Dijkstra's Algorithm Explanation with example

Dijkstra's algorithm, given by a brilliant Dutch computer scientist and software engineer Dr. Edsger Dijkstra in 1959. Dijkstra's algorithm is a greedy algorithm that solves the single-source shortest path problem for a directed and undirected graph that has non-negative edge weight.

For Graph G = (V, E)

$w(u, v) \geq 0$ for each edge $(u, v) \in E$.

This algorithm is used to find the shortest path in Google Maps, in network routing protocols, in social networking applications, and also in many places. For a given graph G Dijkstra's algorithm is helpful to find the shortest path between a source node to a destination node.

For example, if we draw a graph in which nodes represent the cities and weighted edges represent the driving distances between pairs of cities connected by a direct road, then Dijkstra's algorithm when applied gives the shortest route between one city and all other cities. This algorithm is also used in GPS devices to find the shortest path between the current location and the destination. It has broad applications in industry, specially in domains that require modeling networks.

Dijkstra's algorithm differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

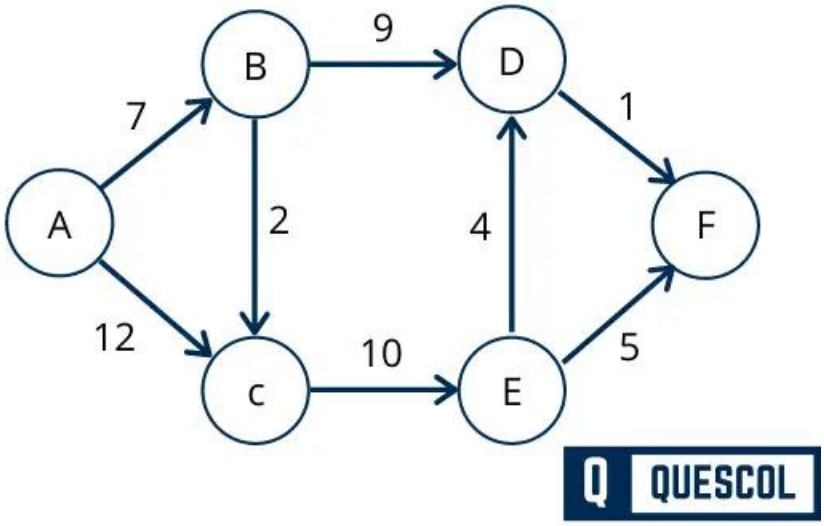
Dijkstra's Algorithm

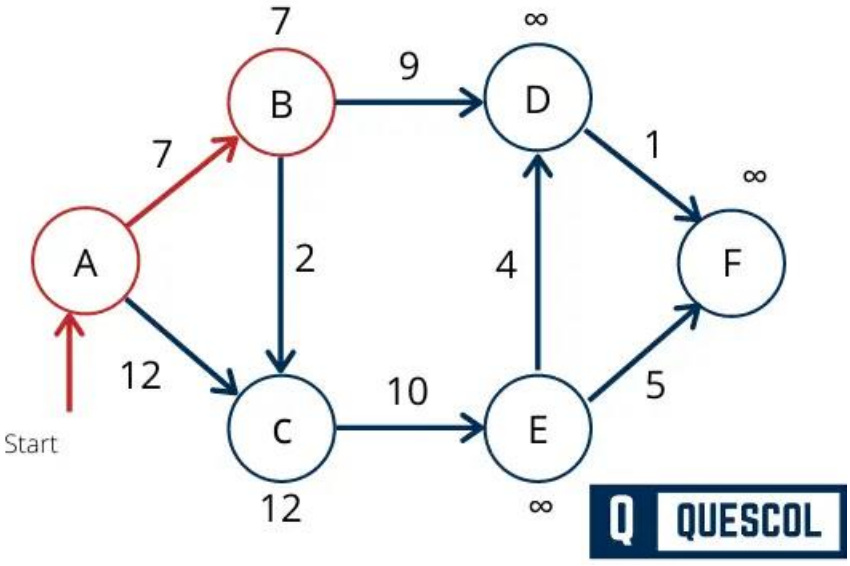
1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3. While sptSet doesn't include all vertices
   - Pick a vertex u which is not there in sptSet and has a minimum distance value.
   - Include u to sptSet.

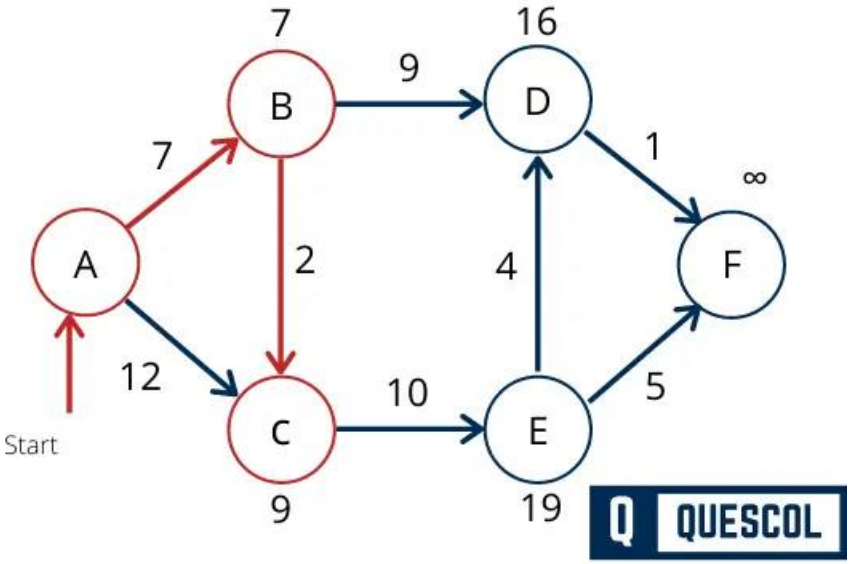
- Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of a distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v. (Algo source : geeksforgeeks.com)

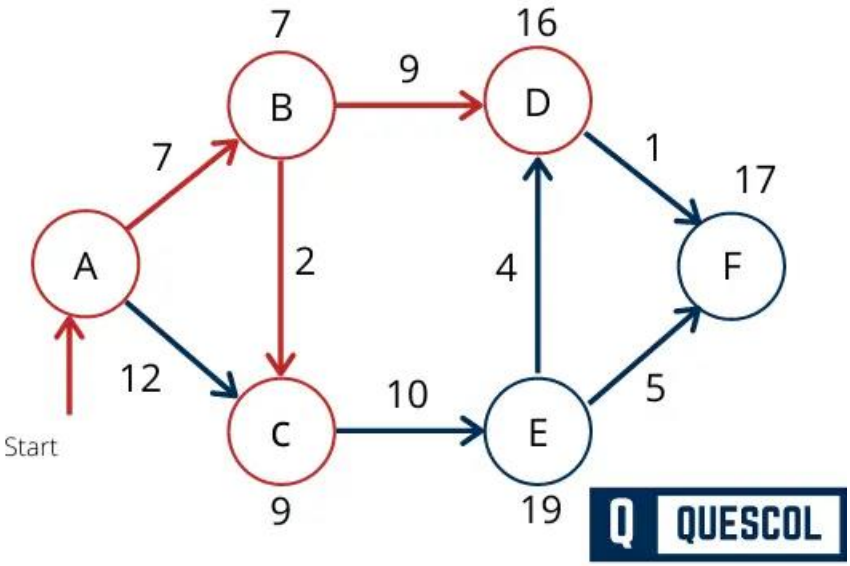
Lets see an example to understand Dijkstra's Algorithm

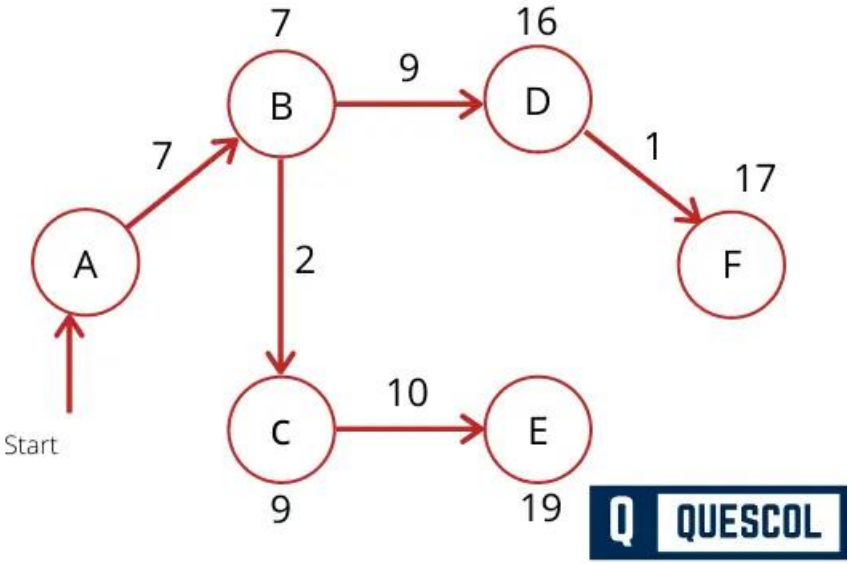
Below is a directed weighted graph. We will find shortest path between all the vertices using Dijkstra'a Algorithm.






7
16
9
B
D
7
1
∞
A
2
4
F
12
10
5
C
E
Start
9
19
QUESCOL


7
16
9
B
D
7
1
17
A
2
4
F
12
10
5
C
E
Start
9
19
QUESCOL



Dijkstra's Algorithm Applications

- To find the shortest path between source and destination
- In social networking applications to map the connections and information
- In networking to route the data
- To find the locations on the map

Disadvantage of Dijkstra's Algorithm

- It follows the blind search, so wastes a lot of time to give the desired output.
- It Negative edges value cannot be handled by it.
- We need to keep track of vertices that have been visited.

Bellman Ford's Algorithm

Bellman Ford algorithm helps us find the shortest path from a vertex to all other vertices of a weighted graph.

It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

Why would one ever have edges with negative weights in real life?

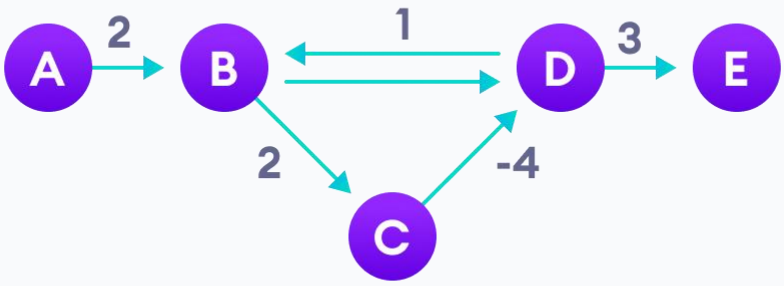
Negative weight edges might seem useless at first but they can explain a lot of phenomena like cashflow, the heat released/absorbed in a chemical reaction, etc.

For instance, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption.

If we want to find the set of reactions where minimum energy is required, then we will need to be able to factor in the heat absorption as negative weights and heat dissipation as positive weights.

Why do we need to be careful with negative weights?

Negative weight edges can create negative weight cycles i.e. a cycle that will reduce the total path distance by coming back to the same point.



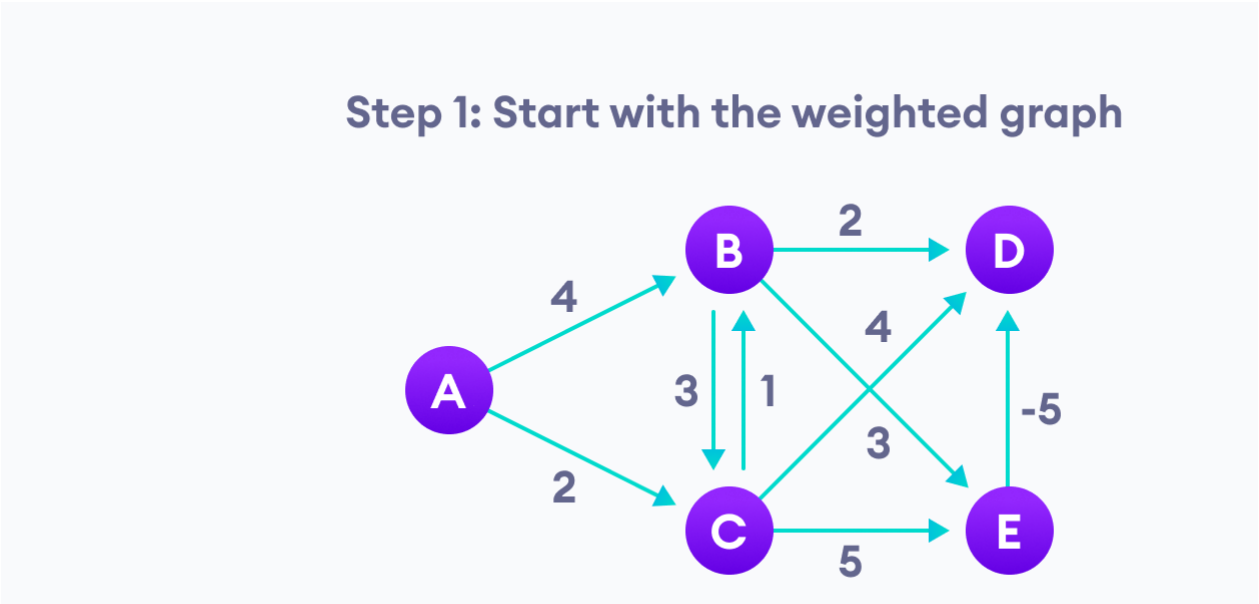
Negative weight cycles can give an incorrect result when trying to find out the shortest path

Shortest path algorithms like Dijkstra's Algorithm that aren't able to detect such a cycle can give an incorrect result because they can go through a negative weight cycle and reduce the path length.

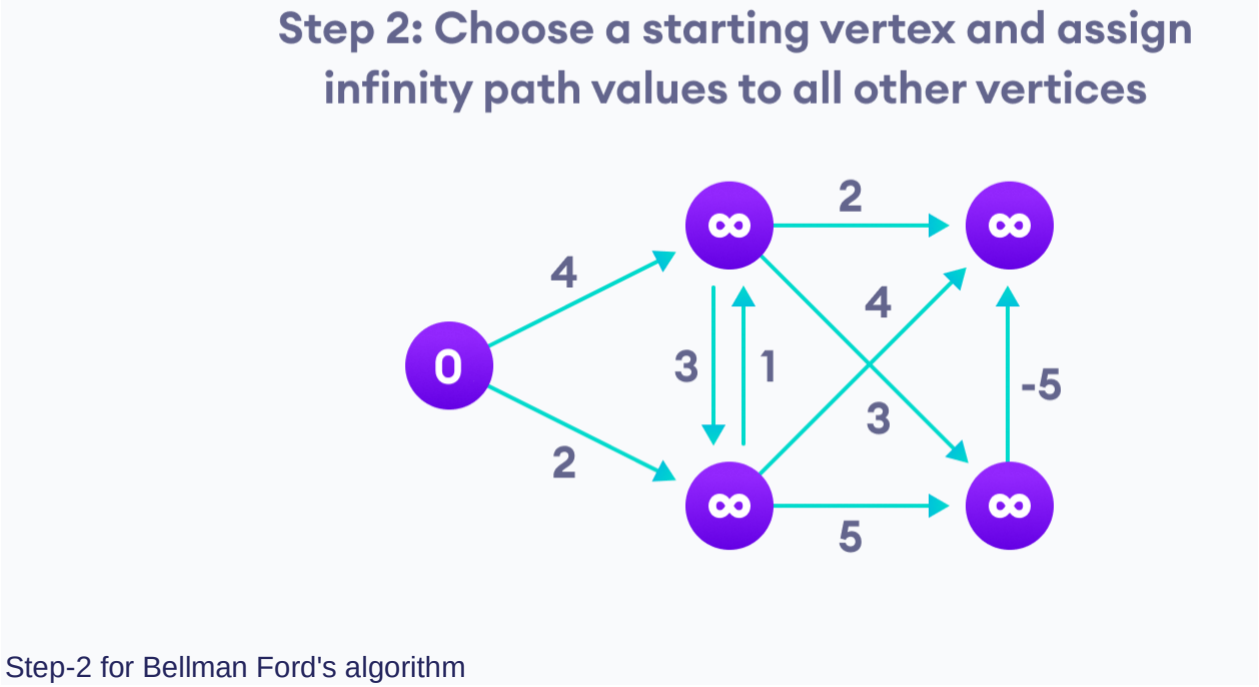
How Bellman Ford's algorithm works

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.

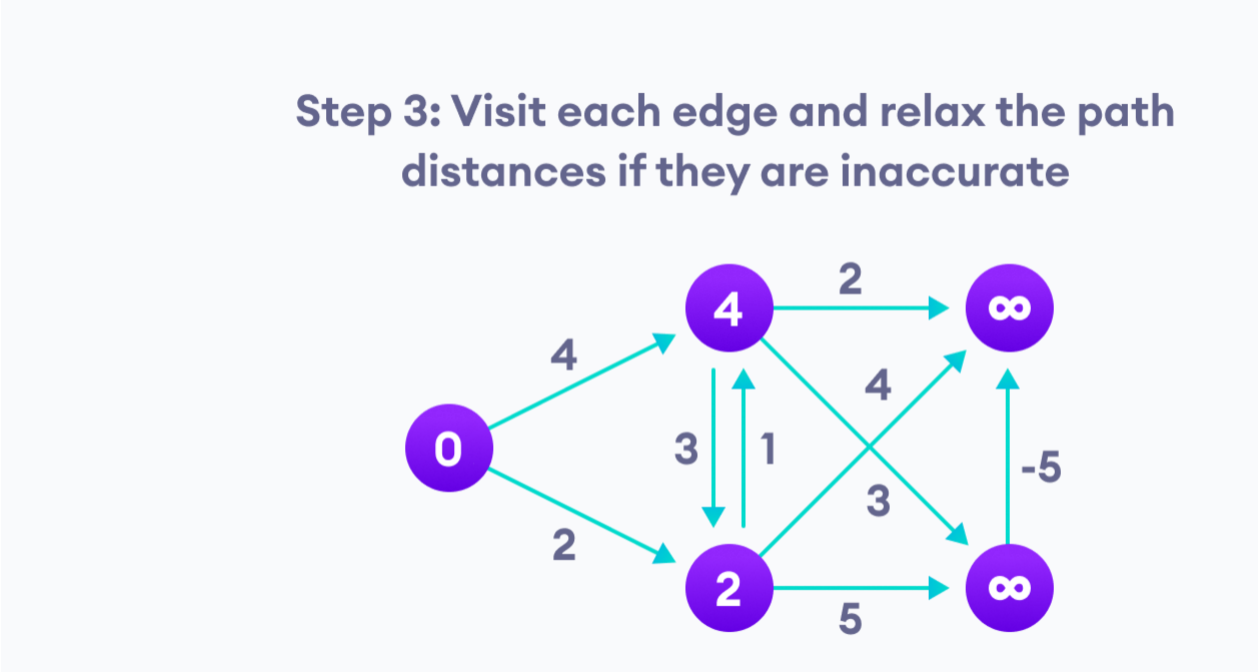
By doing this repeatedly for all vertices, we can guarantee that the result is optimized.



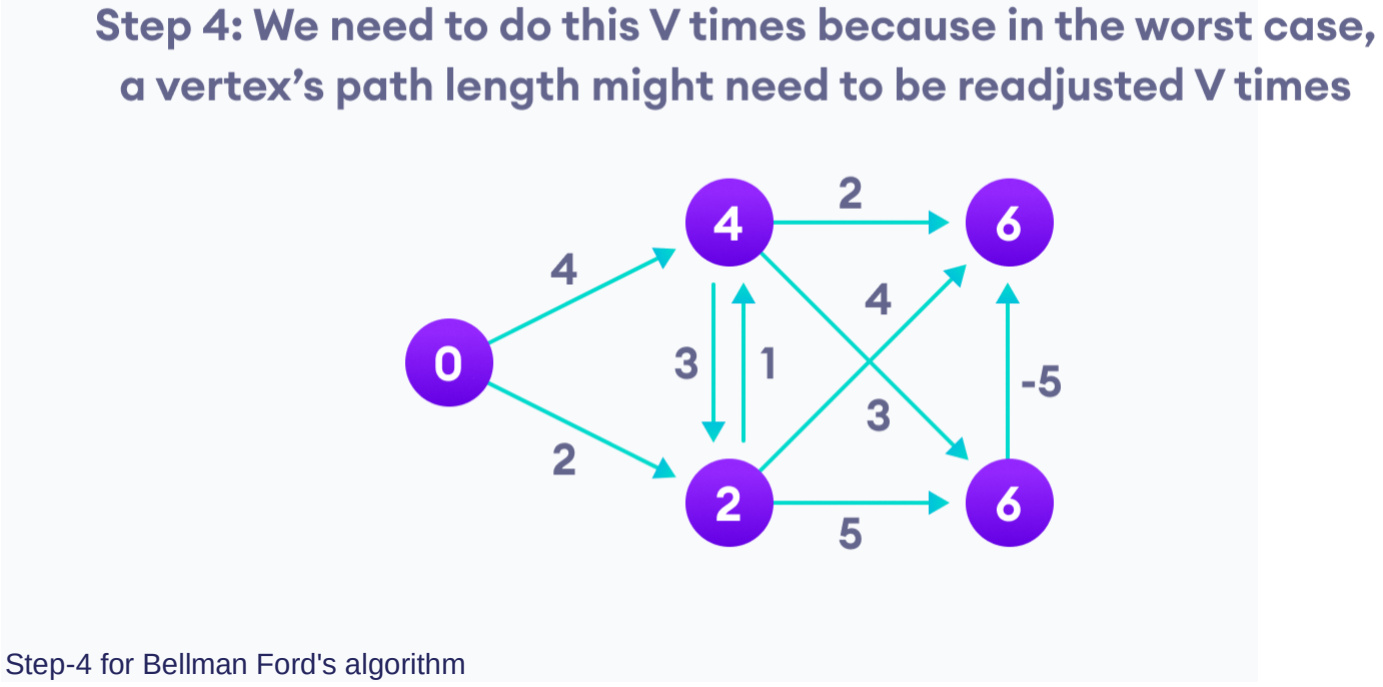
Step-1 for Bellman Ford's algorithm



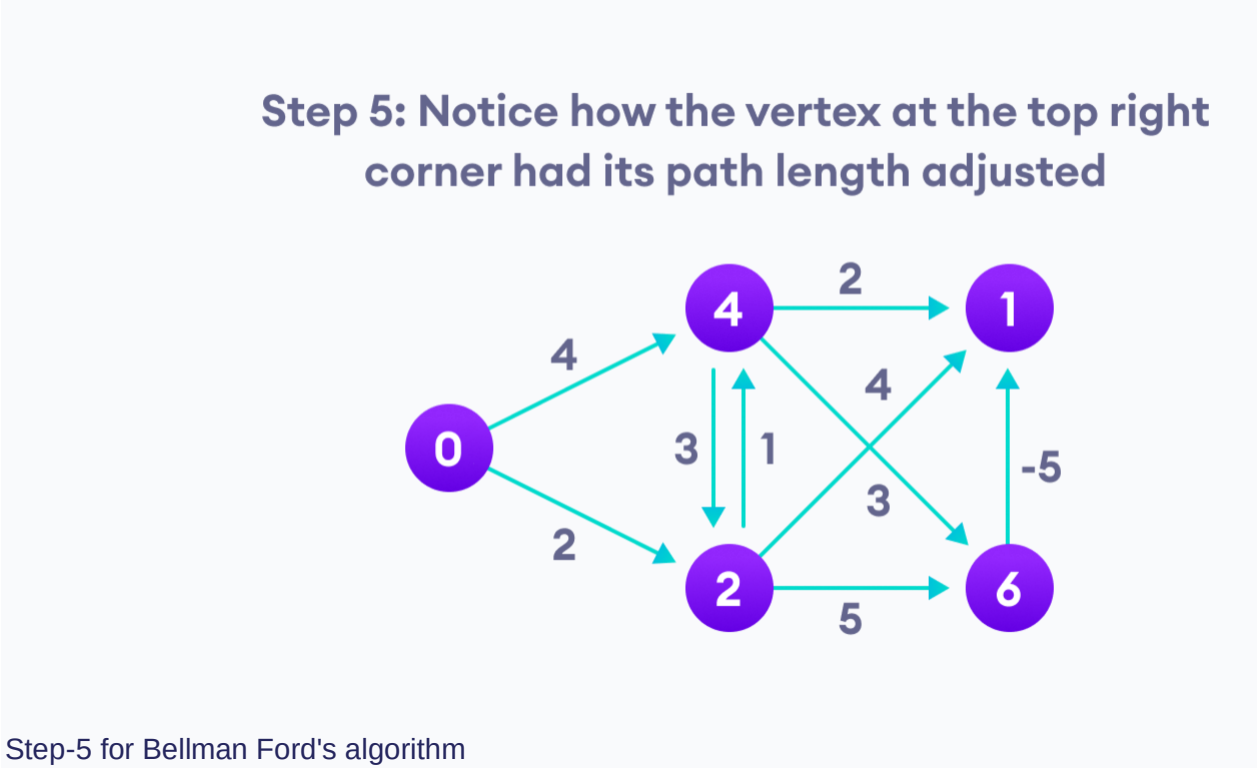
Step-2 for Bellman Ford's algorithm



Step-3 for Bellman Ford's algorithm



Step-4 for Bellman Ford's algorithm



Step-5 for Bellman Ford's algorithm

Step 6: After all the vertices have their path lengths, we check if a negative cycle is present

	B	C	D	E
0	∞	∞	∞	∞
0	4	2	∞	∞
0	3	2	6	6
0	3	2	1	6
0	3	2	1	6

Step-6 for Bellman Ford's algorithm

Bellman Ford Pseudocode

We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

```
function bellmanFord(G, S)
  for each vertex V in G
    distance[V] <- infinite
      previous[V] <- NULL
  distance[S] <- 0


  for each vertex V in G
    for each edge (U,V) in G
      tempDistance <- distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]
        distance[V] <- tempDistance
        previous[V] <- U


  for each edge (U,V) in G
    If distance[U] + edge_weight(U, V) < distance[V}
      Error: Negative Cycle Exists


  return distance[], previous[]
```

Bellman Ford vs Dijkstra

Bellman Ford's algorithm and Dijkstra's algorithm are very similar in structure. While Dijkstra looks only to the immediate neighbors of a vertex, Bellman goes through each edge in every iteration.

```
function bellmanFord(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL

    distance[S] <- 0

    for each vertex V in G

        for each edge (U,V) in G
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    for each edge (U,V) in G
        If distance[U] + edge_weight(U, V) < distance[V}
            Error: Negative Cycle Exists

    return distance[], previous[]
```

Bellman Ford's Algorithm vs Dijkstra's Algorithm

Python, Java and C/C++ Examples

Python
Java
C

C++

```
// Bellman Ford Algorithm in Java

class CreateGraph {

  // CreateGraph - it consists of edges
  class CreateEdge {
    int s, d, w;

    CreateEdge() {
      s = d = w = 0;
    }
  };

  int V, E;
  CreateEdge edge[];

  // Creates a graph with V vertices and E edges
  CreateGraph(int v, int e) {
    V = v;
    E = e;
    edge = new CreateEdge[e];
    for (int i = 0; i < e; ++i)
      edge[i] = new CreateEdge();
  }

  void BellmanFord(CreateGraph graph, int s) {
    int V = graph.V, E = graph.E;
    int dist[] = new int[V];

    // Step 1: fill the distance array and predecessor array
    for (int i = 0; i < V; ++i)
      dist[i] = Integer.MAX_VALUE;

    // Mark the source vertex
    dist[s] = 0;

    // Step 2: relax edges |V| - 1 times
    for (int i = 1; i < V; ++i) {
      for (int j = 0; j < E; ++j) {
        // Get the edge data
        int u = graph.edge[j].s;
```

```
      int v = graph.edge[j].d;
      int w = graph.edge[j].w;
      if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v])
        dist[v] = dist[u] + w;
    }
  }

  // Step 3: detect negative cycle
  // if value changes then we have a negative cycle in the graph
  // and we cannot find the shortest distances
  for (int j = 0; j < E; ++j) {
    int u = graph.edge[j].s;
    int v = graph.edge[j].d;
    int w = graph.edge[j].w;
    if (dist[u] != Integer.MAX_VALUE && dist[u] + w < dist[v]) {
      System.out.println("CreateGraph contains negative w cycle");
      return;
    }
  }

  // No negative w cycle found!
  // Print the distance and predecessor array
  printSolution(dist, V);
}

// Print the solution
void printSolution(int dist[], int V) {
  System.out.println("Vertex Distance from Source");
  for (int i = 0; i < V; ++i)
    System.out.println(i + "\t\t" + dist[i]);
}

public static void main(String[] args) {
  int V = 5; // Total vertices
  int E = 8; // Total Edges

  CreateGraph graph = new CreateGraph(V, E);

  // edge 0 --> 1
  graph.edge[0].s = 0;
  graph.edge[0].d = 1;
  graph.edge[0].w = 5;
```

```
    // edge 0 --> 2
    graph.edge[1].s = 0;
    graph.edge[1].d = 2;
    graph.edge[1].w = 4;

    // edge 1 --> 3
    graph.edge[2].s = 1;
    graph.edge[2].d = 3;
    graph.edge[2].w = 3;

    // edge 2 --> 1
    graph.edge[3].s = 2;
    graph.edge[3].d = 1;
    graph.edge[3].w = 6;

    // edge 3 --> 2
    graph.edge[4].s = 3;
    graph.edge[4].d = 2;
    graph.edge[4].w = 2;

    graph.BellmanFord(graph, 0); // 0 is the source vertex
  }
}
```

Bellman Ford's Complexity

Time Complexity

Best Case Complexity	O(E)
Average Case Complexity	O(VE)
Worst Case Complexity	O(VE)

Space Complexity

And, the space complexity is `O(V)`.

Bellman Ford's Algorithm Applications

1. For calculating shortest paths in routing algorithms
2. For finding the shortest path