THE GRAPH

WEIGHTED GRAPH

WEIGHTED GRAPH

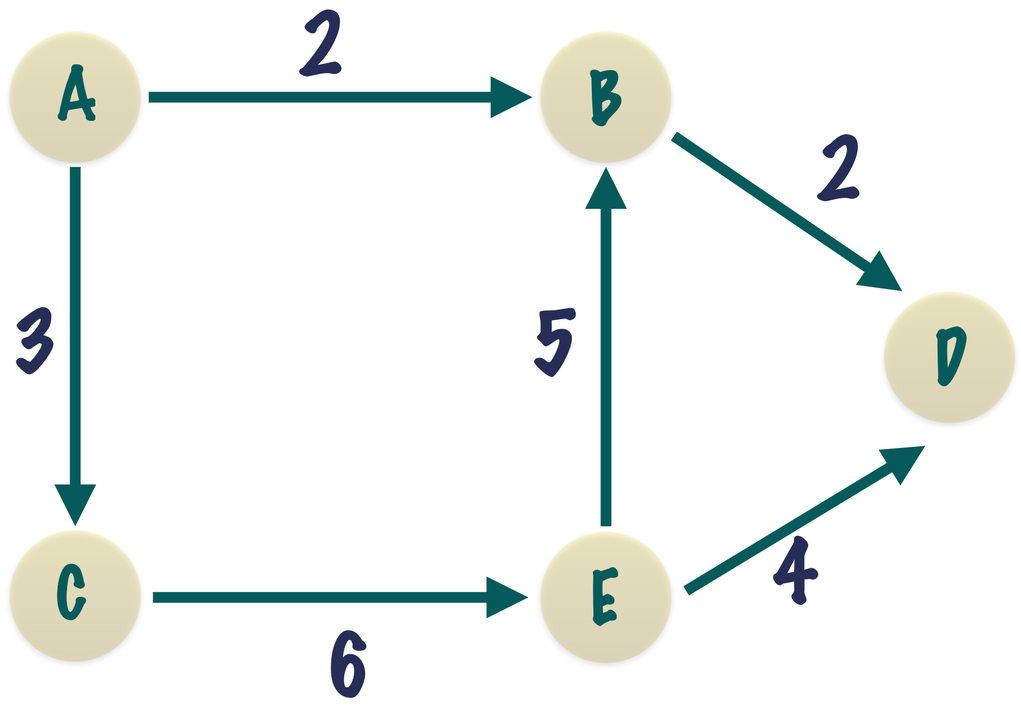
DEFINITION: A GRAPH HAVING A WEIGHT, OR NUMBER, ASSOCIATED WITH EACH EDGE.

THE WEIGHTS CAN BE -VE OR +VE!

THIS EDGE HAS A WEIGHT OF 2!

WEIGHTED GRAPH

DEFINITION: A GRAPH HAVING A WEIGHT, OR NUMBER, ASSOCIATED WITH EACH EDGE.



THE WEIGHTS CAN BE -VE OR +VE!

THESE WEIGHTS CAN REPRESENT ANYTHING!

IF IT WERE A MAP, THESE WEIGHTS COULD REPRESENT TIME TAKEN OR DISTANCE OR TRAFFIC CONDITIONS BETWEEN ANY TWO POINTS!

THE GRAPH

SHORTEST PATH ALGORITHMS

SHORTEST PATH IN WEIGHTED GRAPH

SHORTEST PATH IN WEIGHTED GRAPH

THE ALGORITHM HERE IS QUITE SIMILAR TO WHAT WE HAVE DISCUSSED BEFORE IN FINDING THE SHORTEST PATH IN AN UNWEIGHTED GRAPH

THERE ARE 3 MAJOR DIFFERENCES!

FINDING THE SHORTEST PATH IN A WEIGHTED GRAPH IS CALLED DIJKSTRA'S ALGORITHM!

SHORTEST PATH IN WEIGHTED GRAPH

THERE ARE 3 MAJOR DIFFERENCES!

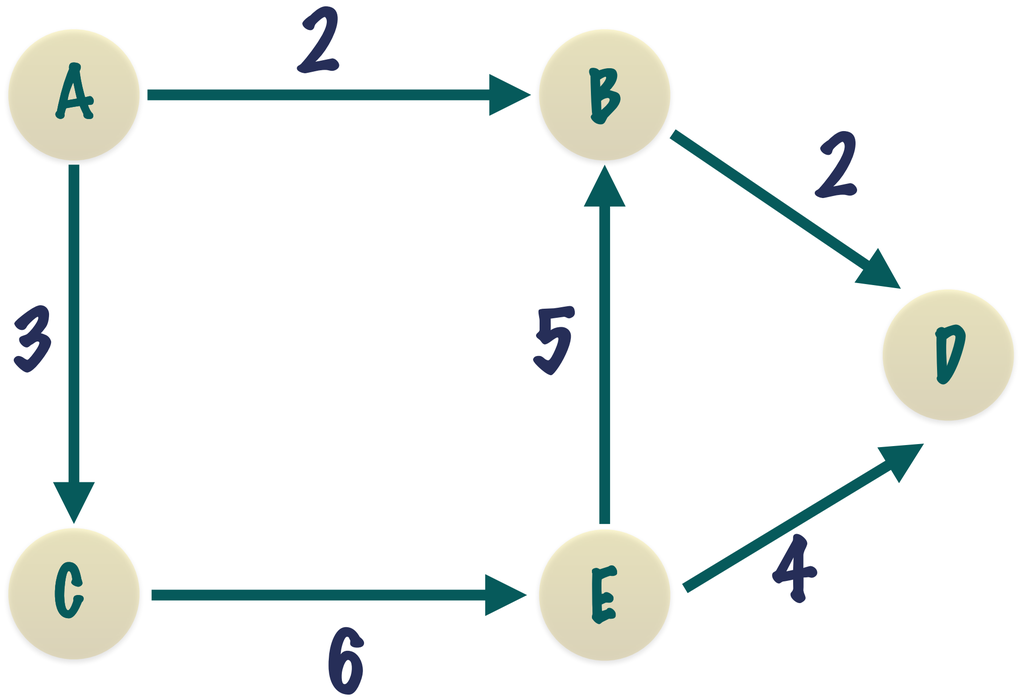
WE STILL USE THE DISTANCE TABLE TO STORE INFORMATION

1. THE DISTANCE FROM A NODE NOW HAS TO ACCOUNT FOR THE **WEIGHT OF THE EDGES TRAVERSED**

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

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

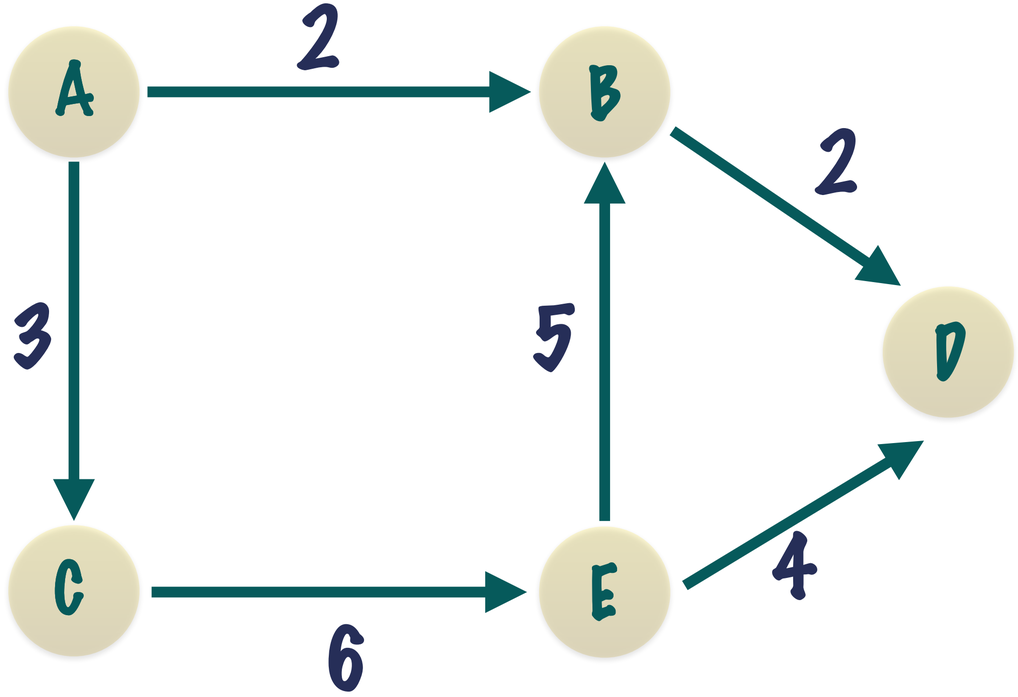
SHORTEST PATH IN WEIGHTED GRAPH



DISTANCE FROM A TO B?

A -> B 2

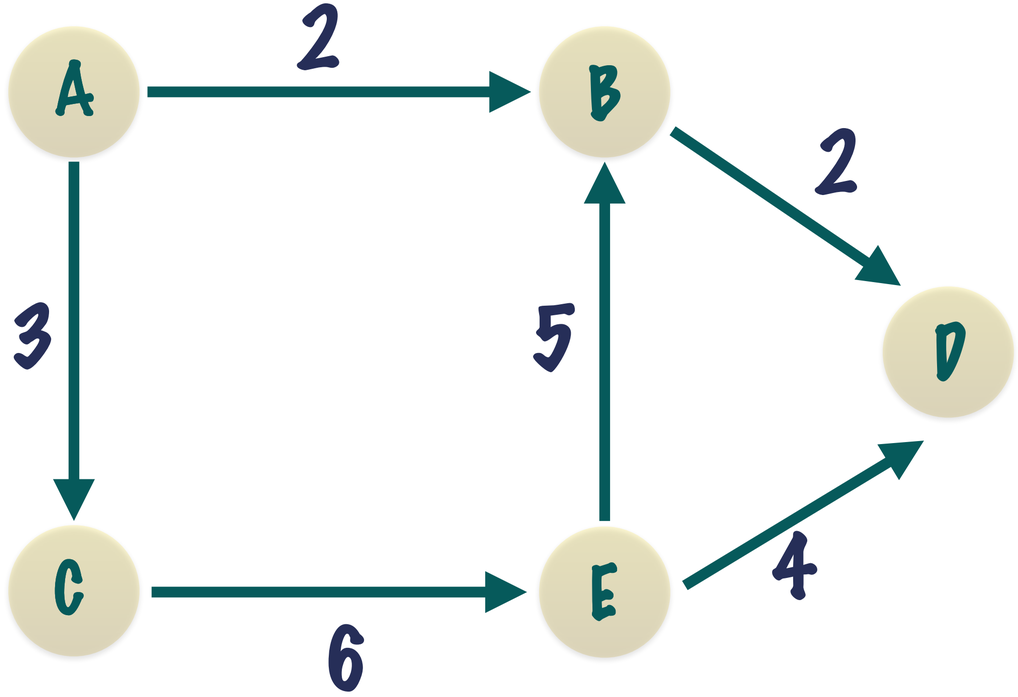
SHORTEST PATH IN WEIGHTED GRAPH



DISTANCE FROM A TO D?

A -> B -> D 2 + 2 = 4

SHORTEST PATH IN WEIGHTED GRAPH



DISTANCE FROM A TO D -
DIFFERENT ROUTE?

A -> C -> E -> D 3 + 6 + 4 = 13

SHORTEST PATH IN WEIGHTED GRAPH

THERE ARE 3 MAJOR DIFFERENCES!

EACH VERTEX HAS NEIGHBOURS

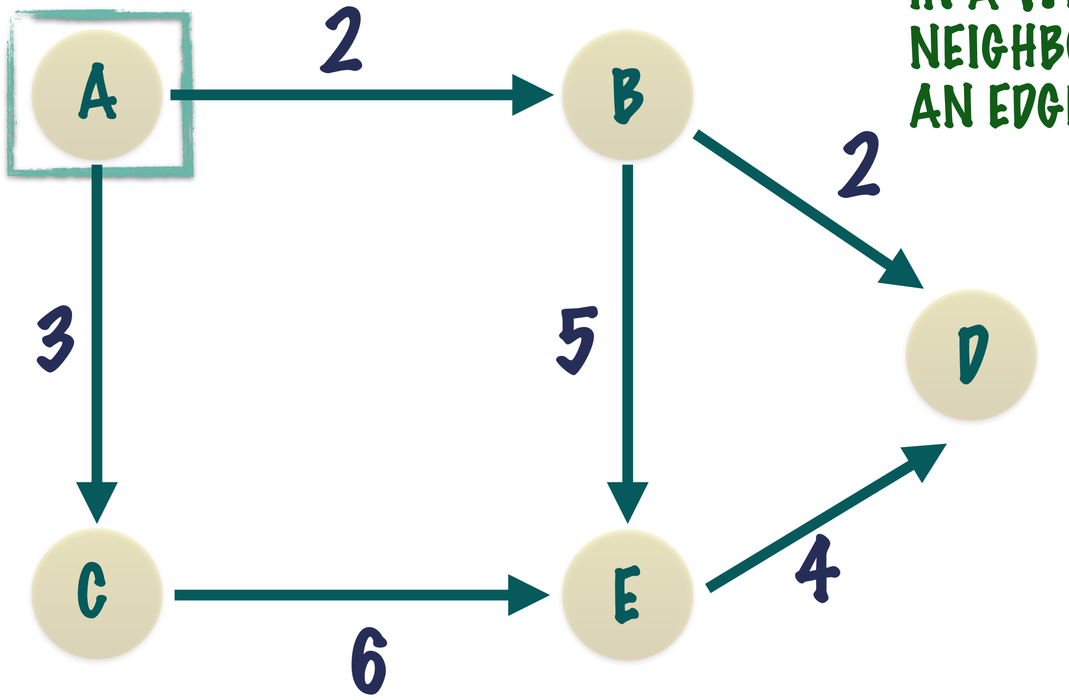
2 IN A WEIGHTED GRAPH - VISIT THE NEIGHBOUR WHICH IS CONNECTED BY AN EDGE WITH THE LOWEST WEIGHT

USE A PRIORITY QUEUE TO IMPLEMENT THIS

TO GET THE NEXT VERTEX IN THE PATH POP THE ELEMENT WITH THE LOWEST WEIGHT!

SHORTEST PATH IN WEIGHTED GRAPH

IN A WEIGHTED GRAPH - VISIT THE NEIGHBOUR WHICH IS CONNECTED BY AN EDGE WITH THE LOWEST WEIGHT



SOURCE: A DESTINATION: D

START AT NODE A

THE NEXT NODE IS?

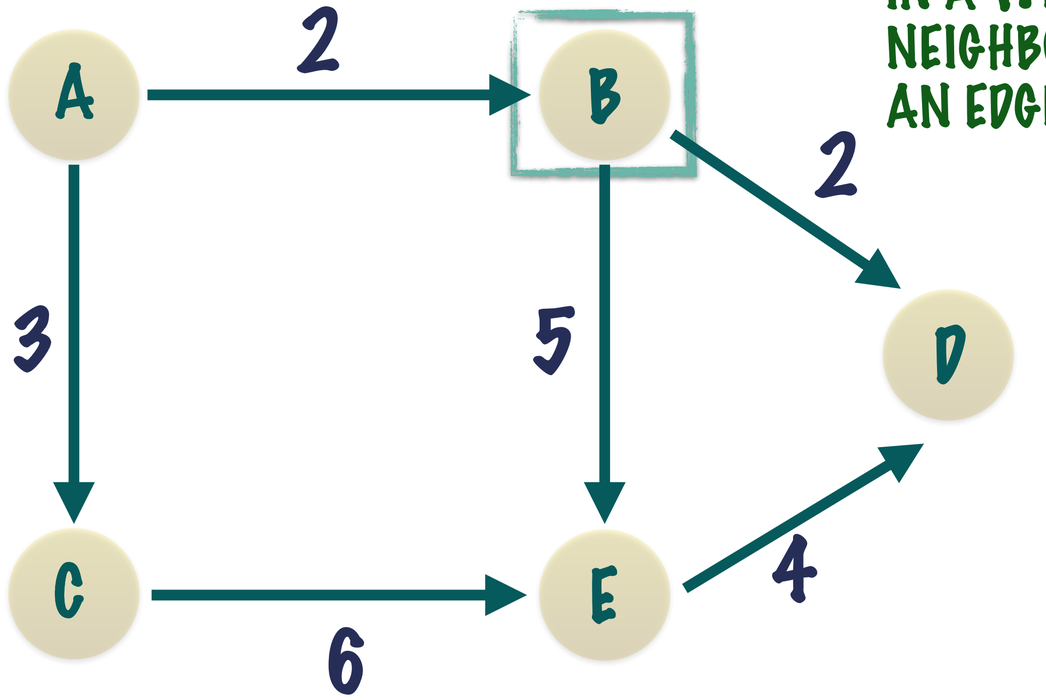
C OR B?

2 < 3

B

SHORTEST PATH IN WEIGHTED GRAPH

IN A WEIGHTED GRAPH - VISIT THE NEIGHBOUR WHICH IS CONNECTED BY AN EDGE WITH THE LOWEST WEIGHT



SOURCE: A DESTINATION: D

NOW AT NODE B

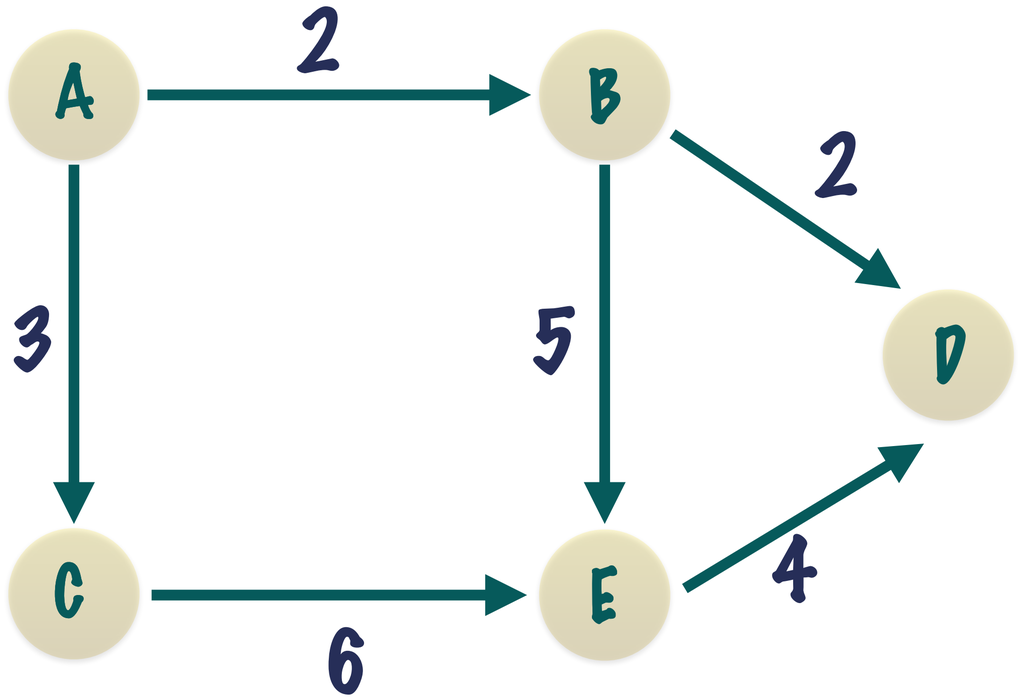
THE NEXT NODE IS?

E OR D?

2 < 5

D

SHORTEST PATH IN WEIGHTED GRAPH



IN A WEIGHTED GRAPH - VISIT THE NEIGHBOUR WHICH IS CONNECTED BY AN EDGE WITH THE LOWEST WEIGHT

THIS IS CALLED A "GREEDY ALGORITHM"!

THIS IS CALLED A "GREEDY ALGORITHM"!

ERR... WHAT IS A GREEDY ALGORITHM?

A GREEDY ALGORITHM BUILDS UP A SOLUTION STEP BY STEP

A EVERY STEP IT ONLY OPTIMIZES FOR THAT PARTICULAR STEP - IT DOES NOT LOOK AT THE OVERALL PROBLEM

IT ONLY CONSIDERS THE MOST OBVIOUS BENEFIT OF CHOOSING THE BEST POSSIBLE NEXT STEP

THIS IS CALLED A "GREEDY ALGORITHM"!

ERR... WHAT IS A GREEDY ALGORITHM?

GREEDY ALGORITHMS OFTEN FAIL TO FIND THE BEST SOLUTION

THEY DO NOT OPERATE ON ALL THE DATA SO THEY MAY NOT SEE THE BIG PICTURE

GREEDY ALGORITHMS ARE USED FOR OPTIMIZATION PROBLEMS

THIS IS CALLED A "GREEDY ALGORITHM"!

ERR... WHAT IS A GREEDY ALGORITHM?

GREEDY ALGORITHMS ARE USED FOR OPTIMIZATION PROBLEMS

MANY ALGORITHMS HAVE GREEDY SOLUTIONS

GREEDY SOLUTIONS ARE ESPECIALLY USEFUL TO FIND APPROXIMATE SOLUTIONS TO VERY HARD PROBLEMS WHICH ARE CLOSE TO IMPOSSIBLE TO SOLVE (TECHNICAL TERM NP HARD)

E.G. THE TRAVELING SALESMAN PROBLEM

SHORTEST PATH IN WEIGHTED GRAPH

THERE ARE 3 MAJOR DIFFERENCES!

3

IT'S POSSIBLE TO VISIT A VERTEX MORE THAN ONCE!

WE CHECK WHETHER NEW DISTANCE (VIA THE ALTERNATIVE ROUTE) IS SMALLER THAN OLD DISTANCE

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

IF NEW DISTANCE < ORIGINAL DISTANCE [NEIGHBOUR]

UPDATE THE DISTANCE TABLE

PUT THE VERTEX IN QUEUE (ONCE AGAIN)!

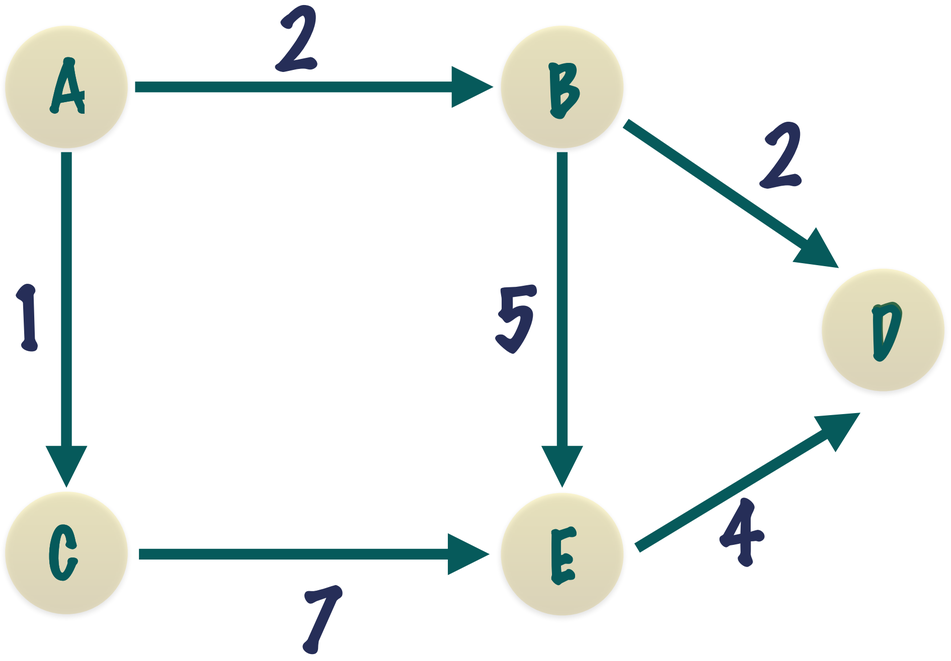
RELAXATION

SHORTEST PATH IN WEIGHTED GRAPH

SOURCE: A DESTINATION: E

LET'S SAY WE HAVE ONE PATH TO E = A->C->E

CURRENT DISTANCE OF E = 1 + 7 = 8



LET'S SAY WE GET TO E AGAIN= A->B->E

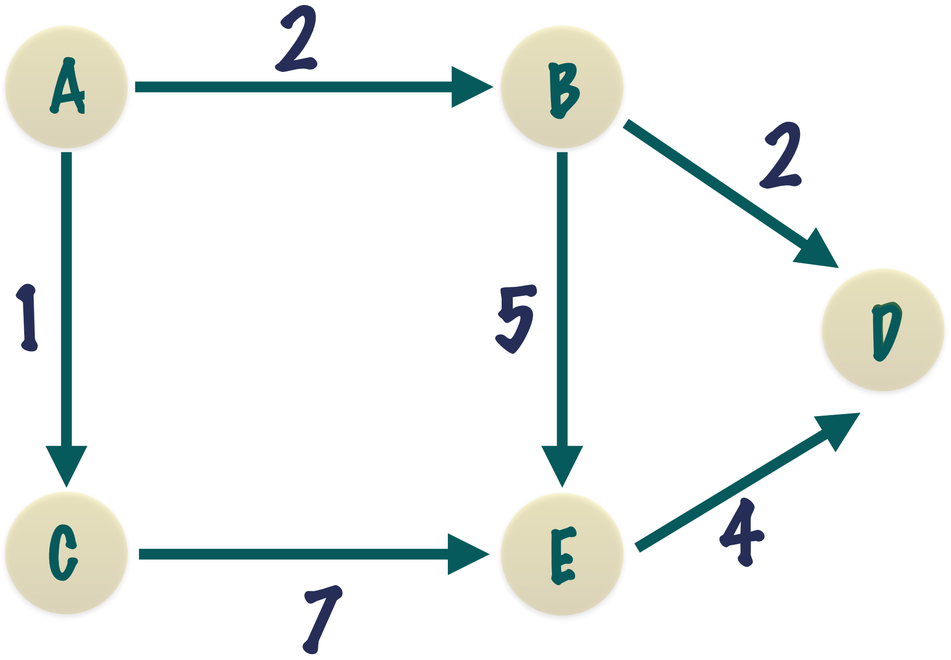
NEW DISTANCE OF E = 2 + 5 = 7

7 < 8

REWRITE THE DISTANCE AND PATH TO E

SHORTEST PATH IN WEIGHTED GRAPH

REWRITE THE DISTANCE AND PATH TO E



THIS PROCESS IS CALLED RELAXATION!

SHORTEST PATH IN WEIGHTED GRAPH

1. THE DISTANCE FROM A NODE NOW HAS TO ACCOUNT FOR THE WEIGHT OF THE EDGES TRAVERSED

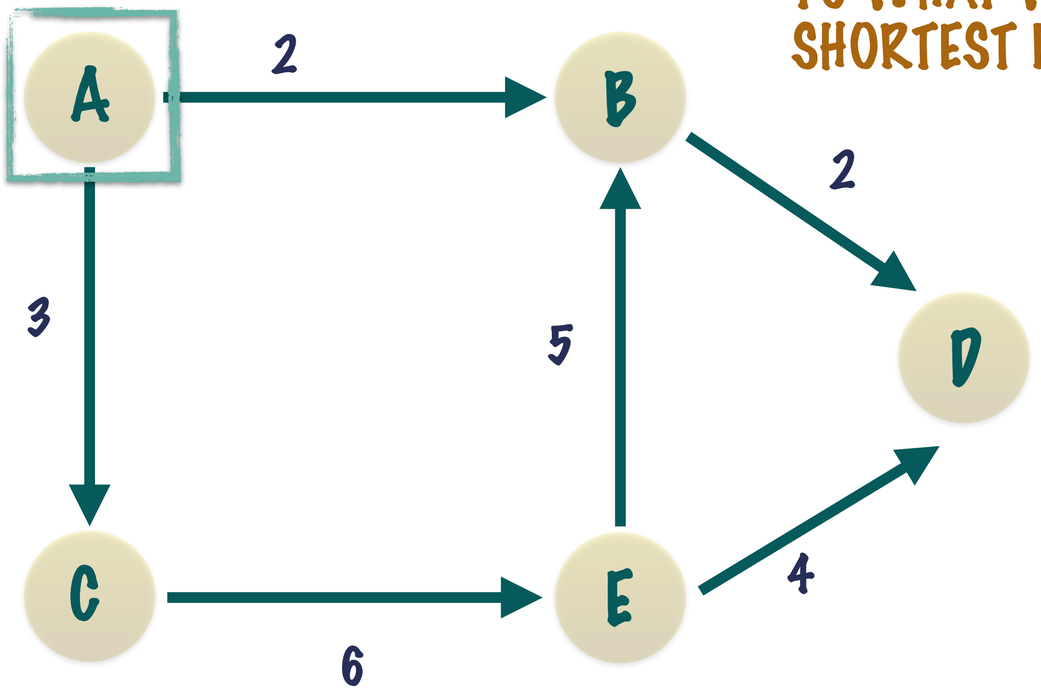
2. IN A WEIGHTED GRAPH - VISIT THE NEIGHBOUR WHICH IS CONNECTED BY AN EDGE WITH THE LOWEST WEIGHT

3. IT'S POSSIBLE TO VISIT A VERTEX MORE THAN ONCE!

   WE CHECK WHETHER NEW DISTANCE (VIA THE ALTERNATIVE ROUTE) IS SMALLER THAN OLD DISTANCE

SHORTEST PATH IN WEIGHTED GRAPH

THE ALGORITHM HERE IS QUITE SIMILAR TO WHAT WE HAVE DISCUSSED BEFORE IN SHORTEST PATH IN UNWEIGHTED GRAPH

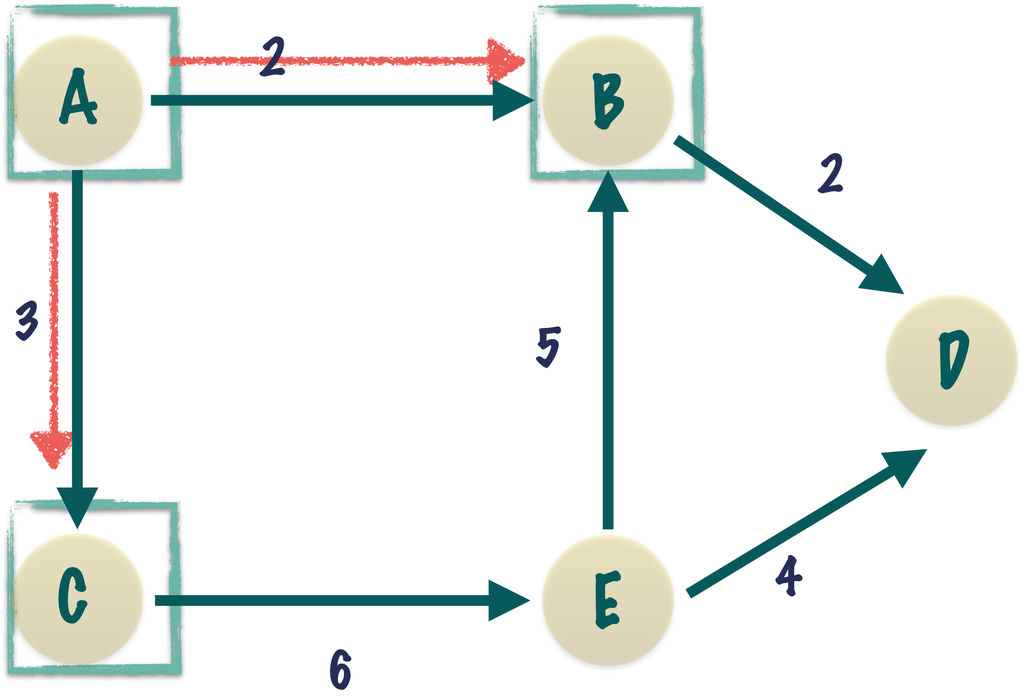


LET 'A' BE THE SOURCE VERTEX!

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

WE INITIALISE OUR DISTANCE TABLE AS FOLLOWS

SHORTEST PATH IN WEIGHTED GRAPH

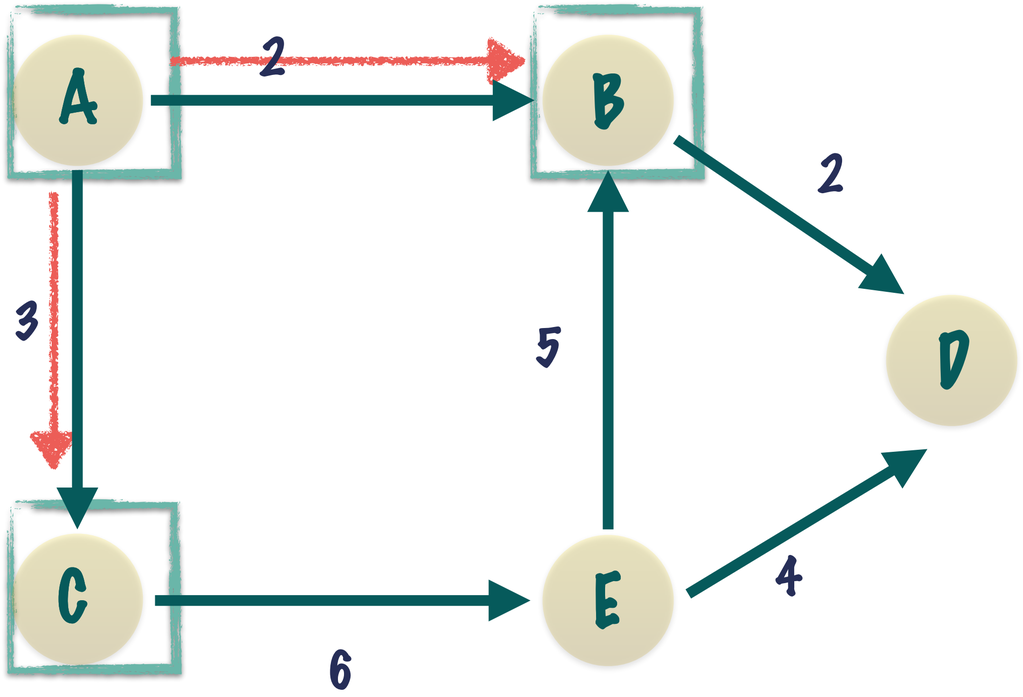


LET 'A' BE THE SOURCE VERTEX!

UPDATE THE DISTANCE TABLE WITH THE DISTANCE TO B AND C FROM A

VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	INF	
E	INF	

SHORTEST PATH IN WEIGHTED GRAPH



B 2

C 3

LET 'A' BE THE SOURCE VERTEX!

UPDATE THE DISTANCE TABLE WITH THE DISTANCE TO B AND C FROM A

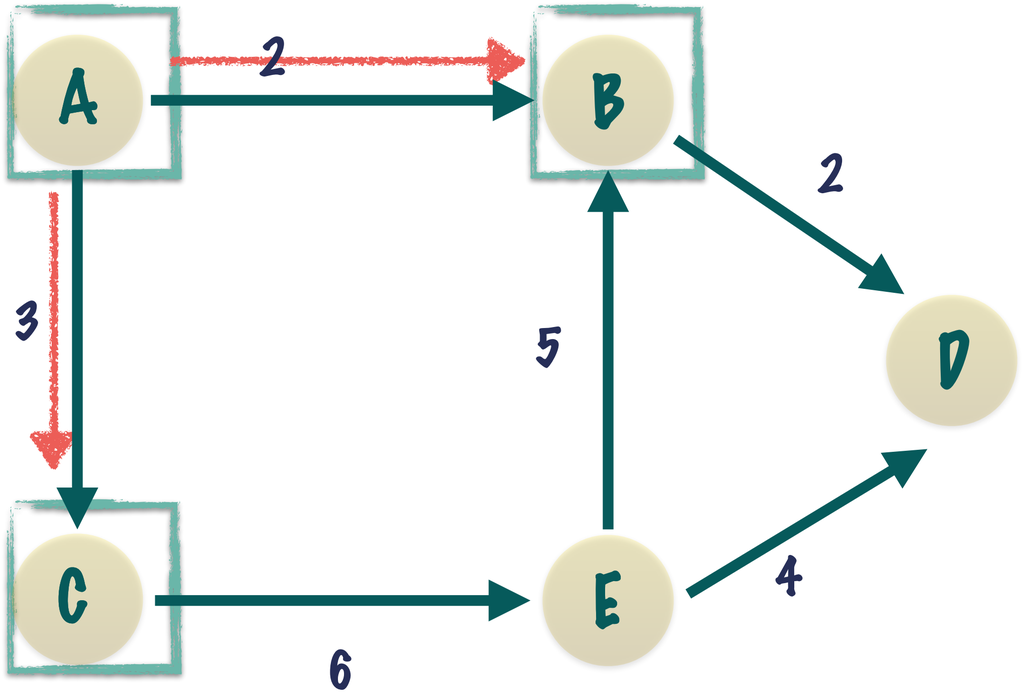
VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	INF	
E	INF	

PUT B AND C ALONG WITH THEIR DISTANCE IN THE PRIORITY QUEUE!

SHORTEST PATH IN WEIGHTED GRAPH

B 2

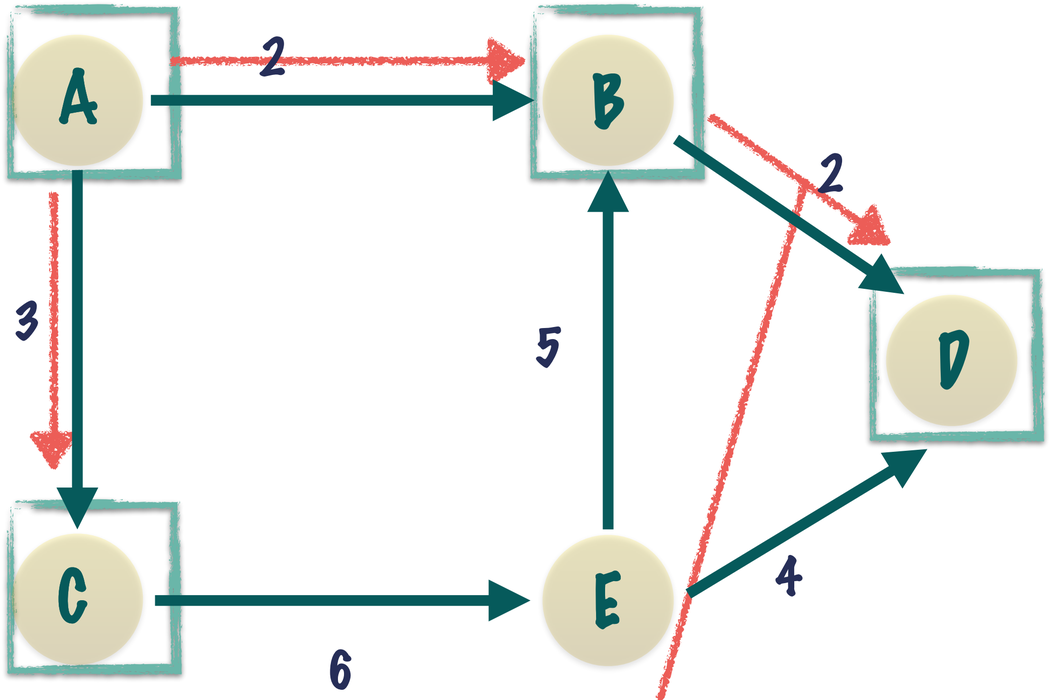
C 3



PROCESS B FROM THE PRIORITY QUEUE AS IT HAS THE SHORTER DISTANCE

VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	INF	
E	INF	

SHORTEST PATH IN WEIGHTED GRAPH



C 3

NEIGHBOURS OF B!

VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	4	B
E	INF	

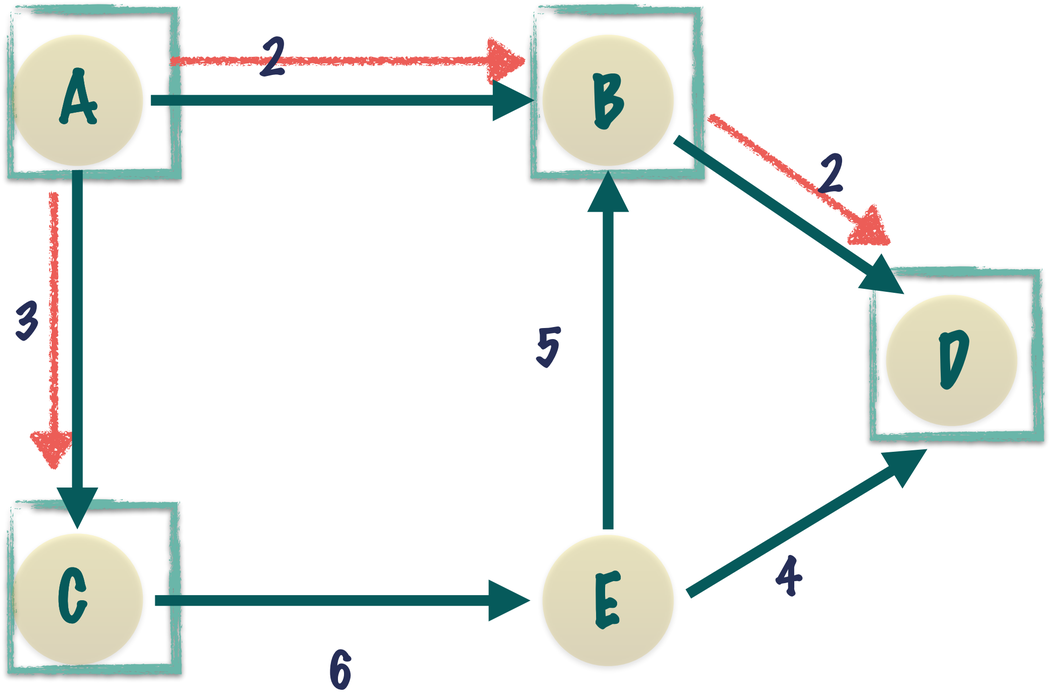
DISTANCE [D] =
DISTANCE [B] + WEIGHT OF EDGE VERTEX[B, D]

DISTANCE [D] = 2 + 2 = 4

SHORTEST PATH IN WEIGHTED GRAPH

C 3

D 4

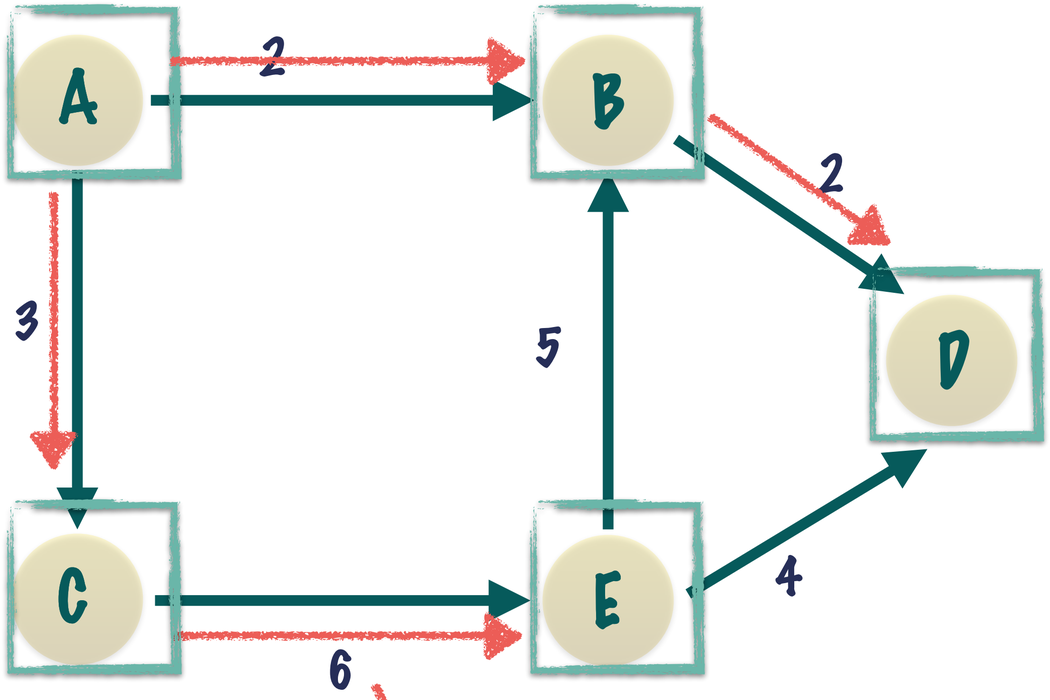


VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	4	B
E	INF	

NOW NEXT PUT D IN PRIORITY QUEUE

PROCESS C - IT HAS THE SHORTEST DISTANCE

SHORTEST PATH IN WEIGHTED GRAPH



CONSIDER NEIGHBOURS OF C!

VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

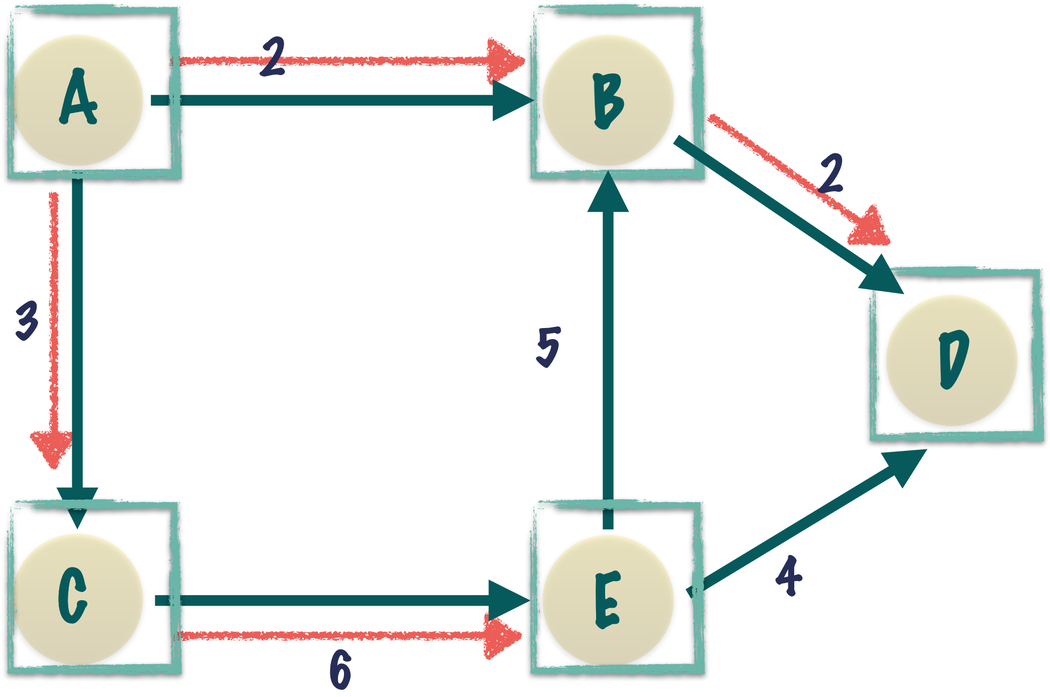
DISTANCE [E] =
DISTANCE [C] + WEIGHT OF EDGE VERTEX[C, E]

DISTANCE [E] = 3 + 6 = 9

SHORTEST PATH IN WEIGHTED GRAPH

D 4

E 9

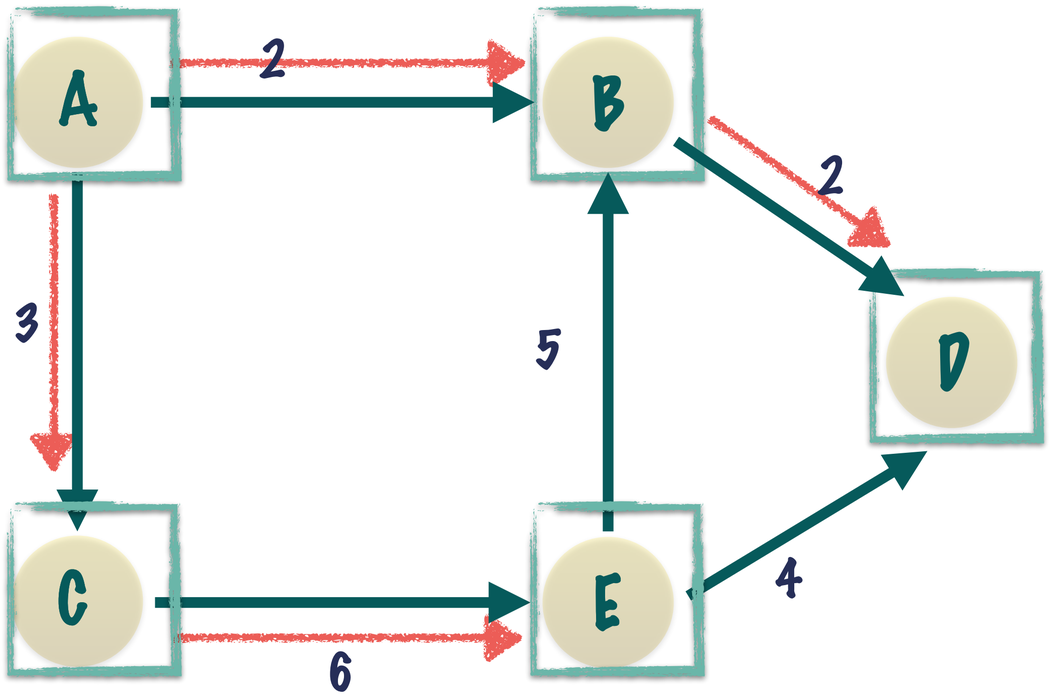


VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

NOW NEXT PUT E IN PRIORITY QUEUE

PROCESS D - IT HAS THE SHORTEST DISTANCE

SHORTEST PATH IN WEIGHTED GRAPH



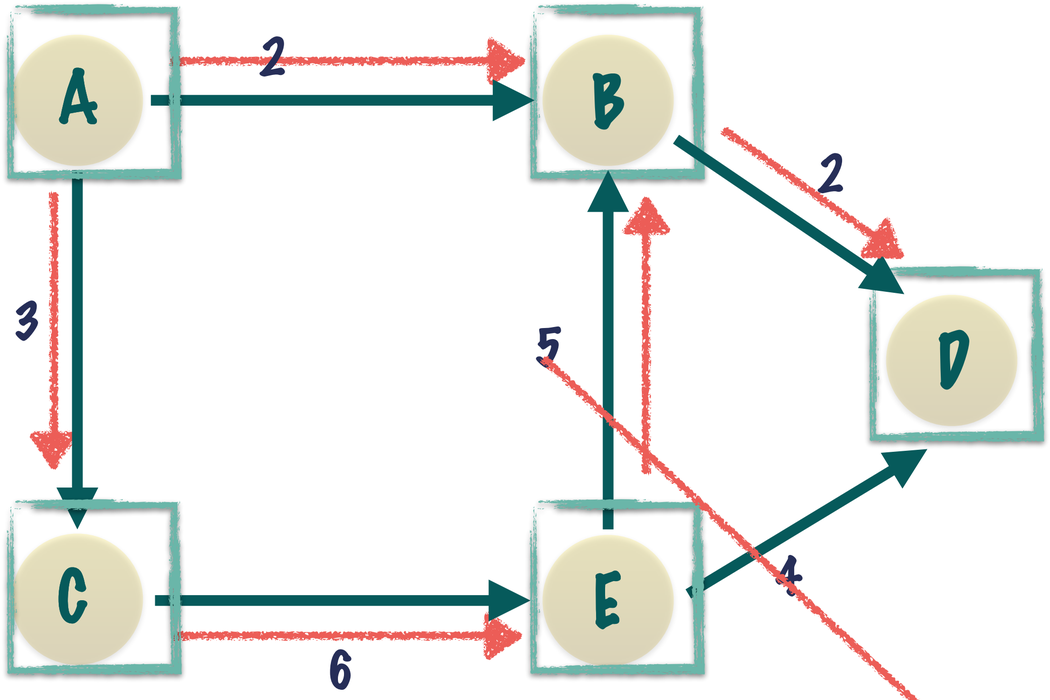
E 9

D HAS NO NEIGBOURS

VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

PROCESS E - IT HAS THE SHORTEST DISTANCE

SHORTEST PATH IN WEIGHTED GRAPH



CONSIDER NEIGHBOURS OF E

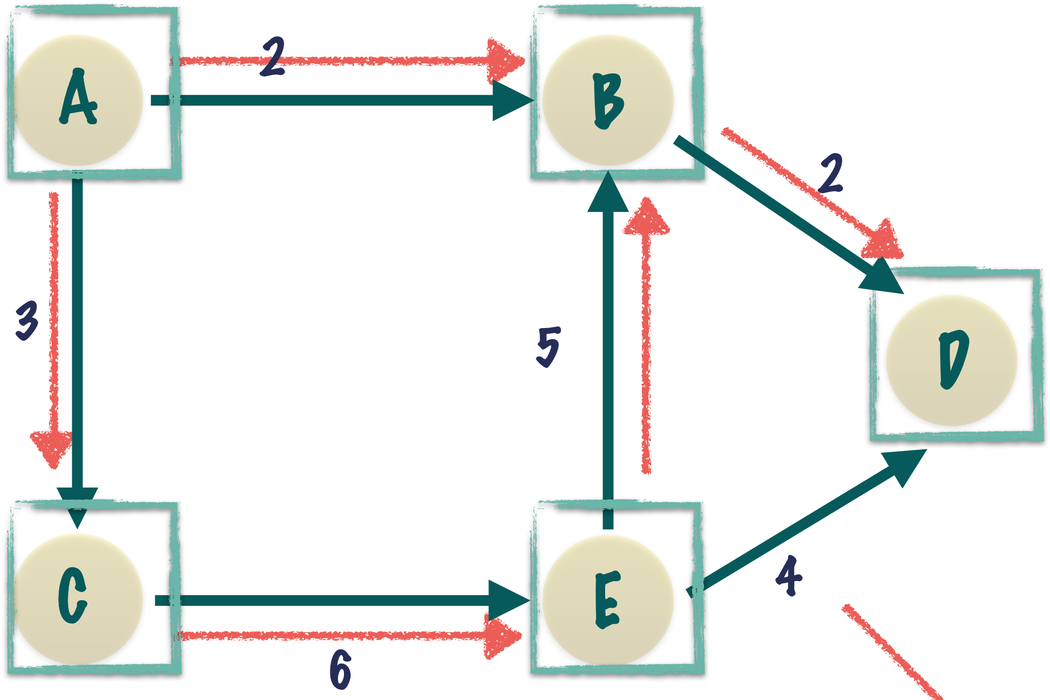
VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

CONSIDER B FIRST

DISTANCE [B] < DISTANCE [E] + WEIGHT OF EDGE VERTEX[E, B]

2 < 9 + 5

SHORTEST PATH IN WEIGHTED GRAPH



CONSIDER NEIGHBOURS OF E

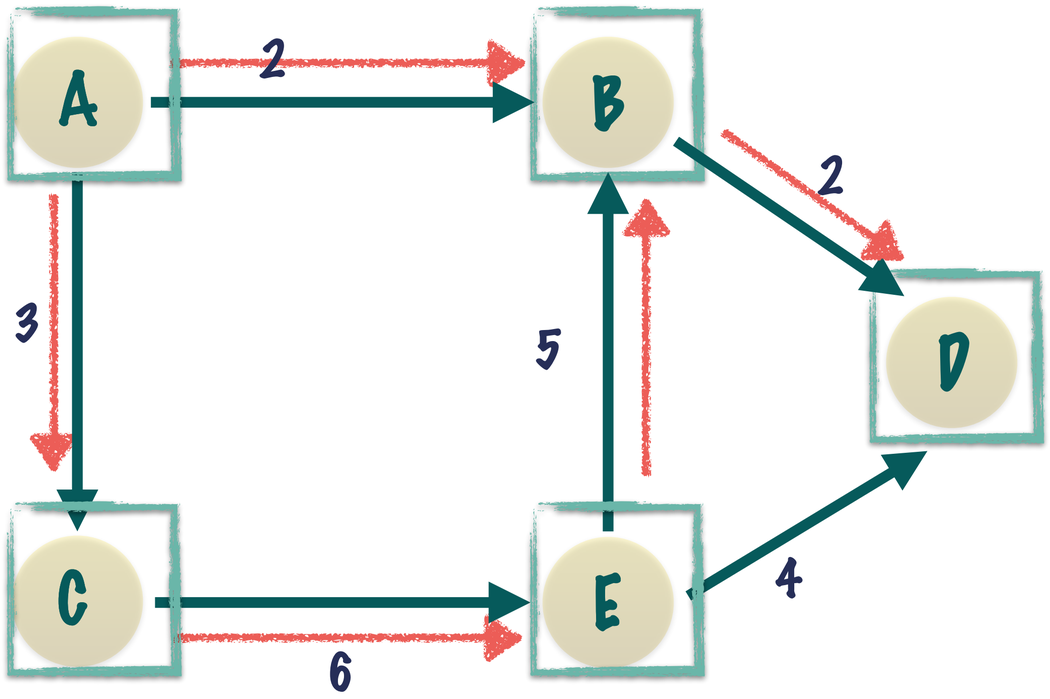
VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

CONSIDER D NEXT

DISTANCE [D] < DISTANCE [E] + WEIGHT OF EDGE VERTEX[E, D]

4 < 9 + 4

SHORTEST PATH IN WEIGHTED GRAPH

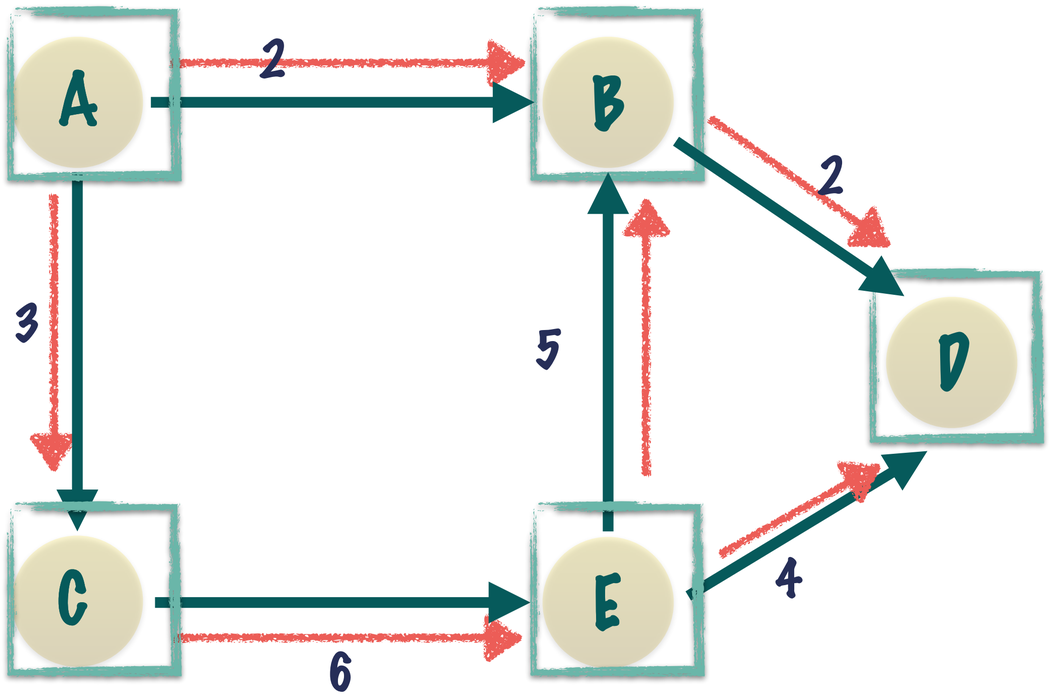


VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

WE ARE VISITING THE VERTICES AGAIN LOOKING FOR A SMALLER PATH FROM OTHER ROUTES!

THE PATHS ARE NOT SHORTER TO EITHER B OR D SO THE DISTANCE TABLE REMAINS THE SAME

SHORTEST PATH IN WEIGHTED GRAPH



VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	2	A
C	3	A
D	4	B
E	9	C

WE USE THE STACK AS BEFORE TO **BACKTRACK** THE PATH FROM THE DESTINATION VERTEX TO THE SOURCE VERTEX

SHORTEST PATH IN WEIGHTED GRAPH

THE ALGORITHM'S EFFICIENCY DEPENDS ON HOW PRIORITY QUEUE IS IMPLEMENTED. IT'S TWO OPERATIONS - UPDATING THE QUEUE AND POPPING OUT FROM THE QUEUE DETERMINES THE RUNNING TIME!

RUNNING TIME IS : O(E LOG V)
[IF BINARY HEAPS ARE USED FOR PRIORITY QUEUE]

RUNNING TIME IS : O(E + V*V)
[IF ARRAY IS USED FOR PRIORITY QUEUE]

DISTANCE TABLE DATA STRUCTURE

```java
/**
 * A class which holds the distance information of any vertex.
 * The distance specified is the distance from the source node
 * and the last vertex is the last vertex just before the current
 * one while traversing from the source node.
 */
public static class DistanceInfo {

    private int distance;
    private int lastVertex;

    public DistanceInfo() {
        // The initial distance to all nodes is assumed
        // infinite.
        distance = Integer.MAX_VALUE;
        lastVertex = -1;
    }

    public int getDistance() {
        return distance;
    }

    public int getLastVertex() {
        return lastVertex;
    }

    public void setDistance(int distance) {
        this.distance = distance;
    }

    public void setLastVertex(int lastVertex) {
        this.lastVertex = lastVertex;
    }
}
```

FOR EVERY VERTEX STORE

1. THE DISTANCE TO THE VERTEX FROM THE SOURCE
2. THE LAST VERTEX IN THE PATH FROM THE SOURCE

NOTE THAT WE SET THE INITIAL DISTANCE OF ALL THE NODES TO INFINITE, AS WE FIND OUR PATH TO THESE NODES WE'LL UPDATE IT WITH THE RIGHT DISTANCE

VERTEX INFO - TRACK THE CURRENT DISTANCE FOR EVERY VERTEX

```java
/**
 * A simple class which holds the vertex id and the weight of
 * the edge that leads to that vertex from its neighbour
 */
public static class VertexInfo {

    private int vertexId;
    private int distance;

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

    public int getVertexId() {
        return vertexId;
    }

    public int getDistance() {
        return distance;
    }
}
```

TRACK EVERY VERTEX ID AND IT'S DISTANCE FROM THE SOURCE

GETTERS FOR THE INFORMATION IN THE VERTEX INFO

BUILD THE DISTANCE TABLE - SETUP

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

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

    distanceTable.get(source).setDistance(0);
    distanceTable.get(source).setLastVertex(source);

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

SET UP A PRIORITY QUEUE WHICH RETURNS NODES IN THE ORDER OF THE SHORTEST DISTANCE FROM THE SOURCE - "THE GREEDY SOLUTION"

ADD A DISTANCE TABLE ENTRY FOR EACH NODE IN THE GRAPH

ADD THE SOURCE TO THE PRIORITY QUEUE AND SET UP A MAPPING TO THE VERTEX INFO FOR EVERY VERTEX IN THE QUEUE

BUILD THE DISTANCE TABLE - PROCESS

```
while (!queue.isEmpty()) {
    VertexInfo vertexInfo = queue.poll();
    int currentVertex = vertexInfo.getVertexId();

    for (Integer neighbour : graph.getAdjacentVertices(currentVertex)) {
        // Get the new distance, account for the weighted edge.
        int distance = distanceTable.get(currentVertex).getDistance()
                + graph.getWeightedEdge(currentVertex, neighbour);

        // If we find a new shortest path to the neighbour update
        // the distance and the last vertex.
        if (distanceTable.get(neighbour).getDistance() > distance) {
            distanceTable.get(neighbour).setDistance(distance);
            distanceTable.get(neighbour).setLastVertex(currentVertex);

            // We've found a new short path to the neighbour so remove
            // the old node from the priority queue.
            VertexInfo neighbourVertexInfo = vertexInfoMap.get(neighbour);
            if (neighbourVertexInfo != null) {
                queue.remove(neighbourVertexInfo);
            }

            // Add the neighbour back with a new updated distance.
            neighbourVertexInfo = new VertexInfo(neighbour, distance);
            queue.add(neighbourVertexInfo);
            vertexInfoMap.put(neighbour, neighbourVertexInfo);
        }
    }
}
```

ACCESS THE PRIORITY QUEUE TO FIND THE CLOSEST VERTEX - FOLLOWING THE GREEDY ALGORITHM

THE DISTANCE TO THE ADJACENT VERTEX IS NOW THE DISTANCE OF THE CURRENT VERTEX + WEIGHT OF THE EDGE

IF THE NEW PATH TO THE NEIGHBOR IS SHORTER UPDATE THE CURRENT DISTANCE AND LAST VERTEX TO THE NEIGHBOR

REMOVE THE OUTDATED VERTEX PATH INFORMATION FROM THE QUEUE - WE HAVE A NEW SHORTER ROUTE

ADD THE UPDATED NEIGHBOR INFORMATION TO THE QUEUE SO WE CAN GET THE NEXT CLOSEST NODE

SHORTEST PATH

```
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