h
i
4
8
7
9
11
2
4
14
8
7
6
10
1
2

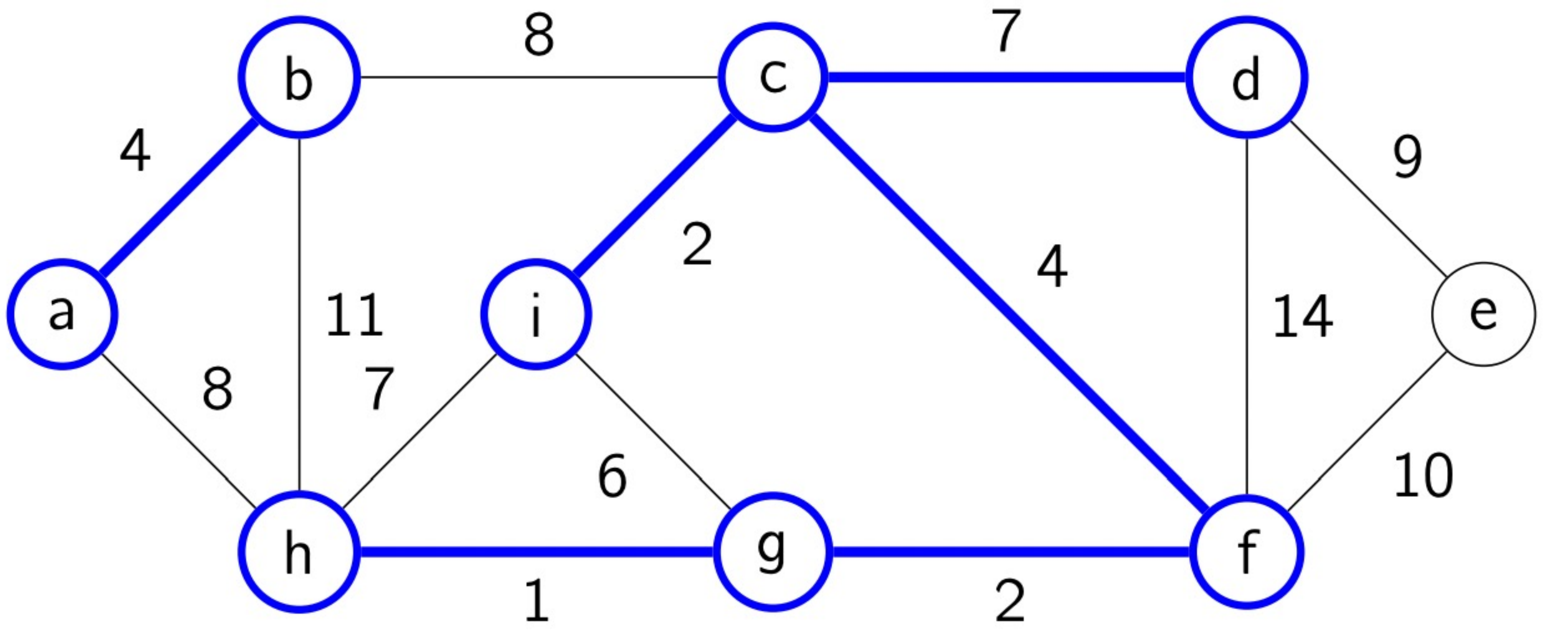


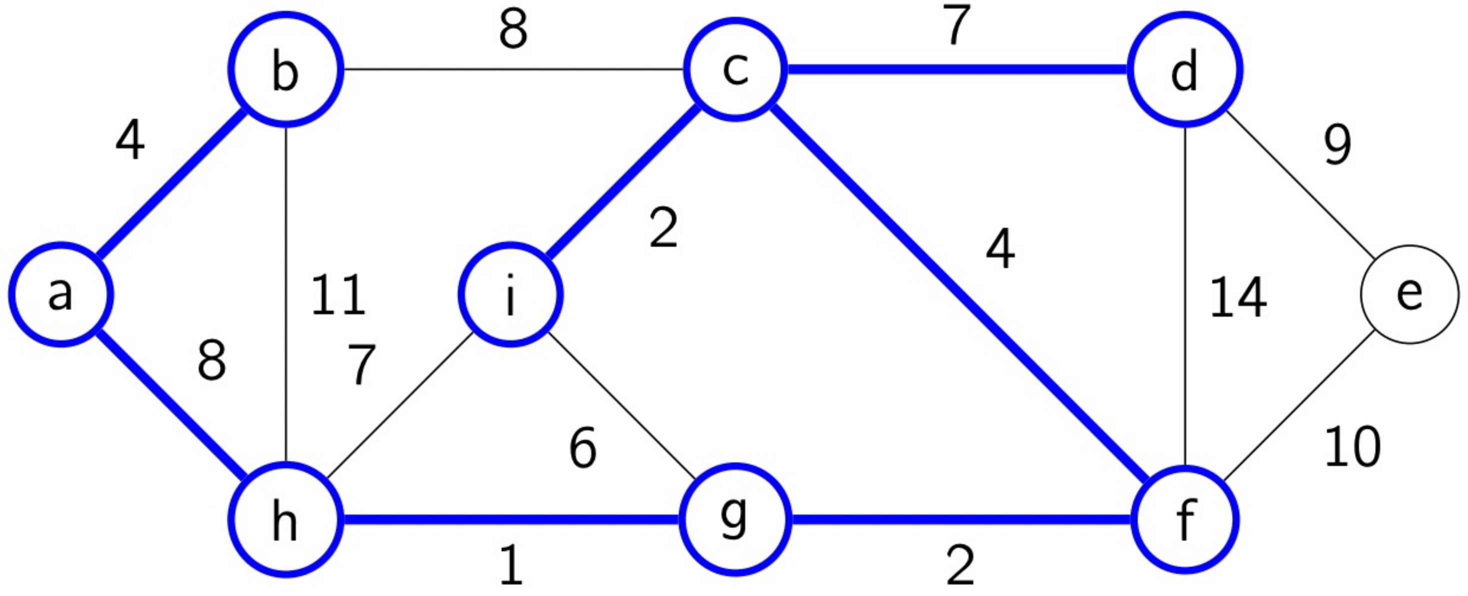
a
b
c
d
e
f
g
h
i
8
7
4
9
2
4
11
14
8
7
6
1
2
10


a
b
c
d
e
f
g
h
i
4
8
8
11
7
2
7
4
9
14
6
10
1
2

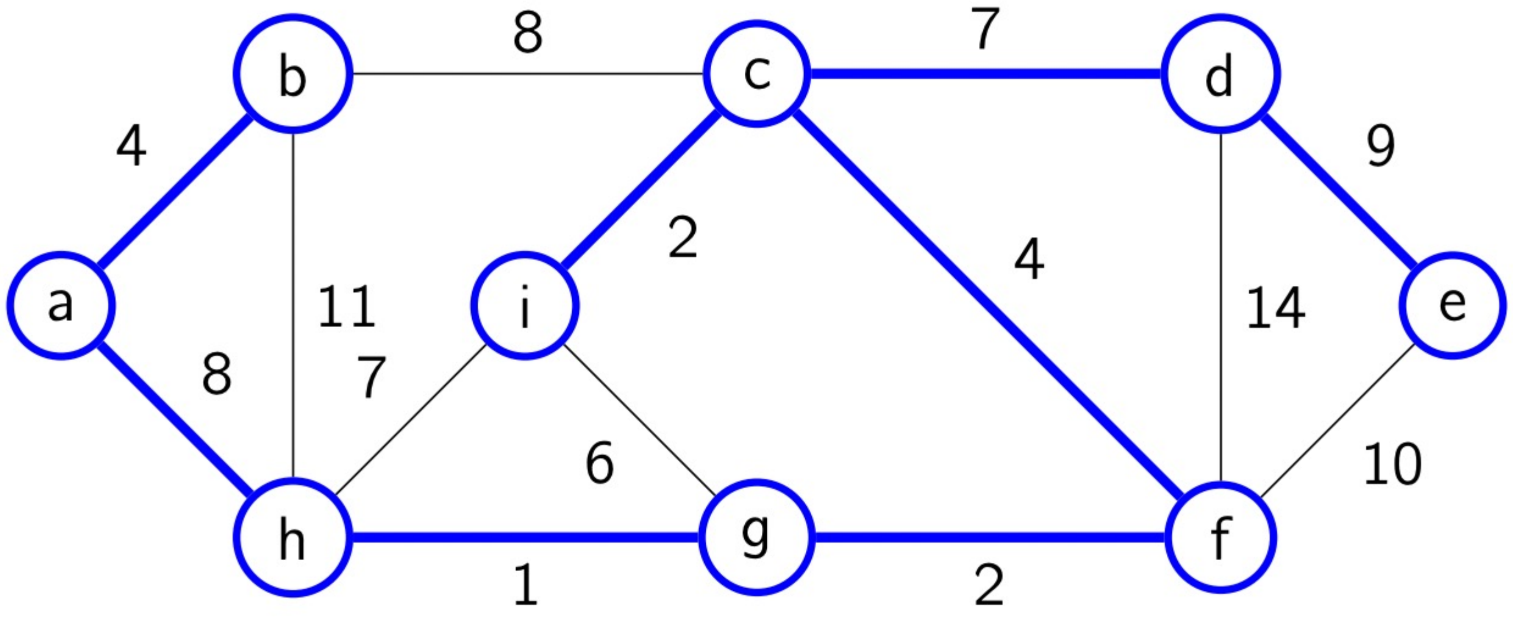

a
b
c
d
e
f
g
h
i
4
8
7
9
2
4
14
11
8
7
6
10
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
2
4
11
14
8
7
6
10
1
2


a
b
c
d
e
f
g
h
i
4
8
7
9
2
4
11
14
8
7
6
10
1
2


a
b
c
d
e
f
g
h
i
4
8
11
8
7
2
7
4
14
9
10
6
1
2

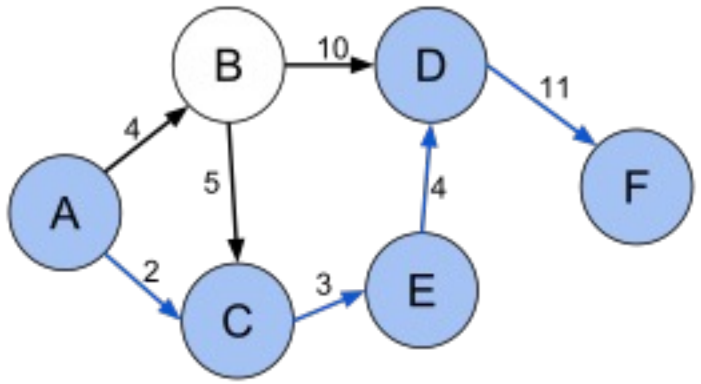
THE MST



SHORTEST PATH

What is the shortest path problem?

In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.



SHORTEST PATH - APPLICATIONS

1. Digital Mapping Services like Google Maps: Suppose you want to travel from on city to another city. You use Google maps to find the shortest route. How will it work? Assume the city you are in to be the source vertex and your destination to be another vertex.

There will still be many cities between your destination and starting point. Assume those cities to be intermediate vertices. The distance and traffic between any two cities can be assumed to be the weight between each pair of vertices. Google maps will apply Dijkstra algorithm on the city graph and find the shortest path.

2. IP Routing: Dijkstra's Algorithm can be used by link state routing protocols to find the best path to route data between routers.

DIJKSTRA'S ALGORITHM FOR SHORTEST PATH - PSEUDOCODE

- Dijkstra's algorithm is a graph algorithm for finding the shortest path from a source node to all other nodes in a graph(single source shortest path).

- Type of greedy algorithm

- Only works on weighted graphs with positive weights.

DIJKSTRA'S ALGORITHM FOR SHORTEST PATH - PSEUDOCODE

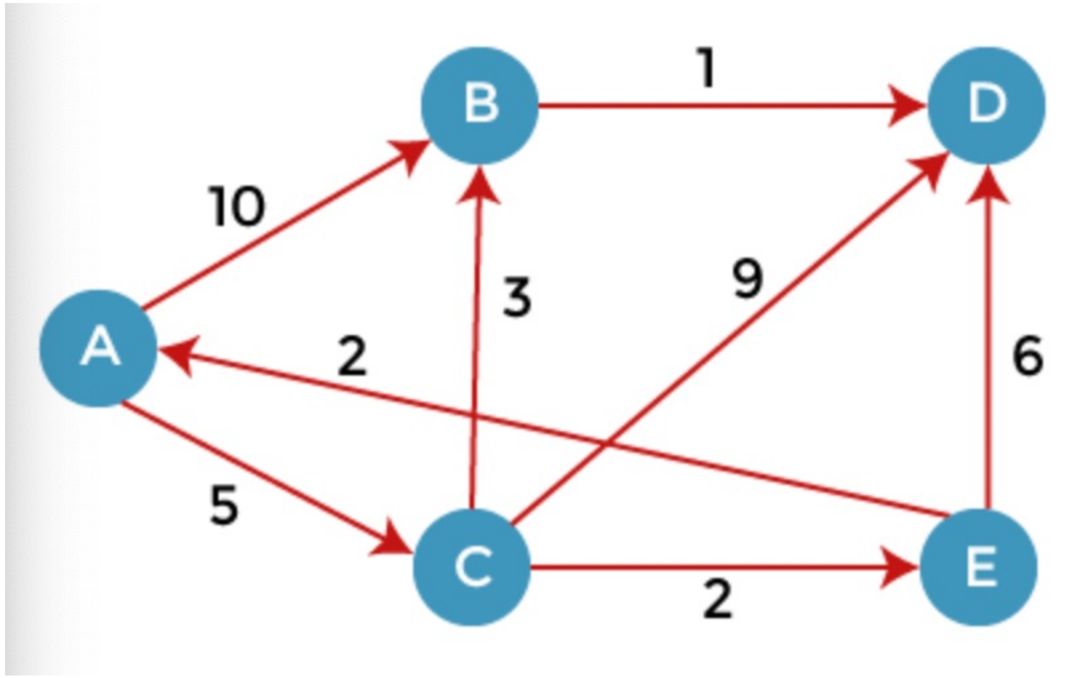
- Dijkstra's algorithm is a graph algorithm for finding the shortest path from a source node to all other nodes in a graph(single source shortest path).
- Type of greedy algorithm
- Only works on weighted graphs with positive weights.

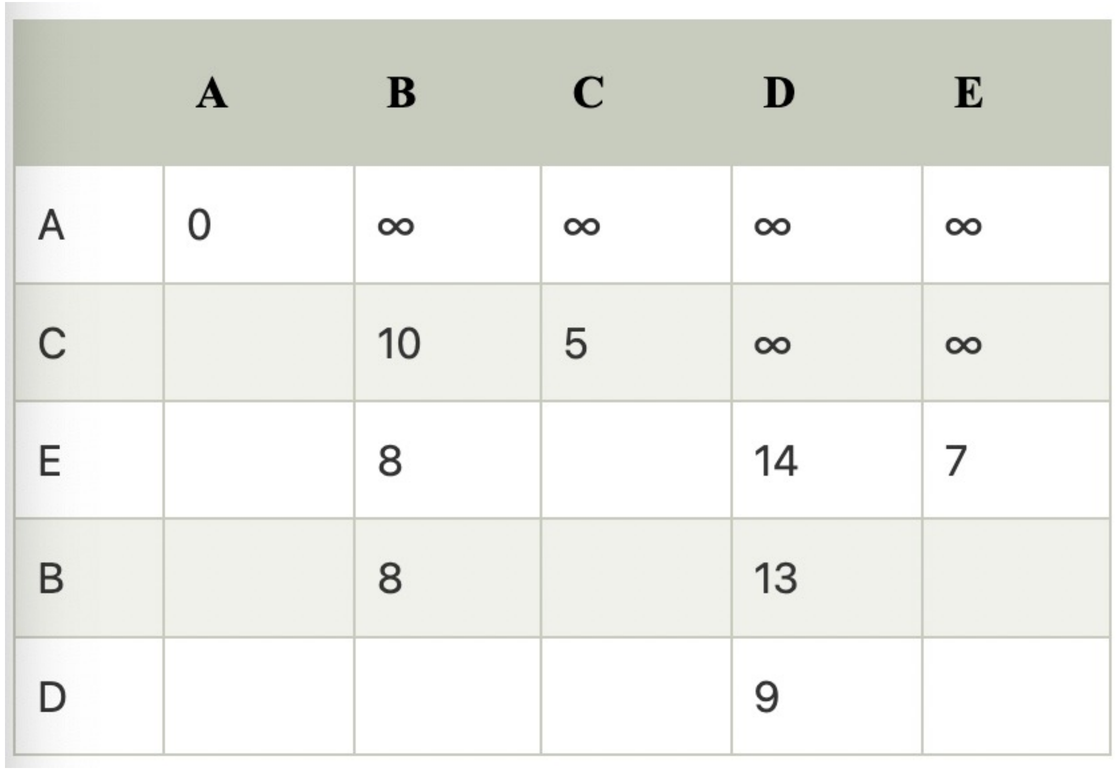
Time Complexity: For each vertex, search through all vertices to find the closest vertex within the graph -> O (V^2)

This is the least efficient implementation of Dijkstra! Trying using a priority queue instead of a set.

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:
        distance[v] = infinity

    distance[source] = 0
    G = the set of all nodes of the Graph

    while G is non-empty:
        Q = node in G with the least dist[ ]
        mark Q visited
        for each neighbor N of Q:
            temp_dist = distance[Q] + dist_between(Q, N)
            if temp_dist < distance[N]
                distance[N] := temp_dist

    return distance[ ]
```

SHORTEST PATH - EXAMPLE





	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	
D				9	

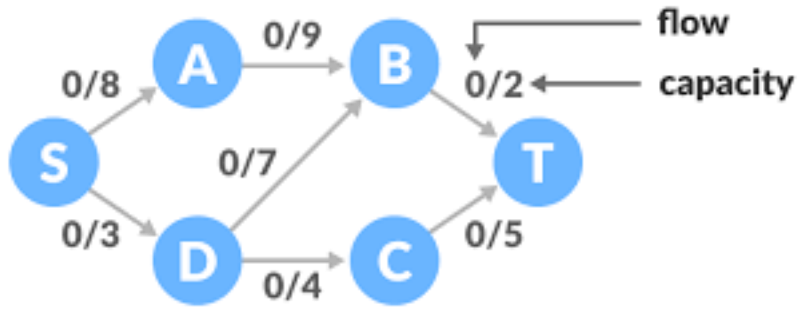
MAXIMUM FLOW PROBLEM

Given a graph which represents a flow network where every edge has a capacity. Also, given two vertices, source S and sink T in the graph, find out the maximum possible flow from S to T with the following constraints:

1. Flow on an edge does not exceed the given capacity of the edge
2. Inflow is equal to outflow for every vertex except S and T

Some applications:

1. Water flowing through a pipe
2. Maximum goods that can be circulated on a network of roads in response to a demand



MAXIMUM FLOW PROBLEM - TERMINOLOGIES

Residual Graph: A graph that indicates additional possible flow. If there is such a path from source to sink then there is a possibility to add flow.

Residual Capacity: Original capacity of the edge minus the flow through the edge.

Minimal Cut: Bottleneck capacity, which decides the maximum possible flow from source to sink through an augmenting path.

Augmenting Path: Augmenting path can be done in two ways:

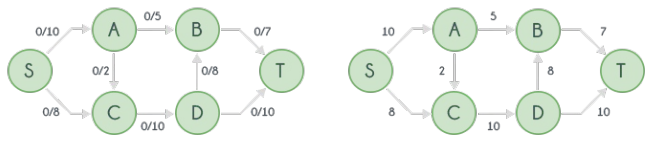
- Non-full forward edges
- Non-empty backward edges

MAXIMUM FLOW PROBLEM - PSEUDOCODE

```
1.Start with initial flow as 0.
2.While there exists an augmenting path from the source to the sink:
    1. Find an augmenting path using any path-finding algorithm, such as breadth-first search or depth-
       first search. You can also arbitrarily choose one.
    2. Determine the amount of flow that can be sent along the augmenting path, which is the minimum
       residual capacity along the edges of the path.
    3. Increase the flow along the augmenting path by the determined amount.
3.Return the maximum flow.
```

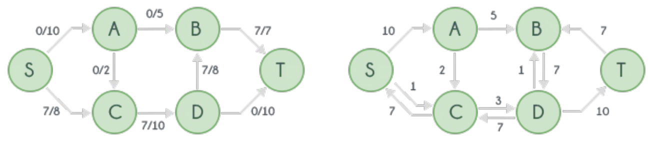
MAXIMUM FLOW PROBLEM - EXAMPLE

Network (G) **Residual Graph (G_R)**

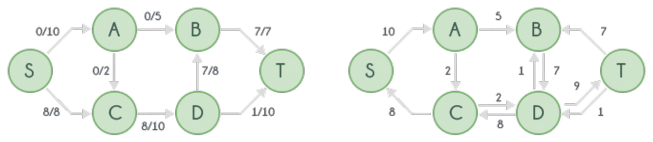


Flow = 0

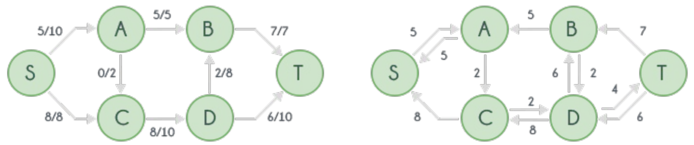
Path 1: S - C - D - B - T → Flow = Flow + 7



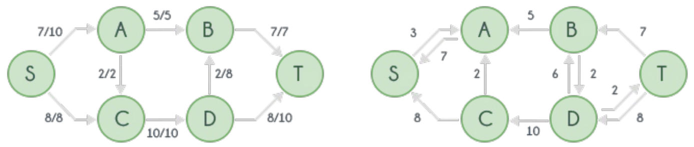
Path 2: S - C - D - T → Flow = Flow + 1



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



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



No More Paths Left
Max Flow = 15

MAXIMUM FLOW PROBLEM – COMPLEXITY

Steps:

1. Find an augmenting path: O (E) because DFS every time
2. Compute the bottleneck capacity: each augmenting path has $O(V)$ edges and each of those edges can have O(F) increases in its capacity in the worst case.
3. Augment each edge and total flow : O(F) each

Thus, overall time complexity = O (E * F) + O (V * F) = O (E * F)