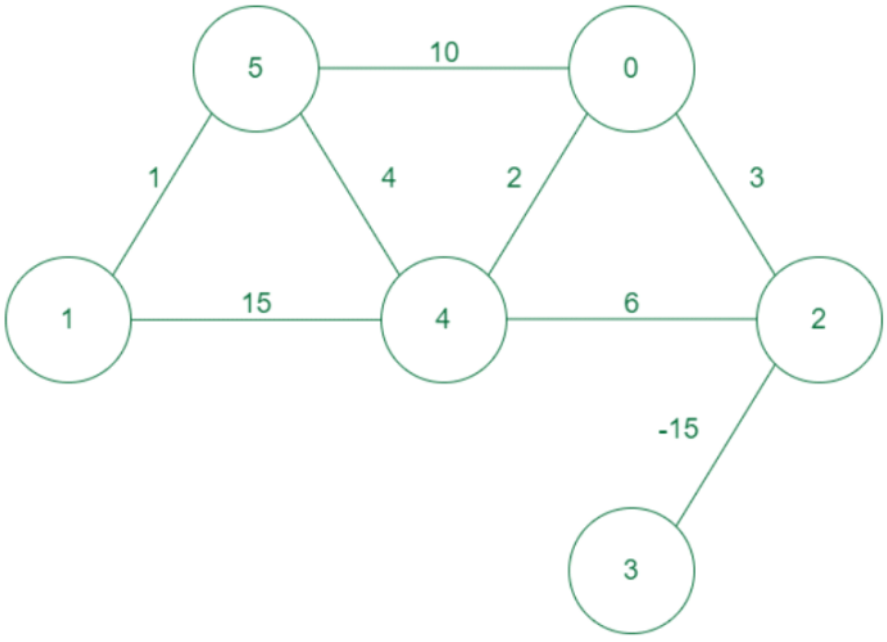
Weighted Graph, Shortest Path, Dijsktra and Bellman

Wednesday, 31 December 2025 10:24 am

https://www.geeksforgeeks.org/dsa/applications-advantages-and-disadvantages-of-weighted-graph/

What is Weighted Graph?

A **weighted graph** is defined as a special type of graph in which the edges are assigned some weights which represent cost, distance, and many other relative measuring units.



Example of a weighted graph

Applications of Weighted Graph:

- **2D matrix games:** In 2d matrix, games can be used to find the optimal path for maximum sum along starting to ending points and many variations of it can be found online.
- **Spanning trees:** Weighted graphs are used to find the minimum spanning tree from graph which depicts the minimal cost to traverse all nodes in the graph.
- **Constraints graphs:** Graphs are often used to represent constraints among items. Used in scheduling, product design, asset allocation, circuit design, and artificial intelligence.
- **Dependency graphs:** Directed weighted graphs can be used to represent dependencies or precedence order among items. Priority will be assigned to provide a flow in which we will solve the problem or traverse the graph from highest priority to lowest priority. Such graphs are often used in large projects in laying out what components rely on other components and are used to minimize the total time or cost to completion while abiding by the dependencies.
- **Compilers:** Weighted graphs are used extensively in compilers. They can be used for type inference, for so-called data flow analysis, and many other purposes such as query optimization in database languages.
- **Artificial Intelligence:** Weighted graphs are used in artificial intelligence for decision-making processes, such as in game trees for determining the best move in a game.

Real-Time Applications of Weighted Graph:

- **Transportation networks:** Using weighted graphs, we can figure things out like the path that takes the least time, or the path with the least overall distance. This is a simplification of how weighted graphs can be used for more complex things like a GPS system. Graphs are used to study traffic patterns, traffic light timings and much more by many big tech companies such as OLA, UBER, RAPIDO, etc. Graph networks are used by many map programs such as Google Maps, Bing Maps, etc.
- **Document link graphs:** Link weighted graphs are used to analyze relevance of web pages, the best sources of information, and good link sites by taking the count of the number of views as weights in the graph.
- **Epidemiology:** Weighted graphs can be used to find the maximum distance transmission from an infectious to a healthy person.
- **Graphs in quantum field theory:** Vertices represent states of a quantum system and the edges represent transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude. Research to find the maximum frequency along a path can be done using weighted graphs.

Advantages of Weighted Graph:

- **Better representation of real-world scenarios**: Weighted graphs are a more accurate representation of many real-world scenarios, where the relationships between entities have varying degrees of importance. For example, in a road network, some roads may have higher speed limits or more lanes, and these differences can be represented using weights.
- **More accurate pathfinding**: In a weighted graph, finding the shortest path between two nodes takes into account the weights of the edges, which can lead to more accurate results. This is particularly useful in applications where finding the optimal path is critical, such as in logistics or transportation planning.
- **More efficient algorithms**: Many graph algorithms are more efficient when applied to weighted graphs, such as Dijkstra's algorithm for finding the shortest path. This is because the weights provide additional information that can be used to optimize the search.
- **More flexible analysis:** Weighted graphs allow for more flexible analysis of the relationships between nodes. For example, it is possible to calculate the average weight of edges, or to identify nodes with unusually high or low weights. This can provide insights into the structure of the graph and the relationships between its nodes.

Disadvantages of Weighted Graph:

- **Increased complexity:** Weighted graphs are more complex than unweighted graphs, and can be more difficult to understand and analyze. This complexity can make it harder to develop and debug algorithms that operate on weighted graphs.
- **Higher memory usage:** Weighted graphs require more memory than unweighted graphs, because each edge has an associated weight. This can be a problem for applications that have limited memory resources.
- **More difficult to maintain:** Weighted graphs can be more difficult to maintain than unweighted graphs, because changes to the weights of edges can have a ripple effect throughout the graph. This can make it harder to add or remove edges, or to update the weights of existing edges.
- **Not suitable for all applications:** Weighted graphs are not always the best choice for every application. For example, if the relationships between nodes are binary (i.e., either present or absent), an unweighted graph may be more appropriate.
- **Bias towards certain properties:** Weighted graphs can introduce bias towards certain properties, such as shortest path or highest weight. This can be a problem in applications where a more even distribution of weights is desired.

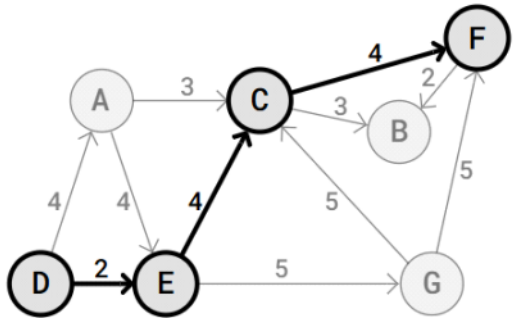
https://www.w3schools.com/dsa/dsa_theory_graphs_shortestpath.php

The Shortest Path Problem

The shortest path problem is famous in the field of computer science.

To solve the shortest path problem means to find the shortest possible route or path between two vertices (or nodes) in a Graph.

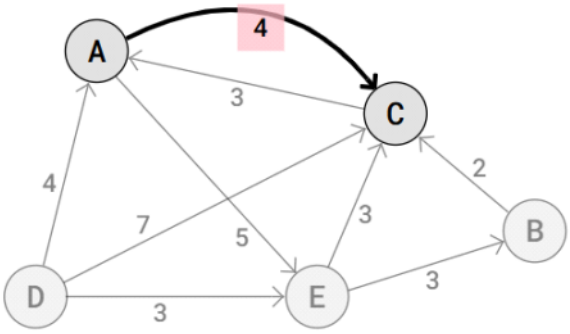
In the shortest path problem, a Graph can represent anything from a road network to a communication network, where the vertices can be intersections, cities, or routers, and the edges can be roads, flight paths, or data links.



The shortest path from vertex D to vertex F in the Graph above is D->E->C->F, with a total path weight of 2+4+4=10. Other paths from D to F are also possible, but they have a higher total weight, so they can not be considered to be the shortest path.

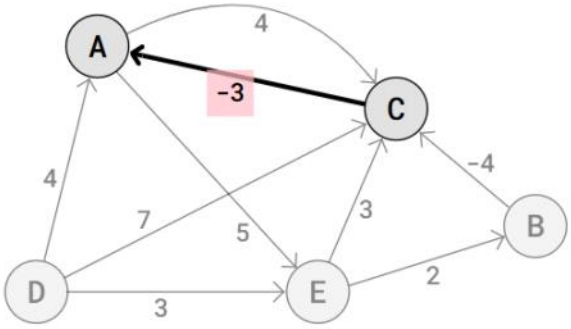
Positive and Negative Edge Weights

Some algorithms that find the shortest paths, like Dijkstra's algorithm, can only find the shortest paths in graphs where all the edges are positive. Such graphs with positive distances are also the easiest to understand because we can think of the edges between vertices as distances between locations.



If we interpret the edge weights as money lost by going from one vertex to another, a positive edge weight of 4 from vertex A to C in the graph above means that we must spend $4 to go from A to C.

But graphs can also have negative edges, and for such graphs the Bellman-Ford algorithm can be used to find the shortest paths.



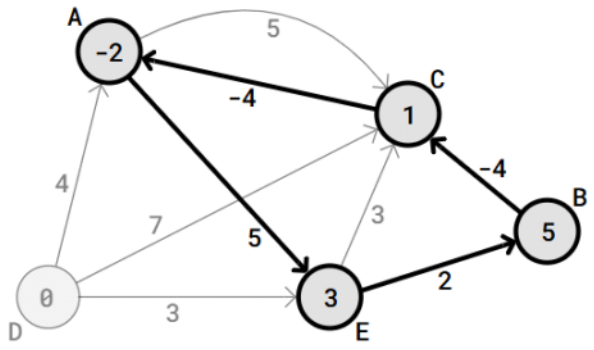
And similarly, if the edge weights represent money lost, the negative edge weight -3 from vertex C to A in the graph above can be understood as an edge where there is more money to be made than money lost by going from C to A. So if for example the cost of fuel is $5 going from C to A, and we get paid $8 for picking up packages in C and delivering them in A, money lost is -3, meaning we are actually earning $3 in total.

Negative Cycles in Shortest Path Problems

Finding the shortest paths becomes impossible if a graph has negative cycles.

Having a negative cycle means that there is a path where you can go in circles, and the edges that make up this circle have a total path weight that is negative.

In the graph below, the path A->E->B->C->A is a negative cycle because the total path weight is 5+2-4-4=-1.



The reason why it is impossible to find the shortest paths in a graph with negative cycles is that it will always be possible to continue running an algorithm to find even shorter paths.

Let's say for example that we are looking for the shortest distance from vertex D in graph above, to all other vertices. At first we find the distance from D to E to be 3, by just walking the edge D->E. But after this, if we walk one round in the negative cycle E->B->C->A->E, then the distance to E becomes 2. After walking one more round the distance becomes 1, which is even shorter, and so on. We can always walk one more round in the negative cycle to find a shorter distance to E, which means the shortest distance can never be found.

Luckily, the the Bellman-Ford algorithm, that runs on graphs with negative edges, can be implemented with detection for negative cycles.

https://www.w3schools.com/dsa/dsa_algo_graphs_dijkstra.php

Dijkstra's shortest path algorithm was invented in 1956 by the Dutch computer scientist Edsger W. Dijkstra during a twenty minutes coffee break, while out shopping with his fiancée in Amsterdam.

The reason for inventing the algorithm was to test a new computer called ARMAC.

Dijkstra's algorithm finds the shortest path from one vertex to all other vertices.

It does so by repeatedly selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighboring vertices.

Dijkstra's algorithm is often considered to be the most straightforward algorithm for solving the shortest path problem.

Dijkstra's algorithm is used for solving single-source shortest path problems for directed or undirected paths. Single-source means that one vertex is chosen to be the start, and the algorithm will find the shortest path from that vertex to all other vertices.

Dijkstra's algorithm does not work for graphs with negative edges. For graphs with negative edges, the Bellman-Ford algorithm that is described on the next page, can be used instead.

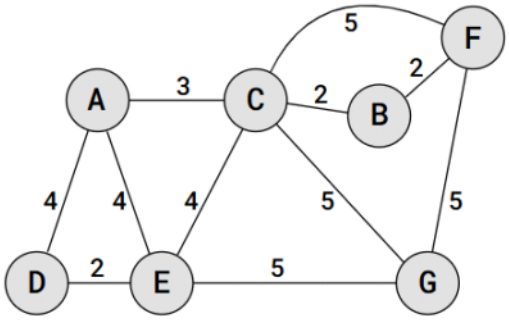
To find the shortest path, Dijkstra's algorithm needs to know which vertex is the source, it needs a way to mark vertices as visited, and it needs an overview of the current shortest distance to each vertex as it works its way through the graph, updating these distances when a shorter distance is found.

How it works:

1. Set initial distances for all vertices: 0 for the source vertex, and infinity for all the other.
2. Choose the unvisited vertex with the shortest distance from the start to be the current vertex. So the algorithm will always start with the source as the current vertex.
3. For each of the current vertex's unvisited neighbor vertices, calculate the distance from the source and update the distance if the new, calculated, distance is lower.
4. We are now done with the current vertex, so we mark it as visited. A visited vertex is not checked again
5. Go back to step 2 to choose a new current vertex, and keep repeating these steps until all vertices are visited.
6. In the end we are left with the shortest path from the source vertex to every other vertex in the graph.

Manual Run Through

Consider the Graph below.

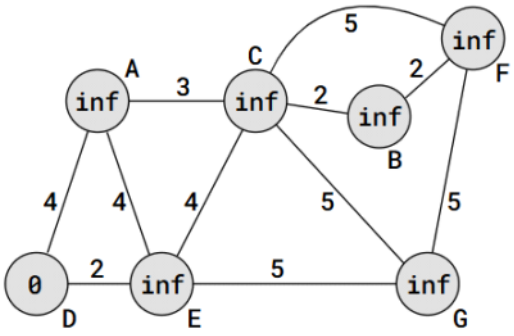


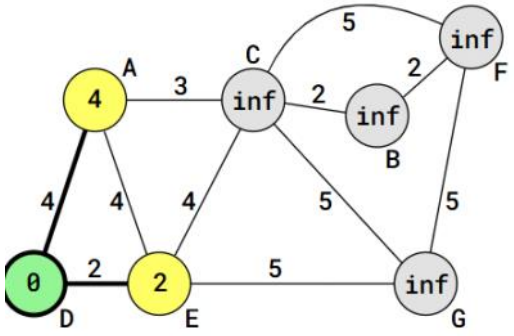
We want to find the shortest path from the source vertex D to all other vertices, so that for example the shortest path to C is D->E->C, with path weight 2+4=6.

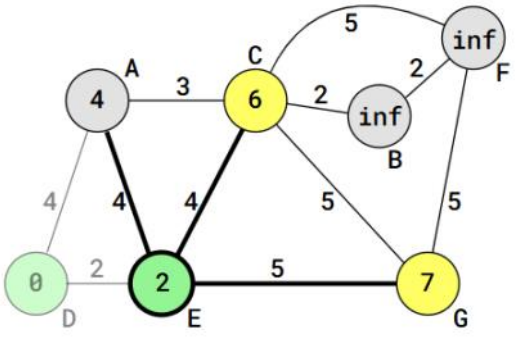
To find the shortest path, Dijkstra's algorithm uses an array with the distances to all other vertices, and initially sets these distances to infinite, or a very big number. And the distance to the vertex we start from (the source) is set to 0.

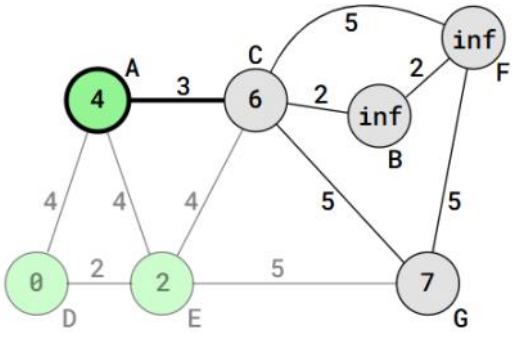
```
distances = [inf, inf, inf, 0, inf, inf, inf]
#vertices   [ A ,  B ,  C , D,  E ,  F ,  G ]
```

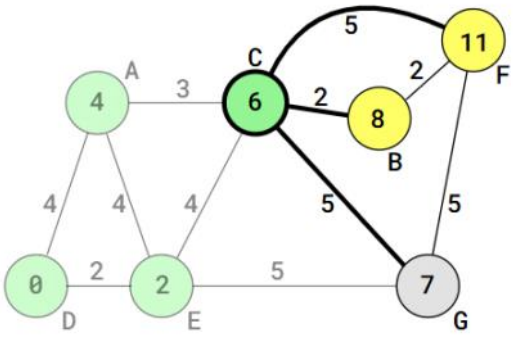
The image below shows the initial infinite distances to other vertices from the starting vertex D. The distance value for vertex D is 0 because that is the starting point.

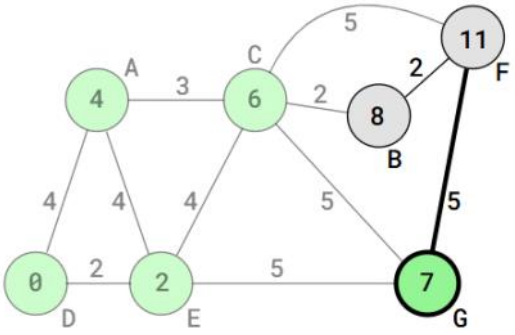


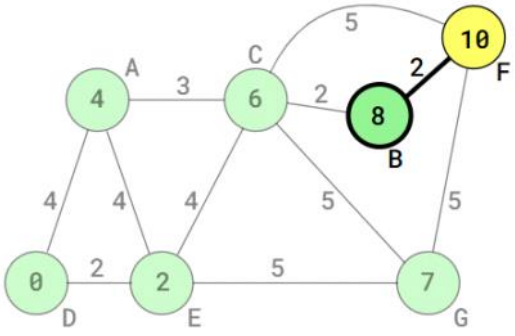








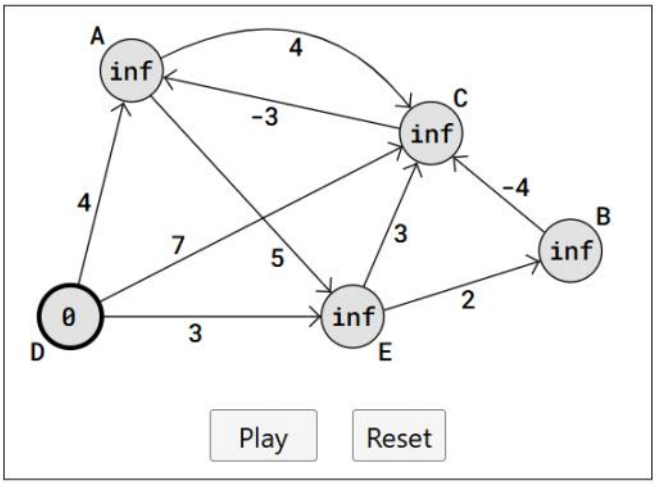




The Bellman-Ford Algorithm

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).



The Bellman-Ford algorithm can also be used for graphs with positive edges (both directed and undirected), like we can with Dijkstra's algorithm, but Dijkstra's algorithm is preferred in such cases because it is faster.

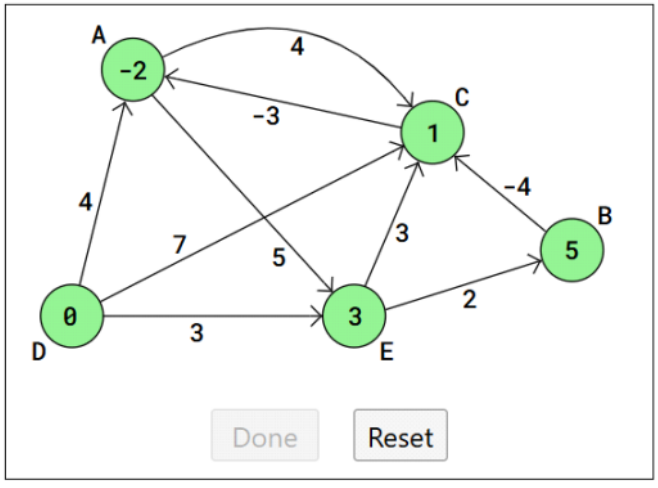
Using the Bellman-Ford algorithm on a graph with negative cycles will not produce a result of shortest paths because in a negative cycle we can always go one more round and get a shorter path.

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.

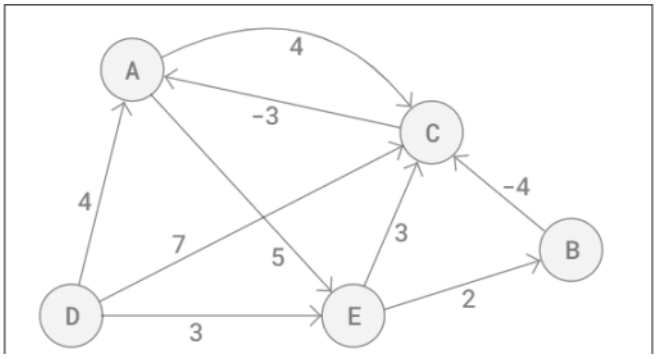


Manual Run Through

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.

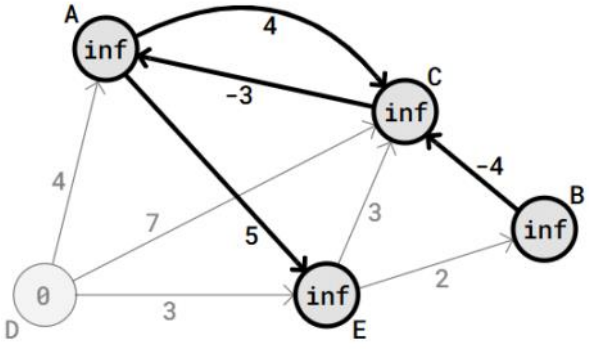
And this check of all edges is done $V - 1$ times, with V being the number of vertices in the graph.

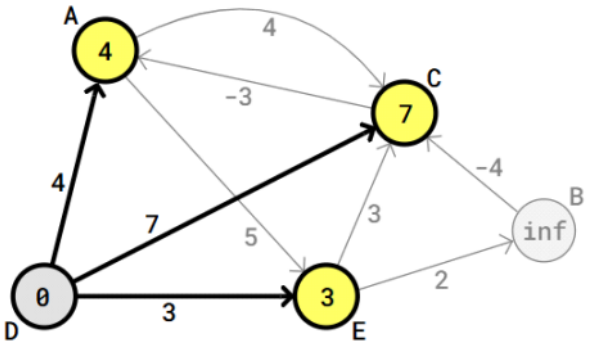
This is how the Bellman-Ford algorithm checks all the edges in the adjacency matrix in our graph 5-1=4 times:

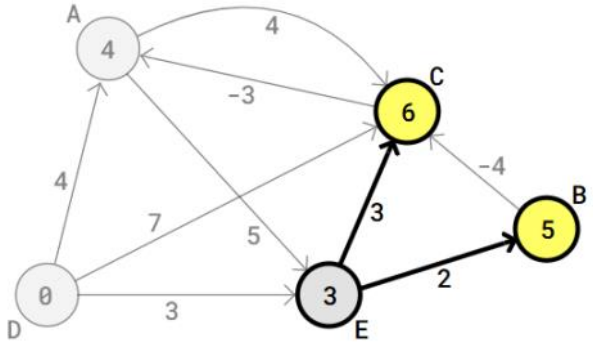


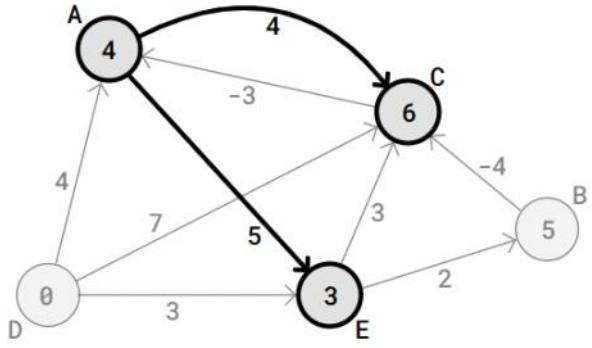
	A	B	C	D	E
A			4		5
B			-4		
C	-3				
D	4		7		3
E		2	3		

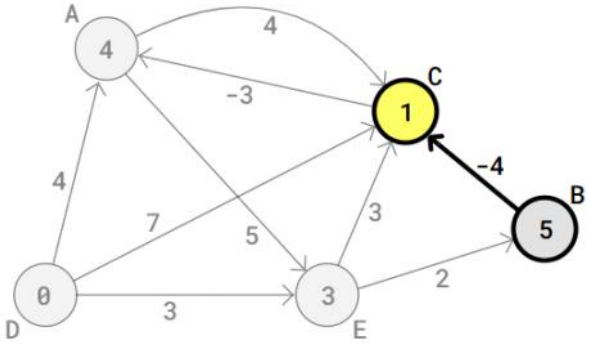
```
Checked all edges 0 times.
```

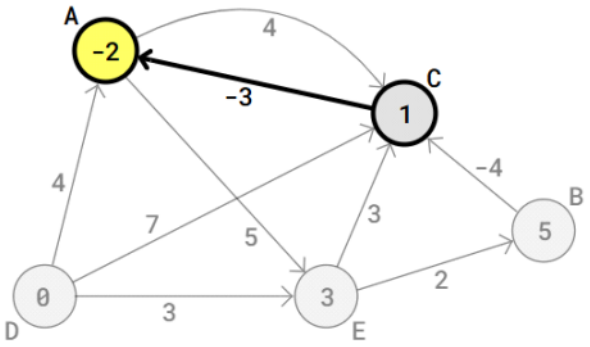



A
4
4
-3
C
7
4
7
5
3
-4
B
inf
2
0
3
D
3
E


A
4
4
-3
C
6
4
7
5
3
-4
B
5
2
0
3
D
3
E


A
4
4
-3
C
6
4
7
5
3
-4
B
5
2
0
3
D
3
E


A
4
4
-3
C
1
4
7
5
3
-4
B
5
2
0
3
D
3
E



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.