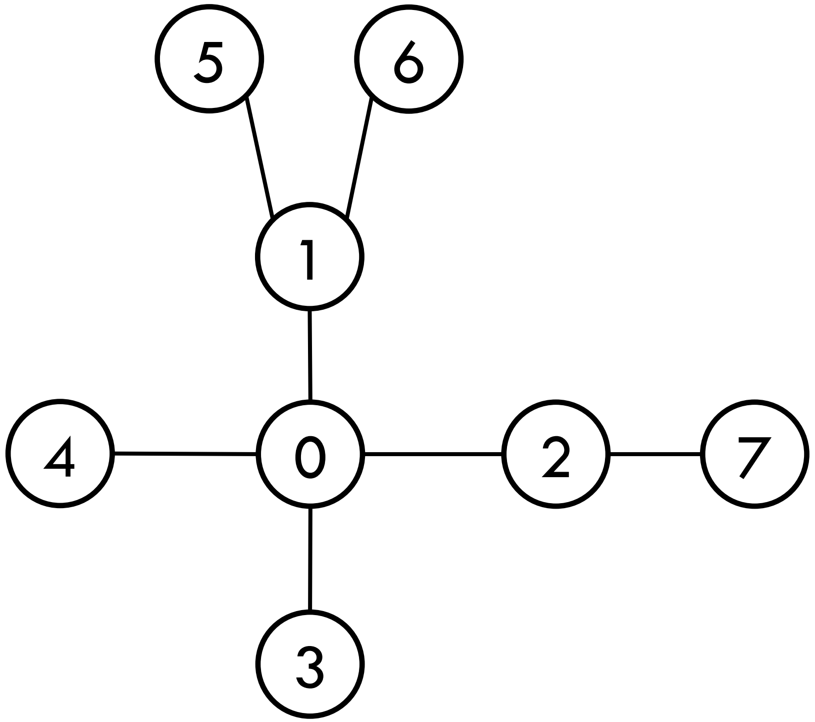
ALGORITHMS AND DATA STRUCTURES
LECTURE 9 – GRAPHS (PART II)

Aigerim Aibatbek
Aigerim.aibatbek@astanait.edu.kz

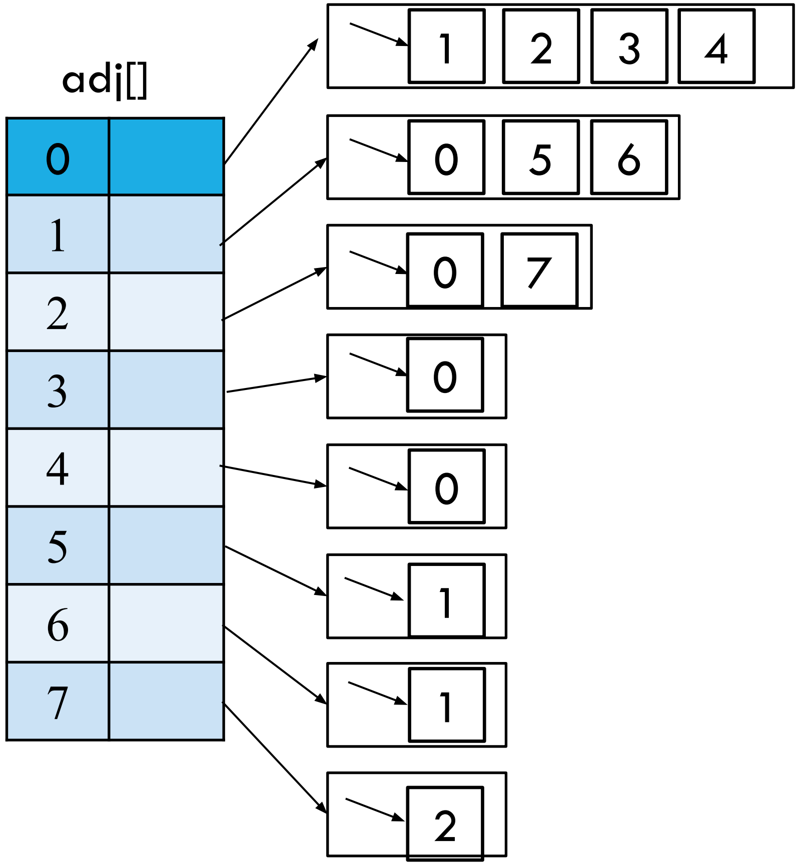
CONTENT

1. Adjacency List Review
2. Search
3. Depth-first search
4. Breadth-first search
5. Edge-weighted graphs
6. The shortest path
7. Dijkstra’s algorithm

ADJACENCY LIST (REVIEW)



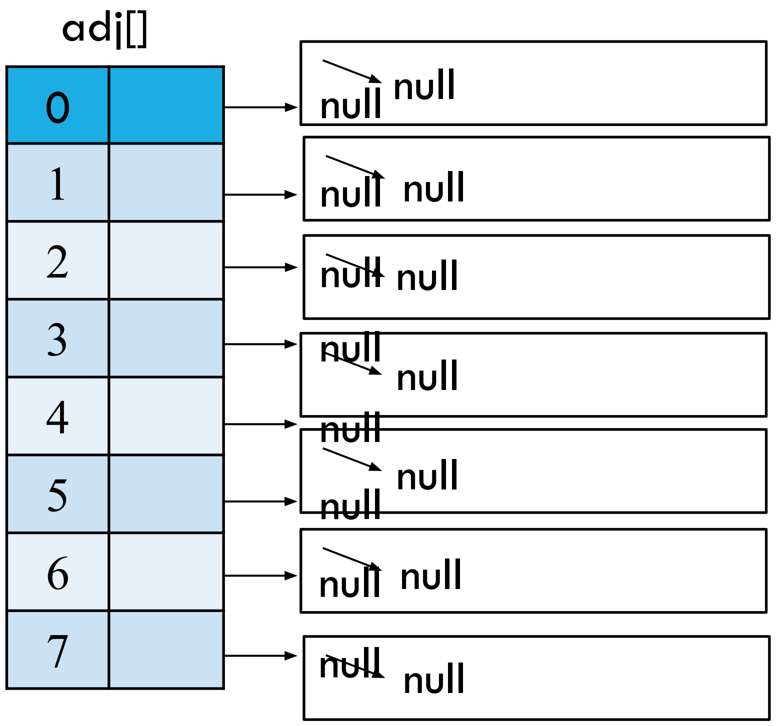
AdjList[0] = (1, 2, 3, 4)
AdjList[1] = (0, 5, 6)
AdjList[2] = (0, 7)
AdjList[3] = (0)
AdjList[4] = (0)
AdjList[5] = (1)
AdjList[6] = (1)
AdjList[7] = (2)



ADJACENCY LIST (REVIEW)

```java
public class MyGraph{

    private int V, E;
    LinkedList<Integer> adj[];

    MyGraph(int nodes){
        V = nodes;
        this.adj = new LinkedList[nodes];

        for(int v = 0; v < V; v++) {
            adj[v] = new LinkedList<>();
        }
    }

    public void addEdge(int u, int v) {
        adj[u].add(v);
        adj[v].add(u);
    }

    public void printGraph() {

        for(int v = 0; v < V; v++) {

            System.out.print("AdjList[" + v + "]: ");

            for(int w = 0; w < adj[v].size(); w++) {
                System.out.print(adj[v].get(w) + " ");
            }
            System.out.println();
        }
    }
```

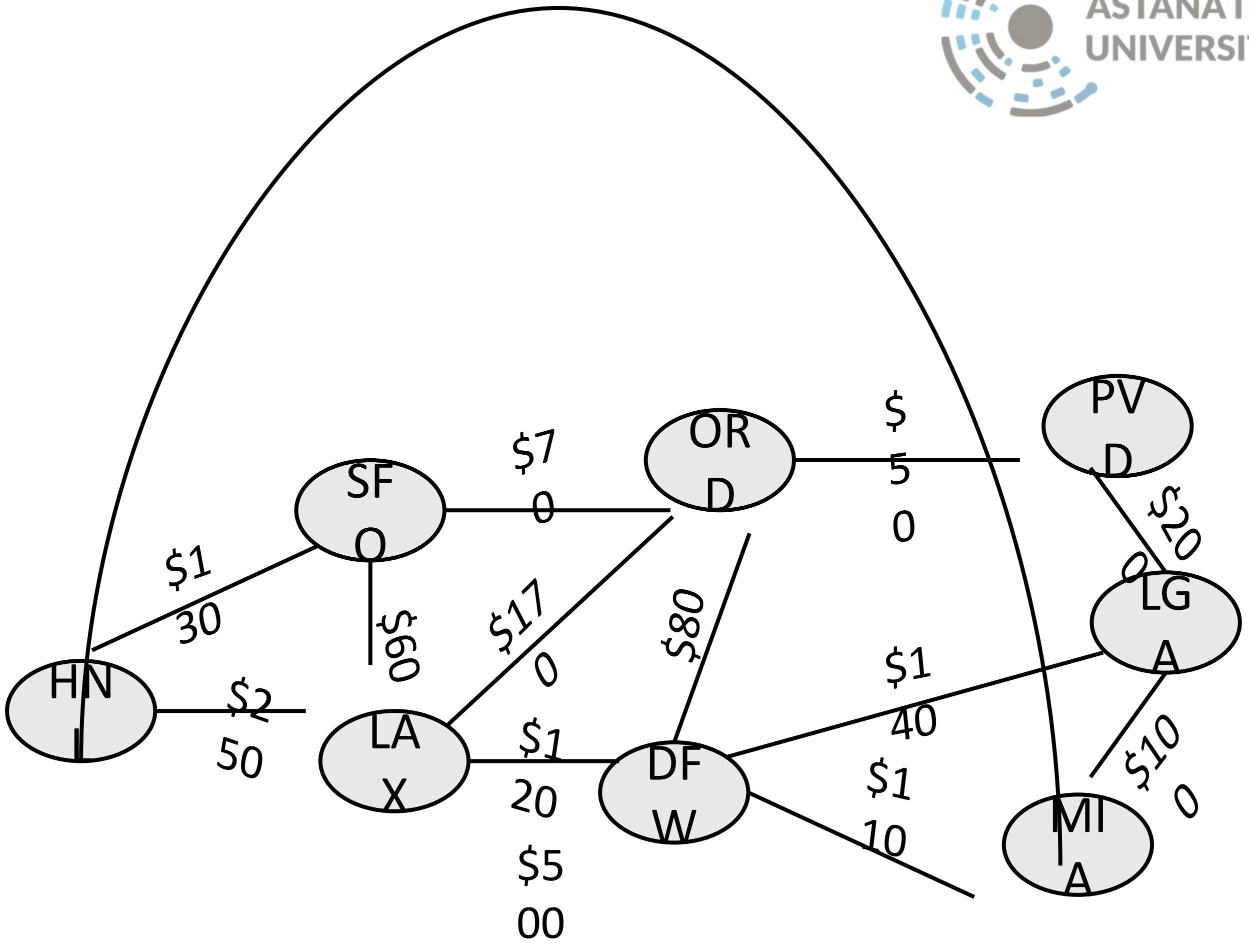
```java
    public static void main(String[] args) {

        MyGraph g = new MyGraph(8);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(0, 3);
        g.addEdge(0, 4);
        g.addEdge(1, 5);
        g.addEdge(1, 6);
        g.addEdge(2, 7);

        g.printGraph();

    }
}
```



SEARCH

- Why do we need to search graphs?

1. Path problems: e.g. What is the **shortest** path from node A to node B?
2. Connectivity problems: e.g, If we can **reach** from node A to node B?
3. Spanning tree problems: e.g. Find the minimal spanning tree



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

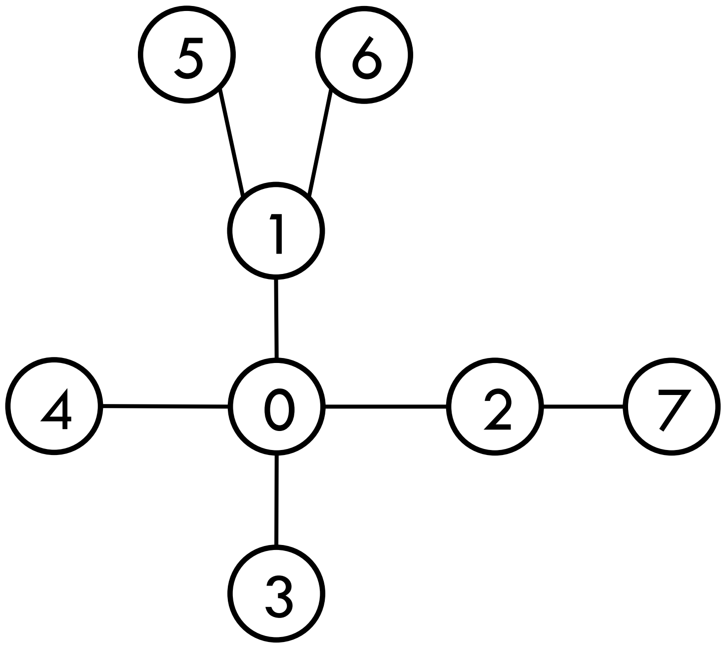
SEARCH

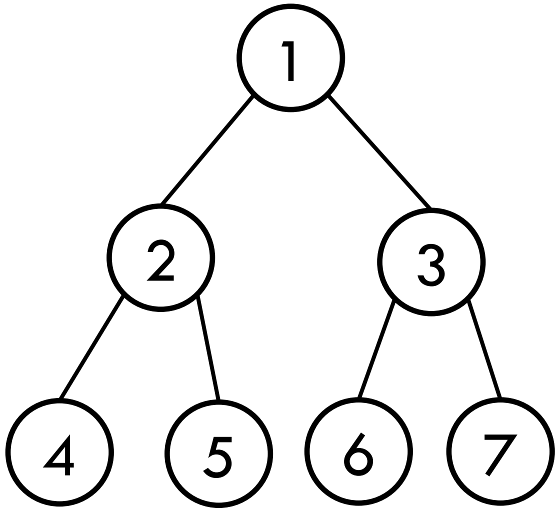
- There are two standard graph traversal techniques:

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)

- In both DFS and BFS, the nodes of the undirected graph are visited in a **systematic** manner so that every node is visited exactly one.

DEPTH-FIRST SEARCH



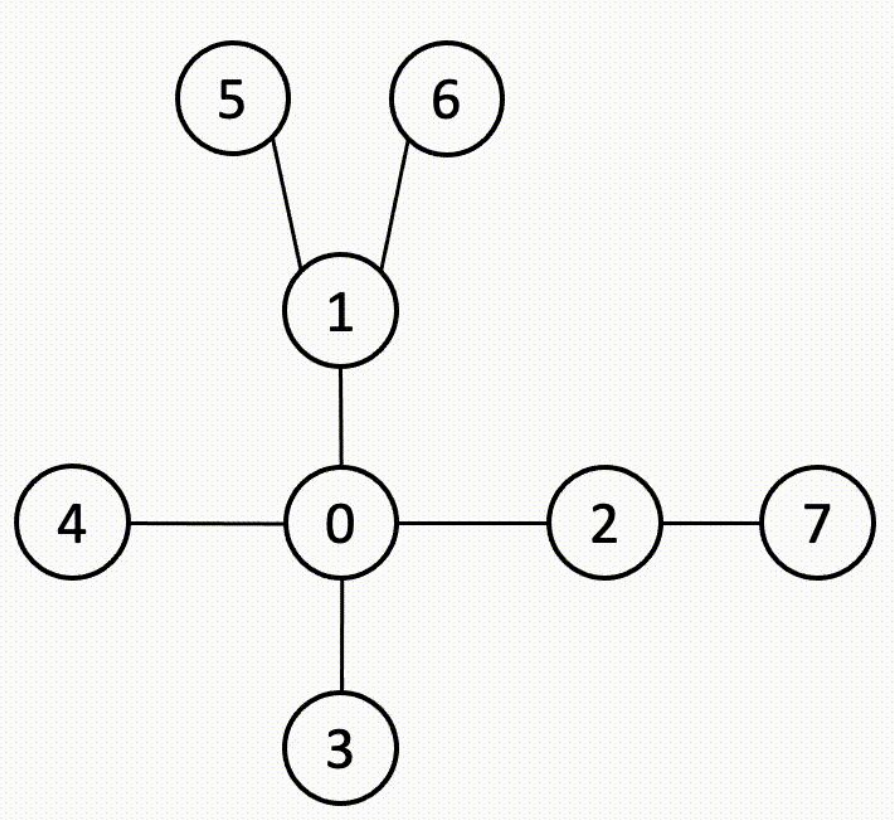


DEPTH-FIRST SEARCH

- DFS follows the following rules:

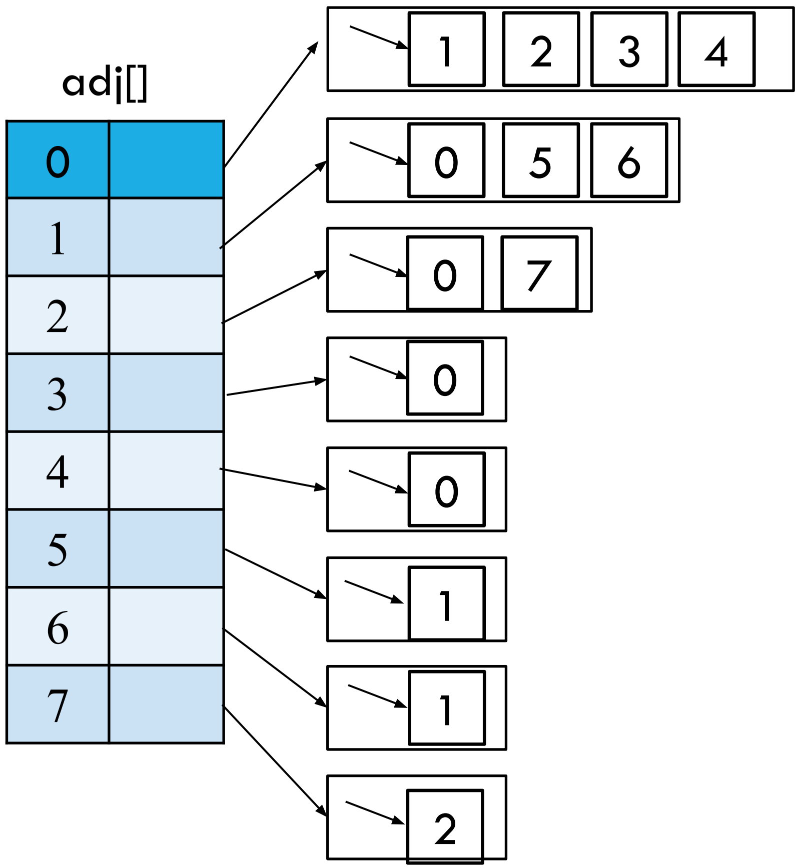
1. Select an unvisited node x, visit it, and treat as the current node
2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
4. Repeat steps 3 and 4 until no more nodes can be visited.
5. If there are still unvisited nodes, repeat from step 1.

- A **stack data structure** is used to support backtracking when implementing the DFS

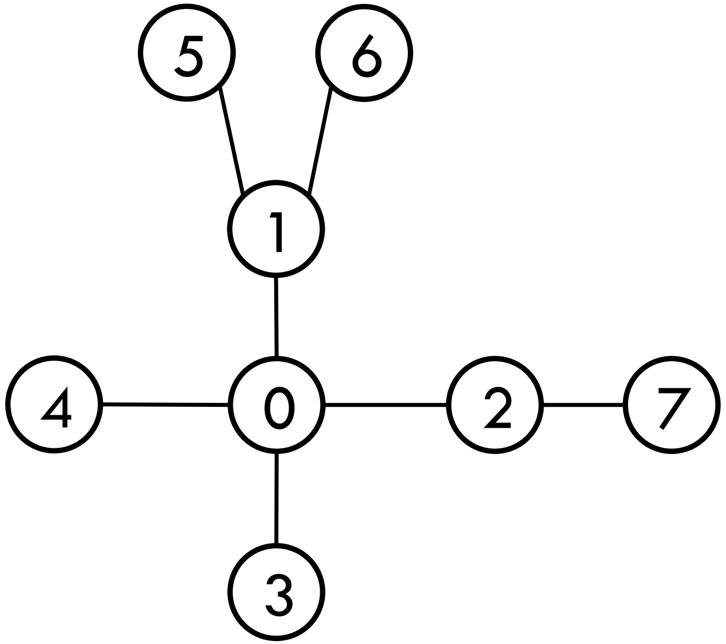


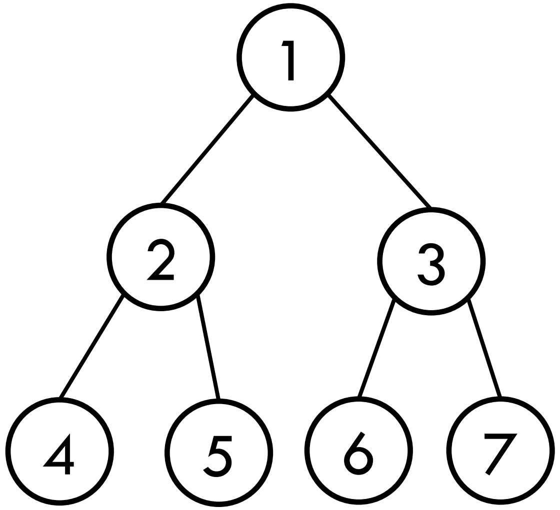
DEPTH-FIRST SEARCH

```
public void dfs() {
    boolean[] visited = new boolean[V];
    for(int v = 0; v < V; v++) {
        if(!visited[v]) {
            visitVertex(v, visited);
        }
    }
}

public void visitVertex(int v, boolean[] visited) {
    visited[v] = true;
    System.out.print(v + " ");
    for(int w = 0; w < adj[v].size(); w++) {
        if(!visited[adj[v].get(w)]) {
            visitVertex(adj[v].get(w), visited);
        }
    }
}
```



BREADTH-FIRST SEARCH



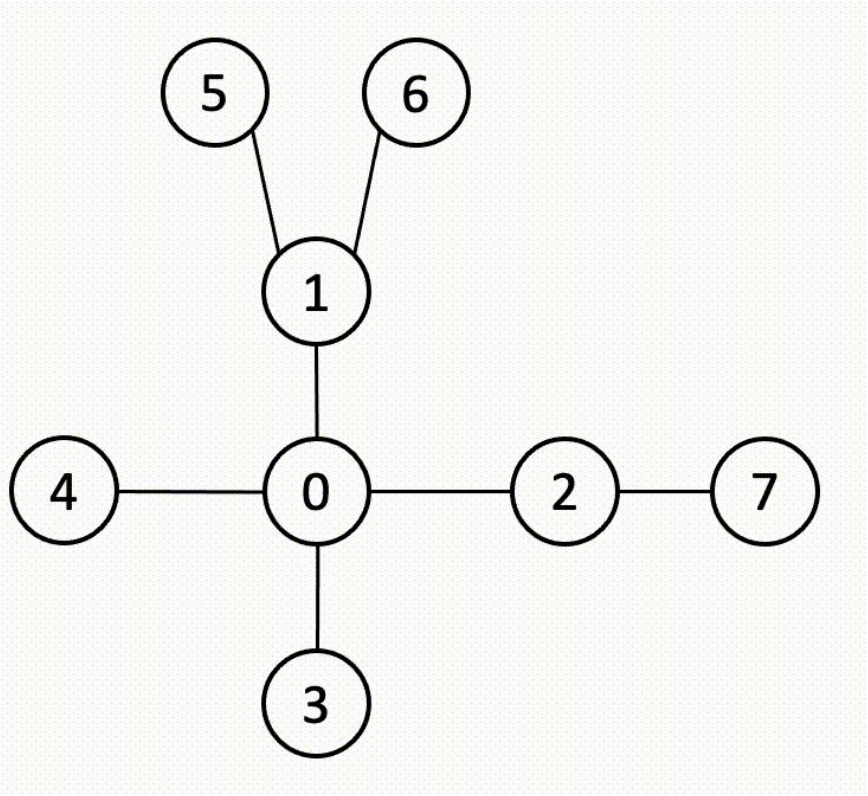


BREADTH-FIRST SEARCH

- BFS follows the following rules:

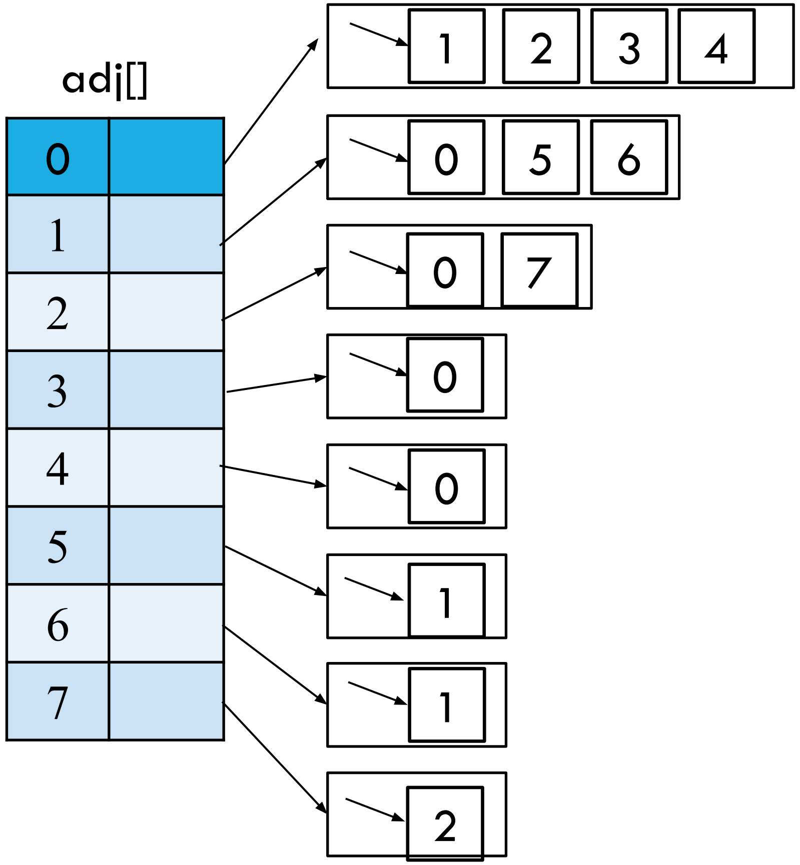
1. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
2. From each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z. The newly visited nodes from this level form a new level that becomes the next current level.
3. Repeat step 2 until no more nodes can be visited.
4. If there are still unvisited nodes, repeat from Step 1.

- A **queue data structure** is used when implementing the BFS



BREADTH-FIRST SEARCH

```java
public void bfs(int start) {

    boolean[] visited = new boolean[V];
    visited[start] = true;
    Queue<Integer> q = new LinkedList<>();
    q.add(start);

    while(!q.isEmpty()) {
        int u = q.poll();
        System.out.print(u + " ");
        for(int w = 0; w < adj[u].size(); w++) {
            if(!visited[adj[u].get(w)]) {
                visited[adj[u].get(w)] = true;
                q.add(adj[u].get(w));
            }
        }
    }
}
```



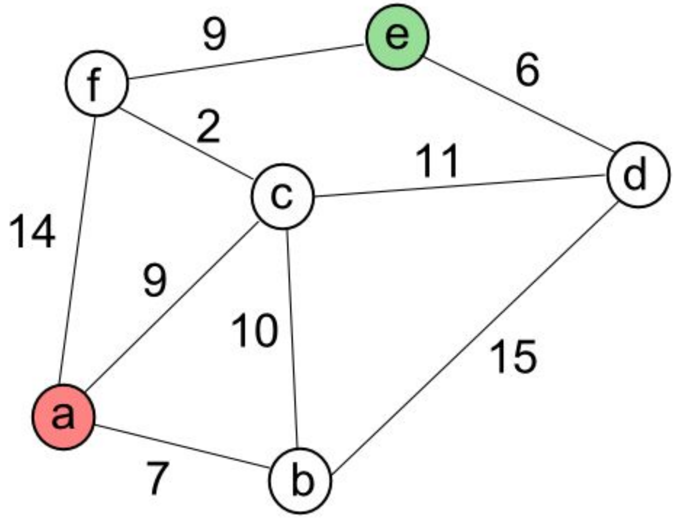
EDGE-WEIGHTED GRAPHS

An edge-weighted graph is a graph model where we associate weights or costs with each edge

Example Applications: **Route** for *Yandex taxi* where the **weight** might represent

- **Distance**
- Approximate **time**
- Average **speed**
- Or all the above for that section of road

Weight calculation is entirely up to the designer



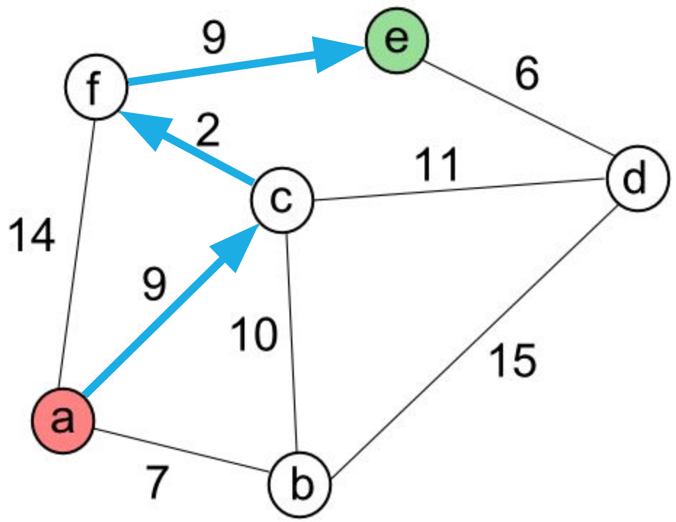

```java
public class Edge<Vertex> {
    private Vertex source;
    private Vertex dest;
    private Double weight;

    public Edge(Vertex source,
                Vertex dest,
                Double weight) {
        this.source = source;
        this.dest = dest;
        this.weight = weight;
    }

    //getters & setters
}
```

THE SHORTEST PATH

Find the lowest-cost way to get from one vertex to another

A **path weight** is the sum of the weights of that path's edges

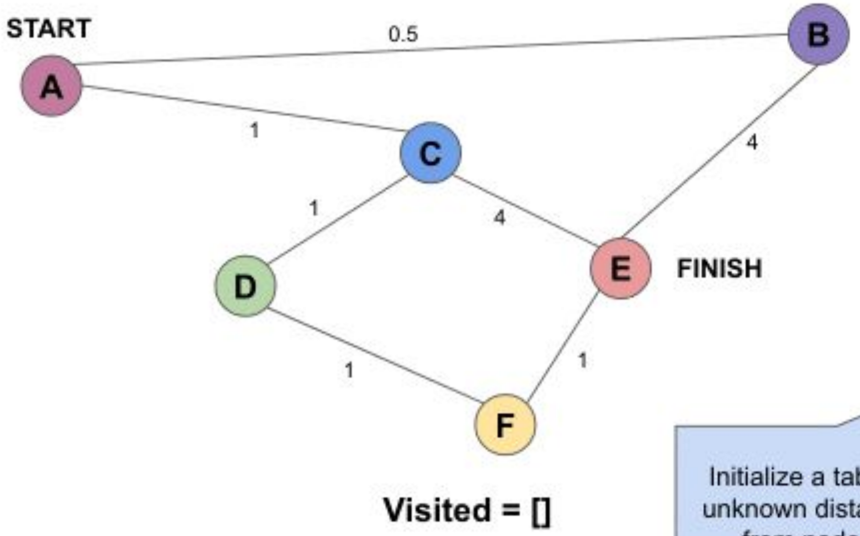
The **shortest path** from vertex **a** to vertex **e** in an edge-weighted digraph is a directed path from **a** to **e** with the property that no other such path has a lower weight



DIJKSTRA’S ALGORITHM

Dijkstra’s algorithm solves the single-source shortest-paths problem in edge-weighted digraphs with nonnegative weights

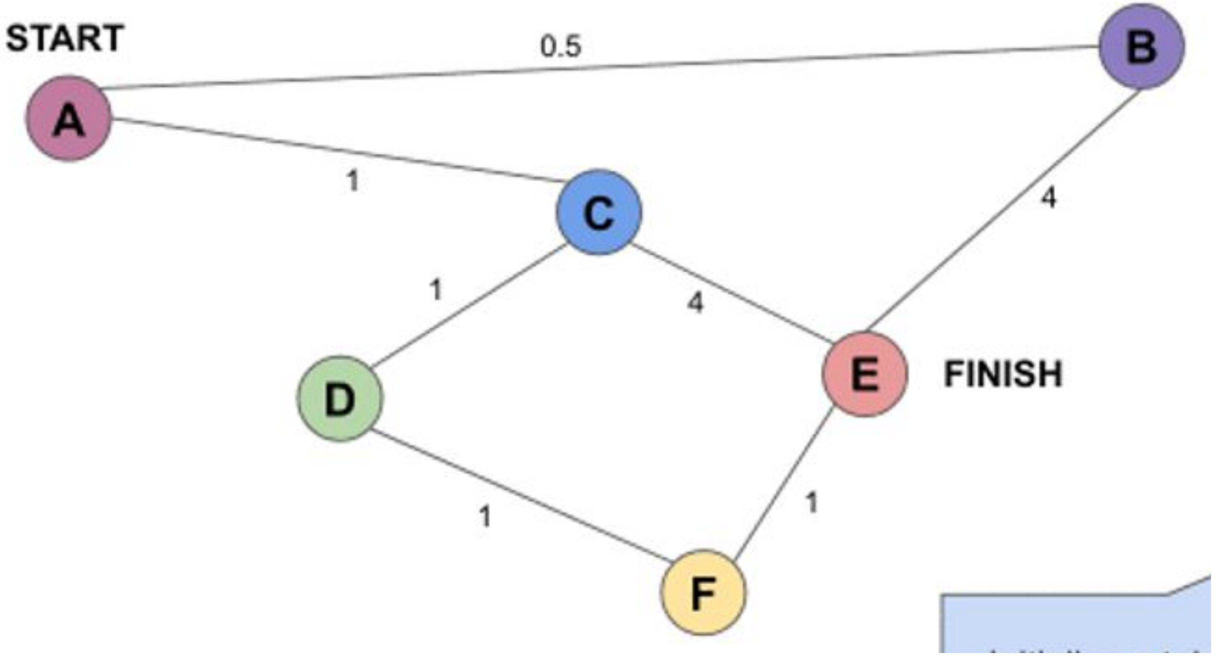
The method keeps track of the current shortest distance between each node and the source node and updates these values whenever a shorter path is discovered



Initialize a table of unknown distances from node A

	Distance	Last Node
A	0	
B	?	?
C	?	?
D	?	?
E	?	?
F	?	?

DIJKSTRA'S ALGORITHM



Visited vertex	B	C	D	E	F
	0.5.	1	inf	inf.	inf
B	0.5.	1	inf	4.5	inf

Choose the shortest path, which is B, and visit it
Change the distance to other vertices, if there is found a shorter path

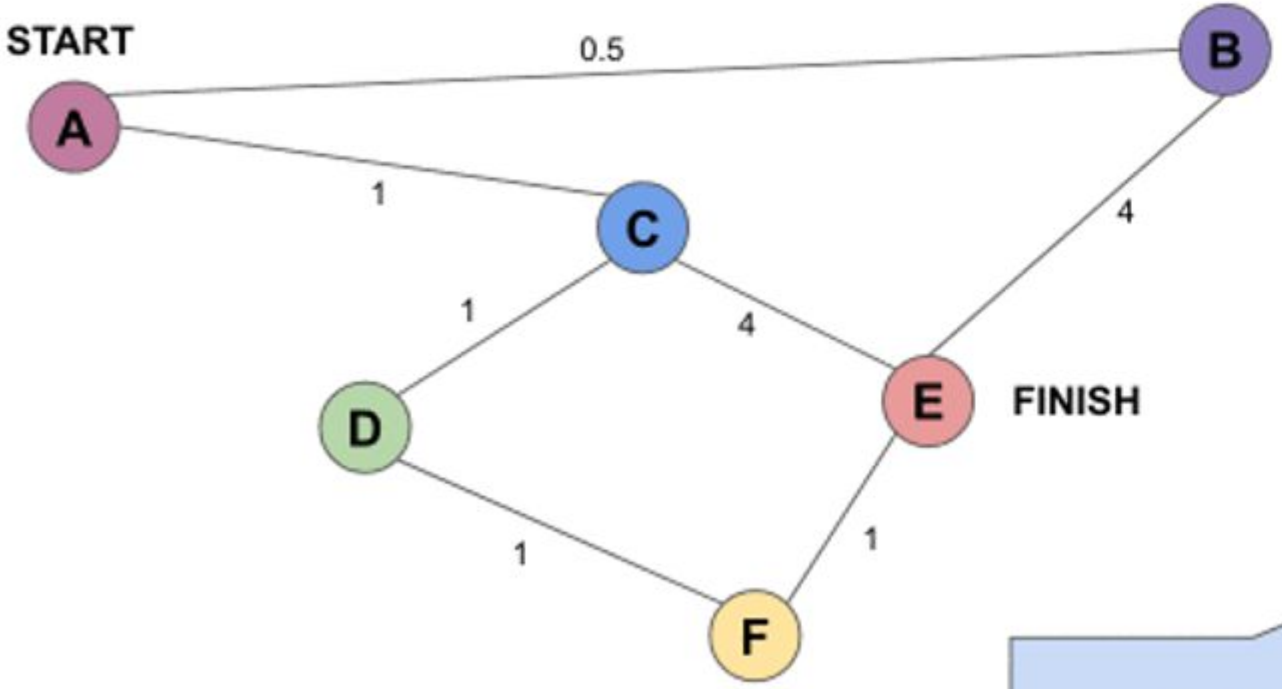
DIJKSTRA'S ALGORITHM

When the algorithm finds the shortest path between two nodes, that node is tagged as "visited" and added to the path

The method is repeated until the path contains all the nodes in the graph

Only graphs with **positive weights** can be used by Dijkstra's Algorithm. This is because the weights of the edges must be added

DIJKSTRA'S ALGORITHM

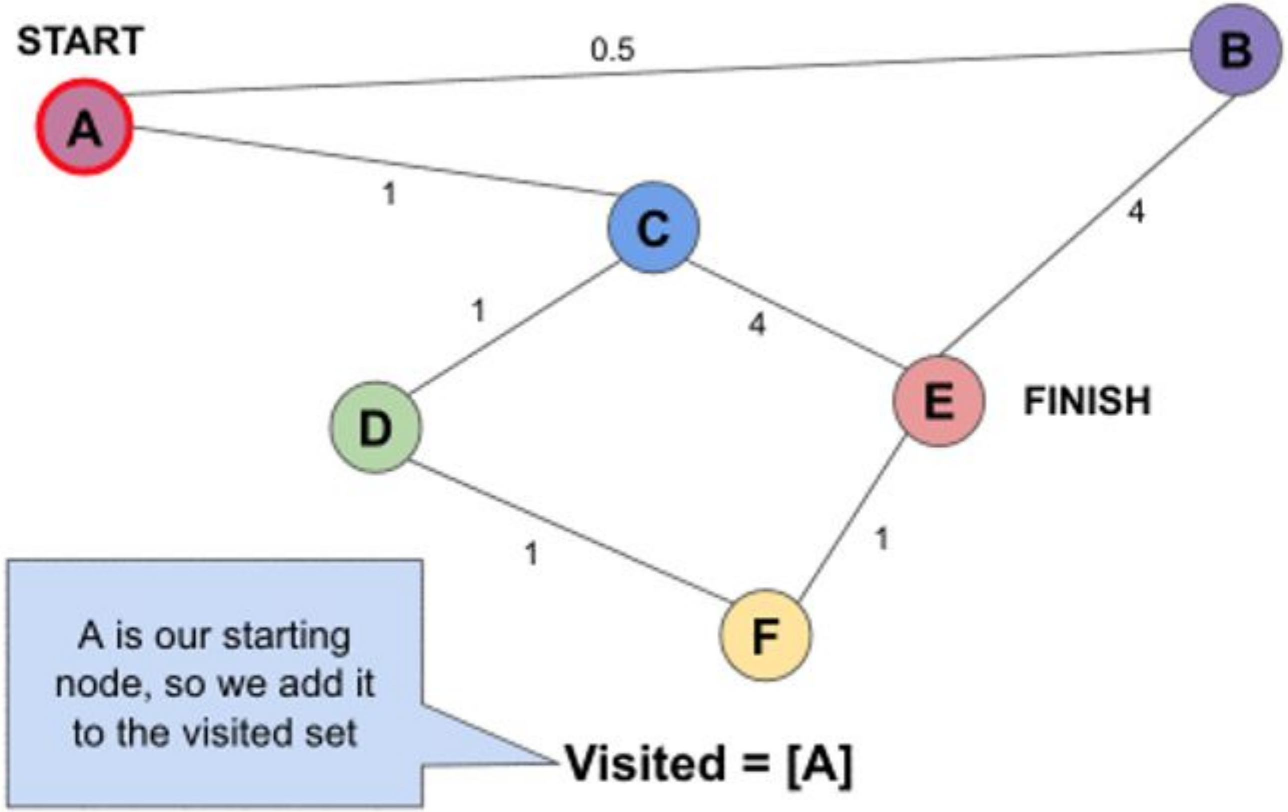


Visited = []

Initialize a table of unknown distances from node A

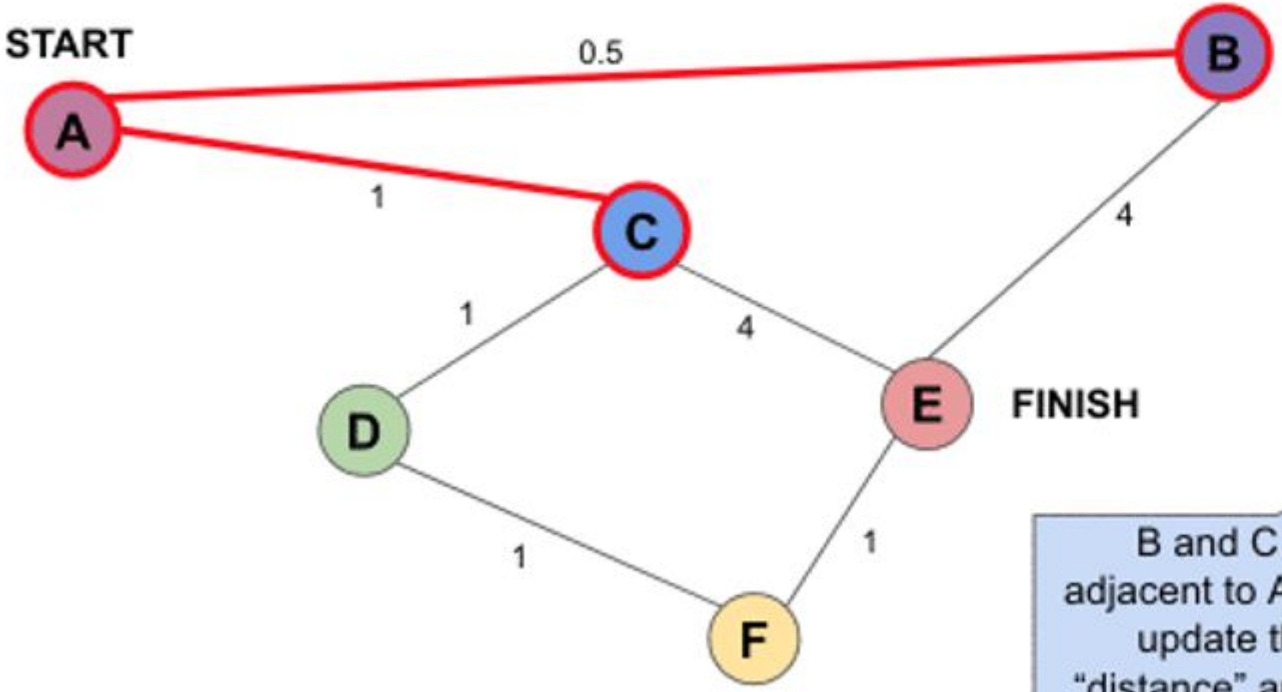
	Distance	Last Node
A	0	
B	?	?
C	?	?
D	?	?
E	?	?
F	?	?

DIJKSTRA'S ALGORITHM



	Distance	Last Node
A	0	
B	?	?
C	?	?
D	?	?
E	?	?
F	?	?

DIJKSTRA'S ALGORITHM

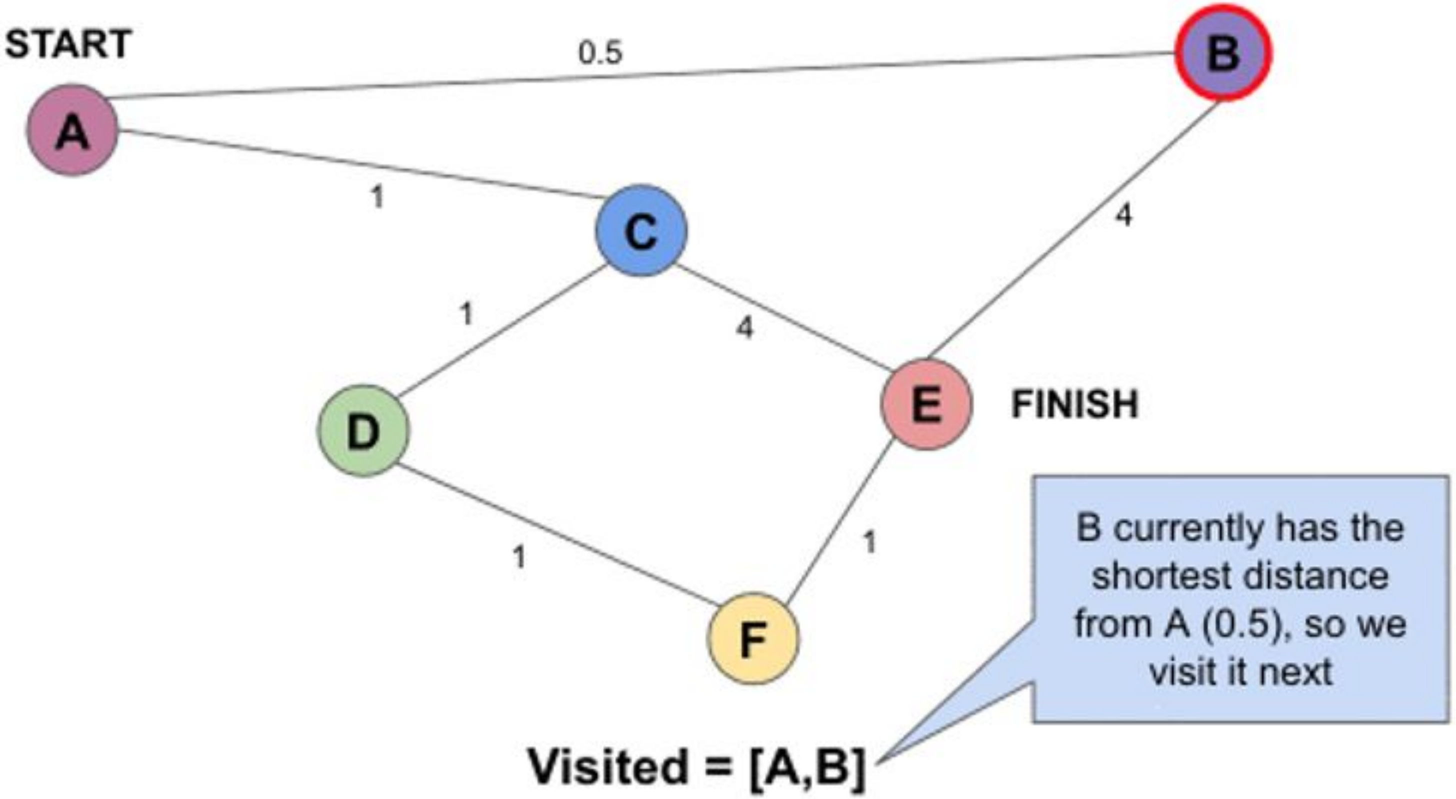


Visited = [A]

B and C are adjacent to A, so we update their "distance" and "last node" values

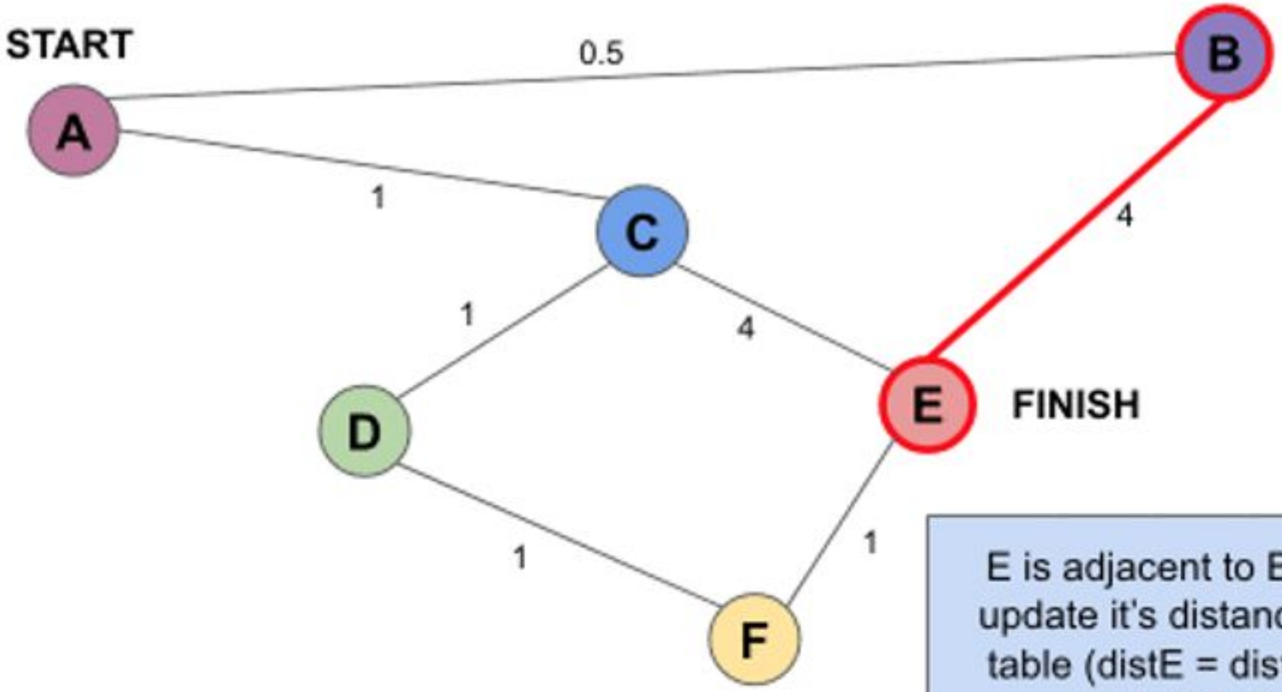
	Distance	Last Node
A	0	
B	0.5	A
C	1	A
D	?	?
E	?	?
F	?	?

DIJKSTRA'S ALGORITHM



	Distance	Last Node
A	0	
B	0.5	A
C	1	A
D	?	?
E	?	?
F	?	?

DIJKSTRA’S ALGORITHM

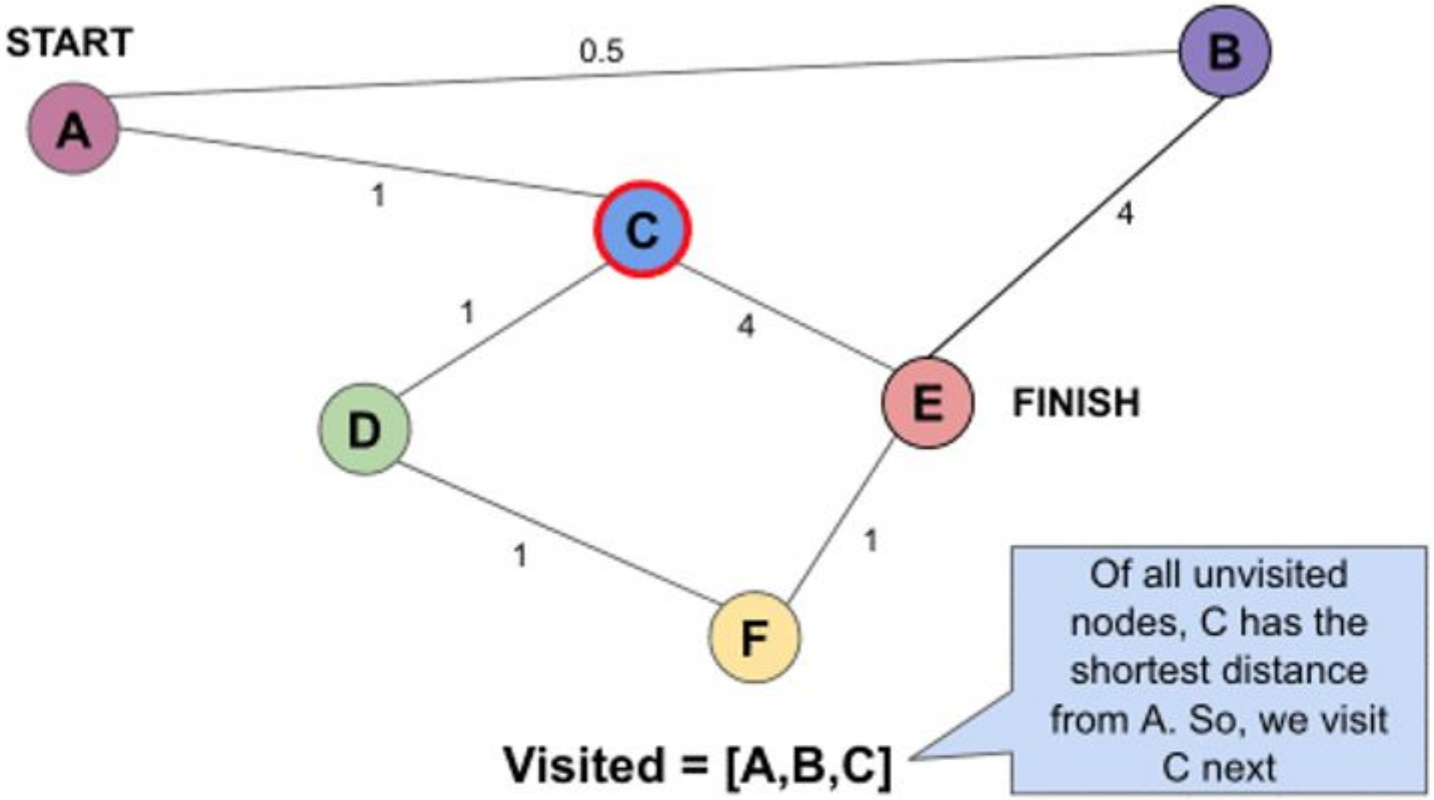


Visited = [A,B]

E is adjacent to B, so we update it's distance in our table (distE = distAToB + distBToE)

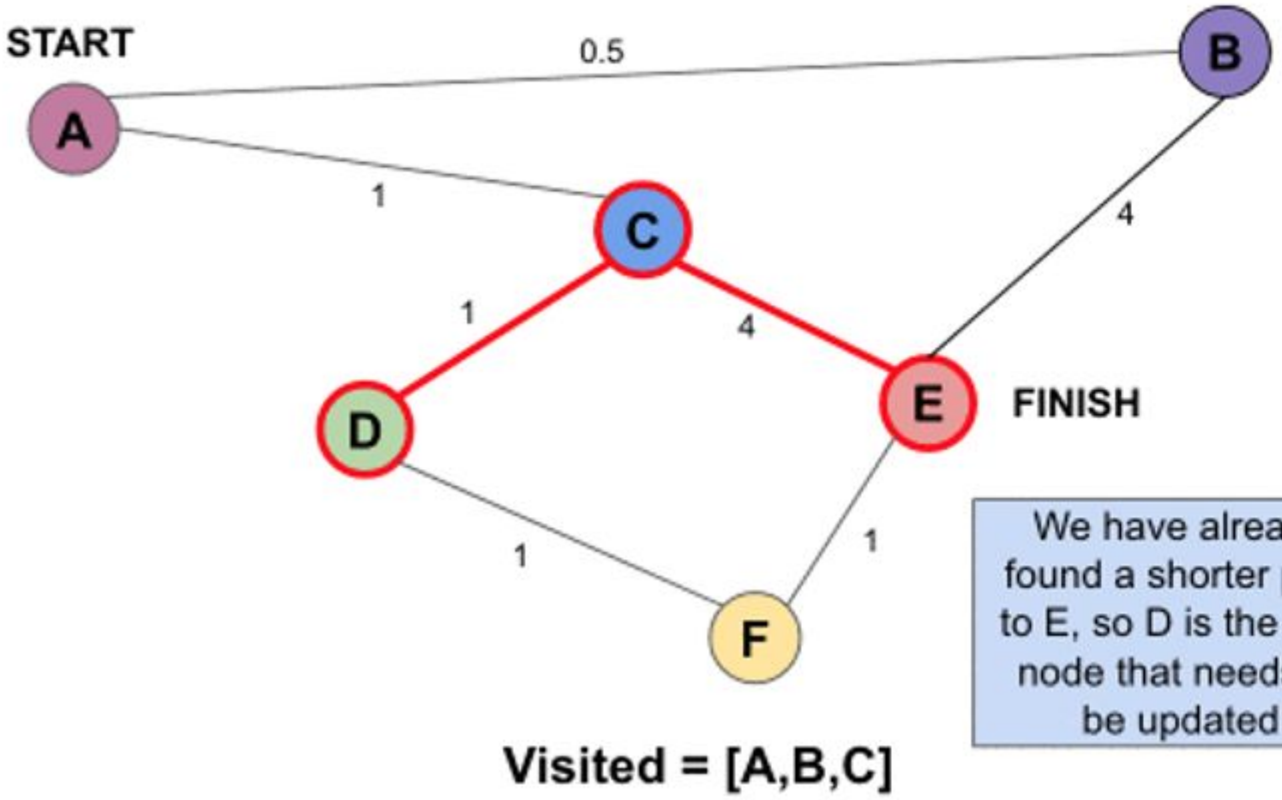
	Distance	Last Node
A	0	
B	0.5	A
C	1	A
D	?	?
E	**4.5**	**B**
F	?	?

DIJKSTRA’S ALGORITHM



	Distance	Last Node
A	0	
B	0.5	A
C	1	A
D	?	?
E	4.5	B
F	?	?

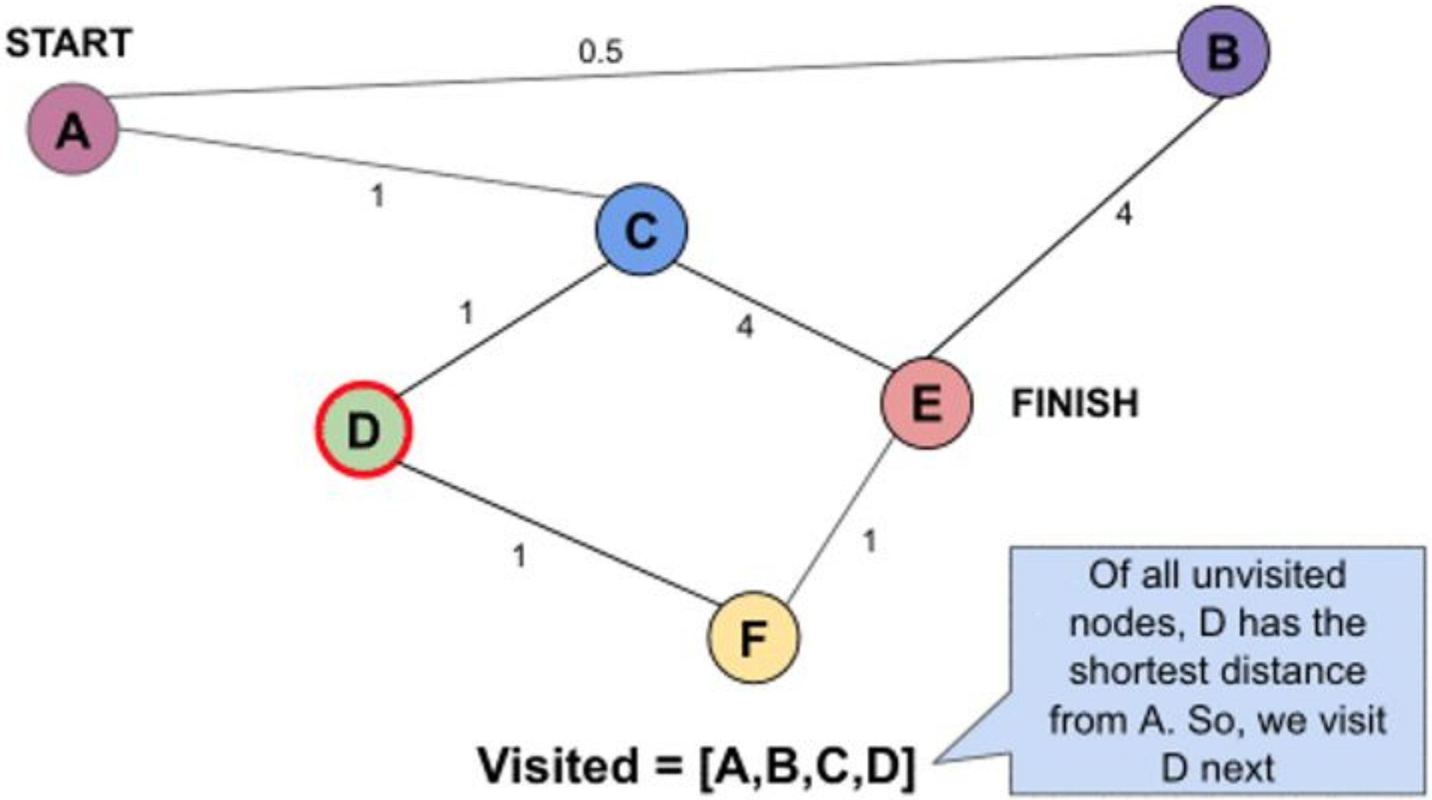
DIJKSTRA'S ALGORITHM



We have already found a shorter path to E, so D is the only node that needs to be updated

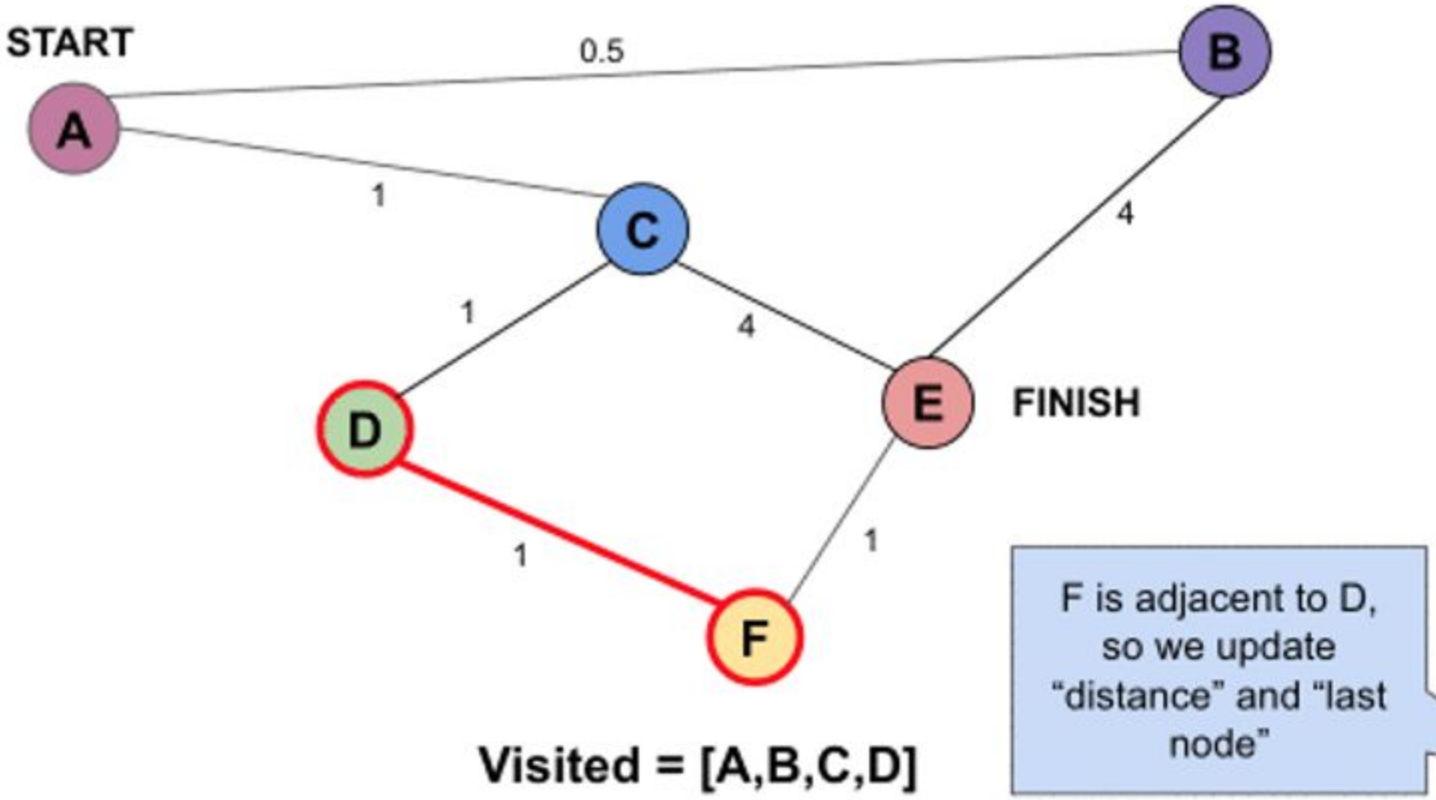
	Distance	Last Node
A	0	
B	0.5	A
C	1	A
D	**2**	**C**
E	4.5	B
F	?	?

DIJKSTRA'S ALGORITHM



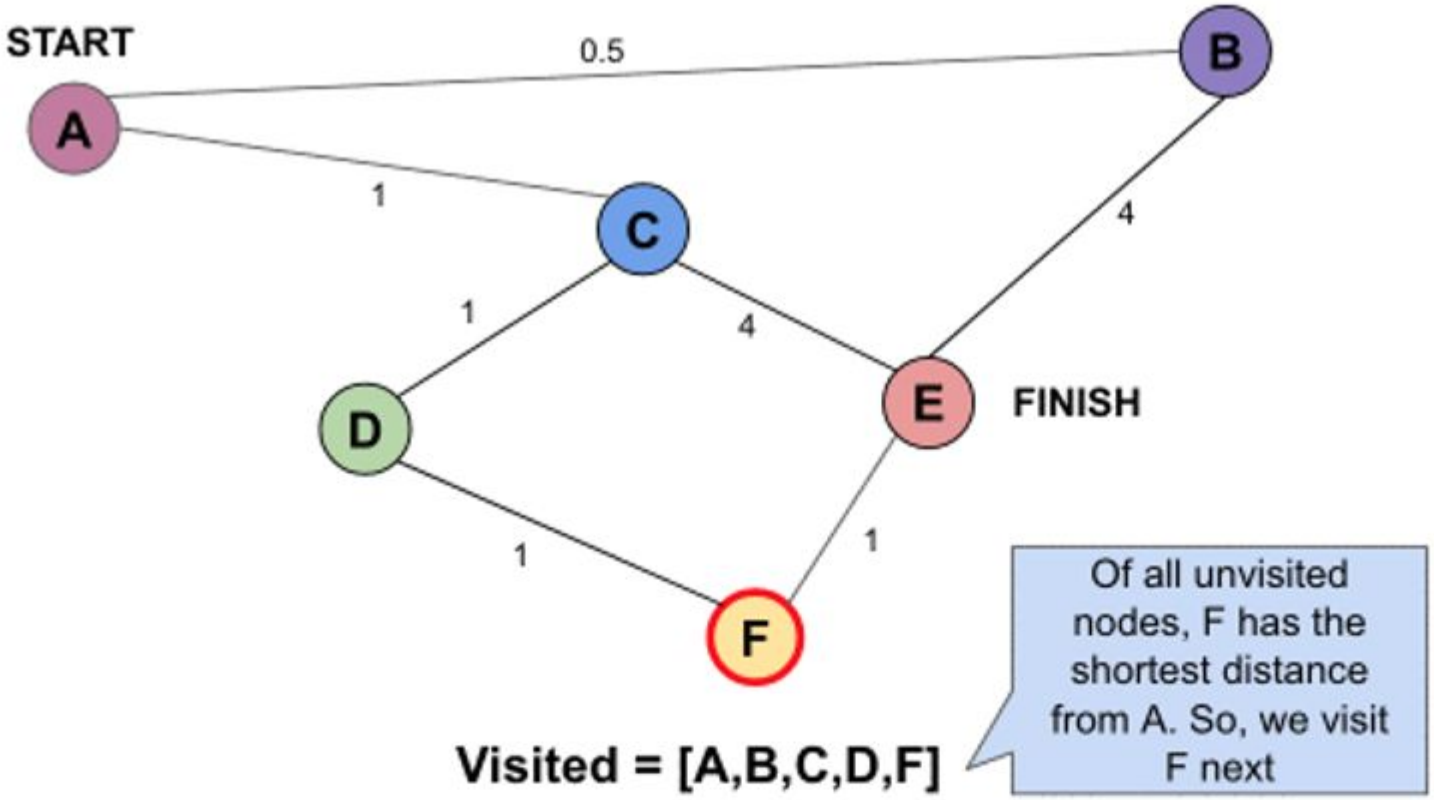
	Distance	Last Node
A	0	
B	0.5	A
C	1	A
D	2	C
E	4.5	B
F	3	D

DIJKSTRA'S ALGORITHM



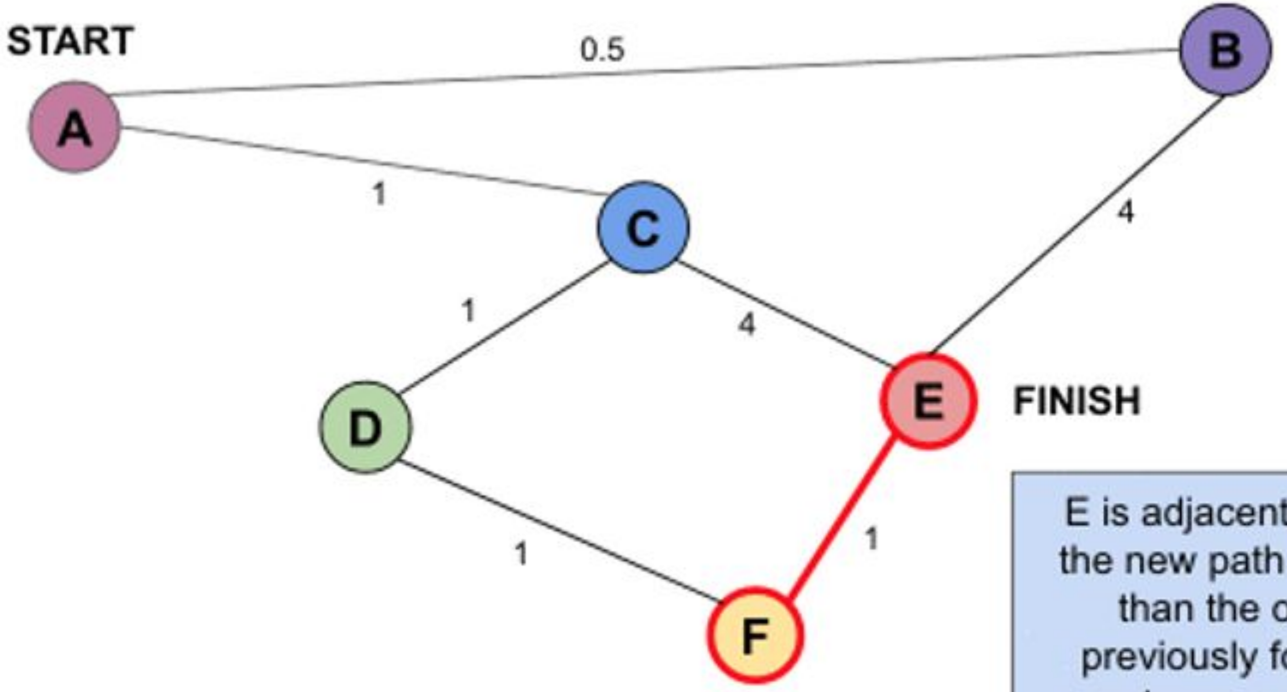
	Distance	Last Node
A	0	
B	0.5	A
C	1	A
D	2	C
E	4.5	B
F	**3**	**D**

DIJKSTRA'S ALGORITHM



	Distance	Last Node
A	0	
B	0.5	A
C	1	A
D	2	C
E	4.5	B
F	**3**	**D**

DIJKSTRA’S ALGORITHM



Visited = [A,B,C,D,F]

E is adjacent to F, and the new path is shorter than the one we previously found! As such, we update the distance

	Distance	Last Node
A	0	
B	0.5	A
C	1	A
D	2	C
E	**4**	**F**
F	3	D

LITERATURE

Algorithms, 4th Edition, by Robert Sedgewick and Kevin Wayne, Addison-Wesley

- Chapter 4

Grokking Algorithms, by Aditya Y. Bhargava, Manning

- Chapters 6-8

GOOD LUCK!