Lab 2

Arta Seyedian

Program

In this program, an adjacency matrix made up of binary values must be processed as a graph and evaluated in terms of each vertex's relationship with itself and the other vertices in the graph. The data structure used in this program were a three-dimensional array which are read from and store the vales of a matrix in the input file, and whose dimensions match the dimensions of the header number above each matrix in the input file. The algorithm used to traverse the graph and determine the paths was a depth-first approach, which starts at the root node and goes as far as possible along each branch before backtracking and exploring other possible avenues. The three-dimensional array is an array of matrices; an input file with 3 different matrices of varying dimensions would all be stored in this array and expanded using additional brackets. This allows for all the matrices to be read continuously and potentially accessed as a database of matrices.

I chose a depth-first search because it is easier to implement with a recursive method. In detail, a depth-first search starts at the root node, which is row 0 of a matrix, and evaluates each integer in that row for a 1 or a 0. If it is a 1, then the index number of that row sub-array which corresponds to the row which serves as the adjacent vertex in question is passed to the recursive call, and the process is repeated. Once a branch has been exhausted, all that needs to happen to return to the previous node is for the recursive call to break.

One design choice that I had to make that may appear redundant but is essential to the proper functioning of my program is in the deepSearch method in the graph class. After the recursive call, which is nested in an if-statement, there is another if statement which evaluates if q, which is the node being examined for possible connection, that is true if q is equal to the size of vertBank -1 , which would be the zero-indexed equivalent of the size of the array of vertices. If true, the stack then deletes the most recent item in the path, which would be the node whose recursive call was just exited. This happens because once a node has exhausted and printed all its possible paths, it must be removed from the path when backtracking to ensure accuracy of the subsequently printed paths. There is another such else-if clause which essentially the same thing does if there was no positive hit for a connection between the node and the possible adjacent node.

The complexity of the depth-first search is O(n + m), where n is the number of nodes and m is the number of edges between the nodes. This variable can range anywhere from O(1) if there was just one connection per node and $O(n^2)$ if each node connected to every other node and itself.

Enhancements include making the array three-dimensional and printing out a message when the path has already been visited in addition to printing out when a path does not exist. I also included documentation. There may be other enhancements that I might have overlooked.

Project-Specific Discussion

An iterative method would have included at least one more for-loop that would have simply done the work of the recursive method. Recursion worked naturally with the structure of the tree, as it was easy to imagine “diving” through each branch and backtracking to test out other paths. An iterative loop would have remained within the same stack frame, whereas each successive would be a new stack frame that would be popped upon return. Recursive calls therefore consume more memory because of its use of additional stack frames. Using an iterative approach would not have significantly altered the overall design of the data structure used; however, the resulting product might have appeared less elegant and/or been more resource consuming.