Graphs

Content

Applications of Depth First Search

Applications of Depth First Search

1. **Check if a path is present between source to destination**

   Approach
   The problem here is to check whether there is any path present between src and dest. To solve this problem we will use Recursion.
   We know that if there is a path between src vertex and dest vertex then there must also exist a path between src's neighbor vertex and dest vertex too. So we can develop some expectation and faith for this problem. **The expectation** from the hasPath(src, dest) function is that it will return us a Boolean value, indicating whether the path exists between src and dest.
   Keeping **the faith**> on the recursive call, hasPath(nbr, dest), that it works perfectly and will tell us whether there exists a path between src's neighbor and dest vertex. And **meet this expectation with faith** by trusting the faith's result and returning the same result.

   If hasPath(nbr, dest) returns false then no path exists between nbr and dest. So, it implies that no path exists between src and dest too. And if it returns true then it means path exists between nbr and dest which means path also exists between src and dest.

   And if src has n number of neighbors then we call hasPath() for each of the neighboring vertex unless a true is returned, and if true is received through any call then the further iterations are halted and true is returned.

   And the base case for this function will be when src becomes equal to dest. It means we have reached our destination and therefore we return true. But if we notice, as this is a recursive function so we can get to see some redundant calls. As src will make recursive calls to each of its neighbors, suppose that it made the first call to its neighbor 1, nbr1. So now this nbr1 will further make calls to each of its neighbors.

   And not to forget, src is also one of nbr1's neighbors. So src will again make a recursive call to its neighbor and first of all nbr1 will be called. And it will give us a run time error of stack overflow. So, to handle this situation, we make use of the Boolean array, visited, which is already present in the signature of the hasPath() function in the code editor.

Every time we enter the hasPath() function corresponding to a certain src, we update the visited array's src index value to true.

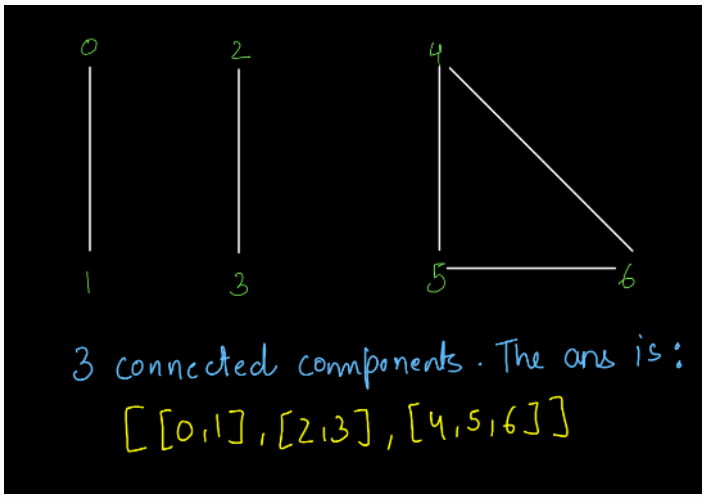
And before making a call for any neighbor, we first check for the corresponding value of this neighbor's visited array. And make a call only if it's false. As it implies that this vertex is being visited for the first time.

Implementation

Visit the link - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/has_path_using_dfs.cpp

2. **Get all the connected components for a given graph**.

Examples



Approach

Let us start from the first vertex i.e. vertex 0. We will create a new arraylist as we are going to traverse a new component. Now, we are at vertex 0. We will mark it visited and then add it to the new ArrayList that we have created. Then, by applying DFS, we will reach its neighboring vertex. Since the only neighbor of vertex 0 is 1, we will reach vertex 1.

Remember that this is the DFS of a graph. We will visit only those neighbors that are unvisited. Here, the only neighbor is 1 and it is unvisited, so we move to it.
Repeat the same steps as above i.e. add the vertex to the component ArrayList and mark it visited. Now, 1 has only one neighbor i.e. vertex 0 and it is already visited. So, we can not go there. Since there is no more neighboring vertex to visit, this component is

complete and we will add this component's ArrayList to our answer ArrayList of ArrayList.

Now, as discussed above, we have to apply DFS for every vertex. So, we will move to the next vertex that is vertex 1. Now, this vertex is already marked visited. So, we will not apply DFS on it.
We move to the next vertex i.e. vertex 2. Since this vertex is not visited, we apply DFS on it and create a new ArrayList as we are traversing into a new component.
Note: We get to know that we are traversing a new component as after traversing 0 and 1, we did not have any neighbor to traverse to but there are a total of 7 vertices in the graph numbered 0-6. This means that 0 and 1 were part of one connected component that is not connected to the rest of the graph.

So, as we can see from the image above, we have traversed the vertex 2 and marked it visited. Now, by applying DFS, we move to the next and the only unvisited neighbor is vertex 3.So, we have filled one more component. We applied the DFS for vertex 3. Now, let us apply the DFS for vertex 4. So, we see that vertex 4 is also already visited. So, we won't apply DFS at vertex 4 also.

Now, let us move at vertex 5. Here again, we have created a new ArrayList and in that ArrayList, we have added 5 and marked it as visited.Here, we have 2 neighbors that are unvisited. So, we can go to any neighbor that comes in the Euler path. So, we will now visit vertex 5.Now, vertex 5 has only one unvisited neighbor i.e. vertex 6. So, we will visit this vertex too. Also, after visiting vertex 6, we have no other vertex to go to. So, this component will also be completed and added to our answer ArrayList.
So, we have completed the DFS for vertex 4 as well. Now, let us try the DFS for vertex 5. Since it is already visited, we will not traverse it and the same will be the case for vertex 6.
Now that we have applied DFS on all the vertices, we have got our answer. This is how we will find the connected components of a graph.

Implementation

Visit the link - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/get_connected_components.cpp

3. Print all paths from a source to destination

Approach

First, we will make printAllPaths() function and its return type will be void. And in addition, we add String in the argument of the function which will store the path covered so far.
Changing this return type to void also implies that printAllPaths() function returns nothing therefore it makes no sense to capture the result of its recursive calls.Which also implies the invalidation of statements 6 and 7 from above code for printAllPaths() function.

Moving to the most important point; even after doing the above changes, we may get a path or few printed but what about all possible paths.
To take care of this, it is really important that we explore all possibilities.

To print all paths, we need to allow even the once visited vertex to contribute to other possible paths. For this, it's important to set the value, corresponding to src, in the visited array as false, after we explore all possibilities through the src vertex.

Implementation

Visit the link - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/print_all_paths_from_source_to_destination.cpp

4. Print Hamiltonian Path and cycle for a graph

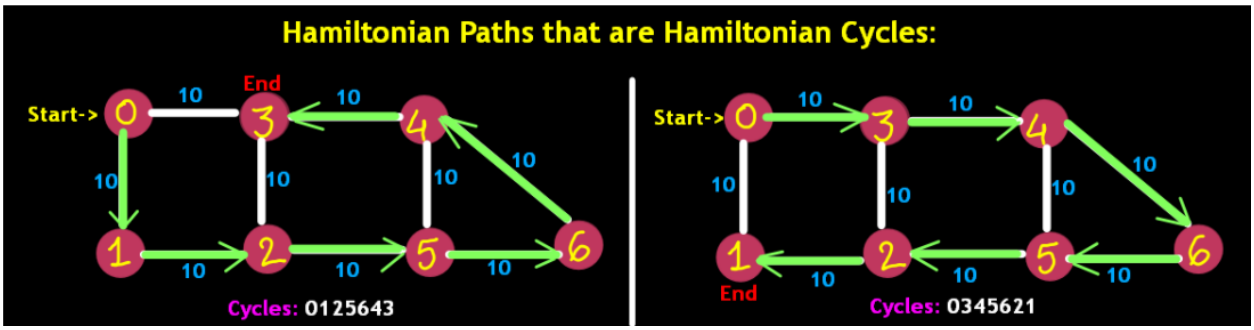
Approach
So friend, before we jump to the code of this problem, let me tell you that this problem is very similar to the Print All Paths problem.

In Print All Paths, we used to find the path between a given source and destination vertex. Each time we visited any vertex, we used to set the value corresponding to the vertex as true in the visited array.

And when we were done exploring all neighbor's of this vertex then we again set the value as false corresponding to this vertex in the visited array.
Also if the base case was hit, that is when source becomes equal to destination, we printed the path so far.We will follow a similar approach in this problem as well.But this time the base case will change.

Since we need to visit each vertex of the graph this time, the base case will become the stage where all vertices have been visited.

Also, also, also, we need to check that the path obtained is Hamiltonian Path or Hamiltonian Cycle. For doing so, we check whether there is an edge between source and last visited vertex.



Implementation

Visit the link - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/hamiltonian_path_and_cycle.cpp