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?

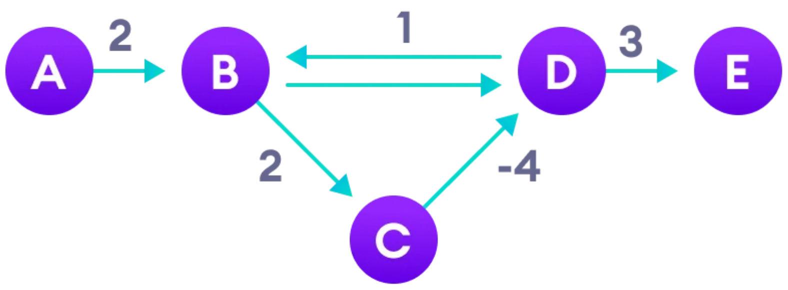
Negative weight edges might seem useless at first but they can explain a lot of phenomena like cashflow, the heat released/absorbed in a chemical reaction, etc.

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.



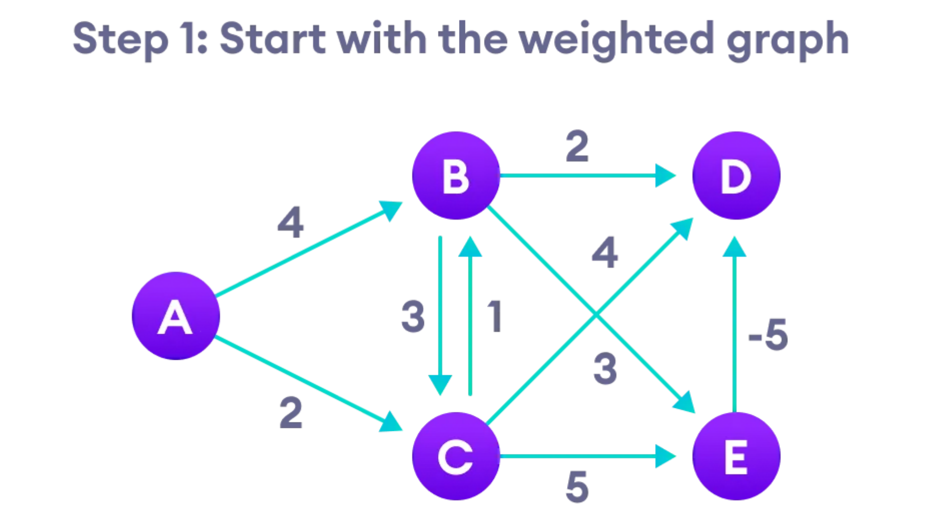
Negative weight cycles can give an incorrect result when trying to find out the shortest path

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.

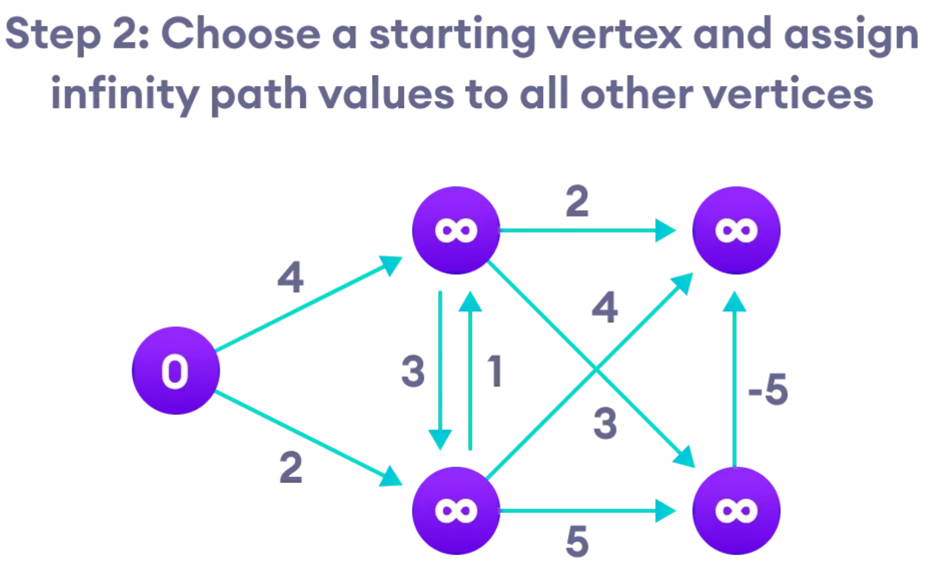
How Bellman Ford's algorithm works

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.

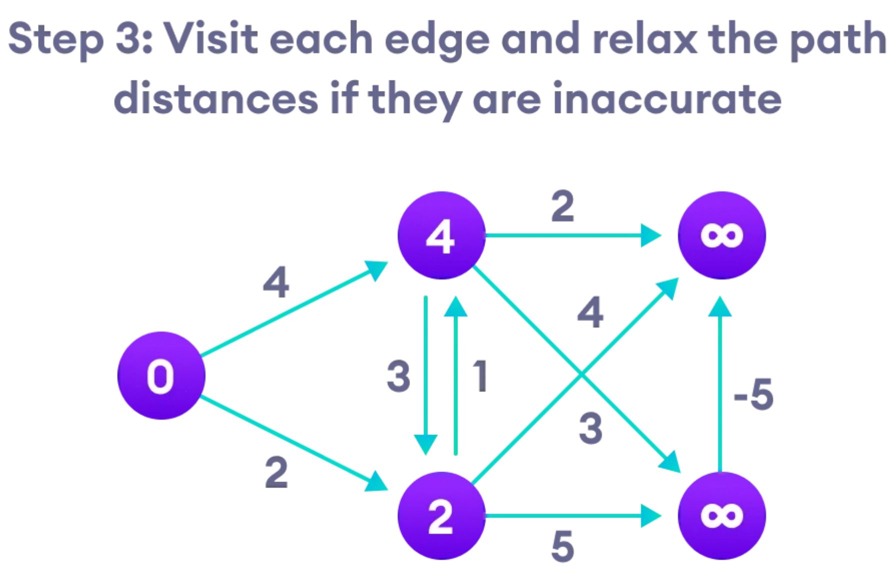
By doing this repeatedly for all vertices, we can guarantee that the result is optimized.



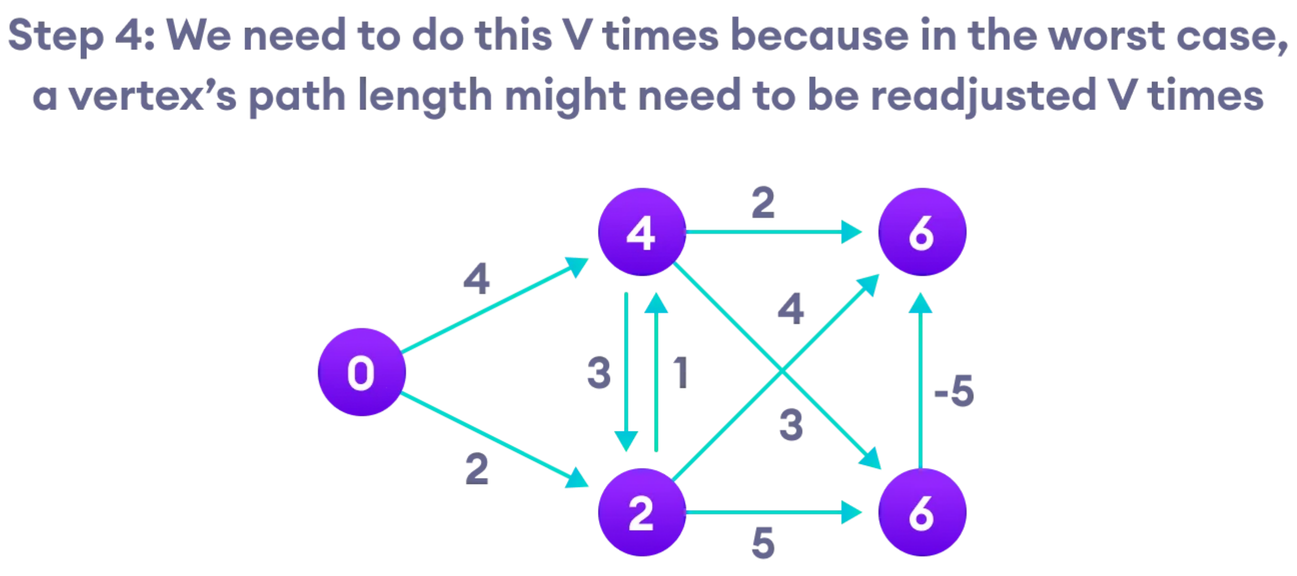
Step-1 for Bellman Ford's algorithm


Step 2: Choose a starting vertex and assign infinity path values to all other vertices
0
∞
∞
∞
∞
4
2
3
1
2
4
3
5
-5

Step-2 for Bellman Ford's algorithm

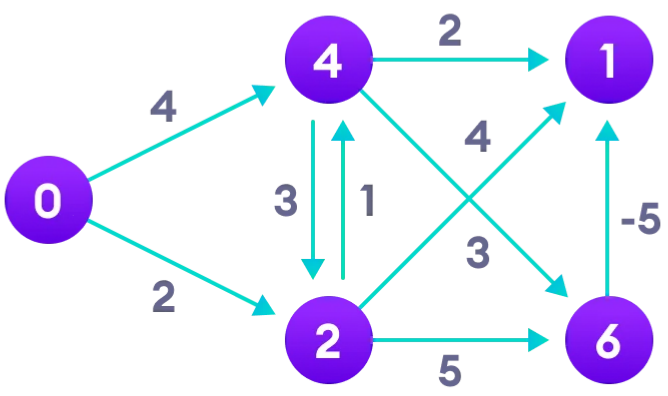

Step 3: Visit each edge and relax the path distances if they are inaccurate
0
4
2
∞
∞
4
2
3
1
2
4
3
-5
5

Step-3 for Bellman Ford's algorithm


Step 4: We need to do this V times because in the worst case, a vertex's path length might need to be readjusted V times
0
4
2
6
6
4
2
3
1
2
4
3
-5
5

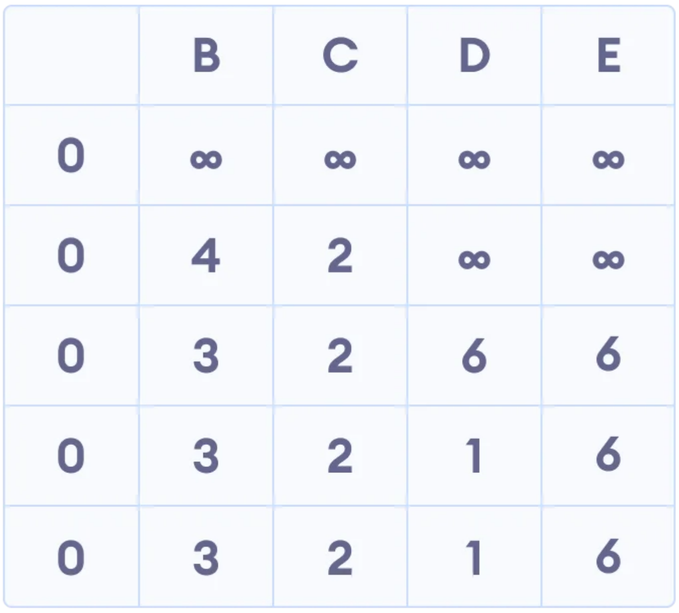
Step-4 for Bellman Ford's algorithm

Step 5: Notice how the vertex at the top right corner had its path length adjusted



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.

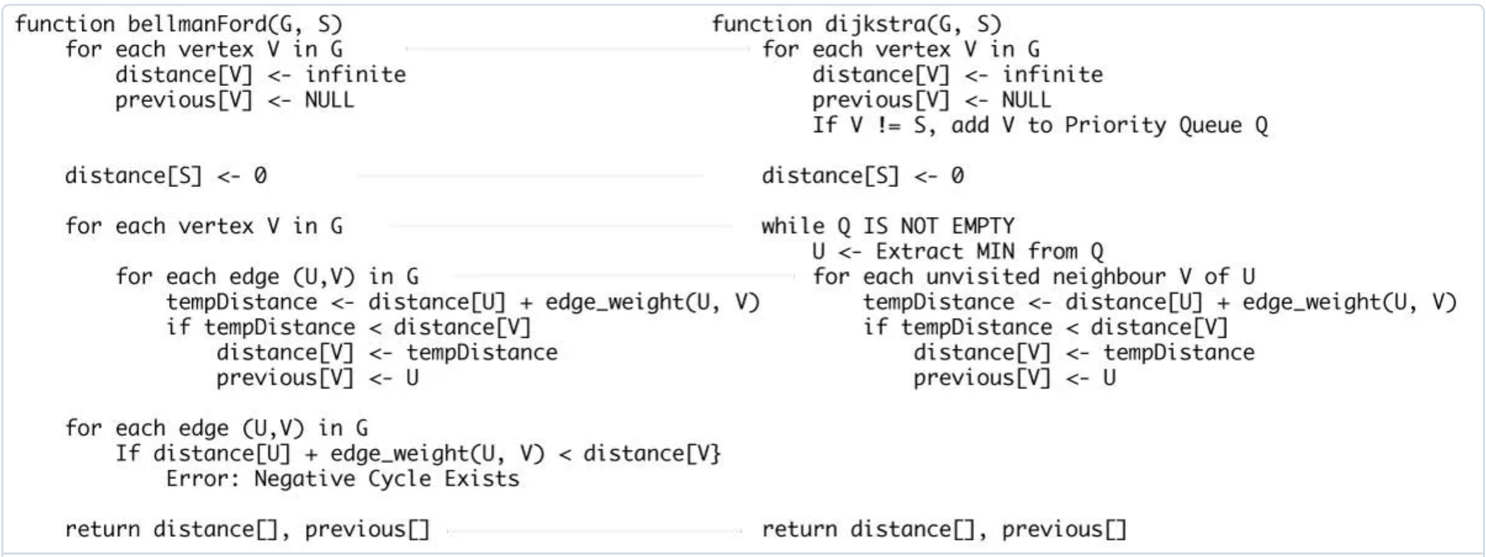
We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

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[]
```

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q

    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    return distance[], previous[]
```

Bellman Ford's Algorithm vs Dijkstra's Algorithm

Python, Java and C/C++ Examples

Python Java C C++

```c
// Bellman Ford Algorithm in C

#include <stdio.h>
#include <stdlib.h>

#define INFINITY 99999

//struct for the edges of the graph
struct Edge {
  int u;  //start vertex of the edge
  int v;  //end vertex of the edge
  int w;  //weight of the edge (u,v)
};

//Graph - it consists of edges
struct Graph {
  int V;        //total number of vertices in the graph
  int E;        //total number of edges in the graph
  struct Edge *edge;  //array of edges
};

void bellmanford(struct Graph *g, int source);
void display(int arr[], int size);

int main(void) {
  //create graph
  struct Graph *g = (struct Graph *)malloc(sizeof(struct Graph));
  g->V = 4;  //total vertices
```

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