Mike Ogrysko – Lab 3 - Analysis

Data Structures: Description and Justification

For this lab, I created a Node object to store each value in the matrix. Each node had a pointer for the next item (*next*) in the row and the next item (*below*) in the column. I also stored each Node's row and column (*row* and *col*, respectively). I created a class called *LinkedList*. In this class, I created a method called *buildGraph()*, which took the *Scanner input* from the file and the matrix size (*adjMatrixSize*). In *buildGraph()*, I read each value from the matrix, stored in a Node object, and linked the items that formed each row. Then, I iterated through and linked the items in each column. I created several list methods to assist with the goals of the lab:

nodeCount() – returns the number of Nodes in the list
printLinkedList() – displays and prints the list as a matrix
removeLoops() – finds loops in the matrix and replaces 1's with 0's prior to finding paths
checkRowEmpty() – looks for all 0's in a single row in a matrix; assists with finding paths
setCurrentRow() – searches for an returns the current node; works with *checkRowEmpty()*

I felt that these choices would allow me to use many of the same functions created previously to find node paths (with small changes). I feel that this was a correct assumption. Much like last lab, I was able to read the file to get the matrix size and then iteratively loop through to store the matrix in my linked list (using *buildGraph()*). Previously, I mostly did the same thing with the array. Here is a comparison of the methods used to represent the matrix and find and display the paths:

Lab 2 Method	**Lab 3 Method**	**Changes**
displayHeaders()	displayHeaders()	No Change
processAdjMatrix()	processAdjMatrix()	Removed the iteration to read and represent the graph; replaced with methods to read and create list and print list; Replaced arrayIteration() with listIteration()
arrayIteration()	listIteration()	Changed the name
noPathMessage()	noPathMessage()	No change
displayPaths()	displayPaths()	Changed to take linked list instead of array
findPaths()	findPaths()	Changed to take and call linked list; Added currentRow node, which calls setCurrentRow(); changed if statement to check against currentRow instead of position in array; iterate through rows if the list instead of the array

In addition, the application still recursively builds a path and stores it in a singly linked list implementation of a stack. From the previous 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 continued to use 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.

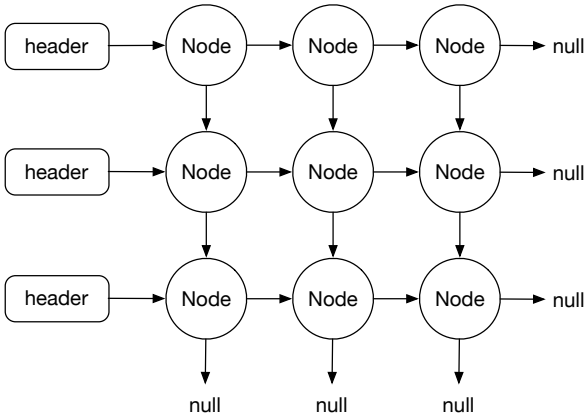
I also continued to use Boolean array, *resultsArray*, to track whether or not a path was found for a particular set of start and ending nodes. If a path was found for a current node, true is passed to the array. From there, this array is read by method *noPathMessage.* If there are no true values in 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. I prefer the array implementation because I feel that it aligns better with the matrix that we are attempting to represent and process. Because I was able to reuse/repurpose many of my ideas and methods from Lab 2, I do not think that the implementation change made a huge difference. With the linked implementation, I did like the idea of creating the Node with multiple pointers and storing whatever data I wanted into the Node. I found this very easy to work with.

Design: Description and Justification

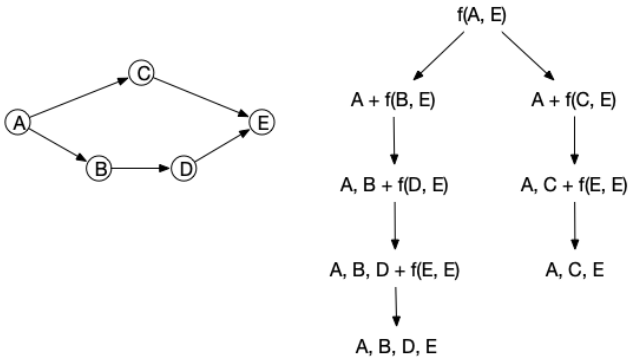
I broke the application development into two phases:
1. Creation of the linked list structure
2. Reading the matrix and ingesting into the linked list
3. Modification of the existing methods process the paths

For the development of the linked list structure, I created a Node class to store the data, row, and column information, as well as the pointers to the Node to the right and below. A Node is created by specifying *data*, *row*, and *col*. The *buildGraph()* method reads the matrix and stores the values as Nodes in the list. Ultimately, this structure is created:



The printLinkList() method iterates through the Nodes in the list and, using the pointers, prints out the data stored in each Node. I modeled this design off of an article I found on the subject: https://www.geeksforgeeks.org/construct-a-linked-list-from-2d-matrix-iterative-approach/.

As mentioned above, I reused/repurposed many of the same methods I created in the previous lab by making adjustments to account for the linked list instead of the array. I continued to use the same recursive method to find the paths – starting with two nodes – start and end – and running them through the function. The starting node was 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. My illustration from Lab 2 below:



Again, I used the *displayPaths* method, 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." I used a new method, *setCurrentRow()*, to find the startNode in the linked list and return the first Node in the row. 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. I adjusted this method by creating a Node called currentRow and again calling *setCurrentRow()* to establish the current node. Once I had the current row, I modified my for-if statement to run the recursive method if the *currentRow* had a column value equal to *i* and if *data* was equal to 1. I iterated to the next value in the row by using the *next* pointer – *currentRow.next*.

For the message that is printed when no paths are found, I reused *noPathMessage()* to read *resultsArray* and determine if paths were found. If no paths were found, *noPathMessage* prints "No Path Found". I also reused *displayHeaders()* to print the initial headers found in the file and onscreen. I changed the name of *arrayIteration()* to *listIteration()* and made no additional changes. In *processAdjMatrix()* I pulled out the code that iterated over the array and replaced it with methods that performed the same functions on the linked list structure. I called *listIteration()* instead of *arrayIteration()*.

I felt that the recursive solution was still the best one for this application. I think the way that I implemented the linked structure took advantage of the work I did in Lab 2 and much of it could be reused or repurposed.

Efficiency: Time and Space

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

Method	**Time Complexity**
Main()	O(n)
displayHeaders()	O(1)
processAdjMatrix()	O(n)
listIteration() (2x for)	$O(n^2)$
noPathMessage()	O(n)
rowEmpty()	O(n)
displayPaths()	O(1)
findPaths() for-if	O(n)
pop()	O(1)
push()	O(1)
peek()	O(1)
size()	O(1)
isEmpty()	O(1)
display()	O(n)
printFile()	O(n)
buildGraph() (2x for)	$O(n^2)$
nodeCount()	O(1)
printLinkedList() (2x while)	$O(n^2)$
removeLoops() (2x while)	$O(n^2)$
checkRowEmpty() (for-if)	O(n)
setCurrentRow() (2x while)	$O(n^2)$

The sample data I used ranged from 4-100 (per matrix) records for processing, so the O(n) and $O(n^2)$ 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

Others may have a different view on this for their implementations, but I felt that the linked list structure I implemented was easily interchangeable with the array implementation with some small changes. I am still working on finding the best ways to implement checks on the data ingested. I think I have made some improvements in this area, but there is still room for improvement. I ended up iterating through the file separately to do some checks before building the graphs and finding the paths. I also added some checks to my methods.

The discussion question in mod 9 really helped me to start this lab early. In that discussion, I was able to code a good amount of the solution to represent the matrix as a linked list structure. I shared it with my discussion cohort and received some great feedback and pointed questions to consider as I continued development in this lab.

Test Cases

#	Input	Output	Description
1	PathsGraphInput	PathsGraphOutput	Test file provided for the assignment – expected results are shown; accounted for the blank line at the end of the input
2	PathsGraphInputAll23456	PathsGraphOutputAll23456	All 1's for matrices sized 2, 3, 4, 5, & 6 – expected results are shown
3	PathsGraphInputBlank	PathsGraphOutputBlank	Empty file – Prints message that file is empty and processing is stopping
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 – catches the uneven graph and returns an error
8	PathsGraphInput13matrices	PathsGraphOutput13matrices	Contains 13 different matrices – expected results shown
9	PathsGraphInput10x10	PathsGraphOutput10x10	Contains 1 10x10 graph – expected results shown
10	PathsGraphInput6by6	PathsGraphOutput6by6	The same 6x6 matrix in one file six times – expected result shown each time
11	PathsGraphInputMissingSizes	PathsGraphOutputMissingSizes	Test file with sizes missing; is caught with an error message to check the file
12	PathsGraphInputNodeCountLess	PathsGrapOutputNodeCountLess	Test file with one matrix where the number of columns is less than the number of rows; generates error message and ends the program
13	PathsGraphInputNodeCountMore	PathsGrapOutputNodeCountMore	Test file with one matrix where the number of columns is more than the number of rows; generates error message and ends the program