Bellman-Ford Algorithm

The bellman-ford algorithm is an algorithm that finds the shortest path between a given source vertex and all other vertices in the graph. This algorithm can be used on both weighted and unweighted graphs. Like Dijkstra's shortest path algorithm, the bellman-ford algorithm is guaranteed to find the shortest path in a graph.

If the graph has negative edge costs, then Dijkstra's algorithm does not work. Although it is computationally more expensive than Dijkskta's algorithm, Bellman -ford is capable of handling graphs that contain **negative edge weights**, so it is more versatile.
It is worth noting that if there exists a negative cycle in the graph then there is no shortest path. Going around the negative cycle an infinite number of parents would continue to decrease the cost of the path(even though the path length is increasing). Because of this reason, Bellman-Ford can also **detect negative cycles** which is a useful feature.

The Bellman-Ford algorithm operates on the graph input G with V vertices and E edges. A single source vertex S must be provided as well, as the Bellman-Ford Algorithm is a single source shortest path algorithm. No destination vertex needs to be supplied, however, because bellman-ford calculates the shortest distance to all vertices in the graph from the source vertex.

The Bellman-Ford algorithm, like Dijkstra's algorithm, uses the principle of relaxation to find increasing accurate path length. Bellman-Ford though tackles two main issues with this process:

- If there are negative weight cycles, the search for the shortest path will go on forever.
- Choosing a bad ordering for relaxations leads to exponential relaxations.

The detection of the negative cycles is important but the main contribution of this algorithm is in its ordering of relaxations. Relaxation is the most important step in Bellman-Ford. It is what increases the accuracy of the distance to any given vertex. Relaxation works by continuously shortening the calculated distance between two vertices by comparing that distance with other known distances.

Bellman-Ford algorithm works by overestimating the length of the path from the starting vertex to all other vertices. Then it iteratively relaxes those estimates by finding new paths that are shorter than the previously overestimated paths.

Example: Let's say I think the distance to the baseball stadiums is 30 miles. However, I know that the distance to the corner right before the stadium is 15 miles and I know that from the corner to the stadium the distance is 1 mile. Clearly, the distance from me to the stadium is at most 16 miles. So, I can update my belief to reflect that. That is one cycle of relaxation, and it's done over and over until the shortest paths are found.

The problem with Djikstra's algorithm is that once a vertex u is declared known, it is possible that from some other, unknown vertex v there is a path back to u that is very negative. In such a case,

taking a path from s to v back to u is better than going from s to u without using v. A combination of Dijkstra's algorithm and an unweighted algorithm will solve the problem. Initialize the queue with s. Then, at each stage, we DeQueue a vertex v. We find all vertices w adjacent to v such that,

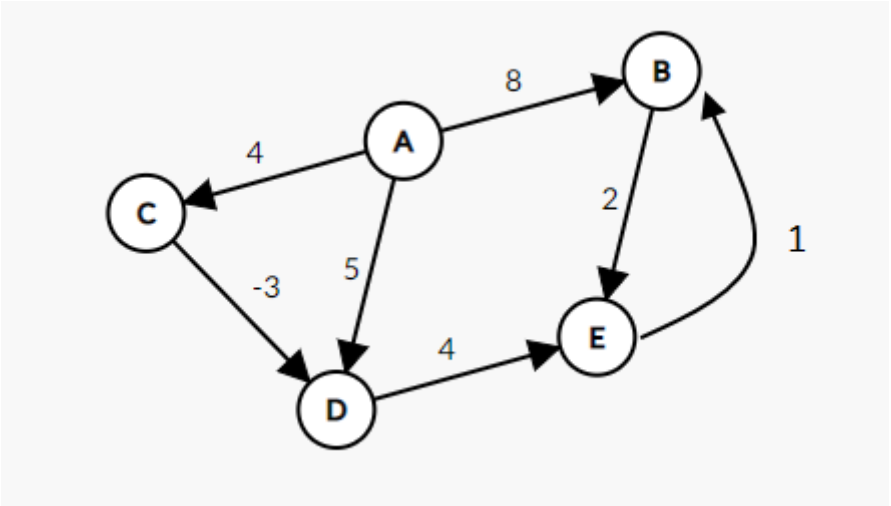
Distance to v + weight(v,w) < old distance to w

We update w's old distance and path, and place w on a queue if it is not already there. A bit can be a set for each vertex to indicate the presence in the queue. We repeat the process until the queue is empty.

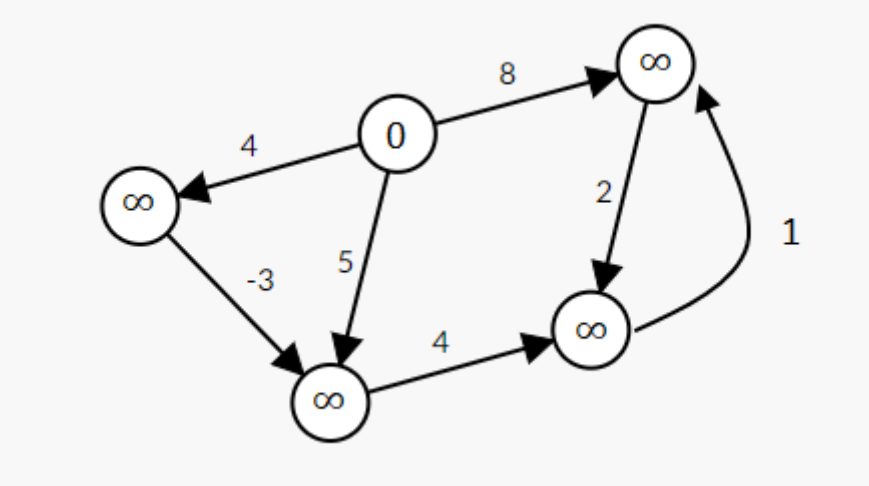
```
function bellmanFord(Graph g, s)
{
		//   create a new hashmap of string v/s Integer
		HashMap<String, Integer> map ;

		//   fill hashmap with large value say 100000 across all vertices
		for(every vertex of graph) {
			map.put(vertexName, 100000);
		}

		V = total number of vertices

		//   relax every edge V-1 times
		for ( i = 1; i <= V; i++) {

			//   loop on edges
			for (every edge of graph G with v1 and v2 as end points and weight
w) {

				//   get the old cost from the map
				int oc = map.get(v1);
				int nc = map.get(v2) + w;

				if (oc > nc) {

					if (i <= V - 1)
						//   put the new weight across v2
						map.put(v2, nc);
					else
						throw new Exception("-ve wt cycle present");
				}

			}

		}
```

```
                return map;
}
```

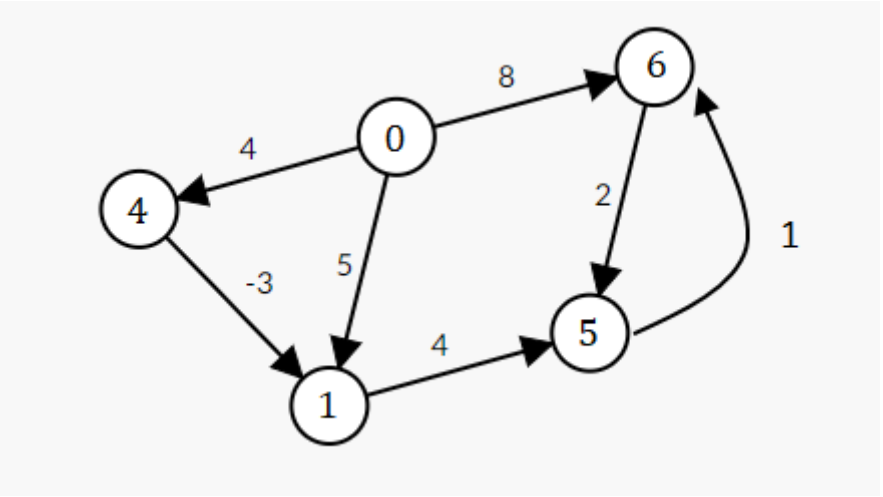
Time Complexity: O(VE), we are iterating on E edges V times.
Example: Consider the graph below



Choose a starting vertex let A and assign infinity value to all other vertices.



For V=1, the graph obtained will be



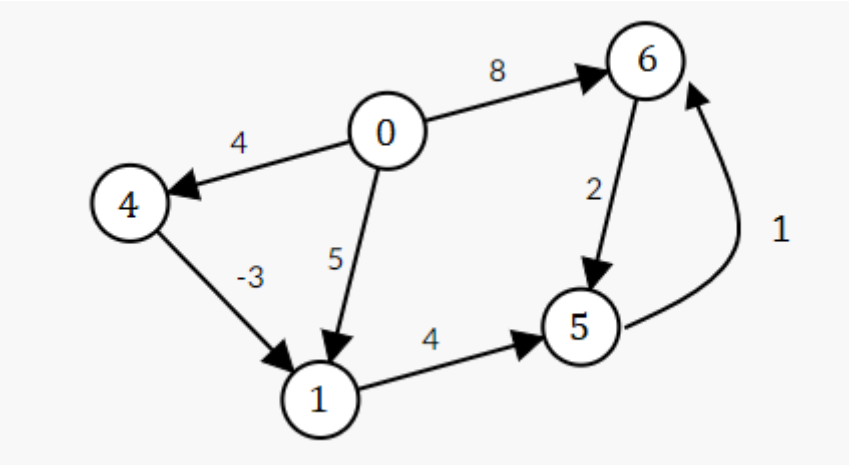
How?
We have the edges as

1. A -> B : 8
2. A -> C : 4
3. A -> D : 5
4. C -> D : -3
5. D -> E : 4
6. B -> E : 2
7. E -> B : 1

Now consider the following steps while iterating over the edges and see how the above graph is obtained.

- For the edge-1 B is explored at a cost/weight of 8 (Cost of exploring A as stored in map + cost of the edge between A and B). Similarly, C and D are explored at a cost of 4 and 5 using edge-2 and edge-3 respectively.
- Now using edge-4 C->D: -3, we can say that D is now explored at a cost of 1. How? By adding the cost of exploring C from the map and cost of the edge between C and D, i.ie, 4+(-3)=1.
- Now for edge-5, E can be explored at a cost of 5 which is the cost of exploring D + cost of the edge between E and D (1 + 4).
- For edge-6, the cost of exploring B = 8, adding the edge cost of the edge from B to E, we get 10 which is less than the previous cost of exploring E, so no update.
- Cost of exploring E = 5, cost of edge from E to B = 1, total cost = 6 which is less than B existing cost which is 8, so update the cost of B in map.

Now, you can further calculate for V=2, V=3, and V=4.
The final graph will look like



Now, what if our graph changes at V=5?
It indicates that the graph contains a negative weight cycle because in the worst case, a vertex path length might be needed to be readjusted V-1 times.