**Bellman-Ford Algorithm**

Bellman-Ford algorithm is a procedure used to find all shortest path in a graph from one source to all other nodes. The algorithm requires that the graph does not contain any cycles of negative length, but if it does, the algorithm is able to detect it.

The algorithm was introduced by American mathematicians Richard Bellman and Lester Ford

**Description**

The Bellman-Ford algorithm is based on the relaxation operation. The relaxation procedure takes two nodes as arguments and an edge connecting these nodes. If the distance from the source to the first **node (A)** plus the edge length is less than distance to the second node, than the first node is denoted as the predecessor of the second node and the distance to the second node is recalculated **(distance(A) + edge.length).** Otherwise no changes are applied.

Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.  
We have discussed [Dijkstra’s algorithm](https://www.geeksforgeeks.org/archives/27697) for this problem. Dijksra’s algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). Dijkstra doesn’t work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

**Following are the detailed steps:**

**Input:** Graph and a source vertex src  
**Output:** Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

**Steps:**

**1)** This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.

**2)** This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.

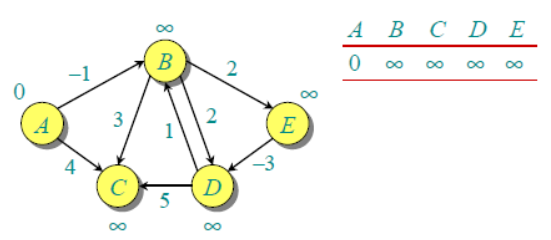
**Do following for each edge u-v**If dist[v] > dist[u] + weight of edge uv, then update dist[v]  
 dist[v] = dist[u] + weight of edge uv

**3)** This step reports if there is a negative weight cycle in graph.   
 **Do following for each edge u-v** If dist[v] > dist[u] + weight of edge uv,   
 Then “Graph contains negative weight cycle”

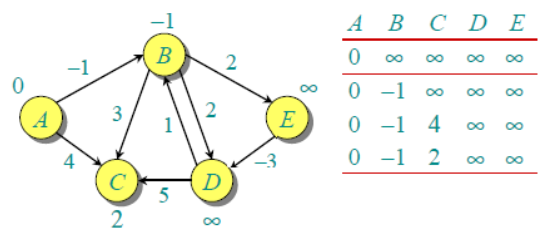
The idea of step 3 is, step 2 guarantees shortest distances if graph doesn’t contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

**How does this work?**

Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates shortest paths with at-nost 2 edges, and so on. After the i-th iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges

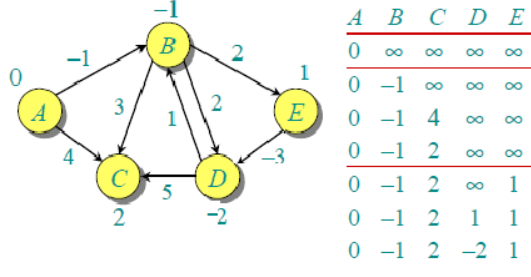
**Example**

Let us understand the algorithm with following example graph. The images are taken from this source. Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so all edges must be processed 4 times.



Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances.

The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.

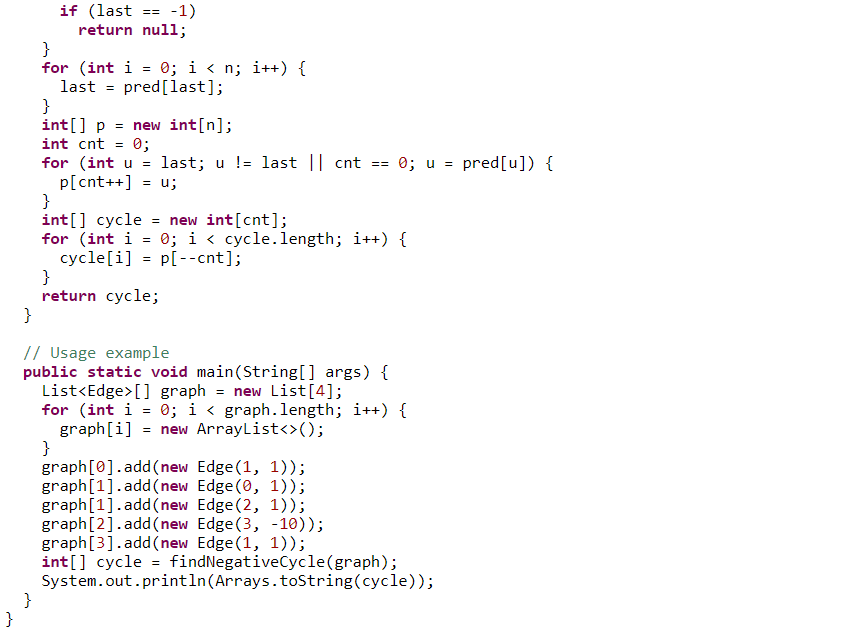


The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).

The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don’t update the distances.

**Shortest paths. Bellman–Ford algorithm in O(V\*E). Negative cycle detection.**

****

****