Mike Ogrysko – Lab 2 - Analysis

Data Structures: Description and Justification

For this lab, I used a 2d array to store each adjacency matrix because it was required per the assignment's instructions. Other than the fact that use of an array was a requirement, use of the array was appropriate because the matrix size was provided, and each matrix has the same number of rows and columns. I was able to read the file to get the array size, initialize the array, and then iteratively loop through to store the matrix in the array easily.

The application recursively builds a path and stores it in a singly linked list implementation of a stack. In this implementation, I created *push*, *pop*, *isEmpty*, *peek*, *size*, *display*, and *printFile* methods. I needed to be able to add and remove nodes to and from the stack, so these methods assisted with manipulating and displaying the stack while the recursive method, *findPaths*, runs. The LIFO characteristic of the stack assisted with the push/pop that needed to occur in *findPaths*. I was able to reverse the path stored in the stack in the *display* and *printFile* methods. Also, the stack needed to grow depending on the size of the path, so a linked application seemed to be appropriate as it could grow as necessary and would not need to be initialized with a size, which would be unknown, each time.

I used a Boolean array, *visitArray*, to track whether or not a *to* node had been visited while finding the path. This was one of the stopping cases in the recursive method – if the node had not been visited and the node has a value equal to 1, then add the node to the stack and run the recursive method.

I created another Boolean array, *resultsArray*, to track whether or not a path was found for a particular set of start and ending nodes. If a path was found for a current node, true is passed to the array. From there, this array is read by method *noPathMessage*. If there are no true values in the array, "No Path Found" is displayed for a particular set of nodes. The application ultimately runs a nested for loop to loop through the 2d array and run the method *displayPaths* for each *to* and *from* node in the matrix. After *displayPaths* runs, *noPathMessage* runs by reading the current node's *resultsArray* and determining if paths were found. If no paths were found, *noPathMessage* prints "No Path Found". I felt this was necessary in the event that no paths were found for a set of nodes.

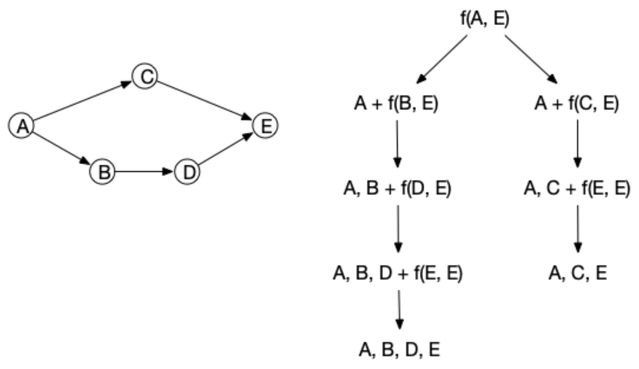
All-in-all, I thought that my use of the data structures described and justified above were appropriate.

Design: Description and Justification

I broke the application development into two phases:

1. Reading the matrix and ingesting into an array
2. Development of a recursive function to process the paths

I had a hard time visualizing the recursive function from the start. I think the best explanation I found for this type of problem talked about the idea of recursion abstractly. Start with two nodes – start and end – and run them through the function. The starting node is added to the path and the connecting node becomes the new starting node that is run through the function with the ending node. This continues until the starting node and ending node are the same or until all paths are exhausted. I have tried to illustrate this in the image below:



Ultimately, I created the *displayPaths* function, which initializes a Boolean variable *firstPass* and sets it to true. In the method, I created an initial stopping case: if the starting node does not have a connecting node (the starting node's row is empty), then print out the message "No Path Found." Assuming the stopping case does not apply, the *findPaths* method is called. This method includes a stopping case: if the current node is equal to the ending node and *firstPass* is equal to false (meaning that it is not the initial time through the method), print out the path. Assuming the stopping case is not reached, indicate that the current node has been visited (using *visitArray[currentNode]*). Then, set *firstPass* equal to true. Next, iterate through the node's row. If the value is 1 and the node has not been visited, push the node to the path (*localPathStack*) and run *findPaths* again with the node as the new current node. This continues until the current node equals the ending node or the iteration is exhausted. The path is printed out to the screen and to the file. The *displayPaths* method continues until all paths are found for the start and end nodes. For the message that is printed when no paths are found, I created another function, *noPathMessage*, to read *resultsArray* and determine if paths were found. If no paths were found, *noPathMessage* prints "No Path Found".

I created the *displayHeaders*, *arrayIteration,* and *processAdjMatrix* methods to break up main(). The *displayHeaders* method just contains the headers to be printed at the top of the screen/file. The processAdjMatrix method reads the file, creates a single array, removes the loops, and calls the *arrayIteration* method. The *arrayIteration* calls *displayPaths* and *noPathMessage* for each set of nodes in the array.

This was the best way that I could determine to create a recursive solution. As I mentioned before, I struggled with the idea of visualizing the recursion. I imagine an iterative solution requiring repetition of the similar code. It seems like I would need to identify all of the *from* nodes for each *to* node and then run a series of conditional statements to connect them. Perhaps a while statement could be used to keep an iterative method running until the end node is reached.

Efficiency: Time and Space

Looking at the algorithms in my program, I believe time complexity would look like this:

Method	**Time Complexity**
displayHeaders()	O(1)
processAdjMatrix()	O(n)
arrayIteration()	O(n)
noPathMessage()	O(n)
rowEmpty()	O(n)
displayPaths()	O(1)
findPaths()	O(n)
pop()	O(1)
push()	O(1)
peek()	O(1)
size()	O(1)
isEmpty()	O(1)
display()	O(n)
printFile()	O(n)

The sample data I used ranged from 4-100 (per matrix) records for processing, so the O(n) methods would be impacted at a fairly low level. In terms of space complexity, I believe that all of the items are range of O(1).

Lessons Learned and Changes for Next Time

When presented with a problem, I am inclined to start coding immediately. I have to force myself to start by writing out solutions to the problem before diving in to code. I continue to learn a lot about reading and writing files. I wrote in one of my discussion prompts that I tend to think in terms of iterative solutions rather than recursively. This lab forced me to think recursively, and I think that was a good thing because it gave me practice thinking about and building recursive solutions. Thinking abstractly about the function and drawing the picture (from the Design section above) really helped understand how a recursive function could be implemented.

I mentioned this in the last lab, but if I was doing this assignment again, I would definitely read the project description earlier so that I could consider how this should be implemented over time (I did quick read-though but did not put much thought into it). I read Lab 3, and obviously there is an opportunity to use linked structures to build the graph. I used a linked implementation of a stack to build and store the path. I need to consider additional ways to use them to solve the same problem.

Test Cases

#	Input	Output	Description
1	PathsGraphInput	PathsGraphOutput	Test file provided for the assignment – expected results are shown
2	PathsGraphInputAll23456	PathsGraphOutputAll23456	All 1's for matrices sized 2, 3, 4, 5, & 6 – expected results are shown
3	PathsGraphInputBlank	PathsGraphOutputBlank	Empty file – headers print with nothing else – error message would be useful
4	PathsGraphInputNo5	PathsGraphOutputNo5	5x5 matrix with all 0's – expected result is returned – no paths found
5	PathsGraphInputNone10wLoopTest	PathsGraphOutputNone10wLoopTest	10x10 matrix with just loops set to one, all others are set to 0 – expected result shown, the loops are removed, and no paths found
6	PathsGraphInputRandom6810	PathsGraphOutputRandom6810	Randomly generated matrices sized 6, 8, and 10 – expect results shown
7	PathsGraphInputUneven	PathsGraphOutputUneven	4x2, 4x6, and 3x3 graph – returns results that are inconsistent – additional work needed to catch the uneven entries
8	PathsGraphInput13matrices	PathsGraphOutput13matrices	Contains 13 different matrices – expected results shown
9	PathsGraphInput6by6	PathsGraphOutput6by6	The same 6x6 matrix in one file six times – expected result shown each time
10	PathsGraphInputMissingSizes	PathsGraphOutputMissingSizes	The test file with sizes missing – expected results not shown – missing sizes need to be accounted for