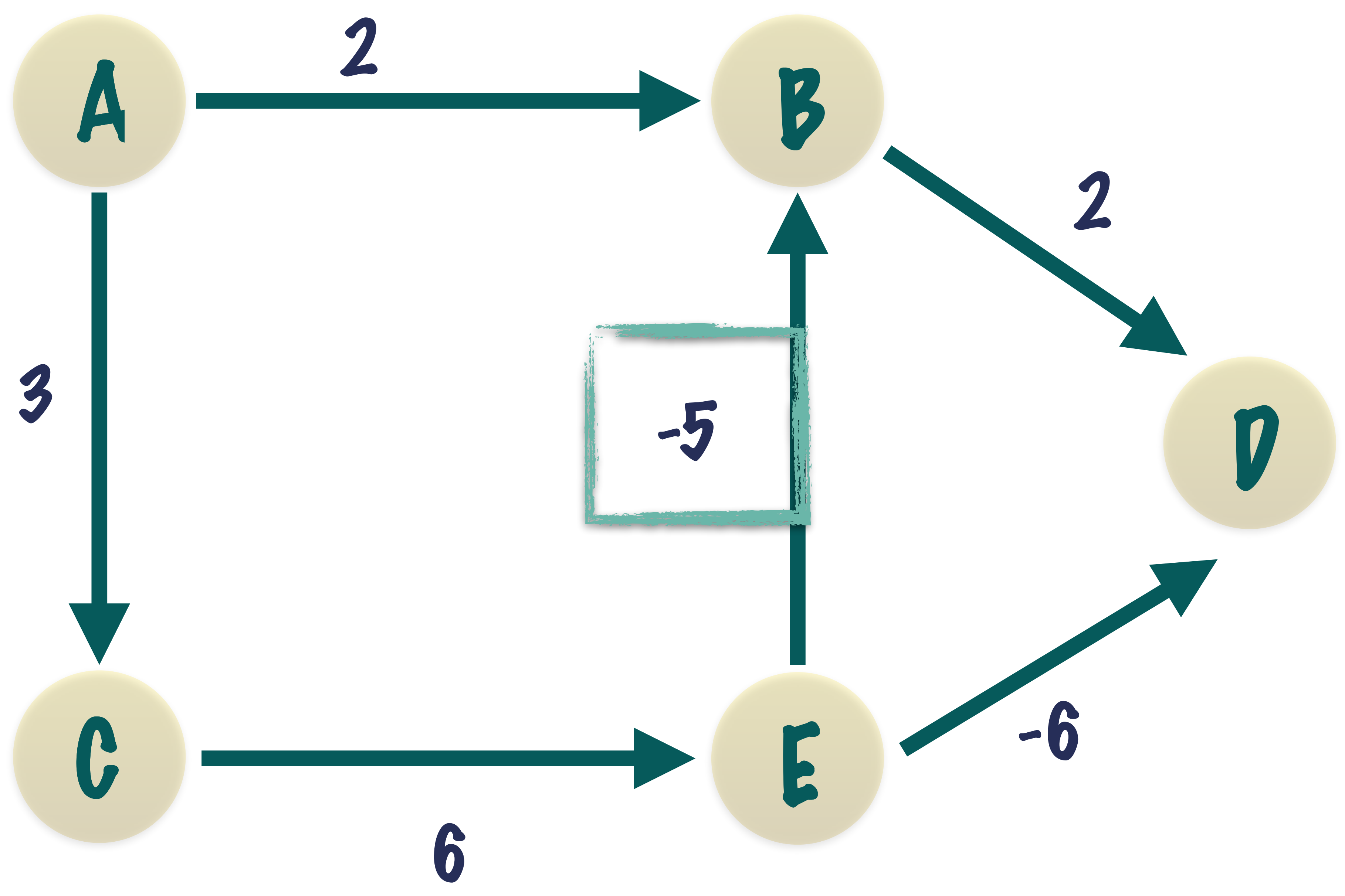
THE GRAPH

SHORTEST PATH ALGORITHMS

SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH



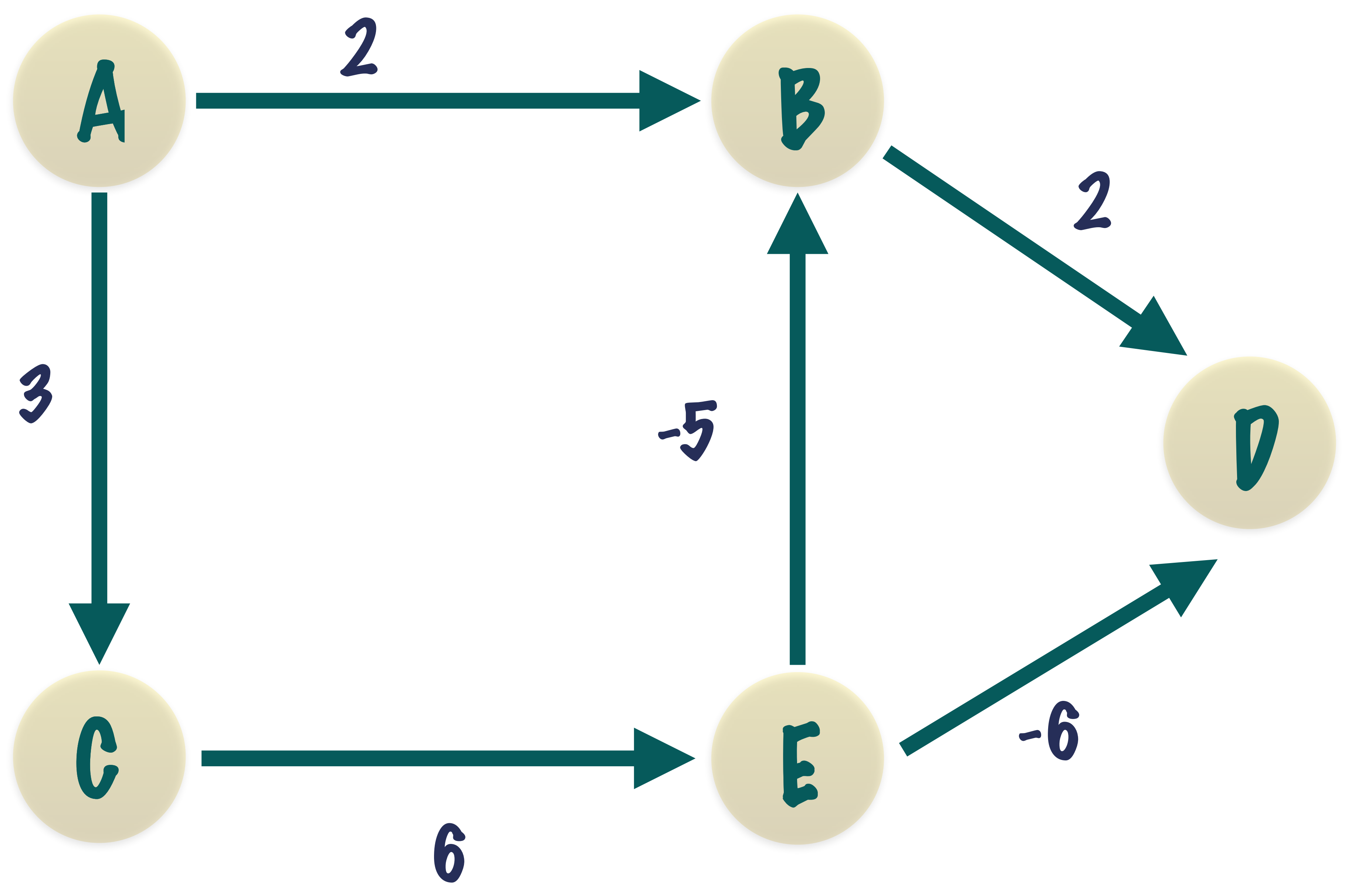
GRAPHS CAN HAVE NEGATIVE WEIGHTS ON THE EDGES AS WELL

FINDING THE SHORTEST PATH IN GRAPHS WITH NEGATIVE WEIGHTS IS A DIFFERENT ALGORITHM

THE BELLMAN-FORD ALGORITHM

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

THE BELLMAN-FORD ALGORITHM



THE ALGORITHM IS COMBINATION OF DIJKSTRA'S AND SHORTEST UNWEIGHTED PATH ALGORITHM!

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

WE CONTINUE TO USE THE DISTANCE TABLE TO STORE INFORMATION

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

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

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

HOWEVER THIS ALGORITHM IS NOT "GREEDY" AND WE DO NOT NEED A PRIORITY QUEUE TO TRAVERSE NODES

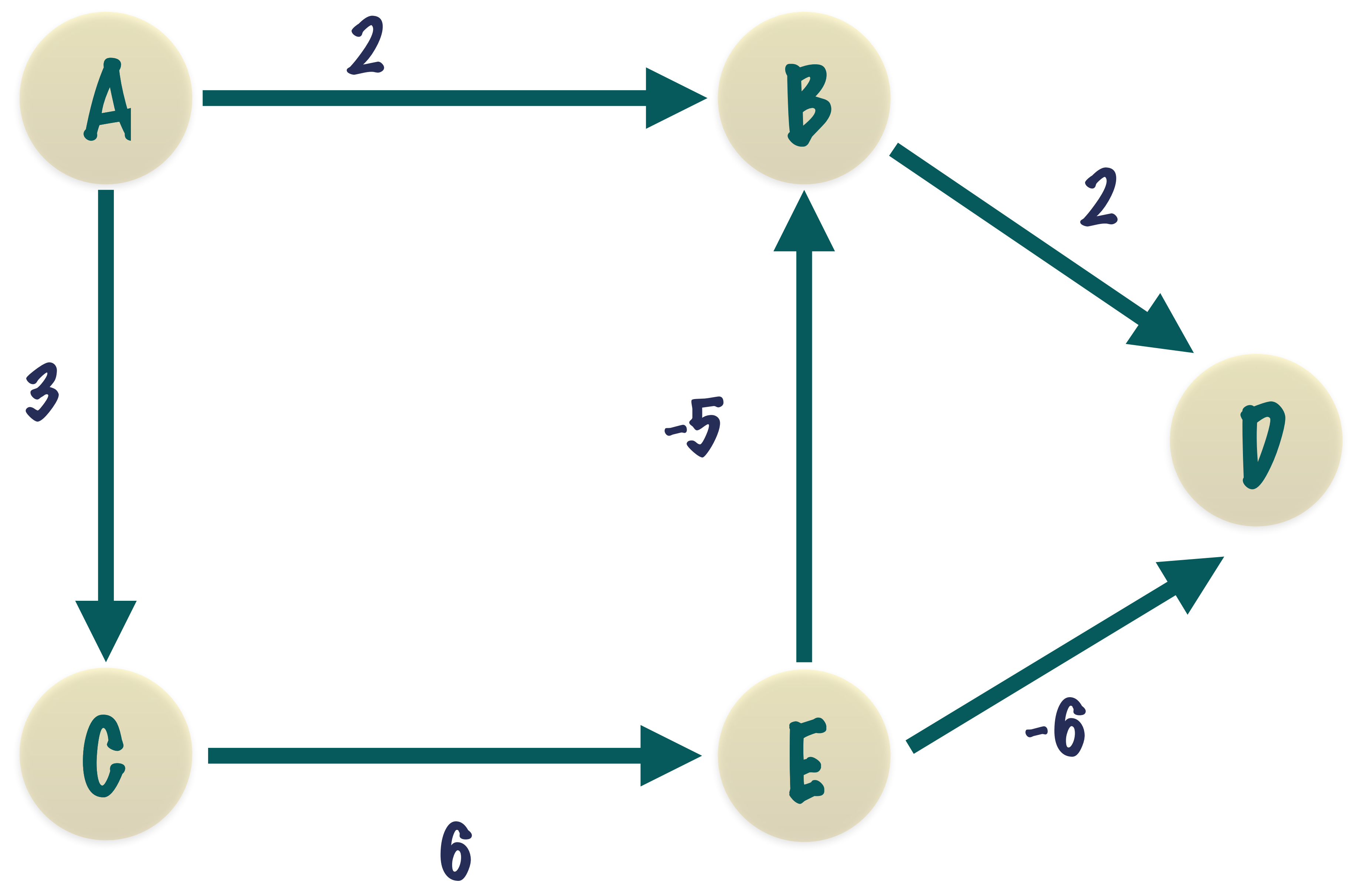
GREEDY ALGORITHMS WORK WELL WHEN ALL EDGES ARE POSITIVE ALLOWING YOU TO CHOOSE THE MOST OPTIMAL STEP AND EVERY VERTEX

WITH NEGATIVE DISTANCES THE SUM OF DISTANCES ARE NOT MONOTONICALLY INCREASING - A SINGLE NEGATIVE PATH CAN CHANGE THE SOLUTION COMPLETELY!

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

WITH NEGATIVE DISTANCES THE SUM OF DISTANCES ARE NOT MONOTONICALLY INCREASING - A SINGLE NEGATIVE PATH CAN CHANGE THE SOLUTION COMPLETELY!

WITH NEGATIVE WEIGHTS IT'S POSSIBLE THAT A PATH WHICH SEEMS SUB-OPTIMAL CURRENTLY LEADS TO A NEGATIVE EDGE WHICH MAKES IT THE BEST PATH!

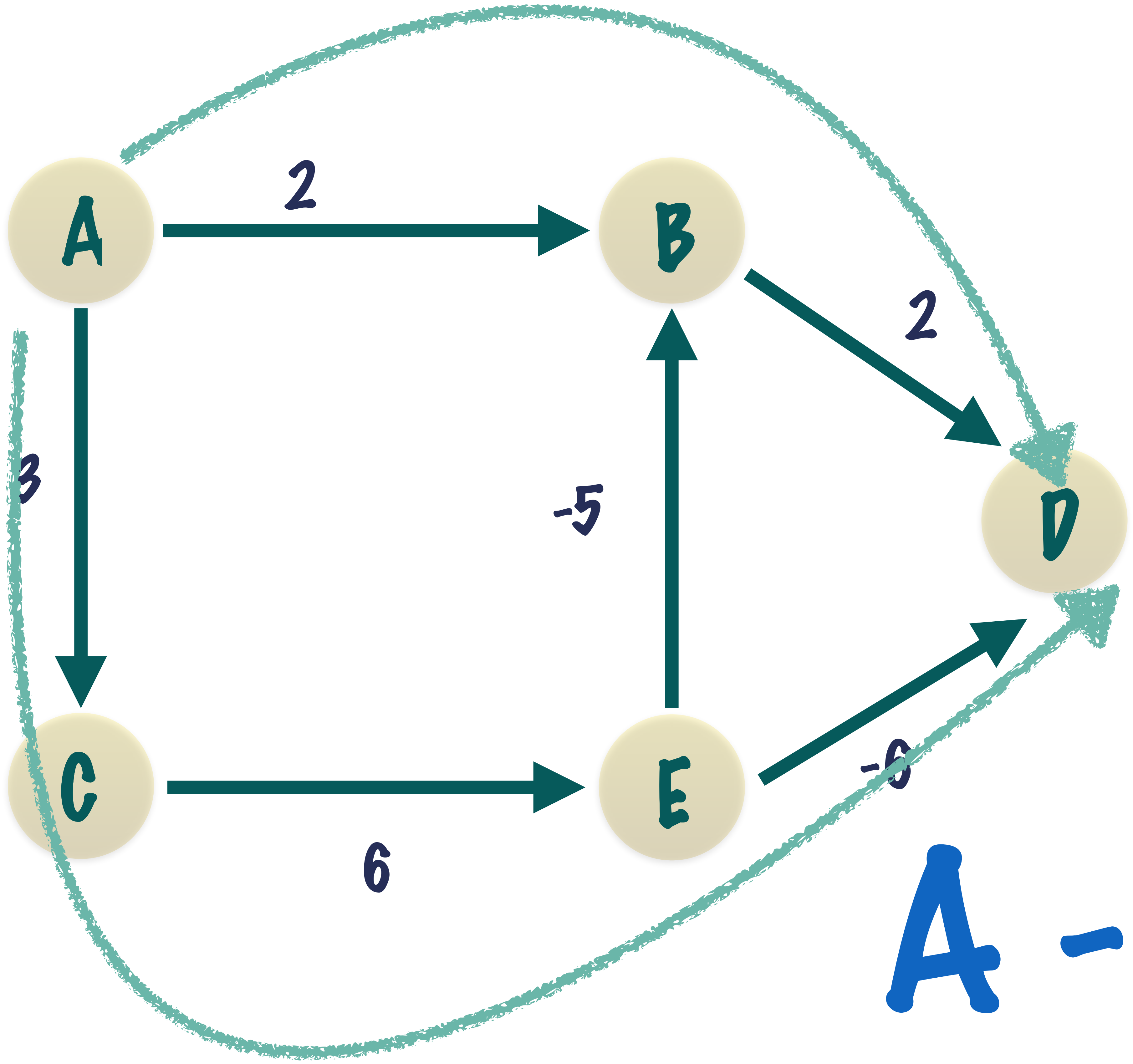


LET'S TAKE AN EXAMPLE

A -> D

THE GRAPH

SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH



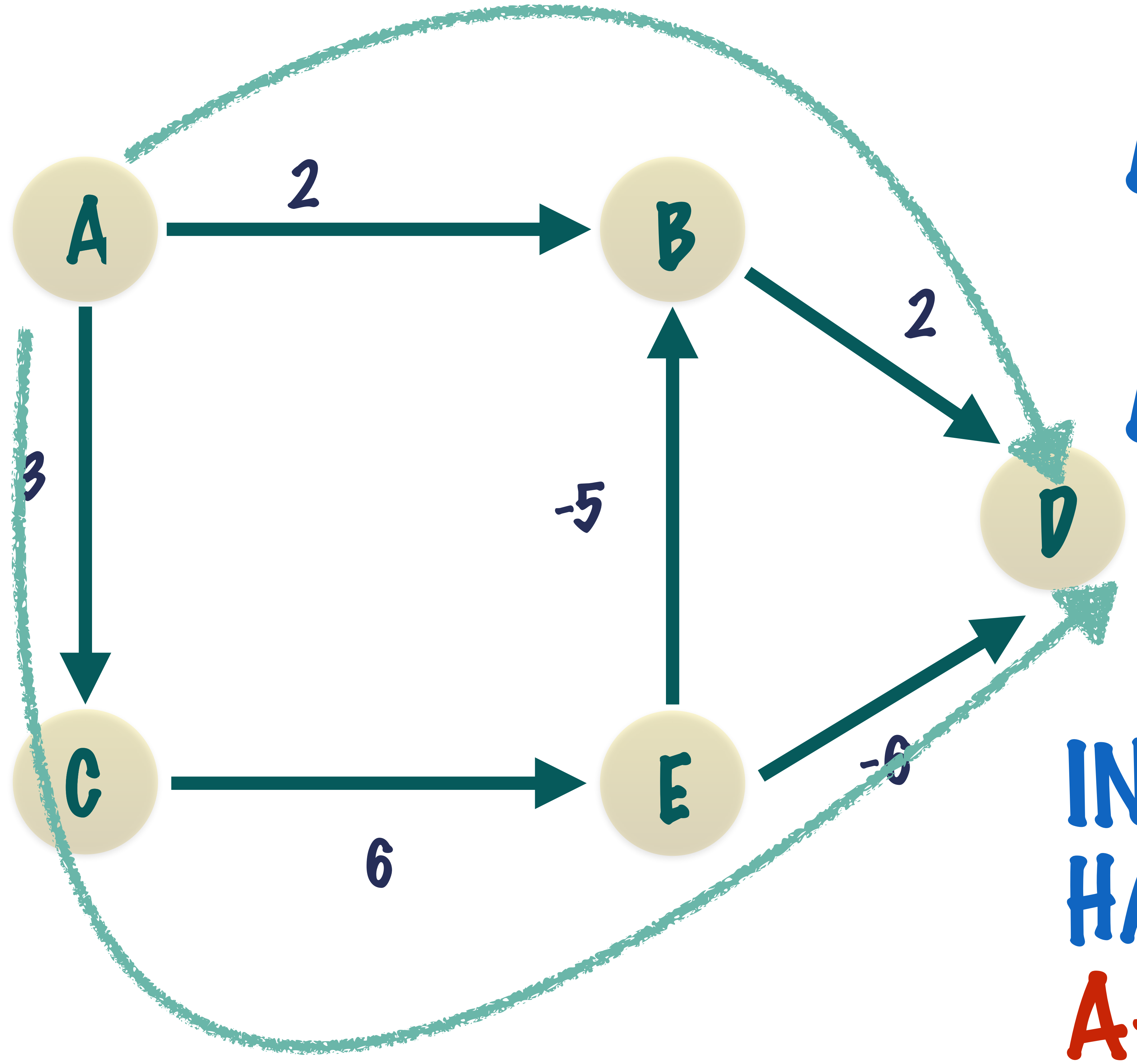
LET'S TAKE AN EXAMPLE

A -> D

TWO PATHS

	DISTANCE
A -> B -> D	4
A -> C -> E -> D	3

THE GRAPH
SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH



	DISTANCE
A -> B -> D	4
A -> C -> E -> D	3

IN THE GREEDY ALGORITHM WE WOULD HAVE CHOSEN THE A->B->D PATH OVER THE A->C->E->D BUT THAT WOULD HAVE BEEN SUB-OPTIMAL AS LATER ON WE WOULD DISCOVER THE SMALLER WEIGHT PATH.

DIJKSTRA'S ALGORITHM USES A PRIORITY QUEUE TO GREEDILY SELECT THE CLOSEST VERTEX AND VISITS ALL IT'S ADJACENT EDGES - THE PROCESS CALLED RELAXATION

BY CONTRAST, THE BELLMAN-FORD ALGORITHM PROCESS ALL THE EDGES I.E. RELAXES ALL THE EDGES

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

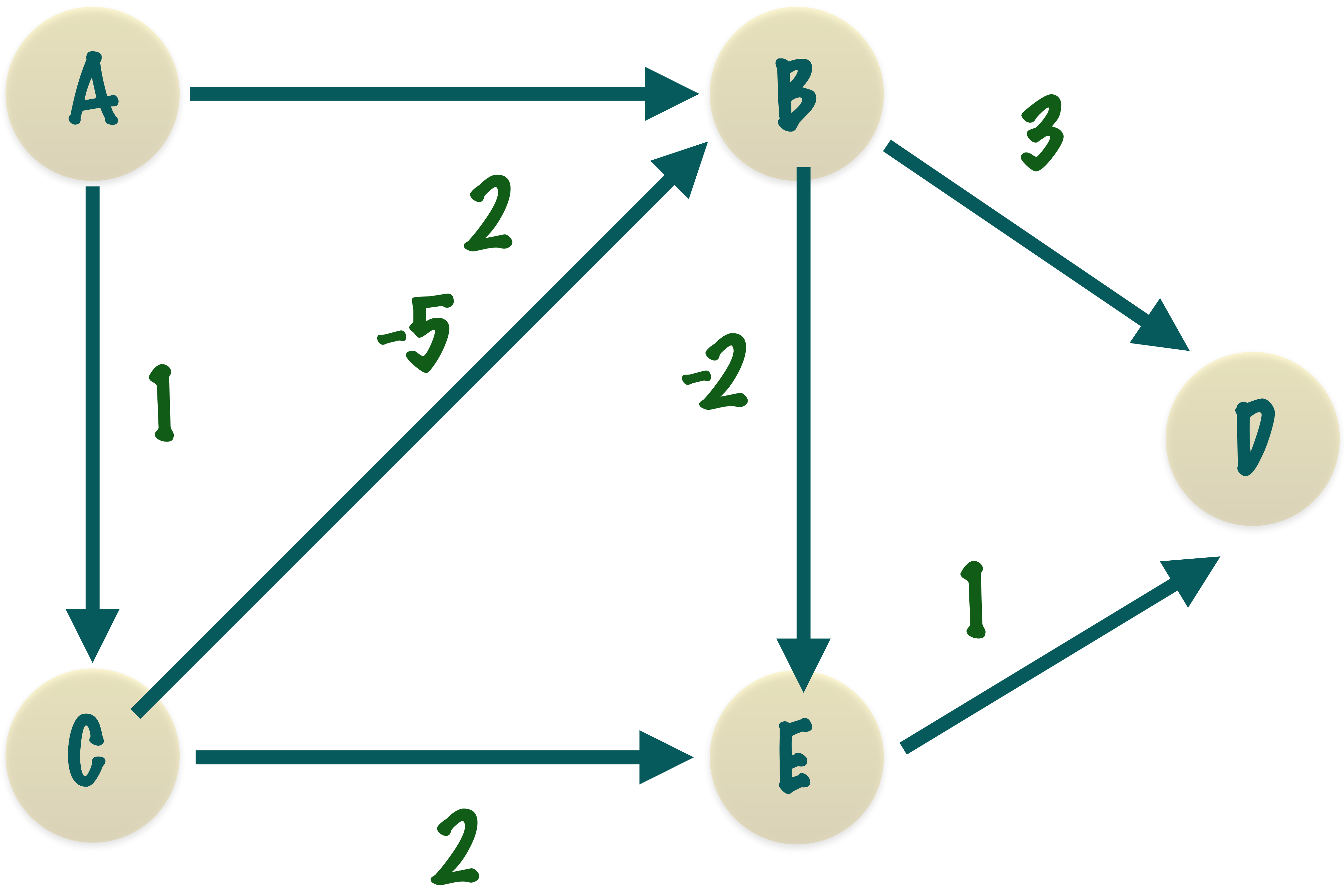
BY CONTRAST, THE BELLMAN-FORD ALGORITHM PROCESS **ALL** THE EDGES I.E. RELAXES ALL THE EDGES

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

WE DO THIS COMPUTATION FOR EVERY EDGE AND DO IT (V-1) TIMES FOR EACH EDGE!

THE GRAPH

SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH



WE'LL START FROM VERTEX A AT RANDOM

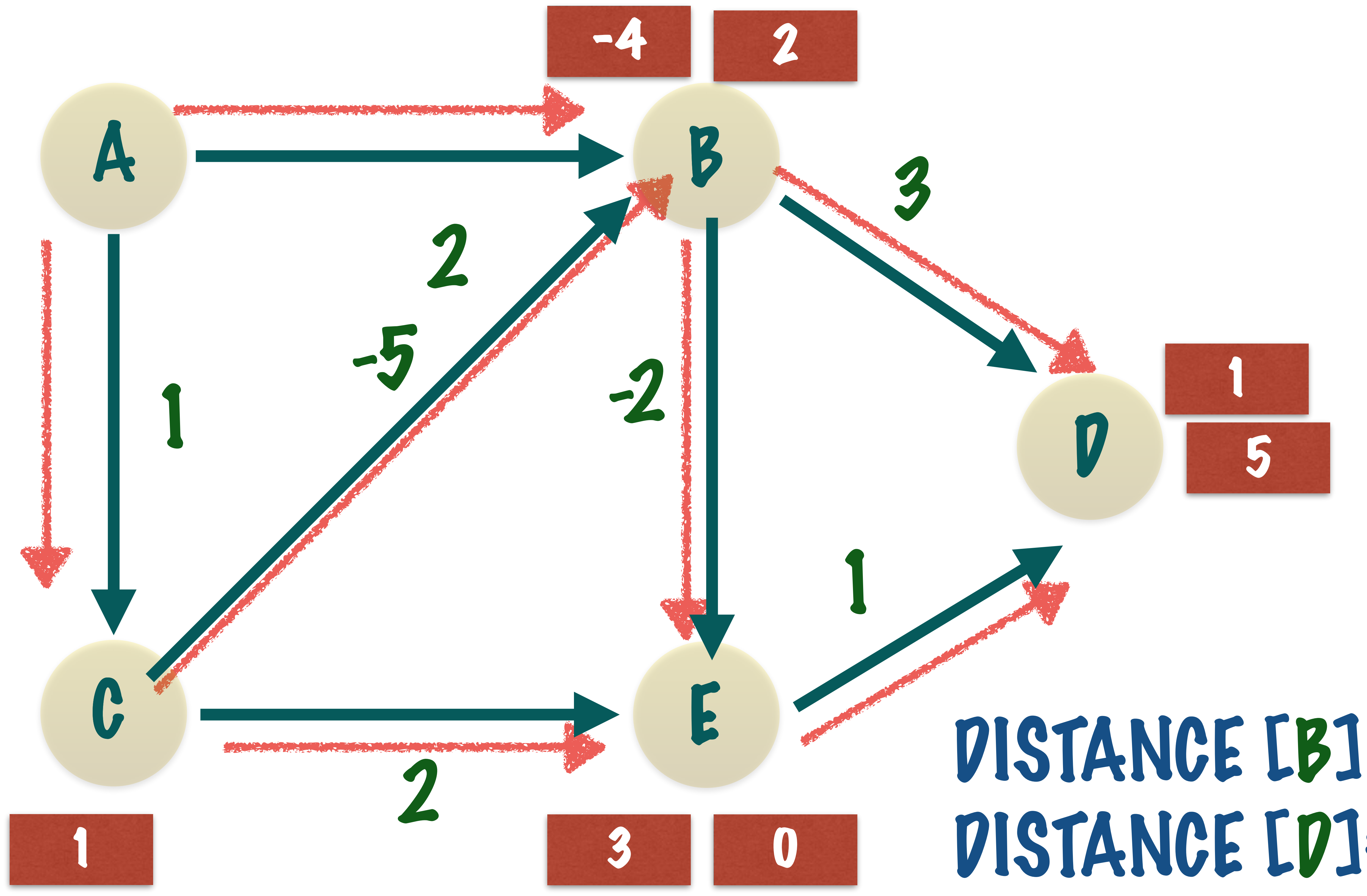
INITIALIZE THE DISTANCE TABLE

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

SINCE THIS ALGORITHM COVERS ALL THE EDGES V-1 TIMES, WE CAN START WITH ANY VERTEX, AND THE RESULT WILL BE THE SAME

THE GRAPH

SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH



THE FIRST ITERATION

DISTANCE [B] = DISTANCE [A] + WEIGHT OF EDGE [A, B]

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

DISTANCE [E]= DISTANCE [B] + WEIGHT OF EDGE [B, E]

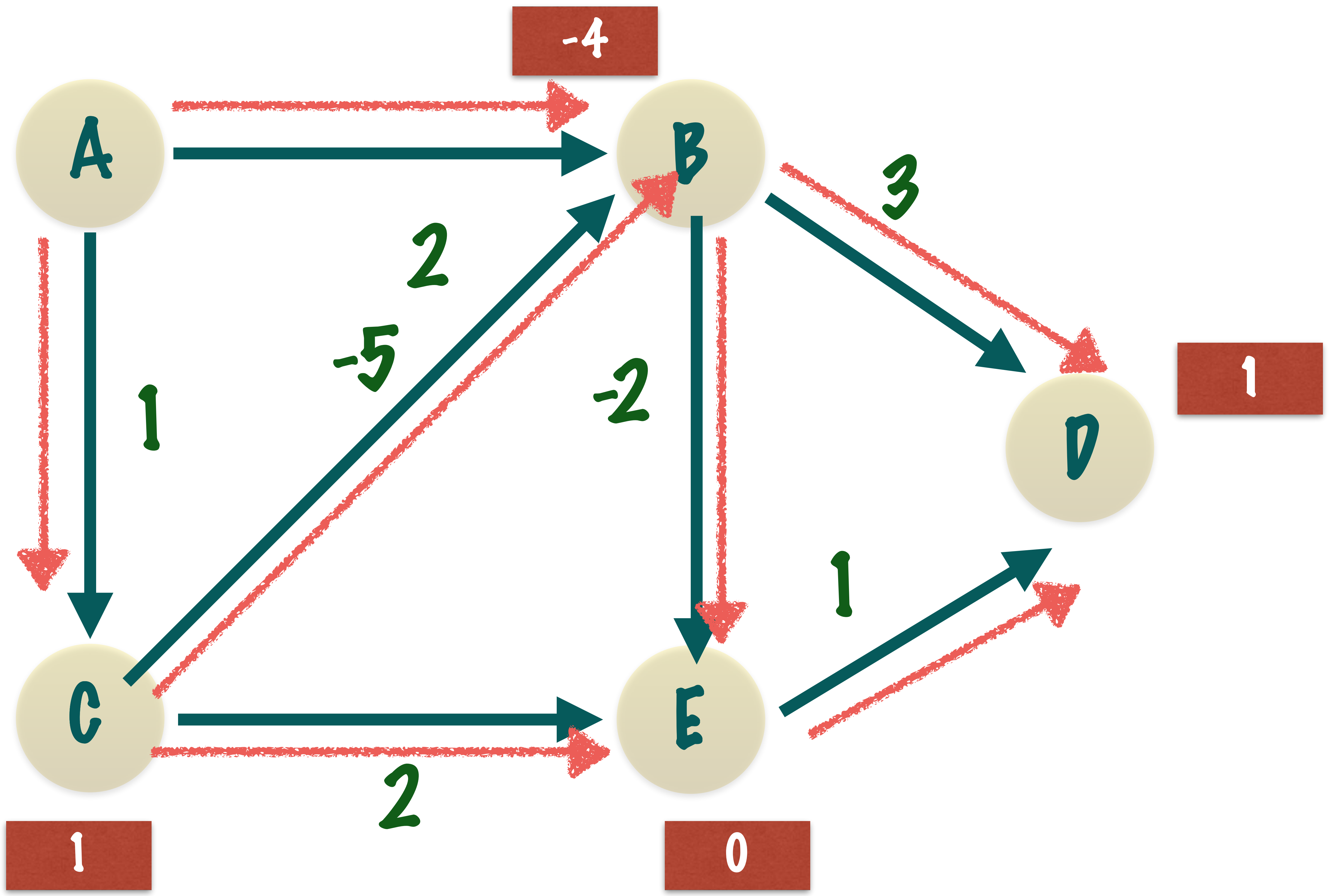
DISTANCE [D]= DISTANCE [E] + WEIGHT OF EDGE [E,D]

DISTANCE [C] = DISTANCE [A] + WEIGHT OF EDGE [A, C]

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

DISTANCE [D]= DISTANCE [C] + WEIGHT OF EDGE [C, D]

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

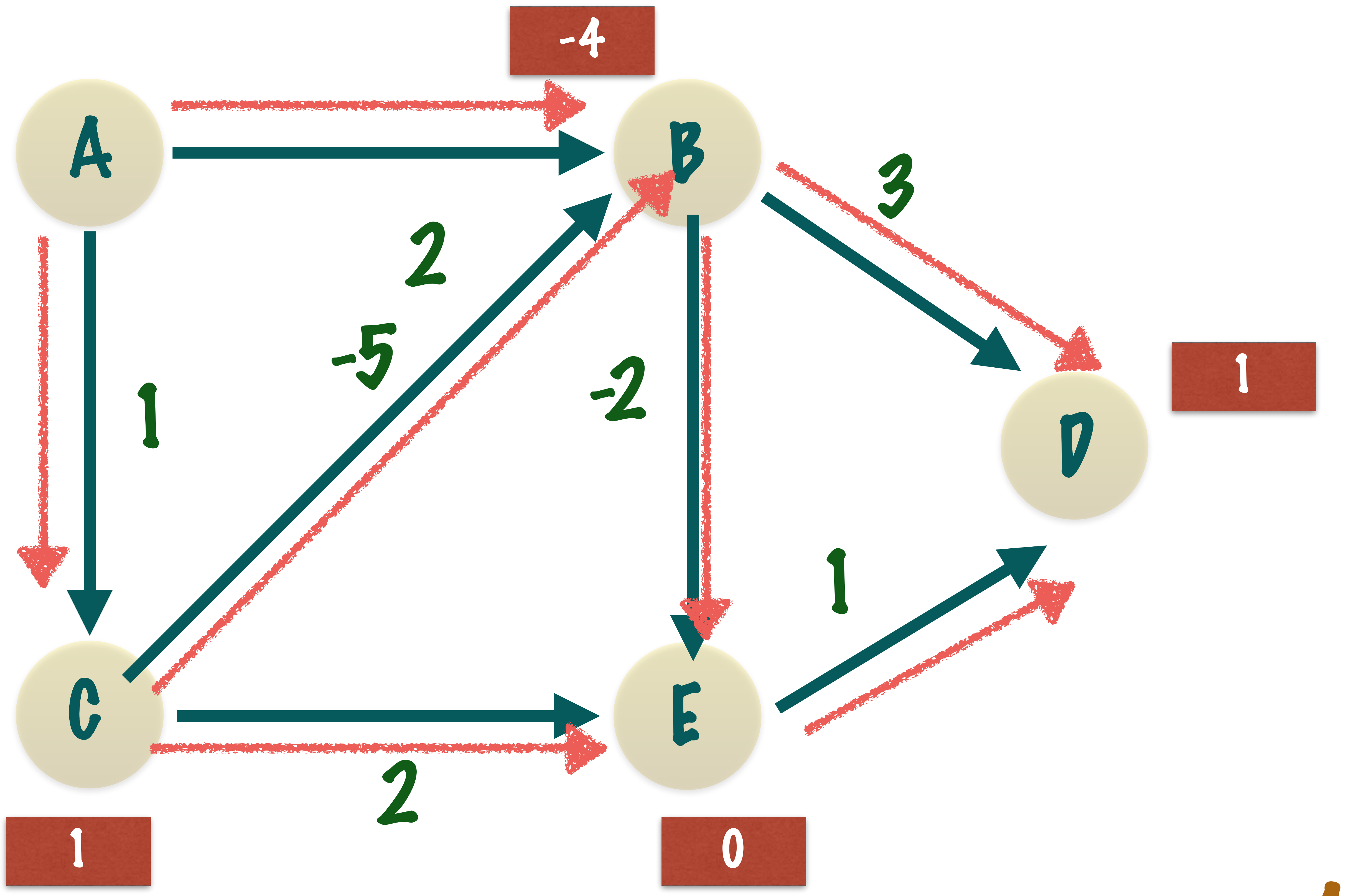


THE FIRST ITERATION

DISTANCE [B] = DISTANCE [A] + WEIGHT OF EDGE [A, B]
DISTANCE [D]= DISTANCE [B] + WEIGHT OF EDGE [B, D]
DISTANCE [E]= DISTANCE [B] + WEIGHT OF EDGE [B, E]
DISTANCE [D]= DISTANCE [E] + WEIGHT OF EDGE [E,D]
DISTANCE [C] = DISTANCE [A] + WEIGHT OF EDGE [A, C]
DISTANCE [E]= DISTANCE [C] + WEIGHT OF EDGE [C, E]
DISTANCE [D]= DISTANCE [C] + WEIGHT OF EDGE [C, D]

VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	-4	C
C	1	B
D	1	E
E	0	B

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

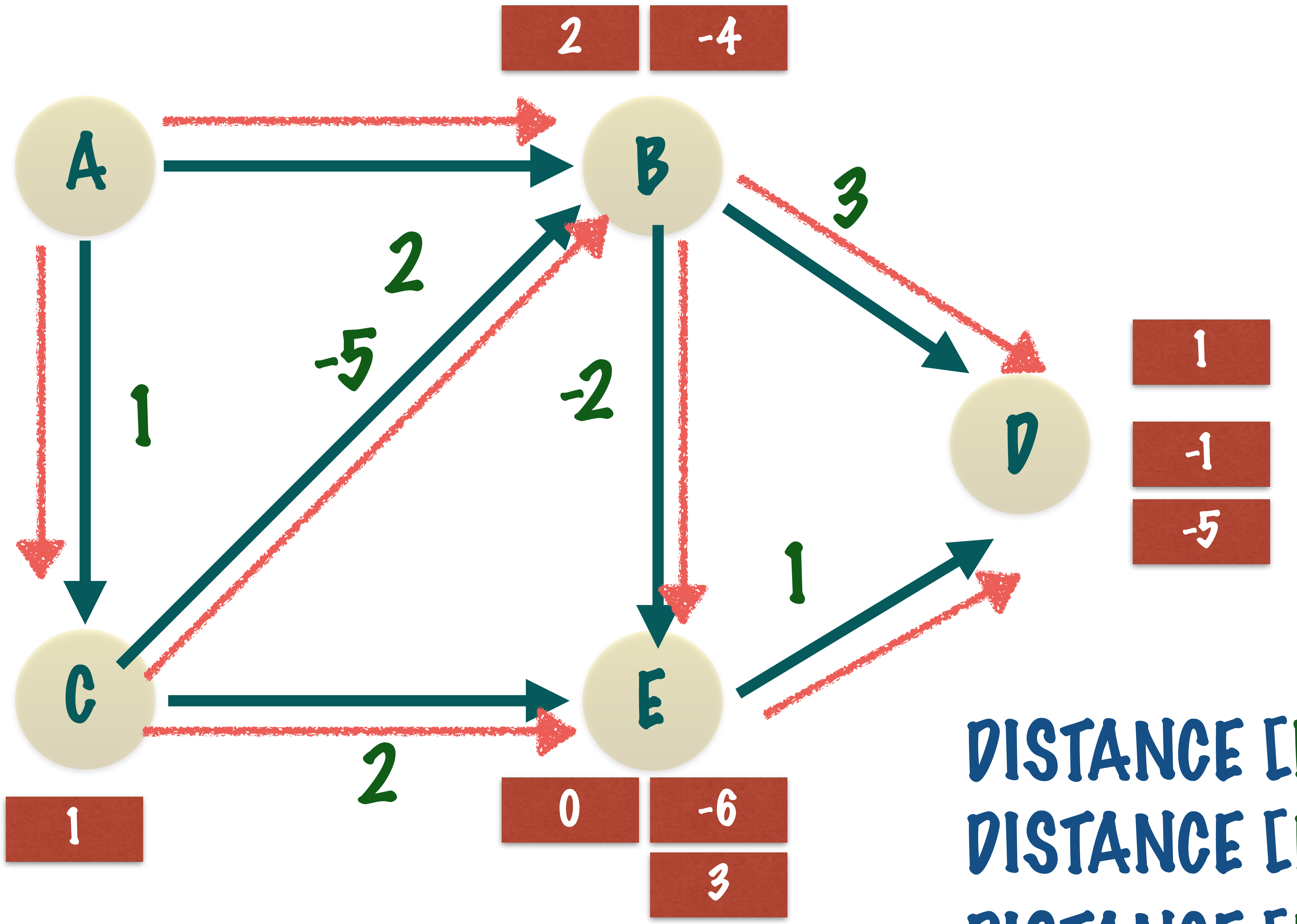


VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	-4	C
C	1	B
D	1	E
E	0	B

AFTER THE 1ST ITERATION ONLY B AND C'S DISTANCES ARE ACCURATE WHICH ARE AT 1 EDGE DISTANCE FROM SOURCE!

THE GRAPH

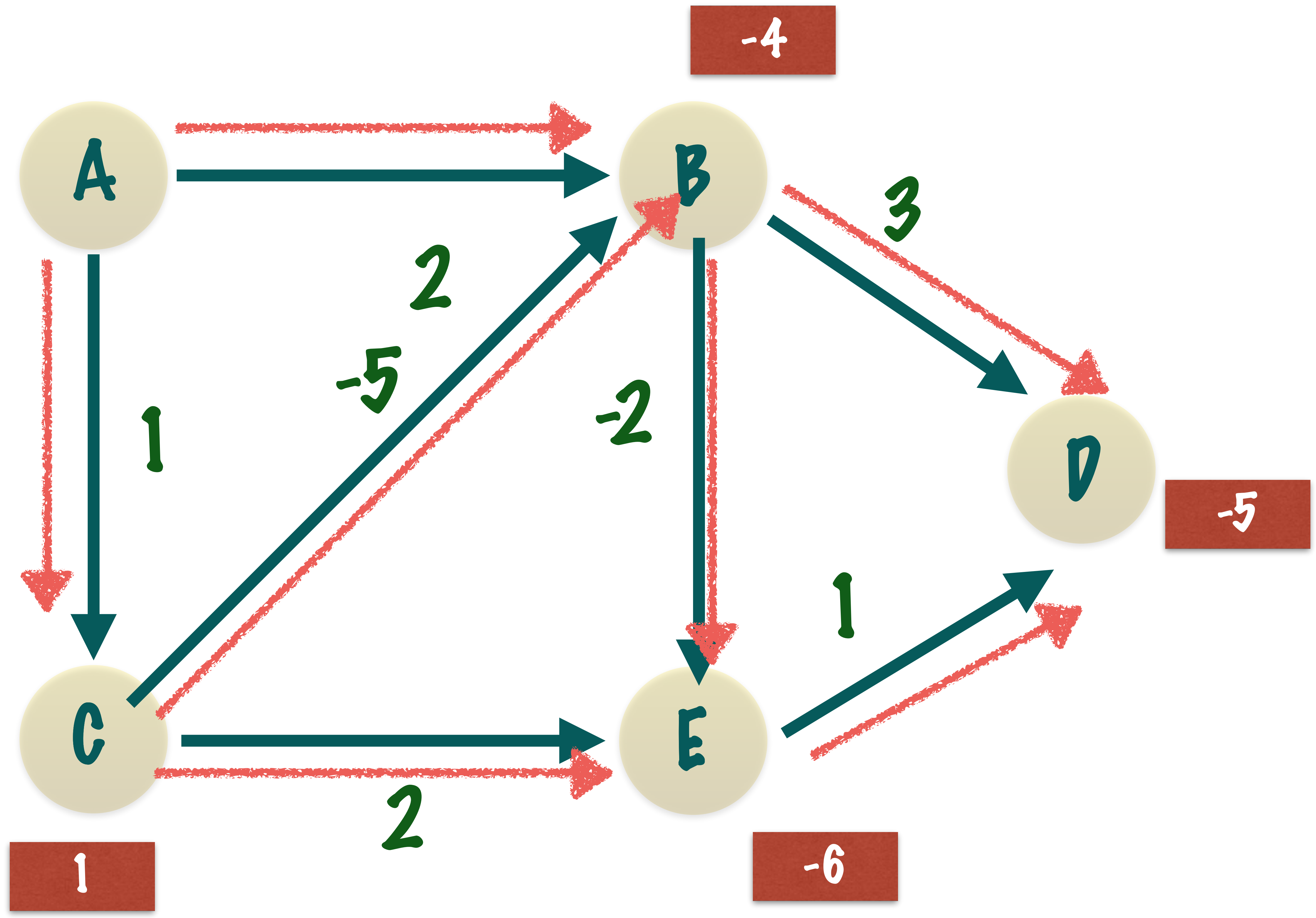
SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH



THE SECOND ITERATION

DISTANCE [B] = DISTANCE [A] + WEIGHT OF EDGE [A, B]
DISTANCE [D]= DISTANCE [B] + WEIGHT OF EDGE [B, D]
DISTANCE [E]= DISTANCE [B] + WEIGHT OF EDGE [B, E]
DISTANCE [D]= DISTANCE [E] + WEIGHT OF EDGE [E, D]
DISTANCE [C] = DISTANCE [A] + WEIGHT OF EDGE [A, C]
DISTANCE [E]= DISTANCE [C] + WEIGHT OF EDGE [C, E]
DISTANCE [B]= DISTANCE [C] + WEIGHT OF EDGE [C, B]

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH



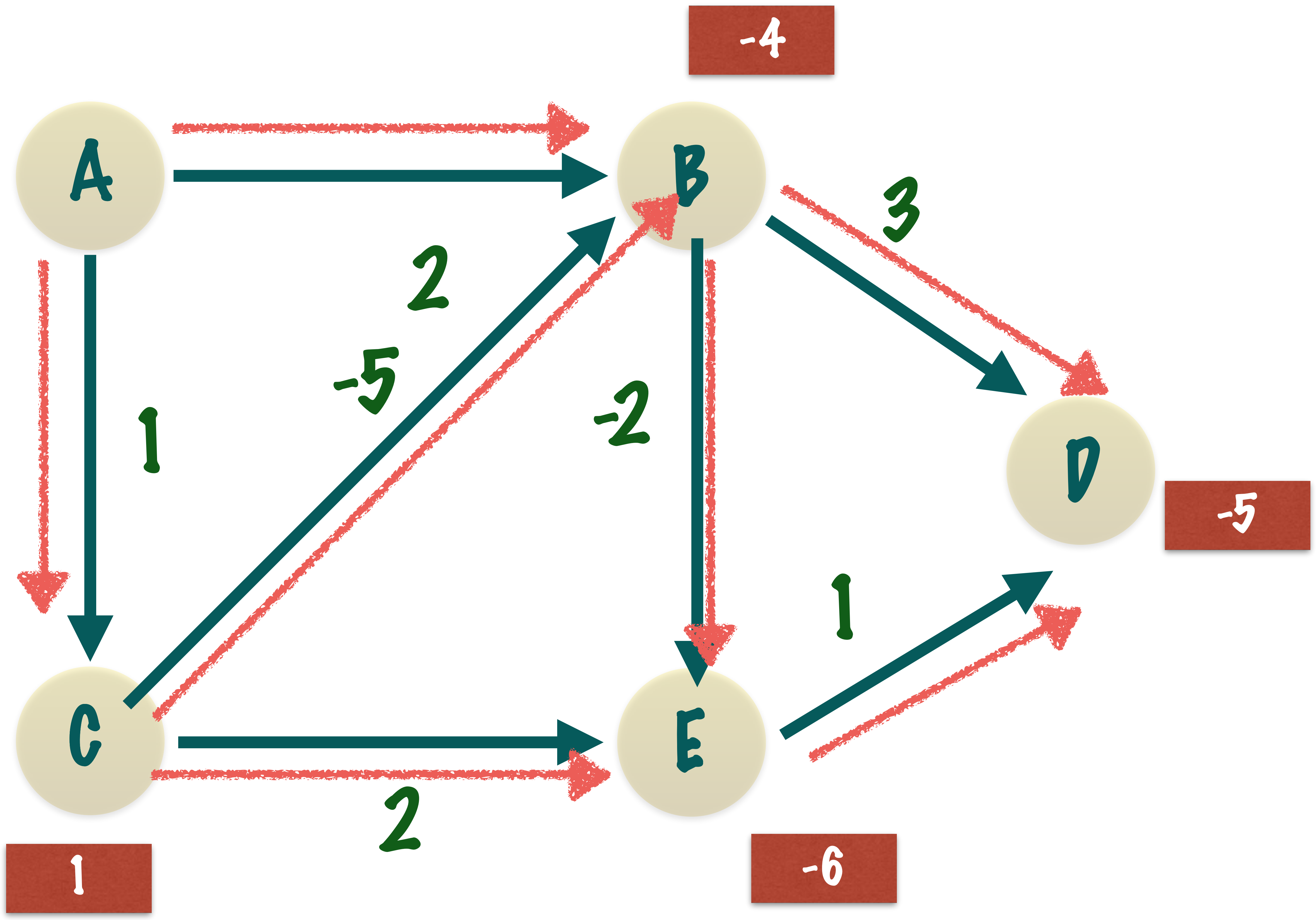
THE SECOND ITERATION

DISTANCE [B] = DISTANCE [A] + WEIGHT OF EDGE [A, B]
DISTANCE [D]= DISTANCE [B] + WEIGHT OF EDGE [B, D]
DISTANCE [E]= DISTANCE [B] + WEIGHT OF EDGE [B, E]
DISTANCE [D]= DISTANCE [E] + WEIGHT OF EDGE [E, D]
DISTANCE [C] = DISTANCE [A] + WEIGHT OF EDGE [A, C]
DISTANCE [E]= DISTANCE [C] + WEIGHT OF EDGE [C, E]
DISTANCE [B]= DISTANCE [C] + WEIGHT OF EDGE [C, B]

VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	-4	C
C	1	B
D	-5	B
E	-6	B

THE GRAPH

SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH



VERTEX	DISTANCE	LAST VERTEX
A	0	A
B	-4	C
C	1	B
D	-5	B
E	-6	B

AFTER THIS ITERATION ALL DISTANCES ARE ACCURATE!

IT JUST SO HAPPENED THAT WE GOT ACCURATE DISTANCES FROM SOURCE A WITH 2 ITERATIONS

TO GUARANTEE THAT WE FIND THE SHORTEST PATH WE NEED V - 1 ITERATIONS

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

TO GUARANTEE THAT WE FIND THE SHORTEST PATH WE NEED V - 1 ITERATIONS

WHY?

THE LONGEST POSSIBLE PATH IN A GRAPH HAS V - 1 EDGES

AFTER 1 ITERATION ALL VERTICES WHICH ARE 1 EDGE AWAY FROM THE SOURCE ARE ACCURATE

AFTER 2 ITERATIONS ALL VERTICES WHICH ARE 2 EDGES AWAY FROM THE SOURCE ARE ACCURATE

AFTER 3 ITERATIONS ALL VERTICES WHICH ARE 3 EDGES AWAY FROM THE SOURCE ARE ACCURATE

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

TO GUARANTEE THAT WE FIND THE SHORTEST PATH WE NEED V - 1 ITERATIONS

WHY?

THE LONGEST POSSIBLE PATH IN A GRAPH HAS V - 1 EDGES

AFTER 1 ITERATION ALL VERTICES WHICH ARE 1 EDGE AWAY FROM THE SOURCE ARE ACCURATE

AFTER 2 ITERATIONS ALL VERTICES WHICH ARE 2 EDGES AWAY FROM THE SOURCE ARE ACCURATE

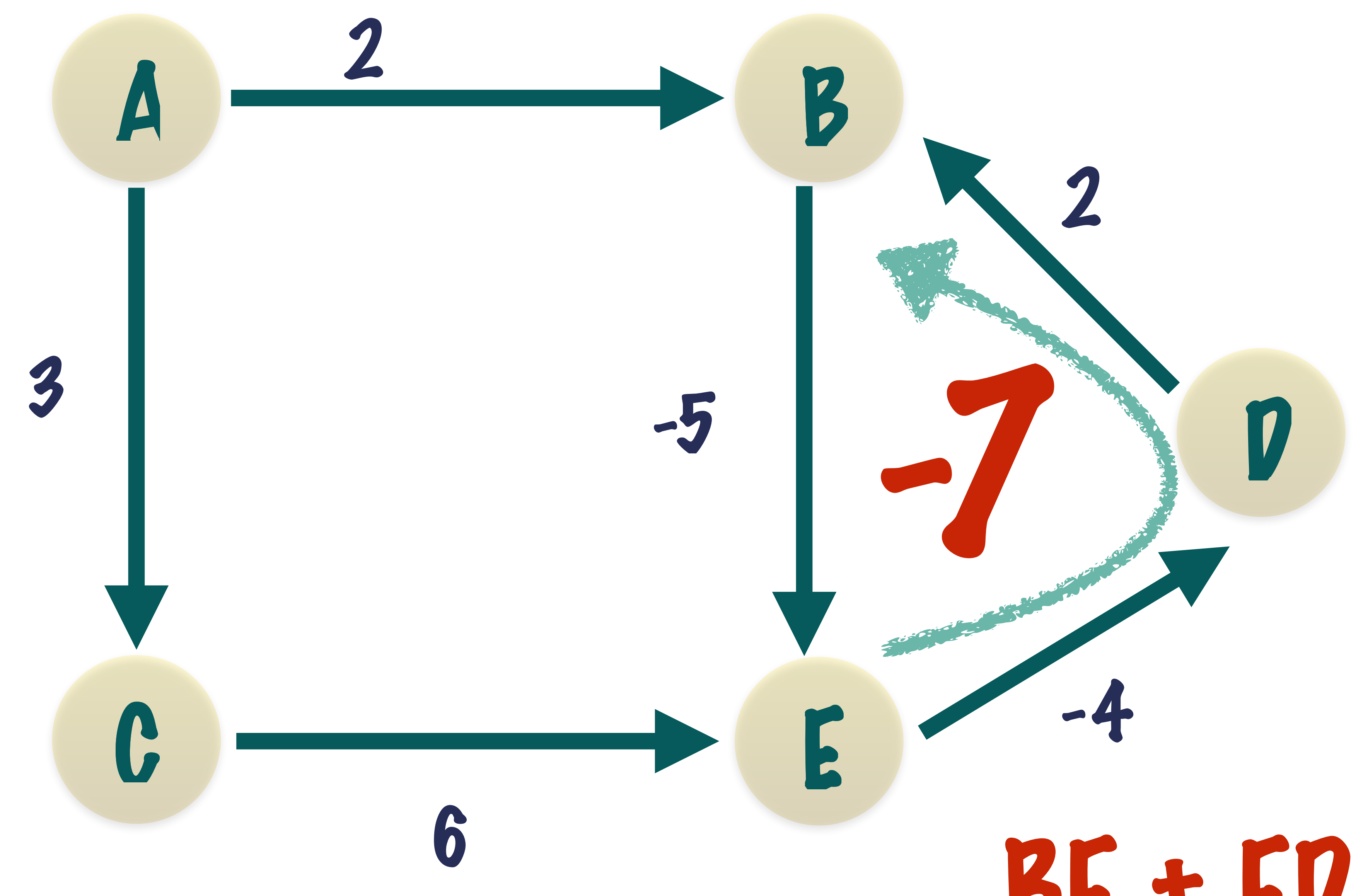
AFTER 3 ITERATIONS ALL VERTICES WHICH ARE 3 EDGES AWAY FROM THE SOURCE ARE ACCURATE

WITH **V - 1** ITERATIONS WE CAN GUARANTEE THAT THE VERTEX FURTHEST AWAY FROM THE SOURCE WILL ITS DISTANCE ACCURATELY CALCULATED

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

BUT THERE IS ONE THING THAT WE NEED TO TAKE CARE OF - NEGATIVE CYCLES

CONSIDER B, E, D



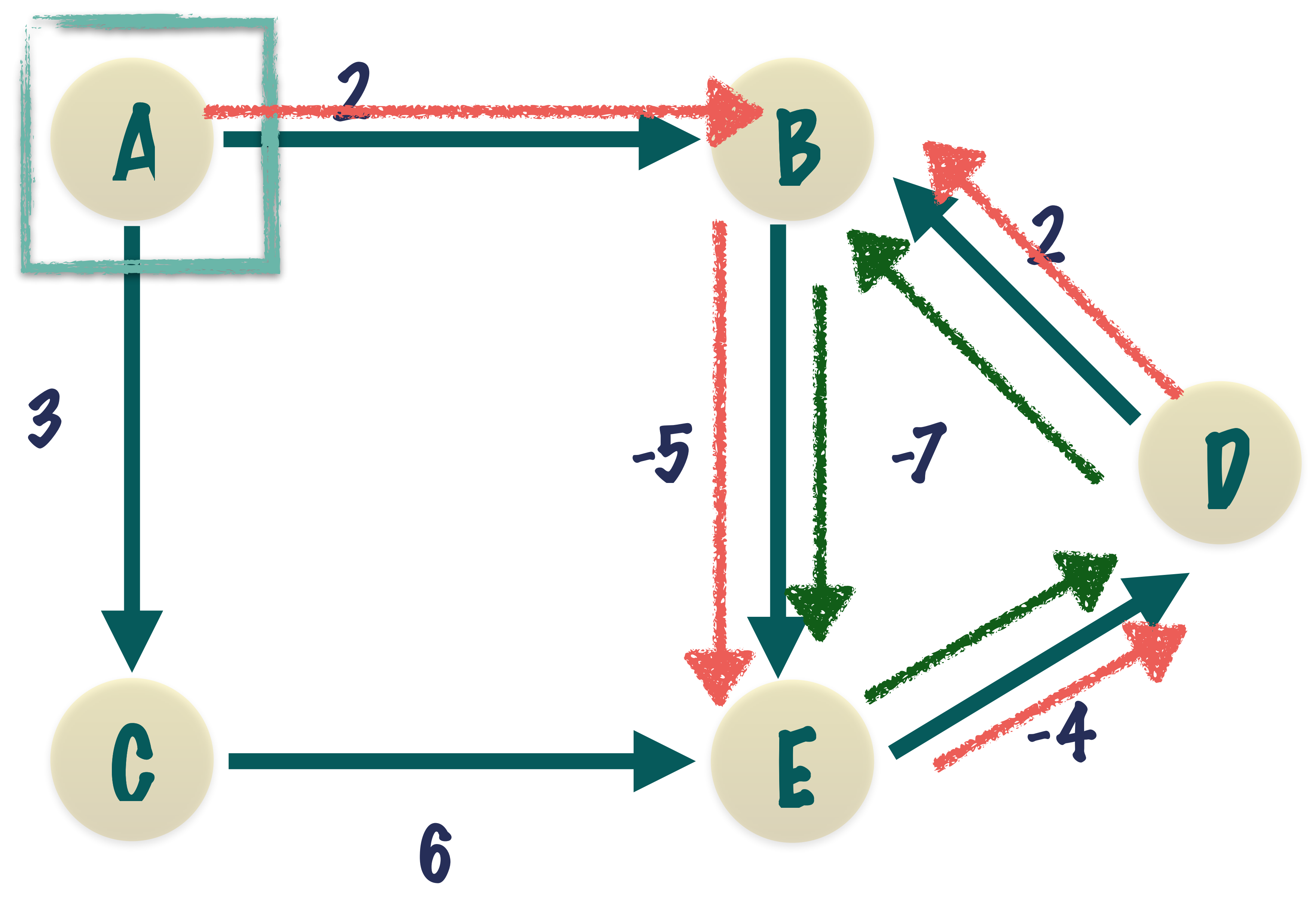
B -> E THEN FROM E -> D AND FINALLY BACK TO D -> B

BE + ED + DB = (-5) + (-4) + (2) = -7

THE PATH IS A CYCLE WITH A NEGATIVE DISTANCE!

THE GRAPH
SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

IF A IS THE SOURCE VERTEX AND WE HAVE TO FIND THE MINIMUM PATH TO B



DIRECT PATH FROM A TO B

DISTANCE = 2

ONE ROUND OF THE CYCLE

DISTANCE = 2 - 7 = -5

SECOND ROUND OF CYCLE

DISTANCE = -5 - 7 = - 12

A -> B -> E -> D -> B -> E -> D -> B

THE GRAPH

SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

DIRECT PATH FROM A TO B

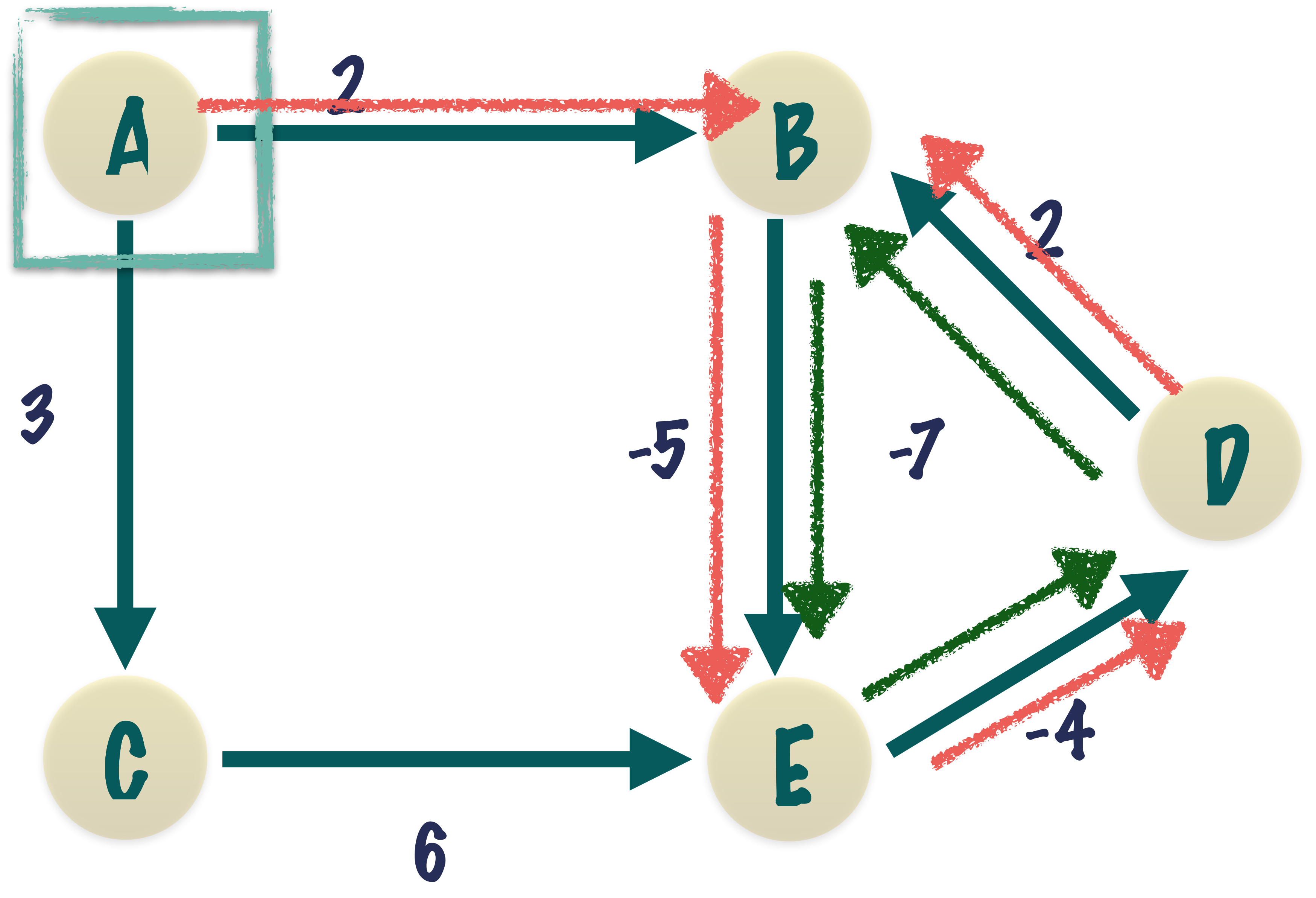
DISTANCE = 2

ONE ROUND OF THE CYCLE

DISTANCE = 2 - 7 = -5

SECOND ROUND OF CYCLE

DISTANCE = -5 - 7 = - 12

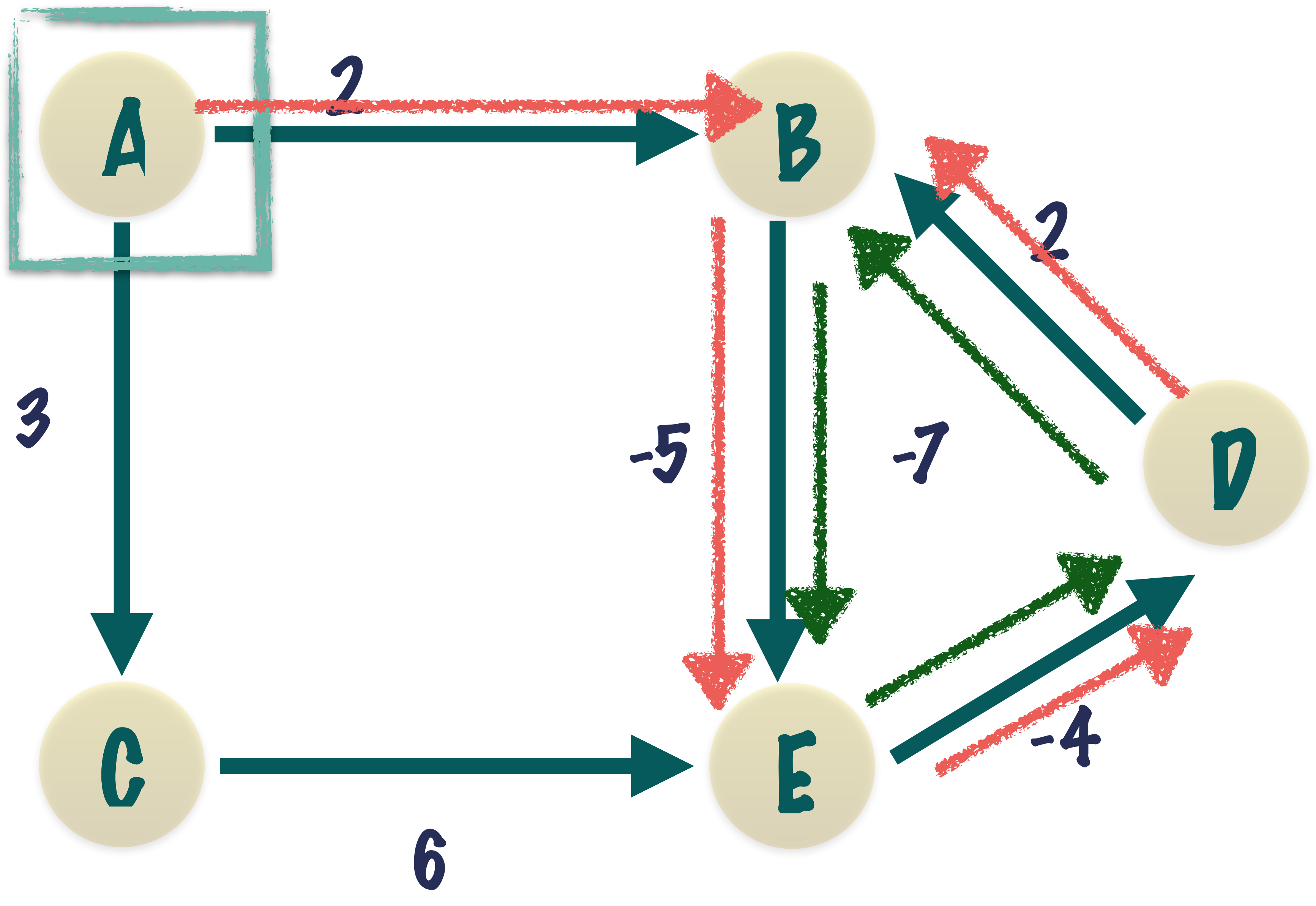


A -> B -> E -> D -> B -> E -> D -> B

THERE IS NO SHORTEST OR CHEAPEST PATH!
EVERY TIME WE GO AROUND WE WILL FIND SOMETHING SHORTER AND CHEAPER

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

THERE IS NO SHORTEST OR CHEAPEST PATH!
EVERY TIME WE GO AROUND WE WILL FIND SOMETHING SHORTER AND CHEAPER



SO HOW DO WE DETECT A CYCLE?

AFTER (V-1) ITERATION, WE WILL DO ONE MORE ITERATION (RELAXATION)

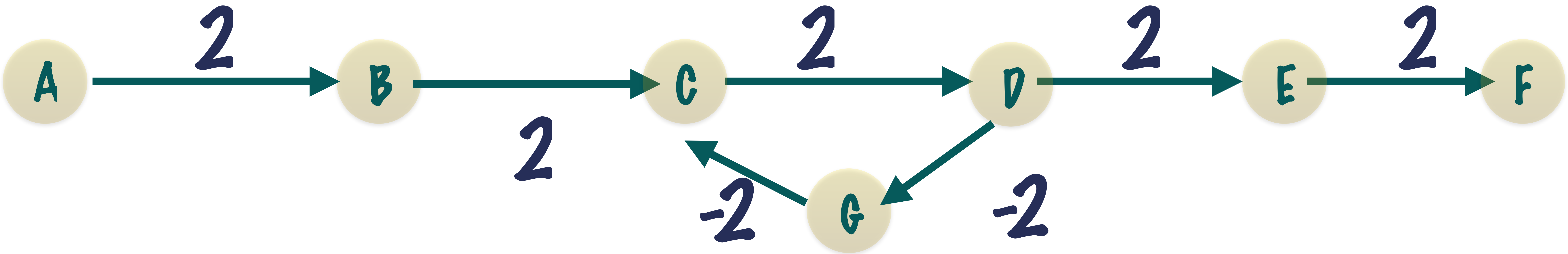
IF DISTANCE OF ANY VERTEX STILL GETS UPDATED, THEN THERE IS DEFINITELY A CYCLE!

THE GRAPH

SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

IF DISTANCE OF ANY VERTEX STILL GETS UPDATED, THEN THERE IS DEFINITELY A CYCLE!

NOTE: IF NO NEGATIVE CYCLES EXIST THEN THE LONGEST PATH BETWEEN TWO VERTICES IS AT MOST V - 1 EDGES



THE PRESENCE OF A NEGATIVE CYCLE CAN MAKE A PATH BETWEEN TWO VERTICES LONGER THAN THE LONGEST POSSIBLE PATH IN A GRAPH I.E GREATER THAN V - 1

THE GRAPH SHORTEST PATH IN NEGATIVE WEIGHTED GRAPH

IN THE WORST CASE WE TRAVERSE
ALL EDGES V - 1 TIMES

RUNNING TIME IS : O(E*V)
[IF ADJACENCY LISTS ARE USED]

RUNNING TIME IS : $O(V^3)$ [IF ADJACENCY MATRIX ARE USED]

E = V*V IN ADJACENCY MATRIX

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. Set it to
        // a very large value rather than the maximum integer value. Bellman Ford
        // supports negative weights and adding anything to this distance can
        // make it a negative value which interferes with the algorithm.
        distance = 100000;
        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 A LARGE VALUE RATHER THAN INTEGER.MAX

THIS IS BECAUSE ADDING TO INTEGER.MAX CAUSES THE INTEGER TO OVERFLOW AND BECOME A NEGATIVE VALUE - THIS NEGATIVE VALUE CAN INTERFERE WITH THE ALGORITHM

BUILD THE DISTANCE TABLE - SETUP

```java
public static Map<Integer, DistanceInfo> buildDistanceTable(Graph graph, int source) {
    Map<Integer, DistanceInfo> distanceTable = new HashMap<>();
    for (int j = 0; j < graph.getNumVertices(); j++) {
        distanceTable.put(j, new DistanceInfo());
    }

    // Set up the distance of the specified source.
    distanceTable.get(source).setDistance(0);
    distanceTable.get(source).setLastVertex(source);

    LinkedList<Integer> queue = new LinkedList<>();
```

ADD A DISTANCE TABLE ENTRY FOR EACH NODE IN THE GRAPH

SET UP A SIMPLE QUEUE TO EXPLORE ALL THE VERTICES REGARDLESS OF PRIORITY

BUILD THE DISTANCE TABLE - PROCESS

```java
// Relaxing (processing) all the edges numVertices - 1 times
for (int numIterations = 0; numIterations < graph.getNumVertices() - 1; numIterations++) {
    // Add every vertex to the queue so we're sure to access all the edges
    // in the graph.
    for (int v = 0; v < graph.getNumVertices(); v++) {
        queue.add(v);
    }

    // Keep track of the edges visited so we visit each edge just once
    // in every iteration.
    Set<String> visitedEdges = new HashSet<>();
    while (!queue.isEmpty()) {
        int currentVertex = queue.pollFirst();

        for (int neighbor : graph.getAdjacentVertices(currentVertex)) {
            String edge = String.valueOf(currentVertex) + String.valueOf(neighbor);

            // Do not visit edges more than once in each iteration.
            if (visitedEdges.contains(edge)) {
                continue;
            }
            visitedEdges.add(edge);

            int distance = distanceTable.get(currentVertex).getDistance()
                    + graph.getWeightedEdge(currentVertex, neighbor);
            // If we find a new shortest path to the neighbour update
            // the distance and the last vertex.
            if (distanceTable.get(neighbor).getDistance() > distance) {
                distanceTable.get(neighbor).setDistance(distance);
                distanceTable.get(neighbor).setLastVertex(currentVertex);
            }
        }
    }
}
```

ITERATE THROUGH ALL EDGES NUMBER OF VERTICES - 1 TIMES

ADD ALL VERTICES TO THE QUEUE SO WE CAN EXPLORE EVERY EDGE IN THIS GRAPH

KEEP TRACK OF VISITED EDGES SO WE DO NOT VISIT AN EDGE MORE THAN ONCE PER ITERATION

EDGE IS REPRESENTED AS A STRING "0 1" IS AN EDGE GOING FROM VERTEX 0 TO VERTEX 1

CHECK THE DISTANCES OF THE VERTICES AND UPDATE IF SHORTER ROUTES ARE FOUND

BUILD THE DISTANCE TABLE - NEGATIVE CYCLES CHECK

```java
// Add all the vertices to the queue one last time to check for
// a negative cycle.
for (int v = 0; v < graph.getNumVertices(); v++) {
    queue.add(v);
}

// Relaxing (processing) all the edges one last time to check if
// there exists a negative cycle
while (!queue.isEmpty()) {
    int currentVertex = queue.pollFirst();
    for (int neighbor : graph.getAdjacentVertices(currentVertex)) {
        int distance = distanceTable.get(currentVertex).getDistance()
                + graph.getWeightedEdge(currentVertex, neighbor);
        if (distanceTable.get(neighbor).getDistance() > distance) {
            throw new IllegalArgumentException("The Graph has a -ve cycle");
        }
    }
}

return distanceTable;
```

ADD ALL VERTICES ONCE AGAIN TO THE QUEUE TO CHECK ALL EDGES ONE LAST TIME FOR CYCLES

IF THE DISTANCE TABLE CAN BE UPDATED FOR ANY VERTEX AFTER NUMBER OF VERTICES - 1 ITERATIONS IT MEANS THERE IS A NEGATIVE CYCLE IN THE GRAPH

THROW AN EXCEPTION IN THAT CASE, WE CAN'T FIND THE SHORTEST PATH IN A GRAPH WITH NEGATIVE CYCLES

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