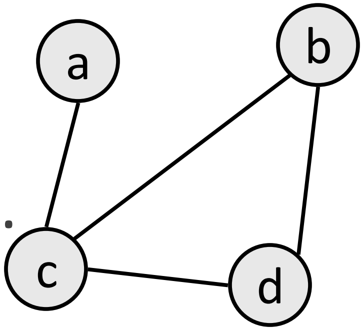
Graph – BFS & DFS

Graphs

- **graph**: A data structure containing:
  - a set of **vertices** *V*, *(sometimes called nodes)*
  - a set of **edges** *E*, where an edge represents a connection between 2 vertices.
    - Graph $G = (V, E)$
    - an edge is a pair (v, w) where v, w are in V

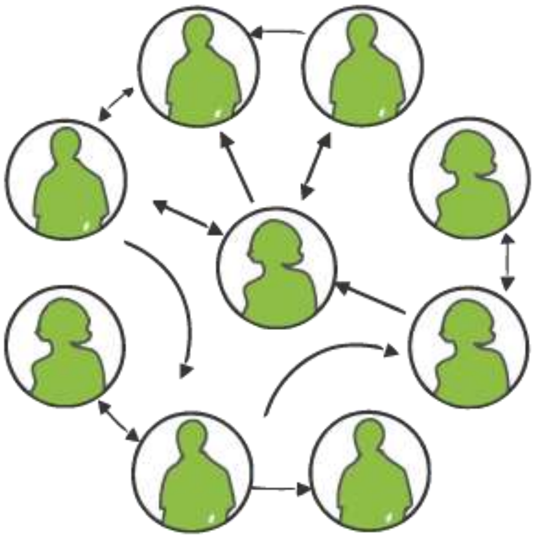


- the graph at right:
  - $V = \{a, b, c, d\}$
  - $E = \{(a, c), (b, c), (b, d), (c, d)\}$

- **degree**: number of edges touching a given vertex.
  - at right: a=1, b=2, c=3, d=2

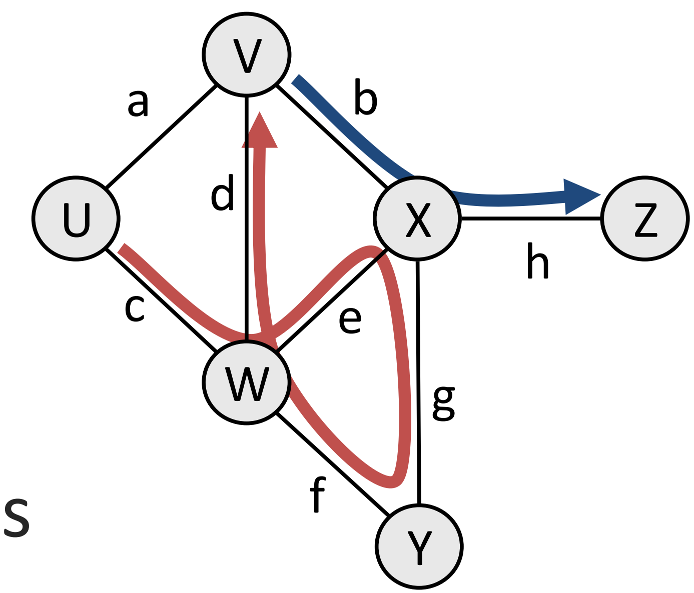
Graph examples

- For each, what are the vertices and what are the edges?
  - Web pages with links
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Facebook friends
  - Course pre-requisites
  - Family trees
  - Paths through a maze



Paths

- **path**: A path from vertex *a* to *b* is a sequence of edges that can be followed starting from *a* to reach *b*.
  - can be represented as vertices visited, or edges taken
  - example, one path from *V* to *Z*: {b, h} or {V, X, Z}
  - What are two paths from U to Y?

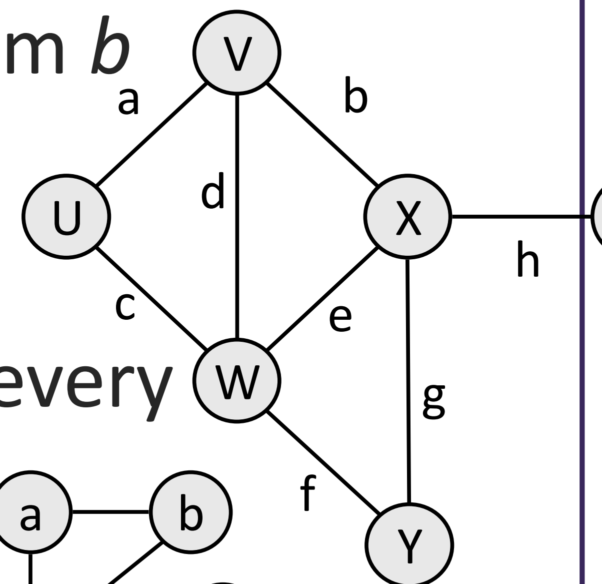


- **path length**: Number of vertices or edges contained in the path.

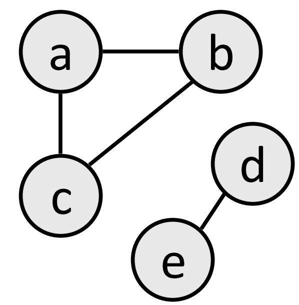
- **neighbor** or **adjacent**: Two vertices connected directly by an edge.
  - example: V and X

Reachability, connectedness

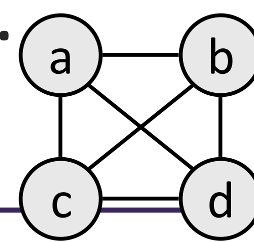
- **reachable**: Vertex *a* is *reachable* from *b* if a path exists from *a* to *b*.



- **connected**: A graph is *connected* if every vertex is reachable from any other.
  – Is the graph at top right connected?

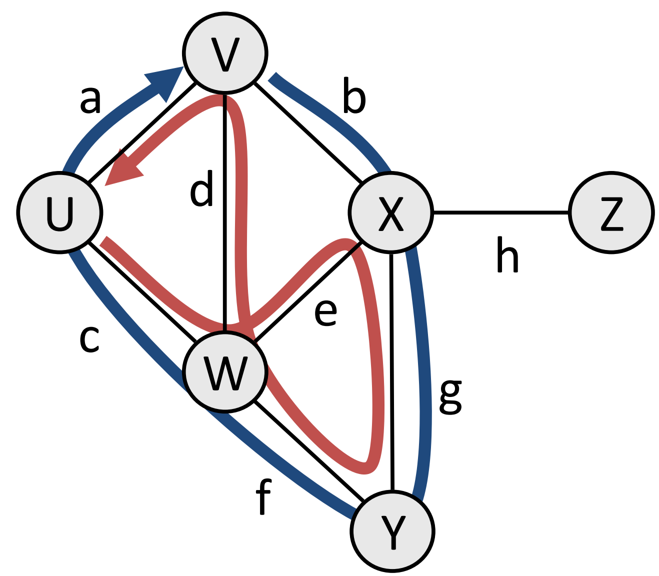


- **strongly connected**: When every vertex has an edge to every other vertex.



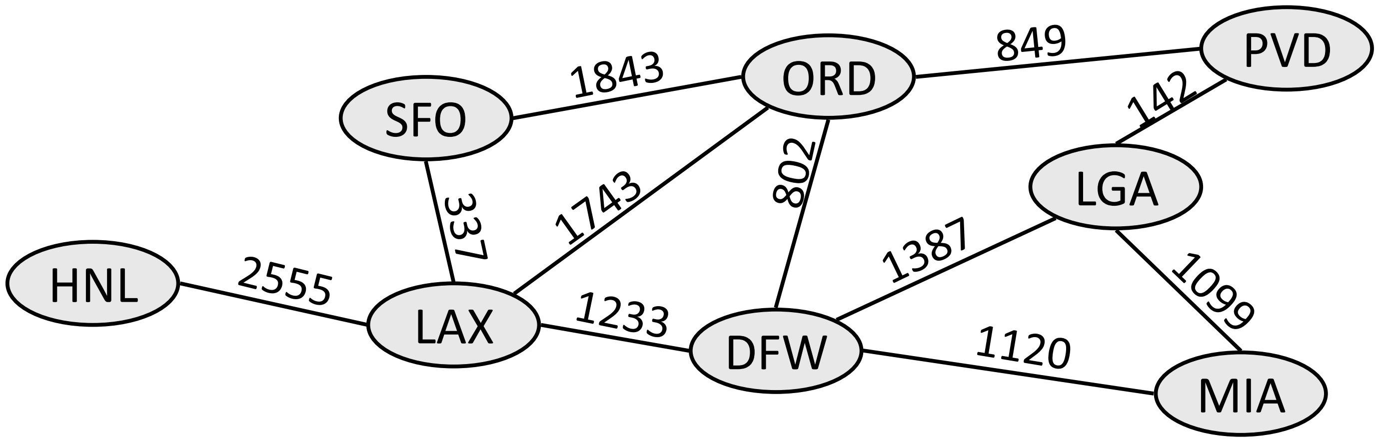
Loops and cycles

- **cycle**: A path that begins and ends at the same node.
  - example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
  - example: {c, d, a} or {U, W, V, U}.
  - **acyclic graph**: One that does not contain any cycles.
- **loop**: An edge directly from a node to itself.
  - Many graphs don't allow loops.



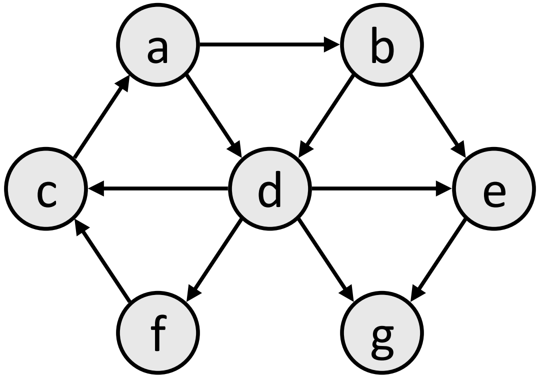
Weighted graphs

- **weight**: Cost associated with a given edge.
  - Some graphs have weighted edges, and some are unweighted.
  - Edges in an unweighted graph can be thought of as having equal weight (e.g. all 0, or all 1, etc.)
  - Most graphs do not allow negative weights.



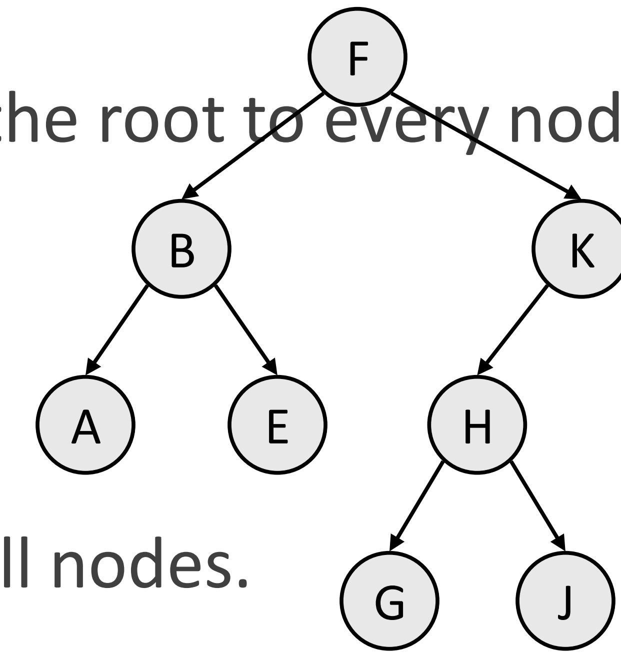
Directed graphs

- **directed graph** ("digraph"): One where edges are *one-way* connections between vertices.
  – If graph is directed, a vertex has a separate in/out degree.
  – A digraph can be weighted or unweighted.
  – Is the graph below connected? Why or why not?

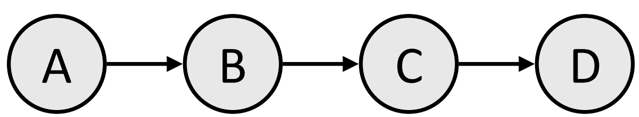


Linked Lists, Trees, Graphs

- A *binary tree* is a graph with some restrictions:
  - The tree is an unweighted, directed, acyclic graph (DAG).
  - Each node's in-degree is at most 1, and out-degree is at most 2.
  - There is exactly one path from the root to every node.



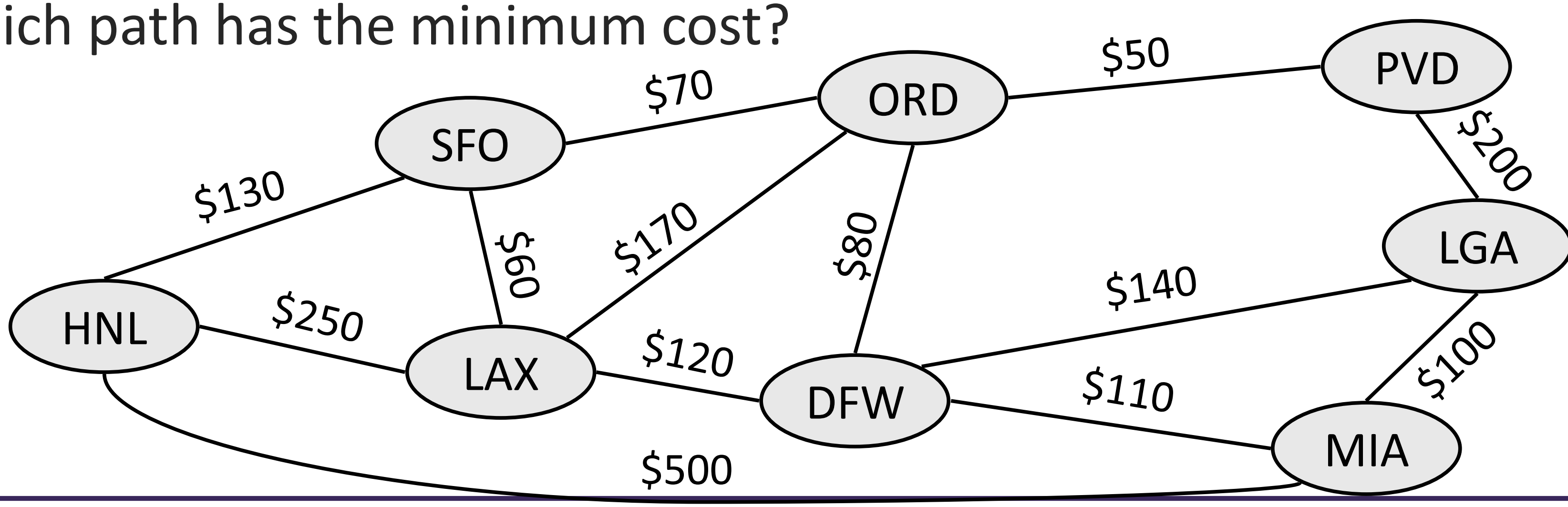
- A *linked list* is also a graph:
  - Unweighted DAG.
  - In/out degree of at most 1 for all nodes.



Searching for paths

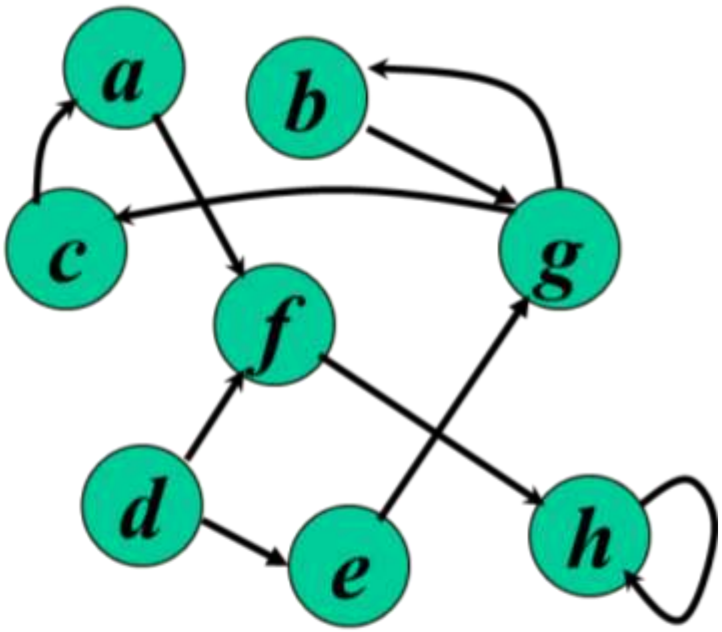
- Searching for a path from one vertex to another:
  - Sometimes, we just want *any* path (or want to know there *is* a path).
  - Sometimes, we want to minimize path *length* (# of edges).
  - Sometimes, we want to minimize path *cost* (sum of edge weights).

- What is the shortest path from MIA to SFO?
  Which path has the minimum cost?

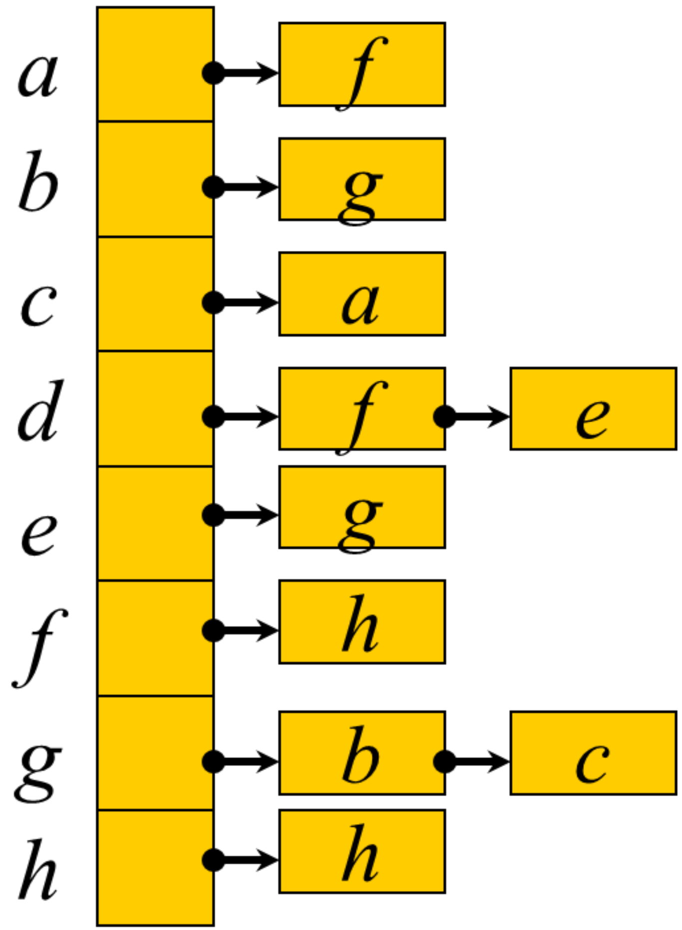


Adjacency matrix

	a	b	c	d	e	f	g	h
a	0	0	0	0	0	1	0	0
b	0	0	0	0	0	0	1	0
c	1	0	0	0	0	0	0	0
d	0	0	0	0	1	1	0	0
e	0	0	0	0	0	0	1	0
f	0	0	0	0	0	0	0	1
g	0	1	1	0	0	0	0	0
h	0	0	0	0	0	0	0	1

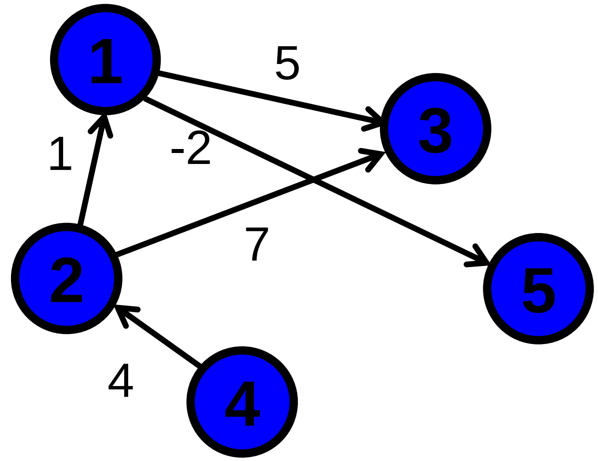


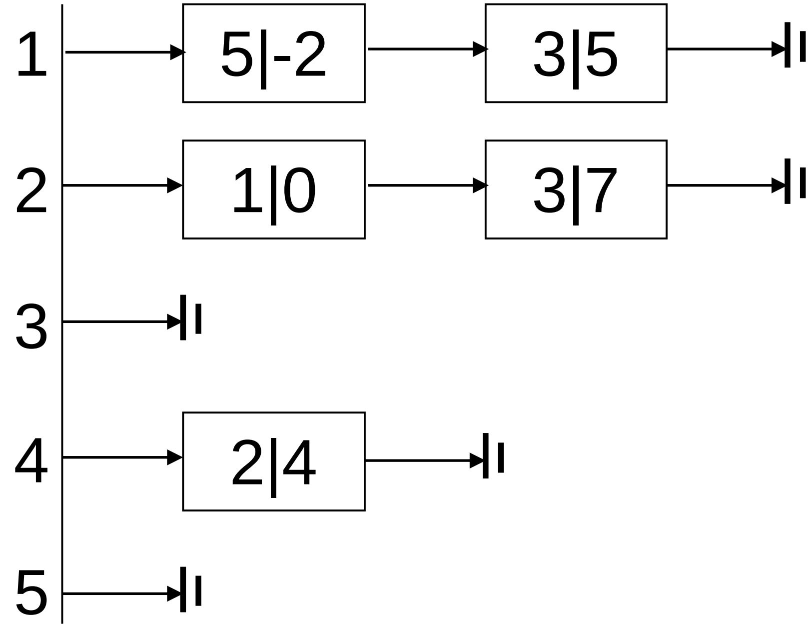
Adjacency List



weighted graph

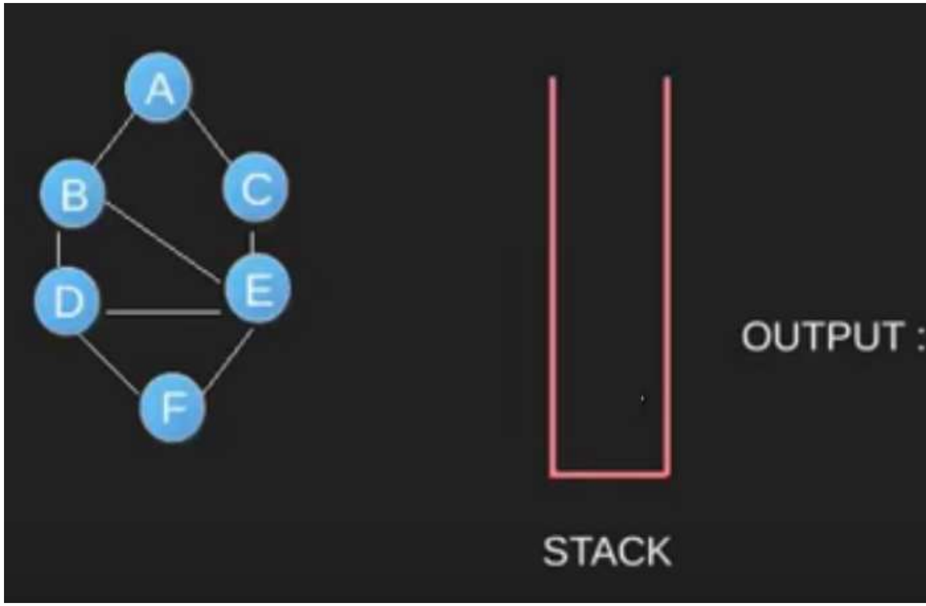
s\t	1	2	3	4	5
1			5		-2
2	1		7		
3					
4		4			
5					

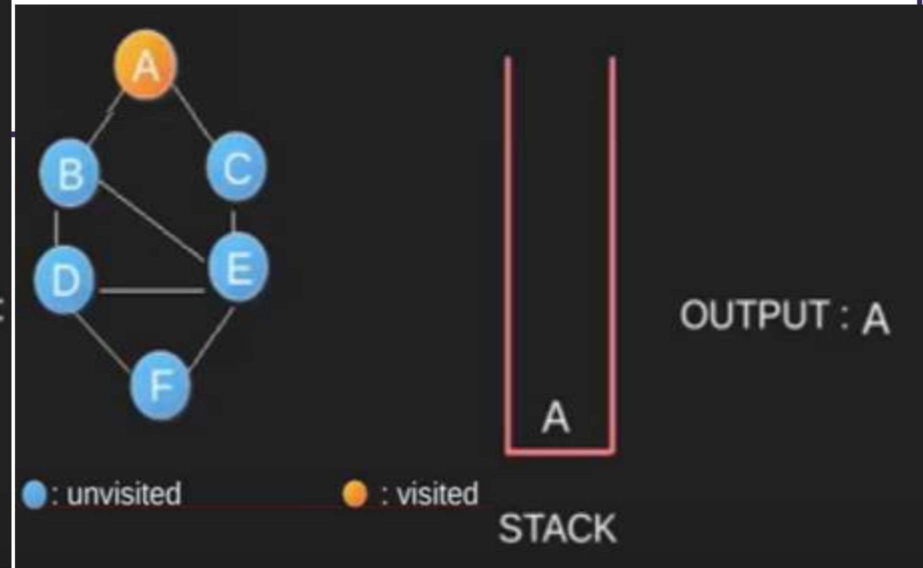


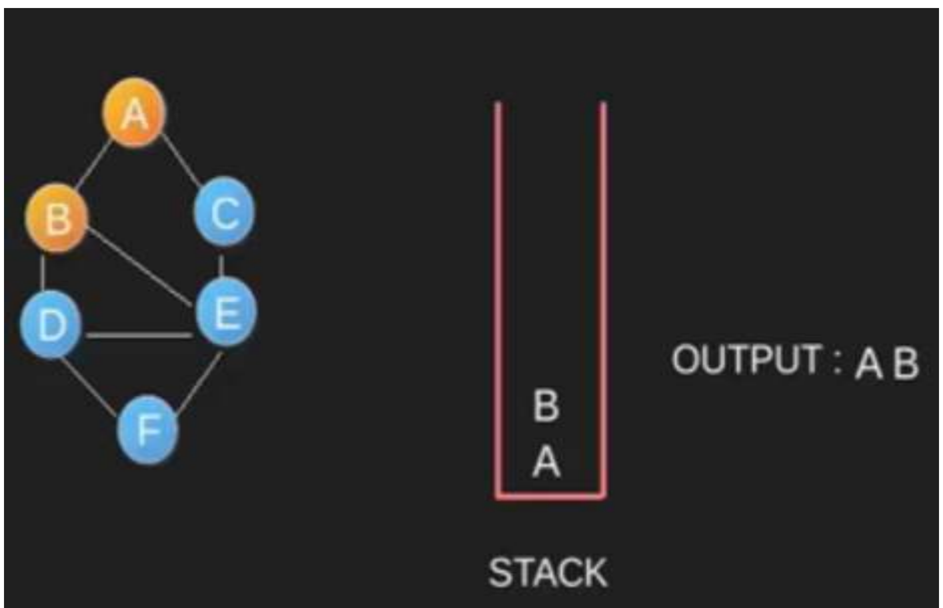


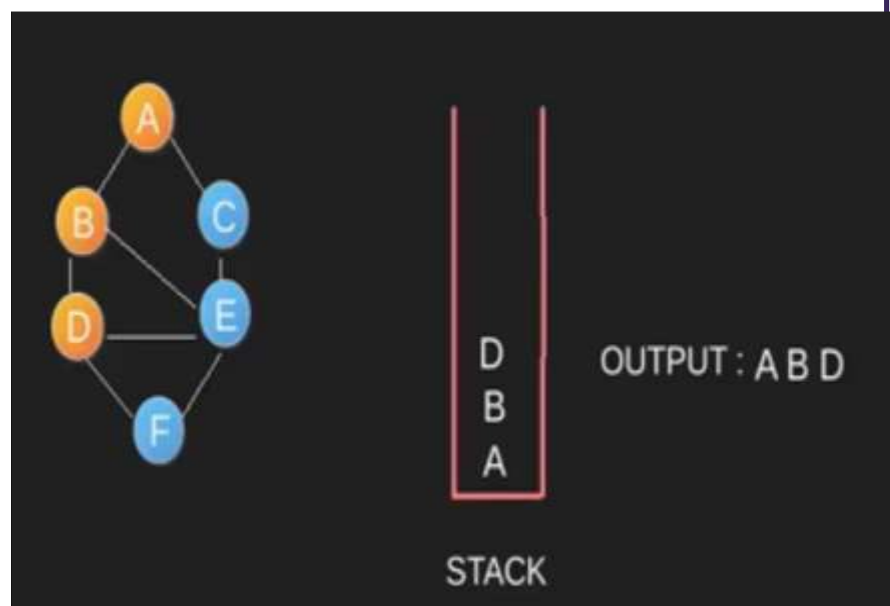
Depth-first search

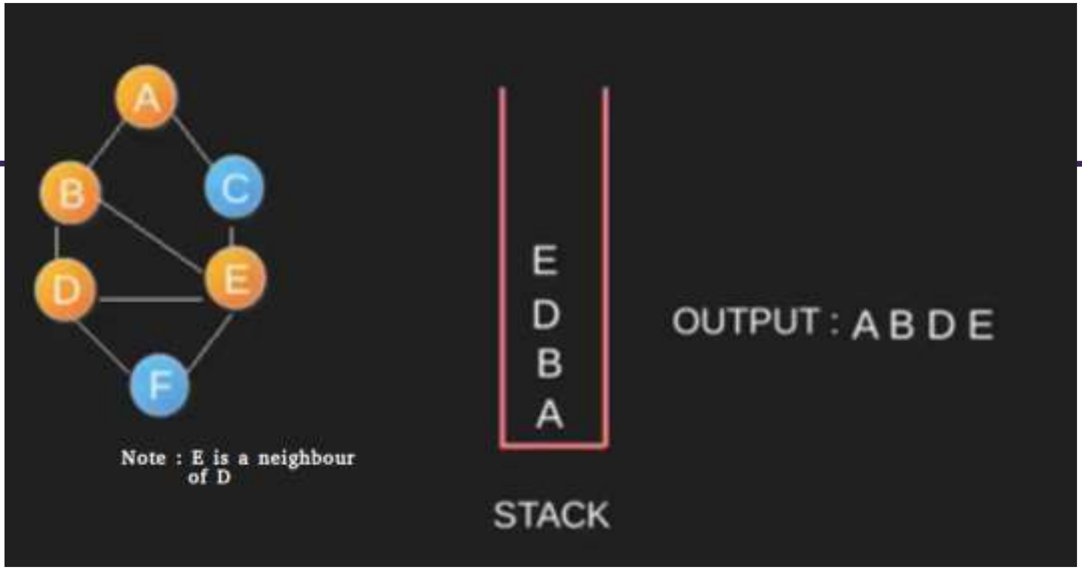
- **depth-first search** (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking.
  - Often implemented recursively.
  - Many graph algorithms involve *visiting* or *marking* vertices.

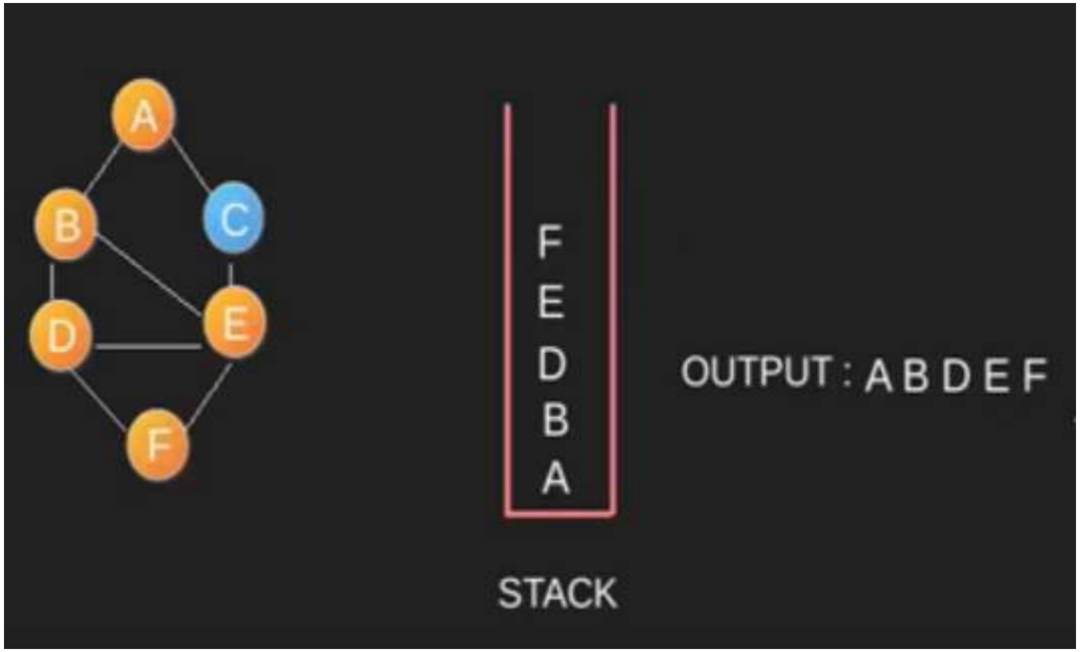

A
B
C
D
E
F
STACK
OUTPUT :

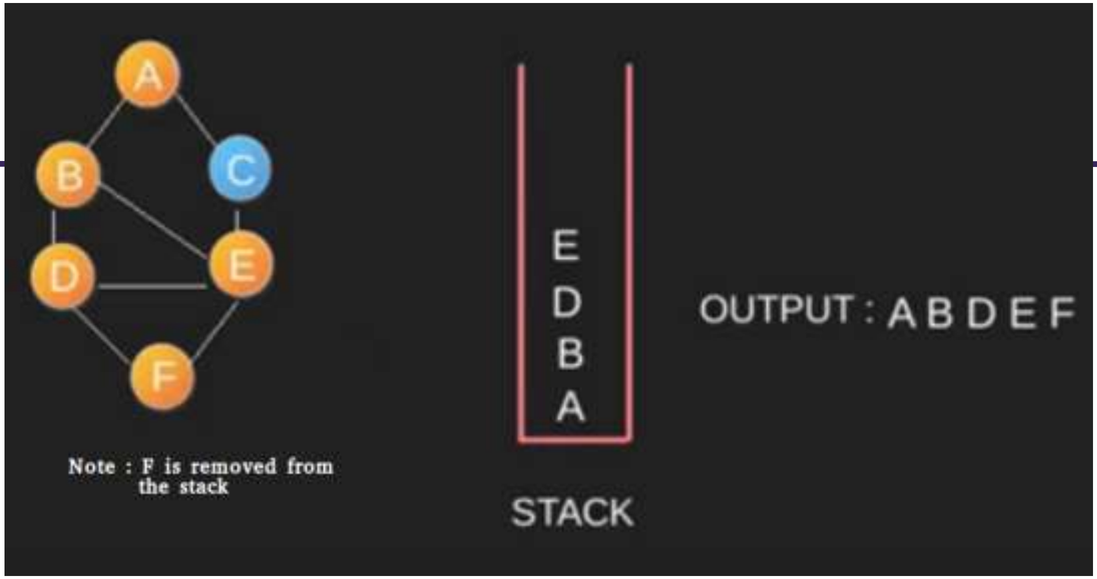

A
B
C
D
E
F
A
STACK
OUTPUT : A
: unvisited
: visited

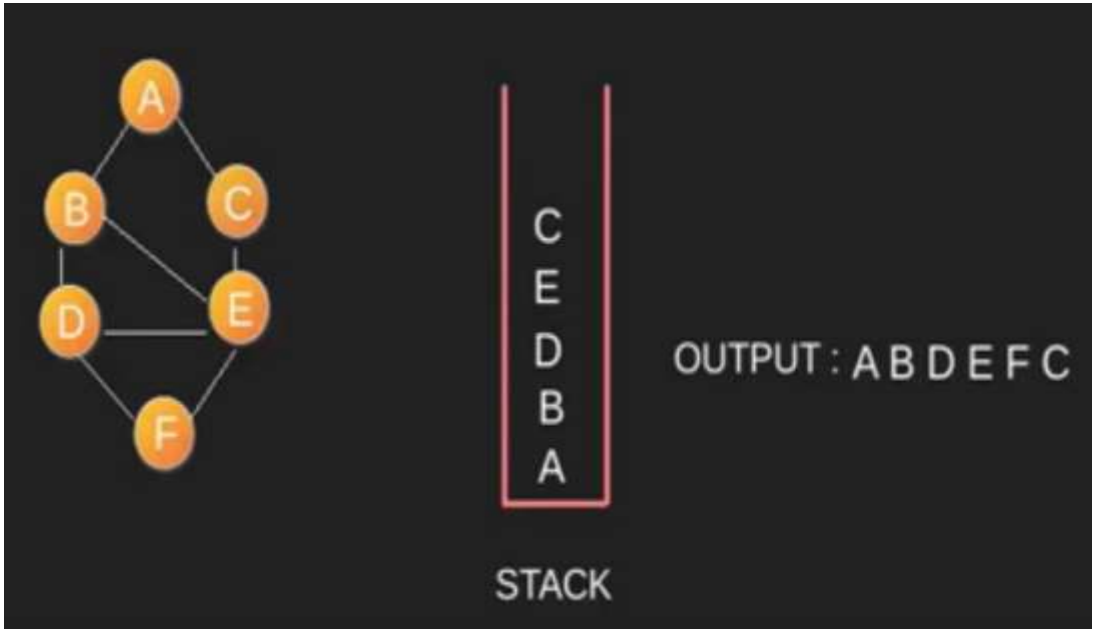

A
B
C
D
E
F
B
A
STACK
OUTPUT : A B

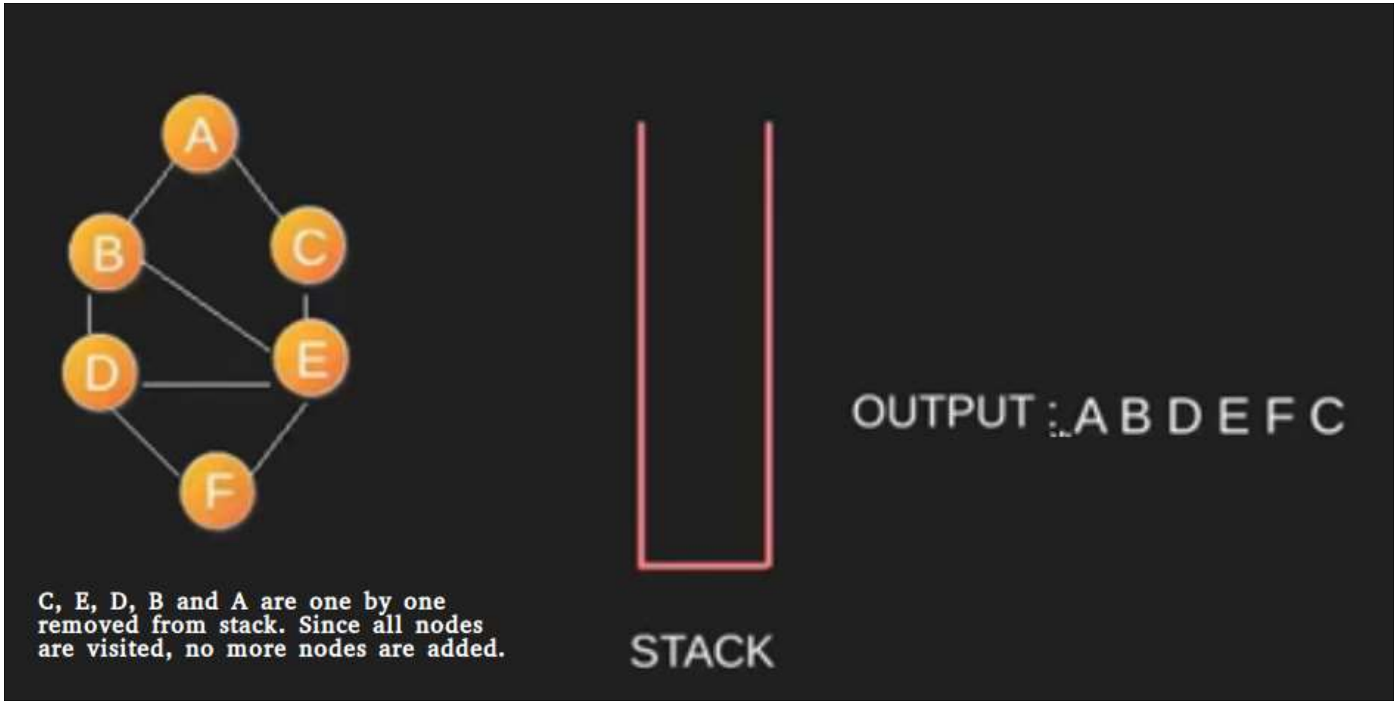

A
B
C
D
E
F
D
B
A
STACK
OUTPUT : A B D


A
B
C
D
E
F
Note : E is a neighbour of D
E
D
B
A
STACK
OUTPUT : A B D E


A
B
C
D
E
F
F
E
D
B
A
STACK
OUTPUT : A B D E F


A
B
C
D
E
F
Note : F is removed from the stack
E
D
B
A
STACK
OUTPUT : A B D E F


A
B
C
D
E
F
C
E
D
B
A
STACK
OUTPUT : A B D E F C

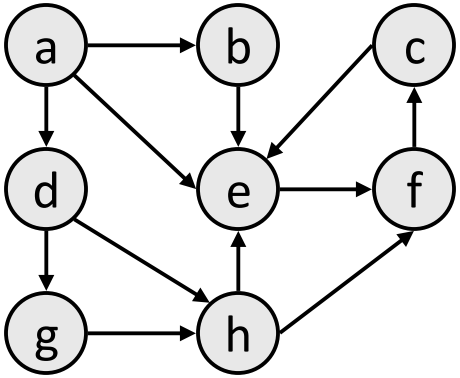

A
B
C
D
E
F
C, E, D, B and A are one by one removed from stack. Since all nodes are visited, no more nodes are added.
STACK
OUTPUT : A B D E F C

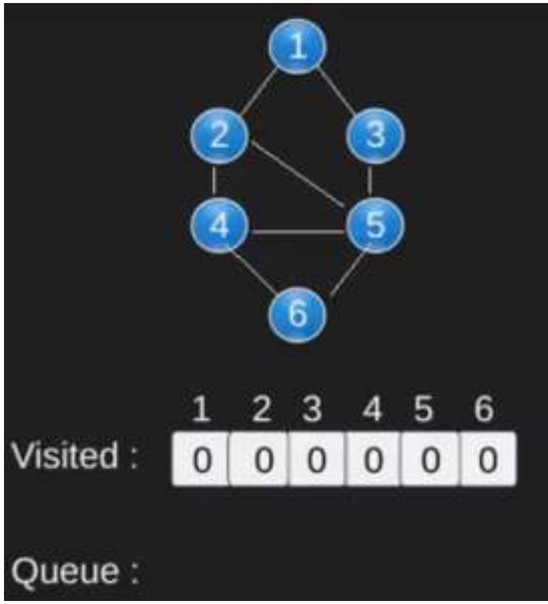
DFS pseudocode

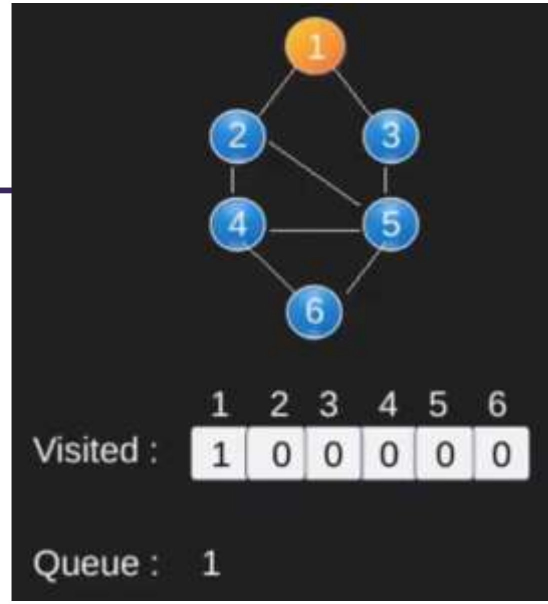
```
DFS-iterative (G, s):
//Where G is graph and s is source vertex let S be stack
   S.push( s )
   //Inserting s in stack mark s as visited.
   while ( S is not empty):
   //Pop a vertex from stack to visit next
     v = S.top( )
     S.pop( )
     //Push all the neighbours of v in stack that are not visited
     for all neighbours w of v in Graph G:
               if w is not visited :
               S.push( w )
               mark w as visited
```

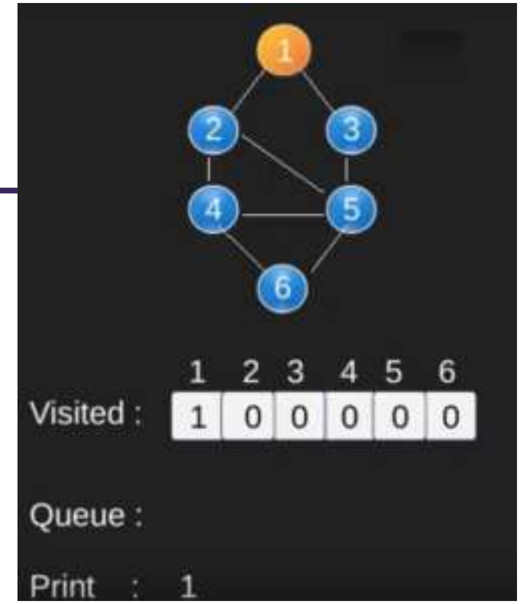
Breadth-first search

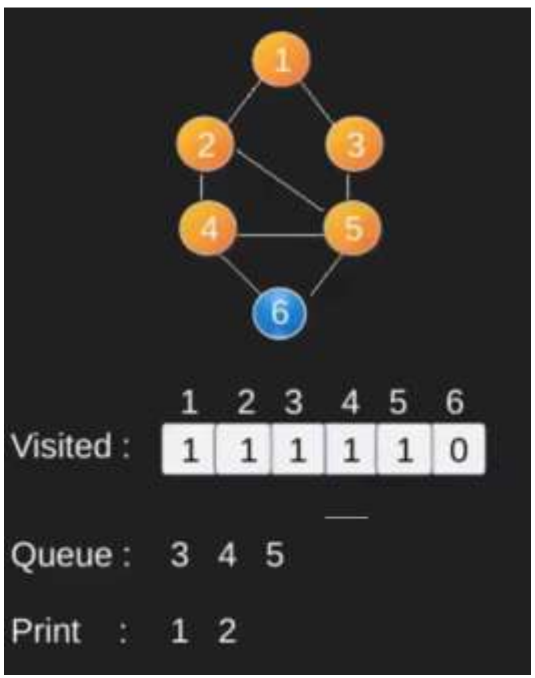
- **breadth-first search** (BFS): Finds a path between two nodes by taking one step down all paths and then immediately backtracking.
  - Often implemented by maintaining a queue of vertices to visit.

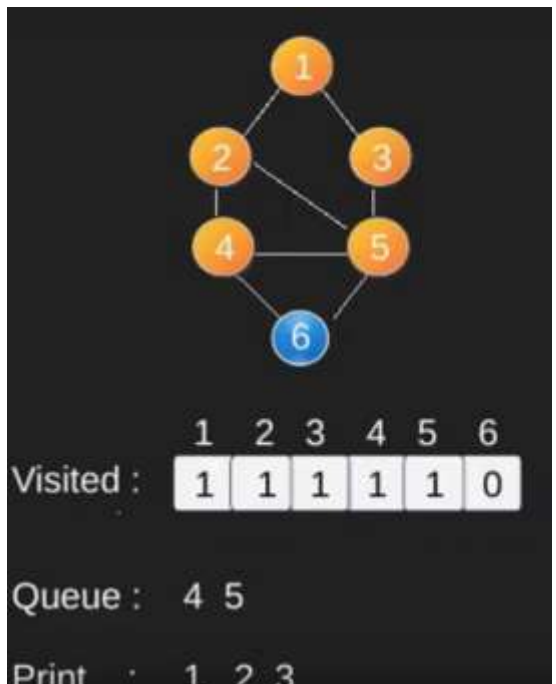


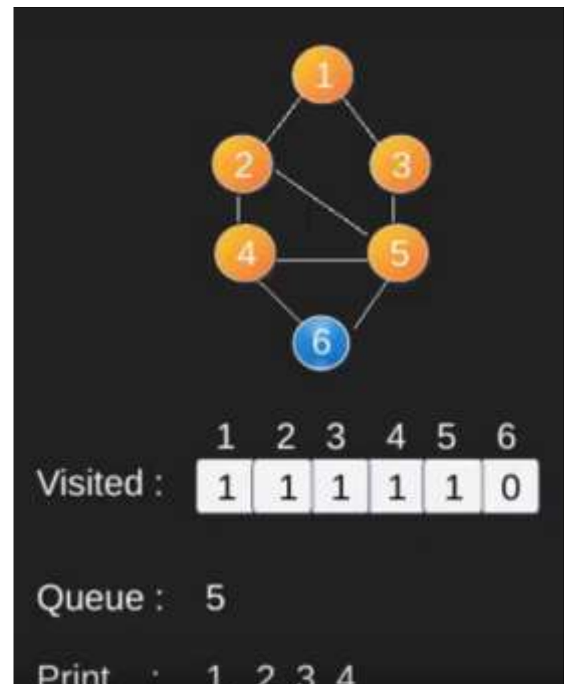

1
2
3
4
5
6
1 2 3 4 5 6
Visited : 0 0 0 0 0 0
Queue :

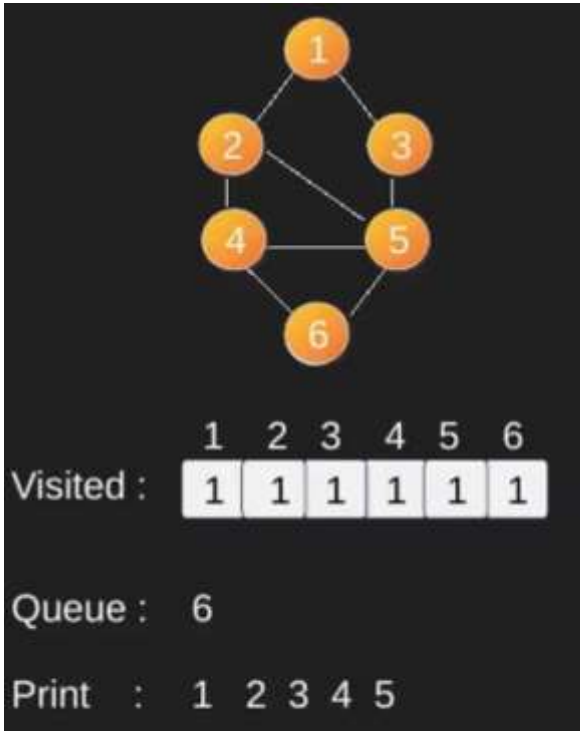

1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 0 0 0 0 0
Queue : 1

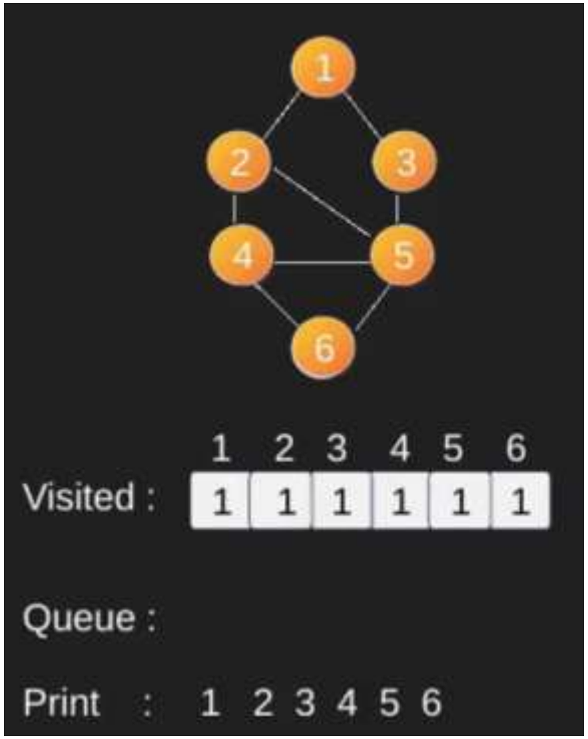

1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 0 0 0 0 0
Queue :
Print : 1


1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 1 1 0
Queue : 3 4 5
Print : 1 2


1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 1 1 0
Queue : 4 5
Print : 1 2 3


1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 1 1 0
Queue : 5
Print : 1 2 3 4


1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 1 1 1
Queue : 6
Print : 1 2 3 4 5


1
2
3
4
5
6
1 2 3 4 5 6
Visited : 1 1 1 1 1 1
Queue :
Print : 1 2 3 4 5 6

BFS pseudocode

```
BFS (G, s)
//Where G is the graph and s is the source node let Q be queue.
        Q.enqueue( s )
//Inserting s in queue until all its neighbour vertices are marked.
mark s as visited.
        while ( Q is not empty)
//Removing that vertex from queue,whose neighbour will be visited now
                v = Q.dequeue( )
//processing all the neighbours of v
    for all neighbours w of v in Graph G
    if w is not visited
        Q.enqueue( w )
//Stores w in Q to further visit its neighbour
        mark w as visited.
```

BFS observations

- *optimality*:
  - always finds the shortest path (fewest edges).
  - in unweighted graphs, finds optimal cost path.
  - In weighted graphs, *not* always optimal cost.
- *retrieval*: harder to reconstruct the actual sequence of vertices or edges in the path once you find it
  - conceptually, BFS is exploring many possible paths in parallel, so it's not easy to store a path array/list in progress
  - solution: We can keep track of the path by storing predecessors for each vertex (each vertex can store a reference to a *previous* vertex).
- DFS uses less memory than BFS, easier to reconstruct the path once found; but DFS does not always find shortest path. BFS does.

DFS, BFS runtime

- What is the expected runtime of DFS and BFS, in terms of the number of vertices V and the number of edges E ?
- Answer: O(|V| + |E|)
  - where |V| = number of vertices, |E| = number of edges
  - Must potentially visit every node and/or examine every edge once.

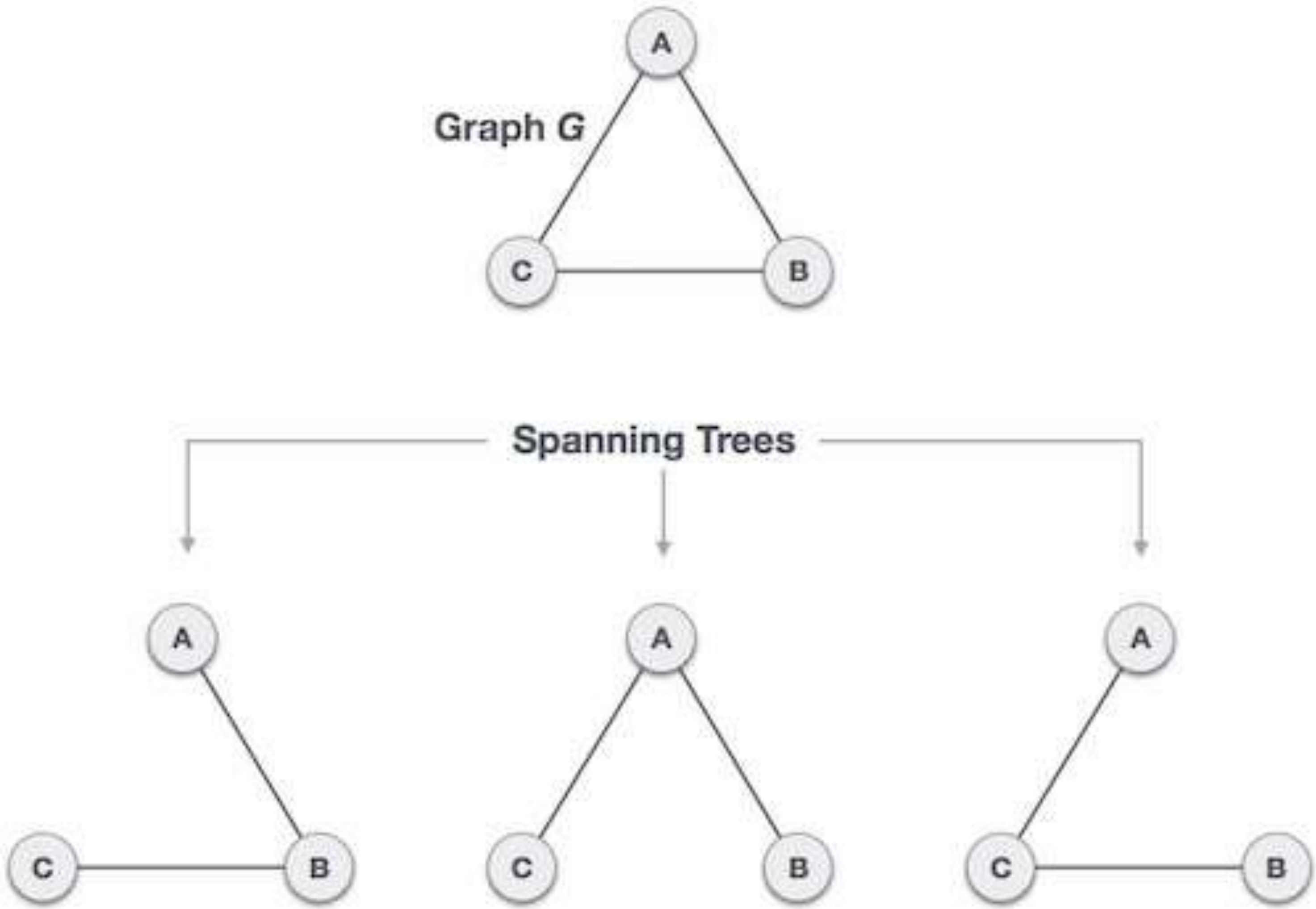
Minimum spanning tree

Kruskals’s and prim’s algorithm

Spanning tree

- A spanning tree of a graph is an undirected tree consisting of only those edges necessary to connect all the nodes in the original graph.
- A spanning tree has the properties that
  - For any pair of nodes there exists only one path between them
  - Insertion of any edge to a spanning tree forms a unique cycle.

Spanning tree - example



Minimum spanning tree

- The cost of a spanning tree of a weighted undirected graph is the sum of the costs (weights) of the edges in the spanning tree.
- A minimum cost spanning tree is a spanning tree of least cost.
- Two techniques for constructing minimum cost spanning tree
  - Kruskals’s algorithm
  - Prim’s algorithm

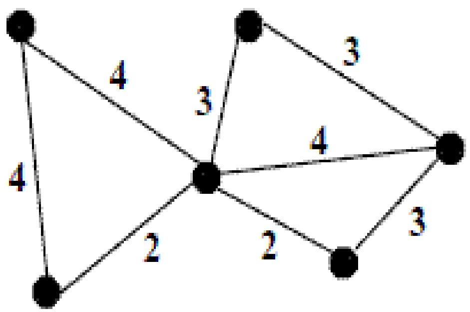
Kruskal’s algorithm

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

Kruskal's Algorithm

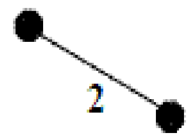
1

Given a network…………



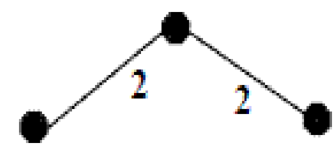
2

Choose the shortest edge (if there is more than one, choose any of the shortest)……..



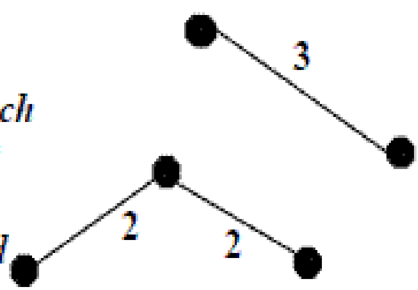
3

Choose the next shortest edge and add it……...



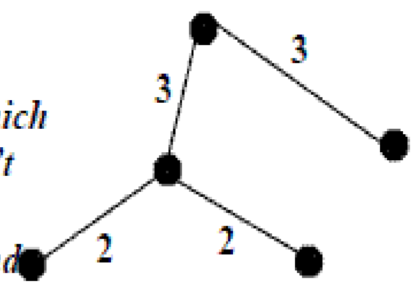
4

Choose the next shortest edge which wouldn't create a cycle and add it.



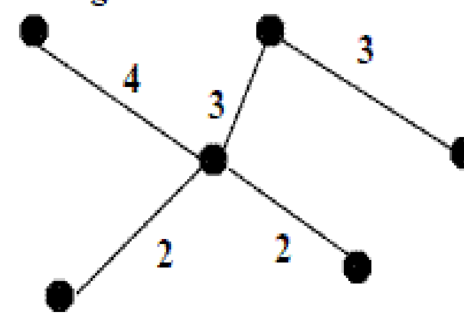
5

Choose the next shortest edge which wouldn't create a cycle and add it.



6

Repeat until you have a minimal spanning tree.



Algorithm

Function Kruskal(G = (N, A): graph; length: A → R+): set of edges

{initialization}

(i) Sort A by increasing length

(ii) n ← the number of nodes in N

(iii) T ← Ø {Solution Set that will contain the edges of the minimum spanning tree}

(iv) Initialize n sets, each containing a different element of set N

{greedy loop}

(v) repeat

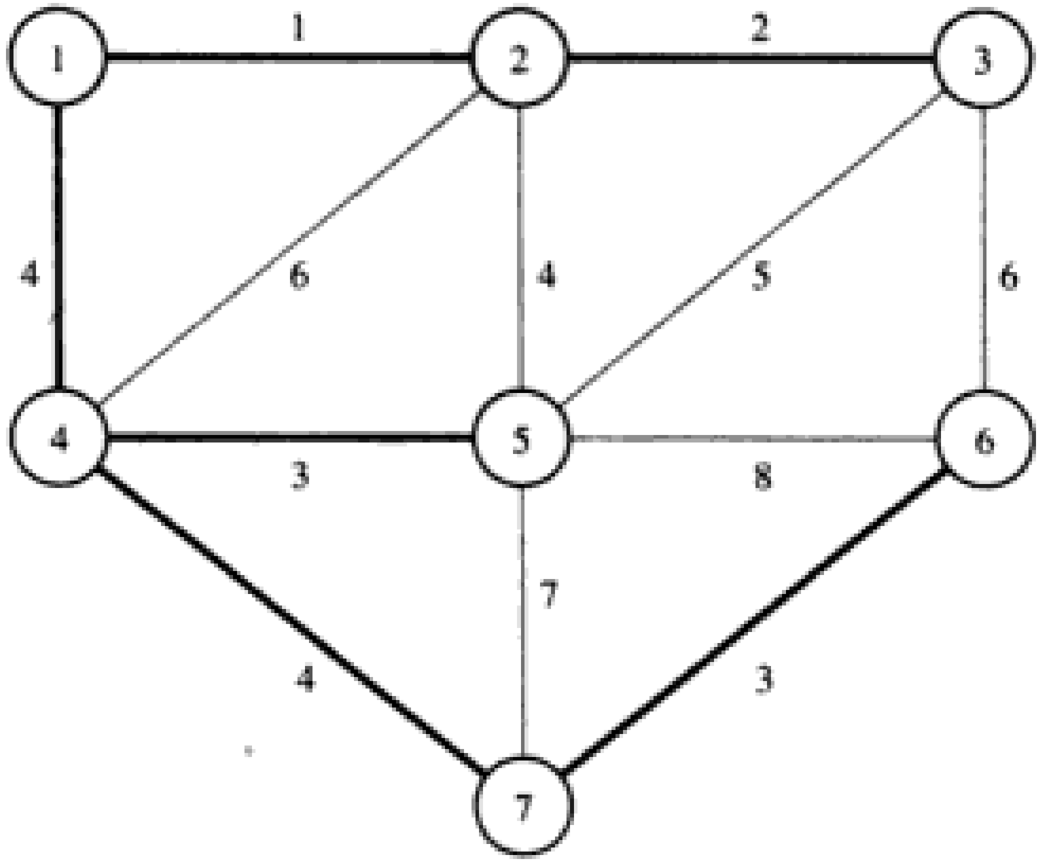
```
        e ← {u, v} ← such that e is the shortest edge not yet considered
        ucomp ← find(u)
        vcomp ← find(v)
        if ucomp ≠ vcomp then
                merge(ucomp, vcomp)
                T ← T U {e}
until T contains n - 1 edges
```

(vi) return T

procedure

- The set A of edges are sorted in increasing order of their length.
- The solution set T of edges is initially empty.
- As the algorithm progresses, edges are added to set T.
- We examine the edges of set A one by one and if an edge joins two nodes in different connected components, we add it to T.
- Consequently, the two connected components now form only one component. Otherwise the edge is rejected if it joins two nodes in the same connected component, and therefore cannot be added to T as it forms a cycle.
- The algorithm stops when n-1 edges for n nodes are added in the solution set T.
- At the end of the algorithm only one connected component remains, and T is then a minimum spanning tree for all the nodes of G.
- The complexity for the Kruskal's algorithm is in **Θ**(a log a) where a is total number of edges and n is the total number of nodes in the graph G.

- Find the minimum spanning tree for the following graph using Kruskal's Algorithm



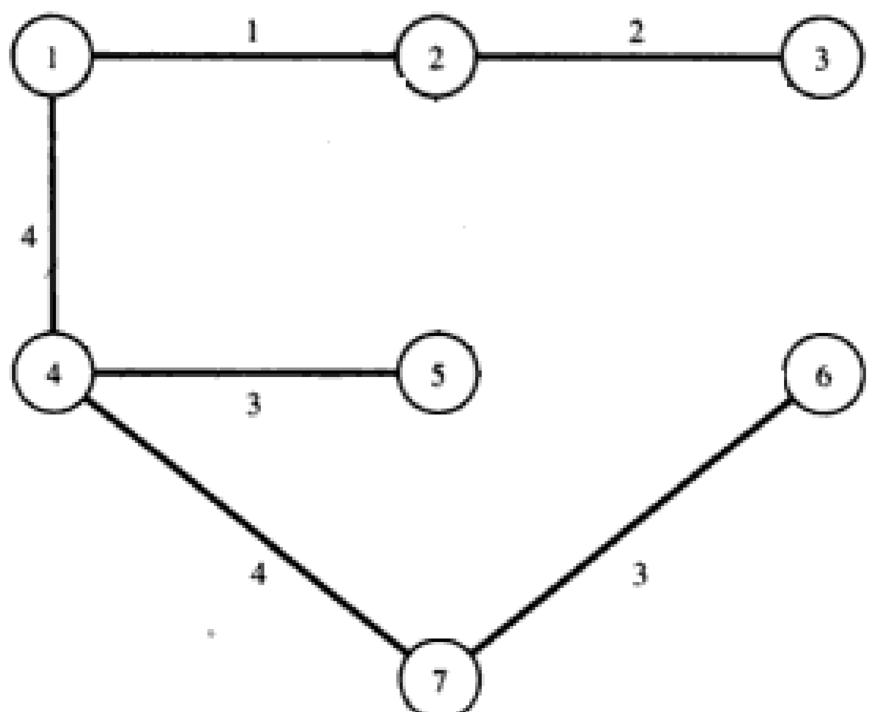
- **Solution:**
- **Step 1:** Sort A by increasing length

In increasing order of length the edges are: {1, 2}, {2, 3}, {4, 5}, {6, 7}, {1,4}, {2, 5}, {4,7}, {3, 5), {2, 4}, {3,6}, {5,7} and {5,6}.

- Step 2

Step	Edges considered - {u, v}	Connected Components
Initialization	-	{1} {2} {3} {4} {5} {6} {7}
1	{1,2}	{1,2}(3} {4} {5} {6} {7}
2	{2,3}	{1,2,3} {4} {5} {6} {7}
3	{4,5}	{1,2,3} {4,5} {6} {7}
4	{6,7}	{1,2,3} {4,5} {6,7}
5	{1,4}	{1,2,3,4,5} {6,7}
6	{2,5}	**Rejected**
7	{4,7}	{1,2,3,4,5,6,7}

- When the algorithm stops, solution set T contains the chosen edges {1, 2}, {2, 3}, {4, 5}, {6,7}, {1, 4} and {4,7}.
- This minimum spanning tree is shown below whose total length is **17**.



Prim’s algorithm

- In prim’s algorithm the minimum spanning tree grows in a natural way, starting from an arbitrary root.
- At each stage we add a new branch to the tree already constructed; the algorithm stops when all the nodes have been reached.

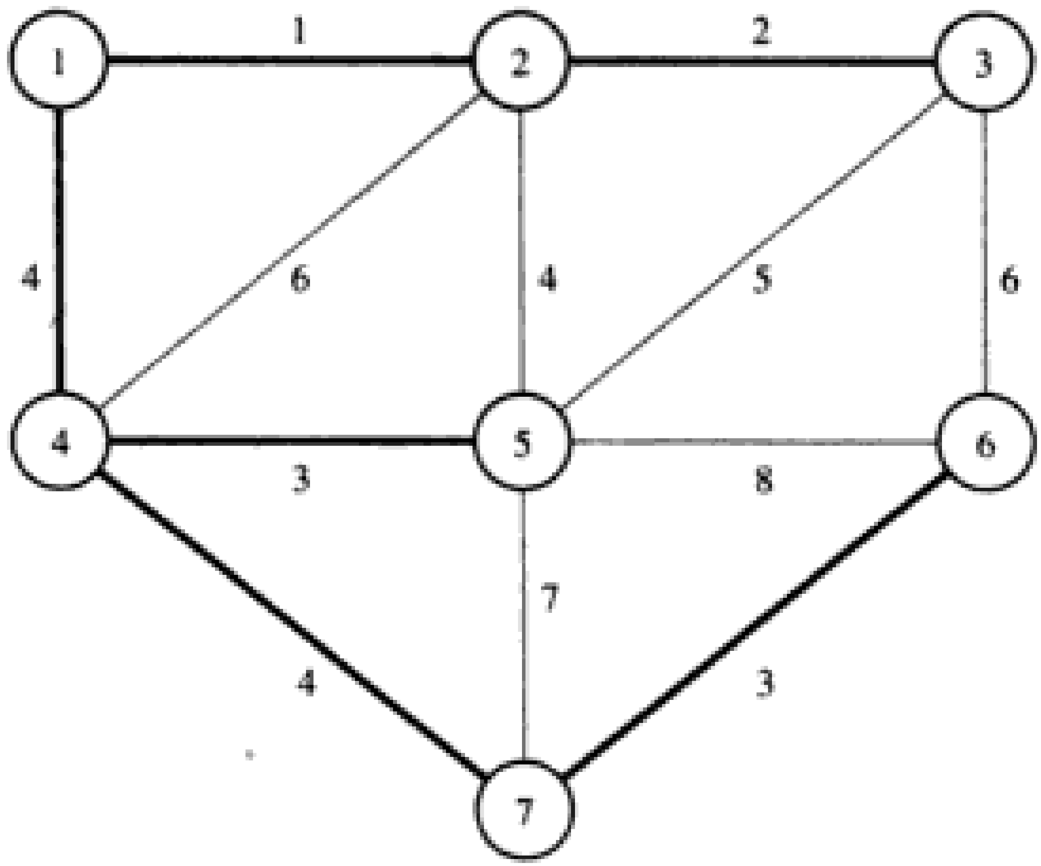
- Let B be a set of nodes, and A is a set of edges.
- Initially, B contains a single arbitrary node, and solution set T is empty.
- At each step Prim's algorithm looks for the shortest possible edge {u, v} such that u ε B and v ε N\B.
- It then adds v to set B and {u, v} to solution set T.
- In this way the edges in T form a minimum spanning tree for the nodes in B.
- We continue thus as long as B ≠ N.
- The complexity for the Prim’s algorithm is $\mathbf{\Theta(n^2)}$ where n is the total number of nodes in the graph G.

Algorithm

Function Prim(G = (N, A): graph; length: A — R+): set of edges

```
{initialization}
T ← Ø
B ← {an arbitrary member of N}
while B ≠ N do
        find e = {u, v} of minimum length such that
        u ε B and v ε N \ B
        T ← T U {e}
        B ← B U {v}
return T
```

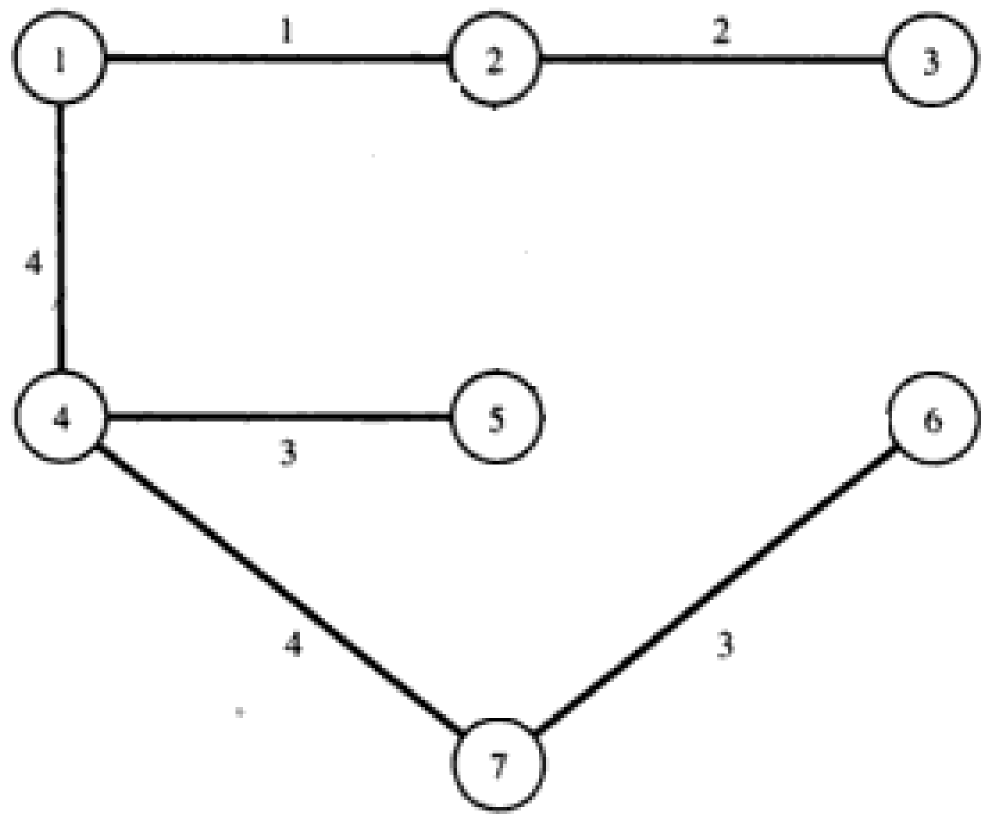
- Find the minimum spanning tree for the graph using Prim’s Algorithm



- **Step 1**
- We arbitrarily choose node 1 as the starting node.
- **Step 2**

Step	Edge Selected {u, v}	Set B	Edges Considered
Initialization	-	{1}	--
1	{1,2}	{1,2}	**{1,2}** {1,4}
2	{2,3}	{1,2,3}	{1,4} **{2,3}** {2,4} {2,5}
3	{1,4}	{1,2,3,4}	**{1,4}** {2,4} {2,5} {3,5} {3,6}
4	{4,5}	{1,2,3,4,5}	{2,4} {2,5} {3,5} {3,6} **{4,5}** {4,7}
5	{4,7}	{1,2,3,4,5,7}	{2,4} {2,5} {3,5} {3,6} **{4,7}** {5,6} {5,7}
6	{6,7}	{1,2,3,4,5,6,7}	{2,4} {2,5} {3,5} {3,6} {5,6} {5,7} **{6,7}**

- When the algorithm stops, T contains the chosen edges {1, 2}, {2,3}, {1,4}, {4,5}, {4,7} and {7,6}.
- This minimum spanning tree is shown below whose total length is **17**.



Running time of MST

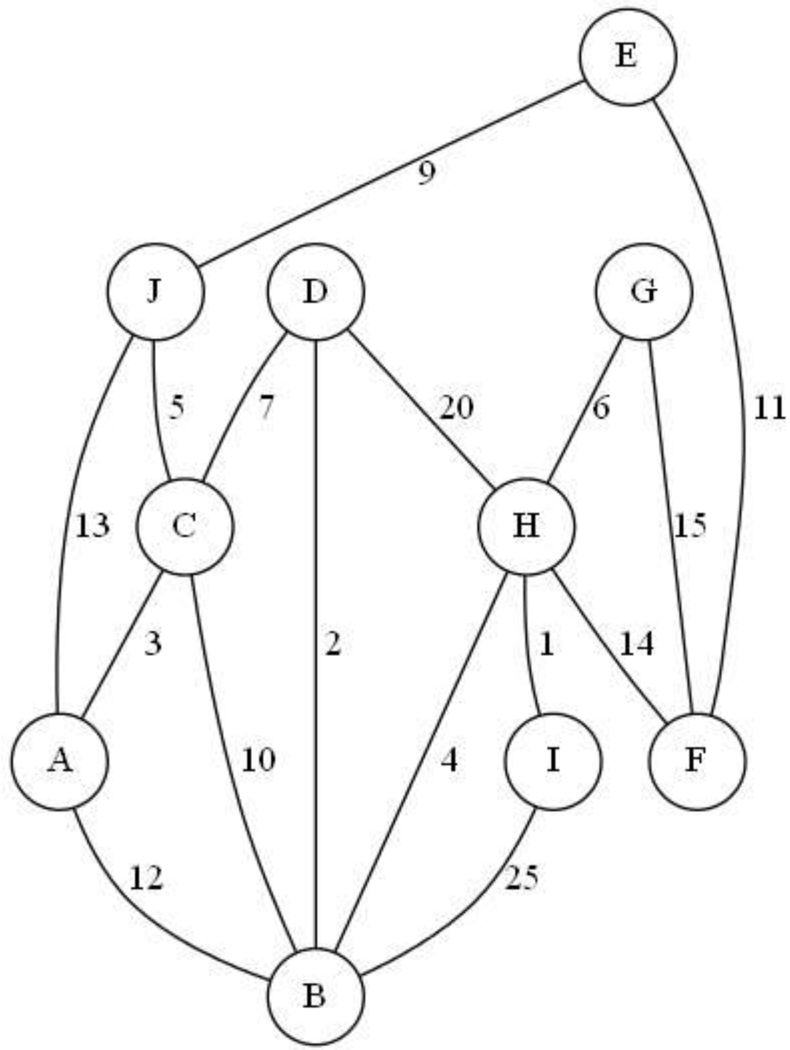
- When using binary heaps, the runtime of the Kruskal’s algorithm is O(E log V)
- When using binary heaps, the runtime of the Prim’s algorithm is O(E log V). If binary heaps are not used then the run time of prim’s is $O(V^2)$
- When using the Fibonacci heaps, the runtime of the Prim’s algorithm becomes: O(E + Vlog V)
- So, when an undirected graph is dense (i.e.,|V| is much small than |E|), then Prim’s algorithm is more efficient

Practice problems

Answer the following questions with either **true or false.**

1. Prim's and Kruskal's algorithms will always return the same Minimum Spanning tree (MST) – **false**
2. Prim's algorithm for computing the MST only work if the weights are positive - **false**
3. An MST for a connected graph has exactly V-1 edges, V being the number of vertices in the graph. - **true**
4. A graph where every edge weight is unique (there are no two edges with the same weight) has a unique MST. - **true**
5. The MST can be used to find the shortest path between two vertices. - **false**

- Find MST using prims’s and kruskal’s for the given grah



- Answer
  - Prim’s
  - The following edges are added to the MST in the given ordering: (A,C), (C,J), (C,D), (B,D), (B,H), (H,I), (H,G), (E,J), (E,F)
  - Kruskal’s
  - The following edges are added to the MST in the given ordering: (H,I), (B,D), (A,C), (B,H), (C,J), (G,H), (C,D), (E,J), (E,F)

Dijkstra's algorithm

- Dijkstra's Algorithm allows you to calculate the shortest path between one node (you pick which one) and *every other node in the graph*.

Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

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

Approach: Greedy

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

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

- The graph has the following:
  - vertices, or nodes
  - weighted edges that connect two nodes: (u,v) denotes an edge, and w(u,v) denotes its weight
- This is done by initializing three values:
  - dist, an array of distances from the source node s to each node in the graph, initialized the following way: dist(s) = 0; and for all other nodes v, dist(v) = INFINITY. This is done at the beginning because as the algorithm proceeds,
    the dist from the source to each node v in the graph will be recalculated and finalized when the shortest distance to v is found
  - Q, a queue of all nodes in the graph. At the end of the algorithm's progress, Q will be empty.
  - S, an empty set, to indicate which nodes the algorithm has visited. At the end of the algorithm's run, S will contain all the nodes of the graph

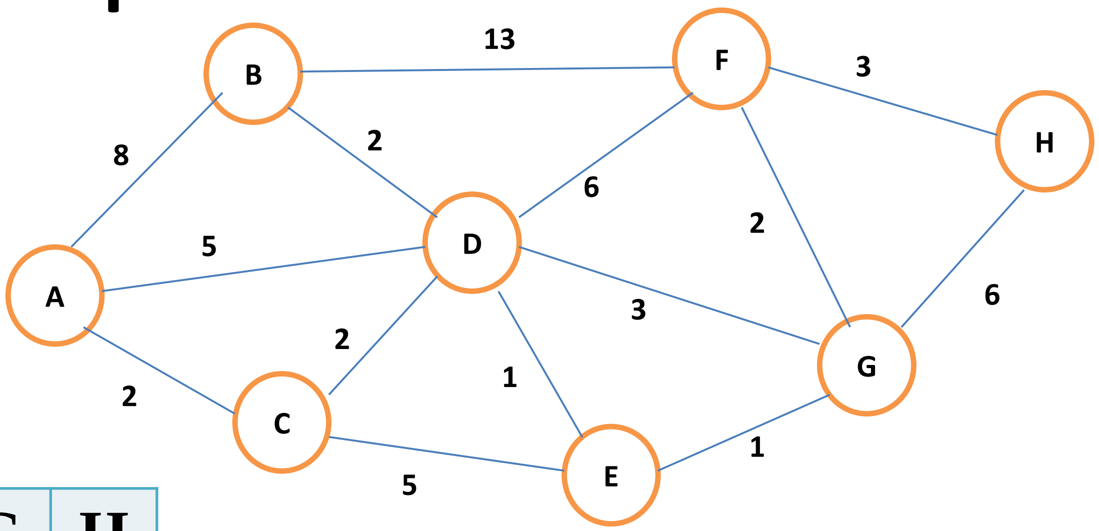
- The algorithm proceeds as follows:
  - While *Q* is not empty, pop the node *v*, that is not already in *S*, from *Q* with the smallest *dist (v)*. In the first run, source node s will be chosen because *dist(s)* was initialized to 0. In the next run, the next node with the smallest dist value is chosen.
  - Add node *v* to *S*, to indicate that *v* has been visited
  - Update *dist* values of adjacent nodes of the current node *v* as follows: for each new adjacent node *u*,
  - if *dist(v) + weight(u,v) < dist(u),* there is a new minimal distance found for *u*, so update *dist(u)* to the new minimal distance value;
  - otherwise, no updates are made to *dist(u)*.

  The algorithm has visited all nodes in the graph and found the smallest distance to each node. *dist* now contains the shortest path tree from source *s*.

Dijkstra pseudocode

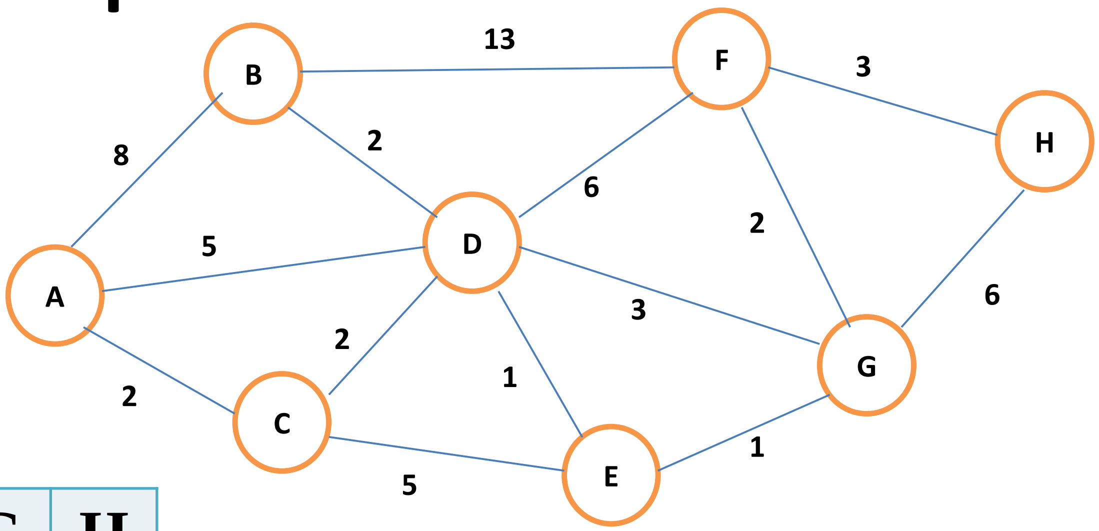
```
function Dijkstra(Graph, source)
        dist[source] := 0      // Distance from source to source is set to 0
        for each vertex v in Graph: // Initializations
        if v ≠ source
                dist[v] := infinity // Unknown distance function from source to
                                each node set to infinity
        add v to Q // All nodes initially in Q
        while Q is not empty: // The main loop
                v := vertex in Q with min dist[v] // In the first run-through, this
                                                vertex is the source node
                remove v from Q
        for each neighbor u of v: // where neighbor u has not yet been removed
                                from Q.
                alt := dist[v] + length(v, u)
                if alt < dist[u]: // A shorter path to u has been found
                        dist[u] := alt // Update distance of u
return dist[]
```

Example



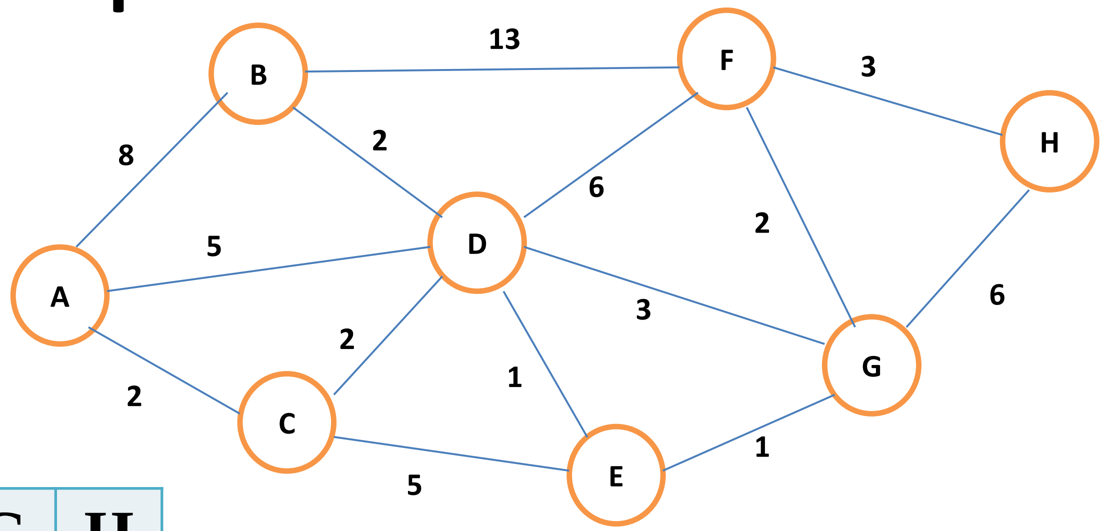
V	A	B	C	D	E	F	G	H
A	0_A	8_A	2_A	5_A	∞	∞	∞	∞

Example



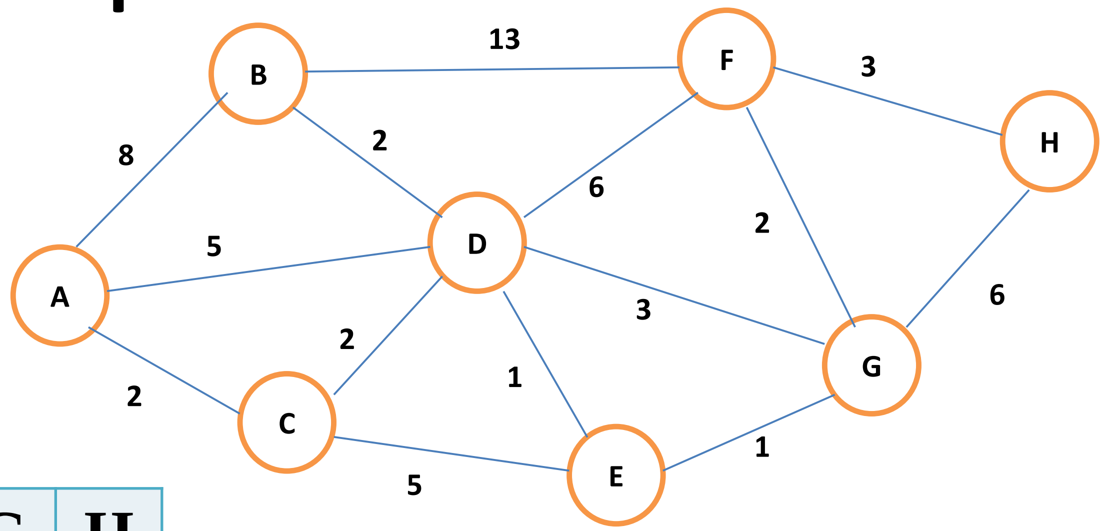
V	A	B	C	D	E	F	G	H
A	0_A	8_A	2_A	5_A	∞	∞	∞	∞
C		8_A	2_A	4_C	7_C	∞	∞	∞

Example



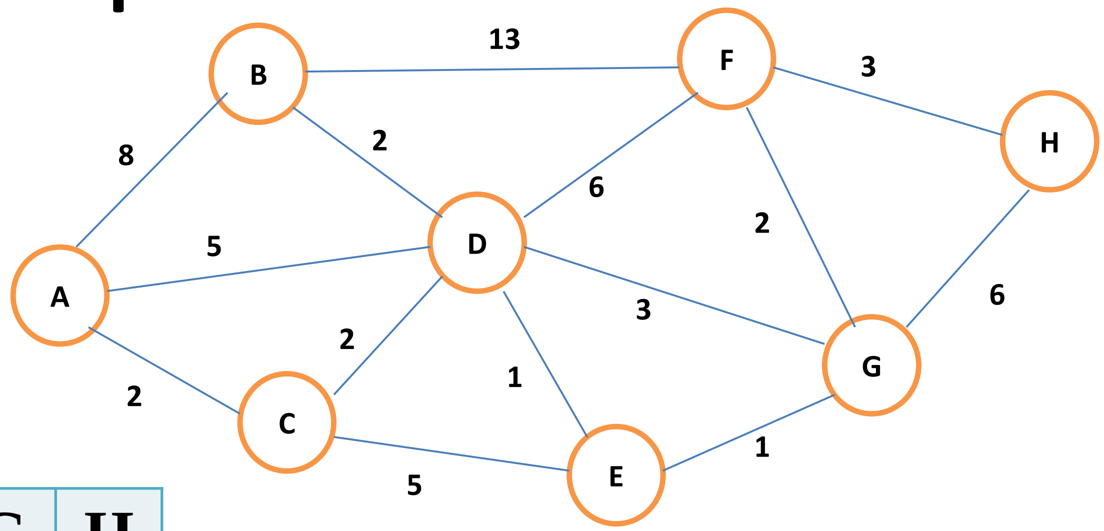
V	A	B	C	D	E	F	G	H
A	0_A	8_A	2_A	5_A	∞	∞	∞	∞
C		8_A	2_A	4_C	7_C	∞	∞	∞
D		6_D		4_C	5_D	10_D	7_D	∞

Example



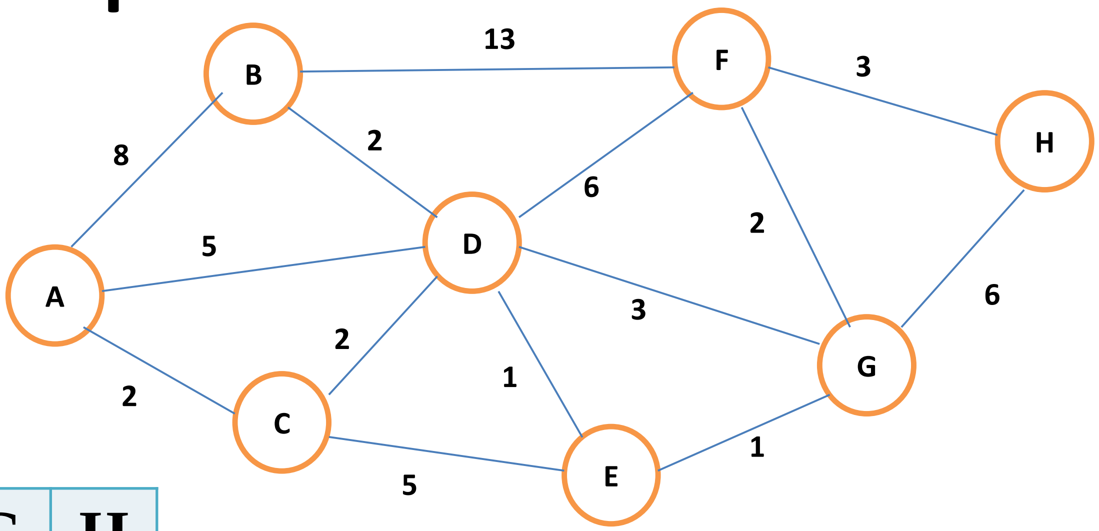
V	A	B	C	D	E	F	G	H
A	0_A	8_A	2_A	5_A	∞	∞	∞	∞
C		8_A	2_A	4_C	$\mathbf{7_C}$	∞	∞	∞
D		$\mathbf{6_D}$		4_C	$\mathbf{5_D}$	$\mathbf{10_D}$	$\mathbf{7_D}$	∞

Example



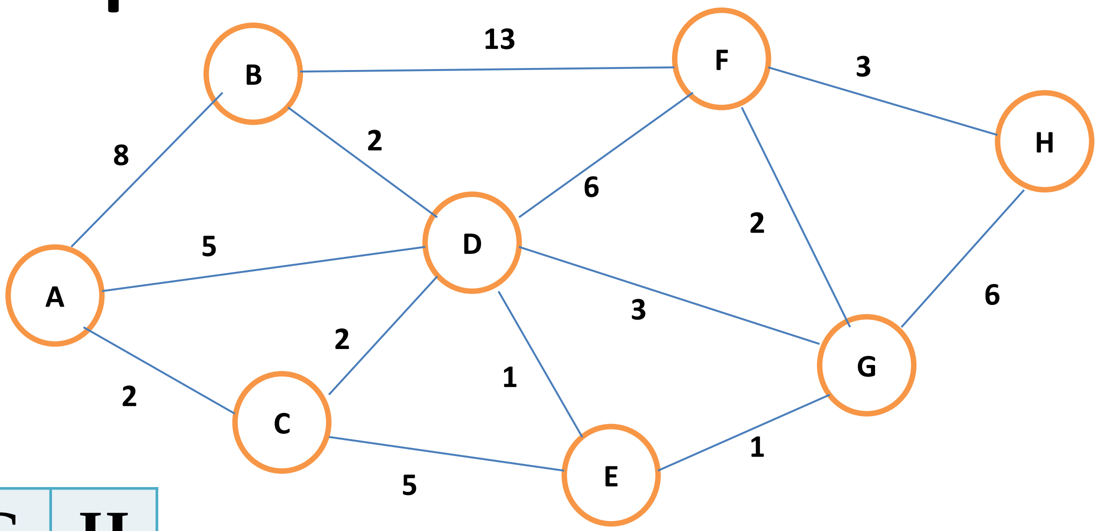
V	A	B	C	D	E	F	G	H
A	0_A	8_A	2_A	5_A	∞	∞	∞	∞
C		8_A	2_A	4_C	7_C	∞	∞	∞
D		6_D		4_C	5_D	10_D	7_D	∞
E		6_D			5_D	10_D	6_E	∞

Example



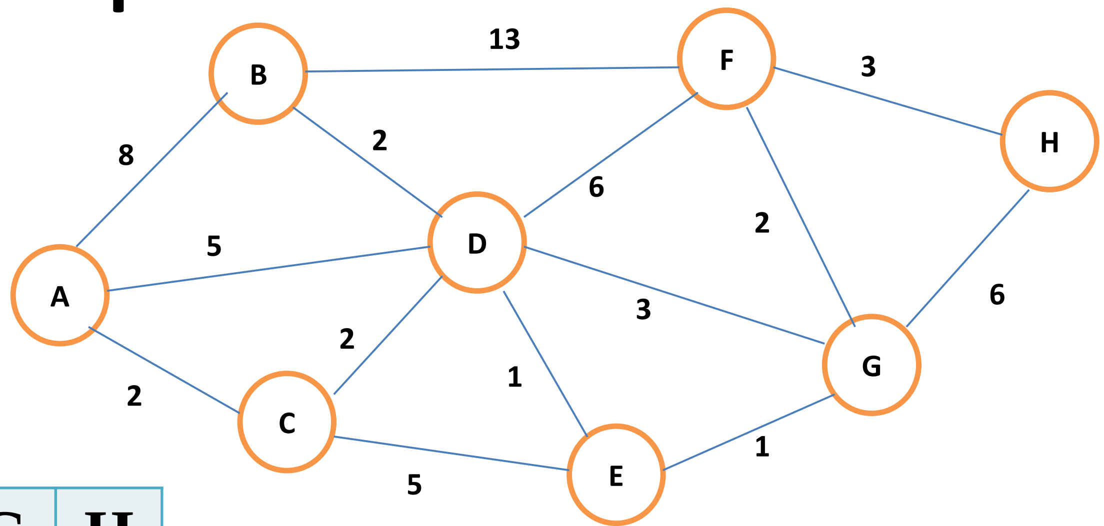
V	A	B	C	D	E	F	G	H
A	0_A	8_A	2_A	5_A	∞	∞	∞	∞
C		8_A	2_A	4_C	7_C	∞	∞	∞
D		6_D		4_C	5_D	10_D	7_D	∞
E		6_D			5_D	10_D	6_E	∞
B		6_D				10_D	6_E	∞

Example



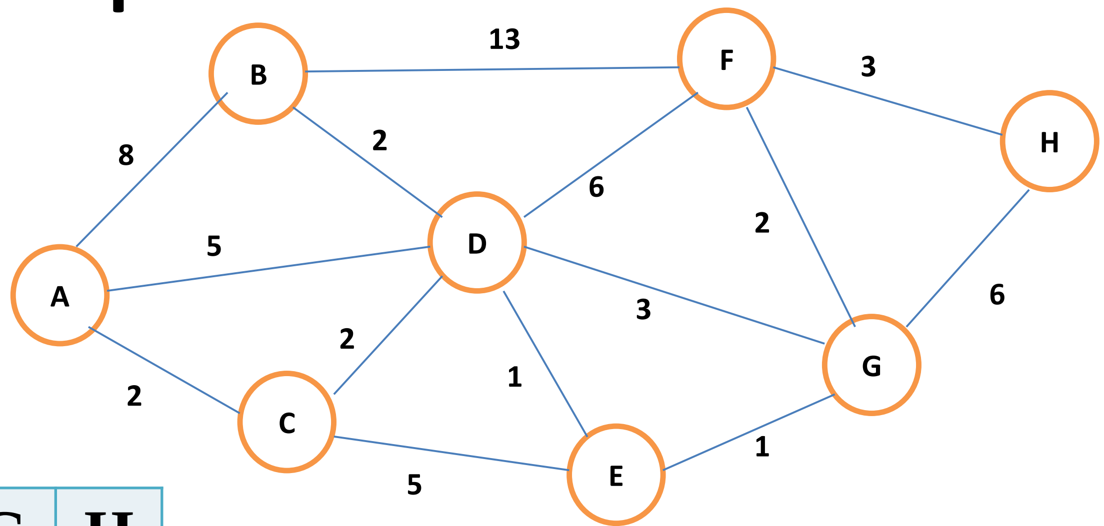
V	A	B	C	D	E	F	G	H
A	0_A	8_A	2_A	5_A	∞	∞	∞	∞
C		8_A	2_A	4_C	7_C	∞	∞	∞
D		6_D		4_C	5_D	10_D	7_D	∞
E		6_D			5_D	10_D	6_E	∞
B		6_D				10_D	6_E	∞
G						8_G	6_E	12_G

Example



V	A	B	C	D	E	F	G	H
A	0_A	8_A	2_A	5_A	∞	∞	∞	∞
C		8_A	2_A	4_C	7_C	∞	∞	∞
D		6_D		4_C	5_D	10_D	7_D	∞
E		6_D			5_D	10_D	6_E	∞
B		6_D				10_D	6_E	∞
G						8_G	6_E	12_G
F						8_G		11_F

Example



V	A	B	C	D	E	F	G	H
A	0_A	8_A	2_A	5_A	∞	∞	∞	∞
C		8_A	2_A	4_C	7_C	∞	∞	∞
D		6_D		4_C	5_D	10_D	7_D	∞
E		6_D			5_D	10_D	6_E	∞
B		6_D				10_D	6_E	∞
G						8_G	6_E	12_G
F						8_G		11_F
H								11_F

From this table suppose you
Want to find the shortest path
From A to G
Then start from destination end
ACDEG

Implementations and Running Times

The simplest implementation is to store vertices in an array or linked list. This will produce a running time of

$O(|V|^2 + |E|)$

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of

$O((|E|+|V|) \log |V|)$

Dijkstra's Algorithm - Why It Works

- As with all greedy algorithms, we need to make sure that it is a correct algorithm (e.g., it *always* returns the right solution if it is given correct input).

- A formal proof would take longer than this presentation, but we can understand how the argument works intuitively.

- If you can’t sleep unless you see a proof, see the second reference or ask us where you can find it.

Dijkstra's Algorithm - Why use it?

- As mentioned, Dijkstra’s algorithm calculates the shortest path to every vertex.
- However, it is about as computationally expensive to calculate the shortest path from vertex *u* to every vertex using Dijkstra’s as it is to calculate the shortest path to some particular vertex *v*.
- Therefore, anytime we want to know the optimal path to some other vertex from a determined origin, we can use Dijkstra’s algorithm.

Time Complexity: Using List

The simplest implementation of the Dijkstra's algorithm stores vertices in an ordinary linked list or array

- Good for dense graphs (many edges)

- $|V|$ vertices and $|E|$ edges
- Initialization $O(|V|)$
- While loop $O(|V|)$
  - Find and remove min distance vertices $O(|V|)$
- Potentially $|E|$ updates
    - Update costs $O(1)$

Total time $O(|V^2| + |E|) = O(|V^2|)$

Time Complexity: Priority Queue

For sparse graphs, (i.e. graphs with much less than $|V^2|$ edges) Dijkstra's implemented more efficiently by *priority queue*

- Initialization $O(|V|)$ using $O(|V|)$ buildHeap
- While loop $O(|V|)$
  - Find and remove min distance vertices $O(\log |V|)$ using $O(\log |V|)$ deleteMin

- Potentially $|E|$ updates
  - Update costs $O(\log |V|)$ using decreaseKey

Total time $O(|V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$
- $|V| = O(|E|)$ assuming a connected graph

- **Applications of Dijkstra's algorithm:**
  - It is used in finding Shortest Path.
  - It is used in geographical Maps.
  - To find locations of Map which refers to vertices of graph.
  - Distance between the location refers to edges.
  - It is used in IP routing to find Open shortest Path First.
  - It is used in the telephone network.