Solution Set

CIS 121—Data Structures and Algorithms—Fall 2020

BFS & DFS—Monday, October 12/Tuesday, October, 13

Readings

- Lecture Notes Chapter 16: Graph Representations & BFS

Problems

Problem 1. Design an algorithm to determine whether or not an undirected graph has a cycle.

Solution

We can perform a BFS or DFS and just keep track of which elements have been seen. For example, we can run a DFS and store vertices we have seen in a set and just track whether or not any previously seen node is encountered again by checking if it is in the set. Since we are simply doing a BFS or DFS, this algorithm runs in linear time.

Problem 2. Design an algorithm to determine whether or not a *connected* graph has a cycle in $O(n)$ time.

Solution

Perform the same algorithm as problem 1. However, terminate early if you explore at least n edges. Recall that a tree has exactly $n-1$ edges. An additional edge would signify that two nodes are connected by two independent paths. Thus, there is a cycle in the graph. This algorithm will take $O(n)$ time to check each vertex and $O(n)$ to check each edge (since we are checking at most n edges). Thus, the running time is $O(n)$. Note that a graph with n edges must have a cycle regardless of whether it's connected. This example is just simple to prove with BFS.

Problem 3. Design an algorithm to find the shortest path between nodes u and v in a connected, unweighted graph.

Solution

Since the graph is unweighted, we can just run BFS starting from u and for each node x that we visit, we just keep a pointer to its parent node (the node we visited x from). When we reach v, we stop and find the shortest path by backtracking through the pointers that we kept (i.e. we could see that v's parent was d, d's parent was c, and c's parent was u, so our path would be $u \to c \to d \to v$). We just do a BFS and backtrack no more than $O(n)$ times (the longest path in a graph is $n-1$ edges), so this algorithm also runs in linear time.

Additional Problems

Problem 4. Recursively generate all the permutations of the character sequence 'ABCD'.

Solution

The key to understanding how we can generate all permutations of a given string is to imagine the string (which is essentially a set of characters) as a complete graph where the nodes are the characters of the string. This basically reduces the permutations generating problem into a graph traversal problem: given a complete graph, visit all nodes of the graph without visiting any node twice. How many different ways are there to traverse such a graph?

It turns out, each different way of traversing this graph is one permutation of the characters in the given string!

We can use DFS to traverse this graph of characters. The important thing to keep in mind is that we must not visit a node twice in any "branch" of the depth-first tree that runs down from a node at the top of the tree to the leaf which denotes the last node in the current "branch".

The code solution is on this website: http://exceptional-code.blogspot.com/2012/09/generating-all-permutations.html