Brian Loughran
Recursive Graph Search Analysis Document
Johns Hopkins Unviersity
Data Structures

General Design

The major data structure used in this program is the Graph object, which contains a two dimensional Boolean array and an integer size of the array. The Boolean array dictates the paths in the graph, where the value True designates a path from the y coordinate of the array to the x coordinate of the array. If the value is false, there is no path from y to x. This array is mapped from a binary representation (where 1 indicates a path and 0 indicates no path), with a custom method, addAdjListAtNode() which reads input from a text file line by line. There are also various helper methods, such as getGraphString() which converts a valid graph object into a nicely formatted string, which can be outputted to a text file.

Another major data structure is the PathsString() object. The PathsString() object is used to keep track of the valid paths in the graph. The object contains just a single String variable, initialized as "". The object contains several helper methods, such as append(), which adds a path, getNumPaths() which returns the number of paths, clear() which clears current paths, and getPaths() which returns the paths, or returns "No Paths Found" if there are no paths. The class methods also take care of formatting for the string as new paths are added and removed.

Information is read from the input file in the main() method. This input is read in line-by-line, and any input that is improperly formatted is ignored by the program. The output of the program is written to an output.txt file. The first thing that is written to the output file is any input that is not properly formatted to inform the user that those lines will be ignored by the search algorithm. Naturally, the next thing that is printed to the output file is the graphs that were actually recognized and will be included in the search. Once all the information is read into the program, all paths are found and sent to the output file. Some important enhancements include error handling if the graph is not all 1's and 0's, displaying the number of paths from one node to the next (helpful if there are a lot and you do not want to count), and displaying the number of valid graphs (again helpful if there are many).

The search algorithm used is a recursive depth first search algorithm. Depth first search starts at a root node and explores as far as possible along any path before backtracking. We use an array to keep track of nodes that have been visited, as to avoid endless cycles (with the only exception being if the start and end nodes are the same). The worst case runtime for this algorithm is $O(V + E)$ where V is the number of vertices, and E is the number of edges.

Alternative Approaches

One alternative approach considered was to create a Node() object, then create an adjacency list for each node. A graph, then, would consist of a list of Node() objects. It would be nice to have an ArrayList to keep track of the connected nodes, as it would make iterating over each easy and the .append() and .remove() functions would be very helpful with adding and removing connections on the Graph(). But since ArrayLists are strongly discouraged, the adjacency list would have to be a Boolean list. Arrays are still appropriate for storing the adjacency list, since an array with the size of the number of nodes can fully and obviously represent all connections for a particular node. At this stage, I came to the conclusion that having a two dimensional list is exactly analogous to a list of objects that each had a list, so I decided that the two dimensional list of adjacencies was simpler, and required less objects to create than representing the Graph() as a list of Node() objects.

Learning and Looking Back

From this project I learned a lot about recursion and setting up the details of its implementation. This is an interesting problem, because the end case is rather complex. If we want to find just one path, we can end simply when we reach the destination. However, if we want to find all the paths, we can end only when all paths have been searched. We also have to keep track of the solution through multiple functional runs (this was taken care of in the PathsString() object). This is a complex recursive problem, but one that is naturally suited to recursion as a solution.

One thing I may do differently would be to employ an iterative approach to the depth first search algorithm. Anything that is written recursively can be written iteratively, however if you are using an iterative approach it may be more beneficial to create a Node() object and represent the Graph() as a list of Node() objects. If you kept track of the current path, you could do a while(not all paths searched) loop to search all possible paths. Then you could get connections by doing a for() loop over all adjacent nodes from the current node adjacency list. You could avoid loops by searching the current path array for the adjacent node. You can keep track of all possible paths in a similar way as with the recursive implementation. This would give you all you need to run the algorithm. This could potentially have an easier implementation than the recursive implementation.