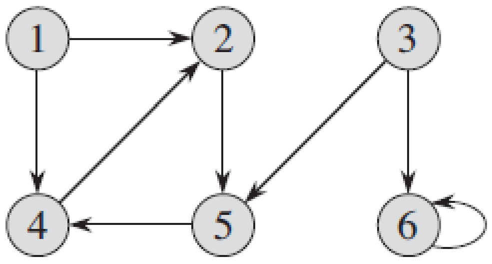
COMP 2611, DATA STRUCTURES
LECTURE 18

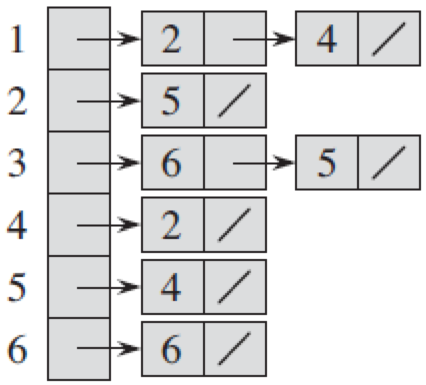
GRAPHS

- **Weighted Graphs**
- **Searching a Graph: Depth-first search**
- **Searching a Graph: Breadth-first search**
- **Dijkstra's Shortest Path Algorithm**

Graphs: Representation



(a)



(b)

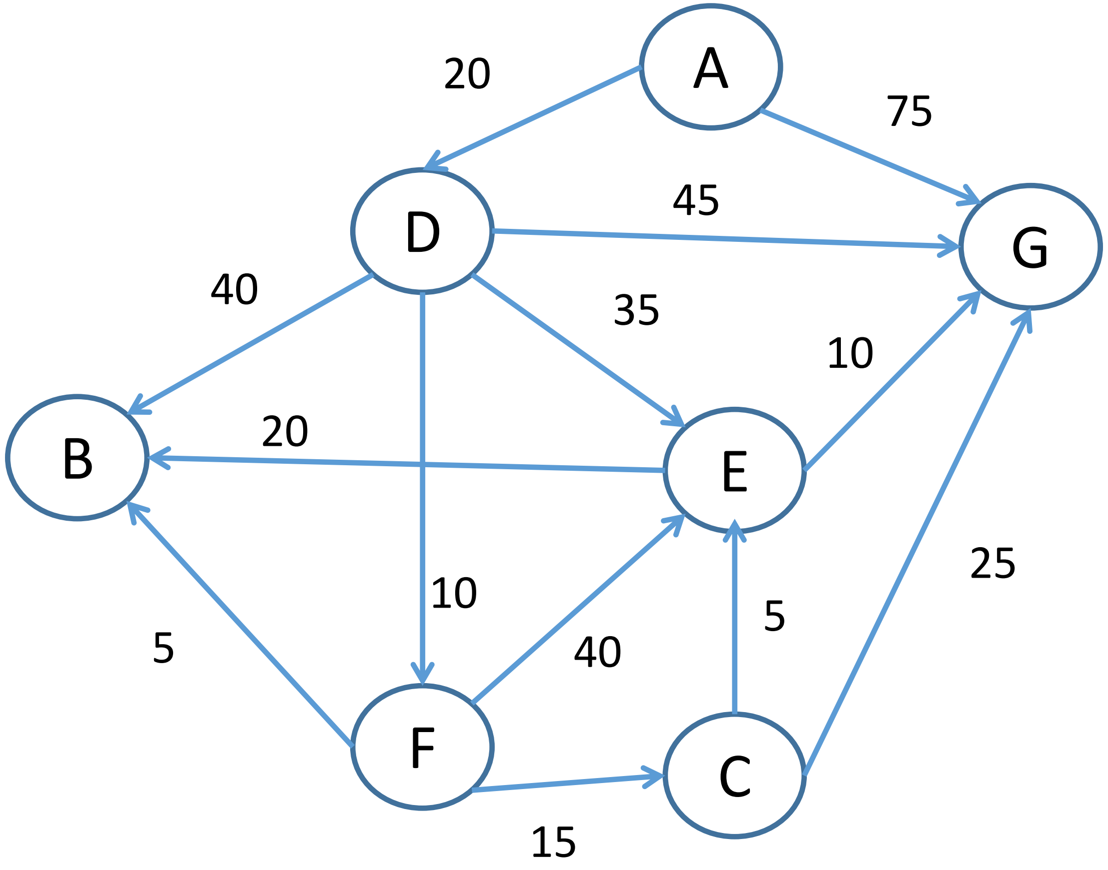
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

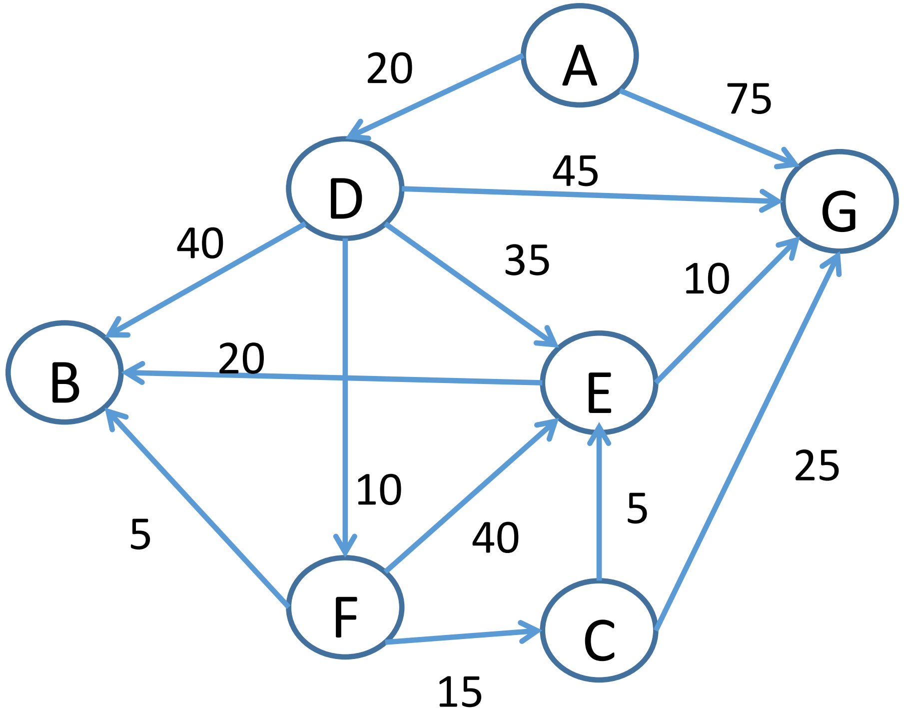
Adjacency List

Adjacency Matrix

Weighted Graphs



Weighted Graphs: Adjacency Matrix Representation



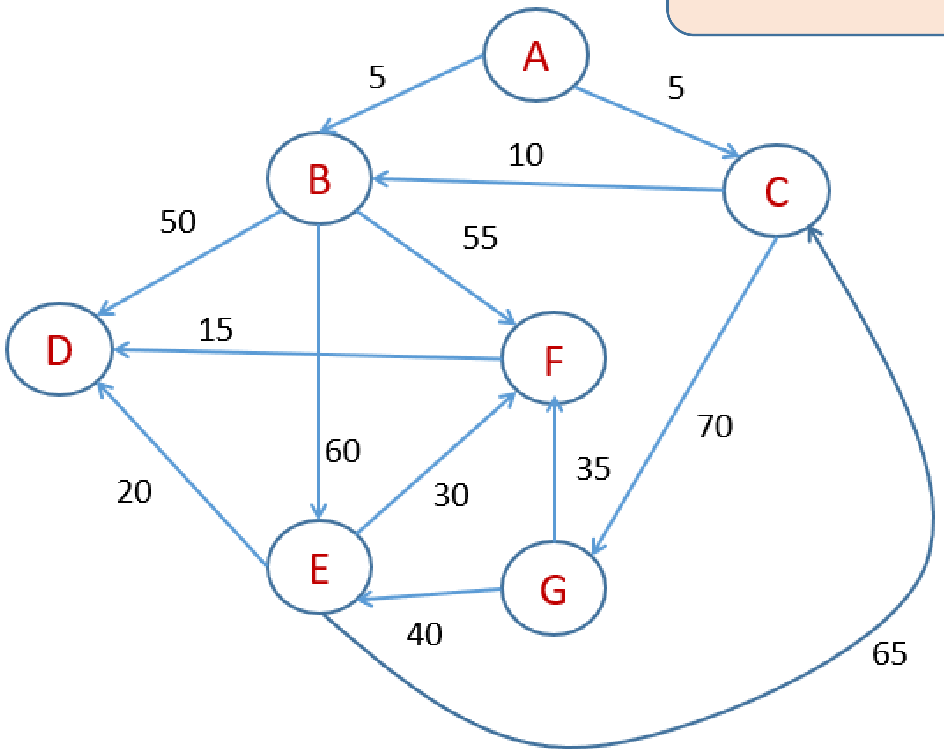
Graph

	A	B	C	D	E	F	G
A	0	∞	∞	20	∞	∞	75
B	∞	0	∞	∞	∞	∞	∞
C	∞	∞	0	∞	5	∞	25
D	∞	40	∞	0	35	10	45
E	∞	0	∞	∞	0	∞	10
F	∞	5	15	∞	40	0	∞
G	∞	∞	∞	∞	∞	∞	0

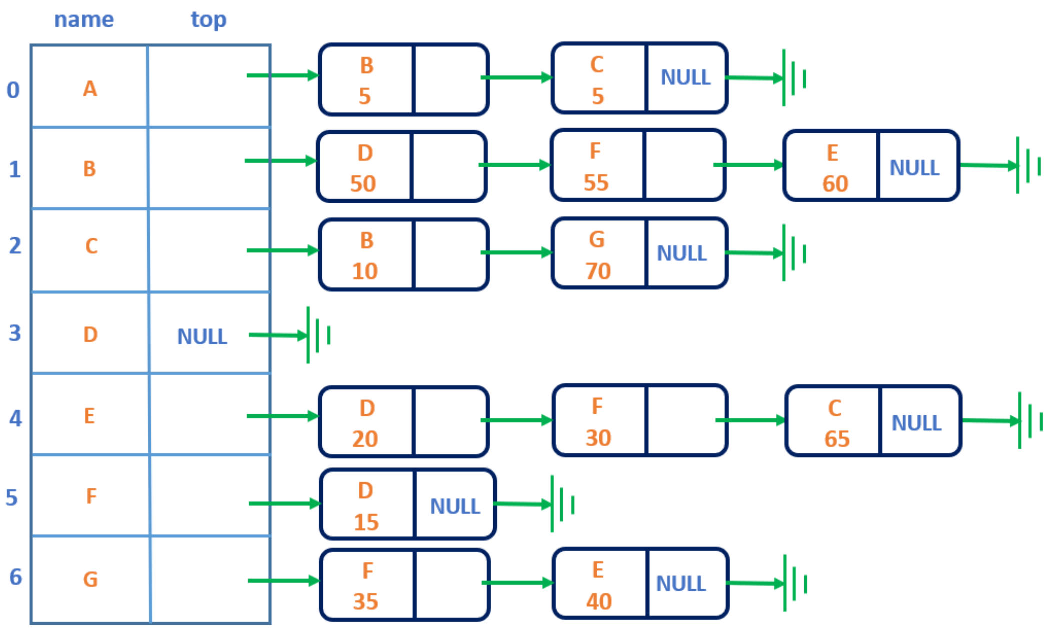
Adjacency Matrix

Graphs: Where Have I Seen This Before?

Hashing: Resolving Collisions with Chaining

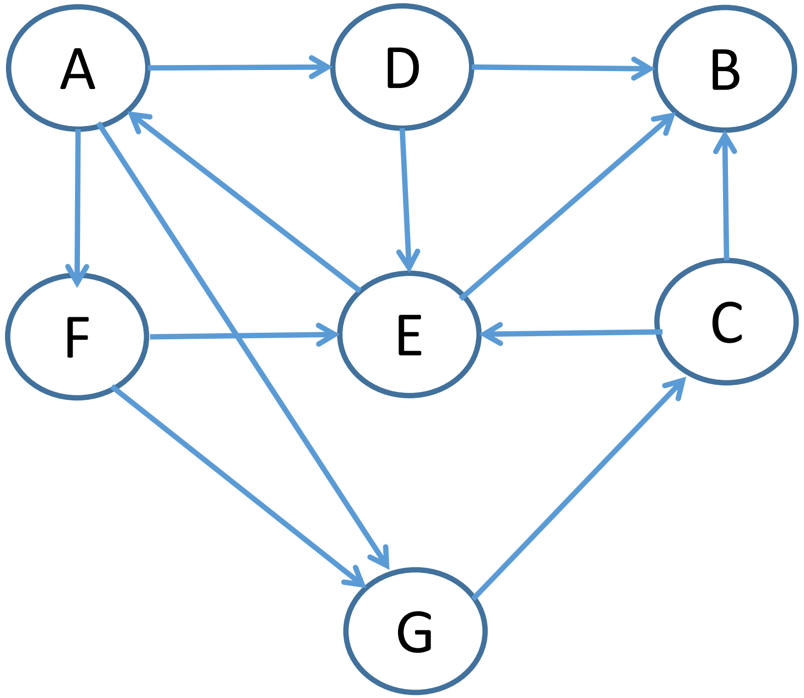


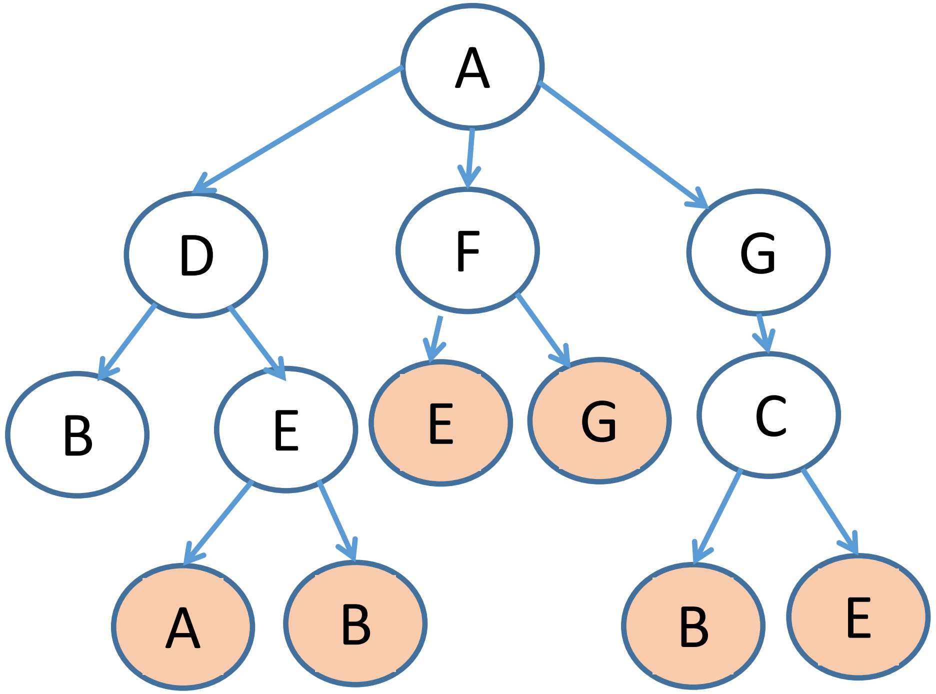
Graph



Adjacency List

Traversing a Graph: Breadth-First Search





Breadth-first traversal: A D F G B E C

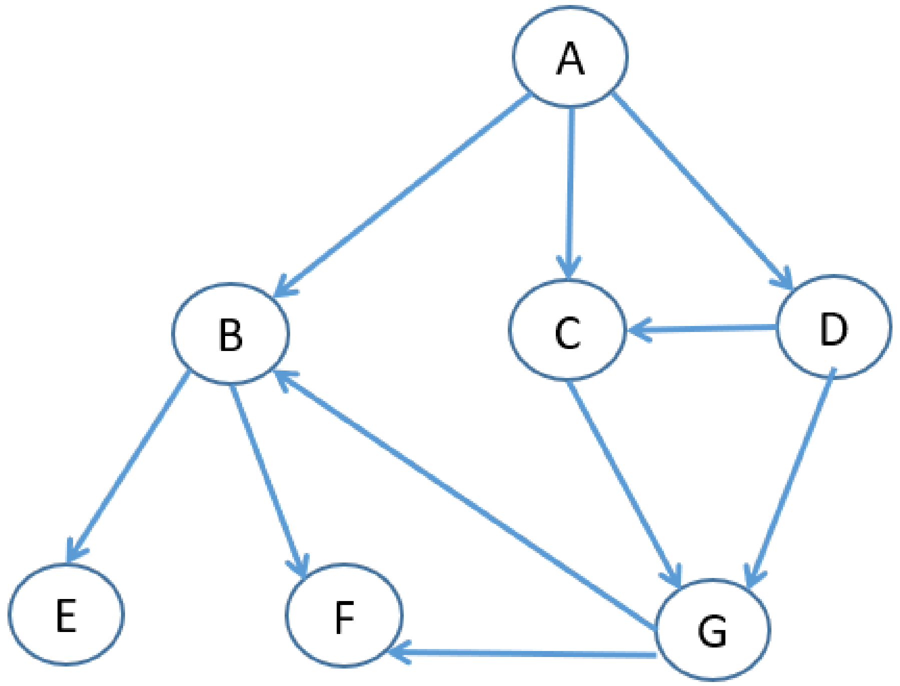
Algorithm for Breadth-First Search

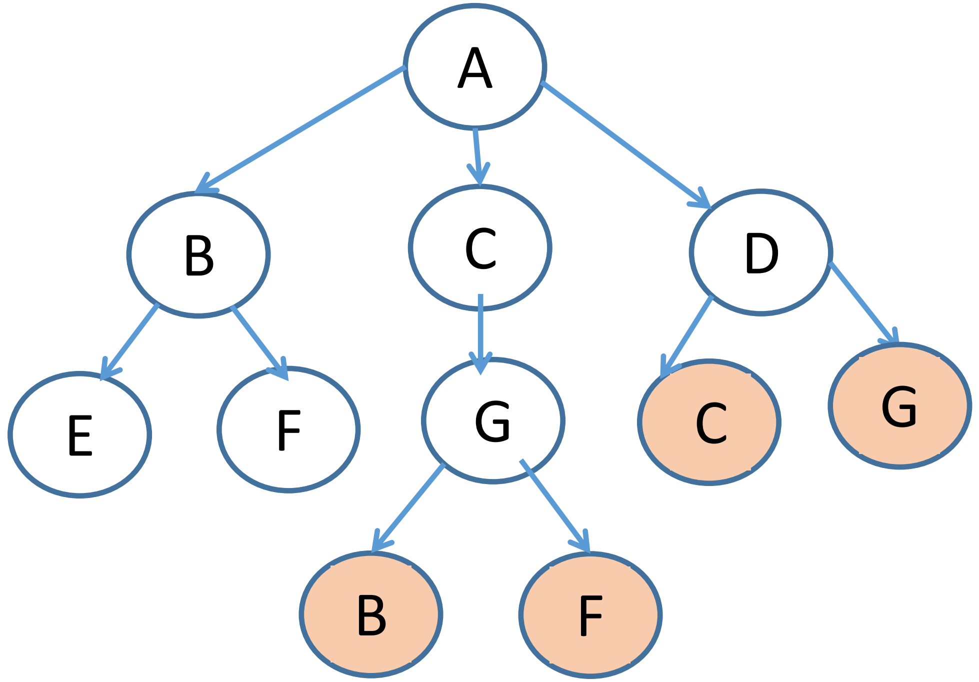
```
breadthFirstTraversal (G, source):

    for each vertex u ∈ set of vertices in G
        u.colour = WHITE

    G.vertices[source].colour = GREY

    Q = initQueue()
    enqueue(q, source)

    while Q is not empty
        u = dequeue (q)
        for each v ∈ adjacency list of u
            loc = findVertex (G, v)
            if G.vertices[loc].colour == WHITE
                G.vertices[loc].colour = GREY
                enqueue(loc)
        u.colour = BLACK
```

IMPORTANT

source and *loc* are locations in the array of vertices in *G*.

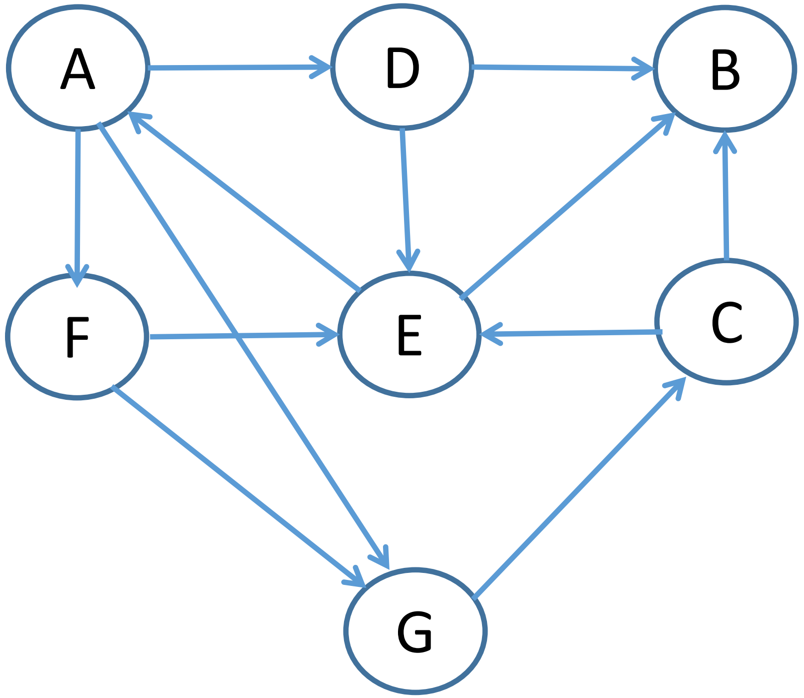
Breadth-First Search Example

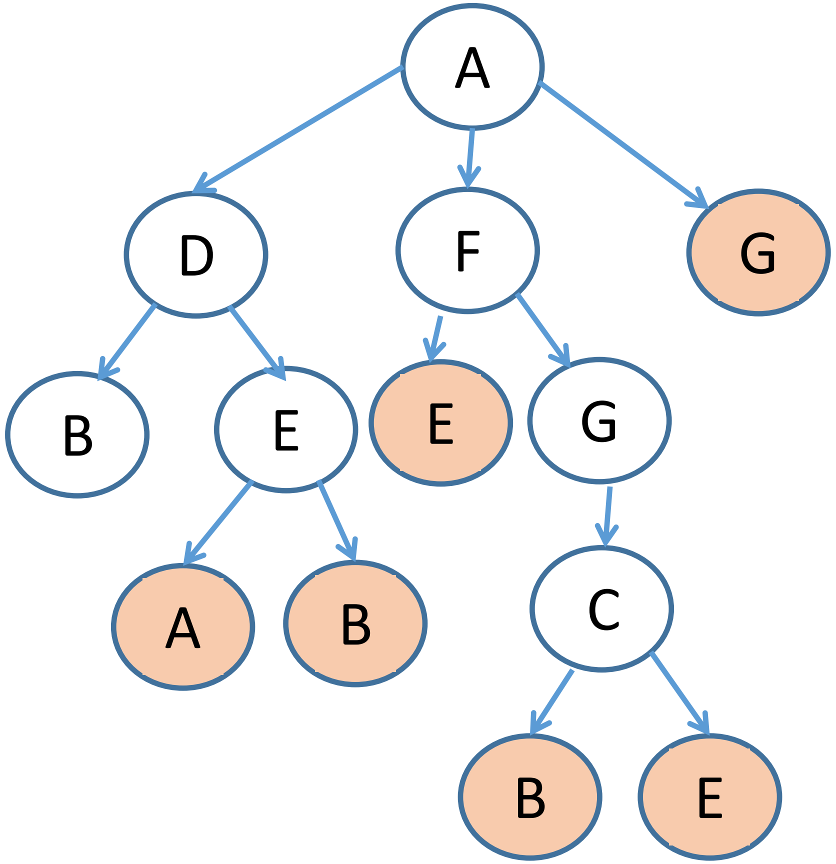




Breadth-first traversal: A B C D E F G

Traversing a Graph: Depth-First Search



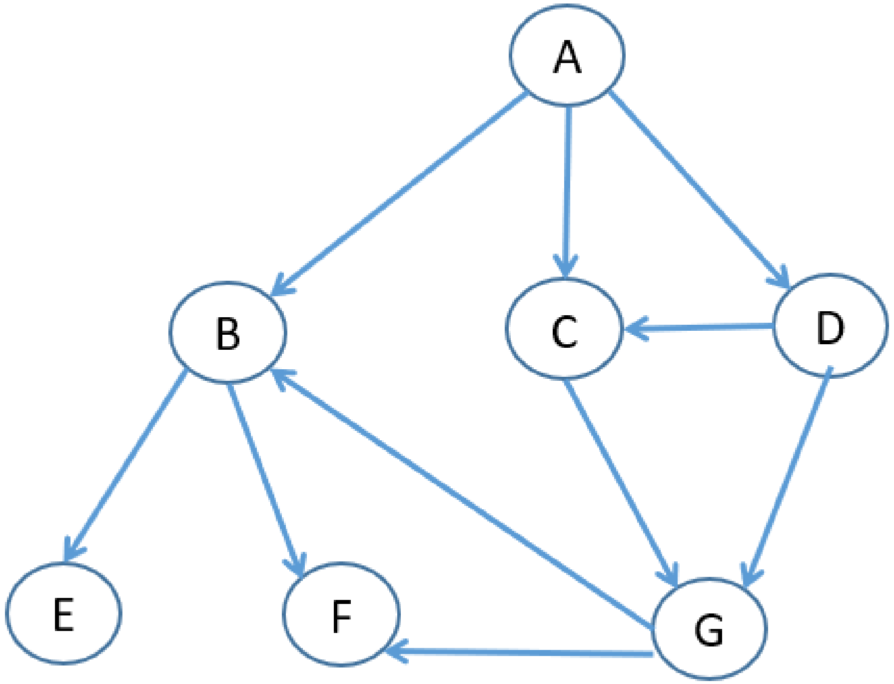


Depth-first traversal: A D B E F G C

Algorithm for Depth-First Search

```
depthFirstTraversal (G):

    for each vertex u ∈ set of vertices in G (V)
        u.colour = WHITE

    for each vertex u ∈ set of vertices in G (V)
        if u.colour == WHITE
            dfTraverse(G, u)

dfTraverse (G, u):

    u.colour = GREY
    for each v ∈ adjacency list of u
        if v.colour == WHITE
            dfTraverse (G, v)

    u.colour = BLACK
```

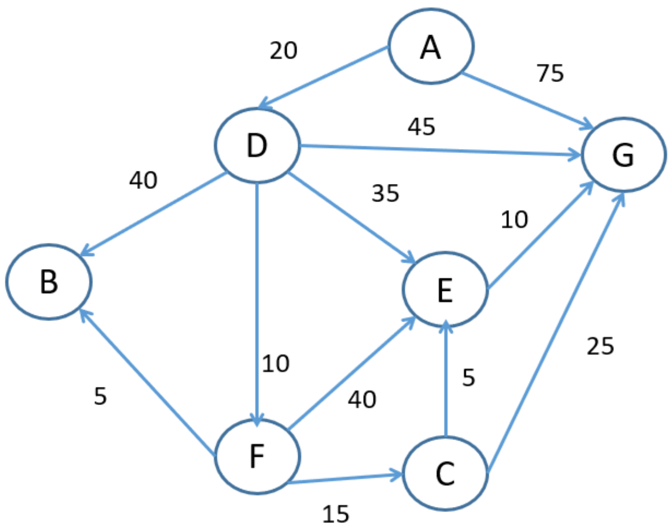
Depth-First Search Example



Find the depth-first traversal of this graph.

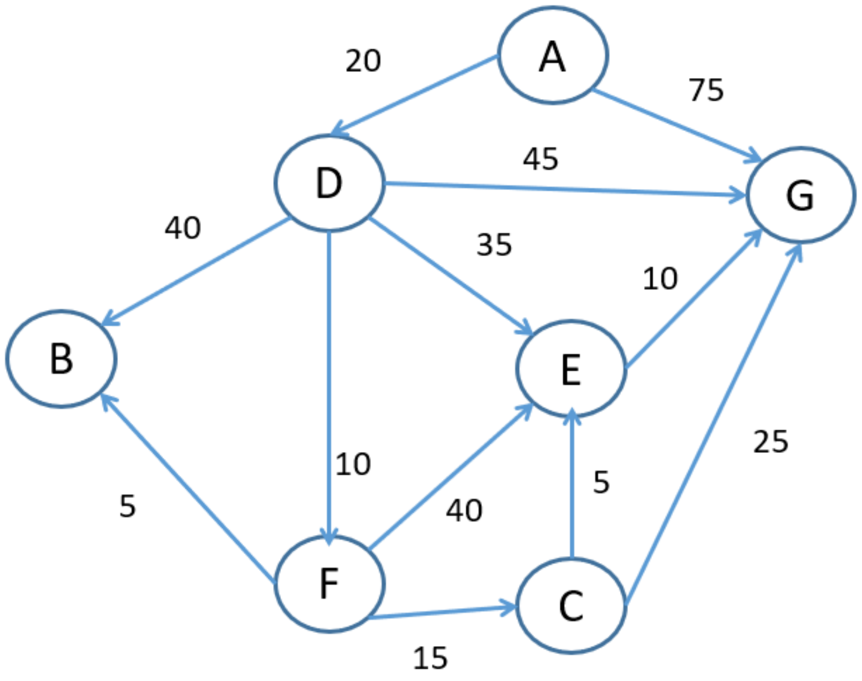
Dijkstra’s Shortest Path Algorithm

- Dijkstra’s algorithm can be used to find the shortest path from a source vertex (e.g., *A*) to every other vertex in the graph.
- The algorithm assumes that the edge weights are non-negative.



vertex	A	B	C	D	E	F	G
parent	nil	nil	nil	nil	nil	nil	nil
cost	0	∞	∞	∞	∞	∞	∞

- V.cost holds the current cost of a path from *A* to a vertex *V*
- V.parent holds the parent of *V* on the current shortest path from *A* to *V*.
- A min-priority queue, Q, will hold vertices based on their current cost. Initially, all the verticles will be placed on *Q*. Vertex *A*, with a cost of 0, will be at the top.

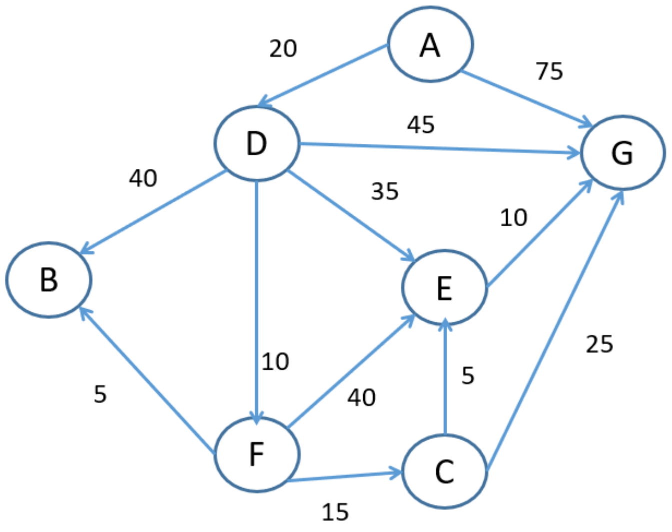


vertex	A	B	C	D	E	F	G
parent	nil	nil	nil	nil	nil	nil	nil
cost	0	∞	∞	∞	∞	∞	∞

Min-Priority Queue

A	

- Take A off the queue.
- Consider all the edges leaving A. We will process them in alphabetical order. The edge (A, D) gives us a path to D at a cost of 20. This is lower than the current cost to D (∞) so we update the cost to 20, set the parent of D to A.
- The edge (A, G) gives us a path to G at a cost of 75. This is lower than the current cost to G (∞) so we update the cost to 75, set the parent of G to A.



vertex	A	B	C	D	E	F	G
parent	nil	nil	nil	nil	nil	nil	nil
cost	0	∞	∞	∞	∞	∞	∞

vertex	A	B	C	D	E	F	G
parent	nil	nil	nil	A	nil	nil	A
cost	0	∞	∞	20	∞	∞	75

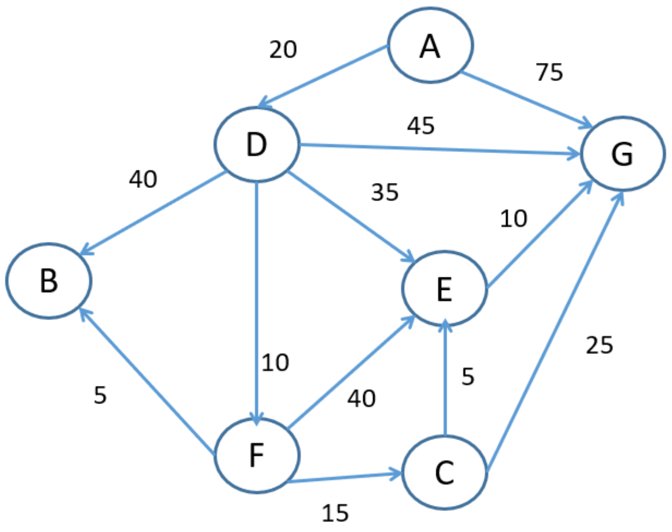
Min-Priority Queue

D

- Take D off the queue.

- Consider all the edges leaving D. The edge (D, B) gives us a path to B at a cost of 20 + 40 = 60 (cost to D + weight of (D, B)). This is lower than the current cost to B (∞) so we update the cost to 60, set the parent of B to D.

- The edge (D, E) gives us a path to E at a cost of 20 + 35 = 55 (cost to D + weight of (D, E)). This is lower than the current cost to E (∞) so we update the cost to 55, set the parent of E to D.



vertex	A	B	C	D	E	F	G
parent	nil	nil	nil	A	nil	nil	A
cost	0	∞	∞	20	∞	∞	75

vertex	A	B	C	D	E	F	G
parent	nil	D	nil	A	D	nil	A
cost	0	60	∞	20	55	∞	75

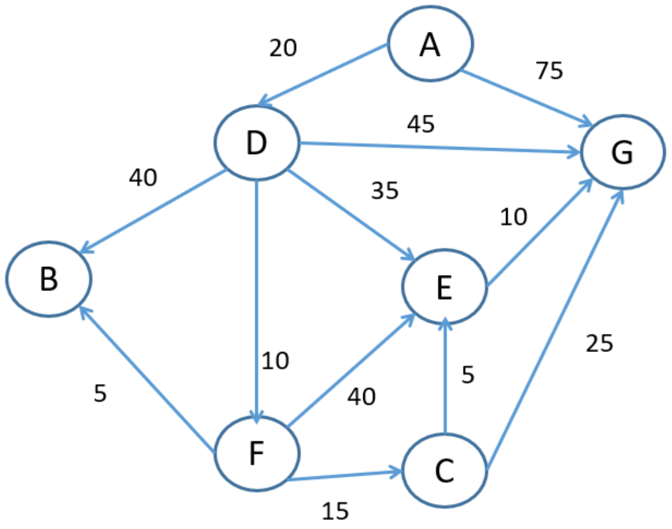
Min-Priority Queue

D	

- (Still processing D)

- The edge (D, F) gives us a path to F at a cost of 20 + 10 = 30 (cost to D + weight of (D, F)). This is lower than the current cost to F (∞) so we update the cost to 30, set the parent of F to D.

- The edge (D, G) gives us a path to G at a cost of 20 + 45 = 65 (cost to D + weight of (D, G)). This is lower than the current cost to G (75) so we update the cost to 65, set the parent of G to D.



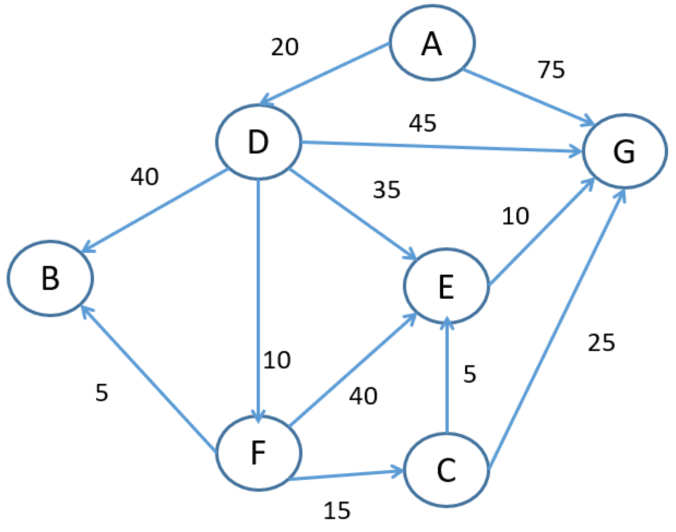
vertex	A	B	C	D	E	F	G
parent	nil	D	nil	A	D	nil	A
cost	0	60	∞	20	55	∞	75

vertex	A	B	C	D	E	F	G
parent	nil	D	nil	A	D	D	D
cost	0	60	∞	20	55	30	65

Min-Priority Queue

F	

- Take F off the queue. Consider all the edges leaving F.
- The edge (F, B) gives us a path to B at a cost of 30 + 5 = 35 (cost to F + weight of (F, B)). This is lower than the current cost to B (60) so we update the cost to 35, set the parent of B to F.
- The edge (F, C) gives us a path to C at a cost of 30 + 15 = 45 (cost to F + weight of (F, C)). This is lower than the current cost to C (∞) so we update the cost to 45, set the parent of C to F.



vertex	A	B	C	D	E	F	G
parent	nil	D	nil	A	D	D	D
cost	0	60	∞	20	55	30	65

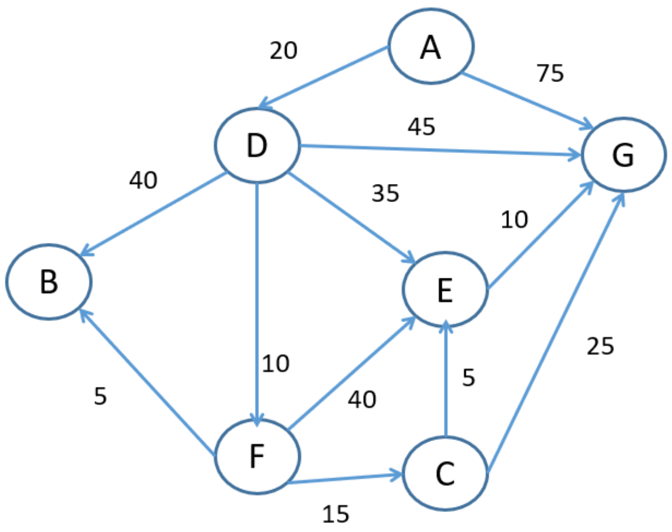
vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	D	D	D
cost	0	35	45	20	55	30	65

Min-Priority Queue

F	

- (Still processing F)

- The edge (F, E) gives us a path to E at a cost of 30 + 40 = 70 (cost to F + weight of (F, E)). This is higher than the current cost to E (55) so we leave E as it is.



vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	D	D	D
cost	0	35	45	20	55	30	65

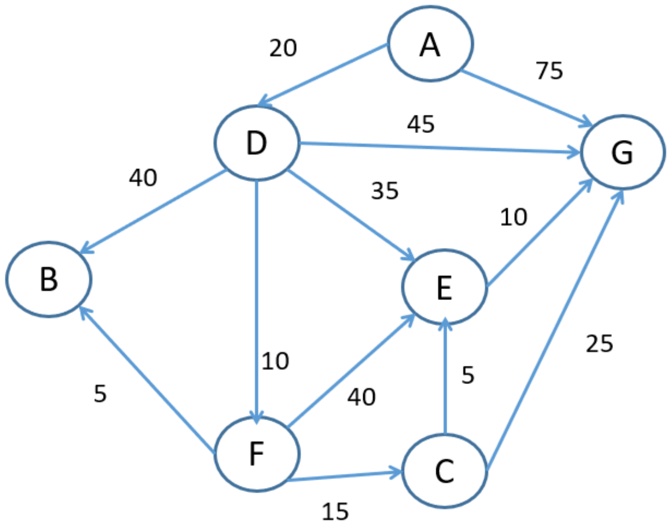
vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	D	D	D
cost	0	35	45	20	55	30	65

Min-Priority Queue

B	

- Take B off the queue. Consider all the edges leaving B.

- There are no edges leaving B. So, we take the next element off the queue.



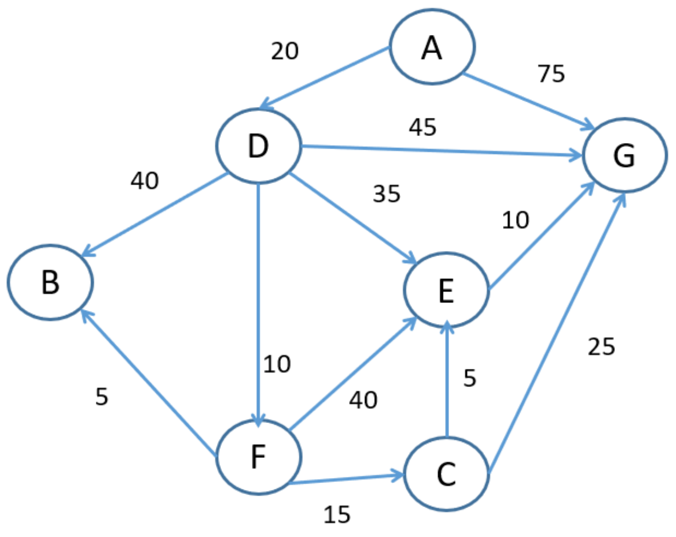
vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	D	D	D
cost	0	35	45	20	55	30	65

vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	D	D	D
cost	0	35	45	20	55	30	65

Min-Priority Queue

C	

- Take C off the queue. Consider all the edges leaving C.
- The edge (C, E) gives us a path to E at a cost of 45 + 5 = 50 (cost to C + weight of (C, E). This is lower than the current cost to E (55) so we update the cost to 50, set the parent of E to C.
- The edge (C, G) gives us a path to G at a cost of 45 + 25 = 70 (cost to C + weight of (C, G). This is higher than the current cost to G (65) so we leave G as it is.



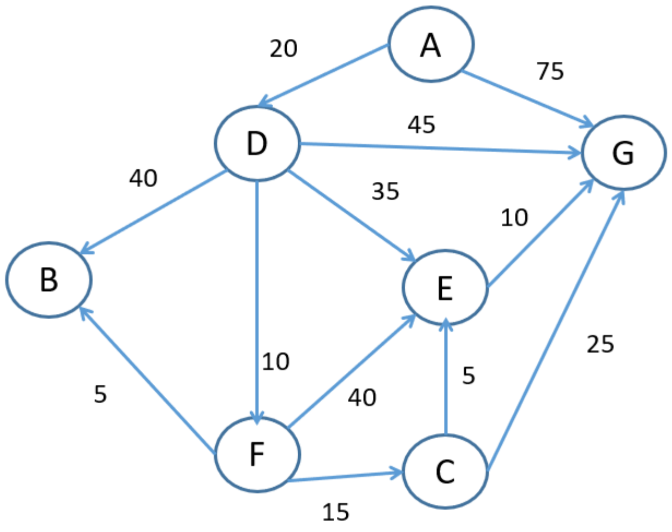
vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	D	D	D
cost	0	35	45	20	55	30	65

vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	C	D	D
cost	0	35	45	20	50	30	65

Min-Priority Queue

E	

- Take E off the queue. Consider all the edges leaving E.
- The edge (E, G) gives us a path to G at a cost of 50 + 10 = 60 (cost to E + weight of (E, G). This is lower than the current cost to G (65) so we update the cost to 60, set the parent of G to E.



vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	C	D	D
cost	0	35	45	20	50	30	65

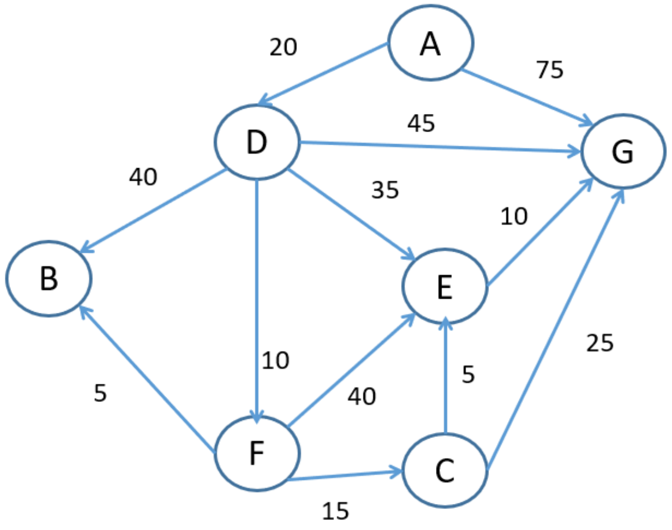
vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	C	D	E
cost	0	35	45	20	50	30	60

- Take G off the queue. Consider all the edges leaving G.
- There are no edges leaving G.
- The queue is now empty, so the algorithm terminates. The results are:

Cost to B: 35, Path: A → D → F → B
Cost to C: 45, Path: A → D → F → C
Cost to D: 20, Path: A → D
Cost to E: 50, Path: A → D → F → C → E
Cost to F: 30, Path: A → D → F
Cost to G: 60, Path: A → D → F → C → E → G

Min-Priority Queue

G	

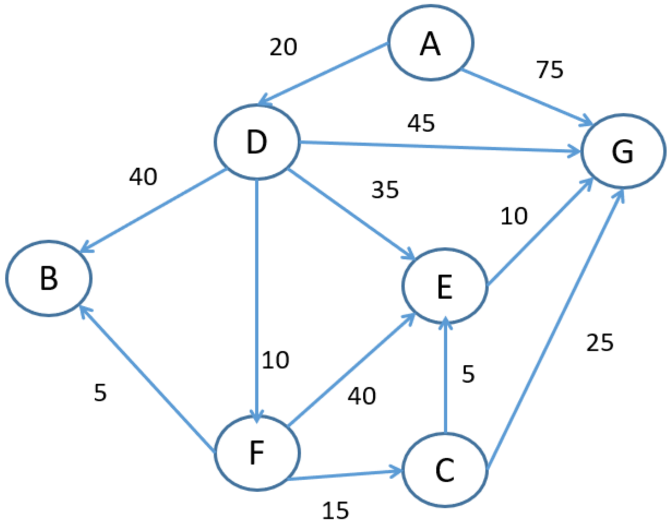


vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	C	D	E
cost	0	35	45	20	50	30	60

vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	C	D	E
cost	0	35	45	20	50	30	60

- Given table, how to get paths?

```
Cost to B: 35, Path: A → D → F → B
Cost to C: 45, Path: A → D → F → C
Cost to D: 20, Path: A → D
Cost to E: 50, Path: A → D → F → C → E
Cost to F: 30, Path: A → D → F
Cost to G: 60, Path: A → D → F → C → E → G
```



vertex	A	B	C	D	E	F	G
parent	nil	F	F	A	C	D	E
cost	0	35	45	20	50	30	60

Dijkstra's Shortest Path Algorithm

- Dijkstra's Algorithm is an example of a *single-source shortest path algorithm* which finds the shortest path from a source node to a destination node.

- There are others such as the Bellman-Ford Algorithm.