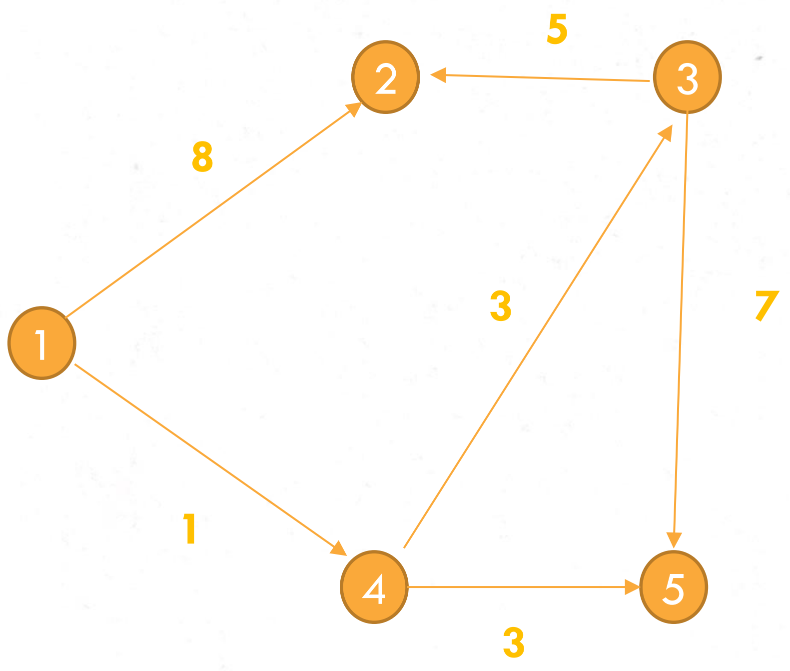
DYNAMIC PROGRAMMING TECHNIQUE

BELLMAN-FORD ALGORITHM

DYNAMMIC PROGRAMMING TECHNIQUE

Dijkestra’s algorithm

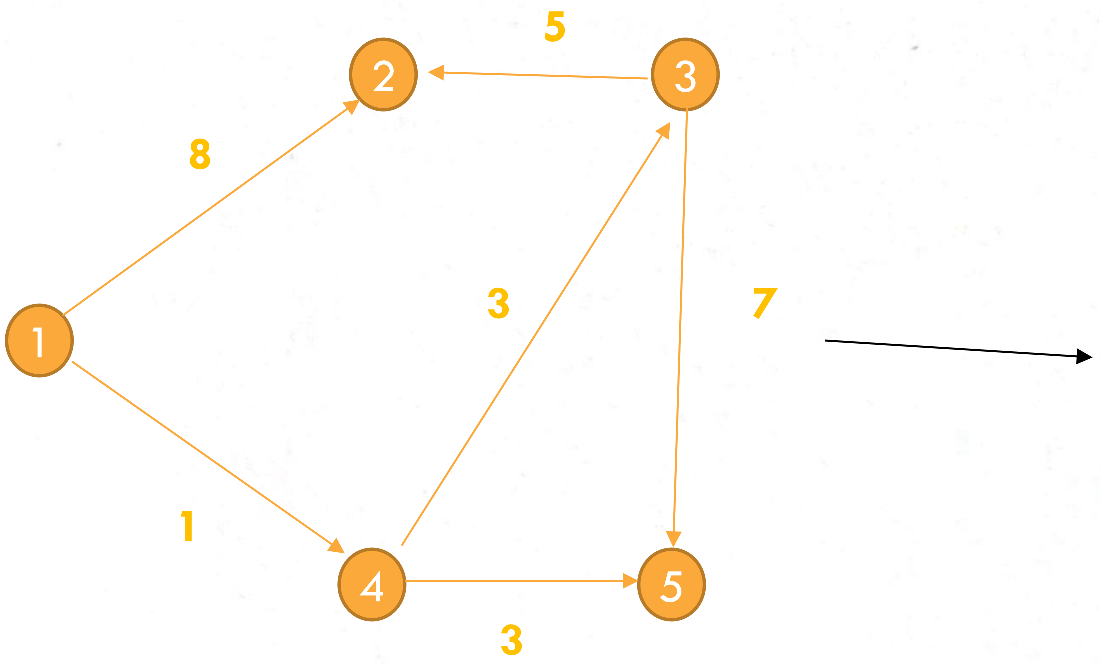


technique:.

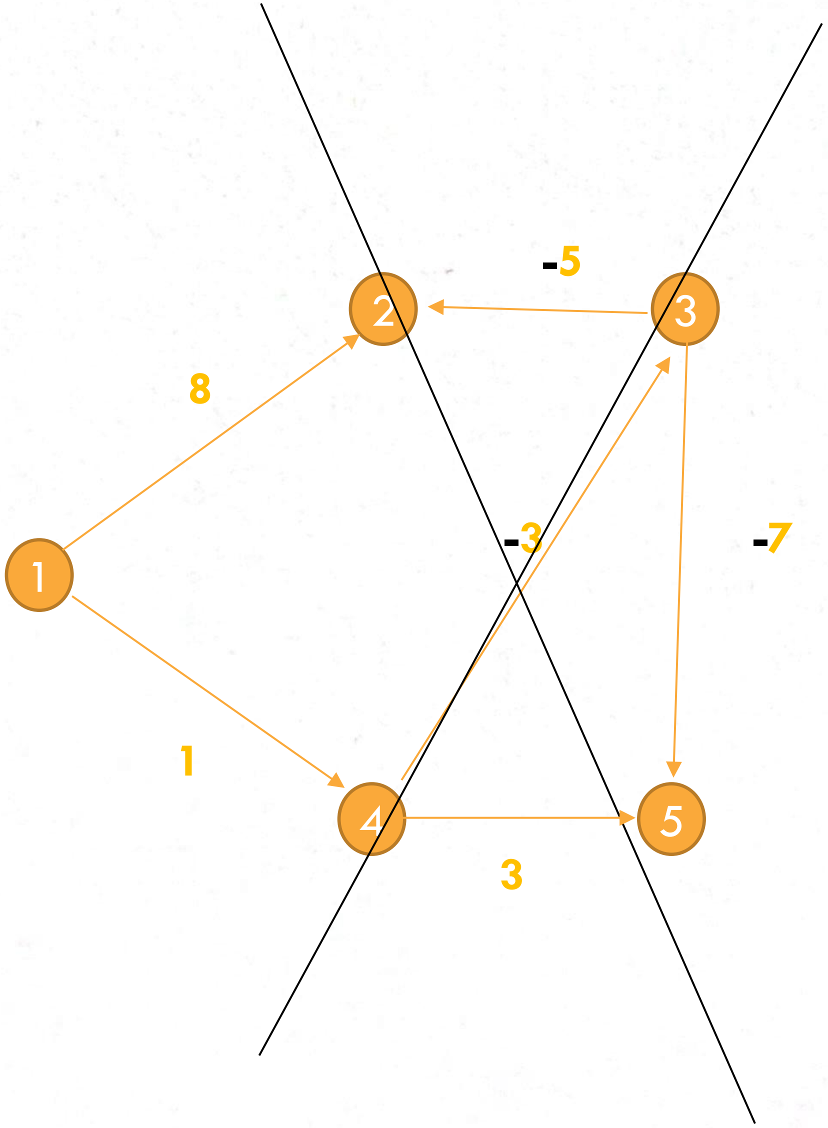
Greedy Programming

goal: “single-source shortest path”

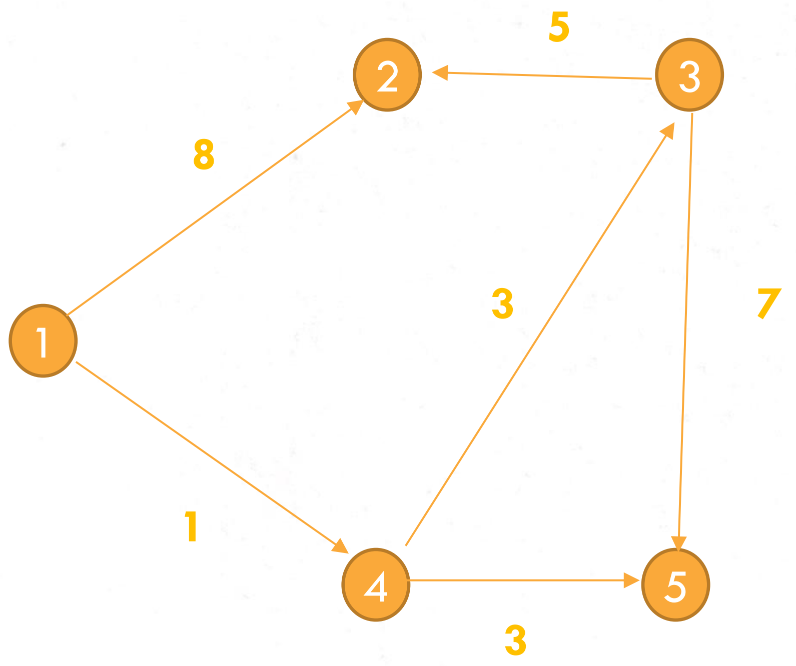
Dijkestra's algorithm



Negative weight



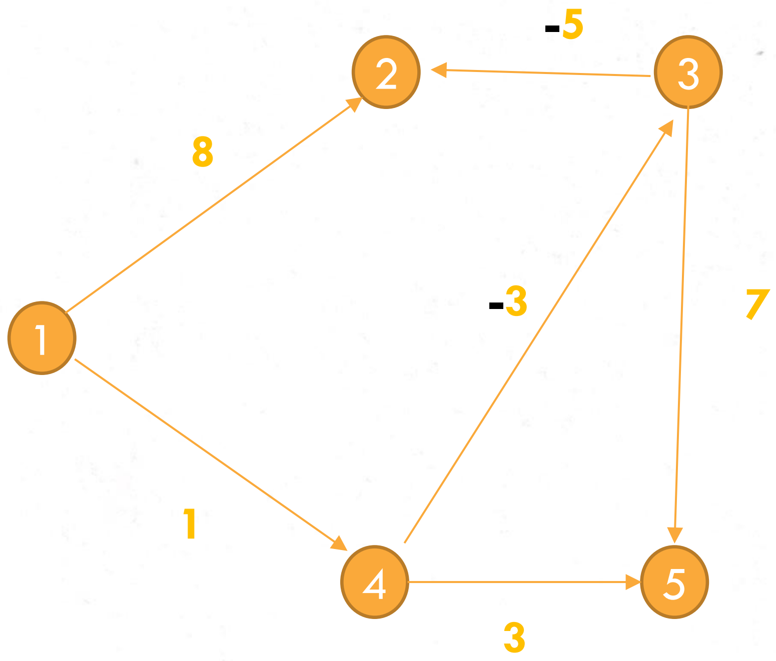
Dijkestra's algorithm



- Dijkstra:

- Works only with graphs having non-negative edge weights (no negative weights allowed).
- Cannot handle negative weight edges or detect negative cycles.

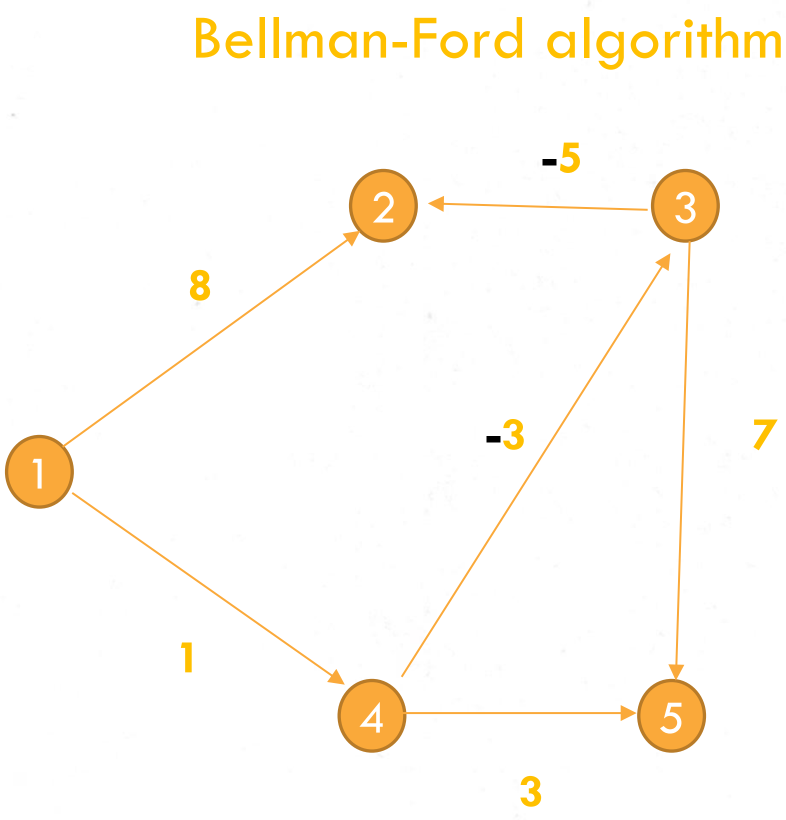
Bellman-Ford algorithm



technique:.

Dynammic Programming

goal: "single-source shortest path"



Bellman-Ford algorithm advantages over Dijkestra's algorithm :

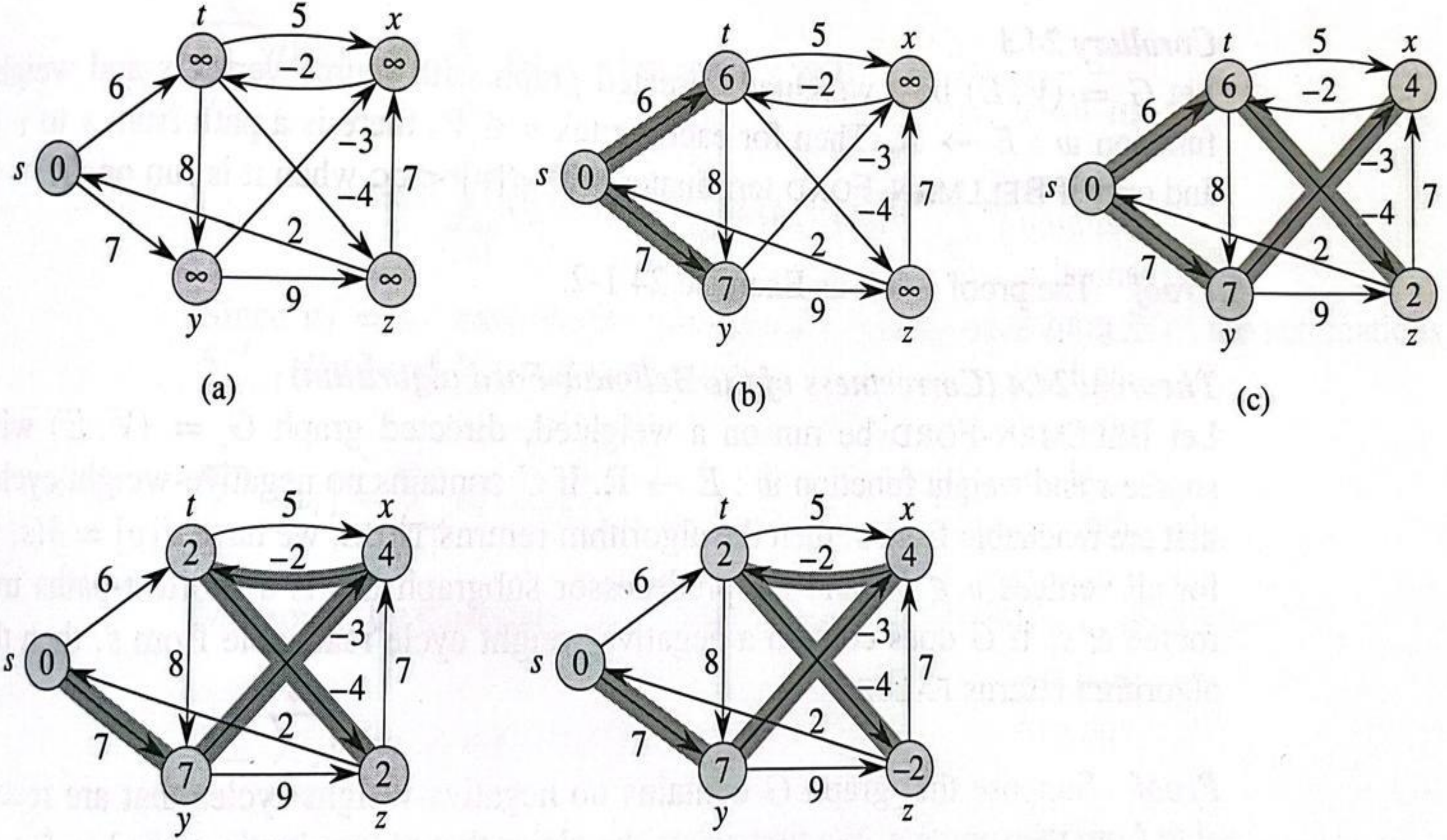
- Works with graphs having negative edge weights.
- Can detect negative weight cycles (cycles where the total sum of edge weights is negative).
- If a negative cycle exists reachable from the source, Bellman-Ford reports it (no shortest path exists).

```
BELLMAN-FORD(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  for i ← 1 to |V[G]| − 1
       do for each edge (u, v) ∈ E[G]
              do RELAX(u, v, w)
  for each edge (u, v) ∈ E[G]
       do if d[v] > d[u] + w(u, v)
             then return FALSE
  return TRUE
```

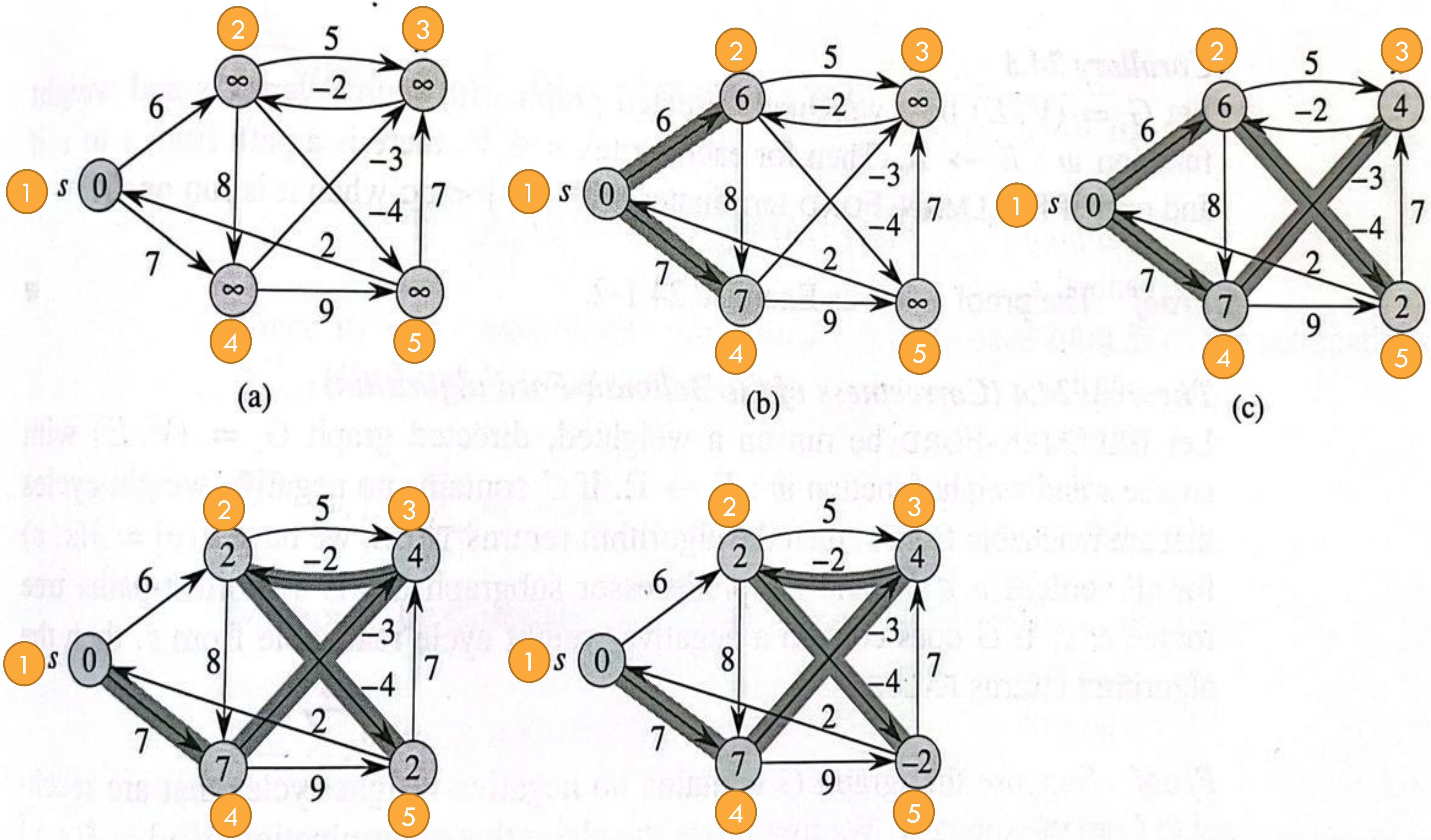
```
Relax(u, v, w)
  if d[v] > d[u] + w(u, v)
     d[v] = d[u] + w(u, v)
```

```java
import java.util.Arrays;
public class Bellman_Ford {

    static int[] bellmanFord(int V, int[][] edges, int src) {
        // Initially distance from source to all other vertices
        // is not known(Infinite).
        int[] dist = new int[V];
        Arrays.fill(dist, (int)1e8);
        dist[src-1] = 0;

        // Relaxation of all the edges V times, not (V - 1) as we
        // need one additional relaxation to detect negative cycle
        for (int i = 1; i <= V-1; i++) {
            for (int[] edge : edges) {
                int u = edge[0]-1;
                int v = edge[1]-1;
                int wt = edge[2];
                if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {

                    // Update shortest distance to node v
                    dist[v] = dist[u] + wt;
                }
            }
        }
        for (int[] edge : edges) {
                int u = edge[0]-1;
                int v = edge[1]-1;
                int wt = edge[2];
                if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {

                    // If this is the Vth relaxation, then there is
                    // a negative cycle
                        return new int[]{-1};
                }
            }
        return dist;
    }
}
```

```java
public class App3 {

    public static void main(String[] args) {

        // Number of vertices in the graph
        int V = 5;

        // Edge list representation: {source, destination, weight}
        int[][] edges = new int[][] {
            {2, 4, 2},
            {5, 4, -1},
            {3, 5, 1},
            {2, 3, 1},
            {1, 2, 5}
        };

        // Source vertex for Bellman-Ford algorithm
        int src = 1;

        // Run Bellman-Ford algorithm from the source vertex
        int[] ans = Bellman_Ford.bellmanFord(V, edges, src);

        // Print shortest distances from the source to all vertices
        for (int dist : ans)
            System.out.print(dist + " ");

    }
}
```

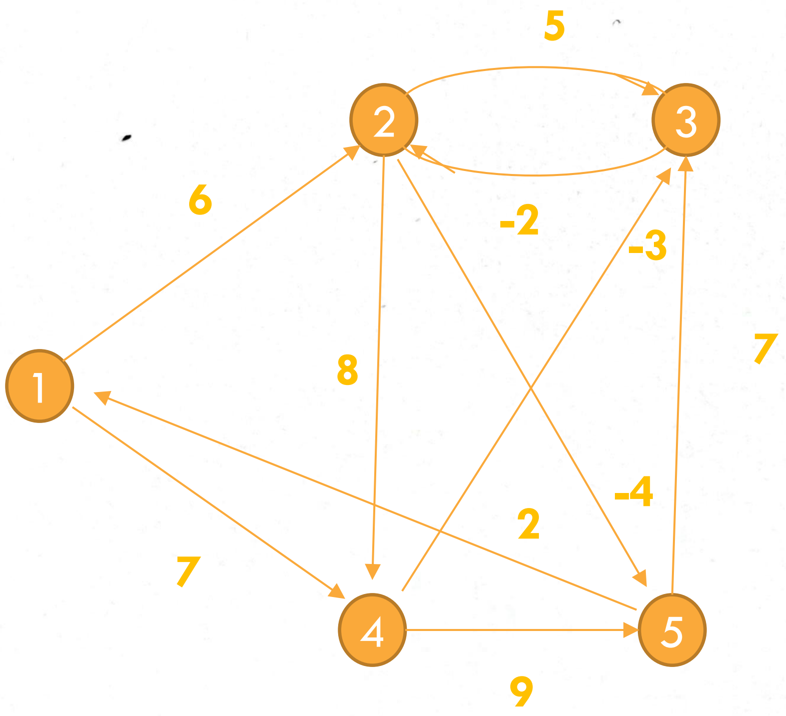


t
x
s
y
z
5
−2
6
8
−3
−4
7
2
7
9
(a)
(b)
(c)



(a) (b) (c)

EX:



EX:

Graph Details (Revised):

- (1 → 2) with weight 6
- (1 → 4) with weight 7
- (2 → 3) with weight 5
- (2 → 4) with weight 8
- (3 → 2) with weight -2
- (5 → 1) with weight 2
- (5 → 3) with weight 7
- (2 → 5) with weight -4
- (4 → 5) with weight 9
- (4 → 3) with weight -3

```
BELLMAN-FORD(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  for i ← 1 to |V[G]| − 1
       do for each edge (u, v) ∈ E[G]
              do RELAX(u, v, w)
  for each edge (u, v) ∈ E[G]
       do if d[v] > d[u] + w(u, v)
             then return FALSE
  return TRUE
```

EX: Initialization

- Initially, set distances as follows:
- Distance to vertex 1 (the source) = 0
- Distance to all other vertices = ∞

dist = [0, ∞, ∞, ∞, ∞]

EX: First loop

Iteration 1 (Relaxing Edges):

1. Relax edge (1 → 2) with weight 6:dist[2] = min(∞, 0 + 6) = 6
2. Relax edge (1 → 4) with weight 7:dist[4] = min(∞, 0 + 7) = 7
3. Relax edge (2 → 3) with weight 5:dist[3] = min(∞, 6 + 5) = 11
4. Relax edge (2 → 4) with weight 8:dist[4] = min(7, 6 + 8) = 7 (No update)
5. Relax edge (3 → 2) with weight -2:dist[2] = min(6, 11 - 2) = 6 (No update)
6. Relax edge (5 → 1) with weight 2:dist[1] = min(0, ∞ + 2) = 0 (No update)
7. Relax edge (5 → 3) with weight 7:dist[3] = min(11, ∞ + 7) = 11 (No update)
8. Relax edge (2 → 5) with weight -4:dist[5] = min(∞, 6 - 4) = 2
9. Relax edge (4 → 5) with weight 9:dist[5] = min(2, 7 + 9) = 2 (No update)
10. Relax edge (4 → 3) with weight -3:dist[3] = min(11, 7 - 3) = 4

After Iteration 1, the distances are:

dist = [0, 6, 4, 7, 2]

EX: Second loop

Iteration 2 (Relaxing Edges Again):
Now we repeat the process of relaxing all the edges.

1. Relax edge (1 → 2) with weight 6:dist[2] = min(6, 0 + 6) = 6 (No update)
2. Relax edge (1 → 4) with weight 7:dist[4] = min(7, 0 + 7) = 7 (No update)
3. Relax edge (2 → 3) with weight 5:dist[3] = min(4, 6 + 5) = 4 (No update)
4. Relax edge (2 → 4) with weight 8:dist[4] = min(7, 6 + 8) = 7 (No update)
5. Relax edge (3 → 2) with weight -2:dist[2] = min(6, 4 - 2) = 2
6. Relax edge (5 → 1) with weight 2:dist[1] = min(0, 2 + 2) = 0 (No update)
7. Relax edge (5 → 3) with weight 7:dist[3] = min(4, 2 + 7) = 4 (No update)
8. Relax edge (2 → 5) with weight -4:dist[5] = min(2, 2 - 4) = -2
9. Relax edge (4 → 5) with weight 9:dist[5] = min(-2, 7 + 9) = -2 (No update)
10. Relax edge (4 → 3) with weight -3:dist[3] = min(4, 7 - 3) = 4 (No update)

After Iteration 2, the distances are:

dist = [0, 2, 4, 7, -2]

EX: Third loop

Iteration 3 (Relaxing Edges):

Now we repeat the process of relaxing all the edges.

1. Relax edge (1 → 2) with weight 6:dist[2] = min(2, 0 + 6) = 2 (No update)
2. Relax edge (1 → 4) with weight 7:dist[4] = min(7, 0 + 7) = 7 (No update)
3. Relax edge (2 → 3) with weight 5:dist[3] = min(4, 2 + 5) = 4 (No update)
4. Relax edge (2 → 4) with weight 8:dist[4] = min(7, 2 + 8) = 7 (No update)
5. Relax edge (3 → 2) with weight -2:dist[2] = min(2, 4 - 2) = 2 (No update)
6. Relax edge (5 → 1) with weight 2:dist[1] = min(0, -2 + 2) = 0 (No update)
7. Relax edge (5 → 3) with weight 7:dist[3] = min(4, -2 + 7) = 4 (No update)
8. Relax edge (2 → 5) with weight -4:dist[5] = min(-2, 2 - 4) = -2 (No update)
9. Relax edge (4 → 5) with weight 9:dist[5] = min(-2, 7 + 9) = -2 (No update)
10. Relax edge (4 → 3) with weight -3:dist[3] = min(4, 7 - 3) = 4 (No update)

After Iteration 3, the distances are:

dist = [0, 2, 4, 7, -2]

EX: Fourth loop

Iteration 4 (Relaxing Edges):

Now we repeat the process of relaxing all the edges one last time.

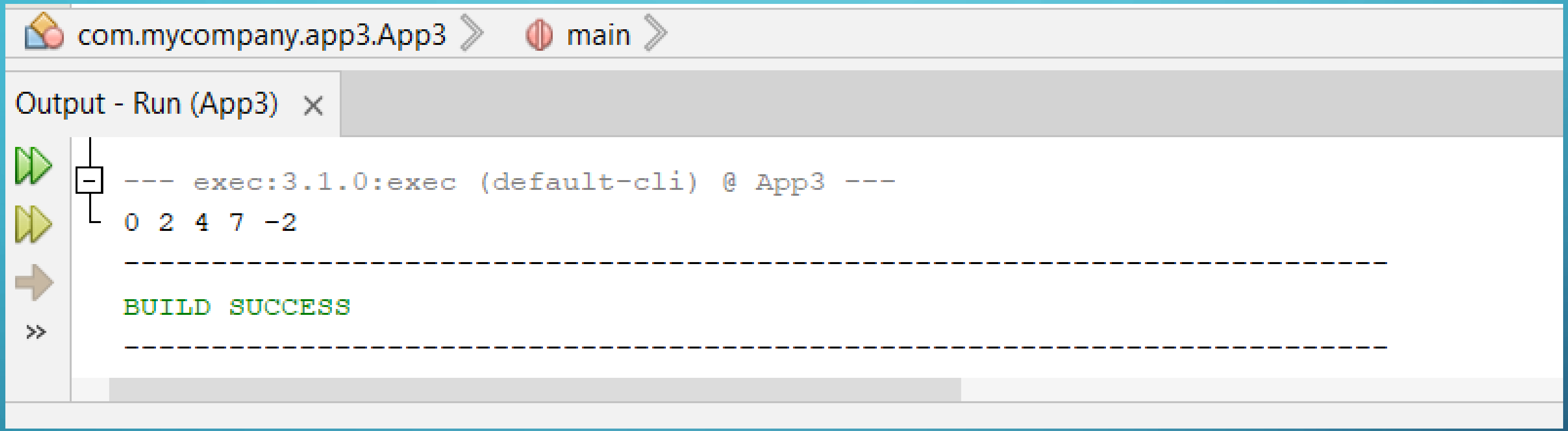
1. Relax edge (1 → 2) with weight 6:dist[2] = min(2, 0 + 6) = 2 (No update)
2. Relax edge (1 → 4) with weight 7:dist[4] = min(7, 0 + 7) = 7 (No update)
3. Relax edge (2 → 3) with weight 5:dist[3] = min(4, 2 + 5) = 4 (No update)
4. Relax edge (2 → 4) with weight 8:dist[4] = min(7, 2 + 8) = 7 (No update)
5. Relax edge (3 → 2) with weight -2:dist[2] = min(2, 4 - 2) = 2 (No update)
6. Relax edge (5 → 1) with weight 2:dist[1] = min(0, -2 + 2) = 0 (No update)
7. Relax edge (5 → 3) with weight 7:dist[3] = min(4, -2 + 7) = 4 (No update)
8. Relax edge (2 → 5) with weight -4:dist[5] = min(-2, 2 - 4) = -2 (No update)
9. Relax edge (4 → 5) with weight 9:dist[5] = min(-2, 7 + 9) = -2 (No update)
10. Relax edge (4 → 3) with weight -3:dist[3] = min(4, 7 - 3) = 4 (No update)

After Iteration 4, the distances are:

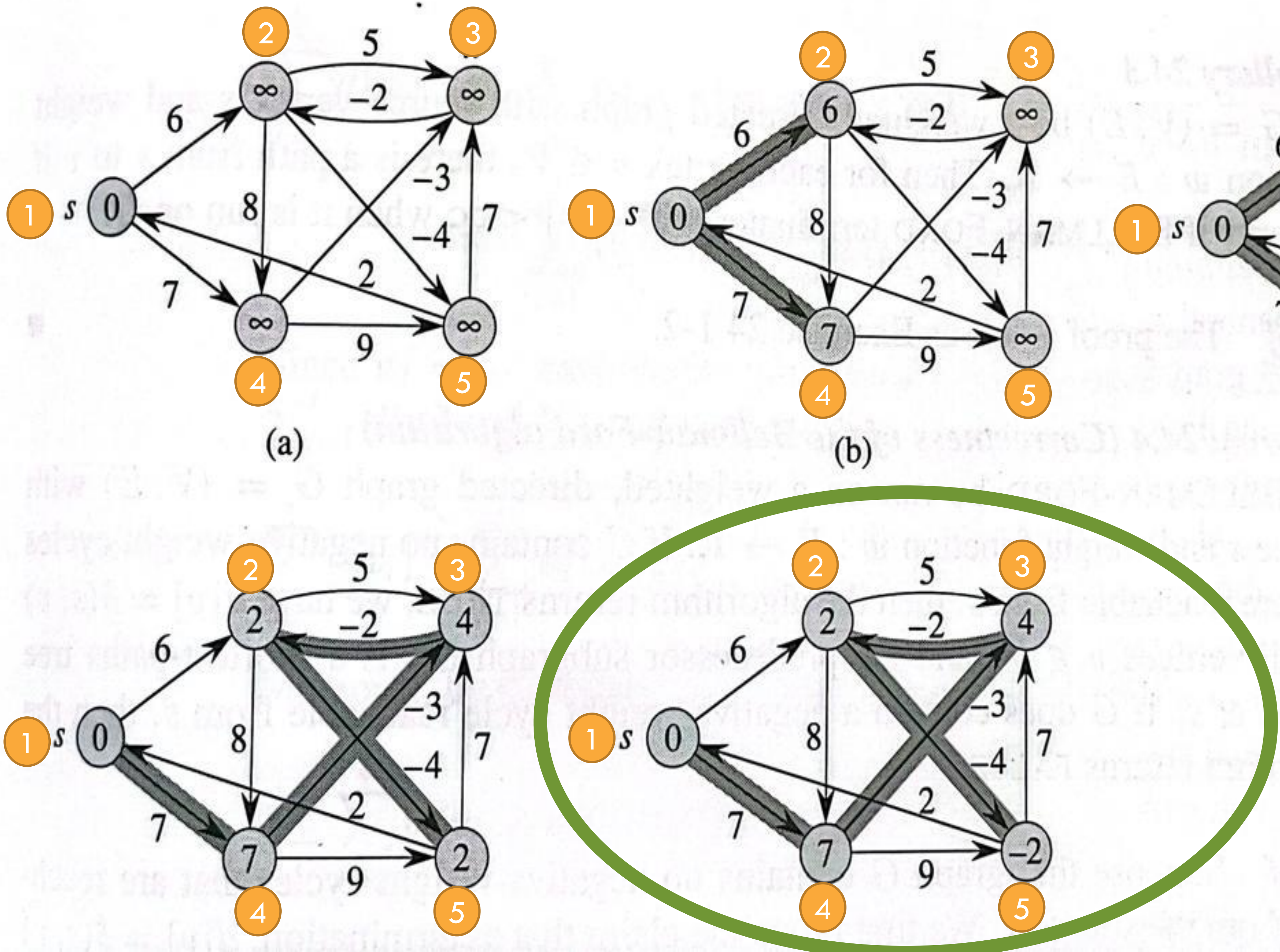
dist = [0, 2, 4, 7, -2]

```java
public class App3 {

    public static void main(String[] args) {

        // Number of vertices in the graph
        int V = 5;

        // Edge list representation: {source, destination, weight}
        int[][] edges = new int[][] {
            {1, 2, 6},
            {1, 4, 7},
            {2, 4, 8},
            {2, 3, 5},
            {3, 2, -2},
            {5, 1, 2},
            {3, 5, 7},
            {4, 5, 9},
            {2, 5, -4},
            {4, 3, -3},
        };

        // Source vertex for Bellman-Ford algorithm
        int src = 1;

        // Run Bellman-Ford algorithm from the source vertex
        int[] ans = Bellman_Ford.bellmanFord(V, edges, src);

        // Print shortest distances from the source to all vertices
        for (int dist : ans)
            System.out.print(dist + " ");

    }
}
```

THE OUTPUT:.

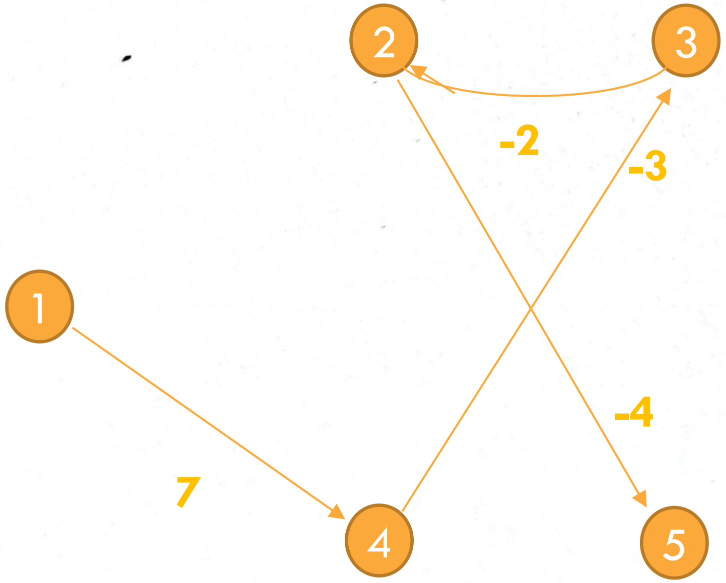

com.mycompany.app3.App3
main
Output - Run (App3)
--- exec:3.1.0:exec (default-cli) @ App3 ---
0 2 4 7 -2

BUILD SUCCESS



1
2
3
4
5
s
0
∞
6
7
5
–2
8
–3
–4
7
2
9
4
–2
(a)
(b)
(c)

EX: Final



Explanation:

- The algorithm performs **V - 1** iterations over all edges to relax them.
- In each iteration, it checks all **E** edges.
- So, total work is approximately (V - 1) * E, which simplifies to **O(V × E)**.

- Bellman-Ford maintains an array to store the shortest path distances from the source vertex to each of the V vertices.
- The array size is V, and the space used by this array is O(V).

Algorithm	Time Complexity	Space Complexity
Bellman-Ford	O(V × E)	O(V)