Brian Loughran
Linked 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 one dimensional linked list array and an integer size of the array. The linked list array dictates the paths in the graph, where the linked list contains all of the neighbors of the node index of the array. This linked list 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.

The linked list is implemented in the Neighbors() and the LinkedNode() classes. LinkedNode() contains the integer value of the neighbor node, and a pointer to the next LinkedNode() in the linked list. If there is no next node, the next variable is set to null. The Neighbors() class does most of the management of the LinkedNode() class. The only variable stored in Neighbors() is head, which stores the head of the list, however it has several good helper methods such as add() which appends a node to the end of the linked list, toString() which converts the linked list to a string which can be easily output to a file, and toBinaryArray() and toBinaryString() which work together to convert the linked list to a string which mirrors the input and can be easily output to a file.

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), displaying the number of valid graphs (again helpful if there are many, and displaying both the binary implementation (mirrors the input) and the linked implementation of the graph. Solutions were identical using linked and array implementations.

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

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 to be considered is how I dealt with the node numbers and the index of the Graph() nodes list. These values, which represent the same thing, were always offset by one in my program, which meant that conversions were happening between these values all the time, and it was difficult to keep track of. For example, the Graph() contains a list of linked lists. So the neighbors of node 1 are stored at index 0 of the graph, the neighbors of node 2 are stored at index 2, and so on. The conversions between these two values were always done on the fly, making it hard to track. It may have been simpler to line these values up. One option would be to change the node numbering system (the first node could be index 0). Another option would be to add an unconnected node at index 0 (which would take up almost no memory per graph), that way the neighbors of node 1 are stored at index 1 of the Graph() object. Either solution would have worked to line up the indices, which would make the code more readable.

Learning and Looking Back

From this project I learned a lot about linked lists and setting up the details of its implementation. For this particular implementation, there was the added wrinkle of creating a list of linked lists, which added some complexity. It was important to make sure you were looking at the right index in the Graph() object, otherwise you would be looking at entire incorrect Neighbor() lists. It is also important to keep track of exactly where you are in the Graph() and where you have been, otherwise you can get lost or get stuck in endless loops. Using a linked list allows you to dynamically allocate memory for the adjacency list, which can be beneficial if you have a rather large, sparse graph. In this case it would require much less memory than an array implementation.

One thing I may do differently next time is to do a little cleanup of my most important method, getAllPaths(). This is the recursive method which searches the graph and sends to output all the paths in the graph. But some code that is contained in this method can be done in helper methods. For example, when you are appending a path to the current path, there is an annoying while loop and iterater variable which manages placing the int value of the valid graph neighbor on the current path. This could easily be done in an aptly named helper function, and would reduce the perceived complexity of the most important function. This would make the code far

more readable, and get that annoying code out of the equation without negatively effecting the performance of the code.