CS 15 Graph Traversals

Introduction

Today's lab will focus on graph `traversal` algorithms. The question of graph traversal is - given a starting vertex `s` and a destination vertex `d`, can you report a valid path from `s` to `d`? Today, you will implement two strategies for graph traversal:

1. `breadth-first search` (`BFS`)
2. `depth-first search` (`DFS`)

Your implementations will function with a `Graph` class that we have written for you. This class stores vertices in a 1-dimensional array and edges in a 2-dimensional array. It also has useful helper functions. Remember that you are a client of the `Graph` class, so there's no need for you to do anything in that class.

BFS Review

`Breadth-first search` explores a graph in waves. That is, starting at a given vertex `s`, `BFS` finds every vertex reachable in one step from `s`, then every vertex reachable in two steps from `s`, and so on until all vertices in the graph have been visited. Recall that `BFS` **always** find a **shortest path** from `s` to `d` in an unweighted graph (today's graphs will be unweighted). There can be several shortest paths, but we'll only have to find one in this lab.

DFS Review

`Depth-first search` does a 'deep dive' on one path of the graph until it has no more unexplored vertices, and then 'bubbles up' until it finds a new path to explore. Consider the tree traversal algorithms, `preorder`, `inorder`,

and `postorder` traversals. Each of these three is an implementation of `DFS` in a tree.

BFS Implementation Details

Note: there are many valid ways of implementing this strategy. We've picked one for this lab. Please stick with the approach given in the starter code. If you want to explore others on your own later, that's great!

The Gist

We start at our source vertex, then we visit all of its neighbors (that is, nodes that we can be reached in one step from the source vertex). Next, we visit all of its neighbors' neighbors (i.e., nodes that we can reach in two steps), and so on. Eventually, we will have explored all the nodes and failed or we will have reached the desired destination. In the latter case, we have found a path!

Marking vertices

We want to explore each vertex only once; this ensures we find the shortest possible path from source to destination, so although there are multiple ways to get from the source to the destination, we only care about the one that can get us there in the least amount of steps. In order to prevent revisiting vertices, we will **mark** visited nodes. Initially, all vertices are unmarked. Whenever we explore a new vertex, we mark it so that we do not explore it a second time.

Auxiliary Data Structures

We'll use a `Queue` as an auxiliary data structure for `BFS`, plus an array to track the path from `source` to `destination`.

- The `Queue` will contain the nodes to be explored, as we saw in class. `s` is the first node to be enqueued. We dequeue a vertex from this queue when we explore it.
- The `predecessor array` helps us track the path from any vertex back to `source`. At any given moment, there is a "current" vertex whose neighbors we are exploring. For each of the neighbors, we denote that its predecessor is the current vertex. We use -1 to indicate that we do

not know the predecessor of a vertex (or that there is no path between `source` and that node).

Getting Started

Start by setting all vertices (except the source) as unmarked. Then initialize the predecessor array so that no vertex has a (known) predecssor - i.e., all values are set to -1. Finally, enqueue the source vertex in the `Queue`.

Core of the Algorithm

The main part of the algorithm is as we say in class: while the `Queue` is not empty, take the first element off it, and `explore` it.

Cleaning up

The BFS algorithm can end in two ways:

1. We can explore all possible paths without finding the destination, in which case, there is no path. You can't get there from here.
2. At some point in the exploration, we have found (and marked) the target destination. That means that we have a path from the target to the source and we need to reconstruct it. For that, we need to look at the predecessor array. We will walk backwards from the destination to the source.

DFS Implementation Details

`DFS` is very similar, but uses a `Stack` rather than a `Queue`. You can implement this in one of two ways:

1. Use recursion! The function call stack will work as an implicit `Stack`.
2. Use a `Stack` object.

Your Job!

For the lab, we simply ask that you implement the `bfs()` and `dfs()` functions in `PathFinder.cpp`. That's it! You're highly encouraged to check out the `Graph` class - particularly `Graph.h` - this will reveal the API you can use to mark vertices as visited, etc.. The lab code is in `/comp/15/files/lab_graph_traversals`.

Running PathFinder

Once you have finished implementing `bfs()` and `dfs()`, you can run PathFinder using `./PathFinder cityFile`. We have already provided you a cityFile named `cityData.txt` with simple flight path configurations. We recommend that you take a look at it and come up with some "to" and "from" cities that you will want to test with the PathFinder program.

Here is an example of how to run and test the program:

```
vm-hw00{kquint02}149: ./PathFinder cityData.txt
Command? bfs
City 1? LA
City 2? SanFran
LA ---> SanFran through Jet Blue.
end of flight
Command? dfs
City 1? Boston
City 2? Houston
Boston ---> Chicago through United. Chicago ---> Houston through
    American Airlines.
end of flight
Command? quit
Thanks for finding paths and such
```

Submitting

You will need to submit the following files:

```
PathFinder.cpp
PathFinder.h
City.cpp
City.h
Graph.cpp
Graph.h
Makefile
README
```

You must submit them using Gradescope to the assignment `lab_graph_traversals`. Submit these files directly, do not try to submit them in a folder.

Note

You may have noticed that we don't need the marking array. We can just the Queue for the nodes, and we can deduce that a node is marked by the

presence of a predecessor (except for the source). We encourage you to use the structures we set up, however, to keep the code simpler.