Graphs

Content

Applications of Breadth First Search

Applications of Breadth First Search

1. **Find the shortest path from a source to other vertices in an unweighted graph.**

   Approach

   If there are no negative weight cycles, then we can solve in O(E + VLogV) time using Dijkstra's algorithm. Since the graph is unweighted, we can solve this problem in O(V + E) time. The idea is to use a modified version of Breadth-first search in which we keep storing the predecessor of a given vertex while doing the breadth-first search. We first initialize an array dist[0, 1, …., v-1] such that dist[i] stores the distance of vertex i from the source vertex and array pred[0, 1, ….., v-1] such that pred[i] represents the immediate predecessor of the vertex i in the breadth-first search starting from the source. Now we get the length of the path from source to any other vertex in O(1) time from array d, and for printing the path from source to any vertex we can use array p and that will take O(V) time in the worst case as V is the size of array P. So most of the time of the algorithm is spent in doing the Breadth-first search from a given source which we know takes O(V+E) time. Thus the time complexity of our algorithm is O(V+E).

   Implementation

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

2. **Get Path from source to destination**

   Approach

1. Create a queue which will store path(s) of type vector
2. Initialise the queue with first path starting from src
3. Now run a loop till queue is not empty
4. Get the frontmost path from queue
5. Check if the lastnode of this path is destination

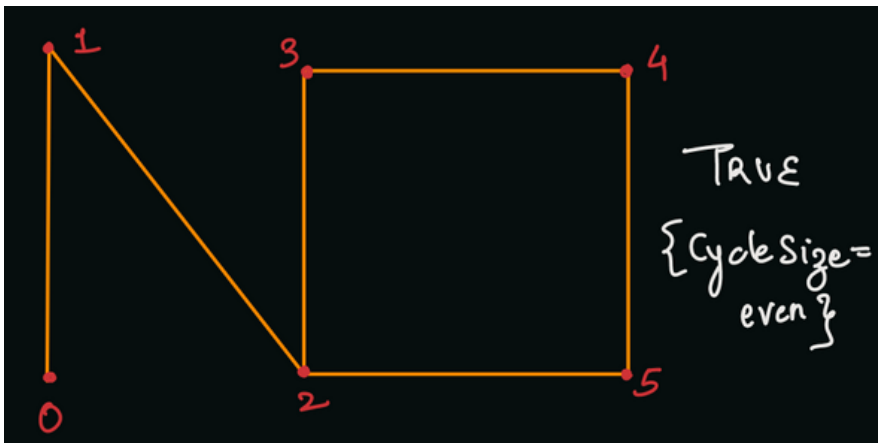
6. If true then print the path
7. Run a loop for all the vertices connected to the
8. Current vertex i.e. lastnode extracted from path
9. If the vertex is not visited in current path
    a) create a new path from earlier path and
       append this vertex
    b) insert this new path to queue

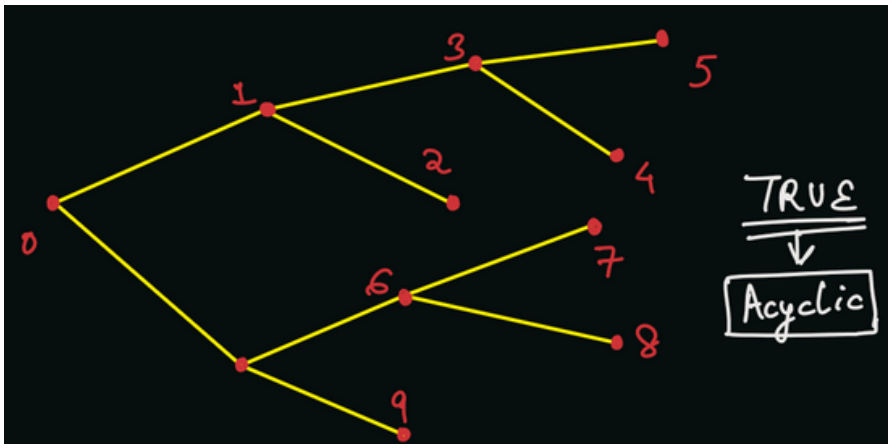
Implementation

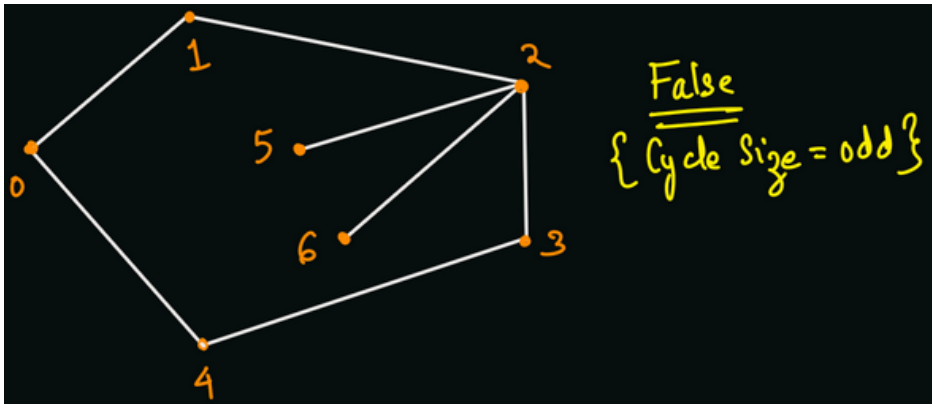
Visit the link - https://github.com/lavishabhambri/Weekly-Algo-Newsletter/blob/main/Graphs/Codes/get_path_using_BFS.cpp

3. Check if a given graph is Bipartite or not

Examples







Approach

According to the definition, we need to distribute vertices into two subsets which are mutually exclusive: No vertex should appear in both the sets, or the intersection of sets should be empty.

Also, they need to be exhaustive: Every vertex should appear in either one of the two sets, or the union of sets should be the entire set {0, 1, 2, ... n-1}.

Let us first analyze WHAT we will have to do in the algorithm.

1. If the graph is acyclic, i.e. there is no cycle present in the graph, then the graph is always bipartite.
2. If there exists any one cycle of odd length, i.e. the number of vertices in the cycle are odd in counting, then the graph is not bipartite.
3. If all the cycles (if present) are of even length, i.e. the number of vertices in each cycle are even in counting, then the graph is bipartite.

The algorithm, which we are going to use to check bipartiteness of a graph is known as graph-coloring algorithm, which uses BFS traversals. A graph is bipartite if and only if it is **two-colorable.**

Now, we will discuss **HOW** to check the presence of odd or even length cycles in the graph. We will start a breadth-first traversal, starting from each vertex which hasn't been visited yet. But, instead of just pushing the node's value into the queue, we will also push a visiting time of the node.We will try to add the nodes with even visiting time in the first set, and the nodes with odd visiting time in the second set.We know already that during the bfs traversal, if we find any node which is already visited, then there exists a cycle.

If the length of cycle is even, then the visiting time of the last node will be the same from both the paths (if even from the first path, then it will be even from the second path as well, and vice-versa).

Else, since the length of cycle is odd, the visiting time of the last node in the cycle will be different from both the paths (if even from the first path, then it will be odd from the second path, and vice-versa).

Hence, we cannot put the last node in either the first set or the second set. Thus, the graph will not be bipartite in an odd-length cycle case.

You can clearly see in the example, that the last node of the odd-length cycle in the above case is 4.

1. If we place node = 4 in the odd set, then we will have an edge 2-4 within the set.
2. If we place node = 4 in the even set, then we will have an edge 3-4 within the set.
3. Hence, we cannot place the node = 4 in either of the two sets.

Once we've visited all vertices, and did not find any odd-length cycle, and successfully assigned them to one of the two disjoint sets, we know that the graph is **bipartite** and we have constructed its partitioning.

Note: If you get this problem in a face-to-face interview, then you must ask about the corner cases like:

Q) Is the graph connected, i.e. there is only one single-connected component?

Q) Can there be edges of type (u, u), i.e. does self loops exist in the graph ?

Q) Can there be multiple parallel edges, i.e. does there exist more than 1 edge (u, v) in the graph ?

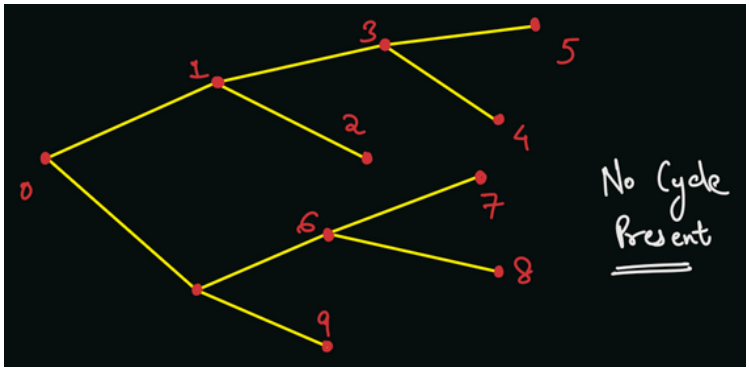
For simplicity, we can assume that there are no self loops and parallel edges in the current problem. But, the graph can have more than 1 component. Thus, we must call for the BFS traversal from each unvisited vertex in {0, 1, 2, 3, ... n-1}.

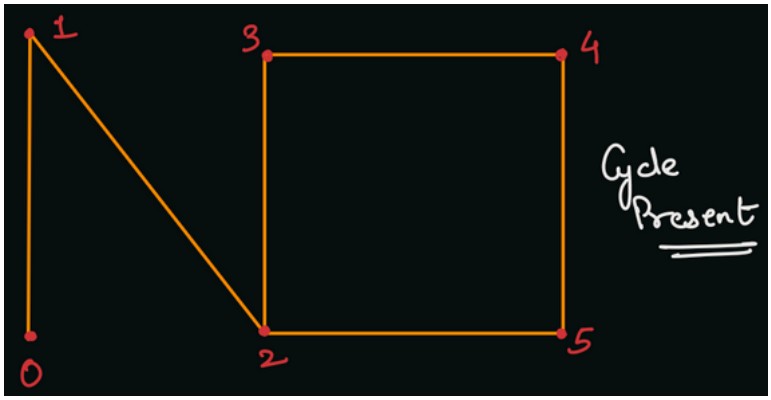
Implementation

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

4. Check if a given graph is cyclic or not

Examples





Approach

The solution we are going to discuss in this article uses Breadth **First Search (BFS)** Algorithm.

What does it mean, when we say that there is a cycle present in the graph?

It means that there exists a pair of vertices (u, v) such that there is not an unique path from node u to node v. In simple words, there exists at least two different paths of reaching node v from the node u.

To find if there exists such two paths, we can do a BFS traversal of the given graph. For every visited vertex "v", we will push all the neighbouring vertices "u" into the queue. Now, here is the **crux** of the algorithm. If the neighboring vertex is already visited, then there exists at least two different paths from node v to node u. It is because we were able to visit the vertex u, as it must have been popped out from the queue earlier. Now, since there exists two different paths from node v to node u, there is a cycle present in the graph, (and nodes u and v must be part of that cycle).

Q) Can you figure out a problem here? (HINT: Graph is undirected, i.e. edges are bidirectional) ?

Since the graph is undirected, it means there will be edge u to v, as well as edge v to u. Hence, node u will be present in the adjacency list of node v and similarly vice-versa.We are given in the problem statement that the graph may or may not be connected. Hence, we will have to check whether there exists a cycle in any component of the graph.

To check cycles in all components, we will start BFS from each unvisited node in [0, n-1], and report true if there exists a cycle in any component. Once we will find the neighbours of node v, then we will get node u as a neighbour which is already visited. We will report that there is a cycle present, but the bidirectional edge does not mean a cycle. Thus, we should also maintain the information of the parent node, and not consider the parent node as the neighbour of the child node. We use a parent array to keep track of the parent vertex for a vertex so that we do not consider the visited parent as a cycle.

Q) Will the algorithm work if we start the BFS from any one node, let us suppose node 0?

We are given in the problem statement that the graph may or may not be connected. Hence, we will have to check whether there exists a cycle in any component of the graph. To check cycles in all components, we will start BFS from each unvisited node in [0, n-1], and report true if there exists a cycle in any component.

Note: If you get this problem in a face-to-face interview, then you must ask about the corner cases like:

Q)Can there be edges of type (u, u), i.e. does self loops exist in the graph ?

Q) Can there be multiple parallel edges, i.e. does there exist more than 1 edge (u, v) in the graph ?

For simplicity, we can assume that there are no self loops and parallel edges in the current problem.

Implementation

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