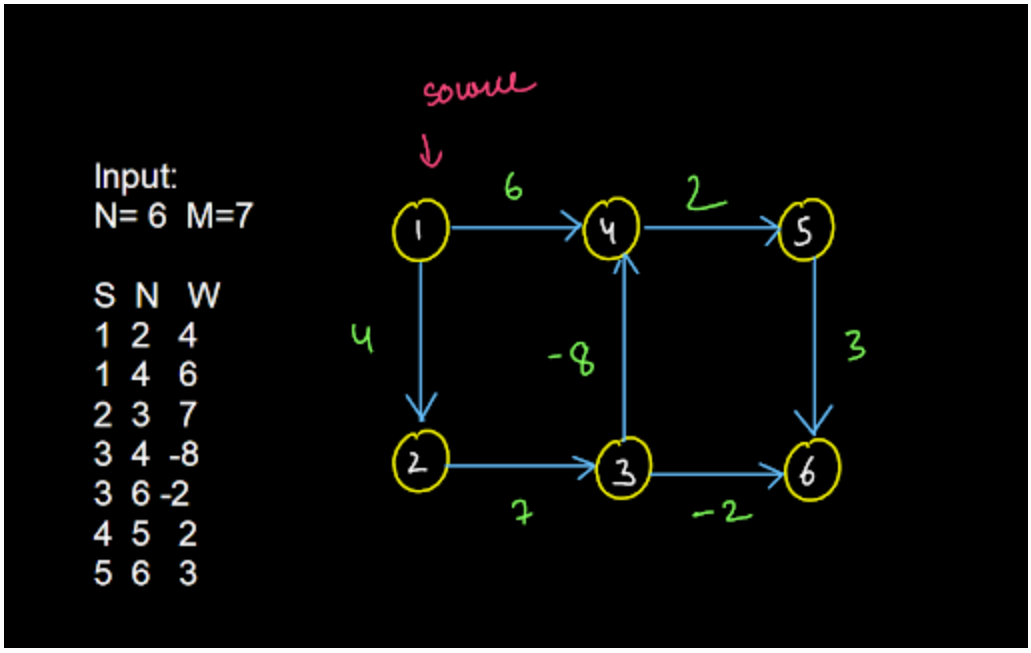
Graphs

Content

1. Bellman Ford

Problem

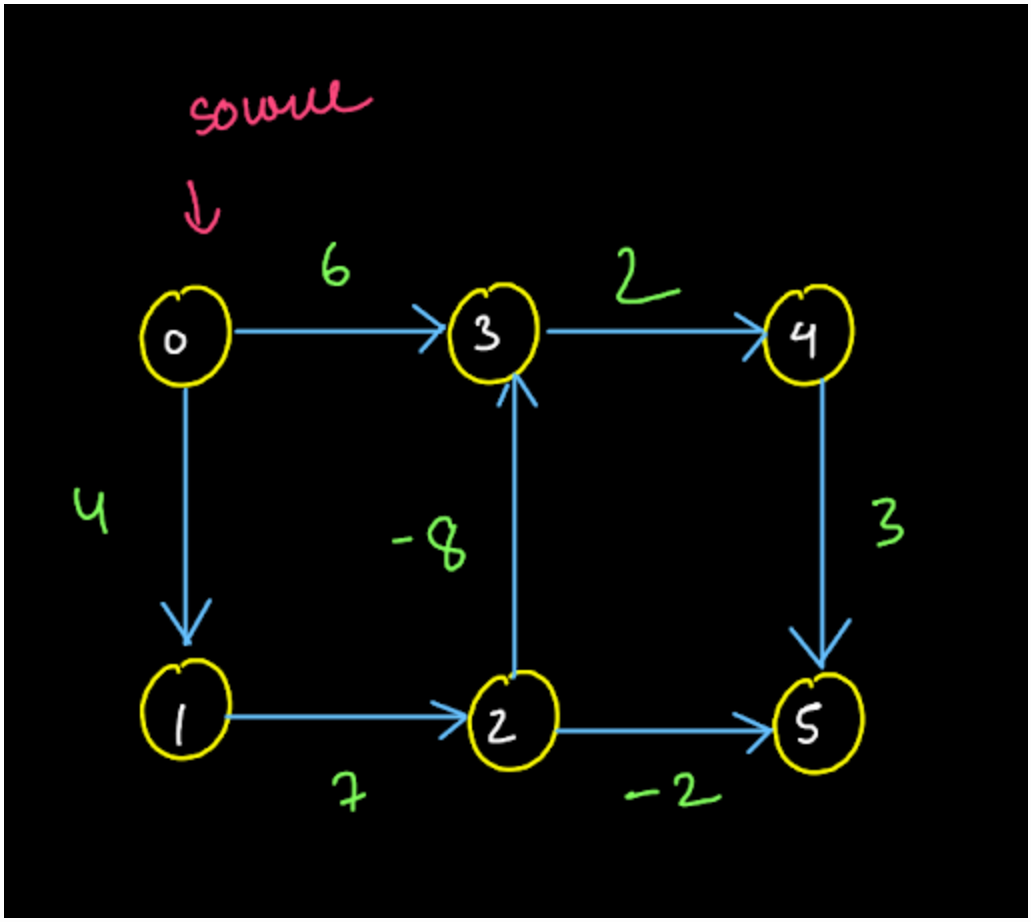
We are given two integer numbers N and M where N is the number of vertices and M is the number of edges of a graph. We are also given those M edges. Each edge will have a source, neighbor and a weight. We have to give the shortest path (in terms of weight) from the source vertex (i.e. the vertex numbered 1) to all other vertices.



For instance, in the above input graph, the shortest path from source (vertex 1) to vertex 4 is of cost 3 (the path is 1-2-3-4).

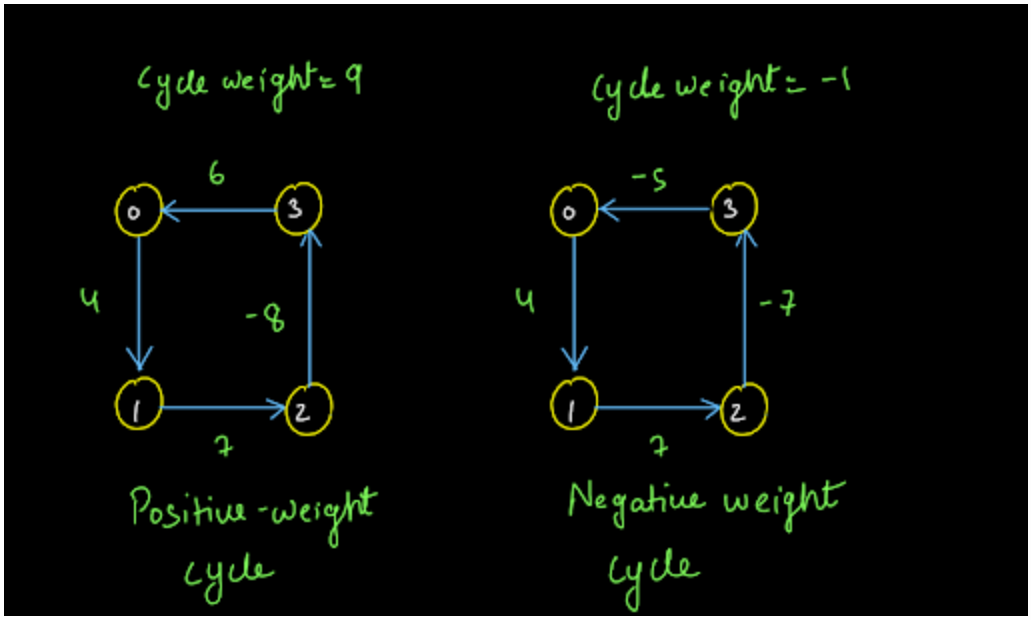
Approach (What of the Problem?)

Let's first convert our graph to a graph in which the vertices will start from 0. We are doing this to just ease our procedure. So, the above graph will look like:

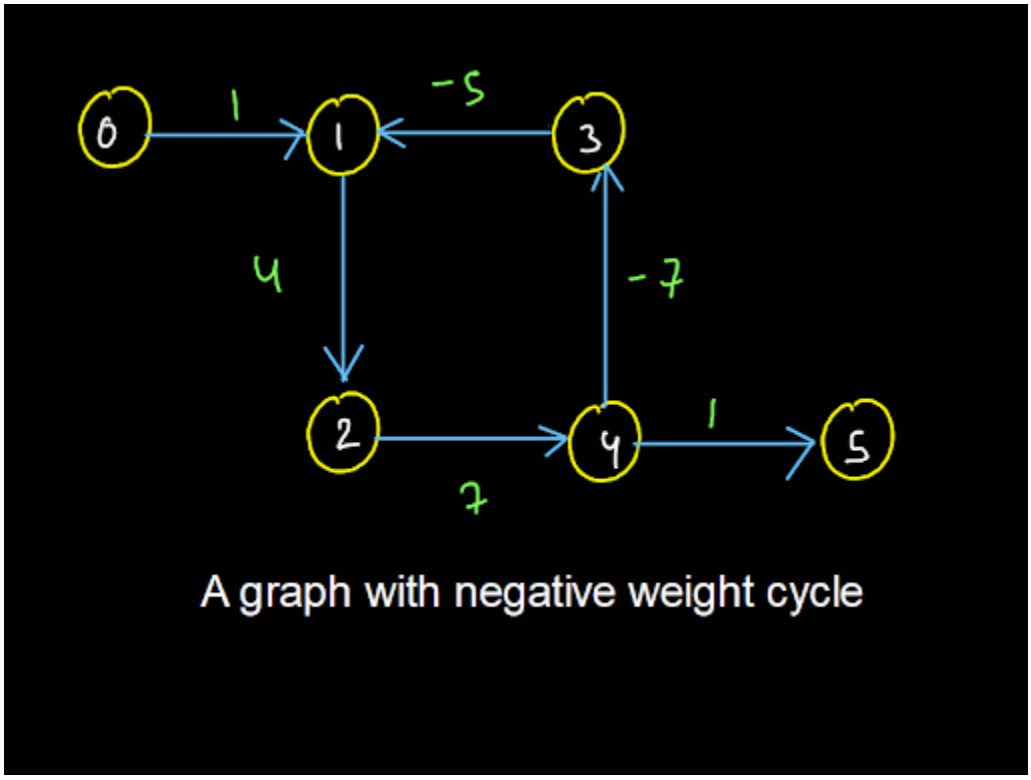


When to use Bellman Ford Algorithm?

Graphs can be of two types. They are cyclic graphs and acyclic graphs. The graphs that do not have any cycle are acyclic whereas the others are cyclic. So, if the graph is cyclic then to apply the Bellman Ford Algorithm to detect the shortest paths to all the other vertices from the source vertex, **it is necessary that the graph must not have any negative weight cycle.** The edges may have a negative weight but the overall weight for a cycle should not be negative (see img-3).



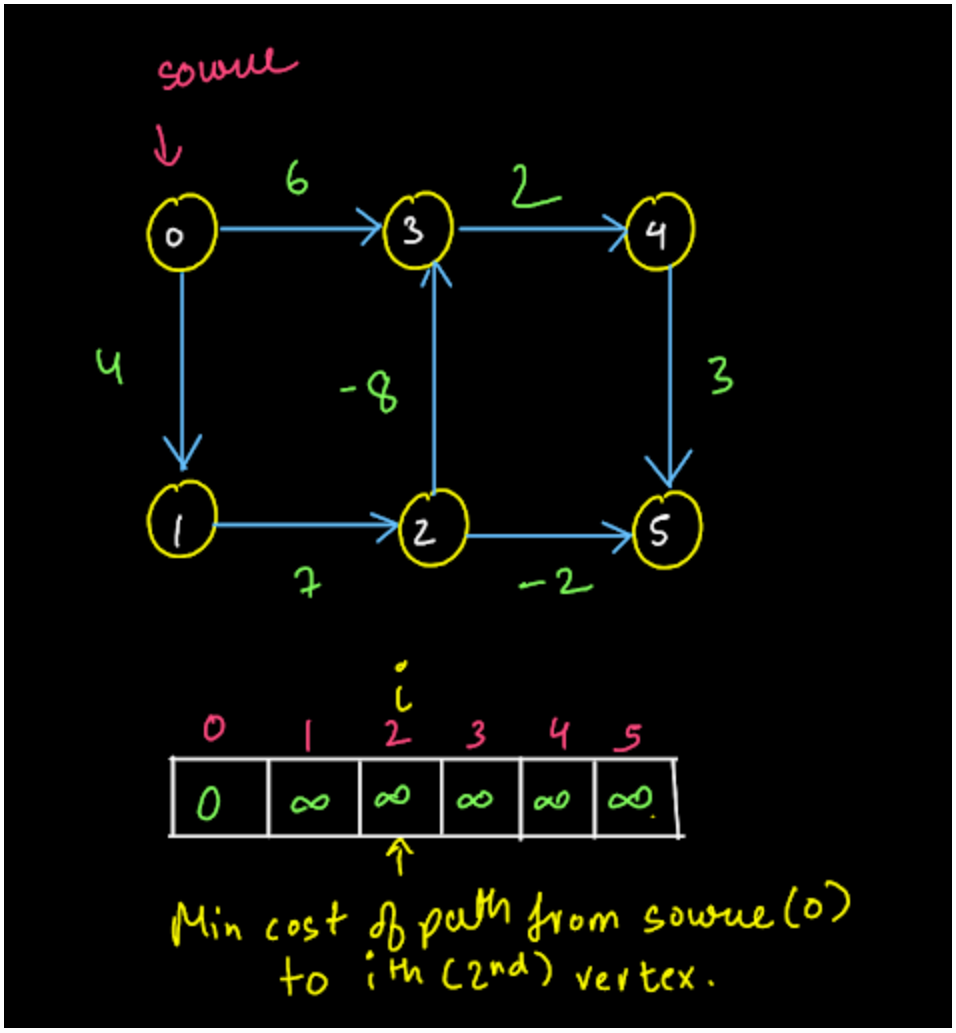
Can you guess why we don't allow a negative weight cycle for calculating the shortest path? Have a look at the diagram given below:



Can you find the shortest path from vertex 0 to vertex 5? You may say that it is 0-1-2-4-3-1-2-4-5 and the cost will be 1. So, here you have traversed the negative weight cycle once. What if we ask you to traverse this cycle twice? The cycle has a weight of -1. So, -1 will be added twice and the total cost will become 0. What if we ask you to complete the cycle thrice? The total cost will become -1. So, you can see that there is no end to this. We can traverse the negative cycle as many times as we want and the cost will keep on decreasing. To find the minimum cost, we will traverse the cycle an infinite number of times. This does not make any sense and hence the minimum cost does not make much sense when we have a negative cost/weight cycle.

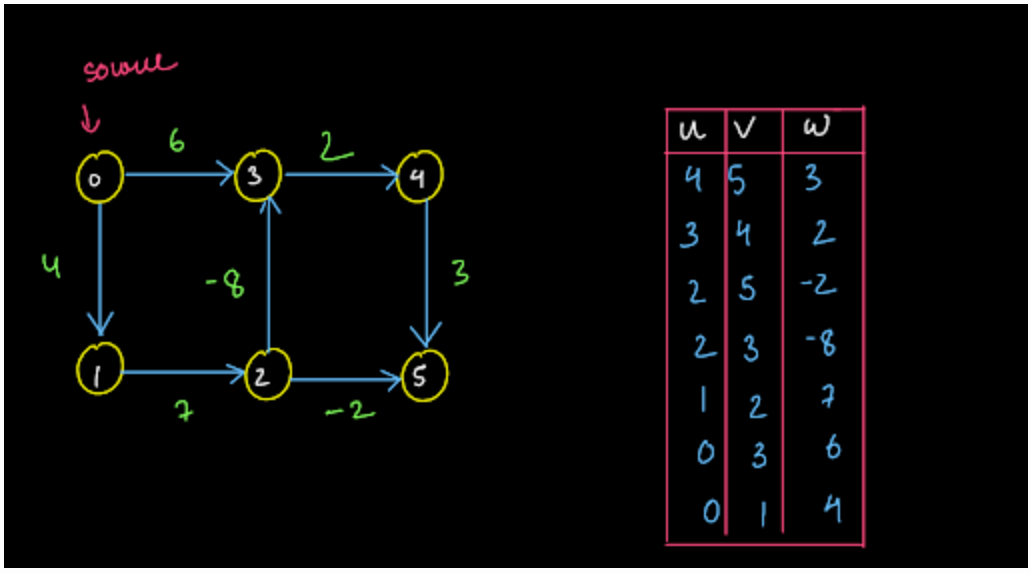
The application of Bellman Ford Algorithm:

We are going to take an array of the same size as that of the number of vertices in the graph. Initially, the array will be filled with infinity (maximum integer value in Java) at every position (except 0) depicting the minimum cost of path from source vertex to the ith vertex is infinity initially. What will be the value at index 0? At index 0, we will store the minimum cost of the path from 0 to 0 itself. What will that be? It will be 0 only. Why? Think about this!!!



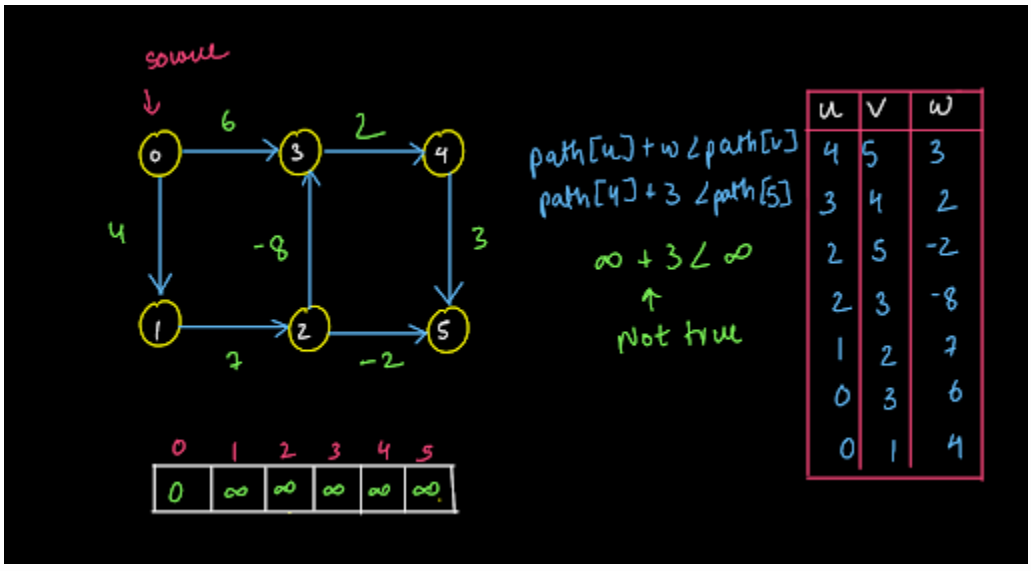
Relaxation Equation

We know that we will be given the edges as the input. Let us arrange these edges in a random order as shown below where "u" is the source "v" is the neighbor and "w" is the weight of the edges:

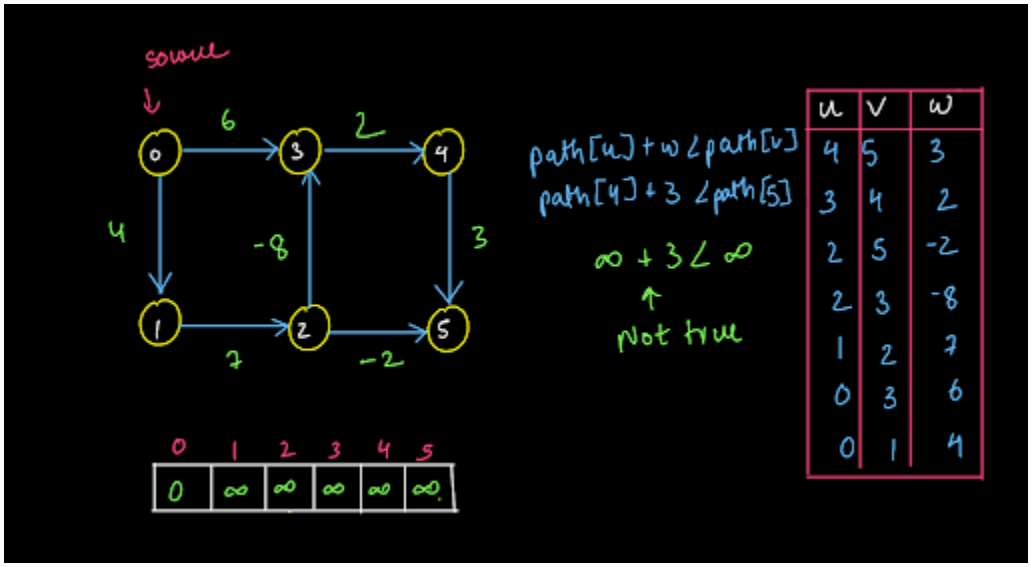


Now, we will start with the procedure of Bellman Ford Algorithm. **We have to perform a total of v-1 iterations. Here, v is the number of vertices of the graph.** Why v-1? We will talk about this later. In these v-1 iterations, we will check whether path[u] + w < path[v]. If it is not, we will continue and if it is then we will insert path[u] + w in path[v]. **This inequality i.e. path[u] + w < path[v] is called relaxation equation.**

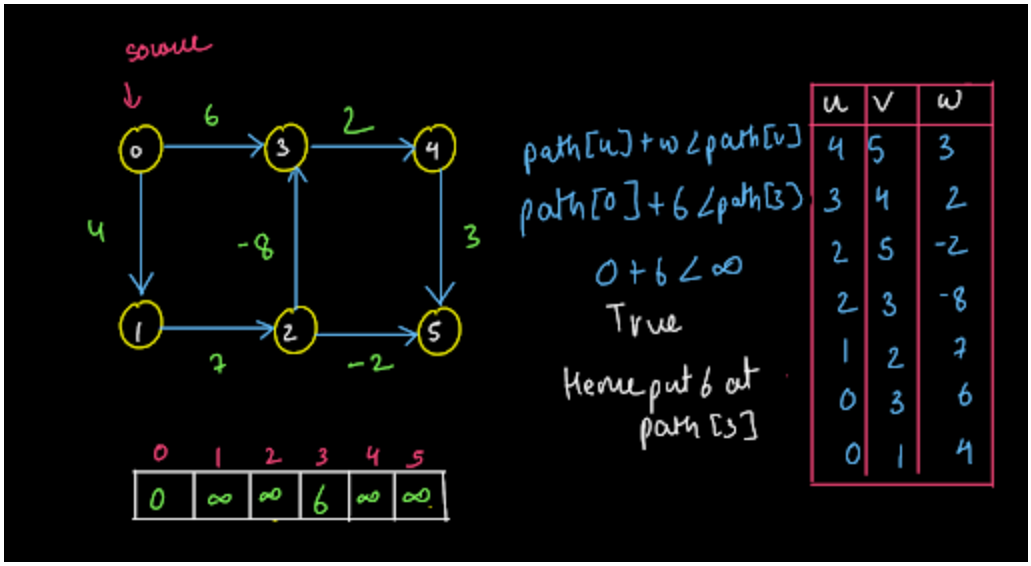
So, let us see the procedure once. Have a look at the diagram given below:



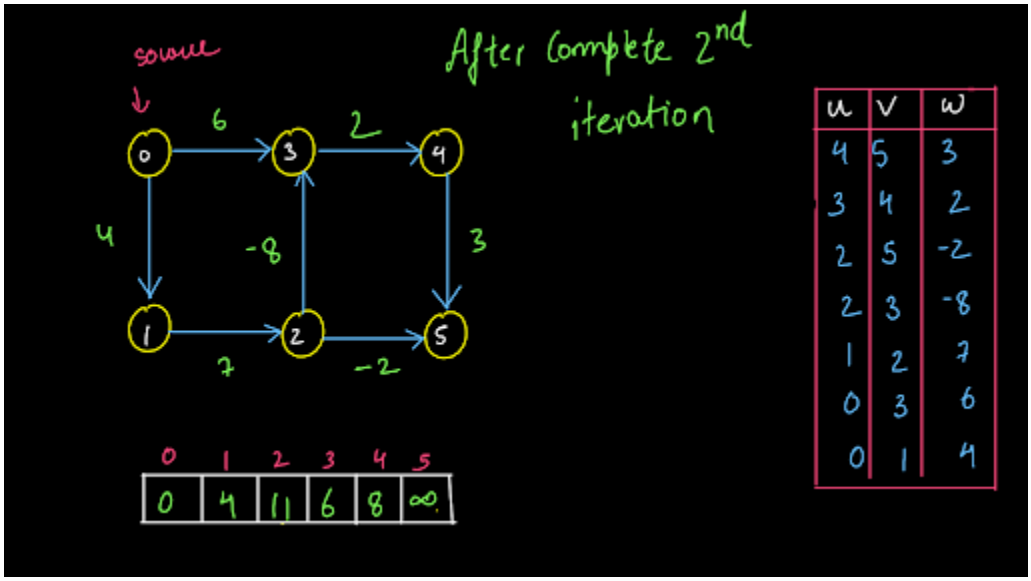
Here, the inequality does not hold true. Hence, we will not do anything and will continue the procedure. In fact, this inequality will not hold true till we reach u=0. So, let us directly move to it.



Here, u=0 and v=3 and w=6. The relaxation equation holds true here and hence we put path[u] + w at path[v]. The same will be the case for the next u-v-w triplet too. So, this was the first iteration. We have to perform a total of v-1 iterations. We have to apply the same procedure i.e. validate the relaxation equation for all the edges in every iteration. If path[u] + w < path[v] then path[v]=path[u] + w else continue the procedure. Let us perform one more iteration to get the procedure completely. Again, we start from the order that we have taken above and try to validate the relaxation equation. The first triplet where u=4, v=5 and w=3 will not validate our relaxation equation. So, let us start with the next triplet. Here path[u] + w < path[v] i.e. path[3] + 2 < path[4] i.e. 6+2 < infinity. Hence we will put 6+2=8 at path[4].

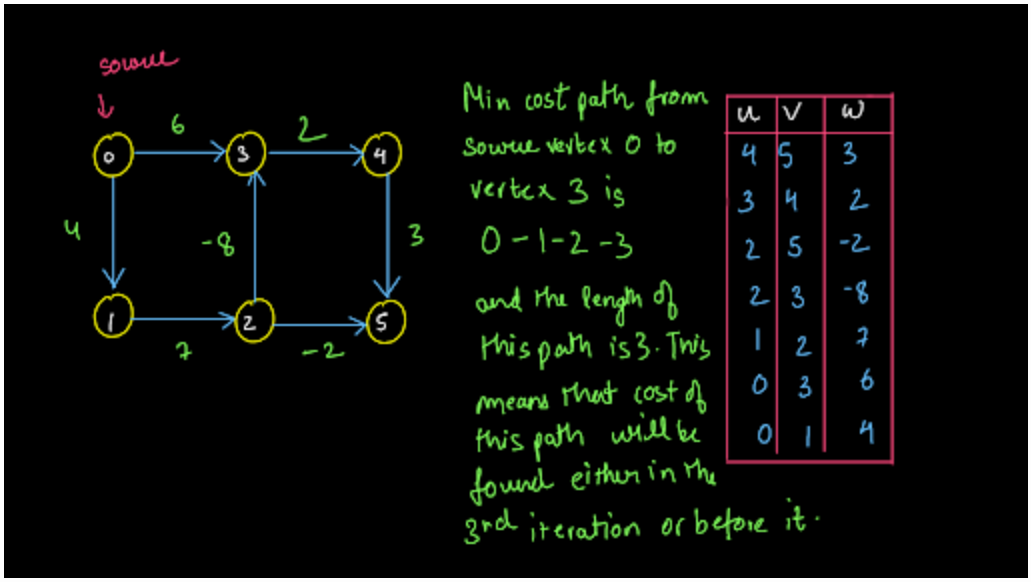


Similarly, in this iteration, a few more modifications will take place. Which triplets will validate the equation in this iteration? Try to find it out yourself. The array after this iteration is shown below:

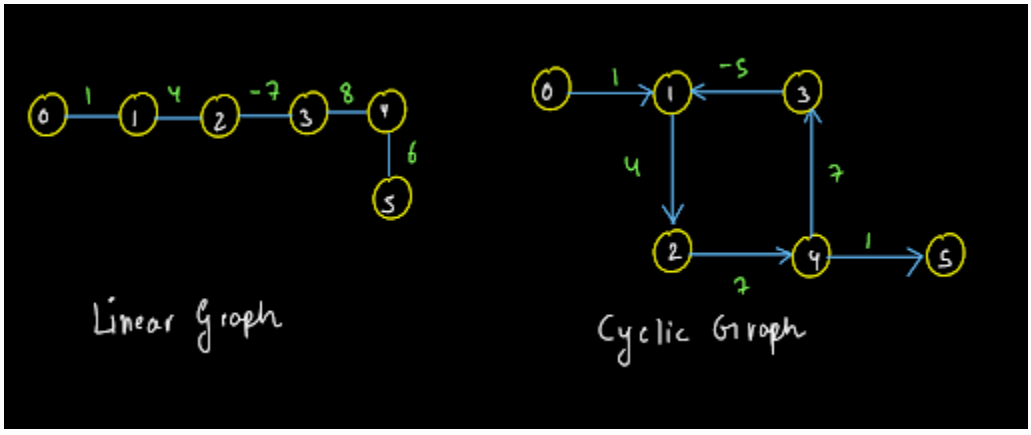


Why of the Problem

In Bellman Ford Algorithm, if the shortest path between two vertices has a length "i" then, the cost of that path will be found at or before the ith iteration.



So, if the length of the shortest path between two vertices is "i" then it will be found at or before the ith iteration. This is the main concept behind Bellman Ford Algorithm. Now the question is why we have v-1 iterations? If we have v-1 iterations this means that we are sure that the length of the path will not be more than v-1 as the i length path's cost is found in the ith iteration. So, how are we sure about this? Let us have a look at this too. Let us take two different and extreme cases of the length of the paths. We have a linear graph and a graph with a cycle as shown below:



The maximum length of a path that is possible for a linear graph is v-1 where v is the number of vertices in the graph. Hence, v-1 iterations will be enough for finding the minimum cost of all the paths. Now, you might think that the max length in a cyclic graph can be greater than v-1. Yes, it may be but it will not be. What? See, we are clear with the fact that the negative weight cycles are not allowed. So, the cycle will only have a positive weight overall. Hence, completing a cycle in a graph will not give us the minimum cost. There will always be a non-cyclic path with less cost. Hence, the max path length will be v-1 here too and therefore we have exactly v-1 iterations in this procedure.

Implementation

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/bellmanFord.cpp

Time Complexity - O(V*E), where V = no. of vertices in graph & E = no. of edges in graph

Space Complexity - O(1)

Application (Negative Weight Cycle Detection)

Problem

We are given 2 integers N and M as the input where N is the number of vertices in the graph and M is the number of edges. We are also given those M edges as input. We have to tell whether there is a negative cycle in the graph or not. If there is a cycle with negative weight then we will return 1 else 0. We have already talked about the negative weight cycle in our previous problem.

Why of the problem?

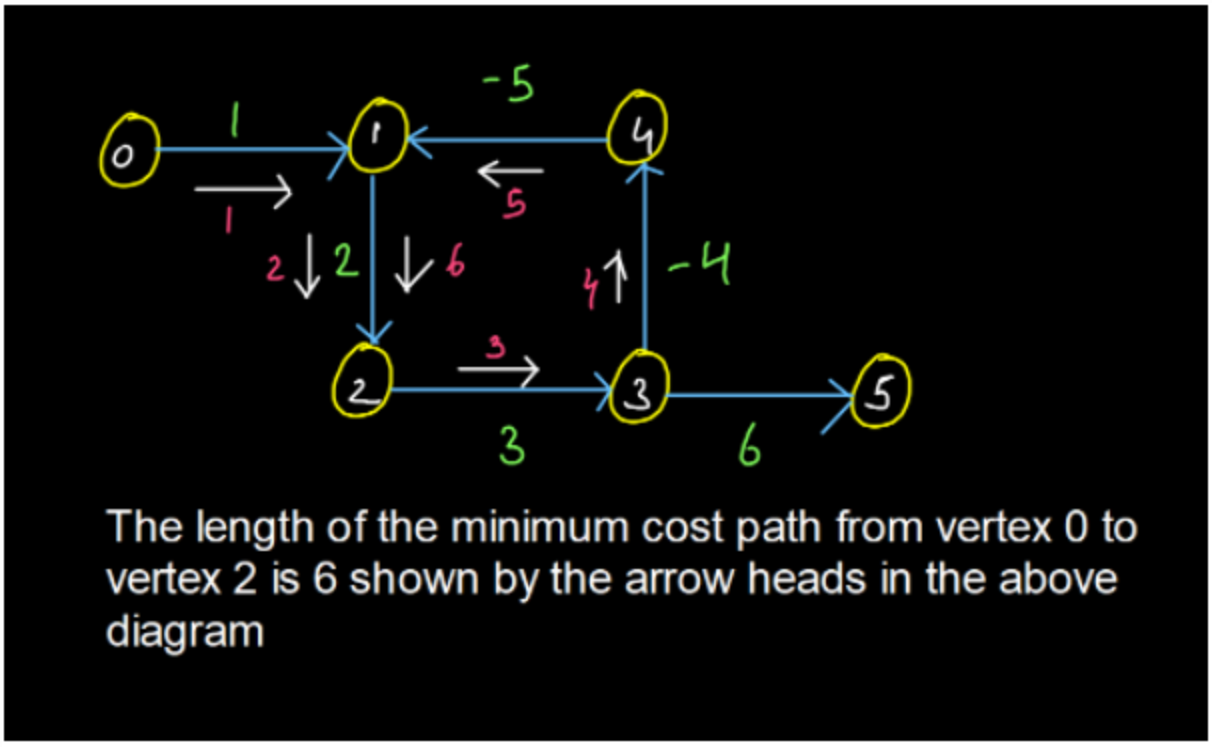
We know that we mostly discuss the what and how of the problem first but here we will first discuss the **why** of the problem. So, the procedure will be discussed in the **what** and the code in the **how** of the problem. But, in short, we will be using the Bellman Ford Algorithm to detect a negative cycle.

We have already studied in the previous problem that Bellman Ford Algorithm cannot detect the minimum cost path in a graph with negative weight cycle. Do you remember the reason why?

This is because if a cycle has an overall negative weight then we can move in the cycle an infinite number of times to add the negative cycle weight to the total weight and the cost/weight will keep on reducing.

We also studied that Bellman Ford algorithm has exactly v-1 iterations where v is the number of vertices in the graph. This is because of the fact that the maximum length of a path in the graph will be v-1. This holds true for linear graphs as well as for a cyclic graph as in a cyclic graph, we will never traverse the entire cycle as it will have a positive weight and it would add up to the cost and increase it and we want the minimum cost.

But, if we allow the negative weight cycle to be present in a graph, the Bellman Ford can have more than v-1 iterations as the length of the path can be greater than v-1. Have a look at the image shown below:



As in the above diagram, the minimum cost path from vertex 0 to vertex 2 is 0-1-2-3-4-1-2 and the total cost of this path is -1 and the length is 6. In fact, if we complete the cycle once again then the path length would be increased further. So, the shortest path from vertex 0 to vertex 2 in the above graph with a negative weight cycle will be of infinite length. This does not happen in a graph without a negative weight cycle. In a graph without negative weight cycle, we can get the answer in exact v-1 iterations and after that, the minimum cost path will not change for any vertex from the given source vertex. This is something that we will use to solve our problem.

What of the Problem?

Now that we know the **why** of the problem, we can easily figure out how the Bellman Ford Algorithm can help us detect a negative cycle. So, as we have discussed above, a graph without a negative weight cycle will give the answer in

exactly v-1 iterations of Bellman Ford Algorithm and will not change the cost for any vertex even if we apply any more iterations.

This is not the case with a negative weight cycle graph. The answer will keep on changing after v-1 iterations also. So, we will use this to detect whether there is a negative cycle in the graph or not.

The steps that we would follow are given below:

1. Run the first v-1 iterations of Bellman Ford Algorithm as we used to do it.
2. After this, perform just one more iteration. If the minimum cost value changes for any vertex from the given source vertex, the graph has a negative weight cycle and so we will return 1.
3. If there is no change in the minimum cost of any vertex from the given source vertex after v iterations, the graph does not have a negative weight cycle and we will return 0.

Implementation

https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/negativeWeightCycle.cpp