Graph Traversal

Hello, welcome to competitive programming!

Today we will cover Depth First Search and Breadth First Search.

Objectives

You will be able to implement both of these algorithms and show how to use them to solve some classic graph algorithms like connected components and …

DFS and BFS

DFS Basics

DFS is one of the first search algorithms computer science students learn for graphs and trees. It is very simple. Starting with the root, you mark the current node as visited, and then recursively visit any unvisited children.

It takes up memory proportional to the depth of the tree or graph you are processing. This usually gives good performance, but if you are dealing with a infinitely deep tree, such as a virtual representation of a decision tree where nodes are calculated and not stored, this can cause trouble.

Recursive DFS Code (from the book)

Here is a recursive implementation. For most of the problems you will solve, an adjacency list is the best implementation. There are exceptions, such as if the graph is extremely dense, and hopefully you will recognize that case from the problem statement.

One of the customs in competitive programming is to create these crazy typedefs for integer pairs, vectors of integer pairs, and integer vectors.

This particular representation uses a pair to represent an edge; the first part gives us the neighbor the edge connects to, the second part is the weight.

We also set two constants, visited equal to 1, and unvisited equal to -1. The `dfs_num` vector will keep track of the nodes we have visited or not, and the adjacency list is kept in `AdjList`. The only other trick here is that we have to cast `size` to an integer since size returns a different type; basically an unsigned quantity.

Notice how this code is similar to a dynamic programming algorithm, where we are only concerned about whether or not we visited a node.

BFS Basics

BFS is the dual of DFS. Whereas DFS prefers to go deep, BFS prefers to go wide.

BFS is interesting for a few reasons. First, if there are more that one node that fulfills the search criterea, BFS will find the one closest to the root of the search. Second, BFS can handle virtual trees of infinite depth without trouble, assuming they aren't too wide.

The tradeoff is that BFS takes a lot more memory – potentially – than DFS. If you consider the effect of BFS on a binary tree you can see what happens; toward the end of the algorithm all the leaf nodes will be in the queue. Given that half of a binary tree's nodes are leaves, you will need to plan on the queue possibly growing to be pretty large. Usually this is not a problem though.

BFS Implementation, also from the book

To implement this, we initialize a vector `d` of size (V) with an initial value `INF`. We can just pick some huge integer constant to represent infinity. The infinities are how we are going to keep track of whether we have visited a node yet.

After we push the source node into the queue, we can begin.

BFS tends to be iterative instead of recursive; it's not that hard to write it recursively, but the recursion depth may be a problem for a large graph. Anyway, as long as there are elements in the queue, we pop them out, and enqueue any children that haven't already been visited, after we set their distances.

Solving Problems

Connected Components

There are many kinds of problems you can solve using these traversals. One of them is to find all the groups of connected components in an undirected graph.

All you have to do is loop though your node vector. If a nodes is marked as unvisited, you just DFS it, and increment your count of connected components.

Flood Fill

Another use for DFS or BFS is called flood fill. The use here is to give each connected component of a 2d space a color. The `dr` and `dc` vectors are interesting; they give us the directions on a graph for the eight canonical directions. You'll see where we recurse that the code is much cleaner than if we had to make eight recursive calls explicitly, or else nest some for loops.

Variables `r` and `c` are the coordinates, `c1` is the color of the component that interests us, and `c2` is the new color we will assign it. As a bonus, we are also able to find the size of the component.

Topological Sort

Topological sort is another classic example. The idea is we want to get a list of nodes that we can visit in such a way that we always visit a nodes parents before the nodes children. If you think about it; it's just a DFS in which you write down the nodes in the order that you could visit them.

The normal DFS code can do this with a simple modification: after we have called the children, we push the current node onto a vector. When you are done, the vector will have a topological sort, but note that it will be in reverse order.

Calling It

Inside of your main function, you will start by clearing out the `ts` vector and setting your dfs vector to `UNVISITED`. Afterwards, the code is similar to the connected components code. The reason you need to do this is if the graph you have is not completely connected. Note that we assume the graph has no cycles.

Next
Transcript — Graphs 3 →