THE GRAPH

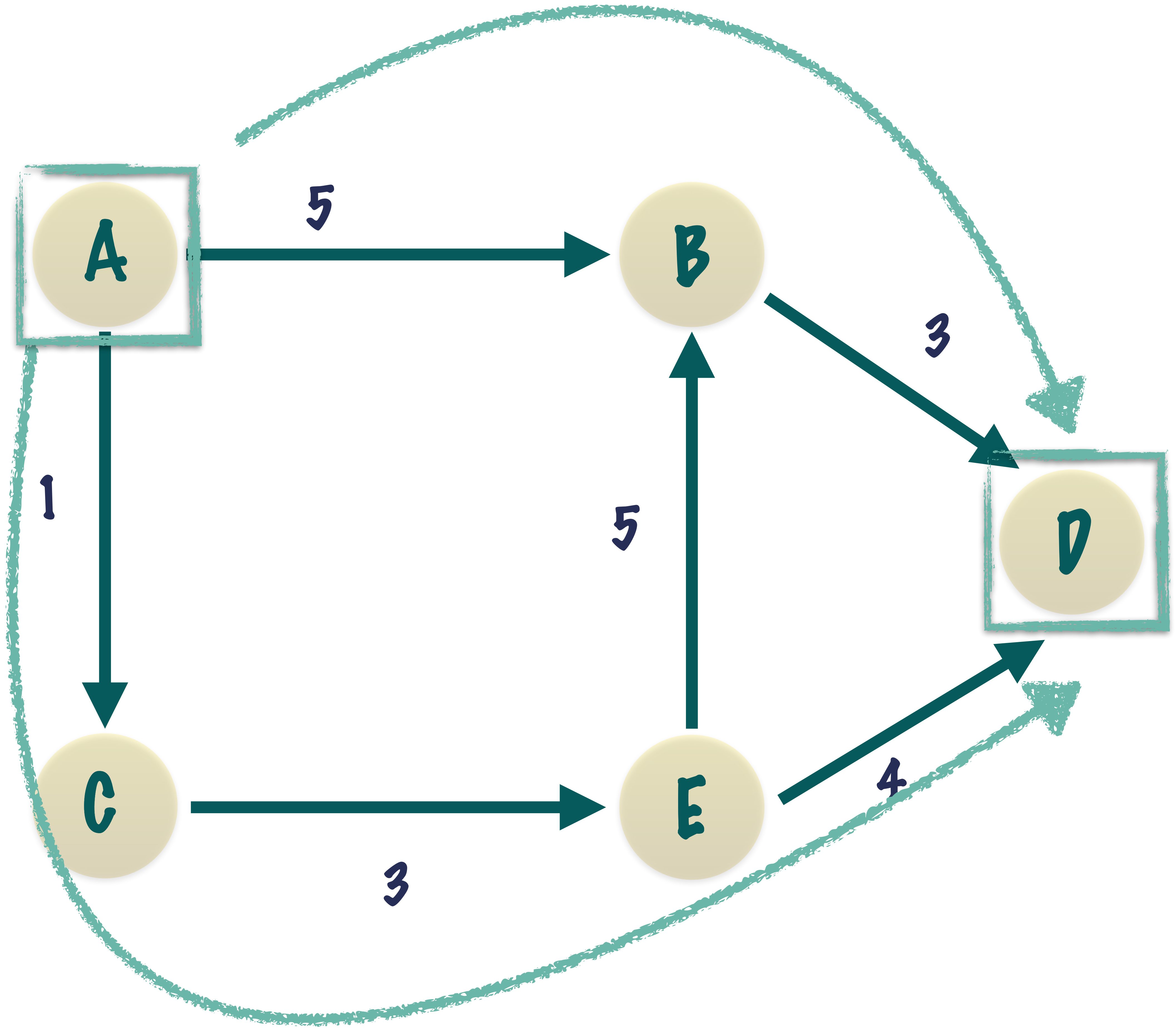
DESIGN A SHORTEST PATH ALGORITHM WITH THE FEWEST EDGES A FOR A GRAPH WITH POSITIVE EDGE WEIGHTS

DESIGN A SHORTEST PATH ALGORITHM WITH THE FEWEST EDGES A FOR A GRAPH WITH POSITIVE EDGE WEIGHTS

LET'S UNDERSTAND THE PROBLEM FIRST

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH

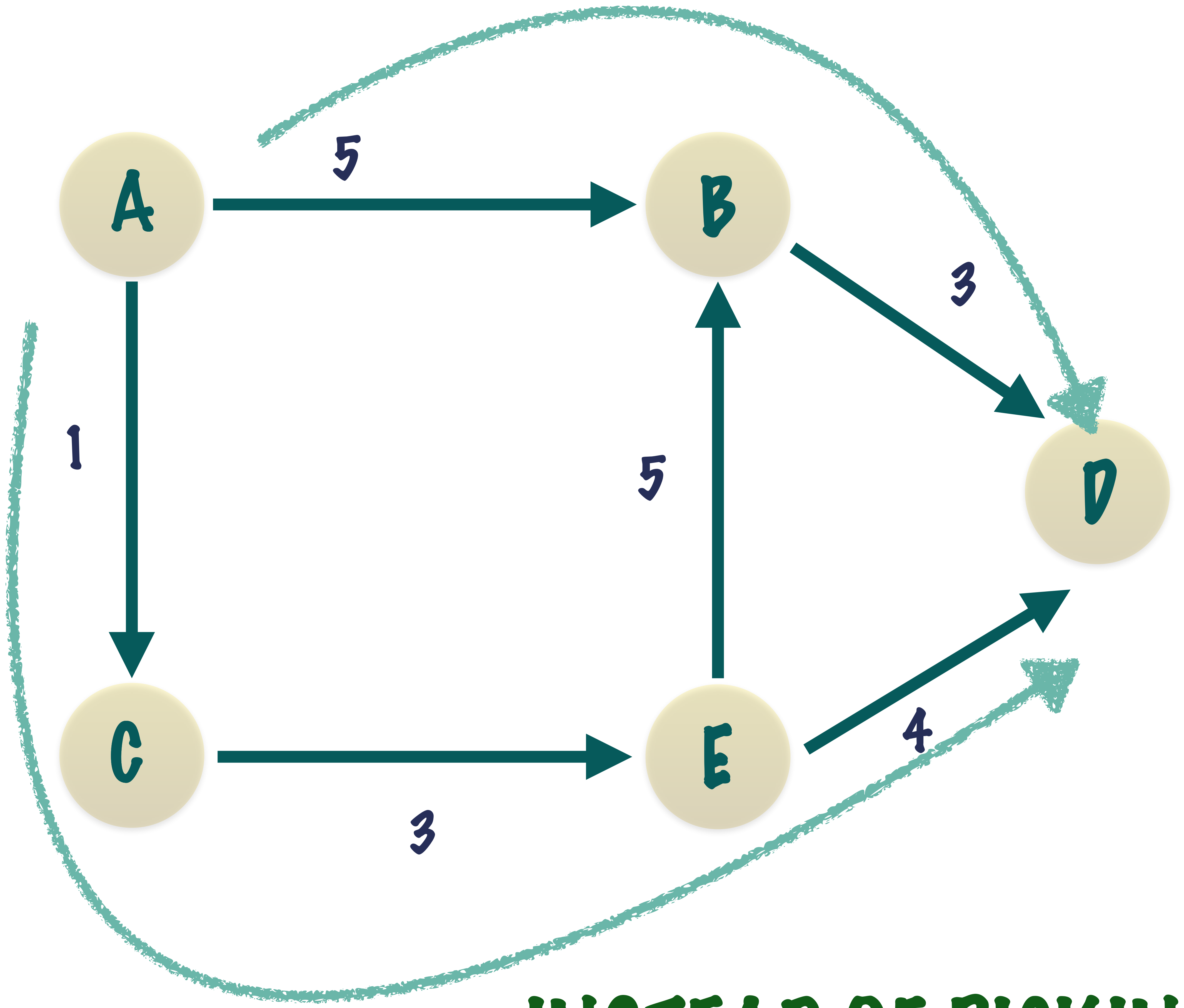
LET'S SAY WE WANT TO GO FROM A->D



THERE ARE TWO PATHS

WITH THE SAME WEIGHT = 8

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH



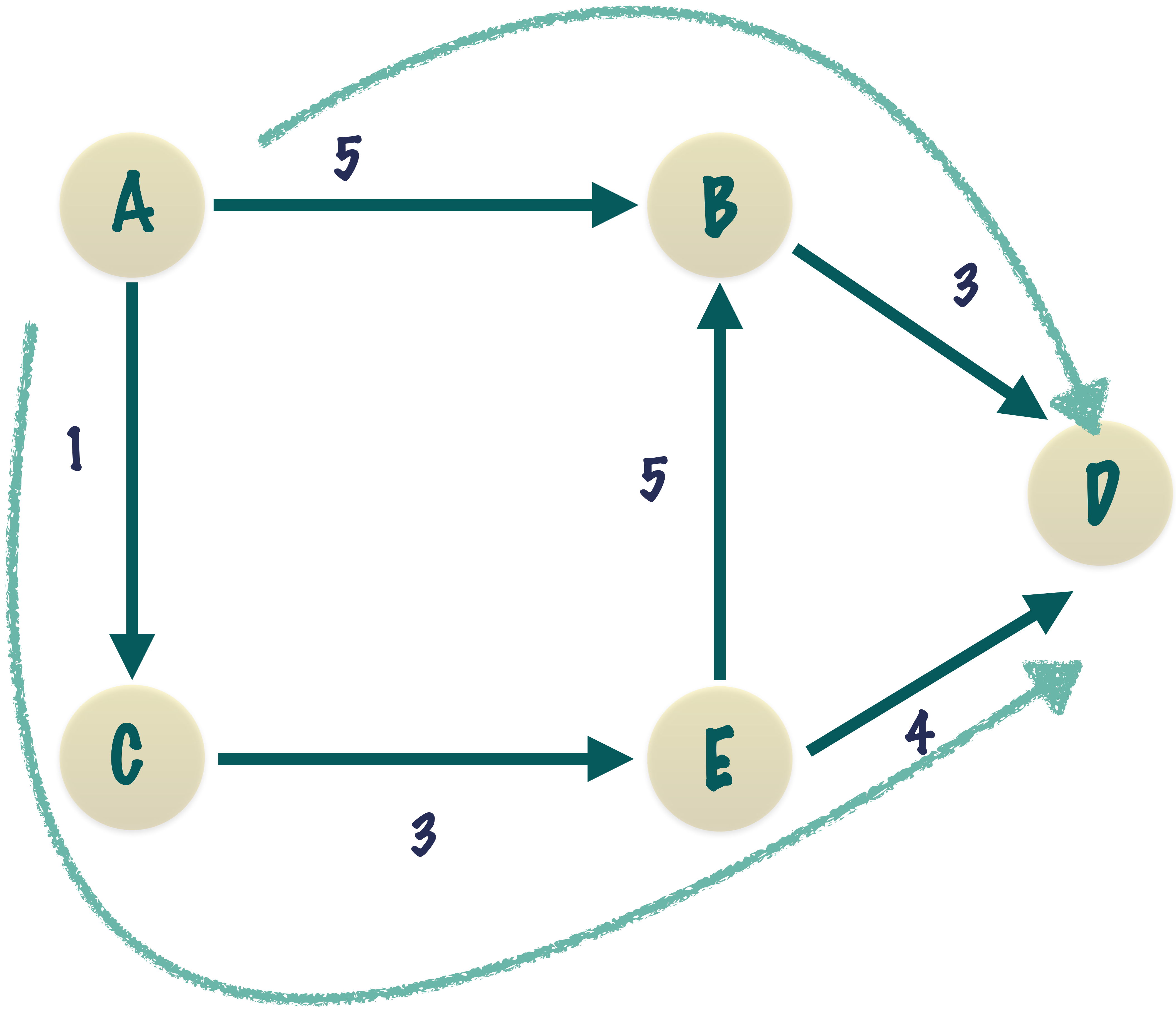
THERE ARE TWO PATHS

WITH THE SAME WEIGHT = 8

	DISTANCE
A -> B -> D	8
A -> C -> E -> D	8

INSTEAD OF PICKING ONE OF THESE AT RANDOM - WE USE THE NUMBER OF EDGES IN THE PATH TO CHOOSE THE SHORTEST ROUTE!

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH



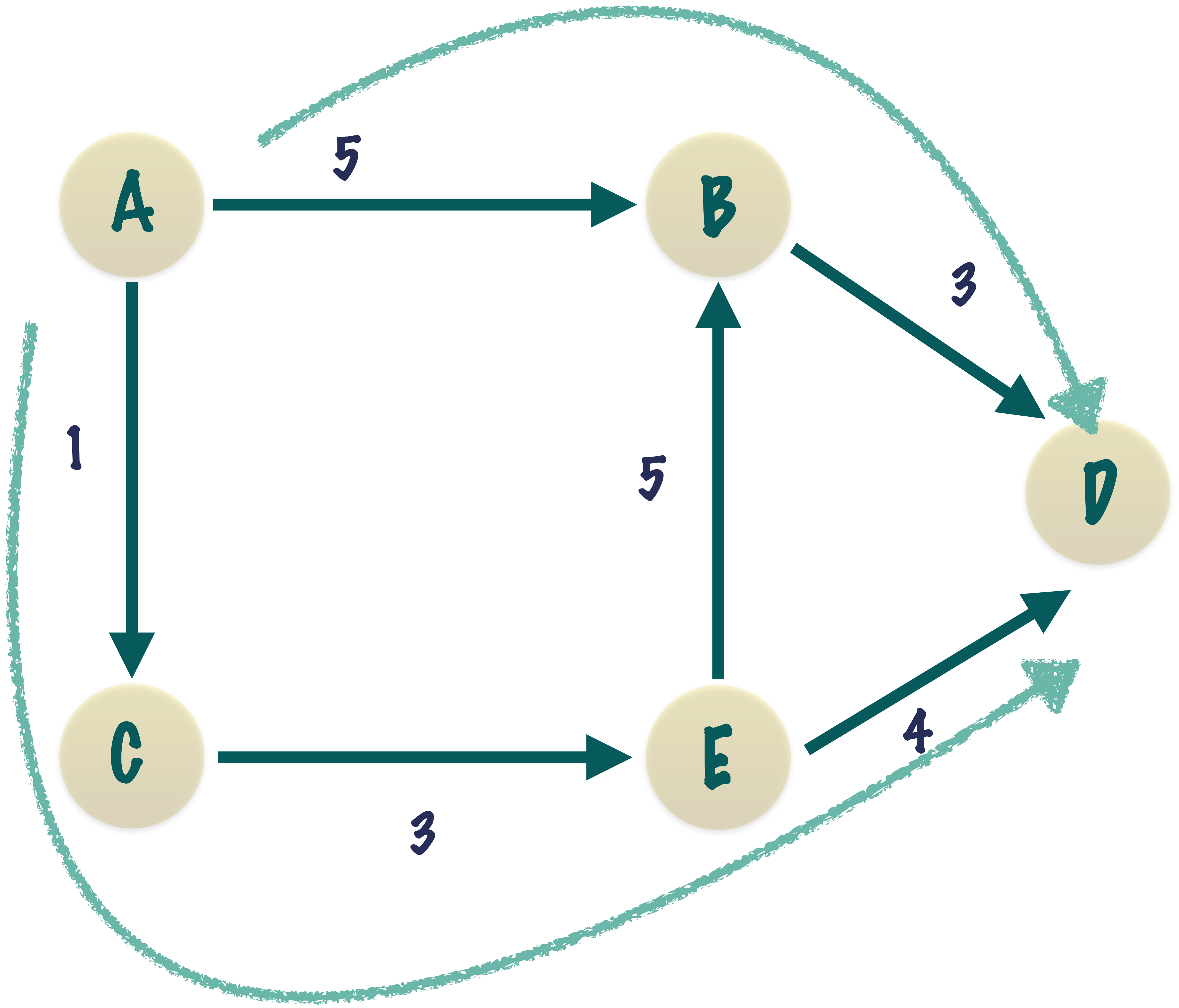
	DISTANCE	EDGES
A -> B -> D	8	2
A -> C -> E -> D	8	3

BREAK THE TIE USING THE NUMBER OF EDGES!

A -> B -> D

IS THE SHORTEST PATH

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH



WE WILL USE A VARIATION OF DIJKSTRA'S ALGORITHM

THE SHORTEST PATH IS NOW A TUPLE

(DISTANCE, NUMBER OF EDGES)

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH

WE WILL USE A VARIATION OF DIJKSTRA'S ALGORITHM

THE SHORTEST PATH IS NOW A TUPLE

(DISTANCE, NUMBER OF EDGES)

2 SPECIFIC CHANGES IN THE WAY WE USE THE ALGORITHM

1 WHAT IS STORED IN THE PRIORITY QUEUE

OLD

B	3
E	4

NEW

B	3	1
E	4	1

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH

THE SHORTEST PATH IS NOW A TUPLE (DISTANCE, NUMBER OF EDGES)

1 WHAT IS STORED IN THE PRIORITY QUEUE

OLD

B	3
E	4

NEW

B	3	1
E	4	1

IF TWO VERTICES HAVE THE SAME DISTANCE, THE ONE WITH FEWER EDGES HAS THE HIGHEST PRIORITY I.E. IS THE SHORTER PATH

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH

THE SHORTEST PATH IS NOW A TUPLE (DISTANCE, NUMBER OF EDGES)

2 HOW THE DISTANCE TABLE IS UPDATED AND USED

USE A NEW DISTANCE TABLE WITH AN ADDITIONAL COLUMN SHOWING THE NUMBER OF EDGES TO REACH THAT VERTEX FROM THE SOURCE

VERTEX	DISTANCE	LAST VERTEX	EDGES
A	0	A	0
B	INF		
C	INF		
D	INF		
E	INF		

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH

2 HOW THE DISTANCE TABLE IS UPDATED AND USED

OLD

WE UPDATE DISTANCE TABLE IF

DISTANCE [NEIGHBOUR] > DISTANCE [VERTEX] + WEIGHT OF EDGE [VERTEX, NEIGHBOUR]

NEW

WE UPDATE DISTANCE TABLE IF

DISTANCE [NEIGHBOUR] > DISTANCE [VERTEX] + WEIGHT OF EDGE [VERTEX, NEIGHBOUR]

OR

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH

2 HOW THE DISTANCE TABLE IS UPDATED AND USED

NEW

DISTANCE [NEIGHBOUR] > DISTANCE [VERTEX] + WEIGHT OF EDGE [VERTEX, NEIGHBOUR]

OR

DISTANCE [NEIGHBOUR] = DISTANCE [VERTEX] + WEIGHT OF EDGE [VERTEX, NEIGHBOUR]

EDGES [NEIGHBOUR] > EDGES [VERTEX] + 1

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH

2 HOW THE DISTANCE TABLE IS UPDATED AND USED

NEW

DISTANCE [NEIGHBOUR] > DISTANCE [VERTEX] + WEIGHT OF EDGE [VERTEX, NEIGHBOUR]

OR

DISTANCE [NEIGHBOUR] = DISTANCE [VERTEX] + WEIGHT OF EDGE [VERTEX, NEIGHBOUR]

EDGES [NEIGHBOUR] > EDGES [VERTEX] + 1

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH

2 HOW THE DISTANCE TABLE IS UPDATED AND USED

NEW

DISTANCE [NEIGHBOUR] > DISTANCE [VERTEX] + WEIGHT OF EDGE [VERTEX, NEIGHBOUR]

OR

DISTANCE [NEIGHBOUR] = DISTANCE [VERTEX] + WEIGHT OF EDGE [VERTEX, NEIGHBOUR]

EDGES [NEIGHBOUR] $\boxed{>}$ EDGES [VERTEX] + 1

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH

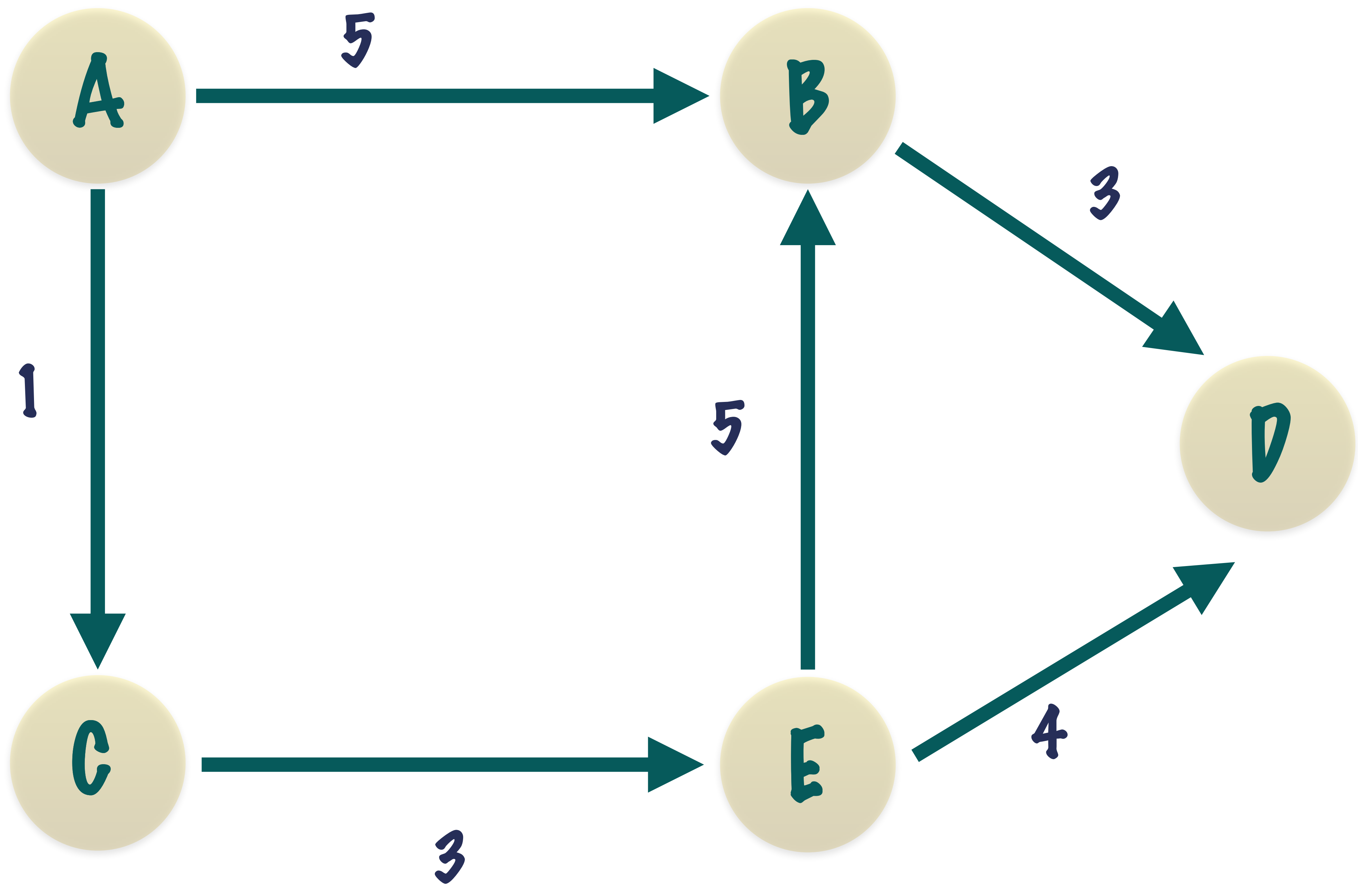
DISTANCE [NEIGHBOUR] = DISTANCE [VERTEX] + WEIGHT OF EDGE [VERTEX, NEIGHBOUR]

EDGES [NEIGHBOUR] > EDGES [VERTEX] + 1

IT SIMPLY MEANS THAT IF TWO PATHS HAVE SAME DISTANCE, WE CHOOSE THE ONE WITH FEWER EDGES!

SHORTEST PATH WITH THE FEWEST EDGES IN A POSITIVE WEIGHTED GRAPH

IT SIMPLY MEANS THAT IF TWO PATHS HAVE SAME DISTANCE, WE CHOOSE THE ONE WITH FEWER EDGES!



	DISTANCE	EDGE
A -> B -> D	8	2
A -> C -> E -> D	8	3

DISTANCE EDGE INFO DATA STRUCTURE

```java
public static class DistanceEdgeInfo {

    private Integer distance;
    private Integer numEdges;
    private Integer lastVertex;

    public DistanceEdgeInfo() {
        distance = Integer.MAX_VALUE;
        lastVertex = -1;
        numEdges = Integer.MAX_VALUE;
    }

    public Integer getDistance() {
        return distance;
    }

    public Integer getLastVertex() {
        return lastVertex;
    }

    public Integer getNumEdges() {
        return numEdges;
    }

    public void setInfo(int lastVertex, int distance, int numEdges) {
        this.distance = distance;
        this.lastVertex = lastVertex;
        this.numEdges = numEdges;
    }
}
```

REPRESENTS 3 BITS OF INFORMATION ABOUT ANY VERTEX:

1. THE DISTANCE FROM THE SOURCE
2. THE NUMBER OF EDGES FROM THE SOURCE
3. THE LAST VERTEX IN THE PATH

A SINGLE SETTER TO SET ALL THE INFORMATION FOR THIS VERTEX

THE VERTEX INFO FOR THE PRIORITY QUEUE

```java
public static class VertexInfo {

    private Integer vertexId;
    private Integer distance;
    private Integer numEdges;

    public VertexInfo(int vertexId, int distance, int edges) {
        this.vertexId = vertexId;
        this.distance = distance;
        this.numEdges = edges;
    }

    public Integer getVertexId() {
        return vertexId;
    }

    public Integer getDistance() {
        return distance;
    }

    public Integer getNumEdges() {
        return numEdges;
    }
}
```

VERTEX INFO WHICH HOLDS THE DISTANCE AND NUMBER OF EDGES TO THE VERTEX TO USE IN THE PRIORITY QUEUE

ADD EVERY EDGE TO THE PRIORITY QUEUE

KEEP TRACK OF THE VERTICES ALREADY VISITED, EACH EDGE SHOULD ADD A NEW VERTEX TO THE SET TILL WE GET NUMBER OF VERTICES - 1 EDGES

THE EDGE MAP TRACKS THE EDGES ADDED TO THE SPANNING TREE TO SEE IF IT FORMS A CYCLE

THE SPANNING TREE IS THE SET OF EDGES CONNECTING ALL THE NODES OF THE GRAPH, AN EDGE IS REPRESENTED BY "01" IF IT CONNECTS VERTICES 0 AND 1

BUILD THE DISTANCE TABLE - SETUP

```
public static Map<Integer, DistanceEdgeInfo> buildDistanceTable(Graph graph, int source) {
    Map<Integer, DistanceEdgeInfo> distanceTable = new HashMap<>();
    PriorityQueue<VertexInfo> queue = new PriorityQueue<>(new Comparator<VertexInfo>() {
        @Override
        public int compare(VertexInfo v1, VertexInfo v2) {
            if (v1.getDistance().compareTo(v2.getDistance()) != 0) {
                return  v1.getDistance().compareTo(v2.getDistance());
            }
            return v1.getNumEdges().compareTo(v2.getNumEdges());
        }
    });

    for (int j = 0; j < graph.getNumVertices(); j++) {
        distanceTable.put(j, new DistanceEdgeInfo());
    }

    distanceTable.get(source).setInfo(source, 0 /* distance */, 0 /* numEdges */);

    VertexInfo sourceVertexInfo = new VertexInfo(source, 0, 0);
    queue.add(sourceVertexInfo);

    Map<Integer, VertexInfo> vertexInfoMap = new HashMap<>();
    vertexInfoMap.put(source, sourceVertexInfo);
```

THE PRIORITY QUEUE CHECKS BOTH THE DISTANCE AND THE NUMBER OF EDGES FOR A VERTEX. IF THE DISTANCE IS THE SAME ONLY THEN THE NUMBER OF EDGES IS CHECKED

THE REST OF THE SETUP IS EXACTLY LIKE DIJKSTRA'S ALGORITHM

BUILD THE EDGE MAP AND SPANNING TREE - PROCESS

```
while (!queue.isEmpty()) {
    VertexInfo currentVertexInfo = queue.poll();

    for (Integer neighbour : graph.getAdjacentVertices(currentVertexInfo.getVertexId())) {
        // Get the distance and number of edges from the current vertex to the neighbour.
        int distance = distanceTable.get(currentVertexInfo.getVertexId()).getDistance()
                + graph.getWeightedEdge(currentVertexInfo.getVertexId(), neighbour);
        int edges = distanceTable.get(currentVertexInfo.getVertexId()).getNumEdges() + 1;

        int neighbourDistance = distanceTable.get(neighbour).getDistance();
        if (neighbourDistance > distance || ((neighbourDistance == distance)
                && (distanceTable.get(neighbour).getNumEdges() > edges))) {

            // Update the distance table for the neighbour with the new information
            distanceTable.get(neighbour).setInfo(
                    currentVertexInfo.getVertexId(), distance, edges);

            VertexInfo neighbourVertexInfo = vertexInfoMap.get(neighbour);
            if (neighbourVertexInfo != null) {
                queue.remove(neighbourVertexInfo);
            }

            // Set up the updated neigbour vertex info with the new distance
            // and number of edges.
            neighbourVertexInfo = new VertexInfo(neighbour, distance, edges);
            queue.add(neighbourVertexInfo);
            vertexInfoMap.put(neighbour, neighbourVertexInfo);
        }
    }
}

return distanceTable;
```

REMOVE THE HIGHEST PRIORITY ELEMENT FROM THE QUEUE - THE GREEDY ALGORITHM

FOR ADJACENT VERTICES FIND THE NEW DISTANCE AND THE NEW NUMBER OF EDGES TO GET TO THAT VERTEX

VISIT AND UPDATE THE VERTEX IF THE NEW DISTANCE IS SMALLER OR IF THE DISTANCES ARE THE SAME THE NEW NUMBER OF EDGES IS SMALLER

THE REST OF THE CODE IS IDENTICAL TO DIJKSTRA'S ALGORITHM

SHORTEST PATH

```java
public static void shortestPath(Graph graph, Integer source, Integer destination) {
    Map<Integer, DistanceInfo> distanceTable = buildDistanceTable(graph, source);

    Stack<Integer> stack = new Stack<>();
    stack.push(destination);

    int previousVertex = distanceTable.get(destination).getLastVertex();
    while (previousVertex != -1 && previousVertex != source) {
        stack.push(previousVertex);
        previousVertex = distanceTable.get(previousVertex).getLastVertex();
    }

    if (previousVertex == -1) {
        System.out.println("There is no path from node: " + source
                + " to node: " + destination);
    }
    else {
        System.out.print("Smallest Path is " + source);
        while (!stack.isEmpty()) {
            System.out.print(" -> " +stack.pop());
        }
        System.out.println(" Dijkstra  DONE!");
    }
}
```

BUILD THE DISTANCE TABLE FOR THE ENTIRE GRAPH

BACKTRACK USING A STACK, START FROM THE DESTINATION NODE

BACKTRACK BY GETTING THE LAST VERTEX OF EVERY NODE AND ADDING IT TO THE STACK

IF NO VALID LAST VERTEX WAS FOUND IN THE DISTANCE TABLE, THERE WAS NO PATH FROM SOURCE TO DESTINATION