Data Structure & Algorithms

Lec 14: Graph Algorithms

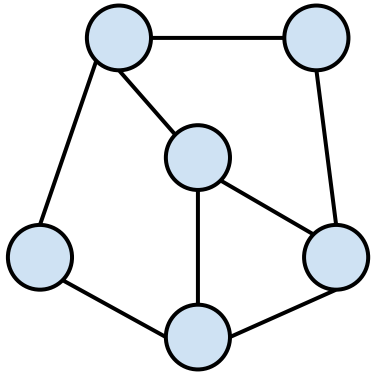
Fall 2017 - University of Windsor
Dr. Sherif Saad

Outlines

1. Introduction to Graph
2. Breadth First Search Algorithm
3. Depth First Search Algorithm
4. Dijkstra's Shortest Path Algorithm

What is a graph?

- It is a nonlinear data structure that consists of a set of nodes (vertices) and a set of links (edges) that relate the nodes to each other.
- Graphs are mathematical structures that represent pairwise relationships between objects. Where objects are represented by a set of vertices and the links are represented by a set of edges.



G = (V,E)

What is a graph?

Formal Definition

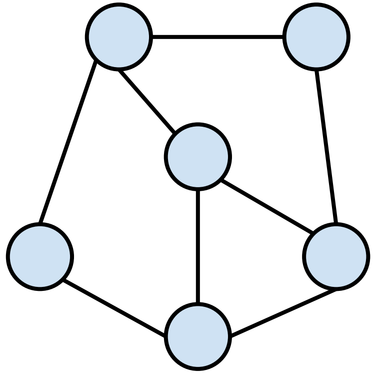
A graph G is defined as follows:

G = (V,E)

Where:

V(G): is a finite, nonempty set of vertices

E(G): is a finite, nonempty set of edges.



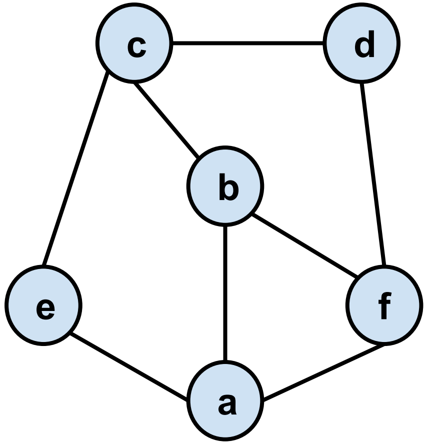
G = (V,E)

Types of Graphs

Undirected Graph: All the edges are bidirectional, this means the edges have no direction and the edges are unordered, $(v_0,v_1) = (v_1,v_0)$

V(G1) = {a, b, c, d, e, f}

E(G1) = {(a,b), (a,e), (a,f), (b,f), (b,c), (c,d), (c, e), (d,f) }



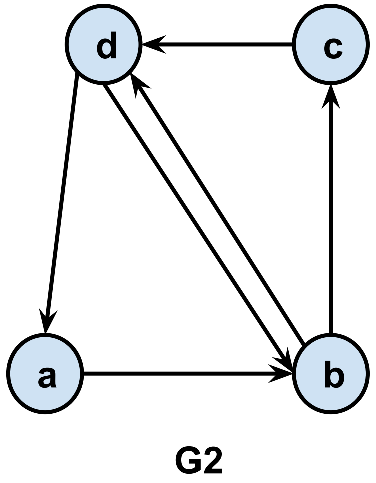
G1

Types of Graphs

Directed Graph: All the edges in the graph have direction, this means each edge is a directed pair of vertices $(v_0, v_1) \neq (v_1, v_0)$

V(G2) = {a, b, c, d}

E(G2) = {(a,b), (b,c), (b,d), (c,d), (d,a), (d,b) }

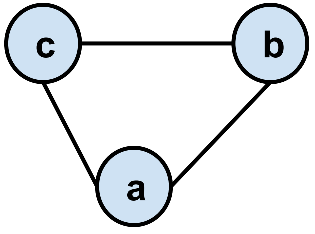


Types of Graphs

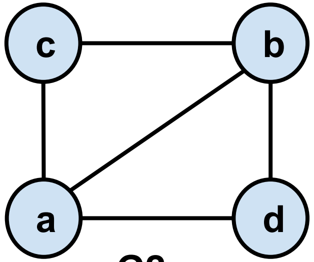
Complete Graph is a graph that has the maximum number of edges. This means a direct edge connects every vertex to every other vertex in the graph. G1 is a complete graph and G2 is not a complete (incomplete) graph.

A complete undirected graph with n vertices has n(n-1)/2 edges.

An complete directed graph with n vertices has n(n-1) edges.



G1

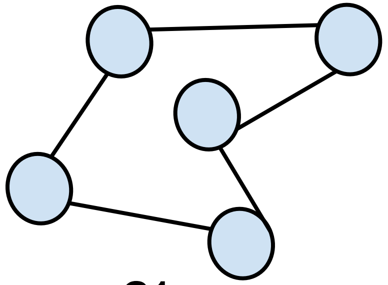


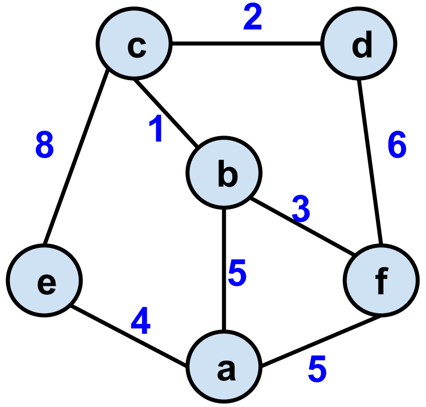
G2

Types of Graphs

Labeled Graph is a graph G with a set V of vertices and set E of edges where each vertex has a unique label so that all vertices are considered distinct for purposes of enumeration

Weighted Graph each edge is assigned a numerical weight or cost.

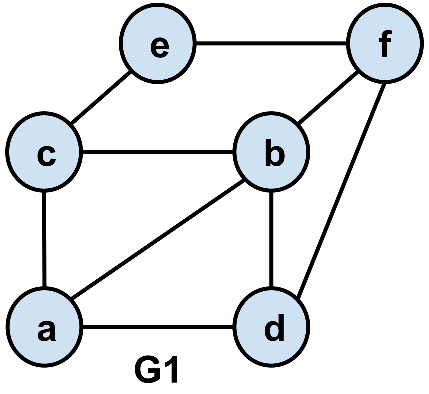

G1

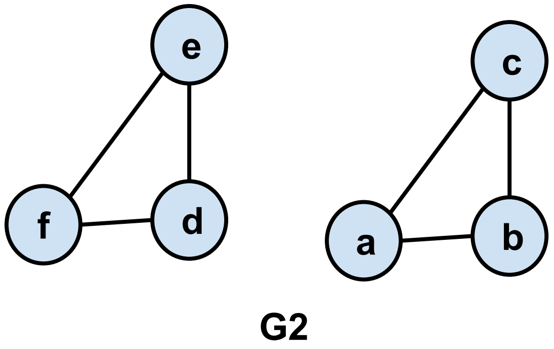


G2

Types of Graphs

An graph is connected when there is a path between every pair of vertices. In a connected graph, there are no unreachable vertices. Starting from any vertex v ∈ V we can reach every other vertex u ∈ V.





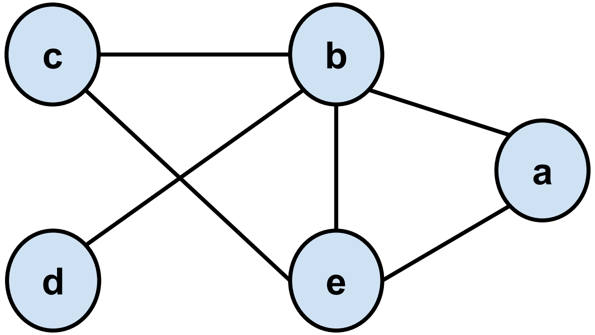
Graph Vertex Degree

The degree of a vertex in a graph is the number of edges incident to the vertex or the number of edges that enter or exit from the vertex.

$\deg(G_2) = 2|E|$

Max $\deg(G_2)$= 4 and Min $\deg(G_2)$ = 1

deg(a) = 2, deg(b) = 4, deg(c) = 2, deg(d) = 1, deg(e)= 3,



G2

In direct graph for each vertex we have in-degree (number of edges enter the vertex)and out-degree (number of edges exist the vertex.

Path and Simple Path

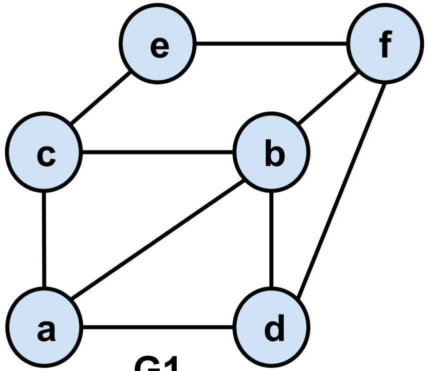
A simple path is a path in which all vertices, except possibly the first and the last, are distinct

A cycle is a simple path in which the first and the last vertices are the same.

P_1 = {a, d, f, e} is a simple path in G_1

P_2= {a,d,b,f,a} is a cycle.

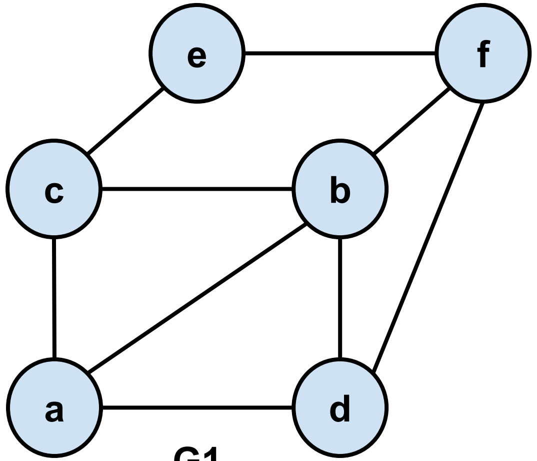
P_3= {f, d, b, a ,c, b, f} is a path in G_1 but not a simple path



G1

Graph Representation

There are many methods to represent a graph structure; the most common methods are adjacency matrix and adjacency list.



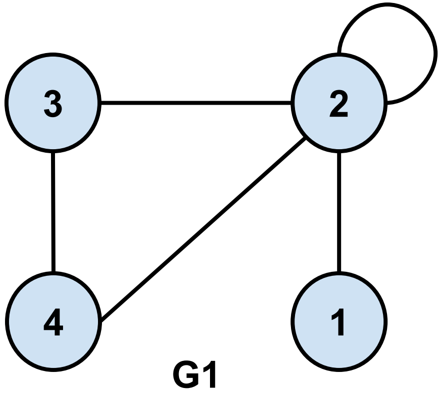
G1

Graph Representation

Adjacency Matrix: The adjacency matrix of graph G is a two-dimensional

n by n array (adj_mat[n][n]), where n is the number of nodes in G.

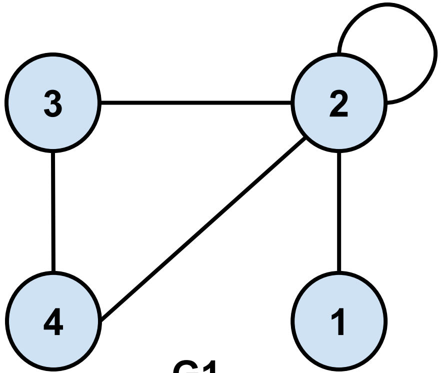
if there is an edge e between vi and vj in G then adj_mat[i][j]= 1 and if there is no such edge then adj_mat[i][j]= 0



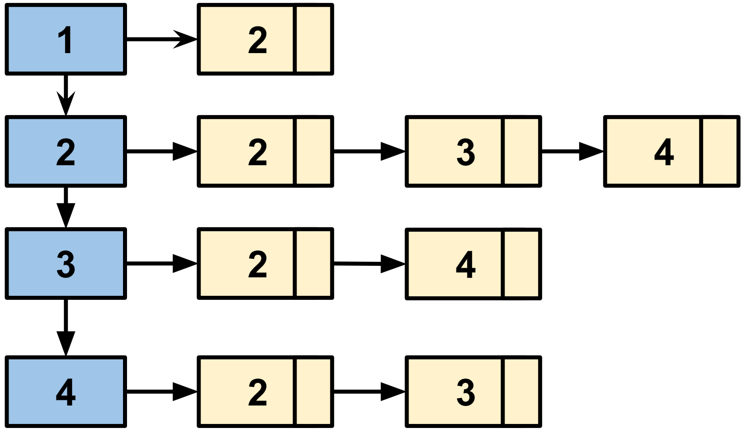
0	1	0	0
1	1	1	1
0	1	0	1
0	1	1	0

Graph Representation

Adjacency Lists: An adjacency list is a linked list L of linked lists. Each node in i L is a pointer to a linked list, which contains all the vertices that are adjacent to vertex i



G1



Traversing a Graph

Traversing a graph means visiting every vertex and edge in the graph exactly once in a well defined order.

It is important to keep track of visited vertices to avoid visiting the same vertex more than one.

There are many ways to traverse graphs the most common methods are:

- Breadth First Search (BFS)
- Depth First Search (DFS)

Breadth First Search

A a general algorithm for graph traversal (searching tree or graph data structure)

Works on directed and undirected graphs

Implemented using a queue data structure.

Time Complexity:

- $O(|V|+|E|)$ traversed without repetition
- $O(b^d)$ in implicit graph (where b is the branching factor and d is the depth)

Breadth First Search

How it works?

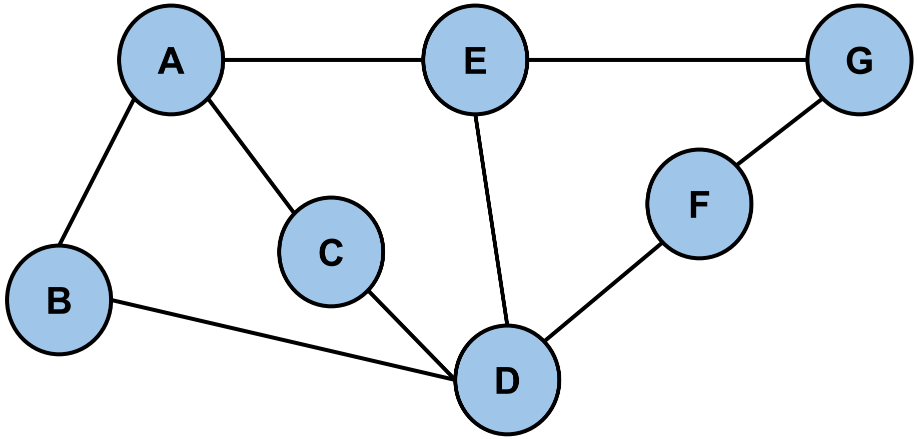
- Traverse the graph one level (layer) at a time. We need to keep track of visited (explored) nodes, so we do not visited a node infinite times. (graph could have a cycle)
- Starting from a source vertex, we traverse the graph breadthwise. Where all the vertices on the same level are visited first, then we move to the next level in the graph and so on.
- Vertices in the same level have the same distance (number of edges) to the source vertex.

Breadth First Search

There are many applications for BFS:

- Search Engines uses BFS to build search index.
- Shortest Path and Minimum Spanning Tree for unweighted graph
- Implementation of garbage collection algorithms
- Detect cycle in undirected graph.
- To find all neighbors nodes in P2P networks like BitTorrent
- Finding all nodes within a given distance from a source node.

Breadth First Search

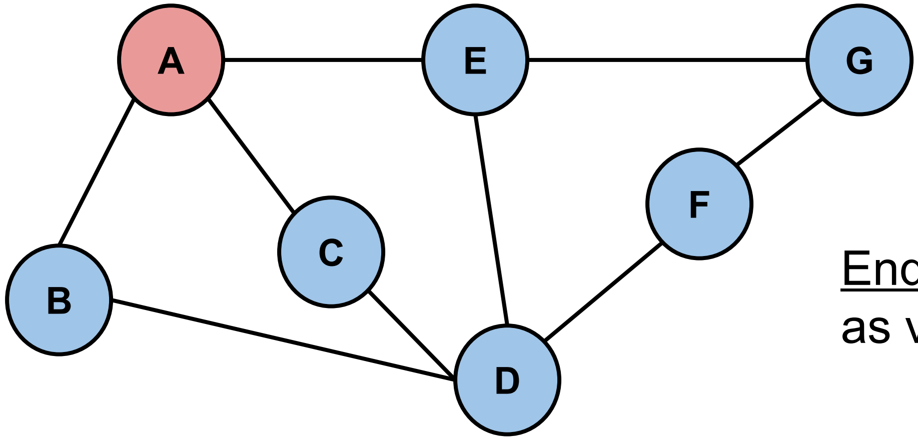


Traverse the graph starting from **A**

Queue(FIFO): { }

Visited Node:

Breadth First Search

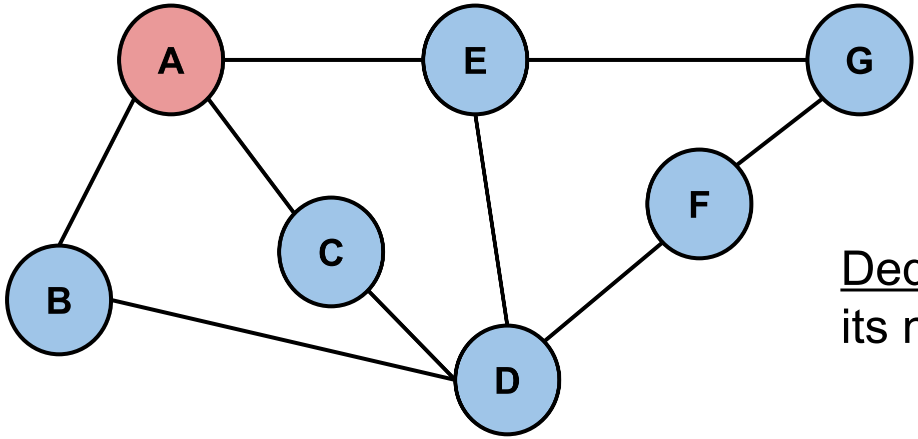


Enqueue A and marked as visited node

Queue(FIFO): {A}

Visited Node: A

Breadth First Search

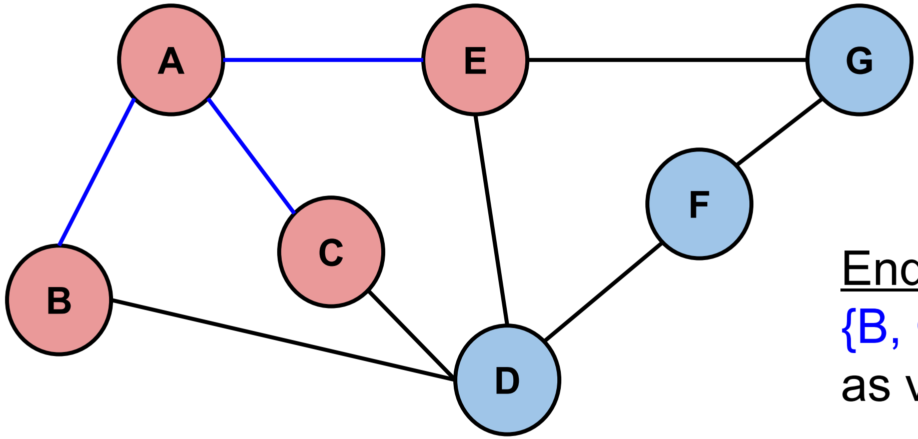


Dequeue A and discover its neighbors

Queue(FIFO): { A }

Visited Node: A

Breadth First Search

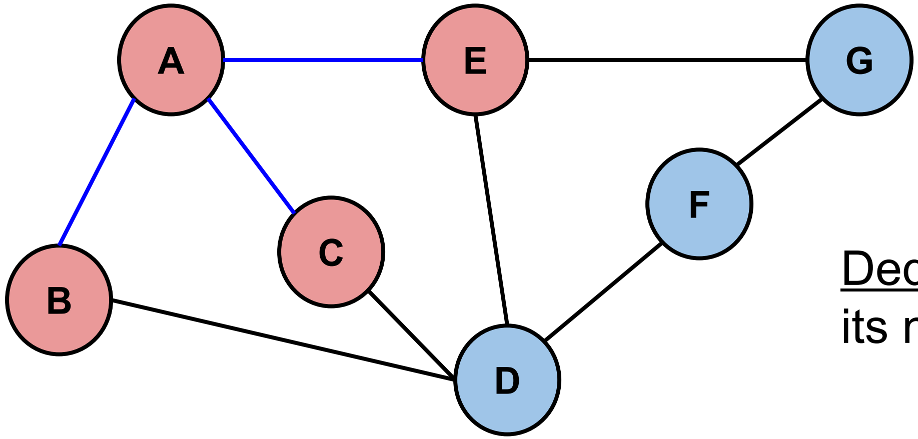


Enqueue A's neighbors {B, C, E} and mark them as visited nodes

Queue(FIFO): {B →C → E}

Visited Node: A, B, C, E

Breadth First Search

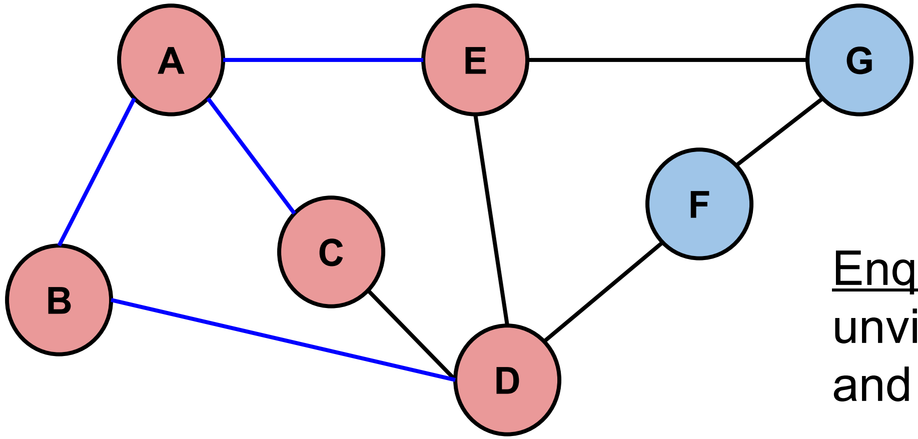


Dequeue B and discover its neighbors

Queue(FIFO): {B→ C → E}

Visited Node: A, B, C, E

Breadth First Search

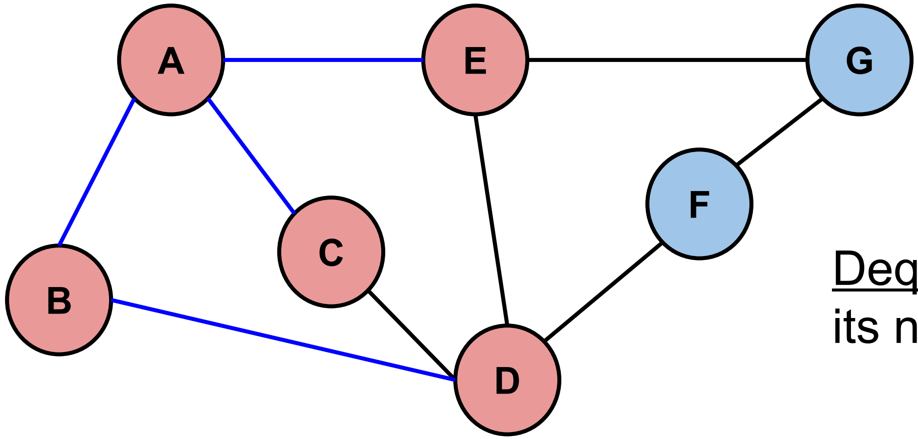


Enqueue D as its the only unvisited neighbor of B and mark D as visited

Queue(FIFO): {C → E →D}

Visited Node: A, B, C, E, D

Breadth First Search

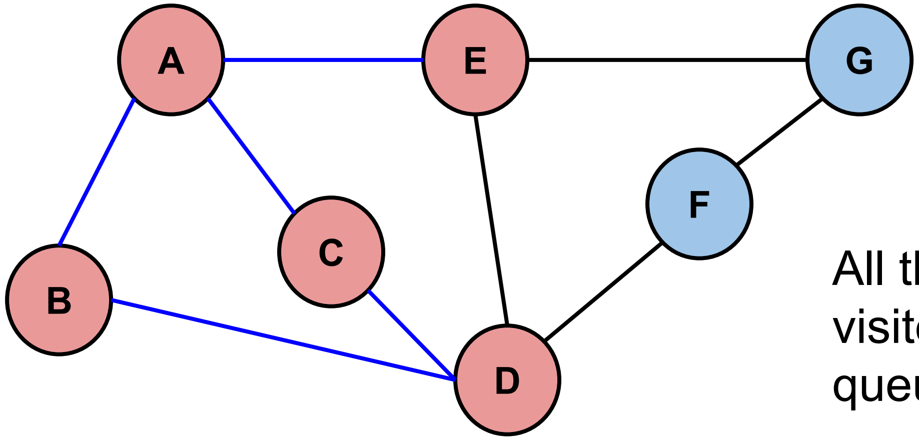


<u>Dequeue</u> C and discover its neighbors

Queue(FIFO): {C→ E →D}

Visited Node: A, B, C, E, D

Breadth First Search

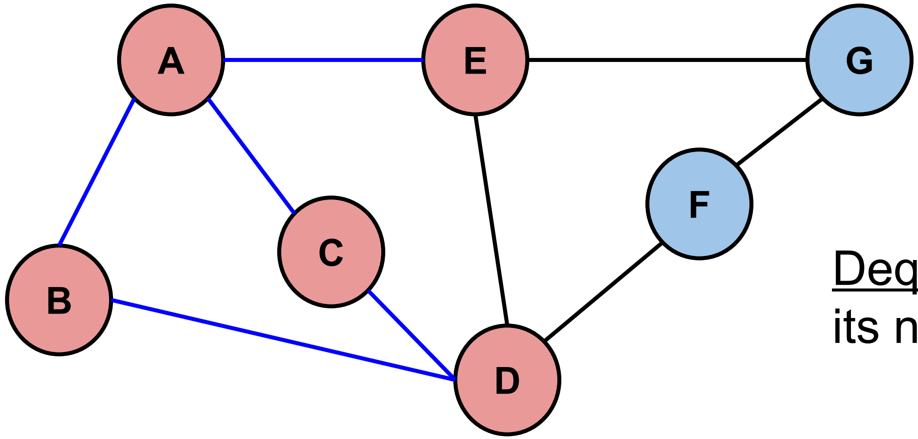


All the neighbors of C are visited nodes, we will not queue any new node

Queue(FIFO): {E →D}

Visited Node: A, B, C, E, D

Breadth First Search

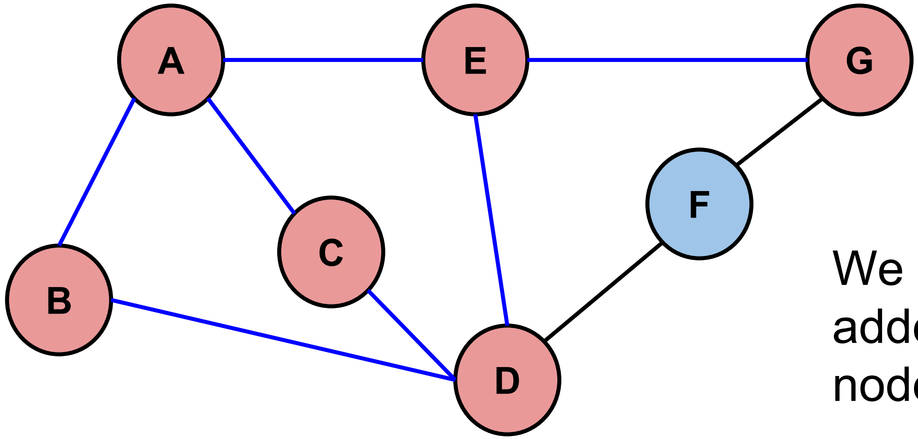


Dequeue E and discover its neighbors

Queue(FIFO): {E→ D}

Visited Node: A, B, C, E, D

Breadth First Search

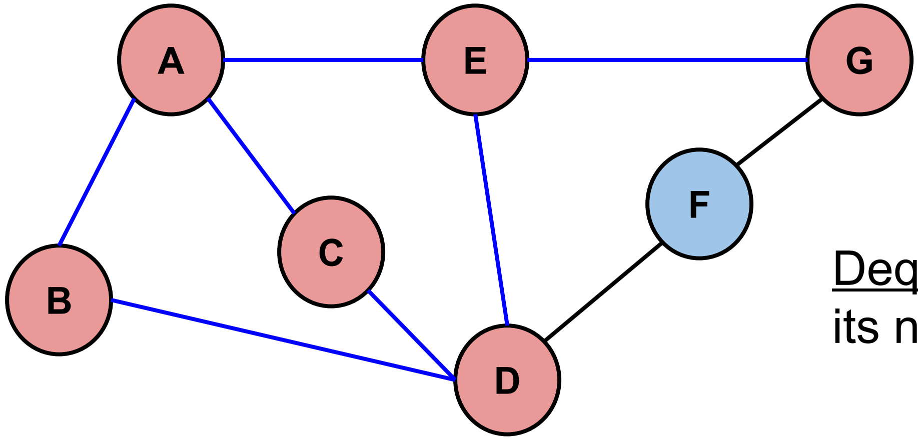


We only <u>enqueue</u> G, and added to the visited nodes

Queue(FIFO): {D → G}

Visited Node: A, B, C, E, D, G

Breadth First Search

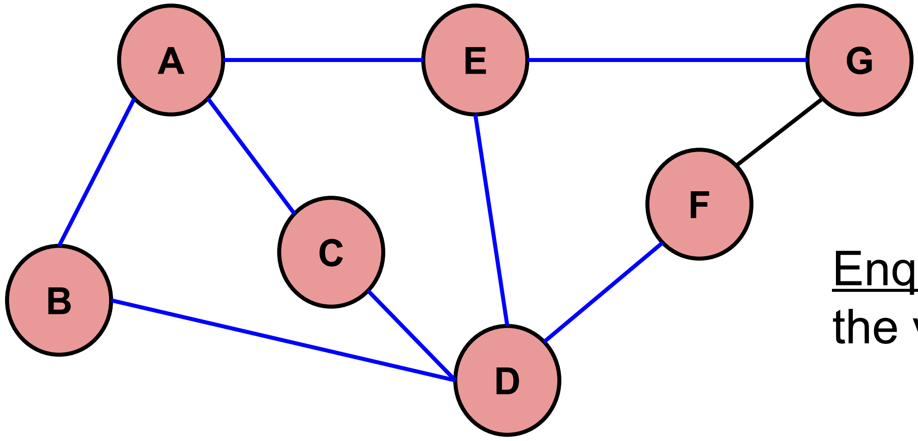


Dequeue D and discover its neighbors

Queue(FIFO): { D→ G}

Visited Node: A, B, C, E, D, G

Breadth First Search

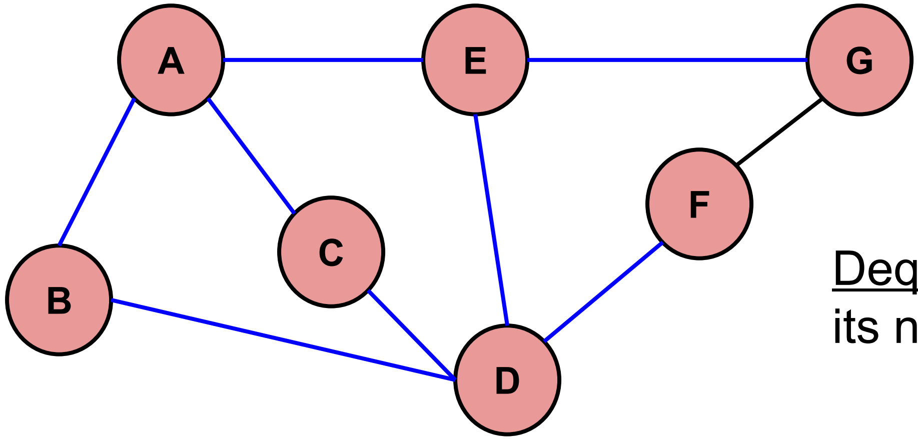


Enqueue F and added to the visited nodes

Queue(FIFO): {G → F}

Visited Node: A, B, C, E, D, G, F

Breadth First Search

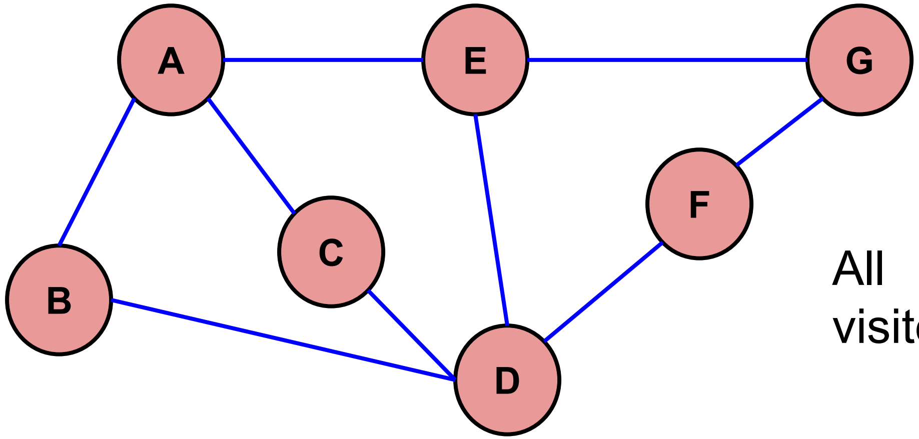


<u>Dequeue</u> G and discover its neighbors

Queue(FIFO): {G → F}

Visited Node: A, B, C, E, D, G, F

Breadth First Search

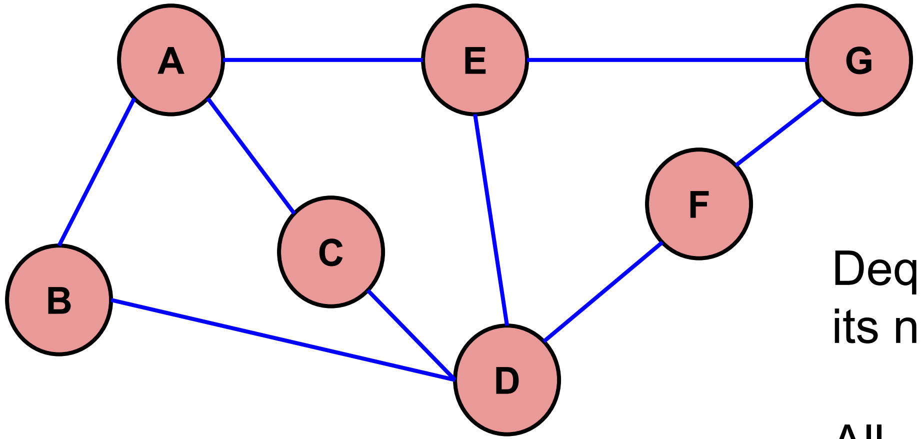


All G's neighbors are visited nodes

Queue(FIFO): {F}

Visited Node: A, B, C, E, D, G, F

Breadth First Search



Queue(FIFO): {}

Visited Node: A, B, C, E, D, G, F

Dequeue F and discover its neighbors.

All F's neighbors are visited nodes

The Queue is empty, we are done

Breadth First Search: Pseudocode

```
def BFS(G,v):
    # Let Q be an empty queue
    Q = Que()

    for u in V of G:
        visited[u]= False

    Q.enque(v)

    while Q.isEmpty() == False:
        v = Q.deque()
        if v not in visited:
            visited[v] = True

            for w in neighbours of v and visited[w] is False:
                Q.enque(w)
```

Depth First Search

A general algorithm for graph traversal (searching tree or graph data structure)

Works on directed and undirected graphs

Implemented using a stack data structure.

Time Complexity:

- O(|V|+|E|) traversed without repetition
- O(b^d) in implicit graph (where b is the branching factor and d is the depth)

Depth First Search

How it works?

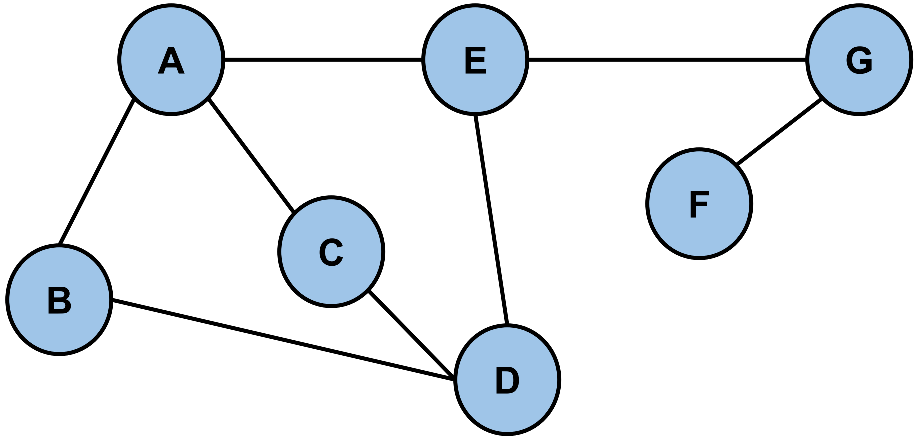
- Go forward (in depth) as long as it is possible, if not then backtrack
- Backtrack means you reached a dead-end (e.g. leaf node in a search tree)
- We need to keep track of visited (explored) nodes, so we do not visited a node infinite times.

Depth First Search

There are many applications for DFS:

- Detecting cycle in graph
- Path finding
- Solving puzzles with only one solutions.
- Tasks scheduling.

Depth First Search

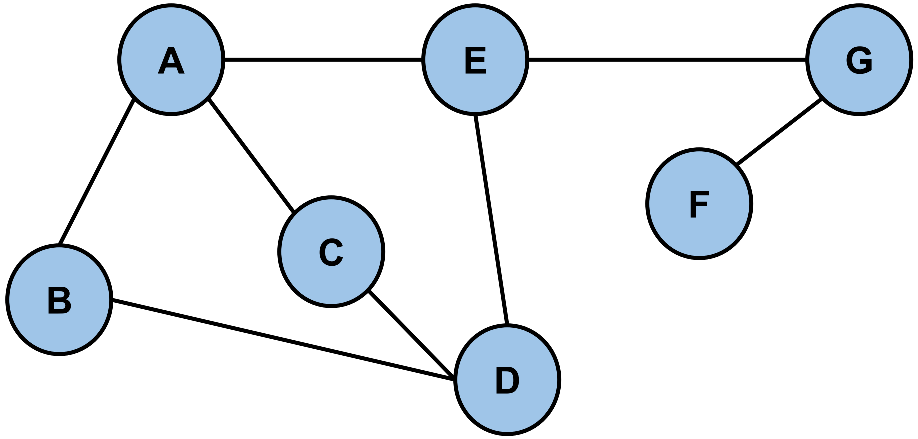


Traverse the graph starting from **A**

Stack (FILO): { }

Visited Node:

Depth First Search

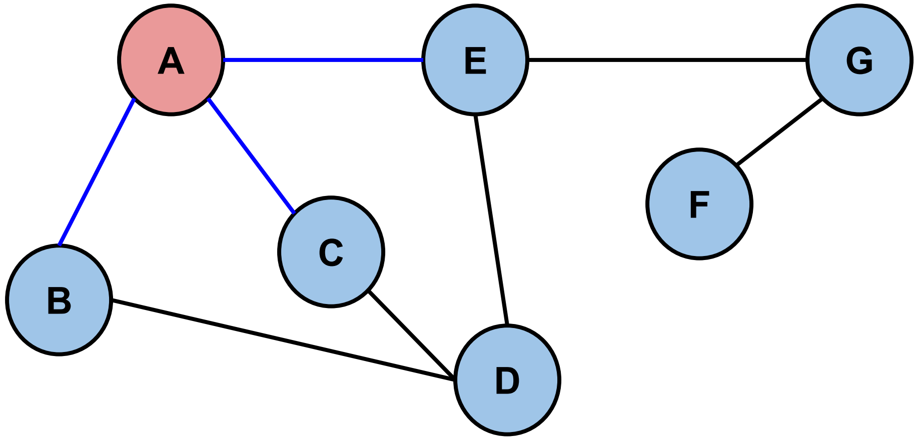


Push **A** into the stack

Stack (FILO): { A }

Visited Node:

Depth First Search

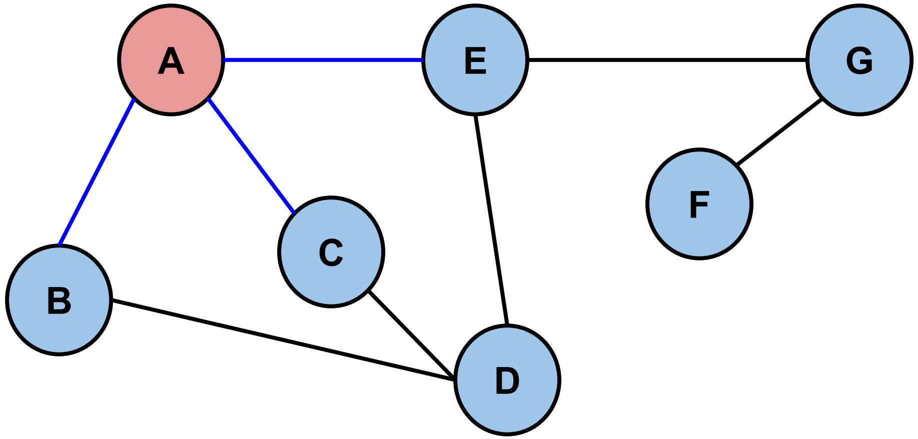


Pop A from the stack and discover its neighbors. Add A to visited nodes

Stack (FILO): { }

Visited Node: A

Depth First Search

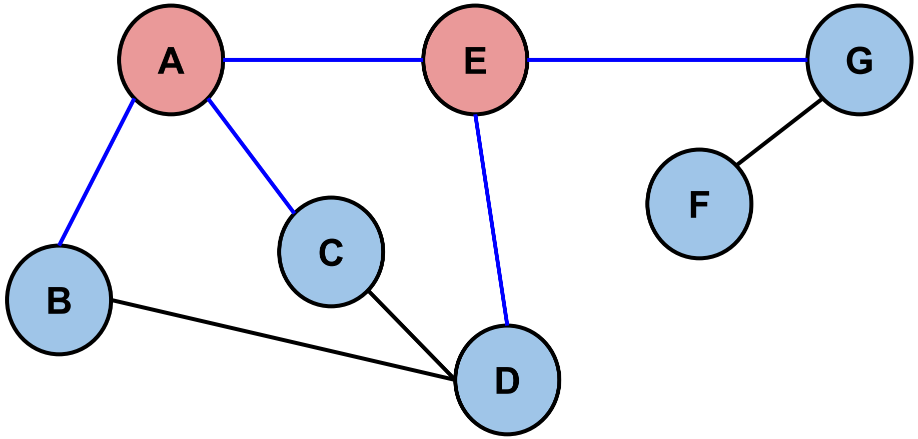


Push A's neighbors into the stack {B, C, E}

Stack (FILO): { E→ C → B}

Visited Node: A,

Depth First Search

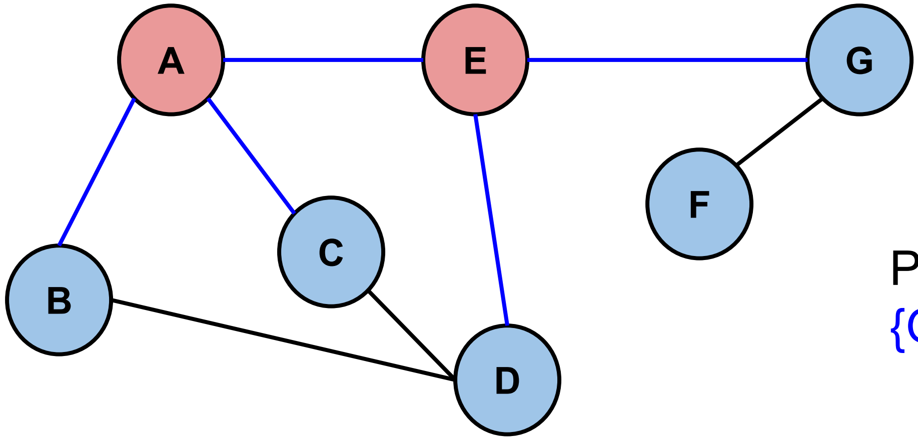


Pop E from the stack, discover its neighbors and marked as visited

Stack (FILO): { E→ C → B}

Visited Node: A, E

Depth First Search

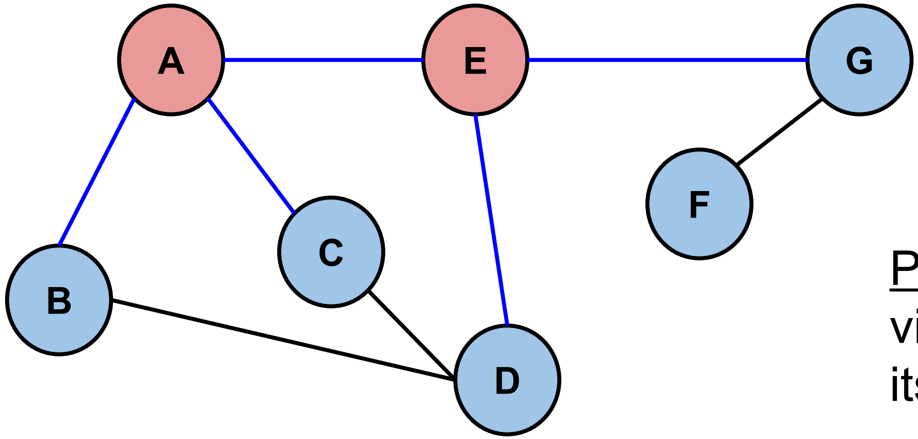


Push E's neighbors {G, D} into the stack

Stack (FILO): { G → D→ C → B}

Visited Node: A, E

Depth First Search

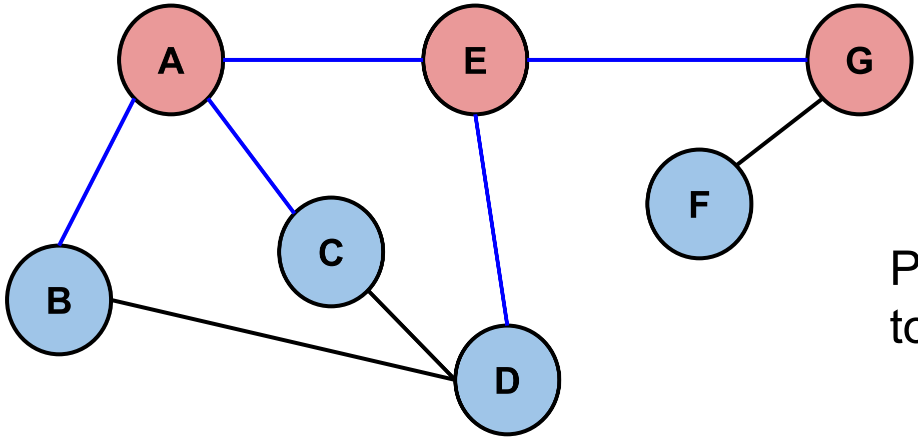


Pop G, marked as visited and discover its neighbors

Stack (FILO): { G → D→ C → B}

Visited Node: A, E

Depth First Search

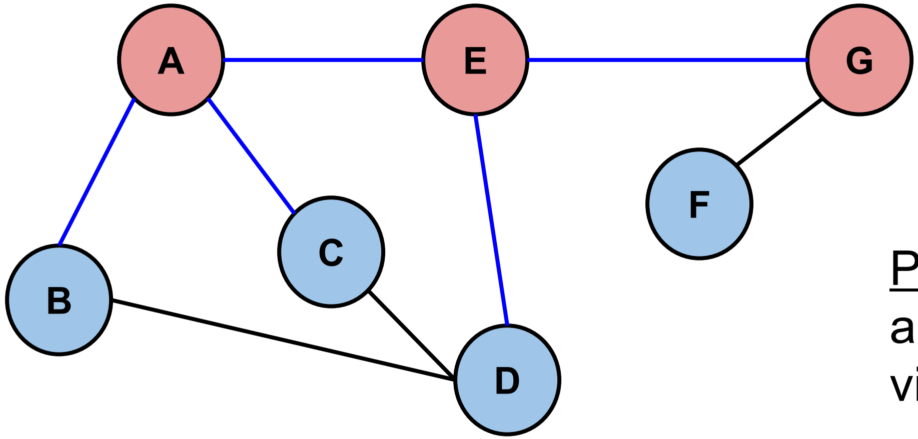


Push G's neighbors to the stack.

Stack (FILO): { F → D→ C → B}

Visited Node: A, E, G

Depth First Search

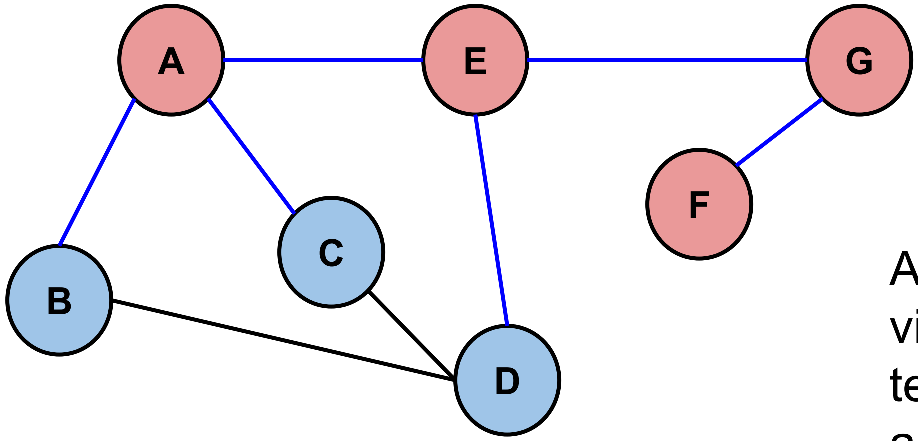


Pop F from the stack and marked as visited.

Stack (FILO): { F → D→ C → B}

Visited Node: A, E, G

Depth First Search

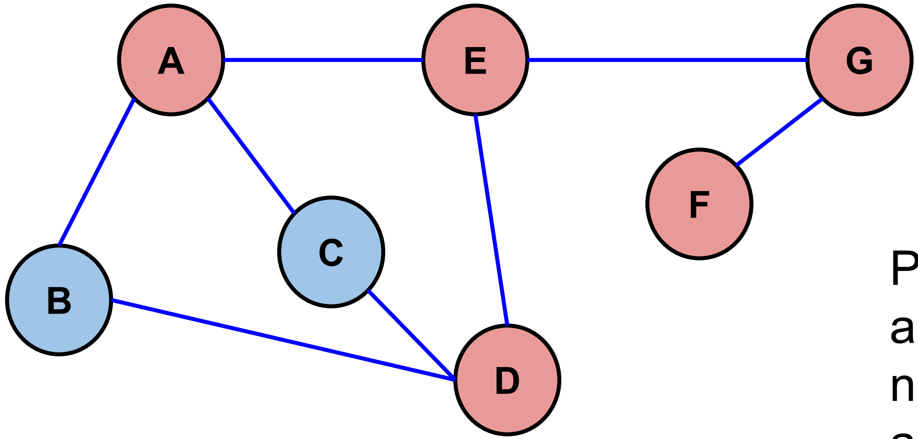


All F neighbors are visited. F is a terminal point we do a backtrack.

Stack (FILO): { D→ C → B}

Visited Node: A, E, G, F

Depth First Search

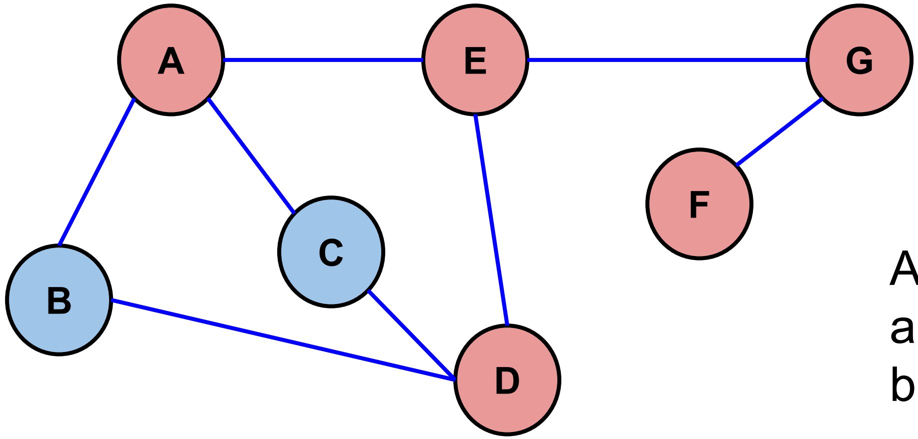


Pop D from the stack and discover its neighbors. Marked as visited

Stack (FILO): { D→ C → B}

Visited Node: A, E, G, F, D

Depth First Search

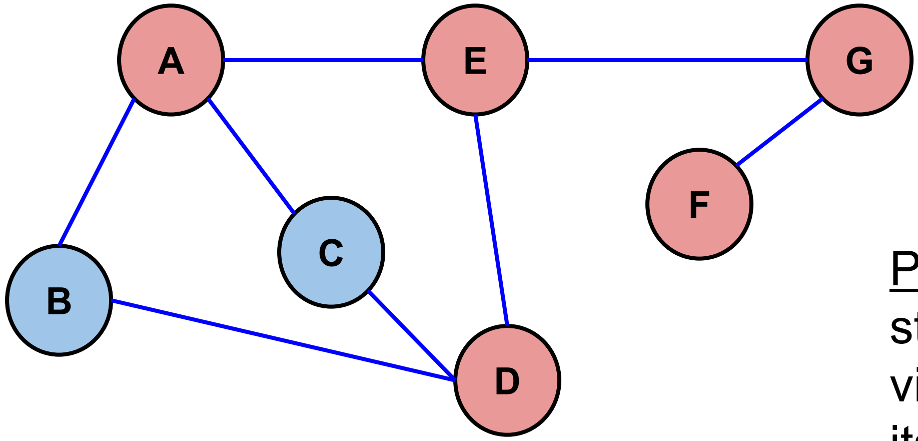


All neighbors of D are visited we backtrack.

Stack (FILO): { C → B}

Visited Node: A, E, G, F, D

Depth First Search

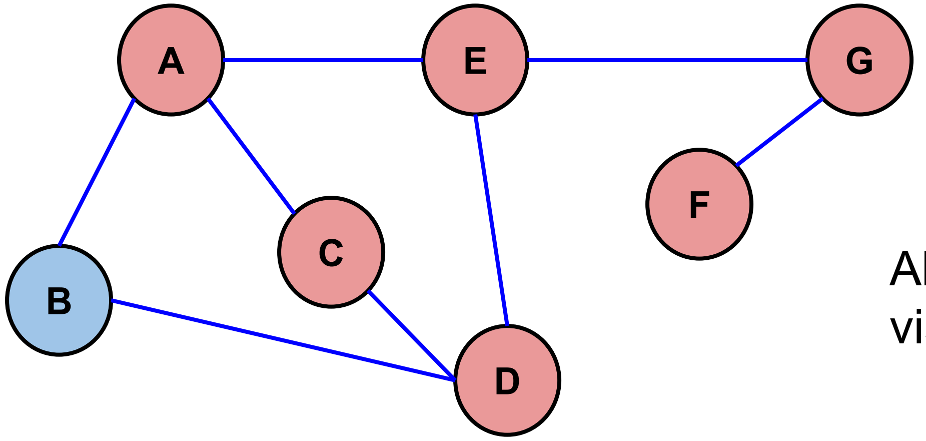


Pop C from the stack, marked as visited and discover its neighbors

Stack (FILO): { C → B}

Visited Node: A, E, G, F, D

Depth First Search

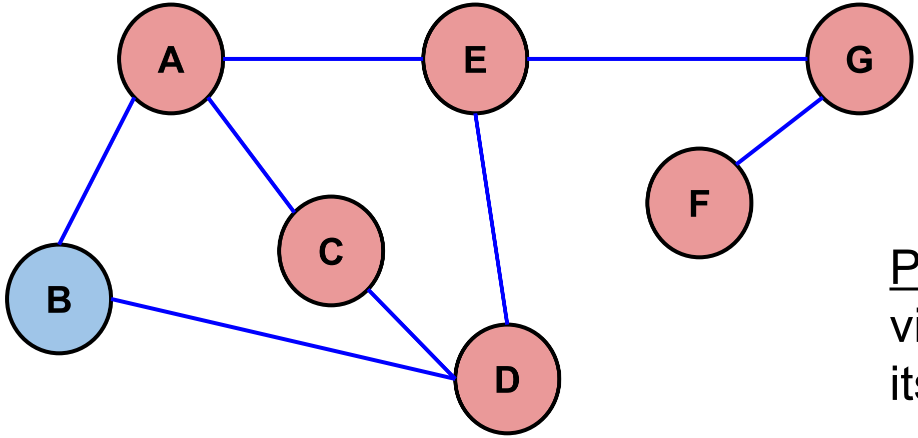


All C's neighbors are visited.

Stack (FILO): { B }

Visited Node: A, E, G, F, D, C

Depth First Search

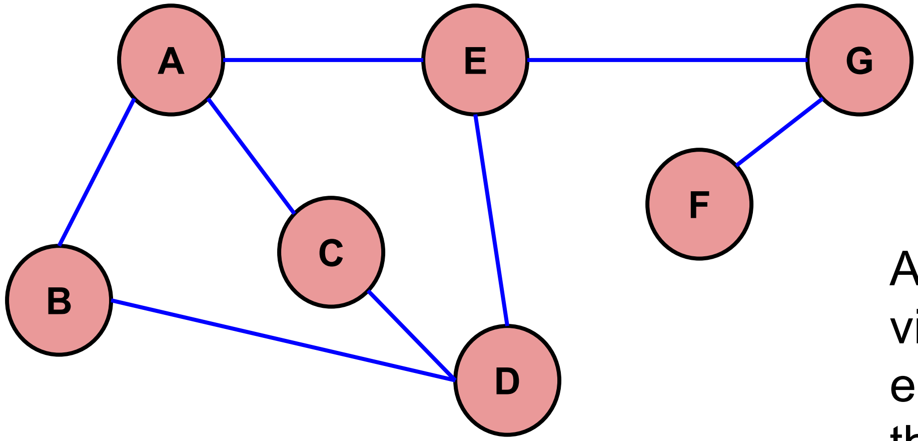


Pop B, marked as visited and discover its neighbors

Stack (FILO): { B }

Visited Node: A, E, G, F, D, C

Depth First Search



All B's neighbors are visited. The stack is empty, we visited all the nodes in the graph

Stack (FILO): { B }

Visited Node: A, E, G, F, D, C, B

Depth First Search: Pseudo Code

```
def DFS(G,v):
    # let S be a stack
    for u in V of G:
        visited[u] = False
    S = stack()
    S.push(v)
    while S.isEmpty() == False:
        v = S.pop
        if v not in visited:
            visited[v] = True
            for w in neighbours of v and visited[w] is False:
                S.push(w)
```

Shortest Path Algorithms

In graph theory, the shortest path problem, is the problem of finding the shortest path between two vertices in a weight graph.

If the graph is unweighted then the shortest path is the path with least number of edges.

For weight graphs there are many algorithms the most common ones are:

- Bellman Ford's Algorithm (works with negative weight)
- Dijkstra's Algorithm (works with positive weights)
- Floyd–Warshall's Algorithm (works with negative weights, but no egative cycle)

Dijkstra’s Algorithm

The algorithm find shortest paths from a source vertex v to all other vertices in the graph.

Solve the single-source shortest path problem.

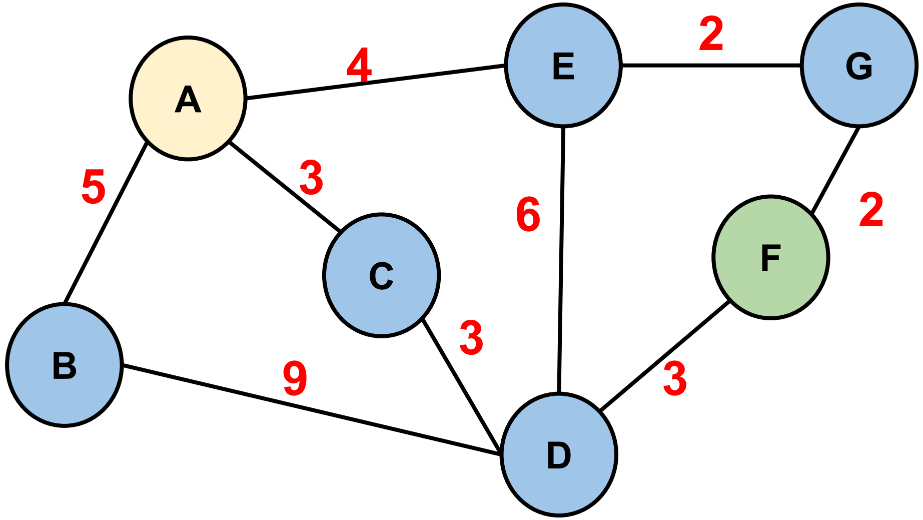
The graph must be connected

All the weights must have positive weights (does not work with negative weights)

Dijkstra's Algorithm has run-time complexity of $O(|E| + |V| \log |V|)$

Dijkstra's Algorithm

Find the shortest path from A to F



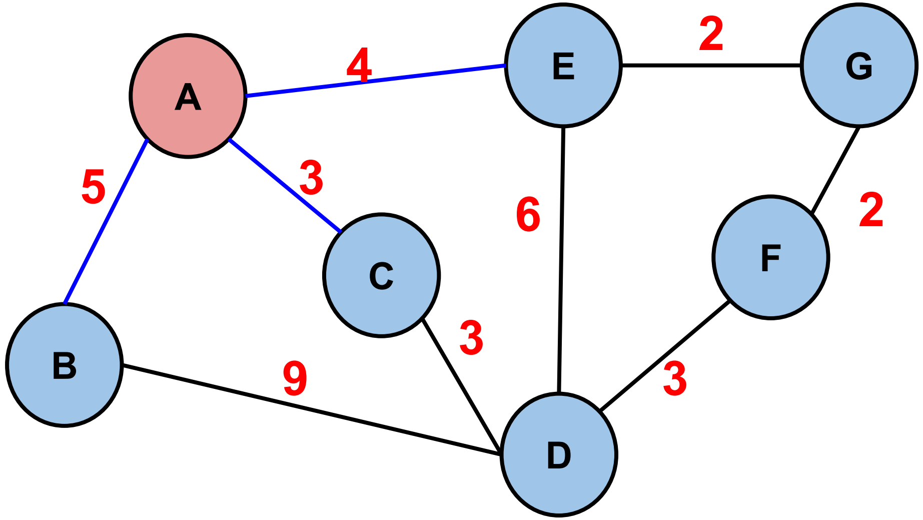
Unvisited Node: {A, B, C, D, E, F, G}

Visited Node: { }

Vertex	Shortest Distance from A	Previous Vertex
A	0	A
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	
G	∞	

Dijkstra's Algorithm

Find the shortest path from A to F



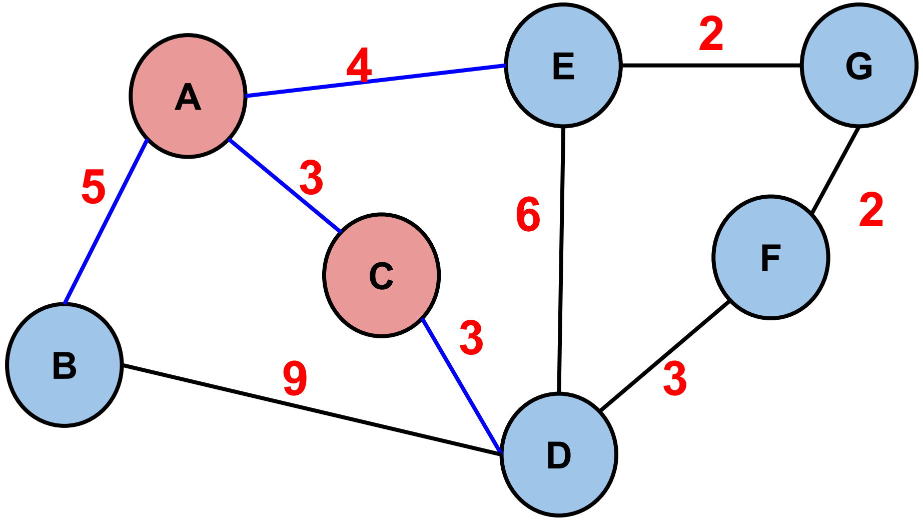
Unvisited Node: {B, C, D, E, F, G}

Visited Node: { A }

Vertex	Shortest Distance from A	Previous Vertex
A	0	A
B	5	A
C	3	A
D	∞	-
E	4	A
F	∞	-
G	∞	-

Dijkstra’s Algorithm

Find the shortest path from A to F



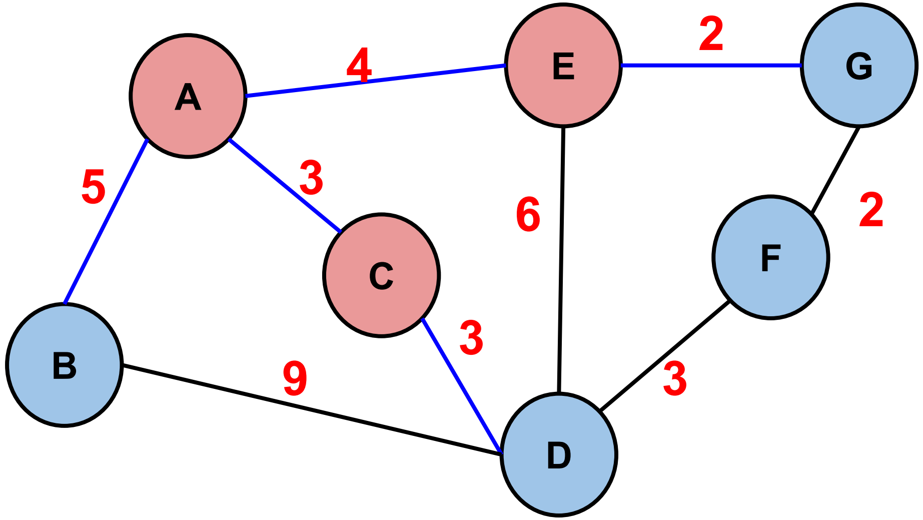
Unvisited Node: {B, D, E, F, G}

Visited Node: { A, C }

Vertex	Shortest Distance from A	Previous Vertex
A	0	A
B	**5**	A
C	**3**	A
D	**(3+3) 6**	C
E	$\boxed{\mathbf{4}}$	A
F	∞	
G	∞	

Dijkstra’s Algorithm

Find the shortest path from A to F



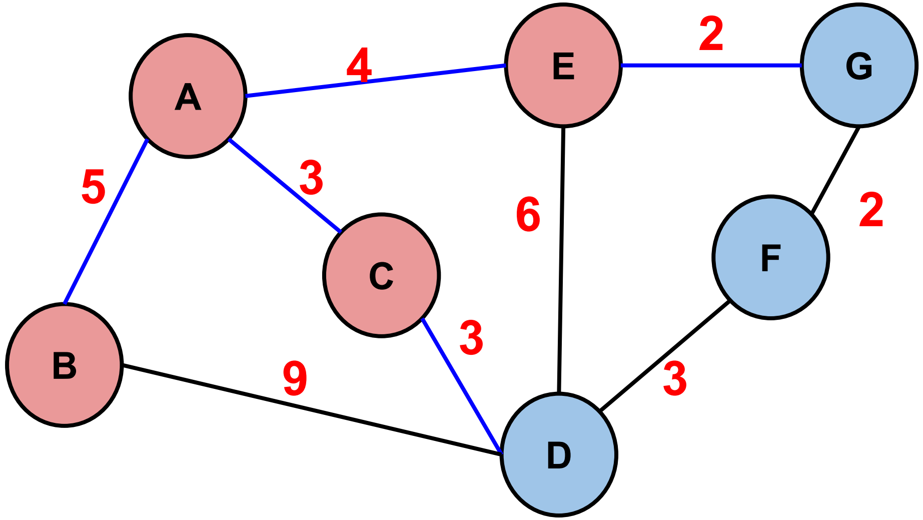
Unvisited Node: {B, D, F, G}

Visited Node: { A, C, E }

Vertex	Shortest Distance from A	Previous Vertex
A	0	A
B	**5**	A
C	**3**	A
D	**6**	C
E	**4**	A
F	∞	-
G	**(4+2) 6**	E

Dijkstra’s Algorithm

Find the shortest path from A to F



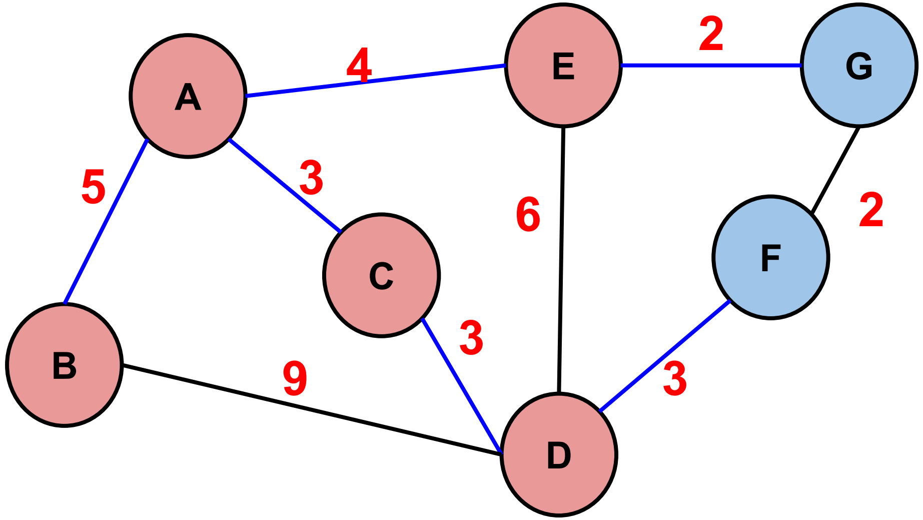
Unvisited Node: {D, F, G}

Visited Node: { A, C, E, B }

Vertex	Shortest Distance from A	Previous Vertex
A	0	A
B	**5**	A
C	**3**	A
D	**6**	C
E	**4**	A
F	∞	-
G	**6**	E

Dijkstra's Algorithm

Find the shortest path from A to F



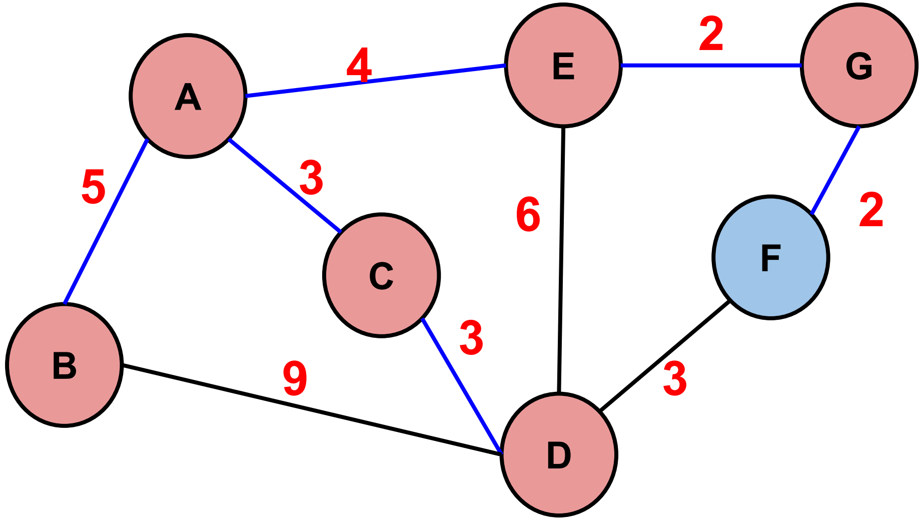
Unvisited Node: {F, G}

Visited Node: { A, C, E, B, D}

Vertex	Shortest Distance from A	Previous Vertex
A	0	A
B	**5**	A
C	**3**	A
D	**6**	C
E	**4**	A
F	**(6+3) 9**	D
G	$\boxed{\mathbf{6}}$	E

Dijkstra's Algorithm

Find the shortest path from A to F



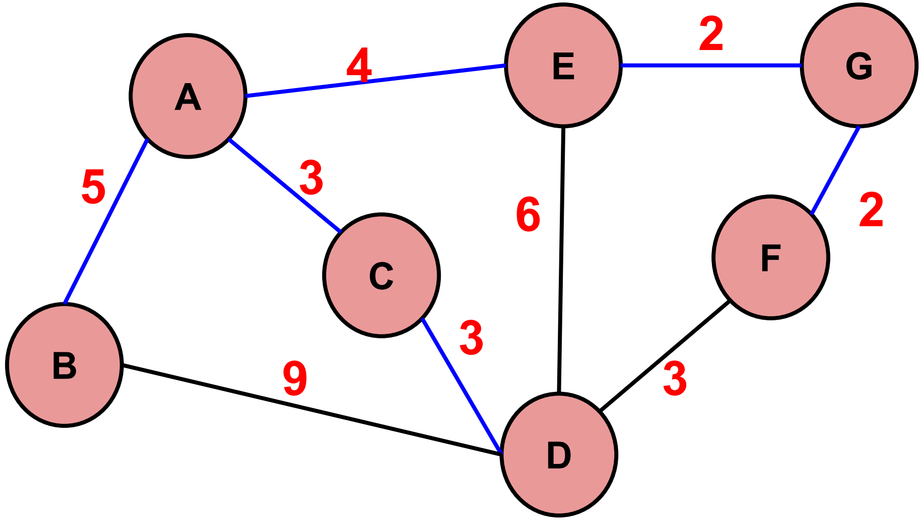
Unvisited Node: {F}

Visited Node: { A, C, E, B, D}

Vertex	**Shortest Distance from A**	**Previous Vertex**
A	0	A
B	**5**	A
C	**3**	A
D	**6**	C
E	**4**	A
F	(6+2) 8	G
G	**6**	E

Dijkstra’s Algorithm

Find the shortest path from A to F

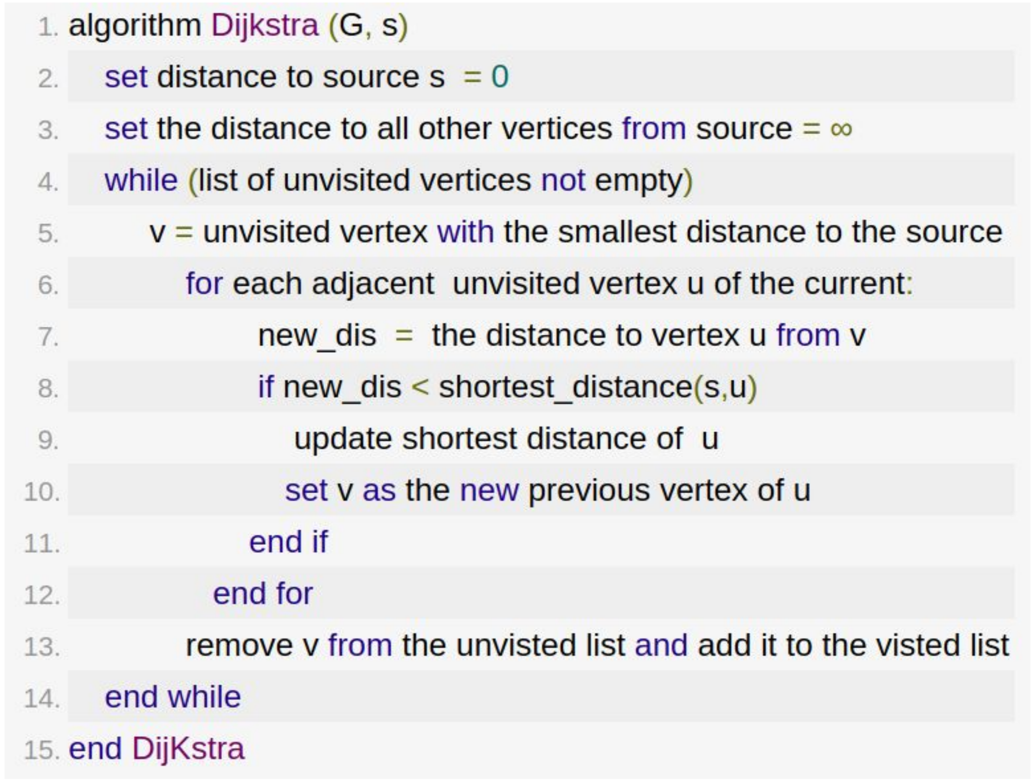


Unvisited Node: {}

Visited Node: { A, C, E, B, D, F}

Vertex	Shortest Distance from A	Previous Vertex
A	0	A
B	**5**	A
C	**3**	A
D	**6**	C
E	**4**	A
F	**8**	G
G	**6**	E

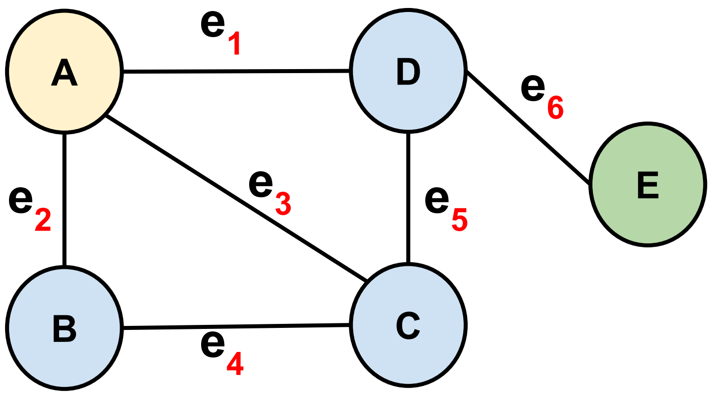
Dijkstra’s Algorithm: Pseudocode



```
algorithm Dijkstra (G, s)
    set distance to source s  = 0
    set the distance to all other vertices from source = ∞
    while (list of unvisited vertices not empty)
        v = unvisited vertex with the smallest distance to the source
            for each adjacent  unvisited vertex u of the current:
                new_dis  =  the distance to vertex u from v
                if new_dis < shortest_distance(s,u)
                    update shortest distance of  u
                    set v as the new previous vertex of u
                end if
            end for
        remove v from the unvisted list and add it to the visted list
    end while
end DijKstra
```

Self -Assessment

Write an algorithm to find all the paths from A to E in the following graph


A
D
E
B
C
e_1
e_2
e_3
e_4
e_5
e_6