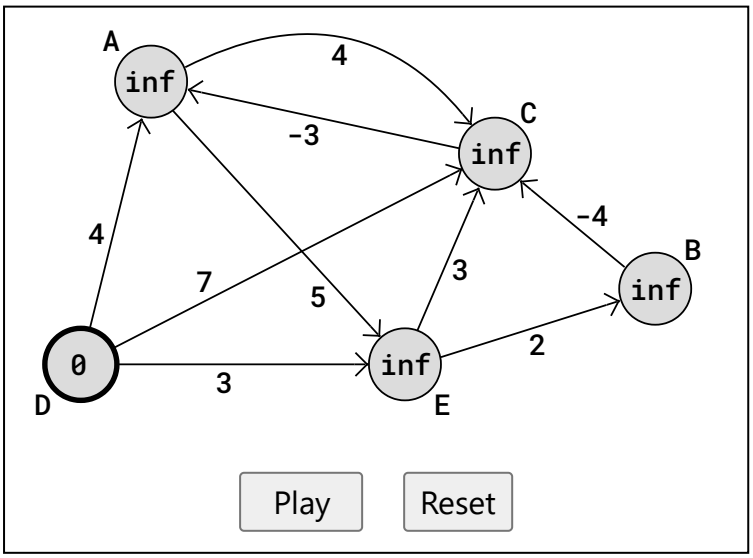
DSA Bellman-Ford Algorithm

Previous Next

The Bellman-Ford Algorithm

The Bellman-Ford algorithm is best suited to find the shortest paths in a directed graph, with one or more negative edge weights, from the source vertex to all other vertices.

It does so by repeatedly checking all the edges in the graph for shorter paths, as many times as there are vertices in the graph (minus 1).



Using the Bellman-Ford algorithm on a graph with negative cycles will not produce a result of shortest paths because in a negative cycle we can always go one more round and get a shorter path.

A negative cycle is a path we can follow in circles, where the sum of the edge weights is negative.

Luckily, the Bellman-Ford algorithm can be implemented to safely detect and report the presence of negative cycles.

How it works:

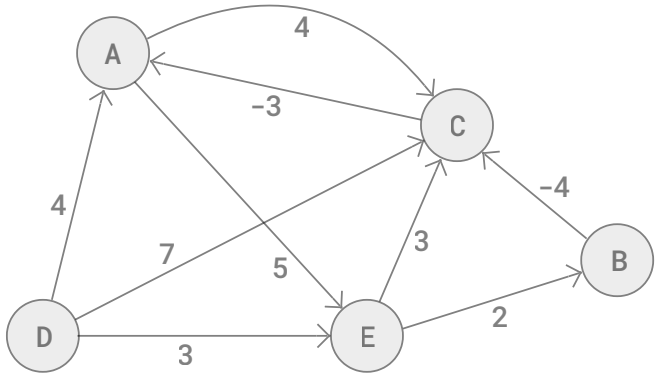
1. Set initial distance to zero for the source vertex, and set initial distances to infinity for all other vertices.
2. For each edge, check if a shorter distance can be calculated, and update the distance if the calculated distance is shorter.
3. Check all edges (step 2) $V - 1$ times. This is as many times as there are vertices (V), minus one.
4. Optional: Check for negative cycles. This will be explained in better detail later.

The animation of the Bellman-Ford algorithm above only shows us when checking of an edge leads to an updated distance, not all the other edge checks that do not lead to updated distances.

Manual Run Through

The Bellman-Ford algorithm is actually quite straight forward, because it checks all edges, using the adjacency matrix. Each check is to see if a shorter distance can be made by going from the vertex on one side of the edge, via the edge, to the vertex on the other side of the edge.

And this check of all edges is done $V - 1$ times, with V being the number of vertices in the graph.

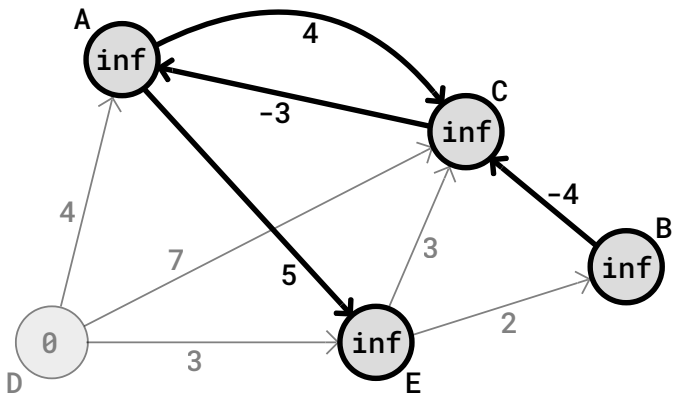


	A	B	C	D	E
A			4		5
B			-4		
C	-3				
D	4		7		3
E		2	3		

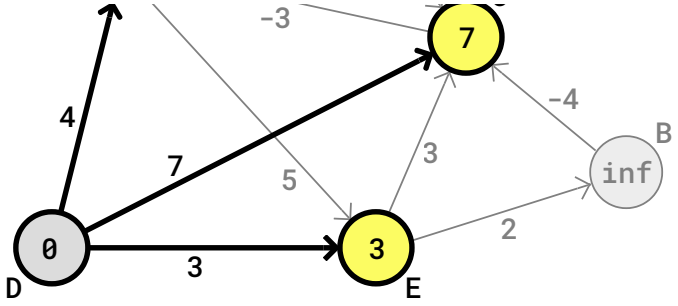
Checked all edges 0 times.

Play Reset

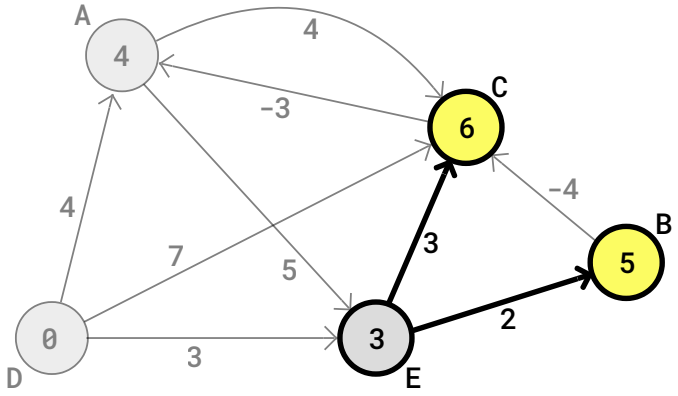
The first four edges that are checked in our graph are A->C, A->E, B->C, and C->A. These first four edge checks do not lead to any updates of the shortest distances because the starting vertex of all these edges has an infinite distance.



After the edges from vertices A, B, and C are checked, the edges from D are checked. Since the starting point (vertex D) has distance 0, the updated distances for A, B, and C are the

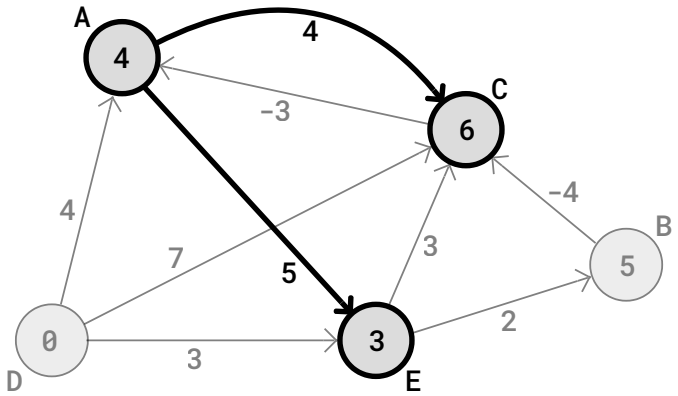


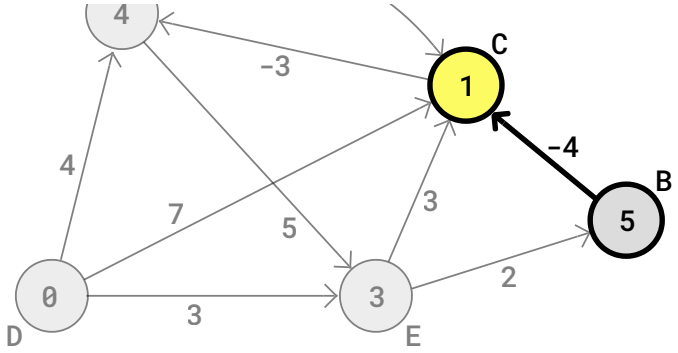
The next edges to be checked are the edges going out from vertex E, which leads to updated distances for vertices B and C.



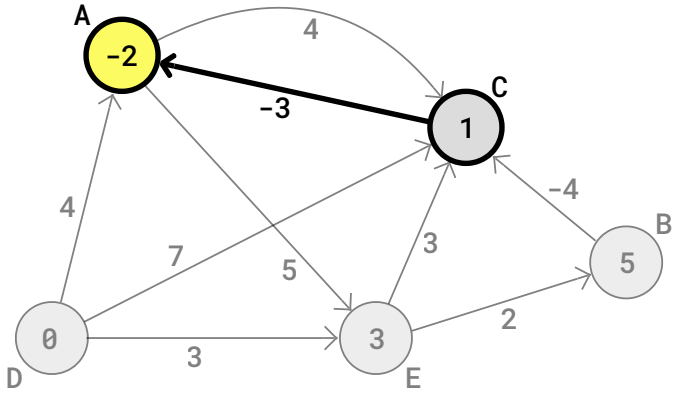
The Bellman-Ford algorithm have now checked all edges 1 time. The algorithm will check all edges 3 more times before it is finished, because Bellman-Ford will check all edges as many times as there are vertices in the graph, minus 1.

The algorithm starts checking all edges a second time, starting with checking the edges going out from vertex A. Checking the edges A->C and A->E do not lead to updated distances.





Checking the next edge C->A, leads to an updated distance 1-3=-2 for vertex A.



The check of edge C->A in round 2 of the Bellman-Ford algorithm is actually the last check that leads to an updated distance for this specific graph. The algorithm will continue to check all edges 2 more times without updating any distances.

Checking all edges $V - 1$ times in the Bellman-Ford algorithm may seem like a lot, but it is done this many times to make sure that the shortest distances will always be found.

Implementation of The Bellman-Ford Algorithm

Implementing the Bellman-Ford algorithm is very similar to how we implemented Dijkstra's algorithm.

We start by creating the `Graph` class, where the methods `__init__`, `add_edge`, and `add_vertex` will be used to create the specific graph we want to run the Bellman-Ford algorithm on to find the shortest paths.

```
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
            #self.adj_matrix[v][u] = weight  # For undirected graph

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data
```

The `bellman_ford` method is also placed inside the `Graph` class. It is this method that runs the Bellman-Ford algorithm.

```
    def bellman_ford(self, start_vertex_data):
        start_vertex = self.vertex_data.index(start_vertex_data)



                                print(f"Relaxing edge {self.vertex_data

        return distances
```

Line 18-19: At the beginning, all vertices are set to have an infinite long distance from the starting vertex, except for the starting vertex itself, where the distance is set to 0.

Line 24-26: If the edge exist, and if the calculated distance is shorter than the existing distance, update the distance to that vertex `v`.

The complete code, including the initialization of our specific graph and code for running the Bellman-Ford algorithm, looks like this:

Example

Python:

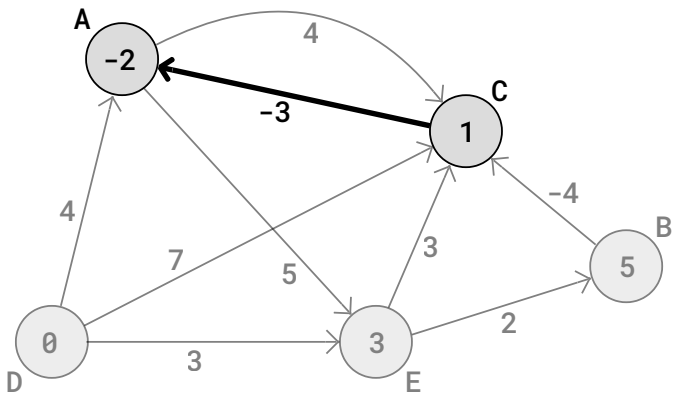
```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
            #self.adj_matrix[v][u] = weight  # For undirected gra

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data

```

Try it Yourself »

Negative Edges in The Bellman-Ford Algorithm

To say that the Bellman-Ford algorithm finds the "shortest paths" is not intuitive, because how can we draw or imagine distances that are negative? So, to make it easier to understand

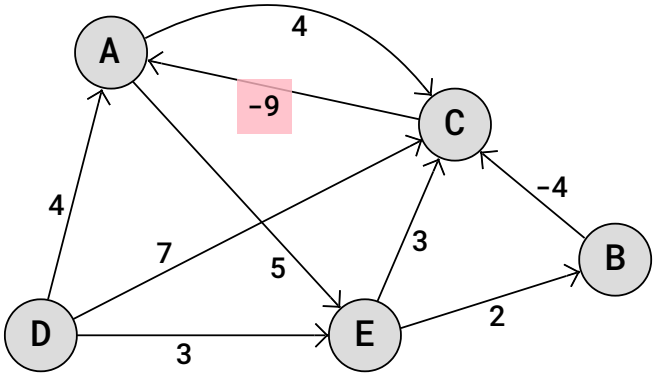
made by driving that edge between those two vertices.



With this interpretation in mind, the -3 weight on edge C->A could mean that the fuel cost is $5 driving from C to A, and that we get paid $8 for picking up packages in C and delivering them in A. So we end up earning $3 more than we spend. Therefore, a total of $2 can be made by driving the delivery route D->E->B->C->A in our graph above.

Negative Cycles in The Bellman-Ford Algorithm

If we can go in circles in a graph, and the sum of edges in that circle is negative, we have a negative cycle.



By changing the weight on edge C->A from -3 to -9, we get two negative cycles: A->C->A and A->E->C->A. And every time we check these edges with the Bellman-Ford algorithm, the distances we calculate and update just become lower and lower.

cycles.

Detection of Negative Cycles in the Bellman-Ford Algorithm

After running the Bellman-Ford algorithm, checking all edges in a graph $V - 1$ times, all the shortest distances are found.

But, if the graph contains negative cycles, and we go one more round checking all edges, we will find at least one shorter distance in this last round, right?

So to detect negative cycles in the Bellman-Ford algorithm, after checking all edges $V - 1$ times, we just need to check all edges one more time, and if we find a shorter distance this last time, we can conclude that a negative cycle must exist.

Below is the `bellman_ford` method, with negative cycle detection included, running on the graph above with negative cycles due to the C->A edge weight of -9:

Example

Python:

```
        def bellman_ford(self, start_vertex_data):
            start_vertex = self.vertex_data.index(start_vertex_data)
            distances = [float('inf')] * self.size
            distances[start_vertex] = 0

            for i in range(self.size - 1):
                for u in range(self.size):
                    for v in range(self.size):
                        if self.adj_matrix[u][v] != 0:
                            if distances[u] + self.adj_matrix[u][v] < d
                                distances[v] = distances[u] + self.adj_
                                print(f"Relaxing edge {self.vertex_data

            # Negative cycle detection
```

```
35                          return (True, None)  # Indicate a negative

                    return (False, distances)  # Indicate no negative cycle and
```

Try it Yourself »

Line 30-33: All edges are checked one more time to see if there are negative cycles.

Line 34: Returning `True` indicates that a negative cycle exists, and `None` is returned instead of the shortest distances, because finding the shortest distances in a graph with negative cycles does not make sense (because a shorter distance can always be found by checking all edges one more time).

Line 36: Returning `False` means that there is no negative cycles, and the `distances` can be returned.

Returning The Paths from The Bellman-Ford Algorithm

We are currently finding the total weight of the the shortest paths, so that for example "Distance from D to A: -2" is a result from running the Bellman-Ford algorithm.

But by recording the predecessor of each vertex whenever an edge is relaxed, we can use that later in our code to print the result including the actual shortest paths. This means we can give more information in our result, with the actual path in addition to the path weight: "D->E->B->C->A, Distance: -2".

This last code example is the complete code for the Bellman-Ford algorithm, with everything we have discussed up until now: finding the weights of shortest paths, detecting negative cycles, and finding the actual shortest paths:

Example

Python:

```
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
            #self.adj_matrix[v][u] = weight  # For undirected gra

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data

```


Line 19: The `predecessors` array holds each vertex' predecessor vertex in the shortest path.

Line 28: The `predecessors` array gets updated with the new predecessor vertex every time an edge is relaxed.

Line 40-49: The `get_path` method uses the `predecessors` array to generate the shortest path string for each vertex.

Time Complexity for The Bellman-Ford Algorithm

The time complexity for the Bellman-Ford algorithm mostly depends on the nested loops.

The outer for-loop runs $V - 1$ times, or V times in case we also have negative cycle detection. For graphs with many vertices, checking all edges one less time than there are vertices makes little difference, so we can say that the outer loop contributes with $O(V)$ to the time complexity.

So in total, we get the time complexity for the Bellman-Ford algorithm:

$$O(V^3)$$

However, in practical situations and especially for sparse graphs, meaning each vertex only has edges to a small portion of the other vertices, time complexity of the two inner for-loops checking all edges can be approximated from $O(V^2)$ to $O(E)$, and we get the total time complexity for Bellman-Ford:

$$O(V \cdot E)$$

The time complexity for the Bellman-Ford algorithm is slower than for Dijkstra's algorithm, but Bellman-Ford can find the shortest paths in graphs with negative edges and it can detect negative cycles, which Dijkstra's algorithm cannot do.

DSA Exercises

Test Yourself With Exercises

Exercise:

In the adjacency matrix below:

	A	B	C	D	E
A			4		5
B			-4		
C	-3				
D	4		7		3
E		2	3		

What is the edge weight of the edge going from D to E?

Submit Answer »

Start the Exercise

‹ Previous Sign in to track progress Next ›

COLOR PICKER

Top Tutorials

HTML Tutorial
CSS Tutorial
JavaScript Tutorial
How To Tutorial
SQL Tutorial
Python Tutorial
W3.CSS Tutorial
Bootstrap Tutorial
PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference
AngularJS Reference
jQuery Reference

Top Examples

HTML Examples
CSS Examples
JavaScript Examples
How To Examples
SQL Examples
Python Examples
W3.CSS Examples
Bootstrap Examples

Get Certified

HTML Certificate
CSS Certificate
JavaScript Certificate
Front End Certificate
SQL Certificate
Python Certificate
PHP Certificate
jQuery Certificate