**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #6**

Due: November 19th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab, we were asked to implement fundamental methods for the construction, modification, and display of graphs. Graphs are data structures used typically to track relationships between objects, and this is achieved by having different combinations of adjacent lists, adjacency matrices, and edge lists. The main objective of this lab is for us to familiarize ourselves with how these data structures represent the same situations with their respective benefits and deficits as determined by their design.

**Proposed Solution Design and Implementation**

**Part #1 (Implementation of Functions):**

Initially, we had to create functions to insert edges, delete edges, and display in a basic way the three types of graph. For all three, this meant simply understanding the way data is stored, and entering the data from the parameters respectively. Given that the methods of the adjacency list, or AL, were provided by Dr. Fuentes beforehand, only the matrix, or AM, and the edge list, or EL, needed implementation: Namely, inserting elements into an AM went at constant time, as the source and destination of edges could be used as row and column indices with no need for traversal; On the other hand, the EL involved linear insertion, as to maintain an EL sorted first by source, then by destination, involves not simply appending said edges, but also placing them between the proper edges already in place.

Similarly, deleting edges and displaying said graphs were again constant and linear respectively, again because said methods did not and did involve traversal of their respective graphs.

Lastly were the functions that would convert any given graph into either of the other two. Again, fairly straightforward, one simply had to create a new graph of the desired type, being sure to transfer the essential parameters of vertex number, weightedness, and directedness, then simply construct a for loop (or in the case of the AM, a nested for loop) that would insert the original graph's edges into the new graph using the previously created *insert\_edge()* method, again taking note of how to describe the parameters of the edges, be that with the accessible *source* and *dest* variables of the EL, the *i* and *j* indices of the AM, or half of each with the AL.

These conversion functions, in particular the *as\_AM()* methods of the AM and EL, could then be used to access the draw method within the AM class that was provided beforehand such that any of the constructed graphs could be more visually displayed as lines and circles.

**Part #2 (Breadth-First Search and Depth-First Search):**

Now that the classes of the graphs were fully functional, the second part of the lab challenged us to implement in all three classes the breadth-first search (BFS) and depth-first search (DFS) procedures. The model given was that of the Fox-Chicken-Grain-Person problem, where you, the person, must use a boat, capable of holding only you and one other item, to take a fox, chicken, and a bag of grain across a river, with the catch being that you can’t leave the fox and chicken alone nor the chicken and grain alone, lest the former of either pair eat the latter.

To implement graphs into such a dilemma, a 16-node graph would be created, with each node containing a 4-digit combination of zeroes and ones (thus resembling binary numbers) and each digit representing one of the four players involved on either the left (0) or right (1) side of the river. From here, the edges representing the possible transitions from one legal state to the next would have to be inserted such that a path from 0000 (where everyone is on the left) to 1111 (where everyone is on the right) could be found.

Given the phrasing, I realized too late that both the insertion of these edges and the subsequent finding of desired paths were most likely intended to be done in full by the program. Having forgotten the puzzle myself, I immediately felt more compelled to solve it myself, and then simply insert the edges manually. Were I to go back and fix this, I would say that the way to have the program create the proper edges would be to have it start at 0000 and have it check beforehand using a separate method if a given edge was a legal transition, and only inserting the edge after this was found true. Said separate method would involve checking both the source and destination nodes to see if either had a 00--/11-- or -00-/-11- pattern, as those would be illegal states. From there, the program could either attempt insertion between 0000 and every other node before moving onto 0001, or it could interrupt its insertion from one node to continue insertion from a successfully connected node.

That said, I had a difficult time as it was implementing the aforementioned searches. Truthfully, I hadn’t quite succeeded for the BFS methods for the AM or EL graphs. While the concept seemed fairly straightforward, I found I couldn’t quite make the connections between the removal of items from the frontierQueue and the tracking of a path from node 0 to node 15, namely because of the 13-8/7-2 split pathways that both start and end at 4 and 11 respectively. When the rows from the AM or the edges from the EL were popped from the queue, I succeeded in recording the order of these procedures in BFS order, but failed to distinctly choose between one or the other split pathway when saving them to a list; however, I did manage to properly record the 13-8 pathway in the AL graph. My assumption was that because the search was going from node to node, if I simply recorded in the path list the last node to be inputted, I would get a list that went from 0 to 15, and that just so happened to mean that the 7-2 pathway was excluded from record.

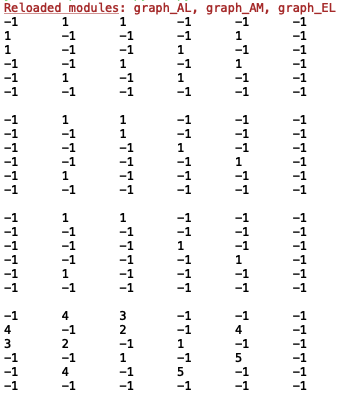
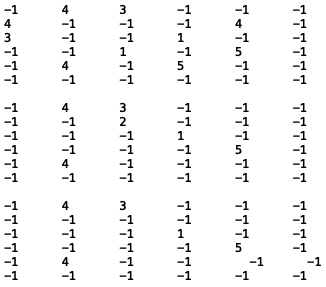
As for the DFS methods, that went by much more smoothly. Something about the stack being last-in, first-out lent the procedure better clarity for me. Here again it might have to do with the order in which the nodes, rows, and edges were being popped: Since the last of each element were the first ones to be expanded upon with each iteration, a sort of cyclical nature arose where the then furthest elements expanded into the next-nearest and next-furthest elements, their neighbors, until eventually, there was a clear-cut line from the further element, 15, to the first element, 0, and any extra pathways showed up at the end, thus allowing for easy display by simply cutting off the for loop display at element 15.

**Experimental Results**

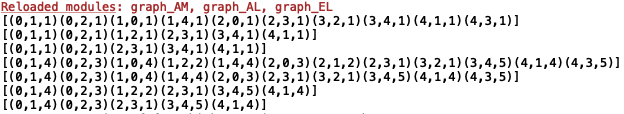
**Part #1:**

Everything worked exactly as intended: Graphs were converted easily thanks to a consistent insert method, and the delete method also properly deleted both the intended edge and its undirected counterpart. The display for the matrices in particular came out a bit wonky, but such is the nature of tabulation and negative numbers.

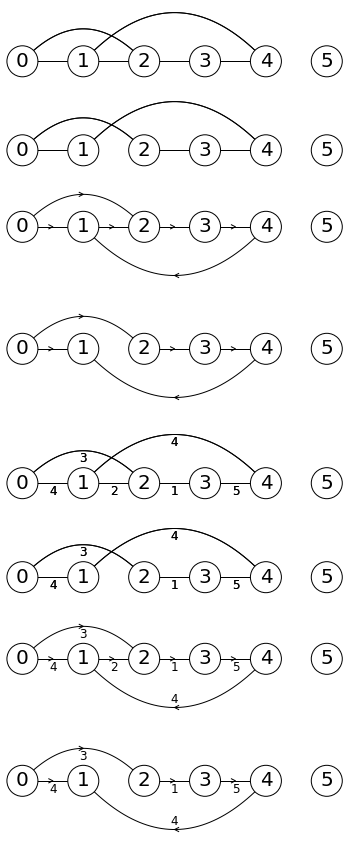
AM Graphs:

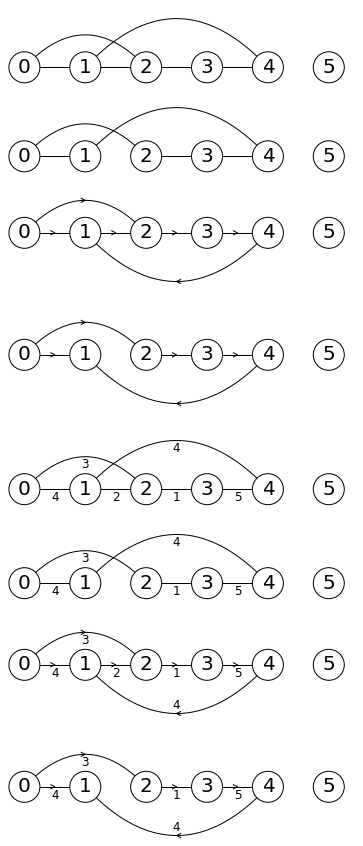
EL Graphs:



AM Graphs as AL Drawings:



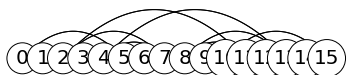
EL Graphs as AL Drawings:



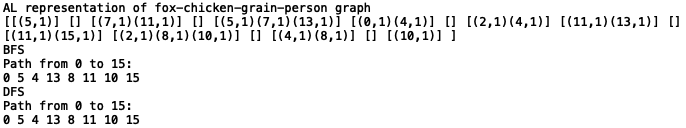
**Part #2:**

The graph for the boat problem did come out, though somewhat difficult to interpret. The corresponding representations come out correctly as well, and their BFS and DFS results came out, as I mentioned, somewhat unequally in terms of number of elements, but nevertheless correctly sorted.

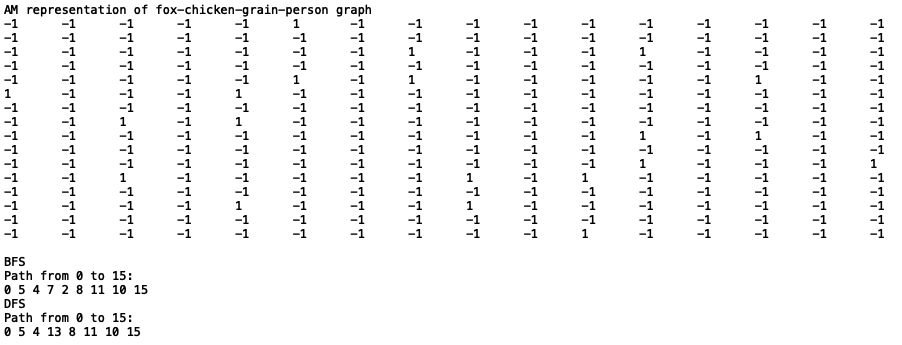
AL Drawing of Part 2:



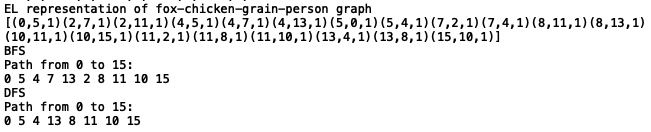
AL Representation of Part 2 (w/ corresponding BFS/DFS results):



AM Representation of Part 2 (w/ corresponding BFS/DFS results):



EL Representation of Part 2 (w/ corresponding BFS/DFS results):



**Tables:**

**Running Times**

**(E = # of edges, V = # of vertices)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **AL** | **AM** | **EL** |
| **insert\_edge()** | O(1) | O(1) | O(E) |
| **delete\_edge()** | O(E) | O(1) | O(E) |
| **display()** | O(E+V) | O(E2) | O(E) |
| **as\_AL()** | O(1) | O(E2) | O(E) |
| **as\_AM()** | O(E+V) | O(1) | O(E) |
| **as\_EL()** | O(E+V) | O(E2) | O(1) |
| **BFS** | O(E+V) | O(E+V) | O(E) |
| **DFS** | O(E+V) | O(E2) | O(E) |

**Conclusion**

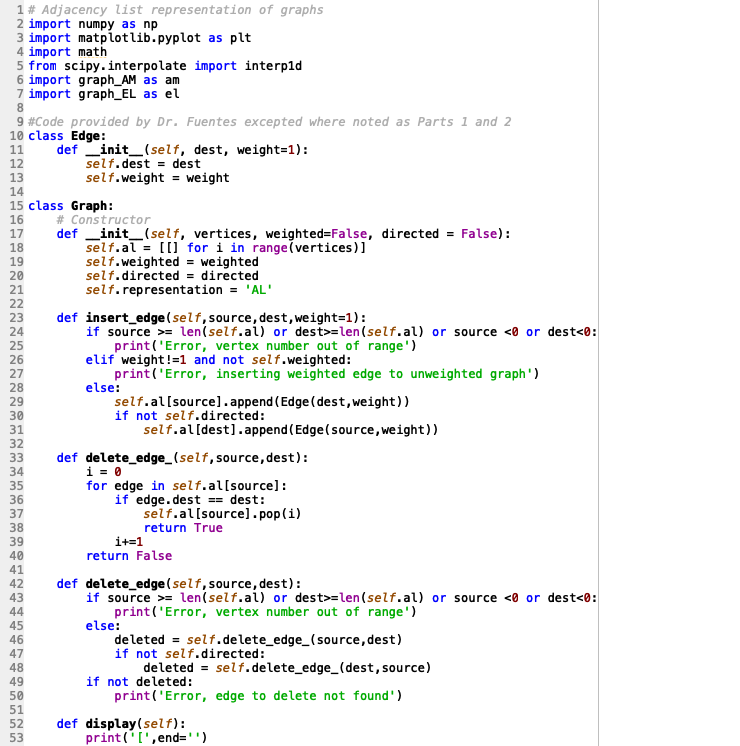
From this lab, I became more familiar with adjacency lists, adjacency matrices, and edge lists as forms of graphs that can be used to depict all possible courses of action simultaneously. I definitely learned how to translate the meanings of one such graph to another, which is always useful when so many options are available in the world of computer science.

I also had an opportunity to tackle a practical usage of these graphs with the challenge of a puzzle which, although very fun to figure out on my own, certainly showed me that I still need to better understand how to properly traverse these graphs to obtain useful information. While I understand generally the method of exploring all immediate neighbors before exploring to their neighbors that is breadth-first search versus the method of going as far out as possible and then recording any shorter extremities afterwards that is depth-first search, I must practice utilizing these skills properly.

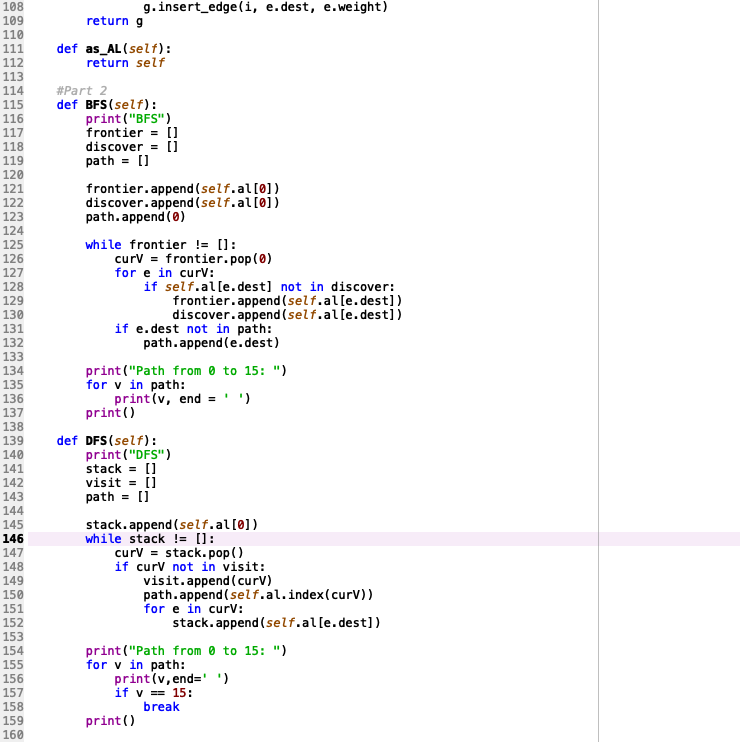
**Appendix**



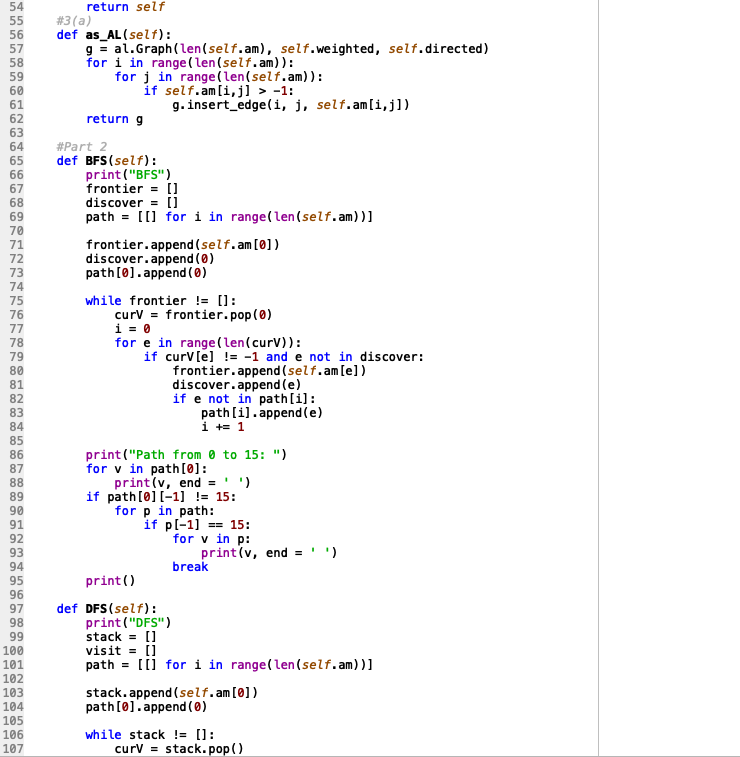


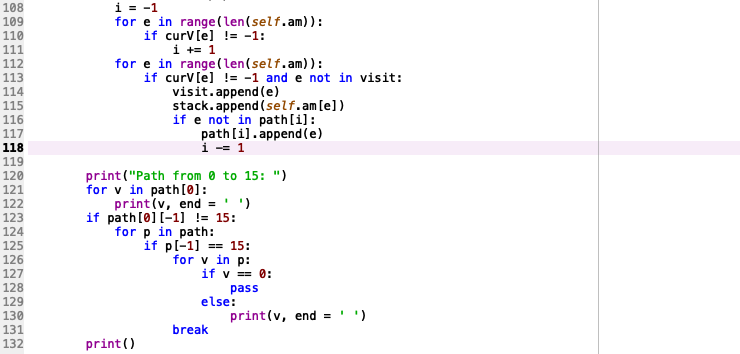


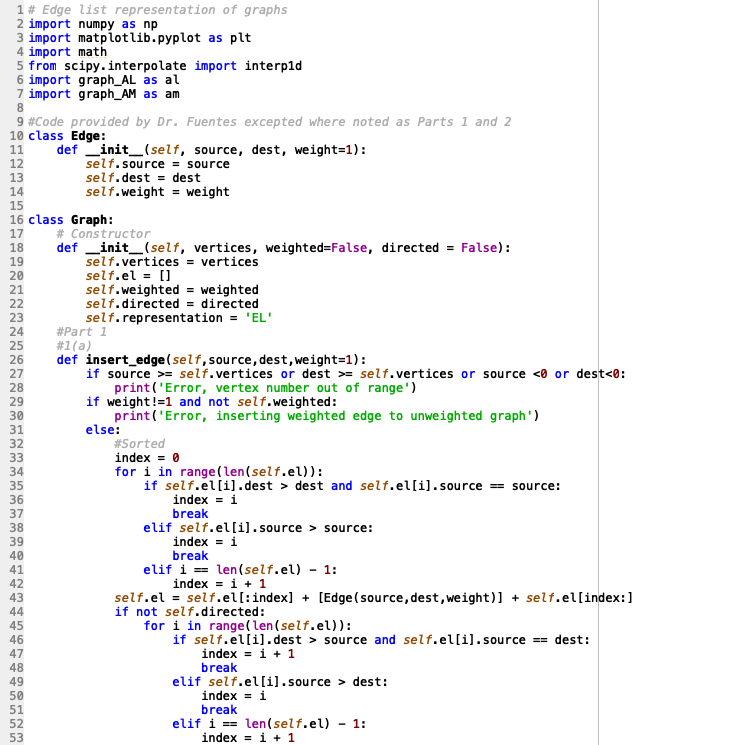


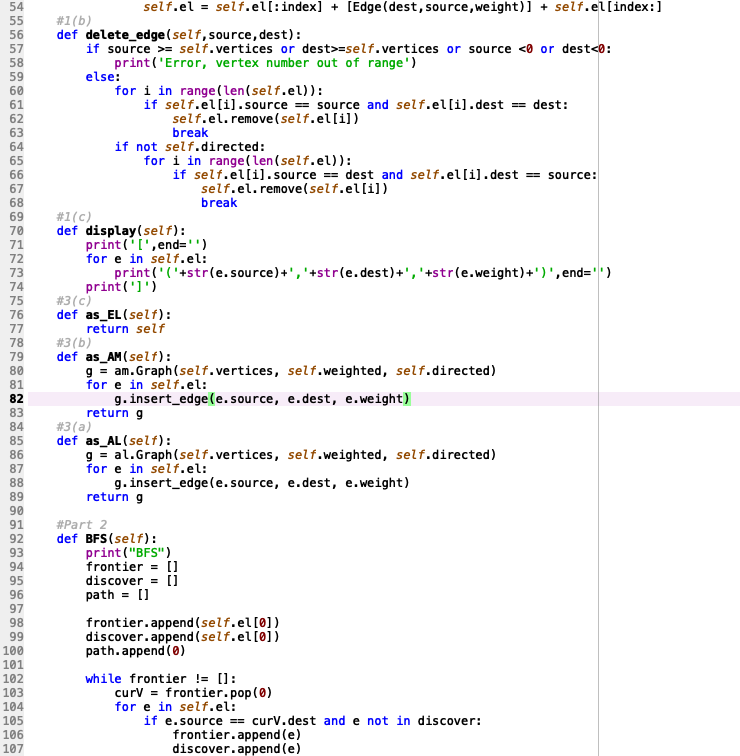














I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.