LAB REPORT #6

Sergio Ortiz
November 14, 2019

Professor – Olac Fuentes
TA – Anindita Nath
CS 2302 Data Structures – Fall 2019

Introduction

In this lab, we were first asked to implement several functions that would allow us to use the different graph implementations, such as inserting and deleting edges. The different implementations we worked with are the adjacency list, adjacency matrix, and edge list classes. The adjacency list class was mostly finished, so it was our job to use this as a base to build the other classes. Then, we were asked to create methods that would return different implementations of the current graph. For example, in the adjacency list class, we had to write functions that would return an adjacency matrix and an edge list representation. The same had to be done with the other two implementations.

Proposed Solution Design and Implementation

Part 1:

- Insert_edge (Adjacency Matrix) – My idea for this method was relatively simple, simply go to the desired location on the matrix and change the initial value of -1 to the weight of the edge. The only catch was that if the graph was undirected, then that meant that the edge pointed both ways. Therefore, I decided I would also place the weight on the mirror of the current vertex.
- Delete_edge (Adjacency Matrix) – For this method, my idea was the exact same as the previous method, but instead of inserting an actual value, I wanted to replace the current value with -1, since this was the empty value. Again, the same precaution of the undirected graph had to be taken with this method.
- Display (Adjacency Matrix) – My idea for this was basically like traversing a 2D matrix and then printing out its contents. Aside from this, I simply added some flair to make it look nice and like an actual matrix, such as advancing a line when we reached the end of a row to make it have some format in the console.
- Insert_edge (Edge List) – My original idea was simply to append the source, destination, and weight to the list. I actually proceeded with this implementation through the project, which gave me some complications later on. However, I revised this later to check if the edge was not already in the list before adding it.
- Delete_edge (Edge List) – The idea to implement this was to search the list for the element and then pop it out of the list. Perhaps a bit inefficient but I felt like it would suffice for the purposes of this lab.
- Display (Edge List) – For this method, I simply decided to print out the source, destination, and weight to simulate the format doctor Fuentes showed us on the board.
- As_AM (Adjacency List) – I decided to pretty much just create an instance of the adjacency matrix graph, traverse the list, and append the source, destination and weight. The two representations are very similar, so I figured I would not have much trouble simply extracting the necessary values off the list.

- As_EL (Adjacency List) – Here is where things got a bit more complicated. Given that I was still working with an edge list insert method that was not checking to see if the edge was already in the list before inserting it, I had to take care of that in this method. Therefore, the way that I solved this was by using a hash table with the keys representing the vertices and each key contained a list with the other vertices it was connected too. Using this, my method would check if the edge we were trying to insert was already in the graph before inserting it. Once I fixed the insert method on the edge list class, this whole process was kind of overkill, but I decided to leave it because it took a while to formulate this idea.
- As_AL (Adjacency Matrix) – At first, I was thinking of simply traversing the matrix and then inserting the element if we encountered anything besides a -1, extracting the necessary elements. However, a slight error occurred when the graph was undirected, since my algorithm would encounter the same edge twice and insert it twice. At first, I was trying to find a way to only traverse the top half of the matrix, but I found this to be too difficult to do and ended up recycling the hash table idea from before, checking if the edge was already in the graph before inserting it.
- As_EL (Adjacency Matrix) – I decided to take a bold risk and simply copy the previous method over and then initialize an edge list instead of an adjacency list. To my pleasant surprise, it worked exactly as planned.
- As_AL (Edge List) – For this method, I decided to simply traversed the list and then add the corresponding values to an instance of the adjacency list class.
- As_AM (Edge List) – I did the exact same as the previous method but with an adjacency matrix. It was surprisingly easy to go from edge lists to the other implementations, but not vice versa.

Part 2:

My idea for this part was to first create a graph in which every vertex was connected to every other vertex to which the riddle allowed it to go. For example, from state 0000, we can either go to the other side by ourselves (0001), take the fox with us (1001), take the chicken with us (0101), or take the grain with us (0011). After creating this graph, I would use either breadth first search or depth first search on the graph from vertex 0 to vertex 15. However, I made a condition where if in the current state, either the chicken and fox were together or the chicken and grain were together and the guy was not there, then we would not take that path. In this way, we would traverse the path that would solve the riddle. Also, as I traversed the graph, I was taking note of the path, by placing the previous vertex in each index of an array. Of course, vertexes that were never visited did not make this path. Using this path array, I traversed it and printed out the path.

Experimental Results

Part 1:

Here is my implementation of a method to insert an edge to an adjacency matrix. It is quite simple, we simply traverse the matrix until we find the desired position. The only catch was to insert the weight at the mirror of the matrix in case the graph was not directed. Right after we have my implementation of the method to delete an edge from the graph, which is the exact same but in this case we would replace the value currently in the index with a -1. Note that my implementation does not check if there is a value besides negative one in the index since we have to search for it anyways. I could have optimized this by keeping track of which edges we had inserting, but I decided it was not necessary for the purposes of this lab since arrays or lists in this case have constant access time. Right after, we have my implementation of displaying the contents of the matrix.

```
#insert_edge() - inserts an edge to the graph
#inputs - source: the source of the edge
#       - dest: the destination of the edge
#       - weight : the weight of the edge
#outputs - None
def insert_edge(self,source,dest,weight=1):
    self.am[source][dest] = weight
    if self.directed != True:
        self.am[dest][source] = weight
```

```
#delete_edge() - deletes an edge from the graph
#inputs - source: the source of the edge
#       - dest: the destination of the edge
#outputs - None
def delete_edge(self,source,dest):
    self.am[source][dest] = -1
    if self.directed != True:
        self.am[dest][source] = -1
```

```
#display() - prints out the contents of the matrix
#inputs - None
#outputs - None
def display(self):
    print('[',end='')
    for i in range(len(self.am)):
        print('[',end='')
        for edge in self.am[i]:
            print('('+str(edge)+')' ,end='')
        print(']',end=' ')
        print()
    print(']')
```

Now, to test these methods, I would run the test graphs class, here is a snippet of those tests.

```
g = graph1.Graph(6,weighted=True,directed = True)
g.insert_edge(0,1,4)
g.insert_edge(0,2,3)
g.insert_edge(1,2,2)
g.insert_edge(2,3,1)
g.insert_edge(3,4,5)
g.insert_edge(4,1,4)
g.delete_edge(1,2)
g.display()
```

The output of this was the following.

```
adjacency matrix:

[[(-1)(4)(3)(-1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(-1)(-1)]
[(-1)(-1)(-1)(1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(5)(-1)]
[(-1)(4)(-1)(-1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(-1)(-1)]
]
```

So, as we can see, the insert, delete, and display functions are working properly in our adjacency matrix class. Now, I proceeded to do the same methods but in the edge list class. My initial implementation of the insert method for the edge list class was simply inserting the new value to

the list. However, I modified it to match the specifications of the edge list class, by checking if the edge was not already in the list with a series of if statements. Next, the delete edge method would search for the edge and if it found it, would pop it from the list. Finally, the display method would simply traverse the list and print the desired elements.

```
#insert_edge() - inserts an edge to the graph
#inputs - source: the source of the edge
#       - dest: the destination of the edge
#       - weight : the weight of the edge
#outputs - None
def insert_edge(self,source,dest,weight=1):
    if not ((dest, source, weight) in self.el):
        if not ((source, dest, weight) in self.el):
            self.el.append((source, dest, weight))
```

```
#delete_edge() - deletes an edge from the graph
#inputs - source: the source of the edge
#       - dest: the destination of the edge
#outputs - None
def delete_edge(self,source,dest):
    for i in range(len(self.el)):
        if self.el[i][0] == source:
            if self.el[i][1] == dest:
                self.el.pop(i)
                return
```

```
#display() - prints out the contents of the matrix
#inputs - None
#outputs - None
def display(self):
    print('[',end='')
    for i in range(len(self.el)):
        #print('[',end='')
        print('('+str(self.el[i][0])+','
                +str(self.el[i][1])+','
                +str(self.el[i][2])+')',end='')
        #print(']',end=' ')
    print(']')
```

Now to test this, I used the exact same code as the adjacency matrix implementation, which outputted the following.

```
edge list:

[(0,1,4)(0,2,3)(2,3,1)(3,4,5)(4,1,4)]
```

Therefore, our implementations for the different graphs work as expected. Which is required in order to go on to the next step, writing methods to convert each of the graphs to the other ones. First, I started with the Adjacency List. To convert this into an adjacency matrix, first I initialized a graph with the same length of the list. Also, quite importantly, I initialized it with the same parameters that determined if it was weighted or directed as the current graph. This was essential, especially with the adjacency matrix since the implementation could look completely different depending on if it was weighted or not. Then after this I simply traversed the list and added the elements. The only note here is that the source of the edge is the index i, the position we are currently on in the list.

```
#as_AM() - returns an adjacency matrix representation of the graph
#inputs - None
#outputs - an adjacency matrix representation of the graph
def as_AM(self):
    g = graph1.Graph(len(self.al), self.weighted, self.directed)
    for i in range(len(self.al)):
        for j in range(len(self.al[i])):
            g.insert_edge(i, self.al[i][j].dest, self.al[i][j].weight)
    return g
```

Also, quickly note that graph1 is the adjacency matrix class imported over. Now, this method would output the following.

```
adjacency matrix

[[(-1)(4)(3)(-1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(-1)(-1)]
[(-1)(-1)(-1)(1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(5)(-1)]
[(-1)(4)(-1)(-1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(-1)(-1)]
]

adjacency list to adjacency matrix

[[(-1)(4)(3)(-1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(-1)(-1)]
[(-1)(-1)(-1)(1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(5)(-1)]
[(-1)(4)(-1)(-1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(-1)(-1)]
]
```

The top matrix is a direct call to the adjacency matrix class from which we built the matrix we expected. Then, the bottom matrix is what the method that converts the adjacency list to a matrix outputted. As we can see, they are identical. Next, I worked on the method to convert the adjacency list to an edge list. Since I was working with an edge list insert method that was simply inserting instead of first checking if the edge was already in the list, I had to take care of that here. The way I did this was by building a hash map in which the keys were the vertices and each one had a list containing all the vertices it was connected too. Before inserting an edge, my method would check if it was previously in the graph. If not, then it would insert the edge.

```python
#as_EL() - returns an edge list representation of the graph
#inputs - None
#outputs - an edge list representation of the graph
def as_EL(self):
    g = graph.Graph(len(self.al))
    hash_map = {}
    for i in range(len(self.al)):
        hash_map[i] = []
        for j in range(len(self.al[i])):
            hash_map[i].append(self.al[i][j].dest)
            if not(self.al[i][j].dest in hash_map):
                g.insert_edge(i, self.al[i][j].dest)
            else:
                if not(i in hash_map[self.al[i][j].dest]):
                    g.insert_edge(i, self.al[i][j].dest)
    return g
```

Of course, all of this could have been avoided if I simply took care of this in the edge lists insert method, but I did not think of this at the time. The following method outputs the following.

```
edge list

[(0,1,4)(0,2,3)(2,3,1)(3,4,5)(4,1,4)]

adjacency list to edge list

[(0,1,4)(0,2,3)(2,3,1)(3,4,5)(4,1,4)]
```

Like previously, the top edge list is what we should get, and the bottom one is what we got, which in this case are thankfully identical. Now, I proceeded to implement these methods in the adjacency matrix. This is where the methods began holding some weight, since we would need to convert all the methods to an adjacency list in order to perform the draw method. First, I began working on returning an adjacency list implementation of the adjacency matrix. At first, I was simply traversing the entire matrix and adding the elements, but if the matrix was unweighted, then my algorithm was adding the same elements twice. Therefore, I recycled the same hash map idea as before to check if the edge was not in the list before adding it.

```
#as_AL() - returns an adjacency list implementation of the graph
#inputs - None
#outputs - an adjacency list implementation of the graph
def as_AL(self):
    g = graph.Graph(len(self.am), self.weighted, self.directed)
    hash_map = {}
    for i in range(len(self.am)):
        hash_map[i] = []
        for j in range(len(self.am[i])):
            if self.am[i][j]!= -1:
                if not(j in hash_map):
                    hash_map[i].append((j))
                    g.insert_edge(i, j, self.am[i][j])
                else:
                    if not(i in hash_map[j]):
                        hash_map[i].append(j)
                        g.insert_edge(i, j, self.am[i][j])
    return g
```

The method outputs the following.

```
adjacency list

[[(1,4)(2,3)] [] [(3,1)] [(4,5)] [(1,4)] [] ]

adjacency matrix to adjacency list

[[(1,4)(2,3)] [] [(3,1)] [(4,5)] [(1,4)] [] ]
```

My method to return an edge list of the adjacency matrix is an exact copy paste of the previous method but instead we initialize an edge list graph. Thankfully, it worked, so I did not even bother to check if we were doing extra work. The method outputs the following.

```
edge list

[(0,1,4)(0,2,3)(2,3,1)(3,4,5)(4,1,4)]

adjacency matrix to edge list

[(0,1,4)(0,2,3)(2,3,1)(3,4,5)(4,1,4)]
```

Finally, I began working on the methods to turn the edge list into the other implementations. First, the edge list to adjacency list method. For this method, I simply traversed the edge list and added the necessary values to the list, letting the edge list insert method do all the heavy lifting here.

```
#as_AL() - returns an adjacency list representation of the graph
#inputs - None
#outputs - an adjacency list representation of the graph
def as_AL(self):
    g = graph.Graph(self.vertices, self.weighted, self.directed)
    for i in range(len(self.el)):
        g.insert_edge(self.el[i][0], self.el[i][1], self.el[i][2])
    return g
```

The method would output the following.

```
adjacency list

[[(1,4)(2,3)] [] [(3,1)] [(4,5)] [(1,4)] [] ]

edge list to adjacency list

[[(1,4)(2,3)] [] [(3,1)] [(4,5)] [(1,4)] [] ]
```

The method to return an adjacency matrix implementation is the exact same. The edge list is quite simple without any caveats, which made these methods very easy to implement. That method outputs the following.

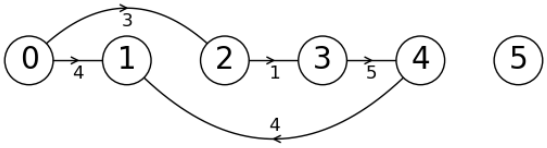
```
adjacency matrix

[[(-1)(4)(3)(-1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(-1)(-1)]
[(-1)(-1)(-1)(1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(5)(-1)]
[(-1)(4)(-1)(-1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(-1)(-1)]
]

edge list to adjacency matrix

[[(-1)(4)(3)(-1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(-1)(-1)]
[(-1)(-1)(-1)(1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(5)(-1)]
[(-1)(4)(-1)(-1)(-1)(-1)]
[(-1)(-1)(-1)(-1)(-1)(-1)]
]
```

Finally, the last task of part one was to implement the draw functions in all the classes. This simply consisted of getting an adjacency list implementation of the current list and then calling that class' draw function.

```
#draw() - draws the graph
#inputs - None
#outputs - None
def draw(self):
    al = self.as_AL()
    al.draw()
```

```
#draw() - draws the graph
#inputs - None
#outputs - None
def draw(self):
    al = self.as_AL()
    al.draw()
```

Both outputted the following.



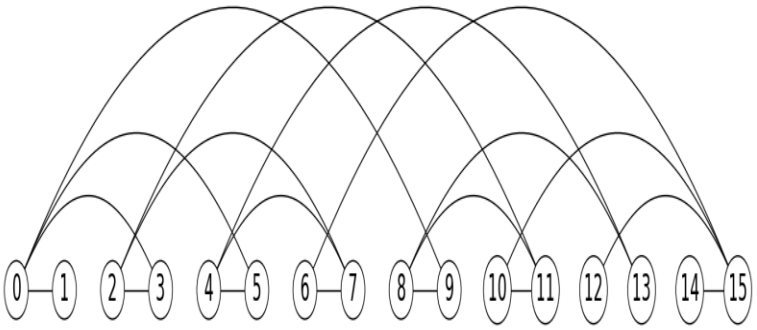
Part 2:

The first part to this riddle was to create a graph with all the possible moves. I am not sure if my approach is incorrect, but I basically manually looked at where we could go from each vertex and added it. I based my graph on a table that I did manually, which I built by saying what we could do at each move. Basically, the person could either go by themselves to the other side or take one of the items in the same side with him. The table looked like this.

Vertex	Vertices it can visit
0000	0001, 1001, 0101, 0011
0001	0000
0010	0011, 1011, 0111
0011	0010, 0000
0100	0101, 1101, 0111
0101	0100, 0000
0110	0111, 1111
0111	0110, 0010, 0100
1000	1001, 1101, 1011
1001	1000, 0000
1010	1011, 1111
1011	1010, 0010, 1000
1100	1101, 1111
1110	1111
1111	1110, 0110, 1010, 1100

Then, I built the graph based on this table by manually inserting every edge. The process to do this as well as how the graph looked is shown below. Notice that I used the integer representation of the states because the insert_edge method takes in integers, and I felt it would be too much of a hassle to write something to go around this condition. Once we had this graph, we could use the different searches to find a path to solve the riddle.

```
g = graph.Graph(16)
g.insert_edge(0, 1)
g.insert_edge(0, 9)
g.insert_edge(0, 5)
g.insert_edge(0, 3)
g.insert_edge(2, 3)
g.insert_edge(2, 11)
g.insert_edge(2, 7)
g.insert_edge(4, 5)
g.insert_edge(4, 13)
g.insert_edge(4, 7)
g.insert_edge(6, 7)
g.insert_edge(6, 15)
g.insert_edge(8, 9)
g.insert_edge(8, 13)
g.insert_edge(8, 11)
g.insert_edge(10, 11)
g.insert_edge(10, 15)
g.insert_edge(12, 15)
g.insert_edge(14, 15)
```



I only implemented the algorithm once with breadth first search and once with depth first search both in the adjacency list and then simply converted the other two graphs to adjacency lists and called these methods. The first thing I did was create a dictionary with keys ranging from 0 – 15 with the values of each key being a string representation of the number. I did this so that it would be easier to write the conditions that would check whether a movement was valid.

```
74    #chicken_bfs() - solves the chicken riddle with bfs
75    #inputs - start_vetrex : where we want bfs to start, should be
76    #                        just 0 but i didnt know how to initialize it
77    #outputs - the path to solve the chicken riddle
78    def chicken_bfs(self, start_vertex):
79        hash_map = {
80                0 : '0000',
81                1 : '0001',
82                2 : '0010',
83                3 : '0011',
84                4 : '0100',
85                5 : '0101',
86                6 : '0110',
87                7 : '0111',
88                8 : '1000',
89                9 : '1001',
90                10: '1010',
91                11: '1011',
92                12: '1100',
93                13: '1101',
94                14: '1110',
95                15: '1111',
                  }
```

Then, I created the frontier_q, discovered_set, and path. The path was initialized to a length as the same as the list and each slot was given a value of -1. Then, I appended the start vertex to each of these, and in the path, I put its same location. Then, I proceeded with regular breadth first search, but before visiting a vertex, I checked if it was a valid transition. This is where the hash table came handy. I used the binary representation of the number to check if either the fox could eat the chicken, or the chicken could eat the grain and the human was not there. In binary words, if the first two digits were ones and the last one was a zero, the next two were a one and the last one a zero, and vice versa. If the vertex was valid then we would travel to that vertex. Then at the end, I traversed the path array and made a new list containing the vertices we checked in order. Although it returned them as integers, we can look at their binary representations to see how they fit in to the riddle.

```
        frontier_q = []
        discovered_set = []
        path = [-1 for i in range(len(self.al))]
        frontier_q.append(start_vertex)
        discovered_set.append(start_vertex)
        path[start_vertex] = start_vertex
        while len(frontier_q) != 0:
            current_v = frontier_q.pop(0)
            for i in range(len(self.al[current_v])):
                if not self.al[current_v][i].dest in discovered_set:
                    if hash_map[self.al[current_v][i].dest][-1] == '0':
                        if not(hash_map[self.al[current_v][i].dest][0] == '1' and hash_map[self.al[current_v][i].dest][1] == '1'):
                            if not(hash_map[self.al[current_v][i].dest][1] == '1' and hash_map[self.al[current_v][i].dest][2] == '1'):
                                frontier_q.append(self.al[current_v][i].dest)
                                discovered_set.append(self.al[current_v][i].dest)
                                path[self.al[current_v][i].dest] = current_v
                    if hash_map[self.al[current_v][i].dest][-1] == '1':
                        if not(hash_map[self.al[current_v][i].dest][0] == '0' and hash_map[self.al[current_v][i].dest][1] == '0'):
                            if not(hash_map[self.al[current_v][i].dest][1] == '0' and hash_map[self.al[current_v][i].dest][2] == '0'):
                                frontier_q.append(self.al[current_v][i].dest)
                                discovered_set.append(self.al[current_v][i].dest)
                                path[self.al[current_v][i].dest] = current_v
        returning_path = []
        returning_path.append(len(path) - 1)
        search_val = path[-1]
        while search_val != 0:
            returning_path.insert(0, search_val)
            search_val = path[search_val]
        returning_path.insert(0, search_val)
        return returning_path
```

I essentially did the same thing with the depth first search but using the depth first search algorithm.

```
    #chicken_dfs() - solves the chicken riddle using dfs
    #inputs - start_vetrex : where we want bfs to start, should be
    #                        just 0 but i didnt know how to initialize it
    #outputs - the path to solve the chicken riddle
    def chicken_dfs(self, start_vertex):
        hash_map = {
                    0 : '0000',
                    1 : '0001',
                    2 : '0010',
                    3 : '0011',
                    4 : '0100',
                    5 : '0101',
                    6 : '0110',
                    7 : '0111',
                    8 : '1000',
                    9 : '1001',
                    10: '1010',
                    11: '1011',
                    12: '1100',
                    13: '1101',
                    14: '1110',
                    15: '1111',
                    }
```

```
151         stack = []
152         visited_set = []
153         stack.append(start_vertex)
154         path = [-1 for i in range(len(self.al))]
155         path[start_vertex] = start_vertex
156         while len(stack) != 0:
157             current_v = stack.pop()
158             if not current_v in visited_set:
159                 for i in range(len(self.al[current_v])):
160                     if hash_map[self.al[current_v][i].dest][-1] == '0':
161                         if not(hash_map[self.al[current_v][i].dest][0] == '1' and hash_map[self.al[current_v][i].dest][1] == '1'):
162                             if not(hash_map[self.al[current_v][i].dest][1] == '1' and hash_map[self.al[current_v][i].dest][2] == '1'):
163                                 visited_set.append(current_v)
164                                 stack.append(self.al[current_v][i].dest)
165                                 if not self.al[current_v][i].dest in visited_set:
166                                     path[self.al[current_v][i].dest] = current_v
167                     if hash_map[self.al[current_v][i].dest][-1] == '1':
168                         if not(hash_map[self.al[current_v][i].dest][0] == '0' and hash_map[self.al[current_v][i].dest][1] == '0'):
169                             if not(hash_map[self.al[current_v][i].dest][1] == '0' and hash_map[self.al[current_v][i].dest][2] == '0'):
170                                 visited_set.append(current_v)
171                                 stack.append(self.al[current_v][i].dest)
172                                 if not self.al[current_v][i].dest in visited_set:
173                                     path[self.al[current_v][i].dest] = current_v
174         returning_path = []
175         returning_path.append(len(path) - 1)
176         search_val = path[-1]
177         while search_val != 0:
178             returning_path.insert(0, search_val)
179             search_val = path[search_val]
180         returning_path.insert(0, search_val)
181         return returning_path
```

Then to implement these methods on the other graphs, I got a bit resourceful and simply created an adjacency list representation of the graph and then called these methods on that representation, like the below screenshot demonstrates.

```
82      #chicken_bfs() - solves the chicken riddle with bfs
83      #inputs - start_vetrex : where we want bfs to start, should be
84      #                        just 0 but i didnt know how to initialize it
85      #outputs - the path to solve the chicken riddle
86      def chicken_bfs(self, start_vertex):
87          new_graph = self.as_AL()
88          return new_graph.chicken_bfs(start_vertex)
89
90      #chicken_dfs() - solves the chicken riddle using dfs
91      #inputs - start_vetrex : where we want bfs to start, should be
92      #                        just 0 but i didnt know how to initialize it
93      #outputs - the path to solve the chicken riddle
94      def chicken_dfs(self, start_vertex):
95          new_graph = self.as_AL()
96          return new_graph.chicken_dfs(start_vertex)
```

Now the results that these algorithms give us are a bit strange. It took me a while to figure it out, but essentially there are two solutions to the riddle. The breadth first search returns one and the depth first search another one, given to the nature of how they visit the vertices. They both travel to both paths, but give priority to one, which is the one that is displayed. The breadth first search displays the following solution.

```
The path for the riddle using bfs is
[0, 5, 4, 13, 8, 11, 10, 15]
```

If we look at the binary representations of this, we can see that this means that first, we take the chicken to the right side. Then, we go back to the left and bring the fox to the right side. After this, we take the chicken back to the left side. Then, we grab the grain and bring it to the right side. The last step is simply to go to the left and bring the chicken over. The depth first search gives a different solution, but it essentially differs at only one step.

```
The path for the riddle using dfs is

[0, 5, 4, 7, 2, 11, 10, 15]
```

In this solution, we first take the chicken over to the right side. Then, we go back to the left side and bring over the grain to the right. Then, we take the chicken to the left side and grab the fox. Then, we take the fox to the right side. Finally, we go back to the left and bring the chicken back. It is almost the same as the previous solutions. If we look at both paths visited vertices, they both find both of these solutions, but give priority to either depending on which vertices were visited first.

Conclusion

This lab was very fun to me because once you fundamentally understood how graphs and their different implementations worked, it was fun to proceed with the rest of the methods. It was interesting seeing how we could jump between different implementations because they all essentially gave off the same date, we just had to find how to extract and manipulate it. It was also very fun to see how we could represent a mathematical riddle like this one using graphs and then solving it using different search methods.

Appendix

```
1 #CS 2302 Data Structures Fall 2019 MW 10:30
2 #Sergio Ortiz
3 #Assignment - Lab #6
4 #Instructor - Olac Fuentes
5 #Teaching Assistant - Anindita Nath
6 #November 15, 2019
7 #This program will convert between various graphs
8 #implementations
9 #Next, it will solve the riddle:
10 #You have a fox, a chicken and a sack of grain. You must cross a river with only one of them at a time
11 #If you leave the fox with the chicken he will eat it;
12 #if you leave the chicken with the grain he will eat it.
13 #How can you get all three across safely?
14
15 import graph_AL as graph
16 import graph_AM_Sergio_Ortiz as graph1
17 import graph_EL_Sergio_Ortiz as graph2
18
19 #adjacency_list() - builds an adjacency list and converts it
20 #                   to all the other graph types
21 #inputs - None
22 #outputs - None
23 def adjacency_list():
```

```
    print('adjacency list:')
    print()
    al = show_AL()
    print()
    print('adjacency matrix')
    print()
    show_AM()
    print()
    print('adjacency list to adjacency matrix')
    print()
    al_am = al.as_AM()
    al_am.display()
    print()
    print('edge list')
    print()
    show_EL()
    print()
    print('adjacency list to edge list')
    print()
    al_el = al.as_EL()
    al_el.display()
    print()

#adjacency_matrix() - builds an adjacency matrix and converts it
#                     to all the other graph types
#inputs - None
#outputs - None
def adjacency_matrix():
    print('adjacency matrix:')
    print()
    am = show_AM()
    print()
    print('adjacency list')
    print()
    show_AL()
    print()
    print('adjacency matrix to adjacency list')
    print()
    am_al = am.as_AL()
    am_al.display()
    print()
    print('edge list')
    print()
    show_EL()
    print()
    print('adjacency matrix to edge list')
    print()
    am_el = am.as_EL()
    am_el.display()
    print()

#chicken_breadth() - solves the chicken riddle using bfs
#inputs - g: the graph we will search
#outputs: None
def chicken_breadth(g):
    chicken_path = g.chicken_bfs(0)
    print('The path for the riddle using bfs is ')
    print(chicken_path)

#chicken_build_AL() - builds an adjacency list for the chicken riddle
#inputs: None
#ouputs: A graph containing all the possible moves in the chicken riddle
def chicken_build_AL():
    g = graph.Graph(16)
    g.insert_edge(0, 1)
    g.insert_edge(0, 9)
    g.insert_edge(0, 5)
    g.insert_edge(0, 3)
    g.insert_edge(2, 3)
    g.insert_edge(2, 11)
    g.insert_edge(2, 7)
    g.insert_edge(4, 5)
```

```
    g.insert_edge(4, 13)
    g.insert_edge(4, 7)
    g.insert_edge(6, 7)
    g.insert_edge(6, 15)
    g.insert_edge(8, 9)
    g.insert_edge(8, 13)
    g.insert_edge(8, 11)
    g.insert_edge(10, 11)
    g.insert_edge(10, 15)
    g.insert_edge(12, 15)
    g.insert_edge(14, 15)
    g.draw()
    return g

#chicken_build_AM() - builds an adjacency matrix for the chicken riddle
#inputs: None
#ouputs: A graph containing all the possible moves in the chicken riddle
def chicken_build_AM():
    g = graph1.Graph(16)
    g.insert_edge(0, 1)
    g.insert_edge(0, 9)
    g.insert_edge(0, 5)
    g.insert_edge(0, 3)
    g.insert_edge(2, 3)
    g.insert_edge(2, 11)
    g.insert_edge(2, 7)
    g.insert_edge(4, 5)
    g.insert_edge(4, 13)
    g.insert_edge(4, 7)
    g.insert_edge(6, 7)
    g.insert_edge(6, 15)
    g.insert_edge(8, 9)
    g.insert_edge(8, 13)
    g.insert_edge(8, 11)
    g.insert_edge(10, 11)
    g.insert_edge(10, 15)
    g.insert_edge(12, 15)
    g.insert_edge(14, 15)
    g.draw()
    return g

#chicken_build_EL() - builds an edge list for the chicken riddle
#inputs: None
#ouputs: A graph containing all the possible moves in the chicken riddle
def chicken_build_EL():
    g = graph2.Graph(16)
    g.insert_edge(0, 1)
    g.insert_edge(0, 9)
    g.insert_edge(0, 5)
    g.insert_edge(0, 3)
    g.insert_edge(2, 3)
    g.insert_edge(2, 11)
    g.insert_edge(2, 7)
    g.insert_edge(4, 5)
    g.insert_edge(4, 13)
    g.insert_edge(4, 7)
    g.insert_edge(6, 7)
    g.insert_edge(6, 15)
    g.insert_edge(8, 9)
    g.insert_edge(8, 13)
    g.insert_edge(8, 11)
    g.insert_edge(10, 11)
    g.insert_edge(10, 15)
    g.insert_edge(12, 15)
    g.insert_edge(14, 15)
    g.draw()
    return g

#chicken_depth() - solves the chicken riddle using dfs
#inputs - g: the graph we will search
#outputs: None
def chicken_depth(g):
```

```
    chicken_path = g.chicken_dfs(0)
    print()
    print('The path for the riddle using dfs is ')
    print()
    print(chicken_path)

#edge_list() - builds an edge list and converts it
#                   to all the other graph types
#inputs - None
#outputs - None
def edge_list():
    print('edge list:')
    print()
    el = show_EL()
    print()
    print('adjacency list')
    print()
    show_AL()
    print()
    print('edge list to adjacency list')
    print()
    el_al = el.as_AL()
    el_al.display()
    print()
    print('adjacency matrix')
    print()
    show_AM()
    print()
    print('edge list to adjacency matrix')
    print()
    el_am = el.as_AM()
    el_am.display()
    print()

#main() - main method where everything happens
#inputs - None
#outputs - None
def main():
    print('(1) Adjacency List')
    print('(2) Adjacency Matrix')
    print('(3) Edge List')
    which_graph = int(input('Which graph would you like to see? '))
    print()
    while which_graph < 1 or which_graph > 3:
        which_graph = int(input('Enter a valid graph '))
    if which_graph == 1:
        adjacency_list()
    elif which_graph == 2:
        adjacency_matrix()
    else:
        edge_list()

    print('Now for the chicken riddle')
    print('(1) Adjacency List')
    print('(2) Adjacency Matrix')
    print('(3) Edge List')
    which_graph = int(input('Which graph would you like to use? '))
    print()
    while which_graph < 1 or which_graph > 3:
        which_graph = int(input('Enter a valid graph '))
    print()
    print('(1) Breadth First Search')
    print('(2) Depth First Search')
    which_search = int(input('Which search would you like to use? '))
    while which_search < 1 or which_search > 2:
        which_search = int(input('Enter a valid search '))
    if which_graph == 1:
        g = chicken_build_AL()
    elif which_graph == 2:
        g = chicken_build_AM()
    else:
        g = chicken_build_EL()
```

```
240
241     if which_search == 1:
242         chicken_breadth(g)
243     else:
244         chicken_depth(g)
245
246 #show_AL() - shows the adjacency list by displaying and drawing it
247 #inputs - None
248 #outputs - g: an adjacency list
249 def show_AL():
250     g = graph.Graph(6,weighted=True,directed = True)
251     g.insert_edge(0,1,4)
252     g.insert_edge(0,2,3)
253     g.insert_edge(1,2,2)
254     g.insert_edge(2,3,1)
255     g.insert_edge(3,4,5)
256     g.insert_edge(4,1,4)
257     g.delete_edge(1,2)
258     g.display()
259     g.draw()
260     return g
261
262 #show_AM() - shows the adjacency matrix by displaying and drawing it
263 #inputs - None
264 #outputs - g: an adjacency matrix
265 def show_AM():
266     g = graph1.Graph(6,weighted=True,directed = True)
267     g.insert_edge(0,1,4)
268     g.insert_edge(0,2,3)
269     g.insert_edge(1,2,2)
270     g.insert_edge(2,3,1)
271     g.insert_edge(3,4,5)
272     g.insert_edge(4,1,4)
273     g.delete_edge(1,2)
274     g.display()
275     g.draw()
276     return g
277
278 #show_El() - shows the edge list by displaying and drawing it
279 #inputs - None
280 #outputs - g: an edge list
281 def show_EL():
282     g = graph2.Graph(6,weighted=True,directed = True)
283     g.insert_edge(0,1,4)
284     g.insert_edge(0,2,3)
285     g.insert_edge(1,2,2)
286     g.insert_edge(2,3,1)
287     g.insert_edge(3,4,5)
288     g.insert_edge(4,1,4)
289     g.delete_edge(1,2)
290     g.display()
291     g.draw()
292     return g
293
294
295 if __name__ == "__main__":
296     main()
297
```

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.