More Tricks with DFS and BFS

Hello, and welcome to competitive programming. Today we'll cover a few more things you can do using BFS and DFS that will turn out to be helpful.

Objectives

You will see algorithms that will help you check if a graph is bipartite, find cut edges and cut points, find cycles, and find strongly connected components.

Bipartite Checking

A graph is bipartite if it has two subgraphs such that every single edge connects a node from one subgraph to the other. Another way to say this is to say that the graph is 2-colorable; every vertex has neighbors of the opposite color, and no two neighbors share the same color.

Once you have the code for a traversal, it's easy to modify it to check for bipartiteness. Use two colors, 0 and 1, and start the root with color 1. Then, for each node, set the neighbors to the opposite color. If a neighbor already has a color set, it should be the opposite of the current node, or else you have a non-bipartite graph.

Here's an example. Let's start with node A and color it.

(next)

We then check all it's neighbors and color them the opposite color.

(next)

This colors d,f, and g. Suppose we are using BFS here, so let's take our next node as f. From f, we check it's children and color b.

(next)

Next we visit d, which colors c, and so forth.

If there were an edge between g and f, you can see that the algorithm would detect that the graph is not bipartite.

Checking for Cycles

So far we have had two states in our graphs: visited and unvisited. We can add a third state, explored, to indicate that we have seen the node, but have not yet returned back from visiting its children.

When you see a node for the first time, you change its state from unvisted to explored. Next you check the neighbors. If the neighbor is unvisited, it means the current node is the parent. If the neighbor is already visited, then you have either a forward edge or a cross edge. If the neighbor is marked explored, the edge goes backwards and it means you have a cycle.

Once you are done with all the neighbors, you set the current node state to visited.

This relies on recursion, so you will need to use DFS for this.

Finding Cut Nodes and Edges

Consider this graph.

I have annotated each of the nodes with two numbers. The superscript number is called the dfs number; it tells you the order in which the DFS algorithm visited the node. The subscript number is called `dfs_low`, it contains the lowest dfs number reachable during the DFS itself.

When we enter a node for the first time, we set `dfs_num` to be the next number up, and `dfs_low` to be the same. As we traverse, we may end up updating the entry for `dfs_low`.

In this graph, we start with A, then B, then C. From C we will eventually take the edge back to A, and so A, B, and C will have `dfs_low` of 0.

When we go to D, we would set both `dfs_num` and `dfs_low` to 3. As we continue traversing, we take the edge from e to d, and that sets `def_low` of those three nodes to 3.

You may want to pause this for a bit to be sure you see where all these numbers came from, and then resume when you are ready.

There are three things you can do with this.

The first is that you can see what nodes belong to a cycle. They will all have the same `dfs_low` number. Here, `a`, `b`, and `c` belong to a cycle, as well as `d`, `e`, and `f`.

The second is that you can identify cut nodes, also known as articulation nodes. An articulation node is one where, if you delete it, then the graph becomes disconnected.

There are two such nodes here; c and d.

Here's how you can tell by looking at `dfs_num` and `dfs_low`: if you have a vertex `u` and a neighbor `v`, and `dfs_low` of `v` is greater than or equal to `dfs_num` of `u`, then `u` is an articulation node. So, for instance, the `dfs_low` of `d` is 3, which is greater than the `dfs_num` of `c`. This means the only way to reach `d` is through `c`, If there were another path to `d`, then `d`'s `dfs_low` would be smaller.

Similarly, the `dfs_low` of `e` and `f` are 3, which is equal to the `dfs_num` of `d`, so `d` is an articulation point.

The third thing you can do is identify cut edges. It is similar, but the equation is now a strict inequality: if `dfs_low[v]` is strictly greater than `dfs_num[u]` then the edge

`u-v` is a bridge. There is only one instance of that in this graph, the edge `c-d`.

Strongly connected components.

Finally, if the graph is directed, we can use the same technique to determine strongly connected components. When `dfs_low` is equal to `dfs_num` for a node, then that node is the root of a strongly connected component. All the other members of that component will share the same `dfs_low`.

Well, that's a lot for one video. My advice is to make up a graph or two where you know what the properties are, and then try these algorithms on them by hand. The code for this is fairly simple, once you have a working DFS; you just update the `dfs_num` and `dfs_low` arrays as you go.+

Next
Transcript — Greedy Algorithms →