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

Askar Khaimuldin
askar.khaimuldin@astanait.edu.kz

ASTANA IT UNIVERSITY

CONTENT

SEARCH

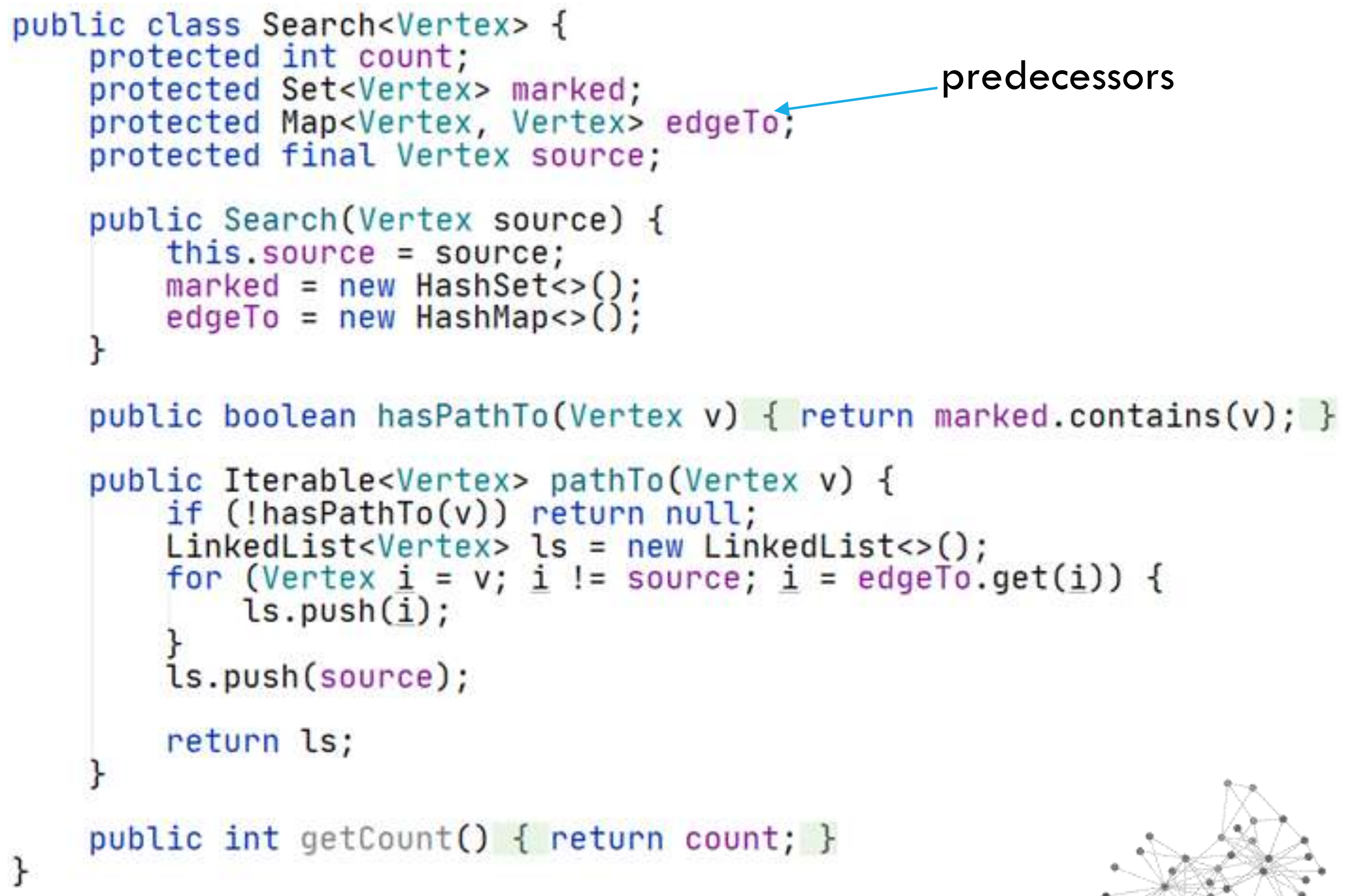
Question type 1: Is there a **path** from node A to node B?

Question type 2: What is the **shortest** path from node A to node B?

Return the path from vertex X_0 to vertex X_n (u to v) in form of collection of vertices

Traversing Types:

- Depth-first search
- Breadth-first search



```java
public class Search<Vertex> {
    protected int count;
    protected Set<Vertex> marked;
    protected Map<Vertex, Vertex> edgeTo;
    protected final Vertex source;

    public Search(Vertex source) {
        this.source = source;
        marked = new HashSet<>();
        edgeTo = new HashMap<>();
    }

    public boolean hasPathTo(Vertex v) { return marked.contains(v); }

    public Iterable<Vertex> pathTo(Vertex v) {
        if (!hasPathTo(v)) return null;
        LinkedList<Vertex> ls = new LinkedList<>();
        for (Vertex i = v; i != source; i = edgeTo.get(i)) {
            ls.push(i);
        }
        ls.push(source);

        return ls;
    }

    public int getCount() { return count; }
}
```

predecessors

DEPTH-FIRST SEARCH

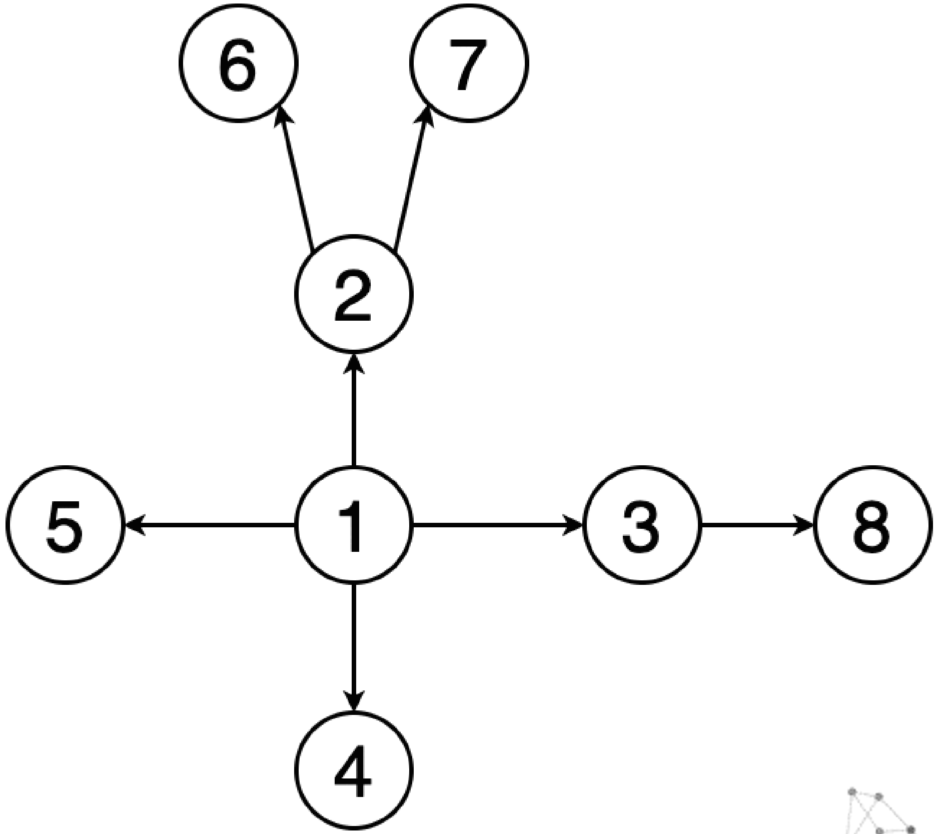
In depth-first search (**DFS**) we start at a specific vertex and explore as far as feasible along each branch before retracing our steps (backtracking)

Cycles may exist in graphs, unlike trees (a path where the first and last vertices are the same)

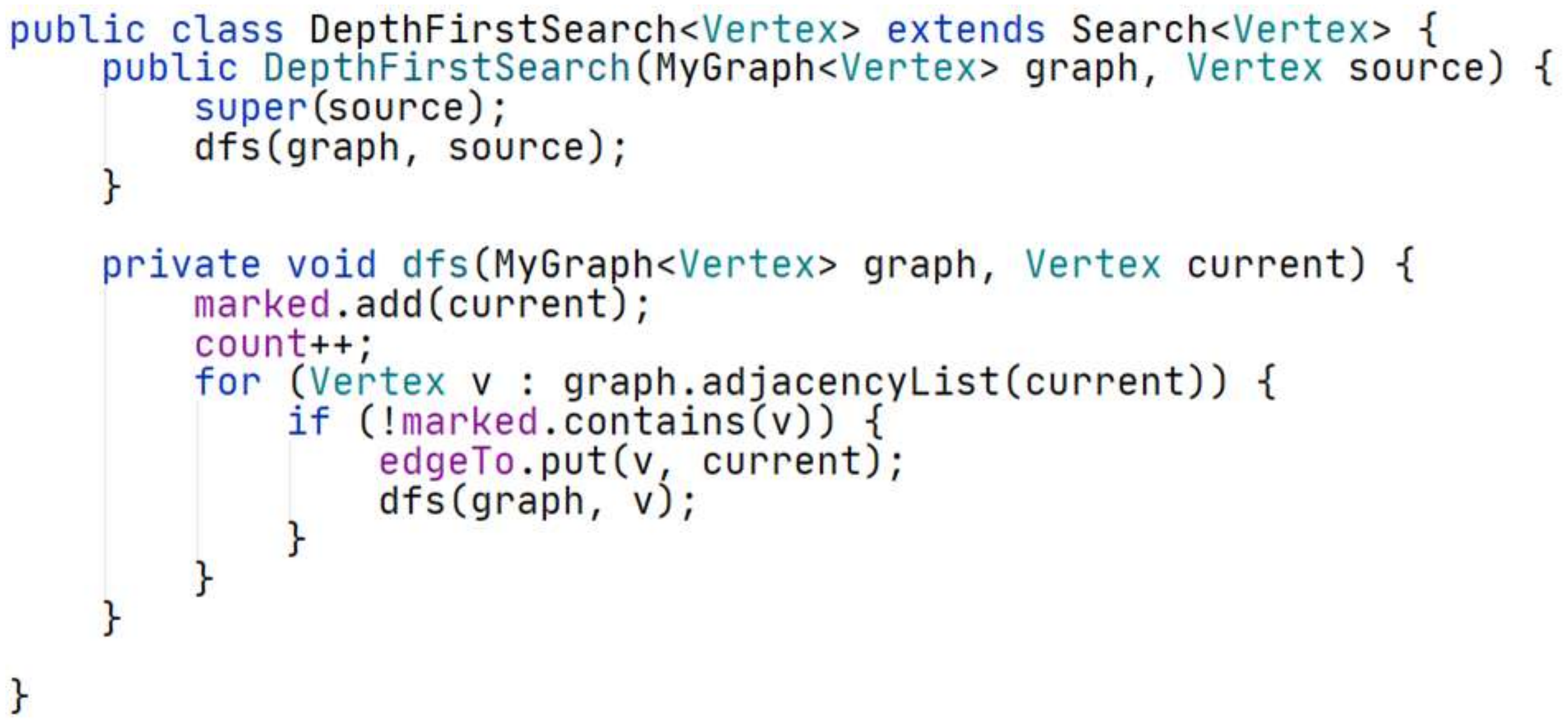
Therefore, we must keep track of which vertices have been visited

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

Example Applications: Puzzles and Mazes



DEPTH-FIRST SEARCH



```java
public class DepthFirstSearch<Vertex> extends Search<Vertex> {
    public DepthFirstSearch(MyGraph<Vertex> graph, Vertex source) {
        super(source);
        dfs(graph, source);
    }

    private void dfs(MyGraph<Vertex> graph, Vertex current) {
        marked.add(current);
        count++;
        for (Vertex v : graph.adjacencyList(current)) {
            if (!marked.contains(v)) {
                edgeTo.put(v, current);
                dfs(graph, v);
            }
        }
    }

}
```

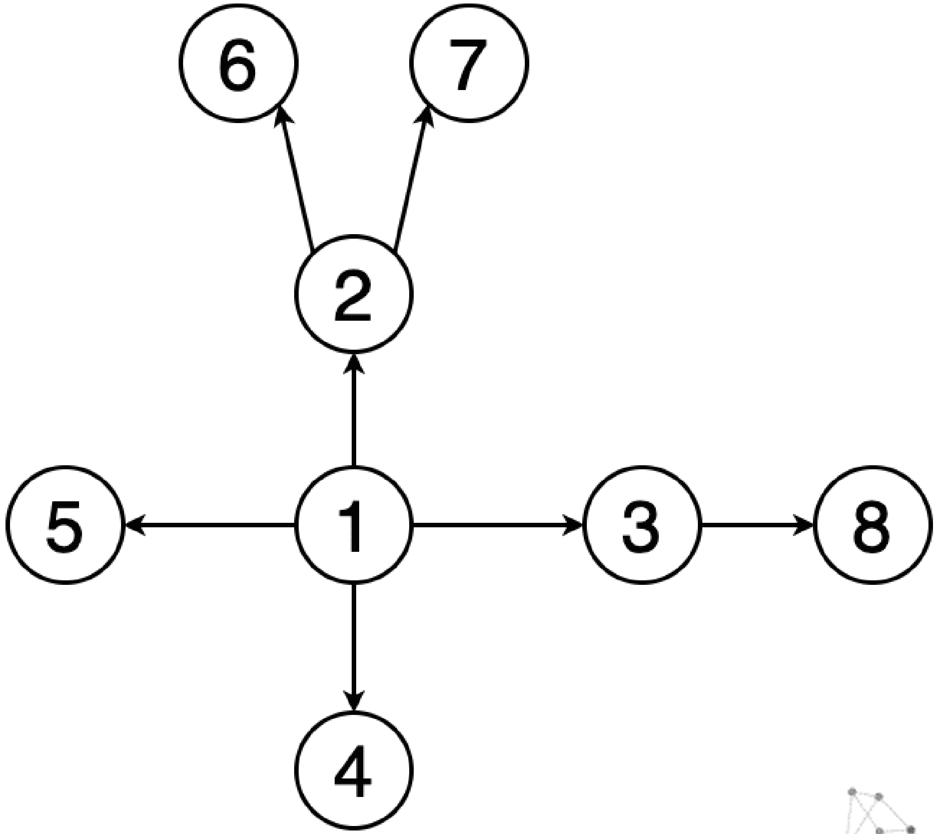
BREADTH-FIRST SEARCH

In breadth-first search (**BFS**), before going on to the vertices in the next level, we start at a certain vertex and study all of its neighbors at the current depth

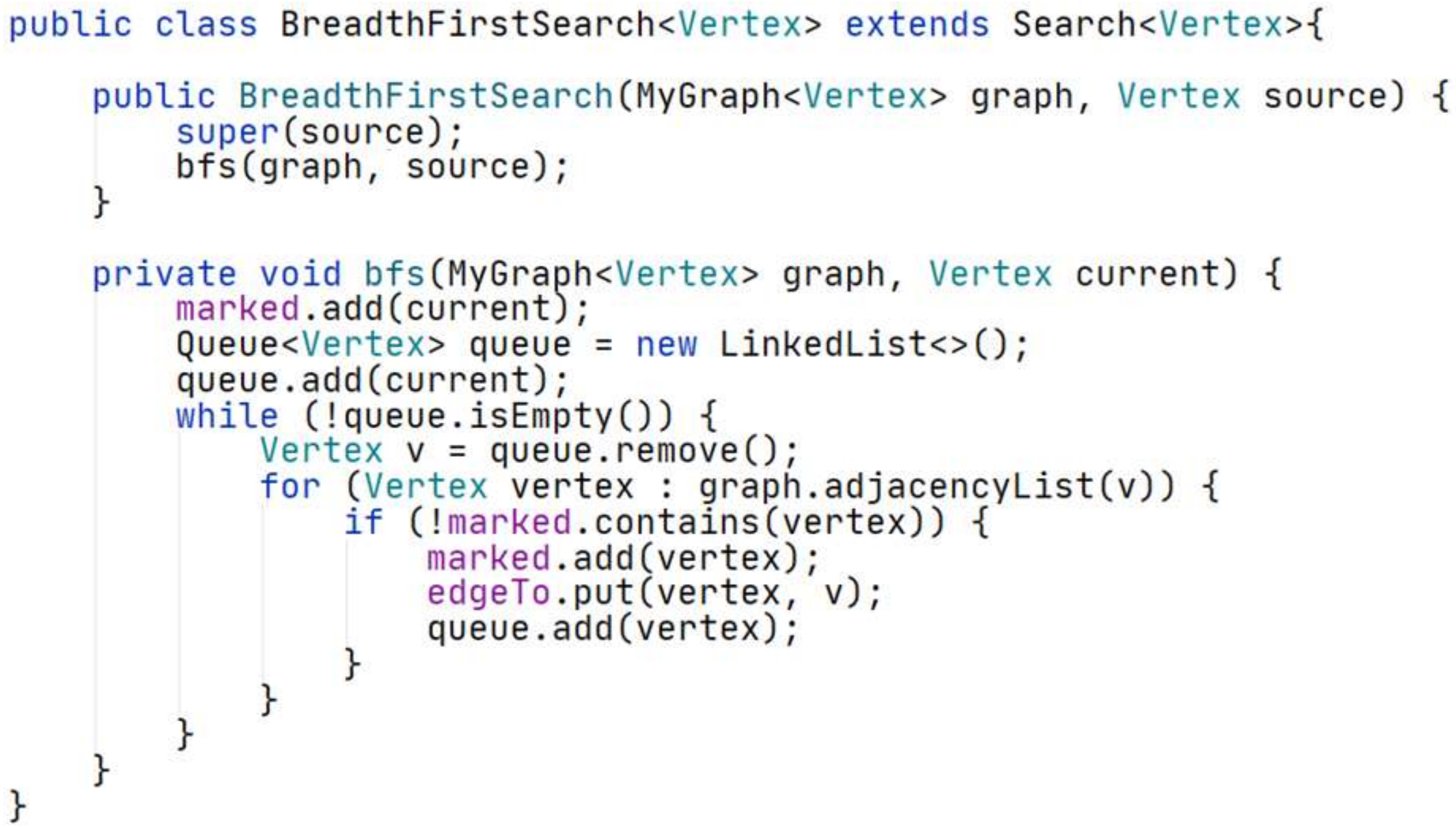
We must also keep track of the visited vertices in BFS

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

Example Applications: Maps and Mazes

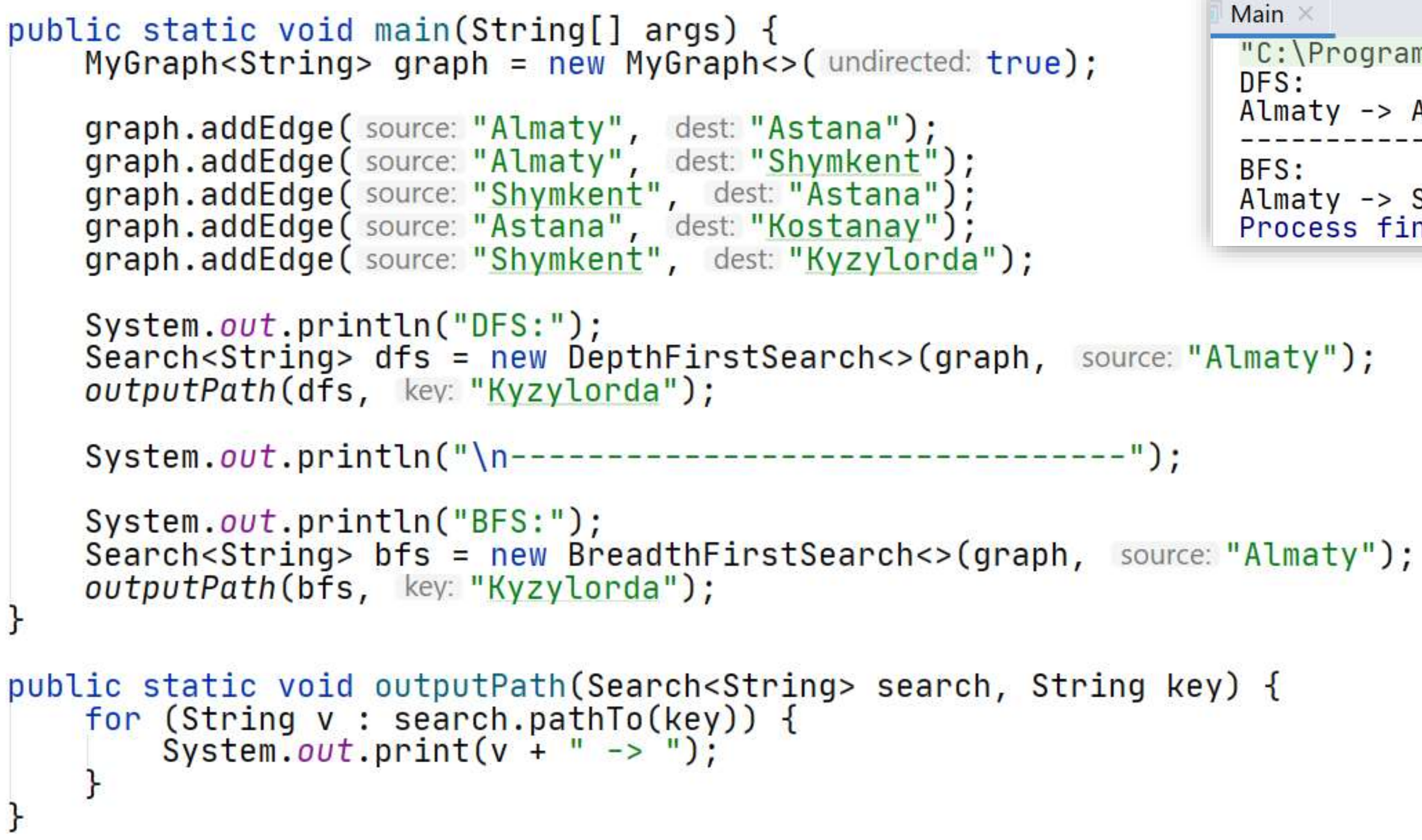


BREADTH-FIRST SEARCH

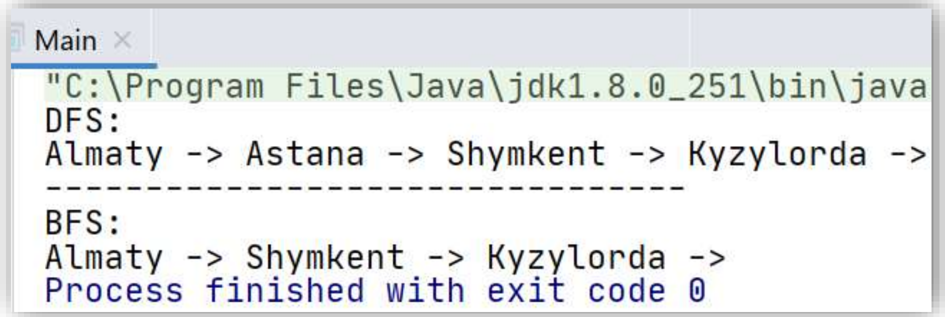


```java
public class BreadthFirstSearch<Vertex> extends Search<Vertex>{

    public BreadthFirstSearch(MyGraph<Vertex> graph, Vertex source) {
        super(source);
        bfs(graph, source);
    }

    private void bfs(MyGraph<Vertex> graph, Vertex current) {
        marked.add(current);
        Queue<Vertex> queue = new LinkedList<>();
        queue.add(current);
        while (!queue.isEmpty()) {
            Vertex v = queue.remove();
            for (Vertex vertex : graph.adjacencyList(v)) {
                if (!marked.contains(vertex)) {
                    marked.add(vertex);
                    edgeTo.put(vertex, v);
                    queue.add(vertex);
                }
            }
        }
    }
}
```

RESULTS



```java
public static void main(String[] args) {
    MyGraph<String> graph = new MyGraph<>( undirected: true);

    graph.addEdge( source: "Almaty",  dest: "Astana");
    graph.addEdge( source: "Almaty",  dest: "Shymkent");
    graph.addEdge( source: "Shymkent",  dest: "Astana");
    graph.addEdge( source: "Astana",  dest: "Kostanay");
    graph.addEdge( source: "Shymkent",  dest: "Kyzylorda");

    System.out.println("DFS:");
    Search<String> dfs = new DepthFirstSearch<>(graph,  source: "Almaty");
    outputPath(dfs,  key: "Kyzylorda");

    System.out.println("\n--------------------------------");

    System.out.println("BFS:");
    Search<String> bfs = new BreadthFirstSearch<>(graph,  source: "Almaty");
    outputPath(bfs,  key: "Kyzylorda");
}

public static void outputPath(Search<String> search, String key) {
    for (String v : search.pathTo(key)) {
        System.out.print(v + " -> ");
    }
}
```



Main ×

```
"C:\Program Files\Java\jdk1.8.0_251\bin\java
DFS:
Almaty -> Astana -> Shymkent -> Kyzylorda ->
--------------------------------
BFS:
Almaty -> Shymkent -> Kyzylorda ->
Process finished with exit code 0
```

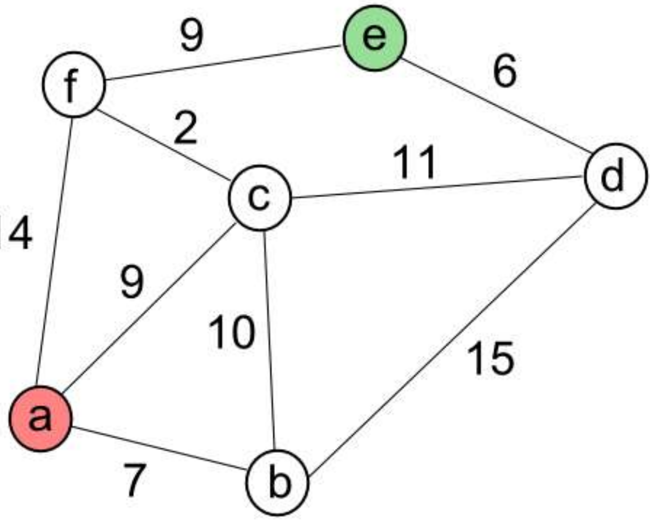
EDGE-WEIGHTED GRAPHS

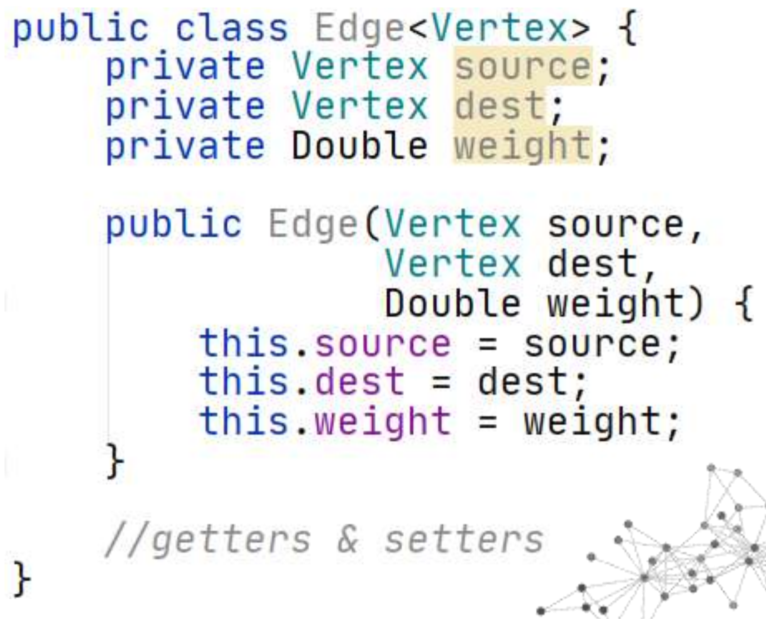
An edge-weighted graph is a graph model where we associate weights or costs with each edge

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





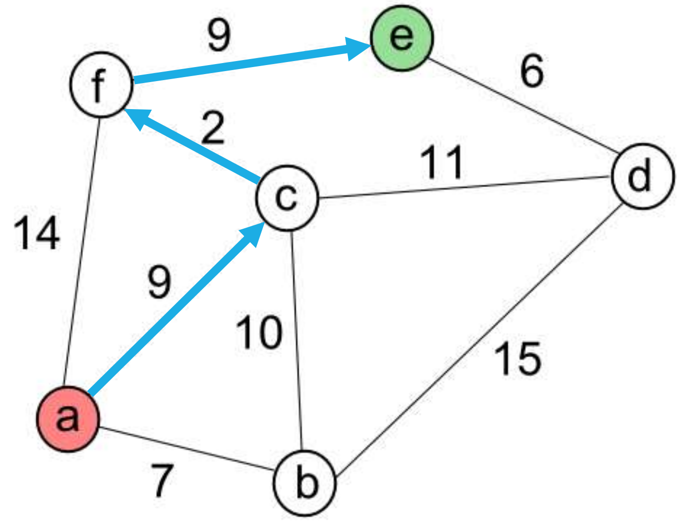
```
public class Edge<Vertex> {
    private Vertex source;
    private Vertex dest;
    private Double weight;

    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

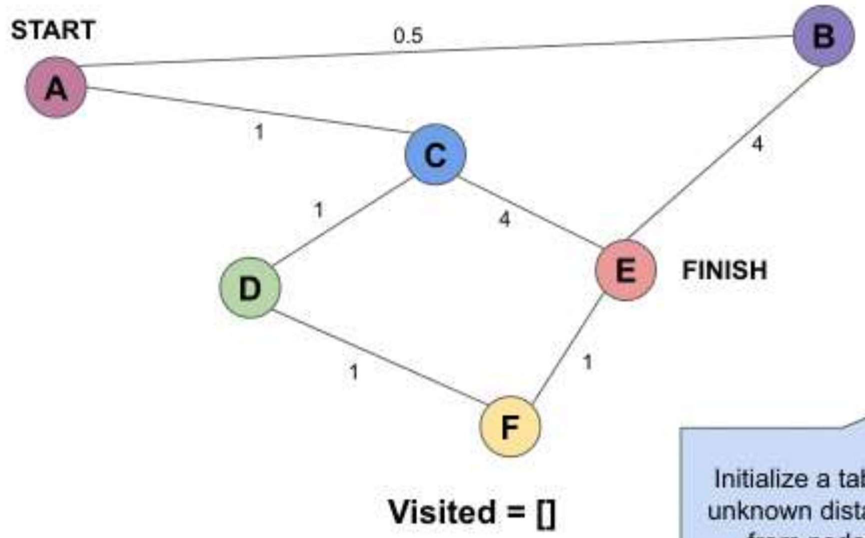
The **shortest path** from vertex **a** to vertex **e** in an edge-weighted digraph is a directed path from **a** to **e** with the property that no other such path has a lower weight



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	?	?

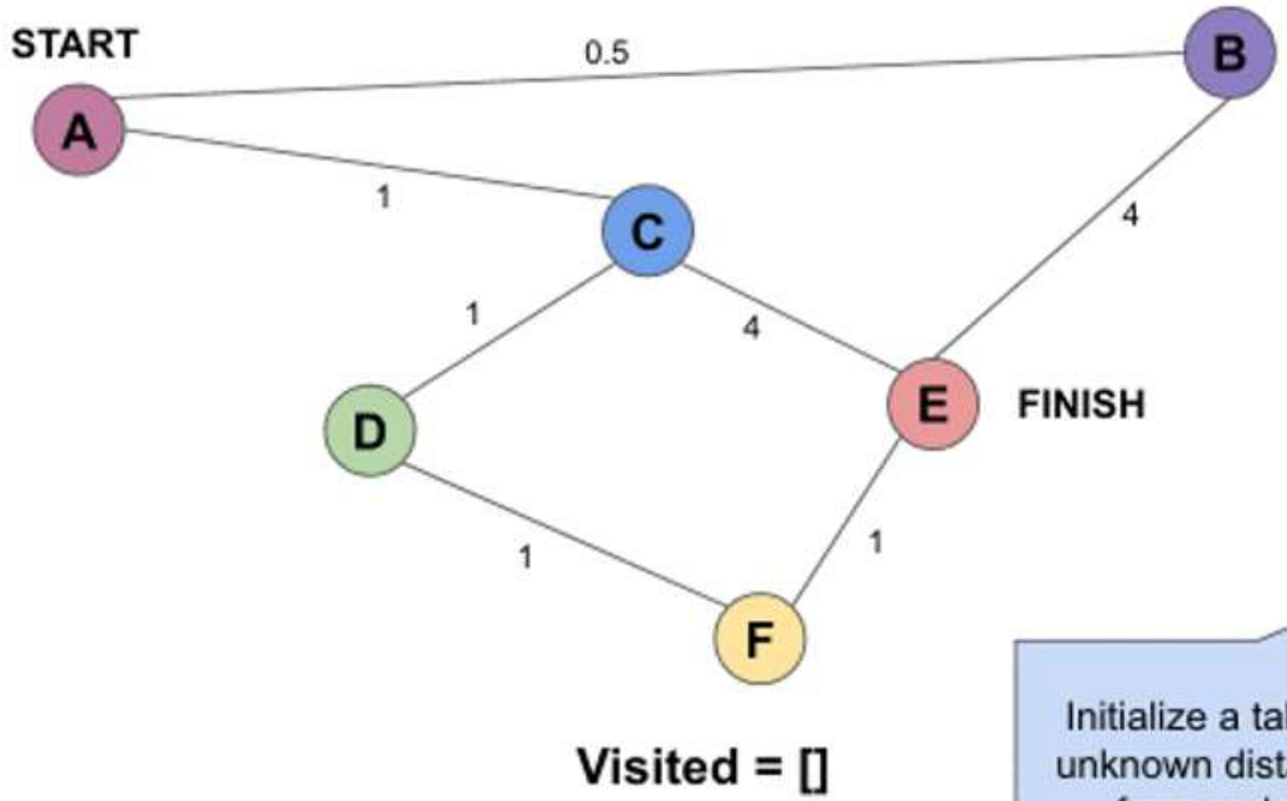
DIJKSTRA’S ALGORITHM

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

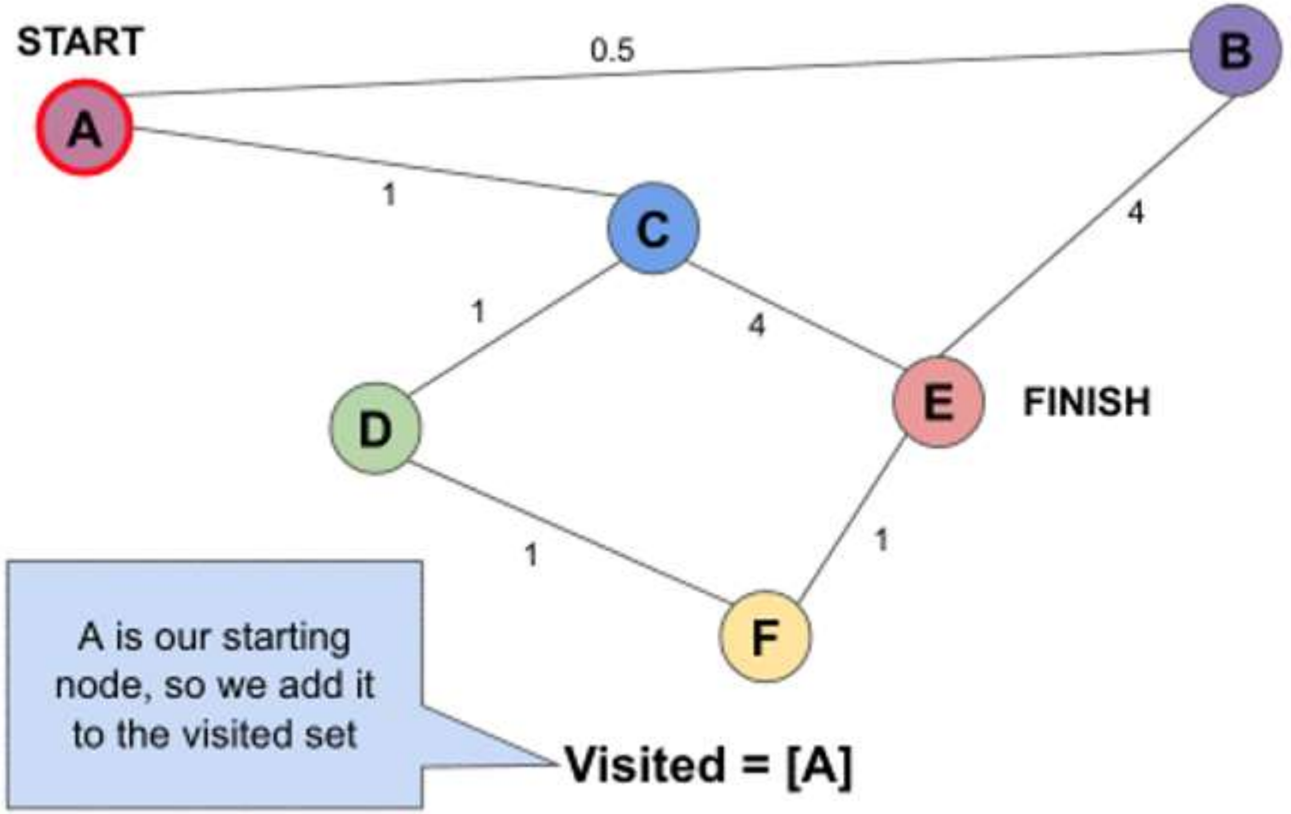
DIJKSTRA'S ALGORITHM



Initialize a table of unknown distances from node A

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

DIJKSTRA'S ALGORITHM

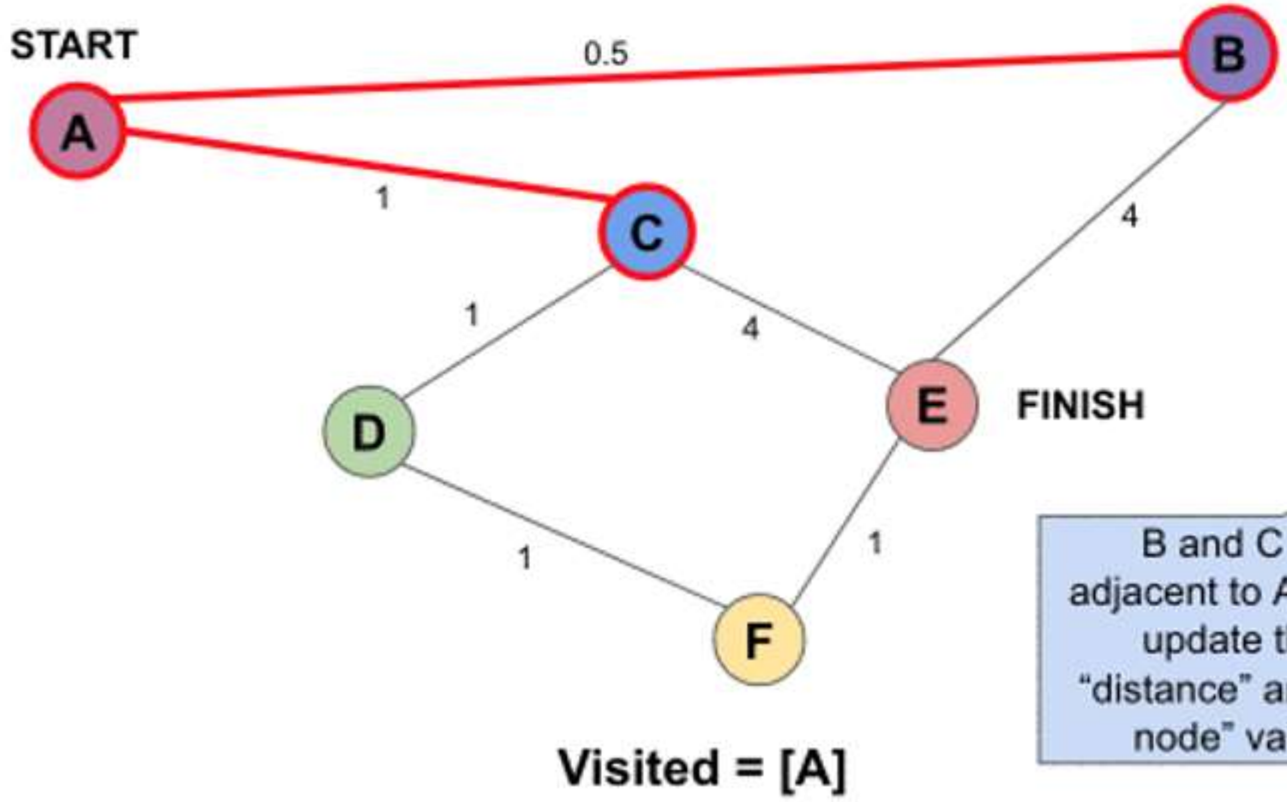


A is our starting node, so we add it to the visited set

Visited = [A]

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

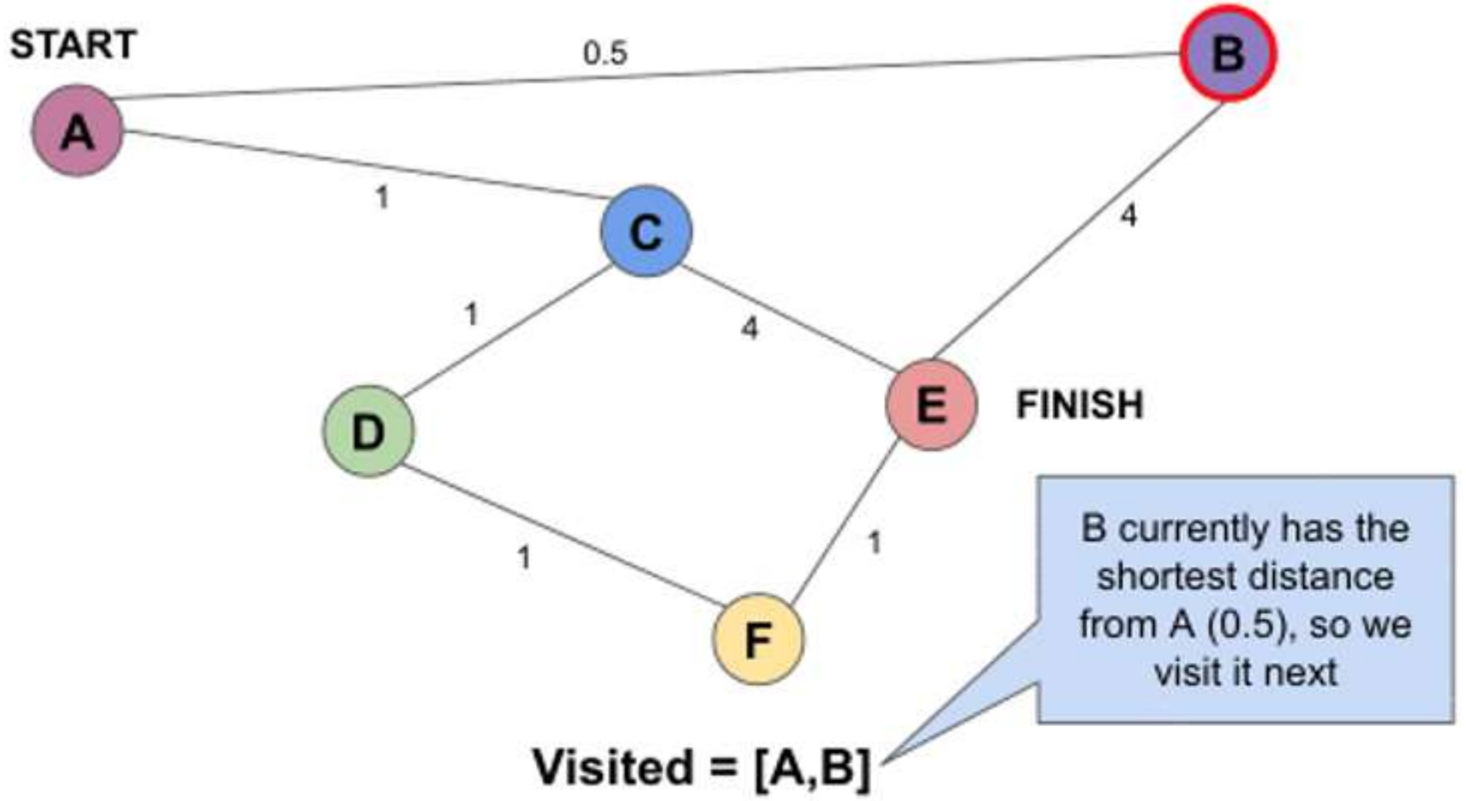
DIJKSTRA'S ALGORITHM



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	?	?

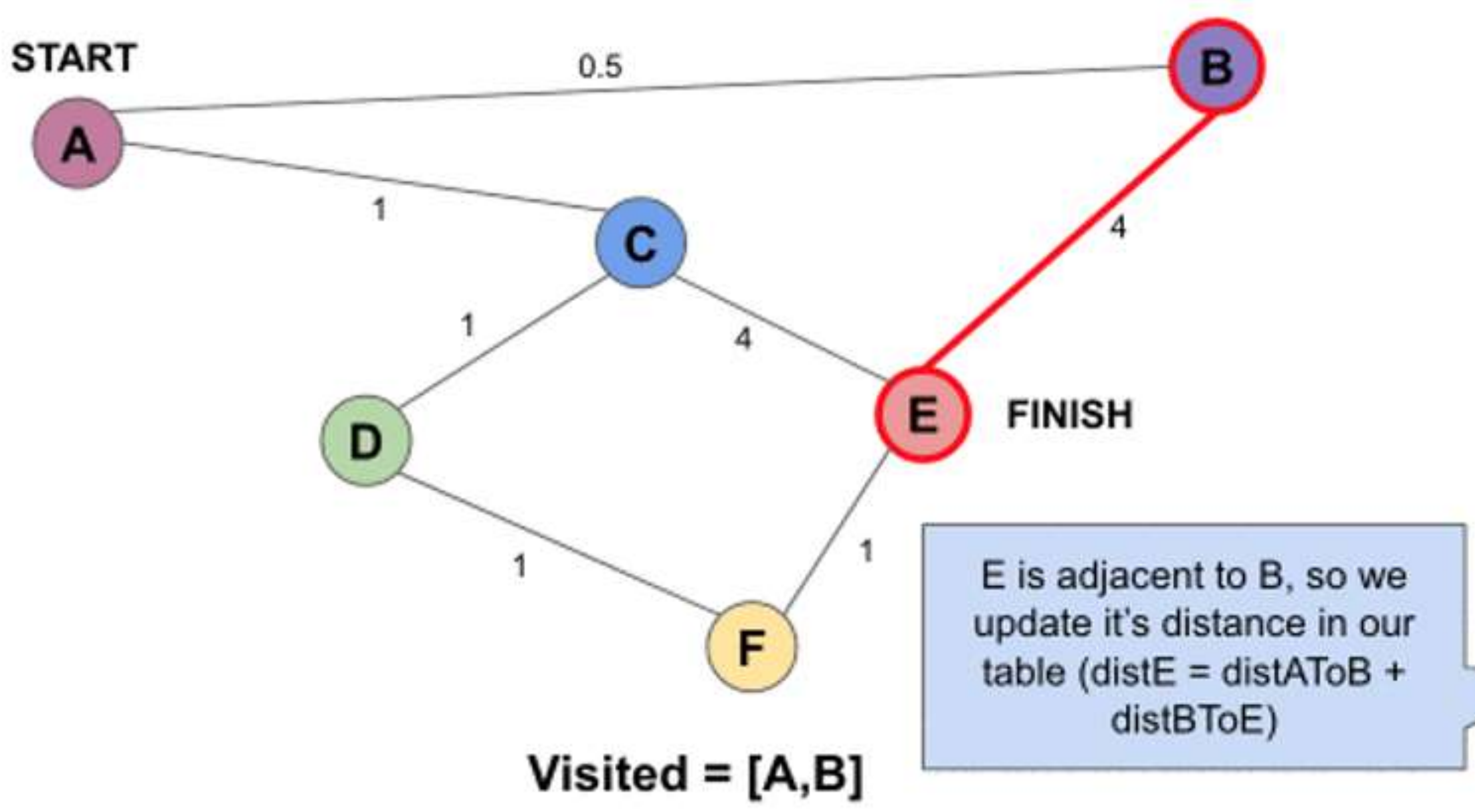
DIJKSTRA'S ALGORITHM



B currently has the shortest distance from A (0.5), so we visit it next

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

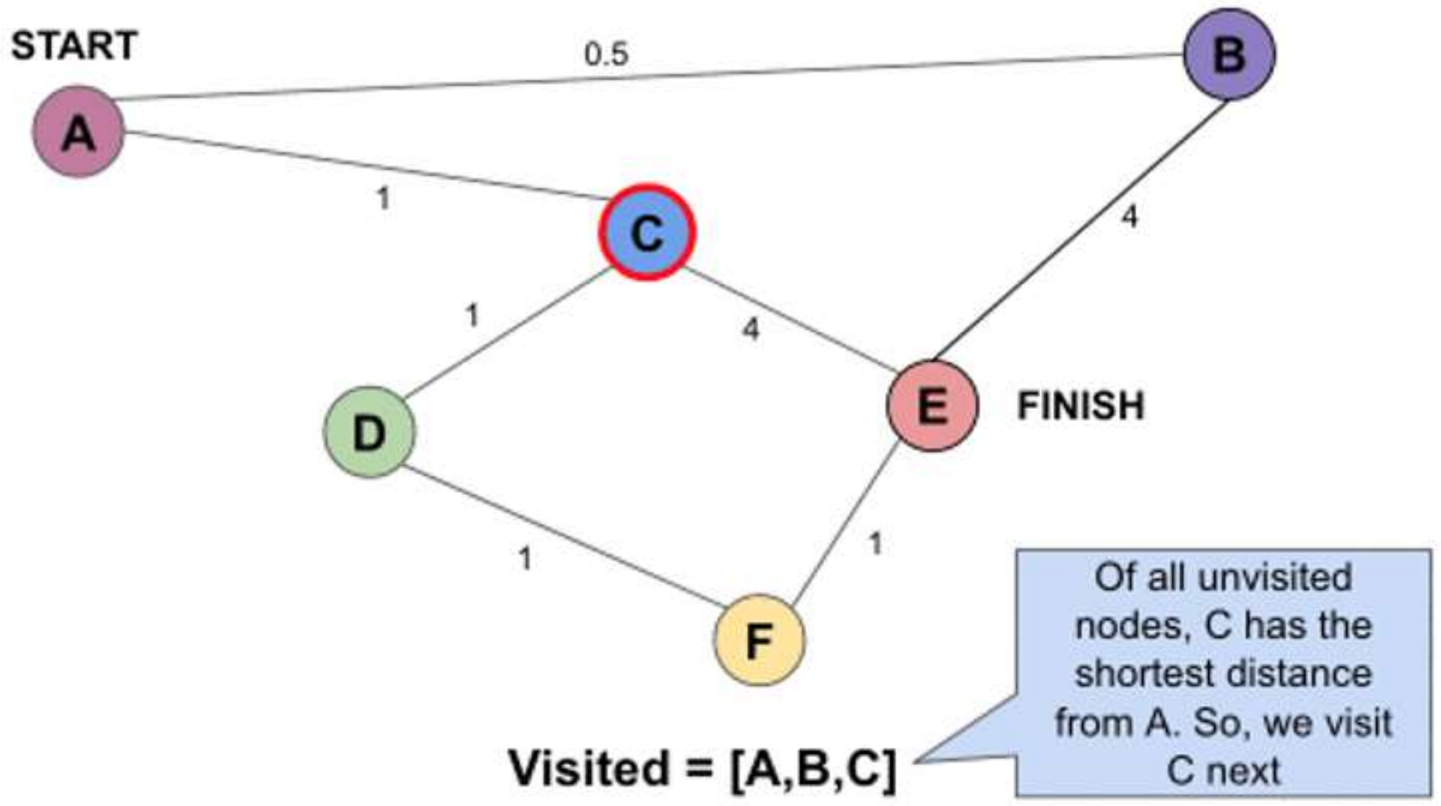
DIJKSTRA'S ALGORITHM



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

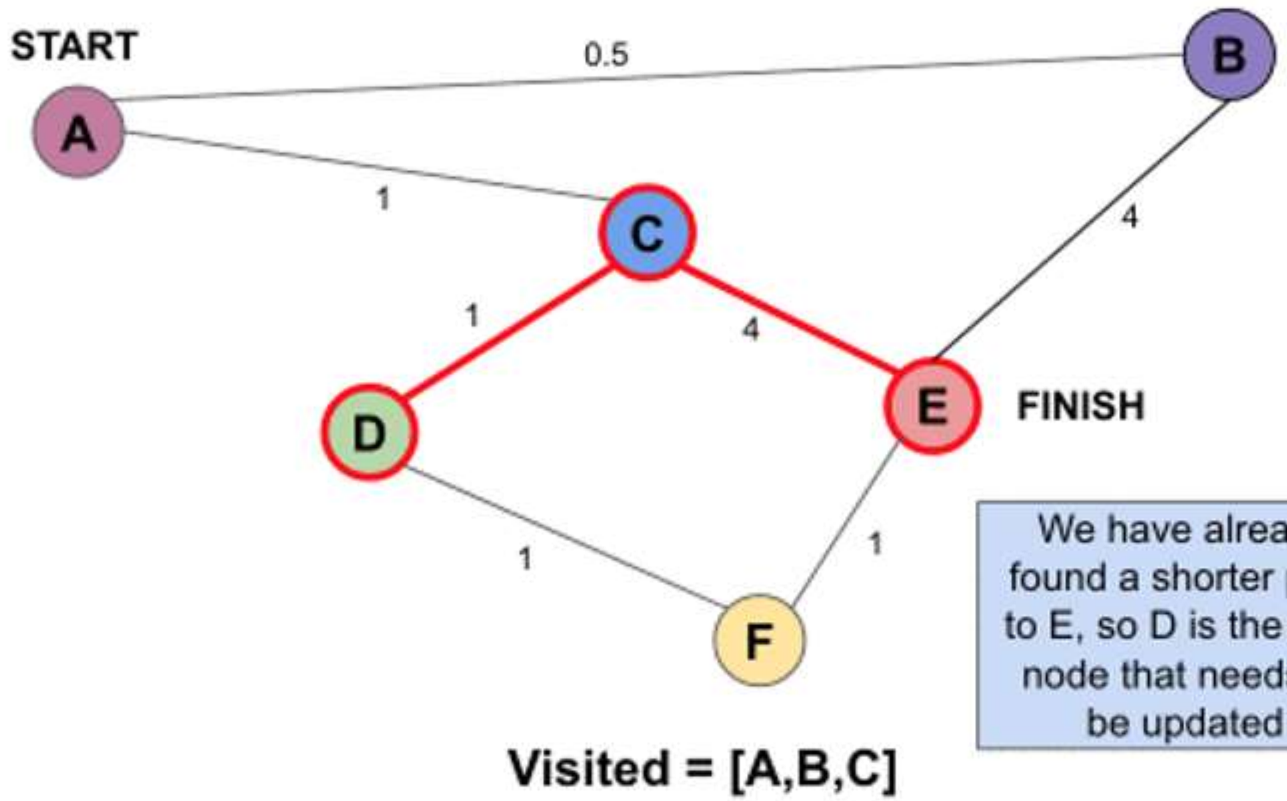


Visited = [A,B,C]

Of all unvisited nodes, C has the shortest distance from A. So, we visit C next

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

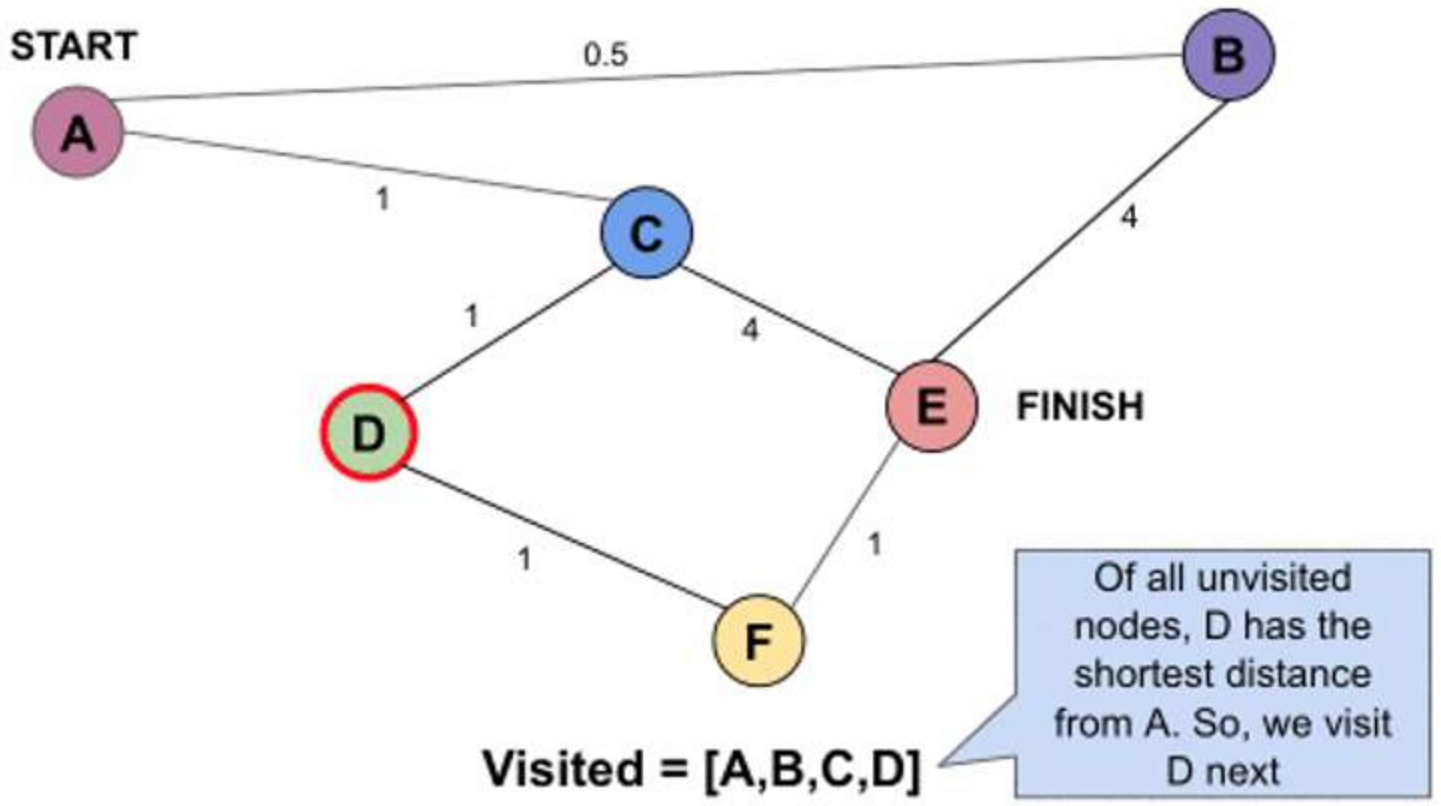
DIJKSTRA'S ALGORITHM



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

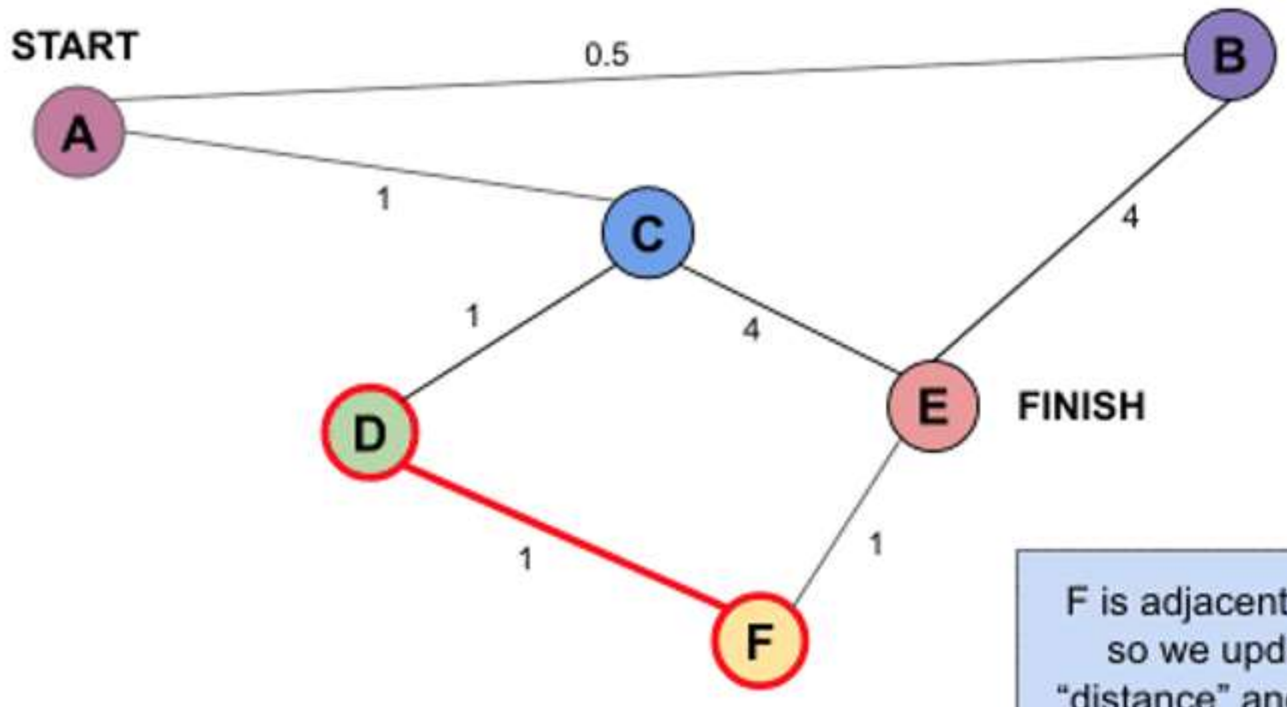
	Distance	Last Node
A	0	
B	0.5	A
C	1	A
D	**2**	**C**
E	4.5	B
F	?	?

DIJKSTRA'S ALGORITHM


START
A
0.5
B
1
C
4
1
4
D
E
FINISH
1
1
F
Visited = [A,B,C,D]
Of all unvisited nodes, D has the shortest distance from A. So, we visit D next

	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

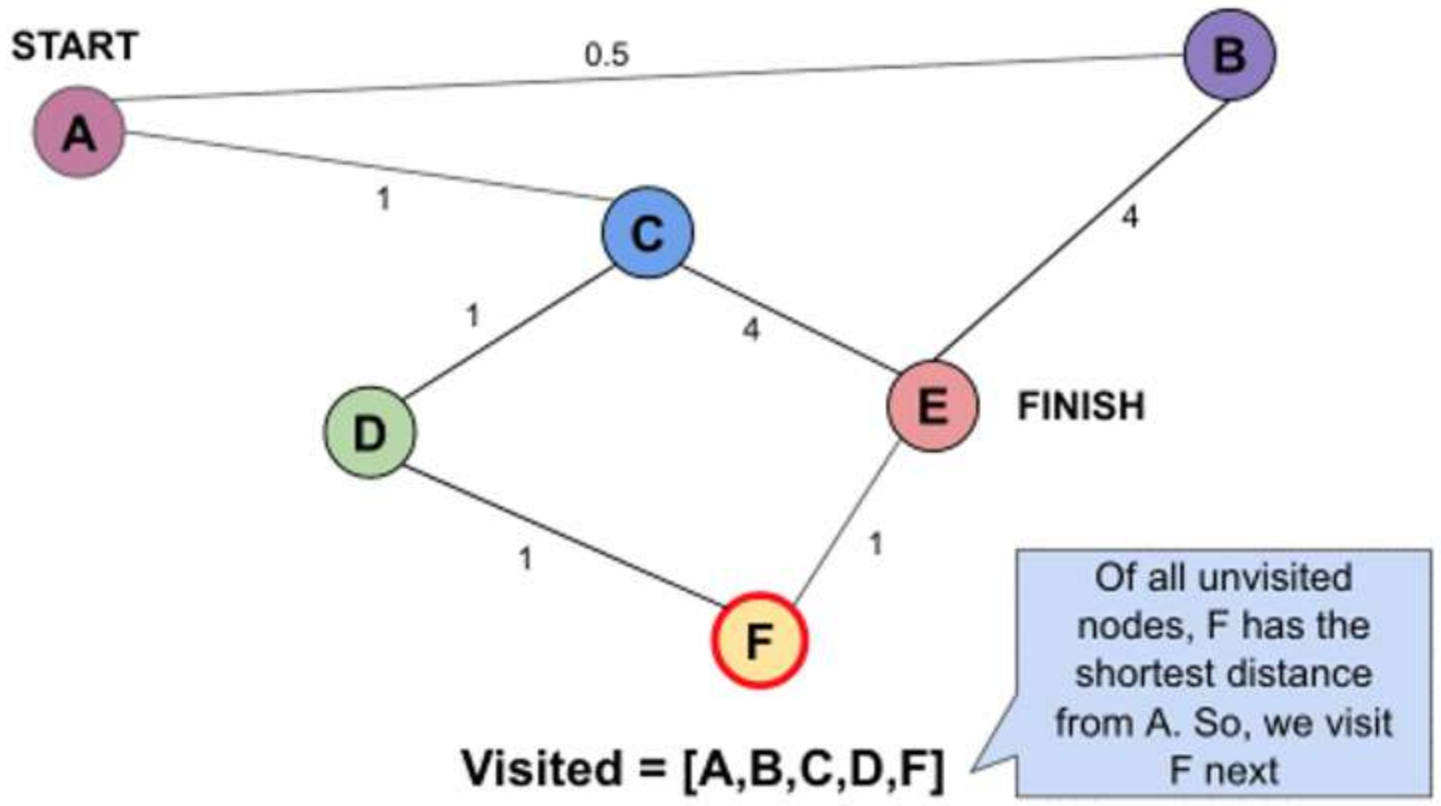


Visited = [A,B,C,D]

F is adjacent to D, so we update "distance" and "last node"

	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

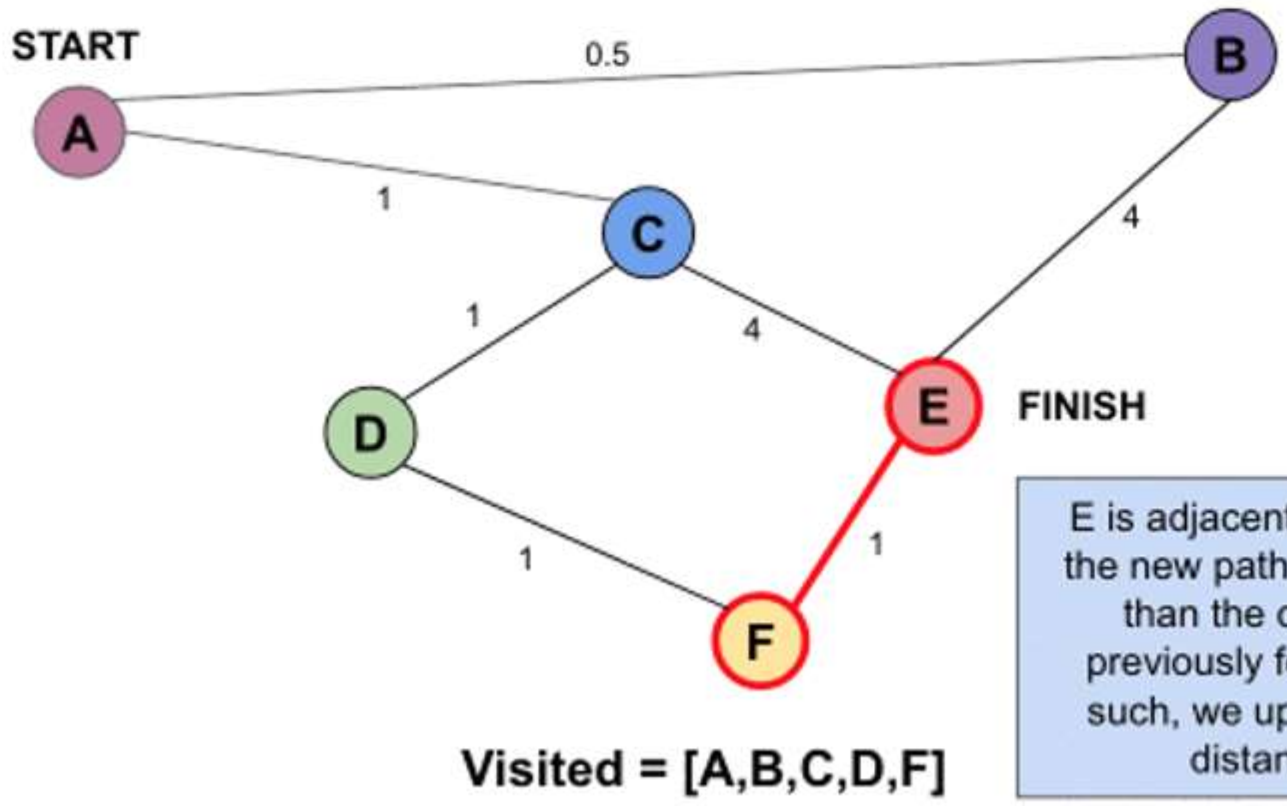


Visited = [A,B,C,D,F]

Of all unvisited nodes, F has the shortest distance from A. So, we visit F next

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

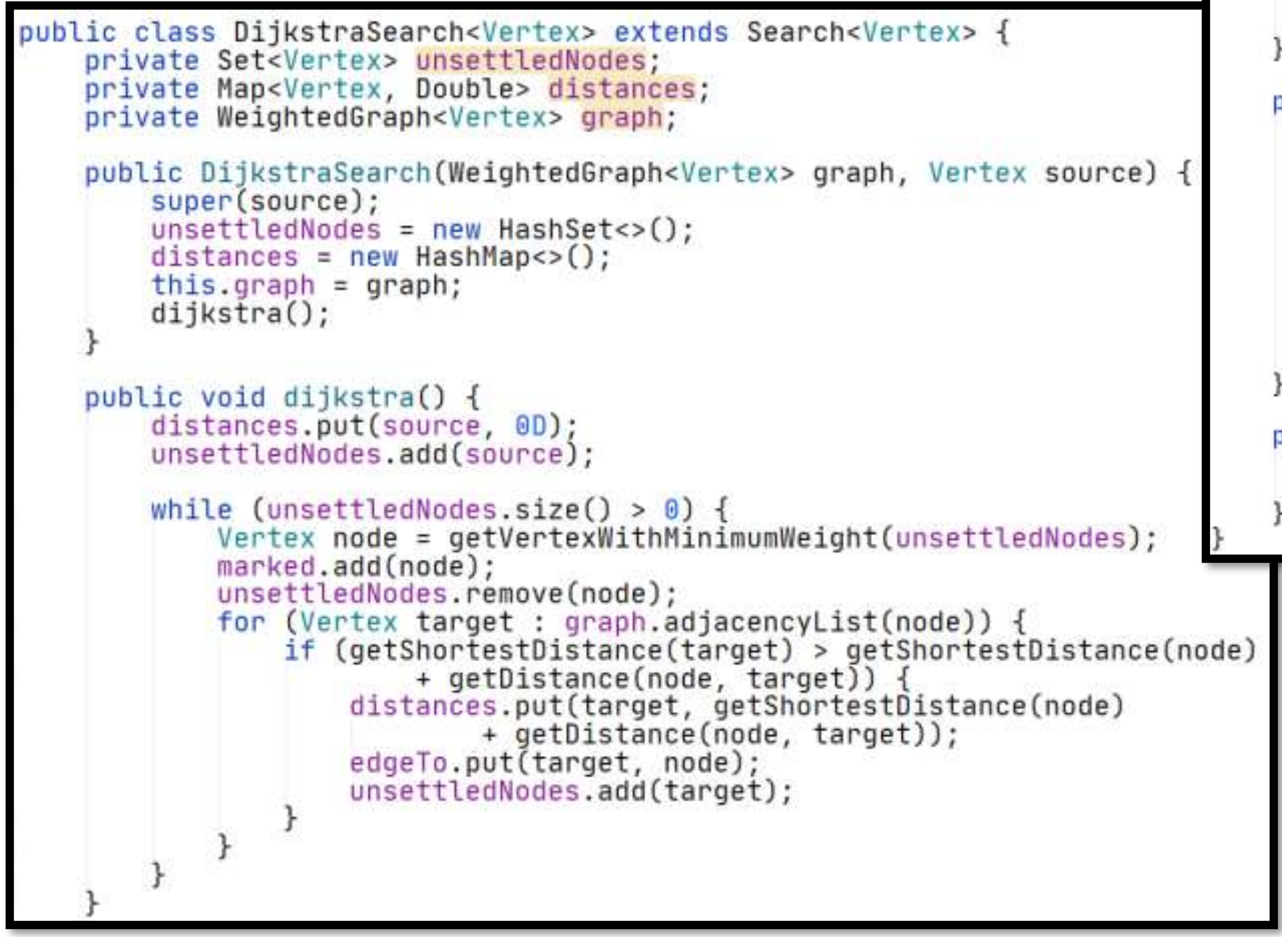
DIJKSTRA’S ALGORITHM



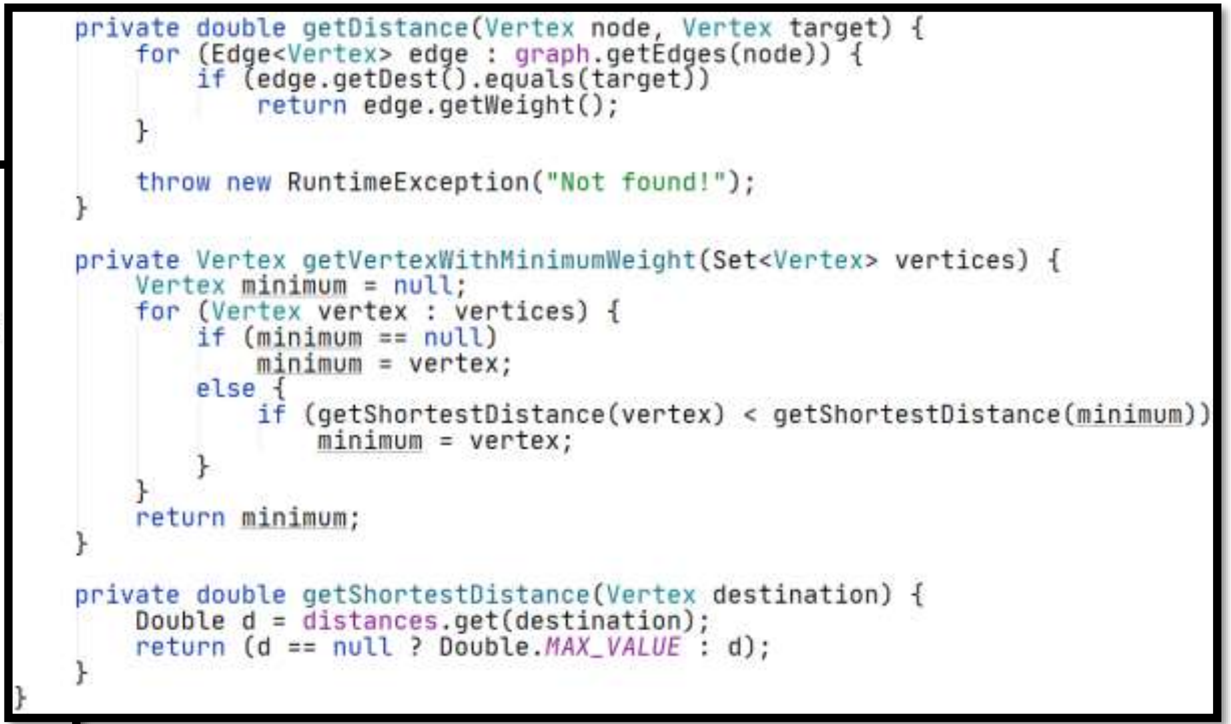
E is adjacent to F, and the new path is shorter than the one we previously found! As such, we update the distance

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

IMPLEMENTATION



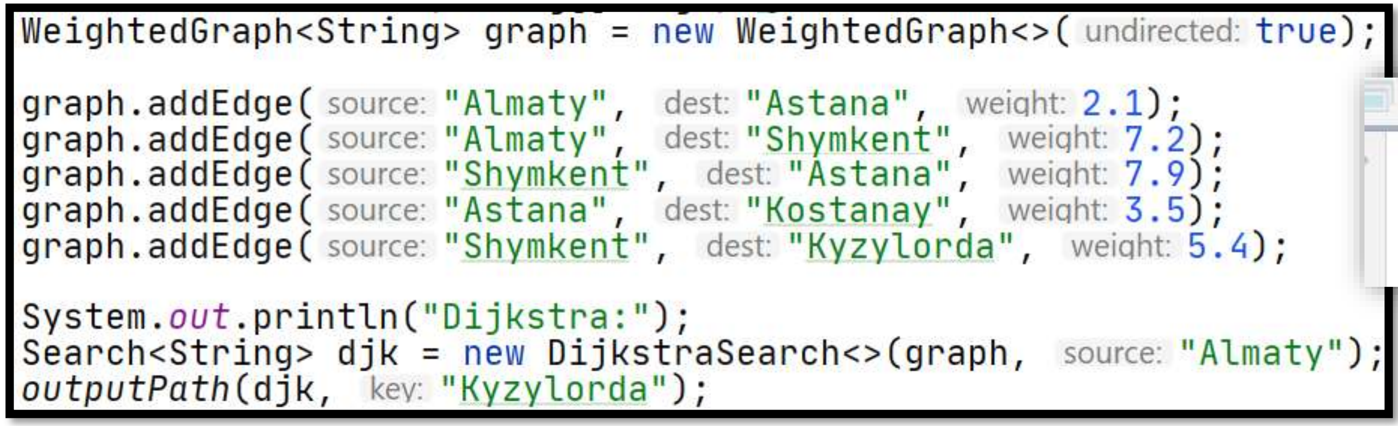
```
public class DijkstraSearch<Vertex> extends Search<Vertex> {
    private Set<Vertex> unsettledNodes;
    private Map<Vertex, Double> distances;
    private WeightedGraph<Vertex> graph;

    public DijkstraSearch(WeightedGraph<Vertex> graph, Vertex source) {
        super(source);
        unsettledNodes = new HashSet<>();
        distances = new HashMap<>();
        this.graph = graph;
        dijkstra();
    }

    public void dijkstra() {
        distances.put(source, 0D);
        unsettledNodes.add(source);

        while (unsettledNodes.size() > 0) {
            Vertex node = getVertexWithMinimumWeight(unsettledNodes);
            marked.add(node);
            unsettledNodes.remove(node);
            for (Vertex target : graph.adjacencyList(node)) {
                if (getShortestDistance(target) > getShortestDistance(node)
                        + getDistance(node, target)) {
                    distances.put(target, getShortestDistance(node)
                            + getDistance(node, target));
                    edgeTo.put(target, node);
                    unsettledNodes.add(target);
                }
            }
        }
    }
```



```
    private double getDistance(Vertex node, Vertex target) {
        for (Edge<Vertex> edge : graph.getEdges(node)) {
            if (edge.getDest().equals(target))
                return edge.getWeight();
        }

        throw new RuntimeException("Not found!");
    }

    private Vertex getVertexWithMinimumWeight(Set<Vertex> vertices) {
        Vertex minimum = null;
        for (Vertex vertex : vertices) {
            if (minimum == null)
                minimum = vertex;
            else {
                if (getShortestDistance(vertex) < getShortestDistance(minimum))
                    minimum = vertex;
            }
        }
        return minimum;
    }

    private double getShortestDistance(Vertex destination) {
        Double d = distances.get(destination);
        return (d == null ? Double.MAX_VALUE : d);
    }
}
```

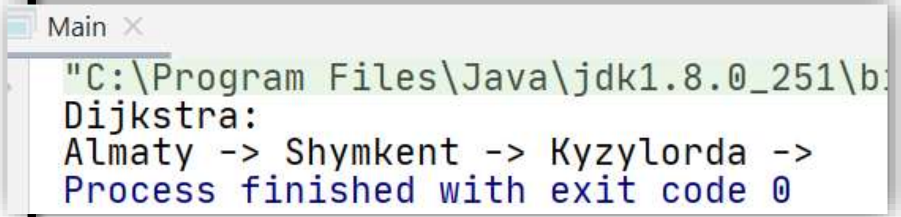
https://github.com/aghia7/example/tree/master/src

RESULTS

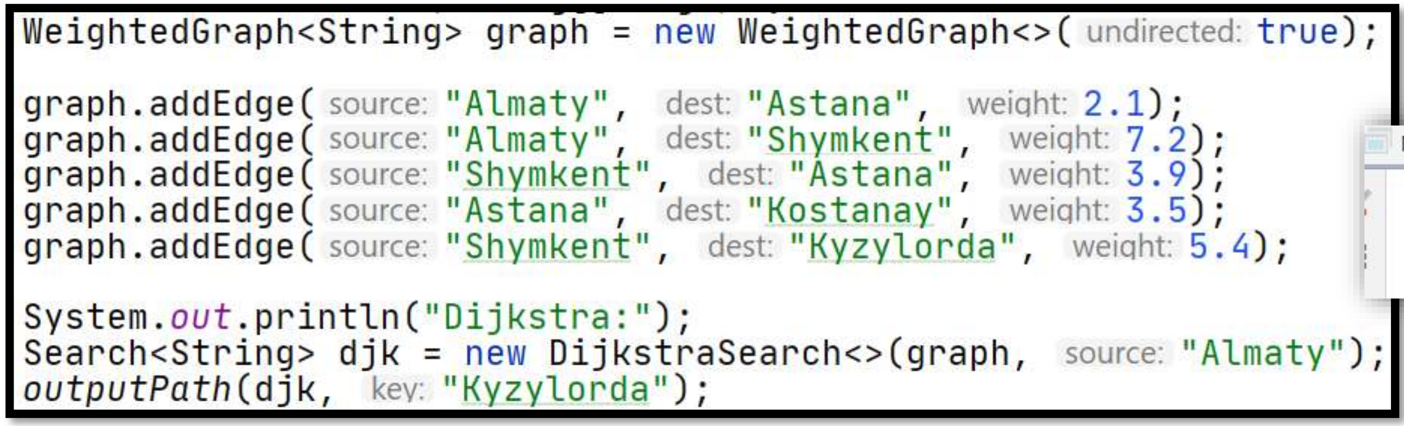


```
WeightedGraph<String> graph = new WeightedGraph<>( undirected: true);

graph.addEdge( source: "Almaty",  dest: "Astana",  weight: 2.1);
graph.addEdge( source: "Almaty",  dest: "Shymkent",  weight: 7.2);
graph.addEdge( source: "Shymkent",  dest: "Astana",  weight: 7.9);
graph.addEdge( source: "Astana",  dest: "Kostanay",  weight: 3.5);
graph.addEdge( source: "Shymkent",  dest: "Kyzylorda",  weight: 5.4);

System.out.println("Dijkstra:");
Search<String> djk = new DijkstraSearch<>(graph,  source: "Almaty");
outputPath(djk,  key: "Kyzylorda");
```

Shymkent-Astana : 7.9

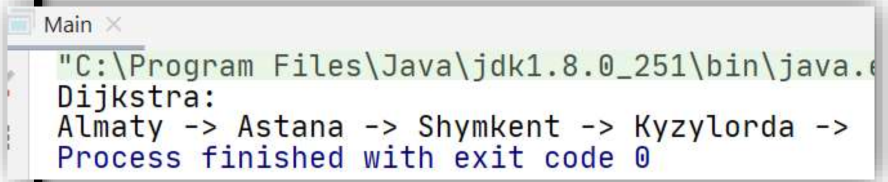


```
Main
"C:\Program Files\Java\jdk1.8.0_251\b
Dijkstra:
Almaty -> Shymkent -> Kyzylorda ->
Process finished with exit code 0
```

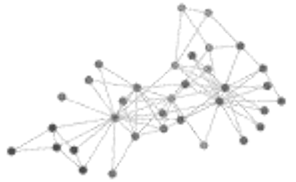


```
WeightedGraph<String> graph = new WeightedGraph<>( undirected: true);

graph.addEdge( source: "Almaty",  dest: "Astana",  weight: 2.1);
graph.addEdge( source: "Almaty",  dest: "Shymkent",  weight: 7.2);
graph.addEdge( source: "Shymkent",  dest: "Astana",  weight: 3.9);
graph.addEdge( source: "Astana",  dest: "Kostanay",  weight: 3.5);
graph.addEdge( source: "Shymkent",  dest: "Kyzylorda",  weight: 5.4);

System.out.println("Dijkstra:");
Search<String> djk = new DijkstraSearch<>(graph,  source: "Almaty");
outputPath(djk,  key: "Kyzylorda");
```

Shymkent-Astana : 3.9



```
Main
"C:\Program Files\Java\jdk1.8.0_251\bin\java.e
Dijkstra:
Almaty -> Astana -> Shymkent -> Kyzylorda ->
Process finished with exit code 0
```



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!