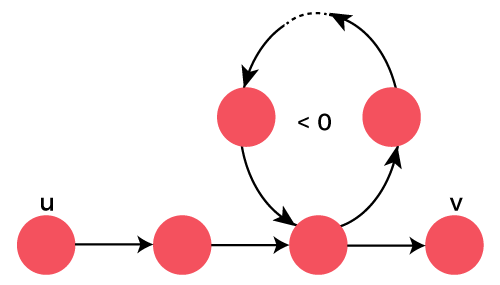
# Bellman-Ford Algorithm Java

In **dynamic programming**, there are many algorithms to find the shortest path in a graph. Some of them are **Dijkstra's algorithm, BFS, DFS, Floyd, all-pair shortest path problem,** and **bidirectional** **algorithm**. The most commonly used algorithm is **Dijkstra's algorithm.** The limitation of the algorithm is that it cannot be applied if the graph has negative edge weights. Another difference is that the Dijkstra algorithm looks only to the immediate neighbors of a vertex, Bellman-Ford goes through each edge in every iteration.

To overcome this problem, the **Bellman-Ford algorithm** can be applied. It deals with the negative edge weights. In this section, we will understand the Bellman-Ford algorithm with example and also **implement the Bellman ford algorithm in a Java program.**

## Bellman-Ford Algorithm

It is a single-source shortest path (minimum weight) algorithm very similar to Dijkstra's algorithm. It can be applied in a graph if we want to find the shortest path. Note that it deals with the **negative edge weights**. The limitation of the algorithm is that there should not be **negative cycles** (a cycle whose sum of edges produces a negative value) in the graph. Consider the following graph with cycle.



The runtime complexity of the algorithm is **O(v\*e)** and space complexity is **O(v).** One should use the algorithm if the graph has negative edge weights. Therefore, the Bellman-Ford algorithm can be applied in the following situations:

* Negative Edge Weights
* Positive Edge Weights > 1
* Undirected Cycles

The algorithm is slower than Dijkstra's algorithm when all arcs are negative.

## Working of The Bellman-Ford Algorithm

The working of the Bellman-Ford algorithm is the same as Dijkstra's algorithm. The only difference is that it does not use the priority queue. It repetitively loops over all the edges and updates the distances at the start node, the same as in Dijkstra's algorithm.

If a graph **G=(V, E)** contains a negative weight cycle, then some shortest paths may not exist. Bellman-Ford algorithm finds all shortest path lengths from a source **s ϵ V** to all **v ϵ V** or determines that a negative weight cycle exists.

Given a weighted directed graph G(V, E) with source (s) and weight function **w: E -> R**, the algorithm returns a boolean value **TRUE** if and only if the graph contains no negative-weight cycles that are reachable from the source. The algorithm produces the shortest path and its weights. If there is such a cycle, the algorithm indicates that **no solution exists**.

## Algorithm

1. INITIALIZE-SINGLE-SOURCE (G, s)
2. **for** i <- 1 to |V[G]|-1
3. **do** **for** each edge (U, V) ϵ E[G]
4. **do** RELEAX (U, V, W)
5. **for** each edge (U, V) ϵ E[G]
6. **do** **if** d[V] > d[U] + w (U, V)
7. then **return** FALSE
8. **return** TRUE

## Bellman Ford Pseudo Code

1. BellamanFord (G, V, E, S)
2. Step 1:
3. **for** each vertex v ϵ G **do**
4. dist[v] = ∞
5. dist[source] = 0
7. Step 2:
8. Relax (u, v, w)
9. **for** i = 1 to |v|-1
10. **for** each edge (u, v) ϵ G **do**
11. (dist[v] > dist[u]+w(u, v))
12. (dist[v] <- dist[u]+w(u, v))
13. Step 3:
14. **for** each edge (u, v) ϵ G
15. **if** (dist[u]+w(u, v)< dist[v])
16. **return** "Graph contains negative weight cycle"
17. **return** distance

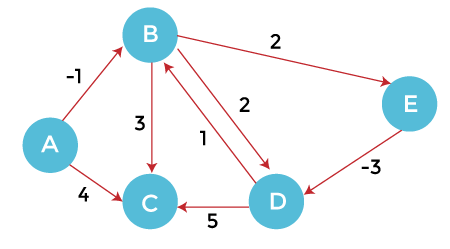
## Complexity

| **Time Complexity** | Best Case | O(E) |
| --- | --- | --- |
| Average Case | O(VE) |
| Worst Case | O(VE) |
| **Space Complexity** | | O(V) |

Let's understand the algorithm with an example.

## Bellman-Ford Example

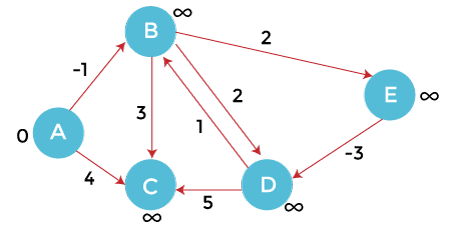
Consider the following directed graph (G).

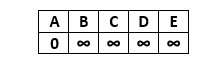


Order of edges: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D)

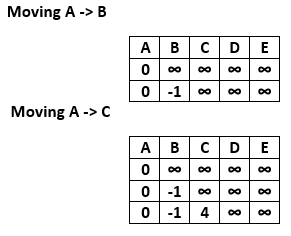
In the above graph (G), A is the vertex node for all other vertexes.

In order to find the shortest path, first, we will initialize the source vertex (A) as 0 and other vertices with infinity (**∞**). After that, we will traverse towards each vertex from the source node. Update the value of the node during the traversal. We have created the following table for distance updation.

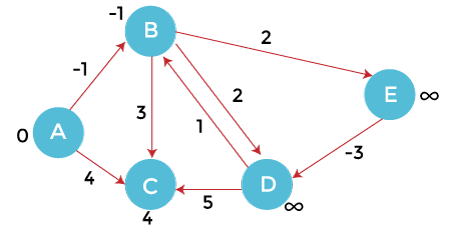




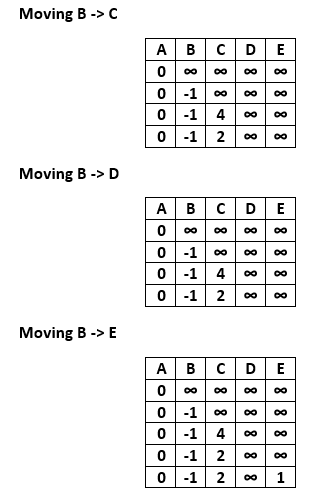
From the source vertex A, we can move to vertex B and C.



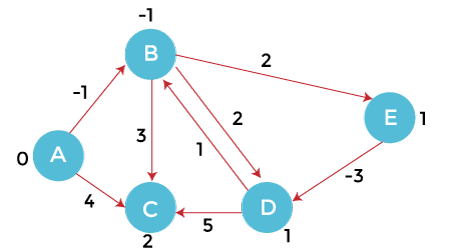
After updating the distances, we get the following graph.



From vertex B, we can move to vertex C, D and E. Calculate the distance from B to other vertices, we get



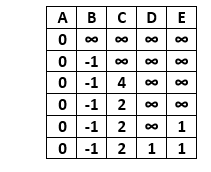
After updating the distances, we get the following graph.



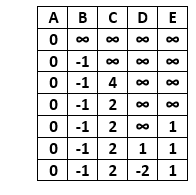
From vertex C we cannot move to any vertex, so we will visit the next vertex i.e. D.

From vertex D, we can move to vertex B and C. Calculate the distance from vertex D to other vertices.

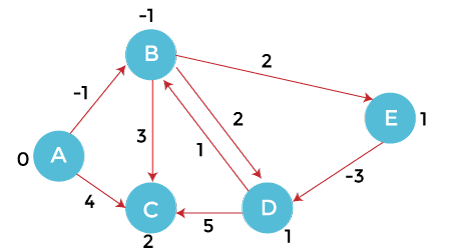
**Moving D -> B,** we observe that the vertex B is already has the minimum distance, so we will not update the distance at this time.



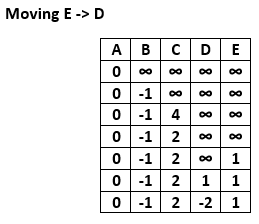
**Moving D-> C,** we observe that the vertex C already has the minimum distance, so we will not update the distance at this time.



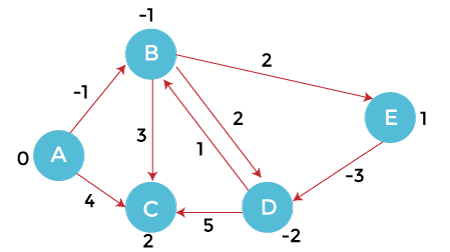
After updating the distances, we get the following graph.



From vertex E, we can move to vertex D only. Calculate the distance from vertex E to D.



After updating the distances, we get the following graph.



We observe that values decrease monotonically.