**Bellman Ford's Algorithm**

It is similar to [Dijkstra's algorithm](https://www.programiz.com/dsa/dijkstra-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?**

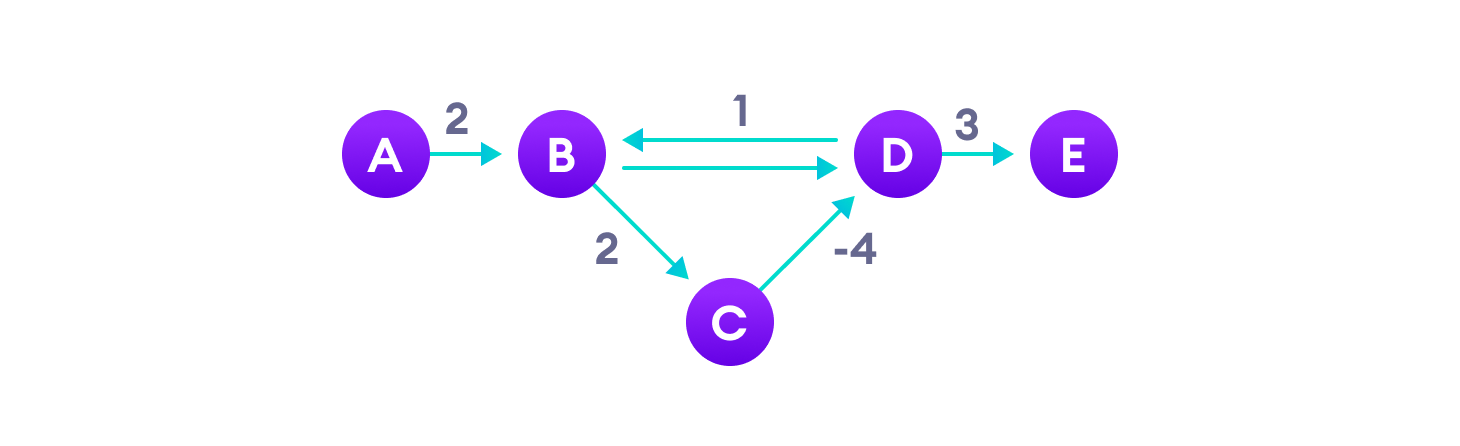
Negative weight edges might seem useless at first but they can explain a lot of phenomena like cashflow, the heat released/absorbed in a chemical reaction, etc.

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.

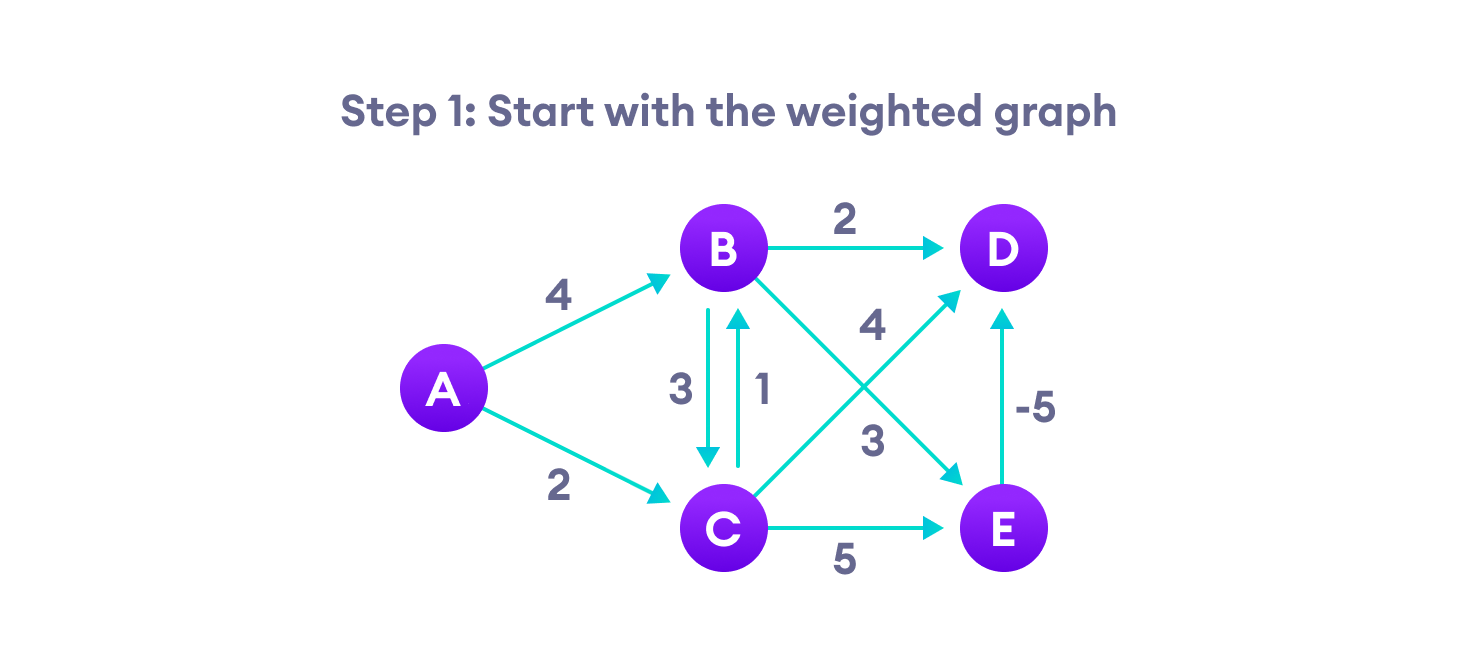
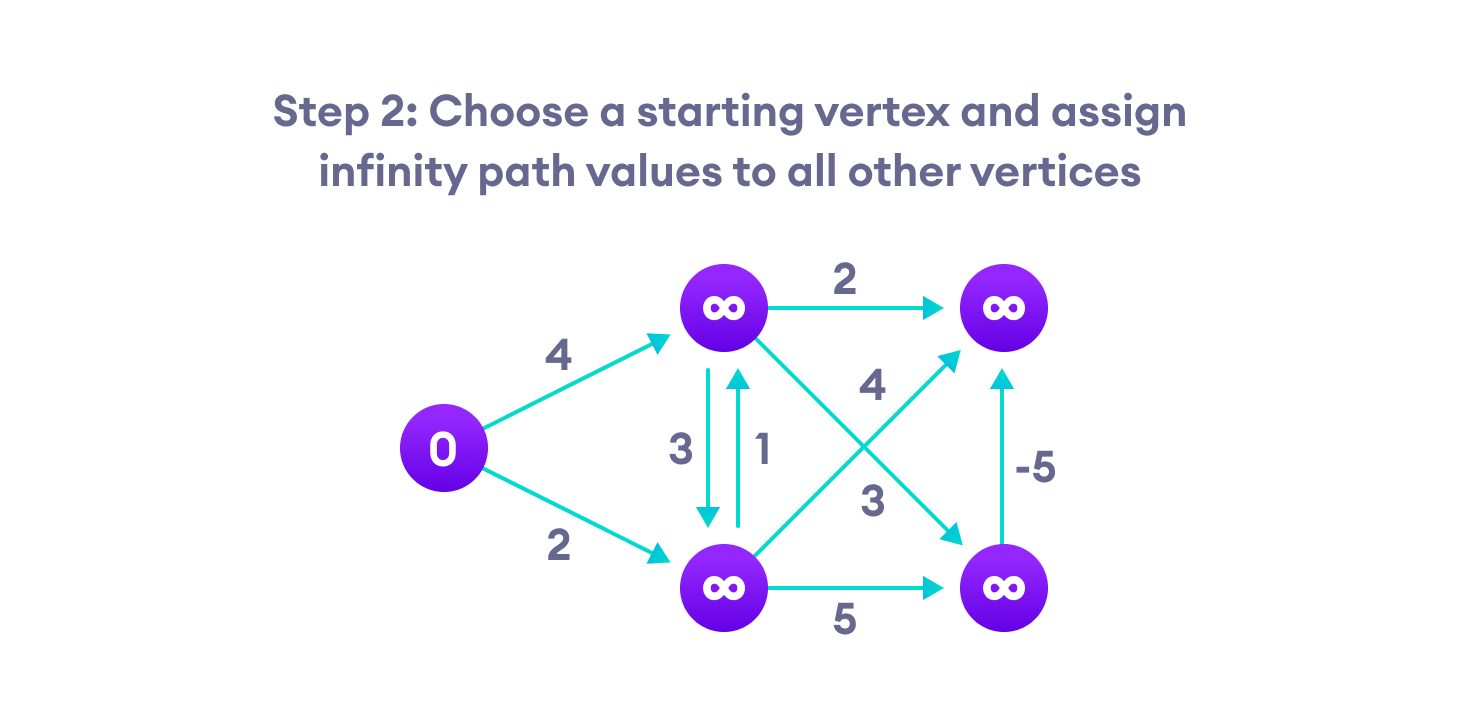
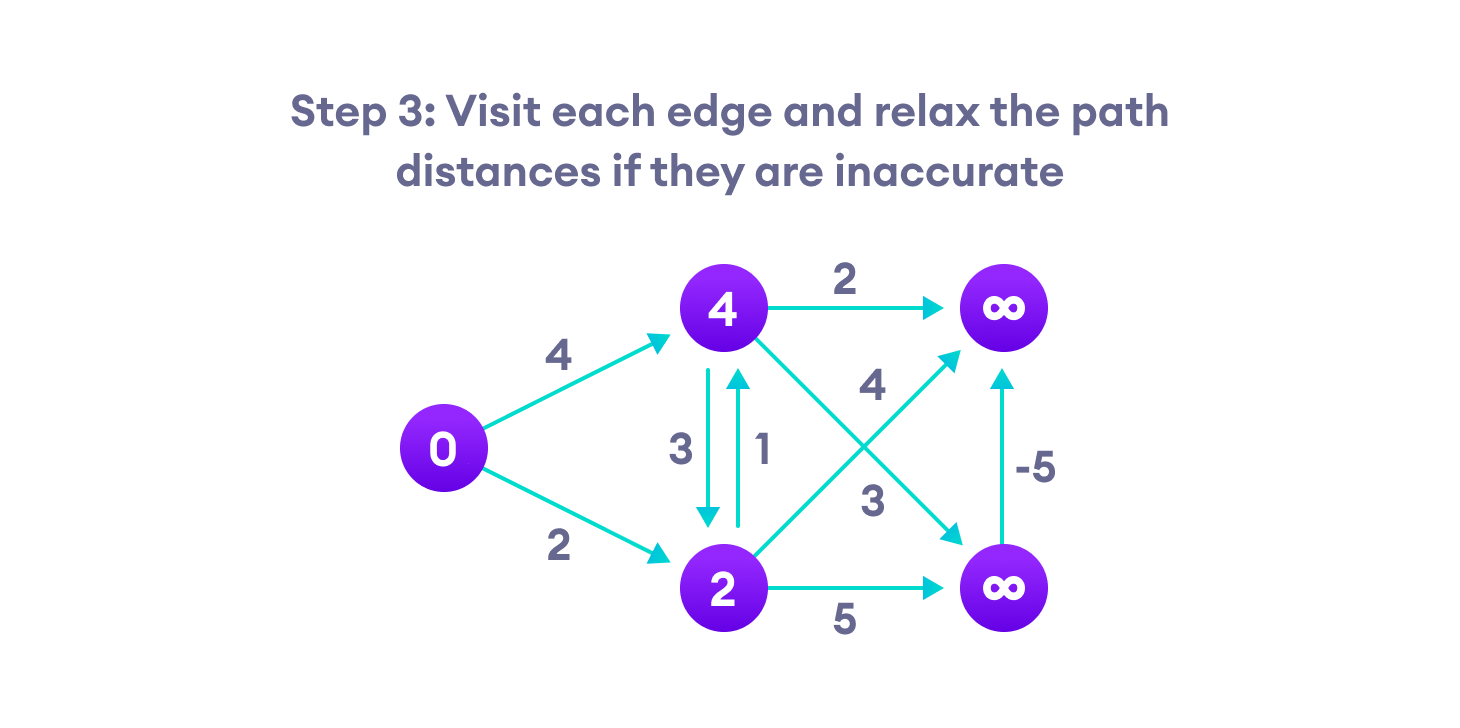
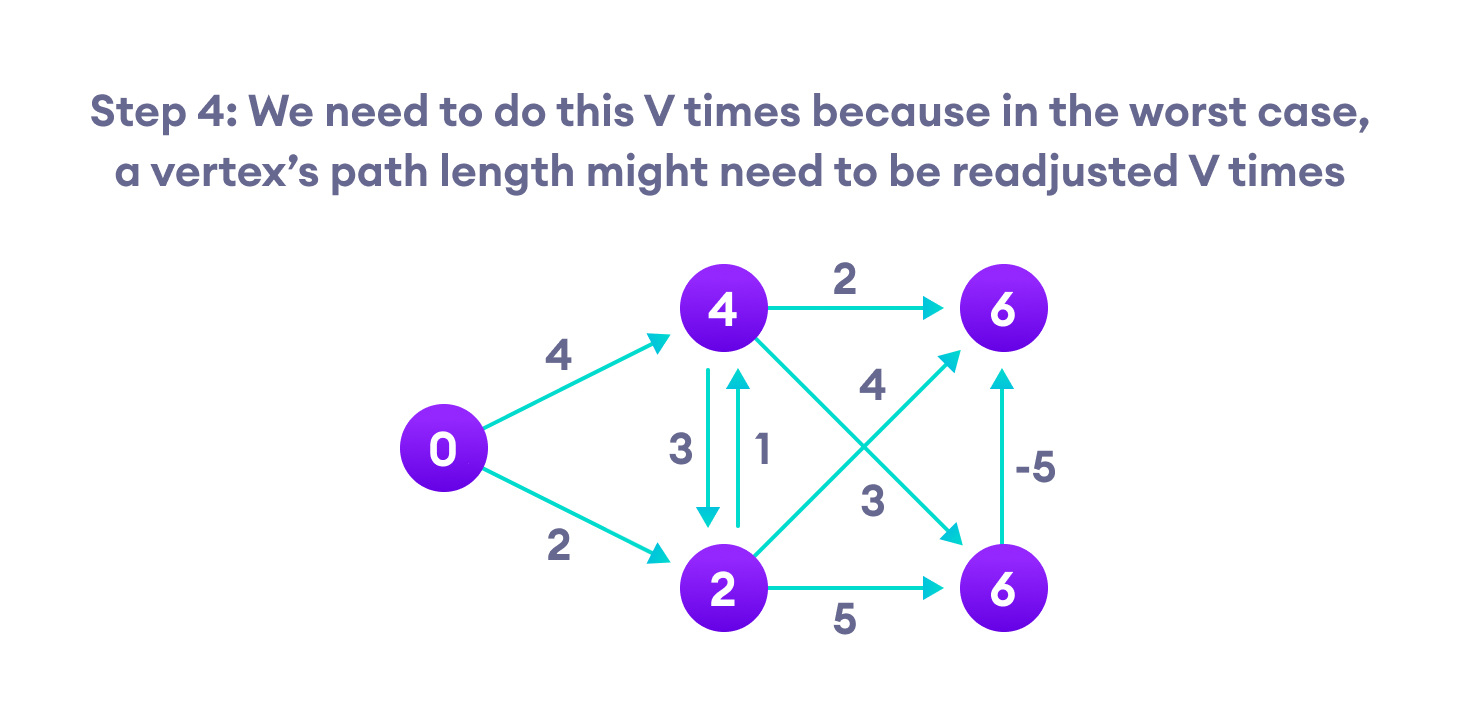
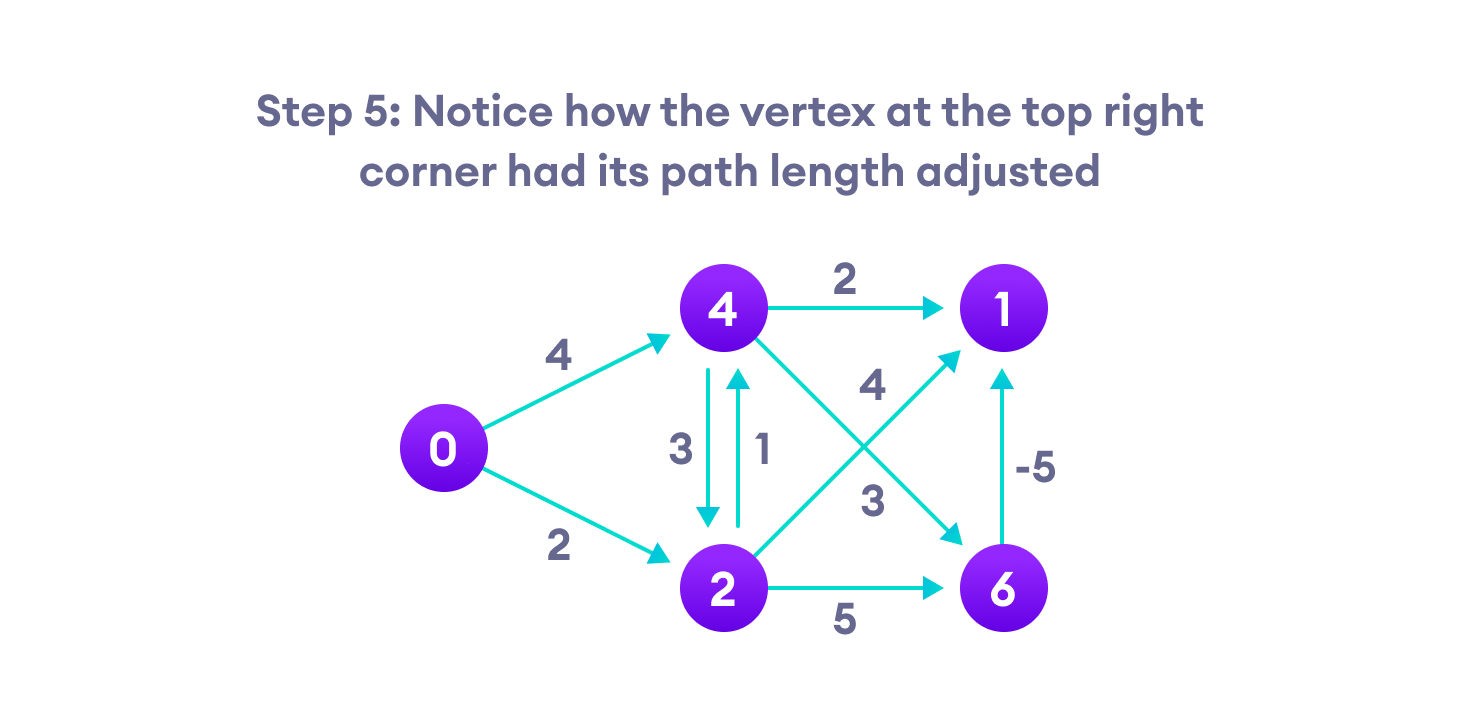
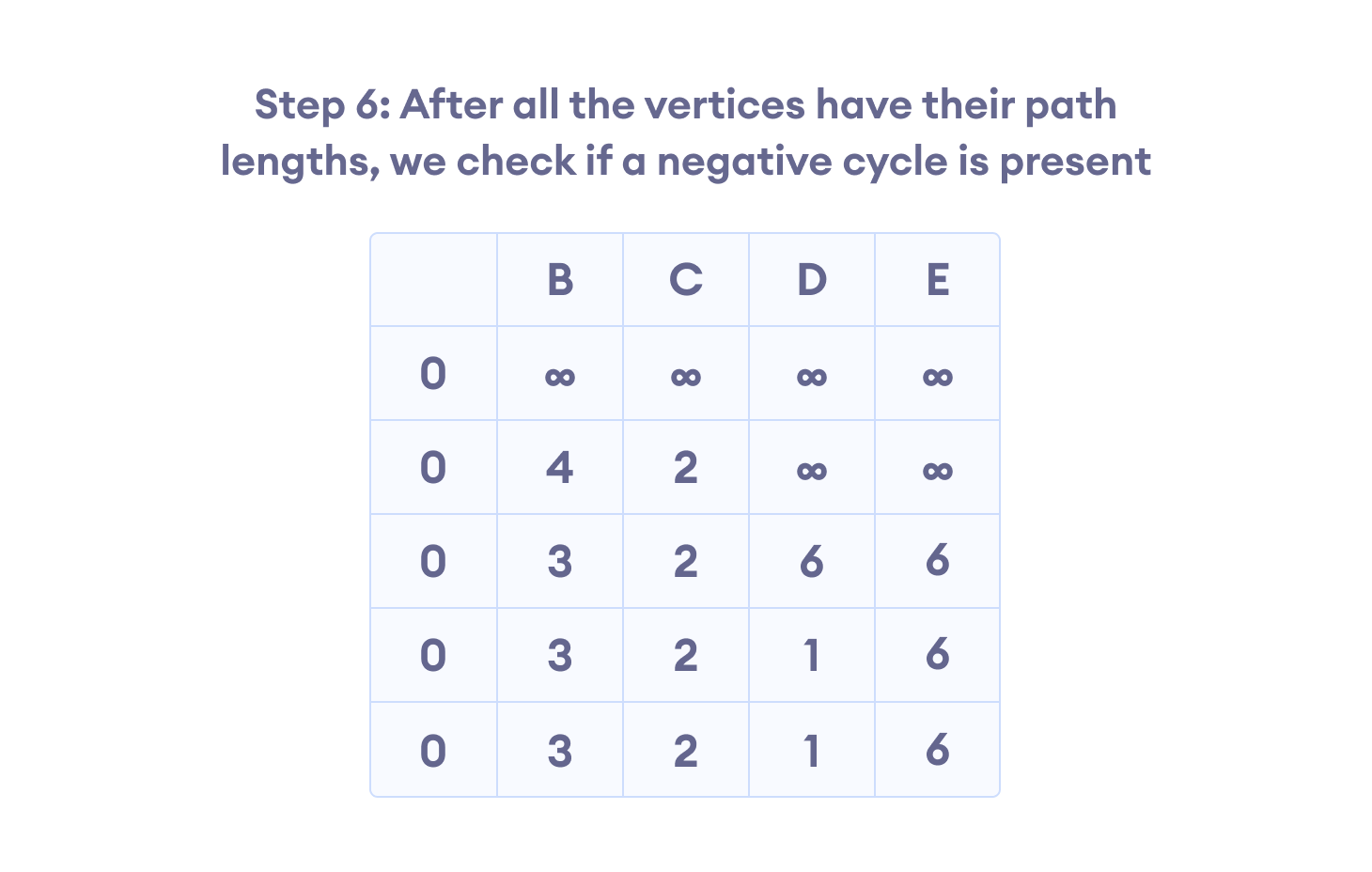
Negative weight cycles can give an incorrect result when trying to find out the shortest path

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 algorithmStep-2 for Bellman Ford's algorithmStep-3 for Bellman Ford's algorithmStep-4 for Bellman Ford's algorithmStep-5 for Bellman Ford's algorithmStep-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

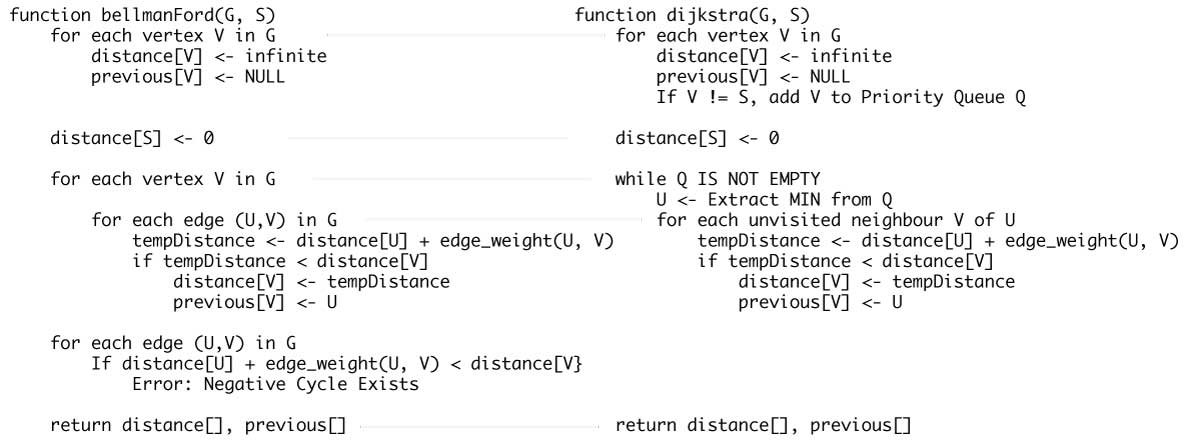
If distance[U] + edge\_weight(U, V) < distance[V}

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.

Bellman Ford's Algorithm vs Dijkstra's Algorithm

// Bellman Ford Algorithm in Java