Graphs

Content

1. Dijkstra Algorithm

The problem is named as "Shortest Path in Weights" . The algorithm, which we are going to learn today, is known as Dijkstra's

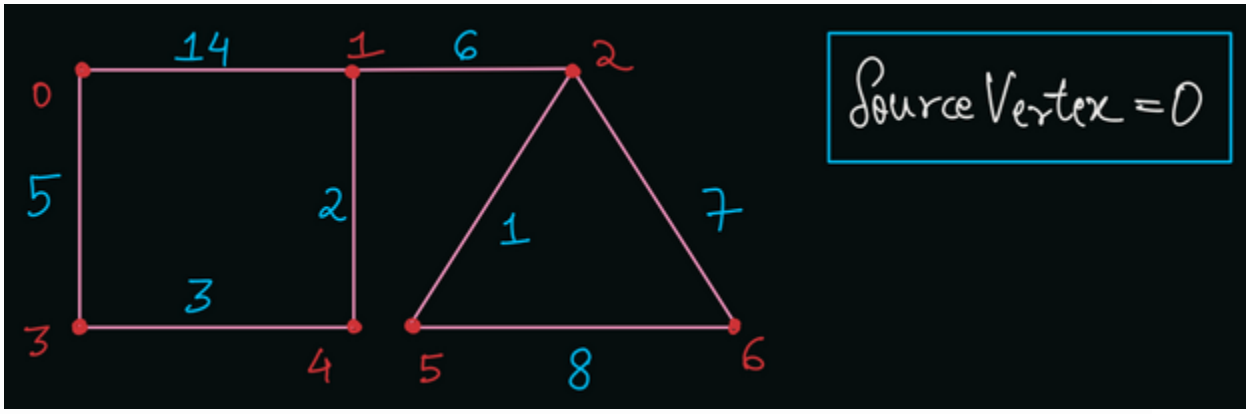
Algorithm

The algorithm is very intuitive, as it follows a **Greedy Approach.** The algorithm is also known as **"Single Source Shortest Paths Algorithm".**
So, enough talk, let's jump into the problem!

Problem

1. You are given an undirected graph and a source vertex. The vertices represent cities and the edges represent distance in kms.
2. The graph is weighted in nature. The weights of all edges are **non-negative.**
3. You are required to find the shortest path to each city (in terms of kms) from the source city along with the total distance on the path from source to destinations.

Example



Approach

1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

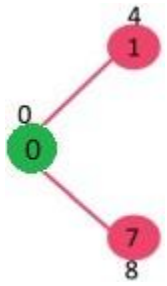
3) While *sptSet* doesn't include all vertices

….**a)** Pick a vertex u which is not there in *sptSet* and has a minimum distance value.

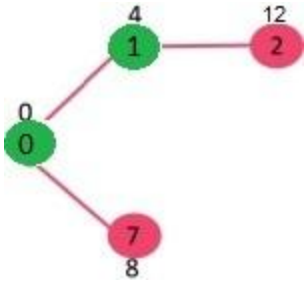
….**b)** Include u to *sptSet*.

….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

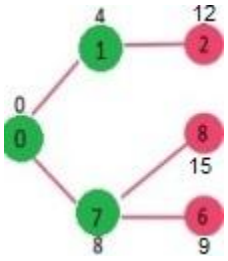
The set *sptSet* is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.

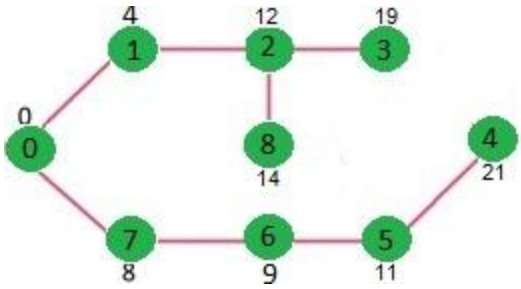


Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



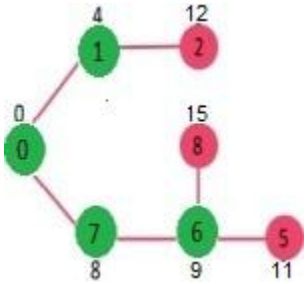
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.

We repeat the above steps until *sptSet* includes all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).

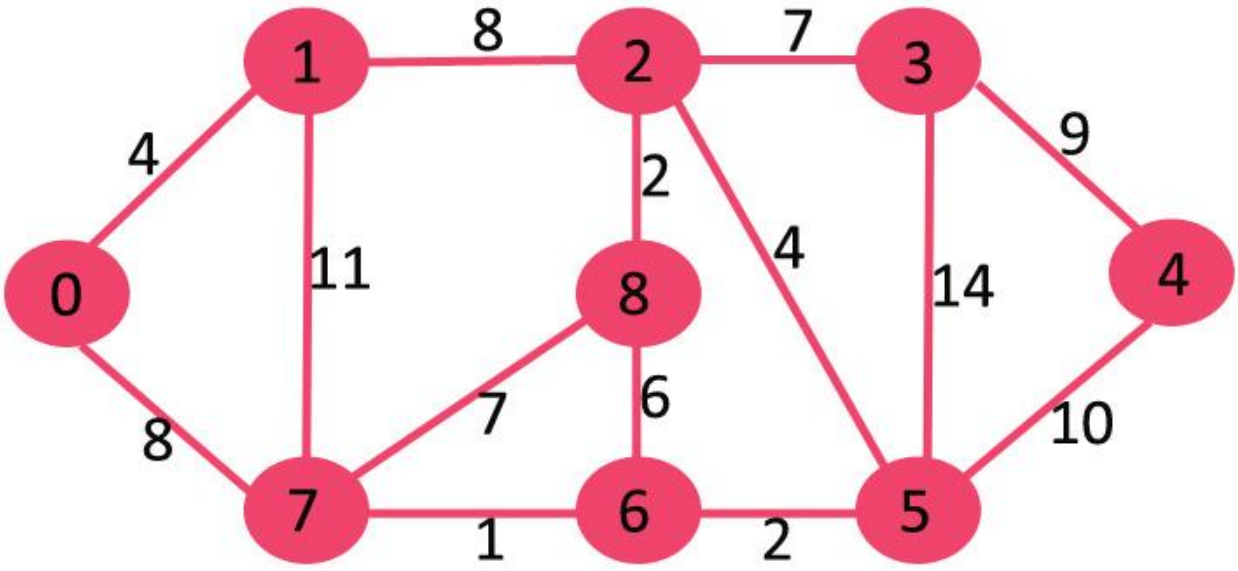


How to implement the above algorithm?

We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store the shortest distance values of all vertices.



Let us understand with the following example:



Implementation

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

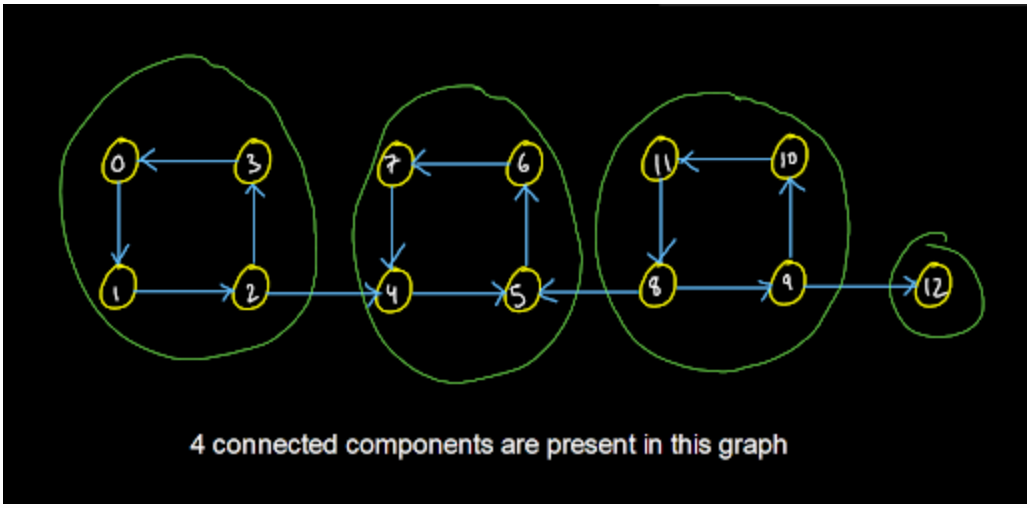
2. Kosaraju Algorithm

Problem

We are given two integers N and M where N is the number of vertices and M is the number of edges of a directed graph. We need to find the number of Strongly Connected Components in the graph. What is a strongly connected component?

Strongly connected component

A strongly connected component of a di-graph (directed graph) is a component in which we can visit all the vertices from any vertex in the component. For instance, have a look at the diagram given below:



In each of the strongly connected components above, we can see that we can reach every vertex from every other vertex. Note that strongly connected components is a concept of directed graphs only. In non-directed graphs, the edges are bi-directional due to which there is no such concept of strongly connected components and we only have connected components

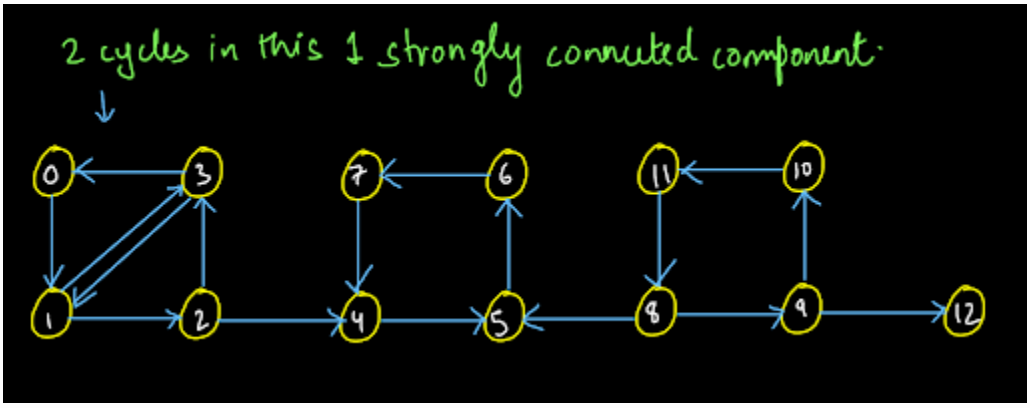
Note that we have considered a single vertex also as a strongly connected component. This means that every vertex should be considered as a strongly connected component. Then, the answer should not be 4 rather 13 + 3= 16. Well, if you have thought this far, give yourself a pat on the back. But, the Kosaraju Algorithm finds the minimum number of strongly connected components and the answer to that is 4 only. Why? Think!!!

We recommend you refer to the Kosaraju Algorithm video to understand the problem completely and also understand the meaning of strongly connected components.

Cycles and Strongly Connected Components

First of all, we have to make you very clear with the fact that the number of cycles in a directed graph is not equal to the number of strongly connected components. Though, from the above diagram, it may look like they are equal but they are not.

It is true that we can find a strongly connected component in a graph only if we have a cycle in the graph as a cycle causes every vertex to connect to every other vertex but it is also true that the number of cycles is not equal to the number of strongly connected components. Have a look at the diagram given below:



So, it is clear now that the number of strongly connected component is not equal to the number of cycles.

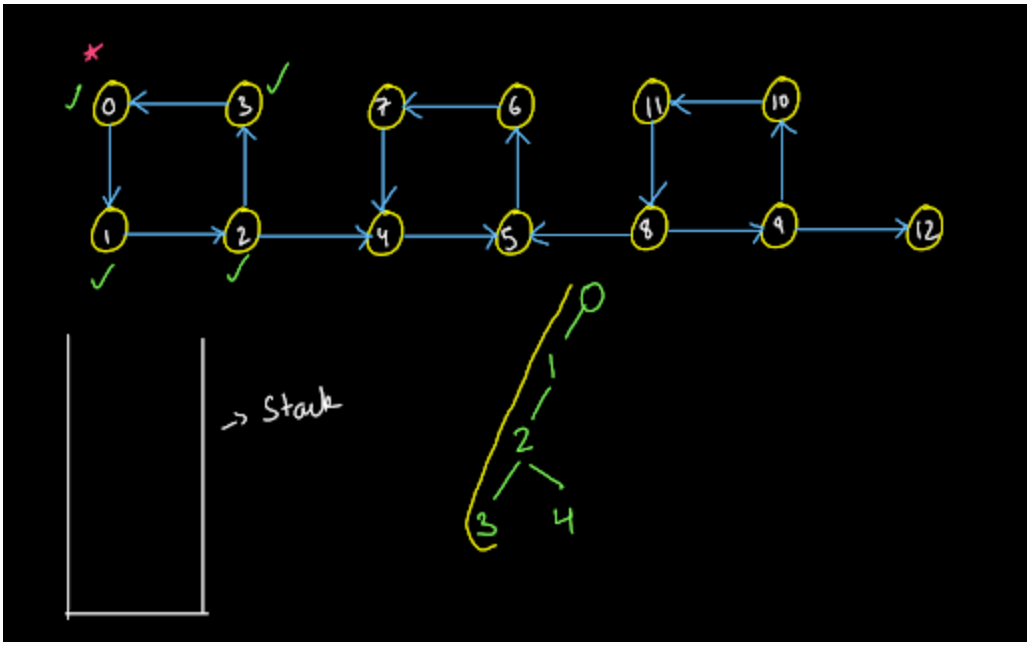
What of the problem

The Kosaraju Algorithm Process

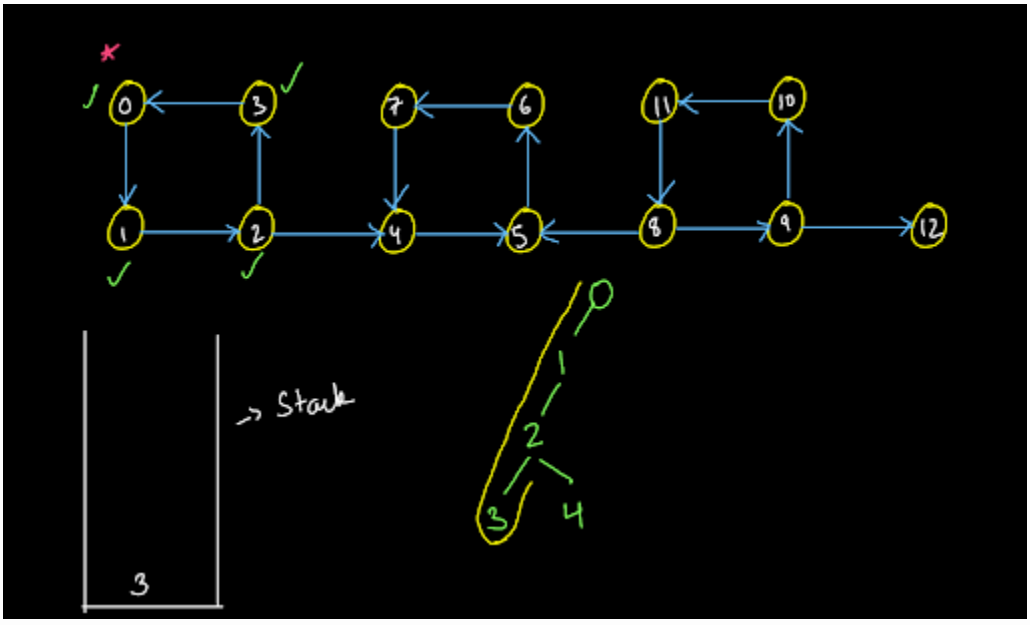
Apply DFS and Fill the Stack

We will choose any random vertex in the graph and start the DFS. Also, while backtracking i.e. while returning back from recursion, we have to add that vertex into a stack that we will create. So, let us start the procedure.

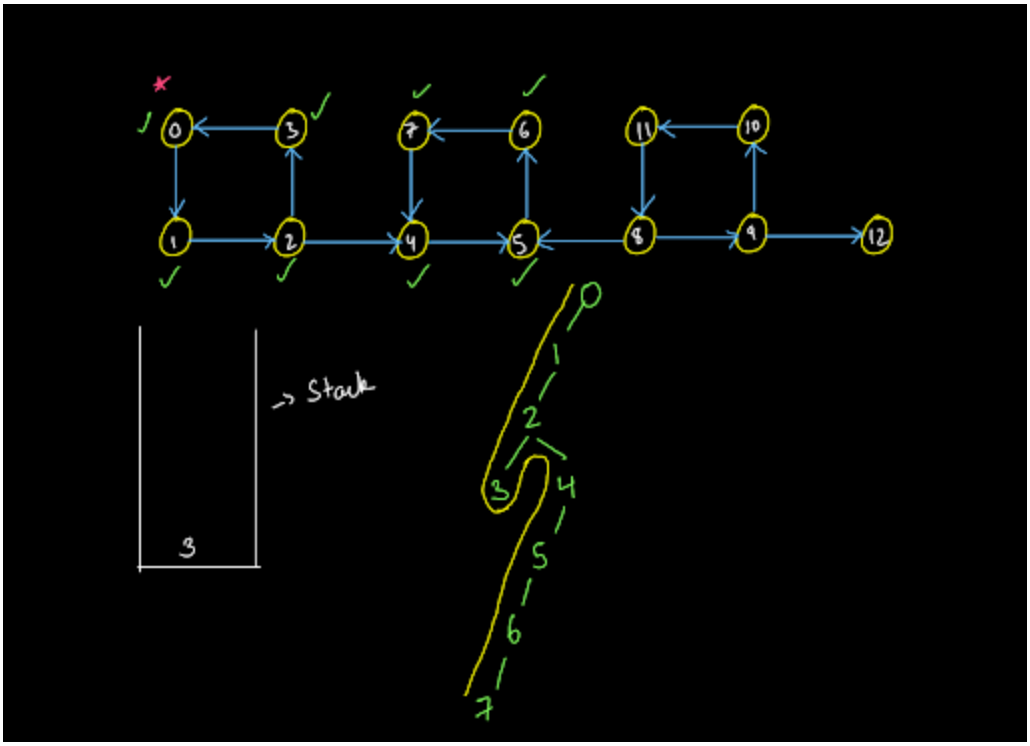
As already discussed, we can choose any vertex to start our DFS with. Hence, we have selected the vertex 0. The DFS of vertex 0 is also shown in the diagram below.



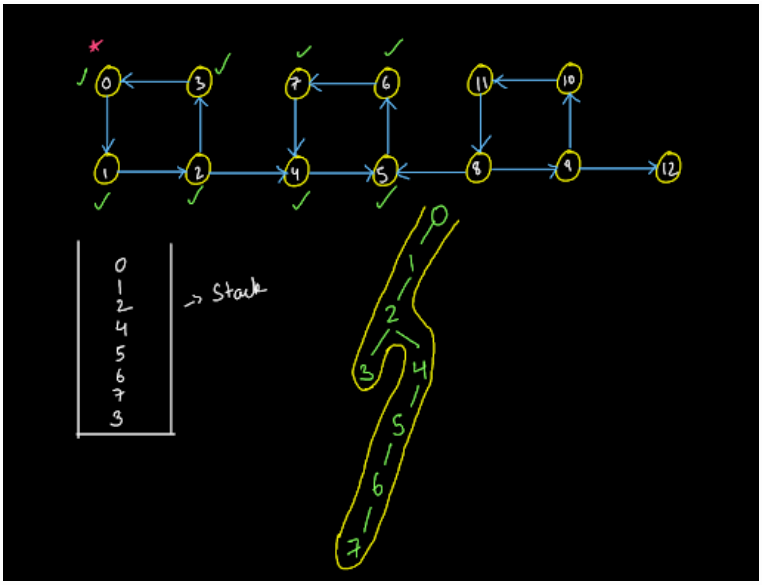
So, as you can see that we started from vertex 0 and performed the Depth First Traversal of the graph and kept on marking the vertices that we have already visited. When we reach the vertex 3, we do not have any unvisited vertex left. Hence now we will backtrack. While backtracking, we will add the vertex to the stack as shown below:



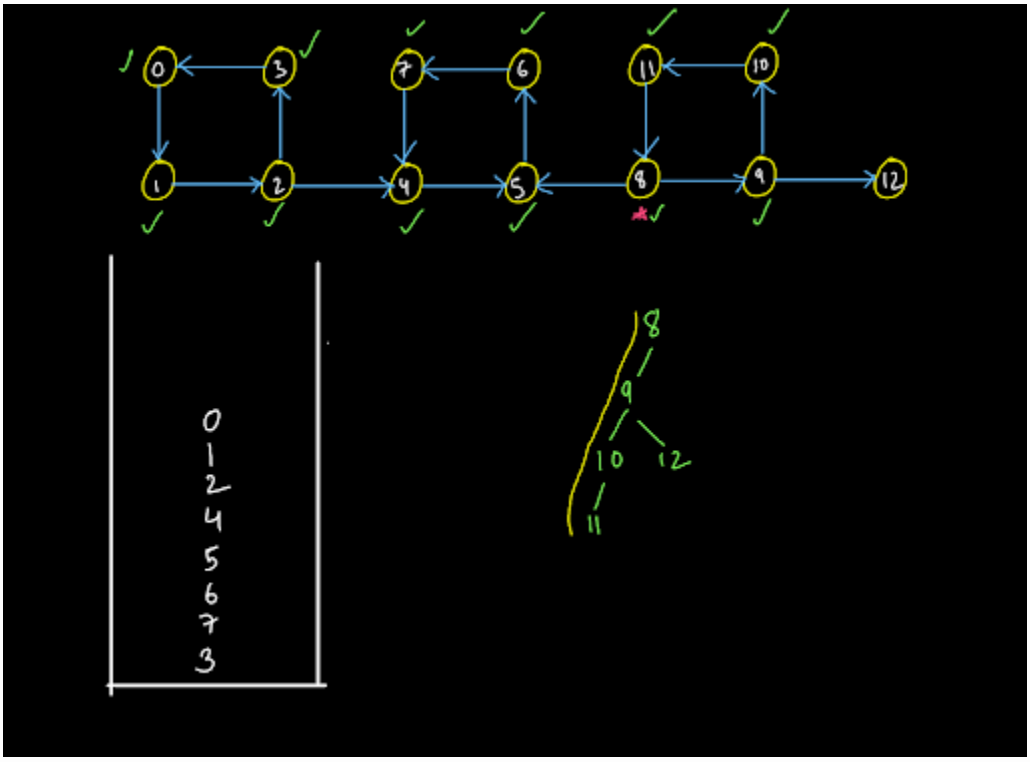
Now, let us continue with the DFS further.



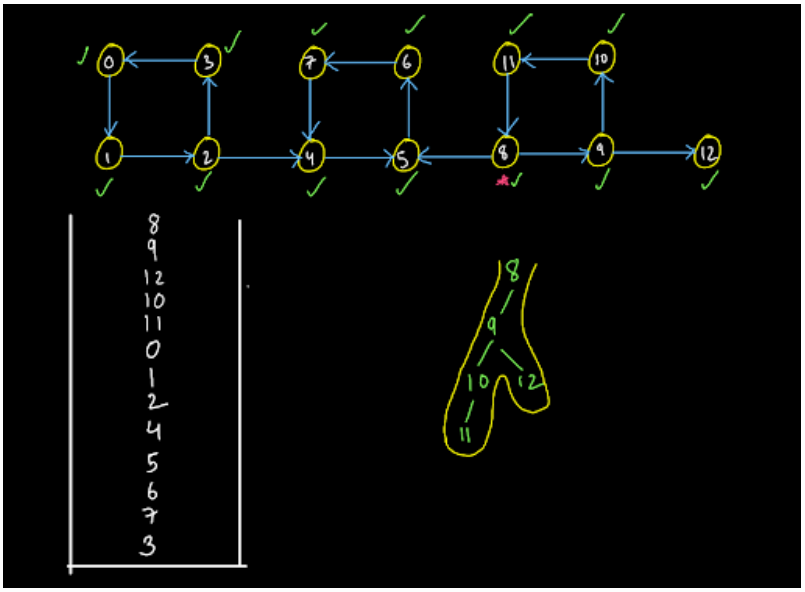
Now we have reached the vertex 7. From here, we cannot visit any other vertex as the only neighbor of 7 is 4 and it is already visited. So, we will backtrack and reach the vertex 0 and while backtracking, we add the vertices into the stack as shown below:



Now, have we visited every vertex of the graph? No! we haven't. So, let us choose any random vertex again from the remaining vertices and apply the same procedure. So, let us choose vertex 8 now. The DFS is shown below:



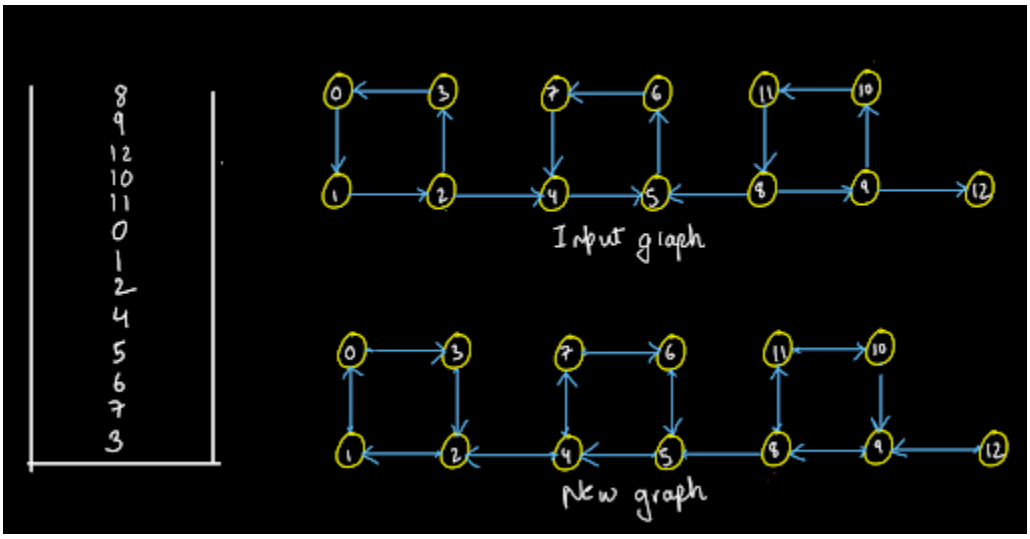
Now, we will again backtrack and add all the vertices into the stack. The vertex 12 has not been visited yet. So, it will also be visited and then backtracked. Finally, the stack will look as shown in the figure below and we would have traversed the entire graph.



Let us now see the next step of this procedure.

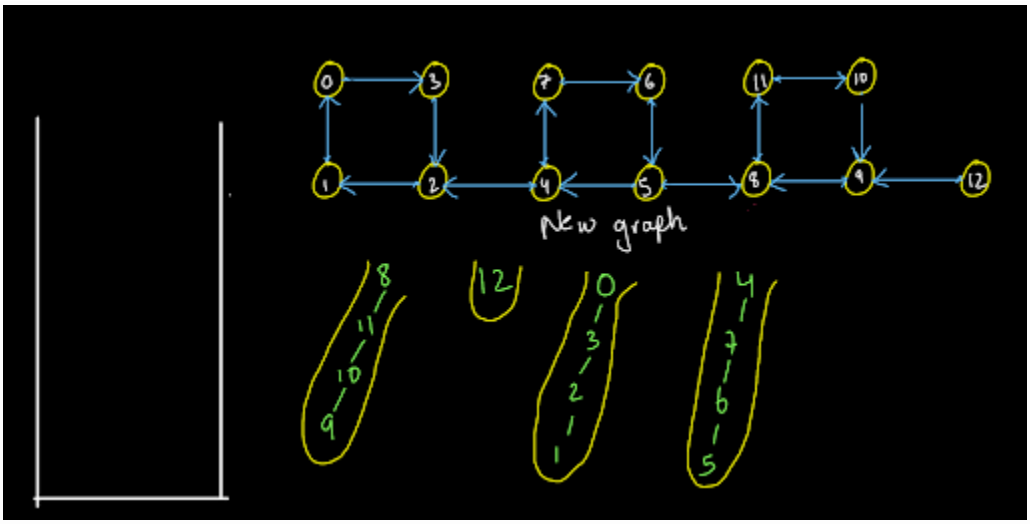
Reverse the Edges and Apply DFS

We will now create a new graph which will have its edges in the reverse direction of the edges of the input graph. This is shown in the figure below:



Now, we will apply the same procedure that we applied in GET CONNECTED COMPONENTS problem in foundation. Do you remember it? If don't we recommend you refer to the solution video for it here.

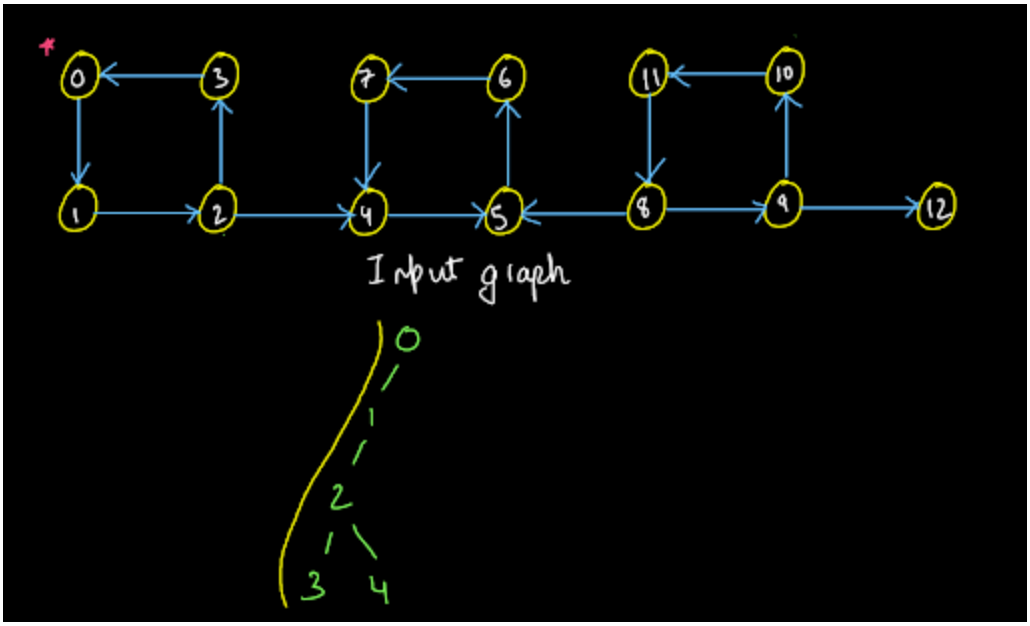
We will apply the DFS for every vertex. The only difference is that now we will take out the vertex from the stack and apply the DFS on it. Once we complete one DFS, we will increment the count variable depicting the count of strongly connected components in the graph. We have shown all the trees of DFS below and this number of trees is actually the number of strongly connected components.



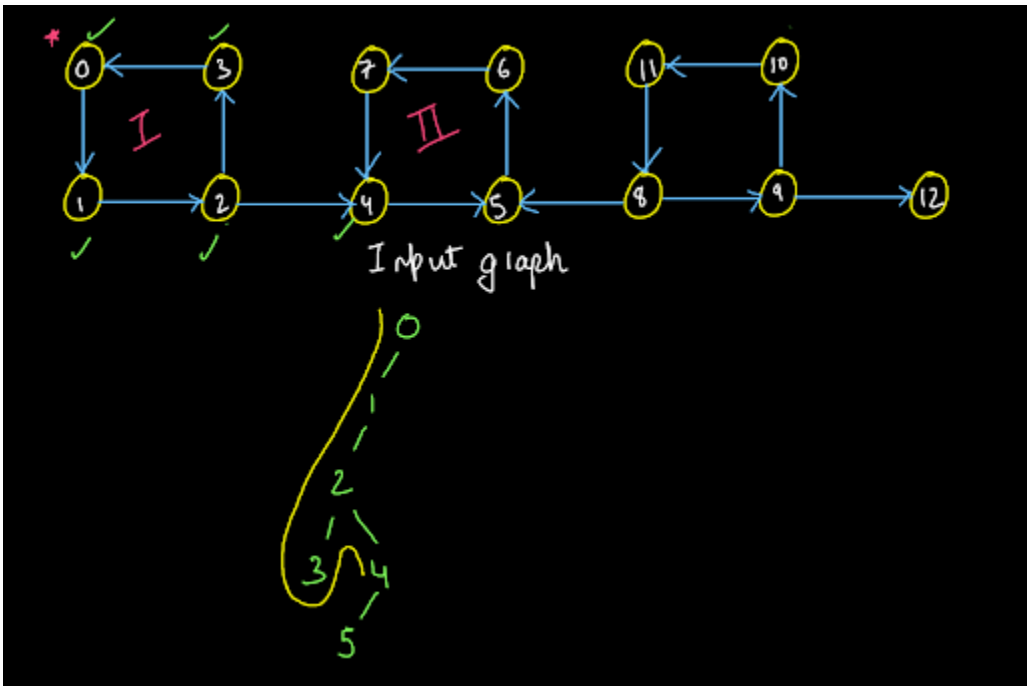
We recommend you refer to the solution video (5:35-11:28) to understand the above procedure. Also, you should have expertise over the get connected components problem. Only then, you will be able to get the complete insight into this procedure. Now, let us move to the why of the procedure.

Why of the Problem?

We have understood the procedure. Now, it is the time to understand **why** that procedure works or what we are trying to do in this procedure. See, strongly connected components are still like connected components and the method for finding both of them should be similar i.e. applying DFS at every vertex. But the problem was that if we choose any random vertex to apply DFS, we will end up with a wrong answer in case of strongly connected components. Why? Have a look at the figure shown below:



We start from the vertex 0 and apply DFS. We visit all the neighbors in the usual way we do in DFS. Now, we reach vertex 3 and its neighbor is already visited. So, now is the time to backtrack. When we backtrack from vertex 3, we will reach vertex 4 and start its DFS as shown in the figure below:

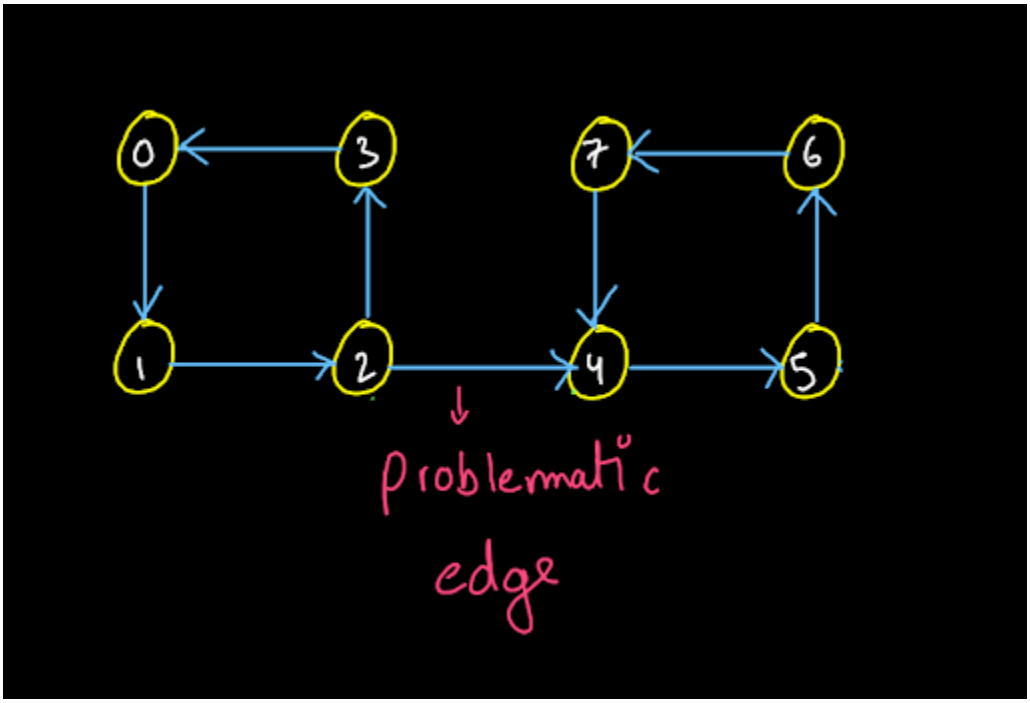


This is wrong as when this happens, we are actually counting both 1st and 2nd component as a single strongly connected component which they are not. Kosaraju figured this out and he understood that we cannot start applying the DFS from any vertex in the graph. We need to have a special order of vertices and then apply the DFS on the vertices in that order.

So, the procedure that we did with the stack that we created generated that particular order in which DFS can be applied. Now, let us try to understand one more thing that why this stack order and reversing the edges of the graph works.

Why the Stack Order and Reversing Graph's Edges Work?

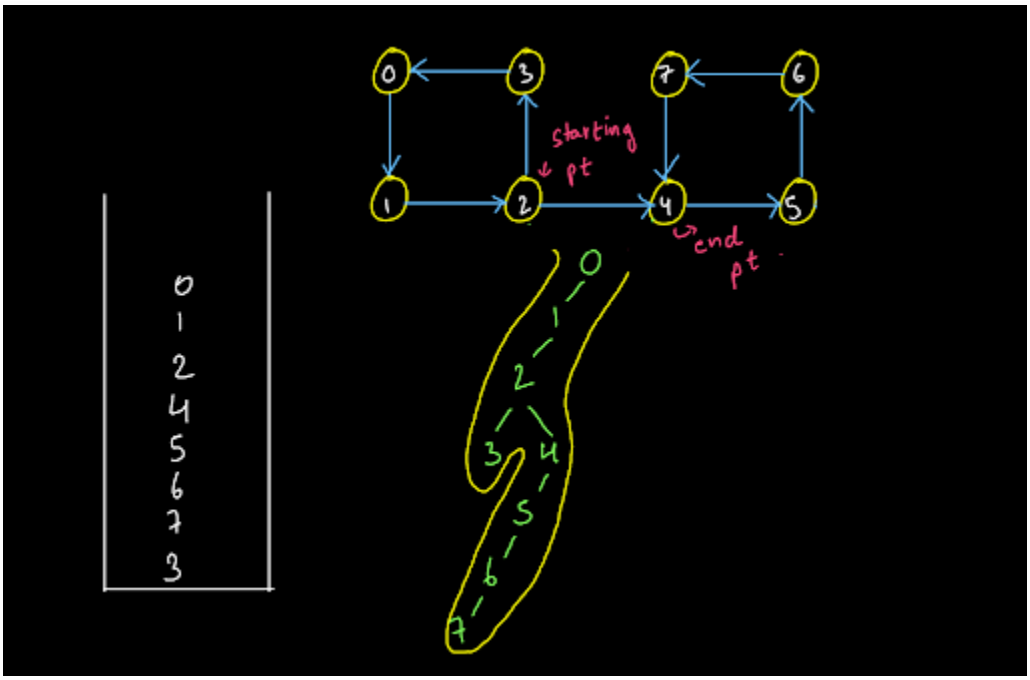
Let us take only a part of our input graph as shown in the figure.



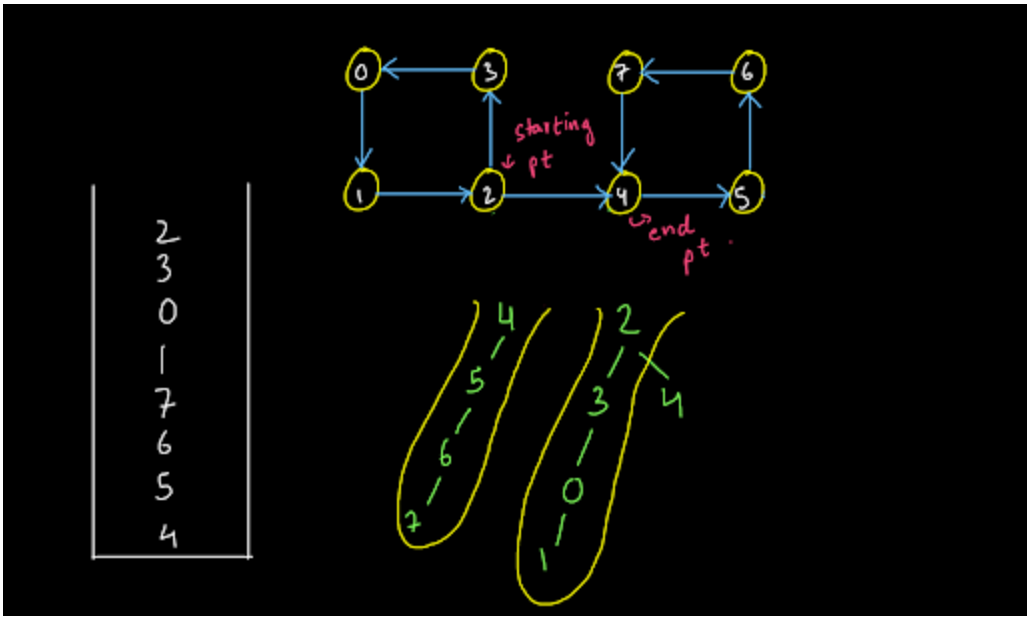
These are two different strongly connected components but as we have seen above, this problematic edge causes the normal DFS procedure to count them as one edge.

Let us have a look back at our procedure. To find the specific order, we chose a random edge to start DFS with and while backtracking, we added it into the stack. Whenever we choose a vertex from (any random vertex) from the component 1 of the graph i.e. the component that is at the side of the starting point of the problematic edge, we will get that vertex at the top of the stack. Why? Think!!! For a hint, this will be due to backtracking.

Let us say we choose vertex 0, so you will see the vertex 0 at the top of the stack only after backtracking.



The interesting point is that even if we select any random vertex from the second component to start the DFS with i.e. from the component that lies on the end point of the problematic edge, we will still get a vertex of the first component only at the top of the stack as shown below:



As shown above, we start from the vertex 4 and apply DFS. When thee vertices in component 2 are all visited, we cannot reach component 1 from it. So, we have to again, select any random vertex from component 1 and apply DFS on it separately. Also, since the second component is already visited, we don't reach it again and the vertex of the 1st component that we selected is again at the top of the stack.

So, we conclude that the vertex of the component at the side of the staring point of the problematic edge will always be at the top of the stack whether you start DFS from any vertex.

Now, we have reversed the edges of the graph. Note that strongly connected components are formed due to cycles in the graph. They are not equal to the number of cycles but they are formed as a result of cycles. Even after reversing the direction of the edges, the cycles still remain, just their directions are reversed and hence the strongly connected components remain as it is even after reversing the edges.

When we apply DFS on this graph, the vertex at the top of the stack is from component 1 and it will be traversed completely but we will never reach

component 2 as the direction of the problematic edge is reversed. Hence component 1 will be counted as a separate strongly connected component and so will be the component 2.

Hence, by following the specific order for DFS and reversing the edges of the graph, we have overcome the problem caused by the unidirectional edge that connects 2 strongly connected components. This is the concept of Kosaraju Algorithm.

We recommend you refer to the solution video to understand the why of the problem completely. Now that we have understood the what and why of the problem, let us know the how by coding it.

Implementation

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